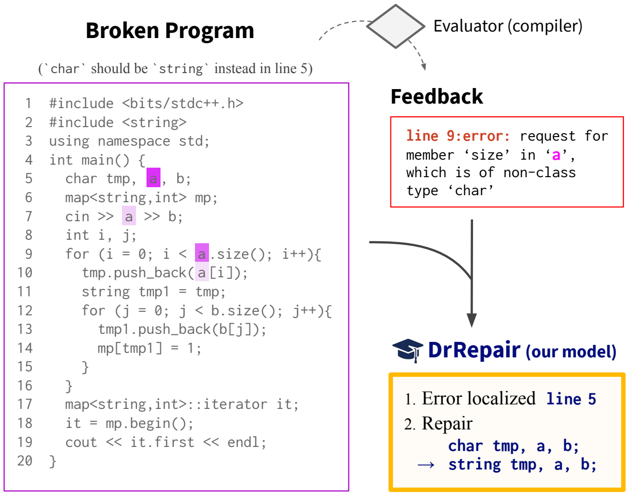
Graph-based, Self-Supervised Program Repair from Diagnostic Feedback

Michihiro Yasunaga [1] **Percy Liang** [1]

Abstract

We consider the problem of learning to repair programs from diagnostic feedback (e.g., compiler error messages). Program repair is challenging for two reasons: First, it requires reasoning and tracking symbols across source code and diagnostic feedback. Second, labeled datasets available for program repair are relatively small. In this work, we propose novel solutions to these two challenges. First, we introduce a program-feedback graph, which connects symbols relevant to program repair in source code and diagnostic feedback, and then apply a graph neural network on top to model the reasoning process. Second, we present a self-supervised learning paradigm for program repair that leverages unlabeled programs available online to create a large amount of extra program repair examples, which we use to pre-train our models. We evaluate our proposed approach on two applications: correcting introductory programming assignments (DeepFix dataset) and correcting the outputs of program synthesis (SPoC dataset). Our final system, DrRepair, significantly outperforms prior work, achieving 68.2% full repair rate on DeepFix (+22.9% over the prior best), and 48.4% synthesis success rate on SPoC (+3.7% over the prior best).



(`char` should be `string` instead in line 5)

```
1  #include <bits/stdc++.h>
2  #include <string>
3  using namespace std;
4  int main() {
5    char tmp, a, b;
6    map<string,int> mp;
7    cin >> a >> b;
8    int i, j;
9    for (i = 0; i < a.size(); i++){
10     tmp.push_back(a[i]);
11     string tmp1 = tmp;
12     for (j = 0; j < b.size(); j++){
13       tmp1.push_back(b[j]);
14       mp[tmp1] = 1;
15     }
16   }
17   map<string,int>::iterator it;
18   it = mp.begin();
19   cout << it.first << endl;
20 }
```

```
line 9:error: request for member ‘size’ in ‘a’, which is of non-class type ‘char’
```

1. Error localized `line 5`
2. Repair

```
  char tmp, a, b;
→ string tmp, a, b;
```

Example taken from SPoC dataset (`909A-45398788.cpp`)

Figure 1. Given a broken program and diagnostic feedback (compiler error message), our goal is to localize an erroneous line and generate a repaired line.

1. Introduction

Automatic program repair has the potential to dramatically improve the productivity of programming. In particular, a common source of program errors are compiler errors, which include use of unresolved symbols, missing delimiters (e.g. braces), and type errors. These errors are commonly observed in both beginner programmers (Parihar et al., 2017) and professional developers (Seo et al., 2014), as well as in the predicted code of program synthesis (Kulal et al., 2019). Accordingly, the use of machine learning in fixing compiler errors has garnered significant interest recently (Gupta et al., 2017; Hajipour et al., 2019; Mesbah et al., 2019).

In this work, we consider the problem of learning to repair programs based on diagnostic feedback (compiler error messages). Figure 1 illustrates the setup. Given a broken program and diagnostic feedback, we aim to localize an erroneous line in the program and generate a repaired line. Learning program repair has two major challenges: First, the system needs to connect and *jointly* reason over the broken source code and the diagnostic feedback (Fitzgerald et al., 2008). Second, existing works rely on manual effort to curate labeled datasets for program repair (e.g. ⟨broken program, fixed program⟩ pairs), which does not scale up (Mesbah et al., 2019). Here we present *DrRepair*, a novel approach to program repair that addresses these two challenges. Our key innovations are two-fold: 1) modeling of program repair with *program-feedback graphs* and 2) self-supervised learning with unlabeled programs.

Program-feedback graph. Program repair requires reasoning jointly over the symbols (e.g. identifiers, types, operators) across source code and diagnostic feedback. For instance, in the example given in Fig. 1, while the compiler message points to line 9, the error is related to the type of identi-

[1] Stanford University, Stanford, CA. Correspondence to: Michihiro Yasunaga <myasu@cs.stanford.edu>.

Proceedings of the 37^{th} International Conference on Machine Learning, Online, PMLR 119, 2020. Copyright 2020 by the author(s).

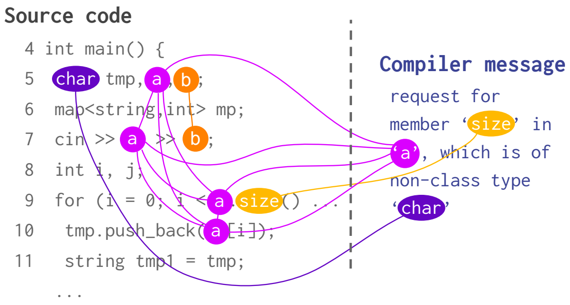
fier ‘a’, and one needs to track how ‘a’ has been used or declared earlier to resolve this error. To formalize this reasoning process, we propose a joint graph representation of a program and diagnostic feedback that captures the underlying semantic structure of symbols in the context of program repair (program-feedback graph). Specifically, it takes all identifiers (e.g. a, b) in the source code and any symbols in the diagnostic arguments (e.g. ‘a’, ‘char’) as nodes, and connects instances of the same symbols with edges to encode the semantic correspondence (Fig. 2). We then design a neural net model with a graph-attention mechanism (Veličković et al., 2018) on the program-feedback graph to model the symbol tracking process described above. While prior works in program repair purely apply sequence-to-sequence (seq2seq) models to programs (Gupta et al., 2017; Hajipour et al., 2019) or rely on the program’s Abstract Syntax Tree (AST) representations (Mesbah et al., 2019; Tarlow et al., 2019), our program-feedback graph directly connects symbols involved in the reasoning process of program repair, and allows efficient information flow across them.

