JKSCI

한국컴퓨터정보학회논문지
Journal of The Korea Society of Computer and Information
Vol. 30 No. 11, pp. 191-203, November 2025
https://doi.org/10.9708/jksci.2025.30.11.191

Design and Implementation of an LCS and Weight-Based Birthmark System for Detecting Python Code Theft

Seong-Min Kim*, Won-Chan Lee*, Heewan Park**

*Student, Department of IT Software, Halla University, Wonju, Korea
**Professor, Department of IT Software, Halla University, Wonju, Korea

[Abstract]

This paper proposes a novel method for accurately measuring the similarity between Python codes. Existing approaches primarily rely on k-gram based token matching, which lacks sufficient consideration of code structure and flow. To address these limitations, the proposed method introduces an s-gram segmentation technique and a weighted comparison scheme, combined with the Longest Common Subsequence (LCS) algorithm to analyze structural similarity. The approach was evaluated on various open-source Python projects collected from GitHub. Experimental results demonstrate that the proposed method effectively reduces similarity scores between codes developed by different authors while maintaining high similarity scores for codes from the same author, outperforming existing techniques. These findings suggest that the proposed method can contribute significantly to Python code theft detection and software quality management.

▸ **Key words**: Software birthmark, Code theft detection, Python bytecode, Bytecode similarity, Longest Common Subsequence (LCS)

[요 약]

본 논문에서는 파이썬 코드의 유사도를 보다 정확하게 측정하기 위한 새로운 기법을 제안한다. 기존 연구에서는 k-gram 기반의 단순 명령어 일치에 의존하여 코드 구조와 흐름을 충분히 반영하지 못하는 한계가 있었다. 이를 개선하기 위해 본 연구는 s-gram 분할 방식과 가중치 기반 비교 기법을 도입하고, LCS(Longest Common Subsequence) 알고리즘을 활용하여 코드의 구조적 유사성을 함께 분석한다. 제안된 방법은 GitHub에서 수집한 다양한 파이썬 오픈소스 프로젝트에 적용되어, 기존 방법 대비 서로 다른 제작자의 코드 간 유사도를 효과적으로 낮추고, 동일 제작자의 코드에서는 높은 유사도를 유지하는 성과를 보였다. 본 연구 결과는 파이썬 코드 도용 탐지 및 품질 관리 분야에 실질적인 기여를 할 수 있을 것으로 기대한다.

▸ **주제어**: 소프트웨어 버스마크, 코드 도용 탐지, 파이썬 바이트코드, 바이트코드 유사도, 최장 공통 부분 수열

• First Author: Seong-Min Kim, Corresponding Author: Heewan Park
*Seong-Min Kim (zcake1113@naver.com), Department of IT Software, Halla University
*Won-Chan Lee (lwc000316@naver.com), Department of IT Software, Halla University
**Heewan Park (heewanpark@halla.ac.kr), Department of IT Software, Halla University
• Received: 2025. 09. 08, Revised: 2025. 11. 12, Accepted: 2025. 11. 13.

http://www.ksci.re.kr pISSN:1598-849X | eISSN:2383-9945

I. Introduction

최근 소프트웨어 교육 및 개발 현장에서 파이썬(Python)은 그 문법의 단순성과 높은 접근성으로 인해 가장 널리 사용되는 프로그래밍 언어 중 하나로 자리 잡았다. 특히 교육 현장에서는 파이썬을 활용한 과제 제출 및 코드 평가가 활발히 이루어지고 있으며, 이에 따라 코드 유사도 검사를 통한 표절 탐지의 중요성 또한 증가하고 있다. 하지만 파이썬은 문법이 단순하고 자유도가 높아, 동일한 기능을 다양한 방식으로 구현할 수 있기 때문에 단순한 문자 수준의 비교만으로는 신뢰할 수 있는 유사도 판단이 어렵다[1-3].

이러한 배경에서, 파이썬 언어의 특징과 그로 인한 코드 도용 탐지 가능성에 주목하여 파이썬이 컴파일된 바이트 코드를 k개의 조각으로 나누어 유사도를 계산하는 연구가 진행되었다[4]. 그러나 해당 연구는 해당 코드의 중요성을 고려하지 않고 프로그램 조각들의 순차적인 유사도 분석에만 초점을 맞추었기 때문에 프로그램의 의미를 고려한 정밀한 유사도 판단에는 한계가 있었다.

이에 본 연구에서는 기존에 연구된 유사도 비교 기법의 한계를 보완하고자, 명령어를 구성하는 바이트 코드의 개수에 따라서 중요도를 차등 반영하기 위해서 가중치를 부여하는 기법과, 코드의 전체적인 흐름 및 구조적 유사성을 함께 고려할 수 있는 LCS (Longest Common Subsequence) 기반 분석 기법을 결합하여 이전보다 향상된 파이썬 코드 유사도 측정 방법을 제안한다. 그리고 실험을 통해서 기존 방법 대비 서로 다른 제작자의 코드 간 유사도를 효과적으로 낮추고, 동일 제작자의 코드에서는 높은 유사도를 유지하는 성과를 보여준다.

본 논문의 구성은 다음과 같다. 2장에서는 코드 도용이나 표절 탐지에 대한 기존 연구를 살펴보고, 3장에서는 본 논문에서 제안하는 새로운 기법에 대해서 상세하게 소개한다. 4장에서는 실험을 통해서 제안된 아이디어의 우수성을 검증하고, 5장에서 결론 및 향후 과제에 대해서 살펴본다.

II. Preliminaries

1. Related Work

1.1 Research Trends

최근 소프트웨어 개발의 확산과 함께, 소스코드 및 실행 파일 수준에서의 표절 및 도용 문제에 대한 관심이 높아지고 있다. 이에 따라 다양한 코드 유사도 분석 및 표절 탐지 기법이 제안되어 왔으며, 해당 기법들은 소스코드 기반 탐지와 바이너리 코드 기반 탐지로 나뉜다[1-3]. 대부분 문법 구조 또는 의미적 유사성을 기반으로 하는 기법들이 연구되고 있다.

소스코드 표절 탐지 분야에서는 토큰화(Tokenization), 파스 트리(Parse tree) 분석, AST(Abstract Syntax Tree) 기반 유사도 분석, 그리고 k-gram 기반 시퀀스 유사도 측정 기법 등이 연구되었다. 특히 k-gram 기법은 코드 내 연속된 명령어 혹은 토큰들의 고정 길이 집합을 생성하여 비교하는 방식으로서 구조적 유사성을 일정 부분 반영할 수 있다는 장점이 있다[4].

소스코드 표절 탐지 분야에서 문법 구조 기반 기법의 대표적인 예로는 토큰화(Tokenization)와 k-gram 기반 시퀀스 유사도 분석을 활용한 MOSS[5]와, 토큰화 및 파스 트리(Parse tree) 분석 기법을 사용하는 JPlag[6]이 있다. 해당 방식들은 실제로 소스코드 표절 탐지 도구로 많이 활용되고 있다.

한편, 의미 기반 기법으로는 AST(Abstract Syntax Tree) 기반 분석에 GNN(Graph Neural Network) 및 딥러닝을 활용한 접근법[7]이 제안되어 연구된 바 있다. 의미 기반 기법도 소스코드 유사도 측정에 효과적인 방식이지만, 실제 소스코드 도용 사례에서는 단순 복사와 붙여넣기 방식이 빈번하게 발생하는 경향이 있다.

바이너리 코드 수준의 도용 탐지 기법은, 소스코드를 확보하지 못하더라도 실행 파일(exe 등)을 직접 분석하여 유사성 또는 도용 여부를 판단할 수 있다는 점에서 중요하다. 이 분야에서는 주로 역공학(Reverse Engineering) 기법과 함께, 기계어 명령어 시퀀스 분석, 제어 흐름 그래프(Control Flow Graph, CFG) 기반 비교, 해시 기반 특성 추출, 기계학습(Machine Learning) 기법이 활용된 벡터 임베딩(Vector Embedding) 기반 비교 기법 등이 활용된다.

문법 구조를 기반으로 하는 대표적인 도구로는 대표적인 도구로는 BinDiff[8], DarunGrim[9]이 있으며, 관련 연구로는 BinSim[10]이 있다. 이들은 주로 함수 단위의 구조적 유사성 분석, 명령어 시퀀스 매칭, 그리고 통계적 특성 비교를 통해 바이너리 간 유사도를 평가한다.

