바이트코드 분석을 이용한 자바 프로그램 표절검사기법

(A Plagiarism Detection Technique for Java Program Using Bytecode Analysis)

지 정 훈 † 우 균 †† 조 환 규 †††

(Jeong-Hoon Ji) (Gyun Woo) (Hwan-Gue Cho)

요 약 대부분의 표절검사 시스템들은 소스코드를 이용해 유사도를 계산하고 표절 프로그램을 찾아낸다. 소스코드를 이용하여 표절검사를 수행할 경우, 소스코드 보안문제가 발생할 수 있다. 목적 코드를 이용한 표절검사는 소스코드 보안문제에 대한 좋은 대안이 될 수 있다. 본 논문에서는 자바 프로그램의 표절검사에 대하여 소스코드 없이 바이트코드를 이용해 표절검사를 수행하는 방법을 제시한다. 바이트코드를 이용한 표절검사는 크게 두 단계로 진행된다. 먼저, 자바 클래스 파일로부터 메소드의 코드영역을 분석해 토큰 시퀀스를 생성한 다음 적응적 지역정렬을 이용해 유사도를 계산한다. 실험 결과, 소스코드와 바이트코드의 유사도는 비슷한 분포를 보였다. 또한, 소스코드 쌍과 바이트코드 쌍의 유사도 상관관계가 충분히 높게 측정되었다. 본 논문에서 제안한 바이트코드 표절검사 시스템은 소스코드를 이용해 직접 표절을 검사하기 전 단계에서 1차적인 검증도구로 활용할 수 있다.

키워드 : 표절, 프로그램 표절검사, 바이트코드, 유사도, 프로그램 분석

Abstract Most plagiarism detection systems evaluate the similarity of source codes and detect plagiarized program pairs. If we use the source codes in plagiarism detection, the source code security can be a significant problem. Plagiarism detection based on target code can be used for protecting the security of source codes. In this paper, we propose a new plagiarism detection technique for Java programs using bytecodes without referring their source codes. The plagiarism detection procedure using bytecode consists of two major steps. First, we generate the token sequences from the Java class file by analyzing the code area of methods. Then, we evaluate the similarity between token sequences using the adaptive local alignment. According to the experimental results, we can find the distributions of similarities of the source codes and that of bytecodes are very similar. Also, the correlation between the similarities of source code pairs and those of bytecode pairs is high enough for typical test data. The plagiarism detection system using bytecode can be used as a preliminary verifying tool before detecting the plagiarism by source code comparison.

Key words : plagiarism, program plagiarism detection, bytecode, similarity, program analysis

· 이 논문은 2007년도 정부재원(교육인적자원부 학술연구조성사업비)으로 한국학술진흥재단의 지원을 받아 연구되었음(KRF-2007-052-10000)
· 이 논문은 제34회 추계학술대회에서 '바이트코드 분석을 이용한 자바 프로그램 표절검사기법'의 제목으로 발표된 논문을 확장한 것임

† 학생회원 : 부산대학교 컴퓨터공학과
jhji@pusan.ac.kr
†† 종신회원 : 부산대학교 컴퓨터공학과 교수
woogyun@pusan.ac.kr
††† 정 회 원 : 부산대학교 컴퓨터공학과 교수
hgcho@pusan.ac.kr

논문접수 : 2008년 1월 11일
심사완료 : 2008년 6월 4일

정보과학회논문지: 소프트웨어 및 응용 제35권 제7호(2008.7)

1. 서 론

표절은 대학이나 기업에서 큰 문제가 될 수 있다[1]. 한 연구에 따르면 전체 학생들 중 10%이상의 학생들이 과거에 표절을 해본 경험이 있다는 통계가 조사되었다[2]. 또한 MIT(Massachusetts Institute of Technology) 대학에서도 기초프로그래밍 교과목에서 30% 이상의 학생들이 표절행위로 인해 징계를 받았다[3]. 기업에서도 표절로 인해 심각한 문제가 발생할 수 있다. 예를 들어, 오랜 기간과 많은 개발 비용을 들여 개발한 제품이 표절된다면 기업은 막대한 손실을 입게 될 것이다. 이처럼 표절로 인해 많은 문제가 발생하는 것은 대학이나 기업에서 표절에 대한 의식이 낮다고 볼 수 있다. 또한 표절을 방지를 위한 사전 교육과 표절을 정확하게 검출하기 위한 연구가 필요하다.

전산학에서 문서 표절에 대한 연구는 크게 일반문서 표절 연구와 프로그램 표절 연구로 나눌 수 있다. 일반문서 표절에 대한 연구는 영어나 한글과 같은 자연어로 작성된 문서에 대해 유사성을 분석하고 결과를 바탕으로 표절 여부를 판별하는 것이다. 그리고 프로그램 표절에 대한 연구는 같은 목적으로 작성된 서로 다른 프로그램을 기계적으로 분석해 유사성을 밝혀내는 연구다. 지금까지도 프로그램 표절에 대한 많은 연구가 진행되고 있으며, 소스코드를 기계적으로 분석하여 자동으로 표절을 검출하는 시스템들이 개발되었다[4-7]. 자동화된 표절검사 시스템으로는 MOSS[4], YAP[5], JPlag[6], SIM[7] 등이 있다.

지금까지 연구된 대부분의 프로그램 표절검사 시스템들은 소스코드의 유사성을 분석하여 프로그램 표절을 판별한다. 소스코드를 이용한 표절검사에서는 소스코드를 소스코드 보안에 대한 문제가 발생할 수 있다. 대학 강의에서 학생들이 제출한 소스코드의 경우에는 보안이 큰 문제가 아닐 수 있지만, 기업 원천기술에 해당하는 소스코드의 경우, 소스코드의 보안은 매우 중요한 문제이다. 만약, 프로그램 표절검사를 위해 소스코드를 사용하지 않고 컴파일된 실행코드를 이용한다면, 표절검사를 위해 소스코드를 공개하지 않아도 될 것이다.

컴파일된 실행코드를 비교하여 표절을 밝히는 것은 소스코드를 비교하는 것보다 훨씬 어려운 작업이다. 그 이유는 프로그래머가 소스코드의 일부분을 표절하여 수정하더라도, 컴파일된 실행코드는 크게 달라지기 때문이다[8]. 하지만 실행코드 분석을 통해 두 소스코드 사이의 유사성을 찾을 수 있다면, 소스코드를 외부에 노출시키지 않아도 되기 때문에 소스코드를 이용한 표절검사보다 안전하다.

본 논문에서는 바이트코드 분석을 통해 자바 프로그램의 표절을 검사하는 기법에 대해 소개한다. 자바 프로그램 표절검사를 위해 소스코드를 비교하지 않고 클래스 파일만을 이용하여 표절검사를 수행한다. 바이트코드 표절검사는 크게 두 단계로 나 수행된다. 전단부에서는 클래스 파일을 입력으로 받아 바이트코드를 추출하는 바이트코드 선형화[9]를 수행한다. 선형화 작업은 프로그램을 정적으로 분석하여 수행에 필요한 메소드의 코드만 추출하는 것이다. 다음으로 표절검사의 후단부에서는 지역정렬 알고리즘[10]을 이용해 선형화된 두 바이트코드 사이의 유사도를 계산하고 유사구간을 밝혀낸다.

자바 바이트코드는 구조가 복잡하지 않고 분석하기 쉽기 때문에 다른 기계어들에 비해 표절검사에 적합하다. 또한 표절검사에서는 유사구간을 밝혀내는 것이 중요한데, 클래스 파일에 포함되어 있는 디버깅 정보를 이용하면 바이트코드의 유사구간을 이용하여 소스코드의 유사구간을 연결할 수 있다.

