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 №5

з дисципліни
«Алгоритми і структури даних»

Виконав:
студент групи ІМ-42
Туров Андрій Володимирович
номер у списку групи: 28

Перевірив:
Сергієнко А. М.

Київ 2025

Завдання

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

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

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.

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.

Варіант 28

$\overline{n_1 n_2 n_3 n_4} = 4228$;
Кількість вершин — $10 + n_3 = 12$.
Розміщення вершин — прямокутником з вершиною в центрі.

Текст програм

```rust
use std::f32::consts::PI;

use raylib::prelude::*;

const FONT_SIZE: i32 = 32;
const CHAR_WIDTH: f32 = 0.27;

const VERTEX_RADIUS: f32 = 20.0;
const VERTEX_WIDTH: u32 = 3;
```

```
const ARROWHEAD_LEN: f32 = VERTEX_RADIUS * 0.5;
const ARROWHEAD_ANGLE: f32 = PI / 6.0;

pub fn draw_text(
    d: &mut RaylibDrawHandle,
    font: &Font,
    text: &str,
    position: Vector2,
    color: Color,
) {
    d.draw_text_ex(font, text, position, FONT_SIZE as f32, 0.0,
    ↪  color);
}

pub fn draw_vertex(
    d: &mut RaylibDrawHandle,
    center: Vector2,
    weight: &str,
    font: &Font,
    color: Color,
) {
    (0..VERTEX_WIDTH).for_each(|w| {
        d.draw_circle_lines(
            center.x as i32,
            center.y as i32,
            VERTEX_RADIUS - w as f32,
            color,
        );
    });
    if !weight.is_empty() {
        let text_len = weight.chars().count();
        let x_offset: f32 = text_len as f32 * FONT_SIZE as f32 *
        ↪  CHAR_WIDTH;
        let y_offset: f32 = FONT_SIZE as f32 * 0.5;
        draw_text(
            d,
            font,
            weight,
            Vector2 {
                x: center.x - x_offset,
                y: center.y - y_offset,
            },
            Color::BLACK,
        )
    }
}

fn draw_arrowhead(d: &mut RaylibDrawHandle, position: Vector2,
↪  direction: Vector2, color: Color) {
    d.draw_line_v(
        position,
```

```
            position - direction.rotated(ARROWHEAD_ANGLE) *
             ↪  ARROWHEAD_LEN,
            color,
        );
        d.draw_line_v(
            position,
            position - direction.rotated(-ARROWHEAD_ANGLE) *
             ↪  ARROWHEAD_LEN,
            color,
        );
    }

    pub fn draw_straight_edge(
        d: &mut RaylibDrawHandle,
        center_from: Vector2,
        center_to: Vector2,
        directional: bool,
        color: Color,
    ) {
        let direction = (center_to - center_from).normalized();

        let from = center_from + direction * VERTEX_RADIUS;
        let to = center_to - direction * VERTEX_RADIUS;
        d.draw_line_v(from, to, color);
        if directional {
            draw_arrowhead(d, to, direction, color);
        }
    }

    pub fn draw_angled_edge(
        d: &mut RaylibDrawHandle,
        center_from: Vector2,
        center_to: Vector2,
        directed: bool,
        color: Color,
    ) {
        const EDGE_BASE_ANGLE: f32 = 0.05 * PI;

        let direction = (center_to - center_from).normalized();
        let from = center_from + direction * VERTEX_RADIUS;
        let to = center_to - direction * VERTEX_RADIUS;

        let vector = to - from;
        let vector_middle = vector * 0.5;
        let mid_offset = vector_middle.length() *
         ↪  EDGE_BASE_ANGLE.tan();

        let midpoint = from + vector_middle + direction.rotated(0.5 *
         ↪  PI) * mid_offset;
        d.draw_line_v(from, midpoint, color);
        d.draw_line_v(midpoint, to, color);
        if directed {
```

```
        draw_arrowhead(d, to, (to - midpoint).normalized(),
        ↪  color);
    }
}

pub fn draw_looping_edge(d: &mut RaylibDrawHandle, center:
↪  Vector2, color: Color) {
    const POINTS: usize = 16;
    const RADIUS: f32 = 12.0;
    const START_ANGLE: f32 = -0.9 * PI;
    const END_ANGLE: f32 = 0.75 * PI;

    let step = (END_ANGLE - START_ANGLE) / POINTS as f32;
    let start_point = center
        + Vector2 {
            x: VERTEX_RADIUS,
            y: -0.85 * VERTEX_RADIUS,
        };

    let points: [Vector2; POINTS] = std::array::from_fn(|i|
    ↪  Vector2 {
        x: start_point.x + RADIUS * f32::cos(START_ANGLE + (step
        ↪  * i as f32)),
        y: start_point.y + RADIUS * f32::sin(START_ANGLE + (step
        ↪  * i as f32)),
    });
    let last_point = points[POINTS - 1];
    d.draw_line_strip(&points, color);

    let direction = (last_point - points[POINTS -
    ↪  4]).normalized();
    draw_arrowhead(d, last_point, direction, color);
}
```

