UNIVERSITY OF OKLAHOMA

GRADUATE COLLEGE

EXPLORING LATENT PROGRAM SPACES FOR PROGRAM SYNTHESIS

A THESIS

SUBMITTED TO THE GRADUATE FACULTY

in partial fulfillment of the requirements for the

Degree of

Master of Science

By Kendall Tauser

Norman, Oklahoma

2025

EXPLORING LATENT PROGRAM SPACES FOR PROGRAM SYNTHESIS

A THESIS APPROVED FOR THE
SCHOOL OF COMPUTER SCIENCE

BY THE COMMITTEE CONSISTING OF

Dr. Richard Veras, Chair

Dr. Jie Cao

Dr. Andrew H. Fagg

Contents

List of Figures

List of Tables

Abstract

Formal grammars are the canonical means of describing a space of programs. The finite set of rules describing the space can also be used for sampling programs within this space. One can formulate this system as a reinforcement learning problem where one represents non-terminals as states and production rules as actions. The problem then becomes how to represent a partially completed program in an effective manner for such a model working to build programs. This thesis looks into sampling programs from various domain specific languages and constructing continuous embeddings of such programs to serve in downstream machine learning tasks, including for program expansion. Qualitative and quantitative analysis of doc2vec-based embeddings is done along with development of a quantitative metric for analyzing how effectively embeddings retain the structure of partial and complete programs, with comparisons to other text-based embedding systems.

Keywords: program synthesis, continuous representations, embeddings, formal languages, reinforcement learning, machine learning

Chapter 1

Introduction

One of the holy grails of the computing field is to acquire the technology such that computers are able to program themselves, and it is an objective that has seemingly been in the crosshair of the computing community since its inception. As will be discussed, this goal has typically taken shape in the sub-field of program synthesis, wherein various techniques have been employed to build software that writes other (typically very limited) software.

More recently, the advent of large language models (LLMs) [1, 2, 3, 4] have captivated the computing community, including in the program synthesis field. But transformers are certainly not perfect in multiple regards. The biggest is the vast quantities of compute that are necessary to generate code that may or may not compile, both at training and inference time.

One problem with utilizing LLMs for program synthesis is that the token space of an LLM is not congruent with the tokens/syntax of a programming language. As a result of this, for an LLM to write programs, both the syntax of the language must be learned along with the semantics. This results in additional computational burden overall. One can then formulate the question: can we reduce computational overhead by hard-coding the syntax of a language into our model architecture, and then try and learn good semantics on top of it? This thesis attempts to look into that question, but found that there are prerequisite questions that precede answering that question fully. Specifically, the original aim was to create programs starting from a formal grammar, and expanding such grammar to generate programs. This process can be formalized roughly as a Markov Decision Process (making it suitable for reinforcement learning) but without the assumption of a particular state n being independent of all previously visited states $n-1...0$ (i.e., the best expansion decision is likely to be dependent on previous states).

My contribution in this thesis is working on means to account for the need for representing this context when making expansion decisions. The proxy for this context is to construct program embeddings, or creating a mapping from either a completed

or partially completed program abstract syntax tree to a continuous vector. These embeddings can be used in downstream machine learning tasks and act as a proxy for storing program context to inform the next best expansion decision.

Given the motivation to construct these embeddings, a few second-order questions can be derived:

- How are spaces of embedded programs organized? Are they simply random? If there is structure, what are the main features of programs that are extracted to form clusters?
- Can one interpolate between partially-completed programs and completed programs within the space easily?
- Is there any way to assess the fitness of these spaces quantitatively in a general way?
- Based on any of these metrics that may or may not exist, what, if any, embedding scheme is the best per that metric?

The goal behind this research is to develop techniques for measuring how well we can retain structural properties of a program in a continuous representation.

This operates under the assumption that AST structure can be an analogue for program semantics. Since these embeddings need to serve as context for program expansion, then the desire is the semantics of any partially completed program are retained in any embedding representation. If one can retain this structure in an embedding, then this can serve as a domain for performing reinforcement-learning based program synthesis in future work.

To complete this work, I propose a software framework called called lang-explorer, which provides machinery for defining grammars (context-free or context-sensitive), sampling programs from such grammars, modules for constructing embeddings from such datasets, and machinery for adding reinforcement learning implementations to generate sampled programs from the same grammars in the future.

Lang-explorer is designed to be extendable with new embedding and expansion techniques. The grammar construction system allows one to construct any context-free or context-sensitive grammar, with the ability to parametrize individual grammars. The grammar expansion tooling allows for adding new modules to make program expansion decisions within the framework (including RL modules). The embedding framework supports adding new modules for creating continuous representations of programs beyond what has been implemented for this thesis. This functionality is exposed as a command line interface tool.

This thesis looks into the details of this framework, and the methods and experiments used to construct program embedding spaces and analyze them qualitatively. This work also includes descriptions and results for a quantitative pairwise embedding similarity metric based on the Weisfeiler-Lehman graph isomorphism test [5]. The Weisfeiler-Lehman isomorphism test and corresponding kernel are able to extract a set of labels from a graph, and the similarity between such sets provides an approximation of the similarity between the original graphs. In my case, the kernel and metric can be used to find the similarity of abstract syntax trees (ASTs). This metric is used within the experiments of this work to compare the embedding systems that were tested with various domain specific languages defined within the lang-explorer framework.

To reiterate more concretely, this thesis offers the following contributions:

- A software framework (lang-explorer) written in Rust [6] for defining formal grammars, with machinery to construct generators to sample programs from the defined program space, derive latent representations for defined languages, and implement machine learning models for expanding grammars to produce programs.
- A re-implementation of Doc2Vec [7] for creating embeddings of ASTs generated from lang-explorer. This system is implemented as an embedder module directly in lang-explorer.
- A qualitative analysis of generated embeddings of various domain-specific languages and their properties.
- A quantitative comparison between Doc2Vec based program embeddings alongside various modern text-based embedding models.

Chapter 2

Literature Review

2.1 Grammars

Grammars are an incredibly useful construction for describing various program spaces. Languages ranging from simple toy languages all the way to production programming languages can be represented using grammars and the rules encoded within them. This expressive power comes from a finite number of rules that can be used to define a (potentially) infinite space of possible programs.

The most common representation of a grammar is with the tuple $\mathcal{G} = (\mathcal{V}, \mathcal{T}, \mathcal{S}, \mathcal{P})$, where $\mathcal{V}$ is a set containing all non-terminal symbols used within the production rules, $\mathcal{T}$ is the set of all terminal symbols used within the production rules, $\mathcal{S} \in \mathcal{V}$ is a symbol that is denoted as the start symbol, and $\mathcal{P}$ is the set of production rules of the form $\alpha A \beta \rightarrow \alpha \gamma \beta$, where $\alpha, \beta \in \{\mathcal{V} \cup \mathcal{T} \cup \epsilon\}^*$, $\gamma \in \{\mathcal{V} \cup \mathcal{T} \cup \epsilon\}^+$ and $A \in \mathcal{V}$ (please note ϵ represents henceforth the empty string).

This specifically represents the structure of a context-sensitive grammar (CSG), which is a superset of other simpler languages such as context-free and regular languages. For example, context free languages/grammars [8, 9] (CFGs) can be represented by context-sensitive grammars if one adds the constraint that $\alpha, \beta = \emptyset$. Additionally, regular languages can be represented by context-sensitive grammars if $\alpha, \beta = \emptyset$ as well as the right-hand sides of each production rule are either right regular or left-regular (i.e., production rules are either of the form $\alpha A \beta \rightarrow \alpha B \gamma \beta$, $\alpha A \beta \rightarrow \alpha \gamma \beta$, $\alpha A \beta \rightarrow \alpha \gamma B \beta$, or $\alpha A \beta \rightarrow \alpha \gamma \beta$, where $A, B \in \mathcal{V}$ and $\gamma \in \mathcal{T}$). There exist more general grammars, but their definition is omitted here for brevity. Context-sensitive and context-free grammars are the primary subsets of grammars that are considered in this thesis.

A grammar can be expanded, where one begins at the start symbol, and chooses from the available production rules for such symbol to replace the current symbol with additional symbols. This is done recursively until only a sequence of non-terminals

remain, where the sequence of terminals is the completed program.

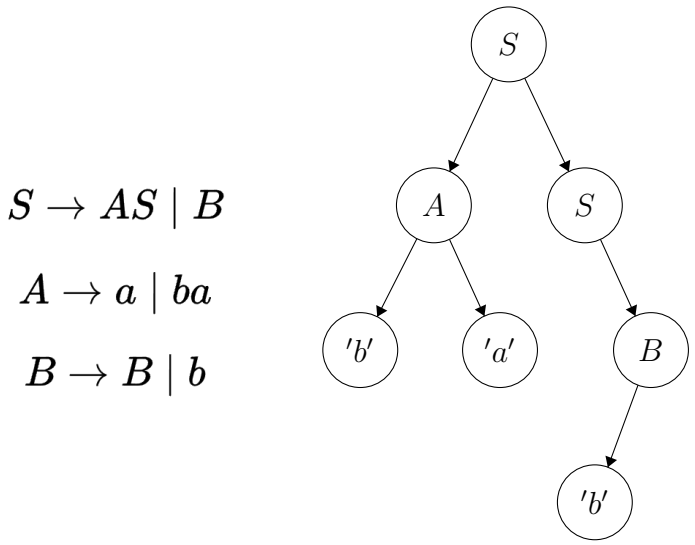


Figure 2.1: A simple grammar with a sample expansion tree to the right.

Each level of the tree represents a new set of terminals and non-terminals that replace their parent within the expansion. The first row of nodes after the root represents expansion of rule 1 using the 1st production, where S is replaces with A and S, the expansion of A to 'a' and 'b' represents rule 2. The expansion of S to B represents the expansion of the 2nd production of rule 1. Finally, the expansion of B to 'b' represents rule 3. After all non-terminals have been resolved to one or more terminals, one can traverse through the tree with in-order traversal to retrieve the output program of 'bab'. The tree shown above can be referred to as the equivalent AST representation for the program 'bab'.

Many (if not all) programming languages are formally defined [10, 11, 12], in that there exists a grammar of some form that describes the entire space of all possible programs within that language. This is very useful, as it means one can create syntactically valid programs within a program space by recursively expanding the rules of the grammar until a final sequence of terminal tokens is emitted.

2.2 Program synthesis

There exist significant quantities of research has been done in the field of teaching computers to program themselves; this section hopes to provide a brief survey of the rich corpus of research that has preceded this document. The two main paradigms in the literature regarding how to learn good programs is either using the supervised approach or a reinforcement learning/reward-based approach. The supervised approach typically involves a learner (which could be an RL agent) being given a program input, and being tasked with creating a program that transforms the input into a desired

output (the label) [13, 14, 15]. The RL/reward approach involves a learner creating a program, and then receiving a reward based on some metric (this could be the performance of the program, some quantitative measure of its correctness, or whether the program even compiled/executed at all), and learning to maximize reward. This technique is almost exclusively associated with reinforcement learning agents. This section of the literature review will provide some brief insight into the work that has been done around both of these paradigms.

2.2.1 Supervised Methods

There has been work to utilize deep neural networks in the realm of program synthesis. Deepcoder [13] tries to learn a mapping between input-output pairs and the set of functions comprising a program that satisfies the input-output pair. One example of this could be data input/output pairs generated from a program that is sorting the provided sequence. The aim of DeepCoder is to maximize the probability of predicting the presence of the Sort() operator within a ground truth program generated from their custom DSL. The custom DSL is designed to provide utilities that one would require for solving competitive programming programs. Once a program P has been generated, P can be executed with a random input i to generate an output o, such that $P(i) = o$. The DeepCoder model then encodes i and o into a series of latent vectors. The latent vectors are given as input to the model, which will emit a probability distribution of the presence of attributes of the ground-truth program P. Deepcoder specifically casts the problem of inductive program synthesis into a supervised one, which can be helpful for generating datasets, but limits the generalization of the technique.

2.2.2 RL Reward Methods

There has been more research around using reinforcement learning as the system for performing program synthesis. Specifically, work is based around formalizing a grammar as an MDP, where (in the case of a context-free grammar) non-terminals are states, and the production rules corresponding to each non-terminal are actions. The role of the reinforcement learning agent is to take actions to expand the grammar until a complete program is generated. The program can then be executed and a metric derived from it's execution can be fed back into the model as a reward, and the model can adjust its behavior accordingly. One can also use input-output pairs as a reward signal that can be fed back into a policy network. There are multiple variants of this paradigm in the literature.

The has also been work to use the above pattern of reinforcement learning to expand a grammar, but use an additional deduction engine as a means of pruning the program search space by eliminating infeasible partial programs [16]. The problem that

the authors work to solve is transforming sequences of input scalars in some way to match an output sequence (the label), and this transformation is done by executing the synthesized program. The deduction engine stops the expansion of infeasible partial programs by detecting when it will be impossible for the given program to ever satisfy the given input/output specification. This deductive feedback is incorporated into the computation of the policy gradient in addition to statistical feedback.

Other works have utilized tree search to look through the program space and choose optimal programs [15]. The goal in this paper was to synthesize a subset of the RISC-V instruction set, so synthesis was regarded less as a synthesis of a tree of terminals and non-terminals, but as a tree corresponding to different sequences of machine instructions. The policy would then choose actions on this priority search tree until the reward of the synthesized program was above some threshold, and the resulting program would be regarded as a solution.

There have also been concerted efforts to explicitly predict tokens from a formal language alphabet Σ as a means of performing program synthesis [14]. This was more common when recurrent models like LSTMs and RNNs were more common in cutting-edge literature.

Additional research exists around utilizing program synthesis as a general means of problem solving. The aim of Dreamcoder [17] is to both learn a domain specific language, as well as learn heuristics and intuition in order to create programs that can solve problems. The key notion that they take advantage of is the idea that learning can be represented via program induction, or that the extrapolation of new programs can be a form of learning to solve new problems from priors. As the authors note, this representation is useful because knowledge encoded as programs is inherently human-readable, and there is an element of universality to programs in that they can represent any Turing-complete computation.

2.3 Large Language Models

Large language models have gained significant attention in the past years as various facets of transformer technology and scale have improved to produce models that can write code, with various degrees of success. Models such as OpenAI's ChatGPT [1], DeepSeek's V3 [4], DeepSeek R1 [18], Facebook's Codellama [3] and general purpose Llama [2], and others have made strides to be better at general purpose next-token prediction, and this includes creating code. These models produce tokens by forming a matrix E where the columns are embeddings of all tokens within the context (i.e., tokens issued in the prompt as well as those generated by the model) and the number of rows is equivalent to the embedding dimension. This context window flows through the model and generates a probability distribution over the available tokens. The

model then samples from this distribution to decide upon the next token, and the process repeats with the latest token added to the front of the context matrix until a termination token is emitted.

The primary mechanism utilized in these models to generate such probability distribution is the attention head [19]. Specifically, multi-head attention, where token embeddings are duplicated among multiple attention heads and are concatenated together once attention has been performed. Within an attention head, the context matrix E is linearly projected into Query Q, Key K, and Value V matrices using learned weight matrices. As per the original paper, attention is defined as the following:

$$\text{Attention}(Q, K, V) = \text{softmax}\Big(\frac{QK^T}{\sqrt{d}}\Big)V,$$

where the softmax operation is performed column-wise against $\frac{QK^T}{\sqrt{d}}$. The idea behind this is that tokens can "attend" to each other (wherein the inner product is performed between all queries and keys corresponding to columns in E) in a permutation-invariant way. This results in E encoding richer relationships between tokens, and leads to better next-token prediction.

Although these models have shown themselves to be capable to producing remarkable outputs, there are still concerns. Namely, the problem of hallucinations, or models creating facts or statements that do not correspond to factual reality, persists [20, 21, 22]. There are also concerns about the long-term scalability of the transformer paradigm given the sheer quantity of high-quality data that is required to train them, and the need for significantly more data to achieve greater generalization and reasoning capabilities [23].

2.3.1 Large Language Model-Based Methods

There is substantial research around using large language models (LLMs) and transformer architectures for doing program synthesis. There exist instances of some of the frontier LLMs that are specifically fined-tuned for coding. Some examples of this are CodeLlama [3], DeepSeek Coder [24], Qwen2.5-Coder [25], and many more. There also exist more specialized LLMs that are fine tuned for a specific language or DSL. A recent example of this is StarCoder [26], which is a code LLM specifically fine-tuned for Python. StarCoder was trained on over 35 billion Python tokens, and used the standard transformer architecture to learn to predict Python tokens given a prompt.

There is recent research taking place wherein LLMs generate code through an evolutionary approach, and LLMs serve as full-fledged coding agents. Agents accomplish this by iteratively generating code and calling tools to test the code, and repeating based on test feedback until better results are achieved [27, 28].

It is worth noting that all of these examples parse and produce code in terms of sequences of tokens, foregoing any hierarchical structure to the code, since attention mechanisms compute on token sequences.

2.4 Embeddings

Neural networks need fixed-length, (typically) real-valued tensors of data to serve as their inputs, in spite of the fact that there are no guarantees that a given problem domain will be able to supply data in such a form. This is relevant if your data is categorical/discrete. One example of this is natural language processing, where the input is typically one or more discrete words, and it is the researcher's duty to attempt to extract patterns or some other insight from such input. One typical solution is to employ one-hot-encoding [29, 30]. This is where a unique vector of the form $\vec{v} =< 0, 0, ...0, 1, 0, ..., 0, 0 >$, is created for each of the n categories. The k'th element of $\vec{v}$ is a one, while all other elements are 0, and $|\vec{v}| = n$. One-hot encoding can be a useful encoding, but if n is very large, then it can become intractable to utilize this approach.

This is where more advanced techniques can be employed to create vectors of fixed, smaller dimension. Instead of having one element represent each category, you attempt to learn from data how to represent more features within a lower dimensional space. Embeddings are an example of this, where one is typically trying to learn an approximate mapping of the form $f : S \rightarrow \mathbb{R}^k$, where $k << n$ and S is the set of all categories/data-points. The space that these embeddings reside within are called an embedding space. Creating these vectors by hand can be slow and tedious, and thus machine learning can and has been utilized to learn f from data through various training techniques.

The aim is that these learned embeddings encode segments of the semantics of the original data in particular directions of the space. This means that moving through the space (i.e., performing vector algebra on elements of the latent space) corresponds to some change in the semantic meaning of a corresponding entity in the original dataset. The colloquial example given within natural language processing literature is that embeddings should encode the semantics of words such that expressions such as $vector(\text{"}King\text{"}) - vector(\text{"}Man\text{"}) + vector(\text{"}Woman\text{"}) \approx vector(\text{"}Queen\text{"})$ [31] (or, my example of $vector(\text{"}Computer\text{"}) - vector(\text{"}Monitor\text{"}) \approx vector(\text{"}Server\text{"})$) hold to be approximately true.

If these representations are approximately accurate mathematical representations of the target domain, then such vectors can be used as a useful analogue for the domain that can be used within machine learning models.

The field of embeddings research is complex, with much work done to create la-

tent representations of various distributions of data structures. They can be used for domain specific tasks such as creating latent representations of Structured Query Language (SQL) database queries [32], encoding assembly instructions for basic block throughput estimation [33], encoding of symbols within a formal grammar [13], and encoding tokens for LLMs, and much more.

A common technique for creating embeddings is utilizing autoencoders [34]. This model paradigm involves having a neural network learn a continuous representation of some data (e.g. graphs, documents, photos, audio, etc.). Autoencoders are typically implemented as a self-supervised algorithm, where a model learns to predict the same output as the input, but where there exist tensors in the middle layers of the model that are extracted and serve as continuous representations of the input data. This can be considered a more general-purpose embedding framework, assuming one has the means for vectorizing the input data or otherwise encoding it before feeding it through the model.

2.4.1 word2vec

There exist more domain-specific embedding techniques. For example, the canonical word2vec [31, 35] learns a distributed representation for a corpus of words based on the relation between a prediction word and its surrounding context words found within the corpus. Word2vec proposes two model architectures to learn such embeddings: skip-gram and continuous-bag-of-words (CBOW).

In the skip-gram model, the training objective is to maximize the probability of predicting context words given a center word (where the count and position of context words can be tuned). This happens by passing the center word vector through a shallow classifier which yields a probability distribution of a given context word. This is repeated several times to predict each context word within a certain range of the center word. The model can then be trained through backpropagation to increase the probability of choosing the correct context word and reducing the probability of choosing the wrong word. Embedding weights are updated through this process, and the end result is a matrix of vectors that encode the semantics of the words they correspond to.

Conversely, the continuous-bag-of-words implementation aims to maximize the probability of predicting a center word given a set of context words (which can also be tuned, both in quantity and relative position). All context words vectors are optionally concatenated or averaged together to create a single vector that is then passed through a shallow classifier to again yield a probability distribution of the center word. Once again, the model learns to maximize the probability of predicting the correct center word, which updates the embedding weight matrix and hopefully yields good embed-

dings at the end of the training process.

It may be useful to think of these two models of being the inverse of each other, the inputs of one are the predicted outputs of the other and vice-versa. But, the core classifier and objective function are the same, and back-propagating changes to the embedding matrix is the method for constructing the resulting embeddings.

In regards to loss for classification, there are multiple loss functions that were originally proposed in the paper. One of the loss functions provided is called negative sampling, which is of the form:

$$\text{loss} = -\log \sigma(\mathbf{v}_{w_O}^T h) - \sum_{w_j \in \mathcal{W}_{\text{neg}}} \log \sigma(-\mathbf{v}_{w_j}^T h),$$

where $\mathbf{v}_{w_O}$ represents the vector in the classifier layer of the true word (i.e., the word in the corpus we are trying to predict given the context), and $\mathbf{v}_{w_j}$ where $w_j \in \mathcal{W}_{\text{neg}}$ represent the columns in the classifier matrix corresponding to a sample of words of size q (q being the number of negative words we wish to sample) that are not the true word. h is the input embedding vector of the input word in the case of the skip-gram model, but of course in CBOW there are multiple input words such that the input vectors are averaged, giving $h = \frac{1}{C} \sum_{c=1}^{C} \mathbf{v}_{w_c}$.

The goal behind this loss is to maximize the probability of predicting the positive word while minimizing the probability of predicting a sample of negative words (words in the corpus that are not included in the context around the center word). The rationale behind updating only a subset of negative words is due to the fact that the dimension of the output layer corresponds to the number of words in the corpus, which could be vast, and very expensive to compute. Thus the authors chose to sample a subset of negative words as a trade-off between learning efficiency and computational tractability.

The words that word2vec learns are typically natural language words and phrases, but as will be shown, this can generalize to be other sets of symbols.

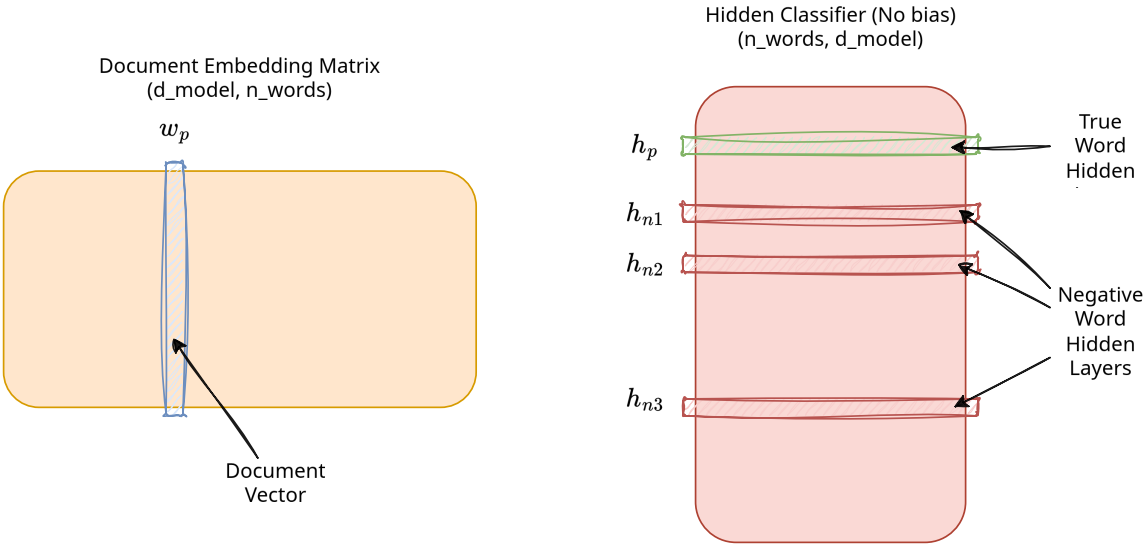
2.4.2 doc2vec

An extension of word2vec is doc2vec [7], which is able to construct embeddings for entire documents of words (which are referred to in the literature as paragraph vectors). Similarly to word2vec, doc2vec proposes two model architectures for learning paragraph vectors: distributed memory and distributed-bag-of-words.

In the distributed memory implementation, the training objective is to maximize the probability of predicting a center word in a document given both context word vectors (configurable in size) as well as the paragraph vector. The context word vectors and paragraph vector are optionally concatenated or averaged together, and

then passed through a shallow classifier yielding a probability distribution of the center word. Just like in word2vec, the model learns to maximize the probability of predicting the center word, but in this case updates both the word *and* paragraph vectors through backpropagation to create the output document embeddings.

The distributed-bag-of-words implementation of doc2vec similarly looks to maximize the probability of predicting words (in this case, a configurable number of context words) from a document. This is done by passing a paragraph vector through a shallow classifier to again predict the probability of the given context words. The model is trained to maximize the probability of predicting each of the context words, updating the paragraph vector as a result. At the end of training, one is left with a set of paragraph vectors that encode the semantics of the training corpus.



$$\mathcal{L} = -\log \sigma(w_p \cdot h_p) - \left(\log \sigma(-w_p \cdot h_{n1}) + \log \sigma(-w_p \cdot h_{n2}) + \log \sigma(-w_p \cdot h_{n3})\right)$$

Figure 2.2: Implementation details of doc2vec DBOW. The task of the model is to perform a linear transformation on the positive word as well as negatively sampled words. The sample of negative words is to reduce the costs of backpropagation if there are many words in your corpus.

There exist alternatives to doc2vec for creating paragraph vectors. One potentially easier method is to simply create a weighted average of the vectors of the words comprising the document. But, the downside of this is that the order of the words and relative positions are lost in the training process, and the authors in [7] indicate that those are relevant features of the text that are worth being encoded. This makes sense intuitively, as well, as there are many instances in which the ordering of words within a document affects the semantics of the text.

Similar to word2vec [31], doc2vec can use the same loss functions to maximize the probability of predicting a true word while minimizing the probability of predicting a

negative word. This includes negative sampling as was described in section 2.4.1. The same loss can be utilized for the distributed memory model, as we concatenate/average the input word vectors and document vector similarly to the CBOW implementation of the negative sampling loss function. The same can be said for using negative sampling for the DBOW model with the original skip-gram implementation.

2.4.3 graph2vec

The same technique is applied in graph2vec [36], which is used to create distributed representations of graphs. Graph2Vec is a special implementation of the doc2vec distributed bag of words model, where each graph is a document, and labels corresponding to subgraphs within the graph are considered the "words" contained within. Graph2vec relies on the Weisfeiler-Lehman graph kernel [5] as a means for creating subgraph labels to attach to any given graph in the corpus. The WL-kernel algorithm is shown in Algorithm 1, and a visual representation of the same algorithm is shown in Figure 3.3. Graph2vec constructs these word sets from graphs by iteratively creating new node labels for every node based on an ordered concatenation of the current node label along with all of its neighbor's labels in the current iteration. After each iteration, the new labels of all nodes are added to a found list, and the process is re-run until all iterations are complete. A similar process for reference is the process of "message-passing" as is found in traditional graph neural networks [37, 38, 39].

By iteratively passing labels between nodes and creating new labels that encode information about the labels of all neighbors, one is incrementally passing more information between nodes about the overall location of g in the graph as a whole and its relation to other nodes. This allows labels of all subgraphs of diameter $\leq k$ to be captured in the set of all labels that are created through this iterative process. The set of output labels becomes the set of "words" that describe g within the graph2vec algorithm.

Embeddings are learned for each graph (document) using the provided words (subgraph labels) using the same algorithm as described in doc2vec [7].

2.4.4 code2vec

Code2vec [40] is a specialized embedding algorithm to learn distributed representations of code ASTs. The goal of code2vec is to learn to maximize $P(\mathcal{L}|\mathcal{C})$, where $\mathcal{C}$ is a code snippet, and $\mathcal{L}$ is a label/method name that should correspond to the provided code snippet. In other words, we want to maximize the probability of predicting the correct label $\mathcal{L}$ given a code snippet $\mathcal{C}$. This is accomplished by representing programs as bags of path contexts, constructing code vectors from such bags, and then comparing code vectors with learned embeddings for code labels to make a prediction on which

method name should be attributed to the code snippet.

More specifically, code2vec learns 3 sets of embeddings:

- An embedding matrix $\mathcal{P}$, where each row corresponds to a path-context from the training corpus, and the number of columns corresponds to a tuneable embedding dimension.
- An embedding matrix $\mathcal{N}$, where each row corresponds to a terminal token within the training corpus. Again, the, width is a tuneable hyperparameter of the model.
- An embedding matrix $\mathcal{L}$, where each row corresponds to a method name.

Additionally, an attention vector a is learned for computing the final weighted sum of context vectors forming the final code vector. The authors emphasize the importance of this attention mechanism, having the model learn which vectors to assign more weight to impacts overall results.

A bag of path contexts is a series of paths along a program AST that start at a terminal node, traverse the AST using some path, and then terminate at another terminal node in the AST. Thus, a context is represented as the tuple (x_s, p, x_t), where p is the path and x_s and x_t are the start and ending terminals, respectively.

The code2vec model learns to predict labels by taking each path context and concatenating the embeddings for the start terminal, path, and ending terminal, which are then passed through a single linear layer. This is done in parallel for a set of path contexts for a code-snippet. A linear combination of the learned vectors is computed with a^T serving as the coefficients. This final summed vector is the output code vector. Code2vec then takes the dot product of the computed code vector with all label vectors and normalize in order to build the probability distribution of guesses for the assigned label for the code snippet. In other words, the score for each label is computed as the dot product of the complete code vector with *each* of the label vectors.

2.4.5 GraphMAE

There have been other methods explored for representing graphs (and by extension, ASTs), including through the use of autoencoders [41]. GraphMAE is a graph autoencoder that utilizes self-supervised learning to learn latent representations for a set of graphs using graph neural networks (GNNs). Graph neural networks function by operating on feature vectors assigned to each node in a graph, and the models themselves implement a learned transformation on such feature vectors. GraphMAE utilizes an encoder and decoder graph neural networks [37], typically using either Graph Convolutional Networks [39], or Graph Attention Networks [38]. Node features are passed

through the encoder and decoder modules, with the intermediate representations between the encoder and decoder modules serving as the constructed embeddings.

The same authors published an extension paper on GraphMAE [42], where they add feature regularization through randomized feature masking to improve upon some performance issues faced in the first paper.

2.5 Existing applications of Embeddings in Program Synthesis

Given the unique algebraic properties of vector representations of code and other complex data, another interesting avenue is the use of latent program spaces as a means of performing program synthesis. Latent Program Networks [43] is an attempt at using latent programs to generate solutions for the ARC AGI Competition [44]. The ARC (which expands to Abstraction and Reasoning Corpus) AGI competition is a benchmark to test AI systems on their general reasoning ability. This is done by providing models with a few input/output pairs of a coarse grid image, and then testing the model on its ability to extract the pattern from the training data to a new grid input. An example set of training input/output pairs is in Figure 2.3.

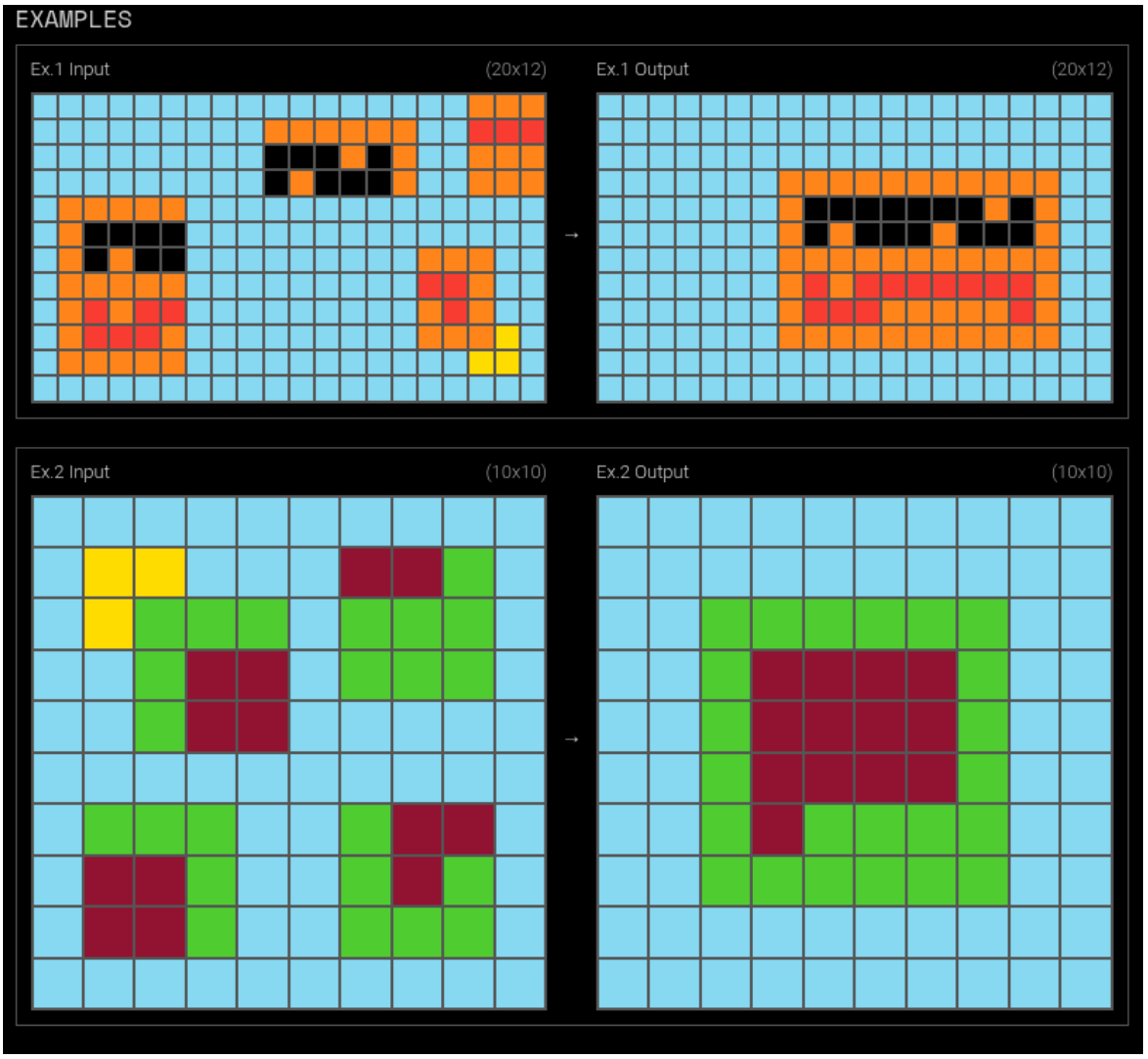


Figure 2.3: Example ARC prize example (Public Evaluation Set V2 Hard problem 32). Models are given these training examples, and an additional unique input grid is given to a model and it must produce the correct output.

