제한된 프로그램 소스 집합에서 표절 탐색을 위한 적응적 알고리즘

(An Adaptive Algorithm for Plagiarism Detection in a Controlled Program Source Set)

지 정 훈 [†] 우 균 [††] 조 환 규 [†††]
(Jeong Hoon Ji) (Gyun Woo) (Hwan Gue Cho)

요 약 본 논문에서는 대학생들의 프로그래밍 과제물이나 프로그래밍 경진대회에 제출된 프로그램과 같이 동일한 기능을 요구받는 프로그램 소스 집합들에서 표절행위가 있었는지를 탐색하는 새로운 알고리즘을 제시하고 있다. 지금까지 보편적으로 사용되어 온 대표적인 알고리즘은 부분 스트링간의 완전 일치를 통한 Greedy-String-Tiling이나 두 스트링간의 지역정렬(local alignment)을 이용한 유사도 분석이 주된 방법론이었다. 본 논문에서는 해당 프로그램 소스의 집합에서 추출된 키워드들의 빈도수에 기반한 로그 확률값을 가중치로 하는 적응적(adaptive) 유사도 행렬을 만들어 이를 기반으로 주어진 프로그램의 유사구간을 탐색하는 새로운 방법을 소개한다. 우리는 10여개 이상의 프로그래밍 대회에서 제출된 실제 프로그램으로 본 방법론을 실험해 보았다. 실험결과 이 방법은 이전의 고정적 유사도 행렬(match이면 +1, mismatch이면 -1, gap이면 -2)에 의한 유사구간 탐색에 비하여 여러 장점이 있음을 알 수 있었으며, 제시한 적응적 유사도 행렬을 보다 다양한 표절탐색 목적으로 사용할 수 있음을 알 수 있었다.

키워드 : 프로그램 표절탐색, 적응적 지역정렬, 유사도

Abstract This paper suggests a new algorithm for detecting the plagiarism among a set of source codes, constrained to be functionally equivalent, such are submitted for a programming assignment or for a programming contest problem. The typical algorithms largely exploited up to now are based on Greedy-String Tiling, which seeks for a perfect match of substrings, and analysis of similarity between strings based on the local alignment of the two strings. This paper introduces a new method for detecting the similar interval of the given programs based on an adaptive similarity matrix, each entry of which is the logarithm of the probabilities of the keywords based on the frequencies of them in the given set of programs. We experimented this method using a set of programs submitted for more than 10 real programming contests. According to the experimental results, we can find several advantages of this method compared to the previous one which uses fixed similarity matrix (+1 for match, -1 for mismatch, -2 for gap) and also can find that the adaptive similarity matrix can be used for detecting various plagiarism cases.

Key words : program plagiarism detection, adaptive local alignment, similarity

1. 서론 및 연구동기

문서에 기초한 창작물의 표절(plagiarism)은 갈수록 사회문제화 되고 있다. 특히 다양한 디지털 저장매체의 발달과 인터넷의 남용으로 말미암아 저작물의 표절은 그 정도가 심해지고 있으며 그 방법 또한 이전과 달리 교묘해지고 있다. 특히 단순한 호사가의 악의 없는 표절도 문제가 될 수 있지만 대학과 같은 교육기관에서의 표절은 대학사회 내에서 큰 문제가 될 수 있다. 최근 외국의 사례가 보여주듯이 인터넷에서 기존의 과제들을 적절히 짜깁기하여 의뢰한 숙제를 만들어주는 유료 사이트는 갈수록 기승을 부리고 있다. 유료 사이트를 이용하지 않은 경우에도 표절이 빈번하게 발생할 수 있다.

· 이 논문은 부산대학교 자유과제 학술연구비(2년)에 의하여 연구되었음
† 학생회원 : 부산대학교 컴퓨터공학과
jhji@pusan.ac.kr
†† 종신회원 : 부산대학교 컴퓨터공학과 교수
woogyun@pusan.ac.kr
(Corresponding author임)
††† 정 회 원 : 부산대학교 컴퓨터공학과 교수
hgcho@pusan.ac.kr
논문접수 : 2006년 8월 16일
심사완료 : 2006년 10월 30일

대학 내의 특정 교과목의 과제물이 학기 단위로 비슷하다는 특성을 악용하여 이후 학기 수강생에게 이전 학기 과제물을 알려주는 일이 이러한 경우에 속한다.

표절을 검출하는 문제는 프로그래밍 분야에서 더욱 심각하게 대두된다. 인문학관련 과제들은 그나마 강사가 채점을 위하여 읽어보는 단계에서 적절한 수준으로 적발할 수 있지만 프로그램 소스와 같이 그 언어가 자연언어와 매우 다른 경우에는 표절 탐색이 더욱 어렵다. 특히, 인터넷을 이용한 원격강의나 e-Learning이 보편화되는 추세를 감안할 때 각 수강생들이 제출한 과제물을 정확하고 공정하게 평가하는 것은 어떤 교육내용보다 중요하다. 하지만 별도로 제한되지 않은 공간에서 온라인으로 제출하는 과제물이 독립적인 수강생 자신만의 노력으로 이루어졌는지 아니면 이미 공개된 자료를 참조하여 베낀 것인지, 또는 다른 학생들의 과제물을 표절하여 제출한 것인지를 판별하는 문제가 더욱 중요하게 되었다[1]. 게다가 원격교육을 이용한 수강생의 수는 오프라인의 강의실 수강생의 수보다 훨씬 많을 것이므로 정확하면서도 자동화된 어떤 표절검사 시스템이 필수적으로 요구된다[2,3].

본 연구에는 이러한 저작물 중에서 프로그래밍 소스의 표절관계를 탐색하기 위한 기초적인 자료를 조사하고 이를 바탕으로 어떻게 프로그램의 표절을 찾아낼 수 있는지에 대하여 논의하고자 한다. 일반적으로 프로그램의 표절은 어떤 프로그램 A와 프로그램 B 사이에 유사한 면이 있는지를 소스코드 수준에서 살펴보는 것이다.

그러나 일반 문서의 표절연구에서와 마찬가지로 프로그램 표절연구에는 몇 가지 문제가 있다. 먼저 두 프로그램 중 어떤 하나가 다른 하나를 표절하였는지를 판별하는 객관적인 기준은 없다. 극단적으로 말하자면 어떤 두 프로그램이 독립적으로 작성되었다고 하더라도 거의 같을 가능성을 배제할 수 없다. 특히 어떤 문제를 해결하고자 하는 방법이 제한적으로 주어졌을 경우, 예를 들어 배열을 이용한 버블소트(bubble sort)를 작성하라고 제한하면 그 사항을 충실히 따른 프로그램은 거의 같을 수밖에 없다. 결국 이 문제는 확률의 문제로 귀결될 수밖에 없는데, 즉 어떤 길이의 프로그램 A가 다른 프로그램 B와 표절없이 거의 유사할 경우에 대한 확률모델을 제시하고 그에 근거하여 두 프로그램의 표절에 대한 가능성을 밝히는 것이 최선을 방책이라고 할 것이다.

게다가 표절이라는 행위를 기계적으로 판단하기란 어렵다. 예를 들어 두 학생이 서로 협력하여 토론을 한 뒤 같은 아이디어로 작성한 프로그램의 경우와 한 학생이 다른 학생의 프로그램의 한 부분을 불법적으로 복사하여 이를 고쳐 낸 경우를 비교해 볼 때, 그 프로그램의 결과만을 가지고 표절행위를 판별할 수 없는 어려움이 있다[4]. 즉 표절은 그 불법적 행위의 과정을 말하는 것이므로 그 결과물이 표절행위를 증명해주는 것이 아니다. 이 때문에 본 연구의 방법으로 표절을 탐색하는 것은 표절의 가능성이 있는 프로그램들의 집합을 1차 선별해주는 것이지 그 결과가 바로 표절된 결과임을 증명해주는 것이 아니다. 본 논문이 제시한 방법론의 주된 기여는, 수작업을 통한 표절가능 프로그램 쌍을 찾아내는 작업을 보다 빠르게 수행할 수 있도록, 잘못된 판별(false-negative나 false-positive)의 수를 최소화하는 전처리 단계를 제공했다는 점이다.

