МГТУ им. Н.Э.Баумана

Лабораторная работа №6

По курсу: "Анализ алгоритмов"

Муравьиный алгоритм

Работу выполнил: Луговой Дмитрий, ИУ7-51Б

Преподаватель: Волкова Л.Л.

Москва, 2019

Оглавление

Введение

Задача коммивояжера формулируется как задача поиска минимального по стоимости замкнутого маршрута по всем вершинам без повторений на полном взвешенном графе с n вершинами. Содержательно вершины графа являются городами, которые должен посетить коммивояжер, а веса ребер отражают расстояния (длины) или стоимости проезда.

Цель работы: изучить муравьиный алгоритм по материалам решения задачи коммивояжера.

Задачи работы

Задачами данной лабораторной являются:

1) Описать методы решения.
2) Реализовать алгоритм и описать полученные результаты.
3) Выбрать класс данных и составить набор данных.
4) Провести параметризацию метода на основе муравьиного алгоритма для выбранного класса данных.
5) Провести сравнительный анализ двух методов.
6) Дать рекомендации о применимости метода решения задачи коммивояжера на основе муравьиного алгоритма.

1 Аналитическая часть

В данном разделе содержится описание задачи коммивояжера и методы её решения.

1.1 Задача коммивояжера

В общем случае задача коммивояжера (странствующего торговца) может быть сформулирована следующим образом: найти самый выгодный (самый короткий, самый дешевый, и т.п.) маршрут, начинающийся в исходном городе и проходящий ровно один раз через каждый из указанных городов, с последующим возвратом в исходный город.

Проблему коммивояжера можно представить в виде модели на графе, то есть, используя вершины и ребра между ними. Таким образом, M вершин графа соответствуют M городам, а ребра (i, j) между вершинами i и j — пути сообщения между этими городами. Каждому ребру (i, j) можно сопоставить критерий выгодности маршрута $c_{ij} \geq 0$,который можно понимать как, например, расстояние между городами, время или стоимость поездки. В целях упрощения задачи и гарантии существования маршрута обычно считается, что модельный граф задачи является полностью связным, то есть, что между произвольной парой вершин существует ребро.

Гамильтоновым циклом называется маршрут, включающий ровно по одному разу каждую вершину графа. Таким образом, решение задачи коммивояжёра — это нахождение гамильтонова цикла минимального веса в полном взвешенном графе.

1.2 Стандартный алгоритм

Пусть дано M - число городов, D - матрица смежности, каждый элемент которой - вес пути из одного города в другой. Существует метод грубой силы решения поставленной задачи, а именно полный перебор всех возможных гамильтоновых циклов в заданном графе с нахождением минимального по весу. Этот метод гарантированно даст идеальное решение (глобальный минимум по весу). Однако стоит учитывать, что сложность такого алгоритма составляет M! и время выполнения программы, реализующий такой подход, будет расти экспоненциально в зависимости от размеров входной матрицы.

1.3 Муравьиный алгоритм

Поскольку на практике чаще всего необходимо получить решение как можно быстрее, при этом требуемое решение не обязательно должно быть наилучшем, был разработан ряд методов, называемых эвристическими, которые решают поставленную задачу за гораздо меньшее время, чем метод полного перебора. В основе таких методов лежат принципы из окружающего мира, которые в дальнейшем могут быть формализованы.

Одним из таких методов является муравьиный метод. Он применим к решению задачи коммивояжера и основан на идее муравейника. Модель данного метода: у муравья есть 3 чувства:

- зрение (муравей может оценить длину ребра);
- обоняние (муравей может унюхать феромон - вещество, выделяемое муравьем, для коммуникации с другими муравьями);
- память (муравей запоминает свой маршрут).

Благодаря введению обоняния между муравьями возможен непрямой обмен информацией.

Введем вероятность $P_{k,ij}(t)$ выбора следующего города j на маршруте муравьем k, который в текущий момент времени t находится в городе i.

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

где

$\tau_{i,j}$ – феромон на ребре ij;

$\eta_{i,j}$ – привлекательность города j;

α – параметр влияния длины пути;

β – параметр влияния феромона.

Очевидно, что при $\beta = 0$ алгоритм превращается в классический жадный алгоритм, а при $\alpha = 0$ он быстро сойдется к некоторому субоптимальному решению. Выбор правильного соотношения параметров является предметом исследований, и в общем случае производится на основании опыта.

