J Inf Process Syst, Vol.20, No.5, pp.627~639, October 2024
https://doi.org/10.3745/JIPS.04.0320
ISSN 2092-805X (Electronic)

JOURNAL OF INFORMATION PROCESSING SYSTEMS JIPS

An Enhanced Patch Optimization Technique for Multi-Chunk Bugs in Automated Program Repair

Abdinabiev Aslan Safarovich, Jisung Kim, and Byungjeong Lee*

Abstract

Automated program repair techniques leveraging deep learning have shown remarkable performances in bug repair. These techniques commonly employ pre-trained neural machine translation (NMT) models to generate patches for a buggy part of the source code. However, when dealing with multiple buggy code chunks in various locations, current methods face challenges in effectively selecting and combining these patches for optimal repair. This paper identifies limitations within one of the existing methods used for optimizing patches related to multiple buggy code chunks and proposes an enhanced patch optimization technique to address these shortcomings. The primary aim of this study is to improve the process of selecting and combining patches generated for groups of buggy chunks. Through experiments conducted on a dataset, this paper demonstrates the efficacy of the proposed patch optimization technique, showcasing its potential to enhance the overall bug repair process. This study highlights the importance of patch optimization in bug repair by addressing limitations and enhancing the repair process.

Keywords

Automated Program Repair, Machine Learning, Multi-Chunk Bugs, Patch Optimization

1. Introduction

Automated program repair (APR) has emerged as a pivotal area in software engineering, leveraging advanced techniques to identify and fix bugs in source code automatically. Learning-based approaches, particularly those utilizing neural machine translation (NMT) models, have demonstrated significant promise in this domain. These approaches often use deep-learning models to generate patches that amend one or several buggy sections of code.

However, despite these advancements, current APR techniques face significant challenges when dealing with multiple buggy chunks spread across different parts of the code. Researchers have proposed different methods to address such types of bugs. For example, one of the first multi-chunk repair techniques, HERCULES [1], identifies similar buggy location siblings first and then applies a repair scheme to the identified siblings. However, it tried to fix the multi-chunk bugs where the buggy chunks require the same fix pattern. Next, Recoder [2] trained the NMT model using the abstract syntax tree (AST) of the buggy code to generate edit sequences in a specified form. CURE [3] applied ensemble

Manuscript received February 15, 2024; first revision July 7, 2024; accepted September 9, 2024.
* **Corresponding Author:** Byungjeong Lee (bjlee@uos.ac.kr)
Dept. of Computer Science, University of Seoul, Seoul, Korea (aslan@uos.ac.kr, kimjisung78@uos.ac.kr, bjlee@uos.ac.kr)
Current affiliation for author, Jisung Kim, is Team of Alternative Service, Department of General Affairs, Daejeon Correctional Institution, Daejeon, Korea

learning with several models and a code-aware strategy that contains valid-identifier-check and length-control check tasks. Recently, some hybrid APR approaches have also been proposed, such as GAMMA [4] and RAP-Gen [5], which combine template-based and learning-based methods. SelfAPR [6] used self-supervised learning by generating the training samples from historical commits from the same buggy project. These techniques mainly focus on single-chunk or single-method bugs or do not provide information about the whole process of how they combine the subpatches in the case of multi-chunk bug fixing. However, to fix the multi-chunk bugs that spread throughout the program, APR techniques can fix the bugs by generating subpatches (the term "subpatch" in this paper indicates a candidate patch for some part of the buggy source code) per buggy chunk or group of buggy chunks.

Therefore, after generating the subpatches per buggy chunk or group of buggy chunks, effectively constructing the final patches using these subpatches is important.

One of the multi-chunk APR approaches was proposed by Kim and Lee [7], which is based on generating numerous candidate subpatches for a method-level group of buggy chunks using their buggy block preprocessing method and a fine-tuned CodeBERT [8], and combining them to make a set of final patches. However, the exponential increase in the patch space with the number of combinations makes it impractical to combine all the generated subpatches. Therefore, in [7], a patch optimization step was used to filter out some patches, rank, and combine a certain number of the most correct candidate subpatches. However, our observation shows that this optimization approach has some limitations, especially in the patch ranking and combination phases. These limitations notably decrease the overall program repair performance of the approach.

In this paper, we propose our enhanced patch optimization method by briefly explaining the detected limitations and proposing our solutions for them. We conducted an experiment on Defects4j [9], one of the popular benchmark datasets among APR researchers, and the results demonstrated the effectiveness of our proposed improvements.

The rest of the paper is organized as follows. Section 2 introduces background information on our study. In Section 3, we present our methodology, discussing the identified limitations and our proposed solutions. Section 4 outlines the experiments conducted and presents the results. Finally, we conclude the paper in Section 5.

