Министерство образования и науки Российской Федерации

**ФГАОУ ВО «УрФУ имени первого Президента России Б.Н. Ельцина»**

Институт естественных наук и математики

Школа бакалавриата.

Оценка работы \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Руководитель от УрФУ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

*Тема: Изучение синхронизируемости автоматов*

ОТЧЕТ

по учебной практике по получению первичных умений и навыков

Руководитель практики от УрФУ Ананичев Дмитрий Сергеевич   
Студент Колесова Мария Николаевна

Специальность Математика и Компьютерные науки   
Группа МЕН-260207

Екатеринбург, 2018

Определения

Детерминированный конечный автомат(ДКА) - (Q, Σ, δ)

* Q - непустое конечное множество состояний автомата
* Σ - входной алфавит
* δ - функция переходов, δ: Q × Σ → Q

ДКА – синхронизируемый, если существует слово w над алфавитом Σ такое, что ∀q ∈ Q δ(q, w) = p (отображает все состояния ДКА в фиксированное состояние); w – синхронизирующее слово.

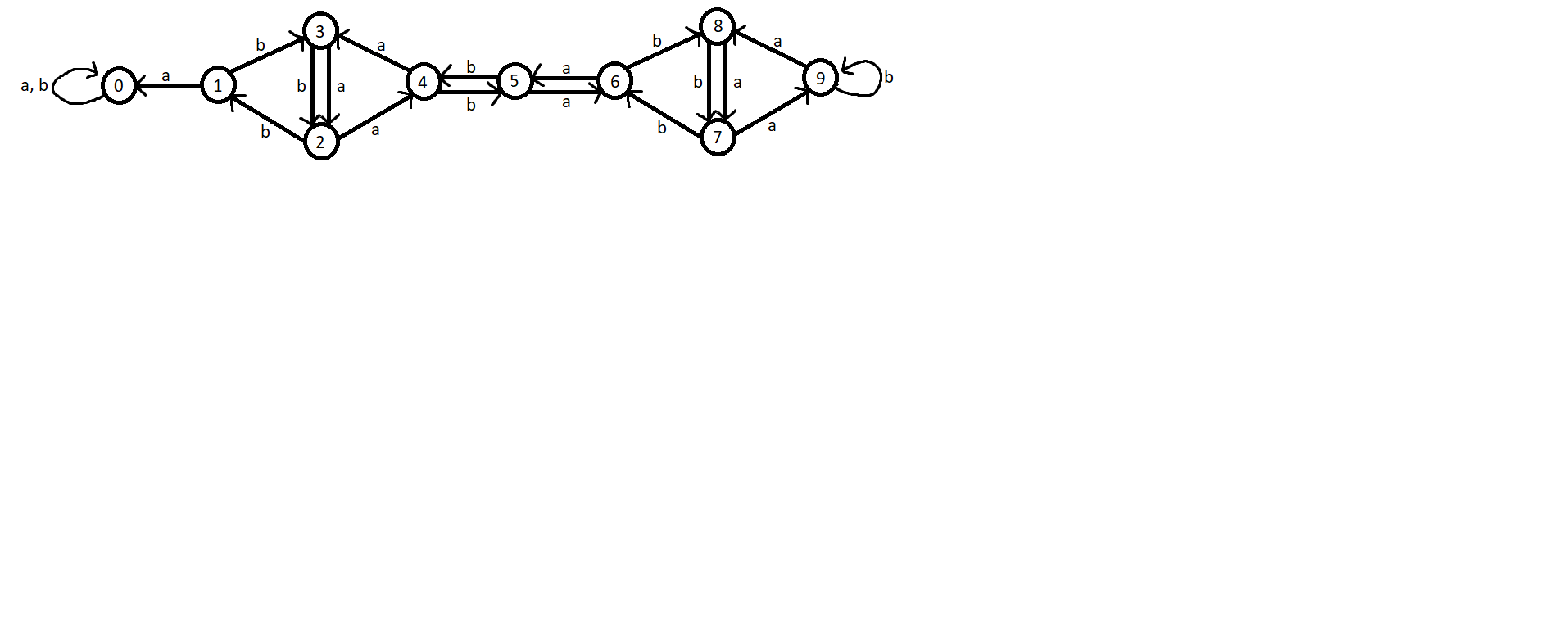
Порог синхронизации автомата – длина минимального синхронизирующего слова.

