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

Факультет инфокоммуникационных технологий

Лабораторня работа №4

Выполнили:
Стафеев И.А., Лапшина Ю.С., Килебе Нтангу Дьевина
Проверил
Мусаев А.А.

Санкт-Петербург,
2023

СОДЕРЖАНИЕ

Стр.

ВВЕДЕНИЕ

Цель работы: изучить структуры данных стек и очередь и связанные с ними алгоритмы обхода графов. Для достижения цели были поставлены следующие задачи:

- изучить структуры данных стек и очередь;
- изучить алгоритмы обхода графов;
- сравнить, для каких задач подходит поиск в глубину, а для каких - поиск в ширину;
- написать программу для определения правильной скобочной последовательности;
- написать программу для поиска кратчайшего пути на карте;
- написать программу для поиска выхода из лабиринта.

1 Определение правильной скобочной последовательности

Для реализации данной программы неободимо использовать стек. Для этого был задействован класс LifoQueue модуля queue. С его помощью был реализован принцип LIFO (last in — first out, «последним пришёл — первым вышел»).

Алгоритм определения скобочной последовательности представлен в виде функции *is_psp*. Так как по определению правильная скобочная последовательность (далее - псп) - последовательность, в которой каждой «открывающей» скобке найдется в пару «закрывающая», причём они должны принадлежать одному и тому же типу (круглые, фигурные, квадратные), был создан словарь *brackets_tuple*, где как раз и представлены пары скобок разного типа. Стек хранится в переменной *stack*, использующей класс $LifoQueue$.

Во время перебора всех скобок, осуществляется проверка на вид скобки («открывающий» или «закрывающий»). Если скобка «открывающая», то она и её индекс добавляются в стек. Если нет - производится проверка стека. Если стек пуст, то псп неправильная, а индекс первой скобки, нарушающей порядок - индекс рассматриваемой «закрывающей» скобки (это значит, что всем «открывающим» скобкам уже нашлась пара и «закрывающая» скобка осталась без пары). Если стек не пуст, то осуществляется проверка типа скобки. Если тип рассматриваемой скобки не совпадает с типом последней скобки, добавленной в стек, то псп неправильная, а индекс первой скобки, нарушающей порядок - снова индекс рассматриваемой «закрывающей» скобки (данный случай означает, что в пару «открывающей» скобке сопоставлена «закрывающая» скобка неподходящего типа).

После перебора всех скобок производится проверка на пустоту стека, так как в нём могли остаться скобки, а значит, псп неправильная. В таком случае индексом первой скобки, нарушающей порядок будет индекс скобки, находящейся в самом низу стека. Если стек был пуст - псп правильная.

Код реализованной программы представлен на рисунке: 1.

```
1 usage   dymonyx *
def is_psp(sequence:str):
    brackets_tuple = {')':'(', ']':'[','}':'{'}
    stack = LifoQueue()
    for i, bracket in enumerate(sequence):
        if bracket in list(brackets_tuple.values()):
            stack.put([i, bracket])
        elif bracket in list(brackets_tuple.keys()):
            if not stack.qsize():
                return (f'сп неправильная, индекс первой неправильной скобки:{i}')
            else:
                _, last = stack.get()
                if last != brackets_tuple[bracket]:
                    return (f'сп неправильная, индекс первой неправильной скобки:{i}')
    if stack.qsize():
        while stack.qsize():
            last_index, last = stack.get()
        return (f'сп неправильная, индекс первой неправильной скобки:{last_index}')
    return "сп правильная"
```

Рисунок 1 — Код проверки правильности скобочной последовательности

Пример работы программы также представлен на рисунке: 2.

```
введите длину скобочной последовательности: 15
({()([][()]({[(
сп неправильная, индекс первой неправильной скобки:6
```

Рисунок 2 — Пример работы программы по проверке правильности скобочной последовательности

2 Поиск кратчайшего пути на карте

Для этой задачи была выбрана карта Бельгии. Требуется найти кратчайший путь между городами Брюссель и Бастонь. По сути карта является графом, поэтому для нахождения путей можно применить алгоритмы на графах. Понятно, что обычный обход в глубину и в ширину не всегда может найти кратчайший путь во взвешенном графе, поэтому необходимо модифицировать алгоритмы для нахождения всех путей между двумя точками, чтобы потом вычислить натболее короткий из найденных путей. В обоих случаях будем считать, что *startnode* - это Брюссель, а *targetnode* - это Бастонь. Здесь представлены только алгоритмы, программной реализации нет, так как для задания она не требовалась. Алгоритм нахождения кратчайшего пути между заданными городами с использованием обхода в ширину представлен на листинге 1.

