프로그램 자동 정정을 위한 LLM 모델의 효과적인 토큰 길이 제한 해결 기법

정은서, 아슬란 압디나비예프, 이병정
서울시립대학교 컴퓨터과학과
{eunseoj, aslan, bjlee}@uos.ac.kr

Towards Effectively Resolving Token Length Limits of LLM Models for Automatic Program Repair

Eunseo Jung, Abdinabiev Aslan Safarovich, Byungjeong Lee
Department of Computer Science, University of Seoul

Abstract

In automatic program repair technology, selecting an appropriate deep learning model plays a pivotal role because it directly influences the experimental results. However, when applying large language models such as CodeBERT and CodeLlama, there are constraints on the number of input and output tokens. If the input exceeds the model's token capacity or demands more output tokens than it can handle, the model may produce incomplete results. Therefore, it is necessary to tailor existing data by including relevant information according to the model's specifications, enabling the model to generate complete results. This paper complements the existing studies by addressing the token limit issue in deep learning models via effective data lightweighting and reconstruction.

1. Introduction

The technology of Automatic Program Repair (APR) contributes significantly to software development. Since a single code error can undermine the reliability of programs and software, preventing this problem beforehand is crucial. The current technology for APR has been developed using various approaches, which include learning-based, pattern-based, and heuristic-based approach. The overall process of learning-based approach[1, 2] of APR involves inputting a program with bugs, identifying the buggy locations, extracting appropriate data for bug correction, and feeding it into a pretrained deep learning model[3]. Then, candidate patches generated by the model are reviewed, ultimately verifying the correctly repaired code. However, a variety of large language models(LLM) used in APR have an inherent limitation which makes it impossible to repair the entire code of a program due to model's token length limit. Even if the model correctly fixes the buggy line, these patches are classified as incorrect patches because it failed to generate until the end of the code. This paper aims to compensate for this inherent limitation of large language models by effectively lightweighting buggy programs to fit within the model's token limit.

To sum up, we make the following contributions:

- We propose lightweighting the buggy method and patch reconstruction to handle input tokens exceeding the deep learning model's token limit.
- We propose a novel methodology to solve the token length limit issue.
- Our goal is to accurately repair bugs distributed across multiple locations.

2. Related Work

DEAR[4] is a deep learning based approach that supports fixing bugs that require dependent changes at once to one or multiple consecutive statements in one or multiple chunks of code. DEAR used RNN model which does not have a limit on the number of tokens. However, its performance may decrease when a large number of tokens are input. CURE[2] is a NMT-based program repair technique that contains a code-aware search strategy, and a sub-word tokenization technique to create a smaller search space that contains and finds more correct patches. However, since CURE only used data containing fewer than 1024 tokens, it encounters limitations when handling data exceeding the model's token length limit. MCRepair[1] is a learning-based technique that fix multi-chunk bugs by utilizing a buggy block, patch optimization and CodeBERT[3]. However, certain unrepaired bugs in MCRepair were attributed to the CodeBERT's token length limit of 512, which impeded the completion of the method.

3. Methodology

3.1 Token Length Problem

Figure 1 is an example of Chart-7 that has more than one buggy line in a single method. Chart-7 is from Defects4J[5], which is a well-known dataset for Java programs. Figure 1. (a) is a developer patch of Chart-7. To consider a bug as correctly repaired, both buggy lines highlighted in green must be modified from 'minMiddelInddex' to 'maxMiddleIndex'. Figure 1. (b) is an actual example of one of the generated patch of MCRepair[1] by CodeBERT[3]. Even though CodeBERT, which token limit is 512, correctly fixed statements, it resulted in an incomplete patch for the buggy method in Chart-7, which contains 655 tokens. If an input method's token length exceeds the model's token length limit, even with correct repairs, it fails to generate a complete method, thereby yielding an improperly fixed method. Below we will explain our solution for this problem in detail.

3.2 Resolving Token Length Limit

The overall process of APR in this study is illustrated in Fig. 2 In step 1, when a buggy file is inputted, buggy blocks[1] are constructed by extracting methods containing buggy lines in a suitable form for input into the pre-trained large language model[3]. Then, buggy blocks are checked for the token limit and divided into two forms according to the LLM's token limit. If the token length of a buggy method exceeds the model's token limit, it goes through lightweight process outlined in Subsection 3.2.1 and it is called lightweight buggy method(BM). However, if the method's token length falls within the model's token limit, the original buggy block remains unchanged and is referred to as a *short BM*. In step 2, buggy blocks are fed into LLM, which generates candidate methods with buggy lines fixed, known as candidate patches(CP). In step 3, *lightweight CPs* and *short CPs* are candidate patches generated by LLM for *lightweight BM* and *short BM*, respectively. *Lightweight CPs* goes through patch reconstruction process by concatenating with the removed part of its original buggy block.

