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

   Відмінність 2: матриця ваг W формується таким чином:

   (а) матриця B розміром $n * n$ заповнюється згенерованими випадковими числами в діапазоні $[0, 2.0)$;

   (б) одержується матриця C:

   $$c_{ij} = \text{ceil}(b_{ij} * 100 * a_{undir_{ij}}),$$
   $$c_{ij} \in C,\ b_{ij} \in B,\ a_{undir_{ij}} \in A_{undir}$$

   (в) одержується матриця D, у якій

   $$d_{ij} = 0, \text{ якщо } c_{ij} = 0,$$
   $$d_{ij} = 1, \text{ якщо } c_{ij} > 0;$$
   $$d_{ij} \in D, c_{ij} \in C;$$

   (г) одержується матриця H, у якій

   $$h_{ij} = 1, \text{ якщо } d_{ij} \neq d_{ji},$$
   $$\text{та } h_{ij} = 0 \text{ в іншому випадку;}$$

   (д) Tr — верхня трикутна матриця з одиниць ($tr_{ij} = 1$ при $i < j$);

   (е) 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:

   - графи представляти у вигляді динамічних списків, обхід графа, додавання, віднімання вершин, ребер виконувати як функції з вершинами відповідних списків;
   - 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.

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.
Знаходження мінімального кістяка за допомогою алгоритма Краскала.

Текст програм

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

use raylib::prelude::*;

const VERTEX_FONT_SIZE: i32 = 32;
const WEIGHT_FONT_SIZE: i32 = 24;

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

const EDGE_BASE_ANGLE: f32 = 0.05 * PI;
const WEIGHT_TEXT_OFFSET: f32 = 0.35; // 0.0..=0.5

pub fn draw_text_pro(
    d: &mut RaylibDrawHandle,
    font: &Font,
    text: &str,
    position: Vector2,
    rotation: f32,
    font_size: f32,
    color: Color,
) {
    d.draw_text_pro(
        font,
        text,
        position,
        Vector2 {
            x: font.measure_text(text, font_size, 0.0).x * 0.5,
            y: 0.0,
        },
        rotation * 180.0 / PI,
        font_size,
        0.0,
        color,
    );
}

pub fn draw_text(
    d: &mut RaylibDrawHandle,
    font: &Font,
    text: &str,
```

```
    position: Vector2,
    font_size: f32,
    color: Color,
) {
    draw_text_pro(d, font, text, position, 0.0, font_size,
     ↪  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 y_offset: f32 = VERTEX_FONT_SIZE as f32 * 0.5;
        draw_text(
            d,
            font,
            weight,
            Vector2 {
                x: center.x,
                y: center.y - y_offset,
            },
            VERTEX_FONT_SIZE as f32,
            Color::BLACK,
        )
    }
}

fn draw_arrowhead(d: &mut RaylibDrawHandle, position: Vector2,
 ↪  direction: Vector2, color: Color) {
    const ARROWHEAD_LEN: f32 = VERTEX_RADIUS * 0.5;
    const ARROWHEAD_ANGLE: f32 = PI / 6.0;
    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_edge_weight(
    d: &mut RaylibDrawHandle,
    center_from: Vector2,
    center_to: Vector2,
    weight: &str,
    font: &Font,
    color: Color,
    angled: bool,
) {
    let direction = center_to - center_from;
    let normal = direction.normalized().rotated(0.5 * PI);
    let offset = direction * WEIGHT_TEXT_OFFSET;

    let mut text_pos = center_from + offset;
    let mut angle = Vector2 { x: 1.0, y: 0.0
    ↪  }.angle_to(direction);
    if angled {
        text_pos += normal * (offset.length() *
        ↪  EDGE_BASE_ANGLE.tan());
        angle = Vector2 { x: 1.0, y: 0.0 }.angle_to(text_pos -
        ↪  center_from);
    }
    draw_text_pro(
        d,
        font,
        weight,
        text_pos,
        angle,
        WEIGHT_FONT_SIZE as f32,
        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,
) {
    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::Display};

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

// n₁n₂n₃n₄ = 4228
pub const DEFAULT_ROWS: &[usize] = &[4, 3, 5];
pub const VERTEX_COUNT: usize = 12; // 10 + n₃ = 12
const RANDOM_SEED: u64 = 4228;

#[derive(Clone)]
pub struct Graph(pub Vec<Vec<Option<u32>>>);

impl Display for Graph {
    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 let Some(weight) = self.0[i][j] {
                    write!(f, "{:>3} ", weight)?;
                } else {
                    write!(f, "{:>3} ", "∞")?;
                }
            }
            writeln!(f)?;
        }
        Ok(())
    }
}

