**МИНОБРНАУКИ РОССИИ**

**Санкт-Петербургский государственный**

**электротехнический университет**

**«ЛЭТИ» им. В.И. Ульянова (Ленина)**

**Кафедра МО ЭВМ**

отчет

**по лабораторной работе №4**

**по дисциплине «Построение и анализ алгоритмов»**

Тема: Задача Коммивояжера

|  |  |  |
| --- | --- | --- |
| Студент гр. 3344 |  | Коняева М. В. |
| Преподаватель |  | Фирсов М.А. |

Санкт-Петербург

2025

**Цель работы.**

Изучить принцип работы алгоритмов нахождения пути коммивояжера на графах.

**Задание.**

Группа вариантов 1-5:

Решить ЗК двумя методами в соответствии с вариантом: 1) Методом ВиГ. 2)Приближённым методом.

Дано: матрица весов графа, все веса неотрицательны; стартовая вершина.

Найти: путь коммивояжёра (последовательность вершин) и его стоимость.

Вариант 1.

МВиГ: Алгоритм Литтла.

Приближённый алгоритм: 2-приближение по МОД (АДО МОД).

Замечание к варианту 1. АДО МОД является 2-приближением только для евклидовой матрицы. Начинать обход МОД со стартовой вершины.

Замечание к алгоритму Литтла.

При каждом выборе дуги́ XY для ветвления получаем левую ветвь, в которой включена дуга XY и запрещена некоторая дуга Y'X' (правую ветвь, в которой запрещена дуга XY, сейчас не рассматриваем). Обратите внимание, что если выбрана дуга AB, то не обязательно запрещённой оказывается дуга BA. Пока не было шага по левой ветви, выбор дуги AB действительно приводит к запрещению дуги BA. Если же шаг по левой ветви уже был, то не всегда запретить надо обратную дугу. Например:

- Уже выбрана дуга AB, выбирается дуга BC —> запретить надо CA.

- Уже выбраны дуги AB и CD, выбирается дуга BC —> запретить надо DA.

**Описание алгоритма.**

Описание алгоритма Литтла.

Алгоритм Литла предназначен для решения задачи коммивояжера с использованием метода ветвей и границ. Входные данные представляют собой матрицу смежности графа.

Сначала выполняется редукция матрицы: для каждой строки находится минимальный элемент, который затем вычитается из всех элементов этой строки. Аналогичная операция выполняется для столбцов. Далее ищется “тяжёлый ноль” — нулевой элемент, у которого сумма минимальных значений в его строке и столбце максимальна.

На этом этапе происходит ветвление. В одной ветви из матрицы исключаются строка и столбец, содержащие выбранный ноль, а также запрещается движение, которое могло бы образовать цикл. В другой ветви этот ноль заменяется на бесконечность (∞). Стоимость каждой ветви фиксируется.

Когда находится первое допустимое решение, оно принимается за минимально возможное. В дальнейшем ветви с заведомо большим значением отсекаются. Если в процессе вычислений обнаруживается более выгодный маршрут, он заменяет текущий оптимальный. Такой подход позволяет избежать полного перебора всех возможных вариантов, обеспечивая нахождение оптимального решения.

Рекурсивный вызов метода branch: среднее количество узлов в дереве оценивается как O(c^n), где c < 2 (обычно c ≈ 1.26), но точное значение зависит от структуры матрицы). На каждом уровне рекурсии выполняются операции поиска нулевого элемента и редукции матрицы, каждая из которых имеет сложность O(n^2)

Следовательно, сложность рекурсивного вызова метода составляет O(c^n \* n^2), а общая сложность алгоритма также равна O(c^n \* n^2).

Оценка сложности по памяти:  
В худшем случае рекурсивный вызов метода branch приводит к полному обходу бинарного дерева, в котором число узлов приблизительно равно O(c^n), где n— количество вершин в графе. На каждом уровне рекурсии создается копия матрицы смежности, занимающая O(n^2) памяти. Таким образом, общая сложность по памяти составляет O(c^n \* n^2)

Описание алгоритма АДО МОД.

Алгоритм решения задачи коммивояжёра с использованием минимального остовного дерева (MST) начинается с построения MST с помощью алгоритма Прима. Сначала инициализируются массивы для хранения минимальных весов рёбер, родительских вершин и информации о том, включена ли вершина в MST. Начинаем с первой вершины, присваиваем ей минимальный вес 0 и помещаем её в приоритетную очередь. Затем, пока очередь не пуста, извлекаем вершины с минимальным весом и добавляем их в MST. Для каждой соседней вершины, если её ещё не добавили в MST и если вес рёбер меньше текущего минимального, обновляем её вес и добавляем в очередь с новым весом. После завершения этого этапа у нас есть минимальное остовное дерево, представляемое списком рёбер, соединяющих все вершины графа.

Затем, на основе полученных рёбер MST, строится список смежности для каждой вершины. После этого выполняется обход этого дерева в глубину (DFS). Для этого создаём стек и список посещённых вершин. Начинаем обход с заданной стартовой вершины, добавляем её в стек и продолжаем обход до тех пор, пока стек не станет пустым. Каждую посещённую вершину добавляем в путь обхода и отмечаем как посещённую. Соседей текущей вершины добавляем в стек в порядке возрастания веса рёбер. В результате мы получаем путь, который является обходом всех вершин в MST.

После выполнения обхода замыкаем путь, добавляя стартовую вершину в конец, чтобы получить полный цикл. Затем вычисляем суммарную стоимость маршрута, проходя по всем рёбрам пути и суммируя их веса. Если между двумя вершинами нет ребра, путь невозможен, и алгоритм сообщает об ошибке. В конце возвращается общая стоимость маршрута и сам путь, который является приближённым решением задачи коммивояжёра.

Сложность по времени алгоритма:

**Алгоритм Прима:**

* Сортировка рёбер: не требуется, так как используется приоритетная очередь для выбора минимальных рёбер из ограниченного множества.
* Приоритетная очередь (для выбора минимальных рёбер): O(logV) для каждой операции вставки и извлечения, что даёт O(ElogV) для всех рёбер.
* Создание множества вершин: O(V)
* Общий результат Прима: O(ElogV).
* Обход DFS по MST: O(V+E).
* Общая сложность: O(ElogV).

