МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ
ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
«ВЯТСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ»
ФАКУЛЬТЕТ ПРИКЛАДНОЙ МАТЕМАТИКИ И ТЕЛЕКОММУНИКАЦИЙ
КАФЕДРА «ПРИКЛАДНАЯ МАТЕМАТИКА И ИНФОРМАТИКА»

Допускаю к защите
Заведующий кафедрой ПМиИ
_____/ Иномистов В.Ю.
(подпись) (Ф.И.О.)

ГРАФЫ ПОВОРОТОВ ВТОРОГО УРОВНЯ

Пояснительная записка дипломной работы

ТПЖА.010441.090 ПЗ

Разработал студент гр. ПМИ-41 _____ / Стерлягов А.А. / _____

Руководитель к.ф.-м.н., доцент каф. ПМиИ_____ / Пушкарев И.А. / _____

Киров 2015

Содержание

Введение

Преобразование Донахью, впервые введённое Р. Донахью в [1] является важным объектом для изучения по двум причинам:

- его можно считать адекватной комбинаторной моделью хаоса, более содержательной чем обычно рассматриваемые (такие, как «преобразования пекаря» [2]);
- оно напоминает задачи, возникающие при попытках математически смоделировать процессы, протекающие на наноуровне.

Однако прогресс, достигнутый к настоящему моменту в изучении этого преобразования сравнительно невелик, так как оно плохо поддаётся изучению стандартными методами.

Одним из возможных подходов является рассмотрение «приближений конечной глубины». Первым шагом стала работа [3], однако имеет смысл продолжить изучение вопросов, поставленных в ней.

Данная работа посвящена попытке сделать второй шаг в указанном направлении, а именно изучить поворот второго уровня в плоских кубических деревьях с висячим корнем и граф поворотов второго уровня, описать их свойства и характеристики.

1 Обзор информации, связанной с проблематикой работы

1.1 Основные определения

В данной работе используется терминология и система определений, введенная в [3] и [4].

Рассматриваемые в дальнейшем преобразования работают на множестве плоских кубических деревьев с висячим корнем (для краткости можно использовать аббревиатуру ПКДВК). Все вершины ПКДВК делятся на три типа:

- висячий корень – выделенная вершина, которая имеет ровно одного сына;
- нелистья – вершины, которые имеют ровно двух сыновей;
- листья – вершины, не имеющие сыновей.

Пусть α_0 – сын корня произвольного ПКДВК T, α_1 – правый сын вершины α_0, α_2 – правый сын вершины α_1 и т. д., α_m – правый сын вершины α_{m-1} и при этом лист. Последовательность вершин $(\alpha_0, \alpha_1, \dots, \alpha_m)$ называется старшей правой цепью [4]. Все вершины старшей правой цепи дерева T, в свою очередь, являются корнями дочерних деревьев. Дерево с корнем α_i обозначим $T_i = T_i(\alpha_i)$. Данную ситуацию можно описать формулой

$$T = |T_0(\alpha_0), \dots, T_{m-1}(\alpha_{m-1}), \alpha_m > \quad (1)$$

которая называется правым разложением дерева T [4].

Аналогично определяется и левое разложение дерева T:

$$T =< \beta_0, S_1(\beta_1), S_2(\beta_2), \dots, S_t(\beta_t)| \quad (2)$$

где β_0 – самый левый лист,

β_t – сын корня [4].

Для обозначения чисел Каталана используется соотношение

$$C_n = \frac{1}{n+1}\binom{2n}{n}, \quad (3)$$

так что количество ПКДВК с n листьями равно C_{n-1}.

1.2 Преобразование Донахью

Как уже говорилось, впервые рассматриваемое преобразование было введено Р. Донахью в [1]. Позднее, независимо от Донахью, данное преобразование было рассмотрено в [5] под названием «фрактальный поворот плоских деревьев». Итогом исследований стала работа [4], в которой было рассмотрено множество интересных свойств этого преобразования.

Определение 1.1 (преобразование Донахью) [4].

(1) База индукции. Если T – тривиальное дерево, состоящее только из корня и сына корня, то $T' = T$.