The authors built a system for this by by encoding ARC I/O pairs into a latent program representation, and moving the vector through the latent space to a location that is best explained by the data. Once the latent program has been modified, it is passed to a decoder to execute the latent program on an input to generate an output. Systems such as LPNs provide an interesting take on representing programs as vectors, and using the latent space for search rather than in the discrete case.

It is important to note that embeddings are used in other program synthesis regimes, but not necessarily with the same emphasis as in LPNs. For example, DeepCoder [13] embeds inputs and outputs to pass into the DeepCoder model.

2.6 Text-based Embeddings

Along with specialized embeddings for specific problems, there has been an increase of more general-purpose text-embedding models. The most prevalent example of the application of these models is for retrieval-augmented generation (RAG) [45, 46] for various natural language processing tasks. The common pattern is to encode a series of documents using one of these models as text vectors, store such vectors in a vector database, and then encode incoming prompts or queries with the same embedding system, and doing nearest-neighbor search among the document store to retrieve relevant documents. These documents can then be fed into the prompt of some system, like an LLM, and can improve the quality of LLM outputs.

The latest models build on top of the success of attention in the LLM space, and typically perform the same pattern of encoding a sentence into tokens, performing various rounds of attention on the token embeddings, and then perform some kind of transformation on the context window to retrieve an output embedding. These models have also seen some of the recent improvements to LLMs be applied to their architectures as well. For example, Nomic AI's V2 embedding model [47] has included the same Mixture-of-Experts architecture as was pioneered in DeepSeek R1 [18].

Snowflake has developed their own suite of embedding models [48, 49], that are also transformer-based systems. The authors of these models indicate the biggest focus was on improving multilingual capabilities over previous generations. Additionally, Mixed-Bread AI has created an open-source embedding model following a similar transformer paradigm [50].

A key insight to take away from these models is that once again, all explicit hierarchical information in an input sentence is thrown out within these models, and only a sequence of tokens remain.

Chapter 3

Methods

3.1 Languages

Given the objective of this work, one key component is the choice of language spaces that will be defined for exploration. The exact grammars and the variants used in experiments for these languages are provided in the appendix.

3.1.1 TACO Expressions

The Tensor Algebra Compiler (TACO) is a specialty compiler developed to transform tensor algebra operations into C compute kernels to execute the corresponding operation on the appropriate input tensors on both CPUs and GPUs [51]. One input to the TACO compiler is a tensor algebra expression of the form:

$$A(i, j) = X(a, b) * Y(c, d),$$

where the resulting tensor A has various dimensions (i.e., i and j), and A is created based on the operations of various tensors on the right hand side. In this case, this operation is the matrix-multiplication between X and Y (both tensors of rank 2).

One advantage of choosing to use TACO as my compiler of choice is due to it's support for sparse tensor operations. Sparse operations differentiate themselves from dense operations in that one or more of the input tensors are sparse (i.e., a large proportion of the entries are zero). Because of this, there are typically many frivolous computations if one computes on sparse tensors like one would dense ones. Additionally, it doesn't make sense to store sparse tensors in a dense format, since most space would be zeroes and not contain useful information. These memory and compute constraints necessitate customizations when creating compute kernels to compute on sparse tensors. This is one of TACO's strengths, and this increased generality is what made me decide upon its corresponding expression language.

The TACO expression language was implemented within lang-explorer for experimentation in embedding schemes as will be described in later sections.

3.1.2 TACO Schedules

Due to the nature of tensor algebra operations and their general ability to be parallelized, TACO grants the user the option to provide a schedule to alter the execution order of the chosen tensor algebra expression. This schedule is optionally provided to the compiler in addition to the TACO expression describing the desired operation. The goal behind this is to create schedules that take advantage of the underlying hardware on which the program will be executed in order to achieve better performance, optimizing for things like cache locality, and improved memory access patterns.

The grammar for TACO schedule programs is given in the appendix. The schedule language itself is a comma-separated list of scheduling directives, where each directive is the name of the directive, followed by a series of parameters enclosed in parentheses. Please note that a subset of the full TACO schedule functionality was implemented.

Provided TACO expressions can be parsed by the compiler into an AST, after which each schedule directive provided in the schedule program is used to apply a deterministic transformation to the AST. A few examples of this include:

- The unroll directive being applied to a particular loop index (with n iterations) in order to unroll the entire loop and create n copies of the enclosed statements in its place, forcing the compiler to unroll the loop.
- The interchange directive being applied to two loop indices in order to switch the positions of those loops within the kernel.
- The parallelize directive being utilized along a particular loop index in order to perform all subsequent loop iterations in parallel, resulting in an empirical performance for that loop.

Figure 3.1 is an example of how TACO scheduling directives can alter the output code for a simple matrix vector tensor kernel.

$y(l) = A(i,j) * x(k)$ TACO compiles to

```
for (int32_t l = 0; l < y1_dimension; l++) {
  double yval = 0.0;
  for (int32_t i = 0; i < A1_dimension; i++) {
    for (int32_t j = 0; j < A2_dimension; j++) {
      int32_t jA = i * A2_dimension + j;
      for (int32_t k = 0; k < x1_dimension; k++) {
        yval += A_vals[jA] * x_vals[k];
      }
    }
  }
  y_vals[l] = yval;
}
```

$y(l) = A(i,j) * x(k)$
$+ \ split(i, i0, i1, 5)$ TACO compiles to

```
for (int32_t l = 0; l < y1_dimension; l++) {
    double yval = 0.0;
    for (int32_t i0 = 0; i0 < ((A1_dimension + 4) / 5); i0++) {
      for (int32_t i1 = 0; i1 < 5; i1++) {
        int32_t i = i0 * 5 + i1;
        if (i >= A1_dimension)
          continue;

        for (int32_t j = 0; j < A2_dimension; j++) {
          int32_t jA = i * A2_dimension + j;
          for (int32_t k = 0; k < x1_dimension; k++) {
            yval += A_vals[jA] * x_vals[k];
          }
        }
      }
    }
    y_vals[l] = yval;
  }
```

$y(l) = A(i,j) * x(k)$
$+ \ split(i, i0, i1, 5) \ + \ unroll(i1, 5)$ TACO compiles to

```
for (int32_t l = 0; l < y1_dimension; l++) {
  double yval = 0.0;
  for (int32_t i0 = 0; i0 < ((A1_dimension + 4) / 5); i0++) {
      #pragma unroll 5
      for (int32_t i1 = 0; i1 < 5; i1++) {
        int32_t i = i0 * 5 + i1;
        for (int32_t j = 0; j < A2_dimension; j++) {
          int32_t jA = i * A2_dimension + j;
          for (int32_t k = 0; k < x1_dimension; k++) {
            yval += A_vals[jA] * x_vals[k];
          }
        }
      }
    }
    y_vals[l] = yval;
  }
```

Figure 3.1: Short example of how TACO Schedules can be used to change the structure of a tensor algebra kernel. The first program is plain TACO without a schedule; the second program has the split operator applied to split the i loop, and the third program has split and unroll applied to the inner-most loop.

The first program is generated for basic matrix-vector multiplication with no scheduling directive applied. In the second program, the split directive is added so that the i loop gets split into two nested loops. The inner loop has a fixed size based on the split factor, in this case 5. In the third program, the same split rule is applied, but an unroll operation is applied to the $i1$ loop. The code presented here is a bit of an oversimplification of what would actually be output by the TACO compiler (this would include some extra code to handle the case that the original i dimension is not divisible by 5), but one can see the resulting program will have the $i1$ loop unrolled 5 times.

The TACO scheduling language was also implemented within lang-explorer.

3.1.3 CSS

Cascading Style Sheets (CSS) is a language used for formatting and presentation of HTML or XML documents. CSS is most prominently used by websites in order to format and add styling to HTML web pages. Although there are more features to the language that were omitted, the most commonly used subset of the language consists of a series of blocks, each prefixed with a selector, and succeeded by a list of properties and corresponding arguments. The selector is an indicator of which document elements the provided properties will be applied to. The properties describe the various transformations to apply to those elements, including

- Setting the font size/color/weight for text within an element.
- Setting the color/background for an element.
- Setting border radius/thickness for an element.
- Adjusting the size/positioning of an element, absolutely or relative to other elements.

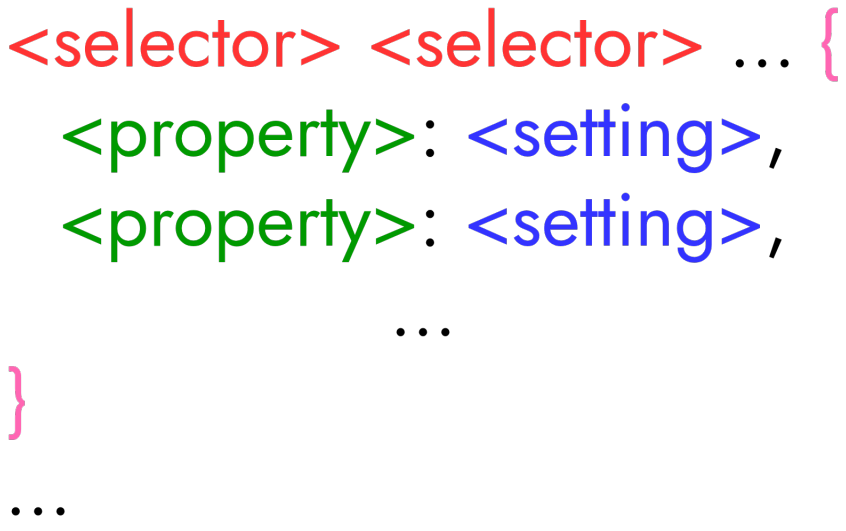


Figure 3.2: Simple diagram describing the structure of CSS programs. There can be one or more *selectors* that will apply the contained properties in the block to the selected objects within a webpage. You can have many of these blocks in a single file.

3.1.4 Karel

Karel [52] is a language used as problem domain in previous works around RL-based program synthesis [14]. The language gives instructions to a robot in an imaginary gridworld, whose main task is to move around the world, pick up, and place flags at

different places in the gridworld. The language allows one to create a sequence of statements, primarily allowing the agent to move, turn, and pick up/place flags within the gridworld. The language also has conditionals and loop functionality to allow for more complex operations.

3.2 Language Embeddings

A key prerequisite for program synthesis that is a focus in this thesis is program representation. The aim is if one is able to represent intermediate and completed programs in a latent space well, then those vectors can be used downstream as a source of context for a program synthesis model, whatever form it may take. The following is a review of the methods that were implemented in order to create latent representations of the various formal languages described above.

3.2.1 Graph labeling

As noted in the literature review, the canonical implementation of graph2vec [36] utilizes the Weisfeiler-Lehman graph kernel [5] to extract all subgraphs from a given input graph. The traditional implementation of concatenating node labels is to take the string-serialized representation of a node's label and the label of all neighbors, sort them, and then create a new string as a concatenation of all labels. Concatenated labels causes a potential issue regarding label size, as the size of each label will drastically increase as the number of iterations increases. This results in a potentially exponential increase in memory usage for capturing of all subgraphs of a given degree. The implementation of the labeling algorithm is detailed in Algorithm 1.

Algorithm 1: WL-Kernel implementation with fixed-size labels

Input: Vector of nodes N, number of iterations n

Output: Set of found features V, Nodes with adjusted features N

```
1  V ← { };
2  foreach i in 0..n do
3      new_features ← { };
4      foreach node in N do
5          new_feature ← hash(concat(node.feature, node.neighbors.features));
6          new_features[node] = new_feature;
7          V.insert(new_feature);
8      end
9      foreach node in N do
10         feature ← new_features[node];
11         node.features ← feature;
12     end
13 end
14 return V, N
```

The concatenation function must be a deterministic function mapping a set of labels to a new label. One possible technique, which is the one implemented in graph2vec and is done in the lang-explorer implementation, is using some sort of deterministic (and preferably fast) hash function which takes the current ordered labels as input and outputs a new label of fixed size. That allows all labels to remain a fixed size and stabilize memory consumption. Specifically, I utilize the CityHash64 algorithm as implemented within the fasthash crate in Rust [53]. This algorithm takes arbitrary input bytes (which all existing labels are cast into) and outputs a single unsigned 64-bit integer. The output space is much smaller than the original graph2vec implementation, as they use md5 hashes, which are comprised of 128 bits. Figure 3.3 gives a visual demonstration of the algorithm.

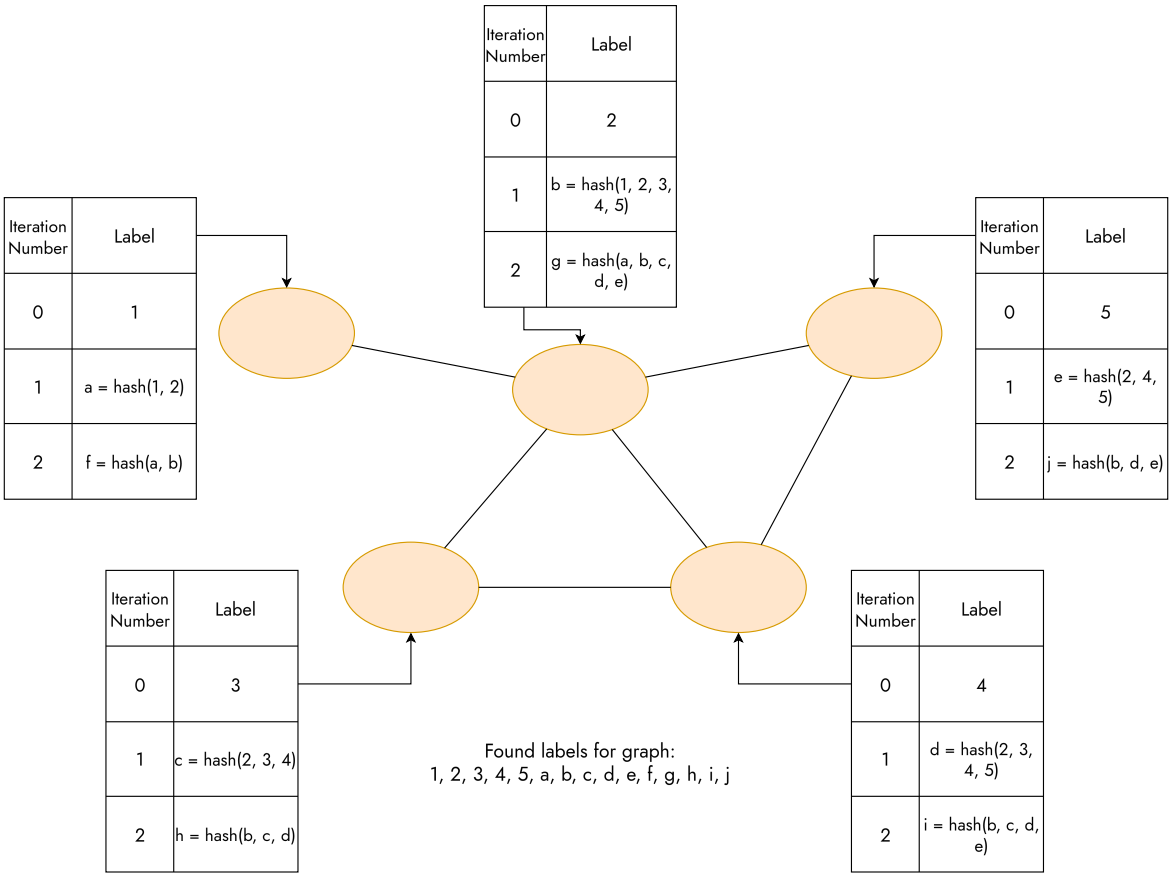


Figure 3.3: Example iteration of the WL-Kernel on an undirected graph (all ASTs will be treated as undirected to propagate information upwards as well as down the graph). In this example, each node starts with a number as their initial label. Each iteration hashes the ordered set of a node i's label and the labels of all i's neighbors. All label sets from each iteration are concatenated together to return the completed label set for a particular AST.

This technique is the key feature extraction technique that is utilized within my experiments. As noted in the literature review, this is the technique that is used within graph2vec for extracting graph features as words.

3.2.2 Out-of-vocabulary handling

One issue with many of the embedding techniques discussed in the literature review is that the embeddings matrix is of fixed size, where each of the corpus words/documents is mapped to a vector, and the behavior of how to embed any new vectors outside of the training corpus is not well-defined. There are multiple ways in which this could be handled. One technique involves creating an additional catch-all vector that serves as the embedding for any new words (optionally in a document) that were not included in the training corpus. In doc2vec and by extension graph2vec, we have vectors for all documents, and the number of rows in the hidden layer of the model directly corresponds to the number of words in the training corpus. Fortunately, each new

document vector can be created outside of the original model and trained by being plugged into the hidden layers. With regards to words, an additional catch-all word that accounts for any words not originally in the training corpus was utilized.

3.2.3 Doc2Vec Re-creation

As part of the work for this thesis, I undertook the task of replicating the part of the results of the original graph2vec paper by implementing doc2vec [7] from scratch using the Burn machine-learning framework [54] in Rust [6]. This was done in part for pedagogical reasons, but also in part to keep all tooling in the same language as the rest of the lang-explorer codebase. It was also done to have a hackable implementation since the standard implementation [55] is over a decade old, and has an API that is not conducive to extensive modifications without thorough knowledge of the codebase, which I lack.

The overall functionality of the base model is the same as in the original paper [7], but there are a few key differences with implementation. For one, the only supported loss function is negative sampling, as that is directly baked into the forward pass of the model for reducing excessive computation. Additional tuning was done with various learning rates and epoch counts that vary from the canonical implementation to get better results. One major issue is both PyTorch and Burn do not have support for partial differentiation of model tensors; it is all or nothing. As a result of this, my implementation does backpropagation on all parameters as well as runs the optimizer on all parameters even if the gradient is zero. This leads to substantial slowdowns that need to be remediated in future work with more granular tensors.

3.3 Program Expansion Techniques

The following two subsections provide a formal look at the mechanics of how I chose to approach the program expansion problem. Given the fact that multiple tiers of formal languages operate under various constraints, so to do the various expansion architectures vary depending on the class of language one wishes to expand. The two classes which machinery is is explicitly designed for are context-free and context-sensitive languages. Obviously, regular languages can be expanded using context-free machinery, and both regular and context-free languages can be expanded with context-sensitive machinery. But, the increasing complexity and compute requirements along with reduced guarantees of correctness for context-sensitive machinery makes it preferable to use the least complicated tooling for the job.

3.3.1 Expanding Context-Free Languages

Lang-explorer supports expanding context-free grammars. As a short review, a context-free language is a language described by a grammar with production rules of the form $A \rightarrow \gamma$ where $A \in \mathcal{V}$ and $\gamma \in \{\mathcal{V} \cup \mathcal{T} \cup \epsilon\}^+$. As a result, from the start symbol, the task of an expander is to recursively find all non-terminal symbols, find the corresponding production rule for the non-terminal, choose a production, and create children of the non-terminal that are the production symbols. The algorithm is executed in a depth-first fashion such that the resulting expansion tree has only terminal symbols in the leaves. This is how expansion is implemented within lang-explorer, and the algorithm is more formally expressed in Algorithm 2.

Algorithm 2: GenerateProgramCtxFree

Input: Grammar G, Expander E

Output: ProgramInstance I

```
1 prod ← G.getProduction(G.getRoot());
2 return GenerateProgramRecursiveCtxFree(G, prod, E);
```

Algorithm 3: GenerateProgramRecursiveCtxFree

Input: Grammar G, Production P, Expander E

Output: Program instance I

```
1  nt ← P.nonTerminal;
2  rule ← E.expand(G, production);
3  children ← [ ];
4  foreach symbol in rule do
5      if symbol is non-terminal then
6          prod ← G.getProductionWithLHS(nt);
7          child ← GenerateProgramRecursiveCtxFree (G prod, E);
8          append child to children;
9      end
10     else if symbol is ϵ or terminal then
11         append new ProgramInstance(item) to children;
12     end
13 end
14 I.children ← children;
15 return I;
```

3.3.2 Expanding Context-Sensitive Languages

Additionally, lang-explorer supports expanding context sensitive grammars in order to improve the utility of the framework. This resulted in a different algorithm and problem formulation than the simpler context-free expansion framework. Please recall

the main distinction between productions of a CFG and a CSG is that productions of the CSG are of the form $\alpha A\beta \rightarrow \alpha\gamma\beta$, where $\alpha, \beta \in \{\mathcal{V} \cup \mathcal{T} \cup \epsilon\}^*$, $\gamma \in \{\mathcal{V} \cup \mathcal{T} \cup \epsilon\}^+$ and $A \in \mathcal{V}$, whereas $\alpha, \beta = \emptyset$ in a CFG. This additional context around each left-hand side non-terminal forces one to throw out the simple depth-first expansion paradigm and start from scratch. Since one needs to account for context when deciding what production rule to use, it is useful to store the "frontier" of the expansion tree (i.e., the leaves of the expansion tree visited in-order) in a list.

With this list, one can find all places where there exist subsequences that are equivalent to the left-hand side of all production rules in a grammar. It is then the job of the grammar expander to choose one of the provided productions and one of the possible indices in the frontier where the grammar will be expanded. This problem of choosing which element in this matrix to expand upon is referred to as the Frontier Expansion problem throughout the rest of this document.

Once the production has been chosen along with the index in the frontier, the grammar expander then additionally chooses which production rule to use, and the non-terminal center element in the frontier at index i is removed, the symbols of the production rule are inserted back into the frontier in order starting at index i, and the children are added as children to the parent non-terminal. This process is repeated until the frontier is devoid of non-terminals, and the final frontier list of symbols is the generated program. The algorithm for this procedure is defined more formally in Algorithm 4, and a visualization of the expansion frontier is shown in Figure 3.4.

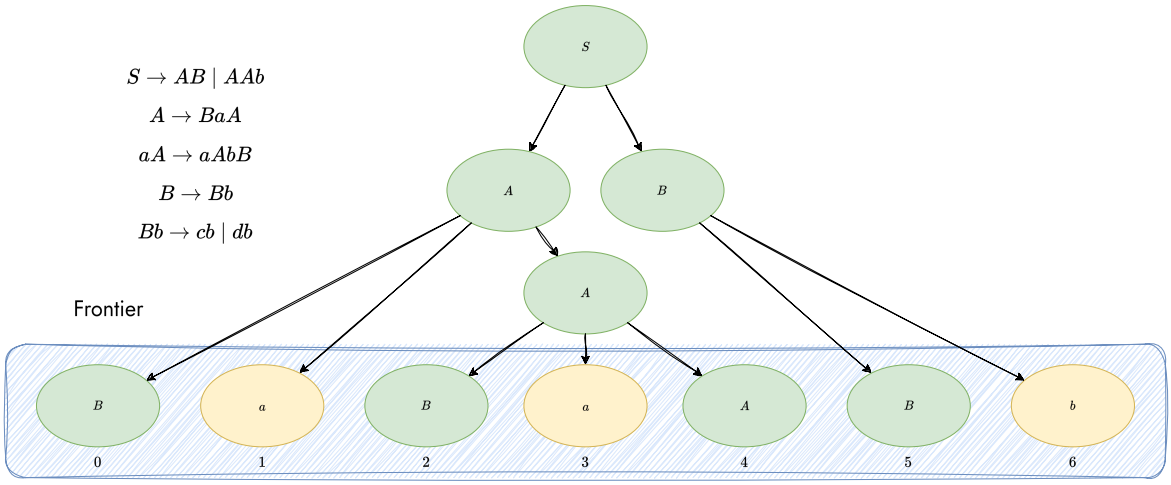
As an additional note, it becomes clear that there certainly could exist frontiers in which there are no productions that can be matched within the frontier. The default behavior is to fail the expansion, although future work would involve backtracking and using additional heuristics to avoid taking paths that lead to a similar frontier state wherein no expansions can be applied.

Algorithm 4: GenerateProgramCtxSensitive

Input: Grammar G, Expander E

Output: ProgramInstance I

```
1  I ← new ProgramInstance(G.getRoot());
2  frontier ← [I];
3  while frontier has non-terminals do
4      globalLHSSlots ← [ ];
5      foreach lhs in G.getAllProductions() do
6          localLHSSlots ← [ ];
7          foreach item, idx in frontier do
8              found ← True;
9              foreach symbol, offsetIdx in lhs.getLHSSymbols() do
10                 if symbol ≠ frontier[idx + offsetIdx] then
11                     found ← False;
12                     break;
13                 end
14             end
15             if found = True then
16                 localLHSSlots.append(idx + lhs.prefix.len());
17             end
18         end
19         if localLHSSlots.len() > 0 then
20             globalLHSSlots.append((lhs, localLHSSlots));
21         end
22     end
23     lhs, idx ← E.chooseLhsAndSlot(G, globalLHSSlots)
24     prod ← G.getProductionWithLHS(lhs);
25     rule ← E.expand(G, prod);
26     removedNt ← frontier.remove(idx);
27     foreach symbol in rule do
28         p ← new ProgramInstance(symbol);
29         frontier[idx] ← p;
30         idx ← idx + 1;
31         removedNt.children.append(p);
32     end
33 end
34 return I;
```



Frontier Decision Table

	0	1	2	3	4	5	6
$S \to AB \mid AAb$	0	0	0	0	0	0	0
$A \to BaA$	0	0	0	0	1	0	0
$aA \to aAbB$	0	0	0	0	1	0	0
$B \to Bb$	1	0	1	0	0	1	0
$Bb \to cb \mid db$	0	0	0	0	0	1	0

Figure 3.4: Visualization of the context-sensitive frontier decision problem. Before we can choose the rule to expand, we first have to pick a place along the frontier for a rewrite to take place. The matrix beneath the frontier graph, which represents every frontier index as a column and every production as a row. If a rewrite of the given production can occur at index i, the matrix gets a value of 1, otherwise 0.

3.4 Program Synthesis Decision Function Approximation

With the availability of rich embeddings for programs, I choose to implement a series of models to expand formal grammars for generating syntactically valid programs, and working towards learning good semantics. The idea behind how this is accomplished is to encode priors about the structure of code such that less computational effort must be spent learning the syntax, and more focus can be given towards the arguably more difficult task of learning better semantics and heuristics. The following provides an outline of the various expander systems implemented to create programs from the provided languages in section 3.1, with varying levels of utility.

3.4.1 Monte-Carlo Expander

The most basic expander implemented is a Monte-Carlo expander. This expander uniformly samples from a given production's possible expansion rules. That is, no naive

or context-aware prioritization of the possible expansion rules is done. This expander is designed to be a baseline expansion mechanism with nothing but randomness informing it's decisions.

This expander can be generalized to the frontier decision problem for context-sensitive grammars. In this case, the expander simply chooses a random production whenever there are more than 0 possible expansion paths.

3.4.2 Weighted-Monte-Carlo Expander

The Weighted-Monte Carlo expander is a small extension of the base Monte-Carlo expander. The difference is that for any given production and set of possible expansion rules, one can provide a "weight" alongside any given production rule. If all rules have a weight assigned, then the softmax of the weights is taken and the next expansion is sampled from the resulting probability distribution. Softmax is applied to the full set of weights, and again the expander samples from the provided distribution. If no expansion rules within a production have a weight assigned, then the expansion rules are sampled uniformly like in section 3.4.1. This is intended to be an enhanced baseline on top of base Monte-Carlo, since coercing the expander to frequent certain expansion rules more often (i.e., avoiding the ϵ expansion within the start production 99 percent of the time) helps lead to richer program generation.

Similar to the regular Monte-Carlo expander, for the frontier decision problem the weighted Monte-Carlo expander defaults to choosing a random production that has more than 0 possible positions available in the frontier, and then chooses a random position for the production within the list.

3.4.3 Learned Expander

The next expander implemented is the learned expander. With the Learned expander, a model (of which various are implemented, with more information to follow), is utilized to provide the probability distribution to be sampled for the next expansion rule is learned and output by the model, with the input to the model being the embedding vector of the current partially-expanded program. The intuition behind this architecture is to encode priors about the current program context in such a way that it can be accounted for by a model in providing a probability distribution for the next expansion rule.

Variable Linear Expander

The Naive linear expander is the most simple of the proposed architectures for performing production-rule prediction. The model consists of an MLP with parameters

for number of layers, activations, and bias. Input and output sizes are the embedding dimension and the number of production rules, respectively. The idea of this model would be to take the current partial program embedding as input to the model of the particular production we are looking to expand, and outputting logits to be normalized to a probability distribution which can be sampled to decide the production rule to use for expansion. As a consequence, this results in a unique linear model per production rule. The task of constructing these models can be automated, but decreases the generalization of the system as a whole, especially if one creates a custom variant of a grammar at runtime (e.g. creating new indices with the Taco Expression grammar). This model structure is demonstrated on the left-hand side of Figure 3.5.

Fixed Expander

The Fixed Expander is another proposed model architecture that looks to solve the model-per-production problem. The idea is to have a single MLP that similarly takes as input the embedding of the current partially constructed program, but produces another fixed-size vector. Each production rule has an associated learned vector, which can be compared with the output from the model, and the production rule (from the set of rules for the production we are looking to expand) with highest $\mathcal{L}_2$ similarity to the vector output from the model can be used for expansion. This model allows a single model for all expansion rules, which can increase the generalization of the technique. The fixed expander is described on the right-hand side of Figure 3.5.

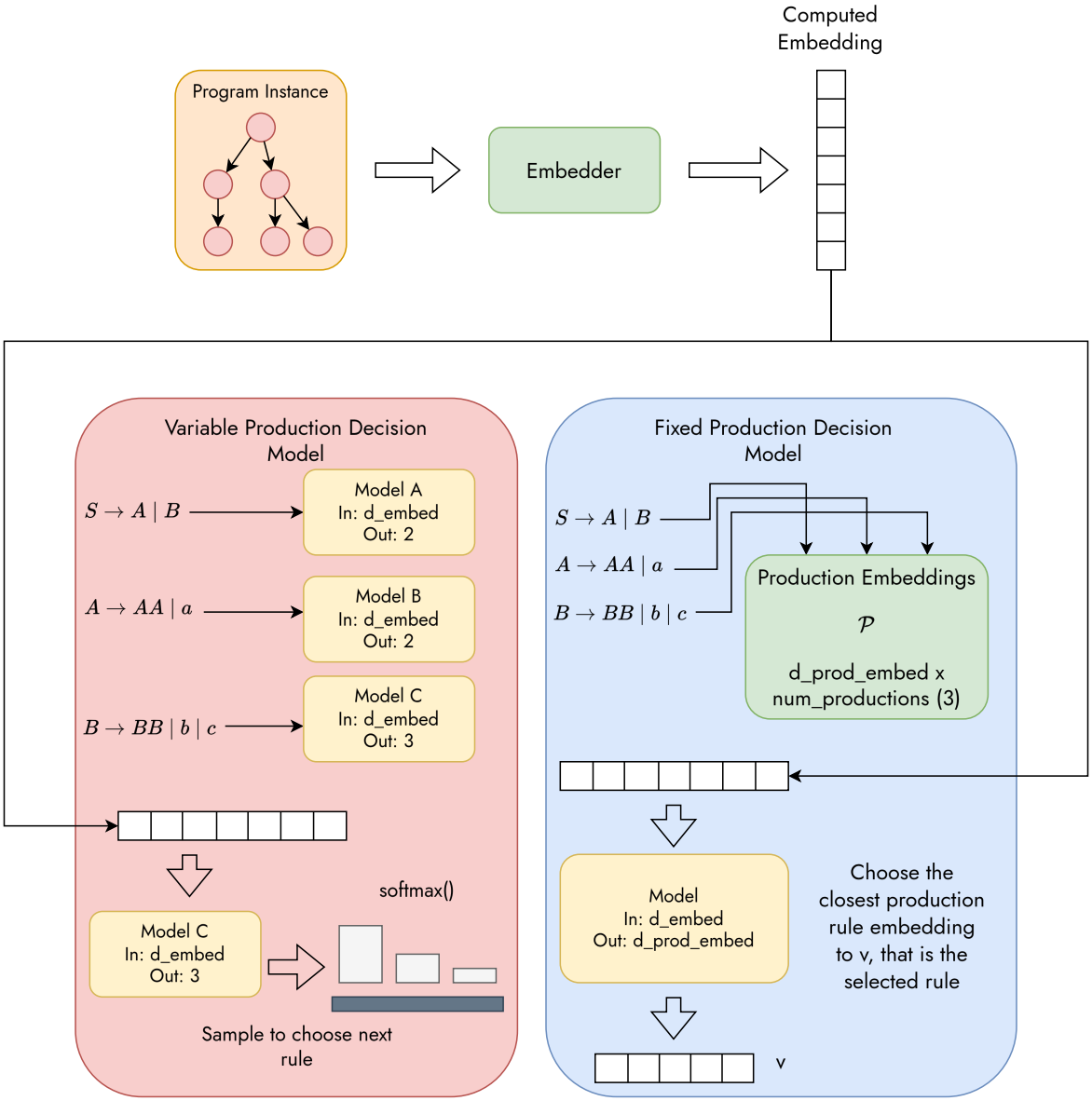


Figure 3.5: Overview of the structure of the learned expander. One begins with a partial program instance that is encoded into a vector representation using some embedding technique. Then, one of the two current models can used for making the expansion decision. The left side describes the Variable Linear expander model. Each production has some model which takes the embedding as input and will output k logits, where k is the number of production rules associated with the production. One can then perform softmax over the logits and sample from the distribution to choose the expansion rule. In the fixed linear expander on the right, each production rule is assigned a learnable embedding. The model projects the input embedding into the production rule embedding space, and the nearest neighbor among all production rules for the current production is chosen as the expansion.

Novel Learned Expanders

The LearnedExpander interface within lang-explorer is designed to be a flexible interface, allowing for new, novel models for learning the frontier and production decision

functions. Future work will involve the implementation of more models that fit this specification to augment the current capabilities of lang-explorer.

