МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ

Федеральное государственное бюджетное образовательное учреждение высшего образования
"МИРЭА - Российский технологический университет"

РТУ МИРЭА

Институт информационных технологий (ИТ)
Кафедра математического обеспечения и стандартизации информационных технологий (МОСИТ)

ОТЧЁТ ПО ПРАКТИЧЕСКОМУ ЗАДАНИЮ №5

по дисциплине «Структуры и алгоритмы обработки данных»

Тема: «Основные алгоритмы работы с графами.»

Отчет представлен к рассмотрению: Студентка группы ИНБО-01-20	«1» ноября 2021 г.	________________ (подпись)	Тульцова А.Д.
Преподаватель	«1» ноября 2021 г.	________________ (подпись)	Сорокин А.В.

Москва, 2021 г.

СОДЕРЖАНИЕ

Цель работы

Получение практических навыков по выполнению операций над структурой данных «граф».

Постановка задачи

Выполнить разработку программы управления графом, в соответствии с вариантом, на основе класса «Граф». Предусмотреть в качестве данных: количество вершин в графе, структура для хранения графа.

Вариант 11.

Представление графа в памяти	Задачи варианта
Матрица смежности	Ввод с клавиатуры графа (применение операции вставки ребра в граф). Определить, является ли граф связанным. Составить программу нахождения кратчайшего пути в графе от заданной вершины к другой заданной вершине методом «Дейкстры».

Дано:

Произвольный граф (ориентированный или неориентированный, связный или несвязный) с известным количеством вершин.

Результат.

Отображение графа в виде матрицы смежности.

Реализованные операции варианта.

Подход к решению.

1) Разработан класс графа, реализующий, согласно варианту, следующие методы: создание графа посредством применения операций вставки ребра в граф, вывод матрицы смежности графа, проверка графа на связность, нахождение величины кратчайшего пути и вывод кратчайшего пути от заданной вершины к другой заданной вершине методом «Дейкстры».
2) Разработан класс узла графа, содержащий информационную часть и поле индекса.
3) Разработан консольный пользовательский интерфейс для тестирования работоспособности программы.

4) Разработаны методы обработки графа:
1. Метод вставки ребра в граф – добавление ребра с заданным весом между двумя заданными вершинами;
2. Метод получения индекса узла – возврат индекса для заданного узла графа;
3. Метод проверки на сильную связность – нахождение количества достижимых вершин для заданного узла графа через обход «в глубину» для проверки на сильную связность;
4. Метод проверки на слабую связность – нахождение количества достижимых вершин для заданного узла графа через обход «в глубину» для проверки на слабую связность;
5. Метод проверки графа на связность – запуск метода проверки на сильную связность для неориентированного графа или общее исследование связности для ориентированного графа;
6. Метод «Дейкстры» – поиск величин кратчайших путей до каждой вершины для заданной вершины;
7. Метод восстановления кратчайшего пути – возврат кратчайшего маршрута из одной заданной вершины в другую;
8. Метод нахождения величины кратчайшего пути – возврат величины кратчайшего пути между двумя заданными вершинами графа, найденного с помощью метода алгоритма Дейкстры.

5) Разработаны методы приложения для тестирования:
1. Метод для тестирования вывода графа – организация ввода графа и вывода его матрицы смежности;
2. Метод для тестирования нахождения кратчайшего пути – организация нахождения кратчайшего пути от одной заданной вершины к другой с помощью алгоритма Дейкстры и вывода результатов работы алгоритмов в консоль;
3. Метод для тестирования проверки графа на связность – организация проверки графа на связность и вывода результата работы алгоритмов в консоль.

Алгоритмы операций на псевдокоде.

