Сидоркина И. Г., Белоусов С. А., Хукаленко К. С., Нехорошкова Л. Г.

АЛГОРИТМ ПОИСКА ПЛАГИАТА В ИСХОДНОМ КОДЕ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

Аннотация. *Программирование характеризуется большим числом разнообразных правил, приемов, методов и средств его выполнения, применение которых зависит от квалификации, опыта и индивидуальных особенностей программистов. Анализируются особенности алгоритмов анализа плагиата программного кода, величины семантического шума в текстах программ на основе исследуемых методов. Приводится алгоритм с использованием комбинированного подхода некоторых текстовых и семантических алгоритмов. Показано, в какое представление переводится исходный код программ в большинстве современных алгоритмов, далее описаны классы современных алгоритмов поиска плагиата в исходных текстах программ. В результате представлен усовершенствованный алгоритм поиска плагиата, предлагаемый для использования в учебной практике для выявления плагиата среди лабораторных работ студентов. Полученный алгоритм объединяет в себе как плюсы текстовых алгоритмов, так и семантических, при этом основная вычислительная часть имеет очень хороший параллелизм, что сокращает время ее выполнения при наличии вычислительных мощностей.*

Ключевые слова: *плагиат, исходный код, программный код, токен,,, семантика, семантические алгоритмы, коэффициент совпадения, коэффициент схожести, метрика, комбинированный алгоритм.*

Введение

Плагиат в исходных кодах программ встречается как в коммерческой разработке программного обеспечения, так и в учебной практике. Проблема особенно актуальна в сфере высшего образования, которая заключается не только в присвоении чужих работ, но и подрыве самой сути образовательного процесса. Необходимость инструментов для выявления плагиата непосредственно связана и с защитой интеллектуальной собственности. Таким образом, задача выявления плагиата приобретает все большую актуальность в различных сферах человеческой деятельности, и как следствие необходимы методы и средства, позволяющие автоматизировать этот процесс.

Формально определить понятие плагиата крайне сложно. Обычно под этим понимают случай, когда между исходными кодами двух программ есть существенная (на уровне языка программирования) общая часть. При этом производная программа получается из оригинальной несложными преобразованиями, цель которых — скрыть факт заимствования вставкой лишних операторов, изменением порядка следования в программе независимых операторов, разбиением одной функции на две, изменением имен переменных и т.п. Такое определение в большинстве случаев пригодно, ибо серьезные изменения исходного кода, сделанные для сокрытия плагиата вручную, крайне трудоемки, если же для этого используются автоматические средства, то по виду исходного кода это легко определяется человеком.

1. Классификация существующих алгоритмов

Существующие в данной области алгоритмы принято классифицировать следующим образом [**Prechelt L., Malpohl G., Philippsen M.**]:

- текстовые алгоритмы:
- структурные алгоритмы;
- семантические алгоритмы;

DOI: 10.7256/2305-6061.2013.3.9602

Текстовые алгоритмы представляют программу в виде строки над алфавитом, символы которого представляют оператор или группу операторов языка программирования. Причем аргументы операторов (которые сами могут быть операторами) игнорируются, что делает многие элементарные действия по сокрытию плагиата (например, изменения имен переменным) бесполезными. Символ такого алфавита традиционно называют токеном.

Текстовые алгоритмы [**Huang X., Hardison R. C., Miller W.**] включают в себя как наиболее эффективные современные алгоритмы поиска плагиата в исходных кодах программ, так и самые старые алгоритмы. Примером последних может служить подсчет некоторых характеристик программ (например, количество операторов ветвления в программе), а затем полученные векторы характеристик сравниваются между собой. Очевидна крайняя неэффективность такого подхода.

