Санкт-Петербургский Государственный Университет

Математико-механический факультет

Кафедра системного программирования

Проверка избыточности и минимизация множества тестов

Курсовая работа студентки 445 группы

Байцеровой Юлии Сергеевны

Научный руководитель **Вояковская Н.Н.**

(Старший преподаватель кафедры системного программирования СПбГУ)

Санкт-Петербург

2013

Содержание

1. Введение

Компилятор преобразует программу, написанную на языке высокого уровня, в программу, пригодную для исполнения машиной. Ошибки в компиляторе чреваты отказами или ошибками при выполнении исполняемых файлов, построенных компилятором, поэтому обеспечение корректности функционирования компилятора является важнейшей задачей. Корректность функционирования компиляторов проверяется тестированием. В большинстве случаев количество тестов для компилятора исчисляется тысячами, и т.к. они пишутся большими командами разработчиков, многие тестовые случаи повторяются. В связи с этим возникает потребность в выявлении похожих и, следовательно, формировании набора уникальных тестов.

В качестве инструментального компилятора мной использовался Codegen, продукт, разработанный сотрудниками Lanit-Tercom'a и представляющий собой многоплатформенный компилятор языка разработки бизнес-приложений Rules Language в языки C, Java, Cobol, C# и некоторые другие. Продукт разрабатывался 15 лет, и за это время к компилятору было написано больше 14000 тестов.

2. Постановка задачи

В рамках данной курсовой мной разрабатывался инструмент для оптимизации тестирования компилятора Codegen. В процессе поиска решения данной задачи возникало множество вопросов, например:

1. В каком виде представлять тесты для последующего анализа?
2. Писать ли свой инструмент представления или использовать уже созданные разработчиками Codegen’а инструменты?
3. Какой алгоритм сравнения выбрать?
4. Как оптимизировать поиск похожих тестов среди 14000?

Таким образом, основная задача разбивается на следующие подзадачи:

1. Изучение способов представления исходного кода:
    a) Исходный код
    b) В виде n-мерного пространства
    c) Токенизированное представление
    d) В виде дерева или ориентированного графа
2. Изучение методов поиска плагиата в исходных кодах программ (ввиду схожести задачи):
    a) Атрибутные методы
    b) Структурные методы
    c) Комбинированные методы

3. Обзор способов представления исходного кода и алгоритмов поиска плагиата в программах

3.1. Способы представления

Рассмотрим нашу задачу как задачу поиска плагиата в исходных кодах программ. Среди способов представления исходного кода основными являются следующие:

1. **Параметризованное представление**

Ранние системы по обнаружению плагиата представляли программу, как точку в n-мерном пространстве натуральных чисел с нулем, i-ая координата которой – количественная характеристика какого-либо свойства (атрибута) всей программы. Например, средняя длина строки кода, количество объявленных и используемых переменных, количество операторов ветвления и так далее. Если точки двух программ лежат рядом, то одна из них предполагается плагиатом другой.

Обычно такие системы называют «подсчитывающими отличительные черты» (attribute-counting systems), чтобы определить, насколько близко лежат точки двух программ, обычно из них подсчитывается метрика (например, Евклидово расстояние) или же определяется их корреляция.

Вычисляя i-ую характеристику (т.е. координату) на протяжении всей программы, мы получаем усредненное значение, поэтому теряем слишком много информации о структуре программы. У таких систем два недостатка: они выдают очень много ложных совпадений и при крайне поверхностном изменении кода оригинала не находят плагиат. Если же мы используем только ту часть какой-либо программы в нашей, то найти плагиат с их помощью практически невозможно. В некоторых таких системах используются и более сложные характеристики, основанные на использовании графа потока управления (control-flow graph), но это более трудоемкие алгоритмы и, к тому же, небольшие изменения структуры программы могут вызвать сильные качественные изменения таких характеристик.

2. Исходный код

Другие системы обнаружения плагиата рассматривают исходный код «как есть». Например, так поступают детекторы плагиата, которые работают с кодом так же, как и с обычными текстами. Но они крайне неэффективны для решения задачи поиска плагиата, т.к. переименование функций и переменных или несущественные изменения в коде

являются серьезными препятствиями для их правильной работы. Иногда используется параметризованное представление кода. Один из его вариантов таков: имена функций и переменных заменяются при первой встрече в коде на ноль, а при последующих - на расстояние до предыдущей позиции. Обычно детекторы, основанные на этих двух представлениях, лучше находят плагиат, чем системы «подсчитывающие отличительные черты», а также способны находить плагиат в случаях, когда скопирована только часть программы.

