**СОДЕРЖАНИЕ**

[ВВЕДЕНИЕ 3](#_Toc56507684)

[1 ТЕОРЕТИЧЕСКАЯ ЧАСТЬ 4](#_Toc56507685)

[1.1 Постановка задачи 4](#_Toc56507686)

[1.2 Описание объектно-ориентированного подхода 4](#_Toc56507687)

[1.3 Описание предметной области 7](#_Toc56507688)

[2 ПРАКТИЧЕСКАЯ ЧАСТЬ 18](#_Toc56507689)

[2.1 Описание объектов системы, их свойств и методов 18](#_Toc56507690)

[2.2 Результаты тестирования программного продукта 23](#_Toc56507691)

[ЗАКЛЮЧЕНИЕ 28](#_Toc56507692)

[СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ 29](#_Toc56507693)

[ПРИЛОЖЕНИЕ 30](#_Toc56507694)

ВВЕДЕНИЕ

Среди дисциплин и методов дискретной математики теория графов и особенно алгоритмы на графах находят наиболее широкое применение в программировании. Между понятием графа и понятием отношения, имеется глубокая связь — в сущности это равнообъемные понятия. Возникает естественный вопрос, почему же тогда графам оказывается столь явное предпочтение? Дело в том, что теория графов предоставляет очень удобный язык для описания программных (да и многих других) моделей.

Этот тезис можно пояснить следующей аналогией. Понятие отношения также можно полностью выразить через понятие множества. Однако независимое определение понятия отношения удобнее — введение специальных терминов и обозначений упрощает изложение теории и делает ее более понятной. То же относится и к теории графов.

Стройная система специальных терминов и обозначений теории графов позволяют просто и доступно описывать сложные и тонкие вещи. Особенно важно наличие наглядной графической интерпретации понятия графа.

Целью данного курсового проекта является разработка иерархии классов реализующих ориентированный и смешанный графы и операции над ними

Программный продукт должен:

* давать возможность пользователю создавать графы программно
* позволять выводить матрицу смежности созданного графа
* реализовать метод объединения заданных графов
* реализовать метод пересечения заданных графов

Объектом исследования будет Microsoft Visual Studio 2019.

Предметом изучения являются ориентированный и смешанный графы. В качестве языка разработки был выбран C++.

1 ТЕОРЕТИЧЕСКАЯ ЧАСТЬ

* 1. Постановка задачи

Необходимо разработать иерархию классов, реализующих различные типы графов и операции над ними.

Типы графов:

1. ориентированный граф
2. смешанный граф

Операции над графами:

1. пересечение графов
2. объединение графов

В качестве исходных данных к программе выступают:

* индивидуальное задание
* справочная информация по языке программирования С++
* задания и методические указания по выполнению курсового проекта.

В качестве выходной информации будут выступать:

* диаграмма разработанной иерархии классов
* код разработанных классов
* результаты выполнения методов графа
  1. Описание объектно-ориентированного подхода

Что же понимается под объектом и каковы другие основополагающие понятия данного подхода?

Прежде всего, введем понятие класса. Класс - это абстракция множества сущностей реального мира, объединенных общностью структуры и поведения.

Объект - это элемент класса, то есть абстракция определенной сущности.

Подчеркнем, что объекты активны, у них есть не только внутренняя структура, но и поведение, которое описывается так называемыми методами объекта. Например, может быть определен класс "пользователь", характеризующий "пользователя вообще", то есть ассоциированные с пользователями данные и их поведение ( методы ). После этого может быть создан объект "пользователь Иванов" с соответствующей конкретизацией данных и, возможно, методов.

Следующую группу важнейших понятий объектного подхода составляют инкапсуляция, наследование и полиморфизм.

Основным инструментом борьбы со сложностью в объектно-ориентированном подходе является инкапсуляция - сокрытие реализации объектов (их внутренней структуры и деталей реализации методов ) с предоставлением во вне только строго определенных интерфейсов.

Понятие " полиморфизм " может трактоваться как способность объекта принадлежать более чем одному классу. Введение этого понятия отражает необходимость смотреть на объекты под разными углами зрения, выделять при построении абстракций разные аспекты сущностей моделируемой предметной области, не нарушая при этом целостности объекта. (Строго говоря, существуют и другие виды полиморфизма, такие как перегрузка и параметрический полиморфизм, но нас они сейчас не интересуют.)

Наследование означает построение новых классов на основе существующих с возможностью добавления или переопределения данных и методов. Наследование является важным инструментом борьбы с размножением сущностей без необходимости. Общая информация не дублируется, указывается только то, что меняется. При этом класс -потомок помнит о своих "корнях".

Очень важно и то, что наследование и полиморфизм в совокупности наделяют объектно-ориентированную систему способностью к относительно безболезненной эволюции. Средства информационной безопасности приходится постоянно модифицировать и обновлять, и если нельзя сделать так, чтобы это было экономически выгодно, ИБ из инструмента защиты превращается в обузу.

Мы еще вернемся к механизму наследования при рассмотрении ролевого управления доступом. Пополним рассмотренный выше классический набор понятий объектно-ориентированного подхода еще двумя понятиями: грани объекта и уровня детализации.

Объекты реального мира обладают, как правило, несколькими относительно независимыми характеристиками. Применительно к объектной модели будем называть такие характеристики гранями . Мы уже сталкивались с тремя основными гранями ИБ - доступностью, целостностью и конфиденциальностью. Понятие грани позволяет более естественно, чем полиморфизм, смотреть на объекты с разных точек зрения и строить разноплановые абстракции.

Понятие уровня детализации важно не только для визуализации объектов, но и для систематического рассмотрения сложных систем, представленных в иерархическом виде. Само по себе оно очень простое: если очередной уровень иерархии рассматривается с уровнем детализации n > 0, то следующий - с уровнем (n - 1). Объект с уровнем детализации 0 считается атомарным.

Понятие уровня детализации показа позволяет рассматривать иерархии с потенциально бесконечной высотой, варьировать детализацию как объектов в целом, так и их граней.

Весьма распространенной конкретизацией объектно-ориентированного подхода являются компонентные объектные среды, к числу которых принадлежит, например, JavaBeans. Здесь появляется два новых важных понятия: компонент и контейнер.

Неформально компонент можно определить как многократно используемый объект, допускающий обработку в графическом инструментальном окружении и сохранение в долговременной памяти.