본 논문의 구성은 다음과 같다. 2절에서는 관련연구로 프로그램 표절검사에 사용되는 주요 방법들과 클래스 파일의 구조에 대해 알아보고, 3절에서는 바이트코드를 이용한 표절검사 방법에 대해 기술한다. 4절에서는 실험을 통해 소스코드 표절검사와 바이트코드 표절검사를 비교해보고, 상관관계 분석을 통해 바이트코드를 이용한 표절검사가 가능함을 보인다. 마지막으로 5절에서 결론을 맺는다.

2. 관련연구

2.1 프로그램 표절방법

프로그램 표절은 프로그래밍 관련 교과목에서 빈번하게 발생한다. 프로그램 표절은 복사 또는 부분 수정을 통해 짧은 시간에 동일한 기능을 수행하는 프로그램을 작성하는 것을 말한다[11]. 표절을 하는 학생들은 프로그램의 알고리즘이나 구조적 특징을 제대로 이해하지 못하고 코드 복사나 변수명 변경과 같은 단순한 코드 변환 방법을 많이 사용한다. 이전의 표절연구를 통해 잘 알려진 표절방법들을 정리하면 다음과 같다.

1. 단순 소스코드 복사 및 주석 삽입/삭제/변경
2. 소스코드 포맷 변경 및 식별자(identifier) 이름 변경
3. 코드 블록 또는 연산자나 피연산자의 위치 변경
4. 함수(메소드)의 위치 변경 및 합수 합성/분리
5. 변수의 자료형 변환 및 부가적인 변수 및 문장 추가
6. 동일한 의미의 제어구문으로 변경
7. 프로그램의 전체적인 제어구조 변경

위의 표절방법들 중 1~2 방법은 가장 간단한 표절방법이며, 다른 방법들에 비해 상대적으로 찾아내기 쉽다. 3~5 방법은 학생들이 표절검사를 피해가기 위해 가장 많이 사용하는 방법들이다. 이 방법들은 프로그램의 흐

름이나 구조를 모르더라도 쉽게 새로운 소스코드를 만들 수 있기 때문이다. 마지막으로 6~7 방법은 다른 방법들에 비해 상대적으로 어렵고, 표절된 소스코드를 찾아내는 것 또한 상당히 어렵다. 프로그램의 전체 구조를 이해하고 바꾸기 위해서는 새로운 프로그램을 만드는 것과 거의 동일한 시간이 필요하기 때문이다. 그리고 표절검사에서도 전체적으로 구조가 다른 프로그램들에 대해 인위적으로 표절된 것인지 우연히 비슷하게 만들어진 것인지 판단하는 것은 상당히 어렵다[12].

자바 프로그램의 표절에서도 1~7 방법은 빈번하게 사용된다. 바이트코드를 이용해 신뢰성 있는 표절검사를 하기 위해서는 1~7 방법에 대해 소스코드를 이용해 표절검사를 했을 때와 비슷한 수준까지 정확한 검사를 할 수 있어야 한다.

2.2 프로그램 표절검사 기법

프로그램 표절검사 방법은 크게 속성계수법(attribute counting)과 구조적 표절검사방법으로 나눌 수 있다[13]. 속성계수법은 프로그램에 사용된 키워드의 수, 특정 키워드가 프로그램에 나타난 횟수 등을 특성변수로 하여 소프트웨어 측정값(software metric)을 사용하여 두 프로그램의 유사도를 계산한다. 속성계수법은 프로그램 소스코드를 해시값(hashing value)으로 바꾸어 비교하기 때문에 빠르게 표절검사를 수행할 수 있지만, 두 프로그램 사이의 유사구간을 찾을 수 없다는 단점이 있다. 최근에는 압축 알고리즘 이용한 검사 방법이 소개되었다[14]. SID 표절검사 시스템은 Kolmogorov 복잡도에 근거한 압축 알고리즘을 이용하여 두 프로그램의 유사도를 계산한다. 이 방법은 두 프로그램 소스코드를 하나의 문자열 x와 y로 보고, 각각의 문자열에 대한 Kolmogorov 복잡도 $K(x)$, $K(y)$와 두 문자열 x와 y를 연결한 문자열의 Kolmogorov 복잡도 $K(xy)$의 차이를 이용해 유사도를 계산한다.

구조적 검사방법은 프로그램 소스코드에서 키워드 정보를 추출하여 파스트리(parse tree)나 선형의 토큰 시퀀스[6,7,10,15]와 같은 새로운 형태의 객체로 변환하여 이들 사이의 유사도를 비교한다. 구조적 검사방법은 속성계수법에 비해 복잡하고 구현하기가 어렵지만 유사도 계산과 함께 유사구간을 알 수 있다는 장점이 있다. 또한 프로그램의 구조적 특징을 고려하기 때문에 2.1절에서 나열한 표절공격에 효과적으로 대응할 수 있다.

파스트리를 이용한 연구[16]에서는 ANTLR LL(k) 파서 생성기를 표절검사에 알맞게 수정하여 소스코드로부터 파스트리(V_T)를 생성한다. 다음으로 파스트리 V_T로부터 생성 가능한 모든 서브트리를 벡터로 하여 두 프로그램 사이의 유사도를 재귀적(recursive)으로 계산한다. 이 방법은 2.1절의 표절공격 4, 5와 같은 구조적 표절공격 검출 성능이 우수하다.

Gitchell[7], 지정훈[10], 강은미[15]의 연구에서는 생물정보학에서 DNA 서열의 부분적인 유사성을 찾아내는데 사용되는 지역정렬(local alignment)기법을 프로그램 표절검사에 적용하였다. 지역정렬을 이용한 표절검사에서는 두 프로그램 소스코드(P_1,P_2)로부터 사전에 정의된 키워드들을 추출하여 일련의 키워드 서열(S_1,S_2)을 구성한다. 다음으로, 이들 두 서열들의 유사도 점수를 지역정렬 알고리즘을 이용하여 구한다. 지역정렬은 점수(score)에 기반을 둔 유사도 계산 알고리즘으로 부분적으로 유사한 구간을 찾아준다. 최근에는 한글문서 표절검사[17]에서도 지역정렬 방법이 우수한 성능을 보인 연구사례가 있다.

JPlag[6]도 토큰 시퀀스를 이용하는 대표적인 표절검사 시스템이다. JPlag는 Greedy-String-Tiling 알고리즘을 이용하여 유사도를 계산한다. 이 알고리즘은 두 토큰 시퀀스에서 가장 길게 연속적으로 일치하는 구간을 반복하여 찾는 방법이다.

이밖에도 표절검사에 소스코드를 직접 비교하지 않고, GNU 컴파일러의 RTL 코드나 자바의 바이트코드와 같은 중간 코드를 표절검사에 이용한 연구가 있었다. Arwin의 연구에서는 GNU 컴파일러 수트(compiler suite)에 의해 생성되는 RTL 코드를 비교하여 프로그램의 유사도를 계산하였다[18]. GNU 컴파일러는 컴파일링 프로세스 전단부에서 소스코드를 RTL 코드로 변환한 뒤 후단부에서 RTL 코드를 목적코드(object code)로 변환한다. Arwin은 RTL 코드를 비교함으로써, 서로 다른 프로그래밍 언어로 작성된 프로그램들의 표절을 검사하였다. 예를 들어, RTL 코드를 이용하면 C언어와 파스칼로 작성된 두 프로그램 사이의 유사도를 측정할 수 있다.

Baker는 자바 바이트코드를 비교하여 소스코드의 표절을 찾기 위한 연구를 진행하였다[8]. Baker의 연구에서는 자바 역어셈블러(disassembler)를 이용해서 자바 프로그램의 바이트코드를 문자열 코드로 추출하였다. 그리고 추출된 바이트코드를 유닉스 시스템의 텍스트 비교 도구(tool)인 Dup와 Siff, Diff를 이용해서 표절된 프로그램을 찾았다. Baker의 표절검사 연구는 바이트코드를 이용하여 자바 프로그램의 표절검사가 가능함을 보여주었다.

