An Intelligent Scheduling System for Large-Scale Online Judging

En Zhang, Fan Wu, and Xuesong Lu(✉)

School of Data Science and Engineering,
East China Normal University, Shanghai, China
{zhangen,wufan_01}@stu.ecnu.edu.cn, xslu@dase.ecnu.ednu.cn

Abstract. Online judge (OJ) systems have been widely used for programming skill evaluation in various fields, including programming education, programming competition and talent recruitment. Existing OJ systems put the codes into a judge queue according to the order of user submission, and use the judge server to evaluate the correctness of the codes in turn. With the surge in the number of code submissions, this scheduling method causes the rapid increase of average response time for judge requests, resulting in a decline in user experience. To alleviate the problem, we develop an intelligent scheduling system, which consists of two modules. In the first module, we employ a deep representation learning model to predict the running time of the codes in the judge queue; in the second module, the judge queue is divided into fixed-size windows. The codes in each window are sorted according to their predicted running time in ascending order, and are scheduled to the judge server using the shortest job first algorithm. The experimental results show that, 1) the constructed prediction model predicts the running time of the codes accurately; 2) compared with the scheduling algorithm of existing OJ systems, the proposed scheduling algorithm can effectively reduce the average response time for large-scale online judging. Furthermore, by varying the code running time distribution and window size in the judge queue, we demonstrate the performance improvements of the proposed intelligent scheduling system under different settings, compared with the existing systems.

Keywords: online judge · intelligent scheduling · running time prediction · shortest job first · deep representation learning model

1 Introduction

Online Judge (OJ) [26] systems automatically assess the correctness of codes by running them with the pre-defined use cases and comparing the output with the standard results. Because of the convenience, OJ systems have been widely used in programming education [7], programming competitions [27], and talent recruitment[1,2]. For example, the China Computer Federation (CCF) uses an

[1] https://www.hackerrank.com/.
[2] https://www.qualified.io/.

W. Hong and G. Kanaparan (Eds.): ICCSE 2023, CCIS 2023, pp. 265–279, 2024.
https://doi.org/10.1007/978-981-97-0730-0_24

OJ system to organize a series of programming exams for "Software Professional Certification", usually with more than 10,000 examinees in a single exam [25]. For another example, on Chinese University MOOC[3], there might be up to 100,000 students who take the same programming course and practice programming in the built-in OJ system at the same time. In a typical OJ system, the codes submitted by users are inserted into a queue according to the submission order, and the judge server runs the codes in the queue based on the "First Come First Serve (FCFS)" scheduling algorithm, determining their correctness and returning the results to the corresponding users. As the amount of users and the number of code submissions grow, this scheduling method causes the rapid increase of the average response time for judging requests, i.e., the time between a code submission and the return of the evaluation results, thereby seriously affecting users' programming experience. Figure 1 shows the experimental results of average response time according to the FCFS scheduling algorithm, where the average running time of the experimented codes is only 96(±84) ms. However, when the number of codes in the queue grows from 100 to 1,000, the average response time of the judging request rapidly increases from less than 5 s to more than 80 s.

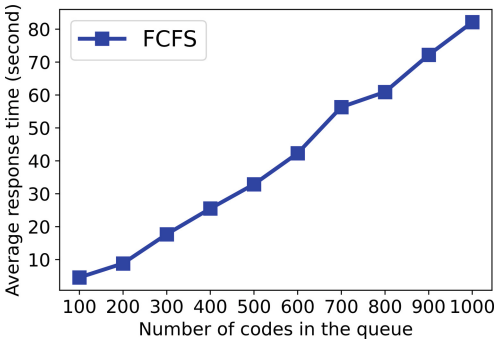

Fig. 1. Average response time vs. number of codes in the queue

In order to improve user experience, especially to reduce the anxiety of waiting for assessment results during competitions or exams, how to reduce the average response time in large-scale online judging scenarios is worth investigating. Some related works have proposed to build scalable OJ systems from a system architecture perspective to alleviate the load pressure caused by a single system. For example, Wang et al. [25] develop the MetaOJ system, which reduces the response time under high loads by building a distributed OJ system. Nerantzis et al. [19] develop the MI-OPJ system based on the micro-service architecture, which utilize Kubernetes to orchestrate the container services and realize the elastic scaling of the services of the OJ system.

[3] https://www.icourse163.org/.