После того, как муравей успешно проходит маршрут, он оставляет на всех пройденных ребрах феромон, обратно пропорциональный длине пройденного пути:

$$\Delta\tau_{k,ij} = \frac{Q}{L_k} \tag{1.2}$$

где

Q – количество феромона, переносимого муравьем;

L_k – стоимость k-го пути муравья (обычно длина).

После окончания условного дня наступает условная ночь, в течение которой феромон испаряется с ребер с коэффициентом ρ. Количество феромона на следующий день вычисляется по следующей формуле:

$$\tau_{i,j}(t+1) = (1-\rho)\tau_{i,j}(t) + \Delta\tau_{i,j}(t), \tag{1.3}$$

где

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

$\tau_{i,j}(t)$ – количество феромона на дуге ij;

$\Delta\tau_{i,j}(t)$ – количество отложенного феромона.

Таким образом, псевдокод муравьиного алгоритма можно представить так:

1. Ввод матрицы расстояний D, количества городов M;
2. Инициализация параметров алгоритма — α β, Q, $tmax$, ρ;
3. Инициализация ребер — присвоение "привлекательности" η_{ij} и начальной концентрации феромона τ_{start};
4. Размещение муравьев в случайно выбранные города без совпадений;
5. Инициализация начального кратчайшего маршрута L_p = null и определение длины кратчайшего маршрута L_{min} = inf;
6. Цикл по времени жизни колонии t = 1, tmax;
   (a) Цикл по всем муравьям k = 1, M;
      i. Построить маршрут $T_k(t)$ по правилу 1.1 и рассчитать длину получившегося маршрута $L_k(t)$;
      ii. Обновить феромон на маршруте по правилу 1.2;
      iii. Если $L_k(t) < L_{min}$, то $L_{min} = L_k(t)$ и $L_p = T_k(t)$;
   (b) Конец цикла по муравьям;
   (c) Цикл по всем ребрам графа;
      i. Обновить следы феромона на ребре по правилу 1.3;
   (d) Конец цикла по ребрам;
7. Конец цикла по времени;
8. Вывести кратчайший маршрут L_p и его длину L_{min}.

1.4 Вывод

Таким образом, существуют две группы методов для решения задачи коммивояжера - точные и эвристические. К точным относится метод полного перебора, к эвристическим - муравьиный метод. Применение муравьиного алгоритма обосновано в тех случаях, когда необходимо быстро найти решение или когда для решения задачи достаточно получения первого приближения. В случае необходимости максимально точного решения используется алгоритм полного перебора.

2 Конструкторская часть

В этом разделе содержатся схемы алгоритмов решения задачи коммивояжера.

2.1 Схемы алгоритмов

На рисунках 2.1 и 2.2 представлена схема муравьиного алгоритма.

