Hindawi
Journal of Sensors
Volume 2023, Article ID 9826740, 1 page
https://doi.org/10.1155/2023/9826740

Retraction

Retracted: Software Engineering Code Workshop Based on B-RRT*FND Algorithm for Deep Program Understanding Perspective

Journal of Sensors

Received 19 December 2023; Accepted 19 December 2023; Published 20 December 2023

This article has been retracted by Hindawi following an investigation undertaken by the publisher [1]. This investigation has uncovered evidence of one or more of the following indicators of systematic manipulation of the publication process:

(1) Discrepancies in scope
(2) Discrepancies in the description of the research reported
(3) Discrepancies between the availability of data and the research described
(4) Inappropriate citations
(5) Incoherent, meaningless and/or irrelevant content included in the article
(6) Manipulated or compromised peer review

The presence of these indicators undermines our confidence in the integrity of the article's content and we cannot, therefore, vouch for its reliability. Please note that this notice is intended solely to alert readers that the content of this article is unreliable. We have not investigated whether authors were aware of or involved in the systematic manipulation of the publication process.

Wiley and Hindawi regrets that the usual quality checks did not identify these issues before publication and have since put additional measures in place to safeguard research integrity.

We wish to credit our own Research Integrity and Research Publishing teams and anonymous and named external researchers and research integrity experts for contributing to this investigation.

The corresponding author, as the representative of all authors, has been given the opportunity to register their agreement or disagreement to this retraction. We have kept a record of any response received.

References

[1] A. Xu, "Software Engineering Code Workshop Based on B-RRT*FND Algorithm for Deep Program Understanding Perspective," *Journal of Sensors*, vol. 2022, Article ID 1564178, 11 pages, 2022.

Hindawi
Journal of Sensors
Volume 2022, Article ID 1564178, 11 pages
https://doi.org/10.1155/2022/1564178

Research Article

Software Engineering Code Workshop Based on B-RRT*FND Algorithm for Deep Program Understanding Perspective

Aiqiao Xu

College of Science and Technology, Ningbo University, Ningbo, Zhejiang 315300, China

Correspondence should be addressed to Aiqiao Xu; fm@bbc.edu.cn

Received 20 August 2022; Accepted 13 September 2022; Published 26 September 2022

Academic Editor: Yaxiang Fan

Developers will perform a lot of search behaviors when facing daily work tasks, searching for reusable code fragments, solutions to specific problems, algorithm designs, software documentation, and software tools from public repositories (including open source communities and forum blogs) or private repositories (internal software repositories, source code platforms, communities, etc.) to make full use of existing software development resources and experiences. This paper first takes a deep programmatic understanding view of the software development process. In this paper, we first define the software engineering code search task from the perspective of deep program understanding. Secondly, this paper summarizes two research paradigms of deep software engineering code search and composes the related research results. At the same time, this paper summarizes and organizes the common evaluation methods for software engineering code search tasks. Finally, the results of this paper are combined with an outlook on future research.

1. Introduction

Developers face a large number of search behaviors in their daily work tasks, searching for reusable code fragments, solutions to specific problems, algorithm designs, software documentation, and software tools from public repositories (including open source communities and forum blogs) or private repositories (internal software repositories, source code platforms, communities, etc.) to make full use of existing software development resources and experiences and to improve the reusability of software, while reducing development costs and improving development efficiency. A study conducted in 1997 showed that code search has become the most common activity in software development activities [1]. The study showed that developers would construct an average of 12 query statements per workday to search for problems encountered. It can be seen that code search is gaining importance in software development activities. A scholar at Peking University has conducted a literature summary of work related to code search tools. The authors established a classification system to categorize code search tools in terms of code repository organization, query texts, search models, and evaluation methods [2]. Code search can be classified from search forms, such as code search based on natural language query, code clone detection, code search based on test cases, code search oriented to defect detection, and application programming interface search. The common point is to locate code files or fragments that meet user requirements from user requirements, combining information retrieval, program understanding, and machine learning techniques [3]. The framework of code understanding based on deep neural networks is shown in Figure 1.

Among them, the code search based on natural language query is closer to the actual needs of developers and is the hot spot in the current code search research. The code search task is similar to the traditional ad hoc task, but the code search task also has many difficulties, which are introduced in the next section.

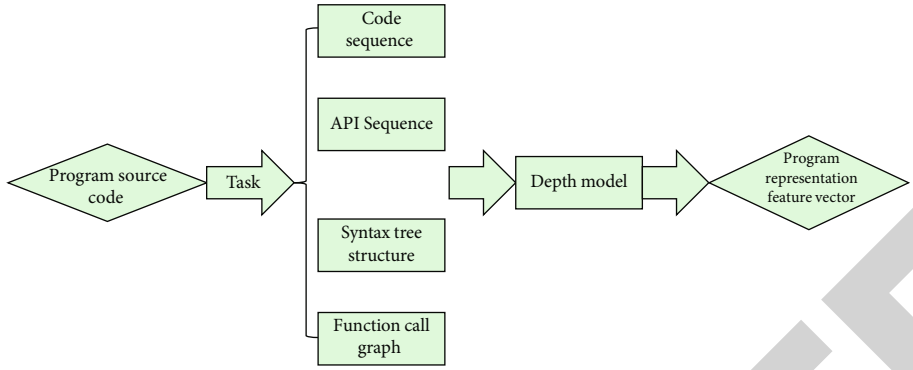

Figure 1: Deep neural network-based code understanding framework.

2. Research Background

The code search based on natural language query is closer to the actual needs of developers and is the hot spot in the current code search research. The code search task is similar to the traditional ad hoc task, but the code search task also has the following difficulties.

2.1. Cross-Modal Matching Problem. In the code search task, the query and the document are in different modalities. The query is mainly expressed in natural language form, while the target document is a source code text with program language syntax constraints, and the cross-modal semantic gap problem needs to be considered when computing the match [4].

2.2. The Query Intent Understanding Problem. The intentions expressed by queries in code search tasks are more diverse. They include user development requirements, constructive requirements to achieve a specific technical route, and API requirements to use a specific interface. It is necessary to combine software engineering domain knowledge with semantic understanding models when constructing retrieval models to understand user query requirements [5].

2.3. Program Understanding Issues. The code search model first needs to build an understanding of the source code fragment of the retrieved object. It includes the code base syntax semantics, API functional information, code structure features, and code functional characteristics. Among them, identifiers and program structure are the core of building program understanding [6].

As can be seen, the main problem faced by current code search research is how to understand program functionality and query intent matching based on program understanding, i.e., joint modeling of natural language and code fragments based on program understanding. In this paper, we present a review of recent code search research progress from the perspective of program understanding.

