Abstract Syntax Tree Coarsening via Deep Attention-Based Node Pooling Networks

Anthony Rinaldi
arinaldi@cs.toronto.edu
Department of Computer Science, University of Toronto

Supervised By:

Avinash Gopal
avi@metabob.com
Metabob Inc.

Sushant Sachdeva
sachdeva@cs.toronto.edu
Department of Computer Science, University of Toronto

Abstract

Files of code, and collections of files (codebases), can be easily converted to an abstract syntax tree (AST) representation that contains sufficient semantic/structural information to run a program. These tree structures lend themselves naturally to be used in graph neural networks (GNNs) for static analysis of code files and codebases. However, both ASTs and GNNs have a memory setback, in that the former can be quite large and the latter do not scale well with large inputs. To solve this compatibility issue between the two, historical approaches have used heuristics to coarsen ASTs, thereby reducing their size before passing them to GNNs. These approaches suffer in their inability to rely on data to *learn* how to best coarsen a graph to retain information. In this work, we propose GAA (Graph Attention-based Auto-encoder), a data-driven approach to learning how to condense ASTs, with the goal of passing them to a downstream GNN task. We train an attention-based GNN auto-encoder network on thousands of ASTs coming from a custom augmented AST parser used on public Python GitHub repositories. We show that graphs can be shrunk by 92.5% without any reduction in performance on downstream graph classification tasks.

MScAC Final Report - December 2023

1 Background Information

Metabob is a company founded around a single goal: to help developers save time and the tedium of tracking down and solving critical bugs and security issues within their projects. Bugs in the code can lead to system failures, data loss and other issues. Artificial Intelligence (AI) models can help identify bugs early, which can help improve the overall quality of the software and reduce the risk of failures or data loss.

Currently, all attention is being focused on a single product that detects flaws, along with how they interact with other locations in the codebase and provides recommendations on how to fix them. Metabob is unique from its competitors in that it identifies problems with existing code, rather than generating code from human-typed prompts. Further, Metabob can find flaws that span across an entire codebase (i.e., a project or product), whereas other tools simply look for problems in a single file, without understanding the dependency between files.

Metabob uses an ensemble of AI tools in its pipeline. First, a graph neural network (GNN) identifies bugs in the codebase based on the abstract syntax tree (AST) representation of the codebase. Then a large language model (LLM) generates a code fix recommendation based on contextual information in the codebase, as well as the detected error type from the GNN. GNNs are known to have memory issues for large graphs since the number of activations in the network scale linearly with the number of nodes, and the number of gradients linearly with the number of edges.[1] Since the ASTs grow with the size of the codebases, GNNs may not work with large codebases during training or inference. To circumvent this problem, Metabob previously would algorithmically shrink the AST using heuristics. The goal of this project is to remove the dependency on human-designed heuristics by using a GNN to shrink the ASTs instead. This will allow for a data-driven, learnable algorithm mapping from a larger space of ASTs to a reduced space of ASTs. In theory, this should improve the detection rate of the downstream bug detection GNN since the AST shrinking GNN will preserve more structural and semantic information in the shrunken AST compared to the heuristic approach.

The first step in this project required the development of an AST parser which adds additional edges than a typical AST to account for imports, function/method call resolution, and variable lookup. Thousands of GitHub repos were parsed using this new custom parser, which would serve as the inputs for training the shrinking/coarsening GNN. An auto-encoder-based GNN was trained on these parsed GitHub repos, resulting in a fitted model. Next, to create the training data for the downstream bug detection GNN, GitHub pull requests (PRs) were cloned, parsed and passed through the coarsening GNN. PRs were used so that we could pinpoint where the bug in the code was (i.e., what was changed in the PR), thus marking nodes in the coarsened AST as buggy or non-buggy. We then train the downstream model using this newly generated data and evaluate its performance versus the original heuristic approach. The industry supervisor was responsible for helping with all steps of the process, as well as training and evaluating the bug detection GNN. The academic supervisor was responsible for assisting with validating concepts/ideas, pointing out flaws with our approach, and recommending possible directions to consider and experiment with.

2 Research Goals and Outcomes

The goal of this project is to identify the best data-driven (i.e., learnable) method for shrinking an AST enough to be processed easily by subsequent GNN classification tasks. This task is important since it reduces the dependency of the current pipeline on a heuristic, human-generated graph coarsening algorithm. The heuristic coarsening algorithm was suitable for the early stages of the product, but as Metabob evolves and aims to increase its detection rate and performance, we must opt for a more sophisticated graph coarsening method.

The formal definition of the research problem is as follows. Given an AST parsed from a codebase, we define the augmented AST (AAST) as the AST with extra edges added from [25], as well as a set of proprietary edges. Let this AAST be defined as $G = (V, E)$, where V is the set of vertices in the graph (i.e., $V = \{v_1, ..., v_n\}$) and E is the set of directional edges in the graph (i.e., $E = \{(v_i, v_j) = e_{ij} | 1 \leq i \leq n, 1 \leq j \leq n\}$). The goal is then to find some new graph $G' = (V', E')$ such that $V' \subset V, E' \subset E$, implying that $|V'| < |V| = n$ and $|E'| < |E|$. It is easy to find any trivial solution to this problem, so we impose the constraint that we want to solution G' to

[1] This is true for most GNNs, but scaling can happen exponentially depending on the network design.

have *maximal information*. The term maximal information is intentionally left ambiguous for the time being. With $H = (V', E') \subset G = (V, E)$ meaning precisely $V' \subset V$ and $E' \subset E$. Along with $f : \text{Graphs} \to \mathbb{R}$ representing an information function on graphs, the problem becomes defined by the following equation:

$$G' = \arg\max_{H \subset G} f(H) \tag{1}$$

Even with an appropriately defined information function, this problem is intractable. To approximately solve the problem, we use deep neural networks to learn how to find a subset of the original AAST with high information. To simplify the problem further, we typically constrain the size of G', for example, $|V'| < 1/100|V|$. We typically only restrict the number of nodes in the graph and do not limit the number of edges.[2]

While we specify the task as outputting a smaller graph with maximal information, many works in the current literature only output this smaller graph as an intermediate output (i.e., network activation) on its way to perform some higher level/alternative task such as graph-level or node-level classification. Therefore, these approaches cannot be used directly in our methods, since we wish not to work on some downstream task, but rather just shrink the graph without any supplemental task (e.g., classification) information. Nevertheless, ideas and methods from these works can be used in our approach. There are two main approaches we will look at in the graph pooling literature: **flat** and **hierarchical** pooling.

Flat Pooling

Flat pooling takes a graph and learns a single graph-level representation vector for the entire graph. This happens in a single step, as opposed to the hierarchical approach that happens gradually. This graph representation can then be used in graph-level tasks. Some of these methods aggregate (usually mean) the nodes of the network and apply some non-linearity during aggregation to form a graph-level representation [20]. More recent approaches use attention-based representations [14, 9], which were an inspiration for our method.

