GGF: A Graph-based Method for Programming Language Syntax Error Correction

Liwei Wu
mg1632006@smail.nju.edu.cn
Nanjing University
Nanjing, Jiangsu, China

Fei Li
mg1732005@smail.nju.edu.cn
Nanjing University
Nanjing, Jiangsu, China

Youhua Wu
mg1632007@smail.nju.edu.cn
Nanjing University
Nanjing, Jiangsu, China

Tao Zheng
zt@nju.edu.cn
Nanjing University
Nanjing, Jiangsu, China

ABSTRACT

Syntax errors combined with obscure error messages generated by compilers usually annoy programmers and cause them to waste a lot of time on locating errors. The existing models do not utilize the structure in the code and just treat the code as token sequences. It causes low accuracy and poor performance on this task. In this paper, we propose a novel deep supervised learning model, called Graph-based Grammar Fix(GGF), to help programmers locate and fix the syntax errors. GGF treats the code as a mixture of the token sequences and graphs. The graphs build upon the Abstract Syntax Tree (AST) structure information. GGF encodes an erroneous code with its sub-AST structure, predicts the error position using pointer network and generates the right token. We utilized the DeepFix dataset which contains 46500 correct C programs and 6975 programs with errors written by students taking an introductory programming course. GGF is trained with the correct programs from the DeepFix dataset with intentionally injected syntax errors. After training, GGF could fix 4054 (58.12%) of the erroneous code, while the existing state of the art tool DeepFix fixes 1365 (19.57%) of the erroneous code.

CCS CONCEPTS

- Software and its engineering → Software defect analysis;
- Computing methodologies → Neural networks.

KEYWORDS

Syntax Error Correction,Deep Learning,GGNN

ACM Reference Format:
Liwei Wu, Fei Li, Youhua Wu, and Tao Zheng. 2020. GGF: A Graph-based Method for Programming Language Syntax Error Correction. In 28th International Conference on Program Comprehension (ICPC '20), October 5–6, 2020, Seoul, Republic of Korea. ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3387904.3389252

ICPC '20, October 5–6, 2020, Seoul, Republic of Korea

ACM ISBN 978-1-4503-7958-8/20/05…$15.00
https://doi.org/10.1145/3387904.3389252

1 INTRODUCTION

Syntax errors in programs are errors detected by syntax checkers (e.g compilers) for not complying with the syntax rules of programming languages. A simple syntax error is often obscured by the misleading error messages generated by compilers. These error messages make syntax errors hard to be located, costing programmers (especially novice programmers) a lot of time.

```
1  #include<stdio.h>
2  int main (){
3      int a,b;
4      scanf("%d %d",
   &a, &b);
5      if (a == 0)
6          b = b + 1;
7      }
8      int c=a+b;
9      printf("%d\n", c);
10 }
```

(a) The erroneous code

```
1  #include<stdio.h>
2  int main (){
3      int a,b;
4      scanf("%d %d",
   &a, &b);
5      if (a == 0) {
6          b = b + 1;
7      }
8      int c=a+b;
9      printf("%d\n", c);
10 }
```

(b) The correct code

Figure 1: The code example

For example, a piece of simple code is shown in Fig. 1a and the correct code is shown in Fig. 1b. There is an obvious syntax error in Fig. 1a (missing a left brace in the 'if' structure). A compilation of this program using GNU Compiler Collection (GCC) 5.5.0 compiler generates the error messages shown in Fig. 2. The error messages generated by the compiler are misleading. None of them gives a hint to programmers' that the error is caused by a missing of left brace, making it even more difficult for programmers to figure out how to fix the syntax error. This simple example given here to some extent demonstrates misleading error

```
main2.c:10:11: error: 'a' undeclared here (not in a function)
    int c=a+b;
          ^
main2.c:10:13: error: 'b' undeclared here (not in a function)
    int c=a+b;
            ^
main2.c:11:12: error: expected declaration specifiers or '...' before
string constant
    printf("%d\n", c);
           ^
main2.c:11:20: error: expected declaration specifiers or '...' before 'c'
    printf("%d\n", c);
                   ^
main2.c:12:1: error: expected identifier or '(' before '}' token
 }
 ^
```

Figure 2: The error messages of the example code

messages could reduce programmers' productivity. In a real-world program with thousands of lines of code, the impact of this problem will be more serious.

The example shows that it is necessary to develop more efficient methods for locating and fixing syntax errors. We define the problem we address in this paper as following [27]: Given a source code file with some syntax errors, how can we help one accurately pinpoint their locations and produce suggestions to fix them? There are two challenges in solving the problem.

The first challenge is that the traditional Recurrent Neural Network(RNN)[16, 19] encoder can not effectively encode the code sequence. In recent papers, some new encoders have been proposed [3, 4] which utilize Abstract Syntax Tree (AST) information and have greatly improved results. However, we can not use the existing encoders directly since the AST information of the erroneous code is incomplete.

To solve this problem, we propose a novel deep supervised learning model, called Graph-based Grammar Fix(GGF). GGF treats the code as a graph. When parsing the erroneous code, the parser may crash but some parts of the AST have been created which is called sub-AST in this paper. The graph is built with sub-AST structure information. Identical variable identifiers are also connected in the graph. A graph example is shown in Fig. 3. Since the parser may crash in the parsing process, there may be some isolated points and some error edges in this graph. To tackle the problem, GGF treats the code as a mixture of graphs and token sequences. Specifically, GGF uses both Gated Recurrent Unit (GRU, a variant of RNN)[9] and Gated Graph Neural Networks (GGNN, a kind of graph encoder)[21] as encoders.

