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» (ИУ7)**

НАПРАВЛЕНИЕ ПОДГОТОВКИ **09.03.04 Программная инженерия**

О Т Ч Е Т

по лабораторной работе № 6

Название: Решение задачи коммивояжёра методом полного перебора и муравьиным алгоритмом

Дисциплина: Анализ алгоритмов

Студент ИУ7-52Б _____ Е.В. Брянская
(Группа) (Подпись, дата) (И.О. Фамилия)

Преподаватель _____ Л.Л. Волкова
(Подпись, дата) (И.О. Фамилия)

Москва, 2020

Оглавление

Введение

В этой лабораторной работе будут рассматриваться два алгоритма, решающие задачу коммивояжёра.

Задача о коммивояжёре (travelling-salesman problem) является одной из знаменитых задач теории комбинаторики, была поставлена в 1934 году и заключается в поиске самого выгодного (минимального по стоимости) маршрута, проходящего строго по одному разу по всем приведённым городам с последующим возвратом в исходный город. Таким образом, выбор подходящего пути осуществялется среди гамильтоновых циклов.

Гамильтонов цикл - это такой цикл (замкнутый путь), который проходит через каждую вершину ровно по одному разу.

Задача коммивояжёра имеет ряд практических применений, к примеру, она использовалась для составления маршрутов лиц, занимающихся выемкой монет из таксофонов. В этом случае, в качестве пунктов, которые нужно посетить, выступали места установки таксофонов, а стоимость – время в пути между двумя точками.

Также она используется в задаче о сверлильном станке. Сверлильный станок изготавливает металлические листы с определённым количеством отверстий, координаты которых заранее известны. Нужно найти кратчайший путь через все отверстия, то есть наименьшее время, затрачиваемое на изготовление одной детали.

Для решения этой задачи есть несколько алгоритмов, в этой лабораторной работе будут рассмотрены: алгоритм полного перебора и муравьиный алгоритм.

1. Аналитическая часть

В этом разделе будут поставлены цель и основные задачи лабораторной работы, которые будут решаться по мере её выполнения.

1.1. Цель и задачи

Цель данной работы: провести сравнительный анализ метода полного перебора и эвристического метода на базе муравьиного алгоритма.

Для достижения поставленной цели необходимо решить следующий ряд **задач**:

1) дать описание базовой задачи;

2) описать алгоритмы;

3) реализовать все рассмотренные алгоритмы;

4) провести парметризацию муравьиного алгоритма для выбранного класса задач, то есть определить такие комбинации параметров или их диапазонов, при которых метод даёт наилучшие результаты.

1.2. Задача о коммивояжёре

В задаче о коммивояжёре, которая тесно связана с задачей о гамильтоновом цикле, коммивояжёр должен посетить n городов. Можно сказать, что коммивояжёру нужно совершить тур, или гамильтонов цикл, посетив каждый город ровно по одному разу и, завершив путешествие в том же городе, из которого он выехал. С каждым переездом из города i в город j связана некоторая стоимость пути $c(i, j)$, выраженная целым неотрицательным числом, и коммивояжёру нужно совершить тур таким образом, чтобы общая стоимость (т.е. сумма стоимостей всех переездов) была минимальной. [1]

Для решения этой задачи предлагается два следующих алгоритма.

1.3. Алгоритм полного перебора

Этот алгоритм заключается в полном переборе всех возможных комбинаций точек (городов). На вход подаётся число городов N и матрица стоимостей C. Так как количество городов равно N, то последовательно будут рассматриваться все перестановки из $N-1$ положительных целых чисел. Будет анализироваться каждый из этих возможных туров, и будет выбран тот, у которого наименьшая стоимость. [2]

Этот алгоритм достаточно точный, но продолжительность таких вычислений может занять много времени.

1.4. Муравьиный алгоритм

В его основе лежит моделирование колонии муравьёв, которая существует t дней. Колония представляет собой систему, в которой строго определены правила автономного поведения особей. Особенностью является то, что члены колонии могут быстро находить кратчайший путь и обмениваться этой информацией. В начале каждого t-ого дня осуществляется проход очередной партии муравьёв по городам.

В природе происходит непрямой обмен информацией, при котором одна особь изменяет некоторую область окружающей среды, а другие используют эту информацию, когда попадают в эту область. Каждый муравей оставляют особое вещество – феромон, по которому ориентируются следующие за ним особи. Чем больше муравьёв проходит по какому-либо участку, тем выше на нём концентрация этого вещества, тем самым насекомые помечают наиболее выгодные пути.

В алгоритме происходит моделирование такого поведения на некотором графе, ребра которого представляют собой возможные пути перемещения, и, как результат, наиболее обогащённый феромонами путь по рёбрам этого графа и будет решением поставленной задачи.

В основе муравьиного алгоритма лежат следующие принципы.

- В силу того, что каждый город должен быть посещён только один раз, у каждого муравья в колонии хранится информация об уже посещённых пунктах (далее этот список будет обозначаться как J);
- Муравьи обладают «зрением», то есть желанием посетить тот или иной город j, и эта величина рассчитывается по формуле 1.1.

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

  где D_{ij} - расстояние между городами i и j.

- Как говорилось выше, немаловажную роль в выборе следующего участка пути, играет феромон, который оставили другие муравьи, его количество обозначается как

τ_{ij}. И вероятность того, что дальнейший маршрут k-ого муравья будет построен по текущему ребру определяется по формуле 1.2.

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

где α, β – коэффициенты стадности и жадности, то есть, параметры, которые задают веса феромона, при $\alpha = 0$ алгоритм вырождается до жадного алгоритма (то есть будет выбран ближайший город).

В случае, если муравью удалось построить маршрут, удовлетворяющий всем требованиям, на задействованых рёбрах увеличивается количество феромона. Насколько изменится величина этого вещества на конкретном участке, рассчитывается следующим образом (формула 1.3).

$$\Delta\tau_{ij,k}(t) = \begin{cases} \frac{Q}{L_k(t)}, (i,j) \in T_k(t); \\ 0, (i,j) \notin T_k(t), \end{cases} \tag{1.3}$$

где Q – параметр, имеющий значение порядка длины оптимального пути, L_k – длина маршрута, $T_k(t)$ – маршрут, пройденный k-ым муравьём к моменту времени t.

Помимо всего прочего, алгоритм учитывает, подобно природной экосистеме, испарение феромона с наступлением ночи t-ого дня, причём с такой скоростью, чтобы не забывались хорошие решения и не возникало преждевременной сходимости. Рассчёты производятся по формуле 1.4.

$$\tau_{ij}(t+1) = (1-\rho) \cdot \tau_{ij}(t) + \Delta\tau_{ij}(t); \;\; \Delta\tau_{ij}(t) = \sum_{k=1}^{m} \Delta\tau_{ij,k}(t), \tag{1.4}$$

где $\rho \in [0,1]$ – коэффициент испарения, m – количество муравьёв в колонии.

