СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Для оценки правильности решения алгоритмической задачи чаще всего используют тесты – набор данных, протестировав на которых задачу можно выяснить, дает ли она правильный ответ. Результат решения, который дала задача, сравнивают с результатом, который дает верный алгоритм. Если задача решена неверно, тестирующая система сообщит об этом, показав тип ошибки: ошибку может вызвать неправильный ответ(WA), превышение допустимого времени работы программы(TL), ошибка при исполнении программы(RE), превышение допускаемого объема памяти(ML) и другие. Иногда, решение задачи может быть довольно простым, не требующее специальных алгоритмов, но и решениях таких задач со стороны студента может возникать ошибка. Работа представляет из себя небольшое исследование, цель которого выяснить, можно ли исправить неверное решение участника, изменив узлы в “AST” – древовидной структуре программы, являющейся промежуточным этапом между переводом алгоритма в bytecode и зная верный алгоритм решения.

1. Постановка задачи

Данное исследование представляет из себя решение двух подзадач: первая задача заключается в демонстрации структуры абстрактного синтаксического дерева, представление его в виде ориентированного графа и описание алгоритма, с помощью которого можно визуализировать граф. Вторая подзадача состоит в изменении узлов AST, получение измененного кода программы и проверка правильности реализованных изменений – компиляция получившейся программы.

2. Решение поставленной задачи

Работа с абстрактными синтаксическими деревьями возможно благодаря библиотекам ast, astor. Скачивание альтернативных решений алгоритмической задачи происходит с помощью методов библиотек Beautiful Soup и request. Представление графа возможно благодаря библиотекам networkx и matplotlib.

Представление синтаксического дерева будем рассматривать на примере простой программы:

print(1+2)

Программа выводит на экран сумму двух констант, в программе отсутствуют переменные, значения которых могут изменяться. Вот так выглядит соответствующее программе AST:

```
Module(
    body=[
        Expr(
            value=Call(
                func=Name(id='print', ctx=Load()),
                args=[
                    BinOp(
                        left=Constant(value=1),
                        op=Add(),
                        right=Constant(value=2))],
                keywords=[]))],
    type_ignores=[])
```

Чтобы получить такое представление программы, необходимо воспользоваться методами AST parse(возвращает объект типа Module) и dump(принимает на вход объект типа Module и возвращает синтаксическое дерево). AST состоит из узлов Call (описывает вызываемые функции), Name, BinOp (описывает константы, с которыми происходят арифметические операции) и других.

Для визуализации и графического представления дерева был реализован алгоритм, итог работы которого – два словаря: первый представляет из себя соответствие каждого из узлов дерева конкретному числу, второй словарь – соответствие вершин друг другу(иначе – аналог списка смежности для представления графов). Итог работы алгоритма выглядит следующим образом:

Первый словарь:

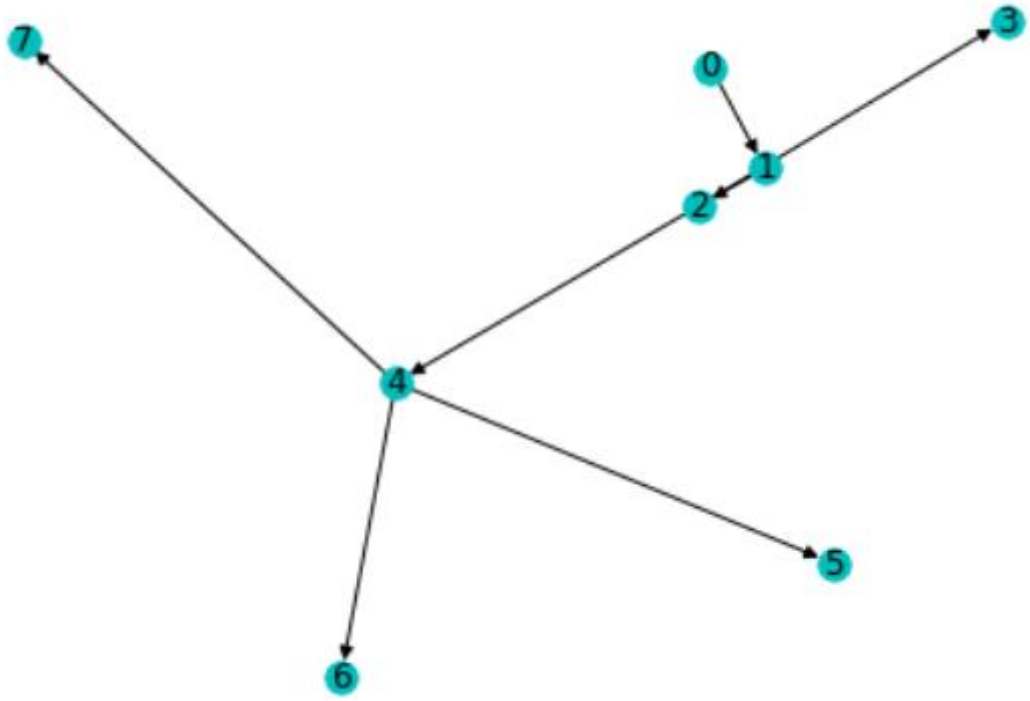
```
{"Module(body=[Expr(value=Call(func=Name(id='print', ctx=Load()), args=[Bi
nOp(left=Constant(value=1), op=Add(), right=Constant(value=2))], keywords=
[]))], type_ignores=[])": 0, "Expr(value=Call(func=Name(id='print', ctx=Lo
ad()), args=[BinOp(left=Constant(value=1), op=Add(), right=Constant(value=
2))], keywords=[]))": 1, "Call(func=Name(id='print', ctx=Load()), args=[Bi
nOp(left=Constant(value=1), op=Add(), right=Constant(value=2))], keywords=
[])": 2, "Name(id='print', ctx=Load())": 3, 'BinOp(left=Constant(value=1),
```

```
op=Add(), right=Constant(value=2))': 4, 'Constant(value=1)': 5, 'Add()':
6, 'Constant(value=2)': 7}

Второй словарь:

{0: [1], 1: [2], 2: [3, 4], 3: [], 4: [5, 6, 7], 5: [], 6: [], 7: []}
```

