COS 484 Final Project Report
Modifying a Graph-based, Self-Supervised Neural Network for Program Repair

Aaron Wu
aaronwu@princeton.edu

Henry Tang
hrtang@princeton.edu

Ruijie Fang
ruijief@princeton.edu

Abstract

We study DrRepair, a neural architecture designed for doing automated program repair for C/C++ programs. DrRepair was originally proposed by (Yasunaga and Liang, 2020), with its main innovation being the usage of a graph attention mechanism on a program feedback graph, enabling it to obtain a high score on test sets such as DeepFix. This report presents, in expository form, a discussion of the architecture of DrRepair, and several experimental studies that involve modifying the DrRepair architecture.

1 Introduction

Whenever writing code, debugging takes up a significant portion of time. Currently, many IDEs and text editors are able to catch syntax or other simple type errors using classical algorithms. However, this still leaves much to be desired. DrRepair aims to improve the detection and automatic repair of programs by using natural language processing (NLP) techniques on the source code, as well as the generated error messages.

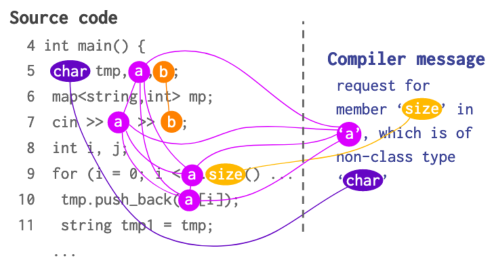
DrRepair is a graph-based neural architecture for automatically fixing syntax errors of C programs using semi-supervised learning. Given a faulty program as input, DrRepair constructs a *program feedback graph* $G = (V, E)$ with

- V = tokens in the diagnostic arguments, i.e. source code and compiler message. Example tokens in Figure 1 include `char` and `size`.

- Every edge in E links two vertices that share the same token

The model then employs a Seq2Seq model with *graph attention mechanism* and LSTM-based encoder/decoders to solve the repair task.

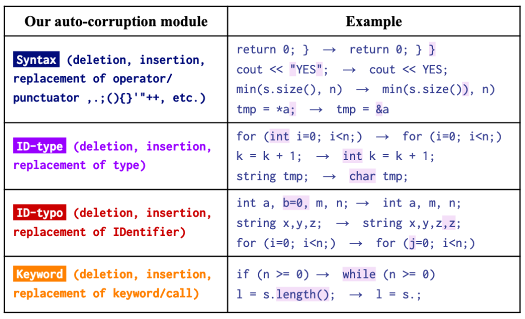
The DrRepair model focuses on fixing four types of compile-time errors, shown in Figure 2.



```
int main() {
 char tmp, a, b;
 map<string,int> mp;
 cin >> a >> b;
 int i, j;
 for (i = 0; i < a.size() ...
  tmp.push_back(a[i]);
  string tmp1 = tmp;
 ...
```

```
request for
member 'size' in
'a', which is of
non-class type
'char'
```

Figure 1: Illustration of program feedback graph.



Our auto-corruption module	Example

Syntax (deletion, insertion, replacement of operator/ punctuator ,.;(){}'"++, etc.)	`return 0; }` → `return 0; } }`
	`cout << "YES";` → `cout << YES;`
	`min(s.size(), n)` → `min(s.size()), n)`
	`tmp = *a;` → `tmp = &a`

ID-type (deletion, insertion, replacement of type)	`for (int i=0; i<n;)` → `for (i=0; i<n;)`
	`k = k + 1;` → `int k = k + 1;`
	`string tmp;` → `char tmp;`

ID-typo (deletion, insertion, replacement of IDentifier)	`int a, b=0, m, n;` → `int a, m, n;`
	`string x,y,z;` → `string x,y,z,z;`
	`for (i=0; i<n;)` → `for (j=0; i<n;)`

Keyword (deletion, insertion, replacement of keyword/call)	`if (n >= 0)` → `while (n >= 0)`
	`l = s.length();` → `l = s.;`

Figure 2: The four categories of error fixed by DrRepair.

2 Related Work

Program repair using non-learning-based methods. Most work along this line focuses on fixing errors in program semantics, and has been done by researchers in the field of programming languages, and many feature heavy use of formal methods. The representative works usually involve the use of an SMT solver (Rothenberg and Grumberg, 2016; DeMarco et al., 2014), a program synthesizer (Kulal et al., 2019) or other constraint solvers to solve program constraints, the solution of which can be represented as a repaired program (Rothenberg and Grumberg, 2016; DeMarco et al., 2014; Könighofer and Bloem, 2013; Le Goues et al., 2012a,b). Although the resulting repair procedure takes exponential time, this line of work has an interesting characteristic: Many repair procedure are precise, i.e. once the procedure emits an output, the output represents the correct program. For instance, in the recent work of (Rothenberg and Grumberg, 2016), the proposed procedure is *sound* and *complete*, and returns the set of all minimal repaired programs.

Program repair using learning-based methods. Motivated by the success and popularity of deep nets and Seq2Seq models in NLP, there has been a recent flurry of work that uses a learning-based approach to tackle program repair. The work predating DrRepair mostly use sequence models (Gupta et al., 2017; Gupta et al.; Parihar et al., 2017; Ahmed et al., 2008). Contrary to the program repair procedures using non learning-based methods, these models require a training stage and a large dataset, run fast (less than exponential time in terms of the size of input) during inference, and the repaired output is not guaranteed to be correct. In addition, most existing literature (including DrRepair) in learning-based program repair focus on syntax repair, whereas the non-learning based methods we cite above focus on semantics repair.

3 Data

The main data used in this study is the DeepFix dataset, which comprises of 44,386 C programs written by students (Gupta et al., 2017). Of these 44,386 programs, 6,971 do not compile correctly. The 6,971 erroneous programs were set aside as the test set.

To generate the training/development set, an automated corruption procedure named DrPerturb was used on the remaining 37,451 correct programs (Yasunaga and Liang, 2020). DrPerturb contains four corruption modules that purposefully induce the types of errors DrRepair focuses on fixing (see Figure 2). To create a corrupted version of correct code, a number 1-5 is chosen uniformly at random, which determines how many corruptions to apply. Then, that number of corruptions sampled randomly from the modules (with replacement) are applied to the code in question, and the resulting compiler error message is saved. 2000 examples are set aside as the development set, which leaves 35,451 examples for training.

