ISSN 2383-630X(Print) / ISSN 2383-6296(Online)
Journal of KIISE, Vol. 52, No. 8, pp. 670-676, 2025. 8
https://doi.org/10.5626/JOK.2025.52.8.670

패치 경량화와 문맥 정보를 활용한 프로그램 자동 정정 개선

(Enhancing Automated Program Repair using Patch Lightweighting and Context Information)

정 은 서 † (Eunseo Jung)　아슬란 압디나비예프 †† (Abdinabiev Aslan Safarovich)　이 병 정 ††† (Byungjeong Lee)

요 약 프로그램 자동 정정 연구에서 대규모 언어 모델(LLM)은 매우 중요한 역할을 한다. 하지만 토큰 제한 문제로 인해 LLM이 처리할 수 있는 토큰 개수의 한계를 넘는 경우 LLM의 기능을 최대한 활용하지 못하고 버그나 에러를 올바르게 정정하지 못하는 문제가 발생한다. 따라서 본 연구에서는 패치 경량화와 유사한 메서드 정보를 활용하여 LLM의 토큰 길이 제한을 극복하여 더 많은 버그를 정정하고자 한다. 버그가 있는 메서드와 가장 높은 유사도를 가지고 있는 메서드를 문맥 메서드로 활용하고, 메서드 중 길이가 긴 메서드의 경우 패치 경량화 방법을 활용하여 LLM이 처리 가능한 토큰으로 메서드가 구성되도록 한다. 이러한 방법을 적용한 결과, 적은 토큰으로도 효율적인 버그 정정이 가능함을 확인하였다.

키워드: 패치 경량화, 문맥 메서드, 대규모 언어 모델, 프로그램 자동 정정

Abstract Large Language Models (LLMs) play a crucial role in the Automated Program Repair(APR) field. However, their effectiveness is constrained by token limitations. When the number of tokens exceeds the model's capacity, it struggles to fully utilize its capabilities, often failing to correctly detect and fix bugs. This study proposed an approach that could leverage patch lightweighting and context information to overcome these constraints. By incorporating the most semantically similar method as a context method and applying patch lightweighting to long methods, we ensured that the methods remained within the LLM's token limit. Through this approach, experimental results demonstrated that effective bug fixing could be achieved with fewer tokens, improving repair efficiency.

Keywords: patch lightweighting, context method, large language model, automated program repair

· 본 연구성과물은 2024년도 정부(교육부)의 재원으로 한국연구재단의 지원을 받아 수행된 기초연구사업임 (No. RS-2024-00465887)
· 본 논문은 정부(과학기술정보통신부)의 재원으로 한국연구재단의 지원을 받아 수행된 연구임 (NO. RS-2022-NR068754)
· 이 논문은 2024 한국소프트웨어종합학술대회에서 'LLM 기반 프로그램 자동 정정에서 토큰 길이 문제 처리를 위한 패치 경량화와 복원 활용'의 제목으로 발표된 논문을 확장한 것임

† 학생회원 : 서울시립대학교 컴퓨터과학과 학생
eunseoj@uos.ac.kr
†† 비 회 원 : 서울시립대학교 컴퓨터과학과 학생
aslan@uos.ac.kr
††† 종신회원 : 서울시립대학교 컴퓨터과학부 교수(Univ. of Seoul)
bjlee@uos.ac.kr
(Corresponding author임)

논문접수 : 2025년 3월 24일
(Received 24 March 2025)
논문수정 : 2025년 5월 14일
(Revised 14 May 2025)
심사완료 : 2025년 5월 28일
(Accepted 28 May 2025)

정보과학회논문지 제52권 제8호(2025. 8)

1. 서 론

소프트웨어 개발의 생산성과 품질을 향상시키기 위해 버그를 자동으로 수정하는 프로그램 자동 정정 연구(Automated Program Repair, APR)는 매우 중요하다. 의존성이 높은 대규모 소프트웨어에서 발생한 버그나 에러는 심각한 경제적·사회적 손실을 초래할 수 있으며, 이를 사람이 직접 수정하는 과정은 엄청난 비용과 시간이 소요된다. 따라서 APR 기술을 적용하면 보다 빠르고 정확하게 오류를 수정하여 개발자의 부담을 줄이고 소프트웨어의 안정성을 높이는 데 기여할 수 있다. 최근 APR 연구에서는 다양한 대규모 언어 모델(Large Language Model, LLM)을 활용하는 시도가 활발하게 이루어지고 있다. 하지만 LLM의 토큰 제한으로 인해 버그가 발생한 파일이나 프로젝트 모두 입력하여 버그 정정을 시도하는 것은 현실적으로 적용하기 어려운 점이 존재한다. 따라서 이러한 문제를 해결하기 위해 패치 경량화 기법과 문맥 정보 활용을 통해 길이가 긴 메서드 버그를 정정하는 것을 목표로 한다. 패치 경량화를 통해 토큰 길이를 모델이 처리할 수 있는 범위로 토큰 길이를 줄이고, 동일 파일 내에서 버그가 있는 메서드와 임베딩 유사도가 높은 문맥 메서드를 함께 활용한다. 이러한 방법을 통해 LLM의 토큰 제한을 극복하여 효율적으로 메서드 단위의 버그를 정정할 수 있는지 Java 프로젝트 기반의 벤치마크 데이터셋인 Defects4J[1]를 통해 검증한다.

