Кисляков Данил 797, вариант 14

Сначала скачиваем graphviz. Подключаем numpy, collections и graphviz.Digraph.

Лучше используйте conda install python-graphviz вместо pip install graphviz

---

- Алгоритм:
  1. Читаем регулярку в обратной польской записи и слово, разделённые переносом строки БЕЗ ПРОБЕЛОВ
  2. Строим НКА
  3. Ищем наибольший суфикс слова, используя НКА и рисуем НКА или пишем об ошибке.
- Строим НКА:
  - Идём по строке регулярки в обратной польской записи
    - Прочли 1 -> добавляем вершину
    - Прочли a/b/c -> добавляем пару вершин и ребро между ними
    - Прочли $*$/./+ -> совершаем соответствующую операцию над 2 (для $*$ -- 1) лежащими в вершине стека подавтоматами и кладём на их место результат применения операции
    - Прочли что-то ещё -> кидаем ошибку
  - Попутно обрабатываем другие ошибки
  - В конце выкидываем лишние вершины (которые были стартовыми при создании пары вершин при чтении a/b/c)
- Ищем наибольший суфикс слова:
  - Для всех суфиксов слова, от большего к меньшему, пытаемся рекурсивно найти путь из старта в какую-то терминальную вершину по этому суфиксу.
  - Если нашли -> возвращаем длину текущего суфикса (наибольший по алгоритму)
  - Если не нашли -> пишем INF
  - Рекурсия глубины не более (длина суфикса * 2) т.к. по построению НКА у нас не может быть два перехода по 1 подряд
- Рисуем НКА:
  - Рисуем НКА, используя graphviz.Digraph

На тестах лектора решение верное, автоматы строятся верно, пусть и, возможно, не оптимально