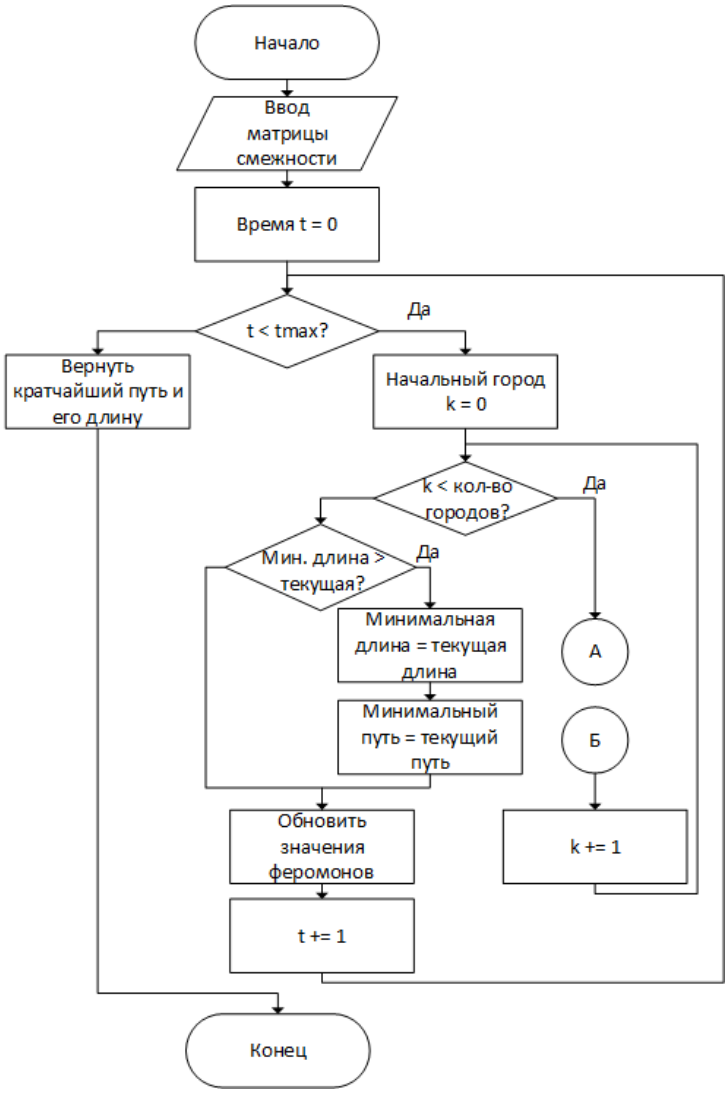

Рис. 2.1: Схема муравьиного алгоритма

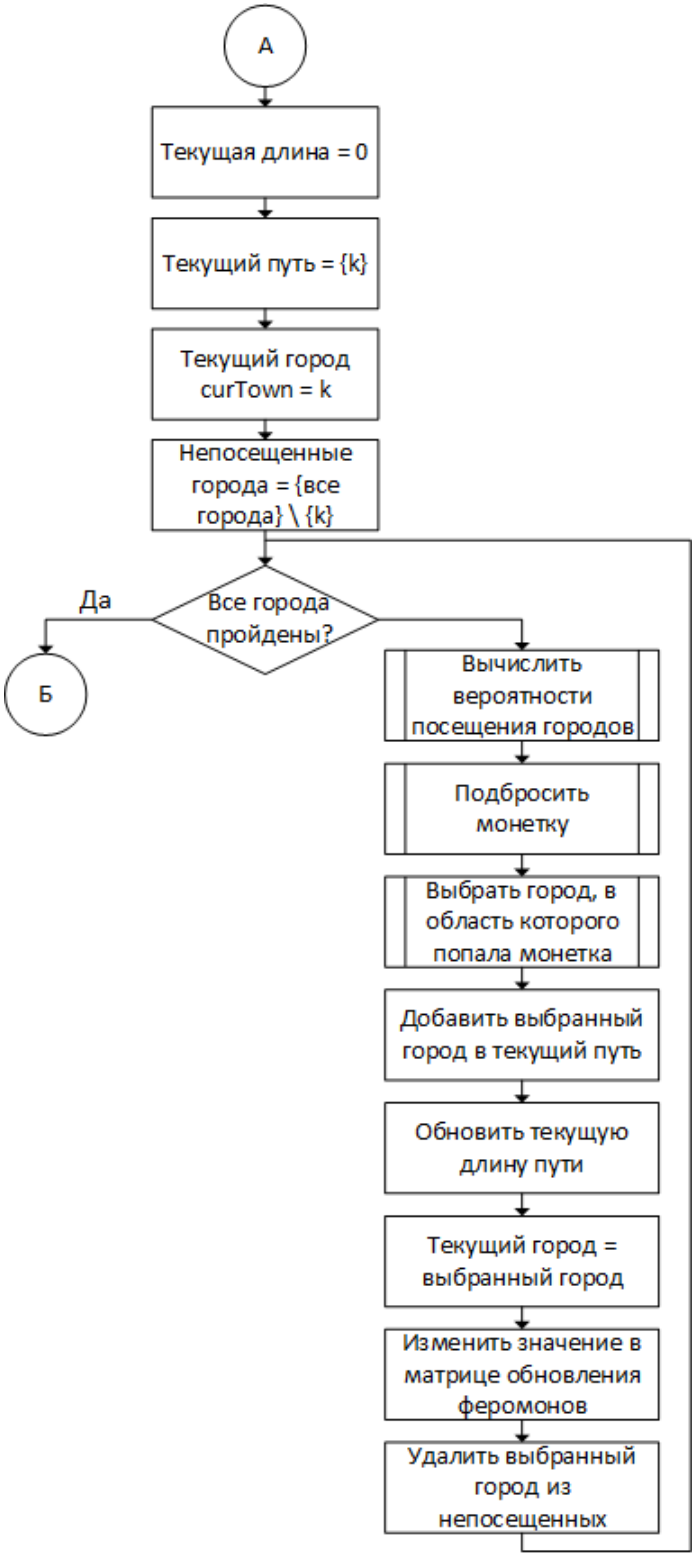

Рис. 2.2: Схема муравьиного алгоритма(продолжение)

