МІНІСТЕРСТВО ОСВІТИ ТА НАУКИ УКРАЇНИ
КИЇВСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ ІМЕНІ ТАРАСА ШЕВЧЕНКА
ФАКУЛЬТЕТ КОМП'ЮТЕРНИХ НАУК ТА КІБЕРНЕТИКИ
КАФЕДРА ОБЧИСЛЮВАЛЬНОЇ МАТЕМАТИКИ

Звіт до лабораторної роботи №3 на тему «Мурашиний алгоритм»

Керівник групи
Скибицький Нікіта

Київ — 2019

У виконанні роботи брали участь:

- Основні учасники:
  - Скибицький Нікіта
  - Сергієнко Тетяна
  - Тихонравова Юлія
  - Ковальчук Віктор
  - Кузьмінова Катерина
  - Антипова Аліса
- Також допомагали:
  - Пушкін Денис
  - Єрмаков Артур
  - Бельо Андрій
  - Гронь Ілля

Зміст

1 Постановка задачі

Задано орієнтований граф $G = (V, E)$. Ребро $e_i \in E$ графа характеризуються довжиною ℓ_i. Задано початкову вершину $s \in V$ і цільову (кінцеву, фінальну) вершину $f \in V$.

Необхідно знайти найкоротший шлях із s до f.

2 Неформальний опис алгоритму

Розглянемо популяцію з N мурах, які протягом M ітерацій намагаються знайти найкоротший шлях (наприклад, шлях доставки листя до мурашника).

Уявімо, що на кожній ітерації кожна мурашка проходить якийсь шлях, залишаючи на своєму шляху феромени, і керуючись вже наявними із попередніх ітерацій феромонами для вибору шляху.

3 Код програмного продукнту

3.1 Представлення графу у пам'яті програми

```
A, B, C, D, E, F, G = 0, 1, 2, 3, 4, 5, 6

START, END = A, G

g = {
```

```
        A: {B: Edge(2), C: Edge(3), D: Edge(6)},
        B: {E: Edge(4), F: Edge(5)},
        C: {E: Edge(2), F: Edge(3)},
        D: {E: Edge(5), F: Edge(2)},
        E: {G: Edge(2)},
        F: {G: Edge(1)},
        G: {},
```

де ребро моделюється наступним класом:

```
class Edge:
    def __init__(self, length):
        self.length, self.feroment, self.delta = length, length, 0

    def __repr__(self):
        return f'Edge(length={self.length}, feroment={self.feroment}, delta={self.delta})'
```

Як бачимо, у ребра є довжина (`length`), на ньому є певна інтенсивність феромені (`feroment`), і необхідне оновлення феромент (`delta`).

3.2 Клас, який моделює мурашку

3.2.1 Конструктор

```
class Ant:
    def __init__(self, start, target):
        self.tabu_list = []
        self.vertice = start
        self.target = target
        self.path_length = 0
        self.path = []
        self.alive = True
```

Як ми бачимо з коду, кожна мурашка пам'ятає список вже пройдених вершин (`tabu_list`), знає у якій вершині вона зараз знаходиться (`vertice`), знає, куди їй треба йти (`target`), підтримує у пам'яті загальну довжину пройденого шляху (`path_length`), сам цей шлях (`path`) і знає, чи вона "жива" (`alive`). *Мурашка вважається "мертвою", якщо вона не змогла дістатися мурашника.*

3.2.2 Метод для вибору напрямку кроку

```
    def step(self):
        pre_probability = {
            to : (g[self.vertice][to].feroment + \
                1 / g[self.vertice][to].length)
            for to in (set(g[self.vertice].keys()) - set(self.tabu_list))
        }
```

```
        if not pre_probability:
            self.alive = False
            return

        sum_pre_probability = sum(pre_probability.values())

        probability = {
            to : pre_probability[to] / sum_pre_probability
            for to in pre_probability
        }

        choose_from, choice_probability = [], []

        for t in probability:
            choose_from.append(t)
            choice_probability.append(probability[t])

        step_to = choice(choose_from, p=choice_probability)

        self.path_length += g[self.vertice][step_to].length
        self.path.append((self.vertice, step_to))
        self.tabu_list.append(self.vertice)
        self.vertice = step_to
```