3. PINT$_B$: 바이트코드를 이용한 표절검출 도구

3.1 시스템 구조

PINT$_B$(Plagiarism INvestigating Tool using Bytecode)는 크게 두 부분으로 나누어 표절검사를 수행한다. 표절검사의 전단부에서는 클래스 파일을 입력으로 받아서 중간 표현인 토큰 시퀀스(token sequence)를 생성한

다. 그리고 후단부에서 생성된 토큰 시퀀스 쌍들에 대해 지역정렬을 이용해 두 자바 프로그램 사이의 유사도를 산출한다. 그림 1은 **PINT**$_B$의 시스템 구조도를 나타낸다.

그림 1(a)에서는 자바 프로그램들의 메인 클래스(main class)를 입력으로 받아서, 상수 풀을 검사한다. 상수 풀에는 프로그램이 필요로 하는 모든 클래스에 대한 링크 정보가 들어 있다. 클래스 수집기에서는 이 정보를 이용하여 모든 클래스를 로드한다. 그 다음, **PINT**$_B$는 메소드 호출 그래프 분석을 프로그램 실행에 관련된 메소드들의 바이트코드를 일련의 토큰 시퀀스로 만든다. 메소드 호출 그래프는 메소드 호출에 관련된 바이트코드를 분석하여 정적으로 프로그램을 시작에서부터 끝까지 추적하면서 메소드 호출 순서에 따라 토큰 시퀀스를 만든다. 토큰 시퀀스를 생성할 때, 메소드의 재귀호출에 대해서는 무한루프가 될 수 있기 때문에 메소드 호출을 최초 1회만 추적한다.

그림 1(b)에서는 선형화된 토큰 시퀀스들 사이의 유사도를 산출한다. **PINT**$_B$는 이를 바탕으로 표절 프로그램들을 검출한다. 유사도 산출은 적응적 지역정렬을 토큰 시퀀스에 적용하여 계산한다. 지역정렬은 생물정보학에서 DNA 서열들 사이의 부분적 유사 기능을 밝히는데 사용되는 정렬기법이다. **PINT**$_B$는 지역정렬을 바이트코드 토큰시퀀스에 적용하여 유사도를 계산하고 유사 구간을 찾는다.

3.2 클래스 파일 구조

바이트코드는 가상기계에 의해 실행되어지는 프로그램에 대한 이진코드(binary code)다. 일반적으로 바이트코드는 한 바이트의 연산코드(opcode)와 연산을 위한 피연산자(operand)로 명령어가 구성되며, 이들 명령어는 가상기계에 의해 수행된다. 자바 바이트코드는 총 203개의 명령어로 이루어져 있으며, 컴파일러에 의해 클래스 파일로 저장되고. 자바가상기계(Java Virtual Machine)에 의해 실행된다.

자바 프로그램은 클래스 파일들의 집합이다. 하나의 프로그램은 n개의 클래스 파일로 이루어지며, 하나의 클래스 파일은 헤더, 상수 풀(constant pool), 필드 및 메소드 테이블, 속성(attribute) 테이블로 구성된다. 자바가상기계는 main 클래스에서 시작하여 프로그램 실행에 필요한 클래스 파일들을 동적으로 읽어 들인다.

클래스 파일의 처음에는 클래스 파일을 나타내는 식별자인 'CAFEBABE'가 4바이트의 16진수 형태로 들어 있다. 식별자 다음에는 클래스 파일의 버전 정보와 상수 풀이 위치한다. 상수 풀에는 클래스 파일에서 사용하는 모든 상수 정보들이 저장되어 있다. 상수 풀은 클래스 파일에서 약 60%정도의 크기를 차지하며, 클래스 파일의 모든 정보 접근은 상수 풀을 통해 이루어진다. 상수 풀 다음으로는 클래스에 대한 접근 정보와 필드(멤버변수)와 메소드에 대한 정보들이 위치한다. 그림 2는 클래스 파일의 구조를 나타낸다.

자바 프로그램은 메소드 단위로 실행되며, 메소드의 실행코드인 바이트코드는 메소드 테이블에 위치한다. 예를 들어, *Pa.java* 소스코드에 main() 메소드와 3개의 사용자 정의 메소드가 정의되어 있다면, *Pa.class*에는 기본 생성자를 포함하여 총 5개의 메소드 테이블이 생성된다. 클래스 파일의 필드와 메소드 테이블의 구조는 동일하다. 필드와 메소드 테이블은 기본적으로 접근플래그와 이름과 타입에 대한 상수 풀 인덱스, 속성 테이블 정보를 포함한다.

자바에는 총 7가지의 속성테이블이 정의되어 있다. 필드와 메소드에 사용되는 속성테이블은 다르다. 필드 테이블에는 인스턴스 변수와 관련된 속성 정보만 올 수 있다. 그리고 메소드 테이블에는 메소드의 실행과 관련

