МГТУ им. Баумана

Лабораторная работа №6

По курсу: "Анализ алгоритмов"

Муравьиный алгоритм

Работу выполнил: студент группы ИУ7-53Б
Наместник Анастасия

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

Москва, 2020

Оглавление

Введение

Муравьиный алгоритм(алгоритм оптимизации подражанием муравьиной колонии) — один из эффективных полиномиальных алгоритмов для нахождения приближённых решений задачи коммивояжёра, а также решения аналогичных задач поиска маршрутов на графах[5].

Иными словами, это - семейство приближенных алгоритмов для решения различных сложных оптимизационных задач. Идея этих алгоритмов основана на моделировании поведения муравьиной колонии, выполняющей поиск пути от муравеййника к источнику пищи. Первый вариант муравьиного алгоритма, предназначенный для приближенного решения задачи коммивояжера, был разработан Марко Дориго в 1992 году[6].

Муравьиные алгоритмы серьезно исследуются европейскими учеными с середины 1990-х годов. Уже получены хорошие результаты муравьиной оптимизации для многих сложных комбинаторных задач: задачи коммивояжера, оптимизации маршрутов грузовиков, раскраски графа, квадратичной задачи о назначениях, оптимизации сетевых графиков, задачи календарного планирования и других. Особенно эффективны муравьиные алгоритмы при on-line оптимизации процессов в распределенных нестационарных системах, например трафиков в телекоммуникационных сетях[4].

Целью данной лабораторной работы является изучение муравьиных алгоритмов и приобретение навыков параметризации методов на примере муравьиного алгоритма, примененного к задаче коммивояжера.

В данной лабораторной работе требуется решить пять задач:

- изучить алгоритм полного перебора для решения задачи коммивояжера;
- изучить муравьиный алгоритм для решения задачи коммивояжера;

- программно реализовать описанные выше алгоритмы;
- провести параметризацию муравьиного алгоритма на классе данных;
- провести сравнительный анализ скорости работы реализованных алгоритмов.

1 Аналитическая часть

В этом разделе будут представлены теоретические сведения, необходимые для программной реализации муравьиного алгоритма, примененного к задаче коммивояжера.

1.1 Понятие коммивояжера

Коммивояжёр (фр. commis voyageur) — бродячий торговец. Задача коммивояжёра — важная задача транспортной логистики, отрасли, занимающейся планированием транспортных перевозок. Коммивояжёру, чтобы распродать нужные и не очень нужные в хозяйстве товары, следует объехать n пунктов и в конце концов вернуться в исходный пункт. Требуется определить наиболее выгодный маршрут объезда. В качестве меры выгодности маршрута (точнее говоря, невыгодности) может служить суммарное время в пути, суммарная стоимость дороги, или, в простейшем случае, длина маршрута[7].

1.2 Решение полным перебором

Совершенно очевидно, что задача может быть решена перебором всех вариантов объезда и выбором оптимального. Но при таком подходе количество возможных маршрутов очень быстро возрастает с ростом n (оно равно n! — количеству способов упорядочения пунктов). К примеру, для 100 пунктов количество вариантов будет представляться 158-значным числом. Мощная ЭВМ, способная перебирать миллион вариантов в секунду, будет решать такую задачу на протяжении примерно 3*10144 лет. Увеличение производительности ЭВМ в 1000 раз даст хоть и меньшее в 1000 раз, но по-прежнему очень большое время перебора вариантов. Не

спасает ситуацию даже то, что для каждого варианта маршрута имеется 2n равноценных, отличающихся выбором начального пункта (n вариантов) и направлением обхода (2 варианта). Перебор с учётом этого наблюдения сокращается незначительно.

Таким образом, хотя такой подход и гарантирует точное решение задачи, уже при небольшом числе городов решение за допустимое количество времени невозможно.[8].

1.3 Решение с помощью муравьиного алгоритма

В то время как простой метод перебора всех вариантов чрезвычайно неэффективный при большом количестве городов, эффективными признаются решения, гарантирующие получение ответа за время, ограниченное полиномом от размерности задачи. С помощью муравьиных алгоритмов находятся субоптимальные решения, локальные минимумы целевой функции, приближающиеся к абсолютному минимуму[4].

В основе алгоритма лежит поведение муравьиной колонии — маркировка более удачных путей большим количеством феромона. Рассмотрим биологическую модель поведения такой колонии.

В реальном мире муравьи (первоначально) ходят в случайном порядке и по нахождении продовольствия возвращаются в свою колонию, прокладывая феромонами тропы. Если другие муравьи находят такие тропы, они, вероятнее всего, пойдут по ним. Вместо того, чтобы отслеживать цепочку, они укрепляют её при возвращении, если в конечном итоге находят источник питания. Со временем феромонная тропа начинает испаряться, тем самым уменьшая свою привлекательную силу. Чем больше времени требуется для прохождения пути до цели и обратно, тем сильнее испарится феромонная тропа. На коротком пути, для сравнения, прохождение будет более быстрым, и, как следствие, плотность феромонов остаётся высокой. Испарение феромонов также имеет функцию избежания стремления к локально-оптимальному решению. Если бы феромоны не испарялись, то путь, выбранный первым, был бы самым привлекательным. В этом случае, исследования пространственных решений были бы ограниченными. Таким образом, когда один муравей находит (например, короткий) путь от колонии до источника пищи, дру-

гие муравьи, скорее всего пойдут по этому пути, и положительные отзывы в конечном итоге приводят всех муравьёв к одному, кратчайшему, пути. Этапы работы муравьиной колонии представлены на рис. 2.1.