Listing 1: Алгоритм поиска кратчайшего пути обходом в ширину

```
1  procedure BFS(start node, target node)
2      Initialize queue queue
3      Initialize array path
4      Initialize array all_paths
5      Add start node to the end of path
6      Enqueue path in queue
7      while queue is not empty do
8          Dequeue the left element from queue and put it in path
9          Get last element from path and put it in last
10         if last is the target node then
11             Add path to the end of all_paths
12         end if
13         for all neighbors of node last  do
14             if neighbour is unvisited then
15                 Copy path to new_path
16                 Add neighbour to the end of new_path
17                 Enqueue new_path in queue
18             end if
```

```
19          end for
20      end while
21
22      Initialize int min_path=10^12
23      for all path in all_paths  do
24          Sum weights of each edges v_i -> v_{i-1}
25          if sum is less then min_sum then
26              Set sum as value for min_sum
27          end if
28      end for
29      return min_sum
```

Алгоритм нахождения кратчайшего пути с использованием обхода в глубину представлен на листинге 2

Listing 2: Алгоритм поиска кратчайшего пути обходом в глубину

```
1  procedure DFS(start node, target node, visited, path)
2      Mark start node as visited
3      Add start node to the end of path
4      if start node is target node then
5          Add path to the end of all_paths
6      else
7          for all neighbors of node start node  do
8              if neighbour is unvisited then
9                  DFS(neighbour, target node, visited, path)
10             end if
11         end for
12     end while
13     Remove start node from path
14     Mark start node as unvisited
15
16 program main()
17     Initialize array all_paths
18     DFS(start node, target node, [false....false], [])
```

```
19      Initialize int min_path=10^12
20      for all path in all_paths  do
21          Sum weights of each edges v_i -> v_{i-1}
22          if sum is less then min_sum then
23              Set sum as value for min_sum
24          end if
25      end for
26      print  min_sum
```

Несмотря на то, что оба алгоритма имеют одинаковую временную сложность, они отличаются по используемой памяти. Поиск в ширину значительно проигрывает по памяти поиску в глубину. Однако есть и некоторые другие аспекты, когда один алгоритм предпочтительнее другого.

Использовать поиск в глубину стоит, если:

- Необходимо найти кратчайший путь между двумя вершинами, а не все пути
- Действует сильно ограничение по памяти
- Искомые пути являются примерно равными по длине
- Если решения находятся на глубоком уровне

Использовать поиск в ширину стоит, если:

- Необходимо найти все пути между двумя вершинами, а не только кратчайший
- Ограничения по памяти незначительные или отсутствуют
- Искомые пути являются могут сильно отличаться по длине
- Если решения не находятся на глубоком уровне

3 Поиск выхода из лабиринта

Будем решать задачу в общем случае и сразу для трехмерного лабиринта. Есть трехмерная матрица $n \times m \times k$, заполненная нулями и единицами, где 0 обозначает проход, а 1 - стену. Требуется найти выход из лабиринта, учитывая, что точка входа задается пользователем.

Одно из возможных решений заключается в том, что матрицу можно представить как граф, где каждая точка матрицы - это вершина графа, а смежные с ней вершины - это точки из соседства фон Неймана для точки матрицы (то есть точки, имеющие общую грань с исходной). Тогда можно применить алгортм поиска в ширину для полученного графа, запустив его из начальной точки. Основанием для выхода из поиска будем считать точку (x, y, z), что $x \in \{0, n-1\}$, или $y \in \{0, m-1\}$, или $z \in \{0, k-1\}$, и при этом эта точка еще не посещена (нужно, чтобы исключить выход из самой начальной точки). Таким образом будет найден путь до ближайшего выхода из лабиринта, если он существует.

Для получения соседей точки написаны две функции - *check_in_borders* и *get_neighbours*. Первая возвращает булевое значение, попадают ли координаты точки в границы исходной матрицы, а вторая возвращает для аргумента-точки все точки из окрестности фон Неймана, которые при это попадают в границы матрицы. Код функций представлен на рисунке 3.

```python
def check_in_borders(x, y, z, n, m, k):
    """Проверяет, что координаты точки находятся в пределах поля"""
    return 0 <= x <= n - 1 and 0 <= y <= m - 1 and 0 <= z <= k - 1


def get_neighbours(x, y, z, n, m, k):
    """Возвращает соседние точки для (x, y), которые принадлежат полю"""
    shifts = [(-1, 0, 0), (1, 0, 0), (0, -1, 0), (0, 1, 0),  (0, 0, -1), (0, 0, 1)]
    neighbours = [(x + i, y + j, z + p) for i, j, p in shifts]
    return [i for i in neighbours if check_in_borders(*i, n, m, k)]
```