3.5 System Architecture

Now that I have elaborated on the individual components of the lang-explorer framework, I want to show how they all piece together. Figure 3.6 is a system diagram of the full lang-explorer framework. The lang-explorer framework is written in Rust [6]. This was done because it is a modern compiled programming language with strong compile-time checked typing, and a borrow checker system to ensure that no race conditions on data can take place within safe Rust. These properties make the language less desirable perhaps for writing ad-hoc scripts or glue code, but it can be very useful when writing code you wish to be deployed to a production environment in the future. It was also done out of personal preference for writing code in a language with strongly-typed variables. The lang-explorer source code and experimental results are all available publicly on Github [56].

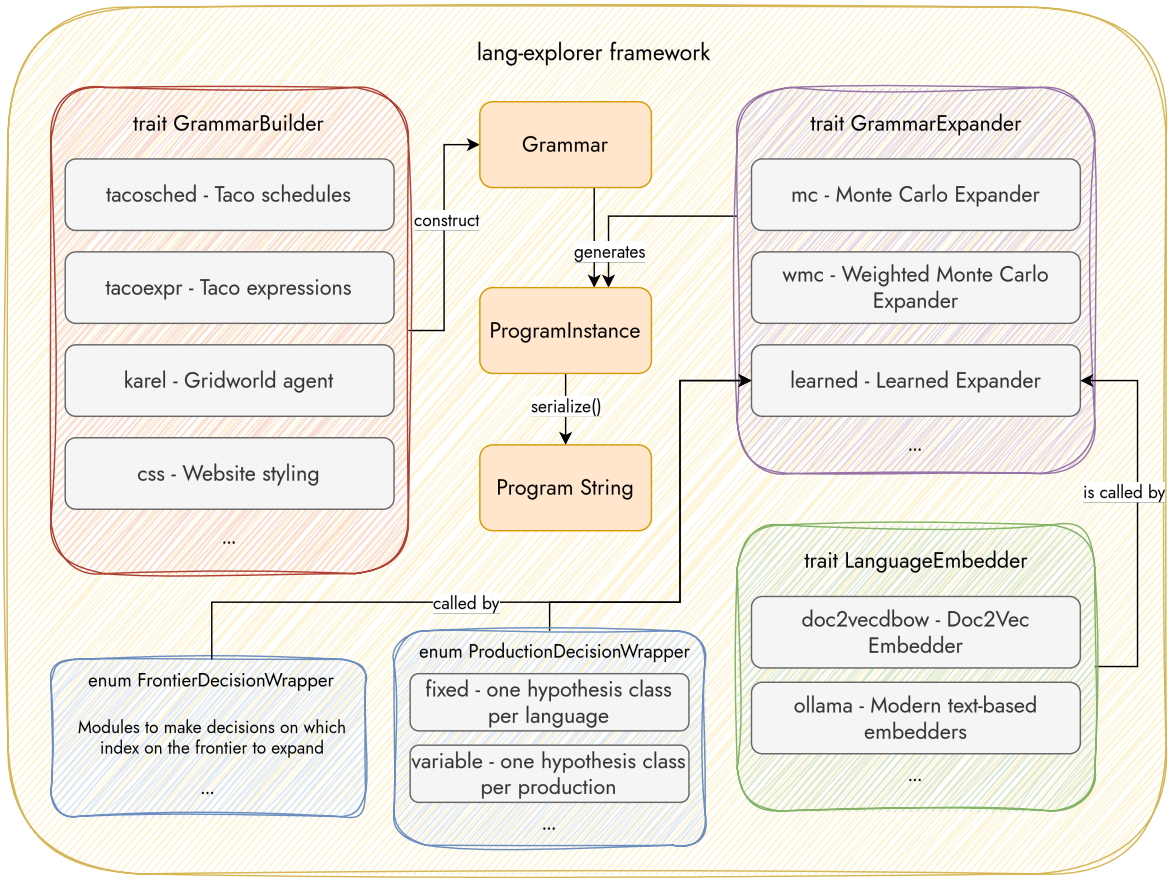


Figure 3.6: The architecture diagram for lang-explorer. Objects that implement the GrammarBuilder trait are able to construct grammars which will be expanded into ProgramInstances via GrammarExpander modules. The finished program instance can then be serialized or used in downstream processing.

The lang-explorer system is comprised of several modules that work together to generate programs. These modules implement certain traits that allow for polymorphic behavior of certain types of objects. Within Rust, there exists the concept of traits. For the purposes of this thesis, one can think of traits as being equivalent to interfaces in other languages. One trait instantiated and utilized by lang-explorer is the GrammarBuilder trait, which is implemented by various objects that can construct Grammar objects for a particular language. This includes the builders for building the different languages described above in section 3.1. A grammar object describes all the productions of the language, along with the root symbol. Grammars can then be expanded by expander objects that implement the GrammarExpander trait. As described in Figure 3.6, the LearnedExpander uses more modules to make its expansion decisions. This includes an embedding system which implements the LanguageEmbedder trait, a Burn module for the production decision problem, and the frontier decision problem in the context sensitive case. Once a grammar has been expanded, a ProgramInstance object is emitted, which is the program still in general AST form. The emitted program can be serialized to final program form.

3.6 Additional Infrastructure

In addition to the construction of the lang-explorer framework, additional infrastructure was utilized to make some of the experiments of this thesis possible. Primarily, I created a Kubernetes [57] cluster out of several consumer desktops to host various pieces of software for my research as well as for those in my lab group. This includes ollama [58] and gensim's doc2vec [55] implementation using a self-built API wrapper around the Python library.

Ollama is a piece of software that allows one to run various machine learning models locally. This includes embeddings models, which are used as a comparison method in my experiments. Running many instances of ollama on multiple nodes allows lang-explorer to make many concurrent requests and create many embeddings in parallel, increasing the speed in which experiments can be completed. Additionally, the custom wrapper around gensim's doc2vec implementation allows compute to be offloaded from my main workstation to speed up experiments, and allow others to perform experiments with a standard API. Finally, the cluster runs load balancing infrastructure with Nginx [59] to serve a TLS-terminated URL for other internal users of the services.

Figure 3.7 provides an overview of the cluster infrastructure as a whole and how traffic is load-balanced among different pods on different nodes.

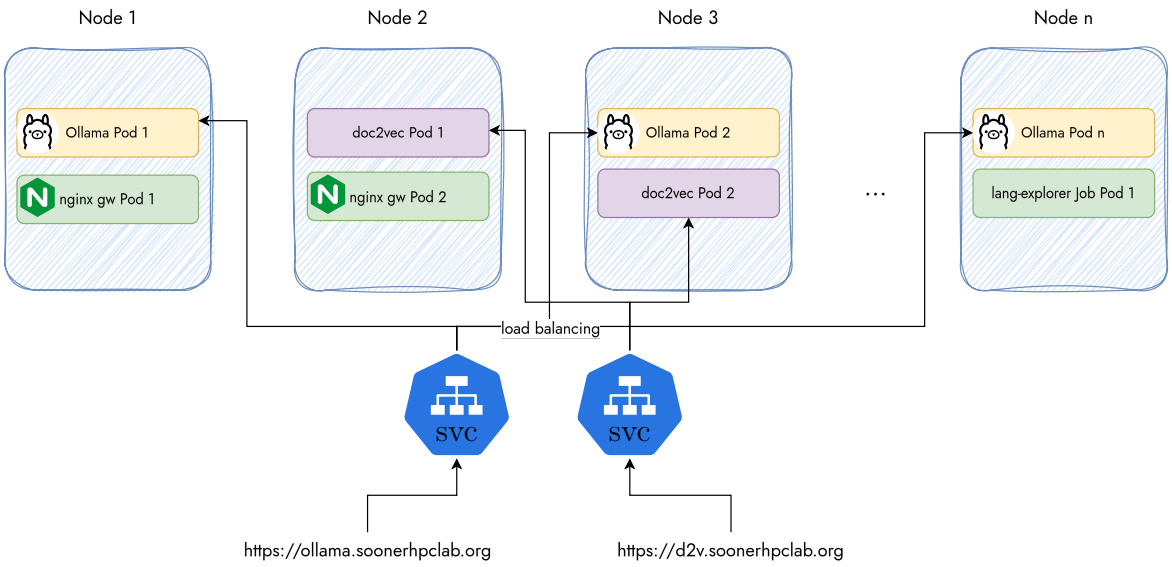


Figure 3.7: A high-level overview of the Kubernetes cluster running within the lab environment. Each node can have pods assigned to it. Each pod is in essence one or more containerized processes. In this case, we have multiple ollama instances running on multiple nodes, and multiple Python APIs wrapping gensim's doc2vec implementation [55]. Each set of pods has a service on top of it (which is a means to load-balance transparently between all pod instances), and an additional load balancer in front of it to serve the API URLs for the final services and terminate TLS. Lang-explorer jobs can be run on the cluster, or can be run in the lab network from other machines.

Chapter 4

Experiments

A similar compute pipeline was used to generate programs and embeddings for all experiments conducted. The goal of the pipeline is to have a series of embeddings corresponding to synthesized programs for analysis. This is demonstrated in Figure 4.1.

There were multiple ways that embeddings were constructed. One could use either the custom implementation of doc2vec within lang-explorer, or call the doc2vec-gensim Python wrapper API sitting on the cluster. The latter was chosen for all experiments conducted due to its superior speed. For additional text-based embeddings, programs can be sent to the Ollama API sitting on the cluster.

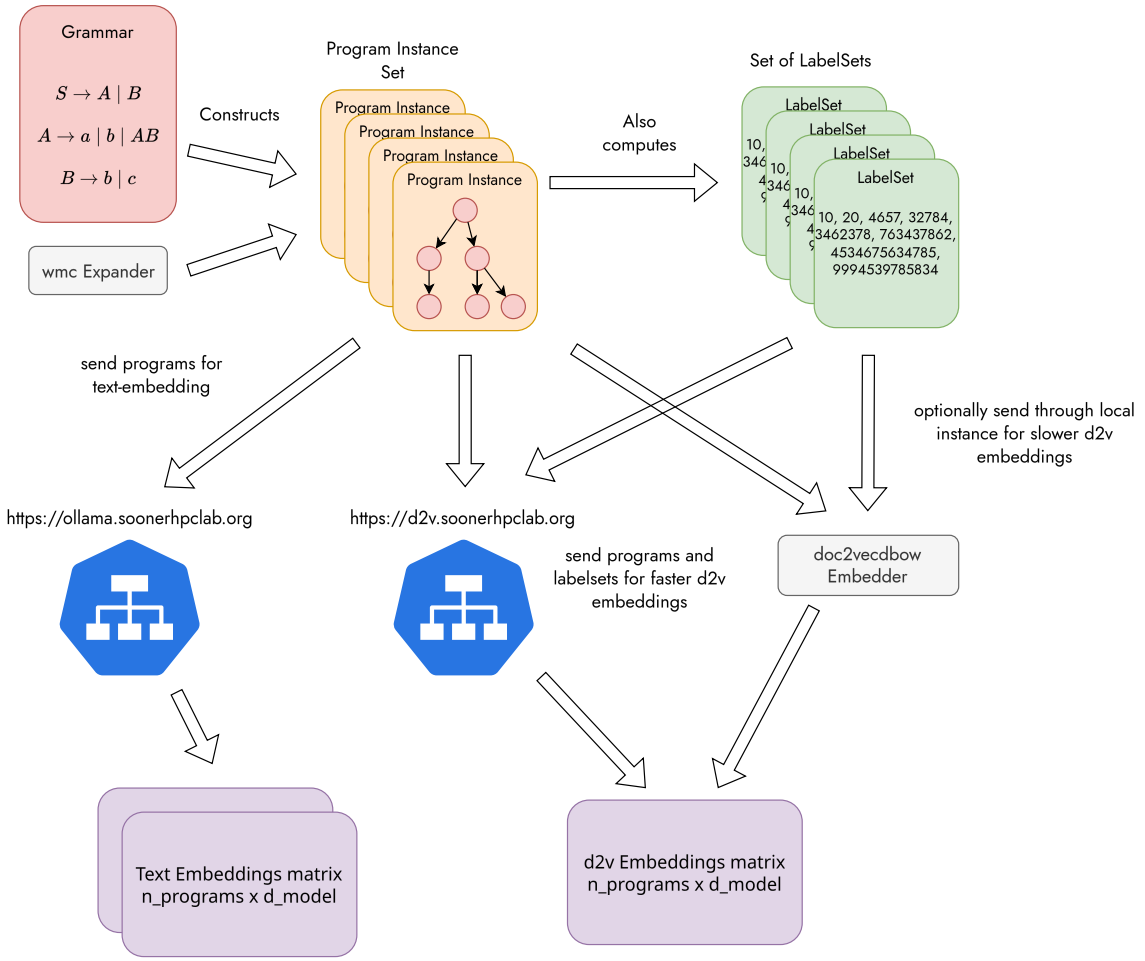


Figure 4.1: Visual for how experiments were conducted. For each language, a grammar was constructed. This grammar calls the Weighted Monte-Carlo expander to generate a set of ProgramInstances (generic AST representations). Optionally, labelsets can be extracted using the WL-kernel for each program. These programs are then passed over the network to Ollama within the cluster, which return embeddings for each program with the corresponding text embedding model. Returned embeddings are represented above as a matrix. Additionally, the programs are sent to doc2vec for embedding, either locally or to a Python API wrapping the gensim implementation.

The following sections enumerate the experiments performed on various constructed datasets from the languages enumerated in section 3.1.

4.1 Qualitative Analysis

As noted in chapter 3, lang-explorer has the capability to generate set of sampled programs from a grammar definition, and subsequently create embeddings using various backends for each program. The objective of this section is to provide some qualitative and quantitative analysis of the structure of these embedding spaces, and ultimately to see how well the semantics of programs are retained within these spaces.

In order to generate the datasets that were embedded and used for downstream

analysis, the Weighted Monte-Carlo expander was used for sampling programs from the program spaces defined by the grammars of languages in section 3.1. The weights assigned to particular rules were chosen in order to avoid the system creating many very small programs (which result in the program stalling since generated programs are de-duplicated) or very long programs that on some occasions never terminate. These weights were applied primarily to the recursive rules of each grammar.

With the ability to generate a set of random programs, it becomes important to include partially completed programs in the dataset, since in the context of program synthesis we will need to create embeddings of the current partially completed program for use in our frontier expansion decision model. Partial programs are a subset of the set of all sub-trees for any given program AST. Partially completed programs are bootstrapped from the set of completed programs. Since the set of all sub-trees would be very large for any graph of reasonable size, a smaller subset of all sub-trees were extracted as the partial-program subset.

Lang-explorer's internal representation of an AST is a ProgramInstance, which is a recursively defined type. Each node in a completed ProgramInstance (with its own children still attached) was extracted as a subgraph to be added to the dataset. As a result, the process of extracting all subgraphs is a matter of running breadth-first search over the original ProgramInstance and returning the list of nodes in the graph.

4.1.1 Embedding t-SNE Plots

The first experiment run was to generate a random dataset in each language, embed it using doc2vec, and then analyze the structure of the embeddings using t-SNE [60]. The aim behind this is to determine the structure of the space and see how programs cluster together. Each dot is colored based on the length of the program; shorter programs have darker colors, while longer programs have lighter colors. The following plots show the results for each language, and some more holistic discussion follows.

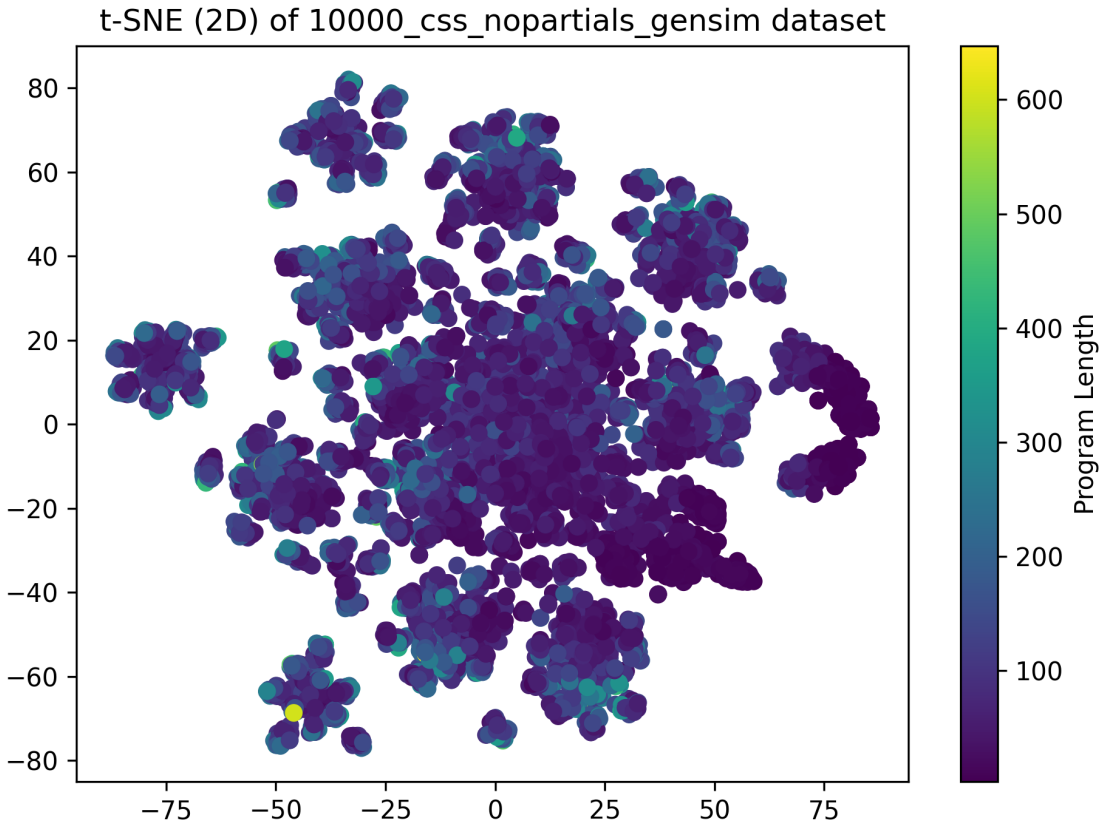


Figure 4.2: t-SNE embeddings of 10000 randomly sampled complete CSS programs, with 128 dimensions. There is clear clustering among shorter programs towards the right side of the plot, with several other clusters throughout the space.

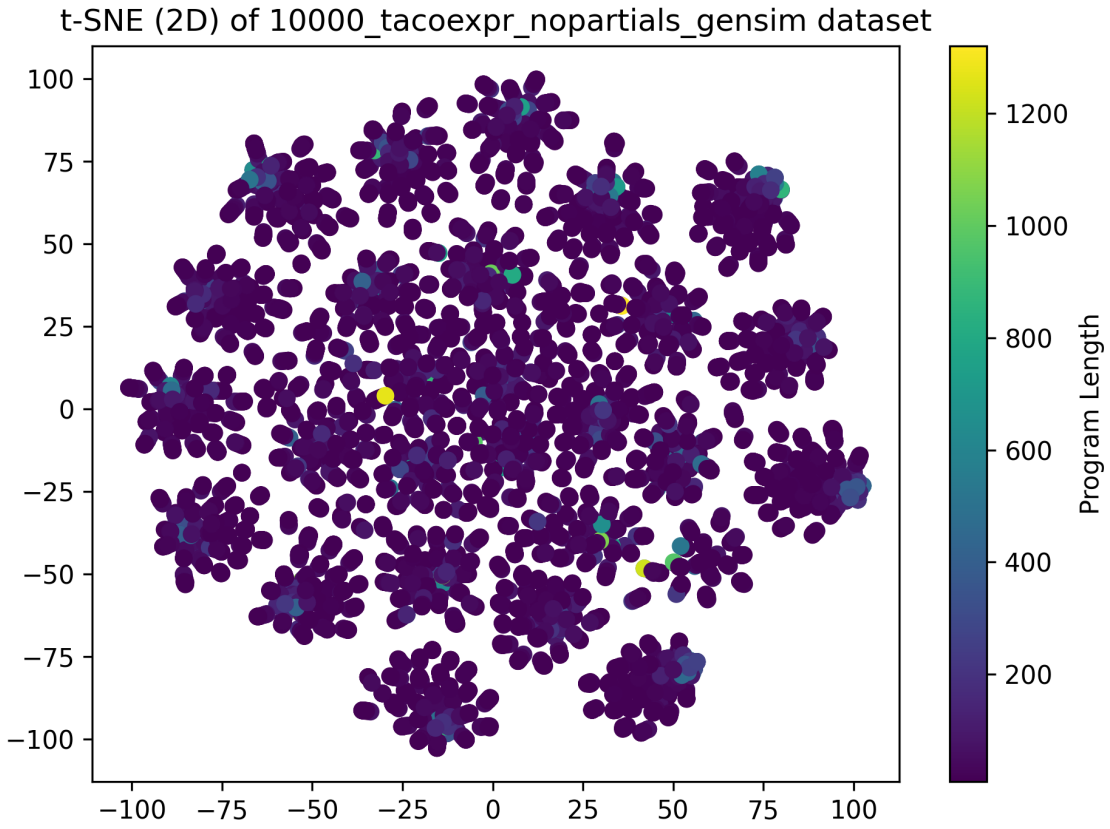


Figure 4.3: t-SNE dimensionality reduction plot of the embeddings of 10000 randomly sampled complete TACO Expressions, with 128 dimensions. Empirical analysis of this diagram indicates that the clusters are formed based on the last tensor on the right hand side of the expression.

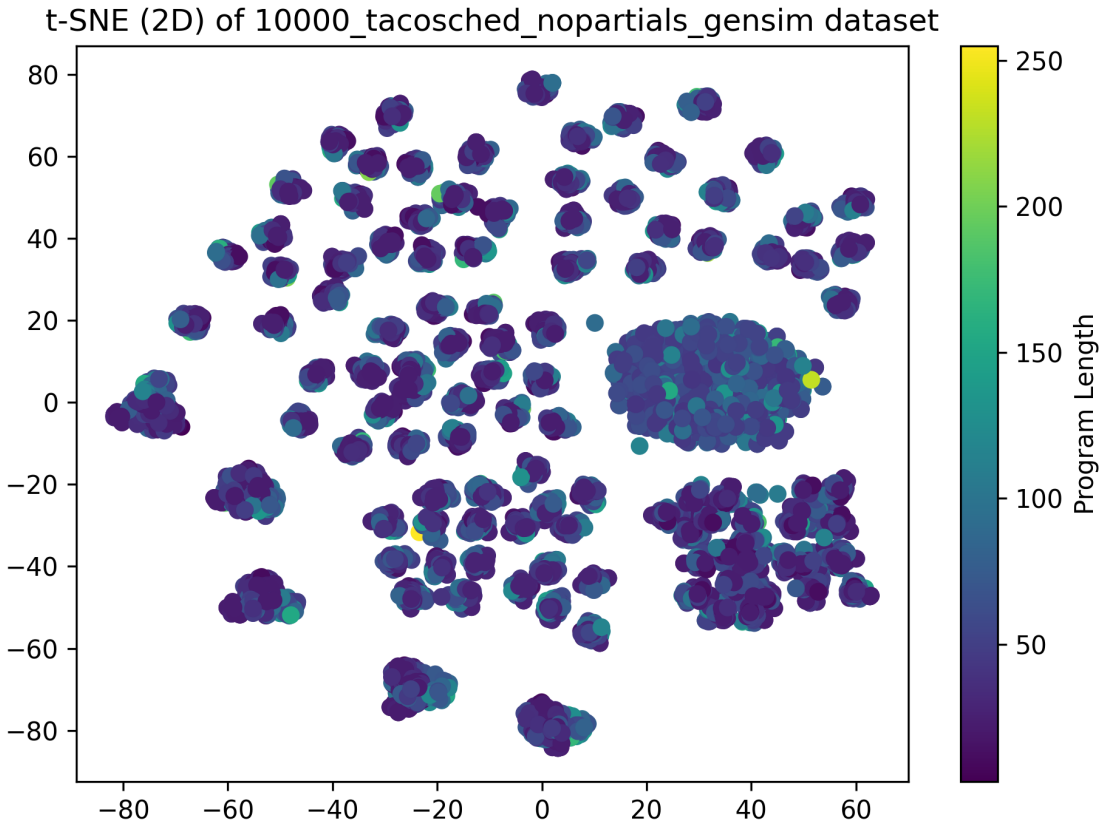


Figure 4.4: t-SNE dimensionality reduction plot of the embeddings of 10000 randomly sampled complete TACO Schedules, with 128 dimensions. Empirical analysis, similar to the above plot of TACO expressions, indicates that each cluster is delimited by the last scheduling directive found in each program.

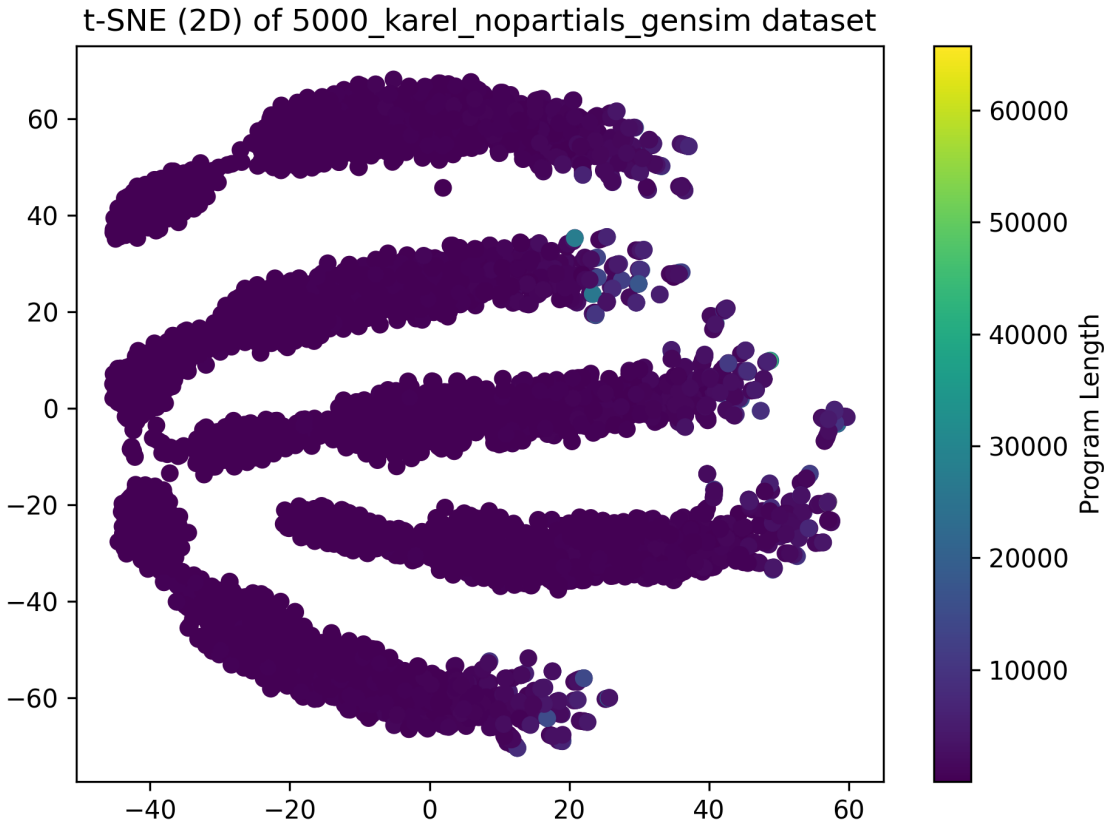


Figure 4.5: t-SNE dimensionality reduction plot of the embeddings of 5000 randomly sampled Karel programs, with 128 dimensions. Based on my analysis of the programs this set, it is not clear what the discriminating feature is between these 5 different "claw marks" seen on the plot.

With the exception of the Karel dataset, there is a common pattern of clusters of programs that form within the plots. Based on analysis of the individual programs that correspond to those clusters, the most common feature that would affect the clustering of programs within these plots was the contents of the bottom-right hand size of the expansion tree. Since all constructed grammars were recursively right-expanding, this portion of the graph typically corresponds to the very end of the program. In the case of TACO schedules, this means embedding closeness will typically be delineated by the similarity of the last scheduling directive first. For TACO Expressions, embedding similarity will typically correspond to the similarity of the last tensor in the expression. CSS embedding similarity was found to typically be based on the similarity of the last CSS block, and more specifically the similarity of the last properties within said block.

This behavior of these doc2vec embeddings is explored further in section 4.1.3, and Figure 4.19 describes this phenomenon visually.

4.1.2 Overlap between Partial and Complete Programs

In order to further judge the structure of the embedding spaces created with doc2vec, I wanted to assess the location in embedding space of partial programs relative to complete programs. Are they clustered together? Is there some sort of partition in the space for complete vs partial programs? It would be optimal if partial programs that are similar to the original complete program are correspondingly close by in embedding space.

In order to test this, a set of embeddings for each language (with partial programs include this time) were created, and t-SNE plots of the embeddings of each program set. In this case, partial programs in the plots were colored green, while complete programs were colored blue. The following plots are the results of these experiments for each language tested.

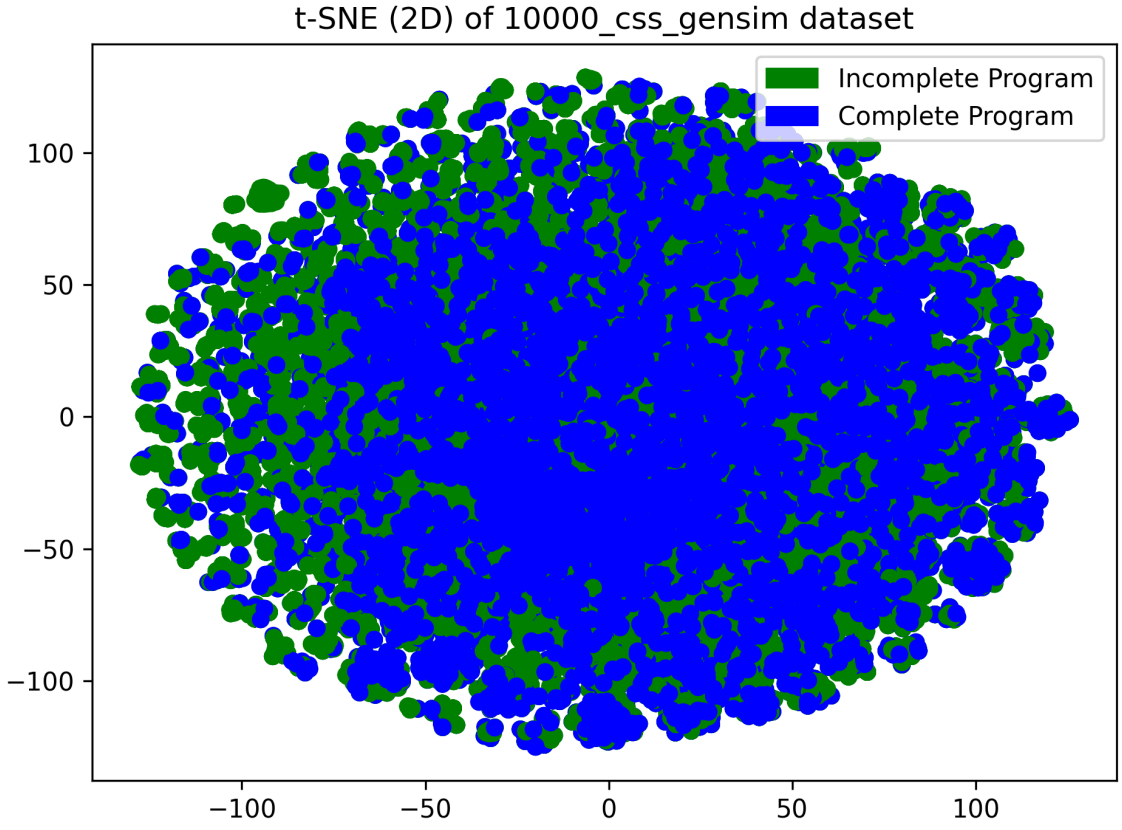


Figure 4.6: t-SNE plot of 10000 complete CSS programs with an additional 83695 partial programs. It is reassuring the see that partial and complete programs overlap each other substantially, with clusters containing both complete and partial programs.

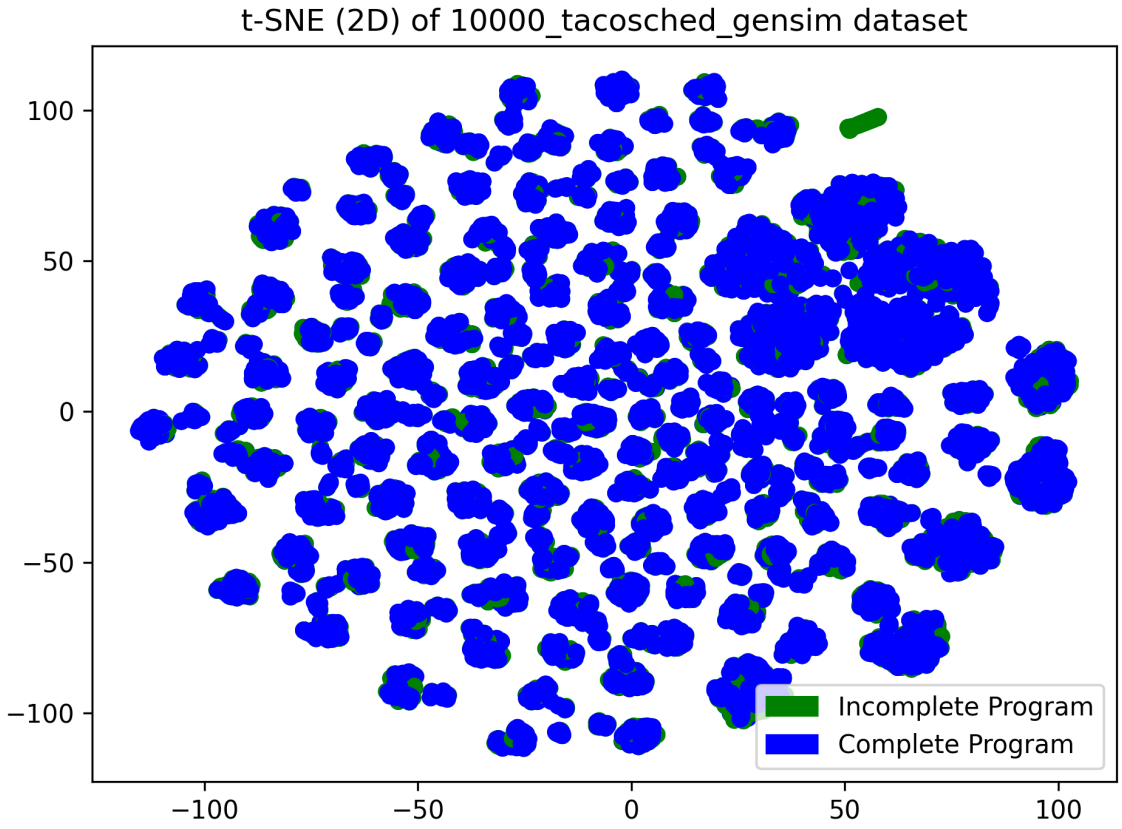


Figure 4.7: Another t-SNE plot, this time with 10000 complete TACO Schedule programs with an additional 12535 partial programs. There is strong indication that one can interpolate among partial and complete programs quite easily depending on the closeness of partial and complete programs based on this diagram, but more quantitative analysis would be required.

