МГТУ им. Баумана

Лабораторная работа №6

По курсу: "Анализ алгоритмов"

Задача коммивояжёра

Работу выполнил: Подвашецкий Дмитрий, ИУ7-54Б

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

Москва, 2019

Оглавление

Введение

Задача коммивояжёра — одна из самых известных задач комбинаторной оптимизации, заключающаяся в поиске самого выгодного маршрута, проходящего через указанные города хотя бы по одному разу с последующим возвратом в исходный город. В условиях задачи указываются критерий выгодности маршрута (кратчайший, самый дешёвый, совокупный критерий и тому подобное) и соответствующие матрицы расстояний, стоимости и тому подобного. Как правило, указывается, что маршрут должен проходить через каждый город только один раз — в таком случае выбор осуществляется среди гамильтоновых циклов.

Данную задачу можно решить точным методом или же эвристическим. В качестве точного метода будет использован полный перебор, а в качестве эвристического - метод муравьиой колонии.

Задачами данной лабораторной являются:

1. изучение метода полного перебора и метода муравьиной колонии для решения задачи коммивояжёра;

2. реализация данных двух методов;

3. экспериментальное подтверждение различий во временно́й эффективности рассматриваемых алгоритмов для различных классов задач;

4. описание и обоснование полученных результатов в отчете о выполненной лабораторной работе, выполненного как расчётно-пояснительная записка к работе.

1 Аналитическая часть

Задача коммивояжёра относится к классу NP-трудных и не известен алгоритм, который позволет гарантировано её решить за полиномиальное по числу городов N. Однако для небольшого числа городов (N $<$ 15) существует множество способов решения.

В данной лабораторной работе будут исследованы один точный (полный перебо) и один эвристический (муравьиная колония) метод. Точные методы позволяют найти наилучший путь, а так же доказать, что найденый путь является таковым. В то время как эвристические методы работают существенно быстрее точных, но не гарантируют оптимальности найденого пути.

Полный перебор заключается в перестановки N-1 чисел (при зафиксированном стартовом городе) и поиске пути с минимальной стоимостью (длинной, временем и тд).

Метод муравьиной колонии основан на биологической идеи - принципе существования муравьиной колонии. Во время работы данного алгоритма происходит маркировка наиболее удачных путей феромоном.

Работа начинается с размещения муравьёв в вершинах графа (городах), затем начинается движение муравьёв — направление определяется вероятностным методом, на основании формулы вида:

$$P_{K,ij} = \begin{cases} \dfrac{(\tau_{ij}^{\alpha}(t)) * (\eta_{ij}^{\beta})}{\Sigma(\tau_{iq}^{\alpha}(t)) * (\eta_{iq}^{\beta})} & \text{если он не был в городе i} \\ 0 & \text{иначе} \end{cases}$$

α, β - настроечные параметры $\alpha + \beta = \text{const}$

τ_{ij} - кол-во ферамона на ребре ij

$\eta_{ij} = 1/D_{ij}$, D_{ij} - длина (стоимость) ребра ij

После того, как все муравьи закончили поиск, происходит перерасчет феромонов по формуле:

$$\tau_{ij}(\text{t+1}) = \tau_{ij}{}^{*}(\text{1-}\rho) + \Sigma\Delta\tau_{k,ij}(\text{t})$$

ρ - коэф. рассеивания ферамона

$$\Delta\tau_{k,ij} = \begin{cases} \dfrac{Q}{L_k} & \text{если ij ребро принадлежит маршруту k-го муравья} \\ 0 & \text{иначе} \end{cases}$$

