MultiFix: Learning to Repair Multiple Errors by Optimal Alignment Learning

Hyeon-Tae Seo[1,2], **Yo-Sub Han**[1], **Sang-Ki Ko**[3]
[1]Yonsei University
[2]Korea Electronics Technology Institute
[3]Kangwon National University
{dchs504,emmous}@yonsei.ac.kr, sangkiko@kangwon.ac.kr

Abstract

We consider the problem of learning to repair erroneous C programs by learning optimal alignments with correct programs. Since the previous approaches fix a single error in a line, it is inevitable to iterate the fixing process until no errors remain. In this work, we propose a novel sequence-to-sequence learning framework for fixing multiple program errors at a time. We introduce the edit-distance-based data labeling approach for program error correction. Instead of labeling a program repair example by pairing an erroneous program with a line fix, we label the example by paring an erroneous program with an optimal alignment to the corresponding correct program produced by the edit-distance computation. We evaluate our proposed approach on a publicly available dataset (DeepFix dataset) that consists of erroneous C programs submitted by novice programming students. On a set of 6,975 erroneous C programs from the DeepFix dataset, our approach achieves the state-of-the-art result in terms of full repair rate on the DeepFix dataset (without extra data such as compiler error message or additional source codes for pre-training).

1 Introduction

Recurrent neural networks (RNNs) (Hochreiter and Schmidhuber, 1997; Cho et al., 2014) are one of the fundamental concepts in deep learning to process sequential data such as text, speech and time-series data. In particular, RNNs already have become a standard and general-purpose tool for various natural language tasks and successfully replaced conventional methods.

The encoder-decoder sequence-to-sequence architecture (Bahdanau et al., 2015; Luong et al., 2015; Sutskever et al., 2014) is becoming the de facto standard for translating source sequences into target sequences. While the most popular application of the sequence-to-sequence learning is machine translation, there are many approaches for adopting this method to other fields such as dialogue systems, text summarization, program synthesis, grammatical error detection and correction.

Automatic program repair is one of the applications where the sequence-to-sequence learning framework has been successfully employed. By automatically localizing the compilation errors and suggesting the possible fixes to the programmers, we can dramatically improve the productivity of programmers. While there have been many approaches to the problem of repairing programs using rule-based algorithms, it is very difficult to implement a useful rule-based repair program as there are too many cases to consider. Therefore, applying deep neural networks and learning-based framework to automatic program repair is an inevitable consequence.

In this work, we consider the problem of learning to repair erroneous C programs based on optimal alignment learning. Figure 1 describes our approach. Given a potentially erroneous program p, our goal is to train a sequence-to-sequence model that takes p as input sequence and produces a sequence of edits that repairs compilation errors from p. There are two major challenges in learning to repair programs. First, it is very difficult to localize multiple errors in a program and correct the errors simultaneously. Previous approaches (Gupta et al., 2017, 2019; Hajipour et al., 2019) exploited sequence-to-sequence models to learn the mapping between erroneous programs and line fixes. They train a model that repairs only one error at a time. If there exist multiple errors in several lines of code, then it is inevitable to iterate the same process.

In light of these observations, we propose the following ideas to improve the previous program repair models.

1. We propose a sequence-to-sequence model that learns the relationship between erroneous programs and their optimal alignments to correct programs that compile without errors. As

Findings of the Association for Computational Linguistics: EMNLP 2021, pages 4850–4855

a result, our model can fix multiple errors at a time while the previous approaches are able to fix only one error at a time.

2. We present a program repair framework that consists of a single network trained with many types of compilation errors. While most of the previous approaches attempt to use separate models to fix each type of errors (for instance, missing declarations and typos), our approach relies on a single sequence-to-sequence model that learns to fix different types of errors in multiple lines of input program.

2 Background

Our approach is based on the sequence-to-sequence learning framework which is originally proposed for the task of machine translation. On top of the idea of sequence-to-sequence learning, we incorporate several ideas for optimizing the training performance on the domain of program repair including the optimal alignment output encoding and synchronized position embedding. Here we briefly provide an overview of each idea.

Learning-based program repair. Learning-based automatic program repair is gaining its popularity especially for correction of introductory programming assignments submitted by novice students (Pu et al., 2016; Ahmed et al., 2018). Gupta et al. (2017) introduced a sequence-to-sequence model with an attention mechanism that fixes errors by constructing program text from different kinds of tokens such as types, keywords, special characters (e.g., semicolons), functions, literals and variables in C programs. Ahmed et al. (2018) trained a sequence-to-sequence prediction model with RNN networks for automatically repairing compile-time errors from the student programs.