**Сложность по памяти алгоритма:**

* Матрица смежности (входной граф): O(V^2)
* Список рёбер (используется в алгоритме Прима): O(E)
* Приоритетная очередь: O(V)
* Хранение MST (в виде списка смежности): O(V+E)
* Структуры DFS (массив посещённых вершин, стек рекурсии): O(V)
* Общая сложность по памяти: O(V^2)

**Описание функций и структур данных.**

Класс Matrix представляет собой структуру данных для хранения матрицы стоимостей (например, для задачи коммивояжера или аналогичных задач). В нем определены следующие функции:

1. **Конструктор Matrix(int size)**: Инициализирует квадратную матрицу заданного размера, заполняя все элементы нулями.
2. **Конструктор Matrix(std::vector<std::vector<double>> matrix)**: Инициализирует матрицу с заранее заданными значениями.
3. **Конструктор по умолчанию Matrix()**: Пустой конструктор.
4. **setCell(int row, int col, int val)**: Устанавливает значение в ячейку матрицы в строке row и столбце col на значение val.
5. **getCell(int row, int col)**: Возвращает значение в ячейке матрицы по индексу строки row и столбца col.
6. **getSize()**: Возвращает размерность матрицы.
7. **deleteRowCol(int row, int col)**: Удаляет строку и столбец, заменяя все элементы в строке и столбце на INF, чтобы они не учитывались.
8. **getMinInRow(int row)**: Находит минимальный элемент в строке row.
9. **getMinInCol(int col)**: Находит минимальный элемент в столбце col.
10. **print()**: Выводит матрицу в консоль для отладки, заменяя значения INF на -1 для удобства отображения.
11. **generate(bool is\_symmetric = false, int min = 1, int max = 100)**: Генерирует случайную матрицу. Если is\_symmetric равно true, матрица будет симметричной (элементы выше главной диагонали будут зеркально отображены относительно главной диагонали).
12. **readMatrixFromConsole()**: Считывает матрицу с консоли, заменяя все -1 на INF.
13. **saveToFile(const std::string& filename)**: Сохраняет матрицу в файл с заданным именем filename.
14. **loadFromFile(const std::string& filename)**: Загружает матрицу из файла с заданным именем filename.

Реализован класс BranchAndBound для решения задачи коммивояжера методом ветвей и границ (Branch and Bound).

1. **reduce(Matrix& matrix):** Эта функция выполняет редукцию матрицы путём вычитания минимальных значений из каждой строки и каждого столбца. Для каждой строки из неё вычитается минимальный элемент, а для каждого столбца — минимальный элемент. Результатом является обновлённая матрица и вычисленная сумма сокращений, которая добавляется к общей стоимости.
2. **calcBestArc(Matrix& matrix):** Функция находит наилучшую дугу (рёбро) для дальнейшего включения в путь. Она ищет все нулевые элементы в матрице (кандидаты на выбор рёбер), затем вычисляет для каждого из них стоимость отказа (сумма минимальных элементов в строках и столбцах, в которых находятся эти нули). Выбирается дуга с максимальной стоимостью отказа.
3. **calcForbiddenArc(Matrix& matrix, std::vector<std::pair<int, int>>& path, int from, int to):** Эта функция определяет запрещённую дугу для пути. Для данного пути (последовательности рёбер) она находит все рёбра, которые ведут в вершину X и из вершины Y, и запрещает обратную дугу (например, если из вершины W есть ребро к X и из вершины Y выходит ребро к Z, то запрещается дуга Z -> W).
4. **calcBnB(Matrix& matrix, int cost, std::vector<std::pair<int, int>>& path, std::vector<std::pair<int, int>>& bestPath, int& bestCost):** Это рекурсивная функция, которая реализует сам алгоритм ветвей и границ. На каждом шаге она выполняет редукцию матрицы, вычисляет новую стоимость пути и, если эта стоимость не превышает наилучшего найденного решения, выбирает наилучшее ребро для продолжения пути. Функция работает с двумя ветвями: первой, когда добавляется новое ребро в путь, и второй, когда ребро исключается.
5. **formWay(std::vector<std::pair<int, int>>& arcs, int start\_vertex):** Эта функция восстанавливает путь из списка рёбер. Двигаясь по рёбрам вперёд и назад, она формирует последовательность вершин, представляющую путь от начальной вершины до конечной.

Реализован класс ModTSP, который решает задачу коммивояжёра с использованием метода минимального остовного дерева (MST). Описание основных функций класса:

1. **primMST(Matrix& matrix)**: Эта функция строит минимальное остовное дерево (MST) с помощью алгоритма Прима. На вход она получает матрицу, представляющую граф, а на выходе возвращает список рёбер MST. В процессе работы она использует приоритетную очередь для выбора наименьшего веса рёбер, чтобы добавить вершины в MST. Каждое добавление вершины сопровождается выводом информации о добавленных рёбрах и вершинах.
2. **dfsMST(const std::vector<std::vector<std::pair<int, double>>>& mstAdj, int start\_vertex)**: Эта функция выполняет обход минимального остовного дерева (MST) в глубину (DFS), начиная с указанной вершины. Она возвращает список вершин в порядке их посещения. Сначала в стек добавляется стартовая вершина, затем рекурсивно посещаются все соседние вершины. Соседи сортируются по весу рёбер, чтобы обход происходил в порядке увеличения веса.
3. **approximateTSP(Matrix& matrix, int start\_vertex)**: Эта функция находит приближённое решение задачи коммивояжёра с использованием минимального остовного дерева. Алгоритм включает в себя несколько шагов:
   * Строится минимальное остовное дерево (MST) с помощью алгоритма Прима.
   * Создаётся список смежности для этого дерева.
   * Выполняется обход дерева в глубину (DFS), начиная с указанной стартовой вершины.
   * Замыкается путь, добавляя стартовую вершину в конец маршрута.
   * Рассчитывается суммарная стоимость маршрута. Функция возвращает пару: суммарную стоимость маршрута и сам путь (список вершин в порядке посещения).