Метод вставки ребра в граф:

```
процедура connect(первая_вершина, вторая_вершина, вес := 1):
   первая_вершина := self.get_index_from_node(первая_вершина)
   вторая_вершина := self.get_index_from_node(вторая_вершина)
   матрица_смежности[первая_вершина][вторая_вершина] := вес
```

Метод получения индекса узла:

```
функция get_index_from_node(узел):
   если принадлежность(узел, int):
      возврат узел
   иначе:
      возврат узел.индекс
```

Метод проверки на сильную связность:

```
функция connectivity_DFS(целочисленный индекс_узла, visited := массив[]):
   visited.добавить_элемент(индекс_узла)
   для каждого i от 0 до длина(матрица_смежности) - 1:
      если матрица_смежности[индекс_узла][i] есть не None и i не в visited:
         connectivity_DFS(i, visited)
   возврат длина(visited)
```

Метод проверки на слабую связность:

```
функция weak_connectivity_DFS(целочисленный индекс_узла, visited :=
массив[]):
   visited.добавить_элемент(индекс_узла)
   для каждого i от 0 до длина(матрица_смежности) - 1:
      если (матрица_смежности[индекс_узла][i] есть не None или
матрица_смежности[i][индекс_узла] есть не None) и i не в visited:
         weak_connectivity_DFS(i, visited)
   возврат длина(visited)
```

Метод проверки графа на связность:

```
функция connectivity(булевский directed):
   если directed есть Ложь:
      для каждого индекс_узла от 0 до длина(матрица_смежности) – 1:
         visited := множество()
         если connectivity_DFS(индекс_узла, visited) <
длина(матрица_смежности):
            возврат массив[Ложь]
   иначе:
      для каждого индекс_узла от 0 до длина(матрица_смежности) – 1:
         visited := множество()
         если connectivity_DFS(индекс_узла, visited) <
длина(матрица_смежности):
            если weak_connectivity_DFS(индекс_узла, visited) <
длина(матрица_смежности):
               возврат массив[Ложь]
            иначе:
               возврат массив[Истина, Ложь]
   возврат массив[Истина, Истина]
```

Метод «Дейкстры»:

```
функция dijkstra(узел):
   узел := get_index_from_node(узел)
   из collections импортировать default_словарь
   граф := default_словарь(список)
   для каждого row от 0 до длина(матрица_смежности) - 1:
      для каждого col от 0 до длина(матрица_смежности) - 1:
         если матрица_смежности[row][col] есть не None и
матрица_смежности[row][col] != 0:
            граф[row] := граф[row] + [(матрица_смежности[row][col], col)]
   nodes_to_visit := массив[]
   nodes_to_visit.добавить_элемент_в_конец((0, узел))
   visited := множество()
   min_dist := {i: ∞ для каждого i от 0 до длина(матрица_смежности) - 1}
   min_dist[узел] := 0
   пока длина(nodes_to_visit) > 0:
      вес, текущий_узел := минимальный_элемент(nodes_to_visit)
      nodes_to_visit.удалить_элемент((вес, текущий_узел))
      если текущий узел в visited:
         принудительный_запуск_следующего_прохода_цикла
      visited.добавить_элемент(текущий_узел)
      для след_вес, след_узел в graph[текущий_узел]:
         если вес + след_вес < min_dist[след_узел] и след_узел не в visited:
            min_dist[след_узел] := вес + след_вес
```

```
            nodes_to_visit. добавить_элемент_в_конец((вес + след_вес,
след_узел))
   возврат min_dist
```

Метод восстановления кратчайшего пути:

```
функция path_restoring(узел1, узел2):
   visited := [None] * длина(матрица_смежности)
   узел1 := self.get_index_from_node(узел1)
   узел2 := self.get_index_from_node(узел2)
   visited[0] := узел2 + 1
   пред_индекс := 1
   вес := shortest_path(узел1, узел2)
   пока узел2 != узел1:
      для каждого i от 0 до длина(матрица_смежности) - 1:
         если матрица_смежности[i][узел2] есть не None и
матрица_смежности[i][узел2] != 0:
            temp := вес - матрица_смежности[i][узел2]
            если temp == shortest_path(узел1, i):
               вес := temp
               узел2 := i
               visited[пред_индекс] := i + 1
               пред_индекс := пред_индекс + 1
   пока None в visited:
      visited.удалить_элемент(None)
   возврат развернуть_массив(visited)
```

Метод нахождения величины кратчайшего пути:

```
функция shortest_path(узел1, узел2):
   узел2 := get_index_from_node(узел2)
   возврат dijkstra(узел1)[узел2]
```

Код программы.