3. Materials and Methods

3.1. Basic Theory

3.1.1. Program Understanding. Program understanding is a key activity in software engineering, and the importance of program understanding in software engineering was clarified by the NATO Conference on Software Engineering held in 1968. Software engineers rely on understanding of programs when performing tasks such as software reuse, maintenance, migration, and reverse engineering [7].

In a collaborative development scenario, developers need to understand the software architecture and interface design patterns to develop and implement software functions, while in software maintenance, maintainers need to understand the main functions and implementation methods of existing projects as a basis for further defect checking and repair. "Understanding" is essentially a mapping from the conceptual domain of the object of understanding (e.g., text and source code) to the conceptual domain with which the subject of understanding (human, model) is familiar by means of learning. The subject of program comprehension is the whole software system or part of it and is aimed at studying how it works [8]. Program understanding tasks include constructing models at each level of abstraction from code models to application domain problems, understanding software using domain knowledge and constructing cognitive models between software artifacts and usage scenarios, and judging their role and relationships with other components by reading source code. Its ultimate purpose is to support software maintenance, evolution, and reengineering [9].

Classified according to the implementation approach, program understanding can be divided into two main approaches: analysis-based and learning-based. Analysis-based program understanding approaches rely heavily on the analyst's personal knowledge and experience, and constructing a program understanding model is equivalent to manually constructing a set of relevant features [10]. Such approaches are often coupled with specific software systems, and the relevant rules of experience are more difficult to be transferred to other projects. Also, as the size of the software increases, the resources required for analysis will increase plus rise and efficiency will decrease.

3.1.2. Deep Program Understanding. The availability of large amounts of open source code on the Internet provides sufficient data for learning-based program understanding methods. Studies have shown that program source code has properties similar to natural language, and this naturalness provides a theoretical basis for combining statistical

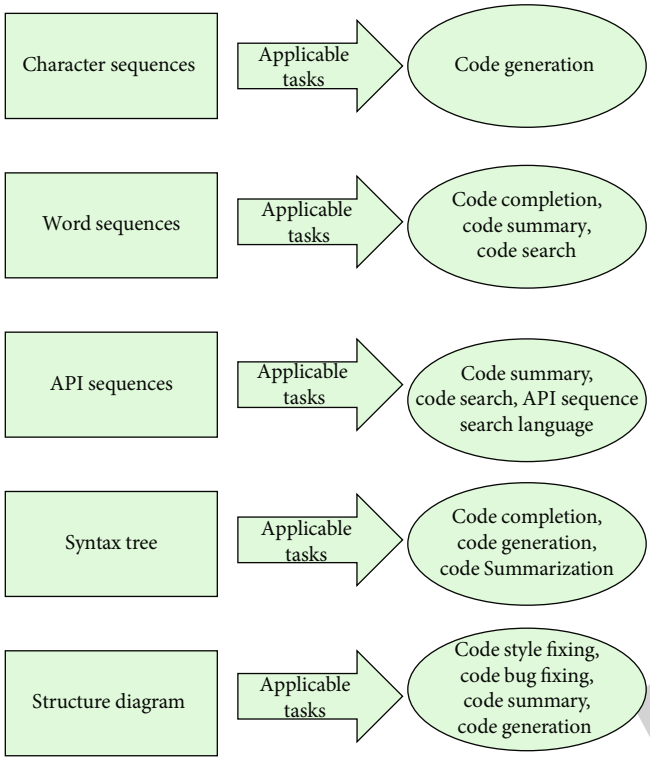

Figure 2: Software engineering code understanding representation situation.

models, especially deep models, for source code analysis and understanding [11]. Some scholars have conducted an in-depth analysis of the impact of source code localness on statistical modeling [12]. Domestic scholars have provided a detailed overview of the progress and challenges in deep source code modeling [13]. The software engineering code understanding representation is shown in Figure 2.

The deep learning-based program understanding framework, which consists of two main stages, first constructs the corresponding representation of source code text according to the different tasks and on this basis constructs the source code feature vectors by deep models and applies them to specific tasks [14].

Specifically, the program source code is first transformed into CharSequence, TokenSequence, APISequence, AbstractsyntaxTree (AST), and FunctionCallGraph (CFP) according to the task [15].

Based on this, source code different representations are used for specific tasks after being converted into feature vectors by neural networks. Source code has different representation form characteristics and applicable tasks. The modeling of programs based on word representations is similar to the modeling of natural language understanding models and is simple to implement and easy to migrate [16]. However, word-based code modeling approaches are often plagued by the OutOfVocabulary (OOV) problem due to the presence of developer-defined identifier cases [17]. Character sequence-based modeling can solve the OOV problem, but character symbol-based representations have difficulty learning word meanings and are therefore weaker in terms of representational power. The APIs contained in the code are often designed to be standardized, while the wording is more fixed, so learning for API sequences has good results in both code search and summarization tasks [18]. This is shown in Figure 3.

It can be observed that the use of word sequences alone as a representation of source code suffers from a lack of representational power, so combining multimodal approaches to model source code is gradually coming to the forefront of researchers' minds. Syntax tree-based code modeling can improve the problem of inadequate learning of program structure in sequence modeling. However, most of the existing studies use sequence sampling to obtain node sequences from syntax trees for representation, and the utilization of tree structure is not sufficient. Scholars at Northwestern Polytechnic University propose an improved method for fusing syntax tree information into code representation methods, where syntax trees are used as a parallel corpus of code sequences and modeled with natural language fragments based on alignment with source code sequences, improving the effectiveness of source code search tasks. The ability of graph neural networks to model natural language text structure has also inspired its exploration in code modeling. Source code data is easier to construct graph representations than natural language text, so program understanding models that incorporate graph structures have a better prospect.

As can be seen, deep code understanding research is the basis for many software engineering tasks, and the feature vectors from different representations can provide sufficient feature information for code search tasks. The next section defines the code search task from a deep program understanding perspective.