Hierarchical Pooling

The hierarchical pooling approaches align more closely with our auto-encoder-based approach and ideas can be borrowed for inspiration. Hierarchical pooling takes the graph and either merges together nodes and edges (*node cluster pooling*) or removes nodes and edges (*dropout pooling*) to achieve a reduced version of the graph. Node cluster pooling generates a cluster assignment matrix that dictates how nodes should be merged in subsequent layers of the network. These approaches have proven to be very successful [26, 21, 3], however, nodes in deeper layers do not have a one-to-one correspondence with nodes in the original graph, so there is information loss here. For our purposes, we need to preserve node location information, so we cannot merge nodes together to form clusters, thus node cluster pooling is not suitable. Node dropout pooling, on the other hand, does sustain a one-to-one correspondence with nodes in deeper and shallower layers of the network. This approach (Top-k pooling) comes up with a score for each node in the graph and keeps the k nodes with the highest score (where k is a hyperparameter).[3] Most of these methods vary in the way they calculate node scores [11, 27, 17, 10]. Some methods also select nodes in a different manner than Top-k pooling [27]. Although dropout pooling aligns closer with our auto-encoder-based approach, few dropout pooling papers use their methods in an auto-encoder [11, 12].

With hierarchical node dropout pooling, we can gradually reduce the size of the graph to a latent representation, and then gradually build up the graph, using a decoder, to its original size. This is the most basic auto-encoder setup where one uses the latent representation (i.e., the output of the encoder) of the network as the reduced version of input [2]. The very first GNN-based auto-encoder is the Graph U-Net [11], which will serve as the baseline comparison for much of this paper. A more sophisticated auto-encoder [12] is used as our starting point that we will be improving.

In this report, we propose a novel GNN auto-encoder setup, which we call GAA (Graph Attention-based Auto-encoder) that approximately solves Equation 1. We improved the limitations of existing

[2]Since we are removing nodes to form a subset, edges will also be correspondingly removed, so we only need specify a restriction on one or the other.

[3]Full mathematical detail will be provided in section 3

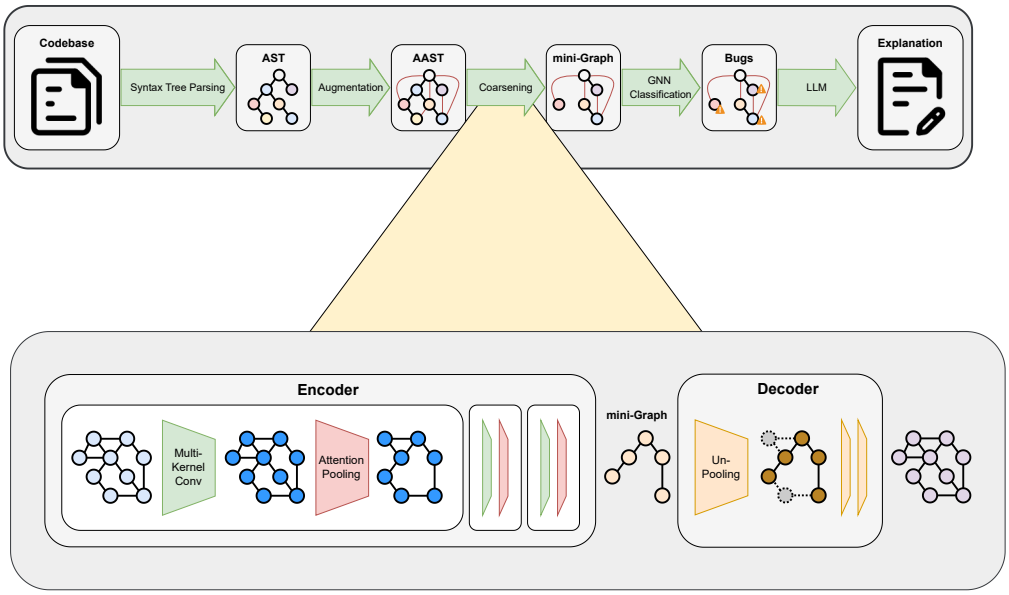

Figure 1: **Model Architecture and Coarsening Breakdown.** **Top**: All steps in the model pipeline that go from a codebase to explained errors in the codebase. **Bottom**: Expanded view of the graph coarsening architecture used to reduce graph sizes while retaining node and topology information.

literature to make the network more expressive and learn how to retain information better. GAA uses a multi-headed attention-based node scoring module to determine how to shrink the input graph.

3 Methods

3.1 Overall Metabob Pipeline

Although the Metabob pipeline is already spoken about, we formalize it here. Figure 1 (top-half) gives an overview of how the entire process works in inference (i.e., not training). First, a codebase (i.e., multiple code files) is parsed into its typical AST structure. From there, we augment the AST with additional edges from [25], as well as some proprietary additional edges, creating an AAST. The AST is parsed using the *tree-sitter* C library (with bindings in Python[4]) and edges are added based on local import resolution, function call resolution, class/method resolution, and variable resolution. This AAST gives the tree more topology for message passing in the subsequent Graph Convolution Network (GCN) [16]. At this point in the pipeline, we also generate node embeddings based on their [*row*, *column*] location in the codebase [22] and the text stored at the node using fastText embeddings [4]. Our solution, GAA, will then coarsen the AAST into a smaller structure we call the *mini-Graph*. The mini-Graph is then passed through another GNN, such as Graph Attention Network (GAT) [24, 6], for bug-detection where nodes are labelled as either buggy or non-buggy, as well as a bug type (e.g., logical error). The buggy nodes are used to pull contextual information from the codebase and along with the type of error, are passed to an LLM to generate a bug description. If the user wishes, they may generate a recommended fix, which will rely on the contextual codebase information, the bug type, and the LLM-generated problem explanation.

3.2 Graph Attention-Based Auto-Encoder

The GAA architecture contains an encoder E and a decoder D that coarsen and expand their inputs, respectively (see Figure 1 bottom half). The encoder is the method by which we shrink our input graph. The decoder is used to help determine the information contained in the latent graph $G' = E(G)$ (recall Equation 1). The decoder allows us to reconstruct the input from our smaller representation, in an attempt to perfectly recreate the input (i.e., $D(E(G))$). If the decoder can reconstruct the input

[4] https://github.com/tree-sitter/tree-sitter-python

well, then the latent graph G' contains a lot of information on the original graph, and vice versa. Thus, our measure of information, to solidify the ambiguous notion presented before, is the ability to reconstruct the original graph G from the latent graph G' using the decoder $D(G')$. The way in which we measure the similarity between $D(G')$ and G will be described later.

3.2.1 Layer Definitions

Convolution Layers

The network is expected to take in an arbitrary-sized graph and reduce its size dynamically (i.e., there will be a different number of nodes removed for each input graph). For this reason, we must use an inductive graph convolution operator in our network, rather than a transductive operator. We take inspiration from GraphSAGE [13] for our inductive graph layer. GraphSAGE defines the convolution operator as:

$$h_{l+1,i} = W_1 h_{l,i} + W_2 \, aggregate_{j \in \mathcal{N}(i)} h_{l,j} \tag{2}$$

Here, $h_{l,i}$ denotes the vector representation of node i in layer l of the network. W_1 is a learnable parameter for updating the current node representation based on its representation in the previous layer. $aggregate$ is a method to combine all the node representations of the neighbours of node i (i.e., $\mathcal{N}(i)$). GraphSAGE [13] uses mean aggregation, however, other forms are possible (e.g., max, min, sum). W_2 represents a learnable parameter for updating a node's representation based on the aggregated information of its neighbours.

