# Can We Automatically Fix Bugs by Learning Edit Operations?

Aidan Connor*
*Computer Science*
*William and Mary*
Williamsburg, VA, USA
ajconnor@email.wm.edu

Aaron Harris*
*Computer Science*
*William and Mary*
Williamsburg, VA, USA
amharris04@email.wm.edu

Nathan Cooper
*Computer Science*
*William and Mary*
Williamsburg, VA, USA
nacooper01@email.wm.edu

Denys Poshyvanyk
*Computer Science*
*William and Mary*
Williamsburg, VA, USA
denys@cs.wm.edu

***Abstract*—There has been much work done in the area of automated program repair, specifically through using machine learning methods to correct buggy code. Whereas some degree of success has been attained by those efforts, there is still considerable room for growth with regard to the accuracy of results produced by such tools. In that vein, we implement Hephaestus, a novel method to improve the accuracy of automated bug repair through learning to apply edit operations. Hephaestus leverages neural machine translation and attempts to produce the edit operations needed to correct a given buggy code segment to a fixed version. We examine the effects of using various forms of edit operations in the completion of this task. Our study found that all models which learned from edit operations were not as effective at repairing bugs as models which learned from fixed code segments directly. This evidences that learning edit operations does not offer an advantage over the standard approach of translating directly from buggy code to fixed code. We conduct an analysis of this lowered efficiency and explore why the complexity of the edit operations-based models may be suboptimal. Interestingly, even though our Hephaestus model exhibited lower translation accuracy than the baseline, Hephaestus was able to perform successful bug repair. This success, albeit small, leaves the door open for other researchers to innovate unique solutions in the realm of automatic bug repair.**

***Index Terms*—automatic program repair, neural networks, neural-machine translation, software defect analysis, negative results**

## I. Introduction

A 2018 report from the Consortium for IT Software Quality states that 16.87% of the costs incurred by poor-quality software can be attributed to the task of fixing and debugging defects in software [10]. With this vast cost looming over the industry, considerable research has been dedicated to the idea of fixing bugs in an automated manner. We must consider that manual bug repair necessitates comprehension, localization, refactoring, and the correction itself. Each of these phases varies greatly with the complexity of the software product and the experience of the software developer. A process which can determine the existence of a bug in the source code and propose a corrected code replacement could reduce short- and long-term costs associated with program repair. In this study, we concern ourselves with the latter half of such a process: repairing code which is already known to be buggy.

*These authors contributed equally to this work.

The discussion of automated bug repair starts with a consideration of the nature of the problem. The initial state is that a segment of code is known to have a defect. A software engineer may attempt several iterations of alteration before landing on the perceived fix. Considering a given pair of buggy and fixed code segments, which we call $m_{\textbf{bug}}$ and $m_{\textbf{fix}}$ respectively, there are numerous ways to analyze the transformation from the former into the latter. A successful transformation is the one that results in a code segment $m_{\textbf{fix}}$ that no longer exhibits the error that was present in $m_{\textbf{bug}}$; that is, it is syntactically and logically correct, as well as being able to compile and run without an error.

A naive approach would be to attempt some sort of comparison algorithm that would identify the type of bug and replace it with a prescribed patch, substituting identifiers as needed. Such an approach would likely take more time to develop and stock with prescriptions than it would save over the span of its usage. Following the development path of efforts in this area, a statistical replacement model may also be effective. However, we have a problem represented by data points, an observable pattern, and an unknown ideal functional to perform the necessary work; a learning approach emerges as the most appropriate solution. From there, we must consider which learning approach is best suited for the task.

A learning approach that has seen some success in automatic bug repair is neural machine translation (NMT), a technique which takes advantage of neural networks and supervised learning. NMT it is also widely used in the field of human language translation. For example, an NMT model can be created to translate from English to German and vice versa by training the model with a dataset containing a parallel corpus of English text and its German counterpart. However, directly applying the NMT approach to source code repair presents some inefficiencies. Specifically, many bug fixes involve changes to only one or two lines in the source code and leave the rest of the program unchanged. Therefore, naively training an NMT model to directly predict the fixed code results in suboptimal performance because much of the model's computational capacity is wasted on duplicating unchanged code. Previous work [19, 16], has attempted to mitigate this inefficiency by only predicting the specific statements that need to be changed or edit operations that

need to be performed on the Abstract Syntax Tree (AST). While predicting changes at the statement level as done in [19] is more efficient, finer-grained changes that work at the individual token level would be more optimal. Additionally, working at the AST level as done in [16] requires specialized parsers which are not easily extended to multiple programming languages.

To overcome the above issues, we created Hephaestus, a novel method intended to improve the accuracy of automated bug repair. Hephaestus leverages neural machine translation to predict the edit operations, which are derived by the well known Levenshtein Distance algorithm [12], needed to correct a given buggy code segment to a fixed version. These edit operations work at the token level of source code. Centralizing training on edit operations allows the model to focus on generating only the exact steps needed to fix a particular bug rather than generating an entirely working fixed version of the code. Furthermore, operating at the source code level allows for Hephaestus to work on any language without language-specific parsers. We evaluate Hephaestus on the program repair portion of the popular CodeXGlue [11] benchmark, which was also used to assess the performance of program repair models in past studies [4, 8]. We find that even though our performance was worse compared to directly predicting the fixed code, Hephaestus was able to successfully repair a nontrivial proportion of the tested code segments. Moreover, the vast majority of edit operations that Hephaestus produced were syntactically correct. We later explore the reasons behind Hephaestus' suboptimal performance.

## II. Related Work

The basis of our research is built on an existing body of work by Tufano et al. [17]. Their study demonstrates a method for repairing code through the identification of bug-fix patterns in large software repositories. After training models via an NMT approach, the authors were able to achieve between a 9% and 50% success rate in identifying the correct fix for a given bug. Also of note is the authors' emphasis on examining target code at the method level vice the file or class level, as well as their decision to exclude exceptionally large methods from their study. The paper further defines a method for using AST operations to provide a granular measure of the edit distance between the buggy and fixed versions of the code. Much of the methodology of their work is duplicated in our paper, such as the assessment of bug-fix pairs and the tools used.

Additional previous work by Tufano et al. [18] delves into the background of the methodology in [17], exploring the decision to pursue an NMT-based approach. The paper also discusses the usage of Deep Learning approaches in qualitative analysis regarding "meaningful" changes to code. Meaningful changes are defined by Tufano et al. as commits which were reviewed through a pull request; these so-called meaningful changes are the primary focus of their paper.

Contemporary work by Chen et al. [5] covers methods used in determining the appropriate fixes for single-line code changes. Unlike [17] and [18], this work relies on the creation of a heavily curated dataset of single-line bug fixes to train its model. Whereas this alone separates the two bodies of work, we leverage the analysis provided with regard to limiting vocabulary to inform our decisions. The problem of an unlimited vocabulary set, predominantly in the form of identifiers in code such as variable names, is one which we encounter.

The work of Chakraborty et al. [4], another NMT-based approach called CODIT, is similar on the surface to that of Tufano et al. [17]. As with the other referenced papers, concepts resurface such as tokenization and issues with vocabulary sizes and abstractions. Both papers choose to focus on smaller sizes of patch targets, though CODIT did realize some moderate success with larger patches. The primary use of this work in our research is to provide an alternative lens with which to view the problem of automated bug repair.