Hajipour et al. (2019) propose to adopt a deep generative model for sampling diverse fixes for given erroneous programs. Very recently, Yasunaga and Liang (2020) propose a graph-based self-supervised program repair framework. They utilize a variant of graph neural networks called the graph attention network (Velickovic et al., 2018) to enable more efficient information flow between relevant tokens in a program. Moreover, they leverage diagnostic feedback offered by compiler messages and show that it plays a crucial role in locating errors and learning how to fix them.

Meanwhile, Mesbah et al. (2019) (DeepDelta) tackle the problem of repairing Java build errors by extracting abstract syntax trees (AST) changes between the failed and resolved programs. Later, Tarlow et al. (2020) (Graph2Diff) utilize the graph neural networks to encode the input program and generate the fix. We do not compare DeepDelta and Graph2Diff with our result as they consider different types of build errors in Java while we consider common compiler errors in C including typos, missing declarations, type errors, missing delimiters and so on.

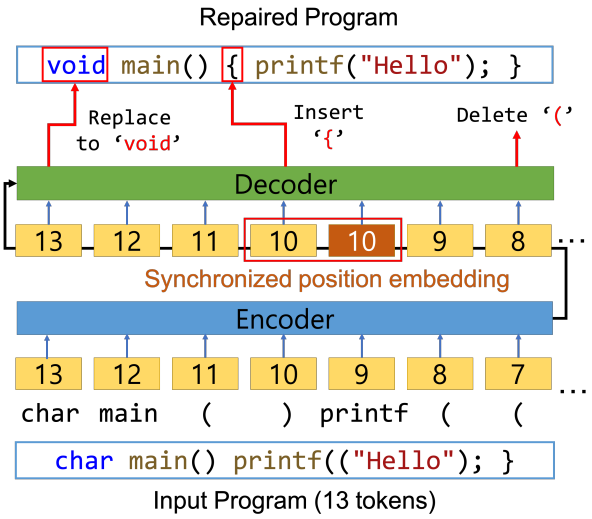

Figure 1: An overview of our program repair model (MultiFix). The decoder outputs for maintaining the current token are omitted.

Edit-distance and optimal alignment. The edit-distance between two strings x and y is the smallest number of atomic operations (insertion, deletion or replacement) that transform x to y. Given a set of input symbols Σ, let

$$\Omega = \{(a \to b) \mid a, b \in \Sigma \cup \{\lambda\}\} \setminus \{(\lambda \to \lambda)\}$$

be a set of edit operations. Namely, Ω is a set of all edit operations for *deletions* $(a \to \lambda)$, *insertions* $(\lambda \to a)$ and *replacement* $(a \to b)$.

Let h be the morphism from Ω^* into $\Sigma^* \times \Sigma^*$ defined by setting $h((a_1 \to b_1) \cdots (a_n \to b_n)) = (a_1 \cdots a_n, b_1 \cdots b_n)$.

We say that $\omega \in \Omega^*$ is an *alignment* of strings $x, y \in \Sigma^*$ if $h(\omega) = (x, y)$. For example, edit sequence $\omega = (a \to \lambda)(b \to b)(\lambda \to c)(c \to c)$ over Ω is an alignment between abc and bcc, $h(\omega) = (abc, bcc)$. We associate a non-negative edit cost $c(\omega)$ to each edit operation $\omega \in \Omega$, where c is a function $\Omega \to \mathbb{R}_+$. We can extend the func-

tion to give the cost of an alignment $\omega = \omega_1 \cdots \omega_n$ in the natural way:

$$c(\omega) = \sum_{i=1}^{n} c(\omega_i).$$

The edit-distance $d(x, y)$ of two strings x and y is the minimal cost of an alignment ω between x and y:

$$d(x, y) = \min\{c(\omega) \mid h(\omega) = (x, y)\}.$$

We say that ω is *optimal* if $d(x, y) = c(\omega)$.

Position embedding. The name of position (positional) embedding has been used in papers (Gehring et al., 2017; Vaswani et al., 2017) introducing non-recurrent sequence-to-sequence models. In order to maintain a sense of order between input symbols without recurrent architecture, they produced additional embedding that contains position information and added the embedding to the input embedding. For example, the Transformer (Vaswani et al., 2017) used sinusoidal functions for the position embedding to enable model to generalize well to longer sequences that are not encountered during training.

