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 дисциплине «Анализ Алгоритмов»

Тема Задача коммивояжера

Студент Нисуев Н.Ф.

Группа ИУ7-52Б

Преподаватель Волкова Л. Л., Строганов Д.В.

2024 г.

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Задача коммивояжера — одна из наиболее известных и старейших задач комбинаторной оптимизации. В 1831 г. в Германии вышла книга под названием "Кто такой коммивояжер и что он должен делать для процветания своего предприятия". Одна из рекомендаций этой книги гласила: "Важно посетить как можно больше мест возможного сбыта, не посещая ни одно из них дважды". Это была первая формулировка задачи коммивояжера [1].

Цель лабораторной работы — рассмотрение алгоритмов решения задачи коммивояжера. Для достижения поставленной цели необходимо выполнить следующие задачи:

— сформулировать задачу коммивояжера;

— рассмотреть методы решения с использованием полного перебора и муравьиного алгоритма;

— реализовать указанные алгоритмы;

— провести сравнительный анализ времени работы алгоритмов;

— выполнить параметризацию для муравьиного алгоритма.

1 Аналитическая часть

В данном разделе будет сформулирована задача коммивояжера, а также будут рассмотрены 2 метода решения этой задачи: муравьиный алгоритм и алгоритм полного перебора.

1.1 Формулировка задачи коммивояжера

Пусть задан граф $G = (V, E)$, где V — множество вершин ($|V| = n$), а E — множество ребер ($|E| = m$). Каждое ребро $(i, j) \in E$ имеет длину c_{ij}, которая задается матрицей расстояний $C = \|c_{ij}\|$. Если между вершинами i и j нет ребра, соответствующий элемент матрицы считается равным бесконечности ($c_{ij} = \infty$) [1].

Произвольное подмножество попарно несмежных ребер графа G называется паросочетанием в G. Паросочетание $A \subset E(G)$ называется совершенным, если каждая вершина графа G инцидентна единственному ребру из . Произвольная совокупность простых попарно непересекающихся циклов в графе G, покрывающая все вершины графа G, называется 2-фактором в G. **Гамильтоновым циклом** в графе G называется 2-фактор, состоящий из одного цикла [2].

Требуется найти гамильтонов цикл, то есть цикл, проходящий через каждую вершину графа ровно один раз и возвращающийся в начальную точку, минимальной длины.

1.2 Алгоритм полного перебора

Алгоритм полного перебора для решения задачи коммивояжера заключается в рассмотрении всех возможных маршрутов в графе с целью нахождения минимального. Суть этого метода состоит в последовательном переборе всех вариантов обхода городов с выбором оптимального маршрута. Однако число возможных маршрутов стремительно увеличивается с ростом количества городов n, так как сложность алгоритма составляет $n!$. Несмотря на то, что алгоритм полного перебора гарантирует точное решение задачи, его использование приводит к значительным временным затратам уже при сравнительно небольшом числе городов.

1.3 Муравьиный алгоритм

Муравьиный алгоритм — это метод решения задачи коммивояжера, основанный на моделировании поведения муравьиной колонии [3].

Каждый муравей прокладывает маршрут, используя информацию о феромонах, оставленных другими муравьями на графе. В процессе движения муравей оставляет феромон на своем пути, чтобы другие могли ориентироваться на него. Постепенно феромоны на оптимальном маршруте накапливаются, так как он используется наиболее часто.

Характеристики муравья:

— **зрение** — муравей способен оценивать длину ребер.

— **память** — запоминает посещенные вершины.

— **обоняние** — реагирует на феромоны, оставленные другими муравьями.

Целевая функция

Для оценки привлекательности перехода используется функция видимости (1.1):

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

где D_{ij} — расстояние между вершинами i и j.

Формула вероятности перехода

Вероятность перехода муравья k из текущей вершины i в вершину j рассчитывается по формуле (1.2):

$$P_{kij} = \begin{cases} \frac{\tau_{ij}^a \eta_{ij}^b}{\sum_{q \in J_{ik}} \tau_{iq}^a \eta_{iq}^b}, & \text{если вершина } j \text{ еще не посещена муравьем } k, \\ 0, & \text{иначе,} \end{cases} \tag{1.2}$$

где:

— a — параметр влияния феромона;

— b — параметр влияния длины пути;

— τ_{ij} — количество феромонов на ребре (i, j);

— η_{ij} — видимость (обратная расстоянию).

