Санкт-Петербургский Национальный Исследовательский Университет
Информационных Технологий, Механики и Оптики

Факультет инфокоммуникационных технологий

Лабораторная работа №5

Выполнил:

Абдулов И.А.

Проверил:

Мусаев А.А.

Санкт-Петербург,

2022

СОДЕРЖАНИЕ

Стр.

ВВЕДЕНИЕ

Опытному и уважающему себя программисту необходимо уметь реализовывать некоторые алгоритмы и структуры данных, знание которых облегчает решение многих повседневных задач, возникающих у типичного программиста.

Цель данной работы – ознакомление с алгоритмами поиска в глубину и ширину на языке программирования Python.

Данная работа представляет собой отчет о выполненных заданиях:

1. Написать программу, которая определяет, является ли введенная скобочная структура правильной.
2. Придумать и решить задачу для алгоритма поиска в глубину и для алгоритма поиска в ширину.
3. Написать алгоритм, который найдет выход из лабиринта.

1 Задание 1

Определение правильности скобочной последовательности требует наличия счетчика, изначально равного нулю. Будем последовательно просматривать символы скобочной последовательности. Если символ – открывающая скобка, то к счетчику прибавим единицу, иначе прибавим минус единицу. Если в какой-либо момент времени счетчик станет отрицательным, это значит, что в тот момент времени количество закрывающих скобок превышало количество открывающих, то есть последовательность неверна. Если после прохода по символам всей строки счетчик равен нулю, то количество открывающих и закрывающих скобок совпадает, последовательность верна.

Определение первого символа, нарушающего верность последовательности требует дополнительного счетчика который укажет индекс символа первой не закрытой скобки или первой скобки, не имеющей соответствующей открытой.

```
18  def is_Correct(s):
19      print(s)
20      c, d = 0, 0 # '(' + 1 ')' -  1
21      ans1, ans2 = True, 0
22      for i in range(len(s)):
23          if s[i] == '(': c, d = c+1, d+1
24          else: c, d = c-1, d+1
25          if c == 0: d = 0
26          if c < 0:
27              ans1, ans2 = False, i
28              break
29      if ans1 and c==0: print('OK', end='\n\n')
30      else:
31          if ans1 == True: ans2 = len(s)-d
32          print('Номер первого символа, нарушающего правильность:', ans2+1, end='\n\n')
```

Рисунок 1.1 — Код проверки скобочной последовательности на правильность

```
()()
OK

(((())))
OK

)(
Номер первого символа, нарушающего правильность: 1

(()(()
Номер первого символа, нарушающего правильность: 3

(
Номер первого символа, нарушающего правильность: 1
```

Рисунок 1.2 — Пример работы кода по определению вида скобочной последовательности

Вывод: алгоритм определения вида скобочной последовательности требует внимательности в реализации и создается с использованием счетчика, который изменяется в зависимости от одного из двух видов символов, а именно открывающей и закрывающей скобки.

2 Задание 2

«Обход в глубину» или «Поиск в глубину» – это рекурсивный алгоритм поиска всех вершин графа или древовидной структуры данных. Цель алгоритма – пометить каждую вершину, как посещенную, избегая циклов.

Для применения алгоритма поиска в глубину на практике была придумана следующая задача.

Дано дерево, необходимо найти его диаметр.

В первой строке два числа: 1. Количество вершин в дереве 2. Количество ребер. Затем следует m пар чисел – ребра дерева.

Программа должна вывести диаметр дерева

Идея решения данной задачи проста: необходимо поиском в глубину найти два наибольших расстояния от корня дерева до его листьев. Диаметром дерева будет сумма двух наибольших расстояний.

```
def dfs(v, p=-1, c=0):
    global maxlen1, maxlen2
    for u in adj[v]:
        if u != p:
            dfs(u, v, c+1)
    if len(adj[v])==1:
        if c >= maxlen1:
            maxlen2, maxlen1 = max(maxlen1, maxlen2), c
        elif c > maxlen2:
            maxlen2 = c
```

