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»

Лабораторная работа №6
по курсу "Анализ алгоритмов"

Тема Муравьиный алгоритм

Студент Воякин А. Я.

Группа ИУ7-54Б

Оценка (баллы) ______

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

Москва — 2020 г.

Оглавление

Введение

В последние два десятилетия при оптимизации сложных систем исследователи все чаще применяют природные механизмы поиска наилучших решений. Это механизмы обеспечивают эффективную адаптацию флоры и фауны к окружающей среде на протяжении миллионов лет. Сегодня интенсивно разрабатывается научное направление Natural Computing — «Природные вычисления», объединяющее методы с природными механизмами принятия решений, а именно:

1. Genetic Algorithms — генетические алгоритмы;
2. Evolution Programming — эволюционное программирование;
3. Neural Network Computing — нейросетевые вычисления;
4. DNA Computing — ДНК-вычисления;
5. Cellular Automata — клеточные автоматы;
6. Ant Colony Algorithms — муравьиные алгоритмы.

Эти механизмы обеспечивают эффективную адаптацию флоры и фауны к окружающей среде на протяжении миллионов лет. Имитация самоорганизации муравьиной колонии составляет основу муравьиных алгоритмов оптимизации — нового перспективного метода природных вычислений. Колония муравьев может рассматриваться как многоагентная система, в которой каждый агент (муравей) функционирует автономно по очень простым правилам. В противовес почти примитивному поведению агентов, поведение всей системы получается на удивление разумным.

Муравьиные алгоритмы серьезно исследуются европейскими учеными с середины 90х годов. На сегодня уже получены хорошие результаты муравьиной оптимизации таких сложных комбинаторных задач, как: задачи коммивояжера, задачи оптимизации маршрутов грузовиков, задачи раскраски графа, квадратичной задачи о назначениях, оптимизации сетевых графиков, задачи календарного планирования и других. Особенно эффективны муравьиные алгоритмы при online-оптимизации процессов в распределенных нестационарных системах, например трафиков в телекоммуникационных сетях [1].

1. Аналитическая часть

В данном разделе будет описан муравьиный алгоритм на примере решения задачи коммивояжера.

1.1 Цель и задачи

Основной целью данной работы является изучение особенностей работы муравьиного алгоритма. Для того, чтобы добиться этой цели необходимо выполнить ряд задач:

1. примененить знания программирования для реализации муравьиного алгоритма;
2. получить практические навыки во время выполнения задания;
3. экспериментально подтвердить различия во временной эффективности работы муравьиного алгоритма при разных значениях коэффицента важности величины пути и коэффициента важности мощности феромона при помощи разработанного программного обеспечения на материале замеров процессорного времени;
4. описать и обосновать полученные результаты в отчете о выполненной лабораторной работе, выполненном как расчётно-пояснительная записка к работе.

1.2 Задача коммивояжера

Задача коммивояжера формулируется как задача поиска минимального по стоимости замкнутого маршрута по всем вершинам без повторений на полном взвешенном графе с n вершинами. Вершины графа являются городами, которые должен посетить коммивояжер, а веса ребер отражают расстояния или стоимости проезда. Эта задача является NP-трудной, и точный переборный алгоритм ее решения имеет факториальную сложность [2].

1.3 Описание муравьиного алгоритма

Муравьиные алгоритмы представляют собой вероятностную жадную эвристику, где вероятности устанавливаются, исходя из информации о качестве решения, полученной из предыдущих решений. Идея муравьиного алгоритма - моделирование поведения муравьёв, связанного с их способностью быстро находить кратчайший путь от муравейника к источнику пищи и адаптироваться к изменяющимся условиям, находя новый кратчайший путь.

Моделирование поведения муравьёв связано с распределением феромона на тропе – ребре графа в задаче коммивояжёра. При этом вероятность включения ребра в маршрут отдельного муравья пропорциональна количеству феромона на этом ребре, а количество откладываемого феромона пропорционально длине маршрута. Чем короче маршрут, тем больше феромона будет отложено на его рёбрах, следовательно, большее количество муравьёв будет включать его в синтез собственных маршрутов. Моделирование такого подхода, использующего только положительную обратную связь, приводит к преждевременной сходимости – большинство муравьёв двигается по локально оптимальному маршруту. Избежать этого можно, моделируя отрицательную обратную связь в виде испарения феромона.