2. 관련 연구

MCRepair[2]는 버그가 있는 메서드와 가장 많은 식별자를 공유하는 문맥 메서드를 하나의 buggy block으로 구성한다. 이후 미세 조정(Fine-tuning)한 CodeBERT[3] 모델을 통해 버그를 수정한다. 그러나 처리 가능한 토큰 개수를 고려하지 않아 메서드의 토큰 개수가 모델이 처리할 수 있는 개수보다 많은 메서드들을 올바르게 수정하지 못했다.

Rap-Gen[4]은 버그 코드가 입력되면 어휘적 및 의미적 유사성을 고려하여 독립적인 코드베이스에서 가장 적절한 버그-수정 쌍을 검색한다. 이후 검색된 쌍과 버그 코드를 활용하여 미세 조정한 CodeT5[5] 모델을 통해 패치를 생성한다. 하지만 단일 라인 정정에 초점을 맞춰 버그를 정정하기 때문에 복합적인 버그 수정에는 한계가 있을 수 있다.

RepairLLaMA[6]는 CodeLlama-7B[7] 모델을 미세 조정한 후 패치를 생성하고, 최적의 코드 입력과 코드 생성 방식의 조합을 통해 메서드 단위의 버그를 수정하는 것을 목표로 한다. 하지만 미세 조정 진행 시 입력 데이터와 생성한 데이터의 총합이 1,024개를 넘지 않은 데이터 쌍만을 사용하였다.

ITER[8]는 T5[9] 모델을 이용하여 버그 위치 확인, 패치 생성, 패치 검증을 여러 번 반복하여 여러 위치에 분포한 버그를 수정한다. 하지만 사용한 모델의 토큰 제한으로 인해 최대 384개의 토큰을 입력하고 최대 76개의 토큰을 생성하도록 설정하고 실험을 진행하였고, 메서드 외부에서 필요한 문맥 정보를 추가적으로 활용하지 않았다.