Unlike the above work, we investigate how to improve the responsiveness of a single system in the face of large-scale judging loads, and the proposed method can be easily integrated into any existing OJ system, including those based on the distributed and micro-service architectures mentioned above. Specifically, we build an intelligent scheduling system that reduces the average response time of judging requests by prioritizing the judgements of codes in the queue with a shorter running time. The entire scheduling system is divided into two modules. First, in order to predict the running time of the codes in the queue, we employ a deep representation learning model to predict the code running time. In particular, we select three code representation models, i.e., graph2vec [18], ASTNN [29], and CodeBERT [8], to construct the prediction model and evaluate the performance. Through comparison, the ASTNN-based model is finally selected to predict the code running time. Second, we modify the FCFS scheduling algorithm to "Shortest Job First (SJF)" scheduling algorithm to prioritize the judgments of codes with a shorter predicted running time. In order to prevent the codes with a long running time from being unable to be judged for a long period of time, we divide the code queue into fixed-size windows, and in each window, the codes are sorted in ascending order according to their predicted running time and judged using the SJF scheduling algorithm. At the window level, the FCFS scheduling algorithm is still used to ensure that the codes with a long running time can be judged in the windows where they are inserted.

It is worth mentioning that in traditional CPU architectures, predicting code running time incurs considerable additional overhead, which greatly diminishes the judging response improvement brought by the above scheduling system. Fortunately, today's CPU-GPU heterogeneous architecture makes this overhead negligible: predicting a code's running time using GPUs is much faster than judging its correctness in the judge server, which is typically run on CPUs. As a result, except for the prediction and sorting time of the codes in the first window of the judge queue, which needs to be factored into the response time, the rest of the codes can complete the running time prediction and sorting during the waiting time for judgement, thus having no impact on the response time. The experimental results show that the model constructed in this paper can accurately predict the running time of the code, and the proposed SJF scheduling algorithm can effectively reduce the average response time of the judging requests. In addition, we vary the distribution of code running time in the dataset, and demonstrate how the proposed intelligent scheduling system reduces the average response time compared to the FCFS scheduling under different distributions. Finally, we vary the size of the window in the judge queue and demonstrate how the proposed system reduces the average response time under different combinations of queue length and window size.

In summary, we contribute in this work on the following three aspects:

- We propose to improve the responsiveness of a stand-alone OJ system for large-scale online judging by changing the scheduling algorithm of the judge server from FCFS to window-based SJF. To the best of our knowledge, this paper is the first to propose such an approach.

- In order to adopt the SJF scheduling algorithm, we construct a deep learning model to predict the running time of the codes, as a basis of the SJF scheduling algorithm. The experimental results show that the constructed model can accurately predict the code running time.
- We conduct experiments to demonstrate that the proposed intelligent scheduling system can effectively reduce the average response time of an OJ system under large-scale online judging. We vary the code running time distribution and the queue/window size to demonstrate the performance improvement over the traditional scheduling system under different settings.

2 Related Work

The intelligent scheduling system proposed in this paper contains two main parts: code running time prediction and code judging based on SJF scheduling algorithm. The former part belongs to the research category of running time prediction and the latter part belongs to the category of improving OJ systems. This section presents the related work in the two areas respectively.

2.1 Code Running Time Prediction

Predicting code running time has important applications in both industry and education. In industry, accurate prediction of code running time ensures that specific tasks in real-time systems do not exceed a preset time, guaranteeing system stability. In education, predicting code running time can give students real-time feedback and motivate them to write more efficient code.

Puschner and Koza [21] propose to split a complete code into different components, such as sequential, looping, branching, etc., and estimate the maximum running time of each component, and finally add them together to get the estimation of the maximum running time of the complete code. Park [20] combines the static analysis based on simple timing diagrams and the dynamic analysis based on execution paths to obtain tighter upper bounds on the running times of sequential programs. Benz and Bringmann [4] point out that accurate static code analysis relies heavily on loop boundaries, control flow constraints, and other program flow facts that need to be labeled by the user, and is difficult to apply in real-time systems. They improve code running time prediction using a scene-aware analysis based on the mode of operation, application-related program flow facts, and image resolution.