С учётом особенностей задачи коммивояжёра, мы можем описать локальные правила поведения муравьёв при выборе пути.

1. Муравьи обладают «памятью». Поскольку каждый город может быть посещён только один раз, то у каждого муравья есть список уже посещённых городов. Обозначим через $J_{i,k}$ список городов, которые необходимо посетить муравью k, находящемуся в городе i.

2. Муравьи обладают «зрением», которое определяет степень желания посетить город j, если муравей находится в городе i. Будем считать, что видимость обратно пропорциональна расстоянию между городами.

3. Муравьи обладают «обонянием», с помощью которого 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 $\tau_{ij}(t)$.

4. На основании предыдущих утверждений мы можем сформулировать вероятностно-пропорциональное правило, определяющее вероятность перехода k-ого муравья из города i в город j:

$$P_{ij,k}(t) = \begin{cases} \frac{(\tau_{ij}(t))^{\alpha}(\eta_{ij}(t))^{\beta}}{\sum\limits_{l \in J(i,k)} (\tau_{il}(t))^{\alpha}(\eta_{il}(t))^{\beta}}, j \in J(i,k) \\ 0, j \notin J(i,k) \end{cases}, \tag{1.1}$$

   где $\tau_{ij}(t)$ – уровень феромона, $\eta_{ij}(t)$ – эвристическое расстояние, а α и β – константные параметры.

   Выбор города является вероятностным, в общую зону всех городов бросается случайное число, которое и определяет выбор муравья. При $\alpha = 0$ алгоритм вырождается до жадного алгоритма, по которому на каждом шаге будет выбираться ближайший город.

5. При прохождении ребра муравей оставляет на нём некоторое количество феромона, которое должно быть связано с оптимальностью сделанного выбора. Пусть есть маршрут, пройденный муравьём k к моменту времени t, T – длина этого маршрута, $L_k(t)$ - цена текущего решения для k-ого муравья а Q – параметр, имеющий значение порядка цены оптимального решения. Тогда откладываемое количество феромона

$$\Delta\tau_{ij,k}(t) = \begin{cases} \frac{Q}{L_k(t)}, (i,j) \in T_k(t) \\ 0, (i,j) \notin T_k(t) \end{cases}, \tag{1.2}$$

   а испаряемое количество феромона

$$\tau_{ij}(t+1) = (1-p)\tau_{ij}(t) + \sum_{k=1}^{m} \Delta\tau_{ij,k}(t), \qquad (1.3)$$

где m – количество муравьёв в колонии [3].

1.4 Вариации муравьиного алгоритма

Ниже приведены вариации муравьиного алгоритма.

1. **Элитарная муравьиная система**. Из общего числа муравьёв выделяются так называемые «элитные муравьи». По результатам каждой итерации алгоритма производится усиление лучших маршрутов путём прохода по данным маршрутам элитных муравьёв и, таким образом, увеличение количества феромона на данных маршрутах. В такой системе количество элитных муравьёв является дополнительным параметром, требующим определения. Так, для слишком большого числа элитных муравьёв алгоритм может «застрять» на локальных экстремумах.

2. **Max-Min муравьиная система**. Добавляются граничные условия на количество феромонов (τmax,τmin). Феромоны откладываются только на глобально лучших или лучших в итерации путях. Все рёбра инициализируются значением τmax.

3. **Ранговая муравьиная система(ASrank)**. Все решения ранжируются по степени их пригодности. Количество откладываемых феромонов для каждого решения взвешено так, что более подходящие решения получают больше феромонов, чем менее подходящие.

4. **Длительная ортогональная колония муравьёв (COAC)**. Механизм отложения феромонов COAC позволяет муравьям искать решения совместно и эффективно. Используя ортогональный метод, муравьи в выполнимой области могут исследовать их выбранные области быстро и эффективно, с расширенной способностью глобального поиска и точностью.