특히 최근에는 딥러닝(Deep Learning)을 활용하여 바이너리 임베딩(Binary Embedding)을 생성한 후, 코드의 의미론적 유사성을 비교하는 접근도 활발히 연구되고 있다[11, 12].

이러한 기법들은 난독화(Obfuscation), 소프트웨어 패치(Software Patch) 전후 비교, 컴파일러 최적화, 코드 인젝션(Code Injection) 등 다양한 우회 기법에도 일정 부분

강인성을 보이며, 실제 악성코드 분석이나 라이선스 위반 탐지 등 다양한 분야에서 활용되고 있다[13].

요약하자면 문법 구조 기반 탐지는 주로 정적 분석에 기반하며, 의미적 유사성 기반 탐지는 동적 분석에 더 가까운 경우가 많다.

본 연구의 선행 연구에서는 바이너리 코드 기반의 문법 구조 탐지 방법으로, k-gram 기법을 활용한 파이썬 코드 유사도 측정 도구를 구현하였다. 먼저 확장자가 .py인 파일 또는 .exe인 파일로부터 각각 compileall 모듈[14]과 pyinstxtractor 도구[15]를 사용하여 .pyc 파일을 추출한 후, 해당 바이트코드를 기반으로 유사도를 분석하였다. 이때 추출된 바이트코드는 start line 정보를 기준으로 모아서 시퀀스로 구성되었으며, 이는 고정된 길이로 분할하는 일반적인 k-gram 방식보다 더 유연하고 실제 코드 구조를 반영한 효율적인 비교 지표로 작용하였다[4].

그러나 이러한 방식은 코드의 길이가 짧거나 구조가 단순한 경우(예: 매우 짧은 소스코드) 노이즈에 취약한 한계를 드러냈다. 이러한 한계를 보완하고 코드 유사도를 보다 정밀하게 측정하기 위해, 본 연구에서는 파이썬 바이트코드의 구조와 함께 k-gram 응용 기법, 가중치 기반 비교 방식, 그리고 LCS 기반 분석 기법을 결합한 새로운 분석 방법을 제안한다.

1.2 Theoretical Background

본 연구에서 제안하는 파이썬 코드 유사도 분석 기법을 이해하기 위해서는 파이썬 바이트코드의 구조와 이를 분석하는 데 활용되는 k-gram 기법, 가중치 기반 비교 방식, 그리고 LCS 기반 분석 기법에 대한 이해가 선행되어야 한다. 본 장에서는 이러한 핵심 개념들을 순차적으로 설명한다.

파이썬에서 확장자 .pyc 파일은 .py 확장자를 가진 소스 코드를 파이썬 컴파일러가 바이트코드로 컴파일하여 생성한 파일이다. 이 바이트코드는 파이썬 가상 머신에서 실행되며, 실행 시 소스 코드를 다시 컴파일할 필요 없이 빠르게 수행할 수 있도록 한다. 바이트코드는 기계어와 유사한 명령어 집합이지만, 특정 하드웨어에 종속되지 않고 파이썬 인터프리터가 이해할 수 있는 중간 언어 수준의 코드 형태를 갖는다.

k-gram 기법은 시퀀스 데이터를 k라는 일정한 길이로 나누어 유사도를 측정하는 방식으로, 코드 유사도 분석에서 널리 활용되는 전처리 방법이다. 본 연구에서는 이를 응용하여 파이썬 바이트코드에서 새로운 명령어가 시작되는 start line 정보를 기준으로 동일한 줄에 속한 opname(바이트코드 명령어 이름) 시퀀스를 하나의 그룹으로 묶어서 사용하고 이를 k-gram과 구별하여 s-gram이라고 명명하였다. k-gram은 단순히 k 크기로 바이트 코드를 잘라서 사용한다는 개념이라고 본다면, s-gram은 명령어 단위로 구분된 바이트코드들을 하나의 s-gram으로 구분한다. 기존의 k-gram 방식이 고정된 길이로 시퀀스를 분할하는 것과 달리, s-gram은 start line 단위로 나뉘기 때문에 그룹 간 요소 수가 일정하지 않은 특징이 있다.

가중치 기반 비교 방식은 모든 분석 단위를 동일하게 취급하지 않고, 특정 패턴이나 중요도가 높은 그룹에 더 큰 영향을 부여하는 방식이다. 선행 연구에서 s-gram 기반 유사도 계산은 그룹 간 요소 수의 차이를 고려하지 않고 각 그룹을 동일한 단위로 간주해 유사도를 계산했으나, 본 연구에서는 각 s-gram의 요소 개수에 따라 가중치를 부여하여 비교의 정확성을 높였다.

LCS 기반 분석 기법은 두 시퀀스 간에서 순서를 유지하면서 가장 긴 공통 부분 수열을 찾아 유사도를 측정하는 알고리즘이다. 이 방식은 일부 요소가 삽입되거나 삭제되는 등의 변화에도 견고하며, 다양한 방식으로 작성된 파이썬 코드처럼 구조화가 어려운 데이터에서도 공통된 패턴을 효과적으로 추출할 수 있다.

III. Proposed Method

본 절에서는 파이썬 바이트코드 기반의 유사도 분석을 위해 제안한 s-gram 분할 방식, 가중치 기반 유사도 계산 기법, 그리고 LCS 기반 시퀀스 비교 기법의 세부 과정을 설명한다.

1. Python Bytecode Similarity Analysis Process

본 연구에서는 파이썬 코드 유사도 분석을 자동화된 프로세스로 구현하기 위해, 파이썬의 requests[16] 외부 라이브러리와 GitHub API(Application Programming Interface)[17]를 이용하여 온라인에 공개된 파이썬 기반 오픈소스 레포지토리를 수집하였다. 자동 수집 시스템은 GitHub에서 설정한 언어 필터 및 별도 키워드를 기반으로 레포지토리를 탐색하며, .git 데이터를 통해 코드 파일을 로컬에 저장한다. 이를 통해 다양한 스타일과 구조를 가진 .py 파일을 확보할 수 있었으며, 수집된 파일들은 이후 유사도 분석에 사용되었다.

Fig. 1은 파이썬 유사도 도구의 전체적인 실행 흐름도를 보여준다.

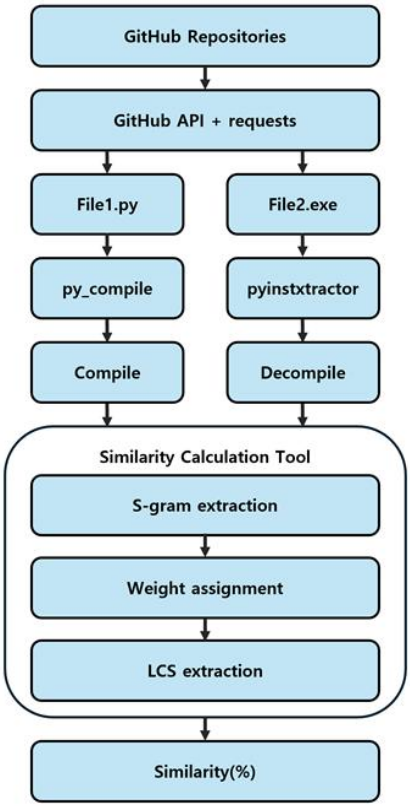


Fig. 1. Flow chart of Python Similarity Tool

수집된 소스코드는 바이트코드 수준에서 비교하기 위해 먼저 .pyc 형태로 변환된다. 일반적으로 파이썬 소스 파일은 실행 시 자동으로 .pyc 파일이 생성되지만, 본 연구에서는 분석 일관성을 위해 명시적으로 컴파일 과정을 거쳤다. 이때 사용된 도구는 compileall 모듈이며, 이 모듈은 기존 연구에서 사용한 py_compile[18] 모듈보다 확장성이 높아, 전체 디렉터리 단위 또는 다수의 .py 파일을 자동으로 일괄 컴파일할 수 있다는 장점이 있다. 이러한 점은 다량의 데이터를 다루는 본 연구의 자동화 환경에서 특히 유리하다.

또한, 일부 레포지토리에서는 .exe 형식의 실행 파일만 존재하는 경우도 있기 때문에, 해당 파일을 .pyc로 역변환하기 위한 디컴파일 도구로 pyinstxtractor[15]를 활용하였다. 이 도구는 실행 파일에서 내포된 파이썬 바이트코드를 추출해 .pyc 파일 형태로 복원해주며, 이전 연구에서도 동일하게 사용된 바 있어 검증된 도구로 간주된다.