Contemporary work by Jiang et al. [7] also uses an NMT-based approach, but the authors begin by addressing several concerns with existing NMT-based approaches. The most pressing of these concerns are the likelihood that the correct fix for a given bug does not exist within the model's output space and the model's lack of awareness of strict code syntax. In order to overcome the latter problem, the authors pre-train their model on the programming language in question so it can learn to generate more developer-like outputs. Notably, the work of Tufano et al. [17, 18] is referenced in this work and used for a comparison.

Yuan and Banzhaf [19] approach the problem of automatic program repair in an adjacent, yet distinct, manner. The relevance to our research is the grouping of fine-granularity edits into larger statement-level edits.

Similar work to ours was performed by Tarlow et al. [16] where they train a model to generate edit operations that manipulate a buggy program's AST into a fixed version of the program's AST. However, as mentioned above, AST-level interaction requires specialized parsers and tree differencing algorithms, making this approach difficult to apply to other programming languages. Instead, our approach works at the source code level and uses the well known Levenshtein Distance algorithm for constructing the edit operations.

All related works discussed heretofore implement some form of neural network to generate repairs for buggy code. Relatively earlier work by Andersen et al. [2] outlines principles for algorithmically inferring program repair (*i.e.,* without the use of a neural network) during the process of collateral evolution. This mostly involves modifying function calls and their arguments when the API of an implemented library changes. The bugs which need repair in these instances refer to API usage scenarios which were compliant with the old version of the library and become erroneous when the library's API is updated. The algorithmic approach in [2] to repair these bugs is very accurate, but it is only applicable to instances of this narrow use case.

Mousavi et al. [13] performed a survey of automatic software repair methodologies, identifying two distinct areas of program repair. The first area identified is *runtime* level software repair, where live software is rescued from a fault

and restored to a running state. The second area is *source code* level, where code is examined and bugs and repair patterns are learned in repositories. The authors point to several obstacles, including overfitting and the disparity between predicted bug fix operations and those that would mimic a human software developer. Our research deviates from that of Mousavi et al. on several key elements, namely that overfitting is not believed to be a core issue in our implementation, and that we are not concerned with the parity between the specified edit operations and operations made by a live programmer. Instead, we concern ourselves primarily with the prediction power of our model with respect to bug fixing accuracy.

## III. Background

Whereas the idea of translating between buggy and fixed code is a relatively recent development, the concept of translating human languages using computers is far from a new concept. Rules-based and statistical models were attempted for a time. The translation from one language to another can be thought of in this sense as a series of predictions of which word or symbol will come next, given a history of all words or symbols seen up to that point. These systems were the basis of various speech and language recognition methods in the past [15].

In 2003, Bengio et al. proposed a solution based in part on these statistical language models. The primary obstacle was the *curse of dimensionality*, the name given to the problem caused by the vastness of the domain of possible words in the training set versus the testing set. There is a reasonable probability that the model will encounter words during testing that it did not see in training. Then during prediction, the model may have difficulty inferring the meaning of the previously unseen words. The authors proposed a method using a neural network to learn from two input corpora, and were able to surpass the state-of-the-art at the time. [3]

The existence of this field of study under the moniker Neural Machine Translation arrived in 2014, and there has been a prolific growth in the number of papers published in this body of work since [15]. As a result, NMT models have become more standardized over time. In an effort to provide a common toolkit and benchmark, researchers at Harvard SEAS and SYSTRAN created OpenNMT [9]. This is the toolkit which we use in our study.

Figure 1 demonstrates a simplified version of the NMT process. Embedding representations of an input sequence are learned in the encoder; then from the input, the decoder learns to predict the next symbol or word in the sequence. This prediction is the impetus for the use of code abstraction. Without abstraction, the vocabulary space of a large source code corpus would be too vast to be meaningfully used in NMT learning [8]. However, abstraction also introduces major limitations which are discussed in Sections 4 and 7.

In concept, the jump from traditional language translation to buggy $\rightarrow$ fixed code translation is simply a matter of regarding the buggy and fixed versions of code as varieties of language – two dialects of a common mother tongue. There are, of course, several distinguishing features that separate the two ideas. When translating between human languages, it is typical to replace the majority of the input sequence. In contrast, the changes required to fix a code segment may be minimal (such as a missing semicolon), or considerable (replacing entire lines or blocks of code). Importantly, unlike human language translation, the output of code repair translation should *not* have the same meaning as its input; the input is known to be buggy and the output is intended to be fixed.

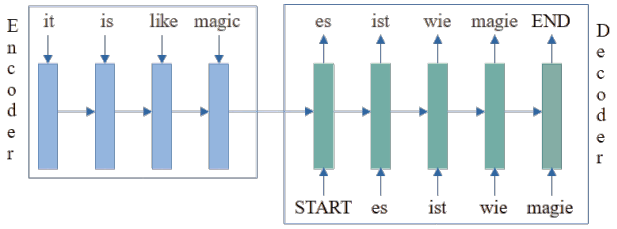



Fig. 1: Example of NMT encoding and decoding.

## IV. Approach

### A. Levenshtein Edit Operations

The most successful approaches thus far to automatic broad-range bug repair have come from [17] and [7], as well as their related studies. As mentioned in Section 2, these studies leverage NMT methodologies to translate directly from buggy code segments to fixed code segments, with some variation in the pre-training approach. For example, given $\boldsymbol{m}_{\mathbf{bug}}$ as an input sequence, the NMT model in [17] would attempt to produce directly the sequence of tokens comprising $\boldsymbol{m}_{\mathbf{fix}}$. Our novel approach, which we name Hephaestus, differs with regard to the NMT output. Given $\boldsymbol{m}_{\mathbf{bug}}$ as an input sequence, a Hephaestus NMT model attempts to produce a sequence of steps, called *edit operations*, which transforms the inputted $\boldsymbol{m}_{\mathbf{bug}}$ into $\boldsymbol{m}_{\mathbf{fix}}$. Our approach therefore determines a possible $\boldsymbol{m}_{\mathbf{fix}}$ *indirectly* via edit operations as intermediaries.

Edit operations are used to modify the tokens of a token sequence or delimited string; different edit operations modify the tokens in different ways. The most basic form of edit operation is the Levenshtein edit operation, or Levenshtein operation for short. A Levenshtein operation affects only one token in a sequence, and comes in three different "flavors":

- *Insertion*: A token is inserted and the number of tokens increases by one.
- *Deletion*: A token is removed and the number of tokens decreases by one.
- *Replacement*: A token is substituted for another token and the number of tokens remains constant.

### B. Compound Edit Operations

We define a *compound edit operation*, or compound operation for short, as a group of one or more edit operations. Applying the compound operation to some $\boldsymbol{m}_{\mathbf{bug}}$ will produce exactly the same result as if the constituent edit operations were applied, in order, to $\boldsymbol{m}_{\mathbf{bug}}$. The operations in the grouping

can be basic Levenshtein operations or compound operations themselves. Similar to Levenshtein operations, compound operations come in three flavors:

- *Insertion*: No tokens are removed and a sequence of at least one token is added.
- *Deletion*: A contiguous sequence of at least one token is removed and no tokens are added.
- *Replacement*: Either of the following:
  - No tokens are removed and no tokens are added (this can occur when condensing two operations that effectively cancel each other, such as an insertion directly followed by a deletion at the same index).
  - A contiguous sequence of at least one token is removed at some index and a sequence of at least one token is added at that same index.