표절 연구의 또 다른 어려움은 표절에 대해 구체적이고 정량적인 평가를 수행해야 한다는 점이다. 다시 말해서 프로그램 A가 다른 프로그램 B를 표절했다고 한다면 어떤 부분을 어떻게 표절했는지를 지적해야 한다. 단순한 구문수준(syntax level)에서의 표절은 쉽게 판별할 수 있지만, 내부 자료구조의 표절, 알고리즘의 표절을 정량화하여 밝혀내기란 매우 어려운 작업이다. 특히 자료구조나 알고리즘의 표절은 해당 분야의 전문가(human expert)만이 할 수 있는 고도의 지능적 작업이므로 이를 컴퓨터 시스템이 기계적으로 측정하여 판단하는 것은 매우 어렵다. 따라서 본 논문에서는 두 프로그램의 유사도를 일단 구문수준에서 시작하여 좀 더 확장된 형태로 고려할 수 있는 연구에 집중하고자 한다.

표절연구에서의 마지막 어려움은 실험자료로 쓰일 실제적인 표절자료를 확보하는 것이다. 결국 개발된 시스템이 잘 동작하는지를 증명하기 위해서는 실제적인 상황에서 제대로 동작함을 보여야 한다. 하지만 표절행위는 일반적으로 불법적인 것으로 인식되어 있기 때문에 이를 정상적인 방법으로 구하기란 매우 힘들다. 한편 인위적으로 어떤 프로그램을 주고 표절한 프로그램을 양산할 수도 있지만 이는 실제상황에서 만들어진 것이 아니므로 정확한 의미에서 표절데이터로 보기는 힘들 것이다.

본 연구에는 위에서 제시한 여러 문제에도 불구하고 현재까지 개발된 기존의 표절 탐색보다는 진일보된 새로운 방법을 제시하고 그 방법이 보다 타당함을 다양한 데이터와 실험의 특성값을 비교함으로 보이고자 한다. 그리고 지금까지 수집된 몇 가지 표절 데이터를 이용하여 우리가 개발한 방법이 기존의 방법보다 더 우수함을 보인다. 본 논문에서는 특정 프로그램 집합으로 제한된 환경에서의 표절 검사에 초점을 맞추고 있다. 여기서 제한된 환경에서의 프로그램들이란, 입력과 출력 형식, 동작되어야 하는 프로그램의 기능이 매우 엄격하게 정의된 프로그래밍 과제물이라든지, 프로그래밍 경진대회에서 제출된 문제들에 대한 프로그램들을 말한다. 본 논문에서는 이러한 제한된 환경에서의 프로그램 집합에 대해서, 본 논문에서 제시하는 방법의 우수성을 보이고자 한다.

2. 관련연구

우리와 언어체계가 다른 영어권에서는 이미 오래전부터 일반 언어에 대한 표절연구가 진행되어 왔다. 알파벳에 기반한 영문은 교착어인 한글에 비해서 그 단어의 위치정보가 한정되어 있으므로 표절탐색이 비교적 용이하므로 문장의 단어 순서를 스트링 비교와 혼합한 연구가 많이 있다. 최근 들어서는 인공언어의 일종으로서 프로그래밍 언어의 표절에 대한 연구가 상당히 진행되어 왔다. Clough는 일반적인 영문서와 프로그래밍 언어의 표절에 관한 최근까지의 방법과 도구들을 잘 비교하여 정리하였다[5]. James, McInnis, Devline은 최근 연구를 통해 각 도구와 방법들의 효율성을 비교하여 제시한 바 있다[6].

국내에서 관련 연구는 주로 문서들의 군집화(clustering)이나 정보 조회(information retrieval)에 관한 것이었다. 표절에 관한 연구는 비교적 최근의 일인데 그 중 대표적인 것은 생물학에서 사용되는 유전자 정렬기법(genome alignment technique)[7]을 원용하여 프로그램의 유사도를 비교한 논문이 있다[1]. 그리고 실제 국내외에서 연구된 표절탐색 방법을 조사한 자료조사 논문이 있다[8]. 그리고 공공기관에서 일반적인 프로그램의 표절판정을 위한 도구로서 두 프로그램간의 유사한 코드끼리 상호 비교해주는 도구 exEYEs가 프로그램심의조정위원회에서 개발되어 무료로 제공되고 있다. 공개판을 받을 수 있는 사이트는 www.pdmc.or.kr 이다.

지금까지 알려진 프로그램 소스의 표절검사 방법은 크게 특정한 키워드가 프로그램에 나타난 횟수를 주요 특성변수로 파악하여 이들의 형태를 비교하는 특성변수 계산법(attribute counting)과 프로그램의 파스트리(parse tree)와 같이 프로그램의 특정한 구조를 분석하여 그것을 비교하는 구조기반 비교방법이 있다[9]. 특성변수 계산법은 일반적인 문서의 유사도나 군집화에 보편적으로 쓰이는 방식으로 각 문서의 주요 키워드의 빈도수를 정규화한 벡터로 만들어 이들 간의 상관계수를 계산하여 그로부터 두 문서의 유사도를 추출하는 방식이다. 이 방식은 대용량의 문서를 거시적인 관점에서 군집화하는 데에는 유리하지만 표절탐색과 같이 미시적인 부분을 탐색하는 데에는 적합치 않은 방법이다. 특히 프로그램과 같이 구조화된 문서는 그 특성을 고려한 비교만이 구체적인 표절부분을 적시할 수 있어 유용하다.

본 논문은 이중에서도 특정한 제한상황(controlled)에서의 프로그램 소스에 대해서만 고려한다. 본 논문에서 말하는 제한된 소스(controlled source)란 동일 입력 데이터에 대하여 같은 결과 또는 올바른 결과(correct answer)가 나오는 프로그램을 말한다. 주로 프로그래밍 관련 교과목에서 특정한 과제물에 대한 결과물로 제출된 프로그램 집합이나, 프로그래밍 경시대회에서 특정한 문제에 대하여 제출된 문제를 제한된 소스라고 부른다. 프로그래밍 과제를 자동으로 채점하고 이들 간의 표절여부에 대한 연구는 영미권에서 오래전부터 있어 왔다[3,10]. 프로그램의 표절탐색을 위한 특성변수 계산법은 소프트웨어 메트릭(software metric)이라는 수치값을 이용하는 방법으로서 매우 간단한 방법이지만 부분 표절을 검출할 수 없다는 그 한계로 인하여 최근에는 거의 모두 구조 기반의 탐색법이 사용되고 있다[9].

구조기반 비교에 의한 프로그램 소스비교는 크게 3단계로 구성되어있다. 1단계는 프로그램 소스에서 주요 객체(keywords)등을 추출하여 비교가 용이하도록 새로운 객체를 만드는 작업이다. 이 작업에는 프로그램 소스 전체를 하나의 문자 스트링으로 보고 특별한 작업없이 그대로 이용하는 방법[9]과 전체 프로그램을 프로그램 파스트리로 재배열하여 그렇게 배열된 트리의 순회법(traversal)을 이용하여 어떤 선형의 순서를 만들어 내는 방법[11], 프로그램의 구문단계에서 대략적으로 수행하여 그 수행되는 함수들의 순서에 따라 주요 키워드를 추출하여 새롭게 정렬하는 방법이 있다[1].

프로그램 소스 상에서 각 함수들의 위치는 큰 의미가 없으므로 이들을 하나의 스트링으로 보고 스트링을 비교하는 기존 알고리즘을 사용하는 것은 표절 공격에 취약할 수 있다. 물론 이런 공격에 대비하기 위하여 일치되는 부분 블록(substring)들을 차례대로 지워나가는 Greedy-String-Tiling 방법이 흔히 사용되고 있다. 지금까지 알려진 가장 안정적인 도구인 JPlag과 YAP계열 모두 Greedy-String-Tiling 방법에 기초하고 있다[12].

이와는 전혀 다른 방법으로 문서의 세부구조를 고려하지 않고 문서나 프로그램의 Kolmogorov 복잡도와 정보이론(information theory)에 기초한 방법으로 전체 문서의 유사도를 비교하는 방법이 있다[13]. 주어진 스트링의 Kolmogorov 복잡도는 해당 스트링을 출력할 수 있는 최소크기의 프로그램(추상적 의미의)의 크기로 정의된다. 하지만 실제 이 프로그램의 크기를 구현하는 것은 불가능하므로 일반적으로는 해당 스트링을 RSA방법으로 압축했을 때의 크기를 대신 사용한다. 두 문서 A와 B가 유사할 경우 A와 B를 합쳐서(concatenation) 구성된 A·B 파일을 압축한 크기와 A혹은 B만을 압축한 크기를 비교하면 두 문서의 유사도를 알 수 있다. RSA 알고리즘은 파일의 앞부분을 이용한 동적 사전을 만들어 점진적으로 압축을 하는 방법을 사용하므로 A와 B가 유사할수록 뒤에 붙어있는 B를 압축하는 효율은 좋아진다.

