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 «Фундаментальные Науки»

КАФЕДРА ФН-12 «Математическое моделирование»

ОТЧЕТ

ПО ЛАБОРАТОРНОЙ РАБОТЕ НА ТЕМУ:

Задача коммивояжера

Студент: _____ Мациевский И. М.

дата, подпись Ф.И.О.

Преподаватель: _____ Волкова Л. Л.

дата, подпись Ф.И.О.

Москва, 2023

Содержание

Введение

Цель лабораторной работы: разработать два метода для решения задачи коммивояжера.

Для достижения поставленной цели требуется решить следующие **задачи**.

1. Описать метод полного перебора для решения задачи коммивояжёра, указать его преимущества и недостатки. Описать схему алгоритма для метода. Реализовать алгоритм.

2. Описать метод решения задачи коммивояжёра на основе муравьиного алгоритма, указать его преимущества и недостатки. Описать схему алгоритма для метода. Реализовать алгоритм.

3. Выполнить оценку трудоёмкости составленных алгоритмов по разработанным схемам алгоритмов.

4. Провести сравнительный анализ двух рассмотренных методов решения задачи коммивояжёра.

5. Выполнить параметризацию последнего метода по трём его параметрам. Для параметризации в качестве критерия качества получаемого решения использовать максимальное значение (при желании вспомогательный критерий — медианное значение) отклонения длины полученного маршрута от эталонной длины. Одно медианное значение, как и одно максимальное, получается для серии результатов для каждого графа из набора данных, для одного графа минимум M=10 запусков поиска решения муравейником.

1 Аналитическая часть

1.1 Задача коммивояжера

Задача коммивояжера — это классическая задача комбинаторной оптимизации, которая формулируется следующим образом: имеется граф, в котором вершины представляют города, а рёбра — пути между городами. Каждому ребру сопоставлен вес, который обычно представляет собой расстояние или стоимость перемещения между соответствующими городами. Задача состоит в том, чтобы найти самый выгодный (кратчайший или наименее затратный) маршрут, проходящий через каждый город ровно один раз и возвращающийся в исходный город.

1.2 Метод полного перебора

Метод полного перебора — это способ решения задачи коммивояжера, при котором алгоритм перебирает все возможные варианты посещения вершин графа и выбирает оптимальный. Для задачи коммивояжера метод полного перебора проверяет все возможные порядки посещения городов и находит тот, который является минимальным по суммарному пройденному расстоянию. Преимуществом этого метода является простота его написания, но он неэффективен из-за проверки всех возможных маршрутов.

1.3 Муравьиный алгоритм

Муравьиный алгоритм — это оптимизационный алгоритм, основанный на поведении муравьев при поиске пути от колонии к источнику пищи.
Ниже описаны основные шаги муравьиного алгоритма.

1. **Инициализация феромона и видимости.** Для каждого ребра графа устанавливаются начальные значения феромона. В начале алгоритма феромон на каждом ребре равен небольшому положительному числу. Также определяется мера "видимости"для каждого ребра, которая может быть обратной длине ребра или другой характеристике.

2. **Распределение муравьев.** Каждый муравей начинает свой маршрут из начального города. На каждом шаге муравей выбирает следующий город с учётом феромона и видимости.

3. **Обновление феромона.** После того как все муравьи завершили свой маршрут, феромон обновляется на всех рёбрах графа. Феромон испаряется со временем, и на каждом шаге алгоритма добавляется новый феромон в соответствии с "успешностью"маршрута муравья.

4. **Критерии остановки.** Алгоритм выполняется до достижения критериев остановки, таких как максимальное количество итераций или сходимость результатов.

Преимущества.

1. **Параллелизм.** Муравьи могут двигаться параллельно и исследовать различные части пространства поиска, что делает алгоритм подходящим для параллельных вычислений.

2. Алгоритм учитывает **глобальную информацию** (феромон на рёбрах) и **локальную информацию** (видимость рёбер).

3. Муравьиный алгоритм применим к многим комбинаторным задачам, таким как задача коммивояжёра.

2 Конструкторская часть

2.1 Схема алгоритма для метода полного перебора

1. **Инициализация.** Требуется задать список городов (1, 2, ..., n), создать переменную для отслеживания минимальной длины маршрута($minLen$), создать массив для хранения текущей перестановки городов.

2. **Генерация перестановок.** Используя алгоритм генерации перестановок, генерируются все возможные перестановки городов.

