TriGraph: A Probabilistic Subgraph-Based Model for Visual Code Completion in Pure Data

Anisha Islam
Department of Computing Science
University of Alberta
Edmonton, Canada
aislam4@ualberta.ca

Abram Hindle
Department of Computing Science
University of Alberta
Edmonton, Canada
hindle1@ualberta.ca

***Abstract*—Pure Data (PD) is a visual programming language for computer music that allows users to create applications through a graph-based, drag-and-drop interface, using objects and connections to manage program flow. There is a lack of tool support for computer musicians using PD, particularly for code completion. In this paper, we introduce TriGraph, a graph-based probabilistic model specifically designed for code completion in PD. TriGraph uses statistical analysis of 2-node and 3-node subgraph frequencies to predict nodes and connections in PD graphs. Using a dataset of parsed PD files, we train and evaluate 5 TriGraph models, assessing their performance in predicting nodes and edges in PD graphs. Our evaluations indicate that the models achieve an average Mean Reciprocal Rank (MRR) score of 0.39 for node prediction, placing the correct answer within the top 3 suggestions, and outperforming the *n*-gram-based KenLM model on similar tasks. For edge prediction, the models achieve an average MRR score of 0.57, with results showing that incorporating both 2-node and 3-node subgraphs yields better results than using only 3-node subgraphs. These findings suggest that TriGraph could enhance the productivity of PD programmers by providing code completion support that may speed up development, reduce errors, and assist in discovering available options. These potential benefits highlight its promise as a valuable support tool for end-user programmers in graphical environments.**

***Index Terms*—Visual Programming Language, Pure Data, Probabilistic Models, Graph Analysis, Code Completion**

I. Introduction

Pure Data (PD) [1], [2] is a popular visual programming language (VPL) for computer musicians to develop musical applications [3]–[8]. Unlike textual programming languages, PD uses a graph-based structure where objects or nodes are arranged on a plane and connected by connections or edges, which manage the flow of instructions between objects. Figure 1 shows a PD program that generates a 516 Hz sine wave, reduces the signal's amplitude by half, and outputs the audio. In this PD program, each rectangular box represents an object, and the connecting lines represent the edges that communicate signals such as control or audio signals.

Computer musicians are categorized as end-user programmers [9], and end-user programmers exceed professional programmers in number [10]–[12]. Despite this, they receive little support tailored to their needs in visual programming languages like PD. Professional programmers using textual programming languages benefit from code completion tools to enhance productivity. In contrast, computer musicians lack similar tools to help navigate the wide range of nodes and connections in PD's graph-like, graphical user interface (GUI)-based environment, making existing code completion tools unsuitable for their needs.

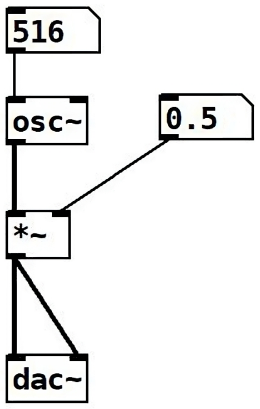


Fig. 1: PD program for generating sound

To address code completion in visual programming languages like PD, this paper presents **TriGraph**, a graph-based probabilistic model that predicts nodes and connections in PD graphs by analyzing subgraph frequency statistics. By focusing on smaller components of the graph, such as 2-node and 3-node subgraphs, TriGraph offers a granular approach to understanding and predicting the overall structure and relationships within the PD graph. We aim to address the needs of computer musicians by providing a method for visual code completion, assisting in predicting the nodes and edges of their PD graphs, thereby potentially improving efficiency.

We use the PD dataset provided by Islam *et al.* [13], partition and sample the PD projects into 5 different training and testing sets, and extract parsed PD files for each dataset. Subsequently, we construct graphs from the list of nodes and connections in the parsed PD files. We next identify unique nodes and count their occurrences. Following this, we extract 2-node and 3-node subgraphs, compute their frequencies, and collect data on all three-node combinations observed in our corpus, regardless of the connections between the nodes. Using our corpus of unique tokens from the training set PD files and the frequencies of the 2-node and 3-node subgraphs, we develop TriGraph, a prediction model for Pure Data graphs based on each of the 5 training sets. This model analyzes the frequencies and connections of these subgraphs, predicting both nodes and potential edges within a 3-node subgraph.

Finally, we assess the performance of our TriGraph models on our test PD graphs by calculating the Mean Reciprocal Rank (MRR) [14] score for each test graph, covering both node prediction and edge prediction. We also compare TriGraph's performance in node prediction with that of the widely used *n*-gram based KenLM Language Model Toolkit [15]–[18], to highlight the effectiveness of our graph-based approach.

We address the following research questions in this paper:

RQ1: In the scenario of a PD graph featuring an unknown node, how effectively can our TriGraph model predict which node will fill the unknown position?

RQ2: Given three nodes in a PD graph that could potentially be interconnected, how effectively can our TriGraph model identify the most probable edges connecting these 3-node combinations?

We found that, our TriGraph models achieve an average MRR of 0.39 for node prediction and 0.57 for edge prediction, meaning the correct predictions typically rank within the top 2-3 positions. In contrast, the order 3 KenLM models, built using paths from our PD files, score an average MRR of 0.30, placing correct node predictions around the 3rd or 4th position. These results show that our graph-based TriGraph model outperforms the KenLM model in node prediction when given similar context. Additionally, we found that the performance of edge prediction is enhanced when both 2-node and 3-node subgraphs are used, as opposed to using only 3-node subgraphs. This indicates that incorporating 2-node subgraphs along with 3-node subgraphs provides a more detailed understanding of the graph structure.

Our contributions can be described in three main points.

1) We introduce TriGraph, a novel graph-based probabilistic model for PD graphs, which predicts nodes and connections using subgraph frequency analysis. This model has the potential to be used for visual code completion in computer music, addressing a gap in support tools for end-user programmers.
2) We empirically demonstrate that our TriGraph model outperforms the *n*-gram based KenLM model in node prediction ranking, based on evaluations with a PD dataset.
3) We show that edge prediction is more accurate when using both 2-node and 3-node subgraphs compared to using only 3-node subgraphs.

Our findings show that probabilistic models effectively predict PD graph structures, addressing challenges in capturing the unique properties of PD graphs and laying a foundation for future research in this area.

II. Related Work

We categorize the relevant literature in four ways, as detailed below.

A. End-User Programmers and Available Support Tools

Visual programming languages enable end-user programmers, who typically have limited or no programming background, to create multimedia applications tailored to their work domains by using drag-and-drop components on a GUI instead of writing textual code [19], [20]. Ko *et al.* [21] defines end-user programmers as individuals who develop applications mainly for personal use, unlike professional programmers who produce code for public use. The number of such users surpasses the number of professional programmers by a 30-to-1 ratio [10], [22]. Despite not being professional programmers, end-user programmers encounter software engineering challenges such as exploring available options in the VPL, testing, and debugging [21].

Examples of visual programming languages designed for specific applications such as game development, real-time and interactive sound synthesis, music transcription, and speech processing include Scratch, Pure Data, and Max/MSP. Recent studies have concentrated on developing software engineering tools tailored to these visual programming language domains. For instance, Scratch users benefit from linters that identify issues and offer suggestions on solving them [23], testing frameworks [24], [25], and similarity measurement between Scratch projects [26]. Studies have also explored clone detection and defect prediction models for VPLs like Max/MSP and Pure Data [9], [27], [28]. While Scratch has hint generation tools [29] and code completion tools [30], no such tools currently exist for Pure Data, which is characterized by its graphical layout.

B. Link Prediction

Link prediction aims to determine whether an edge should exist between two nodes based on the current graph structure and potential future connections [31], [32]. This is particularly relevant to our second research question, as PD graphs are directed, and predicting edges in these graphs aligns with link prediction in graph theory.

Existing link prediction methods include techniques based on similarity scores, probabilistic models, and dimensionality reduction [33]. Wang *et al.* [34] examined knowledge graph embedding models for link prediction to address challenges in incomplete knowledge graphs. Neural networks have also been applied to link prediction, with Cukierski *et al.* [31] using similarity scores and supervised classification on extracted graph features to distinguish real edges from fake ones in social network graphs. Zhang *et al.* [35] utilized graph neural networks to analyze enclosing subgraphs around the links for link prediction.

While traditional approaches rely on similarity scores, graph neural networks, or embeddings, our method prioritizes a simpler, more interpretable edge prediction model aligned with TriGraph's subgraph-based node prediction methodology.

Future research could explore advanced techniques to further enhance the performance of our edge prediction model.

C. Code Completion and Suggestion Using Statistical Language Models

Source code is repetitive, making it statistically modelable by language models for learning patterns that can be valuable in code prediction and suggestion tasks [36]. In the realm of code completion, there is a notable scarcity of tools specifically designed for visual programming languages. A commonly used model for code completion is the *n*-gram model [37], which predicts the next tokens based on the previous *n* - 1 tokens.