Рисунок 3 — Код функций для полученния точек из окрестности фон Неймана

Для нахождения выхода из лабиринта (и пути до выхода) написана функция bfs, которая является небольшой модификацией обычного алгоритма поиска в ширину: в качестве очереди используется объект класса *deque* из модуля *collections*; массив *visited* посещенных вершин является трехмерным; после завершения цикла алгоритма начинается цикл восстановления пути, который использует словарь *parents*, который для каждой посещенной вершины сохраняет вершину, из которой в нее пришли; внутри цикла добавляется условие выхода, описанное раньше. Код алгоритма представлен на рисунке 4.

```python
def bfs(start: tuple[int], n: int, m: int, k: int, field: list[list[list[int]]]):
    """Поиск в глубину. Находит ближайший выход из лабиринта"""
    queue = deque([start])
    end = None
    parents = {}
    visited = [[[0 for _ in range(m)] for _ in range(n)] for _ in range(k)]
    visited[start[0]][start[1]][start[2]] = 1

    while queue:
        (z0, x0, y0) = queue.popleft()
        if (x0 in (0, n - 1) or y0 in (0, m - 1) or z0 in (0, k - 1)) and not visited[z0][x0][y0]:
            end = (z0, x0, y0)
            break
        visited[z0][x0][y0] = 1

        for (x, y, z) in get_neighbours(x0, y0, z0, n, m, k):
            if not visited[z][x][y] and field[z][x][y] == 0:
                parents[z, x, y] = (z0, x0, y0)
                queue.append((z, x, y))

    if end is None:  # Выхода не оказалось
        return []

    # Получение пути
    path = []
    while end != start:
        path.append(end)
        end = parents[end]
    path.append(end)
    return path[::-1]
```

Рисунок 4 — Код модицицированного алгоритма поиска в ширину

Чтобы наглядно показать работу программы, был создан класс *Board*, отображающий исходную матрицу с нанесенным на нее путем до выхода из лабиринта. В двухмерном случае это просто вывод в консоль исходной матрицы, где точки пути заменены на символ крестика, а трехмерном случае используется модуль *matplotlib* для создания трехмерной фигуры

лабиринта. Описывать подробно устройство классов здесь нет смысла, в решении задачи они играют незначительную роль.

Нетрудно заметить, что асимтотическая сложность алгоритмам поиска выхода составит $O(n \times m)$ в двумерном случае и $O(n \times m \times k)$ в трехмерном случае. Пример работы алгоритма для двумерной матрицы представлен на рисунке 5, для трехмерной - на рисунке 6. Пользователь должен сначала ввести размер матрицы, затем саму матрицу и точку входа в лабиринт.