Self-supervised learning. Motivated by the vast amount of program data available online (e.g. GitHub has 28 million public repositories), we propose a self-supervised learning paradigm for program repair that leverages unlabeled programs to create a large amount of extra training data. Specifically, we collect working programs from online resources related to our problem domain (programming contests in our case), and design a procedure that corrupts a working program into a broken one, thereby generating new examples of ⟨broken program, fixed program⟩. In our experiments, we prepare extra data ~10 times the size of original datasets in this way, use it to pre-train our models, and fine-tune on the target task. We also describe an effective corruption procedure that covers a diverse set of errors. While prior works in program repair rely on labeled datasets (Mesbah et al., 2019; Tarlow et al., 2019; Kulal et al., 2019), we are the first to present a self-supervised learning method for program repair that leverages unlabeled programs online.

We evaluate the efficacy of our proposed approach on two applications, using publicly available datasets:

a) Correcting introductory programming assignments. We use DeepFix dataset (Gupta et al., 2017), where the task is to repair broken C programs submitted by students.
b) Correcting the output code in program synthesis. We use the SPoC dataset (Kulal et al., 2019), where the task is to translate pseudocode into C++ implementation, and programs synthesized by prior models (seq2seq) often fail to compile. We apply our repair model to correct the candidate programs generated in this task.

Experimental results show that our approach (DrRepair) outperforms prior work significantly, achieving 68.2% full repair on the DeepFix test set (+22.9% absolute over the prior best), and 48.4% synthesis success rate on the SPoC test set (+3.7% absolute over the prior best at the time of this work). Additionally, our analysis shows that the use of a program-feedback graph is particularly helpful for fixing errors that require reasoning over multiple lines of code, and that self-supervised pre-training facilitates the learning of program repair for the types of errors with fewer training examples in the original dataset.



```
 4 int main() {
 5  char tmp, a, b;
 6  map<string,int> mp;
 7  cin >> a >> b;
 8  int i, j;
 9  for (i = 0; i < a.size() ...
10   tmp.push_back(a[i]);
11   string tmp1 = tmp;
    ...
```

request for member ‘size’ in ‘a’, which is of non-class type ‘char’

Figure 2. Illustration of **program-feedback graph**, corresponding to the example in Fig. 1. The graph captures long-range dependencies of symbols to help model the reasoning process of program repair.

2. Problem statement

Figure 1 illustrates the program repair task. The system is given (a) a broken program with L lines, $x = (x_1, ..., x_L)$, and (b) diagnostic feedback provided by a compiler, $f = (i_{\text{err}}, m_{\text{err}})$, where i_{err} denotes the reported line number, and m_{err} the error message (a sequence of tokens). If the compiler returns multiple error messages, we use only the first one.[1] Our task is to identify the index of an erroneous line $k \in \{1, \ldots, L\}$ (*error localization*), and generate a repaired version of the line y_k (*repair*). Let $y = y_{1:L}$ denote the fixed version of the full program ($y_i = x_i$ for $i \neq k$). In the example given in Figure 1, $x_5 =$ “char tmp, a, b;”, $i_{\text{err}} = 9$, $m_{\text{err}} =$ “request for … type ‘char’ ”, and $k = 5$, $y_k =$ “string tmp, a, b;”. Note that the line number reported by a compiler (i_{err}) does not necessarily match the line we need to repair (k).

3. Approach

We approach program repair from two angles. First, we propose a *program-feedback graph* to model the reasoning process involved in program repair. Second, we introduce a self-supervised learning paradigm that leverages unlabeled programs to create a large amount of extra training data.

3.1. Modeling

To model program repair, we start off with a sequence-to-sequence learning setup, and incorporate the information of

[1] Note that here we are defining a module that repairs a single line of code in a program. We describe how we apply this repair module to programs with multiple errors in §4. We also explain the application-dependent evaluation metrics in §4.

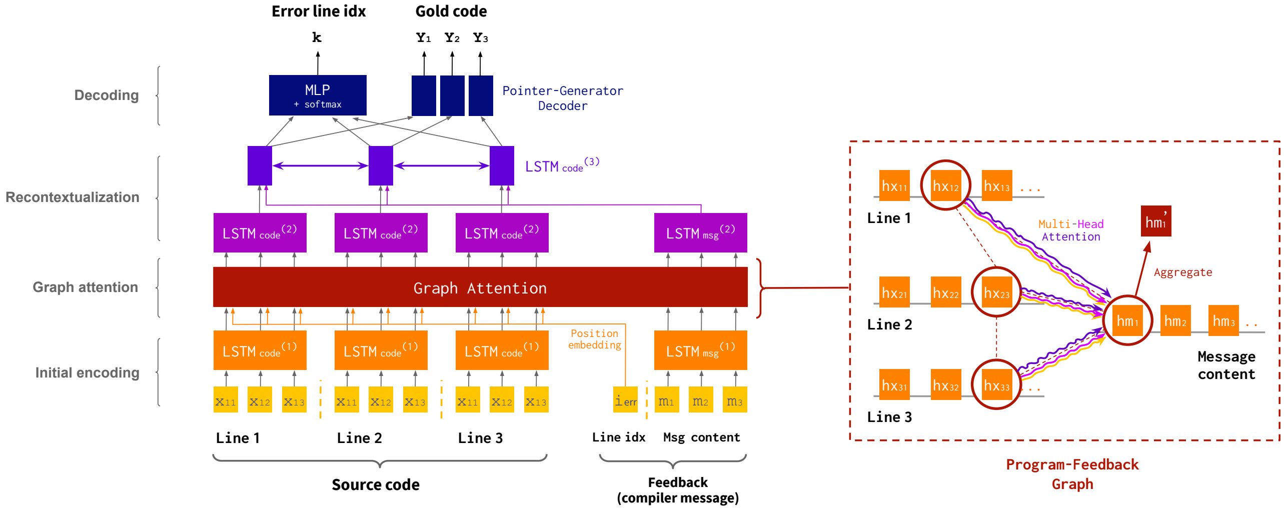


Figure 3. DrRepair model. It takes in a program $x = (x_1, ..., x_L)$ and diagnostic feedback from a compiler $f = (i_{\text{err}}, m_{\text{err}})$ as inputs (**bottom**), encodes them via LSTM and graph attention layers, and decodes the error line index k and repaired code y_k (**top**). The right-hand side illustrates the graph attention mechanism. Best viewed in color.

