Федеральное государственное бюджетное образовательное учреждение

высшего образования

«Саратовский государственный технический университет

имени Гагарина Ю.А.»

Институт прикладных информационных технологий и коммуникаций

Кафедра информационно-коммуникационных систем и программной инженерии

Направление 09.03.01 Информатика и вычислительная техника

ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА

АВТОМОРФИЗМЫ ГИПЕРГРАФОВ ОСОБОГО ВИДА

Выполнил студент

Володин Иван Олегович

курс 4

группа б1ИВЧТ41

Руководитель работы:

к.ф-м.н., доцент кафедры ИКСП

Хворостухина Екатерина Владимировна

Допущен к защите

Протокол № _____ от _____2018 г.

Саратов 2019

Зав. кафедрой ИКСП д.т.н., профессор _____А.А. Сытник

СОДЕРЖАНИЕ

Y

ВВЕДЕНИЕ

В последнее время наблюдается возросший интерес к изучению задач и методов теории гиперграфов [1]. Это связано с тем, что гиперграфы нашли применение в задачах управления большими системам, при моделировании и проектировании электрических систем, компьютерных сетей, процессов, при обработке изображений и во многих других сферах. К тому же гиперграф — это естественное обобщение графа (ребрами гиперграфа являются произвольные подмножества его вершин), конечной плоскости, разбиения множества. Многие авторы значительное внимание уделяли исследованию полугрупп эндоморфизмов и полугрупп автоморфизмов графов и гиперграфов [2,3], так как, исследуя свойства таких полугрупп отображений, можно изучать свойства исходного объекта. Так, применению методов теории групп при изучении графов посвящены известные работы Р.Фрухта, Ф.Харари, Г.Сабидусси и многих других (см., например, [4]). Методы алгебры и комбинаторики применяются и при исследовании гиперграфов [5,6].

Цель выпускной квалификационной работы – найти комбинаторные формулы для расчета количества автоморфизмов гиперграфов некоторых классов. Объектами исследования являются гиперграфы особого вида, а предметом исследования – автоморфизмы таких гиперграфов.

Для достижения цели данной работы были поставлены следующие задачи:

– изучить основные понятия теории гиперграфов;

– построить и проанализировать различные виды гиперграфов;
– вывести формулы, которые позволят точно рассчитать количество автоморфизмов для гиперграфа изучаемого класса;
– применить полученные формулы для расчета числа автоморфизмов на конкретных примерах;
– разработать приложение, предназначенное для расчета количества автоморфизмов гиперграфов особого вида.

Выпускная квалификационная работа состоит из четырех разделов, введения, заключения и списка литературы.

В первом разделе излагаются основные понятия теории гиперграфов и теории множеств, изучаются основные способы представления гиперграфов, также дается обзор областей применения теории гиперграфов.

Во втором разделе содержатся анализ основных видов гиперграфов, вывод формул для вычисления количества автоморфизмов гиперграфов рассматриваемых классов, примеры применения полученных формул.

В третьем разделе приводится описание программного продукта и содержание основных файлов разработанного приложения.

В четвертом разделе показаны результаты тестирования приложения на примерах.

В приложении содержится код разработанного программного продукта.

1.Основные понятия теории гиперграфов

Приведем основные понятия теории графов и гиперграфов [1], [2], [7], [8], [9].

1.1 Понятие гиперграфа

Определение 1. Граф представляет собой упорядоченную пару G = (V, E) множеств, первое из которых состоит из вершин, или узлов, графа, а второе из его ребер – пар вершин. Ребро связывает между собой две вершины. В случае если пары вершин упорядоченные, то граф называют ориентированным; противном случае – неориентированным.

Пример 1. На рисунке 1 представлен неориентированный граф, содержащий три вершины и три ребра.

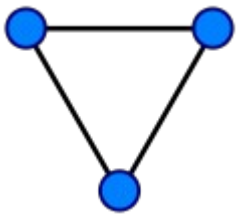

Рисунок 1.1 − Граф

Определение 2. Гиперграфом H=(V,E) называется пара (V,E),где V – это некоторое конечное множество, элементы которого называются вершинами гиперграфа, а E – набор произвольных подмножеств V, называемых ребрами (или гиперребрами) гиперграфа.

Определение 3. Гиперграф H=(V,E) называется конечным, если множество его вершин V является конечным множеством, то есть состоит из конечного числа элементов.

Определение 4. Гиперграф H=(V,E) называется бесконечным, если множество его вершин V бесконечно, то есть состоит из бесконечного числа элементов.

Далее в работе будем рассматривать только конечные гиперграфы.

Для наглядности гиперграфы, как и графы, часто изображают в виде рисунка, где вершинам соответствуют точки, а ребрам – линии или области, содержащие соответствующие вершины.

Пример 2. На рисунке 1.2 показан пример гиперграфа, где ребрам соответствуют закрашенные области.

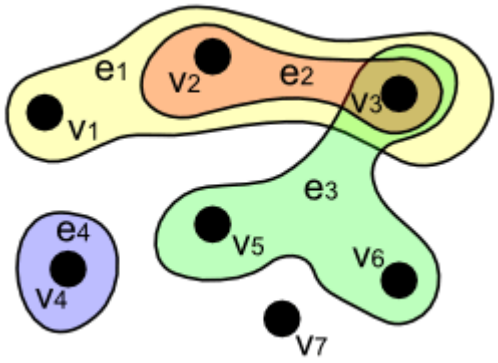

Рисунок 1.2 − Гиперграф

Определение 5: Вершины называются смежными, если они принадлежат одному ребру. Множество вершин, содержащееся в некотором ребре, называется ограниченным, и неограниченным в противном случае.

Определение 6: Длиной или мощностью ребра называется количество вершин, принадлежащих данному ребру.

Определение 7: Автоморфизм гиперграфа H=(V,E) − это взаимно однозначное отображение λ множества вершин V на

себя, которое сохраняет смежность, т.е. удовлетворяет условию:

$$(\forall X \subset V)(X \in E \Longleftrightarrow \lambda(X) \in E).$$

Другими словами, автоморфизм гиперграфа – это подстановка на множестве его вершин, оставляющая без изменения его список ребер.

На рисунке 1.3 изображен пример автоморфизма гиперграфа H=(V,E), V={A,B,C,D,E}, E={{A,B,C},{A,D,E}}.

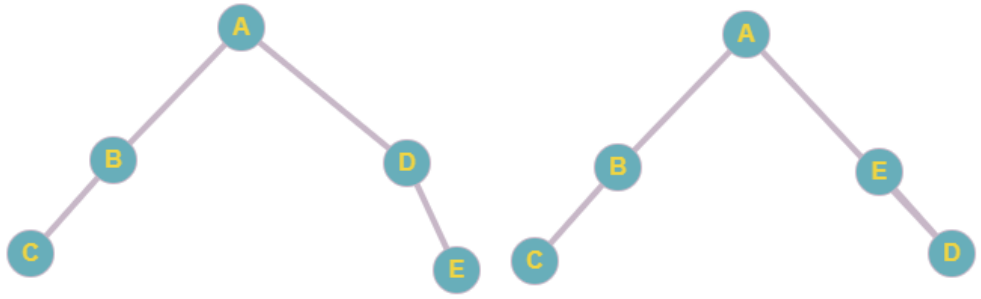

Рисунок 1.3 − автоморфизмы гиперграфа

Множество всех автоморфизмов данного гиперграфа H образует группу относительно операции композиции автоморфизмов, которую принято обозначать Aut H.

Определение 8: Биекция − это такое отображение, где каждому элементу одного множества соответствует ровно один элемент другого множества, при этом определено обратное отображение, которое обладает тем же свойством.

1.2 Примеры гиперграфов

Пример 1. В школьном классе обучается 30 человек. Всего в этом классе ведется 15 предметов, и, допустим, по каждому предмету есть 5 лучших учащихся. Тогда можно построить гиперграф H=(V,E), где вершинами будут люди, обучающиеся

в этом классе, а ребрами − предметы, и вершина v будет входить в ребро e, если человек v является одним из пяти лучших по предмету e, всего ребер в этом гиперграфе будет 15 (по 5 вершин каждое),а вершин 30.

Пример 2. Наглядный пример гиперграфа дает схема Петербургского метрополитена, которая изображена на рисунке 1.5, где в качестве вершин выступают станции метро, а в качестве ребер – линии метро.

Рисунок 1.4 − Схема Петербургского метрополитена

1.3 Способы задания гиперграфов

Помимо рассмотренного выше геометрического (или графического) способа задания гиперграфа, также часто используют алгебраический и матричный способы задания гиперграфов [10].

1.3.1 Алгебраический способ задания гиперграфов

Гиперграф H=(V,E) представляет пару V и E, поэтому, чтобы задать его, можно явно перечислить его вершины и ребра. Например, на рисунке 1.5 изображен гиперграф H=(V,E), где

V={v1, v2, v3, v4, v5 ,v6, v7};

E={e1 ,e2, e3, e4}={{v1, v2, v3}, {v2, v3}, {v3, v5, v6}, {v4}}. Заметим, что вершина 7 не содержится ни в одном ребре, то есть является изолированной.

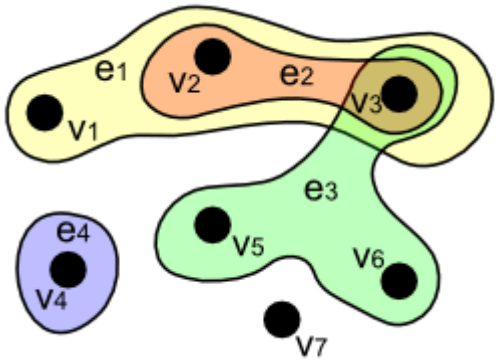

Рисунок 1.5 – Гиперграф H

1.3.2 Матричный способ задания гиперграфов

Пусть H=(V;E) – гиперграф, где $V=\{v1,v2....v_n\}$ и $E=\{e1,e2,.....e_m\}$. Тогда матрица инцидентности A для гиперграфа H определяется следующим образом:

$A=(a_{ij})$,где

-a_{ij}=1 если $v_i \in e_j$,

- $a(i,j)$=¿0 в противном случае, i=1,2,..,n, j=1,2,...,m.

Эта матрица может быть записана в виде таблицы. Матрица инцидентности для гиперграфа H, изображенного на рисунке 1.6, приведена в таблице 1.

Пусть H = (V;E) — гиперграф, тогда матрица смежности A(H) гиперграфа H определяется следующим образом: это квадратная матрица, где строки и столбцы индексированы вершинами гиперграфа H и для всех x, y ∈ V , x ≠ y элемент матрицы a(x, y) = |{e ∈ E : x, y ∈ e}| и a(x ,x) = 0. Пример матрицы смежности приведен в таблице 2 для гиперграфа, представленного на рисунке 1.6

Таблица 1 − Матрица инцидентности для гиперграфа H

	u1	u2	u3	u4
X1	1	0	1	0
X2	0	1	1	0
X3	1	1	0	1
X4	1	1	0	1
X5	0	0	1	1

Таблица 2 − Матрица смежности для гиперграфа H

	X1	X2	X3	X4	X5
X1	0	1	1	1	1
X2	1	0	1	1	1
X3	1	1	0	3	1
X4	1	1	3	0	1
X5	1	1	1	1	0

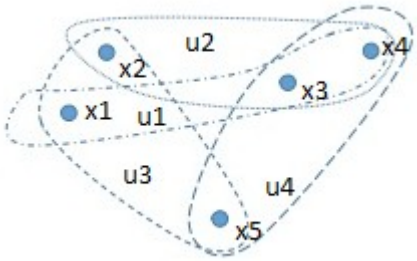

Рисунок 1.6 − Гиперграф H

1.4 Виды гиперграфов

Гиперграф называется k-однородным, если ∀e ∈ E имеет место |e| = k или другими словами каждое ребро состоит ровно из k-вершин. Такой гиперграф представлен на рисунке 1.7.

Заметим, что обыкновенный граф, не содержащий петель, является 2-однородным гиперграфом.

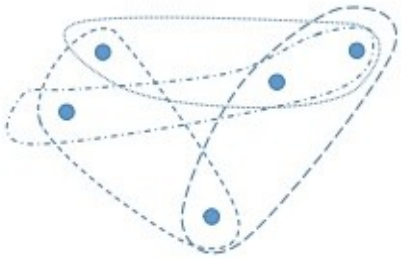

Рисунок 1.7 − 3-однородный гиперграф

K-пересекающийся гиперграф − это гиперграф, любые два ребра которого содержат не менее k общих вершин, т.е. выполняется условие

$$(\forall e, r \in E)(\exists X \subset V \bigwedge |X| = k)(e \cap r = X).$$

Очевидно, что у k-пересекающегося гиперграфа каждое ребро содержит не менее k-вершин.

Строго k-пересекающийся гиперграф − это гиперграф, ребра которого содержат ровно k общих вершин, т.е. выполняется условие

$$(\exists X \subset V \wedge |X| = k)(\forall e, r \in E)(e \cap r = X).$$

Заметим, что строго k-пересекающийся гиперграф является также k-пересекающимся гиперграфом, обратное в общем случае неверно.

Пример строго 1-пересекающегося гиперграфа представлен на рисунке 1.8, общей вершиной для всех его ребер является вершина 0: X={0}.

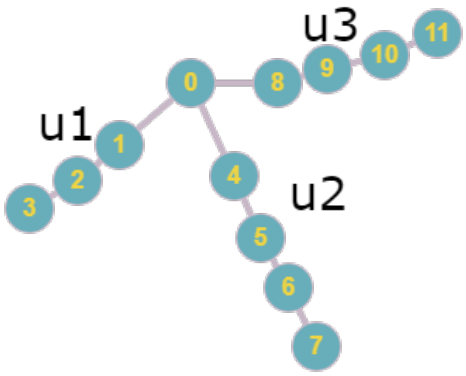

Рисунок 1.8 − однопересекающийся гиперграф

2. Вывод формул для подсчета количества автоморфизмов гиперграфов особого вида

2.1 Однореберный гиперграф

Подсчитаем количество автоморфизмов для гиперграфов особого вида.

Теорема 1: Пусть H=(V,E) − гиперграф, у которого $V=\{v_1,v_2,\ldots,v_n\}$, а множество ребер состоит из единственного ребра, содержащего все вершины гиперграфа, т.е. E={e1}={{ $v_1,v_2,\ldots,v_n$ }}. Тогда число автоморфизмов такого гиперграфа будет равно n!.