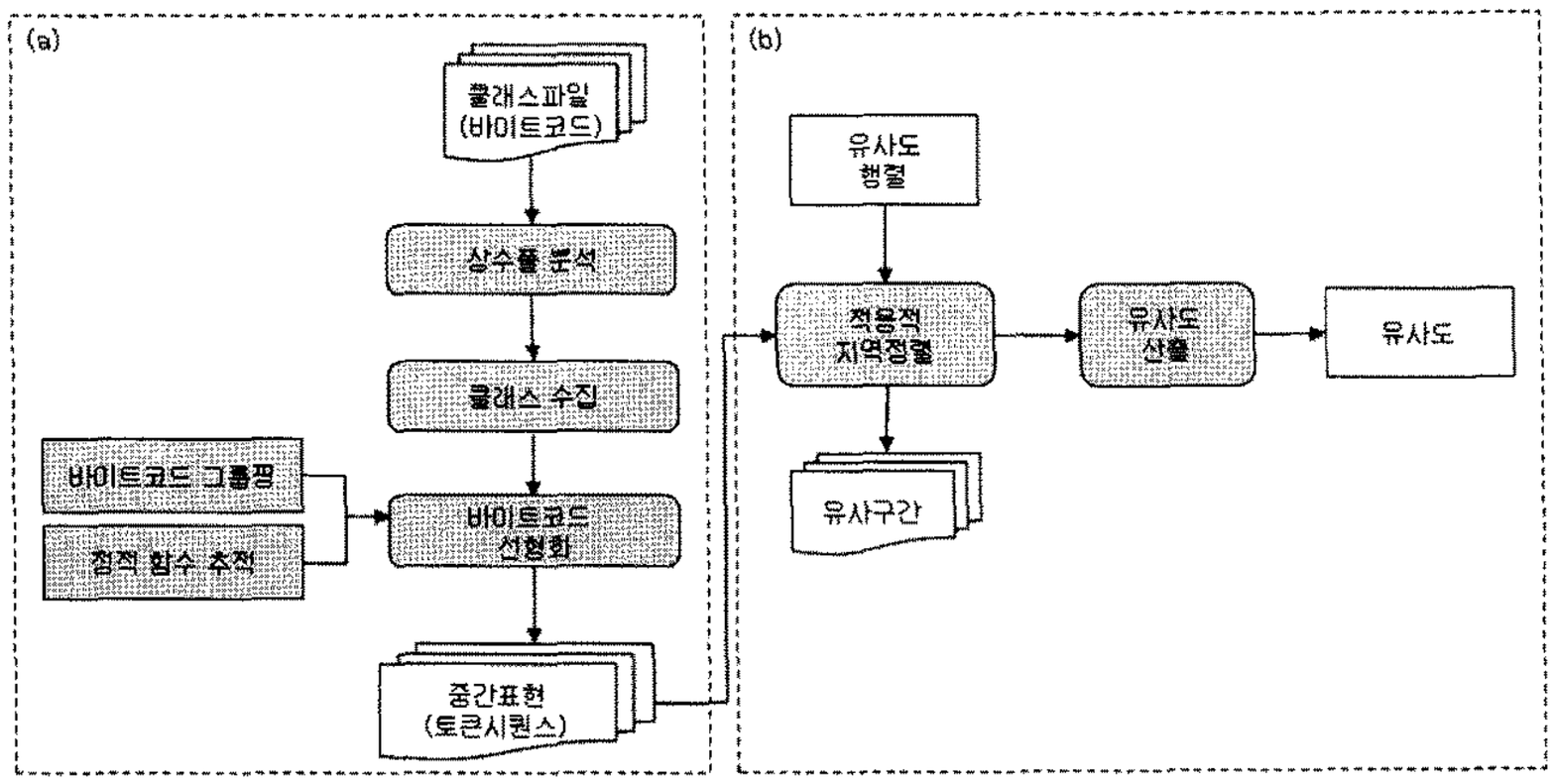


그림 1 **PINT**$_B$ 표절검사 시스템 구조도

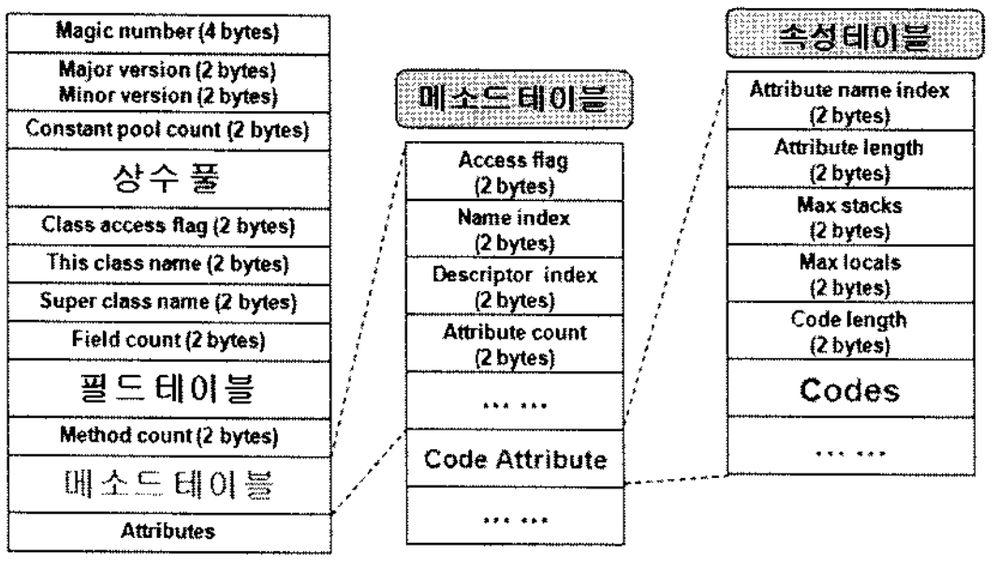


그림 2 클래스 파일 구조

된 속성 정보가 포함된다. 메소드와 관련이 있는 속성들 중, 자바 프로그램 표절검사에서 가장 중요한 속성은 코드 속성(code attribute)이다. 코드 속성에는 메소드의 실행을 위한 바이트코드가 들어 있다. 본 논문의 **PINT**$_B$ 표절검사 시스템은 메소드의 코드 속성 영역에 포함된 바이트코드를 이용해 프로그램의 유사성을 분석한다.

3.3 바이트코드 선형화

바이트코드 선형화 작업은 자바 프로그램의 클래스 파일로부터 바이트코드를 분석해 프로그램 수행에 필요한 바이트코드만을 일련의 시퀀스로 나열하는 작업이다. 바이트코드를 추출하기 위해 **PINT**$_B$는 선형화에서 바이트코드 그룹핑과 메소드 호출 그래프 추적을 수행한다.

바이트코드는 8비트의 이진코드로 이루어져 있으며, 자바에서는 203개의 코드가 사용된다. 예를 들어, 스택 상에 놓인 두 개의 정수를 더하는 코드 iadd는 01100000이며 16진수로는 60H이다. 이 때, 오퍼랜드들은 명령어에 포함되지 않는다. 대부분의 바이트코드들은 스택의 최상위에 놓인 값들을 대상으로 명령을 수행하기 때문에 오퍼랜드를 필요로 하는 명령어는 많지 않다. **PINT**$_B$는 바이트코드의 명령어만을 표절검사에 사용한다.

바이트코드 그룹핑은 동일한 의미의 바이트코드들에 대해 같은 그룹으로 묶는 작업이다. 자바에는 성능향상을 위해 거의 비슷한 연산을 수행하는 동일한 종류의 바이트코드가 여러 개 정의되어 있다. 예를 들어, 지역변수 풀에서 스택으로 값을 옮기는 명령어인 iload와 동일하지만 오퍼랜드를 필요로 하지 않는 명령어인 iload_1, iload_2, iload_3이 정의되어 있다. 이들 명령어는 지역변수가 선언된 위치에 따라 컴파일러에 의해 선택적으로 사용된다. 본 논문에서는 바이트코드의 중복정의로 인한 차이를 줄이기 위해 203개의 바이트코드를 59개의 바이트코드 그룹으로 묶었다. 203개의 바이트코드는 명령어가 수행하는 연산의 종류를 기준으로 분류하였다. 그림 3은 바이트코드 그룹을 보여준다.

바이트코드 그룹핑은 표절검사에서 미세한 차이로 인한 유사도 감소를 줄임으로써, 표절검사의 성능을 향상시킨다. 변수의 위치를 옮기는 것은 빈번하게 사용되는 표절방법이기 때문에 **PINT**$_B$는 표절검사에서 이들 명령어들을 동일한 토큰으로 선형화한다. 그림 4는 소스코드에서 변수 선언 위치에 따른 바이트코드 차이를 보여준다.

그림 4의 *Pa*와 *Pb* 프로그램은 *main*() 메소드에 선언된 변수의 위치가 다르지만 같은 동작을 수행하는 프로그램이다. *Pa*와 *Pb*의 바이트코드를 살펴보면, 선언된 변수의 위치에 따라 사용된 바이트코드가 다름을 볼 수 있다. **PINT**$_B$는 프로그램 *Pa*와 *Pb*의 바이트코드를 그림 5와 같이 선형화한다.

또한, **PINT**$_B$는 바이트코드 선형화에서 메소드 호출 그래프 추적을 통하여 프로그램의 흐름을 표절검사에 반영한다. 일반적으로 프로그램 표절검사는 소스코드의 처음부터 끝까지 순차적으로 비교한다. 그림 4에서 *Pa*와 *Pb* 클래스는 선언된 메소드의 순서가 다르다. *Pa* 클래스는 *sum*()→*main*()의 순서로 선형화되지만, *Pb* 클래스는 *main*()→*sum*()→min()의 순서로 코드가 선형화된다. 순차적 선형화에서 *Pb* 클래스는 프로그램에서 사용되지 않는 min() 메소드까지 함께 토큰 시퀀스에 포함된다. 사용되지 않는 메소드의 삽입이나 메소드의 위치 변경은 2.1절에서 나열한 바와 같이 자주 사용되는 표절방법이다. **PINT**$_B$의 메소드 호출 그래프 분석을 이용하면 실제 사용되는 메소드만을 선형화에 포함시키기 때문에 메소드와 관련된 표절공격에 대해 효과적으로 대처할 수 있다.

