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

Факультет Инфокоммуникационных Технологий

**Лабораторная работа №4**

Выполнили:

Новиков Н. В.

Ходакова М. А.

Борисевич А. В.

Проверил Мусаев А. А.

Санкт-Петербург,

2023

**Содержание**

[ВВЕДЕНИЕ 3](#_Toc150191149)

[1. Правильная скобочная структура 4](#_Toc150191150)

[2. Алгоритмы поиска 6](#_Toc150191151)

[**2.1 Алгоритм поиска в глубину** 6](#_Toc150191152)

[**2.2 Поиск в ширину** 9](#_Toc150191153)

[3. Программа для решения лабиринта 10](#_Toc150191154)

[**3.1 Для решения квадратной матрицы** 10](#_Toc150191155)

[**3.2 Решение кубической матрицы** 12](#_Toc150191156)

[ЗАКЛЮЧЕНИЕ 13](#_Toc150191157)

# ВВЕДЕНИЕ

Цель лабораторной работы №4 состояла в знакомстве с LIFO и FIFO. Для её достижения были поставлены следующие задачи:

1. Написание программы, которая определяет, является ли введенная скобочная структура правильной.
2. Составление алгоритма поиска кратчайшего пути в глубину и в ширину;
3. Написать программу, решения лабиринта.

# 1. Правильная скобочная структура

Пользователь вводит скобочную структуру в формате строки, каждую

скобку помещаем в список struc как отдельный строковый символ.

Пишем функцию, которая принимает список. В stack будут храниться

открывающие скобки, check отвечает на вопрос (это правильная скобочная пара?), er\_num – список, в котором хранятся порядковые номера элементов,

из-за которых ломается структура (может быть несколько ошибок).

Проходимся по списку: если i-тый элемент является открывающей скобкой, то добавляем в stack порядковый номер элемента и сам элемент. Иначе, если i-тый элемент является закрывающей скобкой и длина stack не равна 0 (если длина stack была равна 0, то это неправильная скобочная пара, так как открывающей скобки у этой пары нет, следовательно, check – ложь), то берем последнюю открывающую скобку из stack, убираем ее из stack и проверяем вид скобки, если скобка такого же вида, то продолжаем, если нет, тогда это неправильная скобочная пара, значит check – ложь.

Если после прохождения всех элементов списка по циклу длина stack не равна 0 (т.е. остались открывающие скобки без пары), тогда кладем в er\_num порядковый номер самого первого элемента в стеке (т.к. это первый элемент, с которого все ломается).

Если check – Истина и длина stack равна 0, тогда скобочная структура является правильной, выводим correct. Иначе скобочная структура неправильная, выводим incoorect и минимальный порядковый номер элемента из er\_num, с него ломается структура.

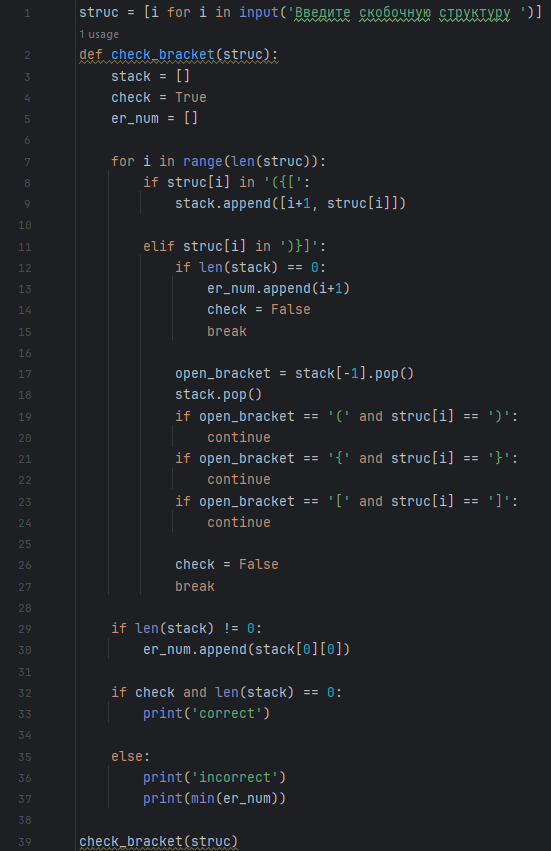


Рисунок 1 - Проверка правильной скобочной структуры

# 2. Алгоритмы поиска

## **2.1 Алгоритм поиска в глубину**

Алгоритм поиска в глубину позволяет построить обход графа, при котором посещаются все вершины, доступные из начальной вершины. В общем случае алгоритм поиска в глубину работает так:

1. Пойти в какую-нибудь смежную вершину.
2. Обойти все, что доступно из этой вершины.
3. Вернуться в начальную вершину.
4. Повторить алгоритм для всех остальных вершин, смежных из начальной.

