Reliable Compilation Optimization Selection Based on Gate Graph Neural Network

Jiang Wu, Jianjun Xu, Xiankai Meng

College of Computer

National University of Defense Technology

wujiang_nudt@163.com, jjxu@nudt.edu.cn, mengxiankai12@nudt.edu.cn

Abstract—**For different programs or applications, it is necessary to select the appropriate compilation optimization pass or subsequence for the program. To solve this problem, machine learning is widely used as an efficient technology. However, the most important problem in using machine learning is the extraction of program features. How to ensure the integrity and effectiveness of program information is the key to the problem. In addition, when compiling and optimizing the selection problem, the measurement indicators are often program performance, code size, etc. There is not much research on program reliability which needs the longest measurement time and the most complicated measurement methods. This paper proposes a GGNN-based compilation optimization pass selection model. We extend the deep neural network based on GGNN, and build a learning model which learns heuristics for program reliability. The experiment was performed under the clang compilation framework. The alternative compilation optimization pass adopts the C language standard compilation optimization passes. Compared with the traditional machine learning method, our model improves the average accuracy by 5% ~ 11% in the optimization pass selection for program reliability. At the same time, experiments show that our model has strong scalability.**

Keywords- compilation optimization selection; AST; GGNN; reliability; clang

I. INTRODUCTION

In the past few decades, compiler developers have designed and implemented a large number of compilation optimization options in response to compilation optimization needs in various complex situations. In actual development, the standard compilation optimization pass provided by the compiler is difficult to adapt the requirements for the program to be compiled in complex scenarios. On the one hand, the program to be compiled has different semantics and compilation goals. It is difficult to obtain the optimal optimization effect by using the standard compilation optimization pass directly. If an inappropriate optimization pass is used, it may even bring negative effects about program performance, etc. On the other hand, with the continuous development of the hardware architecture, the compilation environment becomes increasingly complex, and the compilation optimization pass should be adjusted accordingly. Therefore, how to choose the best compilation optimization pass for the program to be compiled among the intricate optimization options. Become a challenging scientific problem. The algorithms used in this field mainly include heuristic search algorithms and machine learning algorithms. The heuristic search algorithm uses a heuristic method to search the optimal compilation optimization pass in the compilation optimization option combination space. For example, the VISTA interactive compilation system [1] uses a combination of genetic algorithms and human-assisted guidance to search for optimal compilation optimization passes; the open source framework " OpenTuner " [2] uses a variety of evolutionary algorithms, including genetic algorithms, to get a speedup of up to 2.8 times; Jantz et al. [3] use genetic algorithms to select the optimal compilation optimization pass for the JIT compiler. And some other selection schemes based on some multi-objective optimization algorithms, for example, Lok et al. [4] [5] use SPEA2, NSGA-II and IBEA to select the compilation optimization pass for the program to be compiled that meets the target code execution speed, scale and other goals.

However, the heuristic search algorithm can generate efficient compilation optimization sequences, but it takes a lot of time to run the entire iterative process. Gradually researchers began to use machine learning algorithms to select compilation optimization sequences. A large number of algorithms based on SVM and LR are widely used. The work [6] used code runtime characteristics to characterize the program to be compiled to train the logistic regression model; Ashouri et al. [7] analyzed the dependencies between optimization options in the compiler's LLVM, using program dynamic characteristics to train the Bayes network, then use this model to predict the optimization options that should appear in the next stage until the prediction is completed; the open source compiler "Milepost GCC" [8], which is a modularized, modified form of GCC4.4 scalable compiler that supports static feature extraction of the program to be compiled, trains machine learning models, and predicts the compilation effect of the compiled optimization sequence. A large number of machine learning algorithms perform feature extraction on programs, both dynamic and static features., it is difficult to extract program information completely and efficiently. Most work has tried to transfer natural language methods and does not capitalize on the unique opportunities offered by code's known semantics. For example, long-range dependencies induced by using the same variable or function in distant locations are often not considered. Such models miss out on the opportunity to capitalize on the rich and well-defined semantics of source code. Therefore, constructing a graph to represent complete program information and training in conjunction with a graph neural network is a more effective way to ensure the integrity of the program information as much as possible.