Another research area closely related to code running time prediction is code complexity prediction. Although the goals of the two prediction problems are slightly different, the way the code features are extracted in the proposed methods can be cross-referenced. For example, Gulwani et al. [10] propose both control flow refinement and progress invariant methods and combine them to estimate the complexity of codes with nested and multi-path loop structures. Chatterjee et al. [5,6] use sorting functions and linear programming to compute worst-case upper bounds for non-polynomials in codes. There is also a body of work that

focuses on the problem of predicting code complexity in recursive functions. For example, Albert et al. [1] use "evaluation trees" to mine recursive relationships and derive complexity bounds for subsumption and sorting based on the mining results. Ishimwe et al. [14] use pattern matching to learn recursive relations in recursive programs, and represent asymptotic complexity bounds for recursive programs by obtaining closed-form schemes. With the public availability of huge amounts of code data and the widespread use of neural networks, deep learning-based approaches have emerged. For example, Sikka et al. [22] propose a deep learning-based code complexity prediction method, which uses graph2vec [18] to represent the code, and constructs a multi-classification model to predict the complexity of the code.

Due to the significant differences in the structural features of different codes, traditional prediction methods do not generalize well. Therefore, in this paper, we collect a large amount of codes and train a deep learning model to predict the code running time.

2.2 Improving OJ Systems

As OJ systems are widely used, researchers propose various improvement methods to cope with the application of OJ systems in different scenarios. For example, in order to adapt the OJ system to educational scenarios, Sun et al. [23] design the YOJ platform for classroom teaching scenarios by focusing on the curriculum, and develop modules such as course management, problem discussion, function evaluation, offline assignments, and online tests to facilitate teaching and learning. Francisco and Ambrosio [9] develop an automatic question grouping system based on the difficulty classification of questions to assist in the design of assignments and tests, with the goal of improving the ease of use of OJ systems. They also develop a student reporting system based on behavioral data to provide feedback to students on their learning performance. Hou et al. [13] utilize the error reporting information from compilation tools and judging servers to give feedback to users, in order to improve their programming training and testing efficiency. Wang et al. [25] build a stand-alone OJ system to alleviate the service response latency problem caused by large-scale cross-region online programming testing, by building a distributed OJ system and deploying it on the cloud. In order to cope with the load pressure caused by large-scale online training during the Covid-19 epidemic, Nerantzis et al. [19] follow the micro-service architecture to reconstruct an OJ system, which can be massively scaled using software such as Kubernetes, Apache Kafka, and NextJS. Other improvement works of OJ systems include [28,30] and so on.

Among them, the works in [25] and [19] have the closet goals to our work, that is, improving the responsiveness of the OJ system to deal with large-scale online judging by improving the back-end structure. The difference is that these two works propose distributed and micro-service architectures, respectively, to rebuild the existing stand-alone OJ system, while we try to improve the system responsiveness by improving the intrinsic judging scheduling mechanism in existing OJ systems. The approach proposed in this paper is orthogonal to the

approaches in [25] and [19], which is more lightweight and can be flexibly adapted to any existing OJ system.

3 The Intelligent Scheduling System for Online Judging

The overall architecture of the intelligent scheduling system proposed in this paper is shown in Fig. 2. The code submitted by OJ users is first put into an judging queue according to the submission order, and the system divides the queue into fixed-size windows. In each window, the system predicts the running times of the codes, and sorts the codes in ascending order according to the predicted running time. Subsequently, the system combines all the windows into a queue again following the original window order, and judges the codes in the queue sequentially. The whole system is divided into two key modules: code running time prediction and judging scheduling based on running time prediction.

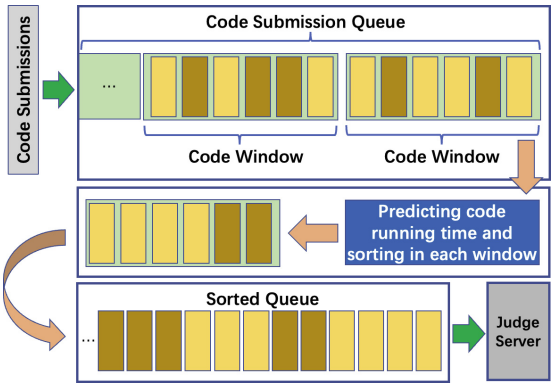

Fig. 2. The overall architecture of the proposed intelligent scheduling system

3.1 Code Running Time Prediction Based on Deep Representation Models

In recent years, code representation models based on deep learning have been applied to various tasks in the field of software engineering, including program classification [3], clone detection [2], code repair [12], etc., and have achieved the performance far beyond that of traditional algorithms. Therefore, in this paper we also predict the running time of code using deep representation learning models, since accurate prediction is very important to the subsequent scheduling algorithm.