2. Background

2.1 Bug Types

```
public int findMax (int [] arr) {
    int max = Integer.MIN_VALUE;
        for (int num : arr) {
–       if (num < max) {
+       if (num > max) {
–           max = max;
+           max = num;
        }
    }
    return max;
}
```

(a)

```
public int findMax(int[] arr) {
–   int max = 0;
+   int max = Integer.MIN_VALUE;
        for (int num : arr) {
–       if (num < max) {
+       if (num > max) {
–           max = max;
+           max = num;
        }
    }
    return max;
}
```

(b)

Fig. 1. Examples of (a) a single-chunk bug and (b) a multi-chunk bug ("–" and "+" indicate deleted and added lines).

Software bugs can be classified into single-chunk and multi-chunk bugs according to their complexity. A single-chunk bug refers to one or more consecutive buggy lines in a single place of the source code. In contrast, a multi-chunk bug contains two or more single-chunk bugs. Here a "buggy line" denotes a source code line that requires updating (changing some parts of the line), deletion (fully removing from the code) or insertion (adding some new lines before the line). Fig. 1 shows examples of both single-chunk and multi-chunk bugs. In Fig. 1(a), we can see a buggy method findMax, where the buggy lines “if (num<max) {” and “max=max” are located in a single place making a single buggy chunk. While, in Fig. 1(b) shows two buggy chunks separated with a correct line “for (int num: arr) {”.

2.2 Related Work

Multi-chunk bugs are complicated because of several problems, such as interdependency, large patch space, etc. Different APR techniques use different approaches to fix multi-chunk bugs. For example, HERCULES [1] identifies similar buggy location siblings first and then applies a repair scheme for the identified sibling. However, it applies the same repair pattern for each location of the siblings, therefore it will have trouble when it encounters a multi-chunk bug that requires different repair patterns. CURE [3] applied ensemble learning with several models and code-aware strategy that contains valid-identifier-check and length-control check tasks. It provides fixed multi-chunk bug results but does not provide its detailed patch optimization process. Recoder [2] applied AST to NMT to generate edit sequences in a specified form. And it used three encoders for three different purposes, such as learning AST of the buggy method, learning the edit sequences and an AST path. Recoder does not have a patch combination step and fixed a small number of multi-chunk bugs. Hybrid approaches GAMMA [12] and RAPGen [5] combined template-based and learning-based approaches. However, they are mainly focus on the single chunk repair and did not provide details when the multi-function repair where the generated patches should be combined. SelfAPR [6] trained the deep-learning model with the training data generated by perturbing the historical version of the same project that is under fix. However, it fixed only single-chunk bugs and could not fix the multi-location bugs.

3. Methodology

3.1 Overall Multi-Chunk Program Repair architecture

Fig. 2 shows an overall multi-chunk program repair architecture of this study. The input is a buggy program that can have several buggy methods (BMs) and fields. First, BMs are determined and extracted with their related information. Then, buggy blocks are made using the buggy block preprocessing method [7] (cn is the number of buggy blocks). Each buggy block is a concatenation of the BM, its related context, and the bug marking tags (<bug></bug> is used to indicate buggy line, <context></context> is used to indicate context information, etc.). Then, in the fine-tuning stage, a model (i.e., CodeBERT [8]) is fine-tuned with a large dataset of buggy & fixed project pairs, where the made buggy block is a source and the fixed method (FM) is a label. In generation stage, the fine-tuned CodeBERT takes these the buggy blocks, which are made from the buggy program that is under test, and generates *SP* candidate subpatches for each of them ($cd(i,j)$ denotes the j-th candidate subpatch for i-th buggy block). Then, a patch optimization is applied to the generated subpatches through three steps: patch filtering, patch ranking,

and patch combination. After patch combination phase is finished, the created patches are evaluated. First, they are checked for compilability, then the compilable patches are checked for plausibleness using their test cases (a patch is called plausible if it passes all of the test cases), and for correctness by developers individually.

3.2 An Enhanced Patch Optimization

After generating a massive number of subpatches for each buggy block using the fine-tuned CodeBERT, there is a need for a phase that selects the most likely correct ones and combines them. Patch optimization strategy used in [7] does this job in three steps: patch filtering to filter out clearly incorrect subpatches, patch ranking to rank the subpatches according to their suspiciousness of being correct, and patch combination to combine the certain number of them from the top (Fig. 2).

