Approaches for Representing Software as Graphs for Machine Learning Applications

Vitaly Romanov
Innopolis University
Russia
v.romanov@innopolis.ru

Vladimir Ivanov
Innopolis University
Russia
v.ivanov@innopolis.ru

Giancarlo Succi
Innopolis University
Russia
g.succi@innopolis.ru

2020 International Computer Symposium (ICS) | 978-1-7281-9255-0/20/$31.00 ©2020 IEEE | DOI: 10.1109/ICS51289.2020.00109

***Abstract*—Machine learning (ML) is making its way into the source code analysis. Most of the time, this happens with the help of Natural Language Processing (NLP) techniques. However, NLP techniques often represent their input as a sequence of tokens. This assumption is reasonable when processing text because the words related to the same object usually follow each other. However, in source code, this assumption can be inadequate simply because of the source code execution nature. Graphs can be much more adequate for representing source code. They can capture the dependency structure of a program. Due to the recent advances in the area of machine learning on graphs, researchers started to explore the graph-based representation of software in the scope of machine learning applications. There is no single way to represent a program in the form of a graph. For this reason, researchers explored different alternatives, such as function call graphs (FCG), data flow graphs (DFG), control flow graphs (CFG), or their mixtures. In this survey, we overview approaches for representing software as graphs and how these representations help to solve machine learning tasks.**

***Keywords*—** statistical learning, machine learning, graphs, source code representation, source code, software

I. Introduction

A. Motivation

Unlike in other areas, machine learning has not yet seen any major application in the area of source code development. One of the issues is often attributed to the inherent complexity of the source code, which makes it very difficult for machine learning systems to understand. The latest advances in machine learning on graphs promise a better approach for modeling source code. Representations of a program in the form of control-flow, data-flow, or function call graphs are very intuitive and allow capturing much-longer-distance dependencies in the source code than the token-level representation does. In this paper, we overview how existing approaches use graph representations for source code and what kind of tasks they are trying to solve.

B. Advantages of graphs for software analysis

Graphs are a very evident way to represent source code. The properties of the graphs constructed from source codes have been under investigation for a long time. Different aspects of such graphs were studied. Some works focus on graph properties [47, 36, 17, 29, 37, 14] , and other [23, 22] - on inferring program behavior using static and dynamic analysis. Understanding the program behavior is of great practical importance because sometimes it allows detecting errors in the early stages.

C. Statistical learning for software analysis

One of the limitations of the deterministic graph analysis tools is their inability to detect source code problems that are ambiguous. Learning techniques assign confidence to their decisions and help detect issues by also allowing some degree of error. Moreover, they can be used to identify context-specific problems.

It is important to note that learning methods allow solving tasks that remain infeasible otherwise. Such tasks include variable naming [2, 15, 33, 6, 26, 3], function summarization [21, 32, 34, 3], bug detection [2, 19, 41, 16], and others (see Table III. Solving these problems would require extracting high-level features from a low-level representation, which is not straightforward with algorithmic approaches.

D. Need for literature overview

Although the use of graph representation for software analysis is not new, its use in conjunction with statistical learning is a relatively recent trend. Nevertheless, the survey of research literature shows that there is already some progress. It is valuable to draw some conclusions about the current status of research.

We are not aware of any survey that overviews the current status of the software analysis using machine learning on graphs, except for a few very specific areas. Chen and Monperrus conducted a study of embeddings on source code [13]. In some works, the models for learning embeddings assume the graph structure of the input. This work provides an overview of approaches to train embeddings and does not investigate the applications of such embeddings on downstream tasks. In their work on code clone detection, White et al. provided an overview of different representation techniques in application to source code clone detection [43]. However, their overview was very narrow. A short overview of approaches for generating natural language from source code was done in [30]. This overview did not target graph representations in particular. A comprehensive overview of machine learning for source code was done in [1]. The authors considered a large volume of literature and did an extensive study on applications. However, there were updates within the last years, and graph-based approaches are not really presented in the survey.

The examples that are given above focus on specific tasks or on a very general overview of the field. Our goal is to focus on the specific data representation format and see how it is used in software analysis with machine learning.

With this in mind, our goal is to investigate the answer to the following research questions:

- **RQ1** What are the most common approaches for representing source code in graph format?
- **RQ2** What are the applications of machine learning on graphs for software analysis?