a program-feedback graph through a graph attention model, which we describe below. Given an input program $x_{1:L}$ and its feedback $f = (i_{\text{err}}, m_{\text{err}})$, we first tokenize each line x_i and the compiler message m_{err} into a sequence of symbols: $x_i = (x_{i1}, x_{i2}, ...)$ and $m_{\text{err}} = (m_1, m_2, ...)$. As seen in our motivating example in Fig. 1, program repair requires reasoning and tracking symbols across different lines of code and compiler messages (e.g., given the compiler error about '`a`', a programmer will jump to the source code line reported by the message, and then track how '`a`' has been used/declared in earlier lines). These long-range dependencies of tokens are difficult to capture using previous seq2seq or AST-based models, which only propagate information locally at the line or syntax level (Gupta et al., 2017; Mesbah et al., 2019). To enable more efficient information flow, we introduce a program-feedback graph G that *directly* connects tokens relevant to the reasoning of program repair.

3.1.1. Program-feedback graph

A program-feedback graph $G = (V, E)$ has nodes V that consist of tokens in the diagnostic arguments (those within ' ' in the message, i.e., `size`, `a`, `char` in Fig. 2), their occurrences in the source code, and all remaining identifiers in the code (e.g. `a`, `b`, `i`, `j`). The type of each token, such as identifier (for `a`), operator (for `=`) and data type (for `char`), is recognized by the C/C++ tokenizer in Gupta et al. (2017).

We then form the graph by connecting identical tokens in V with undirected edges (E) to capture the semantic correspondence. The resulting graph is as a set of cliques, one for each symbol (e.g. '`a`'). We keep the program-feedback graph simple for two reasons: 1) we use the graph and graph-attention to specifically capture the (long-range) connections of tokens crucial to program repair reasoning, and perform other local information propagation via LSTMs (we elaborate in §3.1.2), and 2) it is nontrivial to analyze the code further (e.g. parsing) to add information to the graph, as the program can be syntactically ill-formed. Compared to AST-based graph representations (Allamanis et al., 2018; Tarlow et al., 2019), our program-feedback graph is more relaxed and robust to errors in source code.

3.1.2. Model architecture

Fig. 3 illustrates our program repair model. It has an encoder that takes in a program x and feedback f, and a decoder that predicts a distribution over which line is erroneous k and a repaired line y_k. The encoder has three stages: 1) initial encoding $\mathbf{h} = \text{InitEnc}(x, f)$ which encodes each input token at the line level, 2) graph attention $\mathbf{g} = \text{GraphAttn}(\mathbf{h})$ which propagates information across tokens on a program-feedback graph, and 3) recontextualization $\mathbf{s} = \text{ReContext}(\mathbf{g})$ which contextualizes token representations at the line level again to produce an embedding $\mathbf{s}_i$ for each line i. Finally, Decode($\mathbf{s}$) outputs a distribution over the erroneous line index and a repaired line (k, y_k). We describe each of the model stages in detail below.

Initial encoding. Given source code $x_{1:L}$ and feedback $f = (i_{\text{err}}, m_{\text{err}})$ (Fig. 3 bottom), we encode each line x_i and compiler message m_{err} with two bidirectional LSTM networks (Hochreiter & Schmidhuber, 1997), $\text{LSTM}^{(1)}_{\text{code}}$ and $\text{LSTM}^{(1)}_{\text{msg}}$. For the tokens in the source code, we also inject the information of the reported line index (i_{err}) by concatenating the outputs of $\text{LSTM}^{(1)}_{\text{code}}$ with the positional encoding (Vaswani et al., 2017) of the line offset $\Delta i = i_{\text{err}} - i$, and applying a feedforward network. We denote the representation of each token in the code and message at this point as $\mathbf{h}_{x_{ij}}$ and $\mathbf{h}_{m_\ell}$, respectively. This stage is analogous to the input encoding in Kulal et al. (2019).

Error type	Common compiler messages	Statistics				Relevant auto-corruption module (our proposal)
		DeepDelta	DeepFix	SPoC	Avg.	
Expected ... • operator/punctuator • primary expression	expected @@@ (e.g. expected ‘;’ before..., expected ‘}’ at end of input, expected primary-expression before...) missing @@@ (e.g. terminating " character)	9%	48% • 37% • 11%	35% • 29% • 6%	30% • 23% • 7%	**Syntax** (deletion, insertion, replacement of op/punc) **ID-typo** (deletion, insertion of IDentifier)
Identifier type/declaration conflict	redeclaration/conflicting declaration @@@ invalid conversion from <type> to <type> no match for ‘operator @@@’ (operand types are @@@)	9%	5%	18%	11%	**ID-type** (deletion, insertion, replacement of type)
Identifier undeclared	@@@ was not declared	62%	33%	31%	42%	**ID-typo** (deletion, replacement of IDentifier)
Others	‘else’ without a previous ‘if’ no matching function for call to...	20%	14%	16%	17%	**Keyword** (deletion, insertion, replacement) Above modules (e.g. **Syntax**, **ID-type**, **ID-typo**) can also cause errors clustered here

Table 1. Analysis of common compiler errors in three settings: experienced developers (DeepDelta), beginner programmers (DeepFix), and predicted code of program synthesis (SPoC). For DeepDelta, the statistics is taken from Mesbah et al. (2019). The rightmost column shows the program perturbation modules that we design to generate corresponding types of errors.

Graph attention. Next, to model the reasoning (symbol tracking) process in program repair, we use a graph attention network (Veličković et al., 2018) to allow information to flow across symbols in the program-feedback graph G (Fig. 3 right). In a N-layer graph attention network, each layer computes contextualized representations of tokens via

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

$$\mathbf{h}^n = \text{MLP}([\mathbf{h}^{n-1}; \mathbf{c}^n]) \qquad (2)$$

where $\mathbf{h}^{n-1}$, $\mathbf{h}^n$ denote the input/output representation of each token at the n-th layer. Initially, $\mathbf{h}^0$ is $\mathbf{h}_{x_{ij}}$ or $\mathbf{h}_{m_\ell}$, and the final output $\mathbf{g} = \mathbf{h}^N$. $\text{Attention}_G(\mathbf{h}_t)$ computes attention weights over the neighboring nodes of a token t on the graph G, $\mathcal{N}_G(t)$, and takes the weighted average of the token representations among $\mathcal{N}_G(t)$. MLP is a feedforward network. For a more detailed description about graph attention, we refer readers to Veličković et al. (2018).