Контейнеры могут включать в себя множество компонентов, образуя общий контекст взаимодействия с другими компонентами и с окружением. Контейнеры могут выступать в роли компонентов других контейнеров .

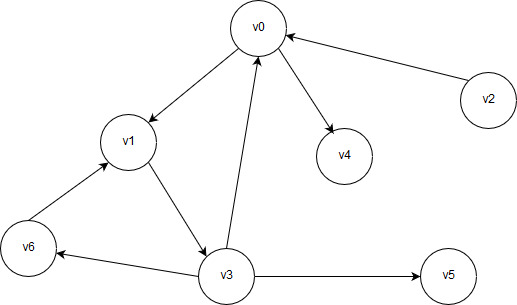
Компонентные объектные среды обладают всеми достоинствами, присущими объектно-ориентированному подходу:

* инкапсуляция объектных компонентов скрывает сложность реализации, делая видимым только предоставляемый вовне интерфейс;
* наследование позволяет развивать созданные ранее компоненты, не нарушая целостность объектной оболочки;
* полиморфизм по сути дает возможность группировать объекты, характеристики которых с некоторой точки зрения можно считать сходными.

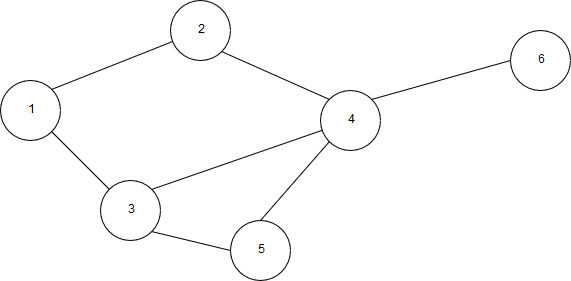
Понятия же компонента и контейнера необходимы нам потому, что с их помощью мы можем естественным образом представить защищаемую ИС и сами защитные средства. В частности, контейнер может определять границы контролируемой зоны (задавать так называемый "периметр безопасности").

1.3 Описание предметной области

Графы делятся на ***ориентированные*** *и* ***неориентированные****.* Ориентированный граф — такой граф, в котором можно двигаться от вершины к вершине только в одном направлении. Например, на рисунке 1.1 можно двигаться из вершины v0 в v1, а из v1 в v0 нельзя. В неориентированных графах можно двигаться в обе стороны. Например, на рисунке 1.2 из вершины 1 можно двигаться в вершину 2, и наоборот.

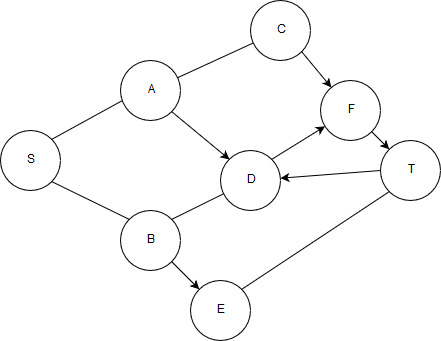
[](http://info.rully.ru/materials/graphs/img/1.png)

**Рисунок 1.1 — Ориентированный граф**

[](http://info.rully.ru/materials/graphs/img/2.png)

**Рисунок 1.2 — Неориентированный граф**

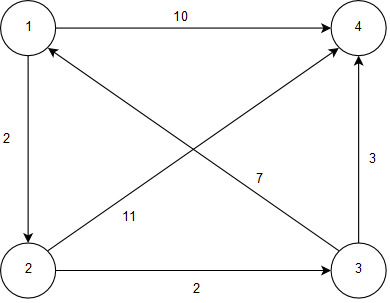
Граф, в котором присутствуют как ориентированные ребра, так и неориентированные, называется ***смешанным***.

[](http://info.rully.ru/materials/graphs/img/3.png)

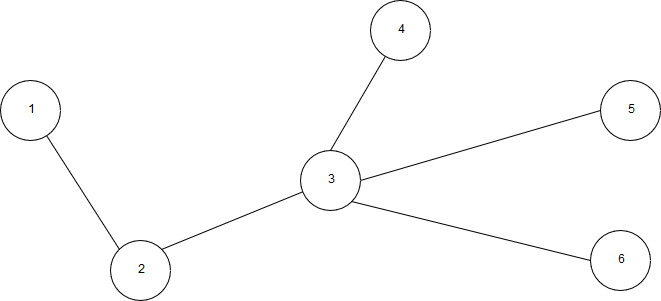
**Рисунок 1.3 — Смешанный граф**

Кроме того, графы делятся на ***взвешенные*** и ***невзвешенные***. Граф, в котором каждому ребру в соответствие поставлено некоторое числовое значение — *вес*, называется взвешенным графом (рисунок 1.4). Если никакого числового значения рёбрам не поставлено, то граф называется невзвешенным. Чаще всего, в названии графа указывают как его ориентированность или неориентированность, так и его взвешенность или невзвешенность.

Граф, в котором между любой парой вершин существует, как минимум, один путь, называется ***связным*** (рисунок 1.4). Если в графе существует хотя бы одна вершина, не связанная с другими, он называется ***несвязным*** (рисунок 1.5).

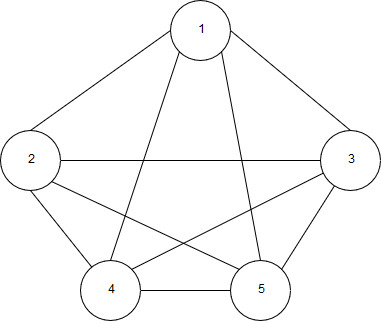
[](http://info.rully.ru/materials/graphs/img/4.png)

**Рисунок 1.4 — Взвешенный связанный ориентированный граф**

[](http://info.rully.ru/materials/graphs/img/5.png)

**Рисунок 1.5 — Несвязный граф**

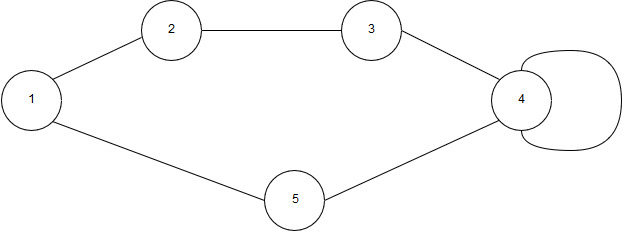
Граф, в котором число рёбер близко к максимальному (когда каждая вершина графа связана с любой другой вершиной графа рёбрами), называется ***плотным графом***. Пример представлен на рисунке 6.