Recent studies have applied *n*-gram models for code completion [38]. Raychev *et al.* [38] used the history of API method calls as training sentences for an *n*-gram and recurrent neural network model to generate candidate program completions for partial programs, achieving 90% of predictions in the top three results. However, their work did not address code completion for graph-like visual programming languages, focusing instead on API calls in textual languages. Other probabilistic models, such as decision trees, have also been employed to predict code fragments. For example, Raychev *et al.* [39] used a decision tree model trained on ASTs of programs to predict JavaScript and Python program elements.

Despite these advancements, no work has been done in the visual programming language prediction domain, specifically for Pure Data, using probabilistic graph-based models for code prediction. Additionally, *n*-gram models generally struggle with predicting code segments due to their reliance solely on previous tokens, whereas source code, whether textual or visual, has complex dependencies [40].

D. Code Completion and Suggestion Using Neural Language Models

Representing source code as graphs or abstract syntax trees (ASTs) significantly improves the performance of prediction models [39], [41], [42], which can be beneficial for predicting nodes and edges in PD. However, existing textual code completion tools that utilize ASTs often employ neural networks [43] and deep learning frameworks, which require substantial computational resources. For instance, Ciniselli *et al.* [44], [45] used BERT [46] and transformer-based [47] models for predicting single tokens, sentences, and blocks of code. Kim *et al.* [42] used a transformer architecture based on ASTs for next token prediction in Python. Wang *et al.* [48] used a pre-order depth-first traversal of a flattened AST to predict the nodes of an AST graph using a graph neural network [49] and attention mechanism [50].

While neural networks, deep learning models, and large language models (LLMs) have revolutionized code completion for textual programming languages, they come with their own challenges, such as high computational resource requirements and non-interpretability [51]. Using LLMs should be approached with caution due to the potential for data leakage, output variability, and other issues [52].

III. METHODOLOGY

We utilized the publicly available PD dataset from Islam *et al.* [13], [53], [54] to construct our graph-based probabilistic model for PD code completion. The dataset includes mirrors of the original Git [55] repositories and an SQLite [56] database containing extracted metadata of publicly available 6,534 PD projects on GitHub. This metadata encompasses the revisions of the PD files, parsed source code of PD files, and commit related information from the project repositories [13].

Our goal is to predict nodes and edges in PD graphs by analyzing 2-node and 3-node subgraphs. We divided the PD projects into 5 training and test sets, extracting and parsing PD files for each. From these files, we constructed graphs based on nodes and connections, identified unique nodes like `msg` and `tgl`, counted their occurrences, and stored this data in our corpus. We also counted occurrences of 2-node and 3-node subgraphs and added these to the corpus.

Using the corpus of unique tokens and observations of the subgraphs, we developed TriGraph, a prediction model for PD graphs that uses subgraph probabilities to predict the content of unknown nodes. By analyzing the occurrences and connections within 2-node and 3-node subgraphs, we also predicted potential connections between nodes in a 3-node subgraph, implementing two methods for edge prediction for comparative analysis.

For our evaluation metric, we selected Mean Reciprocal Rank (MRR) because it aligns with our model's ranked output, similar to the ranked suggestions in IDE code completion systems. MRR prioritizes the rank of the correct prediction, effectively measuring how quickly users find relevant suggestions. It also inherently incorporates metrics like *accuracy@k*, *top-k*, and *recall@k* while penalizing lower-ranked answers, making it ideal for evaluating code completion models.

To evaluate our TriGraph model, we treated each PD file in our test set as a graph, replacing each node with a blank to predict the missing element, and predicting the most probable connections for 3-node subgraphs. We assessed performance across all 5 test sets, saving the top 10 predictions for each node and subgraph to calculate the MRR score. Additionally, we compared the node prediction performance of TriGraph with the KenLM language model. For edge prediction, we implemented two versions: one utilizing only 3-node subgraphs and another combining both 2-node and 3-node subgraphs.

To address unseen node combinations in the test set, we applied smoothing, a method that assigns small probabilities to unseen words, preventing the model from assigning zero probability to words missing from the training set but present in the test set [51]. TriGraph adapts traditional *n*-gram smoothing methods, such as Kneser-Ney smoothing [57] and stupid backoff [58], by relying on lower-order subgraph probabilities down to single nodes after discounting when a 3-node subgraph is missing. Unlike the traditional Kneser-Ney smoothing, which uses lower-order *n*-gram probabilities when higher order *n*-grams are missing, TriGraph utilizes lower-order subgraph probabilities due to its graph-based nature. In contrast, KenLM uses modified Kneser-Ney smoothing [59], which is a variation of the original Kneser-Ney smoothing.

The code for TriGraph is publicly accessible to facilitate future research [60]. The training process for our model is divided into three stages: Data Preparation, Graph Construction, and Subgraph Analysis. These stages are described in detail in the following sections.

A. Data Preparation

We partitioned the PD projects into an 80-20 split for training and testing. This process was repeated 5 times, creating different train-test sets each time with a randomly selected set of projects for training and testing. Given the limited availability of public PD datasets, this strategy helped evaluate the generalizability of the model across different training and test sets, making the results more robust.

For each set of train-test sets, we collected the SHA-256 [61] hashes of the parsed PD file contents, which correspond to the revisions of the PD files, for both the training and test projects. Then, we refined the test hashes by eliminating any matches found in the training set to assess our model's performances on paths generated from previously unseen parsed contents. This process guarantees that our training and test data originate from distinct projects, minimizing overlaps and data leakage between them. Our training sets contain an average of 168,912 PD files (graphs), while our test sets include an average of 34,412 graphs across the 5 sets.

B. Graph Construction

Using the dataset provided by Islam *et al.* [13], we parsed the contents of PD files to generate graphs from the PD source code. We extracted a list of connections between the PD file objects using the `Contents` table from the dataset for each parsed content. We constructed a directed graph using the connection information extracted from each parsed content. Notably, in the reconstructed graph, nodes represent object types rather than values or additional parameters.

Following that, we computed the frequency of each unique node, as well as pairs and triplets of nodes within the graph, and recorded the results. For instance, in the first model, our training set includes 34,565 unique nodes, with `msg` being the most frequent, appearing 2,880,151 times across all parsed PD programs. Message boxes are containers for one or more messages, which are transmitted to their designated outlets or destinations upon activation of the box [1], [2].

C. Subgraph Analysis

We collected data on 2-node and 3-node subgraphs from each PD file graph to build our training subgraph corpus. For the 2-node subgraphs, we examined the graph's nodes and edges to understand their connectivity. Then, we applied a depth-first search of length two from each node in the undirected version of the graph to identify subgraphs containing 3 nodes. For instance, for the first model, our training corpus contains 217,806 unique 2-node subgraphs and 1,285,324 unique 3-node subgraphs.

To uniquely identify the 2-node and 3-node subgraphs, we generated a key for each subgraph using the indices of the elements in the subgraph, concatenated with their adjacency matrix. For example, for a subgraph where `msg` connects to `floatatom`, with `msg` having an index of 100 and `floatatom` having an index of 95 in our corpus of unique tokens, the key to uniquely identify this subgraph would be `95,100,0010`. Here, `0010` represents the adjacency matrix between these two nodes. For a 2-node subgraph, we constructed a 2×2 matrix where the first row represents connections originating from the node with the earlier index in our unique tokens corpus, and the second row represents connections from the subsequent node. Similarly, we constructed 3×3 adjacency matrices for the 3-node subgraphs. It should be noted that, during the process of assigning indices to the tokens in our unique tokens corpus, the tokens were sorted in lexicographical order.

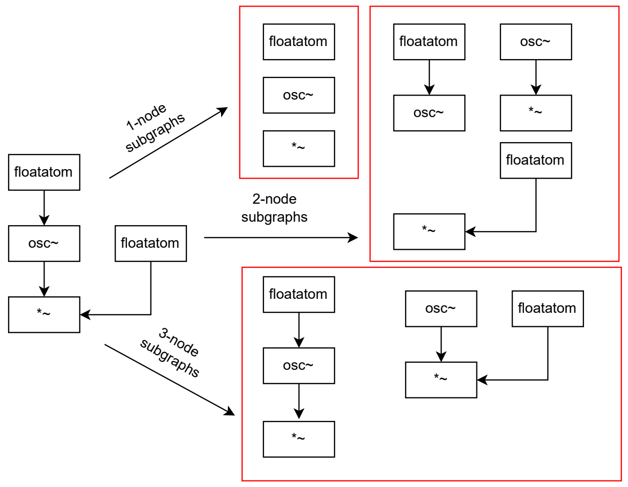
We recorded the frequency of each subgraph to facilitate probability calculations. Additionally, we stored information on all three-node combinations seen in the training graphs to assist in our prediction stage.