Входные данные

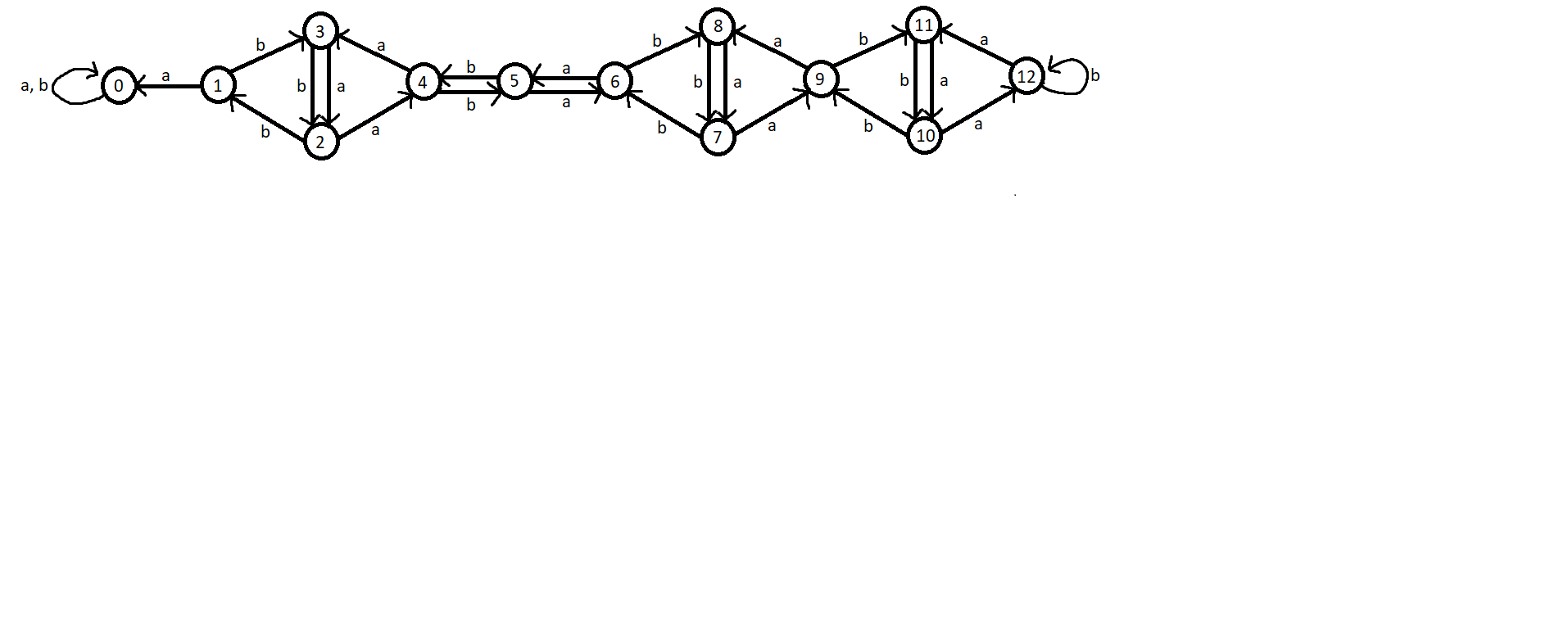
1.



2.



3.



Задача: найти порог синхронизации для каждого автомата (фиксированное состояние – 0)

Решение задачи

Обозначим i-е состояние автомата как qi.

Для решения задачи потребуются конфигурационные данные для каждого автомата:

* всевозможные циклы (пример для 1 автомата - cycles1 = [(0,), (3, 2, 1), (4, 3, 2), (4,)])
* длины циклов (пример для 1 автомата - cycles1conf = [1, 3, 3, 1])

Синхронизирующее слово обладает свойством: ∀q Є Q: δ(q, w) = q0. То есть, чтобы найти минимальное синхрослово для автомата, нужно найти минимальное синхрослово w: δ(qi-1, w) = q0, где qi-1 – самая дальняя вершина, и проверить, подходит ли это слово для остальных вершин.

Изначально минимальное расстояние (т.е. и длина слова |w|) от последней вершины до q0 равно n - 1. (|w| = n – 1)

1. Найдем всевозможные пути из последней вершины в q0  длины |w|. Для этого я составляю всевозможные шаблоны, которые показывают, какой цикл включать в маршрут.