Обновление феромонов

После завершения движения всех муравьев уровень феромонов на ребрах обновляется по формуле (1.3):

$$\tau_{ij}(t+1) = (1-p)\tau_{ij}(t) + \Delta\tau_{ij}, \tag{1.3}$$

где p — коэффициент испарения феромона, а $\Delta\tau_{ij}$ определяется как:

$$\Delta\tau_{ij} = \sum_{k=1}^{N} \Delta\tau_{ij}^{k}, \tag{1.4}$$

$$\Delta\tau_{ij}^{k} = \begin{cases} \frac{Q}{L_k}, & \text{если ребро } (i,j) \text{ посещено муравьем } k, \\ 0, & \text{иначе,} \end{cases} \tag{1.5}$$

где Q — параметр, связанный с длиной оптимального пути, а L_k — длина маршрута муравья k.

Элитные муравьи

Для улучшения временных характеристик муравьиного алгоритма вводят так называемых элитных муравьев. Элитный муравей усиливает ребра наилучшего маршрута, найденного с начала работы алгоритма. Количество феромона, откладываемого на ребрах наилучшего текущего маршрута T^+, принимается равным Q/L^+, где L^+ — длина маршрута T^+. Этот феромон побуждает муравьев к исследованию решений, содержащих несколько ребер наилучшего на данный момент маршрута T^+. Если в муравейнике есть e элитных муравьев, то ребра маршрута T^+ будут получать общее усиление:

$$\Delta\tau_e = e \cdot Q/L^+. \tag{1.6}$$

Описание алгоритма

1. Муравей исключает из дальнейшего выбора вершины из список посещенных вершин, которые хранятся в памяти муравья (список запретов J_{ik}).

2. Муравей оценивает привлекательность вершин на основе видимости, которая обратно пропорциональна расстоянию между вершинами.

3. Муравей ощущает уровень феромонов на ребрах графа, который указывает на предпочтительность маршрута.

4. После прохождения ребра (i, j) муравей оставляет на нем феромон, причем его количество зависит от длины маршрута L_k, пройденного муравьем, и параметра Q.

Таким образом, алгоритм постепенно находит оптимальный маршрут за счет коллективного взаимодействия муравьев и их способности к адаптации на основе накопленных феромонов.

ВЫВОД

В результате аналитического раздела была представлена графовая формулировка задача коммивояжера, а также рассмотрены 2 метода ее решения: муравьиный алгоритм и алгоритм полного перебора.

2 Конструкторская часть

В данном разделе будут определены требования к программному обеспечению и приведены схемы алгоритма полного перебора и муравьиного алгоритма для решения задачи коммивояжера.

2.1 Требования к программному обеспечению

К разрабатываемой программе предъявлен ряд требований:

Входные данные: Взвешенный неориентированный граф, заданный матрицей стоимостей.

Выходные данные: Оптимальный гамильтонов цикл и субоптимальный гамильтонов цикл при использовании алгоритма полного перебора и муравьиного алгоритма соответственно.

2.2 Схемы алгоритмов

На рисунках 2.1 — 2.2 представлены схема алгоритма полного перебора и схема муравьиного алгоритма.

