Frontiers in Education, 32nd ASEE/IEEE Frontiers in Education Conference, Vol. 1, 2002, pp. T1B-7 - T1B-12.

AUTOMATED EVALUATION METHODS WITH ATTENTION TO INDIVIDUAL DIFFERENCES- A STUDY OF A COMPUTER-BASED COURSE IN C

Tammy Rosenthal[1], *Patrick Suppes*[2], *Nava Ben-Zvi*[3]

Abstract *–– This study focuses on students' performance in a distance-learning computer-based C course offered by EPGY (Education Program for Gifted Youth) at Stanford University. Its main goal is testing the feasibility of automated evaluation. Three automated methods consisting of a criterion set and weighted scoring methods were developed empirically, then applied to four hundred programs of ten types of problems and thoroughly analyzed. For most of the assignments, the more significant criteria for evaluation were related to execution time features, (i.e. number of operations and calls to functions executed by the program), memory storage required to store the variables of the program, and code's length. The results showed that the criterion measures often differed by several orders of magnitude between students' programs and were dependent on the specific problem domain. A low correlation was found between the evaluation criteria indicating that the criteria are nearly independent. Consequently, in qualitative tests for achievement and ability students' performances exhibited high dimensionality.*

Index Terms ––Program evaluation, students' individual differences, students' performances, automated criteria.

BACKGROUND

The Education Program for Gifted Youth (EPGY) at Stanford University is an ongoing project dedicated to developing and offering multimedia computer-based distance-learning courses. The courses offered are in mathematics, physics, computer science and other subjects and target pre-college students of high ability. Currently, about 3000 students annually are taking EPGY courses of which several hundred are students of CS courses. The EPGY introductory CS course in C (C11A) is the focus of this study [9]. The main goal of C11A is to provide students with a general introduction to CS and programming within the context of learning the C programming language. The course as a whole contains 55 short lectures of 5-10 minutes each, 250 quizzes given by the software at the end of the lectures with automated immediate feedback to the student, and 29 programming assignments that the students are required to write, run and then send to their tutor by e-mail as attached files. Each multimedia lecture typically consists of 40-80 frames with synchronized sound (audio) that runs as a whole movie. Each student receives 3 CDs: one system CD which contains the EPGY software, and two lecture CDs, as well as a package of written materials. The student needs first to install the system CD. Then the software will instruct him what to do and how to progress. For example, the software will tell the student which lecture CD to insert. It will also specify the title of the next lecture. The software will actually play the lecture itself and once the lecture ends it will present automated exercises and quizzes to the student on the lecture topics. For each answer the student enters, the software will display an automated message indicating if the answer was correct along with a detailed explanation about the correct answer and error type if one occurred. Figure 1 presents a sample of an exercise in which students are required to locate and correct syntax of a given code editing it line by line.

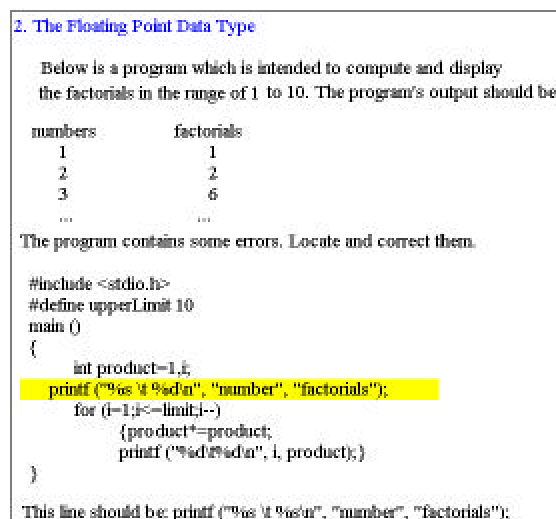

FIGURE. 1
A S SCREEN SHOT FROM A N AUTOMATED EXERCISE

All the answers entered by students on the quizzes are stored by the software in a special file. Once a week, the software will remind the student to send his "report" (done by choosing the "report" option from the software menu). The report contains all his answers to the quizzes. Once the student sends his report it will be received and processed by EPGY and then transferred to the appropriate tutor for further follow up. In fact, students get two types of feedback