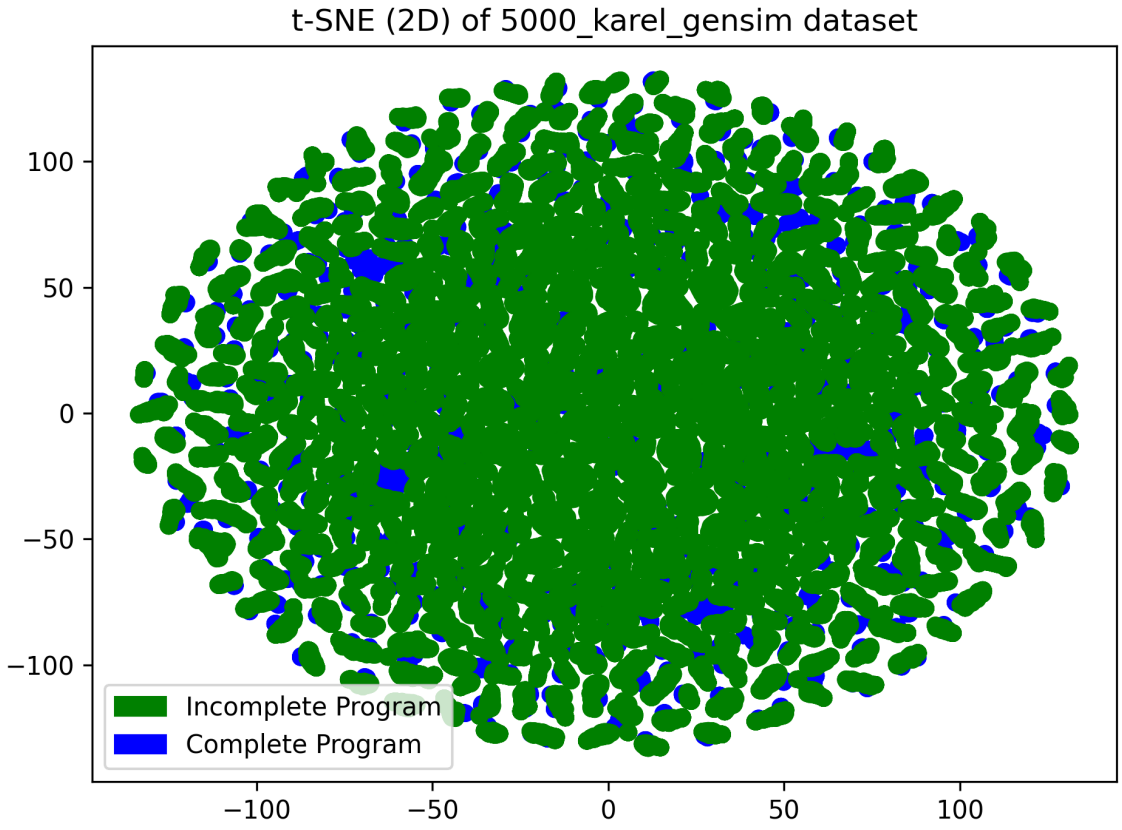


Figure 4.8: Final t-SNE plot of 10000 complete Karel programs with an additional 123191 partial programs. The programs appear to completely fill out the low-dimensional space, in contrast to figures 4.7.

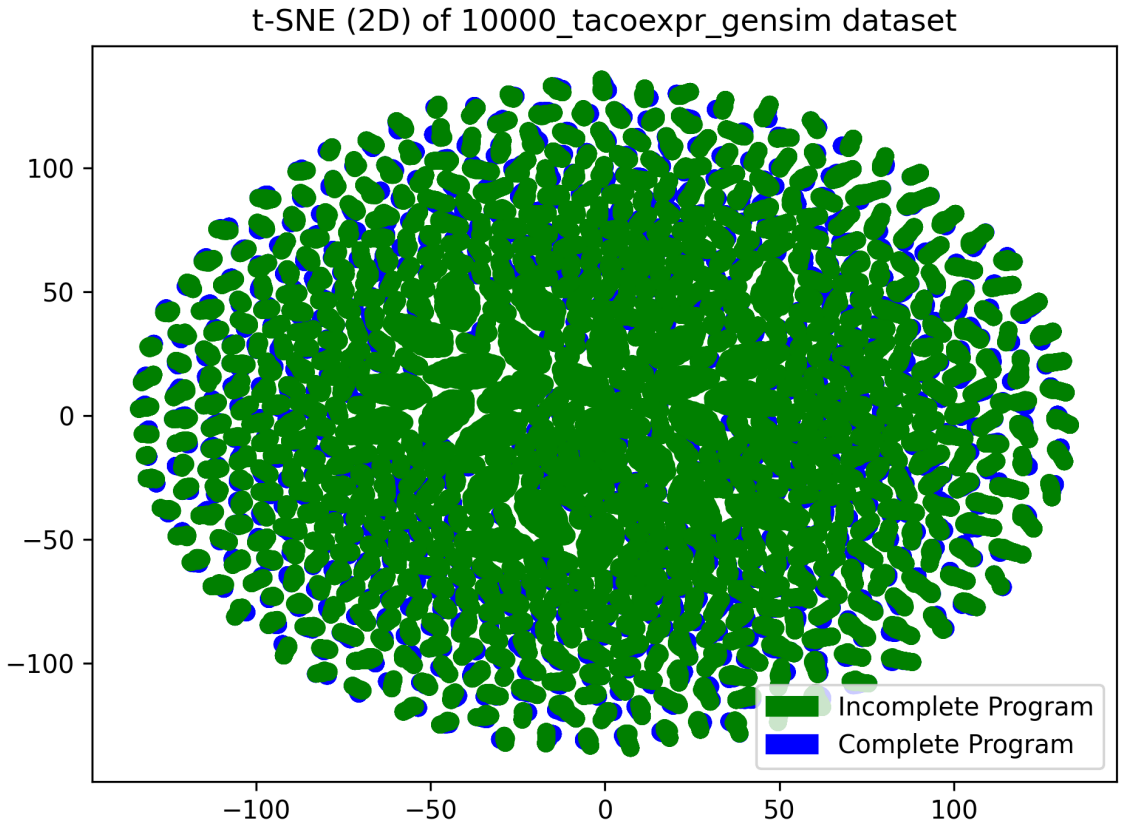


Figure 4.9: Plot of 10000 complete TACO expression programs, with an additional 95348 partial programs. Interestingly, the overall pattern of the entire space being filled matches closely with the pattern generated in figure 4.8.

From this reduced-dimension perspective, it appears that complete programs can and do sit within the space along with partial programs. This helps to reinforce the idea that one could use these embeddings in the context of program expansion to get to a portion of the embedding space that is close to other programs that are optimal. Of course, since this is a low-dimensional representation, more concrete analysis should be done to assess the validity of this claim.

4.1.3 Nearest Neighbor Comparisons

As a further experiment to judge the efficacy of the constructed embeddings, various programs were sampled from generated datasets in each of the languages outlined in section 3.1 and performed nearest neighbor analysis to other programs in the dataset. This was done by comparing the Euclidean distances of the embedding vectors created for each program by doc2vec. The following figures document those results and some analysis follows. The aim in enumerating nearest neighbors for multiple languages is to identify various properties of the embeddings and how they are structured that are invariant of any particular language/sample of programs that are embedded.

The following are some highlighted nearest neighbor samples from testing, with some discussion of the results to follow.

The first programs below come from a CSS dataset of 15000 complete CSS programs with an additional 121591 partial programs embedded as well.

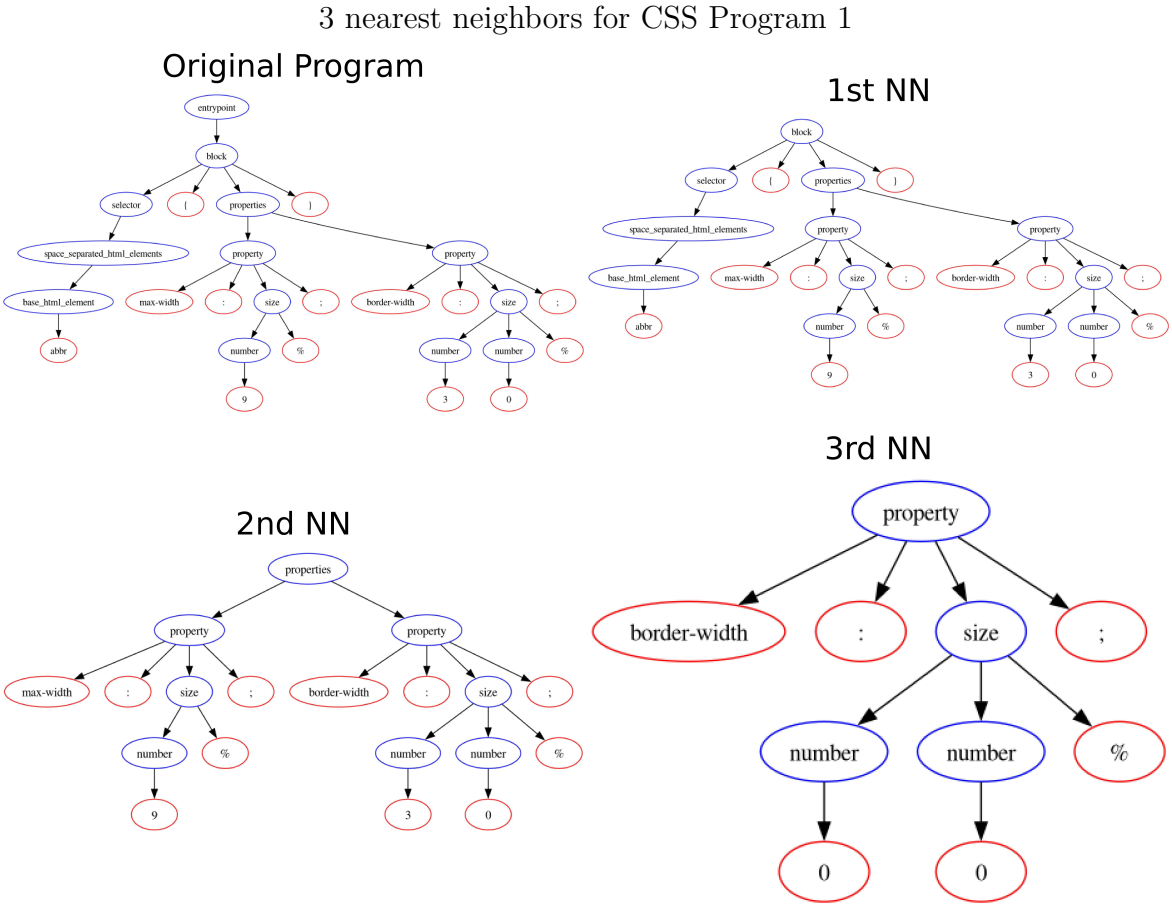


Figure 4.10: First example of the nearest neighbors for a sampled CSS program. An interesting feature of this plot is the structure of the 3rd nearest neighbor is shared structure among all other graphs. Additionally, the first nearest neighbor is the same as the original program with the exception of the entrypoint parent node.

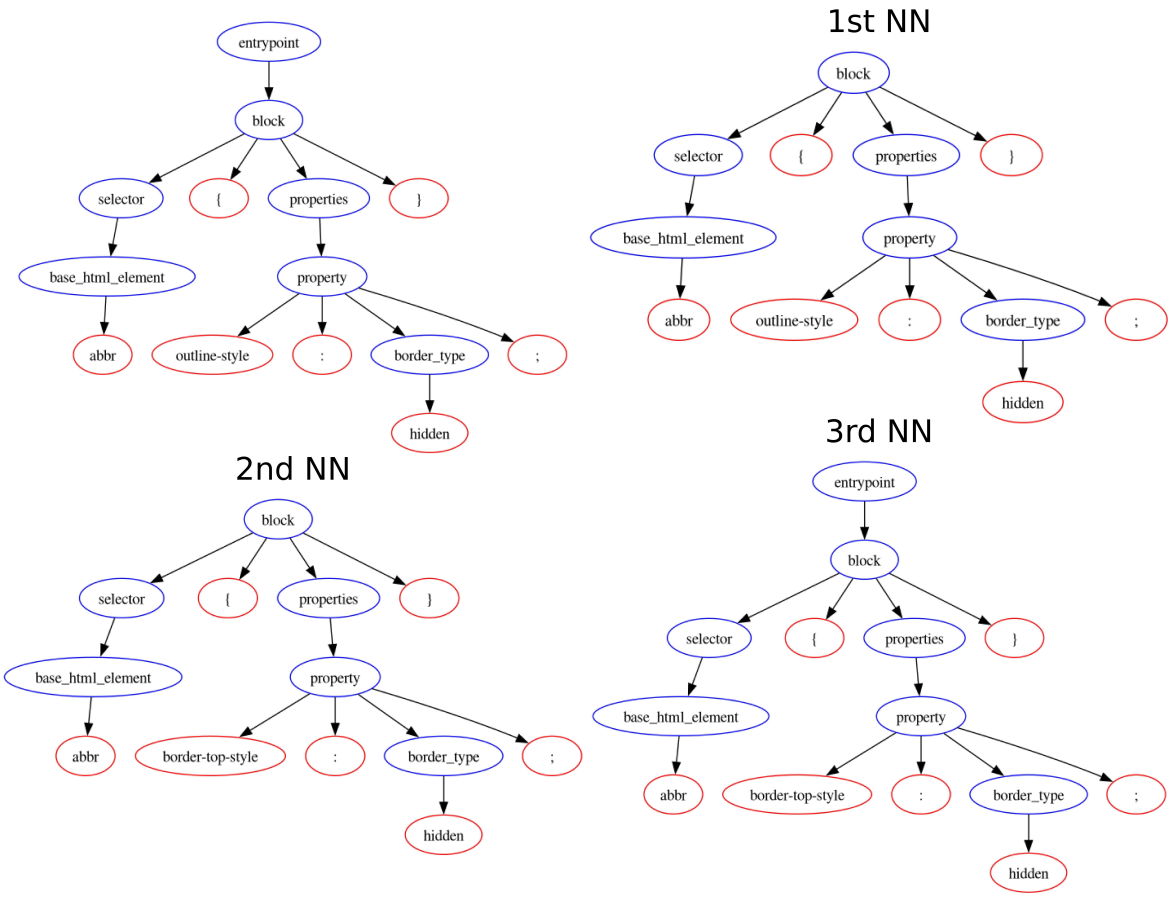


Figure 4.11: Another example of the nearest neighbors for a CSS program. In this case, you can see that the nearest neighbors of the original program are structurally almost identical to the original program. The only change is the property from outline-style to border style.

The next sample of programs comes from a dataset of 15000 randomly sampled complete TACO Schedule programs with an additional 18388 incomplete programs.

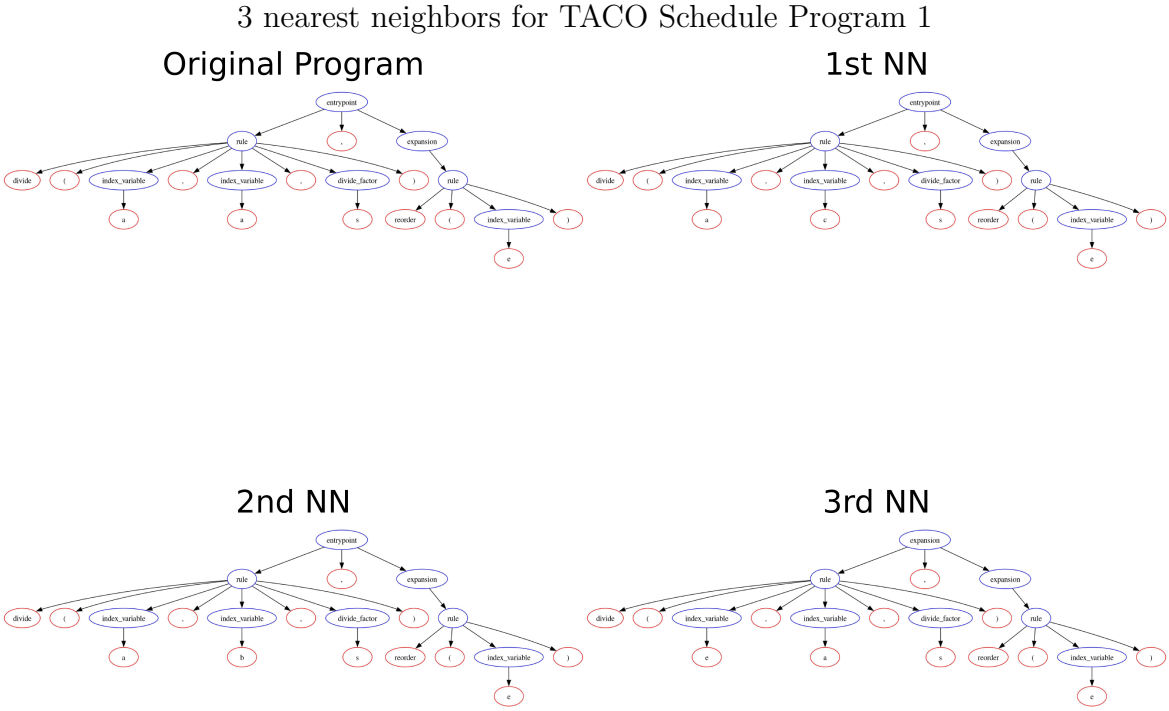


Figure 4.12: First example of a sampled TACO schedule and its nearest neighbors in embedding space. As you can see, the first neighbor is identical to the original in regards to its structure. The only difference is the change in the second index variable on the divide operator. The other neighbors are similarly close with small changes in chosen index variables as well.

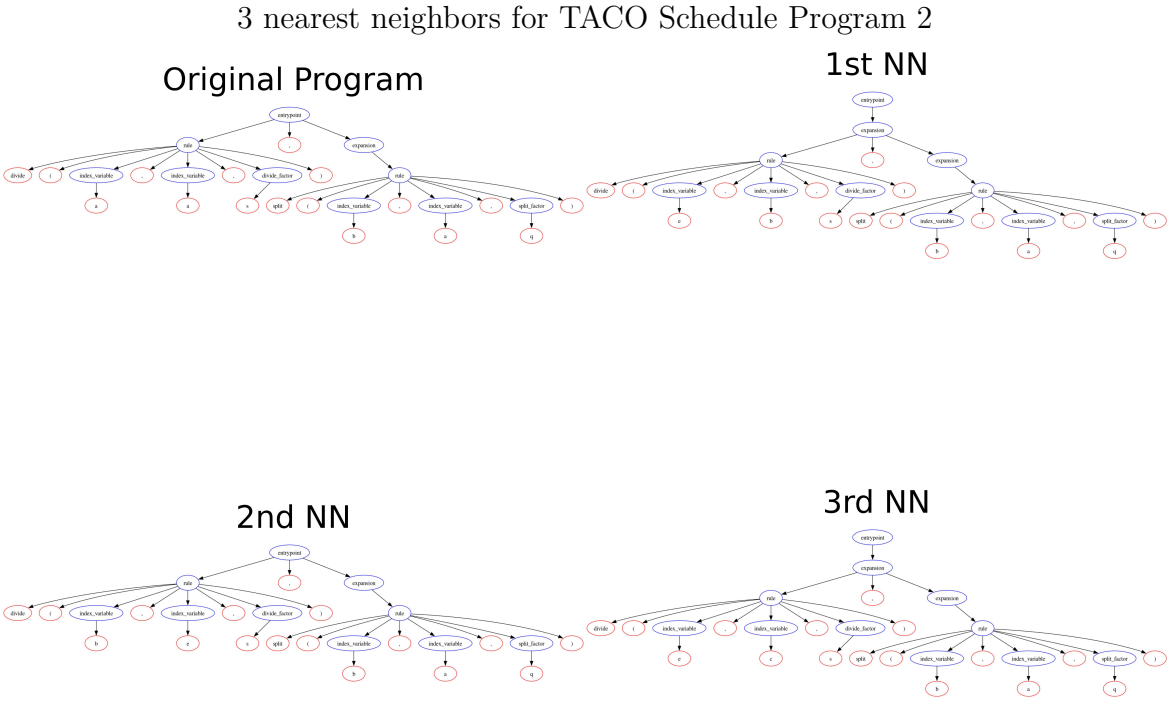


Figure 4.13: Another example of a sampled TACO schedule program and its nearest neighbors. Similar to 4.12, the core structure of the program is preserved across the neighbors for the most part, with small perturbations to the leaf nodes and the root of the tree (with additional expansion rules) being the main differences.

The next samples come from a dataset of 15000 complete Taco Expression programs with an additional 135273 incomplete TACO Expression programs.

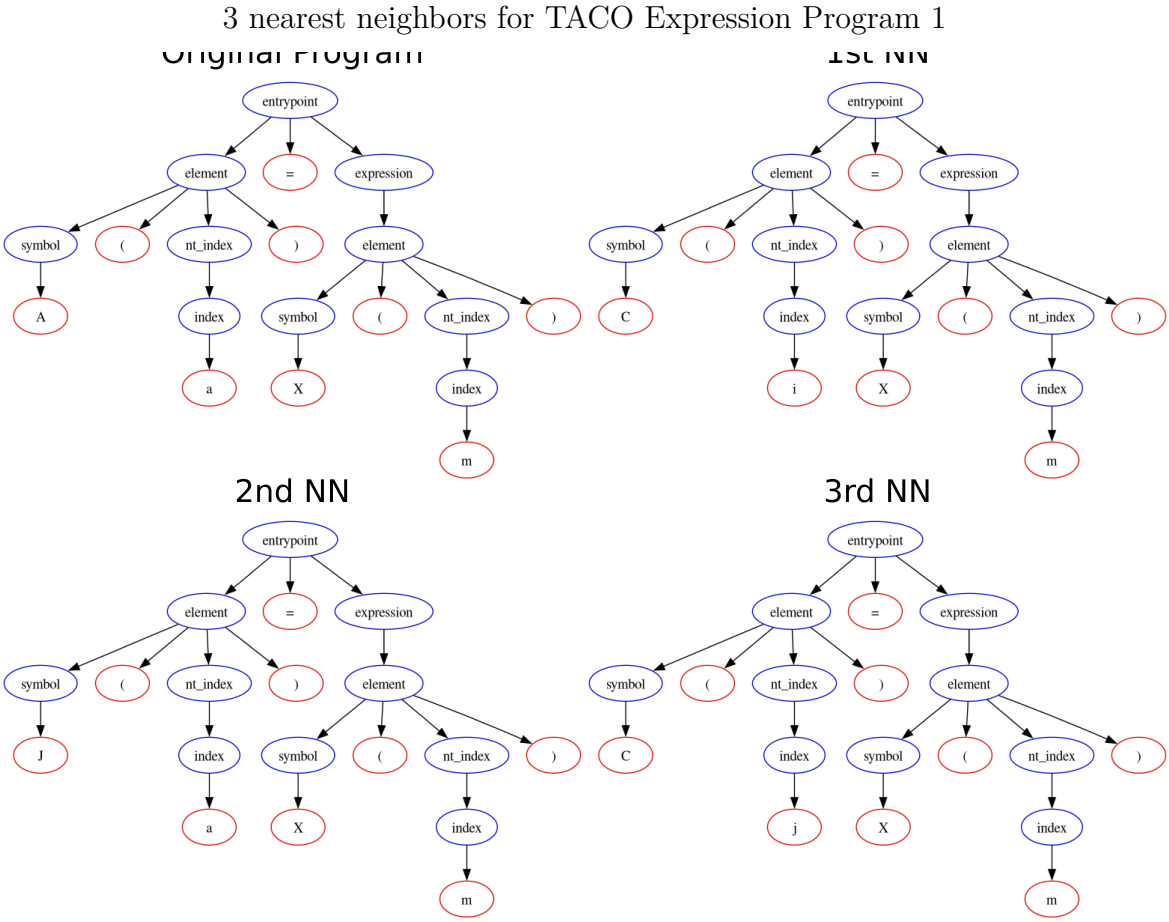


Figure 4.14: First example of nearest neighbors for a TACO expression. In this case, all trees are structurally equivalent (corresponding to an identity program, which is an actual valid program within TACO), with the only changes being symbols. It makes sense that these programs would be similar, since they are part of the same equivalence class.

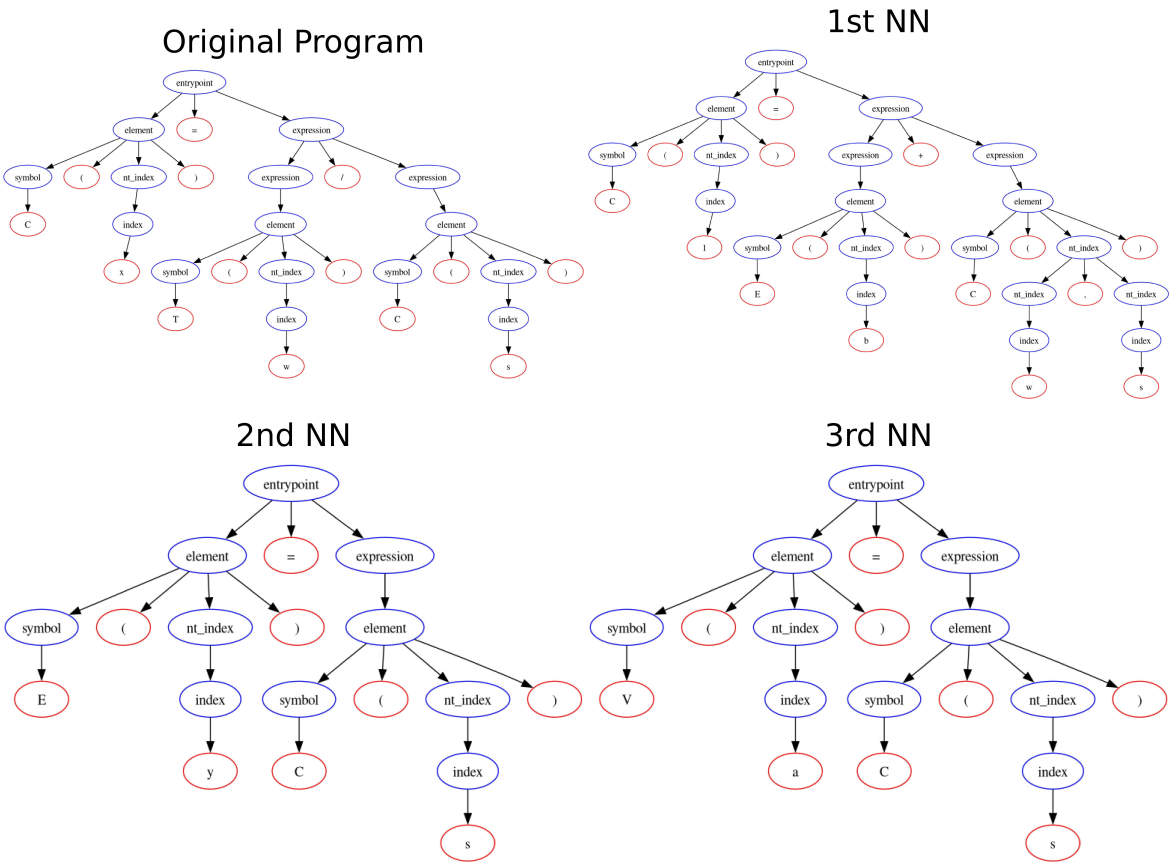


Figure 4.15: Another example of nearest neighbors for a given TACO expression. In this case, the first program is C(x) = T(w) / C(s), where the first neighbor is C(l) = E(b) + C(w, s), the second neighbor is E(y) = C(s), and the third neighbor is V(a) = C(s). So, it would appear that the closest neighbor is a structurally very similar program, with two tensors with a single operation to equate to a 1-dimensional output tensor. The further neighbors deviate more, but interestingly all share the same last tensor of C(s).

The final samples come from a dataset of 5000 complete Karel programs with an additional 128164 incomplete programs.

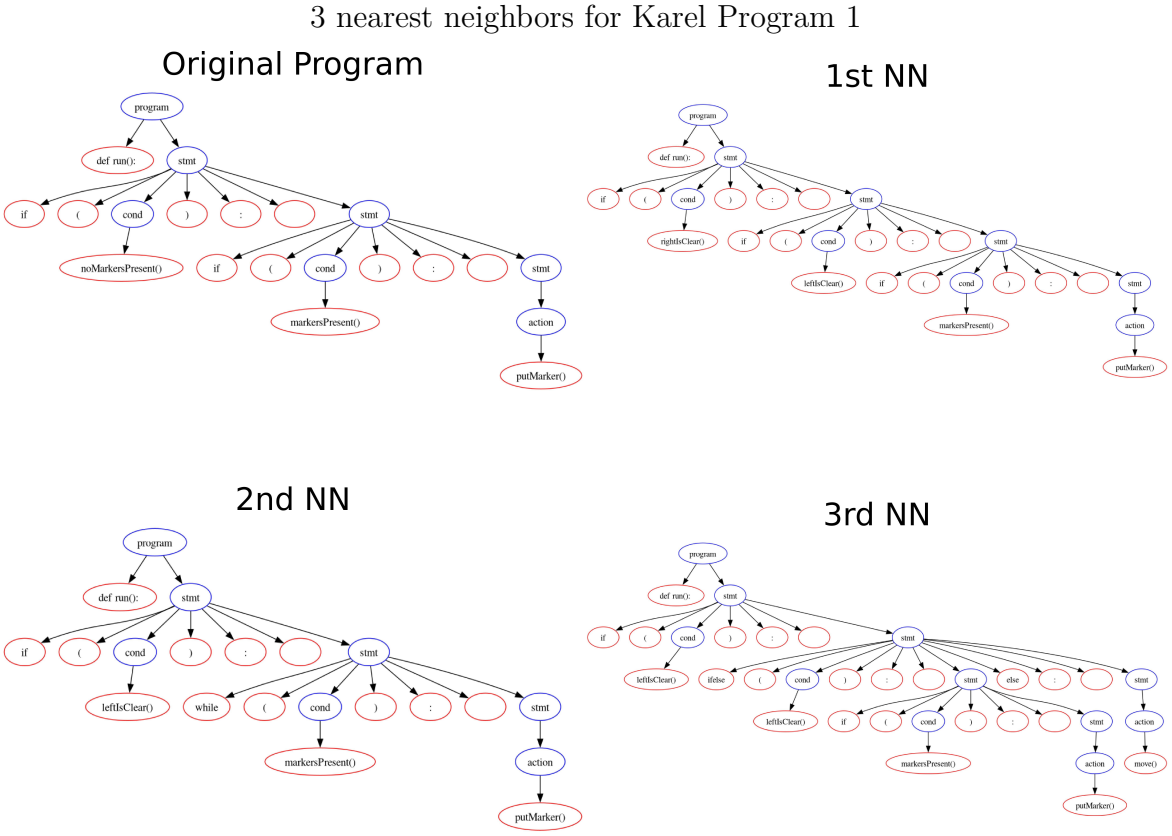


Figure 4.16: First example of a sampled Karel program and it's 3 nearest neighbors. As you can see, the first nearest neighbor has additional conditionals before the last conditional of markersPresent(), but the furthest right-hand side of the AST tree is equivalent to the original program.

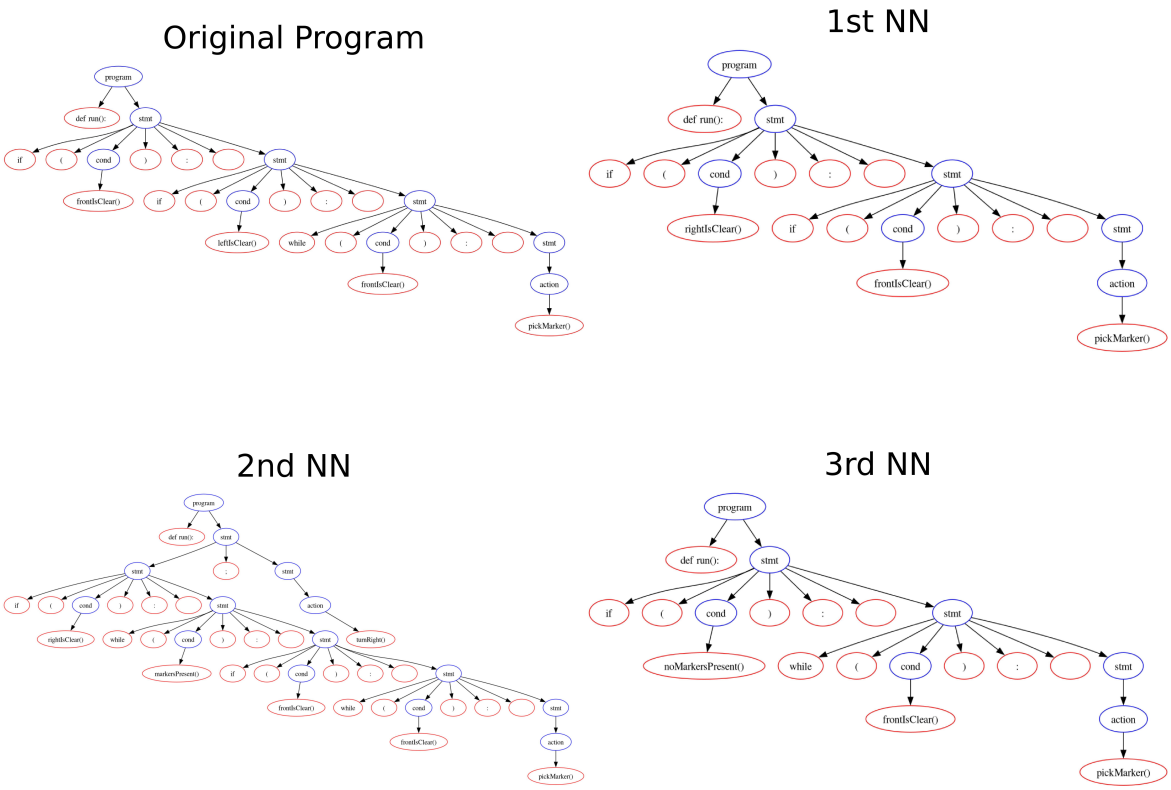


Figure 4.17: Another example of a sampled Karel program with neighbors. Once again, the right tail of the tree is nearly identical among all the graphs. The primary difference among the four plots is first nearest neighbor has switched from using a while loop at the bottom of the tree to an 'if'.