이와 같이 변환된 .pyc 파일들로부터 바이트코드를 추출하기 위해, 파이썬 내장 모듈 marshal[19]과 dis[20]를 이용하여 각 함수 및 코드 블록 단위의 바이트코드를 분석하고, 그 중 opname이라 불리는 명령어 시퀀스를 획득하였다. 이 opname 시퀀스는 코드의 로직 흐름을 일정 수준 반영할 수 있는 중간 코드이며, 이는 유사도 분석의 핵심 데이터로 활용된다.

다음 단계에서는 이 opname 시퀀스를 기반으로 s-gram을 생성한다. 본 연구에서 제안한 s-gram은 기존의 k-gram 방식과 달리, 단순히 일정 개수로 고정된 구간을 자르는 것이 아니라, 각 명령어의 start line을 기준으로 묶는 방식을 채택하였다. 이 방식은 파이썬 코드의 문법적/논리적 블록 구조를 보다 정확히 반영할 수 있다는 장점이 있으며, 결과적으로 보다 정밀한 비교가 가능해진다. 다만, 이로 인해 각 s-gram의 길이가 다르다는 문제가 생기는데, 이는 다음 단계인 가중치 기반 유사도 계산 과정에서 보완된다.

마지막으로, 생성된 s-gram에 대해 각 그룹의 길이를 기준으로 가중치를 부여하고, 이후 각 코드 시퀀스 간의 유사도를 계산하기 위해 LCS 알고리즘[21]을 적용한다. LCS는 코드의 흐름을 유지하면서 공통된 명령어 시퀀스를 추출할 수 있도록 도와주는 방식으로, 일부 코드가 삽입되거나 삭제되는 등 구조가 약간 달라지더라도 핵심적인 유사성은 검출할 수 있도록 설계되었다. 이러한 방식은 특히 기능은 유사하나 변수명 변경, 조건 추가, 반복문 확장 등 다양한 방식으로 코드가 변경되는 경우에도 효과적으로 대응할 수 있다.

결과적으로, 본 연구에서 제안하는 파이썬 바이트코드 기반 유사도 분석 프로세스는 다양한 형태의 파이썬 코드에 대해 자동화된 수집, 정제, 분석 과정을 통해 높은 정밀도와 신뢰도를 갖춘 유사도 계산 결과를 도출할 수 있다. 특히, 유사도 산출 시 두 비교군의 가중치 총합 중 더 작은 값을 기준으로 나누는 방식을 적용하며, 이 과정에서 두 값의 균형을 보다 공정하게 반영하기 위해 조화 평균(Harmonic Mean)을 활용하였다. 이를 통해 명령어 수가 많은 코드와 적은 코드 간 비교에서도 편향을 줄이고, 보다 객관적인 유사도 평가가 가능하도록 하였다.

2. Similarity Calculation Formula and Algorithm

기존 방식에서는 .pyc 파일로부터 추출한 바이트코드를 이용해 유사도를 측정하였으며, 이 과정은 Fig. 2에 나타나 있다. 이후 Fig. 3과 같이 바이트코드 내부의 code object로부터 opname 시퀀스를 추출하고, 동일한 start line에 속한 명령어들을 하나의 그룹으로 묶어 s-gram을 생성하였다.

본 연구에서는 바이트코드 추출과 s-gram 생성 후 차별화된 가중치 부여와 유사도 산출 방식을 적용하였으며, 이후 세부 구현과 효과를 다룬다.

```
0          0    RESUME
1          2    NOP
2          4    LOAD_CONST
           6    LOAD_CONST
           8    IMPORT_NAME
           10   STORE_NAME
           12   LOAD_CONST
           14   LOAD_CONST
           16   IMPORT_NAME
           18   STORE_NAME
           20   LOAD_CONST
           22   LOAD_CONST
           24   IMPORT_NAME
           26   STORE_NAME
```

Fig. 2. Python Bytecode with Start Line

Fig. 2는 Start Line을 기준으로 분할한 파이썬 바이트코드의 구조를 시각화한 것으로, 각 라인에서 실행되는 바이트코드 명령어들의 분포와 흐름을 파악할 수 있다. 해당 구조를 분석함으로써 코드의 실행 단위 및 논리적 구간을 명확히 식별할 수 있었으며, 이는 이후 유사도 측정을 위한 시퀀스 분할에 기초 자료로 활용되었다.

```
0  [RESUME]

1  [NOP]

2  [LOAD_CONST, LOAD_CONST, IMPORT_NAME, STORE_NAME,
    LOAD_CONST, LOAD_CONST, IMPORT_NAME, STORE_NAME,
    LOAD_CONST, LOAD_CONST, IMPORT_NAME, STORE_NAME]
```

Fig. 3. s-gram Extraction

Fig. 3 또한 Start Line을 기준으로 추출된 opname 묶음을 나타낸 것으로, 명령어의 유형과 빈도 분포를 통해 코드의 기능적 특징을 분석하고, 명령어 그룹 간의 상대적 중요도를 평가하는 데 활용되었다. 이를 통해 코드 간 유사도 비교 시 단순한 명령어 일치 여부뿐만 아니라 명령어 조합의 구조적 유사성까지 반영할 수 있는 기반을 마련하였다.

이번에 제안하는 방식은 이러한 기존 구조를 유지하되, 다음과 같은 두 가지 측면에서 개선을 도모하였다.

```
0  [RESUME]
   weight: 1

1  [NOP]
   weight: 1

2  [LOAD_CONST, LOAD_CONST, IMPORT_NAME, STORE_NAME,
    LOAD_CONST, LOAD_CONST, IMPORT_NAME, STORE_NAME,
    LOAD_CONST, LOAD_CONST, IMPORT_NAME, STORE_NAME]
   weight: 12
```

Fig. 4. Weight Assignment

첫째, Fig. 4는 각 s-gram의 길이에 따라 부여된 가중치 값의 예시를 시각적으로 보여준다. 기존에는 모든 s-gram을 동일한 중요도로 간주했지만, 본 연구에서는 각 s-gram의 길이를 기반으로 중요도를 반영하였다.

이후 유사도 계산 시에는 단순한 교집합의 개수가 아니라, 교집합에 속한 s-gram들의 가중치의 총합을 구한다. 그리고 이 값을 두 비교군 중에서 가중치 총합이 더 적은 쪽의 전체 가중치 합으로 나누어 유사도를 산출한다. 이 방식은 명령어 수가 많은 그룹의 영향력을 더 정확히 반영할 수 있다는 장점이 있다.

```
File1 bytecode

1  [RESUME]

2  [NOP]

3  [LOAD_CONST, LOAD_CONST, IMPORT_NAME, STORE_NAME,
    LOAD_CONST, LOAD_CONST, IMPORT_NAME, STORE_NAME,
    LOAD_CONST, LOAD_CONST, IMPORT_NAME, STORE_NAME]

4  [LOAD_CONST, LOAD_CONST, IMPORT_NAME,
    CALL_INTRINSIC_1, POP_TOP]

5  [LOAD_CONST, LOAD_CONST, IMPORT_NAME, STORE_NAME]

6  [LOAD_CONST, LOAD_CONST, IMPORT_NAME, STORE_NAME]

7  [LOAD_CONST, LOAD_CONST, IMPORT_NAME, STORE_NAME]

8  [LOAD_CONST, LOAD_CONST, IMPORT_NAME, STORE_NAME]

9  [LOAD_CONST, LOAD_CONST, IMPORT_NAME,
    IMPORT_FROM, STORE_NAME, IMPORT_FROM,
    STORE_NAME, POP_TOP]

10 [LOAD_CONST, LOAD_CONST, IMPORT_NAME,
    CALL_INTRINSIC_1, POP_TOP]

11 [PUSH_NULL, LOAD_BUILD_CLASS, LOAD_CONST]

Total weight: 56
```

Fig. 5. Example of File 1

```
File2 bytecode

1  [RESUME]

2  [NOP]

3  [LOAD_CONST, LOAD_CONST, IMPORT_NAME, STORE_NAME,
    LOAD_CONST, LOAD_CONST, IMPORT_NAME, STORE_NAME,
    LOAD_CONST, LOAD_CONST, IMPORT_NAME, STORE_NAME]

4  [LOAD_CONST, LOAD_CONST, IMPORT_NAME,
    CALL_INTRINSIC_1, POP_TOP]

5  [LOAD_CONST, LOAD_CONST, IMPORT_NAME,
    IMPORT_FROM, STORE_NAME, POP_TOP]

6  [PUSH_NULL, LOAD_BUILD_CLASS, LOAD_CONST]

Total weight: 28
```

Fig. 6. Example of File 2

둘째, Fig. 5와 Fig. 6은 이전 절차에 따라 서로 다른 두 샘플 코드에서 추출된 s-gram 시퀀스를 보여주며, LCS 기반의 시퀀스 비교 기법을 추가로 적용하였다.