В контексте решения задания два и поиска кратчайшего пути, напишем программу, действующую по следующему алгоритму (допустим, нам необходимо найти путь из вершины 1 в вершину 2, количество путей считаем равным количеству ребёр между вершинами):

Начинаем обход в глубину с вершины, из которой мы строим маршрут из 1,

1. Выясняем, что 1 смежна, например, с 3 и 4,
2. Выбираем первое ребро (3) и снова смотрим список смежных вершин,
3. Выясняем, что 3 смежна с 4 и 1,
4. Следующей точкой выбираем 4 и смотрим список смежных вершин,
5. Возвращаемся к предыдущему пункту в списке — к 4,
6. Рассматриваем следующую вершину в списке смежности — 5,
7. Продолжая обход, мы обнаружим новый маршрут — 1, 3, 4, 5, 2,
8. Заканчивая обход списков смежности, мы постоянно возвращаемся к
9. предыдущей вершине. Другими словами, перебрав все пути из 4, мы
10. откатываемся к 3, а затем — к 1.

Чтобы алгоритм не зациклился, нужно по мере обхода графа сохранять вершины, которые мы посетили и отбрасывать их при переборе. Вершину надо удалять, когда мы откатываемся назад по маршруту, потому что в другой ветке мы снова можем попасть в тот же город, но уже другим путем. Код программы, работающей по данному алгоритму, приведен на рисунке 2.

Граф в программе должен быть представлен в виде списка смежности, в данном случае он назван graph (сам граф изображен на рисунке 2). За хранение имен пройденных вершин отвечает параметр visited.

Параметр start содержит значение текущей вершины, а end — значение конечной вершины. Если текущая вершина совпадает с конечной, значит был найден очередной путь, добавляем все города из списка посещенных в список маршрутов routes.

Массив visited заполняется внутри функции. В первой строке мы помещаем в него текущую вершину start, а в последней — удаляем ее.

Параметр routes не только входной, но и выходной. Это массив, в который складываются найденные маршруты. Когда функция dfsобойдет весь граф, в массиве окажутся все способы добраться из начальной точки в конечную.

В случае, если текущая вершина не совпадает с конечной выполняется следующее: внутри цикла for перебираются все соседние вершины с текущей.

Если какая-то из них есть в массиве visited, она не рассматривается. В противном случае вершина принимается за текущую, и функция вызывается рекурсивно уже от неё.

После завершения работы функции (то есть, когда все вершины пройдены), программа выводит минимальный по длине путь из начальной вершины в конечную.

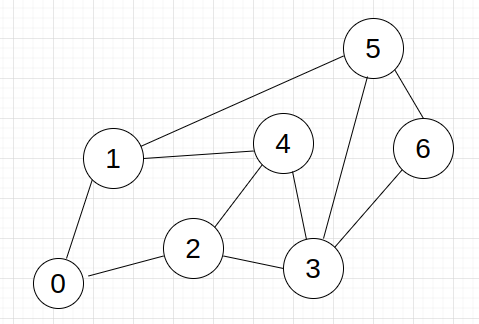


Рисунок 2 - Поиск в глубину

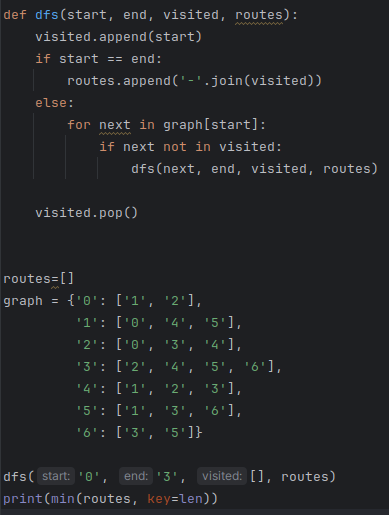


Рисунок 3 - Программа поиска в глубину

## **2.2 Поиск в ширину**

Поиск в ширину — один из методов обхода графа. Пусть задан граф и выделена исходная вершина s. Алгоритм поиска в ширину систематически обходит все ребра для «открытия» всех вершин, достижимых из s, вычисляя при этом расстояние (минимальное количество рёбер) от s до каждой достижимой из s вершины.