3.4 프로그램 유사도 계산

바이트코드 선형화를 마치면 입력된 프로그램 집합의 모든 프로그램에 대해 그림 5와 같은 선형화된 토큰 시퀀스 파일이 생성된다. 입력 프로그램 집합에 n개의 프로그램이 있다면, $n(n-1)/2$개의 토큰 시퀀스 파일은 **PINT**$_B$의 후단부 작업의 입력이 된다. 표절검사 후단부

그룹	바이트코드	속성	사용
bc_add	iadd, fadd, dadd	덧셈 산술연산	var1 = 100 + 200;
bc_push	bipush, sipush, …	스택에 상수 값 삽입	constant -> stack
bc_load	iload, iload_1, iload_2, fload, …	스택값 이동	local variable -> stack
bc_store	istore, istore_1, istore_2, fstore, …	변수값 저장	stack -> local variable
bc_invoke	invokevirtual, invoke_static, …	메소드 호출	obj.method();
… …	… …	… …	… …

그림 3 바이트코드 분류 표

Pa 클래스의 소스코드

```
1.  public class Pa {
2.      public static int sum(int m, int n){
3.          return m + n;
4.      }
5.
6.      public static void main(String args){
7.          int num1 = 100;
8.          int num2 = 200;
9.          int sum;
10.         sum = pa.sum(num1, num2);
11.     }
12. }
```

*Pa*의 *main* 메소드에 대한 바이트코드

```
1.  bipush 100
2.  istore_1
3.  sipush 200
4.  istore_2
5.  iload_1
6.  iload_2
7.  invokestatic    #2: //Method sum:(II)I
8.  istore_3
9.  return
```

Pb 클래스의 소스코드

```
1.  public class Pb {
2.      public static void main(String args){
3.          int sum;
4.          int num1 = 100;
5.          int num2 = 200;
6.          sum = pa.sum(num1, num2);
7.      }
8.
9.      public static int sum(int m, int n){
10.         return m + n;
11.     }
12.
13.     public static int min(int m, int n){
14.         return m - n;
15.     }
16. }
```

*Pb*의 *main* 메소드에 대한 바이트코드

```
1.  bipush 100
2.  istore_2
3.  bipush 100
4.  istore_3
5.  iload_2
6.  iload_3
7.  invokestatic    #2; //Method sum:(II)I
8.  istore_1
9.  return
```

그림 4 지역변수 위치 변화에 따른 바이트코드 차이

```
1:  bc_push
2:  bc_store
3:  bc_push
4:  bc_store
5:  bc_load
6:  bc_load
7:  bc_invokestatic
8:  bc_load
9:  bc_load
10: bc_add
11: bc_return
12: bc_store
13: bc_return
```

그림 5 *Pa*와 *Pb*의 바이트코드 선형화

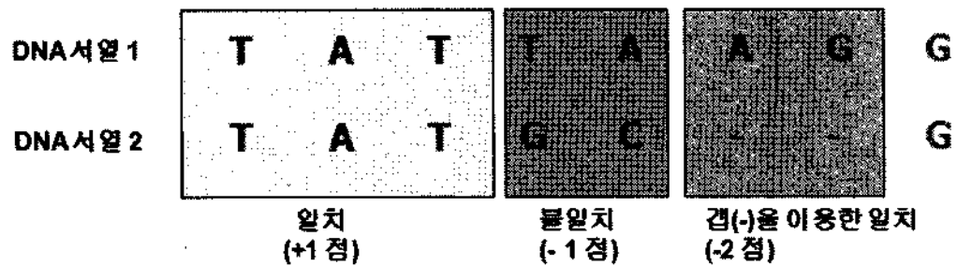


그림 6 Smith-Waterman 지역정렬 알고리즘

작업에서는 토큰 시퀀스 쌍들에 대해 유사도를 계산하고 유사구간을 찾는다. 유사도는 프로그램이 전체적으로 얼마나 유사한지를 나타내고, 유사구간은 두 프로그램에서 표절로 의심되는 구간을 의미한다.

바이트코드의 유사도는 적응적 지역정렬(adaptive local alignment) 알고리즘에 의해 계산된다[10]. 적응적 지역정렬 알고리즘은 제한된 소스코드 집합의 표절탐색을 위한 연구에서 제안되었으며, 프로그래밍 교과목 강의와 ACM에서 주최하는 ICPC(International Collegiate Programming Contest)에서 실제로 사용된 바 있다. 적응적 지역정렬 알고리즘을 이용한 유사도 계산을 요약하면 다음과 같다.

기본적인 지역정렬 알고리즘은 Smith와 Waterman에 의해 제안된 것으로 DNA 서열분석에 주로 사용되는 점수(score)에 기반을 둔 알고리즘이다[19]. 그림 6은 Smith-Waterman 알고리즘의 동작을 보여준다.

일반적으로 지역정렬은 두 서열, $D_1 = \{d_1, d_2, ..., d_m\}$과 $D_2 = \{d_1, d_2, ..., d_n\}$에서 i번째 서열과 j번째 서열이 일치하면($d_i = d_j$) +1, 불일치하면($d_i \neq d_j$) -1, 갭을 이용한 일치이면(d_i or d_j = '_') -2 점을 할당한다. 서열이 연속적으로 일치하면 일치 점수가 계속 누적된다. 그리고 지역정렬은 두 서열에 조금 다른 부분이 있더라도, 유사구간이 끊기지 않고, 불일치와 갭에 의한 감점(penalty)을 주고 구간을 이어간다.

PINT$_B$는 토큰 시퀀스 정렬을 위해 적응적 지역정렬 알고리즘을 사용한다. 적응적 지역정렬은 일치와 불일치, 갭을 이용한 일치에 고정적인 점수(각각 +1, -1, -2)를 부여하지 않고, 프로그램 그룹의 키워드 빈도 벡터(keyword frequency vector) 특성에 따라 적응적으로 점수를 부여한다. 여기서 키워드는 프로그램 언어에 따라 사전에 정의된(predefined) 토큰을 말한다. 적응적 지역정렬은 키워드 빈도를 바탕으로 유사도 행렬을 결정한다. 적응적 지역정렬은 사용 빈도가 높은 키워드의 일치에 대해서는 낮은 점수를 부여하고, 사용 빈도가 낮은 키워드의 일치에는 높은 점수를 부여한다. 적응적 지역정렬의 유사도 계산은 먼저 키워드 빈도 벡터를 구한다. n개의 프로그램으로 구성된 그룹 P의 키워드 벡터를 f^P라고 할 때, 키워드 k_i가 사용된 빈도 f_i^P는 전체

키워드의 사용 횟수에 대한 키워드 빈도의 비율이다. 그리고 키워드 벡터의 모든 원소의 합은 1이다. 다음으로, 적응적 지역정렬은 유사도 행렬을 결정한다. 바이트코드를 이용한 표절검사에서는 키워드 빈도 벡터는 3.2절의 선형화 작업에서 수집한 바이트코드의 빈도 벡터가 된다.

적응적 지역정렬의 유사도 행렬은 키워드 빈도 벡터 f_i^P에 따라 결정된다. 두 프로그램에서 사용된 임의의 바이트코드를 (b_i, b_j)라고 하면, b_i와 b_j가 일치한다면 $-\alpha\log_2 f_i \bullet f_j$를 부여하고, 두 바이트코드가 일치하지 않을 경우에는 $\beta\log_2 f_i \bullet f_j$를 부여한다. 그리고 갭에 의해 강제적으로 일치시킬 경우에는 $4\beta\log_2 f_{i\,or\,j}$의 점수를 부여한다. 적응적 지역정렬의 유사도 행렬은 프로그램 그룹에 따라 다르게 결정된다. 그리고 유사도 행렬에서 α와 β의 합은 항상 1이 되도록 하며, 이 값들은 일치와 불일치에 대한 가중치로써 0과 1사이에서 사용자가 임의로 설정할 수 있다($\alpha+\beta=1$).