The rest of the paper is organized as follows. Section II describes our methodology. Section III outlines common building blocks for representing software in the form of graphs. Section IV overviews tasks that are solved using models for machine learning on graphs. Section V discusses answers to our research questions.

978-1-7281-9255-0/20/$31.00 ©2020 IEEE
DOI 10.1109/ICS51289.2020.00109

TABLE I
KEYWORDS FOR SEARCH QUERIES

Group 1	Group 2
software analysis	machine learning applications
source code analysis	TreeRNN TreeLSTM
source code graph	graph neural network
source code data flow	graph convolutional networks
source code control flow	GCN GNN

II. METHODOLOGY

A. Search Strategy

The collection of the literature is a tedious process. We decided to take advantage of the academic search engine Google Scholar. Our choice of a single academic search engine was motivated by the study performed by Gusenbauer [24]. Our search strategy was composed of the following steps:

- Query articles
- Identify relevant results
- Identify more relevant articles by
  - Following links to cited articles
  - Following links to articles that cite the current article
  - Follow links via "Related articles", implemented in Google Scholar
  - Repeat for newly found relevant articles
- Modify search query and repeat

Graphs are structured representations of the source code (in contrast with token-based representations). The use of machine learning in conjunctions with graphs is relatively novel. Our goal was to find articles about applications of statistical learning approaches on graphs to source code analysis. To achieve this, we constructed a set of queries from two groups of keywords presented in Table I.

The relevance of articles was identified at the moment of search using the title and the abstract. Articles that presented interest were selected as the starting points of an in-depth search using citations. Overall, 108 articles were collected and studied. Out of those 108 articles, 43 were deemed relevant to answer the research questions stated above.

B. Inclusion Criteria

In this paper, we are going to focus on research that utilizes one of several graph-like representations for source code. Graph representations are different from conventional token-based representations because they capture complex relationships between source code elements. Besides graphs themselves, we consider tree-like representations as well. We also track, which machine learning tasks researchers try to address, and what kind of information they extracted from the source code.

The goals outlined above resulted in the following inclusion criteria:

- Papers should present techniques for source code analysis using machine learning;
- The source code should be represented in a graph-like structure;
- The machine learning problem should represent an end task (excludes intermediate tasks like learning embeddings);
- Machine learning approaches should use one of the methods for processing graph information, such as Graph Neural Networks or Tree Neural Networks, or other related approaches;
- We focused on studying the recent advancements, potentially excluding some pioneering work, especially if it was reported to be inferior to the latest approaches.

These inclusion criteria resulted in 43 papers that were considered relevant. The distribution of papers over different publication sites (including conferences and journals) is presented in Figure 1. Our

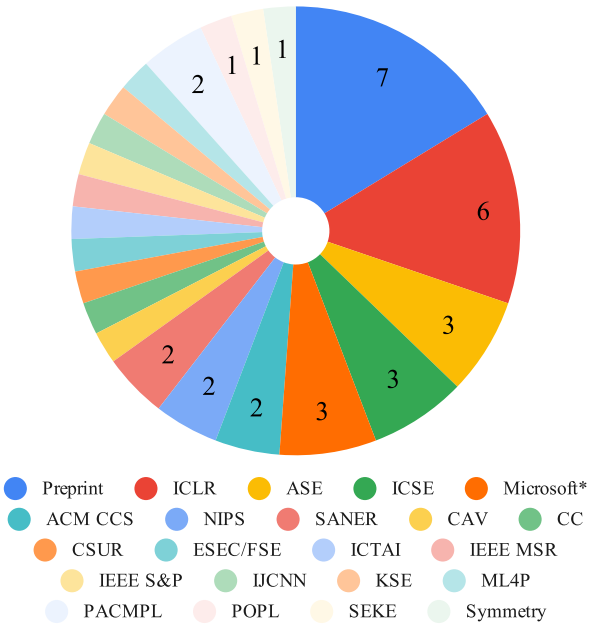

Fig. 1. Distribution of paper demographics. Label Microsoft refers to papers that were published by Microsoft in OpenReview. Preprint refers to papers that are published in arXiv and have and were cited by others.

citation record began in 2014, and most of the papers were published in the year 2019. When considering the distribution of publication sites, six of the papers were published in ICLR. Seven papers were found only as preprints, but they also have already been cited by other researchers. There are three papers published by Microsoft that were not published at any conference or journal. However, these three papers were submitted to openreview.net and had numerous citations. The majority of the papers were published in different journals and conferences.