For instance, Figure 2 illustrates the corpus creation process, where a PD graph is reconstructed from the parsed content stored in our dataset, along with the extracted subgraphs and their frequencies. The depicted corpus contains 3 unique 1-node subgraphs, 3 unique 2-node subgraphs, and 2 unique 3-node subgraphs. Each subgraph's occurrence count is recorded to facilitate score calculations during prediction. Subgraphs are uniquely identified using the indices of their nodes and their adjacency matrix. For instance, the subgraph `floatatom` → `osc~` is represented by the key `1,2,0100`, where `1` and `2` correspond to the indices of `floatatom` and `osc~` in the example corpus, respectively, and `0100` encodes the adjacency matrix, indicating an edge from `floatatom` to `osc~`.

IV. Training and Evaluating the Baseline KenLM Model

KenLM is a popular *n*-gram-based language model, commonly applied in speech recognition, processing, and related technologies, as well as in machine translation [62]–[66]. It employs modified Kneser-Ney smoothing [59] and is renowned for its efficiency and scalability [15]–[18].

We chose KenLM as the baseline for our node prediction model since both models rely on statistical probabilities for prediction. Unlike our graph-based TriGraph model, which uses graph properties, KenLM predicts nodes based on sequential token context. Despite not being graph-based, KenLM is a suitable baseline, since *n*-gram models have been successfully applied to source code related tasks such as code completion, API method call prediction, syntax error detection, and code template generation [36], [38], [67], [68]. Comparing our model to KenLM provides insights into traditional token-based approaches in visual code completion, emphasizing the benefits of our graph-based method.



Index	1-node subgraphs	#Occurrences
0	*~	1
1	floatatom	2
2	osc~	1

2-node subgraphs	#Occurrences
1,2,0100	1
0,1,0010	1
0,2,0010	1

3-node subgraphs	#Occurrences
0,1,2,000001100	1
0,1,2,000100100	1

Fig. 2: Corpus creation

A. *Choosing an Appropriate N-gram Order*

We began by training the KenLM models using n-gram orders of 3, 4, and 5 for all five training sets. After that, we selected a random subset of 250 PD graphs from the each of the 5 test datasets to evaluate the KenLM models' performance. Figure 3 presents the MRR distribution for models 1–5 across orders 3–5. We observed that the median MRR values for the 3-gram models range from 0.27 to 0.30, while for the 4-gram models, the median is between 0.29 and 0.31, and for the 5-gram models, it ranges from 0.29 to 0.32.

We found that the 4-gram and 5-gram models generally outperformed the 3-gram models, though the difference in median MRR between 3-grams and 4 or 5-grams was less than 0.02 across all 5 models. Additionally, the performance of the 4 and 5-gram models was quite similar, with minimal or no improvement in median MRR. Considering the runtime required to train and test higher-order models, and the relatively small performance gain, we decided to use the 3-gram models for comparison with our graph-based node prediction models. Using a 3-gram model also ensures that KenLM and TriGraph operate with a similar context size, as we used 2-node and 3-node subgraphs for node prediction in TriGraph.

B. *Training the 3-gram KenLM Model*

To train the 3-gram KenLM model, we began by parsing our training graphs and extracting paths by following the edges between objects. Each path is a sequence of tokens separated by spaces. By default, KenLM pads each sequence (or paths, in our case) with start and end tokens. We then used these paths from our training graphs to train an order-3 KenLM model.

C. *Evaluating the 3-gram KenLM Model*

To evaluate the KenLM model for the node prediction scenario, for each test graph and each node within the test graph, we extracted paths ending in that node containing at most 3 nodes, ensuring fairness with our TriGraph model.

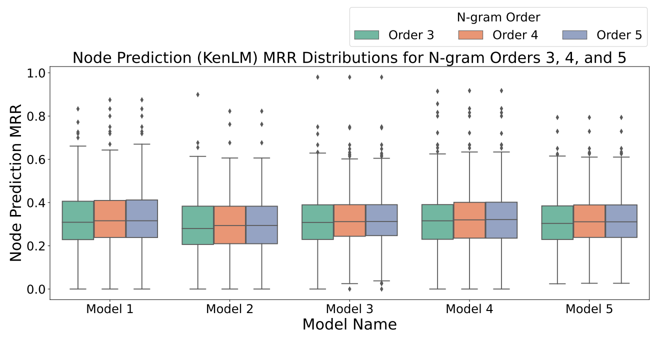


Fig. 3: Node prediction (KenLM) MRR distributions for 3 to 5-grams

For each node in the test graph, we gathered the paths ending in that node and replaced the node with a blank. We then filled the blank with each word from our vocabulary of unique tokens, scoring the paths using the `score` function from KenLM. We maintained a heap of the top 10 tokens with the highest scores and compared the index of the actual node in this heap to calculate its reciprocal rank. This process was repeated for all 5 training and testing sets.

V. Defining, Training, and Evaluating the TriGraph Model

After constructing our corpus of subgraphs, we aim to explore and analyze the predictive capabilities and practical applications of our TriGraph model. We want to understand how well the model can predict missing nodes, and establish the most probable connections between nodes in a PD graph. The following sections describe our research questions in detail, outlining the methodologies used to answer them.

A. RQ1: Predicting an Unknown Node of a PD Graph

Our first research question asks: *"In the scenario of a PD graph featuring an unknown node, how effectively can our TriGraph model predict which node will fill the unknown position?"* This research question highlights the practical application of our graph-based probabilistic model for visual code prediction. By analyzing subgraph probabilities, we predict which nodes are likely to occupy an unknown position in a PD graph, aiding end-user programmers in making informed decisions during their work.

Our approach to node prediction is based on a common method for adding nodes in a PD graph through the PD GUI. In PD, nodes can be added in two ways: (i) selecting them from a drop-down menu, or (ii) creating an empty object box and manually entering the node name. Our work simulates scenario (ii), where users create an empty box but are unsure what to input. Despite a lack of user studies on the most common approach, our node prediction strategy follows a recognized method of adding nodes in PD. Future work could use observations or surveys to confirm how often this method is used to add new nodes.

We first identify all 3-node subgraphs involving the unknown node and score them by their probabilities. By aggregating the predictions from these subgraphs, we select the top 10 candidates with the highest scores to fill the unknown position. The probability of a subgraph is determined using the probabilities of the 1-node, 2-node, and 3-node subgraphs in our corpus.

The probability of an *n*-node subgraph S_n is calculated as:

$$P_n(S_n) = \frac{\text{Occurrences of } S_n \text{ in our corpus}}{\sum \text{Occurrences of all } n\text{-node subgraphs}}$$

where, *n* can be 1, 2, or 3. If a node is not present in our corpus, we assign it a small probability value ϵ. For our model, the ϵ value is as follows:

$$\epsilon = 0.5 \times \left(\frac{1}{\sum \text{Occurrences of all unique nodes}} \right)$$

Here, by `occurrences`, we refer to the number of times a particular item has been observed in our corpus. Additionally, the ϵ values across our 5 models range from $2.66 \times e^{-8}$ to $2.92 \times e^{-8}$.

The algorithm for scoring a subgraph first checks if all nodes in subgraph S exist in the unique token corpus. If they do, it searches for the 3-node subgraph key; if found, the score is set to the corresponding probability. If not, a discount factor δ of 0.05 is applied to penalize the absence, and the algorithm evaluates smaller 2-node subgraphs. The discount factor (0.05) was selected arbitrarily, as it is a commonly used threshold for p-values in statistical tests; however, future research could investigate its impact. For each 2-node connection, it checks for presence in the corpus, calculating the probability if found, or applying the discount if not. If no 2-node subgraphs exist, probabilities for individual nodes are computed, or a small default probability is assigned if nodes are missing. The final score is then returned. The scoring process is summarized by the following equation.

$$P(S_n, n) = \begin{cases} \epsilon & \text{if } n = 0 \\ P_n(S_n) & \text{if } S_n \in \text{corpus}_n \\ \delta \times \prod\limits_{x \in \text{subgraphs}(S_{n-1}, n-1)} P(x, n-1) & \text{else} \end{cases}$$

To identify potential candidates for an unknown node in a subgraph, we search our corpus for nodes that have been previously observed with the other two nodes. For example, in the subgraph `BLANK` $\rightarrow$ `msg`; `BLANK` $\rightarrow$ `floatatom`, we look for nodes seen with `msg` and `floatatom`. We compile these nodes as candidates, place them in the unknown position, and score the subgraph. If no candidates are found, indicating that either or both of the other nodes have not been seen, we iterate through all unique tokens to predict the unknown node.

