Code Generation From Flowcharts with Texts: A Benchmark Dataset and An Approach

Zejie Liu[1], Xiaoyu Hu[1], Deyu Zhou[1]*, Lin Li[2], Xu Zhang[1] and **Yanzheng Xiang[1]**
[1]Key Laboratory of Computer Network and Information Integration,
Ministry of Education, Southeast University, China
[2]Nanjing Origin Information Technology Company, China
{liuzejie,xiaoyuhu,d.zhou,xuzhang123,yz_xiang}@seu.edu.cn
jerax.ga@gmail.com

Abstract

Currently, researchers focus on generating codes from the requirement documents. However, current approaches still perform poorly on some requirements needing complex problem-solving skills. In reality, to tackle such complex requirements, instead of directly translating requirement documents into codes, software engineers write codes via unified modeling language diagrams, such as flowcharts, an intermediate tool to analyze and visualize the system. Therefore, we propose a new source code generation task, that is, to generate source code from flowcharts with texts. We manually construct a benchmark dataset containing 320 flowcharts with their corresponding source codes[1]. Obviously, it is not straightforward to employ the current approaches for the new source code generation task since (1) the flowchart is a graph that contains various structures, including loop, selection, and others which is different from texts; (2) the connections between nodes in the flowchart are abundant and diverse which need to be carefully handled. To solve the above problems, we propose a two-stage code generation model. In the first stage, a structure recognition model is employed to transform the flowchart into pseudo-code containing the structural conventions of a typical programming language such as *while*, *if*. In the second stage, a code generation model is employed to convert the pseudo-code into code. Experimental results show that the proposed approach can achieve some improvement over the baselines.

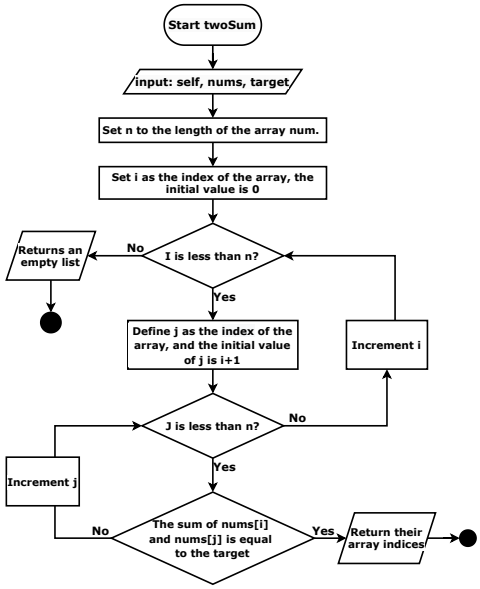
1 Introduction

Recently, automated source code generation from requirements documents has been a fashionable task that maps specific descriptions to various executable codes in software engineering and artificial intelligence. Developments in deep learning have facilitated the effectiveness of transformations between natural language and source code. However, generating code that solves a specified task requires searching in the huge structured space of

Requirement document:

Given an array of integers nums and an integer target, return indices of the two numbers such that they add up to target.

Flowchart:



Code:

```
def twoSum(nums, target):
    n = len(nums)
    for i in range(n):
        for j in range(i + 1, n):
            if nums[i]+nums[j] == \
                    target:
                return [i, j]
    return []
```

Table 1: An example of the flowchart and its corresponding code. The input to the model is the flowchart, and the output is the code.

*Corresponding author.

[1]https://github.com/LiuZeJie97/Code-Generation-From-Flowcharts-with-Texts-A-Benchmark-Dataset-and-An-Approach

Findings of the Association for Computational Linguistics: EMNLP 2022, pages 6098 - 6106

possible programs, with a very sparse reward signal, and the solutions can look dramatically different even for the same problem (Li et al., 2022). Therefore, most prior work has been limited to generating short code snippets from sentences (Oda et al., 2015; Yin et al., 2018). However, these works' sentences are more straightforward than the requirement documents in real scenarios. For example, to generate the code in Table 1, the model needs to understand the task requirements and accurately select traversal algorithm from many candidates (e.g. dynamic programming, greedy, divide and conquer).

In practice, to solve complex code generation tasks, rather than translating requirements documents directly into code, software engineers write code through Unified Modelling Language(UML), which intends to provide a standard way of visualizing the design of a system (Hutchinson et al., 2014). For all UML, flowchart plays an important role in the analysis of system requirements, preliminary design, and detailed design (Sendall and Kozaczynski, 2003). Before writing code, drawing a flowchart to illustrate the algorithm to be used and the steps to follow can significantly reduce task's difficulty. There is evidence that, in industrial practice, flowcharts have been widely used for problem understanding (i.e. analysis) and documentation. Thus, we propose a new task to generating executable code from a flowchart with text.