Another challenge is how to fix the code. There are three methods in previous researches. The first is seq-to-seq architecture. In Natural Language Process domain, it is the standard method in the syntax correction which decodes the whole correct texts. However, this architecture is not suitable for the programming language syntax correction, because additional errors usually are added to the output code in the decoding process. The other two methods both leverage an iterative modification mechanism which iteratively predicts the modification action and applies it to the wrong code until the code is fixed. The second is a line replacement mechanism proposed by [13]. It predicts an action which inserts, deletes or replaces a line at a time. This method is more powerful in handling the insertion of a token sequence. However, the result of this model is affected by the programmer's coding style in the training dataset. The third is a token replacement mechanism proposed by [27]. It locates the error position and tries all three actions (insertion, deletion, and replacement) on some predicted tokens. This causes exponential time growth as the number of errors in the code increases. In their paper, they only test their model by fixing the code containing only one error.

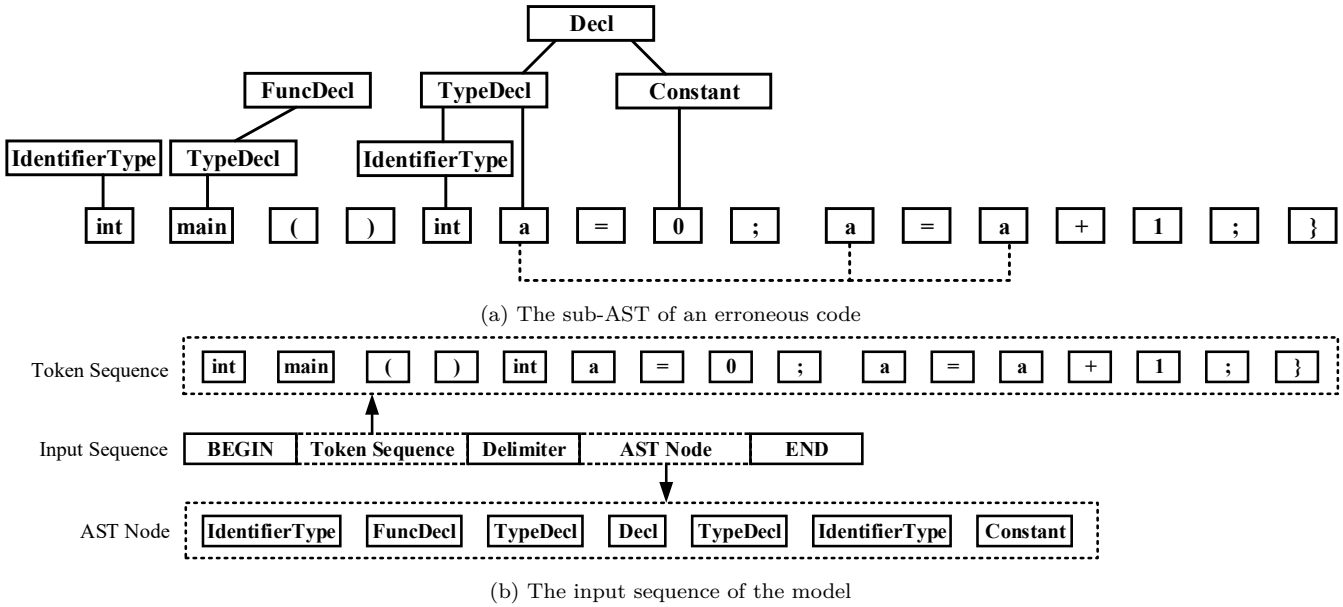
To solve the problems in the existing methods, we propose a new token replacement mechanism which can locate the error and predict the correct token. The error is located by an open interval which has two cases. If the interval contains one token, GGF will replace or delete the token here. If the interval is empty, GGF will insert a token here. Compared to the seq-to-seq architecture, GGF only changes one token at a time and does not need to predict the whole correct texts. When more than one error candidate is found, GGF will pick the candidate with the highest probability and fix it. This could lead to the elimination of other error candidates in the next iteration. However, in the seq-to-seq architecture, the model needs to fix all error candidates at once in the decoding process and this may introduce additional errors in previous decoding steps. Compared to the line replacement mechanism, our method does not need to leverage the text structure which makes our model independent of the coding style in the training dataset. Compared to the token replacement mechanism proposed by [27], our method predicts the probability of different actions so does not need to try the three action types and can fix code containing multiple errors more efficiently.

DeepFix[13], the existing state of the art tool, proposes their dataset [14] which contains 46500 correct C programs and 6975 programs with errors written by students taking an introductory programming course. We intentionally injected syntax errors to the correct programs and trained GGF with it. The model was evaluated on the DeepFix test dataset and got 58.12% accuracy.

2 RELATED WORK

2.1 Programming language syntax error correction

In history, many people are trying to improve the grammar error message in the compiling process [1, 23, 31]. Early research attempts to find a rule-based algorithm to tackle errors at the parsing stage [2, 17, 24], but the results of these methods are not very good, because they lack probabilistic error correction [30]. More recent research focuses on applying language models to syntax error detection and correction. Campbell et al. [7] used the traditional n-gram



(a) The sub-AST of an erroneous code

(b) The input sequence of the model

Figure 3: An example of mixing sub-AST graphs and token sequences

model to locate errors in Java code by calculating the entropy for each syntax error. The higher the entropy is, the more likely it is the true error position. Bhatia et al. [6] is the first paper using a recurrent language model to fix the syntax error. They leveraged the error position proposed by compilers to locate the position and use the language model to predict the token to replace. However, the misleading error messages generated by compilers make the method less effective. Santos et al. [27] also used the language model but they did not leverage the information proposed by the compiler. They trained two recurrent language models, a forward one and a backward one. For each position, models calculate a score to predict whether there is an error here. The score is composed of a global score and a local score. The global score measures the consistency of the distribution predicted by the forward language model and the backward distribution. The local score measures the probability of the token here given by the two language models. Gupta et al. [13] used a seq-to-seq network to iteratively fix one error at a time. For each iteration, an oracle is used to reject, stop or accept this fix. Their neural network utilizes the architecture of the neural network in this paper [29] with the attention mechanism [5].

