Министерство образования и науки Российской Федерации

Федеральное государственное автономное образовательное учреждение высшего образования «Санкт-Петербургский национальный исследовательский университет информационных технологий, механики и оптики»

Мегафакультет трансляционных информационных технологий

Факультет инфокоммуникационных технологий

Дисциплина: Алгоритмы и структуры данных

**Отчет по Лабораторной работе №3**

Выполнила: Микулина Алиса Романовна

Группа: K3143, 1 курс

Преподаватель: Харьковская Татьяна Александровна

Санкт-Петербург

27.06.2022

**Описание задания**

**Задание 1.**

В задаче нужно собрать лабиринт и проверить, есть ли связь между двумя данными вершинами. Что-то на очень легком, особенно после деревьев.

**Задание 2.**

Все тот же лабиринт, теперь нужно найти количество не связанных друг с другом кластеров. (Да, я знаю, что это компоненты, а не кластеры, но я, наверное, предпочту называть их так)))

**Задание 3.**

Теперь работаем с курсами. Дан ориентированный граф, нужно определить, нет ли циклов в порядке следования курсов.

**Задание 4.**

Имеем все те же курсы. Теперь все курсы уже не имеют циклов, нужно его топологически отсортировать, чтобы курсы линейно шли друг за другом.

**Задание 9.**

Не задача, а мракобесие какое-то. Ищем циклы, что я уже не очень люблю, так еще и надо смотреть не отрицателен ли суммарный вес каждого цикла. Не оч приятная штука.

**Задание 11.**

Очень приятная задача на БФС. Имеется ряд возможных алхимических превращений, нужно найти кратчайший способ получить из одного вещества другое. Просто, красиво, приятно.

**Задание 13.**

Грядки. Имеется поле, на нем размечены грядки и пустая территория. Нужно вывести количество грядок. Очень тяжело оптимизировать.

**Задание 18.**

Имеются координаты городов, между ними нужно построить дороги так, чтобы из любой точки можно было попасть в любую точку, и при этом общая сумма длин дорог была минимальной.

**Описание решения и исходный код**

**Задача 1.**

Казалось бы задачка существует чисто по приколу, чтобы без усилий получить один балл. Да? Ага, конечно. Я заметила, что она у меня неправильно решена ровно как села писать отчет. То есть у меня уже решены все задачи, проходят на acmp, а у меня тут просто проверка связи не работает…

Проблема была в том, что я просто проверяла, есть ли прямая связь между данными на поис нодами. Почему-то я не учитывала возможность того, что ячейки могут не быть соединены напрямую, но путь будет.

Поэтому сейчас быстренько переписала на рекурсивный поиск. Работает, и то хорошо)))

Так что даже на таких вот задачах можно затупить и решить как-то странно, при этом до последнего этого не замечая :)))))

def **find\_route**(connections, c\_one, c\_two):

    if not c\_one in connections or connections[c\_one] == []:

        return 0

    if c\_two in connections[c\_one]:

        return 1

    else:

        children = connections[c\_one]

        for child in children:

            ans = **find\_route**(connections, child, c\_two)

            if ans == 1:

                return 1

    return 0

with **open**('input\_1.txt') as f:

    num\_nodes, num\_connections = **list**(**map**(**int**, f.**readline**().**split**()))

    connections = {}

    for i in **range**(num\_nodes):

        connections[i] = []

    for i in **range**(num\_connections):

        first, second = **list**(**map**(**int**, f.**readline**().**split**()))

        connections[first - 1].append(second - 1)

**print**(connections)

    c\_one, c\_two = **list**(**map**(**int**, f.**readline**().**split**()))

whether\_found = **find\_route**(connections, c\_one - 1, c\_two - 1)

**print**(whether\_found)

**Задача 2.**

Тут нужно искать количество несоединенных друг с другом кластеров. Я это делала по алгоритму из лецкии, то есть у меня есть список уже посещенных вершин. Если мы уже обходили вершину в качестве части какого-то кластера, то мы ее уже не будем проверять, а проверять будем только необойденные. Достаточно наивно, но чет мой мозг очень долго въезжал в суть данного чудесного алгоритма.