Для большей модификации алгоритма используются «элитные» муравьи, которые усиливают рёбра наилучшего маршрута, найденного с начала работы алгоритма. [3]

В результате работы алгоритма находится кратчайший маршрут среди всех тех, которые были найдены. Но важно обратить внимание на то, что не всегда этот маршрут совпадёт с решением, которое можно получить с помощью алгоритма полного перебора, ввиду определённых причин. Можно получить возможно не совсем точное, но очень близкое к идеальному значение.

Вывод

Были поставлены цель и задачи текущей лабораторной работы, также дано описание рассматриваемых алгоритмов.

2. Конструкторская часть

Рассмотрим работу алгоритмов на матрицах стоимостей размера $N \times N$.

2.1. Алгоритм полного перебора

Сначала составляются все возможные маршруты, проходящие через все города ровно по одному разу и возвращающиеся в начальный пункт.

Затем каждая такая перестановка анализируется на предмет существования, и, если такой маршрут можно проложить, то находится его длина с помощью матрицы стоимостей, если же нельзя, то проверяется следующий и т.д.

В случае, если полученное значение длины оказывается меньше текущего минимального, то и этот маршрут, и его длина запоминаются в качестве потенциального решения задачи, и дальнейшие сравнения будут проходить уже с этими величинами. Таким образом, просматриваются все возможные варианты маршрутов.

Схема алгоритма представлена на Рис.2.1, 2.2.

2.2. Муравьиный алгоритм

В начале работы алгоритма задаются равные значения феромонов на каждой вершине.

Затем осуществляется проход по каждому из отведённых дней, создаётся колония из N муравьёв, причём каждому муравью ставится в соответствие свой город, в качестве начальной позиции. Для каждой особи составляется список вершин, которые обязательно должны быть пройдены, и по мере передвижения уже посещённые будут из него удаляться.

Процесс построения маршрута будет продолжаться до тех пор, пока этот список не станет пустым, или, пока не произойдёт тупиковая ситуация, когда из вершины никуда дальше перейти нельзя.

Чтобы обеспечить выполнение условия замкнутости, дополнительно в список непосещённых вершин добавляется начальная, когда все остальные вершины успешно пройдены.

Для того, чтобы определить, через какой узел маршрут пройдёт дальше, алгоритм делает следующее: выполняет проход по списку непосещённых вершин и проверяет по таблице стоимостей, возможно ли перейти из текущей позиции в рассматриваемую. И, исходя из этого, применяются соответсвующие формулы рассчёта. В результате, будет либо выбрана какая-либо вершина, либо сделан вывод о том, что муравей попал в тупик.

По мере того, как прокладывается маршрут, данные о его длине и компонентах постоянно обновляются.

Если маршрут успешно построен, то сравнивается полученный результат с промежуточным ответом, последний обновляется, если первый оказывается меньшим по длине. Затем алгоритм учитывает испарение феромона, а также вклад «элитных» муравьёв, увеличивающих концентрацию феромона на минимальном по длине участке, и всей колонии в целом.

Выполнив все необходимые действия, алгоритм возвращает итоговый результат.

Схема алгоритма представлена на Рис.2.3, 2.4, 2.5, 2.6, 2.7.

2.3. Автоматизация параметризации

Автоматизация параметризации проводится на базе муравьиного алгоритма, поскольку, как видно из формул 1.2, 1.4, результат сильно зависит от используемых параметров, которые различаются в зависимости от данных конкретной задачи. Поэтому, для поиска нужных значений нужно создать алгоритм, который бы перебирал их, выполнял поставленную задачу для каждого набора параметров и путём анализа выбирал тот, у которого наилучшие показатели.

2.4. Требования к ПО

Для корректной работы алгоритмов и проведения тестов необходимо выполнить следующее.

- Обеспечить возможность ввода количества городов и выбора алгоритма через консоль.
- В случае ввода некорректных данных вывести соответствующее сообщение. Программа не должна аварийно завершаться.

- Реализовать функцию, осуществляющую испытания муравьиного алгоритмами с различными значениями параметров, вывести результаты на экран.

2.5. Заготовки тестов

При проверке на корректность работы необходимо провести следующие тесты:

- стандартный тест;
- число городов $N = 2$;
- нет решения;
- два или более маршрута с одинаковой протяженностью.

Вывод

В этом разделе разобраны основные принципы выбранных алгоритмов, приведены схемы работы каждого из них, сформулированы требования к программному обеспечению и сделаны заготовки тестов.