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

        let mut adj_undir: Vec<Vec<u32>> = iter
            .clone()
            .take(ITER_LEN)
            .map(|i| f32::min(i * k, 1.0) as u32)
```

```
    .collect::<Vec<_>>()
    .chunks(VERTEX_COUNT)
    .map(|row| row.to_vec())
    .collect();
for i in 0..VERTEX_COUNT {
    for j in (i + 1)..VERTEX_COUNT {
        adj_undir[j][i] = adj_undir[i][j];
    }
}

let c: Vec<Vec<u32>> = iter
    .take(ITER_LEN)
    .enumerate()
    .map(|(i, w)| {
        (w * 100.0 * adj_undir[i / VERTEX_COUNT][i %
        ↪  VERTEX_COUNT] as f32).ceil() as u32
    })
    .collect::<Vec<_>>()
    .chunks(VERTEX_COUNT)
    .map(|row| row.to_vec())
    .collect();

let d: Vec<Vec<u32>> = c
    .iter()
    .map(|row| row.iter().map(|i| (*i > 0) as
    ↪  u32).collect())
    .collect();

let h: Vec<Vec<u32>> = d
    .iter()
    .enumerate()
    .map(|(i, row)| {
        row.iter()
            .enumerate()
            .map(|(j, _)| (d[i][j] != d[j][i]) as u32)
            .collect()
    })
    .collect();

let mut weights: Vec<Vec<Option<u32>>> =
↪  vec![vec![Some(0); VERTEX_COUNT]; VERTEX_COUNT];
for i in 0..VERTEX_COUNT {
    for j in (i + 1)..VERTEX_COUNT {
        let weight = c[i][j] * (d[i][j] + h[i][j] * (i <
        ↪  j) as u32);
        weights[i][j] = if weight != 0 { Some(weight) }
        ↪  else { None };
        weights[j][i] = if weight != 0 { Some(weight) }
        ↪  else { None };
    }
}
```

```
        Graph(weights)
    }

    pub fn kruskal_step(edges: &mut VecDeque<(usize, usize,
    ↪  u32)>, step: &mut KruskalStep) -> bool {
        if step.tree.len() == step.uf.rank.len() - 1 {
            println!("MST built.");
            step.current = None;
            return false;
        }

        if let Some(current) = step.current {
            if step.uf.find(current.0) == step.uf.find(current.1)
            ↪  {
                println!("{:?} would create a cycle", current);
            } else {
                println!("{:?} added to MST.", current);
                step.uf.union(current.0, current.1);
                step.tree.push(current);
            }
            step.current = edges.pop_front();
        } else if let Some(next) = edges.pop_front() {
            step.current = Some(next);
        } else {
            return false;
        }
        true
    }
}

pub struct UnionFind {
    parent: Vec<usize>,
    rank: Vec<usize>,
}

impl UnionFind {
    pub fn new(n: usize) -> Self {
        UnionFind {
            parent: (0..n).collect(),
            rank: vec![0; n],
        }
    }

    pub fn find(&mut self, x: usize) -> usize {
        if self.parent[x] != x {
            self.parent[x] = self.find(self.parent[x]);
        }
        self.parent[x]
    }
```

```
    pub fn union(&mut self, x: usize, y: usize) -> bool {
        let x_root = self.find(x);
        let y_root = self.find(y);

        if x_root == y_root {
            return false;
        }

        match self.rank[x_root].cmp(&self.rank[y_root]) {
            std::cmp::Ordering::Less => {
                self.parent[x_root] = y_root;
            }
            std::cmp::Ordering::Greater => {
                self.parent[y_root] = x_root;
            }
            std::cmp::Ordering::Equal => {
                self.parent[y_root] = x_root;
                self.rank[x_root] += 1;
            }
        }

        true
    }
}

pub struct KruskalStep {
    pub current: Option<(usize, usize, u32)>,
    pub tree: Vec<(usize, usize, u32)>,
    uf: UnionFind,
}

impl KruskalStep {
    pub fn new(vertex_count: usize) -> Self {
        KruskalStep {
            current: None,
            tree: Vec::with_capacity(vertex_count - 1),
            uf: UnionFind::new(vertex_count),
        }
    }

