Computer Algorithms for Plagiarism Detection

ALAN PARKER, MEMBER, IEEE, AND JAMES O. HAMBLEN, MEMBER, IEEE

Abstract—**This paper presents a survey of computer algorithms used for the detection of student plagiarism. A summary of several algorithms is provided. Common features of the different plagiarism detection algorithms are described. Ethical and administrative issues involving detected plagiarism are discussed.**

INTRODUCTION

THE term "plagiarize" is defined [1] as "to take (ideas, writings, etc.) from (another) and pass them off as one's own." A plagiarized program can defined as a program which has been produced from another program with a small number of routine transformations. Routine transformations, typically text substitutions, do not require a detailed understanding of the program. Unfortunately, student plagiarism of programming assignments has been made easier by large class sizes, the introduction of personal computers, computer networks, and easy-to-use screen editors. Plagiarism of computer programs can become quite common in large undergraduate classes. One or two instances of blatant plagiarism are detected in most large introductory undergraduate computer classes. A number of more subtle cases of plagiarism may go undetected.

Faidhi and Robinson [2] characterize six levels of program modification in a plagiarism spectrum as shown in Fig. 1. Examples C programs demonstrating level 0, 1, 3, and 5 plagiarism are shown in Fig. 2. For this example, a well-known Sieve of Eratosthenes benchmark program is used. Level 0 is the original program without modifications. In level 1, only comments and indentation are changed. Level 3 changes the identifier names and positions of variables, constants, and procedures. In level 5, program loop control structures are changed to an equivalent form using a different control structure (i.e., "for" changed to "if").

With a few simple editor operations it is possible to produce a plagiarized program with a vastly different visual appearance. This makes the manual detection of plagiarized programs difficult in large classes.

ALGORITHMS TO DETECT PLAGIARISM

Programs that automatically grade student programs have been common for more than a decade [3]–[5]. A newer development is the automatic detection of plagiarism in student programs.

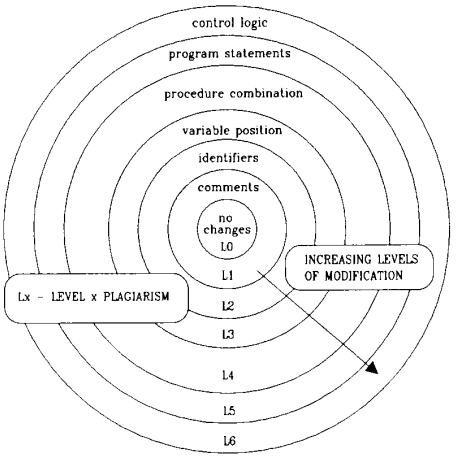


Fig. 1. Program plagiarism spectrum.

A number of algorithms to detect plagiarism can be found in the technical literature. An easy to implement algorithm based on string comparisons from [6] is shown below:

1) Remove all comments.
2) Ignore all blanks and extra lines, except when needed as delimiters.
3) Perform a character string compare between the two files using UNIX utilities, *diff*, *grep*, and *wc*.
4) Maintain a count of the percentages of characters which are the same. This measure is called character correlation.

This algorithm is run for all possible program pairs. A summary containing the character correlation in descending order is generated and examined. This algorithm uses several UNIX utilities, *diff* compares two files, *grep* searches for strings, *wc* counts words or lines, and *sed* edits files.

This simple algorithm will detect many cases of plagiarism. It is easy to implement; however, it requires a substantial amount of computer time. The majority of students who copy programs change comments, the white space, and perhaps a few variable names in the program; however, they are reluctant to make major modifications to the program logic and flow. Faidhi and Robinson classify this as level 1 and level 2 plagiarism [2]. More complex changes require a thorough understanding of the program.

Manuscript received April 13, 1988; revised November 30, 1988.
The authors are with the School of Electrical Engineering, Georgia Institute of Technology, Atlanta, GA 30332.
IEEE Log Number 892720.