На рисунке 2.3 представлена схема основной части алгоритма полного перебора.

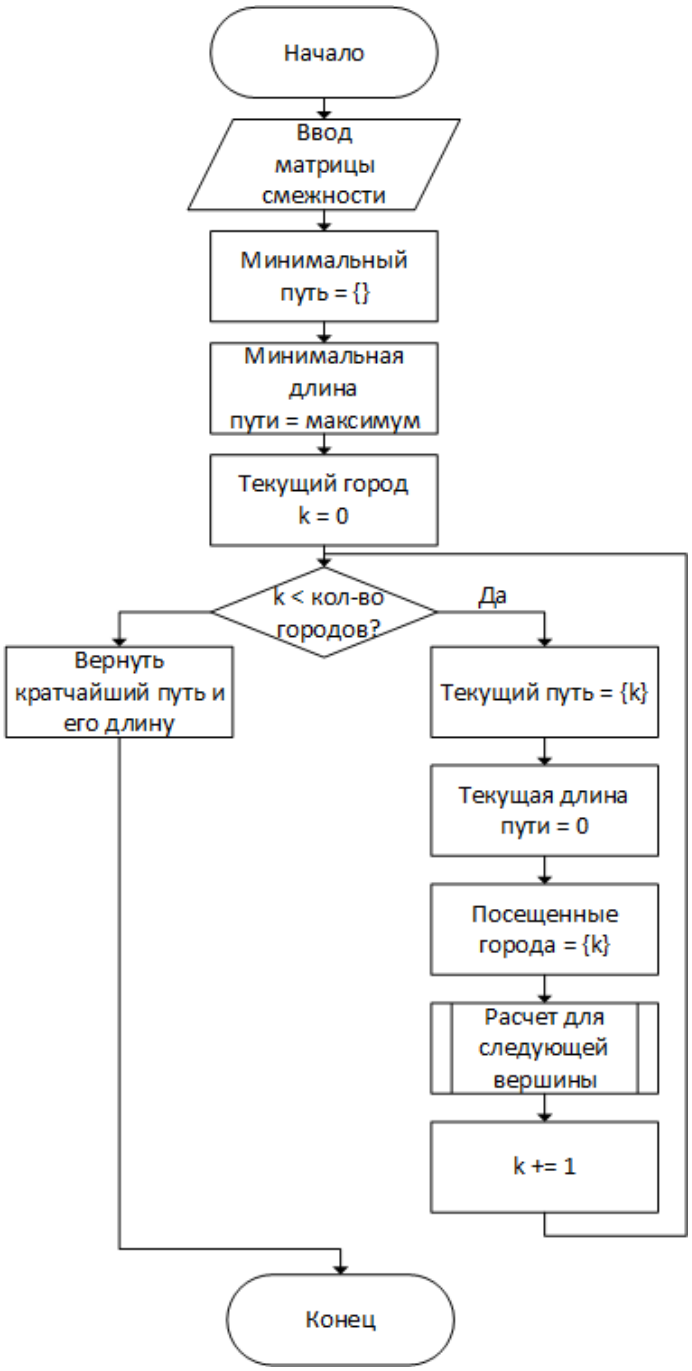

Рис. 2.3: Схема основной части алгоритма полного перебора

На рисунке 2.4 представлена схема рекурсивной части алгоритма полного перебора.