3.2. Research Methodology

*3.2.1. B-RRT~*FND Algorithm.* The RRT algorithm is one of the most representative sampling-based path planning algorithms, which can achieve the purpose of expanding the tree by randomly generating sampling points from the space, using the starting point as the root node, and growing from the nearest and collision-free sampling nodes in the direction of the link in a specified step, until the link between the leaf nodes and the target point is collision-free; then, the path planning is completed. The basic steps of the algorithm are as follows: step 1: given the map space M, the starting points Pstart and Pend. Step 2: Prand is obtained by random sampling in the space. Step 3: start from the nearest node Pnear point to grow toward Prand with step s and define the newly generated point as Pnew. Step 4: there is a certain probability that the sampled points will not grow along the randomly sampled point Prand but choose to grow directly to the end point Pend. Step 5: when the tree grows to Pend or the line between Pnew and Pend does not intersect with the obstacle, the feasible path σ is generated. Although the RRT algorithm has probabilistic completeness and can quickly obtain feasible solutions in space, its process of searching for solutions is blind, specifically because Prand is obtained by random sampling, which makes the tree grow

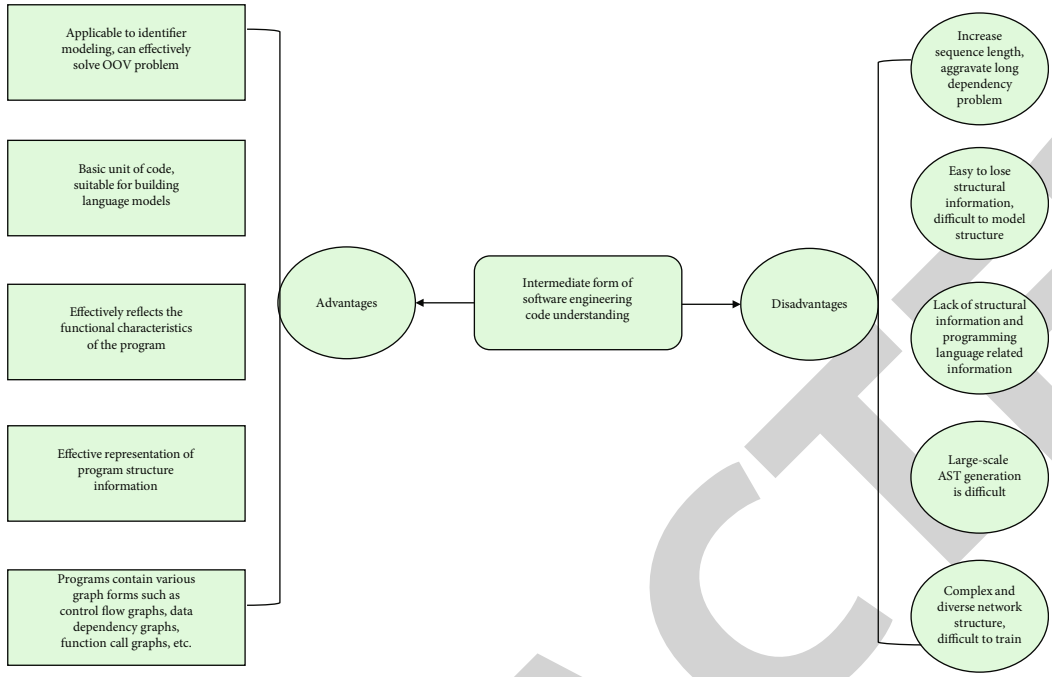

Figure 3: Intermediate form of code understanding.

in a random direction, and the algorithm lacks memorability of node expansion, leading to redundancy.

The RRT* algorithm belongs to the optimal algorithm, and the asymptotically optimal path can be obtained by obtaining enough sampling points in the update iteration. The algorithm adds two operations of parent node reselection and node reconnection: parent node reselection: with Pnew as the center of the circle, nodes in the region with R as the radius are selected as alternative parents, and the path cost with these nodes as parents is calculated, and the node that minimizes the path cost is selected as the new parent. The node with the lowest path cost is selected as the new parent and connected, and if there is a collision in the path, other alternate parents are selected. Node reconnection: take Pnew as the center of the circle, select a range with R as the radius, try to change the parent node of the node within the range to Pnew; if doing so can reduce the total path cost, then disconnect the node from its parent node and connect it to Pnew; if the connection has a collision, then give up this connection and continue to select other nodes within the range to try in turn. If the random sampling point Pn is taken exactly on the optimal path $\sigma*$ from Pstart to Pend in the space, the path cost is reduced due to Pn becoming the parent of its neighboring nodes after enough times of sampling. Since the algorithm scales up the search while the parent node reselection and node reconnection operations traverse all points in the tree, causing a huge memory burden, the search efficiency of the algorithm can be improved by fixing the number of nodes. The RRT*FN algorithm (Fixed-nodesRRT*) introduces the concept of maximum number of nodes based on the RRT* algorithm, which sets the maximum number of nodes allowed in the tree and randomly deletes a childless node except the end node when the number of nodes in the state space is greater than the preset node. The steps of RRT*FN algorithm are as follows: step 1: same as RRT algorithm step 1, step 2, and step 3. Step 2: Perform parent node reselection and node reconnection. Step 3: every time a Pnew is generated, check the number of nodes present in the space, if it is greater than the maximum number of nodes FixNodes randomly delete the childless leaf nodes that are not the last node of the path. Step 4: when the tree grows to Pend or Pnew and Pend's connection does not intersect with the obstacle, the feasible path σ is generated. Repeat step 1 to step 3 for asymptotic optimization of the path solution.

B-RRT*FND algorithm: inspired by the RRT*FN algorithm, in order to further improve the search efficiency of the RRT*FN algorithm and apply the algorithm to the dynamic environment, this paper proposes the B-RRT*FND algorithm (BidirectionalRRT*Fix-NodeDynamic), which improves the algorithm for the original RRT*FN algorithm by making the following improvements, combining the bidirectional greedy search strategy with RRT*FN algorithm is combined to further speed up the planning speed of the RRT*FN algorithm. It also makes use of dynamic update and path repair to enable the algorithm to be applied to the case of unknown and moving obstacles.

Greedy bidirectional search: the improved algorithm combines a bidirectional greedy search strategy with the RRT*FN algorithm to solve the problem of blindness in the growth of unilateral trees. The RRT*FN algorithm has no advantage over the RRT* algorithm in search speed, can only reduce the residual sampling points to avoid redundant growth, has the effect of limiting the tree size to improve the program running speed when the number of iterations is high leading to a large tree size, and is not related to obtaining path solutions quickly. With the addition of the two-way greedy search strategy, the algorithm