Main Program (LEVEL 0 Plagiarism)

```
/* Christopher Kern, January, 1983, BYTE */

#include <stdio.h>

#define TRUE 1
#define FALSE 0
#define SIZE 8190

char  flags[SIZE+1];

main()
{
        int i, prime, k, count, iter;

        printf("10 iterations \n");
        for(iter=1;iter<=10;iter++) {
                count=0;
                for(i=0;i<=SIZE;i++)
                        flags[i]=TRUE;
                for(i=0;i<=SIZE;i++) {
                        if(flags[i]) {
                                prime=i+i+3;
                                for(k=i+prime; k<=SIZE; k+=prime)
                                        flags[k]=FALSE;
                                count++;
                        }
                }
        }
        printf("%d %d\n", prime, count);
}
```

LEVEL 1 Plagiarism

```
/* George P. Burdell */

#include <stdio.h>

#define TRUE 1
#define FALSE 0
#define SIZE 8190


char  flags[SIZE+1];

main()
{
int i, prime, k, count, iter;
/* declare variables in program */


printf("10 iterations \n");
/*print number of iterations*/

for(iter=1;iter<=10;iter++) {
/* main loop */


count=0;
for(i=0;i<=SIZE;i++)
  flags[i]=TRUE;
for(i=0;i<=SIZE;i++) {
  if(flags[i]) {
  prime=i+i+3;
  for(k=i+prime; k<=SIZE; k+=prime)
    flags[k]=FALSE;
  count++;
  }
}
  }

  printf("%d %d\n", prime, count);
/* print result */

}
```

LEVEL 3 Plagiarism

```
/* George P. Burdell */

#include <stdio.h>

#define FALSE 0
#define UNUSED 4
#define ALWAYS_TRUE 1

char  prime[8191];

main()
{
int i, j, primes, k, count, iteration;
/* declare variables in program */


printf("10 iterations \n");
/*print number of iterations*/

for(iteration=1;iteration<=10;iteration++) {
/* main loop */

counter=0;
for(i=0;i<=SIZE;i++)
  prime[i]=ALWAYS_TRUE;
for(i=0;i<=SIZE;i++) {
  if(prime[i]) {
  primes=i+i+3;
  for(k=i+primes; k<=SIZE; k+=primes)
    prime[k]=FALSE;
  counter++;
  }
}
  }

  printf("%d %d\n", primes, counter);
/* print the result */

}
```

LEVEL 5 Plagiarism

```
/* George P. Burdell */

#include <stdio.h>

#define ALWAYS_TRUE 1
#define FALSE 0
#define UNUSED 4

char  prime[8191];

main()
{
int i, j, primes, k, count, iteration;
/* declare variables in program */

printf("10 iterations \n");
/*print number of iterations*/

for(iteration=1;iteration<=10;iteration++) {
/* main loop */

counter=0;
for(i=1;i<=SIZE+1;++i)
  prime[i-1]=ALWAYS_TRUE;
    i=0;
    do  {
      if(prime[i]) {
        primes=i+i+3;
        for(k=i+primes; k<=SIZE; k+=primes)
          prime[k]=FALSE;
        count++;
        i++;
      }
    while(i<=SIZE);
    }
  }

  printf("%d %d\n", primes, counter);
 /* print the result */

 }
```

Fig. 2. Program plagiarism example.

Software Metrics

Much of the work in the development of algorithms for plagiarism detection has centered around Halstead's theory of software science [7]–[9]. The study of software metrics lies in the field of software engineering and it has its roots in both psychology and computer science. Halstead introduced a number of measures based on simple program statistics n_1, n_2, N_1, and N_2 defined as

n_1 = number of unique operators
n_2 = number of unique operands
N_1 = number of operator occurrences
N_2 = number of operand occurrences.

Two common measures using the above are

$$V = (N_1 + N_2) \log_2 (n_1 + n_2)$$

and

$$E = [n_1 N_2 (N_1 + N_2) \log_2 (n_1 + n_2)]/(2n_2).$$

V represents the volume of a program and E represents a measure of the mental effort required for the program's creation. Halstead's metrics have been used to predict software complexity for both experienced and novice programmers. Interestingly, a strong correlation has been shown between Halstead's measures and the following:

a) The number of bugs in a program [10]–[13]
b) The programming time required for a program [14], [15]
c) The debugging time required for a program [16], [17]
d) The algorithmic purity of a program [18], [19].

