논문 2023-4-16 http://dx.doi.org/10.29056/jsav.2023.12.16

PDG 기반의 소스 코드 유사도 검사 도구에 대한 연구

윤성철*, 김수현**, 이임영**†

A Study on PDG-based Source Code Similarity Checking Tools

SeongCheol Yoon*, Su-hyun Kim**, Im-Yeong Lee**†

요 약

최근 소프트웨어 개발 시 OSS를 사용해 개발 시간 및 비용을 줄인다. 하지만 OSS 라이선스를 준수하지 않고 사용할 경우 소송을 당하는 등 다양한 문제가 발생한다. 이러한 문제는 소프트웨어 산업의 성장을 저해할 수 있기에 OSS 라이선스를 검사하는 것이 중요하다. 이를 소스코드 유사도 검사를 통해 사용된 OSS를 검출할 수 있다. 하지만 텍스트 기반의 소스코드 유사도 검사는 간단한 소스코드 수정을 통해 유사도 검사를 회피할 수 있다. 이를 해결하고자 본 논문에서는 PDG 기반의 소스코드 유사도 검사 도구를 제안한다. 이를 통해 프로그램의 구조적 형태로 유사도를 비교한다. 텍스트 기반의 소스코드 유사도 검사 도구는 같은 소스코드를 검사했을 때의 유사도 평균 96.5%이로 측정되고 본 제안방식은 평균 100%로 보다 더욱 정확한 유사도 검사를 도출할 수 있다. 그 외 수정된 소스코드에 대해 텍스트 기반의 소스코드 유사도 검사보다 높은 정확도를 보인다. 이를 통해 소스코드 내 OSS를 정확히 파악할 수 있다.

Abstract

Modern software development uses OSS to reduce development time and costs. However, non-compliance with OSS licenses can lead to various problems, including lawsuits. These issues can hinder the growth of the software industry, so it's important to check OSS licenses. This can be done through source code similarity checking to detect the OSS used. However, simple source code modifications can circumvent text-based source code similarity checks. To solve this problem, this paper proposes a PDG-based source code similarity-checking tool that compares similarities in the structural form of programs. Text-based source code similarity-checking tools are measured to have an average similarity of 96.5% when checking the same source code. In comparison, the proposed method can achieve a more accurate similarity check with an average of 100%. In addition, it is more accurate than text-based source code similarity checks for modified source code. This allows you to identify OSS in your source code accurately.

한글키워드 : 유사도, 소스코드 유사도, 프로그램 종속성 그래프, 그래프, 그래프 유사도

keywords : Similarity, Source code similarity, PDG, Graph, Graph similarity

* 순천향대학교 소프트웨어융합학과
** 순천향대학교 컴퓨터소프트웨어공학과
† 교신저자: 이임영(email: imylee@sch.ac.kr)
접수일자: 2023.11.27. 심사완료: 2023.12.14.
게재확정: 2023.12.20.

1. 서 론

최근 소프트웨어는 많은 분야에서 사용되고

있다. IoT(Internet of Things)와 AI(Artificial Intelligence)에서도 소프트웨어가 사용되고 있고 일상적으로 사용하고 있는 스마트폰과 컴퓨터 또한 소프트웨어를 기반으로 작동하고 있다.

이와 같이 소프트웨어가 다양한 분야에서 활용됨에 따라 소프트웨어의 설계서인 소스코드 또한 매우 중요해졌다. 소스코드는 프로그램을 상세하기 기술한 문서이며 컴파일러에 입력 시 바로 프로그램을 만들 수 있을 정도로 중요한 문서이다. 이처럼 소스코드가 소프트웨어 산업에서 핵심적인 역할을 함에 따라 소스코드를 무단으로 복제, 수정, 사용하는 사례들이 발생한다[1].

대표적인 예로 현재 개발자들은 프로그램 개발 시 직접 모든 코드를 작성하는 것이 아닌 OSS(Open-Source Software)를 이용해 개발한다. 이때 OSS 라이선스를 준수하지 않고 무단으로 사용하는 사례가 다수 존재한다[2]. OSS는 저작권법에 보호를 받는 저작물이기에 라이선스를 준수하지 않고 사용 시 소송을 당할 수 있다. 또는 상업용 프로그램의 소스코드를 전부 공개하거나 이미 판매한 제품을 리콜하는 등의 치명적인 문제가 발생할 수 있다[3].

위와 같은 문제를 소스코드 유사도 검사를 이용해 탐지할 수 있다[4-6]. 소스코드 유사도 검사란 두 소스코드가 얼마나 유사한지를 측정하는 기술이다. 이를 통해 해당 소스코드가 타인의 소스코드인지, OSS를 사용했는지 등에 대한 여부를 확인해 OSS 무단 사용을 탐지할 수 있다.