Recontextualization. We allow the information updated via the graph to propagate on the local context again, by passing the token representations $\mathbf{g}$ to additional sequence networks, $\text{LSTM}^{(2)}_{\text{code}}$ and $\text{LSTM}^{(2)}_{\text{msg}}$. We obtain an embedding of each line i by concatenating their final hidden states,

$$\mathbf{r}_i = \left[\text{LSTM}^{(2)}_{\text{code}}(\mathbf{g}_{x_{i\cdot}})^{\text{final}};\ \text{LSTM}^{(2)}_{\text{msg}}(\mathbf{g}_{m.})^{\text{final}}\right] \qquad (3)$$

which is further contextualized to be the final line embedding $\mathbf{s}_i$, via $\mathbf{s}_{1:L} = \text{LSTM}^{(3)}_{\text{code}}(\mathbf{r}_{1:L})$ (Fig. 3 top).

Decoding. Given the line embeddings $\mathbf{s}_{1:L}$, we model the probability of a line $k \in \{1, \dots, L\}$ being erroneous via a feedforward network, and model its repair y_k, via a pointer-generator decoder (See et al., 2017):

$$p(k \mid \mathbf{s}_{1:L}) = \text{softmax}(\text{MLP}(\mathbf{s}_{1:L})) \qquad (4)$$

$$p(y_k \mid \mathbf{s}_{1:L}) = \text{PtrGen}(\mathbf{s}_k). \qquad (5)$$

Training. A training example consists of a broken program x, feedback f, an erroneous line index k, and the repaired line y_k. The loss on a given example is the standard negative log-likelihood, $-\log p(k, y_k \mid x, f)$. The error localization and repair components are learned jointly. In §3.2 and §4.1, we will discuss how we generate training examples of this form for pre-training and target applications.

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.;

Table 2. Proposed program perturbation modules for generating self-supervised data.

3.2. Self-supervised learning

Labeled datasets for program repair ($\langle x, y \rangle$ pairs) are limited in size (10–100K data points) (Mesbah et al., 2019), but there is a vast amount of unlabeled programs available online: for instance, GitHub[2] alone has 28 million public repositories as of 2019. Can we leverage this freely-available code to improve the learning of program repair?

With this motivation, we propose a new self-supervised learning paradigm that utilizes unlabeled, working programs to create a large amount of training data for program repair. Specifically, we first collect a large set of working programs y's (ones that compile, in our setting), related to the domain of interest. We design a randomized procedure $\mathcal{P}$ that automatically corrupts y into a broken program x to generate a new training example $\langle$broken code x, ground-truth $y\rangle$. We repeatedly apply this procedure to the collected programs, and use the generated training data to perform pre-training (Erhan et al., 2010) of our model, facilitating it to learn

[2] https://github.com/

useful representations for program repair (*self-supervised pre-training*). Later, we fine-tune the model on a labeled, original (in-domain) dataset.

Below, we describe the details of our program corruption and data generation process.

Program corruption procedure. To design an effective corruption procedure that covers a diverse set of program errors, we first analyzed common compiler errors in three settings: experienced developers, beginner programmers, and predicted code of program synthesis. For each case, we collected statistics from Mesbah et al. (2019), DeepFix dataset (Gupta et al., 2017) and SPoC dataset (Kulal et al., 2019), and grouped the errors into four major categories: "Expected...", "Type/declaration conflict", "Identifier undeclared", and "Others" (details in Table 1).

Motivated by this analysis, we design a set of perturbation modules (heuristics), $\mathcal{M}$, that aim to modify source code to cause the above types of errors. Specifically, $\mathcal{M}$ consists of

- **Syntax**, which randomly deletes, inserts or replaces an operator/punctuation, such as `,.;(){}[]"++<<`. This module causes various errors such as "`expected @@@`".
- **ID-type**, which randomly deletes, inserts or replaces an identifier (ID) type such as `int`, `float`, `char`. This causes errors such as conflicting types and redeclaration.
- **ID-typo**, which randomly deletes, inserts or replaces an identifier. This module causes errors such as missing primary expressions and undeclared identifiers.
- **Keyword**, which randomly deletes, inserts or replaces a use of program language keyword or library function, such as `if` and `size()`. This module can cause other miscellaneous errors.

Table 2 provides concrete examples of each module. Note that each module makes a single change to source code at a time. Given the perturbation modules $\mathcal{M}$, our program corruption procedure (named *DrPerturb*) samples 1–5 modules from $\mathcal{M}$ (with replacement) and applies them to an input program sequentially. We sample each module with probability 0.3, 0.5, 0.15, 0.05, respectively, motivated by the distribution of errors found in our analysis (Table 1).

We will show in our experiments that DrPerturb is significantly more effective than baseline corruption procedures such as randomly deleting tokens.

Data preparation details. As the program domain in our applications (DeepFix, SPoC) is C/C++ implementation of introductory algorithms, we turn to programs available on `codeforces.com`, which contains C++ code submitted by programming contest participants. We collect accepted programs and filter out outliers (e.g. those longer than 100 lines), following the procedure in Kulal et al. (2019). This yields 310K C++ programs that compile successfully, which is roughly 10 times the size of the original training data available in our applications (37,415 programs in DeepFix, 14,784 in SPoC). For each program, we then create roughly 50 corrupted versions by applying DrPerturb and keeping ones that fail to compile. This yields $\sim$1.5M extra training examples of $\langle$broken code x, feedback f, correct code $y\rangle$, which we use to pre-train our program repair model.

Note that the collected program data share the same source with SPoC (`codeforces.com`),[3] but not exactly with DeepFix, which is C programming assignments. Nevertheless, we find that the collected data is highly effective in both tasks, which we elaborate on in §4.