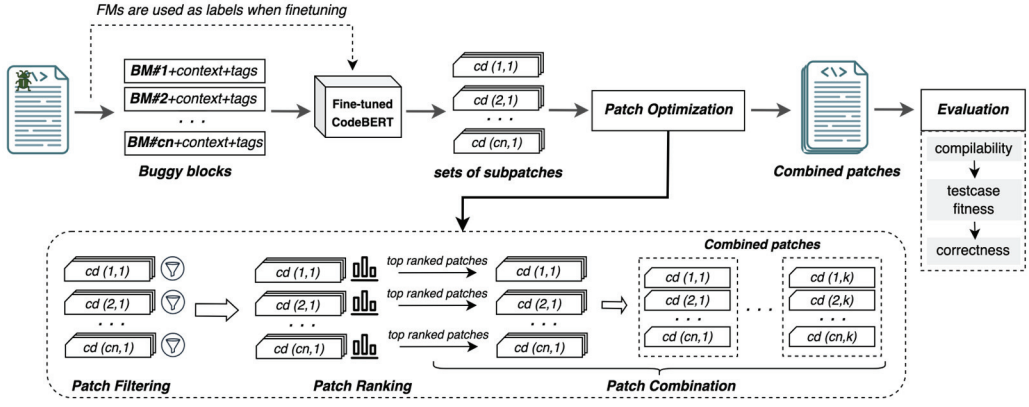

Fig. 2. Overall architecture of multi-chunk program repair.

3.2.1 Patch filtering

This step is used to filter out unnecessary patches that are clearly not a correct patch. The work [7] filters out the patches that are duplicated (DP), have syntax errors (SE), or include termination code (i.e., System.exit), which are determined using a code parser library.

However, after analyzing the patches generated by the fine-tuned model, we realized a significant number of patches that are created just by adding java standard stream codes (i.e., System.out, System.in and System.err). And these patches are often syntactically closer to the buggy code compared to a correct patch, which raises the probability of not using the correct patch in combination. In addition, these stream codes are mostly not used to fix the software bugs. Therefore, in this paper we add a new patch filtering rule to determine such patches and remove them to reduce patch space. First, we will count the numbers of standard inputs, outputs, and errors in a buggy code that we denote as N_{in}, N_{out}, and N_{err}, respectively. Then we count the corresponding numbers in a patch. If any number in the patch is bigger than that of the buggy code, we remove the patch as it includes additional input, output, or error streams.

Our model was fine-tuned with Java dataset since the target language of our study is Java. So, our heuristic was developed based on empirical observations from our specific context and Java language. However, it is expected that the proposed method is able to handle program bugs written in other

languages if we add filtering heuristic specific to the languages and fine-tune the model with datasets of the languages.

3.2.2 Patch ranking

After filtering out unnecessary patches, this step is used to rank remaining patches according to their probability of being correct. Since checking the correctness of subpatches using testcases is impossible, they are ranked according to their syntactic and semantic similarity with the buggy code. The work [1] applied weighted sum of two ranking measures: Action similarity and N-gram similarity (Eq. 1):

$$Sim(cd(i,j)) = \alpha * S_{act}(cd(i,j)) + \beta * S_{ngram}(cd(i,j)) \quad (1)$$

where S_{act} is an action similarity of $cd(i,j)$; S_{ngram} is an n-gram similarity of $cd(i,j)$; α and β are relative correction factors.

Action similarity: Fig. 3(a) presents a buggy method, a developer provided patch and one of the correct candidate patches. The similarity considers the difference between the expected and performed number of actions, as a correct patch tends to minimally change its buggy code [10]. To determine the number of the expected actions, the work [7] takes "Repaired locations" from the developer-provided patch. At "Repaired locations," it calculates the numbers of expected insertions that a location indicates "FAULT_OF_OMISSION" (Act_{expI}) and the other update or deletion actions (Act_{expO}). Then, it calculates the numbers of performed insertions (Act_{perI}) and other actions (Act_{perO}) using an AST difference library. Finally, it calculates the overall similarity score by obtaining the minimum value between expected and performed actions, and penalizing the distance between them:

$$S_{act} = (minTotal + p \times disTotal) \,/\, (minTotal + disTotal) \quad (2)$$

where, $minTotal = min(Act_{expI}, Act_{perI}) + min(Act_{expO}, Act_{perO})$; $disTotal = |Act_{expI} - Act_{perI}| + |Act_{expO} - Act_{pero}|$; p is a penalty $(0 < p < 1)$.

However, our observations showed that the work has some issues with determining the repaired locations. The first problem is related to divided locations. When the buggy location is divided into two parts and both parts were updated in the developer provided patch as locations 4 and 5 in Fig. 3, the work [7] considers this as two expected updates. However, even if a candidate patch fixes these buggy locations, the AST difference library returns a single performed update, and this difference causes the similarity score to drop. Moreover, according to Yang et al. [11], the developer patches in the benchmark datasets have many irrelevant changes. In Fig. 3(a), the return statement added before line 8 can be considered as irrelevant, because the second return statement added before line 9 is enough for the bug to be fixed and there is no need for another return statement. And these irrelevant changes cause a fake increase in the number of expected actions.