[1] Tammy Rosenthal, Stanford University, Education Program for Gifted Youth, Ventura Hall, Stanford, CA 94305-4115 tammy@epgy.stanford.edu
[2] Patrick Suppes, Stanford University, Education Program for Gifted Youth, Ventura Hall, Stanford, CA 94305-4115 suppes@ockham.stanford.edu
[3] Nava Ben-Zvi, Hebrew University; Hadassah College Jerusalem, 37 Haneviim St., Jerusalem, Israel, 91010 President@hadassah-col.ac.il

on their answers – an automated online feedback from the software and a manual feedback from the tutor (by e-mail). The software registers the location students have quit and thus next time they wish to work it will continue from this current location. For programming assignments, the software will refer the student to the specifications in the assignment sheet, which was sent to the student by mail within the written materials package. The students work separately on their programs and send them as attached files for manual evaluation by the tutors. The main motivation for this study grew as a result of conclusions drawn from running the first sessions of C11A. Our main observations were:

- The process of manual evaluations is expensive. For example, if 50 students are active within each, then each one sends 29 programs. Thus, each month the tutor needs to evaluate and comment on 500 programs *manually*, which might require 80-120 hours. This problem of "cost" in evaluating students' achievements in programming courses has emerged in instructing other programming courses. For example, in the Stanford CS introductory classes [Roberts, Lilly, Rollins 1995], the human resources allocated for evaluating and grading the programming assignments were very large. A total of approximately 100 section leaders for 1000 students on a ratio of 1 to 10 are allocated, as well as extra manpower to coordinate, instruct and manage them. The costs incurred are enormous.
- It turns out that some types of errors are very common. This leads to almost-standard types of comments. For example, in an assignment, called *mulByAdd,* students were required to multiply two numbers by using the addition operations only. One common mistake was that some students did not check which is the smaller number. Thus, for two input numbers with large difference (2 and 10,000) they performed 10,000 iterations (to add 2 to itself 10,000 times) instead of 2 iterations (to add 10,000 twice). Accordingly, a common evaluation message explained that for efficiency considerations it is required to select the smaller number as the number of iterations.
- Much of the evaluator's work may be automated. For example, lack of I/O prompts within the program telling the user what is expected of him can be measured, since they all include the 'printf' function call within the prompt statement. More crucial: the total number of high-level operations can be measured. By evaluating these features an automated tutor can tell the student whether he could reduce the number of operations within the program to make the code more efficient and concise, or reduce the memory storage required to store the program's variables.
- Several criteria have already been identified in the CS literature as important for evaluating programs. For example, the *execution time* [5] and the amount of memory (or storage) required for the program are two major concerns to analyze performance of programs [3]. Storage can be measured by using the *sizeof* operator on the variables declared within the programs. Since dynamic memory and pointers are not included in C11A, the *sizeof* results will yield all the required storage information. Also, *execution time* can be evaluated either by measuring the exact *run time* of the program, performing Big *O* estimates, or by using the criterion we have defined that counts the *exact* number of operations executed within the program for a certain input. Figure 2 presents the *execution time* computed in three ways for the selection sort algorithm of an array of data with size N.

Array size (*N*)	Running Times		Big–*O* $O(N^2)$	Exact number of operations
10	0.12	Msec	*O* (100)	170
20	0.39	Msec	*O* (400)	435
40	0.46	Msec	*O* (1600)	1265
100	8.72	Msec	*O* (10,000)	6155
200	33.33	Msec	*O* (40,000)	22,305
400	135.42	Msec	*O* (160,000)	84,605
1000	841.67	Msec	*O* (1,000,000)	511,505
2000	3.35	Sec	*O* (4,000,000)	2,023,005
4000	13.42	Sec	*O* (16,000,000)	8,046,005
10,000	83.90	Sec	*O* (100,000,000)	50,115,005

FIGURE. 2
RESULTS FOR EXECUTION TIME COMPUTED IN THREE WAYS FOR AN ARRAY OF DATA WITH SIZE *N*

These reasons motivated us to conduct this empirical study and to examine whether a more cost-effective model for evaluation can be established. Thus, the main goal of this study is to estimate to what extent the evaluation of computer programs constructed by students in the context of a course can be automated. For this purpose we developed new automated methods for programs' evaluation, analyzed the results of applying them first to a smaller pilot of programs (150), refined them, and then analyzed the results on a larger set of programs (400). Eventually, the goal was to characterize a new model for program evaluation, automated partially, but also with attention to the limitations of the model and to sustaining individual differences of students in their programming performance.