When it comes to the problem of program repair, the position embedding becomes a necessary tool for successfully training RNN-based network with relatively longer sequences of program tokens compared to natural language sentences. The most similar approach to our method is by Gupta et al. (2017) that utilizes the idea of using additional embedding about line numbers of codes for generating fixes for the codes. The recent work by Yasunaga and Liang (2020) also employs the position embedding to encode the line offset from the erroneous line reported by compiler.

3 Our Approach

We employ the following techniques to improve the previous learning-based program repair.

Encoding optimal alignment as target. The previous approaches to the program repair based on the sequence-to-sequence learning framework aim to learn pairs of erroneous programs and corresponding line fixes. However, it is very difficult to learn to produce multiple lines of fixes within this framework as it is almost impossible to learn the alignment between the input tokens and the output tokens. In order to resolve this problem, we encode the target sequence as an optimal alignment of the input program to the target program.

Synchronized position embedding. We employ the position embedding into the input vectors to decoder. It assists them to correctly locate the most relevant token in the input token sequence. We further control the position information from the decoder to actively align the position information with the encoder. If the decoder predicts the current output token that corresponds to an insertion edit-operation, then the current position number should not be decremented as the potential subsequent edits must be performed at the input token with the same position number. We call it the synchronized position embedding. Our experimental results show that the synchronized position embedding is extremely helpful for our model to predict the accurate sentence compared to baseline models without synchronized position embedding.

Overall architecture. Figure 1 illustrates our proposing model architecture. Our model has encoder that takes a (erroneous) program p along with the position indices of the tokens in p. Then, the decoder starts to decode outputs, which are an alignment between the input tokens and the potential output tokens. While decoding, the decoder utilizes the same position embedding weights that are used in the encoder. Finally, we apply the predicted alignment to the input program tokens to generate the output program.

4 Experimental Setup

In this section, we summarize the setup of our experiments including the datasets used for evaluation of our model and several training details.

4.1 Datasets

The DeepFix dataset (Gupta et al., 2017) contains C programs submitted by programming novice students in an introductory programming course. The dataset contains 37,415 correct programs (compiled without error) and 6,971 erroneous programs.

For fair comparison to the prior works on learning-based program repair including DeepFix, RLAssist, SampleFix and DrRepair, we also use the DeepFix dataset for training and testing our approach. It is well-known that C/C++ compiler ignores whitespace such as spaces, tabs and new lines with an exception of text literals. This implies that we do not need to maintain the whitespaces in

Model	FRR	RMR
DeepFix (2017)	33.4	40.8
SampleFix (2019)	40.9	56.3
DS-SampleFix (2019)	44.4	61.0
DrRepair (2020)	34.0	-
MultiFix (TF)	31.2	41.4
MultiFix (TF) + DrPerturb	44.7	56.6
MultiFix	37.5	42.8
MultiFix + DrPerturb	**55.6**	**62.5**
DeepFix + Beam Search	44.7	63.9
DS-SampleFix + Beam Search	45.2	65.2
DrRepair + Compiler + Pretrain	68.2	-
MultiFix + DrPerturb + BS	**74.6**	**84.5**

Table 1: Results for performance comparison of DeepFix, SampleFix, DrRepair and MultiFix. Note that the authors of DrRepair did not report RMR in their paper.

Input	Output	Acc.	RMR
Code	Code	73.5	36.7
Code + Pos	Code	75.4	37.8
Code + Pos	Code + Pos	75.5	36.8
Code + Pos	Code + SyncPos	80.6	38.7
Code	Align	83.6	39.0
Code + Pos	Align	83.9	40.9
Code + Pos	Align + Pos	83.7	40.2
Code + Pos	Align + SyncPos	**88.1**	**41.6**

Table 2: Results for performance comparison of different input and output embedding methods. Note that all models here are trained with a subset of DeepFix dataset for typographic errors.

the programs if our approach does not rely on line-level information from the input programs. Therefore, we discard the line numbers from the input sequence as our approach does not utilize the line number information for both localizing the error and producing the line fix.

