Representing Programs with Dependency and Function Call Graphs for Learning Hierarchical Embeddings

Vitaly Romanov, Vladimir Ivanov and Giancarlo Succi
Innopolis University, Innopolis, Russia

Keywords: Source Code, Embeddings, Hierarchical Embeddings, Graph, Dataset, Machine Learning, Python, Java.

Abstract: Any source code can be represented as a graph. This kind of representation allows capturing the interaction between the elements of a program, such as functions, variables, etc. Modeling these interactions can enable us to infer the purpose of a code snippet, a function, or even an entire program. Lately, more and more work appear, where source code is represented in the form of a graph. One of the difficulties in evaluating the usefulness of such representation is the lack of a proper dataset and an evaluation metric. Our contribution is in preparing a dataset that represents programs written in Python and Java source codes in the form of dependency and function call graphs. In this dataset, multiple projects are analyzed and united into a single graph. The nodes of the graph represent the functions, variables, classes, methods, interfaces, etc. Nodes for functions carry information about how these functions are constructed internally, and where they are called from. Such graphs enable training hierarchical vector representations for source code. Moreover, some functions come with textual descriptions (docstrings), which allows learning useful tasks such as API search and generation of documentation.

1 INTRODUCTION

Given the current rate of software development, it is desirable to develop instruments that can assist the development process and help to bring more quality software to life. Specifically, when considering the development tasks associated with source code documentation. Very often, parts of a software project lack necessary documentation. Combining text and source code analysis together can enable search through undocumented code or even facilitate the process of creating the documentation.

Unfortunately, modern instruments for software analysis are limited. The reason for this is simply that those tools are unable to understand the purpose of a program. In this situation, analysis tools are unable to provide relevant recommendations or feedback.

Another area that lacks proper instruments is the creation of the documentation. Source code documentation is an important factor for software quality, especially in the case of long-term development and collaboration. Often, it is necessary to understand the purpose of the program to get any valuable insight. However, without proper documentation, it is sometimes hard to do even for experienced programmers. A tool that can facilitate the interpretation of an undocumented code will make this process easier.

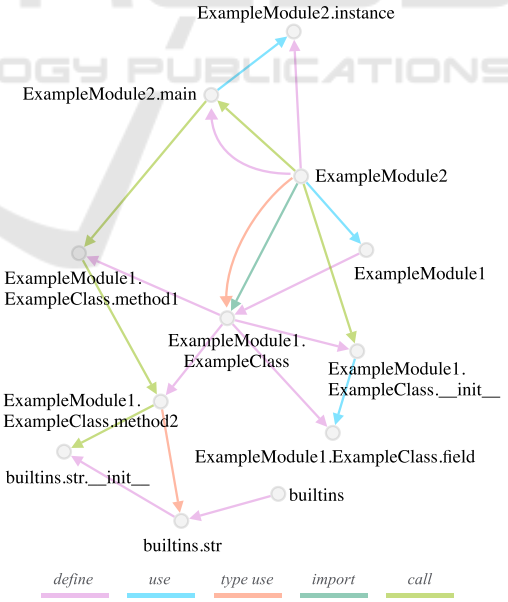

Figure 1: Source code graph that captures definition and usage of a simple Python class. Original source code is given in Listing 1.

Lately, there were some developments in the ap-

Romanov, V., Ivanov, V. and Succi, G.
Representing Programs with Dependency and Function Call Graphs for Learning Hierarchical Embeddings.
DOI: 10.5220/0009511803600366
In *Proceedings of the 22nd International Conference on Enterprise Information Systems (ICEIS 2020) - Volume 2*, pages 360-366
ISBN: 978-989-758-423-7

plications of statistical learning and deep learning techniques for solving tasks that are related to program interpretation. Such tasks include generation of software description (Yao et al., 2019) (Matskevich and Gordon, 2018) (Alon et al., 2018a) (Aghamohammadi et al., 2018), API search (Gu et al., 2018) (Husain et al., 2019), bug detection (Henkel et al., 2018) (Li et al., 2019) (Pradel and Sen, 2018) and others. These problems are hard to formalize, and there is a general consensus that they are better addressed with the use of deep learning techniques.