We maintain a max-heap of size 10 to store the top-scoring candidates from subgraphs containing the unknown node. We store negative log scores to ensure the candidate with the highest negative log score (lowest real score) stays at the top, replacing it if the heap reaches capacity. Finally, we record the index of the correct prediction to calculate the MRR score, assigning -1 as rank if the correct node is not in the top 10.

The algorithm used to determine the rank of the node prediction TriGraph model is outlined in Algorithm 1.

Algorithm 1 Predict an *unknown* node in a PD graph

```
Require: 3-node subgraphs containing the unknown node: subgraphs
Ensure: Rank of predicted token: rank
 1: Initialize heap $H$ with size limit 10
 2: for each $S$ in subgraphs do
 3:     $c_{unknown} \leftarrow$ potential candidate nodes for unknown
 4:     for each $c$ in $c_{unknown}$ do
 5:         $S\_c \leftarrow S$ with $c$ in unknown position
 6:         $P(S\_c) \leftarrow -log(P(S\_c_3, 3))$
 7:         $p \leftarrow (c, P(S\_c))$
 8:         if $|H| >= 10$ then
 9:             if $P(S\_c) < biggest(H)$ then
10:                 $popBiggest(H)$
11:             end if
12:         end if
13:         $Insert(H, p)$
14:     end for
15: end for
16: Sort $H$ by $P(S\_c)$ (ascending)
17: $rank \leftarrow$ index of original unknown node in $H$
18: if original unknown node not found in $H$ then
19:     $rank \leftarrow -1$
20: end if
21: return $rank$
```

For example, suppose we aim to predict the + node shown in Figure 4. The process begins by replacing the node with an unknown placeholder, which our model will attempt to predict. First, we identify the 3-node subgraphs that contain this unknown node. Then, potential candidates for the missing node are selected based on nodes that have previously co-occurred with the other nodes in the subgraph.

Using the corpus in Figure 2, consider the subgraph `msg` $\rightarrow$ `unknown` $\rightarrow$ `floatatom`. The candidate node must be one that has previously appeared with both `floatatom` and `msg`. Since this specific node combination has not been observed in the corpus, we iterate through the entire vocabulary to predict the unknown node. Conversely, if we had encountered a known combination, such as `floatatom` and `*~`, the corpus would suggest `osc~` as the candidate node.

Once candidate nodes are selected, they are inserted into each extracted subgraph and scored using a predefined scoring function. Finally, the 10 highest-scoring candidates are ranked to determine the most likely prediction for the unknown node. This procedure is illustrated in Figure 4.

B. *RQ2: Predicting Edges between the Nodes of a 3-node PD Subgraph*

The second research question aims to identify the most likely method of connecting three nodes within a PD graph. We try to answer: *"Given three nodes in a PD graph that could potentially be interconnected, how effectively can our TriGraph model identify the most probable edges connecting these 3-node combinations?"* By capturing structural relationships between nodes, this research question helps end-user programmers understand PD graph connections and guide data flow based on observed node interactions.

We extract 3-node subgraphs from our test graphs to predict possible connections among their nodes. Two distinct edge prediction models are implemented based on the number of nodes considered. We initialize a max-heap H of size 10. If all nodes of a 3-node subgraph are found in the unique tokens corpus, we retrieve their adjacency matrices and assign them to H. The first model predicts edges based solely on previously encountered 3-node subgraphs. If the heap is empty, indicating the nodes have not been seen together or some nodes are unknown, the model returns -1, signifying an inability to predict unseen node connections.

The second model, however, incorporates both 2-node and 3-node subgraphs. If the heap is empty, it checks for adjacency matrices between pairs of nodes within the subgraph using a map linking 2-node pairs to their adjacency matrices. Potential 3×3 adjacency matrices are generated by combining these matrices, which are then added to the heap. The rank of the true adjacency matrix in the heap is determined and returned; if not found, -1 is returned.

Additionally, to address node pairs in the test set with no previously observed connections, we apply smoothing, assuming a default scenario of no connections. This is treated as if the connection appeared at least once ($adj_{node0,node1} =$ "0000", $count\_adj_{node0,node1} = 1$). This strategy is utilized in the second model, ensuring an adjacency matrix is present for each node pair before calculating the score.

The negative log probability score of each generated matrix is assigned as follows:

$$neg\_log\_prob = -\log\left(\frac{count\_adj_{01} \times count\_adj_{12} \times count\_adj_{02}}{(\sum \text{Occurrences of all 2-node subgraphs})^3}\right)$$

where

$count\_adj_{01}$ = occurrences of the adjacency matrix between node 0 and node 1

$count\_adj_{12}$ = occurrences of the adjacency matrix between node 1 and node 2

$count\_adj_{02}$ = occurrences of the adjacency matrix between node 0 and node 2

The algorithms for predicting connections between the nodes of a 3-node subgraph in both versions of the edge prediction TriGraph model are detailed in Algorithm 2 and Algorithm 3.

Algorithm 2 Predict edges in subgraph: 3-node subgraphs

```
Require: 3-node subgraph $subgraph$, 3-node-to-adjacency matrix map $3\_node\_map = (key, list((adjacency\_matrix, neg\_log\_prob)))$, Unique token corpus $unique\_tokens$, True adjacency matrix: $true\_matrix$
Ensure: Rank of $true\_matrix$: $rank$
1: Initialize heap $H$ with size limit 10
2: if all nodes of $subgraph$ in $unique\_tokens$ then
3:     $key$ ← generate key using node indices from $unique\_tokens$
4:     $adjacency\_matrices_{key} \leftarrow 3\_node\_map[key]$
5:     $H \leftarrow adjacency\_matrices_{key}$
6: end if
7: if $|H| == 0$ then
8:     $rank \leftarrow -1$
9:     return $rank$
10: end if
11: Sort $H$ by negative log probability scores (ascending)
12: $rank$ ← index of $true\_matrix$ in $H$
13: if $true\_matrix$ not found in $H$ then
14:     $rank \leftarrow -1$
15: end if
16: return $rank$
```

For example, suppose `floatatom` $\rightarrow$ `osc~` $\rightarrow$ `*~` is a 3-node subgraph with $\rightarrow$ indicating connections between the nodes. The goal of our edge prediction model is to predict the most likely connection between these three nodes using the corpus, without prior knowledge of their actual connections. The first edge prediction model, as described in Algorithm 2, predicts edges based on previously observed 3-node subgraphs. It begins by searching for connections involving the three nodes in the corpus. In this case, the model searches for connections between `*~`, `floatatom`, and `osc~`, using their indices, which are 0, 1, and 2, respectively, according to the example corpus in Figure 2. The corpus shows two

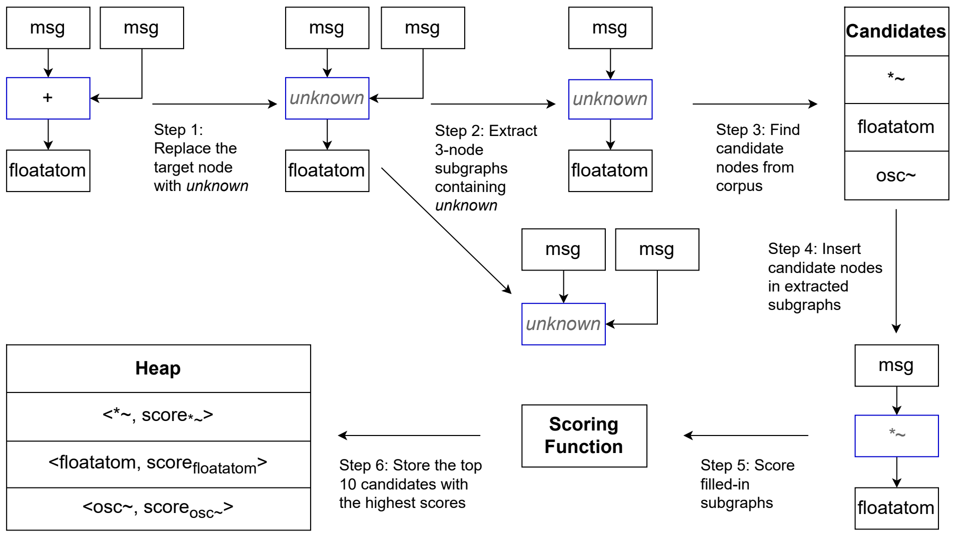

msg
msg
+
floatatom
Step 1: Replace the target node with *unknown*
msg
msg
unknown
floatatom
Step 2: Extract 3-node subgraphs containing *unknown*
msg
unknown
floatatom
Step 3: Find candidate nodes from corpus
Candidates
*~
floatatom
osc~
Step 4: Insert candidate nodes in extracted subgraphs
msg
msg
unknown
msg
*~
floatatom
Step 5: Score filled-in subgraphs
Scoring Function
Step 6: Store the top 10 candidates with the highest scores
Heap
<*~, score*~>
<floatatom, scorefloatatom>
<osc~, scoreosc~>

Fig. 4: Node prediction algorithm of TriGraph

previously observed connections between nodes 0, 1, and 2, with adjacency matrices `000001100` and `000100100`. These matrices are placed in a heap, each assigned a score based on the number of occurrences. The 10 highest-scoring matrices are stored, representing the most likely connections between the nodes. If the nodes have not been observed together, the model returns -1 as the rank for the query.