LCS는 두 시퀀스 간에서 공통된 순서를 유지하는 가장 긴 부분 수열을 찾아내는 알고리즘으로, 단순히 동일한 s-gram의 존재 여부만을 비교하는 기존 방식보다 더 정밀한 비교가 가능하다.

이는 코드의 중간에 명령어가 삽입되거나 일부 순서가 바뀌더라도, 전체적인 구조와 흐름이 유사하다면 LCS 알고리즘을 통해 그 유사성이 효과적으로 탐지될 수 있다. 이를 통해 단순 교집합 방식보다 더 구조적이고 의미 있는 유사도 판단이 가능하다.

```
LCS of File1 & File2

1  [RESUME]

2  [NOP]

3  [LOAD_CONST, LOAD_CONST, IMPORT_NAME, STORE_NAME,
    LOAD_CONST, LOAD_CONST, IMPORT_NAME, STORE_NAME,
    LOAD_CONST, LOAD_CONST, IMPORT_NAME, STORE_NAME]

4  [LOAD_CONST, LOAD_CONST, IMPORT_NAME,
    CALL_INTRINSIC_1, POP_TOP]

5  [PUSH_NULL, LOAD_BUILD_CLASS, LOAD_CONST]

Total weight: 22
```

Fig. 7. LCS of File 1 and File 2

마지막으로 Fig. 7에서는 이들 간에 적용된 LCS 결과를 시각적으로 나타낸다. 유사도 측정은 두 비교 대상 간의 LCS의 총 가중치를 기준으로 하되, 이 값을 두 비교군 중에서 더 적은 s-gram의 가중치를 보유한 쪽의 전체 가중치로 나누는 방식이다. 이는 코드 길이가 크게 다를 경우 유사도가 왜곡되는 문제를 줄이기 위해 조화 평균 기반 계산 방식을 채택하였다. 수식을 통해 설명하면 다음과 같다.

첫째, 파일 X와 파일 Y의 s-gram을 x와 y라 한다.

둘째, x와 y의 최장 공통 부분 수열을 $LCS(x, y)$라 정의한다.

셋째, x의 가중치를 W_x라 정의한다.

넷째, x와 y 중 더 적은 가중치를 가진 집합을 $Min(x, y)$라고 정의한다.

마지막으로 유사도는 다음과 같이 계산한다.

$$Similarity(\%) = \frac{W_{LCS(x,y)}}{Min(W_x, W_y)} \times 100$$

IV. Experiments

1. Experimental Dataset

본 연구에서 실험에 사용된 데이터셋은 GitHub에서 수집한 공개 파이썬 프로젝트들로 구성되어 있다. 이들 프로젝트는 서로 다른 제작자들에 의해 개발된 코드로, 일반적으로 상호 간의 유사도는 낮게 나타나는 것이 자연스럽다. 따라서 본 연구에서 제안하는 분석 기법이 기존 도구에 비해 격차가 있는 유사도 수치를 도출한다면, 이는 이전보다 정밀하고 민감한 유사도 측정이 가능함을 의미한다.

반면, 동일한 제작자가 유사한 목적을 위해 여러 버전의 코드를 작성하는 경우도 있다. 이러한 경우에는 코드 간 구조나 로직이 유사할 가능성이 높아 상대적으로 높은 유사도 수치를 보이는 것이 일반적이다.

본 연구에서는 실험의 대표 사례로 미로 탐색기 구현을 목적으로 한 GitHub 프로젝트들을 선정하였다. 이 주제는 다양한 알고리즘, 구현 방식, 난이도 등에 따라 코드가 달라질 수 있어, 동일한 제작자라도 서로 다른 구현이 가능하며, 서로 다른 제작자 간에는 더욱 뚜렷한 차이를 보일 수 있다. 이러한 특성은 코드 유사도 측정 기법의 정확도를 검증하기 위한 실험 대상으로 적합하다고 판단하였다.

해당 프로젝트들은 GitHub API와 Python의 requests 라이브러리를 이용하여 자동으로 수집되었으며, 이를 바탕으로 실험을 진행하였다.

실험에 앞서, 각 알고리즘과 서로 다른 제작자의 코드에 대한 일부 데이터셋을 번호 기반으로 정리한 실험용 테이블을 제시하여, 이후 실험 결과의 이해를 돕고자 한다.

그리고 이어지는 실험에서는 구체적인 결과 제시를 위해, 전체 데이터셋 중 일부 샘플을 대상으로 유사도 비교 결과를 먼저 서술한 뒤, 전체 데이터셋의 평균 유사도에 대해 설명한다.

Table 1. Algorithm ID for Similarity Measurement

ID	Similarity Algorithm
K3G	k-gram Similarity (k=3)
SG	s-gram Similarity
SGW	s-gram + Weight Similarity
SGL	s-gram + LCS Similarity
SGWL	s-gram + Weight + LCS Similarity

Table 1은 각 알고리즘 ID에 대응하는 유사도 측정 알고리즘을 상세히 정리한 표이다.

기본적으로 s-gram 방식을 사용하며, 새로운 알고리즘을 추가할 때마다 유사도 측정 성능을 비교하기 위해 이와 같이 나타냈다. 이를 통해 각 알고리즘이 어떤 방식으로 코드 유사도를 계산하는지 확인할 수 있다.

Table 2. Sample ID from Maze Solver Projects

ID	Developer	Algorithm	Remark
A1	Dev A	Dijkstra	Automatic
A2	Dev A	Dijkstra	-
A3	Dev A	Prims	Automatic
A4	Dev A	Prims	Variant A
A5	Dev A	Prims	Variant B
B1	Dev B	A-star	-
B2	Dev B	BFS	-
B3	Dev B	DFS	-

Table 2는 데이터셋의 각 ID에 따라 해당 샘플의 제작자와 사용된 알고리즘 정보를 함께 제시한 표이다.

해당 표는 실험에 사용된 일부 샘플을 발췌하여 보다 구체적인 분석을 돕기 위해 작성되었으며, 이를 통해 실험에 사용된 다양한 코드 샘플의 특징을 한눈에 파악할 수 있다.

2. Experiment on Credibility

본 연구에서 실험의 신뢰성을 확보하기 위해 신뢰도 평가를 진행한다. 이는 실험 결과의 일관성을 확인하고, 연구 도구와 방법의 재현 가능성을 검증하기 위한 과정이다.

Table 3. Experiment Between Different Groups

	K3G	SG	SGW	SGL	SGWL
A1:B1	69.30	37.61	22.51	26.61	13.88
A1:B2	52.28	28.02	15.15	20.77	10.30
A1:B3	51.01	27.62	14.75	20.48	10.03
A2:B1	67.61	36.70	21.39	26.61	13.88
A2:B2	50.30	27.05	15.15	19.81	10.00
A2:B3	49.08	26.67	14.75	19.52	9.74
A3:B1	68.74	39.45	23.45	28.44	14.82
A3:B2	49.80	28.02	15.94	20.29	10.20
A3:B3	48.60	27.62	15.53	20.00	9.93
A4:B1	67.61	39.45	23.45	28.44	14.82
A4:B2	49.31	28.02	15.94	20.29	10.20
A4:B3	48.02	27.62	15.53	20.00	9.93
A5:B1	69.68	40.37	24.58	29.36	15.76
A5:B2	52.98	29.95	16.73	21.74	10.59
A5:B3	51.69	29.52	16.30	21.43	10.32

Table 3은 서로 다른 제작자 간 코드에 대해 기존 유사도 측정 도구와 본 연구에서 제안한 기법으로 분석한 평균 유사도 수치를 비교한 것이다.

서로 다른 개발자가 작성한 소스코드는 일반적으로 유사도가 낮게 나타나야 하며, 본 연구에서는 이와 같은 특성을 기준으로 유사도 측정의 타당성을 검증하였다.