    pub fn weight_sum(&self) -> usize {
        self.tree.iter().map(|(_, _, w)| *w as usize).sum()
    }
}
```

Файл 2: graph.rs

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

use std::collections::VecDeque;
```

```rust
use graph::{DEFAULT_ROWS, Graph, KruskalStep, VERTEX_COUNT};
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 OVERLAY_FONT_SIZE: i32 = 24;

const K: f32 = 1.0 - 2.0 * 0.01 - 8.0 * 0.005 - 0.05;

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

fn draw_all_vertices(d: &mut RaylibDrawHandle, font: &Font, rows:
↪  &[usize]) -> 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,
            Color::BLACK,
        );
    });

    vertex_coords
}

fn draw_all_edges(
    d: &mut RaylibDrawHandle,
    graph: &Graph,
    vertex_coords: &[VertexPos],
    font: &Font,
    directed: bool,
    hide_edges: bool,
    step: &KruskalStep,
) {
    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 let Some(weight) = graph.0[i][j] {
                let step_current =
                 ↪  step.current.unwrap_or((usize::MAX,
                 ↪  usize::MAX, 0));
                let color = if step_current.0 == i &&
                 ↪  step_current.1 == j {
                    Color::BLUE
                } else if step
                    .tree
                    .iter()
                    .any(|(row, column, _)| *row == i && *column
                     ↪  == j)
                {
                    Color::RED
                } else if hide_edges {
                    Color::WHITE.alpha(0.0)
                } else {
```

```
                    Color::BLACK
                };
                let mut angled = false;

                if i == j {
                    draw::draw_looping_edge(d, origin.v, color);
                } else if (graph.0[j][i].is_some() && 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);
                    angled = true;
                } else {
                    draw::draw_straight_edge(d, origin.v,
                    ↪  destination.v, directed, color);
                }

                if i < j {
                    draw::draw_edge_weight(
                        d,
                        origin.v,
                        destination.v,
                        &weight.to_string(),
                        font,
                        color,
                        angled,
                    );
                }
            }
        }
    }
}

fn draw_controls(d: &mut RaylibDrawHandle, font: &Font,
↪  hide_edges: bool) {
    draw::draw_text(
        d,
        font,
        "<F3> Hide edges",
        Vector2 {
            x: 0.12 * WIN_WIDTH as f32,
            y: 0.01 * WIN_HEIGHT as f32,
        },
```

```
            OVERLAY_FONT_SIZE as f32,
            if hide_edges { Color::RED } else { Color::BLACK },
        );
        draw::draw_text(
            d,
            font,
            "<Space> Step",
            Vector2 {
                x: 0.1 * WIN_WIDTH as f32,
                y: 0.96 * WIN_HEIGHT as f32,
            },
            OVERLAY_FONT_SIZE as f32,
            Color::BLACK,
        );
    }

    fn main() {
        let (mut rl, thread) = raylib::init()
            .size(WIN_WIDTH, WIN_HEIGHT)
            .log_level(TraceLogLevel::LOG_WARNING)
            .title("ASD Lab 2.6")
            .build();

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

        let matrix = Graph::generate(K);
        let mut edges: Vec<(usize, usize, u32)> = matrix
            .0
            .iter()
            .enumerate()
            .flat_map(|(i, row)| {
                row.iter()
                    .enumerate()
                    .filter_map(move |(j, &cell)| cell.map(|w| (i, j,
                    ↪  w)))
            })
            .filter(|(i, j, _)| *i < *j)
            .collect();
        edges.sort_by_key(|(_, _, w)| *w);

        let mut edges_deque = VecDeque::from(edges);
        let mut step = KruskalStep::new(VERTEX_COUNT);

        let mut hide_edges = false;

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

        while !rl.window_should_close() {
            if rl.is_key_pressed(KeyboardKey::KEY_SPACE) {
                Graph::kruskal_step(&mut edges_deque, &mut step);
```

```
        } else if rl.is_key_pressed(KeyboardKey::KEY_F3) {
            hide_edges = !hide_edges;
        }
        let mut d = rl.begin_drawing(&thread);

        d.clear_background(Color::WHITE);
        let vertex_pos = draw_all_vertices(&mut d, &font,
        ↪  DEFAULT_ROWS);
        draw_all_edges(
            &mut d,
            &matrix,
            &vertex_pos,
            &font,
            false,
            hide_edges,
            &step,
        );

        draw_controls(&mut d, &font, hide_edges);
        draw::draw_text(
            &mut d,
            &font,
            &(String::from("Weight sum: ") +
            ↪  &step.weight_sum().to_string()),
            Vector2 {
                x: 0.8 * WIN_WIDTH as f32,
                y: 0.01 * WIN_HEIGHT as f32,
            },
            OVERLAY_FONT_SIZE as f32,
            Color::BLUE,
        );
    }
}
```

