|  |
| --- |
| Федеральное государственное бюджетное образовательное учреждение  высшего образования  **"МИРЭА – Российский технологический университет"**  **РТУ МИРЭА** |

Институт комплексной безопасности и специального приборостроения

Кафедра КБ-14 «Цифровые технологии обработки данных»

**ОТЧЕТ   
о выполнении лабораторной работы №3**

**«Алгоритмы на графах»**

**по дисциплине   
«Алгоритмы и структуры данных»**

**Вариант № 27**

Выполнил: студент 2 курса

группы \_\_\_\_\_\_\_\_\_\_\_

шифр \_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_

*(фио студента)*

Проверил: *\_\_\_\_\_\_\_\_\_\_\_\_\_\_*

Москва 2022 г.

**Вариант № 27.**

**Алгоритм:** определить внешний радиус невзвешенного ориентированного   
графа методом обхода в ширину. (Внешним радиусом графа   
будем называть наибольшее среди кратчайших расстояние от   
центра до какого-либо узла.) Вывести значение, а также   
соответствующие ему цепи.

**Способ представления графа:** Список смежности

**Теория о Графах.**

Граф — это математический объект, который состоит из точек и линий, которые их соединяют. Точки называют вершинами графа, а линии — ребрами. Граф, ребра которого имеют направления, называется ориентированным, если же ребра графа не имеют направления, то такой граф называется неориентированным.



Рисунок 1. – Пример ориентированного графа.

Матрица смежности — это вид представления графа в виде матрицы, когда пересечение столбцов и строк задаёт дуги. Используя матрицу смежности, можно задать вес дуг и ориентацию. Каждая строка и столбец матрицы соответствуют вершинам, номер строки соответствует вершине, из которой выходит дуга, а номер столбца - в какую входит дуга. Пример матрицы смежности графа, изображенного на рисунке 1, представлен на рисунке 2.



Рисунок 2. – Пример матрицы смежности.

Список смежности — один из способов представления графа в виде коллекции списков вершин. Каждой вершине графа соответствует список, состоящий из «соседей» этой вершины.

Для поиска центра графа методом поиска в ширину составим при помощи него матрицу расстояний. Матрицей расстояний назовем матрицу, где в i-ой строке каждый j элемент – это минимальное расстояние от вершины i до вершины j (если существует путь проходящий через вершины графа).

Далее для каждой строки матрицы, где все элементы ненулевые, найдем максимальное значение расстояния. А потом из списка максимальных расстояний найдем минимальное, номер строки, которой принадлежит данное расстояние и является центром графа, а расстояние – радиус

Далее обходом в глубину от центра графа найдем цепи с длинной равной радиусу.

**Листинг программы.**

import json

from typing import Dict, List, Optional, Tuple

class Vertex:

    \_index: int

    label: str

    def \_\_init\_\_(self, index: int, label: Optional[str] = ""):

        self.\_index = index

        self.label = label

    @property

    def index(self) -> int:

        return self.\_index

    def \_\_repr\_\_(self):

        return f"Vertex[i: {self.\_index}, label: {self.label}]"

class Edge:

    \_vertices: Tuple[int, int]

    len: int

    def \_\_init\_\_(self, v1: int = None, v2: int = None, len: int = 1):

        self.\_vertices = (v1, v2)

        self.len = len

    @property

    def vertices(self) -> Tuple[int, int]:

        return self.\_vertices

    def \_\_repr\_\_(self):

        return f"Edge[{self.\_vertices[0]} -> {self.\_vertices[1]}, {self.len}]"

class Graph:

    \_vertices: List[Vertex]

    \_edges: List[Edge]

    def \_\_init\_\_(self) -> None:

        self.\_vertices = []

        self.\_edges = []

    def \_init\_vertices(self, dim: int):

        self.\_vertices = []

        for i in range(dim):

            self.\_vertices.append(Vertex(i))

    def \_load\_adj\_matrix(self, data: List[List[int]], dim: int) -> None:

        self.\_init\_vertices(dim)

        for i, vertex in enumerate(data):

            for j, edge\_len in enumerate(vertex):

                if edge\_len > 0:

                    self.add\_e(i, j, edge\_len)

    def \_load\_adj\_list(self, data: List[List[int]], dim: int) -> None:

        self.\_init\_vertices(dim)

        for i, adj in enumerate(data):

            for j in adj:

                self.add\_e(i, j, 1)

    def \_load\_edge\_list(self, data: List[List[int]], dim: int) -> None:

        self.\_init\_vertices(dim)

        for edge in data:

            if len(edge) == 3:

                v1, v2, edge\_len = edge

            else:

                v1, v2, edge\_len = \*edge, 1

            self.add\_e(v1, v2, edge\_len)

    def \_load\_inc\_matrix(self, data: List[List[int]], dim: int) -> None:

        self.\_init\_vertices(dim)

        transposed\_data = list(zip(\*data))  # транспонирование

        for edge in transposed\_data:

            v1 = None

            v2 = None

            v1\_v2\_len = 0

            v2\_v1\_len = 0

            for j in range(dim):

                if edge[j] != 0 and v1 is None:

                    v1 = j

                    v1\_v2\_len = edge[j]

                elif edge[j] != 0 and v2 is None:

                    v2 = j

                    v2\_v1\_len = edge[j]

                    break

            if v1\_v2\_len > 0:

                self.add\_e(v1, v2, v1\_v2\_len)

            if v2\_v1\_len > 0:

                self.add\_e(v2, v1, v2\_v1\_len)

    def \_load\_labels(self, labels: Dict[str, str]) -> None:

        for index, label in labels.items():

            index = int(index)

            self.\_vertices[index].label = label

    def load(self, filename: str) -> None:

        with open(filename, "r", encoding="utf8") as f:

            graph\_json: dict = json.load(f)

        \_format = graph\_json.get("format", None)

        if \_format is None:

            raise Exception("Не указан формат для задания графа! Смотри документацию!")

        \_dim: Optional[str] = graph\_json.get("dim", None)

        if \_dim is None or not isinstance(\_dim, int):

            raise Exception("Не укаказан/верный размер графа! Смотри документацию!")

        \_data: Optional[List[List[int]]] = graph\_json.get("data", None)

        if \_data is None:

            raise Exception(

                "Не заданны данные для построения графа! Смотри документацию!"

            )

        \_labels: Optional[Dict[str, str]] = graph\_json.get("labels", None)

        if \_format == "adj\_matrix":

            self.\_load\_adj\_matrix(\_data, \_dim)

        elif \_format == "adj\_list":

            self.\_load\_adj\_list(\_data, \_dim)

        elif \_format == "edge\_list":

            self.\_load\_edge\_list(\_data, \_dim)

        elif \_format == "inc\_matrix":

            self.\_load\_inc\_matrix(\_data, \_dim)

        else:

            raise Exception("Неизвестный формат задания графа! Смотри документацию!")

        if \_labels:

            self.\_load\_labels(\_labels)

    def first(self, v: int) -> None | int:

        for edge in self.\_edges:

            if edge.vertices[0] == v:

                return edge.vertices[1]

        return None

    def next(self, v: int, i: int) -> None | int:

        state = 0

        next\_index = None

        for edge in self.\_edges:

            if edge.vertices[0] != v:

                if state == 0:

                    continue

                if state == 1:

                    break

            else:

                state = 1

                if edge.vertices[1] > i:

                    next\_index = edge.vertices[1]

                    break

        return next\_index

    def vertex(self, v: int, i: int) -> None | int:

        state = 0

        vertices\_set: List[int] = []

        for edge in self.\_edges:

            if edge.vertices[0] != v:

                if state == 0:

                    continue

                if state == 1:

                    break

            else:

                state = 1

                vertices\_set.append(edge.vertices[1])

        if i < len(vertices\_set):

            return vertices\_set[i]

        else:

            return None

    def add\_v(self, index: int, label: str = "") -> None:

        for vertex in self.\_vertices:

            if vertex.index == index:

                raise Exception(f"Вершина с таким индексом уже существует! ({index})")

        self.\_vertices.append(Vertex(index, label))

    def add\_e(self, v1: int, v2: int, edge\_len: int = 1) -> None:

        found\_v1 = False

        found\_v2 = False

        for vertex in self.\_vertices:

            if vertex.index == v1:

                found\_v1 = True

            elif vertex.index == v2:

                found\_v2 = True

        if not (found\_v1 and found\_v2) or v1 == v2:

            raise Exception(f"Невозможная пара индексов вершин! ({v1}, {v2})")

        else:

            self.\_edges.append(Edge(v1, v2, edge\_len))

    def del\_v(self, index: int) -> None:

        for vertex in self.\_vertices:

            if vertex.index == index:

                break

        else:

            raise Exception(f"Вершина с таким индексом не существует! ({index})")

        edges\_to\_remove = []

        for edge in self.\_edges:

            if edge.vertices[0] == index or edge.vertices[1] == index:

                edges\_to\_remove.append(edge)

        for edge in edges\_to\_remove:

            self.\_edges.remove(edge)

    def del\_e(self, v1: int, v2: int) -> None:

        for edge in self.\_edges:

            if edge.vertices[0] == v1 and edge.vertices[1] == v2:

                break

        else:

            raise Exception(f"Ребра с таким набором вершин не существует! ({v1}, {v2})")

    def to\_adj\_matrix(self) -> List[List[int]]:

        adj\_matrix = [

            [0 for \_ in range(len(self.\_vertices))] for \_ in range(len(self.\_vertices))