3.2.3 Метод для пошуку цілісного шляху:

```
    def solve(self):
        while self.vertice != self.target and self.alive:
            self.step()

        if self.alive:
            for f, t in self.path:
                g[f][t].delta += .1 * g[f][t].length / self.path_length**2
```

3.3 Програма-драйвер

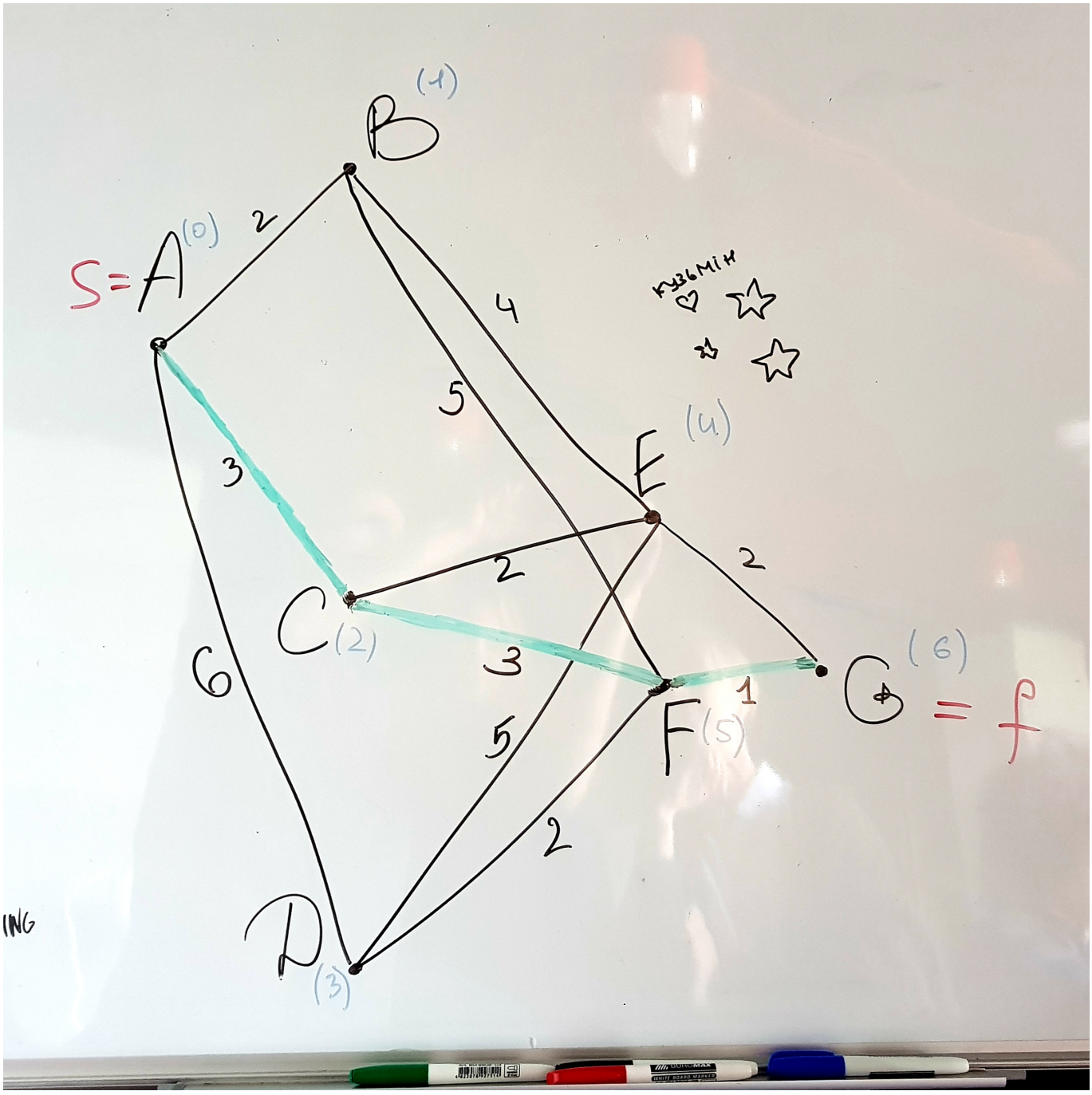
```
    for i in range(m):
        for j in range(n):
            print(f'{i:0>3} {j:0>3}')
            ant = Ant(start=START, target=END)
            ant.solve()

        for f in g:
            for t in g[f]:
                g[f][t].feroment, g[f][t].delta = \
                    .7 * g[f][t].feroment + g[f][t].delta, 0
```

