Федеральное государственное бюджетное образовательное учреждение высшего образования

«Пермский национальный исследовательский политехнический университет»

**Отчет по творческой работе на тему:**

**“Задача коммивояжера”**

Выполнил:

студент группы РИС-23-3б

Мачерет Николай Александрович

Пермь, 2024

**Введение**

Задача коммивояжера является одной из классических задач комбинаторной оптимизации и теории графов. Её цель — найти кратчайший путь, проходящий через все заданные города ровно один раз и вернуться в исходную точку. Эта задача важна для множества практических применений в логистике, планировании маршрутов и других областях, требующих оптимизации путей.

**Описание класса Graph**

В данной работе реализован класс **Graph**, предоставляющий набор методов для работы с графами и включающий алгоритм решения задачи коммивояжера.

**Определение класса**

class Graph **{**

protected**:**

int \_max\_size**;** // Максимальное количество вершин

int \_size**;** // Текущее количество вершин

int**\*\*** \_adjacency**;** // Матрица смежности (весов ребер)

void \_nullify**();** // Инициализация матрицы смежности

public**:**

Graph**(**int max\_size**);** // Конструктор с заданным максимальным размером

Graph**(**const Graph**&** g**);** // Конструктор копирования

**~**Graph**();** // Деструктор

int max\_size**()** const**;** // Получение максимального размера

int size**()** const**;** // Получение текущего размера

int adjacency**(**int i**,** int j**)** const**;** // Получение веса ребра между вершинами

vector**<**int**>** line**(**int i**)** const**;** // Получение строки матрицы смежности

vector**<**int**>** column**(**int j**)** const**;** // Получение столбца матрицы смежности

void add\_node**();** // Добавление вершины

void delete\_node**(**int n**);** // Удаление вершины

void edit\_edge**(**int n1**,** int n2**,** int weight**,** bool symmetric **=** **true);** // Редактирование веса ребра

void add\_edge**(**int n1**,** int n2**,** int weight **=** 1**);** // Добавление ребра

void delete\_edge**(**int n1**,** int n2**,** bool symmetric **=** **true);** // Удаление ребра

void read**(**istream**&** in**);** // Чтение графа из потока

void show**(**ostream**&** out**)** const**;** // Вывод графа в поток

vector**<**int**>** neighborhood**(**int n**)** const**;** // Получение соседей вершины

vector**<**int**>** dijkstra**(**int n**)** const**;** // Алгоритм Дейкстры

vector**<**int**>** salesman**()** const**;** // Алгоритм решения задачи коммивояжера

**};**

**Реализация методов класса**

1. **Конструктор и деструктор**

Конструктор инициализирует матрицу смежности, задает максимальный размер графа и вызывает метод **\_nullify()** для обнуления матрицы.

Graph**::**Graph**(**int max\_size**)** **{**

**this->**\_max\_size **=** max\_size**;**

**this->**\_adjacency **=** **new** int**\*** **[**max\_size**];**

**for** **(**int i **=** 0**;** i **<** max\_size**;** **++**i**)**

**this->**\_adjacency**[**i**]** **=** **new** int**[**max\_size**];**

**this->**\_nullify**();**

**}**

Конструктор копирования инициализирует новый граф на основе существующего.

Graph**::**Graph**(**const Graph**&** g**)** **{**

int max\_size **=** g**.**max\_size**();**

**this->**\_max\_size **=** max\_size**;**

**this->**\_adjacency **=** **new** int**\*** **[**max\_size**];**

**for** **(**int i **=** 0**;** i **<** max\_size**;** **++**i**)**

**this->**\_adjacency**[**i**]** **=** **new** int**[**max\_size**];**

**this->**\_size **=** g**.**size**();**

**for** **(**int i **=** 0**;** i **<** max\_size**;** **++**i**)**

**for** **(**int j **=** 0**;** j **<** max\_size**;** **++**j**)**

**this->**\_adjacency**[**i**][**j**]** **=** g**.**adjacency**(**i**,** j**);**

**}**

Деструктор освобождает выделенную под матрицу смежности память.