Доказательство: Действительно, если гиперграф содержит n вершин, то число их перестановок равно n!. Так, переставляя первую вершину, у нас имеется n вариантов, переставляя вторую, имеем n-1 вариантов, и так далее, переставляя n-ю вершину, у нас останется только один вариант. Таким образом, получаем n*(n-1)*(n-2)*....=n! способов. Ясно, что такие перестановки множества вершин являются биекциями. По условию теоремы все вершины гиперграфа содержатся в одном ребре, т.е. смежные, следовательно, любая перестановка этих вершин определяет отображение, являющееся автоморфизмом гиперграфа. Таким образом, число автоморфизмов гиперграфа H равно n!. Теорема доказана.

Пример 2.1. Рассмотрим пример. На рисунке 2.1 дан гиперграф с 4 вершинами и одним ребром, которое содержит в себе все вершины данного гиперграфа.

Рисунок 2.1 − Гиперграф

Вершину под цифрой 0 можно переместить четырьмя разными способами (см. рисунок 2.2).

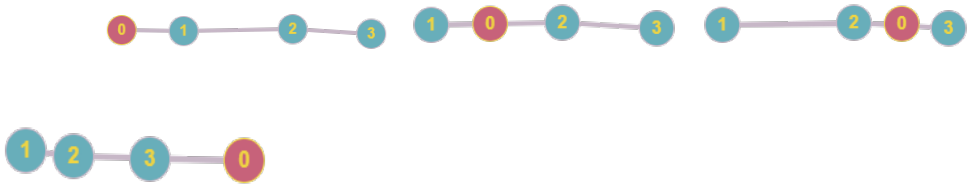

Рисунок 2.2 − Перестановки вершины 0 гиперграфа

После «перемещения» вершины 0 в одну из вершин 0,1,2,3, для «перемещения» вершины под цифрой 1 остается три разных способа (см. рисунок 2.3).

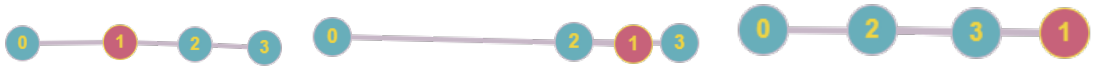

Рисунок 2.3 − Перестановки вершины 1 гиперграфа

Рассуждая по аналогии, получается, что для вершины под цифрой 2 будет 2 варианта перемещения, а для вершины под цифрой 3 остается всего один способ.

Таким образом, число автоморфизмов для рассматриваемого гиперграфа H рассчитывается по формуле: 4*(4-1)*(4-2)*(4-3)=4*3*2*1=4!=24. Таким образом, число автоморфизмов гиперграфа H равно 24.

2.2 Строго k-пересекающийся гиперграф с ребрами разной мощности

Теорема 2: Пусть дан строго k-пересекающийся гиперграф H=(V,E), множество ребер которого состоит из m элементов. Пусть n_i-количество вершин в ребре e_i, i=1,2,...,m, причем $n_i \neq n_j$ при i≠j, i,j=1,2,...,m. Тогда количество автоморфизмов гиперграфа H равно $(n_1-k)!*(n_2-k)!*...(n_m-k)!*k!$.

Доказательство: По условию теоремы ребра гиперграфа H содержат разные количества вершин. Следовательно, всякий автоморфизм данного гиперграфа может отображать вершины произвольного ребра e гиперграфа H только в вершины этого же ребра e.

Поскольку гиперграф H=(V,E) – строго k-пересекающийся, то в гиперграфе H имеется множество вершин $X=\{v_1,v_2,...,v_k\}$ такое, что всякое ребро e содержит это множество: $(\forall e \in E)(X \subset e)$. Это означает, что эти вершины $v_1,v_2,...,v_k$ можно отображать только в вершины множества X, так как в противном случае вершины из множества X не будут смежны с вершинами других ребер, что противоречит определению автоморфизма. Из этого следует, что всякий автоморфизм λ гиперграфа H удовлетворяет свойству: для любого ребра $e \in E$ множество $X \subset e$ может быть отображено только на множество X, а множество вершин e\X – на множество e\X.

Рассмотрим ребро e_1 гиперграфа H. Это ребро содержит n_1 вершин. Таким образом, множество $e_1\setminus\{v_1,v_2,...,v_k\}$ должно отображаться только на это же множество. Это множество состоит из (n_1-k) вершин. Ясно, что количество таких отображений совпадает с количеством перестановок данного множества. Действительно, переставляя первую вершину в этом ребре, у нас имеется (n_1-k) вариантов, переставляя вторую, имеем (n_1-k-1) вариантов, и так далее, переставляя (n_1-

k)-ю вершину, у нас останется только один вариант. Таким образом, получаем $(n_1\text{-}k)*(n_1\text{-}k\text{-}1)*\ldots*1=(n_1\text{-}k)!$ вариантов. Аналогичными рассуждениями находим число перестановок множеств $e_i\backslash\{v_1,v_2,\ldots,v_k\}$ для остальных ребер e_i, i=2,3,..,m. В результате имеем: $(n_i\text{-}k)!$.

Из теоремы 1 получаем, что число перестановок множества $\{v_1,v_2,\ldots,v_k\}$ равно k!.

Очевидно, что число допустимых перестановок вершин ребра e_i составит: $(n_i\text{-}k)!*k!$, i=1,2,…,m.

Ясно, что каждая перестановка множества вершин гиперграфа H, произведенная в соответствии с введенным правилом, определяет подстановку множества вершин V, сохраняющую смежность вершин, и, следовательно, является автоморфизмом гиперграфа H. А общее число автоморфизмов гиперграфа H определяется по формуле: $(n_1-k)!*(n_2-k)!*\ldots(n_m-k)!*k!$. Теорема доказана.

Пример 2.2. На рисунке 2.4 представлен гиперграф H, количество автоморфизмов которого можно рассчитать по теореме 2.

Данный гиперграф состоит из трех ребер: u_1,u_2,u_3, причем $n_1=5$, $n_2=6$, $n_3=4$. Гиперграф H является строго 2-пересекающимся гиперграфом, вершины под цифрами 0,10 являются общими для всех ребер: $\{0,10\}\subset u_1,\{0,10\}\subset u_2,\{0,10\}\subset u_3$, то есть k=2. Следовательно, выполняются условия теоремы 2, и мы можем рассчитать количество автоморфизмов гиперграфа H по формуле: $(n_1-k)!*(n_2-k)!*(n_3-k)!*k!=(5-2)!*(6-2)!*(4-2)!*2!=576$. Таким образом, число автоморфизмов гиперграфа H равно 576.

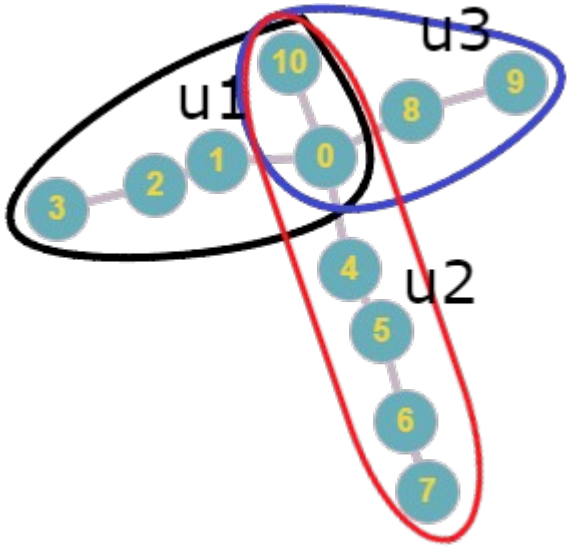

Рисунок 2.4 − Гиперграф H

2.3 Строго t-пересекающийся гиперграф с равномощными ребрами

Теорема 3: Пусть дан строго t-пересекающийся гиперграф G=(V,E), множество ребер которого состоит из m элементов. Пусть n-количество вершин в каждом ребре e_i, i=1,2,…,m, а x – общее количество вершин в гиперграфе: |V|=x. Тогда количество автоморфизмов гиперграфа G равно

$$\left(\prod_{k=0}^{m-1}(x-t-k*(n-t))*((n-t-1)!)^m\right)*t!.$$

Доказательство: По условию теоремы ребра гиперграфа G содержат одинаковое количество вершин. Следовательно, всякий автоморфизм данного гиперграфа может отображать множество вершин произвольного ребра e гиперграфа G на множество вершин любого ребра этого гиперграфа.

Поскольку гиперграф G=(V,E) – строго t-пересекающийся, то в гиперграфе H имеется множество вершин $X=\{v_1,v_2,\ldots,v_t\}$ такое, что всякое ребро e содержит это множество: $(\forall e\in E)(X\subset e)$. Это означает, что множество вершин $v_1,v_2,\ldots,v_t$ можно

отображать только на множество X, т.е. переставлять их, так как в противном случае вершины из множества X не будут смежны с вершинами других ребер, что противоречит определению автоморфизма. Из этого следует, что всякий автоморфизм λ гиперграфа H удовлетворяет свойству: для любого ребра $e \in E$ множество $X \subset e$ может быть отображено только на множество X, а множество вершин e\X – в множество V\X (при сохранении смежности). Число перестановок множества X равно t!.

Опишем идею определения числа автоморфизмов гиперграфа G. Рассмотрим произвольное ребро e гиперграфа G. Это ребро содержит n вершин. Множество $e\{v_1,v_2,\ldots,v_t\}$ может отображаться на любое множество $e_i\X$, i=1,2,…,m. Каждое такое множество состоит из (n-t) вершин. Переставляя первую вершину ребра e, у нас имеется (x-t) вариантов (x-t =|V\X|), однако переставляя вторую, имеем (n-t-1) вариантов (поскольку оставшиеся вершины ребра e могут быть отображены только на вершины ребра, которому принадлежит образ первой вершины), переставляя третью, имеем (n-t-2) вариантов и так далее, переставляя (n-t)-ю вершину, у нас останется только один вариант. Таким образом, получаем (x-t)*(n-t-1)* (n-t-2)….*1=(x-t)*(n-t-1)! вариантов. Далее, рассматривая другие ребра гиперграфа H, для отображения первой вершины имеем (x-t-k*(n-t)) вариантов, где k – количество ребер уже рассмотренных ранее. Переставляя вторую вершину ребра, имеем (n-t-1) вариантов, переставляя третью, имеем (n-t-2) вариантов и так далее, переставляя (n-t)-ю вершину, у нас останется только один вариант. Таким образом, получаем (x-t-k*(n-t))*(n-t-1)*(n-t-2)….*1=(x-t-k*(n-t))*(n-

t-1)! вариантов для k+1 ребра (k=0,1,..,m-1). Отметим также, что порядок рассмотрения ребер не имеет значения при определении количества автоморфизмов.

Таким образом, общее число автоморфизмов гиперграфа G определяется по формуле: $\left(\prod_{k=0}^{m-1}(x-t-k*(n-t))*((n-t-1)!)^m\right)*t!$.

Пример 2.3. На рисунке 2.5 представлен гиперграф G, количество автоморфизмов которого можно рассчитать по теореме 3. Множество вершин состоит из 13 элементов: x=13.

Множество ребер данного гиперграфа состоит из четырех ребер: u_1, u_2, u_3, u_4 мощности 4, следовательно, m=4, n=4. Гиперграф G является строго 1-пересекающимся гиперграфом, поскольку вершина 12 является общей для всех ребер: $\{12\}\subset u_1, \{12\}\subset u_2, \{12\}\subset u_3 \{0,10\}\subset u_1, \{0,10\}\subset u_2, \{0,10\}\subset u_3, \{12\}\subset u_4$ $\{0,10\}\subset u_1, \{0,10\}\subset u_2, \{0,10\}\subset u_3$, то есть t=1. Значит, выполняются условия теоремы 3, и мы можем рассчитать количество автоморфизмов гиперграфа G по формуле: $\prod_{k=0}^{3}(13-1-k*(4-1))*((4-1-1)!)^4*1!=(12*(12-3)*(12-6)*(12-9)*(3-1)^4)*1!=12*9*6*$. Таким образом, число автоморфизмов гиперграфа H равно 41472.

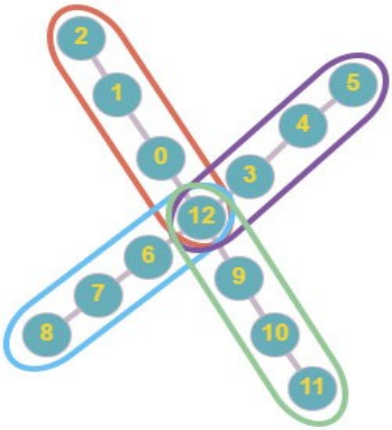

Рис 2.5 – Гиперграф G

3.Описание программного продукта

В данном разделе описывается учебное приложение, предназначенное для подсчета числа автоморфизмов гиперграфов на основе выведенных формул.

3.1. Требования, предъявляемые к программному продукту

Создаваемый программный продукт должен удовлетворять следующим требованиям.

– Иметь простое и понятное меню.

– Иметь возможность менять путь к файлу входных данных.

– Иметь возможность менять путь к файлу результатов.

– Выводить ошибку если путь к файлу был указан некорректно.

– Иметь возможность загружать гиперграф в виде матрицы инцидентности из файла.

– Автоматически определять размерность матрицы инцидентности для загружаемого гиперграфа.

– Выводить ошибку, если попытаться рассчитать количество автоморфизмов, не загрузив при этом гиперграф.

– Определять, к какому из двух рассматриваемых классов принадлежит загруженный гиперграф.

– Иметь возможность производить расчет количества автоморфизмов гиперграфа по теореме 2.

– Иметь возможность производить расчет количества автоморфизмов гиперграфа по теореме 3.

– Выдавать ошибку, если попытаться рассчитать количество автоморфизмов гиперграфа по формуле для другого класса гиперграфов.

– Иметь возможность выводить все автоморфизмы гиперграфа в файл результатов.

3.2. Описание архитектуры приложения

Блок – схема, описывающая работу приложения, представлена на рисунке 3.1. Алгоритмы, используемые в программе, требуют вычисления факториалов и, следовательно, имеют факториальную сложность.