Пример для 1 автомата: шаблон – [2, 0, 1, 0]. Соотнесем его с циклами первого автомата [(0,), (3, 2, 1), (4, 3, 2), (4,)]). То есть нулевой цикл включается в маршрут 2 раза, второй – 1 раз, а первый и третий не включаются. Тогда маршрут получится такой: (4-3-2-4)-3-2-1-0-(0)-(0). В скобках указаны добавленные циклы.

1. Для каждого полученного маршрута посчитаем длину, используя известный массив, содержащий длины циклов. Если цикл в шаблоне не указан, то к длине прибавляем 1 (расстояние до следующей вершины), иначе число из массива.

Пример для 1 автомата: для маршрута (4-3-2-4)-3-2-1-0-(0)-(0) длина будет складываться как (3)+1+1+1+1+(1)+(1), в скобках длины циклов, взятые из конфигурационных данных - cycles1conf = [1, 3, 3, 1], где i-ому значению соответствует i-ый цикл из cycles1 = [(0,), (3, 2, 1), (4, 3, 2), (4,)]).

1. Если длина равна |w|, то маршрут добавляем в список возможных маршрутов.
2. Пробегая по каждому маршруту из списка, находим все возможные слова, которые ему могут соответствовать.

Пример: маршрут 4-3-2-1-0. Возможные слова: a-b-b-a, a-a-b-a.

Для каждого полученного слова w проверяем, является ли оно синхронизирующим. То есть проверяем, что ∀q Є Q\{qn-1}: δ(q, w) = 0

1. Если проверка выполняется хотя бы для одного слова, то синхронизирующее слово найдено. Иначе, переходим к шагу 1, увеличивая на 1 длину маршрута.

Найденные слова:

1) a-b-b-a-b-b-a-b-a (длина 9)

2) b-a-b-a-a-b-a-a-a-a-a-b-a-a-a-a-a-b-a-a-a-a-a-b-a-a-b-b-a-b-a-a-b-a,

b-a-b-a-a-b-a-a-a-a-a-b-a-a-a-a-a-b-a-a-a-a-a-b-a-a-b-b-a-b-a-b-b-a (длина 34)

3)(пока еще вычисляется)

Доказательство корректности алгоритма:

Изначально длина минимально синхрослова равна длине минимально пути из последней вершины в нулевую и, если слов такой длины не найдено, то длина увеличивается на 1 и повторяется поиск, что гарантирует то, что синхрослово большей длины не будет найдено раньше.

Приложение

import sys

import itertools

def get\_automat(data):

count = int(data.readline())

global automat

automat = [{} for i in range(count)]

lines = data.read().split('\n')

for i in range(len(lines)):

split\_strings = lines[i].split()

keys = automat[int(split\_strings[0])].keys()

if int(split\_strings[1]) in keys:

automat[int(split\_strings[0])][int(split\_strings[1])] += split\_strings[2]

else:

automat[int(split\_strings[0])][int(split\_strings[1])] = split\_strings[2]

def make\_pattern\_for\_step(step\_number, len\_pattern):

number = str(bin(step\_number)[2:])

pattern = ''

for j in range(len\_pattern - len(number)):

pattern += '0'

pattern += number

return pattern

def dec(a, summ, k, i, result):

if summ < 0:

return

if summ == 0:

ar = []

for j in range(0, i):

ar.append(a[j])

result.append(ar)

else:

if summ - k >= 0:

a[i] = k

dec(a, summ - k, k, i + 1, result)

if k - 1 > 0:

dec(a, summ, k - 1, i, result)

return

def make\_list\_using\_cycles(len\_use\_list, len\_p):

global cyclesconf

list\_using\_cycles = []

result = []

zero\_array = []

decomp\_number = 1

for p in range(len\_p):

for i in range(decomp\_number):

zero\_array.append(0)

dec(zero\_array, decomp\_number, decomp\_number, 0, result)

for sublist in result:

if len(sublist) <= len\_use\_list:

possible\_using\_cycle = []

for num in sublist:

possible\_using\_cycle.append(num)

for i in range(len\_use\_list - len(possible\_using\_cycle)):

possible\_using\_cycle.append(0)

variants = [[x for x in itertools.permutations(possible\_using\_cycle)]]

variants = list(set(variants[0]))

for variant in variants:

way\_len = 0

for j in range(len(variant)):

