МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ

ФЕДЕРАЛЬНОЕ государственное БЮДЖЕТНОЕ

образовательное учреждение

высшего образования

«НОВОСИБИРСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ»

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Кафедра вычислительной техники



**РАСЧЕТНО-ГРАФИЧЕСКАЯ РАБОТА**

**по дисциплине:** *«Алгоритмы и структуры данных»*

**Вариант 10, 1**

Выполнила:Проверил:

Студентка гр. *АВТ-709*, *АВТФ* *Старший преподаватель*

*Андерсон Дарья Романенко Т. А.*

«\_\_\_» \_\_\_\_\_\_ 2019 г.«\_\_\_» \_\_\_\_\_\_ 2019 г.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

(подпись) (подпись)

Новосибирск

2019

Оглавление

[Цели работы 3](#_Toc27907865)

[Общее задание 3](#_Toc27907866)

[Задача 1. Задание 4](#_Toc27907867)

[Задача 2. Задание 5](#_Toc27907868)

[1. Диаграмма взаимосвязи объектов, реализующих АТД «Простой граф», вспомогательных объектов, АТД задач 1- 2 6](#_Toc27907869)

[2. АТД «Простой граф» 7](#_Toc27907870)

[3. Клиентское определение класса «Простой граф» 10](#_Toc27907871)

[4. АТД «Вершина графа» 10](#_Toc27907872)

[5. Клиентское определение класса «Вершина графа» 12](#_Toc27907873)

[6. АТД «Ребро графа» 12](#_Toc27907874)

[7. Клиентское описание АТД «Ребро графа» 14](#_Toc27907875)

[8. АТД «Итератор вершин графа» 14](#_Toc27907876)

[9. Клиентское описание АТД «Итератор вершин графа» 15](#_Toc27907877)

[10. АТД «Итератор ребер графа» 16](#_Toc27907878)

[11. Клиентское описание АТД «Итератор ребер графа» 17](#_Toc27907879)

[12. АТД «Итератор исходящих ребер графа» 17](#_Toc27907880)

[13. Клиентское описание АТД «Итератор исходящих ребер графа» 18](#_Toc27907881)

[14. АТД «Задача 1» 18](#_Toc27907882)

[15. Клиентское описание АТД «Задача 1» 19](#_Toc27907883)

[16. АТД «Задача 2» 19](#_Toc27907884)

[17. Клиентское описание АТД «Задача 2» 19](#_Toc27907885)

[18. Краткое описание алгоритма, теоретическая оценка трудоемкости задачи 1 19](#_Toc27907886)

[19. Краткое описание алгоритма, теоретическая оценка трудоемкости задачи 2 20](#_Toc27907887)

[20. Заключение 20](#_Toc27907888)

[Список использованных источников 21](#_Toc27907889)

[Приложение 22](#_Toc27907890)

# **Цели работы**

Освоение технологии разработки комплексного программного обеспечения для решения задач в различных прикладных областях.

# **Общее задание**

Разработать АТД «Простой граф»:

* **Конструктор ( )** по умолчанию: создает пустой L - граф с нулевым числом
* вершин и ребер,
* **Конструктор (V, D, F)** создает граф с **V** вершинами, без ребер, типа **D** (ориентированный / неориентированный), формы представления **F** (L- граф/M-граф),
* **Конструктор (V, E, D, F)** создает граф с **V** вершинами, с **E** случайными ребрами, типа
* **D** (ориентированный / неориентированный), формы представления **F** (L- граф/M-граф),
* **Конструктор (G)** - конструктор копирования создает объект – копию графа **G**,
* **Деструктор ( )** уничтожает внутренние структуры графа,
* **V ( )** - возвращает число вершин в графе,
* **E ( )** - возвращает число ребер в графе,
* **Directed ( )** - возвращает тип графа (ориентированный / неориентированный)
* **Dense ( )** - возвращает форму представления графа (L- граф / M- граф),
* **K ( )** - возвращает коэффициент насыщенности графа,
* **ToListGraph ( )** преобразует граф к L- графу,
* **ToMatrixGraph ( )** преобразует граф к M- графу,
* **InsertV ( )** добавляет вершину к графу и возвращает адрес дескриптора вновь созданной вершины,
* **DeleteV (v)** - удаляет вершину из графа, заданную адресом дескриптора **v**,
* **InsertE(v1, v2)** - добавляет ребро (**v1**, **v2**) к графу, соединяющую вершины, заданные адресами дескрипторов **v1** и **v2**, и возвращает адрес дескриптора вновь созданного ребра - **e**,
* **DeleteE (v1, v2)** удаляет ребро, соединяющее вершины, заданные адресами дескрипторов **v1** и **v2**,
* **GetEdge (v1, v2)** возвращает адрес дескриптора ребра соединяющего вершины, заданные дескрипторами **v1** и **v2**.

Разработать ассоциированные с графом типы:

* АТД «Дескриптор вершины графа»:

**name** – имя вершины,

**data** – данные, связанные с вершиной,

**index** – индекс вершины в структуре графа или -1

* АТД «Дескриптор вершины графа»:

**Конструктор ()**: поле **name** не определено, поле **data** не определено,

**Конструктор (name, data)**: **name** - имя вершины, **data** - данные, связанные с вершиной,

**GetName ( )** - возвращает имя вершины,

**GetData ( )** - возвращает данные, связанные с вершиной,

**SetName (name )** – задает имя вершины,

**SetData (data)** – записывает данные data в дескриптор вершины.

* АТД «Дескриптор ребра графа»:

**v1** - дескриптор вершины, из которой исходит ребро,

**v2** - дескриптор вершины, в которую входит ребро,

**w** - вес ребра,

**data** - данные, связанные с ребром,

**Конструктор (v1, v2)**: **v1** - дескриптор вершины, из которой исходит ребро, **v2** - дескриптор вершины, в которую входит ребро,

**Конструктор (v1, v2, w)**: **v1** - дескриптор вершины, из которой исходит ребро, **v2** - дескриптор вершины, в которую входит ребро, **w** - вес ребра,

**Конструктор (v1, v2, w, data)**: **v1** - дескриптор вершины, из которой исходит ребро, **v2** - дескриптор вершины, в которую входит ребро, **w** - вес ребра, **data** - данные, связанные с ребром

**v1( )** - возвращает дескриптор вершины, из которой исходит ребро,

**v2( )** - возвращает дескриптор вершины, в которую входит ребро,

**GetW ( )** - возвращает вес ребра,

**SetW (w)** - изменение веса ребра,

**GetData ( )** - возвращает данные, связанные с ребром,

**SetData (data)** - изменение данных, связанных с ребром.

* АТД «Итератор вершин графа»:

**Конструктор ( )** - создает итератор вершин графа,

**beg ( )** - возвращает итератор, установленный на первую вершину графа,

**end ( )** - возвращает итератор, соответствующий окончанию переходов итератора,

**operator ++** - переход к следующей вершине графа,

**operator \*** - возвращает дескриптор вершины графа, на которую указывает итератор.

* АТД «Итератор ребер графа»:

**Конструктор ( )** - создает итератор ребер графа,

**beg ( )** - возвращает итератор, установленный на первое ребро графа,

**end ( )** - возвращает итератор, соответствующий окончанию переходов итератора,

**operator ++** - переход к следующему ребру графа,

**operator \*** - возвращает дескриптор ребра графа, на которое указывает итератор.

* АТД «Итератор исходящих ребер вершины»:

**Конструктор (v)** - создает итератор исходящих ребер графа для вершины, заданной дескриптором **v**,

**beg ( )** - возвращает итератор, установленный на первое исходящее ребро вершины,

**end ( )** - возвращает итератор, соответствующий окончанию переходов итератора,

**operator ++** - переход к следующему исходящему ребру,

**operator \*** - возвращает дескриптор исходящего ребра вершины, на которое указывает итератор.

# **Задача 1. Задание**

Спроектировать и реализовать шаблонный класс для АТД «Задача 1» в соответствии с вариантом и использовать для решения задачи на неориентированном или ориентированном графе.

Интерфейс АТД «Задача 1» включает операции:

* **Конструктор (g)** - создает объект задачи 1, ассоциированный с графом **g**, и выполняет решение задачи для графа **g**,
* **Конструктор (T)** - конструктор копирования создает копию объекта – задачи **T**,
* **Деструктор ( )** - уничтожает внутренние структуры объекта задачи,
* **Set (g)** – связывает объект задачи с графом **g** и выполняет решение задачи 1 для графа **g**,
* **Restart ( )** – повторно выполняет решение задачи 1 для графа **g**,
* **Result ( )** – возвращает результат решения задачи 1.

Вариант 10: поиск простого цикла заданной длины, включающего заданную вершину.

# **Задача 2. Задание**

Спроектировать и реализовать шаблонный класс для АТД «Задача 2» в соответствии с вариантом и использовать для решения задачи на взвешенном графе.

Интерфейс АТД «Задача 2» включает операции:

* **Конструктор (g)** - создает объект задачи 2, ассоциированный с графом **g**, и выполняет решение задачи для графа **g**,
* **Конструктор (T)** - конструктор копирования создает копию объекта – задачи **T**,
* **Деструктор ( )** - уничтожает внутренние структуры объекта задачи,
* **Set (g)** – связывает объект задачи с графом **g** и выполняет решение задачи 2 для графа **g**,
* **Restart ( )** – повторно выполняет решение задачи 2 для графа **g**,
* **Result( )** – возвращает результат решения задачи 2.

Вариант 1: определение диаметра и пути, соответствующего диаметру, на основе алгоритма Флойда.

# **Диаграмма взаимосвязи объектов, реализующих АТД «Простой граф», вспомогательных объектов, АТД задач 1- 2**

# **АТД «Простой граф»**

Простой граф представляет собой граф, который может иметь структуру, представленную либо в виде матрицы смежностей, либо в виде списка смежностей. Граф может быть ориентированным и неориентированным. Класс предоставляет доступ к хранимым в нём вершинам и рёбрам, типы которых задаются пользователем. Доступ к вершинам осуществляется по их дескрипторам. Доступ к рёбрам осуществляется со дескриптору исходящей вершины ребра и дескриптору входящей вершины ребра.

***Данные:***

*Параметры:*

Коллекция вершин;

Коллекция рёбер;

Тип графа **dense**;

Ориентированность графа **directed**.

*Структура данных:*

Матрица/Массив списков.

***Операции:***

*Конструктор без параметров*

**Вход:** нет

**Предусловие:** нет

**Процесс:** создание пустого неориентированного графа списочной формы

**Выход:** нет

**Постусловия:** создан пустой граф

*Конструктор с параметрами*

**Вход:** количество вершин (**V**), ориентированность/не ориентированность графа (**D**), матричная/списочная форма графа (**F**)

**Предусловие:** нет

**Процесс:** создание графа типа **F**, ориентированности **D**, содержащего **V** вершин

**Выход:** нет

**Постусловия:** создан граф типа **F**, ориентированности **D**, с **V** вершинами

*Конструктор с параметрами*

**Вход:** количество вершин (**V**), количество рёбер (**E**), ориентированность/не ориентированность графа (**D**), матричная/списочная форма графа (**F**)

**Предусловие:** нет

**Процесс:** создание графа типа **F**, ориентированности **D**, содержащего **V** вершин и **Е** случайно сгенерированных рёбер

