Chapter 23
Graph Neural Networks in Software Mining

Collin McMillan

Abstract Software Mining encompasses a broad range of tasks involving software, such as finding the location of a bug in the source code of a program, generating natural language descriptions of software behavior, and detecting when two programs do basically the same thing. Software tends to have an extremely well-defined structure, due to the linguistic confines of source code and the need for programmers to maintain readability and compatibility when working on large teams. A tradition of graph-based representations of software has therefore proliferated. Meanwhile, advances in software repository maintenance have recently helped create very large datasets of source code. The result is fertile ground for Graph Neural Network representations of software to facilitate a plethora of software mining tasks. This chapter will provide a brief history of these representations, describe typical software mining tasks that benefit from GNNs, demonstrate one of these tasks in detail, and explain the benefits that GNNs can provide. Caveats and recommendations will also be discussed.

23.1 Introduction

Software Mining is broadly defined as any task that seeks to solve a software engineering problem by analyzing the myriad artifacts in projects and their connections (Hassan and Xie, 2010; Kagdi et al, 2007; Zimmermann et al, 2005). Consider the task of writing documentation. A human performing this task may gain comprehension of the software by reading the source code and understanding how different parts of the code interact. Then he or she may write documentation explaining the behavior of the system based on that comprehension. Likewise, if a machine is to automate writing that documentation, the machine must also analyze the software in order to comprehend it. This analysis is often called "Software Mining."

Collin McMillan
Department of Computer Science, University of Notre Dame, e-mail: cmc@nd.edu

While human comprehension of software is a cognitive process that occurs naturally as engineers read and interact with that software (Letovsky, 1987; Maalej et al, 2014), machine comprehension must be formally defined and quantifiable. Typically this boils down to a vectorized representation of each software artifact. For example, each identifier name in a function may be assigned an, e.g., 100-length vector denoting its position in a word embedding space. Then the function may be the average of those vectors for the identifier names it contains. Or it may be the output of a recurrent neural network given those identifier name vectors, or perhaps only the names that occur in particular locations. The point is that machine comprehension of software is often quantifiable as a vectorized representation of the artifacts composing that software.

Evidence is accumulating that Graph Neural Networks are an effective means to obtain these vectorized representations and thus improve machine comprehension of software. There is a long tradition in the Software Engineering research literature of treating software as a graph. Control flow graphs, call graphs, abstract syntax trees, execution path graphs, and many others are frequently the output of both static and dynamic analysis. Meanwhile, advances in software repository management have enabled the creation of datasets covering billions of lines of code. The result is fertile ground for GNNs.

This chapter covers the history and state-of-the-art in representing software as a graph for GNNs, followed by a high-level discussion of current approaches, a detailed look at a specific approach, and caveats for future researchers.

23.2 Modeling Software as a Graph

Software is a high-value target for GNNs partly because software tends to be very highly structured as a graph or set of graphs. Different software mining tasks may take advantage of different graph structures from software. Graph representations of software go far beyond any specific software mining task. Graph representations are baked into the way compilers convert source code into machine code (e.g., parse trees). They are used during linking and dependency resolution (e.g., program dependence graphs). And they have long the basis for many visualization and support tools to help programmers understand large software projects (Gema et al, 2020; Ottenstein and Ottenstein, 1984; Silva, 2012).

When considering how to make use of these different graph structures in software, basically the questions one must ask are: "what are the nodes?" and "what are the edges?" These questions take two forms in software engineering research: a macro- and a micro-level representation. The macro-level representation tends to concern connections among large software artifacts, such as a graph in which every source code file is a node and every dependency among the files is an edge. The micro-level representation, in contrast, tends to include small details, such as a graph in which every token in a function is a node, and every edge is a syntactic link between the nodes, such as are often extracted from an Abstract Syntax Tree.

This section compares and contrasts these representations as they relate to using GNNs for Software Mining tasks.

23.2.1 Macro versus Micro Representations

Graph structures in software may be broadly classified as either macro- or micro-level. In theory, the distinction is superfluous because a micro-level representation may be scaled up to arbitrary size. For example, an entire large program may be represented as one large abstract syntax tree. But in practice, time and space constraints necessitate a separation of macro- and micro-level representations. In a recent collection of Java programs (LeClair and McMillan, 2019), the average number of nodes in the AST of a function is over 120, with at least one edge per node. The average number of functions per program is over 1800, and there are over 28,000 programs in the dataset. The reality is that a micro-level representation of an entire program is often not feasible, so a macro-level representation is introduced to capture the "big picture."

23.2.1.1 Macro-level Representations

A macro-level graph representation of software captures the high-level structure and intent behind a program while avoiding a deep dive into details required to implement that intent. Inspiration for macro-level representations is often drawn from software design documents, such as those formally defined via UML (Braude and Bernstein, 2016; Horton, 1992). An example is a class diagram for an object-oriented program. Each class is a node in the graph. Edges in the graph may variously be dependency, inheritance, realization, composition, among others. Nodes may also have attributes that refer to the member variables and methods of a class.