The work [7] does not account for such cases. To solve the problems, we first determine the divided locations using a code formatter tool and consider them as a single location. Then, we determine irrelevant buggy locations using a dataset purification technique proposed in [11] and remove them from the "Repaired locations." The tables on the right side of Fig. 3(b) and Fig. 3(c) show values related to *Repaired Locations#1* and *Revised Repaired Locations#1* of our example, respectively. Due to the differentiation, Act_{expI} and Act_{expO} are not 2 and 2 as in Fig. 3(b), but 1 and 1 as in Fig. 3(c),

respectively. And $cd(1,1)$ has a higher action similarity using *Revised Repaired Locations#1* instead of *Repaired Locations#1*.

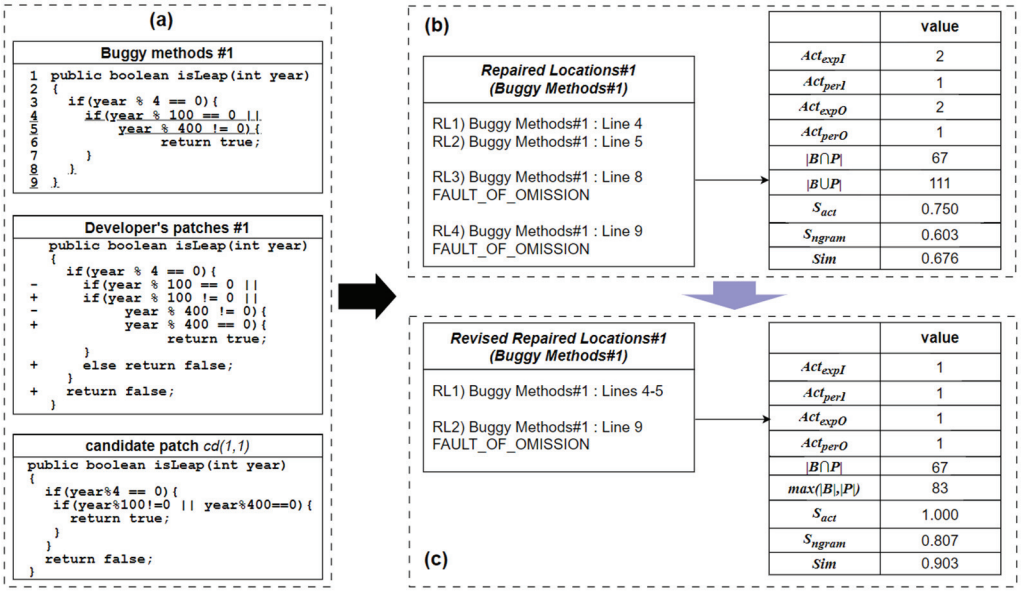

Fig. 3. The details of patch ranking improvements. (a) An example of buggy method, its developer-provided patch and a correct candidate patch, (b) calculation of the similarity score before the revisions, (c) calculation of the similarity score after the revisions, RL = repaired location.

N-gram similarity. The similarity considers the structure and syntax changes between a candidate patch and its buggy code because a correct patch has a similar structure and syntax compared to its buggy code [10]. The work [7] applied the Jaccard formula to the similarity. However, after analyzing some correct candidate patches, we observed that these patches are getting relatively low similarities because the formula uses the union denominator. Therefore, to solve the problem, we replaced the denominator of the equation to max to increase the score (Eq. 3):

$$S_{ngram} = |B \cap P| \,/\, max(|B|, |P|), \tag{3}$$

where, B and P are the sets of n-gram tokens from a buggy chunk and its candidate patch, respectively.

By changing the n-gram similarity, we focus more on the modification bugs with smaller changes. In our example, $cd(1,1)$ has a higher n-gram similarity using the new denominator instead of the previous one (Fig. 3).

3.2.3 Patch combination

The final step is to select the top certain number of patches from each ranked set of candidate patches and combine them to build final patches. The work takes k candidate patches from each set and combines them, where k is calculated based on the number of buggy blocks and maximum number of combined patches using Eq. (4):

$$k = max\ (\{n \mid n^{cn} \leq M,\ \ n \leq SP\}) \tag{4}$$

where, cn is the number of buggy blocks; M is the maximum number of combined patches; SP is the number of generated patches for a buggy block.

However, this combination approach has two problems. Firstly, k is the same for all the candidate sets regardless of the complexity of their buggy blocks, which causes to take a relatively small number of subpatches for buggy blocks that have many buggy locations for combination. Secondly, if the number of buggy blocks increases, the number of the combined patches and k values decrease exponentially. For example, if the number of buggy blocks $cn = 5$ and the maximum number of combined patches $M = 10000$, then $k = 6$ and using it we can have $6^5 = 7776$ combined patches which is 2224 less than desired number of combined patches. If the number of buggy blocks is increased by 1 and $cn = 6$, then $k = 4$ and we have $4^6 = 4096$, which is not even half of the desired number of combined patches.