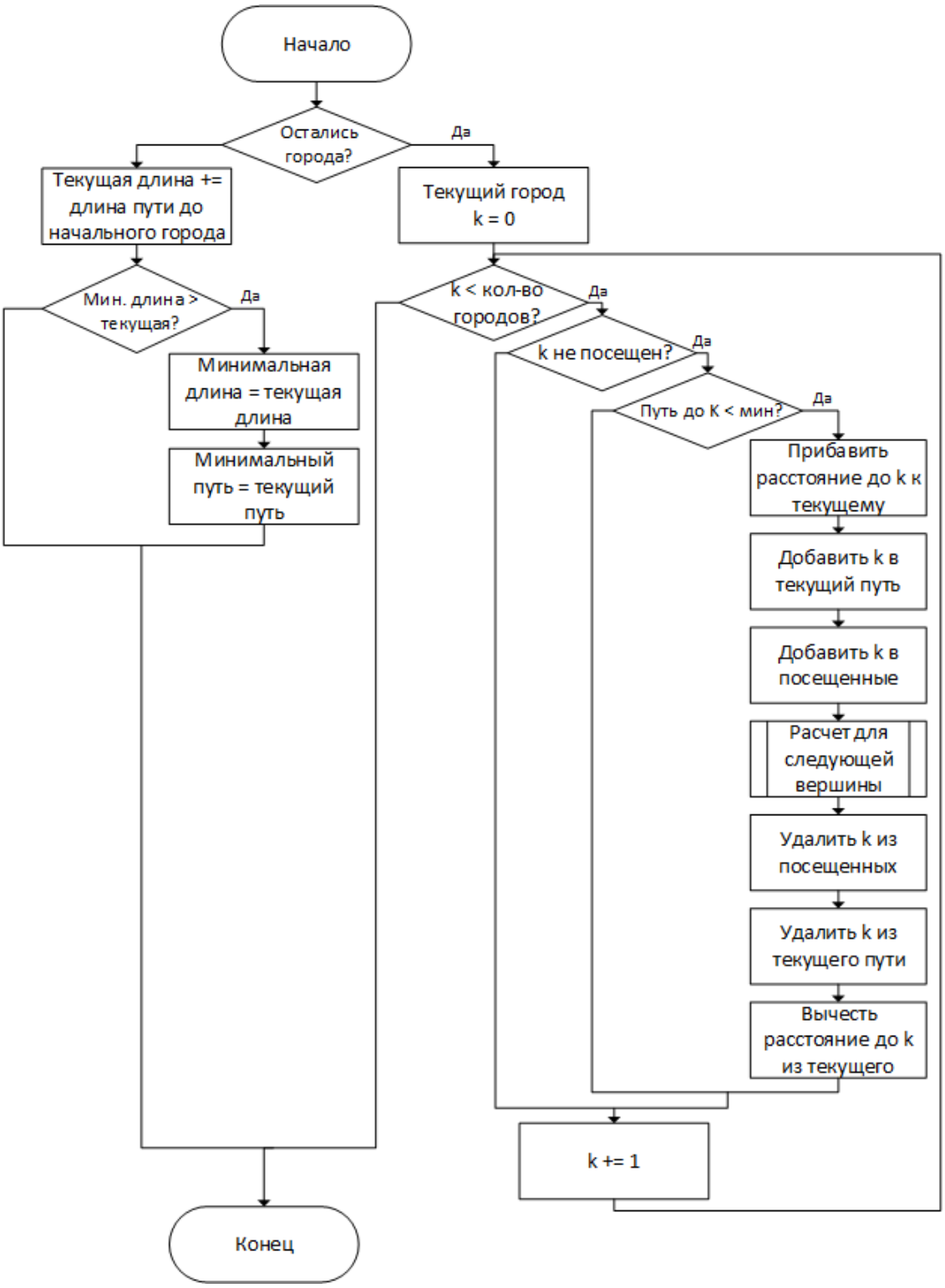

Рис. 2.4: Схема рекурсивной части алгоритма полного перебора

2.2 Вывод

Как видно из схем алгоритмов, количество блоков операций над данным в алгоритме полного перебора меньше, чем в муравьином. Это объясняется более сложной моделью данных в случае муравьиного метода. При этом стоит отметить, что в стандартный алгоритм полного перебора внесено одно изменение, а именно проверка текущей длины пути: если она уже больше длины минимального на текущий момент маршрута, то дальше маршрут по этому пути не строится. Такая модификация позволяет увеличить быстродействие программы.

3 Технологическая часть

В данном разделе приведены требования к программному обеспечению, средства реализации и листинги кода

3.1 Требования к ПО

Программа на вход получает матрицу смежности графа. Выход программы: последовательность целых чисел - минимальный по стоимости маршрут и действительное число - суммарная стоимость этого маршрута.