(2) Индукционный переход. Пусть T – ПКДВК и для всех ПКДВК S с меньшим количеством листьев образ S' уже определен. Далее, пусть $T = |T_0(\alpha_0), \dots, T_{n-1}(\alpha_{n-1}), \alpha_n >$ – правое разложение дерева T. Тогда образом T называется дерево T', левое разложение которого есть $T' =< \alpha_0, T_0'(\alpha_1), \dots, T_{n-1}'(\alpha_n)|$.

На рисунке 1.1 продемонстрировано, как действует преобразование Донахью на ПКДВК.

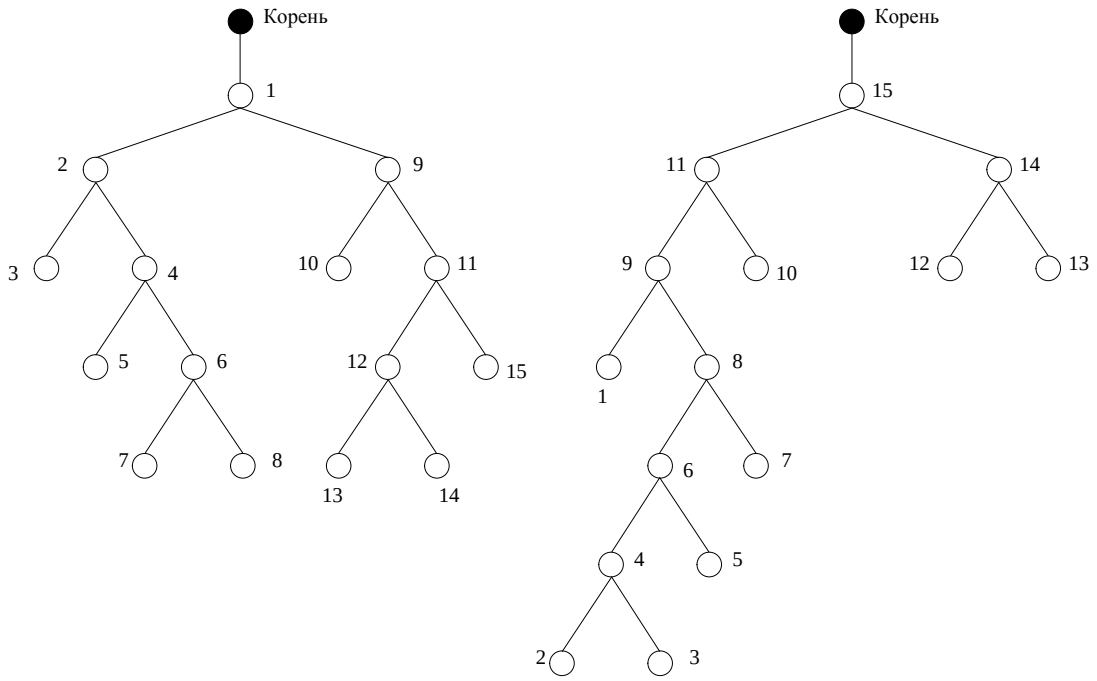


Рисунок 1.1 – Пример преобразования Донахью

Возможно ввести и другие определения данного преобразования, основанные на других комбинаторных интерпретациях чисел Каталана, например таких, как плоские деревьях с висячим корнем или триангуляции многоугольников [4].

1.3 Поворот первого уровня

1.3.1 Простой поворот

Определение 1.2. Пусть $T = |T_0(\alpha_0), \dots, T_{n-1}(\alpha_{n-1}), \alpha_n >$ – правое разложение ПКДВК T. Простым поворотом называется преобразование, которое переводит дерево T в дерево $\gamma(T) =< \alpha_0, T_0(\alpha_1), \dots, T_{n-1}(\alpha_n)|$ [3].

Рисунок 1.2 показывает, как преобразуется дерево при простом повороте.