In [5]:

```
import numpy as np
import collections as coll
from graphviz import Digraph


class FSA:  # НКА
    def __init__(self, reg_exp):
        self.error = False  # наличие ошибок во входнй строке
        size = len(reg_exp) + 1  # максимальное число вершин в автомате, которое можем получить в п
роцессе построения
        self.relations_table = [['0' for i in range(size)] for j in range(size)]  # матрица [from][
to] symbol
        self.terminals = [False for i in range(size)]  # терминальные вершины
        is_start = [False for i in range(size)]  # запоминаем, какие вершины назначили стартами в п
одавтоматах(позже удалим)
        current_max_vert = 0  # текущий максимальный номер вершины
        parser = coll.deque()  # стек для чтения обратной польской
        first = ()  # пары начало/конец обрабатываемых подавтоматов
        second = ()
        for symbol in reg_exp:
            if symbol == '1':
                is_start[current_max_vert] = True
                parser.append((current_max_vert, current_max_vert))  # создали вершину
                current_max_vert += 1
            elif symbol == 'a' or symbol == 'b' or symbol == 'c':
                is_start[current_max_vert] = True
                parser.append((current_max_vert, current_max_vert + 1))  # пару вершин
                self.relations_table[current_max_vert][current_max_vert + 1] = symbol  # соеденили
ребром
                self.terminals[current_max_vert + 1] = True  # вторая -- терминал
```

```
                current_max_vert += 2
            elif symbol == '*':
                if len(parser) == 0:  # недостаточно подавтоматов в стеке
                    self.error = True
                    break
                first = parser.pop()  # вытащили подавтомат из стека
                is_start[first[0]] = False  # не нужно удалять, т.к. на неё будут завязаны петли
                for i in range(first[0] + 1, first[1] + 1):  # добавляем рёбра из терминальных верш
н в стартовую
                    if self.terminals[i]:
                        self.relations_table[i][first[0]] = '1'
                self.terminals[first[0]] = True  # сделали корень терминалом
                parser.append((first[0], first[1]))
            else:
                if len(parser) < 2:  # недостаточно подавтоматов в стеке
                    self.error = True
                    break
                second = parser.pop()  # вытащили подавтоматы из стека
                first = parser.pop()
                if symbol == '.':
                    for i in range(first[0], first[1] + 1):  # провели рёбра из терминальных вершин
первого в потомков корня второго
                        for j in range(second[0] + 1, second[1] + 1):
                            if self.terminals[i]:
                                self.relations_table[i][j] = self.relations_table[second[0]][j]
                    if not self.terminals[second[0]]:  # если корень второго -- не терминал,
терминалы первого больше не терминалы
                        for i in range(first[0], first[1] + 1):
                            self.terminals[i] = False
                    parser.append((first[0], second[1]))
                elif symbol == '+':
                    for i in range(second[0] + 1, second[1] + 1):  # подвесили потомков корня второ
о к корню первого
                        self.relations_table[first[0]][i] = self.relations_table[second[0]][i]
                    if self.terminals[second[0]]:  # если корень второго -- терминал, то корень пер
вого тоже
                        self.terminals[first[0]] = True
                    parser.append((first[0], second[1]))
                else:  # прочли символ не из алфавита
                    self.error = True
                    break
        if len(parser) != 1:  # остались лишние символы
            self.error = True
        for vert in range(len(is_start) - 1, 0, -1):  # удаляем лишние "фиктивные" стартовые
вершины
            if is_start[vert]:
                self.relations_table.pop(vert)
                for i in range(len(self.relations_table)):
                    self.relations_table[i].pop(vert)
                self.terminals.pop(vert)


def find(fsa, node, word, symbol_pos_in_word):
    if symbol_pos_in_word == len(word):  # если прошли всё слово
        if fsa.terminals[node]:  # если мы в терминал => слово подходит
            return True
        else:  # иначе не получилось
            return False
    for i in range(len(fsa.relations_table[node])):  # для рёбер, по которым можем пройти по текуще
му символу или 1, ищем дельше
        if fsa.relations_table[node][i] == word[symbol_pos_in_word]:
            if find(fsa, i, word, symbol_pos_in_word + 1):
                return True
        elif fsa.relations_table[node][i] == '1':  # не более 1 еденицы подряд по построению
            if find(fsa, i, word, symbol_pos_in_word):
                return True
    return False  # если не нашли путь из этой вершины, возвращаем False


def print_max_suff_length(fsa, word):
    for first_symb in range(len(word) + 1):  # идём по симолам слова
        if find(fsa, 0, word, first_symb):  # пока суфикс, начинающийся с этого символа, не из наше
го языка
            print(len(word) - first_symb)  # если нашли, пишем длину суфикса (она максимальна, т.к.
все большие проверили ранее)
            return
    print("INF")  # если не нашли, пишем INF (это происходит, например, в случае "a","b")
```

```
    return


def print_fsa(fsa):  # код для отрисовки
    f = Digraph('finite_state_machine', filename='fsm.gv')
    f.attr(rankdir='LR', size='6')
    f.attr('node', shape='doublecircle')
    for vert in range(len(fsa.terminals)):  # добавили терминальные вершины
        if fsa.terminals[vert]:
            f.node(str(vert))
    f.attr('node', shape='circle')
    is_visited = [False for i in range(len(fsa.terminals))]  # обработтаные вершины
    queue = coll.deque()  # очередь обработки
    queue.append(0)  # добавим стартовуб вершину
    is_visited[0] = True
    while (len(queue) != 0):
        vert = queue.popleft()  # извлекаем первый элемент из очереди
        for edge in range(len(fsa.relations_table[vert])):  # для всех его потомков
            if fsa.relations_table[vert][edge] != '0':
                f.edge(str(vert), str(edge), label=fsa.relations_table[vert][edge])  # добавим рёбр
а в рисуемый граф
                if not is_visited[edge]:  # добавим непосещённых потомков в очередь
                    queue.append(edge)
                    is_visited[edge] = True
    f.view()  # просмотреть рисунок автомата


reg_exp = input()
word = input()  # прочли регулярку в обратной польской записи и слово, разделённые переносом строк
и БЕЗ ПРОБЕЛОВ
fsa = FSA(reg_exp)  # распарсили регулярку в НКА
if fsa.error:  # если регулярка была некорректна, пишем ERROR
    print("ERROR")
else:  # иначе
    print_max_suff_length(fsa, word)  # пишем длину самого длинного суфикса слова, принадлежащего
яыку, задаваемому регуляркой
    print_fsa(fsa)  # рисуем НКА
```