2.2 Natural language grammatical error correction

Natural language grammar correction is a well-researched problem. In history, some papers in this area have focused on identifying and correcting specific types of grammatical errors such as misuse of verb forms, articles, and prepositions [8, 15, 26]. The recent works consider common grammar errors, often relying on language models, and machine translation [11, 18, 22, 25]. The seq-to-seq model methods are better than the language model methods. However, the seq-to-seq model cannot be borrowed to the programming language domain. The reason is that the dependence of the grammar in the programming language is farther than the dependence in the natural language. The length of the input tokens in the code is much larger than the length in the natural language. The seq-to-seq model needs to decode the complete code and some errors may be introduced during this process.

2.3 Code as graphs

Recently, a latest paper [3] represents programs with graphs. They constructed graphs from code based on the code's known semantics and trained a Gated Graph Neural Networks(GGNN) [21] on the graph. They evaluated the model on two tasks: VARNAMING, in which a network attempts to predict the name of a variable given its usage, and VARMISUSE, in which the network learns to reason about selecting the correct variable that should be used at a given program location. Compared to the less structured methods borrowed from the natural language domain, the GGNN method got a much better result. In our task, the code has errors which result that graphs cannot be constructed from the code, so we treat the code as a mixture of graphs and sequences.

3 METHOD

In this section, we will describe the method used in GGF. GGF fixes the code in a loop. The core component of GGF is a neural network which gets the erroneous code and predicts one fixing action at each iteration in this loop until the code is fixed with no compile errors. The action is composed of

the error position and the right tokens to replace the error tokens. The error position is represented by an open interval (p_1^o, p_2^o) where $1 \le p_2^o - p_1^o \le 2$. When $p_2^o - p_1^o = 1$, the predicted token will be inserted to the gap between the p_1^o-th token and the p_2^o-th token. When $p_2^o - p_1^o = 2$, the $(p_1^o + 1)$-th token will be replaced with the predicted token. Deletion is represented by a special token. The architecture of the neural network is shown in Fig. 4.

3.1 Gated Graph Neural Networks

Gated Graph Neural Networks(GGNN) [21] is an important module in GGF, but it is not widely used as GRU [9]. GGNN is an iterative graph propagation method used to learn the representation of nodes in a graph. A graph $G = (\mathrm{V}, E, X)$ is composed of a set of nodes V, node features X and a list of directed edges $E = (e_1, e_2, ..., e_K)$ where K is the number of edge types. We annotate each $v \in \mathrm{V}$ with a real-valued vector $x^{(v)} \in \mathbb{R}^d$ where d is the input size. The hidden state $h^{(v)}$ is initialized by the label embedding $x^{(v)} \in \mathbb{R}^d$. To propagate the message through the graph, messages of type k is sent from node v to its neighbours. The message $m_k^{(v)}$ is calculated from the hidden state of each node. The node v aggregates all the messages from its neighbours and updates its hidden state by GRU iteratively. The message is propagated by:

$$m_k^{(v)} = f_k(h^{(v)}) \tag{1}$$

$$\tilde{m}^{(v)} = g(m_k^{(u)} \mid \text{there is an edge of type } k \text{ from } u \text{ to } v) \tag{2}$$

$$h'^{(v)} = \mathrm{GRU}(h^{(v)}, \tilde{m}^{(v)}) \tag{3}$$

where v represents a node, k is the edge type, f_k is the message function and g is the aggregating function.

In our model, we regard all edges as the same edge type. The function g is the element-wise summation and the function f_k is the identity function.

3.2 Code as the mixture of token sequences and graphs

Contrary to previous work [13], we represent the code as a mixture of the graph and the token sequence. We treat the AST as a graph to take advantage of AST structure information and AST node information. However, we can only generate a part of AST with few AST nodes since the code has syntax errors. This leads us to treat the code as a token sequence with a sub-AST graph. As shown in Fig. 3a, the code is parsed by a LALR parser[10] and some error happened in the parse process. We only get a sub-AST in Fig. 3a. From the perspective of the sequence, we take the code as the concatenation of the token sequence and the AST node sequence. The two sequences are connected by a special delimiter token. The AST node sequence is generated by traversing the sub-AST forest in pre-order [12]. The sequence representation of the code in Fig. 3a is shown in Fig. 3b.

From the perspective of the graph, the AST node tokens in the input sequence are connected to their children and parents. The connection of AST nodes is shown as the solid lines in Fig. 3a. The identifier tokens with the same name are also connected in the graph showed by the dashed line in Fig. 3a. It makes the information exchange of the same token independent of the location. To represent the graph structure, an adjacent matrix is created according to the lines in Fig. 3a. The item in the adjacent matrix is 1 if the tokens are connected, otherwise 0. By the connected input sequence, we combine the literal token information with sub-AST structure information.

3.3 Encoding erroneous code

The input of GGF is a sequence of tokens $I = [w_1, w_2, ..., w_L]$ and an adjacent matrix $A \in \mathbb{R}^{L \times L}$ where L is the length of the token sequence. The token sequence here contains both code tokens and the AST node tokens. The adjacent matrix records the links of the connected input sequence. A linear function follows the word embedding to transform the vector size from the embedding size n to the hidden size d. The corresponding token embedding is calculated by:

$$x_i = E[w_i], \qquad i = 1, 2, ..., L \tag{4}$$

$$h_i^{(0)} = W_t x_i + b, \qquad i = 1, 2, ..., L \tag{5}$$

where $E \in \mathbb{R}^{V \times n}$ is the embedding matrix. V is the vocabulary size and n is the embedding size. $W_t \in \mathbb{R}^{d \times n}$ and $b \in \mathbb{R}^d$ are the parameters. The encoder layer is composed of a BiGRU layer [9] following by a GGNN layer [21]. The result of the ith encoding layer is calculated by:

$$[t_1^{(i)}, t_2^{(i)}, ..., t_L^{(i)}] = \mathrm{BiGRU}^i([h_1^{(i-1)}, h_2^{(i-1)}, ..., h_L^{(i-1)}]) \tag{6}$$

$$[h_1^{(i)}, h_2^{(i)}, ..., h_L^{(i)}] = \mathrm{GGNN}^i([t_1^{(i)}, t_2^{(i)}, ..., t_L^{(i)}], A) \tag{7}$$

where $h_j^{(i)} \in \mathbb{R}^d, j = 1, 2, ..., L$ are the result of the ith encoding layer and $t_j^{(i)} \in \mathbb{R}^d, j = 1, 2, ..., L$ are the intermediate values in the ith encoding layer. So the result of the encoding module is the $h_j^{(L_e)}, j = 1, 2, ..., L$ where L_e is the encoding layer number. For the sake of simplicity, L_e is omitted when the output of the encoding module is mentioned below.