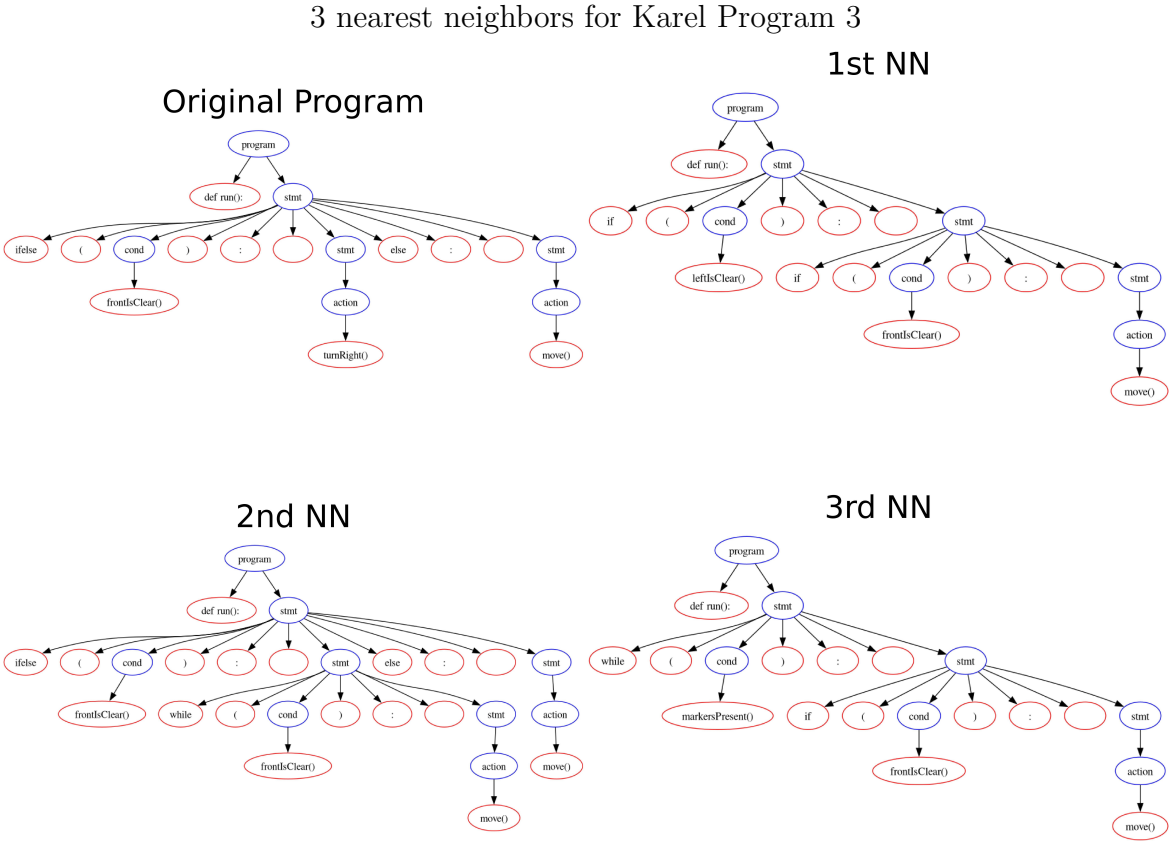


Figure 4.18: Nearest neighbors for final sampled Karel program. In this case, all the given programs appear to run a similar check whether the front is clear, and subsequently move if so at the end of the program. This corresponds to having a similar bottom right side of the tree among all nearest neighbors.

The overall similarity patterns reflect what is described above in section 4.1.1. Within all of these examples and others included in the appendix, there was a consistent pattern of the bottom right of AST trees being the feature most commonly shared among nearest neighbors. This pattern is articulated in Figure 4.19 below.

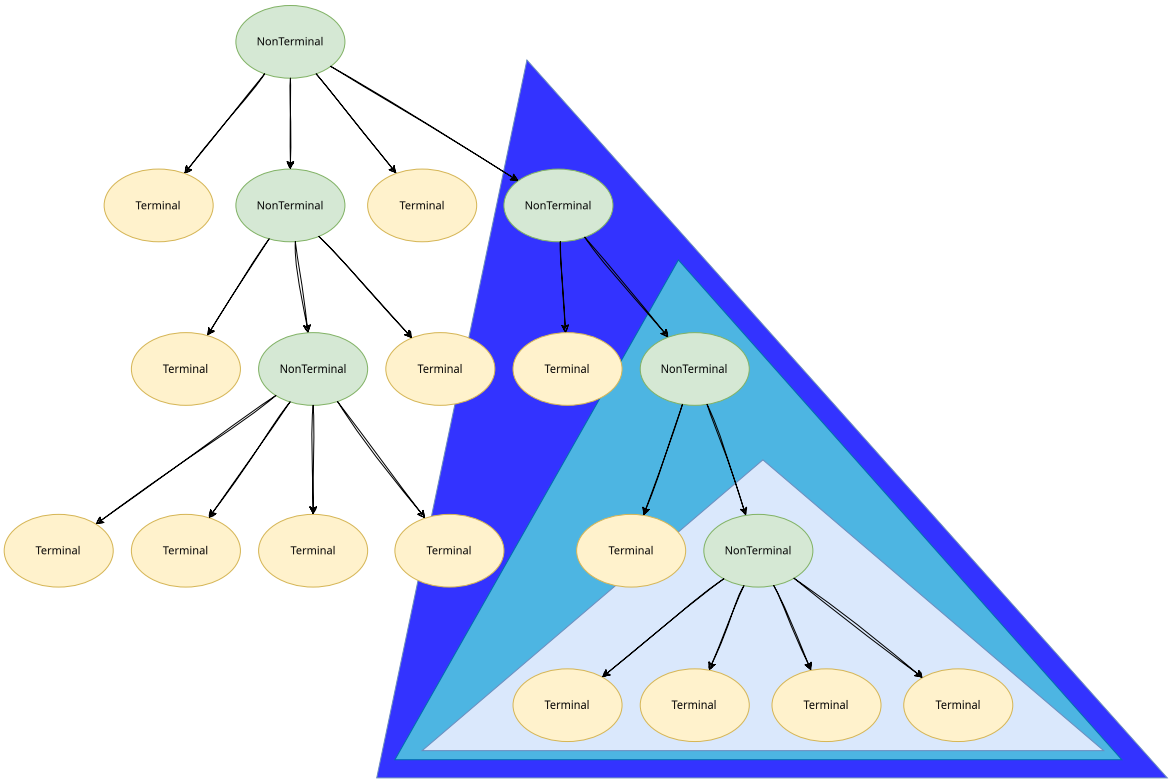


Figure 4.19: Diagram of the perceived similarities of graphs. What was found empirically is that embeddings that are more similar have greater similarities between the corresponding program ASTs. More specifically, programs that were closest may have similarity in the structure encompassed within the largest triangle above, less similar programs could have overlap of structure within the second largest triangle, and even less similar embeddings have similar ASTs with shared structure in the third (and smallest) shared triangle.

4.2 Quantitative Similarity Analysis

In addition to qualitative analysis of the embeddings created with doc2vec, it was necessary to provide some quantitative measure of the efficacy of these embeddings for retaining graph semantics. This also involved comparing quantitative performance with embeddings created from various other SoTA text-embedding models. The models chosen to compare against include Mixed-Bread AI's large embedding model [50], Nomic AI's Embedding model [47], and Snowflake's Arctic Embedding model [48].

One means for measuring the "goodness" of a series of embeddings is trying to quantitatively measure the similarity between two embeddings (using Euclidean distance), measuring the similarity of the two ASTs the vectors are trying to represent, and then comparing the two similarities. If the ASTs are similar and the embeddings are similar, then the embeddings can be considered by the isomorphism metric to be a good continuous representation of the structure. If the embeddings are similar and the ASTs are not, or vice versa, then we can conclude we do not have a very good

representation with the current embedding scheme.

The main challenge with this is creating a quantitative metric by which to compare two graphs. I elected to compare graphs based on how close they were to being isomorphic. This was implemented by comparing the labelsets produced by the Weisfeiler-Lehman graph kernels [5] executed on program ASTs. These sets can then be represented as vectors and compared to each other using some norm measure (in this case Euclidean distance was used). Figure 4.20 describes this process more thoroughly. Each element in each vector represents the number of a particular label within that set. Once each vector is created, with the addition of padded zeroes for labels that are in one vector but not the other, one can compare the distance between each vector and compute a final similarity metric.

Feature Labels:

$$a_1 = \{1, 1, 1, 2, 2, 3, 4, 5, 6, 7, 8, 9, 9\}$$

$$a_2 = \{1, 2, 3, 5, 6, 6, 6, 7, 8, 8, 9, 10, 11\}$$

	1	2	3	4	5	6	7	8	9	10	11
v_1	3	2	1	1	1	1	1	1	2	0	0
v_2	1	1	1	0	1	3	1	2	1	1	1

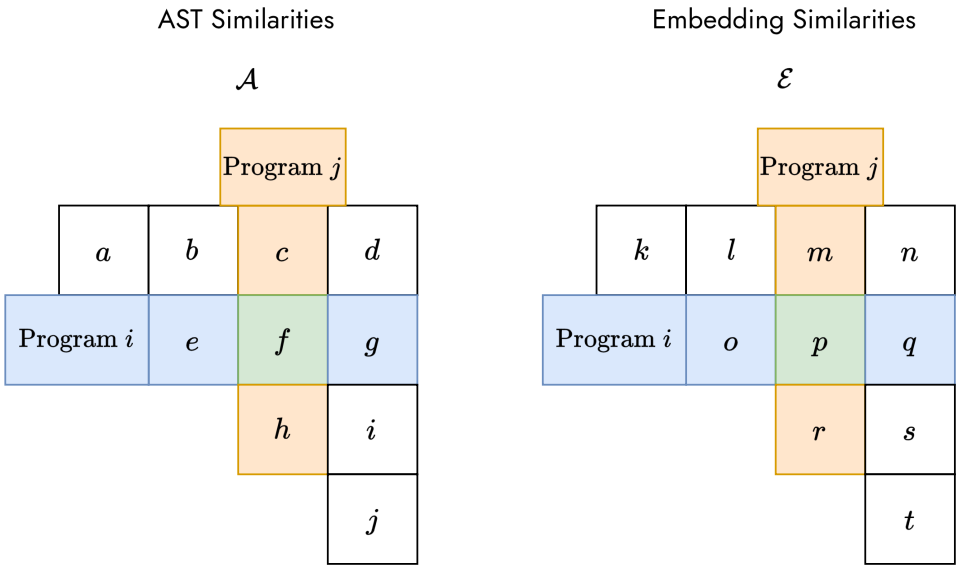
$$||v_1 - v_2||_2 = \sqrt{14}$$

Figure 4.20: Visual example of computing similarity between two ASTs. Once the WL-kernel is run on each AST to extract labels (as shown in sets a_1 and a_2), one can then create feature vectors corresponding to each AST, where each element is the number of each label within the label set. As an example, the first element of v_1 is a 3 because there exist 3x 1's within a_1. Different feature vectors can then be compared for similarity with either the $\mathcal{L}_2$ or $\mathcal{L}_1$ norm.

Now that there is a means for measuring pairwise similarity between any two continuous vectors and their corresponding ASTs, similarity scores for all pairs of embeddings/ASTs in a dataset can be computed. This results in computing a similarity matrix (omitting the lower triangle as the matrix is symmetric with 0s on the diagonal) describing the pairwise similarity between all embeddings/ASTs and their neighbors. The top of Figure 4.21 shows the upper triangular matrices containing the pairwise comparisons.

These two matrices on their own can be very large, and do not provide much information regarding the overall similarity between the embeddings and their underlying discrete representation. This motivates the desire to compute a kind of KL-divergence metric in two dimensions, with non-normalized values. We want to see how different the similarities for each pairwise comparison are from each other.

The approach chosen to compare these matrices is to simply compute the average absolute difference between each pairwise element in each matrix. Figure 4.21 provides a complete overview of the procedure. For each pair of programs i and j, the WL-kernel similarity is computed to give the scalar f. Additionally, each i and j are embedded using a text embedding model, and the Euclidean distance is computed to yield the scalar p. Average similarity takes the average of the difference between the baseline pairwise similarity compared to the corresponding embedding Euclidean distance.



Average Similarity:

$$\frac{1}{|\mathcal{A}|} \sum_{a_i \in \mathcal{A}, e_i \in \mathcal{E}} |a_i - e_i| \rightarrow \frac{1}{10}\left(|a-k| + |b-l| + \ldots + |i-s| + |j-t|\right)$$

Figure 4.21: Overview of how similarities are averaged for each embedding similarity matrix compared to the baseline AST similarity matrix. The pairwise similarities for each embedding in each matrix are then compared against the baseline AST pairwise similarity using the average similarity function.

One problem to account for is if the magnitudes of pairwise similarities vary substantially between the AST distribution and the embedding distribution. This could lead to arbitrarily worse scores simply because scalar values are very different on average. This was accounted for by normalizing each of the distributions using min-max

normalization of the form:

$$x_{\text{new}} = \frac{x_{\text{old}} - \min(x)}{\max(x) - \min(x)}.$$

As with standard pairwise averaging, smaller values represent higher average similarity and are thus considered better under this framework. The following tables describe the results of computing these metrics using datasets from each of the languages described in section 3.1. Since these tables describe average pairwise similarity difference, smaller is better.

Distribution	Simple Average	Normalized Simple Average
doc2vecgensim	**11.787715**	**0.127106**
nomic-embed-text	4.991453	0.167829
mxbai-embed-large	3.461594	0.149734
snowflake-arctic-embed:137m	4.204036	0.136181

Table 4.1: Similarity results for TACO schedule embedding distributions compared to AST distribution. Counterintuitively, performance is almost inverted depending on whether results are normalized. The poor performance of doc2vec in particular is most explained by the low mean/variance of the doc2vec similarity scores compared to AST scores, resulting in high absolute pairwise differences. This motivates the need for normalization, which indicates slightly better performance by doc2vec over the others.

Distribution	Simple Average	Normalized Simple Average
doc2vecgensim	**6.520979**	**0.090355**
nomic-embed-text	3.453399	0.128993
mxbai-embed-large	2.641928	0.125813
snowflake-arctic-embed:137m	3.706221	0.135822

Table 4.2: Similarity results for TACO Expression embedding distributions compared to AST distribution. Similar to table 4.1, doc2vec appears to perform best if you normalize results, and the inverse is true with non-normalized pairwise similarities.

Distribution	Simple Average	Normalized Simple Average
doc2vecgensim	**12.883659**	**0.388166**
nomic-embed-text	3.568656	0.296640
mxbai-embed-large	4.214493	0.276651
snowflake-arctic-embed:137m	6.096455	0.280733

Table 4.3: Similarity results for Karel embedding distributions compared to AST distribution. Counter to the results for TACO schedules and expressions (4.1 and 4.2), doc2vec seems to do consistently worse than the others, regardless of normalization.

Distribution	Simple Average	Normalized Simple Average
doc2vecgensim	**5.943372**	**0.167782**
nomic-embed-text	3.857277	0.121263
mxbai-embed-large	2.344701	0.120693
snowflake-arctic-embed:137m	3.856680	0.117318

Table 4.4: Similarity results for CSS embedding distributions compared to AST distribution. Similar to the Karel data in 4.3, doc2vec seems to do consistently worse regardless of normalization of similarities.

As one can see, normalizing the distributions of similarities has a big effect on the overall metric. The results for doc2vec appear to be mixed, as the TACO languages have better results with doc2vec, but the opposite is true for Karel and CSS.

In addition to computing a single metric to compare the difference of pairwise similarities, histograms of these similarities were computed to get a better idea of how embeddings were spread out throughout the space. Figures 4.22 and 4.23 describe the similarities between ASTs within a generated dataset for a particular language, using the similarity techniques described above. Similarly, Figures 4.24 and 4.25 show how pairwise similarities vary depending on the language and the embedding model.

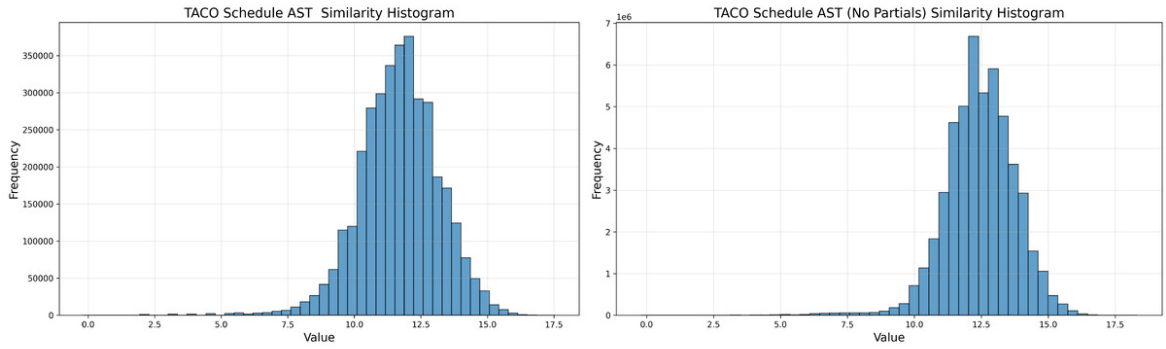


Figure 4.22: AST Similarity Distributions for generated TACO schedules, both for datasets with partials (left), and without partials (right). Both distributions appear to be roughly Gaussian, but the mean is perhaps slightly higher for the no-partials distribution.

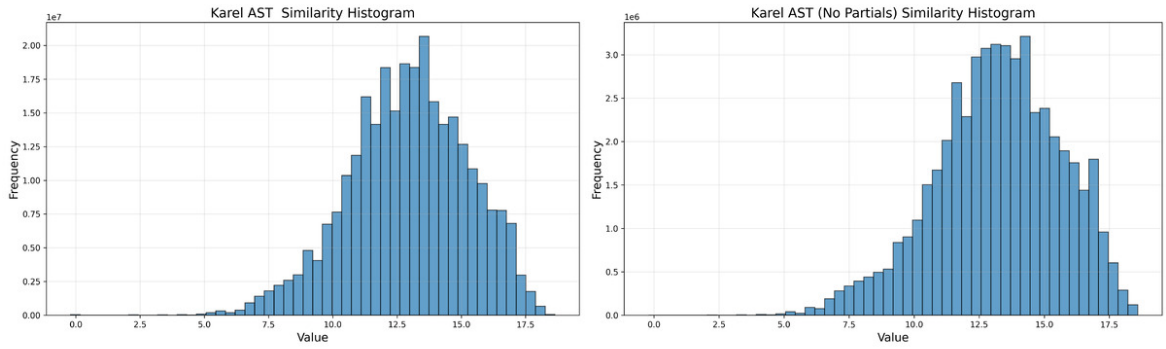


Figure 4.23: AST Similarity distributions for generated Karel programs, both for datasets with partials (left), and without partials (right). Similar to 4.22, the distribution of pairwise similarities looks roughly Gaussian, but with much higher variance.

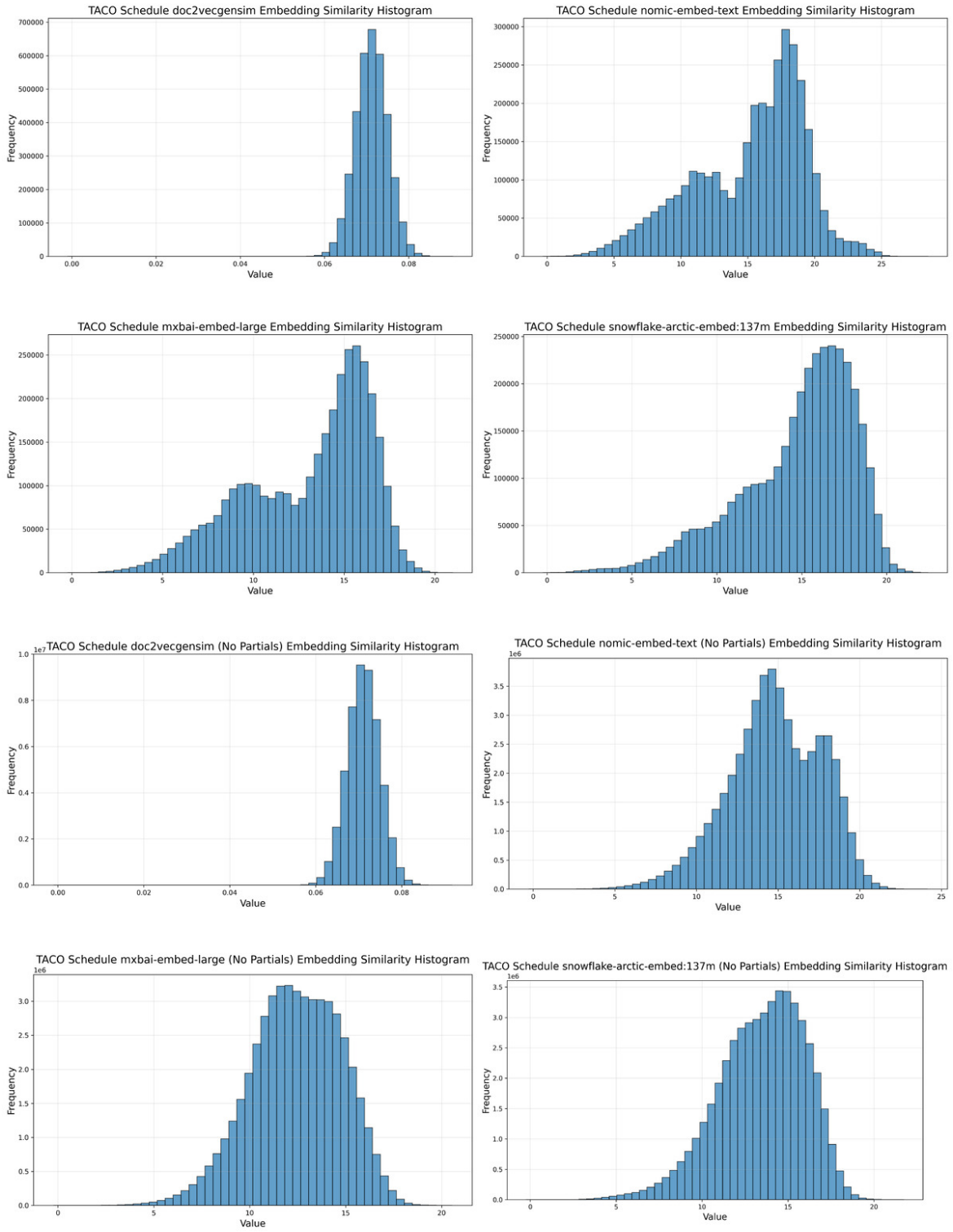


Figure 4.24: Histograms of the pairwise Euclidean similarity for every embedding of every program in a synthetic TACO Schedule dataset. The top 4 plots describe the distribution for a dataset with partial programs included, while the bottom four plots do not include partial programs (i.e., only complete programs were embedded). An interesting feature of the plots is the low variance of both doc2vec gensim plots, indicating that on average vectors are roughly the same (relatively small) distance from each other.

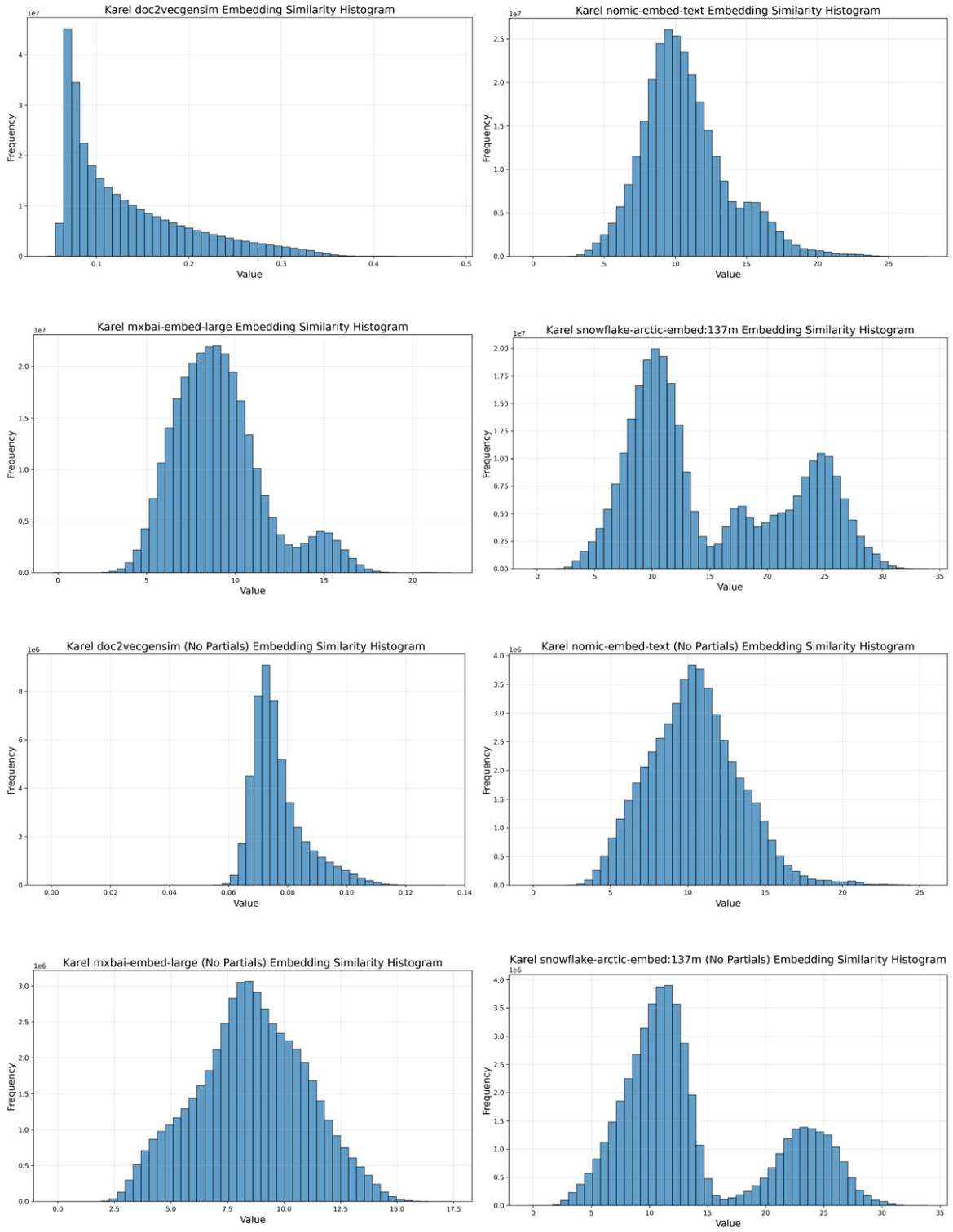


Figure 4.25: Histograms of pairwise Euclidean similarity for every embedding of every program in a synthetic Karel dataset. Similar to 4.24, the top 4 plots are for a dataset with partial programs, while the bottom four are the same distribution without partials included.

4.3 Correlation Analysis

Additional correlation analyses were conducted to observe the relation between AST similarity and embedding similarity, both for doc2vec and the other text-embedding models. This involved running a correlation analysis on a subset of computed pairwise similarity scores for synthetic datasets generated for each language described in this thesis, with embedding scores sourced from every tested model. Figures 4.26, 4.27, 4.28, and 4.29 show the correlation between normalized AST and embedding similarity for Karel programs. Even though normalized values will not change the correlation results, they were chosen to provide plots that are less distorted than they may be with non-normalized values.

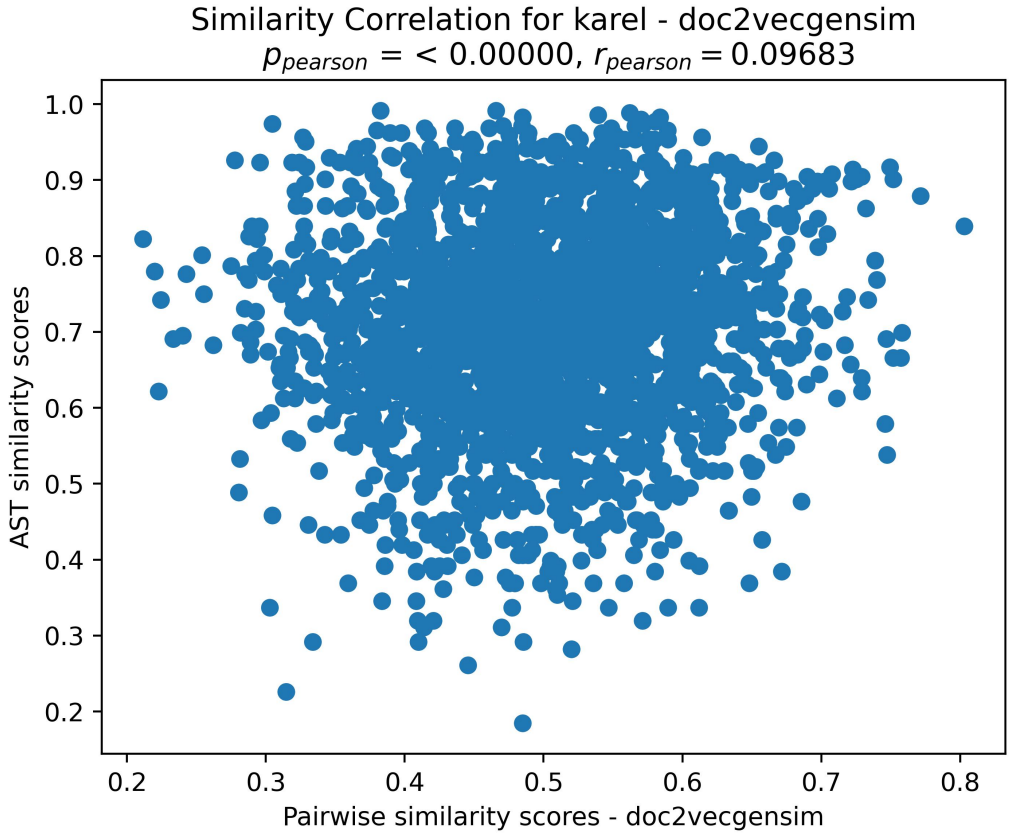


Figure 4.26: Similarity Correlation plot for Karel programs with embeddings from Doc2vec. The Pearson p and r-values are provided at the top of the figure, with the alternate hypothesis being a positive correlation.

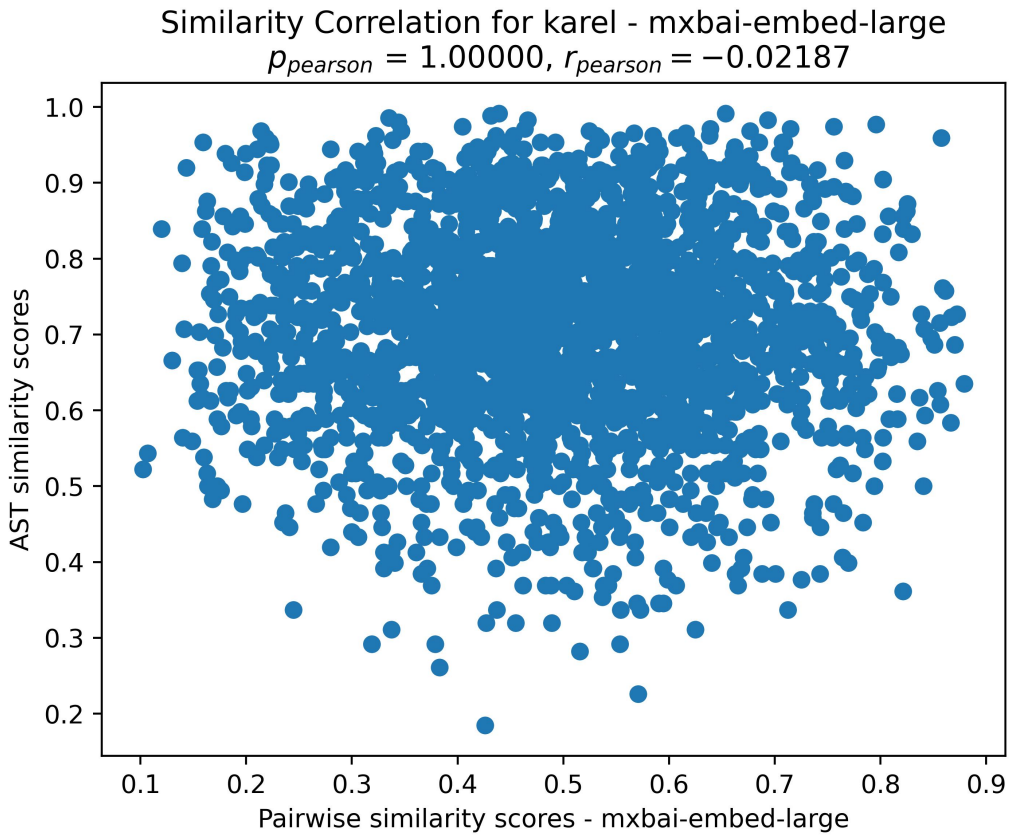


Figure 4.27: Similarity Correlation plot for Karel programs from mxbai-embed-large. Pearson p and r-values are shown the the top of the figure, with the alternate hypothesis being a positive correlation.