In practice, selecting a macro-level representation for a software mining task using GNNs tends to be severely constrained by what can actually be obtained from the dataset. Often this constraint precludes the use of behavior-based graphs such as use case diagrams, because proper use case diagrams are rare, and those that are available are usually not in a consistent format. For example, because some engineers might follow different conventions, or only provide these diagrams informally. Software repositories tend to be replete with source code but lack documentation, especially design documentation (Kalliamvakou et al, 2014).

Therefore, by far, the most popular macro-level graph representations tend to be ones that can be extracted directly from source code. A decision often arises related to the degree of granularity, which usually is a choice between packages/directories, classes/files, or methods/functions. The class diagram is relatively easy to locate every class in a software project, then analyze each class to find their dependencies, inheritances, and etc. Package diagrams are similar, having the advantage of quickly providing a very high level view of a program – even large projects may only have a

few dozen packages. But a very popular alternative is a function/method call graph, in which each function in a program is a node and each call relationship from one function to another is a directed edge between two nodes. Call graphs are popular within Software Engineering literature because they are relatively easy to extract while giving enough detail for a strong macro-level view of a program without overwhelming data sizes (recall a typical program has around 1800 functions (LeClair and McMillan, 2019)).

23.2.1.2 Micro-level Representations

A micro-level representation describes a portion of the software in great detail. Micro-level representations have been the focus of a majority of research using GNNs for software mining. Allamanis et al (2018b) describe one approach, pointing out that the "backbone of a program graph is the program's abstract syntax tree." However, as mentioned above, it is often not feasible to build a model relying on the entire AST of an entire program. Instead, a typical practice is to generate the AST for small portions of code, such as individual functions. Each function is treated as a graph, independent of all other functions.

The benefit of treating each function as a separate graph is that a GNN model can be trained on each independently. A prediction model of nearly any kind will require independent, self-contained examples. There will be some context about which an output prediction is generated (or against which a sample prediction is used for training). By treating each function as an independent graph, a GNN can be trained using each function as the context. This is a tidy solution in software mining for two reasons. First, many tasks in software mining involve predictions about specific functions, such as whether that function is likely to contain a fault (see the next section). Second, graphs of functions derived from the AST exhibit a community structure. In a typical function, there are many connections among nodes inside the function, but relatively few connections from nodes inside the function to nodes outside the function – the variables, conditionals, loops, and etc., in the code of a function interact closely with each other, while must less frequently referring to something outside the function such as the use of a global variable or call.

One may concoct any number of micro-level representations of software, based on different tokens in the source code and relationships of those tokens. For example, control flow relationships have occasionally been highlighted as often more valuable for comprehension than data dependencies (Dearman et al, 2005; Ko et al, 2006). At other times, method invocations (Mcmillan et al, 2013; Sillito et al, 2008) or signatures (Roehm et al, 2012) are proposed as providing superior information for different software mining tasks. Yet the pattern is that a micro-level representation is generated for many small portions of a software system, and these portions are treated as independent of each other. A GNN can take advantage of these micro-level representations by learning from each one as a different sample.

23.2.2 Combining the Macro- and Micro-level

Macro- and micro-level representations may be combined. One strategy would be to compute both macro- and micro-level representations independently, then concatenate them into one large context matrix. Such a model may be referred to as "dual encoder" (Chidambaram et al, 2019; Yang et al, 2019h) or "cascading" (Wang et al, 2017h) in that they learn two representations of the same object but at different levels of granularity. An alternative would be to use the output of the micro-level representation to seed the macro-level representation, for example, by learning a representation of each function using the AST and then using it as the initial value for the nodes in a function call graph.

23.3 Relevant Software Mining Tasks

Graph neural networks are becoming a staple of research in software mining tasks. The history of deep learning for software mining tasks is chronicled in several surveys (Allamanis et al, 2018a; Lin et al, 2020b; Semasaba et al, 2020; Song et al, 2019b). Allamanis et al (2018a) cast a particularly wide net and broadly classify software mining tasks that rely on neural networks as either "code generational" or "code representational." This classification is based on a big picture view of the models used for these tasks. In a code generational task, the output of the model is source code. Tasks in this category include automatic program repair (Chen et al, 2019e; Dinella et al, 2020; Wang et al, 2018d; Vasic et al, 2018; Yasunaga and Liang, 2020), code completion (Li et al, 2018a; Raychev et al, 2014), and compiler optimization (Brauckmann et al, 2020). These models tend to be trained with large volumes of code vetted somehow to ensure quality, with the aim of learning norms in code that lead to that quality. Then, during inference, the goal is to bring arbitrary code into closer conformance with those norms. For example, a model may be presented with code containing a bug, and that bug may be repaired by changing the code to be more like the model's predictions (which, it is hoped, represent the norms learned in training).

In contrast to code generational tasks are code representational tasks. These tasks use source code primarily as the input to a neural model during training but have a wide variety of outputs. Tasks in this category include code clone detection (Ain et al, 2019; Li et al, 2017c; White et al, 2016), code search (Chen and Zhou, 2018; Sachdev et al, 2018; Zhang et al, 2019f), type prediction (Pradel et al, 2020), and code summarization (Song et al, 2019b). In models designed to solve these tasks, the goal is usually to create a vectorized representation of code, which is then used for a specific task that may only be tangentially related to the code itself. For instance, for source code search, a neural model may be used to project the source code in a large repository into a vector space. Then a different model is used to project a natural language query into the same vector space. The code nearest to the query in the vector space is considered as the search result for that query. Code clone

detection is similar: code is projected into a vector space, and very nearby code may be considered a clone in that space.