3. Токенизированное представление

Пусть у нас есть две строки кода: for (int i = 0; i <= n; i++) и for (int j = 0; j < n + 1; j++). Очевидно, что у них одинаковая функциональность, и плагиатор мог из одной получить другую без особых усилий, а для ранее описанных представлений они совершенно различны. Чтобы бороться с такого рода средствами сокрытия плагиата, было придумано токенизированное представление кода. Основная идея этого представления – это сохранение существенных и игнорирование всех поверхностных (то есть легко модифицируемых) деталей кода программы. Процедура токенизации выглядит примерно так:

- Каждому оператору языка (кроме пустого – его игнорируем), который не является операндом, приписываем код, назначенный заранее для каждого класса операторов. Также коды можно приписывать блочным операторам (например, dcl/enddcl), подключениям библиотек и заголовочных файлов
- Строим строку из полученных кодов, сохраняя порядок следования их в исходном коде программы. Один символ строки (токен) – код одного оператора.

Таким образом мы автоматически игнорируем названия функций и переменных (классов, объектов и так далее), разделительные символы, предотвращаем влияние мелких изменений кода программы.

Нужно отметить, что процесс токенизации и разбиение операторов на классы зависим от используемого в исходном коде языка программирования.

К одному классу операторов обычно относят те, который соответствует один идентификатор языка программирования, все вызовы функций, вызовы методов классов, объявления переменных элементарных типов, объявление экземпляров классов.

4. В виде дерева или ориентированного графа

Наконец, исходный код программы можно представлять в виде таких сложных структур, как деревья и ориентированные графы. Выбор вида дерева достаточно широк и остается за разработчиками детектора, но в основном для поиска плагиата используются суффиксные и AST-деревья. Такое представление не чувствительно к перестановкам частей кода программ, чего нет в предыдущих представлениях, но методы сравнения таких представлений гораздо сложнее.

3.2. Алгоритмы поиска плагиата

Принято выделять следующие подходы к оценке близости программ: атрибутно-подсчетный, структурный и комбинированный, сочетающий в себе первых два.

3. 2. 1. Атрибутные методы

Появились исторически первыми. Их смысл заключается в численном выражении некоторых признаков (атрибутов) программы, например, размер программы или число переменных, и сравнении полученных чисел для разных программ. Программы с близкими численными характеристиками атрибутов потенциально похожи. Можно комбинировать несколько признаков так, чтобы программа была представлена некоторым набором числовых характеристик. Две программы могут считаться похожими, если соответствующие числа из их наборов совпадают или близки. Таким образом, оценка близости программ сводится к сравнению чисел или векторов, которые получаются путем несложного анализа непосредственно исходного кода. Основным недостатком атрибутных техник является то, что несвязанные между собой параметры программы плохо описывают ее в целом. Следовательно, при таком подходе разные программы получают близкие характеристики.

3. 2. 2. Структурные методы

Исследуют свойства программы не изолированно, а как бы в контексте, устанавливают взаимосвязь различных характеристик, их совместное поведение. Чтобы отбросить лишнюю информацию и выделить нужные зависимости, программа предварительно переводится в более компактное представление (выполняется токенизация исходного кода). Классическим примером структурного подхода является построение дерева программы с последующим сравнением деревьев для разных программ. Недостатком структурных методов является их сложность и вычислительная

трудоемкость. Кроме того, структурные методы обычно опираются на синтаксис конкретного языка программирования. Адаптация метода для другого языка требует значительных усилий. Сложность реализации алгоритмов, использующих структурные методы, является платой за точность этих алгоритмов.

Рассмотрим наиболее известные структурные методы поиска плагиата в исходных кодах программ.

- **Метод выравнивания строк**

Пусть у нас есть две программы, представим их в виде строк токенов s и t соответственно (возможно различной длины). Теперь мы можем воспользоваться методом локального выравнивания строк, разработанным для определения схожести строк ДНК. Выравнивание двух строк получается с помощью вставки в них пробелов таким образом, чтобы их длины стали одинаковыми. Заметим, что существует большое количество различных выравниваний двух строк.

Достоинства:

1) Использование процедуры кластеризации.

2) Токенизированное представление программ.