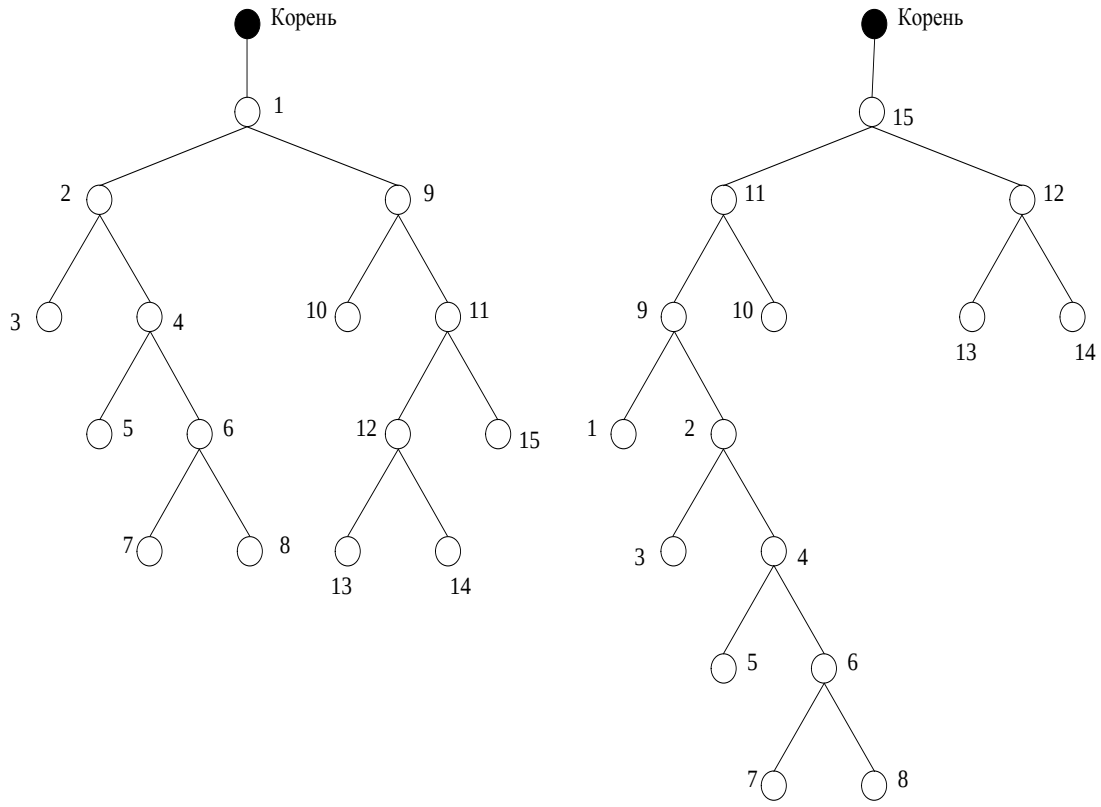


Рисунок 1.2 – Пример простого поворота ПКДВК

1.3.2 Граф поворотов первого уровня

Определение 1.3. Композицией числа n называется его разложение в сумму упорядоченных слагаемых [6].

Рассмотрим ориентированный граф G_n, вершинами которого являются всевозможные композиции числа $n-1$.

Ребрам графа G_n соответствуют всевозможные ПКДВК с n листьями, а именно из вершины $\overline{x} = (x_0, x_1, \dots, x_m)$ в вершину $\overline{y} = (y_0, y_1, \dots, y_t)$ идут ребра, соответствующие тем и только тем ПКДВК, у которых правое разложение имеет вид

$$T = |T_0(\alpha_0), \dots, T_t(\alpha_t), \alpha_{t+1} > \qquad (4)$$

и одновременно левое разложение имеет вид

$$T =< \beta_0, S_0(\beta_1), \dots, S_m(\beta_{m+1})| \quad (5)$$

где количество листьев дерева $T_i(\alpha_i)$ равно y_i, а количество листьев дерева $S_j(\beta_{j+1})$ равно x_j.

Стоит сразу отметить, что ребра идущие из вершины $\overline{x}$ в вершину $\overline{y}$ соответствуют деревьям вида (5), для которых

$$T_0(\alpha_0) =< \beta_0, S_0(\beta_1), \dots, S_{m-1}(\beta_m)|, \quad (6)$$

или, что симметрично, деревьям вида (4), для которых

$$S_m(\beta_{m+1}) = |T_1(\alpha_1), \dots, T_t(\alpha_t), \alpha_{t+1} >. \quad (7)$$

В таком случае очевидны соотношения

$$x_m = \sum_{i=1}^{t} y_i + 1 \quad (8)$$

$$y_0 =. \sum_{i=0}^{m-1} x_i + 1 \quad (9)$$

Количество таких деревьев (то есть количество кратных ребер, идущих из вершины $\overline{y}$ в вершину $\overline{x}$) равно

$$\theta(\overline{x}, \overline{y}) = \prod_{i=0}^{m-1} C_{x_i-1} \times \prod_{j=1}^{t} C_{y_j-1}. \quad (10)$$

Количество вершин графа G_n равно 2^{n-2} [7], а количество ребер равно числу Каталана C_{n-1}.