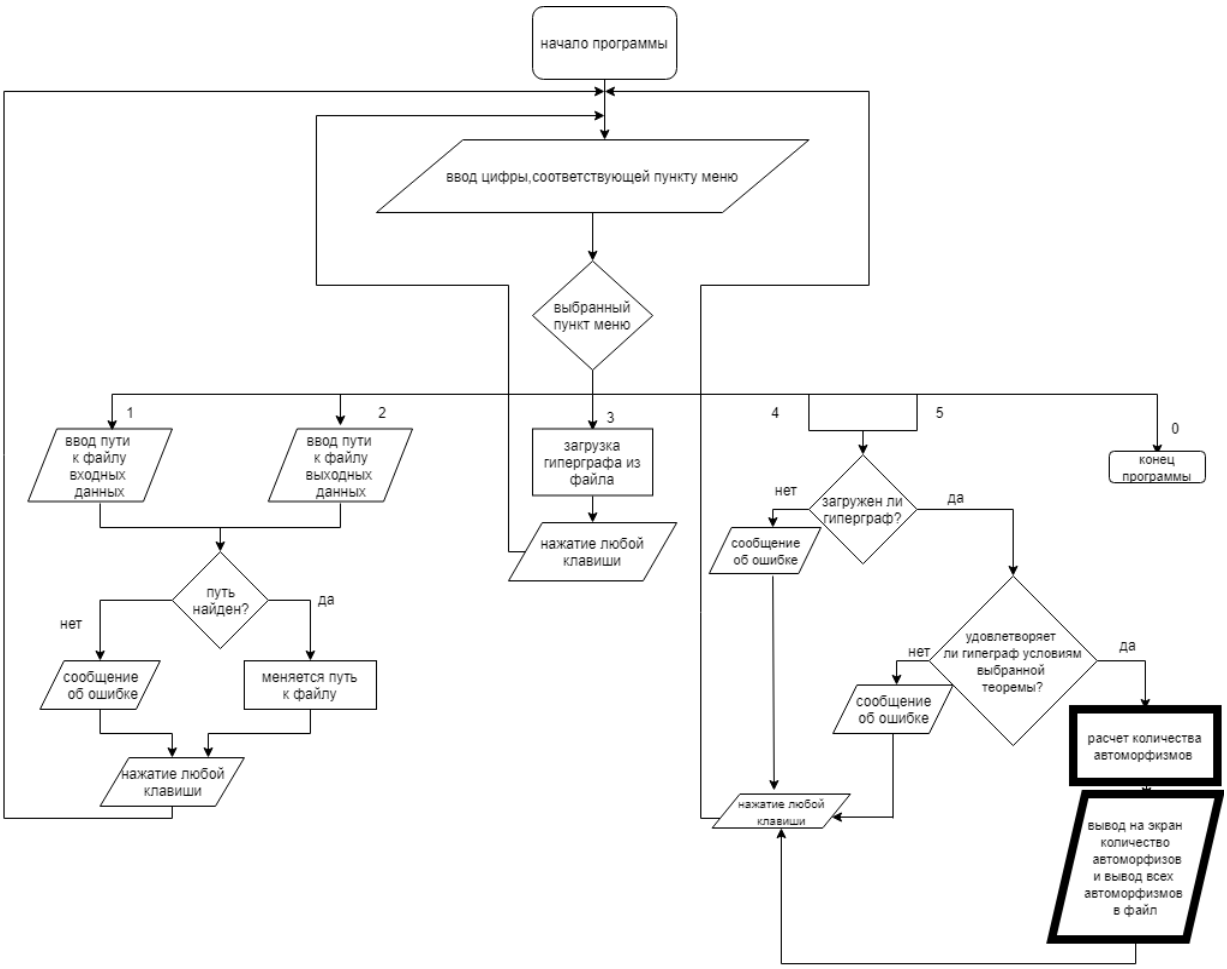

Рисунок 3.1 – Блок – схема алгоритмов программы

Вначале пользователю предлагается выбрать номер пункта меню. Ветви 1 и 2 описывают действия, связанные с изменением пути к файлам с входными и выходными данными, пункт 3 отвечает за загрузку матрицы инцидентности, ветви 4 и 5 содержат проверку принадлежности гиперграфа рассматриваемым в работе классам. Если ответ отрицательный, то выдается ошибка и сообщение о том, что нужно рассчитывать количество автоморфизмов по другой теореме. Основная часть программы, отвечающая за расчет формул и формирование множества автоморфизмов согласно теоремам 2 и 3, содержится в блоках, выделенных жирной рамкой. Логические схемы алгоритмов генерации автоморфизмов представлены на рисунках 3.2-3.3. Рассчитанное количество

автоморфизмов гиперграфа выводится на экран, а после этого количество автоморфизмов и все автоморфизмы записываются в файл выходных данных.

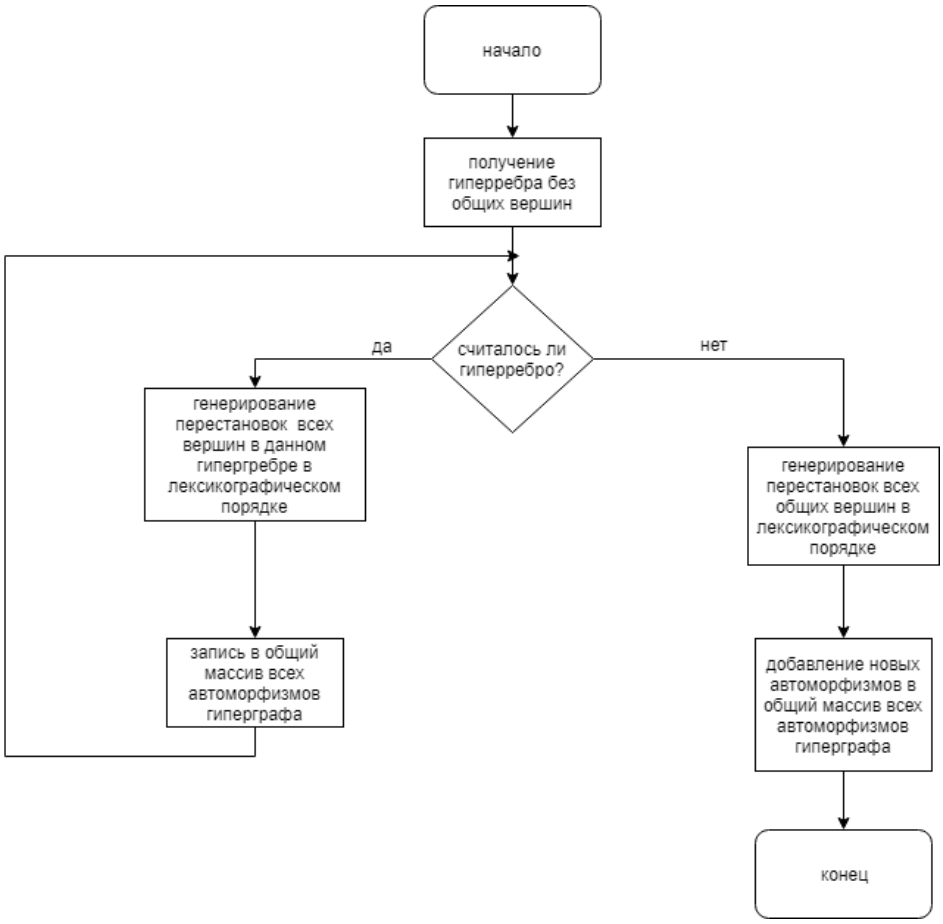

Рисунок 3.2 – Блок – схема алгоритма генерации автоморфизмов гиперграфа с использованием теоремы 2

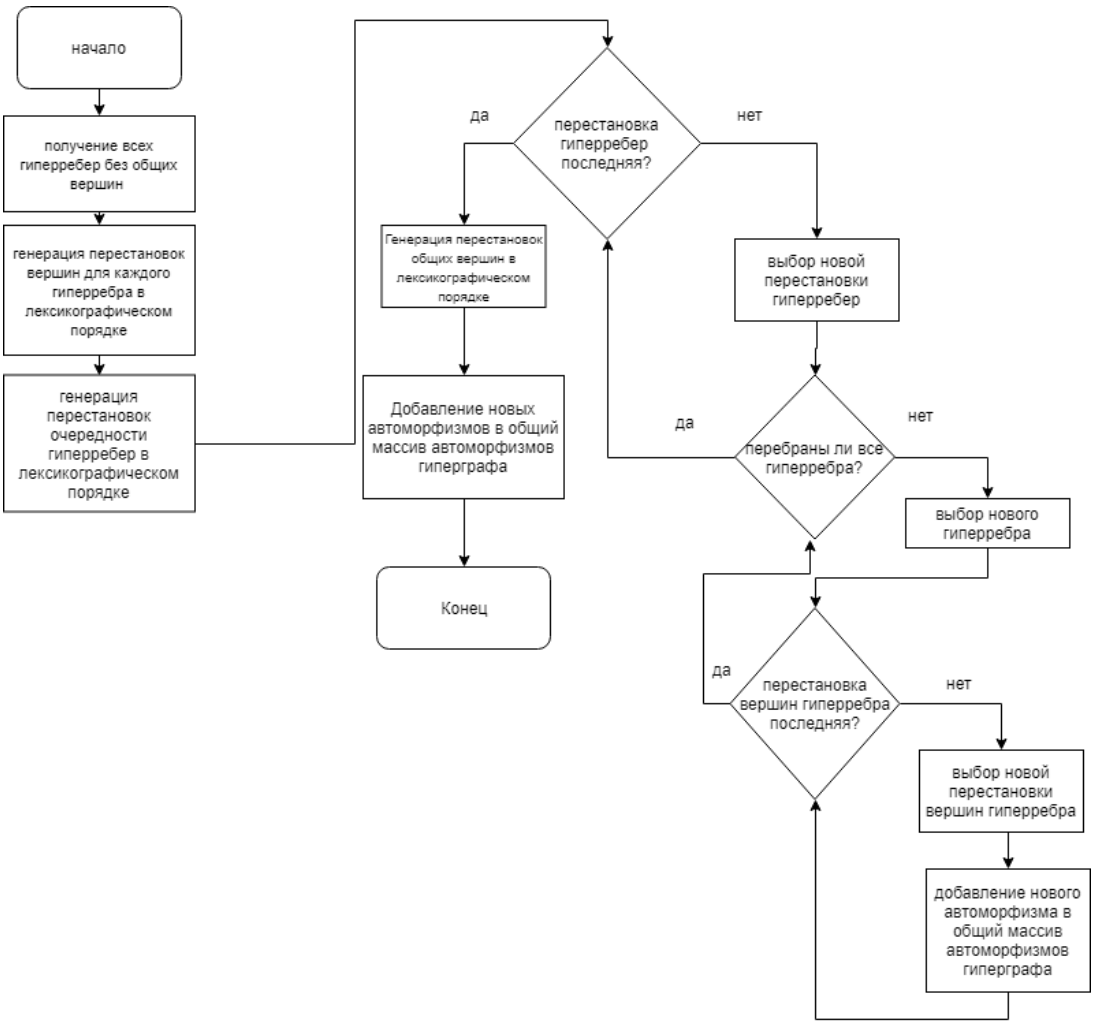

Рисунок 3.3 – Блок – схема алгоритма генерации автоморфизмов гиперграфа с использованием теоремы 3

Данная программа была реализована на объектно-ориентированном языке программирования C# в среде разработки “Visual Studio 2017” [11-13] и содержит 6 классов. Диаграмма классов приложения представлена на рисунке 3.4.

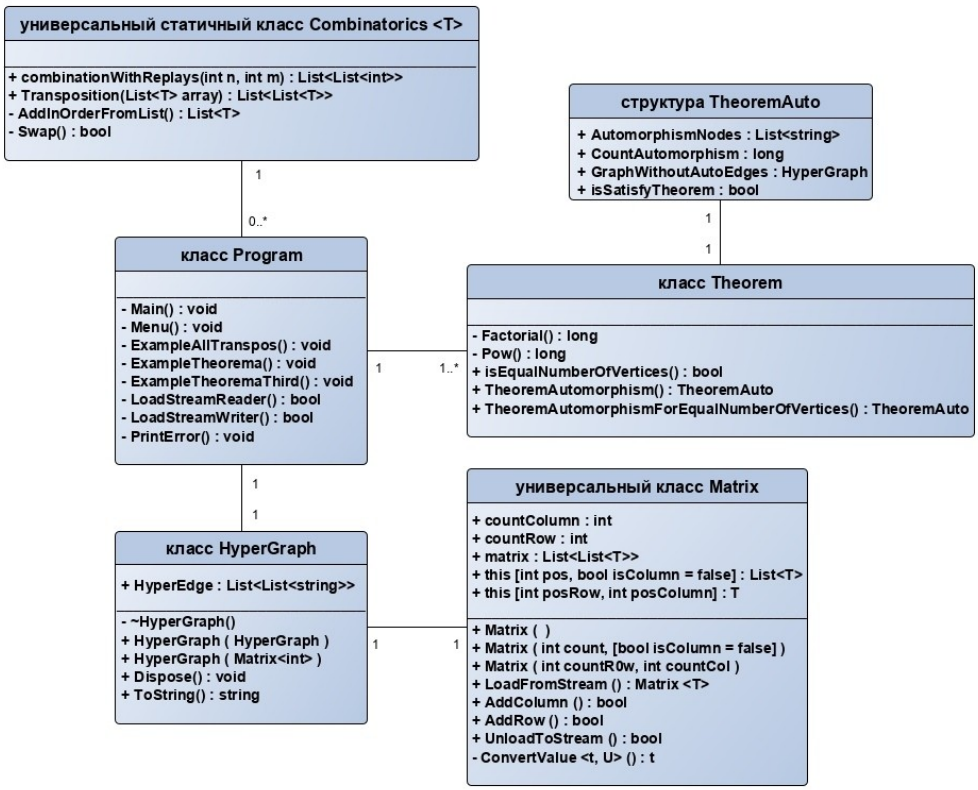

Рисунок 3.4 – Диаграмма классов

Приложение состоит из 5 файлов: Combinatorics.cs, HyperGraph.cs, Matrix.cs, Program.cs, Theorem.cs. Кратко охарактеризуем каждый из них.