현재 많은 소스코드 유사도 검사 도구는 텍스트 기반의 소스코드 유사도 검사 기법이다. 하지만 텍스트 기반의 소스코드 유사도 검사는 소스코드 순서 변경, 소스코드 구조 및 데이터 구조 변경 등 간단한 소스코드 수정을 통해 유사도 검사를 회피할 수 있다.

이러한 텍스트 기반의 소스코드 유사도 검사의 한계를 극복할 수 있는 그래프 기반의 소스코드 유사도 검사가 존재한다. 그래프 기반의 소스코드 유사도 검사는 소스코드를 그래프 형식으로 변환해 그래프화된 소스코드 간의 유사도를 측정하는 방식이다. 이 경우 주로 소스코드의 의미나 실행 순서, 구조적 특징 등을 고려하여 유사도 검사를 진행하게 된다.

본 논문에서는 텍스트 기반의 소스코드 유사도 검사에서 발생하는 문제를 해결하기 위해 PDG 기반의 소스코드 유사도 검사 도구를 제안한다. PDG 기반의 소스코드 유사도 검사를 통해 두 소스코드가 실제로 구성되어있는 종속 관계 등을 통해 유사도를 비교할 수 있고 이를 통해 기존 텍스트 기반의 소스코드 유사도 검사 방식에서 발생했던 한계를 극복할 수 있다.

2. 관련 연구

본 장에서는 소스코드 유사도 검사 기법에 대한 기존 연구들에 대해 기술한다. 소스코드 유사도 검사 기법은 텍스트 기반, 그래프 기반의 소스코드 유사도 검사로 구분된다.

2.1 텍스트 기반의 소스코드 유사도 검사

텍스트 기반의 소스코드 유사도 검사는 소스코드가 작성된 형태 그대로 검사를 진행한다. 텍스트 기반의 소스코드 유사도 검사는 간단하게 구현할 수 있고 정확한 표면적 검사를 진행할 수 있다는 장점이 존재한다.

하지만 소스코드를 구문적이나 의미적으로 비교할 수 없기에 이러한 변경이 이루어진 소스코드에 대해 정확하게 해당 소스코드의 유사도를 비교할 수 없다는 단점이 존재한다.

Pintér 기법은 가장 긴 공통 부분 시퀀스 기반 소스코드 유사도 기법을 제안했다[7]. 해당 제안 방식은 복잡한 전처리, 추가적인 추상화, 프로그

래밍 언어에 대한 종속성 없이 유사도 검사를 진행하는 것을 목표로 한다. 해당 논문에선 이를 가장 긴 공통 부분 수열에 기반한 유사도 검사를 통해 해결 해결했다. 해당 제안방식은 기존 전처리, 추상화 과정이 존재하는 텍스트 기반의 소스코드 유사도 검사 기법의 한계를 극복하며 정확도를 향상하는 방식을 제안했다.

Rizvee 기법은 소스코드 표절을 자동으로 탐지하는 강력하고 효율적인 텍스트 기반의 소스코드 유사도 검사 기법에 대해 제안했다[8]. 해당 제안방식은 프로그램을 코드 구조에 기반한 간단한 데이터 표현 형식으로 변환한 후 비교할 프로그램 간의 문자열 매칭을 수행해 최대한 많은 일치하는 문자열을 찾는다. 해당 제안방식은 두 프로그램 간에 발견된 유사성 수치를 나타내기 위해 가중치 점수를 사용하며 좋은 정밀도와 높은 재현율을 제공하는 방식을 제안했다.

2.2 그래프 기반의 소스코드 유사도 검사

그래프 기반의 소스코드 유사도 검사는 소스코드를 의미론적 그래프, 종속성 그래프 등 그래프로 변환해 유사도를 측정하는 방식이다. 그래프 기반의 소스코드는 프로그램의 의미, 실행 과정 등을 통해 유사도 비교를 진행하기에 텍스트 기반의 소스코드 유사도 검사에서 발생한 유사도 감소 문제를 해결할 수 있다.

그래프 기반의 소스코드 유사도 검사는 의미론 기반의 소스코드 유사도 검사, 추상 구문 트리 기반의 소스코드 유사도 검사, 소스코드 흐름도 기반의 소스코드 유사도 검사 기법이 있다.

2.2.1 추상 구문 트리 기반의 소스코드 유사도 검사

추상 구문 트리 기반 소스코드 유사도 검사는 소스코드를 추상 구문 트리로 변환하여 유사도 검사를 진행하는 방식이다. 추상 구문 트리란 프로그래밍 언어로 작성된 소스코드의 추상 구문 구조의 트리를 의미한다.

추상 구문 트리 기반의 소스코드 유사도 검사는 소스코드를 추상 구문 트리로 변환해 구조적 유사성 탐지에 높은 정확도를 보이는 장점이 존재한다. 하지만 구조적으로 유사하지만 다른 목적으로 작성된 코드를 유사한 코드로 식별하는 거짓 양성 오류가 발생하는 단점이 존재한다.