The use of graph neural networks is ballooning in both categories of software mining tasks. In code generational tasks, the focus tends to be on modifications to a program graph such as an AST that bring that graph into closer conformity with the model's expectations. While some approaches focus on code as a sequence (Chen et al, 2019e), the recent trend has been to recommend graph transformations or highlight non-conforming areas of the graph (Dinella et al, 2020; Yasunaga and Liang, 2020). This is useful in code because a recommendation may relate to code elements that are quite far away from each other, such as the declaration of a variable and a use of that variable. In contrast, in code representational tasks, the focus tends to be on creating ever more complex graph representations of code and then using GNN architectures to exploit that complexity. For example, the first GNN-based approaches tended to use only the AST (LeClair et al, 2020), while newer approaches use attention-based GNNs to emphasize the most important edges out of a multitude that can be extracted from code (Zügner et al, 2021). Despite differences in code generational and representational tasks, the trend in both categories has strongly favored GNNs.

Consider the task of code summarization, which exemplifies the trend towards GNNs. Code summarization is the task of writing natural language descriptions of source code. Typically these descriptions are used in documentation for that source code, e.g., JavaDocs. The evolution of this research area is shown in Figure 23.1. The term "code summarization" was coined around 2010, and several years of active research followed using templated and IR-based solutions. Then around 2017, solutions based on neural networks proliferated. At first, these were essentially seq2seq models in which the encoder sequence is the code and decoder sequence is the description. Starting around 2018, the state-of-the-art moved to linearized AST representations. Graph neural networks were proposed around this time as a better solution (Allamanis et al, 2018b), but it would be another year or more for GNN-based approaches to appear in the literature. GNNs are poised to underpin the state-of-the-art. In the next section, we dive into the details of a GNN-based solution, showing why it works and areas of future growth.

23.4 Example Software Mining Task: Source Code Summarization

This section describes source code summarization as an example software mining task that benefits from GNNs. Source code summarization, as mentioned above, is the task of writing natural language descriptions of source code. The input to a code summarization model includes at least the source code being described, though may also include other details about the software project from which the code originates. The output is the natural language description. This task is considered "code repre-

sentational" because it primarily relies on a learned representation of code in order to make predictions about the description.

23.4.1 Primer GNN-based Code Summarization

As a primer towards GNN-based code summarization, consider a technique presented by LeClair et al (2020). This model is intended to be a straightforward application of convolutional GNNs in the vein of graph2seq (Xu et al, 2018c).

	IR	M	T	A	S	G
Haiduc et al (2010)	x					
Sridhara et al (2011)		x	x			
Rastkar et al (2011)	x	x	x			
De Lucia et al (2012)	x					
Panichella et al (2012)	x	x				
Moreno et al (2013)	x		x			
Rastkar and Murphy (2013)	x					
McBurney and McMillan (2014)		x	x			
Rodeghero et al (2014)	x					
Rastkar et al (2014)		x				
Cortés-Coy et al (2014)	x					
Moreno et al (2014)	x					
Oda et al (2015)				x		
Abid et al (2015)		x	x			
Iyer et al (2016)				x		
McBurney et al (2016)	x	x				
Zhang et al (2016a)		x	x			
Rodeghero et al (2017)		x				
Fowkes et al (2017)	x					
Badihi and Heydarnoori (2017)		x	x			
Loyola et al (2017)				x		
Lu et al (2017b)				x		
Jiang et al (2017)				x		
Hu et al (2018c)				x		
Hu et al (2018b)				x	x	
Wan et al (2018)				x	x	
Liang and Zhu (2018)				x	x	
Alon et al (2019a,b)				x	x	
Gao et al (2019b)				x		
LeClair et al (2019)				x	x	
Nie et al (2019)				x	x	
Haque et al (2020)				x	x	
Haldar et al (2020)				x	x	
LeClair et al (2020)				x	x	x
Ahmad et al (2020)				x	x	
Zügner et al (2021)				x	x	x
Liu et al (2021)				x	x	x

Table 23.1: Overview of papers on the topic of source code summarization, from the paper to coin the term "code summarization" in 2010 to the following ten years. Note the evolution from IR/template-based solutions to neural models and now to GNN models. Column *IR* indicates if the approach is based on Information Retrieval. *M* indicates manual features/heuristics. *T* indicates templated natural language. *A* indicates Artificial Intelligence (usually Neural Network) solutions. *S* means structural data such as the AST is used (for AI-based models). *G* means a GNN is the primary means of representing that structural data.

23.4.1.1 Model Input / Output

The input to this technique is a micro-level representation of code: it is just the AST of a single subroutine. The nodes in the graph are all nodes in the GNN, whether they are visible to the programmer or not. The only edge type is the parent-child relationship in the AST. Consider the code and example summaries in Example 23.1 and the AST of this code in Figure 23.1.