One of the possible improvements that can be done in the area of source code modeling with deep learning techniques is changing the representation of the code. Oftentimes, the code is treated as a sequence of tokens. Such representation loses a lot of useful structure in the code that is available to a human since the human knows programming language grammar. Another, more natural approach to source code representation, is with graphs (Alon et al., 2018b). Recently, due to the advancements in the area of graph modeling with deep learning, the approaches that represent the source code in the form of graphs started to emerge (Allamanis et al., 2017), (Alon et al., 2019), (Nguyen et al., 2017).

Source code has an inherently hierarchical structure. A program consists of a sequence of functions calls. A function can be implemented with the help of other functions or simply using the standard library of a programming language. Thus, any program can be represented with a hierarchy of function calls. We believe that understanding how to model this hierarchy will enable interpreting the program's purpose.

Unfortunately, it is hard to evaluate how useful a graph representation can be using only existing datasets. Most published datasets aim at modeling abstract syntax tree (AST) of a function without considering how this function is used in the rest of the code. We believe that to understand the purpose of a function, one should inspect both how the function is used throughout the source code and the body text of the function itself. Moreover, when a textual description of a function is added, a statistical model receives the maximum amount of information that a programmer can receive. Considering these three properties of a function, in our opinion, should give a full description of its purpose, and enable learning more expressive vector representations (embeddings) for source code.

The understanding of the purpose of a function enables solving multiple tasks that can facilitate the software development process and reduce the risk of error. The use of textual descriptions can enable the use of natural language interfaces in the software development process, such as API search. The contribution of this paper is in preparing datasets for representing source code for Python and Java as dependency and function call graphs [1]. Besides a more conventional function call graph, this dataset contains information about the dependencies of a particular function. The list of dependencies can include types of variables, references for variables, or class fields. We plan to use this dataset for learning hierarchical embeddings and studying the application of such embeddings for solving different machine learning (ML) tasks for source code.

The rest of the paper is organized as follows. Section 2 describes the idea behind the proposed approach. Section 3 explains how we created the datasets for Python and Java. Section 4 explores the related work. Section 5 concludes the paper.

2 METHOD DESCRIPTION

The approach is based on the idea that information about the function body, how and where the function is used, and the textual description of a function provide full information necessary to determine the purpose of a function. We draw inspiration from the area of natural language processing. Given that the source code, written in any programming language, can be treated as a form of language, we can apply existing NLP techniques to create a language model (Hindle et al., 2012). However, some differences make the source code very different from natural language.

First, the source code usually describes a sequence of transformations. Unlike in natural language, where there is a significant prevalence of nouns, source code is mostly described by actions, and the intent of a program is mostly defined through the composition of these actions (Hindle et al., 2012).

The second difference is related to the language vocabulary. Natural languages usually have an evolving but relatively stable vocabulary with a wide variety of concepts. In source code, on the other hand, vocabulary can change significantly from a program to a program due to different naming of variables. Moreover, the variables that are essentially the same can have different names in different scopes. In the case of natural language, such situations are handled by co-reference resolution, which is a probabilistic technique. In the case of source code, those references, in most of the cases, can be resolved in a deterministic way (at least in the run-time). This creates an opportunity to track how different variables and objects are

[1] https://github.com/VitalyRomanov/source-graph-dataset

used across the entire program, discern between them, and identify their purpose.

Despite the long history, many natural language models are limited in a way that they primarily operate on the token level. Creating a programming language model creates a unique opportunity to utilize the graph-like hierarchical structure of a program. It is a known fact that programs are often organized in a fashion that allows reusing the source code. In the graph of an entire program, functions represent subgraphs.

Computing embeddings for subgraphs is an open problem. Source code graphs allow addressing this problem in an interesting way. The problem of learning the embeddings for subgraphs can be interpreted as a problem of learning a composition function that accepts nodes and edges as inputs and produces an embedding for the entire subgraph. In the case of the source code graph, the entire subgraph is additionally represented as a single node (function) that has connections with the rest of the source code. Being able to learn the composition function allows finding the representation for new functions that were not present in the dataset, and, in general, opens up an opportunity for more intuitive transfer learning on source code, which is different from more conventional sequence-to-sequence approaches for transfer learning (Devlin et al., 2017).