Недостатки:

1) На базе этого алгоритма нельзя организовать базу данных, ускоряющую проверку один-против-всех.

- **Алгоритм жадного строкового замощения**

Эвристический алгоритм получения жадного строкового замощения (The Greedy String Tiling), получает на вход две строки символов над определенным алфавитом (у нас это множество допустимых токенов), а на выходе дает набор их общих непересекающихся подстрок близкий к оптимальному.

Достоинства:

1) Преимущества токенизированного представления.

2) Общие подстроки меньшей длины, чем MinimumMatchLength игнорируются, поэтому алгоритм не принимает в расчет небольшие случайно совпавшие участки кода.

3) При разбиении совпавшего участка кода на две и более части вставкой одного-нескольких блоков или одиночных операторов, а также перестановкой небольшого количества независимых операторов, функция схожести слабо изменяется.

4) Алгоритм нечувствителен к перестановкам больших фрагментов кода.

Недостатки:

1) Возможность совпадения токенизированного представления программ, но отсутствия совпадения в исходных кодах программ.

2) Разбиение совпадения на блоки, вставкой или заменой оператора на похожий (например, for на while), каждый длиной меньше MinimumMatchLength, ведет к полному игнорированию совпадения.

3) Из-за эвристик, используемых в алгоритме, совпадения, длиной меньшей, чем MinimumMatchLength, будут проигнорированы.

4) Нельзя организовать базу данных, ускоряющую проверку один-против-всех.

- **Метод отпечатков**

В этом алгоритме токенезированная программа представляется в виде набора отпечатков (меток, fingerprints), так чтобы эти наборы для похожих программ пересекались. Этот метод позволяет организовать эффективную проверку по базе данных. Метод отпечатков можно представить в виде четырех нижеследующих шагов:

1) Последовательно хэшируем подстроки токенизированной программы P длины k (фиксированный параметр).

2) Выделяем некоторое подмножество их хэш-значений, хорошо характеризующее P. Проделываем те же шаги для токенизированнных программ T_1, T_2, … , T_n и помещаем их выбранные хэш-значения в хэш-таблицу.

3) С помощью хэш-таблицы (базы) получаем набор участков строки P, подозрительных на плагиат.

4) Анализируем полученные на предыдущем шаге данные и делаем выводы. Существует несколько реализаций шага 2, но наиболее качественный это использование метода просеивания.

Достоинства:

1) Можно организовать базу данных, ускоряющую проверку один-против-всех.

2) Преимущества токенизированного представления.

3) Общие подстроки, меньше пороговой длины игнорируются, поэтому алгоритм не принимает в расчет малые, случайно совпавшие участки кода.

4) При разбиении совпавшего участка кода на две и более части вставкой одного-нескольких блоков или одиночных операторов, а также перестановкой небольшого количества независимых операторов, функция схожести слабо изменяется. (Длина совпадения должна быть значительно больше некоторой пороговой длины.)

5) Алгоритм нечувствителен к перестановкам больших фрагментов кода.

Недостатки:

1) Возможность совпадения токенизированного представления программ, но отсутствия совпадения в исходных кодах программ.

2) Разбиение совпадения на блоки, вставкой или заменой оператора на похожий (например, for на while), каждый длиной меньше k, ведет к полному игнорированию совпадения.

- **Алгоритм Хескела**

Пусть у нас есть две программы, представим их в виде строк токенов a и b соответственно. Одним из критериев сходства строк считается длина их наибольшей общей последовательности (НОП). Мы всегда можем найти такой элемент строки a_i, что НОП строк $a' = a_{|a|}a_{|a|-1}\ldots a_i a_1 \ldots a_{i-1}$ и b будет значительно меньше (максимум в два раза), чем НОП(a, b) (если НОП(a, b) > 1). Чтобы избежать этого явления можно воспользоваться алгоритмом сравнения строк Хескела, он требует нескольких подходов, но работает за линейное время. Разобьем строки a и b на k-граммы (подстроки длины k). Найдем те k-граммы, которые встречаются в a и b только по одному разу. Для каждой такой пары проверим совпадают ли элементы строк, непосредственно лежащие над ними; если это так, то проведем ту же проверку и для них и так далее, пока несовпадение не будет найдено. Аналогично для строк, лежащих ниже соответствующих k-граммов. Получаем набор общих непересекающихся подстрок a и b. Их общая длина может служить мерой схожести программ соответствующих a и b.