The AST in Figure 23.1 is the only input to the model, from which the model must generate an English description. Technically, the AST is srcml (Collard et al, 2011) preprocessed (e.g., splitting identifies such as `sendGuess` into `send` and `guess`) using community standard procedures (LeClair and McMillan, 2019). The reference output description in Example 23.1 is the actual JavaDoc summary written by a human programmer. The summary labeled "gnn ast" is the prediction from this approach. The summary labeled "flat ast" is the output from an immediate predecessor that used an RNN on a linearization of the AST. The only difference between the GNN and flat AST approach is the structure of the encoder; all other model details are identical. Yet, we note that the GNN-based approach matched the reference exactly, while the flat AST approach matched only a few words. Shortly we will analyze this example to provide intuition about why the model performed so well.

23.4.1.2 Model Architecture

The model architecture, as mentioned, is essentially a 2-hop graph2seq design based on a convolutional GNN. While we leave the details of the model to the relevant paper (LeClair et al, 2020), a bird's-eye view of the model is in Figure 23.2.

summaries

reference	sends a guess to the server
ast-attendgru-gnn (LeClair et al, 2020)	sends a guess to the socket
ast-attendgru-flat (LeClair et al, 2019)	attempts to initiate a <UNK> guess

source code

```
public void sendGuess(String guess) {
  if( isConnected() ) {
    gui.statusBarInfo("Querying...", false);
    try {
      os.write( (guess + "\\r\\n").getBytes() );
      os.flush();
    } catch (IOException e) {
      gui.statusBarInfo("Failed to send guess.", true);
      System.err.println("IOException during send guess");
    }
  }
}
```

Example 23.1: The function `sendGuess()` and summary descriptions.

The model input is derived only from a single subroutine being described: the code as a sequence and the AST nodes and edges (Figure 23.2 area A). A word embedding projects tokens in the sequence and nodes in the AST into the same vector space, which is possible because the vocabulary is the same in both the sequence and the node input (area B). A 2-hop convolutional GNN is used to form a vectorized representation of the AST (area C). The output after the second hop is a matrix in which each column is a vector representing a node in the AST. A GRU is then applied to this matrix to capture information about the order in which the nodes appear. Meanwhile, a GRU is also applied to the sequence directly (area D). The decoder is a simple GRU representation of the summary (area H). Attention is applied between the decoder output and the sequence GRU output, as well as the GNN output (area E). The attended matrices are then concatenated into a context matrix (area F) and connected to an output dense layer (area G).

A key feature of the model is the attention between the decoder and the GNN output. The purpose of this attention is to highlight the nodes in the AST that are the most related to the words in the decoder sequence. We will describe below how this attention was made much more effective by the shared word embedding (area B).

23.4.1.3 Experiment

An experiment demonstrated improvement of the GNN model over various baselines, and explored the effects of various model design decisions. The experiment used a dataset of 2.1m Java methods and associated JavaDoc summaries (LeClair et al, 2020). Essentially the conditions were that 80% of the projects in the dataset were assigned for the training set, and 10% each for validation/testing. Duplicates and other defects were removed from the dataset in accordance with community standards (LeClair and McMillan, 2019). The model was trained with methods from the projects in the training set. The training ran for 10 epochs, and the model with the highest validation accuracy was selected for testing. The predictions from the tests were then compared with reference summaries.

Three findings stand out in findings reported by LeClair et al (2020). First, the GNN-based approaches outperform the most-similar baseline (ast-attendgru-flat) by about 1 BLEU point (about a 5% improvement). Since the only difference between the "flat" model and this GNN-based one is the AST encoder portion of the model, the improvement can be attributed to the use of the GNN (as opposed to an RNN) for the AST encoding. Improvement was also observed over two other baselines. The vanilla graph2seq model, which had only the AST and not the sequence encoder (Figure 23.2 area A), was roughly equivalent to the flat AST model in terms of aggregate BLEU score but this score obscures some details of the performance, which we will see in the next section.

The second key finding is that a hop distance of two results in the best overall performance. While models with GNN iterations ranging between one and ten all achieve higher scores than the baselines, the model performs best with two iterations. One explanation is that nodes in the AST are only relevant to each other

within a distance of about two. The AST is a tree, so information is propagated up and down levels of the tree. For two hops, this means information from a node will propagate to its parent in the first hop and then to its grandparent and siblings in the second hop. It is possible that nodes beyond this scope are not that relevant to the model for code summarization. However, another explanation is that the method of aggregating information in each hop is less efficient after two hops – this interpretation would be consistent with findings by Xu et al (2018c) that aggregation procedure is critical to GNN deployment. Either way, the practical advice for model designers is that the optimal number of GNN iterations for this task is not that high.

The third key finding is that the use of the GRU after the GNN layer (Figure 23.2 after area C) improves overall performance. The models labeled with the suffix `+GRU` use this GRU layer, as described in Section 23.4.1.2. The model labeled with the suffix `+dense` calculates attention between the decoder and the output matrix from the GNN. This model did not perform as well. A likely explanation is that source code has not only a tree structure via the AST – it also has an order from start to end. The GRU after the GNN captures this order and seems to result in a better representation of the code for summarization.