The second model, as described in Algorithm 3, combines 2-node and 3-node subgraphs to predict possible connections between 3 nodes. If the 3-node combination has not been observed before, the model generates 3×3 adjacency matrices by combining 2×2 matrices of the 2-node subgraphs that involve the pairs of nodes in the 3-node subgraph. For instance, in this case, the model will look for connections between nodes 0 and 1; 1 and 2; and 0 and 2. A default "no connection" scenario is added, and 3×3 adjacency matrices are generated by combining the 2×2 matrices. The model assigns a score to each generated matrix and stores the 10 highest-scoring matrices in the heap. The rank of the true adjacency matrix is determined and returned. If the true matrix is not found, the model returns -1 as the rank for this query. An example of a 3×3 matrix generated by combining the adjacency matrices of nodes 0, 1, and 2 from the 2-node subgraph corpus in Figure 2 is `000101100`. This matrix is derived from the 2×2 adjacency matrices representing the connections between nodes 0 and 1; 1 and 2; and 0 and 2.

VI. Evaluation and Results

This section presents the results of our research questions. For each parsed content from our test hashes, we construct graphs using the methodology from Section III-B. We replace each node with a blank to predict the missing node, following the procedure in Section V-A. Additionally, we extract 3-node subgraphs from our test graphs and predict connections among nodes within each subgraph using the algorithms in Section V-B. We record the index of the correct prediction from the heap and calculate the MRR score for both node and edge prediction scenarios using the following formula:

$$\text{MRR} = \frac{\sum_{i=1}^{N} \text{RR}_i}{N}$$

where N represents the total number of nodes (queries) in the graph for the node prediction scenario and the total number of 3-node subgraphs (queries) in the graph for the edge prediction scenario. The reciprocal rank (RR) of a node is calculated as follows:

$$\text{RR} = \begin{cases} \frac{1}{\text{rank}} & \text{if rank} > 0 \\ 0 & \text{else} \end{cases}$$

In this context, `rank` represents the index of the correct prediction in the heap.

Additionally, we used the Mann-Whitney-Wilcoxon (MWW) test [69] to assess the statistical significance of the differences in results between our models for node and edge prediction. Specifically, the MWW test was applied to compare the MRR scores of all 5 TriGraph models with all 5 KenLM models for node prediction and to evaluate the MRR scores of the two versions of our edge prediction algorithms, determining whether the observed differences were statistically significant. In the following sections, we discuss our TriGraph model's performance in node and edge prediction and compare TriGraph's node prediction performance with the KenLM model.

A. RQ1: Node Prediction Performance

Table I shows that TriGraph achieves a mean MRR value between 0.38 and 0.40 across the 5 test sets, indicating that the

Algorithm 3 Predict edges in subgraph: 2 and 3 node subgraphs

```
Require: 3-node subgraph $subgraph$, 2-node-to-adjacency matrix map $2\_node\_map = (key, list((adj\_mat, neg\_log\_prob)))$, 3-node-to-adjacency matrix map $3\_node\_map = (key, list((adj\_mat, neg\_log\_prob)))$, Unique token corpus $unique\_tokens$, True adjacency matrix: $true\_matrix$
Ensure: Rank of $true\_matrix$: $rank$
 1: Initialize heap $H$ with size limit 10
 2: if all nodes of $subgraph$ in $unique\_tokens$ then
 3:     $key$ ← generate key using node indices from $unique\_tokens$
 4:     $adjacency\_matrices_{key} \leftarrow 3\_node\_map[key]$
 5:     $H \leftarrow adjacency\_matrices_{key}$
 6: end if
 7: if $|H| == 0$ then
 8:     $node_0\_node_1, node_1\_node_2, node_0\_node_2 \leftarrow [], [], []$
 9:     for each pair of nodes (i, j) in [(0, 1), (1, 2), (0, 2)] do
10:         if both $node_i$ and $node_j$ in $unique\_tokens$ then
11:             $key$ ← generate key using node indices from $unique\_tokens$
12:             $node_i\_node_j \leftarrow 2\_node\_map[key]$
13:         end if
14:     end for
15:     for each pair of nodes (i, j) in [(0, 1), (1, 2), (0, 2)] do
16:         $node_i\_node_j.push\_back(\{\text{“0000”}, 1\})$
17:     end for
18:     for $item_{01}$ in $node_0\_node_1$ do
19:         for $item_{12}$ in $node_1\_node_2$ do
20:             for $item_{02}$ in $node_0\_node_2$ do
21:                 $adj\_mat$ ← 3×3 matrix from $item_{01}$, $item_{12}$, $item_{02}$
22:                 $p \leftarrow (adj\_mat, neg\_log\_prob_{adj\_mat})$
23:                 if $|H| >= 10$ then
24:                     if $neg\_log\_prob_{adj\_mat} < biggest(H)$ then
25:                         $popBiggest(H)$
26:                     end if
27:                 end if
28:                 $Insert(H, p)$
29:             end for
30:         end for
31:     end for
32: end if
33: Sort $H$ by negative log probability scores (ascending)
34: $rank$ ← index of $true\_matrix$ in $H$
35: if $true\_matrix$ not found in $H$ then
36:     $rank \leftarrow -1$
37: end if
38: return $rank$
```

TABLE I: Summary statistics of node prediction MRR by KenLM and TriGraph model

	Model Name	Total	Mean	Min	Q1	Q2	Q3	Max
KenLM	1	37,620	0.31	0	0.22	0.29	0.38	1.0
	2	34,303	0.30	0	0.21	0.29	0.37	1.0
	3	39,586	0.31	0	0.23	0.30	0.38	1.0
	4	23,576	0.31	0	0.22	0.30	0.38	1.0
	5	31,070	0.30	0	0.22	0.30	0.37	1.0
	Average	**33,231**	**0.30**	**0**	**0.22**	**0.29**	**0.37**	**1.0**
TriGraph	1	37,620	0.39	0	0.29	0.39	0.48	1.0
	2	34,303	0.38	0	0.28	0.39	0.48	1.0
	3	39,586	0.39	0	0.31	0.39	0.48	1.0
	4	23,576	0.40	0	0.31	0.41	0.50	1.0
	5	31,070	0.39	0	0.31	0.39	0.47	1.0
	Average	**33,231**	**0.39**	**0**	**0.30**	**0.39**	**0.48**	**1.0**

correct prediction usually ranks within the top 2-3 predictions. Additionally, 0.09% to 0.17% of the graphs evaluated across the test sets achieved an MRR of 1.0, while 1.59% to 2.25% received an MRR of 0, highlighting instances where the model failed to predict the nodes correctly. We also evaluated the KenLM model on our test datasets, which yielded a mean MRR score ranging from 0.30 to 0.31, suggesting that the correct node typically ranks within the top 3-4 positions predicted by the model.

Based on Figure 5, TriGraph clearly outperforms KenLM in terms of node prediction performance as shown by the higher median. We also observe that between 0.62% and 1.38% of test graphs across the 5 test sets yielded an MRR score of 0 for KenLM. Additionally, TriGraph achieved an MRR score of 1.0 for 0.09% to 0.17% of test graphs, while KenLM achieved this for only 0.008% to 0.04% of test graphs, indicating our model's superior ability to accurately predict nodes across more test graphs.

We also conducted the MWW test to evaluate whether the difference in MRR scores for node prediction between our TriGraph model and KenLM is statistically significant. Our null hypothesis, *H0: There is no significant difference between the node prediction MRR scores of the TriGraph model and KenLM*, was tested at a significance level of $\alpha = 0.05$. The resulting p-values for MRR scores across all 5 test sets were less than $2.2 \times e^{-16}$, which is below the α threshold. Thus, we reject the null hypothesis and conclude that the MRR scores of the TriGraph model and KenLM are from different distributions. This result suggests that our TriGraph model outperforms KenLM in node prediction, likely due to its use of subgraph analysis instead of a linear path structure.

B. RQ2: Edge Prediction Performance

Table II shows that our second TriGraph model for edge prediction achieves a mean MRR value between 0.55 and 0.61 across the 5 test sets, indicating that the correct prediction typically ranks in the second position according to our model. Additionally, among the evaluated graphs by this model, 3.58% to 5.68% test graphs achieved an MRR of 1.0, while 2.83% to 6.84% graphs received an MRR of 0, indicating instances where the model did not correctly predict the edges.