**Тестирование.**

Таблица 1 – Результаты тестирования

|  |  |  |  |
| --- | --- | --- | --- |
| № | Входные данные | Выходные данные | Комментарий |
| 1 | [INF1, 3, 4, 1],  [1, INF1, 3, 4],  [9, 2, INF1, 4],  [8, 9, 2, INF1]  Стартовая вершина: 0 Алгоритм Литтла | Оптимальный путь: [0, 3, 2, 1, 0]  Общая стоимость: 6 | Верно |
| 2 | [INF1, 20, 18, 12, 8],  [5, INF1, 14, 7, 11],  [12, 18, INF1, 6, 11],  [11, 17, 11, INF1, 12],  [5, 5, 5, 5, INF1]  Стартовая вершина: 0  Алгоритм Литтла | Оптимальный путь: [0, 4, 2, 3, 1, 0]  Общая стоимость: 41 | Верно |
| 3 | [inf, 12, 11, 3, 3]  [12, inf, 14, 11, 3]  [11, 14, inf, 6, 3]  [3, 11, 6, inf, 4]  [3, 3, 3, 4, inf]  Стартовая вершина: 0 Алгоритм Литтла Алгоритм АДО МОД | Оптимальный путь: [0, 1, 4, 2, 3, 0]  Общая стоимость: 27 Маршрут: [0, 3, 4, 1, 2, 0]  Приблизительная стоимость: 35 | Верно |
| 4 | [inf, 20, 2, 6, 16]  [20, inf, 9, 4, 19]  [2, 9, inf, 16, 19]  [6, 4, 16, inf, 6]  [16, 19, 19, 6, inf]  Стартовая вершина: 0 Алгоритм Литтла Алгоритм АДО МОД | Оптимальный путь: [0, 2, 1, 3, 4, 0]  Общая стоимость: 37  Маршрут: [0, 2, 3, 1, 4, 0]  Приблизительная стоимость: 57 | Верно |

Вывод с комментариями для 4 теста.  
Введите строку template (образец): dsds

Введите строку text (текст): dsdsdfdsdsfdssd

Загруженная матрица:

[inf, 20, 2, 6, 16]

[20, inf, 9, 4, 19]

[2, 9, inf, 16, 19]

[6, 4, 16, inf, 6]

[16, 19, 19, 6, inf]

Метод Литтла:

=============================

Текущий путь: [], Текущий город: 0, Текущая нижняя оценка: 0

Редукция матрицы:

Матрица до редукции:

[inf, 20, 2, 6, 16]

[20, inf, 9, 4, 19]

[2, 9, inf, 16, 19]

[6, 4, 16, inf, 6]

[16, 19, 19, 6, inf]

Из строки 0 вычли 2

Из строки 1 вычли 4

Из строки 2 вычли 2

Из строки 3 вычли 4

Из строки 4 вычли 6

Из столбца 0 ничего не вычли

Из столбца 1 ничего не вычли

Из столбца 2 ничего не вычли

Из столбца 3 ничего не вычли

Из столбца 4 вычли 2

Матрица после редукции:

[inf, 18, 0, 4, 12]

[16, inf, 5, 0, 13]

[0, 7, inf, 14, 15]

[2, 0, 12, inf, 0]

[10, 13, 13, 0, inf]

Общее сокращение: 20.0

Нижняя граница стоимости: 20.0, Текущая лучшая стоимость: inf

Для элемента [0][2] штраф 9

Для элемента [1][3] штраф 5

Для элемента [2][0] штраф 9

Для элемента [3][1] штраф 7

Для элемента [3][4] штраф 12

Для элемента [4][3] штраф 10

Выбрано ребро 3 -> 4 с максимальным коэффициентом 12

Ветка 1: Включаем ребро 3 -> 4, его вес: 6

Исключаем ребро 4 -> 3

=============================

Текущий путь: [(3, 4)], Текущий город: 4, Текущая нижняя оценка: 20.0

Редукция матрицы:

Матрица до редукции:

[inf, 18, 0, 4, inf]

[16, inf, 5, 0, inf]

[0, 7, inf, 14, inf]

[inf, inf, inf, inf, inf]

[10, 13, 13, inf, inf]

Из строки 0 ничего не вычли

Из строки 1 ничего не вычли

Из строки 2 ничего не вычли

Из строки 3 ничего не вычли

Из строки 4 вычли 10

Из столбца 0 ничего не вычли

Из столбца 1 вычли 3

Из столбца 2 ничего не вычли

Из столбца 3 ничего не вычли

Из столбца 4 ничего не вычли

Матрица после редукции:

[inf, 15, 0, 4, inf]

[16, inf, 5, 0, inf]

[0, 4, inf, 14, inf]

[inf, inf, inf, inf, inf]

[0, 0, 3, inf, inf]

Общее сокращение: 13.0

Нижняя граница стоимости: 33.0, Текущая лучшая стоимость: inf

Для элемента [0][2] штраф 7

Для элемента [1][3] штраф 9

Для элемента [2][0] штраф 4

Для элемента [4][0] штраф 0

Для элемента [4][1] штраф 4

Выбрано ребро 1 -> 3 с максимальным коэффициентом 9

Ветка 1: Включаем ребро 1 -> 3, его вес: 4

Исключаем ребро 4 -> 1

=============================

Текущий путь: [(3, 4), (1, 3)], Текущий город: 3, Текущая нижняя оценка: 33.0

Редукция матрицы:

Матрица до редукции:

[inf, 15, 0, inf, inf]

[inf, inf, inf, inf, inf]

[0, 4, inf, inf, inf]

[inf, inf, inf, inf, inf]

[0, inf, 3, inf, inf]

Из строки 0 ничего не вычли

Из строки 1 ничего не вычли

Из строки 2 ничего не вычли

Из строки 3 ничего не вычли

Из строки 4 ничего не вычли

Из столбца 0 ничего не вычли

Из столбца 1 вычли 4

Из столбца 2 ничего не вычли

Из столбца 3 ничего не вычли

Из столбца 4 ничего не вычли

Матрица после редукции:

[inf, 11, 0, inf, inf]

[inf, inf, inf, inf, inf]

[0, 0, inf, inf, inf]

[inf, inf, inf, inf, inf]

[0, inf, 3, inf, inf]

Общее сокращение: 4.0

Нижняя граница стоимости: 37.0, Текущая лучшая стоимость: inf

Для элемента [0][2] штраф 14

Для элемента [2][0] штраф 0

Для элемента [2][1] штраф 11

Для элемента [4][0] штраф 3

Выбрано ребро 0 -> 2 с максимальным коэффициентом 14

Ветка 1: Включаем ребро 0 -> 2, его вес: 2

Исключаем ребро 2 -> 0

=============================

Текущий путь: [(3, 4), (1, 3), (0, 2)], Текущий город: 2, Текущая нижняя оценка: 37.0

Редукция матрицы:

Матрица до редукции:

[inf, inf, inf, inf, inf]

[inf, inf, inf, inf, inf]

[inf, 0, inf, inf, inf]

[inf, inf, inf, inf, inf]

[0, inf, inf, inf, inf]

Из строки 0 ничего не вычли

Из строки 1 ничего не вычли

Из строки 2 ничего не вычли

Из строки 3 ничего не вычли

Из строки 4 ничего не вычли

Из столбца 0 ничего не вычли

Из столбца 1 ничего не вычли

Из столбца 2 ничего не вычли

Из столбца 3 ничего не вычли

Из столбца 4 ничего не вычли

Матрица после редукции:

[inf, inf, inf, inf, inf]

[inf, inf, inf, inf, inf]

[inf, 0, inf, inf, inf]

[inf, inf, inf, inf, inf]

[0, inf, inf, inf, inf]

Общее сокращение: 0.0

Нижняя граница стоимости: 37.0, Текущая лучшая стоимость: inf

Для элемента [2][1] штраф 0

Для элемента [4][0] штраф 0

Выбрано ребро 2 -> 1 с максимальным коэффициентом 0

Ветка 1: Включаем ребро 2 -> 1, его вес: 9

Исключаем ребро 4 -> 0

=============================

Текущий путь: [(3, 4), (1, 3), (0, 2), (2, 1)], Текущий город: 1, Текущая нижняя оценка: 37.0

Включенные ребра: [(3, 4), (1, 3), (0, 2), (2, 1)]

Завершённый путь: [0, 2, 1, 3, 4, 0], Итоговая стоимость: 37

Найден новый лучший путь!

Ветка 2: Исключаем ребро 2 -> 1

=============================

Текущий путь: [(3, 4), (1, 3), (0, 2)], Текущий город: 2, Текущая нижняя оценка: 37.0

Редукция матрицы:

Матрица до редукции:

[inf, inf, inf, inf, inf]

[inf, inf, inf, inf, inf]

[inf, inf, inf, inf, inf]

[inf, inf, inf, inf, inf]

[0, inf, inf, inf, inf]

Из строки 0 ничего не вычли

Из строки 1 ничего не вычли

Из строки 2 ничего не вычли

Из строки 3 ничего не вычли

Из строки 4 ничего не вычли

Из столбца 0 ничего не вычли

Из столбца 1 ничего не вычли

Из столбца 2 ничего не вычли

Из столбца 3 ничего не вычли

Из столбца 4 ничего не вычли

Матрица после редукции:

[inf, inf, inf, inf, inf]

[inf, inf, inf, inf, inf]

[inf, inf, inf, inf, inf]

[inf, inf, inf, inf, inf]

[0, inf, inf, inf, inf]

Общее сокращение: 0.0

Нижняя граница стоимости: 37.0, Текущая лучшая стоимость: 37

Обрезка ветви (граница выше текущего лучшего решения)

Ветка 2: Исключаем ребро 0 -> 2

=============================

Текущий путь: [(3, 4), (1, 3)], Текущий город: 3, Текущая нижняя оценка: 37.0

Редукция матрицы:

Матрица до редукции:

[inf, 11, inf, inf, inf]

[inf, inf, inf, inf, inf]

[0, 0, inf, inf, inf]

[inf, inf, inf, inf, inf]

[0, inf, 3, inf, inf]

Из строки 0 вычли 11

Из строки 1 ничего не вычли

Из строки 2 ничего не вычли

Из строки 3 ничего не вычли

Из строки 4 ничего не вычли

Из столбца 0 ничего не вычли

Из столбца 1 ничего не вычли

Из столбца 2 вычли 3

Из столбца 3 ничего не вычли

Из столбца 4 ничего не вычли

Матрица после редукции:

[inf, 0, inf, inf, inf]

[inf, inf, inf, inf, inf]

[0, 0, inf, inf, inf]

[inf, inf, inf, inf, inf]

[0, inf, 0, inf, inf]

Общее сокращение: 14.0

Нижняя граница стоимости: 51.0, Текущая лучшая стоимость: 37

Обрезка ветви (граница выше текущего лучшего решения)

Ветка 2: Исключаем ребро 1 -> 3

=============================

Текущий путь: [(3, 4)], Текущий город: 4, Текущая нижняя оценка: 33.0

Редукция матрицы:

Матрица до редукции:

[inf, 15, 0, 4, inf]

[16, inf, 5, inf, inf]

[0, 4, inf, 14, inf]

[inf, inf, inf, inf, inf]

[0, 0, 3, inf, inf]

Из строки 0 ничего не вычли

Из строки 1 вычли 5

Из строки 2 ничего не вычли

Из строки 3 ничего не вычли

Из строки 4 ничего не вычли

Из столбца 0 ничего не вычли

Из столбца 1 ничего не вычли

Из столбца 2 ничего не вычли

Из столбца 3 вычли 4

Из столбца 4 ничего не вычли

Матрица после редукции:

[inf, 15, 0, 0, inf]

[11, inf, 0, inf, inf]

[0, 4, inf, 10, inf]

[inf, inf, inf, inf, inf]

[0, 0, 3, inf, inf]

Общее сокращение: 9.0

Нижняя граница стоимости: 42.0, Текущая лучшая стоимость: 37

Обрезка ветви (граница выше текущего лучшего решения)