Zheng 기법은 추상 구문 트리 기반 소스코드 스니펫 유사도 검사 기법을 제안했다[9]. 기존 방식들은 다른 언어 간 문법 차이로 인한 비교의 어려움을 해결하기 위해 외부 API(Application Programming Interface) 혹은 외부 클래스 등을 이용해 해결해야 했다. 해당 제안방식은 이러한 문제를 추상 구문 트리 기반 유사도 검사 기법을 통해 해결하고자 이와 같은 기법을 제안했다.

2.2.2 흐름도 기반의 소스코드 유사도 검사

흐름도 기반 소스코드 유사도 검사는 두 소스코드를 흐름도(Flow Chart)로 변환해 흐름도 간의 유사도를 비교하는 방식이다. 소스코드를 흐름도로 변환한 후 그래프 유사도 알고리즘을 이용해 유사도를 측정한다.

흐름도 기반의 유사도 검사는 기존텍스트 기반 유사도 검사의 한계였던 의미론적, 구조적 변경을 탐지할 수 있다는 장점이 존재한다. 또한, 코드의 의미를 고려하기에 가짜 코드 등의 의미 없는 코드를 삽입한 경우를 탐지할 수 있다는 장점이 존재한다.

하지만 흐름도 기반의 유사도 검사는 의미론 기반의 소스코드 유사도 검사와 같이 흐름도를 만드는 과정과 그래프 유사도 비교 과정에서 텍스트 기반의 소스코드 유사도 검사보다 높은 계산 오버헤드가 발생한다는 단점이 존재한다. 또한, 구현이 어렵고 소스코드 흐름도 정규화 등의 복잡한 과정이 포함된다.

Zhang 기법은 프로세스 마이닝 기술을 이용한 흐름도 생성 기반의 소스코드 유사도 검사 기법을 제안했다[10]. 해당 제안방식은 소스코드를 전처리 과정을 통해 유사도 분석에 불필요한 정보를 제거하고 이 코드를 계측한 뒤 실행하여 계측 로그를 생성한다. 이후 로그에서 흐름도를 추출한 뒤 그래프 유사도 알고리즘을 통해 비교한다.

해당 제안방식은 기존 방식들에 비해 불투명 술부, 반복문 해체, 함수 인라인화 및 아웃라인화 등이 적용된 소스코드에 대해 정확한 유사도 측정을 진행할 수 있다.

2.3 MOSS(Measure Of Software Similarity)

MOSS는 현재 스탠포드 대학에서 웹 서비스 형식으로 제공하고 있는 무료 소스코드 정적 분석 도구이다[11]. MOSS는 텍스트 기반의 소스코드 유사도 검사 도구로써 Document Fingerprinting 기술을 이용해 모든 서브 문자열을 다른 문자열과 비교하는 방식으로 소스코드 유사도 검사를 진행한다.

문서 내의 의미 없는 구문을 제거한 후, 제거된 문서의 k-gram 해시를 생성한 뒤 생성된 해시 중 임의의 해시를 선택해 문서의 Fingerprint로 선택한다. 이후 선택된 Fingerprint와 일치하는 해시가 많은 쌍을 찾아 소스코드 유사도 검사를 진행한다[12]. 유사도 검사가 끝나면 HTML 페이지를 통해 사용자에게 소스코드 간의 유사도 검사 결과를 출력한다.

MOSS는 해시 함수를 이용함에 따라 처리 속도가 빠르고 가볍다는 장점이 존재한다. 하지만 MOSS의 경우 사용을 위해선 별도의 계정 생성 과정, uid 등록 과정 등의 추가적인 작업이 반드시 필요하고 공백, 주석 등 의미 없는 코드를 제거하는 전처리 과정이 반드시 필요하다.

2.4 PDG(Program Dependence Graph)

PDG는 프로그램의 각 작업에 대한 데이터 종속성 및 제어 종속성을 명시적으로 나타내기 위해 제안된 중간 프로그램 표현 방식이다. 기존 데이터 종속성 및 제어 종속성을 효율적으로 표현하기 위해 제안되었다. PDG는 데이터 및 제어 종속성을 명시적으로 나타내기에 효율적으로 이를 파악할 수 있다는 장점이 존재한다[13].

PDG를 이용해 소스코드 유사도 검사를 진행할 경우 코드의 구조적 유사성을 더욱 쉽게 파악할 수 있으며 코드 요소 간의 종속성을 고려해 유사도를 평가하여 실제 코드 또는 함수 간의 관계를 계층적으로 분석할 수 있다[14].

3. 제안방식

본 장에서는 본 논문에서 제안하는 PDG 기반의 소스코드 유사도 검사 기법에 대해 기술한다. 본 제안방식은 PDG 변화 단계, 그래프 변환 단계, 그래프 유사도 측정 단계 3단계로 이루어진다. 그림 1은 본 제안방식의 시나리오이다.

