Министерство науки и высшего образования Российской Федерации
Федеральное государственное бюджетное образовательное учреждение высшего образования
«Московский государственный технический университет имени Н.Э. Баумана
(национальный исследовательский университет)»
(МГТУ им. Н.Э. Баумана)

ФАКУЛЬТЕТ «Информатика и системы управления»

КАФЕДРА «Программное обеспечение ЭВМ и информационные технологии»

Отчет по лабораторной работе №6 по курсу "Анализ алгоритмов"

Тема Муравьиный алгоритм

Студент Романов С.К.

Группа ИУ7-55Б

Оценка (баллы)

Преподаватели Волкова Л.Л., Строганов Ю.В.

Москва — 2023 г.

Оглавление

ВВЕДЕНИЕ

Муравьиный алгоритм – один из эффективных полиномиальных алгоритмов для нахождения приближённых решений задачи коммивояжёра, а также решения аналогичных задач поиска маршрутов на графах. Суть подхода заключается в анализе и использовании модели поведения муравьёв, ищущих пути от колонии к источнику питания, и представляет собой метаэвристическую оптимизацию.

Цель лабораторной работы

Целью данной лабораторной работы является изучение муравьиного алгоритма и приобретение навыков параметризации методов на примере муравьиного алгоритма.

Задачи лабораторной работы:

В рамках выполнения работы необходимо решить следующие задачи:

- решить задачу поиска кратчайшего пути при помощи алгоритма полного перебора и муравьиного алгоритма;
- замерить и сравнить время выполнения алгоритмов;
- протестировать муравьиный алгоритм на разных переменных;
- сделать выводы на основе проделанной работы.

В ходе работы будут затронуты следующие темы:

- применение муравьиного алгоритма для решения задачи поиска кратчайшего пути;
- параметризация муравьиного алгоритма;
- алгоритм полного перебора;
- обработка графа.

1 Аналитическая часть

1.1 Обзор алгоритмов поиска

В данной работе были использованы два алгоритма поиска — алгоритм полного перебора и муравьиный алгоритм.

1.1.1 Алгоритм полного перебора

Алгоритм полного перебора — это алгоритм, предусматривающий перебор всех вариантов решения задачи. Он используется для поиска оптимального решения на небольших наборах данных.

Алгоритм полного перебора для задачи поиска кратчайшего пути:

Пронумеруем все города от 1 до n. Базовому городу присвоим номер n. Каждый тур по городам однозначно соответствует перестановке целых чисел $1, 2, ..., n-1$.

Algorithm 1 Алгоритм полного перебора для задачи поиска кратчайшего пути

```
1: Вход: матрица расстояний между городами
2: Выход: минимальное расстояние
3: Переменные: n — количество городов, A — матрица расстояний, x —
   перестановка целых чисел, min — минимальное расстояние
4: min ← ∞
5: for i ← 1 to n! do
6:     x ← перестановка i
7:     вычислить расстояние по перестановке x
8:     if расстояние < min then
9:         min ← расстояние
10:    end if
11: end for
12: возврат min
```

Сложность алгоритма полного перебора:

Временная сложность алгоритма полного перебора для поиска кратчайшего пути определяется как $O(n!)$.

1.1.2 Муравьиный алгоритм

Муравьиный алгоритм — один из эффективных алгоритмов для нахождения приближённых решений задачи коммивояжёра, а также решения аналогичных задач поиска маршрутов на графах. Он базируется на модели поведения муравьёв, ищущих пути от колонии к источнику питания.

Алгоритм моделирует общее поведение муравьёв при поиске пути к источнику питания. Он имитирует поведение муравьёв, изменяя вероятность перехода из одной вершины в другую в зависимости от функции оценки и влияния окружающей среды.

Муравьиный алгоритм для задачи поиска кратчайшего пути:

Моделирование поведения муравьев связано с распределением феромона на тропе — ребре графа в задаче коммивояжера. При этом вероятность включения ребра в маршрут отдельного муравья пропорциональна количеству феромона на этом ребре, а количество откладываемого феромона пропорционально длине маршрута. Чем короче маршрут, тем больше феромона будет отложено на его ребрах, следовательно, большее количество муравьев будет включать его в синтез собственных маршрутов. Моделирование такого подхода, использующего только положительную обратную связь, приводит к преждевременной сходимости — большинство муравьев двигается по локально оптимальному маршруту. Избежать, этого можно, моделируя отрицательную обратную связь в виде испарения феромона. При этом если феромон испаряется быстро, то это приводит к потере памяти колонии и забыванию хороших решений, с другой стороны, большое время испарения может привести к получению устойчивого локально оптимального решения. Теперь, с учетом особенностей задачи коммивояжера, мы можем описать локальные правила поведения муравьев при выборе пути.

- муравьи имеют собственную «память». Поскольку каждый город может быть посещеи только один раз, у каждого муравья есть список уже посещенных городов — список запретов. Обозначим через $J_{i,k}$ список городов, которые необходимо посетить муравью k, находящемуся в городе i;

- муравьи обладают «зрением» — видимость есть эвристическое желание посетить город j, если муравей находится в городе i. Будем считать, что видимость обратно пропорциональна расстоянию между городами i и j — D_{ij}

$$\eta_{ij} = \frac{1}{D_{ij}} \tag{1.1}$$

- муравьи обладают «обонянием» — они могут улавливать след феромона, подтверждающий желание посетить город j из города i, на основании опыта других муравьев. Количество феромона на ребре (i,j) в момент времени t обозначим через $\tau_{ij}(t)$.