4. Experiments

We conduct an extensive evaluation of our approach via two applications: DeepFix[4] (Gupta et al., 2017) and SPoC[5] (Kulal et al., 2019), which are recent benchmarks for program repair and program synthesis, respectively.

4.1. Experimental setup

We summarize the setup of DeepFix and SPoC, and describe how we apply our program repair model to those tasks.

4.1.1. DeepFix

Task. The DeepFix dataset contains C programs submitted by students in an introductory programming course, of which 37,415 are correct (compile) and 6,971 are broken (do not compile). The average program length is 25 lines. The broken programs are called *raw test set* and may contain multiple errors. The task is to repair them into ones that compile (*full repair*; evaluation metric is full repair rate).

Data processing. To generate training/dev data for repair models, we corrupt the correct programs in DeepFix using DrPerturb. We call this the *synthetic* data, as apposed to the raw test set. We also call this the *original* train/dev data to distinguish it from the *extra* data prepared for pre-training, which is not exactly in the same domain as DeepFix.

How to apply the repair model. At test time, as the broken programs may contain errors in multiple lines, we apply the repair model iteratively until the program compiles or we reach the attempt limit of 5, as in Gupta et al. (2017).

4.1.2. SPoC

Task. The SPoC dataset consists of 18,356 C++ programs (avg. length 15 lines) collected from `codeforces.com`. For each program $t = t_{1:L}$ (with L lines), every line of code is annotated with natural language pseudocode, $s_{1:L}$. The task is to synthesize the target program t from pseudocode

[3] We made sure that the programs collected for pre-training do not overlap with the exact programs in SPoC test sets.

[4] `https://bitbucket.org/iiscseal/deepfix`

[5] `https://sumith1896.github.io/spoc`

Evaluation on DeepFix data

Repair Model	Single Localize (Our synthetic dev)	Single Repair (Our synthetic dev)	Full Repair (DeepFix raw test)
DeepFix (Gupta et al., 2017)	-	-	27.0*
RLAssist (Gupta et al., 2019b)	-	-	26.6*
SampleFix (Hajipour et al., 2019)	-	-	45.3*
Our base (no compiler message)	95.0	70.8	34.0*
Our base	97.1	70.9	62.5
Our base + graph	97.9	74.8	66.4
Our base + graph + pre-train (DrRepair)	**98.9**	**80.2**	**68.2**

Table 3. Performance of our repair model and prior work on **DeepFix** data. We report the single step error localization / repair accuracy (%) on our **synthetic dev** set (column 2-3), and the full repair success rate (%) on DeepFix **raw test** set (column 4). "DrRepair" refers to our full model, which outperforms prior work by significant margins. (*) compiler messages not used.

Ablation on SPoC dev

Repair Model	Single Localize (SPoC dev)	Single Repair (SPoC dev)
Our base	92.0	48.6
Our base + graph	93.1	53.0
Our base + graph + pre-train (DrRepair)	**94.9**	**56.2**
If use pseudocode		
Our base	93.2	65.2
Our base + graph + pre-train (DrRepair)	**96.1**	**68.0**

Table 4. Performance of our repair model on **SPoC** data. We measure the single step error localization / repair accuracy (%) on the SPoC Dev set. "DrRepair" refers to our full model.

Evaluation on SPoC test

Synthesis Method	SPoC TestP		SPoC TestW	
	B=10	B=100	B=10	B=100
Baselines				
Top 1 (no search)	17.8	17.8	30.7	30.7
Best first search	26.5	32.5	42.5	51.0
Prior best (Kulal et al., 2019)	28.4	34.2	44.4	53.7
Our DrRepair	30.2	37.5	46.6	55.9
Our DrRepair w/ pseudocode	**31.4**	**38.5**	**48.0**	**57.0**

Table 5. Program **synthesis success rate** (%) at search budgets B on the **SPoC Test** sets. Our search method equipped with DrRepair consistently outperforms the previous best in all settings.

s within a budget of B attempts (search iterations). Prior work (Kulal et al., 2019) uses a seq2seq translation system to map each pseudocode line s_i into a set of 100 candidate code pieces $\mathcal{C}_i = \{t_{ic_i} \mid c_i \in [100]\}$, where candidate piece t_{ic_i} has probability p_{ic_i}. A full candidate program t is a concatenation of candidate code pieces, and has score $p(t)$:

$$t = \texttt{concat}_{i=1}^{L} t_{ic_i}, \quad p(t) = \textstyle\prod_{i=1}^{L} p_{ic_i}, \tag{6}$$

where c_i is to be searched for each line i. Kulal et al. (2019) then considers various search algorithms (e.g. best first search using the score) to efficiently find the correct program t from this space of candidates.

Why & how to apply the repair model. As Kulal et al. (2019) observed, the top candidates produced by this scoring metric exhibited syntactic or semantic incoherence (e.g. conflicting types) and fail to compile, because each candidate score p_{ic_i} is calculated by line-level translation of pseudocode, ignoring the global context. To address this issue, Kulal et al. (2019) combined best first search with error localization; here we propose a search algorithm that also follows best first search, but attempts to *repair* the current candidate program with our repair model if it does not compile, and adds the repaired code piece $t_{ic'_i}$ into the pool of candidate code pieces $\mathcal{C}_i$, with an updated score $p_{ic'_i}$.

Data processing. We follow the data splits in Kulal et al. (2019), which consists of Train, Dev, TestP, and TestW. We use TestP / TestW for the final evaluation of program synthesis, and use Train / Dev to train or validate our repair model. To prepare train / dev data for the repair model, for each program $y = y_{1:L}$ in SPoC, we sample an error line index k and substitute line y_k with a candidate $c_{kj} \in \mathcal{C}_k$ generated from pseudocode line s_k. We then collect any modified program y' that produces a compiler error f. We call this *original* train/dev data, to distinguish with the *extra* data prepared for pre-training.

4.2. Hyperparameters & training details

We set the dimension of input token embeddings and position embeddings to be 200 and 100. The LSTMs and graph attention networks have a state size of 200. We use 3, 2, 1 and 2 layers for LSTM$^{(1)}$, graph attention net, LSTM$^{(2)}$ and LSTM$^{(3)}$, respectively, with dropout rate 0.3 applied to each layer (Srivastava et al., 2014). The parameters of the models are optimized by Adam (Kingma & Ba, 2015), with batch size 25, learning rate 0.0001, and gradient clipping 1.0 (Pascanu et al., 2012), on a GPU (GTX Titan X).