In addition, from the perspective of compiling optimization goals, most researches focus on the execution speed of the target

machine code [9] [10]. Statistics shows that the target code is used in machine learning algorithms. The acceleration ratio as the optimization target accounted for the vast majority of the research, accounting for more than 80% of this part of the research. Another optimization goal that researchers are concerned about is the size of the target code [11] [12]. It is a very important optimization goal, especially in the case of the current widespread application of embedded programs, reducing the storage space as much as possible can bring significant benefits. However, there is not much research on the use of machine learning for compilation optimization orienting program reliability research.

In order to extract the program information as completely as possible, while taking advantage of the advantages of machine learning, we combine GGNN, program reliability analysis and compilation optimization selection problems. We abstract the C raw code into a graph with data flow and type hierarchies, and then build a program optimization oriented graph neural network for program reliability. Our work replaces the need for compile-time or static code features, merging feature and heuristic construction into a graph and send it to a graph neural network. Then learning to get which clang standard compilation optimization can bring the highest reliability gain for a specific C code. By using the PIN [13] tool for verification, our model has an average accuracy improvement of 5% ~ 11% compared to traditional machine learning algorithms without our extended GGNN. At the same time, our model is also highly scalable and can adjust the size of the output layers to solve different problems.

II. PROGRAM AS GRAPH

A. Abstract Syntax Tree

As an intermediate representation of the source code for parsing and semantic analysis, AST [14] is a tree-structured data describing the syntax rules and execution order of the code, which is obtained after the code is parsed using irrelevant context rules. In the AST, leaf nodes represent identifiers in the source code, while non-leaf nodes represent syntactic structures. As the parse tree of the source code, the AST basically covers the following syntax structures: *Selecting structure* (IF, SWITCH, etc.); *Loop structure* (WHILE, FOR, etc.); *Sequence structure* (expressions, assignment statements, etc.). Therefore, AST, as an intermediate representation of the source code, can effectively retain the syntactic context information related to the programming language.

B. Function Call Graph

FCG [15] is used to characterize information related to control flow in source code. Each node in a function call graph represents a function, and the edges in it represent the calling relationship between functions. Understanding the calling relationship between functions is of great help to understand the hierarchical structure of the program, and clarifying the function calling relationship is a key part of program analysis.

C. Data Flow Graph

DFG [16] explicitly contains the data logic of the two aspects of data transfer and data processing in the source code. Nodes in DFG represent entities, such as variable declarations, operands,

operators, structures, etc., and the edges in them represent the data relationships that exist between these entities. DFG can describe the data logic and program functions of the source code and is used to analyze the dynamic runtime data flow information of the program. From the perspective of data transmission, DFG describes the movement and transformation of data streams from input to output. Because it can clearly reflect the logic that the program must complete, it has become one of the most commonly used methods of program analysis.

int add(int m^1)

{

    int x^1 = Foo(m^2);

    int y^1 = 2;

    y^2 = x^2 + 1;

    return y^3;

}

int Foo(int m)

{

    int n = **Square**(m);

