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

Институт кибербезопасности и цифровых технологий

Кафедра КБ-2 «Институт кибербезопасности и цифровых технологий»

**ОТЧЕТ   
о выполнении домашней работе**

**«Реализация сортировки линейных структур данных»**

**по дисциплине   
«Технологии и методы программирования»**

**Вариант № 78**

Выполнил: студент 2 курса

группы БИСО-01-21

Русаков А.М.

шифр ИТ6-4387086

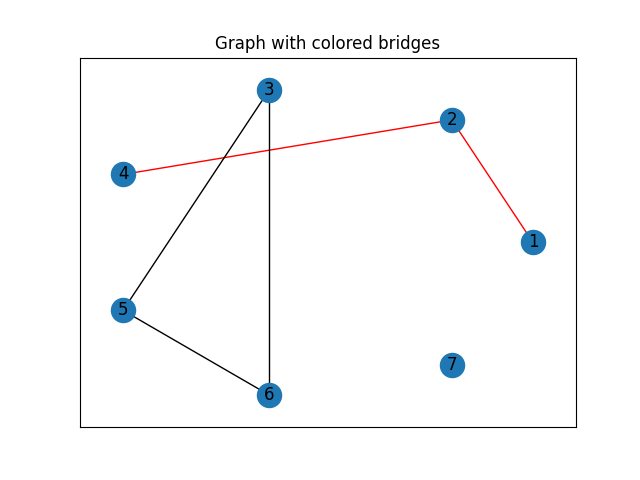
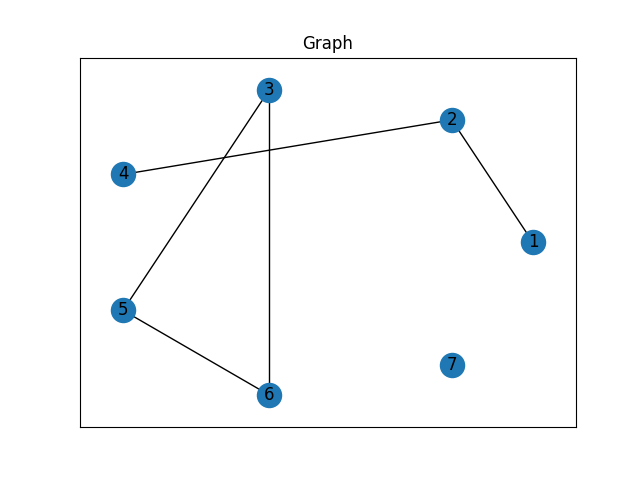
Москва 2022 г.

**Вариант № 78.**

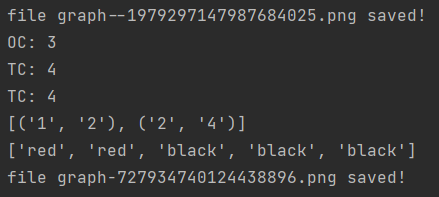
**Алгоритм: Поиск мостов графа**

**Способ представления графа: Матрица смежности**

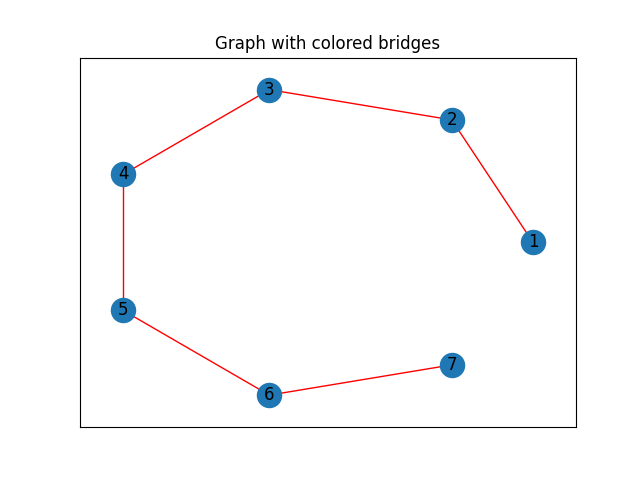
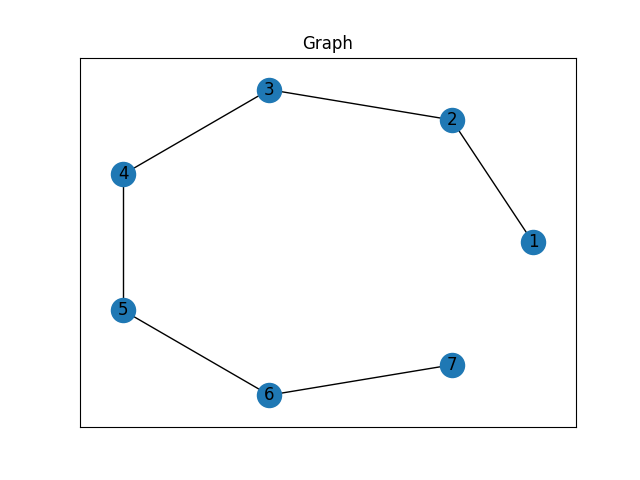
**Пример графа и его мостов (отмечены красным) №1:**