Достоинства:

1) Линейная трудоемкость алгоритма

Недостатки:

1) Возможность совпадения токенизированного представления программ, но отсутствия совпадения в исходных кодах программ.

2) Небольшое количество уникальных k-граммов в больших программах, соответственно многие совпадения, не содержащие в себе таких k-граммов, будут проигнорированы.

3) Вставка в найденный блок или изменение на семантически эквивалентный оператор во многих случаях будет приводить к игнорированию той части блока, в которой не содержится уникальный k-грамм.

4) Нельзя организовать базу данных, ускоряющую проверку один - против - всех.

3. 2. 3. Комбинированный подход

Его целесообразно использовать для поиска плагиата в большой базе программ. Для этого на первом этапе с помощью одного из атрибутных методов можно отсеивать заранее непохожие программы. На втором этапе выполняется более детальное сравнение оставшихся программ каким-либо структурным методом. Таким образом, за счет предварительного несложного анализа сокращается количество попарных сравнений при поиске плагиата, а, следовательно, растет эффективность.

4. Обзор метрик для вычисления расстояния между деревьями

В связи с тем, что в качестве представления сравниваемых кодов двух программ мной были выбраны деревья, генерируемые парсером Codegen'a, я проанализировала самые распространенные метрики для вычисления расстояния между деревьями. Это метрики Стэнли Селкова, Габриэля Валиенте и метрика Кай-Чжун Чжан и Дэннила Шаша.

- **Алгоритм Стэнли Селкова**

Метрика Стэнли Селкова введена им для вычисления расстояния между помеченными упорядоченными деревьями. Помеченным упорядоченным деревом он называет непустое конечное множество T с функцией маркировки λ, такое что:

1) T имеет особую вершину, называемую корнем дерева
2) Остальные вершины (исключая корень) разделены на $m \geq 0$ непересекающихся множеств $T_1, \dots, T_m$, и каждое из этих множеств является деревом. Эти множества называются поддеревьями дерева T
3) С каждой из вершин $v \in T$ связана метка $\lambda(v)$. Метка для корня дерева T обозначается как $\lambda(T)$. Пусть даны два дерева: A и B с поддеревьями $A_1, \dots, A_m$ и $B_1, \dots, B_n$ соответственно, тогда A и B равны, если $\lambda(A) = \lambda(B)$, $m = n$ и $A_i = B_i$ для $1 \leq i \leq m$

Далее вводятся так называемые операции редактирования:

1) Операция замены метки $L(s_i, s_k)$, примененная к дереву T, дает дерево T* с $\lambda(T^*) = s_k$ и поддеревьями $T_1, \dots, T_m$
2) Операция вставки $I(A)$ для $0 \leq i \leq m$ и дерева A, примененная к дереву T в i-ой позиции, дает дерево T* с $\lambda(T^*) = s_j$ и поддеревья $T_1, \dots, T_i, A, T_{i+1}, \dots, T_m$
3) Операция удаления $D(T_i)$ для $1 \leq i \leq m$, примененная к T в i-ой позиции, дает дерево T* с $\lambda(T^*) = s_j$ и поддеревьями $T_1, \dots, T_{i-1}, T_{i+1}, \dots, T_m$

С каждой операцией редактирования связывается неотрицательная стоимость следующим образом. С каждой парой меток (s_i, s_j) связывается стоимость $C_L(s_i, s_j)$ применения операции $L(s_i, s_j)$. Для каждой метки s_i выражениями $C_I(s_i)$ и $C_D(s_i)$ обозначаются стоимости применения операций I(T) и D(T) соответственно, где T – дерево с одной вершиной и $\lambda(T) = s_i$. Для произвольного дерева T:

$$C_I(T) = \Sigma_{v \epsilon T} C_I(\lambda(v)) \text{ и } C_D(T) = \Sigma_{v \epsilon T} C_D(\lambda(v))$$

Пусть есть деревья A и B и множество последовательностей операций редактирования, которые, будучи примененными к дереву A, дают дерево, равное B. Тогда вычисление расстояния между деревьями сводится к вычислению функции δ(A, B), обозначающую минимальную сумму стоимостей для каждой последовательности.

Сложность алгоритма: $O(|T_A| \times |T_B|)$, где $|T_A|$, $|T_B|$ - число листьев деревьев T_A и T_B соответственно.