Any edit operation as defined can be represented by the function $\boldsymbol{op}(i, j, S)$, where $i$ and $j$ are non-negative integers such that $j \geq i$, and $S = \{s_0, s_1, \dots\}$ is a sequence of tokens. The function operates by first deleting all tokens within the index range $[i, j)$, then inserting the token sequence $S$ at index $i$. Every type of edit operation can be represented by various configurations of $i$, $j$, and $S$:

| | |
|---|---|
| Levenshtein insertion: | $j = i$ and $\|S\| = 1$ |
| Compound insertion: | $j = i$ and $\|S\| \geq 1$ |
| Levenshtein deletion: | $j = i + 1$ and $\|S\| = 0$ |
| Compound deletion: | $j > i$ and $\|S\| = 0$ |
| Levenshtein replacement: | $j = i + 1$ and $\|S\| = 1$ |
| Compound replacement: | ($j = i$ and $\|S\| = 0$) or ($j > i$ and $\|S\| > 0$) |

Algorithm 1 determines the flavor of an edit operation according to the above configurations of $i$, $j$, and $S$.

We refer to the grouping process which forms compound operations as *condensing*; *i.e.,* edit operations are *condensed* into compound operations. Condensing is represented mathematically by the equation $\boldsymbol{c}(\boldsymbol{E}) = \boldsymbol{E}'$, where $\boldsymbol{E} = \{\boldsymbol{e}_0, \boldsymbol{e}_1, \dots\}$ is the original sequence of edit operations, $\boldsymbol{c}$ is the condensing function, and $\boldsymbol{E}' = \{\boldsymbol{e}'_0, \boldsymbol{e}'_1, \dots\}$ is the condensed sequence of compound operations. Because elements of $\boldsymbol{E}$ are grouped together to form elements of $\boldsymbol{E}'$, it follows that $|\boldsymbol{E}'| \leq |\boldsymbol{E}|$. In our study, we condense edit operations according to three different strategies: *basic condensing*, *loose condensing*, and *strict condensing*.

*1) Basic Condensing:* A trivial case where $\boldsymbol{E}' = \boldsymbol{E}$; every basic compound operation $\boldsymbol{e}' \in \boldsymbol{E}'$ corresponds with exactly one $\boldsymbol{e} \in \boldsymbol{E}$.

*2) Loose Condensing:* $\boldsymbol{E}$ is condensed according to *loose compatibility*. A sequence of edit operations is said to be *loosely compatible* if and only if the application of its constituent operations, in order, is equivalent to the application of some singular $\boldsymbol{op}(i, j, S)$. More colloquially, edit operations are loosely compatible with one another if and only if they modify a contiguous section of tokens. Algorithm 2 determines the loose compatibility of edit operations and what happens when such operations are condensed.

When loosely condensing $\boldsymbol{E}$ into $\boldsymbol{E}'$, it is not necessary, nor is it likely, that every $\boldsymbol{e} \in \boldsymbol{E}$ is loosely compatible. Rather, $\boldsymbol{E}$ is effectively partitioned into subsequences, where every edit operation in each subsequence is loosely compatible. Then, each subsequence is represented as its own $\boldsymbol{op}(i, j, S)$ which together define the final condensed sequence $\boldsymbol{E}'$. Thus, the

**Algorithm 1** Returns the flavor of the edit operation $\boldsymbol{e}$. Note that $\boldsymbol{e}$ is an object representation of the function $\boldsymbol{op}(i, j, S)$; *e.g.,* $\boldsymbol{e}.S$ refers to the $S$ property of $\boldsymbol{e}$. The returned value is one of {"Insertion", "Deletion", "Replacement"}.

```
procedure FLAVOR(e)
    r ← e.j − e.i
    if r == 0 and |e.S| == 0 then
        return "Replacement"
    else if r == 0 then
        return "Insertion"
    else if |e.S| == 0 then
        return "Deletion"
    else
        return "Replacement"
    end if
end procedure
```

**Algorithm 2** Attempts to condense the edit operation $\boldsymbol{x}$ into the edit operation $\boldsymbol{e}$ according to loose compatibility. Edit operation $\boldsymbol{x}$ is assumed to occur after edit operation $\boldsymbol{e}$. Returns True if the condensing was successful, *i.e.,* if $\boldsymbol{e}$ and $\boldsymbol{x}$ are loosely compatible; returns False otherwise.

```
procedure ADDLOOSE(e, x)
    l ← e.i + |e.S|
    if x.j < e.i or x.i > l then
        return False
    else if x.i < e.i then
        if x.j ≤ l then
            e.S ← x.S + e.S[(x.j − e.i) to end]
            e.i ← x.i
        else
            e.S ← x.S
            e.i ← x.i
            e.j ← e.j + x.j − l
        end if
    else
        if x.j ≤ l then
            e.S ← e.S[start to (x.i − e.i)] + x.S+
                  e.S[(x.j − e.i) to end]
        else
            e.S ← e.S[start to (x.i − e.i)] + x.S
            e.j ← e.j + x.j − l
        end if
    end if
    return True
end procedure
```

**Algorithm 3** Condenses the sequence of edit operations $\boldsymbol{E}$ into the sequence of compound operations $\boldsymbol{E}'$ according to loose compatibility. Returns $\boldsymbol{E}'$.

**Require:** $|\boldsymbol{E}| > 0$
1: **procedure** CONDENSELOOSE($\boldsymbol{E}$)
2: $\quad\boldsymbol{E}' \leftarrow \{\boldsymbol{E}[0]\}$
3: $\quad$**for** each $\boldsymbol{e} \in \boldsymbol{E}[1$ to end$]$ **do**
4: $\quad\quad$**if not** ADDLOOSE($\boldsymbol{E}'$[last], $\boldsymbol{e}$) **then**
5: $\quad\quad\quad\boldsymbol{E}' \leftarrow \boldsymbol{E}' + \{\boldsymbol{e}\}$
6: $\quad\quad$**end if**
7: $\quad$**end for**
8: $\quad$**return** $\boldsymbol{E}'$
9: **end procedure**

**Algorithm 4** Condenses the sequence of edit operations $\boldsymbol{E}$ into the sequence of compound operations $\boldsymbol{E}'$ according to strict compatibility. Returns $\boldsymbol{E}'$. Note that the **if** condition on line 4 employs short-circuiting.

**Require:** $|\boldsymbol{E}| > 0$
1: **procedure** CONDENSESTRICT($\boldsymbol{E}$)
2: $\quad\boldsymbol{E}' \leftarrow \{\boldsymbol{E}[0]\}$
3: $\quad$**for** each $\boldsymbol{e} \in \boldsymbol{E}[1$ to end$]$ **do**
4: $\quad\quad$**if not** (
$\quad\quad\quad$FLAVOR($\boldsymbol{e}$) == FLAVOR($\boldsymbol{E}'$[last]) **and**
$\quad\quad\quad$ADDLOOSE($\boldsymbol{E}'$[last], $\boldsymbol{e}$)
$\quad\quad$) **then**
5: $\quad\quad\quad\boldsymbol{E}' \leftarrow \boldsymbol{E}' + \{\boldsymbol{e}\}$
6: $\quad\quad$**end if**
7: $\quad$**end for**
8: $\quad$**return** $\boldsymbol{E}'$
9: **end procedure**

subsequences of edit operations in $\boldsymbol{E}$ are combined according to loose compatibility, resulting in an $\boldsymbol{E}'$ such that $|\boldsymbol{E}'| \leq |\boldsymbol{E}|$. Algorithm 3 defines explicitly the loose condensing process.