No.	Java Code
1	private void updateBounds (TimePeriod period, int index) {
…	…
34	if (this.maxMiddleIndex >= 0) {
35	long s = getDataItem(this.maxMiddleIndex).getPeriod() .getStart().getTime();
36	long e = getDataItem(this.maxMiddleIndex).getPeriod() .getEnd().getTime();
37	long maxMiddle = s + (e - s) / 2;
…	…
45	if (this. minEndIndex >= 0) {
46	long minEnd = getDataItem(this.minEndIndex).getPeriod() .getEnd().getTime();
…	…
63	}
	Total: 63 lines, 655 tokens

(a) Developer patch of Chart-7

No.	Java Code
1	private void updateBounds (TimePeriod period, int index) {
…	…
34	if (this.maxMiddleIndex >= 0) {
35	long s = getDataItem(this.maxMiddleIndex).getPeriod() .getStart().getTime();
36	long e = getDataItem(this.maxMiddleIndex).getPeriod() .getEnd().getTime();
37	long maxMiddle = s + (e - s) / 2;
…	…
45	if (this.maxEndIndex >= 0 //End of Generated Patch
	Total: 45 lines, 512 tokens

(b) Actual example of incomplete patch for Chart-7

Figure 1. Example patches for Chart-7

Short CPs move to next step without patch reconstruction because there was no modification during input. Then, these CPs are sent to next step for patch optimization. In step 4, first, CPs that are not considered correctly repaired are filtered based on predefined conditions in patch filtering step. Next, in patch ranking step, filtered CPs are ranked in order of correctness score. Finally, in patch combination step, CPs from multiple methods are recombined in a single file to form patch combinations. After patch combination, the combined patches are validated using testcases and manually verified by developers for their correctness. Below, we will explain each step in detail.

3.2.1 Lightweighting Buggy Method

This section introduces a method for extracting lightweight buggy methods from the original buggy method. If there are multiple buggy lines within a single method, scores are measured for each statement individually and these values are summed to derive the final score. The overall process of lightweighting the method is outlined in Algorithm 1. Firstly, the token limit of the LLM that are going to be used should be identified. Then, the model's token limit is set as the input limit for the model which is TL(token limit) in the algorithm. The purpose for this setting is to avoid the problem of incomplete code generation caused by exceeding the token limit when fixing bugs. After confirming the maximum number of tokens of the model, tokens in the buggy method is checked(line 2). If the total number of tokens contained in a single method does not exceed the maximum token limit of LLM, that single method does not go through the lightweighting process. If the number of tokens in a buggy method exceeds the maximum token length of the model(line 4), we lightweight the buggy method by removing less related lines. To do so, the buggy method is first divided into statements(line 5). If there are more than one buggy line, the calculation process is repeated for each buggy line(line 6). To give scores to each statement we used 2 formulas stated below.

First, the inverse distance(*invd*) between each statement(s) and buggy line(b) is defined as in formula (1) (line 7). $ln(s)$ and $ln(b)$ represents the line number of statement and buggy line, respectively. The buggy line itself get value of 1 and closer the line distance between the statement and buggy line, the higher the value will be close to 1. This formula is uniformly applied when there are multiple buggy lines, by designating one buggy line as buggy line itself and the remaining buggy lines as statements.

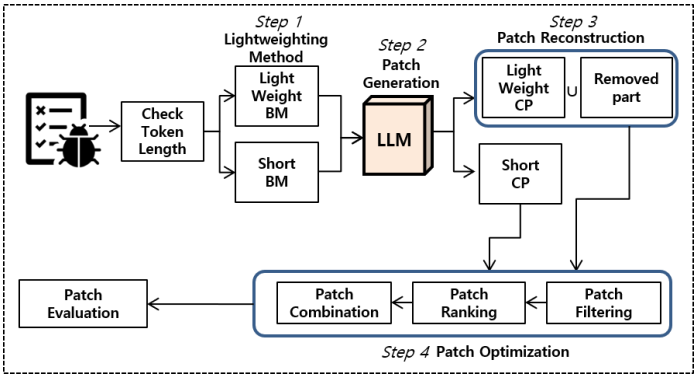

Figure 2. Overview of this study

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

Next, the similarity(*sim*) between a buggy line and the statements in buggy method is defined as the ratio of shared identifiers as in formula (2) (line 8). $IDs(s)$ is a set of identifiers in a statement and $IDs(b)$ is a set of identifiers in a buggy line. If all the identifiers and keywords in buggy line are also found in the statement, the value will reach the maximum value of 1.