Furthermore, our first TriGraph model for edge prediction, which considers only 3-node subgraphs, exhibits similar performance, with a mean MRR ranging from 0.54 to 0.60 across

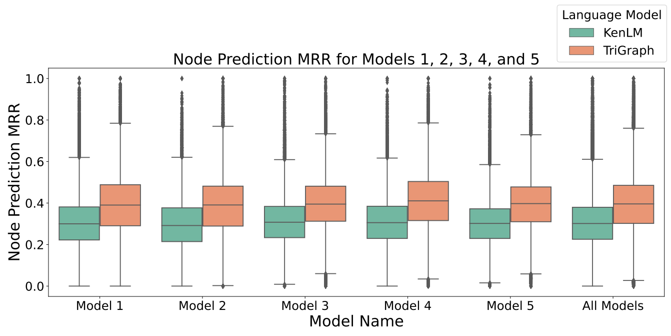


Fig. 5: Comparison of node prediction MRR distributions: KenLM vs. TriGraph

TABLE II: Summary statistics of edge prediction MRR by TriGraph models

Subgraphs Considered	Model Name	Total	Mean	Min	Q1	Q2	Q3	Max
3-node subgraphs	1	37,178	0.55	0	0.38	0.60	0.76	1.0
	2	33,963	0.54	0	0.33	0.59	0.74	1.0
	3	39,101	0.57	0	0.42	0.61	0.76	1.0
	4	23,273	0.60	0	0.48	0.65	0.77	1.0
	5	30,780	0.56	0	0.41	0.60	0.74	1.0
	Average	**32,859**	**0.56**	**0**	**0.40**	**0.61**	**0.75**	**1.0**
2- & 3-node subgraphs	1	37,178	0.56	0	0.39	0.61	0.76	1.0
	2	33,963	0.55	0	0.35	0.60	0.75	1.0
	3	39,101	0.58	0	0.43	0.62	0.76	1.0
	4	23,273	0.61	0	0.49	0.66	0.78	1.0
	5	30,780	0.57	0	0.42	0.61	0.75	1.0
	Average	**32,859**	**0.57**	**0**	**0.41**	**0.62**	**0.76**	**1.0**

the 5 test sets, with 3.03% to 7.48% of test graphs achieving an MRR score of 0, while 3.47% to 5.6% achieving an MRR score of 1.0. This suggests that using both 3-node and 2-node subgraphs for edge prediction improves the results, although the difference is small.

Figure 6 compares the performance of the two versions of models in our edge prediction scenario, demonstrating that the second edge prediction model performs slightly better by incorporating the 2-node subgraphs along with the 3-node subgraphs from our corpus. This could imply that 2-node subgraphs capture simpler patterns of connections and complement the information from 3-node subgraphs, offering a more comprehensive understanding of the graph structure.

In addition, we performed the MWW test to determine if the difference in MRR scores between these two models is statistically significant. We tested the null hypothesis *H0: There is no significant difference between the MRR scores of the two edge prediction models* with a significance level α of 0.05. The p-values from the tests for the MRR scores of both models range from $1.74 \times e^{-8}$ to $3.35 \times e^{-5}$, all of which are below α for all 5 test sets. Thus, we reject the null hypothesis and conclude that the MRR scores from the 3-node subgraph-based edge prediction model and the combined 2-node and 3-node subgraph model come from different distributions. This indicates that using both 2-node and 3-node subgraphs improves performance over using only 3-node subgraphs for edge prediction.

VII. Limitations

There are some limitations to our TriGraph model which are listed below:

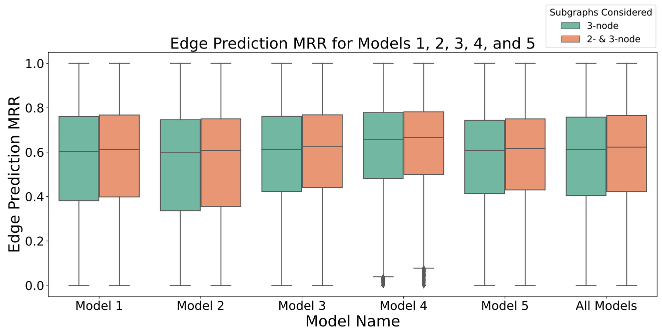


Fig. 6: Comparison of edge prediction MRR distributions based on the type of subgraphs considered (3-node vs. 2- & 3-node subgraphs)

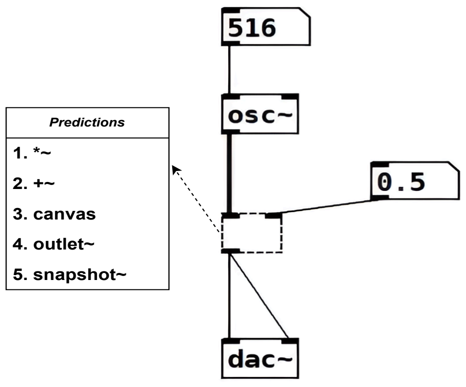
A. *Model Lacks Integration with PD GUI*

Currently, TriGraph functions independently using data from a PD dataset. PD features a GUI that computer musicians use to build musical applications. This model is an initial effort to assist end-user programmers with predictions commonly available to professional programmers through IDE-integrated code prediction models. Integrating TriGraph into the PD GUI is a crucial next step to enhance end-user programming efficiency by providing real-time suggestions. Figure 7 depicts a potential integration of our TriGraph models within the PD GUI for node and edge prediction. In both cases, the displayed predictions represent the actual outputs generated by our model.

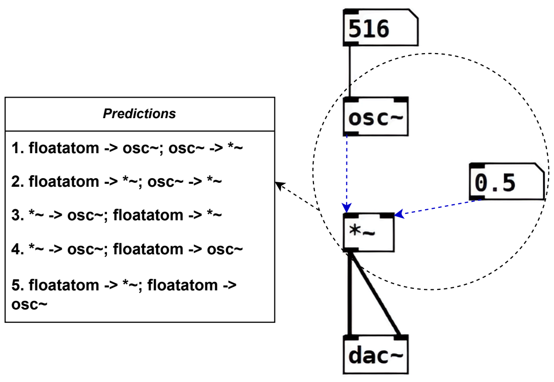
For instance, Figure 7a presents a conceptual integration of our node prediction model within the PD GUI, illustrating a scenario in which a user seeks suggestions for an unidentified node. In this example, the $\star\sim$ node has been replaced with a BLANK, and the predictions generated by our model represent potential candidates for the missing node. This scenario aligns with the motivation behind our first research question, which addresses situations where users create an empty object but are uncertain about the appropriate object to use. By utilizing our model's predictions, users can make informed decisions about the object type to use. Notably, in this example, the correct prediction appears as the top-ranked suggestion, demonstrating the model's capability to provide relevant recommendations.

Similarly, Figure 7b illustrates the edge prediction process within a 3-node subgraph, where the nodes `osc`$\sim$, `floatatom`, and $\star\sim$ have the potential to be interconnected. This example directly relates to our second research question, which investigates how our model can predict the structural relationships between nodes in a PD graph. The blue edges represent the actual connections between these nodes, while our model generates a ranked list of predicted edges. In this particular case, the correct edge configuration appears as the second-ranked suggestion, highlighting the model's ability to capture underlying graph structures effectively.

In this way, by integrating our model with the PD GUI, PD programmers could benefit from live suggestions, enhancing their programming efficiency.



(a) Integration of our node prediction TriGraph model into the PD GUI


516
osc~
0.5
*~
dac~
Predictions
1. floatatom -> osc~; osc~ -> *~
2. floatatom -> *~; osc~ -> *~
3. *~ -> osc~; floatatom -> *~
4. *~ -> osc~; floatatom -> osc~
5. floatatom -> *~; floatatom -> osc~

(b) Integration of our edge prediction TriGraph model into the PD GUI

Fig. 7: Illustration of the likely integration of our node and edge prediction TriGraph models into the PD GUI

B. Corpus Limited to 2- and 3-Node Subgraphs

Our corpus is constructed solely from 2-node and 3-node subgraphs. However, the model's performance might have seen improvement had we incorporated larger subgraphs since doing so would have provided additional contexts for predicting unknown tokens. However, in statistical language models like *n*-grams, 3-grams have proven effective for code completion tasks [36]. Rosenfeld *et al.* [70] mentioned that 3-grams are commonly used for models with millions of tokens, while 2-grams are often used for smaller models. Given that our PD corpus contains approximately a million 3-node subgraphs, we utilized a combination of both 2-node and 3-node subgraphs for our node and edge prediction tasks.