3. **Вычисление длины маршрута.** Для каждой сгенерированной перестановки рассчитывается длина маршрута, равная сумме расстояний между последовательными городами, а также расстояния между первым и последним городами для создания замкнутого маршрута.

4. **Обновление минимальной длины.** Если длина текущего маршрута меньше, чем $minLen$, $minLen$ обновляется.

5. **Сохранение текущей перестановки.** Если длина текущего маршрута меньше, чем $minLen$, сохраняется текущая перестановка.

6. **Повторение.** Процессы 2-5 повторяются до тех пор, пока не будут рассмотрены все возможные перестановки.

7. **Вывод результата.** После завершения всех переборов выводится минимальная длина маршрута и соответствующая перестановка городов.

2.2 Реализованные методы для полного перебора

Ниже представлены методы класса $BruteForceRiver$.

1. *solve* Инициализирует вектор $rivers$ с порядком рек от 0 до $numRivers-1$. Использует $next\_permutation$ для генерации всех возможных перестановок рек в порядке от 0 до $numRivers-1$. Для каждой сгенерированной перестановки рассчитывает общее время перемещения по рекам с помощью метода $calculateTourTime$. Сравнивает текущее время с минимальным временем, и, если оно меньше, обновляет минимальное время и сохраняет текущую перестановку как лучшую.

2. $calculateTourTime$ Принимает перестановку городов (рек) и рассчитывает общее время перемещения по рекам в порядке, заданном перестановкой. Суммирует времена перемещения между последовательными реками, а также время перемещения от последней к первой реке для создания замкнутого маршрута.

2.3 Схема для метода решения на основе муравьиного алгоритма

1. **Инициализация**. Создается колония виртуальных муравьев, количество муравьев равно количеству вершин в графе. Инициализируется стартовое положение каждого муравья. Все рёбра инициализируются начальным количеством феромона.

2. **Поиск решения**. Муравьи отправляются в свободное путешествие. Муравей последовательно посещает вершины, пока не попадет в тупик или не дойдет до стартовой вершины. Решения муравья зависит от раскиданного феромона и расстояний до вершин.

3. **Локальное обновление феромонов**. В зависимости от того, выполнил ли муравей поставленную задачу, напрямую зависит, будет ли он откладывать феромоны. Если задача не выполнена, то муравей не оставляет феромоны на своем пути, иначе муравей, в зависимости от длины своего пути, оставляет на своем пути феромоны. Это позволяет усилить путь для будущих муравьев и учитывать локальную информацию о качестве пути.

4. **Определение лучшего решения**. Если муравей выполнил поставленную задачу, его путь и длина этого пути сохраняется. (Если, конечно, длина этого пути, короче уже имеющегося)

5. **Глобальное обновление феромонов**. После того, как все муравьи завершат свою работу, делается глобальное обновление феромонов, которое учитывает локальное обновление феромона и испарение феромона.

6. **Повторение**. Процесс построения решений и обновления феромонов повторяется заданное количество раз или до достижения критерия остановки. После каждого прогона муравьев качество решения обычно улучшается.

2.4 Реализованные методы для муравьиного алгоритма

Ниже представлены классы и методы, требующиеся для реализации муравьиного алгоритма.

1. **Ant(int numRivers)**. Конструктор класса, создает муравья. Инициализирует маршрут муравья случайной перестановкой рек и устанавливает начальное значение длины маршрута в 0.

2. **reset()**. Сбрасывает посещенные реки и длину маршрута муравья.

3. **Graph(int numRivers)**. Конструктор класса, создает граф с заданным количеством рек $numRivers$. Заполняет матрицу времени перемещения случайными значениями от 1 до 20.

4. **GraphClass(int numGraphs, int numRivers)**: Конструктор класса, создает класс данных с заданным количеством графов $numGraphs$, каждый содержащий заданное количество рек $numRivers$.

5. **ACO(int numRivers, double alpha, double beta, int numAnts, const vector<vector<doul timeMatrix)**. Конструктор класса, инициализирует параметры муравьиного алгоритма, такие как количество рек, параметры *alpha* и *beta*, количество муравьев, и матрицу времени перемещения.

6. **runACO(int maxIterations)**. Запускает муравьиный алгоритм. В каждой итерации создаются муравьи, строится маршрут, обновляются феромоны и обновляется лучший маршрут.

7. **initializeAnts()**. Инициализирует муравьев, задавая каждому начальный порядок посещения рек.

8. **antTourConstruction()**. Строит маршрут для каждого муравья.