Evaluating the automated methods included analysis of: Students' feedback to receiving automated evaluation comments on their programs; and critical comments given by CS experts with regard to the automated criteria and methods. The focus of this study is on evaluation features such as efficiency, clarity, and modularity. Verification is not included in this scope. But, since it's an important phase in program evaluation a practical method is suggested as part of the newly suggested model for evaluation. The student selection of this study consisted of a total of 60 students: 40 students who took the first five sessions of the EPGY introductory C course (C11A) and who completed the course successfully within the period of July 98-99. This total of 40 consists of 26 individual remote gifted students and

14 from a private pilot school. The 26 individual students took the course in their own homes and contacted their instructor via e-mail. The 14 students from the private pilot school were in the 6th and 7th grades and had an instructor in the classroom lab while taking the course, so they could ask her questions and get immediate help with debugging questions or problems they faced. The analysis of applying automated methods on students' programs and of individual differences in learning C were done with respect to this group. The second pilot group consisted of 20 students who took the course in the period: October 2000 - October 2001. These 20 students contributed to the evaluation phase of the automated methods and model. Their feedback to the automated-message simulation was analyzed.

AUTOMATED METHODS FOR EVALUATION

The newly developed methods include: a criterion set established to evaluate students programs; a formula giving different weights to the criteria and computing scores for each program; a scoring method to evaluate student's performance in the entire set of assignments examined.

The criterion set consists of ten measurable features for program performance. The criteria were designed to follow the accumulated experience in instructing and evaluating students' programs manually, common errors students made, as well as the CS literature on efficiency of algorithms. Figure 3 presents the criterion set. In this figure, for example, the criterion named *operations* counts the number of operations executed by a program for a given input. This criterion corresponds to the *execution time* measure and correlates to the measures of *Big-O* and *run time* presented in Figure 2. The *operations* criterion was developed to suit the needs of C11A of smaller programs for which an exact count of operations gives more accurate information.

Operations - Number of operations as executed for a certain input.
functions - Number of calls to functions as *executed* for certain input.
mem – Number of bytes required for storing the variables of the program.
relative - The difference between the number of lines of the longest and the shortest functions.
len - Code's length in lines.
com – Total number of comments included in the program.
consts - Number of constants declared in the program.
io - Number of prompts (I/O).
significant - Significant names given to variables and functions in scale of 1-5. 1-best, 5-worst.

FIGURE. 3
CRITERIA SET FOR PROGRAM EVALUATION

The results from applying the criteria to students' programs showed that the significance of the criteria depended to a great extent on the specific problem domain. For example, in an early assignment named *min3*, the number of comparisons executed for a given input was more significant to evaluate performance. In a more advanced assignment, such as *prime100*, the set of criteria to evaluate the performance was larger: *operations, functions*, *mem*, *length, relative, user* and *com.* In general, for the more complex assignments, the set of the significant criteria for evaluation was larger. Also, in more complex assignments, more significant differences occurred between the criteria values. The more significant criteria for the evaluation for most of the assignments were:
operations – counts the total number of operations within the program as executed for a given input; *functions* - counts the number of calls to functions as executed for a given input; *mem* – measures the amount of memory storage required to store the variables of the program; and *length* - measures the length in code excluding comments or blank lines. The range of the minimum and maximum values measured for these criteria was often several orders of magnitude. For example, in the *prime100* assignment *mem* ranged between 2 and 218 bytes. In the *mulbyadd* assignment *operations* ranged between 12 and 20032 operations. In the *perfect* assignment *functions* ranged between 22 and 1764 calls to functions. Figure 4 presents the average number of operations executed by 13 students over 10 assignments.

In this Figure the range of the average count of operations in students' programs is between 229 and 3431 operations. The large differences in the criterion measures resulted from diversity in choice of algorithms, as well as in student choice of implementation methods such as using the sieve of *Eratosthenes* algorithm to compute the prime numbers in a given range, as opposed to a simple nested *for* loop that checks divisors of each number in the range. Figure 5 presents two codes with different number of calls to functions: one executing 99 calls and a second 525 calls. The difference is related to the location in the program where the call to the *sqrt* function is made either outside or inside of the inner loop.