**Выход:** нет

**Постусловия:** создан граф типа **F**, ориентированности **D**, с **V** вершинами и **Е** случайными рёбрами

*Конструктор копирования*

**Вход:** копируемый граф **G**

**Предусловие:** нет

**Процесс:** копирование данных графа **G**

**Выход:** нет

**Постусловия:** создана копия графа **G**

*Деструктор*

**Вход:** нет

**Предусловия:** нет

**Процесс:** удаление графа

**Выход:** нет

**Постусловия:** граф удален

*Получение числа вершин в графе*

**Вход:** нет

**Предусловие:** нет

**Процесс:** получение размера коллекции вершин графа

**Выход:** число вершин в графе

**Постусловия:** нет

*Получение числа вершин в графе*

**Вход:** нет

**Предусловие:** нет

**Процесс:** получение размера коллекции рёбер графа

**Выход:** число ребер в графе

**Постусловия:** нет

*Получение коэффициента заполняемости графа*

**Вход:** нет

**Предусловие:** нет

**Процесс:** подсчёт коэффициента заполняемости

**Выход:** коэффициент заполняемости графа

**Постусловия:** нет

*Получение направленности графа*

**Вход:** нет

**Предусловие:** нет

**Процесс:** проверка параметра направленности графа - **directed**

**Выход:** true – если граф ориентированный, false – если неориентированный

**Постусловия:** нет

*Получение формы представления графа*

**Вход:** нет

**Предусловие:** нет

**Процесс:** проверка параметра формы графа - **dense**

**Выход:** true - если граф матричного типа (**М**), false - если списочного типа (**L**)

**Постусловия:** нет

*Преобразование графа к М-типу*

**Вход:** нет

**Предусловие:** нет

**Процесс:** преобразование внутренней структуры графа к матричному **М**-типу

**Выход:** нет

**Постусловия:** граф принял **M**-форму

*Преобразование графа к L-типу*

**Вход:** нет

**Предусловие:** нет

**Процесс:** преобразование внутренней структуры графа к списочному **L**-типу

**Выход:** нет

**Постусловия:** граф принял **L**-форму

*Вставка вершины в граф*

**Вход:** нет

**Предусловие:** нет

**Процесс:** создание новой вершины **V** и её вставка в конец коллекции вершин

**Выход:** дескриптор созданной вершины **V**

**Постусловия:** в граф добавлена новая вершина **V**

*Удаление вершины из графа*

**Вход:** дескриптор удаляемой вершины **V**

**Предусловие:** существование вершины **V** в графе

**Процесс:** удаление вершины с дескриптором **V** из графа

**Выход:** true – при выполнении предусловия, иначе – false

**Постусловия:** вершина **V** удалена из графа

*Получение вершины из графа*

**Вход:** имя вершины name

**Предусловия:** существование вершины с именем name в графе

**Процесс:** получение дескриптора вершины **V** с именем name

**Выход:** дескриптор вершины – при выполнении предусловия, иначе – генерация исключения

**Постусловия:** нет

*Вставка ребра в граф*

**Вход:** дескрипторы исходящей вершины **V1** и входящей вершины **V2**, которые будут соденинены ребром

**Предусловия:** вершины с дескрипторами **V1** и **V2** содержатся в графе между ними нет ребра, идущего из вершины с дескриптором **V1** в вершину с дескриптором **V2**

**Процесс:** создание нового ребра, исходящего из вершины с дескриптором **V1** в веришну с дескрипотором **V2** и её вставка в коллекцию рёбер

**Выход:** дескриптор созданного ребра – при выполнении предусловия, иначе – генерация исключения

**Постусловия:** ребро добавлено в граф

*Удаление ребра из графа*

**Вход:** дескрипторы исходящей вершины **V1** и входящей вершины **V2**, соединённых удаляемым ребром.

**Предусловия:** вершины с дескрипторами **V1** и **V2** содержатся в графе и соединены ребром, исходящим из вершины с дескриптором **V1** и входящим в вершину с дескриптором **V2**

**Процесс:** удаление ребра, соединяющего вершины с дескрипторами **V1** и **V2**

**Выход:** true – при выполнении предусловия, иначе – false

**Постусловия:** из графа удалено ребро

*Проверка на существование ребра в графе*

**Вход:** дескрипторы исходящей вершины **V1** и входящей вершины **V2**, идентифицирующих ребро

**Предусловия:** вершины с дескрипторами **V1** и **V2** содержатся в графе.

**Процесс:** проверка на существование ребра между вершинами с дескрипоторами **V1** и **V2**

**Выход:** true – если вершины с дескрипторами **V1** и **V2** соединены ребром, иначе – false

**Постусловия:** нет

*Получение ребра из графа*

**Вход:** дескрипторы исходящей вершины **V1** и входящей вершины **V2**, идентифицирующих ребро.

**Предусловия:** вершины с дескрипторами **V1** и **V2** содержатся в графе и соединены ребром, исходящим из вершины с дескриптором **V1** и входящим в вершину с дескриптором **V2**

**Процесс:** получение ребра между вершинами **V1** и **V2**

**Выход:** идентификатор ребра – если выполнено предусловие, иначе – генерация исключения

**Постусловия:** нет

# **Клиентское определение класса «Простой граф»**

template <class VertexT, class EdgeT>

class Graph {

Graph(); //Конструктор пустого L-графа

Graph(int vertexCount, bool directed, bool dense); //Конструктор с заданным числом вершин, заданной ориентированности и заданного типа

Graph(int vertexCount, int edgeCount, bool directed, bool dense); //Конструктор с заданным числом вершин, случайных рёбер, заданной ориентированности и типа

Graph(const Graph<VertexT, EdgeT> &G); //Конструктор копирования

~Graph(); //Деструктор

int getVertexCount(); //Число вершин

int getEdgeCount(); //Число рёбер

double getRatio(); //Заполненность графа

bool isDirected(); //Направленность графа

bool isDense(); //Тип графа

void toDenseGraph(); //Преобразование к M-типу

void toSparseGraph(); //Преобразование к L-типу

VertexT\* insertVertex(); //Вставка новой вершины

bool deleteVertex(VertexT \*v); //Удаление вершины

EdgeT\* insertEdge(VertexT \*v1, VertexT \*v2); //Вставка нового ребра

bool deleteEdge(VertexT \*v1, VertexT \*v2) //Удаление ребра

bool hasEdge(VertexT \*v1, VertexT \*v2); //Проверка на существованиеие ребра

EdgeT\* getEdge(VertexT \*v1, VertexT \*v2); //Получение ребра

};

# **АТД «Вершина графа»**

Разработанный абстрактный тип данных представляет собой вершину, предназначенную для вставки в граф. Вершина имеет поле имени вершины и поле данных вершины, типы которых определяются параметрами шаблона.

Получение и установка параметров вершины осуществляется с помощью её методов.

***Данные:***

*Параметры*

Имя вершины **name**.

Данные вершины **data**.

***Операции:***

*Конструктор*

**Вход**: нет.

**Предусловия**: нет.

**Процесс**: создание пустой вершины с неопределёнными именем и данными.

**Выход**: нет.

**Постусловия**: создана пустая вершина.

*Конструктор*

**Вход**: имя вершины **name** и данные вершины **data**.

**Предусловия**: нет.

**Процесс**: создание вершины с именем **name** и данными **data**.

**Выход**: нет.

**Постусловия**: создана вершина с заданными именем и данными.

*Установка имени вершины*

**Вход**: имя вершины **name**.

**Предусловия**: нет.

**Процесс**: установка имени вершины в значение **name**.

**Выход**: нет.

**Постусловия**: имя вершины установлено в значение **name**.

*Установка данных вершины*

**Вход**: данные вершины **data**.

**Предусловия**: нет.

**Процесс**: установка данных вершины в значение **data**.

**Выход**: нет.

**Постусловия**: данные вершины установлены в значение **data**.

*Получение данных вершины*

**Вход**: нет.

**Предусловия**: данные были установлены.

**Процесс**: обращение к полю объекта **data**.

**Выход**: данные вершины **data** или неопределённое значение, если не выполнено предусловие.

**Постусловия**: нет

*Получение имени вершины*

**Вход**: нет.

**Предусловия**: имя было установлено.

**Процесс**: обращение к полю объекта **name**.

**Выход**: имя вершины **name**.

**Постусловия**: нет.

*Провекра установленности имени вершины*

**Вход**: нет.

**Предусловия**: нет.

**Процесс**: проверка установленности поля **name**.

**Выход**: true, если имя уставнолено, иначе false.

**Постусловия**: нет.

*Провекра установленности данных вершины*

**Вход**: нет.

**Предусловия**: нет.

**Процесс**: проверка установленности поля **data**.

**Выход**: true, если данные уставнолены, иначе false.

**Постусловия**: нет.

# **Клиентское определение класса «Вершина графа»**

template <class NameT, class DataT>

class Vertex {

NameT name; //Имя вершины

DataT data; //Данные вершины

bool nameSet, dataSet;//признаки установки имени и данных

public:

Vertex(); //Конструктор пустой вершины

Vertex(NameT name, DataT data);//Конструктор с параметрами

//имени и данных

void setName(NameT name); //Установка имени

void setData(DataT data); //Установка данных

NameT getName(); //Получение имени

DataT getData(); //Получение данных

bool isNameSet(); //Проверка установленности имени

bool isDataSet(); //Проверка установленности данных

};

# **АТД «Ребро графа»**

Абстрактный тип данных представляет собой ребро, предназначенное для соединения вершин в графе. Ребро имеет поле веса ребра, поле данных ребра, поле исходящей вершины и поле входящей вершины, типы которых определяются параметрами шаблона.

Получение и установка параметров ребра осуществляется с помощью его методов.

***Данные:***

*Параметры*

Вес вершины **weight**.

Данные вершины **data**.

***Операции:***

*Конструктор*

**Вход**: дескрипторы вершин **V1** и **V2**, которые соединяются ребром.

**Предусловия**: нет.

**Процесс**: создание ребра с неопределёнными весом и данными между **V1** и **V2**.

**Выход**: нет.

**Постусловия**: создано ребро с неопределёнными весом и данными между **V1** и **V2**.

*Конструктор*

**Вход**: дескрипторы вершин **V1** и **V2**, которые соединяются ребром, вес **weight**.

**Предусловия**: нет.

**Процесс**: создание ребра с весом **weight** между **V1** и **V2**.

**Выход**: нет.

**Постусловия**: создано ребро с весом **weight** и неопределёнными данными между **V1** и **V2**.

*Конструктор*

**Вход**: дескрипторы вершин **V1** и **V2**, которые соединяются ребром, вес **weight,** данные **data**.

**Предусловия**: нет.

**Процесс**: создание ребра с весом **weight** и данными **data** между **V1** и **V2**.

**Выход**: нет.

**Постусловия**: создано ребро с весом **weight** и данными **data** между **V1** и **V2**.

*Установка веса ребра*

**Вход**: вес ребра **weight**.

**Предусловия**: нет.

**Процесс**: установка веса ребра в значение **weight**.

**Выход**: нет.

**Постусловия**: вес ребра установлен в значение **weight**.

*Установка данных ребра*

**Вход**: данные ребра **data**.

**Предусловия**: нет.

**Процесс**: установка данных ребра в значение **data**.

**Выход**: нет.

**Постусловия**: данные ребра установлены в значение **data**.

*Получение данных ребра*

**Вход**: нет.

**Предусловия**: данные были установлены.

**Процесс**: обращение к полю ребра **data**.

**Выход**: данные ребра **data** или неопределённое значение, если не выполнено предусловие

**Постусловия**: нет.

*Получение веса ребра*

**Вход**: нет.

**Предусловия**: вес был установлен.

**Процесс**: обращение к полю объекта **weight**.

**Выход**: вес вершины **weight**.

**Постусловия**: нет.

*Проверка установленности веса ребра*

**Вход**: нет.

**Предусловия**: нет.

**Процесс**: проверка установленности поля **weight**.

**Выход**: true, если вес уставнолен, иначе false.

**Постусловия**: нет.

*Проверка установленности данных ребра*

**Вход**: нет.

**Предусловия**: нет.

**Процесс**: проверка установленности поля **data**.

**Выход**: true, если данные уставнолены, иначе false.

**Постусловия**: нет.

*Получение исходящей вершины ребра*

**Вход**: нет.

**Предусловия**: нет.

**Процесс**: обращение к полю исходящей вершины ребра.

**Выход**: дескриптор исходящей вершины ребра.

**Постусловия**: нет.

*Получение входящей вершины ребра*

**Вход**: нет.

**Предусловия**: нет.

**Процесс**: обращение к полю входящей вершины ребра.

**Выход**: дескриптор входящей вершины ребра.

**Постусловия**: нет.

# **Клиентское описание АТД «Ребро графа»**

template <class VertexT, class WeightT, class DataT>

class Edge {

VertexT \*v1, \*v2; //Вершины, которые соединяет ребро

WeightT weight; //Вес

DataT data; //Данные ребра

bool weightSet, dataSet;//признаки установки веса и данных

public:

Edge(VertexT \*v1, VertexT \*v2); //Ребро с неустановленным

//весом и данными

Edge(VertexT \*v1, VertexT \*v2, WeightT weight); //Ребро с

//установленным весом

Edge(VertexT \*v1, VertexT \*v2, WeightT weight, DataT data);

//Ребро с установленными весом и данными

void setWeight(WeightT weight); //Установка веса

void setData(DataT data); //Установка данных

WeightT getWeight(); //Получение веса

DataT getData(); //Получение данных

bool isWeightSet(); //Установленность веса

bool isDataSet();}; //Установленность данных

VertexT\* getVertex1(); //получение дескриптора исх. вершины

VertexT\* getVertex2();//получение дескриптора вх. вершины

# **АТД «Итератор вершин графа»**

Абстрактный тип данных представляет собой вложенный в АТД «Простой граф» класс, предназначенный для последовательного доступа к вершинам графа. Итератор можно устанавливать на первую вершину графа в порядке обхода, на поледнюю вершину в порядке обхода, получать доступ к текущей вершине, а также переходить к следующей.