23.4.1.4 What benefit did the GNN bring?

A question remains regarding what benefit can be attributed to the use of a GNN. While we and others may observe an improvement in overall BLEU scores when using a GNN (LeClair et al, 2020; Zügner et al, 2021; Liu et al, 2021), a key point is that the GNN contributes *orthogonal* information to the model. This section explores how.

Concentration of Improvement:

The improvement is concentrated among a set of subroutines where the GNN adds significant improvement. It is not the case that the BLEU scores increase marginally for all subroutines – there is a set of subroutines that benefits the most. Consider Figure 23.3. The pie chart divides the test set into subroutines from the experiment describe above into five groups: one group where ast-attendgru-gnn performed the best, one group where ast-attendgru-flat performed the best, one group where they tied, one group for attendgru, and one group for other ties including when all models made the same prediction. For simplicity, we use BLEU-1 scores (BLEU-1 is unigram precision, single words predicted correctly).

What we observe is that each model achieves the highest BLEU-1 score for 20-25% of the subroutines. For about 12% of the subroutines, the AST-based models were tied, meaning that in total over 50% of the subroutines benefited from AST information (GNN plus flat AST models). But there still exists a large set of subroutines where attendgru outperformed all others. However, consider the bar chart in Figure 23.3. The "all" columns show the BLEU-1 score for that approach – note that ast-attendgru-gnn is only marginally higher than others. The "best" columns show the score for the set where that model achieved the highest BLEU-1 score (the

set with that model's name indicated in the pie chart). We observe that the BLEU-1 scores for ast-attendgru-gnn are much higher for this set than others.

Demonstrating Improvement in Example 23.1:

A deeper dive into the subroutine `sendGuess()` from Example 23.1 demonstrates the improvement that a GNN provides. Recall that the ast-attendgru-gnn model calculates attention between each position in the decoder and each node in the output from the GNN (Section 23.4.1.2, Figure 23.2 area E). The result is an *m* x *n* matrix where *m* is the length of the decoder sequence and *n* is the number of nodes (in the implementation, *m*=13 and *n*=100). Thus each position in the attention matrix represents the relevance of an AST node to a word in the output summary. In fact, the attention matrix for ast-attendgru-flat has the same meaning: the models are identical except that ast-attendgru-gnn encodes the AST with a GNN then a GRU, while the flat model uses only the GRU. Comparing the values in these attention matrices provides a useful contrast of the two models because they show the contribution of the AST encoding to the prediction.

The benefit of a GNN becomes apparent in the attention networks in Figure 23.3. Both models have a very similar attention activation to the tokens in the source code sequence (Figures 23.3a and 23.3c). Both models show close attention to position 2 of the code sequence, which is the word "send". This is not surprising considering that "send" appears in the method's name. Yet, ast-attendgru-flat still incorrectly predicts the first word of the summary as "attempts", while ast-attendgru-gnn correctly predicts "sends." The explanation lies in the attention to AST nodes. The flat model focuses on node 37 (Figure 23.3d), which is an expr_stmt node immediately after the `try` block, just before the call to `os.write()`, indicated as area 1 in Figure 23.1. The reason for this focus suggested by the original paper on that model (LeClair et al, 2019) is that the flat AST model tends to learn broadly similar code structure such as "if-block, try-block, call to `os.write()`." Under this explanation, methods in the training set with this if-try-call-catch pattern are associated with the word "attempts."

In contrast, the GNN-based model focuses on position 8, which is the word "send" in the method name, just like in the attention to the code sequence (Figure 23.3b). The result is that the GNN-based AST encoding reinforces the attention paid to this word when predicting the first word of the output. Consider the method's AST in Figure 23.1. Position 8 is the node for "send" indicated at area 2. In a 2-hop GNN, this node will share information with its parent (name), grandparent (function), and sibling (guess). During training, the model learned that words associated with the AST nodes "function" and "name" are likely candidates for the first word of the summary, so the model knows to highlight this word.

In short, the GNN model outperformed because it conveys a lopsided benefit to a particular subset of the subroutines, and a likely reason it conveys this benefit is that it learns to associate AST tokens with particular locations in the code summary.

23.4.2 Directions for Improvement

The view of software as a graph described in Section 23.2 provides two directions for improvement: micro- and macro-level representations. Essentially the choice is whether to attempt to squeeze more information out of the source code being described (micro-level) or to draw upon more information from outside that source code (macro-level). If the aim is to generate summaries of a Java method, then one may learn more information about the details of that method, or one may use information from the classes, packages, dependencies, and etc., around the method. Micro- and macro-level improvements tend to be complementary rather than competitive. Learning more about the macro-level graph information benefits models of micro-level information and visa versa (Haque et al, 2020).

23.4.2.1 Example Micro-level Improvement

Liu et al (2021) present a notable example of an improvement to GNN-based code summarization using a richer micro-level graph representation of software. The essentials of the approach are similar to (LeClair et al, 2020) described above: the input to the model is the source code of a subroutine, and the output is a description of the subroutine. The encoder is based on a GNN, and the input to this GNN is the AST of the subroutine. The nodes in the graph are AST nodes, and the edges are the AST parent-child relationships. However, one novel aspect is that the model also considers other types of edges, namely control flow and data dependencies (these are unified as a Code Property Graph (Yamaguchi et al, 2014)). The benefit to this structure is that nodes in the AST will receive information directly from other relevant parts of the code, rather than only the nodes nearby in the AST.