$$sim(s,b) = \frac{|IDs(s) \cap IDs(b)|}{|IDs(b)|} \quad (2)$$

After calculating the values of statements for each buggy line, the inverse distance value is multiplied by α, and the similarity value is multiplied by β. The sum of α and β values is 1. Since statement values are calculated for each buggy line, if there are multiple buggy lines, the relatedness value may exceed the maximum value of 1. Therefore, after multiplication by α and β, the values are divided by the number of buggy lines. The relatedness value for each statement is defined as in formula (3) (line 9). This formula is consistently applied among buggy lines.

$$rel(s,b) = \frac{\alpha * invd(s,b) + \beta * sim(s,b)}{number\ of\ buggy\ lines} \quad (\alpha + \beta = 1) \quad (3)$$

Based on the relatedness value, statements with low scores are excluded until overall tokens reach below the token limit of the model(line 11-14). Although the lightweight method provides buggy line with less context to the model than the original buggy method due to the model's token limit, it is expected to maintain the performance. This expectation stems from its ability to offer the most relevant context to the buggy line during bug fixing.

3.2.2 Patch Generation by LLM

After *lightweight BMs* are formed, LLM takes both the *lightweight BMs* and the *short BMs* as input. Then, the LLM generates both *lightweight CPs* and *short CPs* for lightweight BMs and short BMs, respectively.

3.2.3 Patch Reconstruction

In this section, *lightweight CPs* are concatenated with the removed part of the original buggy method. If *lightweight CPs* proceed directly to the patch optimization process, there is a risk that these patches may be considered incorrect during evaluation because they cannot fully replace the original method. Therefore, it is necessary to replace only the buggy lines in original methods with fixed lines in lightweight CPs. The overall process of patch reconstruction is outlined in Algorithm 2. First, we compare the *lightweight BMs* and *lightweight CPs* to identify the fixed parts(line 1). We identify actions(insertion, update, and deletion) performed by CodeBERT using GumTree[6]. After determining which action fixed buggy lines, we replace buggy lines of the *lightweight BMs* with the fixed lines in the *lightweight CPs*(line 3) and get modified *lightweight BMs*(line 4). Then, the final patch reconstruction is done by comparing buggy lines in the original buggy method with the *lightweight BMs*(line 6) and replacing them with lines fixed by the model(line7). This process is possible because both the original buggy method's buggy block and the *lightweight BM*'s buggy block share the same bug tokens and buggy line is located at between two bug tokens. To clarify the terms

Algorithm 1: Extracting Lightweight Buggy Method

Input: BM: Buggy Method, TL: Model token limit,
BL: Buggy line
Output: Lightweight BM: Buggy Method

```
lightweightBM ← BM
tokensCount ← CountTokens(BM)

if tokensCount > TL then
    BMSet ← BreakBMIntoStatements(BM)
    for each BL in BMset do
        CalculateInverseDistanceScore(BL, BMSet)
        CalculateSimilarityScore(BL, BMSet)
        Sum ScoreStatement(BMSet) / Count(BL)
    end for
    while tokensCount > TL do
        DeleteLowestScoreStatement(BMSet)
        tokensCount ← CountTokens(BMSet)
    end while
        lightweightBM ← JoinStatements(BMSet)
end if
return lightweightBM
```

used in Algorithm 2, the characteristics of CPs and BMs are stated in Table 1. Except for *lightweight CPs*(lwCP), all other BMs and CPs have bug tokens that help identify buggy lines. Next, *lightweight CPs*(lwCP), *lightweight BMs*(lwBM'), and *reconstructed CPs*(rcCP) all have fixed lines because they are the updated versions of the buggy method, with the buggy lines replaced by the fixed lines of the *lightweight CPs*. Lastly, *lightweight BMs*(lwBM) and original buggy block(BB) contain buggy lines.

3.2.4 Patch Optimization and Evaluation

After the patch reconstruction, both *reconstructed CPs* and *short CPs* proceed to the final step: patch optimization and patch evaluation. Patch optimization consists of three main steps[7]. Firstly, in the patch filtering stage, duplicated patches are removed, and any patches containing syntax errors or including termination code are eliminated. Next, in the patch ranking stage, the highest possibility of correctness for each patch is measured by applying action similarity and n-gram similarity and they are ranked based on their values. Lastly, in the patch combination stage, candidate patches for each method in a single Java file are combined. After patch optimization, patches proceed to patch validation stage, where the combined patches are validated with test cases and manually checked by developers for correctness.