PDG 변환 단계는 사람이 이해할 수 있는 언어 형태로 작성된 소스코드를 그래프 형식 언어인 dot 파일 형식으로 변환하는 단계이다. dot 파일 형식은 그래프 시각화에 사용되는 파일 형식으로써 그래프 데이터 구조를 표현하고 시각적으로 표현하기 위해 그래프 시각화 도구를 이용해 시각화할 수 있는 파일이다.

dot 파일은 그래프 정의부가 존재하고 그래프 정의부 내에 노드 정의부, 엣지 정의부로 구성된다. 그림 2는 dot 파일 형식의 예시이다.

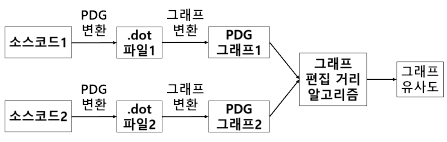

그림 1. 제안방식 시나리오
Fig. 1. proposed scheme scenario

```
digraph G {
        // 노드 정의
        A [label="Node A"];
        B [label="Node B"];
        C [label="Node C"];

        // 엣지 정의
        A -> B [label="Edge 1"];
        B -> C [label="Edge 2"];
        C -> A [label="Edge 3"];
}
```

그림 2. dot 파일 형식 예시
Fig. 2. dot file format example

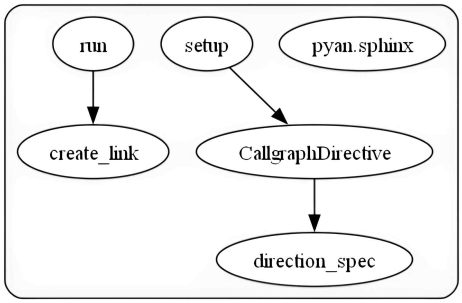

그림 3. PDG 예시
Fig. 3. PDG example

3.1 PDG 변환 단계

본 제안방식에서는 소스코드를 PDG로 변환하는 작업을 수행하기 위해서 종속성 그래프를 생성하는 OSS인 Pyan3를 이용해 PDG 변환을 구현했다. Pyan3는 소스코드를 입력받고 소스코드를 분석하여 소스코드의 종속성 그래프를 생성한다. 소스코드를 함수 별로 분리해 노드로 표현하고 각 함수 간 호출 여부를판단해 노드 간의 엣지로 표현한다.

dot 파일은 Pyan3 내의 클래스인 DoWriter 클래스를 통해 진행된다. 해당 클래스 내의 init 메소드를 통해 입력받은 그래프에 대한 DoWriter 클래스의 인스턴스를 생성한다. 이후 start_graph 메소드를 통해 주 그래프를 dot 파일 형식으로 변환하는 작업을 수행한다. 그다음으로 start_subgraph 메소드를 통해 주 그래프에 속해있는 서브 그래프를 표현한다.

이후 write_node 메소드를 통해 그래프의 노드를 작성한다. 그래프의 노드는 소스코드 내 각 함수로 할당되고 노드의 레이블은 함수명으로 사용된다. 노드가 생성되면 노드 간의 관계를 표현하기 위해 write_edge 메소드를 이용해 노드 사이의 엣지를 표현한다.

이러한 dot 파일을 이용해 소스코드 내 함수 간의 종속성을 표현한 PDG를 생성한다. dot 파일을 이용해 생성된 PDG는 그림 3과 같다.

이렇게 생성된 dot 파일 및 PDG를 이용해 그래프 변환 단계를 진행한다. 그래프 변환 단계는 dot 형식의 파일을 그래프 시각화 도구와 호환될 수 있는 그래프 형식으로 변환하는 단계이다.

그래프를 시각화함으로써 코드의 구조 및 종속성을 더욱 쉽게 이해할 수 있다.

3.2 그래프 변환 단계

그래프 변환 단계는 파이썬 라이브러리 NetworkX와 Pydot을 이용해 구현했다. dot 파일 두 개의 dot 파일을 입력받아 이를 각각 Pydot을 이용해 NetworkX와 호환되는 그래프 객체로 변환하는 과정을 진행한다. 과정은 다음과 같다.

dot 파일을 읽어 dot 형식으로 작성되어있는 문자열을 그래프로 파싱하기 위해 graph_definition 메소드를 이용해 graph_definition 클래스의 인스턴스를 생성하고 전역 변수를 초기화한 후 이를 저장한다. 이후 파싱 과정을 진행한다. 우선 graph_definition 메소드를 이용해 파서 객체를 생성한 뒤, parseWithTabs 메소드를 이용해 탭 문자를 설정한다. 그리고 graphparser.parseString 메소드를 이용해 dot 파일에서 읽은 dot 형식의 문자열을 파싱하고 만약 파싱에 성공했을 경우 파싱된 결과를 저장해 이후 그래프 변환에 사용한다.