9. **selectNextRiver(int ant, const vector<bool>& visited, int currentRiver)**. Выбирает следующую реку для посещения муравьем с учетом феромонов и видимости.

10. **updatePheromones()**. Обновляет значения феромонов на ребрах графа.

11. **updateBestTour()**. Обновляет лучший маршрут, если найден новый лучший маршрут.

12. **calculateTourLength(const vector<int>& tour)**. Вычисляет длину маршрута муравья.

13. **getDeviation(const vector<int>& tour)**. Вычисляет отклонение длины маршрута от лучшей длины.

14. **getMaxDeviation()**. Возвращает максимальное отклонение длины маршрута среди всех муравьев.

3 Технологическая часть

Для реализации выбран язык C++. На листинге 1 представлена реализация программы (Реализация 1).

Листинг 1 – Исходный код

```
#include <iostream>
#include <vector>
#include <limits>
#include <cstdlib>
#include <ctime>
#include <algorithm>
#include <cmath>

using namespace std;

const double INF = numeric_limits<double>::infinity();

class Ant {
public:
    vector<int> tour;
    vector<bool> visited;
    double tourLength;

    Ant(int numRivers) : tourLength(0) {
        tour.resize(numRivers);
        visited.resize(numRivers, false);
        for (int i = 0; i < numRivers; ++i) {
            tour[i] = i;
        }
        for (int i = tour.size() - 1; i > 0; --i) {
            int j = rand() % (i + 1);
            swap(tour[i], tour[j]);
        }
    }

    void reset() {
        fill(visited.begin(), visited.end(), false);
        tourLength = 0;
    }
};
```

```
class Graph {
public:
    vector<vector<double>> timeMatrix;

    Graph(int numRivers) {
        timeMatrix.resize(numRivers, vector<double>(numRivers));
        for (int i = 0; i < numRivers; ++i) {
            for (int j = 0; j < numRivers; ++j) {
                if (i != j) {
                    // Задаем случайные значения времени перемещения
                    timeMatrix[i][j] = rand() % 20 + 1; // Для примера,
   случайные значения от 1 до 20
                }
            }
        }
    }
};

class GraphClass {
public:
    vector<Graph> graphs;

    GraphClass(int numGraphs, int numRivers) {
        graphs.resize(numGraphs, Graph(numRivers));
    }
};

class ACO {
public:
    ACO(int numRivers, double alpha, double beta, int numAnts, const
   vector<vector<double>>& timeMatrix)
        : numRivers(numRivers), alpha(alpha), beta(beta), numAnts(
   numAnts), timeMatrix(timeMatrix),
          pheromones(numRivers, vector<double>(numRivers, 1.0 /
   numRivers)),
          visibility(numRivers, vector<double>(numRivers, 1.0 /
   numRivers)), ants(numAnts), bestTourLength(INF) {}

    void runACO(int maxIterations);

    double getDeviation(const vector<int>& tour);
```

```
    double getMaxDeviation();

private:
    int numRivers;
    double alpha;
    double beta;
    int numAnts;
    const vector<vector<double>>& timeMatrix;
    vector<vector<double>> pheromones;
    vector<vector<double>> visibility;
    vector<vector<int>> ants;
    vector<int> bestTour;
    double bestTourLength;

    void initializeAnts();
    void antTourConstruction();
    void updatePheromones();
    void updateBestTour();
    double calculateTourLength(const vector<int>& tour);
    int selectNextRiver(int ant, const vector<bool>& visited, int
   currentRiver);
};

void ACO::initializeAnts() {
    for (auto& ant : ants) {
        ant.resize(numRivers);
        for (int i = 0; i < numRivers; ++i) {
            ant[i] = i;
        }
        random_shuffle(ant.begin() + 1, ant.end());
    }
}

void ACO::antTourConstruction() {
    for (auto& ant : ants) {
        vector<bool> visited(numRivers, false);
        ant[0] = rand() % numRivers;
        visited[ant[0]] = true;

        for (size_t j = 1; j < ant.size(); ++j) {
            ant[j] = selectNextRiver(ant[j], visited, ant[j - 1]);
```

```
            visited[ant[j]] = true;
        }
    }
}

int ACO::selectNextRiver(int ant, const vector<bool>& visited, int
   currentRiver) {
    double totalProbability = 0.0;

    for (int i = 0; i < numRivers; ++i) {
        if (!visited[i]) {
            totalProbability += pow(pheromones[currentRiver][i], alpha)
    * pow(visibility[currentRiver][i], beta);
        }
    }

    double randomValue = static_cast<double>(rand()) / RAND_MAX;
    double cumulativeProbability = 0.0;

    for (int i = 0; i < numRivers; ++i) {
        if (!visited[i]) {
            double probability = pow(pheromones[currentRiver][i], alpha
   ) * pow(visibility[currentRiver][i], beta) / totalProbability;

            cumulativeProbability += probability;

            if (randomValue <= cumulativeProbability) {
                return i;
            }
        }
    }

    // Если все реки посещены, возвращаем текущую реку
    return currentRiver;
}

void ACO::runACO(int maxIterations) {
    for (int iteration = 0; iteration < maxIterations; ++iteration) {
        initializeAnts();