- **Метрика Кай-Чжун Чжан и Дэннила Шаша**

Эта метрика, аналогично метрике Селкова, основана на трех операциях редактирования:

1) Операция замены. Замена одной метки узла на другую.
2) Операция удаления. Удаление узла (все дети удаленного узла становятся детьми его родителя).
3) Операция вставки. Вставка узла (все дети узла, который становится родителем вставляемого узла, становятся детьми вставляемого узла).

Пусть γ – функция стоимости, которая сопоставляет каждой операции редактирования $a \rightarrow b$ неотрицательное вещественное число $\gamma(a, b)$.

Функция γ является метрикой, т.е.

1) $\gamma(a \rightarrow b) \geq 0$; $\gamma(a \rightarrow a) = 0$
2) $\gamma(a \rightarrow b) = \gamma(b \rightarrow a)$
3) $\gamma(a \rightarrow c) \leq \gamma(a \rightarrow b) + \gamma(b \rightarrow c)$

Формально расстояние между T_1 и T_2 определяется следующим образом:

$\delta(T_1, T_2) = \min \{ \gamma(S) |$ S – последовательность операций редактирования, превращающая дерево T_1 в дерево $T_2\}$.

Сложность алгоритма: $O(|T_A| \times |T_B| \times \min(depth(T_A), |leaves(T_A)|) \times \min(depth(T_B), |leaves(T_B)|))$.

- **Метрика Габриэля Валиенте**

Метрика для вычисления расстояния между деревьями, введенная Габриэлем Валиенте, называется расстоянием «снизу-вверх».

Расстояние «снизу-вверх» между двумя непустыми деревьями T_1 и T_2 равно $1 - f / \max(n_1, n_2)$, где f – размер наибольшего общего леса для деревьев T_1 и T_2, а n_1 и n_2 – число листьев в деревьях T_1 и T_2 соответственно.

Сложность алгоритма: $O(|T_A| \times |T_B| \times \log(T_A + T_B))$.

5. Обзор системы обнаружения плагиата NoCrib

Сотрудниками Томского политехнического университета Хаустовым П. А. и Кацманом Ю. Я. была разработана система поиска плагиата NoCrib, в которой за основу взяты trie-деревья и расстояние Левенштейна. Исходный код представляется в виде лексем. Для хранения структуры кода используются trie-деревья. В вершинах дерева хранятся уровни вложенности, представленные в виде последовательности лексем. Для вершин данного дерева определяется операция вычисления степени схожести. Операция определения степени схожести осуществляется с помощью вычисления расстояния Левенштейна D_L. Пусть длина первой строки L_1, а длина второй строки L_2, тогда степень схожести этих двух строк равняется:

$Sim = \exp \{-4*D_L^2/L_1*L_2\}$

При таком способе представления исходного кода поиск оператора в дереве удобно осуществлять по уровням вложенности, которые в дереве соответствуют ярусам. Таким образом, имеет смысл рассматривать дерево по ярусам, начиная от самого внешнего, который представлен вершиной, соединенной с корнем дерева, и заканчивая самым удаленным от корня. Для оператора одного из исходных кодов, степень вхождения которого в сравниваемый исходный код мы хотим определить, известно содержимое каждого из уровней вложенности. Таким образом, можно представить оператор исходного кода, как ветку аналогичного trie-дерева. На каждом из ярусов trie-дерева можно перебрать все вершины V и определить степень схожести Sim_v с соответствующей вершиной ветки текущего оператора. Для получения итоговой степени схожести этого оператора с частью ветки trie-дерева до заданной вершины V нужно умножить степень схожести Sim_v на итоговую степень схожести предка вершины V. Таким образом, если ветка сравниваемого оператора состоит из H вершин, то из всех листьев на ярусе глубины H необходимо выбрать лист V_{max} с наибольшей итоговой степенью схожести. Итоговая степень вхождения оператора в исходный код и будет равна итоговой степени схожести вершины V_{max}. Если ярус глубины H не содержит листьев, то степень вхождения текущего оператора полагается равной нулю.

6. Предлагаемое решение

В ходе изучения продукта Codegen мной было выяснено, что этот компилятор умеет строить бинарные парсерные деревья для тестов. Поэтому в качестве представления тестов мной была выбрана линейная скобочная запись парсерных деревьев, а в качестве алгоритма сравнения – подсчет схожести между деревьями по матрице различия. На этот алгоритм меня натолкнула статья Кэрола Романовского и Ракеша Наги. В ней они описывают алгоритм вычисления так называемого значения разреженности для двух неупорядоченных деревьев. Взяв за основу их идею вычисления различия между деревьями по матрице различия, я вычислила функцию схожести двух деревьев. Этот алгоритм был выбран мной благодаря его быстродействию, что очень важно из-за огромного количества сравниваемых тестов.