1.5 Вывод

В данном разделе были изучены различные вариации муравьиного алгоритма.

2. Конструкторская часть

В данном разделе в соответствии с описанием алгоритмов, приведенными в аналитической части работы, будет рассмотрена схема муравьиного алгоритма.

2.1 Схемы алгоритмов

На рисунке 2.1 представлена обобщённая схема муравьиного алгоритма.

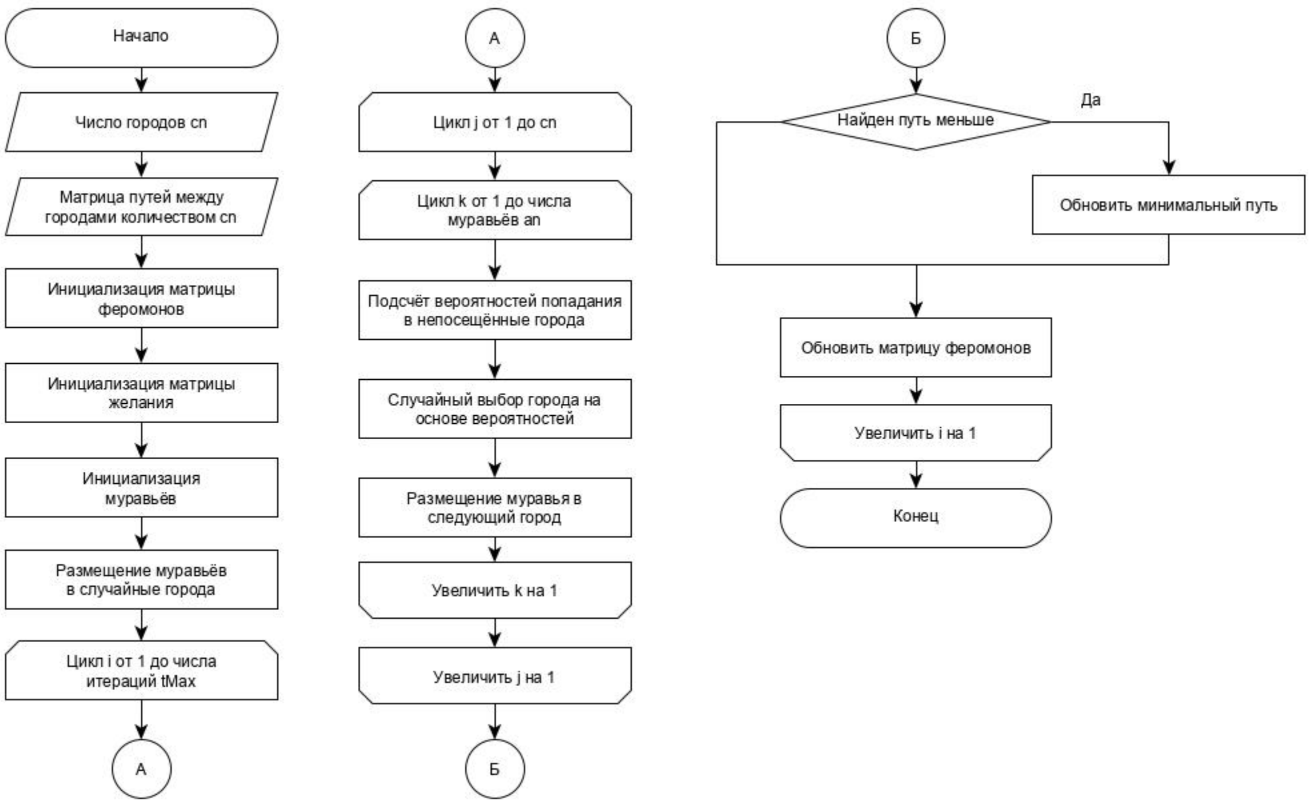

Рис. 2.1: Схема муравьиного алгоритма

2.2 Вывод

В данном разделе были рассмотрены принципы работы и схемы муравьиного алгоритма.