[](http://info.rully.ru/materials/graphs/img/6.png)

**Рисунок 1.6 — Плотный граф**

Граф с противоположным свойством, имеющий малое число рёбер, называется ***разреженным* *графом*.**

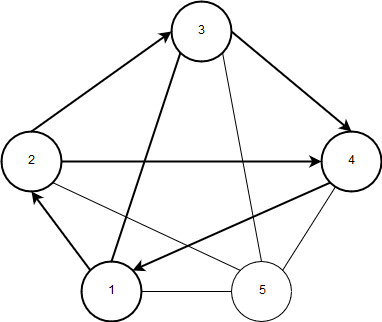
***Петлёй*** называется ребро, которое соединяет вершины v1 и v2, причём v1 и v2 совпадают. Иными словами, петля — это ребро, которое начинается и заканчивается в одной вершине. Пример представлен на рисунке 7.

[](http://info.rully.ru/materials/graphs/img/7.png)  
**Рисунок 1.7 — Граф с петлёй**

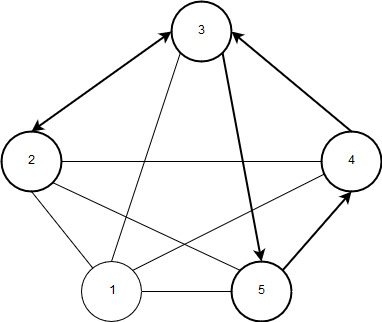
**Инцидентность** — понятие, используемое только в отношении ребра и вершины. Если v1, v2 - вершины, а e = (v1, v2) - соединяющее их ребро, тогда вершина v1 и ребро e инцидентны, вершина v2 и ребро e тоже инцидентны. Две вершины (или два ребра) инцидентными быть не могут.

***Смежность*** — понятие, используемое в отношении только двух рёбер, либо только двух вершин: два ребра, инцидентные одной вершине, называются смежными; две вершины, инцидентные одному ребру, также называются смежными.

***Маршрут*** — это проход по графу через заданную последовательность вершин. Если начальная и конечная вершины последовательности совпадают, то маршрут *замкнут* (рисунок 1.9), иначе маршрут *открыт* (рисунок 1.8).

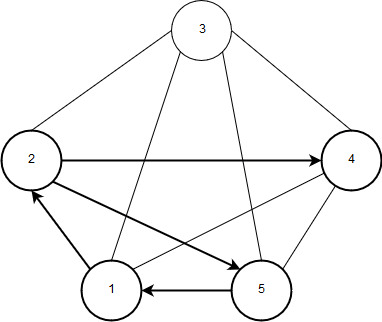
[](http://info.rully.ru/materials/graphs/img/8.png)

**Рисунок 1.8 — Открытый маршрут: вершины 2-4-1-2-3-4-1**

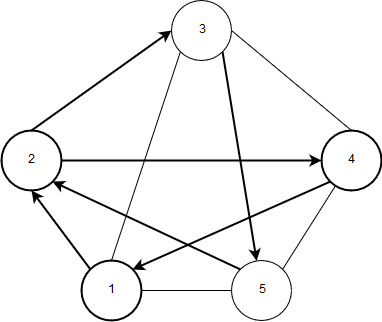
[](http://info.rully.ru/materials/graphs/img/9.png)

**Рисунок 1.9 — Замкнутый маршрут: вершины 2-3-5-4-3-2**

***Цепь*** *—* маршрут, в котором любое ребро графа входит не более одного раза. Если все вершины такого маршрута не повторяются, то цепь называется *простой*. В цепи, изображенной на рисунке 1.10, вершины 2 и 4 называются *концами* цепи. ***Циклом*** называется цепь в которой начальная и конечная точки маршрута являются одной вершиной. На рисунке 11 вершина 2 является началом и концом циклического маршрута.

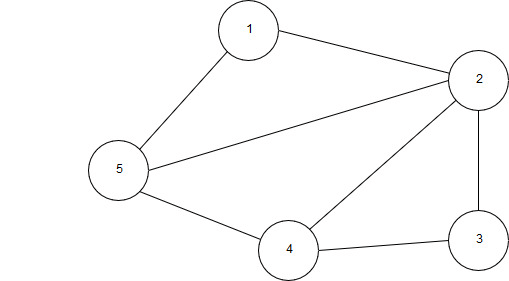
[](http://info.rully.ru/materials/graphs/img/10.png)

**Рисунок 1.10 — Открытая цепь (вершины 2-5-1-2-4)**

[](http://info.rully.ru/materials/graphs/img/11.png)

**Рисунок 1.11 — Замкнутая цепь (вершины 2-4-1-2-3-5-2)**

***Матрица смежности графа*** — это способ представления графа в виде квадратной матрицы, в которой каждый элемент принимает одно из двух значений: 0 или 1 для невзвешенного графа. Значения 1 и 0 отображают существование ребра между вершинами. Например, на рисунке 12 — невзвешенный неориентированный граф, где между вершинами 1 и 2 есть ребро, а значит в матрице смежности на пересечении первой строки и второго столбца и второй строки и первого столбца будут стоять единицы. А на рисунке 13 — невзвешенный неориентированный граф, где из вершины 1 в вершину 2 проходит путь, следовательно в матрице смежности на пересечении первой строки и второго столбца будет стоять единица, а на пересечении второй строки и первого столбца — 0. Если граф взвешенный, то элементами матрицы смежности будут числа, соответствующие весам ребер, соединяющих вершины, на пересечении которых в таблице стоит элемент (рисунок 14).