We expand on the above formulation by allowing for multiple learning *heads*, meaning we can learn more ways to aggregate neighbourhood information. Our convolution layer is as follows:

$$h^{(k)}_{l+1,i} = W^{(k)}_1 h_{l,i} + W^{(k)}_2 \, aggregate_{j \in \mathcal{N}(i)} h_{l,j} \tag{3}$$

$$h_{l+1,i} = \sum_{k=1}^{K} \sigma\left(h^{(k)}_{l+1,i}\right) \tag{4}$$

Where $h^{(k)}_{l+1,i}$ represents the k^{th} node representation of node i in layer $l+1$. Now the learnable parameters W_1 and W_2 have superscripts to represent the K different version of the parameters. The aggregation function we use is the mean, as in GraphSAGE [13]. We then combine the K representation of each node using an activation function σ, followed by summing each activation.

To improve the expressivity of the convolution layers, we typically use the second-degree adjacency/connectivity matrix A^2, rather than the first-degree adjacency matrix A. This is typical in the literature [11, 12, 13] and theoretically makes sense as it allows for more information flow throughout the network and less sparse (more dense) neighbourhoods.

Node Scores

The other part of the network that we must define is how to calculate the node scores for each node to determine which nodes to drop at each layer of the network. This is done using two methods: (1) an attention-based mechanism and (2) a directional-based learnable linear projection. These two methods are then added to come up with an overall representativeness score for the node that determines its importance in the graph relative to other nodes. First, to calculate the attention-based score, we take the inner product of each node with its neighbours and sum the result:[5]

$$\alpha_{l,i} = \sum_{j \in \mathcal{N}(i)} h^{\top}_{l,i} h_{l,j} \qquad s.t. \quad \exists e_{ij} \tag{5}$$

Where $\alpha_{l,i}$ is the attention-based importance score of node i in layer l of the network.

[5]We also apply some engineering tricks such as normalizing node features and using layer normalization of activations, but these specifics are more implementation details than model details.

Next, to calculate the directional-based score we concatenate node embeddings and apply a linear projection, summed over all neighbours/outgoing edges:

$$\beta_{l,i} = \sum_{j \in \mathcal{N}(i)} W_a \left[h_{l,i} \parallel h_{l,j} \right] \qquad s.t. \quad \exists e_{ij} \tag{6}$$

Where W_a is a learnable parameter. Combining Equation 5 and Equation 6 gives us the overall importance score of node:

$$a_{l,i} = \alpha_{l,i} + \beta_{l,i} \tag{7}$$

Just as in the convolution layer (subsubsection 3.2.1), we use a multi-headed parallelized approach to calculating node scores. The multi-headed node scores are calculated as follows:

$$a_{l,i} = \sum_{k=1}^{K} \left(\alpha_{l,i}^{(k)} + \beta_{l,i}^{(k)} \right) \tag{8}$$

$$= \sum_{k=1}^{K} \left(\sigma_1 \left(\gamma^{(k)} \sum_{j \in \mathcal{N}(i)} h_{l,i}^{\top} h_{l,j} \right) + \sigma_2 \left(\sum_{j \in \mathcal{N}(i)} W_a^{(k)} \left[h_{l,i} \parallel h_{l,j} \right] \right) \right) \tag{9}$$

Where σ_1 and σ_2 are two, possibly different, activation functions used for the different components of the score calculation. $\gamma^{(k)}$ is a learnable parameter that helps scale the node-similarity component of the score to allow for different learning between the different heads. Without this parameter, all K heads would report the same score (for the first term in the formula) for a given node.

Pooling

Using the generated node scores (see above), we apply Top-k pooling as in [11] to select the nodes that remain in the subsequent layer. Pooling not only removes nodes (and any edges corresponding to those nodes), but also updates the node representations based on their scores. Pooling occurs at each layer in the encoder after the respective graph convolution layer is first applied.

Un-Pooling

To perform un-pooling (in the decoder), we ensure that E and D have a symmetric structure so that whatever nodes were removed in layer l of E are added back in layer $L - l$ of D (where L is the total number of layers in the encoder). The network does not need to guess which nodes to add back and where. It receives full information on which nodes are being added back, and what the topology (edges) of these nodes look like. All that the decoder must do is learn the representation of these nodes properly since the nodes are added back with no embedding. Although this seems like there is too much information leakage and that the decoder is slightly cheating, there is no better alternative suggested by current literature.

3.2.2 Model Definition

To construct a model using the above layers, one creates a single encoder layer using a pair of convolution layer and a Top-k pooling layer. This encoder layer is repeated L times. The output of the L^{th} Top-k pooling layer is the latent representation of the graph/coarsened graph/encoder output.

The decoder D is constructed by stacking L pairs of un-pooling and convolution layers. The output of the decoder is the output of the L^{th} decoder convolution layer. This output will be the same shape and have the same edge information as the input graph G, however, the node embeddings may not be the exact same.

3.2.3 Latent Enhancement

The original AASTs that are passed to the network are typically very well connected (see Figure 2 left). Once consecutive pooling layers are applied, fewer nodes remain, thus fewer edges remain. The

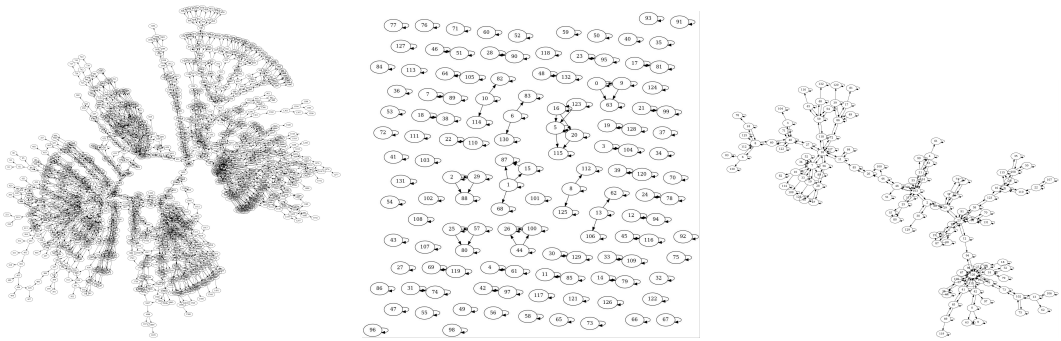

Figure 2: **Example AST at Points in the Model Pipeline.** **Left**: AST produced directly from the codebases. This includes augmented edges added for function resolution, import resolution, etc. **Middle**: Graph produced after coarsening using GAA i.e., latent graph. **Right**: Graph produced after re-connecting the components of the latent graph based on their proximity in the original AST.

coarsened graph G' is no longer well-connected like the original graph G (see Figure 2 middle). This makes any subsequent GNN task on G' difficult since there is little information propagation between the nodes. To circumvent this problem we propose a solution to re-connect the latent graph G' based on the topology of the original graph G.

We apply two connecting algorithms successively to ensure that G' becomes roughly as connected as G. We measure connectivity as the number of *islands* present in a graph (i.e., the number of connected components).