    int k = **Mod**(m);
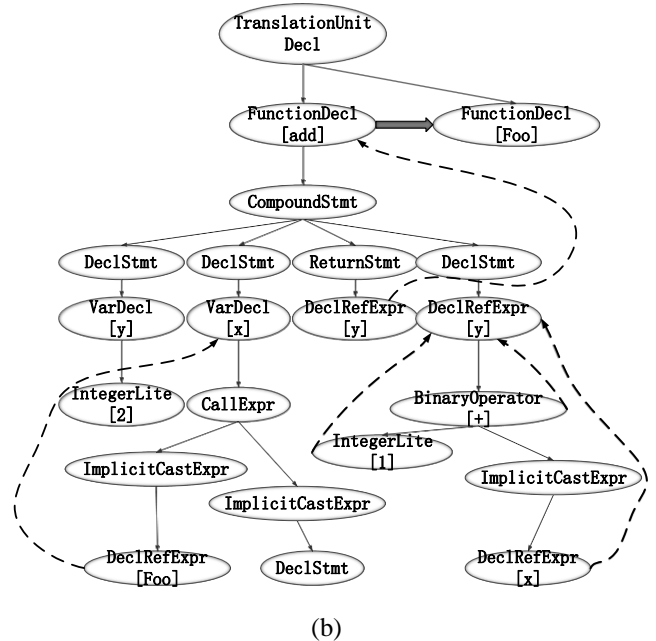
    return n*k;

}

(a)

(b)

Figure 1.   Example of the co-AST.

By comparing the characteristics of AST, FCG, and DFG, it is clear that each different form of the intermediate expression form only describes the program source code from a certain angle. The AST only contains static information related to the grammatical structure, and the latter two are used to describe the runtime dynamic information related to the control flow and the data flow. In particular, the angles of the latter two references are different. FCG starts with a coarse-grained function, while DFG starts with fine-grained variables, operators, and operands. The source code is special executable text, and both static syntax information and dynamic runtime information are important. Therefore, the fusion of the code information carried in the AST,

FCG, and DFG helps reduce the information loss caused by the transformation of source code to intermediate expressions.

We have established a joint program analysis graph co-AST, which combines the characteristics of AST, FCG, and DFG. As shown in Figure 1, based on the source code of Fig. 1 (a), we constructed the co-AST graph as Fig. 1 (b). The complete co-AST graph has a total of seven types of edges. We introduce the edges of FCG and DFG into the original AST structure. As shown in Figure 1 (b), the solid arrow is the function call identifier, and the dashed curve arrow is the identification of the data stream This combination greatly enriches the information in the program graph, thereby speeding up the spread of information in GGNN and improving the effect of model training.

III. CONSTRUCTION OF EXTENDED GGNN

The graph model used in this paper involves a directed graph $G = (V, E)$, where V represents a set of nodes of size $|V|$ and E represents a set of size $|E|$. The nodes in V are represented by node number i, the directed edges in E are represented by e_{ij}, and e_{ij} represents the edge pointed by node i to node j. For different types of edges in the graph, use the edge type set $L_K = \{l_1, l_2, ..., l_k\}$ to represent. The connection relationship between the nodes in the graph is represented by the connection matrix A. There are two design schemes for the dimension of A. The first design is $A \in R^{|V| \times 2|V|}$, directed edge e_{ij} in the figure is seen as two different types of access edges, one is the outgoing edge of node i and the other is the incoming edge of node j. The second design is $A \in R^{|V| \times |V|}$, it only considers the directed edge e_{ij} as the incoming edge of node j. The connection matrix in this paper uses the second scheme.

The element A_{ij} in the i row and the j column is a $d \times d$ matrix (d represents the node state vector dimension). A_{ij} is also called the propagation matrix on edge e_{ij}, which represents the information propagation rules from node i to node j. For example, Fig. 2 (c) shows the connection matrix corresponding to the data flow graph shown in Fig. 2 (b), where the two rectangular-framed matrices are the propagation matrices corresponding to e_{3y} and e_{yz} respectively.

