Московский физико–технический институт

Т.Ф. ХИРЬЯНОВ

ЛЕКЦИИ ПО ИНФОРМАТИКЕ

Весенний семестр, 2016–2017 учебный год

При поддержке:
Г. Демьянов, VK
С. Клявинек, VK
Отдельная благодарность:
«Полиция КПМ»: А. Кожарин, VK

Москва, 2017 г.

Оглавление

ЛЕКЦИЯ 1

Повторение

1.1. Ссылочная модель данных

Ссылочная модель данных - объекты существуют независимо от времен.

```
>>> 2+3
5
```

Создаются два объекта типа int.

```
2.__add__(3) - метод добавления
```

```
x = 2+3 # x ссылается на объект 5
x = 'Hello' # возникает объект строки
```

После вывода результата объекты 2, 3, 5 удаляются, т.к. нет ссылки на эти объекты. Если x перестал ссылаться на 5, объект 5 уничтожается.

```
x = 5
y = x
x = 'Hello'
y = None
```

После того, как y начал ссылаться на None, объект 5 уничтожается.

```
x = (2+3+5)**5**5 # приоритет у возведения в степени
```

T.e. в начале выполняется 5**5, а потом (2+3+5)**25.

Начиная с версии Python 3.6, можно разделять числа так:

```
x = 100_001
y = 0xAF_DC
```

1.2. Пространство имен

4 пространства: локальные, окружающие, глобальные, строенные (LEGB).

y = 10*x+7 Сначала будет происходить поиск локального x, затем в надпространстве, затем в глобальных, в последнюю очередь - во встроенных переменных.

Если написать max = 10, то функция max перестанет работать.

Пример:

Программа №1.2.1.

```
1    def f(A):
2        A = A + 10 # если написать A += 10 ошибки не будет
3    B = [1, 2, 3]
4    f(B)
5    print(*B) # 1 2 3
```

Строки и числа — неизменяемые объекты. f является именем объекта functional.

def — по сути это операция создания нового объекта.

Любой вызов функции порождает свое пространство имен, которое перестанет существовать после выполнения return.

A начинает ссылаться на [1, 2, 3]. После конкатинации A начинает ссылаться на [1, 2, 3, 4]. A ссылается на глобальный объект, и начинает ссылаться на локальный объект. После return уничтожается A и список [1, 2, 3, 4].

Или можно записать так:

Программа №1.2.2.

```
1    def f(A):
2        A.append(10)
3    B = [1, 2, 3]
4    f(B)
5    print(*B) # 1 2 3
```

Функция должна что-то возвращать. В частном случае, можно возвращать несколько параметров. Нарушить ссылочную модель можно только "залезая" в глобальные имена.

Программа №1.2.3.

```
1    d=1
2    def f(A):
3        global d
4        A.append(d)
5        d = d + 1
```

1.3. ООП

Тип тоже является объектом типа тип.

Программа №1.3.1.

```
1    class Base:
2        x = 10
3        def g(self, x0):
4            self.x = x0
5    b = Base()
6    print(b.x)
```

Программа №1.3.2.

```
1    class Base:
2        x = 10
3        def g(self, x0)
4            self.x += x0
5    b = Base()
6    print(b.x)
```

Нужно создавать атрибуты только в методе init!

1.4. Элементы функционального программирования

1.4.1. Функция map

```
x, y, z = map(int, input().split()) # считывание трех чисел с клавиатуры
```

Функция map применяет к каждому объекту функцию, которую мы написали. В нее же можно написать свою функцию:

```
x, y, z = map(lambda x: int(x)**2, input().split())
A == list(map(int, range(100)))
B = map(float, int(x) for x in input().split())
```

map возвращает объект типа map.

1.4.2. Применение lambda-функций

$x^2 + e^{1/x} + \ln x$

```
x = decomposed_value # нужно объявлять функцию
(lambda x: x**2 + exp(1/x)+ln(x))(2) #хороший пример использования lambda
```

1.4.3. Функция enumerate

Программа №1.4.1.

```
1    A = [10, 20, 30]
2    for i,x in enumerate(A): # A используется как итерируемый объект
3        print(i,x) # (0,10), (1,20), (2,30) - эту конструкцию нам вернет enumerate(A)
4        x = x + 1 # Значение в массиве не изменилось, мы только испортили x
```

1.4.4. Функция zip

```
A = [1, 2, 3, 4, 5]
B = 'Hello'
c = list(zip(A,B)) # результат есть zip-object
```

ЛЕКЦИЯ 2

Графы

2.1. Графы

Граф — множество вершин и инцедентных им ребер.

$$G = (V, E)$$

$$v \in V, \qquad e \in E$$

Говорят, что ребро e инцедентно вершине v, если она является его концом.

Допустимы графы:

$$G = (\varnothing, \varnothing)$$

$$G = (1, \varnothing)$$

Недопустим граф:

$$G = (\varnothing, a)$$

Граф — "упрощенная модель".

У ребра 2 конца. Это не обязательно отрезок.

Ребро может быть петлей.

2 разных ребра могут быть инцедентно двум вершинам — кратные ребра.

У классического графа 2 конца. Но может быть ориентированный граф. Т.е. либо у ребра 2 конца, либо у него есть начало и конец. Тогда ребро называется дуга. Короткое название **орграф**.

2.2. Задача Эйлера о семи кёнигсбергских мостах

Как пройти по всем городским мостам (через реку Преголя), не проходя ни по одному из них дважды?

Введем дополнительные понятия:

Степень вершины — количество инцедентных ей ребер.

Граф G' = (V', E') является подграфом G, если V' $\subset$ V, E' $\subset$ E.

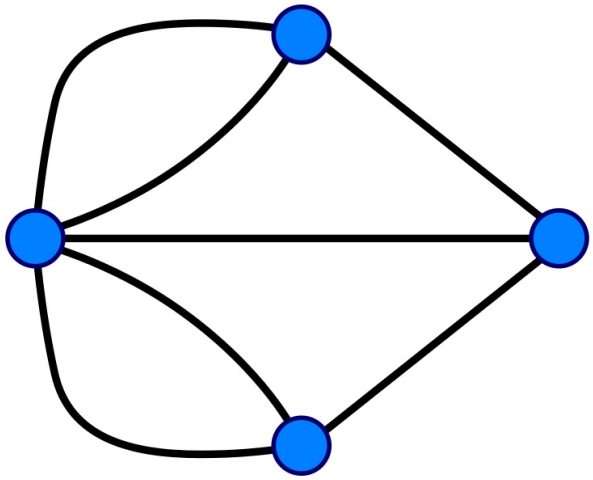

Рис. 2.1. Граф к задаче о мостах

Путь — последовательность ребер (в которой конец каждого ребра есть начало следующего).

Путь тоже является графом, а точнее это орграф, подграф исходного.

Любой неориентированный граф можно представить как ориентированный.

Цикл — путь, в котором начало пути (начало первого ребра) совпадает с концом (конец последнего ребра).

Рассмотрим граф A—B.

Возможны пути

$$[AB, BA]$$

Простой путь — путь, у которого не повторяются ребра (вершины повторятся могут).

Простой цикл — цикл, у которого не повторяются ребра (вершины повторятся могут).

Вернемся к задаче.

Пусть вершина верхняя вершина — старт. Пройдем по ребру и выкинем его (т.е. степень у вершины понизится). Продолжим процесс аналогично. В итоге какой бы путь мы не строили, степени у всех промежуточных вершин понизятся на четное число, а у вершин финиша и начала понизятся на нечетное число. Т.о. это невозможно. Подробнее в вики.

Эйлеров цикл — простой цикл, включающий все ребра графа.

Эйлеров граф — граф, в котором существует Эйлеров цикл.

Полуйэлеров граф — граф, в котором есть Эйлеров путь, но нет Эйлерого цикла.

2.3. Связность графов

Граф является связным, если для $\forall A,B \in V$ существует путь от A к B.

$A \longrightarrow B \longrightarrow C$ — несвязный граф.

Компонента связности — связный подграф, в который включены все вершины исходного, связаные с принадлежащими подграфами. Связный граф имеет 1 компоненту связности. Крайний случай: вершины без ребер. Количество компонент связности от 1 до количества вершин.

Слабая связность графа — "забываем" про направленность графов и смотрим на связность.

Сильно связный граф — граф связан при условии направленности.

"Вес" ребра — некоторая числовая характеристика ребра (расстояние, время прохождения, стоимость, энергия реакции и т.д.). Это необязательно положительное число.

Взвешенный граф — граф, у которого все ребра имеют вес.

2.4. Хранение графа в памяти ПК

Введем понятие: смежные вершины — «соседи», т.е. это вершины, которые имеют общее ребро. Ациклический граф — орграф без цикла.

2.4.1. Формы хранения

Есть 3 основные формы хранения:

1. Список ребер (множество ребер)

AB 5
BC 3
CD 1
DE 2

2. Матрица смежности

Матрица смежности не умеет хранить кратные ребра (если только массив не трехмерный (но это бред)).

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

Можно также составить матрицу взвешенности, если записать в эту матрицу вес каждого ребра.

3. Списки смежности

$A : B$
$B : A, \quad C$
$C : B, \quad D$
$D : C, \quad E$
$E : D$

2.4.2. Реализация на Python

1. Список ребер

Программа №2.4.1.

```
1    G = ('AB', 5),
2        ('BC', 3),
3        ('CD', 1),
4        ('DE', 2)
```

Но кортежами не очень удобно.

Программа №2.4.2.

```
1    G = 'AB' : 5
2        'BC' : 3
3        'CD' : 1
4        'DE': 2
```

Задачи:

(a) Проверка смежности.

(b) Перебор "соседей".

2. Таблица смежности

Программа №2.4.3.

```
1    G = [[0,1,0,0,0],
2        [1,0,1,0,0],
3        [0,1,0,1,0],
4        [0,0,1,0,1],
5        [0,0,0,1,0]]
```

Решение задач:

(a) Проверка смежностей

Программа №2.4.4.

```
G[i][k] == 1 - значит смежные
```

(b) Перебор соседей: нужно пробежать по строке.