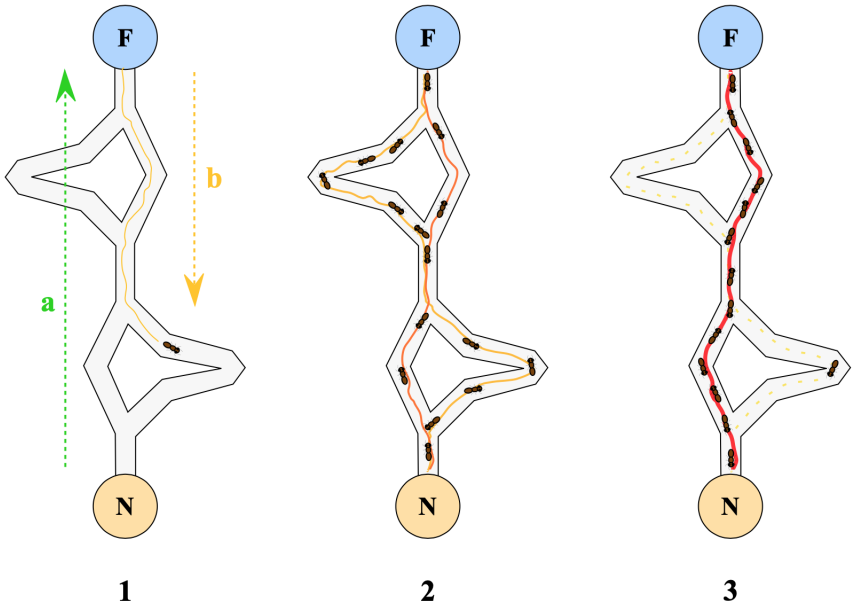

Рис 2.1: Работа муравьиной колонии

Оригинальная идея исходит от наблюдения за муравьями в процессе поиска кратчайшего пути от колонии до источника питания.

1. Первый муравей находит источник пищи (F) любым способом (a), а затем возвращается к гнезду (N), оставив за собой тропу из феромонов (b).

2. Затем муравьи выбирают один из четырёх возможных путей, затем укрепляют его и делают привлекательным.

3. программно реализовать описанные выше алгоритмы;

4. Муравьи выбирают кратчайший маршрут, так как феромоны с более длинных путей быстрее испаряются.

Теперь рассмотрим каждый шаг в цикле более подробно:

1. Создание муравьев

Стартовая точка, куда помещается муравей, зависит ограничений, накладываемых условиями задачи. Потому что для каждой задачи способ размещения муравьёв является определяющим. Либо все они помещаются в одну точку, либо в разные с повторения, либо без повторений.
На этом же этапе задается начальный уровень феромона. Он инициализируется небольшим положительным числом для того, чтобы на начальном шаге вероятности перехода в следующую вершину не были нулевыми.

2. Поиск решения

Вероятность перехода из вершины i в вершину j определяется по следующей формуле1.1

$$p_{i,j} = \frac{(\tau_{i,j}^{\alpha})(\eta_{i,j}^{\beta})}{\sum(\tau_{i,j}^{\alpha})(\eta_{i,j}^{\beta})} \tag{1.1}$$

где $\tau_{i,j}$− расстояние от города i до j;

$\eta_{i,j}$−количество феромонов на ребре ij;

α− параметр влияния длины пути;

β− параметр влияния феромона.

3. Обновление феромона

Уровень феромона обновляется в соответствии с приведённой формулой:
После того, как муравей успешно проходит маршрут, он оставляет на всех пройденных ребрах след, обратно пропорциональный длине пройденного пути. Итого, новый след феромона вычисляется по формуле 1.2:

$$\tau_{i,j} = (1 - \rho)\tau_{i,j} + \Delta\tau_{i,j}, \tag{1.2}$$

где $\rho_{i,j}$ - доля феромона, который испарится;

$\tau_{i,j}$ - количество феромона на дуге ij;

$\Delta\tau_{i,j}$ - количество отложенного феромона, вычисляется по формуле 1.4.

1.4 Формализация задачи коммивояжера в терминах подхода муравьиных алгоритмов

Рассмотрим, как реализовать четыре составляющие самоорганизации муравьев при оптимизации маршрута коммивояжера. Многократность взаимодействия реализуется итерационным поиском маршрута коммивояжера одновременно несколькими муравьями. При этом каждый муравей рассматривается как отдельный, независимый коммивояжер, решающий свою задачу. За одну итерацию алгоритма каждый муравей совершает полный маршрут коммивояжера.Положительная обратная связь реализуется как имитация поведения муравьев типа «оставление следов – перемещение по следам». Чем больше следов оставлено на тропе — ребре графа в задаче коммивояжера, тем больше муравьев будет передвигаться по ней. При этом на тропе появляются новые следы, привлекающие дополнительных муравьев. Для задачи коммивояжера положительная обратная связь реализуется следующим стохастическим правилом: вероятность включения ребра графа в маршрут муравья пропорциональна количеству феромона на нем.

Теперь с учетом особенностей задачи коммивояжёра, можно описать локальные правила поведения муравьев при выборе пути.