        antTourConstruction();
        updatePheromones();
```

```
            updateBestTour();
        }

        cout << "Лучший маршрут: ";
        for (int river : bestTour) {
            cout << river << " ";
        }
        cout << endl << "Длина маршрута: " << bestTourLength << endl;

        double maxDeviation = getMaxDeviation();
        cout << "Максимально отклонение: " << maxDeviation << endl;
    }

    double ACO::calculateTourLength(const vector<int>& tour) {
        double length = 0.0;

        for (int i = 0; i < numRivers - 1; ++i) {
            length += timeMatrix[tour[i]][tour[i + 1]];
        }

        length += timeMatrix[tour.back()][tour.front()]; // Возвращение в
       начальную реку

        return length;
    }

    double ACO::getDeviation(const vector<int>& tour) {
        double tourLength = calculateTourLength(tour);
        return abs(tourLength - bestTourLength);
    }

    double ACO::getMaxDeviation() {
        double maxDeviation = 0.0;

        for (auto& ant : ants) {
            double deviation = getDeviation(ant);
            maxDeviation = max(maxDeviation, deviation);
        }

        return maxDeviation;
    }
```

```
void ACO::updatePheromones() {
    constexpr double evaporationRate = 0.5;

    for (int i = 0; i < numRivers; ++i) {
        for (int j = 0; j < numRivers; ++j) {
            pheromones[i][j] *= (1.0 - evaporationRate);

            for (int k = 0; k < numAnts; ++k) {
                double pheromoneChange = 0.0;

                if (find(ants[k].begin(), ants[k].end(), i) != ants[k].
   end() &&
                    find(ants[k].begin(), ants[k].end(), j) != ants[k].
   end()) {
                    pheromoneChange = 1.0 / calculateTourLength(ants[k
   ]);
                }

                pheromones[i][j] += pheromoneChange;
                pheromones[j][i] = pheromones[i][j];
            }
        }
    }
}

void ACO::updateBestTour() {
    for (auto& ant : ants) {
        double tourLength = calculateTourLength(ant);
        if (tourLength < bestTourLength) {
            bestTourLength = tourLength;
            bestTour = ant;
        }
    }
}


class BruteForceRiver {
public:
    BruteForceRiver(const vector<vector<double>>& timeMatrix)
        : timeMatrix(timeMatrix), numRivers(timeMatrix.size()) {}
```

```
    void solve() {
        vector<int> rivers(numRivers);
        for (int i = 0; i < numRivers; ++i) {
            rivers[i] = i;
        }

        double minTime = INF;
        vector<int> bestTour;

        do {
            double currentTime = calculateTourTime(rivers);
            if (currentTime < minTime) {
                minTime = currentTime;
                bestTour = rivers;
            }
        } while (next_permutation(rivers.begin() + 1, rivers.end()));

        cout << "Лучший маршрут: ";
        for (int river : bestTour) {
            cout << river << " ";
        }
        cout << endl << "Суммарное время: " << minTime << endl;
    }

private:
    const vector<vector<double>>& timeMatrix;
    const int numRivers;

    double calculateTourTime(const vector<int>& tour) const {
        double time = 0.0;
        for (int i = 0; i < numRivers - 1; ++i) {
            time += timeMatrix[tour[i]][tour[i + 1]];
        }
        time += timeMatrix[tour.back()][tour.front()]; // Возвращение в
   начальную реку
        return time;
    }
};
```

```
int main() {
    cout << "Какой алгоритм использовать?" << endl;
    cout << "1 -  полного перебора, 2 - муравьиный алгоритм, 3 - муравьиный
   алгоритм с графом из случайных значений" << endl;
    int n;
    cin >> n;
    if (n == 2) {
        srand(static_cast<unsigned>(time(0)));

        // Граф
        vector<vector<double>> timeMatrix = {
            {0, 2, 9, 11},
            {1, 0, 6, 4},
            {15, 7, 0, 8},
            {6, 3, 12, 0}
        };

        // Параметры муравьиного алгоритма
        int numRivers = timeMatrix.size();
        double alpha = 1.0;
        double beta = 2.0;
        int numAnts = 20;
        int maxIterations = 100;

        // Создание объекта ACO и запуск алгоритма
        ACO aco(numRivers, alpha, beta, numAnts, timeMatrix);
        aco.runACO(maxIterations);
    }
    else if (n == 3) {
        srand(static_cast<unsigned>(time(0)));

        GraphClass graphClass(3, 10); // создание класса данных с тремя
   графами

        // количество рек, alpha, beta, количество муравьев
        ACO aco(graphClass.graphs[0].timeMatrix.size(), 1.0, 2.0, 20,
   graphClass.graphs[0].timeMatrix);
        aco.runACO(10); // 10 запусков для каждого графа
    }
    else {
        // матрица времени перемещения между реками
```

```
    vector<vector<double>> timeMatrix = {
        {0, 2, 9, 10},
        {1, 0, 6, 4},
        {15, 7, 0, 8},
        {6, 3, 12, 0}
    };

    BruteForceRiver bruteForceRiver(timeMatrix);
    bruteForceRiver.solve();
    }
    return 0;
}
```