C. Assumptions Regarding Node Connections in a PD Graph

In TriGraph, we assumed that two connected objects have only 1 connection between them. However, PD objects can have multiple inlets and outlets, leading to multiple connections between two objects. While our model simplifies this for code completion, future models could improve upon this.

D. Prediction Limitations of Nodes and Edges

Our current TriGraph model predicts nodes and edges within 3-node subgraphs. While PD nodes are typically associated with additional parameters like text and canvas positions, our model currently focuses on predicting the object type of each node. Future versions could enhance its ability to predict nodes and edges in smaller subgraphs and incorporate parameter predictions for each node.

E. Statistical Language Model-Based Predictions

We build a corpus using subgraph frequencies and predict nodes and edges based on a scoring system derived from subgraph occurrences. This method sets a foundation for predicting visual code structures, and future research could enhance prediction accuracy using neural language models.

VIII. Conclusion

This paper introduces TriGraph, a subgraph-based probabilistic model that presents multiple code completion strategies for the visual programming language Pure Data (PD). TriGraph utilizes statistical analysis of subgraph patterns to predict nodes and connections within PD graphs, with a focus on both 2-node and 3-node subgraphs. By training and evaluating our TriGraph model on a dataset of PD files, we demonstrate its effectiveness in both node and edge prediction. Our TriGraph model achieves an average Mean Reciprocal Rank (MRR) score of 0.39 for node prediction, meaning that the correct suggestion often appears within the model's top 3 recommendations, surpassing the 3-gram KenLM model, which yields an MRR of 0.30. For edge prediction, our TriGraph model attains an MRR of 0.57, suggesting that the correct answer is typically within the top 2 suggestions. Our analysis further indicates that edge prediction accuracy improves when combining 2-node and 3-node subgraphs rather than relying solely on 3-node subgraphs. Furthermore, TriGraph's approach has the potential to be beneficial for other graph-based visual programming languages where edges represent data flow between nodes, such as Unreal Engine's Blueprint or Max/MSP. By offering code completion support designed for visual, graph-based programming environments like PD, our TriGraph model explores the potential to enhance support tools for end-user programmers, representing a step toward improving workflow efficiency for computer musicians.

Acknowledgment

The Natural Sciences and Engineering Research Council of Canada (NSERC) funded this research through their Discovery Grant program (RGPIN-2022-03464).

References

[1] Miller Puckette et al. Pure Data: another integrated computer music environment. *Proceedings of the Second Intercollege Computer Music Concerts*, pages 37–41, 1996.

[2] IEM. Pure Data. https://puredata.info/. Accessed: 2024-01-20.

[3] Pedro Bruel and Marcelo Queiroz. A Protocol for creating Multiagent Systems in Ensemble with Pure Data. In *ICMC*, 2014.

[4] Gilberto Bernardes, Carlos Guedes, and Bruce Pennycook. EarGram: An Application for Interactive Exploration of Concatenative Sound Synthesis in Pure Data. In *From Sounds to Music and Emotions: 9th International Symposium, CMMR 2012, London, UK, June 19-22, 2012, Revised Selected Papers 9*, pages 110–129. Springer, 2013.

[5] Marcos Alonso, Günter Geiger, and Sergi Jorda. An Internet Browser Plug-in for Real-time Sound Synthesis using Pure Data. In *ICMC*, 2004.

[6] Marius Miron, Matthew EP Davies, and Fabien Gouyon. AN OPEN-SOURCE DRUM TRANSCRIPTION SYSTEM FOR PURE DATA AND MAX MSP. In *2013 IEEE International Conference on Acoustics, Speech and Signal Processing*, pages 221–225. IEEE, 2013.

[7] Luiz Naveda and Ivani Santana. "Topos" toolkit for Pure Data: exploring the spatial features of dance gestures for interactive musical applications. In *ICMC*, 2014.

[8] Jamie Bullock and Ali Momeni. ml.lib: Robust, Cross-platform, Open-source Machine Learning for Max and Pure Data. In *NIME*, pages 265–270, 2015.

[9] Gregory Burlet and Abram Hindle. An Empirical Study of End-user Programmers in the Computer Music Community. In *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories*, pages 292–302. IEEE, 2015.

[10] Mohammad Amin Kuhail, Shahbano Farooq, Rawad Hammad, and Mohammed Bahja. Characterizing Visual Programming Approaches for End-User Developers: A Systematic Review. *IEEE Access*, 9:14181–14202, 2021.

[11] Kathryn T Stolee, Sebastian Elbaum, and Anita Sarma. End-User Programmers and their Communities: An Artifact-based Analysis. In *2011 International Symposium on Empirical Software Engineering and Measurement*, pages 147–156. IEEE, 2011.

[12] Christopher Scaffidi, Mary Shaw, and Brad Myers. Estimating the Numbers of End Users and End User Programmers. In *2005 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC'05)*, pages 207–214. IEEE, 2005.

[13] Anisha Islam, Kalvin Eng, and Abram Hindle. Opening the Valve on Pure-Data: Usage Patterns and Programming Practices of a Data-Flow Based Visual Programming Language. In *2024 IEEE/ACM 21st International Conference on Mining Software Repositories (MSR)*, pages 492–497. IEEE, 2024.

[14] Ellen M Voorhees et al. The TREC-8 Question Answering Track Report. In *Trec*, volume 99, pages 77–82, 1999.

[15] Kenneth Heafield. kenlm. https://github.com/kpu/kenlm. Accessed: 2024-03-25.

[16] Kenneth Heafield. KenLM Language Model Toolkit. https://kheafield.com/code/kenlm/. Accessed: 2024-03-25.

[17] Kenneth Heafield. KenLM: Faster and Smaller Language Model Queries. In *Proceedings of the Sixth Workshop on Statistical Machine Translation*, pages 187–197, 2011.

[18] Kenneth Heafield, Ivan Pouzyrevsky, Jonathan H Clark, and Philipp Koehn. Scalable Modified Kneser-Ney Language Model Estimation. In *Proceedings of the 51st Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*, pages 690–696, 2013.

[19] Muhammad Idrees, Faisal Aslam, Khurram Shahzad, and Syed Mansoor Sarwar. Towards a Universal Framework for Visual Programming Languages. *Pak. J. Engg. Appl. Sci. Vol*, pages 55–65, 2018.

[20] Muhammad Idrees and Faisal Aslam. A Comprehensive Survey and Analysis of Diverse Visual Programming Languages. *VFAST Transactions on Software Engineering*, 10(2):47–60, 2022.

[21] Amy J Ko, Robin Abraham, Laura Beckwith, Alan Blackwell, Margaret Burnett, Martin Erwig, Chris Scaffidi, Joseph Lawrance, Henry Lieberman, Brad Myers, et al. The State of the Art in End-User Software Engineering. *ACM Computing Surveys (CSUR)*, 43(3):1–44, 2011.

[22] Margaret M. Burnett and Christopher Scaffidi. End-User Development. Interaction Design Foundation - IxDF, 2014. Accessed: 2024-07-02.

[23] Gordon Fraser, Ute Heuer, Nina Körber, Florian Obermüller, and Ewald Wasmeier. LitterBox: A Linter for Scratch Programs. In *2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering Education and Training (ICSE-SEET)*, pages 183–188. IEEE, 2021.

[24] Katharina Götz, Patric Feldmeier, and Gordon Fraser. Model-based Testing of Scratch Programs. In *2022 IEEE Conference on Software Testing, Verification and Validation (ICST)*, pages 411–421. IEEE, 2022.

[25] Adina Deiner, Patric Feldmeier, Gordon Fraser, Sebastian Schweikl, and Wengran Wang. Automated Test Generation for Scratch Programs. *Empirical Software Engineering*, 28(3):79, 2023.

[26] Lingxi Zhang, Zhiyong Feng, Wei Ren, and Hong Luo. Siamese-Based BiLSTM Network for Scratch Source Sode Similarity Measuring. In *2020 International Wireless Communications and Mobile Computing (IWCMC)*, pages 1800–1805. IEEE, 2020.

[27] Nicolas Gold, Jens Krinke, Mark Harman, and David W Binkley. CLONING IN MAX/MSP PATCHES. In *ICMC*. Citeseer, 2011.

[28] Kalvin Eng, Abram Hindle, and Alexander Senchenko. Predicting Defective Visual Code Changes in a Multi-Language AAA Video Game Project. In *2023 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 485–494. IEEE, 2023.

[29] Benedikt Fein, Florian Obermüller, and Gordon Fraser. CATNIP: An Automated Hint Generation Tool for Scratch. In *Proceedings of the 27th ACM Conference on on Innovation and Technology in Computer Science Education Vol. 1*, pages 124–130, 2022.

[30] Elisabeth Griebl, Benedikt Fein, Florian Obermüller, Gordon Fraser, and René Just. On the Applicability of Language Models to Block-Based Programs. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pages 2374–2386. IEEE, 2023.