이에 따라, SG와 SGWL의 결과를 비교한 결과, 서로 다른 개발자가 작성한 소스코드에서 SGWL이 SG보다 약 2배 가량 낮은 유사도를 보여 유사도 감소 효과가 크게 나타나는 것을 확인할 수 있었다.

3. Experiment on Resilience

본 연구에서 다양한 조건하에서도 실험 결과가 유효한지 확인하기 위해 강인도 평가를 수행한다. 이는 실험 결과가 여러 변수에 영향을 받지 않고, 다양한 환경에서도 동일한 의미를 갖는지 검증하는 과정이다.

Table 4. Experiment Within the Similar Group

	K3G	SG	SGW	SGL	SGWL
A1:A2	92.61	88.99	86.38	87.35	84.42
A1:A3	81.58	72.26	68.71	64.95	62.54
A1:A4	81.34	72.21	67.68	62.87	60.22
A1:A5	89.14	81.70	76.41	69.36	67.14
A2:A3	86.06	82.90	77.42	74.24	70.95
A2:A4	85.46	82.90	78.75	75.88	74.09
A2:A5	85.48	75.18	66.51	63.47	57.98
A3:A4	99.46	97.49	96.00	96.36	94.21
A3:A5	92.41	89.03	85.19	86.02	82.32
A4:A5	92.92	90.21	85.20	87.24	82.44
B1:B2	83.80	75.23	74.67	71.56	71.11
B1:B3	83.80	75.23	74.67	71.56	71.11
B2:B3	98.91	98.55	96.44	98.07	96.04

Table 4는 동일한 제작자가 작성한 코드(예: 기능 확장, 구조 및 알고리즘 변경 등)에 대해 기존 방식과 제안한 방식의 유사도를 비교한 것이다.

동일한 제작자가 작성한 소스코드는 일반적으로 유사도가 높게 나타나야 하며, 본 연구에서는 이와 같은 특성을 기준으로 유사도 측정의 타당성을 검증하였다.

그리하여, 두 방식 모두 높은 유사도를 나타냈지만, 제안 방식 모두 전반적으로 높은 유사도를 나타냈으며, 제안 방식 또한 대부분의 사례에서 70% 이상의 유사도를 유지

하였다. 일부 구조가 변경된 경우에도 50% 이상의 유사도가 유지되어, 실질적인 코드 유사성 판단에서 일관된 결과를 보였다.

4. Experiment on Comparison with Existing Tools: Generalization and Quantification

본 연구에서는 대중적으로 널리 사용되는 코드 표절 검사 도구인 MOSS와의 비교를 통해 SGWL의 성능을 정량적으로 평가하고, 그 결과를 바탕으로 기존 도구들과의 일반화 가능성을 검토한다. 특히, 본 연구에서 제시하는 SGWL은 소스코드와 바이트코드를 교차 검증할 수 있는 기능을 제공하는 점에서 기존의 MOSS와 같은 도구들과의 차별성을 보인다. 그러나 SGWL의 유사도 검사 기법은 기존 도구들과 유사한 결과를 도출하며, 이를 통해 두 방법 간의 결과를 비교하고, SGWL의 신뢰성과 효율성을 평가하고자 한다. 또한, 본 연구에 앞서 MOSS 도구는 서로 다른 두 개의 소스코드를 비교할 때, 종합적인 유사도를 제공하는 것이 아니라 각 파일별 기준에 따라 개별적인 유사도 평가를 수행한다는 점을 명시해 둔다.

Table 5. MOSS Experiment Within the Different Group

	Similarity to File 1	Similarity to File 2	SGWL
A1:B1	0	0	13.88
A1:B2	0	0	10.30
A1:B3	0	0	10.03
A2:B1	0	0	13.88
A2:B2	0	0	10.00
A2:B3	0	0	9.74
A3:B1	0	0	14.82
A3:B2	0	0	10.20
A3:B3	0	0	9.93
A4:B1	0	0	14.82
A4:B2	0	0	10.20
A4:B3	0	0	9.93
A5:B1	0	0	15.76
A5:B2	0	0	10.59
A5:B3	0	0	10.32

Table 5는 MOSS 도구를 사용하여 서로 다른 제작자 간의 코드 유사도와 Table 3에서 제시된 SGWL의 유사도 결과를 비교한 것이다.

MOSS 도구는 코드의 구조적 특성을 분석하여 표절 유사도를 검사하는 방식으로 작동하며, 이로 인해 모든 검사 대상에서 0%의 유사도가 도출되었다. 반면, SGWL은 바이트코드를 순차적인 시퀀스로 나누어 유사도 검사를 수행하기 때문에 일부 유사한 부분이 발견되어 평균적으로 10% 초반대의 유사도 결과를 보인다.

이를 통해 기존의 소스코드 유사도 도구인 MOSS와 비교하여, SGWL의 유사도 측정 방식에서 발생할 수 있는 차이를 이해할 수 있으며, 이를 바탕으로 MOSS 도구와의 비교를 통해 서로 다른 제작자 간의 코드 유사도 기준을 대체적인 경향으로 파악할 수 있다.

Table 6. MOSS Experiment Within the Similar Group

	Similarity to File 1	Similarity to File 2	SGWL
A1:A2	72	84	84.42
A1:A3	53	57	62.54
A1:A4	48	55	60.22
A1:A5	65	51	67.14
A2:A3	66	61	70.95
A2:A4	68	67	74.09
A2:A5	52	35	57.98
A3:A4	91	97	94.21
A3:A5	83	60	82.32
A4:A5	81	55	82.44
B1:B2	61	29	71.11
B1:B3	55	27	71.11
B2:B3	94	91	96.04

Table 6은 MOSS 도구를 사용하여 동일 제작자가 작성한 소스코드에 대한 유사도 결과와 Table 4에서 제시된 SGWL의 유사도 결과를 비교한 것이다.

MOSS 도구의 경우, 낮은 유사도 값들의 평균은 56.15%, 높은 유사도 값들의 평균은 70.61%였으며, 전체 평균 유사도는 63.76%로 나타났다. 반면, SGWL의 평균 유사도는 68.62%로, MOSS 도구보다 높은 값을 보였다. 특히, A3:A4의 경우를 제외한 대부분의 비교에서 두 유사도 측정 도구는 비슷한 결과를 나타냈다.

이 결과는 MOSS 도구와 SGWL이 표절 검사에서 유사한 성능을 보임을 확인시켜 준다. MOSS 도구는 코드 표절 검사에 널리 사용되지만, 주로 소스코드 파일에만 한정되어 있어 컴파일된 바이트코드나 실행 파일에 대한 검사에는 한계가 있다. 반면, SGWL은 바이트코드와 소스코드를 모두 포괄하는 유사도 검사 방법을 제공함으로써, MOSS 도구와 유사한 성능을 보이면서도 더 넓은 범위에서 유용하게 사용될 수 있음을 시사한다.

따라서, SGWL은 기존의 MOSS 도구와 비교하여 실용적인 대안이 될 수 있으며, 특히 다양한 환경에서 적용 가

능한 표절 검사 도구로서 활용될 가능성이 크다고 할 수 있다. 이러한 점에서 SGWL은 표절 검사 도구의 범위를 확장하고, 다양한 코드 포맷에 대해 유효한 검사를 제공할 수 있는 잠재력을 가지고 있음을 보여준다.

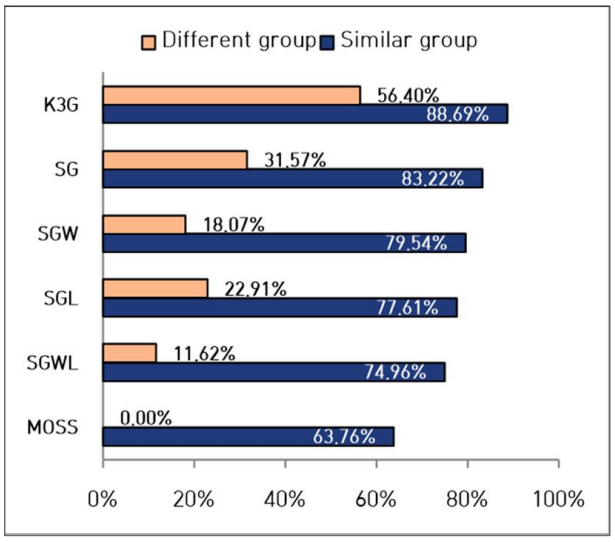


Fig. 8. Average similarities within and between groups for five algorithms and the MOSS tool based on a subset of data

Fig. 8은 Table 2에 제시된 미로 탐색기 샘플을 기준으로 각 알고리즘과 MOSS 도구의 유사도 비교 결과에 대한 종합 평균을 그래프 형태로 나타낸 것이다.