Рисунок 2.1 — Код функции поиска в глубину при решении задачи

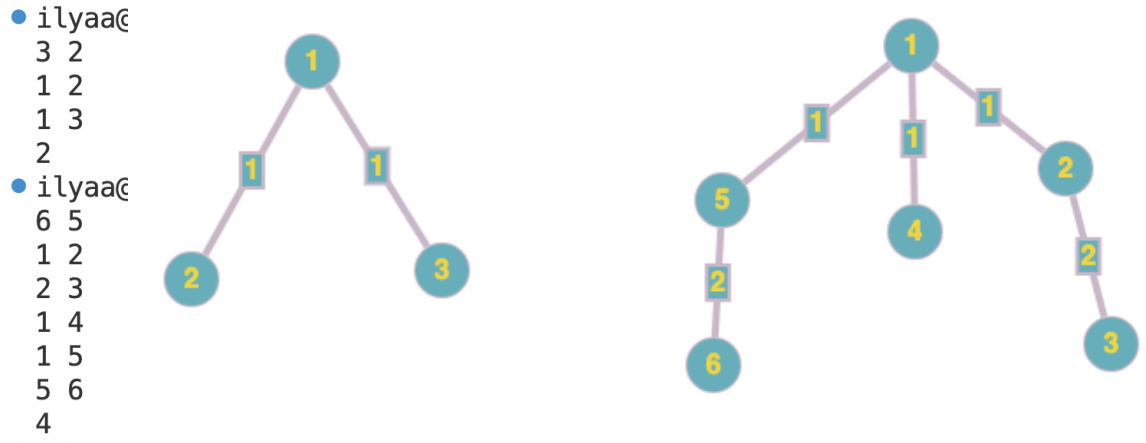

Рисунок 2.2 — Пример программы с применением поиска в глубину

Алгоритм поиска в ширину (англ. breadth-first search, BFS) позволяет найти кратчайшие пути из одной вершины невзвешенного графа до всех остальных вершин. Под кратчайшим путем подразумевается путь, содержащий наименьшее число ребер.

Для применения алгоритма поиска в ширину на практике была придумана следующая задача.

Дано дерево, необходимо найти эксцентриситет вершины.

В первой строке три числа: 1. Количество вершин в дереве 2. Количество ребер 3. Вершина. Затем следует m пар чисел – ребра дерева.

Требуется вывести эксцентриситет введенной вершины

Идея решения данной задачи такова: считаем, что дистанция до введенной вершины равна нулю, будем считать расстоянием до вершин дерева сумму дистанции родительской вершины плюс один, посещая только ранее не посещенные вершины. Тогда максимальная из дистанций посещенных вершин будет искомым эксцентриситетом.

```
21    queue = [s]
22    vis = [False]*(n+1)
23    dist = [-1]*(n+1)
24    dist[s] = 0
25    while len(queue) != 0:
26        v = queue.pop(0)
27        vis[v] = True
28        for u in adj[v]:
29            if not vis[u]:
30                dist[u] = dist[v]+1
31                queue.append(u)
32    print(max(dist))
```

Рисунок 2.3 — Код функции поиска в ширину при решении задачи

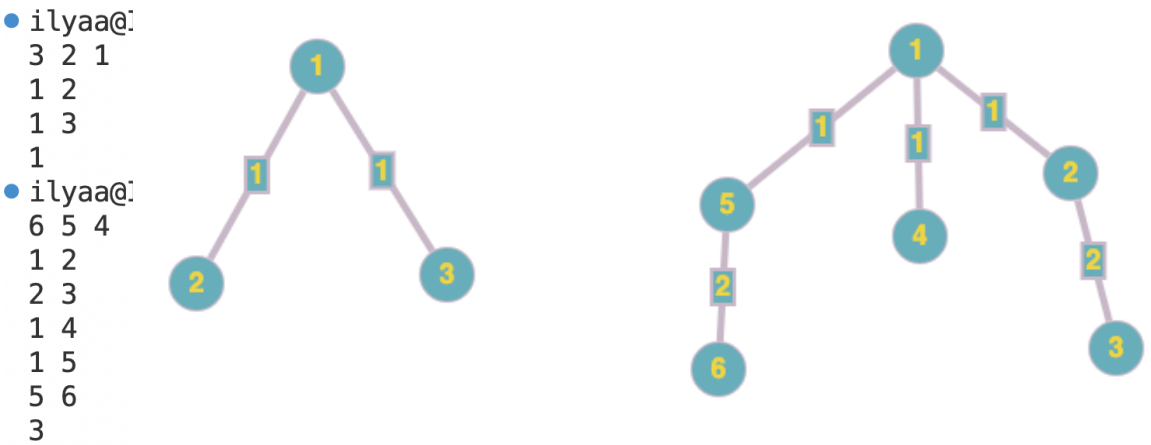

Рисунок 2.4 — Пример программы с применением поиска в ширину

Вывод: алгоритмы поиска в глубину и ширину отличаются программной реализацией, но могут быть оба применены для решения одной и той же задачи.

3 Задание 3

Для решения задачи с поиском выхода из лабиринта, в котором 0 – это проход, а 1 – это стена, был выбран алгоритм поиска в ширину, потому что его нерекурсивную реализация в данной программе будет проще модифицировать.