Но так-то это логично, поэтому хорошо, что работает)))

def **count\_clusters**(connections, num\_nodes):

    visited = [0 for i in **range**(num\_nodes)]

    clust\_num = 0

    for host in connections.keys():

        if not visited[host]:

            children = connections[host]

**explore**(host, children, visited, connections)

            clust\_num += 1

    return clust\_num

def **explore**(host, children, visited, connections):

    visited[host] = 1

    for child in children:

        if not visited[child]:

            new\_host = child

            new\_children = connections[new\_host]

**explore**(new\_host, new\_children, visited, connections)

    return

with **open**('input\_2.txt') as f:

    num\_nodes, num\_connections = **list**(**map**(**int**, f.**readline**().**split**()))

    connections = {}

    for i in **range**(num\_nodes):

        connections[i] = []

    for i in **range**(num\_connections):

        first, second = **list**(**map**(**int**, f.**readline**().**split**()))

        connections[first - 1].append(second - 1)

        connections[second - 1].append(first - 1)

**print**(connections)

**print**(**count\_clusters**(connections, num\_nodes))

**Задача 3.**

Первая версия адского издевательства над мозгом – поиск циклов в ориентированном графе. Глазками-то как просто, правда? Сразу цикл видно, сразу все хорошо, а тут надо как-то железке объяснить, как квакать…

Ну учим квакать, получается, в черно-белом цикле. Изначально у нас все вершинки белые. Когда мы в них заходим в цикле, красим в серый. Когда выходим из вершинки, красим ее в черный, и больше в нее никогда не лезем, потому что мы ее уже обошли.

А если мы пытаемся залезть в цикле в уже и так серую ячейку, то, получается, мы нашли цикл. Можно радостно квакнуть и вывести единичку. Если за цикл ни разу не квакнули, то выводим нолик – циклов нет.

def **find\_cycles**(connections, num\_nodes):

    colours = ['white' for i in **range**(num\_nodes)]

    for i in **range**(num\_nodes):

        if colours[i] == 'white':

            try:

                new = connections[i][0]

                colours[i] = 'grey'

                trial = **explore**(connections, colours, new)

                if trial != 1:

                    colours[new] = 'black'

                else:

                    return 1

            except:

                continue

    return 0

def **explore**(connections, colours, host):

    if **len**(connections[host]) == 0:

        return

    children = connections[host]

    for child in children:

        if colours[child] == 'grey':

            return 1

        elif colours[child] == 'black':

            return

        else:

            new\_host = child

            colours[new\_host] = 'grey'

            trial = **explore**(connections, colours, new\_host)

            if trial != 1:

                colours[new\_host] = 'black'

                return

            else:

                return 1

with **open**('input\_3.txt') as f:

    num\_nodes, num\_connections = **list**(**map**(**int**, f.**readline**().**split**()))

    connections = {}

    for i in **range**(num\_nodes):

        connections[i] = []

    for i in **range**(num\_connections):

        first, second = **list**(**map**(**int**, f.**readline**().**split**()))

        connections[first - 1].append(second - 1)

**print**(**find\_cycles**(connections, num\_nodes))

**Задача 4.**

Квакать научились, теперь циклов у нас в графах, допустим, нет. Нужно теперь взять и все уроки в курсах (кстати да, это мы курсы проверяли на наличие циклов), расположить в правильном линейном порядке.

На самом деле я опять решила поизучать лекцию, вникнуть во всю боль своего существования, переписать алгоритм три раза, и наконец поняла что к чему. Возможно, причина в том, что это было в 3 ночи (но нет, не, 100% нет, прям точно).

Ну в итоге оно заработало нормально, я поняла, что надо брать первыми те элементы, у которых нет детей, снимать их с графа, опять искать те, у которых нет, итд итп пока не дойдем до опустошенного графа.

Потом берем все снятое, переворачиваем, и ура!!!! Оно лежит правильно! Можно теперь запускать свой курс по тому как закрывать 5 предметов за две недели, я-то теперь quite experienced)))

def **linear**(connections, num\_nodes):

    line = []

    for key in connections.keys():

        line = **go\_deep**(connections, key, line)

        if key not in line:

            line.**append**(key)

    for i in **range**(**len**(line)):

        line[i] += 1

    return line[::-1]

def **go\_deep**(connections, node, line):

    if connections[node] != []:

        children = connections[node]

    else:

        if node not in line:

            line.append(node)

        return line

    for child in children:

        line = **go\_deep**(connections, child, line)

        if child not in line:

            line.append(child)

    return line

with **open**('input\_4.txt') as f:

    num\_nodes, num\_connections = **list**(**map**(**int**, f.**readline**().**split**()))

    connections = {}

    for i in **range**(num\_nodes):

        connections[i] = []

    for i in **range**(num\_connections):

        first, second = **list**(**map**(**int**, f.**readline**().**split**()))

        connections[first - 1].append(second - 1)

**print**(connections)

**print**(**linear**(connections, num\_nodes))

**Задача 9.**

Внимание: учимся квакать громче! Теперь ищем не просто циклы, а циклы, суммарный вес которых отрицателен.