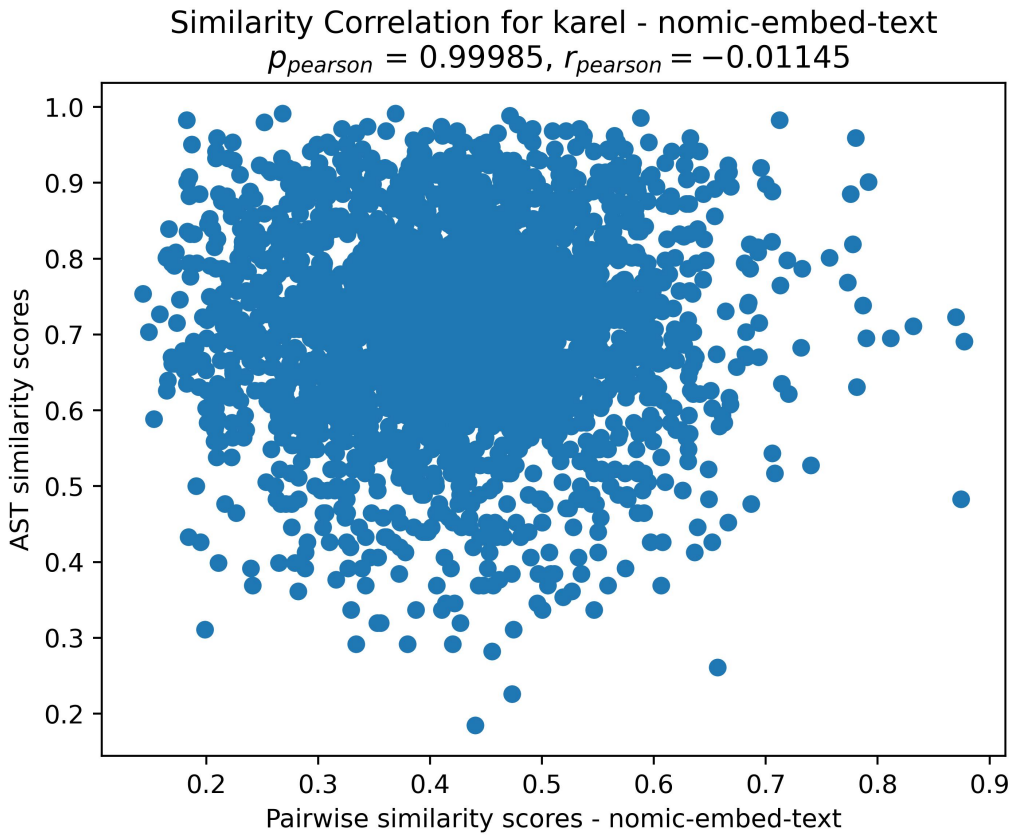


Figure 4.28: Similarity Correlation plot for Karel programs utilizing the nomic-embed-text model. Pearson p and r-values are shown the the top of the figure, where the alternate hypothesis is a positive correlation.

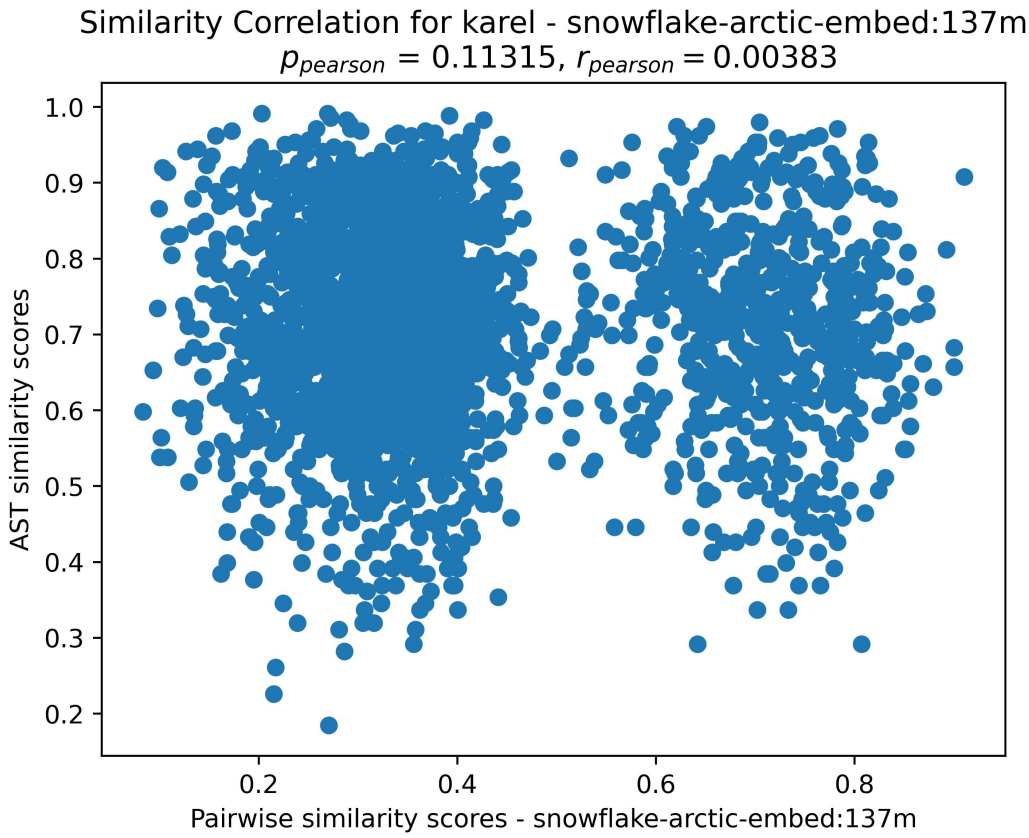


Figure 4.29: Similarity Correlation plot for Karel programs from snowflake-arctic-embed:137m. Pearson p and r-values are shown the the top of the figure, where the alternate hypothesis is a positive correlation.

The initial appearance of these correlation plots is less than promising. For doc2vec, there is a minor correlation between AST similarity and embedding similarity, although very weak. Doc2vec does do consistently better than the other text-based embedding models, which is a promising result. This is aided with lower p-values for doc2vec, indicating we can reject the null hypothesis of zero correlation between AST similarity and doc2vec embedding similarity. Figure 4.29 shows a unique partition of the data points into two clusters. My current theory behind this large split in the plot is perhaps due to various clusters of Karel embeddings in the embedding space created by the Snowflake model. This could lead to clusters that are near each other, but then far apart from other clusters, leading to a disparity in average pairwise embedding distances. The phenomenon correlates to the multi-hump histogram of Snowflake pairwise embedding distances in Figure 4.25.

The same experiments were conducted with similar to worse results. Those plots are included in Section 5.4 of the appendix.

4.4 Discussion and Limitations

The experiments highlighted within this section are the culmination of the efforts that were made to attempt to create and analyze the program spaces that are constructed using various techniques with various domain-specific languages. The first experiments with t-SNE plots of sampled datasets (section 4.1.1) indicate that there is clustering that takes place with completed programs. For some languages, it appears that doc2vec likes to prioritize grouping of programs by the edges of the program AST, but will also keep close permutations of the graph close together in embedding space as well.

The second experiments involving the overlap of complete and partial programs (section 4.1.2) tend to indicate that partial graphs that are similar to complete graphs will indeed cluster together. This is a useful property of the space, and helps to indicate the usefulness of this embedding paradigm in future reinforcement learning work.

The sampled nearest neighbor plots in section 4.1.3 help to show how doc2vec chooses to cluster programs together. Although there are instances where close permutations to the original program are close together in embedding space, the more common trend is that nodes closer to the end of program ASTs are weighed more heavily in determining similarly (which usually corresponds to the last tensor in a TACO expression, the last scheduling directive in TACO schedules, the last property block in CSS, etc.) over other regions of the AST. The reasoning behind this is still not very well understood and more experiments need to be designed to gain more insight into this phenomenon.

Section 4.2 looks into how to provide a quantitative measure of the fitness of an embedding implementation. This involves comparing pairwise similarities between embeddings and their corresponding AST embeddings in order to acquire a similarity score. It is important to note that the big assumption this metric relies on is that AST structure tells the entire story about the true semantic similarity of any two programs. If two ASTs are structurally very similar, this metric will reward that, even if the semantic structure of the programs could be very different. This result most directly contributes to the original research questions posed at the beginning of this thesis. Averaged absolute pairwise difference allows one to quantitatively measure how structural similarity is retained among various embeddings.

There are indications in the sampled nearest neighbors experiments that structure can be a good determinant of similarity, but there also exist counterexamples. One good example in favor of structural similarity is figure 4.15. In that example, semantically similar identity programs are placed close together in embedding space. On the other hand, a good counterexample to that notion is figure 4.11. In that example, various programs that style **abbr** HTML elements found their embeddings to be close

together. Structurally, the ASTs are nearly identical, but two programs apply the property **outline-style: hidden** while the other two apply **border-top-style: hidden**. One could argue that these two programs could produce very different results: one is hiding all outlines around that element, while the other is only hiding the top border styling of any **abbr** elements. Of course, that is a subjective assessment, and more generally, likely *all* issues of semantic similarity are subjective That is a core difficulty of this kind of work, and more must be done to try and look for more general metrics that can capture this nuance.

Additionally, plots were added to describe the pairwise similarity distributions to emphasize the motivation behind doing min-max normalization. This is most prevalent with doc2vec. In figures 4.24 and 4.22, one can clearly see the stark difference in the means of the distributions for TACO schedule pairwise AST similarity and doc2vec pairwise similarities (see the top-left most histogram). This results in large element-wise differences and worse scores.

These histograms also help to indirectly show how the embedding spaces are laid out. For example, in the bottom-right most histogram in 4.25, there is a distinctive double-hump in the histogram. These humps exists regardless of the existence of partial programs, and it leads one to wonder how embeddings could be spread throughout the space. This is mostly interesting because there is so much variety between both the different embedding models and the languages. doc2vec embeddings are all very close to each other, indicating perhaps they are optimized to live within a small subspace of $\mathbb{R}^{128}$. The different dimensions of each of the newer embedding models likely has something to do with the larger average magnitudes, since the Nomic, and Snowflake embedding models output 768-dimensional vectors, and Mixed Bread's model outputs 1024-dimensional vectors. But there is no clear explanation for the overall variety in histogram shape.

Normalized average pairwise similarities appear to show that doc2vec works best for the TACO languages, but less so for Karel and CSS. It would be interesting to see how the metric varies for other languages with other structural properties.

The final experiments conducted involved performing a correlation analysis between AST similarities and embedding similarities for each language and embedding model discussed in this thesis. The overall correlations were not very promising, but there was a slight advantage offered by Doc2vec over the other models, though not consistent across all languages. Because of this, more research must be conducted to find both better embedding techniques that capture semantic similarity better (whatever those semantics may be), as well as better metrics for measuring semantic similarity of ASTs.

It is worth reiterating that all the experiments performed in this section were done with synthetic program datasets that each contained roughly 10000-20000 programs.

All provided grammars are infinite grammars, so by definition a very limited portion of program space was sampled in these experiments. Similarly, the program grammars are subsets of the real grammars that define the languages in the real world, so there are certainly details from the real languages that are left out from these experimental results. Finally, programs were sampled from program spaces relatively uniformly, and future research may benefit for imbuing other human-constructed heuristics so that randomly generated programs are sampled from a region of program space that is more consistent with real-world programs. This could also be resolved with the construction of parsers for each of the provided languages to generate datasets from the internet.

Chapter 5

Future Work

The construction of this thesis has answered many questions I had before beginning, but like most research has left me with more. With regard to building embeddings, the limitations around sample size within a program space are something that should be resolved with more compute resources and the ability to distribute the embedding table across multiple GPUs. Additionally, further quantitative methods for measuring the effectiveness of the embedding mapping are required to build robust systems using this technology.

As noted in the introduction, the goal of this research is ultimately to automatically create good programs by expanding formal grammars, using whatever definition one may provide for good. The next steps involved in achieving this are to complete the work around the Learned Expander as described in Section 3.4.3. This mainly entails designing the training regime and how backpropagation will be implemented for each of the models. Infrastructure for measuring performance of generated programs to return rewards must also be constructed.

There are a number of hyperparameters that were created/discovered over the course of this work, and many of them were not adjusted. This includes higher order algorithms such as the choice of optimizer used, the number of WL kernel iterations when extracting subgraph-labels in the document creation process, the dimensions on the embeddings themselves, epoch counts, adaptive learning rates, the negative sampling count, and much more. It would be interesting to run some kind of hyperparameter search for all of the models and systems within this framework, as this could yield marginal improvements in results without needing substantial amounts of human critical thought.

Additionally, the current architectures proposed for various tasks within this thesis are quite simplistic, where the heuristics used for design decisions are very high-level. As a result, there exists potential for architectures to not best fit the problem space, and a resulting need to tune model architectures more to the specific problem domain. Thus, it might also be worth performing some kind of neural architecture search to find

better architectures for production expansion models [61, 62], as well as for embedding classifiers like those used in the Doc2Vec work. One of the goals of neural architecture search is to operate on smaller mathematical primitives than what is traditionally done in machine learning model design, and work towards building optimal architectures from the ground up. This appears to be another interesting area of research in spite of the computational costs.

All models discussed in this thesis, and in fact all that I have found in general in my research have been statically defined mathematical functions. So on top of searching for good static architectures, it may be worth considering the implications and benefits of adjusting all the discussed models to perform computation dynamically, or making metadecisions for downstream computations to be made. The field of dynamic neural networks (DNNs) [63] is an active one, and could be an interesting direction to take this or other machine learning research conducted in the future. Neural Architecture search and DNNs could be particularly interesting for doc2vec/code2vec since they use simple linear classifiers for parts of each model. There have been rebuttals made specifically towards code2vec that the current models do not generalize sufficiently for various downstream tasks [64]; perhaps model structure is part of the problem.

Future work can also involve exploration into multi-objective optimization methods for synthesizing new programs, as a programmer is typically trying to optimize for various factors (speed, readability, correctness, etc.) when writing programs. Additionally, more work should be done to increase generalization of this approach (which is lacking, especially in comparison to the generality of token spaces used in modern transformers). Finally, it is my hope that the grammars for more advanced DSLs as well as general purpose languages can be codified within lang-explorer, and further research can be conducted to observe the capabilities of this framework in generating latent representations and subsequently programs from such languages.

Bibliography

[1] OpenAI et al. *GPT-4 Technical Report*. 2024. arXiv: 2303.08774 [cs.CL].

[2] Aaron Grattafiori et al. *The Llama 3 Herd of Models*. 2024. arXiv: 2407.21783 [cs.AI].

[3] Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Romain Sauvestre, Tal Remez, Jérémy Rapin, Artyom Kozhevnikov, Ivan Evtimov, Joanna Bitton, Manish Bhatt, Cristian Canton Ferrer, Aaron Grattafiori, Wenhan Xiong, Alexandre Défossez, Jade Copet, Faisal Azhar, Hugo Touvron, Louis Martin, Nicolas Usunier, Thomas Scialom, and Gabriel Synnaeve. *Code Llama: Open Foundation Models for Code*. 2024. arXiv: 2308.12950 [cs.CL].

[4] DeepSeek-AI et al. *DeepSeek-V3 Technical Report*. 2025. arXiv: 2412.19437 [cs.CL].

[5] Nino Shervashidze, Pascal Schweitzer, Erik Jan van Leeuwen, Kurt Mehlhorn, and Karsten M. Borgwardt. "Weisfeiler-Lehman Graph Kernels". In: *J. Mach. Learn. Res.* 12.null (Nov. 2011), 2539–2561. ISSN: 1532-4435.

[6] Steve Klabnik and Carol Nichols. *The Rust Programming Language, 2nd edition*. No Starch Press, 2023.

[7] Quoc Le and Tomas Mikolov. "Distributed Representations of Sentences and Documents". In: *Proceedings of the 31st International Conference on Machine Learning*. Ed. by Eric P. Xing and Tony Jebara. Vol. 32. Proceedings of Machine Learning Research 2. Bejing, China: PMLR, 2014, pp. 1188–1196.

[8] Donald E Knuth. "Semantics of context-free languages". In: *Mathematical systems theory* 2.2 (1968), pp. 127–145.

[9] Arto Salomaa and Ian N. Sneddon. *Theory of Automata*. Pergamon Press Reprint, 1969. ISBN: 0080133762.

[10] *Java syntax reference*. https://docs.oracle.com/javase/specs/jls/se7/html/jls-18.html, Accessed October 4th, 2025.

[11] *Grammar summary - The Rust Reference*. https://doc.rust-lang.org/reference/grammar.html, Accessed October 4th, 2025.

[12] *Golang syntax reference*. https://go.dev/ref/spec, Accessed October 4th, 2025.

[13] M. Balog, A. L. Gaunt, M. Brockschmidt, S. Nowozin, and D. Tarlow. "DeepCoder: Learning to Write Programs". In: *Proceedings International Conference on Learning Representations (ICLR)*. Apr. 2017.

[14] Rudy Bunel, Matthew Hausknecht, Jacob Devlin, Rishabh Singh, and Pushmeet Kohli. "Leveraging Grammar and Reinforcement Learning for Neural Program Synthesis". In: *International Conference on Learning Representations*. 2018.

[15] Riley Simmons-Edler, Anders Miltner, and Sebastian Seung. "Program Synthesis Through Reinforcement Learning Guided Tree Search". In: abs/1806.02932 (June 2018).

[16] Yanju Chen, Chenglong Wang, Osbert Bastani, Isil Dillig, and Yu Feng. "Program Synthesis Using Deduction-Guided Reinforcement Learning". In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*. Vol. 12225 LNCS. Springer, 2020, pp. 587–610. ISBN: 9783030532901. DOI: 10.1007/978-3-030-53291-8_30.

[17] Kevin Ellis, Catherine Wong, Maxwell Nye, Mathias Sablé-Meyer, Lucas Morales, Luke Hewitt, Luc Cary, Armando Solar-Lezama, and Joshua B. Tenenbaum. "DreamCoder: bootstrapping inductive program synthesis with wake-sleep library learning". In: *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. PLDI 2021. Virtual, Canada: Association for Computing Machinery, 2021, 835–850. ISBN: 9781450383912. DOI: 10.1145/3453483.3454080.

[18] DeepSeek-AI et al. *DeepSeek-R1: Incentivizing Reasoning Capability in LLMs via Reinforcement Learning*. 2025. arXiv: 2501.12948 [cs.CL].

[19] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Łukasz Kaiser, and Illia Polosukhin. "Attention is all you need". In: *Proceedings of the 31st International Conference on Neural Information Processing Systems*. NIPS'17. Long Beach, California, USA: Curran Associates Inc., 2017, 6000–6010. ISBN: 9781510860964.

[20] Yue Zhang, Yafu Li, Leyang Cui, Deng Cai, Lemao Liu, Tingchen Fu, Xinting Huang, Enbo Zhao, Yu Zhang, Yulong Chen, Longyue Wang, Anh Tuan Luu, Wei Bi, Freda Shi, and Shuming Shi. "Siren's Song in the AI Ocean: A Survey on Hallucination in Large Language Models". In: *Computational Linguistics* (Sept. 2025), pp. 1–46. ISSN: 0891-2017. DOI: 10.1162/COLI.a.16. eprint: https://direct.mit.edu/coli/article-pdf/doi/10.1162/COLI.a.16/2535477/coli.a.16.pdf.

[21] Andrew Jesson, Nicolas Beltran-Velez, Quentin Chu, Sweta Karlekar, Jannik Kossen, Yarin Gal, John P. Cunningham, and David Blei. “Estimating the Hallucination Rate of Generative AI”. in: *Advances in Neural Information Processing Systems*. Ed. by A. Globerson, L. Mackey, D. Belgrave, A. Fan, U. Paquet, J. Tomczak, and C. Zhang. Vol. 37. Curran Associates, Inc., 2024, pp. 31154–31201.

[22] Ziwei Ji, Nayeon Lee, Rita Frieske, Tiezheng Yu, Dan Su, Yan Xu, Etsuko Ishii, Ye Jin Bang, Andrea Madotto, and Pascale Fung. “Survey of Hallucination in Natural Language Generation”. In: *ACM Computing Surveys* 55.12 (Mar. 2023), 1–38. ISSN: 1557-7341. DOI: 10.1145/3571730.

[23] Vishaal Udandarao, Ameya Prabhu, Adhiraj Ghosh, Yash Sharma, Philip H.S. Torr, Adel Bibi, Samuel Albanie, and Matthias Bethge. “No "zero-shot" without exponential data: pretraining concept frequency determines multimodal model performance”. In: *Proceedings of the 38th International Conference on Neural Information Processing Systems*. NIPS ’24. Vancouver, BC, Canada: Curran Associates Inc., 2024. ISBN: 9798331314385.

[24] Daya Guo, Qihao Zhu, Dejian Yang, Zhenda Xie, Kai Dong, Wentao Zhang, Guanting Chen, Xiao Bi, Yu Wu, Y. K. Li, Fuli Luo, Yingfei Xiong, and Wenfeng Liang. “DeepSeek-Coder: When the Large Language Model Meets Programming - The Rise of Code Intelligence”. In: *ArXiv* abs/2401.14196 (2024). arXiv: 2401.14196.

[25] Binyuan Hui, Jian Yang, Zeyu Cui, Jiaxi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang, Bowen Yu, Keming Lu, Kai Dang, Yang Fan, Yichang Zhang, An Yang, Rui Men, Fei Huang, Bo Zheng, Yibo Miao, Shanghaoran Quan, Yunlong Feng, Xingzhang Ren, Xuancheng Ren, Jingren Zhou, and Junyang Lin. *Qwen2.5-Coder Technical Report*. 2024. arXiv: 2409.12186 [cs.CL].

[26] Raymond Li et al. “StarCoder: may the source be with you!” English (US). in: *Transactions on Machine Learning Research* 2023 (Dec. 2023). Publisher Copyright: © 2023, Transactions on Machine Learning Research. All rights reserved. ISSN: 2835-8856.

[27] Anonymous. “ShinkaEvolve: Towards Open-Ended and Sample-Efficient Program Evolution”. In: *Submitted to The Fourteenth International Conference on Learning Representations*. under review. 2025.

[28] Alexander Novikov, Ngân V~u, Marvin Eisenberger, Emilien Dupont, Po-Sen Huang, Adam Zsolt Wagner, Sergey Shirobokov, Borislav M. Kozlovskii, Francisco J. R. Ruiz, Abbas Mehrabian, M. Pawan Kumar, Abigail See, Swarat Chaudhuri, George Holland, Alex Davies, Sebastian Nowozin, Pushmeet Kohli,

Matej Balog, and Google Deepmind. “AlphaEvolve: A coding agent for scientific and algorithmic discovery”. In: *ArXiv* abs/2506.13131 (2025).

[29] Kedar Potdar, Taher Pardawala, and Chinmay Pai. “A Comparative Study of Categorical Variable Encoding Techniques for Neural Network Classifiers”. In: *International Journal of Computer Applications* 175 (Oct. 2017), pp. 7–9. DOI: 10.5120/ijca2017915495.

[30] Ekaterina Poslavskaya and Alexey Korolev. *Encoding categorical data: Is there yet anything 'hotter' than one-hot encoding?* 2023. arXiv: 2312.16930 [cs.LG].

[31] Tomas Mikolov, Kai Chen, Gregory S. Corrado, and Jeffrey Dean. “Efficient Estimation of Word Representations in Vector Space”. In: *International Conference on Learning Representations*. accepted as poster. 2013.

[32] Debjyoti Paul*, Jie Cao*, Feifei Li, and Vivek Srikumar. “Database Workload Characterization with Query Plan Encoders”. In: *Proceedings of the VLDB Endowment* 15.4 (2021), pp. 923–935.

[33] Charith Mendis, Alex Renda, Dr.Saman Amarasinghe, and Michael Carbin. “Ithemal: Accurate, Portable and Fast Basic Block Throughput Estimation using Deep Neural Networks”. In: *Proceedings of the 36th International Conference on Machine Learning*. Ed. by Kamalika Chaudhuri and Ruslan Salakhutdinov. Vol. 97. Proceedings of Machine Learning Research. PMLR, 2019, pp. 4505–4515.

[34] Kamal Berahmand, Fatemeh Daneshfar, Elaheh Sadat Salehi, Yuefeng Li, and Yue Xu. “Autoencoders and their applications in machine learning: a survey”. In: *Artificial Intelligence Review* 57 (2 Feb. 2024). ISSN: 15737462. DOI: 10.1007/s10462-023-10662-6.

[35] Xin Rong. “word2vec Parameter Learning Explained”. In: (2016). arXiv: 1411.2738 [cs.CL].

[36] Annamalai Narayanan, Mahinthan Chandramohan, Rajasekar Venkatesan, Lihui Chen, Yang Liu, and Shantanu Jaiswal. “graph2vec: Learning Distributed Representations of Graphs”. In: ACM, July 2017. DOI: 10.1145/1235.

[37] Jie Zhou, Ganqu Cui, Shengding Hu, Zhengyan Zhang, Cheng Yang, Zhiyuan Liu, Lifeng Wang, Changcheng Li, and Maosong Sun. “Graph neural networks: A review of methods and applications”. In: *AI Open* 1 (2020), pp. 57–81. ISSN: 2666-6510. DOI: https://doi.org/10.1016/j.aiopen.2021.01.001.

[38] Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Liò, and Yoshua Bengio. “Graph Attention Networks”. In: *International Conference on Learning Representations* (2018). accepted as poster.

[39] Thomas N. Kipf and Max Welling. "Semi-Supervised Classification with Graph Convolutional Networks". In: *International Conference on Learning Representations*. 2017.

[40] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. "code2vec: learning distributed representations of code". In: *Proc. ACM Program. Lang.* 3.POPL (Jan. 2019). DOI: 10.1145/3290353.

[41] Zhenyu Hou, Xiao Liu, Yukuo Cen, Yuxiao Dong, Hongxia Yang, Chunjie Wang, and Jie Tang. "GraphMAE: Self-Supervised Masked Graph Autoencoders". In: *Proceedings of the 28th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*. KDD '22. Washington DC, USA: Association for Computing Machinery, 2022, 594–604. ISBN: 9781450393850. DOI: 10.1145/3534678.3539321.

[42] Zhenyu Hou, Yufei He, Yukuo Cen, Xiao Liu, Yuxiao Dong, Evgeny Kharlamov, and Jie Tang. "GraphMAE2: A Decoding-Enhanced Masked Self-Supervised Graph Learner". In: *Proceedings of the ACM Web Conference 2023*. WWW '23. Austin, TX, USA: Association for Computing Machinery, 2023, 737–746. ISBN: 9781450394161. DOI: 10.1145/3543507.3583379.

[43] Matthew Macfarlane and Clement Bonnet. "Searching Latent Program Spaces". In: *ICML 2025 Workshop on Programmatic Representations for Agent Learning*. 2025.

[44] Francois Chollet, Mike Knoop, Greg Kamradt, Walter Reade, and Addison Howard. *ARC Prize 2025*. https://kaggle.com/competitions/arc-prize-2025. Kaggle, Accessed October 20th, 2025. 2025.

[45] Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, Sebastian Riedel, and Douwe Kiela. "Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks". In: *Advances in Neural Information Processing Systems*. Ed. by H. Larochelle, M. Ranzato, R. Hadsell, M.F. Balcan, and H. Lin. Vol. 33. Curran Associates, Inc., 2020, pp. 9459–9474.

[46] Tyler Thomas Procko and Omar Ochoa. "Graph Retrieval-Augmented Generation for Large Language Models: A Survey". In: *2024 Conference on AI, Science, Engineering, and Technology (AIxSET)*. 2024, pp. 166–169. DOI: 10.1109/AIxSET62544.2024.00030.

[47] Zach Nussbaum, John X. Morris, Brandon Duderstadt, and Andriy Mulyar. *Nomic Embed: Training a Reproducible Long Context Text Embedder*. 2024. arXiv: 2402.01613 [cs.CL].

[48] Luke Merrick, Danmei Xu, Gaurav Nuti, and Daniel Campos. *Arctic-Embed: Scalable, Efficient, and Accurate Text Embedding Models*. 2024. arXiv: 2405.05374 [cs.CL].

[49] Puxuan Yu, Luke Merrick, Gaurav Nuti, and Daniel Campos. *Arctic-Embed 2.0: Multilingual Retrieval Without Compromise*. 2024. arXiv: 2412.04506 [cs.CL].

[50] Sean Lee, Aamir Shakir, Darius Koenig, and Julius Lipp. *Open Source Strikes Bread - New Fluffy Embeddings Model*. Accessed October 10th, 2025. 2024. URL: https://www.mixedbread.ai/blog/mxbai-embed-large-v1.

[51] Fredrik Kjolstad, Shoaib Kamil, Stephen Chou, David Lugato, and Saman Amarasinghe. "The tensor algebra compiler". In: *Proc. ACM Program. Lang.* 1.OOPSLA (Oct. 2017). DOI: 10.1145/3133901.

[52] Richard E. Pattis. *Karel the Robot: A Gentle Introduction to the Art of Programming*. 2nd. USA: John Wiley & Sons, Inc., 1994. ISBN: 0471107026.

[53] *Fasthash Documentation*. https://docs.rs/fasthash/latest/fasthash/city/struct.Hasher64.html, Accessed on September 20th, 2025.

[54] Nathaniel Simard, Louis Fortier-Dubois, Dilshod Tadjibaev, Guillaume Lagrange, and Burn Framework Contributors. *Burn*. Version 0.14.0. Accessed October 12th, 2025. Aug. 2024.

[55] Radim Řehůřek and Petr Sojka. "Software Framework for Topic Modelling with Large Corpora". English. In: *Proceedings of the LREC 2010 Workshop on New Challenges for NLP Frameworks*. http://is.muni.cz/publication/884893/en. Valletta, Malta: ELRA, May 2010, pp. 45–50.

[56] Kendall Tauser. *lang-explorer*. https://github.com/fire833/lang-explorer, Accessed on November 26th, 2025. Dec. 2025.

[57] Kubernetes. *Kubernetes/Kubernetes: Production-Grade Container Scheduling and management*. https://github.com/kubernetes/kubernetes, Accessed on September 25th, 2025.

[58] Ollama. *Ollama/Ollama: Get up and running with OpenAI GPT-Oss, DeepSeek-R1, gemma 3 and other models*. https://github.com/ollama/ollama, Accessed on November 3rd, 2025.

[59] *NGINX Gateway Fabric Documentation*. https://docs.nginx.com/nginx-gateway-fabric/, Accessed on September 26th, 2025.

[60] Laurens van der Maaten and Geoffrey Hinton. "Visualizing Data using t-SNE". in: *Journal of Machine Learning Research* 9.86 (2008), pp. 2579–2605.

[61] Colin White, Mahmoud Safari, Rhea Sukthanker, Binxin Ru, Thomas Elsken, Arber Zela, Debadeepta Dey, and Frank Hutter. *Neural Architecture Search: Insights from 1000 Papers*. Jan. 2023. DOI: 10.48550/arXiv.2301.08727.

[62] Krishna Teja Chitty-Venkata, Murali Emani, Venkatram Vishwanath, and Arun K. Somani. "Neural Architecture Search Benchmarks: Insights and Survey". In: *IEEE Access* 11 (2023), pp. 25217–25236. DOI: 10.1109/ACCESS.2023.3253818.

[63] Yizeng Han, Gao Huang, Shiji Song, Le Yang, Honghui Wang, and Yulin Wang. "Dynamic Neural Networks: A Survey". In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 44.11 (2022), pp. 7436–7456. DOI: 10.1109/TPAMI.2021.3117837.

[64] Hong Jin Kang, Tegawendé F. Bissyandé, and David Lo. "Assessing the Generalizability of Code2vec Token Embeddings". In: *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 2019, pp. 1–12. DOI: 10.1109/ASE.2019.00011.

Appendices

5.1 Enumeration of grammars

The following is an enumeration of the grammars that were used in experimentation to generate sample programs.