Consider Figure 23.1 area 3, which is an AST node corresponding to the string variable "guess" in Example 23.1. The ast-attendgru-gnn approach would propagate information from that variable to the parents, grandparents, and siblings (in the two hops configuration). These would be the "name" and "decl" AST nodes. These nodes have locations in the word embedding associated with them, and these nodes also appear in practically every subroutine in the dataset. So, the model will learn how these nodes are used and associate them with what a human would call a variable declaration. The effect in this example is that the model will learn that the word "guess" is a variable name declaration.

The approach by Liu *et al.* improves over ast-attendgru-gnn because it can learn this relationship in addition to several others. The experiment with ast-attendgru-gnn showed evidence that AST structural information can lead to a better representation of code – it is useful to know that "guess" is a variable name declaration. But other relationships also exist. The variable "guess" is used in the call to `os.write()`. This relationship is a data dependency and is useful to human readers (Freeman, 2003). A human attempting to comprehend this code would likely note that whatever is passed into the subroutine as a parameter via the variable "guess" is subsequently written out via a method call. The benefit to Liu *et al.*'s approach is that it captures

this relationship and uses it to form a more-complete GNN-based representation of the code.

A caveat is that as more edge types are added to the graph, more information will be propagated among nodes, which may have effects that are difficult to explain. Imagine in Figure 23.1 if an edge were to exist between "guess" at area 4 and "guess" at area 1, denoting a data dependency. A typical GNN design would propagate information across this edge. The result would be that the nodes around the location that uses "guess" would gain information from the nodes where "guess" is defined. But now imagine a control dependency from the `try` block start to the call to `os.write()`. The information would then also propagate from the `try` block to the use of "guess" over the control flow edge and then from the use of "guess" to the definition of "guess" over the data flow edge. This connection is difficult to explain – it is not clear what it means for a `try` block to be connected to the parameter list. A human may proffer an explanation for this particular subroutine, but a model such as ast-attendgru-gnn would always propagate information across these edges, even when it does not make sense to do so.

Liu *et al.* solve this problem by using an attentional GNN proposed by Zhu et al (2019b). Essentially, this GNN adds an attention layer as a gate prior to propagating information across an edge. The input to this gate includes the node embedding for the node at the origin of the edge, plus an edge embedding for that type of edge. The result is that the model learns during training when to propagate information from a node over a particular type of edge. That way, information from the, e.g., `try` block may or may not propagate to the parameter list, depending on whether that particular connection was useful during training. Liu *et al.* use the learned representation of code to help locate similar code comments in a database of those comments. However, the big picture idea is to use an attentional GNN to emphasize some edges in the code over others when the graph representation of code becomes large and complex, and this idea may serve as inspiration for a variety of software mining tasks. It is an example of how better micro-level representations of code can assist these software mining tasks.

23.4.2.2 Example Macro-level Improvement

One inspiration for macro-level improvement to neural code summarization is from (Aghamohammadi et al, 2020). Their approach focuses on generating summaries of code in Android projects. The approach is divided into two parts. The first part centers around an attentional encoder-decoder model similar to the attendgru baseline described by LeClair et al (2019). They use this model to generate an initial code summary based solely on the words inside the subroutine itself. The second part is to augment the initial summary with phrases from the summaries of other subroutines in the same project. The approach is to obtain a dynamic call graph of the Android program, which represents the actual runtime control flow from one subroutine to the next. Then a subset of the subroutines in this call graph is selected using PageRank – the idea is to emphasize the subroutines, which are called many times or hold

other importance measurable from the structure of the call graph (McMillan et al, 2011). The summaries from these subroutines are then appended to the initial summary.

Aghamohammadi et al (2020)'s approach demonstrates an advantage to macro-level information. The macro-level information is the dynamic call graph of the entire program, and it is used to augment summaries created from the source code itself. The summaries tend to be longer and to provide more contextual information to readers. Recall `sendGuess()` in Example 23.1, for which ast-attendgru-gnn wrote "sends a guess to the socket." The approach by Aghamohammadi et al (2020) may (hypothetically) find that the subroutine that calls `sendGuess()` is a mouse click handler subroutine, and so would append, e.g., "called when the mouse is used to click the button." Human readers of documentation benefit from knowing how subroutines are used, so summaries that include this macro-level information tend to be considered more valuable by those readers (Holmes and Murphy, 2005; Ko et al, 2006; McBurney and McMillan, 2016).

Macro-level representations of code for software mining tasks are likely fertile ground for GNN-based technologies. The dynamic call graphs which Aghamohammadi et al (2020) extract contain information from actual runtime use, and a GNN may serve as a useful tool in generating a representation of this information. Yet, applications of GNNs to macro-level data for software mining tasks are still in their infancy.