В итоге предложенное мной решение можно сформулировать двумя пунктами:

1. Использовать в качестве представления теста линейную скобочную запись дерева, которое строит сам компилятор.
2. Использовать алгоритм вычисления схожести между деревьями по матрице различия для сравнения двух тестов.

7. Описание решения

Для вычисления разницы между деревьями мной был выбран алгоритм, который состоит в следующем:

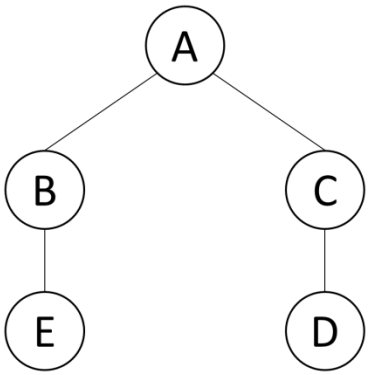
Пусть у нас есть два дерева A и B.

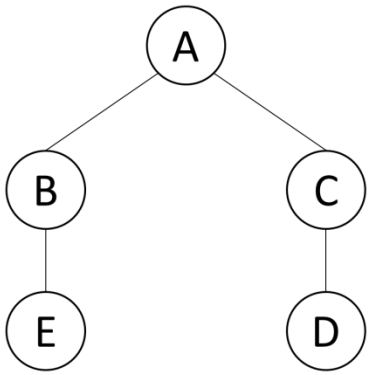
1) Строим общий список узлов для деревьев
2) Вычисляем матрицы смежности для деревьев, в которых отображены связи между узлами
3) Находим матрицу разницы, исключив связи, которые есть в обеих матрицах смежности
4) Считаем число связей в полученной матрице разницы
5) Делим это число на сумму связей в матрицах смежности

Полученное число вычитаем из 1, умножаем на 100% и получаем схожесть двух деревьев.

Эта функция была получена мной из краевых условий на эту функцию следующим образом. Рассмотрим случаи:

1) Деревья идентичны (т.е. когда функция принимает максимальное значение, равное 1)





	A	B	C	D	E
A	0	1	1	0	0
B	0	0	0	0	1
C	0	0	0	1	0
D	0	0	0	0	0
E	0	0	0	0	0

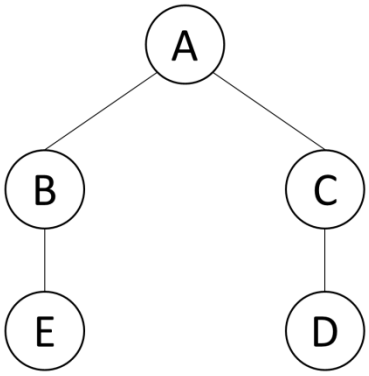
	A	B	C	D	E
A	0	1	1	0	0
B	0	0	0	0	1
C	0	0	0	1	0
D	0	0	0	0	0
E	0	0	0	0	0

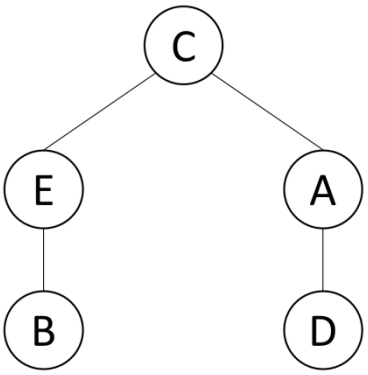
Матрица различия:

	A	B	C	D	E
A	0	0	0	0	0
B	0	0	0	0	0
C	0	0	0	0	0
D	0	0	0	0	0
E	0	0	0	0	0

Схожесть: $1 - d/x = 1 - 0/x = 1$

2) Деревья абсолютно разные (т.е. когда функция принимает минимальное значение, равное 0)