****

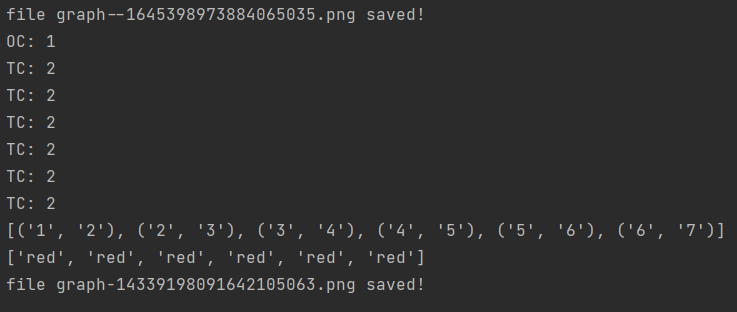
**Пример работы программы:**



**Пример графа и его мостов (отмечены красным) №2:**

****

**Пример работы программы:**



**ВЫВОДЫ**

Был реализован Абстрактный Тип Данных «Граф», с помощью которого было выполнено задание варианта: поиск мостов графа.

**Приложение 1 Листинг кода**

## ADTGraph.py

import networkx as nx

import matplotlib.pyplot as plt

class Node:

def \_\_init\_\_(self, mark: str = '', idx: int = 0):

self.mark = mark

self.idx = idx

self.visited = False

self.component = 0

def \_\_repr\_\_(self):

return f'{self.mark}: {self.component=}'

class Edge:

def \_\_init\_\_(self, start: Node, end: Node, weight=0):

self.start = start

self.end = end

self.weight = weight

class ADTGraph:

MAX\_VERTEX\_COUNT = 20

def \_\_init\_\_(self):

self.g = nx.DiGraph()

self.nodes: dict[str, Node] = {}

self.edges: dict[tuple[Node, Node], Edge] = {}

self.A: list[list[int]] = [[0 for \_ in range(self.MAX\_VERTEX\_COUNT)] for \_ in range(self.MAX\_VERTEX\_COUNT)]

self.idx = 0

def \_\_get\_node\_by\_index(self, idx: int):

for key, value in self.nodes.items():

if value.idx == idx:

return key

return None

# ADD\_V(<имя>) - добавить УЗЕЛ

def add\_v(self, name: str):

n = Node(name, self.idx)

self.nodes[name] = n

self.idx += 1

self.g.add\_node(name)

# ADD\_Е(v, w) - добавить ДУГУ

def add\_e(self, start: str, end: str):

n\_start = self.nodes[start]

n\_end = self.nodes[end]

edge = Edge(n\_start, n\_end)

self.edges[(n\_start, n\_end)] = edge

self.A[n\_start.idx][n\_end.idx] = 1

self.g.add\_edge(start, end)