Graph**::~**Graph**()** **{**

**for** **(**int i **=** 0**;** i **<** **this->**\_size**;** **++**i**)**

**delete[]** **this->**\_adjacency**[**i**];**

**delete[]** **this->**\_adjacency**;**

**}**

1. **Методы управления графом**

Метод добавления вершины увеличивает размер графа и инициализирует соответствующие элементы матрицы смежности.

void Graph**::**add\_node**()** **{**

int**&** size **=** **this->**\_size**;**

size**++;**

**for** **(**int i **=** 0**;** i **<** size**;** **++**i**)** **{**

**this->**\_adjacency**[**size**][**i**]** **=** **-**1**;**

**this->**\_adjacency**[**i**][**size**]** **=** **-**1**;**

**}**

**}**

Метод удаления вершины сдвигает элементы матрицы, чтобы удалить выбранную вершину.

void Graph**::**delete\_node**(**int n**)** **{**

int**&** size **=** **this->**\_size**;**

**for** **(**int i **=** n**;** i **<** size **-** 1**;** **++**i**)**

**for** **(**int j **=** 0**;** j **<** n**;** **++**j**)**

**this->**\_adjacency**[**i**][**j**]** **=** **this->**\_adjacency**[**i **+** 1**][**j**];**

**for** **(**int j **=** n**;** j **<** size **-** 1**;** **++**j**)**

**for** **(**int i **=** 0**;** i **<** n**;** **++**i**)**

**this->**\_adjacency**[**i**][**j**]** **=** **this->**\_adjacency**[**i**][**j **+** 1**];**

**for** **(**int i **=** n**;** i **<** size **-** 1**;** **++**i**)**

**for** **(**int j **=** n**;** j **<** size **-** 1**;** **++**j**)**

**this->**\_adjacency**[**i**][**j**]** **=** **this->**\_adjacency**[**i **+** 1**][**j **+** 1**];**

**for** **(**int i **=** 0**;** i **<** size**;** **++**i**)** **{**

**this->**\_adjacency**[**size**][**i**]** **=** **-**1**;**

**this->**\_adjacency**[**i**][**size**]** **=** **-**1**;**

**}**

size**--;**

**}**

Метод редактирования ребра устанавливает вес ребра между двумя вершинами, и, если **symmetric == true**, делает это симметрично.

void Graph**::**edit\_edge**(**int n1**,** int n2**,** int weight**,** bool symmetric**)** **{**

**if** **(**symmetric**)** **{**

**this->**\_adjacency**[**n1**][**n2**]** **=** weight**;**

**this->**\_adjacency**[**n2**][**n1**]** **=** weight**;**

**}** **else** **{**

**this->**\_adjacency**[**n1**][**n2**]** **=** weight**;**

**}**

**}**

Методы добавления и удаления ребра используют **edit\_edge()** для изменения матрицы смежности.

void Graph**::**add\_edge**(**int n1**,** int n2**,** int weight**)** **{**

edit\_edge**(**n1**,** n2**,** weight**);**

**}**

void Graph**::**delete\_edge**(**int n1**,** int n2**,** bool symmetric**)** **{**

edit\_edge**(**n1**,** n2**,** **-**1**,** symmetric**);**

**}**

1. **Методы ввода/вывода и вспомогательные методы**

Метод **read()** считывает граф из входного потока.

void Graph**::**read**(**istream**&** in**)** **{**

**this->**\_nullify**();**

string line**;**

vector**<**int**>** line\_splited**;**

getline**(**in**,** line**);**

**this->**\_size **=** stoi**(**line**);**

**for** **(**int i **=** 0**;** i **<** **this->**\_size**;** **++**i**)** **{**

getline**(**in**,** line**);**

line\_splited **=** line\_to\_vec**(**line**);**

line\_splited**.**resize**(this->**\_size**);**

**for** **(**int j **=** 0**;** j **<** **this->**\_size**;** **++**j**)** **{**

**if** **(**line\_splited**[**j**]** **==** 0**)**

**this->**\_adjacency**[**i**][**j**]** **=** **-**1**;**

**else**

**this->**\_adjacency**[**i**][**j**]** **=** line\_splited**[**j**];**

**}**

**}**

**}**

Метод **show()** выводит граф в выходной поток.