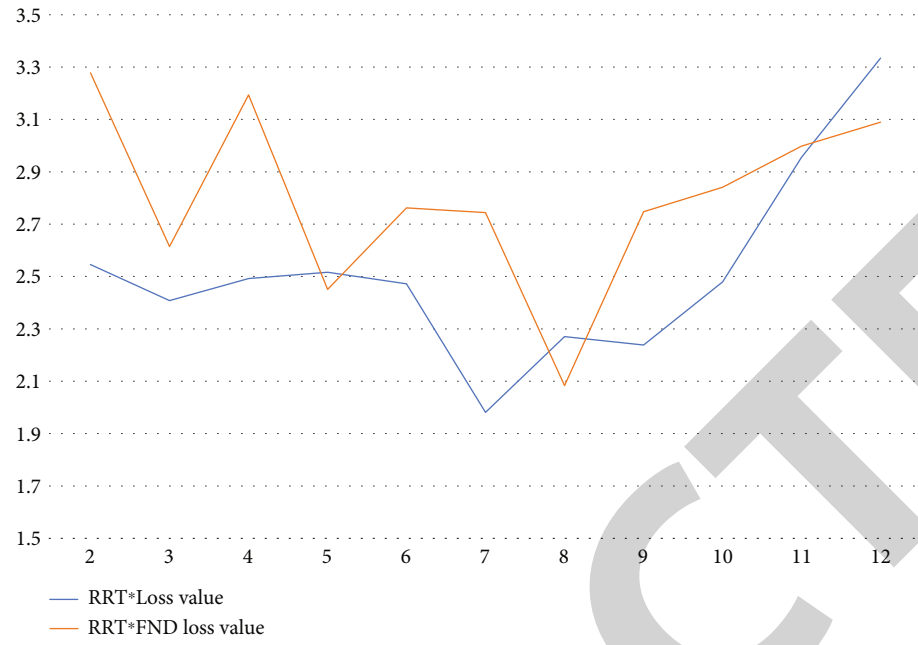

Figure 4: Comparison of RRT* loss value and RRT*FND loss value.

plans a path with more obvious directionality and can obtain an initial path more quickly. The greedy bidirectional search needs to establish two random search trees, Tree1 and Tree2, at the start and target points simultaneously, and the two trees grow toward each other, respectively, using the greedy strategy in the growth process. In this paper, the two-way greedy search strategy is combined with the RRT*FN algorithm to improve the defects of the traditional one-way random tree growth with poor purpose. Two trees are created from the starting point Pstart and the target point Pend, and each grows greedily toward the other. The green line is the starting tree and the blue line is the end tree. The black circles indicate obstacles. The nodes in the state space are generated in the order of subscripts, if the number of nodes in each tree is specified to be no more than 5.

In this paper, the RRT* loss value and RRT*FND loss value are low in each hidden layer node tree, and the total number of 11 nodes is between 2 and 12, and both loss values are above 2. The results show that the RRT* loss value and RRT*FND loss value are low, as shown in Figure 4.

In this paper, on each hidden layer node tree again, B-RRT ~ *FND loss value and RRT*FND loss value are higher, much higher than the number of nodes selected in this paper requires a total of 11 nodes from 2 to 12; both loss values are about 6 or more, and the results show that B-RRT ~ *FND loss value and RRT*FND loss value are higher, as shown in Figure 5.

In order to verify the effectiveness of the algorithm in this paper, the BRRT*FND algorithm is compared with other algorithms under three maps, and its performance indexes are evaluated. To facilitate the simulation analysis and to take into account the reasonableness, because the RRT class algorithm is based on random sampling, there is chance in the simulation process, and each measurement result may cause large differences due to different sampling point locations. In order to exclude the influence brought by chance, this paper conducts 50 independent experiments for each case and collates the results for comparison and analysis.

The two-way algorithm has certain advantages in fast solution finding, and the overall path cost is further reduced due to the improved algorithm using a two-way greedy search strategy. Since the B-RRT* algorithm, the RRT*FN algorithm and the B-RRT*FND algorithm are all optimal solution algorithms; the iterative process path solution parameters of the path solution are collated in this paper, as shown in Figure 6.

It can be seen that the path solution length and search time of B-RRT*FND algorithm are due to the other two algorithms. The length of the B-RRT* algorithm after 500 iterations is basically the same as that of the B-RRT*FND algorithm, but because it does not have a fixed number of nodes, the running time tends to increase concisely with the number of iterations, and after 1500 iterations, the running time of the improved algorithm and the number of iterations are approximately linear. At 3000 iterations, the B-RRT* algorithm takes 36.117 s, and the B-RRT*FND is 29.221 s. The algorithm performance comparison (II) is shown in Figure 7.

3.2.2. Main Evaluation Method. Set (Ω, ζ, P) is a conceptual space, and x is the set of all wandering variables on the space involved. The risk measure ρ is a mapping x from a x_ρ subset of $R$to the real numbers, denoted as $\rho : X \in x_\rho \leftrightarrow \rho(X) \in R$.

First define the g function called distortion function (distortionfunction) $g : [0, 1] \longrightarrow [0, 1]$ if it is a monotonically nondecreasing function and satisfies$g(0) = 0, g(1) = 1$.

Next, define the $\rho_g : x \longrightarrow R$ risk measure, often called distortionriskmesure if $\rho_g(X)$ satisfies

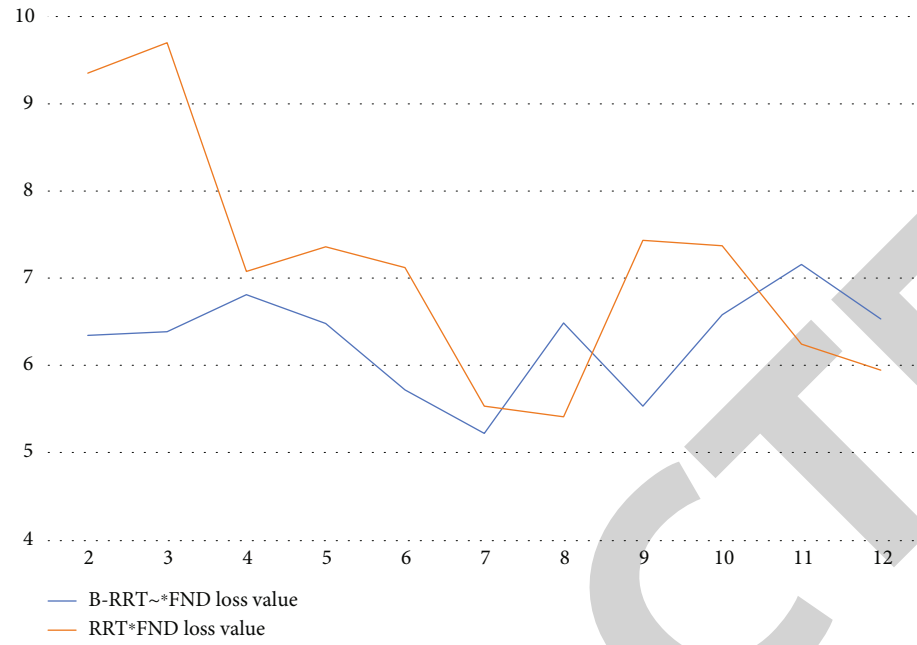

Figure 5: Comparison of B-RRT ~ *FND loss value and RRT*FND loss value.