Примеры рассмотренных графов изображены на рисунках 1.3 и 1.4.

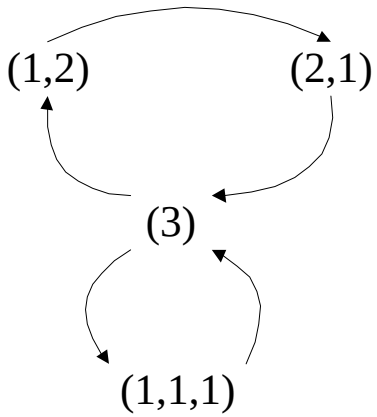


Рисунок 1.3 – Граф G_4

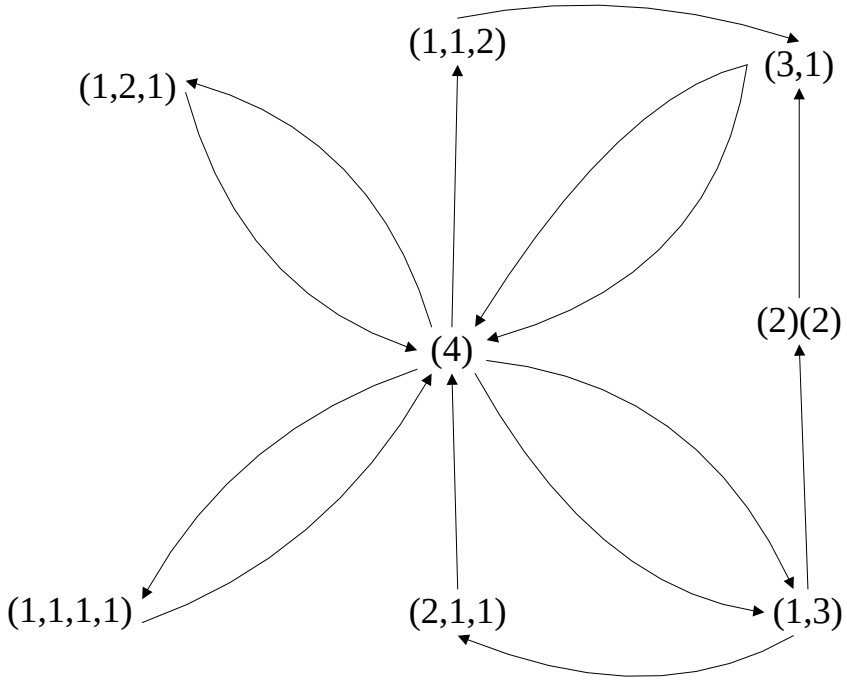


Рисунок 1.4 – Граф G_5

В [3] отмечаются два простых, но важных свойства графа G_n.

Теорема 1.1.

(1) Граф G_n эйлеров.

(2) Все простые ориентированные циклы графа G_n проходят через вершину $(n-1)$.

Доказательства этих фактов приводятся в [3].

Определение 1.4.

(1) Граф G_n называется графом поворотов первого уровня.

(2) Разбиение всех ребер графа G_n на ориентированные циклы называется поворотом первого уровня. Множество таких разбиений непустое именно потому, что граф эйлеров.

(3) Циклы, на которые конкретный поворот разбивает ребра графа G_n называются орбитами этого поворота.

Теорема 1.2. Простой поворот и преобразование Донахью являются поворотами первого уровня.

В [3] изучается вопрос о количестве орбит поворота первого уровня. Для этого рассматриваются эйлеровы циклы в графе G_n.

Вершины в G_n можно упорядочить лексикографически: $(n-1)$ является первой вершиной, $(n-2,1)$ – второй, и так далее, последней вершина $\underbrace{(1,1,\dots,1)}_{n-1}$. Соответствующая этому упорядочению матрица Кирхгофа графа G_n отличается от верхнетреугольной только первым столбцом [3]. Поэтому главный минор матрицы, получаемый вычеркиванием первого столбца и первой строки, является определителем верхнетреугольной матрицы и равен произведению диагональных элементов этой матрицы – полустепеней входа всех вершин графа G_n, кроме вершины $(n-1)$.