There have also been several studies on converting flowcharts to code. Wu et al. and Wang et al. proposed rule-based methods that can identify the loop and selection semantics in the flowchart and automatically convert it into pseudo-code which use structural conventions of a normal programming language such as *while*, *if*, but the details in the pseudo-code are still in text, which makes these codes unable to execute directly on the computer. In contrast, we tackle the task of executable code(python) generation from flowcharts with text. Most closely related to our task is Oda et al. (2015), they propose a task to generate code from pseudo-code. However, each line of pseudo-code in their dataset's is independent of the other, models can generate code snippets for each line regardless of other lines. However, in our dataset, the connections between nodes in the flowchart are abundant and diverse which need to be carefully handled. Thus, although flowcharts and pseudo-codes can be converted into each other, our dataset is different from theirs.

We constructed a benchmark dataset, FC2Code, which contains 320 flowcharts with natural language and code pairs. We obtained the code from the programming competition platform LeetCode and manually drew the flowcharts with natural language. Previous models cannot be used directly for the new source code generation task for the following reasons. Firstly, a flowchart is a graph containing various structures, including loops and selections, which is different from the text. Secondly, in a natural scene, each node of the flowchart is not independent. As seen from Table 3, 62% of nodes in FC2Code are related with other nodes, which means that the model cannot generate code snippets for each node without considering its neighbors.

To solve these problems above, we propose a two-stage code generation model. In the first stage, the Structure Recognition Model transforms the flowchart into a pseudo-code that containing some structure information such as *while*, *if*. In the second stage, a Code Generation Mode is employed to merge the information in the node of the flowchart and convert the pseudo-code into executable code. The experiments show that it is necessary to enhance each node's representation with its neighbors according to the structure of the flowchart.

2 Related Work

2.1 Generating Source Code from Requirements Document

Automatic generation of source code from requirements documents has recently been a hot topic in artificial intelligence communities, such as mapping natural language directly to executable programs in the form of logical forms (Zelle and Mooney, 1996), database queries (Yu et al., 2018, 2019), general purpose code snippets (Oda et al., 2015; Yin et al., 2018), complete executable code that can solve a specific task (Liu et al., 2020; Li et al., 2022). Ling et al. generated Java and Python source code from Natural Language (NL) with card attributes for games. Agashe et al. presented JuICe, a large-scale NL-code dataset based on jupyter notebooks. The notebooks comprise interleaved NL markdown and code cells, and the model needs to generate the target code cell based on the previous markdown. Iyer et al. introduced the task of generating class member functions given English documentation and the programmatic context, such as Variables and Methods, provided by the rest of

the class. Different from the above works, this work introduces a new task of generating programs from flowcharts with texts.

2.2 Generating Pseudo-code from Flowchart

There have been some rule-based methods for automatically generating pseudo-code from flowcharts. Carlisle et al. proposed a modeling and simulation system, RAPTOR, which can automatically generate code based on selection and loop primitives offered by users. Wu et al. introduced a structure identification algorithm that automatically identifies loops and selections according to the flowchart's structure. Wang et al. proposed a method that can generate code for semi-structured flowchart which contains semi-structured semantics such as *break* and *return*. However, all these works only focus on identifying the structure in the flowchart and generating pseudo-code with natural language instead of executable source code.

3 Dataset Construction

We created a new dataset called FC2Code (FlowChart to Code), which consists of code and flowchart pairs. As can be seen in Figure 1. The basic process of dataset construction can be divided into 3 steps: 1) Code Extraction. 2) Flowchart Sketch Generation. 3) Nature Language Annotation. The following sections describe the above three steps in detail.

Code Extraction. Firstly, we extract 320 codes from LeetCode[2], an online programming competition platform. The code crawled in public available code repositories such as Github often has project-related operations or global variables and the logic of some source code is often not clear enough, and even has bugs. Thus, we extract the Python codes from an online programming competition platform, LeetCode. LeetCode provides open-domain tasks with high-quality official solutions, each solution contains well-tested code with a detailed explanation. Because each function in the code will be converted into a flowchart, we filter out the code containing multiple functions to make sure each flowchart contains complete information that can generate executable code. We select 253 problems with 320 codes from LeetCode.

Flowchart Sketch Generation. Secondly, we use *pyflowchart*[3] to automatically convert the code into flowchart sketches. Specifically speaking, each line of code is transformed into a node in the flowchart, then the tool automatically connects edges to each node according to their execution order in the code. Note that the content of the nodes in the flowchart sketches is still code snippets.