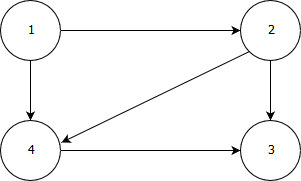
[](http://info.rully.ru/materials/graphs/img/12.png)

**Рисунок 1.12 — Невзвешенный неориентированный граф**

Матрица смежности для графа, изображенного на рисунке 1.12, представлена следующим образом:

Таблица 1.1 — Матрица смежности для графа на рисунке 1.12

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Вершина | 1 | 2 | 3 | 4 | 5 |
| 1 | 0 | 1 | 0 | 0 | 1 |
| 2 | 1 | 0 | 1 | 1 | 1 |
| 3 | 0 | 1 | 0 | 1 | 0 |
| 4 | 0 | 1 | 1 | 0 | 1 |
| 5 | 1 | 1 | 0 | 1 | 0 |

[](http://info.rully.ru/materials/graphs/img/13.png)

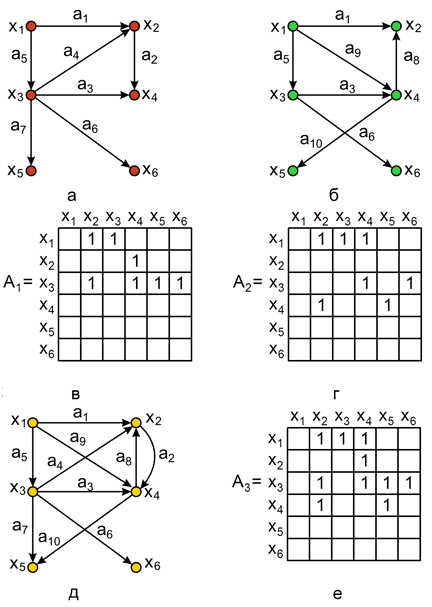
**Рисунок 1.13 — Невзвешенный ориентированный граф**

Матрица смежности для графа, изображенного на рисунке 1.13, представлена следующим образом:

Таблица 2 — Матрица смежности для графа с рисунка 1.13

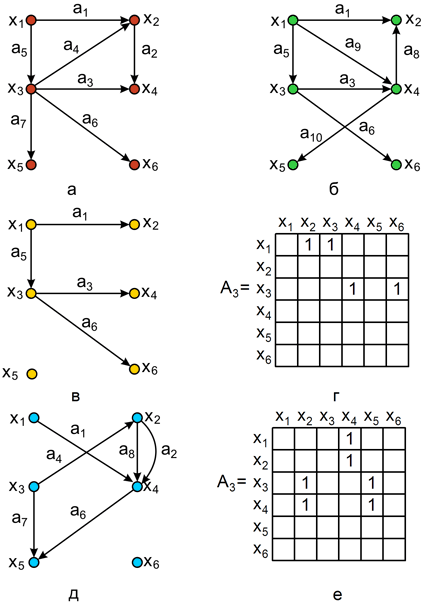
|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Вершина | 1 | 2 | 3 | 4 |
| 1 | 0 | 1 | 0 | 1 |
| 2 | 0 | 0 | 1 | 1 |
| 3 | 0 | 1 | 0 | 0 |
| 4 | 1 | 0 | 1 | 0 |

**Объединение** графов G1 и G2 , представляет такой граф, что множество его вершин является объединением Х1 и Х2 , а множество ребер — объединением A1 и A2 . Граф G3 , полученный операцией объединения графов G1 и G2 , изображен на рисунке 1.14 (д), а его матрица смежности — на рисунке 1.14(е). Матрица смежности результирующего графа получается операцией поэлементного логического сложения матриц смежности исходных графов G1 и G2 .



**Рисунок 1.14 — Объединение графов**

**Пересечение** графов G1 и G2 , представляет собой граф G3, сожержащий только те ребра, которые есть в первом и во втором графах. Таким образом, множество вершин графа G3 состоит из вершин, присутствующих одновременно в G1 и G2 .



**Рисунок 1.15 — Пересечение графов**

2 ПРАКТИЧЕСКАЯ ЧАСТЬ

2.1 Описание объектов системы, их свойств и методов

Диаграмма классов (class diagram) служит для представления статической структуры модели системы в терминологии классов объектно-ориентированного программирования. Диаграмма классов может отражать, в частности, различные взаимосвязи между отдельными сущностями предметной области, такими как объекты и подсистемы, а также описывает их внутреннюю структуру и типы отношений. На данной диаграмме не указывается информация о временных аспектах функционирования системы. С этой точки зрения диаграмма классов является дальнейшим развитием концептуальной модели проектируемой системы.

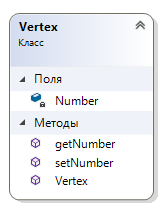
Диаграмма классов представляет собой некоторый граф, вершинами которого являются элементы типа классификатор, которые связаны различными типами структурных отношений. Следует заметить, что диаграмма классов может также содержать интерфейсы, пакеты, отношения и даже отдельные экземпляры, такие как объекты и связи. Когда говорят о данной диаграмме, имеют в виду статическую структурную модель проектируемой системы. Поэтому диаграмму классов принято считать графическим представленном таких структурных взаимосвязей логической модели системы, которые не зависят или инвариантны от времени.

Диаграмма классов состоит из множества элементов, которые в совокупности отражают декларативные знания о предметной области. Эти знания интерпретируются в базовых понятиях языка UML, таких как классы, интерфейсы и отношения между ними и их составляющими компонентами. При этом отдельные компоненты этой диаграммы могут образовывать пакеты для представления более общей модели системы. Если диаграмма классов является частью некоторого пакета, то ее компоненты должны соответствовать элементам этого пакета, включая возможные ссылки на элементы из других пакетов.

В общем случае пакет статической структурной модели может быть представлен в виде одной или нескольких диаграмм классов. Декомпозиция некоторого представления на отдельные диаграммы выполняется с целью удобства и графической визуализации структурных взаимосвязей предметной области. При этом компоненты диаграммы соответствуют элементам статической семантической модели. Модель системы, в свою очередь, должна быть согласована с внутренней структурой классов, которая описывается на языке UML.

Во время решения поставленной задачи были разработаны три основных класса и один класс-точка входа в программу: Edge, Vertex, Graph и MyGraph.

Класс Vertex приведен на рисунке 2.1. Он представляет собой вершину графа.

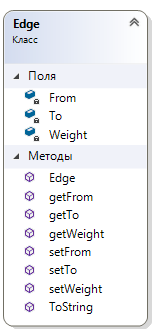


**Рисунок 2.1 — Класс вершины графа**

Таблица 2.1 — Описания полей класса Vertex

|  |  |
| --- | --- |
| Атрибут | Описание |
| Number | Поле. Представляет собой ключ вершины |
| getNumber() | Метод. Геттер для ключа вершины |
| setNumber() | Метод. Сеттер для ключа вершины |
| Vertex(key) | Метод. Конструктор вершины |

Класс Edge приведен на рисунке 2.2. Он представляет собой ребро графа. Если точнее, какая вершина с какой соединяется.