3.2 Средства реализации

Для реализации представленных алгоритмов был выбран язык C++. Время работы алгоритмов было замерено с помощью функции system_clock() из библиотеки chrono. Для тестирования использовался компьютер на базе процессора Intel Core i5 (4 физических ядра, 8 логических).

3.3 Листинги кода

В процессе разработки программы был спроектирован шаблон класса Matrix, служащий для работы с матрицей смежности, а также с матрицами для хранения количества феромона на ребрах.

В Листинге 3.1 показана реализация муравьиного алгоритма.

Листинг 3.1: Функция муравьиного поиска кратчайшего пути

```
1  Route ant(const Matrix<double> &distances, const int &tMax, const double &
       alpha, const double &p)
2  {
3      const size_t nTowns = distances.size();
4      // Pheromone coefficient
5      const double q = distances.average() * nTowns;
6      const double betta = 1 - alpha;
7      // Attractions of the towns
8      Matrix <double> attractions(distances);
9      attractions.inverse();
10     // Pheromones between towns
11     Matrix<double> teta(nTowns, START_TETA);
```

```
12      // Pheromone daily change
13      Matrix<double> deltaTeta(nTowns);
14
15      Route minRoute;
16      minRoute.length = -1;
17      std::vector<double> probabilities(nTowns, 0.0);
18
19      for (int t = 0; t < tMax; t++)
20      {
21          // Day
22          deltaTeta.zero();
23          // Ants loop
24          for (int k = 0; k < nTowns; k++)
25          {
26              std::vector<int> curPath = {k};
27              double curLength = 0;
28              int curTown = k;
29
30              // Ants path
31              std::vector<int> unvisited = getUnvisited(curPath, nTowns);
32              while (curPath.size() != nTowns)
33              {
34                  fill(probabilities.begin(), probabilities.end(), 0.0);
35
36                  // Probabilities of visiting towns
37                  for (const auto &town : unvisited)
38                  {
39                      int index = getIndex(unvisited, town);
40                      if (fabs(distances[curTown][town]) > EPS)
41                      {
42                          double sum = 0;
43                          for (auto n : unvisited)
44                              sum += pow(teta[curTown][n], alpha) * pow(attractions
                                    [curTown][n], betta);
45                          probabilities[index] = pow(teta[curTown][town], alpha) * pow
                                (attractions[curTown][town], betta) / sum;
46                      }
47                      else
48                          probabilities[index] = 0;
49                  }
50
51                  // Choosing town
52                  double coin = tossCoin();
53                  size_t townIndex = 0;
54                  double curProbability = 0;
55                  for (size_t j = 0; j < nTowns; j++)
56                  {
57                      curProbability += probabilities[j];
58                      if (coin < curProbability)
59                      {
```

```
60                          townIndex = j;
61                          break;
62                      }
63                  }
64                  int chosenTown = unvisited[townIndex];
65                  curPath.push_back(chosenTown);
66                  curLength += distances[curTown][chosenTown];
67                  deltaTeta[curTown][chosenTown] += q / curLength;
68                  curTown = chosenTown;
69                  unvisited.erase(unvisited.begin() + townIndex);
70              }
71              // Update path
72              curLength += distances[curPath[curPath.size() - 1]][curPath[0]];
73              if (minRoute.length < -EPS || (curLength < minRoute.length))
74              {
75                  minRoute.length = curLength;
76                  minRoute.path = curPath;
77              }
78          }
79          // Night
80          teta *= (1.0 - p);
81          teta += deltaTeta;
82          teta.replaceZero(MIN_TETA);
83      }
84      return minRoute;
85  }
```

В Листингах 3.1 и 3.2 представлена реализация алгоритма полного перебора, представленная главной функцией bruteForce и рекурсивной hamilton.

