Оглавление

Введение

Сравнение программ это процесс сопоставления исходных кодов и нахождение похожих частей.

Во время учебных контестов возможны обманы, с копированием и изменением кода. Плагиат, так-же, возможен при копировании исходных кодов открытых проектов и использовании, в последствии, в закрытых проектах, что противоречит некотором лицензионным соглашениям (например GPL)

Целью данной курсовой работы является создание программы на языке Python для сравнения двух программ на языке C, которая выводит похожие части и, затем, коэффициент от 0 до 100

Данная курсовая доступна в системе контроля версий[1]

1. Описание библиотеки «pycparser»[2]

Библиотека предназначена для генерации синтаксического дерева программы, написанной на языке C. Генерация занимает несколько этапов:

Лексический анализ (для этого используется модуль, названный PLY(Python Lex-Yacc)

Lex — (*Nix) программа для генерации лексических анализаторов

Yacc — (*Nix) программма для генерации синтаксических анализаторов(парсеров)

Построение формата, для представления дерева:

Существует конфигурационный файл(_c_ast.cfg) и py-скрипт(_ast_gen.py) для генерации формата, который представляется в файле c_ast.py

Построение синтаксического дерева[1]

Пример:

Для программы:

```
static void foo(int k)
{
      j = p && r || q;
      return j;
}
```

Выдаст:

```
FileAST:
  FuncDef:
    Decl: foo, [], ['static']
      FuncDecl:
        ParamList:
          Decl: k, [], []
            TypeDecl: k, []
              IdentifierType: ['int']
        TypeDecl: foo, []
          IdentifierType: ['void']
    Compound:
      Assignment: =
        ID: j
        BinaryOp: ||
          BinaryOp: &&
            ID: p
            ID: r
          ID: q
      Return:
        ID: j
```

1 В информатике это конечное, помеченное, ориентированное дерево, в котором внутренние вершины сопоставлены с (помечены) операторами языка программирования, а листья с соответствующими операндами. Таким образом листья являются пустыми операторами и представляют только переменные и константы. Синтаксические деревья используются в парсерах для промежуточного представления программы между деревом разбора (конкретным синтаксическим деревом) и структурой данных, которая затем используется в качестве внутреннего представления компилятора или интерпретатора компьютерной программы для оптимизации и генерации кода. Возможные варианты подобных структур описываются абстрактным синтаксисом (http://ru.wikipedia.org/wiki/Абстрактное_синтаксическое_дерево)

Синтаксические деревья представляются в виде графа, вершины которого являются классами-наследниками Node. В классе Node определены методы show() (рекурсивный метод, для отображения дерева) и виртуальный метод children() (для получения всех «детей» этого класса).

Так-же там определен класс NodeVisitor, который предназначен для рекурсивного прохода по дереву. Подробнее со структурой и исходным кодом можно ознакомиться на сайте проекта.

Каждый класс, наследующий Node() и представляющий синтаксическое дерево содержит в себе:

- def __init__(*args) — конструктор класса.
- def children() — реализация виртуальной функции, определенной в классе предке.
- attr_names — список с атрибутами конструкции. Нужен для функции show() и переводу обратно на язык C.

2. Алгоритм сравнения синтаксических деревьев.

Алгоритм состоит из нескольких частей:

1. Для всех полных поддеревьев генерируем строки из названий операторов.
2. Заводим словарь: {Хеш_от_строки:[список корней поддеревьев]}
3. Сравниваем между собой все корни поддеревьев с одинаковыми хешами из разных файлов.

На пункт 3 нужно обратить особое внимание:

В нашем случае рассматриваются поддеревья с точностью до замены названий переменных и функций. Поскольку в языке С блочная структура программы, и переменная видна только в своем блоке и во внутренних, то мы можем с легкостью переопределять переменную во внутреннем блоке. Это значит что эти две переменные будут ссылаться на разные ячейки памяти, хоть и будут называться одинаково. Это значит, что нам придется создавать массив, в котором будут сопоставляться каждой переменной свой идентификационный номер(id). Перед тем, как мы будем выполнять пункт 3, мы должны пробежаться по массиву и «заменить» названия переменных на их id, и в будущем будем смотреть не на названия, а на их id.

В конце мы должны пробежаться по массиву и вывести все похожие части исходных кодов, и коэффициент, который вычисляется в зависимости от количества вершин во всех похожих подграфах.

Коэффициент схожести графов считается по формуле (100*N^2/(T1*T2) (где N — количество совпадающих вершин, T1 — количество вершин в первом дереве, а T2 — количество вершин во втором дереве), что дает само по себе огромную погрешность и является слабым местом программы.

3. Структура программы

В системе контроля версий следующая структура папок:

/examples/*

/deps/*

main.py

ctoc.py

gencfg.py

expand.py

Папка examples хранит в себе примеры исходных файлов с ответами программы. Файлы имеют вид filef_n.c, files_n.c и fileo_n.out для первого, второго и выходного файла соответственно.