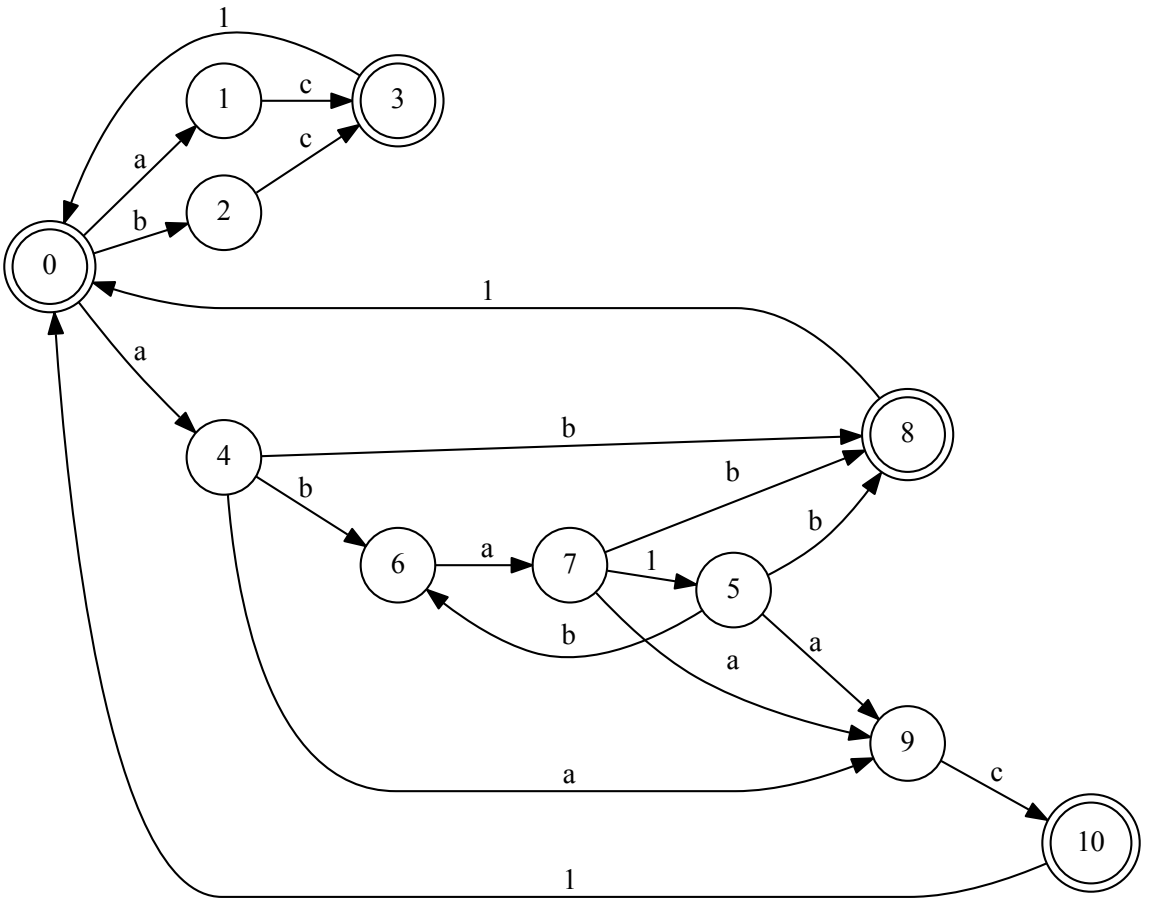
```
ab+c.aba.*.bac.+.+*
babc
2
```

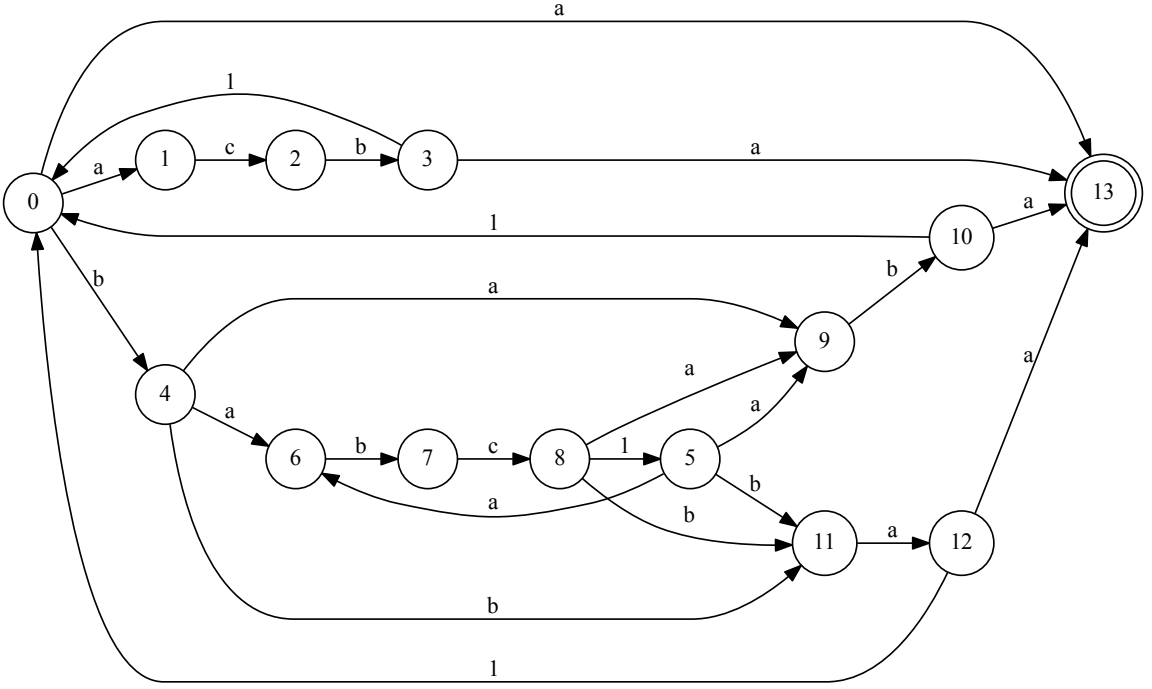
In [8]:

```
reg_exp = input()
word = input()  # прочли регулярку в обратной польской записи и слово, разделённые переносом строк
и БЕЗ ПРОБЕЛОВ
fsa = FSA(reg_exp)  # распарсили регулярку в НКА
if fsa.error:  # если регулярка была некорректна, пишем ERROR
    print("ERROR")
else:  # иначе
    print_max_suff_length(fsa, word)  # пишем длину самого длинного суфикса слова, принадлежащего
яыку, задаваемому регуляркой
    print_fsa(fsa)  # рисуем НКА
```

```
acb..bab.c.*.ab.ba.+.+*a.
cbaa
1
```

Повторюсь, что на тестах лектора решение верное, автоматы строятся верно, пусть и, возможно, не оптимально. Далее вы можете видеть сами автоматы для тестов ab+c.aba.*.bac.+.+* babc и acb..bab.c.*.ab.ba.+.+*a. cbaa соответственно:


1
1
c
3
a
c
b
2
0
1
a
4
b
8
b
b
6
a
7
1
5
b
b
a
a
9
a
c
10
1


a
1
a
1
c
2
b
3
a
13
0
a
1
10
b
a
b
9
4
a
a
a
6
b
7
c
8
1
5
a
b
a
b
11
a
12
b
a
1