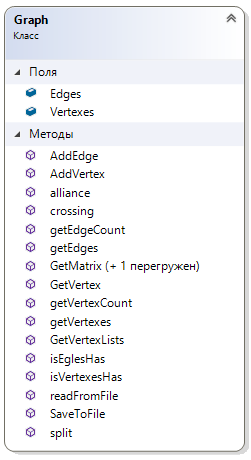
**Рисунок 2.2 — Класс ребер графа**

Таблица 2.2 — Описания полей класса Edge

| Атрибут | Описание |
| --- | --- |
| From | Поле. Содержит вершину, из которой направлено ребро |
| To | Поле. Содержит вершину, в которое направлено ребро |
| Weight | Поле. Содержит вес ребра |
| Edge(from,to, [weight]) | Метод. Конструктор класса, получающий вершины начала и конца ребра. При необходимости, принимает вес |
| getFrom() | Метод. Геттер |
| getTo() | Метод. Геттер |
| getWeight() | Метод. Геттер |
| setFrom() | Метод. Сеттер |
| setTo() | Метод. Сеттер |
| setWeight() | Метод. Сеттер |
| ToString() | Метод. Строчное отображение класса |

Класс Graph приведен на рисунке 2.3. Он представляет собой полноценный граф. Состоит из списка ребер и списка вершин. Если добавлять ребра в двух направлениях, граф будет неориентированным, если же добавлять некоторые вершины в двух направлениях, а некоторые в одном — граф будет смешанным. Класс также содержит общие методы работы с графом.

Также, некоторые методы проверяют вводимые значения на нулевые, так что нельзя будет использовать NULL в качестве ребра или вершины графа



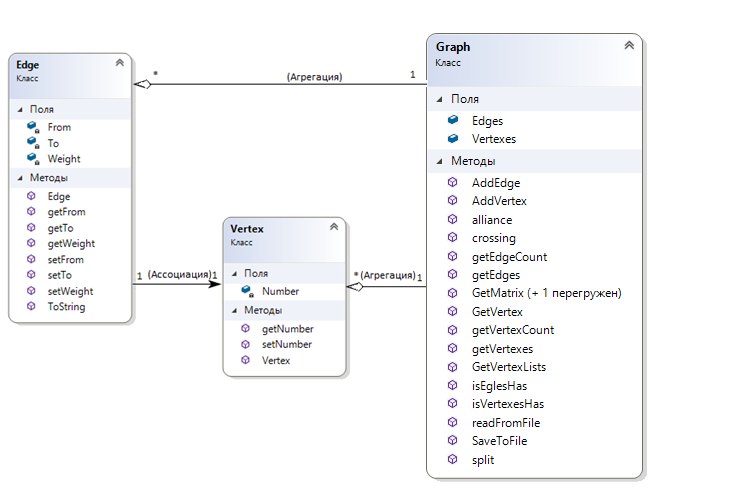
**Рисунок 2.3 — Класс графа**

В таблице 2.3 представлено описание полей и методов класса.

Таблица 2.3 — Описания полей класса Graph

|  |  |
| --- | --- |
| Атрибут | Описание |
| Edges | Поле. Список ребер |
| Vertexes | Поле. Список вершин |
| AddEdge(from,to) | Метод добавления ребра |
| Alliance(graph1, graph2) | Метод объединения графов |
| Crossing(graph1, graph2) | Метод пересечения графов |
| getEdgeCount() | Метод. Возвращает количество ребер |
| getEdges() | Метод. Геттер |
| getMatrix() | Метод. Возвращает или печатает матрицу смежности |
| getVertex(graph, vertex) | Метод. Печатает вершины, до которых идет ребро от заданной вершины |
| getVertexCount() | Метод. Возвращает количество вершин |
| getVertexes() | Метод. Геттер |
| getVertexLists(vertex) | Метод. Возвращает вершины, до которых идет ребро от заданной вершины |
| isEglesHas() | Метод. Проверяет, есть ли в графе заданное ребро |
| isVertexesHas() | Метод. Проверяет, есть ли в графе заданная вершина |
| saveToFile(filename) | Метод. Сохраняет данный граф в текстовый файл. |
| readFromFile(filename) | Метод. Читает граф из файла. |
| Split(phrase, delimeter) | Метод. Разделяет строку по делиметру. Используется при чтении графа из файла |

Общая диаграмма классов проекта представлена на рисунке 2.4.



**Рисунок 2.3 — общая диаграмма классов**

2.2 Результаты тестирования программного продукта

Самое время проверить работоспособность созданных классов. Используем для этого класс-точку входа MyGraph.

Первым делом создадим Граф, добавим в него вершины и ребра и попробуем вывести матрицу смежности на экран.

Листинг 1 — Создание, заполнение графа и вывод его матрицы смежности

Graph\* graph1 = new Graph();

Vertex\* g1v1 = new Vertex(1);

Vertex\* g1v2 = new Vertex(2);

Vertex\* g1v3 = new Vertex(3);

Vertex\* g1v4 = new Vertex(4);

Vertex\* g1v5 = new Vertex(5);

Vertex\* g1v6 = new Vertex(6);

Vertex\* g1v7 = new Vertex(7);

graph1->AddVertex(g1v1);

graph1->AddVertex(g1v2);

graph1->AddVertex(g1v3);

graph1->AddVertex(g1v4);

graph1->AddVertex(g1v5);

graph1->AddVertex(g1v6);

graph1->AddVertex(g1v7);

graph1->AddEdge(g1v1, g1v2);

graph1->AddEdge(g1v1, g1v3);

graph1->AddEdge(g1v3, g1v4);

graph1->AddEdge(g1v2, g1v5);

graph1->AddEdge(g1v2, g1v6);

graph1->AddEdge(g1v6, g1v5);

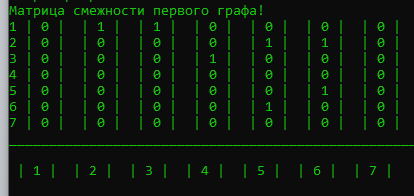
graph1->AddEdge(g1v5, g1v6);

//матрица смежности

std::cout << "Матрица смежности первого графа!\n";

graph1->GetMatrix(graph1);

Получаем следующий результат



**Рисунок 2.5 — Матрица смежности графа**

Сверив матрицу смежности с ребрами, которые мы добавляли в граф, можно заметить, что матрица отображает все верно.