3. Технологическая часть

В данном разделе приведены требования к программному обеспечению, средствам реализации, а также листинги кода.

3.1 Средства реализации

Для реализации программы был использован язык программирования C++, так как он был подробно изучен в курсе объектно-ориентированного программирования в университете[4].

Для замера времени использовалась функция, приведенная на листинге[5]. Данная функция считает реальное процессорное время в тиках. Для ее работы была подключена библиотека time.h.

3.2 Реализация алгоритмов

В листингах 3.1, 3.2, 3.3, 3.4 и 3.5 описана реализация муравьиного алгоритма.

Листинг 3.1: Класс муравья Ant

```
class Ant
{
public:
  size_t path_len;
  std::vector<bool> visited;
  std::vector<size_t> path;

  Ant(const size_t graph_size);

  void visit_city(const size_t city, const size_t cur_path_len,
  const size_t cur_path_dist);
  void clear_visits();
  void make_default_path();
  bool is_visited(const size_t city) const;
};
```

Листинг 3.2: Методы класса Ant

```
Ant::Ant(const size_t graph_size) : path_len(0)
{
  for (size_t i = 0; i < graph_size; i++)
  {
    path.push_back(0);
    visited.push_back(false);
  }
```

```
}

void Ant::visit_city(const size_t city, const size_t cur_path_len,
  const size_t cur_path_dist)
{
  path_len += cur_path_dist;
  path[cur_path_len] = city;
  visited[city] = true;
}

void Ant::clear_visits()
{
  for (size_t i = 0; i < visited.size(); i++)
    visited[i] = false;
  path_len = 0;
}

void Ant::make_default_path()
{
  path_len = 0;
  visit_city(path[path.size() - 1], 0, 0);
}

bool Ant::is_visited(const size_t city) const
{
  return visited[city];
}
```

Листинг 3.3: Класс алгоритма ACO

```
class ACO
{
private:
  const std::vector<std::vector<int>> dist_graph;
  const size_t cities_count;

  std::vector<std::vector<double>> pher_graph;
  std::vector<std::vector<double>> desire_graph;

  std::vector <Ant> ants;
  size_t ants_count;

  std::vector<double> paths_probs;

  double alpha = 0.5;
  double rho = 0.5;
  size_t tMax = 100;
  double beta = 1 - alpha;

  const double Q = 5;
  const double ants_factor = 1;
  const double init_pher_value = 1;

public:
```

```
    size_t min_len = 0;
    std::vector<size_t> min_path;

    ACO(const Graph<int>& graph);

    void execute();
    void change_params(double alpha, double rho, size_t tMax);

private:
    void make_default_state();
    void init_ants();
    void init_pher_graph();
    void pave_ants_paths();
    size_t get_next_city(const Ant& ant, const size_t cur_city);
    void update_min_path();
    void update_pheromones();
    void make_default_ants();
    size_t select_next_city();
    double get_sum_probabilities();
};
```

Листинг 3.4: Методы инициализации и запуск алгоритма

```
ACO::ACO(const Graph<int>& graph) :
  dist_graph(graph.graph), cities_count(graph.size)
{
  // init pher_graph
  for (size_t i = 0; i < cities_count; i++)
  {
    std::vector<double> line;
    for (size_t j = 0; j < cities_count; j++)
      line.push_back(init_pher_value);
    pher_graph.push_back(line);
  }

  // init desire_graph
  for (size_t i = 0; i < cities_count; i++)
  {
    std::vector<double> line;
    for (size_t j = 0; j < cities_count; j++)
      line.push_back(dist_graph[i][j] == 0 ? 0 :
        1.0 / dist_graph[i][j]);
    desire_graph.push_back(line);
  }

  // init ants_count
ants_count = cities_count * ants_factor;
  for (size_t i = 0; i < ants_count; i++)
  {
    Ant ant(cities_count);
    ants.push_back(ant);
  }

  // init paths_probs
```

```
    for (size_t i = 0; i < cities_count; i++)
      paths_probs.push_back(0);
}

void ACO::execute()
{
  make_default_state();
  init_pher_graph();
  init_ants();

  for (size_t i = 0; i < tMax; i++)
  {
    pave_ants_paths();
    update_min_path();
    update_pheromones();
    make_default_ants();
  }
}

void ACO::change_params(double alpha, double rho, size_t tMax)
{
  this->alpha = alpha;
  this->beta = 1 - alpha;
  this->rho = rho;
  this->tMax = tMax;
}

void ACO::make_default_state()
{
  min_len = 0;
  min_path.clear();
}

void ACO::init_ants()
{
  for (size_t i = 0; i < ants_count; i++)
  {
    ants[i].clear_visits();
    ants[i].visit_city(rand() % cities_count, 0, 0);
  }
}

void ACO::init_pher_graph()
{
  for (size_t i = 0; i < cities_count; i++)
    for (size_t j = 0; j < cities_count; j++)
      pher_graph[i][j] = init_pher_value;
}
```

