Using Program Dependency Graphs for plagiarism detection in Python

Thomas Schaper (student), Ana Lucia Varbanescu (supervisor)
University of Amsterdam

ABSTRACT

Plagiarism in computer science education programs is a significant problem, requiring resilient, reliable, automated tools for efficient detection. Plagiarism detection tools based on Program Dependency Graphs (*PDG*) fulfill these requirements, but do not directly support all programming languages. For example, for Python, an increasingly popular programming languages in computer science education, traditional PDG-based methods do not work, as they create too many incorrect edges. In this work we propose the *PyDG* framework, the first solution for PDG-based plagiarism detection for Python programs. PyDG's approach is based on creating a slightly restricted Python language. Our empirical analysis demonstrates that PyDG successfully improves plagiarism detection by complementing existing tools.

KEYWORDS

Plagiarism detection, Python, Static analysis, Program dependency graph

INTRODUCTION

Plagiarism in computer science programs is a big problem. Over 70% of students admit to committing plagiarism at least once during their graduate program [11]. Automated tools are very important for detecting possible cases of plagiarism as the amount of pairs that need to be checked is large. However, as the reliance on these tools is well known, students can leverage the weaknesses of these tools to commit plagiarism without getting caught.

There are multiple specialized tools for detecting plagiarism in source code [9, 10]. Most of these tools are resilient against attacks such as identifier renaming and formatting changes. However, they are vulnerable to other attacks, such as code insertion and statement reordering [7]. A plagiarism-detection technique that is resilient against these attack is based on the Program Dependency Graph (*PDG*) of a program [7].

A PDG is a directed graph where the nodes are statements and the edges represent dependencies — both data and control ones. In fact, the PDG is the union of the edges present in a Control Dependency Graph (*CDG*) and a Data Dependency Graph (*DDG*) [4]. Given the advantages of PDG-based plagiarism detection, our goal is to enable their use for Python, a programming language quickly growing in popularity in various education programs.

However, creating PDGs for Python poses multiple problems. These problems mainly occur when creating the DDG, as types of variables cannot be statically determined and many techniques, such as static interprocedural analysis, are not possible in Python. This causes incorrect data dependency edges, which in turn causes all graphs for functions of equal length to look the same, resulting in many false positives for plagiarism detection.

In this work we propose *PyDG*, a framework that automatically generates PDGs suitable for Python plagiarism detection. Specifically, PyDG reduces the amount of incorrect edges by creating a restricted Python language in which certain assumptions about mutation and aliasing are always true. Our empirical analysis shows that PyDG is capable of generating PDGs with a low amount of incorrect data dependency edges for unrestricted Python source code. Furthermore, we show that, using these generated PDGs and a matching analysis as proposed by [7] we can analyze code submitted by students for plagiarism. Our results show PyDG adds value to the plagiarism detection process, as it complements state-of-the-art tools such as MOSS.

THE PYDG ARCHITECTURE

To detect plagiarism in Python using PDGs we created a processing pipeline. First we parse the Python source code into an Abstract Syntax Tree (*AST*), and perform pre-processing by simplifying the AST. This simplified AST is used to create a CDG and a DDG which are combined into a PDG. This PDG is then post-processed by pruning and used for the matching.

Pre-processing

There are expressions in Python which contain control flow jumps. As a PDG is created on statement level, these jumps are not visible in the resulting graph. To prevent this we transform a subset of these expressions: `list`, `set`, `dict` and generator comprehensions, and conditional expressions (also known as ternary operators). All generators are rewritten as a loop that produces a value in a temporary variable. A conditional expression is rewritten as an `if` statement, where a temporary variable is assigned in the then and in the `else` block.

PDG construction

There are two options for creating PDGs: using static analysis or using runtime information [3]. Both approaches pose challenges. To get runtime information the code needs to be run, creating possible security issues and requiring a standardize way of testing the submitted code needs to be created. When using runtime information the test code for creating the PDGs needs to cover all edge cases, as these edge cases could possibly drastically change the PDG.