	A	B	C	D	E
A	0	1	1	0	0
B	0	0	0	0	1
C	0	0	0	1	0
D	0	0	0	0	0
E	0	0	0	0	0

	A	B	C	D	E
A	0	0	0	1	0
B	0	0	0	0	0
C	1	0	0	0	1
D	0	0	0	0	0
E	0	1	0	0	0

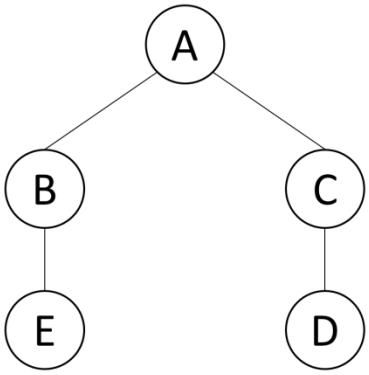
Матрица различия:

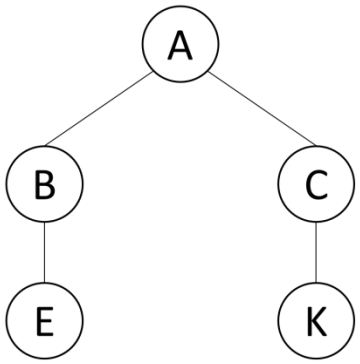
	A	B	C	D	E
A	0	1	1	1	0
B	0	0	0	0	1
C	1	0	0	1	1

D	0	0	0	0	0
E	0	1	0	0	0

Схожесть: 1 – d\x = 1 – 8\x = 0 → x = 8 (что равно сумме связей в матрицах смежности)

3) Деревья почти идентичны





	A	B	C	D	E	K
A	0	1	1	0	0	0
B	0	0	0	0	1	0
C	0	0	0	1	0	0
D	0	0	0	0	0	0
E	0	0	0	0	0	0
K	0	0	0	0	0	0

	A	B	C	D	E	K
A	0	1	1	0	0	0
B	0	0	0	0	1	0
C	0	0	0	0	0	1
D	0	0	0	0	0	0
E	0	0	0	0	0	0
K	0	0	0	0	0	0

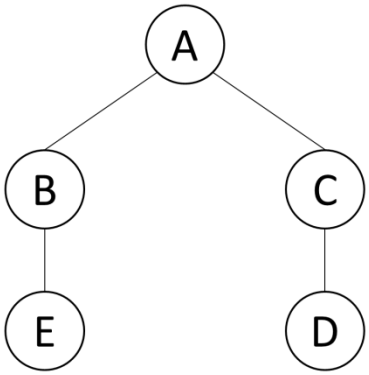
Матрица различия:

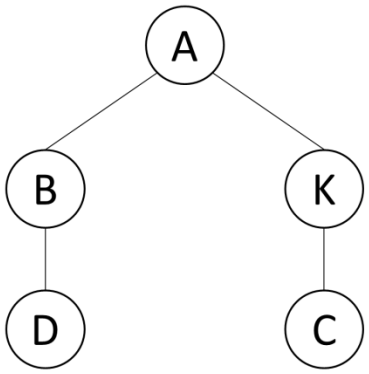
	A	B	C	D	E	K
A	0	0	0	0	0	0
B	0	0	0	0	0	0
C	0	0	0	1	0	1
D	0	0	0	0	0	0
E	0	0	0	0	0	0

K	0	0	0	0	0	0

Схожесть: $1 - 2/x = 0.75 \rightarrow x = 8$

4) Деревья почти абсолютно разные





	A	B	C	D	E	K
A	0	1	1	0	0	0
B	0	0	0	0	1	0
C	0	0	0	1	0	0
D	0	0	0	0	0	0
E	0	0	0	0	0	0
K	0	0	0	0	0	0

	A	B	C	D	E	K
A	0	1	0	0	0	1
B	0	0	0	1	0	0
C	0	0	0	0	0	0
D	0	0	0	0	0	0
E	0	0	0	0	0	0
K	0	0	1	0	0	0

Матрица различия:

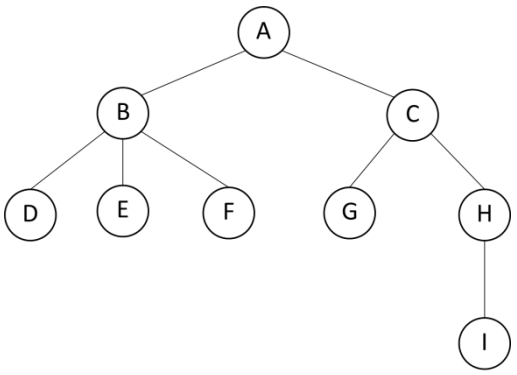
	A	B	C	D	E	K
A	0	0	1	0	0	1
B	0	0	0	1	1	0
C	0	0	0	1	0	0
D	0	0	0	0	0	0
E	0	0	0	0	0	0
K	0	0	1	0	0	0

