Міністерство освіти і науки України
Національний технічний університет України
«Київський політехнічний інститут імені Ігоря Сікорського»
Факультет інформатики та обчислювальної техніки
Кафедра обчислювальної техніки

Лабораторна робота №6

з дисципліни
«Алгоритми і структури даних»

Виконав:

студент групи ІМ-44
Мундурс Нікіта Юрійович
номер у списку групи: 16

Перевірив:

Сергієнко М. А.

Київ 2025

Завдання

1. Представити ненапрямлений граф із заданими параметрами так само, як у лабораторній роботі №3.

   Відмінність 1: коефіцієнт $k = 1.0 - n_3 * 0.01 - n_4 * 0.005 - 0.05$

   Отже, матриця суміжності A_{dir} напрямленого графа за варіантом формується таким чином:

   1) встановлюється параметр (seed) генератора випадкових чисел, рівне номеру варіанту $n_1n_2n_3n_4$;
   2) матриця розміром $n*n$ заповнюється згенерованими випадковими числами в діапазоні [0, 2,0);
   3) обчислюється коефіцієнт $k = 1.0 - n_3 * 0.01 - n_4 * 0.005 - 0.15$, кожен елемент матриці множиться на коефіцієнт k;
   4) елементи матриці округлюються: 0 – якщо елемент менший за 1.0, 1 – якщо елемент більший або дорівнює 1.0.

   Матриця A_{undir} ненапрямленого графа одержується з матриці A_{dir} так само, як у ЛР №3.

   Відмінність 2: матриця ваг W формується таким чином:

   1) матриця B розміром $n*n$ заповнюється згенерованими випадковими числами в діапазоні [0, 2,0) (параметр генератора випадкових чисел той же самий, $n_1n_2n_3n_4$);
   2) одержується матриця C:
      $$c_{ij} = \text{ceil}(b_{ij} \cdot 100 \cdot a_{undir_{i,j}}), \qquad c_{i,j} \in C,\ b_{ij} \in B,\ a_{undir_{i,j}} \in A_{undir},$$
      де ceil — це функція, що округляє кожен елемент матриці до найближчого цілого числа, більшого чи рівного за дане;
   3) одержується матриця D, у якій
      $$d_{ij} = 0, \text{ якщо } c_{ij} = 0,$$
      $$d_{ij} = 1, \text{ якщо } c_{ij} > 0, \quad d_{ij} \in D, c_{ij} \in C;$$
   4) одержується матриця H, у якій
      $$h_{ij} = 1, \text{ якщо } d_{ij} \neq d_{ji},$$
      та $h_{ij} = 0$ в іншому випадку;
   5) Tr – верхня трикутна матриця з одиниць (tr_{ij} при $i < j$)
   6) матриця ваг W симетрична, і її елементи одержуються за формулою: $w_{ij} = w_{ji} = (d_{ij} + h_{ij} * tr_{ij}) * c_{ij}$.

2. Створити програму для знаходження мінімального кістяка за алгоритмом Краскала при n_4 – парному і за алгоритмом Пріма – при непарному. При цьому у програмі:

1) графи представляти у вигляді динамічних списків, обхід графа, додавання, віднімання вершин, ребер виконувати як функції з вершинами відповідних списків;
2) у програмі виконання обходу відображати покроково, черговий крок виконувати за натисканням кнопки у вікні або на клавіатурі.

3. Під час обходу графа побудувати дерево його кістяка. У програмі дерево кістяка виводити покроково у процесі виконання алгоритму. Це можна виконати одним із двох способів:
   1) або виділяти іншим кольором ребра графа;
   2) або будувати кістяк поряд із графом.

При зображенні як графа, так і його кістяка, вказати ваги ребер.