C. Data Extraction

We are interested in extracting specific information from the papers. In order to extract this information, we target related sections of those papers. The list of insights that we try to extract is:

- **Data representation format** used for representing the source code before passing it to the learning algorithm. This includes the specific information that was extracted from the source code in order to represent a program in the format of nodes and edges.
- **Machine learning task** that the paper tries to solve. Machine learning approaches recently adopted different input data formats that are different from convectional vector representations. It is a known fact that the correct perspective on the input data is quite important for a machine learning model to learn anything useful. The overview of machine learning tasks on graphs for source code will allow us to understand the directions that are currently considered and are the most promising to address.

For some papers, we were not able to identify the dataset because it was not publicly available. Some works do not consider any specific programming languages, and therefore do not present this information. Not all researchers provide information about the tools they used for processing the source code. Some of them implemented their custom software for this purpose. We extracted relevant information to the best of our effort and presented our findings in the following sections.

III. Graph Building Blocks

In this section we overview elements that are often used to construct a graph-based representation for source code. The list of commonly used elements and relevant papers are summarized in Table II. This section tries to answer RQ1.

A. Common Source Code Representations

In the area of source code analysis, structured representations are consistently utilized for decades. Such representations are useful for program analysis and optimization. We attribute Abstract Syntax Trees (AST), control-flow and data-flow graphs to the conventional representations of source code. It is important to note that some works presented here do not use the exact form of flow graphs, but merely augment other representations with data and control flow edges.

AST representation is obtained directly from the source code of a program. Nodes in an AST represent source code constructs, and edges represent dependencies. This type of representation is easiest to obtain. ASTs can have many elements with repeated functionality, such as for and while loops, and inherently very language-specific. Additionally, they usually have the most nodes.

Control Flow graphs are constructed from ASTs. Nodes in such graphs are statements and edges represent the transfer of control between statements. Usually, procedures have entry and exit nodes. These graphs are simpler and can provide more language-agnostic view of the program.

Data-Flow graphs represents data dependency between statements. They usually contains no conditionals, and, as a consequence, no control-flow statements. At the same time, such representation is harder to obtain, and sometimes it requires run-time analysis of a program.

Flow graphs are usually constructed for a single procedure or function. **Inter-Procedural Flow** graphs connect the flow graphs of disjoint procedures into a single flow graph through enter and exit nodes, that are defined for every procedure. These representations are not as easy to acquire for any language and researchers sometimes utilize compiler level information, as in [18].

B. Enriching Graph

Besides traditional representations of the source code, some researchers use extra edges in the graph to enrich the representation with potentially useful information. We categorized different types of edges that are often used besides standard AST, control, or data flow edges into several groups listed below. We are not aware of any work that evaluated the importance of each particular type of additional edges.

Type Information edges are used to show that the current node uses or belongs to a certain type. This can be implemented by connecting a variable node with a type node [8] and with supertypes for the current type [41, 42].

Order edges point to a node in the graph that is lexically mentioned after the current node in the source code. In this case, nodes are usually variables. This type of edges is usually used to incorporate token level representation into the graph.

Next Sibling edges connect the current node to the node for the next operand of the current statement. For example, the operands of the binary expression would have a connection from the first operand to the second.

Usage edges are used to connect the current node with the location where it was last written, read, or simply mentioned. In [8] this type of edge was used point to the next usage of the object represented by this node.

Property edges connect nodes in the graph with special property nodes. These edges can be used to point to the names of the variables [15]. Some use node attributes instead of property edges, where a node has a specific type, and this information is utilized during neural network training [48].

TABLE II
Summary of common relationships in graphs and papers that use them

Graph Relationships	Papers
AST	[21, 12, 3, 15], [19, 31, 2, 9], [45, 16, 49, 48], [46, 4, 7, 26], [39, 34, 20, 40], [27, 28, 42]
Control Flow	[41, 15, 5, 31], [45, 49, 7, 39], [38, 44, 40, 27]
Data Flow	[10, 15, 5, 45], [49, 7, 40, 27], [28]
Inter-procedural flow	[18, 27, 41]
Type Information	[8, 41, 42, 6]
Order	[8, 15, 19, 21], [2, 49]
Next Sibling	[8]
Usage	[8, 15, 21, 2]
Computed From	[15]
Property	[15, 21, 49, 44]
Backward	[2]
Function Call	[31, 28]

Backward edges are simply directed in the opposite direction of the primary edges. This allows for information to flow backward in the graph. This type of edge can be useful when message passing techniques are used.