1. Муравьи имеют собственную «память». Поскольку каждый город может быть посещён только один раз, то у каждого муравья есть список уже посещенных городов - список запретов. Обозначим через J список городов, которые необходимо посетить муравью k , находящемуся в городе i .

2. 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.

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

4. На этом основании мы можем сформулировать вероятностно-пропорциональное правило, определяющее вероятность перехода k-ого муравья из города i

в город j.

5. Пройдя ребро (i, j) , муравей откладывает на нём некоторое количество феромона, которое должно быть связано с оптимальностью сделанного выбора. Пусть $T_k(t)$ есть маршрут, пройденный муравьем k к моменту времени t , $L_k(t)$ - длина этого маршрута, а Q - параметр, имеющий значение порядка длины оптимального пути. Тогда откладываемое количество феромона может быть задано в виде:

$$\Delta\tau_{i,j}^k = \{\, Q\ /L_k\text{Если k-ый мурваей прошел по ребру ij;0Иначе} \qquad (1.3)$$

где Q - количество феромона, переносимого муравьем;

Тогда

$$\Delta\tau_{i,j} = \tau_{i,j}^0 + \tau_{i,j}^1 + ... + \tau_{i,j}^k \qquad (1.4)$$

где k - количество муравьев в вершине графа с индексами i и j.

1.5 Вывод

В данном разделе были рассмотрены теоретические сведения, необходимые для программной реализации муравьиного алгоритма, примененного к задаче коммивояжера..

2 | Конструкторская часть

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

Схема реализации алгоритма полного перебора представлена на рис. 2.2.

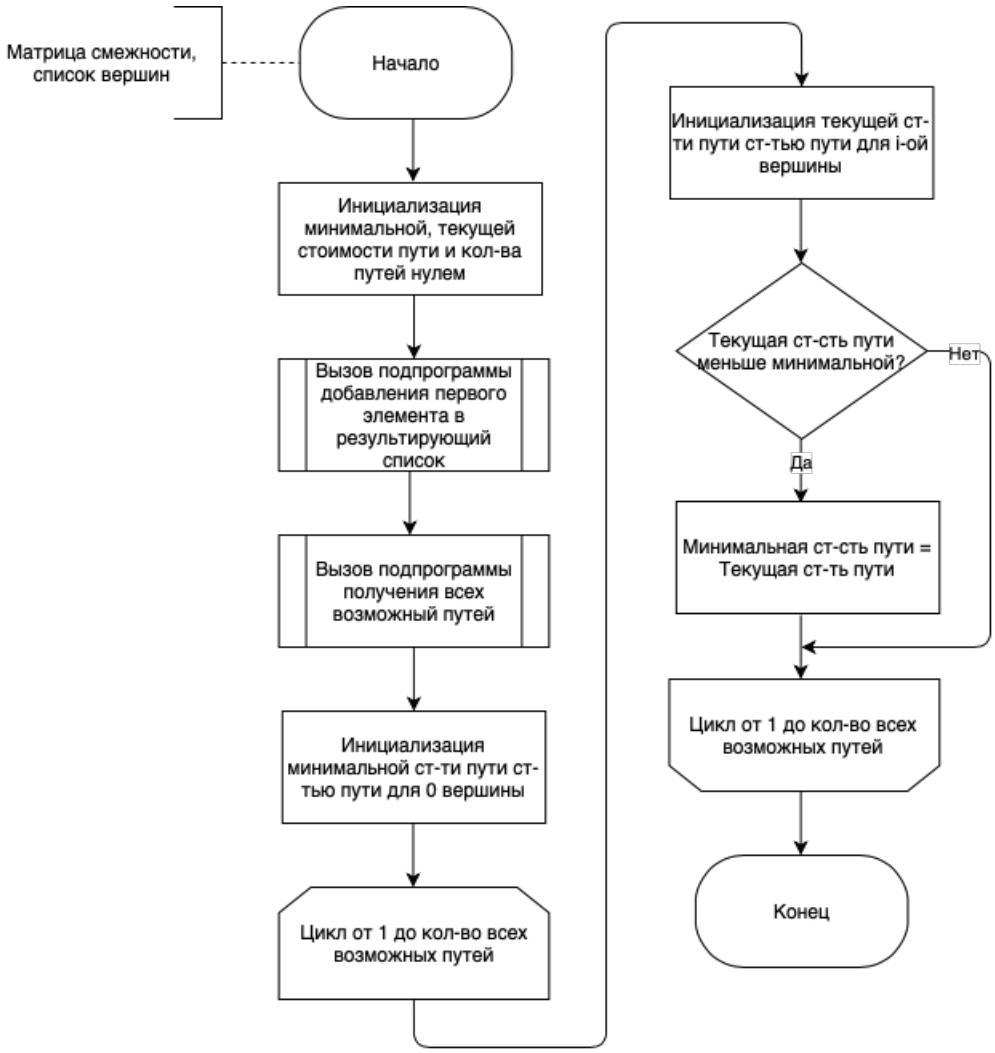

Рис 2.2: Схема реализации алгоритма полного перебора

Схема реализации муравьиного алгоритма представлена на рис. 2.3.