Примеры работы

На рисунках 1-3 представлены примеры работы метода перебора.

1. Входные файлы: граф с картой пермещения по рекам. Результат приведён на рис. 1.

```
Какой алгоритм использовать?
1 -  полного перебора, 2 - муравьиный алгоритм, 3 - муравьиный алгоритм с графом из случайных значений
1
Лучший маршрут: 0 3 2 1
Суммарное время: 18
```

Рис. 1 – Пример работы 1

2. Входные файлы: граф с картой пермещения по рекам. Результат приведён на рис. 2.

```
Какой алгоритм использовать?
1 -  полного перебора, 2 - муравьиный алгоритм, 3 - муравьиный алгоритм с графом из случайных значений
1
Лучший маршрут: 0 2 3 1
Суммарное время: 21
```

Рис. 2 – Пример работы 2

3. Входные файлы: граф с картой пермещения по рекам. Результат приведён на рис. 3.

```
Какой алгоритм использовать?
1 -  полного перебора, 2 - муравьиный алгоритм, 3 - муравьиный алгоритм с графом из случайных значений
1
Лучший маршрут: 0 2 1 3
Суммарное время: 268
```

Рис. 3 – Пример работы 3

На рисунках 4-6 представлены примеры работы муравьиного алгоритма. Входные данные такие же, как в примерах 1-3.

1. Входные файлы: граф с картой пермещения по рекам. Результат приведён на рис. 4.

```
Какой алгоритм использовать?
1 -  полного перебора, 2 - муравьиный алгоритм, 3 - муравьиный алгоритм с графом из случайных значений
2
Лучший маршрут: 2 1 0 3
Длина маршрута: 18
```

Рис. 4 – Пример работы 4

2. Входные файлы: граф с картой пермещения по рекам. Результат приведён на рис. 5.

```
Какой алгоритм использовать?
1 -  полного перебора, 2 - муравьиный алгоритм, 3 - муравьиный алгоритм с графом из случайных значений
2
Лучший маршрут: 2 3 1 0
Длина маршрута: 21
```

Рис. 5 – Пример работы 5

3. Входные файлы: граф с картой пермещения по рекам. Результат приведён на рис. 6.

```
Какой алгоритм использовать?
1 -  полного перебора, 2 - муравьиный алгоритм, 3 - муравьиный алгоритм с графом из случайных значений
2
Лучший маршрут: 0 2 1 3
Длина маршрута: 268
```

Рис. 6 – Пример работы 6

На рисунках 7-9 представлены примеры работы муравьиного алгоритма. Карта перемещения по рекам сгенерирована автоматически.