Полустепень вершины $(x_1,\dots,x_m)$ равна количеству деревьев с левым разложением вида (5), то есть произведению

$$in(v) = \prod_{i=0}^{m} C_{x_i - 1}. \quad (11)$$

Соответственно, степень, в которой число C_{i-1}, входит как множитель в рассматриваемый минор, равна количеству $f_{n-1,i}$ появлений числа i как слагаемого во всевозможных композициях числа $n-1$.

В работе [6] показано, что $f_{n,1} = (n+2)2^{n-3}$ при $n \geq 2$ (исключая очевидное $f_{1,1} = 1$), и $f_{n,i} = f_{n-i+1,1} = (n-i+3)2^{n-i-2}$ при $n \geq i$.

Соответственно, количество остовных деревьев в графе G_n, растущих из вершины $(n-1)$ есть

$$T(G_n) = \prod_{k=1}^{n-2} (C_{k-1})^{(n-k+2)2^{n-k-3}}. \quad (12)$$

Стоит отметить, что в эйлеровом орграфе количество остовных деревьев, растущих из вершины всегда одинаково, вне зависимости от выбора этой вершины (этот факт является следствием теоремы BEST, которая подробнее описана в [8]).

Далее согласно той же теореме BEST, количество эйлеровых циклов в графе G_n равно

$$C(G_n) = T(G_n) \prod_{i_1 + i_2 + \cdots + i_k = n-1} \left(C_{i_1 - 1} C_{i_2 - 1} \dots C_{i_k - 1} - 1\right)!, \quad (13)$$

где произведение берется по всем композициям числа $(n-1)$.

Таким образом, в данном разделе была проанализирована теоретическая информация о преобразовании Донахью и поворотах первого уровня в ПКДВК.

2 Актуальность темы дипломной работы. Постановка задачи

На основе анализа теоретической информации можно сформулировать следующие задачи:

- определить понятия поворота второго уровня в ПКДВК и графа поворотов второго уровня;
- изучить свойства графа поворотов второго уровня;
- определить характеристики различных графов поворотов второго уровня.

3 Граф поворотов второго уровня

3.1 Основные определения

Определение 2.1. Пусть $(x_0, \dots, x_m)$ – композиция числа n. Тогда совокупность $(x_{01}, \dots, x_{0t_1})(x_{11}, \dots, x_{1t_2}) \dots (x_{m1}, \dots, x_{mt_m})$, где $(x_{i1}, \dots, x_{it_i})$ – композиция числа x_i называется композицией композиции числа n.

Рассмотрим орграф G'_n, вершинами которого являются композиции композиций числа $n-1$.

Ребро из вершины $\overline{x} = (x_{01}, \dots, x_{1l_0})(x_{11}, \dots, x_{1l_1}) \dots (x_{m1}, \dots, x_{ml_m})$ в вершину $\overline{y} = (y_{01}, \dots, y_{0k_0})(y_{11}, \dots, y_{1k_1}) \dots (y_{t1}, \dots, y_{tk_t})$ проведем, если существует ПКДВК T такое, что

$$\begin{gathered} T = |T_0(\alpha_0), \dots, T_t(\alpha_t), \alpha_{t+1} > = \\ < \beta_0, S_0(\beta_1), S_1(\beta_2), \dots, S_m(\beta_{m+1})| \\ T_i = |T_{i0}(\delta_{i0}), \dots, T_{ik_i}(\delta_{ik_i}), \delta_{ik_i+1} > \\ S_j =< \gamma_{j0}, S_{j0}(\gamma_{j1}), S_{j1}(\gamma_{j2}), \dots, S_{jl_j}(\gamma_{jl_j+1})| \end{gathered} \tag{14}$$

и при этом в T_{ij} ровно y_{ij} листьев, а в S_{ij} ровно x_{ij}.

Определение 2.2. Полученный граф называется графом поворотов второго уровня.

Примеры графов поворотов второго уровня приведены на рисунках 3.1 и 3.2.