이 방식은 이론적으로 볼 때 표절행위자들이 가장 공격하기 어려운 탐색방법이지만 한 가지 단점은 부분적인 표절에 대해서는 큰 도움을 주지 못한다는 점이다. 다시 말해서 주어진 두 문서가 정보이론적으로 유사한지를 밝히는 데에는 도움을 주지만 어떤 일부분이 서로 일치하는지에 대해서는 아무런 정보를 주지 못한다. 그러나 교묘한 표절은 세부적인 부분을 여러 군데에서 무단으로 도용하는 방법을 사용하므로 이 방법은 그 원취지와 안정성에도 불구하고 실제 사용하기에는 어려움이 많다. 이 방법이 유용한 경우는 관련연구자들도 지적한 바와 같이 초대규모의 전유전체(whole genome)의 상동성(homology)을 계산하여 진화적 거리(evolutionary distance)를 연구하는 데는 도움을 줄 수는 있지만 이에 비해서 그 크기가 작은 200 줄 안팎의 과제물 프로그램의 표절여부를 판단하는 데는 적절하지 못하다.

지금까지 개발된 프로그램 표절탐색 방법론을 살펴볼 때 더 개선되어야 할 사항은 다음과 같다. 먼저 프로그램의 수가 대규모가 될 때를 대비하여 표절 탐색과정에 걸리는 수행시간이 일단 빨라야 한다. 그리고 부분 표절이나 짜깁기를 통한 표절을 탐색할 수 있어야 한다. 또한 프로그래밍 언어의 특성을 악용한 여러 가지 표절 공격법에 유연하게 대처할 수 있어야 한다.

3. 적응적 유사도 행렬

3.1 프로그램 선형화

이 절에서는 지역정렬 기법에 의해 두 프로그램의 유사도를 적응적으로 측정하기 위해 필요한 이론적인 모델을 기술한다. 지역정렬 기법을 통해 두 프로그램의 유사도를 비교할 때에는 프로그램 선형화, 지역정렬, 유사도 산출 등의 과정을 거친다(그림 1 참고).

그림 1에서 첫 번째 단계인 토큰 추출은 두 프로그램에서 원하는 토큰을 추출하는 단계이다. 이 단계를 위해서는 우리가 살펴보고자 하는 토큰 집합을 정해야 한다. 토큰 집합은 프로그래밍 언어에 따라 다른데 통상 해당 프로그래밍 언어의 키워드 집합이 된다. 지역정렬 알고리즘에서 사용할 키워드 벡터를 K라고 하자. 지역정렬에서 고려할 키워드 개수를 r이라고 하면 키워드 벡터는 $K=\langle k_1, k_2, \ldots, k_r \rangle$로 정의할 수 있다.

프로그램 선형화에 사용되는 키워드 벡터 K는 프로그래밍 언어에 종속적이다. 이 논문에서 구현한 유사도 검출 프로그램은 C/C++ 언어를 대상으로 하고 있다. 현재 구현된 유사도 검출 프로그램에서는 C/C++ 언어 키워드 중 81개의 키워드를 선정하여 프로그램 선형화에 이용하고 있다($r=81$).

3.2 적응적 유사도 행렬

그림 1의 두 번째 단계인 지역정렬 단계에서는 유사도 행렬에 따라 두 토큰 스트림을 비교한다. 이 때, 유사도 행렬이 사용되는데 키워드 개수가 r일 때 유사도 행렬 M은 $(r+1)\times(r+1)$ 행렬이다. 행 수 및 열 수가 키워드 개수보다 하나 많은 이유는 갭(gap)을 나타내는 특수기호 k_{r+1}='_'이 포함되었기 때문이다. 갭 기호 '_'은 두 서열의 길이를 맞추어 주기 위해서 넣어 주는 특별한 기호이다. 유사도 행렬 M의 k_i행 k_j열의 원소 $M(k_i, k_j)$는 키워드 k_i, k_j의 일치 여부에 따른 점수(혹은 벌점)를 나타낸다.

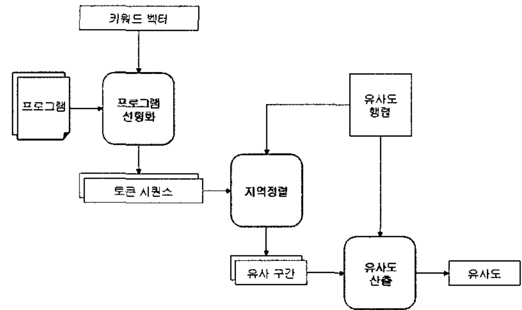


그림 1 지역 정렬 알고리즘에 의한 유사도 산출 과정

기존의 지역정렬 알고리즘에서는 유사도 행렬의 점수를 정수로 나타낸다. 키워드 k_i와 k_j가 일치할 경우에는 $M(k_i, k_j)$는 1, 일치하지 않으면 -1을 부여하고 있다. 갭을 이용하여 일치시켜야 하는 경우에는 -2점을 부여한다. 이렇게 정수 단위로 유사도 행렬을 정의하는 방법은 기존 생물학적 DNA 서열에 대해 유사도를 산출할 때 사용하던 방법이다.

이 절에서 설명할 알고리즘은 특정 프로그램 집합에 따라 유사도 행렬을 적응적으로 변경한다. 어떤 프로그램에 사용된 키워드 빈도는 키워드마다 서로 다르다. 키워드 빈도는 프로그램 집합에 대해서도 확장하여 생각할 수 있다. 같은 목적으로 작성된 프로그램 그룹에 대해서 키워드 빈도는 매우 다르게 나타날 수 있는데, 적응적 지역정렬 알고리즘에서는 키워드 빈도를 바탕으로 유사도 행렬을 결정한다.

적응적 지역정렬 알고리즘에서는 빈도가 높은 키워드의 일치에 대해서는 낮은 점수를 부여하고, 빈도가 낮은 키워드의 일치에 대해서는 높은 점수를 부여한다. 어떤 프로그램 A에서 사용 빈도가 매우 낮은 키워드 k를 사용하였다고 할 때, 다른 프로그램 B에 같은 키워드가 사용될 확률은 매우 낮다. 그러므로 이렇게 매우 낮은 빈도를 보이는 키워드를 함께 사용한 프로그램에 대해서는 상대적으로 높은 유사도가 있다고 볼 수 있다. 따라서 키워드 빈도에 따라 유사도 행렬의 점수를 변경하는 것이 자연스럽다.

적응적으로 유사도 행렬을 결정하기 위해서는 먼저 키워드 빈도를 산출해야 한다. n개의 프로그램으로 구성된 프로그램 그룹 $\mathbb{P}=\{p_1,p_2,\ldots,p_n\}$가 있다고 할 때, 키워드 빈도 벡터 $f^{\mathbf{P}}=\langle f_1^{\mathbf{P}},f_2^{\mathbf{P}},\ldots,f_r^{\mathbf{P}}\rangle$는 각 키워드 k_i가 해당 프로그램 그룹 내의 모든 프로그램에서 사용된 횟수로부터 결정된다. 키워드 k_i가 프로그램 그룹 $\mathbb{P}$에서 사용된 횟수를 $occur(\mathbb{P},k_i)$라고 하면, k_i가 사용된 빈도 $f_i^{\mathbf{P}}$는 다음과 같이 정의한다.

$$f_i^{\mathbf{P}}=occur(\mathbb{P},k_i)/\sum_{j=1}^{r}occur(\mathbb{P},k_j)$$

정의에 따르면 임의의 키워드 k_i의 빈도 $f_i^{\mathbf{P}}$는 0이상 1 이하의 값$(0\le f_i^{\mathbf{P}}\le 1)$이며, 키워드 벡터의 모든 원소 합은 1이다$(\sum_{i=1}^{r}f_i^{\mathbf{P}}=1)$.

