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 №2.5

з дисципліни

«Алгоритми і структури даних»

Виконав:
Студент групи ІМ-34

Никифоров Артем Михайлович

Номер у списку групи:16

Перевірила:
Молчанова А. А.

Київ 2024

Постановка задачі:

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

   Відмінність: коефіцієнт $k = 1.0 - n_3 * 0.01 - n_4 * 0.005 - 0.15$.

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

   1) встановлюється параметр (seed) генератора випадкових чисел, рівне номеру варіанту $n_1 n_2 n_3 n_4$;
   2) матриця розміром $n \cdot n$ 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.

2. Створити програму, яка виконує обхід напрямленого графа вшир (BFS) та вглиб (DFS).

   - обхід починати з вершини із найменшим номером, яка має щонайменше одну вихідну дугу;
   - при обході враховувати порядок нумерації;

- у програмі виконання обходу відображати покроково, черговий крок виконувати за натисканням кнопки у вікні або на клавіатурі.

3. Під час обходу графа побудувати дерево обходу. У програмі дерево обходу виводити покроково у процесі виконання обходу графа. Це можна виконати одним із двох способів:
   - або виділяти іншим кольором ребра графа;
   - або будувати дерево обходу поряд із графом.
4. Зміну статусів вершин у процесі обходу продемонструвати зміною кольорів вершин, графічними позначками тощо, або ж у процесі обходу виводити протокол обходу у графічне вікно або в консоль.
5. Якщо після обходу графа лишилися невідвідані вершини, продовжувати обхід з невідвіданої вершини з найменшим номером, яка має щонайменше одну вихідну дугу.

Завдання для конкретного варіанту:

n1n2n3n4 = 3416

Кількість вершин = 10 + n3 = 11

Розташування колом з вершиною в центрі, тому що n4 = 6

Текст програми:

```
import turtle
import math
import random
from collections import deque

random.seed(3416)

matrix_dir = [[random.uniform(0.0, 2.0) for j in range(11)]
for i in range(11)]

k = 1.0 - 1 * 0.01 - 6 * 0.005 - 0.15

for i in range(len(matrix_dir)):
    for j in range(len(matrix_dir[1])):
        matrix_dir[i][j] *= k
        if matrix_dir[i][j] < 1:
            matrix_dir[i][j] = 0
        else:
            matrix_dir[i][j] = 1

for i in range(len(matrix_dir)):
    print(matrix_dir[i], sep="\n")
print(sep="\n")

screen = turtle.Screen()
screen.setup(width=600, height=600)
screen.bgcolor("white")
turtle.speed(0)
turtle.hideturtle()

algorithm_check = int(input("Виберіть алгоритм для
відображення 1 - BFS, 2 - DFS \n"))

def draw_circles_in_circle():
    radius = 200
    num_circles = 10
    angle = 360 / num_circles
    for i in range(num_circles):
        x = radius * math.cos(math.radians(angle * i))
        y = radius * math.sin(math.radians(angle * i))
        turtle.penup()
        turtle.color('black')
```

```
        turtle.goto(x, y - 20)
        turtle.pendown()
        turtle.begin_fill()
        turtle.circle(20)
        turtle.end_fill()
        turtle.penup()
        turtle.goto(x, y - 10)
        turtle.color('white')
        turtle.write(str(i+1), align="center",
font=("Arial", 12, "normal"))
        turtle.penup()

def draw_11():
    turtle.color('black')
    turtle.penup()
    turtle.goto(0, -20)
    turtle.pendown()
    turtle.begin_fill()
    turtle.circle(20)
    turtle.end_fill()
    turtle.color('white')
    turtle.penup()
    turtle.goto(0, -10)
    turtle.write(str(11), align="center", font=("Arial", 12,
"normal"))
    turtle.penup()

def draw_edges_dir(matrix_dir):
    num_vertices = len(matrix_dir)
    for i in range(num_vertices):
        for j in range(num_vertices):
            if matrix_dir[i][j] == 1:
                x1, y1 = get_vertex_position(i)
                x2, y2 = get_vertex_position(j)
                if matrix_dir[j][i] == 1 or (i == 2 and j ==
7) or (i == 7 and j == 2) or (i == 1 and j == 6) or (i == 9
and j == 4):
                    cursed_line_dir(x1, y1, x2, y2)
                else:
                    draw_dir_line(x1, y1, x2, y2)

def cursed_line_dir(x1, y1, x2, y2):
    turtle.penup()
    turtle.goto(x1, y1)
```

```
    turtle.pendown()
    turtle.color('red')
    turtle.width(1)

    if x1 == x2:
        control_offset = 15
        cx1, cy1 = (x1 + x2) / 2, (y1 + y2) / 2
        if y1 < y2:
            cy1 += control_offset
        else:
            cy1 -= control_offset
        turtle.goto(cx1, cy1)
    else:
        control_offset = 15
        cx1, cy1 = (x1 + x2) / 2, (1.2*y1 + y2) / 2
        if x1 < x2:
            cx1 += control_offset
        else:
            cx1 -= control_offset
        turtle.goto(cx1, cy1)
    turtle.goto(x2, y2)
    turtle_angle = math.degrees(math.atan2(y2 - y1, x2 -
x1))
    turtle.setheading(turtle_angle)
    turtle.stamp()

def draw_circle_dir(x, y):
    turtle.color('blue')
    turtle.penup()
    turtle.goto(x, y)
    turtle.pendown()
    turtle.circle(30)
    turtle.stamp()
    turtle.penup()

def get_vertex_position(vertex_index):
    if vertex_index == 10:
        return 0, 0
    else:
        radius = 200
        num_vertices = 10
        angle = 360 / num_vertices
        x = radius * math.cos(math.radians(angle *
vertex_index))
```

```
        y = radius * math.sin(math.radians(angle *
vertex_index))
        return x, y

def draw_dir_line(x1, y1, x2, y2):
    turtle.penup()
    turtle.goto(x1, y1)
    turtle.pendown()
    turtle.color('blue')
    turtle.width(1)
    end_x_line = x1 + 0.95 * (x2 - x1)
    end_y_line = y1 + 0.95 * (y2 - y1)
    turtle.goto(end_x_line, end_y_line)
    turtle_angle = math.degrees(math.atan2(end_y_line - y1,
end_x_line - x1))
    turtle.setheading(turtle_angle)
    turtle.stamp()

def draw_traversed_edge(x1, y1, x2, y2, curve=False):
    turtle.penup()
    turtle.goto(x1, y1)
    turtle.pendown()
    turtle.color('green')
    turtle.width(2)

    if curve:
        if x1 == x2:
            control_offset = 15
            cx1, cy1 = (x1 + x2) / 2, (y1 + y2) / 2
            if y1 < y2:
                cy1 += control_offset
            else:
                cy1 -= control_offset
            turtle.goto(cx1, cy1)
        else:
            control_offset = 15
            cx1, cy1 = (x1 + x2) / 2, (1.2*y1 + y2) / 2
            if x1 < x2:
                cx1 += control_offset