To solve the first problem, we calculate separate k value for each candidate set according to the number of buggy locations in its buggy block using Eq. (5):

$$k_i = max\ (\{n \mid n^{cn} \leq \frac{M \times {bLocs_i}^{cn-1}}{\prod_{j=1}^{cn} bLocs_{j,\ j \neq i}}, \quad n \leq SP\}), \tag{5}$$

where, $bLocs_i$ is the number of buggy locations in i-th buggy block.

Using this equation, we generate $initialtopKs = (k_1, k_2, \dots, k_{cn})$, a list of k values for each candidate set and these k values change proportionally to the number of buggy locations in their buggy blocks.

We can solve the second problem by incrementing some k values in $initialtopKs$. However, as we have too many increment options, we use Algorithm 1 to find the most optimal one.

Algorithm 1. Get optimal $topKs$

```
    INPUT:
    M, multiplier, initialtopKs, SP
1   function getOptimalTopKs(M, SP, initialtopKs, multiplier):
2       mainList ← []
3       for each k from initialtopKs do:
4           product ← (∏_{j=1}^{kl} initialtopKs(j)) / k   // calculate a product of the other k values
5
6           kIncrementsList ← []
            iterator ← k
7           while True do:
8               add iterator to kIncrementsList
9               if iterator * product ≤ M then
10                  if iterator + 1 ≤ min (k * mutiplier, ceil(SP * k/sum(initialtopKs)))
11                      increment iterator
12                  else
13                      break
14              else
15                  break
16          add kIncrementsList to mainList
17      combinations = makeCombinations(mainList)
18      filteredcombinations = filterCombinations(M, combinations)
19      sortedcombinations = sortCombinations(filteredcombinations)
20  return the first item of sortedcombinations
```

Algorithm 1 receives $initialtopKs$ with some additional parameters and returns $topKs$, the most optimal k values. First, the algorithm generates a list of all applicable incremented values for each k value in $initialtopKs$ using lines from 3 to 16. Next, we make combinations from the taken increments to consider all options. Here, applicability of each increment is determined using the line 9, which checks whether the number of combined patches does not exceed M, and line 10, which checks for an upper limit to prevent a memory related error. The memory error occurs when the number of buggy blocks is too large, and each block has a small number of buggy locations. For example, if the number of buggy blocks $cn = 20$ and each block has a single location, then the calculated k value is 1 for all the candidate sets (i.e., $initialtopKs = (1,1,\dots,1)$) and we can increment each k up to 10000 without the upper limit. So, the system will have to create 10000^{20} combinations which causes to raise the memory error. After obtaining the combinations, we exclude the combinations whose product of items (number of combined patches) exceed M using line 18. Next, we sort the left combinations according to the product and standard deviations (std) of items to return the combination with the maximum product and the minimum std values. We are taking the combination with minimum std value to avoid too small and too large values. The returned combination, which is $topKs$, defines the number of subpatches that is taken from each candidate set. Using the obtained $topKs$, we combine the subpatches and generate a list of final combined patches. Finally, the combined patches are checked for correctness.

In this algorithm, we assess the complexity of the buggy method in a naive way according to the number of buggy locations. However, sometimes a bug with a single buggy location can be more difficult to fix than a bug with several buggy locations. In this scenario, the algorithm can take a smaller number of subpatches for a single location bug. If we had a better way to assess the complexity of bugs, the algorithm would perform better.

4. Experiments

4.1 Experiment Preparation

We conducted an experiment using two datasets: Bugs2Fix [12] dataset was used for training and validation purposes when fine-tuning the CodeBERT and we evaluated the proposed improvements using Defects4J [9], which is a well-known benchmark dataset among researchers. The Bugs2Fix dataset is a large collection of Java buggy and fixed code pairs. It provides approximately 787k raw buggy and fixed source code pairs that were mined from GitHub bug-fix commits. Additionally, some of this source code was converted into buggy-fixed method pairs, which come in two versions: small, containing around 58k pairs, and large, containing 65k pairs. We used the raw buggy and fixed code pairs in our training data.

There are two older and newer versions of Defects4J, and we used the first version, which has 6 modules (Table 1).

Program repair was performed under perfect fault localization using actual buggy lines provided by developers. Training and evaluation are performed on a 16-core server with Ubuntu 18.04 LTS, Docker environment, 512 GB RAM, one NVIDIA RTX A6000 GPU and two Intel Zeon Gold 6226R 2.9 GHz CPUs. JavaParser [13] and Spoon [14] libraries were used for extraction of buggy method and fields. Fault localization information was obtained using Gumtree [15] difference library from the training dataset. The CodeBERT was implemented using HuggingFace [16] and PyTorch [17]. The time-out per

bug was set to 5.5 hours. We implemented our work using Java 8 and 11, and Python 3.x programming languages.