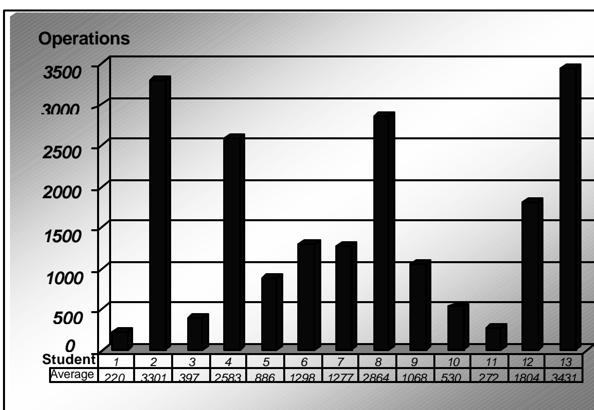

FIGURE. 4
AVERAGE NUMBER OF OPERATIONS FOR 10 ASSIGNMENTS

```
//Implementation A: the call to sqrt is done outside
// of the inner loop; altogether  there are 99 calls to sqrt.

#include <stdio.h>
#include <math.h>
main()
{  int i,j,rooti,prime;
  for (i=2 ; i <= 100 ; i++)
     { rooti = sqrt(i);
              prime = 0;
              for (j = 2 ; j <= rooti ; j++)
              { if (i % j == 0) prime = 1; }
              if (prime != 1) printf("%d\n",i);    }
}
_______________________________________________
//Implementation B: the call to sqrt is done in the control line
//of the inner loop. A total of 525 calls to sqrt are done.

#include <stdio.h>
#include <math.h>
main()
{   int i,j,prime;
  for (i=2 ; i <= 100 ; i++)
   {  prime = 0;
     for (j = 2 ; j <= sqrt(i) ; j++)
     {          if (i % j == 0) prime = 1;          }
         if (prime != 1) printf("%d\n",i);      } }
```

FIGURE. 5
DIFFERENCE IN CALLS TO FUNCTIONS

Figure 6 presents the average number of calls to functions executed by 13 students over 10 assignments examined. The differences resulted from choice of algorithms, implementation methods, or specific locations in the programs where the calls are made [see Figure 5].

Significant differences were measured in students' performance –individually, as well as between the two groups (home and school students). On the average, the performance measures for most of the criteria were better for the home than for the school students. For some of the more basic assignments and for the home students, a significant positive correlation (0.7) occurred between the criteria of: *length* and *operations*. Namely, students who tended to write shorter programs used less operations, and vice versa. But, for most of the assignments, and in particular for the more advanced ones, no significant correlation was found. As a whole, only 13% of the tests made resulted in significant correlation results. This may indicate that the criteria are relatively independent. None of the criteria is redundant, since each contributes different information to the evaluation process. The scoring methods developed are based on the criteria and computed scores by assigning larger weights to the more significant criteria. Significant differences were found between students' scores and between the two groups (home and school students). On the average, the home students had better scores both in specific assignments and in their total performance. The automated method and the manual one (used previously by the instructor for the same programs and students) were also compared. Thus, when the students were grouped by dividing their grade levels into four ranges, good matches were found. For 86% of the total pilot, students who belonged to the *n*th manual grade level belonged to the corresponding *n*th automatic score range.

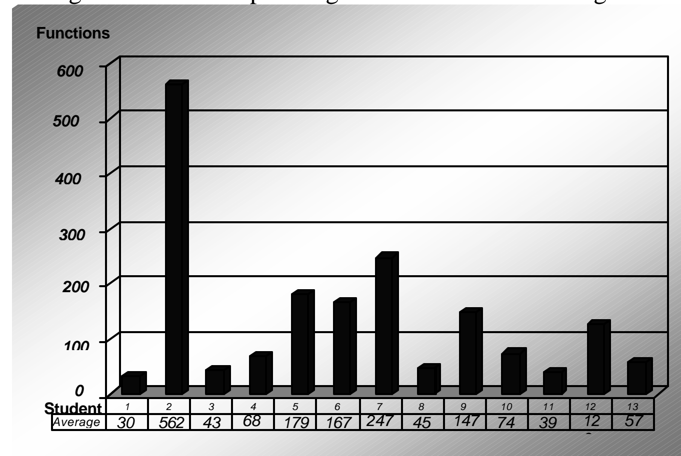

FIGURE. 6
AVERAGE NUMBER OF CALLS TO FUNCTIONS AS EXECUTED OVER 10 ASSIGNMENTS

A MODEL FOR PROGRAM EVALUATION

The suggested model for program evaluation consists of an automated part (of both verification and evaluation testing) and a manual one. Since the results showed that students' performance in programs depends on the problem domains, the suggested model is constructed in a problem-oriented way using different sets of criteria to evaluate the assignments. In this scheme, the *student* first sends a program to a special address given to him in advance, such as programs@abel.stanford.edu. A special system program, named for example, Program identifier, identifies which assignment was sent and accordingly sends it to the appropriate program to conduct verification and evaluation tests. At the next phase verification tests are conducted. The verification tests are done by running the program with different sets of predefined and random inputs from the legal input range as well as additional special test cases.