Это еще большее мучение для мозга, правда, я опять решала это ночью, но ниважна)))

В общем, опять учимся на черно-белом. Тут мало того, что нужно найти цикл, так еще и надо посчитать его стоимость, и проигнорить его, если вес положительный, и вывести единичку, если отрицательный.

Ночью у меня, видимо случились приколы с головой и я запихнула функцию в функцию, потому что мне было колоссально лень передавать лишние переменные, а теперь я переписывать уже точно не буду, потому что если я что-то поменяю оно сломается и я ни за что сейчас не готова чинить это обратно. Поэтому давайте просто смиримся с тем, что у меня функция поиска лежит внутри функции поиска цикла, и вызывается либо из себя самой же, либо из функции-родителя…

Так вот, как громко квакать. По сути, то же самое. Мы берем три цвета, белый, серый и черный. Правда теперь, каждый раз, когда пытаемся найти цикл, еще и считаем стоимость пройденных путей. Если мы заходим в серую вершинку, но при этом у нас положительная сумма пройденного пути, мы тихо квакаем, красим вершинку в черный как обычную вершинку, и все вершинки по пути тоже.

Если же мы зашли в серую вершинку, а сумма у нас отрицательная, то можно громко квакнуть! Мы нашли отрицательный цикл! Выводим единичку и забываем об этом алгоритме навсегда.

class **Graph**:

    def **\_\_init\_\_**(self, num\_nodes=0):

        self.dict\_nodes = **dict**()

        self.num\_nodes = num\_nodes

        for i in **range**(num\_nodes):

            self.dict\_nodes[i + 1] = **set**()

    def **add**(self, first, second, cost):

        self.dict\_nodes[first].add((second, cost))

    def **find\_cycles**(self):

        def **find**(a=1, sumc=0):

            path = self.dict\_nodes[a]

            color[a] = ('gray', sumc)

            for i in path:

                if color[i[0]][0] == 'gray':

                    if sumc + i[1] - color[i[0]][1] < 0:

                        nonlocal flag

                        flag = True

                elif color[i[0]][0] == 'white':

**find**(i[0], sumc + i[1])

            color[a] = ('black', sumc)

            nonlocal ready

            ready.**discard**(a)

        color = **dict**()

        ready = **set**()

        flag = False

        for i in **range**(self.num\_nodes):

            color[i + 1] = ('white', 0)

            ready.**add**(i + 1)

        while ready:

**find**(ready.**pop**())

        return flag

with **open**('input\_9.txt') as f:

    num\_nodes, num\_connections = **list**(**map**(**int**, f.**readline**().**split**()))

    g = **Graph**(num\_nodes)

    for i in **range**(num\_connections):

        first, second, cost = **list**(**map**(**int**, f.**readline**().**split**()))

        g.**add**(first, second, cost)

**print**(**int**(g.**find\_cycles**()))

**Задача 11.**

Эту задачу я пошла решать сразу после 4 ибо а почему нет собственно) Поиск в ширину я прошла еще в прошлом семестре на программировании, поэтому сейчас было уже попроще.

Написала я, значит, задачу. Она выводит все правильные решения, а на acmp падает на времени… \* крик щитоспинки \*

Ну я в итоге переписала алгоритм. Он работает на честном слове, а так же на том, что у меня есть массив used, в котором хранятся уже пройденные варианты, чтобы не ходить циклично 2000000000 раз пока все совсем не упадет.

В общем я просто беру нодик, нахожу его детей, добавляю в массив всех детей детей и так по очереди пока не найду нужный элемент. При этом, как уже сказала, циклично не хожу. При каждом шаге вперед считаю этот шаг, получается, что выведу я первый возможный вариант нахождения.

Как-то так, тесты прошлись, оно оптимизировалось, я довольна.

class **Graph**():

    def **\_\_init\_\_**(self) -> None:

        self.instructions = {}

        self.i\_have = ''

        self.i\_want = ''

        self.used = []

    def **find\_my**(self):

        if self.i\_have == self.i\_want:

            return 0

        else:

            if self.i\_have not in self.instructions:

                return -1

            self.used.**append**(self.i\_have)

            self.i\_have = self.instructions[self.i\_have]

            if self.i\_want in self.i\_have:

                return 1

            ans = self.**go\_deep**(step=1)

        return ans

    def **go\_deep**(self, step):

        if self.i\_have == []:

            return - 1

        next\_step = []

        for elem in self.i\_have:

            if elem in self.instructions:

                self.used.**append**(elem)

                potential = self.instructions[elem]

                for pot in potential:

                    if (pot not in next\_step) and (pot not in self.used):