3.4 Finding the error position

To find the error position, we predict the begin position p_1^o and the end position p_2^o by two pointer-networks. The token between the begin position p_1^o and the end position p_2^o is the erroneous code snippet. The begin position token and the end position token are not contained in the erroneous code snippet. To express some task-related information, two trainable query vectors q_1 and q_2 are added to the pointer-networks. They add some task-related bias to the pointer-networks. The begin position distribution p_1 is calculated by:

$$h_j^1 = \tanh(W_h^1 h_j + W_q^1 q_1)^T q_1 \tag{8}$$

$$p_1 = \mathrm{softmax}([h_1^1, h_2^1, ..., h_{L_t}^1]) \tag{9}$$

where $q_1 \in \mathbb{R}^d$ is the trainable query vector, the superscript T represents the transpose operation, h_j^1 is the logit of the

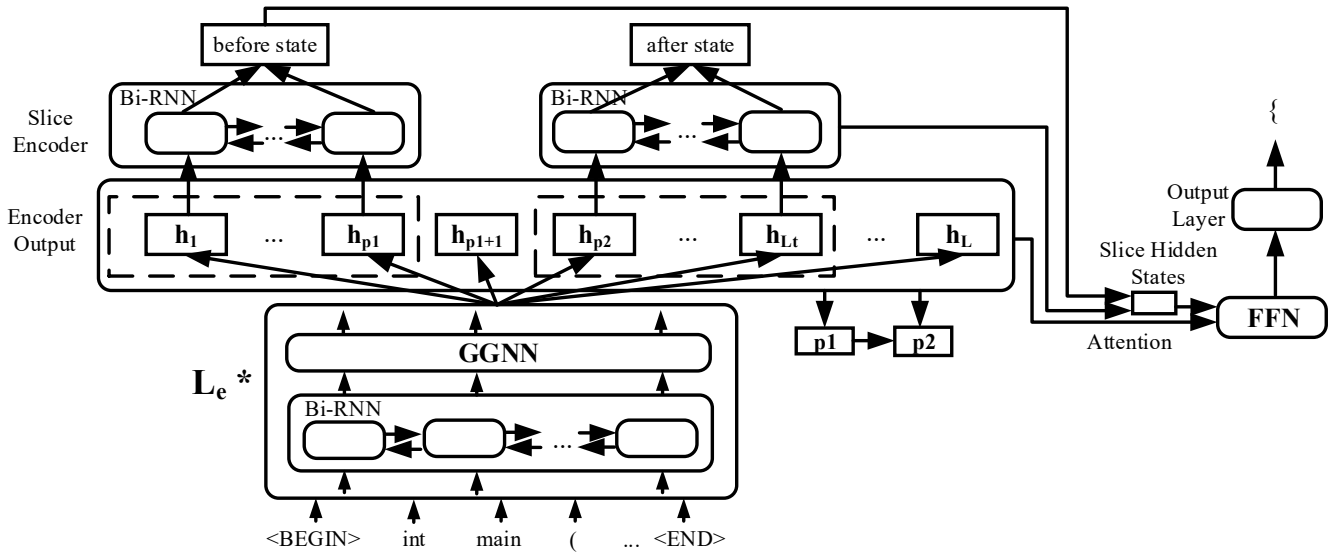


Figure 4: The Model Architecture

corresponding position, $W_h^1, W_q^1 \in \mathbb{R}^{d\times d}$ are the weight matrixes and L_t is the length of the code tokens. The trainable query vector q_1 is initialized from a standard normal distribution. The query vector q_2 is calculated by a GRU cell with q_1 and t_q:

$$t_q = \sum_{i=1}^{L_t}(p_{1[i]}h_i) \tag{10}$$

$$q_2 = \mathrm{GRU}_u(q_1, t_q) \tag{11}$$

where the subscript $[i]$ means getting i-th value in the vector, h_i is the output of the encoder and t_q is a weighted context over the encoder output. Because the model predicts one token at a time, the scope of calculating p_2 is limited in $[p_1^o+1, p_1^o+2]$. The end position distribution p_2 is calculated with the q_2 as the query vector by:

$$h_j^2 = \tanh(W_h^2 h_j + W_q^2 q_2)^T q_2 \tag{12}$$

$$p_2 = \mathrm{softmax}([h^2_{p_1^o+1}, h^2_{p_1^o+2}]) \tag{13}$$

where p_1^o is the begin position. In the evaluation process, the begin position and the end position is calculated by:

$$p_1^o = \mathrm{argmax}(p_1) \tag{14}$$

$$p_2^o = \mathrm{argmax}(p_2) \tag{15}$$

In the training process, the begin position p_1^o and the end position p_2^o are the proposed in the dataset.

3.5 Generating the right token

The GGF will generate one token at a time. To generate the right token, we calculate the context vector h_d. Here, a BiGRU layer is used to encode the text before the begin position p_1^o and the text after the end position p_2^o. The context vector is calculated by:

$$h_p^d = \mathrm{BiGRUState}([h_1, h_2, ..., h_{p_1^o}]) \tag{16}$$

$$h_a^d = \mathrm{BiGRUState}([h_{p_2^o}, h_{p_2^o+1}, ..., h_{L_t}]) \tag{17}$$

$$h^d = [h_p^d; h_a^d] \tag{18}$$

where *BiGRUState* is a BiGRU layer. Because the meaning of the identifier in the code is mainly determined by its context, selecting right tokens in context is easier and more efficient than sampling right tokens in the vocabulary.