PyDG uses static analysis and restricts the supported language to enable disambiguation (see A1 through A5). The key is to find the sweet spot for restrictions/assumptions that do not hurt programmers, and enable analysis. As local variables can be statically resolved, the challenge in building the PDG is finding all mutations of variables. To do this we first identify two classes of statements: (1) guaranteed mutation, where statements are guaranteed to mutate a variable (e.g. assignment of a variable), and (2) possible mutation (e.g. calling a method on a variable). As a data dependency from node A to node B indicates that the statement of node B contains a variable which would have an incorrect value if the statement of node A was executed after the statement of node B, possible mutations create data dependencies. As all false edges (i.e. edges present in the graph while there is no dependency between the statements in the program) are caused by the second class of statements, decreasing the amount of false edges can be achieved by reducing as much as possible the types of statements from the second class.

In our approach we remove all statements from the second class: we either move them to the first class, assuming that they always mutate, or we remove them completely, assuming that they never mutate. As the goal of the assumptions is to minimize the amount of false edges, a statement of the second class should only be moved to the first class if the chance that it mutates a variable is high. The result of this analysis is encoded into a set of 5 main assumptions, A1-A5, which should hold for most idiomatic Python code, and therefore the generated PDGs should be valid for most idiomatic Python code.

A1: Restricted aliasing With A1 we assume that aliasing of variables is restricted. When it is not possible to determine that variable `a` and `b` do not alias each other, mutating `a` might also mutate `b`. A1 assumes that aliasing can only occur from direct assignment in a function, e.g. `a = b`, but that functions always return new objects and that function arguments do not alias each other.

A2: Behavior of attributes Python objects have attributes, just like languages such as Java and C++. In these languages the semantics for getting, setting and deleting, if at all possible, attributes is defined. In Python however, getting, setting and deleting attributes can be done by special methods which can mutate the object in arbitrary ways. With A2 we assume that all attributes have the same properties as instance attributes, *simple attributes*. This means that overriding methods such as `__getattr__` and `__getattribute__` do not violate A2 if it follows this behavior.

A3: Operators With A3 we assume that these operator functions, like `__add__`, are pure. Binary and unary operators like `a + b` are *syntactic sugar* in Python for `a.__add__(b)`. A3 assumes that these methods do not mutate their arguments and that they produce a new object each time they are called.

A4: Functions and methods With A4 we assume that functions and methods do not mutate their parameters. It is possible for functions and methods in Python to mutate their parameters, as most objects in Python are mutable. A4 assumes functions to be pure, as global state is not modeled in the PDG, and methods always only mutate the object they are defined on.

A5: Exceptions The fifth assumption is that only blocks inside a `try` statement can raise exceptions. Modeling exceptions in a CDG can be done in different ways, but most methods depend on some form of checked exceptions [1, 2]. As Python has no checked exceptions, these techniques are not suitable. While creating a control dependency to the previous statement for each statement is technically correct this would result in a graph that is unsuitable for plagiarism checking, because slight alterations will drastically change the PDG. As suggested in [1], we chose to only model exceptions that are catched. We extended this technique to assume that only the block inside a `try` statement can raise exceptions, not the individual statements. The `raise` statement, which always raises an exception, is directly connected to the special exit node of a function, even if this happens in a `try` block as it is not always possible to statically determine which except clause will catch the raised exception.

Besides these main assumptions we need 6 extra assumptions for generator expressions, conditional expressions, comprehensions, global state, nested methods and augmented assignment.

Post-processing

In Python, it is normal to have *functionally-useless statements*. Functionally-useless statements are statements that do not have side effects and do not modify the control flow. An example of such a statement is the *docstring*, which is a string literal that documents a function, method, class or module. In the post processing phase these functionally-useless statements are removed. A node is considered functionally-useless if the sum of indegree and outdegree is less than two.

Matching

For each submission (from each student) every function in each file is converted to a PDG, as described in the previous sections, resulting in the set S_i of PDGs for submission i. The PDGs in S_i are filtered based on size, as proposed by [7] to remove all trivial functions. A pair of two submissions, i and j with $i \neq j$, are marked as possible plagiarism if there exists a graph G in S_i that is similar to a graph G' in S_j. A graph G is similar to graph G' if G is *γ-isomorphic* as defined in [7], i.e. if there exists such a graph $G_s \subseteq G$ that is subgraph isomorphic to G' and satisfies $\gamma \cdot |G| \leq |G_s|$. We find this subgraph G_s by finding the Maximum Common Subgraph (*MCS*) of G and G', as the MCS is the largest graph that is subgraph isomorphic to G and G'. Therefore if the MCS does not satisfy $\gamma \cdot |G| \leq |G_s|$, no other graph that is subgraph isomorphic to G and G' will. We have implemented finding the MCS using the described algorithm in [8].