***Данные:***

*Параметры*

Текущая вершина в графе **current**.

Указатель на граф **graph**.

Признак конца просмотра **end**.

***Операции:***

*Конструктор*

**Вход**: указатель на граф **g**.

**Предусловия**: нет.

**Процесс**: создание итератора вершин для графа **G**.

**Выход**: нет.

**Постусловия**: создан итератор вершин для графа **G**.

*Установка итератора на начало*

**Вход**: нет.

**Предусловия**: граф не пуст.

**Процесс**: установка итератора на первую вершину графа.

**Выход**: нет.

**Постусловия**: итератор установлен на первую вершину или не установлен при невыполненном предусловии.

*Переход к следующей вершине*

**Вход**: нет.

**Предусловия**: итератор установлен.

**Процесс**: сдвиг итератора на следующую вершину.

**Выход**: нет.

**Постусловия**: итератор установлен на следующую вершину при выполнении предусловия или в состоянии «не установлен», если предусловие не выполнено.

*Проверка неустановленности итератора*

**Вход**: нет.

**Предусловия**: нет.

**Процесс**: проверка состояния итератора.

**Выход**: true – если итератор не установлен, иначе – false.

**Постусловия**: нет.

*Доступ к текущей вершине*

**Вход**: нет.

**Предусловия**: итератор установлен.

**Процесс**: доступ к текущей вершине.

**Выход**: дескриптор вершины – при выполненном предусловии, иначе – исключение.

**Постусловия**: нет.

*Утановка итератора на конец*

**Вход**: нет.

**Предусловия**: нет.

**Процесс**: установка итератора на последнюю вершину.

**Выход**: нет.

**Постусловия**: нет.

# **Клиентское описание АТД «Итератор вершин графа»**

class VertexIterator {

Graph<VertexT, EdgeT> \*graph; //Обрабатываемый граф

bool end; //Состояние итератора

int current; //Текущее положение

public:

VertexIterator(Graph<VertexT, EdgeT> &g); //Конструктор

bool begin(); //Установка на начало

bool operator++(); //Сдвиг на след. элемент

bool onEnd(); //Проверка выхода из коллекции

VertexT\* operator\*(); //Получение доступа к вершине

void toend(); //установка на последнюю вершину

};

# **АТД «Итератор ребер графа»**

Абстрактный тип данных представляет собой вложенный в АТД «Простой граф» класс, предназначенный для последовательного доступа к рёбрам графа. Итератор можно устанавливать на первое ребро графа в порядке обхода, на поледнее ребро в порядке обхода, получать доступ к текущему ребру, а также переходить к следующему.

***Данные:***

*Параметры*

Текущее ребро графа.

Указатель на граф **graph**.

***Операции:***

*Конструктор*

**Вход**: указатель на граф **g**.

**Предусловия**: нет.

**Процесс**: создание итератора рёбер для графа **g**.

**Выход**: нет.

**Постусловия**: создан итератор рёбер для графа **g**.

*Установка итератора на начало*

**Вход**: нет.

**Предусловия**: граф содержит рёбра.

**Процесс**: установка итератора на первое ребро графа.

**Выход**: **true**, если выполнено предусловие, иначе **false**.

**Постусловия**: итератор установлен на первое ребро или не установлен при невыполненном предусловии.

*Переход к следующему ребру*

**Вход**: нет.

**Предусловия**: итератор установлен.

**Процесс**: сдвиг итератора на следующее ребро.

**Выход**: true – если выполнено предусловие, иначе – false.

**Постусловия**: итератор установлен на следующее ребро при выполнении предусловия или не установлен, если рёбра закончились.

*Проверка неустановленности итератора*

**Вход**: нет.

**Предусловия**: нет.

**Процесс**: проверка состояния итератора.

**Выход**: true – если итератор не установлен, иначе – false.

**Постусловия**: нет.

*Доступ к текущему ребру*

**Вход**: нет.

**Предусловия**: итератор установлен.

**Процесс**: доступ к текущему ребру.

**Выход**: дескриптор ребра – выполнении предусловии, иначе – генерация исключения.

**Постусловия**: нет.

*Утановка итератора на конец*

**Вход**: нет.

**Предусловия**: проверка наличия рёбер.

**Процесс**: установка итератора на последнее ребро.

**Выход**: true – если итератора установлен на последнее ребро, false – при невыполнении предусловия.

**Постусловия**: нет.

# **Клиентское описание АТД «Итератор ребер графа»**

class EdgeIterator {

Graph<VertexT, EdgeT> \*graph; //Обрабатываемый граф

bool end; //Состояние итератора

int curV1, curV2; //Текущее положение

public:

EdgeIterator(Graph<VertexT, EdgeT> &g); //Конструктор

bool begin(); //Установка на начало

bool operator++(); //Сдвиг на след. элемент

bool onEnd(); //Проверка выхода из коллекции

EdgeT\* operator\*(); //Получение доступа к ребру

bool toend(); //установка на последнее ребро

};

# **АТД «Итератор исходящих ребер графа»**

Абстрактный тип данных представляет собой вложенный в АТД «Простой граф» класс, предназначенный для последовательного доступа к исходящим рёбрам вершин графа. Итератор можно устанавливать на первое исходящее ребро вершины графа в порядке обхода, на поледнее исходящее ребро вершины графа в порядке обхода, получать доступ к текущему ребру вершины графа, а также переходить к следующему.

***Данные:***

*Параметры:*

Текущее ребро графа.

Указатель на граф **graph**.

Номер рассматриваемой вершины **curV1**.

***Операции:***

*Конструктор*

**Вход**: указатель на граф **G** и дескриптор вершины **V**.

**Предусловия**: проверка наличия вершины **V** в графе.

**Процесс**: создание итератора исходящих рёбер для вершины **V** графа **G**.

**Выход**: нет.

**Постусловия**: создан итератор рёбер для вершины **V** графа **G** – при выполенном предусловии, иначе – генерация исключения.

*Установка итератора на начало*

**Вход**: нет.

**Предусловия**: вершина содержит исходящие рёбра.

**Процесс**: установка итератора на первое исходящее ребро вершины.

**Выход**: true – если выполнено предусловие, иначе – false.

**Постусловия**: итератор установлен на первое исходящее ребро.

*Переход к следующему исходящему ребру*

**Вход**: нет.

**Предусловия**: итератор установлен.

**Процесс**: сдвиг итератора на следующее исходящее ребро.

**Выход**: true –если выполнено предусловие, иначе – false.

**Постусловия**: итератор установлен на следующее исходящее ребро при вып. предусловии или не установлен, если рёбра закончились.

*Проверка неустановленности итератора*

**Вход**: нет.

**Предусловия**: нет.

**Процесс**: проверка состояния итератора.

**Выход**: true – если итератор не установлен, иначе – false.

**Постусловия**: нет.

*Доступ к текущему ребру:*

**Вход**: нет.

**Предусловия**: итератор установлен.

**Процесс**: доступ к текущему ребру.

**Выход**: дескриптор исходящего ребра – при выполненном предусловии, иначе – генерация исключения.

**Постусловия**: нет.

*Утановка итератора на конец*

**Вход**: нет.

**Предусловия**: проверка наличия исходящих рёбер.

**Процесс**: установка итератора на последнее исходящее ребро.

**Выход**: true – если итератора установлен на последнее исходящее ребро, false – при невыполнении предусловия.

**Постусловия**: нет.

# **Клиентское описание АТД «Итератор исходящих ребер графа»**

class OutputEdgeIterator {

Graph<VertexT, EdgeT> \*graph; //Обрабатываемый граф

bool end; //Состояние итератора

int curV1, curV2; //Текущее положение

public:

OutputEdgeIterator (Graph<VertexT, EdgeT> &g, VertexT &v); //Конструктор

bool begin(); //Установка на начало

bool operator++(); //Сдвиг на след. элемент

bool onEnd(); //Проверка выхода из коллекции

EdgeT\* operator\*();//Получение доступа к исходящему ребру

bool toend(); //установка на последнее исходящее ребро

};

# **АТД «Задача 1»**

Поиск простого цикла заданной длины из заданной вершины (модификаця Гамильтонов цикл).

***Данные:***

*Параметры*

Обрабатываемый граф **graph**

***Операции:***

*Конструктор*

**Вход:** граф для решения задачи

**Предусловие:** нет

**Процесс:** решение задачи для данного графа

**Выход:** нет

**Постусловия:** построен граф

*Конструктор копирования*

**Вход:** граф для решения задачи

**Предусловие:** нет

**Процесс:** создание копии объекта задачи

**Выход:** нет

**Постусловия:** создана копия объекта задачи

*Получение решения*

**Вход:** нет

**Предусловие:** наличие простого (гамильтонова) цикла

**Процесс:** поиск простого (гамильтонова) цикла в графе

**Выход:** список ребер в порядке их вхождения в цикл или NULL при невыполнении предусловия

**Постусловия:** нет

# **Клиентское описание АТД «Задача 1»**

class Tasks{

private:

Graph<VertexT, EdgeT>\* HCgraph; //Обрабатываемый граф

public:

Tasks(Graph<VertexT, EdgeT> \*g); //Конструктор

Tasks(const Task1<VertexT, EdgeT> &t1); //Конструктор копирования

~Tasks();//Деструктор

void set(); //Связывает объект задачи с графом g и выполняет решение задачи 1 для графа g

void restart();//Повторно выполняет решение задачи 1 для графа g

int\* result();//Возвращает результат решения задачи 1

};

# **АТД «Задача 2»**

Определение диаметра и пути, соответствующего диаметру, на основе алгоритма Флойда.

***Данные:***

*Параметры:*

Обрабатываемый граф **graph**.

***Операции:***

*Конструктор*

**Вход:** граф для решения задачи

**Предусловие:** нет

**Процесс:** решение задачи для данного графа

**Выход:** нет

**Постусловия:** построен граф

*Конструктор копирования*

**Вход:** граф для решения задачи

**Предусловие:** нет

**Процесс:** создание копии объекта задачи

**Выход:** нет

**Постусловия:** создана копия объекта задачи

*Получение решения*

**Вход:** список смежных вершин

**Предусловие:** нет

**Процесс:** определение диаметра и пути, соответствующего диаметру

**Выход:** диаметр и путь, соответствующего диаметру данного графа

**Постусловия:** нет

# **Клиентское описание АТД «Задача 2»**

template <class VertexT, class EdgeT>

class Task2{

private:

Graph<VertexT, EdgeT>\* graph; //Обрабатываемый граф

int n; //Число вершин графа

stringstream res; //Результат

public:

Task2(Graph<VertexT, EdgeT> \*g, VertexT \*v); //Конструктор

Task2(const Task1<VertexT, EdgeT> &t1); //Конструктор копирования

~Task2();//Деструктор

void set(Graph<VertexT, EdgeT> \*g, VertexT \*v); //Связывает объект задачи с графом g и выполняет решение задачи 1 для графа g

void restart();//Повторно выполняет решение задачи 1 для графа g

string result();//Возвращает результат решения задачи 1

};

# **Краткое описание алгоритма, теоретическая оценка трудоемкости задачи 1**

Задача 1 была реализованна с помощью алгоритма поиска в глубину. Обход графа – это процедура перебора (посещения) всех вершин графа начиная с заданной вершины. Общее время обхода в глубину оценивается,  как **O(V + E)**.