Попробуем пройтись по вершинам графа и вывести вершины, в которые можно попасть

Листинг 2 — Вывод вершин, в которые можно попасть из заданной вершины

std::cout << "Вершины, в которые можно попасть из вершин первого графа!\n";

Graph().GetVertex(graph1, g1v1);

Graph().GetVertex(graph1, g1v2);

Graph().GetVertex(graph1, g1v3);

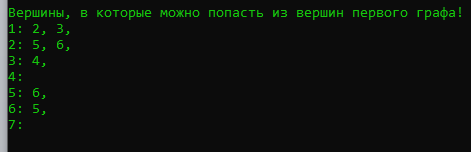
Graph().GetVertex(graph1, g1v4);

Graph().GetVertex(graph1, g1v5);

Graph().GetVertex(graph1, g1v6);

Graph().GetVertex(graph1, g1v7);

Как результат, получаем следующее



**Рисунок 2.6 - Вывод вершин, в которые можно попасть из заданной вершины**

Таким образом мы удостоверились, что методы класса работают, а также что вывод соответствует матрице смежности. Перейдем к конкретным методам индивидуального задания

Создадим другой граф, заполним его и выведем матрицу.

Листинг 3 — Создание другого графа

Graph\* graph2 = new Graph();

Vertex\* g2v1 = new Vertex(1);

Vertex\* g2v2 = new Vertex(2);

Vertex\* g2v3 = new Vertex(3);

graph2->AddVertex(g2v1);

graph2->AddVertex(g2v2);

graph2->AddVertex(g2v3);

graph2->AddEdge(g2v1, g2v2);

graph2->AddEdge(g2v1, g2v3);

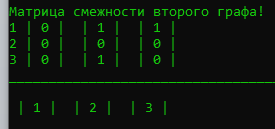
graph2->AddEdge(g2v3, g2v2);

//матрица смежности

std::cout << "Матрица смежности второго графа!\n";

Graph().GetMatrix(graph2);

Как результат, получаем следующее



**Рисунок 2.7 — Матрица смежности другого графа**

Теперь используем метод объединения графов. Для проверки результата, выведем матрицу полученного графа.

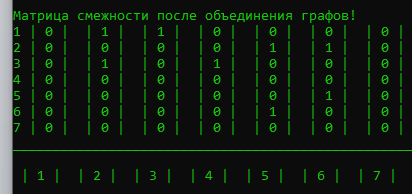
Листинг 4 — Использование метода объединения графов

Graph\* alliance = Graph().alliance(graph1, graph2);

std::cout << "Матрица смежности после объединения графов!\n";

Graph().GetMatrix(alliance);

Получаем результат



**Рисунок 2.8 — Результат объединения графов**

По рисунку можно заметить, что ребра, которых не было в первом графе (из 3 в 1), но были во втором, появились в результате объединения графов, что говорит о правильной работоспособности.

Таким же образом проверим метод пересечения графов

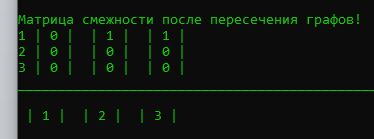
Листинг 5 — Использование метода пересечения графов

Graph\* crossing = Graph().crossing(graph1, graph2);

std::cout << "Матрица смежности после пересечения графов!\n";

Graph().GetMatrix(crossing);

Результатом получаем



**Рисунок 2.9 — Результат пересечения графов**

Сохраним объединение графов в фай, затем попробуем прочесть сохраненный файл в другую переменную и вывести результат на экран.

Листинг 6 – Сохранение графа в файл и чтение из него

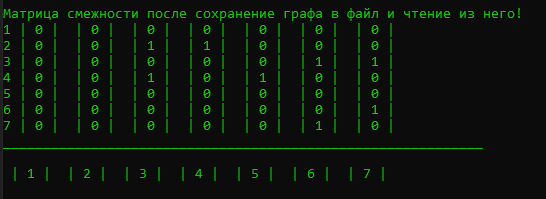
alliance->SaveToFile("alliance.txt");

graph2=Graph().readFromFile("alliance.txt");

std::cout << "Матрица смежности после сохранение графа в файл и чтение из него!\n";

Graph().GetMatrix(graph2);

Результатом получаем



**Рисунок 2.10 — Результат сохранения и чтения графа из файла**

Прочитанный граф полностью совпадает с записанным объединением ранее созданных графов.

Протестировав всю функциональность можно сказать, что разработанные классы выполняют свою работу без проблем.

ЗАКЛЮЧЕНИЕ

В результате выполнения курсового проекта была разработана иерархия классов, реализующая ориентированный и смешанный графы.

Она позволяет:

* Создавать собственные графы программно
* Добавлять вершины и ребра выбранным графам
* Получать граф на основе пересечения двух других заданных графов
* Получать граф на основе соединения двух других графов
* Выводить матрицу смежности заданного графа
* Получать и выводить список вершин, в которые можно попасть из текущей вершины заданного графа

Таким образом, задачи курсового проекта можно считать полностью выполненными: проведена постановка задачи, описаны принципы написания приложений используя объектно-ориентированный подход, благодаря описанию области реализован программный продукт, который успешно прошел стадию тестирования.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Основные положения теории графов [Электронный ресурс]. — Режим доступа: https://studfile.net/preview/6179451/page:2/. — Дата доступа: 17.11.2020.
2. Объектно-ориентированный подход [Электронный ресурс]. — Режим доступа: https://intuit.ru/studies/courses/10/10/lecture/298?page=2. — Дата доступа: 17.11.2020.
3. Алгоритмы на графах [Электронный ресурс]. — Режим доступа: http://math.gsu.by/wp-content/uploads/courses/structure/L9.1.html. — Дата доступа: 17.11.2020.
4. Введение в теорию графов [Электронный ресурс]. — Режим доступа: http://info.rully.ru/materials/graphs/. — Дата доступа: 17.11.2020.
5. Диаграмма классов [Электронный ресурс]. — Режим доступа: https://it.wikireading.ru/5119. — Дата доступа: 17.11.2020.