Папка deps содержит в себе пакеты, которые нужны для запуска программы на ванильной версии python 3.2.

В файле gencfg.py находятся 4 функци:

assist() - функция, которая по данной вершине АСД возвращает ее строковое представления(нужна для построения хеш таблицы)

init_Decl() - замена конструктору Decl()

init_ID() - замена конструктору ID()

MakeInit() – функция, которая вызывается с самого начала main(), для подмены конструктора Decl,ID и замены таблицы аттрибутов.

В файле expand.py находятся 2 функции:

expand_decl() - функция для раскрытия объявления переменных или объявления типа.

expand_init() - функция для раскрытия инициализации переменной.

В файле ctoc.py нас будет интересовать только одна функция и класс:

print_file() - вывод исходного кода.

CGenerator — класс, который схож по алгоритму на NodeVisitor, но возвращается

строки и «аккумулирует» их. Посещая каждый лист рекурсивно и возвращая строку исходного кода.

В файле main.py находится один функтор, одна функция-декоратор и 8 функций о которых речь пойдет в следующей главе.

3.1. Структура main.py

В файле main.py объявлен один функтор, одна функция-декоратор и 8 функций:

Функтор func_count нужен для построения таблицы переменных.

Декоратор timer(f) нужен для замер времени. Используется для нахождения узких мест в программе.

Функция AddParsingArguments() нужна для вывода справки, разбора входных аргументов и настройки выходного потока.

hashes_func( ) - функция, которая берет дерево и создает словарь {Хеш:[набор вершин]}

parsing_file( ) - функция, в процессе которой файл «парсится» и получается АСД, а так-же словарь {Хеш:[набор вершин]}. А так-же строящий таблицу имен. В процессе построения редактируется поле id у объектов, описателями которых являются классы ID и Decl.

comp_subtrees( ) - функция, которая сравнивает поддеревья, строя таблицу сопоставления имен. Алгоритм жадный, поэтому при первой нестыковке все разваливается.

UncryptDecl( ) - функция, которая строит таблицу имен и типов.

Таблица типов нужна для раскрытия «сложных» типов(Struct, Union или Enum). При наличии структуры внутри структуры, внутренняя структура раскроется и таким образом мы сможем сравнивать конструкции без надобности проверять наличие внутренних подструктур и раскрытия их в во время проверки.

3.2. Использование в командной строке.

При использовании программы в командной строке доступны несколько параметров:

-h/--help — вызов справки

-f/--first — путь до первого файла. По умолчанию: examples/main1_1.c

-s/--second — путь до второго файла. По умолчанию: examples/main2_1.c

-n/--number — кол-во совпадений, которые надо вывести на экран. По умолчанию:10

--dt — отладка таблицы имен.

4.Тестирование

4.1. Тестирование вывода таблицы имен для деревьев.

Протестируем программу на простом файле examples/main_dt.c:

```
int main(){
  double i = 0, b = 1;
  for(int i = 1; i < 100; ++i){
    i++;
    b =3;
  }
}
```

Она выведет:

```
$ ./main.py --dt -f examples/main_dt.c
Execution Time: 2.275111
First name table
#---------------------------------------
{1: ['i', ['double']], 2: ['b', ['double']], 3: ['i', ['int']]}
#---------------------------------------
FileAST:
  FuncDef:
    Decl: main, [], [], [], 1
      FuncDecl:
        TypeDecl: main, []
          IdentifierType: ['int']
    Compound:
      Decl: i, [], [], [], 1
        TypeDecl: i, []
          IdentifierType: ['double']
        Constant: int, 0
      Decl: b, [], [], [], 2
        TypeDecl: b, []
          IdentifierType: ['double']
        Constant: int, 1
      For:
        DeclList:
          Decl: i, [], [], [], 3
            TypeDecl: i, []
              IdentifierType: ['int']
            Constant: int, 1
        BinaryOp: <
          ID: i, 3
          Constant: int, 100
        UnaryOp: ++
          ID: i, 3
        Compound:
          UnaryOp: p++
            ID: i, 3
          Assignment: =
            ID: b, 2
            Constant: int, 3
```

Между выделением(#---..) находится сама таблица имен в формате {id:[название, тип]}

Тип выглядит как кортеж элементов, причем если это Struct/Union, то она полностью раскрывается и выглядит как [Struct/Union, «Набор полей»] в том числе со вложенными структурами. Перечисление выглядит как [Enum, «кол-во элементов»].
Справа от Decl и ID находятся его атрибуты и поле id(последнее число). Как мы можем заметить внутри цикла For переменная i — с id 3, то есть отлична от i, которая объявлена вне цикла. У нее id — 1.

4.2. Тестирование сравнения поддеревьев.