3. Списки смежности

Словарь множеств смежностей:

Программа №2.4.5.

```
1    G = {'A':{'B':5}
2         'B':{'A':5, 'C':3}
3         'C': {'B':3, 'D':1}
4         'D': {'C':1, 'E':2}
5         'E': {'D':2}}
```

Проверка факта смежностей:

```
'B' in G['A'] # O(1)
```

Перебор соседей:

```
for v in G['A']:
```

$O(N_{e_{max}})$, N_e - средняя степень вершины.

ЛЕКЦИЯ 3

Обход графа в глубину

3.1. Множества и словари

Задание множеств:

```
A = None # Пустое множество
A = {1, 2 'hello'} # Явное перечисление элементов
A = set('hello') # Множество букв в строке
B = [1, 2, 1, 2]; A = set(B) # Множество из списка или любого итерируемого объекта
```

Работа с элементами множеств:

```
C = {1, 2, 'hello'}
for elem in range(C): # Перебираем все элементы множества
    print(elem)
sorted(C) # Список из отсортированных элементов множества
1 in C # Проверка принадлежности
2 not in C
A.add(3) # Добавление элемента
A.remove(3) # Удаление элемента, который есть в множестве
A.discard(4) # Удаление элемента, которого может и не быть в множестве
A.pop() # Извлечение случайного элемента из множества с удалением его
```

Задание словарей

```
D = {} # Пустой словарь
D = {1: 'a', 2: 'b'} # Явное перечисление
D = dict([(1, 'a'), (2, 'b')]) # Словарь из списка пар элементов
D = dict(zip([1, 2], ['a', 'b'])) # Словарь из итерируемого объекта, возвращающего пары
значений
D = {i: chr(i + ord('a')) for i in range(1, 3)} # Генератор словарей
```

Работа с элементами словаря

```
len(D) # Количество элементов в словаре
D[key] # Поиск по ключу, который есть в словаре
key in D # Проверка принадлежности словарю
D[key] = value # Установка или изменение значения
del D[key] # Удаление ключа, который есть в словаре
value = D.pop(key) # Удаление ключа вместе с возвращением значения
value = D.pop(key, no_key_value)
key, value = D.popitem() # Извлечение из словаря пары (ключ, значение) с удалением ключа
D.get(key, no_key_value) # Значение по ключу, no_key_value, если ключа нет
D[key] = D.get(key, 0) + 1 # Самая простая реализация счетчика
for key in D: # Перебираем все ключи
    print(key, D[key])
for key, value in D.items(): # Перебираем все пары (ключ, значение)
    print(kay, value)
for value in D.values(): # Перебор всех значений
    print(value)
sorted(D) # Отсортированный список ключей
sorted(D.values()) # Отсортированный список значений
sorted(D.items()) # Отсортированный по ключу список пар (ключ, значение
sorted(D.items(), key = lambda x: x[1])
```

3.2. Реализация записи графа

Будем задавать граф как список ребер. Количество вершин, потом количество ребер.

5 6

После зададим ребра:

0 1
0 2
1 2
1 3
2 3
0 4

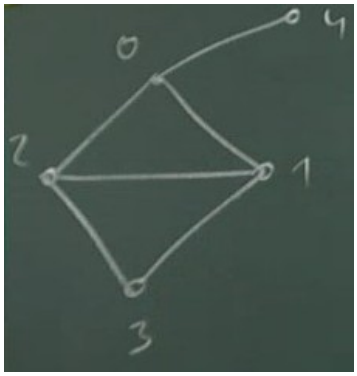

Рис. 3.1. Пример графа

Программа №3.2.1. Считывание графа как матрицы и как списка смежностей

```
1   def read_graph_as_matrix():
2       N, M = [int(x) for x in input().split()]
3       graph = [[0]*N for i in range(N)] # матрица смежностей
4       for edge in range(M):
5           a, b = [int(x) for x in input().split()]
6           graph[a][b] = 1
7           graph[b][a] = 1
8       return graph
9
10  def print2d(A):
11      for line in A:
12          print(*line)
13      print()
14
15  def read_graph_as_lists():
16      N, M = [int(x) for x in input().split()]
17      graph = [[] for i in range(N)]
18      for edge in range(M):
19          a, b = [int(x) for x in input().split()]
20          graph[a].append(b)
21          graph[b].append(a) # Для ориентированного графа строка не нужна
22      return graph
23
24  graph = read_graph_as_lists()
25  print2d(graph)
```

В итоге `read_graph_as_matrix()` даст нам такой результат:

$$\begin{matrix} 0 & 1 & 1 & 0 & 1 \\ 1 & 0 & 1 & 1 & 0 \\ 1 & 1 & 0 & 1 & 0 \\ 0 & 1 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \end{matrix}$$

a `read_graph_as_lists()`:

1 2 4
0 2 3
0 1 3
1 2
0

3.3. Алгоритм обхода графа в глубину

3.3.1. Алгоритм

Перебираем соседей по часовой стрелке. Граф считаем неориентированным.

Основное правило: для того, чтобы пойти на праздник надо вначале позвать всех своих друзей на праздник, но только тех, кто еще не позван.

Рис. 3.2. Граф с выбранным началом

Далее по часовой стрелке от 12 часов выбираем следующую вершину и зовём ее на праздник. Будем отмечать порядок обхода (номера показывают не индексы в графе, а просто порядок вызова).

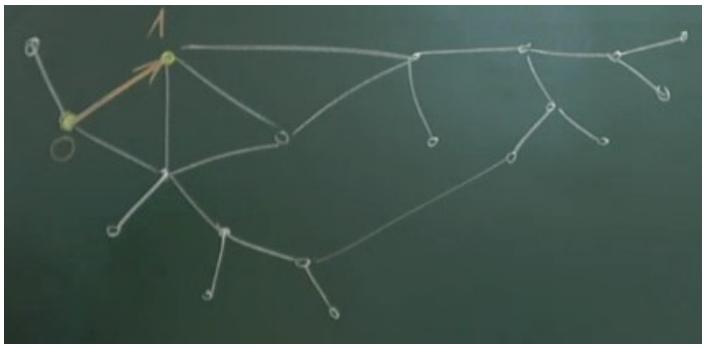

Рис. 3.3. Точка 1 перекрашена в "серый" цвет

Теперь вершину 1 красим в "серый" цвет. 0-ой ждёт ответа от 1-го.

1-ый начинает перебирать всех своих не позванных соседей по часовой стрелке от 12 часов.

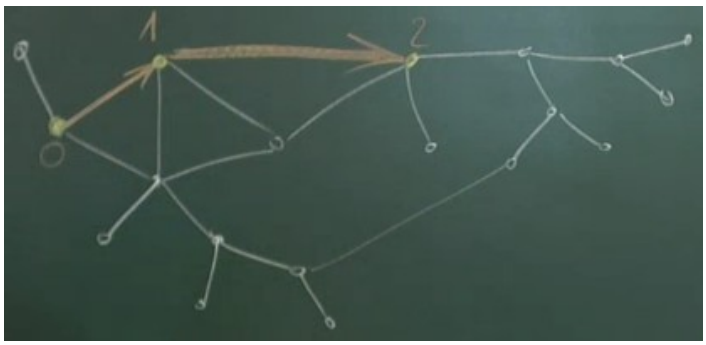

Рис. 3.4. Точка 2 перекрашена в "серый" цвет

И так процесс продолжается.

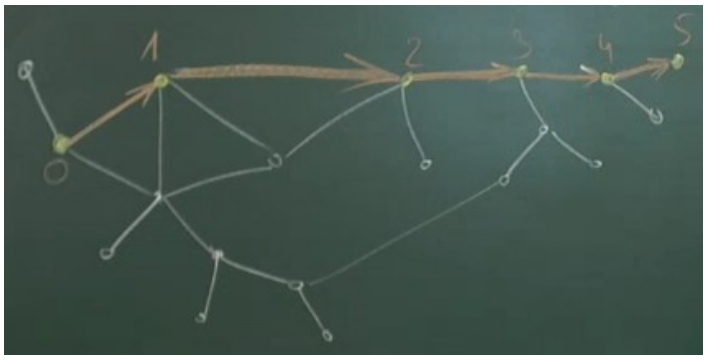

Рис. 3.5. Точки 0-5 перекрашены в "серый" цвет

Теперь 5-ый перебирает всех своих соседей. Но у него только один сосед — это 4-ый, и он уже позван. Т.о. 5-ый уже всех позвал. 5-я точка перекрашивается в "чёрный" цвет и идёт на праздник. 4-му возвращается от 5-го команда, что 5-ый всех позвал. Дальше 4-ый продолжает звать друзей и зовёт 6-го. 6-ая точка перекрашивается в "серый" цвет.

6-ой перебирает всех друзей и убеждается, что он всех позвал. Точка перекрашивается в "чёрный" цвет и идёт на праздник. 4-му возвращается команда, что 6-ой позвал всех друзей.

Т.о. 4-ый тоже позвал всех друзей. Точка перекрашивается в "чёрный" и дает команду 3-ему.

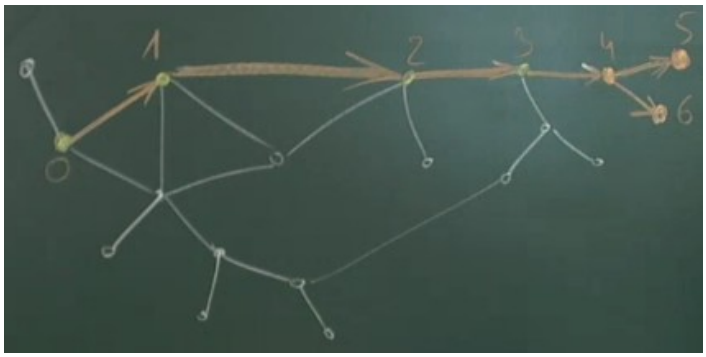

Рис. 3.6. Точка 4 перекрашена в "чёрный" цвет

3-ий зовёт оставшихся.

Процесс повторяется...