Файл 3: main.rs

Вивід програми

Матриця ваг:

```
  0 166 197 148 inf inf 162 inf inf inf 171 inf
166   0 inf 179 inf inf inf 150 inf 136 153 188
197 inf   0 inf inf 180 154 141 inf 132 138 inf
148 179 inf   0 inf 198 inf 199 164 162 inf 167
inf inf inf inf   0 143 190 inf 125 inf 187 126
inf inf 180 198 143   0 inf 182 inf 157 inf 125
162 inf 154 inf 190 inf   0 inf 118 inf 176 inf
inf 150 141 199 inf 182 inf   0 inf 153 inf inf
inf inf inf 164 125 inf 118 inf   0 inf 187 148
inf 136 132 162 inf 157 inf 153 inf   0 inf 114
171 153 138 inf 187 inf 176 inf 187 inf   0 159
inf 188 inf 167 126 125 inf inf 148 114 159   0
```

Сума ваг мінімального кістяка — 1465.

Зображення

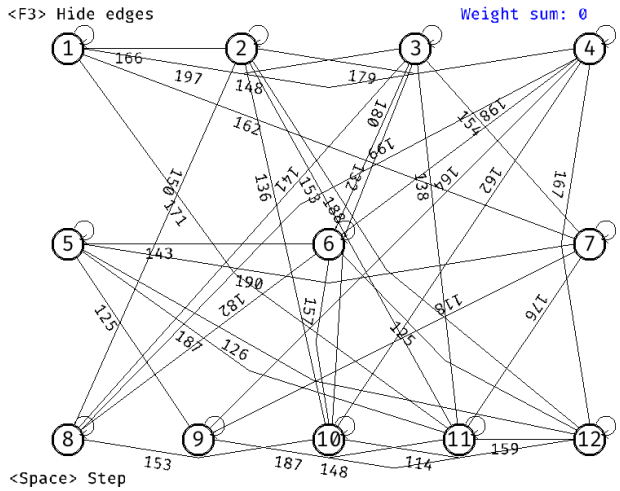

Рис. 1: Граф на початку програми

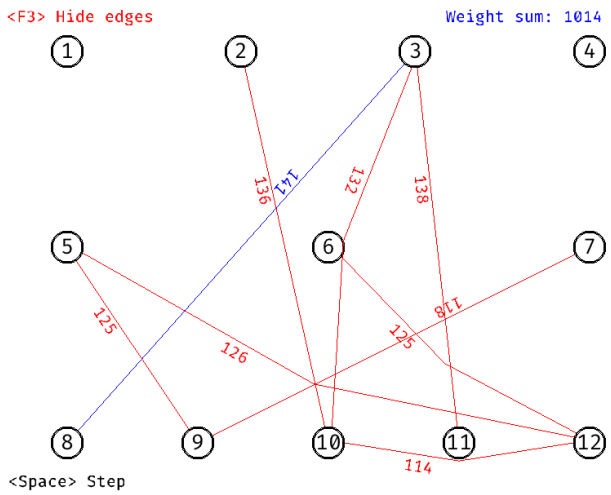

Рис. 2: Граф під час побудови кістяка (ребра приховані)

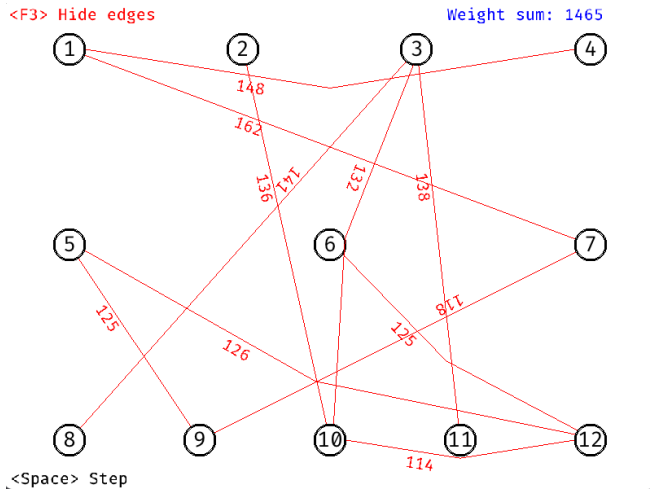

Рис. 3: Побудований кістяк

Висновок

Модифікував представлення графа у програмі для зберігання ваг ребер. Реалізував алгоритм Краскала для покрокової побудови мінімального кістяка ненапрямленого графа. Використав структуру disjoint set union (так званий `UnionFind`) для запобігання утворенню циклів.