3. 패치 경량화와 문맥 정보 활용

3.1 개요

본 연구의 프로그램 자동 정정은 훈련단계와 추론단계로 구성되며, 전체 개요는 그림 1과 같다. 먼저 훈련

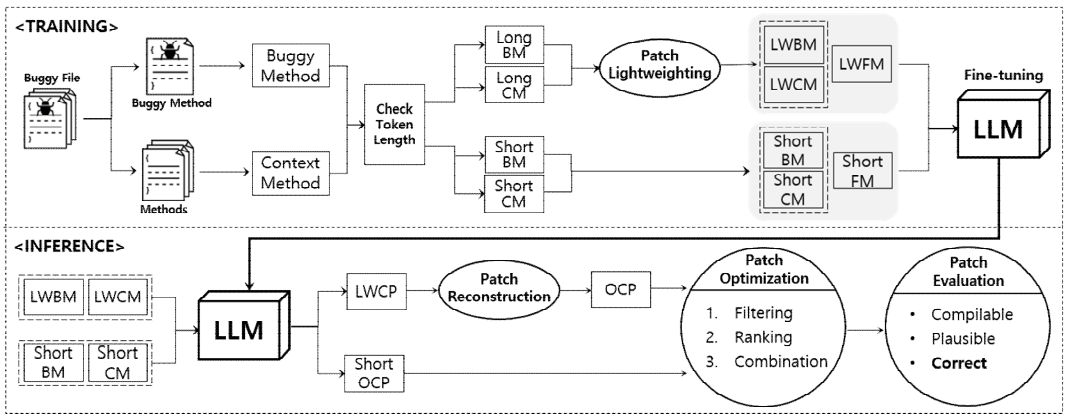

그림 1 전체 개요
Fig. 1 Overview

단계에서 버그가 발생한 파일이 입력되면, 버그가 발생한 메서드 즉 버기 메서드(Buggy Method, BM)와 기타 메서드들(Methods)로 분리한다. 그 다음 BM과 기타 메서드들 간에 유사도를 비교하여 가장 유사한 메서드를 찾고, 이를 문맥 메서드(Context Method, CM)로 추출한다. 이후 사용할 LLM의 토큰 제한 개수를 확인하여, BM이나 CM의 토큰 개수가 모델의 토큰 제한 보다 큰 값을 가진 긴 메서드일 경우 경량화를 진행한다. 만약 토큰 개수를 넘지 않는다면 경량화를 거치지 않고 그대로 사용한다. 모델 훈련을 위해 경량화된 버기 메서드(Lightweight Buggy Method, LWBM)와 경량화된 문맥 메서드(Lightweight Context Method, LWCM)는 경량화된 수정된 메서드(Lightweight Fixed Method, LWFM)와 쌍을 이루어 입력되고, 짧은 메서드(Short BM)와 짧은 문맥 메서드(Short CM)는 짧은 수정된 메서드(Short FM)와 쌍을 이루어 입력된다.

다음으로, 추론단계에서 앞서 훈련된 모델이 실제 데이터를 통해 올바르게 버그를 정정하는지 확인한다. 훈련단계와 동일하게 버그가 발생한 파일에서 BM과 CM을 추출하고, 길이가 긴 메서드일 경우 경량화를 진행한 후 미세 조정된 모델에 입력한다. 이후 모델은 경량화된 후보패치(Lightweight Candidate Patch, LWCP)를 생성하고, LWCP는 본래의 메서드 형태를 갖춘 후보패치(Original Candidate Patch, OCP)가 되도록 패치 복원을 거친다. Short BM을 통해 생성된 짧은 후보 패치(Short OCP)는 추가 복원 과정없이 그대로 사용한다. 이후 OCP와 Short OCP는 패치 최적화와 패치 평가를 통해 버그를 올바르게 수정했는지 최종 확인하는 과정을 거친다.

3.2 문맥 메서드 추출(Retrieve Context Method)

본 연구에서는 버그를 정정하기 위해 그림 2와 같이 BM과 가장 유사한 문맥 메서드 (CM)를 추출한다. 먼저 버그가 발생한 파일을 입력 받으면 BM와 그 외 메서드로 분리하고, 모든 메서드를 임베딩한다. 여기서 임베딩(Embedding)이란 기계 학습 및 인공 지능 시스템이 인간처럼 복잡한 지식 영역을 이해할 수 있도록 실제 객체를 수치화된 벡터로 표현한 것이다. 임베딩을 통해 딥러닝 모델은 메서드간 의미적 유사도를 효과적으로 분석할 수 있다. 기타 메서드 임베딩 값 중 BM의 임베딩 값과 가장 높은 코사인 유사도를 갖는 메서드를 검색하고, 이를 CM으로 추출한다. 이렇게 추출된 CM은 BM과 함께 버그 정정 과정에 활용된다.

만약 메서드가 임베딩을 처리하는 모델의 최대 토큰 개수를 넘는 경우 슬라이딩 윈도우(Sliding Window) 방식을 적용하여 임베딩한다. 슬라이딩 윈도우 기법이란, 연속된 데이터에서 일정한 크기의 윈도우를 설정하고, 그 윈도우를 하나씩 이동시키면서 데이터를 처리하는 기법이다. 본 기법을 적용하여, 토큰 제한으로 인해 코드 앞부분의 특징만 활용할 수 있는 한계점을 극복하고 코드 내용 전체를 반영한 임베딩 값을 구할 수 있다. 또한 중첩(overlap)을 적용하여 각 윈도우의 일정 부분이 겹치도록 하여 문맥이 단절되지 않고 모델이 연속적인 코드 흐름을 이해할 수 있도록 한다. 예를 들어, 길이가 긴 메서드를 512토큰 제한이 있는 모델을 통해 임베딩하는 경우, 100토큰의 중첩을 적용하여 분할되는 텍스트의 범위는 0-512, 412-924, 824-1336, … 와 같이 설정된다. 이렇게 분할된 각 메서드 윈도우를 임베딩한 후, 이들의 평균값을 구해 최종적으로 전체 메서드의 임베딩 벡터값으로 사용한다. 이러한 방법을 통해 긴 메서드도 효과적으로 임베딩할 수 있고, 연속된 정보의 손실 또한 최소화할 수 있다.