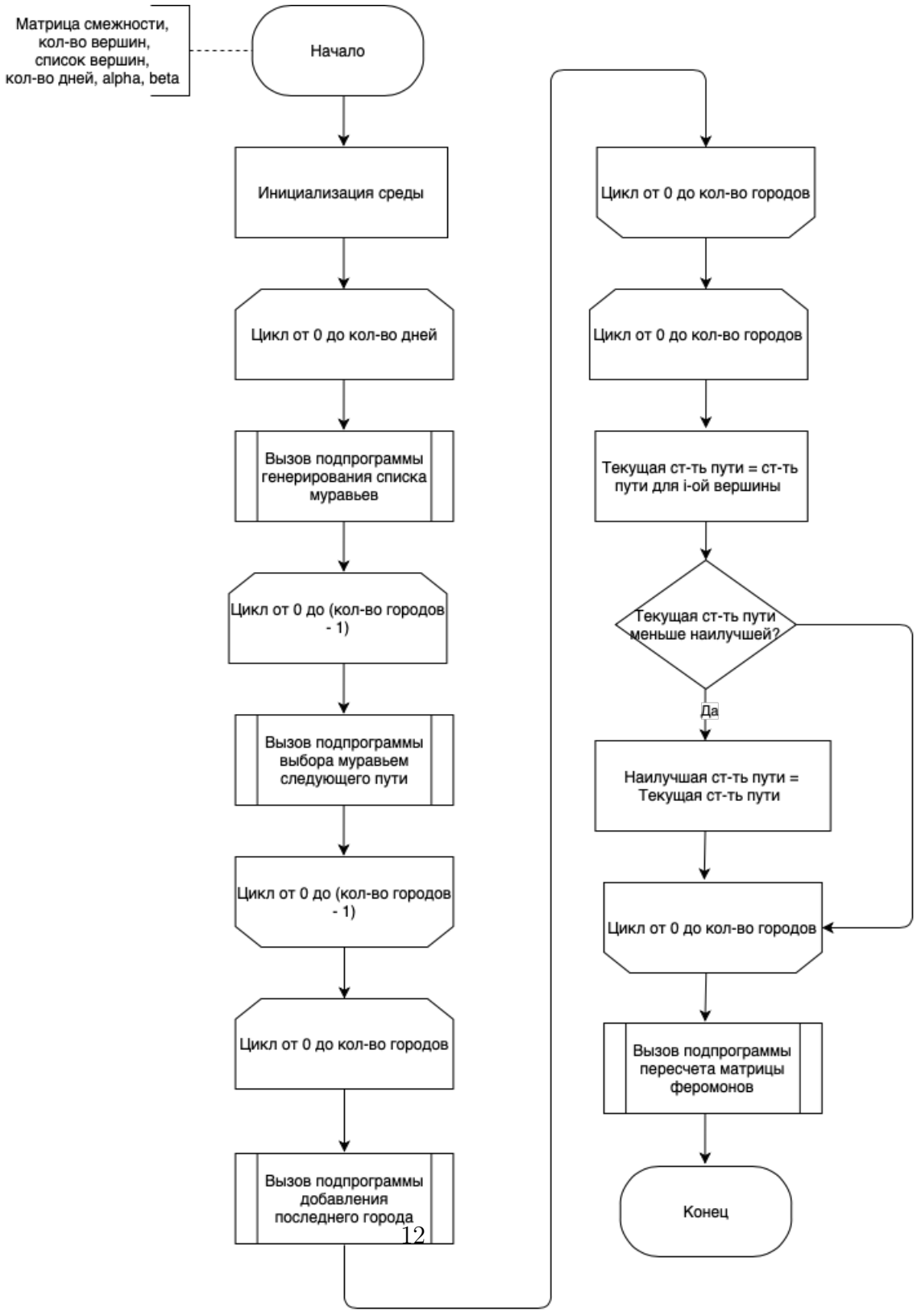
Матрица смежности,
кол-во вершин,
список вершин,
кол-во дней, alpha, beta
Начало
Инициализация среды
Цикл от 0 до кол-во дней
Вызов подпрограммы генерирования списка муравьев
Цикл от 0 до (кол-во городов - 1)
Вызов подпрограммы выбора муравьем следующего пути
Цикл от 0 до (кол-во городов - 1)
Цикл от 0 до кол-во городов
Вызов подпрограммы добавления последнего города
Цикл от 0 до кол-во городов
Цикл от 0 до кол-во городов
Текущая ст-ть пути = ст-ть пути для i-ой вершины
Текущая ст-ть пути меньше наилучшей?
Да
Наилучшая ст-ть пути = Текущая ст-ть пути
Цикл от 0 до кол-во городов
Вызов подпрограммы пересчета матрицы феромонов
Конец

Рис 2.3: Схема реализации муравьиного алгоритма

2.1 Вывод

В данном разделе были рассмотрены 2 схемы: схема программной реализации алгоритма полного перебора и схема программной реализации муравьиного алгоритма.

3 | Технологическая часть

3.1 Выбор языка программирования

В данной лабораторной работе использовался язык программирования - C [2], так как данный язык программирования предоставляет удобные библиотеки и инструменты для работы со структурами данных и обеспечивает хорошую производительность программного продукта. В качестве интегрированной среды разработки использовалась Visual studio [1].

3.2 Сведения о модулях программы

Программа состоит из следующих модулей:

- main.c - главный файл программы, в котором располагается точка входа в программу;
- ant_algorithm.c - реализация муравьиного алгоритма;
- brute_force.c - реализация алгоритма полного перебора;
- matrix.c - модуль для работы с матрицей смежности;
- array.c - модуль для организации списка;
- city.c - модуль для организации списка городов;
- parser.c - модуль для представления результата в графическом виде.