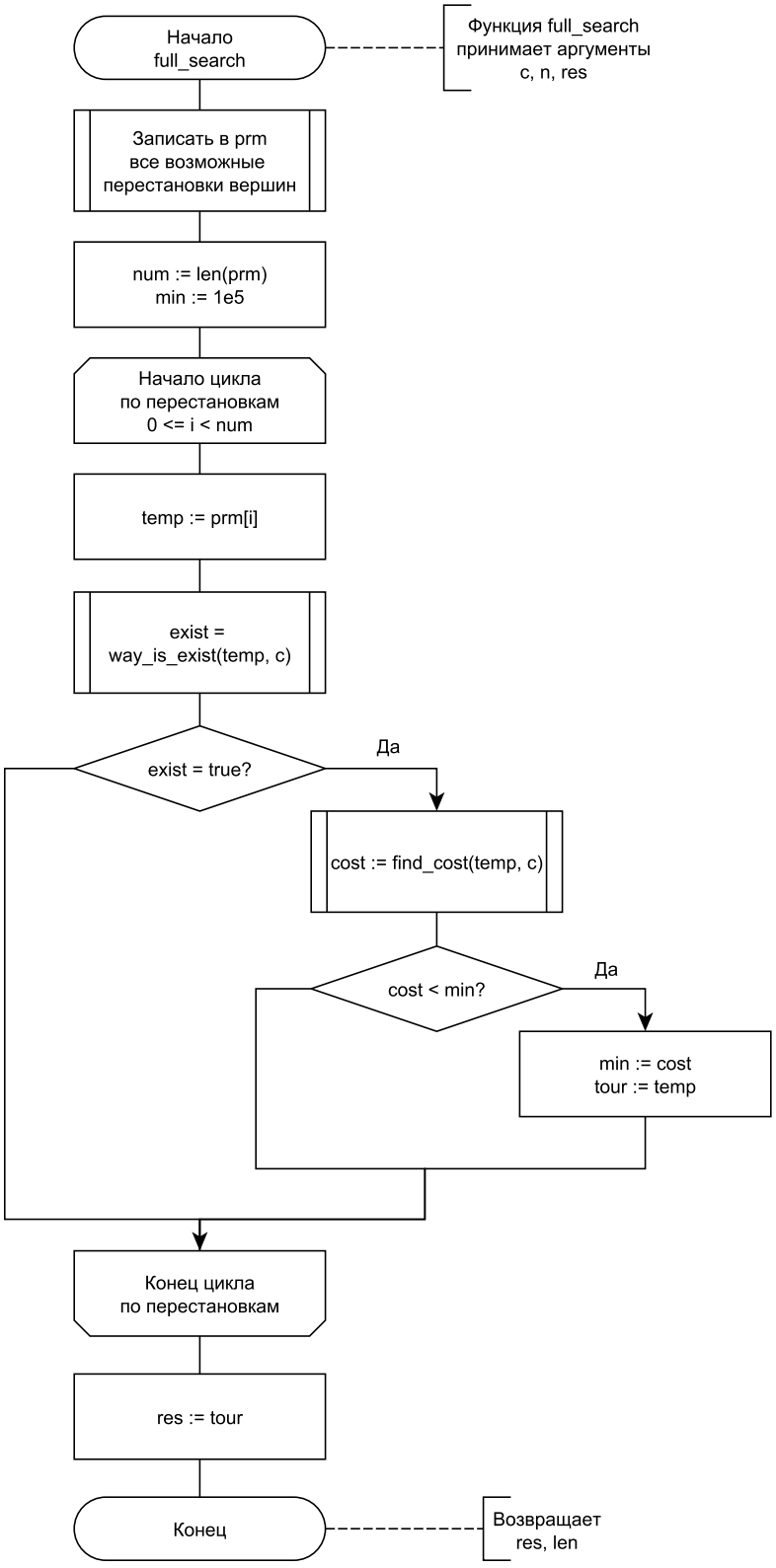

Рис. 2.1 — Алгоритм полного перебора (часть 1)

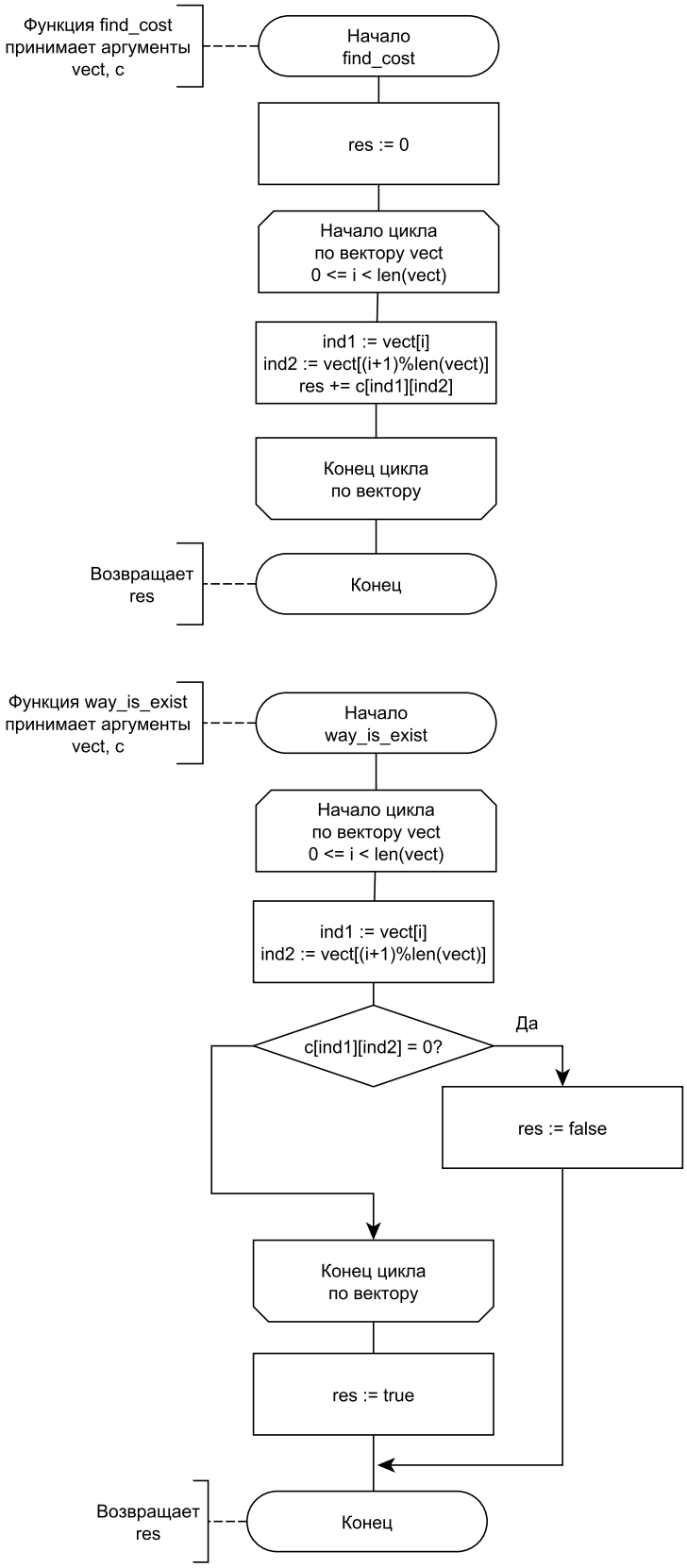

Рис. 2.2 — Алгоритм полного перебора (часть 2)