Algorithm 1

See subsection A.1 for a detailed algorithm definition.

First, we look at the directed version of the original graph. For each (target) node in the latent space, we look for a (directed) path from that node to any other node in the latent space. We look for this through both outgoing and incoming edges. If a path exists between the target node and the rest of the latent graph (based on the original graph G), we add an edge between the two based on which path is the shortest in the original graph G.

Next, we look at the undirected version of the original graph G_u and perform the same test. If there is a path between the current target node and any other node in the latent graph (based on G_u), we record the length of this path. Whichever node has the shortest path to the target node (based on G_u) is connected directly to the target node in the latent graph.

We first look at the directed, then undirected version of the graph because we want to add edges that are as similar to the original structure of the graph. Suppose for a given node, there are two candidate nodes to connect it with in the latent graph. One of the candidates has a directed path to the given node, while the other candidate has an undirected path to the given node. We prefer connecting the given node with the first candidate since this additional edge preserves the directionality of the original graph. For this reason, we first consider the directed original graph, then the undirected original graph.

We denote the output of the first connectivity enhancing algorithm as $G'_1 = (V'_1 = V', E'_1)$.

Algorithm 2

See subsection A.2 for a detailed algorithm definition.

The first algorithm (algorithm 1) does not completely fix the lack of connectivity in the reduced graph G'. There are still typically more islands in the G'_1 than G even after adding the additional edges. The second algorithm improves connectivity more and ensures that the number of islands in G'_2 is the same as G, so that connectivity is relatively similar, where G'_2 is the output of the second connectivity enhancing algorithm.

The second connecting algorithm first identifies all the islands in G, and the nodes associated with each island. It does the same for the latent graph G'_1. For each island (o_i) in G, we identify which islands in G'_1 are a subset. For each of the subset islands (s_j) in G'_1, we find the shortest path from

any node in that island, to any node that is in $o_i \setminus s_j$ (and also in the latent graph). An edge is added between these two nodes. This ensures that each island s_j is connected to some other node in o_i (i.e., its parent island), and not just a node already present in s_j.

3.3 Experiments

To explore the effectiveness of our method, we conducted two experiments: **(1)** Reconstruction of graphs and **(2)** Bug detection in codebases. The former compares our approach to a simpler autoencoder approach [11] in terms of the reconstruction ability of original graphs from the latent representation. The latter experiment tests this downstream usability by comparing the proposed method to the existing heuristic alternative (i.e., Metabob's current pipeline) and to a baseline method [11] in terms of which results in more accurate bug detections. This is a more practical experiment since this is how the method will be used in practice and will determine if downstream tasks can informatively use the outputs of the method, or if they are better off using a heuristic parse or baseline coarsening as the input.

3.3.1 Training Setting

Optimization

We train the network using the Adam optimizer [15] and Mean-Squared Error (MSE) reconstruction loss. The loss is calculated between the original node embeddings in G and the reconstructed node embeddings $D(E(G))$. The exact formulation is as follows:

$$L(\theta) = \frac{1}{n}\sum_{i=1}^{n}(v_i - D_\theta(E_\theta(v_i)))^2 \tag{10}$$

Where there are n nodes in the graph and θ parameterizes both E and D and they are trained jointly. We slightly abuse the notation for E and D, applying it to a node, rather than the entire graph.

Regularization

We ran experiments with regularization added to the loss function in the form of penalizing latent representations that were not well connected. The goal of these efforts was to avoid using algorithm 1 and algorithm 2 for enhancing connectivity and have the network naturally enforce connectivity. None of the aforementioned modifications to the loss function proved useful in terms of the connectivity of the latent space. More details on the attempted loss modifications can be found in subsection A.3.

Implementation Details

We apply an exponential learning rate scheduler/annealing with a rate of 0.975 and weight decay of 1e-4 to all experiments to help training reside at a minimum once it reaches a suitable location. Additionally, we use gradient clipping with a value of 0.5 to help with exploding gradients during training and to keep gradient steps as smooth as possible.

We also use early stopping that tracks the validation loss of each epoch and stops training when the validation loss has not improved (decreased) in more than 10 epochs. This helps reduce overfitting and saves time when training models without using an unnecessary amount of epochs.

Model Constraints

As mentioned in section 2, we must specify some constraints on our coarsening approach to help make Equation 1 tractable. We seek solutions to the equation that reduce the size of the graph (in number of nodes) by at least 80%. This constrains the solution set to a much smaller size, although still intractable, and will help when approximately solving the original research question.

3.3.2 Data

We use various datasets for running experiments on our model. Given the inductive nature of our training, we show our model entire graphs during training and train using multiple unique graphs. This is opposed to the transductive training setting where training happens on a single graph by only showing specific parts of the graph during training. We thus need to train on datasets with (1) multiple

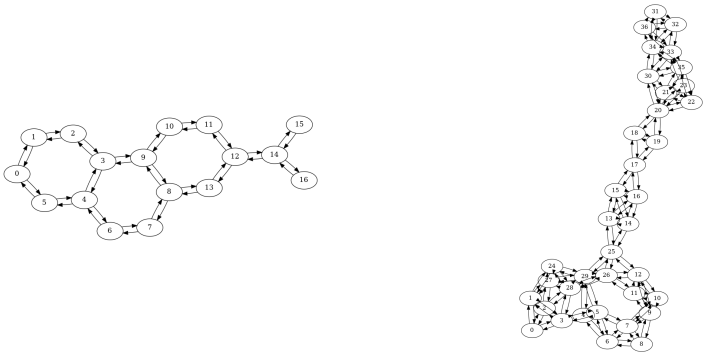

Figure 3: **Example External Graphs** The connectivity and structure of the external graph datasets are much different than the internal AAST data. **Left**: Example graph from MUTAG [7] dataset. **Right**: Example graph from ENZYMES [5] dataset.

graphs and (2) dense node embeddings. We create our own custom dataset using GitHub repositories called **AASTs**. The details of constructing this dataset are contained in subsection A.4. The other datasets we use are **PROTEINS** [5] where graphs are enzymes and non-enzymes and node features are amino acids, **ENZYMES** [5] where graphs are enzymes from the BRENDA database, **MNIST** [8] where graphs are the classical MNIST digits stored in graph form, **MUTAG** [7] where graphs are nitroaromatic chemical compounds, and **AIDS** [1] where graphs are molecular compounds from the AIDS Antiviral Screen Database of Active Compounds. Figure 3 shows example graphs from MUTAG and ENZYMES datasets. The structure is quite different from out AAST dataset, so these dataset serve as a good test to whether out method can generalize to multiple graph structures.

All datasets are split into 56% training, 24% validation and 20% testing. We set the seeds for all random number generators in `Python` (e.g., `numpy`, `PyTorch`) to ensure that every model trained on any dataset receives the same split and is directly comparable to all other models. No transformations are applied to the dataset, other than operations that exist within the network itself.

3.3.3 Baseline Comparison

We choose to compare our model to the standard baseline graph auto-encoder, Graph U-Nets [11] (U-Nets for short). We train the U-Net model with all the same train settings specified above in subsubsection 3.3.1. We compare GAA and U-Nets on all the aforementioned datasets using validation loss as the metric to determine the superior model. Later, in section 4 we also go into more analysis on the latent graphs that each of the models output and which representation would be preferred.