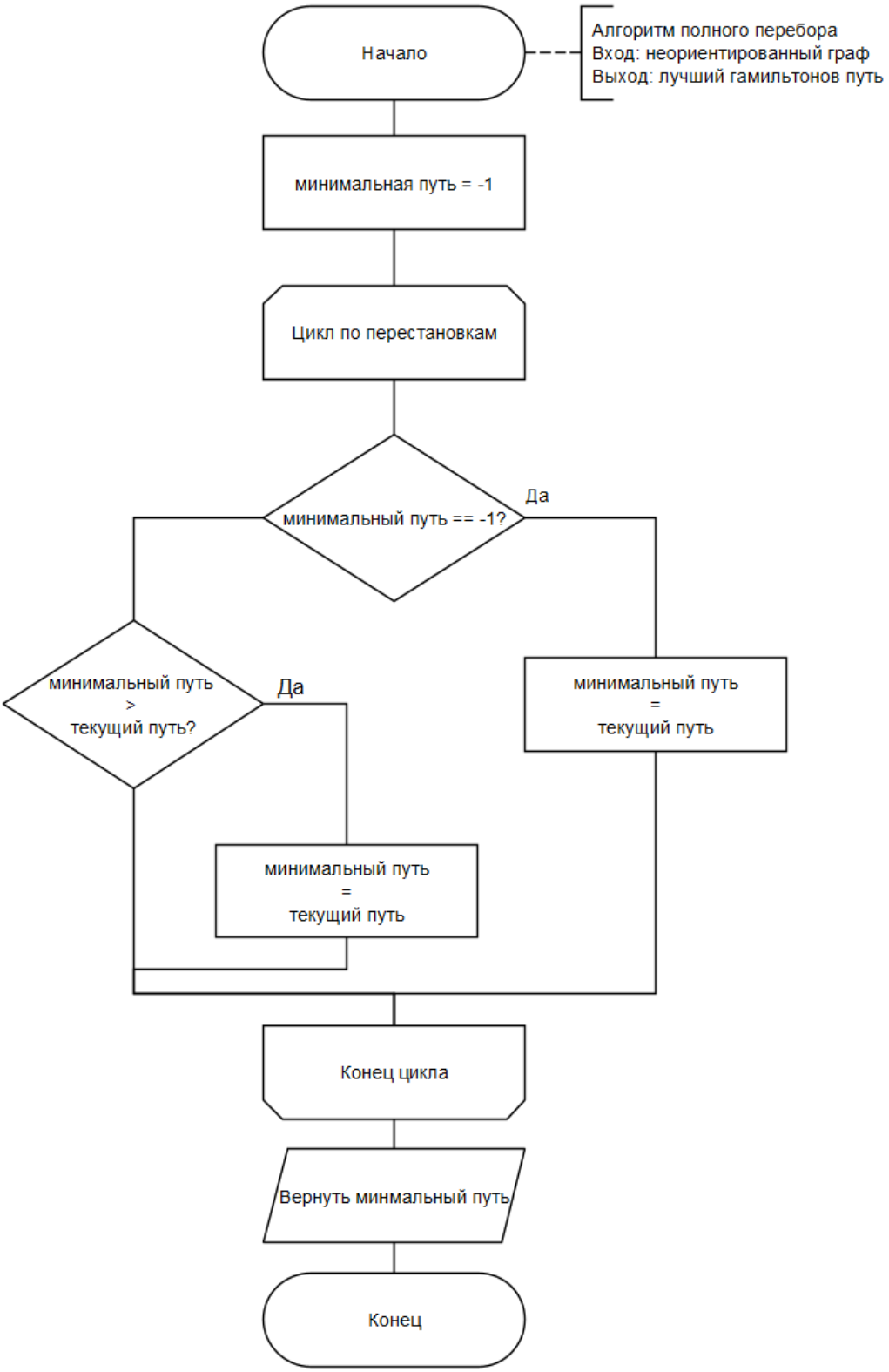

Рисунок 2.1 – Схема алгоритма полного перебора

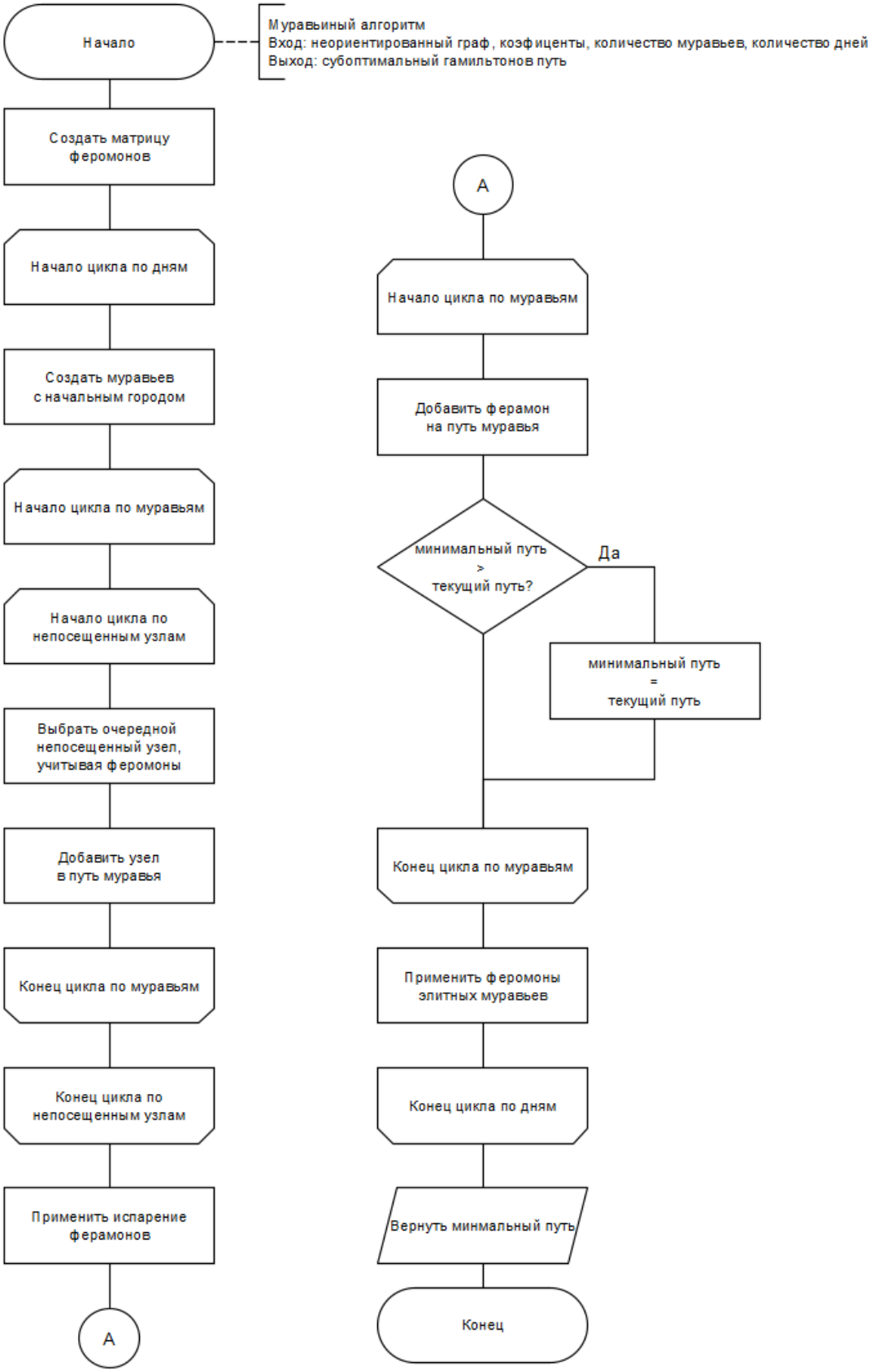

Рисунок 2.2 – Схема муравьиного алгоритма

ВЫВОД

В результате конструкторского раздела были определены требования к программному обеспечению и приведены схемы алгоритма полного перебора и муравьиного алгоритма для решения задачи коммивояжера.

3 Технологическая часть

В данном разделе будут приведены средства реализации, листинги кода.

3.1 Средства реализации

В данной работе для реализации был выбран язык программирования *Rust* [4]. Выбор обусловлен скоростью выполнения и наличием опыта работы с ним.

3.2 Реализация алгоритмов

В листингах 3.3 — 3.4 представлены реализации алгоритмов решения задачи коммивояжера. В листинге 3.1 представлена структура данных, с которой работают алгоритмы. В листинге 3.2. Расстояния между в между городами в примерах взяты на основе реальных расстояний [5—7].