            else:
                cx1 -= control_offset
            turtle.goto(cx1, cy1)
        turtle.goto(x2, y2)
    else:
```

```
        end_x_line = x1 + 0.95 * (x2 - x1)
        end_y_line = y1 + 0.95 * (y2 - y1)
        turtle.goto(end_x_line, end_y_line)
    turtle_angle = math.degrees(math.atan2(y2 - y1, x2 -
x1))
    turtle.setheading(turtle_angle)
    turtle.stamp()

def bfs_visual(graph, start):
    queue = deque([start])
    visited = set([start])
    result = []
    trav_tree_bfs = [[0]*len(graph) for _ in
range(len(graph))]
    while queue:
        node = queue.popleft()
        result.append(node)

        for neighbor in range(len(graph[node])):
            if graph[node][neighbor] == 1 and neighbor not
in visited:
                x1, y1 = get_vertex_position(node)
                neighbor_x, neighbor_y =
get_vertex_position(neighbor)

                if matrix_dir[node][neighbor] == 1 and
(matrix_dir[neighbor][node] == 1 or (node == 2 and neighbor
== 7) or (node == 7 and neighbor == 2) or (node == 1 and
neighbor == 6) or (node == 9 and neighbor == 4)):
                    draw_traversed_edge(x1, y1, neighbor_x,
neighbor_y, curve=True)
                else:
                    draw_traversed_edge(x1, y1, neighbor_x,
neighbor_y)
                trav_tree_bfs[node][neighbor] = 1
                queue.append(neighbor)
                visited.add(neighbor)

                x, y = get_vertex_position(neighbor)
                turtle.penup()
                turtle.speed(5)
                turtle.width(2)
                turtle.goto(x, y)
                turtle.pendown()
```

```
                turtle.color('green')
                turtle.begin_fill()
                turtle.circle(5)
                turtle.end_fill()
                turtle.penup()
                input("Натискайте Enter щоб зробити
наступний крок")

    screen.update()
    print("Матриця суміжності дерева обходу")
    draw_traversal_tree_matrix(trav_tree_bfs)
    print("")
    return result

def print_vertex_numeration(result):
    print("Список відповідності номерів вершин і їх нової
нумерації набутої в процесі обходу:")
    for index, vertex in enumerate(result):
        print(f"Вершина {vertex+1} визначена як {index+1}")

def dfs_visual(graph, start):
    stack = [start]
    visited = set()
    result = []
    trav_tree_dfs = [[0]*len(graph) for _ in
range(len(graph))]

    while stack:
        node = stack.pop()

        if node not in visited:
            visited.add(node)
            result.append(node)
            x, y = get_vertex_position(node)
            turtle.penup()
            turtle.speed(5)
            turtle.width(2)
            turtle.goto(x, y)
            turtle.pendown()
            turtle.color('green')
            turtle.begin_fill()
            turtle.circle(5)
            turtle.end_fill()
            turtle.penup()
```

```
            input("Натискайте Enter в консолі щоб зробити ще
один крок")

            for neighbor in range(len(graph[node]) - 1, -1,
-1):
                if graph[node][neighbor] == 1 and neighbor
not in visited:
                    x1, y1 = get_vertex_position(node)
                    neighbor_x, neighbor_y =
get_vertex_position(neighbor)

                    if matrix_dir[node][neighbor] == 1 and
(matrix_dir[neighbor][node] == 1 or (node == 2 and neighbor
== 7) or (node == 7 and neighbor == 2) or (node == 1 and
neighbor == 6) or (node == 9 and neighbor == 4)):
                        draw_traversed_edge(x1, y1,
neighbor_x, neighbor_y, curve=True)
                    else:
                        draw_traversed_edge(x1, y1,
neighbor_x, neighbor_y)

                    trav_tree_dfs[node][neighbor] = 1
                    stack.append(neighbor)

    screen.update()
    print("Матриця суміжності дерева обходу")
    draw_traversal_tree_matrix(trav_tree_dfs)
    print("")
    return result

def draw_traversal_tree_matrix(trav_tree_dfs):
    print("\nМатриця дереву обходу:")
    for row in trav_tree_dfs:
        print(' '.join(map(str, row)))

draw_circles_in_circle()
draw_11()
draw_edges_dir(matrix_dir)
if algorithm_check == 1:
    start_node = 0
    bfs_result = bfs_visual(matrix_dir, start_node)
    print("Результат обходу в ширину:", [vertex + 1 for
vertex in bfs_result])
```

```
    print_vertex_numeration(bfs_result)
if algorithm_check == 2:
    start_node = 0
    dfs_result = dfs_visual(matrix_dir, start_node)
    print("Результат обходу в глибину:", [vertex + 1 for
vertex in dfs_result])
    print_vertex_numeration(dfs_result)
screen.exitonclick()
```

