Дискретная математика для программистов.
Лабораторная № 6. (ПМ-2) (Весна 2024)
«Основные понятия теории графов»
ВАРИАНТ 70

ПОРЯДОК ВЫПОЛНЕНИЯ РАБОТЫ

(0,1,13); (0,2,13); (0,3,18); (0,4,18); (0,5,16); (1,3,12); (1,4,12); (1,6,17); (3,5,15); (3,6,19); (4,3,13); (5,2,12); (5,3,12); (5,7,15); (6,4,19); (6,5,15); (6,7,19)

Дан взвешенный граф списком ребер. Построить матрицу смежности.

Написать программу, реализующую перевод матрицы смежности в матрицу инцидентности.

Код для ввода графа с клавиатуры:

```
1 usage
def input_graph():
    graph = input("Введите граф: ")  # получаем граф с клавиатуры

    # превращаем строку в список ребер
    graph = graph.replace("), ", ")|")  # замена запятых для разделения ребер
    edge_list = graph.split("|")  # ребра разделенные |

    graph = list()  # граф в виде ребер
    vertices = set()  # множество уникальных вершин

    for edge_str in edge_list:
        edge_str = edge_str.strip("()")  # удаление скобок вокруг ребра
        u, v, weight = map(int, edge_str.split(","))  # разбиваем ребра на вершины и вес
        graph.append((u, v, weight))
        vertices.add(u)
        vertices.add(v)

    return graph, len(vertices)

```

Построение матрицы смежности:

```
1 usage
def adjacency(graph, num_vertices):
    # создание пустой матрицы смежности
    adjacency_matrix = [[0] * num_vertices for _ in range(num_vertices)]

    # заполнение матрицы смежности
    for edge in graph:
        u = edge[0]
        v = edge[1]
        weight = edge[2]
        adjacency_matrix[u][v] = weight  # обновляем значение в матрице смежности

    return adjacency_matrix

```

Построение матрицы инцидентности:

```python
1 usage
def incidence(graph, num_vertices):
    # создание пустой матрицы инцидентности
    num_edges = len(graph)
    incidence_matrix = [[0] * num_edges for _ in range(num_vertices)]

    # заполнение матрицы инцидентности
    for i, edge in enumerate(graph):
        u = edge[0]
        v = edge[1]
        incidence_matrix[u][i] = 1
        incidence_matrix[v][i] = -1

    return incidence_matrix


```

Вывод результатов:

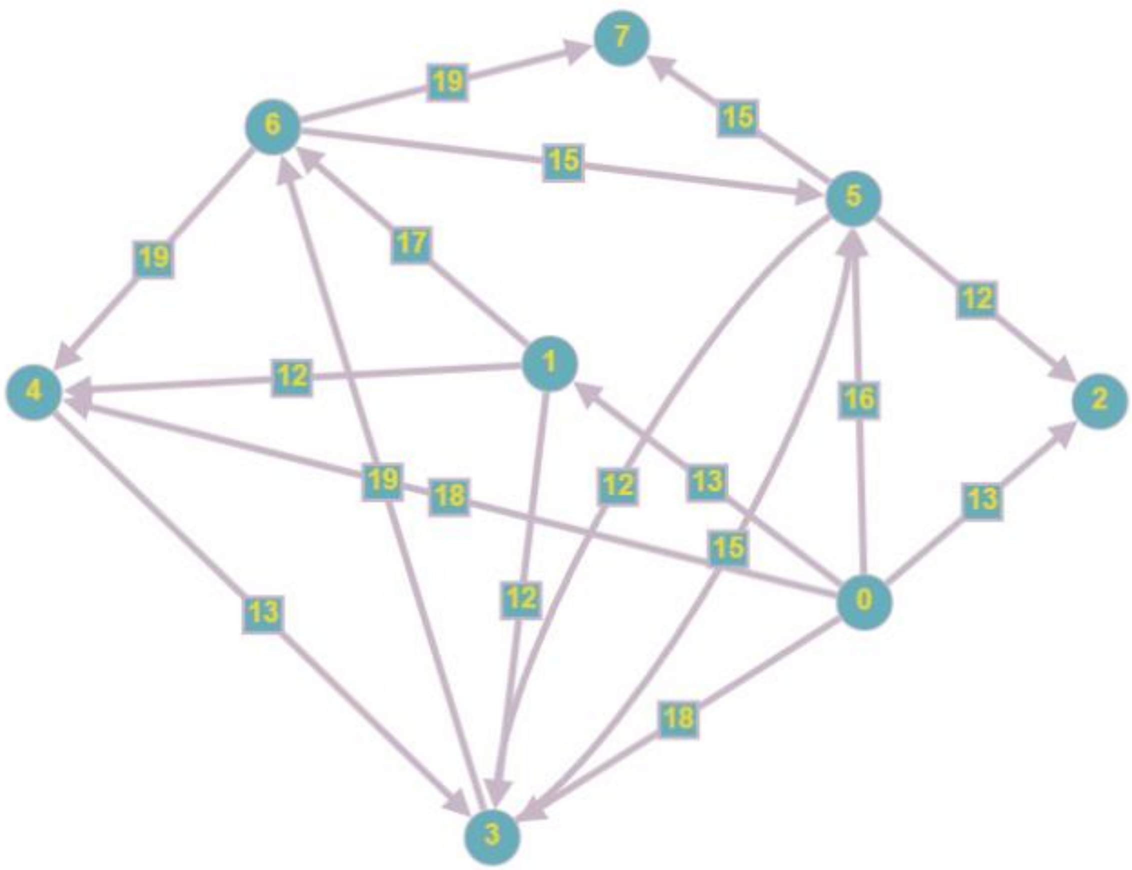
```python
2 usages
def print_matrix(matrix, text):
    print()
    print(text)
    for row in matrix:
        print(" ".join("{:2}".format(cell) for cell in row))

```

```python

# получение графа
graph, num_vertices = input_graph()

# матрица смежности
adjacency_matrix = adjacency(graph, num_vertices)
print_matrix(adjacency_matrix, "Матрица смежности:")

# матрица инцидентности
incidence_matrix = incidence(graph, num_vertices)
print_matrix(incidence_matrix, "Матрица инцидентности:")
```

Результат работы программы:

```
Матрица смежности:
 0 13 13 18 18 16  0  0
 0  0  0 12 12  0 17  0
 0  0  0  0  0  0  0  0
 0  0  0  0  0 15 19  0
 0  0  0 13  0  0  0  0
 0  0 12 12  0  0  0 15
 0  0  0  0 19 15  0 19
 0  0  0  0  0  0  0  0

Матрица инцидентности:
 1  1  1  1  1  0  0  0  0  0  0  0  0  0  0  0  0
-1  0  0  0  0  1  1  1  0  0  0  0  0  0  0  0  0
 0 -1  0  0  0  0  0  0  0  0  0 -1  0  0  0  0  0
 0  0 -1  0  0 -1  0  0  1  1 -1  0 -1  0  0  0  0
 0  0  0 -1  0  0 -1  0  0  0  1  0  0  0 -1  0  0
 0  0  0  0 -1  0  0  0 -1  0  0  1  1  1  0 -1  0
 0  0  0  0  0  0  0 -1  0 -1  0  0  0  0  1  1  1
 0  0  0  0  0  0  0  0  0  0  0  0  0 -1  0  0 -1
```

Имеем следующий граф:



Выяснить, является ли граф Эйлеровым.
Предусмотреть консольный ввод исходных данных и вывод результатов работы программы на экран.

