International Journal of Information, Control and Computer Sciences
ISSN: 2517-9942
Vol:14, No:6, 2020

GRCNN: Graph Recognition Convolutional Neural Network for Synthesizing Programs from Flow Charts

Lin Cheng, Zijiang Yang

Abstract—Program synthesis is the task to automatically generate programs based on user specification. In this paper, we present a framework that synthesizes programs from flow charts that serve as accurate and intuitive specification. In order doing so, we propose a deep neural network called GRCNN that recognizes graph structure from its image. GRCNN is trained end-to-end, which can predict edge and node information of the flow chart simultaneously. Experiments show that the accuracy rate to synthesize a program is 66.4%, and the accuracy rates to recognize edge and node are 94.1% and 67.9%, respectively. On average, it takes about 60 milliseconds to synthesize a program.

Keywords—program synthesis, flow chart, specification, graph recognition, CNN.

I. Introduction

PROGRAM synthesis enables people to program computers without training in coding. It has been used in many domains such as data wrangling, graphs, and code repair [1]. A good example is FlashFill [2], which allows spreadsheet users to provide a few examples and generates a program that conforms to the examples.

To synthesize a program, specification must be provided. Specification in formal language can accurately represent the user intent and is used in deductive program synthesis [3]. However, very few have the knowledge of formal language, so it cannot benefit most end users. Under-specification is used in programming by example [2], [4], and programming by demonstration [5]. Under-specification does not require language knowledge and is accessible by most end users. However, because there may be more than one program that satisfies the specification, how to choose the correct program that captures the user intent is still an open problem.

In this paper, we propose a new technique called GRCNN (Graph Recognition Convolutional Neural Network) that takes as input a flow chart as accurate specification, and uses deep convolutional neural network (CNN) to analyze the given image, and compiles the obtained information into program code. GRCNN is an end-to-end network that shares the computation of a rich convolutional feature vector and predicts edge and node information simultaneously. A flow chart is a diagram that represents the workflow of a program. It is used widely in textbooks to teach coding and illustrate programs. Moreover, flow charts are intuitive, which allows users to focus on programming logic instead of language details. Thus they are also frequently used in program designing stage. Fig. 1 shows a flow chart representing the workflow of a function that computes the absolute value of an input. Because flow charts can accurately describe programs, synthesizing program by flow chart may precisely capture users' intent.

L. Cheng and Z. Yang are with the Department of Computer Science, Western Michigan University, Michigan, USA (e-mail: lin.cheng@wmich.edu and zijiang.yang@wmich.edu).

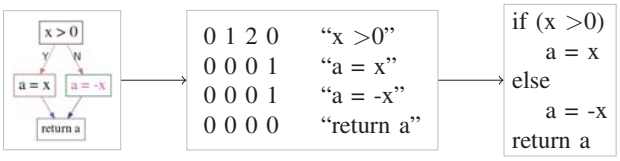

Fig. 1 Overview of GRCNN. The input is the flow chart of `abs` function. The middle is the adjacent matrix and text of each node generated by GRCNN. The output is the synthesized source code

Due to recent progress in deep learning, CNN is powerful enough to detect and recognize complicated information from image. Therefore, we can obtain a graph data structure from a flow chart with the help of deep CNN. Because neural networks are differentiable functions, our method does not suffer from the combinatorial explosion problem that plagues traditional program synthesis methods.

Fig. 1 givens an overview of our approach. First, an image of flow chart is resized to a fixed size and fed to GRCNN. Then, GRCNN generates the graph information including an adjacent matrix for the edges and a list of strings for the nodes. Finally, we compile the graph information to source code.

Our evaluation showed that it is feasible to share the convolutional vector between edge and node networks. Experiments on our synthetic test dataset show that the accuracy rate to synthesize a program is 66.4%, and the accuracy rates to predict edges and nodes are 94.1% and 67.9%, respectively. Experiments on another dataset, which is manually converted from a textbook, show that that the accuracy rates to synthesize a program is 63.6%, and the accuracy rates to predict edge and nodes are 72.7% and 81.8%, respectively. The average time to synthesize a program is about 60 milliseconds.