Table 1. Bugs in Defects4J

Modules	Chart	Closure	Lang	Math	Mockito	Time	Total
Number of bugs	26	131	65	106	38	27	393

The most complex phases of our technique are the patch generation and patch optimization phases. The complexity of the patch generation is directly related to the CodeBERT and its parameters, and you can refer to [7] for more information. The complexity of our enhanced patch optimization can be calculated based on the complexity of the new patch combination step, which is:

$$O(\ cn^2\ *\ max(cn,\ max(\{k \mid k_i = \frac{M}{\prod_{j=1}^{\boldsymbol{cn}} \boldsymbol{initialtopKs(j),\ j \neq i}} - \boldsymbol{initialtopKs(i)}\}))\ ).$$

where $\boldsymbol{initialtopKs}$ is calculated using Eq. (5), and $\boldsymbol{cn}$ is the number of buggy blocks.

$\boldsymbol{Act_{perI}}$ and $\boldsymbol{Act_{perO}}$ in Eq. (2) were also detected using the Gumtree. α, β in Eq. (1) and p in Eq. (2) were set to 0.5. SP was 500, M in Eq. (4) and Eq. (5) was set to 10,000 and $\boldsymbol{multiplier}$ in Algorithm 1 was set to 2. N-gram similarity was measured using tri-gram (3-gram).

We compare our study with the work [7] and two other learning-based techniques: CURE [3] and Recoder [2]. We did not choose HERCULES [1] as our baseline, because it does not provide data for perfect fault localization.

We released the enhanced patch optimization related source code and the generated patches publicly available in the following link: https://github.com/Aslan7197/enhancedPatchOptimization.git. To find the overall process, you can refer to [7].

4.2 Research Questions

RQ1: How does the proposed approach perform against its baselines?

RQ2: How do the proposed improvements contribute to performance?

4.3 Experiment Results

RQ1: How does the proposed approach perform against its baselines?

Table 2 demonstrates the number of bugs fixed by our study and the other baselines in each module of Defects4J under perfect fault localization. As we can see in the table, the program repair technique including our enhanced patch optimization was able to fix a total of 79 bugs which is 14 more than the work [7] that uses simple patch optimization, 22 more than CURE, and 15 more than Recoder. Here, 18th bug of Math module was excluded from the fixed bugs of the work [7]. Because it did not check its encoding boundaries and was incorrect.

Overall, our study outperformed the baselines on all the modules besides Chart and Closure, where the results were the same in Closure. We showed the highest improvement in Math module by repairing 7, 11, and 12 more bugs than the work [7], CURE, and Recoder, respectively. Fig. 4 demonstrates a Venn diagram for Table 2, which shows the fixed bug ids by each baseline. Here, 7 unique bugs were fixed by only our study, where 5 bugs are multi-chunk including CL_6, L_20, M_43, M_62, and M_86.

Table 2. Comparison of the works

Technique	Chart	Closure	Lang	Math	Mockito	Time	Total
Kim and Lee [7]	5	**21**	9	23	4	3	65
CURE [3]	**10**	14	9	19	4	1	57
Recoder [2]	**10**	**21**	10	18	2	3	64
This work	6	**21**	**12**	**30**	**6**	**4**	**79**

Numbers in bold indicate the largest number in the column.

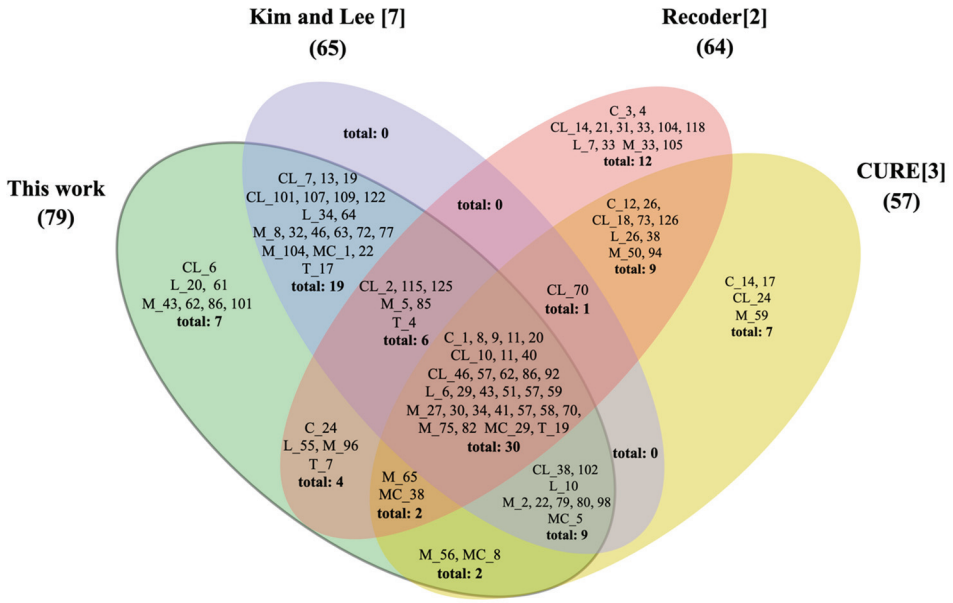

Fig. 4. Venn diagram for Table 2. C = Chart, CL = Closure, L = Lang, M = Math, MC = Mockito, T = Time.