We select three representative deep code representation learning models. The first model is graph2vec [18], which has been used to build models for predicting code complexity [22], and thus is closely related to our work. Graph2vec borrows the idea from word2vec [17] and doc2vec [16] to train graph representations by

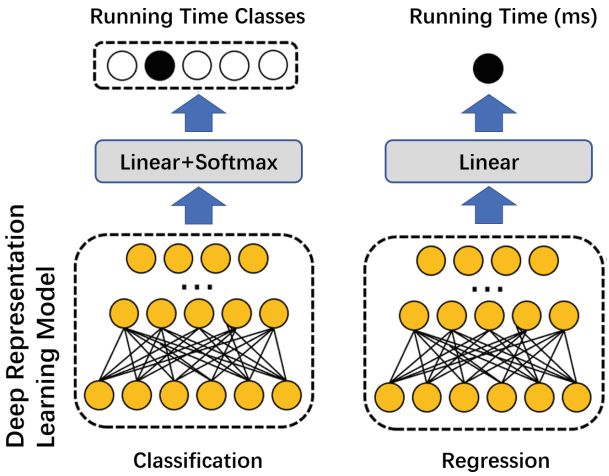

Fig. 3. The classification and regression model for evaluating the predictive performance of code representation learning models.

maximizing the probability of occurrence of the associated subgraphs of each node in the graph. When applied to code representation learning in this paper, we first convert the code into Abstract Syntax Tree (AST) intermediate representation, and consider the AST as a graph. Then we input the ASTs into Graph Attention Networks (GAT) [24] and train the code representation by applying the graph2vec algorithm. The second model is ASTNN [29], which performs prominently well among small-scale code representation models [11]. ASTNN splits the AST of a code into a set of statement subtrees, each of which corresponds to a complete statement, and then inputs the encoded sequence of subtrees into the bi-directional GRU, and finally performs max pooling over the hidden states of all time steps to obtain the representation of the entire code. The third model is CodeBERT [8], which is the first programming language pre-training model with strong code representation capability. The input of CodeBERT is a piece of code text and its natural language comments. Two pre-training tasks are designed to learn the code representation, which are masked token prediction and replaced token detection. After training, the encoding of the [CLS] token can be used to represent the input code. In order to evaluate the performance of the above three representation learning models, we construct both classification and regression models to predict the code running time, as shown in Fig. 3. In the classification task, we categorize the code running time into 5 classes, which are 0–50 ms, 50–100 ms, 100–200 ms, 200–500 ms, and 500–1000 ms, according to the distribution of actual code running time in the dataset. We add a linear layer with the Softmax activation on top of the code representation model to predict the classes and employ cross-entropy as the loss function. In the regression task, we similarly add a linear layer on top of the representation model, outputting a single value, which is finally converted into a value between 0–1000, representing

the predicted running time in milliseconds. The regression model uses the mean square error as the loss function.

3.2 Code Judging Scheduling Based on Running Time Prediction

Based on the above evaluation tasks, we select the best deep model to predict the code running time in preparation for judging scheduling. Specifically, we first predict the running time of each code in the queue. In traditional architectures, predicting code running time, whether using symbol-based or machine-learning-based approaches, incurs considerable additional overhead, which affects the judging response time. However, in the GPU-CPU heterogeneous architecture, we can use a GPU to run deep models to efficiently and accurately predict the code running time and use a CPU to run the judging service of the OJ system, without interfering with each other. In Sect. 4, we will see that predicting code running time using a GPU is much faster than judging the code in the OJ's judge server. As such, the prediction overhead does almost not increase the response time of the code judging.

Algorithm 1: Average response time for judging code using the SJF scheduling.

Input: Code queue Q formed according to user submission time, window size w
Output: Average response time for judging codes in Q: rt_{avg}.

```
divide Q into disjoint windows of size w;
foreach window W in Q do
    foreach code c in window do
        prt = predictRunTime(c) /* Predicting the running time prt of code c
        using a GPU */
    end
    sort the codes in W according to their prt in the ascending order
    RT_w = judgeAndCalRspTime(W) /* Feed W to the judge server and
    calculate the response time for each code */
end
calculate rt_avg using RT_w of all the windows in Q
return rt_avg
```

To implement the SJF scheduling algorithm, the codes in the queue need to be sorted in ascending order of predicted running time from shortest to longest and then fed to the judge server. A straightforward approach is to sort the entire queue, which may minimize the average response time. However, this approach makes the codes with long predicted running time unable to be judged for a quite long time, which affects the programming experience of the corresponding users. Therefore, we adopt a window-based SJF scheduling approach. Specifically, we divide the entire queue into windows of fixed size, sort the codes according to the predicted running time and use the SJF scheduling inside each window. On

the other hand, different windows still maintain the order in the initial queue, and are sent to the judge server in turn, i.e., the FCFS scheduling is still used at the window level. Finally, we obtain the judging response time of each code and compute the average time for the entire queue. The complete procedure for calculating the average judging response time is shown in Algorithm 1.