На листинге 3.1 представлена подпрограмма точки входа main().

Листинг 3.1: Код подпрограммы main()

```
1
2  int main(int argc, char *argv[])
3  {
4      srand(time(NULL));
5
6      array cities;
7
8      char city_names[LEN];
9      int matrix[LEN][LEN];
10
11     FILE *f = fopen(argv[1], "r");
12
13     if (!f)
14     {
15         printf("Error.\n");
16         return ERROR_OPEN_FILE;
17     }
18
19     int count = parser(f, city_names, matrix);
20
21     fclose(f);
22
23     create_cities_array(&cities, count);
24
25     print_cities(cities, city_names);
26     print_matrix(matrix, count);
27
28     array result = get_shortest_path(cities, matrix);
29     int min_path_simple = get_path_cost(result, matrix);
30     print_array(result, "result");
31     red();
32     printf("result simple = %d\n", min_path_simple);
33
34     array result_ant = ant_algorithm(matrix, count, cities,
           40, 0.4, 0.7, 0.3);
35     int min_path_ant = get_path_cost(result_ant, matrix);
36     print_array(result_ant, "result ant");
37     red();
38     printf("result ant = %d\n", min_path_ant);
```

```
39
40      return OK;
41  }
```

На листинге 3.2 представлена реализация муравьиного алгоритма.

Листинг 3.2: Реализация муравьиного алгоритма

```
1  int calculate_Q(int matrix[LEN][LEN], int count)
2  {
3    int Q = 0;
4
5    for (int i = 0; i < count; i++)
6      for (int j = 0; j < i; j++)
7        Q += matrix[i][j];
8
9    return Q * 2;
10 }
11
12 void add_pheromones(int matrix[LEN][LEN], float
       matrix_pheromones[LEN][LEN], int count, int Q, ant ants[
       ANTS_MAX_COUNT])
13 {
14   int city_first, city_second;
15   int curr_cost;
16   float delta_tao = 0;
17
18   for (int i = 0; i < count; i++)
19   {
20     curr_cost = get_path_cost(ants[i].way, matrix);
21
22     delta_tao += (float)Q / curr_cost;
23
24     // printf("Q = %d; curr_cost = %d; delta_tao = %f\n", Q
           , curr_cost, delta_tao);
25   }
26
27     for (int i = 0; i < count; i++)
28   {
29     for (int j = 0; j < ants[i].way.count - 1; j++)
30     {
31       city_first = ants[i].way.arr[j];
```

```
32        city_second = ants[i].way.arr[j + 1];
33
34        matrix_pheromones[city_first][city_second] =
              matrix_pheromones[city_first][city_second] +
              delta_tao;
35        matrix_pheromones[city_second][city_first] =
              matrix_pheromones[city_second][city_first] +
              delta_tao;
36      }
37      // printf("\n");
38    }
39  }
40
41  void correct_pheromones(float matrix_pheromones[LEN][LEN],
        int count)
42  {
43    for (int i = 0; i < count; i++)
44      for (int j = 0; j < i; j++)
45        if (matrix_pheromones[i][j] <= 0.1)
46          matrix_pheromones[i][j] = matrix\_pheromones[j][i]
              = 0.1;
47  }
48
49  void evaporation(float matrix_pheromones[LEN][LEN], int
        count, float p)
50  {
51    float tmp = 1 - p;
52    for (int i = 0; i < count; i++)
53      for (int j = 0; j < i; j++)
54      {
55        matrix_pheromones[i][j] = tmp * matrix_pheromones[i][
              j];
56        matrix_pheromones[j][i] = tmp * matrix_pheromones[j][
              i];
57      }
58  }
59
60  void print_matrix_pheromones(float matrix_pheromones[LEN][
        LEN], int count)
61  {
```

```
62    for (int i = 0; i < count; i++)
63    {
64      for (int j = 0; j < count; j++)
65        printf("%f ", matrix_pheromones[i][j]);
66      printf("\n");
67    }
68  }
69
70  array ant_algorithm(int matrix[LEN][LEN], int count, array
        cities, int tmax, float p, float alpha, float beta)
71  {
72    int Q = calculate_Q(matrix, count);
73
74    array best_way = copy_arr(cities);
75    add_elem(&best_way, get_elem(best_way, 0));
76
77    int best_cost = get_path_cost(best_way, matrix);
78    int curr_cost = 0;
79
80    float matrix_pheromones[LEN][LEN];
81    fill_matrix(matrix_pheromones, count, PHEROMONE\_MIN);
82
83    ant ants[ANTS_MAX_COUNT];
84
85    for (int t = 0; t < tmax; t++)
86    {
87      generate_ants_array(ants, count);
88
89      for (int i = 0; i < count - 1; i++)
90      {
91        ants_choose_way(ants, matrix_pheromones, matrix,
              count, alpha, beta);
92      }
93
94      for (int i = 0; i < count; i++)
95        add_elem(&ants[i].way, get_elem(ants[i].way, 0));
96
97      for (int i = 0; i < count; i++)
98      {
99        curr_cost = get_path_cost(ants[i].way, matrix);
```

```
100         if (curr_cost < best_cost)
101         {
102           best_cost = curr_cost;
103           best_way = copy_arr(ants[i].way);
104         }
105       }
106
107       evaporation(matrix_pheromones, count, p);
108       add_pheromones(matrix, matrix_pheromones, count, Q,
              ants);
109       correct_pheromones(matrix_pheromones, count);
110     }
111
112     return best_way;
113 }
```

На листинге 3.3 представлена реализация алгоритма полного перебора.