In the process of generating, we try to avoid sampling the token from the token vocabulary directly but copy the token from the erroneous code. However, some keywords and identifiers defined by the library are not contained in the original code, so sampling the output token from the vocabulary is also needed. GGF generates a copy decision c to decide whether to copy from the code. If the copy decision is yes, the fixing token is sampled from the input code, expressed as w^c. In contrast, if the copy decision is no, the fixing token is sampled from the vocabulary, expressed as w^s. So a fixing action can be expressed as a tuple $(p_1^o, p_2^o, c, w^c, w^s)$. The copy decision c is determined by the following equations:

$$h_t^o = \mathrm{MLFFN}_d(h^d) \tag{19}$$

$$h^o = \mathrm{Attention}(h_t^o, [h_1, h_2, ..., h_L]) \tag{20}$$

$$c = \mathrm{sigmoid}(W_c h^o + b_c) \tag{21}$$

where $h^o \in \mathbb{R}^d$ is the copy decision hidden state, $W_c \in \mathbb{R}^{d\times d}$ and $b_c \in \mathbb{R}^d$ are the weight matrix and the bias and c is the probability to copy the token from the original code. MLFFN is a multilayer feedforward neural network which maps the vector $h^d \in \mathbb{R}^{2d}$ to the vector $h_t^o \in \mathbb{R}^d$. The attention used here is the soft attention mechanism [5]. In evaluation process, the decision whether to copy from the input code is

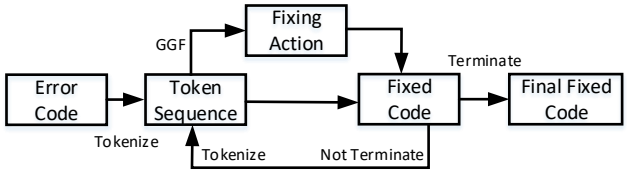


Figure 5: The iterative fixing process.

decided by the probability c. If $c > 0.5$, the sampled token will be copied from the code, else the token will be sampled from the vocabulary. In training process, the decision is proposed by the target.

The copied token w^c is calculated by:

$$h^t = \mathrm{GRU}_u^o(h^o, s_c) \tag{22}$$

$$o^c = \mathrm{softmax}([h_1^T h^t, h_2^T h^t, ..., h_{L_t}^T h^t]) \tag{23}$$

$$w^c = I[\mathrm{argmax}(o^c)] \tag{24}$$

where I is the token list, GRU_u^o is the GRU cell used to update the copy decision hidden state to the output hidden state and $s_c \in \mathbb{R}^d$ is a trainable parameter representing the copy decision. h^T is the output of the encoder.

The token sampled from the vocabulary w^s is calculated by:

$$h^t = \mathrm{GRU}_u^o(h^o, s_s) \tag{25}$$

$$o^s = \mathrm{softmax}(W_o h^t) \tag{26}$$

$$w^s = \mathrm{argmax}(o^s) \tag{27}$$

where $s_s \in \mathbb{R}^d$ is a trainable parameter representing the sample decision and $W_o \in \mathbb{R}^{d \times d}$ is the weight matrix.

3.6 Loss function

A simple linear combination of the cross entropy functions between outputs and targets is used as loss function in GGF:

$$\begin{aligned}\mathrm{Loss} = {} & \mathrm{CrossEntropy}(p_1^t, p_1) + \mathrm{CrossEntropy}(p_2^t, p_2) \\ & + \mathrm{CrossEntropy}(c^t, c) + c^t \mathrm{CrossEntropy}(w^{ct}, o^c) \\ & + (1 - c^t)\mathrm{CrossEntropy}(w^{st}, o^s)\end{aligned} \tag{28}$$

where p_1^t, p_2^t, c^t, w^{ct} and w^{st} are the targets in the train dataset.

3.7 Fix multiple errors iteratively

There are many erroneous code snippets in one code and the length of each code snippets can be more than 1. To fix the complex erroneous code, we introduce an iterative fixing strategy. The iterative fixing process is shown in Fig. 5. For each erroneous code, we fix it in a loop. In each iteration, the code is tokenized as a token sequence. The token sequence is input to the GGF model and a fixing action is output. One fixing action predicts one possible error position and the corresponding fixing tokens. At the end of the iteration, a fixed code is generated according to the token sequence and the fixing action. The fixed code is input to the next iteration until the termination condition is met. We have two termination conditions: the fixed code can be compiled and the number of iterations exceeds the limitation. We choose the last fixed code as the final fixed code.

The iterative fixing strategy is a practical method. One advantage is that the implements of models are less complex. Since models focus on one error rather than multiple errors, the output is simple and the decoders are less complex. The other advantage is that the results may be better. There are several possible errors in one erroneous code. In each iteration, the model doesn't need to fix all errors but the most likely error. After fixing the most likely syntax error, the erroneous code applied the fixing action contains fewer errors. It may help to fix other errors, especially indeterminate ones.

4 EXPERIMENT

4.1 Dataset

We trained GGF with two different training datasets respectively. One is the dataset used in DeepFix [13] for the programming language syntax correction task. The programs in the dataset span 93 programming problems in an introductory programming course and make use of nontrivial C language constructs. There is 46500 correct code in the training dataset. We removed the code which has more than 500 tokens or cannot be tokenized. There is 29325 code in the final DeepFix training dataset. We intentionally injected random syntax errors to the correct code in the training dataset. The generated code was checked by the compiler to ensure that it can not be compiled. We split 3189 code from the training dataset as the validation dataset to check if the model overfits on the training dataset. The other is Codeforces dataset which is crawled from Codeforces[1] website as a supplement. After filtering out the code which has more than 500 tokens or cannot be tokenized, there is 35125 correct code in the Codeforces training dataset. We split 1847 code from the training dataset as the validation dataset.

Since the lack of labelled erroneous code for training, we intentionally introduced syntax errors into the correct code in the training dataset. We insert, delete and modify the tokens randomly in the correct code. We define one token-level insertion, deletion or modification as one mutation operation. Up to 5 mutations are allowed from each correct code. Additionally, the substitute tokens and substituted tokens are the same kinds of tokens such as keywords or identifiers. On the Codeforces training dataset, we did the same syntax error introduction.