If they fail on any of the inputs, an appropriate automated message is sent to the student, with copies to the tutor and the database (for follow up). If the tests succeed, the program is sent to the next phase of automated evaluation. The evaluation tests are conducted according to the significant criteria as identified in advance for the particular assignment. Accordingly, automated messages are generated. The automated report is sent to the tutor. The tutor checks and adds manual comments if he wishes. The combined report (automated and manual that includes also the results of the verification tests) is sent to the student, with a copy to the database for follow-up. The automated evaluation focuses on several sets of measurable criteria of performance (such as exact count of operations or function

calls as executed for a given input). For each type of assignment the selection of the specific criteria is different and depends on the more significant criteria that capture the evaluation according to the results measured. For each assignment, a *best program* is selected from among programs that had better performance according to the most significant weighted criteria. Students' programs submitted are compared with the best solution to the particular problem domain. Accordingly, automated messages are generated. The automated messages focus on measurements of the criteria and on comparing their values to the optimal solution's performance. The manual part covers the features of evaluation that could not be automated (such as contents and context issues). Eventually, the evaluation process of this model will generate detailed comments to students on their program performance by suggesting ways on how they could improve. Figure 7 presents an example of the way this model works for the *mulByAdd* assignment.

```
#include <stdio.h>     // Inputs: 2 , 10000
#include "genlib.h"
#include "simpio.h"
main()
{  int num1, num2, num3, num4;
   printf ("Enter first number:");
   num1=GetInteger();
   printf ("Enter second number:");
   num2=GetInteger();
   num3=0;
   num4=0;
  while (num3!=num2)
 {  num4=num1+num4;
    num3=num3+1;   }
printf ("The result of %d*%d is %d", num1, num2, num4);}
```

FIGURE. 7(a)
STUDENT'S PROGRAM FOR THE *MULBYADD*

```
#include <stdio.h>     // Inputs: 2, 10000
#include "genlib.h"
#include "simpio.h"
main()
{   unsigned int  high,low,product;
    printf("Please enter two integers to multiply.\n");
    printf("First number: ");
    low = GetInteger();
    printf("Second number: ");
    high = GetInteger();
    if (high < low)
  {            product =low ;
                 low=high;
                 high=product;    }
  product=0;
  while (low --> 0)
  { product += high; }
printf("\nThe result of %d*%d is %d.",num1,num2,product);}
```

FIGURE. 7(b)
BEST SOLUTION SELECTED FOR THE *MULBYADD*

At the first stage verification tests are conducted. The verification tests are done by running the program with: a set of predefined inputs (given to the students too); a set of random inputs from the legal input range; and a set of special extreme test cases; Once the verification tests succeed for all the inputs tested, the program is sent to evaluation tests. The evaluation tests focus on comparing the significant criteria measured for the best program as opposed to student's program. The most significant criteria for this type of assignments are computed for the student's program and the values are compared to the best program's values. An automated evaluation comment that may be generated for this case is presented below. The evaluation program analyzes a student's work. The results of the analysis are then put in templates sentences used for the comment.

Dear Bruce,
Your program has passed successfully the verification tests . . . This could have been done more efficiently by using a total of 14 operations containing only 2 iterations instead of the 10,000 done by your program. This can be achieved by comparing the two input numbers, (i.e. 2 and 10,000) and setting the smaller (i.e. 2) to determine the number of iterations executed . . .

At the last phase, the automated report is sent to the tutor who can add now manual comments that were not captured by the automated system. A sample manual evaluation comment from the tutor for this case may include the following text.

The variables names you have chosen (*num3* and *num4)* are not significant enough; Since *num4* is used to accumulate the product of num1 and num2, a better name could be for example, *product* . . .