На этом основании мы можем сформулировать вероятностно-пропорционально
правило 1.2, определяющее вероятность перехода k-ого муравья из города i в город j:

$$\begin{cases} P_{i,j,k}(t) = \frac{[\tau_{ij}(t)]^{\alpha} * [\eta_{ij}]^{\beta}}{\sum_{l \in J_{i,k}} [\tau_{il}(t)]^{\alpha} * [\eta_{il}]^{\beta}}, & j \in J_{i,k}; \\ P_{i,j,k}(t) = 0, & j \notin J_{i,k}, \end{cases} \tag{1.2}$$

Вывод

В данной секции были проанализированы два алгоритма поиска кратчайшего пути − алгоритм полного перебора и муравьиный алгоритм. Алгоритм полного перебора имеет временную сложность $O(n!)$, а временная сложность муравьиного алгоритма зависит от количества итераций и выбранных параметров.

Алгоритм полного перебора используется для поиска оптимального решения на небольших наборах данных, а муравьиный алгоритм позволяет находить приближённые решения для больших наборов данных.

2 Конструкторская часть

В данном разделе представлены схемы муравьиного алгоритма и алгоритма полного перебора для решения задачи коммивояжера.

2.1 Схема алгоритма полного перебора

На рисунке 2.1 представлена схема алгоритма полного перебора решения задачи Коммивояжера. На рисунке 2.2 представлена схема нахождения всех перестановок. На рисунках 2.3 и 2.4 представлена схема реализации муравьиного алгоритма для решения задачи коммивояжера.

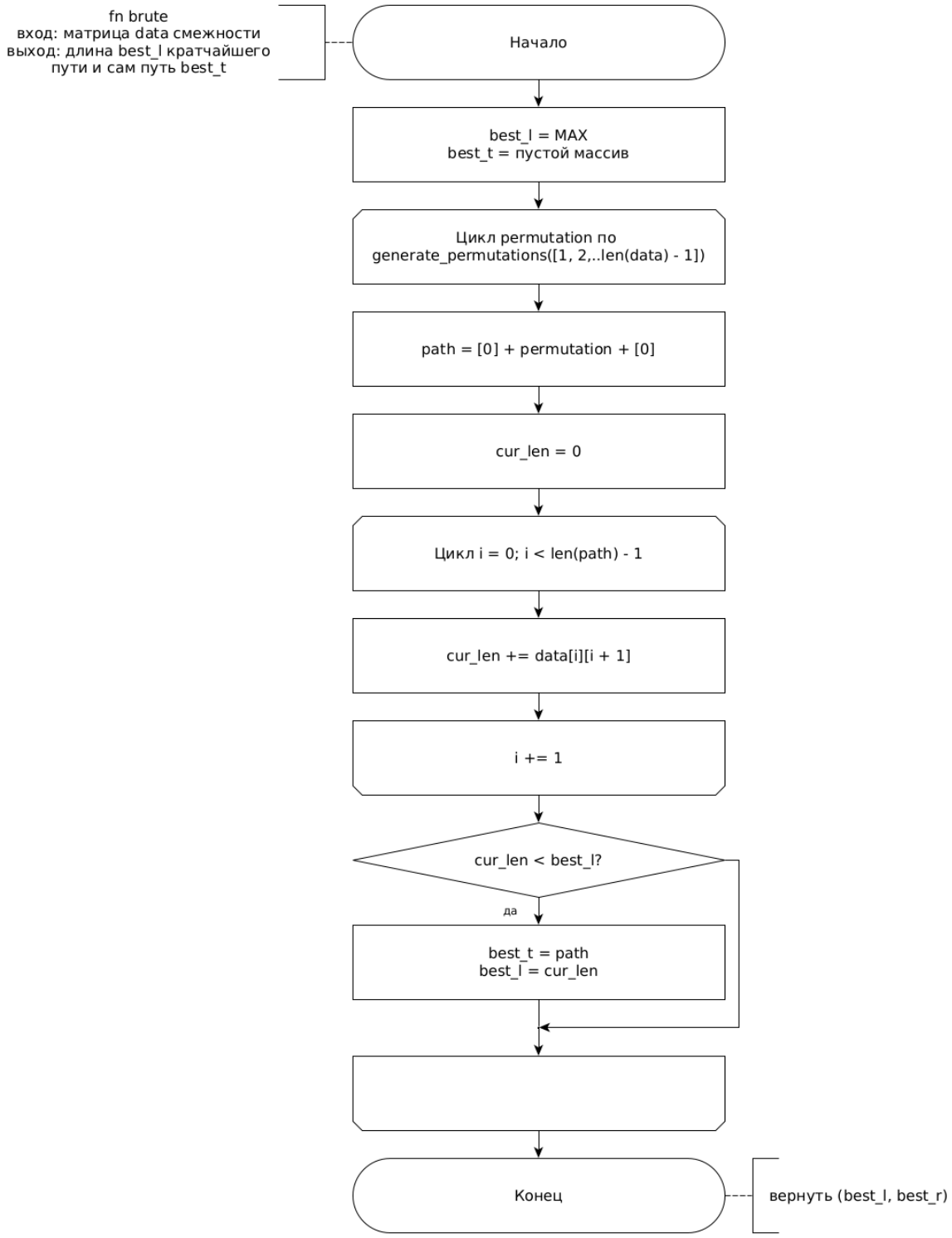

Рис. 2.1: Схема алгоритма полного перебора.