Function Call graph shows the dependencies between functions in a program. An edge in a function call graph exists if one function calls another during its execution. These types of edges were used in some papers to augment graph representation.

IV. Machine learning tasks for source code

In this section, we describe tasks for source code that researchers try to solve using machine learning on graphs. This section tries to answer RQ2.

Graphs are versatile representations and can be utilized in many applications. The most intuitive tasks that can be solved with the help of graphs are the ones that require node classification or link prediction. However, these types of applications are rather straightforward. Some applications listed below involve the analysis of the entire graph - a task that is difficult to solve with classical techniques. The overview of the tasks and related papers is presented in Table III.

Program element naming. The goal of this task is to predict the name of either a variable or a function. This can be utilized in different scenarios. The simplest application is suggesting better variable names. Such a problem was addressed in [2]. Authors represented C# code in the form of a graph that is based on Abstract Syntax Trees (AST), lexical information from the source code, and additional data-flow information. Their work demonstrated the ability of Graph Neural Networks to learn variable naming patterns. Variable naming is especially useful when analyzing obfuscated code [6]. The authors applied CRF to predict variable names. Their analysis showed that a valid function name is a strong predictor for variable names that are used inside this function body. Another CRF model was used in [33] to predict variable names in JavaScript code. Alternatively, better variable names are needed to make decompiled code more readable [26]. In this paper, the authors tried to apply a statistical model to identify variable names of decompiled C code. They use GGNN for encoding structural information and LSTM to encode token-level information. Their graph was built from AST with additional edges to reflect variable usage.

Besides naming variables, this approach can be used for suggesting names for functions. Function naming is often treated as a proxy task to evaluate how well a statistical model can process the body of a function and extract the most important aspect. This task can be viewed as an extreme form of source code summarization. In [3] a path-based approach to program modeling was adapted. A collection of paths was extracted from AST, and these paths were used to learn function representations. These representations showed to be useful in function name prediction task. In [15] a GGNN was applied to learn the same task. In both works, authors choose to work with Java code.

Type annotations. Some programming languages enable type inferencing during compilation. This is very helpful to a programmer, but the inferencing can sometimes fail. In this situation, a programmer has to specify the type manually. A statistical model can be used to suggest the most appropriate type to the developer. In research, this task was mostly used with languages such as JavaScript [33]. However, it can also be useful for Python, where type annotations are to help the developer to understand unfamiliar code and assist the development process.

Bug and vulnerability detection. Bug detection is one of the most exciting applications of machine learning for source code analysis. This task allows for discovering the issues that are not detectable by conventional methods. One of the tasks is variable misuse detection [2]. Other examples include the classification of the program's correctness. In [27], a combination of path-based program representation and Node2vec was used to classify whether a code snippet contains bugs or not. In [16], TreeLSTM model was used. Their model relied solely on the AST of the program. Other approaches decided to include additional information in program representation besides child-parent relationships present in AST. Additional edges that represent control or data flow are usually added. When a program graph is available, algorithms like GCN or GGNN are usually applied to classify program correctness [38, 49, 41]. In [19], a more sophisticated objective was pursued that would allow automatically fixing problems in code. The commit history from GitHub was used to prepare the data for training this model. In [25], a model trained on a graph constructed from program binaries was able to detect unknown malware with fewer misses than the most popular antivirus software.

Language Modeling. The most common application for language modeling is source code autocompletion. Moreover, a proper language model describes the process of code generation and potentially can be used to evaluate syntactic correctness or to suggest a correction to syntactically incorrect source code. The problem of code modeling can be approached in different ways. In [4], the source code is represented as a collection of AST paths. They adopt the task of predicting new nodes in the AST. As the tree grows, new paths are incorporated into the prediction process. In [46], the authors used a model called two-dimensional LSTM to predict new nodes in the AST. In [31], the Bayesian approach was used to learn how to edit current AST to insert new nodes.