Класс узла графа:

```python
class Node:  # Класс узла графа.
    def __init__(self, data, indexloc=None):
        self.data = data
        self.index = indexloc
```

Рисунок 1 – Класс узла графа.

Класс графа:

```python
class Graph:  # Класс, реализующий граф и некоторые операции с ним.

    @classmethod
    def create_from_nodes(self, nodes):
        return Graph(len(nodes), len(nodes), nodes)

    def __init__(self, row, col, nodes=None):
        # Установка матрицы смежности.
        self.adj_mat = [[None] * col for _ in range(row)]
        for i in range(len(self.adj_mat)):
            for j in range(len(self.adj_mat)):
                if i == j:
                    self.adj_mat[i][j] = 0
        self.nodes = nodes
        for i in range(len(self.nodes)):
            self.nodes[i].index = i

    # Вставка взвешенного ребра в граф.
    def connect(self, node1, node2, weight):
        node1, node2 = self.get_index_from_node(node1), self.get_index_from_node(node2)
        self.adj_mat[node1][node2] = weight

    # Получение индекса узла.
    @staticmethod
    def get_index_from_node(node):
        if isinstance(node, int):
            return node
        else:
            return node.index

    # Поиск количества достижимых вершин для узла через обход "в глубину".
    def connectivity_DFS(self, node_index, visited):
        visited.add(node_index)
        for i in range(len(self.adj_mat)):
            if self.adj_mat[node_index][i] is not None and i not in visited:
                self.connectivity_DFS(i, visited)
        return len(visited)

    # Особый поиск количества достижимых вершин для узла через обход "в глубину" для проверки графа на слабую связность.
    def weak_connectivity_DFS(self, node_index, visited):
        visited.add(node_index)
        for i in range(len(self.adj_mat)):
            if ((self.adj_mat[node_index][i] is not None or self.adj_mat[i][node_index] is not None)
                    and i not in visited):
                self.weak_connectivity_DFS(i, visited)
        return len(visited)

    # Проверка на связность.
    def connectivity(self, directed):
        if not directed:
            for node_index in range(len(self.adj_mat)):
                visited = set()
                if self.connectivity_DFS(node_index, visited) < len(self.adj_mat):
                    return [False]
        else:
            for node_index in range(len(self.adj_mat)):
                visited = set()
                if self.connectivity_DFS(node_index, visited) < len(self.adj_mat):
                    visited = set()
                    if self.weak_connectivity_DFS(node_index, visited) < len(self.adj_mat):
                        return [False]
                    else:
                        return [True, False]
        return [True, True]
```