In addition to this "synthetic" training/development dataset, an extra pretraining dataset was generated. 310K C++ programs, which compile correctly and are of length less than 100 lines, from codeforces.com were collected. For each of these programs, 50 corrupted versions were created using DrPerturb, yielding around 1.5 million training examples. Even though C++ is not the same language as C, the original paper's empirical results suggest that pretraining with this data set improves model performance on the C based DeepFix data.

4 Model Architecture

The DrRepair model can be conceptualized as three parts. The first consists of an initial encoding, graph attention, and recontextualization layers. The other two parts come from handling the localization and editing tasks. A simple feedfoward to softmax combination is used for the former. The latter uses a Pointer-Generator Decoder network. On a high level, the model should be viewed as a Seq2Seq type neural network. The RNNs in the first part correspond to the "encoding" sequence and the decoding network in the third part corresponds to the "decoding" sequence. Figure 3 provides an illustration of the architecture.

Initial Encoding: Source code $\mathbf{x}_{1:L}$ and feedback $f = (i, m)$ are the inputs, where i denotes the line number and m denotes the message of error. This input is then encoded using two bidirectional LSTMs: $\mathbf{LSTM}^{(1)}_{\texttt{code}}$ and $\mathbf{LSTM}^{(1)}_{\texttt{msg}}$. A positional encoding is used to associate the error line number i_{err} with the current line number i. This positional encoding is concatenated together with the outputs $\mathbf{LSTM}^{(1)}_{\texttt{code}}$ at each sequence step. All together, this layer outputs $\mathbf{h}_{x_{ij}}$ and error message $\mathbf{h}_{m_i}$. Note that for the "code-only" ablation, $\mathbf{LSTM}^{(1)}_{\texttt{msg}}$ is omitted.

At this step the model has learned short-ranged dependencies within the source code and (separately) within the compiler message. To learn relevant longer range dependencies i.e. those across lines and between the source code and error message, the $\mathbf{h}_{x_{ij}}$, $\mathbf{h}_{m_i}$ are then passed to the graph attention layer.

Graph attention layer: A 2-layer graph attention network is used on the program feedback graph G. Let $\mathbf{h}^{n-1}, \mathbf{h}^{n}$ denote the input/outputs of the n-th layer. The graph attention layer operates on feature vector representations of the nodes in the graph $G = (V, E)$. Each node is assigned a feature vector. Initially the feature vectors takes in the encoded inputs from previous layers, $\mathbf{h}_{x_{ij}}$ or $\mathbf{h}_{m_l}$, as

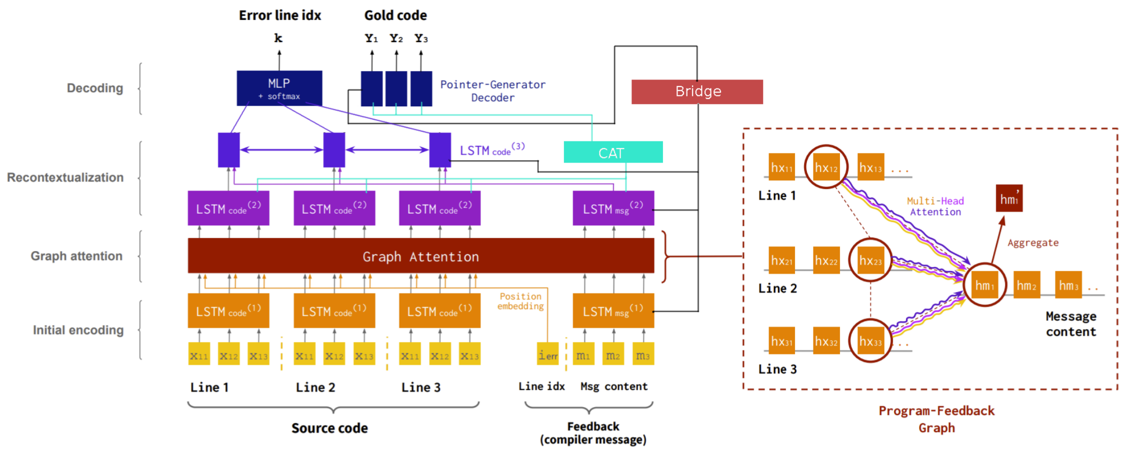


Figure 3: Illustration of model architecture. The dotted box on the right hand side shows the graph attention layer in detail.

$\mathbf{h}^0$, the starting feature vector.

First, the attention coefficients between nodes u, $v \in V$ at layer n of the graph attention network are computed.

$$\mathbf{c}^n = \mathbf{Attention}_G(\mathbf{h}^{n-1})$$

In a most general setting, attention would be computed across all pairs $(u, v) \in V \times V$. However here, we are only interested in incident nodes $(u, v) \in E \subset V \times V$; implementation wise, this is accomplished by the application of a mask that contains the appropriate adjacency information. The attention mechanism used is the same as the multi-headed attention used in transformer layers. The computed attention coefficients are then concatenated with the output of the previous layer and passed through a feddfoward network to obtain the output.

$$\mathbf{h}^n = \mathbf{MLP}([\mathbf{h}^{n-1}; \mathbf{c}^n])$$

Note that in the "no-graph" ablation, this layer is omitted and the outputs of the first set of LSTMs are passed immediately to the recontextualization layer.

Recontextualization layer: The graph attention network allows the model to learn long range dependencies, but we would also like to model how these long range relations impact dependencies on a local level i.e. recontextualization. To do this, the token representations $\mathbf{g}$ are passed to two additional LSTMs denoted by $\mathbf{LSTM}^{(2)}_{\texttt{code}}$ and $\mathbf{LSTM}^{(2)}_{\texttt{msg}}$. An embedding for each line i is acquired by concatenating the hidden states of the LSTMs:

$$\mathbf{r}_i = [\mathbf{LSTM}^{(2)}_{\text{code}}(\mathbf{g}_{x_i}); \mathbf{LSTM}^{(2)}_{\text{msg}}(\mathbf{g}_m)]$$

The final line embedding is given by

$$\mathbf{s}_{1:L} = \mathbf{LSTM}^{(3)}_{\text{code}}(\mathbf{r}_{1:L})$$

As before, note that for the "code only" ablation, $\mathbf{LSTM}^{(2)}_{\texttt{msg}}$ is omitted.