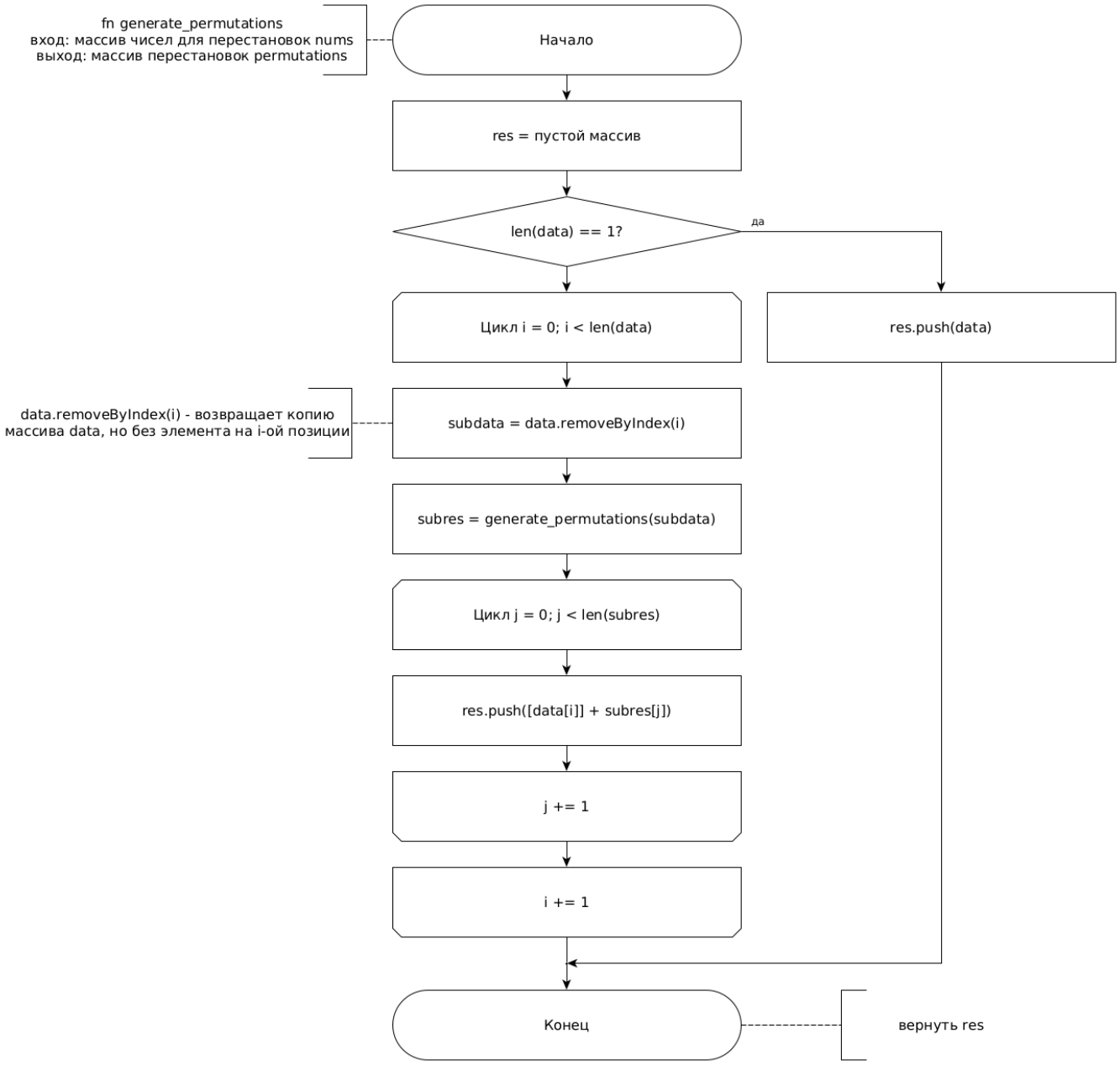

Рис. 2.2: Схема алгоритма нахождения всех перестановок в графе.