Nature Language Annotation. Thirdly, we translate each code in the node into natural languages in Chinese. We set the following principle for annotators:

- When the programmer sees the flowchart, he/she can write the code with the same functionality as the source code.
- We also encourage annotators to describe the same code in different ways. For example, "i++" will be annotated as "Increment i" or "Shift the subscript of an array to the right by one unit."
- In real scenarios, the nodes in the flowchart are usually related to each other, for example, some nodes may incorporate the variable declared or described above. Therefore, we increased the description's abstraction and added some relationships between nodes. For example, when a new variable is defined, the function and usage of the variable will also be annotated, and in the following annotation process, we will no longer directly mention the name of the variable but refer to its function. Table 3 shows situations when one node is connected to another in our dataset.

Annotating nodes in flowcharts is a time-consuming process. Labeling one flowchart takes us about 30 minutes of labor on average. And annotating 320 flowcharts costs us around 160 hours of human labor. Table 2 presents statistics for our dataset.

To verify the quality of the annotation, we let annotators exchange their samples and infer the code based on the annotation results. Finally, we sampled 50 flowcharts with 771 nodes, and find that 82.00% flowcharts and 98.44% nodes from our dataset are solvable.

4 Data Statistics

4.1 Relation Between Nodes

The relations between nodes in the flowchart are abundant and diverse. Our analysis of 50 flowchart

[2]https://leetcode.com/problemset/all/
[3]https://github.com/cdfmlr/pyflowchart/

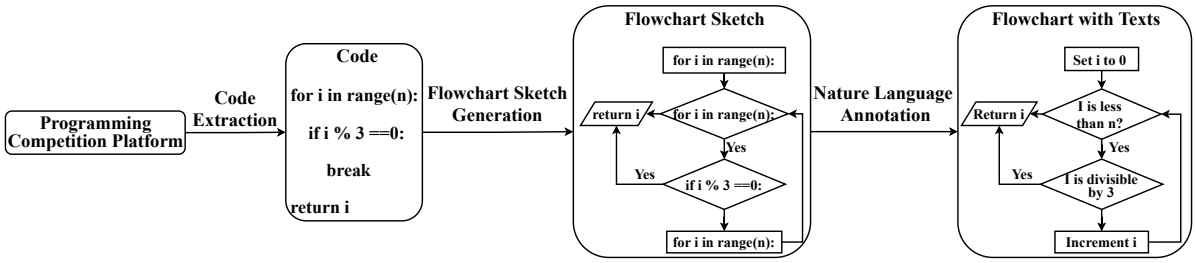


Figure 1: The basic process of dataset construction

	Train	Valid	Test
Flowcharts	220	50	50
Avg Tokens	224.9	264.1	226.9
Avg Nodes	16.9	19.2	18.4
Codes	220	50	50
Avg Tokens	110.8	131.2	118.9
Avg Lines	12.5	14.3	14.5

Table 2: Statistics of FC2Code dataset.

examples reveals the relation types and their proportion in FC2Code. Overall, 62% of nodes are related to another node, and 98% flowcharts contain the nodes that are related to another node. As shown in Table 3, Most nodes (46%) require using the variable declared or described in the previous node. 7% require understanding the functions of the variable. For example, in row 2 of Table 3, given that `dp[i]` represents the i-th odd number, "The first odd number" in the current node should be converted to "`dp[1]`". 4% nodes require understanding the properties of the variable and its data structure. For example, in row 3 of Table 3, the data type of "builder" is an array and the NL in current node is "Return a valid string", thus, we need to convert "builder" to a string before returning it. 9% nodes need to infer what the demonstrative pronoun refers to. In row 4 of Table 3, the NL of current node using a word "their", which requires the model to find what "their" refers to. Lastly, 17% nodes are related to at least 2 other nodes.

4.2 Distance of Two Interrelated Nodes

In the randomly sampled 50 flowcharts, the distance distribution of two nodes with a relationship is shown in Figure 2. The number of nodes decreases as the distance of relations increases. The reason account for it is that local variables are often used repeatedly in code snippets. There are many nodes with distances between 1 and 2. Maybe it's because the code in FC2Code contains a lot of *for* keywords. As shown in Table 1: "I is less than n", "Set i as the index of the array, the initial value is 0" and "Increment i" are related and their distances are between 1 and 2.