3.3.4 Hyperparameter Search

Here we describe how we discovered which hyperparameter values to use in both the GAA and U-Net network. We performed a hyperparameter search over the values specified in subsection A.5. We used the AAST dataset for hyperparameter search since it is the largest dataset and is the intended use case of the proposed GAA model. We perform the search over a subset of 5000 graphs out of the entire AAST dataset. The results obtained from the AAST dataset were transferred and used as the values for training on all other datasets mentioned in this report. Therefore, we end up with a single optimal setting for the GAA model and the U-Net model, both based on the AAST dataset and use these values for the duration of the report. The best hyperparameters that we landed on can be found in subsection A.6.

Model		Dataset					
		AASTs	PROTEINS [5]	ENZYMES [5]	MNIST [8]	MUTAG [7]	AIDS [1]
GAA	Train	0.167	0.092	0.106	0.043	0.047	0.015
	Val	**0.161**	**0.119**	**0.103**	**0.043**	**0.045**	**0.015**
U-Nets	Train	0.237	0.612	0.978	1.202	0.773	0.166
	Val	0.211	0.632	0.639	0.578	0.387	0.105

Table 1: **Graph Reconstrution Comparison.** GAA compared to Graph U-Nets [11] on various graph reconstruction datasets. Scores represent MSE reconstruction loss on both train and validation splits. Validation scores are used for the basis of comparison.

Method	Accuracy	Precision	Recall	F1 Score
Hueristic	0.8286	0.7222	**0.9286**	0.8125
U-Nets	0.7554	0.6739	0.8052	0.7337
GAA	**0.8395**	**0.7692**	0.8824	**0.8219**

Table 2: **Downstream Bug Classification Performance Comparison.** The later two methods use the latent space of the auto-encoder as the coarsened version of the graph and as input to the downstream task. **Hueristic Parser:** existing approach to coarsening AASTs based on human intuition. **U-Net:** Graph auto-encoder architecture proposed by [11]. **GAA:** Our proposed attention-based auto-encoder approach.

4 Results and Discussion

4.1 Graph Reconstruction Results

We first compare our method to U-Nets, one of the original auto-encoder methods for GNNs, on the reconstruction task. The method is simple, relying mainly on graph convolutional operators to perform encoding and decoding, making it a typical baseline for graph reconstruction tasks [18, 19]. For this experiment, we curate many different graph datasets (see subsubsection 3.3.2). The best model architectures and hyperparameters (see subsection A.6 and subsubsection 3.3.4) were used to train on all datasets in an inductive manner, and their performance can be seen in Table 1. Our method, GAA, outperforms the baseline in all domains.

4.2 Bug Detection Results

We next compare our method to U-Nets and the heuristic AST parser that Metabob currently uses to reduce the size of its ASTs for the downstream bug detection task. We test how each of the three different methods performs in the bug detection task based on various performance metrics. To perform this comparison, we compare each of the three models on identical train and validation splits of our proprietary bug detection dataset. The heuristic parser uses the raw parsed (heuristic) AST as the input to the downstream task, while U-Nets and GAA use the latent representation of the auto-encoder as the input to the downstream task. All methods are compared using the same GAT network [24] with the same hyperparameters, ensuring that the only difference between the results is due to differences in the input representations that the GAT network receives. The result of this comparison can be seen in Table 2. Our method outpaces the U-Net in all measures of performance but falls behind the existing heuristic approach in recall only. Potential reasons for this disparity are discussed in the following sections.

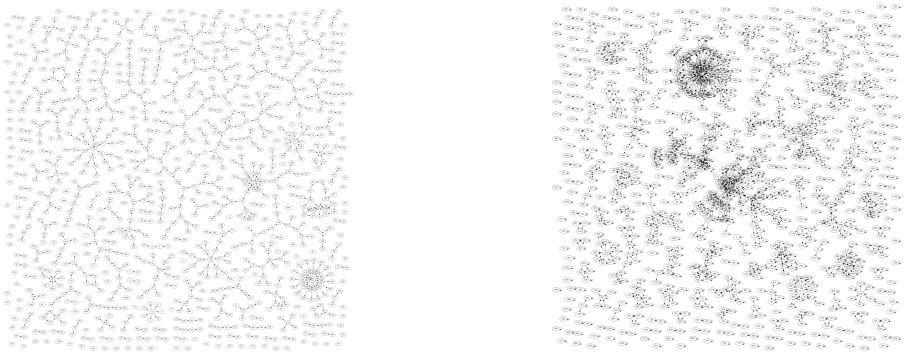

Figure 4: **Example Latent Graph Connectivity Before Enhancement** Both latent graphs look somewhat similar since neither model prioritizes connectivity in any way. **Left**: Latent graph produced by U-Net [11] method. **Right**: Latent graph produced by our GAA method. The GAA method results in a slightly more connected latent space, meaning the method more naturally considers the latent shape compared to U-Net method.

4.3 Discussion

Given that GAA outperforms U-Nets on the reconstruction task, it is clear that node embeddings alone [11] are not a sufficient selection criterion for node reduction in a graph. U-Nets rely strictly on node embeddings to determine which nodes to retain in a network, whereas GAA includes additional information, like node-neighbour similarity and the learned relationship between nodes. This additional information is critical in learning how to select important nodes, as seen by the large performance differences between the two models. Our method successfully incorporates the mutual information between nodes, and somewhat the graph topology, to better select which subset of nodes to represent a graph with, as seen by its ability to more easily reconstruct the graph from its latent embedding.

GAA also outperforms U-Nets in the downstream task. This means that the representations generated from GAA are not only useful in reconstructing the original graph but also contain sufficient information about the graph that can used in subsequent tasks. The connectivity and embeddings of the latent structure for the two methods are compared later to confirm this claim.

We see that GAA performs better than the heuristic parser in the downstream task as well, in all areas except recall. The higher precision and lower recall, compared to the heuristic approach, highlight that there are fewer false positives and more false negatives with the GAA approach. The GAA approach is more conservative in making predictions than the heuristic approach. The reason that the heuristic approach is likely less conservative is because we control the rules that share information between nodes and those that coarsen nodes. There is likely information shared between nodes that may not be ideal, causing more nodes to be classified as bugs incorrectly. On the other hand, the GAA approach learns how to share information between nodes more effectively and is likely more strict in doing so. This means that fewer nodes seem to have a trace of a bug, thus there are fewer bug classifications made overall. This is what results in the lower recall of the network. The other reason we see the lower ability to retrieve bugs (i.e., recall) is because the GAA model is not trained with the objective of performing bug classification, but rather just to reconstruct itself. Some information that may be useful in reconstruction may be harmful in bug detection which is why we see the inability to detect bugs in some cases. If we could somehow incorporate the bug detection objective in training the GAA network, it would likely perform much better than the heuristic alternative since the information contained in the nodes will pertain directly to the classification task.

4.3.1 Latent Representations

We can investigate the resulting latent graphs in two ways: (1) through the topology of the graph and (2) through the node embeddings generated.

