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

Институт комплексной безопасности и специального приборостроения

Кафедра КБ-3 «Управление и моделирование систем»

**ОТЧЕТ   
о выполнении лабораторной работы №2**

**«Реализация алгоритмов на графах»**

**по дисциплине   
«Программная реализация нелинейных структур»**

**Вариант № 44**

Выполнил: студент 2 курса

Группы **\_\_\_\_\_\_**

шифр \_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_  
*(фио студента)*

Проверил:

*\_\_\_\_\_\_\_\_\_\_\_\_\_\_*

Москва 2022 г.

**Вариант № 44.**

**Алгоритм:** **В заданном неориентированном графе вывести все вершины – точки сочленения.**

**Способ представления графа: Матрица инцидентности**

**Теория о Графах.**

Граф — это математический объект, который состоит из точек и линий, которые их соединяют. Точки называют вершинами графа, а линии — ребрами.

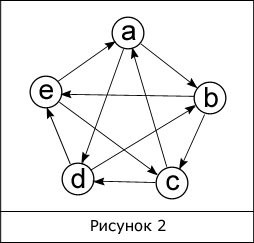
**Вот некоторые важные обозначения, используемые в теории графов:**

* G = (V, E), здесь G – граф, V – его вершины, а E – ребра;
* |V| – порядок (число вершин);
* |E| – размер графа (число рёбер).

В нашем случае (рис. 1) |V|=5, |E|=10;

Когда из любой вершины доступна любая другая вершина, то такой граф называется **неориентированным** связным графом (рис. 1). Если же граф связный, но это условие не выполняется, тогда такой граф называется **ориентированным** или орграфом (рис. 2).

В ориентированных и неориентированных графах имеется понятие степени вершины. **Степень вершины** – это количество ребер, соединяющих ее с другими вершинами. Сумма всех степеней графа равна удвоенному количеству всех его ребер. Для рисунка 2 сумма всех степеней равна 20.

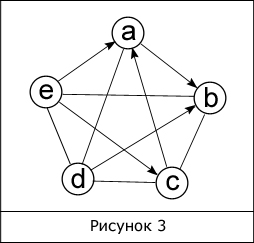
[](https://kvodo.ru/wp-content/uploads/orgraph.jpg)

В орграфе, в отличие от неориентированного графа, имеется возможность двигаться из вершины h в вершину s без промежуточных вершин, лишь тогда, когда ребро выходит из h и входит в s, но не наоборот.

**Ориентированные графы имеют следующую форму записи:**

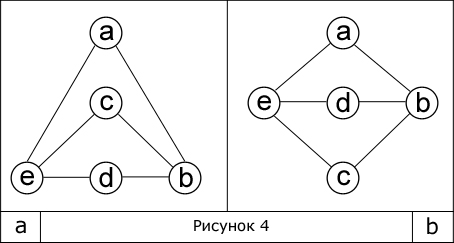
*G = (V, A), где V – вершины, A – направленные ребра.*

Третий тип графов – **смешанные** графы (рис. 3). Они имеют как направленные ребра, так и ненаправленные. Формально смешанный граф записывается так: G = (V, E, A), где каждая из букв в скобках обозначает тоже, что ей приписывалось ранее.

[](https://kvodo.ru/wp-content/uploads/mix_graph.jpg)

В графе на рисунке 3 одни дуги направленные [(e, a), (e, c), (a, b), (c, a), (d, b)], другие – ненаправленные [(e, d), (e, b), (d, c) …].

Два или более графов на первый взгляд могут показаться разными по своей структуре, что возникает вследствие различного их изображения. Но это не всегда так. Возьмем два графа (рис. 4).

[](https://kvodo.ru/wp-content/uploads/izomorph_graph.jpg)

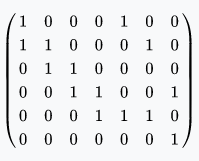
Они эквивалентны друг другу, ведь не изменяя структуру одного графа можно построить другой. Такие графы называются **изоморфными**, т. е. обладающими тем свойством, что какая-либо вершина с определенным числом ребер в одном графе имеет тождественную вершину в другом. На рисунке 4 изображены два изоморфных графа.

Когда каждому ребру графа поставлено в соответствие некоторое значение, называемое весом ребра, тогда такой граф **взвешенный**. В разных задачах в качестве веса могут выступать различные виды измерений, например длины, цены маршруты и т. п. В графическом представлении графа весовые значения указываются, как правило, рядом с ребрами.

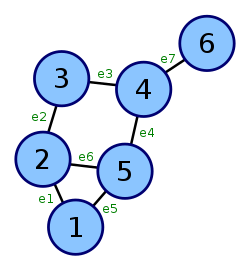
В любом из рассмотренных нами графов имеется возможность выделить путь и, причем не один. **Путь** – это последовательность вершин, каждая из которых соединена с последующей посредством ребра. Если первая и последняя вершины совпадают, то такой путь называется циклом. Длина пути определяется количеством составляющих его ребер. Например, на рисунке 4.а путем служит последовательность [(e), (a), (b), (c)]. Этот путь является подграфом, так как к нему применимо определение последнего, а именно: граф G’ = (V’, E’) является подграфом графа G = (V, E), только тогда, когда V’ и E’ принадлежат V, E.

**Матрица инцидентности** — одна из форм представления графа, в которой указываются связи между инцидентными элементами графа (ребро (дуга) и вершина). Столбцы матрицы соответствуют ребрам, строки — вершинам. Ненулевое значение в ячейке матрицы указывает связь между вершиной и ребром (их инцидентность).