Поиск в ширину имеет такое название потому, что в процессе обхода мы идём вширь, то есть перед тем как приступить к поиску вершин на расстоянии n + 1, выполняется обход вершин на расстоянии n .

Алгоритм поиска кратчайшего пути в невзвешенном графе с использованием BFS (Breadth-First Search) в виде программы приведен на рисунке 4.

Параметр start содержит значение текущей вершины, а end — значение конечной вершины.

В начале работы создаём массив queue — очередь, содержащий кортежи вида (текущая вершина, посещенные вершины). Пока очередь не пуста выполняется следующее:

1. Начинаем обход смежных вершин. Извлекаем из очереди первый кортеж, присваивая переменной point значение текущей вершины, а переменной visited — посещенные вершины.

2. Если текущая вершина совпадает с конечной, значит был найден очередной путь (он состоит из посещенных вершин). Программа возвращает найденный путь.

3. В случае, если текущая вершина не совпадает с конечной выполняется следующее: внутри цикла for перебираются все соседние вершины с текущей.

Если какая-то из них есть в массиве visited, она не рассматривается. В противном случае вершина вместе с путём, которым в неё можно попасть, добавляется в конец очереди.

4. Алгоритм работает до тех пор, пока не будет возвращён путь или пока не будут рассмотрены все вершины. Если рассмотрение вершин окончено, но путь не был найден, возвращается „-“.

Этот алгоритм гарантирует нахождение кратчайшего пути в невзвешенном графе, так как исследует вершины постепенно, начиная с ближайших к начальной вершине.

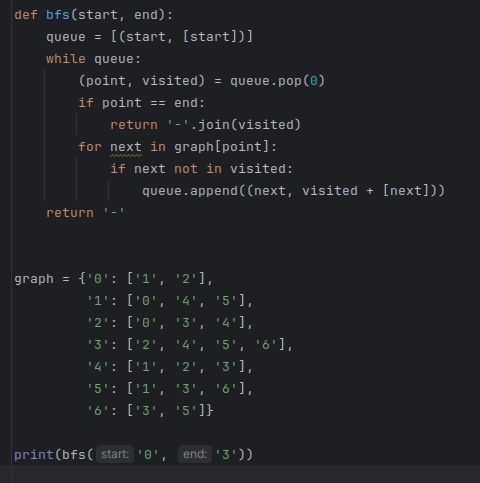


Рисунок 4 - Алгоритм поиска в ширину

# 3. Программа для решения лабиринта

## **3.1 Для решения квадратной матрицы**

Для решения задачи было создано 2 файла. Один из которых служит просто для адекватного вывода. Сначала была создана матрица 15x15, состоящая из 0 и 1. После чего было создано 2 функции. Первая – удаляет из окончательного массива ненужные значения, а вторая удаляет те, которые по бокам не имеют прохода.

Сам алгоритм – выделяем все 0, которые есть в лабиринте и с помощью первых 2 функций находит кратчайший маршрут (Рисунок 5,6)