Практически во всех алгоритмах текстовой группы полагается, что если исходные коды программ являются похожими, то и на уровне строковых представлений у них присутствуют существенные общие части. Причем эти общие части не обязательно должны быть непрерывны. Из современных текстовых алгоритмов и методов выделим следующие:

- алгоритм на основе выравнивания строк — раздвигает символы строк так, чтобы выявить совпадающие участки;
- алгоритма основе строкового замощения [Wise M.J.] — эвристический алгоритм, выдающий наибольшее множество непересекающихся совпадающих подстрок, используются 2 эвристики:
  1) более длинные последовательные совпадения лучше, чем набор меньших и непоследовательных, даже если сумма длин последних больше;
  2) алгоритм игнорирует совпадения, длины которых меньше определенного порога;
- метод просеивания- ищет все общие подстроки фиксированного размера;
- метод отпечатков — сравнивает выборочные участки программ.

Структурные алгоритмы [Baxter I., Yahin A., Moura L., Anna M. S., BierL.], что следует из названия, используют саму структуру программы, обычно это или граф потока управления, или абстрактное синтаксическое дерево. Вследствие такого представления алгоритм сводит на нет многие возможные действия плагиатора. Но все алгоритмы этого класса являются крайне трудоемкими, поэтому не используются на практике.

Семантические алгоритмы [Moussiades L.M., Vakali A.] в чем-то похожи на структурные и текстовые, но в их основе лежат логические выводы, поиск в пространстве состояний. Например, в них может использоваться представление исходного кода программы в виде графа с вершинами двух типов. Одни строятся из последовательности операторов, которым назначается определенная семантика (например, математическое выражение, цикл), другие задают отношение, в котором состоят соседние с ней вершины (например, вхождение). Между двумя вершинами первого типа обычно стоит вершина второго.

2. Комбинированный подход

Предлагаемый подход объединяет в себе идеи некоторых текстовых и семантических алгоритмов. Ниже расписана последовательность действий алгоритма.

2.1.Семантическая часть

1. Преобразуем исходный код сравниваемых программ в поток токенов (токен–оператор или идентификатор языка программирования)
2. Разбиваем полученный поток токенов на слова, где под словом подразумевается простой оператор языка (в большинстве языков таким словом будет набор операторов между разделителями). В большинстве случаев исходный код, соответствующий одному слову занимает одну строчку.

DOI: 10.7256/2305-6061.2013.3.9602

3. Используя полученный набор слов, строим дерево, в узлах которого будут лишь те слова, которые соответствуют условным операторам, циклическим операторам или вызовам пользовательских функций, а листьями дерева будут остальные слова (а также пустое слово, например, для случая, когда в блоке выполнения условного оператора ничего нет).

   При этом все циклические операторы (for, repeat, и т.д.) сводим к эквивалентному оператору while, все операторы условия (тернарный оператор "?:", switch, else, и т.д.) сводим к эквивалентному оператору if, оператор вызова пользовательской функции становится родительским для всех слов, соответствующих исходному коду данной функции.

2.2.Текстовая часть

1. Сравниваем соответствующие узлы следующим образом: каждый условный оператор первой программы с каждым условным оператором второй, аналогично с циклическими операторами и операторами вызова функций. Все прочие ветвления дерева внутри выбранного узла игнорируются, и сравнивается набор слов, входящий прямо или косвенно в выбранный для сравнения узел.

   Группу сравниваемы наборов слов обозначим блоком, коэффициент совпадения блоков будет рассчитываться по формуле, где –коэффициент схожести i-го слова, а n–их количество.

   Полученный коэффициент можно использовать в метрике для выделения блоков, потенциально скопированных (полностью или частично) из другой программы.
2. При сравнении двух наборов слов, сравнивается каждое слово одного набора с каждым словом из другого, получаем соответствующие коэффициенты схожести. Чтобы получить конечный коэффициент схожести для одного слова, выбираем максимальный коэффициент его схожести на любое из слов другого набора.
3. Чтобы сравнить два слова между собой и получить коэффициент схожести, показывающий, какая доля первого слова включена во второе, будем перебирать размер окна и "двигать" его по первому слову, окно — подстрока первого слова.