*3) Strict Condensing:* $\boldsymbol{E}$ is condensed according to *strict compatibility*. A sequence of edit operations is said to be *strictly compatible* if and only if it is loosely compatible and every edit operation is of the same flavor. Hence the term "strict" – strict compatibility has more stringent criteria than loose compatibility. Therefore $|\boldsymbol{E}'_{\textbf{strict}}| \geq |\boldsymbol{E}'_{\textbf{loose}}|$ for any given sequence of edit operations $\boldsymbol{E}$. Algorithm 4 describes explicitly the strict condensing process.

With respect to the three condensing strategies, it is likely that the compound operations in $\boldsymbol{E}'$ will vary depending on the strategy used to condense $\boldsymbol{E}$. Despite the possible variation in content, each $\boldsymbol{E}'$ defines the exact same translation behavior when its constituent edit operations are applied to a token sequence. Figure 2 shows an example of this phenomenon. This variation in content and consistency in behavior across the different condensing strategies is desirable, as it may allow the determination of which form of edit operation is most effective in NMT models. This concept is explored further in RQ2.

### C. Dataset Construction

Each of our datasets is formatted as a set of rows, where each row contains one pair of token sequences. The first sequence in a pair denotes what is translated *from*, and the second sequence in a pair denotes what is translated *into*; *i.e.,* the first sequence is translated into the second sequence. Constructing datasets in this way allows for easy consumption by an NMT model. The model learns from the relationships between the first and second sequences in each pair and gains to some extent the ability to translate arbitrary token sequences.

*1) Control Dataset:* The control dataset is used to establish a baseline against which we can evaluate our experimental results. Thus, the control set is not involved with edit operations in any way; rather, it is used to train an NMT model to translate directly from buggy code to fixed code.

For the control dataset, we use a subset of the Bugs2Fix data provided in Microsoft's CodeXGlue project [11]. The data which we include for the control consist of about 58,000 unique Java method bug-fix pairs, with each method having between 1 and 50 tokens, inclusive. A bug-fix pair (BFP) is defined as a pair $(\boldsymbol{m}_{\textbf{bug}}, \boldsymbol{m}_{\textbf{fix}})$, where $\boldsymbol{m}_{\textbf{bug}}$ is some method source code containing bugs and $\boldsymbol{m}_{\textbf{fix}}$ is its corresponding method in which the bugs are fixed. Each method in the CodeXGlue data has been preformatted according to Section 2.2 of Tufano et al. [17]; that is, their tokens have been abstracted except for the most common literals and identifiers, called *idioms*, which retain their actual values per the source code. A major limitation of the abstraction process as mentioned in [17] is that only BFPs in which $\boldsymbol{m}_{\textbf{fix}}$ is a rearrangement of the tokens in its corresponding $\boldsymbol{m}_{\textbf{bug}}$ may be considered. The inclusion of idioms mitigates this problem to some extent, as it enlarges the vocabulary and increases the number of BFPs which can be considered.

Figure 2 shows an example of one BFP belonging to the control set. Looking at $\boldsymbol{m}_{\textbf{bug}}$ in the figure, tokens 0, 1, 2, and 5 appear often enough in the rest of the data that they are left as idioms. Tokens 3, 6, 9, and 11 do not meet this frequency threshold and are therefore abstracted.

*2) Machine Strings:* In order to include edit operations in our datasets, a method of transforming edit operations into strings is necessary. Let the string representation of an edit operation $\boldsymbol{e}$ be called its *machine string*, denoted by $\boldsymbol{ms}(\boldsymbol{e})$. Machine strings come in two forms: *typed* and *general*. Every edit operation has only one corresponding machine string in each form, and every valid machine string has only one corresponding edit operation.

A *typed* form machine string $\boldsymbol{ms}_{\textbf{typed}}(\boldsymbol{op}(i, j, S))$ is composed like so:

$$\texttt{<}f\texttt{>}\quad i\quad j\quad \texttt{<sep>}\quad s_0\quad s_1\quad \cdots\quad \texttt{</}f\texttt{>}$$

where $f$ is one of `ins`, `del`, or `rep`, depending on if the flavor of the represented edit operation is insertion, deletion, or replacement, respectively.

For example, if $\boldsymbol{e} = \boldsymbol{op}(8, 9, \{\text{“this”}, \text{“.”}, \text{“VAR\_1”}\})$, then $\boldsymbol{ms}_{\textbf{typed}}(\boldsymbol{e})$ is

```
<rep> 8 9 <sep> this . VAR_1 </rep>
```

A *general* form machine string $\boldsymbol{ms}_{\textbf{general}}(\boldsymbol{op}(i, j, S))$ is composed like so:

$$\texttt{<op>}\ \ i\ \ j\ \ \texttt{<sep>}\ \ s_0\ \ s_1\ \ \cdots\ \ \texttt{</op>}$$

For example, using the same $\boldsymbol{e}$ as previously, $\boldsymbol{ms}_{\textbf{general}}(\boldsymbol{e})$ is

```
<op> 8 9 <sep> this . VAR_1 </op>
```

Unlike typed form, general form machine strings do not explicitly store the flavor of their represented edit operations; however, the flavor can still be determined via Algorithm 1. We make the distinction between typed and general form to determine if the form of machine string used during training affects the Hephaestus models' abilities to learn edit operations.

In addition to single edit operations, sequences of edit operations can be converted to machine strings. To convert a sequence of edit operations $\boldsymbol{E}$ to a machine string, every $\boldsymbol{e} \in \boldsymbol{E}$ is first converted to a machine string, then those individual machine strings are concatenated in order, like so:

$$\boldsymbol{ms}(\boldsymbol{E}) = \boldsymbol{ms}(\boldsymbol{e}_0) + \boldsymbol{ms}(\boldsymbol{e}_1) + \cdots$$

*3) Experimental Datasets:* For each BFP in the control set, the minimal sequence of Levenshtein operations is extracted which deterministically translates the $\boldsymbol{m}_{\textbf{bug}}$ into its corresponding $\boldsymbol{m}_{\textbf{fix}}$. This sequence is denoted by $\boldsymbol{E}_{\textbf{fix}}$, where $|\boldsymbol{E}_{\textbf{fix}}|$ is equal to the Levenshtein edit distance between $\boldsymbol{m}_{\textbf{bug}}$ and $\boldsymbol{m}_{\textbf{fix}}$. The application of $\boldsymbol{E}_{\textbf{fix}}$ to its corresponding $\boldsymbol{m}_{\textbf{bug}}$ is represented mathematically by the addition operator like so:

$$\boldsymbol{m}_{\textbf{bug}} + \boldsymbol{E}_{\textbf{fix}} = \boldsymbol{m}_{\textbf{fix}}$$

$\boldsymbol{E}_{\textbf{fix}}$ is then condensed into basic compound operations $(\boldsymbol{E}_{\textbf{fix}})'_{\textbf{basic}}$, strict compound operations $(\boldsymbol{E}_{\textbf{fix}})'_{\textbf{strict}}$, and loose compound operations $(\boldsymbol{E}_{\textbf{fix}})'_{\textbf{loose}}$. The application of any one of these sequences of compound operations to an $\boldsymbol{m}_{\textbf{bug}}$ results in its corresponding $\boldsymbol{m}_{\textbf{fix}}$. Figure 2 shows an example of the extraction of $\boldsymbol{E}_{\textbf{fix}}$ from a BFP and the proceeding condensing of $\boldsymbol{E}_{\textbf{fix}}$ into its compound variants.