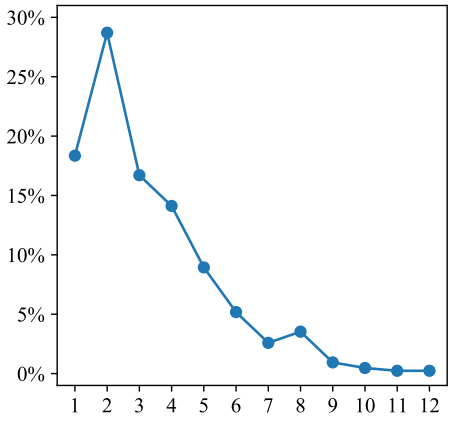


Figure 2: The distance distribution of two nodes with a relationship in FC2Code. The horizontal axis represents the percentage of nodes, the vertical axis represents the distance.

5 Task Definition

Flowchart is composed of nodes and edges. The edge is directed, and the node can be divided into 5 categories: *Operation*, *Start*, *End*, *Condition* and *Inputoutput*. Given a flowchart with n nodes$[\boldsymbol{x}_0, ..., \boldsymbol{x}_n]$. The model needs to transform it into executable code with m code snippets $[\boldsymbol{y}_0, ..., \boldsymbol{y}_m]$.

Note that m and n are not always equal because the code snippets such as *else*, *continue* and *break* do not related to any node in the flowchart. They need to be generated according to the structure of the flowchart. We rewrite $[\boldsymbol{y}_0, ..., \boldsymbol{y}_m]$ as $[\boldsymbol{y}_0, ..., \boldsymbol{y}_n, \boldsymbol{y}'_0, ..., \boldsymbol{y}'_l]$, where $\boldsymbol{y}_i$ is generated from $\boldsymbol{x}_i$ and $\boldsymbol{y}'_i$ is the code snippets that is not related to any node in the flowchart.

The relationships between the nodes $[\boldsymbol{x}_0, ..., \boldsymbol{x}_n]$ and the code snippets $[\boldsymbol{y}_0, ..., \boldsymbol{y}_n]$ are also provided, which can be used in the training phase.

Type	Percentage nodes / flowcharts	Example	
Incorporates the variable declared or described above	46% / 92%	**Previous NL & Code**	The left boundary "low" is equal to 0, and the right boundary "high" is equal to n `low, high = 0, n`
		Current NL:	The left boundary is less than or equal to the right boundary
		Current Code:	`while low <= high:`
Requires understanding of the functions of the variable declared or described above	7% / 38%	**Previous NL & Code**	Declare an array "dp" of length n+1 where dp[i] represents the i-th odd number `dp = [0] * (n + 1)`
		Current NL:	The first odd number is equal to 1
		Current Code:	`dp[1] = 1`
Requires understanding of the properties of the variable and its data structure	4% / 32%	**Previous NL & Code**	Initialize the array "builder" `builder = []`
		Current NL:	Return a valid string "builder"
		Current Code:	`return "".join(builder)`
Requires understanding of what the demonstrative pronoun refers to	9% / 54%	**Previous NL & Code**	The sum of the i-th number and the jth number of the array "num" is target `if nums[i] + nums[j] == target`
		Current NL:	return their array indexs
		Current Code:	`return [i, j]`

The node is related to at least the other 2 nodes in the flowcharts	17% / 74%	**Previous NL & Code**	"tot" is the sum of dresses among washing machines
			`tot = sum(machines)`
			"n" is the number of washing machines
			`n = len(machines)`

		Current NL:	When the sum of dresses cannot be divided into the number of washing machines
		Current Code:	`if tot % n`

Table 3: The relation types and their proportion in FC2Code dataset. Overall, 62% of nodes are related to another node, and 98% flowcharts contain the nodes that are related to another node.

6 Two Stage Code Generation Method

The tokens in the code can be divided into two categories: the first category of tokens describe the execution order of the code snippets, such as *while*, *for*, *if*, *break*, *continue*, *return*. The second category of tokens are used to describe a specific process, such as *assignment* and *comparison*. Similarly, the flowchart is a combination of nodes and edges, the edges describe the execution order of each node. There are two execution orders in the flowchart: *selection* and *loop*, the *loop* in the flowchart should be mapped to the statement *while*, as well as the *selection* should be mapped to statement *if*. Identifying the flowchart structure in advance will reduce the difficulty of the task, and some rule-based methods can identify flowchart structures without errors. Therefore, we can first use a rule-based method to identify the structure of the flowchart and generate a pseudo-code, and then use a code generation model to convert the pseudo-code into executable code. Specifically, given a flowchart with n nodes$[\boldsymbol{x}_0, ..., \boldsymbol{x}_n]$. In the first stage, a *Structure Recognition Model* is used to transform $[\boldsymbol{x}_0, ..., \boldsymbol{x}_n]$ into pseudo-code $[\boldsymbol{z}_0, ..., \boldsymbol{z}_n, \boldsymbol{z}'_0, ..., \boldsymbol{z}'_l]$, $\boldsymbol{z}_i$ is obtained by adding spaces and prefixes like *while*, *if* token before $\boldsymbol{x}_i$, $\boldsymbol{z}'_i$ do not related to any node in the flowchart(e.g. *else*, *continue*, *break*) and will be inserted into $[\boldsymbol{z}_0, ..., \boldsymbol{z}_n]$ as a single line. In the second stage, a *Code Generation Model* is used to transform pseudo-code into executable code $[\boldsymbol{y}_0, ..., \boldsymbol{y}_n, \boldsymbol{y}'_0, ..., \boldsymbol{y}'_l]$.