Q - нормировачаная константа L_k - длина пути k-го муравья

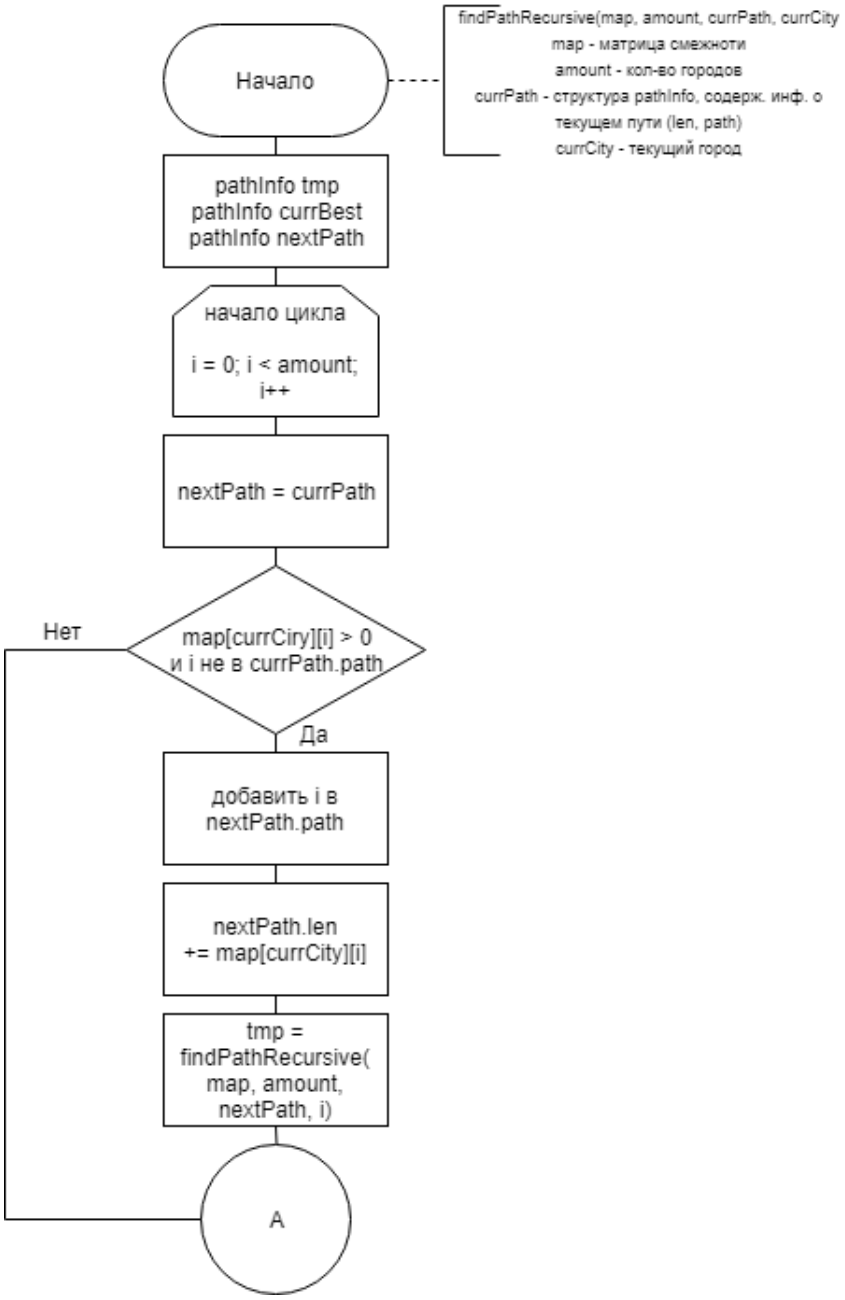
Этот процесс повторяется T_{max} раз.

Вывод

В данном разделе были изучены основные идеи, рассматриваемых в данной лабораторной работе, алгоритмов.