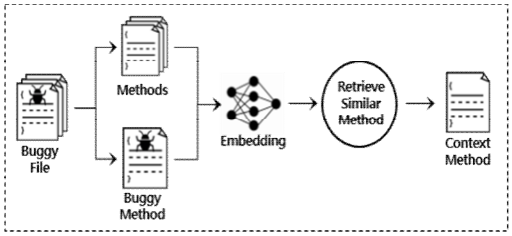

그림 2 문맥 메서드 추출 방법
Fig. 2 How to Extract Context Method

3.3 패치 경량화 (Patch Lightweighting)

패치 경량화는 메서드에서 버그가 발생한 위치, 즉 버기 라인(Buggy Line)을 기준으로 거리 점수와 유사도 점수를 구하고 이를 기준으로 낮은 점수를 받은 라인들을 제거하여, LLM이 처리할 수 있는 토큰 개수 이하가 되도록 메서드를 재구성하는 방법이다. 패치 경량화가 진행되는 과정은 Algorithm1 과 같다. 먼저 버기 메서드의 토큰 개수를 확인하여(라인 1) LLM이 처리 가능한 토큰 개수 이하인 짧은 메서드는 경량화를 진행하지 않는다(라인 2-3). 토큰 개수가 많은 긴 메서드는 아래 수식을 통해 구한 값을 통해 경량화를 진행한다.

먼저 일반 라인(s)과 버기 라인(b) 사이의 거리 점수(invd)를 공식 (1)을 사용하여 구한다(라인 5). 여기서 ln(s)와 ln(b)는 각각 일반 라인와 버기 라인의 줄번호를 나타낸다. 버기 라인은 항상 최대값인 1을 가지며, 일반 라인와 버기 라인의 거리가 가까울수록 1과 가까운 값을 가진다. 해당 공식은 버기 라인이 여러 줄일 때도 동일하게 적용되며, 버기 라인이 여러 줄일 경우 하나의 버기 라인은 버기 라인으로 지정되고, 나머지 버기 라인은 일반 라인으로 간주되어 계산된다.

Algorithm 1: Lightweight Process

Input: M: Method
Output: LWM: Lightweight Method

```
tokens ← calculate_tokens(M)
if token_count(tokens) < LLM_threshold then
    LWM ← M
else
    distance scores ← calculate_distance_score(M)
    similarity_scores ← calculate_similarity_score(M)
    total_scores ← calculate_total_scores
                   (distance_scores, similarity_scores)
    while token_count(M) > LLM_threshold do
        remove_lowest_total_score_line(M)
        tokens ← calculate_tokens(M)
    end while
    LWM ← M
end if
Return LWM
```

$$invd(s,b) = \frac{1}{|ln(b) - ln(s)| + 1} \quad (1)$$

다음으로 라인별 유사도 점수(sim)를 구하기 위해 먼저 메서드의 모든 라인을 임베딩하여 각 라인의 벡터값을 구한다. 이후 구한 벡터값을 통해 공식 (2)를 적용하여, 버기 라인(b)과 그 외 라인(s)간의 코사인 유사도를 계산한다(라인 6). 버기 라인은 항상 최대값인 1을 가지고, 버기 라인과 유사한 라인일수록 1에 가까운 값을 가진다.

$$sim(s,b) = \cos(\theta) = \frac{b \bullet s}{\|b\| \|s\|} \quad (2)$$

마지막으로 유사도 점수와 거리 점수를 종합하여 관련도 점수를 구한다(라인 7). 각 라인의 최종 값인 관련도 점수(rel)는 공식 (3)을 통해 계산된다. α와 β의 합은 1이 되도록 설정했고, 거리 점수에는 α, 유사도 점수에는 β를 곱한다. 만약 버기 라인이 여러 줄일 경우 최대값이 1을 초과할 수 있으므로 구한 값을 버기 라인의 개수로 나눠 최대값이 1이 되도록 조정한다.

$$rel(s,b) = \frac{\alpha \times invd(s,b) + \beta \times sim(s,b)}{number\ of\ buggy\ lines} \quad (3)$$

$$(\alpha + \beta = 1)$$

관련도 점수를 기반으로 점수가 낮은 라인부터 총 토큰의 개수가 모델의 토큰 제한 개수 이하가 될 때까지 삭제한다(라인 8-11). 이 과정을 통해 버그가 있는 라인과 가까운 위치에 있거나 유사도가 높은 라인들이 버그 라인과 함께 경량화된 메서드를 구성한다(라인 12). 이렇게 생성된 경량화된 메서드는 원래의 버기 메서드보다 짧지만, 관련도가 높은 라인만을 선별했기 때문에 버그를 수정하는 데 필요한 정보를 제공할 수 있다. 만약 경량화의 기준이 되는 버기 라인이 없는 CM을 경량화해야하는 경우, BM의 버기 라인과 가장 유사도가 높은 라인을 버기 라인으로 간주하고 동일한 알고리즘으로