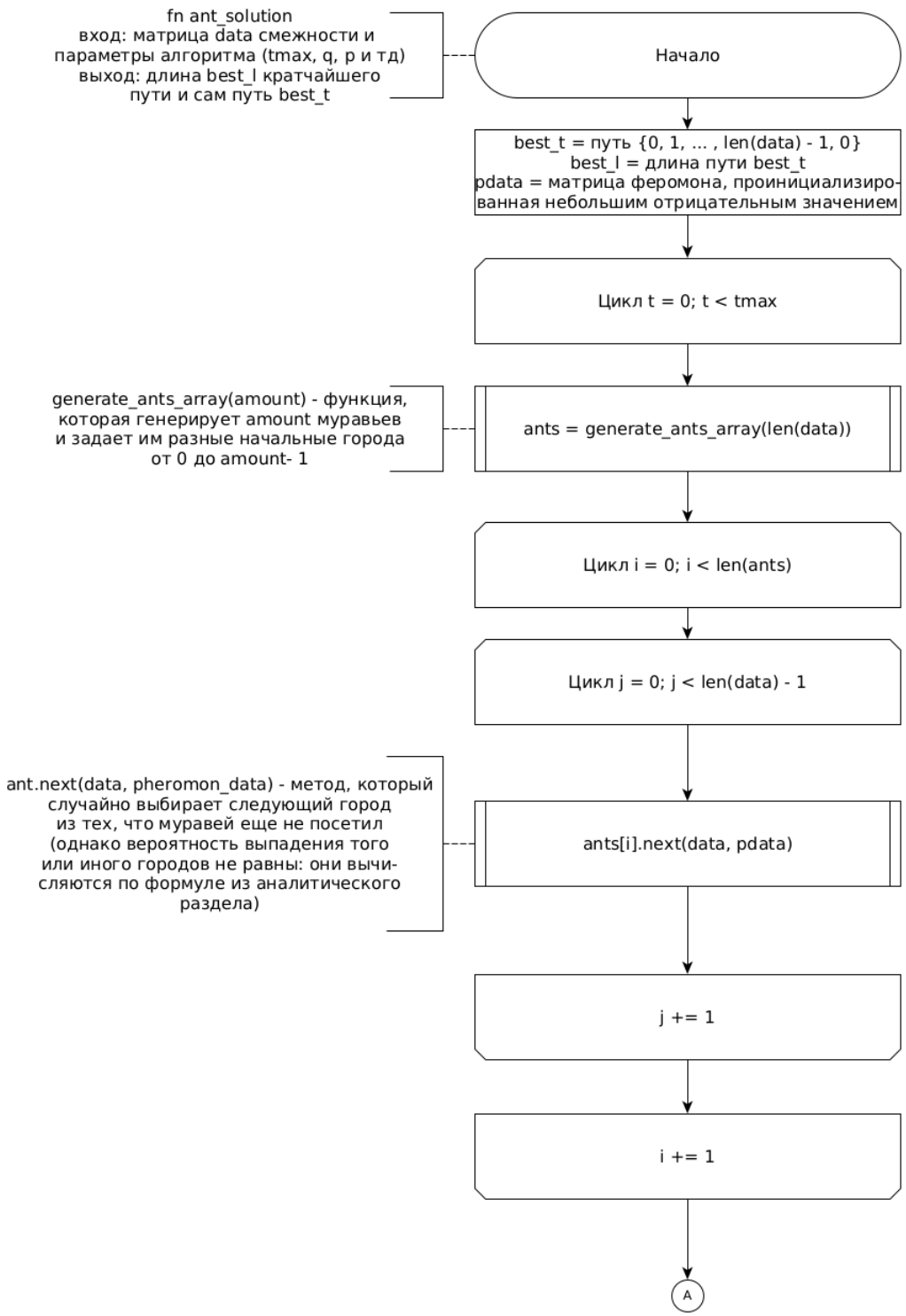

Рис. 2.3: Схема муравьиного алгоритма.

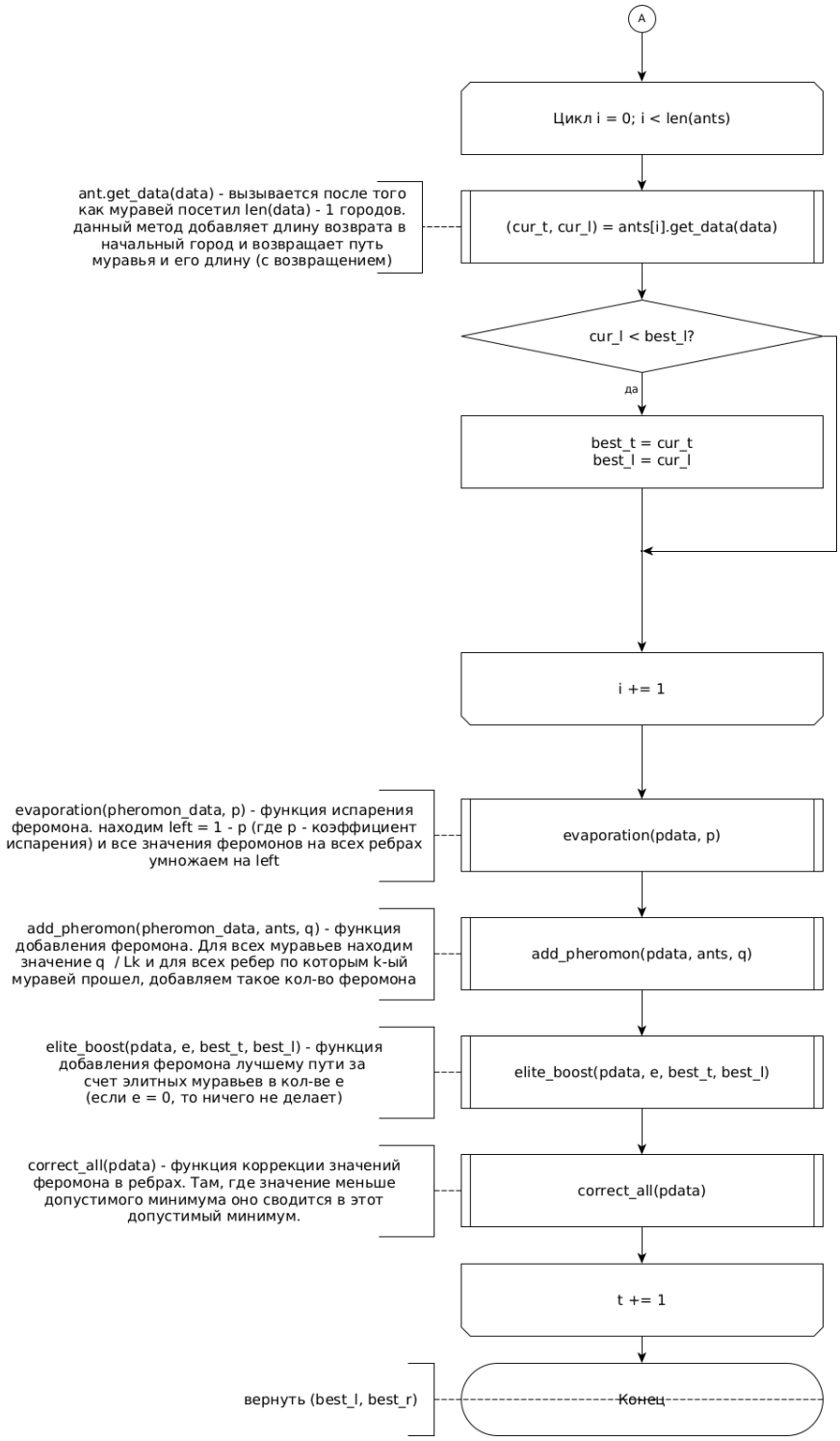

Рис. 2.4: Схема муравьиного алгоритма. Продолжение.

Вывод

Были представлены схемы алгоритмов полного перебора и муравьиного для решения задачи коммивояжера. Также дополнительно была показана схема алгоритма поиска всех перестановок.

3 Технологическая часть

В данном разделе приведены требования к программному обеспечению, средства реализации и сами реализации алгоритмов.

3.1 Требования к программному обеспечению

К программе предъявляется ряд условий:

– на вход подаётся матрица смежности графа;

– на выходе программа выдаёт минимальную длину и путь, на котором данная длина достигнута;

– ПО должно замерять время работы алгоритмов.