   Выполняются следующие шаги:
   1) Размер окна k изменяем в следующем диапазоне: от длинны первого слова len до t (шумовой порог — совпадение подстрок данного размера и меньше считаются случайными, например, 2)
   2) Двигаем окно по первому слову (шаг = 1 символ, соответственно количество шагов = len — k + 1)
   3) Если подстрока, соответствующая положению окна первого слова, входим во второе, помечаем все соответствующие символы первой строки.
   4) По достижении окном текущего размера k конца слова, могут возникнуть следующие ситуации:
      - Отмеченных символов нет (ни одна подстрока размера k первого слова не встречается во втором), коэффициент схожести при данном k равен 0
      - В противном случае, текущий коэффициент для данного k в зависимости от количества помеченных символов линейно находится в интервале [min.. max], где min = k / len, max = 1 + 1/len — 1/k
   5) После перебора всех k выбираем наибольший коэффициент из полученных.

   Выбор границ min и max обусловлен следующим:

   min — если было хоть одно совпадение подстроки длинны k, то логичным будет вывод о том, что хотя бы k /len часть первой строки совпадает с частью второй

   max — при уменьшении размера окна, увеличивается шумовой эффект, уменьшаем полученный коэффициент схожести.

DOI: 10.7256/2305-6061.2013.3.9602

3. Заключение

Полученный алгоритм объединяет в себе как плюсы текстовых алгоритмов, так и семантических, при этом основная вычислительная часть имеет очень хороший параллелизм, что сокращает время ее выполнения при наличии вычислительных мощностей. Стоит отметить, что эмпирических данных о работе данного алгоритма еще недостаточно, но для приемлемой скорости выполнения (до 2 сек, при сравнении двух программ до 1000строк исходного кода) достаточно и обычного ПК, при выполнении основной части программы на графическом процессоре.

Библиография

1. Wise M. J. String similarity via greedy string tiling and running Karp-Rabin matching. // Dept. of CS, University of Sydney. December 1993.
2. Baxter I., Yahin A., Moura L., Anna M. S., BierL. Clone Detection Using Abstract Syntax Trees. // Proceedings of ICSM. IEEE. 1998.
3. Prechelt L., Malpohl G., Philippsen M. JPlag: Finding plagiarisms among a set of programs. // Technical Report No. 1/00, Universityof Karlsruhe, Department of Informatics. March 2000.
4. Moussiades L. M., Vakali A. PDetect: A Clustering Approach for Detecting Plagiarism in Source Code Datasets. // The Computer Journal Advance Access. June 24, 2005
5. Manber U. Finding similar files in a large filesystem. // Proceedings of the USENIX Winter 1994 Technical Conference. San Francisco. 1994. P. 1–10.
6. Huang X., Hardison R. C., Miller W. A space-efficient algorithm for local similarities. // Computer Applications in the Biosciences 6. 1990. P. 373–381.

References (transliterated)

1. Wise M. J. String similarity via greedy string tiling and running Karp-Rabin matching. // Dept. of CS, University of Sydney. December 1993.
2. Baxter I., Yahin A., Moura L., Anna M. S., BierL. Clone Detection Using Abstract Syntax Trees. // Proceedings of ICSM. IEEE. 1998.
3. Prechelt L., Malpohl G., Philippsen M. JPlag: Finding plagiarisms among a set of programs. // Technical Report No. 1/00, Universityof Karlsruhe, Department of Informatics. March 2000.
4. Moussiades L. M., Vakali A. PDetect: A Clustering Approach for Detecting Plagiarism in Source Code Datasets. // The Computer Journal Advance Access. June 24, 2005
5. Manber U. Finding similar files in a large filesystem. // Proceedings of the USENIX Winter 1994 Technical Conference. San Francisco. 1994. P. 1–10.
6. Huang X., Hardison R. C., Miller W. A space-efficient algorithm for local similarities. // Computer Applications in the Biosciences 6. 1990. P. 373–381.

DOI: 10.7256/2305-6061.2013.3.9602