It is worth noting that although Algorithm 1 is a serial pseudo-code written according to the judging process, in practice, the GPU predicting the code running time and the CPU running the judging service work in parallel.

4 Performance Evaluation

In this section, we empirically evaluate the proposed intelligent scheduling system. First, we evaluate the accuracy of the models in predicting the code running time. Then, we evaluate the effectiveness of the SJF scheduling algorithm based on the predicted running time for reducing the average response time. Our code is written in Python and PyTorch, and all experiments are conducted on a machine with 4 cpu-cores and 16 GB memory, equipped with a Tesla V100.

4.1 The Dataset

The code dataset used for the experiments is collected from 242 programming exercises of our OJ system, and a total of 63,355 C codes are collected. The running time distribution of these codes on the experimental machine is shown in Table 1.

Table 1. The distribution of the actual running time of the code in the data set.

Running Time	Number of Codes
0–50 ms	35854
50–100 ms	5977
100–200 ms	5611
200–500 ms	5550
500–1000 ms	10363
Total	63355

We can see that more than half of the code running times are smaller than 50 ms, and only about 16% of the running times are larger than 500 ms. We divide the entire dataset into training set, validation set and test set with proportion 6:2:2, and train the deep learning models to predict the code running time.

In the code judging scheduling experiment, we randomly select N codes from the test set to form the judging queue, and then slice the queue into fixed-size windows. Within each window, we use the ASTNN-based model to predict the code running time, and sort the codes in ascending order according to the predicted results. Finally, the codes in each window are fed into the judge server in turn.

4.2 Evaluation Metrics

In the running time prediction experiment, we treat the prediction task as a classification and regression task, respectively, and use Accuracy and Mean Absolute Error (MAE) as the evaluation metrics. In the classification task, the classes are the five classes in Table 1. In the regression task, the models directly predict the running time of the code.

In the code judging scheduling experiment, we compute the average response time of the codes in the judging queue. We randomly assign a submission time to each code and ensure that all codes in a queue are submitted within one second. The response time for each code is the time that the judge server returns the result subtracting the submission time. Finally, the average of the response times of all codes in the queue is calculated.

4.3 Hyperparameter Settings

When encoding the codes, the lengths of both the input embedding vector and the output token vector are set to 128. The length of the hidden layer vector for both graph2vec and ASTNN is set to 256, and the number of heads in the graph attention module of graph2vec is set to 4. The batch size for training is 64, and the maximum epoch is set to 20. We use adaMax [15] as the optimizer, and set the learning rate to 0.002. We use the default settings of CodeBERT.

In the judging scheduling experiments, the window size for dividing the queue is fixed to 50.

4.4 The Results

Code Running Time Prediction. Table 2 demonstrates the performance of the three running time prediction models. Table 3 demonstrates the performance in different running time classes.

From the two tables, we observe that ASTNN achieves the best performance in both the classification and regression task. ASTNN extracts from the AST the statement subtrees of the corresponding independent components in the code, which may make it capture better the execution structure of the code, and thus be more accurate at predicting the running time. Graph2vec focuses on the subgraph structure associated with each node, which does not necessarily represent a complete piece of code when applied to ASTs, making it worse at capturing the

Table 2. Predictive performance comparison of the three code representation models

code representation model	Accuracy	MAE
graph2vec[8]	0.872	20.64
ASTNN[9]	**0.946**	**15.26**
CodeBERT[10]	0.923	18.71

Table 3. Predictive performance comparison of the three code representation models for different classes of running time (ms)

Models	Accuracy					MAE				
	≤50	50–100	100–200	200–500	>500	≤50	50–100	100–200	200–500	>500
graph2vec	0.884	0.803	0.852	0.824	0.912	7.41	15.70	18.10	31.88	55.69
ASTNN	**0.961**	**0.917**	**0.926**	0.896	**0.956**	**3.21**	**10.91**	**15.73**	**24.82**	**38.04**
CodeBERT	0.953	0.899	0.867	**0.916**	0.909	4.46	13.91	22.73	30.64	42.26

execution structure of the code. CodeBERT does not use C codes in pre-training and does not learn structural features from intermediate representations such as ASTs and control flow graphs. Therefore it performs a bit worse than ASTNN. In order to achieve higher accuracy, in the future, pre-trained models that incorporate structural features such as GraphCodeBERT can be considered to train the prediction model [26].