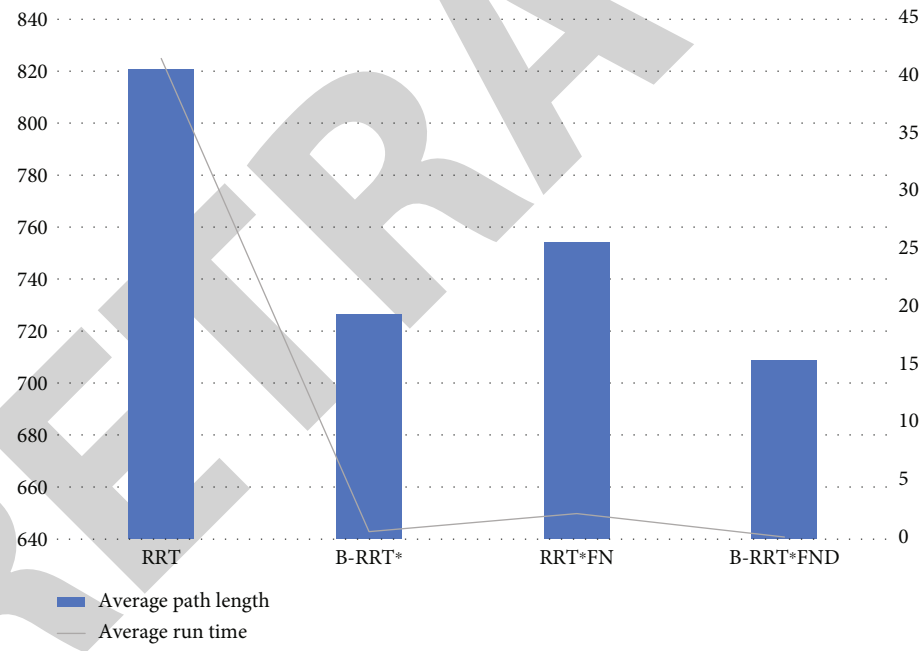

Figure 6: Algorithm performance comparison data (I).

$$\rho_g(X) \coloneqq \int_{-\infty}^{0} \lg\left(S_X(x)\right) - 1)dx + \int_{0}^{\infty} g(S_X(x)))dx, \quad X \in x, \tag{1}$$

where g is the distortion function, in which $S_X(x) = P(X \succ x)$, X is the tail distribution.

The X assumption is that the total risk faced $f : [0,\infty) \longrightarrow [0,\infty)$ by the insurer $f(X)$ is the partition function, representing the insurer transferring part of the risk faced by itself to the reinsurer. The reinsurers charge the insurer for the insurance premiums to supplement the risks they bear because they assume a portion of the insurer's risks. In this paper, we assume that the reinsurance cost criterion has the following form:

$$\mu_r(f(X)) = \int_0^{\infty} r\left(S_{f(x)}(x)\right)dx. \tag{2}$$

Without loss of generality, we assume r that it is not a function, that is, zero almost everywhere, and that the total risk an insurer has to face is the residual risk it will face plus

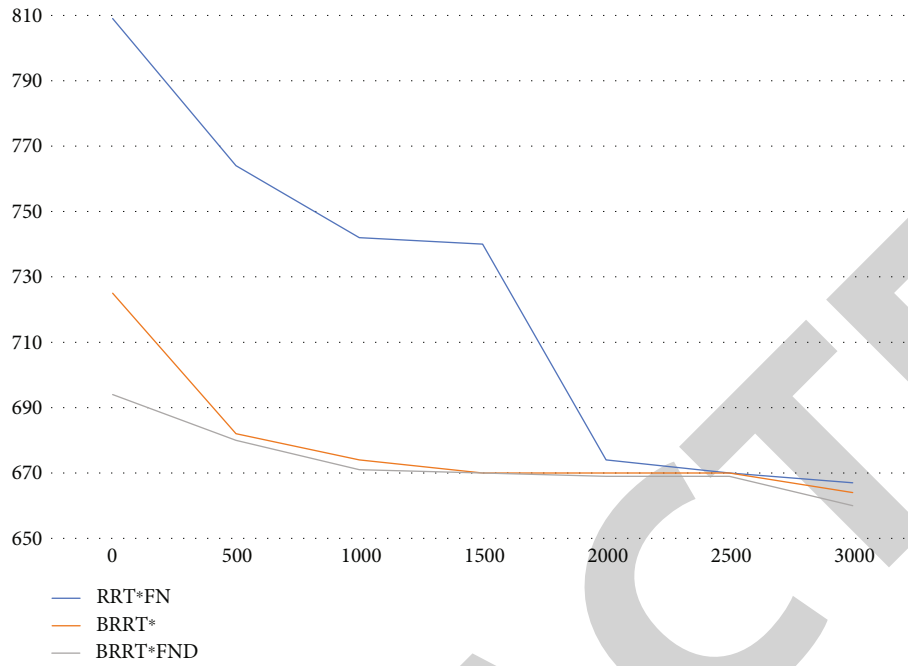

Figure 7: Performance comparison of the algorithms (II).

the cost required to transfer the risk. Expressed in terms of the formula can be expressed as

$$T_f(X) = X - f(X) + \mu_r(f(X)). \quad (3)$$

The basic algorithm theory BP algorithm theory process mainly includes the process calculation of linear propagation of output signal deviation forward and backward and linear propagation of output signal error forward and backward and reverse which are two process calculation processes. That is, the signal error can be adjusted according to the two input directions from the actual input signal direction to the actual expected signal output, respectively, to calculate the signal output, from the direction of the real expected signal output and then to the real expected input direction of the two directions, respectively, to calculate the signal error to adjust the signal error weight range and error threshold. In the study of the propagation method after the forward superposition of the signal, the input node signal is mainly the node on the actual output of the signal after the inverse superposition through the role of the hidden layer, and the actual output node signal can be generated through the nonlinear transformation process [19]. If we find that the actual signal output node position does not coincide with the actual output node direction position of the actual input node expectation signal, the process of backward feed-back propagation method for signal error compensation will be easily generated. The principle of error input signal back propagation processing system is that the system will automatically back propagate its various output signals or error information values to each error input layer of the system through the hidden layer nodes layer by layer and will sequentially transfer its output error signal values to the nodes on each layer corresponding to all other layers of the system error input signal elements, with the system in each layer of the system nodes obtained. The output error input signal values obtained by the system at each layer node are used as the basis for its calculation to automatically adjust the weights among the system's error output signal elements [20].