Program Classification. This task is notable because it requires to perform the classification of the entire graph and not of individual nodes. Program classification was used in a number of different scenarios. In [28, 48, 5], researchers try to address the problem of source code classification. They approach this problem differently. [28] uses GGNN with attention pooling to create a code embedding, that can later be used for classification. [48] creates a more complex model where they split AST into smaller subtrees and then encode the entire program as a sequence of subtrees using LSTM. In [5], the skip-gram-like approach is used, which, nevertheless, yields competitive results, surpassing baselines in quality.

Another area where program classification can be applied is compiler optimizations. In [7, 5], embeddings that were learned on the graph-based representation of the source code were used to classify the kernel placement for computational frameworks such as OpenCL.

All bug and vulnerability detection approaches can also be considered here because they can be reduced to classification problems.

Summarization. The task of source code summarization requires solving two difficult tasks jointly: understanding the purpose of a program and generating a textual description of this program. In practice, this task is used to generate a short textual description for a snippet of code or a function. When applying to functions, it is used to generate function docstring based on source code.

In [32], a tree-based program representation was used in conjunction with a semantic ontology of libraries to create semantic flow digram of python data science programs. Their approach allows converting an API based flow diagram to a more high-level description of a program.

In [21], a GNN model for learning on a graph was used as an encoder, and LSTM - as a decoder. Such a combination allowed creating a model that generates textual descriptions of functions. In [34], a similar approach was used. However, TreeLSTM was used instead of GNN. Their method achieved competitive results when compared with baselines.

Program Translation. With the constant change on the horizon of programming languages, some code becomes obsolete. A problem of migrating the old code to a new programming language is common. The latest research allows using machine translation tools, not in a classical seq2seq setting, but when one has a tree-like input and output. Since the training data is hard to acquire, some use known libraries that implement identical functionality in different programming languages. In [12], a couple of approaches for collecting parallel data for source code translation were described. First, the authors generated training data automatically by taking programs written in CoffeeScript and generating JavaScript counterparts. Because CoffeeScript is compiled to JavaScript by design, the data is guaranteed to be correct. Second, they mined repositories of open source projects that have API implemented in different languages. Then, they used TreeLSTM as the encoder and decoder to train a translation model.

A similar approach was taken in [20]. They used similar architecture to train a model that can translate between different grammars. They demonstrated that a model with a tree-based decoder surpasses a classical sequential decoder.

In [11], the authors decide to apply a more complex architecture. First, they created an AST from the source code and processes leaf nodes with BiGRU. After, embeddings for each token were used to perform aggregation with TreeLSTM. Their decoder is a model for generating trees. Additionally, they do not exactly use their model to translate between languages, but rather for suggesting future changes in the program.

Duplicate and similarity detection. Another problem that is notoriously hard to solve with classical approaches is duplicate detection. The issue is that the same algorithm can be implemented in different ways. If a proper clone detection tool would exist, it could be used to detect identical subprograms in a big project and suggest implementing a dedicated function. More detailed research about clone detection for code with the help of deep learning (using different code representations) was done in [35].

Graph-based models were used in the area of clone detection as well. In [44], an approach motivated by structure2vec was applied. The idea is to create embeddings for two programs and then compare their similarity. Similar programs should be close together in the embeddings space. The same idea is used for other clone detection approaches. In [9], a siamese network with TreeLSTM encoder was used to create embeddings for programs. And in [40], a GGNN encoder was used.

Moreover, if a gap between programming languages and natural language is bridged, search engines for source code become a possibility. In [39], researchers tried to implement a source code search engine that takes multimodal representations of code as an input. The first modality can take source code tokens and the textual description. The second modality can be encoded using AST. Third

TABLE III
SUMMARY OF TASKS THAT CAN BE SOLVED WITH THE HELP OF MACHINE LEARNING ON GRAPHS

ML Task	Papers
Program element Naming	[2, 15, 33, 6, 26, 3]
Type annotations	[33]
Bug and vulnerability detection	[2, 19, 41, 16], [25, 38, 27, 45, 49]
Language modeling	[31, 46, 4]
Classification	[5, 49, 16, 48, 7, 38, 28]
Summarization	[21, 32, 34, 3]
Program Translation	[12, 20, 11]
Similarity search	[9, 48, 39, 44, 40]

modality - control-flow graph. The approach demonstrated a high retrieval score.

V. DISCUSSION

Software development is a complex process that benefits from computer aid. Modern IDEs provide some level of analysis and allow detecting simple problems on the program development stage. However, not all problems can be detected this way. Machine learning approaches can increase the detection rate by allowing some level of false alarms. Moreover, learning approaches can benefit from the complex structural representation of source code, such as graphs.