Ветка 2: Исключаем ребро 3 -> 4

=============================

Текущий путь: [], Текущий город: 0, Текущая нижняя оценка: 20.0

Редукция матрицы:

Матрица до редукции:

[inf, 18, 0, 4, 12]

[16, inf, 5, 0, 13]

[0, 7, inf, 14, 15]

[2, 0, 12, inf, inf]

[10, 13, 13, 0, inf]

Из строки 0 ничего не вычли

Из строки 1 ничего не вычли

Из строки 2 ничего не вычли

Из строки 3 ничего не вычли

Из строки 4 ничего не вычли

Из столбца 0 ничего не вычли

Из столбца 1 ничего не вычли

Из столбца 2 ничего не вычли

Из столбца 3 ничего не вычли

Из столбца 4 вычли 12

Матрица после редукции:

[inf, 18, 0, 4, 0]

[16, inf, 5, 0, 1]

[0, 7, inf, 14, 3]

[2, 0, 12, inf, inf]

[10, 13, 13, 0, inf]

Общее сокращение: 12.0

Нижняя граница стоимости: 32.0, Текущая лучшая стоимость: 37

Для элемента [0][2] штраф 5

Для элемента [0][4] штраф 1

Для элемента [1][3] штраф 1

Для элемента [2][0] штраф 5

Для элемента [3][1] штраф 9

Для элемента [4][3] штраф 10

Выбрано ребро 4 -> 3 с максимальным коэффициентом 10

Ветка 1: Включаем ребро 4 -> 3, его вес: 6

Исключаем ребро 3 -> 4

=============================

Текущий путь: [(4, 3)], Текущий город: 3, Текущая нижняя оценка: 32.0

Редукция матрицы:

Матрица до редукции:

[inf, 18, 0, inf, 0]

[16, inf, 5, inf, 1]

[0, 7, inf, inf, 3]

[2, 0, 12, inf, inf]

[inf, inf, inf, inf, inf]

Из строки 0 ничего не вычли

Из строки 1 вычли 1

Из строки 2 ничего не вычли

Из строки 3 ничего не вычли

Из строки 4 ничего не вычли

Из столбца 0 ничего не вычли

Из столбца 1 ничего не вычли

Из столбца 2 ничего не вычли

Из столбца 3 ничего не вычли

Из столбца 4 ничего не вычли

Матрица после редукции:

[inf, 18, 0, inf, 0]

[15, inf, 4, inf, 0]

[0, 7, inf, inf, 3]

[2, 0, 12, inf, inf]

[inf, inf, inf, inf, inf]

Общее сокращение: 1.0

Нижняя граница стоимости: 33.0, Текущая лучшая стоимость: 37

Для элемента [0][2] штраф 4

Для элемента [0][4] штраф 0

Для элемента [1][4] штраф 4

Для элемента [2][0] штраф 5

Для элемента [3][1] штраф 9

Выбрано ребро 3 -> 1 с максимальным коэффициентом 9

Ветка 1: Включаем ребро 3 -> 1, его вес: 4

Исключаем ребро 1 -> 4

=============================

Текущий путь: [(4, 3), (3, 1)], Текущий город: 1, Текущая нижняя оценка: 33.0

Редукция матрицы:

Матрица до редукции:

[inf, inf, 0, inf, 0]

[15, inf, 4, inf, inf]

[0, inf, inf, inf, 3]

[inf, inf, inf, inf, inf]

[inf, inf, inf, inf, inf]

Из строки 0 ничего не вычли

Из строки 1 вычли 4

Из строки 2 ничего не вычли

Из строки 3 ничего не вычли

Из строки 4 ничего не вычли

Из столбца 0 ничего не вычли

Из столбца 1 ничего не вычли

Из столбца 2 ничего не вычли

Из столбца 3 ничего не вычли

Из столбца 4 ничего не вычли

Матрица после редукции:

[inf, inf, 0, inf, 0]

[11, inf, 0, inf, inf]

[0, inf, inf, inf, 3]

[inf, inf, inf, inf, inf]

[inf, inf, inf, inf, inf]

Общее сокращение: 4.0

Нижняя граница стоимости: 37.0, Текущая лучшая стоимость: 37

Обрезка ветви (граница выше текущего лучшего решения)

Ветка 2: Исключаем ребро 3 -> 1

=============================

Текущий путь: [(4, 3)], Текущий город: 3, Текущая нижняя оценка: 33.0

Редукция матрицы:

Матрица до редукции:

[inf, 18, 0, inf, 0]

[15, inf, 4, inf, 0]

[0, 7, inf, inf, 3]

[2, inf, 12, inf, inf]

[inf, inf, inf, inf, inf]

Из строки 0 ничего не вычли

Из строки 1 ничего не вычли

Из строки 2 ничего не вычли

Из строки 3 вычли 2

Из строки 4 ничего не вычли

Из столбца 0 ничего не вычли

Из столбца 1 вычли 7

Из столбца 2 ничего не вычли

Из столбца 3 ничего не вычли

Из столбца 4 ничего не вычли

Матрица после редукции:

[inf, 11, 0, inf, 0]

[15, inf, 4, inf, 0]

[0, 0, inf, inf, 3]

[0, inf, 10, inf, inf]

[inf, inf, inf, inf, inf]

Общее сокращение: 9.0

Нижняя граница стоимости: 42.0, Текущая лучшая стоимость: 37

Обрезка ветви (граница выше текущего лучшего решения)

Ветка 2: Исключаем ребро 4 -> 3

=============================

Текущий путь: [], Текущий город: 0, Текущая нижняя оценка: 32.0

Редукция матрицы:

Матрица до редукции:

[inf, 18, 0, 4, 0]

[16, inf, 5, 0, 1]

[0, 7, inf, 14, 3]

[2, 0, 12, inf, inf]

[10, 13, 13, inf, inf]