Code Judging Scheduling. The ASTNN-based model is the most accurate among the three prediction models, so we use it to predict the running time of the code in the judging scheduling experiment. Based on the prediction results, we sort the codes in each window and prioritize the codes with shorter predicted running time. We refer to this scheduling algorithm as "SJF based on predicted running time". For comparison, we additionally implement "SJF based on cyclomatic complexity". Cyclomatic complexity is commonly used to measure the static complexity of a code and represents the number of linearly independent paths in the code, which is usually calculated as the number of closed paths in the control flow graph. The SJF algorithm based on cyclomatic complexity prioritizes codes with lower complexity for judging. Finally, in order to obtain the upper bound on the performance of the SJF scheduling algorithm, we sort and judge the codes based on their actual running time. We refer to this algorithm as "SJF based on actual running time". For each algorithm, we vary the number of codes in the judging queue from 100 to 1000 with increments 100, and compute the corresponding average response time of code judging.

Figure 4 shows the results. First, we can see that compared to FCFS scheduling, both SJF based on predicted running time and SJF based on cyclomatic complexity scheduling algorithms reduce the average response time. With the increase of the number of codes in the queue, the average response time is reduced more. This proves that the SJF scheduling algorithm can effectively improve the judging efficiency for large-scale online judging. Second, SJF based on predicted running time can reduce more average response time than SJF based on circle complexity, which indicates that the code running time is more suitable for the SJF scheduling algorithm and the constructed model can predict the running time accurately. Finally, SJF based on actual running time further reduces the average response time than SJF based on predicted running time, indicating

that the scheduling performance of the current system can be further promoted by improving the accuracy of code running time prediction.

It is worth mentioning that in our experiments we can predict the running times of 30 codes per second using a Tesla V100 GPU, i.e., predicting the running time of the codes in a window of size 50 takes a bit more than 1 s. On the other hand, judging the codes in the window takes about 5 s. In other words, predict and sorting the codes in each window is much faster than judging them in the judge server. The overhead of code running time prediction does almost not increase the average judging response time when using a GPU as a standalone prediction module.

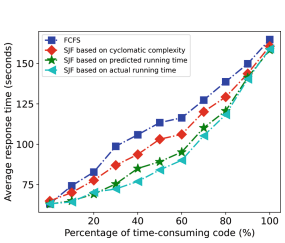

Fig. 4. The average response time of the four scheduling algorithms when the queue length increases.

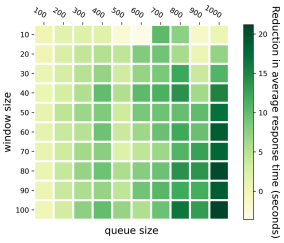

Fig. 5. Average response time for different distributions of code running time

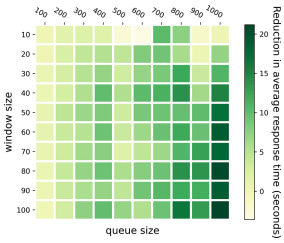

Fig. 6. The reduction in average response time under different combinations of queue length and window size, SJF vs. FCFS.

Impact of Code Running Time Distribution. The SJF scheduling algorithm is effective because the running times of the codes in the queue are quite different. In this section, we vary the distribution of code running times in the queue and observe the impact of the SJF scheduling algorithm on the average response time. Based on the collected dataset, we define codes with running time below 500 ms as "non-time-consuming code" and codes with running time above 500 ms as "time-consuming code". We increase the percentage of time-consuming codes from 0% to 100% with increments 10%, and randomly select 1,000 codes from the test set under each percentage control to calculate the average response time of code judging. Figure 5 shows the results. We can see that when the percentage of time-consuming codes is small or large, the SJF scheduling has a small reduction on the average response time over the FCFS scheduling, because the running times of all codes are more converged. The SJF scheduling algorithm based on predicted running times reduces the average response time the most when there are 50% time-consuming codes. This is instructive for designing code judging queues in OJ systems: one can group codes with different complexities into a queue, and use the scheduling system proposed in this paper to improve the average response time of the whole OJ system.