[31] William Cukierski, Benjamin Hamner, and Bo Yang. Graph-based Features for Supervised Link Prediction. In *The 2011 International Joint Conference on Neural Networks*, pages 1237–1244. IEEE, 2011.

[32] David Liben-Nowell and Jon Kleinberg. The Link Prediction Problem for Social Networks. In *Proceedings of the twelfth international conference on Information and knowledge management*, pages 556–559, 2003.

[33] Ajay Kumar, Shashank Sheshar Singh, Kuldeep Singh, and Bhaskar Biswas. Link prediction techniques, applications, and performance: A survey. *Physica A: Statistical Mechanics and its Applications*, 553:124289, 2020.

[34] Meihong Wang, Linling Qiu, and Xiaoli Wang. A Survey on Knowledge Graph Embeddings for Link Prediction. *Symmetry*, 13(3):485, 2021.

[35] Muhan Zhang and Yixin Chen. Link Prediction Based on Graph Neural Networks. *Advances in Neural Information Processing Systems*, 31, 2018.

[36] Abram Hindle, Earl T Barr, Mark Gabel, Zhendong Su, and Premkumar Devanbu. On the Naturalness of Software. *Communications of the ACM*, 59(5):122–131, 2016.

[37] Lalit R Bahl, Frederick Jelinek, and Robert L Mercer. A Maximum Likelihood Approach to Continuous Speech Recognition. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, PAMI-5(2):179–190, 1983.

[38] Veselin Raychev, Martin Vechev, and Eran Yahav. Code completion with statistical language models. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 419–428, 2014.

[39] Veselin Raychev, Pavol Bielik, and Martin Vechev. Probabilistic Model for Code with Decision Trees. *ACM SIGPLAN Notices*, 51(10):731–747, 2016.

[40] Chun-Hung Hsiao, Michael Cafarella, and Satish Narayanasamy. Using web corpus statistics for program analysis. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*, pages 49–65, 2014.

[41] Anh Tuan Nguyen, Tung Thanh Nguyen, Hoan Anh Nguyen, Ahmed Tamrawi, Hung Viet Nguyen, Jafar Al-Kofahi, and Tien N Nguyen. Graph-Based Pattern-Oriented, Context-Sensitive Source Code Completion. In *2012 34th International Conference on Software Engineering (ICSE)*, pages 69–79. IEEE, 2012.

[42] Seohyun Kim, Jinman Zhao, Yuchi Tian, and Satish Chandra. Code Prediction by Feeding Trees to Transformers. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pages 150–162. IEEE, 2021.

[43] Chang Liu, Xin Wang, Richard Shin, Joseph E Gonzalez, and Dawn Song. NEURAL CODE COMPLETION, 2016.

[44] Matteo Ciniselli, Nathan Cooper, Luca Pascarella, Denys Poshyvanyk, Massimiliano Di Penta, and Gabriele Bavota. An Empirical Study on the Usage of BERT Models for Code Completion. In *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*, pages 108–119. IEEE, 2021.

[45] Matteo Ciniselli, Nathan Cooper, Luca Pascarella, Antonio Mastropaolo, Emad Aghajani, Denys Poshyvanyk, Massimiliano Di Penta, and Gabriele Bavota. An Empirical Study on the Usage of Transformer Models for Code Completion. *IEEE Transactions on Software Engineering*, 48(12):4818–4837, 2021.

[46] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. *arXiv preprint arXiv:1810.04805*, 2018.

[47] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention Is All You Need. *Advances in Neural Information Processing Systems*, 30, 2017.

[48] Yanlin Wang and Hui Li. Code Completion by Modeling Flattened Abstract Syntax Trees as Graphs. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 35, pages 14015–14023, 2021.

[49] Thomas N Kipf and Max Welling. SEMI-SUPERVISED CLASSIFICATION WITH GRAPH CONVOLUTIONAL NETWORKS. *arXiv preprint arXiv:1609.02907*, 2016.

[50] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. NEURAL MACHINE TRANSLATION BY JOINTLY LEARNING TO ALIGN AND TRANSLATE. *arXiv preprint arXiv:1409.0473*, 2014.

[51] Dan Jurafsky and James H Martin. Speech and Language Processing. 3rd, 2022.

[52] June Sallou, Thomas Durieux, and Annibale Panichella. Breaking the Silence: the Threats of Using LLMs in Software Engineering. In *Proceedings of the 2024 ACM/IEEE 44th International Conference on Software Engineering: New Ideas and Emerging Results*, pages 102–106, 2024.

[53] Anisha Islam. Opening the Valve on Pure-Data Dataset. https://archive.org/details/Opening_the_Valve_on_Pure_Data, 2023. Accessed: 2024-02-19.

[54] Anisha Islam, Kalvin Eng, and Abram Hindle. Opening the Valve on Pure Data Dataset. https://doi.org/10.5281/zenodo.10576757, 2024.

[55] Scott Chacon. Git SCM. https://git-scm.com, 2005. Accessed: 2024-01-26.

[56] D. Richard Hipp. SQLite. https://www.sqlite.org/index.html, 2013. Accessed: 2023-11-12.

[57] Hermann Ney, Ute Essen, and Reinhard Kneser. On structuring probabilistic dependences in stochastic language modelling. *Computer Speech & Language*, 8(1):1–38, 1994.

[58] Thorsten Brants, Ashok Popat, Peng Xu, Franz Josef Och, and Jeffrey Dean. Large Language Models in Machine Translation. In *Proceedings of the 2007 Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning (EMNLP-CoNLL)*, pages 858–867, 2007.

[59] Stanley F Chen and Joshua Goodman. An Empirical Study of Smoothing Techniques for Language Modeling. https://people.eecs.berkeley.edu/~klein/cs294-5/chen_goodman.pdf, 1998. Accessed: 2024-11-06.

[60] Anisha Islam and Abram Hindle. TriGraph: A Probabilistic Subgraph-Based Model for Visual Code Completion in Pure Data (Package). https://doi.org/10.5281/zenodo.14027055, 2024.

[61] FIPS Pub. Secure Hash Standard (SHS). *Fips pub*, 180(4), 2012.

[62] Jun-Ya Norimatsu, Makoto Yasuhara, Toru Tanaka, and Mikio Yamamoto. A Fast and Compact Language Model Implementation Using Double-Array Structures. *ACM Transactions on Asian and Low-Resource Language Information Processing (TALLIP)*, 15(4):1–27, 2016.

[63] Vineel Pratap, Andros Tjandra, Bowen Shi, Paden Tomasello, Arun Babu, Sayani Kundu, Ali Elkahky, Zhaoheng Ni, Apoorv Vyas, Maryam Fazel-Zarandi, et al. Scaling Speech Technology to 1,000+ Languages. *Journal of Machine Learning Research*, 25(97):1–52, 2024.

[64] Shu-wen Yang, Po-Han Chi, Yung-Sung Chuang, Cheng-I Jeff Lai, Kushal Lakhotia, Yist Y Lin, Andy T Liu, Jiatong Shi, Xuankai Chang, Guan-Ting Lin, et al. SUPERB: Speech processing Universal PERformance Benchmark. *arXiv preprint arXiv:2105.01051*, 2021.

[65] Kurt Abela, Md Abdur Razzaq Riyadh, Melanie Galea, Alana Busuttil, Roman Kovalev, Aiden Williams, and Claudia Borg. UOM-Constrained IWSLT 2024 Shared Task Submission-Maltese Speech Translation. In *Proceedings of the 21st International Conference on Spoken Language Translation (IWSLT 2024)*, pages 108–113, 2024.

[66] Nils Hjortnaes, Timofey Arkhangelskiy, Niko Partanen, Michael Rießler, and Francis M Tyers. Improving the Language Model for Low-Resource ASR with Online Text Corpora. In *Proceedings of the 1st joint SLTU and CCURL workshop (SLTU-CCURL 2020)*. European Language Resources Association (ELRA), 2020.

[67] Joshua Charles Campbell, Abram Hindle, and José Nelson Amaral. Syntax Errors Just Aren't Natural: Improving Error Reporting with Language Models. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, pages 252–261, 2014.

[68] Ferosh Jacob and Robert Tairas. Code Template Inference Using Language Models. In *Proceedings of the 48th Annual Southeast Regional Conference*, pages 1–6, 2010.

[69] Henry B Mann and Donald R Whitney. On a Test of Whether one of Two Random Variables is Stochastically Larger than the Other. *The Annals of Mathematical Statistics*, pages 50–60, 1947.

[70] Ronald Rosenfeld. Two Decades of Statistical Language Modeling: Where Do We Go From Here? *Proceedings of the IEEE*, 88(8):1270–1278, 2000.