Листинг 3.3: Реализация алгоритма полного перебора

```
1  int get_path_cost(array cities, int matrix[LEN][LEN])
2  {
3    int cost = 0;
4
5    for (int i = 0; i < cities.count - 1; i++)
6      cost += matrix[cities.arr[i]][cities.arr[i + 1]];
7
8    return cost;
9  }
10
11 array get_shortest_path(array cities, int matrix[LEN][LEN])
12 {
13   array result[DEPTH_OF_RECURSION];
14   array res_arr;
15
16   int min_cost;
17   int curr_cost;
18   int index = 0;
19
20   int routes_count = 0;
21
```

```
22    del_elem(&cities, 0);
23    add_elem(&res_arr, 0);
24    get_routes(&cities, &res_arr, result, &routes_count);
25
26    min_cost = get_path_cost(result[index], matrix);
27
28    for (int i = 1; i < routes_count; i++)
29    {
30      curr_cost = get_path_cost(result[i], matrix);
31
32      if (curr_cost < min_cost)
33      {
34        min_cost = curr_cost;
35        index = i;
36      }
37    }
38
39    return result[index];
40  }
41
42  void get_routes(array *cities, array *res_arr, array result
      [DEPTH_OF_RECURSION], int *count)
43  {
44    int elem;
45    array tmp;
46
47    if (!cities->count)
48    {
49      add_elem(res_arr, get_elem(*res_arr, 0));
50      result[*count] = *res_arr;
51      (*count)++;
52      del_elem(res_arr, res_arr->count - 1);
53    }
54
55    for (int i = 0; i < cities->count; i++)
56    {
57      elem = get_elem(*cities, i);
58
59      add_elem(res_arr, elem);
60
```

```
61      tmp = copy_arr(*cities);
62      del_elem(&tmp, i);
63
64      get_routes(&tmp, res_arr, result, count);
65
66      del_elem(res_arr, res_arr->count - 1);
67    }
68  }
```

На листинге 3.4 представлены подпрограммы инициализации и вывода матрицы смежности.

Листинг 3.4: Подпрограммы инициализации и вывода матрицы смежности

```
1  public static int FindMin(intPtr array)
2  void print_matrix(int matrix[LEN][LEN], int const count)
3  {
4    blue();
5    for (int i = 0; i < count; i++)
6      printf("%d\t", i);
7    puts("");
8
9    for (int i = 0; i < count; i++)
10   {
11     printf("%d ", i);
12     for (int j = 0; j < count; j++)
13       printf("%d\t", matrix[i][j]);
14     puts("");
15   }
16   puts("");
17 }
18
19 void fill_matrix(float matrix[LEN][LEN], int count, float
       num)
20 {
21   for (int i = 0; i < count; i++)
22     for (int j = 0; j < i; j++)
23       matrix[i][j] = matrix[j][i] = num;
24 }
```

На листинге 3.5 представлены подпрограммы организации списка вершин (городов).

Листинг 3.5: Подпрограммы организации списка вершин (городов)

```
1  void print_array(array cities, char msg[MSG\_LEN])
2  {
3    green();
4    printf("%s\ncount: %d\n", msg, cities.count);
5    yellow();
6    for (int i = 0; i < cities.count; i++)
7      printf("%d  ", cities.arr[i]);
8    puts("\n");
9    blue();
10 }
11
12 void add_elem(array *cities, const int elem)
13 {
14   cities->arr[cities->count] = elem;
15   (cities->count)++;
16 }
17
18 int get_elem(array array, int const index)
19 {
20   if (index > array.count)
21     return NOT_FOUND;
22
23   return array.arr[index];
24 }
25
26 int del_elem(array *array, int const index)
27 {
28   if (index > array->count)
29     return NOT_FOUND;
30
31   int tmp, elem = array->arr[index];
32
33   for (int i = index; i < array->count; i++)
34   {
35     tmp = array->arr[i];
36     array->arr[i] = array->arr[i + 1];
```

```
37      array->arr[i + 1] = tmp;
38    }
39
40    (array->count)--;
41
42    return elem;
43  }
44
45  array copy_arr(array arr)
46  {
47    array result;
48    result.count = arr.count;
49
50    for (int i = 0; i < arr.count; i++)
51      result.arr[i] = arr.arr[i];
52
53    return result;
54  }
```

3.3 Вывод

В технологической части были представлены модули программы, листинги кода, а также обусловлен выбор языка программирования и приведены использовавшиеся в ходе работы инструменты.

4 | Исследовательская часть

В этом разделе будет проведен сравнительный анализ характеристик полученного программного продукта.

4.1 Характеристики ЭВМ

- MacBook Pro (Retina, 15-inch, Mid 2014).
- 2,5 GHz Intel Core i7.
- Число логических ядер: 8.

4.2 Временные характеристики

Результаты замеров времени приведены в таблице 4.1.

Размерность матрицы	Полный перебор	Муравьиный алгоритм
3	1612	89996
4	1441	151542
5	2319	286035
6	7725	469782
7	43539	710305
8	305564	1039688
9	2500918	1470419
10	22690076	1886565

На рис. 2.4 приведен графики зависимостей времени работы алгоритмов от размерности матрицы смежности.