Neural network is essentially a nonlinear predictive model and, as its name suggests, an algorithm that mimics the human and animal nervous systems for computation. It is based on imitating the neural network system of human and animal-like brains to perform calculations and then to process the content of each module. Neural network algorithm is a derivative of data mining technology, which is one of the types of data mining technology that can be used for big data mining, such as analysis, classification, aggregation, and other data mining functions. Its advantages and disadvantages are very clear; the first advantage is that it is extremely resistant to interference, and the second is that it is capable of deep learning and better memory in a nonlinear situation and can handle more complex situations. At the same time, it has two disadvantages. First, its computation and processing results are low-dimensional and cannot be adapted to a high-dimensional environment, so it has a hard-to-interpret nature. The second is that whether it is supervised or unsupervised learning, it requires a long learning time, and the data is collected using a more traditional neural network approach.

In this paper, we use fuzzy neural networks. This type of neural network (FNN for short) is first, a deep combination of fuzzy theory and neural network algorithms. In the process of data mining and information processing by neural network algorithms, fuzzy theory is incorporated to improve the mapping and the relevance of mathematical relationships. The efficiency of supervised learning and unsupervised

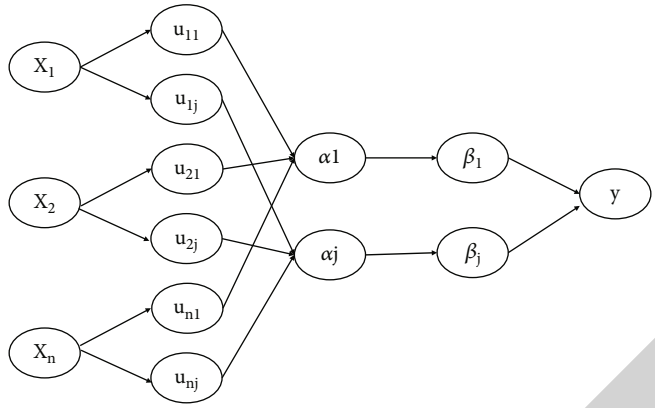

Figure 8: Fuzzy neural network.

learning is better improved. The algorithmic formulas of such neural networks and the related structural diagrams are more commonly used and common and can be found in general textbooks. This type of neural network is shown in the figure; it goes through five levels in the process of training and supervised and unsupervised learning; at the beginning of the two levels, as the level increases, the range of calculations required will be doubled, but as it enters the third level and enters the fourth level and enters the fifth level, the content of calculations will gradually decrease until it becomes one. Of course, this type of graph is first tested for dimensionality at this node in the input layer when the input is made. The specific value assumes that the dimension value is n and the node that needs to be input is n. Depending on the number of nodes needed, it is passed all the way to the layer of the dimensionality function and the related layer of further computed functions, as well as finally to the output layer. This type of fuzzy theory combined neural network has the same nature as the wavelet neural network and the neural network combined with the generalized theory, which both use the traditional gradient form of computation downward to calculate the centroid of the affiliation and the associated required width value and the final output value and the weights that we need. This is shown in Figure 8.

4. Results and Discussion

4.1. Software Engineering Code Search and Document Search. Code search is a cross study of information retrieval and program understanding, similar to document search techniques, and they have more commonality in various techniques such as query understanding, document indexing, and document sorting. However, from the perspective of the search object, due to the characteristics of source code itself, the understanding of code includes two aspects, i.e., the usage scenario of code and the principle of code algorithm. Taking "bubble sort" as the query, the results of document search and code search can be seen that the document search results often contain the original keywords and a series of concepts around the keywords and combinations. In the code search results, the focus is on the specific implementation of the algorithm and the correctness of the code. The core difference between the two lies in how the understanding of the document (code) is achieved. At the same time, matching code to user intent based on characteristics such as source code functional characteristics, application scenarios, and how the functionality is implemented is the biggest difference between the two search approaches. From a formal point of view, Q is the query text, and C is the source code fragment. As can be seen, the difficulty of the code search task is to establish an understanding of the source code fragment and, on this basis, to achieve a match between the user's query intent and the functional semantics of the code fragment. In traditional information retrieval model-based approaches, the code is usually treated as an ordinary natural language document, and the similarity between the user query and the code document is finally calculated by combining the natural language model with vectorization after simple word separation, deactivation, and stemming.

A research team in the UK proposes a method based on textual regularization for traceability between code and documents. Another foreign scholar introduces method name information into the retrieval model to enhance the performance of code retrieval results and combines API understanding to improve the code search system based on information retrieval, whose implementation is simple and straightforward and is a common research baseline. A scholar in Xi'an introduced the combination of topic model and Tfidf model into source code and query representation to improve the accuracy of query and code matching. An Italian research team recategorized code search features and reassigned feature weights according to different semantic categories to improve the retrieval effect. Domestic scholars proposed a code search method combined with ranking learning. Some other scholars further expand the code search feature system on this basis and combine it with ranking learning. Deep code search research then combines deep program understanding research to construct retrieval models. Some scholars have pioneered the problem of the semantic gap between source code and natural language. The query text and code fragments are obtained as feature representations by natural language models and program understanding models. After obtaining feature vectors, matching relations are constructed by deep models.

Natural language model part, word representation, word2vector model, glove model, and fasttext model are

commonly used for representation; contextual representation, Elmo model, Bert model, etc., are commonly used to model sentences directly to obtain representation.

In the part of matching model, domestic play scholars jointly studied the combination of ranking learning training target and depth model, i.e., neural information retrieval model. In addition, domestic scholars have summarized the loss function in the neural retrieval model, and methods such as contrast learning sorting (contrastive learning) and ternary sorting (triplet loss) have gradually received attention, combining reinforcement learning to improve query understanding and thus retrieval effect. In the section on procedural representation models, the UK research team has summarized and categorized them in more detail. Early research treated code fragments as ordinary text and combined with natural language modeling methods to model word sequences and API sequences; some subsequent work gradually paid attention to code structure properties and proposed code structure modeling methods combined with abstract syntax trees, and the stronger graph structure of source code makes recent research on source code modeling combined with graph neural networks gradually gaining attention.

4.2. Deep Software Engineering Code Search. This section compares current code search research progress from the perspective of deep program understanding models. In terms of paper selection basis, this paper uses code search and code retrieval as keywords for searching, covering software engineering, natural language processing, neural computing, and other fields. Finally, more than 40 papers related to deep code search were collected from related conferences, journals, and prepublication platform arXiv. Based on this, case study papers, system design papers, and research papers without clear evaluation metrics were excluded for this paper, leaving 27 papers for the final summary.