# **Краткое описание алгоритма, теоретическая оценка трудоемкости задачи 2**

Задача 2 была реализована с помощью алгоритма Флойда. Алгоритм Флойда - динамический алгоритм для нахождения кратчайших расстояний между всеми вершинами взвешенного ориентированного графа. Три вложенных цикла содержат операцию, исполняемую за константное время. Алгоритм Флойда имеет **O(V3)** кубическую сложность. Входом алгоритма Флойда является граф и его весовая функция, представленная в виде матрицы весов - W, результатом - матрица весов кратчайших путей Dn и матрица предшественников Pn.

# **Заключение**

В результате проделанной работы был разработан АТД «Простой граф», а также ассоциированные с ним классы, реализующие шаблонные типы данных. Коллекция повзоляет реализовывать ориентированные и неориентированные, взвешенные и невзвешенные графы M и L типов.

Спроектирован и реализован шаблонный класс для АТД «Задача 1» для решения задачи: …

Спроектирован и реализован шаблонный класс для АТД «Задача 2» для решения задачи на взвешенном графе: определение диаметра и пути, соответствующего диаметру, на основе алгоритма Флойда.

# **Приложение**

Vertex

template <class NameT, class DataT>

class Vertex {

NameT name; //Имя вершины

DataT data; //Данные вершины

public:

bool nameSet, dataSet;

//Конструкторы

Vertex() : nameSet(false), dataSet(false) {}

Vertex(NameT name, DataT data) : name(name), data(data), nameSet(true), dataSet(true) {}

//Установка, получение имени и данных

void setName(NameT name) {

this->name = name;

nameSet = true;

}

void setData(DataT data) {

this->data = data;

dataSet = true;

}

NameT getName() { return name;}

DataT getData() { return data;}

bool isNameSet() { return nameSet;}

bool isDataSet() { return dataSet;}

};

Edge

template <class VertexT, class WeightT, class DataT>

class Edge {

VertexT \*v1, \*v2; //Вершины, которые соединяет ребро

WeightT weight; //Вес

DataT data; //Данные ребра

bool weightSet, dataSet;

public:

//Конструкторы

Edge(VertexT \*v1, VertexT \*v2) : v1(v1), v2(v2), weightSet(false), weight(0), dataSet(false) {}

Edge(VertexT \*v1, VertexT \*v2, WeightT weight) : v1(v1), v2(v2), weight(weight), weightSet(true), dataSet(false) {}

Edge(VertexT \*v1, VertexT \*v2, WeightT weight, DataT data) : v1(v1), v2(v2), weight(weight), data(data), weightSet(true), dataSet(true) {}

//Установка и получение веса и данных

void setWeight(WeightT weight) {

this->weight = weight;

weightSet = true;

}

void setData(DataT data) {

this->data = data;

dataSet = true;

}

WeightT getWeight() { return weight;}

DataT getData() { return data;}

VertexT\* getVertex1(){ return v1;};

VertexT\* getVertex2(){ return v2;};

bool isWeightSet() { return weightSet;}

bool isDataSet() { return dataSet; }

};

GraphForm

template <class EdgeT>

class GraphForm {

public:

//Вставка и удаление вершин и рёбер

virtual bool insertVertex(int index) = 0;

virtual bool deleteVertex(int index) = 0;

virtual bool insertEdge(int v1, int v2, EdgeT \*t) = 0;

virtual bool deleteEdge(int v1, int v2) = 0;

//Удалить входящие и исходящие из вершины рёбра

virtual int deleteEdgesFromVertex(int index, bool directed) = 0;

//Проверка и получение

virtual bool hasEdge(int v1, int v2) = 0;

virtual EdgeT\* getEdge(int v1, int v2) = 0;

};

//Матричная форма

template <class EdgeT>

class GraphMatrixForm : public GraphForm<EdgeT> {

friend class EdgeIterator;

bool directed;

vector<vector<EdgeT\*>> matrix; //Матрица смежности

public:

GraphMatrixForm(bool directed) : directed(directed) {}

//Вставка и удаление вершин и рёбер

bool insertVertex(int index) {

int size = matrix.size(); //Число вершин

//Неверный номер вершины

if(index < 0 || index > size) return false;

//Создаём новую пустую строку

vector<EdgeT\*> newLine;

newLine.assign(size, NULL);

//Вставляем новую строку:

matrix.insert(matrix.begin() + index, newLine);

++size;

//Вставляем новый столбец:

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

matrix[i].insert(matrix[i].begin() + index, (EdgeT\*)NULL);

return true;

}

bool deleteVertex(int index) {

int size = matrix.size(); //Число вершин

//Неверный номер вершины

if(index < 0 || index >= size) return false;

//Удаляем строку:

matrix.erase(matrix.begin() + index);

--size;

//Удаляем столбец:

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

matrix[i].erase(matrix[i].begin() + index);

return true;

}

bool insertEdge(int v1, int v2, EdgeT \*t) {

int size = matrix.size(); //Число вершин

//Неверный номер вершины

if(v1 < 0 || v2 < 0 || v1 >= size || v2 >= size) return false;

//Петля или ребро уже есть

if(v1 == v2 || matrix[v1][v2] != NULL) return false;

//Вставляем ребро

matrix[v1][v2] = t;

return true;

}

bool deleteEdge(int v1, int v2) {

int size = matrix.size(); //Число вершин

//Неверный номер вершины

if(v1 < 0 || v2 < 0 || v1 >= size || v2 >= size) return false;

//Ребра нет

if(v1 == v2 || matrix[v1][v2] == NULL) return false;

matrix[v1][v2] = NULL;

return true;

}

//Удалить входящие и исходящие из вершины рёбра

int deleteEdgesFromVertex(int index, bool directed) {

int size = matrix.size(); //Число вершин

int deleted = 0;

//Неверный номер вершины

if(index < 0 || index >= size) return 0;

//Удаляем связанные с вершиной рёбра

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

if(matrix[i][index] != NULL) {

delete matrix[i][index];

matrix[i][index] = NULL;

++deleted;

//Стираем симметричное ребро

if(directed == false)

matrix[index][i] = NULL;

}

if(matrix[index][i] != NULL) {

delete matrix[index][i];

matrix[index][i] = NULL;

++deleted;

}

}

return deleted;

}

//Проверка и получение

bool hasEdge(int v1, int v2) {

int size = matrix.size(); //Число вершин

//Неверный номер вершины

if(v1 < 0 || v2 < 0 || v1 >= size || v2 >= size)

return false;

if(v1 == v2) //Петля

return false;

if(matrix[v1][v2] != NULL)

return true;

return false;

}

EdgeT\* getEdge(int v1, int v2) {

int size = matrix.size(); //Число вершин

//Неверный номер вершины

if(v1 < 0 || v2 < 0 || v1 >= size || v2 >= size)

throw 1;

if(v1 == v2 || matrix[v1][v2] == NULL)//Петля

throw 1;

return matrix[v1][v2];

}

//Списочная форма

template <class EdgeT>

class GraphListForm : public GraphForm<EdgeT> {

//Элемент списка

class Node {

public:

EdgeT \*edge; //Само ребро

int v2; //Вторая вершина, которую ребро соединяет

};

vector<list<Node>> edgeList; //Списки смежности

bool directed;

public:

GraphListForm(bool directed) : directed(directed) {}

//Вставка и удаление вершин и рёбер

bool insertVertex(int index) {

int size = edgeList.size(); //Число вершин

if(index < 0 || index > size) //Неверный номер вершины

return false;

//Создаём новый список смежности

list<Node> newList;

//Вставляем

edgeList.insert(edgeList.begin() + index, newList);

++size;

//Обновляем дескрипторы

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

for(list<Node>::iterator j = edgeList[i].begin(); j != edgeList[i].end(); ++j)

if((\*j).v2 >= index)//если текущая вершина имеет больший но-мер, чем вставляемая,

++((\*j).v2);//то увеличиваем этот номер

return true;

}

bool deleteVertex(int index) {

int size = edgeList.size(); //Число вершин

if(index < 0 || index >= size) //Неверный номер вершины

return false;

//Удаляем из списков записи о рёбрах

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

for(list<Node>::iterator j = edgeList[i].begin(); j != edgeList[i].end(); ++j)

if((\*j).v2 == index) {

edgeList[i].erase(j);

break;

}

//Удаляем список смежности

edgeList.erase(edgeList.begin() + index);

--size;

//Обновляем дескрипторы

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

for(list<Node>::iterator j = edgeList[i].begin(); j != edgeList[i].end(); ++j)

if((\*j).v2 > index)//если текущая вершина имеет больший номер, чем удаляемая,

--((\*j).v2);//то уменьшить этот номер

return true;

}

bool insertEdge(int v1, int v2, EdgeT \*t) {

int size = edgeList.size(); //Число вершин

if(v1 < 0 || v2 < 0 || v1 >= size || v2 >= size)//Неверный номер вер-шины

return false;

if(v1 == v2 || hasEdge(v1, v2)) //Петля или ребро уже есть

return false;

//Вставляем ребро

Node node;

node.edge = t;

node.v2 = v2;

edgeList[v1].push\_front(node);

return true;

}

bool deleteEdge(int v1, int v2) {

int size = edgeList.size(); //Число вершин

//Неверный номер вершины

if(v1 < 0 || v2 < 0 || v1 >= size || v2 >= size)

return false;

//Ребра нет

if(v1 == v2 || hasEdge(v1, v2) == false)

return false;

//Удаляем ребро

for(list<Node>::iterator j = edgeList[v1].begin(); j != edgeList[v1].end(); ++j)

if((\*j).v2 == v2) {

edgeList[v1].erase(j);

break;

}

return true;

}

//Удалить входящие и исходящие из вершины рёбра

int deleteEdgesFromVertex(int index, bool directed) {

int size = edgeList.size(); //Число вершин

int deleted = 0;

//Неверный номер вершины

if(index < 0 || index >= size)

return 0;

//Удаляем связанные с вершиной рёбра

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

for(list<Node>::iterator j = edgeList[i].begin(); j != edgeList[i].end(); ++j)

if((\*j).v2 == index) {

delete (\*j).edge;

edgeList[i].erase(j);

++deleted;

//Стираем симметричное ребро

if(directed == false)

for(list<Node>::iterator k = edgeList[index].begin(); k != edgeList[index].end(); ++k)

if((\*k).v2 == i) {

edgeList[index].erase(k);

break;

}

break;

}

if (directed==true){

for(list<Node>::iterator z = edgeList[index].begin(); z != edgeList[index].end(); ++z){

delete (\*z).edge;

deleted++;

}

}

return deleted;

}

//Проверка и получение

bool hasEdge(int v1, int v2) {

int size = edgeList.size(); //Число вершин

//Неверный номер вершины

if(v1 < 0 || v2 < 0 || v1 >= size || v2 >= size)

return false;

//Петля

if(v1 == v2)

return false;

for(list<Node>::iterator j = edgeList[v1].begin(); j != edgeList[v1].end(); ++j)

if((\*j).v2 == v2)

return true;

return false;

}

EdgeT\* getEdge(int v1, int v2) {

int size = edgeList.size(); //Число вершин

//Неверный номер вершины

if(v1 < 0 || v2 < 0 || v1 >= size || v2 >= size)

throw 1;

//Петля

if(v1 == v2)

throw 1;

for(list<Node>::iterator j = edgeList[v1].begin(); j != edgeList[v1].end(); ++j)

if((\*j).v2 == v2)

return (\*j).edge;

throw 1;

}

EdgeIterator

//Итератор рёбер

class EdgeIterator {

bool end;

GraphMatrixForm \*f;

public:

int curI, curJ;

EdgeIterator(GraphMatrixForm \*f) {

this->f = f;

begin();

}

bool begin() {

for(int i = 0; i < f->matrix.size(); ++i)

for(int j = 0; j < f->matrix.size(); ++j)

if(f->matrix[i][j]) {

curI = i;

curJ = j;

end = false;

return true;

}

end = true;

return false;

}

bool onEnd() {

return end;

}

bool toend(){

for(int i=f->matrix.size()-1; i>=0; --i)

for(int j=f->matrix.size()-1; j>=0; --j)

if (f->matrix[i][j]){

curI=i;

curJ=j;

end=false;

return true;

}

end=true;

return false;

}

bool next() {

if(end)

return false;

++curJ;

while(curI < f->matrix.size()) {

while(curJ < f->matrix.size()) {

if(f->matrix[curI][curJ])

return true;

++curJ;

}

++curI;

curJ = (f->directed ? 0 : curI + 1);