Note that $\boldsymbol{E}_{\textbf{fix}}$ is not the only Levenshtein operation sequence which transforms a given $\boldsymbol{m}_{\textbf{bug}}$ into its corresponding $\boldsymbol{m}_{\textbf{fix}}$. Indeed, there is an infinite set of Levenshtein operation sequences $\boldsymbol{\mathcal{E}}$ such that $\forall \boldsymbol{E} \in \boldsymbol{\mathcal{E}}$, $\boldsymbol{m}_{\textbf{bug}} + \boldsymbol{E} = \boldsymbol{m}_{\textbf{fix}}$. $\boldsymbol{E}_{\textbf{fix}}$ is simply the *minimal* member of $\boldsymbol{\mathcal{E}}$, meaning that $\forall \boldsymbol{E} \in \boldsymbol{\mathcal{E}}$, $|\boldsymbol{E}_{\textbf{fix}}| \leq |\boldsymbol{E}|$. Likewise, $(\boldsymbol{E}_{\textbf{fix}})'_{\textbf{basic}}$ is the minimal sequence of all basic compound operation sequences which transform $\boldsymbol{m}_{\textbf{bug}}$ into $\boldsymbol{m}_{\textbf{fix}}$, $(\boldsymbol{E}_{\textbf{fix}})'_{\textbf{strict}}$ is the minimal sequence of the strict compound operation sequences, and $(\boldsymbol{E}_{\textbf{fix}})'_{\textbf{loose}}$ is the minimal sequence of the loose compound operation sequences.

We construct three experimental datasets, one for each of the basic, strict, and loose compound operations which were derived from the Levenshtein operations as described previously. Each row in the datasets is a pair: the first element is a buggy method and the second element is the machine string representation of the sequence of compound operations which transforms the buggy method into its corresponding fixed method. Thus, every BFP in the control set is represented in the experimental sets. Each experimental dataset is also copied to produce two variants: one variant contains the compound operation machine strings in typed form and the other contains the machine strings in general form. Table I describes mathematically the format of the rows in each dataset.

### D. Hephaestus Model Construction

We select training parameters for the Hephaestus models according to three parameter groups:

- *LSTM+General*: These parameters are as close as possible to the highest performing NMT model as described in [17]. An Long Short Term Memory (LSTM) architecture is used, and experimental models are trained with edit operation machine strings in general form.
- *GRU+General*: A Gated Recurrent Unit (GRU) architecture is used instead of an LSTM architecture. All other parameters are unchanged from the LSTM+General group.
- *LSTM+Typed*: Experimental models are trained with machine strings in typed form. All other parameters are unchanged from the LSTM+General group.

Three parameter groups and four datasets on which to train gives twelve Hephaestus models total. For example, the Hephaestus model trained with the loosely condensed operation dataset and GRU+General parameters is referred to as the “loose GRU+General model”. Each model is trained with 80% of their respective dataset and validated with 10% of the dataset using 50,000 training steps. The training process took about 5.5 hours for each model using an NVIDIA Titan RTX GPU.

We then test each Hephaestus model using the remaining 10% of our datasets. The final output for all the models is abstracted Java method source code which is supposedly “fixed”. The experimental models go about this process indirectly: they first output edit operation machine strings, then apply the represented edit operations to the inputted $\boldsymbol{m}_{\textbf{bug}}$ to reach the final output. Models are evaluated based on differences in their final output and the ideal output of the true $\boldsymbol{m}_{\textbf{fix}}$ for each inputted $\boldsymbol{m}_{\textbf{bug}}$.

TABLE I: Dataset Row Formats

| Dataset name | Row format |
|---|---|
| Control | $(\boldsymbol{m}_{\textbf{bug}},\ \boldsymbol{m}_{\textbf{fix}})$ |
| Basic | $(\boldsymbol{m}_{\textbf{bug}},\ \boldsymbol{ms}((\boldsymbol{E}_{\textbf{fix}})'_{\textbf{basic}}))$ |
| Strict | $(\boldsymbol{m}_{\textbf{bug}},\ \boldsymbol{ms}((\boldsymbol{E}_{\textbf{fix}})'_{\textbf{strict}}))$ |
| Loose | $(\boldsymbol{m}_{\textbf{bug}},\ \boldsymbol{ms}((\boldsymbol{E}_{\textbf{fix}})'_{\textbf{loose}}))$ |

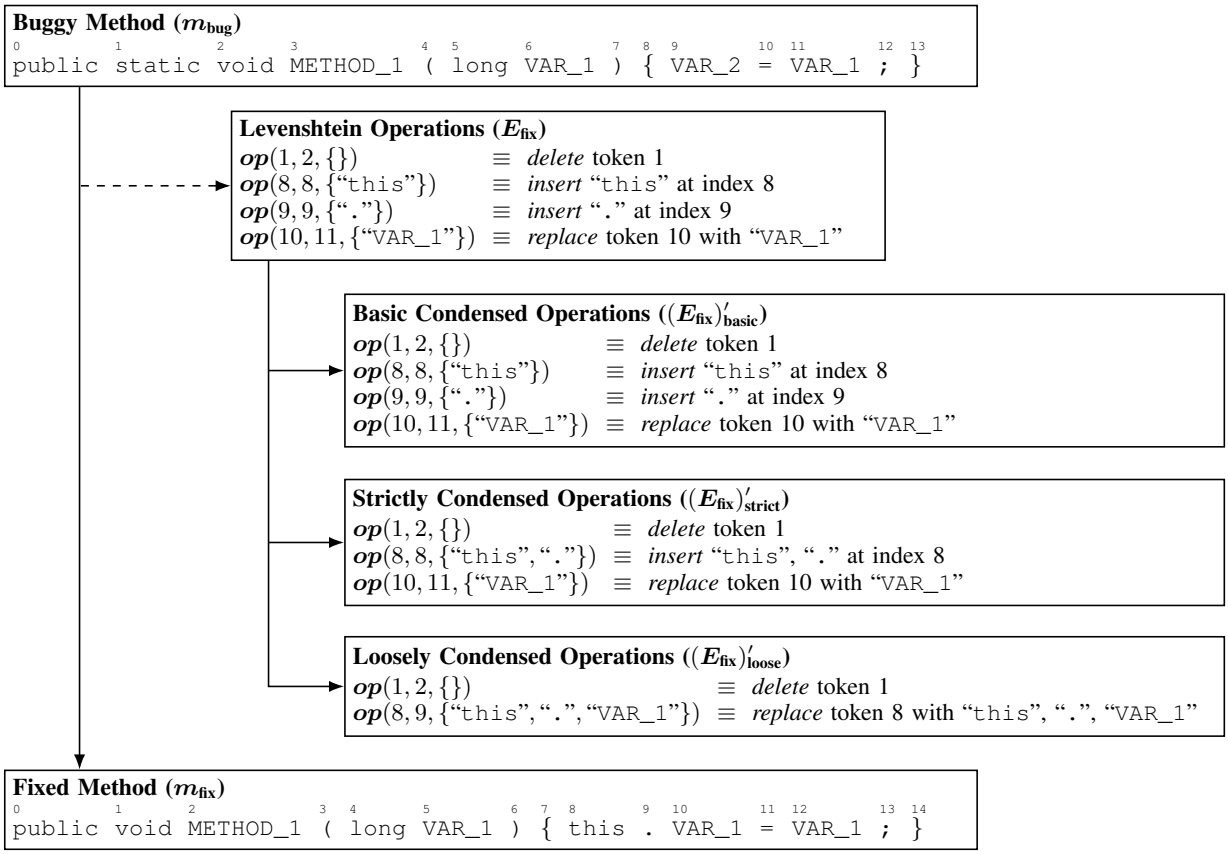



Fig. 2: Example of a real bug-fix pair from the control dataset. Levenshtein operations are derived from the transformation of the buggy method into the fixed method. Then, these Levenshtein operations are condensed into basic, strict, and loose compound operations. The indices of the tokens in the buggy and fixed methods are shown in small numbers.