3.2 Средства реализации

Для реализации данной лабораторной работы необходимо установить следующее программное обеспечение:

- Rust Programming Language v1.64.0 - язык программирования
- Criterion.rs v0.4.0 - Средство визуализации данных
- LaTeX - система документооборота

3.3 Реализация алгоритмов

В следующих листингах представлены следующие алгоритмы: В листинге 3.1 представлена реализация алгоритма полного перебора, в листинге 3.2 представлена реализация муравьиного алгоритма. В листингах 3.3 и 3.4 представлены конфигурационная структура и структура муравья с методами соотвественно.

```rust
use super::Cost;
use itertools::Itertools;

pub struct BruteSolver<'a>
{
    data: &'a [Vec<Cost>],
    start: usize,
    end: usize,
}

impl<'a> BruteSolver<'a>
{
    pub const fn new(data: &'a [Vec<Cost>], start: usize, end: usize) -> Self
    {
        Self { data, start, end }
    }

    pub fn solve(&self) -> (Cost, Vec<usize>)
    {
        let (mut best_l, mut best_t) = (Cost::MAX, Vec::new());

        let left = vec![
            (0..self.start).collect_vec(),
            ((self.start + 1)..self.end).collect_vec(),
            ((self.end + 1)..self.data.len()).collect_vec(),
        ]
        .concat();
        for length in 0..=(self.data.len() - 2) {
            for permutation in left.iter().permutations(length) {
                let route = vec![
                    vec![self.start],
                    permutation.into_iter().copied().collect(),
                    vec![self.end],
                ]
                .concat();
                let l = self.compute_dist(&route);
                if l < best_l {
                    best_l = l;
                    best_t = route;
                }
            }
        }
        (best_l, best_t)
    }

    fn compute_dist(&self, t: &[usize]) -> Cost
    {
        match t.len() {
```

```
            0..=1 => 0 as Cost,
            2 => self.data[t[0]][t[1]],
            _ => t.windows(2).fold(0 as Cost, |acc, el| {
                acc.saturating_add(self.data[el[0]][el[1]])
            }),
        }
    }
}
```

Листинг 3.1: Реализация полного перебора.

```
use super::Cost;
use rand::prelude::*;

mod config;
pub use config::Config;

mod ant;
use ant::Ant;

pub struct AntSolver<'a>
{
    data: &'a [Vec<Cost>],
    ndata: Vec<Vec<f64>>,
    q: f64,
    config: Config,
}

impl<'a> AntSolver<'a>
{
    pub fn new(data: &'a [Vec<Cost>], config: Config) -> Self
    {
        let ndata = data
            .iter()
            .map(|row| row.iter().map(|&e| 1_f64 / e as f64).collect())
            .collect();
        let q = Self::compute_q(data);
        Self {
            data,
            ndata,
            q,
            config,
        }
    }

    fn compute_q(data: &[Vec<Cost>]) -> f64
    {
        // sum of all
        let q = data
```

```
            .iter()
            .fold(0 as Cost, |acc, row| acc + row.iter().sum::<Cost>()) as f64;
        q / data.len() as f64
    }

    pub fn solve(&self) -> (Cost, Vec<usize>)
    {
        let (mut rng, mut pheromon_data, mut best_t, mut best_l) = self.init_params();
        for _ in 0..self.config.tmax {
            let mut ants = self.generate_ants();
            // Run ants
            for a in &mut ants {
                a.walk(
                    self.data,
                    &self.ndata,
                    &pheromon_data,
                    self.config.alpha,
                    self.config.beta,
                    &mut rng,
                );
            }

            // Find best T* and L* after day
            let best_data = ants
                .iter()
                .min_by(|a, b| a.data().0.cmp(&b.data().0))
                .unwrap()
                .data();
            if best_data.0 < best_l {
                best_l = best_data.0;
                best_t = best_data.1.to_vec();
            }

            // Update pheromon: evaporation, add_pheromon by ants,
            // correct_pheromon (min bound), elite_boost (add e * dt to best edges)
            self.evaporation(&mut pheromon_data);
            self.add_pheromon(&ants, &mut pheromon_data);
            self.correct_pheromon(&mut pheromon_data);
            self.elite_boost(&mut pheromon_data, &best_t, best_l);
        }

        (best_l, best_t)
    }

    fn init_params(&self) -> (ThreadRng, Vec<Vec<f64>>, Vec<usize>, Cost)
    {
        let rng = thread_rng();
        let pheromon_data =
```

```
            vec![vec![self.config.pheromon_start; self.data.len()]; self.data.len()];
        let best_t = Vec::new();
        let best_l = usize::MAX;
        (rng, pheromon_data, best_t, best_l)
    }

    fn generate_ants(&self) -> Vec<Ant>
    {
        let (m, data_len) = (self.config.m, self.data.len());
        // placing ants in all places works worse
        (0..m)
            .map(|_| Ant::new(data_len, self.config.start, self.config.end))
            .collect::<Vec<Ant>>()
    }

    fn add_pheromon(&self, ants: &[Ant], pdata: &mut [Vec<f64>])
    {
        let q = self.q;
        ants.iter().for_each(|ant| {
            let (l, route) = ant.data();
            let val = q / l as f64;
            for path in route.windows(2) {
                pdata[path[0]][path[1]] += val;
                pdata[path[1]][path[0]] += val;
            }
        });
    }

    fn correct_pheromon(&self, pdata: &mut [Vec<f64>])
    {
        let low_bound = self.config.pheromon_min;
        pdata.iter_mut().for_each(|row| {
            row.iter_mut().for_each(|v| {
                if *v < low_bound {
                    *v = low_bound;
                }
            });
        });
    }

    fn evaporation(&self, pdata: &mut [Vec<f64>])
    {
        let left = 1.0 - self.config.p;
        pdata
            .iter_mut()
            .for_each(|row| row.iter_mut().for_each(|v| *v *= left));
    }