In summary, the main contributions include:

- We propose a deep neural network that parses graph edge and node information from flow chart.
- We propose to use flow chart as accurate and intuitive specification for program synthesis.
- We have implemented a prototype and conducted empirical study.

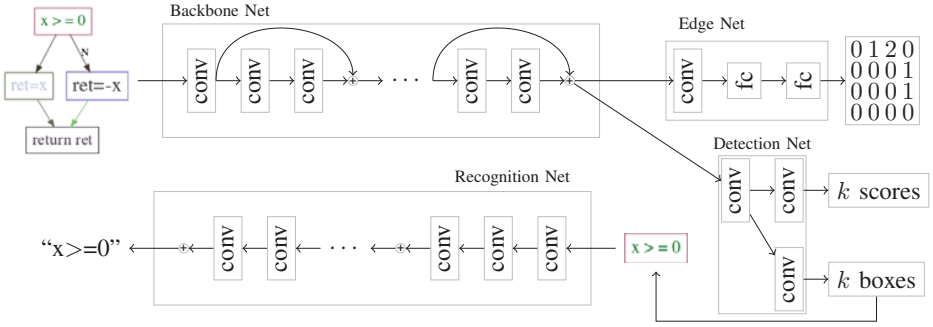

Fig. 2 Architecture of GRCNN

II. Related Work

Traditional methodology of program synthesis is to construct a program space and design search algorithms to find a solution that satisfies the specification. The program space usually grows exponentially with the size of the target program. Different methods are proposed to speedup the search. For example, Flashfill, [2], synthesizes string-transforming programs given input-output examples. It uses dynamic programming to speedup the search. Morpheus, [6], enumerates nested queries and prunes by grouping programs with same input-output pairs. Sketch, [7], and Sqlsol, [8], encode the synthesis problem into logic constraints and delegate the searching algorithm to modern SMT solvers. This approach can boost the performance because modern SMT solvers are implemented for efficiency. Though different methods are proposed to speedup the synthesis process, the underlying complexity is unchanged and scalability is still an issue when it comes to large programs. Another issue is how to capture users' intent. Typically, synthesis algorithms terminate when the first solution is found or the best-ranking solution is found. They may ask a user to provide more examples. However, there is no guarantee that the solution precisely captures the users' intent.

Recently researchers propose to use machine learning to speed up the search for synthesized program. Morpheus [4] uses statistics to rank R program sketches, and uses the rank to guide the search. DeepCoder [9] augments beam search with deep learning recommendation, and the speedup is significant. However, the underlying complexity is still unchanged.

Researchers proposed natural language based program synthesis techniques. SQLizer, [10], synthesizes SQL queries from natural language. Locascio et al, [11], synthesizes regular expressions from natural language. GRCNN differs from these approaches in that our input is different. Flow charts can accurately specify the users' intent, while natural language is ambiguous.

Faster RCNN, [12], and LPRNet, [13], are closely related to our work. Two subnets of GRCNN are built following their ideas. Faster RCNN is a deep neural network for object detection. It slides a small window on a convolutional feature vector and generates box proposals relative to anchors at each position. The box proposal is used to crop the image for a classifier to detect the class of the object in it. Because of the shape and positioning of nodes in a flow chart are different from those in general object-detection jobs, GRCNN chooses different anchors and methods to select the proposals. LPRNet is a license plate recognizing deep convolutional neural network. It reads an image of a license plate and generates a sequence that preserves the spatial order of the characters. A feature of LPRNet is that it has only one deep convolutional network, while other work has both a CNN for feature extraction and a recurrent neural network for prediction. This feature helps to limit the number of subnets in GRCNN, since it already has four subnets. Instead of taking a license plate as input, GRCNN takes as input a crop of image whose boundary is predicted by another network.

III. Network

In this section, we describe the network and its subnet. Then, we describe the loss function and training details.

The input of GRCNN is an image of a flow chart. A flow chart is a graph diagram that represents the work flow of program [14]. Standard flow charts have several shapes for nodes. In this work, we consider all shapes of nodes as rectangles, because we use heuristic algorithms to generate ground truth data for bounding boxes, designing and implementing algorithms for every shape requires considerable engineering effort. However, given ground truth data for bounding boxes, our algorithm can be trained the same way to handle other shapes.