Файл 1: draw.rs

```
use std::{
    collections::VecDeque,
    fmt::{Debug, Display},
    marker::PhantomData,
};

use rand::{Rng, SeedableRng, rngs::SmallRng};

// $\overline{n_1n_2n_3n_4} = 4228$
pub const DEFAULT_ROWS: &[usize] = &[4, 3, 5];
pub const VERTEX_COUNT: usize = 12; // $10 + n_3 = 12$
const RANDOM_SEED: u64 = 4228;

#[derive(Clone)]
```

```rust
pub struct AdjMatrix(pub Vec<Vec<u32>>);

impl Display for AdjMatrix {
    fn fmt(&self, f: &mut std::fmt::Formatter<'_>) ->
    ↪  std::fmt::Result {
        for i in 0..self.0.len() {
            for j in 0..self.0.len() {
                if i == j {
                    write!(f, "\x1b[31m{}\x1b[0m ",
                    ↪  self.0[i][j])?;
                    continue;
                }
                write!(f, "{} ", self.0[i][j])?;
            }
            writeln!(f)?;
        }
        Ok(())
    }
}

#[derive(Debug)]
pub struct Bfs;
#[derive(Debug)]
pub struct Dfs;

pub trait Search {}
impl Search for Bfs {}
impl Search for Dfs {}

#[derive(Debug)]
pub struct SearchStep<S: Search> {
    pub active: usize,
    pub visited: Vec<(usize, usize)>,
    pub queue: VecDeque<usize>,
    pub tree: Vec<(usize, usize)>,
    marker: PhantomData<S>,
}

impl<S: Search> SearchStep<S> {
    pub fn new(active: usize, size: usize) -> Self {
        let mut visited = Vec::with_capacity(size);
        visited.push((active, active));
        let mut queue = VecDeque::with_capacity(size);
        queue.push_back(active);

        Self {
            active,
            visited,
            queue,
            tree: Vec::with_capacity(size),
            marker: PhantomData,
```

```
        }
    }
}

pub trait Queue {
    fn push_queue(&mut self, value: usize);
    fn pop_queue(&mut self) -> Option<usize>;
}

impl Queue for SearchStep<Bfs> {
    fn push_queue(&mut self, value: usize) {
        self.queue.push_back(value);
    }

    fn pop_queue(&mut self) -> Option<usize> {
        self.queue.pop_front()
    }
}

impl Queue for SearchStep<Dfs> {
    fn push_queue(&mut self, value: usize) {
        self.queue.push_back(value);
    }

    fn pop_queue(&mut self) -> Option<usize> {
        self.queue.pop_back()
    }
}

impl AdjMatrix {
    pub fn generate(k: f32) -> Self {
        let mut rng = SmallRng::seed_from_u64(RANDOM_SEED);
        let iter = std::iter::repeat_with(move ||
        ↪  rng.random_range(0.0..2.0));

        AdjMatrix(
            iter.take(VERTEX_COUNT * VERTEX_COUNT)
                .map(|i| f32::min(i * k, 1.0) as u32)
                .collect::<Vec<_>>()
                .chunks(VERTEX_COUNT)
                .map(|row| row.to_vec())
                .collect(),
        )
    }

    pub fn search_next<S: Search>(&self, step: &mut
    ↪  SearchStep<S>) -> bool
    where
        SearchStep<S>: Queue,
    {
        if let Some(next) = step.pop_queue() {
```

```
            step.active = next;
            step.tree.push(
                *step
                    .visited
                    .iter()
                    .find(|(_, to)| *to == next)
                    .unwrap_or(&(next, next)),
            );
            self.0[next].iter().enumerate().for_each(|(i, edge)|
             ↪  {
                if *edge == 1 && !step.visited.iter().any(|(_,
                 ↪  to)| *to == i) {
                    step.visited.push((next, i));
                    step.push_queue(i);
                }
            });
            true
        } else {
            if let Some(unvisited) =
                (0..self.0.len()).find(|vertex|
                 ↪  !step.tree.iter().any(|(_, to)| to ==
                 ↪  vertex))
            {
                step.push_queue(unvisited);
                self.search_next::<S>(step);
                return true;
            }
            false
        }
    }
}

impl<S: Search> From<&SearchStep<S>> for AdjMatrix {
    fn from(value: &SearchStep<S>) -> Self {
        let size = value.tree.len();
        let mut result = vec![vec![0_u32; size]; size];
        for (from, to) in &value.tree {
            if *from != *to {
                result[*from][*to] = 1;
            }
        }
        AdjMatrix(result)
    }
}
```