Рисунок 2 – Класс графа.

```python
# Алгоритм Дейкстры.
def dijkstra(self, node):
    node = self.get_index_from_node(node)
    from collections import defaultdict
    graph = defaultdict(list)  # Инициализация графа словарём.
    # Заполнение графа по матрице смежности.
    for row in range(len(self.adj_mat)):
        for col in range(len(self.adj_mat)):
            if self.adj_mat[row][col] is not None and self.adj_mat[row][col] != 0:
                graph[row] += [(self.adj_mat[row][col], col)]
    nodes_to_visit = []  # Инициализация списка вершин для посещения.
    nodes_to_visit.append((0, node))  # Добавление стартовой вершины в список как первой вершины для посещения.
    visited = set()  # Множество для хранения посещённых вершин.
    min_dist = {i: float('inf') for i in range(len(self.adj_mat))}  # Заполнение расстояний до вершин.
    min_dist[node] = 0  # Заполнение расстояния до стартовой вершины.
    while len(nodes_to_visit):  # Пока nodes_to_visit не пустой:
        weight, current_node = min(nodes_to_visit)  # Выбор ближней вершины.
        nodes_to_visit.remove((weight, current_node))  # Удаление этой вершины из списка вершин для посещения.
        if current_node in visited:  # Если выбранная вершина уже посещена:
            continue  # Запустить следующий проход цикла, не выполняя оставшееся тело цикла.
        visited.add(current_node)  # Добавление выбранной вершины в список посещённых.
        # next_weight - вес связи из текущей вершины, next_node - прикреплённая вершина, в которую необходимо попасть.
        for next_weight, next_node in graph[current_node]:  # Проход по всем соединённым вершинам.
            # Проверка на оптимальность пути.
            if weight + next_weight < min_dist[next_node] and next_node not in visited:
                min_dist[next_node] = weight + next_weight  # Обновление расстояния.
                nodes_to_visit.append((weight + next_weight, next_node))  # Добавление вершины в список вершин для посещения.
    return min_dist  # Возврат множества из словарей {номер_узла: кратчайший путь до него от заданного узла}.

# Восстановление кратчайшего маршрута между node1 и node2.
def path_restoring(self, node1, node2):
    visited = [None] * len(self.adj_mat)  # Массив посещённых вершин.
    node1 = self.get_index_from_node(node1)
    node2 = self.get_index_from_node(node2)
    visited[0] = node2 + 1  # Начальный элемент - конечная вершина.
    pre = 1  # Индекс предыдущей вершины.
    weight = self.shortest_path(node1, node2)  # Вес конечной вершины.
    while node2 != node1:  # Пока не дошли до начальной вершины:
        for i in range(len(self.adj_mat)):  # Проход по всем вершинам.
            if self.adj_mat[i][node2] is not None and self.adj_mat[i][node2] != 0:  # При наличии связи:
                temp = weight - self.adj_mat[i][node2]  # Определение веса пути из предыдущей вершины.
                if temp == self.shortest_path(node1, i):  # Если вес совпал с рассчитанным, то из этой вершины был переход.
                    weight = temp
                    node2 = i
                    visited[pre] = i + 1
                    pre += 1
    while None in visited:
        visited.remove(None)
    return visited[::-1]

# Поиск кратчайшего маршрута между двумя заданными вершинами.
def shortest_path(self, node1, node2):
    node2 = self.get_index_from_node(node2)
    return self.dijkstra(node1)[node2]
```

Рисунок 3 – Класс графа (продолжение).

```python
# Приложение для ввода графа и вывода его матрицы смежности.
def app_adj_mat(self, directed=True):
    print("Построчно производите вставку рёбер в граф в формате:")
    print("номер начальной вершины, номер конечной вершины, вес связи")
    print("Для завершения добавления рёбер подайте на вход пустую строку.")
    print()
    while True:
        print("Ожидается ребро:")
        connection = list(map(int, input().split()))
        if not connection:
            break
        if directed is True:
            self.connect(connection[0] - 1, connection[1] - 1, connection[2])
            print("Добавлено направленное ребро от вершины", connection[0], "к вершине", connection[1], "с весом", connection[2], ".")
        else:
            self.connect(connection[0] - 1, connection[1] - 1, connection[2])
            self.connect(connection[1] - 1, connection[0] - 1, connection[2])
            print("Ребро с весом", connection[2], "успешно добавлено: вершины", connection[0], "и", connection[1], "соединены.")
    print("Вставка рёбер завершена.")
    print()
    print("Построена матрица смежности графа:")
    for row in self.adj_mat:
        print(row)

# Приложение для нахождения кратчайшего пути от заданной вершины к другой заданной вершине и его величины.
@staticmethod
def app_shortest_path():
    print("Рассмотрим алгоритм Дейкстры поиска кратчайшего пути.")
    start_node = int(input("Введите номер начальной вершины: ")) - 1
    end_node = int(input("Введите номер конечной вершины: ")) - 1
    print("Величина кратчайшего пути:", w_graph.shortest_path(start_node, end_node))
    if w_graph.shortest_path(start_node, end_node) != float('inf'):
        print("Кратчайший путь:", w_graph.path_restoring(start_node, end_node))
    else:
        print("Невозможно выполнить проход от заданной начальной вершины к заданной конечной вершине.")

# Приложение для проверки графа на связность.
@staticmethod
def app_connectivity(directed):
    print("Выполняется проверка графа на связность.")
    if w_graph.connectivity(directed)[0] is False:
        print("Результат проверки: граф не является связным.")
    elif w_graph.connectivity(directed)[1] is True:
        if directed is True:
            print("Результат проверки: граф является сильно связным.")
        else:
            print("Результат проверки: граф является связным.")
    else:
        print("Результат проверки: граф является слабо связным.")
```

Рисунок 4 – Класс графа (продолжение).