                        next\_step.**append**(pot)

        if self.i\_want in next\_step:

            return step + 1

        else:

            self.i\_have = next\_step

            next\_step = []

            steps = self.**go\_deep**(step + 1)

            return steps

with **open**('input.txt') as f:

    num = **int**(f.**readline**())

    instructions = {}

    for i in **range**(num):

        lst = f.**readline**().**split**()

        first, second = lst[0], lst[2]

        if first not in instructions:

            instructions[first] = [second]

        else:

            instructions[first].append(second)

    i\_have = f.**readline**().**split**()[0]

    i\_want = f.**readline**().**split**()[0]

gr = **Graph**()

gr.instructions = instructions

gr.i\_have = i\_have

gr.i\_want = i\_want

ans = (gr.**find\_my**())

*# print(ans)*

with **open**('output.txt', 'w') as d:

    d.**write**(**str**(ans))

**Задача 13.**

После 11 я пошла решать 13, ибо почему бы не набрать боооольше баллов))

Задачка про грядки, я решала ее с помощью ДФС. Сначала решила, оно работало по факту, но не работало на acmp, падало на времени. Дальше следовало полное избавление от рекурсии в решении, а также – самое главное – хранение информации о том, была ли вершина посещена, в табличке. Таким образом проверить были мы там за O(1), что гораздо лучше, чем использовать if smth in list – потому что оно очень медленное.

Также я стала хранить детей и родителей в одном массиве, брать родителя сверху, и удалять его перед тем как добавлять детей. Таким образом это тоже работает за O(1) и все вместе ускорило алгоритм от 15с выполнения на максимальных показателях до 0,15 секунды!

Алгоритмы, все-таки, хорошая штука!!!

class **Garden**:

    def **\_\_init\_\_**(self) -> None:

        self.height = 0

        self.length = 0

        self.grid = []

        self.beds = 0

        self.visited = []

    def **count\_beds**(self):

        self.visited = [[False for i in **range**(self.length)] for j in **range**(self.height)]

        for i in **range**(self.height):

            for j in **range**(self.length):

                if self.grid[i][j] == '#':

                    self.beds += 1

                    self.visited[i][j] = True

                    self.**discover\_bed**(i, j, self.beds)

        return self.beds

    def **discover\_bed**(self, x, y, id):

        this\_cluster = [[x, y]]

        possible\_routes = [(-1, 0), (0, -1), (0, 1), (1, 0)]

        while this\_cluster != []:

            x, y = this\_cluster[-1]

            this\_cluster.**pop**()

            for pair in possible\_routes:

                m, p = x + pair[0], y + pair[1]

                if (0 <= m < self.height) and (0 <= p < self.length):

                    if self.visited[m][p] == True:

                        continue

                    elif self.grid[m][p] == '#':

                        self.visited[m][p] = True

                        this\_cluster.**append**([m, p])

            self.grid[x][y] = **str**(**id**)

        return

with **open**('input.txt') as f:

    height, length = **list**(**map**(**int**, f.**readline**().**split**()))

    grid = []

    for i in **range**(height):

        gridline = []

        new\_line = **list**(f.**readline**().**strip**())

        grid.**append**(new\_line)

g = **Garden**()

g.height = height

g.length = length

g.grid = grid

ans = g.**count\_beds**()

with **open**('output.txt', 'w') as d:

    d.**write**(**str**(ans))

**Задача 18.**

Мне очень не понравилась моя 17я задача из обязательного варианта, и я ее что? Правильно, заменила на задачку со звездочкой. Не знаю, правда, почему она со звездочкой, потому что она очень простая. Мы сначала создаем массив, в котором лежит информация о всех возможных соединениях между ячейками.

Дальше кидаем этим отсортированным массивом в алгоритм Крускала, и он его пережевывает.

Для каждой грани мы смотрим, соединены ли уже два нодика этой связи. Если нет, то формируем связь, если одна из вершин уже с чем-то соединена, а вторая нет, то прикрепляем к ней вторую и радуемся жизни. Так мы делаем пока не обойдем все грани.

Дальше мы идем по ребрам второй раз и, если у нас есть группы вершин, не соединенные между собой, то соединяем.

Дальше считаем сумму длин всех ребер, которые мы сохранили и все! Это тот самый граф минимальной суммарной длины, где все ячейки соединены.