In addition to the Halstead metrics, a number of newer software metrics have been introduced [20] to measure effectiveness, style, system costs, reliability, flexibility, and structure. The strong correlation between software metrics and these program characteristics suggests that software metrics should be useful in plagiarism detection.

Software Metric Algorithms

Several algorithms for plagiarism detection are based on software metrics [21]–[27]. These algorithms extract several software metric features from a program and use this set of measures or features to compare programs for plagiarism. The extraction of these features requires scanning, searching a reserved word table, construction of a symbol table, and limited parsing of the programming language. Implementation of these feature extraction algorithms is similar in principle to the operation of a simple compiler [28]–[30]. The entire process can be viewed as a classical pattern recognition system which extracts a number of features from the input data and then uses statistical distance or correlation measures on this feature set to compare the patterns or programs [31]–[33]. Research indicates that program implementations of the more complex software metric based algorithms require several thousand lines of code. Since the implementation of these algorithms requires a compiler-like front end, a modified version of the algorithm is required for each programming language.

Ottenstein made the first presentation [21] of Halstead's simple software metrics n_1, n_2, N_1, and N_2 for plagiarism detection in Fortran programs at Purdue University. Other researchers indicate that this method is not effective for very short programs [2], [23]. Grier [22] at the U.S. Air Force Academy used the following measures for Pascal programs in his ACCUSE program:

A_1 = Total lines
A_2 = Code lines
A_3 = Code comment lines
A_4 = Multiple statement lines
A_5 = Constants and Types
A_6 = Variables declared (and used)
A_7 = Variables declared (and not used)
A_8 = Procedures and functions
A_9 = Var parameters
A_{10} = Value parameters
A_{11} = Procedure variables
A_{12} = For statements
A_{13} = Repeat statements
A_{14} = While statements
A_{15} = Goto statements
A_{16} = Unique operators
A_{17} = Unique operands
A_{18} = Total operators
A_{19} = Total operands
A_{20} = Indenting function

Donaldson, Lancaster, and Sposato [23] at Bowling Green State University have used the following measures for FORTRAN programs:

B_1 = Total number of variables
B_2 = Total number of subprograms
B_3 = Total number of input statements
B_4 = Total number of conditional statements
B_5 = Total number of loop statements
B_6 = Total number of assignment statements
B_7 = Total number of calls to subprograms
B_8 = Total number of statements $B_2 \cdots B_7$

Berghel and Sallach [24] report using the following software metrics for FORTRAN plagiarism detection:

f_1 = code lines
f_2 = total lines
f_3 = continuation statements
f_4 = keywords
f_5 = real variables
f_6 = integer variables
f_7 = total variables
f_8 = assignment statements (initialization)
f_9 = assignment statements
f_{10} = declared reals
f_{11} = declared integers
f_{12} = total operators
f_{13} = total operands
f_{14} = unique operators
f_{15} = unique operands

Rees [25] at the University of Southampton uses a number of software metrics to automatically award points for

Pascal style in addition to plagiarism detection. Typically, program style amounts to 10 percent of the total grade on programming assignments. His program, STYLE, assigns the student's grade for style by examining metrics in the two categories below:

Layout:
1) average line length
2) use of comments
3) use of indentation
4) use of blank lines as separators
5) degree of imbedded spaces within lines

Identifiers:
6) procedure and function units
7) variety of reserved words
8) length of identifiers
9) variety of identifier names
10) use of labels and gotos

The following metrics extracted by the STYLE program are used for plagiarism detection in Rees' and Robson's CHEAT postprocessing system:

C_1 = total number of noncomment characters
C_2 = percentage of embedded spaces
C_3 = number of reserved words
C_4 = number of identifiers
C_5 = total number of lines
C_6 = number of procedures/functions.