유사도 행렬이 결정되면 행렬을 이용해 각 프로그램 쌍에 대해 유사도 점수(SIM_{abs})와 유사도를 구한다. 유사도 점수는 유사도 행렬을 두 프로그램 A와 B의 토큰 시퀀스에 적용했을 때, 구해지는 유사구간의 최고 점수이다. 유사도는 두 프로그램의 유사도 점수 $SIM_{abs}(A,B)$를 0~100% 구간으로 정규화한 값이다. $\mathbf{PINT}_B$는 정규화된 유사도를 구하기 위해 $SIM_{sum}(A,B)$과 $SIM_{\min}(A,B)$를 사용한다. $SIM_{sum}(A,B)$은 일반적인 정규화 방법으로 각 프로그램에 대하여 자신과 정렬시킨 후 유사도 점수의 합으로 나누는 것이다. $SIM_{sum}(A,B)$은 다음과 같이 정의된다.

$$SIM_{sum}(A,B)=\frac{2SIM_{abs}(A,B)}{SIM_{abs}(A,A)+SIM_{abs}(B,B)}$$

$SIM_{min}(A,B)$은 $SIM_{abs}(A,A)$와 $SIM_{abs}(B,B)$중 작은 값을 기준으로 정규화하는 방법이다. 이 방법은 두 프로그램의 길이 차이로 인해 유사도가 낮아지는 것을 보완한다. $SIM_{min}(A,B)$은 다음과 같이 정의된다.

$$SIM_{min}(A,B)=\frac{SIM_{abs}(A,B)}{\min\{SIM_{abs}(A,A),\,SIM_{abs}(B,B)\}}$$

4. 실험 및 평가

4.1 실험데이타

본 논문에서는 바이트코드를 표절검사에 사용 가능한지 알아보기 위해 자바 프로그램을 대상으로 자바 소스코드를 이용한 표절검사 결과와 바이트코드를 이용해 표절검사를 수행한 결과를 비교해 보았다. 표절검사 실험에는 2006학년도 자바프로그래밍 강의에서 학생들이 제출한 소스코드와 자동표절검사 시스템인 JPlag에서 사용한 자바 소스코드를 사용하였다. 표절검사 실험에 사용된 실험데이타를 정리하면 표 1과 같다.

표 1 실험에 사용된 테스트 프로그램 집합: 2006년도 자바프로그래밍 실습문제(G01, G02), 표절검사시스템 JPlag에서 제공하는 소스코드(G03 : AWT를 이용한 GUI 프로그램)

그룹	소스 코드	순서쌍 수	클래스 수	소스코드 라인 수			
				최대	최소	평균	표준 편차
G01	35	595	84	146	46	79.05	23.84
G02	32	496	69	180	50	87.41	33.23
G03	30	435	33	942	199	355.00	131.30

4.2 유사도 분포

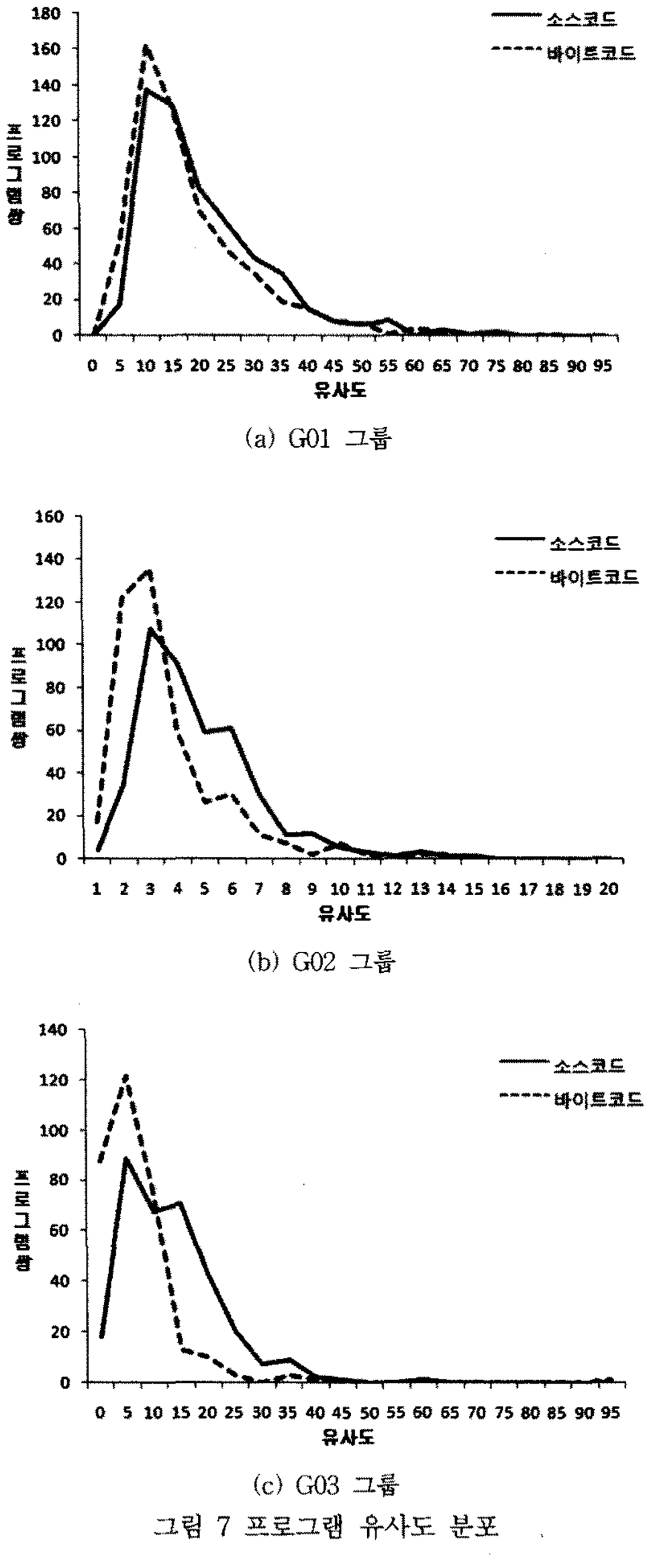
이 절에서는 실험데이타 셋에 대한 유사도 분포에 관해 기술한다. 여기서는 실험데이타 셋에 대하여 소스코드를 이용해 표절검사를 수행했을 때와 바이트코드를 이용해 표절검사를 수행했을 경우에 대해 유사도 분포 결과를 조사하였다.

소스코드를 이용한 표절검사에서는 적응적 지역정렬 연구에서 개발된 표절검사 엔진을 사용하였다. 이 연구에서는 C와 C++ 프로그램에 대해 표절검사를 수행했었다. 본 논문에서는 자바 소스코드 표절검사를 위한 토큰 추출을 위해 JavaCC[20]를 이용하였다. 토큰 시퀀스 추출에서는 자바에서 사용하는 주요 키워드 및 연산자들만 수집한다. 소스코드 표절검사에서는 식별자(identifier), 상수, 주석 등은 표절검사에서 제외된다.

소스코드 표절검사와 바이트코드 표절검사에서 사용된 유사도 계산을 위한 모듈은 동일하다. 유사도 산출에는 적응적 지역정렬을 사용했으며, α와 β값은 0.5로 설정하였다. 그리고 유사도 산출에는 $SIM_{min}(A,B)$ 함수를 사용했다. 그림 7은 프로그램 그룹에 대한 유사도 분포다.

