Introduction to the algorithm of code similarity checking

The method used here is based on an abstract syntax tree (AST) to check code similarity. Specifically, the AST is parsed from the code source file, and then the two ASTs are compared how "similar" they are, and this indicator is used as an indicator of how similar the two code files are.

Key implementation technologies and some implementation details

In the program implementation, we used Clang 13.0.0 to generate our AST.
Since Clang lacks an interface for the Java language, a special processing method is used here to construct the AST tree.
Specifically, Runtime.getRuntime().exec(String cmd) interface is used here to run command line commands in Java language. At the same time, use clang's -ast-dump=json command line option to make the AST output in Json format.
So after receiving the Json string generated by Clang, the question now is how to parse the Json string by java. Here, the rich third-party library of java shows its advantages. The Jackson library [1] is used here as a parsing tool for Json strings. This tool can parse json in the form of a tree.
Due to the compilation characteristics of C++ programs, a large number of nodes in the AST tree we get now actually come from header files, and in fact, regardless of the test cpp file, in our scenario, the corresponding content of the header file is height Similar, so we need to exclude these nodes from the tree. Only keep the AST from the source file.
An assumption is introduced here: all #include statements are at the beginning of the source file, we think this is reasonable-indeed a part of the file (.inc file) is inserted somewhere in the code by #include, but at this time we think this is a part of the user's code .
In the Json generated by clang, the loc item records where the corresponding node was generated. Using this information, we can eliminate a large part of the original huge Json structure due to the header file.
In addition, the Json tree also retains attributes that have little to do with AST, such as the loc attribute, so we can also delete this information when processing the Json tree to retain more critical information.
So the question now is how do we evaluate the similarity of two ASTs, or how do we evaluate the similarity of the two tree structures. A simple idea is that, similar to two strings, we can define the degree of dissimilarity between the two strings by the edit distance and the length and ratio of the two strings. We can also achieve this goal through the edit distance of the two trees.
So to evaluate the tree edit distance here, I used the APTED algorithm [2][3], which is an upgrade of the RTED algorithm [4].

In order to evaluate the tree edit distance, a natural idea is to process it recursively. However, this means an unacceptable exponential time complexity. The APTED algorithm uses dynamic programming and has a polynomial level of time and space complexity. And APTED provides a Java third-party library [5]. We referenced it in the project.
In this way, we have the tree edit distance. This value can be used as an evaluation value of the similarity of the two codes.

Some issues

At present, in my tool, only Clang is used in the establishment of the program AST, which means that it can only better fight against the confusion at the symbol level, and is still relatively insensitive to the confusion at the control flow level. In addition, for a typical scenario: add a lot of dead codes into the user's code , "diluted" the repetitive code, it is completely impossible to fight for my tool. Because clang actually does not have the ability to statically analyze the program. So I imagined a better usage of this tool: before processing similar code, the source file was optimized and compiled and decompiled back to the cpp file to eliminate dead code.
But the problem is that the decompilation tool we imaged seems not exist. Then the other way is to consider optimization at the LLVM IR level, but again, the inverse conversion tool from LLVM IR to AST does not exist. That is to say, it is better to consider looking for similarities at the LLVM IR level. But this is obviously completely different from the current idea of this project(compare AST).
In short, it still seems difficult to try to fight dead code at the AST level, which can be solved by static analysis. A possible effective idea is to split the code to a smaller level (function level), turn the two files into two file groups to compare pair by pair, and look for highly matched code fragments.

Reference

[1]. https://github.com/FasterXML/jackson
[2]. M. Pawlik and N. Augsten. Tree edit distance: Robust and memory- efficient. Information Systems 56. 2016.
[3]. M. Pawlik and N. Augsten. Efficient Computation of the Tree Edit Distance. ACM Transactions on Database Systems (TODS) 40(1). 2015.
[4]. M. Pawlik and N. Augsten. RTED: A Robust Algorithm for the Tree Edit Distance. PVLDB 5(4). 2011.
[5]. https://github.com/DatabaseGroup/apted