In the assignment statement shown in Fig. 2 (a), we are concerned about whether the number 3 can be passed to the variable z, as shown in Fig. 2 (b). To this end, nodes 3 and z can be regarded as the source node and the target node respectively, and their feature vectors are initialized as $h_3^0 = [1,0]$ and $h_z^0 = [0,1]$ (the first dimension of the two-dimensional vector is 1, which means that 3 can reach the node), and the feature vector table of node y is initialized as $h_y^0 = [0,0]$. The propagation matrix A_{ij} determines how the information of each dimension of node i is propagated to the various dimensions of node j. "0" represents no propagation and "1" represents complete propagation. For example, A_{3y} in Fig. 2 (c) indicates that node 3 only passes the information of its first dimension to the first dimension of node y. In this way, the result of multiplying vector h_3 and A_{3y} is still $h_y = [1, 0]$, indicating that the data has not been passed to the target node z. However, A_{yz} in Fig. 2 (c) indicates that the

information of the first dimension of node y is to be transferred to the first dimension of node z. Therefore, the result of multiplying vector h_y and A_{yz} is $h_z = [1,0]$, indicating that data can reach the target node z.

$$y = 3$$
$$z = y$$

(a) Sample code

(b) DFG
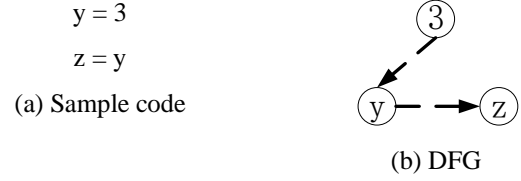
(c) Connection matrix

Figure 2. Example of connection matrix and propagation matrix.

In GGNN [17], the information propagation of nodes on different types of edges is achieved through different multilayer perceptron, and the propagation matrix on the edges is represented by the trainable multilayer perceptron weights $W_e \in R^{d \times d}$. It should be noted that the connection matrix shown in Fig. 2 (c) only represents the weight of the multi-layer perceptron after the graph model has converged on the reachability task. The GGNN model has an iterative t-round of node state information propagation process, as follows: the state information of node i is initialized to a vector $h_i^{(1)} \in R^d$. During the t-th round of iteration, each central node i gathers all neighbor node information to get the node interaction context $m_i^{(t)} \in R^d$, as shown in (1) (where N_i represents the set of neighbor nodes of i). In response to the current interaction context, the node i updates its own state information $h_i^{(t)}$ after t round. The GRU unit is used in GGNN different from GNN. The GRU unit considers the relationship between node status information in different update rounds. That is, when the node updates during the round t-th, the node hidden layer vector expression $h_i^{(t)}$ and the state information $h_i^{(t-1)}$ of the previous round have a time series relationship, as shown in (2). GNN only uses edges as a means of propagation, but does not distinguish the functions of different edges. And GNN does not set independent learnable parameters for edges, which means that some characteristics of edges cannot be learned through the model. This is also the main reason we use GGNN as shown in Fig. 3.