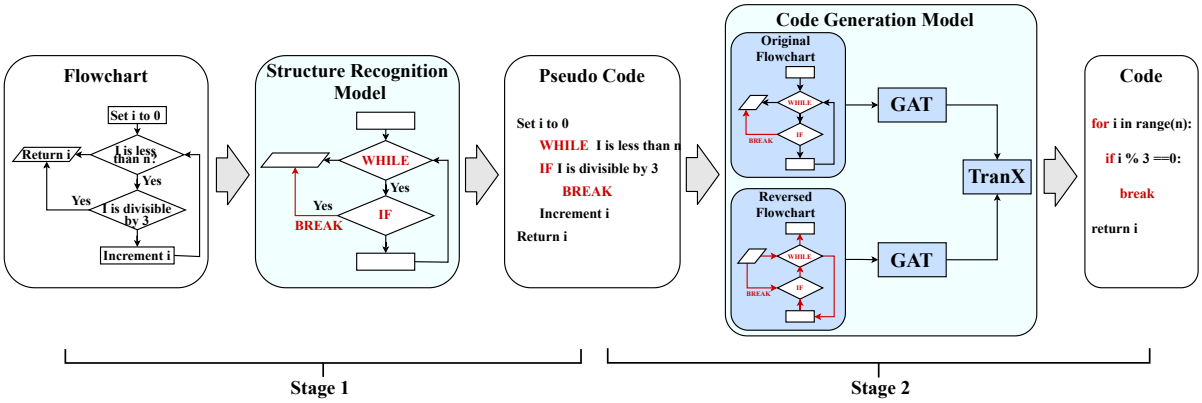


Figure 3: In the first stage, the Structure Recognition Mode is used to transform the flowchart into the pseudo-code. In the second stage, the Code Generation Model is employed to convert the pseudo-code into executable code.

6.1 Structure Recognition Model

In this section we use the Structure Recognition Model to transform flowchart $[\boldsymbol{x}_0, ..., \boldsymbol{x}_n]$ into pseudo-code $[\boldsymbol{z}_0, ..., \boldsymbol{z}_n, \boldsymbol{z}'_0, ..., \boldsymbol{z}'_l]$. Wang et al. proposed to generate pseudo-code from the flowchart in the following steps:

- Step 1: Find out the loop and selection in the flowchart. Flowchart is a combination of two basic structures: selection and loop. Wang et al. found that the flowchart can be seen as a directed graph, in which each loop forms a strongly connected sub-graph. They used this method to find all the loop structures in the flowchart. Then, the structures led by the remaining *Condition* nodes are selection structures.

- Step 2: Identify the nodes (e.g. while) and edges (e.g. continue, break, return) associated with the first category of tokens in the loop. We identified these structures based on their characteristics. For example: 1) The True branch of the continue node will point to the while node. 2) The True branch of the break and return nodes will jump out of the loop.

- Step 3: Determining the scoping of Selection. In the first step, we have found the *Condition* nodes related to selection. In this step, we need to find where the 2 branches of the selection meet.

- Step 4: Generate pseudo-code. To generate the pseudo-code, the model will determine the order of the nodes $[\boldsymbol{x}_0, ..., \boldsymbol{x}_n]$ according to the structure of the flowchart and convert it into the pseudo-code $[\boldsymbol{z}_0, ..., \boldsymbol{z}_n, \boldsymbol{z}'_0, ..., \boldsymbol{z}'_l]$.

We basically follow their method and generate pseudo-code. The full algorithm can be found in their paper (Wang et al., 2012).