Connectivity Comparing the topology of GAA and U-Nets it is clear that there is not a major difference. Both methods, without the augmentation algorithms algorithm 1 and algorithm 2, generate a sparsely connected latent space (see Figure 4). When the augmentation algorithms are applied to

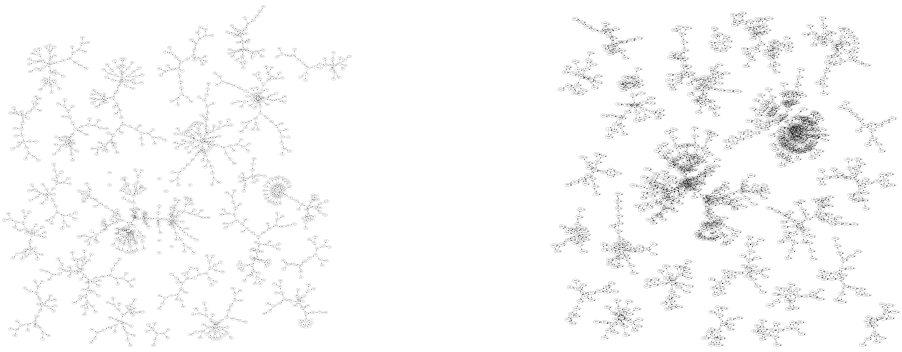

Figure 5: **Example Latent Graph Connectivity After Enhancement** **Left**: Latent graph produced by U-Net [11] method, along with latent connectivity enhancement. **Right**: Latent graph produced by our GAA method, along with latent connectivity enhancement. The GAA method results in a slightly more connected latent space, making it preferred for downstream tasks reliant on the structure of the graph.

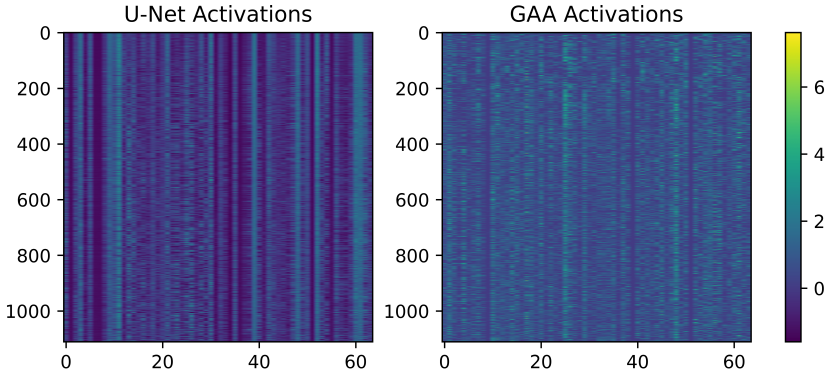

Figure 6: **Example Latent Node Embeddings Produced by Auto-Encoders** Rows represent nodes in the latent space, columns represent the index of the node embeddings, and colours represent the embedding value at a given index. **Left**: Embeddings produced by U-Nets [11] architecture. All nodes have similar embedding values, there is an *over-smoothing* problem. **Right**: Embeddings produced by our proposed GAA architecture. Nodes have more distinct embedding values compared to one another.

improve the connectivity, the resulting graph structures once again look quite similar (see Figure 5). This happens because neither method *cares* about the connectivity of the graph. There is no penalty for creating a graph that is not well connected in the latent space. The more important thing is the embeddings of the nodes. This is in part due to the flaw of the architecture that feeds information about the true edges to the decoder. If the decoder had to figure out the structure of the graph explicitly (which hasn't been done in literature), the latent space would likely be more connected since the neighbourhood structure of the graph must be preserved in order to recreate it. Since this does not exist yet, the model cares little about the graph structure, thus the latent is not well connected.

Although neither method explicitly prioritizes latent connectivity, our GAA approach does implicitly prioritize it through its directional node scoring mechanism. We can see this in Figure 4 where there are fewer islands and more edges than the U-Net approach. This is preferred since we need to perform fewer artificial enhancements to improve the connectivity because there are a greater number of natural connections still present in the latent representation of the graph.

Node Embeddings

We also compare the embeddings produced by GAA and U-Nets. Given that GAA can perform much better on reconstruction and the downstream task, it is expected that the GAA embeddings will be richer than those produced by the U-Net model. The embeddings can be seen in Figure 6.

The U-Net activations are almost constant across all nodes. We can see this in the distinct vertical lines in the node embedding image. On the contrary, the GAA activations are not constant across all nodes and vary much more. This can be seen by the lack of vertical lines. A superior model will produce less similar activations/embeddings for the nodes since the nodes are different. If the method produces similar embeddings for all nodes, it implies that all nodes are the same, which they are not (we know they are not because each node in an AAST has a very different purpose). Reconstruction and downstream tasks are difficult if the model produces similar representations for each node; it will be difficult to differentiate between the nodes. Our GAA approach produces more distinct representations for each node, making it superior for reconstruction and downstream tasks. This idea is confirmed by our quantitative results presented above (Table 1 and Table 2).

5 Conclusion and Future Work

In this paper, we present an auto-encoder-based GNN to approximately solve Equation 1. Our solution proves to retain more information in the latent graph compared to previous GNN auto-encoder approaches. Our GAA approach utilizes a multi-headed attention-based and learnable directional layer to calculate representative node scores used in the reduction process. This powerful node selection technique results in a richer latent representation of the original graph, while also retaining a similar topology to the original graph. In the case where the connectivity of the latent graph is of importance, we offer two algorithms that help enhance the connectivity in a manner that preserves relationships present in the original graph.

We show an efficient way to reduce the size of graphs while retaining as much information as possible. This method can be used in any setting where graphs are too large and a reduction is required to make downstream tasks more manageable. Specific size reduction constraints can be specified for the network to achieve sufficient reduction with minimal information loss. This work also helps prove that attention-based calculations are not only important in language-related tasks [23], but also effective in other domains, such as graph data.

The work will directly benefit the partner organization, Metabob, by improving the detection rate of its bug-detection module, while also reducing the size of datasets in comparison the the heuristic parser. Training of downstream GNN modules should be more efficient since the GAA can reduce the size of the inputs to these models by a significant amount. The qualitative performance of the new model has yet to be compared to the existing heuristic parser, but the hope is that GAA will allow for more and higher quality bug detections. Two new algorithms were proposed in this work, however, there do not seem to be other ways in which these algorithms are beneficial. Both of these algorithms take in a graph G and a subset G'. By adding more edges to G' based on the structure of G, the goal is to make G' better connected in a similar fashion to G. There may be other use cases for these somewhat specific algorithms that we cannot presently think of. Another important impact of this work is the introduction of a parallelized multi-headed framework. Previous approaches use multi-headed GNNs [12], but do not parallelize the implementation, making it inefficient. We implement a parallel multi-head attention scoring mechanism that allows for efficient scaling in the number of heads. This implementation can be used in other GNN work in the future for multi-headed-based approaches of specific techniques. Lastly, our vast number of experiments exposed flaws and shortcomings in our existing techniques. We propose these questions below as future research questions to improve GNNs and GNN auto-encoders in specific.