Листинг 3.1 – структура AncientWorld

```
pub struct AncientWorld {
    pub cities_count: usize,
    pub cities_names: Vec<String>,
    pub cities_roads: Vec<Vec<usize>>,
}
```

Листинг 3.2 – примеры стран древнего мира

```
lazy_static!{
    pub static ref CANAAN: AncientWorld = AncientWorld {
        cities_count: 5,
        cities_names: vec![
            "Jerusalem".to_string(),
            "Jericho".to_string(),
            "Gaza".to_string(),
            "Hebron".to_string(),
            "Ashkelon".to_string(),
        ],
        cities_roads: vec![
```

```
12              vec![0, 27, 77, 30, 60],
13              vec![27, 0, 90, 40, 87],
14              vec![77, 90, 0, 50, 20],
15              vec![30, 40, 50, 0, 45],
16              vec![60, 87, 20, 45, 0],
17          ],
18      };
19
20      pub static ref MESOPOTAMIA: AncientWorld = AncientWorld {
21          cities_count: 10,
22          cities_names: vec![
23              "Babylon".to_string(),
24              "Uruk".to_string(),
25              "Ur".to_string(),
26              "Nippur".to_string(),
27              "Lagash".to_string(),
28              "Nineveh".to_string(),
29              "Ashur".to_string(),
30              "Eridu".to_string(),
31              "Sippar".to_string(),
32              "Eshnunna".to_string(),
33          ],
34          cities_roads: vec![
35              vec![0, 200, 300, 100, 250, 450, 500, 400, 150, 120],
36              vec![200, 0, 100, 150, 300, 500, 550, 300, 300, 200],
37              vec![300, 100, 0, 250, 150, 600, 650, 200, 350, 300],
38              vec![100, 150, 250, 0, 100, 500, 550, 300, 150, 170],
39              vec![250, 300, 150, 100, 0, 600, 650, 200, 300, 300],
40              vec![450, 500, 600, 500, 600, 0, 150, 700, 350, 400],
41              vec![500, 550, 650, 550, 650, 150, 0, 750, 400, 450],
42              vec![400, 300, 200, 300, 200, 700, 750, 0, 350, 300],
43              vec![150, 300, 350, 150, 300, 350, 400, 350, 0, 100],
44              vec![120, 200, 300, 170, 300, 400, 450, 300, 100, 0],
45          ],
46      };
47
48      pub static ref ANCIENT_ROME: AncientWorld = AncientWorld {
49          cities_count: 15,
50          cities_names: vec![
51              "Rome".to_string(),
52              "Pompeii".to_string(),
```

```
53              "Carthage".to_string(),
54              "Athens".to_string(),
55              "Alexandria".to_string(),
56              "Byzantium".to_string(),
57              "Antioch".to_string(),
58              "Ephesus".to_string(),
59              "Lugdunum".to_string(),
60              "Londinium".to_string(),
61              "Massilia".to_string(),
62              "Ravenna".to_string(),
63              "Verona".to_string(),
64              "Brundisium".to_string(),
65              "Capua".to_string(),
66          ],
67          cities_roads: vec![
68              vec![0, 240, 820, 1000, 1300, 1400, 1100, 950, 450, 1200,
                    500, 300, 400, 700, 180],
69              vec![240, 0, 780, 970, 1250, 1400, 1150, 900, 500, 1250,
                    550, 350, 450, 750, 60],
70              vec![820, 780, 0, 1500, 1500, 2000, 1700, 1550, 1300, 2100,
                    950, 1000, 1200, 1700, 800],
71              vec![1000, 970, 1500, 0, 600, 400, 700, 600, 1200, 1600,
                    700, 800, 950, 1200, 950],
72              vec![1300, 1250, 1500, 600, 0, 400, 300, 700, 1500, 1900,
                    1000, 1300, 1500, 1700, 1200],
73              vec![1400, 1400, 2000, 400, 400, 0, 500, 300, 1600, 2000,
                    1200, 1500, 1700, 2000, 1350],
74              vec![1100, 1150, 1700, 700, 300, 500, 0, 400, 1300, 1800,
                    900, 1200, 1400, 1500, 1050],
75              vec![950, 900, 1550, 600, 700, 300, 400, 0, 1000, 1400,
                    750, 950, 1150, 1300, 850],
76              vec![450, 500, 1300, 1200, 1500, 1600, 1300, 1000, 0, 800,
                    450, 300, 350, 750, 350],
77              vec![1200, 1250, 2100, 1600, 1900, 2000, 1800, 1400, 800,
                    0, 1250, 1400, 1600, 1750, 1300],
78              vec![500, 550, 950, 700, 1000, 1200, 900, 750, 450, 1250,
                    0, 250, 350, 550, 400],
79              vec![300, 350, 1000, 800, 1300, 1500, 1200, 950, 300, 1400,
                    250, 0, 150, 500, 250],
80              vec![400, 450, 1200, 950, 1500, 1700, 1400, 1150, 350,
                    1600, 350, 150, 0, 400, 350],
```

```
81              vec![700, 750, 1700, 1200, 1700, 2000, 1500, 1300, 750,
                    1750, 550, 500, 400, 0, 700],
82              vec![180, 60, 800, 950, 1200, 1350, 1050, 850, 350, 1300,
                    400, 250, 350, 700, 0],
83          ],
84      };
85  }
```