При проєктуванні програми також ***слід врахувати наступне***:

1) мова програмування обирається студентом самостійно;
2) графічне зображення всіх графів має формуватися програмою з тими ж вимогами, як у ЛР №3;
3) усі графи обов’язково зображувати в графічному вікні;
4) типи та структури даних для внутрішнього представлення всіх даних у програмі слід вибрати самостійно;

Варіант: 4416

k = 0,91

Кількість вершин: 11

Розміщення вершин: колом з вершиною в центрі

Алгоритм: Краскала

Обрана мова програмування: Python

Текст програми

```
import turtle
import random
import math
import time
class Vertex:
    def __init__(self, number, pos_x, pos_y):
```

```
        self.number = number
        self.pos_x = pos_x
        self.pos_y = pos_y

class Graph:
    def __init__(self, num_vertices):
        self.edges_list = []
        self.vertices_list = set(range(0, num_vertices))

    def add_edge(self, start_vertex, end_vertex, weight):
        self.edges_list.append((start_vertex, end_vertex, weight))

    def remove_edge(self, edge):
        self.edges_list.remove(edge)

    def remove_vertex(self, vertex):
        self.vertices_list.remove(vertex)

    def get_weight(self, start_vertex, end_vertex):
        for edge in self.edges_list:
            if (edge[0] == start_vertex and edge[1] == end_vertex) or (edge[0] ==
end_vertex and edge[1] == start_vertex):
                return edge[2]

    def sort_edges_by_weight(self):
        self.edges_list = sorted(self.edges_list, key=lambda edge: edge[2])

    def print_info(self):
        print("\nСписок вершин графа:", [x + 1 for x in self.vertices_list])
        print("Список ребер графа:")
        for edge in self.edges_list:
```

```
            print((edge[0] + 1, edge[1] + 1, edge[2]))
        print()


class MST:
    def __init__(self):
        self.num_vertices = 0
        self.edges_list = []
        self.vertices_list = set()

    def add_edge_and_vertex(self, edge):
        start_vertex, end_vertex, weight = edge
        if start_vertex not in self.vertices_list:
            self.vertices_list.add(start_vertex)
            self.num_vertices += 1
        if end_vertex not in self.vertices_list:
            self.vertices_list.add(end_vertex)
            self.num_vertices += 1
        self.edges_list.append((start_vertex, end_vertex, weight))

    def print_info(self):
        print("Список вершин мінімального кістяка:", [x + 1 for x in
self.vertices_list])
        print("Список ребер мінімального кістяка:")
        for edge in self.edges_list:
            print((edge[0] + 1, edge[1] + 1, edge[2]))
        print()


def main():
    seed = 3216
    n3 = 1
    n4 = 6
```

```
    N = 11
    directed_graph_matrix = generateMatrix(n3, n4, N, seed)
    undirected_graph_matrix = doUndir(directed_graph_matrix)
    matrix_b = getMatrixB(N, seed)
    matrix_c = getMatrixC(matrix_b, undirected_graph_matrix)
    matrix_d = getMatrixD(matrix_c)
    matrix_w = getMatrixW(matrix_c, matrix_d)
    graph = createGraph(undirected_graph_matrix, matrix_w)
    mst = MST()
    position = createPositions(undirected_graph_matrix)
    drawVertices(position)
    drawEdges(position, undirected_graph_matrix, graph)
    graph.print_info()
    mst.print_info()
    findTree(graph, mst, position)
    getWeight(mst)
    print("\nПошук мінімального кістяка графа завершено!")
    turtle.hideturtle()
    turtle.done()


def generateMatrix(n3, n4, N, seed):
    random.seed(seed)
    adj_matrix = [[random.random() * 2.0 for _ in range(N)] for _ in range(N)]
    k = 1.0 - n3 * 0.01 - n4 * 0.005 - 0.05
    for i in range(N):
        for j in range(N):
            adj_matrix[i][j] *= k
            adj_matrix[i][j] = 0 if adj_matrix[i][j] < 1.0 else 1
    return adj_matrix
```

```
def doUndir(adj_matrix):
    n = len(adj_matrix)
    undirected_adj_matrix = [[0] * n for _ in range(n)]
    for i in range(n):
        for j in range(n):
            if adj_matrix[i][j] == 1:
                undirected_adj_matrix[i][j] = 1
                undirected_adj_matrix[j][i] = 1
    print("\nМатриця суміжності ненапрямленого графа:")
    for row in undirected_adj_matrix:
        print(row)
    return undirected_adj_matrix

def createPositions(undirected_graph_matrix):
    vertices = []
    center_x = 0
    center_y = 0
    radius = 300
    n = len(undirected_graph_matrix)

    for i in range(1, n):
        angle = i * (2 * math.pi / (n-1))
        x = center_x + radius * math.cos(angle)
        y = center_y + radius * math.sin(angle)
        vertex = Vertex(i, x, y)