Файл 2: graph.rs

```
#![allow(clippy::needless_range_loop)]

use graph::{AdjMatrix, Bfs, DEFAULT_ROWS, Dfs, Search,
 ↪  SearchStep, VERTEX_COUNT};
```

```rust
use raylib::{color::Color, prelude::*};
mod draw;
mod graph;

const WIN_WIDTH: i32 = 800;
const WIN_HEIGHT: i32 = 600;
const WIN_MARGIN: f32 = 0.8;

const K: f32 = 1.0 - 2.0 * 0.01 - 8.0 * 0.005 - 0.15;

#[derive(Debug, Clone, Copy)]
struct VertexPos {
    v: Vector2,
    row: usize,
    col: usize,
}

fn draw_all_vertices<S: Search>(
    d: &mut RaylibDrawHandle,
    font: &Font,
    rows: &[usize],
    step: &SearchStep<S>,
) -> Vec<VertexPos> {
    fn current_position(index: usize, rows: &[usize]) -> (usize,
    ↪  usize) {
        let mut cumulative = 0;
        for (row, &count) in rows.iter().enumerate() {
            if index < cumulative + count {
                return (row, index - cumulative);
            }
            cumulative += count;
        }
        (usize::MAX, usize::MAX)
    }
    let winwidth = WIN_WIDTH as f32 * WIN_MARGIN;
    let winheight = WIN_HEIGHT as f32 * WIN_MARGIN;
    let vertex_count = rows.iter().sum();
    let vertex_coords: Vec<VertexPos> =
    ↪  Vec::from_iter((0..vertex_count).map(|i| {
        let (row, col) = current_position(i, rows);

        let x_offset = (WIN_WIDTH as f32 - winwidth) * 0.5;
        let y_offset = (WIN_HEIGHT as f32 - winheight) * 0.5;
        VertexPos {
            v: Vector2 {
                x: (winwidth / (DEFAULT_ROWS[row] - 1) as f32 *
                ↪  col as f32) + x_offset,
                y: (winheight / (DEFAULT_ROWS.len() - 1) as f32 *
                ↪  row as f32) + y_offset,
            },
            row,
```

```
                col,
            }
        }));

        (0..vertex_count).for_each(|i| {
            draw::draw_vertex(
                d,
                vertex_coords[i].v,
                &((i + 1).to_string()),
                font,
                if step.active == i {
                    Color::RED
                } else if step.visited.iter().any(|(_, to)| *to == i)
                 ↪  {
                    Color::PLUM
                } else {
                    Color::BLACK
                },
            );
        });

        vertex_coords
    }

    fn draw_all_edges<S: Search>(
        d: &mut RaylibDrawHandle,
        adj_matrix: &AdjMatrix,
        vertex_coords: &[VertexPos],
        directed: bool,
        step: &SearchStep<S>,
        hide_edges: bool,
    ) {
        for i in 0..vertex_coords.len() {
            let lower = if directed { 0 } else { i };
            for j in lower..vertex_coords.len() {
                let origin = vertex_coords[i];
                let destination = vertex_coords[j];
                let row_absdiff = (destination.row as i64 -
                 ↪  origin.row as i64).abs();
                let col_absdiff = (destination.col as i64 -
                 ↪  origin.col as i64).abs();

                if adj_matrix.0[i][j] == 1 {
                    let color = if i != j &&
                     ↪  step.tree.iter().any(|(from, to)| *from == i
                     ↪  && *to == j) {
                        Color::RED
                    } else if hide_edges {
                        Color::WHITE.alpha(0.0)
                    } else {
                        Color::BLACK
```

```
                };

                if i == j {
                    draw::draw_looping_edge(d, origin.v, color);
                } else if (adj_matrix.0[j][i] == 1 && directed)
                 ↪  // symmetric
                    || (row_absdiff == 0 && col_absdiff > 1) //
                     ↪  same row, goes through others
                    || (col_absdiff == 0 && row_absdiff > 1) //
                     ↪  same col, goes through others
                    || (origin.v.x == destination.v.x) // same x
                     ↪  coordinate, yes, still possible
                    || col_absdiff >= 3
                // honestly ^ whatever this is
                {
                    draw::draw_angled_edge(d, origin.v,
                     ↪  destination.v, directed, color);