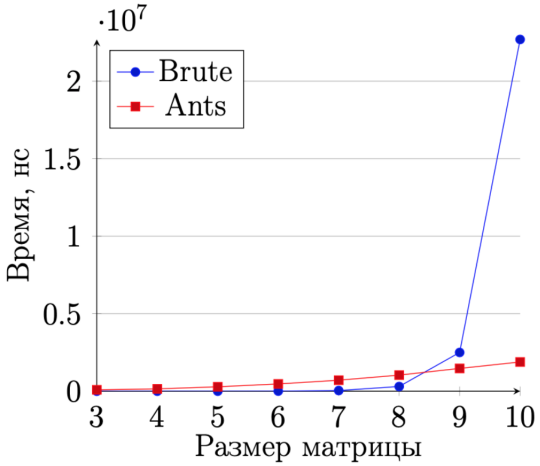

Рис 2.4: Зависимость времени работы алгоритмов от размерности матрицы смежности

В муравьином алгоритме вычисления производятся на основе настраиваемых параметров. Рассмотрим два класса данных и подберем к ним параметры, при которых метод даст точный результат при небольшом количестве итераций.

Будем рассматривать матрицы размерности 10×10, так как иначе получение точного результата алгоритмом велико и менее зависимо от параметров.

В качестве первого класса данных выделим матрицу смежности, в которой все значения незначительно отличаются друг от друга, например, в диапазоне $[1, 25]$. Вторым классом будут матрицы, где значения могут значительно отличаться, например $[1, 2500]$.

Будем запускать муравьиный алгоритм для всех значений $\alpha, \rho \in [0, 1]$, с шагом $= 0.1$, пока не будет найдено точное значение для каждого набора.

В результате тестирования будет выведена таблица со значениями $\alpha, \beta, \rho, , ,$ где Длина — оптимальная длина пути, найденная муравьиным алгоритмом, Разница - Разность между полученным значением и настоящей оптимальной длиной пути, а α, β, ρ — настраиваемые параметры.

4.2.1 Класс данных

0	21	21	4	22	20	6	4	11	23
21	0	22	6	9	21	16	6	2	7
21	22	0	18	23	7	19	19	1	14
4	6	18	0	7	11	16	10	8	19
22	9	23	7	0	15	2	23	1	22
20	21	7	11	15	0	6	7	8	21
6	16	19	16	2	6	0	8	3	14
4	6	19	10	23	7	8	0	23	15
11	2	1	8	1	8	3	23	0	9
23	7	14	19	22	21	14	15	9	0

В таблице 4.1 приведены результаты параметризации метода решения задачи коммивояжера на основании муравьиного алгоритма. Количество дней было взято равным 50. Полный перебор определил оптимальную длину пути 52. Два последних стобца таблицы определяют найденный муравьиным алгоритм оптимальный путь и разницу этого пути с оптимальным путём, найденным алгоритмом полного перебора.

Таблица 4.1: Таблица коэффициентов для класса данных 1.

α	β	ρ	Длина	Разница
0	1	0	52	0
0	1	0.1	52	0
0	1	0.2	52	0
0	1	0.3	52	0
0	1	0.4	53	1
0	1	0.5	52	0
0	1	0.6	52	0
0	1	0.7	52	0
0	1	0.8	52	0
0	1	0.9	52	0
0	1	1	52	0
0.1	0.9	0	52	0
0.1	0.9	0.1	52	0
0.1	0.9	0.2	53	1
0.1	0.9	0.3	52	0
0.1	0.9	0.4	52	0
0.1	0.9	0.5	52	0
0.1	0.9	0.6	53	1
0.1	0.9	0.7	52	0
0.1	0.9	0.8	52	0
0.1	0.9	0.9	52	0
0.1	0.9	1	52	0
0.2	0.8	0	52	0
0.2	0.8	0.1	52	0
0.2	0.8	0.2	52	0
0.2	0.8	0.3	53	1
0.2	0.8	0.4	52	0
0.2	0.8	0.5	52	0
0.2	0.8	0.6	53	1
0.2	0.8	0.7	53	1
0.2	0.8	0.8	52	0
0.2	0.8	0.9	52	0
0.2	0.8	1	52	0
0.3	0.7	0	53	1
0.3	0.7	0.1	52	0
0.3	0.7	0.2	52	0
0.3	0.7	0.3	53	1
0.3	0.7	0.4	52	0
0.3	0.7	0.5	52	0
0.3	0.7	0.6	52	0
0.3	0.7	0.7	52	0
0.3	0.7	0.8	52	0
0.3	0.7	0.9	53	1
0.3	0.7	1	52	0
0.4	0.6	0	53	1
0.4	0.6	0.1	53	1
0.4	0.6	0.2	53	1
0.4	0.6	0.3	52	0
0.4	0.6	0.4	53	1
0.4	0.6	0.5	52	0
0.4	0.6	0.6	52	0
0.4	0.6	0.7	52	0
0.4	0.6	0.8	52	0
0.4	0.6	0.9	52	0
0.4	0.6	1	52	0
0.5	0.5	0	52	0
0.5	0.5	0.1	52	0
0.5	0.5	0.2	52	0
0.5	0.5	0.3	53	1
0.5	0.5	0.4	52	0
0.5	0.5	0.5	52	0
0.5	0.5	0.6	52	0
0.5	0.5	0.7	52	0
0.5	0.5	0.8	52	0
0.5	0.5	0.9	52	0
0.5	0.5	1	52	0
0.6	0.4	0	53	1
0.6	0.4	0.1	53	1
0.6	0.4	0.2	56	4
0.6	0.4	0.3	52	0
0.6	0.4	0.4	52	0
0.6	0.4	0.5	55	3
0.6	0.4	0.6	56	4
0.6	0.4	0.7	52	0
0.6	0.4	0.8	53	1
0.6	0.4	0.9	53	1
0.6	0.4	1	52	0
0.7	0.3	0	52	0
0.7	0.3	0.1	53	1
0.7	0.3	0.2	52	0