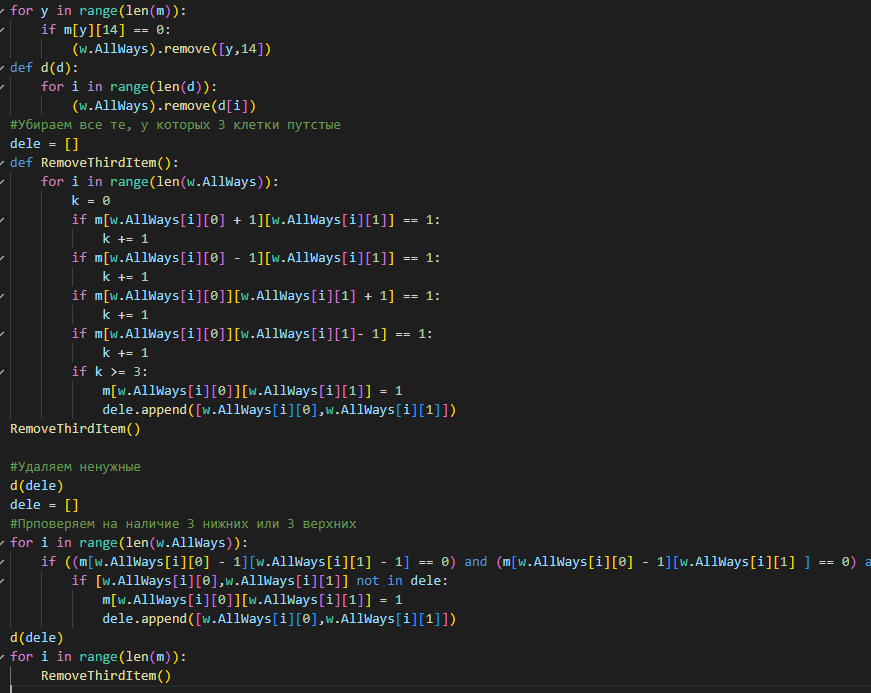


Рисунок 5 - Алгоритм решения лабиринта

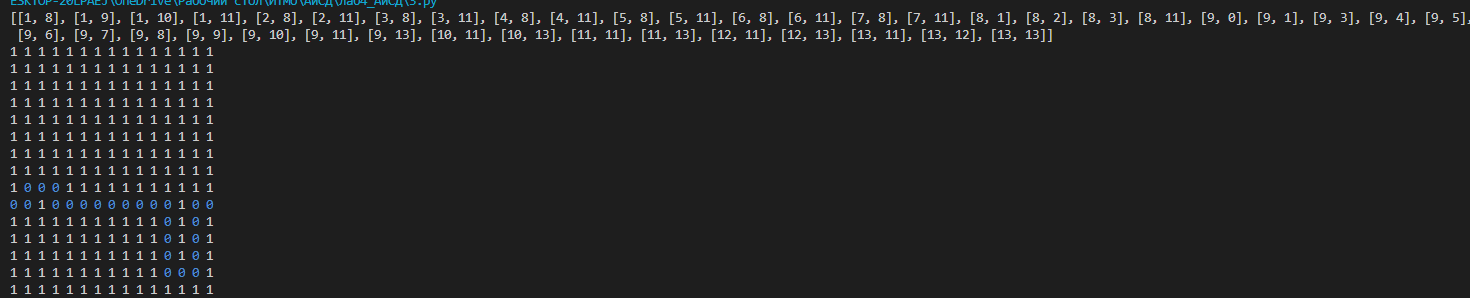


Рисунок 6 - Пример вывода

На выходе имеется либо координаты точек, по которым нужно пройти, либо сам путь на исходной матрице.

## **3.2 Решение кубической матрицы**

Для решения кубической матрицы был задействован алгоритм поиска в ширину (Рисунок 7). Для более удобной работы с массивом был импортирован встроенный модуль deque.

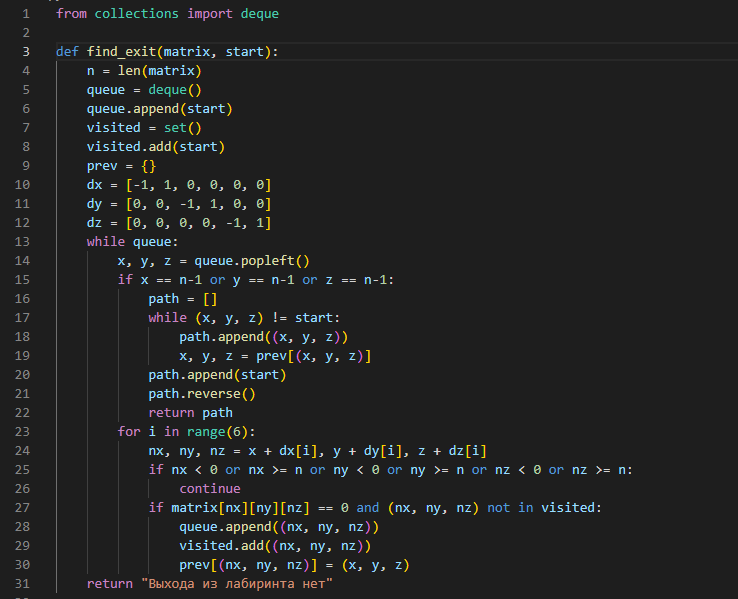


Рисунок 7 - Решение кубической матрицы

Сначала находится размерность матрицы, после чего создается массив для хранения клеток, в которых уже побывали. После чего просматриваем 6 соседних клеток и ищем маршрут до конечной точки

На выходе получаем кратчайший маршрут до выхода (Рисунок 8).



Рисунок 8 - Вывод программы

# ЗАКЛЮЧЕНИЕ

В ходе выполнения лабораторной работы №4 были получены навыки работы с LIFO и FIFO и достигнуты поставленные задачи