if variant[j] == 0:

way\_len += 1

else:

way\_len += cyclesconf[2][j] + 1

if way\_len == len\_p:

list\_using\_cycles.append(variant)

decomp\_number += 1

return list\_using\_cycles

def get\_word():

global get\_automat

global cycles

with open("k5result.txt", 'w') as file:

max\_length\_word = (len(automat) - 1) \*\* 2

len\_possible\_path = 4

while True:

len\_possible\_path += 1

list\_using\_cycles = make\_list\_using\_cycles(len(cycles[2]), len\_possible\_path)

pathes = []

for us\_cycles in list\_using\_cycles:

pathes.append(find\_path(us\_cycles))

pathes.sort(key=sortyLength)

for possible\_path in pathes:

if len(possible\_path) > max\_length\_word:

sys.exit("Word does not exist")

else:

possible\_words = make\_words(possible\_path)

result = check(possible\_words)

if result:

file.write(str(result))

sys.exit(result)

def sortyLength(inputArr):

return len(inputArr)

def find\_path(use\_cycles):

global automat

global cycles

path = []

for last\_node\_number in range(len(automat) - 1, -1, -1):

for i in range(len(use\_cycles) - 1, -1, -1):

'''добавление цикла'''

if cycles[2][i][0] == last\_node\_number and use\_cycles[i] > 0:

for c in range(use\_cycles[i]):

for node in cycles[2][i]:

path.append(node)

path.append(last\_node\_number)

if len(path) == 0:

path.append(last\_node\_number)

else:

if path[-1] != last\_node\_number and i == 0:

path.append(last\_node\_number)

return path

def check(alist\_words):

global automat

words = alist\_words

for node in range(len(automat) - 1):

answer = check\_words(node, words)

if not answer:

return False

words = answer

return words

def check\_words(current\_n, words):

global automat

step\_done = False

possible\_answer = []

for word in words:

current\_node = current\_n

for i in range(len(word)):

next\_node = None

tuples = list(automat[current\_node].items())

for tup in tuples:

if word[i] in tup[1]:

next\_node = tup[0]

step\_done = True

if next\_node is not None and step\_done:

current\_node = next\_node

step\_done = False

if i == len(word) - 1 and current\_node == 0:

possible\_answer.append(word)

if not possible\_answer:

return False

else:

return possible\_answer

def make\_words(path):

global automat

words = []

word = ''

index = 0

change = dict()

for i in range(len(path) - 1):

current\_node = path[i]

next\_node\_number = path[i + 1]

value = automat[current\_node][next\_node\_number]

letters = []

for val in value:

for letter in val:

letters += letter

if len(letters) > 1:

change[index] = letters[1]

word += letters[0]

index += 1

words.append(word)

search\_words(change, words)

return words

def search\_words(list\_changes, words):

original = words[0]

changes = list(list\_changes.items())

keys = list(list\_changes.keys())

length = len(changes)

for i in range(1, 2 \*\* length):

copy = ''

pattern = make\_pattern\_for\_step(i, length)

index\_list = []

for q in range(len(pattern)):

if pattern[q] == '1':

index\_list.append(q)

for k in range(len(original)):

if k not in keys:

copy += original[k]

else:

for index in index\_list:

if changes[index][0] == k:

copy += changes[index][1]

if len(copy) - 1 < k:

copy += original[k]

words.append(copy)

def main():

with open("k5.txt") as file:

global automat

global cycles

global cyclesconf

cycles1 = [(0,), (3, 2, 1), (4, 3, 2), (4,)]

cycles1conf = [1, 3, 3, 1]

cycles2 = [(0,), (3, 2, 1), (4, 3, 2), (5, 4), (6, 5), (8, 7, 6), (9, 8, 7), (9,)]

cycles2conf = [1, 3, 3, 2, 2, 3, 3, 1]

cycles3 = [(0,), (3, 2, 1), (4, 3, 2), (5, 4), (6, 5), (8, 7, 6), (9, 8, 7), (11, 10, 9), (12, 11, 10), (12,)]

cycles3conf = [1, 3, 3, 2, 2, 3, 3, 3, 3, 1]

cycles = [cycles1, cycles2, cycles3]

cyclesconf = [cycles1conf, cycles2conf, cycles3conf]

get\_automat(file)

get\_word()

if \_\_name\_\_ == "\_\_main\_\_":

main()