The input flow chart is resized to a fixed size (400 pixels for height and 200 pixels for width) before being fed to the neural network. If both height and width are less then the fixed size, we pad the image by zeroes. Otherwise, we interpolate the image to the fixed size. The output of GRCNN is a graph representation of the flow chart, including an adjacent matrix for the edge representation and a list of text for content in nodes. In addition to the original text, we insert an *id* to the text to match the text with the *id*-th row of adjacent matrix. In this project, we use an alphabet of 50 characters including English characters in lower case, digit characters, arithmetic operators and other symbols. The source code that our algorithm produces supports sequential statements and control structure including IF-ELSE, WHILE loop, and DO-WHILE loop.

Fig. 2 is the overview of GRCNN. GRCNN has four parts: backbone network, edge network, node detection network, and

TABLE I
BASIC BLOCK ARCHITECTURE

Layer Type	Parameters
Input	$C_{in} \times H \times W$
Conv2D	$C_{out}/4, 3 \times 3$
Conv2D	$C_{out}/4, 3 \times 3$
Conv2D	$C_{out}/4, 3 \times 3$
Conv2D	$C_{out}, 3 \times 3$

TABLE II
BACKBONE NETWORK ARCHITECTURE

Layer Type	Parameters
Input	C × H × W
Basic Block	15, 3 × 3
Max Pool	2 × 2
Basic Block	50, 3 × 3
Max Pool	2 × 2
Basic Block	200, 3 × 3
Basic Block	400, 3 × 3
Max Pool	2 × 2

node recognition network. The backbone network takes as input the raw image of the flow chart and outputs a feature vector. The feature vector is fed to both the edge network to produce the adjacent matrix of the flow chart, and the node detection network to generate the bounding boxes of each node, which is used to crop the node from the original image. Then, the crop of each node is fed to the node recognition network to generate the text in the node.

A. Backbone Network

The backbone network is a deep CNN that takes as input an image of size $C_{in} \times H \times W$. The output is a rich feature vector, which is later used as input to the edge network and the node detection network. The backbone network is a sequence of four basic blocks. Table I shows the architecture of a basic block, which takes as input a feature vector of C_{in} channels and outputs a feature vector of C_{out} channels and the same height and width. Each convolutional layer in the basic block is followed by a batch normalization layer and a ReLU activation layer. Each basic block is followed by a max-pooling layer, except the third one. During training, a dropout layer (p=0.1) is added after each pooling layer.

Optionally, the backbone network can be made a residual learning network (ResNet), [15], by modifying the basic block as follows. We perform a down-sampling on the input vector with 1×1 convolutional layer to C_{out} channels, and add the result to the original output as new output.

B. Edge Network

The edge network takes as input the feature vector generated by the backbone network and outputs an adjacent matrix, which is the edge representation of the flow chart. The element of the adjacent matrix at row i and column j has three possible values: 0, 1, 2, which encode no edge, normal edge or `YES` branch of a decision node, `NO` branch of a decision node from node i to node j, respectively. Because the number of nodes in the flow chart may vary, we pad the adjacent matrix to a fixed size PAD by zeroes. We set $PAD = 6$ in this work. We encode the three values in the adjacent matrix with one-hot-vectors of length 3. Therefore, the edge network outputs a vector of $PAD \times PAD \times 3$ scores.

TABLE III
EDGE NETWORK ARCHITECTURE

Layer Type	Parameters
Conv2D	$400, 3 \times 3$
ReLu	
Max Pool	3×3
Linear	$400 * 16 * 7 \times 400$
Tanh	
Linear	$400 \times 6 * 6 * 3$
Tanh	

Table III shows the architecture of the edge network. It first performs a convolutional layer activated by a ReLU function. Then, a max pooling layer is performed followed by two linear layers actived by Tanh function. We observed the training converges significantly faster when using Tanh activation function in the linear layers than using other activation functions.

The loss function to train the edge network is the multi-class multi-classification hinge loss . Equation 1 is the formula of the loss function, where x is the input vector and y is the target class indices.

$$l_e(x, y) = \sum_{i,j} \frac{max(0, 1 - (x[y[j]] - x[i]))}{x.size(0)} \quad (1)$$

C. Node Detection Network

The node detection network takes as input the convolutional feature vector from the backbone network and outputs a set of rectangular node proposals, each with an objectness score.