Основная функция для тестирования:

```python
# Главная функция.
if __name__ == '__main__':

    # Создание графа.
    node_list = []  # Список узлов.
    quantity = int(input("Введите количество вершин графа: "))
    for node in range(quantity):
        node_list.append(Node(str(node)))
    w_graph = Graph.create_from_nodes(node_list)

    # Вставка рёбер и вывод матрицы смежности.
    directed = bool(int(input("Если граф - ориентированный, то введите '1', иначе - введите '0': ")))
    w_graph.app_adj_mat(directed)

    print()
    # Нахождение кратчайшего пути и его величины методом "Дейкстры".
    w_graph.app_shortest_path()

    print()
    # Проверка графа на связность.
    w_graph.app_connectivity(directed)

    print()
    print("Тестирование завершено.")
```

Рисунок 5 – Основная функция.

Тестирование программы.

Тестирование программы на слабо связном ориентированном графе.

Для тестирования был выбран следующий граф (который является ориентированным):

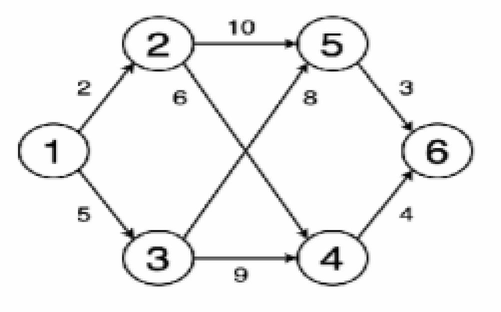

Рисунок 6 – Ориентированный граф для тестирования программы.

Было запущено тестирование: через консоль введено количество вершин графа и указан тип графа, после чего построчно производились операции вставки ребра в граф.

```
Введите количество вершин графа: 6
Если граф - ориентированный, то введите '1', иначе - введите '0':
Построчно производите вставку рёбер в граф в формате:
номер начальной вершины, номер конечной вершины, вес связи
Для завершения добавления рёбер подайте на вход пустую строку.

Ожидается ребро:
1 2 2
Добавлено направленное ребро от вершины 1 к вершине 2 с весом 2 .
Ожидается ребро:
1 3 5
Добавлено направленное ребро от вершины 1 к вершине 3 с весом 5 .
Ожидается ребро:
2 5 10
Добавлено направленное ребро от вершины 2 к вершине 5 с весом 10 .
Ожидается ребро:
3 5 8
Добавлено направленное ребро от вершины 3 к вершине 5 с весом 8 .
Ожидается ребро:
3 4 9
Добавлено направленное ребро от вершины 3 к вершине 4 с весом 9 .
Ожидается ребро:
2 4 6
Добавлено направленное ребро от вершины 2 к вершине 4 с весом 6 .
Ожидается ребро:
5 6 3
Добавлено направленное ребро от вершины 5 к вершине 6 с весом 3 .
Ожидается ребро:
4 6 4
Добавлено направленное ребро от вершины 4 к вершине 6 с весом 4 .
Ожидается ребро:

Вставка рёбер завершена.
```

Рисунок 7 – Ввод графа посредством операций вставки ребра в граф.

В результате работы программы был выведен граф в виде матрицы смежности, в которой каждая строка является набором длин исходящих путей в каждую вершину графа для соответствующей вершины графа. Если с какой-либо вершиной отсутствует непосредственная связь, то в ячейке хранится None. Также следует отметить, что путь от любой вершины до самой себя равен 0.