2 Конструкторская часть

2.0.1 Схемы алгоритмов



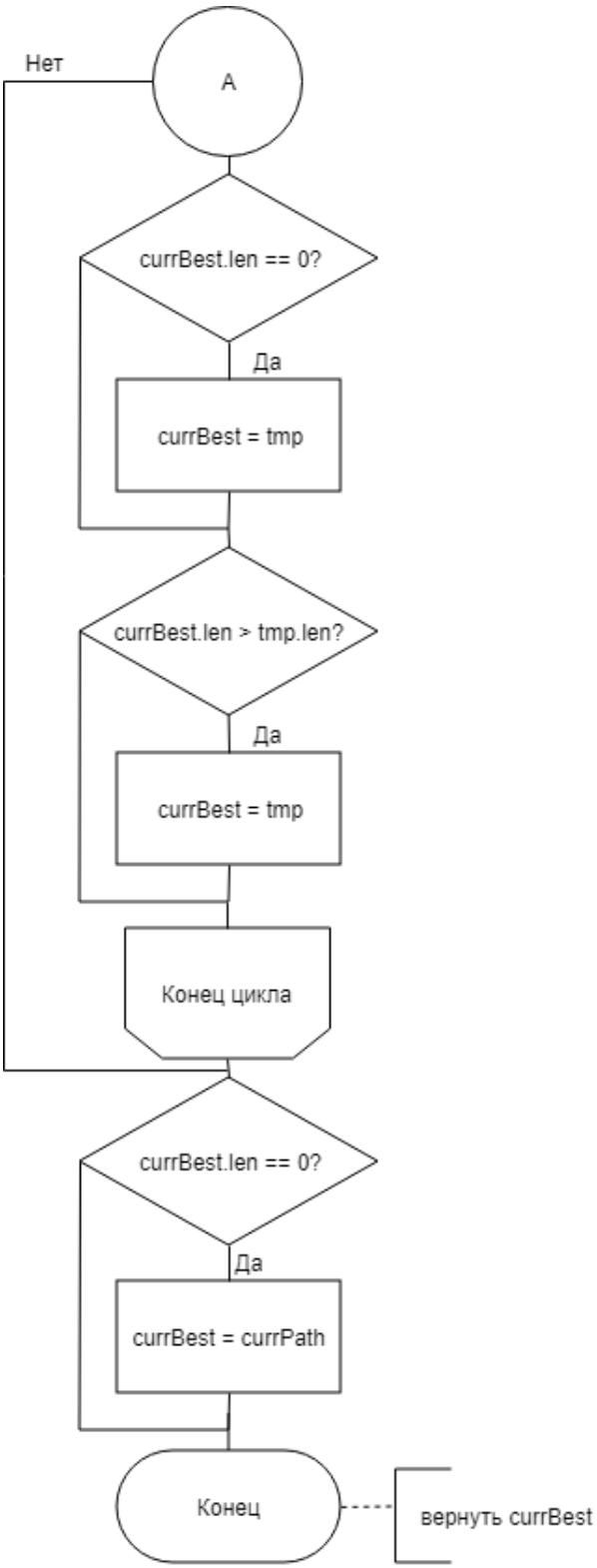


Рис. 1. Схема рекурсивной реализации решения задачи коммивояжёра методом полного перебора

3 Технологическая часть

3.1 Выбор ЯП

В качестве языка программирования был выбрал C++, так как он позволяет реализовать задачу максимально комфортно.

3.2 Замеры времени

Замер времени работы алгоритмов производился при помощи функций clock() из библиотеки time.h.

Также производится усреднение времени работы алгоритмов. Для этого время считается для 10 вызовов, и после делится на 10.

3.3 Требования к ПО

Требования к вводу:

1. Имя файла, содержащего матрицу смежности данного графа

Требования к программе:

1. Корректный ввод, корректный вывод, программа не должна аварийно завершаться

Требования структуре входного файла: На первой строке написанно одно натуральное число - кол-во вершин. Значения в строках должны быть записаны на одной строке. Разделитель - пробел. Конец строки - переход на новую строку.