In our opinion, the best way to measure how well a model captures the purpose of a program or function is by evaluating the tasks of generating the annotation and API search. Both of these tasks require textual descriptions attached to nodes (functions, modules) in the graph. One of the advantages of textual descriptions, often given as docstring for functions, is that they enable the use of more advanced and established NLP techniques for comparing the similarity of text. Thus, textual descriptions can be used as additional regularization for learning graph embeddings. Nodes that have similar textual descriptions should have similar graph embeddings.

With the progress of deep learning, vector representations became a widely applied tool for solving varieties of ML tasks. The advantage of vector representations is that they automatically encode properties of an object, that are useful for solving downstream tasks. Currently, there is a growing list of problems that can be solved by learning vector representations for source code. Some of them include:

- Auto-completion and API Suggestion

  In this task, a program tries to recommend additional API that can be useful for completing the current program. This task can be viewed as link prediction in graphs. This is also a variant of API search;
- API search

  Usually involves searching for API using a query in natural language. Embeddings for functions are used for creating a search index and learning ranking function;
- Bug Detection

  Involves finding bugs that can be revealed from the statistical analysis of the code. Embeddings are used to represent elements of a program;
- Generating Annotations

  Generating textual description for a function or separate lines in the code
- Program Translation

  Translation between programming languages

```
# ExampleModule2.py
from ExampleModule1 import ExampleClass

instance = ExampleClass(None)

def main():
    print(instance.method1())

main()

# ExampleModule1.py

class ExampleClass:
    def __init__(self, argument):
        self.field = argument

    def method1(self):
        return self.method2()

    def method2(self):
        variable1 = 2
        variable2 = str(2)
        return variable2
```

Listing 1: A simple class definition and use in Python. Source code graph for this code is given in Figure 1.

By utilizing the hierarchical structure of a program, we can regularize embeddings of functions to encode the operations and other methods that are used inside this function. Some work on modeling hierarchy of graphs was done in (Ribeiro et al., 2017), (Chen et al., 2018),(Ying et al., 2018). At the same time, the dependency links in the source code graph can be used to learn how and where a specific function can be used without even looking at the implementation of this function.

Based on the ideas described above, we propose a list of criteria that should be met in order for the source code graph to be useful:

1. Presence of the Hierarchy. The graph should contain a hierarchy of function calls. Otherwise, it becomes hard to model how other functions are used;
2. Clarity. Functions in the graph mostly should have a single purpose;
3. Multiple Usages. Functions should ideally be reused from different parts of the code. Otherwise, it becomes hard for a statistical model to understand the purpose of the function;
4. Connectivity. Functions should be connected with the rest of the program through input arguments and return values[2].

All of the tasks described above require some additional level of understanding of the program's purpose. In our opinion, tasks like API search and code annotation are the best for evaluating the ability of a statistical model to capture the program's purpose. If a model can correctly generate different textual descriptions for two similar functions, then it is able to interpret the code. Similarly, one can evaluate how well a model understands the source code using the task of API search.

3 DATASET DESCRIPTION

We constructed a source code graph with the help of an open-source source code indexing tool – Sourcetrail [3]. This tool is capable of creating a graph for C, C++, Java, and Python. The main advantage of Sourcetrail over its competitors is the possibility of exporting nodes and edges of a source code graph with minimal effort. Moreover, it provides a unified interface for creating graphs for several programming languages at once. This utility can capture different kinds of relationships between source code units. The source code units, such as modules, classes, methods, fields, and variables, are treated as nodes in the graph. There are several types of relationships. The list of available relationships depends on the programming language. Despite different kinds of relationships present, the main goal is to capture the function call graph. Other types of connections can still be useful for modeling the source code in the future. We applied the source code indexing tool Sourcetrail to a set of Python packages and their dependencies and also to a set of open-source Java repositories. The details of the resulting datasets are given below. The example of a graph constructed in such way can be found on Figure 1.

Table 1: Node types present in Python source graph.

Node Type	Count
Function	221822
Class field	83077
Class	35798
Module	18097
Class method	14953
Non-indexed symbol	853

Table 2: Edge count in Python graph by edge type.

Edge Type	Count
Call	614621
Define/Contain	431115
Type Use	239543
Import	121752
Inheritance	26525

Dataset construction process included several steps. First, the data was exported from Sourcetrail format. The resulting graphs were analyzed for connected components. Only the largest component remained, and the rest were filtered out. Sometimes indexing tools can produce erroneous edges. The correctness of graphs was verified manually on a small subsample of nodes and edges. With 99% confidence source graph for Python has less than 8% of incorrect connections, and for Java – less than 5%. The estimate was performed on a very small subsample and is expected to go down as more data is manually verified.