Результат работы программы:

```
Построена матрица смежности графа:
[0, 2, 5, None, None, None]
[None, 0, None, 6, 10, None]
[None, None, 0, 9, 8, None]
[None, None, None, 0, None, 4]
[None, None, None, None, 0, 3]
[None, None, None, None, None, 0]
```

Рисунок 8 – Вывод матрицы смежности графа.

Далее программа считывает две вершины графа для нахождения величины кратчайшего пути и вывода кратчайшего пути от одной заданной вершины к другой заданной вершине методом «Дейкстры».

Для тестирования были взяты вершины «1» и «6». По графическому изображению графа видно, что кратчайшим путём от вершины «1» к вершине «6» является путь: 1 – 2 – 4 – 6. Величина кратчайшего пути: 12.

Результат работы программы это подтверждает:

```
Рассмотрим алгоритм Дейкстры поиска кратчайшего пути.
Введите номер начальной вершины: 1
Введите номер конечной вершины: 6
Величина кратчайшего пути: 12
Кратчайший путь: [1, 2, 4, 6]
```

Рисунок 9 – Вывод кратчайшего пути и его величины методом «Дейкстры».

После этого программа выполнила проверку графа на связность.

В случае связности ориентированный граф может быть сильно связным или слабо связным.

Необходимым и достаточным условием сильной связности ориентированного графа является возможность прохода до любой вершины графа для каждой вершины графа.

По графическому изображению графа видно, что он не является сильно связным. Так, для вершины «6» не выполняется условие сильной связности, так как из неё невозможно осуществить проход в другие вершины графа.

С другой стороны, если представить рёбра данного графа неориентированными, то граф будет связным, поскольку для каждой вершины появится возможность прохода к любой другой вершине графа.

Таким образом, данный ориентированный граф является слабо связным.

Результат выполнения программы это подтверждает:

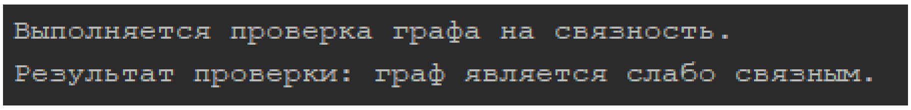

Рисунок 10 – Проверка графа на связность.

Тестирование программы на неориентированном графе.

Проверим теперь работу программы с неориентированным графом. Вообще говоря, работа алгоритмов по обработке неориентированного графа не отличается от работы алгоритмов по обработке ориентированного графа – для указания двусторонней связи можно дважды выполнить операцию вставки ребра в граф: сначала от первой вершины ко второй, затем от второй к первой.

Однако, благодаря методу приложения для ввода графа, стало возможным не только упростить ввод неориентированного графа, но и инкапсулировать его в удобный интерфейс.

Рассмотрим в качестве примера следующий неориентированный граф:

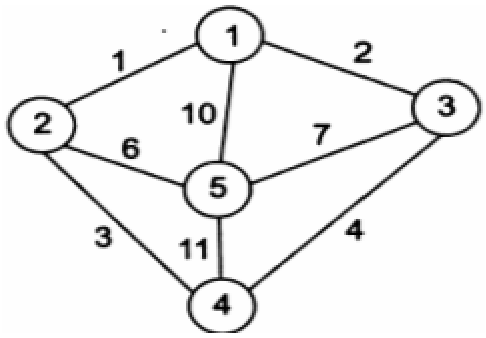

Рисунок 11 – Неориентированный граф для тестирования программы.