Файл Combinatorics.cs

Файл: Combinatorics.cs содержит в себе класс Combinatorics. Данный класс является статичным и относится к generic-классам, включает в себя 4 метода.

Метод Transposition

Метод Transposition имеет область видимости public; принимает список типа T (List <T>); возвращает “матрицу” типа T (List<List<T>>). В качестве аргумента, метод принимает список (массив), содержащий в себе множество элементов типа T. В качестве результата создается

многомерный список, где каждый список – это перестановка входного множества.

Метод AddInOrderFromList

Метод AddInOrderFromList имеет область видимости private; принимает список типа T (List <T>) и массив целых чисел; возвращает список типа T (List <T>). В качестве первого аргумента, метод принимает список, содержащий в себе множество элементов типа T, а второй аргумент - массив, который задает порядок элементов множества. В качестве результата создается одна из перестановок исходного множества.

Метод Swap

Метод Swap имеет область видимости private; принимает два аргумента типа T по ссылке(ref). Возвращает true, если перестановка элементов произошла, в противном случае false.

Метод combinationWithReplays

Метод combinationWithReplays имеет область видимости public; принимает два значения типа int, где первое является мощностью множества элементов, а второе по сколько элементов нужно сочетать выбранную комбинацию. Возвращает “матрицу” типа int (List<List<int>>). В качестве результата создается многомерный список, где каждый список – это сочетания с повторениями целочисленного множества.

Файл HyperGraph.cs

Файл: HyperGraph.cs содержит в себе класс HyperGraph. Данный класс содержит в себе одно public поле (HyperEdge), которое является списком списков строк (List<List<string>>), два конструктора и деструктор.

Поле HyperEdge представляет собой двумерную матрицу инцидентности, где строкам соответствуют гиперребра, а столбцам − вершины гиперграфа.

Конструктор HyperGraph, принимает на вход экземпляр класса Matrix типа int. Экземпляр класса Matrix представляет собой двумерную матрицу инцидентности.

Конструктор HyperGraph, принимает на вход экземпляр класса HyperGraph.

Деструктор ~HyperGraph и метод Dispose производят принудительную очистку памяти.

Файл Matrix.cs

Файл: Matrix.cs содержит в себе класс Matrix. Данный класс относится к generic-классам, так как может принимать элемент любого типа. Matrix описывает двумерную матрицу и методы работы с ней.

Файл Theorem.cs

Файл Theorem.cs содержит в себе класс Theorem. Данный класс содержит в себе функционал работы с теоремами, которые были доказаны в данной работе. Внутри данного класса определенна структура TheoremAuto, содержащая в себе:

- логическую переменную, показывающую является ли гиперграф строго k-пересекающимся;
- количество автоморфизмов;
- вершины, которые являются общими для каждого ребра;
- гиперграф, без общих вершин.

Класс Theorem содержит в себе 5 методов.

Метод TheoremAutomorphism

Метод TheoremAutomorphism имеет область видимости public. Принимает на вход экземпляр класса HyperGraph. Возвращает экземпляр структуры TheoremAuto. В качестве аргумента передается экземпляр гиперграфа, а результатом метода является структура, содержащая в себе количество автоморфизмов, вершины, которые являются общими для каждого гиперребра и гипергграф, без общих вершин.

Метод TheoremAutomorphismForEqualNumberOfVertices

Данный метод имеет область видимости public. Принимает на вход экземпляр класса HyperGraph. Результатом метода является структура, содержащая в себе количество автоморфизмов согласно теореме 3, вершины, которые являются общими для каждого гиперребра и гипергграф, без общих вершин.

Метод isEqualNumberOfVertices

Данный метод имеет область видимости public. Принимает на вход экземпляр класса HyperGraph. Возвращает булево значение, являются ли все гиперребра в гиперграфе одинаковой мощности или нет.

Метод Factorial

Метод Factorial имеет область видимости private. Принимает на вход значение типа int. Возвращает значение типа int. В данном методе происходит расчет факториала числа, которое подается на вход.

Метод Pow

Метод Pow имеет область видимости private. Принимает на вход два значения типа int. Возвращает значение типа int. В данном методе происходит возведение в степень числа, которое подается на вход.

Файл Program.cs

Файл Program.cs содержит в себе класс program. Данный класс осуществляет выполнение функции Main.

Функция Main. Выполняет загрузку матрицы из файла, создает экземпляр гиперграфа на основе матрицы, находит все общие вершины, выводит перестановки оставшихся вершин и считает число автоморфизмов гиперграфа, а результат работы выводится в файл.

Полный листинг программы представлен в Приложении А.

3.3 Описание интерфейса приложения

Данная программа имеет простое меню, поэтому реализована в виде консольного приложения. Меню программы представлено на рисунке 3.5. В верхней части окна приложения показаны полные имена файла с входными данными (input.txt) и файла с результатами (output.txt). С помощью меню пользователь может выполнить загрузку гиперграфа на основе матрицы инцидентности из файла, изменить путь к файлам выходных данных и результатов. Кроме того, здесь же можно провести проверку принадлежности вводимого гиперграфа к классам гиперграфов, к которым применимы теоремы 2,3, и рассчитать число автоморфизмов для введенного гиперграфа на основе

данных теорем, в результате чего все автоморфизмы гиперграфа будут выведены в файл формата txt.

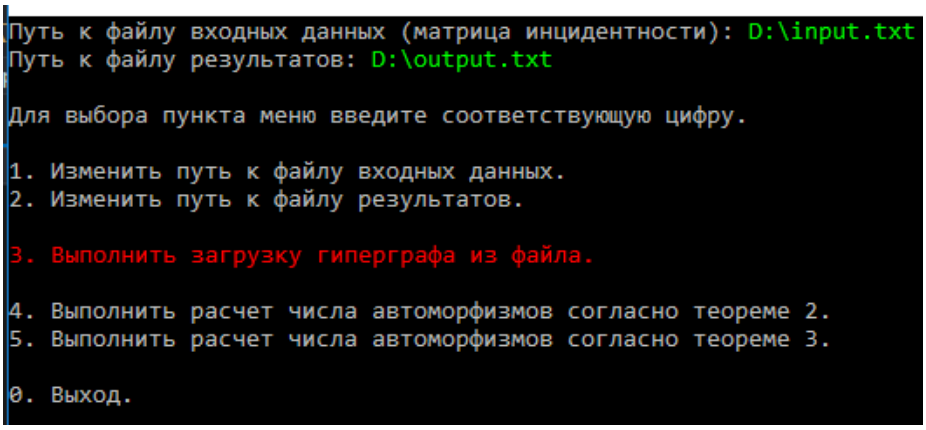

Рисунок 3.5 – Меню программы

4. Результат работы программы. Тестовые примеры

Рассмотрим работу программы на примерах.

4.1. Тестовый пример строго k-пересекающегося гиперграфа с ребрами разной мощности

Пример 4.1. Рассмотрим гиперграф, который был рассмотрен в примере 2.2. Этот гиперграф удовлетворяет условиям теоремы 2. Данный гиперграф представлен на рисунке 4.1

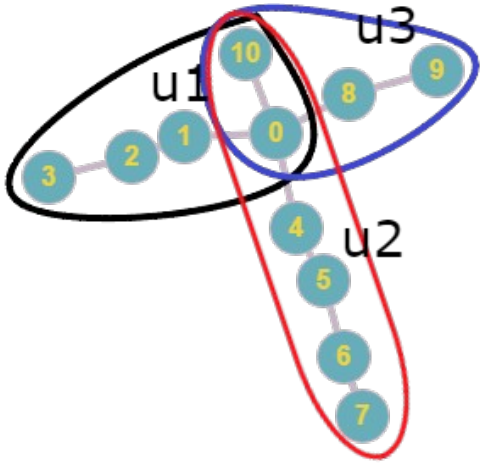

Рисунок 4.1 – Гиперграф H

Таблица 3 – Матрица инцидентности гиперграфа H

	u1	u2	u3
X0	1	1	1
X1	1	0	0
X2	1	0	0
X3	1	0	0
X4	0	1	0
X5	0	1	0
X6	0	1	0
X7	0	1	0
X8	0	0	1
X9	0	0	1

X10	1	1	1

Тогда файл input.txt выглядит:

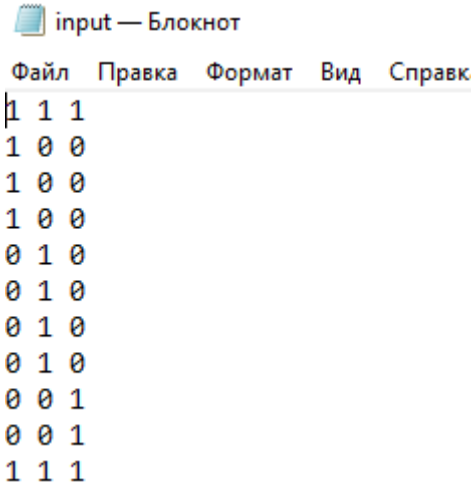

Рисунок 4.2 – Файл input.txt

При запуске программы появляется меню, которое представлено на рисунке 4.3

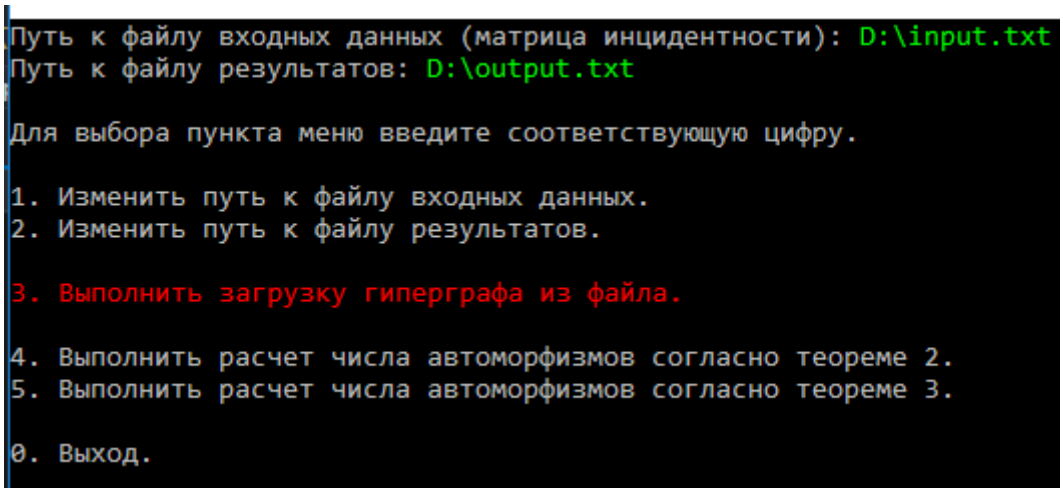

Рисунок 4.3 – Меню программы

Необходимо сделать загрузку гиперграфа из файла, иначе будет выдаваться ошибка о том, что гиперграф не загружен. Пример ошибки представлен на рисунке 4.4

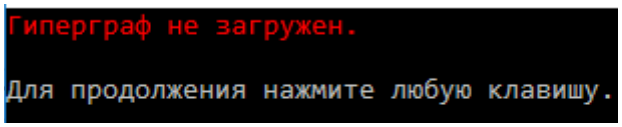

Рисунок 4.4 – Пример ошибки

После успешной загрузки гиперграфа и в результате выполнения 4 пункта меню выводится сообщение: «Принадлежность Теореме 2: True» и количество автоморфизмов данного гиперграфа – 576 (см. пример 2, глава 2). Все общие вершины, количество автоморфизмов и все перестановки гиперграфа записываются в файл «output.txt» Результат работы программы при выполнении 4 пункта меню представлен на рисунках 4.5-4.7

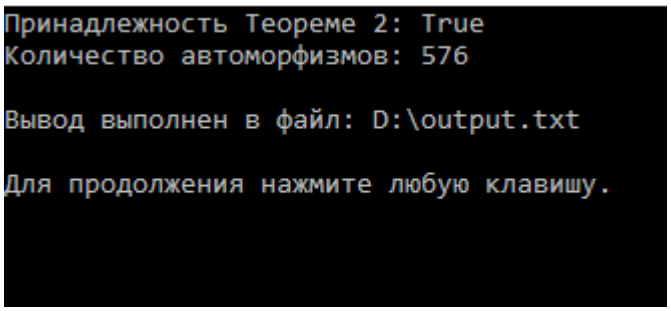

Рисунок 4.5 – Результат работы программы при выполнении 4 пункта меню

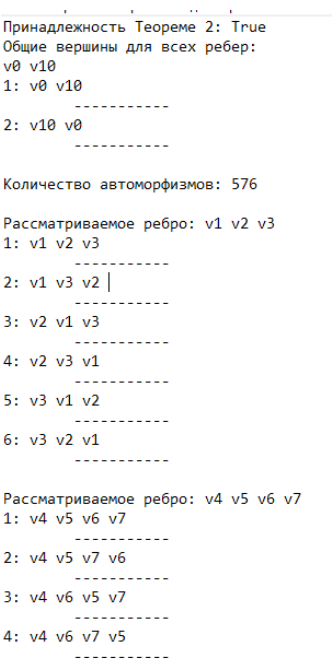

Рисунок 4.6 – Файл output.txt

```
5: v4 v7 v5 v6
        -----------
6: v4 v7 v6 v5
        -----------
7: v5 v4 v6 v7
        -----------
8: v5 v4 v7 v6
        -----------
9: v5 v6 v4 v7
        -----------
10: v5 v6 v7 v4
        -----------
11: v5 v7 v4 v6
        -----------
12: v5 v7 v6 v4
        -----------
13: v6 v4 v5 v7
        -----------
14: v6 v4 v7 v5
        -----------
15: v6 v5 v4 v7
        -----------
16: v6 v5 v7 v4
        -----------
17: v6 v7 v4 v5
        -----------
18: v6 v7 v5 v4
        -----------
19: v7 v4 v5 v6
        -----------
20: v7 v4 v6 v5
        -----------
21: v7 v5 v4 v6
        -----------
22: v7 v5 v6 v4
        -----------
23: v7 v6 v4 v5
        -----------
24: v7 v6 v5 v4
        -----------

Рассматриваемое ребро: v8 v9
1: v8 v9
        -----------
2: v9 v8
        -----------
```