}

end = true;

return false;

}

EdgeT\* getEdge() {

if(end)

throw 1;

if (!(f->matrix[curI][curJ]))

throw 1;

return f->matrix[curI][curJ];

}

};

OutputEdgeIterator

//Итератор исходящих рёбеp

class OutputEdgeIterator {

int curI, curJ;

bool end;

GraphMatrixForm \*f;

public:

OutputEdgeIterator(GraphMatrixForm \*f, int vIndex) {

this->f = f;

curI = vIndex;

begin();

}

bool begin() {

for(curJ = 0; curJ < f->matrix.size(); ++curJ)

if(f->matrix[curI][curJ]) {

end = false;

return true;

}

end = true;

return false;

}

bool toend(){

for(curJ=f->matrix.size()-1; curJ>=0; --curJ)

if (f->matrix[curI][curJ]){

end=false;

return true;

}

end=true;

return false;

}

bool onEnd() {

return end;

}

bool next() {

if(end)

return false;

++curJ;

while(curJ < f->matrix.size()) {

if(f->matrix[curI][curJ])

return true;

++curJ;

}

end = true;

return false;

}

EdgeT\* getEdge() {

if(end)

throw 1;

return f->matrix[curI][curJ];

}

};

};

Graph

//Граф

template <class VertexT, class EdgeT>

class Graph {

friend class VertexIterator;

vector<VertexT\*> vertexVector;//Вектор вставленных

GraphForm<EdgeT> \*data; //Данные графа

bool directed; //Направленный граф

bool dense; //М-граф

int edgeCounter; //Число рёбер

//Получение индекса вершины по дескриптору

int getIndexFromName(string n){

int index=0;

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

if (vertexVector[index]->getName()==n)

break;

++index;

}

if(index == vertexVector.size())

throw "Данной вершины нет в графе";

return index;

};

public:

//Конструкторы:

//Пустой L граф

Graph() {

data = new GraphListForm<EdgeT>(false);

directed = false;

dense = false;

edgeCounter = 0;

}

//Заданы кол-во вершин, направленность, форма

Graph(int vertexCount, bool directed, bool dense) {

if(dense)

data = new GraphMatrixForm<EdgeT>(directed);

else

data = new GraphListForm<EdgeT>(directed);

//Создаём вершины и помещаем в структуру

for(int i = 0; i < vertexCount; ++i)

data->insertVertex(i);

for(int i = 0; i < vertexCount; ++i) {

VertexT \*v = new VertexT();

v->setName(i\_to\_s(sz));

sz++;

vertexVector.push\_back(v);

data->insertVertex(i);

}

this->directed = directed;

this->dense = dense;

edgeCounter = 0;

}

//+ задано кол-во случайных рёбер

Graph(int vertexCount, int edgeCount, bool directed, bool dense) {

if(dense)

data = new GraphMatrixForm<EdgeT>(directed);

else

data = new GraphListForm<EdgeT>(directed);

//Создаём вершины и помещаем в структуру

for(int i = 0; i < vertexCount; ++i)

data->insertVertex(i);

for(int i = 0; i < vertexCount; ++i) {

VertexT \*v = new VertexT();

v->setName(i\_to\_s(sz));

sz++;

vertexVector.push\_back(v);

data->insertVertex(i);

}

this->directed = directed;

this->dense = dense;

edgeCounter = 0;

//Вставляем рёбра

if(edgeCount <= 0) return;

//Корректируем число рёбер, чтобы не превышало максимумы

if(directed) {

if(edgeCount > vertexCount \* (vertexCount - 1))

edgeCount = vertexCount \* (vertexCount - 1);

srand(time(0));

} else {

if(edgeCount > (vertexCount \* (vertexCount - 1)) / 2)

edgeCount = (vertexCount \* (vertexCount - 1)) / 2;

}

//Вставляем

int v1, v2;

while(edgeCounter < edgeCount) {

v1 = rand() % vertexCount;

v2 = rand() % vertexCount;

if(v1 == v2)

continue;

if(data->hasEdge(v1, v2))

continue;

EdgeT \*t = new EdgeT(vertexVector[v1], vertexVector[v2]);

data->insertEdge(v1, v2, t);

t->setWeight((rand() % 9) +1);

if(directed == false)

data->insertEdge(v2, v1, t);

++edgeCounter;

}

}

Graph(const Graph<VertexT, EdgeT> &G) {

delete data;

if(G.isDense())

data = new GraphMatrixForm<EdgeT>(\*(G.data));

else

data = new GraphListForm<EdgeT>(\*(G.data));

directed = G.directed;

dense = G.dense;

}

~Graph() {

while (vertexVector.size()!=0)

deleteVertex(vertexVector.size()-1);

delete data;

}

//Информация о графе

//Число вершин, рёбер, насыщенность

int getVertexCount() {

return vertexVector.size();

}

int getEdgeCount() {

return edgeCounter;

}

double getRatio() {

int max = vertexVector.size() \* (vertexVector.size() - 1);

if(!directed)

max /= 2;

return (double)edgeCounter / (double)max;

}

//Тип графа

bool isDirected() {

return directed;

}

bool isDense() {

return dense;

}

//Преобразования

void toDenseGraph() {

//Создаём структуру хранения М-графа

GraphForm<EdgeT> \*newData = new GraphMatrixForm<EdgeT>(this->directed);

//Создаём вершины

for(int i = 0; i < vertexVector.size(); ++i)

newData->insertVertex(i);

//Переносим рёбра

for(int i = 0; i < vertexVector.size(); ++i)

for(int j = 0; j < vertexVector.size(); ++j)

if(data->hasEdge(i, j))

newData->insertEdge(i, j, data->getEdge(i, j));

delete data;

data = newData;

dense = true;

}

void toSparseGraph() {

//Создаём структуру хранения L-графа

GraphForm<EdgeT> \*newData = new GraphListForm<EdgeT>(this->directed);

//Создаём вершины

for(int i = 0; i < vertexVector.size(); ++i)

newData->insertVertex(i);

//Переносим рёбра

for(int i = 0; i < vertexVector.size(); ++i)

for(int j = 0; j < vertexVector.size(); ++j)

if(data->hasEdge(i, j))

newData->insertEdge(i, j, data->getEdge(i, j));

delete data;

data = newData;

dense = false;

}

//Данные

//Вершины

VertexT\* insertVertex() {

VertexT \*v = new VertexT;

if(data->insertVertex(vertexVector.size()) == false)

throw 1;

v->setName(i\_to\_s(sz));

sz++;

vertexVector.push\_back(v);

return v;

}

VertexT\* insertVertex(int index) {

VertexT \*v = new VertexT;

if(data->insertVertex(index) == false)

throw 1;

vertexVector.insert(vertexVector.begin() + index, v);

return v;

}

bool deleteVertex(VertexT \*v) {

int index = getIndex(v);

edgeCounter -= data->deleteEdgesFromVertex(index, directed);

if(data->deleteVertex(index)) {

vertexVector.erase(vertexVector.begin() + index);

return true;

}

return false;

}

bool deleteVertex(int index) {

edgeCounter -= data->deleteEdgesFromVertex(index, directed);

if(data->deleteVertex(index)) {

vertexVector.erase(vertexVector.begin() + index);

return true;

}

return false;

}

bool deleteVertex(string n) {

VertexT \*v=new VertexT;

bool rez;

try{

v=getVertexFromName(n);

}

catch(char\*){

return false;

}

rez=deleteVertex(v);

return rez;

}

VertexT\* getVertex(int index) {

if(index < 0 || index >= vertexVector.size())

throw "Данной вершины нет в графе";

return vertexVector[index];

}

VertexT\* getVertexFromName(string n){

int i;

for(i=0; i<vertexVector.size(); i++)

if (vertexVector[i]->getName()==n)

return vertexVector[i];

if (i==vertexVector.size())

throw "Данной вершины нет в графе";

};

int getIndex(VertexT \*v) {

int index = 0;

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

if(vertexVector[index] == v)

break;

++index;

}

if(index == vertexVector.size())

throw "Данной вершины нет в графе";

return index;

}

//Рёбра

void insertEdge(string v1, string v2){

VertexT\* \_v1=new VertexT;

VertexT\* \_v2=new VertexT;

try{

\_v1=getVertexFromName(v1);

}

catch(char\*){

throw "Неверное имя вершины 1";

};

try{

\_v2=getVertexFromName(v2);

}

catch(char\*){

throw "Неверное имя вершины 2";

}

try{

insertEdge(\_v1, \_v2);

}

catch (char\* Error){

throw "Петля недопустима, или ребро уже есть";

}

return;

};

EdgeT\* insertEdge(VertexT \*v1, VertexT \*v2) {

EdgeT \*e = new EdgeT(v1, v2);

if(!data->insertEdge(getIndex(v1), getIndex(v2), e))

throw "Петля недопустима, или ребро уже есть";

if(directed == false)

data->insertEdge(getIndex(v2), getIndex(v1), e);

++edgeCounter;

return e;

}

bool deleteEdge(string v1, string v2){

VertexT\* \_v1=new VertexT;

VertexT\* \_v2=new VertexT;

try{

\_v1=getVertexFromName(v1);

}

catch(char\*){

throw "Неверный номер вершины 1";

};

try{

\_v2=getVertexFromName(v2);

}

catch(char\*){

throw "Неверный номер вершины 2";

}

if(!deleteEdge(\_v1, \_v2))

return false;

return true;

};

bool deleteEdge(VertexT \*v1, VertexT \*v2) {

if(data->deleteEdge(getIndex(v1), getIndex(v2))) {

--edgeCounter;

if(directed == false)

data->deleteEdge(getIndex(v2), getIndex(v1));

return true;

} else

return false;

}

bool hasEdge(VertexT \*v1, VertexT \*v2) {

int ind1, ind2;

try {

ind1 = getIndex(v1);

}

catch(int &a) {

return false;

}

try {

ind2 = getIndex(v2);

}

catch(int &a)

{

return false;

}

return data->hasEdge(getIndex(v1), getIndex(v2));

}

bool hasEdge(int v1, int v2) {

if(v1 < 0 || v1 >= vertexVector.size())

return false;

if(v2 < 0 || v2 >= vertexVector.size())

return false;

return data->hasEdge(v1, v2);

}

EdgeT\* getEdge(VertexT \*v1, VertexT \*v2) {

EdgeT\* e;

try{

e=data->getEdge(getIndex(v1), getIndex(v2));

}

catch(int){

throw " False";

}

return e;

}

bool write\_data\_edge(string \_v1, string \_v2, int data){

EdgeT\* e;

VertexT \*v1, \*v2;

try{

v1=getVertexFromName(\_v1);

}

catch(char\*){

return false;

}

try{

v2=getVertexFromName(\_v2);

}

catch(char\*){

return false;

}

e=getEdge(v1, v2);

e->setData(data);

return true;

};

bool write\_weight\_edge(string \_v1, string \_v2, int w){

EdgeT\* e;

VertexT \*v1, \*v2;

try{

v1=getVertexFromName(\_v1);

}

catch(char\*){

return false;

}

try{

v2=getVertexFromName(\_v2);

}

catch(char\*){

return false;

}

e=getEdge(v1, v2);

e->setWeight(w);

return true;

};

int read\_data\_edge(string \_v1, string \_v2){

EdgeT\* e;

VertexT \*v1, \*v2;

try{

v1=getVertexFromName(\_v1);

}

catch(char\*){

throw " False";

}

try{

v2=getVertexFromName(\_v2);

}

catch(char\*){

throw " False";

}

e=getEdge(v1, v2);

if (e->isDataSet())

return e->getData();

else

throw " Null";

};

int read\_weight\_edge(string \_v1, string \_v2){

EdgeT\* e;

VertexT \*v1, \*v2;

try{

v1=getVertexFromName(\_v1);

}

catch(char\*){

throw " False";

}

try{

v2=getVertexFromName(\_v2);

}

catch(char\*){

throw " False";

}

e=getEdge(v1, v2);

if (e->isWeightSet())

return e->getWeight();

else

throw " Null";

};

void print\_graph(){

int i, j;

VertexT\* v;

EdgeT\* e;

if (isDense()){

cout<<" ";

for (i=0; i<getVertexCount(); i++){

v=getVertex(i);

cout<<setw(4)<<v->getName();