The analysis dimensions include source code representation, deep model structure, data set used for model evaluation, and evaluation metrics. The source code representation is planned from the perspective of deep program understanding, mainly including word sequence representation, API sequence representation, tree structure (mainly abstract syntax tree), and graph structure (function call graph, syntax tree subgraph, etc.). The structural aspects of deep models include convolutional networks (CNN), recurrent networks (LSTM), transformer, attention mechanisms, etc.

A Russian scholar has made an early exploration of deep code search, which firstly studied the search and generation of APIs and on this basis proposed the deep code search model Code NN, which laid the basic framework of deep code search model. In China, scholars in this field combined word vectors to model source code in an unsupervised way, and the code text was processed into word sequences, and then, the chapter representation was constructed by word vectors, and matched retrieval was realized on this basis. Later on, some scholars further studied the semantic gap between natural language and APIs in detail on this basis.

Starting from the work of Russian scholars, a series of attempts have been made to apply different deep model architectures to the code search task. Domestic scholars have initially tried to introduce attention mechanism into code matching computation by constructing multilayer attention networks using CNN to capture the deep semantics of source code. Another Japanese scholar applied twin networks to code search tasks to enhance the matching ability between queries and codes. Indian scholars separate keyword matching and syntactic pattern learning to propose an adaptive deep code search model with stronger generalization over a new codebase. Kunming scholars, on the other hand, applied adversarial learning to the training process to improve the matching between code and query text.

The pure sequence modeling approach is difficult to utilize the structural information of the source code, and domestic scholars are the first to use abstract syntax trees to informationally enhance the code representation. Then, the semantic dependencies in the source code are modeled by combining syntax trees on this basis. Finally, self-attentive networks are used for code search for the first time, on which the sequence information and structural information of source code are modeled uniformly in combination with self-attentive networks.

The graph is a prevalent structure in source code, and Yangling County scholars construct subgraphs from syntax trees to model the relationships between different nodes in the code. After that, the query and code feature vectors are mutually augmented using relational graph convolutional network modeling to finally improve the retrieval effect. Then, different representations of the source code are considered as multimodal tasks, and the code sequences, syntax trees, and graphs are fused to model the semantic representation of the code. The final result achieves SOTA effect, which confirms the practical effectiveness of fusing sequence and structural semantics on code understanding tasks.

Pre-trained models have also achieved success in various tasks in the field of natural language processing. Some researchers have also started to focus on the effect of source code pretraining. Domestic scholars combined with the contrast learning training paradigm to train Bert models on code text; foreign scholars focused on Python code for Bert model training and proposed the CuBert model. Based on this, domestic researchers added code structure to the training process and proposed GraphCodeBert, which combined with graph structure to enhance transformer's expressiveness when pretraining code. In contrast, foreign scholars have verified the effectiveness of the CodeBert and other models on specific code comprehension tasks by using a code search task with the training model. The results show that transformer-based pretrained language models also have good results on code comprehension tasks.

4.3. Dataset and Evaluation Methods

4.3.1. Evaluation Metrics. The evaluation metrics for the code search task are consistent with those for the information retrieval task and mainly consider two aspects of the results, namely, the relevance of the retrieved results and the ranking order of the relevant results. The precision P@K calculation method is relatively simple and measures

the proportion of the number of relevant documents in the retrieval results to the retrieval results, where K indicates the number of documents obtained in one retrieval process.

Recall R@K is also simple and measures the number of relevant documents in the search results as a proportion of the overall relevant documents, where K indicates the number of documents obtained in a single search. As can be seen, the precision and recall metrics measure the ratio of relevant documents in the top K search results. MeanReciprocalRank (MRR) introduces the order of relevant documents in the results for result evaluation. The average precision (AveragePrecision) combines precision and document order, where m denotes the total number of relevant documents in the current search results and N denotes the total number of search results.

The discounted cumulative gain (DCG) introduces the relevance rank into the evaluation. The normalized discounted cumulative gain (NDCG) metric is normalized using the DCG score of the best ranked result. In addition to the above metrics, there is also Frank evaluation describing the order of the first correlation result in the search list in the list.

4.3.2. Evaluation Dataset. In this paper, we comb through the code search research work from 2016 to the present and summarize the datasets used to evaluate the effectiveness of the model. In terms of selection criteria, the dataset must be publicly available, while at least two or more works have used the dataset to evaluate the model.

The code search commonly used evaluation datasets is more involved, of which a total of seven works are involved. In terms of dataset construction, the current work focuses on constructing large-scale code snippet-natural language text combinations as training data by automated extraction. The validation data is constructed based on automatic annotation combined with negative sampling methods. The model evaluation uses common code search questions as queries, and fine-grained annotation is performed on the constructed codebase.

According to the data annotation division, among them, CSN and ROSF data were strictly annotated with relevance levels; DCS, NCS, and CosBench first screened the natural language queries and later annotated relevant code snippets on the codebase.

According to the evaluation metrics, CNS and ROSF can be evaluated using NDCG; StaQC is mainly combined with text classification metrics; the remaining datasets are mostly evaluated using MRR, P@K, and Frank metrics.

According to programming languages, the DCS, NCS, ROSF, and CosBench datasets mainly contain Java programming language code data (covering Android). The CoNaLa data, on the other hand, combines Stackoverflow community data mining and manual annotation to build data containing Java and Python programming languages. The StaQC dataset mainly contains annotation results in both Python and SQL. The CodeSearchNet dataset contains Java, Python, Php, Ruby, Go, and JavaScript, covering the widest range of programming languages.

According to the task division, StaQC converts the code search task into a related document classification problem, so it can be studied in combination with text classification methods; the CSN and ROSF data have sufficient annotation level information and can be studied in combination with ranking learning methods; the code fragment-natural language combinations contained in the rest of the dataset can be used for code summarization tasks as well as code search studies based on summarization techniques.

4.3.3. Comparison of Results. In this section, based on the introduction of datasets and evaluation metrics, recent experimental work on code search is sorted out from the perspective of model effectiveness, focusing on deep model-based code search methods. The evaluation metrics then cover recall, precision, MRR, MAP, and NDCG. For statistics, the relevant literature baseline of the proposed dataset is bolded in the literature column accordingly. Some datasets, such as the StaQC dataset, were proposed using classification metrics as baseline, so they are not indicated here.

In this paper, the results are organized by dataset, related literature, and evaluation metrics. From a dataset perspective, the StaQC dataset is the most widely used and relatively influential. The experiments on the DCS dataset are relatively more adequate, and the different literature needles basically cover all evaluation metrics. From the perspective of evaluation metrics, MRR metrics are currently commonly used in code search model validation. And from the programming language perspective, the programming language that has been studied more is Java.