Рисунок 4.7 – Файл output.txt

При выполнении пункта 5 меню для данного примера будет выводиться сообщение: «Число автоморфизмов гиперграфа должно вычисляться по теореме 2». Эта ошибка выдается, потому что загруженный ранее гиперграф не соответствует условиям теоремы 3. Результат работы программы при выполнении 5 пункта меню представлен на рисунке 4.8.

```
Число автоморфизмов графа должно вычисляться по теореме 2.

Для продолжения нажмите любую клавишу.
```

Рисунок 4.8 – Результат работы программы при выполнении 5 пункта меню

4.2. Тестовый пример строго k-пересекающегося гиперграфа с ребрами одинаковой мощности

Пример 4.2. Возьмем гиперграф, который соответствует условиям теоремы 3. Данный гиперграф представлен на рисунке 4.9

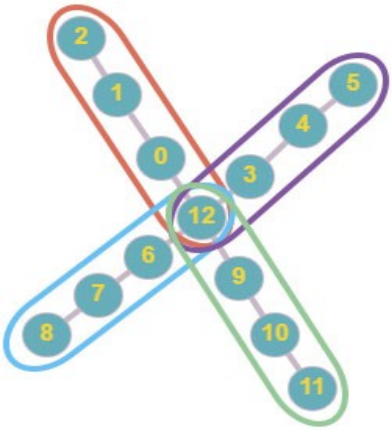

Рисунок 4.9 – Гиперграф G

Таблица 4 – Матрица инцидентности гиперграфа G

	u1	u2	u3	u4
X0	1	0	0	0
X1	1	0	0	0
X2	1	0	0	0
X3	0	1	0	0
X4	0	1	0	0
X5	0	1	0	0
X6	0	0	1	0
X7	0	0	1	0
X8	0	0	1	0
X9	0	0	0	1
X10	0	0	0	1
X11	0	0	0	1

Продолжение таблицы 4

X12	1	1	1	1

Тогда файл input.txt выглядит:

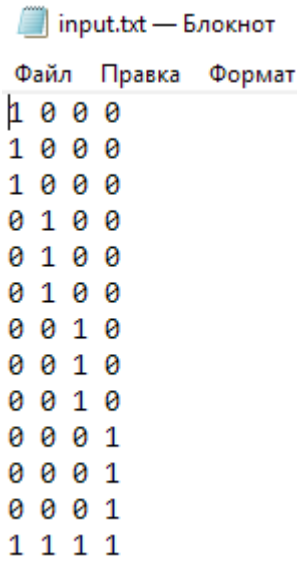

Рисунок 4.10 – Файл input.txt

После успешной загрузки гиперграфа и в результате выполнения 5 пункта меню выводится сообщение: «Принадлежность Теореме 3: True» и количество автоморфизмов данного гиперграфа. В файл «output.txt» записываются все общие вершины, количество автоморфизмов, число которых 31104 (см. пример 2.3) и все перестановки гиперграфа. Результат работы программы при выполнении 5 пункта меню представлен на рисунках 4.11-4.13.

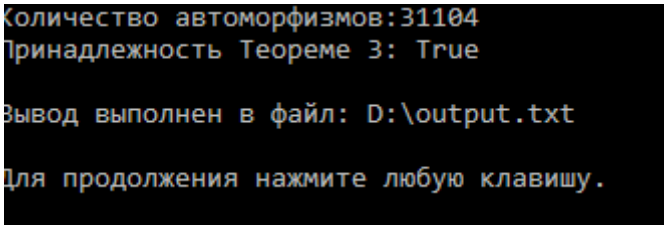

Рисунок 4.11 – Результат работы программы при выполнении 5 пункта меню

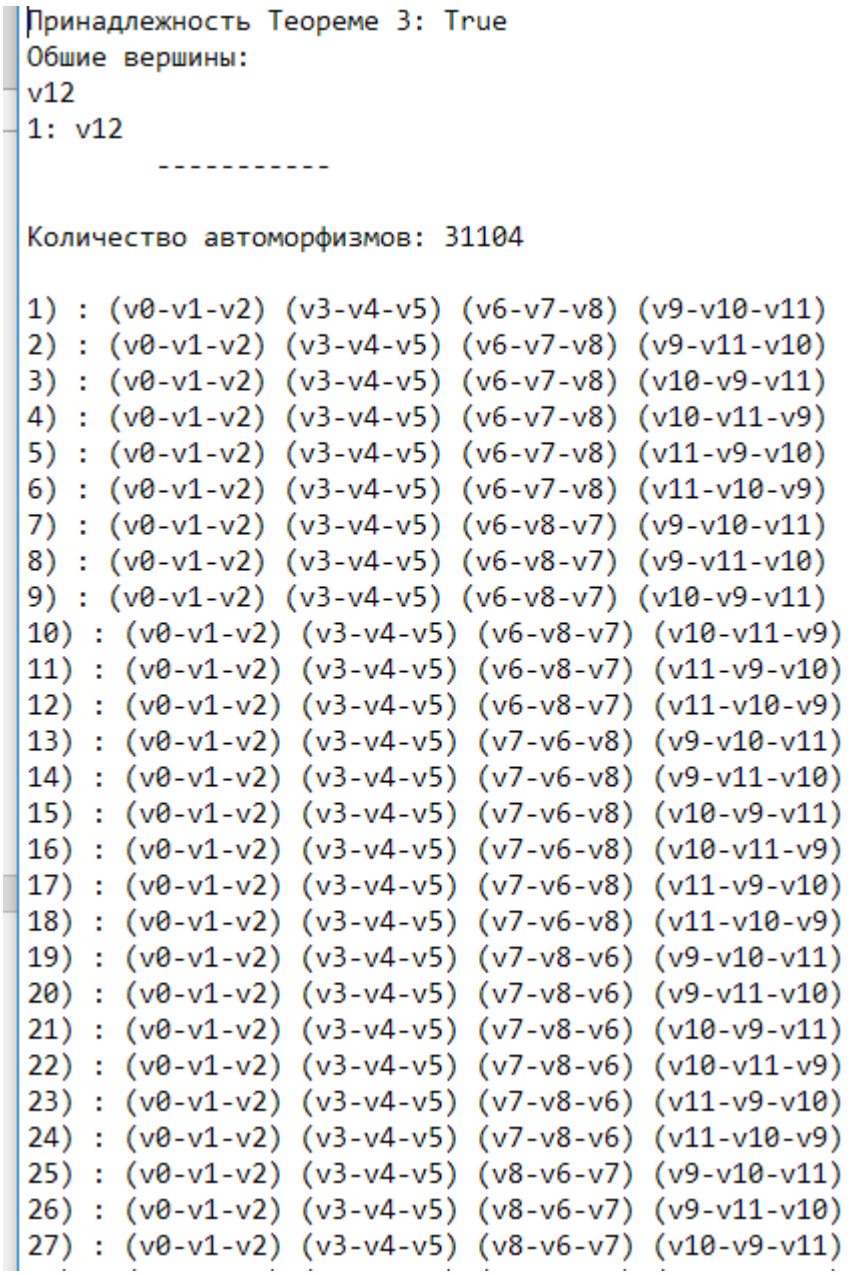

Рисунок 4.12 – Начало файла output.txt

```
31066) : (v11-v10-v9) (v8-v6-v7) (v5-v4-v3) (v1-v2-v0)
31067) : (v11-v10-v9) (v8-v6-v7) (v5-v4-v3) (v2-v0-v1)
31068) : (v11-v10-v9) (v8-v6-v7) (v5-v4-v3) (v2-v1-v0)
31069) : (v11-v10-v9) (v8-v7-v6) (v3-v4-v5) (v0-v1-v2)
31070) : (v11-v10-v9) (v8-v7-v6) (v3-v4-v5) (v0-v2-v1)
31071) : (v11-v10-v9) (v8-v7-v6) (v3-v4-v5) (v1-v0-v2)
31072) : (v11-v10-v9) (v8-v7-v6) (v3-v4-v5) (v1-v2-v0)
31073) : (v11-v10-v9) (v8-v7-v6) (v3-v4-v5) (v2-v0-v1)
31074) : (v11-v10-v9) (v8-v7-v6) (v3-v4-v5) (v2-v1-v0)
31075) : (v11-v10-v9) (v8-v7-v6) (v3-v5-v4) (v0-v1-v2)
31076) : (v11-v10-v9) (v8-v7-v6) (v3-v5-v4) (v0-v2-v1)
31077) : (v11-v10-v9) (v8-v7-v6) (v3-v5-v4) (v1-v0-v2)
31078) : (v11-v10-v9) (v8-v7-v6) (v3-v5-v4) (v1-v2-v0)
31079) : (v11-v10-v9) (v8-v7-v6) (v3-v5-v4) (v2-v0-v1)
31080) : (v11-v10-v9) (v8-v7-v6) (v3-v5-v4) (v2-v1-v0)
31081) : (v11-v10-v9) (v8-v7-v6) (v4-v3-v5) (v0-v1-v2)
31082) : (v11-v10-v9) (v8-v7-v6) (v4-v3-v5) (v0-v2-v1)
31083) : (v11-v10-v9) (v8-v7-v6) (v4-v3-v5) (v1-v0-v2)
31084) : (v11-v10-v9) (v8-v7-v6) (v4-v3-v5) (v1-v2-v0)
31085) : (v11-v10-v9) (v8-v7-v6) (v4-v3-v5) (v2-v0-v1)
31086) : (v11-v10-v9) (v8-v7-v6) (v4-v3-v5) (v2-v1-v0)
31087) : (v11-v10-v9) (v8-v7-v6) (v4-v5-v3) (v0-v1-v2)
31088) : (v11-v10-v9) (v8-v7-v6) (v4-v5-v3) (v0-v2-v1)
31089) : (v11-v10-v9) (v8-v7-v6) (v4-v5-v3) (v1-v0-v2)
31090) : (v11-v10-v9) (v8-v7-v6) (v4-v5-v3) (v1-v2-v0)
31091) : (v11-v10-v9) (v8-v7-v6) (v4-v5-v3) (v2-v0-v1)
31092) : (v11-v10-v9) (v8-v7-v6) (v4-v5-v3) (v2-v1-v0)
31093) : (v11-v10-v9) (v8-v7-v6) (v5-v3-v4) (v0-v1-v2)
31094) : (v11-v10-v9) (v8-v7-v6) (v5-v3-v4) (v0-v2-v1)
31095) : (v11-v10-v9) (v8-v7-v6) (v5-v3-v4) (v1-v0-v2)
31096) : (v11-v10-v9) (v8-v7-v6) (v5-v3-v4) (v1-v2-v0)
31097) : (v11-v10-v9) (v8-v7-v6) (v5-v3-v4) (v2-v0-v1)
31098) : (v11-v10-v9) (v8-v7-v6) (v5-v3-v4) (v2-v1-v0)
31099) : (v11-v10-v9) (v8-v7-v6) (v5-v4-v3) (v0-v1-v2)
31100) : (v11-v10-v9) (v8-v7-v6) (v5-v4-v3) (v0-v2-v1)
31101) : (v11-v10-v9) (v8-v7-v6) (v5-v4-v3) (v1-v0-v2)
31102) : (v11-v10-v9) (v8-v7-v6) (v5-v4-v3) (v1-v2-v0)
31103) : (v11-v10-v9) (v8-v7-v6) (v5-v4-v3) (v2-v0-v1)
31104) : (v11-v10-v9) (v8-v7-v6) (v5-v4-v3) (v2-v1-v0)
```

Рисунок 4.13 – Конец файла output.txt

При выполнении пункта 4 меню для данного примера будет выводиться сообщение: «Число автоморфизмов гиперграфа должно вычисляться по теореме 3». Эта ошибка выдается, потому что загруженный ранее гиперграф соответствует условиям теоремы 3, но не удовлетворяет

условию теоремы 2. Результат работы программы при выполнении 4 пункта меню представлен на рисунке 4.14

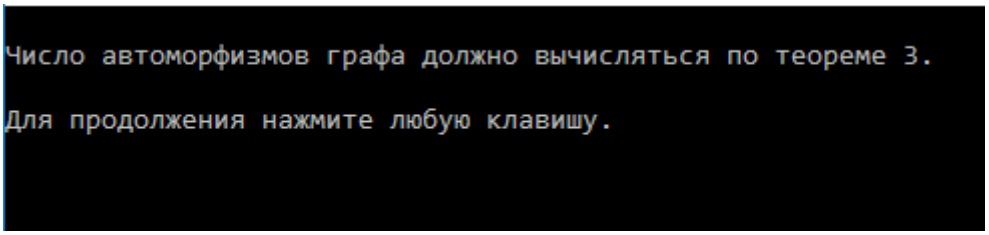

Рисунок 4.14 – Результат работы программы при выполнении 4 пункта меню

4.3 Тестовый пример нестрого k-пересекающегося гиперграфа.

Пример 4.3. Рассмотрим гиперграф, который не является строго k-пересекающимся, то есть не принадлежит ни одному из рассматриваемых классов гиперграфов. Данный гиперграф представлен на рисунке 4.15.

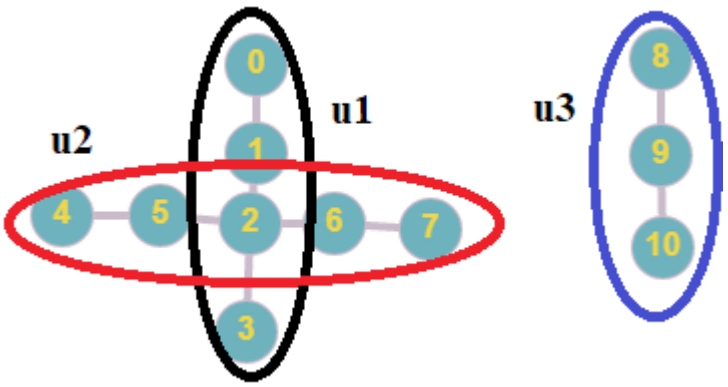

Рисунок 4.15 – Гиперграф T

Таблица 5 – Матрица инцидентности гиперграфа T

	u1	u2	u3
X0	1	1	1
X1	1	0	0

X2	1	1	0
X3	1	0	0
X4	0	1	0

Продолжение таблицы 5

X5	0	1	0
X6	0	1	0
X7	0	1	0
X8	0	0	1
X9	0	0	1
X10	0	0	1

Тогда файл input.txt выглядит:

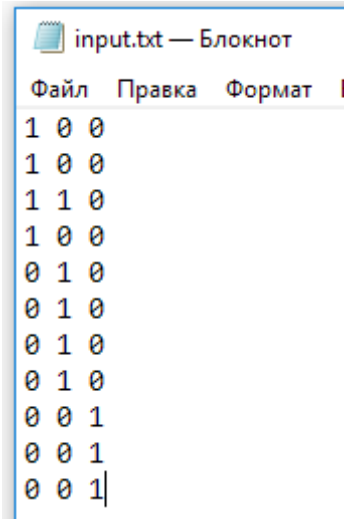

Рисунок 4.16 – Файл input.txt

После успешной загрузки гиперграфа и в результате выполнения 5 пункта меню выводится сообщение: «Принадлежность Теореме 3: False» так как этот гиперграф не соответствует условиям теоремы 3 и не является строго k - пересекающимся. Результат работы программы при выполнении 5 пункта меню представлен на рисунках 4.17 - 4.18.