Python Dataset.

To create the Python source graph dataset, we created a virtual environment and installed a collection of popular packages, including their dependencies. The collection of packages includes Bokeh, Django, Fabric, Flask, Matplotlib, Pandas, Requests, Scrapy, Sklearn, Spacy, and Tensorflow. After installing these packages with dependencies, the python virtual environment contained 151 packages. All of those packages were indexed using Sourcetrail. The count of different types of nodes is shown in the Table 1[4]. The list of different types of edges and their count is shown in Table 2.

Java Dataset.

In the case of Java, we analyzed 15 repositories that are openly available on GitHub. The repositories include the source code for such projects as Apache HTTP Client, crawler4j, Deeplearning4j, JHipster,

[2]This property is not yet fully satisfied in the current version of the dataset.

[3]www.sourcetrail.com

[4]The exact count can change in the future versions of the dataset. Applies to all counts.

Table 3: Node types present in Java source graph.

Node Type	Count
Method	287393
Field	95983
Class	42329
Non-Indexed Symbol	30438
Type Parameter	5885
Enum Constant	4835
Non-Indexed Package	3792
Interface	3366
Annotation	1012
Enum	919
Built-In Type	9

Table 4: Edge count in Java graph by edge type.

Edge Type	Count
Type use	989725
Call	817408
Define/Contain	475303
Use	368910
Annotation use	168862
Override	75186
Inheritance	32018
Is type	24105

log4j, Mahout, Opennlp, Spring Boot, Spring Framework, Stanford NLP, TableSaw, Thingsboard, Unirest, WebMagic, and Weka. Unlike for Python, we did not analyze the packages that are the dependants of these repositories. The count of nodes and edges by type is given in tables 3 and 4 respectively. As with the case of the Python graph, there are many different types of nodes and edges. For now, we focus on the edges that create the function call graph. Other types of edges and nodes can be used in the future.

For each dataset, where it was possible, we extracted docstrings for functions. Thus, some of the functions in the dataset come with a textual description.

The datasets described above represent the source code of selected Python packages and Java repositories in the form of graphs. The nodes of the graph, in this case, are the units of programming languages, such as functions, methods, classes, interfaces, etc. In these datasets, some of the edges represent function calls. Thus, it is possible to use this data for training hierarchical embeddings for the functions. Some of the functions, for which docstring was available, have a textual description attached to them. These descriptions enable solving the tasks that require input or output in the form of natural language. Such ML tasks for code as API search, annotation generation, API recommendation, and others can be tested on the current dataset.

4 RELATED WORK

Existing Datasets.

The idea of representing the source code in the form of graphs is not new and was performed in different studies. However, bringing together a hierarchical representation of a program and textual descriptions is still rare, and the existing public datasets do not provide the data in the desired format.

We looked into datasets for source code search and description generation – the types of datasets that come with textual description for functions. All of the datasets that we have found did not allow constructing an unambiguous hierarchical representation of a program.

One of the datasets, collected by the researchers in Edinburgh University, code-docstring-corpus, contains a parallel corpus of python functions and their docstrings and enables training models for generating function descriptions from the function bodies (Miceli-Barone and Sennrich, 2017). A collection of open-source python repositories was used to collect the dataset. Many functions come from the same library. However, there seems to be no easy way to construct an unambiguous function call graph from this data, mainly due to functions having identical names, and no easy way to identify where the functions were imported from.

The dataset published by GitHub, CodeSearchNet, contains function bodies, links to the source repository, and, sometimes, function descriptions (Husain et al., 2019). The data is available for several languages. The dataset contains functions from a very diverse set of repositories. For this reason, many functions are not really reused, which makes it hard to construct a hierarchical representation.

In Facebook's dataset, Neural-Code-Search-Evaluation-Dataset, a parallel corpus of code snippets and their textual descriptions is given (Sachdev et al., 2018). But, once again, it seems hard to construct a hierarchical representation. For this reason, this dataset does not satisfy our goals.