4.3. Results

We describe our main results on DeepFix and SPoC here. We use "base" to denote the version of our model that replaces graph attention with line-level LSTM layers (a pure sequence model), and "base + graph" the one with graph attention. We train these models on the *original* data (from DeepFix / SPoC) only. "base + graph + pre-train" denotes a "base+graph" model that is pre-trained with self-supervision on the *extra* data and fine-tuned on the *original* data.

DeepFix. Table 3 describes the performance of our re-

Repair Model	Corruption Procedure		
	DrPerturb **(Ours)**	Gupta+17	Random
Our base (no compiler feedback)	**34.0**	24.2	30.1
Our base	**62.5**	50.5	49.4
Our base + graph	**66.4**	54.5	53.0

Table 6. **Effect of different program corruption procedures**. We train repair models using different program corruption methods (our DrPerturb, Gupta et al. (2017)'s, and random token dropout), and evaluate the trained models on the DeepFix raw test set (full repair rate %). **Bold score** indicates the best corruption algorithm.

Repair Model	Full Repair
Our base	62.5
Our base + graph (edges among code only)	64.8
Our base + graph (edges across code-feedback only)	64.9
Our base + graph (final)	**66.4**
Our base + self-attention	66.0

Table 7. **Comparison of different graph architectures**. We evaluate the models on the Deepfix raw test set (full repair rate %).

pair model along with prior work. "Single Repair" column shows the accuracy of repairing a single line (single step) on the *synthetic* dev set, and "Full Repair" column shows the full repair acc. on the *raw* test set, where our repair model is run iteratively (§4.1.1). First, we observe that our base model ("our base" row) achieves 62.5% full repair acc., which outperforms prior works (those above the dashed line) by 15% absolute. We hypothesize that this is because our model uses compiler messages as input, but prior works do not (they consider direct mapping of broken code into its fix). To understand the importance of using compiler messages for program repair, we experimented with a version of our model that does not use compiler messages ("our base, no compiler message"). We find that while it attains comparable scores to "our base" on the *synthetic* dev, the performance drops a lot on the raw test set: 34.0% acc., similar to the prior work (30–40% acc.). This suggests that diagnostic feedback offered by compiler messages plays a crucial role in learning program repair, and without it, the model tends to learn superficial patterns present in the *synthetic* train/dev data (hence the high scores on dev).

Next, we find that our program-feedback graph ("base + graph") provides a 3% boost over "base" in full repair rate, and self-supervised pre-training ("base + graph + pre-train") provides a further improvement of 2%, suggesting that both the program-feedback graph and self-supervision provide complementary improvements. Consequently, with the use of compiler messages, graph and pre-training, our full system DrRepair ("base + graph + pre-train") improves on the prior best (SampleFix) by 22.9% in total, achieving a state-of-the-art result of 68.2% full repair rate.

SPoC. Similar to DeepFix, we measure the single step repair accuracy on the SPoC dev set (Table 4). The use of graph and pre-training both improve the repair performance (4.4% and 3.2% respectively; first three rows). As the SPoC task contains pseudocode, we also experimented with a version of our repair model that takes in pseudocode as input in addition to the broken code and compiler message. This provides a further boost in performance, achieving 68% single repair acc. on SPoC dev set (bottom row).

We then apply our repair model to the program synthesis setting (TestP, TestW), as described in §4.1.2. As seen in Table 5, our synthesis method equipped with DrRepair (bottom row) improves on the best first search significantly (e.g. +6% on TestP/TestW budget 100), suggesting that our repair model is useful for program synthesis as well.

We note that a concurrent work (Zhong et al., 2020) uses semantic constraints of programs to improve the search and achieves state-of-the-art results (46.1% / 62.8% on TestP / TestW). In contrast, our method only requires blackbox access to a compiler or executor. We believe the two approaches are complementary and it would be interesting to combine the approaches.

Example & Visualization. Figure 1 gives a real example of the output of "base+graph", as well as visualization of graph attention, where the pink highlighting in the source code shows the computed attention weights w.r.t. the '`a`' in the compiler message (the darker, the higher). It indicates that the model attends not only to the line reported by compiler (line 9), but also to the line that declared '`a`', which is the source of the error. This way, we can interpret the reasoning performed by our repair model.

4.4. Analysis

We aim to understand 1) the effect of different program corruption procedures and 2) graph representation methods, and 3) when self-supervision or graph is useful.

Different program corruption procedures. We compare DrPerturb (§3.2) with alternative program corruption procedures: Gupta+17, the original DeepFix work that corrupts delimiters or drops variable declarations only (so a subset of our **Syntax** and **ID-typo** module), and Random, a baseline that randomly drops tokens. We apply DrPerturb, Gupta+17, Random on the DeepFix data to create corresponding training sets (containing 156, 113, 170 types of compiler errors respectively). We then evaluate repair models trained on each of those training sets on the DeepFix raw test set (Table 6). We find that models trained by DrPerturb significantly outperform Gupta+17 (+10% repair rate), suggesting that the diverse set of errors covered by DrPerturb is useful. Additionally, while Random produces more distinct types of compiler errors than DrPerturb in terms of the number (170 vs 156), models trained by DrPerturb outperform Random by more than 10%, suggesting that DrPerturb generates a more useful distribution of errors than the Random baseline.

Compiler message type	Frequency in original train data (SPoC)	Repair acc. (SPoC dev)		
		base	+ graph	+ graph + pretrain
`‘@@@’ was not declared ...`	35.2 %	50.2	**58.9**	65.0
`redeclaration of ‘@@@’`	8.9 %	40.7	**43.0**	49.1
`expected ‘@@@’ before ‘@@@’`	3.2 %	67.6	70.7	86.1
`expected primary-expression before ...`	3.0 %	47.4	47.4	49.1
`request for member ‘@@@’ in ‘@@@’, ... (e.g. Figure 1)`	2.9 %	37.9	**56.9**	48.4
`expected initializer before ‘@@@’`	2.1 %	48.8	50.1	**93.0**
`‘@@@’ without a previous ‘@@@’`	1.3 %	37.0	38.7	**44.4**

Table 8. Breakdown of major compiler errors seen in SPoC dev (left), and the corresponding repair accuracy by our model variants (right). **Bold score** indicates a particularly big improvement from "base" to "+ graph" or from "+ graph" to "+ graph + pretrain".