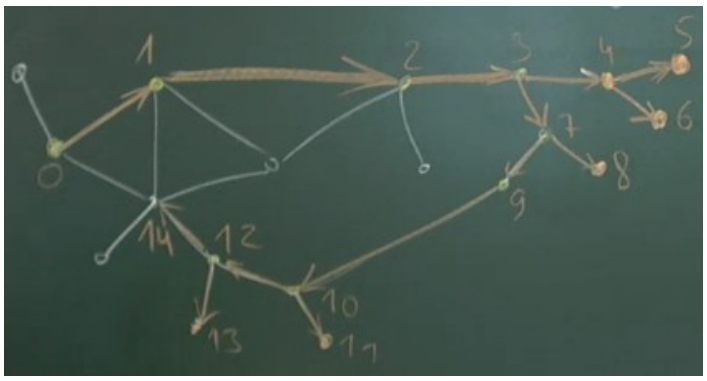

Рис. 3.7. Дошли до 14

14-ый позвал бы 1-го, но он уже позван. Заметим, что из серой вершины мы пытаемся позвать серую вершину, что значит, что в графе есть цикл.

Т.о. 14-ый зовёт следующего по часовой стрелке. 15 точка перекрашивается в "серый". 15-му звать некого, поэтому точка перекрашивается в "чёрный" и дает сигнал 14-му.

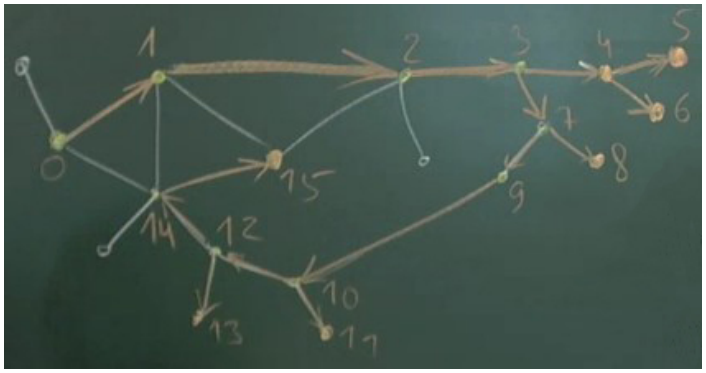

Рис. 3.8. Прошли 15-го

14-ый продолжает звать друзей, зовёт 16-го, 16-ый возвращает сигнал 14-му.

14-му становится некого звать. Он возвращает сигнал 13-му и перекрашивается в "чёрный". 13 возвращает сигнал 12-му и т.д. до 2-го и все они идут на праздник.

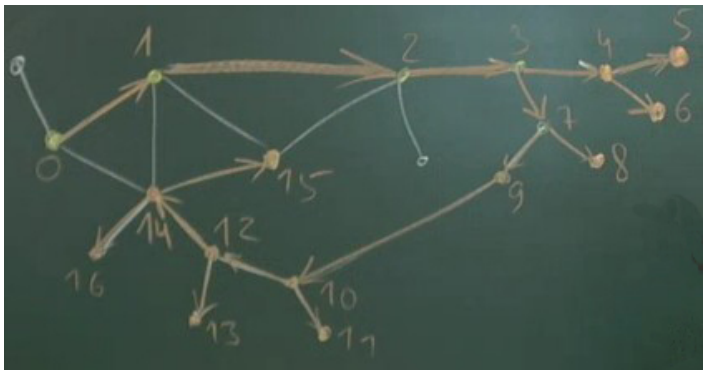

Рис. 3.9. Длинный возврат

2-ой зовёт 17-го, ему в свою очередь звать некого, он идёт на праздник и возвращает сигнал 2-му. 2-ой тоже всех позвал. В итоге сигнал возвращается 0-му, который зовёт 18-го. В итоге, все

точки перекрашены в "чёрный" цвет.

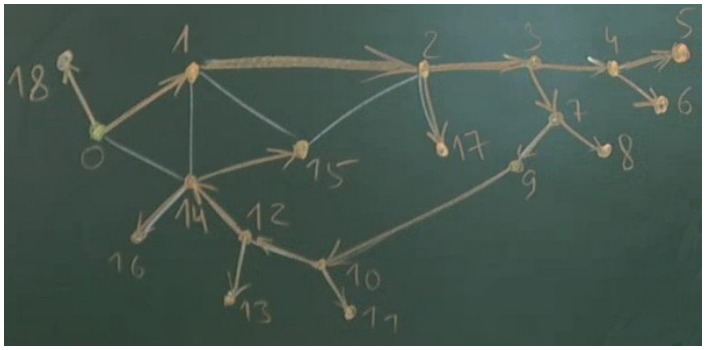

Рис. 3.10. Все позваны

Применение обхода в глубину:

1. Проверка связности графа (будем добавлять вершины в множество пройденых вершин)
2. Выделение компонентов связности
3. Поиск циклов (проверка ацикличности графов)
4. Поиск компонент сильной связности орграфов (алгоритм Косарайю)

3.3.2. Реализация на Python

Программа №3.3.1. Построение алгоритма

```
1    def read_graph_as_lists():
2        N, M = [int(x) for x in input().split()]
3        graph = [[] for i in range(N)]
4        for edge in range(M):
5            a, b = [int(x) for x in input().split()]
6            graph[a].append(b)
7            graph[b].append(a)
8        return graph
9
10   def call_all_friends(me, friends, already_called = None):
11       if already_called is None:
12           already_called = set()
13       """ Правило: тебя позвали на праздник, но пойти можно только тогда, когда
             позовешь всех своих еще не позванных друзей
         """
14       already_called.add(me)
```

```
15        for friend in friends[me]:
16            if friend not in already_called:
17                print(friend, 'был позван на праздник')
18                call_all_friends(friend, friends, already_called)
19                print(friend, 'пошел на праздник')
20
21    graph = read_graph_as_lists()
22    call_all_frineds(0, graph)
```

Запишем теперь более строго:

Программа №3.3.2. Реализация алгоритма обхода графа в глубину

```
 1    def dfs(vertex, graph, used = None): # Depth-first search
 2        if used is None:
 3            used = set()
 4        used.add(vertex)
 5        for neighbour in graph[vertex]:
 6            if neighbour not in used:
 7                dfs(neighbour, graph, used)
 8
 9    graph = read_graph_as_lists()
10    used = set()
11    number_of_components = 0
12    for vertex in range(len(graph)): # Подсчет компонент связности
13        if vertex not in used:
14            dfs(vertex, graph, used)
15            number_of_components += 1
16
17    print('Количество компонент связности:', number_of_components)
```

3.4. Дерево

Дерево — связный граф, в котором

1. Нет простых циклов
2. От a к b только один путь
3. $N_{\text{вершин}} = M_{\text{ребер}} + 1$

Ориентированное дерево — ациклический орграф, в котором только одна вершина имеет нулевую степень захода (корень). Вершины с нулевой степенью исхода называются "листья". Остальные — узлы ветвления. Для корневого дерева можно ввести уровень узла — длина пути от корня до вершины (уровень иерархии).

У любого связного графа есть подграф, являющийся деревом и содержащий все исходные вершины.

Вершина сама по себе тоже является деревом.

Остовное дерево — пограф исходного графа, в котором выброшено максимальное количество ребер так, чтобы связность еще сохранилась. Обход графа в глубину позволяет построить одно из остовных деревьев.

Свойства:

- Дерево не имеет кратных ребер и петель
- Граф является деревом $\Leftrightarrow$ когда любые две различные вершины можно соединить единственным простым путем (простой цепью).
- Любое дерево однозначно определяется расстояниями между его концевыми вершинами со степенью 1 (длиной наименьшей цепи).
- Любое дерево, множество вершин которого более чем счетное является планарным графом.

Шарнир — вершина, при удалении которой количество компонент связности увеличивается.

Мост — ребро, при удалении которого количество компонент связности увеличивается. Такие ребра еще известны как разрезающие ребра или перешейки.

У дерева любая вершина является шарниром, а любое ребро мостом.

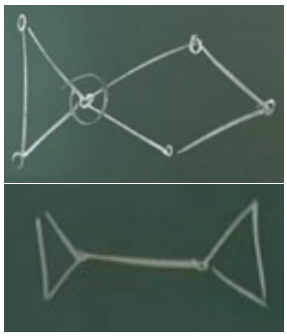

Рис. 3.11. Шарнир и мост

3.5. Алгоритм Косарайю

Задача: поиск сильно связной компоненты орграфа. Возьмем такой орграф. В нем будет 1 слабая компонента.

Запустим обход в глубину. Пусть будет множество вершин, которых мы уже использовали, и список вершин в порядке обхода (типа как стек).

Начнем с вершины A. Из A вызываем B. Они оказываются в множестве использованных вершин, также заполняется стек вызванных вершин. Далее вызовем вершину D (не принципиально D или C), добавим ее в множество и стек аналогично. Далее вызовем E, потом F. После происходит откат, и мы вызываем вершину C. Дальше выбираем любую вершину, которую не использовали. Пусть это будет G. Потом вызываем H, I, J. Остается вершина K, она становится последней.