Localization task: For this task, the final line embedding is passed to a single layer feedforward neural network, followed by a softmax. This gives a probability distribution over the line numbers $k = 1, 2, ..., L$, where $p(k)$ indicates the probability that line k contains an error.

$$p(k|\mathbf{s}_{1:L}) = \text{Softmax}(\mathbf{MLP}(\mathbf{s}_{1:L}))$$

Editing task: The editing task is handled by a Pointer-Generator decoder network, which is an augmentation of the vanilla Seq2Seq decoder model with attention. The key innovation of this network is the use of a learned "soft switch" p_{gen}, which allows the decoder to choose between using words from the input or generating a word from the vocabulary. More specifically, the final output distribution at each decoding step is a weighted combination of the attention distribution and the vocabulary distribution, where p_{gen}, $(1 - p_{\text{gen}})$ are the respective weights. For an in-depth discussion of how this type of network works, see (See et al., 2017).

As applied to this problem, the input is the code from the file to be repaired and the vocabulary is simply the set of all source code tokens present in the data set. The encoded sequence used is $\mathbf{r}_k$. To obtain the initial hidden of the LSTM decoder, the concatenation of all three LSTM layers are used (arrows indicate forward and backward directions):

$$\Big[\overleftarrow{\mathbf{LSTM}^{(1)}_{\text{code}}}(\mathbf{x}_{i1}), \overrightarrow{\mathbf{LSTM}^{(1)}_{\text{code}}}(\mathbf{x}_{is}),$$
$$\overleftarrow{\mathbf{LSTM}^{(2)}_{\text{code}}}(\mathbf{g}_{x_{i1}}), \overrightarrow{\mathbf{LSTM}^{(2)}_{\text{code}}}(\mathbf{g}_{x_{is}}),$$
$$\overleftarrow{\mathbf{LSTM}^{(3)}_{\text{code}}}(\mathbf{r}_i), \overrightarrow{\mathbf{LSTM}^{(3)}_{\text{code}}}(\mathbf{r}_i)\Big]$$

and similarly for the initial cell state, where s denotes the sequence length of a given line. These concatenations are passed through "bridge" feed-forward networks, which conceptually project the representations into a lower dimension and in practice, allow the tensor dimensions to match. The decoder produces a distribution over the vocabulary at each decoding step. Beam decoding is then used to generate the corresponding repair, y_k.

$$p(y_k|\mathbf{s}_{1:L}) = \text{PtrGen}(\mathbf{r_k})$$

During training time, the erroneous line number is given directly to the network. During test time, the erroneous line number is the one with the highest probability from the output distribution of the localization task. It should also be noted that teacher forcing is used in training the decoder.

Training procedure: During training, we use the standard negative log-likelihood loss: $-\log p(k, y_k|x, f)$. The hyperparameters used in training the model are listed in table 1. For all modifications explained in the next section, the hyperparameters were untouched, except for the Base+pretrain+finetune transformer variant. This was changed to 5, as GPU memory was not sufficient when training with the original batch size of 20.

All reproduction and modification experiments on DrRepair were done on the Princeton Adroit cluster with 8 Intel Skylake CPU cores and an NVidia Tesla V100 GPU.

5 Modifications

5.1 Modification 1: Changing LSTMs to GRUs

Our first modification involves changing the LSTM layers to use GRUs instead, including the LSTM in the Pointer-Generator decoder. GRUs do not have a hidden cell state, and thus have fewer trainable parameters. As a result, GRUs are able to run faster during both training and test time. Furthermore, GRUs have empirically been found to provide similar performance to LSTMs (Chung et al., 2014).

One complication involving GRUs is that the model uses the hidden cell states from the LSTM as input into the pointer decoder generator, which is then used as input into the bridge layer and the decoder. To remedy this, we change a number of parameter sizes throughout the model so they do not depend on the dimensions of hidden cell states.

5.2 Modification 2: Changing LSTMs to Transformers

Our second modification involves changing the LSTM encoder layers into transformer encoder layers. Since transformers use the self-attention mechanism instead of storing all context information within a single hidden state, they are theoretically more adept at capturing dependencies (Vaswani et al., 2017). Thus, this modification should enable the model to better grasp dependencies within each line, while the graph attention network still takes care of the longer range line to line and source code-compiler error message dependencies. Furthermore, transformers are more parallelizable and thus train faster on GPUs.

The complications present with transformers are significantly more complex than the previous modification. First, transformers do not have hidden states in the same manner as LSTMs and GRUs. To maintain the same model structure, we replaced LSTM layers 1,2, and 3 with a transformer layer, and extracted the state at the last time step to pass into the CAT and Bridge layers.

Another issue was that the implementation of transformers in Pytorch require the same input and output embedding size. The original LSTM layers 1, 2, and 3, however, had a different hidden size. To remedy this, we passed the output of each transformer encoder layer through feedforward networks so dimensions would match.

For the pointer-generator decoder layer, we retained the use of GRUs. We originally tried changing this to use a transformer decoder layer, however, we found this to be far too complex to handle in our limited amount of time. Here are few of the difficulties encountered:

1. The decoder already uses an attention mechanism; we would need to remove it while still maintaining the rest of the model architecture.
2. During training time, the implementation of the decoder outputs items one at a time, while

Hyperparameter	code only, no graph, base, finetune	pretrain
Batch size	25	20
Num. iterations	150000	400000
Dropout	0.3	0.3
Learning rate	0.0001	0.0001
Gradient clip	10	10
$\mathbf{LSTM}^{(1)}$ hidden size	200	200
$\mathbf{LSTM}^{(2)}$ hidden size	200	200
$\mathbf{LSTM}^{(3)}$ hidden size	200	200
Num. $\mathbf{LSTM}^{(1)}$ layers	3	3
Num. $\mathbf{LSTM}^{(2)}$ layers	1	1
Num. $\mathbf{LSTM}^{(3)}$ layers	2	2
Final MLP dim.	200	200
Code and Message Embedding Dim.	200	200
Position Embedding Dim.	200	200
Activation function	ReLU	ReLU

Table 1: Hyperparameters used during training

a transformer decoder generates the entire sequence at once. Furthermore, the original model concatenates previously generated elements with the GRU outputs at each time step, which significantly complicates the model.

3. Beam decoding is used to generate the actual edited code. However, the implementation of beam decoding is highly non-trivial and takes in a number of different parameters, which all come from the output of the GRUs.

Accommodating for this, as well as other dimension mismatches that appeared, would require editing a dozens of functions over thousands of lines of code. Thus, we decided to stick to using GRUs for the decoder, which made the overall process simpler, at the cost of perhaps a less intuitively understandable model.

5.3 Modification 3: Fine-grained error messages

We develop this modification by observing that the granularity of error messages generated by the C/C++ compiler used for automatic corruption of programs has a noticeable impact on the final model performance: Fine-grained error messages include more helpful tokens in the error string, which in turn help train the graph attention layer from the program feedback graph.