We use the same test dataset as DeepFix [13]. The test dataset contains 6975 real erroneous programs with 16766 compilation error messages together. We tested all the models on this test dataset.

[1] codeforces.com, an online judge website.

Table 1: Results for different error types

Error Types	Fixed Message Number	Total Message Number	Fixed Code Number	Total Code Number
identifier undeclared	2813(51.85%)	5425	1769(51.71%)	3421
expected delimiters	2876(78.43%)	3667	2358(78.63%)	2999
expected identifier	22(1.27%)	1736	485(35.77%)	1356
expected expression	977(74.30%)	1315	774(74.93%)	1033

4.2 Implementation details

All the experiments run on an Intel(R) Core(TM) i7-6950X machine clocked at 3.00GHz with 64GB of RAM and equipped with an NVIDIA GeForce GTX 1080Ti GPU.

The vocabulary is generated from the training dataset and contains 5991 words, including 84 keywords, 5852 identifiers, 48 ast nodes, and 7 special words. All tokens out of vocabulary are represented as an unk token. The embedding size is 400. The hidden size is 400. There are 3 layers of all GRU layers and GGNN layers. A dropout with 0.2 drop probability is inserted into the adjacent GRU layers and GGNN layers. The layer number of the MLFFN layer is also 3. The model is trained by the Adam optimizer [20] with an initial learning rate of 0.0000625 which is linear decayed. The β_1, β_2 and ϵ are set to 0.9, 0.999 and 10^{-8} respectively. We use the GCC 5.5.0 as the C compiler in our experiment. We use the PyCparser which is a parser for the C language, written in pure Python, as the down-top parser to generate the sub-AST structure. In the evaluation process, our model can only modify the erroneous code up to 10 times.

4.3 Evaluation

In our experiment, we compared GGF with two previous models: DeepFix [13] and GrammarGuru [27]. The DeepFix result used here was cloned from their repository[2][14]. Because we do not have the training dataset used in their repository, we also retrained the model of DeepFix with our training dataset. GrammarGuru is trained by the same setting with GGF. GrammarGuru cannot fix multiple errors in the code restricted to its high time complexity, so we compared it to GGF on fixing the erroneous code in one step. They were evaluated by three metrics as in DeepFix paper[13]:

- Exact Match(EM): the percentage of the code successfully passing the compiler check.
- Partial Match(PM): the percentage of the code which does not pass the compiling process but decreases the number of error messages.
- Error Message Resolved(EMR): the percentage of the resolved error messages in the total error messages.

The results of the three models are shown in the Table 2. The GGF(Codeforces dataset) means the GGF model trained by the Codeforces dataset. The results of Deepfix(Website) are cloned from their repository. Other experiments are all trained by the Deepfix training dataset. All experiments use the Deepfix test dataset with the real error code.

Table 2: The result of models

Model	EM	PM	ERM
GGF	58.12%	14.35%	48.69%
GGF (Codeforces dataset)	44.30%	11.53%	12.61%
DeepFix (Website)	33.36%	22.32%	40.77%
DeepFix	19.57%	15.23%	24.75%
GGF (one step)	31.03%	26.75%	28.89%
GrammarGuru (one step)	0.86%	13.43%	-58.91%

The EM score of the GGF trained by the DeepFix dataset is 58.12%, the PM score is 14.35% and the EMR score is 48.69%. The GGF trained with Codeforces dataset gets 44.30% on EM score, 11.53% on PM score, 12.61% on ERM score. In one step evaluation, the GGF gets 31.03% on EM score, 26.75% on PM score, 28.89% on ERM score. The GGF is much better than the previous models in the EM metrics whichever the training dataset we use. Although our model is worse than the DeepFix model in the PM metrics, the sum of the EM and PM metrics is larger than the DeepFix model. This shows that GGF can give more meaningful fixing actions.

The model trained by the DeepFix training dataset has a better effect than the model trained by the Codeforces dataset. It shows that the network learns not only program syntax but also other program information. Even if the model is trained by the Codeforces dataset, the GGF gets high results in the DeepFix test dataset.

The retrained DeepFix model is not as good as they show. There could be two reasons. One is that we injected more complex errors which their model cannot handle. The other is that they used two separate neural networks for fixing the typographic and the missing variable declaration errors in their repository.

4.4 Ablation tests

To evaluate the effectiveness of each sub-network, we did ablation tests on GGF. The result is shown in the Table 3. The following models are all trained with the DeepFix training dataset. First, removing the GGNN layers from the encoder in GGF decreases the EM score by about 21.58% which shows the importance of the GGNN layers. Second, replacing the GRU layers from the encoder in GGF with the additional graph links between the tokens adjacent to each other in the sequence decreases the EM score by about 28.00%

[2] iisc-seal.net/deepfix

Table 3: Ablation Tests

Models	EM	PM	ERM
GGF	58.12%	14.35%	48.69%
GGF - GGNN	36.54%	15.90%	23.39%
GGF - GRU	30.12%	9.29%	-35.08%
GGF - token replacement + sequence decoder	37.61%	22.22%	35.69%
GGF - copy output	47.48%	14.52%	38.30%
GGF - GGNN - copy output	36.72%	14.27%	20.51%

```
1  #include<stdio.h>
2  int main ( ) {
3     int n , r [ 100 ] , t , i ;
4     scanf ( "%d" , & n ) ;
5     r [ 0 ] = 1 ;
6     for ( i = 1 ; i < n ; i ++ ) {
7        scanf ( "%d" , & r [ i ] ) ;
8     }
9     for ( i = 0 ; i < n ; i ++ ) {
10       printf ( "%d" , r[ i ] ) ;
11    }
12 }
13 return 0 ;
14 }
```