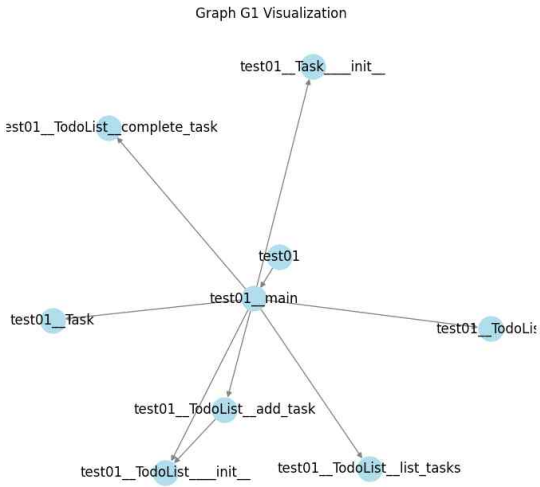

그림 4. 시각화된 NetworkX 그래프
Fig. 4. Visualized NetworkX graph

마지막으로 저장된 결과를 리스트 형식으로 반환해 dot 형식으로 작성된 그래프를 NetworkX와 호환되는 그래프로 변환한다.

변환이 완료된 후에는 그래프 간의 서브플롯 간격을 계산해 그래프를 시각화한다. 그림 4는 dot 파일을 그래프로 변환해 시각화한 그림이다.

3.3 그래프 유사도 측정 단계

그래프 변환 단계가 종료되면 마지막으로 두 그래프 간의 유사도를 측정하는 그래프 유사도 단계가 진행된다.

본 제안방식에서는 두 그래프 간의 유사도를 측정하기 위해 그래프 편집 거리 기반 그래프 유사도 검사 방식을 이용한다. 그래프 편집 거리 계산은 NetworkX 그래프 객체로 변환된 두 그래프를 이용하여 진행된다.

그래프 편집 거리 계산은 노드 매칭 과정, 엣지 매칭 과정, 편집 연산 정의, 최소 비용 계산, 최적 매칭 완성 과정을 통해 진행된다.

노드 매칭 과정은 노드 매칭을 위한 비용 행렬을 초기화한다. 여기서 비용 행렬은 $m(G1$의노드수$)*n(G2$의노드수$)$ 크기의 행렬이다. 이후 두 개의 미처리 노드 목록 pending_u와 pending_v를 사용해 비용 행렬을 계산한다. 비용 행렬 계산은 pending_u와 pending_v의 모든 가능한 노드 쌍에 대해 1 - int (node_match (G1.nodes[u], G2.nodes[v]))를 이용해 일치 여부를 계산한다. 만약 두 노드가 일치할 경우 비용은 0이 되고 일치하지 않을 경우 비용은 1이 된다. 위와 같은 작업을 모든 노드 쌍에 대해 수행한 뒤 최종적으로 모든 노드 쌍에 대한 비용 행렬을 획득하고 이를 편집 거리 계산에 사용한다.

엣지 매칭 과정은 노드 매칭 과정과 동일한 과정을 통해 진행된다.

편집 연산 정의는 사용자가 노드 추가, 노드 삭제, 노드 수정, 엣지 추가, 엣지 삭제, 엣지 수정 등의 작업을 수행할 때 편집 비용을 설정하는 단계이다. 이 또한 비용 행렬 계산 과정은 노드 매칭 과정과 동일하게 진행된다.

최소 비용 계산은 모든 가능한 노드 매칭과 편집 연산을 고려해 두 그래프를 동일하게 만드는 데 필요한 최소 비용을 계산하는 과정이다. 우선 계산된 행렬에서 각 행의 값들을 각 행의 최솟값만큼 뺀다. 그 후, 각 열에서 각 열의 최솟값을 제외한다.

최적 매칭 완성 과정은 최소 비용 계산 후에 행렬에서 0이 되는 값들을 탐색해 노드 매칭을 완성한다. 0이 되는 값의 행과 열에 해당하는 노드들이 서로 매칭되는 노드임을 나타낸다. 이렇게 완성된 최적 매칭된 노드를 기반으로 G1 그래프에서 G2 그래프로 변환하는 데 필요한 최소 연산을 계산한다.

그림 5의 경우 G1에서 G2로 그래프를 변환하기 위해선 노드 추가 1번, 엣지 추가 1번으로 총 2번의 연산이 필요하다. 이때 그래프 편집 거리는 총 연산 횟수인 2가 된다.

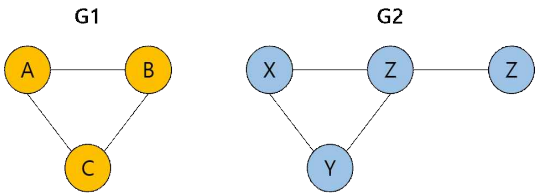

그림 5. 그래프 예시
Fig. 5. graph example

4. 제안방식 분석

표 1은 기존 MOSS와 본 제안방식의 비교분석표이다. MOSS는 구조적 변경을 탐지하지 못해 구조적 변경이 일어나면 유사도가 감소하지만 본 제안방식은 구조적 변경에 영향을 받지 않는다.