In this study, we tried to investigate recent achievements in the area of application of machine learning on graphs to software analysis. Let us discuss our findings through the perspective of our initial research questions:

- **RQ1** What are the most common approaches for representing source code in graph format?
- **RQ2** What are the applications of machine learning on graphs for software analysis?

The first research question is related to the source code representation format. From Table II, it is possible to see the distribution of relationships that are commonly used for software representation. Such representations as AST, CFG, and DFG have been common for a significant period of time. It is only natural to see these representations to get wide adoption in graph-based machine learning. However, we can also see that these representations were designed for efficient program representation and can benefit from additional relationships. The experience of [39] showed that a multi-modal representation of code is beneficial for search applications. We assume that search is representative of a more common problem of understanding a program's purpose.

The list of tasks that are addressed by papers in this study is presented in Table III. In general, these tasks are not unique to software analysis with machine learning on graphs. The main difference that these new graph approaches provide is a more natural way of representing software. Previous approaches tend to use more established techniques from NLP where the text is modeled as a sequence.

It is not exactly evident at the current stage, whether graph-based representations are ultimately better. The evaluation in the discussed papers, most of the time, considered either similar graph-based approaches or simply did not make a vigorous comparison with other existing techniques. Nevertheless, works like [4, 34, 41] demonstrate that graph-based models outperform traditional transformer networks that are often considered close to the state-of-the-art in NLP.

We can also see that many pieces of research adopted the fusion of AST edges, control and data flow information, and other lexical attributes of source code, such as order of tokens, last use in the source code, next use, etc. Such fusion of representations seems to be the most promising.

REFERENCES

[1] Miltiadis Allamanis, Earl T Barr, Premkumar Devanbu, and Charles Sutton. A survey of machine learning for big code and naturalness. *ACM Computing Surveys (CSUR)*, 51(4):1–37, 2018.

[2] Miltiadis Allamanis, Marc Brockschmidt, and Mahmoud Khademi. Learning to represent programs with graphs. In *International Conference on Learning Representations (ICLR)*, 2018.

[3] Uri Alon, Shaked Brody, Omer Levy, and Eran Yahav. code2seq: Generating sequences from structured representations of code. In *International Conference on Learning Representations*, 2019.

[4] Uri Alon, Roy Sadaka, Omer Levy, and Eran Yahav. Structural language models for any-code generation. *arXiv preprint arXiv:1910.00577*, 2019.

[5] Tal Ben-Nun, Alice Shoshana Jakobovits, and Torsten Hoefler. Neural code comprehension: A learnable representation of code semantics. *Advances in Neural Information Processing Systems*, 2018-Decem(NeurIPS):3585–3597, 2018.

[6] Benjamin Bichsel, Veselin Raychev, Petar Tsankov, and Martin Vechev. Statistical deobfuscation of Android applications. *Proceedings of the ACM Conference on Computer and Communications Security*, 24-28-Octo:343–355, 2016.

[7] Alexander Brauckmann, Andrés Goens, Sebastian Ertel, and Jeronimo Castrillon. Compiler-based graph representations for deep learning models of code. *Proceedings of the 29th International Conference on Compiler Construction*, pages 201–211, 2020.

[8] Marc Brockschmidt, Miltiadis Allamanis, Alexander L. Gaunt, and Oleksandr Polozov. Generative code modeling with graphs. In *International Conference on Learning Representations (ICLR)*, 2019.

[9] Lutz Buch and Artur Andrzejak. Learning-Based Recursive Aggregation of Abstract Syntax Trees for Code Clone Detection. *SANER 2019 - Proceedings of the 2019 IEEE 26th International Conference on Software Analysis, Evolution, and Reengineering*, pages 95–104, 2019.

[10] Kwonsoo Chae, Hakjoo Oh, Kihong Heo, and Hongseok Yang. Automatically generating features for learning program analysis heuristics for C-like languages. *Proceedings of the ACM on Programming Languages*, 1(OOPSLA):1–25, 2017.

[11] Saikat Chakraborty, Miltiadis Allamanis, and Baishakhi Ray. Tree2Tree Neural Translation Model for Learning Source Code Changes. pages 1–12, 2018.