The following list of measures were introduced by Faidhi and Robinson [2] at Brunel University to identify features which assist in the detection of plagiarism in Pascal programs:

m_1 = number of characters per line
m_2 = number of comment lines
m_3 = number of indented lines
m_4 = number of blank lines
m_5 = average procedure/function length
m_6 = number of reserved words
m_7 = average identifier length
m_8 = average spaces percentage per line
m_9 = number of labels and gotos
m_{10} = variety of identifiers
m_{11} = number of program intervals
m_{12} = number of colors used in coloring the control flow graph
m_{13} = number of vertices colored with color 3 in coloring the control flow graph
m_{14} = number of vertices colored with color 4 in coloring the control flow graph
m_{15} = Program structure percentage = $100*N_e/(N_e + N_{se} + N_t + N_f)$
m_{16} = Program simple expression structure percentage = $100*N_{se}/(N_e + N_{se} + N_t + N_f)$
m_{17} = Program term structure percentage = $100*N_t/(N_e + N_{se} + N_t + N_f)$
m_{18} = Program factor structure percentage = $100*N_f/(N_e + N_{se} + N_t + N_f)$
m_{19} = Program impurity percentage: percentage of impurities
m_{20} = Module contribution percentage
m_{21} = number of modules

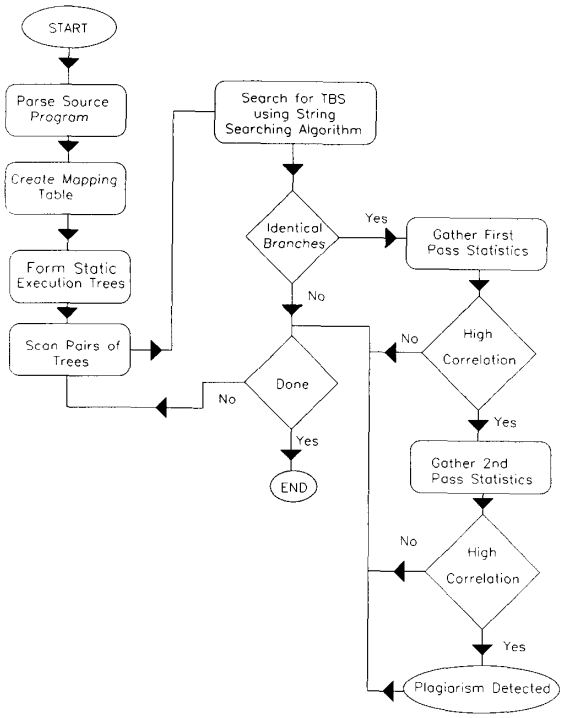


Fig. 3. Plagiarism detection algorithm.

m_{22} = conditional statement percentage
m_{23} = repetitive statement percentage
m_{24} = number of program statements

where

N_e = Number of expressions
N_{se} = Number of simple expressions
N_t = Number of terms
N_f = Number of factors

Novice programmers are most likely to alter m_1–m_{10}. Hidden features related to program structure are contained in m_{11}–m_{24}. This set of metrics was shown to produce more reliable results than earlier measures [25].

An alternate approach introduced by Jankowitz [26] at Southampton University, analyses programs using static execution trees obtained by parsing the main program body and procedures. The algorithm extracts the data from a program to construct the trees and then analyses procedures attached to identical branches. The analysis of procedures is twofold. First, global statistics are obtained on the procedure bodies using the following measures:

y_1 = number of code lines
y_2 = number of variables used
y_3 = number of reserved words
y_4 = number of assignment statements
y_5 = number of If statements
y_6 = number of Repeat/While statements
y_7 = number of For statements
y_8 = number of Case statements
y_9 = number of With statements
y_{10} = number of procedure and function calls.