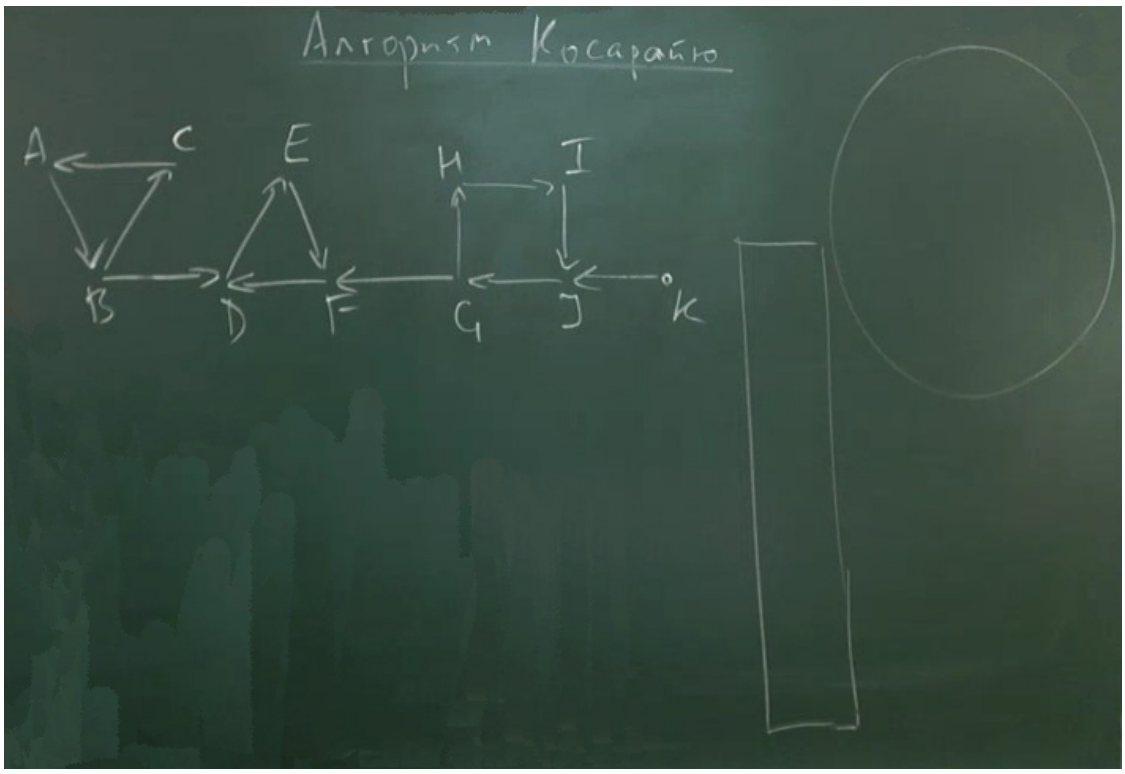

Рис. 3.12. Орграф

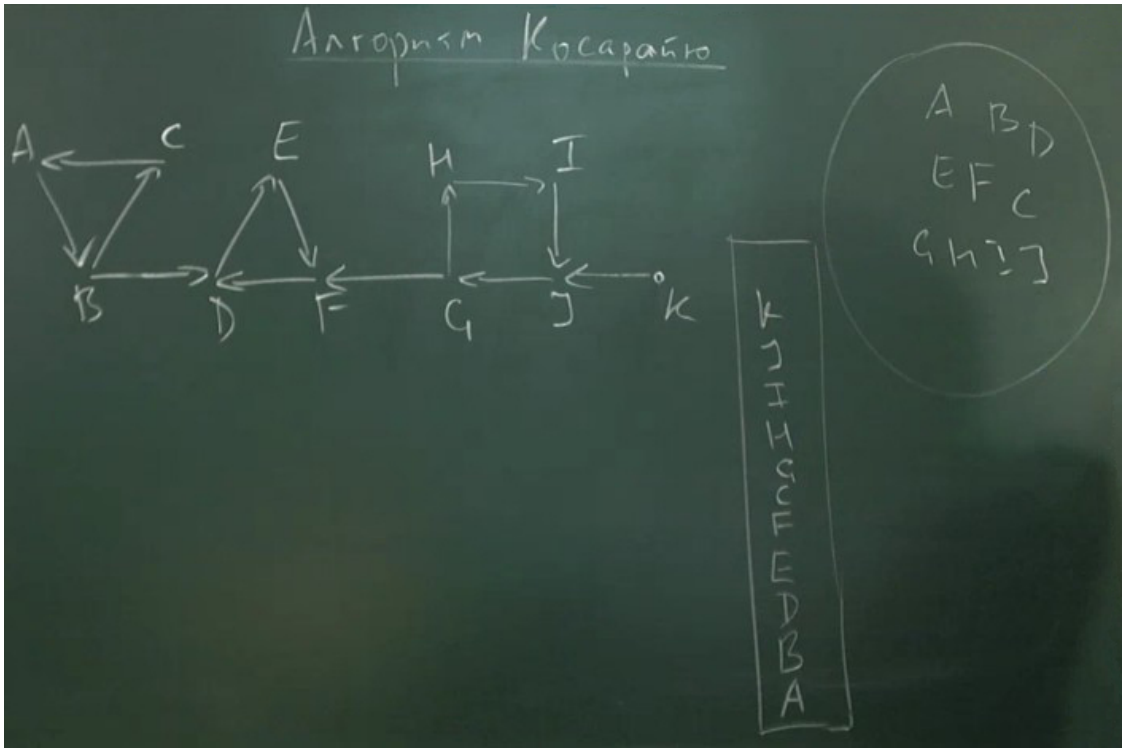

Рис. 3.13. После прохождения по графу

Далее мы разворачиваем наш граф, т.е. меняем все направления. А теперь от верхней вершины в стеке запустим обход в глубину на обращенном графе.

Но от K дойти никуда нельзя. Это и есть сильная компонента. K добавляем в новое множество использованных вершин.

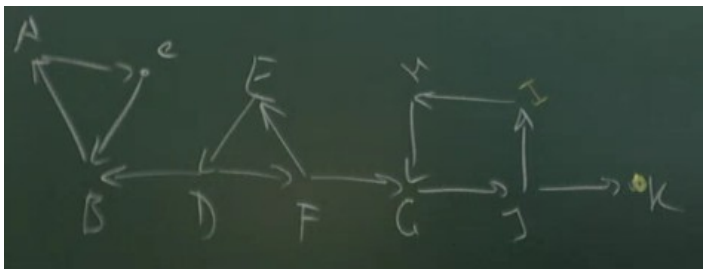

Рис. 3.14. Обращенный граф

Дальше начинаем обход с вершины I, т.е. I—>H—>G—>J. Их добавляем в множество использованных и стираем из стека. Дальше идти опять некуда. Значит, это вторая сильная компонента.

Переходим к вершине C. Выполняем обход: C—>B—>A. Их добавляем в множество использованных и стираем из стека. Дальше идти некуда. Т.о. это еще одна компонента сильной связности.

Переходим к вершине F. В G уже не идем т.к. она использована. Выполняем обход F—>E—>D. Их добавляем в множество использованных и стираем из стека. Дальше идти некуда. Т.о. это еще одна компонента сильной связности.

В итоге получилось 4 компонент сильной связности.

Примечание: не принципиально, оборачивать ли граф сначала или потом (по википедии сначала нужно развернуть граф).

ЛЕКЦИЯ 4

Обход графа в ширину (BFS)

4.1. Визуализация алгоритма BFS (пожар на графе)

Данный алгоритм можно сравнить с разгорающимся лесом (это отразилось на названии переменных в коде, см. ниже). Зажжем первую вершину (выбираем какую-то стартовую). От нее, как по мостикам, по ребрам графа огонь (наш алгоритм) переходит на соседние вершины. После этого сами соседние вершины уже зажигают своих соседей. При этом, чтобы наш алгоритм постоянно не записывал уже зажженные вершины, нам нужно, чтобы вершины «догорали» (необходимо убирать вершины, соседей которых уже проверили, из списка зажженных вершин). Т.е. этот алгоритм вкратце можно описать так:
1) Вершина графа загорается.
2) Передает огонь на все своих соседей.
3) Догорает.
И так происходит до того момента, пока мы не пройдем по всем соседям.

Анимация BFS

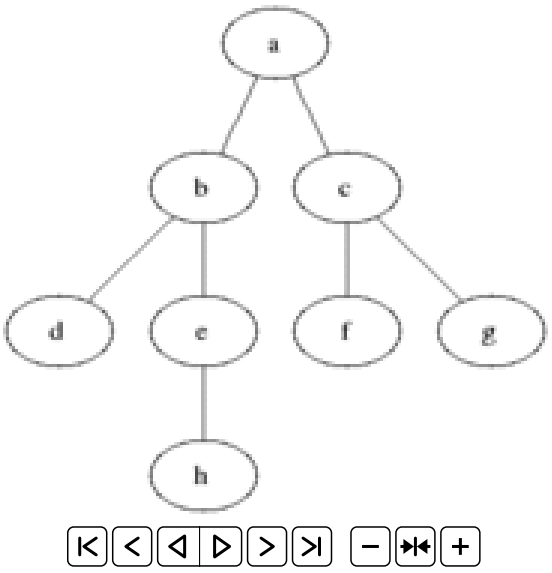

Рис. 4.1. Белый — вершина, которая еще не обнаружена. Серый — вершина, уже обнаруженная и добавленная в очередь. Черный — вершина, извлечённая из очереди.

Более подробная визуализация изложена в википедии.

Прагматический смысл обхода в ширину: *алгоритм позволяет находить расстояния на неориентированном графе от каждой вершины.* При этом остовное дерево BFS имеет наименьший диаметр для данной точки A.

Применение алгоритма:

- Построение остовного дерева
- Поиск расстояния (для не взвешенного графа)

- Обнаружение циклов наименьшей длины (когда происходит попытка зажечь вершину, которая уже горит, причем не ту, из которой позвали)
- Подсчет компонент связности

4.2. Реализация алгоритма BFS

Программа №4.2.1. Реализация алгоритма BFS

```
1    def bfs_fire(G, start, fired = None):
2        if fired is None:
3            fired = set()
4        fired.add(start)
5        time = {start: 0} # Хранение времен их добывания
6        Q = [start]
7        while Q:
8            current = Q.pop(0) # Для списка это не эффективно
9            for neighbour in G[current]:
10               if neighbour not in fired:
11                   fired.add(neighbour)
12                   Q.append(neighbour)
13                   print(current, neighbour) # Для построения остовного дерева
14                   time[neighbour] = time[current] + 1
```

Построчный комментарий кода:
2) Задаём `fired` как пустое множество (по умолчанию этого делать нельзя, т.к. тогда это станет глобальной переменной).
4) Добавляем стартовую вершину в «догоревшие».
5) Хранение времени, за которое мы доходим до вершины (это позволяет находить расстояние до вершины от стартовой).
6) Добавление вершины в очередь «горящих».
7) Цикл работает, пока есть хотя одна горящая вершина.
8) Берем первую вершины из очереди.
9) Проходим по всем соседям вершины.
11) Добавляем нашу вершину в очередь «зажженных».
12) Добавляем в «догоревших» (чтобы снова не записывать его в очередь «зажженных»).
13) Это позволяет получить остовное дерево.
14) Так и получаем расстояние до вершины.