유사도 행렬은 키워드 빈도에 따라 적응적으로 정한다. 본 논문에서는 두 키워드의 빈도 곱의 로그 값으로 유사도 점수를 정했는데, 어떤 신호값(signal)의 세기(intensity)를 상대적으로 비교하기 위하여 그 값에 log를 취한 값(log odd)을 취하는 것은 공학에서 흔히 이용하는 방법이다. 예를 들어 마이크로어레이 실험에서 각 스팟별 신호의 강도를 상대적으로 비교하는 것 역시 그 상대값의 비율에 로그를 취한 것을 사용한다. 이에 대한 이론적인 배경은 없지만 이는 실험자들의 실험과정에서 확립된 경험적 방법이다. 다시 말해서 어떤 특정 신호 값이 p이고 다른 신호값이 q라면 두 신호의 상대적인 차이는 p/q가 아니라 $\log(p/q)$로 계산한다.

실제로 log를 취한 값을 사용할 것인지 아닌지는 실험에 따라서 달라져야 하겠지만 본 유사도 검출의 경우에는 log를 취한 값을 사용하는 것이 바람직하다고 판단하였다. 왜냐하면 키워드 빈도의 차이가 크므로 키워드의 상대적인 비율을 그대로 매칭 가중치에 반영하는 것은 바람직하지 못하기 때문이다. 예컨대, 빈번하게 나타나는 '='와 같은 키워드는 약 17%임에 비해서 잘 나타나지 않는 특별한 연산자들은 약 0.003%(대략 1000개 당 3개 정도)이다. 이 값의 비율을 직접적으로 비교하면 약 5000배에 해당하므로 이 값을 그대로 쓰는 것은 거의 의미가 없다고 할 수 있다. 따라서 본 연구에서는 그 출현빈도가 약 2배 차이가 날 때 실제 가중치는 1 정도 차이가 나는 것이 합리적이라 판단하여 $\log_2$로 계산하기로 했다.

이상에서 설명한 것과 같이 유사도 행렬의 유사도 값은 키워드 곱의 로그 값으로 정한다. 구체적으로 말해서, 키워드 k_i와 k_j가 일치할 때의 점수는 음의 로그로 유사도 점수를 나타내고, 불일치일 경우에는 양의 로그로 벌점을 나타낸다. 키워드 빈도는 0과 1사이의 수치이므로 결과적으로 일치할 경우에는 양의 값을, 불일치할 경우에는 음의 값을 갖게 된다. 프로그램 그룹 $\mathbb{P}$에 대한 유사도 행렬 $M^{\mathbf{P}}$를 정의하면 다음과 같다.

$$M^{\mathbf{P}}(k_i,k_j)=\begin{cases}-\alpha\log_2 f_i^{\mathbf{P}}f_j^{\mathbf{P}} & k_i=k_j\\ \beta\log_2 f_i^{\mathbf{P}}f_j^{\mathbf{P}} & k_i\ne k_j\\ 4\log_2 f_i^{\mathbf{P}} & k_j=_\\ 4\log_2 f_j^{\mathbf{P}} & k_i=_\end{cases}$$

여기서 α와 β의 합은 1로 정한다$(\alpha+\beta=1)$.

위와 같이 유사도 행렬을 결정하면 일치와 불일치에 대한 상대적 가중치를 매개변수 α와 β로 조정할 수 있다는 장점이 있다. 이들 가중치 값을 변경시킴에 따라 유사도의 분포가 어떻게 변화하는지는 4절에서 살펴보겠다.

3.3 프로그램간 유사도 계산

위에서 정의한 유사도 행렬 M은 두 프로그램에서 유사하다고 발견된 유사 구간의 유사도 점수를 산출하는데 사용된다. 두 프로그램의 토큰 시퀀스에 대해 유사한 구간(aligned subrange)이 발견되면, 위에서 정의한 유사도 행렬을 이용하여 유사도 점수를 산출한다. 두 프로그램 A와 B를 정렬하여 유사구간을 산출하는 함수를 $align$이라고 하고 이 함수가 유사구간 쌍 $a_1,a_2,\ldots,a_m$, $b_1,b_2,\ldots,b_m$에 대해 다음과 같이 키워드 쌍의 벡터 형태로 유사구간을 산출한다고 하자.

$$align(A,B)=\langle(a_1,b_1),(a_2,b_2),\ldots,(a_m,b_m)\rangle$$

이 때, 이 구간의 유사도 점수는 다음 함수로 정의된다.

$$SIM_{abs}(A,B)=\sum_{(a,b)\in align(A,B)}M^{\mathbf{P}}(a,b)$$

이는 정수 단위의 유사도 행렬을 이용하는 기존 방법에서의 유사도 산출 방법과 거의 동일하다. 기존 방법에서의 프로그램 유사도 점수 산출에 대해서는 참고문헌 [14]를 참고하기 바란다.

유사도 점수 SIM_{abs}를 0~100% 구간으로 정규화한 것을 유사도라고 한다. 앞서 기술한 유사도는 유사구간의 점수를 합한 것이기 때문에 프로그램 길이에 비례하는 경향을 보인다. 따라서 정확히 비교하기 위해서는 정규화 과정이 필요한데, 정규화된 유사도를 산출하는 방식은 두 가지가 있다. 일반적으로 사용되는 방식은 두 프로그램의 유사도 점수를 각 프로그램에 대하여 자신과 정렬시킨 경우, 즉 최대 유사도 점수가 나오게 되는 경우의 유사도 점수의 합으로 나누는 방식이다. 이 방식의 유사도 $SIM_{sum}(A,B)$의 산출식은 다음과 같다.

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

유사도 점수를 정규화하는 다른 방법으로서 $SIM_{abs}(A,A)$와 $SIM_{abs}(B,B)$ 중 작은 값으로 나누는 방식이 있다.

비교할 두 프로그램 중에서 어느 하나의 프로그램 길이가 매우 큰 경우 SIM_{sum}의 값은 매우 낮아질 수 있다. 길이가 매우 큰 프로그램(예컨대, A)에 대하여 자신과의 유사도 점수($SIM_{abs}(A, A)$)가 매우 크게 되기 때문이다. 작은 점수를 기준으로 정규화한 유사도를 본 논문에서는 $SIM_{min}(A, B)$로 정의한다.

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

다음 절에서는 이상에서 기술한 적응적 유사도와 기존 유사도의 분포 차이를 살펴본다. 특히 적응적 유사도에 대해서는 매개변수 α 값과 β 값을 변경시킴으로써 어떻게 분포가 변화하는지 살펴본다. α 값을 늘리면 일치될 경우의 유사도 배점이 높아진다. 또한 β 값이 감소하므로 일치하지 않았을 경우와 갭을 이용하여 일치하였을 경우의 유사도 벌점이 감소된다. 반대로 α 값을 감소시키면 유사도 배점은 감소하고 벌점은 증가한다.

4. 실험 및 평가

적응적 지역정렬 알고리즘의 효과를 실험해 보기 위해, 간단한 실험을 수행하였다. 고정적 유사도 행렬(fixed similarity matrix)을 채택하였을 경우와 적응적 유사도 행렬(adaptive similarity matrix)을 채택하였을 경우에 대하여 지역정렬 알고리즘을 수행하였다. 각 경우에 유사도 점수 분포와 유사도가 높은 구간을 비교하여 살펴보았다. 우리는 리눅스가 탑재된 Intel Pentium PC에서 유사도 검출 프로그램을 수행하여 유사도를 산출하였다. 유사도 검출 프로그램은 Java 1.5.0로 작성되었기 때문에 하드웨어 환경과 독립적으로 수행 가능하다. 컴파일 결과 얻어진 클래스파일은 Java HotSpot™ Client VM(build 1.5.0_06-b05, mixed mode, sharing)에서 수행하였다.

4.1 테스트 데이터

논문에서 제안한 적응적 유사도 방식의 효과를 살펴보기 위해서는 테스트 데이터 셋이 필요하다. 제안된 알고리즘은 적응적으로 유사도를 산출하므로 테스트 데이터 셋으로 사용될 프로그램 집합은 같은 목적으로 작성된 프로그램들로 구성되어야 한다. 그래서 이 논문에서는 과제 및 경시대회 문제에 대한 프로그램을 선정하였다.

테스트 데이터 프로그램으로는 부산대학교 2005년도 고급컴퓨터프로그래밍 교과에서 과제로 제출된 프로그램과 한국정보올림피아드(KOI: Korea Olympiad in Informatics) 2004년도 및 2006년도 예선 문제, ACM 국제 대학생 프로그래밍 경진대회(ICPC:International Collegiate Programming Contest) 2004년도 및 2005년도 아시아 지역 예선 문제에 대한 출전자 제출 프로그램을 사용하였다. 테스트 데이터 셋을 정리하면 표 1과 같다.