        vertices.append(vertex)

    central_vertex = Vertex(n, center_x, center_y)
    vertices.append(central_vertex)
```

```
    return vertices

def drawVertices(vertices):
    turtle.speed(0)
    turtle.penup()
    radius = 20

    for vertex in vertices:
        x, y = vertex.pos_x, vertex.pos_y
        turtle.goto(x, y - radius)
        turtle.pendown()
        turtle.circle(radius)
        turtle.penup()
        turtle.goto(x, y)
        turtle.write(vertex.number, align="center")

def drawEdges(vertices, undirected_graph_matrix, graph):
    turtle.speed(0)
    turtle.penup()

    n = len(vertices)
    radius = 20

    for i in range(n):
        for j in range(i + 1, n):
            if i != j and undirected_graph_matrix[i][j] == 1 and abs(vertices[i].number -
vertices[j].number) != 5:
                x1, y1 = vertices[i].pos_x, vertices[i].pos_y
                x2, y2 = vertices[j].pos_x, vertices[j].pos_y
                x1 += math.cos(math.atan2(y2 - y1, x2 - x1)) * radius
```

```
            y1 += math.sin(math.atan2(y2 - y1, x2 - x1)) * radius
            x2 -= math.cos(math.atan2(y2 - y1, x2 - x1)) * radius
            y2 -= math.sin(math.atan2(y2 - y1, x2 - x1)) * radius

            turtle.penup()
            turtle.goto(x1, y1)
            turtle.pendown()
            turtle.goto(x2, y2)
            turtle.penup()

            mid_x = (x1 + x2) / 2
            mid_y = (y1 + y2) / 2
            turtle.goto(mid_x, mid_y)
            turtle.color("red")
            turtle.write(graph.get_weight(i, j), align="center")
            turtle.color("black")

        if i != j and undirected_graph_matrix[i][j] == 1 and abs(vertices[i].number - vertices[j].number) == 5:
            x1, y1 = vertices[i].pos_x, vertices[i].pos_y
            x2, y2 = vertices[j].pos_x, vertices[j].pos_y
            x1 += math.cos(math.atan2(y2 - y1, x2 - x1)) * radius
            y1 += math.sin(math.atan2(y2 - y1, x2 - x1)) * radius
            x2 -= math.cos(math.atan2(y2 - y1, x2 - x1)) * radius
            y2 -= math.sin(math.atan2(y2 - y1, x2 - x1)) * radius

            angle = math.degrees(math.atan2(y2 - y1, x2 - x1))

            turtle.penup()
            turtle.goto(x1, y1)
            turtle.setheading(angle)
```

```
            turtle.pendown()
            distance = turtle.distance(x2, y2)
            degrees = math.pi * 2
            b = distance / 2 / math.cos(math.radians(degrees))
            turtle.right(degrees)
            turtle.forward(b)
            current_position = turtle.position()
            turtle.setheading(turtle.towards(x2, y2))
            turtle.goto(x2, y2)
            turtle.penup()

            mid_x, mid_y = current_position
            turtle.goto(mid_x, mid_y)
            turtle.color("red")
            turtle.write(graph.get_weight(i, j), align="center")
            turtle.color("black")

for i in range(n):
    if undirected_graph_matrix[i][i] == 1:
        x, y = vertices[i].pos_x, vertices[i].pos_y
        turtle.penup()
        turtle.goto(x + math.pi / 2 * 10, y + math.pi / 2 * 10)
        turtle.pendown()
        turtle.circle(10)
        turtle.penup()

        turtle.goto(x + math.pi / 2 * 10, y + math.pi / 2 * 10 + 4)
        turtle.color("red")
        turtle.write(graph.get_weight(i, i), align="center", font=('Arial', 7, 'normal'))
        turtle.color("black")
```

```
def drawColoredEdges(vertices, i, j):
    turtle.speed(0)
    turtle.penup()
    turtle.pensize(2)
    turtle.color("blue")
    radius = 20

    if i != j and abs(vertices[i].number - vertices[j].number) != 5:
        x1, y1 = vertices[i].pos_x, vertices[i].pos_y
        x2, y2 = vertices[j].pos_x, vertices[j].pos_y
        x1 += math.cos(math.atan2(y2 - y1, x2 - x1)) * radius
        y1 += math.sin(math.atan2(y2 - y1, x2 - x1)) * radius
        x2 -= math.cos(math.atan2(y2 - y1, x2 - x1)) * radius
        y2 -= math.sin(math.atan2(y2 - y1, x2 - x1)) * radius

        turtle.penup()
        turtle.goto(x1, y1)
        turtle.pendown()
        turtle.goto(x2, y2)
        turtle.penup()

    if i != j and abs(vertices[i].number - vertices[j].number) == 5:
        x1, y1 = vertices[i].pos_x, vertices[i].pos_y
        x2, y2 = vertices[j].pos_x, vertices[j].pos_y
        x1 += math.cos(math.atan2(y2 - y1, x2 - x1)) * radius
        y1 += math.sin(math.atan2(y2 - y1, x2 - x1)) * radius
        x2 -= math.cos(math.atan2(y2 - y1, x2 - x1)) * radius
        y2 -= math.sin(math.atan2(y2 - y1, x2 - x1)) * radius
```

```
        angle = math.degrees(math.atan2(y2 - y1, x2 - x1))