### *E. Hephaestus Model Architecture*

Our neural network models in the LSTM+General and LSTM+Typed groups are built around a two-layer LSTM architecture. This architecture provides internal methods for maintaining error flow throughout the LSTM layers via the constant error carousel (CEC). The CEC ensures that error signals fed forward into the LSTM layers and backpropagated to the LSTM layers are resistant to the effects of the vanishing gradient problem. Internal learning units called the input and output gates control the learning process [6].

Additionally, these models implement dropout layers with a dropout rate of 0.2. Such layers reduce overfitting in the models by randomly removing their influence from the network [14].

Models in the GRU+General group are built around a GRU architecture instead of an LSTM architecture. GRU architectures typically have fewer parameters than their LSTM counterparts, and they use forget gates instead of output gates. This implementation is used as a basis for comparison to determine if the model architecture affects the experimental outcome.

Stochastic gradient descent (SGD) is a widely-employed optimizer for machine learning methodologies. We couple SGD with a cross entropy error measure to train our models. The benefit of cross entropy error as an error measure is the reduction of the gradient to a lower order of curve, which reduces the necessity of initial starts in SGD and improves overall training efficiency. Our approach is at heart a classification problem; either the output is part of the class of correct translations or it is not. Cross entropy is widely used in classification problems, as it maximizes the probability of correctly classifying data in the output layer [1].

## V. Experimental Design

### *A. Evaluation*

We evaluate the performance of each model according to the following metrics:

*1) Perfect Prediction Accuracy:* The final output, *i.e.,* the prediction, of a model is said to be perfectly accurate if it matches exactly the $m_{\text{fix}}$ corresponding to the inputted $m_{\text{bug}}$. Perfect prediction accuracy (PPA) is the percentage of perfectly accurate predictions for all inputted $m_{\text{bug}}$ during testing.

*2) Failed prediction rate:* It is possible that for an inputted $m_{\text{bug}}$, a Hephaestus model will either output a malformed string which cannot be parsed, or output edit operations that are invalid. When this occurs, the output is discarded and dubbed a failed prediction. The failed prediction rate (FPR)

is the percentage of failed predictions for all inputted $m_{\textbf{bug}}$ during testing.

*3) Edit distance decrease:* There is a Levenshtein edit distance value, called the *prediction distance*, between each model prediction and the target $m_{\textbf{fix}}$. The term *true distance* refers to the Levenshtein edit distance between the original $m_{\textbf{bug}}$ and its corresponding $m_{\textbf{fix}}$. The edit distance decrease (EDD) is the true difference minus the prediction distance, and describes how much a model improved the inputted $m_{\textbf{bug}}$. A positive value is desirable as it indicates that the model predicted an output which is closer to the target value. We use this metric to gauge prediction accuracy even when predictions are not perfectly accurate.

*4) Training accuracy:* A measure provided by the internal NMT implementation which describes how well a model learns its training data.

### B. Research Questions

In this study, we attempt to answer the following three experimental research questions:

**RQ1. Is learning edit operations an effective approach to automatic bug repair?**

In other words, is there an advantage to training the models with edit operations instead of plain code text? This is the main focus of our study. In order to determine the effectiveness of learning edit operations, we compare the experimental metrics against the control metrics in Section 6.

**RQ2. What effect does each condensing strategy and machine string form have on the accuracy of bug repair?**

We described the various condensing strategies employed in the construction of our experimental datasets in Section 3 (basic, strict, and loose condensing). The edit operations in the data are further differentiated by the way they are represented as machine strings (general versus typed form). We devised these variations to maximize the Hephaestus models' exposure to different types of edit operations. We hypothesize that training NMT models with edit operations that are condensed and represented in different ways may have an effect on the models' prediction abilities. This is measurable via the performance metrics; if the metrics differ among the experimental models trained on different datasets, then there is evidence that condensing or representing edit operations in different ways affects the fidelity of the model.

**RQ3. What is the effect of using an LSTM-based architecture versus a GRU-based architecture on the accuracy of bug repair?**

A nontrivial difference in the performance of the LSTM and GRU versions of our architecture may indicate lurking variables affecting our study. We test our methods against both architectures to determine the closeness of performance levels.

## VI. Results

### A. Perfect Prediction Accuracy

As shown in Figure 3a, the control models greatly outperformed all versions of the experimental models at perfectly predicting bug fixes. The most accurate control model (LSTM+General) had a PPA of 14.7%, whereas the most accurate experimental model (loose GRU+General) had a PPA of only 8.3%. With respect to condensing strategy and training parameter group among the experimental models, there is no significant difference in PPA.

### B. Failed Prediction Rate

Figure 3b gives the failed prediction rates for all models. The control models maintained 100% capability of producing well-formed predictions. This is because every model always outputs a string, and the string can always be interpreted as a sequence of Java method tokens, as is the case for the control. It does not matter for this metric if the outputted tokens form valid abstracted Java code or not. In contrast, outputting invalid edit operation machine strings will indeed cause prediction failures.

Every experimental model exhibited a nonzero FPR, so it is evidenced that the introduction of edit operations in an NMT model's training regimen will cause some amount of prediction failure. The basic GRU+General model had a particularly high value of 1.56%, although the inconsistency in the data suggests that this is an anomaly. Thus, among the experimental models, the data do not show any significant effects of the experimental variables.

### C. Edit Distance Decrease

Figure 3c shows the average edit distance decrease across all models. Interestingly, every single model exhibited a negative average EDD. This means that on average, every model generated "bug fixes" which were further away from the fixed code than the original buggy code was.

The control models had the most positive of the EDD values, averaging at -1.32. The basic models were the least helpful at reducing edit distance, with an overall average EDD of -2.54. The strict and loose models fall in between, with overall average EDDs of -1.57 and -1.61, respectively.

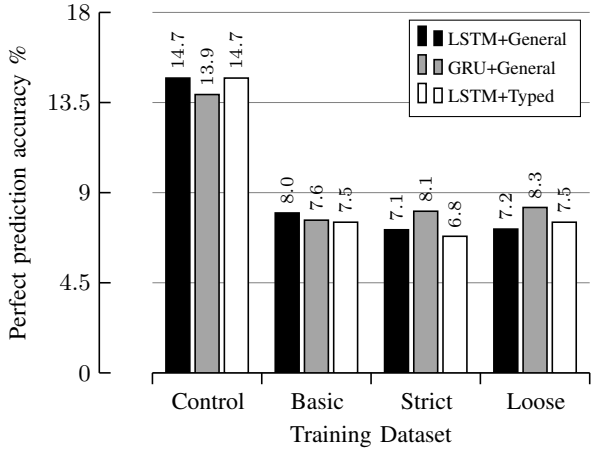
### D. Training Accuracy

Figure 3d shows that the training accuracy of the control model and basic model in the LSTM+General group maintained similar trajectories. Each model exceeded a training accuracy of 90% toward the end of the training period. Notably, the basic model's training accuracy rose more quickly initially than that of the control model. This is not reflected in the end behavior where the control model had a slightly higher training accuracy than the basic model. The strict and loose models trained less accurately than the control by a greater margin (approximately 15%). Both the strict and loose models maintained relatively parallel trajectories in their end behavior, with the strict model narrowly outperforming the loose model.