5.1.1 Taco Expression Grammar

⟨*entrypoint*⟩ ::= ⟨*elem*⟩ '=' ⟨*expr*⟩

⟨*expr*⟩ ::= ⟨*expr*⟩ '*' ⟨*expr*⟩ | ⟨*expr*⟩ '+' ⟨*expr*⟩ | ⟨*expr*⟩ '-' ⟨*expr*⟩ | ⟨*expr*⟩ '/' ⟨*expr*⟩ | ⟨*elem*⟩

⟨*elem*⟩ ::= ⟨*symbol*⟩ '(' ⟨*nt_index*⟩ ')'

⟨*idx*⟩ ::= 'a' | 'b' | 'c' | 'd' | 'e' | 'f' | 'g' | 'h' | 'i' | 'j' | 'k' | 'l' | 'm' | 'n' | 'o' | 'p' | 'q' | 'r' | 's' | 't' | 'u' | 'v' | 'w' | 'x' | 'y' | 'z'

⟨*symbol*⟩ ::= 'A' | 'B' | 'C' | 'D' | 'E' | 'F' | 'G' | 'H' | 'I' | 'J' | 'K' | 'L' | 'M' | 'N' | 'O' | 'P' | 'Q' | 'R' | 'S' | 'T' | 'U' | 'V' | 'W' | 'X' | 'Y' | 'Z'

⟨*nt_index*⟩ ::= ⟨*nt_index*⟩ ',' ⟨*nt_index*⟩ | ⟨*idx*⟩

5.1.2 Taco Schedule Grammar

⟨*entrypoint*⟩ ::= 'ϵ' | ⟨*expansion*⟩ | ⟨*rule*⟩ ',' ⟨*expansion*⟩

⟨*unroll_fact*⟩ ::= 'v' | 'w' | 'z'

⟨*assemble_strat*⟩ ::= 'Append' | 'Insert'

⟨*rule*⟩ ::= 'fuse' '(' ⟨*idx_var*⟩ ',' ⟨*idx_var*⟩ ',' ⟨*fused_idx_var*⟩ ')'
| 'split' '(' ⟨*idx_var*⟩ ',' ⟨*idx_var*⟩ ',' ⟨*split_fact*⟩ ')'
| 'reorder' '(' ⟨*idx_var*⟩ ')'
| 'divide' '(' ⟨*idx_var*⟩ ',' ⟨*idx_var*⟩ ',' ⟨*divide_fact*⟩ ')'
| 'precompute' '(' ⟨*idx_var*⟩ ',' ⟨*workspace_idx_var*⟩ ')'

| 'unroll' '(' ⟨*idx_var*⟩ ',' ⟨*unroll_fact*⟩ ')'
| 'parallelize' '(' ⟨*idx_var*⟩ ',' ⟨*parallelize_hw*⟩ ',' ⟨*parallelize_races*⟩ ')'

⟨*expansion*⟩ ::= ⟨*rule*⟩ ',' ⟨*expansion*⟩ | ⟨*rule*⟩

⟨*parallelize_hw*⟩ ::= 'CPUThread' | 'CPUVector' | 'GPUWarp' | 'NotParallel'

⟨*idx_var*⟩ ::= 'i' | 'j' | 'k' | 'x' | 'y'

⟨*split_fact*⟩ ::= '2' | '4' | '8' | '16' | '32'

⟨*parallelize_races*⟩ ::= 'IgnoreRaces'
| 'Temporary'
| 'NoRaces'
| 'ParallelReduction'
| 'Atomics'

⟨*fused_idx_var*⟩ ::= 'o' | 'p'

⟨*workspace_idx_var*⟩ ::= 'l' | 'm' | 'n'

⟨*divide_fact*⟩ ::= 't' | 'u'

5.1.3 CSS Grammar

⟨*entrypoint*⟩ ::= 'ϵ' | ⟨*block*⟩ | ⟨*block*⟩ ⟨*entrypoint*⟩

⟨*pseudo_elem*⟩ ::= 'first-letter' | 'first-line' | 'marker' | 'before' | 'after' | 'selection'

⟨*overflow_type*⟩ ::= 'visible' | 'hidden' | 'clip' | 'scroll' | 'auto' | 'initial' | 'inherit'

⟨*align_items_type*⟩ ::= 'normal' | 'stretch' | 'center' | 'flex-start' | 'flex-end' | 'start' | 'end' | 'baseline' | 'initial' | 'inherit'

⟨*align_content_type*⟩ ::= 'stretch' | 'center' | 'flex-start' | 'flex-end' | 'space-between' | 'space-around' | 'space-evenly' | 'inherit'

⟨*justify_content_type*⟩ ::= 'flex-start' | 'flex-end' | 'center' | 'space-between' | 'space-around' | 'space-evenly' | 'initial' | 'inherit'

⟨*selector*⟩ ::= ⟨*base_html_elem*⟩
| ⟨*base_html_elem*⟩ '>' ⟨*base_html_elem*⟩
| ⟨*base_html_elem*⟩ '+' ⟨*base_html_elem*⟩
| ⟨*base_html_elem*⟩ '-' ⟨*base_html_elem*⟩
| ⟨*comma_sep_html_elems*⟩
| ⟨*space_sep_html_elems*⟩

⟨*properties*⟩ ::= ’ϵ’ | ⟨*property*⟩ | ⟨*property*⟩ ⟨*property*⟩

⟨*align_self_type*⟩ ::= ’auto’ | ’stretch’ | ’center’ | ’flex-start’ | ’flex-end’ | ’baseline’ | ’initial’ | ’inherit’

⟨*size*⟩ ::= ⟨*number*⟩ ’%’ | ⟨*number*⟩ ⟨*number*⟩ ’%’ | ⟨*number*⟩ ⟨*size_suffix*⟩ | ⟨*number*⟩ ⟨*number*⟩ ⟨*size_suffix*⟩ | ⟨*number*⟩ ⟨*number*⟩ ⟨*number*⟩ ⟨*size_suffix*⟩

⟨*property*⟩ ::= ’align-content’ ’:’ ⟨*align_content_type*⟩ ’;’
| ’align-items’ ’:’ ⟨*align_items_type*⟩ ’;’
| ’align-self’ ’:’ ⟨*align_self_type*⟩ ’;’
| ’background’ ’:’ ⟨*color*⟩ ’;’
| ’border-bottom’ ’:’ ⟨*size*⟩ ’ ’ ⟨*border_type*⟩ ’ ’ ⟨*color*⟩ ’;’
| ’border-bottom-color’ ’:’ ⟨*color*⟩ ’;’
| ’border-bottom-left-radius’ ’:’ ⟨*size*⟩ ’;’
| ’border-bottom-right-radius’ ’:’ ⟨*size*⟩ ’;’
| ’border-bottom-style’ ’:’ ⟨*border_type*⟩ ’;’
| ’border-bottom-width’ ’:’ ⟨*size*⟩ ’;’
| ’border-color’ ’:’ ⟨*color*⟩ ’;’
| ’border-image-width’ ’:’ ⟨*size*⟩ ’;’
| ’border-left’ ’:’ ⟨*size*⟩ ’ ’ ⟨*border_type*⟩ ’ ’ ⟨*color*⟩ ’;’
| ’border-left-color’ ’:’ ⟨*color*⟩ ’;’
| ’border-left-style’ ’:’ ⟨*border_type*⟩ ’;’
| ’border-left-width’ ’:’ ⟨*size*⟩ ’;’
| ’border-radius’ ’:’ ’;’
| ’border-right’ ’:’ ⟨*size*⟩ ’ ’ ⟨*border_type*⟩ ’ ’ ⟨*color*⟩ ’;’
| ’border-right-color’ ’:’ ⟨*color*⟩ ’;’
| ’border-right-style’ ’:’ ⟨*border_type*⟩ ’;’
| ’border-right-width’ ’:’ ⟨*size*⟩ ’;’
| ’border-spacing’ ’:’ ’;’
| ’border-style’ ’:’ ’;’
| ’border-top’ ’:’ ⟨*size*⟩ ’ ’ ⟨*border_type*⟩ ’ ’ ⟨*color*⟩ ’;’
| ’border-top-color’ ’:’ ⟨*color*⟩ ’;’
| ’border-top-left-radius’ ’:’ ⟨*size*⟩ ’;’
| ’border-top-right-radius’ ’:’ ⟨*size*⟩ ’;’
| ’border-top-style’ ’:’ ⟨*border_type*⟩ ’;’
| ’border-top-width’ ’:’ ⟨*size*⟩ ’;’
| ’border-width’ ’:’ ⟨*size*⟩ ’;’
| ’color’ ’:’ ⟨*color*⟩ ’;’
| ’column-width’ ’:’ ⟨*size*⟩ ’;’
| ’display’ ’:’ ⟨*display_type*⟩ ’;’

| 'float' ':' ⟨*float_type*⟩ ';'
| 'font-size' ':' ⟨*size*⟩ ';'
| 'height' ':' ⟨*size*⟩ ';'
| 'justify-content' ':' ⟨*justify_content_type*⟩ ';'
| 'justify-items' ':' ⟨*justify_items_type*⟩ ';'
| 'justify-self' ':' ⟨*justify_self_type*⟩ ';'
| 'left' ':' ⟨*size*⟩ ';'
| 'line-height' ':' ⟨*size*⟩ ';'
| 'list-style' ':' ';'
| 'margin' ':' ⟨*size*⟩ ';'
| 'margin-bottom' ':' ⟨*size*⟩ ';'
| 'margin-left' ':' ⟨*size*⟩ ';'
| 'margin-right' ':' ⟨*size*⟩ ';'
| 'margin-top' ':' ⟨*size*⟩ ';'
| 'max-height' ':' ⟨*size*⟩ ';'
| 'max-width' ':' ⟨*size*⟩ ';'
| 'min-height' ':' ⟨*size*⟩ ';'
| 'min-width' ':' ⟨*size*⟩ ';'
| 'outline-color' ':' ⟨*color*⟩ ';'
| 'outline-style' ':' ⟨*border_type*⟩ ';'
| 'outline-width' ':' ⟨*size*⟩ ';'
| 'overflow' ':' ⟨*overflow_type*⟩ ';'
| 'overflow-x' ':' ⟨*overflow_type*⟩ ';'
| 'overflow-y' ':' ⟨*overflow_type*⟩ ';'
| 'padding' ':' ⟨*size*⟩ ';'
| 'padding-bottom' ':' ⟨*size*⟩ ';'
| 'padding-left' ':' ⟨*size*⟩ ';'
| 'padding-right' ':' ⟨*size*⟩ ';'
| 'padding-top' ':' ⟨*size*⟩ ';'
| 'right' ':' ⟨*size*⟩ ';'
| 'scroll-margin' ':' ⟨*size*⟩ ';'
| 'scroll-margin-block' ':' ⟨*size*⟩ ';'
| 'scroll-margin-block-end' ':' ⟨*size*⟩ ';'
| 'scroll-margin-block-start' ':' ⟨*size*⟩ ';'
| 'scroll-margin-bottom' ':' ⟨*size*⟩ ';'
| 'scroll-margin-inline' ':' ⟨*size*⟩ ';'
| 'scroll-margin-inline-end' ':' ⟨*size*⟩ ';'
| 'scroll-margin-inline-start' ':' ⟨*size*⟩ ';'
| 'scroll-margin-left' ':' ⟨*size*⟩ ';'

| 'scroll-margin-right' ':' ⟨*size*⟩ ';'
| 'scroll-margin-top' ':' ⟨*size*⟩ ';'
| 'scroll-padding' ':' ⟨*size*⟩ ';'
| 'scroll-padding-block' ':' ⟨*size*⟩ ';'
| 'scroll-padding-block-end' ':' ⟨*size*⟩ ';'
| 'scroll-padding-block-start' ':' ⟨*size*⟩ ';'
| 'scroll-padding-bottom' ':' ⟨*size*⟩ ';'
| 'scroll-padding-inline' ':' ⟨*size*⟩ ';'
| 'scroll-padding-inline-end' ':' ⟨*size*⟩ ';'
| 'scroll-padding-inline-start' ':' ⟨*size*⟩ ';'
| 'scroll-padding-left' ':' ⟨*size*⟩ ';'
| 'scroll-padding-right' ':' ⟨*size*⟩ ';'
| 'scroll-padding-top' ':' ⟨*size*⟩ ';'
| 'tab-size' ':' ⟨*number*⟩ ';'
| 'text-align' ':' ⟨*text_align_type*⟩ ';'
| 'top' ':' ⟨*size*⟩ ';'
| 'transition-delay' ':' ⟨*number*⟩ 's' ';'
| 'transition-duration' ':' ⟨*number*⟩ 's' ';'
| 'widows' ':' ⟨*number*⟩ ';'
| 'width' ':' ⟨*size*⟩ ';'
| 'word-spacing' ':' ⟨*size*⟩ ';'
| 'z-index' ':' ⟨*number*⟩ ';'

⟨*class*⟩ ::= 'buttons' | 'footers'

⟨*base_html_elem*⟩ ::= 'a' | 'abbr' | 'address' | 'area' | 'article' | 'aside' | 'audio' | 'b' | 'base' | 'bdi' | 'bdo' | 'blockquote' | 'body' | 'br' | 'button' | 'canvas' | 'caption' | 'cite' | 'code' | 'col' | 'colgroup' | 'data' | 'datalist' | 'dd' | 'del' | 'details' | 'dfn' | 'dialog' | 'div' | 'dl' | 'dt' | 'em' | 'embed' | 'fieldset' | 'figcaption' | 'figure' | 'footer' | 'form' | 'h1' | 'h2' | 'h3' | 'h4' | 'h5' | 'h6' | 'head' | 'header' | 'hgroup' | 'hr' | 'html' | 'i' | 'iframe' | 'img' | 'input' | 'ins' | 'kbd' | 'label' | 'legend' | 'li' | 'link' | 'main' | 'map' | 'mark' | 'meta' | 'meter' | 'nav' | 'noscript' | 'object' | 'ol' | 'optgroup' | 'option' | 'output' | 'p' | 'param' | 'picture' | 'pre' | 'progress' | 'q' | 'rp' | 'rt' | 'ruby' | 's' | 'samp' | 'script' | 'section' | 'select' | 'small' | 'source' | 'span' | 'strong' | 'style' | 'sub' | 'summary' | 'sup' | 'table' | 'tbody' | 'td' | 'template' | 'textarea' | 'tfoot' | 'th' | 'thead' | 'time' | 'title' | 'tr' | 'track' | 'u' | 'ul' | 'var' | 'video' | 'wbr'

⟨*comma_sep_html_elems*⟩ ::= ⟨*base_html_elem*⟩
| ⟨*base_html_elem*⟩ ',' ⟨*comma_sep_html_elems*⟩

⟨*float_type*⟩ ::= 'none' | 'left' | 'right' | 'initial' | 'inherit'

⟨*justify_self_type*⟩ ::= 'auto' | 'normal' | 'stretch' | 'start' | 'left' | 'center' | 'end' | 'right' | 'initial' | 'inherit'

⟨*pseudo_class*⟩ ::= 'link' | 'visited' | 'hover' | 'active'

⟨*size_suffix*⟩ ::= 'c' 'm' | 'm' 'm' | 'i' 'n' | 'p' 'x' | 'p' 't' | 'p' 'c' | 'e' 'm' | 'e' 'x' | 'c' 'h' | 'r' 'e' 'm' | 'v' 'w' | 'v' 'h' | 'v' 'm' 'i' 'n' | 'v' 'm' 'a' 'x'

⟨*text_align_type*⟩ ::= 'left' | 'right' | 'justify' | 'initial' | 'inherit'

⟨*justify_items_type*⟩ ::= 'normal' | 'stretch' | 'start' | 'left' | 'center' | 'end' | 'right' | 'initial' | 'inherit'

⟨*html_elem*⟩ ::= ⟨*base_html_elem*⟩ ':' ':' ⟨*pseudo_elem*⟩ | ⟨*base_html_elem*⟩ ':' ':' ⟨*pseudo_elem*⟩ ':' ⟨*pseudo_class*⟩ | ⟨*base_html_elem*⟩ ':' ⟨*pseudo_class*⟩ | ⟨*base_html_elem*⟩

⟨*color*⟩ ::= '#842d5b' | '#20b01c' | '#7d1dc1' | '#42a1dc' | '#da8454' | '#8ec5d2' | '#a69657' | '#a69657' | '#664ba3' | '#3b6a42' | 'rgb(39, 37, 193)' | 'rgb(37, 138, 166)' | 'rgb(84, 183, 126)' | 'rgb(104, 36, 170)' | 'rgb(207, 106, 144)' | 'rgb(203, 135, 198)' | 'rgb(231, 100, 187)' | 'rgb(143, 143, 143)' | 'rgb(68, 68, 68)' | 'rgb(160, 160, 160)' | 'rgb(141, 141, 141)' | 'rgb(95, 95, 95)'

⟨*block*⟩ ::= ⟨*selector*⟩ '' ⟨*properties*⟩ ''

⟨*display_type*⟩ ::= 'inline' | 'block' | 'contents' | 'flex' | 'grid' | 'inline-block' | 'inline-flex' | 'inline-grid' | 'inline-table' | 'list-item' | 'run-in' | 'table' | 'table-caption' | 'table-column-group' | 'table-header-group' | 'table-footer-group' | 'table-row-group' | 'table-cell' | 'table-column' | 'table-row' | 'none' | 'initial' | 'inherit'

⟨*id*⟩ ::= 'content' | 'header' | 'btn'

⟨*border_type*⟩ ::= 'none' | 'hidden' | 'dotted' | 'dashed' | 'solid' | 'double' | 'groove' | 'ridge' | 'inset' | 'outset' | 'initial' | 'inherit'

⟨*space_sep_html_elems*⟩ ::= ⟨*base_html_elem*⟩
| ⟨*base_html_elem*⟩ ' ' ⟨*space_sep_html_elems*⟩

⟨*number*⟩ ::= '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'

5.1.4 Karel Grammar

⟨*program*⟩ ::= 'def run(): ' ⟨*stmt*⟩

⟨*count*⟩ ::= '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9' | '10' | '11' | '12' | '13' | '14' | '15' | '16' | '17' | '18' | '19'

⟨*action*⟩ ::= 'move()' | 'turnRight()' | 'turnLeft()' | 'pickMarker()' | 'putMarker()'

⟨*cond*⟩ ::= 'frontIsClear()' | 'leftIsClear()' | 'rightIsClear()' | 'markersPresent()' | 'noMarkersPresent()' | 'not' ' ' ⟨*cond*⟩

⟨*stmt*⟩ ::= 'while' '(' ⟨*cond*⟩ ')' ':' ' ' ⟨*stmt*⟩ | 'repeat' '(' ⟨*count*⟩ ')' ':' ' ' ⟨*stmt*⟩ | ⟨*action*⟩ | ⟨*stmt*⟩ ';' ⟨*stmt*⟩ | 'if' '(' ⟨*cond*⟩ ')' ':' ' ' ⟨*stmt*⟩ | 'ifelse' '(' ⟨*cond*⟩ ')' ':' ' ' ⟨*stmt*⟩ 'else' ':' ' ' ⟨*stmt*⟩

5.2 Additional Nearest Neighbor Diagrams

The following is an enumeration of the additional nearest neighbor plots that I created that were not included in the experiments section. This was largely due to their sheer size and difficulty to reasonably parse the images without zooming in substantially.

5.2.1 Additional TACO Schedule Plots

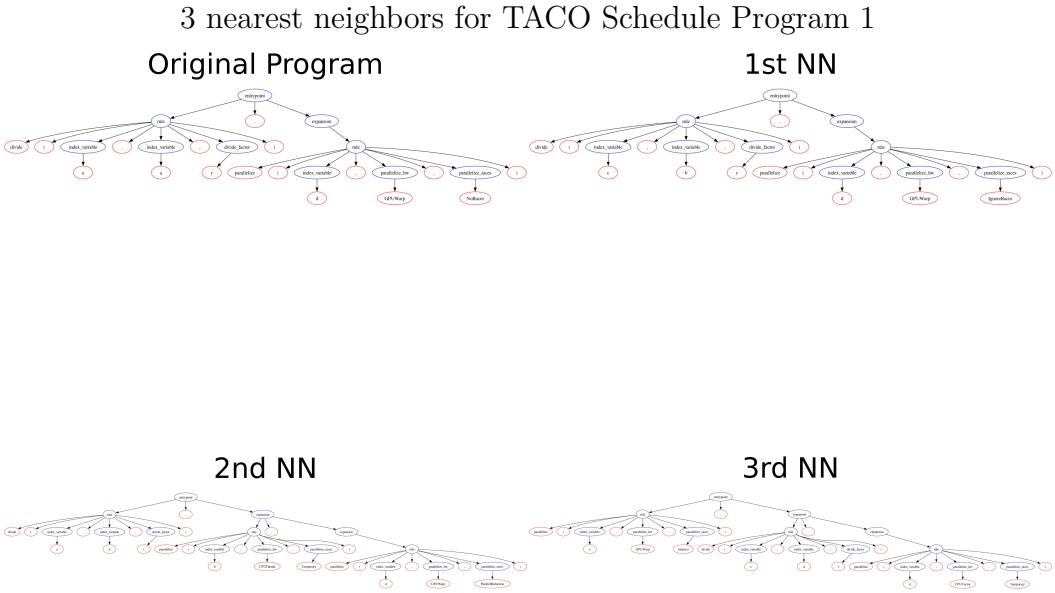


Figure 5.1: Nearest neighbors for a fairly large TACO schedule expression graph.

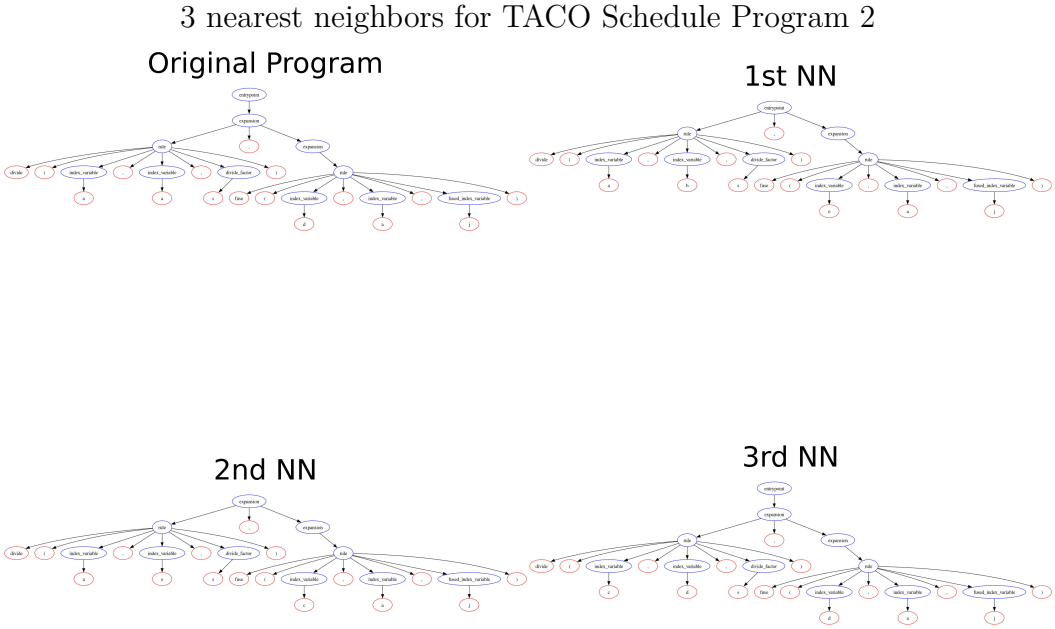


Figure 5.2: Another nearest neighbors plot for another fairly large TACO schedule AST.

5.2.2 Additional TACO Expression Plots

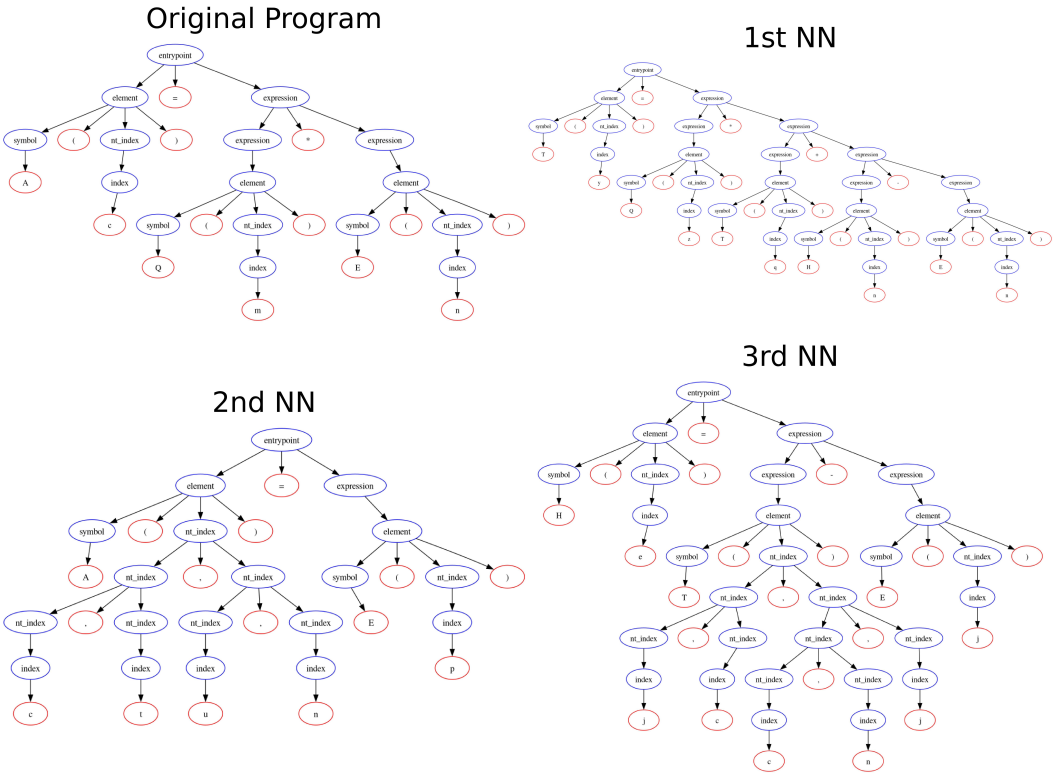


Figure 5.3: Another fairly large TACO expression nearest neighbor plot.

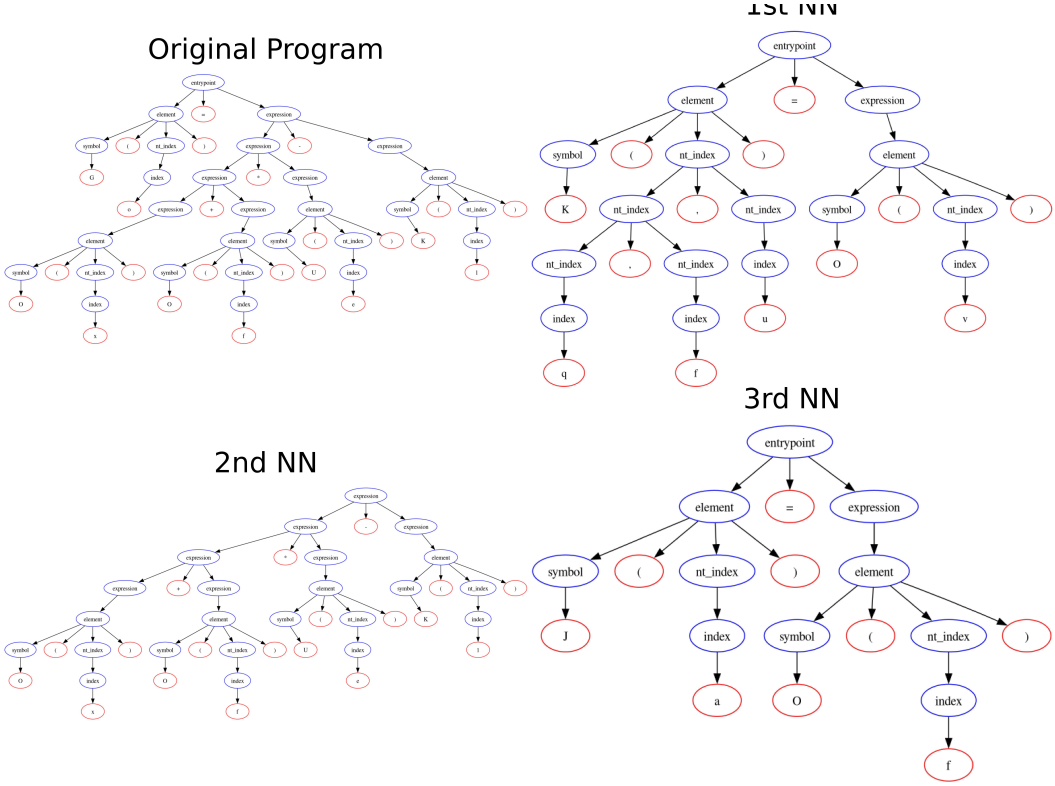


Figure 5.4: Final sampled nearest neighbors plot for TACO expressions. This is interesting sample where the last tensor has different symbols in each neighbor, but the overall structure of being a tensor of rank 1 is the same.

5.2.3 Additional CSS Plots

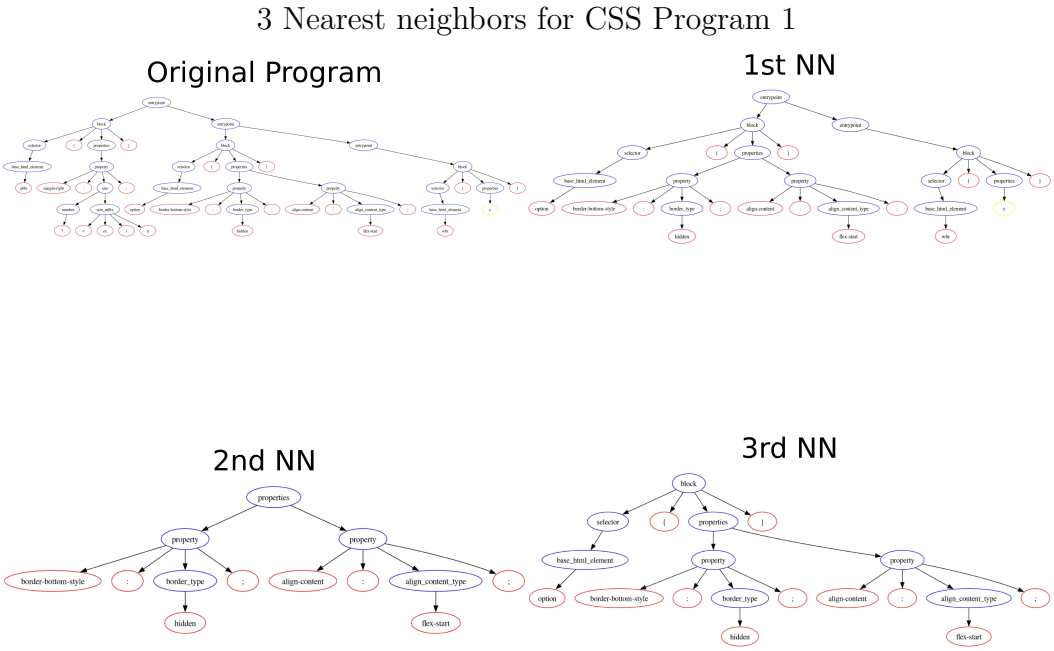


Figure 5.5: Another interesting set of nearest neighbor plots for a CSS program. The last block within both the original and first neighbor is of the form **wbr {}** (i.e. a selector with no properties). The 2nd and 3rd neighbors are subsets of the first selector block within the original program.

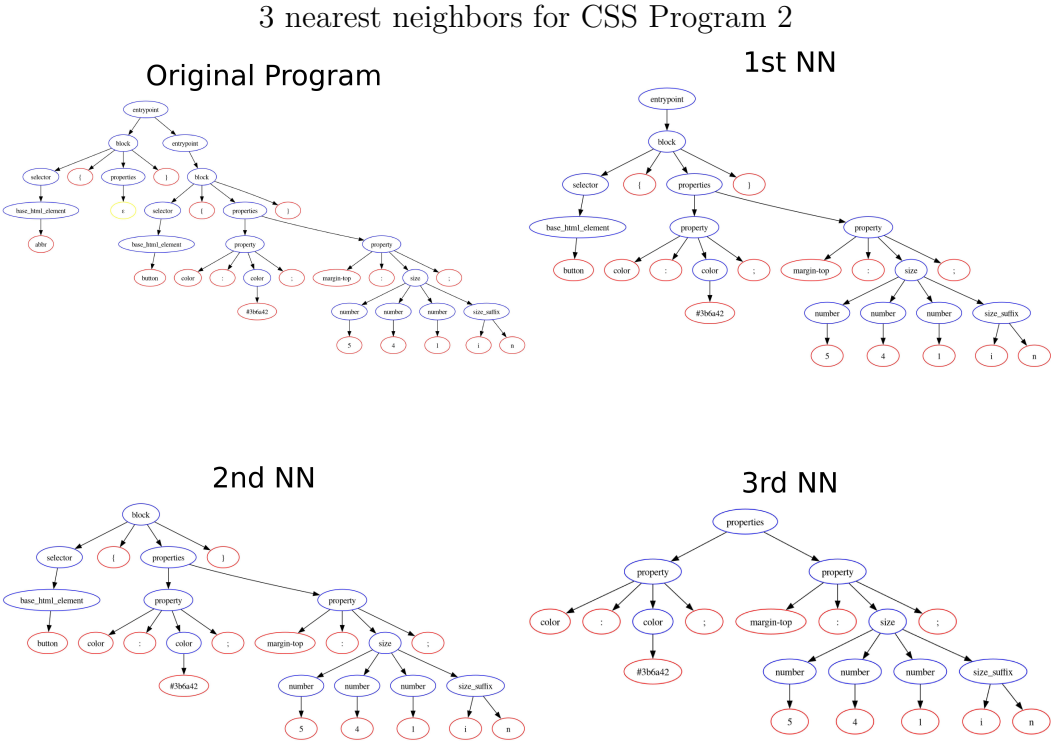


Figure 5.6: Final CSS nearest neighbor plot. This example is a perfect example of the property described in 4.20. Each nearest neighbor is a smaller and smaller subset of the right-hand side of the original program tree.

5.2.4 Additional Karel Plots

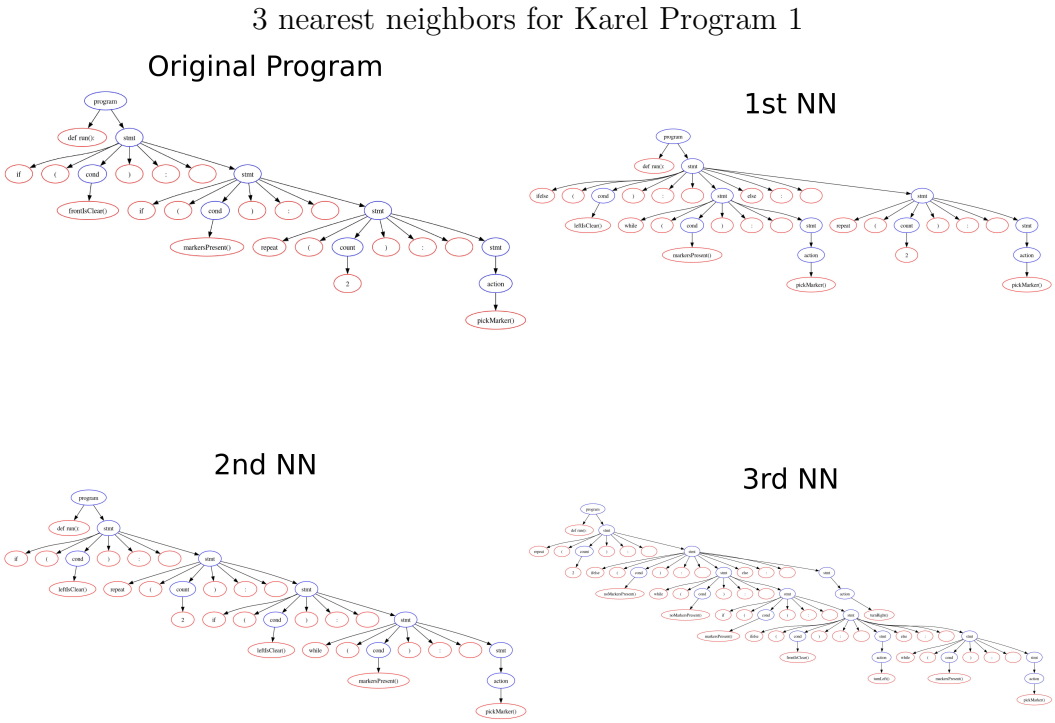


Figure 5.7: A larger Karel nearest neighbors example. The most important thing to note here is the strong structural similarity along the bottom right side of the original program and it's neighbors.