4.3. Алгоритм Дейкстры

Данный алгоритм необходим для поиска кратчайшего маршрута от исходной вершины до всех вершин графа. Его метод работы основан на обходе графа в ширину (как мы помним, данный алгоритм позволяет находить расстояния в невзвешенном графе). Идея аналогичная — «зажигаем» вершины, после этого проходим по соседям «зажженной» вершины. Но теперь в нашем проходе появляется определенный порядок — вначале мы идем к вершинам, путь к которым наиболее краток. Это позволяет раньше добавить вершины в список «сгоревших» и закончить вычислять длину маршрута к ним.

Подробная визуализация алгоритма изложена в википедии.

Условие на граф: не должно быть ребер с отрицательным весом.

4.4. Реализация алгоритма Дейкстры

Программа №4.4.1. Реализация алгоритма Дейкстры (не эффективен)

```
1    def dijkstra(G, start): # G - словарь словарей с весами
2        d = {v: float('+inf') for v in G}
3        d[start] = 0
4        used = set()
5        while len(used) != len(G):
6            min_d = float('+inf')
7            for v in d:
8                if d[v] < min_d and v not in used:
9                    current = v
10                   min_d = d[v]
11           for neighbour in G[current]:
12               l = d[current] + G[current][neighbour]
13               if l < d[neighbour]:
14                   d[neighbour] = l
15           used.add(current)
16       return d # Алгоритм не эффективен
```

Построчный комментарий кода:

1) G — словарь словарей (каждой вершине соответствует список вершин, каждой из которых поставили в соответствие длину ребра).

2) Задаем изначальный словарь длин расстояний до вершин.

3) Задаем длину расстояния до исходной вершины (она ноль, очевидно).

4) Задаем множество пройденных вершин.

5) Пока не все вершины использованы.
6) Задаем расстояние (делаем его бесконечным, чтобы потом можно было найти минимальное).
7-10) Находим вершину с минимальным путем до нее. При первом проходе это будет начальная вершина (до нее путь ноль). При втором проходе она уже будет смотреть и выбирать из соседей исходной вершины и т.д.
11-14) Пробегаемся по соседям нашей вершины и рассчитываем путь до нее. Если он меньше того пути, который сейчас соответствует этой вершине, то мы записываем новое значение — это путь до той вершины, по соседям которой мы пробегаемся плюс путь от нее до соседа.
15) Добавляем проверенную вершину в использованное.

Программа №4.4.2. Реализация алгоритма Дейкстры

```
1   from heapq import*
2   def dijkstra(G, start): # G - словарь словарей с весами
3       d = {v: float('+inf') for v in G}
4       d[start] = 0
5       Q = [(0, start)]
6       used = set()
7       while len(used) != len(G):
8           d_c, current = heappop(Q)
9           if d_c != d[current]:
10              continue
11          for neighbour in G[current]:
12              l = d[current] + G[current][neighbour]
13              if l < d[neighbour]:
14                  d[neighbour] = l
15                  heappush(Q, (l, neighbour))
16          used.add(current)
17      return d
```

Данная программа имеет такую же схему работы, как и предыдущая, но для большей эффективности использована пирамида кортежей.

ЛЕКЦИЯ 5

Задачи на графы

5.1. Поиск минимального остовного дерева

Минимальное остовное дерево (для связного графа) — остовное дерево минимального суммарного веса.
Остовных деревьев может быть несколько. Есть 2 алгоритма поиска минимального остовного дерева: Прима и Краскала. Оба алгоритмы жадные. В первом случае мы выхватываем ребро, которое короче. Связность появится в последний момент. Во втором случае мы постоянно поддерживаем связанность. Условие: связный неориентированный граф.

5.1.1. Алгоритм Краскала

Алгоритм

1. Упорядочиваем все ребра по возрастанию веса (не убыванию).
2. Далее добавляем ребро по порядку к каркасу (вершины добавляются вместе с ребром), если оно не образует цикл с уже имеющимися ранее ребрами.

Подробная визуализация разобрана на википедии.

Реализация

Программа №5.1.1. Реализация алгоритма Краскала

```
1    N, M = [int(x) for x in input().split()]
2    edges = []
3    for i in range(M):
4        v1, v2, weight = map(int, input().split())
5        edges.append((weight, v1, v2)) # Сначала будем добавлять вес
6    edges.sort()
```

```
 7    comp = list(range(N))
 8    tree = []
 9    tree_weight = 0
10    for weight, v1, v2 in edges:
11        if comp[v1] != comp[v2]:
12            tree.append((v1, v2))
13            tree_weight += weight
14            for i in range(N):
15                if comp[i] == comp[v2]:
16                    comp[i] = comp[v1]
```

Сложность алгоритма $O(M \log M + M \cdot N)$.

Построчный комментарий кода:

1) Вводим количество вершин и ребер.

2) Создаем список для ребер.

4) Вводим ребра — вершины, которые они соединяют и вес.

5) Добавляем в список ребер.

6) Сортируем по весу ребер.

7) Создаем список ребер, чтобы работать с индексами и проверять, из одной они компоненты связанности или нет.

8) Сам массив ребер, образующих остовное дерево.

9) Вес графа.

10) Пока не прошли по всем ребрам.

11) Если вершины не принадлежат одной компоненте связанности.

12) Добавляем ребро к остовному дереву.

13) Добавляем в общий вес вес нового ребра.

14) Проходим по всем вершинам.

15-16) Если мы встречаем аналогичную второй вершине компоненту связанности,то перекрашиваем ее в компоненту связанности первой. Так свяжем дерево.

5.1.2. Алгоритм Прима

Алгоритм

1. Выбираем произвольную вершину (она и есть заготовка для дерева).

2. Добавляем к каркасу ребро минимального веса среди ребер, инцидентных какой-либо вершине каркаса и вершине не из каркаса.

Реализация

`dist[i]` — минимальный вес ребра, которым можно присоединить вершину i к каркасу.

Программа №5.1.2. Реализация алгоритма Прима

```
INF = 10**9 # Введем условную бесконечность
dist = [INF]*N # W[i][j] - вес ребра ij, который равен +бесконечность,
если i не смежна j
dist[0] = 0
used = [False]*N
used[0] = True
tree = []
tree_weight = 0
for i in range(N):
    min_d = INF
    for j in range(N):
        if not used[j] and dist[j] < min_d:
            min_d = dist[j]
            u = j
    tree.append((i, u))
    tree_weight += min_d
    used[u] = True
    for v in range(N):
        dist[v] = min(dist[v], W[u][v])
```

Асимптотика алгоритма: $O(N^2)$. С помощью кучи можно ускорить до $O((M+N)\log N)$.

5.2. Алгоритм построения Гамильтонова цикла

Гамильтонов цикл — цикл, проходящий через все вершины по одному разу. Гамильтонов путь — путь, проходящий через все вершины по одному разу.
Это NP – сложная задача, т.к. решение задачи находится перебором, $O(N!)$.

Программа №5.2.1. Реализация

```
visited = [False]*N
path = []
def hamilton(curr):
    path.append(curr)
    if len(path) == N:
```

```
6           if A[path[-1]][path[0]] == 1: # A - таблица смежности
7               return True
8           else:
9               path.pop()
10              return False
11      visited[curr] = True
12      for next in range(N):
13          if A[curr][next] == 1 and not visited[next]:
14              if hamilton(next):
15                  return True
16      visited[curr] = False
17      path.pop()
18      return False
```

ЛЕКЦИЯ 6

Алгоритмы на графах

6.1. Алгоритм Флойда-Уоршела

6.1.1. Описание алгоритма

Задача алгоритма: *нахождение кратчайших расстояний между всеми вершинами взвешенного ориентированного графа.*

Алгоритм: на каждом шаге проверяем новое количество вершин, т.е. на первом шаге имеем путь между двумя соседними вершинами — ребро, соединяющее вершины. На втором шаге выбираем какую-то вершину и смотрим, будет ли короче путь между двумя изначальными вершинами, если пройти через эту выбранную точку. Далее выбираем следующую точку и т.д. Таким образом мы постепенно исследуем все пути из вершины в вершину через постепенно растущее число вершин.

Строим последовательность матриц (методом динамического программирования) $a_{ij}^0 \to a_{ij}^1 \to a_{ij}^2 \to \cdots \to a_{ij}^n$. a_{ij}^k — кратчайшее расстояние от i-ой до j-ой вершины, при этом как промежуточными пользуемся от 1-ой до k-ой.
$a_{ij}^0 = W_{ij}$ — промежуточными вершинами пользоваться нельзя.
Правило поиска следующей матрицы: $a_{ij}^k = \min(a_{ij}^{k-1}, a_{ik}^{k-1} + a_{kj}^{k-1})$.
Сложность алгоритма: $O(N^3)$.

Более подробно алгоритм описан в википедии.

6.1.2. Реализация

Будем запоминать все предыдущие матрицы (что необязательно).

Программа №6.1.1. Реализация алгоритма Флойда-Уоршела

```
1    A = [[[INF]*n for i in range(n)] for k in range(n+1)] # INF - условная
     бесконечность, n - число ребер
2    for i in range(n):
```

```
3       A[0][i][:] = W[i] # При копировании весовой матрицы W расстояние от вершины
    до себя равно нулю; забиваем матрицу рёбер т.е. расстояния в начальный момент.
4   for k in range(1, n+1):
5       for i in range(n):
6           for j in range(n):
7               A[k][i][j] = min(A[k-1][i][j], A[k-1][i][k]+A[k-1][k][j]) # Добав-
    ляем путь от i до j вершины через новую вершину, если такой путь короче
```

Алгоритм не работает с циклами отрицательного веса, т.к. можно «накрутить» минус бесконечность. Но если при этом путь от i-ой до j-ой вершины не содержит такого цикла, то алгоритм сработает правильно.

6.2. Топологическая сортировка

Если граф не содержит циклов, то его вершины можно пронумеровать так, что любое ребро идет от вершины с меньшим номером к вершине с большим номером.

Топологическая сортировка — упорядочивание вершин бесконтурного ориентированного графа согласно частичному порядку, заданному ребрами орграфа на множестве его вершин.

Т.е. если граф не содержит циклов, то его вершины можно пронумеровать так, что любое ребро идет от вершины с меньшим номером к вершине с большим номером.
Применяется в различных прикладных задачах.