6.2 Code Generation Model

In the second stage, we will transform the pseudo-code $[\boldsymbol{z}_0, ..., \boldsymbol{z}_n, \boldsymbol{z}'_0, ..., \boldsymbol{z}'_l]$ into executable code $[\boldsymbol{y}_0, ..., \boldsymbol{y}_n, \boldsymbol{y}'_0, ..., \boldsymbol{y}'_l]$. Because $[\boldsymbol{z}'_0, ..., \boldsymbol{z}'_l]$ are already executable code snippets, they can be directly converted to $[\boldsymbol{y}'_0, ..., \boldsymbol{y}'_l]$ without making any changes. The difficulty lies in converting $[\boldsymbol{z}_0, ..., \boldsymbol{z}_n]$ into $[\boldsymbol{y}_0, ..., \boldsymbol{y}_n]$

For each $\boldsymbol{z}_i$. We first use a bidirectional Long Short-Term Memory (LSTM) to encodes the tokens of $\boldsymbol{z}_i$ into $\boldsymbol{h}_i$, and $\boldsymbol{c}_i$ is the final cell state of LSTM. Then, Graph Attention Networks (Velickovic et al., 2017) is employed to enhance the representation of $\boldsymbol{c}_i$. Specifically speaking, to preserve information about the direction information of the edges in the flowchart, we treat the flowchart as a directed graph and construct the *Reversed Flowchart* by reversing the edges in the flowchart. Then, we use the GAT to fuse the information of $\boldsymbol{c}_i$ with its neighbors $\boldsymbol{c}_j$ according to the original flowchart G_{org} and the reversed flowchart G_{rev} respectively. We set the window size on G_{org} is d_{org}, and the window size on G_{rev} is d_{rev}, then we obtain $\boldsymbol{c}_i$'s new representations $\boldsymbol{c}_{i\_org}$ and $\boldsymbol{c}_{i\_rev}$ respectively. Then, we use MLP to merge $\boldsymbol{c}_{i\_org}$ and $\boldsymbol{c}_{i\_rev}$:

$$\boldsymbol{c}'_i = MLP(\boldsymbol{c}_i : \boldsymbol{c}_{i\_org} : \boldsymbol{c}_{i\_rev})$$

Where [:] denotes vector concatenation. Then, $\boldsymbol{h}_i$ and $\boldsymbol{c}'_i$ is sent to the TranX's decoder (Yin and Neubig, 2018) and generate code snippets $\boldsymbol{y}_i$.

In addition, TranX's decoder employs a hybrid approach of generation and copying, allowing for out-of-vocabulary variable names and literals in $\boldsymbol{z}_i$ to be directly copied to $\boldsymbol{y}_i$. In their work, the copy probability of copying the j-th tokens in $\boldsymbol{z}_i$ is defined using a pointer network. According to Table 3, 46% nodes in the flowchart refer to the variable declared or described above, thus, our model should have the ability to copy tokens from $\boldsymbol{z}_i$'s neighbours $\{\boldsymbol{z}_k\}$, $\{\boldsymbol{z}_k\}$ are the nodes that can reach $\boldsymbol{z}_i$ at most d_{org} steps on graph G_{org}. When the model needs to copy a word from $\boldsymbol{z}_i$ or $\{\boldsymbol{z}_k\}$, we will concatenate their tokens' representation $\boldsymbol{h}_i$ with $\{\boldsymbol{h}_k\}$, and feed them into the point network to calculate the copy probability of each token.

7 Experiment

7.1 BaseLines

We use the following baseline model to transform the pseudo-code $[\boldsymbol{z}_0, ..., \boldsymbol{z}_n, \boldsymbol{z}'_0, ..., \boldsymbol{z}'_l]$ into executable code $[\boldsymbol{y}_0, ..., \boldsymbol{y}_n, \boldsymbol{y}'_0, ..., \boldsymbol{y}'_l]$.

The first baseline model *LSTM* (Agashe et al., 2019) is a neural encoder-decoder model where the encoder computes contextualized representations of input node embeddings using an n-layer BiLSTM, and an LSTM-based decoder produces a sequence of code tokens, while attending to the encoder representations at every time step. For a fair comparison, we concatenate current pseudo-code snippet $\boldsymbol{z}_i$ with 4 context pseudo-code snippets above and 1 context pseudo-code snippet below. Specifically, the inputs of the model are $\{\boldsymbol{z}_{i-4}, ..., \boldsymbol{z}_i, \boldsymbol{z}_{i+1}\}$, the output is $\boldsymbol{y}_i$. We use different segmentation tokens [UP] and [BELOW] to mark pseudo-code above and below respectively, because we find that feeding the model with the direction information of edges can slightly improve model's performance. Another baseline *Transformer* (Agashe et al., 2019) is utilizes the transformer (Vaswani et al., 2017) architecture where the encoder and decoder consist of multiple layers of multi-headed self-attention and position-wise feed forward layers. The inputs and outputs are the same as the LSTM model described above.