Было запущено тестирование: через консоль введено количество вершин графа и указан тип графа, после чего построчно производились операции вставки ребра в граф, при этом для создания ребра было достаточно указывать две вершины в любом порядке.

```
Введите количество вершин графа: 5
Если граф - ориентированный, то введите '1', иначе - введите '0': 0
Построчно производите вставку рёбер в граф в формате:
номер начальной вершины, номер конечной вершины, вес связи
Для завершения добавления рёбер подайте на вход пустую строку.

Ожидается ребро:
1 2 1
Ребро с весом 1 успешно добавлено: вершины 1 и 2 соединены.
Ожидается ребро:
1 3 2
Ребро с весом 2 успешно добавлено: вершины 1 и 3 соединены.
Ожидается ребро:
1 5 10
Ребро с весом 10 успешно добавлено: вершины 1 и 5 соединены.
Ожидается ребро:
2 5 6
Ребро с весом 6 успешно добавлено: вершины 2 и 5 соединены.
Ожидается ребро:
2 4 3
Ребро с весом 3 успешно добавлено: вершины 2 и 4 соединены.
Ожидается ребро:
3 5 7
Ребро с весом 7 успешно добавлено: вершины 3 и 5 соединены.
Ожидается ребро:
3 4 4
Ребро с весом 4 успешно добавлено: вершины 3 и 4 соединены.
Ожидается ребро:
4 5 11
Ребро с весом 11 успешно добавлено: вершины 4 и 5 соединены.
Ожидается ребро:

Вставка рёбер завершена.
```

Рисунок 12 – Ввод графа посредством операций вставки ребра в граф.

В результате работы программы была построена матрица смежности графа:

```
Построена матрица смежности графа:
[0, 1, 2, None, 10]
[1, 0, None, 3, 6]
[2, None, 0, 4, 7]
[None, 3, 4, 0, 11]
[10, 6, 7, 11, 0]
```

Рисунок 13 – Вывод матрицы смежности графа.

Далее программа считывает две вершины графа для нахождения величины кратчайшего пути и вывода кратчайшего пути от одной заданной вершины к другой заданной вершине методом «Дейкстры».

Для тестирования были взяты вершины «1» и «4». По графическому изображению графа видно, что кратчайшим путём от вершины «1» к вершине «4» является путь: 1 – 2 – 4. Величина кратчайшего пути: 4.

Результат работы программы это подтверждает:

```
Рассмотрим алгоритм Дейкстры поиска кратчайшего пути.
Введите номер начальной вершины: 1
Введите номер конечной вершины: 4
Величина кратчайшего пути: 4
Кратчайший путь: [1, 2, 4]
```

Рисунок 14 – Вывод кратчайшего пути и его величины методом «Дейкстры».

После этого программа выполнила проверку графа на связность.

По графическому изображению графа видно, что он является связным, поскольку для любой вершины данного графа существует хотя бы один проход в каждую вершину графа.

Результат выполнения программы подтверждает связность графа:

```
Выполняется проверка графа на связность.
Результат проверки: граф является связным.
```

Рисунок 15 – Проверка графа на сильную связность.

Тестирование программы на сильно связном ориентированном графе.

Теперь проведём тестирование для сильно связного ориентированного графа:

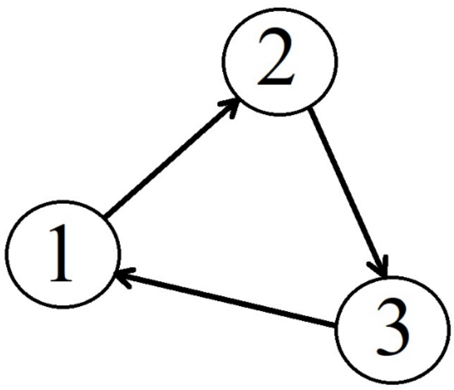

Рисунок 16 – Пример сильно связного ориентированного графа.

Осуществим ввод данных:

```
Введите количество вершин графа: 3
Если граф – ориентированный, то введите '1', иначе – введите '0': 1
Построчно производите вставку рёбер в граф в формате:
номер начальной вершины, номер конечной вершины, вес связи
Для завершения добавления рёбер подайте на вход пустую строку.

Ожидается ребро:
1 2 1
Добавлено направленное ребро от вершины 1 к вершине 2 с весом 1 .
Ожидается ребро:
2 3 1
Добавлено направленное ребро от вершины 2 к вершине 3 с весом 1 .
Ожидается ребро:
3 1 1
Добавлено направленное ребро от вершины 3 к вершине 1 с весом 1 .
Ожидается ребро:

Вставка рёбер завершена.
```

Рисунок 17 – Ввод графа посредством операций вставки ребра в граф.