6.2.1. Алгоритм Кана

A(k) — множество вершин, от которых «зависит» вершина k (т.е. в которую можно прийти от других вершин).
P — последовательность вершин.
Алгоритм:
Пока $|P|$ < N:
    Найти вершину v, у которой A(v) = $\varnothing$
    P.append(v)
    Вычеркнуть v из всех множеств A(k).
Алгоритм очень громоздкий и неэффективный.

6.2.2. Алгоритм Тарьяна

Алгоритм

Сложность алгоритма: $O(n)$.
По факту мы просто осуществляем обход в глубину, в котором мы будем красить вершины.

Подход алгоритма: DFS с покраской вершин (белая/серая/черная).
С любой вершины не used вершины запускаем DFS. Белые вершины — еще не тронутые, серые вершины — те, в которые алгоритм вошел, черные вершины — те, из которого алгоритм вышел. Попытка входа в серую вершины означает наличие цикла, значит алгоритм невозможен. При выходе из вершины (в момент покраски ее в черный цвет) добавляем ее в начало списка.

Реализация алгоритма

Программа №6.2.1. Реализация алгоритма Тарьяна

```
1    Visited = [False]*(n + 1)
2    Ans = []
3
4    def DFS(start):
5        Visited[start] = True
6        for u in V[start]:
7            if not Visited[u]:
8                DFS(u)
9        Ans.append(start)
10
11   for i in range(1, n + 1):
12       if not Visited(i):
13           DFS(i)
14   Ans = Ans[::-1]
```

6.3. Неэффективные алгоритмы

6.3.1. Задача Коммивояжера

Задача: *найти минимальный по весу гамильтонов цикл. Сложность алгоритма:* $O(N!)$.

В графе есть вершины. Коммивояжер захотел посетить все эти вершины и вернутся домой. Гамильтонов цикл точно есть в этой системе (это легко проверить, соединив все вершины по порядку). То есть, задача сводится к тому, чтобы найти минимальный по весу Гамильтонов цикл. Но эта задача плохо реализуется при большом числе вершин, так как основа решения — перебор. Т.е. имеем $N!$ потенциально возможных гамильтоновых циклов, каждый имеет свой вес.

К примеру, на рис. 6.1 изображен оптимальный маршрут коммивояжёра через 15 крупнейших городов Германии. Указанный маршрут является самым коротким из всех возможных 43 589 145 600 вариантов.

Таким образом оптимизировать эту задачу невозможно.

Рис. 6.1. Оптимальный маршрут коммивояжёра через 15 крупнейших городов Германии

6.3.2. Задача о китайском почтальоне

Задача: *пройти по каждому ребру графа минимум 1 раз, чтобы доставить почту. Требуется найти такой цикл минимального суммарного веса.*

То есть, нам необходимо найти цикл минимального суммарного веса, такой, что он проходит по ребру хотя бы 1 раз. Важное замечание — если граф Эйлеров, то Эйлеров цикл и есть решение этой задачи. А вот если его нет, то задача усложняется — решение можно найти только полным перебором. Итоговый алгоритм получается неэффективным, т.к. его асимптотика $O(N!)$.

ЛЕКЦИЯ 7

Деревья. Двоичное дерево поиска.

7.1. Двоичное дерево. Класс дерево.

k-ичное дерево — дерево, в котором количество дочерних вершин у каждой вершины не больше k штук. При этом троичное дерево является одновременно и четверичным — просто четвертого ребра еще нет. Последовательность дочерних вершин может быть неупорядоченной. Нам же интересен случай, когда k-ичное дерево является упорядоченным.

Рассмотрим двоичное дерево, упорядоченное, в общем случае не сбалансированное (в отличии от кучи). Двоичное дерево можно нарисовать так, что мы будем называть дочерние вершины "левая" или "правая"(по аналогии с кучей).

Поддерево (правое/левое) — подграф дерева, корень которого является дочерней вершиной (правой/левой).

В прошлом семестре вводился такой способ хранения данных как односвязный список. Оказывается, можно реализовать такое звено, которое подходит для двоичного дерева поиска. Сделаем 2 указателя: на левое поддерево и правое поддерево, также указатель наверх (т.е. на родителя). Из таких звеньев можно собирать дерево.

Удобно считать пустой граф пустым деревом (хотя по определению дерева это неверно).

Программа №7.1.1. Класс Дерево

```
1    class Node:
2        def __init__(self, key, value): # Создаем звено ключ, значение
3            self.key = key
4            self.value = value
5            self.parent = None
6            self.left = None
7            self.right = None
8
9    class Tree:
10       def __init__(self):
```

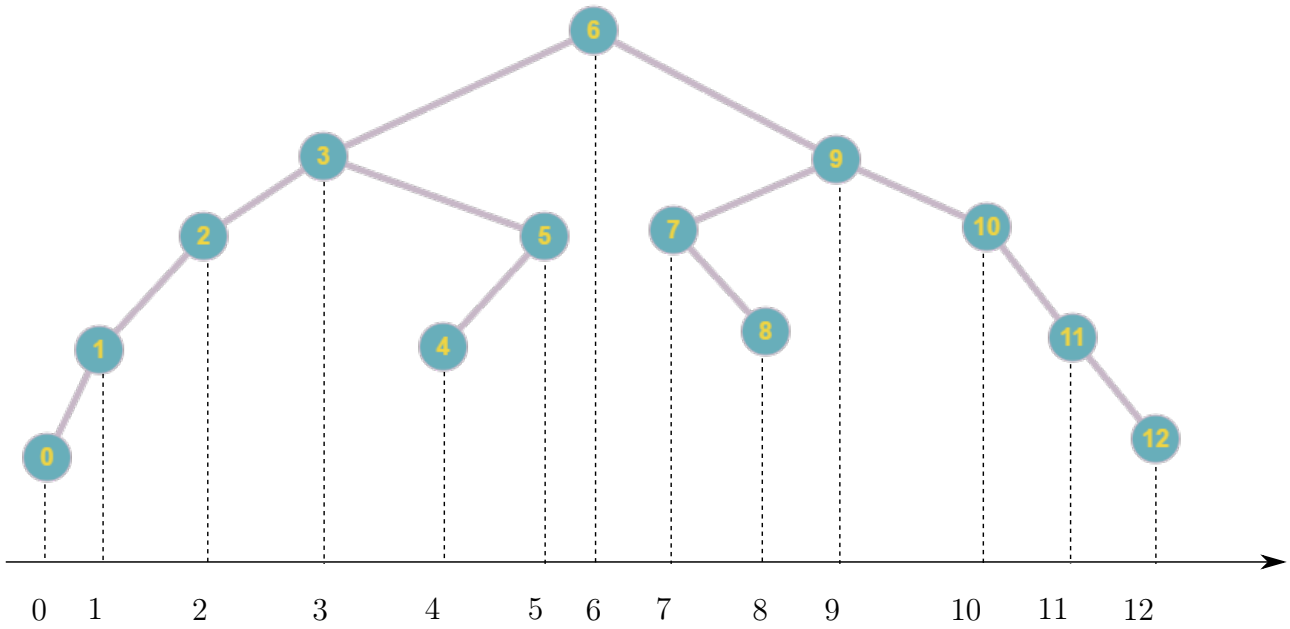

Рис. 7.1. Двоичное дерево поиска

```
11            self.root = None
12        def print(self, node):
13            if node is None: # Благодаря этому нам не важно, пустое наше поддерево
      или нет - крайний случай проверен
14                return
15            self.print(node.left)
16            print((node.key, node.value)) # Это не совсем обратный ход рекурсии
17            self.print(node.right)
```

7.2. Двоичное дерево поиска

Двоичное дерево поиска — это корневое двоичное упорядоченное дерево, построенное по следующему правилу для любого звена node: все ключи левого поддерева $key_i < key_{node}$, все ключи правого поддерева $key_j > key_{node}$.

Возьмем такие числа:

6 3 5 4 2 9 7 8 1 11 10 12 0

и изобразим для них двоичное дерево поиска (рис. 7.1).

Алгоритм построения: сначала дерево пустое и ни одного звена нет. Берем числа поочереди. 3 меньше 6, поэтому 6 становится главной. 3 становится левым поддеревом шестерки, т.к. $3 < 6$ (добавляем числа меньшие корня в левое поддерево, а числа большие корня в правое поддерево). Далее число 5 меньше 5, но больше 3. Поэтому 5 находится в левом поддереве 6, но в правом поддереве тройки. Дальнейшее построение аналогично. Красота такого метода в том, что если

«спроектировать» числа на прямую, получается числовая ось, на которой числа расставлены в порядке возрастания.

Двоичное дерево поиска работает как бинарный поиск. Количество операций сравнения равно высоте дерева $O(\log_2 N)$ (если дерево сбалансировано). Чем оно лучше бинарного поиска в списке? Для добавления элемента требуется то же время ($O(\log_2 N)$). Но в списке после того, как мы нашли, куда вставить элемент, требуется сделать циклический сдвиг.

Если у элемента нет дочерних вершин, а его надо удалить, то это сделать просто. Но вот если у него есть одна дочерняя вершина, мы присоединяем оставшееся дерево к верхнему родителю (можно привести аналогию с подчиненными: если начальника подчиненных уволили, то эти подчиненные становятся подчиненными начальника рангом выше).

В случае когда нужно присоеденить 2 дерева (т.е. если мы удалили вершину, у которого было 2 поддерева, например третью) переходим к левому поддереву удаляемой вершины и к самому правому из него добавляем правое поддерево той вершины, которую мы удалили. Минусы: при удалении корневого элемента длина дерева удваивается.

Но есть более оптимальный вариант: после удаления вершины (например тройки) возьмем самый правый элемент левого поддерева этой вершины (т.е. тройки) и поставим его вместо элемента, которого мы удалили (т.е. вместо стройки), что предотвратит от удваивания длины всего дерева (т.е. если бы у двойки было бы правое поддерево, мы бы нашли самый большой элемент в этом поддереве и поставили бы его на место тройки).

Почему не воспользоваться хеш–таблицей? Хеш–таблица не упорядочена, в отличии от двоичного дерева поиска.