The original error messages associated with each code sample was generated using GCC. In this modification, we augment the corruption procedure by corrupting programs in the Codeforces dataset with error messages from the Clang compiler, which supports generating more fine-grained error messages. We then pre-train original LSTM models on this newly generated data, before fine-tuning the models on the Deepfix dataset.

An extension. As one may argue that data collected by the Codeforces dataset were all written for the same purpose (of solving programming competition problems), they all fit a certain style and are not sampled from the collection of all C programs uniformly at random. Motivated by this question, we also considered introducing new data for the pretraining dataset by introducing new, randomly generated C programs by the CSmith fuzzer. However, we found that it is difficult to make the Csmith-generated programs interoperable with the existing auto-corruption procedure: the randomly generated C programs require significant amount of driver code to correctly compile, and due to simplified assumptions about C/C++ source code present in the existing dataset, the auto-corruption procedure does not deal with the heavy use of C preprocessor macros in a friendly fashion. In addition, although the CSmith-generated code tends to differ in terms of syntax structure, all generated code exhibit very similar use of variable names and code structure (e.g. they feature program prologues and epilogues to set up / clean up resources). Due to all these factors, we decided not to proceed with this modification.

Model	Train Loc.	Train Edit	Dev. Loc.	Dev. Rep.	Test Rep.
Code only LSTM	95.68%	77.68%	95.40%	71.70%	39.9%
No graph LSTM	97.76%	79.52%	97.65%	71.05%	64.7%
Base LSTM	98.08%	81.76%	98.20%	75.35%	71.7%
Base+pretrain+finetune LSTM	99.16%	84.70%	98.65%	78.95%	70.7%
Base+Clang+finetune LSTM	98.69%	86.19%	98.65%	79.70%	71.7%
Code only GRU	95.60%	78.48%	95.95%	70.95%	40.3%
No graph GRU	97.68%	77.84%	97.45%	70.45%	65.5%
Base GRU	99.52%	85.92%	97.10%	74.35%	67.9%
Base+pretrain+finetune GRU	98.92%	80.00%	98.55%	78.10%	70.5%
Code only Transf.	87.20%	70.64%	41.25%	9.80%	13.1%
No Graph Transf.	96.00%	71.68%	90.15%	41.40%	46.6%
Base Transf.	96.16%	76.64%	90.15%	41.40%	42.8%
Base+pretrain+finetune Transf.	97.60%	63.60%	94.80%	49.80%	51.7%

Table 2: Results for LSTM, GRU, Transformer, and Clang ablations. "Loc." refers to identification of the first line that must be repaired, "Dev." refers to development, and "Rep." refers to the successful repair of the line identified by "Loc".

6 Results

Table 2 contains the results of our tests. We provide separate results for the training, development, and test sets. Localize refers to the accuracy at identifying the *first* line that needs to be repaired, edit refers to the successful edit of that first line, and repair refers to the accuracy at actually successfully editing all erroneous lines and generating a working program. At test time, the model had a compile budget of 10. That is, it was allowed ten iterative unsuccessful repair and compile attempts before being given the next test example. Note that the model identifies and fixes one line at a time.

Figures 4 and 5 display training and development results for localize accuracy from the Base models. To see graphs for all metrics and all models, please refer to the Appendix.

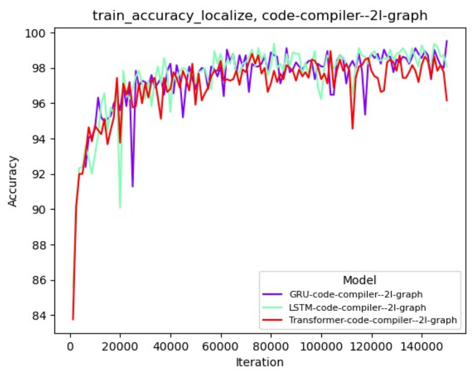


Figure 4: Train accuracy for identifying error line

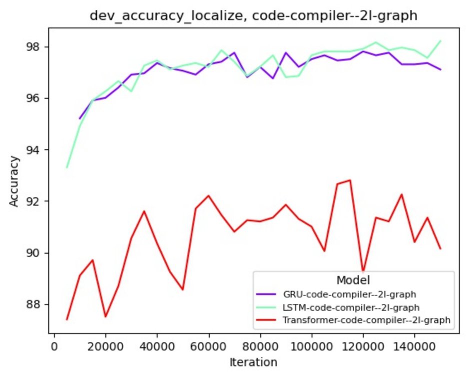


Figure 5: Dev accuracy for identifying error line

7 Analysis

We first note that the Base LSTM model actually does better than the Base+pretrain+finetune LSTM at testing time. This disagrees with the original paper's result, which describes the Base+pretrain+finetune variant as the most successful. There are several possible explanations for this discrepancy. The training of the Base ablation might have simply settled into a better local minimum than the original. The difference between the Base and Base+pretrain+finetune model performance in the original paper was only 1.8%, so it is entirely possible that this is the case. This difference could also be attributed to suboptimal hyperparameter choices.

7.1 Modification 1

GRU models performed similarly compared to LSTM models across all metrics. Most corresponding results across the various metrics in Table 2 between the LSTM and GRU models differ by at most 2%. Even the largest discrepancy, namely Train Edit Accuracy, differs by less than 5%. This is expected, since GRUs have been empirically found to perform similarly compared to LSTMs. During training, GRUs provided on average 1.8x speedup. GRUs have $\frac{3}{4}$ as many parameters as LSTMs. However, they provided a speedup larger than $\frac{4}{3}$ because parts of the original model relied on the hidden and cell states of LSTMs. Using GRUs cell states, which cuts parameters in these layers of the network by half.

7.2 Modification 2

Like GRUs, transformers managed to provide a 1.8x speedup on average during training. Since we trained on GPUs, we were able to take advantage of the parallelizable nature of transformers to achieve faster training compared to LSTMs, which requires passing hidden states one at a time.

On the other hand, we found that transformer models we trained exhibited poor generalization performance on the test/dev sets. We provide a few possible explanations for this behavior.

1. **Compounded Errors**: Localize accuracies tended to be significantly closer to the original variations than the repair accuracies for both test and development sets. On the test and development set, unlike on the training set, we run the model 10 times and check if it generates a working program at the end. Thus, the lower edit accuracy of the transformer model on the training set is compounded when evaluating on the development and test sets, leading to significantly worse performance.