Получаем матрицу смежности графа:

```
Построена матрица смежности графа:
[0, 1, None]
[None, 0, 1]
[1, None, 0]
```

Рисунок 18 – Вывод матрицы смежности графа.

Найдём кратчайший путь от вершины «2» к вершине «1» (веса рёбер примем равными единице): 2 – 3 – 1. Величина кратчайшего пути: 2.

Результат работы программы это подтверждает:

```
Рассмотрим алгоритм Дейкстры поиска кратчайшего пути.
Введите номер начальной вершины: 2
Введите номер конечной вершины: 1
Величина кратчайшего пути: 2
Кратчайший путь: [2, 3, 1]
```

Рисунок 19 – Вывод кратчайшего пути и его величины методом «Дейкстры».

Наконец, результат выполнения программы подтверждает сильную связность графа:

```
Выполняется проверка графа на связность.
Результат проверки: граф является сильно связным.
```

Рисунок 20 – Проверка графа на связность.

Тестирование программы на несвязном графе.

Пусть дан несвязный граф:

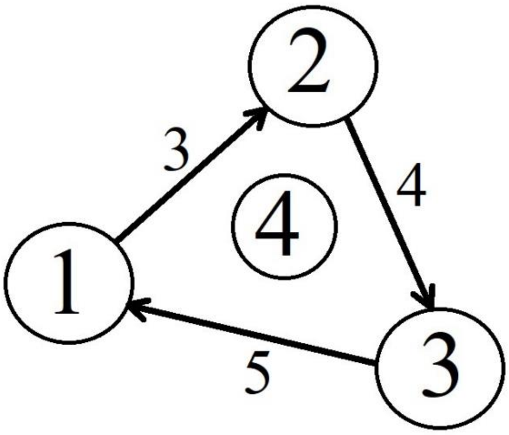

Рисунок 21 – Пример несвязного графа.

Произведём ввод данных:

```
Введите количество вершин графа: 4
Если граф - ориентированный, то введите '1', иначе - введите '0':
Построчно производите вставку рёбер в граф в формате:
номер начальной вершины, номер конечной вершины, вес связи
Для завершения добавления рёбер подайте на вход пустую строку.

Ожидается ребро:
1 2 3
Добавлено направленное ребро от вершины 1 к вершине 2 с весом 3 .
Ожидается ребро:
2 3 4
Добавлено направленное ребро от вершины 2 к вершине 3 с весом 4 .
Ожидается ребро:
3 1 5
Добавлено направленное ребро от вершины 3 к вершине 1 с весом 5 .
Ожидается ребро:

Вставка рёбер завершена.
```

Рисунок 22 – Ввод графа посредством операций вставки ребра в граф.

Произведён вывод матрицы смежности графа:

```
Построена матрица смежности графа:
[0, 3, None, None]
[None, 0, 4, None]
[5, None, 0, None]
[None, None, None, 0]
```

Рисунок 23 – Вывод матрицы смежности графа.

Для вершин «1», «2», «3» не существует путей к вершине «4», поэтому при попытки найти кратчайший путь, например, от вершины «1» к вершине «4» программа выводит исключение:

```
Рассмотрим алгоритм Дейкстры поиска кратчайшего пути.
Введите номер начальной вершины: 1
Введите номер конечной вершины: 4
Величина кратчайшего пути: inf
Невозможно выполнить проход от заданной начальной вершины к заданной конечной вершине.
```

Рисунок 24 – Вывод кратчайшего пути и его величины методом «Дейкстры».

Результат выполнения программы подтверждает несвязность графа:

```
Выполняется проверка графа на связность.
Результат проверки: граф не является связным.

Тестирование завершено.
```

Рисунок 25 – Проверка графа на связность.

Таким образом, тестирование прошло успешно: все поставленные задачи были выполнены.

Вывод

В ходе работы были приобретены умения и навыки разработки и реализации операций над структурой данных «граф».

Список информационных источников

1. Лекции по дисциплине «Структуры и алгоритмы обработки данных» / Л. А. Скворцова, МИРЭА – Российский технологический университет, 2021.