(a) The erroneous code

```
1  #include<stdio.h>
2  int main ( ) {
3     int n , r [ 100 ] , t , i ;
4     scanf ( "%d" , & n ) ;
5     r [ 0 ] = 1 ;
6     for ( i = 1 ; i < n ; i ++ ) {
7        scanf ( "%d" , & r [ i ] ) ;
8     }
9     for ( i = 0 ; i < n ; i ++ ) {
10       printf ( "%d" , r[ i ] ) ;
11    }
12 +
13 return 0 ;
14 }
```

(b) The correct code

Figure 6: The code example with mismatched brace error

which shows the importance of the GRU layers. The previous two experiments show that treating code as the pure token sequences or the pure graphs is not enough for the syntax correction task. Thirdly, we replaced the token replacement mechanism with a GRU sequence decoder with attention which predicts the correct code corresponding to the erroneous code. In the GRU decoder, the prediction is also can be copied from the input code or sampled from the vocabulary. This change decreases the EM score by about 20.51%. The result shows that the token replacement mechanism is very important in the programming language syntax correction task. Fourthly, replacing the output module with the pure sample output module decreases the EM score by 10.64% which shows the copy mechanism is important. Finally, to compare our token replacement mechanism with the line replacement mechanism in the DeepFix, we removed the GGNN layer and the copy output module from GGF. It decreases the EM score by about 21.40%. The results between this comparative model and the DeepFix model show that our token replacement mechanism is better than the line replacement mechanism.

4.5 Results of GGF on different error types

To compare the results of GGF for different error types, we grouped the error message by some templates and compared the changes in the number of different groups. For each group, we also counted the code containing it and the corresponding fixed code. We sorted the groups by the message number decreasingly and showed top-4 in the Table 1. It is obvious that GGF differs greatly in its ability to solve different errors. GGF has very poor results on the 'expected identifier' error. Braces mismatching is the main reason and it causes a completely wrong sub-AST. The wrong sub-AST structure greatly misleads our model. An example is shown in Fig. 6. An erroneous code which has an extra right brace in line 12 is shown in Fig. 6a and the corresponding correct code is shown in Fig. 6b. A compilation of this program using GCC 5.5.0 compiler generates the error messages in Listing. 1

```
.\code_t.c:14:1: error: expected identifier
    or '(' before 'return'
 return 0 ;
 ^
.\code_t.c:15:1: error: expected identifier
    or '(' before '}' token
 }
 ^
```

Listing 1: Error Message of Example Code in Fig. 6a

It is easy to tell that the 'expected identifier' error occurs in this code. If we remove the last two lines in the erroneous code, the code will be correct. So in the down-top parsing process, the parser will mismatch the right brace in line 12 with the left brace in line 2 and a wrong sub-AST structure will be created. This makes GGF hard to fix the erroneous code. It shows that AST information is important for GGF to fix syntax errors. This is also the reason that our second ablation test (GGF - GRU in Table 3) gets a very poor ERM score.

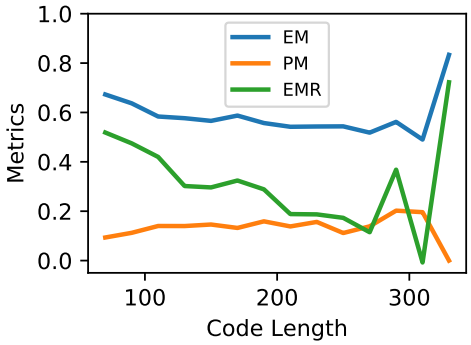


Figure 7: Results for erroneous code with different length.

4.6 The effect of the length of the erroneous code

We grouped the code of similar length in the test dataset. We computed the EM score, PM score and the EMR score per group, as shown in Fig. 7. In the figure, the EM score remains stable for most of the code length. It shows that the results of GGF are not affected by code length. It shows

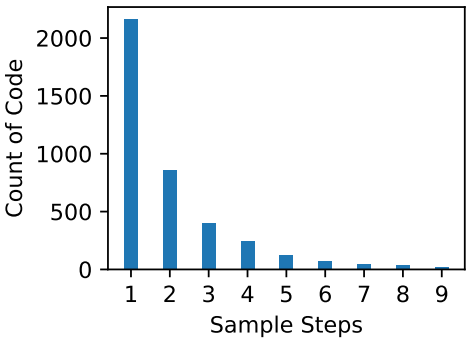


Figure 8: Counting of different sample steps

that some key information of the correction task is learned by GGF which is not affected by code length. When the code length is too small or too large, the curves in the graph fluctuate very badly, because the number of the too short or too long code is very small and the variance of the scores is very high.

4.7 The step number to fix the erroneous code

To show that GGF does not change the original meaning of the erroneous code, the distribution of the step number is shown in Fig. 8. Most of the code is fixed between one and three steps. For the fixed code, GGF retains the original meaning well.

5 CONCLUSION

In this paper, we proposed a novel neural network, called Graph-based Grammar Fix (GGF) to fix the programming language syntax error. GGF utilizes a mixture of the GRU and the GGNN as the encoder module and a token replacement mechanism as the decoder module. The model encodes the token information with sub-AST information and generates the fixing actions. The model trained with the DeepFix dataset gets 58.12% EM score and the model trained with Codeforces dataset gets 44.30% EM score. It proves that the architecture used in GGF is quite useful for the programming language syntax error correction task.

6 ACKNOWLEDGEMENT

The project is supported by the National Natural Science Foundation of China (Grant No. 61373010) and the Key Project of Science and Technology Department of Jiangsu (Grant No. BE2017004).

REFERENCES

[1] Umair Z Ahmed, Pawan Kumar, Amey Karkare, Purushottam Kar, and Sumit Gulwani. 2018. Compilation error repair: for the student programs, from the student programs. In 2018 IEEE/ACM 40th International Conference on Software Engineering: Software Engineering Education and Training (ICSE-SEET). IEEE, 78–87.

[2] Alfred V Aho and Thomas G Peterson. 1972. A minimum distance error-correcting parser for context-free languages. SIAM J. Comput. 1, 4 (1972), 305–312.

[3] Miltiadis Allamanis, Marc Brockschmidt, and Mahmoud Khademi. 2017. Learning to represent programs with graphs. arXiv preprint arXiv:1711.00740 (2017).