                } else {
                    draw::draw_straight_edge(d, origin.v,
                     ↪  destination.v, directed, color);
                }
            }
        }
    }
}

fn draw_controls(d: &mut RaylibDrawHandle, font: &Font, state:
 ↪  KeyboardKey, hide_edges: bool) {
    draw::draw_text(
        d,
        font,
        "<F1>",
        Vector2 {
            x: 0.05 * WIN_WIDTH as f32,
            y: 0.01 * WIN_HEIGHT as f32,
        },
        if state == KeyboardKey::KEY_F1 {
            Color::RED
        } else {
            Color::BLACK
        },
    );
    draw::draw_text(
        d,
        font,
        "<F2>",
        Vector2 {
            x: 0.47 * WIN_WIDTH as f32,
            y: 0.01 * WIN_HEIGHT as f32,
        },
        if state == KeyboardKey::KEY_F2 {
```

```
            Color::RED
        } else {
            Color::BLACK
        },
    );
    draw::draw_text(
        d,
        font,
        "<F3> Hide edges",
        Vector2 {
            x: 0.68 * WIN_WIDTH as f32,
            y: 0.01 * WIN_HEIGHT as f32,
        },
        if hide_edges { Color::RED } else { Color::BLACK },
    );
    draw::draw_text(
        d,
        font,
        "<Space> Step",
        Vector2 {
            x: 0.37 * WIN_WIDTH as f32,
            y: 0.94 * WIN_HEIGHT as f32,
        },
        Color::BLACK,
    );
}

fn print_new_order<S: Search>(step: &SearchStep<S>) {
    println!("New vertex order:");
    step.visited
        .iter()
        .enumerate()
        .map(|(index, (_, to))| format!("{}->{}", to + 1, index +
        ↪  1))
        .for_each(|s| print!("{} ", s));
    println!("\n");
}

fn main() {
    let (mut rl, thread) = raylib::init()
        .size(WIN_WIDTH, WIN_HEIGHT)
        .log_level(TraceLogLevel::LOG_WARNING)
        .title("ASD Lab 2.4")
        .build();

    let font = rl.load_font(&thread,
    ↪  "FiraCode-Regular.ttf").unwrap();

    let matrix = AdjMatrix::generate(K);
    let start_vertex = matrix
        .0
```

```
        .iter()
        .position(|row| row.iter().all(|v| *v != 0))
        .unwrap_or(0);
    let mut bfs = SearchStep::new(start_vertex, VERTEX_COUNT);
    let mut dfs = SearchStep::new(start_vertex, VERTEX_COUNT);
    let mut state = KeyboardKey::KEY_F1;
    let mut hide_edges = false;

    println!("Graph:\n{}", matrix);

    while !rl.window_should_close() {
        if rl.is_key_pressed(KeyboardKey::KEY_F1) {
            state = KeyboardKey::KEY_F1;
        } else if rl.is_key_pressed(KeyboardKey::KEY_F2) {
            state = KeyboardKey::KEY_F2;
        } else if rl.is_key_pressed(KeyboardKey::KEY_F3) {
            hide_edges = !hide_edges;
        }

        if rl.is_key_pressed(KeyboardKey::KEY_SPACE) {
            match state {
                KeyboardKey::KEY_F1 => {
                    if !matrix.search_next::<Bfs>(&mut bfs) {
                        print_new_order(&bfs);

                        let matrix: AdjMatrix = (&bfs).into();
                        println!("BFS tree:\n{}", matrix)
                    }
                }
                KeyboardKey::KEY_F2 => {
                    if !matrix.search_next::<Dfs>(&mut dfs) {
                        print_new_order(&dfs);

                        let matrix: AdjMatrix = (&dfs).into();
                        println!("DFS tree:\n{}", matrix);
                    }
                }
                _ => {}
            }
        }
        let mut d = rl.begin_drawing(&thread);

        d.clear_background(Color::WHITE);
        draw_controls(&mut d, &font, state, hide_edges);

        if state == KeyboardKey::KEY_F1 {
            let vertex_coords = draw_all_vertices(&mut d, &font,
             ↪  DEFAULT_ROWS, &bfs);
            draw_all_edges(&mut d, &matrix, &vertex_coords, true,
             ↪  &bfs, hide_edges);
        } else if state == KeyboardKey::KEY_F2 {
```

```
                let vertex_coords = draw_all_vertices(&mut d, &font,
                 ↪  DEFAULT_ROWS, &dfs);
                draw_all_edges(&mut d, &matrix, &vertex_coords, true,
                 ↪  &dfs, hide_edges);
            }
        }
}
```