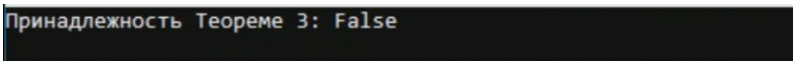

Рисунок 4.17 – Результат работы программы при выполнении 5 пункта меню

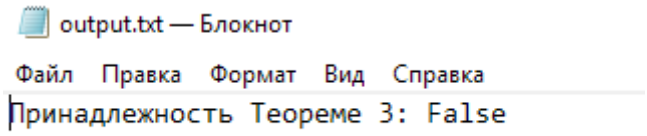

Рисунок 4.18 – Файл output.txt

При выполнении пункта 4 меню для данного примера будет выводиться сообщение: «Принадлежность Теореме 2: False», так как загруженный гиперграф не является строго k-пересекающимся. Результат работы программы при выполнении 5 пункта меню представлен на рисунках 4.19-4.20.

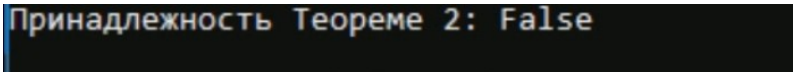

Рисунок 4.19 – Результат работы программы при выполнении 4 пункта меню

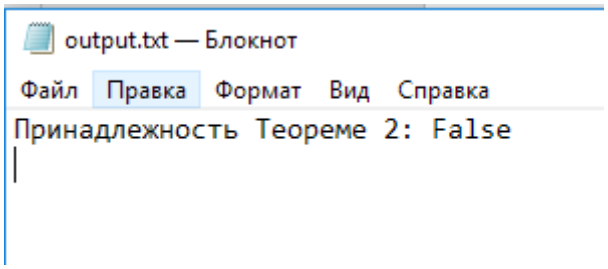

Рисунок 4.20 – Файл output.txt

ЗАКЛЮЧЕНИЕ

В последнее время наблюдается возросший интерес к изучению задач и методов теории гиперграфов. Гиперграфы нашли широкое применение в задачах управления большими системами, в логистике, при моделировании компьютерных сетей, электрических и телекоммуникационных систем и во многих других сферах. Кроме того, теорию гиперграфов изучают многие отечественные и зарубежные ученые, она является предметом их научного интереса.

В ходе выполнения выпускной квалификационной работы были применены знания по теории множеств и теории групп, были изучены основные понятия теории гиперграфов, выведены формулы для расчета количества автоморфизмов строго k-пересекающихся гиперграфов с ребрами разной мощности, а также для строго k-пересекающихся гиперграфов с ребрами одинаковой мощности, разработано приложение, предназначенное для расчета количества автоморфизмов гиперграфов этих классов. Разработанный программный продукт может быть использован в качестве учебного пособия для студентов, обучающихся на математических и технических специальностях, изучающих специальные курсы по дисциплине «Дискретная математика», при изучении таких важных понятий как граф, гиперграф и их автоморфизмы.

Результаты данной работы были представлены на VI Международной юбилейной научной конференции «Проблемы управления, обработки и передачи информации», а также на XV Международной научно-практической конференции

«Проблемы управления в социально-экономических и технических системах» (работа была отмечена дипломом 2 степени), и опубликованы в сборниках трудов этих конференций [14-15].

СПИСОК ИСПОЛЬЗОВАННОЙ ЛИТЕРАТУРЫ

1. Блюмин, С. Л. Полные гиперграфы. Спектры лапласианов. Мультиагентные системы/С.Л. Блюмин// Управление большими системами. – 2010. – N 30. – С. 5–23

2. Бондарь Е.А., Жучок Ю.В. Представления моноида сильных эндоморфизмов конечных {n}-однородных гиперграфов// Фундаментальная и прикладная математика. – 2013. – Т.18, – С. 21-34

3. Хворостухина Е.В. О конкретной характеризации универсальных гиперграфических автоматов // Фундаментальная и прикладная математика. – 2008. – Т. 14. – N 7. – С. 223-231

4. Харари, Ф. Теория графов/ Ф. Харари- М.: Едиториал, 2003. – 296с

5. Хворостухина, Е.В. О гомоморфизмах полугрупп эндоморфизмов гиперграфов/Е.В. Хворостухина// Известия Саратовского университета. Математика. Механика. Информатика. – 2009. – Т. 9, – N 3. – С. 70–75

6. Судоплатов, С. В. Дискретная математика/С. В. Судоплатов [и др.]. - Новосибирск.: НГТУ, 2012. С. 21, 31, 50

7. Bretto, A. Hypergraph Theory: An Introduction / Alain Bretto. – NY: Springer, 2013. - 117 p.

8. Зыков, А. А. Гиперграфы/А. А. Зыков //Успехи математических наук. – 1974. – N 6. – С. 89–154

9. Карпов, Д.В Гиперграф/ Д.В Карпов// Теория графов. – 2017. – N1. – С. 36-39

10. Емеличев, В. А. Лекции по теории графов/В.А Емеличев [и др.]. – М.: Мир, 1970. – 161с.

11. Хейлсберг, А., Торгерсен, М., Вилтамут, С., Голд, П. Язык программирования C#. / А. Хейлсберг., М. Торгерсен, С. Вилтамут, П. Голд М.: Питер, 2012. – 784 с.

12. Шилдт, Г. Подробнее о методах и классах/Г. Шилдт// Полное руководство C# 4.0. – 2011. – N 8. – С. 209-269

13. Шарп, Д. Microsoft Visual C#. Подробное руководство/ Д. Шарп – М.: Питер, 2017. – 845с.

14. Володин, И.О. Автоморфизмы k-пересекающегося гиперграфа / И.О. Володин/ Проблемы управления, обработки и передачи информации. Сборник трудов VI Международной юбилейной научной конференции. Саратов: СГТУ, 2018.

15. Володин, И.О. Автоморфизмы k-пересекающихся гиперграфов с ребрами одинаковой длины / И.О. Володин/ Проблемы управления в социально-экономических и технических системах. Сборник трудов XV Международной научно-практической конференции. Саратов: СГТУ, 2019.

ПРИЛОЖЕНИЕ А

Листинг кода

Файл Combinatorics.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace HyperGraph
{
    static class Combinatorics<T>
    {
        /// <summary>
        /// Множество перестановок (без повторения) элементов входящего множества
        /// </summary>
        /// <param name="array">Входящее множество</param>
        /// <returns></returns>
        public static List<List<T>> Transposition(List<T> array)
        {
            List<List<T>> result = new List<List<T>>();

            int[] temp = new int[array.Count];
            int n = array.Count;
            for (int i = 0; i < n; i++)
                temp[i] = i + 1;

            while (true)
            {
                int j = n - 2;
                while (j != -1 && temp[j] >= temp[j + 1])
                    j--;
                if (j == -1)
                    break;

                int k = n - 1;
                while (temp[j] >= temp[k]) k--;

                Swap(ref temp[j], ref temp[k]);
```

```
            int l = j + 1, r = n - 1;
            while (l < r)
                Swap(ref temp[l++], ref temp[r--]);
            result.Add(AddInOrderFromList(array, temp));
        }

        return result;
    }
    /// <summary>
    /// Создает новый массив (лист) типа T, со значениями в
порядке массива order
    /// </summary>
    /// <param name="list">Лист значений</param>
    /// <param name="order">Массив порядка
значений</param>
    /// <returns></returns>
    private static List<T> AddInOrderFromList(List<T> list, int[]
order)
    {
        List<T> temp = null;

        try
        {
            temp = new List<T>();
            for (int i = 0; i < order.Length; i++)
                temp.Add(list[order[i] - 1]);
        }
        catch (Exception e)
        {
            Console.WriteLine(e.Message);
        }
        return temp;
    }

    private static bool Swap<t>(ref t a, ref t b)
    {
        try
        {
            t temp = a;
            a = b;
            b = temp;
        }
        catch (Exception e )
        {
```

```
            Console.WriteLine(e.Message);
            return false;
        }
        return true;
    }

  }
}
```

```
    /// <summary>
    /// Маска, позволяющая перебрать множество сочетаний элементов с повторениями
    /// </summary>
    /// <param name="n">Мощность множества элементов</param>
    /// <param name="m">По скольку элементов нужно сочетать выборку</param>
    /// <returns>Вернет двумерный список размера n^m, где каждый список имеет размер m</returns>
    public static List<List<int>> combinationWithReplays(int n, int m)
    {
        List<List<int>> test2 = new List<List<int>>();
        bool NextSet(int[] aa, int nn, int mm)
        {
            int j = mm - 1;
            while (j >= 0 && aa[j] == nn) j--;
            if (j < 0) return false;
            if (aa[j] >= nn)
                j--;
            aa[j]++;
            if (j == mm - 1) return true;
            for (int k = j + 1; k < mm; k++)
                aa[k] = 1;
            return true;
        }
        List<int> minus(List<int> from)
        {
            for (int i = 0; i < from.Count; i++)
                from[i]--;
            return from;
        }
```

```
            int h = n > m ? n : m; // размер массива a выбирается как
max(n,m)
            int[] a = new int[h];
            for (int i = 0; i < h; i++)
                a[i] = 1;
            test2.Add(minus(a.Take(m).ToList()));
            while (NextSet(a, n, m))
                test2.Add(minus(a.Take(m).ToList()));
            return test2;
        }
    }
}
```

Файл HyperGraph.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace HyperGraph
{
    class HyperGraph
    {
        public List<List<string>> HyperEdge;

        public HyperGraph(Matrix<int> matrix)
        {
            HyperEdge = new List<List<string>>();

            //  По ребрам (столбцам)
            for (int j = 0; j < matrix.countColumn; j++)
            {
                HyperEdge.Add(new List<string>());
                //  По вершинам (строкам)
                for (int i = 0; i < matrix.countRow; i++)
                {
                    if (matrix[i][j] == 1)
                        HyperEdge[j].Add("v" + i);
                }
            }
        }
```

```
        //  Конструктор копирования
        public HyperGraph(HyperGraph graph)
        {
            HyperEdge = new List<List<string>>();

            foreach (var edge in graph.HyperEdge)
            {
                HyperEdge.Add(new List<string>());
                foreach (var elem in edge)
                    HyperEdge.Last().Add(elem);
            }
            Console.WriteLine();
        }

        //  При уничтожении
        ~HyperGraph()
        {
            this.Dispose();
        }

        //  Очистка памяти
        public void Dispose()
        {
            if (HyperEdge != null)
            {
                for (int i = 0; i < HyperEdge.Count; i++)
                    HyperEdge[i] = null;
                HyperEdge = null;
            }
            //  Так делать не хорошо.
            //  Принудительный вызов сборщика мусора после
удаления ссылок на списки данных.
            //  Очистит теперь неиспользуемую память.
            GC.Collect();
            GC.WaitForPendingFinalizers();
        }
    }
}
```