이 그래프는 MOSS 도구와 5가지 알고리즘의 유사도 검사 성능을 시각적으로 보여준다. 이를 통해 SGWL은 MOSS 도구와 비교하여 매우 일관된 결과를 도출하며, 바이트코드 기반 표절 검사에서 MOSS와 유사한 성능을 보임을 확인하였다. 특히, SGWL은 기존 도구들이 수행할 수 없는 .pyc 파일이나 .exe 파일에 대한 표절 검사에서 우수한 성능을 발휘할 수 있음을 입증하였다.

이 연구 결과는 SGWL이 소스코드와 바이트코드 교차 검증을 통해 유사도 비교를 할 수 있는 능력을 제공함을 시사한다. MOSS 도구는 주로 소스코드 파일에 대한 유사도 검사만을 지원하는 반면, SGWL은 소스코드와 바이트코드를 모두 포괄적으로 다룰 수 있는 장점을 지닌다. 이는 SGWL이 기존 도구들의 한계를 보완하고, 더 넓은 범위에서 코드 표절 검사 기능을 수행할 수 있다는 가능성을 제시한다.

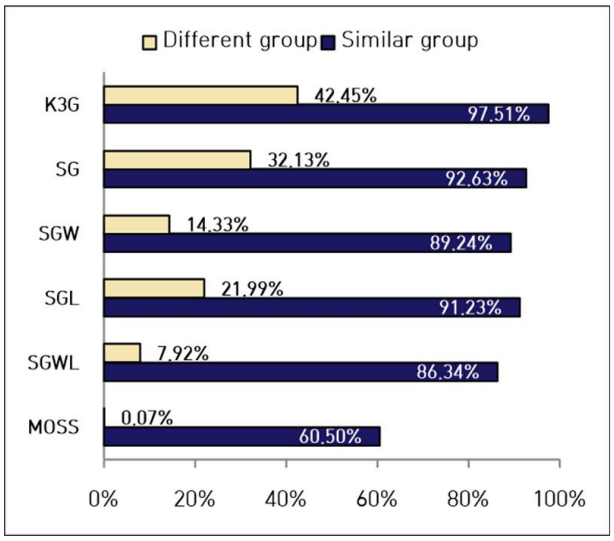


Fig. 9. Average similarities within and between groups for five algorithms and the MOSS tool based on 2048 Program

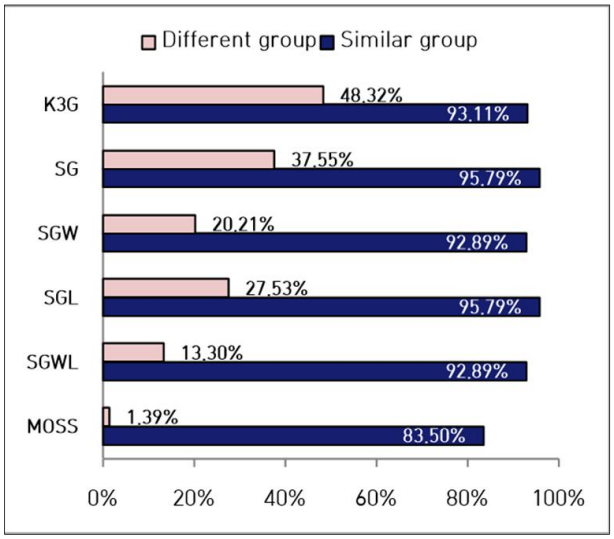


Fig. 10. Average similarities within and between groups for five algorithms and the MOSS tool based on Hangman Program

Fig. 9와 Fig. 10은 각각 5개의 알고리즘과 MOSS 도구를 사용하여 파이썬으로 구현된 2048 게임과 행맨(Hangman) 게임의 소스코드에 대한 유사도 비교 결과를 그래프로 나타낸 것이다.

이 그래프들은 다양한 프로그램에 대해 유사도 검사를 수행한 결과로, 기존 도구들과 비교하여 SGWL이 여러 종류의 코드에 대해 얼마나 일관된 성능을 발휘하는지 확인하는 데 목적이 있다. 특히, 본 연구에서는 다양한 프로그램을 대상으로 유사도 검사 성능을 평가함으로써, SGWL이 다른 코드베이스에도 일반화될 수 있는지, 즉 특정 게임이나 소스코드에 의존하지 않고도 광범위한 환경에서 유효하게 적용될 수 있는지에 대해 검토하고자 한다. 이로

써, SGWL의 효용성을 보다 넓은 범위에서 검증하고, 기존 도구들과의 차별성을 명확히 제시할 수 있다.

연구 결과를 살펴보면, Fig. 9의 2048 게임에서는 SGWL이 MOSS에 비해 상대적으로 더 큰 성능 차이를 보인 반면, Fig. 10의 행맨 게임에서는 MOSS가 SGWL보다 더 우수한 결과를 나타냈다. 이처럼 프로그램의 유형에 따라 세부적인 수치 차이는 존재하지만, 전체적으로 일관된 경향성을 확인할 수 있었다. 즉, SGWL은 기존 소스코드 표절 검사 도구와 유사한 수준의 성능을 유지하면서도, 소스코드가 아닌 컴파일된 .pyc 파일 상태에서도 유사도 검사가 가능하다는 점에서 실질적인 장점을 지닌다.

5. Experimental results

본 절에서는 앞선 실험들이 일부 데이터를 기반으로 한 구체적인 평가였음을 고려하여, GitHub API와 파이썬 requests 라이브러리를 사용하여 얻은 미로 탐색기 전체 데이터를 바탕으로 진행된 실험 결과를 종합적으로 분석하고, 연구의 핵심 결과를 도출하여 그 의미를 해석한다.

다만, MOSS 도구는 API를 통한 일일 요청 제한이 있으므로, 전체 데이터를 기반으로 한 실험을 수행하는 데에는 어려움이 존재하며, 이에 따라 본 절에서는 MOSS 도구의 실험 결과를 포함하지 않는다.

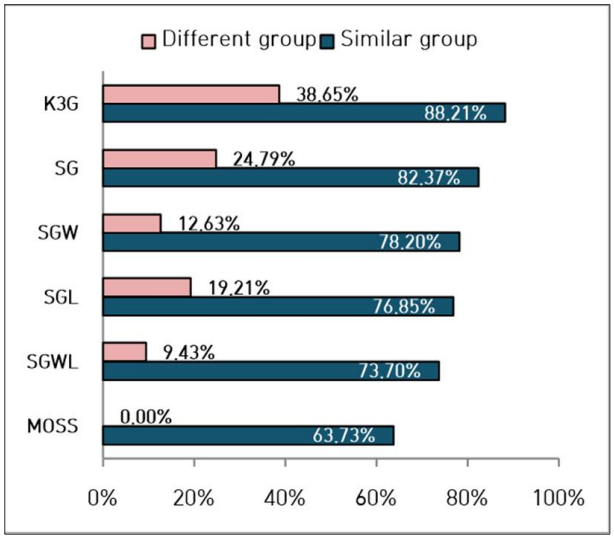


Fig. 11. Average similarities within and between groups for five algorithms and the MOSS tool

Fig. 11에 따르면, 동일 제작자 그룹에서는 기존 방식이었던 SG에 비해 유사도가 평균 8.5% 정도만 낮아진 반면, 서로 다른 제작자 그룹에서는 평균적으로 15% 이상 감소하는 경향을 보였다. 이는 제안 기법이 코드 간 실질적인 구조 차이를 보다 정밀하게 감지함으로써, 유사한 코드는 안정적으로 유지하고, 유사하지 않은 코드는 이전보다 명확히 구별할 수 있도록 했음을 의미한다.

그러나 예상치 못한 결과 또한 도출되었는데, SGW의 유사한 집단과 유사하지 않은 집단의 차이가 65.57%이며, SGWL의 경우 64.28%로 SGW이 유사도 측정에 있어 더욱 정밀한 결과를 나타내는 것으로 볼 수 있다. 다만, 이는 Table 2에서 서로 같은 개발자여도 프로그램에 사용되는 핵심 알고리즘을 서로 다른 알고리즘을 사용하여 제작한 소스코드 간의 유사도 측정에서 차이가 발생한 것이다. Table 4에서 A1, A2의 유사도 측정 비교군으로 A3, A4, A5가 선택된 경우를 보면 SGWL의 유사도가 다른 알고리즘에 비해 현저히 낮은 것을 확인할 수 있다.