표 1에 선정한 프로그램 중 KOI 2006 프로그램은 일

표 1 실험에 사용된 테스트 프로그램 집합: 고급컴퓨터프로그래밍 2005(PRO05), KOI 2004(KOI04) 및 2006 (KOI06), ICPC 2004(ICPC04) 및 2005(ICPC05)

순번	프로그램 그룹	파일 개수	순서쌍 수	소스코드 라인 수			
				최대	최소	평균	표준편차
1	PRO5 1	55	1,485	809	156	298.93	117.01
2	PRO5 2	33	528	2,309	352	882.42	364.24
3	PRO5 3	44	946	1,414	308	912.68	245.23
4	KOI04 1	51	1,275	341	27	80.02	45.60
5	KOI04 2	51	1,275	364	39	163.69	72.02
6	KOI04 3	27	351	555	31	224.90	126.44
7	KOI06 1	57	1,596	175	47	84.79	29.18
8	KOI06 2	34	561	331	81	143.06	62.05
9	KOI06 3	46	1,035	148	22	60.89	26.91
10	KOI06 4	36	630	110	32	63.78	19.31
11	KOI06 5	37	666	208	60	108.32	31.99
12	ICPC04 1	48	1,128	260	29	87.79	39.37
13	ICPC04 2	22	231	139	47	89.18	22.84
14	ICPC04 3	35	595	89	30	55.74	15.57
15	ICPC05 1	153	11,628	144	21	44.46	15.85
16	ICPC05 2	109	5,886	139	24	65.44	22.86
17	ICPC05 3	38	703	188	28	61.47	29.27
18	ICPC05 4	44	946	91	23	45.25	14.28

정 점수 이상이 되는 프로그램만 선정하였다. KOI 대회 규정상 제출된 프로그램은 모두 접수하기 때문에 프로그램 길이가 매우 짧은 프로그램도 원래 프로그램 집합에는 포함되어 있었다. 따라서 테스트 데이터 셋의 신뢰도를 높이기 위해 일정 점수 이상의 프로그램만 선정하였다.

표 1에 수집된 테스트 데이터 프로그램 집합에 대하여 실제 표절 프로그램을 찾아내기 위해서 모든 데이터 셋에 대해 검사해 보았다. 과거 발표된 연구 결과에서는 인위적인 표절을 수행한 프로그램에 대해 표절 검사를 수행하였다. 그러나 이러한 인위적인 표절은 실제 현장에서 행해지고 있는 표절과는 차이가 있다. 따라서 우리는 실제 프로그래밍 경진대회나 실제 과제 제출 프로그램을 대상으로 하여 표절을 검사하였으며, 그 결과 ICPC 프로그램 그룹에서 표절로 의심되는 프로그램을 찾을 수 있었다. 개인 정보 보호를 위해 이 프로그램 그룹을 ICPC 그룹 A라고 지칭하겠다. ICPC 그룹 A에서 표절로 의심되는 코드의 유사 구간에 대해서는 4.3절에서 자세히 살펴보겠다.

4.2 프로그램간 유사도 분포

이 절에서는 앞서 설명한 데이터 셋에 대한 유사도 분포에 관해 기술한다. 여기서는 ICPC 2005 프로그램 그룹 중에서도 가장 순서쌍 수가 많은 프로그램 그룹 1에 대하여 유사도 분포를 조사하였다. 고정적 유사도 행렬을 채택하였을 경우와 적응적 유사도 행렬을 채택하였을 경우에 대해 유사도 분포를 조사하였는데, 적응적 유사도의 경우에는 α값을 0.4에서 0.7까지 변화시켜 가며 유사도 분포를 조사하였다. 유사도 분포를 표로 정리하면 표 2와 같다.

표 2에서 확인할 수 있는 바와 같이 고정적 유사도 행렬을 채택하였을 경우와 적응적 유사도 행렬을 채택하였을 경우에 유사도 절대값 SIM_{abs}분포는 상당히 다른 형태로 나타났다. 그 이유는 SIM_{abs}가 유사도 절대 수치 값을 반환하기 때문이다. 정규화된 유사도를 함께 비교하기 위해 SIM_{abs}와 SIM_{sum}, SIM_{min}의 분포를 그래프로 나타내면 그림 2와 같다. 그림 2(a)는 SIM_{abs}의 분포를, 그림 2(b)는 SIM_{sum}의 분포를, 그림 2(c)는 SIM_{min}의 분포를 그래프로 나타낸 것이다.

그림 2(a)에서 확인할 수 있는 바와 같이, 고정적 유사도 행렬을 채택하였을 경우와 적응적 유사도 행렬을 채택하였을 경우에 유사도 절대 점수, 즉 SIM_{abs}의 분포는 확연히 다르다. 그러나 정규화된 유사도의 경우에는 유사한 분포를 보임을 그림 2(b)와 그림 2(c)를 통해 확인할 수 있다. 한편, 적응적 유사도 행렬을 채택하였을 경우에는 매개변수 α, β에 따라 분포가 달라짐을 알 수

표 2 ICPC05 1 프로그램 그룹에 대한 고정적 유사도 행렬을 채택하였을 경우와 적응적 유사도 행렬을 채택하였을 경우($\alpha=0.4\sim0.7$)의 SIM_{abs}분포 표

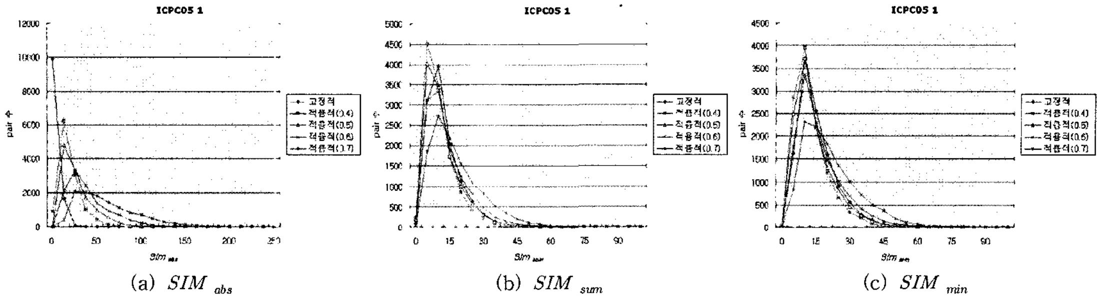
구간	고정적	적응적			
		α = 0.4	α = 0.5	α = 0.6	α = 0.7
0	9919	878	40	5	2
13	1648	6216	4785	2112	370
25	60	2990	3306	3285	2088
38	1	991	1672	2426	2016
50	0	424	942	1373	1806
63	0	94	536	932	1428
75	0	22	221	701	1114
88	0	12	71	375	811
100	0	0	28	207	706
113	0	1	20	110	473
125	0	0	5	46	291
138	0	0	1	23	204
150	0	0	1	19	139
163	0	0	0	8	87
175	0	0	0	4	38
188	0	0	0	1	23
200	0	0	0	1	15
213	0	0	0	0	7
225	0	0	0	0	4
238	0	0	0	0	3
250	0	0	0	0	3
총계	11628	11628	11628	11628	11628

있는데, $\alpha=\beta=0.5$일 경우에는 고정적 유사도 행렬을 채택한 경우와 유사한 분포를 보임을 알 수 있다. α 값이 증가됨에 따라 유사도 분포는 점차 완만한 곡선을 그리는 형태로 변화한다. α 값이 증가되면 유사도 점수는 높아지고 벌점은 작아지므로 유사도 값이 증가하여 보다 완만한 곡선을 그리게 된다.

여기서는 ICPC 2005의 그룹 1에 대해서만 기술하였지만, 다른 데이터 셋에 대해서도 유사도 분포를 조사하였다. 그러나 데이터 셋에 따라 분포가 달라지는 경우는 확인할 수 없었다. 다음 절에서는 ICPC 그룹 A에서 직접 표절로 검출된 코드에 대해서 집중적으로 살펴보고자 한다.