Просто нужно понять алгоритм, на самом деле, в задаче вообще ничего сложного 😊

from **math** import inf

def **build\_graph**(distances):

    length = 0

    united = **set**()

    isolated = {}

    sides = []

    for distance in distances:

        if (distance[1] not in united) or (distance[2] not in united):

            if (distance[1] not in united) and (distance[2] not in united):

                isolated[distance[1]] = [distance[1], distance[2]]

                isolated[distance[2]] = isolated[distance[1]]

            else:

                if not isolated.**get**(distance[1]):

                    isolated[distance[2]].append(distance[1])

                    isolated[distance[1]] = isolated[distance[2]]

                else:

                    isolated[distance[1]].append(distance[2])

                    isolated[distance[2]] = isolated[distance[1]]

            sides.**append**(distance)

            united.**add**(distance[1])

            united.**add**(distance[2])

    for distance in distances:

        if distance[2] not in isolated[distance[1]]:

            sides.**append**(distance)

            first\_group = isolated[distance[1]]

            isolated[distance[1]] += isolated[distance[2]]

            isolated[distance[2]] += first\_group

    for side in sides:

        length += side[0]

    return length

def **calculate\_distance**(x1, y1, x2, y2):

    dist = (((x1 - x2) \*\* 2) + ((y1 - y2) \*\* 2)) \*\* (1 / 2)

    return dist

def **fill**(distances, nodes, num\_nodes):

    for i in **range**(num\_nodes):

        for j in **range**(num\_nodes):

            if j > i:

                continue

            elif i == j:

                continue

            else:

                x1, y1 = nodes[i]

                x2, y2 = nodes[j]

                distance = **calculate\_distance**(x1, y1, x2, y2)

                distances.append((distance, i, j))

    dist = **sorted**(distances, key=lambda x: x[0])

    return dist

with **open**('input\_18.txt') as f:

    num\_nodes = **int**(f.**readline**())

    nodes = {}

    for i in **range**(num\_nodes):

        x, y = **list**(**map**(**int**, f.**readline**().**split**()))

        nodes[i] = [x, y]

    distances = []

    distances = **fill**(distances, nodes, num\_nodes)

**print**(distances)

ans = **build\_graph**(distances)

**print**(**format**(ans,".9f"))

**Описание проведенных тестов.**

**1**

|  |  |
| --- | --- |
| **Input** | **Output** |
| 4 4  1 2  3 2  4 3  1 4  1 4 | 1 |
| 4 2  1 2  3 2  1 4 | 0 |

**2**

|  |  |
| --- | --- |
| **Input** | **Output** |
| 4 2  1 2  3 2 | 2 |
| 4 4  1 2  3 2  4 3  1 4  1 4 | 1 |

**3**

|  |  |
| --- | --- |
| **Input** | **Output** |
| 4 4  1 2  4 1  2 3  3 1 | 1 |
| 5 7  1 2  2 3  1 3  3 4  1 4  2 5  3 5 | 0 |

**4**

|  |  |
| --- | --- |
| **Input** | **Output** |
| 4 3  1 2  4 1  3 1 | [4, 3, 1, 2] |
| 4 1  3 1 | [4, 3, 2, 1] |
| 5 7  2 1  3 2  3 1  4 3  4 1  5 2  5 3 | [5, 4, 3, 2, 1] |

**9**

|  |  |
| --- | --- |
| **Input** | **Output** |
| 4 4  1 2 -5  4 1 2  2 3 2  3 1 1 | 1 |

**11**

Изображение выглядит как стол

Автоматически созданное описание

**13**

Изображение выглядит как стол

Автоматически созданное описание

**18**

|  |  |
| --- | --- |
| **Input** | **Output** |
| 4  0 0  0 1  1 0  1 1 | 3.000000000 |
| 5  0 0  0 2  1 1  3 0  3 2 | 7.064495102 |

**Выводы по проделанной работе.**

Решила отчет начать с тестов. Планировала защититься в понедельник, но местечек не нашлось, да и отчет не готов. Вот только тесты добавила. Пока собиралась с духом писать отчет, мне было скучно, и я решила побегать вокруг ячейки в табличке на запись где стояла Татьяна. Ко мне присоединилась еще толпа и начался хаос какой-то, с грустными смайликами и сердечками :DDD

Как-то так пока, пойду отчет писать, а то мне еще одну лабу с нуля решать надо начинать, а отчеты я пишу очень долго…………..

Такс, ну вот, три часа спустя я дописала абсолютно все. Мне очень понравилась эта лаба, она прям какая-то красивая, и сами задачки элегантные.

Мне нравятся графы, но что-то они не так уж сложно выглядят после деревьев :DDD Настрадалась просто с ними, а теперь радуюсь отсутствию страданий (а они меня уже ждут в следующей лабе))))) Но мне правда нравится, даже когда очень сложно. Я довольна тем, что я понимаю то, как работают разные структуры данных и алгоритмы)

Пойду решать последнюю лабу, пока время есть еще сегодня 😊