This work can be extended in the future by finding additional layers/calculations that can be used to improve the node importance score. Node importance scores are the most important part of the network, so this is where most attention should be paid. We already include some topological information by having a directional learnable layer, however, there is still much room for improvement. Including additional topology information will likely make the coarsening method even stronger and result in a well-connected latent space. Further work can explore better ways of combining the information from each head, such as a weighted average or a learnable average operator. Lastly, our coarsening technique can be used in other tasks that require graph *pooling*, but not coarsening. This is where the end goal is not a smaller version of the graph, rather the intermediate layers are smaller versions of the graph and the output is something completely different, like a graph-level representation for example.

There are additional research questions that arose from this work: (1) how to avoid over-smoothing in deep auto-encoder GNNs, (2) how to make the network more efficient for very large graphs, and (3) how scaling the number of heads affects performance. These questions may be answered in future research at Metabob.

Acknowledgments and Disclosure of Funding

This work was made possible by the generous financial contribution of Mitacs Accelerate Research Grant. Thank you to both of my supervisors for all your help and guidance in producing this research.

References

[1] Yunsheng Bai, Hao Ding, Song Bian, Ting Chen, Yizhou Sun, and Wei Wang. Simgnn: A neural network approach to fast graph similarity computation. In *Proceedings of the Twelfth ACM International Conference on Web Search and Data Mining*, WSDM '19, page 384–392, New York, NY, USA, 2019. Association for Computing Machinery.

[2] Pierre Baldi. Autoencoders, unsupervised learning, and deep architectures. In Isabelle Guyon, Gideon Dror, Vincent Lemaire, Graham Taylor, and Daniel Silver, editors, *Proceedings of ICML Workshop on Unsupervised and Transfer Learning*, volume 27 of *Proceedings of Machine Learning Research*, pages 37–49, Bellevue, Washington, USA, 02 Jul 2012. PMLR.

[3] Filippo Maria Bianchi, Daniele Grattarola, and Cesare Alippi. Spectral clustering with graph neural networks for graph pooling. In *Proceedings of the 37th international conference on Machine learning*, pages 2729–2738. ACM, 2020.

[4] Piotr Bojanowski, Edouard Grave, Armand Joulin, and Tomas Mikolov. Enriching word vectors with subword information. *Transactions of the Association for Computational Linguistics*, 5:135–146, 2016.

[5] Karsten M. Borgwardt, Cheng Soon Ong, Stefan Schönauer, S. V. N. Vishwanathan, Alex J. Smola, and Hans-Peter Kriegel. Protein function prediction via graph kernels. *Bioinformatics*, 21:47–56, 06 2005.

[6] Shaked Brody, Uri Alon, and Eran Yahav. How attentive are graph attention networks? *ArXiv*, abs/2105.14491, 2021.

[7] Asim Kumar Debnath, Rosa L. Lopez de Compadre, Gargi Debnath, Alan J. Shusterman, and Corwin Hansch. Structure-activity relationship of mutagenic aromatic and heteroaromatic nitro compounds. correlation with molecular orbital energies and hydrophobicity. *Journal of Medicinal Chemistry*, 34(2):786–797, 1991.

[8] Vijay Prakash Dwivedi, Chaitanya K Joshi, Anh Tuan Luu, Thomas Laurent, Yoshua Bengio, and Xavier Bresson. Benchmarking graph neural networks. *arXiv preprint arXiv:2003.00982*, 2020.

[9] Xiaolong Fan, Maoguo Gong, Yu Xie, Fenlong Jiang, and Hao Li. Structured self-attention architecture for graph-level representation learning. *Pattern Recognition*, 100:107084, 2020.

[10] H. Gao, Y. Liu, and S. Ji. Topology-aware graph pooling networks. *IEEE Transactions on Pattern Analysis & Machine Intelligence*, 43(12):4512–4518, dec 2021.

[11] Hongyang Gao and Shuiwang Ji. Graph u-nets. In Kamalika Chaudhuri and Ruslan Salakhutdinov, editors, *Proceedings of the 36th International Conference on Machine Learning*, volume 97 of *Proceedings of Machine Learning Research*, pages 2083–2092. PMLR, 09–15 Jun 2019.

[12] Yunhao Ge, Yunkui Pang, Linwei Li, and Laurent Itti. Graph autoencoder for graph compression and representation learning. In *Neural Compression: From Information Theory to Applications – Workshop @ ICLR 2021*, 2021.

[13] William L. Hamilton, Zhitao Ying, and Jure Leskovec. Inductive representation learning on large graphs. In *NIPS*, 2017.

[14] Takeshi D. Itoh, Takatomi Kubo, and Kazushi Ikeda. Multi-level attention pooling for graph neural networks: Unifying graph representations with multiple localities. *Neural Networks*, 145:356–373, 2022.

[15] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. In Yoshua Bengio and Yann LeCun, editors, *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, 2015.

[16] Thomas Kipf and Max Welling. Semi-supervised classification with graph convolutional networks. *ArXiv*, abs/1609.02907, 2016.

[17] Junhyun Lee, Inyeop Lee, and Jaewoo Kang. Self-attention graph pooling. In Kamalika Chaudhuri and Ruslan Salakhutdinov, editors, *Proceedings of the 36th International Conference on Machine Learning*, volume 97 of *Proceedings of Machine Learning Research*, pages 3734–3743. PMLR, 09–15 Jun 2019.

[18] Chuang Liu, Yibing Zhan, Jia Wu, Chang Li, Bo Du, Wenbin Hu, Tongliang Liu, and Dacheng Tao. Graph pooling for graph neural networks: Progress, challenges, and opportunities. *arXiv preprint arXiv:2204.07321*, 2022.

[19] Diego Mesquita, Amauri Souza, and Samuel Kaski. Rethinking pooling in graph neural networks. In H. Larochelle, M. Ranzato, R. Hadsell, M.F. Balcan, and H. Lin, editors, *Advances in Neural Information Processing Systems*, volume 33, pages 2220–2231. Curran Associates, Inc., 2020.

[20] Nicolò Navarin, Dinh Van Tran, and Alessandro Sperduti. Universal readout for graph convolutional neural networks. In *2019 International Joint Conference on Neural Networks (IJCNN)*, pages 1–7, 2019.

[21] Emmanuel Noutahi, Dominique Beaini, Julien Horwood, and Prudencio Tossou. Towards interpretable sparse graph representation learning with laplacian pooling. *CoRR*, abs/1905.11577, 2019.

[22] Z. Raisi and Mohamed A. Naiel. 2d positional embedding-based transformer for scene text recognition. 2021.

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

[24] Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Liò, and Yoshua Bengio. Graph Attention Networks. *International Conference on Learning Representations*, 2018.

[25] 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, 2020.

[26] Zhitao Ying, Jiaxuan You, Christopher Morris, Xiang Ren, Will Hamilton, and Jure Leskovec. Hierarchical graph representation learning with differentiable pooling. In S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 31. Curran Associates, Inc., 2018.

[27] Liang Zhang, Xudong Wang, Hongsheng Li, Guangming Zhu, Peiyi Shen, Ping Li, Xiaoyuan Lu, Syed Afaq Ali Shah, and Mohammed Bennamoun. Structure-feature based graph self-adaptive pooling. In *Proceedings of The Web Conference 2020*, WWW '20, page 3098–3104, New York, NY, USA, 2020. Association for Computing Machinery.