Another project that caught our attention is BOA projects. It aims at mining a large number of source code repositories (Dyer et al., 2013). They have implemented a query language to generate and mine different types of program graphs, including control-flow graphs, control-dependence graphs, data-dependence graphs, and program-dependence graphs. However, the data is not readily available

and should be queried. Moreover, currently, they analyze only for Java 7 and older, according to the project web-page.

Techniques for Source Code Embeddings.

One of the goals of creating the source code graph dataset is learning vector representations for the source code.

An extensive study pf the subject of source code embedding was done in (Chen and Monperrus, 2019). In most of the cases, creating embeddings require building some graph. Several methods were reported to build graphs based on token adjacency (Harer et al.,) (Azcona et al., 2019) (Chen and Monperrus, 2018). Such an approach is very similar to classical embeddings techniques, like Word2Vec, but seems to be over-simplified in the case of program source codes.

Another approach for learning embeddings is using a control-flow graph (DeFreez et al., 2018). While such an approach would resolve many ambiguities in graph construction, the process of creating the control-graph itself is not possible for any arbitrary language. The same problem arises when using precompiled LLVM code for creating the graph (Ben-Nun et al., 2018). Moreover, the program becomes uninterpretable for a programmer. This is one of the reasons we focus on analyzing sources of a program as they are without additional preprocessing.

Some other approaches work on with function call graph and have reported learning meaningful embeddings (Lu et al., 2019) (Pradel and Sen, 2018). However, the data that they have uses is not available publicly.

Several approaches used graph-based source code representation for fixing bugs in the code (Allamanis et al., 2017) (Devlin et al., 2017). Their graph includes both function call information, as well as function AST. However, these graphs are not publicly available.

As of the time of writing this paper, we are not aware of any work that utilizes everything: hierarchical representations, function bodies, and textual descriptions at the same time.

5 CONCLUSION

Instruments that can facilitate the software development process are needed. Due to the latest advancements in statistical learning, it now becomes possible to address such problems as source code search and generation of documentation. Lately, numerous works address these problems. In this paper, we presented our vision for an approach to source code analysis that utilizes hierarchical graph representations, function bodies, and textual descriptions for functions. It is possible to use this information to train a model that captures the program's purpose. We created two datasets for Python and Java projects. Each individual project is represented as a graph. These graphs were later united together into a single graph (for a given programming language). This large graph contains information about both: the internal structure of each function and how this function is used in the source code is present. We believe that this information will be beneficial for learning embeddings for functions. Such embeddings allow solving such ML tasks like API search, generation of documentation, API suggestion, bug detection, and many others. Both datasets are available on GitHub.

Currently, the datasets are still under development, and their improvement is expected within the next several months. The future work includes incorporating AST of programs into the current graph. The datasets can be used for studying properties of source code, e.g., the amount of code reuse. However, the main goal of creating these datasets is to explore the possibility of creating hierarchical embeddings for source code.

REFERENCES

Aghamohammadi, A., Heydarnoori, A., and Izadi, M. (2018). Generating summaries for methods of event-driven programs: an android case study. *arXiv preprint arXiv:1812.04530*.

Allamanis, M., Brockschmidt, M., and Khademi, M. (2017). Learning to represent programs with graphs. *arXiv preprint arXiv:1711.00740*.

Alon, U., Brody, S., Levy, O., and Yahav, E. (2018a). code2seq: Generating sequences from structured representations of code. *arXiv preprint arXiv:1808.01400*.

Alon, U., Zilberstein, M., Levy, O., and Yahav, E. (2018b). A general path-based representation for predicting program properties. *ACM SIGPLAN Notices*, 53(4):404–419.

Alon, U., Zilberstein, M., Levy, O., and Yahav, E. (2019). code2vec: Learning distributed representations of code. *Proceedings of the ACM on Programming Languages*, 3(POPL):1–29.

Azcona, D., Arora, P., Hsiao, I.-H., and Smeaton, A. (2019). user2code2vec: Embeddings for profiling students based on distributional representations of source code. In *Proceedings of the 9th International Conference on Learning Analytics & Knowledge*, pages 86–95.

Ben-Nun, T., Jakobovits, A. S., and Hoefler, T. (2018). Neural code comprehension: A learnable representation of code semantics. In *Advances in Neural Information Processing Systems*, pages 3585–3597.