4.3 표절 의심 코드의 비교

이상의 실험 결과를 통해, 고정적 유사도 행렬을 이용할 경우와 적응적 유사도 행렬을 이용할 경우에 유사도 분포는 크게 다르지 않음을 알 수 있었다. 그러나 구체적으로 유사한 구간은 앞선 실험에서 확인할 수 없었는데, 여기서는 구체적인 유사구간의 차이를 살펴본다. 구체적인 유사구간은 각 토큰별 유사도 값에 따라서 결정되므로 유사도 행렬 값에 따라 달라질 수 있기 때문이다. 구체적인 프로그램에 대해서 유사도가 다르게 나타



(a) SIM_{abs} (b) SIM_{sum} (c) SIM_{min}

그림 2 ICPC05 1 프로그램 그룹에 대한 유사도 분포. 모든 경우에 대해 α가 커질수록 분포 곡선이 완만해 짐을 볼 수 있다.

```
 1 #include <stdio.h>
 2 int main ()
 3 {
 4     int t, n;
 5     int i, j;
 6     int p[500];
 7     int ans[500],max;
 8     scanf("%d", &t);
 9
10     while (t--)
11     {
12         scanf("%d", &n);
13         for (i=0; i<n; i++)
14         {
15             scanf("%d", &p[i]);
16             ans[i] = 0;
17         }
18         for (i=0; i<n; i++)
19             for (j=0; j<i; j++)
20                 if (p[i] >= p[j] && ans[i] < ans[j] + 1)
21                     ans[i] = ans[j] + 1;
22         max = 0;
23         for (i=0; i<n; i++)
24             if (ans[i] > max) max = ans[i];
25         printf("%d n", max + 1);
26     }
27     return 0;
28 }
```

그림 3 ICPC 그룹 A에서 표절로 의심되는 프로그램 P_a

```
 1 #include<stdio.h>
 2 int main()
 3 {
 4     int i, j;
 5     int data[600], cnt[600];
 6     int n, t;
 7     int max;
 8     scanf("%d", &n);
 9     while(n--)
10     {
11         scanf("%d", &t);
12         for( i=0 ; i<t ; i++ )
13         {
14             scanf("%d", &data[i]);
15             cnt[i] = 0;
16         }
17
18         for( i=0 ; i<t ; i++ )
19         {
20             for( j=i ; j>=0 ; j-- )
21             {
22                 if( data[i] >= data[j] && cnt[j]+1 > cnt[i] )
23                 {
24                     cnt[i] = cnt[j]+1;
25                 }
26             }
27         }
28         max = 0;
29         for( i=0 ; i<t ; i++ )
30         {
31             if( max < cnt[i] )
32                 max = cnt[i];
33         }
34         printf("%d n", max);
35     }
36     return 0;
37 }
```

그림 4 P_a의 표절로 의심되는 프로그램 P_b

나는 구간을 살펴보기 위해 표절로 의심되는 코드에 대해서 유사구간을 비교해 보았다.

그림 3 및 그림 4에 제시된 두 프로그램은 ICPC 그룹 A에서 표절로 의심되는 프로그램 쌍이다. ICPC 대회 지역본선은 보통 서울에서 열리는데 각 대학별로 선발된 2-3개 팀으로 이루어진 약 70-80 개의 팀으로 진행된다. 그런데 예선에 참가하는 300여 팀의 예선을 한 장소에서 할 수 없어 인터넷을 이용한 대회로 시험을 치른다. 인터넷 예선은 각 대학별 지도교수(ICPC 코치)의 감독하에 진행되거나 또는 그런 코치가 없는 경우에는 자율적으로 진행된다. 따라서 참가자가 마음만 먹으면 얼마든지 표절과 같이 부정행위가 가능하다.

ICPC 예선 대회가 끝난 후 본 연구자들이 개발한 초기버전으로 돌린 결과 유사하다고 판정된 프로그램의 쌍은 모두 10개가 추출되었다. 우리는 이 프로그램을 각각 하나씩 검사하여 실제 표절이 이루어졌는지 살펴보았다. 그 중 완전히 복사한 프로그램 한 쌍을 찾아내서 이 팀은 이후에 불이익을 받게 하였다. 조사 결과 시합 중 한 팀이 다른 지역의 팀에게 인터넷으로 해답 프로그램을 전송해 준 것으로 나타났다.

설명 편의를 위해 프로그램 P를 라인별로 나열했을 때 i번째 라인에서 j번째 라인(단, $i \le j$) 사이에 있는 코드를 $P[i,j]$로 표시하기로 하자. 위에서 제시된 그림 3과 그림 4에 나타낸 두 프로그램 P_a, P_b를 고정적 유사도 행렬로 비교했을 때 일치구간은 $P_a[1,19]$ 구간과 $P_b[1,20]$로 나왔다. 그런데 적응적 유사도 행렬로 비교했을 때에는 더 넓은 구간인 $P_a[1,28]$ 구간과 $P_b[1,37]$

과 일치한다고 보고가 되었다.

본 연구자들이 이 두 프로그램을 표절로 보는 이유는 변수 **n**과 **t**의미 때문이다. 이 문제의 테스트 데이터는 입력 파일에 모두 **t**개가 있으며 각각의 케이스마다 세부 데이터가 **n**개가 있다고 문제에서 설명을 했기 때문에 적어도 문제에서 제시된 변수를 바로 사용한다면 외부에 **t** 번의 루프(loop)가 있어야 하고, 각 케이스마다 **n**번의 **scanf**가 있어야 정상이다. 그런데 P_b에서는 이 의미가 거꾸로 사용되고 있다. 그리고 P_a의 19번째 라인의 **for** 루프에서 **j**의 제한 조건이 P_b에서는 그 역순(**j--**)로 표시되었다. 그런데 P_b에서는, 다른 **for** 루프는 모두 일반적인 형식인 **for**(X = **0**; X < n ; X++)으로 사용했음에도 불구하고 실제 그 순서에 아무런 의미가 없는 **j**에 대한 **for** 루프만을 다르게 표시했다는 것은 표절의 의심을 사기에 충분하다고 할 것이다.

우리는 본 논문에서 제시한 적응적 방법이 이 차이를 잘 극복해 주었다는 것을 지적하고자 한다. 즉 **for** 루프 조건을 바꾸는 구간이 다소 길긴 하지만 적응적 방법에서는 그 불일치(mismatch)값이 고정적인 불일치 값인 −1보다 작으므로 그 뒤의 매칭을 포함시킨 부분 정렬을 구성하였다. 통상 사용하지 않는 코드를 강제로 넣어서 표절을 하는 방법의 경우에 보통 표절자는 채점자의 눈의 크게 띄지 않는 코드를 넣게 된다. 왜냐하면 아주 특별한 기능이나 특이한 구문의 문장을 넣을 경우 바로 눈에 띄기 때문에 일반적인 지정문(assignment)이나 간단한 if-then-else등을 넣게 된다. 이 경우 고정적 방법에 비해서 그 구간의 매칭 값이 더 작아지기 때문에 적응적 방법에서는 그 구간 밖에 다른 매칭 구간이 있을 경우 그것을 포함하여 표절구간으로 계산하게 된다. 위의 예는 적응적 유사도 행렬이 고정적인 유사도 행렬보다 나은 점을 잘 보여주는 경우라고 할 수 있다.

우리는 ICPC 그룹 A의 프로그램을 대상으로 고정적 방식과 적응적 방식으로 표절 여부를 비교해보았다. 표 3에는 각 유사도 행렬로 비교했을 때 나타난 유사 프로그램의 쌍이 순서대로 정렬되어 있다. 표 3에서 볼 수 있듯이 상위 4개 정도의 쌍은 누가 보더라도 표절을 충분히 의심할 정도로 유사하였다. 그리고 그 유사도의 순서가 고정적과 적응적 모두에서 같게 나타났다. 따라서 이들 간의 비교는 하지 않았다. 우리는 이 중에서 9번, 10번, 63번 프로그램(각각 P_9,P_{10},P_{63}으로 표기)을 선택해서 고정적 방법과 적응적 방법의 결과를 비교해 보고자 한다. 즉 그 유사도의 순위가 각 방법에서 서로 다르게 나타난 (P_9, P_{10})쌍과 (P_9, P_{63})쌍을 서로 비교해 보고자 한다.

여기에서 자세히 살펴보아야 할 것은 잘 사용되지 않는 특이한 구문구조를 가진 프로그램 P_{10}과 비교한 프로그램들의 유사도 값이다. 일반적으로 두 변수, **x, y**의 값을 바꾸는 swap 동작은 별도의 임시 변수를 사용하여 **temp = x ; x = y ; y = temp ;**로 하는 것이 가장 보편적인 방법이다. 하지만 임시변수를 사용하지 않고 **x ^= y ^= x ^= y ;**와 같이 비트별 배타적 논리합(bitwise exclusive or) 동작을 연속으로 3번 사용해도 그 값을 바꿀 수 있다.