void Graph**::**show**(**ostream**&** out**)** const **{**

int size **=** **this->**\_size**;**

out **<<** size **<<** endl**;**

**for** **(**int i **=** 0**;** i **<** size**;** **++**i**)** **{**

**for** **(**int j **=** 0**;** j **<** size **-** 1**;** **++**j**)**

**if** **(this->**\_adjacency**[**i**][**j**]** **<** 0**)**

out **<<** "0 "**;**

**else**

out **<<** **this->**\_adjacency**[**i**][**j**]** **<<** " "**;**

**if** **(this->**\_adjacency**[**i**][**size **-** 1**]** **<** 0**)**

out **<<** "0" **<<** endl**;**

**else**

out **<<** **this->**\_adjacency**[**i**][**size **-** 1**]** **<<** endl**;**

**}**

**}**

Методы **line()** и **column()** возвращают строки и столбцы матрицы смежности соответственно.

vector**<**int**>** Graph**::**line**(**int i**)** const **{**

vector**<**int**>** res**(this->**\_size**,** 0**);**

**if** **(**i **>=** **this->**size**())**

**return** res**;**

**for** **(**int j **=** 0**;** j **<** **this->**\_size**;** **++**j**)** **{**

res**[**j**]** **=** **this->**\_adjacency**[**i**][**j**];**

**}**

**return** res**;**

**}**

vector**<**int**>** Graph**::**column**(**int j**)** const **{**

vector**<**int**>** res**(this->**\_size**,** 0**);**

**if** **(**j **>=** **this->**size**())**

**return** res**;**

**for** **(**int i **=** 0**;** i **<** **this->**\_size**;** **++**i**)** **{**

res**[**i**]** **=** **this->**\_adjacency**[**i**][**j**];**

**}**

**return** res**;**

**}**

**Алгоритм решения задачи коммивояжера**

Алгоритм решения задачи коммивояжера основан на методе ветвей и границ и включает редукцию строк и столбцов матрицы смежности.

1. **Редукция строк и столбцов**

Метод **lines\_reduction()** редуцирует строки, уменьшая все элементы строки на минимальный элемент.

void lines\_reduction**(**Graph**&** g**)** **{**

int size **=** g**.**size**();**

**for** **(**int i **=** 0**;** i **<** size**;** **++**i**)** **{**

int min **=** INT\_MAX**;**

**for** **(**auto e **:** g**.**line**(**i**))**

**if** **(**e **<** min **&&** e **>=** 0**)**

min **=** e**;**

**for** **(**int j **=** 0**;** j **<** size**;** **++**j**)** **{**

**if** **(**g**.**\_adjacency**[**i**][**j**]** **>=** 0**)**

g**.**\_adjacency**[**i**][**j**]** **-=** min**;**

**}**

**}**

**}**

Метод **columns\_reduction()** редуцирует столбцы, уменьшая все элементы столбца на минимальный элемент.

void columns\_reduction**(**Graph**&** g**)** **{**

int size **=** g**.**size**();**

**for** **(**int i **=** 0**;** i **<** size**;** **++**i**)** **{**

int min **=** INT\_MAX**;**

**for** **(**auto e **:** g**.**column**(**i**))**

**if** **(**e **<** min **&&** e **>=** 0**)**

min **=** e**;**

**for** **(**int j **=** 0**;** j **<** size**;** **++**j**)** **{**

**if** **(**g**.**\_adjacency**[**j**][**i**]** **>=** 0**)**

g**.**\_adjacency**[**j**][**i**]** **-=** min**;**

**}**

**}**

**}**

1. **Оценка нулевых элементов**

Метод **zeros\_score()** оценивает нулевые элементы для выбора ребра, исключение которого приведет к наибольшему увеличению стоимости пути.