No.	Java Code
1	private static Node computeFollowNode(Node…
…	…
46	if (cfa != null) {
47	for (Node finallyNode : cfa.finallyMap.get(parent)) {
48	<bug> cfa.createEdge(fromNode, Branch.UNCOND, finallyNode); </bug>
49	}
50	}
…	…
63	}
	Total: 63 lines, 600 tokens

(a) 원본 버기 메서드 (Closure14)
(a) Original Buggy Method (Closure14)

No.	Java Code
30	case Token.DO:
…	…
46	if (cfa != null) {
47	for (Node finallyNode : cfa.finallyMap.get(parent)) {
48	<bug> cfa.createEdge(fromNode, Branch.UNCOND, finallyNode); </bug>
49	}
50	}
…	…
61	return computeFollowNode (fromNode, parent, cfa);
	Total: 32 lines, 300 tokens

(b) 경량화된 버기 메서드 (Closure14)
(b) Lightweight Buggy Method (Closure14)

그림 3 패치 경량화 예시
Fig. 3 Example of Patch Lightweighting

경량화를 진행한다. 경량화된 메서드의 예시는 그림 3과 같다. 그림 3은 Defects4J 데이터 중 Closure14의 BM과 LWBM의 예시이다. 그림 3(a)에서 BM은 63줄, 600토큰으로 구성된 긴 메서드였다. 하지만 경량화 이후 버기 라인과 관련도가 높은 32줄이 선별되어 그림 3(b)처럼 최종적으로 300토큰으로 줄어든 것을 확인할 수 있다.

3.4 패치 생성 (Patch Generation)

모델은 훈련단계에서 대용량의 데이터셋 쌍을 통해 미세 조정되고, 추론단계에서 LWBM과 LWCM 또는 Short BM과 Short CM을 입력받아 각각 LWCP 또는 Short OCP를 생성한다.

3.5 패치 복원 (Patch Reconstruction)

패치 복원은 버그 토큰이 있는 경량화된 버기 메서드(LWBM)와 원래의 버기 메서드(OBM) 그리고 경량화된 후보패치(LWCP)간의 공통적인 특성을 활용하여, 경량화된 후보패치(LWCP)를 온전한 메서드 형태를 가지는 후보패치(OCP)로 만들어주는 방법[10] 이다. 각 후보패치 및 메서드는 표 1과 같은 특성을 가진다. LWCP와 OCP는 수정된 코드를 가지고 있지만 버그 토큰이

표 1 패치 및 메서드 특징
Table 1 Characteristics of Patches and Methods

	LWCP	LWBM	LWCP+	OBM	OCP
Bug Token	X	O	O	O	X
Fixed Code	O	X	O	X	O

없고, LWBM과 OBM은 수정된 코드는 없지만 버그 토큰을 가지고 있다. 이를 바탕으로, 버그 토큰과 수정된 코드를 모두 포함하는 LWCP+를 생성하여 LWCP를 OCP로 복원하고, OCP를 통해 검증단계를 거칠 수 있도록 한다. 먼저 LWBM과 LWCP의 공통된 경량화 구조를 활용해 LWBM의 버그 토큰 위치에 LWCP의 수정된 코드를 삽입하여 LWCP+를 생성한다. 이후 OBM의 버그 토큰위치에 LWCP+의 수정된 코드를 대체하고, 버그 토큰을 제거하여 최종적으로 OCP를 생성한다. 이를 통해 OCP는 온전한 메서드 구조와 수정된 패치를 모두 가지게 되며, 이렇게 복원된 OCP는 패치 최적화 단계를 진행한다.

3.6 패치 최적화와 패치 평가 (Patch Optimization & Patch Evaluation)

OCP는 패치 평가 전 패치 필터링(Patch Filtering), 패치 순위화(Patch Ranking), 패치 조합(Patch Combination)의 3단계로 이루어진 패치 최적화(Patch Optimization)[11]를 진행한다. 먼저 패치 필터링을 통해 중복 생성된 패치는 제거하고, 코드 구문에 오류가 있거나 종료 코드를 포함하는 의미 없는 패치 또한 삭제한다. 다음으로, 각 패치에 n-gram 유사도와 action 유사도 값을 적용해 정확하게 수정된 패치에 높은 순위를 부여하는 패치 순위화를 진행한다. 마지막으로, 단일 파일 내 각 메서드의 후보 패치들을 조합한다. 조합된 후보 패치는 패치 검증(Patch Validation)을 통해 문법적으로 오류가 없는지(Compilable), 문법적 오류도 없고, 테스트 케이스도 모두 통과하는지(Plausible) 확인한다. 만약 모든 테스트 케이스를 통과하는 경우 이를 Plausible Patch, 즉 PL 패치로 분류하고, PL 패치는 개발자가 직접 정확하게(Correct) 수정되었는지 확인한다.