Файл 3: main.rs

Матриці суміжності

Для напрямленого графа:

```
0 1 1 1 0 0 1 0 0 0 1 0
0 1 0 1 0 0 0 1 0 1 1 1
0 0 1 0 0 1 1 1 0 1 1 0
0 0 1 1 0 1 0 1 1 1 0 1
0 0 1 0 0 1 1 0 0 0 1 0
0 0 0 0 0 1 0 1 0 1 0 0
0 0 1 1 0 0 0 0 0 0 1 0
0 0 1 0 0 0 0 0 0 1 0 0
0 1 0 0 0 1 0 0 1 0 1 1
1 0 1 0 1 0 1 1 0 0 0 0
0 0 1 0 0 1 0 0 0 0 1 1
1 0 1 1 0 1 1 0 1 1 1 1
```

Для дерева BFS:

```
0 1 1 1 0 0 1 0 0 0 1 0
0 0 0 0 0 0 0 1 0 1 0 1
0 0 0 0 0 1 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 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
```

Для дерева DFS:

```
0 1 1 1 0 0 1 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 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 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 1 0 0 0 0
0 0 0 0 0 1 0 0 0 0 0 1
0 0 0 0 0 0 0 0 1 1 0 0
```

Нові порядки вершин

BFS:
1→1 2→2 3→3 4→4 7→5
11→6 8→7 10→8 12→9 6→10
9→11 5→12

DFS:
1→1 2→2 3→3 4→4 7→5
11→6 6→7 12→8 9→9 10→10
5→11 8→12

Зображення

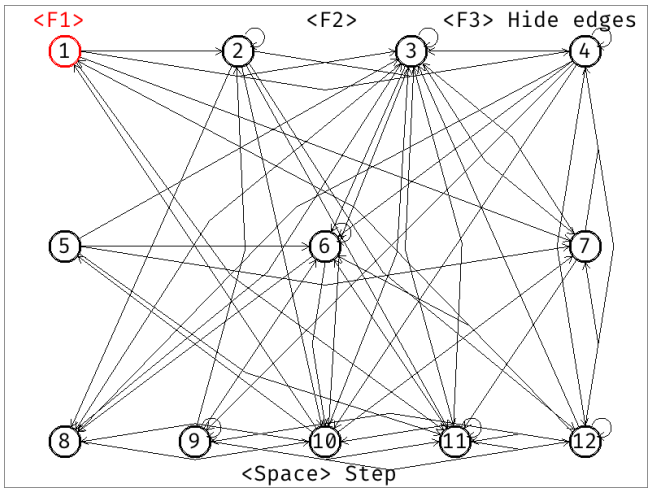

Рис. 1: Граф на початку обходу

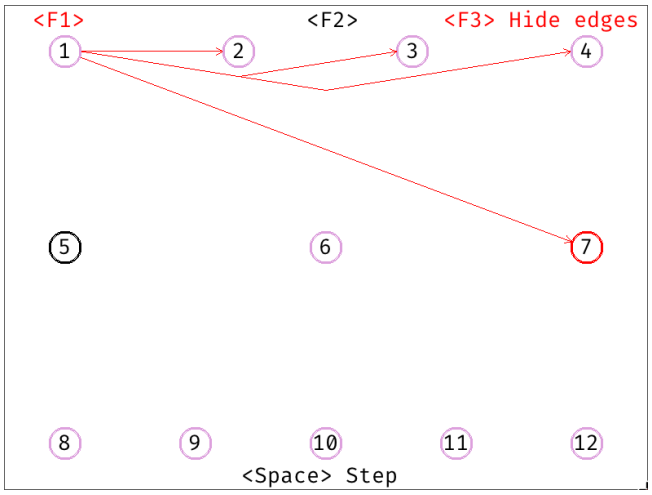

Рис. 2: Граф на середині обходу (ребра приховані)

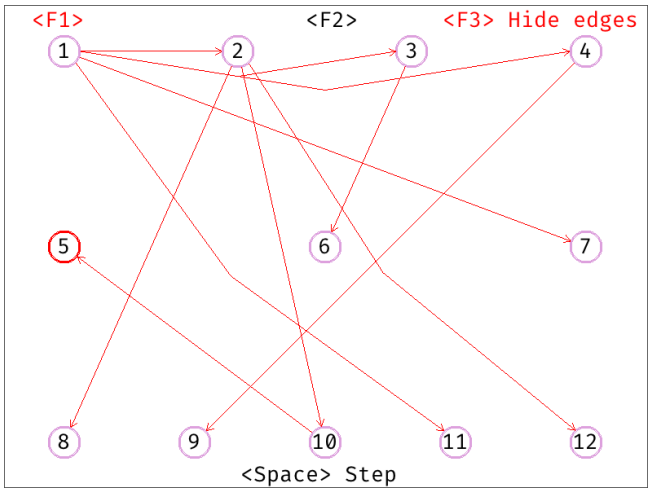

Рис. 3: Дерево обходу BFS

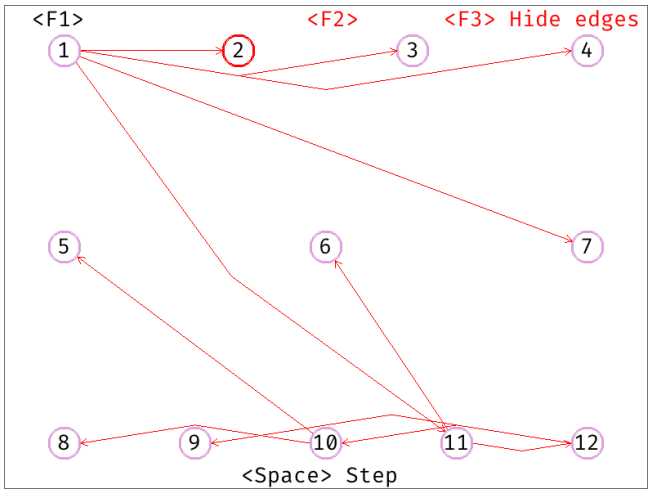

Рис. 4: Дерево обходу DFS

Висновок

Імплементував BFS та DFS алгоритми обходу графа, що виконуються покроково. Під час написання коду помітив, що обидва методи відрізняються лише структурою даних для черги, тому абстрагував крок обходу як операцію над деяким `SearchStep<Q>`, де `Q` задає методи деку `push` та `pop` (для BFS — з різних кінців списку, для DFS — з одного).