Листинг 3.3 – алгоритм полного перебора

```
1  impl AncientWorld {
2      pub fn solve_tsp_by_brute_force(&self) -> (usize, Vec<String>) {
3          if self.cities_count <= 1 {
4              return (0, self.cities_names.clone());
5          }
6
7          fn calculate_distance(path: &[usize], roads: &Vec<Vec<usize>>)
               -> usize {
8              path.windows(2).map(|w| roads[w[0]][w[1]]).sum::<usize>() +
                   roads[*path.last().unwrap()][path[0]]
9          }
10
11         let mut shortest_path = Vec::new();
12         let mut shortest_length = usize::MAX;
13
14         let cities: Vec<usize> = (0..self.cities_count).collect();
15
16         for perm in cities.iter().permutations(self.cities_count) {
17             let path: Vec<usize> = perm.into_iter().map(|&idx|
                   idx).collect();
18             let distance = calculate_distance(&path,
                   &self.cities_roads);
19             if distance < shortest_length {
20                 shortest_length = distance;
21                 shortest_path = path;
22             }
23         }
24
25         let path_names = shortest_path.iter().map(|&idx|
               self.cities_names[idx].clone()).collect();
26         (shortest_length, path_names)
27     }
```

```
}
```

Листинг 3.4 – муравьиный алгоритм

```
impl AncientWorld {
    pub fn solve_tsp_by_ant_colony(
        &self,
        alpha: f64,
        beta: f64,
        evaporation: f64,
        ants: usize,
        days: usize,
        elite_ants: usize,
    ) -> (usize, Vec<String>) {
        if self.cities_count <= 1 {
            return (0, self.cities_names.clone());
        }

        let mut pheromones = vec![vec![1.0; self.cities_count];
           self.cities_count];
        let mut best_path = Vec::new();
        let mut best_length = usize::MAX;

        for _ in 0..days {
            let mut all_paths = Vec::new();
            let mut all_lengths = Vec::new();

            for _ in 0..ants {
                let (path, length) =
                   self.simulate_ant_path(&pheromones, alpha, beta);
                all_paths.push(path.clone());
                all_lengths.push(length);

                if length < best_length {
                    best_length = length;
                    best_path = path;
                }
            }

            self.update_pheromones(
                &mut pheromones,
                evaporation,
```

```
37                  &all_paths,
38                  &all_lengths,
39                  &best_path,
40                  best_length,
41                  elite_ants,
42              );
43          }
44
45          let path_names = best_path.iter().map(|&idx|
                self.cities_names[idx].clone()).collect();
46          (best_length, path_names)
47      }
48
49      fn update_pheromones(
50          &self,
51          pheromones: &mut Vec<Vec<f64>>,
52          evaporation: f64,
53          all_paths: &Vec<Vec<usize>>,
54          all_lengths: &Vec<usize>,
55          best_path: &Vec<usize>,
56          best_length: usize,
57          elite_ants: usize,
58      ) {
59          for i in 0..self.cities_count {
60              for j in 0..self.cities_count {
61                  pheromones[i][j] *= 1.0 - evaporation;
62              }
63          }
64
65          for (path, length) in all_paths.iter().zip(all_lengths.iter()) {
66              for edge in path.windows(2) {
67                  let i = edge[0];