그림 6은 MOSS와 제안방식을 이용해 유사도 검사를 진행한 결과이다. 소스코드 유사도 검사는 기준이 되는 소스코드와 동일한 소스코드, 변수/함수명이 변경된 소스코드, 함수 및 소스코드의 순서를 변경한 소스코드, 변수/함수에 대해 병합/분할이 적용된 소스코드를 기반으로 실험을 진행했다. 각 실험군은 각기 다른 기준 소스코드와 수정이 진행된 소스코드이다.

기준 소스코드와 동일한 소스코드 간 유사도 비교를 진행했을 때의 결과이다. 실험 결과 MOSS는 평균 약 96.5%의 유사도를 측정했다. 반면 제안방식은 모두 100%의 유사도를 측정했다. 즉, 본 제안방식은 MOSS보다 더욱 정확하게 동일한 소스코드를 검출할 수 있다.

기준 소스코드와 변수/함수명을 수정한 소스코드 간 유사도 검사를 진행했을 때 이전과 동일한 결과가 도출되었다.

기준 소스코드와 함수와 소스코드의 순서를 바꾼 소스코드와의 유사도 비교를 진행한 경우 MOSS는 유사도에 큰 영향을 미치게 된다. 하지만 본 제안방식은 소스코드 간의 구조적 정보로 유사도를 파악하기에 순서가 변경된 소스코드 또한 정확하게 유사도 검사를 진행할 수 있다.

기준 소스코드와 함수/변수를 분할/병합한 소스코드와 유사도 검사를 진행했을 때, MOSS는 소스코드가 표면적으로 변하면 많은 영향을 받고 유사도에 큰 영향을 주게 된다. 결국 정상적으로 유사도 측정을 할 수 없다.

하지만 본 제안방식은 이러한 함수/변수 병합/분할 등의 변경이 이루어져도 모든 실험군에서 약 80% 이상의 유사도를 보임에 따라 기존 텍스트 기반의 소스코드 유사도 검사 기법보다 더욱 정확한 유사도 검사를 진행할 수 있다.

즉, 본 제안방식은 텍스트 기반의 소스코드 유사도 검사보다 OSS 식별, 변수/함수명을 변경한 OSS 식별, 소스코드 순서를 변경한 OSS 및 병합/분할을 한 OSS를 정확하게 식별할 수 있다.

5. 결 론

OSS를 이용해 개발할 경우 개발 시간 단축, 비용 단축 등 다양한 이점이 존재한다. 하지만 많은 개발자가 OSS의 사용 규정인 라이선스를 인식하지 못한 채로 사용하는 문제가 발생한다.

표 1. 소스코드 유사도 검사 도구 비교분석표
Table 1. source code similarity checking tools comparison chart

	함수/변수명 변경	구조적 변경	코드 병합/분할 변경	전처리 과정
MOSS	탐지 가능	탐지 불가능	탐지 불가능	필수
제안방식	탐지 가능	탐지 가능	탐지 가능	불필요