23.5 Summary

In this chapter, we presented Software Mining Tasks as an application area for GNNs. A high-level view of any approach is to represent the software as a graph, then create a GNN model able to use this graph to learn to make predictions for a particular purpose. We present two views of software graphs: a micro- and macro-level representation. Micro-level representations predominate. For example, for the task of bug prediction in a subroutine, most approaches tend to look exclusively within those subroutines for patterns associated with that bug. Yet, evidence is emerging that macro-level representations may also benefit these tasks, as the context surrounding code is very likely to contain information necessary to comprehend that code. The future likely lies in combined GNN models of both micro- and macro-level graph representations of software.

We focus in this chapter on the task of source code summarization as an example of how GNN-based models help produce better predictions for software mining tasks. A straightforward approach is described in which the AST of subroutines is used to train a GNN, which leads to a better micro-level representation in many cases. An improvement based on an attentional GNN shows how much more complex graphs can also be exploited for better for this purpose. Yet, these improvements for code summarization likely herald improvements for many software mining tasks. Both code representational and code generational tasks depend heavily on

understanding the nuances of the structure that code, and GNNs are a likely avenue for capturing this structure. This chapter has covered the history of this research, a specific target problem, and recommendations for future researchers.

Editor's Notes: AI for Code is a very fast-growing area in the recent years. Computer software or program is just like a second language compared to human language, which is not surprising that there are many shared attributes or aspects in both languages. Therefore, we have seen this trend that both NLP and Software communities start paying a large amount of attentions in applying GNNs for their domain applications and achieve the great successes in both domains. Just like GNNs for NLP, graph structure learning techniques in Chapter 14, GNN Methods in Chapter 4, GNN Scalability in Chapter 6, Heterogeneous GNNs in Chapter 16, GNN Robustness in Chapter 8 are all highly important building blocks for developing an effective and efficient approach with GNNs for code.

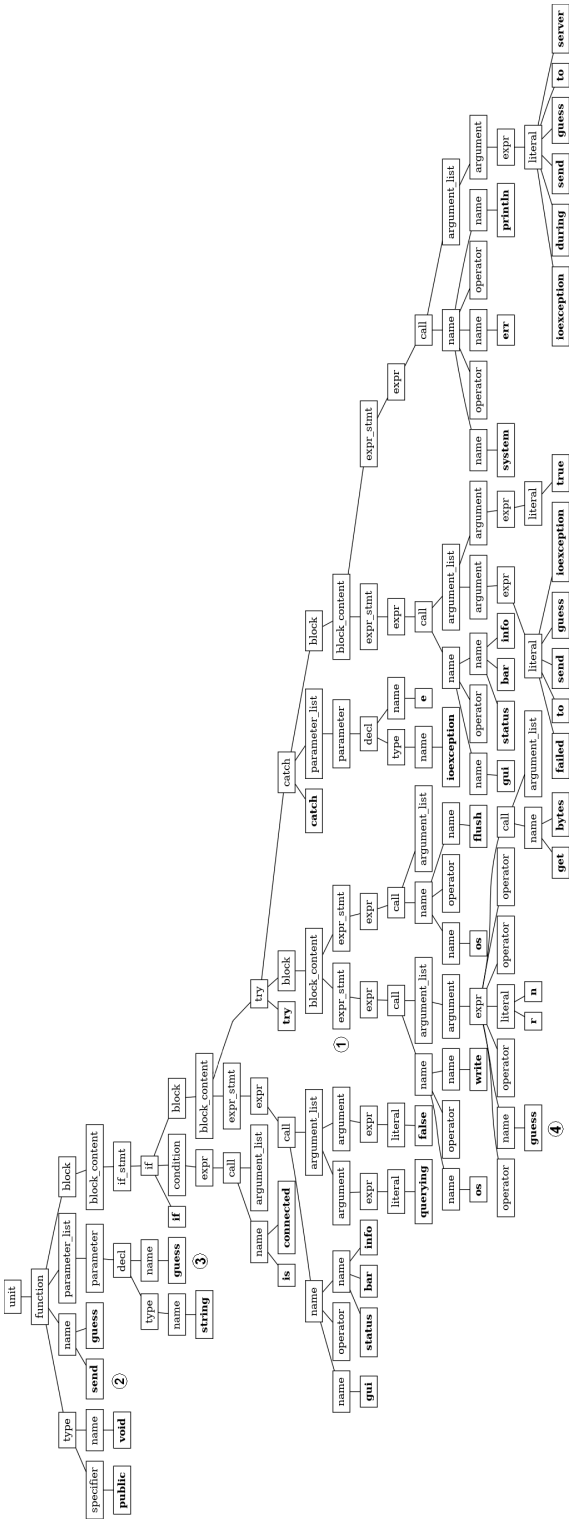


Figure 23.1: Abstract Syntax Tree for the function `sendGuess()` in Example 1. Bold indicates text from source code that is visible to a human reader in the source code file – a depth-first search of the leaf nodes reveals the code sequence. E.g., "public void send guess ..." Non-bold indicates AST nodes that the compiler uses to represent structure. Visible text is preprocessed as it would appear to the model. For example, the name `sendGuess` is split into `send` and `guess`, and both nodes are children of a `name` node, which is a child of `function`. Neither `name` nor `function` is visible to a human reader. The circled areas 1-4 are reference points for discussion in Sections 23.4.1.4 and 23.4.2.