2. **Overfitting**: Transformers performed comparatively better during training as opposed to on the development and test sets, with the accuracy often at most 10% lower than LSTM counterparts. Thus, the first three transformer modifications could have overfitted. In contrast, there is a significant improvement in the development and test accuracies when pretraining is performed and additional data is introduced, which further supports this idea.

3. **Unoptimized Hyperparameters**: When changing the original model to use transformers, we did not change any but one of the original hyperparameters, which include the learning rate, number of transformer layers, and dropout thresholds. Thus, this could have led to suboptimal learning for the transformer.

   The one hyperparameter that we did change was the batch size for the pretrain+finetune transformer model. As noted before, this change was necessary due to insufficient GPU memory. Since we did not increase the number of iterations, this model variant saw less examples in the training set than the others. This explains the lower training accuracy compared to other transformer variants. That being said, the pretrain+finetune version of the transformer modification performs the best out of all the other transformer variants on the test and development set.

4. **Model Irregularities**: For the transformer modifications, we wanted to maintain as much of the original model architecture as possible. However, this also meant we had to make a few design choices that would rarely be implemented in practice. For instance, we extracted the last embedding from each transformer encoder layer as input into the bridge layer. This makes sense for LSTMs, as the last hidden state conceptually captures all previous information, but it is not as meaningful for transformers since the attention mechanism means each embedding is treated similarly.

   This is further evidenced by how PyTorch Transformer encoders do not allow you to get the outputs of these intermediate layers; we had to iterate through each layer in the encoder manually. In addition, the decoder still utilizes GRU as described in Section 5.2. However, passing in outputs from transformers as initial states for GRUs is a strange choice, and thus means our model's architecture is not as optimal as it could be.

7.3 Modification 3

The model pretrained on the dataset with Clang error messages outperformed or matched all the original ablations. Development localization accuracy was the same as the original pretrained model's result, 98.65% accuracy, and improved upon the repair accuracy by 0.75%. In terms of test repair accuracy, the Clang ablation achieved a 1% improvement as compared to the model using the original pretraining dataset.

While we noted earlier in the analysis that pretraining on the original extra Codeforces data yields worse model performance, pretraining on the Clang data set does not exhibit the same impact on model behavior. We believe that the more fine-grained nature of Clang error messages is a reason for this phenomenon. Refer back to Section 5.3 for a more in-depth discussion.

8 Conclusion and Future Work

In this paper, we have documented our reproduction of Yasunaga and Liang's "Graph-based, self-supervised program repair from diagnostic feedback" (Yasunaga and Liang, 2020) and our exploration of three modifications to the original work. We focus on the program repair portion. Our three modifications were: changing LSTMs to GRUs, changing encoder LSTMs to transformers, and changing the original dataset to utilize error messages generated by Clang instead of GCC. Out of these three modifications, we found that the first and third produced similar performance to the original LSTM model, while using transformers performed worse.

Future work could involve changing the decoder to use transformer decoder layers, optimizing hyperparameters, and tweaking CSmith to output quality data that is compatible with the existing training framework.

9 Credits

This is a final project for COS484, Spring 2021 at Princeton University. We thank Professor Danqi Chen, Professor Karthik Narasimhan, and Zexuan Zhong for their support. Figures 1-3 are courtesy of (Yasunaga and Liang, 2020); figure 3 was modified by authors. We thank Princeton Research Computing for allowing generous use of their Adroit cluster.

References

U. Z. Ahmed, P. Kumar, A. Karkare, P. Kar, and S. Gulwani. 2008. Compilation error repair: for the student programs,from the student programs. In *ICSE*.

Junyoung Chung, Çaglar Gülçehre, KyungHyun Cho, and Yoshua Bengio. 2014. Empirical evaluation of gated recurrent neural networks on sequence modeling. *CoRR*, abs/1412.3555.

F. DeMarco, J. Xuan, D. Le Berre, and M. Monperrus. 2014. Automatic repair of buggy if-conditions and missing preconditions with smt. In *Proceedings of the 6th Inter-national Workshop on Constraints in Software Testing, Verification, and Analysis*, pages 30–39. ACM.

R. Gupta, A. Kanade, , and S. Shevade. Deep reinforcement learning for programming language correction. In *AAAI 2019b*.

R. Gupta, S. Pal, A. Kanade, and S. Shevade. 2017. Deepfix: Fixing common c language errors by deep learning. In *AAAI*.

R. Könighofer and R. Bloem. 2013. Repair with on-the-fly program analysis. In *Biere, A., Nahir, A., Vos, T. (eds.) HVC 2012. LNCS vol. 7857.*, pages 56–71. Springer, Heidelberg.

S. Kulal, P. Pasupat, K. Chandra, M. Lee, O. Padon, A. Aiken, and P. Liang. 2019. Spoc: Search-based pseudocodeto code. In *NeurIPS*.

C. Le Goues, M. Dewey-Vogt, S. Forrest, and W. Weimer. 2012a. A systematic study ofautomated program repair: fixing 55 out of 105 bugs for 8 each. In *34th Interna-tional Conference on Software Engineering (ICSE)*, pages 3–13. IEEE.

C. Le Goues, T. Nguyen, S. Forrest, and W. Weimer. 2012b. Genprog: a generic method forautomatic software repair. *IEEE Transactions on Software Engineering*, pages 54–72.

S. Parihar, Z. Dadachanji, R. D. Praveen Kumar Singh, A. Karkare, and A. Bhattacharya. 2017. Automatic grading and feedback using program repair for introductory programming courses. In *ACM Conference on Innovationand Technology in Computer Science Education*. ACM.

Bat-Chen Rothenberg and Orna Grumberg. 2016. Sound and complete mutation-based program repair. In *Proceedings of FM 2016: Formal Methods*, pages 593–611.

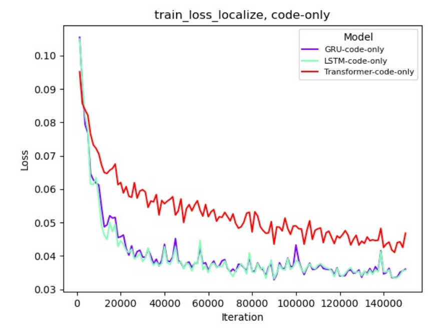
Abigail See, Peter J. Liu, and Christopher D. Manning. 2017. Get to the point: Summarization with pointer-generator networks. *CoRR*, abs/1704.04368.