Листинг 3.5: Основные функции муравьиного алгоритма

```
void ACO::pave_ants_paths()
{
  for (size_t i = 0; i < cities_count - 1; i++)
```

```
  {
    for (size_t j = 0; j < ants_count; j++)
    {
      const size_t cur_city = ants[j].path[i];
      const size_t next_city = get_next_city(ants[j], cur_city);
      const int dist = dist_graph[cur_city][next_city];

      ants[j].visit_city(next_city, i + 1, dist);
    }
  }

  for (size_t j = 0; j < ants_count; j++)
  {
    size_t i_ind = ants[j].path[ants[j].path.size() - 1];
    size_t j_ind = ants[j].path[0];
    const int dist_init_city = dist_graph[i_ind][j_ind];
    ants[j].path_len += dist_init_city;
  }
}

size_t ACO::get_next_city(const Ant& ant, const size_t cur_city)
{
  double sumP = 0;

  for (size_t i = 0; i < cities_count; i++)
  {
    double pher_factor = pow(pher_graph[cur_city][i], alpha);
    double desire_factor = pow(desire_graph[cur_city][i], beta);
    sumP += pher_factor * desire_factor;
  }

  for (size_t i = 0; i < cities_count; i++)
  {
    if (i == cur_city || ant.is_visited(i))
      paths_probs[i] = 0;
    else
    {
      double pher_factor = pow(pher_graph[cur_city][i], alpha);
      double desire_factor = pow(desire_graph[cur_city][i], beta);
      paths_probs[i] = pher_factor * desire_factor / sumP;
    }
  }

  return select_next_city();
}

void ACO::update_min_path()
{
  for (size_t i = 0; i < ants_count; i++)
  {
    const size_t cur_len = ants[i].path_len;
    if (cur_len < min_len || min_len == 0)
    {
```

```
          min_len = cur_len;
          min_path = ants[i].path;
        }
    }
}

void ACO::update_pheromones()
{
    for (size_t i = 0; i < cities_count; i++)
        for (size_t j = 0; j < cities_count; j++)
            pher_graph[i][j] *= (1 - rho);

    for (size_t i = 0; i < ants_count; i++)
    {
        Ant& ant = ants[i];

        double dt = Q / ant.path_len;
        for (size_t j = 0; j < cities_count - 1; j++)
            pher_graph[ant.path[j]][ant.path[j + 1]] += dt;
        pher_graph[ant.path[cities_count - 1]][ant.path[0]] += dt;
    }
}

void ACO::make_default_ants()
{
    for (size_t i = 0; i < ants_count; i++)
    {
        ants[i].clear_visits();
        ants[i].make_default_path();
    }
}

size_t ACO::select_next_city()
{
    double sum_probabilities = get_sum_probabilities();
    double rand_num = ((double) rand() / (RAND_MAX)) * sum_probabilities;
    double total = 0;
    size_t city = 0;

    for (size_t i = 0; i < cities_count && total < rand_num; i++)
    {
        total += paths_probs[i];
        if (total >= rand_num)
            city = i;
    }

    return city;
}

double ACO::get_sum_probabilities()
{
    double sum_probabilities = 0;
    for (size_t i = 0; i < cities_count; i++)
```

```
        sum_probabilities += paths_probs[i];
    return sum_probabilities;
}
```

3.3 Вывод

В данном разделе была рассмотрена конкретная реализация на языке C++ муравьиного алгоритма.