После представления дерева в виде словаря появляется визуализировать его с помощью методов библиотеки matplotlib и networks:



Иначе, структура программы представленна в виде связи определенных узлов, отвечающие за разные операции.

Теперь рассмотрим решение определенной алгоритмической задачи, составим AST и поменяем ряд узлов для изменения результата работы алгоритма.

Условие рассматриваемой задачи выглядит следующим образом:

B. Докажи, что он не прав

ограничение по времени на тест: 2 секунды
ограничение по памяти на тест: 256 мегабайт
ввод: стандартный ввод
вывод: стандартный вывод

Недавно, ваш друг обнаружил одну интересную операцию над массивом a:

1. Выберите два индекса i и j ($i \neq j$);
2. Присвойте $a_i = a_j = |a_i - a_j|$.

Поиграв с данной операцией некоторое время, он пришел к следующему утверждению:

- В любом массиве a из n целых чисел, в котором $1 \leq a_i \leq 10^9$, вы можете найти пару индексов (i, j) такую, что после применения операции выше общая сумма массива a **уменьшится**.

Данное утверждение вам кажется крайне сомнительным, а потому вы хотите найти контрпример для заданного n. Сможете ли вы найти такой контрпример и доказать, что он не прав?

Другими словами, найдите массив a, состоящий из n целых чисел $a_1, a_2, \ldots, a_n$ ($1 \leq a_i \leq 10^9$), такой, что для любой пары индексов (i, j) применение операции выше не уменьшает общую сумму (сумма либо возрастает, либо не меняется).

Входные данные
В первой строке задано одно целое число t ($1 \leq t \leq 100$) — количество наборов входных данных. Далее следуют t наборов входных данных.

В первой и единственной строке каждого набора задано одно целое число n ($2 \leq n \leq 1000$) — длина массива a.

Выходные данные
Для каждого набора входных данных, если не существует контрпримера в виде массива a размера n, выведите `NO`.

В противном случае выведите `YES` и сам массив a ($1 \leq a_i \leq 10^9$). Если существует несколько контрпримеров, выведите любой.

В качестве рассматриваемого кода возьмем 2 решения задачи: верное и неверное:

```
for i in range(int(input())):
    n = int(input())
    if n>19:
        print("NO")
    else:
        T = [3**i for i in range(n)]
        print("YES")
        print(*T)
```

```
_ = int(input())
for i in range(_):
  n = int(input())
  if n >= 18:
      print("NO")
  else:
    print("YES")
    for j in range(n):
        print(3**j, end=" ")
    print()
```

Последнее решение станет верным, если заменить константу в узле If на 20.

Для замены узлов были реализованны несколько классов: класс MyOptimizer1, метод vizit_If в котором позволяет рекурсивно обойти все узлы If; класс MyOptimizer2, в котором определен метод, позволяющий посетить узлы Constant и заменить в них значения на нужные – после доступа к узлам If есть возможность поменять значения в узлах Constant находящиеся именно в узле If. Классы MyOptimizer3 и MyOptimizer4

служат для корректной замены значений аргументов в цикле For – иногда требуется считать один набор данных, а не несколько для проверки праильности решения, ведь лишний ввод данных будет мешать быстро проверять задачу на правильность.