일반적으로 소프트웨어 개발자는 성능, 유지보수성, 개발 비용 등을 고려하여 하나의 기능을 구현할 때 여러 알고리즘을 각각 적용한 버전을 여러 개 배포하는 건 효율적이지 않기에 하나의 최적 알고리즘을 선택해서 배포하는 경향이 있다. 그렇기에 이와 같이 특별한 경우를 제외하고 일반적인 경우를 측정하기 위해 전체 데이터셋에서 A1 샘플과 A2 샘플을 제외한 유사도 측정 결과를 나타내면 Fig. 12와 같이 나타낼 수 있다.

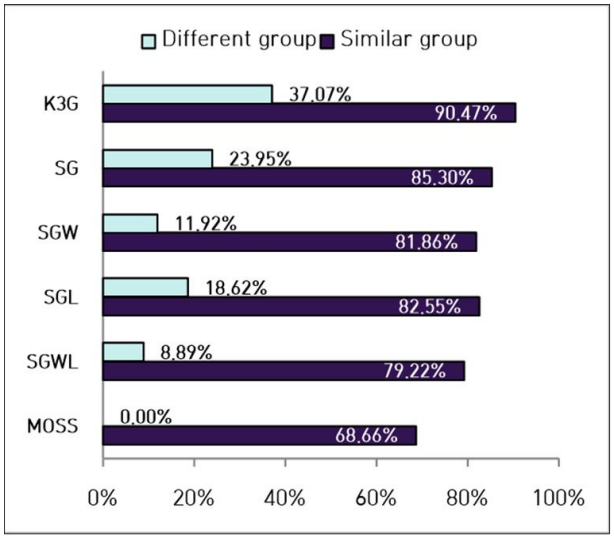


Fig. 12. Average similarities within and between groups for five algorithms and the MOSS tool, excluding samples A1 and A2

이처럼 SGW와 SGWL의 유사한 집단과 유사하지 않은 집단의 차이가 각각 69.94%, 70.33%로 근소한 차이로 SGWL의 성능이 더 우수하다. 또한, 핵심 알고리즘이 변경된 경우에도 전체 프로그램 구조나 동작 방식이 동일하다면, 이는 실질적으로 완전표절에 해당할 수 있다. 그러나 기존 탐지 기법들은 이러한 수준의 유사성을 효과적으

로 판별하지 못하는 한계가 있다. 반면, SGWL은 핵심 알고리즘의 변경까지 구분할 수 있어, 보다 정밀한 표절 탐지가 가능하다. 이와 같은 결과는 제안 방식이 단순한 토큰 일치 수준을 넘어서, 코드 구조와 명령어 흐름까지 함께 고려하여 코드 간 실질적인 유사도를 보다 정확히 평가할 수 있음을 시사한다.

실험 결과, Fig. 12 기준으로 동일 제작자가 작성한 코드(예: 기능 확장, 구조 일부 변경 등)에 대해 제안한 방식은 기존 방식보다 조금 낮아졌지만 높은 유사도 수치를 유지하였으며, 일부 구조가 변경되었더라도 전반적인 흐름과 의미가 유지되는 경우에는 유사도가 비교적 높게 측정되었다.

반면, 서로 다른 제작자에 의해 작성된 코드 간에는 기존 방식인 SG에 비해 유사도가 평균적으로 15% 이상 낮게 측정되어, 실질적인 코드 구조 차이를 보다 정밀하게 구분할 수 있음을 확인하였다. 특히, 가중치 부여와 LCS 알고리즘의 결합은 구조적 유사성이 부족한 코드들에 대해 보다 보수적인 판단을 가능하게 하는 효과로 이어졌다.

이와 같은 결과는 제안 방식이 단순한 토큰 일치 수준을 넘어, 코드 구조와 명령어 흐름까지 함께 고려하여 코드 간 실질적인 유사도를 보다 정확히 평가할 수 있음을 시사한다.

Table 7. Run times of five algorithms over Test 1 ~ Test 5 (ms)

	K3G	SG	SGW	SGL	SGWL
Test 1	83.08	12.03	12.93	26.14	27.25
Test 2	84.67	12.88	13.26	27.25	28.04
Test 3	86.61	14.35	13.62	27.83	28.66
Test 4	85.76	13.29	13.53	27.56	27.52
Test 5	89.52	14.54	13.61	28.50	28.64

추가적으로 알고리즘의 정확도뿐만 아니라 실행 효율성 역시 실제 적용 가능성을 판단하는 데 중요한 요소이기 때문에 각 알고리즘의 실행 속도 성능에 대한 평가를 수행하였다. Table 7은 수집했던 데이터셋의 유사도 측정 시 각 경우의 수들에 따른 실행 시간의 평균을 5번의 측정 결과와 평균을 구체적으로 나타낸 것이다. 측정 결과의 단위는 밀리초(ms) 이며 총 수치가 작을수록 시간 성능에 우수하다는 것을 의미한다.

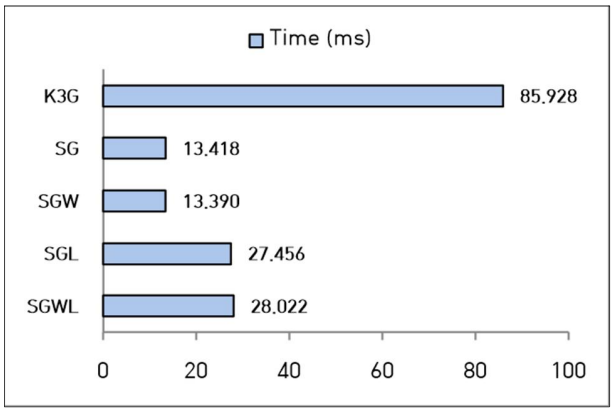


Fig. 13. Average run time of five algorithms over Test 1 ~ Test 5 (ms)

Fig. 13은 보다 직관적인 결과를 위해 제작된 그래프이며 시간 성능 면에서 가장 처음으로 고안되었던 K3G의 시간이 다른 알고리즘에 비해 큰 차이를 보이며 가장 오래 걸렸다. 이는 K3G는 k-gram 방식 특성상, s-gram 방식과 달리 k-gram의 개수가 항상 opname의 개수에서 k - 1을 뺀 고정된 수(N - k + 1)이기 때문이다. 반면, s-gram은 최소 1개에서 최대 opname의 개수만큼 생성될 수 있어 가변적이며, 필터링 과정을 거치므로 상대적으로 더 적은 수의 비교 연산이 수행된다. 따라서 그룹 크기가 가변적인 s-gram을 사용한 SG보다 그룹 크기가 고정적인 K3G은 처리해야 할 비교 대상이 상대적으로 많아져, 실행 시간이 더 길어졌다.

SG와 SGW는 가장 빠른 실행 속도를 보였으며, SGL과 SGWL은 이에 비해 다소 느린 성능을 나타냈다. 이는 LCS 기반 알고리즘이 추가되면서 비교 연산이 증가하고, 그로 인해 전체 실행 시간이 증가한 것으로 해석할 수 있다. 결과적으로 SGWL은 SG보다 실행 속도는 소폭 감소했으나, 해당 성능 저하는 전체 시스템 성능에 미치는 영향이 크지 않아, 실용적인 범위 내에 있다고 볼 수 있다.

추후 후속 연구에서는 MOSS와 같이 구조 기반 유사도 평가 기법을 도입하여, 다른 그룹 간 코드의 유사도가 거의 0%에 가깝게 하는 방향으로 연구를 발전시켜 나갈 예정이다.

V. Conclusions

본 연구에서는 파이썬 코드 유사도 측정을 보다 정밀하게 수행하기 위해 s-gram 기반 분할, 명령어 수 기반의 가중치 부여, 그리고 LCS 기반의 시퀀스 비교 기법을 결합한 새로운 측정 방식을 제안하였다.

이를 위해 GitHub API와 Python의 requests 라이브러리를 이용하여 GitHub의 미로 탐색기 구현을 목적으로 한 프로젝트들을 자동으로 수집하였다.