Из строки 0 ничего не вычли

Из строки 1 ничего не вычли

Из строки 2 ничего не вычли

Из строки 3 ничего не вычли

Из строки 4 вычли 10

Из столбца 0 ничего не вычли

Из столбца 1 ничего не вычли

Из столбца 2 ничего не вычли

Из столбца 3 ничего не вычли

Из столбца 4 ничего не вычли

Матрица после редукции:

[inf, 18, 0, 4, 0]

[16, inf, 5, 0, 1]

[0, 7, inf, 14, 3]

[2, 0, 12, inf, inf]

[0, 3, 3, inf, inf]

Общее сокращение: 10.0

Нижняя граница стоимости: 42.0, Текущая лучшая стоимость: 37

Обрезка ветви (граница выше текущего лучшего решения)

Оптимальный путь: [0, 2, 1, 3, 4, 0]

Общая стоимость: 37

2-приближённое решение ADO MOD 2:

Минимальное остовное дерево (MST):

Добавлено ребро 0 - 2 с весом 2

Добавлено ребро 1 - 3 с весом 4

Добавлено ребро 0 - 3 с весом 6

Добавлено ребро 3 - 4 с весом 6

Дерево из MST {0: [2, 3], 1: [3], 2: [0], 3: [1, 0, 4], 4: [3]}

DFS обход дерева

Добавили вершину 0 в путь

Добавили вершину 2 в путь

Добавили вершину 3 в путь

Добавили вершину 1 в путь

Добавили вершину 4 в путь

Приближённый маршрут: [0, 2, 3, 1, 4, 0], Стоимость: 57

Маршрут: [0, 2, 3, 1, 4, 0]

Приблизительная стоимость: 57

**Выводы.**

Был реализован алгоритм, решающий задачу с использованием приближённого алгоритма АДО МОД и оптимального алгоритма Литтла.

**Приложение А**

**ИСХОДНЫЙ КОД ПРОГРАММЫ**

Имя файла: tsp.py

import copy

import random

import numpy as np

import matplotlib

import matplotlib.pyplot as plt

import networkx as nx

from typing import List, Tuple, Set, Dict

INF1 = float('inf')

class TSPSolver:

def \_\_init\_\_(self, matrix: List[List[float]], startVertex: int = 0):

"""Инициализация: сохраняем оригинальную матрицу, стартовую вершину, лучшую и приближённую стоимость"""

self.originalMatrix = copy.deepcopy(matrix)

self.n = len(matrix)

self.startVertex = startVertex

self.bestPath = None

self.bestCost = INF1

self.approxPath = None

self.approxCost = INF1

def reduceMatrix(self, matrix: List[List[float]]) -> Tuple[List[List[float]], float]:

"""Редукция матрицы: вычитание минимальных значений из строк и столбцов"""

reduction = 0.0

n = len(matrix)

print("\nРедукция матрицы:")

print("Матрица до редукции:")

for row in matrix:

print(row)

# Вычитание минимального значения в каждой строке

for i in range(n):

rowMin = min([x for x in matrix[i] if x != INF1], default=0)

if rowMin > 0 and rowMin != INF1:

for j in range(n):

if matrix[i][j] != INF1:

matrix[i][j] -= rowMin

print(f"Из строки {i} вычли {rowMin}")

reduction += rowMin

else:

print(f"Из строки {i} ничего не вычли")

# Вычитание минимального значения в каждом столбце

for j in range(n):

colMin = min([matrix[i][j] for i in range(n) if matrix[i][j] != INF1], default=0)

if colMin > 0 and colMin != INF1:

for i in range(n):

if matrix[i][j] != INF1:

matrix[i][j] -= colMin

print(f"Из столбца {j} вычли {colMin}")

reduction += colMin

else:

print(f"Из столбца {j} ничего не вычли")

print("Матрица после редукции:")

for row in matrix:

print(row)

print(f"Общее сокращение: {reduction}\n")

return matrix, reduction

def calculatePenalty(self, matrix: List[List[float]]) -> Dict[Tuple[int, int], float | int]:

"""Вычисляет штрафы для нулевых элементов матрицы."""

penalties = {}

for i in range(self.n):

for j in range(self.n):

if matrix[i][j] == 0:

rowMin = min((matrix[i][k] for k in range(self.n) if k != j and matrix[i][k] != INF1), default=0)

colMin = min((matrix[k][j] for k in range(self.n) if k != i and matrix[k][j] != INF1), default=0)

penalties[(i, j)] = rowMin + colMin

print(f"Для элемента [{i}][{j}] штраф {penalties[(i, j)]}")

return penalties

def reconstructPath(self, edges: List[Tuple[int, int]]) -> List[int]:

"""Восстановление пути по списку ребер."""

path\_r = [self.startVertex]

pathL = []

cond = True

currentCity = self.startVertex

while cond:

cond = False

for edge in edges:

if edge[0] == currentCity:

path\_r.append(edge[1])

currentCity = edge[1]

cond = True

break

cond = True

currentCity = self.startVertex

while cond:

cond = False

for edge in edges:

if edge[1] == currentCity:

pathL.insert(0, edge[0])

currentCity = edge[0]

cond = True

break

path = path\_r + pathL

if path[-1] != path[0]:

path.append(path[0])

return path

def findForbiddenEdge(self, path: List[Tuple[int, int]], newEdge: Tuple[int, int]) -> Tuple[int, int]:

"""

Определяет, какое ребро нужно запретить при добавлении newEdge (X → Y).