        turtle.penup()
        turtle.goto(x1, y1)
        turtle.setheading(angle)
        turtle.pendown()
        distance = turtle.distance(x2, y2)
        degrees = math.pi * 2
        b = distance / 2 / math.cos(math.radians(degrees))
        turtle.right(degrees)
        turtle.forward(b)
        turtle.setheading(turtle.towards(x2, y2))
        turtle.goto(x2, y2)
        turtle.penup()


def drawColoredVertices(vertices, i):
    turtle.speed(0)
    turtle.penup()
    turtle.pensize(3)
    turtle.color("green")
    radius = 20

    x, y = vertices[i].pos_x, vertices[i].pos_y
    turtle.goto(x+2, y - radius)
    turtle.pendown()
    turtle.circle(radius)
    turtle.penup()
    turtle.goto(x, y)
def getMatrixB(N, seed):
```

```
    random.seed(seed)
    matrix_b = [[random.random() * 2.0 for _ in range(N)] for _ in range(N)]
    return matrix_b

def getMatrixC(matrix_b, adj_matrix):
    n = len(matrix_b)
    matrix_c = [[0] * n for _ in range(n)]

    for i in range(n):
        for j in range(n):
            matrix_c[i][j] = matrix_b[i][j] * 100 * adj_matrix[i][j]

    rounded_matrix = [[math.ceil(x) for x in row] for row in matrix_c]
    return rounded_matrix

def getMatrixD(matrix_c):
    n = len(matrix_c)
    matrix_d = [[0] * n for _ in range(n)]

    for i in range(n):
        for j in range(n):
            if matrix_c[i][j] > 0:
                matrix_d[i][j] = 1

    return matrix_d

def getMatrixW(matrix_c, matrix_d):
    n = len(matrix_c)
    matrix_w = [[0] * n for _ in range(n)]
```

```
    for i in range(n):
        for j in range(i, n):
            matrix_w[i][j] = matrix_w[j][i] = matrix_d[i][j] * matrix_c[i][j]

    print("\nМатриця ваг графа:")
    for row in matrix_w:
        print("[{}]".format(" ".join("{: <3}".format(num) for num in row)))

    return matrix_w

def createGraph(adjacency_matrix, weight_matrix):
    num_vertices = len(adjacency_matrix)
    graph = Graph(num_vertices)
    for i in range(num_vertices):
        for j in range(i, num_vertices):
            if adjacency_matrix[i][j] == 1:
                graph.add_edge(i, j, weight_matrix[i][j])
    return graph

def find(parent, vertex):
    if parent[vertex] != vertex:
        parent[vertex] = find(parent, parent[vertex])
    return parent[vertex]

def union(parent, rank, vertex1, vertex2):
    root1 = find(parent, vertex1)
    root2 = find(parent, vertex2)
    if root1 != root2:
        if rank[root1] < rank[root2]:
            parent[root1] = root2
```

```
        elif rank[root1] > rank[root2]:
            parent[root2] = root1
        else:
            parent[root2] = root1
            rank[root1] += 1

def findTree(graph, mst, vertices):
    num_vertices = len(vertices)
    counter = 0
    graph.sort_edges_by_weight()
    parent = {}
    rank = {}

    for vertex in graph.vertices_list:
        parent[vertex] = vertex
        rank[vertex] = 0

    time.sleep(5)
    print("\nВідсортовані за порядком зростання ваги ребра графа:")
    for edge in graph.edges_list:
        print((edge[0] + 1, edge[1] + 1, edge[2]))
    time.sleep(8)

    while graph.edges_list:
        edge = graph.edges_list[0]
        start_vertex, end_vertex, weight = edge
        print("\nДосліджуване ребро:", (edge[0] + 1, edge[1] + 1))
        print("Вага ребра:", edge[2])

        if find(parent, start_vertex) != find(parent, end_vertex):
```

```
            if start_vertex not in mst.vertices_list:
                drawColoredVertices(vertices, start_vertex)
            if end_vertex not in mst.vertices_list:
                drawColoredVertices(vertices, end_vertex)
            if start_vertex in graph.vertices_list:
                graph.remove_vertex(start_vertex)
            if end_vertex in graph.vertices_list:
                graph.remove_vertex(end_vertex)
            drawColoredEdges(vertices, start_vertex, end_vertex)
            mst.add_edge_and_vertex(edge)
            union(parent, rank, start_vertex, end_vertex)
            counter += 1
            print("Ребро не утворює цикл. Додаємо до мінімального кістяка")
            print("Номер ребра:", counter)
        else:
            print("Ребро утворює цикл!")

        graph.remove_edge(edge)
        mst.print_info()
        time.sleep(8)
        if mst.num_vertices == num_vertices:
            break

    print("\nРезультати пошуку:")
    mst.print_info()
    graph.print_info()

def getWeight(mst):
    weight = 0
    for edge in mst.edges_list:
```

```
        weight += edge[2]
    print("\rСума ваг ребер мінімального кістяка:", weight)


main()
```