pair**<**int**,** int**>** zeros\_score**(**Graph**&** g**)** **{**

int size **=** g**.**size**();**

int max **=** **-**1**;**

pair**<**int**,** int**>** res**;**

**for** **(**int i **=** 0**;** i **<** size**;** **++**i**)** **{**

**for** **(**int j **=** 0**;** j **<** size**;** **++**j**)** **{**

**if** **(**g**.**adjacency**(**i**,** j**)** **==** 0**)** **{**

int min\_line **=** INT\_MAX**;**

**for** **(**int k **=** 0**;** k **<** size**;** **++**k**)**

**if** **(**k **!=** j **&&** g**.**adjacency**(**i**,** k**)** **>=** 0**)**

min\_line **=** min**(**min\_line**,** g**.**adjacency**(**i**,** k**));**

int min\_column **=** INT\_MAX**;**

**for** **(**int k **=** 0**;** k **<** size**;** **++**k**)**

**if** **(**k **!=** i **&&** g**.**adjacency**(**k**,** j**)** **>=** 0**)**

min\_column **=** min**(**min\_column**,** g**.**adjacency**(**k**,** j**));**

**if** **(**max **<** min\_line **+** min\_column**)** **{**

max **=** min\_line **+** min\_column**;**

res **=** make\_pair**(**i**,** j**);**

**}**

**}**

**}**

**}**

**return** res**;**

**}**

1. **Построение пути**

Основной метод **salesman()** реализует решение задачи коммивояжера, включающий редукцию матрицы и построение оптимального маршрута.

vector**<**int**>** Graph**::**salesman**()** const **{**

Graph g **=** **\*this;**

lines\_reduction**(**g**);**

columns\_reduction**(**g**);**

vector**<**pair**<**int**,** int**>>** path**;**

vector**<**int**>** res**;**

int size **=** g**.**size**();**

**for** **(**int i **=** 0**;** i **<** size**;** **++**i**)** **{**

**if** **(**path**.**size**()** **==** size**)**

**break;**

pair**<**int**,** int**>** zero **=** zeros\_score**(**g**);**

path**.**push\_back**(**zero**);**

g**.**edit\_edge**(**zero**.**first**,** zero**.**second**,** **-**1**);**

g**.**edit\_edge**(**zero**.**second**,** zero**.**first**,** **-**1**);**

**}**

res**.**push\_back**(**path**[**0**].**first**);**

res**.**push\_back**(**path**[**0**].**second**);**

**while** **(**res**.**size**()** **!=** path**.**size**()** **+** 1**)** **{**

**for** **(**auto e **:** path**)** **{**

**if** **(**res**.**back**()** **==** e**.**first**)**

res**.**push\_back**(**e**.**second**);**

**}**

**}**

**return** res**;**

**}**

**Тестирование**

Для тестирования разработанного алгоритма для задачи коммивояжера был использован текстовый файл: **Graph\_Salesman.txt**

**Пример содержимого файла Graph\_Salesman.txt:**

4

0 5 3 3

5 0 9 7

3 9 0 4

3 7 4 0

**Пример кода для тестирования задачи коммивояжера:**

int main**()** **{**

Graph g**(**MAX\_SIZE**);**

ifstream f**;**

f**.**open**(**"Graph\_Salesman.txt"**);**

g**.**read**(**f**);**

f**.**close**();**

vector**<**int**>** path **=** g**.**salesman**();**

**for** **(**auto e **:** path**)**

cout **<<** **(**e **+** 1**)** **<<** " "**;**

cout **<<** endl**;**

cout **<<** path\_lenght**(**g**,** path**)** **<<** endl**;**

**}**

**Ожидаемый результат:**

1 2 4 3 1

19

**Заключение**

В данной работе представлен класс **Graph**, реализующий основные методы работы с графами и алгоритм решения задачи коммивояжера. Алгоритм использует метод ветвей и границ с предварительной редукцией строк и столбцов матрицы смежности, что позволяет эффективно находить оптимальный маршрут для небольших графов.

В современном мире задача коммивояжера может быть применена в различных практических областях, требующих оптимизации маршрутов и логистических задач.

**UML-диаграмма**