Finding the MCS is a NP-complete problem [6], this may take a long time. Therefore, we limit the amount of time that we search for the MCS to a certain threshold which we call the *cutoff-time*.

EXPERIMENTS

We test the functionality of PyDG in two ways: (1) by determining the extent to which our assumptions A1-A5 are applicable to real code, and (2) by checking students' code on students' to determine if the generated PDGs can be used for plagiarism detection.

Code analysis

To determine the applicability A1-A5, we generated the PDGs for the back-end code of *CodeGrade*[1] at commit hash `5ea16a49`. CodeGrade is a blended learning application, partially written in Python, designed for programming education. The Python part of the program is around 9000 lines of code, divided over 385 functions. Only functions of at least 5 nodes and one data dependency edge were included in the analysis, of which we analyzed 151.

Table 1 presents the result of this analysis. Specifically we present the relative amount of incorrect edges in the generated PDGs, calculated as the ratio between the amount of incorrect edges by the total amount of data dependency edges. The results are very good: we observe a very small number of false edges — at most 12.5%. The average amount of incorrect edges across all node sizes is 6.9%.

We note that, while it is desirable to interpret the relative amount of incorrect edges by comparison to other tools/methods to generate Python PDGs, this comparison is currently impossible because no such tool is available.

Plagiarism detection

To assess the effectiveness of using the generated PDGs for plagiarism detection, we ran PyDG on an assignment given in the "Datastructuren en Algoritmen" course of the artificial intelligence BSc program. The latest submission of each student, 98 in total, was checked for plagiarism using PyDG and MOSS. For PyDG, we defined 15 nodes as the minimal size of a non-trivial PDG. We set our matching cutoff-time to 25 seconds, and $\gamma = 0.9$, as suggested by [7]. Matching cutoff-time is a trade-off between accuracy and execution time. For this experiment we set a budget or 1400 CPU hours, which meant 25s per pair ($201453 \times 25s = 5036325s \approx 1399h$).

The assignment had a fixed structure, and part of the code is provided. We therefore added an extra filtering step to minimize false positives. A function f of submission s is considered given or trivial if it is marked as possible plagiarism more than t times. The threshold t was set to 2 for this case study. This value was chosen based on intuition and can be easily changed.

MOSS outputs its results online in a sorted list. This list contains all possible matches, resulting in low quality matches at lower positions. For a fair comparison we decided to only consider the first ten matches MOSS outputs, as suggested by [5].

Our analysis focuses on the potential cases of plagiarism. A *match* is a pair of submissions that is flagged as similar by PyDG or MOSS. In this case study however we used a dataset where not all true positives are known, which means the ground truth is not known. To determine whether a match is a true or false positive, all matches were analyzed by hand and assigned a true or false positive label.

In this context, we define a *true positive* as a match which needs closer inspection by a human. A match that doesn't need further inspection (i.e. the code is *in fact* not similar) is considered a *false positive*. The *relative false negative rate* for a tool A is defined as the amount of false negative matches from A divided by the true positive matches from A. As the ground truth is not known it is not possible to determine the amount of *false negatives*. Therefore, we can only determine the minimal amount of false negatives for a tool A, i.e. the amount of unique true positives found by all tools except A that were not found by A.

Table 1: An analysis of the amount of incorrect data dependency edges in automatically generated PDGs from CodeGrade.

Amount of nodes	Incorrect edges
5 – 16	6.3%
17 – 28	7.5%
29 – 39	12.5%
40 – 51	8.5%
52 – 63	4.6%
Average	6.9%

[1]https://github.com/CodeGra-de/CodeGra.de

Table 2: The amount of true and false positives, minimal amount of false negatives and relative false positive rate for PyDG and MOSS.

True positives		False positives		Minimal false negatives		Relative false positive rate	
PyDG	MOSS	PyDG	MOSS	PyDG	MOSS	PyDG	MOSS
20	8	10	2	1	13	50%	25%

The analyzed results are presented in table 2. PyDG finds 12 more true positives than MOSS. The output of PyDG does contain more false positives than MOSS, and PyDG also has a higher relative false positive rate. Not all true positives from PyDG and from MOSS were the same, i.e. PyDG outputs true positives that MOSS doesn't output and vice versa. PyDG finds 13 true positives that MOSS does not find, MOSS finds one unique true positive.