$$m_i^{(t)} = \Sigma_{j \in N_i} A_{ij} \cdot h_j^{(t-1)}. \tag{1}$$

$$h_i^{(t)} = GRU(h_i^{(t-1)}, m_i^{(t)}). \tag{2}$$

Figure 3. Extended GGNN architecture

Message Propagation
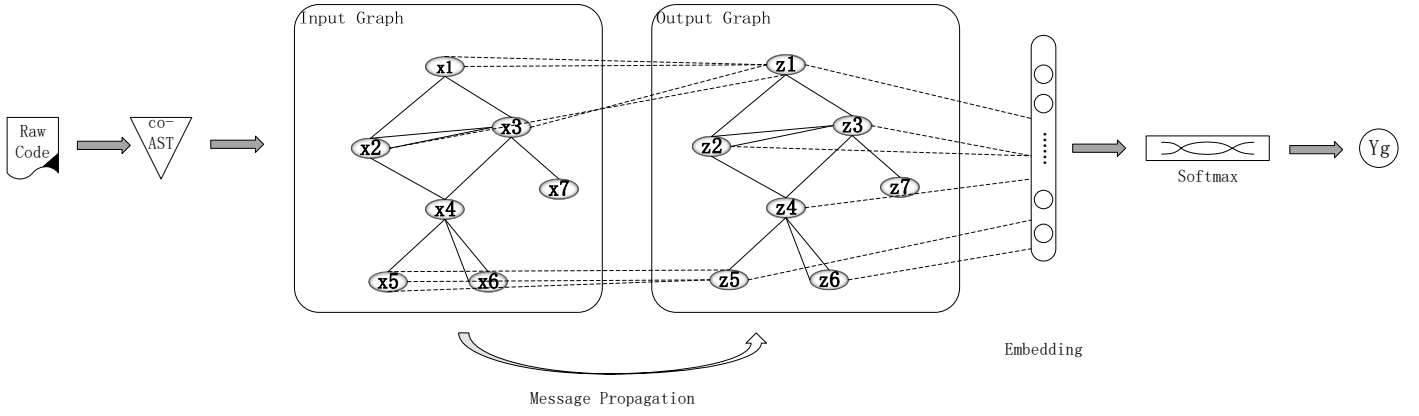
During the information propagation of the graph model, $m_i^{(t)}$ is the interaction context of node i in the whole graph. Whether it is GNN or GGNN, $m_i^{(t)}$ is obtained by directly accumulating the product of feature information $h_j^{(t-1)}$ of the neighbor node j and the propagation matrix A_{ij} on the edge e_{ij}. In the topology of the graph, different nodes have different properties in the topology. In the GNN and GGNN models, the topological properties of the nodes are directly expressed as hidden nodes. Based on this, in the substructure composed of the central node i and its neighbor nodes N_i, our model abandons the way of directly accumulating the product of $h_j^{(t-1)}$ and A_{ij} to calculate $m_i^{(t)}$. We hope that the model automatically learns how to calculate $m_i^{(t)}$ and that the central node could pay more attention to those neighbor nodes whose topology information is important, because these neighbor nodes determine the interaction context of the node i on the graph to a greater extent.

Therefore, we have extended GGNN. We assign different weights to each neighbor node to characterize its importance to the central node α_{ij}, it gets function mapping through neural network $a : R^d \times R^d \to R$, a calculates the correlation coefficient between the central node i and its neighbor j, and uses the *softmax* function to normalize the correlation coefficients of all neighboring nodes, such as (3) shown:

$$\alpha_{ij} = softmax \ (a \ (h_i^{(t-1)}, h_j^{(t-1)})). \tag{3}$$

The weight parameter of the neural network a is only related to the round of information propagation. The same round of information propagation, a is shared by all nodes. Different propagation rounds have different parameters for a. The uncertainty of the number of neighbor nodes j of a node i will result in a variable number of α_{ij}, and it is not possible to directly implement the *softmax* function provided by the framework Tennsorflow. This paper implements *softmax* to adapt to the changing number of neighbor nodes:

$$a_{ij} = a_{ij} - max(a_{i1}, a_{i2}, ..., a_{ij}). \tag{4}$$

$$\alpha_{ij} = \frac{e^{a_{ij}}}{\Sigma_{k \in N_i} e^{a_{ik}}}. \tag{5}$$

So the interaction context $m_i^{(t)}$ of node in of our expanded GGNN is shown in (7):

$$m_i^{(t)} = \sum_{j \in N_i} \alpha_{ij} \cdot A_{ij} \cdot h_j. \tag{6}$$

Because the selection of program compilation optimization pass is to analyze the program according to the embedded expression of the program algorithm graph co-AST, after obtaining the final graph node embedding vector expression $h_i^{(T)}$, the embedding vector h_G of the entire co-AST graph needs to be calculated. This paper proposes a node vector probability fusion method, which generates a graph embedding vector from the node embedding vector. As shown in (7), the $f(h_i^{(T)}, h_i^{(1)})$ is a fully-connected neural network, which learns the probability that node i will be fused based on node attributes $h_i^{(1)}$ and topology information $h_i^{(T)}$. The activation function in f uses *sigmoid*, whose final output is a value of [0, 1]. The g is also implemented by using a fully connected layer neural network, which uses the *tanh* function to activate the output. The calculation of h_G is similar to (6). In the end, the program compilation optimization pass selection l_G is derived from the function *sofmax*:

$$h_G = \sum_{i \in V} f\left(h_i^{(T)}, h_i^{(1)}\right) \cdot g(h_i^{(T)}). \tag{7}$$

$$l_G = softmax \ (h_G). \tag{8}$$

In our extended GGNN, the computation of α_{ij} of central node and its neighbor nodes can be parallelized and computationally efficient. Moreover, it implements the calculation method of the model automatically learning the interaction context, without having to consider the number of neighbor nodes that changes. If using neural networks to learn to calculate the interaction context, it will definitely need to face the problem

that the neural network weight dimensions cannot be unified due to the inconsistent number of neighbors in each node.