Класс, заменяющий узел, может выглядить следующим образом:

```
class MyOptimizer3(ast.NodeTransformer):
    def visit_Call(self, node: ast.Call):
        if(node.func).id == 'range':
            result = ast.Call(func=ast.Name(id='range', ctx=ast.Load()), args=[ast.Constant(value=1)], keywords=[])
            result.lineno = node.lineno
            result.col_offset = node.col_offset
            return result
        return node
```

- здесь мы определяем MyOptimizer3 класс, который расширяет ast.NodeTransformer класс и переопределяет visit_Call метод. В данном случае, если узел Name в функции Call имеет переменную id, равную range – возвращается замененный узел, иначе – возвращается исходный узел.

Узел If, содержащийся в неверном алгоритме, выглядел следующим образом:

```
If(
   test=Compare(
       left=Name(id='n', ctx=Load()),
       ops=[
           GtE()],
       comparators=[
           Constant(value=18)])
```

После изменения константы в If, узел выглядит иначе:

```
If(
    test=Compare(
        left=Name(id='n', ctx=Load()),
        ops=[
            GtE()],
        comparators=[
            Constant(value=20)])
```

Чтобы изменить значение в Constant таким образом, потребовалось экземпляр класса MyOptimizer2, создать переменную, равную экземпляру класса с методом visit, который принимает на вход исходное дерево и меняет значение в Constant; переменной присваевается измененное дерево.

Далее следуют заменить значение в For, которое находится в range – вместо традиционного int(input()) поставим значение, равное единице.

Как такой узел выглядел до преобразований:

```
For(
    target=Name(id='i', ctx=Store()),
    iter=Call(
        func=Name(id='range', ctx=Load()),
        args=[
            Call(
                func=Name(id='int', ctx=Load()),
                args=[
                    Call(
                        func=Name(id='input', ctx=Load()),
                        args=[],
                        keywords=[])],
                keywords=[])],
        keywords=[])
```

После:

```
For(
    target=Name(id='i', ctx=Store()),
    iter=Call(
        func=Name(id='range', ctx=Load()),
        args=[
            Constant(value=1)],
        keywords=[]),
```

- значение в узле поменялось на Constant(value = 1)

В данном случае, при таком подходе изменения узла, можно столкнуться со следующей проблемой: значение будет меняться там, где переменная id в узле ast.Name будет равняться range – в нашем правильном решении задачи присутствует также генерация списка, где переменная id в узле ast.Name будет также равняться range, но менять значение у range в этом списке не требуется, иначе решение задачи будет неверным. Проблему можно решить, создав класс, “восстанавливающий” данный узел до исходного состояния:

```
def visit_ListComp(self, node: ast.ListComp):
    result = ast.ListComp(
                            elt=ast.BinOp(
                                left=ast.Constant(value=3),
                                op=ast.Pow(),
                                right=ast.Name(id='i', ctx=ast.Load())),
                            generators=[
                                ast.comprehension(
                                    target=ast.Name(id='i', ctx=ast.Store()),
                                    iter=ast.Call(
                                        func=ast.Name(id='range', ctx=ast.Load()),
                                        args=[
                                            ast.Name(id='n', ctx=ast.Load())],
                                        keywords=[]),
                                    ifs=[],
                                    is_async=0)])
    result.lineno = node.lineno
    result.col_offset = node.col_offset
    return result
```

- метод позволяет вернуть изменный узел ListComp (иначе список который заполняется с помощью цикла) к его исходному состоянию.

Таким образом, с помощью методов библиотеки astor можно получить исходный код измененных программ:

for i in range(1):
  n = int(input())
  if n > 19:
  print('NO')
  else:
  T = [(3 ** i) for i in range(n)]
  print('YES')
  print(*T)
	_ = int(input())
	for i in range(1):
	  n = int(input())
	  if n >= 20:
	  print('NO')
	  else:
	  print('YES')
	  for j in range(n):
	  print(3 ** j, end=' ')
	  print()

Приведенные выше программы представляют из себя корректный код, скомпилировав который можно получить требуемый результат: в первом случае программа работает только для одного набора входных тестов, во втором – изменен узел и получено верное решение задачи.

3. Заключение

В данной работе были продемонстрированы возможности абстрактных синтаксических деревьев, а именно их аналих, их изменение и их визуализация. В дальнешем, открывается возможность разработать алгоритм, исправляющий неверное решение задачи на основе верной решенной с помощью анализа и изменения определнных узлов в AST в общем случае.

4. Список использованной литературы

1. docs.python.org. AST – абстрактные синтаксические деревья
2. habr.ru. Реализуем преобразование кода на Python
3. habr.ru. Как найти плагиат в контестах по программированию
4. stackoverflow.com. Compile python AST to method
5. pythonpool.com. AST module examples