7.2 Hyperparameters

The hyperparameters of our model are exactly same as TranX except that the hidden size of our model is 128, we use beam search-based decoding with a beam size of 5 during inference, and trained at most 80 epochs, we pick the model at the epoch with highest validation performance. The hyperparameters for the Transformer and the LSTM baselines are exactly same as Agashe et al. (2019). The training set, validation set and test set contain 220, 50, 50 samples, respectively.

7.3 Experiment Result

In this section, we discuss our experimental results. We use corpus-level BLEU score as performance metrics for code generation. All results are averaged over three runs with different random seeds.

Method	BLEU	
	Dev	Test
LSTM	34.02	38.25
Transformer	41.82	44.05
Ours	**54.26**	**55.68**
-graph	48.89	51.53
-direction	53.48	55.27

Table 4: BLEU score for the code generation task on both the dev and test sets of **FC2Code** for all baselines. LSTM and Transformer are two baseline models proposed by Agashe et al.. Without graph means the window size d_{org} and d_{rev} is set to 0, without direction means we do not construct the "reversed flowchart" and treat the flowchart as an undirected graph.

As shown in Table 4, our model outperforms both LSTM and Transformer baselines by more than 10% BLEU score. We demonstrate the effectiveness of each structure of our model through ablation experiments. Firstly, we verify the importance for GAT by setting the window size d_{org} and d_{rev} to 0. To test the importance of preserving the direction information of edges, we do not construct the "reversed flowchart" and treat the flowchart as undirected graph, which enables nodes to perceive information from both the outgoing and incoming edges at the same time during the convolution.

Table 5 illustrates the effects of window size d_{org} and d_{rev}. We find that increasing d_{org} always helps the model performance, however increasing d_{rev} maybe damage the performance. This is most likely owing to the habit of writing code straight up and down, such as we often define a variable above and then refer to it below.

We can fuse the information of each node $\boldsymbol{z}_i$ according to the flowchart, or directly according to pseudo code. Table 6 shows the impact of different

d_{org}	d_{rev}	BLEU	
		dev	test
0	0	48.89	51.53
0	1	**50.93**	**51.77**
0	2	49.05	50.61
2	0	51.54	53.14
4	0	53.33	55.16
8	0	**55.09**	**55.85**
2	1	52.53	54.44
4	1	54.26	55.68
8	1	**55.18**	**56.71**

Table 5: Models benefit from a larger window size. d_{org} means the window size on G_{org}, d_{rev} means the window size on G_{rev}.

d_{org}	d_{rev}	BLEU	
		flowchart	pseudo-code
0	0	**51.53**	**51.53**
0	1	**51.77**	50.92
0	2	50.61	**50.66**
2	0	53.14	**53.64**
4	0	**55.16**	54.34
8	0	**55.85**	55.58
2	1	**54.44**	53.21
4	1	**55.68**	54.21
8	1	**56.71**	55.85

Table 6: We can get the neighbors of each node according to the flowchart or to the pseudo-code. This table shows the impact of different ways of fusing information on model performance.

ways of fusing information. In most settings, using a flowchart as a metric for calculating distances of each node is usually better than using pseudo-code, a good example to explain this can be found in Figure 1. For the code "for i in range(n):", the input to the model is "I is less than n?" , the related node is "Set i as the index of the array ..." and "Increment i". In the flowchart, the distance for both needed nodes is 1. However, in pseudo-code, the distance between "Increment i" and "I is less than n?" is 4, which means that, to achieve the same performance, the window size d_{rev} will be larger if we use pseudo-code as metric instead of flowchart. (In the pseudo-code, "Increment i" will appear at the end of the loop structure, that is, below the pseudo-code snippet "Return their array indices" in Figure 1.)

8 Conclusion

In this paper, we introduced the task of generating source code from flowcharts with texts. To train models for this task, we constructed a new open-domain dataset (**FC2Code**) from the programming competition platform LeetCode. We propose a two-stage code generation model. In the first stage, the Structure Recognition Algorithm is employed to transform the flowchart into pseudo-code containing the structural conventions of a typical programming language such as *while*, *if*. In the second stage, A code generation model is employed to convert the pseudo-code into codes. The experiments show that it is necessary to enhance each node's representation with its neighbors according to the structure of the flowchart. And the ablation experiments further show that considering the direction of edges in the flowchart will improve the model's performance, and when fusing the information of neighbor nodes, compared with pseudo-code, calculating the distance of two nodes using a flowchart is better, which means that the flowchart is important for the second stage.

Limitations

Because our model is specially designed to fuse the information of each node in the flowchart, this model may not be suitable for fusing information in pseudo-code. Therefore, whether merging the information of each node according to the flowchart or according to the pseudo-code is better is still waiting for further study.