IV. EXPERIMENT AND ANALYSIS

In order to verify the effectiveness of the model proposed in this paper in the selection of reliability-oriented program compilation optimization pass, we not only evaluate the model from the perspective of pass selection accuracy, but also analyze the ability of the model to learn topology from the perspective of co-AST graph node embedding expressions.

A. Configuration

In order to cover more program categories in our training set, we have expanded on the standard C test suite MiBench [18]. We still adopt the program classification method of MiBench, but we have made a lot of expansions in the number of programs. We use the open source compilation tool *clang* to parse the raw code to get the program's AST data set. We further add edges representing data flow information and edges representing function call graphs to the AST tree for each program, and finally obtain the final co-AST data set. In our experiments, the co-AST data set is divided into a training set, a validation set, and a test set according to a ratio of 8: 1: 1. We use the PIN tool to evaluate the reliability of the program and generate the training set. Our program reliability evaluation indicators refer to Sridhran [19].

The optimization algorithm used for model training is the SGD of the ADAM optimizer [20]. The loss function uses cross entropy. The weight parameter initialization in the model uses Glorot [21] initialization method. In the experiment, the information propagation layer (information iteration round) is set to 4 layers, and the number of neurons in each propagation layer, that is the propagation matrix vector dimension d, is a hyperparameter. The choice of this hyper-parameter mainly considers the speed of model convergence and the model's loss value. For this reason, we determined after experiments that when the hidden layer vector dimension d is 270, the model's convergence loss value is relatively small, and the model training speed is also relatively fast. Therefore, we set the hidden layer vector dimension d to 270 to complete the subsequent experiments.

B. Result and analysis

In the experiment, we construct the classification task of 4. The main content of this task is to judge, for a specific C raw code, when using the clang standard compilation optimization passes -O1, -O2, -O3, and -OS, which one is more reliable for the program. This is different from many others that focus on the impact of compilation optimization passes on program speed and code size. The benchmark comparison experiment selected in this experiment is TreeBased Convolution Neural Network (TBCNN). To our knowledge, TBCNN is by far the best performing work on source code classification tasks. In addition, LSTM [22] is widely used in text classification tasks and our model is aimed at the improvement of the GGNN model. We also test the LSTM and GGNN models. The experimental results are shown in TABLE I, where exGGNN is our extended GGNN. Therefore, there are four models in the controlled trial, LSTM, TBCNN, GGNN and exGGNN.

TABLE I.   ACCURACY OF OPTIMIZATION PASS SELECTION

Different Model	Accuracy		
	Minimum	*Maximum*	*Average*
LSTM	82.2%	84.3%	83.9%
TBCNN	84.5%	88.6%	86.7%
GGNN	87.3%	93.5%	89.2%
exGGNN	**87.9%**	**98.1%**	**94.1%**

The experimental results in Table 1 show that the accuracy of exGGNN in the code optimization pass selection problem has improved significantly, indicating that our model has achieved the expected results for this problem. Better than LSTM and TBCNN shows that choosing GGNN to deal with such problems is a better choice, and better than GGNN shows that our extension has played a important role. Then, in order to evaluate whether the data flow edge and function call graph edge are useful, we remove one of the 7 types of edges and use the exGGNN model to learn the co-AST graph after deleting a certain edge to implement the optimization pass selection. Observe the effect of each edge on the selection accuracy of the program. The experimental results are shown in TABLE II. The double underline indicates the co-AST graph with this type of edge removed.