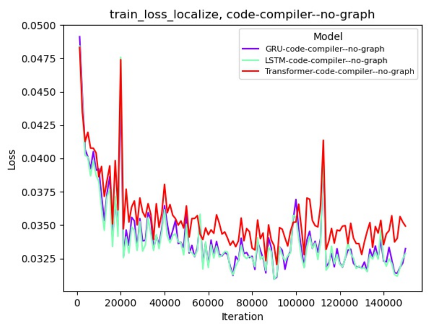
Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. *CoRR*, abs/1706.03762.

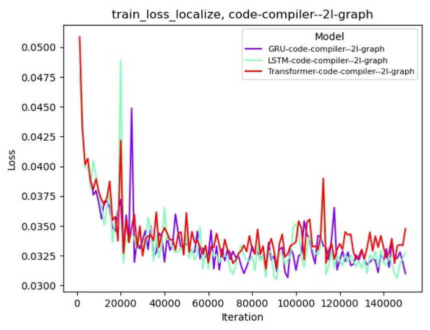
Michihiro Yasunaga and Percy Liang. 2020. Graph-based, self-supervised program repair from diagnostic feedback. *CoRR*, abs/2005.10636.

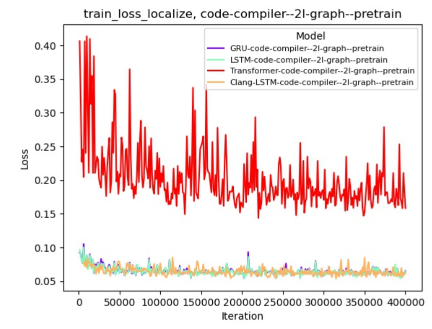
10 Appendix

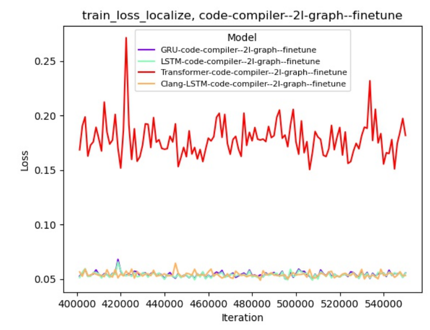
10.1 Train Task Loss



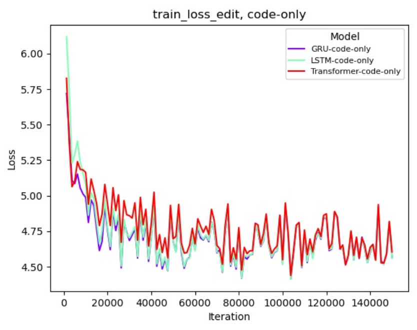


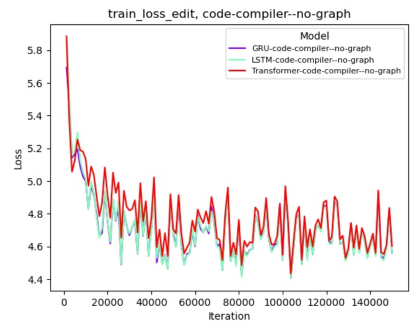


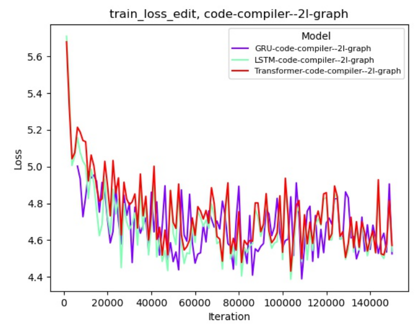


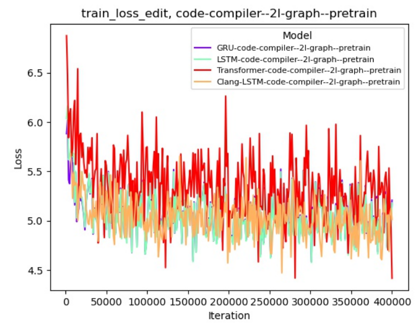


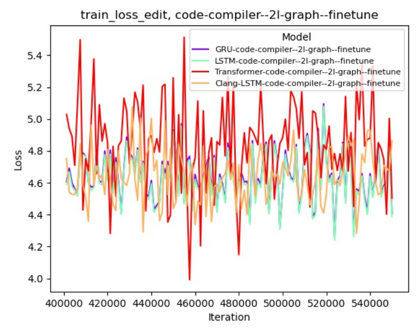
10.2 Train Edit Loss



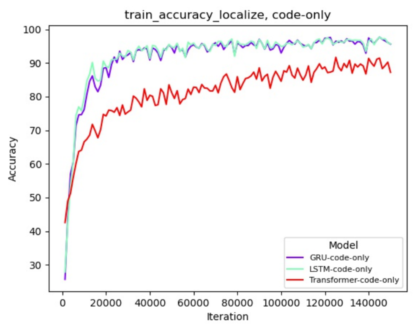


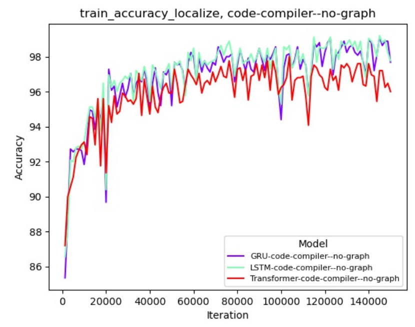


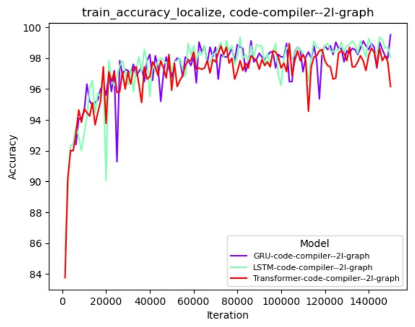


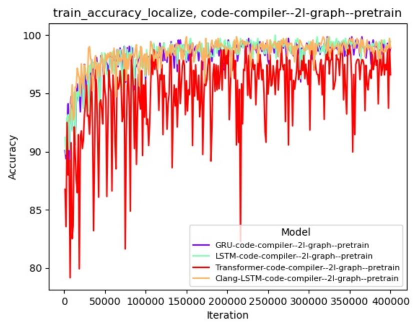


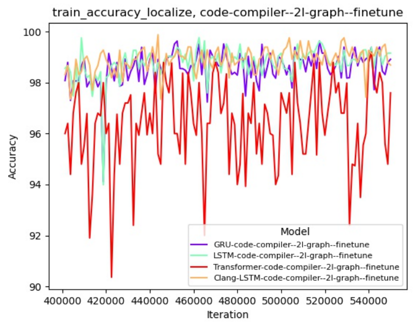