4.4. Software Engineering Code-Related Perspectives. Code search research is gradually gaining attention from the academic community. In this paper, recent progress in code search is reviewed from the perspective of deep program understanding, and the problem can be studied in the following aspects in the future: (1) stable and reproducible evaluation methods: most of the current studies do not open evaluation datasets, and the open datasets have problems such as inconsistent labeling. It is shown that the problems of dataset and open source code make the reproducibility of deep model results problematic. Future research should try to build consistent and clear datasets and evaluation methods and platforms to facilitate code search research. (2) In-depth study of program representation techniques: source code understanding and modeling is the key to the code search task. Most models use sequence modeling for feature extraction representation of source code, and a few works simply stitch and fuse tree and graph structure data to introduce code structure information. Combining graph neural networks for more in-depth structural modeling of code can be attempted in future research. (3) Multimodal source code modeling approach: the source code consists of identifiers and programming language-specific keywords. The programming language-specific keywords provide the structural information of the code, while the code identifiers provide more adequate natural language information. The two modal data can be combined in future research to model the structural semantics as well as the natural semantics of the code in a unified way, which can then be used for code

search tasks. (4) Code search research application issues: current code search research focuses on the problem of reordering based on deep code modeling methods. The code search tool research is more concerned with the collection, cleaning, and management of code data. Some data processing methods and effect optimization methods in code search research are not applicable between the actual application system. The search model can be designed from the practical application purpose in future research.

5. Conclusion

This paper first defines the software engineering code search task from a deep program understanding perspective. Secondly, it summarizes two research paradigms of deep software engineering code search and composes the related research results. At the same time, this paper summarizes and organizes the common evaluation methods for software engineering code search tasks. Finally, the results of this paper provide an outlook on future research.

Data Availability

The dataset is available upon request.

Conflicts of Interest

The author declares no conflicts of interest.

References

[1] M. Waqar, M. A. Zaman, M. Muzammal, and J. Kim, "Test suite prioritization based on optimization approach using reinforcement learning," *Applied Sciences*, vol. 12, no. 13, p. 6772, 2022.

[2] K. Li, Z. Cong, and Y. Xiaoxian, "Code comment generation based on graph neural network enhanced transformer model for code understanding in open-source software ecosystems," *Automated Software Engineering*, vol. 29, no. 2, 2022.

[3] S. H. Peter, B. Firas, S. Pasquale, and L. Philipp, "A systematic mapping study of source code representation for deep learning in software engineering," *IET Software*, vol. 16, no. 4, pp. 351–385, 2022.

[4] W. Li, K. Xianglong, W. Jiahui, and L. Bixin, "An incremental software architecture recovery technique driven by code changes," *Frontiers of Information Technology & Electronic Engineering*, vol. 23, no. 5, 2022.

[5] S. Randeep, B. K. Amit, and K. Ashok, "Improving software design by mitigating code smells," *International Journal of Software Innovation (IJSI)*, vol. 10, no. 1, pp. 1–21, 2022.

[6] R. S. J. José, "Design of open code software to downs and Steiner lateral cephalometric analysis with tracing landmarks," *Digital*, vol. 2, no. 2, pp. 120–142, 2022.

[7] K. Christian, F. Moritz, G. Lea, and S. Klaus, "Incremental software product line verification-a performance analysis with dead variable code," *Empirical Software Engineering*, vol. 27, no. 3, pp. 1–41, 2022.

[8] D. Zishuo, L. Heng, S. Weiyi, and C. T. H. Peter, "Can pretrained code embeddings improve model performance? Revisiting the use of code embeddings in software engineering tasks," *Empirical Software Engineering*, vol. 27, no. 3, pp. 1–38, 2022.

[9] K. Sašo, M. Aleksej, and H. Tjaša, "Software system comparison with semantic source code embeddings," *Empirical Software Engineering*, vol. 27, no. 3, 2022.

[10] R. R. Althar, A. Alahmadi, D. Samanta, M. Z. Khan, and A. H. Alahmadi, "Mathematical foundations based statistical modeling of software source code for software system evolution," *Mathematical biosciences and engineering*, vol. 19, no. 4, pp. 3701–3719, 2022.

[11] R. P. Ferreira, L. O. Vilarinho, and A. Scotti, "Development and implementation of a software for wire arc additive manufacturing preprocessing planning: trajectory planning and machine code generation," *Welding in the World*, vol. 66, no. 3, pp. 455–470, 2022.

[12] L. Na, Z. Haoyu, H. Zhihui, K. Guang, and D. Huadong, "Deep program representation learning analysis for program security," *Journal of Physics: Conference Series*, vol. 1971, no. 1, 2021.

[13] B. Wayne, "Open source software engineering the eclipse way," *Computer*, vol. 54, no. 6, pp. 59–63, 2021.

[14] M. U. R. A. K. A. M. I. Yukasa, T. S. U. N. O. D. A. Masateru, and N. A. K. A. M. U. R. A. Masahide, "Relationship between code reading speed and programmers' age," *IEICE Transactions on Information and Systems*, vol. E104.D, no. 1, pp. 121–125, 2021.

[15] K. Erik, H. Oto, D. Peter, L. Roman, and C. Jan, "PetriNet editor + PetriNet engine: new software tool for modelling and control of discrete event systems using Petri nets and code generation," *Applied Sciences-Basel*, vol. 10, no. 21, p. 7662, 2020.

[16] L. F. Sidou and E. M. Borges, "Teaching principal component analysis using a free and open source software program and exercises applying PCA to real-world examples," *Journal of Chemical Education*, vol. 97, no. 6, pp. 1666–1676, 2020.

[17] N. Fatima, S. Nazir, and S. Chuprat, "Knowledge sharing framework for modern code review to diminish software engineering waste," *International Journal of Advanced Computer Science and Applications (IJACSA)*, vol. 11, no. 6, 2020.

[18] C. Wijesiriwardana, A. Abeyratne, C. Samarage, B. Dahanayake, and P. Wimalaratne, "Secure software engineering: a knowledge modeling based approach for inferring association between source code and design artifacts," *International Journal of Advanced Computer Science and Applications (IJACSA)*, vol. 11, no. 12, 2020.

[19] S. Nazir, N. Fatima, and S. Chuprat, "Situational modern code review framework to support individual sustainability of software engineers," *International Journal of Advanced Computer Science and Applications (IJACSA)*, vol. 11, no. 6, 2020.

[20] N. Fatima, S. Nazir, and S. Chuprat, "Knowledge sharing factors for modern code review to minimize software engineering waste," *International Journal of Advanced Computer Science and Applications (IJACSA)*, vol. 11, no. 1, 2020.