TABLE II.   TABLE TYPE STYLES

Different Edge	Program Category (MiBench)					
	auto-mo-tive	con-sumer	net-work	of-fice	secu-rity	tele-comm
AST	**0.99**	0.02	0.09	**0.95**	0.06	0.09
Operand	**0.92**	0.07	0.07	**0.92**	0.02	0.05
LastUse	**0.92**	0.07	0.07	**0.12**	0.02	0.05
Compute	**0.92**	0.07	0.07	**0.92**	0.02	0.05
Return	**0.94**	0.07	0.07	**0.92**	0.02	0.09
Formal	**0.99**	0.08	0.08	**0.05**	0.02	0.05
Call	**0.92**	0.07	0.07	**0.92**	0.03	0.05

The experimental results show that for most program tasks in MiBench, deleting a certain type of edge has little effect on the accuracy of program optimization pass selection accyracy. There may be information redundancy in the seven types of edges, so any type of edge deletion will not have a significant impact on the accuracy of program classification. But for some programs, deleting these types of edges can significantly reduce or improve the accuracy of program classification. Therefore, the construction of co-EAST is effective and can further improve the extracted program information.

We also compare the convergence trend of exGGNN / GGNN / TBCNN loss values in the co-AST / AST intermediate expression form. As shown in Fig.4, the graph model not only has a smaller final convergence loss value, but also has a faster convergence rate than TBCNN. The reason why the graph network model converges faster is that the graph model has

stronger constraints on AST nodes than TBCNN. More specifically, in the TBCNN model, the convolution operation forces one-way propagation of information from child nodes to the parent node. While the graph model involves, for each node, it is two-way information propagation between all neighboring nodes. This information dissemination can gradually spread to the entire graph structure.
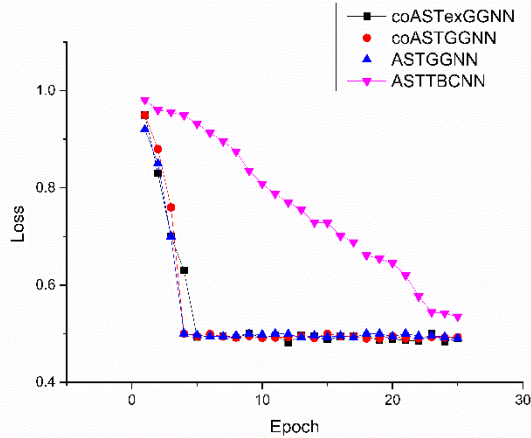

Figure 4.　Variation of loss values for the four models.

In order to select a highly reliable compilation and optimization pass, we propose a learning strategy through GGNN. The experimental results show that our model achieves a higher accuracy on the pass selection problem and is better than similar neural networks models. As the first attempt to combine graph neural networks with program reliability, we obviously achieved our experimental goals. Although the program's running time and code size are important indicators of program evaluation, the reliability of the program can not be ignored, especially in the booming aerospace field, the reliability of the program is always the first consideration. We are working on combining optimization sequence generation and graph neural networks, hoping to find a better solution to the phase ordering problem.

REFERENCES

[1]　Kulkarni P, Zhao W, Moon H, et al. Finding effective optimization phase sequences[J]. ACM SIGPLAN Notices, 2003, 38(7): 12-23.

[2]　Ansel J, Kamil S, Veeramachaneni K, et al. Opentuner: An extensible framework for program autotuning[C]//Proceedings of the 23rd international conference on Parallel architectures and compilation. 2014: 303-316.

[3]　Jantz M R, Kulkarni P A. Performance potential of optimization phase selection during dynamic JIT compilation[C]//Proceedings of the 9th ACM SIGPLAN/SIGOPS international conference on Virtual execution environments. 2013: 131-142.