68                  let j = edge[1];
69                  pheromones[i][j] += 1.0 / *length as f64;
70              }
71              let i = path[path.len() - 1];
72              let j = path[0];
73              pheromones[i][j] += 1.0 / *length as f64;
74          }
75
76          for _ in 0..elite_ants {
```

```
77              for edge in best_path.windows(2) {
78                  let i = edge[0];
79                  let j = edge[1];
80                  pheromones[i][j] += 1.0 / best_length as f64;
81              }
82              let i = best_path[best_path.len() - 1];
83              let j = best_path[0];
84              pheromones[i][j] += 1.0 / best_length as f64;
85          }
86      }
87
88      fn simulate_ant_path(
89          &self,
90          pheromones: &Vec<Vec<f64>>,
91          alpha: f64,
92          beta: f64,
93      ) -> (Vec<usize>, usize) {
94          let mut visited = std::collections::HashSet::new();
95          let mut path = Vec::new();
96          let mut current_city = 0;
97          visited.insert(current_city);
98          path.push(current_city);
99          let mut length = 0;
100
101         while visited.len() < self.cities_count {
102             let chosen_city = self.choose_next_city(pheromones,
                    &visited, current_city, alpha, beta);
103             visited.insert(chosen_city);
104             path.push(chosen_city);
105             length += self.cities_roads[current_city][chosen_city];
106             current_city = chosen_city;
107         }
108
109         length += self.cities_roads[current_city][path[0]];
110         (path, length)
111     }
112
113     fn choose_next_city(
114         &self,
115         pheromones: &Vec<Vec<f64>>,
116         visited: &std::collections::HashSet<usize>,
```

```
117         current_city: usize,
118         alpha: f64,
119         beta: f64,
120     ) -> usize {
121         let mut probabilities = Vec::new();
122         let mut total_prob = 0.0;
123 
124         for next_city in 0..self.cities_count {
125             if !visited.contains(&next_city) {
126                 let pheromone = pheromones[current_city][next_city];
127                 let distance =
                        self.cities_roads[current_city][next_city];
128                 let prob = pheromone.powf(alpha) * (1.0 / distance as
                        f64).powf(beta);
129                 probabilities.push((next_city, prob));
130                 total_prob += prob;
131             }
132         }
133 
134         let mut choice = rand::random::<f64>() * total_prob;
135         for (next_city, prob) in probabilities {
136             if choice < prob {
137                 return next_city;
138             }
139             choice -= prob;
140         }
141 
142         unreachable!("No city found for transit");
143     }
144 }
```

ВЫВОД

В ходе технологической части работы были разработаны муравьиный алгоритм и алгоритм полного перебора для задачи коммивояжера.

4 Исследовательская часть

4.1 Технические характеристики

Характеристики используемого оборудования:

— операционная система — Windows 11 Home

— память — 16 Гб.

— процессор — 12th Gen Intel(R) Core(TM) i7—12700H @ 2.30 ГГц [8]

4.2 Время выполнения алгоритмов

Замеры времени проводились на графах с одинаковым количеством вершин. Каждое значение получено путем взятия среднего из 10 измерений. Зависимости времени решения задачи коммивояжера от количества вершин графа для двух алгоритмов представлены на рисунке 4.1.