이러한 기법은 보통 일반적인 프로그램에 잘 쓰이지 않는 방법이다. 고정적 유사도 행렬을 이 차이를 단순한 차이로 계산하는 반면 적응적 방식에서는 이 구문의 빈도가 매우 낮으므로 만일 이 구문을 쓴 두 프로그램이 있다면 그 부분에서의 일치도 값을 다른 흔한 연산자와 비교해서 매우 높게 설정한다. 반대로 이런 구문과 일치하지 않을 경우(불일치(mismatch) 뿐만 아니라 갭을 이용한 일치(gap matching)도 포함)에는 그에 따른 벌점

표 3 ICPC 그룹 A 프로그램의 유사도 순위

순위	고정적 유사도			적응적 유사도 SIM_{sum}		
	A	B	유사도	A	B	유사도
1	P38	P43	72.92	P38	P43	72.54
2	P9	P70	72.53	P9	P70	71.93
3	P9	P81	71.19	P9	P81	70.69
4	P70	P81	68.51	P70	P81	69.34
5	P42	P81	51.46	P63	P70	54.82
6	P63	P70	50.00	P9	P139	49.47
7	P9	P139	50.00	P42	P81	47.21
8	P9	P10	46.07	P63	P81	46.95
9	P10	P81	45.26	P9	P63	44.33
10	P81	P139	44.81	P43	P101	42.76
11	P63	P81	43.72	P81	P139	40.69
12	P9	P42	42.51	P43	P138	39.66
13	P9	P63	42.39	P9	P10	39.66
14	P42	P70	40.76	P10	P81	38.89
15	P10	P70	40.00	P9	P42	38.56
16	P43	P101	39.08	P42	P70	38.07
17	P43	P138	37.89	P10	P70	34.88
18	P70	P139	35.11	P70	P139	32.01
19	P9	P29	32.38	P101	P138	29.86
20	P10	P42	30.91	P9	P29	29.31
21	P81	P156	30.00	P38	P128	29.25
22	P29	P81	29.67	P29	P81	28.56
23	P93	P127	29.46	P81	P156	28.09
24	P73	P93	29.31	P120	P138	27.98
25	P70	P156	29.19	P70	P156	27.63
26	P29	P70	28.97	P112	P138	27.23
27	P117	P127	28.15	P101	P120	26.98
28	P38	P127	27.64	P38	P127	26.77
29	P139	P42	27.23	P117	P127	26.69
30	P101	P138	26.67	P29	P70	26.63

을 매우 높게 계산하게 된다. 이것은 불일치 또는 일치만을 기준으로 동일한 값을 지정하는 방식에 비해서 분명히 합리적이라고 할 수 있다.

예를 들어, 고정적 방식에서 보면 $P_9[13,37]$과 $P_{10}[20,42]$가 서로 유사한 구간이라고 보고한데 비해서 적응적 방식에서는 $P_9[13,30]$과 $P_{10}[20,35]$ 구간이 유사하다고 보고하였다.

적응적 방식에서 더 짧은 구간을 보고한 이유는 앞서 말한대로 비트별 배타적 논리합 연산자의 사용 빈도가 매우 적어서 그것을 포함한 P_{10}의 36행에서 단절되었기 때문이다. 따라서 적응적 방식이 좀 더 정확하게 유사구간을 찾아내었다고 볼 수 있다. 이에 비해서 P_9와 P_{63}의 경우 적응적 방식으로는 $P_9[11,42]$와 $P_{63}[17,50]$을 유사하다고 보고하였다. 실제 두 프로그램은 P_{63}의 라인 19, 24, 27, 36, 38, 45가 P_9와 다르다. 그러나 적응적 방식에서는 위 6개의 라인에 나타난 문장인 가장 흔하게 나타나는 지정문이므로 그것이 삽입되어 발생하는 벌점(갭을 이용한 일치)이 비교적 적다. 따라서 그 다른

```
 1 #include <stdio.h>
 2
 3 int main( void )
 4 {
 5     int T, N, K, CN[10000];
 6     int i, j, k, tmp, ncost, cost[10000];
 7
 8     scanf("%d", &T);
 9     while(T--) {
10         mcost = 0;
11         scanf("%d", &N);
12         scanf("%d", &K);
13         for(i = 0; i < N; i++) {
14             scanf("%d", &CN[i]);
15         }
16         for(i = 0; i < N - 1; i++) {
17             for(j = i + 1; j < N; j++) {
18                 if(CN[i] > CN[j]) {
19                     tmp = CN[i];
20                     CN[i] = CN[j];
21                     CN[j] = tmp;
22                 }
23             }
24         }
25         for(i = 0; i < N - 1; i++) {
26             cost[i] = CN[i + 1] - CN[i];
27         }
28         for(i = 0; i < N - 2; i++) {
29             for(j = i + 1; j < N - 1; j++) {
30                 if(cost[i] > cost[j]) {
31                     tmp = cost[i];
32                     cost[i] = cost[j];
33                     cost[j] = tmp;
34                 }
35             }
36         }
37         for(i = 0; i < (N - 1) - (K - 1); i++)
38             mcost = mcost + cost[i];
39         printf("%d n", mcost);
40     }
41     return 0;
42 }
```

그림 5 ICPC 그룹 A에서 유사도가 높게 판별된 프로그램 P_9

```
 4 void main()
 5 {
 6     long i,j,k;
 7     long total_count;
 8     long *p;
 9     long *pp;
10
11     cin>>total_count;
12     for(i=0;i<total_count;i++) {
13         long count;
14         long sensor;
15
16         cin>>count;
17         cin>>sensor;
18         p= new long[count];
19         pp=new long[count-1];
20         for(j=0;j<count;j++) {
21             cin>>p[j];
22         }
23         for(j=0;j<count-1;j++) {
24             for(k=j+1;k<count;k++) {
25                 if(p[j]>p[k]) {
26                     p[j]^=p[k]^=p[j]^=p[k];
27                 }
28             }
29         }
30         for(j=0;j<count-1;j++) {
31             pp[j]=p[j+1]-p[j];
32         }
33         for(j=0;j<count-2;j++) {
34             for(k=j+1;k<count-1;k++) {
35                 if(pp[j]>pp[k]) {
36                     pp[j]^=pp[k]^=pp[j]^=pp[k];
37                 }
38             }
39         }
40
41         long result=0;
42         long idx=count-2;
43         sensor--;
44         while(1) {
45             if(sensor<=0) break;
46             result+=pp[idx];
47             idx--;
48             sensor--;
49         }
50         long py;
51         py=(p[count-1]-p[0])-result;
52         if(py<0) py=0;
53         cout<< py <<endl;
54     }
55 }
```

그림 6 고정적 유사도 비교 결과 P_9와 유사하다고 판별된 프로그램 P_{10}

```
 1 #include <stdio.h>
 2
 3 int main(void)
 4 {
 5     int T, t;
 6     int N, n;
 7     int i, j;
 8     int min;
 9     int K;
10     int data[10000];
11     int minus[10000];
12     int temp;
13     int sum;
14
15     scanf("%d", &T);
16     for(t = 0; t < T; t++) {
17         scanf("%d", &N);
18         scanf("%d", &K);
19         sum = 0;
20         for(n = 0; n < N; n++) {
21             scanf("%d", &data[n]);
22         }
23         for (i = 0; i < N-1; i++) {
24             min = i;
25             for (j = i+1; j < N; j++)
26                 if (data[j] < data[min])
27                     min = j;
28             temp = data[min];
29             data[min] = data[i];
30             data[i] = temp;
31         }
32         for (i = 0; i < N-1; i++) {
33             minus[i] = data[i+1] - data[i];
34         }
35         for (i = 0; i < N-2; i++) {
36             min = i;
37             for (j = i+1; j < N-1; j++)
38                 if (minus[j] < minus[min])
39                     min = j;
40             temp = minus[min];
41             minus[min] = minus[i];
42             minus[i] = temp;
43         }
44         for(i = 0; i < N-1-K+1; i++) {
45             sum += minus[i];
46         }
47         printf("%d n", sum);
48     }
49     return 0;
50 }
```

그림 7 적응적 유사도 비교 결과 P_9와 유사하다고 판별된 프로그램 P_{63}

라인 뒤에 좀 더 유사한 코드가 첨가되어 있으면 지역정렬의 특성대로 전체 값을 최대화하기 위하여 이 라인들을 포함하는 구간을 답으로 출력한다.