The Impact of Window Size in the Queue. In order to avoid the situation that codes with long running time are not able to be judged for a long time, we propose to divide the code queue into windows of fixed size, and use the SJF scheduling algorithm for the codes in each window, while different windows still follow the FCFS scheduling algorithm. Therefore, the window size has an impact on the overall performance of the scheduling system. We vary the code queue length from 100 to 1000 with increments 100, and for each queue length, we vary the window size from 10 to 100 with increments 10. Therefore, we obtain a 10×10 matrix, where each element of the matrix represents the average response time reduction of the SJF algorithm based on predicted running time over the FCFS algorithm, for the corresponding combination of queue length and window size. We visualize this matrix using a heat map. The more we reduce the average response time, the darker the color of the corresponding element in the matrix.

Figure 6 illustrates the experimental results. We can observe that the matrix gets progressively darker from the top left to the bottom right, i.e., overall the longer the queue and the larger the window, the more the average response time is reduced by using the SJF scheduling algorithm. The results are as expected, again demonstrating the accuracy of the running time prediction and the effectiveness of the SJF scheduling algorithm. In real systems, the length of the judging queue is usually determined by the system cache, and the administrator can dynamically adjust the window size according to the actual judging load, in order to meet the dynamic balance between the average response time of the codes with a long running time and the average response time of all codes.

5 Conclusion

In this paper, we address the problem of rapidly growing judging response time of OJ systems for large-scale online judging. We propose to reduce the average response time by improving the scheduling mechanism of the judging service, and construct a corresponding scheduling system. The system consists of two modules: the prediction module predicts the running times of the codes using a deep model, and the scheduling module judges the codes using a window-based SJF scheduling algorithm according to the predicted running time. Experimental results demonstrate that by accurately predicting the code running time, the scheduling algorithm proposed in this paper can effectively reduce the average judging response time of OJ systems. Importantly, the constructed intelligent scheduling system can be integrated into any existing OJ system.

In the future, we will continue to improve the scheduling system on two aspects. First, the experimental results prove that the response time using the predicted running time scheduling is still much larger than the response time using the actual running time scheduling. Therefore, we will investigate how to further improve the accuracy of code running time prediction. Second, we currently utilize a GPU for real-time prediction of code running time. For servers not equipped with a GPU, the benefit of intelligent scheduling will be affected by the prediction time. Therefore, we consider to predict and store the running

times of a huge amount of codes offline. For online judging, we will design a retrieval algorithm to obtain the predicted running time of each submitted code, so as to improve the prediction efficiency on CPU-only machines.

Acknowledgement. This work is supported by the grant from the National Natural Science Foundation of China (Grant No. 62277017).

References

1. Albert, E., Arenas, P., Genaim, S., Puebla, G.: Automatic inference of upper bounds for recurrence relations in cost analysis. In: Alpuente, M., Vidal, G. (eds.) SAS 2008. LNCS, vol. 5079, pp. 221–237. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-69166-2_15
2. Alon, U., Brody, S., Levy, O., Yahav, E.: Code2Seq: generating sequences from structured representations of code. arXiv preprint arXiv:1808.01400 (2018)
3. Alon, U., Zilberstein, M., Levy, O., Yahav, E.: Code2Vec: learning distributed representations of code. Proc. ACM Program. Lang. **3**(POPL), 1–29 (2019)
4. Benz, J., Bringmann, O.: Scenario-aware program specialization for timing predictability. ACM Trans. Archit. Code Optim. (TACO) **18**(4), 1–26 (2021)
5. Chatterjee, K., Fu, H., Goharshady, A.K.: Termination analysis of probabilistic programs through Positivstellensatz's. In: Chaudhuri, S., Farzan, A. (eds.) CAV 2016. LNCS, vol. 9779, pp. 3–22. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-41528-4_1
6. Chatterjee, K., Fu, H., Goharshady, A.K.: Non-polynomial worst-case analysis of recursive programs. ACM Trans. Program. Lang. Syst. (TOPLAS) **41**(4), 1–52 (2019)
7. Dong, Yu., Hou, J., Lu, X.: An intelligent online judge system for programming training. In: Nah, Y., Cui, B., Lee, S.-W., Yu, J.X., Moon, Y.-S., Whang, S.E. (eds.) DASFAA 2020. LNCS, vol. 12114, pp. 785–789. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-59419-0_57
8. Feng, Z., et al.: CodebERT: a pre-trained model for programming and natural languages. arXiv preprint arXiv:2002.08155 (2020)
9. Francisco, R.E., Ambrosio, A.P.: Mining an online judge system to support introductory computer programming teaching. In: EDM (Workshops). Citeseer (2015)
10. Gulwani, S., Jain, S., Koskinen, E.: Control-flow refinement and progress invariants for bound analysis. ACM SIGPLAN Not. **44**(6), 375–385 (2009)
11. Han, S., Wang, D., Li, W., Lu, X.: A comparison of code embeddings and beyond. arXiv preprint arXiv:2109.07173 (2021)
12. Han, S., Wang, Y., Lu, X.: Errorclr: Semantic error classification, localization and repair for introductory programming assignments. In: Proceedings of the 46th International ACM SIGIR Conference on Research and Development in Information Retrieval, pp. 1345–1354 (2023)
13. Hou, D., Ma, J., Yin, L.: A framework of online judge systems for assessing programming skills. Int. J. Des. Anal. Tools Integr. Circuits Syst. **8**(1), 45–46 (2019)
14. Ishimwe, D., Nguyen, K.H., Nguyen, T.V.: DynAplex: analyzing program complexity using dynamically inferred recurrence relations. Proc. ACM Program. Lang. **5**(OOPSLA), 1–23 (2021)
15. Kingma, D.P., Ba, J.: Adam: a method for stochastic optimization. arXiv preprint arXiv:1412.6980 (2014)