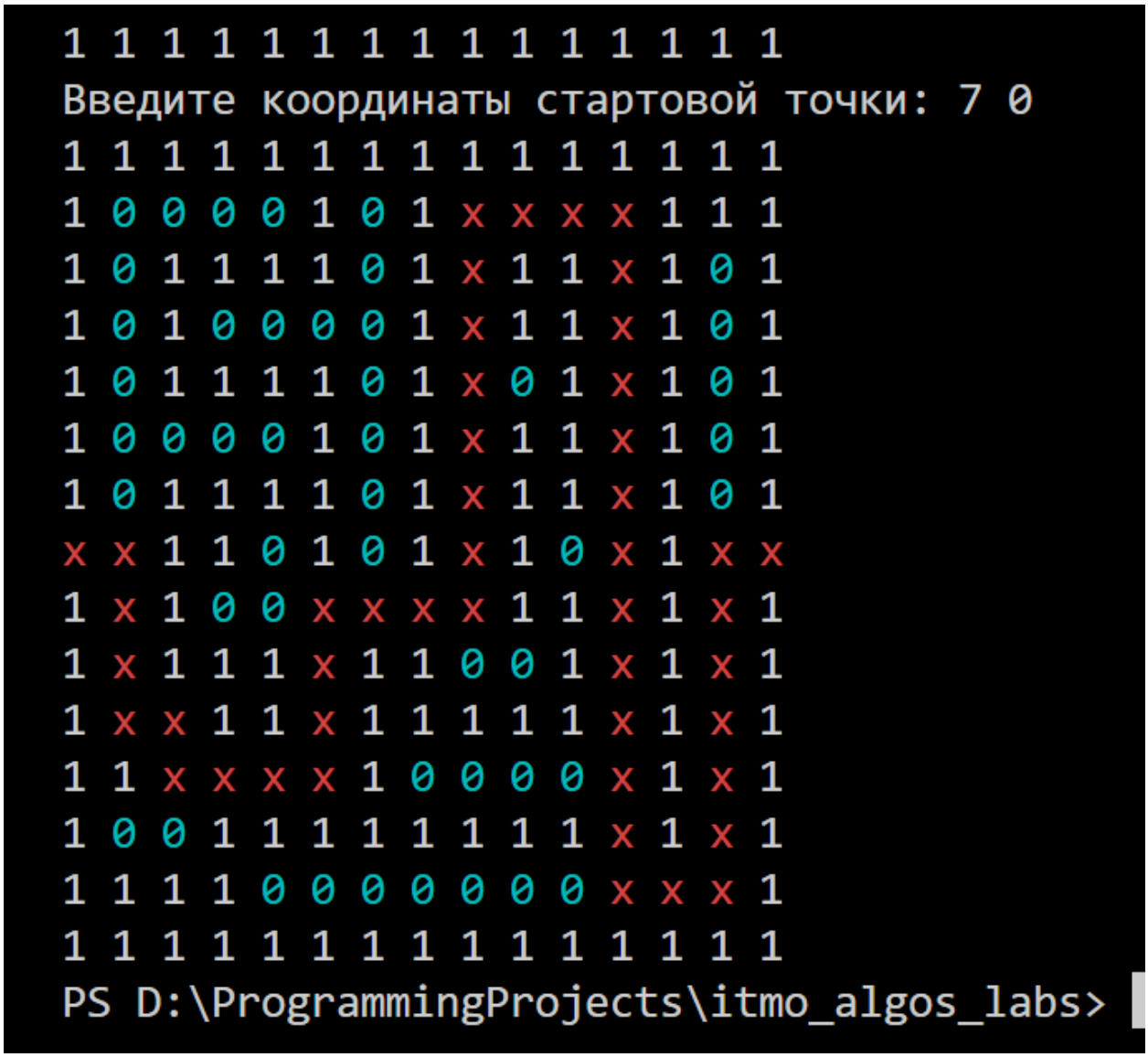

Рисунок 5 — Поиск выхода из двумерного лабиринта

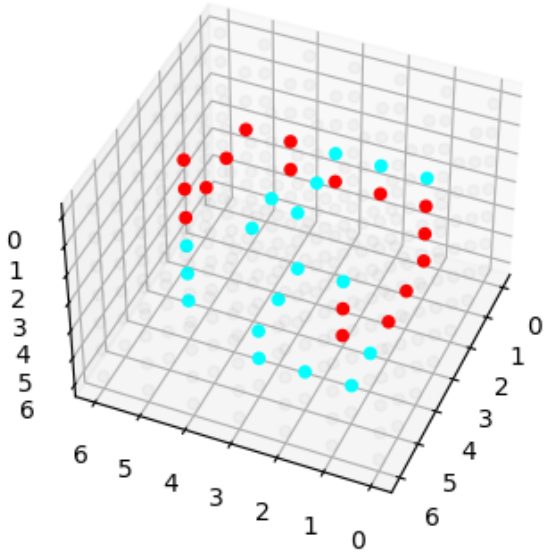

Рисунок 6 — Поиск выхода из трехмерного лабиринта

ЗАКЛЮЧЕНИЕ

Результатом выполнения работы стало повышение навыков по использованию стеков, очередей и применения алгоритмов обхода графов. В ходе работы были написаны: программа для определения правильной скобочной последовательности; программа для поиска кратчайшего пути для карты метро; программа для нахождения выхода из лабиринта, заданного матрицей. Также были сделаны выводы, для каких задач подходят алгоритмы поиска в глубину и в ширину. Таким образом, полученные знания позволят более эффективно применять новые структуры данных и алгоритмы в практической деятельности в зависимости от типа задачи.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Ахо А.В., Хопкрофт Д.Э., Ульман Дж.Д. Алгоритмы и структуры данных [Электронный ресурс] – URL: https://vk.com/wall-114485185_260 (дата обращения 28.10.2023)

2. Python Docs. collections - Container datatypes [Электронный ресурс] - URL: https://docs.python.org/3/library/collections.html#collections.deque (дата обращения 28.10.2023)

3. Matplotlib 3.8.0 documentation [Электронный ресурс] - URL: https://matplotlib.org/stable/index.html (дата обращения 28.10.2023)

4. Quora. Which one is the best DFS or BFS to find all the possible paths between two nodes? [Электронный ресурс] - URL: https://www.quora.com/Which-one-is-the-best-DFS-or-BFS-to-find-all-the-possible-paths-be (дата обращения 29.10.2023)

5. Стек в Python, LifoQueue [Электронный ресурс] - URL: https://pythonpip.ru/examples/stek-python (дата обращения 30.10.2023)