We also utilize another publicly available dataset generated by the program perturbation procedure for a fair comparison to DrRepair (Yasunaga and Liang, 2020) as the experimental results presented in the DrRepair paper are obtained by training with the above dataset. The DrPerturb is one of the main contributions from the DrRepair (Yasunaga and Liang, 2020) paper as it covers a diverse set of program errors while taking the actual error distribution from the DeepFix dataset into account. The DrPerturb training dataset is constructed based on the original DeepFix dataset by creating roughly 50 corrupted versions by applying the DrPerturb.

4.2 Training Details

Hyperparameters. We set the dimension of input token embedding and position embedding to be 32. We use 4 layers with 256 hidden units of bidirectional LSTM unless explicitly mentioned otherwise. And transformer uses 4 multi-head attention, and the model layer is 4.

Beam search. We use beam search decoding for further improving the performance of our model. The beam width is chosen to be 100 for fair comparison with SampleFix and DS-SampleFix as they also draw 100 candidate fixes.

Evaluation metrics. We use the three metrics for evaluating the effectiveness of our model. First, the *full repair rate (FPR)* is a ratio between the number of completely fixed programs and the total number of programs. The *resolved messaged rate (RMR)* is a ratio between the number of resolved error messages by the model and the total number of error messages. Finally, we also measure the *accuracy* of the model which is a ratio between the number of correctly predicted target sequences by the model and the total number of target sequences.

4.3 Baselines

We compare the performance of our model to the following learning-based program repair models: DeepFix (Gupta et al., 2017), SampleFix, DS-SampleFix (Bhattacharyya et al., 2018), and DrRepair (Yasunaga and Liang, 2020). We also implement the Transformer (Vaswani et al., 2017) (TF) version of MultiFix with the idea of synchronized embedding implemented in a similar fashion.

Note that direct comparison of our model and SampleFix may not be fair since it draws 100 candidate fixes at each iteration. For fair comparison, we compare our model with beam width 100 to SampleFix with beam width 5 as SampleFix uses 20 random variables to generate in total 100 fixes.

5 Results and Analysis

5.1 Performance comparison

Table 1 present the experimental results. Overall, MultiFix achieves the best performance compared to all state-of-the-art program repair models. Especially, MultiFix achieves the best performance when we employ the beam search decoding with

Erroneous Code	Ground Truth	DeepFix	MultiFix

`int main() {`
`  int i`
`  j; scanf("%d",&j;`
`  char ar[j];`
`  for(i=0;i<j;i++)`
`  ar[i]=getchar();`
`  return 0;`
`}`
	`int main() {`
	`  int i,j;`
	`  scanf("%d",&j);`
	`  char ar[j];`
	`  for(i=0;i<j;i++)`
	`  ar[i]=getchar();`
	`  return 0;`
	`}`
		`int main() {`
		`  int i , j ;`
		`  j scanf("%d",&j ) ;`
		`  char ar[j];`
		`  for(i=0;i<j;i++)`
		`  ar[i]=getchar();`
		`  return 0;`
		`}`
			`int main() {`
			`  int i ,j;`
			`  scanf("%d",&j ) ;`
			`  char ar[j];`
			`  for(i=0;i<j;i++)`
			`  ar[i]=getchar();`
			`  return 0;`
			`}`

Table 3: An example of erroneous code and fixes generated by DeepFix and MultiFix. MultiFix achieves two repairing results in one iteration, and DeepFix achieves repairing results in three iterations.

beam size 100 which sufficiently improves the previous state-of-the-art results (+6.4% higher than DrRepair in full setting). It should be noted that the performance of the Transformer version of MultiFix is poorer than LSTM version. We suspect that the Transformers are not suitable for modeling formal languages as already pointed out in recent works (Hahn, 2020; Bhattamishra et al., 2020).

5.2 Synchronized position embedding

In Table 2, we observe that the synchronized position embedding is very helpful in learning the alignment between the input and (potential) output token sequences. Especially, the use of standard position embedding in decoding even deteriorates the performance of our model.

5.3 Case Analysis

Table 3 provides an example where MultiFix succeeds in program repair while the other approaches fail. MultiFix can fix multiple errors at once while DeepFix cannot fix the code even through multiple iterations. As expected, MultiFix can fix multiple errors in many program lines by learning to repair the whole token sequence while DeepFix only learns to replace a suspicious line.

6 Conclusions

Our experimental results concerned with the use of beam search decoding and SampleFix (Hajipour et al., 2019) suggest that we can improve the performance of learning-based program repair approaches by adopting diverse sampling techniques such as CVAE as in SampleFix.