ЛЕКЦИЯ 8

Двоичное дерево поиска. Продолжение.

8.1. Класс Дерево. Продолжение.

Программа №8.1.1. Класс Дерево

```
1    class Tree:
2        class Node: # Класс в классе можно делать, т.к. в классе описано свое
         пространство имен
3            def __init__(self, data):
4                self.parent = None
5                self.left = None
6                self.right = None
7                self.key = data
8        def __init__(self):
9            self.root = None
10       def find(self, data):
11           p = self.root
12           while p is not None and p.key != data: # Важна последовательность
         написаний условий, т.к. может быть ошибка при проверке ключа
13               if data > p.key:
14                   p = p.right
15               else:
16                   p = p.left
17           return p
18       def insert(self, data):
19           p = self.find(data) # Одно и то же число не может храниться дважды,
        как в множестве, но значения могут повторяться
20           if p is not None:
21               return
```

```
22            node = Tree.Node(data)
23            if self.root is None:
24                self.root = node
25                return
26            p = self.root
27            while True:
28                if data < p.key:
29                    if p.left is None:
30                        p.left = node
31                        node.parent = p
32                        break
33                    else:
34                        p = p.left
35                else:
36                    if p.right is None:
37                        p.right = node
38                        node.parent = p
39                        break
40                    else:
41                        p = p.right
```

8.2. Балансировка дерева

Двоичное дерево поиска сбалансированно, если для каждой его вершины высота левого и правого поддерева отличаются не более чем на единицу.

Инвариант: та вершина, которая левее других вершин, должна остаться левее всех вершин, т.е двигать вверх–вниз вершины можно, но влево–вправо двигать нельзя, иначе нарушится последовательность чисел.

Алгоритм балансировки подробно описан в википедии.

АВЛ–дерево — сбалансированное по высоте двоичное дерево поиска: для каждой его вершины высота её двух поддеревьев различается не более чем на 1.

Красно–чёрное дерево — это одно из самобалансирующихся двоичных деревьев поиска, гарантирующих логарифмический рост высоты дерева от числа узлов и быстро выполняющее основные операции дерева поиска: добавление, удаление и поиск узла. Сбалансированность достигается за счёт введения дополнительного атрибута узла дерева — «цвета». Этот атрибут может принимать одно из двух возможных значений — «чёрный» или «красный».

Свойства красно–черного дерева:

1. Узел либо красный, либо чёрный.

2. Корень — чёрный. (В других определениях это правило иногда опускается. Это правило слабо влияет на анализ, так как корень всегда может быть изменен с красного на чёрный, но не обязательно наоборот).

3. Все листья — чёрные.

4. Оба потомка каждого красного узла — чёрные.

5. Всякий простой путь от данного узла до любого листового узла, являющегося его потомком, содержит одинаковое число чёрных узлов.

ЛЕКЦИЯ 9

Кодирование

9.1. Равномерное и неравномерное кодирование

Есть два глобальных подхода в кодировании текста: равномерное кодирование и неравномерное кодирование.

Обозначим алфавит допустимых символов $\mathbb{A}$.

В неравномерном кодировании код символов разной длины (например, UNICODE UTF 8 — одна из самых популярных кодировок).

Рассмотрим четырехбуквенное кодирование. Закодируем буквы «А», «Б», «В», «Г» таким образом:

$$\begin{aligned} \text{А} &= 0 \\ \text{Б} &= 1 \\ \text{В} &= 10 \\ \text{Г} &= 111 \end{aligned}$$

Тогда запись «ГАГА» можно закодировать так:

$$\text{ГАГА} = 11101110 = \text{БББГА},$$

т.е. декодирование неоднозначно, такое кодирование плохое.

Условие Фано: *для того, чтобы сообщение, записанное с помощью неравномерного по длине кода, однозначно раскодировалось, достаточно, чтобы никакой код не был началом другого (более длинного) кода.* **Обратное условие Фано** (*ни один код не является концом (суффиксом) другого*) также является достаточным условием однозначного декодирования неравномерного кода.

Тогда пусть

$$\begin{aligned} \text{А} &= 0 \\ \text{Б} &= 110 \\ \text{В} &= 10 \\ \text{Г} &= 111 \end{aligned}$$

Возьмем (для удобства рядом записан столбец в зеркальном отражении):

А =	1	1
Б =	10	01
В =	100	001
Г =	000	000

Тогда

$$\text{БАГАВА} = 10100011001$$

— 1100 нет, т.е в конце ВА. 1000 тоже нет, т.е. по середине ГА. 101 нет, в начале БА. Однозначность есть, хотя и декодировать очень сложно.

Составим суффиксное дерево. Оно не нужно при декодировании, а при доработке дерева (дополнении алфавита) является полезным инструментом. Способ составления: отзеркаливаем код и строим дерево, в котором каждое ребро — цифра в коде.

Кодировка в равномерном кодировании UTF-16. Можно закодировать 2^n различных символов (мощность алфавита).

9.2. Поиск подстроки в строке

9.2.1. Наивный поиск подстроки в строке

Программа №9.2.1. Примитивный поиск подстроки в строке

```
1    s = "abbbbabbaaabbababaabb"
2    subs = "bbbaba"
3    def find(s, sub):
4        for pos in range(0, len(s)-len(sub)+1):
5            for i in range(len(sub)):
6                if sub[i] != s[pos+i]:
7                    break
8            else:
9                return pos
10       return -1
```

Построчный комментарий кода:

4) Имеет смысл проходить по основной строке, пока входящая строка влезает в рассматриваемый участок.

5)–6) Пробегаемся по элементам входящей строки и смотрим, совпадают ли они с элементами основной.

7) Если нет, уже можно переходить к следующему элементу основной строки.

9) Если прошли по всем элементам строки вхождения, можно выдать ту позицию, начиная с

которой есть вхождение.

10) Если вхождения нет, выдается ‘-1’.

Сложность алгоритма $O(N \cdot M)$. В итоге алгоритм получается неэффективным.

9.2.2. Конечный автомат поиска «abcd»

Смотрим на каждый символ только по одному разу! Методика хранения автомата: орграф. Если конечный автомат уже построен, то время поиска $O(N)$, N — длина строки.

Конечный автомат поиска является частным случаем машины Тьюринга, Подход таков:

1. Изначально система в фазе ноль.
2. Сравниваем букву в основной строке с буквой во входящей строке. Если они совпали, то код продвигается на фазу вперед.
3. Сравниваем следующие буквы. Если они совпали, переходим в фазу два и т.д.
4. В случае несовпадения фаза становится нулевой.

Программа №9.2.2. Конечный автомат для поиска подстроки «abcd»

```
1    state = 0
2    for c in s:
3        if state == 0:
4            if c == "a":
5                state = 1
6        elif state == 1:
7            if c == 'b':
8                state = 2
9            elif c == 'a':
10               state = 1
11           else:
12               state = 0
13       elif state == 2:
14           if c == 'c':
15               state = 3
16           elif c == 'a':
17               state = 1
18           else:
19               state = 0
20       elif state == 3:
21           if c == 'a':
```

```
22                 state = 1
23             elif c == 'd':
24                 state = 4
25             else:
26                 state = 0
```

9.3. Расстояние Левенштейна

9.3.1. Определение

Расстояние Левенштейна (также редакционное расстояние или дистанция редактирования) **между двумя строками в теории информации и компьютерной лингвистике** — *это минимальное количество операций вставки одного символа, удаления одного символа и замены одного символа на другой, необходимых для превращения одной строки в другую.*

Есть 2 строки Мама и Мим. Мы можем превратить их друг в друга путем вставки символа, удаления символа. Минимальный путь в данном случае — удаление последнего и замена, т.е. длина пути 2. Так и определяется расстояние Левенштейна.

a[:i], b[:j] — срезы до i–го и j–го символа. $F_{ij} = L(a[:i], b[:j])$ — расстояние Левенштейна. Тогда

$$F_{ij} = \begin{cases} \text{Последниие буквы совпадают, то } F_{(i-1)(j-1)} \\ 1 + \min(F_{(i-1)(j-1)}, F_{(i-1)j}, F_{i(j-1)}) \end{cases}$$

9.3.2. Реализация алгоритма поиска расстояния Левенштейна

Программа №9.3.1. Рекуррентная реализация поиска расстояния Левенштейна

```
1    def lev(a, b):
2        if not a:
3            return len(b)
4        if not b:
5            return len(a)
6        return min(lev(a[1:], b[1:])+(a[0] != b[0]), lev(a[1:], b)+1, lev(a, b[1:])+1)
```

Данный алгоритм записывается компактно, но асимптотика этого алгоритма ужасна.

Программа №9.3.2. Реализация поиска расстояния Левенштейна

```
1    def levenshtein(s1, s2):
2        if len(s1) < len(s2):
```

```
3           return levenshtein(s2, s1)
4
5       if len(s2) == 0:
6           return len(s1)
7
8       previous_row = range(len(s2) + 1)
9       for i, c1 in enumerate(s1):
10          current_row = [i + 1]
11          for j, c2 in enumerate(s2):
12              insertions = previous_row[j + 1] + 1
13              deletions = current_row[j] + 1
14              substitutions = previous_row[j] + (c1 != c2)
15              current_row.append(min(insertions, deletions, substitutions))
16          previous_row = current_row
17
18      return previous_row[-1]
```

ЛЕКЦИЯ 10

Z–функция строки и ее вычисление

Основной материал лекции взят с сайта.

10.1. Z–функция

10.1.1. Определение

Пусть дана строка s длины n. Тогда Z–функция от этой строки — это массив длины n, i-ый элемент которого равен наибольшему числу символов, начиная с позиции i, совпадающих с первыми символами строки s.

Иными словами, `z[i]` – это наибольший общий префикс строки s и её i–го суффикса.

Во избежание неопределённости, мы будем считать строку 0–индексированной — т.е. первый символ строки имеет индекс 0, а последний — $n - 1$.

Первый элемент Z–функции, `z[0]`, обычно считают неопределённым. Мы будем считать, что он равен нулю.