ПРИЛОЖЕНИЕ

**ТЕКСТ ПРОГРАММЫ НА ЯЗЫКЕ C++**

#pragma once

#include "Vertex.h"

#include <string>

// класс ребра

class Edge

{

private:

// вершина из которой направлено ребро

Vertex\* From;

// вершина в которую направлено ребро

Vertex\* To;

// вес ребра

int Weight = 0;

public:

// методы доступа к вершинам ребра и весу

Vertex\* getFrom() const;

void setFrom(Vertex\* value);

Vertex\* getTo() const;

void setTo(Vertex\* value);

int getWeight() const;

void setWeight(int value);

// конструктор, по умолчанию вес 1

Edge(Vertex\* from, Vertex\* to, int weight = 1);

// перевод в строку

std::wstring ToString();

};

#define \_CRT\_SECURE\_NO\_WARNINGS

#include "Edge.h"

#include "Vertex.h"

// реализация гет/сеттеров и конструктора

Vertex \*Edge::getFrom() const

{

return From;

}

void Edge::setFrom(Vertex \*value)

{

From = value;

}

Vertex \*Edge::getTo() const

{

return To;

}

void Edge::setTo(Vertex \*value)

{

To = value;

}

int Edge::getWeight() const

{

return Weight;

}

void Edge::setWeight(int value)

{

Weight = value;

}

Edge::Edge(Vertex \*from, Vertex \*to, int weight)

{

if (from == NULL || to == NULL)return;

setFrom(from);

setTo(to);

setWeight(weight);

}

#pragma once

#include <string>

#include <iostream>

// класс вершин графа

class Vertex

{

private:

// ключевое поле для определения

int Number = 0;

public:

// методы доступа к ключу

int getNumber() const;

void setNumber(int value);

// конструктор

Vertex(int number);

};

#define \_CRT\_SECURE\_NO\_WARNINGS

#include "Vertex.h"

// реализация гет/сеттеров и конструктора

int Vertex::getNumber() const

{

return Number;

}

void Vertex::setNumber(int value)

{

Number = value;

}

Vertex::Vertex(int number)

{

setNumber(number);

}

#pragma once

#define \_CRT\_SECURE\_NO\_WARNINGS

#include "Vertex.h"

#include <vector>

#include <list>

#include <algorithm>

#include "Edge.h"

#include <list>

#include <iostream>

#include "string"

using namespace std;

// класс графа

class Graph

{

public:

// список вершин графа

std::list<Vertex\*> Vertexes = std::list<Vertex\*>();

// список ребер графа

std::list<Edge\*> Edges = std::list<Edge\*>();

// реализация гет/сеттеров

int getVertexCount() const

{

return Vertexes.size();

}

int getEdgeCount() const

{

return Edges.size();

}

list<Vertex\*> getVertexes() const

{

return Vertexes;

}

list<Edge\*> getEdges() const

{

return Edges;

}

// метод добавления вершины

void AddVertex(Vertex\* vertex)

{

if (vertex == NULL)return;

Vertexes.push\_back(vertex);

}

// метод добавления ребра

void AddEdge(Vertex\* from, Vertex\* to)

{

if (from == NULL || to == NULL)return;

auto edge = new Edge(from, to);

Edges.push\_back(edge);

}

// метод получения матрицы смежности графа

int\*\* GetMatrix()

{

// инициализация массива

int\*\* matrix = new int\* [100];

for (int i = 0; i < 100; i++) {

matrix[i] = new int[100];

for (int j = 0; j < 100; j++){

matrix[i][j] = 0;}

}

// запись в массив матрицы смежности

for (Edge\* edge : Edges)

{

int row = edge->getFrom()->getNumber() - 1;

int column = edge->getTo()->getNumber() - 1;

matrix[row][column] = edge->getWeight();

}

return matrix;

}

// метод определения списка вершин, в которые можно попасть из текущегей вершины

std::list<Vertex\*> GetVertexLists(Vertex\* vertex)

{

std::list<Vertex\*> result = std::list<Vertex\*>();

if (vertex == NULL)return result;

for (Edge\* edge : Edges)

{

if (edge->getFrom() == vertex)

{

result.push\_back(edge->getTo());

}

}

return result;

}

// метод проверки, есть ли в текущем графе переданное ребро

bool isEglesHas(Edge\* \_edge) {

if (\_edge == NULL)return false;

for (Edge\* edge : Edges)

{

if (\_edge->getFrom()->getNumber() == edge->getFrom()->getNumber() &&

\_edge->getTo()->getNumber() == edge->getTo()->getNumber())

return true;

}

return false;

}

// метод проверки, есть ли в текущем графе переданная вершина

bool isVertexesHas(Vertex\* \_vertex) {

if (\_vertex == NULL)return false;

for (Vertex\* vertex : Vertexes)

{

if (\_vertex->getNumber()==vertex->getNumber())

return true;

}

return false;

}

//метод пересечения графов

static Graph\* crossing(Graph\* g1, Graph\* g2) {

Graph\* graph = new Graph();

if (g1 == NULL || g2==NULL)return graph;

//пробегаем первый граф по дорогам

for (Edge\* e1 : g1->getEdges())

{

//если такая дорога есть во втором графе

if (g2->isEglesHas(e1)) {

//если таких вершин еще нет, добавляем

if (!graph->isVertexesHas(e1->getFrom()))

graph->AddVertex(e1->getFrom());

if (!graph->isVertexesHas(e1->getTo()))

graph->AddVertex(e1->getTo());

//добавляем в граф пересечения

graph->AddEdge(e1->getFrom(), e1->getTo());

}

}

return graph;

}

// метод объединения графов

static Graph\* alliance(Graph\* g1, Graph\* g2) {

Graph\* graph = new Graph();

if (g1 == NULL || g2 == NULL)return graph;

//пробегаем первый граф по вершинам

for (Vertex\* v1 : g1->getVertexes())

graph->AddVertex(v1);

//пробегаем второй граф по вершинам

for (Vertex\* v2 : g2->getVertexes())

//если таких нет, не добавляем

if(!graph->isVertexesHas(v2))

graph->AddVertex(v2);

//пробегаем первый граф по дорогам

for (Edge\* e1 : g1->getEdges())

graph->AddEdge(e1->getFrom(),e1->getTo());

//пробегаем второй граф по дорогам

for (Edge\* e2 : g2->getEdges())

// если таких нет, то добавляем

if(!graph->isEglesHas(e2))

graph->AddEdge(e2->getFrom(), e2->getTo());

return graph;

}

//вывод матрицы графа на экран

static void GetMatrix(Graph\* graph)