}

cout<<endl;

for(i=0; i<5\*getVertexCount(); i++)

cout<<"\_";

cout<<endl;

for (i=0; i<getVertexCount(); i++){

v=getVertex(i);

cout<<v->getName()<<"|";

for (j=0; j<getVertexCount(); j++)

if (hasEdge(i,j)){

e=getEdge(getVertex(i),getVertex(j));

cout<<setw(4)<<e->getWeight();

}

else

cout<<setw(4)<<"0";

cout<<endl;

}

}

else

{

for (i=0; i<getVertexCount(); i++){

v=getVertex(i);

cout<<"\*"<<v->getName()<<"->";

for (j=0; j<getVertexCount(); j++){

v=getVertex(j);

if (hasEdge(i,j))

cout<<v->getName()<<"->";

}

cout<<endl;

}

}

};

int read\_data\_vertex(string v1){

VertexT \*v;

try

{

v=getVertexFromName(v1);

}

catch(char\*){

throw " False";

};

int d;

d=v->getData();

if (v->dataSet!=false)

return d;

else

throw " Null";

};

bool write\_data\_vertex(string v1, int d){

VertexT \*v;

try

{

v=getVertexFromName(v1);

}

catch(char\*){

return false;

};

v->setData(d);

return true;

};

bool write\_name\_vertex(string v1, string str){

VertexT \*v;

try

{

v=getVertexFromName(v1);

}

catch(char\*){

return false;

};

v->setName(str);

return true;

};

VertexIterator

//Итератор вершин

class VertexIterator {

Graph<VertexT, EdgeT> \*graph; //Обрабатываемый граф

bool end;

int current;

public:

VertexIterator(Graph<VertexT, EdgeT> &g) {

graph = &g;

begin();

}

//Установка в начало

void begin() {

if(graph->vertexVector.size() == 0) {

end = true;

return;

}

current = 0;

end = false;

return;

}

//Сдвиг на следующий элемент

void operator++() {

if(end){

current=-1;

return;

}

current++;

if(graph->vertexVector.size()-1 == current)

end = true;

return;

};

void toend(){

current=graph->vertexVector.size()-1;

end=true;

return;

};

//Проверка на выход

bool onEnd() {

return end;

}

//Получение вершины

VertexT\* operator\*() {

if(current >= graph->vertexVector.size() || current==-1)

throw 1;

return graph->vertexVector[current];

}

string read\_vertex() {

VertexT \*v;

try

{

v=operator\*();

}

catch(int){

throw " False";

};

string d;

d=v->getName();

if (v->nameSet!=false)

return d;

else

throw " Null";

}

int read\_data\_vertex(){

VertexT \*v;

try

{

v=operator\*();

}

catch(int){

throw " False";

};

int d;

d=v->getData();

if (v->dataSet!=false)

return d;

else

throw " Null";

};

void write\_data\_vertex(int d){

VertexT \*v;

try

{

v=operator\*();

}

catch(int){

throw " False";

};

v->setData(d);

return;

};

void write\_name\_vertex(string str){

VertexT \*v;

try

{

v=operator\*();

}

catch(int){

throw " False";

};

v->setName(str);

return;

};

};

EdgeIterator

//Итератор рёбер

class EdgeIterator {

Graph<VertexT, EdgeT> \*graph; //Обрабатываемый граф

typename GraphMatrixForm<EdgeT>::EdgeIterator \*mIt;

typename GraphListForm<EdgeT>::EdgeIterator \*lIt;

bool useM;

public:

int curV1, curV2;

bool end;

EdgeIterator(Graph<VertexT, EdgeT> &g) {

graph = &g;

if(graph->isDense()) {

mIt = new GraphMatrixForm<EdgeT>::EdgeIterator(static\_cast<GraphMatrixForm<EdgeT>\*>(g.data));

useM = true;

} else {

lIt = new GraphListForm<EdgeT>::EdgeIterator(static\_cast<GraphListForm<EdgeT>\*>(g.data));

useM = false;

}

begin();

}

//Установка в начало

bool begin() {

if(useM) return

mIt->begin();

else

return lIt->begin();

}

//Сдвиг на следующий элемент

bool operator++() {

if(useM)

return mIt->next();

else

return lIt->next();

}

//Проверка на выход

bool onEnd() {

//return end;

if(useM)

return mIt->onEnd();

else

return lIt->onEnd();

}

bool toend(){

if(useM)

return mIt->toend();

else

return lIt->toend();

}

//Получение ребра

EdgeT\* operator\*() {

if (onEnd())

throw " Null";

if(useM)

return mIt->getEdge();

else

return lIt->getEdge();

}

string read\_edge(){

EdgeT\* e;

try{

e=operator\*();

}

catch(char\* Error){

throw " False";

}

VertexT \*\_v1, \*\_v2;

int w;

string str1, str2, str3, str4;

\_v1=e->getVertex1();

\_v2=e->getVertex2();

if (\_v1->nameSet==false)

str1="Null";

else

str1=\_v1->getName();

if (\_v2->nameSet==false)

str2="Null";

else

str2=\_v2->getName();

if (e->isWeightSet()==false)

str3="Null";

else

str3=i\_to\_s(e->getWeight());

str4=" Исходящая вершина: "+str1+". Входящая вершина: "+str2+". Вес ребра: "+str3;

return str4;

}

int read\_data\_edge(){

EdgeT\* e;

try{

e=operator\*();

}

catch(char\* Error){

throw " False";

}

int a;

if (e->isDataSet())

return e->getData();

else

throw " Null";

};

bool write\_data\_edge(int d){

EdgeT\* e;

try{

e=operator\*();

}

catch(char\* Error){

return false;

}

e->setData(d);

return true;

};

bool write\_weight\_edge(int w){

EdgeT\* e;

try{

e=operator\*();

}

catch(char\* Error){

return false;

}

e->setWeight(w);

return true;

}

};

OutputEdgeIterator

//Итератор исходящих рёбер

class OutputEdgeIterator {

Graph<VertexT, EdgeT> \*graph; //Обрабатываемый граф

typename GraphMatrixForm<EdgeT>::OutputEdgeIterator \*mIt;

typename GraphListForm<EdgeT>::OutputEdgeIterator \*lIt;

bool useM;

public:

int curV1, curV2;

bool end;

OutputEdgeIterator(Graph<VertexT, EdgeT> &g, VertexT &v) {

graph = &g;

curV1 = graph->getIndex(&v);

if(graph->isDense()) {

mIt = new GraphMatrixForm<EdgeT>::OutputEdgeIterator(static\_cast<GraphMatrixForm<EdgeT>\*>(g.data), curV1);

useM = true;

} else {

lIt = new GraphListForm<EdgeT>::OutputEdgeIterator(static\_cast<GraphListForm<EdgeT>\*>(g.data), curV1);

useM = false;

}

begin();

}

//Установка в начало

bool begin() {

if(useM)

return mIt->begin();

else

return lIt->begin();

}

//Сдвиг на следующий элемент

bool operator++() {

if(useM)

return mIt->next();

else

return lIt->next();

}

bool toend(){

if(useM)

return mIt->toend();

else

return lIt->toend();

}

//Проверка на выход

bool onEnd() {

if(useM)

return mIt->onEnd();

else

return lIt->onEnd();

}

//Получение ребра

EdgeT\* operator\*() {

if (onEnd())

throw " False";

if(useM)

return mIt->getEdge();

else

return lIt->getEdge();

}

string read\_edge(){

EdgeT\* e;

try{

e=operator\*();

}

catch(char\* Error){

throw " False";

}

VertexT \*\_v1, \*\_v2;

int w;

string str1, str2, str3, str4;

\_v1=e->getVertex1();

\_v2=e->getVertex2();

if (\_v1->nameSet==false)

str1="Null";

else

str1=\_v1->getName();

if (\_v2->nameSet==false)

str2="Null";

else

str2=\_v2->getName();

if (e->isWeightSet()==false)

str3="Null";

else

str3=i\_to\_s(e->getWeight());

str4=" Исходящая вершина: "+str1+". Входящая вершина: "+str2+". Вес ребра: "+str3;

return str4;

}

int read\_data\_edge(){

EdgeT\* e;

try{

e=operator\*();

}

catch(char\* Error){

throw " False";

}

int a;

if (e->isDataSet())

return e->getData();

else

throw " Null";

};

bool write\_data\_edge(int d){

EdgeT\* e;

try{

e=operator\*();

}

catch(char\* Error){

return false;

}

e->setData(d);

return true;

};

bool write\_weight\_edge(int w){

EdgeT\* e;

try{

e=operator\*();

}

catch(char\* Error){

return false;

}

e->setWeight(w);

return true;

}

};

};

Task1

class Task1 {

int\*\* Adj; //матрица смежных вершин

int\* c; //ход, на котором посещается вершина

int n; //кол верш в графе

int v0 = 0; //начальная вершина

int cd = 0;

//stack <int> st;

int\* exit; // ответ

Graph<VertexT, EdgeT>\* graph; //Обрабатываемый граф

public:

typename Graph<VertexT, EdgeT>::OutputEdgeIterator\* out;

Task1(Graph<VertexT, EdgeT>& g) {

graph = &g;

}

Task1(const Graph<VertexT, EdgeT>& T) {

graph = T.graph;

}

~Task1() {

delete с, Adj, exit;

}

void set(Graph<VertexT, EdgeT>\* g)

{

graph = g;

restart();

}

void restart(int v0, int n) {

this->n = n;

this->v0 = v0;

cd = graph->getVertexCount();

c = new int [cd];

exit = new int[cd];

Adj = new int\* [cd];

for (int i = 0; i < cd; i++) {

Adj[i] = new int[cd];

c[i] = -1;

}

c[v0] = v0;

exit[0] = v0;

search();

if (!DFS\_Visit(1)) return false;

result(n);

return true;

}

int DFS\_Visit(int k) {

int v, q1 = 0;

for (v = 0; v < cd && !q1; v++)

{

if (Adj[exit[k - 1]][v] > 0)

{

if (k == n && v == v0) q1 = 1; // show(n);

else if (c[v] == -1)

{

c[v] = k; exit[k] = v;

q1 = DFS\_Visit(k + 1);

if (!q1) c[v] = -1;

}

else continue;

}

}

return q1;

}

void search() {

int vv1, vv2;

for (int u = 0; u < cd; u++) {

try {

out = new Graph<VertexT, EdgeT>::OutputEdgeIterator(\*graph, \*graph->getVertex(u));

}

catch (char\*) {

continue;

}

if (!out->begin()) {

continue;

}

for (int j = 0; j < cd; j++) {

Adj[u][j] = 0;

if (graph->hasEdge(u, j)) {

EdgeT\* e = graph->getEdge(graph->getVertex(u), graph->getVertex(j));

Adj[u][j] = 1;

}

}

}

}

void result(int n) {

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

cout << exit[i] << " -> ";

}

cout << v0;

}

};

Task2

//Процедура печати матрицы:

template <class Elem>

void PrintMatrix(const vector<vector<Elem>>& A)

{

for (int i = 0; i < A.size(); i++)

{

for (int j = 0; j < A[i].size(); j++)

{

cout << setw(3) << A[i][j];

}

cout << endl;

}

}

template <class VertexT, class EdgeT>

class Task2 {

private:

Graph<VertexT, EdgeT>\* graph; //Обрабатываемый граф

int n; //Число вершин графа

stringstream res; //Результат

struct TDiam

{

TElem max; //Значение диаметра

int i, j; //Крайние вершины

};

void GenerateWeightMatrix(TMatrix& W)

{

//Заполнение матрицы весов весами рёбер графа:

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

{

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

{

if (graph->hasEdge(i, j))

{

EdgeT\* e = graph->getEdge(graph->getVertex(i), graph->getVertex(j));

if (e->isWeightSet())

W[i][j] = e->getWeight();

else

throw exception("Не установлен вес ребра");

}

else

{

W[i][j] = inf;

}

}

}

}

void Floyd(const TMatrix& W, TMatrix& D, TIntMatrix& P)

{

//Инициализация матрицы весов кратчайших путей:

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

{

D[i][i] = 0;

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

{

if (i != j)

{

D[i][j] = W[i][j];

}

}

}

//Инициализация матрицы предшественников:

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

{

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

{

if (i != j && W[i][j] != inf)

{

P[i][j] = i;

}

else

{

P[i][j] = -1;