QUALITATIVE TESTS FOR ACHIEVEMENT AND ABILITY

In order to determine how much variation and dimensionality in performance are present among our pilot group, we have conducted qualitative tests for achievement and ability. We have included the most significant measures of programming performance as found in the study as well as other individual parameters of students referred to in the literature [4]. Nine parameters were looked at, including, the 4 found as the most significant for program evaluation [*operations, length*, *functions, mem*], the calendar time [the number of days it took each student to complete the course as a whole], students' score in the automated exercises, number of debugging questions students sent , the average time students spent on programs per chapter, and age of student in years, calculated at the time students began the course. The qualitative tests were done by constructing a partial ordering of student performance according to subsets of performance scales. The partial ordering used is *domination*. Student A dominates student B with respect to a set of measures, if all his scores are equal or better than the

ones of student B in all measures, and better on at least one. An *undominated* set of students consists of students who are not dominated by other students with respect to all the selected performance measures. Thus, we characterize the partial orderings by looking into the sizes of the undominated sets as well as the length of the dominance chains of students. For each subset, consisting of all the possible combinations from among our performance measures-set, we compute the sizes of the undominated sets of students with respect to the partial ordering, as well as the lengths of the dominance chains of students. If, given some or all of these measures, only a small group of students are undominated with respect to the partial ordering then we may consider the various performance measures to be highly dependent in character, possibly leading to the conclusion that a single performance measure might suitably capture the student ability. On the other hand, if the set of undominated students is large, then it may indicate that the measures are only weakly correlated , and to characterize student ability requires knowing several measures. Full results and analysis are presented in [7]. Some of our main findings are summarized below and presented in Table 1.

TABLE. 1(a)
MINIMUM UNDOMINATED SETS FOR 8 MEASURES

Number of measures	Minimum sizes of undominated sets	Minimum sizes of undominated sets	Minimum sizes of undominated sets
	40 students	26 gifted students	26 gifted students
	Average program measures	Average program measures	Program measures in *prime100*
2	1 (2.5%)	1 (3.8%)	3 (11%)
3	1	1	4
4	3	3	6
5	4	4	7
6	6	6	8
7	7	7	15
8	16 (40%)	15 (57%)	16 (61%)

TABLE. 1(b)
MAXIMUM UNDOMINATED SETS FOR 8 MEASURES

Number of measures	Maximum sizes of undominated sets	Maximum sizes of undominated sets	Maximum sizes of undominated sets
	40 students	26 gifted students	26 gifted students
	Average program measures	Average program measures	Program measures in *prime100*
2	11 (27%)	10 (38%)	10 (38%)
3	15	14	14
4	16	15	16
5	16	15	17
6	16	15	18
7	16	15	18
8	16 (40%)	15 (57%)	18 (69%)

In this table 8 measures are included: operations, length, functions, memory, completion time, scores in reports, debugging questions, and time spent on programs. We found significant differences in the diversity in performance between our two samples. For the entire pilot, the undomintaed set grew from 27% to 40% of the total sample size, as more measures were included. For the gifted students' sample, the growth was larger: 38% to 57%. For the average counts of the four criteria of program evaluation, we found that the three measures, *operations, length* and *mem* are more correlated. We found differences between the diversity in students' performance depending on whether average programming criteria (for several assignments) were selected, or whether measures in the context of a specific assignment were selected. For the criteria- counts in the context of a specific assignment named *prime100*, the undominated sets grew up to including six measures. We found that age was not a major factor affecting other performance measures.

REFERENCES

[1] Cope, E., Suppes, P. "Gifted Students' Individual Differences in Distance-Learning Computer-Based Physics", *Instructional Science*, 2002, forthcoming.

[2] Briand L. C., Morasca S., and Basilli V. R. "Defining and Validating Measures for Object-Based High-Level Design", *IEEE Transactions on Software Engineering*, Vol. 25, No. 5, pp 722- 743, 1999.

[3] Harel D. "Algorithmics - The Spirit of Computing Second Edition', *Addison-Wesley, England*, 1992.

[4] Malone, T.W., Suppes, P., Macken, E., Zanotti, M., and Kanerva, L. "Projecting Student Trajectories in a Computer-Assisted Instruction Curriculum". *Journal of Educational Psychology,* 1979, **71**, 74-84.

[5] Roberts, E. S. "Programming Abstractions in C, A second Course in Computer Science". *Addison- Wesley Publishing Company.* 1998. ISBN: 0-201-54541-1.

[6] Roberts, E. S., Lilly, J., and Rollins, B. "Using Undergraduates as Teaching Assistant in Introductory Programming Courses: On Update on the Stanford experience. Department of Computer Science Stanford University". *SIGCSE 195 3/95 Nashville*, 1995, TN USA.

[7] Rosenthal, T, Automated Evaluation Methods with Attention to Individual Differences - A Study of a Computer-based Course in C. A Ph.D. Thesis. *The Hebrew University*, Jerusalem, Israel (in preparation.)

[8] EPGY: http://www-epgy.stanford.edu

[9] C11A: http://www-epgy.stanford.edu Under "Course Catalog" , "Computer Science", "C11A".