그림 7(a)는 G01 프로그램 그룹의 모든 프로그램 쌍들에 대한 유사도 분포다. G01 그룹의 경우는 소스코드를 이용한 표절검사 결과와 바이트코드를 이용한 표절검사 결과가 비슷한 분포를 나타내었다. 그리고 그림 7(b)와 7(c)의 G02, G03 그룹의 경우에는 유사도가 30% 미만일 경우에 바이트코드의 유사도가 소스코드에 비해 낮게 분포하였다.

G01 그룹의 전체 유사도 평균을 살펴보면, 소스코드의 유사도 평균은 22.96%이었고, 바이트코드의 유사도 평균은 19.11%로 약 3.85% 가량 차이를 보였다. G02 그룹의 경우에는 21.10%와 15.53%로 바이트코드가 소



(a) G01 그룹

(b) G02 그룹

(c) G03 그룹

그림 7 프로그램 유사도 분포

스코드에 약 5.57% 낮은 결과를 보였다. 이는 30% 이하의 유사도를 보인 소스코드들의 바이트코드에서 많은 차이를 보였기 때문이다. 그러나 G02 그룹에서 가장 높은 유사도를 보인 프로그램 쌍은 소스코드와 바이트코드의 경우 모두 동일한 프로그램 쌍인 것으로 나타났으며, 유사도는 71.74%와 69.46%로 약 2.28%의 근소한 차이를 보였다. 또한 G03 그룹에는 소스코드로 검사했을 때, 유사도가 100%인 동일한 프로그램 쌍이 포함되어 있었다. 이 프로그램 쌍들은 바이트코드를 이용해 검사했을 때도 유사도가 100%로 표절된 프로그램으로 보고하였다. 이는 소스코드를 의도적으로 표절해 유사할 경우, 바이트코드 검사를 통해서 검출할 수 있음을 보여준다.

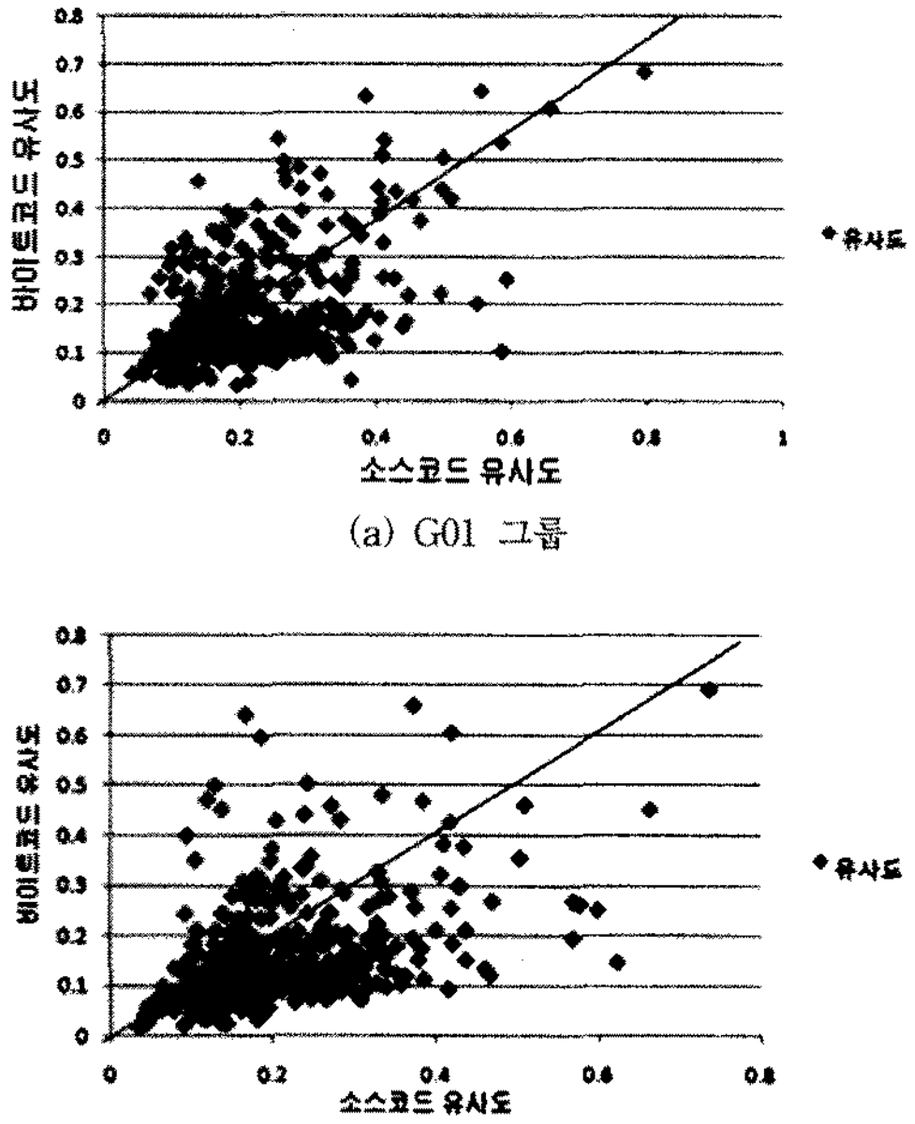
마지막으로 본 논문에서는 바이트코드를 이용한 표절검사가 적합한지 알아보기 위해 소스코드 쌍의 유사도와 바이트코드 쌍의 유사도 사이의 상관관계(correlation)를 측정하였다. 둘 사이의 상관관계가 높다는 것은 바이트코드를 이용하여 자바 프로그램의 표절검사가 가능하다는 것을 나타낸다. 상관관계를 알아보기 위하여 x축을 소스코드 쌍의 유사도로 두고, y축을 이에 대응하는 바이트코드 쌍의 유사도로 설정하였다. 그림 8은 G01그룹의 소스코드 쌍과 바이트코드 쌍의 산포도를 보여준다.

그림 8에서 각 그룹의 상관계수 값은 0.487, 0.318, 0.711로 측정되었다. 그림 8의 붉은 실선은 소스코드 쌍의 유사도와 바이트코드 쌍의 유사도가 동일한 경우를 나타낸다. 그림 8(a)를 살펴보면 유사도가 5%에서 50%까지의 구간에서는 소스코드와 바이트코드의 유사도가 많은 차이를 보임을 알 수 있다. 반면에, 유사도가 50% 이상인 프로그램 쌍들에 대한 유사도 상관계수 값은 0.761로 높게 측정되었다. 높은 유사도를 보이는 프로그램 쌍들에 대해서는 소스코드와 바이트코드의 유사도가 비슷한 것을 볼 수 있다.

그림 8(b)의 경우 소스코드와 바이트코드의 유사도 상관계수가 다른 그룹에 비해 낮은 수치를 보였다. G02 그룹에서 소스코드와 바이트코드를 검토해본 결과, 유사도 차이를 보이는 것은 2.1절의 표절공격 5번과 같이 변수의 타입, 선언위치, 개수에 차이에 따른 것으로 나타났다. 마지막으로 G03 그룹의 경우, 소스코드와 바이트코드 사이의 유사도 상관계수 값이 높게 측정되었다.

그림 8(c)를 살펴보면, 대부분의 프로그램 쌍들에서 두 경우 모두 유사도가 낮다는 것을 알 수 있다. 그리고 표절한 프로그램 쌍에 대해서는 소스코드와 바이트코드의 유사도가 거의 100%에 가깝게 측정되었다.

일반적으로 표절검사에서는 낮은 유사도의 프로그램 쌍보다 높은 유사도의 프로그램 쌍들이 중요하다. 표절검사를 하는 이유는 소스코드 집합의 모든 프로그램 쌍들 중에서 기준 값(threshold value) 이상으로 유사도를 보이는 프로그램 쌍을 찾는 것이 목적이기 때문이다. 그러므로 그림 8의 실험결과는 일정수준 이상의 유사도에서는 소스코드 없이 바이트코드를 이용하여 자바 프로그램의 표절검사가 가능함을 보여준다.