Moreover, the performance difference between two versions of MultiFix model trained with the original DeepFix dataset and the DrPerturb dataset suggests that both the quality and diversity of the training dataset are crucial in learning to repair various types of program errors.

Acknowledgements

This research was supported by the NRF grant (NRF-2020R1A4A3079947), IITP grant (No. 2021-0-00354) and the AI Graduate School Program (No. 2020-0-01361) funded by the Korea government (MSIT).

References

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 *Proceedings of the 40th International Conference on Software Engineering*, pages 78–87.

Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. 2015. Neural machine translation by jointly learning to align and translate. In *Proceedings of the 3rd International Conference on Learning Representations*.

A. Bhattacharyya, B. Schiele, and M. Fritz. 2018. Accurate and diverse sampling of sequences based on a "best of many" sample objective. In *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 8485–8493.

Satwik Bhattamishra, Kabir Ahuja, and Navin Goyal. 2020. On the ability and limitations of transformers to recognize formal languages. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing, EMNLP 2020, Online, November 16-20, 2020*, pages 7096–7116. Association for Computational Linguistics.

Kyunghyun Cho, Bart van Merrienboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. 2014. Learning phrase representations using rnn encoder–decoder for statistical machine translation. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing*, pages 1724–1734.

Jonas Gehring, Michael Auli, David Grangier, Denis Yarats, and Yann N. Dauphin. 2017. Convolutional sequence to sequence learning. In *Proceedings of the 34th International Conference on Machine Learning*, volume 70 of *Proceedings of Machine Learning Research*, pages 1243–1252.

Rahul Gupta, Aditya Kanade, and Shirish K. Shevade. 2019. Deep reinforcement learning for syntactic error repair in student programs. In *Proceedings of the 33rd AAAI Conference on Artificial Intelligence*, pages 930–937.

Rahul Gupta, Soham Pal, Aditya Kanade, and Shirish K. Shevade. 2017. Deepfix: Fixing common C language errors by deep learning. In *Proceedings of the 31st AAAI Conference on Artificial Intelligence*, pages 1345–1351.

Michael Hahn. 2020. Theoretical limitations of self-attention in neural sequence models. *Trans. Assoc. Comput. Linguistics*, 8:156–171.

Hossein Hajipour, Apratim Bhattacharyya, and Mario Fritz. 2019. Samplefix: Learning to correct programs by sampling diverse fixes. *CoRR*, abs/1906.10502.

Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long short-term memory. *Neural Computation*, 9(8):1735–1780.

Thang Luong, Hieu Pham, and Christopher D. Manning. 2015. Effective approaches to attention-based neural machine translation. In *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing*, pages 1412–1421.

Ali Mesbah, Andrew Rice, Emily Johnston, Nick Glorioso, and Edward Aftandilian. 2019. Deepdelta: Learning to repair compilation errors. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 925—936.

Yewen Pu, Karthik Narasimhan, Armando Solar-Lezama, and Regina Barzilay. 2016. Sk_p: A neural program corrector for moocs. In *Companion Proceedings of the 2016 ACM SIGPLAN International Conference on Systems, Programming, Languages and Applications: Software for Humanity*, pages 39—-40.

Ilya Sutskever, Oriol Vinyals, and Quoc V. Le. 2014. Sequence to sequence learning with neural networks. In *Proceedings of the Annual Conference on Neural Information Processing Systems 2014*, pages 3104–3112.

Daniel Tarlow, Subhodeep Moitra, Andrew Rice, Zimin Chen, Pierre-Antoine Manzagol, Charles Sutton, and Edward Aftandilian. 2020. Learning to fix build errors with graph2diff neural networks. In *Proceedings of the 42nd International Conference on Software Engineering*, pages 19–20.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. In *Proceedings of the Annual Conference on Neural Information Processing Systems 2017*, pages 6000–6010.

Petar Velickovic, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Liò, and Yoshua Bengio. 2018. Graph attention networks. In *Proceedings of the 6th International Conference on Learning Representations*.

Michihiro Yasunaga and Percy Liang. 2020. Graph-based, self-supervised program repair from diagnostic feedback. In *Proceedings of the 37th International Conference on Machine Learning*, volume 119 of *Proceedings of Machine Learning Research*, pages 10799–10808.