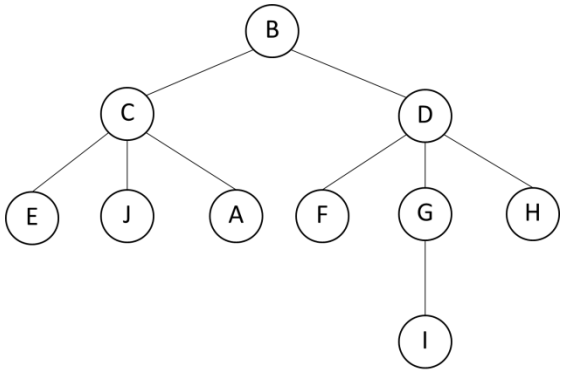
Схожесть: $1 - 6/x = 0.25 \rightarrow x = 8$

Получили функцию вычисления схожести: $f = 1 - d/(n_1 + n_2)$, где n_1 – число ребер

первого дерева, n_2 – число ребер второго дерева.

Рассмотрим этот алгоритм на примере. Пусть есть два дерева:





1) Строим для этих деревьев список вершин: A, B, C, D, E, F, G, H, I, J.
2) Строим матрицы смежности для этих деревьев и ищем общие части:

	A	B	C	D	E	F	G	H	I	J
A	0	1	1	0	0	0	0	0	0	0
B	0	0	0	1	1	1	0	0	0	0
C	0	0	0	0	0	0	1	1	0	0
D	0	0	0	0	0	0	0	0	0	0
E	0	0	0	0	0	0	0	0	0	0
F	0	0	0	0	0	0	0	0	0	0
G	0	0	0	0	0	0	0	0	0	0
H	0	0	0	0	0	0	0	0	1	0
I	0	0	0	0	0	0	0	0	0	0
J	0	0	0	0	0	0	0	0	0	0

	A	B	C	D	E	F	G	H	I	J
A	0	0	0	0	0	0	0	0	0	0
B	0	0	1	1	0	0	0	0	0	0
C	1	0	0	0	1	0	0	0	0	1
D	0	0	0	0	0	1	1	1	0	0
E	0	0	0	0	0	0	0	0	0	0
F	0	0	0	0	0	0	0	0	0	0
G	0	0	0	0	0	0	0	0	1	0
H	0	0	0	0	0	0	0	0	0	0
I	0	0	0	0	0	0	0	0	0	0
J	0	0	0	0	0	0	0	0	0	0

3) Строим матрицу различия:

	A	B	C	D	E	F	G	H	I	J
A	0	1	1	0	0	0	0	0	0	0
B	0	0	0	1	1	1	0	0	0	0
C	1	0	1	0	0	0	1	1	0	1
D	0	0	0	0	0	0	0	0	0	0
E	0	0	0	0	0	0	0	0	0	0
F	0	0	0	0	0	0	0	0	0	0
G	0	0	0	0	0	0	0	0	0	0
H	0	0	0	0	0	0	0	0	1	0
I	0	0	0	0	0	0	0	0	0	0
J	0	0	0	0	0	0	0	0	0	0

4) Считаем число 1 в матрице различия: d = 15
5) Вычисляем схожесть деревьев: (1 - d/(8+9))*100% = 0.1176*100% = 11,76%

8. Заключение

В рамках данной курсовой мной был разработан инструмент для сравнения тестов на похожесть. Этот инструмент будет являться вспомогательным для анализа тестового покрытия компиляторов, что является целью моей дипломной работы.

9. Список литературы

1. Конюхова О. В. Информационная система для поиска плагиата в программных кодах письменных работ студентов.
2. Александр Красс. Обзор алгоритмов обнаружения плагиата в исходных кодах программ.
3. Stanley M. Selkow. The tree-to-tree editing problem.
4. Kaizhong Zhang and Dennis Shasha. Simple fast algorithms for the editing distance between trees and related problems.
5. Хаустов П. А., Кацман Ю. Я. Алгоритм NCP обнаружения плагиата в исходных кодах программ на языках высокого уровня.
6. Carol J. Romanowski and Rakesh Nagi. On comparing bills of materials: A similarity/distance measure for unordered trees.