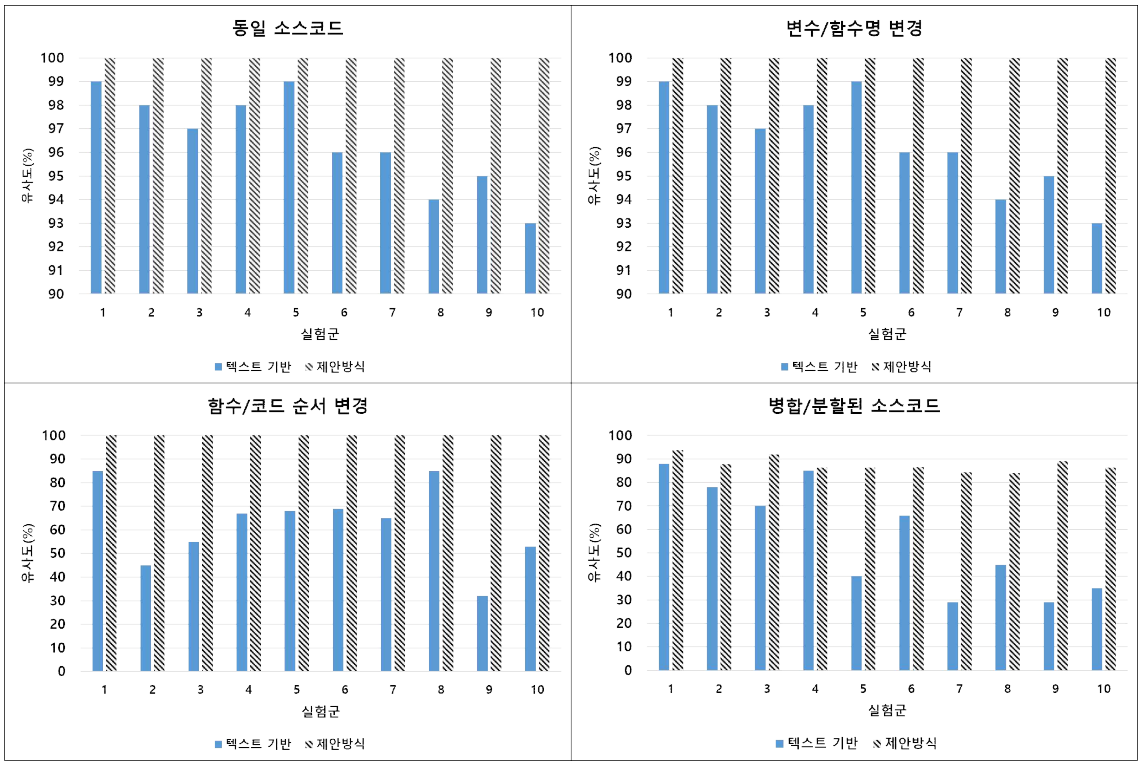

그림 6. 소스코드 유사도 검사 도구 비교 분석 결과
Fig. 6. Source code similarity checking tools comparison results

이러한 문제를 파악하기 위해 소스코드 내 OSS를 파악해야 한다. 이를 소스코드 유사도 검사를 이용해 OSS를 확인할 수 있다.

하지만 기존 텍스트 기반의 소스코드 유사도 검사는 소스코드의 순서를 변경하거나 데이터 및 소스코드 구조를 변경할 경우 유사도가 감소하는 문제가 발생한다. 이 문제를 PDG 기반의 소스코드 유사도 검사를 통해 만족할 수 있다.

PDG 기반의 소스코드 유사도 검사는 소스코드를 PDG로 변환하기에 소스코드를 구조적으로 표현할 수 있고 이를 통해 소스코드의 구조적 유사성을 파악할 수 있다. 또한, 그래프로 변환된 소스코드 간의 유사도 비교를 진행하기에 변수/함수명을 변경해도 유사도가 감소하지 않는다. 코드 순서를 변경한 소스코드, 병합/분할된 소스코드 등 텍스트 기반의 변경을 적용한 소스코드 또한 텍스트 기반의 소스코드 유사도 검사 기법보다 정확하게 유사도 검사를 진행할 수 있다.

본 논문에서는 PDG 기반 소스코드 유사도 검사 기법을 제안했다. 이를 통해 기존 텍스트 기반의 소스코드 유사도 검사에서 발생했던 문제를 해결할 수 있다. 동일한 소스코드 유사도 비교 시 텍스트 기반의 소스코드 유사도 검사 기법보다 높은 정확도를 보이며 텍스트 수정이 적용된 소스코드를 비교할 때 더욱 정확한 소스코드 유사도 검사를 진행할 수 있다.

또한, 데이터 종속성 및 제어 종속성을 고려하기에 프로그램의 간단한 동작 흐름을 파악할 수 있어 프로그램을 이해하는데 도움을 줄 수 있다.

본 제안방식은 동일한 소스코드를 정확히 파

악하므로 OSS를 이용해 임의의 소스코드와 소스코드 유사도 검사를 진행하여 변수/함수명을 변경하거나 소스코드 순서를 변경한 OSS를 정확히 확인할 수 있다.

하지만 본 제안방식 또한 병합/분할된 소스코드 검사 시 유사도가 감소한다. 이는 변수/함수 병합/분할 시 그래프 노드 수에 영향을 미치게 되고 소스코드의 구조가 변경되기 때문이다.

향후 제안방식을 확장해 난독화된 소스코드에 대한 유사도 검사, 그래프 편집거리의 단점을 보완할 수 있는 연구, 소스코드 유사도 검사 과정 중 민감한 소스코드가 권한이 없는 사용자 혹은 공격자에게 유출될 수 있기에 이에 대한 데이터 보호에 대한 연구가 필요할 것으로 사료된다.

본 연구는 문화체육관광부 및
한국콘텐츠진흥원의 2023년도 SW저작권
생태계 조성 기술개발 사업으로 수행되었음
(과제명 : 클라우드 서비스 활용 구축 형태별
대규모 소프트웨어 라이선스 검증 기술개발,
과제번호 : RS-2023-00224818, 기여율: 100%)

참 고 문 헌

[1] C. Alexandra-Cristina and O. Alexandru-Corneliu. (2022). Material Survey on Source Code Plagiarism Detection in Programming Courses. Proceedings of the 2022 International Conference on Advanced Learning Technologies (ICALT), 387-389. DOI: 10.1109/ICALT55010.2022.00119

[2] H. Schoettle. Open Source License Compliance-Why and How?. Computer, 52(9), 63-67. DOI: 10.1109/MC.2019.2915690

[3] J. Ji, P. Ma, S. Wang. (2022). Unlearnable Examples: Protecting Open-Source Software from Unauthorized Neural Code Learning. Proceedings of SEKE 2022. https://people.cs.pitt.edu/~chang/seke/seke22paper/paper066.pdf

[4] R. Maertens, P. Dawyndt, B. Mesuere. (2023). Dolos 2.0: Towards Seamless Source Code Plagiarism Detection in Online Learning Environments. Proceedings of the 2023 Conference on Innovation and Technology in Computer Science Education V. 2 (ITiCSE 2023), 632. DOI: 10.1145/3587103.3594166.