5.3 Additional Similarity Distribution Plots

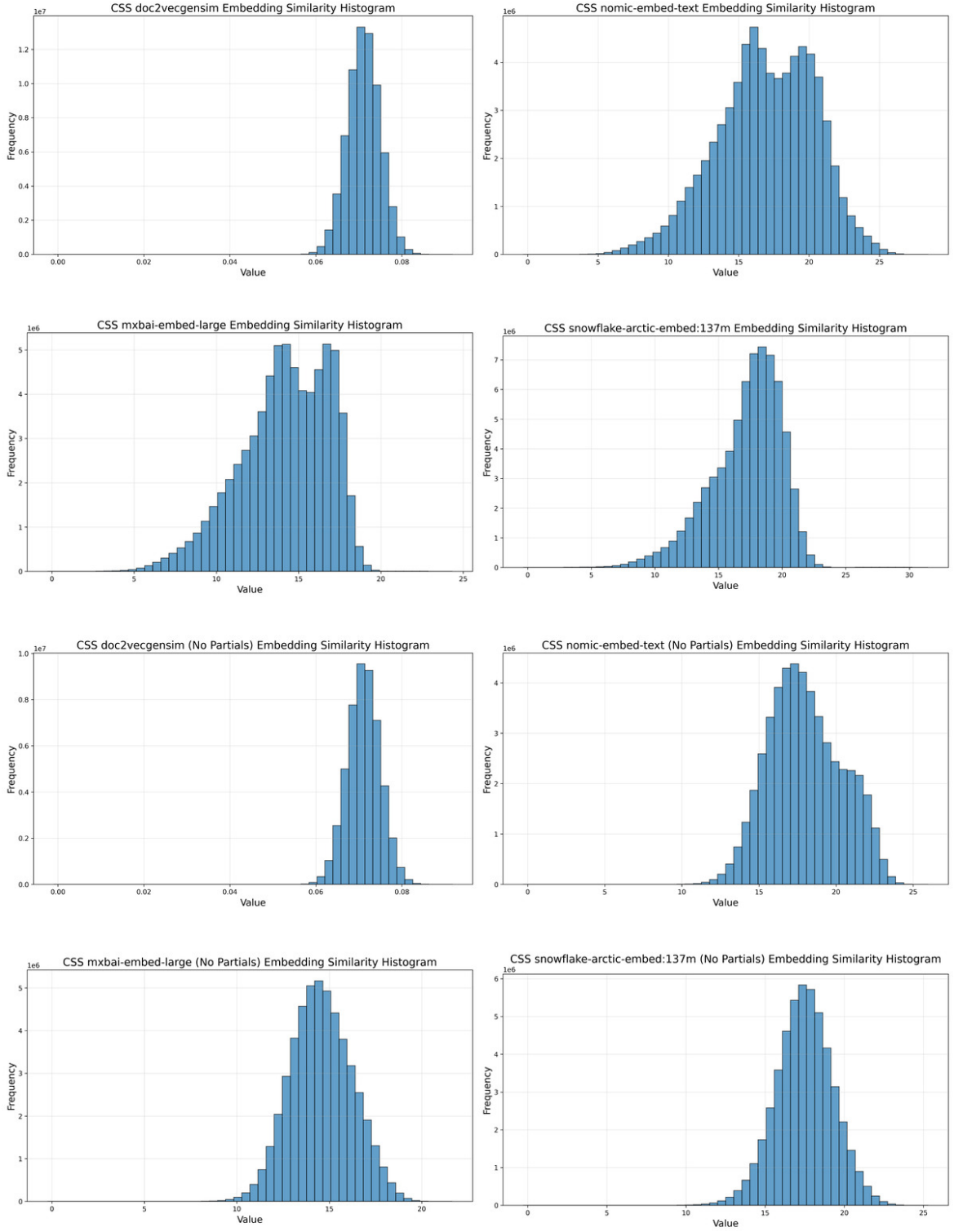


Figure 5.8: Histograms of pairwise similarity distances for programs within a synthetic CSS dataset. The top four plots are for a dataset with partials, and the bottom four plots are for a dataset without.

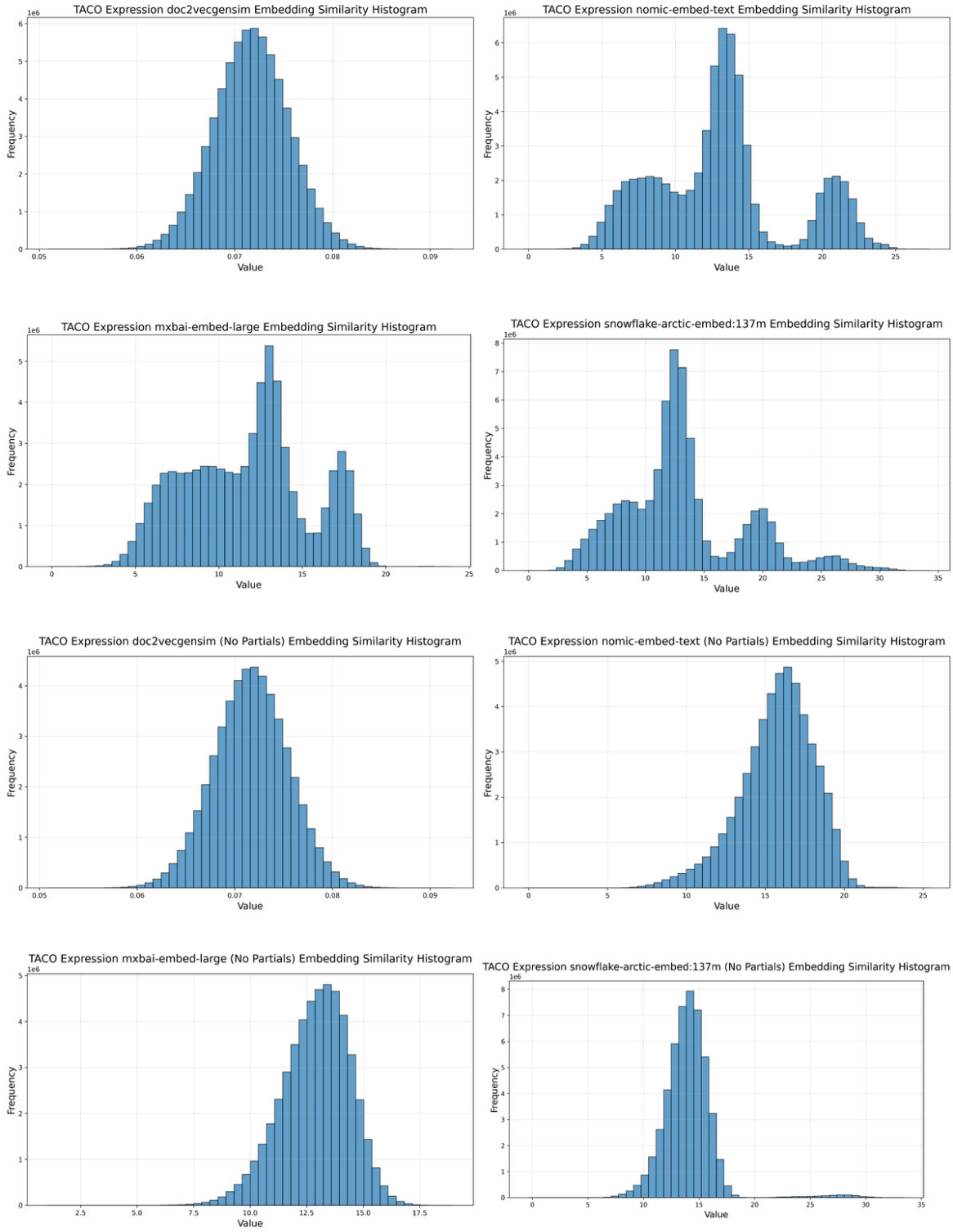


Figure 5.9: Histograms of pairwise similarity distances within a synthetic TACO expression dataset. Similar to 5.8, top plots are for partial programs, bottom four are plots for datasets with no partials.

5.4 Additional Correlation Analysis plots

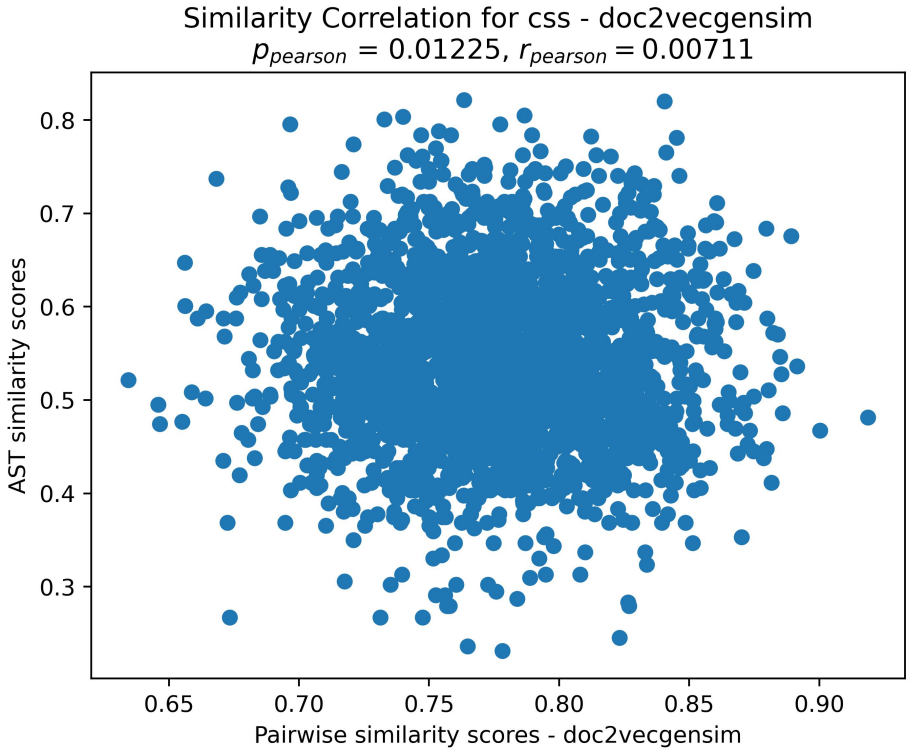


Figure 5.10: Similarity Correlation plot for CSS programs from doc2vec. Pearson p and r-values are shown the the top of the figure.

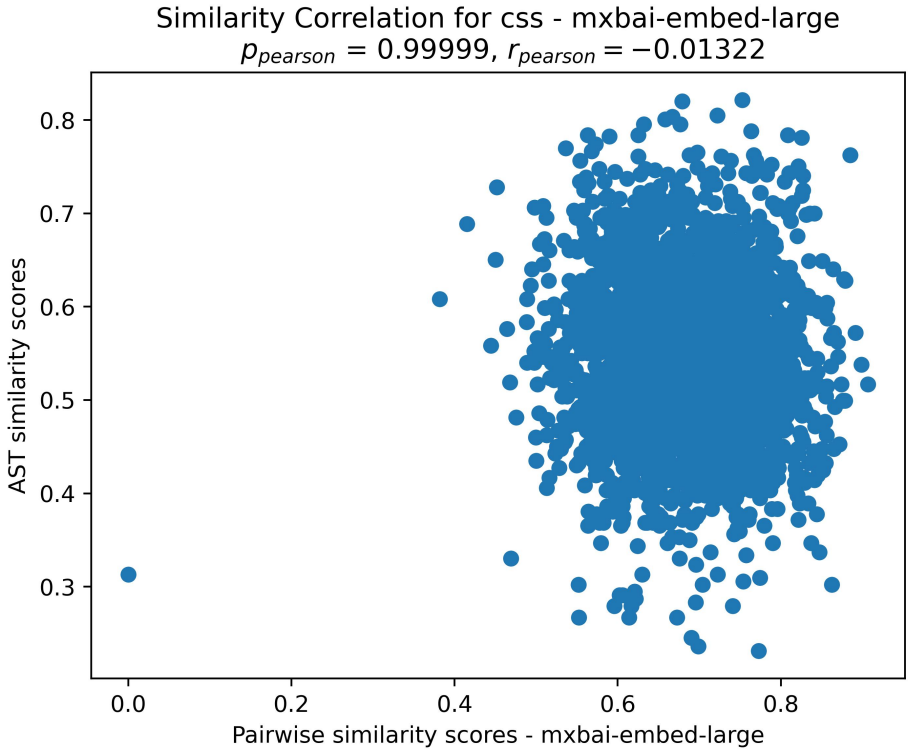


Figure 5.11: Similarity Correlation plot for CSS programs from mxbai-embed-large. Pearson p and r-values are shown the the top of the figure.

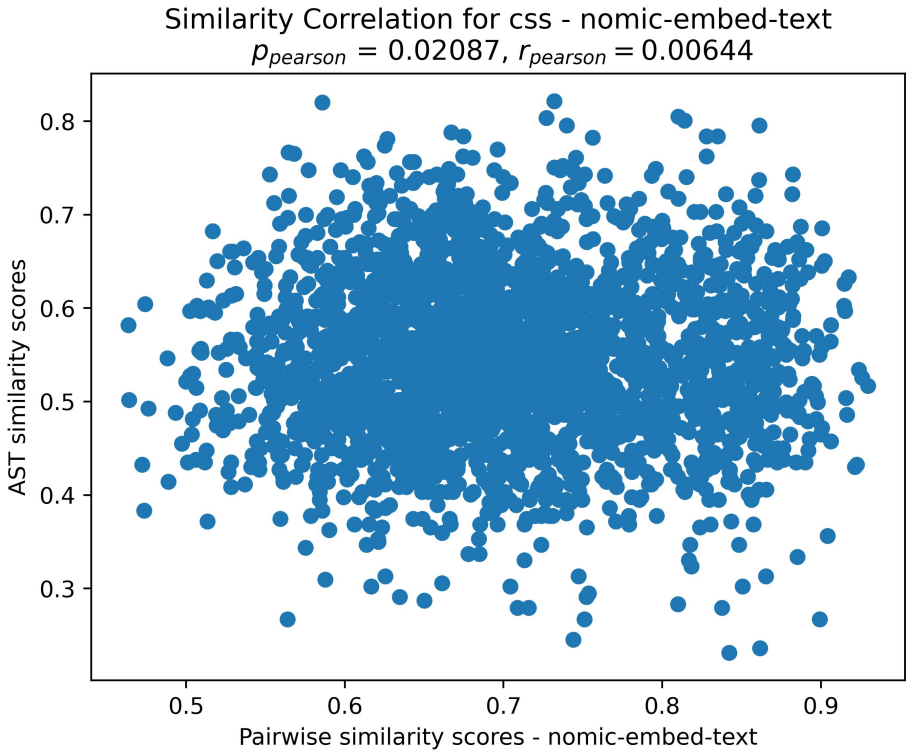


Figure 5.12: Similarity Correlation plot for CSS programs from Nomic's text embedding model. Pearson p and r-values are shown the the top of the figure.

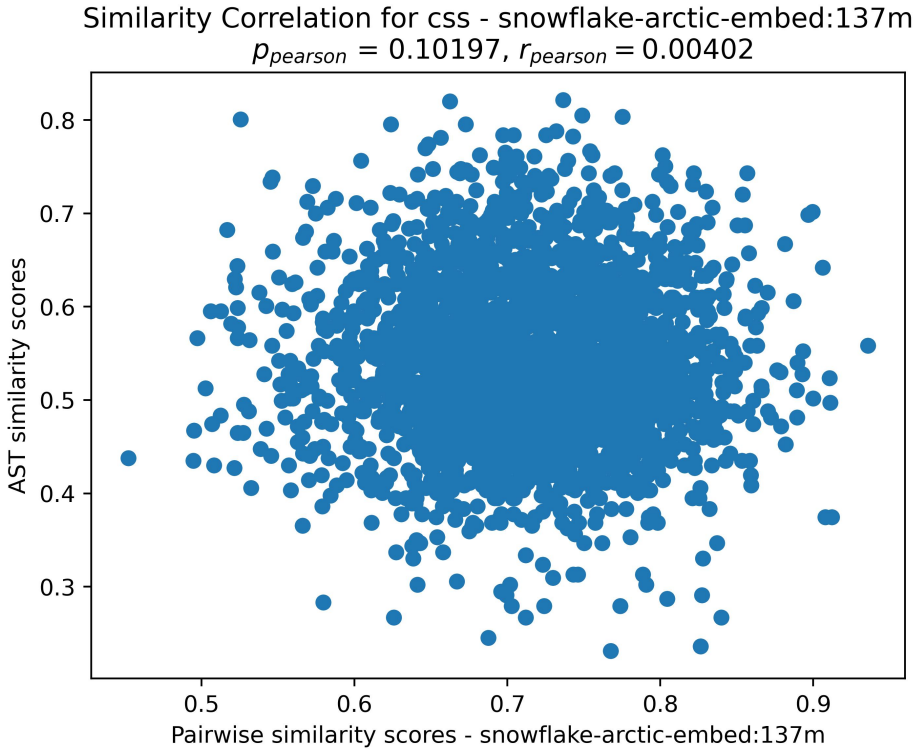


Figure 5.13: Similarity Correlation plot for CSS programs from Snowflake's snowflake-arctic-embed:137m. Pearson p and r-values are shown the the top of the figure.

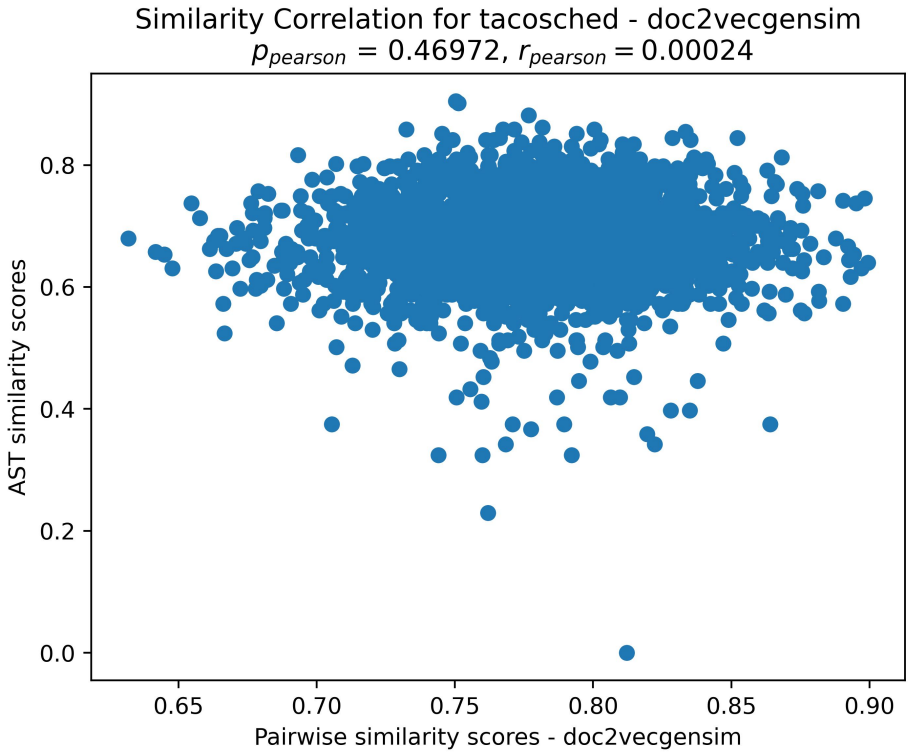


Figure 5.14: Similarity Correlation plot for TACO Schedule programs from doc2vec. Pearson p and r-values are shown the the top of the figure.

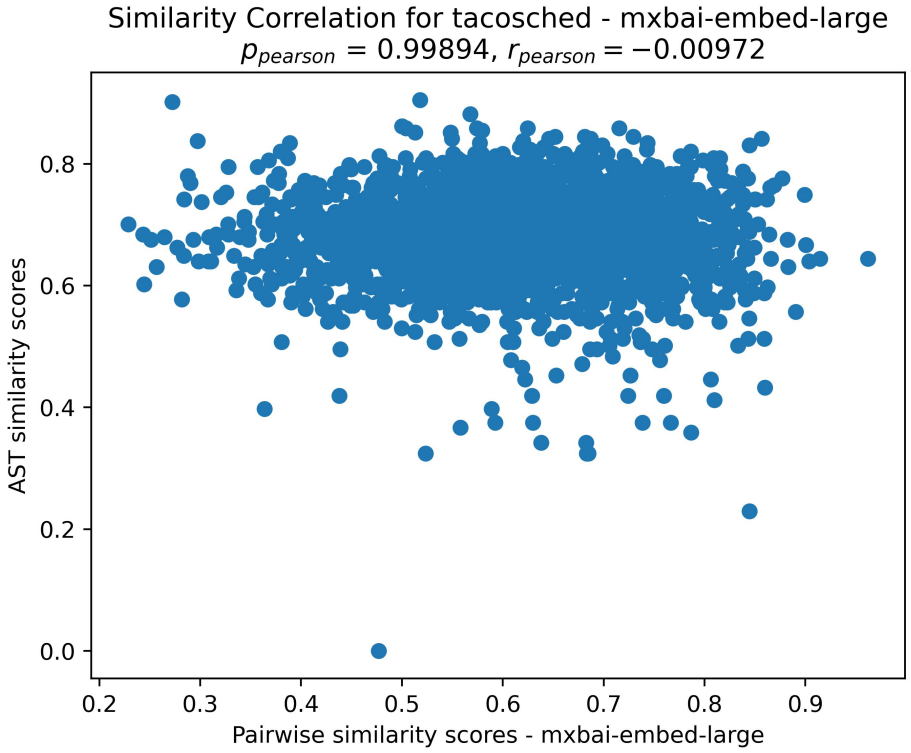


Figure 5.15: Similarity Correlation plot for TACO Schedule programs from mxbai-embed-large. Pearson p and r-values are shown the the top of the figure.

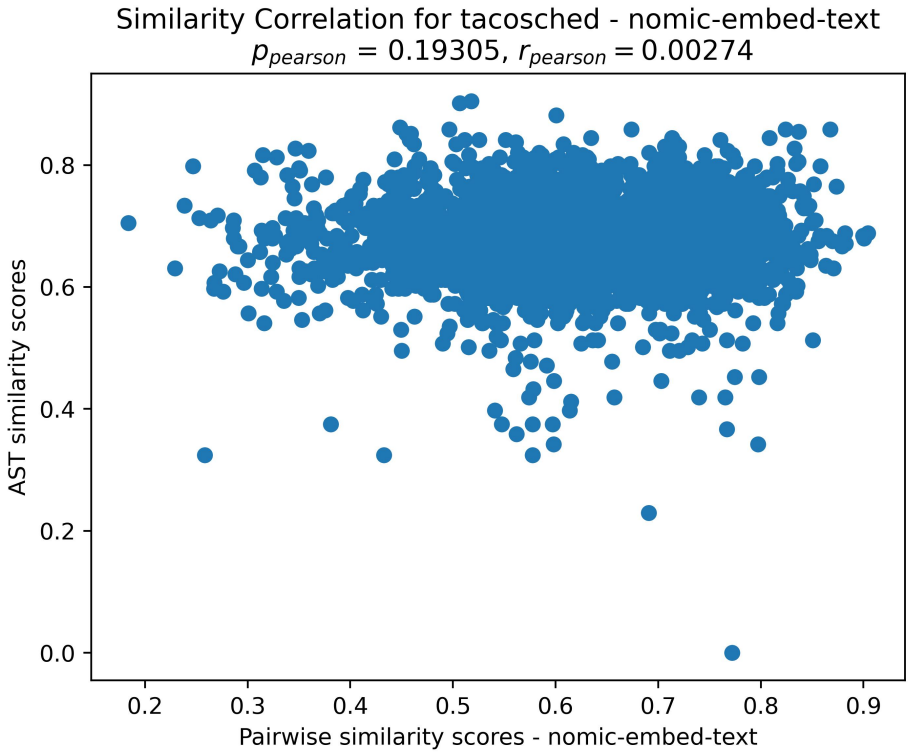


Figure 5.16: Similarity Correlation plot for TACO Schedule programs from Nomic's text embedding model. Pearson p and r-values are shown the the top of the figure.

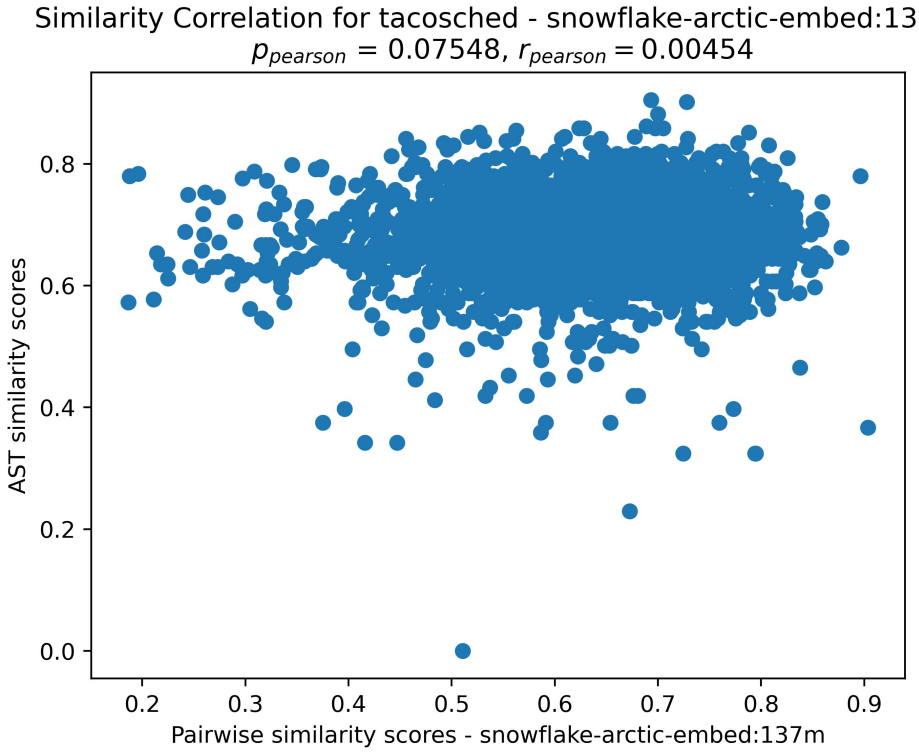


Figure 5.17: Similarity Correlation plot for TACO Schedule programs from Snowflake's snowflake-arctic-embed:137m. Pearson p and r-values are shown the the top of the figure.

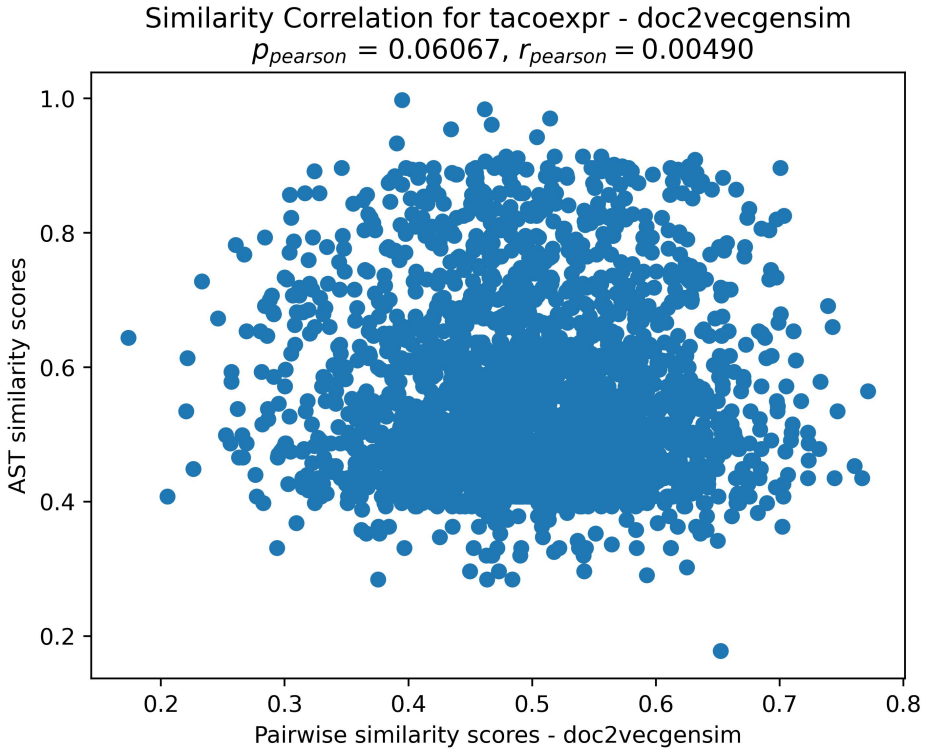


Figure 5.18: Similarity Correlation plot for TACO Expression programs from doc2vec. Pearson p and r-values are shown the the top of the figure.

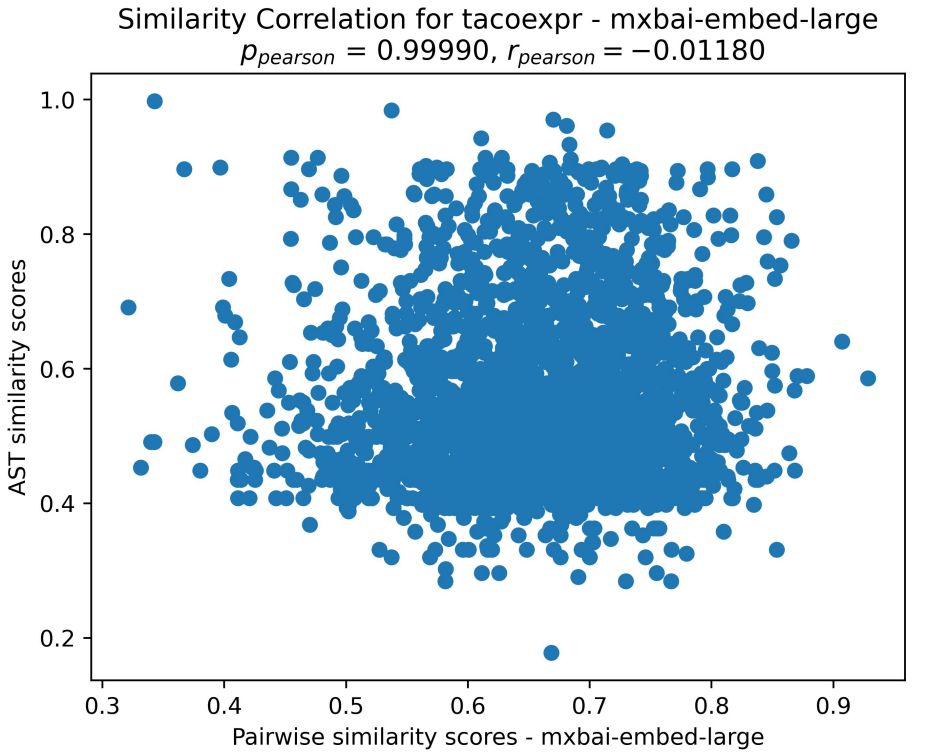


Figure 5.19: Similarity Correlation plot for TACO Expression programs from mxbai-embed-large. Pearson p and r-values are shown the the top of the figure.

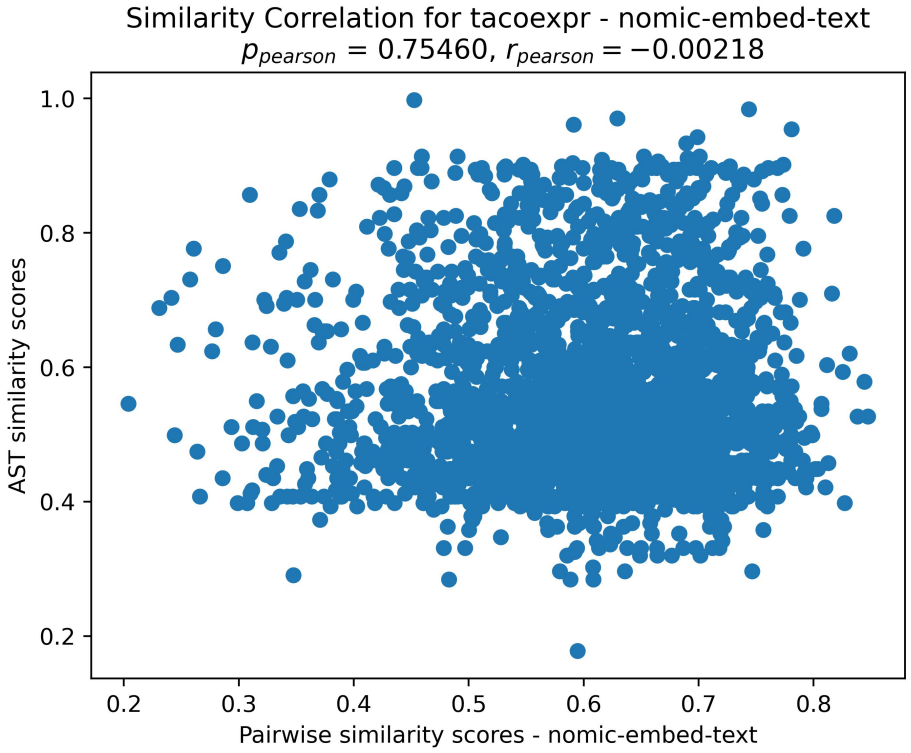


Figure 5.20: Similarity Correlation plot for TACO Expression programs from Nomic's text embedding model. Pearson p and r-values are shown the the top of the figure.

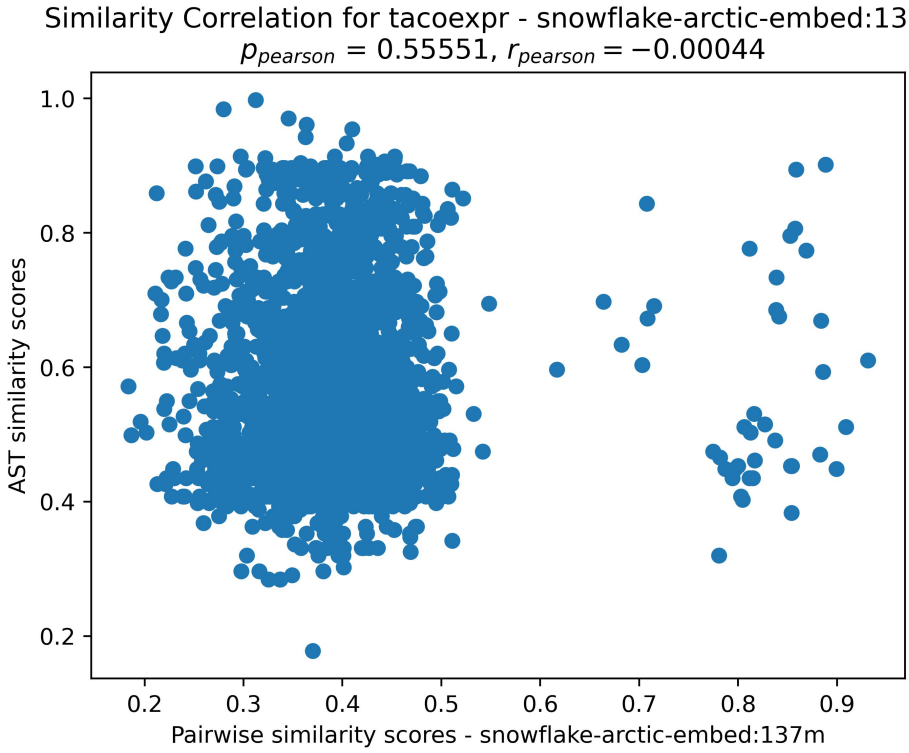


Figure 5.21: Similarity Correlation plot for TACO Expression programs from Snowflake's snowflake-arctic-embed:137m. Pearson p and r-values are shown the the top of the figure.