The network slides a small network over the input convolutional layer. The output of the window is fed into two sibling layers: a box-regression layer and a box-classification layer. Because the small networks work in the sliding window fashion, they are naturally implemented with convolutional networks.

At each sliding window, we predict one region proposal which encodes the four coordinates of a box and one score which estimates the probability of whether the proposal is a node or not. Each region proposal is parameterized relative to a reference box, called anchor. We parameterize the coordinates of the bounding boxes following [16]:

$$t_x = (x - x_a)/w_a, t_y = (y - y_a)h_a,$$
$$t_w = log(w/w_a), t_h = log(h/h_a)$$
$$t_x^* = (x^* - x_a)/w_a, t_y^* = (y^* - y_a)/h_a,$$
$$t_w^* = log(w^*/w_a), t_h^* = log(h^*/h_a),$$

where x, y, w, and h is the box's center point and width and height. Variables x, x_a and x^* are for the predicted box, anchor box, and ground-truth box respectively (likewise for y, w, h). This parameterization converts large integers of bounding box coordinates to variables close to interval $[-1, 1]$, and therefore improves the numerical performance of the algorithm.

To train the node detection network, we assign a binary class label of being a node or not to each anchor. We assign a positive number to two kinds of anchors: (1) The anchor with the highest Intersection-over-Union (IoU) with a ground-truth box, or (2) an anchor that has an IoU higher than 0.9 with any ground-truth box. We assign a negative number to an anchor if its IoU is less than 0.3 for all ground-truth boxes. An anchor that is neither positive nor negative does not contribute to the train objective. Because the number of positive labels and negative anchors may be different and therefore the train may be biased toward one direction, we sample from the more to ensure the same size of positive and negative labels during training.

We apply binary cross entropy loss to the objectness, and smooth-l1 loss to the region proposal. The final loss is the sum of the objectness loss and region proposal loss over all anchors.

For prediction, we choose top 50 anchors with the highest scores, and group them by the condition that anchors with IoU greater than 0.2 are in the same group. Then, we choose from each group the highest score as the final prediction.

Nevertheless, our network differs from the faster RCNN in two ways. (1) Because the nodes in flow charts have similar size and shape, we use one anchor, instead many anchors with different size and ratio, to save computing power. (2) Because the nodes in flow charts do not overlap, we consider proposals in the same group if one has IoU over a threshold with any other one in the group. Meanwhile, fast RCNN considers proposals in the same group if one has IoU over a threshold with the one of highest score.

The architecture of the node detection network is as follows. The intermediate layer is a Conv2D layer with 3 X 3 kernel and 400 output channels. The classifier is a Conv2D layer with 3 X 3 kernel and 1 output channel. The regressor is a Conv2D layer with 3 X 3 kernel and 4 output channels. Each Conv2D layer is activated by ReLU function.

D. Node Recognition Network

The node recognition network takes as input the crop of each node and outputs a vector of size $LENVOC \times 1 \times W$, where $LENVOC$ is the vocabulary size.

Table IV is the architecture of node recognition network, where the base block shares the same architecture as the basic block in the backbone network, Table I,

The output of the node recognition network can be interpreted as probability distribution over the vocabulary at each position along the width direction. CTC loss is a function for training sequential problems such as handwriting recognition or speech recognition. CTC loss does not attempt to learn the character boundaries, and can be applied if the input is a sequence with some order. By adoption CTC loss, we do not need to use another recurrent neural network to predict the text. Instead, we directly predict from the output of the node recognition network.

When predicting, we use greedy search to decode text from the output vector.

Optionally, a spatial transformer network (STN), [17], can be inserted before the node recognition network to further adjust the the boundary of the crop. STN is the network that can be inserted into existing convolutional architectures, giving neural networks the ability to actively spatially transform feature maps.