```

```
135     fn elite_boost(&self, pdata: &mut [Vec<f64>], best_t: &[usize], best_l: Cost)
136     {
137         let val = self.config.e as f64 * self.q / best_l as f64;
138         best_t.windows(2).for_each(|win| {
139             pdata[win[0]][win[1]] += val;
140             pdata[win[1]][win[0]] += val;
141         });
142         let (first, last) = (best_t[0], best_t[best_t.len() - 1]);
143         pdata[first][last] += val;
144         pdata[last][first] += val;
145     }
146 }
```

Листинг 3.2: Реализация муравьиного алгоритма.

```
1  use serde_derive::Deserialize;
2
3  #[derive(Clone, Deserialize)]
4  pub struct Config
5  {
6      pub alpha: f64,
7      pub beta: f64,
8      pub e: usize, // number of elite ants
9      pub p: f64, // evaporation rate \in [0..1]
10     pub m: usize, // number of ants
11     pub tmax: usize,
12     pub pheromon_start: f64,
13     pub pheromon_min: f64,
14     pub start: usize,
15     pub end: usize,
16 }
```

Листинг 3.3: Конфигурационная структура.

```
1  use super::Cost;
2  use rand::prelude::*;
3
4  #[derive(Clone)]
5  pub struct Ant
6  {
7      pub route: Vec<usize>,
8      len: Cost,
9      left: Vec<usize>,
10     dest: usize,
11 }
12
13 impl Ant
14 {
15     pub fn new(cities_amount: usize, start: usize, dest: usize) -> Self
```

```
    {
        let left: Vec<usize> = (0..cities_amount).filter(|&e| e != start).collect();
        Self {
            route: vec![start],
            len: 0 as Cost,
            left,
            dest,
        }
    }

    pub fn walk(
        &mut self,
        d: &[Vec<Cost>],
        nd: &[Vec<f64>],
        pd: &[Vec<f64>],
        alpha: f64,
        beta: f64,
        rng: &mut ThreadRng,
    )
    {
        for _ in 0..self.left.len() {
            self.next(d, nd, pd, alpha, beta, rng);
            if self.route[self.route.len() - 1] == self.dest {
                break;
            }
        }
    }

    pub fn data(&self) -> (Cost, &[usize])
    {
        (self.len, &self.route)
    }

    fn next(
        &mut self,
        d: &[Vec<Cost>],
        nd: &[Vec<f64>],
        pd: &[Vec<f64>],
        alpha: f64,
        beta: f64,
        rng: &mut ThreadRng,
    )
    {
        let cur = self.route[self.route.len() - 1];
        let denominator = self.left.iter().fold(0.0, |acc, &e| {
            f64::powf(pd[cur][e], alpha).mul_add(f64::powf(nd[cur][e], beta), acc)
        });
        let mut pick: f64 = rng.gen();
```

```
        for (index, &j) in self.left.iter().enumerate() {
            let cur_prob = f64::powf(pd[cur][j], alpha) * f64::powf(nd[cur][j], beta);
            pick -= cur_prob / denominator;
            if pick < 0_f64 {
                self.pick(index, d[cur][j]);
                return;
            }
        }
        let index = self.left.len() - 1;
        self.pick(index, d[cur][self.left[index]]);
    }

    fn pick(&mut self, index: usize, diff: Cost)
    {
        self.route.push(self.left.remove(index));
        self.len = self.len.saturating_add(diff);
    }
}
```

Листинг 3.4: Структура муравья и её методы.

3.4 Тестирование функций.

В таблице 3.1 приведены тесты для функции, реализующей алгоритм для решения задачи коммивояжера. Тесты пройдены успешно.

Таблица 3.1: Тестирование функций

Матрица смежности	Ожидаемый наименьший путь
$\begin{pmatrix} 0 & 9 & 12 & 45 \\ 13 & 0 & 2 & 27 \\ 13 & 8 & 0 & 21 \\ 27 & 26 & 25 & 0 \end{pmatrix}$	32, [0, 1, 2, 3]
$\begin{pmatrix} 0 & 21 & -1 & 9 \\ 22 & 0 & 12 & 15 \\ -1 & 17 & 0 & -1 \\ 13 & 20 & -1 & 0 \end{pmatrix}$	9, [0, 3]
$\begin{pmatrix} 0 & 24 & 8 & 28 \\ 24 & 0 & 4 & 12 \\ 14 & 6 & 0 & 26 \\ 21 & 12 & 17 & 0 \end{pmatrix}$	26, [0, 2, 1, 3]

Вывод

Спроектированные алгоритмы были реализованы и протестированы.

4 Исследовательская часть

Ниже приведены технические характеристики устройства, на котором было проведено тестирование ПО:

- Операционная система: Arch Linux [1] 64-bit.
- Оперативная память: 16 Гб.
- Процессор: 11th Gen Intel® Core™ i5-11320H @ 3.20 ГГц[2].

4.1 Пример выполнения

На рисунке 4.1 приведен пример работы программы.