RQ2: How do the proposed improvements contribute to performance?

To answer the question, we analyze the comparison results of our study with the work [7]. Overall, the program repair technique using our enhanced patch optimization phase was able to fix 14 more bugs than the work [7] that uses simple patch optimization (a relative improvement of 21.5%). Furthermore, we fixed 15 unique bugs, where 1, 3, 7, 2, and 1 additional bug in Chart, Lang, Math, Mockito, and Time modules, respectively. Among the uniquely fixed bugs, 8 bugs are multi-chunk bugs.

By applying the new patch filtering rule, we removed another type of unnecessary subpatch that was causing issues in selecting the correct one. We also improved the patch ranking process, making it better at giving higher scores to the most accurate subpatches. Additionally, our updated patch combination method now considers the difficulty of each subpatch, leading to more accurate combinations. These improvements helped us fix more multi-chunk bugs.

However, in the process of repairing additional multi-chunks, the program repair technique failed to fix some single-chunk bugs, including CL_70. The reason for this is related to the patch generation process, and specifically it can be related to the insufficient data in the training dataset for single-chunk bugs. If we train the model with additional dataset for single-chunk bugs, we can expect better repair performance.

5. Conclusion

In this paper, we propose an enhanced patch optimization technique to optimize the patches generated by a deep learning-based bug repair technique for buggy code with multi-chunk bugs. This approach makes several improvements to the work in [7], aiming to increase its performance and applicability. First, we added a new patch filtering rule to remove more unnecessary subpatches from the subpatch space. Then, we made improvements in the patch ranking phase to enhance its ranking quality. Finally, we proposed a new patch combination method to combine patches more effectively by considering the bug difficulty. We conducted an experiment on the Defects4J dataset and fixed a total of 79 bugs, showcasing a 21.5% relative improvement in overall program repair performance. Furthermore, we compared our study with two other related APR techniques and demonstrated its superiority.

However, our study failed to fix certain types of single-chunk bugs that were fixed by the other baseline techniques. One of the main reasons for this was the insufficient fix patterns in the training dataset. If we train the model with additional datasets for more varied bugs, including single-chunk bugs, we can expect better performance. Furthermore, another main problem of our study is the token limitation of CodeBERT. If the number of tokens in a buggy block or a generated candidate subpatch exceeds the token limit of the model, CodeBERT generates incomplete subpatches, even if it converts the buggy chunks into fixed ones. If we find an effective way to resolve the token limit problem of the model, we can expect a significant increase in the number of fixed bugs. Finally, we used a heuristic approach by counting the buggy lines to determine the complexity of the bug in the patch combination phase. Therefore, in the future, we plan to focus on expanding the training dataset to encompass a broader range of bug types, developing a solution for the model's token limit problem, and finding a more effective measure to determine the complexity of the buggy method, as well as improving other aspects of the program repair architecture.

Conflict of Interest

The authors declare that they have no competing interests.

Funding

This work was supported by the 2023 Research Fund of the University of Seoul.

References

[1] S. Saha, R. K. Saha, and M. R. Prasad, “Harnessing evolution for multi-hunk program repair,” in *Proceedings of 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, Montreal, Canada, 2019, pp. 13-24. https://doi.org/10.1109/ICSE.2019.00020

[2] Q. Zhu, Z. Sun, Y. A. Xiao, W. Zhang, K. Yuan, Y. Xiong, and L. Zhang, “A syntax-guided edit decoder for neural program repair,” 2022 [Online]. https://arxiv.org/abs/2106.08253v6.