수집한 프로젝트들은 실제 표절 검사 환경과 유사한 조건을 구성하기 위해, 먼저 .exe 파일로 컴파일한 후 이를 다시 .pyc 파일로 디컴파일하는 과정을 거쳤다. 이후 디컴파일된 .pyc 파일을 기반으로 유사도 측정을 수행하였다.

제안 방식은 기존의 단순 토큰 기반 비교 방식이 갖는 한계를 보완하며, 코드의 구조적 흐름과 명령어 그룹의 상대적 중요도를 함께 반영함으로써 보다 정교한 유사도 분석이 가능함을 실험을 통해 입증하였다. 이는 동일한 기능을 수행하는 프로그램이라도, 소스코드에 사용된 핵심 알고리즘이 서로 다름을 구분할 수 있음을 의미한다.

특히 서로 다른 제작자의 코드 간 유사도는 기존 방식 대비 평균 15% 이상 낮게 측정되었으며, 이는 실제 코드 유사성이 낮은 경우에 대한 구분 능력이 향상되었음을 의미한다. 반면, 동일 제작자가 작성한 코드에서는 구조나 순서 일부가 변경되더라도 일정 수준 이상의 유사도가 유지되어, 코드 도용 또는 응용 버전에 대한 탐지가 효과적으로 이루어졌다.

본 연구에서는 일반화와 정량적 평가를 위해 기존 표절 탐지 도구인 MOSS와의 비교 실험을 수행하였다. 그 결과, 프로그램의 유형에 따라 세부적인 성능 차이는 존재하였으나, 전반적으로 일관된 경향을 확인할 수 있었다. 이는 제안한 SGWL이 기존 소스코드 기반 표절 탐지 도구와 유사한 수준의 성능을 보이면서도, 소스코드가 없는 .pyc 파일 환경에서도 효과적인 유사도 검사가 가능함을 의미한다. 따라서 본 연구의 접근법은 소스코드 접근이 제한된 환경에서의 프로그램 유사도 분석에 활용 가능성이 높다.

시간 성능 측면에서는 이전보다 연산이 추가되어 실행 시간이 다소 증가하였으나, 정밀한 측정이 가능해졌다는 점을 고려할 때 여전히 실용적인 범위 내에 있다. 한편, 일반적인 k-gram 기반 방식은 유사도 및 시간 성능 모두에서 가장 낮은 결과를 보였으며, 이를 응용한 s-gram 방식들은 파이썬 바이트코드 기반 유사도 측정에서 전반적으로 더 우수한 성능을 나타냈다.

또한, 본 연구에서는 일반적인 유사도 측정 도구와 달리 바이트코드를 기반으로 유사도를 측정하였으며, 개발한 도구는 실험에 사용된 .pyc 파일뿐만 아니라 .py 파일 형식에서도 측정이 가능하다. 따라서 .pyc 파일과 .py 파일 간의 교차 유사도 측정도 수행할 수 있다. 이는 소스코드와 컴파일된 바이트코드 간에도 효과적인 유사도 비교를 가능하게 하여, 복잡한 표절 검사 환경에서도 높은 적용성을 기대할 수 있다.

향후 연구에서는 다양한 프로그래밍 언어로의 확장성과 동적 분석 기법과의 결합을 통해서 보다 실용적인 코드 유사도 분석 체계를 구축하는 것을 목표로 한다. 본 연구는 코드 표절 탐지, 소프트웨어 품질 관리, 교육용 코드 검증 등 다양한 분야에서 활용될 수 있을 것으로 기대된다.

REFERENCES

[1] N. Gandhi, K. Gopalan, and P. Prasad, "A Support Vector Machine based approach for plagiarism detection in Python code submissions in undergraduate settings," Frontiers in Computer Science, Vol. 6, pp. 1-10, 2024. DOI: 10.3389/fcomp.2024.1393723

[2] M. Algabri, and F. Alhrazi, "Approach to Plagiarism Detection in Programming Assignments," Journal of Engineering and Technological Sciences, Vol. 3, No. 1, pp. 91-100, 2024. DOI: 10.59421/joeats.v3i1.2478

[3] O. Karnalim, "Python source code plagiarism attacks on introductory programming course assignments," Themes in Science and Technology Education, Vol. 10, No. 1, pp. 17-29, 2017.

[4] H. Park, J. Ko, S. Kim, S. Park, and J. Park, "Implementation of a similarity analysis tool to prevent Python code theft," Proceedings of the Korean Society of Computer Information Conference, Vol. 32, No. 2, pp. 543-546, Seoul, Korea, July 2024.

[5] A. Aiken, "MOSS: A system for detecting software plagiarism," Technical Report, Stanford University, pp. 1-20, 1994. *https://theory.stanford.edu/~aiken/moss/*

[6] L. Prechelt, G. Malpohl, and M. Philippsen, "Finding plagiarisms among a set of programs with JPlag," Journal of Universal Computer Science, Vol. 8, No. 11, pp. 1016-1038, 2002. DOI: 10.3217/jucs-008-11-1016

[7] Y. Kim, "Application of Deep Learning in Source Code Similarity Assessment," Journal of Software Assessment and Valuation, Vol. 20, No. 4, pp. 21-29, 2024.

[8] BinDiff, "Binary Diffing Tool," Zynamics, *https://www.zynamics.com/bindiff.html*

[9] DarunGrim, "Binary Diffing Framework," GitHub Repository, *https://github.com/ohjeongwook/DarunGrim*

[10] J. Ming, D. Xu, Y. Jiang, and D. Wu, "BinSim: Trace-based semantic binary diffing via system call sliced segment equivalence checking," Proceedings of the 26th USENIX Security Symposium, pp. 253-268, Vancouver, Canada, August 2017.

[11] G. Zhao, and J. Huang, "Deepsim: Deep learning code functional similarity," Proceedings of the 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, pp. 141-151, 2018. DOI: 10.1145/3236024.3236068

[12] M. Tufano, C. Watson, G. Bavota, M. Di Penta, M. White, and D. Poshyvanyk, "Deep learning similarities from different representations of source code," Proceedings of the 15th International Conference on Mining Software Repositories, pp. 542-553, Gothenburg, Sweden, May 2018. DOI: 10.1145/3196398.3196431

[13] L. Luo, J. Ming, D. Wu, P. Liu, and S. Zhu, "Semantics-based obfuscation-resilient binary code similarity comparison with applications to software plagiarism detection," Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, pp. 389-400, Hong Kong, China, November 2014. DOI: 10.1145/2635868.2635900

[14] Python Software Foundation, "compileall," Python Standard Library Documentation. *https://docs.python.org/ko/3/library/compileall.html*

[15] Extremecoders-re, "PyInstaller Extractor (pyinstxtractor)," GitHub repository, *https://github.com/extremecoders-re/pyinstxtractor*

[16] K. Reitz, "requests – HTTP for Humans," Requests documentation, *https://requests.readthedocs.io/en/latest/*

[17] GitHub, "GitHub REST API documentation," *https://docs.github.com/en/rest?apiVersion=2022-11-28*

[18] Python Software Foundation, "py_compile," Python Standard Library Documentation. *https://docs.python.org/ko/3/library/py_compile.html*

[19] Python Software Foundation, "marshal," Python Standard Library Documentation. *https://docs.python.org/ko/3.13/library/marshal.html*

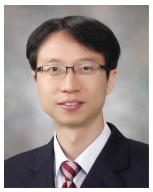
[20] Python Software Foundation, "dis," Python Standard Library Documentation. *https://docs.python.org/ko/3.8/library/dis.html*

[21] Hirschberg, D. S. (1977). Algorithms for the longest common subsequence problem. Journal of the ACM, 24(4), 664-675. DOI: 10.1145/322033.322044

Authors

Seong-Min Kim is currently a student in IT Software Engineering at Halla University. He is interested in development using Python and C#, Cloud-based convergence security, and Scenario-based penetration testing.

Won-Chan Lee is currently a student in IT Software Engineering at Halla University. He is interested in development using Java and Python, DevOps-based backend, and currently has an interest in working with data.



Heewan Park received the B.S. degree in Dept. of Computer Engineering from Dongguk University, Korea in 1997 and received the M.S. and Ph.D. degrees in Dept. of Computer Science from KAIST, Korea in 1999 and 2010, respectively. Dr. Park joined the faculty of the Dept. of IT Software at Halla University, Wonju, Korea in 2011. He is currently a professor in the Dept. of IT Software, Halla University. He is interested in code obfuscation, reverse engineering, software birthmark and software watermark.