4. Case Study

Figure 3 is an example of the lightweight BM of Chart-7 buggy method. After applying the algorithm in Subsection 3.2.1, the number of lines was reduced from 63(Fig. 1 (a)) to 35(Fig. 3) and the total tokens decreased from 655(Fig. 1 (a)) to 504(Fig. 3). The highlighted lines in Fig. 3 represent two buggy lines in the method. The relatedness score for line 37 is calculated as follows: Line 37 first obtains its value by calculating its relatedness with buggy line 35. The inverse distance score is 0.33 due to a two-line distance from the buggy line while the similarity score is 0.16. This is because among the identifiers in line 37, only one identifier 's' matches with those in line 35. The same process is repeated for line 37 and buggy line 36, resulting in an inverse distance score 0.5 and a similarity score 0.16. Then, the inverse distance scores are multiplied by α and β, both set to 0.5. After multiplication, the result is divided by 2 which is the total number of buggy lines. Consequently, the relatedness score of line 37 is calculated as 0.287 and rounded to 0.29. Based on the relatedness value, statements that received low scores are deleted until the total token count is less than the model's token length limit. Then, the remaining statements and buggy lines are concatenated to form a *lightweight BM*. We can find out that lines, including those indicating method names, which were far from the buggy lines, have been removed due to their low distance score. Also, lines that share a lot of similarity with buggy lines but are located at a long distance are not deleted. Through this approach, we expect to correctly fix buggy lines by utilizing *lightweight BM*.

5. Conclusion

In this study, our goal is to create complete candidate patches for buggy methods by applying a lightweighting technique and patch reconstruction. First, original buggy methods are tailored to fit the token

Algorithm 2: Patch Reconstruction

Input: lwCP: lighweight CPs, lwBM: lightweight BMs,
lwBM': lightweight BMs with fixed line,
BB: original buggy block
Output: rcCP: reconstructed CPs

```
Compare(lwCP, lwBM)
if lwCP != lwBM
    lwBM' ← ReplaceBLWithFL(lwBM, lwCP)
    return lwBM'

CompareBugToken(lwBM', BB)
    rcCP ← ReplaceBLWithFLines(lwBM', BB)
    return rcCP
```

*BL: bug line, FL: fixed line

Table 1. Characteristics of BM and CP

	Bug Token	Fixed Line	Buggy Line
lwCP	X	O	X
lwBM	O	X	O
lwBM'	O	O	X
BB	O	X	O
rcCP	O	O	X

No.	Java Code	rel(s,b)
2	long start = period.getStart().getTime();	0.14
3	long end = period.getEnd().getTime();	0.14
6	long minStart = getDataItem(this.minStartIndex) .getPeriod().getStart().getTime();	0.30
…	…	…
34	if (this.maxMiddleIndex >= 0) {	0.20
35	long s = getDataItem(this.minMiddleIndex) .getPeriod().getStart().getTime();	0.79
36	long e = getDataItem(this.minMiddleIndex) .getPeriod().getEnd().getTime();	0.79
37	long maxMiddle = s + (e - s) / 2 ;	0.29
…	…	…
61	this.maxEndIndex = index;	0.02
Total: 35 lines, 504 tokens		

Figure 3. Example of lightweight buggy method of Chart-7

length constraints of deep learning models which include context statements highly related to buggy lines. Then, the buggy lines in the original buggy blocks are replaced with fixed lines from the lightweight versions, composing a complete candidate patch. This approach aims to prevent incomplete patch generation due to the token length limit of the model, thereby avoiding incorrect evaluation and ultimately expecting to fix more multi-location bugs.

Acknowledgement

This work was supported by the National Research Foundation of Korea (NRF) grant funded by the Korea government (MSIT) (NO.2022M3J6A1084845).

References

[1] J. Kim and B. Lee, "MCRepair: Multi-Chunk Program Repair via Patch Optimization with Buggy Block," in Proc. of the 38th ACM SIGAPP Symposium on Applied Computing (SAC), 2023, pp. 1508–1515.
[2] N. Jiang, T. Lutellier, and L. Tan, "CURE: Code-Aware Neural Machine Translation for Automatic Program Repair," in Proc. of the 43rd IEEE/ACM International Conference on Software Engineering (ICSE), 2021, pp. 1161–1173.
[3] Z. Feng et al., "CodeBERT: A Pre-Trained Model for Programming and Natural Languages," in Findings of the Association for Computational Linguistics: EMNLP, 2020, pp. 1536–1547.
[4] Y. Li, S. Wang, and T.N. Nguyen, "DEAR: a novel deep learning based approach for automated program repair.," in Proc. of the 44th IEEE/ACM International Conference on Software Engineering (ICSE), 2022. pp. 511-523.
[5] https://github.com/rjust/defects4j
[6] J.R. Falleri et al., "Fine-grained and accurate source code differencing," in Proc. of 29th ACM/IEEE International Conference on Automated Software Engineering, 2014, pp. 313-324.
[7] A. Safarovich, J. Kim, and B. Lee, "Improving Patch Optimization for Multi-Chunk Bugs in Automated Program Repair," in Proc. of the Korea Computer Congress, 2023, pp. 200-202.