В случае ориентированного графа каждой дуге <x, y> ставится в соответствующем столбце: «1» в строке вершины x и «-1» в строке вершины y; если связи между вершиной и ребром нет, то в соответствующую ячейку ставится «0».



Его матрица инцидентности



Пример графа

**Листинг программы**

import json  
from typing import Dict, List, Optional, Tuple  
  
T\_ADJ\_MATRIX = List[List[int]]  
T\_PATH = List[int]  
  
  
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) -> T\_ADJ\_MATRIX:  
 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  
  
  
T\_BRIDGE = Tuple[int, int]  
  
  
def task(graph: Graph) -> None:  
 def find\_bridges(adj\_matrix: T\_ADJ\_MATRIX) -> List[T\_BRIDGE]:  
 n = len(adj\_matrix)  
 bridges = []  
 visited = [False] \* n  
 disc = [float("inf")] \* n  
 low = [float("inf")] \* n  
 parent = [-1] \* n  
  
 def dfs(u: int, time: int) -> None:  
 visited[u] = True  
 disc[u] = low[u] = time + 1  
  
 for v in range(n):  
 if adj\_matrix[u][v] and not visited[v]:  
 parent[v] = u  
 dfs(v, time + 1)  
 low[u] = min(low[u], low[v])  
 if low[v] > disc[u]:  
 bridges.append((u, v))  
  
 elif adj\_matrix[u][v] and v != parent[u]:  
 low[u] = min(low[u], disc[v])  
  
 for i in range(n):  
 if not visited[i]:  
 dfs(i, 0)  
  
 return bridges  
  
 adj\_matrix = graph.to\_adj\_matrix()  
 bridges = find\_bridges(adj\_matrix)  
  
 adjunction\_vertices = set()  
 adjunction\_vertices.update([v for tup in bridges for v in tup])  
  
 print("Точки сочленения графа: " + ", ".join([str(v) for v in adjunction\_vertices]))  
  
  
if \_\_name\_\_ == "\_\_main\_\_":  
 import sys  
  
 def print\_usage():  
 print(f"Использование: {sys.argv[0]} [ПУТЬ\_ДО\_ФАЙЛА\_С\_ГРАФОМ] [РЕЖИМ]")  
 print(  
 "Режимы:\n"  
 "\texample - загружает граф и выводит его компоненты и матрицу смежности\n"  
 "\ttask - решает задачу из вариант"  
 )  
  
 if len(sys.argv) 3:  
 print\_usage()  
 exit(-1)  
  
 filename, mode, \*\_ = sys.argv[1:]  
  
 graph = Graph()  
 try:  
 graph.load(filename)  
 except Exception as e:  
 print(e)  
 exit(-1)  
  
 match mode:  
 case "example":  
 print("Ребра графа:\n" + "\n".join(str(e) for e in graph.edges))  
 print("Вершины графа:\n" + "\n".join(str(v) for v in graph.vertices))  
 print("Матрица смежности:\n" + "\n".join(str(row) for row in graph.to\_adj\_matrix()))  
 case "task":  
 task(graph)  
  
 case \_:  
 print\_usage()  
 exit(-1)

**Скриншоты работы программы.**

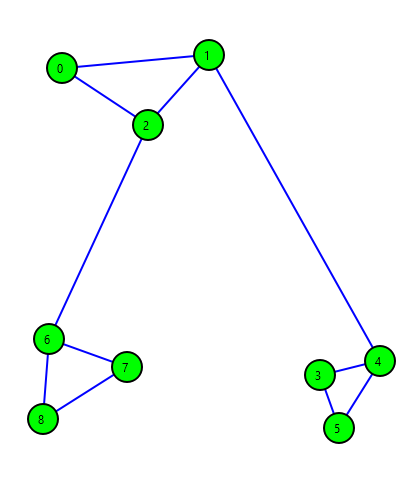


Рисунок 3. - Пример ориентированного нагруженного графа с циклами

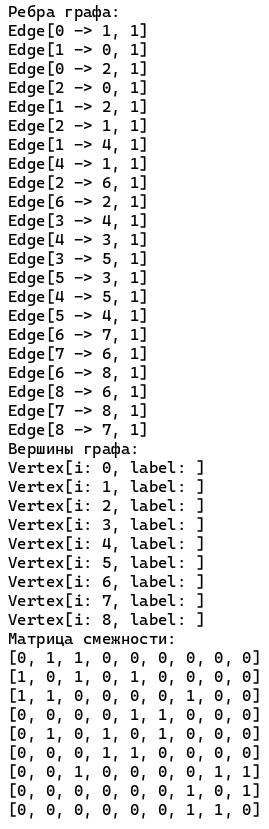


Рисунок 4. - Пример работы программы в режиме example.



Рисунок 5. - Пример работы программы в режиме task.

**Вывод:**

В результате выполнения данной работы была реализована структура Graph предоставляющая удобный интерфейс для работы с графами и их компонентами. Для этого класса был реализован весь необходимый функционал по заданию.

Также был реализован алгоритм для нахождения всех точек сочленения графа.

**Литература:**

1. Буркатовския Ю. Б. Теория графов. — Томск: Издательство Томского политехнического университета, 2014. — Т. 1. — 200 с

2. Дистель Р. Теория графов. — Новосибирск: Издательство Института математики им. С. Л. Соболева СО РАН, 2002. — 336 с. — ISBN 5-86134-101-Х.

3. Касьянов В. Н., Евстигнеев В.А. Графы в программировании: обработка, визуализация и применение. — СПб.: БХВ-Петербург, 2003. — С. 1104. — ISBN 5-94157-184-4.