정리하자면 어떤 프로그램을 표절할 때 가장 일반적인 변수 치환, 문장 늘이기 등의 방법을 사용하는 경우 적응적 방식에서는 그 상대적의 빈도가 높기 때문에 이들의 삽입으로 인한 지역정렬에서의 음수값(벌점)은 상대적으로 작아진다. 따라서 이러한 가장 일반적인 표절 공격에 대해서 적응적 방식은 고정적 방식에 비해서 훨씬 유리함을 알 수 있다. 물론 아주 잘 쓰이지 않는 코드를 프로그램의 기능과 관계없이 일부로 넣어서 적응적 탐색을 방해할 수도 있지만 이런 경우는 따로 잘 사용되는 않는 키워드를 사용한 프로그램의 소스를 검사자가 관찰하기 때문에 비교적 쉽게 막을 수 있다. $P_9[11,42]$와 $P_{63}[17,50]$이 바로 이러한 경우이다.

P_9와 P_{63}의 경우에는 고정적 방법에 의한 유사구간과 적응적 방법에 의한 유사구간이 같게 나왔다. 그러나 고정적 방법에서는 앞서 기술한 불일치로 인한 벌점이 높게 산정되었기 때문에 P_9, P_{63}의 유사도가 P_9, P_{10}의 유사도보다 낮게 평가되었다.

또한, 프로그램에 특정한 구문구조를 반드시 사용하도록 강제할 경우에 적응적 탐색 방법을 매우 적절하게 이용할 수 있다. 구체적으로 말해서, 어떤 프로그래밍 과제에서 특정한 코드를 반드시 포함하여 작성하도록 할 경우가 바로 이런 경우이다. 만일 특정한 소스코드를 반드시 내부에 포함하여 구성하도록 할 경우에 고정적 방법을 사용하게 되면 모든 프로그램들의 쌍에 일치하는 부분이 생긴다. 그런데 다른 표절한 부분이 이 강제된 필수 코드보다 작은 경우라면 제대로 찾아낼 수 없게 된다. 그러나 적응적 방법을 사용하는 경우에는 모든 프로그램에 포함된 키워드들의 가중치는 상대적으로 낮기 때문에 고정적 유사도 행렬보다는 더 나은 결과를 돌려줄 수 있다.

5. 결 론

본 논문의 의미는 다음과 같이 크게 두 가지로 요약할 수 있다. 첫째, 키워드 빈도를 특성 변수로 이용하여 적응적으로 유사도를 산출하는 방법을 제안하였다. 이를 위해, 특정 목적으로 작성된 프로그램 그룹에 대하여 키워드 빈도를 산출하였고 이들 빈도의 로그를 취하여 적응적 유사도 행렬을 정의하였다. 결과로 얻어진 유사도 행렬을 바탕으로 세 가지 유사도 $SIM_{abs}, SIM_{sum}, SIM_{min}$를 정의하였다.

둘째, 이전의 고정적 유사도 행렬에 비하여 적응적 유사도 행렬을 이용하는 것이 표절 탐색에 더 효과적임을 실험을 통해 보였다. 특히 일반적으로 잘 사용하지 않는 특별한 방법을 사용하거나 구문 형태를 고쳐서 그 형식이 드러나지 않게 고치는 일에 많은 비용이 든다고 가정할 때, 각 키워드의 비율의 비중을 고려한 적응적 유사도 행렬에 의한 표절 탐색 방법은 이전의 단순한 일치/불일치로만 계산하는 기존의 방법에 비해서 상당히

효과적이다. 본 논문에서는 실제 표절 데이터를 대상으로 한 실험을 통해 이를 보였다.

앞으로 더 연구해 볼 가치있는 문제는 다음과 같이 나열할 수 있다. 첫째, 적응적 유사도 행렬을 제어하는 두 매개변수 α와 β를 조절하여 가장 최적의 값을 실제 표절 프로그램, 또는 인위적으로 만든 표절 프로그램을 이용하여 찾는 문제도 흥미로운 문제가 된다. 이 경우 프로그램들 간의 유사도 점수는 대략적으로 왼쪽으로 치우친 모양의 Poisson 분포를 따르는 것으로 예상되지만 실제 그 분포에 대한 연구는 추후 더 정교하게 진행되어야 할 것이다. 따라서 실제 테스트용으로 사용될 수 있는 표절 프로그램을 대량으로 수집하여 집중적으로 그 특성을 알아보는 작업이 필요하므로 표절 프로그램을 수집하는 것이 필요하다.

둘째, 매개변수를 확장하는 것에 대한 연구이다. 현재 적응적 유사도 산출 방법에서는, 자유도가 1인 조절변수 $\alpha=1-\beta$를 이용해서 일치, 불일치, 갭을 이용한 일치를 조정하고 있다. 이를 개선하여 각각의 제어하는 3개의 독립된 매개변수를 사용할 수도 있는데, 이럴 경우 최적 값의 매개변수를 구하는 문제도 매우 중요한 문제이다. 이 문제는 앞서 언급한 분포에 대한 연구와도 깊은 관련이 있을 것으로 생각된다.

참 고 문 헌

[1] 조동욱, 소정, 김진용, 최병갑, 김선영, 김지영. 프로그램 표절 감정 툴에 대한 비교, 분석 및 개발 툴에 대한 방향제시. *제20회 한국정보처리학회 추계학술대회 논문집*, 10권, pp. 757-760, 2003.

[2] Brenda Cheang, Andy Kurnia, Andrew Lim, and Wee-Chong Oon. On automated grading of programming assignments in an academic institution. *Computer and Education*, 41:121-131, 2003.

[3] David Jackson. A software system for grading student computer programs. *Computer Education*, 27(3/4):171-180, 1996.

[4] Janet Garter. Collaboration or plagiarism: What happens when students work together. In *Proceeding of the 4th Annual SIGCSE/SIGCUE Conference on Innovation and Technology in Computer Science Education(ITICSE-99)*, volume 31 of SIGCSE Bulletin inroads, page 52-55, N.Y., June 27-July 1 1999. ACM Press.

[5] Paul Clough. Plagiarism in natural and programming languages: An overview of current tools and technologies. Technical report, University of Sheffield, Department of Computer Science, June 2000.

[6] R. James, C. McInnis, and M. Devlin. Plagiarism detection software: How effective is it?, 2002. Except from R. James, C. McInnis, and M. Devlin, *Assessing Learning in Australian Universities: Ideas, strategies and resources for quality in student assessment.* Centre for the Study of Higher Education, Australian Universities Teaching Committee, Melbourne, Australia, 2002. (http://www.cshe.unimelb.edu.au/assessinglearning/docs/PlagSoftware.pdf).

[7] Thomas Schmidt and Jens Stoye. Quadratic time algorithms for finding common intervals in two and more sequences. In *Proceedings of the 15th Annual Symposium on Combinatorial Pattern Matching(CPM 2004)*, volume 3109 of Lecture Notes in Computer Science, pages 347-358. Springer, 2004.

[8] 이효섭, 도경구. 프로그램 표절 검출 방법에 대한 조사. *한국정보과학회 한국컴퓨터종합학술대회 2005 논문집(B)*, 32권, pp. 916-917, 2005.

[9] Kristina L. Verco and Michael 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, July 1996.

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

[11] Jeong-Woo Soon, Seong-Bae Park, and Se-Young 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, August 2006.

[12] Lutz Prechelt, Guido Malpohl, and Michael Philippsen. Finding plagiarisms among a set of programs with JPlag. *Journal of Universal Computer Science*, 8(11):1016-1038, 2002.

[13] Xin Chen, Brent Francia, Ming Li, Brian McKinnon, and Amit Seker. Shared information and program plagiarism detection. *IEEE Transactions on Information Theory*, 50(7):1545-1551, 2004.

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

지 정 훈

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

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~현재 부산대학교 컴퓨터공학과 조교수. 관심분야는 프로그래밍언어 및 컴파일러, 함수형 언어, 그리드컴퓨팅, 소프트웨어 메트릭

조 환 규

정보과학회논문지 : 소프트웨어 및 응용 제 33 권 제 2 호 참조