1. Входные файлы: автоматически сгенерированный граф с картой перемещения по рекам. Результат приведён на рис. 7.

```
Какой алгоритм использовать?
1 -  полного перебора, 2 - муравьиный алгоритм, 3 - муравьиный алгоритм с графом из случайных значений
3
Лучший маршрут: 3 6 8 2 7 1 5 0 9 4
Длина маршрута: 46
```

Рис. 7 – Пример работы 7

2. Входные файлы: автоматически сгенерированный граф с картой перемещения по рекам. Результат приведён на рис. 8.

```
Какой алгоритм использовать?
1 -  полного перебора, 2 - муравьиный алгоритм, 3 - муравьиный алгоритм с графом из случайных значений
3
Лучший маршрут: 0 6 7 4 1 8 5 9 2 3
Длина маршрута: 51
```

Рис. 8 – Пример работы 8

3. Входные файлы: автоматически сгенерированный граф с картой перемещения по рекам. Результат приведён на рис. 9.

```
Какой алгоритм использовать?
1 -  полного перебора, 2 - муравьиный алгоритм, 3 - муравьиный алгоритм с графом из случайных значений
3
Лучший маршрут: 9 6 1 5 8 4 7 3 0 2
Длина маршрута: 64
```

Рис. 9 – Пример работы 9

4 Исследовательская часть

4.1 Оценка трудоемкости

Муравьиный алгоритм.

1. **Инициализация муравьев и феромонов.** Инициализация муравьев: $O(\text{numAnts} \times \text{numRivers})$ — для каждого муравья создается массив и выполняется случайная перестановка. Инициализация феромонов: $O(\text{numRivers}^2)$ — матрица феромонов инициализируется.

2. **Построение маршрутов муравьев.** Выбор следующей реки: $O(\text{numRivers})$ — для каждой непосещенной реки рассчитывается вероятность, и выбирается следующая река.

3. **Обновление феромонов.** Обновление феромонов: $O(\text{numRivers}^2 \times \text{numAnts})$ — для каждой пары рек обновляется феромон для каждого муравья.

4. **Выбор лучшего маршрута.** Выбор лучшего маршрута: $O(\text{numAnts} \times \text{numRivers})$ — проход по всем муравьям для определения лучшего маршрута.

Итоговая трудоемкость:

$$O(\text{numRivers}^2 \times \text{numAnts} \times it),$$

где it — количество итераций.

Метод полного перебора.

1. **Генерация всех возможных перестановок.** Генерация перестановок: $O(\text{numRivers}!)$ - общее количество возможных перестановок.

2. *Вычисление длины каждого маршрута:* Вычисление длины маршрута: $O(\text{numRivers})$ - для каждой перестановки.

3. **Выбор лучшего маршрута.** Выбор лучшего маршрута: $O(\text{numRivers}!)$ - проход по всем перестановкам для определения лучшего маршрута.

Итоговая трудоемкость:

$$O(\text{numRivers}!).$$

4.2 Сравнительный анализ

Метод полного перебора.
Преимущества.

1. Гарантированно находит оптимальное решение, так как исследует все возможные комбинации.

Недостатки.

1. Вычислительно затратен при увеличении числа городов, так как количество возможных маршрутов растет факториально.
2. Неэффективен для задач с большим числом городов.

Муравьиный алгоритм.
Преимущества.

1. Применим для задач с большим числом городов, так как не требует перебора всех возможных вариантов.
2. Использует идеи муравьиного поведения для нахождения приближенного оптимального решения.

Недостатки.

1. Не гарантирует нахождение оптимального решения, так как основан случайных процессах.
2. Требует тщательной настройки параметров, таких как коэффициенты испарения феромонов и влияние расстояния.

Вывод.

В случае, когда возможно использование метода полного перебора и гарантированное нахождение оптимального решения является приоритетом, этот метод может быть предпочтителен для небольших задач. Однако, если решается задача с большим числом городов и важна эффективность, лучше выбрать муравьиный алгоритм, так как при правильной настройке для определенной задачи он дает хорошее соотношение точности и скорости.

Заключение

Цель достигнута: разработана программа для решения задачи коммивояжера несколькими способами. В результате выполнения лабораторной работы были выполнены все задачи.

1. Описан метод полного перебора для решения задачи коммивояжёра, указаны его преимущества и недостатки. Описана схема алгоритма для метода.

2. Описан метод решения задачи коммивояжёра на основе муравьиного алгоритма, указаны его преимущества и недостатки. Описана схема алгоритма для метода.

3. Выполнена оценка трудоёмкости составленных алгоритмов по разработанным схемам алгоритмов.

4. Проведен сравнительный анализ двух рассмотренных методов решения задачи коммивояжёра.

Список литературы

1. Семенов С. С., Педан А. В., Воловиков В. С., Климов И. С., Анализ трудоемкости различных алгоритмических подходов для решения задачи коммивояжера: научная статья 2017. – 16 с.