Файл Matrix.cs

```
using System;
using System.Collections.Generic;
```

```
using System.IO;
using System.Linq;
using System.Text;

namespace HyperGraph
{
   class Matrix<T>
   {
      public List<List<T>> matrix { get; }

      public int countRow { get; private set; }

      public int countColumn { get; private set; }

      public Matrix()
      {
         matrix = new List<List<T>>();
      }

      public Matrix(int count, bool isColumn = false)
      {
         matrix = new List<List<T>>();
         if (!isColumn)
         {
            countRow = 1;
            countColumn = count;
            matrix.Add(new List<T>());
            for (int i = 0; i < count; i++)
               matrix[0].Add(default (T));
         }
         else
         {
            countRow = count;
            countColumn = 1;
            for (int i = 0; i < count; i++)
            {
               matrix.Add(new List<T>());
               matrix[i].Add(default(T));
            }
         }
      }

      public Matrix(int countR0w, int countCol)
      {
```

```
    matrix = new List<List<T>>();
    countRow = countR0w;
    countColumn = countCol;
    for (int i = 0; i < countRow; i++)
    {
        matrix.Add(new List<T>());
        for (int j = 0; j < countColumn; j++)
            matrix[i].Add(default(T));
    }
}

public bool AddRow()
{
    try
    {
        countRow++;
        matrix.Add(new List<T>());
        int i = matrix.Count - 1;
        for (int j = 0; j < countColumn; j++)
            matrix[i].Add(default(T));
    }
    catch (Exception e)
    {
        Console.WriteLine(e.Message);
        return false;
    }
    return true;
}

public bool AddColumn()
{
    try
    {
        countColumn++;
        for (int i = 0; i < countRow; i++)
            matrix[i].Add(default(T));
    }
    catch (Exception e)
    {
        Console.WriteLine(e.Message);
        return false;
    }
    return true;
}
```

```
        // Индексатор
        public T this [int posRow, int posColumn]
        {
            get
            {
                try
                {
                    return matrix[posRow][posColumn];
                }
                catch (Exception e)
                {
                    Console.WriteLine(e.Message);
                    return default(T);
                }
            }
            set
            {
                try
                {
                    this.matrix[posRow][posColumn] = value;
                }
                catch (Exception e)
                {
                    Console.WriteLine(e.Message);
                }
            }

        }
        /// <summary>
        /// Возвращает/Устанавливает List<typeparamref name="T"/
>, содержащий строку (isColumn == false) или столбец
(isColumn == true)
        /// </summary>
        /// <param name="pos">Номер обрабатываемого
элемента</param>
        /// <param name="isColumn">Обрабатывать столбец (true)
или строку (false)</param>
        /// <returns></returns>
        public List<T> this [int pos, bool isColumn = false]
        {
            get
            {
                try
```

```
            {
                if (!isColumn)
                    return new List<T>(matrix[pos]);
                else
                {
                    List<T> result = new List<T>();
                    for (int i = 0; i < countRow; i++)
                        result.Add(matrix[i][pos]);
                    return result;
                }
            }
            catch (Exception e)
            {
                Console.WriteLine(e.Message);
                return null;
            }
        }
        set
        {
            try
            {
                if (!isColumn)
                    matrix[pos] = new List<T>(value);
                else
                {
                    List<T> input = new List<T>(value);
                    for (int i = 0; i < countRow; i++)
                        matrix[i][pos] = input[i];
                }
            }
            catch (Exception e)
            {

            }
        }
    }

    static t ConvertValue<t, U>(U value) where U : IConvertible
    {
        return (t)Convert.ChangeType(value, typeof(t));
    }

    public static Matrix<T>
LoadFromStream(System.IO.StreamReader input)
```

```
        {
            Matrix<T> matrix = new Matrix<T>(0, 0);
            int countCol;
            string r0w;
            try
            {
                //  Поток существует?
                if (input == null)
                    throw new Exception();
                //  Позиция чтения с начала файла
                input.BaseStream.Position = 0;
                //  Получаем первую строку, чтобы узнать количество
столбцов
                r0w = input.ReadLine();
                string[] temp = r0w.Split(' ');
                countCol = temp.Length;

                //  Пересоздаем матрицу, т.к. получили количество
столбцов
                matrix = new Matrix<T>(1, countCol);

                //  Заполнение первой строки
                for (int j = 0; j < countCol; j++)
                {
                    T tmp = ConvertValue<T, string>(temp[j]);
                    matrix[0, j] = tmp;
                }

                //  Перебираем все оставшиеся строки
                while(!input.EndOfStream)
                {
                    r0w = input.ReadLine();
                    temp = r0w.Split(' ');
                    //  Добавление новой строки в матрицу
                    matrix.AddRow();
                    for (int j = 0; j < countCol; j++)
                    {
                        T tmp = ConvertValue<T, string>(temp[j]);
                        matrix[matrix.countRow - 1, j] = tmp;
                    }
                }
            }
            catch (Exception e)
            {
```

```
            Console.WriteLine(e.Message);
        }

        return matrix;
    }

    public static bool UnloadToStream(Matrix<T> matrix,
System.IO.StreamWriter output)
    {
        try
        {
            if (output == null)
                throw new Exception();
            for (int i = 0; i < matrix.countRow; i++)
            {
                for (int j = 0; j < matrix.countColumn; j++)
                {
                    output.Write(matrix[i][j] + " ");
                }
                output.WriteLine();
            }
        }
        catch (Exception e)
        {
            Console.WriteLine(e.Message);
            return false;
        }
        return true;
    }
  }
}
```

Файл Theorem.cs

```
using System;
using System.Collections.Generic;
using System.IO;
using System.Linq;
using System.Text;

namespace HyperGraph
{
  class Theorem
  {
```

```
        //  Структура для результирующих данных
        public struct TheoremAuto
        {
            //  Удовлетворяет ли гиперграф теореме
            public bool isSatisfyTheorem;
            //  Число автоморфизмов
            public Int64 CountAutomorphism;
            //  Список общих вершин
            public List<string> AutomorphismNodes;
            //  Гипер-граф, содержащий гипер-ребра без общих
вершин
            public HyperGraph GraphWithoutAutoEdges;
        }

        public static TheoremAuto
TheoremAutomorphism(HyperGraph Graph)
        {
            //  Копирую гипер-граф, чтобы не изменять исходный
            HyperGraph graph = new HyperGraph(Graph);
            //  Число автоморфизмов
            Int64 count;

            //  Список вершин 0-ого гипер-ребра
            List<string> intersect = new
List<string>(graph.HyperEdge[0]);

            //  Находим пересечение множеств вершин гипер-ребер
каждого с каждым
            for (int i = 1; i < graph.HyperEdge.Count; i++)
            {
                var res = intersect.Select(j =>
j.ToString()).Intersect(graph.HyperEdge[i]);
                //  Сохраняем резудьтат пересечения для дальнейшей
операции
                intersect = new List<string>();
                foreach (var item in res)
                    intersect.Add(item);
            }
            //  В списке intersect расположены общие для всех
гипер-ребер вершины

            //  Удаляем из каждого гипер-ребра все общие вершины
            foreach (var edge in graph.HyperEdge)
                foreach (var elem in intersect)
```

```
                edge.RemoveAll(s => s.Equals(elem));

            //  Согласно формуле, подсчитываем количество
автоморфизмов
            count = Factorial(intersect.Count);     //  !k, где k =
количество общих вершин
            foreach (var edge in graph.HyperEdge)
                count *= Factorial(edge.Count);     //  *(n - k)!, где n =
количество вершин в исходном гипер-ребер,
                                                    //  a (n-k) = количество вершин в
новом гипер-графе без общих вершин

            TheoremAuto result = new TheoremAuto
            {
                isSatisfyTheorem = (intersect.Count > 0) ? true : false,
                CountAutomorphism = count,
                AutomorphismNodes = intersect,
                GraphWithoutAutoEdges = graph
            };
            //  Вовращаемое значение в виде структуры
            return result;
        }

        public static Int64
TheoremAutomorphismForEqualNumberOfVertices(HyperGraph
Graph)
        {
            //  Копирую гипер-граф, чтобы не изменять исходный
            HyperGraph graph = new HyperGraph(Graph);
            //  Число автоморфизмов
            Int64 count = 1;

            //  Список вершин 0-ого гипер-ребра
            List<string> intersect = new
List<string>(graph.HyperEdge[0]);

            //  Находим пересечение множеств вершин гипер-ребер
каждого с каждым
            for (int i = 1; i < graph.HyperEdge.Count; i++)
            {
                var res = intersect.Select(j =>
j.ToString()).Intersect(graph.HyperEdge[i]);
                //  Сохраняем резудьтат пересечения для дальнейшей
операции
```

```
            intersect = new List<string>();
            foreach (var item in res)
                intersect.Add(item);
        }
        //  В списке intersect расположены общие для всех
гипер-ребер вершины

        if (intersect.Count == 0)
            return 0;

        int m = graph.HyperEdge.Count,      //  Общее количество
ребер в гиперграфе
            n = graph.HyperEdge[0].Count,   //  Количество вершин
в каждом ребре
            t = intersect.Count,            //  Количество вершин
пересечения (t-пересекающийся гиперграф)
            x = m*(n - t) + t;              //  Общее количество вершин
в гиперграфе

        for (int k = 0; k < m; k++)
            count *= x - t - k * (n - t);
        count *= Pow(Factorial(n - t - 1), m);
        count *= Factorial(t);

        return count;
    }
    //рассчет по формуле
    public static bool isEqualNumberOfVertices(HyperGraph
Graph)
    {
        int count = Graph.HyperEdge[0].Count;
        foreach (var hyperedge in Graph.HyperEdge)
        {
            if (hyperedge.Count != count)
                return false;
        }
        return true;
    }



    private static Int64 Factorial(int input)
    {
        //  Предрассчитанные значения
        switch (input)
```

```
            {
                case 1: return 1;
                case 2: return 2;
                case 3: return 6;
                case 4: return 24;
                case 5: return 120;
                case 6: return 720;
                case 7: return 5040;
                default:
                    break;
            }
            int x;
            Int64 fact = 1;
            try
            {
                //  Вдруг выйдем за пределы максимального значения
                for (x = input; x > 1; x--)
                    fact *= x;
            }
            catch (Exception e)
            {
                Console.WriteLine("\n" + e.Message);
                Console.ReadKey();
                return -1;
            }
            return fact;
        }

        private static Int64 Pow(Int64 input, Int64 grade)
        {
            if (grade < 0)
                throw new Exception("error");
            Int64 result = 1;
            for (int i = 0; i < grade; i++)
                result *= input;
            return result;
        }
    }
}
```

Файл Program.cs

```
using System;
```

```
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace HyperGraph
{
    class Program
    {
        /* 6 7
        0 0 1 0 1 0 0
        0 0 0 0 0 1 0
        1 1 1 0 0 0 0
        0 0 0 1 0 1 1
        0 1 1 1 0 0 0
        0 0 0 0 1 0 1
        */

        /* 11 3
        1 1 1
        1 0 0
        1 0 0
        1 0 0
        0 1 0
        0 1 0
        0 1 0
        0 1 0
        0 0 1
        0 0 1
        1 1 1
```

```
        */

        /*
        1 0 0 0
        1 0 0 0
        1 0 0 0
        0 1 0 0
        0 1 0 0
        0 1 0 0
        0 0 1 0
        0 0 1 0
        0 0 1 0
        0 0 0 1
        0 0 0 1
        0 0 0 1
        1 1 1 1
        */

        static void Main(string[] args)
        {
            Menu();
        }

        //  Все перестановки графа вывести в файл
        static void ExampleAllTranspos(HyperGraph graph,
System.IO.StreamWriter output)
        {
            //  Массив (лист) всех возможных перестановок одного
множества (гипер-ребра)
            List<List<string>> transpos;
```

```
            //  Обработать все гипер-ребра
            for (int i = 0; i < graph.HyperEdge.Count; i++)
            {
                //  Получить массив перестановок текущего гипер-ребра
                transpos = Combinatorics<string>.Transposition(graph.HyperEdge[i]);

                // Вывод всех вершин текущего гипер-ребра
                output.Write("Рассматриваемое ребро:");
                for (int g = 0; g < graph.HyperEdge[i].Count; g++)
                {
                    output.Write(" {0}", graph.HyperEdge[i][g]);
                }

                //  Вывод всех перестановок текущего гипер-ребра
                output.WriteLine();

                //  Вывод начальной перестановки
                output.Write("1:");
                for (int g = 0; g < graph.HyperEdge[i].Count; g++)
                {
                    output.Write(" {0}", graph.HyperEdge[i][g]);
                }

                output.WriteLine("\r\n\t----------");
                //  Алгоритм перестановок не учитывает начальную, а начинает со следующей
                for (int g = 0; g < transpos.Count; g++)
                {
```

```
                output.Write("{0}: ", g + 2);
                for (int f = 0; f < transpos[g].Count; f++)
                {
                    output.Write("{0} ", transpos[g][f]);
                }

                output.WriteLine("\r\n\t----------");
            }
            output.WriteLine();
        }
    }

    //  Найти общие вершины для всех гипер-ребер, подсчитать количество автоморфизмов гипер-графа и вывести все перестановки оставшихся ребер
    static void ExampleTheorema(HyperGraph graph, System.IO.StreamWriter output)
    {
        //  Получить результат вычислений
        var result = Theorem.TheoremAutomorphism(graph);

        //  Если удовлетворяет условию "в каждом гиперребре одинаковое количество вершин", то нужно применять не эту теорему (2), а другую (3)
        if (Theorem.isEqualNumberOfVertices(graph))
        {
            Console.WriteLine("Принадлежность Теореме 2: " + false);
            Console.WriteLine("\nЧисло автоморфизмов графа должно вычисляться по теореме 3.");
            output.WriteLine("Принадлежность Теореме 2: " + false);
```

```
            }
            else
            {
                //  Выполняется ли условие согласно теореме 2
                Console.WriteLine("Принадлежность Теореме 2: " +
result.isSatisfyTheorem);
                if (result.isSatisfyTheorem)
                {
                    //  Вывод числа автоморфизмов в консоль
                    Console.WriteLine("Количество автоморфизмов: " +
result.CountAutomorphism);
                }
                //  Выполняется ли условие согласно теореме 2
                output.WriteLine("Принадлежность Теореме 2: " +
result.isSatisfyTheorem);
                //  Если да, то вывести
                if (result.isSatisfyTheorem)
                {
                    //  Вывод вершин и числа автоморфизмов в файл
                    output.WriteLine("Общие вершины для всех ребер:
");
                    foreach (var node in result.AutomorphismNodes)
                    {
                        output.Write(node + " ");
                    }

                    output.WriteLine();
                    //  Вывод всех перестановок общих вершин
                    List<List<string>> transpos =
Combinatorics<string>.Transposition(result.AutomorphismNodes);
                    //  Вывод начальной перестановки
```

```
                output.Write("1:");
                for (int g = 0; g < result.AutomorphismNodes.Count;
g++)
                {
                    output.Write(" {0}", result.AutomorphismNodes[g]);
                }

                output.WriteLine("\r\n\t----------");
                //  Алгоритм перестановок не учитывает начальную,
а начинает со следующей
                for (int g = 0; g < transpos.Count; g++)
                {
                    output.Write("{0}: ", g + 2);
                    for (int f = 0; f < transpos[g].Count; f++)
                    {
                        output.Write("{0} ", transpos[g][f]);
                    }

                    output.WriteLine("\r\n\t----------");
                }

                output.WriteLine("\r\nКоличество автоморфизмов: "
+ result.CountAutomorphism + "\r\n");

                //  Вывести перестановки в файл
                ExampleAllTranspos(result.GraphWithoutAutoEdges,
output);
            }
        }
        //  Уничтожить временный граф
        result.GraphWithoutAutoEdges.Dispose();
```

```
        }


        static void ExampleTheoremaThird(HyperGraph Graph,
System.IO.StreamWriter output)
        {
            //  Подсчет количества и получение гипер-графа без
общих вершин
            var result =
Theorem.TheoremAutomorphismForEqualNumberOfVertices(Grap
h);


            //  Если НЕ удовлетворяет условию "в каждом
гиперребре одинаковое количество вершин", то нужно
применять не эту теорему (3), а другую (2)
            if (!Theorem.isEqualNumberOfVertices(Graph))
            {
                Console.WriteLine("Принадлежность Теореме 3: " +
false);
                Console.WriteLine("\nЧисло автоморфизмов графа
должно вычисляться по теореме 2.");
                output.WriteLine("Принадлежность Теореме 3: " +
false);
            }
            else
            {
                //  Выполняется ли условие согласно теореме 3
                output.WriteLine("Принадлежность Теореме 3: " +
result.isSatisfyTheorem);
                if (result.isSatisfyTheorem)
                {
                    //  Вывод числа автоморфизмов в консоль
                    Console.WriteLine("Количество автоморфизмов:" +
result.CountAutomorphism);
```

```
            }
            Console.WriteLine("Принадлежность Теореме 3: " +
result.isSatisfyTheorem);
            //  Если да, то вывести
            if (result.isSatisfyTheorem)
            {
                //  Вывод вершин и числа автоморфизмов в файл
                output.WriteLine("Обшие вершины: ");
                foreach (var node in result.AutomorphismNodes)
                {
                    output.Write(node + " ");
                }

                output.WriteLine();
                //  Вывод всех перестановок общих вершин
                List<List<string>> transpos =
Combinatorics<string>.Transposition(result.AutomorphismNodes);
                //  Вывод начальной перестановки
                output.Write("1:");
                for (int g = 0; g < result.AutomorphismNodes.Count;
g++)
                {
                    output.Write(" {0}", result.AutomorphismNodes[g]);
                }

                output.WriteLine("\r\n\t----------");
                //  Алгоритм перестановок не учитывает начальную,
а начинает со следующей
                for (int g = 0; g < transpos.Count; g++)
                {
                    output.Write("{0}: ", g + 2);
```

```
                for (int f = 0; f < transpos[g].Count; f++)
                {
                    output.Write("{0} ", transpos[g][f]);
                }

                output.WriteLine("\r\n\t----------");
            }

            output.WriteLine("\r\nКоличество автоморфизмов: "
+ result.CountAutomorphism + "\r\n");

            // Получаю гипер-граф, автоморфизмы которого
нужно получить
            // Это гипер-граф, из которого удалены общие для
всех ребер вершины
            HyperGraph graph = new
HyperGraph(result.GraphWithoutAutoEdges);

            // Список со всеми автоморфизмами
            List<string> automorphism = new List<string>();

            // Трехмерный список, содержащий для каждого
ребра (aka 'список ребер') список всех возможных
транспозиций (aka перестановки, число которых n!) всех
вершин данного ребра (а вершины также заключены в список)
            // т.е. обращение comboVertInEdges[0] - список всех
возможных перестановок 0-ого гипер-ребра графа
            // comboVertInEdges[0][1] - список вершин,
входящих в 2-ю (по очередности с 0) перестановку вершин 0-ого
ребра
            // comboVertInEdges[4] - список всех возможных
перестановок 4-ого гипер-ребра графа....
            // Этот список заполняется один раз сразу для всех
ребер, т.к. содержимое ребер (вершин) неизменно
```

```
List<List<List<string>>> comboVertInEdges = new List<List<List<string>>>();


// Двумерный список - список всех возможных перестановок n-ого ребра
List<List<string>> vertInEdge;


// Количество перестановок n-ого ребра, одинаково для всех ребер (т.к. мощность каждого ребра равна)
int size = 0;


// Перебор всех ребер гипер-графа для заполнения трехмерного списка
for (int i = 0; i < graph.HyperEdge.Count; i++)
{
    // Получаем текущую перестановку
    vertInEdge = Combinatorics<string>.Transposition(graph.HyperEdge[i]);
    // Добавляем 0-ю перестановку, которую не учитывает алгоритм
    vertInEdge.Insert(0, graph.HyperEdge[i]);
    // Помещаем все перестановки ребра в общий список
    comboVertInEdges.Add(vertInEdge);
    // Запоминаем значение на будущее
    size = vertInEdge.Count;
}


// Получив все перестановки каждого из ребер, нужно сформировать автоморфизмы гипер-графа
// Учитывая порядок ребер 0,1,2,3,...,n (где n - количество ребер) и то, что для каждого ребра существует X перестановок
```