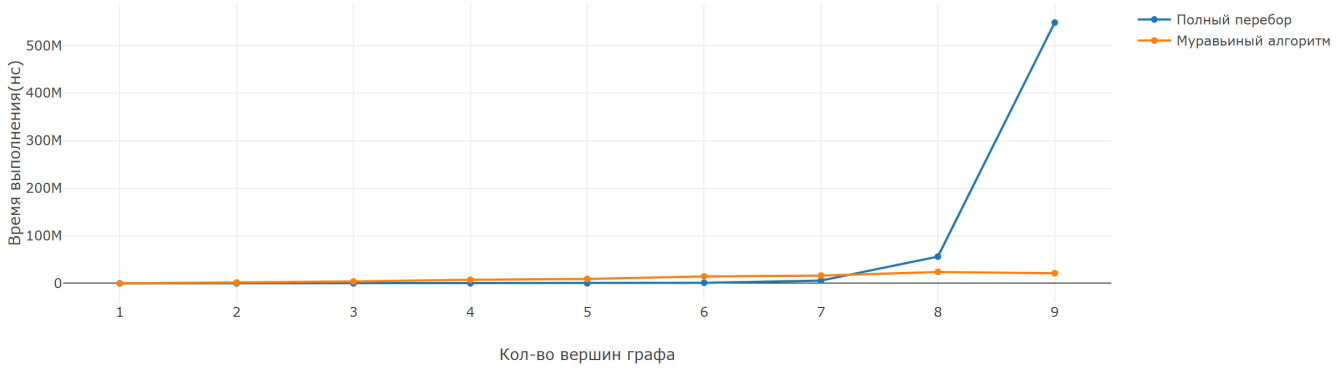

Рисунок 4.1 – Сравнение алгоритмов по времени

4.3 Результаты параметризации

В результате параметризации оказалось, что самыми лучшими параметрами по заданному классу данных оказались при $\alpha = 0.75$, $\rho = 0.1$ и количеством дней равным 200. При этом данные параметры дают лучший результат на всех классах

данных и по всем параметрам сравнения. Результаты параметризации для лучших параметров представлены в таблице 4.1, а вся таблица параметризации и класс данных представлены в приложении А.

Таблица 4.1 – Результаты параметризации муравьиного алгоритма

Параметры			Граф 1			Граф 2			Граф 3		
α	ρ	Дни	min	max	avg	min	max	avg	min	max	avg
0.50	0.50	200	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
0.75	0.10	100	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
0.75	0.10	200	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00

ВЫВОД

Результаты измерений времени показали, что на графа с вершинами меньше 8 муравьиный алгоритм уступает алгоритму полного перебора менее чем в 2.2 раза. Однако с увеличением количества вершин графа муравьиный алгоритм демонстрирует значительное преимущество. На графах с количеством вершин больше 7 муравьиный алгоритм работает быстрее более чем в 2.5 раз. Для проведения замеров времени количество дней в муравьином алгоритме было установлено равным 200.

Также в данном разделе была проведена параметризация и выявлены лучшие параметры для заданного в приложение А классов данных.

ЗАКЛЮЧЕНИЕ

В ходе работы были проанализированы временные и алгоритмические сложности муравьиного алгоритма и метода полного перебора. Также проведены замеры времени выполнения, выполнена параметризация муравьиного алгоритма, что позволило определить оптимальные параметры для набора данных, представленного в приложении А. Наилучшие параметры оказались следующими: $\alpha = 0.75$, ρ=0.1 и количество дней — 200.

В ходе выполнения данной лабораторной работы были решены следующие задачи:

— сформулирована задача коммивояжера;

— рассмотрены методы решения с использованием полного перебора и муравьиного алгоритма;

— реализованы указанные алгоритмы;

— проведен сравнительный анализ времени работы алгоритмов;

— выполнена параметризация для муравьиного алгоритма.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. *Меламед И. И.*, *Сергеев С. И.*, *Сигал И. Х.* Задача коммивояжера. Вопросы теории // Автоматика и телемеханика. — 1989. — № 9. — С. 3—33.
2. *Пережогин А. Л.*, *Потапов В. Н.* О числе гамильтоновых циклов в булевом кубе // Дискретный анализ и исследование операций. — 2001. — Т. 8, № 2. — С. 52—62.
3. *Штовба С.* Муравьиные алгоритмы // Exponenta Pro. Математика в приложениях. — 2003. — Т. 4, № 4. — С. 70—75.
4. The Rust Programming Language [Электронный ресурс]. — Режим доступа: `https://doc.rust-lang.org/book/` (дата обращения: 05.10.2024).
5. *Грей Д.* Ханаанцы. На земле чудес ветхозаветных. — Litres, 2011.
6. Mesopotamia [Электронный ресурс]. — Режим доступа: `https://www.history.com/topics/ancient-middle-east/mesopotamia` (дата обращения: 02.12.2024).
7. Digital Atlas of the Roman Empire [Электронный ресурс]. — Режим доступа: `https://imperium.ahlfeldt.se/` (дата обращения: 02.12.2024).
8. Intel® Core™ i7-12700H Processor [Электронный ресурс]. — Режим доступа: `https://ark.intel.com/content/www/us/en/ark/products/132228/intel-core-i7-12700h-processor-24m-cache-up-to-4-70-ghz.html` (дата обращения: 05.10.2024).