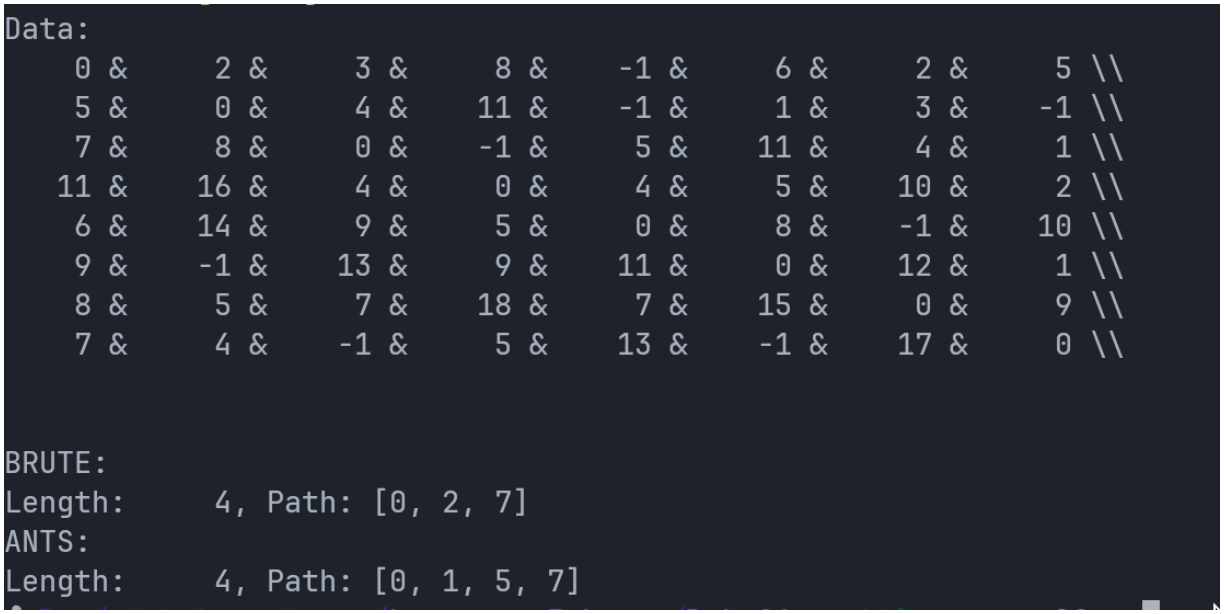

Рис. 4.1: Пример работы программы.

4.2 Время выполнения алгоритмов

Алгоритмы тестировались при помощи инструментов замера времени предоставляемых библиотекой Criterion.rs[3]. Пример функции по замеру времени приведен в листинге 4.1. Количество повторов регулируется тестирующей системой самостоятельно, однако ввиду трудоемкости вычислений, количество повторов было ограничено до 25.

```rust
fn dbscan_bench(c: &mut Criterion)
{
    let plot_config = PlotConfiguration::default().summary_scale(AxisScale::Linear);
    let mut config = read_config(constants::CONFIG_FILE);
    let mut group = c.benchmark_group("RIVERS");
    group.plot_config(plot_config);
    // group.sample_size(25);
    for size in constants::TIME_FROM..=constants::TIME_TO {
        let data = generate_data(
            size,
            constants::VALS_FROM,
            constants::VALS_TO,
            constants::RIVER_FROM,
            constants::RIVER_TO,
        );
        config.m = data.len();
        config.start = 0;
        config.end = config.m - 1;
        let brute = BruteSolver::new(&data, config.start, config.end);

        let ant = AntSolver::new(&data, config.clone());
        group.bench_with_input(BenchmarkId::new("Ant", size), &size, |b, _size| {
            b.iter(|| {
                black_box(ant.solve());
            });
        });
        group.bench_with_input(BenchmarkId::new("Brute", size), &size, |b, _size| {
            b.iter(|| {
                black_box(brute.solve());
            });
        });
    }

    group.finish();
}
```

Листинг 4.1: Пример функции замера времени

График, показывающий время работы последовательного и параллельного алгоритмов в зависимости от количества потоков

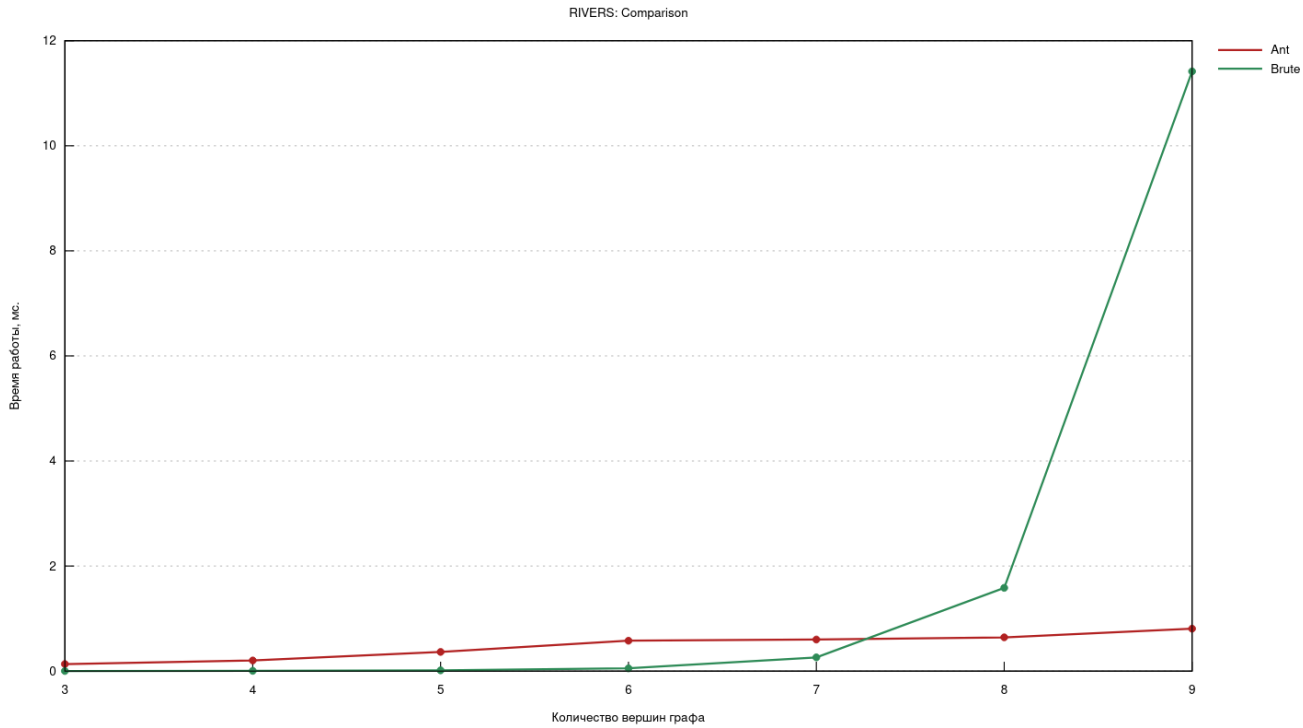

Рис. 4.2: Замеры времени работы

4.3 Автоматическая параметризация

В таблице 4.1 приведена выборка результатов параметризации для матрицы смежности размером 10x10. Количество дней принято равным 100. Полным перебором был посчитан оптимальный путь – он составил 130.