16. Le, Q., Mikolov, T.: Distributed representations of sentences and documents. In: International Conference on Machine Learning, pp. 1188–1196. PMLR (2014)
17. Mikolov, T., Chen, K., Corrado, G., Dean, J.: Efficient estimation of word representations in vector space. arXiv preprint arXiv:1301.3781 (2013)
18. Narayanan, A., Chandramohan, M., Venkatesan, R., Chen, L., Liu, Y., Jaiswal, S.: Graph2Vec: learning distributed representations of graphs. arxiv 2017. arXiv preprint arXiv:1707.05005
19. Nerantzis, O.R., Tselios, A., Karakasidis, A.: Mi-OPJ: a microservices-based online programming judge. In: 2021 IEEE International Conference on Big Data (Big Data), pp. 5969–5971. IEEE (2021)
20. Park, C.Y.: Predicting program execution times by analyzing static and dynamic program paths. Real-Time Syst. **5**(1), 31–62 (1993)
21. Puschner, P., Koza, Ch.: Calculating the maximum execution time of real-time programs. Real-Time Syst. **1**(2), 159–176 (1989)
22. Sikka, J., Satya, K., Kumar, Y., Uppal, S., Shah, R.R., Zimmermann, R.: Learning based methods for code runtime complexity prediction. In: Jose, J.M., et al. (eds.) ECIR 2020. LNCS, vol. 12035, pp. 313–325. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-45439-5_21
23. Sun, H., Li, B., Jiao, M.: YoJ: an online judge system designed for programming courses. In: 2014 9th International Conference on Computer Science and Education, pp. 812–816. IEEE (2014)
24. Veličković, P., Cucurull, G., Casanova, A., Romero, A., Lio, P., Bengio, Y.: Graph attention networks. arXiv preprint arXiv:1710.10903 (2017)
25. Wang, M., Han, W., Chen, W.: Metaoj: a massive distributed online judge system. Tsinghua Sci. Technol. **26**(4), 548–557 (2021)
26. Wasik, S., Antczak, M., Badura, J., Laskowski, A., Sternal, T.: A survey on online judge systems and their applications. ACM Comput. Surv. (CSUR) **51**(1), 1–34 (2018)
27. Watanobe, Y., Rahman, Md.M., Matsumoto, T., Rage, U.K., Ravikumar, P.: Online judge system: requirements, architecture, and experiences. Int. J. Softw. Eng. Knowl. Eng. **32**(06), 917–946 (2022)
28. Xia, M., et al.: PeerLens: peer-inspired interactive learning path planning in online question pool. In: Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems, pp. 1–12 (2019)
29. Zhang, J., Wang, X., Zhang, H., Sun, H., Wang, K., Liu, X.: A novel neural source code representation based on abstract syntax tree. In: 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE), pp. 783–794. IEEE (2019)
30. Zhou, W., Pan, Y., Zhou, Y., Sun, G.: The framework of a new online judge system for programming education. In: Proceedings of ACM Turing Celebration Conference-china, pp. 9–14 (2018)