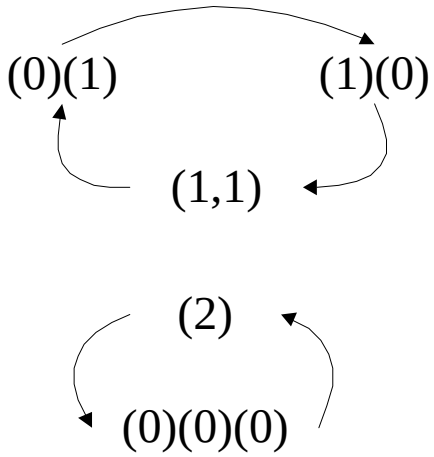


Рисунок 3.1 – Граф поворотов второго уровня для $n = 4$

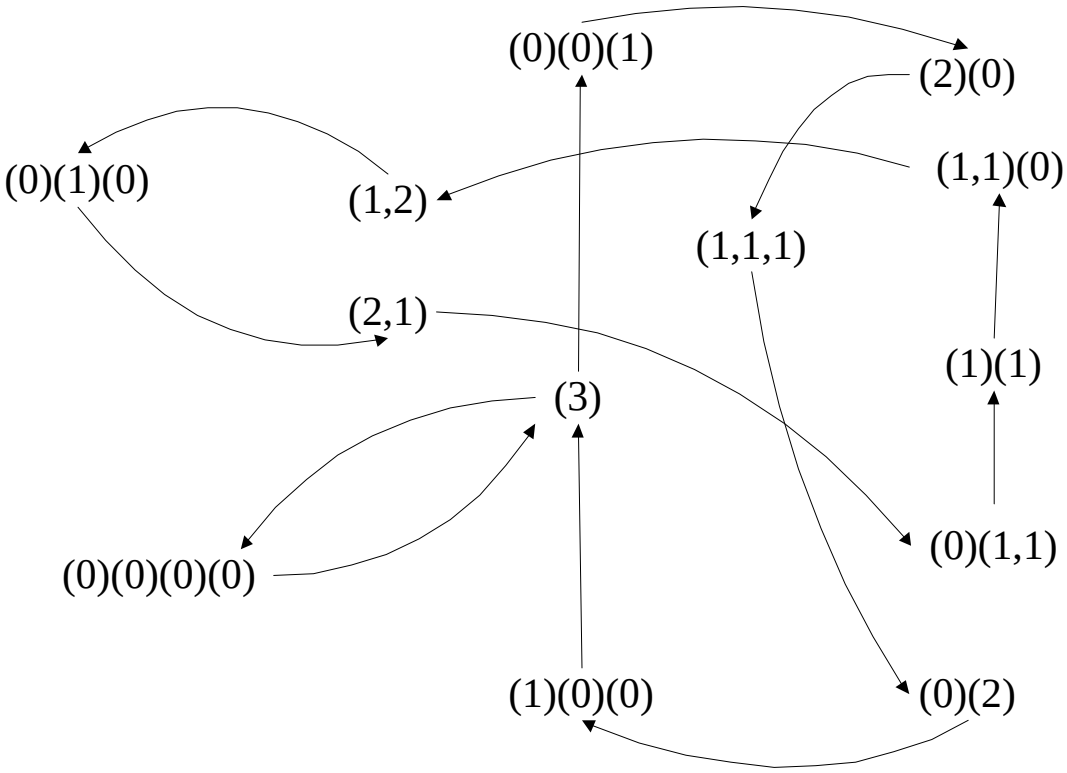


Рисунок 3.2 – Граф поворотов второго уровня для $n = 5$

Если сравнить граф поворотов второго уровня с соответствующим ему графом поворотов первого уровня, то можно заметить, что они отличаются тем, что в графе поворотов второго уровня вершины распадаются на несколько. Например, у графа поворотов второго уровня для $n = 5$ вершины

(3), (2,1), (1,2) и (1,1,1) получились из вершины (4) графа поворотов первого уровня.

3.2 Исследование графа поворотов второго уровня

У графа поворотов второго уровня можно отметить несколько важных свойств.

Теорема 2.1. Каждая компонента связности графа G'_n является эйлеровым графом.