Згенерована матриця суміжності напрямленого графа:

```
[0, 1, 0, 1, 0, 0, 0, 1, 0, 0, 0]
[0, 0, 0, 0, 1, 0, 1, 0, 0, 0, 0]
[1, 0, 1, 1, 0, 0, 0, 0, 0, 1, 0]
[1, 0, 1, 0, 0, 1, 1, 1, 0, 1, 1]
[0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 1]
[0, 0, 0, 1, 0, 0, 0, 1, 0, 1, 0]
[0, 0, 0, 1, 0, 1, 0, 0, 0, 0, 1]
[0, 1, 1, 0, 1, 0, 0, 1, 1, 0, 1]
[1, 0, 1, 0, 0, 0, 1, 1, 1, 1, 1]
[0, 0, 0, 0, 1, 1, 1, 1, 1, 0, 0]
[1, 0, 0, 0, 0, 1, 0, 0, 0, 1, 0]
```

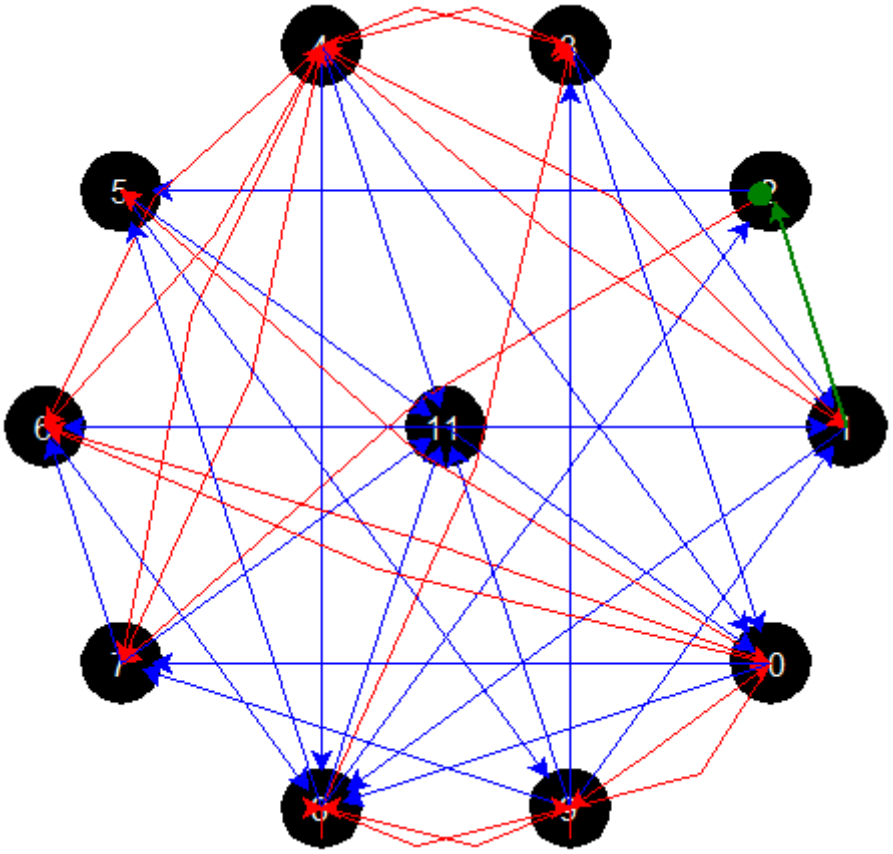
BFS:

Матриця суміжності дерева обходу, список відповідності номерів вершин і їх нової нумерації, скріншоти зображення графа та дерева обходу

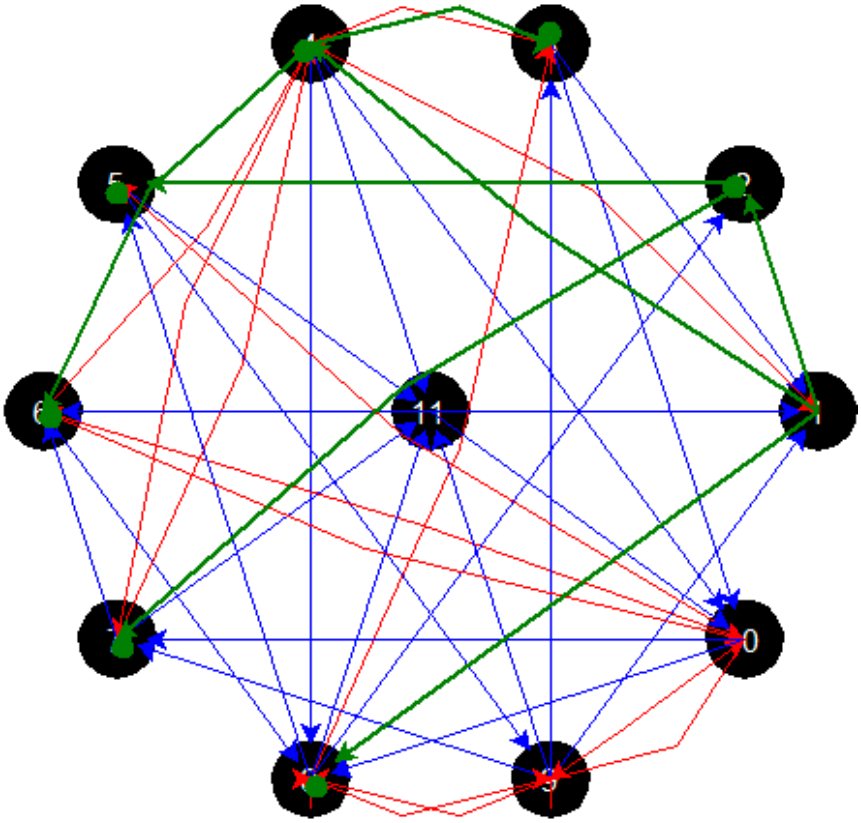
```
Матриця дереву обходу:
0 1 0 1 0 0 0 1 0 0 0
0 0 0 0 1 0 1 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0
0 0 1 0 0 1 0 0 0 1 1
0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 1 0 0
0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0
```

```
Список відповідності номерів вершин і їх нової нумерації набутої в процесі обходу:
Вершина 1 визначена як 1
Вершина 2 визначена як 2
Вершина 4 визначена як 3
Вершина 8 визначена як 4
Вершина 5 визначена як 5
Вершина 7 визначена як 6
Вершина 3 визначена як 7
Вершина 6 визначена як 8
Вершина 10 визначена як 9
Вершина 11 визначена як 10
Вершина 9 визначена як 11
```