TABLE IV
NODE RECOGNITION NETWORK ARCHITECTURE

Layer Type	Parameters
Input	C × H × W
Basic Block	64, 3 × 3
Max Pool	3 × 3
Basic Block	128, 3 × 3
Max Pool	3 × 3
Basic Block	256, 3 × 3
Max Pool	3 × 3
Basic Block	LENVOC, 3 × 3

E. Train and Implementation

The whole network is trained end-to-end. The lost function is the sum of loss of all sub-networks:

$$loss = loss_{edge} + loss_{ndc} + loss_{ndr} + \sum_{node} loss_{nr}, \quad (2)$$

where $loss_{edge}$, $loss_{ndc}$, $loss_{ndr}$ and $loss_{nr}$ are the loss of edge network, classifier and regressor of node detection network and node recognition network, respectively.

We use SGD method to train the network. Learning rate is 0.02, and halved when the error plateaus. The total epoch is 200.

IV. EXPERIMENT

We designed experiments to answer the following research questions: (1) what is the accuracy of GRCNN ? (2) what is the inference performance of GRCNN? (3) Is GRCNN able to synthesize real-world program?

The experiments are conducted on a desktop with Geforce 1070 GPU, Intel i7 CPU, and 16GB memory. GRCNN is implemented with PyTorch.

We tested the following networks.

- GRCNN: which is our basic network
- Separated GRCNN: not sharing the backbone network. See Section IV-D.
- GRCNN with ResNet: add optional residual learning. See Section III-A.
- GRCNN with STN: add option STN network. See Section III-D.
- GRCNN with ResNet and STN.

A. Dataset Generation

We created a synthetic dataset to train and test GRCNN, since there are no existing datasets we can use.

A data sample includes a flow chart image in PNG format and a text file containing the ground truth which is used to train the network. The ground truth includes the adjacent matrix which represents the edge information, and the bounding box and text content for each node.

The dataset contains flow charts with 3 to 6 nodes, and 0 to 2 decisions. The text in each node contains 3 to

TABLE V
ACCURACY IN PERCENTAGE OF GRCNN

	Edge	Squence	Nodes	Graph
GRCNN	94.1	90.6	67.9	66.4
+STN	90.0	93.8	71.3	64.4
+ResNet	91.9	91.3	64.8	60.3
+STN+ResNet	90.9	93.2	70.0	63.2
Separated	93.7	91.1	68.2	65.8

The columns are Edge, Sequence, Nodes and Graph accuracy, respectively.

TABLE VI
TIME COST IN MILLISECONDS OF GRCNN .

	BB	Edge	ND	NR	GRCNN
GRCNN	3.5	0.4	42.4	14.2	60.5
+STN	3.5	0.4	43.3	16.6	63.8
+ResNet	3.5	0.4	44.8	14.0	79.1
+STN+ResNet	3.6	0.5	47.0	16.7	84.3
Separated	3.5	0.4	42.8	15.1	61.8

The columns are the time cost of the backbone, edge, node detection, node recognition network and GRCNN, respectively

9 random characters from an alphabet of 50 characters, including the lower case English characters, digit characters, arithmetic operators, parenthesis et al. Note that the alphabet can be chosen freely without changing the essential network architecture. The train and test dataset contain 9960 and 2490 data samples, respectively.

We draw the flow chart using Graphviz, a popular graph-drawing tool. We set the maximal width to be 200 pixels and maximal height to be 400 pixels.

When drawing the flow chart, we randomize to cover wide range of data samples. The width of lines, including nodes boundaries and edges, is randomly chosen between 1 to 5 pixels. The font size of characters is randomly chosen between 20 to 30. The font color of characters is random RGB color. We draw nodes in rectangles and design a heuristical algorithm to effectively compute high accuracy coordinates of bounding boxes.

B. Accuracy

We measure the Edge Accuracy, Sequence Accuracy, Nodes accuracy and Graph Accuracy of GRCNN prediction. Edge accuracy is the percentage of images whose edge is correctly predicted. We say the edge is correctly predicted if the predicted adjacent matrix is exactly the same as the ground truth. Sequence accuracy is the percentage of nodes whose text content is correctly predicted among all nodes in all images. This item shows how well the node recognition network works for individual nodes. Nodes accuracy is the percentage of images whose nodes are all correctly predicted. This item shows how well the tool predicts the nodes as a whole. Graph accuracy is the percentage of images whose edge and nodes are all correctly predicted.