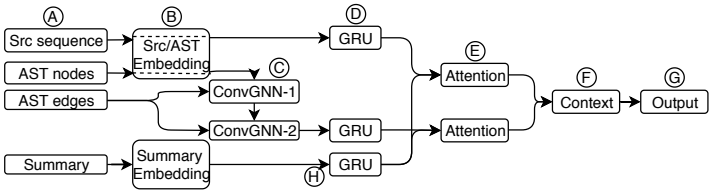


Figure 23.2: High-level diagram of the model architecture for 2-hop model.

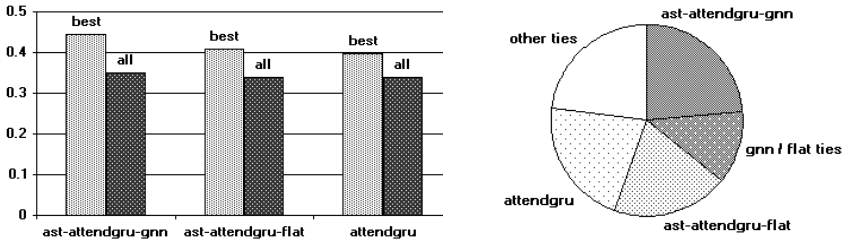
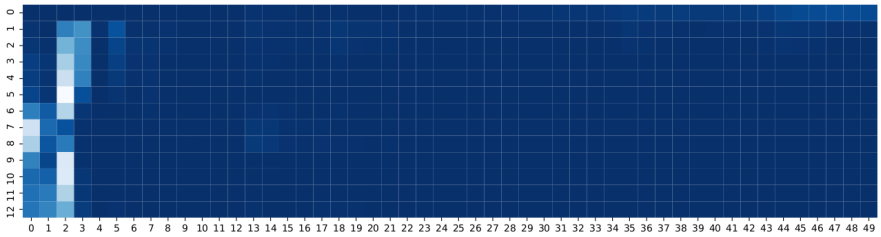
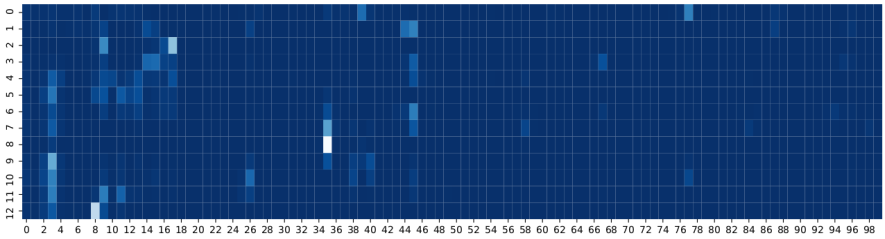


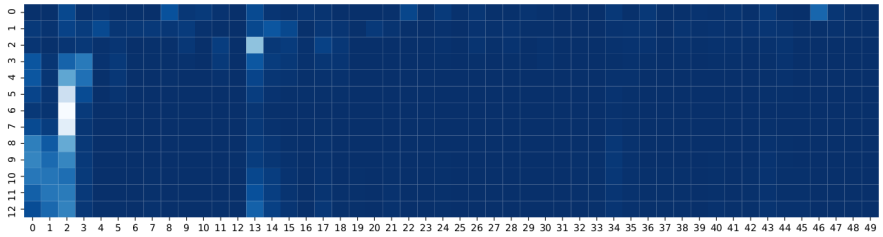
Figure 23.3: (left) Comparison of the BLEU-1 score for the subroutines where each method performed best, to BLEU-1 score for the whole test set. (right) Percent of test set for which each approach received the highest BLEU-1 score.



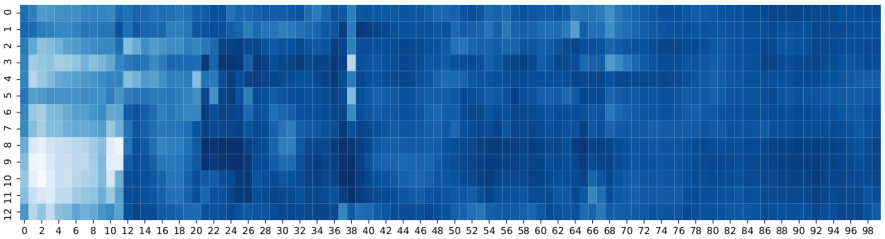
(a) ast-attendgru-gnn attention to source code sequence



(b) ast-attendgru-gnn attention to AST nodes



(c) ast-attendgru-flat attention to source code sequence



(d) ast-attendgru-flat attention to AST nodes

Figure 23.4: Visualization of attention network for ast-attendgru-gnn and ast-attendgru-flat for the subroutine `sendGuess()` in Example 23.1 and AST in Figure 23.1. Matrices are 13x100 because attention is applied between every position in the decoder output (length 13) and every position in the encoder (100 nodes or 100 code tokens). Bright areas indicate high attention. For example, position 2 in the code sequence is heavily emphasized for both models. Position 2 corresponds to the word "send" in the code sequence.