"""

X, Y = newEdge

# 1. Найти все рёбра, которые ведут к X (X уже посещён, и он не может быть вторично посещён)

cond = True

incoming\_to = X

while cond:

cond = False

for u, v in path:

if v == incoming\_to: # Если из вершины v есть ребро к X

cond = True

incoming\_to = u # Вершина, откуда приходит ребро к X (W → X)

break

# 2. Найти все рёбра, которые выходят из Y (Y уже посещён, и он не может быть вторично посещён)

cond = True

outgoing\_from = Y

while cond:

cond = False

for u, v in path:

if u == outgoing\_from: # Если из вершины Y выходит ребро к v

cond = True

outgoing\_from = v # Вершина, в которую ведёт ребро от Y (Y → Z)

break

# Если есть вход в X (W → X) и выход из Y (Y → Z), то мы должны запретить ребро Z → W

if incoming\_to != X or outgoing\_from != Y:

return outgoing\_from, incoming\_to # Запрещаем Z → W

# Если других связей нет, то просто запрещаем Y → X

return Y, X

def branch(self, path: List[Tuple[int, int]], unvisited: Set[int], currentCity: int, pathCost: float, matrix: List[List[float]]):

"""Рекурсивная функция метода ветвей и границ"""

print("\n=============================")

print(f"Текущий путь: {path}, Текущий город: {currentCity}, Текущая нижняя оценка: {pathCost}")

# Базовый случай: если все города посещены, добавляем ребро возврата к стартовой вершине

if len(path) == self.n - 1:

# Добавляем стоимость ребра возврата и ребро в список

completePath = path# + [(currentCity, self.startVertex)]

reconstructedPath = self.reconstructPath(completePath)

finalCost = sum(self.originalMatrix[reconstructedPath[i]][reconstructedPath[i + 1]] for i in range(len(reconstructedPath) - 1))

print(f"Включенные ребра: {path}")

print(f"Завершённый путь: {reconstructedPath}, Итоговая стоимость: {finalCost}")

if finalCost < self.bestCost:

print("Найден новый лучший путь!")

self.bestCost = finalCost

self.bestPath = reconstructedPath

return

# Шаг 1: Редукция матрицы

reducedMatrix, reduction = self.reduceMatrix(copy.deepcopy(matrix))

totalLowerBound = pathCost + reduction

print(f"Нижняя граница стоимости: {totalLowerBound}, Текущая лучшая стоимость: {self.bestCost}")

# Обрезка ветви, если нижняя граница уже хуже текущего лучшего решения

if totalLowerBound >= self.bestCost:

print("Обрезка ветви (граница выше текущего лучшего решения)")

return

# Шаг 2: Вычисление коэффициентов для выбора наилучшего ребра

penalties = self.calculatePenalty(reducedMatrix)

if not penalties:

print("Нет нулевых элементов, пропуск выбора рёбер.")

return

# Выбираем ребро с максимальным коэффициентом

(bestI, bestJ), maxPenalty = max(penalties.items(), key=lambda x: x[1])

print(f"Выбрано ребро {bestI} -> {bestJ} с максимальным коэффициентом {maxPenalty}")

# Шаг 3: Два варианта ветвления

forbiddenEdge = self.findForbiddenEdge(path, (bestI, bestJ))

forbiddenU, forbiddenV = forbiddenEdge

# Ветка 1: Включаем ребро (bestI -> bestJ)

matrixLeftBranch = copy.deepcopy(reducedMatrix)

for j in range(self.n):

matrixLeftBranch[bestI][j] = INF1 # Удаляем строку bestI

for i in range(self.n):

matrixLeftBranch[i][bestJ] = INF1 # Удаляем столбец bestJ

matrixLeftBranch[forbiddenU][forbiddenV] = INF1 # Запрещаем дугу

print(f"Ветка 1: Включаем ребро {bestI} -> {bestJ}, его вес: {self.originalMatrix[bestI][bestJ]}")

print(f"Исключаем ребро {forbiddenU} -> {forbiddenV}")

self.branch(path + [(bestI, bestJ)], unvisited - {bestJ, bestI}, bestJ, totalLowerBound, matrixLeftBranch)

# Ветка 2: Исключаем ребро (bestI -> bestJ)

matrixRightBranch = copy.deepcopy(reducedMatrix)

matrixRightBranch[bestI][bestJ] = INF1 # Запрещаем это ребро

print(f"Ветка 2: Исключаем ребро {bestI} -> {bestJ}")

self.branch(path, unvisited, currentCity, totalLowerBound, matrixRightBranch)

def kruskalsMST(self) -> list[tuple[int, int]]:

"""Алгоритм Краскала: строит минимальное остовное дерево"""

edges = [

(self.originalMatrix[i][j], i, j) # Создаём список всех рёбер (вес, вершина 1, вершина 2),

for i in range(self.n) for j in range(i + 1, self.n) # перебираем все возможные пары вершин (i, j)

if self.originalMatrix[i][j] != INF1 # и добавляем их в список, если между ними существует ребро (вес != INF1)

]

edges.sort()

# Создаём список родителей для каждой вершины (инициализируем каждой вершине её собственный родитель)

parent = list(range(self.n))

# Функция для нахождения корня вершины

def find(v: int) -> int:

# Если родитель вершины не сама вершина, рекурсивно находим её корень

if parent[v] != v:

parent[v] = find(parent[v])

return parent[v] # возвращаем корень

# Функция для объединения двух множеств

def union(v1: int, v2: int):

root1, root2 = find(v1), find(v2)

# Если корни разных множеств, объединяем их

if root1 != root2:

parent[root2] = root1 # соединяем корень второго множества с первым

mst = []

print("Минимальное остовное дерево (MST):")

for weight, u, v in edges:

# Если вершины u и v принадлежат разным компонентам связности (разные корни), то добавляем это ребро в MST

if find(u) != find(v):

union(u, v) # объединяем компоненты

mst.append((u, v)) # добавляем ребро в остовное дерево

print(f"Добавлено ребро {u} - {v} с весом {weight}") # выводим информацию о добавленном ребре

return mst

def approxTSP(self):

"""2-приближённый алгоритм для задачи коммивояжёра (TSP) на основе минимального остовного дерева (MST)"""

# Строим минимальное остовное дерево (MST) с помощью алгоритма Краскала

mst = self.kruskalsMST()

# Строим дерево (в виде списка смежности)

tree = {i: [] for i in range(self.n)}

for u, v in mst:

tree[u].append(v)

tree[v].append(u)

print(f"Дерево из MST {tree}")

print(f"DFS обход дерева")

def dfs(v, visited, path):

"""Обход дерева в порядке глубины (DFS)"""

visited.add(v)

path.append(v)

print(f"Добавили вершину {v} в путь")

for neighbor in sorted(tree[v], key=lambda x: self.originalMatrix[v][x]): # Идём по минимальному ребру

if neighbor not in visited:

dfs(neighbor, visited, path)

visited = set() # Множество для хранения посещённых вершин

path = [] # Список для хранения маршрута

# Запускаем обход дерева с начальной вершины

dfs(self.startVertex, visited, path)