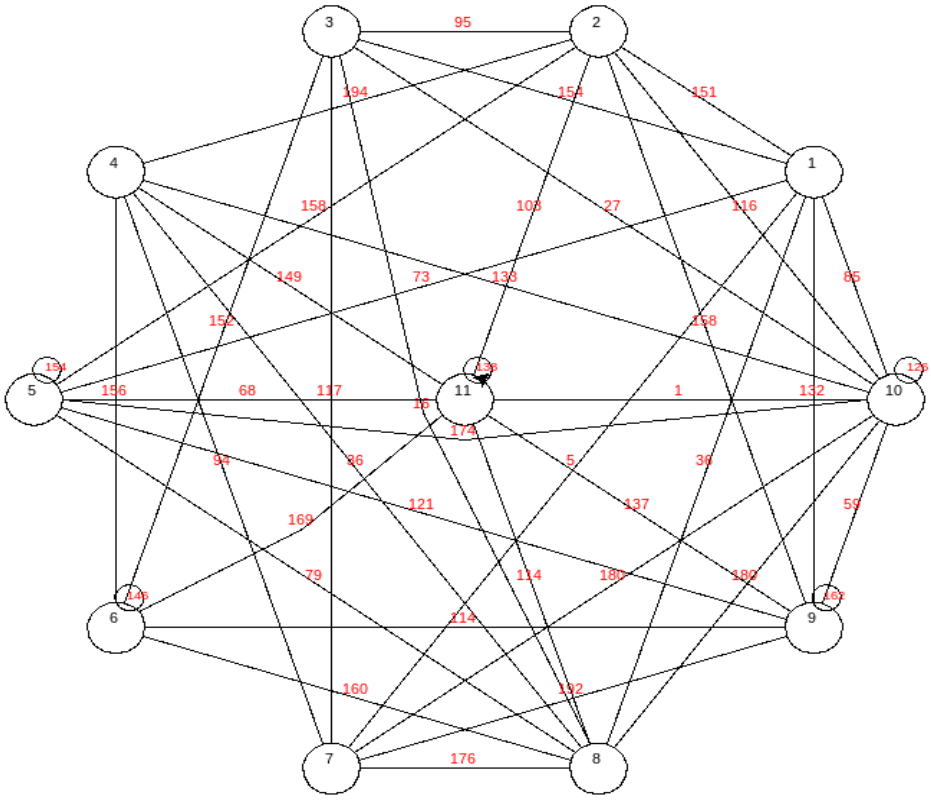
Матриця суміжності ненапрямленого графа та матриця ваг графа

```
Матриця суміжності ненапрямленого графа:
[1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1]
[1, 1, 0, 0, 1, 1, 0, 1, 1, 0, 0]
[1, 0, 1, 1, 1, 1, 0, 0, 1, 1, 0]
[1, 0, 1, 0, 1, 1, 1, 1, 1, 1, 1]
[1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 0]
[1, 1, 1, 1, 1, 0, 1, 0, 0, 1, 0]
[1, 0, 0, 1, 1, 1, 1, 0, 0, 1, 0]
[1, 1, 0, 1, 1, 0, 0, 1, 1, 1, 0]
[1, 1, 1, 1, 1, 0, 0, 1, 0, 1, 1]
[0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1]
[1, 0, 0, 1, 0, 0, 0, 0, 1, 1, 1]
```