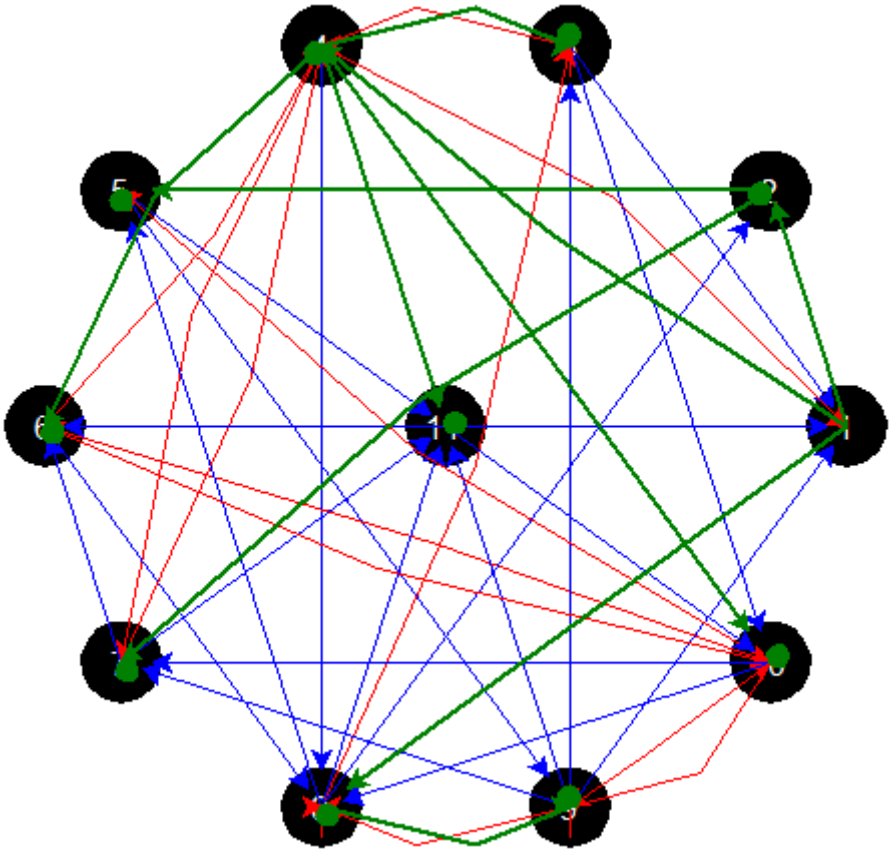
А) на початку



Б) У процесі



В) Після завершення обходу



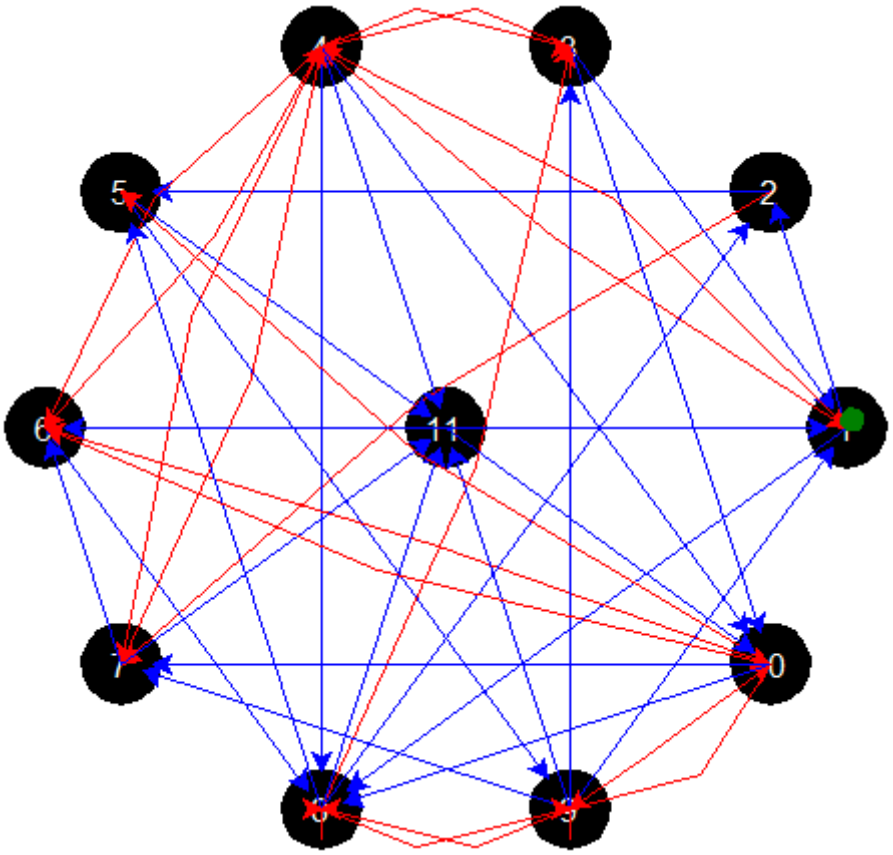
DFS:

Матриця суміжності дерева обходу, список відповідності номерів вершин і їх нової нумерації, скріншоти зображення графа та дерева обходу

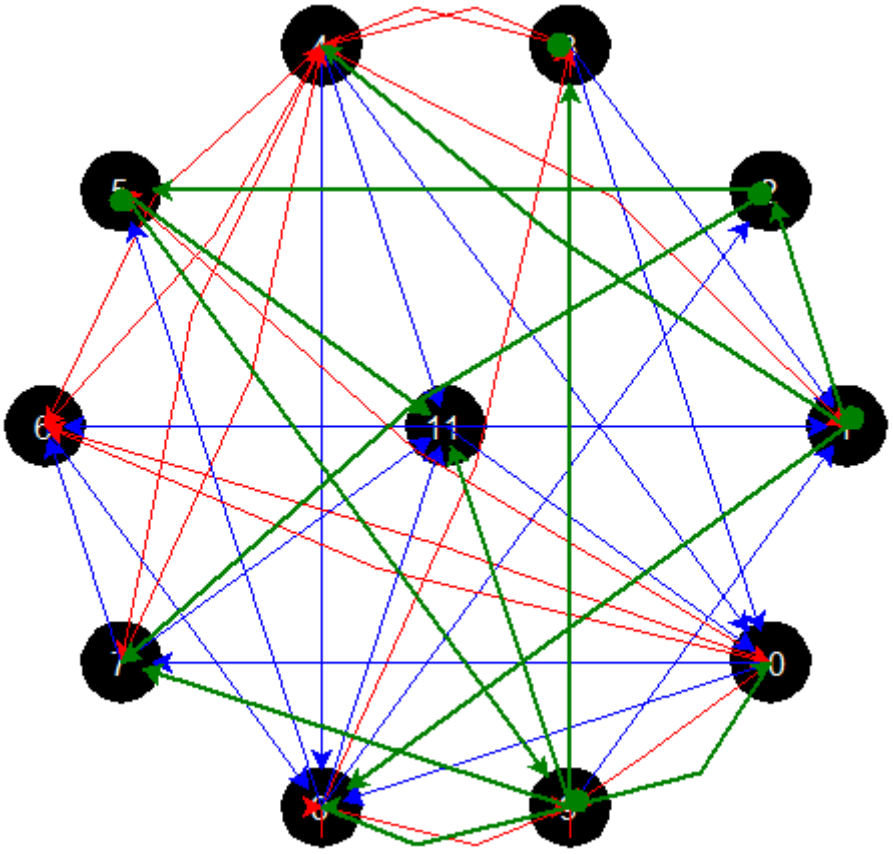
```
Матриця дереву обходу:
0 1 0 1 0 0 0 1 0 0 0
0 0 0 0 1 0 1 0 0 0 0
0 0 0 1 0 0 0 0 0 1 0
0 0 0 0 0 1 1 1 0 1 1
0 0 0 0 0 0 0 0 1 0 1
0 0 0 0 0 0 0 1 0 1 0
0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 1
0 0 1 0 0 0 1 1 0 1 1
0 0 0 0 0 0 1 0 0 0 0
0 0 0 0 0 0 0 0 0 1 0
```

```
Список відповідності номерів вершин і їх нової нумерації набутої в процесі обходу:
Вершина 1 визначена як 1
Вершина 2 визначена як 2
Вершина 5 визначена як 3
Вершина 9 визначена як 4
Вершина 3 визначена як 5
Вершина 4 визначена як 6
Вершина 6 визначена як 7
Вершина 8 визначена як 8
Вершина 11 визначена як 9
Вершина 10 визначена як 10
Вершина 7 визначена як 11
```