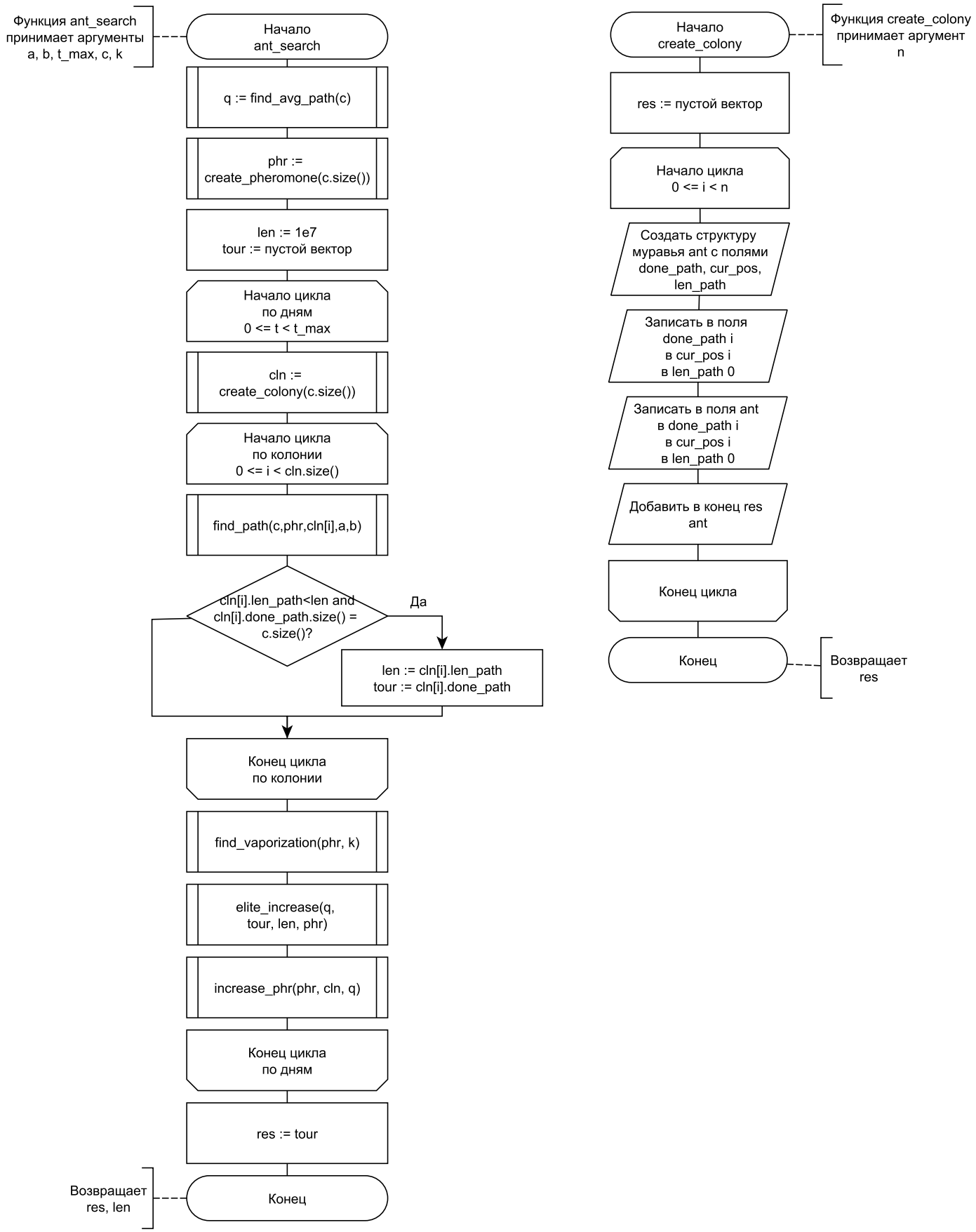

Рис. 2.3 — Муравьиный алгоритм (часть 1)

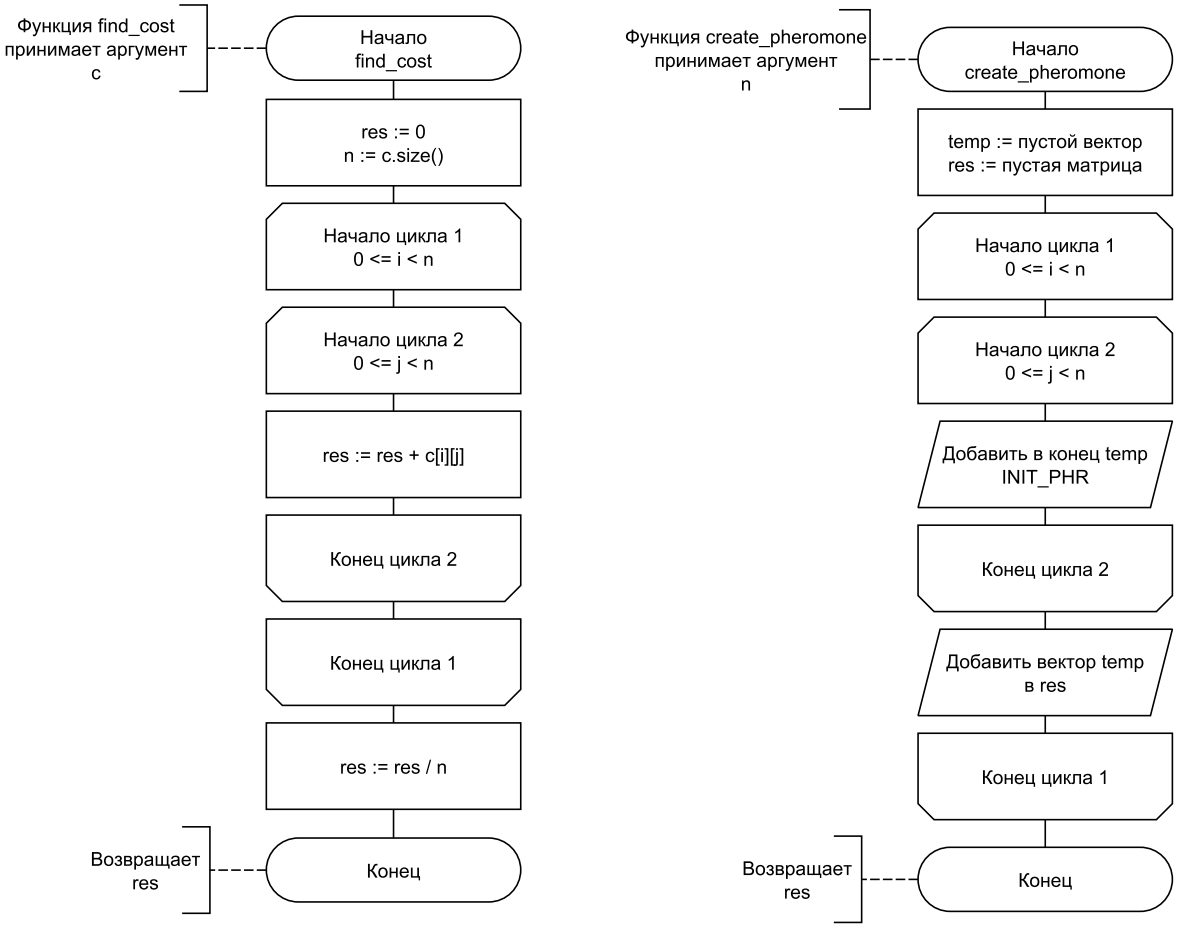

Рис. 2.4 — Муравьиный алгоритм (часть 2)