Acknowledgements

The authors would like to thank the anonymous reviewers for their insightful comments. This work was funded by the National Natural Science Foundation of China (62176053).

References

Rajas Agashe, Srinivasan Iyer, and Luke Zettlemoyer. 2019. JuICe: A large scale distantly supervised dataset for open domain context-based code generation. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*, pages

5436–5446, Hong Kong, China. Association for Computational Linguistics.

Martin C Carlisle, Terry A Wilson, Jeffrey W Humphries, and Steven M Hadfield. 2004. Raptor: introducing programming to non-majors with flowcharts. *Journal of Computing Sciences in Colleges*, 19(4):52–60.

John Hutchinson, Jon Whittle, and Mark Rouncefield. 2014. Model-driven engineering practices in industry: Social, organizational and managerial factors that lead to success or failure. *Science of Computer Programming*, 89:144–161.

Srinivasan Iyer, Ioannis Konstas, Alvin Cheung, and Luke Zettlemoyer. 2018. Mapping language to code in programmatic context. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, pages 1643–1652, Brussels, Belgium. Association for Computational Linguistics.

Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, et al. 2022. Competition-level code generation with alphacode. *arXiv preprint arXiv:2203.07814*.

Wang Ling, Phil Blunsom, Edward Grefenstette, Karl Moritz Hermann, Tomáš Kočiský, Fumin Wang, and Andrew Senior. 2016. Latent predictor networks for code generation. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 599–609, Berlin, Germany. Association for Computational Linguistics.

Hui Liu, Mingzhu Shen, Jiaqi Zhu, Nan Niu, Ge Li, and Lu Zhang. 2020. Deep learning based program generation from requirements text: Are we there yet? *IEEE Transactions on Software Engineering*, 48(4):1268–1289.

Yusuke Oda, Hiroyuki Fudaba, Graham Neubig, Hideaki Hata, Sakriani Sakti, Tomoki Toda, and Satoshi Nakamura. 2015. Learning to generate pseudo-code from source code using statistical machine translation. In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 574–584. IEEE.

Shane Sendall and Wojtek Kozaczynski. 2003. Model transformation: The heart and soul of model-driven software development. *IEEE software*, 20(5):42–45.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. *Advances in neural information processing systems*, 30.

Petar Velickovic, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Lio, and Yoshua Bengio. 2017. Graph attention networks. *stat*, 1050:20.

Liming Wang, Guonv Wang, Mingyuan Zhou, Yanli CHU, Ke Chen, and Ping Chen. 2012. Research on and implementation of the algorithm from the program flowchart to the code. *JOURNAL OF XIDIAN UNIVERSITY*, 39(6):70–77.

Xiang-Hu Wu, Ming-Cheng Qu, Zhi-Qiang Liu, Jian-Zhong Li, et al. 2011. Research and application of code automatic generation algorithm based on structured flowchart. *Journal of Software Engineering and Applications*, 4(09):534.

Pengcheng Yin, Bowen Deng, Edgar Chen, Bogdan Vasilescu, and Graham Neubig. 2018. Learning to mine aligned code and natural language pairs from stack overflow. In *2018 IEEE/ACM 15th international conference on mining software repositories (MSR)*, pages 476–486. IEEE.

Pengcheng Yin and Graham Neubig. 2018. TRANX: A transition-based neural abstract syntax parser for semantic parsing and code generation. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, pages 7–12, Brussels, Belgium. Association for Computational Linguistics.

Tao Yu, Rui Zhang, Kai Yang, Michihiro Yasunaga, Dongxu Wang, Zifan Li, James Ma, Irene Li, Qingning Yao, Shanelle Roman, Zilin Zhang, and Dragomir Radev. 2018. Spider: A large-scale human-labeled dataset for complex and cross-domain semantic parsing and text-to-SQL task. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, pages 3911–3921, Brussels, Belgium. Association for Computational Linguistics.

Tao Yu, Rui Zhang, Michihiro Yasunaga, Yi Chern Tan, Xi Victoria Lin, Suyi Li, Heyang Er, Irene Li, Bo Pang, Tao Chen, Emily Ji, Shreya Dixit, David Proctor, Sungrok Shim, Jonathan Kraft, Vincent Zhang, Caiming Xiong, Richard Socher, and Dragomir Radev. 2019. SParC: Cross-domain semantic parsing in context. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, pages 4511–4523, Florence, Italy. Association for Computational Linguistics.

John M Zelle and Raymond J Mooney. 1996. Learning to parse database queries using inductive logic programming. In *Proceedings of the national conference on artificial intelligence*, pages 1050–1055.