}

}

}

//Алгоритм Флойда:

for (int k = 0; k < n; k++)

{

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

{

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

{

if (D[i][j] > D[i][k] + D[k][j])

{

D[i][j] = D[i][k] + D[k][j];

P[i][j] = P[k][j];

}

}

}

}

}

TDiam FindDiam(const TMatrix& D)

{

TDiam diam = { -inf, -1, -1 };

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

{

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

{

if (D[i][j] != inf && D[i][j] > diam.max)

{

diam.max = D[i][j];

diam.i = i;

diam.j = j;

}

}

}

return diam;

}

void PrintPath(const TIntMatrix& P, int s, int f)

{

if (f == s)

{

res << f << " -> ";

}

else

{

PrintPath(P, s, P[s][f]);

res << f << " -> ";

}

}

public:

//Конструкторы-деструкторы

Task2(Graph<VertexT, EdgeT>\* g) : graph(g)

{

restart();

}

Task2(const Task2<VertexT, EdgeT>& t1) : graph(t1.g), n(t1.n) { }

void set(Graph<VertexT, EdgeT>\* g)

{

graph = g;

restart();

}

void restart()

{

res.str("");

n = graph->getVertexCount();

if ((graph->getEdgeCount()) < 1)

throw exception("В графе нет рёбер");

TMatrix W(n, TVec(n)); //Матрица весов

GenerateWeightMatrix(W); //Генерация матрицы весов

TMatrix D(n, TVec(n)); //Матрица весов кратчайших путей

TIntMatrix P(n, TIntVec(n)); //Матрица предшественников

Floyd(W, D, P);

TDiam diam = FindDiam(D); //Поиск диаметра графа

res << "Диаметр графа = " << diam.max << endl;

if (diam.max < inf)

{

res << "Путь, соответствующий диаметру:" << endl;

PrintPath(P, diam.i, diam.j);

res << endl;

}

}

string result()

{

return res.str();

}

};

Main

int enter\_menu(){

int menu;

cin>>menu;

while (!cin.good()){

cout<<"Неверный формат данных!"<<endl;

cout<<"Введите верный пункт меню: ";

cin.clear();

\_flushall();

cin>>menu;

}

return menu;

}

void main()

{

int menu;

Graph<Vertex<string, int>, Edge<Vertex<string, int>, int, int>> \*graph;

Vertex<string, int> \*vertex;

Edge<Vertex<string, int>, int, int> \*edge;

Graph<Vertex<string, int>, Edge<Vertex<string, int>, int, int>>::VertexIterator \*vIt;

Graph<Vertex<string, int>, Edge<Vertex<string, int>, int, int>>::EdgeIterator \*eIt;

Graph<Vertex<string, int>, Edge<Vertex<string, int>, int, int>>::OutputEdgeIterator \*oIt;

Graph<Vertex<string, int>, Edge<Vertex<string, int>, int, int>>::Gam\* Tt1;

setlocale (0, "Rus");

//system("mode 110,50");

cout<<"Выберите пункт меню: "<<endl;

cout<<"1 - Создать граф с V вершинами, без ребер, типа D, формы пред-ставления F"<<endl;

cout<<"2 - Создать пустой L-граф с нулевым числом вершин и рёбер"<<endl;

cout<<"3 - создать граф с V вершинами, с E случайными ребрами, типа D , формы F"<<endl;

cout<<"-1 - Выход"<<endl;

bool flag1=false;

while (!flag1){

cout<<">>";

menu=enter\_menu();

switch (menu){

case 1:{

int v;

bool directed, dense;

string dir, den;

cout<<"Введите количество вершин: ";

cin>>v;

cout<<"Укажите, ориентированный ли граф: ";

cin>>dir;

cout<<"Укажите форму графа (0 - матрица, 1 - список): ";

cin>>den;

if (dir=="true")

directed=true;

else

directed=false;

if (den=="true")

dense=true;

else

dense=false;

graph=new Graph<Vertex<string, int>, Edge<Vertex<string, int>, int, int>>(v, directed, dense);

cout<<"Граф создан"<<endl;

flag1=true;

};

break;

case 2:{

graph=new Graph<Vertex<string, int>, Edge<Vertex<string, int>, int, int>>();

cout<<"Граф создан"<<endl;

flag1=true;

};

break;

case 3:{

int v, e;

bool directed, dense;

string dir, den;

cout<<"Введите количество вершин: ";

cin>>v;

cout<<"Введите количество рёбер: ";

cin>>e;

cout<<"Укажите, ориентированный ли граф: ";

cin>>dir;

cout<<"Укажите форму графа (1 - матрица, 0 - список): ";

cin>>den;

if (dir=="true" || dir=="True")

directed=true;

else

directed=false;

if (den=="true" || dir=="True")

dense=true;

else

dense=false;

graph=new Graph<Vertex<string, int>, Edge<Vertex<string, int>, int, int>>(v, e, directed, dense);

cout<<"Граф создан"<<endl;

flag1=true;

};

break;

case -1: {

return;

};

break;

default: {

cout<<"Неверно выбран пункт меню!"<<endl;

cout<<"Введите верный пункт: ";

};

}

}

cout<<"Выберите пункт меню: "<<endl;

cout<<"0 - Распечатать граф"<<endl;

cout<<"1 - Опрос числа вершин в графе"<<endl;

cout<<"2 - Опрос числа рёбер в графе"<<endl;

cout<<"3 - Опрос типа графа"<<endl;

cout<<"4 - Опрос формы представления графа"<<endl;

cout<<"5 - Опрос коэффициента насыщенности графа"<<endl;

cout<<"6 - Преобразовать граф к L-форме"<<endl;

cout<<"7 - Преобразовать граф к M-форме"<<endl;

cout<<"8 - Добавить вершину к графу"<<endl;

cout<<"9 - Удалить вершину из графа"<<endl;

cout<<"10 - Вставить ребро в граф"<<endl;

cout<<"11 - Удалить ребро из графа"<<endl;

cout<<"12 - Операции над вершиной"<<endl;

cout<<"13 - Итератор вершин"<<endl;

cout<<"14 - Итератор рёбер"<<endl;

cout<<"15 - Итератор исходящих рёбер"<<endl;

cout<<"16 - Операции над ребром"<<endl;

cout<<"20 - Поиск простого цикла из заданной вершины"<<endl;

cout<<"21 - Определение диаметра и пути, соотв. диаметру (алгоритм Флойда)"<<endl;

cout<<"-1 - Выход"<<endl;

while (2>1){

cout<<">>";

menu=enter\_menu();

switch (menu){

case 0:{

cout<<"Печать графа: "<<endl;

graph->print\_graph();

cout<<"Введите пункт меню: ";

};

break;

case 1:{

cout<<"Число вершин в графе: "<<graph->getVertexCount()<<endl;

cout<<"Введите пункт меню: ";

};

break;

case 2:{

cout<<"Число рёбер в графе: "<<graph->getEdgeCount()<<endl;

cout<<"Введите пункт меню: ";

};

break;

case 3:{

cout<<"Тип графа: ";

if (graph->isDirected()==true)

cout<<"ориентированный"<<endl;

else

cout<<"неориентированный"<<endl;

cout<<"Введите пункт меню: ";

};

break;

case 4:{

cout<<"Форма предстваления графа: ";

if (graph->isDense()==true)

cout<<"M-граф"<<endl;

else

cout<<"L-граф"<<endl;

cout<<"Введите пункт меню: ";

};

break;

case 5:{

cout<<"Коэффициент насыщенности: "<<graph->getRatio()<<endl;

};

break;

case 6:{

graph->toSparseGraph();

cout<<"Граф преобразован"<<endl;

cout<<"Введите пункт меню: ";

};

break;

case 7:{

graph->toDenseGraph();

cout<<"Граф преобразован"<<endl;

cout<<"Введите пункт меню: ";

};

break;

case 8:{

try{

graph->insertVertex();

}

catch (int Error){

cout<<Error;

};

cout<<"Вершина добавлена в граф"<<endl;

cout<<"Введите пункт меню: ";

};

break;

case 9:{

string v1;

cout<<"Введите удаляемую вершину: ";

cin>>v1;

if (graph->deleteVertex(v1)==true)

cout<<"Вершина удалена"<<endl;

else

cout<<"Данной вершины нет в графе"<<endl;

cout<<"Введите пункт меню: ";

};

break;

case 10:{

string v1, v2;

cout<<"Введите вершину, из которой выходит ребро: ";

cin>>v1;

cout<<"Введите вершину, в которую входит ребро: ";

cin>>v2;

try{

graph->insertEdge(v1, v2);

}

catch(char\* Error){

cout<<Error<<endl;

cout<<"Введите пункт меню: ";

break;

};

cout<<"Ребро добавлено в граф"<<endl;

cout<<"Введите пункт меню: ";

};

break;

case 11:{

string v1, v2;

bool rez;

cout<<"Введите вершину, из которой выходит ребро: ";

cin>>v1;

cout<<"Введите вершину, в которую входит ребро: ";

cin>>v2;

try{

rez=graph->deleteEdge(v1, v2);

}

catch(char\* Error){

cout<<Error<<endl;

cout<<"Введите пункт меню: ";

break;

};

if (rez)

cout<<"Ребро удалено"<<endl;

else

cout<<"Ребра, проходящего через заданные вершины, не существует"<<endl;

cout<<"Введите пункт меню: ";

};

break;

case 12:{

bool flag=false;

cout<<"Операции над вершиной:"<<endl;

cout<<" 1 - Прочитать данные о вершине"<<endl;

cout<<" 2 - Записать данные о вершине"<<endl;

cout<<" 3 - Записать новое имя вершины"<<endl;

cout<<" -1 -Выход из меню"<<endl;

cout<<" >>";

while (!flag){

menu=enter\_menu();

switch (menu){

case 1:{

int cur;

string v;

cout<<" Введите имя вершины: ";

cin>>v;

try{

cur=graph->read\_data\_vertex(v);

}

catch (char\* Error){

cout<<Error<<endl;

cout<<" Введите пункт меню: ";

break;

}

cout<<" Данные вершины: "<<cur<<endl;

cout<<" Введите пункт меню: ";

};

break;

case 2:{

int cur;

string v;

bool rez;

cout<<" Введите имя вершины: ";

cin>>v;

cout<<" Введите данные: ";

cin>>cur;

rez=graph->write\_data\_vertex(v, cur);

if (!rez){

cout<<" False"<<endl;

cout<<" Введите пункт меню: ";

break;

}

cout<<" Данные о вершине записаны"<<endl;

cout<<" Введите пункт меню: ";

};

break;

case 3:{

bool rez;

string v, str;

cout<<" Введите старое имя вершины: ";

cin>>v;

cout<<" Введите новое имя: ";

cin>>str;

rez=graph->write\_name\_vertex(v, str);

if(!rez){

cout<<" False"<<endl;

cout<<" Введите пункт меню: ";

break;

}

cout<<" Имя вершины перезаписано"<<endl;

cout<<" Введите пункт меню: ";

};

break;

case -1:{

flag=true;

};

break;

default: {

cout<<" Введите верный пункт: ";

};

break;

};

};

}

break;

case 13:{

bool flag=false;

vIt=new Graph<Vertex<string, int>, Edge<Vertex<string, int>, int, int>>::VertexIterator(\*graph);

cout<<"Итератор вершин:"<<endl;

cout<<" 1 - Установить итератор вершин на начальную вершину"<<endl;

cout<<" 2 - Установить итератор вершин на конечную вершину"<<endl;

cout<<" 3 - Переход к следующей позиции"<<endl;

cout<<" 4 - Прочитать вершину по текущей позиции итератора"<<endl;

cout<<" 5 - Прочитать данные вершины по текущей позиции итератора"<<endl;

cout<<" 6 - Записать данные о вершины по текущей позиции итератора"<<endl;

cout<<" 7 - Записать имя вершины по текущей позиции итератора"<<endl;

cout<<" -1 -Выход из меню итератора"<<endl;

cout<<" >>";

while (!flag){

menu=enter\_menu();

switch (menu){

case 1:{

vIt->begin();

cout<<" Итератор вершин графа установлен на первую вершину"<<endl;

cout<<" Введите пункт меню: ";

};

break;

case 2:{

vIt->toend();

cout<<" Итератор вершин графа установлен на последнюю вершину"<<endl;

cout<<" Введите пункт меню: ";