В структуре данных очередь в рамках данной программы будут находится пары индексов матрицы. Добавляться в очередь будут не посещенные нулевые клетки поля лабиринта слева, справа, снизу или сверху от извлеченной из очереди клетки. Как только в очереди появится пара индексов таких, что индекс находится на правой границе лабиринта, цикл поиска в ширину завершит работу.

```
moves = [[-1, 0], [1, 0], [0, -1], [0, 1]] # возможные ходы
vis = [[False for i in range(m)] for j in range(n)] # посещенные клетки
queue = [[i, 0] for i in range(n) if matrix[i][0]==0]
p = [[[-1, -1] for i in range(m)] for j in range(n)] # список родительских клеток
vi, vj = -1, -1
while len(queue) != 0:
    a, b = queue.pop(0)
    vis[a][b] = True
    if b == m-1: # нашел выход
        vi, vj = a, b
        break
    for move in moves:
        i, j = a+move[0], b+move[1]
        if i < 0 or i > n or j < 0 or j > m or \
            matrix[i][j] == 1 or vis[i][j]: continue
        p[i][j] = [a, b]
        queue += [[i, j]]
```

Рисунок 3.1 — Код функции поиска в ширину для данной задачи

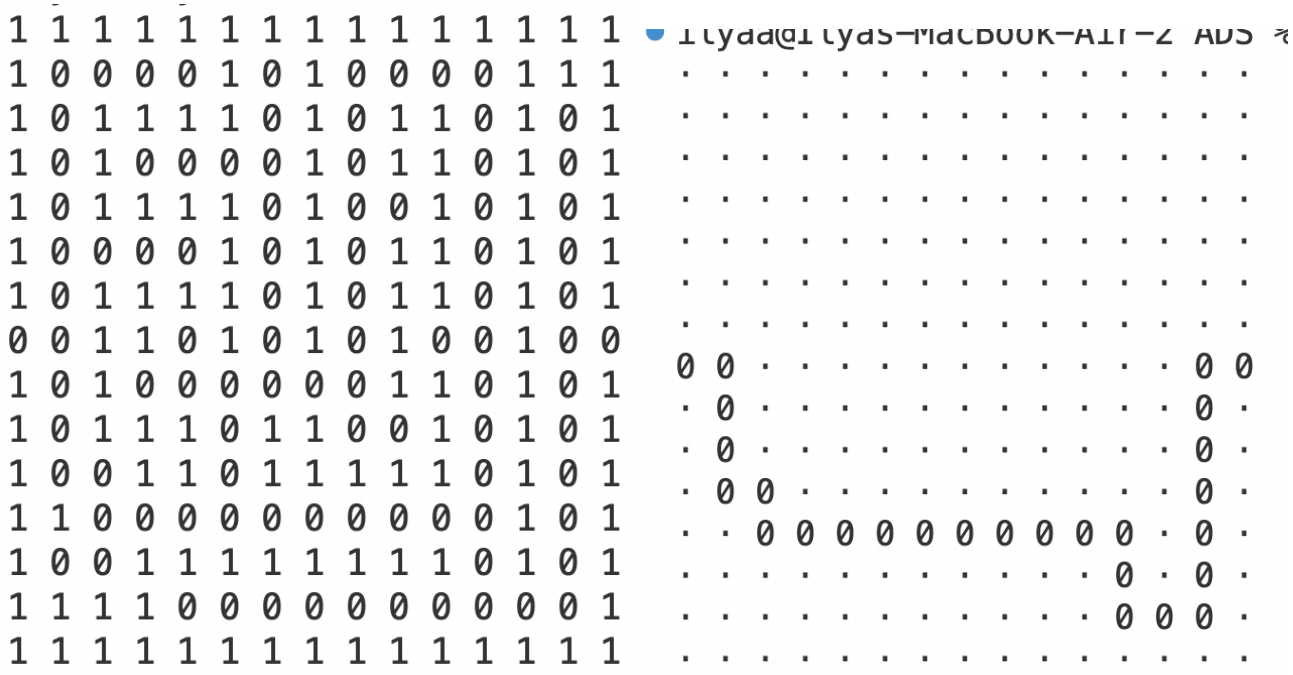

Рисунок 3.2 — Пример работы программы для заданной в условии задачи матрицы

ЗАКЛЮЧЕНИЕ

Таким образом, для каждой задачи была написана программа на языке программирования Python. Были изучены алгоритмы DFS и BFS, которые важны в теории графов и используются для решения многих интересных задач, варианты которых представлены в отчете. Для каждого алгоритма приведена программная реализация.

Все программы можно найти на репозитории в GitHub [1].

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. GitHub [Электронный ресурс]: `https://github.com/estle/itmo-uni/tree/main/sem1/ADS/lab5` (дата обращения 15.12.2022).