Пример 1:

Протестируем для двух исходных файлов main1_1.c и main2_1.c, которые отличаются только именами переменных:

```
int main(){
  int a,b,c;
  return a+b-c;
}
```

и

```
int main(){
  int q,w,e;
  return q+w-e;
}
```

Программа выдаст:

```
$ ./main.py -f examples/main1_1.c -s examples/main2_1.c -o
examples/main1.o
Coefficient of Coincidens is  100.0
Fragment Number: 1
#---------------------------------------
0> int main()
0> {
0>   int a;
0>   int b;
0>   int c;
0>   return (a + b) - c;
0> }

1> int main()
1> {
1>   int q;
1>   int w;
1>   int e;
1>   return (q + w) - e;
1> }
#---------------------------------------
```

Что будет являться верным, так как программы совпадают.

Пример 2

Протестируем для двух исходных файлов main1_2.c и main2_2.c, которые отличаются одним включенным циклом for:

```
int main(){
  for (int i = 0; i < 100; ++i){
    if (i == 50) return i;
  }
}
```

и

```
int main(){
  for (int i = 0; i < 100; ++i)
    for (int j = 0; j < 100; ++j){
      if (j == 50) return j;
    }
}
```

Программа выдаст:

```
$ ./main.py -f examples/main1_2.c -s examples/main2_2.c -o
examples/main2.o
Coefficient of Coincidens is  53.77777777777778
Fragment Number: 1
#----------------------------------------
Beggining from the line 2
0> for (int i = 0; i < 100; ++i)
0> {
0>   if (i == 50)
0>     return i;
0> }

Beggining from the line 3
1> for (int j = 0; j < 100; ++j)
1> {
1>   if (j == 50)
1>     return j;
1> }
#----------------------------------------
Fragment Number: 2
#----------------------------------------
Beggining from the line 1
0> int main()

Beggining from the line 1
1> int main()
#----------------------------------------
```

Что будет являться верным:совпадает объявление функции и внтуренний цикл for.

Пример 3

Протестируем для двух исходных файлов main1_2.c и main2_2.c, которые отличаются тем, что внутри вложенного for используется переменная j, а не i:

```
int main(){
  for (int i = 0; i < 100; ++i)
    for (int j = 0; j < 100; ++j){
      if (i == 50) return i;
    }
}
```

и

```
int main(){
  for (int i = 0; i < 100; ++i)
    for (int j = 0; j < 100; ++j){
      if (i == 50) return i;
    }
}
```

Программа выдаст:

```
$ ./main.py -f examples/main1_2.c -s examples/main2_2.c -o
Coefficient of Coincidens is  49.0
Fragment Number: 1
#----------------------------------------
Beggining from the line 0
0> {
0>   if (i == 50)
0>     return i;
0> }

Beggining from the line 0
1> {
1>   if (i == 50)
1>     return i;
1> }
#----------------------------------------
Fragment Number: 2
#----------------------------------------
0> int i = 0

1> int j = 0
#----------------------------------------
Fragment Number: 3
#----------------------------------------
Beggining from the line 1
0> int main()

Beggining from the line 1
1> int main()
#----------------------------------------
Fragment Number: 4
#----------------------------------------
Beggining from the line 2
0> i < 100

Beggining from the line 3
```

```
1> j < 100
#----------------------------------------
Fragment Number: 5
#----------------------------------------
Beggining from the line 2
0> ++i

Beggining from the line 3
1> ++j
#----------------------------------------
```

Что будет являться верным, так как все эти компоненты совпадают, но вместе не образуют совпадающий код.

Как мы можем заметить коэффициент каждый раз уменьшается

Заключение

В ходе работы написана программа, которая сравнивает исходные коды и выводящая коэффициент похожести от 0 до 100, устойчивая к изменению названий переменных.

В планах (в нужном порядке реализации):

1. Приведение в порядок уже написанного кода.
2. Придумать функцию для вычисления схожести с меньшей погрешностью.
3. Сделать программу менее требовательной к ресурсам
4. Сделать программу устойчивой к перемене местами кусков кода независимых от данных.
5. Расширение работы программы до языка C++ и Go.
6. Полностью переписать код на язык C++ (для повышения скорости работы) с использованием библиотеки Boost:Spirit(Wave)

Список Литературы.

1. Ахо А., Д ж. Ульман. Теория синтаксического анализа, перевода и компиляции (в 2-х т.). — М. : Мир, 1978.
2. Ахо А., Хопкрофт Дж., Ульман Дж. Построение и анализ вычислительных алгоритмов.— М.: Мир, 1979. - 536 с.
3. Красс А. Обзор алгоритмов обнаружения плагиата в исходных кодах программ: сайт URL: http://rain.ifmo.ru/cat/data/theory/unsorted/plagiarism-2006/article.pdf
4. Лифшиц. Проект по обнаружению плагиата в исходных кодах: сайт URL: http://logic.pdmi.ras.ru/~yura/detector/
5. Бизли Д. Python. Подробный справочник - Addison-Wesley, 2009 — 717 с.
6. Mark Pilgrim. Dive into Python - Apress, 2009 — 500 с.

[1] https://github.com/bigbes92/c-compare — Ссылка на хостинг проекта
[2] Адрес проекта: http://code.google.com/p/pycparser , актуальная версия на момент написания — 2.04