4. 실 험

4.1 실험환경

본 연구에서는 코드 생성과 임베딩을 위해 모두 토큰 길이가 512개까지 처리 가능한 CodeBERT 모델을 사용하였으며, 768차원의 임베딩 벡터를 활용했다. 모델의 토큰 제한에 맞추기 위해 버기 메서드의 경량화 임계값은 300토큰, 문맥 메서드의 경량화 임계값은 200토큰으로 설정하고 실험을 진행하였다. CodeBERT 모델이 경량화된 메서드 구조를 이해할 수 있도록 Bugs2fix[12]의 대용량 Java 데이터셋을 경량화하여 훈련단계에 사용했다. 실제 검증은 Defects4J 1.2버전 데이터에 실제 버그 위치를 제공하여 실험을 진행하였고, 각 메서드 당 500개의 패치를 생성했다. Defects4J는 실제 오픈소스 프로젝트에서 수집된 Java 프로젝트 기반의 벤치마크 데이터셋으로, 프로그램 자동 정정(APR), 소프트웨어 테스트 연구 등에 사용된다.

실험은 NVIDIA RTX A6000 GPU 2개, Intel Xeon Gold 6226R 2.9GHz CPU 2개, 512GB RAM, Ubuntu 22.04 LTS 환경에서 진행되었다.

4.2 실험결과

4.2.1 버그 정정 결과

실험을 통해 Defects4J 1.2버전에서 메서드의 토큰 길이가 512개를 넘는 77개의 버그 중 9개의 버그를 올바르게 수정하였다. 수정한 9개 버그 정보는 표2와 같다. 수정한 메서드들의 평균 토큰 수는 816개, 최대 토큰 1,975개로 이루어진, 모델이 처리 가능한 범위를 넘는 긴 메서드였지만, 모델은 경량화된 정보와 관련된 문맥 정보를 통해 올바르게 버그를 정정했다. 정정한 버그의 특징을 살펴보면 Closure78과 Math38은 연속하지 않은 위치에 분포한 한 줄 이상으로 구성된 멀티 청크(Multi Chunk)버그임에도 수정이 필요한 위치의 라인을 모두 올바르게 정정했다. 또한 버그를 수정한 패턴을 분석한 결과, Chart7, Closure18, Closure73, Lang24, Math38은 버그가 발생한 부분을 다른 토큰들로 대체(Replace)하여 버그를 수정했고, Closure31, Closure78, Closure126, Math50은 해당 위치의 토큰들을 제거(Delete)하여 버그를 수정했다. 또한 관련 연구[12]와 비교했을 때 Closure78와 Closure126(표 2에서 *로 표시된 항목)을 추가적으로 정정했다. 이러한 차이는 관련

표 2 수정한 버그 정보
Table 2 Fixed Bug Details

Project	Multi Chunk	Fix Pattern	Token Length
Chart7	X	Replace	578
Closure18	X	Replace	742
Closure31	X	Delete	903
Closure73	X	Replace	582
Closure78*	O	Delete	624
Closure126*	X	Delete	641
Lang24	X	Replace	657
Math38	O	Replace	1,975
Math50	X	Delete	649

연구가 긴 버기 메서드만을 경량화하여 버그 정정을 시도한 반면, 본 연구에서는 버기 메서드 외부에서 관련 정보를 추가로 제공하여 더 많은 버그를 정정할 수 있었다. 또한 경량화 알고리즘에서도 기존에는 공통된 식별자만을 고려하여 유사도를 계산했던 반면, 본 연구에서는 코드 라인 단위의 유사도를 반영한 정교한 계산 방식을 적용하여, 유사도가 높은 라인을 효과적으로 선별하여 경량화했다. 이러한 접근 방식의 차이가 버그 정정 성능 향상에 기여한 것으로 판단된다.