α	β	ρ	Длина	Разница
0.7	0.3	0.3	52	0
0.7	0.3	0.4	53	1
0.7	0.3	0.5	53	1
0.7	0.3	0.6	52	0
0.7	0.3	0.7	53	1
0.7	0.3	0.8	57	5
0.7	0.3	0.9	52	0
0.7	0.3	1	52	0
0.8	0.2	0	59	7
0.8	0.2	0.1	53	1
0.8	0.2	0.2	56	4
0.8	0.2	0.3	53	1
0.8	0.2	0.4	52	0
0.8	0.2	0.5	56	4
0.8	0.2	0.6	53	1
0.8	0.2	0.7	52	0
0.8	0.2	0.8	52	0
0.8	0.2	0.9	52	0
0.8	0.2	1	53	1
0.9	0.1	0	56	4
0.9	0.1	0.1	53	1
0.9	0.1	0.2	52	0
0.9	0.1	0.3	56	4
0.9	0.1	0.4	52	0
0.9	0.1	0.5	53	1
0.9	0.1	0.6	56	4
0.9	0.1	0.7	56	4
0.9	0.1	0.8	55	3
0.9	0.1	0.9	53	1
0.9	0.1	1	53	1
1	0	0	71	19
1	0	0.1	61	9
1	0	0.2	53	1
1	0	0.3	59	7
1	0	0.4	59	7
1	0	0.5	60	8
1	0	0.6	60	8
1	0	0.7	74	22

4.3 Вывод

В результате сравнения алгоритма полного перебора и муравьиного алгоритма по времени из таблицы 4.1 были получены следующие результаты:

- при относительно небольших размерах матрицы смежности (а именно от 3 до 8) алгоритм полного перебора работает значительно быстрее (при размере 3 — $\approx$ в 50 раз, при размере 6 — $\approx$ в 60 раз);
- при размерах матрицы смежности 9 и выше, время работы алгоритма полного перебора начинает резко возрастать, и становится при размере 9 на 78% медленнее муравьиного алгоритма, а на размере 10 алгоритм полного перебора работает $\approx$ в 12 раз дольше, нежели муравьиный алгоритм.

На основе проведенной параметризации класса данных можно сделать следующий вывод: для класса данных, содержащего приблизительно равные значения, наилучшими наборами стали ($\alpha = 0.5$, $\beta = 0.5$, $\rho =$ любое), так как они показали наиболее стабильные результаты, равные эталонному значению оптимального пути, равного 52 единицам.

Заключение

В ходе лабораторной работы был изучен муравьиный алгоритм и были приобретены навыки параметризации методов на примере муравьиного алгоритма, примененного к задаче коммивояжера.

В рамках выполнения работы решены следующие задачи:

1. изучен алгоритм полного перебора для решения задачи коммивояжера;
2. изучен муравьиный алгоритм для решения задачи коммивояжера;
3. программно реализованы описанные выше алгоритмы;
4. проведена параметризация муравьиного алгоритма на классе данных;
5. проведен сравнительный анализ скорости работы реализованных алгоритмов.

Литература

[1] Visual Studio [Электронный ресурс], режим доступа:https://visualstudio.microsoft.com/ru/ (дата обращения: 01.10.2020)

[2] Справочник по языку C [Электронный ресурс], режим доступа:https://docs.microsoft.com/ru-ru/cpp/c-language/c-language-reference?view=msvc-160 (дата обращения: 15.12.2020)

[3] Random Класс [Электронный ресурс], режим доступа:https://docs.microsoft.com/ru-ru/dotnet/api/system.random?view=netcore-3.1 (дата обращения: 02.12.2020)

[4] Муравьиные алгоритмы [Электронный ресурс], режим доступа:http://www.machinelearning.ru/wiki/index.php?title=Муравьиные_алгоритмы (дата обращения: 20.12.2020)

[5] Optimization, Learning and Natural Algorithms [Электронный ресурс], режим доступа:https://link.springer.com/chapter/10.1007/0-306-48056-5_9 (дата обращения: 21.12.2020)

[6] Лекция 10. Муравьиные алгоритмы [Электронный ресурс], режим доступа:http://hpc-education.ru/files/lectures/2011/ershov/ershov_2011_lectures10.pdf (дата обращения: 21.12.2020)

[7] Коммивояжер [Электронный ресурс], режим доступа:https://kartaslov.ru/значение-слова/коммивояжёр (дата обращения: 23.12.2020)

[8] Решение задачи коммивояжера полным перебором [Электронный ресурс], режим доступа:http://mech.math.msu.su/ shvetz/54/inf/perl-problems/chCommisVoyageur.xhtml (дата обращения: 20.12.2020)