10.3 Train Localize Accuracy Loss



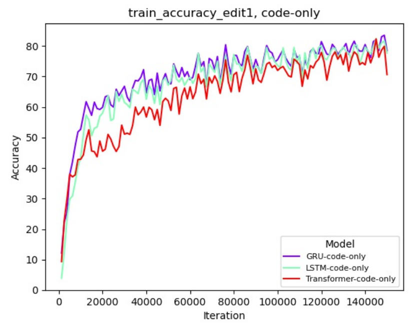


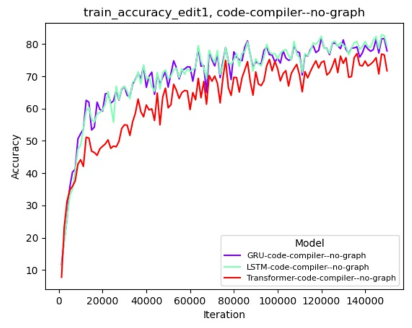


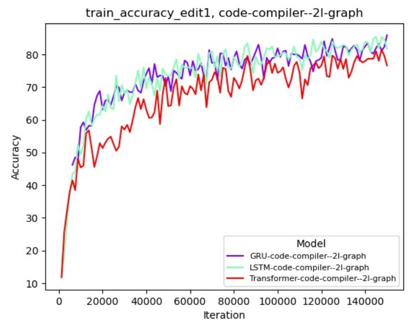


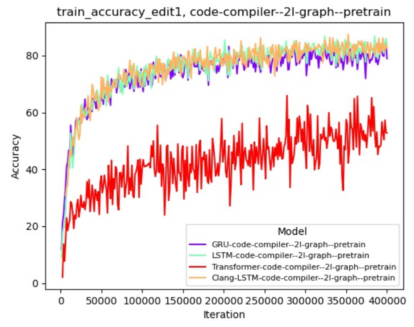


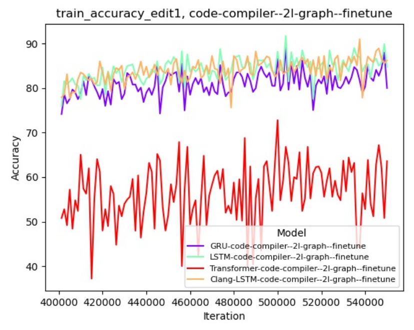
10.4 Training Edit Accuracy using Compiler Identified Error Line



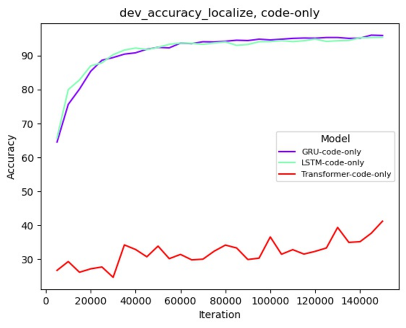


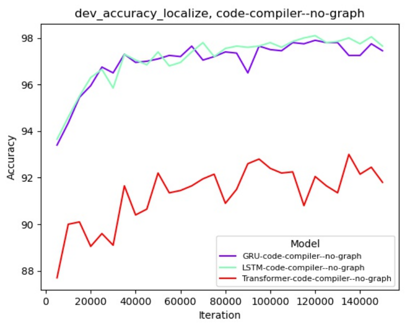


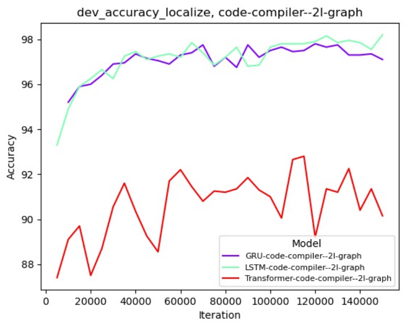


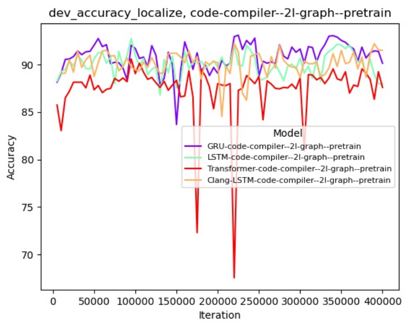


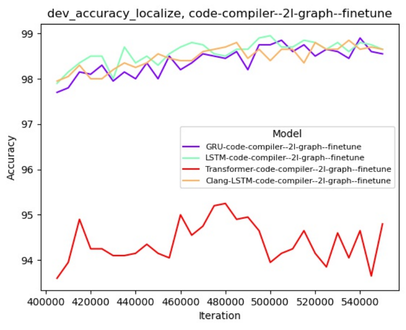
10.5 Development Localize Accuracy



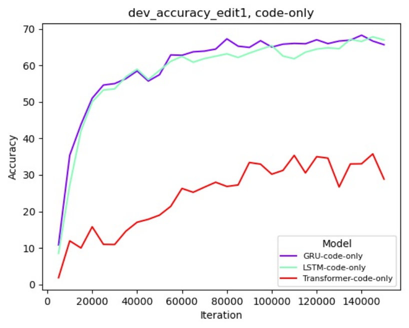


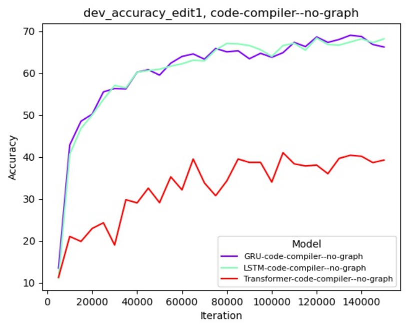


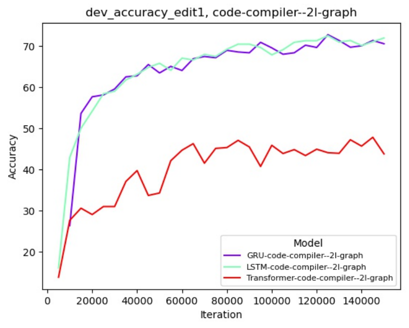


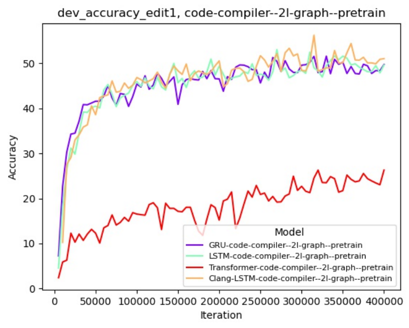


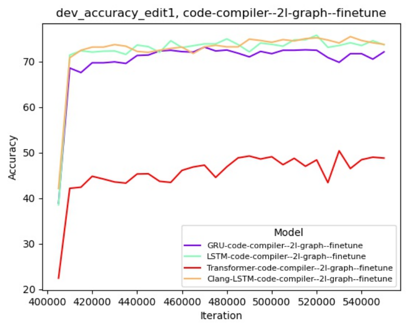
10.6 Development Edit Accuracy using Compiler Identified Error Line