A Appendix

A.1 Algorithm 1

Algorithm 1: Connectivity Enhancing Algorithm 1

Input : $G = (V, E)$ original directed graph
$G' = (V', E')$ latent graph
$G' \subset G \iff V' \subset V, E' \subset E$
Output : $G'_1 = (V'_1 = V', E'_1)$ latent graph with additional edges

for each latent node $v_i \in V'$ **do**
  $incoming\_paths \leftarrow []$
  $outgoing\_paths \leftarrow []$
  for each latent node $v_j \in V' \setminus v_i$ **do**
    if $\exists$ $\text{Path}_G(v_i, v_j)$ **then**
      $outgoing\_paths \leftarrow outgoing\_paths + (v_j, \text{PathLength}_G(v_i, v_j))$
    else
      if $\exists$ $\text{Path}_G(v_j, v_i)$ **then**
        $incoming\_paths \leftarrow incoming\_paths + (v_j, \text{PathLength}_G(v_j, v_i))$
  if length$(incoming_paths) > 0$ **or** length$(outgoing_paths) > 0$ **then**
    $best\_incoming \leftarrow \min(incoming\_paths)$
    $best\_outgoing \leftarrow \min(outgoing\_paths)$
    if $best\_incoming < best\_outgoing$ **then**
      $E' \leftarrow E' \cup \{e_{ji}\}$; ▷Where v_j is the node with the shortest inbound path to node v_i
    else
      $E' \leftarrow E' \cup \{e_{ij}\}$; ▷Where v_j is the node with the shortest outbound path from node v_i
  else
    $G_u \leftarrow$ undirected(G) ; ▷Repeat everything above using undirected G
    for each latent node $v_j \in V' \setminus v_i$ **do**
      if $\exists$ $\text{Path}_{G_u}(v_i, v_j)$ **then**
        $outgoing\_paths \leftarrow outgoing\_paths + (v_j, \text{PathLength}_G(v_i, v_j))$
    if length$(outgoing_paths) > 0$ **then**
      $E' \leftarrow E' \cup \{e_{ij}\}$; ▷Where v_j is the node with the shortest undirected path to node v_i
return G'

A.2 Algorithm 2

Algorithm 2: Connectivity Enhancing Algorithm 2

Input : $G = (V, E)$ original directed graph
$G'_1 = (V'_1, E'_1)$ latent enhanced graph
Output : $G'_2 = (V'_2 = V', E'_2)$ latent graph with additional edges

$original\_islands \leftarrow \text{FindIslands}(G)$
$latent\_islands \leftarrow \text{FindIslands}(G'_1)$
for island $o_i \in original\_islands$ **do**
  for island $s_j \in latent\_islands$ **do**
    if $s_j \subset o_i$ **then**
      $path \leftarrow \text{ShortestPath}_{G_u}(s_j, o_i \setminus s_j)$; ▷Where $\text{ShortestPath}_A(b, c)$ calculates the shortest path from any node in set b to any node in set c based on the edges in graph A using Dijkstra's algorithm
      $j \leftarrow path[0]$
      $i \leftarrow path[-1]$
      $E'_1 \leftarrow E'_1 \cup \{e_{ji}\}$; ▷Where $\text{PathLength}_{G_u}(v_j, v_i)$ is minimal out of all $v_j \in s_j, v_i \in o_i \setminus s_j$
return G'_1

A.3 Loss Modifications

We attempted two variations on the loss function as follows:

$$L(\theta) = \frac{1}{n}\sum_{i=1}^{n}(v_i - D_\theta(E_\theta(v_i)))^2 + \alpha \cdot \frac{|E| - |E'|}{|E|}$$

Where $|E|$ and $|E'|$ are the number of edges in the original and latent graph, respectively. α is a parameter for how much we weigh the additional loss compared to the original. We call this **fractional edge loss**, which penalizes the latent graph for having much fewer edges than the original graph. The second loss we attempt is:

$$L(\theta) = \frac{1}{n}\sum_{i=1}^{n}(v_i - D_\theta(E_\theta(v_i)))^2 \cdot \alpha\frac{|\text{FindIslands}(G')|}{|E'|}$$

Where $|E'|$ is the number of edges in the latent, α is a parameter for how much to weight the additional loss, and $|\text{FindIslands}(G)|$ is the number of islands in graph G. We call this **island loss**.

These two variations to the regular MSE loss function for the network were optimized directly as the objective for our training.

A.4 AAST Custom Dataset

To generate our custom AAST dataset, we use open-sourced repositories on GitHub. Limiting our search to the Python language, and only looking at repositories with a certain amount of popularity (i.e., number of stars) ensures that we are using high-quality code to generate our AASTs. We cloned around 5000 of said repositories, parsed them using our custom AST parser, added additional edges to form an AAST, and extracted node embeddings from the graph. To generate our node embeddings we concatenate information about (1) the start location of the code for a node, (2) the end location of the code for a node, (3) a word embedding of the type of node (e.g., attribute, import, etc.), and (4) a word embedding of the text a node contains if the node is terminal. (1) and (2) are converted to embeddings using [22] and the word embeddings are all generated using fastText [4].

A.5 Hyperparameter Search Space

The hyperparameter values we searched with are:

Parameter	Values
Batch Size	$[8, 16, 32]$
Epochs	200
Learning Rate	[1e-3, 5e-3, 1e-4]
Weight Decay	[1e-4, 5e-4]
γ (exponential lr decay)	$[0.975, 0.99, 1.0]$
Number of Heads	$[1, 2]$ [6]
Activation	[`LeakyReLU`, `ReLU`, `SELU`, `Sigmoid`, `Tanh`]
Dropout	$[0.0, 0.3, 0.5, 0.7]$
Pooling ratios	$[[0.3, 0.25]$, $[0.25, 0.3]$, $[0.5, 0.3]$, $[0.3, 0.5]$, $[0.5, 0.5, 0.5]$, $[0.3, 0.4, 0.6]$, $[0.5, 0.5, 0.5, 0.3]$, $[0.3, 0.4, 0.6, 0.3]]$
Bias	[`True`, `False`]
Normalization	[`LayerNorm`, `BatchNorm`]

Table 3: Hyperparameter Search Space

A.6 Best Hyperparameters

The best hyperparameters for the GAA model are:

Parameter	Value
Batch Size	8
Epochs	200
Learning Rate	1e-4
Weight Decay	1e-4
γ (exponential lr decay)	1.0
Number of Heads	2[1]
Activation	`LeakyReLU`
Dropout	0.0
Pooling ratios	$[0.3, 0.25]$
Bias	`True`
Normalization	`LayerNorm`

[1]We only had the GPU capacity for a model with 2 heads.

Table 4: GAA Best Hyperparameterse

The best hyperparameters for the U-Net model are:

Parameter	Value
Batch Size	8
Epochs	200
Learning Rate	1e-3
Weight Decay	5e-4
γ (exponential lr decay)	0.95
Activation	`SELU`
Dropout	0.3
Pooling ratios	$[0.3, 0.25]$
Bias	`True`
Normalization	`LayerNorm`

Table 5: U-Net Best Hyperparameterse