[3] N. Jiang, T. Lutellier, and L. Tan, "CURE: code-aware neural machine translation for automatic program repair," in *Proceedings of the 43rd IEEE/ACM International Conference on Software Engineering (ICSE)*, Madrid, Spain, 2021, pp. 1161-1173. https://doi.org/10.1109/ICSE43902.2021.00107

[4] Q. Zhang, C. Fang, T. Zhang, B. Yu, W. Sun, and Z. Chen, "Gamma: revisiting template-based automated program repair via mask prediction," in *Proceedings of 2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, Luxembourg, 2023, pp. 535-547. https://doi.org/10.1109/ASE56229.2023.00063

[5] W. Wang, Y. Wang, S. Joty, and S. C. H. Hoi, "Rap-Gen: retrieval-augmented patch generation with CodeT5 for automatic program repair," in *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, San Francisco, CA, USA , 2023, pp. 146-158. https://doi.org/10.1145/3611643.3616256

[6] H. Ye, M. Martinez, X. Luo, T. Zhang, and M. Monperrus, "SelfAPR: self-supervised program repair with test execution diagnostics," in *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, Rochester, MI, USA, 2022, pp. 1-13. https://doi.org/10.1145/3551349.3556926

[7] J. Kim and B. Lee, "MCRepair: multi-chunk program repair via patch optimization with buggy block," in *Proceedings of the 38th Annual ACM/SIGAPP Symposium on Applied Computing (SAC)*, Tallinn, Estonia, 2023, pp. 1508-1515. https://doi.org/10.1145/3555776.3577762

[8] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, et al., "CodeBERT: a pre-trained model for programming and natural languages," in *Proceedings of Findings of the Association for Computational Linguistics EMNLP*, Virtual Event, 2020, pp. 1536-1547.

[9] R. Just, D. Jalali, and M. D. Ernst, "Defects4J: a database of existing faults to enable controlled testing studies for Java programs," in *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, San Jose, CA, USA, 2014, pp. 437-440. https://doi.org/10.1145/2610384.2628055

[10] S. Wang, M. Wen, B. Lin, H. Wu, Y. Qin, D. Zou, X. Mao, and H. Jin, "Automated patch correctness assessment: how far are we?," in *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, Virtual Event, Australia, 2020, pp. 968-980. https://doi.org/10.1145/3324884.3416590

[11] D. Yang, Y. Lei, X. Mao, D. Lo, H. Xie, and M. Yan, "Is the ground truth really accurate? Dataset purification for automated program repair," in *Proceedings of 2021 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, Honolulu, HI, USA, 2021, pp. 96-107. https://doi.org/10.1109/SANER50967.2021.00018

[12] M. Tufano, C. Watson, G. Bavota, M. Di Penta, M. White, and D. Poshyvanyk, "An empirical investigation into learning bug-fixing patches in the wild via neural machine translation," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, Montpellier, France, 2018, pp. 832-837. https://doi.org/10.1145/3238147.3240732

[13] Github, "Javaparser," 2017 [Online]. Available: https://github.com/javaparser/javaparser.

[14] R. Pawlak, M. Monperrus, N. Petitprez, C. Noguera, and L. Seinturier, "SPOON: a library for implementing analyses and transformations of java source code," *Software: Practice and Experience*, vol. 46, no. 9, pp. 1155-1179, 2016. https://doi.org/10.1002/spe.2346

[15] J. R. Falleri, F. Morandat, X. Blanc, M. Martinez, and M. Monperrus, "Fine-grained and accurate source code differencing," in *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, Vasteras, Sweden, 2014, pp. 313-324. https://doi.org/10.1145/2642937.2642982

[16] T. Wolf, L. Debut, V. Sanh, J. Chaumond, C. Delangue, A. Moi, et al., "Transformers: state-of-the-art natural language processing," in *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, Virtual Event, 2020, pp. 38-45. https://doi.org/10.18653/v1/2020.emnlp-demos.6

[17] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, et al., "PyTorch: an imperative style, high-performance deep learning library," *Advances in neural Information Processing Systems*, vol. 32, pp. 8024-8035, 2019.

Abdinabiev Aslan Safarovich **https://orcid.org/0000-0001-8380-2399**

He received B.S. degree in Computer Science from National University of Uzbekistan in 2020. He also received the M.S. degree in Computer Science and Engineering from University of Seoul, Korea, in 2024. Since March 2024, he has been a Ph.D. student of the Department of Computer Science and Engineering at the University of Seoul, Korea. His current research interests mainly focus on improving the software quality.

Jisung Kim **https://orcid.org/0000-0002-2158-0994**

He received B.S. degree in Information Media Engineering from Shingu College, Korea, in 2018. He also received M.S. degree in Computer Science from University of Seoul, Korea, in 2023. Since October 2023, he has been an alternative service personnel of Daejeon Correctional Institution to perform his military duty. His current research interests mainly focus on software engineering, software testing, and natural language processing.

Sangjin Lee **https://orcid.org/0000-0002-2750-7608**

He received the B.S., M.S., and Ph.D. degrees in Computer Science from Seoul National University in 1990, 1998, and 2002, respectively. He was a researcher of Hyundai Electronics, Corp. from 1990 to 1998. Currently, he is a professor of the Department of Computer Science and Engineering at the University of Seoul, Korea. His research areas include software engineering and machine learning.