4.2.2 경량화 방법별 정정 결과

표 3은 77개의 버그 중 메서드의 토큰 길이가 512개를 초과하는 긴 메서드를 대상으로 한 실험 결과를 나타낸다. 실험은 거리 점수만 반영한 경우(α=1, β=0), 유사도 점수만 반영한 경우(α=0, β=1), 그리고 두 점수를 모두 반영한 경우(α=0.5, β=0.5) 패치 경량화를 수행하고 정정한 버그 결과를 보여준다. 본 실험은 각 점수 방식이 긴 메서드 정정에 얼마나 효과적인지 비교하고, 거리와 유사도 점수를 모두 고려한 관련도 점수의 유효성을 검증하는 데 목적이 있다. 실험 결과, 거리와 유사도를 모두 반영한 경우 총 10개의 긴 메서드를 정정하였으며, 거리 또는 유사도만을 적용하여 경량화를 한 경우 일부 버그를 올바르게 정정하지 못한 것을 확인했다. 또한 거리와 유사도 점수를 결합한 방식에서만 정정할 수 있는 버그(Unique Fixed)가 1개 존재하였으며, 이는 관련도 점수 기반 경량화 방식이 물리적인 거리와 의미적 유사성이라는 기준을 통합함으로써 정보 손실을 최소화하고, 보다 효과적인 경량화를 가능하게 하는 것을 의미한다. 하지만 본 실험은 상대적으로 소규모의 데이터셋을 기반으로 수행되었기 때문에, 결과의 일반성과 실용성을 확보하기 위해서는 다양한 데이터셋을 활용한 확장 실험이 필요하다. 추가적인 데이터셋을 통한 실험에서 거리와 유사도 점수를 통합한 방식이 더 많은 버그를 정정할 수 있음을 확인한다면, 관련도 점수 기반 경량화 기법이 실제 개발 환경에서도 효과적인 전략임을 뒷받침하는 중요한 근거가 될 수 있다.

표 3 경량화 방법별 정정 결과 비교
Table 3 Bug Fixing Results by Lightweighting

Lightweight Method	Fixed	Not Fixed	Unique Fixed
Distance & Similarity	10	0	1
Distance Only	9	1	0
Similarity Only	7	3	0

5. 토 의

실험을 통해 긴 메서드를 경량화하고 문맥 정보를 활용해 추가적인 문맥정보를 LLM에 함께 입력하여 관련 연구[12] 대비 더 많은 버그를 정정할 수 있었다. 하지만 올바르게 정정하지 못한 버그들을 분석한 결과 관련도가 높은 문맥정보를 함께 제공함에도 새로운 패턴의 코드 삽입이 필요한 버그나 프로젝트 범위의 메서드 호출이 필요한 버그를 올바르게 수정하지 못한 한계점이 발견되었다. 이는 실험에 사용한 LLM인 CodeBERT 모델이 기존 코드 패턴을 학습하여 수정하는 부분에서 강점을 보이지만, 새로운 코드블록을 삽입해야 하는 경우 상대적으로 취약할 가능성이 있다. 따라서, 코드 생성에 강점이 있는 모델을 활용한 추가 실험이 필요하다. 또한 보다 넓은 범위의 프로젝트 단위의 문맥 정보가 필요하지만, 본 실험에서는 버그가 발생한 메서드와 동일한 파일 내 정보만을 활용했기 때문에 충분한 문맥 정보가 제공되지 않았을 가능성이 존재한다. 이를 보완하기 위해 프로젝트 단위의 정보 검색을 적용할 필요가 있다. 또한 본 연구는 Java 코드 기반으로 실험을 진행했기 때문에 다른 프로그래밍 언어에서도 동일하게 적용될 수 있는지는 검증되지 않았다. 아울러, 실제 데이터를 기반으로 구축된 공개 데이터셋인 Defects4J를 통해 성능을 검증하였기 때문에 LLM이 해당 데이터를 사전에 학습했을 위험성이 존재한다. 따라서 널리 알려지지 않은 데이터셋을 이용한 추가적인 검증이 필요하다. 마지막으로 버그 정정의 평가 기준으로 테스트 케이스를 통과한 후보패치를 개발자가 직접 검토하는 과정에서 주관이 개입될 수 있다. 따라서 객관적인 평가를 위해 정확성을 측정할 수 있는 지표를 도입할 필요가 있다.