Different graph representations. Table 7 shows an ablation study for the architecture of program-feedback graph. We find that the edges connecting symbols in source code ("edges among code only" row), and the edges spanning across source code and a compiler message ("edges across code-feedback only" row) are equally important, and the final program-feedback graph ("final" row) is the most effective. We also experimented with a version of our model that uses self-attention (i.e. we consider the complete graph), which we find comparable or slightly less effective ("self-attention" row). This suggests that the most important edges in the graph are those in our program-feedback graph, which connect symbols with semantic correspondence.

When is graph & self-supervision useful? We study what kinds of compiler errors a program-feedback graph or self-supervised pre-training is most useful for fixing. Table 8 shows the breakdown of major compiler errors seen in the SPoC dev set (left), and the repair accuracy of our model variants for each error type (right). We used the SPoC data as it exhibited more diverse errors than DeepFix.

We observe that the use of program-feedback graph is particularly helpful for compiler errors such as "`@@@ was not declared`" and "`request for member @@@...`" (those with bold scores in the "+graph" column), which typically require analyses of multiple lines of code (recall our example in Fig. 1). This suggests that a program-feedback graph indeed allows better information flow across source code lines and compiler messages, compared to the baseline sequence model ("base"). Additionally, we observe that self-supervised pre-training improves repair accuracy across most of the error types, but is noticeably helpful for errors that were relatively rare in the *original* training data (e.g. the bottom two), for which the use of a program-feedback graph only helped a little. This suggests that the extra training examples created in our self-supervision method help mitigate such data scarcity issues in original training data.

5. Related work and discussion

Graph neural networks. Graph neural nets (GNN), such as graph attention net (Veličković et al., 2018), graph convolutional net (Kipf & Welling, 2017), graph isomorphism net (Xu et al., 2019) have been shown to be effective for modeling graph-based data. Several works use GNNs to model the structure of text (Yasunaga et al., 2017; Zhang et al., 2018) and more recently, source code (Allamanis et al., 2018; Brockschmidt et al., 2019). Allamanis et al. (2018) present program graph that augments AST with data flow edges across variables, which is passed to GNNs to solve the task of variable name prediction. Brockschmidt et al. (2019) build on it and design a graph-based generative model for source code. Gupta et al. (2019a) propose a tree convolution model to encode ASTs. Distinct from the above works, we focus on the problem of program repair, and design the program-feedback graph to represent the dependencies between source code and diagnostic feedback. Our results show that GNNs can fruitfully represent these program-feedback dependencies for program repair.

Self-supervised pre-training. The idea of using unlabeled data to pre-train neural networks has been shown effective across many fields, including computer vision (Vincent et al., 2008; Erhan et al., 2010), NLP (Peters et al., 2018; Devlin et al., 2019), graphs (Hu et al., 2020), and programming languages (Feng et al., 2020). Typically, the self-supervised pre-training objective is different from the target task: For instance, in image recognition, Vincent et al. (2008) pre-train networks via a denoising autoencoder; in NLP, Devlin et al. (2019) pre-train networks via masked language modeling and then fine-tune on a target task such as question answering. In contrast, our pre-training task *is* the program repair task (our target task), as we prepare the pre-training data by corrupting unlabeled programs and obtaining diagnostic feedback to synthesize program repair examples. Additionally, our pre-training task is conditioned on diagnostic feedback, which is a new type of structure from a pre-training perspective and provides better generalization at the test time as we show in §4.3.

Learning program repair. There is increasing interest in automatic correction of introductory programming assignments (Pu et al., 2016; Parihar et al., 2017; Ahmed et al., 2018). DeepFix (Gupta et al., 2017) is an early work that uses a seq2seq model to translate a broken code into fixed one. RLAssist (Gupta et al., 2019b), SampleFix (Hajipour et al., 2019) improve on it by introducing reinforcement learning or better sampling methods. While these works purely use sequence models, we propose to use a graph representation of source code and diagnostic feedback to capture long-range dependencies of symbols across them.

Another line of work learns from labeled datasets of how programmers edit code (e.g. error resolution records) (Just et al., 2014; Chen et al., 2019). Mesbah et al. (2019) model a Java build error resolution record using seq2seq. Tarlow et al. (2019) generalize it to more diverse error types, and propose a repair model that uses the graph structure of AST. Bader et al. (2019) present a hierarchical clustering algorithm to learn program repair patterns. While these works rely purely on labeled datasets of program repair, we propose a self-supervised learning paradigm that leverages a large amount of unlabeled data to create extra training examples for program repair.

Finally, several works focus on repairing specific types of bugs, e.g., variable misuse (Vasic et al., 2019), name-based bugs (Pradel & Sen, 2018) and Javascript bugs (Dinella et al., 2020). Other works focus on modeling program execution (Wang et al., 2018) or edits (Zhao et al., 2019). We refer readers to Monperrus (2018) for a more comprehensive review of automated program repair.

6. Conclusion

This paper makes two contributions to program repair from diagnostic feedback. First, we proposed the program-feedback graph to model the reasoning process in program repair. We find this particularly useful when the repair requires analyzing multiple lines of code. Second, we introduced a self-supervised learning paradigm that creates extra program repair examples by corrupting unlabeled programs and obtaining feedback from an evaluator (compiler). We find this effective for overcoming the scarcity of labeled data for program repair.

While we primarily focus on program repair in this paper, we note that our framework of learning to edit based on feedback is a potentially powerful and more general paradigm with many applications, from learning to edit essays based on written feedback, to learning from users in interactive dialogue, etc. (Liu et al., 2018). The key is that rather than using a single number reward (e.g. compile or not) as in reinforcement learning, obtaining high bandwidth feedback via diagnostic feedback can be much more informative if we incorporate it effectively, for instance through the use of graph neural networks as we presented in this work.

Reproducibility

All code and data are available at `https://github.com/michiyasunaga/DrRepair`. Experiments are available at `https://worksheets.codalab.org/worksheets/0x01838644724a433c932bef4cb5c42fbd`.

Acknowledgments

We thank John Hewitt, Mina Lee, Sumith Kulal, Pang Wei Koh, Robin Jia, Ananya Kumar, Ruiqi Zhong, and anonymous reviewers for insightful feedback and discussions. This work was supported in part by NSF CAREER Award IIS-1552635, a PECASE Award, and an Amazon Research Award.