TABLE I
COMPARISON OF PLAGIARISM ALGORITHMS BASED ON SOFTWARE METRICS

	Algorithm Number						
Program Measure	**1**	**2**	**3**	**4**	**5**	**6**	**7**
Number of characters per line						m1	z1
Number of comment lines		a3		f2		m2	
Number of indented lines		a20				m3	
Number of blank lines						m4	
Average procedure/function length						m5	
Number of reserved words				f4	c3	m6	y3,z2
Average identifier length						m7	
Average space percentage per line					c2	m8	
Number of labels and gotos		a15				m9	
Unique operands	n2	a17,a6,a9,a10,all	b1	f5,f6,f10,f11,f15	c4	m10	y2, z3
Number of program intervals						m11	
Number of colors used						m12	
in coloring the control graph							
Number of vertices colored with color 3						m13	
in coloring the control graph							
Number of vertices colored with color 4						m14	
in coloring the control graph							
Total Operands	N2	a19	b6	f7, f13		m15	y4
Unique Operators	n1	a16,a5		f8,f9, f14		m16	y4
Total Operators	N1	a18		f12		m17	
Program factor structure percentage						m18	
Program impurity percentage		a7				m19	
Module contribution percentage.		a8	b7			m20	y10
Number of modules.			b2		c6	m21	
Conditional statement percentage.			b4			m22	y5,y8, z4, z7
Repetitive statement percentage.		a12, a13, a14	b5	f3		m23	y6, y7, z5, z6
Number of program statements.		a1,a2	b8	f1	c5,c1	m24	y1
Reference sequence order							z9
Multiple statement lines		a4					

If this phase indicates a high degree of correlation between procedure bodies then the following additional measures are applied for each IF, REPEAT/WHILE, FOR, CASE, and WITH statement:

z_1 = length of statement
z_2 = reserved words
z_3 = variables used
z_4 = ifs counted
z_5 = repeat/whiles counted
z_6 = fors counted
z_7 = cases counted
z_8 = withs counted
z_9 = reference sequence order

This procedure is summarized in Fig. 3. By breaking up the analysis into separate sections, faster execution is obtained by avoiding unnecessary comparisons.

For comparison, Table I lists the software metrics used by these algorithms. In this table, various software metrics are categorized that are directly related to metrics used by other researchers.

Simple algorithms rely heavily on Halstead's variable and operand based metrics. In addition to these metrics, the more complex algorithms include newer software metrics which consider procedures and flow control structures.

PLAGIARISM POLICIES

To effectively deal with the plagiarism problem a number of actions [34]–[36] are required in addition to the implementation of a automatic plagiarism detection system. The first step is to inform the students in writing of policies regarding plagiarism. This is especially important if the university's honor code does not deal with computer-related ethics. In such instances, a policy handout should be distributed to students at the start of the course. Many student honor codes are being updated to address computer-related ethical issues such as plagiarism, software licenses, and public domain software. Course policy handouts should clearly state if individual or team efforts are required on programming assignments.

Current plagiarism detection algorithms should be viewed as a tool providing assistance only. It is difficult to set absolute detection limits for these algorithms since many program features can be influenced by program length and suggestions provided by the text, teaching assistants, or instructor. Manual evaluation of computer programs flagged as "plagiarism" is still required. It is useful to sort pairs of programs on the basis of detected plagiarism. After a manual examination of the most suspicious programs, the instructor can determine the appropriate detection threshold values for each assignment. One useful technique is to have students suspected of plagiarism explain the details and structure of their program orally without prior notice.

Once plagiarism has been detected the problem becomes selecting the appropriate disciplinary action. Disciplinary actions can range from a zero on the assignment, reduction by one letter grade in the course, to failure of the course, and ultimately to expulsion from the univer-

sity. In all cases, the penalty for plagiarism must be more severe than the penalty for a late or nonworking assignment. At many schools these problems are referred to a student honor committee. In such cases, it can be difficult to explain program plagiarism to a nontechnical person, when the two programs appear at first glance to be different.

Conclusions

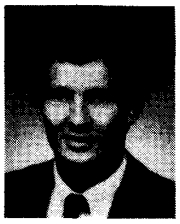
The ultimate goal of automatic plagiarism detection is the reduction of plagiarism. Many cases of plagiarism can be detected by these algorithms which would be easily missed by a human grader. Instances of plagiarism can be greatly reduced, if not eliminated, with the use of a plagiarism detection system and the appropriate administrative procedures and policies.