[4] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. 2018. code2vec: Learning Distributed Representations of Code. arXiv preprint arXiv:1803.09473 (2018).

[5] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. 2014. Neural machine translation by jointly learning to align and translate. arXiv preprint arXiv:1409.0473 (2014).

[6] Sahil Bhatia and Rishabh Singh. 2016. Automated correction for syntax errors in programming assignments using recurrent neural networks. arXiv preprint arXiv:1603.06129 (2016).

[7] Joshua Charles Campbell, Abram Hindle, and José Nelson Amaral. 2014. Syntax errors just aren't natural: improving error reporting with language models. In Proceedings of the 11th Working Conference on Mining Software Repositories. ACM, 252–261.

[8] Martin Chodorow, Joel R Tetreault, and Na-Rae Han. 2007. Detection of grammatical errors involving prepositions. In Proceedings of the fourth ACL-SIGSEM workshop on prepositions. Association for Computational Linguistics, 25–30.

[9] Junyoung Chung, Caglar Gulcehre, KyungHyun Cho, and Yoshua Bengio. 2014. Empirical evaluation of gated recurrent neural networks on sequence modeling. arXiv preprint arXiv:1412.3555 (2014).

[10] Frank DeRemer and Thomas Pennello. 1982. Efficient computation of LALR (1) look-ahead sets. ACM Transactions on Programming Languages and Systems (TOPLAS) 4, 4 (1982), 615–649.

[11] Tao Ge, Furu Wei, and Ming Zhou. 2018. Reaching Human-level Performance in Automatic Grammatical Error Correction: An Empirical Study. arXiv preprint arXiv:1807.01270 (2018).

[12] Michael Goodrich, Roberto Tamassia, and David Mount. 2007. DATA STRUCTURES AND ALOGORITHMS IN C++. John Wiley & Sons.

[13] Rahul Gupta, Soham Pal, Aditya Kanade, and Shirish Shevade. 2017. DeepFix: Fixing Common C Language Errors by Deep Learning.. In AAAI. 1345–1351.

[14] Rahul Gupta, Soham Pal, Aditya Kanade, and Shirish Shevade. 2017. DeepFix: Fixing Common C Language Errors by Deep Learning. http://www.iisc-seal.net/deepfix [Online; accessed 09-04-2018].

[15] Na-Rae Han, Martin Chodorow, and Claudia Leacock. 2006. Detecting errors in English article usage by non-native speakers. Natural Language Engineering 12, 2 (2006), 115–129.

[16] John J Hopfield. 1982. Neural networks and physical systems with emergent collective computational abilities. Proceedings of the national academy of sciences 79, 8 (1982), 2554–2558.

[17] Clinton L Jeffery. 2003. Generating LR syntax error messages from examples. ACM Transactions on Programming Languages and Systems (TOPLAS) 25, 5 (2003), 631–640.

[18] Jianshu Ji, Qinlong Wang, Kristina Toutanova, Yongen Gong, Steven Truong, and Jianfeng Gao. 2017. A nested attention neural hybrid model for grammatical error correction. arXiv preprint arXiv:1707.02026 (2017).

[19] Michael I Jordan. 1997. Serial order: A parallel distributed processing approach. In Advances in psychology. Vol. 121. Elsevier, 471–495.

[20] Diederik P Kingma and Jimmy Ba. 2014. Adam: A method for stochastic optimization. arXiv preprint arXiv:1412.6980 (2014).

[21] Yujia Li, Daniel Tarlow, Marc Brockschmidt, and Richard Zemel. 2015. Gated graph sequence neural networks. arXiv preprint arXiv:1511.05493 (2015).

[22] Hwee Tou Ng, Siew Mei Wu, Ted Briscoe, Christian Hadiwinoto, Raymond Hendy Susanto, and Christopher Bryant. 2014. The CoNLL-2014 shared task on grammatical error correction. In Proceedings of the Eighteenth Conference on Computational Natural Language Learning: Shared Task. 1–14.

[23] Sagar Parihar, Ziyaan Dadachanji, Praveen Kumar Singh, Rajdeep Das, Amey Karkare, and Arnab Bhattacharya. 2017. Automatic grading and feedback using program repair for introductory programming courses. In Proceedings of the 2017 ACM Conference on Innovation and Technology in Computer Science Education. ACM, 92–97.

[24] Terence Parr and Kathleen Fisher. 2011. LL (*): the foundation of the ANTLR parser generator. ACM Sigplan Notices 46, 6 (2011), 425–436.

[25] Alla Rozovskaya, Kai-Wei Chang, Mark Sammons, Dan Roth, and Nizar Habash. 2014. The Illinois-Columbia system in the CoNLL-2014 shared task. In Proceedings of the Eighteenth Conference on Computational Natural Language Learning: Shared Task. 34–42.
[26] Alla Rozovskaya and Dan Roth. 2010. Generating confusion sets for context-sensitive error correction. In Proceedings of the 2010 conference on empirical methods in natural language processing. Association for Computational Linguistics, 961–970.
[27] Eddie A Santos, Joshua C Campbell, Abram Hindle, and José Nelson Amaral. 2017. Finding and correcting syntax errors using recurrent neural networks. PeerJ PrePrints (2017).
[28] Rishabh Singh, Sumit Gulwani, and Armando Solar-Lezama. 2013. Automated feedback generation for introductory programming assignments. Acm Sigplan Notices 48, 6 (2013), 15–26.
[29] Ilya Sutskever, Oriol Vinyals, and Quoc V Le. 2014. Sequence to sequence learning with neural networks. In Advances in neural information processing systems. 3104–3112.
[30] Richard A Thompson. 1976. Language correction using probabilistic grammars. IEEE Trans. Comput. 100, 3 (1976), 275–286.
[31] Jooyong Yi, Umair Z Ahmed, Amey Karkare, Shin Hwei Tan, and Abhik Roychoudhury. 2017. A feasibility study of using automated program repair for introductory programming assignments. In Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering. ACM, 740–751.