4 Тестування програмного продукту

4.1 Граф

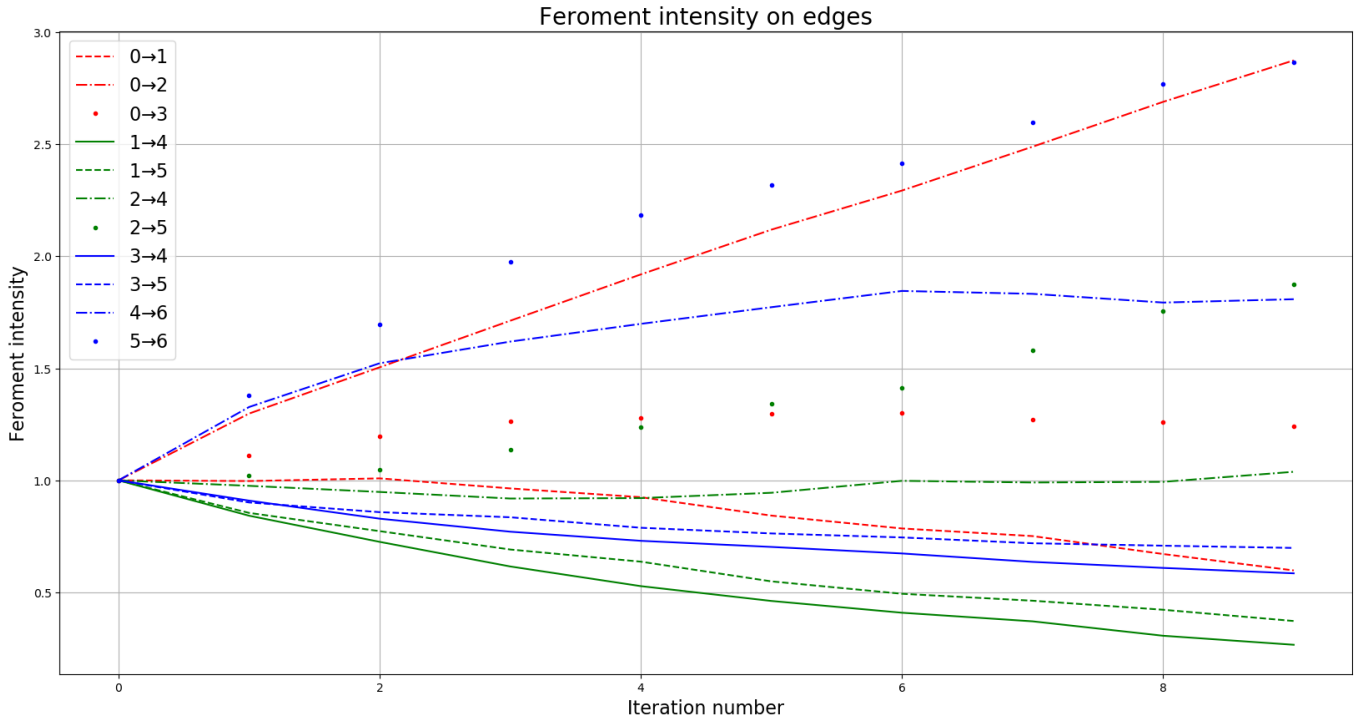
Ось так виглядає наш тестовий граф:



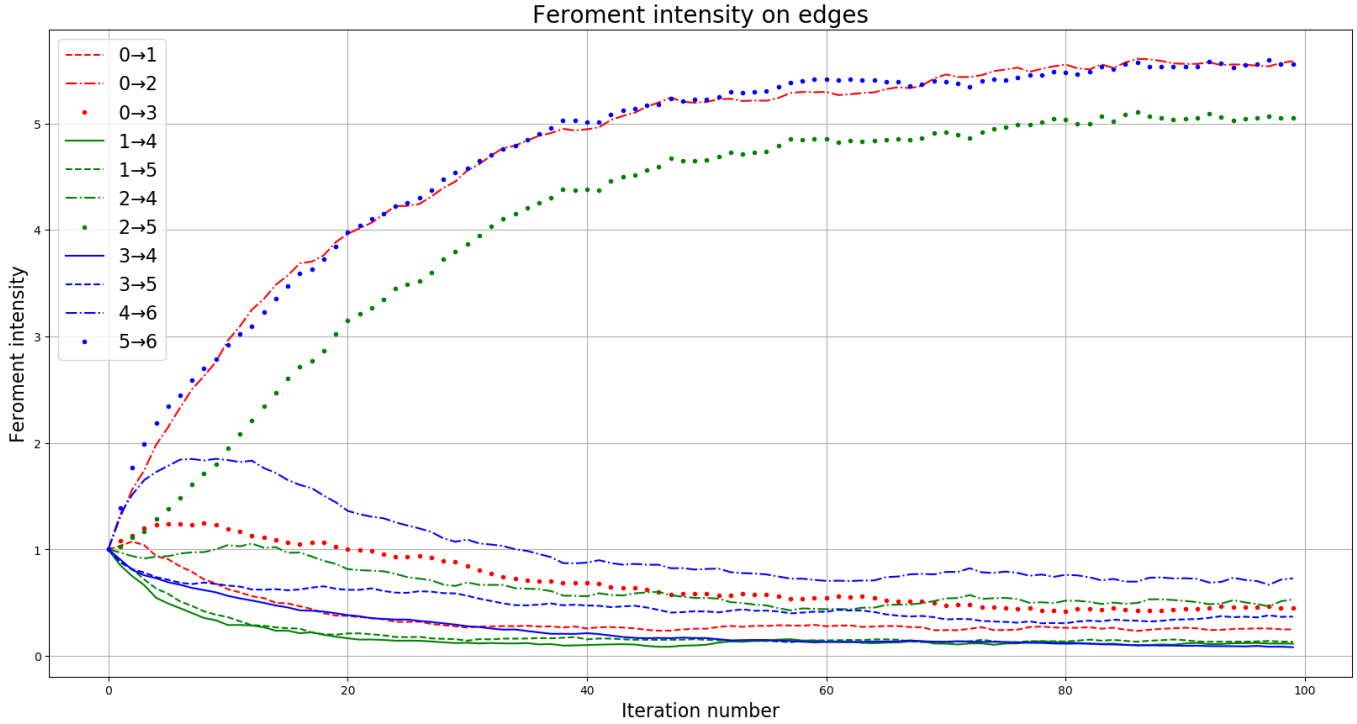
4.2 Графіки

Інтенсивність феромонів від ітерації:

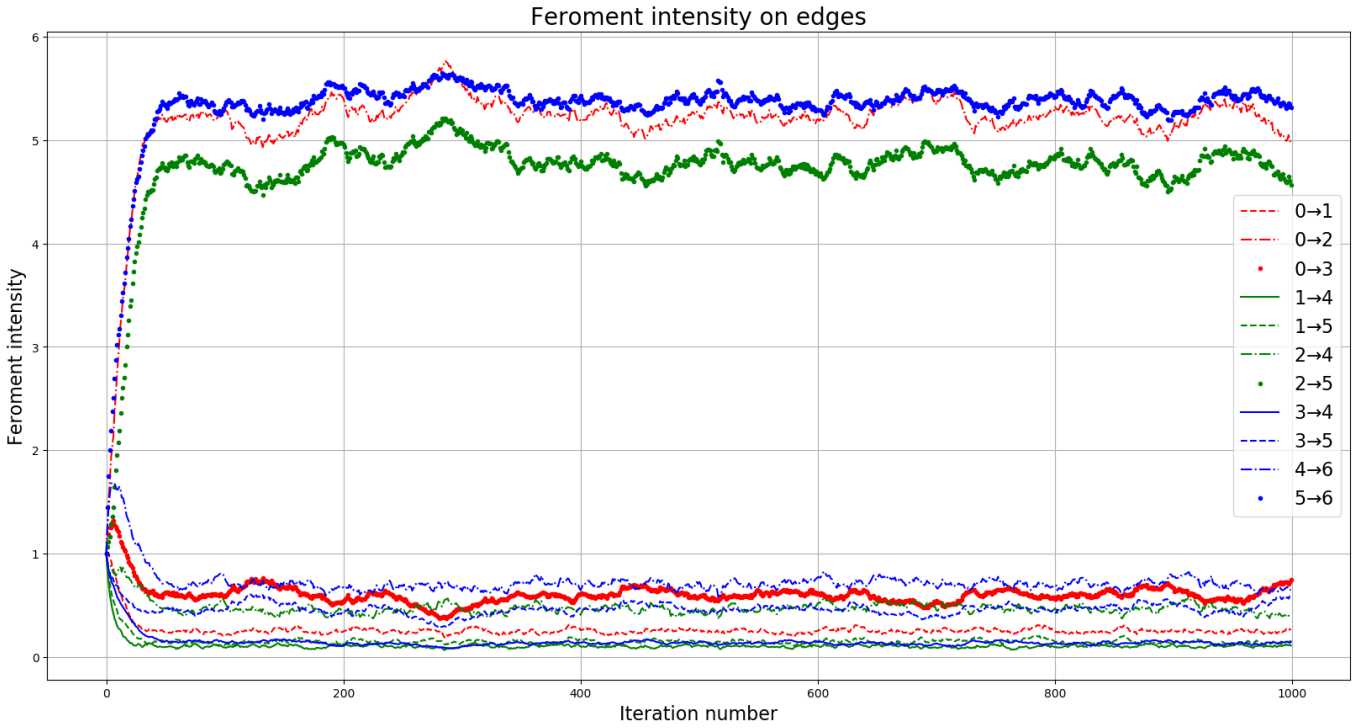
4.2.1 10 ітерацій



4.2.2 100 ітерацій



4.2.3 1000 ітерацій



4.3 Швидкодія

Середній час виконання ітерації — **0.179** секунди, або **3** хвилини на **1000** ітерацій.

Зауважимо, що алгоритм багатоагентний і ідеально паралелиться, тому насправді нас цікавить час виконання однією мурахою однієї ітерації.

Нескладно зрозуміти, що час виконання однієї мурахо-ітерації мізерний, а саме **0.000179** секунди, тобто одна мурашка може виконати понад **5000** ітерацій за одну секунду.

На більшому графі ($|V| = 30$, $|E| = 100$) швидкодія передбачувано знизиться, але все одно складе принаймні **100** ітерацій на секунду.

Тобто, маючи **16** логічних процесорів (а саме стільки їх у моєму ноутбуці) і розпаралеливши алгоритм можна розв'язати у **50** разів складнішу (і вже цілком реалістичну) задачу десь за **30** хвилин.

Непоганий результат, враховуючи що сама постановка задачі NP-повна.