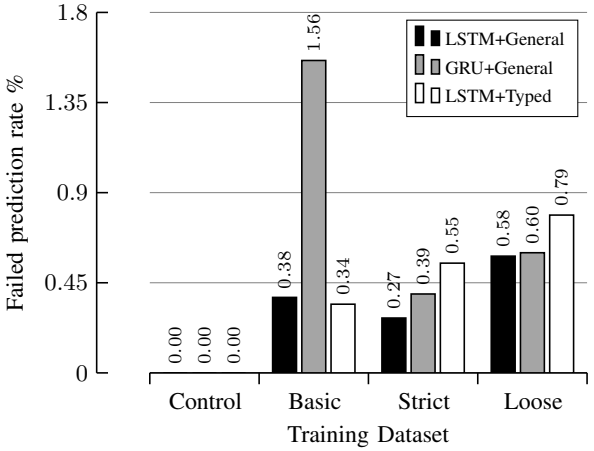
### E. Research Questions

*RQ1. Is learning edit operations an effective approach to automatic bug repair?*

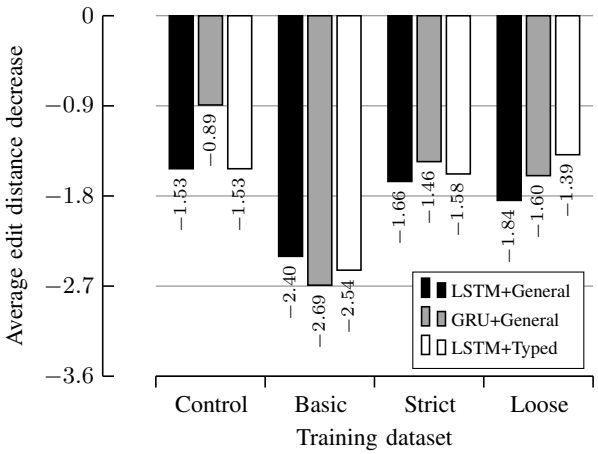
The results of this experiment suggest that learning edit operations *does not* offer advantages over the baseline approach. The control models performed better according to



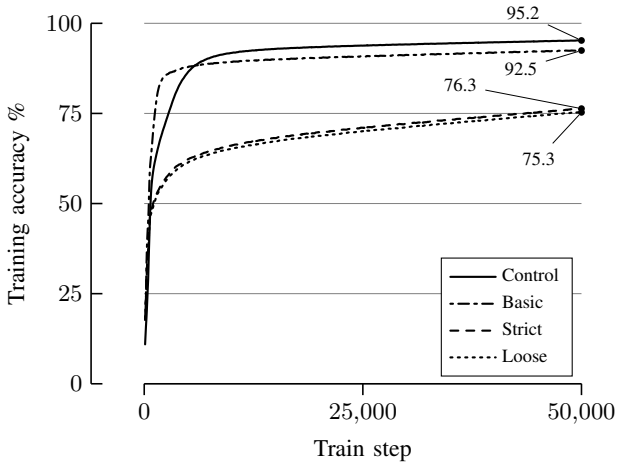

(a) Perfect prediction accuracies of all models.




(b) Failed prediction rates of all models.




(c) Average edit distance decreases for all models.




(d) Training accuracies of LSTM+General models. Labeled values are the final training accuracies.

Fig. 3: Compilation of results statistics. Note that the control model in the LSTM+General group and the control model in the LSTM+Typed group are identical. This is because the LSTM+Typed parameters only affect the form of edit operation machine strings, and the control models do not in any way interact with edit operations.

every performance metric that was tested. This is likely due to the experimental models experiencing higher entropy than the control when making predictions. The experimental Hephaestus models must determine a sequence of edit operations, decode them, and apply them to the inputted buggy method in order to predict fixed source code. In contrast, the control models output their predictions directly in the form of source code. The data show that any possible advantages gained by learning edit operations (as described in Section 1) are significantly outweighed by the complexity that such a learning process introduces to the system.

The fact that edit operations-based NMT models experience prediction failures is another disadvantage of the Hephaestus approach. Further analysis revealed that the main cause of prediction failure was that some generated edit operations modified token indices that were out-of-bounds with respect to the inputted buggy method. This failure case manifests most often when predicting fixes for buggy methods with fewer tokens; in this situation, outputted edit operations will have a higher chance of modifying out-of-bounds token indices. Additional information regarding failure analysis is available in the source documentation (see Section 9).

*RQ2. What effect does each condensing strategy and machine string form have on the accuracy of bug repair?*

The data show that the experimental models' ability to fix bugs was influenced slightly by condensing strategy. The differences in PPA between the basic, strict, and loose models are negligible, but there are differences according to the training accuracy and average EDD values. Despite having significantly lower final training accuracy, the strict and loose models had slightly more positive EDD values than the basic models (a difference of about 0.96). Thus, it is evidenced that condensing edit operations into strict and loose forms is beneficial over not condensing them at all.

The only notable effect of machine string form is shown by Figure 3b, where for the strict and loose models, training with typed form machine strings caused marginally more prediction failures than with general form machine strings. This is likely due to the fact that typed form machine strings have more stringent formatting guidelines and are therefore more difficult to generate.

*RQ3. What is the effect of using an LSTM-based architecture versus a GRU-based architecture on the accuracy of bug repair?*

Whereas there is some variation between the PPAs as shown in Figure 3a, the variation is not meaningful enough to consider as a key difference between the models. Of note, the GRU architecture performed mildly worse in the control models (by approximately 0.8%), and mildly better in the strict and loose models (by approximately 1% in each case).

## VII. Threats to Validity

Whereas care was taken to ensure the rigor of this study, there are factors which could necessitate mitigation in the application of our findings:

**Construct Validity**: From a formatting perspective, our implementation of edit operations was chosen based on convenience. It is simple to limit the representation of any edit operation to one function, namely $\boldsymbol{op}(i, j, S)$. However, this may not be the optimal approach; different formatting may produce different results.

**External Validity**: Our study is limited by our dataset; the Hephaestus methodology was only applied against the preformatted Bug2Fix CodeXGlue dataset. Whereas the population size is sufficiently large, it focuses solely on Java codebases. Based on this limitation, we can only say that our method is ineffective for Java-based examples.

**Internal Validity**: We inherit the limitations stated by Tufano et al., namely regarding the code abstraction process. Source code abstraction reduces the vocabulary size significantly and allows the use of NMT for meaningful predictions. However, this reduction of vocabulary size brings with it a cost: only BFPs in which $\boldsymbol{m}_{\textbf{fix}}$ is a rearrangement of the tokens in its corresponding $\boldsymbol{m}_{\textbf{bug}}$ may be considered [17]. One mitigation to this problem is the inclusion of idioms in the data, but only so many idioms can be included before the vocabulary size is once again too large to be useful in NMT learning. Thus, the problem remains significant.

## VIII. Future Work

Based on the results of this study, several opportunities for further research present themselves:

**Failed Prediction Analysis**: It was determined that most failed predictions were caused by generated indices outside the valid range for a given string. What changes can be made to this model to restrict the prediction range? Is there an alternative preprocessing measure that could alleviate this issue?