Листинг 3.2: Главная функция алгоритма полного перебора

```
1   Route bruteForce(const Matrix<double> &distances)
2   {
3       int n = distances.size();
4       std::vector<bool> visited(n, false);
5       std::vector<int> curPath;
6       Route minRoute;
7       minRoute.length = DBL_MAX;
8       double curLen = 0;
9       for (int i = 0; i < n; i++)
10      {
11          curPath.clear();
12          curPath.push_back(i);
13          fill(visited.begin(), visited.end(), 0);
14          visited[i] = true;
15          curLen = 0;
16          hamilton(distances, minRoute, curPath, visited, curLen);
17      }
18      return minRoute;
19  }
```

Основная функция вызывает в цикле для каждой вершины рекурсивную функцию, в которой проверяются все непосещенные вершины и если такие есть, то эта функция вызывается для следующих вершин пока не будет пройден полноценный маршрут.

Листинг 3.3: Рекурсивная функция алгоритма полного перебора

```
1  void hamilton(const Matrix<double> &distances,Route &minRote, std::vector<int
       > &curPath, std::vector<bool> &visited, double &curLen)
2  {
3      if (curPath.size() == distances.size())
4      {
5          double tmp = distances[curPath.back()][curPath[0]];
6          if (curLen + tmp < minRote.length)
7          {
8              minRote.path = curPath;
9              minRote.length = curLen + tmp;
10         }
11         return;
12     }
13     for (int i = 0; i < distances.size(); i++)
14     {
15         if (!visited[i])
16         {
17             double tmp = distances[curPath.back()][i];
18             if (curLen + tmp > minRote.length)
19                 continue;
20             curLen += tmp;
21             curPath.push_back(i);
22             visited[i] = true;
23             hamilton(distances, minRote, curPath, visited, curLen);
24             visited[i] = false;
25             curPath.pop_back();
26             curLen -= tmp;
27         }
28     }
29 }
```

3.4 Вывод

Была реализован алгоритм, использующий конвейерную обработку данных, и осуществлена запись результатов в лог.

4 Экспериментальная часть

В данном разделе будут приведены примеры работы программы, а также будет приведена интерпретация сформированного программой лог-файла.

4.1 Примеры работы

На рисунках 4.1 - 4.3 приведены примеры работы программы. В консоль выводятся входные данные (данные для первой ленты конвейера) и результат прохождения этих данных через конвейер.

Как видно из приведенных результатов: один входной элемент, который должен быть обработан на всех трех лентах, обрабатывается последовательно и его суммарное время обработки равно 1 секунда * 3 ленты = 3 секунды. При этом уже для трех входных объектах суммарное время конвейерной обработки становится гораздо меньше суммарного времени последовательной обработки: 5 секунд $<$ 9 секунд. Аналогичная ситуация наблюдается и для входной очереди из пяти элементов.

4.2 Анализ лога

Для того, чтобы отследить работу потоков, необходимо получить лог-файл, отражающий время начала работы ленты над очередным заданием и время окончания этой работы. Например, на рисунке 4.4 представлен лог-файл для ситуации, когда генератор подал на вход конвейеру пять элементов.

В первом столбце показано время записи, во втором лента конвейера, на которой производилась запись, далее указано начало или конец обработки и номер элемента, который обрабатывался. Если это начало обработки, то в конце указывается количество миллисекунд, которые должен обрабатываться этот элемент. Из представленного файла видно, что как только первый входной объект 0 item был обработан на первой ленте, его начинает обрабатывать вторая лента. В это время на первой ленте уже идет обработка следующего элемента. Как только на второй ленте закончилась обработка 0 item, начинает работать третья лента конвейера.

Время обработки текущего элемента на всех трех лентах конвейера одинаковое, поэтому ленты простаивают в течение работы программы, кроме ситуации в самом начале, когда первый элемент еще не поступил в очередь на обработку ко второй и третьей лентам.

Рассмотрим другой пример, когда на вход подаются задачи, требующие на обработку одинакового количества времени, однако на второй ленте конвейера обработка будет занимать в 2 раза больше времени, чем на других лентах. На рисунке 4.5 представлен лог работы программы.

Как видно из лог-файла, 3 лента конвейера простаивает 1 секунду на каждом элементе, кроме первого.

4.3 Вывод

При анализе лога было выяснено, что при разном времени обработки на разных лентах, некоторые из них могут простаивать.

Заключение

В ходе лабораторной работы был изучен и реализован алгоритма, использующий конвейерную обработку данных с использованием методов распараллеливания процессов. Были выявлены оптимальные для конвейерной обработки параметры входных задач - сбалансированное время обработки данных на всех лентах конвейера.