References

Ahmed, U. Z., Kumar, P., Karkare, A., Kar, P., and Gulwani, S. Compilation error repair: for the student programs, from the student programs. In *ICSE*, 2018.

Allamanis, M., Brockschmidt, M., and Khademi, M. Learning to represent programs with graphs. In *ICLR*, 2018.

Bader, J., Scott, A., Pradel, M., and Chandra, S. Getafix: Learning to fix bugs automatically. In *OOPSLA*, 2019.

Brockschmidt, M., Allamanis, M., and Gaunt, A. Generative code modeling with graphs. In *ICLR*, 2019.

Chen, Z., Kommrusch, S. J., Tufano, M., Pouchet, L.-N., Poshyvanyk, D., and Monperrus, M. Sequencer: Sequence-to-sequence learning for end-to-end program repair. *IEEE Transactions on Software Engineering*, 2019.

Devlin, J., Chang, M.-W., Lee, K., and Toutanova, K. Bert: Pre-training of deep bidirectional transformers for language understanding. In *NAACL*, 2019.

Dinella, E., Dai, H., Li, Z., Naik, M., Song, L., and Wang, K. Hoppity: Learning graph transformations to detect and fix bugs in programs. In *ICLR*, 2020.

Erhan, D., Bengio, Y., Courville, A., Manzagol, P.-A., Vincent, P., and Bengio, S. Why does unsupervised pre-training help deep learning? In *Journal of MachineLearning Research*, 2010.

Feng, Z., Guo, D., Tang, D., Duan, N., Feng, X., Gong, M., Shou, L., Qin, B., Liu, T., Jiang, D., and Zhou, M. Codebert: A pre-trained model for programming and natural languages. 2020.

Fitzgerald, S., Lewandowski, G., McCauley, R., Murphy, L., Simon, B., Thomas, L., and Zander, C. Debugging: finding, fixing and flailing, a multi-institutional study of novice debuggers. *Computer Science Education*, 2008.

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

Gupta, R., Kanade, A., and Shevade, S. Neural attribution for semantic bug-localization in student programs. In *NeurIPS*, 2019a.

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

Hajipour, H., Bhattacharya, A., and Fritz, M. Samplefix: Learning to correct programs by sampling diverse fixes. In *arXiv:1906.10502*, 2019.

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

Hu, W., Liu, B., Gomes, J., Zitnik, M., Liang, P., Pande, V., and Leskovec, J. Pre-training graph neural networks. In *ICLR*, 2020.

Just, R., Jalali, D., and Ernst, M. D. Defects4j: A database of existing faults to enable controlled testing studies for java programs. In *ISSTA*, 2014.

Kingma, D. and Ba, J. Adam: A method for stochastic optimization. In *ICLR*, 2015.

Kipf, T. N. and Welling, M. Semi-supervised classification with graph convolutional networks. In *ICLR*, 2017.

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

Liu, B., Tür, G., Hakkani-Tür, D., Shah, P., and Heck, L. Dialogue learning with human teaching and feedback in end-to-end trainable task-oriented dialogue systems. In *NAACL*, 2018.

Mesbah, A., Rice, A., Johnston, E., Glorioso, N., and Aftandilian, E. Learning to repair compilation errors. In *ESEC/FSE*, 2019.

Monperrus, M. The living review on automated program repair. *Technical Report hal-01956501. HAL/archives-ouvertes.fr.*, 2018.

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

Pascanu, R., Mikolov, T., and Bengio, Y. On the difficulty of training recurrent neural networks. *arXiv:1211.5063*, 2012.

Peters, M., Neumann, M., Iyyer, M., Gardner, M., Clark, C., Lee, K., and Zettlemoyer, L. Deep contextualized word representations. In *NAACL*, 2018.

Pradel, M. and Sen, K. Deepbugs: a learning approach to name-based bug detection. In *OOPSLA*, 2018.

Pu, Y., Narasimhan, K., Solar-Lezama, A., and Barzilay, R. sk_p: a neural program corrector for moocs. In *SPLASH Companion*, 2016.

See, A., Liu, P. J., and Manning, C. D. Get to the point: Summarization with pointer-generator networks. In *ACL*, 2017.

Seo, H., Sadowski, C., Elbaum, S., Aftandilian, E., and Bowdidge, R. Programmers' build errors: A case study at google. In *ICSE*, 2014.

Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., and Salakhutdinov, R. Dropout: A simple way to prevent neural networks from overfitting. *JMLR*, 2014.

Tarlow, D., Moitra, S., Rice, A., Chen, Z., Manzagol, P.-A., Sutton, C., and Aftandilian, E. Learning to fix build errors with graph2diff neural networks. In *arXiv:1911.01205*, 2019.

Vasic, M., Kanade, A., Maniatis, P., Bieber, D., and Singh, R. Neural program repair by jointly learning to localize and repair. In *ICLR*, 2019.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, L., and Polosukhin, I. Attention is all you need. In *NeurIPS*, 2017.

Veličković, P., Cucurull, G., Casanova, A., Romero, A., Liò, P., and Bengio, Y. Graph attention networks. In *ICLR*, 2018.

Vincent, P., Larochelle, H., Bengio, Y., and Manzagol, P.-A. Extracting and composing robust features with denoising autoencoders. In *ICML*, 2008.

Wang, K., Singh, R., and Su, Z. Dynamic neural program embeddings for program repair. In *ICLR*, 2018.

Xu, K., Hu, W., Leskovec, J., and Jegelka, S. How powerful are graph neural networks? In *ICLR*, 2019.

Yasunaga, M., Zhang, R., Meelu, K., Pareek, A., Srinivasan, K., and Radev, D. R. Graph-based neural multi-document summarization. In *CoNLL*, 2017.

Zhang, Y., Qi, P., and Manning, C. D. Graph convolution over pruned dependency trees improves relation extraction. In *EMNLP*, 2018.

Zhao, R., Bieber, D., Swersky, K., and Tarlow, D. Neural networks for modeling source code edits. In *arXiv:1904.02818*, 2019.

Zhong, R., Stern, M., and Klein, D. Semantic scaffolds for pseudocode-to-code generation. *arXiv:2005.05927*, 2020.