};

break;

case 3:{

++\*vIt;

cout<<" Итератор вершин графа установлен на следующую позицию"<<endl;

cout<<" Введите пункт меню: ";

};

break;

case 4:{

string cur;

try{

cur=vIt->read\_vertex();

}

catch (char\* Error){

cout<<Error<<endl;

cout<<" Введите пункт меню: ";

break;

}

cout<<" Вершина на текущей позиции итератора: "<<cur<<endl;

cout<<" Введите пункт меню: ";

};

break;

case 5:{

int cur;

try{

cur=vIt->read\_data\_vertex();

}

catch (char\* Error){

cout<<Error<<endl;

cout<<" Введите пункт меню: ";

break;

}

cout<<" Данные вершины на текущей позиции итератора: "<<cur<<endl;

cout<<" Введите пункт меню: ";

};

break;

case 6:{

int cur;

cout<<" Введите данные: ";

cin>>cur;

try{

vIt->write\_data\_vertex(cur);

}

catch (char\* Error){

cout<<Error<<endl;

cout<<" Введите пункт меню: ";

break;

}

cout<<" Данные вершины записаны по текущей позиции итератора"<<endl;

cout<<" Введите пункт меню: ";

};

break;

case 7:{

string cur;

cout<<" Введите имя: ";

cin>>cur;

try{

vIt->write\_name\_vertex(cur);

}

catch (char\* Error){

cout<<Error<<endl;

cout<<" Введите пункт меню: ";

break;

}

cout<<" Имя вершины записано по текущей позиции итератора"<<endl;

cout<<" Введите пункт меню: ";

};

break;

case -1:{

flag=true;

delete vIt;

};

break;

default: {

cout<<" Неверно выбран пункт меню!"<<endl;

cout<<" Введите верный пункт: ";

};

break;

}

}

};

break;

case 14:{

bool flag=false;

eIt = new Graph<Vertex<string, int>, Edge<Vertex<string, int>, int, int>>::EdgeIterator(\*graph);

cout<<"Итератор рёбер: "<<endl;

cout<<" 1 - Установить итератор рёбер на начальное ребро"<<endl;

cout<<" 2 - Установить итератор рёбер на конечное ребро"<<endl;

cout<<" 3 - Переход к следующей позиции"<<endl;

cout<<" 4 - Прочитать исходящую, входящую вершины ребра и вес"<<endl<<" по текущей позиции итератора"<<endl;

cout<<" 5 - Прочитать данные ребра по текущей позиции итератора"<<endl;

cout<<" 6 - Записать данные о ребре по текущей позиции итератора"<<endl;

cout<<" 7 - Записать вес ребра по текущей позиции итератора"<<endl;

cout<<" -1 -Выход из меню итератора"<<endl;

cout<<" >>";

while (!flag){

menu=enter\_menu();

switch (menu){

case 1:{

bool rez;

rez=eIt->begin();

if (rez)

cout<<" Итератор рёбер графа установлен на первое ребро"<<endl;

else

cout<<" False. Рёбер в графе нет"<<endl;

cout<<" Введите пункт меню: ";

};

break;

case 2:{

bool rez;

rez=eIt->toend();

if(rez)

cout<<" Итератор рёбер графа установлен на последнее ребро"<<endl;

else

cout<<" False. Рёбер в графе нет"<<endl;

cout<<" Введите пункт меню: ";

};

break;

case 3:{

bool rez;

rez=++\*eIt;

if(rez)

cout<<" Итератор рёбер графа установлен на следующую позицию"<<endl;

else

cout<<" Null"<<endl;

cout<<" Введите пункт меню: ";

};

break;

case 4:{

string cur;

try{

cur=eIt->read\_edge();

}

catch(char\* Error){

cout<<" Null"<<endl;

cout<<" Введите пункт меню: ";

break;

};

cout<<cur<<endl;

cout<<" Введите пункт меню: ";

};

break;

case 5:{

int cur;

try{

cur=eIt->read\_data\_edge();

}

catch (char\* Error){

cout<<Error<<endl;

cout<<" Введите пункт меню: ";

break;

}

cout<<" Данные ребра на текущей позиции итератора: "<<cur<<endl;

cout<<" Введите пункт меню: ";

};

break;

case 6:{

int cur;

bool rez;

cout<<" Введите данные: ";

cin>>cur;

rez=eIt->write\_data\_edge(cur);

if (!rez){

cout<<" False"<<endl;

cout<<" Введите пункт меню: ";

break;

}

cout<<" Данные ребра записаны по текущей позиции итератора"<<endl;

cout<<" Введите пункт меню: ";

};

break;

case 7:{

int w;

bool rez;

cout<<" Введите вес: ";

cin>>w;

rez=eIt->write\_weight\_edge(w);

if(!rez){

cout<<" False"<<endl;

cout<<" Введите пункт меню: ";

break;

}

cout<<" Вес ребра записан по текущей позиции итератора"<<endl;

cout<<" Введите пункт меню: ";

};

break;

case -1:{

flag=true;

delete eIt;

};

break;

default: {

cout<<" Неверно выбран пункт меню!"<<endl;

cout<<" Введите верный пункт: ";

};

break;

}

}

};

break;

case 15:{

bool flag=false;

int v;

cout<<"Введите вершину: ";

cin>>v;

oIt= new Graph<Vertex<string, int>, Edge<Vertex<string, int>, int, int>>::OutputEdgeIterator(\*graph, \*graph->getVertex(v));

cout<<"Итератор исходящих рёбер: "<<endl;

cout<<" 1 - Установить итератор исходящих рёбер на начальное ребро"<<endl;

cout<<" 2 - Установить итератор исходящих рёбер на конечное ребро"<<endl;

cout<<" 3 - Переход к следующей позиции"<<endl;

cout<<" 4 - Прочитать исходящую, входящую вершины ребра и вес"<<endl<<" по текущей позиции итератора"<<endl;

cout<<" 5 - Прочитать данные ребра по текущей позиции итератора"<<endl;

cout<<" 6 - Записать данные о ребре по текущей позиции итератора"<<endl;

cout<<" 7 - Записать вес ребра по текущей позиции итератора"<<endl;

cout<<" -1 -Выход из меню итератора"<<endl;

cout<<" >>";

while (!flag){

menu=enter\_menu();

switch (menu){

case 1:{

bool rez;

rez=oIt->begin();

if (rez)

cout<<" Итератор исходящих рёбер графа установлен на первое ребро"<<endl;

else

cout<<" False. Рёбер в графе нет"<<endl;

cout<<" Введите пункт меню: ";

};

break;

case 2:{

bool rez;

rez=oIt->toend();

if(rez)

cout<<" Итератор исходящих рёбер графа установлен на последнее ребро"<<endl;

else

cout<<" False. Рёбер в графе нет"<<endl;

cout<<" Введите пункт меню: ";

};

break;

case 3:{

bool rez;

rez=++\*oIt;

if(rez)

cout<<" Итератор исходящих рёбер графа установлен на следующую позицию"<<endl;

else

cout<<" Null"<<endl;

cout<<" Введите пункт меню: ";

};

break;

case 4:{

string cur;

try{

cur=oIt->read\_edge();

}

catch(char\* Error){

cout<<" Null"<<endl;

cout<<" Введите пункт меню: ";

break;

};

cout<<cur<<endl;

cout<<" Введите пункт меню: ";

};

break;

case 5:{

int cur;

try{

cur=oIt->read\_data\_edge();

}

catch (char\* Error){

cout<<Error<<endl;

cout<<" Введите пункт меню: ";

break;

}

cout<<" Данные ребра на текущей позиции итератора: "<<cur<<endl;

cout<<" Введите пункт меню: ";

};

break;

case 6:{

int cur;

bool rez;

cout<<" Введите данные: ";

cin>>cur;

rez=oIt->write\_data\_edge(cur);

if (!rez){

cout<<" False"<<endl;

cout<<" Введите пункт меню: ";

break;

}

cout<<" Данные ребра записаны по текущей позиции итератора"<<endl;

cout<<" Введите пункт меню: ";

};

break;

case 7:{

int w;

bool rez;

cout<<" Введите вес: ";

cin>>w;

rez=oIt->write\_weight\_edge(w);

if(!rez){

cout<<" False"<<endl;

cout<<" Введите пункт меню: ";

break;

}

cout<<" Вес ребра записан по текущей позиции итератора"<<endl;

cout<<" Введите пункт меню: ";

};

break;

case -1:{

delete oIt;

flag=true;

};

break;

default: {

cout<<" Неверно выбран пункт меню!"<<endl;

cout<<" Введите верный пункт: ";

};

break;

}

}

};

break;

case 16:{

bool flag=false;

cout<<"Операции над ребром:"<<endl;

cout<<" 1 - Прочитать вес ребра"<<endl;

cout<<" 2 - Прочитать данные о ребре"<<endl;

cout<<" 3 - Записать данные о ребре"<<endl;

cout<<" 4 - Записать вес ребра"<<endl;

cout<<" -1 -Выход из меню"<<endl;

cout<<" >>";

while (!flag){

menu=enter\_menu();

switch (menu){

case 1:{

string v1, v2;

int w;

cout<<" Введите имя исходящей вершины: ";

cin>>v1;

cout<<" Введите имя входящей вершины: ";

cin>>v2;

try{

w=graph->read\_weight\_edge(v1, v2);

}

catch(char\* Error){

cout<<Error<<endl;

cout<<" Введите пункт меню: ";

break;

};

cout<<" Вес ребра: "<<w<<endl;

cout<<" Введите пункт меню: ";

};

break;

case 2:{

int d;

string v1, v2;

cout<<" Введите имя исходящей вершины: ";

cin>>v1;

cout<<" Введите имя входящей вершины: ";

cin>>v2;

try{

d=graph->read\_data\_edge(v1, v2);

}

catch (char\* Error){

cout<<Error<<endl;

cout<<" Введите пункт меню: ";

break;

}

cout<<" Данные ребра: "<<d<<endl;

cout<<" Введите пункт меню: ";

};

break;

case 3:{

int d;

string v1, v2;

bool rez;

cout<<" Введите имя исходящей вершины: ";

cin>>v1;

cout<<" Введите имя входящей вершины: ";

cin>>v2;

cout<<" Введите данные: ";

cin>>d;

rez=graph->write\_data\_edge(v1, v2, d);

if (!rez){

cout<<" False"<<endl;

cout<<" Введите пункт меню: ";

break;

}

cout<<" Данные о вершине записаны"<<endl;

cout<<" Введите пункт меню: ";

};

break;

case 4:{

bool rez;

string v1, v2;

int w;

cout<<" Введите имя исходящей вершины: ";

cin>>v1;

cout<<" Введите имя входящей вершины: ";

cin>>v2;

cout<<" Введите вес: ";

cin>>w;

rez=graph->write\_weight\_edge(v1, v2, w);

if(!rez){

cout<<" False"<<endl;

cout<<" Введите пункт меню: ";

break;

}

cout<<" Вес ребра перезаписан"<<endl;

cout<<" Введите пункт меню: ";

};

break;

case -1:{ flag=true;};

break;

default: {

cout<<" Неверно выбран пункт меню!"<<endl;

cout<<" Введите верный пункт: ";

};

break;

};

};

}

break;

case 20:{

Tt1 = new Graph<Vertex<string, int>, Edge<Vertex<string, int>, int, int>>::Gam(\*graph);

bool result = false;

cout << "Введите вершну: " << endl;

int v;

cin >> v;

cout << "Введите длину цикла: " << endl;

int cic;

cin >> cic;

result = Tt1->DFS(v, cic);

if (result) {

cout << endl << "Цикл: " << endl;

}

cout<<"Задача решена"<<endl;

cout<<"Введите пункт меню: ";

};

break;

case 21:{

Task2<Vertex<string, int>, Edge<Vertex<string, int>, int, int>>\* taskTwo;

taskTwo = new Task2<Vertex<string, int>, Edge<Vertex<string, int>, int, int>>(graph);

cout<<"Результат:" << endl << taskTwo->result()<<endl;

cout<<"Задача решена"<<endl;

cout<<"Введите пункт меню: ";

};

break;

case -1: {

return;

}; break;

default: {

cout<<"Неверно выбран пункт меню!"<<endl;

cout<<"Введите верный пункт: ";

};

break;

}

}

getch();

return;

}