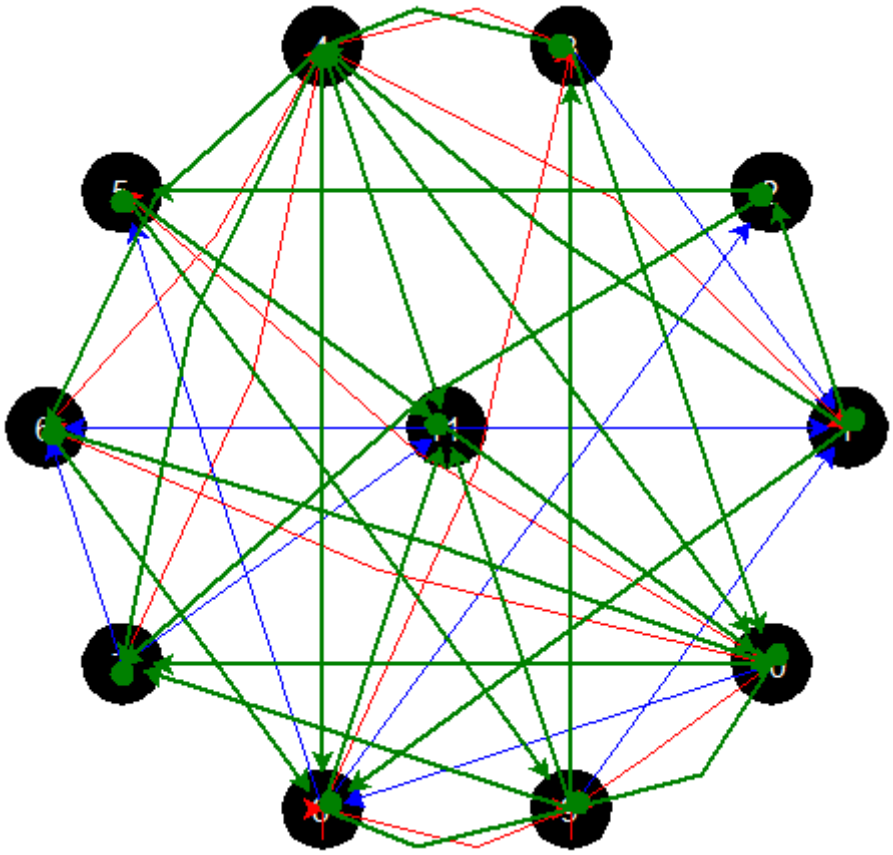
А) На початку



Б) У процесі



В) Після завершення обходу



Висновки:

Обидва алгоритми мають свої переваги та недоліки. Вибір алгоритму залежить від конкретної задачі та умов.

Використовувати BFS потрібно тоді, коли треба знайти найкоротший шлях у графі, якщо він має велику ширину, але малу глибину.

Використовувати DFS потрібно тоді, коли треба знайти найкоротший шлях у графі, якщо він має невелику ширину, але велику глибину.

BFS находить найкоротший шлях, але споживає багато пам’яті

DFS споживає мало пам’яті, але не обов’язково знаходить найкоротший шлях