**Source Code Abstraction Method**: In light of the idea that different syntactic complexities may affect the perfect prediction rates in Hephaestus, does changing the abstraction method of the training dataset affect this metric? One avenue of overcoming this concern is incorporating the approach taken by Jiang et al. [7] of pre-training the model to have awareness of developer-like syntax generalities.

**Model Architecture**: This study performed all experiments guided by the framework of the Hephaestus model, focusing on the LSTM and GRU implementations. As novel learning methods and architectures emerge, reconsidering the architecture selected for this task may provide different results. Additionally, use of other NLP tools such as GPT-based architectures may prove beneficial.

**Functional Testing**: In this study, we are primarily concerned with learning to generate edit operations which transform one input element into one output element as presented in the Bugs2Fix dataset. Whereas we believe this baseline is appropriate for this initial study, we note that there is potentially more than one acceptable final state for code that qualifies as "fixed". Further testing could be done to determine if outputs that do not fail certain metrics (such as prediction failure due to invalid output) are capable of compiling and running successfully. This would involve extra software layers in addition to the methods presented in our study. In machine translation task testing, Bilingual Evaluation Understudy (BLEU) score can be used to determine the similarity of the machine output and the translation by a human interpreter; a metric of this nature would be useful in our context.

## IX. Conclusions

In this study, we presented Hephaestus, a novel approach to learning to translate from buggy to fixed code via the introduction of Levenshtein edit operations and their condensed forms: basic, strict, and loose compound operations. After training and testing the Hephaestus models, we have determined that the introduction of these specific methods for training NMT-based systems to learn bug fixes did not provide a benefit to the task.

Whereas the methods prescribed in Hephaestus failed to meet or exceed the baseline, it is important to note that they did not fail entirely. Each approach used was successful in greater than 7% of tested instances. As the chance of randomly applying edit operations to a given buggy code input and successfully producing the fixed version are minuscule, we can see that edit operations are capable of performing automated bug repair to some degree.

The code used to perform this study along with replication documentation can be found at https://github.com/WM-SEMERU/hephaestus.

## X. Acknowledgements

The authors have been supported in part by the NSF CCF-1955853 and CCF-2007246 grants. Any opinions, findings, and conclusions expressed herein are the authors' and do not necessarily reflect those of the sponsors.

## References


[1] Yaser Abu-Mostafa, Malik Magdon-Ismail, and Lin Hsuan-Tien. *Learning from Data: A Short Course*. AMLBook, Mar. 2012. ISBN: 978-1-60049-006-4.

[2] Jesper Andersen et al. "Semantic Patch Inference". In: *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering* (Sept. 2012). DOI: 10.1145/2351676.2351753. URL: https://ieeexplore.ieee.org/document/6494961.

[3] Yoshua Bengio et al. "A Neural Probabilistic Language Model". In: *Journal of Machine Learning Research* (Feb. 2003). URL: https://www.jmlr.org/papers/volume3/bengio03a/bengio03a.pdf.

[4] Saikat Chakraborty, Miltiadis Allamanis, and Baishakhi Ray. "CODIT: Code Editing with Tree-Based Neural Machine Translation". In: *IEEE Transactions on Software Engineering* (May 2019). DOI: 10.1109/TSE.2020.3020502. URL: https://ieeexplore.ieee.org/document/9181462.

[5] Zimin Chen et al. "SEQUENCER: Sequence-to-Sequence Learning for End-to-End Program Repair". In: *IEEE Transactions on Software Engineering* (Sept. 2019). DOI: 10.1109/TSE.2019.2940179. URL: https://ieeexplore.ieee.org/document/8827954.

[6] Sepp Hochreiter and Jurgen Schmidhuber. "Long Short-Term Memory". In: *Neural Computation* (Nov. 1997). URL: https://dl.acm.org/doi/10.1162/neco.1997.9.8.1735.

[7] Nan Jiang, Thibaud Lutellier, and Lin Tan. "CURE: Code-Aware Neural Machine Translation for Automatic Program Repair". In: *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)* (May 2021). DOI: 10.1109/icse43902.2021.00107. URL: http://dx.doi.org/10.1109/ICSE43902.2021.00107.

[8] Rafael-Michael Karampatsis et al. "Big code != big vocabulary". In: *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering* (June 2020). DOI: 10.1145/3377811.3380342. URL: http://dx.doi.org/10.1145/3377811.3380342.

[9] Guillaume Klein et al. "OpenNMT: Open-Source Toolkit for Neural Machine Translation". In: *Proceedings of ACL 2017, System Demonstrations*. Vancouver, Canada: Association for Computational Linguistics, July 2017, pp. 67–72. URL: https://www.aclweb.org/anthology/P17-4012.

[10] Herb Krasner. "The Cost of Poor Quality Software in the US: A 2018 Report". In: *Consortium for IT Software Quality* (Sept. 2018). URL: https://www.it-cisq.org/the-cost-of-poor-quality-software-in-the-us-a-2018-report/The-Cost-of-Poor-Quality-Software-in-the-US-2018-Report.pdf.

[11] Shuai Lu et al. *CodeXGLUE: A Machine Learning Benchmark Dataset for Code Understanding and Generation*. URL: https://arxiv.org/abs/2102.04664. (accessed: 03.15.2021).

[12] Frederic P. Miller, Agnes F. Vandome, and John McBrewster. *Levenshtein Distance: Information Theory, Computer Science, String (Computer Science), String Metric, Damerau?Levenshtein Distance, Spell Checker, Hamming Distance*. Alpha Press, 2009. ISBN: 6130216904.

[13] S. Amirhossein Mousavi, Donya Azizi Babani, and Francesco Flammini. "Obstacles in Fully Automatic Program Repair: A survey". In: *CoRR* abs/2011.02714 (2020). arXiv: 2011.02714. URL: https://arxiv.org/abs/2011.02714.

[14] Nitish Srivastava et al. "Dropout: a simple way to prevent neural networks from overfitting". In: *The Journal of Machine Learning Research* (Jan. 2014). URL: https://dl.acm.org/doi/10.5555/2627435.2670313.

[15] Felix Stahlberg. "Neural Machine Translation: A Review". In: *CoRR* abs/1912.02047 (2019). arXiv: 1912.02047. URL: http://arxiv.org/abs/1912.02047.

[16] Daniel Tarlow et al. "Learning to Fix Build Errors with Graph2Diff Neural Networks". In: *CoRR* abs/1911.01205 (2019). arXiv: 1911.01205. URL: http://arxiv.org/abs/1911.01205.

[17] Michele Tufano et al. "An Empirical Study on Learning Bug-Fixing Patches in the Wild via Neural Machine Translation". In: *ACM Transactions on Software Engineering and Methodology* (Sept. 2019). DOI: 10.1145/3340544. URL: https://doi.org/10.1145/3340544.

[18] Michele Tufano et al. "On Learning Meaningful Code Changes via Neural Machine Translation". In: *41st ACM/IEEE International Conference on Software Engineering* (May 2019). DOI: 10.1109/ICSE.2019.00021. URL: https://doi.org/10.1109/ICSE.2019.00021.

[19] Yuan Yuan and Wolfgang Banzhaf. "Toward Better Evolutionary Program Repair: An Integrated Approach". In: *ACM Transactions on Software Engineering and Methodology* (Jan. 2020). DOI: 10.1145/3360004. URL: https://dl.acm.org/doi/10.1145/3360004.