4. Экспериментальная часть

В данном разделе будет проведен сравнительный анализ работы реализованного муравьинного алгоритма при различных параметрах.

4.1 Сравнительный анализ

Для сравнения работы муравьиного алгоритма при различных параметрах замеры выполнялись на графе из 10 узлов. Параметры α и η варьируются от 0 до 1 с шагом 0.25, а количество итераций tMax - от 100 до 200 с шагом 100. Результаты эксперимента представлены в таблице 4.1. Результат работы алгоритма перебором - 12.

4.2 Вывод

Из данной таблицы можно увидеть, что для данного набора параметров при $\alpha = 1$, $\eta = 0.5$ и tMax = 100 муравьиный алгоритм выдает наихудший результат. При параметрах $\alpha = 0.25$, $\eta = 1$ и tMax = 200 муравьинный алгоритм наиболее приближен к результату, полученному полным перебором. При правильном подборе параметров муравьиный алгоритм выдает результат, близкий к наилучшему, при этом работая намного быстрее полного перебора (на 99.6% быстрее на графе из 10 узлов).

Таблица 4.1: Сравнение работы муравьиного алгоритма при различных параметрах

α	ρ	T_{max}	Мин. путь
0	0	100	16
0	0	200	16
0	0.25	100	18
0	0.25	200	16
0	0.5	100	16
0	0.5	200	16
0	0.75	100	16
0	0.75	200	16
0	1	100	16
0	1	200	16
0.25	0	100	16
0.25	0	200	16
0.25	0.25	100	16
0.25	0.25	200	16
0.25	0.5	100	19
0.25	0.5	200	16
0.25	0.75	100	16
0.25	0.75	200	16
0.25	1	100	14
0.25	1	200	12
0.5	0	100	16
0.5	0	200	18
0.5	0.25	100	16
0.5	0.25	200	16
0.5	0.5	100	16
0.5	0.5	200	16
0.5	0.75	100	16
0.5	0.75	200	16
0.5	1	100	15
0.5	1	200	13
0.75	0	100	22
0.75	0	200	16
0.75	0.25	100	21
0.75	0.25	200	16
0.75	0.5	100	18
0.75	0.5	200	16
0.75	0.75	100	16
0.75	0.75	200	16
0.75	1	100	18
0.75	1	200	15
1	0	100	22
1	0	200	23
1	0.25	100	20
1	0.25	200	25
1	0.5	100	27
1	0.5	200	22
1	0.75	100	24
1	0.75	200	18
1	1	100	22
1	1	200	21

Заключение

В данном разделе был проведен сравнительный анализ работы реализованного муравьиного алгоритма при различных параметрах, из которого можно сделать вывод, что при правильном подборе параметров муравьиный алгоритм находит оптимальный ответ за приемлимое время, намного отличающееся (на 99.6% быстрее на графе из 10 узлов) от времени нахождения пути полным перебором.

Литература

[1] Штовба С.Д. Муравьиные алгоритмы // Exponenta Pro. Математика в приложениях. – 2003. – №4

[2] Шутова Ю.О., Мартынова Ю.А. ИССЛЕДОВАНИЕ ВЛИЯНИЯ РЕГУЛИРУЕМЫХ ПАРАМЕТРОВ МУРАВЬИНОГО АЛГОРИТМА НА СХОДИМОСТЬ. Томский политехнический университет, 634050, Россия, г. Томск, пр. Ленина, 30, 2014. С. 281-282.

[3] Чураков Михаил, Якушев Андрей Муравьиные алгоритмы. 2006. С. 9- 11.

[4] https://cppreference.com/ [Электронный ресурс]

[5] [Электронный ресурс] Документация по функции замера времени. https://proginfo.ru/time/