Пример:
2
0 1
2 0

3.4 Сведения о модулях программы

Программа состоит из:

- main.cpp - главный файл программы

- antcolony.cpp - файл с реализациец алгоритма муравьиной колонии (Листинг 3.1.)
- recursive.cpp - файл с реализацией алгоритма полного перебора (Листинг 3.2.)
- utility.cpp - файл с реализацией доп. функций (т.к. чтение матрица) (Листинг 3.3.)

Листинг 3.1: Стандартный алгоритм

```
1  pathInfo runColony(environment &env)
2  {
3          pathInfo curBest;
4
5          for (size_t i = 0; i < env.tMax; i++)
6          {
7                  for (size_t j = 0; j < env.cities; j++)
8                  {
9                          pathInfo tmp =  runAnt(env, j);
10                         tmp.len += env.map[tmp.path[0]][
                               tmp.path[tmp.path.size()-1]];
11
12                         if (tmp.path.size() == env.cities
                               && curBest.len == 0)
13                                 curBest = tmp;
14                         if (tmp.path.size() == env.cities
                               && curBest.len > tmp.len)
15                                 curBest = tmp;
16                 }
17
18                 recalculateTau(env);
19         }
20
21         return curBest;
22 }
23
24
25 pathInfo runAnt(environment &env, size_t baseCity)
26 {
27         pathInfo curPath;
28
29         while (1)
30         {
31                 std::vector<double> probs;
32                 double probsDivider = 0;
33                 if (!isInVector(curPath.path, baseCity))
34                         curPath.path.push_back(baseCity);
```

```
35                  else
36                          break;
37
38                  for (size_t i = 0; i < env.cities; i++)
39                  {
40                          if (env.map[baseCity][i] > 0 && !
                               isInVector(curPath.path, i))
41                          {
42                                  probsDivider += std::pow(
                                       env.tau[baseCity][i],
                                       env.alpha)*
43                                  std::pow(std::pow(env.map[
                                       baseCity][i], -1), env.
                                       beta);
44                          }
45                  }
46
47                  for (size_t i = 0; i < env.cities; i++)
48                  {
49                          if (isInVector(curPath.path, i) ||
                               env.map[baseCity][i] <= 0)
50                                  probs.push_back(0);
51                          else
52                                  probs.push_back(std::pow(
                                       env.tau[baseCity][i],
                                       env.alpha)*
53                                  std::pow(std::pow(env.map[
                                       baseCity][i], -1), env.
                                       beta)/
54                                  probsDivider);
55                  }
56
57                  double probsSum = 0;
58                  for (auto it : probs)
59                          probsSum += it;
60
61                  if (probsSum < 0.1)
62                          break;
63
64                  double randVal = (double(rand()) / (
                       RAND_MAX)) + 0.000001;
65
66                  probsSum = 0;
67                  for (size_t i = 0; i < env.cities; i++)
68                  {
69                          probsSum += probs[i];
70                          if (probsSum >= randVal && !
```

```
                                isInVector(curPath.path, i))
71                          {
72                                  curPath.len += env.map[
                                       baseCity][i];
73
74                                  baseCity = i;
75                                  break;
76                          }
77                  }
78          }
79
80          size_t curPathSize = curPath.path.size();
81          if (curPathSize == env.cities)
82          {
83                  double dTau = env.Q / curPath.len;
84                  for (size_t i = 0; i < curPathSize - 1; i
                        ++)
85                  {
86                          env.dTau[curPath.path[i]][curPath.
                               path[i+1]] += dTau;
87                          env.dTau[curPath.path[i+1]][
                               curPath.path[i]] += dTau;
88                  }
89          }
90
91          return curPath;
92  }
93
94  void recalculateTau(environment &env)
95  {
96          for (size_t i = 0; i < env.cities; i++)
97          {
98                  for (size_t j = 0; j < env.cities; j++)
99                  {
100                         env.tau[i][j] = env.tau[i][j]*(1 -
                                env.ro) + env.dTau[i][j];
101                         env.dTau[i][j] = 0;
102                 }
103         }
104 }
```