CONCLUSION

Reliable plagiarism detection using automated tools is a necessity for many education programs. The increased use of Python in these programs has rendered some of the previously acceptable techniques, like PDG-based plagiarism detection, impossible to use. In this work, we proposed PyDG, the first framework able to use PDGs for plagiarism detection in Python.

In this work, we have shown that our novel approach, limiting the Python language using a limited set of restrictions, makes it possible to create PDGs for Python. We empirically demonstrated that these PDGs have a reasonably low amount of incorrect dependency edges compared to the ground truth constructed by hand. We have further demonstrated that it is possible to devise a methodology to detect possible cases of plagiarism that MOSS, a state-of-the-art plagiarism detection tool, does not detect, while also detecting most true positives from MOSS. Therefore, we have shown that it is feasible to use PDGs for plagiarism detection in Python code.

Based on these results, we believe PyDG-based plagiarism detection to be a useful addition to existing plagiarism detection tools. The generated graphs could also be used for other purposes where small errors in the graph are tolerable, e.g., for improving linting or the debugging experience in Python. We also suspect that our approach is scalable and could be used for other high level programming languages where no static type information is known.

We identify two directions of future work. They are short and long term. On the short term, better PDG validation for the graphs generated by PyDG, as well as better PDG matching algorithms are required to further validate the correctness and feasibility of the methods proposed in this work.

On the longer term, employing better analysis to further reduce the amount of incorrect dependency edges is also desirable. Although the current version of PyDG is capable of creating PDGs with a reasonably low amount of incorrect data dependency edges (at most 12.5%), the amount of incorrect edges could be further reduced.

ROLE OF THE STUDENT

This research was performed by Thomas Schaper under the supervision of Ana Lucia Varbanescu (in the period April-June 2018). The topic was proposed by the student. The design and implementation of the program, the choice and implementation of the matching algorithm, the design of the experiments and success metrics, the processing of the results and formulation of the conclusion were all done by the student.

REFERENCES

[1] M. Allen and S. Horwitz, *Slicing java programs that throw and catch exceptions*, ACM SIGPLAN Notices, 38 (2003), pp. 44–54.

[2] A. Amighi, P. de C. Gomes, D. Gurov, and M. Huisman, *Sound control-flow graph extraction for java programs with exceptions*, in Software Engineering and Formal Methods, Springer Berlin Heidelberg, 2012, pp. 33–47.

[3] Z. Chen, L. Chen, Y. Zhou, Z. Xu, W. C. Chu, and B. Xu, *Dynamic slicing of python programs*, in 2014 IEEE 38th Annual Computer Software and Applications Conference, 7 2014.

[4] J. Ferrante, K. J. Ottenstein, and J. D. Warren, *The program dependence graph and its use in optimization*, ACM Transactions on Programming Languages and Systems, 9 (1987), pp. 319–349.

[5] J. Hage, P. Rademaker, and N. van Vugt, *A comparison of plagiarism detection tools*, Utrecht University. Utrecht, The Netherlands, 28 (2010).

[6] J. Hartmanis, *Computers and intractability: a guide to the theory of np-completeness (michael r. garey and david s. johnson)*, SIAM Review, 24 (1982), pp. 90–91.

[7] C. Liu, C. Chen, J. Han, and P. S. Yu, *Gplag: detection of software plagiarism by program dependence graph analysis*, in Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining, ACM, August 2006, pp. 872–881.

[8] J. J. McGregor, *Backtrack search algorithms and the maximal common subgraph problem*, Software: Practice and Experience, 12 (1982), pp. 23–34.

[9] L. Prechelt, G. Malpohl, and M. Philippsen, *Finding plagiarisms among a set of programs with jplag*, J. UCS, 8 (2002), p. 1016.

[10] S. Schleimer, D. S. Wilkerson, and A. Aiken, *Winnowing: local algorithms for document fingerprinting*, in Proceedings of the 2003 ACM SIGMOD international conference on Management of data - SIGMOD '03, 2003.

[11] D. Sraka and B. Kaucic, *Source code plagiarism*, in Proceedings of the ITI 2009 31st International Conference on Information Technology Interfaces, 6 2009.