# Добавляем начальную вершину в конец маршрута для замкнутого пути

path.append(self.startVertex)

totalCost = sum(self.originalMatrix[path[i]][path[i + 1]] for i in range(len(path) - 1))

self.approxPath = path

self.approxCost = totalCost

print(f"Приближённый маршрут: {path}, Стоимость: {totalCost}")

def visualize(self, little=True, ado=True):

matplotlib.use('TkAgg')

"""Визуализация MST, оптимального пути и 2-приближённого маршрута"""

G = nx.DiGraph() # Ориентированный граф

positions = {}

# Располагаем узлы по окружности

for i in range(self.n):

angle = 2 \* np.pi \* i / self.n

positions[i] = (np.cos(angle), np.sin(angle))

G.add\_node(i)

# Добавляем рёбра из исходной матрицы и создаём словарь весов

edgeLabels = {}

for i in range(self.n):

for j in range(i + 1, self.n):

if self.originalMatrix[i][j] != INF1:

G.add\_edge(i, j, weight=self.originalMatrix[i][j])

G.add\_edge(j, i, weight=self.originalMatrix[j][i])

edgeLabels[(i, j)] = self.originalMatrix[i][j]

edgeLabels[(j, i)] = self.originalMatrix[j][i]

if little:

# Визуализация оптимального пути

plt.figure(figsize=(8, 6))

optimalPathEdges = [(self.bestPath[i], self.bestPath[i + 1]) for i in range(len(self.bestPath) - 1)]

optimalLabels = {edge: edgeLabels[edge] for edge in optimalPathEdges} # Для оптимального пути

nx.draw\_networkx\_edges(G, pos=positions, edgelist=optimalPathEdges, width=3, edge\_color='blue', arrows=True, arrowsize=20, connectionstyle="arc3,rad=0.2")

# Рисуем узлы и рёбра для контекста

nx.draw(G, pos=positions, with\_labels=True, node\_color='lightblue', edge\_color='gray', node\_size=700, font\_size=12, connectionstyle="arc3,rad=0.1")

nx.draw\_networkx\_edge\_labels(G, pos=positions, edge\_labels=optimalLabels, font\_size=10) # Подписи для пути

plt.title("Оптимальный путь")

plt.show()

if ado:

# Визуализация 2-приближённого маршрута

plt.figure(figsize=(8, 6))

pathEdges = [(self.approxPath[i], self.approxPath[i + 1]) for i in range(len(self.approxPath) - 1)]

approxLabels = {edge: edgeLabels[edge] for edge in pathEdges} # Для приближённого маршрута

nx.draw\_networkx\_edges(G, pos=positions, edgelist=pathEdges, width=3, edge\_color='red', style="dashed", arrows=True, arrowsize=20, connectionstyle="arc3,rad=0.2")

# Рисуем узлы и рёбра для контекста

nx.draw(G, pos=positions, with\_labels=True, node\_color='lightblue', edge\_color='gray', node\_size=700, font\_size=12, connectionstyle="arc3,rad=0.1")

nx.draw\_networkx\_edge\_labels(G, pos=positions, edge\_labels=approxLabels, font\_size=10) # Подписи для маршрута

plt.title("2-приближённый маршрут")

plt.show()

def solve(self, little=True, ado=True, vizual=False):

"""Запуск алгоритмов"""

if little:

print("\nМетод Литтла:")

self.branch([], set(range(self.n)) - {self.startVertex}, self.startVertex, 0, self.originalMatrix)

print("Оптимальный путь:", self.bestPath)

print("Общая стоимость:", self.bestCost)

print()

if ado:

print("\n2-приближённое решение ADO MOD 2:")

self.approxTSP()

print("Маршрут:", self.approxPath)

print("Приблизительная стоимость:", self.approxCost)

if vizual:

self.visualize(little, ado)

def generateCostMatrix(n, symmetric=True, max\_weight=100):

"""Генерация матрицы смежности со случайными весами для задачи TSP"""

matrix = [[INF1] \* n for \_ in range(n)]

for i in range(n):

for j in range(i + 1 if symmetric else 0, n):

# Генерация случайного веса рёбер между вершинами

if i != j:

weight = random.randint(1, max\_weight)

matrix[i][j] = weight

if symmetric:

matrix[j][i] = weight # для симметричной матрицы

return matrix

def saveMatrixToFile(matrix, filename):

"""Сохранение матрицы в файл"""

with open(filename, 'w') as f:

for row in matrix:

f.write(' '.join(map(str, row)) + '\n')

def loadMatrixFromFile(filename):

"""Загрузка матрицы из файла"""

matrix = []

with open(filename, 'r') as f:

for line in f:

matrix.append(list(map(float, line.split())))

return matrix

if \_\_name\_\_ == "\_\_main\_\_":

# costMatrix = [

# [INF1, 20, 18, 12, 8],

# [5, INF1, 14, 7, 11],

# [12, 18, INF1, 6, 11],

# [11, 17, 11, INF1, 12],

# [5, 5, 5, 5, INF1]

# ]

# costMatrix = [

# [INF1, 3, 4, 1],

# [1, INF1, 3, 4],

# [9, 2, INF1, 4],

# [8, 9, 2, INF1]

# ]

# costMatrix = [

# [INF1, 20, 12, 11, 5],

# [20, INF1, 14, 17, 11],

# [12, 14, INF1, 6, 11],

# [11, 17, 6, INF1, 5],

# [5, 11, 11, 5, INF1]

# ]

n = 5

costMatrix = generateCostMatrix(n, symmetric=True, max\_weight=20)

#

# saveMatrixToFile(costMatrix, 'cost\_matrix.txt')

#

# costMatrix = loadMatrixFromFile('cost\_matrix.txt')

print("Загруженная матрица:")

for row in costMatrix:

print(row)

solver = TSPSolver(costMatrix, startVertex=0)

solver.solve(little=True, ado=True, vizual=True)