Acknowledgment

The authors would like to acknowledge the helpful comments and suggestions provided by the reviewers.

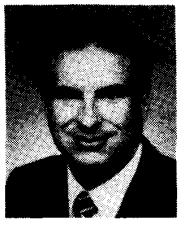
References

[1] *Webster's New World Dictionary of the American Language*, D. B. Guralnik, Ed. Cleveland, OH: William Collins World Pub., 1976.
[2] J. A. W. Faidhi and S. K. Robinson, "An empirical approach for detecting program similarity and plagiarism within a university programming environment," *Comput. Educ.*, vol. 11, pp. 11-19, 1987.
[3] R. F. Bacon, "An investigation of an automatic machine grading system for assembly language instruction," Naval Postgraduate School, Monterey, CA, June 1969.
[4] J. B. Hext and J. W. Winigs, "An automatic grading scheme for simple programming exercises," *Commun. ACM*, vol. 12, pp. 272-275, May 1969.
[5] K. J. Ottenstein, "An algorithmic approach to the detection and prevention of plagiarism," *Sigcse Bulletin*, vol. 8, pp. 30-41, 1976.
[6] J. O. Hamblen and A. Parker, "A new undergraduate computer arithmetic software laboratory," *IEEE Trans. Educ.*, to be published.
[7] A. Fitzsimmons and T. Love, "A review and evaluation of software science," *Computing Surveys*, vol. 10, no. 1, pp. 3-18, 1978.
[8] M. H. Halstead, *Elements of Software Science*. New York: Elsevier, 1977.
[9] —, "An experimental determination of the purity of a trivial algorithm," *ACM SIGME Performance Evaluation Review*, vol. 2, pp. 10-15, 1973.
[10] D. E. Bell and J. E. Sullivan, "Further investigations into the complexity of software," MITRE Tech. Rep. MTR-2874, MITRE Bedford, MA, 1974.
[11] L. M. Cornell and M. H. Halstead, "Predicting the number of bugs expected in a program module," Comput. Sci. Dep. Tech. Rep. CSD-TR-205, Purdue University, West Lafayette, IN, 1976.
[12] Y. Funami and M. H. Halstead, "A software physics analysis of Akiyama's debugging data," in *Proc. MRI 24th Internat. Symp.: Software Eng.*, New York: Polytechnic Press, 1976, pp. 133-138.
[13] R. D. Gordon and M. H. Halstead, "An experiment comparing FORTRAN programming times with the software physics hypothesis," *AFIPS Conf. Proc.*, vol. 45, pp. 935-937, 1976.
[14] M. H. Halstead, "Natural laws controlling algorithm structure," *SIGPLAN Notices*, vol. 7, no. 2, pp. 19-26, 1972.
[15] S. P. Sheppard *et al.*, "Experimental evaluation of on-line program construction," in *Proc. COMPSAC '80*, New York: IEEE, 1980, pp. 505-510.
[16] B. Curtis *et al.*, "Third time charm: Stronger prediction of programmer performance by software complexity metrics," in *Proc. Fourth Internat. Conf. Software Eng.*, New York: IEEE, 1979, pp. 356-360.
[17] L. T. Love and A. Bowman, "An independent test of the theory of software physics," *SIGPLAN Notices*, vol. 11, pp. 42-29, 1976.
[18] J. L. Elshoff, "Measuring commercial PL/1 programs using Halstead's criteria," *SIGPLAN Notices*, vol. 11, pp. 38-46, 1976.
[19] B. Curtis, "The measurement of software quality and complexity," *Software Metrics*, A. Perlis, F. Sayward, and M. Shaw, Eds. Cambridge, MA: M.I.T. Press, 1981, pp. 203-224.
[20] T. Gilb, *Software Metrics*. Cambridge, MA: Winthrop, 1977.
[21] L. M. Ottenstein, "Quantitative estimates of debugging requirements," *IEEE Trans. Software Eng.*, vol. SE-5, pp. 504-514, 1979.
[22] S. A. Grier, "A tool that detects plagiarism in PASCAL programs," *Sigcse Bulletin*, vol. 13, pp. 15-20, 1981.
[23] J. L. Donaldson *et al.*, "A plagiarism detection system," *Sigcse Bulletin*, vol. 13, pp. 21-25, 1981.
[24] H. L. Berghel and D. L. Sallach, "Measurements of program similarity in identical task environments," *Sigplan Notices*, vol. 19, pp. 65-76, 1984.
[25] M. J. Rees, "Automatic assessment aids for Pascal programs," *SigPlan Notices*, vol. 17, no. 10, pp. 33-42, 1982.
[26] H. T. Jankowitz, "Detecting plagiarism in student Pascal programs," *Comput. J.*, vol. 31, no. 1, pp. 1-8, 1988.
[27] J. R. Rinewalt *et al.*, "Development and validation of a plagiarism detection model for the large classroom environment," *CoED*, vol. 6, pp. 9-13, July 1986.
[28] A. Aho and J. Ullman, *Principles of Compiler Design*. Reading, MA: Addison Wesley, 1977.
[29] W. Waite and G. Goos, *Compiler Construction*. New York: Springer-Verlag, 1984.
[30] A. Schreiner and G. Freidman, *Introduction to Compiler Construction with UNIX*. Englewood Cliffs, NJ: Prentice Hall, 1985.
[31] *Digital Pattern Recognition*, K. S. Fu, Ed. New York: Springer-Verlag, 1976.
[32] *Syntactic Pattern Recognition*, K. S. Fu, Ed. New York: Springer-Verlag, 1977.
[33] B. Batchelor, *Pattern Recognition: Ideas in Practice*. New York: Plenum Press, 1978.
[34] M. Shaw *et al.*, "Cheating policy in computer science department," *SIGCSE Bulletin*, vol. 12, Part 2, pp. 72-76, 1980.
[35] J. M. Cook, "Defining ethical and unethical student behaviors using departmental regulations and sanctions," *SIGCSE Bulletin*, vol. 19, no. 1, pp. 462-468, Feb. 1987.
[36] P. L. Miller *et al.*, "Panel discussion: Plagiarism in computer science courses," *Sigcse Bulletin*, vol. 13, pp. 26-27, 1981.