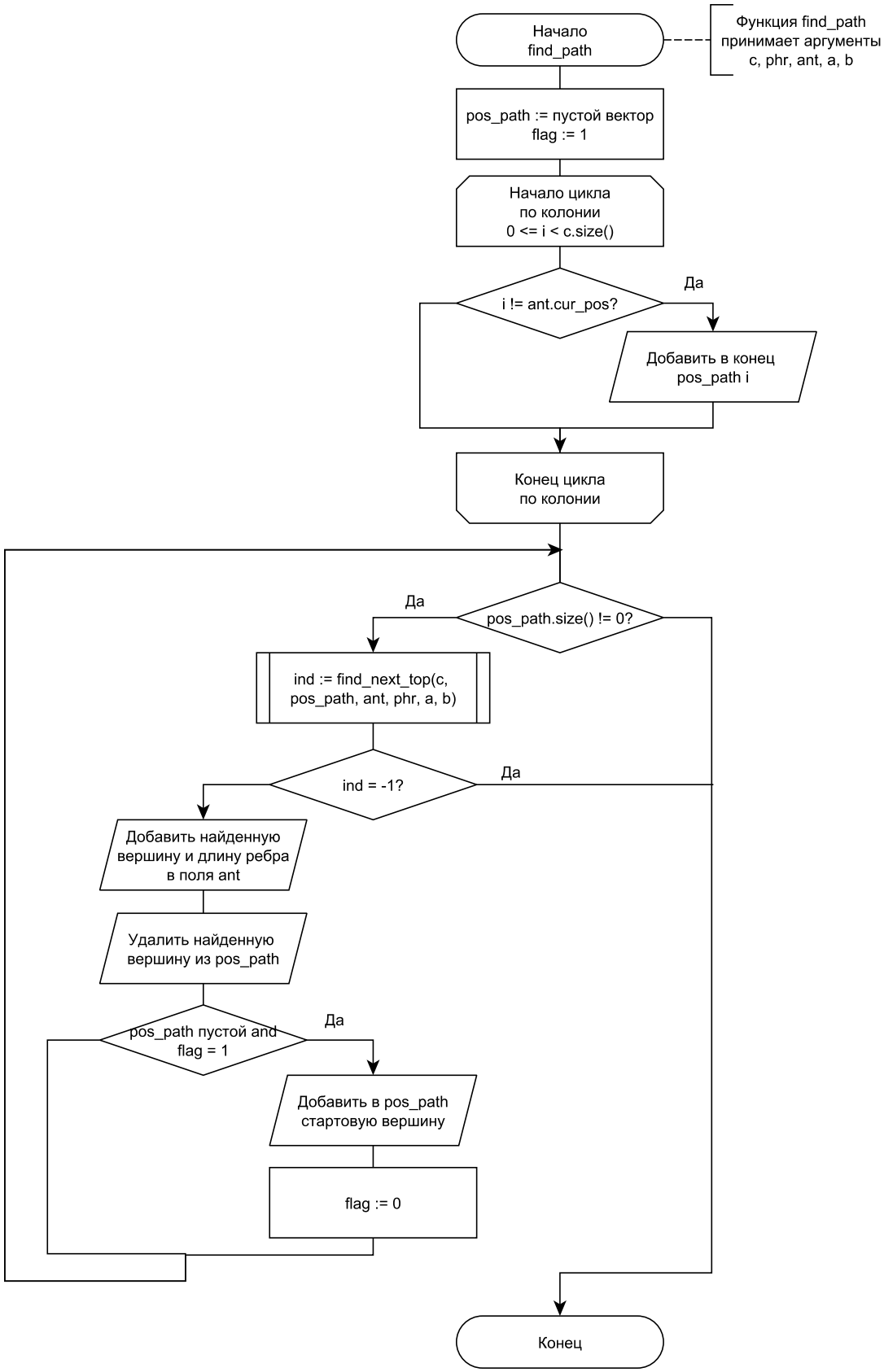

Рис. 2.5 — Муравьиный алгоритм (часть 3)

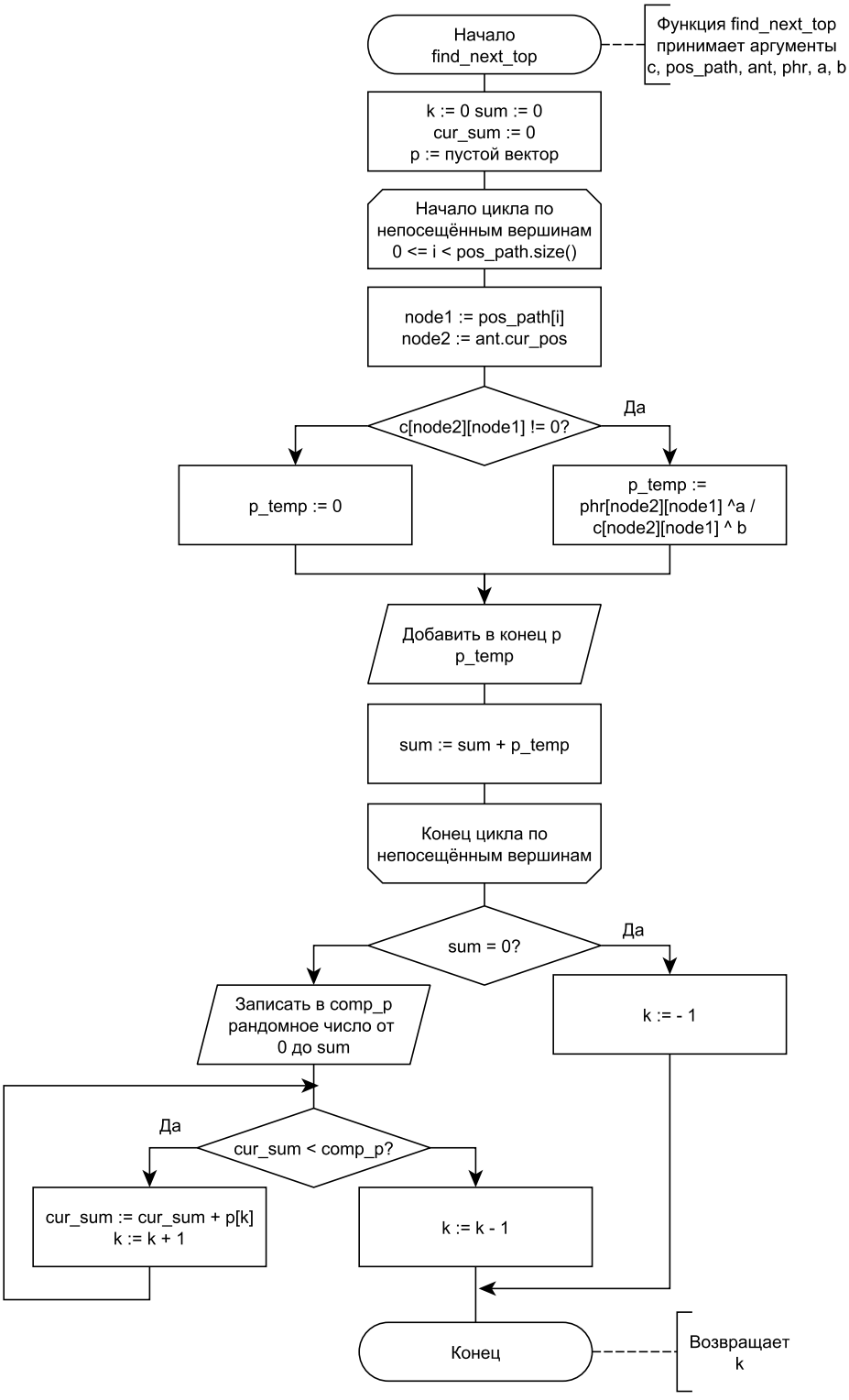

Рис. 2.6 — Муравьиный алгоритм (часть 4)