[4]　Lokuciejewski P, Plazar S, Falk H, et al. Multi-objective exploration of compiler optimizations for real-time systems[C]//2010 13th IEEE

International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing. IEEE, 2010: 115-122.

[5]　Lokuciejewski P, Plazar S, Falk H, et al. Approximating Pareto optimal compiler optimization sequences—a trade‐off between WCET, ACET and code size[J]. Software: Practice and Experience, 2011, 41(12): 1437-1458.

[6]　Cavazos J, Fursin G, Agakov F, et al. Rapidly selecting good compiler optimizations using performance counters[C]//International Symposium on Code Generation and Optimization (CGO'07). IEEE, 2007: 185-197.

[7]　Ashouri A H, Bignoli A, Palermo G, et al. Micomp: Mitigating the compiler phase-ordering problem using optimization sub-sequences and machine learning[J]. ACM Transactions on Architecture and Code Optimization (TACO), 2017, 14(3): 1-28.

[8]　Fursin G, Kashnikov Y, Memon A W, et al. Milepost gcc: Machine learning enabled self-tuning compiler[J]. International journal of parallel programming, 2011, 39(3): 296-327.

[9]　Martins L G A, Nobre R, Cardoso J M P, et al. Clustering-based selection for the exploration of compiler optimization sequences[J]. ACM Transactions on Architecture and Code Optimization (TACO), 2016, 13(1): 1-28.

[10]　Ashouri A H, Mariani G, Palermo G, et al. Cobayn: Compiler autotuning framework using bayesian networks[J]. ACM Transactions on Architecture and Code Optimization (TACO), 2016, 13(2): 1-25.

[11]　Foleiss J H, da Silva A F, Ruiz L B. The effect of combining compiler optimizations on code size[C]//2011 30th International Conference of the Chilean Computer Science Society. IEEE, 2011: 187-194.

[12]　Plotnikov D, Melnik D, Vardanyan M, et al. An Automatic tool for tuning compiler optimizations[C]//Ninth International Conference on Computer Science and Information Technologies Revised Selected Papers. IEEE, 2013: 1-7.

[13]　Luk C K, Cohn R, Muth R, et al. Pin: building customized program analysis tools with dynamic instrumentation[J]. Acm sigplan notices, 2005, 40(6): 190-200.

[14]　Shen V R L. Novel Code Plagiarism Detection Based on Abstract Syntax Tree and Fuzzy Petri Nets[J]. International Journal of Engineering Education, 2019, 1(1).

[15]　Hassen M, Chan P K. Scalable function call graph-based malware classification[C]//Proceedings of the Seventh ACM on Conference on Data and Application Security and Privacy. 2017: 239-248.

[16]　Weyerhaeuser C, Mindnich T, Baeumges D, et al. Augmented query optimization by data flow graph model optimizer: U.S. Patent 10,241,961[P]. 2019-3-26.

[17]　Li, Yujia, Tarlow, Daniel, Brockschmidt, Marc, et al. Gated Graph Sequence Neural Networks[J]. Computer Science, 2015.

[18]　Guthaus M R , Ringenberg J S , Ernst D , et al. MiBench: A free, commercially representative embedded benchmark suite[C]// Workload Characterization, 2001. WWC-4. 2001 IEEE International Workshop on. IEEE, 2002.

[19]　Sridharan V, Kaeli D R. Eliminating microarchitectural dependency from architectural vulnerability[C]//2009 IEEE 15th International Symposium on High Performance Computer Architecture. IEEE, 2009: 117-128.

[20]　Kingma D P, Ba J. Adam: A method for stochastic optimization[J]. arXiv preprint arXiv:1412.6980, 2014.

[21]　Glorot X, Bengio Y. Understanding the difficulty of training deep feedforward neural networks[C]//Proceedings of the thirteenth international conference on artificial intelligence and statistics. 2010: 249-256.

[22]　Dyer C, Ballesteros M, Ling W, et al. Transition-based dependency parsing with stack long short-term memory[J]. arXiv preprint arXiv:1505.08075, 2015.