[12] Xinyun Chen, Chang Liu, and Dawn Song. Tree-to-tree neural networks for program translation. *6th International Conference on Learning Representations, ICLR 2018 - Workshop Track Proceedings*, (NeurIPS), 2018.

[13] Zimin Chen and Martin Monperrus. A Literature Study of Embeddings on Source Code. pages 1–8, 2019.

[14] Giulio Concas, Michele Marchesi, Sandro Pinna, and Nicola Serra. Power-laws in a large object-oriented software system. *IEEE Transactions on Software Engineering*, 33(10):687–708, 2007.

[15] Milan Cvitkovic, Badal Singh, and Anima Anandkumar. Deep learning on code with an unbounded vocabulary. In *Machine Learning for Programming (ML4P) Workshop at Federated Logic Conference (FLoC)*, 2018.

[16] Hoa Khanh Dam, Trang Pham, Shien Wee Ng, Truyen Tran, John Grundy, Aditya Ghose, Taeksu Kim, and Chul Joo Kim. Lessons learned from using a deep tree-based model for software defect prediction in practice. *IEEE International Working Conference on Mining Software Repositories*, 2019-May:46–57, 2019.

[17] Alessandro PS De Moura, Ying-Cheng Lai, and Adilson E

Motter. Signatures of small-world and scale-free properties in large computer programs. *Physical Review E*, 68(1):017102, 2003.

[18] Daniel DeFreez, Aditya V. Thakur, and Cindy Rubio-Gonzalez. Path-based function embedding and its application to error-handling specification mining. *ESEC/FSE 2018 - Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 423–433, 2018.

[19] Elizabeth Dinella, Hanjun Dai, Google Brain, Ziyang Li, Mayur Naik, Le Song, Georgia Tech, and Ke Wang. Hoppity: Learning Graph Transformations To Detect and Fix Bugs in Programs. *Iclr 2020*, pages 1–17, 2020.

[20] Mehdi Drissi, Olivia Watkins, Aditya Khant, Vivaswat Ojha, Pedro Sandoval, Rakia Segev, Eric Weiner, and Robert Keller. Program Language Translation Using a Grammar-Driven Tree-to-Tree Model. 2, 2018.

[21] Patrick Fernandes, Miltiadis Allamanis, Marc Brockschmidt, and United Kingdom. Structured Neural Summarization. pages 1–18, 2019.

[22] Darius Foo, Jason Yeo, Hao Xiao, and Asankhaya Sharma. The dynamics of software composition analysis. *arXiv preprint arXiv:1909.00973*, 2019.

[23] Anjana Gosain and Ganga Sharma. Static analysis: A survey of techniques and tools. In *Intelligent Computing and Applications*, pages 581–591. Springer, 2015.

[24] Michael Gusenbauer. Google scholar to overshadow them all? comparing the sizes of 12 academic search engines and bibliographic databases. *Scientometrics*, 118(1):177–214, 2019.

[25] Nguyen Viet Hung, Pham Ngoc Dung, Tran Nguyen Ngoc, Vu Dinh Phai, and Qi Shi. Malware detection based on directed multi-edge dataflow graph representation and convolutional neural network. *Proceedings of 2019 11th International Conference on Knowledge and Systems Engineering, KSE 2019*, pages 1–5, 2019.

[26] Jeremy Lacomis, Pengcheng Yin, Edward Schwartz, Miltiadis Allamanis, Claire Le Goues, Graham Neubig, and Bogdan Vasilescu. DIRE: A neural approach to decompiled identifier naming. *Proceedings - 2019 34th IEEE/ACM International Conference on Automated Software Engineering, ASE 2019*, pages 628–639, 2019.

[27] Yi Li, Shaohua Wang, Tien N Nguyen, and Son Van Nguyen. 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, 2019.

[28] Mingming Lu, Yan Liu, Haifeng Li, Dingwu Tan, Xiaoxian He, Wenjie Bi, and Wendbo Li. Hyperbolic function embedding: Learning hierarchical representation for functions of source code in hyperbolic space. *Symmetry*, 11(2), 2019.

[29] Christopher R Myers. Software systems as complex networks: Structure, function, and evolvability of software collaboration graphs. *Physical Review E*, 68(4):046116, 2003.

[30] Graham Neubig. Survey of Methods to Generate Natural Language from Source Code. (2013):2013–2016, 2014.