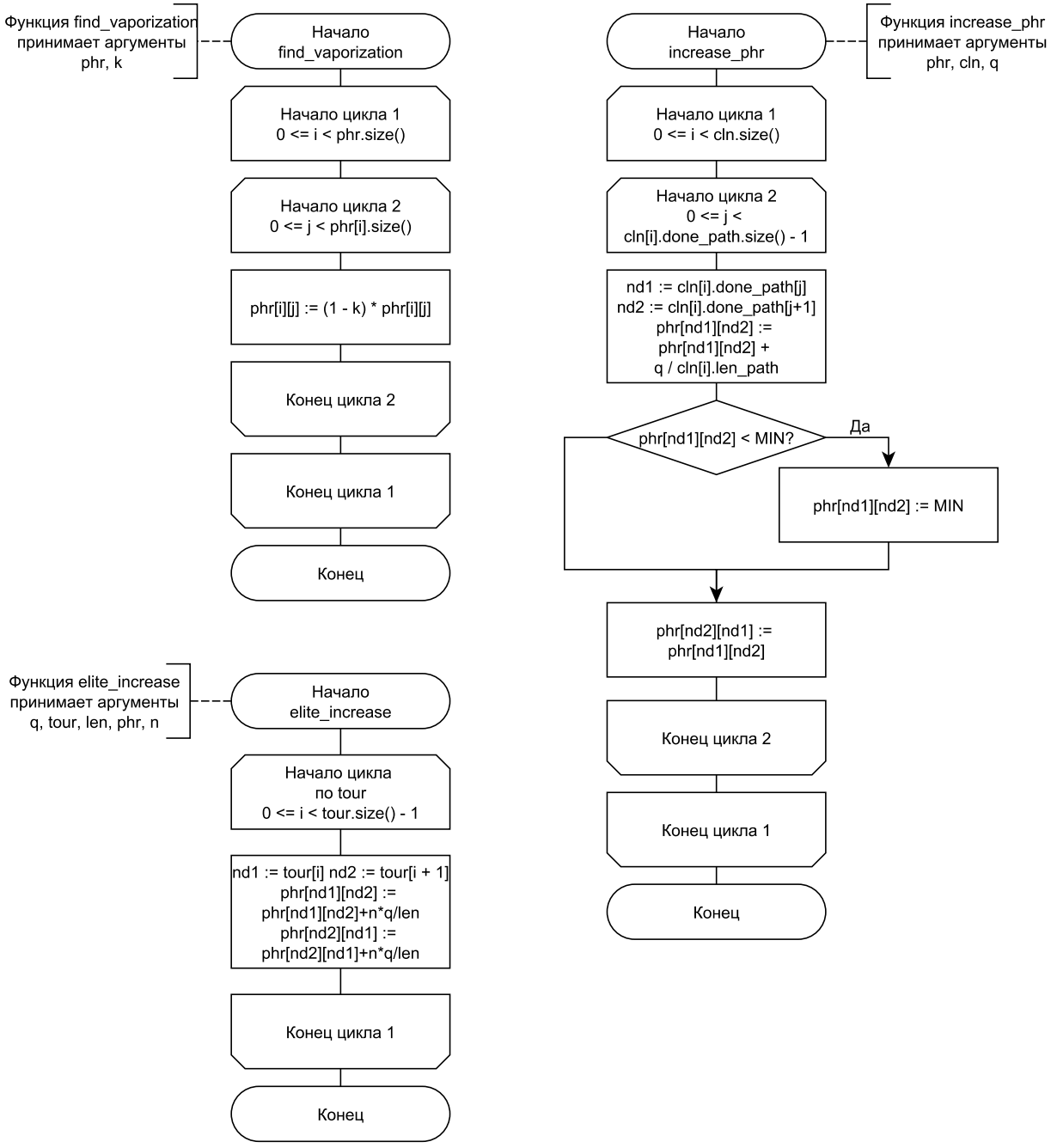

Рис. 2.7 — Муравьиный алгоритм (часть 5)

3. Технологическая часть

В данном разделе будут приведены листинги функций разрабатываемых алгоритмов.

3.1. Выбранный язык программирования

Для выполнения этой лабораторной работы был выбран язык программирования C++, так как есть большой навык работы с ним и с подключаемыми библиотеками, которые также использовались для проведения тестирования и замеров. [4]

Использованная среда разработки - Visual Studio. [5]

3.2. Листинг кода

Ниже представлены Листиги 3.1, 3.2 функций, реализующих алгоритмы решения задачи коммивояжёра.

Листинг 3.1 — Алгоритм полного перебора

```
matrix_t create_permutations(int n)
{
  vector<int> vct;
  matrix_t res;

  for (int i = 0; i < n; i++)
    vct.push_back(i);

  res.push_back(vct);
  while (next_permutation(vct.begin(), vct.end()))
    res.push_back(vct);

  return res;
}

bool way_is_exist(vector<int> vect, matrix_t& c)
{
  for (int i = 0; i < vect.size(); i++)
    if (c[vect[i]][vect[(i + 1) % vect.size()]] == 0)
      return false;

  return true;
```

```
}

int find_cost(vector<int> vect, matrix_t& c)
{
  int res = 0;

  for (int i = 0; i < vect.size(); i++)
    res += c[vect[i]][vect[(i + 1) % vect.size()]];
  return res;
}

void full_search(matrix_t& c, int n)
{
  vector<int> temp, tour;
  matrix_t prm = create_permutations(n);
  int num = prm.size(), min = 1e5, cost;

  for (int i = 0; i < num; i++)
  {
    temp = prm[i];
    if (way_is_exist(temp, c))
    {
      cost = find_cost(temp, c);

      if (cost < min)
      {
        min = cost;
        tour = temp;
      }
    }
  }

  cout << "Found tour: ";
  for (int i = 0; i < tour.size(); i++)
    cout << tour[i] << " ";
  cout << endl;

  cout << "Its length: " << min << endl;
}
```