Таблица 4.1: Выборка из параметризации для матрицы размером $10x10$.

α	β	ρ	Длина	Разница	Путь
0.9	0.1	0	6	0	[0, 5, 9]
0.9	0.1	0.1	6	0	[0, 5, 9]
0.9	0.1	0.2	6	0	[0, 5, 9]
0.9	0.1	0.3	16	10	[0, 5, 6, 2, 9]
0.9	0.1	0.4	6	0	[0, 5, 9]
0.9	0.1	0.5	6	0	[0, 5, 9]
0.9	0.1	0.6	6	0	[0, 5, 9]
0.9	0.1	0.7	6	0	[0, 5, 9]
0.9	0.1	0.8	6	0	[0, 5, 9]
0.9	0.1	0.9	20	14	[0, 6, 2, 9]
0.9	0.1	1	6	0	[0, 5, 9]
1	0	0	18	12	[0, 1, 7, 9]
1	0	0.1	6	0	[0, 5, 9]
1	0	0.2	20	14	[0, 6, 2, 9]
1	0	0.3	6	0	[0, 5, 9]
1	0	0.4	6	0	[0, 5, 9]
1	0	0.5	21	15	[0, 1, 9]
1	0	0.6	18	12	[0, 5, 6, 9]
1	0	0.7	6	0	[0, 5, 9]
1	0	0.8	21	15	[0, 1, 9]

α	β	ρ	Длина	Разница	Путь
0.8	0.2	0.4	2100	0	[0, 7, 4, 9]
0.8	0.2	0.5	2100	0	[0, 7, 4, 9]
0.8	0.2	0.6	2127	27	[0, 6, 5, 9]
0.8	0.2	0.7	2100	0	[0, 7, 4, 9]
0.8	0.2	0.8	2100	0	[0, 7, 4, 9]
0.8	0.2	0.9	2319	219	[0, 9]
0.8	0.2	1	2100	0	[0, 7, 4, 9]
0.9	0.1	0	2100	0	[0, 7, 4, 9]
0.9	0.1	0.1	2100	0	[0, 7, 4, 9]
0.9	0.1	0.2	2100	0	[0, 7, 4, 9]
0.9	0.1	0.3	2100	0	[0, 7, 4, 9]
0.9	0.1	0.4	2127	27	[0, 6, 5, 9]
0.9	0.1	0.5	2319	219	[0, 9]
0.9	0.1	0.6	2319	219	[0, 9]
0.9	0.1	0.7	2319	219	[0, 9]
0.9	0.1	0.8	2100	0	[0, 7, 4, 9]
0.9	0.1	0.9	2319	219	[0, 9]
0.9	0.1	1	2100	0	[0, 7, 4, 9]
1	0	0	2100	0	[0, 7, 4, 9]
1	0	0.1	2258	158	[0, 6, 1, 9]

Вывод

При небольших размерах графа (от 3 до 7) алгоритм полного перебора выигрывает по времени у муравьиного. Например, при размере графа 5, полный перебор работает быстрее примерно в 57 раз. Однако, при увеличении размера графа (от 9 и выше), ситуация меняется в обратную сторону: муравьиный алгоритм начинает значительно выигрывать по времени у алгоритма полного перебора. Наиболее стабильные результаты автоматической параметризации получаются при наборе $\alpha = 0.1..0.5$, $\beta = 0.1..0.5$, $\rho =$ любое. При таких параметрах полученный результат не отличается более чем на 1 от эталонного, и, в около 75% (на промежутке $\rho = 0.0..1.0$) случаев полученный результат совпадает с эталонным. Наиболее нестабильные результаты полученны при $\alpha = 1.0$, $\beta = 0.0$, $\rho =$ любое.

ЗАКЛЮЧЕНИЕ

В рамках данной лабораторной работы лабораторной работы была достигнута её цель: изучен муравьиный алгоритм и приобретены навыки параметризации методов на примере муравьиного алгоритма.

Также выполнены следующие задачи:

- реализованны два алгоритма решения задачи поиска кратчайшего пути;
- замерено время выполнения алгоритмов;
- муравьиный алгоритм протестирован на разных переменных;
- сделаны выводы на основе проделанной работы;

Использовать муравьиный алгоритм для решения задачи коммивояжера выгодно (с точки зрения времени выполнения), в сравнении с алгоритмом полного перебора, в случае если в анализируемом графе вершин больше либо равно 9. Так, например, при размере графа 11, муравьиный алгоритм работает быстрее чем алгоритм полного перебора в 15 раз. Стоит отметить, что муравьиный алгоритм не гарантирует что найденный путь будем оптимальным, так как он является эвристическим алгоритмом, в отличии от алгоритма полного перебора.

Цели лабораторной работы по изучению и исследованию муравьиного алгоритма были достигнуты

Литература

[1] Arch Linux [Электронный ресурс]. Режим доступа: https://archlinux.org/. Дата обращения: 19.10.2022.

[2] Процессор Intel® Core™ i5-11320H [Электронный ресурс]. Режим доступа: https://ark.intel.com/content/www/ru/ru/ark/products/217183/intel-core-i511320h-processor-8m-cache-up-to-4-50-ghz-with-ipu.html. Дата обращения: 19.10.2022.

[3] Criterion [Электронный ресурс]. Режим доступа: https://github.com/bheisler/criterion.rs. Дата обращения: 19.10.2022.