The accuracy of GRCNN and its subnets are described in the first row of Table V. The Edge, Sequence, Nodes and Graph accuracy are 94.1%, 90.6%, 67.9% and 66.4%, respectively. Note that the Graph accuracy is the result of joint probability of all edges and all nodes, so it is lower than Edge and Sequence accuracy.

TABLE VII
RESULT OF REAL-WORLD PROGRAM SYNTHESIS.

program	Graph	Edge	Nodes	#Nodes	Nodes
abs	1	1	1	4	4
swap	1	1	1	3	3
max	0	1	0	4	3
sum	0	0	1	6	6
max3	1	1	1	6	6
log	1	1	1	4	4
radius	1	1	1	3	3
poly	1	1	1	6	6
factorial	1	1	1	6	6
quadrant	0	0	1	5	5
cntpos	0	0	0	6	3
sum	7	8	9	53	49
percentage	63.6	72.7	81.8	-	92.5

The columns are program name, Graph accuracy, Edge accuracy, Nodes accuracy, number of nodes, correctly predicted nodes.

We conducted ablation study to identify how well the optional enhancements are. In Table V, rows 2-4 show the accuracy of GRCNN with different enhancements. We find that the edge accuracy and graph accuracy decreased with those enhancements. Our explanation is that the extra trainable weights in the enhancements caused the network biased toward the nodes detection and recognition networks, but caused the decrease in edge accuracy, and therefore caused decrease in the graph accuracy. We also find all STN enhancements improved the accuracy of node detection and recognition networks.

C. Performance

Table VI shows the performance of GRCNN and its subnets. The overall time cost of GRCNN is about 60 milliseconds, and the performance of other networks are close to the performance of GRCNN. Among the subnets, the time cost of node detection network is the major part, 70.1%, of the overall cost. The rest are the node recognition network (23.5%), backbone network (5.8%), and edge network (1%).

D. End-to-End vs Separated Network

Because edge and node information do not depend on each other, it is natural to consider to use two separate networks to predict edges and nodes. We designed experiments to see how well both ideas work.

We made two clones of GRCNN and modified as follows. For one clone, we disable the node detection network and node recognition network. For the other clone, we disable the edge detection network. We trained the two networks separately with the same dataset and hyper parameters.

The row *Separated* in Table V and VI show the accuracy and performance of the separated networks. The performance is close to GRCNN, therefore it is feasible to share the computation of the backbone network. The sharing saves 3.5 milliseconds, which is 5.8% of the time cost.

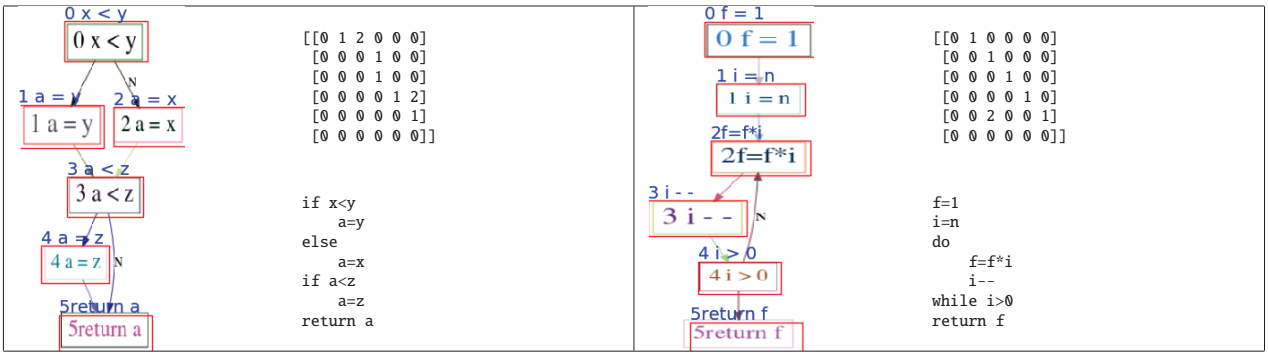

Fig. 3 Samples of GRCNN prediction. The predicted bounding boxes are drawn in red lines. The predicted text is drawn above the bounding box in blue characters. On the right is the predicted adjacent matrix and synthesized source code

E. Real-world Program Synthesis

In order to see how well GRCNN synthesizes real-world programs, we created a test dataset of 11 programs, which are selected from the problems in a programming textbook [18].