Листинг 3.2 — Муравьиный алгоритм

```
matrix_double_t create_pheromone(int n)
{
  matrix_double_t res;
  vector<double> temp;

  for (int i = 0; i < n; i++)
  {
    for (int j = 0; j < n; j++)
    temp.push_back(INIT_PHR);
    res.push_back(temp);
  }

  return res;
}

double find_avg_path(matrix_t& c)
{
  int n = c.size();
  double res = 0;

  for (int i = 0; i < n; i++)
    for (int j = 0; j < n; j++)
      res += c[i][j];

  return res / n;
}

vector<ant_t> create_colony(int n)
{
  vector<ant_t> res;

  for (int i = 0; i < n; i++)
  {
    ant_t ant;
    ant.done_path.push_back(i);
    ant.cur_pos = i;
    ant.len_path = 0;

    res.push_back(ant);
```

```
40    }
41
42    return res;
43  }
44
45  bool is_exist(int index, vector<int> path)
46  {
47    for (int i = 0; i < path.size(); i++)
48      if (path[i] == index)
49        return true;
50    return false;
51  }
52
53  void update_ant(int new_pos, int len, ant_t& ant)
54  {
55    ant.done_path.push_back(new_pos);
56    ant.len_path += len;
57    ant.cur_pos = new_pos;
58  }
59
60  int find_next_top(matrix_t& c, vector<int>& pos_path, ant_t& ant,
      matrix_double_t& phr, double a, double b)
61  {
62    int k = 0;
63    vector<double> p;
64    double p_temp, sum = 0, cur_sum = 0, comp_p;
65
66    for (int i = 0; i < pos_path.size(); i++)
67    {
68      int node = pos_path[i];
69      if (c[ant.cur_pos][node])
70        p_temp = pow(phr[ant.cur_pos][node], a) / pow(c[ant.cur_pos][node
      ], b);
71      else
72        p_temp = 0;
73
74      p.push_back(p_temp);
75      sum += p_temp;
76    }
77
```

```
78    if (sum == 0)
79      return -1;
80
81    comp_p = (double)rand() / RAND_MAX * sum;
82
83    while (cur_sum < comp_p)
84    {
85      cur_sum += p[k];
86      k++;
87    }
88
89    return k-1;
90  }
91
92  void find_path(matrix_t& c, matrix_double_t& phr, ant_t& ant, double a,
        double b)
93  {
94    vector<int> pos_path;
95    int ind, flag = 1;
96
97    for (int i = 0; i < c.size(); i++)
98      if (i != ant.cur_pos)
99        pos_path.push_back(i);
100
101   while (pos_path.size() != 0)
102   {
103     ind = find_next_top(c, pos_path, ant, phr, a, b);
104     if (ind == -1)
105       break;
106
107     update_ant(pos_path[ind], c[ant.cur_pos][pos_path[ind]], ant);
108
109     pos_path.erase(pos_path.begin() + ind);
110     if (pos_path.size() == 0 && flag)
111     {
112       pos_path.push_back(ant.done_path[0]);
113       flag = 0;
114     }
115   }
116 }
```

```
117
118 void find_vaporization(matrix_double_t& phr, double k)
119 {
120   for (int i = 0; i < phr.size(); i++)
121     for (int j = 0; j < phr[i].size(); j++)
122       phr[i][j] = (1 - k) * phr[i][j];
123 }
124
125 void increase_phr(matrix_double_t& pheromone, vector<ant_t> colony,
      double q)
126 {
127   for (int i = 0; i < colony.size(); i++)
128     for (int j = 0; j < colony[i].done_path.size() - 1; j++)
129     {
130       int node1 = colony[i].done_path[j];
131       int node2 = colony[i].done_path[j+1];
132       pheromone[node1][node2] += q / colony[i].len_path;
133       if (pheromone[node1][node2] < MIN_K_PHR * INIT_PHR)
134         pheromone[node1][node2] = MIN_K_PHR * INIT_PHR;
135       pheromone[node2][node1] = pheromone[node1][node2];
136     }
137 }
138
139 void elite_increase(double q, vector<int>& tour, int len,
      matrix_double_t& phr)
140 {
141   int num_el_ant = 2;
142
143   for (int i = 0; i < tour.size() - 1; i++)
144   {
145     int node1 = tour[i];
146     int node2 = tour[i + 1];
147
148     phr[node1][node2] += num_el_ant * q / len;
149     phr[node2][node1] += num_el_ant * q / len;
150   }
151 }
152
153 void ant_search(double a, double b, matrix_t& c, double k_vpr)
154 {
```

```
155   double q = find_avg_path(c);
156   matrix_double_t pheromone = create_pheromone(c.size());
157   int t_max = 100, len = 1e7;
158   vector<int> tour;
159
160   for (int t = 0; t < t_max; t++)
161   {
162     vector<ant_t> colony = create_colony(c.size());
163
164     for (int i = 0; i < colony.size(); i++)
165     {
166       find_path(c, pheromone, colony[i], a, b);
167       if (colony[i].len_path < len && colony[i].done_path.size() == c.
      size() + 1)
168       {
169         len = colony[i].len_path;
170         tour = colony[i].done_path;
171       }
172     }
173
174     find_vaporization(pheromone, k_vpr);
175
176     elite_increase(q, tour, len, pheromone);
177
178     increase_phr(pheromone, colony, q);
179   }
180
181   cout << "Found tour: ";
182   for (int i = 0; i < tour.size(); i++)
183     cout << tour[i] << " ";
184   cout << "\nIts length: " << len << endl;
185 }
```

3.3. Автоматизация параметризации

Была написана специальная функция параметризации, ориентированная на проработки муравьиного алгоритма на разных наборах параметров. Её реализация указана в листинге 3.3.

Листинг 3.3 — Тесты

```
void find_params(matrix_t& c, int goal_len)
{
  int cur_len, min_len = 1e5;
  vector<int> temp_res, result;
  params_t prm, prm_res;
  double q = find_avg_path(c);

  for (double a = 0; a <= 1; a += 0.1)
  {
    double b = 1 - a;

    for (double ro = 0; ro <= 1; ro += 0.1)
    {
      int temp_min = 1e5;
      prm = create_params(a, q, ro, 30);

      for (int k = 0; k < 3; k++)
      {
        cur_len = ant_search(prm.a, prm.b, c, prm.ro, prm.q, prm.t_max,
   temp_res);
        if (cur_len < temp_min)
        {
          temp_min = cur_len;
          result = temp_res;
        }
      }

      cout << prm.a << " " << prm.ro << " " << prm.t_max << " " <<
   temp_min << " " << temp_min - goal_len << endl;

      if (temp_min < min_len)
      {
        min_len = temp_min;
        prm_res = prm;
      }
    }
    cout << endl;
  }

```

```
38    cout << "Наилучший набор параметров для данной задачи: " << prm_res.a << "
       " << prm_res.b << " " << prm_res.ro << " " << prm_res.t_max << endl;
39
40    cout << "Найденный путь: ";
41    for (int i = 0; i < result.size(); i++)
42      cout << result[i] << " ";
43    cout << "\Eron длина: " << min_len << endl;
44  }
```

3.4. Результаты тестов

Для тестирования были написаны функции, проверяющие, согласно заготовкам выше, случаи. Выводы о корректности работы делаются на основе сравнения результатов.