10.1.2. Реализация

Тривиальный алгоритм

Программа №10.1.1. Тривиальный алгоритм

```
1    def z_function_trivial(s):
2        n = len(s)
3        z = [0]*n
4        i = 1
5        while i < n:
6            while i + z[i] < n and s[z[i]] == s[i+z[i]]: # Пока мы не дошли до
         конца и символ на позиции z[i] равен символу на рассматриваемой позиции + z[i]
         (т.е., это и есть основная проверка)
```

```
7                z[i] += 1
8            i += 1
9        return z
```

Сложность такого алгоритма $O(N^2)$.

Эффективный алгоритм

Чтобы получить эффективный алгоритм, будем вычислять значения `z[i]` по очереди — от i=1 до $n-1$, и при этом постараемся при вычислении очередного значения `z[i]` максимально использовать уже вычисленные значения.

Программа №10.1.2. Эффективная реализация Z–функции

```
1    def z_function(s):
2        z = [0]*len(s)
3        left = 0
4        right = 0
5        x = 0
6        for i in range(1, len(s)):
7            if i <= right:
8                x = min(z[i-left], right - i + 1)
9            else:
10               x = 0
11           while i+x < len(s) and s[x] == s[i+x]:
12               x += 1
13           if i + x - 1  > right:
14               left, right = i, i + x - 1
15           z[i] = x
16       return z
```

Этот алгоритм выполняется за линейное время.

10.1.3. Применение

Применения Z–функции:

- Поиск подстроки в строке.
- Поиск количества различных подстрок в строке.
- Сжатие строки.

Запишем реализацию поиска количества различных подстрок в строке.

Программа №10.1.3. Поиск количества различных подстрок в строке

```
1    def count_different_substrings(s):
2        n = len(s)
3        start = s[0]
4        res = 0
5        for i in range(1, n):
6            t = start[::-1]
7            k = len(t) - max(z_function(t))
8            res += k
9            start += s[i]
10       return res
```

ЛЕКЦИЯ 11

Z и префикс функция строки

11.1. Z–функция строки

Z–функция строки — функция от номера символа. Z–функция — это массив длинны `len(S) = N`, `z[i]` — длина совпадающего префикса у строки `S` и `S[i:]`.

`z[0]` не определено. Но мы будем считать, что `z[0] = 0`.

"a a a a a"
z=[0, 4, 3, 2, 1]
"a b a c a b a"
z=[0, 0, 1, 0, 3, 0]

Зачем нужна z–функция? Будем искать строчку `p=aba` и все ее вхождения в строке `abacabadabacaba`. Склеим две строки символом, которого точно нет ни там ни там:

`s = "aba#abacabadabacaba"`.

Т.к. стоит символ #, длина искомой подстроки не может быть больше 3.

$$z = [0, 0, 1, 0, 3, 0, 1, 0, 3, 0, 1, 0, 3, 0, 1, 0, 3, 0, 1]$$

Там, где `z[i] == len(p)`, т.е. там, где величина Z–функции равна длине подстроки, у нас есть совпадение, т.е. там подстрока содержится в строке. Позиция вхождения: найдена подстрока в строке, `pos = i-len(p)-1` — номер вхождения.

Тривиальное вычисление Z–функции (требует $O(N^2)$).

Программа №11.1.1. Тривиальное вычисление Z–функции

```
1    N = len(s)
2    z=[0]
3    left = right = 0
4    for i in range(1, N):
```

```
5         x = 0
6         while i + x < N and s[x] == s[i+x]:
7             x += 1
8         z[i] = x
9         if i + x - 1> right: # Сохраянем z--блок
10            left, right = i, i + x - 1
```

z–блок — срез строки `s[i:i+z[i]]`, т.е. это часть строки, совпавшая с подстрокой.

На момент вычисления `z[i]` существует самый правый отрезок совпадения. Длина этого отрезка равна разнице его правого и левого конца + 1.

Программа №11.1.2.

```
1     N = len(s)
2     z=[0]
3     left = right = 0
4     for i in range(1, N):
5         x = min(z[i-left], right - i + 1) if i<=right else 0
6         while i + x < N and s[x] == s[i+x]:
7             x += 1
8         z[i] = x
9         if i + x - 1> right: # Сохраянем z--блок
10            left, right = i, i + x - 1
```

Этот алгоритм работает за линейное время.

11.2. Префикс–функция строки. Алгоритм Кнута — Морриса — Пратта.

11.2.1. Префикс–функция строки

Собственным суффиксом строки называется суффикс, не совпадающий со всей строкой, совпадающий с ее префиксом.

Префикс–функция строки π`[i]` — массив длинной строки, где π`[i]` — длина наибольшего по длине собственного суффикса подстроки (среза) `s` начиная от начала и до позиции `i (s[:i+1])`.

"a a a a a"

pi=[0, 1, 2, 3, 4]

"a b a c a b a"

pi = [0, 0, 1, 0, 1, 2, 3]

Заметим, что эта функция всегда растет на единицу.

Программа №11.2.1. Тривиальный алгоритм

```
1    N = len(s)
2    pi = [0]*N
3    for i in range(1, N):
4        for k in range(i+1):
5            if s[0:k] == s[i-k+1:i+1]:
6                pi[i] = k
```

Асимптотика $O(N^3)$.

Программа №11.2.2. Эффективный алгоритм

```
1    def prefix(s):
2        n = len(s)
3        pi = [0]*n
4        for i in range(1, n):
5            j = pi[i-1]
6            while j > 0 and s[i] != s[j]:
7                j = pi[j-1]
8            if s[i] == s[j]:
9                j += 1
10            pi[i] = j
11        return pi
```

11.2.2. Поиск подстроки в строке

Эта задача является классическим применением префикс-функции (и, собственно, она и была открыта в связи с этим).

Дан текст t и строка s, требуется найти и вывести позиции всех вхождений строки s в текст t.

Обозначим для удобства через n длину строки s, а через m — длину текста t.

Образуем строку s + # + t, где символ # — это разделитель, который не должен нигде более встречаться. Посчитаем для этой строки префикс-функцию. Теперь рассмотрим её значения, кроме первых n+1 (которые, как видно, относятся к строке s и разделителю). По определению, значение π[i] показывает наидлиннейшую длину подстроки, оканчивающейся в позиции i и совпадающего с префиксом. Но в нашем случае это π[i] — фактически длина наибольшего блока совпадения со строкой s и оканчивающегося в позиции i. Больше, чем n, эта длина быть не может — за счёт разделителя. А вот равенство π[i] = n (там, где оно достигается), означает, что в

позиции i оканчивается искомое вхождение строки s (только не надо забывать, что все позиции отсчитываются в склеенной строке s+#+t).

Таким образом, если в какой-то позиции i оказалось π[i] = n, то в позиции i - (n + 1) - n + 1 = i - 2 n строки t начинается очередное вхождение строки s в строку t.

Как уже упоминалось при описании алгоритма вычисления префикс-функции, если известно, что значения префикс-функции не будут превышать некоторой величины, то достаточно хранить не всю строку и префикс-функцию, а только её начало. В нашем случае это означает, что нужно хранить в памяти лишь строку s + # и значение префикс-функции на ней, а потом уже считывать по одному символу строку t и пересчитывать текущее значение префикс-функции.

Итак, алгоритм Кнута-Морриса-Пратта решает эту задачу за O(n+m) времени и O(n) памяти.

Подробнее материал лекции изложен на сайте.

ЛЕКЦИЯ 12

Автоматы

12.1. Машина Тьюринга

Это абстрактный исполнитель, живущий на бесконечной ленте, в клетках которой находятся буквы α принадлежащие фиксированному алфавиту A. Каретка машины Тьюринга может двигаться по ленте влево или вправо. Каретка может видеть, что нарисовано на ленте (на текущей клетке). Также у нее есть возможность изменять символы на ленте, т.е. она может записать, изменить. А также у нее есть состояние q. При этом это состояние принадлежит множеству допустимых состояний Q. Подмножество состояний — состояние останова (остановки). Это подмножество конечно. Поведение этой машины детерминированно (оно задается увиденным символом и предыдущим состоянием). Она из исходного состояния переходит в новое состояние, в котором определено: 1) Состояние системы 2) Считанный символ 3) Действие.

Множества Q и A конечные. Возможно очень большие, но конечные. Мощность множества — число конечных состояний в нем.

Программа, которую мы написали, не хранится нигде. Типа в памяти каретки. Нужно её куда–то записать. Самое простое — написать не ленте. Тогда каретка, которая едет на двух лентах сразу — универсальная (программируемая) машина Тьюринга. О скорости работы нет разговора. Есть вопрос о вычислимости алгоритма.

12.2. Вычислимость функций (алгоритмов)

Что по сути такое алгоритм? Есть определенные возможные входные данные. Есть множество значений — множество возможных результатов. Алгоритм — своего рода функция, которая переводит множество определения в множество значений. Но невычислимые функции. Вычислимые функции (алгоритмы) — это алгоритмы. Функции называются вычислимыми, если есть возможность посчитать её через машину Тьюринга. То, что мы называем раличными алгоритмами (все виды сортировок) — это, с точки зрения вычислимости - один алгоритм. Нам ведь не важен путь (как и не важна скорость). Нам важно, что есть возможность получить результат, не более того.

А какие алгоритмы невычислимые? Например, доказано, что нельзя вычислить вычислимость программы. Т.е. невозможно написать программу, которая посчитает, закончится программа или нет.

Исполнители A и B называются алгоритмически эквивалентными, если можно сэмулировать A на B и B на A.

12.3. Клеточные автоматы. Игра жизнь Джона Конвая

Простейшие автоматы — это клетки, живущие не линии. В клетках — нули и единицы. Состояние клетки зависит только от самой клетки и от двух ее ближайших соседей. В каждый следующий момент времени клетка меняет свое состояние в зависимости от своего и соседей состояний. При этом для всех клеток алгоритмы одинаковы. Всего есть 256 клеточных автоматов (возможных комбинаций для данных состояний триад клеток). Среди них есть и совсем простые. Интересны несколько из них. Одно — правило 30, т.к. порождает случайные хаотические структуры. Также есть правила жизни Джона Конвея. Если клетка была жива, то остается живой при 2 или 3 соседях. А если была мертва, то оживает при наличии трех соседей.