(a) G01 그룹

(b) G02 그룹

(c) G03 그룹

그림 8 소스코드와 바이트코드의 유사도 상관관계

5. 결론 및 향후 연구

본 논문에서는 자바 프로그램의 표절검사에서 소스코드 없이 바이트코드만으로 표절검사를 수행하는 방법을 제안하였다. 표절검사를 위해 **PINT**$_B$는 바이트코드 선형화와 지역정렬을 이용해 프로그램의 유사도를 계산하였다. 바이트코드 선형화에서는 정적 함수 추적을 통해 프로그램의 실행에 관련된 바이트코드만을 추출함으로써 불필요한 코드(dummy code) 삽입, 메소드 합성, 위치변경과 같은 표절공격에 대한 효율성을 높였다. 그리고 바이트코드 그룹핑을 통해 바이트코드의 미세한 차이로 인한 유사도 감소현상을 줄임으로써 표절검사의 성능을 향상시켰다.

지역정렬을 이용한 토큰 시퀀스의 유사도 계산에서는 프로그램 집합에서 사용된 키워드 빈도에 따라 적응적으로 유사도 점수가 계산되는 적응적 유사도 행렬을 사용하였다. 적응적 유사도 행렬은 키워드 사용빈도에 따라 다르게 점수를 할당하였다. 이 방법은 프로그램 집합의 특성을 고려하여 표절검사의 정확성을 높였다.

바이트코드를 이용해 표절검사를 할 경우, 소스코드 없이 표절검사가 가능하기 때문에 소스코드 보안이 중요한 경우에서 효과적으로 이용될 수 있다. 본 논문에서 구현한 **PINT**$_B$는 자바 프로그램 표절검사를 위한 1차적인 검증도구로 이용할 수 있다. 바이트코드를 이용한 표절검사의 단점은 소스코드의 작은 차이가 목적 코드에서는 바이트코드에서는 큰 차이로 나타날 수 있기 때문에 표절의 정확성이 떨어질 수 있다는 것이다.

이와 같은 단점을 보완하기 위해, 향후 연구로는 빈번하게 사용되는 소스코드 표절 공격들에 대해 컴파일러에 의해 생성되는 바이트코드의 패턴을 분석하여 바이트코드 표절검사와 소스코드 표절검사의 차이를 줄이는 방법에 대해 연구가 필요하다. 또한, 실제로 표절된 자바 프로그램을 많이 확보하여 표절검사를 수행해 보고 다른 시스템들과 비교해 보는 작업이 필요하다.

참 고 문 헌

[1] S. Mann and Z. Frew. Similarity and originality in code: plagiarism and normal variation in student assignments. In ACE'06: *Proceedings of the 8th Australian conference on Computing education*, pages 143-150, 2006.

[2] M. Joy and M. Luck. Plagiarism in programming assignments. *IEEE Transactions of Education*, 42(2):129-133, 1999.

[3] C. Daly and J. Horgan. Patterns of plagiarism. *SIGCSE Bull.*, 37(1):383-387, 2005.

[4] A. Aiken. Moss(measure of software similarity) plagiarism detection system. 1998.

[5] M. J. Wise. Detection of similarities in student programs: Yap'ing may be preferable to plague'ing. SIGCSE Bull., 24(1):268-271, 1992.

[6] L. Prechelt, G. Malpohl, and M. Philippsen. Finding plagiarisms among a set of programs with JPlag. *Journal of Universal Computer Science*, 8(11):1016-1038, 2002.

[7] D. Gitchell and N. Tran. Sim: a utility for detecting similarity in computer programs. *SIGCSE Bull.*, 31(1):266-270, 1999.

[8] B. S. Baker and U. Manber, Deducing Similarities in Java Sources from Bytecodes, In *Proceedings of the USENIX Annual Technical Conference*. 179-190. 1998.

[9] J. Ji, G. Woo, and H. Cho. A source code linearization technique for detecting plagiarized programs. In *ITiCSE'07: Proceedings of the 12th annual SIGCSE conference on Innovation and technology in computer science education*, pages 73-

-77, 2007.

[10] 지정훈, 우균, 조환규. 제한된 프로그램 소스 집합에서 표절 탐색을 위한 적응적 알고리즘. *정보과학회논문지: 소프트웨어 및 응용*, 33(12),1090-1102, 2006.

[11] A. Parker and J. O. Hamblen. Computer algorithms for plagiarism detection. *IEEE Transaction on Education*, 32(2):94-99, 1989.

[12] J. Ji, G. Woo, S. Park and H. Cho. Understanding evolution process of program source for investigating software authorship and plagiarism. In *Proceedings of the 2nd International Conference on Digital Information Management*, MLDM Poster, pages 55-69, 2007.

[13] K. L. Verco and M. J. Wise. Software for detecting suspected plagiarism: Comparing structure and attribute-counting systems. In *Proceedings of the 1st Australian Conference on Computer Science Education*, pages 130-134, Sydney, Australia, 1996.

[14] X. Chen, B. Francia, M. Li, B. McKinnon, and A. Seker. Shared information and program plagiarism detection. *IEEE Transactions on Information Theory*, 50(7):1545-1551, 2004.

[15] 강은미, 황미녕, 조환규. 유전체 서열의 정렬 기법을 이용한 소스 코드 표절 검사. *정보과학회논문지: 컴퓨팅의 실제*, 9(3):352-367, 2003.

[16] J. Son, S. Park and S. Park. Program plagiarism detection using parse tree kernels. In *Proceedings of the 9th Pacific Rim International Conference on Artificial Intelligence (PRICAI 2006), Volume 4099 of Lecture Notes in Computer Science*, pages 1000-1004. Springer, 2006.

[17] 류창건, 김형준, 박병준, 최혜정, 조환규. 한글 말뭉치를 이용한 한글 표절 탐색 모델 개발. *한국정보과학회 2007 추계 학술발표 논문집*, 34(2A), 58~59, 2007.

[18] C. Arwin and M. Tahaghoghi. Plagiarism detection across programming languages. In *Proceedings of the 29th Australasian Computer Science Conference*. 48(171), 277-286. 2006.

[19] T. F. Smith and M. S. Waterman, Identification of common molecular subsequences, *Journal of Molecular Biology* 147, 195-197, 1981.

[20] JavaCC : Java Compiler Compiler, https://javacc.dev.java.net/

지 정 훈

2003년 경성대학교 컴퓨터공학 학사. 2005년 경성대학교 컴퓨터공학 석사. 2005년~현재 부산대학교 컴퓨터공학과 박사과정. 관심분야는 프로그래밍언어 및 컴파일러, 프로그램 표절검사, 자바가상기계, 프로그램 시각화

우 균

1991년 한국과학기술원 전산학 학사. 1993년 한국과학기술원 전산학 석사. 2000년 한국과학기술원 전산학 박사. 2000년~2002년 동아대학교 컴퓨터공학과 전임강사. 2002년~2004년 동아대학교 컴퓨터공학과 조교수. 2004년~2007년 부산대학교 컴퓨터공학과 조교수. 2007년~현재 부산대학교 컴퓨터공학과 부교수. 관심분야는 프로그래밍언어 및 컴파일러, 함수형 언어, 그리드컴퓨팅, 소프트웨어 메트릭

조 환 규

1984년 서울대학교 계산통계학과 학사졸업. 1990년 한국과학기술원 공학박사 학위. 1991년부터 현재까지 부산대학교 컴퓨터공학과 교수. 관심분야는 알고리즘 이론, 생물정보학