6. 결 론

본 연구에서는 패치 경량화와 문맥 메서드 활용을 통해 LLM이 처리할 수 있는 토큰 범위 내에서 효과적으로 버그를 정정하는 방법을 제안하였다. 패치 경량화를 통해 긴 메서드를 모델의 토큰 제한에 맞게 효과적으로 경량화 하였고, 버그가 발생한 메서드와 유사한 문맥 메서드 정보를 제공함으로써 LLM을 활용한 프로그램 자동 정정의 성능을 향상시켜 관련 연구 대비 더 많은 버그를 정정했다. 그러나 버그 정정 과정에서 추가적인 코드 삽입 및 프로젝트 단위의 정보가 필요한 경우, 이를 보완하여 수정해야 할 필요성이 확인되었다. 따라서 향후 연구에서는 코드 간의 관계를 철저하게 분석한 후 코드 삽입을 포함한 코드 수정 패턴을 보다 정교하게 학습하고, 프로젝트 단위의 정보를 효과적으로 활용하는 방법에 대해 연구할 예정이다.

References

[1] R. Just, D. Jalali and M. D. Ernst, "Defects4J: A Database of Existing Faults to Enable Controlled Testing Studies for Java Programs," *Proc. of the 2014 International Symposium on Software Testing and Analysis (ISSTA)*, pp. 437-440, 2014.

[2] J. Kim and B. Lee, "MCRepair: Multi-Chunk Program Repair via Patch Optimization with Buggy Block," *Proc. of the 38th ACM SIGAPP Symposium on Applied Computing (SAC)*, pp. 1508 - 1515, 2023.

[3] Z. Feng et al., "CodeBERT: A Pre-Trained Model for Programming and Natural Languages," *Findings of the Association for Computational Linguistics: EMNLP,* pp. 1536 - 1547, 2020.

[4] W. Wang, Y. Wang, S. Joty and C.H. Hoi, "RAP-Gen: Retrieval-Augmented Patch Generation with CodeT5 for Automatic Program Repair," *Proc. of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, pp. 146-158. 2023.

[5] Y. Wang, W. Wang, S. Joty and S. Hoi, "CodeT5: Identifier-aware Unified Pre-trained Encoder-Decoder Models for Code Understanding and Generation," *Proc. of the 2021 Conference on Empirical Methods in Natural Language Processing (EMNLP),* pp.8696-8708, 2021.

[6] A. Silva, S. Fang and M. Monperrus, "RepairLLaMA: Efficient Representations and Fine-Tuned Adapters for Program Repair," arXiv preprint arXiv:2312.12950, 2023.

[7] B. Roziere et al, "Code Llama: Open Foundation Models for Code", arXiv preprint arXiv:2308.15698, 2023.

[8] H. Ye and M. Monperrus, "ITER: Iterative Neural Repair for Multi-Location Patches," *Proc. of the 46th IEEE/ACM International Conference on Software Engineering (ICSE),* pp. 1-13, 2024.

[9] C. Raffel, N. Shazeer, A. Roberts, K. Lee, S. Narang, M. Matena, Y. Zhou, W. Li and P. Liu, "Exploring the Limits of Transfer Learning with Unified Text-to-Text Transformer", *Journal of Machine Learning Research,* pp. 1-67, 2020.

[10] E. Jung, A. Safarovich and B. Lee, "Utilizing Patch Lightweighting and Reconstruction to Handle Token Length Issues in LLM based Automatic Program Repair," *Proc. of the Korea Software Congress,* pp. 381-383, 2024. (in Korean)

[11] A. Safarovich, J. Kim, and B. Lee, "Improving Patch Optimization for Multi-Chunk Bugs in Automated Program Repair," *Proc. of the Korea Computer Congress,* pp. 200-202, 2023.

[12] M. Tufano et al, "An empirical study on learning bug-fixing patches in the wild via neural machine translation", *ACM Transactions on Software Engineering and Methodology (TOSEM),* pp. 1-67, 2019.

정 은 서

2017년 한국외국어대학교 경제학과 및 영어학과(학사). 2023년~현재 서울시립대학교 컴퓨터과학과 석사과정. 관심분야는 딥러닝, 프로그램 자동 정정, 소프트웨어 테스팅

아슬란 압디나비예프

2020년 우즈베키스탄 국립대학교 IT전공(학사). 2024년 서울시립대학교 컴퓨터과학과(석사). 2024년~현재 서울시립대학교 컴퓨터과학과 박사과정. 관심분야는 소프트웨어 품질 향상, 소프트웨어공학

이 병 정

1990년 서울대학교 계산통계학과(학사) 1990년~1998년 ㈜SK하이닉스반도체 연구원. 1998년 서울대학교 전산과학과(석사) 2002년 서울대학교 전기컴퓨터공학부(박사) 2002년~현재 서울시립대학교 컴퓨터과학부 교수. 관심분야는 소프트웨어공학, 소프트웨어 진화