Alan Parker (M'85) was born in Slough Bucks, England, on November 30, 1959. He received the B.S. degree in applied mathematics and the M.S. degree in applied mathematics and the M.S. degree in electrical engineering from Georgia Institute of Technology, Atlanta, in 1980 and 1981 and the Ph.D. degree in electrical engineering from North Carolina State Univesity, Raleigh, in 1984.

He is an Assistant Professor of Electrical Engineering at the Georgia Institute of Technology and teaches undergraduate and graduate courses in computer engineering. In 1981, he was employed as a Microwave Design Engineer at Scientific Atlanta, GA. From 1984 to 1985, he was a Senior Associate Engineer with IBM in the Advanced Systems Architecture Department, Boca Raton FL. His research interests include parallel processing, computer architecture, and high-speed numeric and signal processing.



James O. Hamblem (S'74-M'77) was born in Lafayette, IN, on August 15, 1954. He received the B.S. degree from Georgia Institute of Technology, Atlanta, in 1974, the M.S. degree from Purdue University, Lafayette, IN, in 1976, and the Ph.D. degree from Georgia Institute of Technology in 1984, all in electrical engineering.

He is an Assistant Professor of Electrical Engineering at the Georgia Institute of Technology, Atlanta. He teaches undergraduate and graduate courses in computer engineering. From 1979 to 1984, he was a Graduate Research Assistant at Georgia Institute of Technology. From 1977 to 1978, he was a Senior Engineer at Martin Marietta Aerospace, Denver, CO, and from 1976 to 1977, he was a Systems Analyst at Texas Instruments, Austin, TX. His research interests include parallel computer architectures, VLSI design, and continuous system simulation.

Dr. Hamblen is a member of Eta Kappa Nu, Tau Beta Pi, and Phi Kappa Phi.