Chen, H., Perozzi, B., Hu, Y., and Skiena, S. (2018). Harp: Hierarchical representation learning for networks. In

Thirty-Second AAAI Conference on Artificial Intelligence.

Chen, Z. and Monperrus, M. (2018). The remarkable role of similarity in redundancy-based program repair. *arXiv preprint arXiv:1811.05703*.

Chen, Z. and Monperrus, M. (2019). A literature study of embeddings on source code. *arXiv preprint arXiv:1904.03061*.

DeFreez, D., Thakur, A. V., and Rubio-González, C. (2018). Path-based function embedding and its application to specification mining. *arXiv preprint arXiv:1802.07779*.

Devlin, J., Uesato, J., Singh, R., and Kohli, P. (2017). Semantic code repair using neuro-symbolic transformation networks. *arXiv preprint arXiv:1710.11054*.

Dyer, R., Nguyen, H. A., Rajan, H., and Nguyen, T. N. (2013). Boa: A language and infrastructure for analyzing ultra-large-scale software repositories. In *2013 35th International Conference on Software Engineering (ICSE)*, pages 422–431. IEEE.

Gu, X., Zhang, H., and Kim, S. (2018). Deep code search. In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*, pages 933–944. IEEE.

Harer, J. A., Kim, L. Y., Russell, R. L., Ozdemir, O., Kosta, L. R., Rangamani, A., Hamilton, L. H., Centeno, G. I., Key, J. R., Ellingwood, P. M., et al. Automated software vulnerability detection with machine learning.

Henkel, J., Lahiri, S. K., Liblit, B., and Reps, T. (2018). Code vectors: Understanding programs through embedded abstracted symbolic traces. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 163–174.

Hindle, A., Barr, E. T., Su, Z., Gabel, M., and Devanbu, P. (2012). On the naturalness of software. In *2012 34th International Conference on Software Engineering (ICSE)*, pages 837–847. IEEE.

Husain, H., Wu, H.-H., Gazit, T., Allamanis, M., and Brockschmidt, M. (2019). Codesearchnet challenge: Evaluating the state of semantic code search. *arXiv preprint arXiv:1909.09436*.

Li, Y., Wang, S., Nguyen, T. N., and Van Nguyen, S. (2019). Improving bug detection via context-based code representation learning and attention-based neural networks. *Proceedings of the ACM on Programming Languages*, 3(OOPSLA):1–30.

Lu, M., Liu, Y., Li, H., Tan, D., He, X., Bi, W., and Li, W. (2019). Hyperbolic function embedding: Learning hierarchical representation for functions of source code in hyperbolic space. *Symmetry*, 11(2):254.

Matskevich, S. and Gordon, C. S. (2018). Generating comments from source code with ccgs. In *Proceedings of the 4th ACM SIGSOFT International Workshop on NLP for Software Engineering*, pages 26–29.

Miceli-Barone, A. V. and Sennrich, R. (2017). A parallel corpus of python functions and documentation strings for automated code documentation and code generation. In *Proceedings of the Eighth International Joint Conference on Natural Language Processing (Volume 2: Short Papers)*, pages 314–319.

Nguyen, T. D., Nguyen, A. T., Phan, H. D., and Nguyen, T. N. (2017). Exploring api embedding for api usages and applications. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, pages 438–449. IEEE.

Pradel, M. and Sen, K. (2018). Deepbugs: A learning approach to name-based bug detection. *Proceedings of the ACM on Programming Languages*, 2(OOPSLA):1–25.

Ribeiro, L. F., Saverese, P. H., and Figueiredo, D. R. (2017). struc2vec: Learning node representations from structural identity. In *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 385–394.

Sachdev, S., Li, H., Luan, S., Kim, S., Sen, K., and Chandra, S. (2018). Retrieval on source code: a neural code search. In *Proceedings of the 2nd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*, pages 31–41.

Yao, Z., Peddamail, J. R., and Sun, H. (2019). Coacor: code annotation for code retrieval with reinforcement learning. In *The World Wide Web Conference*, pages 2203–2214.

Ying, Z., You, J., Morris, C., Ren, X., Hamilton, W., and Leskovec, J. (2018). Hierarchical graph representation learning with differentiable pooling. In *Advances in neural information processing systems*, pages 4800–4810.