# FIRST(v) - возвращает индекс первой вершины, смежной с вершиной v.

# Если вершина v не имеет смежных вершин, то возвращается "нулевая" вершина ?.

def first(self, v: str):

n = self.nodes[v]

for index, value in enumerate(self.A[n.idx]):

if value == 1:

return self.\_\_get\_node\_by\_index(index)

return None

# NEXT(v, i)- возвращает индекс вершины, смежной с вершиной v, следующий за индексом i.

# Если i — это индекс последней вершины, смежной с вершиной v, то возвращается ?.

def next(self, v: str, i: int):

n = self.nodes[v]

for v\_idx in range(i + 1, self.idx + 1):

if self.A[n.idx][v\_idx] == 1:

return self.\_\_get\_node\_by\_index(v\_idx)

return None

# VERTEX(v, i) - возвращает вершину с индексом i из множества вершин, смежных с v.

def vertex(self, v: str, i: int):

n = self.nodes[v]

for index, value in enumerate(self.A[n.idx]):

if value == 1 and i == 1:

return self.\_\_get\_node\_by\_index(index)

i -= value

return None

# DEL\_Е(v, w) – удалить ДУГУ

def del\_e(self, v: str, w: str):

n1 = self.nodes[v]

n2 = self.nodes[w]

self.A[n1.idx][n2.idx] = 0

self.A[n2.idx][n1.idx] = 0

del self.edges[(n1, n2)]

self.g.remove\_edge(n1.mark, n2.mark)

def dfs1(self, v: str, order: list[str]):

# count = 0

# components[v] = comp\_num

# while n := self.next(v, count):

# if n not in components:

# self.dfs(n, comp\_num, components)

# count += 1

n = self.nodes[v]

n.visited = True

count = 0

while u := self.next(v, count):

un = self.nodes[u]

if not un.visited:

self.dfs1(u, order)

count += 1

order.append(v)

def dfs2(self, v: str, cnt\_count: int):

n = self.nodes[v]

n.component = cnt\_count

count = 0

while u := self.next(v, count):

un = self.nodes[u]

if un.component == 0:

self.dfs2(u, cnt\_count)

count += 1

def generate\_components(self) -> int:

order = []

for key in self.nodes:

n = self.nodes[key]

if not n.visited:

self.dfs1(key, order)

order = order[::-1]

component\_count = 1

for key in order:

n = self.nodes[key]

if n.component == 0:

self.dfs2(key, component\_count)

component\_count += 1

return len(set([n.component for n in self.nodes.values()]))

def reset(self):

for key, value in self.nodes.items():

value.visited = False

value.component = 0

def draw(self, \*, colors=None, title=None):

#pos = nx.planar\_layout(self.g)

# pos = nx.kamada\_kawai\_layout(self.g)

pos = nx.circular\_layout(self.g)

nx.draw\_networkx\_edges(self.g, pos, arrows=False, edge\_color=colors)

nx.draw\_networkx\_nodes(self.g, pos)

nx.draw\_networkx\_labels(self.g, pos)

plt.title(title)

plt.show()

plt.savefig(f'graph-{hash(self.g) + hash(tuple(colors) if colors else None) + hash(title)}.png')

print(f'file graph-{hash(self.g) + hash(tuple(colors) if colors else None) + hash(title)}.png saved!')

## main.py

from copy import deepcopy

from ADTGraph import ADTGraph

def find\_bridges(graph: ADTGraph):

edges = deepcopy(graph.edges)

original\_components = graph.generate\_components()

print(f'OC: {original\_components}')

graph.reset()

bridges: list[tuple[str, str]] = []

for start, end in edges.keys():

test\_graph = deepcopy(graph)

test\_graph.del\_e(start.mark, end.mark)

test\_components = test\_graph.generate\_components()

if test\_components > original\_components:

print(f'TC: {test\_components}')

bridges.append((start.mark, end.mark))

# test\_graph.draw()

color\_map = ['red' if edge in bridges else 'black' for edge in graph.g.edges]

print(bridges)

print(color\_map)

graph.draw(colors=color\_map, title="Graph with colored bridges")

def main():

graph = ADTGraph()

graph.add\_v('1')

graph.add\_v('2')

graph.add\_v('3')

graph.add\_v('4')

graph.add\_v('5')

graph.add\_v('6')

graph.add\_v('7')

graph.add\_e('1', '2')

graph.add\_e('2', '3')

graph.add\_e('3', '4')

graph.add\_e('4', '5')

graph.add\_e('5', '6')

graph.add\_e('6', '7')

graph.draw(title="Graph")

find\_bridges(graph)

if \_\_name\_\_ == '\_\_main\_\_':

main()