        ]

        for edge in self.\_edges:

            adj\_matrix[edge.\_vertices[0]][edge.\_vertices[1]] = edge.len

        return adj\_matrix

    def as\_dict(self) -> Dict:

        return {"Vertices": self.\_vertices, "Edges": self.\_edges}

    @property

    def vertices(self) -> List[Vertex]:

        return self.\_vertices

    @property

    def edges(self) -> List[Edge]:

        return self.\_edges

def BFS(matrix: List[List[int]]) -> List[List[int]]:

    distance\_matrix = [[0] \* len(matrix) for i in range(len(matrix))]

    def \_bfs(v: int):

        queue = [v]

        visited = set()

        while queue:

            cur\_node = queue.pop(0)

            for i, e in enumerate(matrix[cur\_node]):

                if i in visited:

                    continue

                if e > 0:

                    distance = e + distance\_matrix[v][cur\_node]

                    distance\_matrix[v][i] = distance

                    queue.append(i)

                    visited.add(i)

    for i in range(len(matrix)):

        \_bfs(i)

    return distance\_matrix

def find\_center(distance\_matrix: List[List[int]]) -> int:

    max\_distances = []

    for i, row in enumerate(distance\_matrix):

        if min(row) == 0:

            continue

        else:

            max\_distances.append((i, max(row)))

    max\_distances.sort(key=lambda tup: tup[1])

    return max\_distances[0]

def find\_chains(matrix: List[List[int]], v: int, distance: int) -> List[List[int]]:

    stack = []

    chains = []

    def \_dfs(v: int, \_distance: int):

        stack.append(v)

        for i, e in enumerate(matrix[v]):

            if i in stack:

                continue

            if e:

                if \_distance - e == 0:

                    chains.append([\*stack.copy(), i])

                elif \_distance - e >= 0:

                    \_dfs(i, \_distance - e)

        stack.remove(v)

    \_dfs(v, distance)

    return chains

if \_\_name\_\_ == "\_\_main\_\_":

    from pprint import pprint

    graph = Graph()

    graph.load("graph\_task27.json")

    distance\_matrix = BFS(graph.to\_adj\_matrix())

    print("Матрица расстояний:")

    pprint(distance\_matrix)

    center\_v, radius = find\_center(distance\_matrix)

    print(f"Центр графа: {center\_v}, радиус: {radius}")

    chains = find\_chains(graph.to\_adj\_matrix(), center\_v, radius)

    print("Цепи соответствующие радиусу:")

    print("\n".join("->".join(map(str, chain)) for chain in chains))

**Скриншот работы программы:**

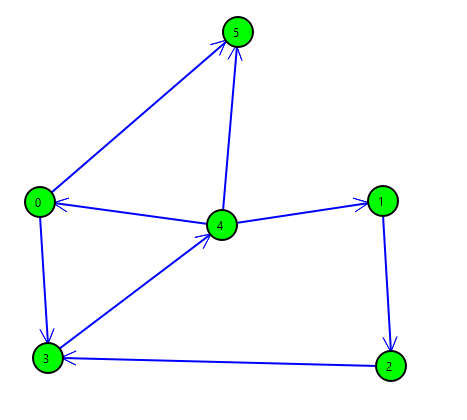


Рисунок 3. – Пример ориентированного не нагруженного графа.

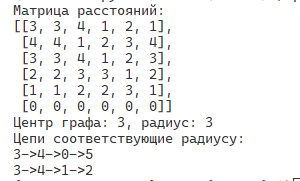
****

Рисунок 4. – Пример работы программы для графа из рисунка 3.

**Выводы.**

В результате выполнения данной работы была изучена структура граф, ее свойства и операции над ней. Был реализован базовый класс Graph который предоставляет возможность работы с элементам и графа, а так же базовые операции на ним. Так же был реализован алгоритм поиска в ширину для поиска центра графа, радиуса и цепей, соответствующих ему.

**Литература:**

1. Алгоритмы: построение и анализ. Кормен Т.Х., Лейзерсон Ч.И.

2. https://e-maxx.ru/algo/bfs

3. https://ru.wikipedia.org/wiki/Центр\_графа