[5] J. Vedran. (2011, Sep). Evaluation of similarity metrics for programming code plagiarism detection method. Central European Conference on Information and Intelligent Systems, 83-88. https://www.proquest.com/conference-papers-proceedings/evaluation-similarity-metrics-programming-code/docview/1322999508/se-2?accountid=16991

[6] H. Cheers, Y. Lin, S. P. Smith. (2021) Academic Source Code Plagiarism Detection by Measuring Program Behavioral Similarity. IEEE Access, 9, 50391-50412. DOI: 10.1109/ACCESS.2021.3069367

[7] Á. Pintér, S. Szénási. (2023, May). Longest Common Subsequence-based Source Code Similarity. 2023 IEEE 17th International Symposium on Applied Computational Intelligence and Informatics (SACI) (pp. 123-129). NewYork:IEEE. DOI: 10.1109/SACI58269.2023.10158551

[8] R. A. Rizvee, M. F. Arefin, M. B. Abid. (2022, Oct). A Robust Objective Focused Algorithm to Detect Source Code Plagiarism. 2022 IEEE 13th Annual Ubiquitous Computing, Electronics & Mobile Communication Conference (UEMCON), 109-115. DOI: 10.1109/UEMCON54665.2022.9965688

[9] M. Zheng, X. Pan, D. Lillis. (2018, Dec).

CodEX: Source Code Plagiarism Detection Based on Abstract Syntax Trees. 26th AIAI Irish Conference on Artificial Intelligence and Cognitive Science(AICS 2018). https://www.researchgate.net/profile/Mengya-Zheng-2/publication/329513153_CodEX_Source_Code_Plagiarism_Detection_Based_on_Abstract_Syntax_Trees/links/5c0bf510299bf139c74993d9/CodEX-Source-Code-Plagiarism-Detection-Based-on-Abstract-Syntax-Trees.pdf

[10] F. Zhang, L. Li, C. Liu, Q. Zeng. (2020). Flow Chart Generation-Based Source Code Similarity Detection Using Process Mining. Hindawi. DOI: 10.1155/2020/8865413

[11] J. H. Park, Y. S. Choi, J. M. Choi. (2014). Software Similarity Analysis via Stack Usage Pattern. Journal of KIISE : Computing Practices and Letters, 20(6), 349-353. https://www.dbpia.co.kr/journal/articleDetail?nodeId=NODE02432570, Jun. 2014.

[12] Y. S. Nam, S. C. Kim, J. H. Kim, H. S. Kim, C. G. Ha, S. O. Choi & C. J. Yoo. (2023, Jun). Source Code Evaluation and Similarity Inspection System : Code Odor Detect Assistant. Proceedings of KIIT Conference, 1080-1084. https://www.dbpia.co.kr/journal/articleDetail?nodeId=NODE11485685, Jun. 2023.

[13] J. Ferrante, K. J. Ottenstein, and J. D. Warren. (1987). The program dependence graph and its use in optimization. ACM Transactions on Programming Languages and Systems, 9(3), 319-349. DOI: 10.1145/24039.24041

[14] K. J. Ottenstein, L. M. Ottenstein. (1984). The program dependence graph in a software development environment. ACM SIGPLAN Notices, 19(19), 177-184. DOI: 10.1145/390011.808263

저 자 소 개

윤성철(SeongCheol Yoon)

2023.2 순천향대학교 컴퓨터소프트웨어공학과 졸업
2023.3-현재 순천향대학교 소프트웨어융합학과 석사과정
<주관심분야> 정보보호, 암호학, 소스 코드 유사도

김 수 현(Su-Hyun Kim)

2010.2 순천향대학교 정보기술공학부 졸업
2012.2 순천향대학교 컴퓨터학과 석사
2016.2 순천향대학교 컴퓨터학과 박사
2016.3-2018.9 순천향대학교 IoT보안연구센터 연구교수
2018.10-2023.2 정보통신산업진흥원 책임
2023.3-현재 순천향대학교 컴퓨터소프트웨어공학과 교수
<주관심분야> 암호 프로토콜, IoT 보안, 클라우드 컴퓨팅 보안

이 임 영(Im-Yeong Lee)

1981.2 홍익대학교 전자공학과 졸업
1986.2 오사카대학 통신공학전공 석사
1989.2 오사카대학 통신공학전공 박사
1989-1994 한국전자통신연구원 선임연구원
1994-현재 순천향대학교 컴퓨터소프트웨어공학과 교수
<주관심분야> 암호이론, 암호 프로토콜, 컴퓨터보안