Чтобы ориентированный граф, был Эйлеровым:

1) Количество ребер, входящих и выходящих из каждой вершины, должно быть одинаковым

```python
1 usage
def is_eulerian(incidence_matrix, adjacency_matrix):
    # находим количество вершин
    num_vertices = len(adjacency_matrix)

    # проверка на четность ребер для каждой вершины
    n = len(incidence_matrix[0])
    m = len(incidence_matrix)
    in_degrees = [0] * n
    out_degrees = [0] * n

    # подсчет входящих и исходящих степеней для каждой вершины
    for i in range(m):
        for j in range(n):
            if incidence_matrix[i][j] == 1:
                out_degrees[i] += 1
                in_degrees[j] += 1

    # проверка на четность ребер для каждой вершины
    for vertex in range(num_vertices):
        if in_degrees[vertex] != out_degrees[vertex]:
            return None, f"количество входящих ребер для вершины {vertex} равно {in_degrees[vertex]}, " \
                         f"а исходящих {out_degrees[vertex]}"
```

2) Он должен быть сильно связным

```python
    # проверка на сильную связность графа

    # проверка связности для исходного графа
    visited = [False] * num_vertices
    dfs(0, adjacency_matrix, visited)  # начинаем обход с первой вершины

    if not all(visited):
        return False, f"граф - несвязный"

    # получение транспонированной матрицы
    transpose_matrix = get_transpose(adjacency_matrix)

    # проверка связности для транспонированного графа
    visited = [False] * num_vertices
    dfs(0, transpose_matrix, visited)  # Начинаем обход с первой вершины

    # все вершины были посещены
    if all(visited):
        return True
    else:
        return False, f"граф - несвязный"
```

Проверяем связность исходного графа с помощью обхода в глубину. Если граф оказался связным, транспонируем его и снова проверяем на связность тем же алгоритмом. Если и транспонированный граф оказался связным, значит наш граф является сильно связным.

Транспонирование графа:

```python
def get_transpose(adjacency_matrix):
    num_vertices = len(adjacency_matrix)
    transpose_matrix = [[0] * num_vertices for _ in range(num_vertices)]

    for i in range(num_vertices):
        for j in range(num_vertices):
            transpose_matrix[j][i] = adjacency_matrix[i][j]

    return transpose_matrix
```

Обход в глубину:

```python
3 usages
def dfs(vertex, adjacency_matrix, visited):
    visited[vertex] = True
    for neighbor in range(len(adjacency_matrix)):
        if adjacency_matrix[vertex][neighbor] == 1 and not visited[neighbor]:
            dfs(neighbor, adjacency_matrix, visited)

```

Результат работы программы:

```python
# вывод результата проверки на Эйлеров граф
eulerian_cycle, error = is_eulerian(incidence_matrix, adjacency_matrix)
print()
if eulerian_cycle:
    print("Граф является Эйлеровым")
else:
    print("Граф не является Эйлеровым, потому что", error)

```

```
Граф не является Эйлеровым, потому что количество входящих ребер для вершины 0 равно 1, а исходящих 5
```

Считая граф неориентированным, найти кратчайшие пути между всеми вершинами. Предусмотреть консольный ввод исходных данных и вывод результатов работы программы на экран.

Алгоритм Дейкстры для поиска кратчайшего пути:

```
1 usage
def dijkstra(N, S, graph):
    valid = [True] * N # доступность вершин
    weight = [math.inf] * N # длины путей
    weight[S] = 0
    for k in range(N):
        min_weight = math.inf # текущая минимальная длина пути
        ID_min_weight = -1 # индекс вершины с минимальной длиной пути
        for k in range(N):
            if valid[k] and weight[k] < min_weight:
                min_weight = weight[k]
                ID_min_weight = k
        for z in range(N):
            if weight[ID_min_weight] + graph[ID_min_weight][z] < weight[z]:
                weight[z] = weight[ID_min_weight] + graph[ID_min_weight][z]
    valid[ID_min_weight] = False
    return weight
```

Продемонстрируем поиск для вершины 0:

```
# вывод кратчайших путей из одной вершины в другую
paths = dijkstra(num_vertices, 0, adjacency_matrix)
print()
print("Кратчайшие пути из выбранной вершины:", *paths)
```

```
Кратчайшие пути из выбранной вершины: 0 13 13 18 18 16 0 0
```