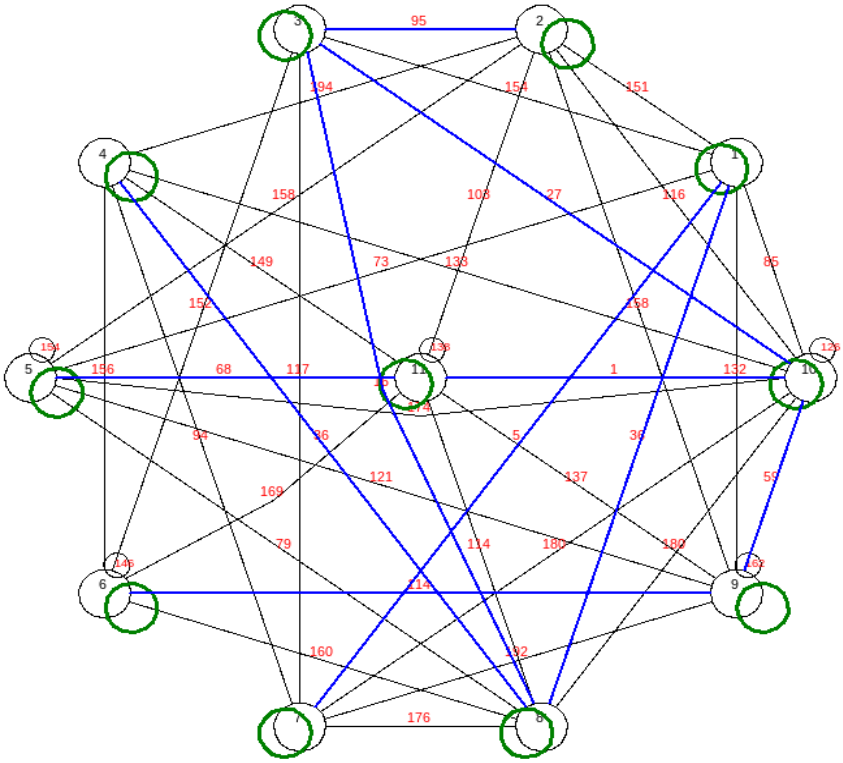
```
Матриця ваг графа:
[185 122 127 158 190 140 147 46  96  0   113]
[122 131 0   0   32  45  0   149 181 0   0  ]
[127 0   194 200 79  123 0   0   126 55  0  ]
[158 0   200 0   188 12  125 89  176 165 134]
[190 32  79  188 0   163 163 178 117 155 0  ]
[140 45  123 12  163 0   36  0   0   177 0  ]
[147 0   0   125 163 36  133 0   0   192 0  ]
[46  149 0   89  178 0   0   131 187 94  0  ]
[96  181 126 176 117 0   0   187 0   160 115]
[0   0   55  165 155 177 192 94  160 190 100]
[113 0   0   134 0   0   0   0   115 100 118]
```

Зображення графа та його мінімального кістяка

Граф



Мінімальний кістяк



Сума ваг ребер знайденого мінімального кістяка

```
Сума ваг ребер мінімального кістяка: 457
```

Висновок: під час виконання лабораторної роботи я навчився використовувати алгоритм Краскала для пошуку мінімального кістяка графа. Також для зберігання компонент зв’язності кістяка було використано union–find data structure. Ребра та вершини мінімального кістяка виділено кольором у графічному вікні, а також у консоль виведено протокол пошуку.