Все тесты пройдены успешно. Сами тесты представлены ниже (Листинг 3.4).

Листинг 3.4 — Тесты

```
1  bool test(matrix_t& c)
2  {
3    vector<int> res1, res2;
4    int len1, len2;
5
6    len1 = full_search(c, c.size(), res1);
7    len2 = ant_search(0.5, 0.5, c, 100, res2);
8
9    if (len1 == len2 || res1.size() == 0 && res2.size() == 0)
10     return true;
11   else
12     return false;
13 }
14 void test_standart(matrix_t& c)
15 {
16   cout << endl << __FUNCTION__;
17   if (test(c))
18     cout << ":\tPASSED\n";
19   else
20     cout << ":\tFAILED\n";
21 }
22
```

```
23 void test_size_2(matrix_t& c)
24 {
25   cout << endl << __FUNCTION__;
26   if (test(c))
27     cout << ":\tPASSED\n";
28   else
29     cout << ":\tFAILED\n";
30 }
31
32 void test_no_solution(matrix_t& c)
33 {
34   cout << endl << __FUNCTION__;
35   if (test(c))
36     cout << ":\tPASSED\n";
37   else
38     cout << ":\tFAILED\n";
39 }
40
41 void test_equal(matrix_t& c)
42 {
43   cout << endl << __FUNCTION__;
44   if (test(c))
45     cout << ":\tPASSED\n";
46   else
47     cout << ":\tFAILED\n";
48 }
49
50 void run_tests()
51 {
52   matrix_t c1 = { {0, 1, 2}, {1, 0, 3}, {2, 3, 0} };
53   matrix_t c2 = { {0, 10}, {10, 0} };
54   matrix_t c3 = { {0, 0, 17}, {0, 0, 0}, {17, 0, 0} };
55   matrix_t c4 = { {0, 4, 4}, {4, 0, 4}, {4, 4, 0} };
56
57   test_standart(c1);
58   test_size_2(c2);
59   test_no_solution(c3);
60   test_equal(c4);
61 }
```

3.5. Оценка времени

Процессорное время измеряется с помощью функции QueryPerformanceCounter библиотеки windows.h. [6] Осуществление замеров показано ниже (Листинг 3.5).

Листинг 3.5 — Замеры процессорного времени

```
double PCFreq = 0.0;
__int64 CounterStart = 0;

void start_measuring()
{
  LARGE_INTEGER li;
  QueryPerformanceFrequency(&li);

  PCFreq = double(li.QuadPart) / 1000;

  QueryPerformanceCounter(&li);
  CounterStart = li.QuadPart;
}

double get_measured()
{
  LARGE_INTEGER li;
  QueryPerformanceCounter(&li);

  return double(li.QuadPart - CounterStart) / PCFreq;
}

void run_measuring_time(int size)
{
  matrix_t c = random_matrix(0, 100, size);
  vector<int> res;
  double q = find_avg_path(c);
  int count = 0;

  cout << "-> Количество городов: " << size << "\t";

  start_measuring();
  while (get_measured() < 3 * 1000)
  {
    ant_search(0.5, 0.5, c, 0.8, q, 30, res);
```

```
36      count++;
37    }
38
39    double t = get_measured() / 1000 / count;
40    cout << t << endl;
41  }
```

Вывод

Таким образом, приведены листинги кода каждой из функций, реализующих алгоритмы решения задачи коммивояжёра, а также листинг тестовых функций, направленных на проверку корректности их работы.

4. Исследовательская часть

4.1. Характеристики ПК

При проведении замеров времени использовался компьютер, имеющий следующие характеристики:

- ОС - Windows 10 Pro
- Процессор - Inter Core i7 10510U (1800 МГц)
- Объём ОЗУ - 16 Гб

4.2. Измерения

Для проведения замеров процессорного времени использовались матрицы размера $N \times N$, где $N \in \{2, 4, 8, 16, 32, 64, 128, 256\}$. Их содержимое генерируется случайным образом.

Каждый замер проводится 5 раз для получения более точного среднего результата.

В таблице таблице 4.1 представлены результаты.!!!!!!!!!!

Таблица 4.1 — Результаты измерений процессорного времени

N	2	4	8	16	32	64	128	256
Время	4.609e-5	2.136e-4	0.00117	0.007	0.0458	0.308	2.64	16.525

Можно заметить, что при каждом увеличении N в 2 раза, время, в свою очередь, увеличивается примерно от 4.7 до 6.26 раз. ??????????

Заключение

В ходе лабораторной работы была достигнута поставленная цель, а именно, проведён сравнительный анализ метода полного перебора и муравьиного алгоритма.

В процессе выполнения были решены все задачи. Описаны решаемая задача коммивояжёра, все рассматриваемые алгоритмы. Все проработанные алгоритмы реализованы, кроме того, были проведены замеры процессорного времени работы, оценена трудоёмкость муравьиного алгоритма и проведена его параметризация. На основе полученных данных проведён сравнительный анализ, сделаны выводы.

!!!!!!!!!!!!!!!!!!!!!!

Список литературы

1. Задача коммивояжёра [Электронный ресурс]. Режим доступа: http://www.avprog.narod.ru/student/kommi.htm свободный (дата обращения: 30.11.2020)

2. Кормен, Томас Х. и др Алгоритмы: построение и анализ, 3-е изд. : Пер. с англ. - М. : ООО "И.Д. Вильямс 2018. - 1328 с. : ил. - Парал. тит. англ. - ISBN 978-5-8459-2016-4 (рус.).

3. Ульянов М.В. Ресурсно-эффективные компьютерные алгоритмы. Разработка и анализ - Учебное пособие. М.: НАУКА, ФИЗМАТЛИТ, 2007. - 376 с. ISBN 978-5-9221-0950-5

4. Документация по C++ [Электронный ресурс]. Режим доступа: https://docs.microsoft.com/ru-ru/cpp/cpp/?view=msvc-160, свободный (дата обращения: 30.11.2020)

5. Документация по Visual Studio 2019 [Электронный ресурс]. Режим доступа: https://docs.microsoft.com/ru-ru/visualstudio/windows/?view=vs-2019, свободный (дата обращения: 30.11.2020)

6. QueryPerformanceCounter function [Электронный ресурс]. Режим доступа: https://docs.microsoft.com/en-us/windows/win32/api/profileapi/nf-profileapi-queryperformancecounter, свободный (дата обращения: 01.12.2020)

7. Клейнберг Дж., Тардос Е. Алгоритмы: разработка и применение. Классика Computer Science /Пер. с англ. Е. Матвеева. - СПб.: Питер, 2016. - 800 с.: ил. - (Серия "Классика computer science"). ISBN 978-5-496-01545-5