Table VII is the testing result. It shows that 63.6% of programs are correctly predicted, 72.7% edges are correctly predicted, 81.8% nodes are correctly predicted.

In Fig. 3, we draw two samples to visually demonstrate the input and output of GRCNN. The first one is the function that finds the max of three numbers. The second one is the `factorial` function.

V. Conclusion

We presented GRCNN, a deep convolutional neural network that parses graph data structure from a flow chart, and we automatically generate source code that matches the flow chart. GRCNN predicts the edge information and nodes information simultaneously. Experiments show that we can share the computation of the feature vector. GRCNN achieves 66.4% accuracy on our test dataset and close accuracy on a real-world dataset. GRCNN takes about 60 milliseconds to synthesize a program from an image of flow chart.

References

[1] S. Gulwani, "Dimensions in program synthesis," in *Proceedings of the 12th international ACM SIGPLAN symposium on Principles and practice of declarative programming*. ACM, 2010, pp. 13–24.

[2] ——, "Automating string processing in spreadsheets using input-output examples," *ACM Sigplan Notices*, vol. 46, no. 1, pp. 317–330, 2011.

[3] Z. Manna and R. Waldinger, "A deductive approach to program synthesis," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 2, no. 1, pp. 90–121, 1980.

[4] Y. Feng, R. Martins, J. Van Geffen, I. Dillig, and S. Chaudhuri, "Component-based synthesis of table consolidation and transformation tasks from examples," in *ACM SIGPLAN Notices*, vol. 52, no. 6. ACM, 2017, pp. 422–436.

[5] T. A. Lau and D. S. Weld, "Programming by demonstration: An inductive learning formulation," in *International Conference on Intelligent User Interfaces: Proceedings of the 4 th international conference on Intelligent user interfaces*, vol. 5, no. 08. Citeseer, 1998, pp. 145–152.

[6] C. Wang, A. Cheung, and R. Bodik, "Synthesizing highly expressive sql queries from input-output examples," in *ACM SIGPLAN Notices*, vol. 52, no. 6. ACM, 2017, pp. 452–466.

[7] A. Solar-Lezama and R. Bodik, *Program synthesis by sketching*. Citeseer, 2008.

[8] L. Cheng, "Sqlsol: An accurate sql query synthesizer," in *International Conference on Formal Engineering Methods*. Springer, 2019, pp. 104–120.

[9] M. Balog, A. L. Gaunt, M. Brockschmidt, S. Nowozin, and D. Tarlow, "Deepcoder: Learning to write programs," *arXiv preprint arXiv:1611.01989*, 2016.

[10] N. Yaghmazadeh, Y. Wang, I. Dillig, and T. Dillig, "Sqlizer: query synthesis from natural language," *Proceedings of the ACM on Programming Languages*, vol. 1, no. OOPSLA, p. 63, 2017.

[11] N. Locascio, K. Narasimhan, E. DeLeon, N. Kushman, and R. Barzilay, "Neural generation of regular expressions from natural language with minimal domain knowledge," *arXiv preprint arXiv:1608.03000*, 2016.

[12] S. Ren, K. He, R. Girshick, and J. Sun, "Faster r-cnn: Towards real-time object detection with region proposal networks," in *Advances in neural information processing systems*, 2015, pp. 91–99.

[13] S. Zherzdev and A. Gruzdev, "Lprnet: License plate recognition via deep neural networks," *arXiv preprint arXiv:1806.10447*, 2018.

[14] G. B. Shelly and M. E. Vermaat, *Discovering Computers, Complete: Your Interactive Guide to the Digital World*. Cengage Learning, 2011.

[15] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 770–778.

[16] R. Girshick, J. Donahue, T. Darrell, and J. Malik, "Rich feature hierarchies for accurate object detection and semantic segmentation," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2014, pp. 580–587.

[17] M. Jaderberg, K. Simonyan, A. Zisserman *et al.*, "Spatial transformer networks," in *Advances in neural information processing systems*, 2015, pp. 2017–2025.

[18] D. M. Etter, *Introduction to C*. Prentice Hall, 1998.