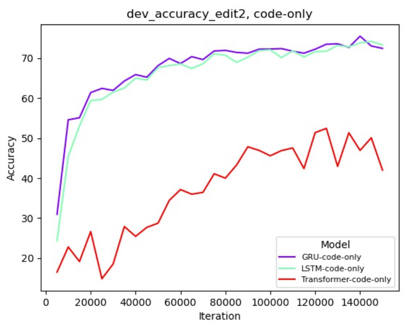


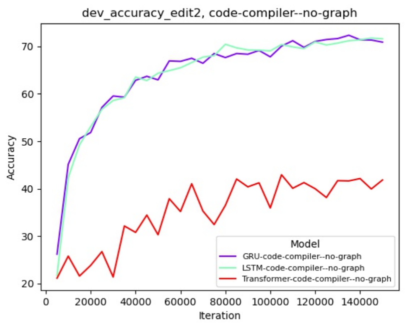


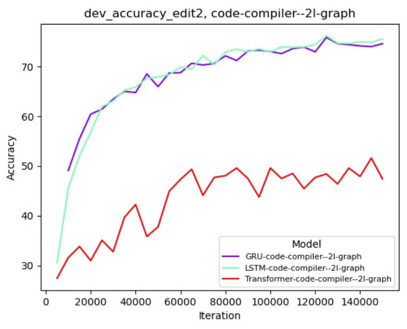


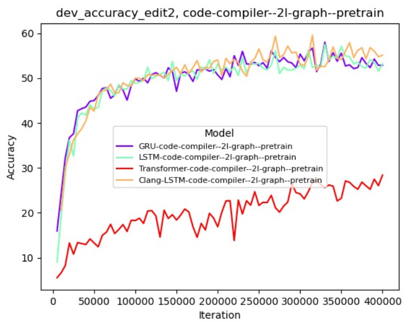


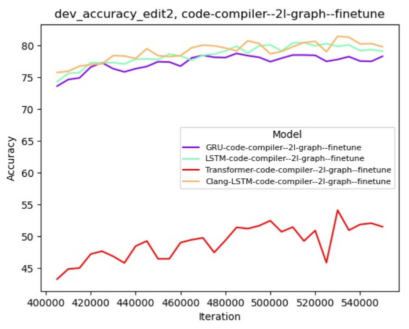
10.7 Development Edit Accuracy using DrRepair Generated Error Line



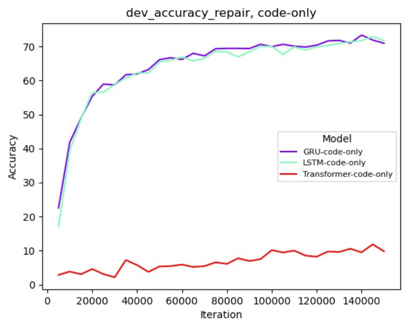


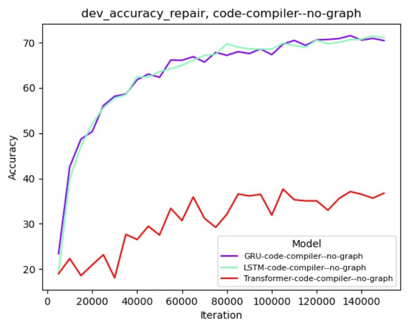


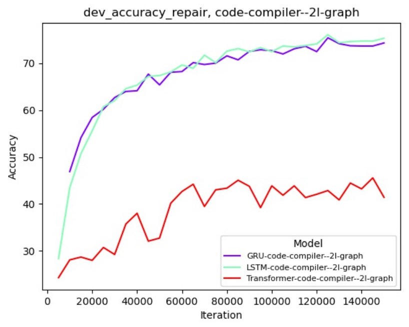


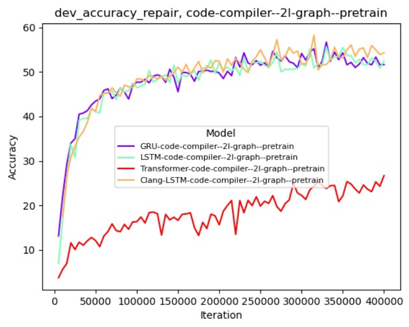


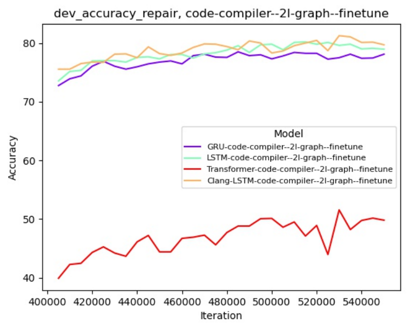
10.8 Development Repair Accuracy


dev_accuracy_repair, code-only
Accuracy
Iteration
Model
GRU-code-only
LSTM-code-only
Transformer-code-only


dev_accuracy_repair, code-compiler--no-graph
Accuracy
Iteration
Model
GRU-code-compiler--no-graph
LSTM-code-compiler--no-graph
Transformer-code-compiler--no-graph


dev_accuracy_repair, code-compiler--2l-graph
Accuracy
Iteration
Model
GRU-code-compiler--2l-graph
LSTM-code-compiler--2l-graph
Transformer-code-compiler--2l-graph


dev_accuracy_repair, code-compiler--2l-graph--pretrain
Accuracy
Iteration
Model
GRU-code-compiler--2l-graph--pretrain
LSTM-code-compiler--2l-graph--pretrain
Transformer-code-compiler--2l-graph--pretrain
Clang-LSTM-code-compiler--2l-graph--pretrain


dev_accuracy_repair, code-compiler--2l-graph--finetune
Accuracy
Iteration
Model
GRU-code-compiler--2l-graph--finetune
LSTM-code-compiler--2l-graph--finetune
Transformer-code-compiler--2l-graph--finetune
Clang-LSTM-code-compiler--2l-graph--finetune