// существует X^n комбинаций перестановок гипер-графа

//

// Если два ребра по 3 вершины, то количество перестановок = 3!= 6 и есть 6*6 комбинаций этих перестановок

// Если 3 ребра по 8 вершин, то количество перестановок = 8! и есть 8! * 8! * 8! = (8!)^3 комбинаций этих перестановок

//

// В качестве аргументов передаются n='количество комбинаций' и m='количество ребер'

// Результат - двумерный список, где каждый список содержит последовательность вида

// 0 0 0 (для трех ребер), что предполагает комбинацию 0-х перестановок каждого ребра

// 0 0 1 - комбинация 0-й перестановки 0-ого и 1-ого ребра И 1-й перестановки 2-ого ребра

// 0 0 2 - комбинация 0-й перестановки 0-ого и 1-ого ребра И 2-й перестановки 2-ого ребра

// 0 1 0 - комбинация 0-й перестановки 0-ого ребра; 1-й перестановки 1-ого ребра; 0-й перестановки 2-ого ребра

// ...

var combination = Combinatorics<int>.combinationWithReplays(size, graph.HyperEdge.Count);

// Однако, выше приведенные комбинации работают лишь с перестановками непосредственно вершин, сохраняя позиции ребер

// Порядок (1-е ребро)-(2-е ребро)-(3-е ребро) остается неизменным. Меняется лишь 'порядок' внутри ребер

// Чтобы добавить движение в плоскости ребер, нужно найти все возможные перестановки (без повторений) ребер

```
            //  И применить это условие к выше приведенным
результатам


            //  Подготовка массива для создания перестановок
ребер
            List<int> tmp = new List<int>();
            for (int i = 0; i < graph.HyperEdge.Count; i++)
                tmp.Add(i);


            //  Список перестановок ребер
            //  Вида:
            //  0 1 2 (для трех ребер)
            //  0 2 1
            //  1 0 2
            //  1 2 0
            //  2 0 1
            //  2 1 0 = 3! = 6 штук
            var comboEdges =
Combinatorics<int>.Transposition(tmp);
            comboEdges.Insert(0, tmp);


            //  т.е. сначала мы выбираем комбинации
перестановок вершин ребра, учитывая 0-ю перестановку ребер
            //  потом выбираем комбинации перестановок
вершин ребра, учитывая 1-ю перестановку ребер
            //  то же самое, учитывая 2-ю перестановку ребер


            List<string> edge;  //  Одно конкретное ребро из
графа
            string fullString = "", edgeString; //  Строковые
представления
```

```
                // Перебор всех перестановок последовательностей ребер
                for (int k = 0; k < comboEdges.Count; k++)
                {
                    // Для k-й перестановки ребер перебираем все возможные комбинации перестановок вершин
                    for (int i = 0; i < combination.Count; i++)
                    {
                        // Строковое представление одного автоморфизма
                        fullString = "";
                        // Для k-й перестановки ребер и для i-й комбинации перестановок вершин
                        // Должны перебрать все гипер-ребра
                        for (int j = 0; j < combination[i].Count; j++)
                        {
                            // Строковое представление одного гипер-ребра
                            edgeString = "(";

                            // edge - это упорядоченный список вершин определенной перестановки определенного гипер-ребра
                            // comboVertInEdges - трехмерный список, где первые [] отвечают за номер гипер-ребра
                            // аргумент [ comboEdges[k][j] ] говорит, что нужно получить j-й номер гипер-ребра из k-й перестановки гипер-ребер
                            // т.е. итератор j осуществляет обработку всех гипер-ребер конкретного автоморфизма,
                            //     а итератор k указывает, в какой последовательности нужно перебирать гипер-ребра (0-1-2 или 0-2-1 или 2-0-1 или ...)
                            //
```

```
                        //  аргумент [ combination[i][j] ] говорит, что при обращении к определенному гипер-ребру нужно получить номер некоторой из возможных комбинаций вершин данного гипер-ребра
                        //  где итератор i отвечает за номер комбинации (от 0 до (8!)^3, если 3 ребра по 8 вершин)
                        //    а итератор j отвечает за номер перестановки данного гипер-ребра (от 0 до мощности ребер)
                        edge = comboVertInEdges[comboEdges[k][j]][combination[i][j]];


                        //  Получив конкретное гипер-ребро, перебираем его вершины
                        foreach (var vert in edge)
                            edgeString += vert + '-';   //  Записываем в строку
                        //  Добавляем в общую строку данного автоморфизма
                        fullString += edgeString.Remove(edgeString.Length - 1, 1) + ") ";
                    }
                    //  Записываем строковый вид автоморфизма в общий список
                    automorphism.Add(fullString);
                }
            }


            //  Вывод
            for (int i = 0; i < automorphism.Count; i++)
                output.WriteLine(i + 1 + ") : " + automorphism[i]);
        }
    }
}
```

```
        static void Menu()
        {
            string inputPath = "D:\\input.txt", outputPath = "D:\\
output.txt";
            System.IO.StreamReader input;
            System.IO.StreamWriter output;
            //  Экземпляр гипер-графа, созданный на основе
загружаемой матрицы инцидентности
            HyperGraph graph = null;


            while (true)
            {
                Console.Write("Путь к файлу входных данных (матрица
инцидентности): ");
                Console.ForegroundColor = ConsoleColor.Green;
                Console.WriteLine(inputPath);
                Console.ResetColor();
                Console.Write("Путь к файлу результатов: ");
                Console.ForegroundColor = ConsoleColor.Green;
                Console.WriteLine(outputPath);
                Console.ResetColor();


                Console.WriteLine("\nДля выбора пункта меню введите
соответствующую цифру.\n");
                Console.WriteLine("1. Изменить путь к файлу входных
данных.");
                Console.WriteLine("2. Изменить путь к файлу
результатов.\n");
```

```
            if (graph == null)
            {
               Console.ForegroundColor = ConsoleColor.Red;
               Console.WriteLine("3. Выполнить загрузку
гиперграфа из файла.\n");
               Console.ResetColor();
            }
            else
            {
               Console.WriteLine("3. Выполнить загрузку
гиперграфа из файла.\n");
            }


            Console.WriteLine("4. Выполнить расчет числа
автоморфизмов согласно теореме 2.");
            Console.WriteLine("5. Выполнить расчет числа
автоморфизмов согласно теореме 3.");


            Console.WriteLine("\n0. Выход.\n");


            ConsoleKeyInfo key = Console.ReadKey(true);


            input = null; output = null;
            switch (key.Key)
            {
               case ConsoleKey.D1:
                  {
                     Console.Clear();
                     Console.WriteLine("Пример: " + @"D:\\
input.txt");
                     Console.WriteLine("Введите полный путь к
файлу входных данных:");
```

```
                    var path = Console.ReadLine();
                    if (System.IO.File.Exists(path))
                    {
                        inputPath = path;
                        Console.WriteLine("Файл существует.");
                    }
                    else
                    {
                        PrintError("Файл не найден. Искомый путь: "
+ path);
                    }
                } break;
            case ConsoleKey.D2:
                {
                    Console.Clear();
                    Console.WriteLine("Пример: " + @"D:\\
output.txt");
                    Console.ForegroundColor = ConsoleColor.Red;
                    Console.Write("Внимание!");
                    Console.ResetColor();
                    Console.WriteLine(" Файл будет перезаписан
или создан.");
                    Console.WriteLine("Введите полный путь к
файлу результатов:");
                    var path = Console.ReadLine();
                    if (LoadStreamWriter(out output, path))
                    {
                        outputPath = path;
                    }
                } break;
            case ConsoleKey.D3:
```

```
                {
                    Console.Clear();

                    if (LoadStreamReader(out input, inputPath))
                    {
                        input.BaseStream.Position = 0;
                        try
                        {
                            graph = new
HyperGraph(Matrix<int>.LoadFromStream(input));
                            Console.WriteLine("Гиперграф загружен.");
                        }
                        catch (Exception e)
                        {
                            PrintError(e.Message);
                        }
                        input.Close();
                    }
                } break;
            case ConsoleKey.D4:
                {
                    Console.Clear();
                    if (graph != null)
                    {
                        try
                        {
                            LoadStreamWriter(out output, outputPath);
                            //  Вычислить количество автоморфизмов
графа и вывести перестановки
                            ExampleTheorema(graph, output);
```

```
                        output.Close();
                        Console.WriteLine("\nВывод выполнен в
файл: " + outputPath);
                    }
                    catch (Exception e)
                    {
                        PrintError(e.Message);
                    }
                }
                else
                {
                    PrintError("Гиперграф не загружен.");
                }

            } break;
        case ConsoleKey.D5:
            {
                Console.Clear();
                if (graph != null)
                {
                    try
                    {
                        LoadStreamWriter(out output, outputPath);
                        //  Вычислить количество автоморфизмов
графа и вывести их
                        ExampleTheoremaThird(graph, output);
                        output.Close();
                        Console.WriteLine("\nВывод выполнен в
файл: " + outputPath);
                    }
                    catch (Exception e)
```

```
                    {
                        PrintError(e.Message);
                    }
                }
                else
                {
                    PrintError("Гиперграф не загружен.");
                }
            } break;
        case ConsoleKey.D0: { if (input != null)
{ input.Close(); } if (output != null) { output.Close(); }
Environment.Exit(0); } break;
        default: break;
    }
    if (input != null)
    {
        input.Close();
    }

    if (output != null)
    {
        output.Close();
    }

    Console.WriteLine("\nДля продолжения нажмите
любую клавишу.");
    Console.ReadKey();
    Console.Clear();
  }

}
```

```
        static bool LoadStreamReader(out System.IO.StreamReader
stream, string path)
        {
            try
            {
                stream = new System.IO.StreamReader(path);
                return true;
            }
            catch (Exception e)
            {
                PrintError(new string[] { e.Message, "Укажите
корректный путь к файлу входных данных." });
                stream = null;
                return false;
            }
        }


        static bool LoadStreamWriter(out System.IO.StreamWriter
stream, string path)
        {
            try
            {
                stream = new System.IO.StreamWriter(path);
                return true;
            }
            catch (Exception e)
            {
                PrintError(e.Message);
                stream = null;
                return false;
```

```
            }
        }

        static void PrintError(string message)
        {
            Console.ForegroundColor = ConsoleColor.Red;
            Console.WriteLine(message);
            Console.ResetColor();
        }
        static void PrintError(string[] messages)
        {
            Console.ForegroundColor = ConsoleColor.Red;
            foreach (var message in messages)
            {
                Console.WriteLine(message);
            }

            Console.ResetColor();
        }
    }
}
```