[31] Anh Tuan Nguyen and Tien N Nguyen. Graph-based statistical language model for code. In *Proceedings - International Conference on Software Engineering*, volume 1, pages 858–868, 2015.

[32] Evan Patterson, Ioana Baldini, Aleksandra Mojsilovic, and Kush R. Varshney. Teaching machines to understand data science code by semantic enrichment of dataflow graphs. 2018.

[33] Veselin Raychev, Martin Vechev, and Andreas Krause. Predicting program properties from" big code". In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 111–124, 2015.

[34] Yusuke Shido, Yasuaki Kobayashi, Akihiro Yamamoto, Atsushi Miyamoto, and Tadayuki Matsumura. Automatic Source Code Summarization with Extended Tree-LSTM. *Proceedings of the International Joint Conference on Neural Networks*, 2019-July:1–14, 2019.

[35] Michele Tufano, Cody Watson, Gabriele Bavota, Massimiliano Di Penta, Martin White, and Denys Poshyvanyk. Deep learning similarities from different representations of source code. *Proceedings - International Conference on Software Engineering*, pages 542–553, 2018.

[36] Sergi Valverde and Ricard Sole. Hierarchical small-worlds in software architecture. *Dynamics of Continuous Discrete and Impulsive Systems: Series B; Applications and Algorithms*, 14:1, 01 2007.

[37] Sergi Valverde and Ricard V Solé. Network motifs in computational graphs: A case study in software architecture. *Physical Review E*, 72(2):026107, 2005.

[38] Anh Viet Phan, Minh Le Nguyen, and Lam Thu Bui. Convolutional neural networks over control flow graphs for software defect prediction. *Proceedings - International Conference on Tools with Artificial Intelligence, ICTAI*, 2017-Novem:45–52, 2018.

[39] Yao Wan, Jingdong Shu, Yulei Sui, Guandong Xu, Zhou Zhao, Jian Wu, and Philip Yu. Multi-modal attention network learning for semantic source code retrieval. *Proceedings - 2019 34th IEEE/ACM International Conference on Automated Software Engineering, ASE 2019*, pages 13–25, 2019.

[40] Wenhan Wang, Ge Li, Bo Ma, Xin Xia, and Zhi Jin. Detecting code clones with graph neural network and flow-augmented abstract syntax tree. In *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 261–271. IEEE, 2020.

[41] Yu Wang, Fengjuan Gao, Linzhang Wang, and Ke Wang. Learning a Static Bug Finder from Data. 1(January), 2019.

[42] Jiayi Wei, M Goyal, Greg Durrett, and Isil Dillig. Lambdanet: Probabilistic type inference using graph neural networks. In *ICLR 2020*, 2020.

[43] Martin White, Michele Tufano, Christopher Vendome, and Denys Poshyvanyk. Deep learning code fragments for code clone detection. *ASE 2016 - Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, pages 87–98, 2016.

[44] Xiaojun Xu, Chang Liu, Qian Feng, Heng Yin, Le Song, and Dawn Song. Neural network-based graph embedding for cross-platform binary code similarity detection. *Proceedings of the ACM Conference on Computer and Communications Security*, pages 363–376, 2017.

[45] Fabian Yamaguchi, Nico Golde, Daniel Arp, and Konrad Rieck. Modeling and discovering vulnerabilities with code property graphs. *Proceedings - IEEE Symposium on Security and Privacy*, pages 590–604, 2014.

[46] Yixiao Yang, Xiang Chen, and Jiaguang Sun. Improve language modelling for code completion by tree language model with tree encoding of context. *Proceedings of the International Conference on Software Engineering and Knowledge Engineering, SEKE*, 2019-July:675–680, 2019.

[47] Stephen S. Yau and Paul C. Grabow. A Model for Representing Programs Using Hierarchical Graphs. *IEEE Transactions on Software Engineering*, SE-7(6):556–574, 1981.

[48] Jian Zhang, Xu Wang, Hongyu Zhang, Hailong Sun, Kaixuan Wang, and Xudong Liu. A Novel Neural Source Code Representation Based on Abstract Syntax Tree. *Proceedings - International Conference on Software Engineering*, 2019-May:783–794, 2019.

[49] Yaqin Zhou, Shangqing Liu, Jingkai Siow, Xiaoning Du, and Yang Liu. Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks. In *Advances in Neural Information Processing Systems*, pages 10197–10207, 2019.