{

if (graph == NULL )return ;

int\*\* matrix = graph->GetMatrix();

for (int i = 0; i < graph->getVertexCount(); i++)

{

std::wcout << i + 1;

for (int j = 0; j < graph->getVertexCount(); j++)

{

std::wcout << L" | " << matrix[i][j] << L" | ";

}

std::wcout << std::endl;

}

std::wcout << L"\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_" << std::endl;

std::wcout << L" " << std::endl;

for (int i = 0; i < graph->getVertexCount(); i++)

{

string temp = " | " + to\_string(i + 1) + " | ";

std::cout << temp;

}

}

//вывод на экран вершин, в которые можно попасть из вершины заданного графа

static void GetVertex(Graph\* graph, Vertex\* vertex)

{

if (graph == NULL || vertex == NULL)return ;

std::wcout << vertex->getNumber() << L": ";

for (Vertex\* v : graph->GetVertexLists(vertex))

{

std::wcout << v->getNumber() << L", ";

}

std::wcout << std::endl;

}

void SaveToFile(string filename) {

ofstream fout(filename); // создаём объект класса ofstream для записи и связываем его с файлом cppstudio.txt

int\*\* matrix = GetMatrix();

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

{

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

{

fout << matrix[i][j];

if (j + 1 < getVertexCount())fout << " ";

}

if (i + 1 < getVertexCount())fout << std::endl;

}

fout.close(); // закрываем файл

}

//метод чтения из файла

Graph\* readFromFile(string filename) {

Graph\* graph1 = new Graph();

char buff[50];

ifstream file(filename); // файл из которого читаем

file.getline(buff, 50);

//разбиваем первую строку

vector<string> listValues = split(buff, " ");

//создаем по ней нужное количество вершин

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

graph1->AddVertex(new Vertex(i));

//после проходим каждую строку и создаем ребра

int row = 0;

do {

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

{

if (std::stoi(listValues[i]) != 0)

graph1->AddEdge(graph1->getVertexes()[row], graph1->getVertexes()[i]);

}

file.getline(buff, 50);

listValues = split(buff, " ");

row++;

} while (listValues.size() > 1);

file.close(); // обязательно закрываем файл что бы не повредить его

return graph1;

}

// метод разделение строки по разделителю

vector<string> split(char\* phrase, string delimiter) {

vector<string> list;

string s = string(phrase);

size\_t pos = 0;

string token;

while ((pos = s.find(delimiter)) != string::npos) {

token = s.substr(0, pos);

list.push\_back(token);

s.erase(0, pos + delimiter.length());

}

list.push\_back(s);

return list;

}

};#define \_CRT\_SECURE\_NO\_WARNINGS

#include "Graph.h"

#include <iostream>

#include "string"

#include "Graph.h"

int main()

{

setlocale(LC\_ALL, "Russian");

std::cout << "Старт программы!\n";

Graph\* graph1 = new Graph();

Vertex\* g1v1 = new Vertex(1);

Vertex\* g1v2 = new Vertex(2);

Vertex\* g1v3 = new Vertex(3);

Vertex\* g1v4 = new Vertex(4);

Vertex\* g1v5 = new Vertex(5);

Vertex\* g1v6 = new Vertex(6);

Vertex\* g1v7 = new Vertex(7);

graph1->AddVertex(g1v1);

graph1->AddVertex(g1v2);

graph1->AddVertex(g1v3);

graph1->AddVertex(g1v4);

graph1->AddVertex(g1v5);

graph1->AddVertex(g1v6);

graph1->AddVertex(g1v7);

graph1->AddEdge(g1v1, g1v2);

graph1->AddEdge(g1v1, g1v3);

graph1->AddEdge(g1v3, g1v4);

graph1->AddEdge(g1v2, g1v5);

graph1->AddEdge(g1v2, g1v6);

graph1->AddEdge(g1v6, g1v5);

graph1->AddEdge(g1v5, g1v6);

//матрица смежности

std::cout << "Матрица смежности первого графа!\n";

graph1->GetMatrix(graph1);

std::wcout << std::endl;

std::wcout << std::endl;

std::wcout << std::endl;

std::wcout << std::endl;

std::wcout << std::endl;

std::wcout << std::endl;

Graph\* graph2 = new Graph();

Vertex\* g2v1 = new Vertex(1);

Vertex\* g2v2 = new Vertex(2);

Vertex\* g2v3 = new Vertex(3);

graph2->AddVertex(g2v1);

graph2->AddVertex(g2v2);

graph2->AddVertex(g2v3);

graph2->AddEdge(g2v1, g2v2);

graph2->AddEdge(g2v1, g2v3);

graph2->AddEdge(g2v3, g2v2);

//матрица смежности

std::cout << "Матрица смежности второго графа!\n";

Graph().GetMatrix(graph2);

std::wcout << std::endl;

std::wcout << std::endl;

std::wcout << std::endl;

std::wcout << std::endl;

std::wcout << std::endl;

std::wcout << std::endl;

//куда ведут дороги

std::cout << "Вершины, в которые можно попасть из вершин первого графа!\n";

Graph().GetVertex(graph1, g1v1);

Graph().GetVertex(graph1, g1v2);

Graph().GetVertex(graph1, g1v3);

Graph().GetVertex(graph1, g1v4);

Graph().GetVertex(graph1, g1v5);

Graph().GetVertex(graph1, g1v6);

Graph().GetVertex(graph1, g1v7);

std::wcout << std::endl;

std::wcout << std::endl;

std::wcout << std::endl;

std::wcout << std::endl;

std::wcout << std::endl;

std::wcout << std::endl;

Graph\* alliance = Graph().alliance(graph1, graph2);

std::cout << "Матрица смежности после объединения графов!\n";

Graph().GetMatrix(alliance);

std::wcout << std::endl;

std::wcout << std::endl;

std::wcout << std::endl;

Graph\* crossing = Graph().crossing(graph1, graph2);

std::cout << "Матрица смежности после пересечения графов!\n";

Graph().GetMatrix(crossing);

std::wcout << std::endl;

std::wcout << std::endl;

std::wcout << std::endl;

alliance->SaveToFile("alliance.txt");

graph2=Graph().readFromFile("alliance.txt");

std::cout << "Матрица смежности после сохранение графа в файл и чтение из него!\n";

Graph().GetMatrix(graph2);

std::wcout << std::endl;

std::wcout << std::endl;

std::cout << "Конец программы!\n";}