Доказательство. Пусть $(x_{01}, \dots, x_{1l_0})(x_{11}, \dots, x_{1l_1}) \dots (x_{m1}, \dots, x_{ml_m})$ – некоторая вершина одной из компонент связности графа G_n. Для каждого дерева

$$\begin{aligned} T &= |T_0(\alpha_0), \dots, T_t(\alpha_t), \alpha_{t+1} > \\ T_i &= |T_{i0}(\delta_{i0}), \dots, T_{ik_i}(\delta_{ik_i}), \delta_{ik_i+1} > \end{aligned} \qquad (15)$$

которое формирует ребро, входящее в вершину, существует единственное дерево

$$\begin{aligned} S &= < \beta_0, S_0(\beta_1), \dots, S_t(\beta_{t+1})| \\ S_i &= < \gamma_{i0}, T_{i0}(\gamma_{i1}), \dots, T_{ik_i}(\gamma_{ik_i+1})| \end{aligned} \qquad (16)$$

которое формирует ребро, исходящее из вершины. Справедливо и обратное утверждение. Следовательно, для каждой вершины графа поворотов второго уровня количество входящих в нее ребер и исходящих из нее ребер равно, значит выбранная компонента связи эйлерова, а, следовательно, эйлеровы и все остальные компоненты связности.

■

Определение 2.3. Поворотом второго уровня называется разбиение графа поворотов второго уровня на ориентированные циклы.

Теорема 2.2. Любой поворот второго уровня является поворотом первого уровня. Обратное неверно.

Доказательство. Можно заметить, что по построению граф поворотов второго уровня определяет повороты первого уровня, при этом отсекая некоторые. Действительно, на рисунке 1.4 можно увидеть, что существует поворот первого уровня $(1,1,1,1) \to (4) \to (1,3)$, который невозможно реализовать в поворотах второго уровня, так как вершины $(0)(0)(0)(0)$ и $(0)(2)$ находятся в разных компонентах связности, что продемонстрировано на рисунке 2.2.

■

Теорема 2.3. В графе поворотов второго уровня для деревьев с n листьями через вершины, которые являются композициями числа $(n-1)$, проходят все простые циклы.

Доказательство. Если принять во внимание факт, что в графе поворотов первого уровня все простые циклы проходят через вершину $(n-1)$ [3], и то, что вершины в графе поворотов второго уровня – разорванные вершины графа поворотов первого уровня, то справедливость утверждения становится очевидна.

■

Для дальнейшего изучения графов поворотов второго уровня необходима информация о количестве в нем ребер, вершин и эйлеровых циклов. Соответствующие соотношения вывести не так просто, поэтому для выдвижения гипотез необходимо иметь какие-либо эмпирические данные. Для этого нужно реализовать программу, с помощью которой можно было бы рассчитать все необходимые значения для любого графа G'_n.

4 Разработка программного обеспечения

4.1 Обоснование выбора средств реализации

Как уже отмечалось, исследовать свойства графов поворотов второго уровня вручную не представляется возможным. Поэтому была поставлена задача реализовать программу для автоматического построения графа и расчета его параметров.

Для реализации приложения был выбран язык C#. Такой выбор языка обусловлен несколькими причинами:

- удобная работа с длинной арифметикой при помощи структуры BigInteger;
- наличие специального языка запросов LINQ, который значительно упрощает работу с большими объемами данных;
- встроенная реализация списка объектов, поддерживающая поиск, выполнение сортировки и других операций.

Графический интерфейс реализован с помощью технологии WPF. Она предоставляет следующие преимущества:

- использование аппаратно-независимой графикой – приложение будет выглядеть одинаково на любом поддерживаемом устройстве;
- наличие графических примитивов, таких как прямоугольники, эллипсы и графические пути, которые могут включать в себя более простые примитивы;
- использование аппаратного ускорения графики через DirectX.

Реализуемая программа должна рассчитывать следующие показатели для графа поворотов второго уровня:

- число вершин;
- число ребер;
- матрицу смежности;

- количество компонент связности;
- количество эйлеровых циклов.

4.2 Реализация основных функций

Схемы алгоритмов реализованных функций приведены в приложении А, а исходные коды – в приложении Б.

Для того, чтобы рассчитать показатели для графа поворотов второго уровня, сначала необходимо сгенерировать все композиции и все композиции композиций числа $n-1$. За это отвечают функции GenerateEdgesFirstLevel() и GenerateEdgesSecndLevel() соответственно. Для генерации композиций используется рекурсивная функция Step(). Далее, пусть $(x_0, \dots, x_m)$ – есть некоторая композиция числа $n-1$. Для того, чтобы получить какую-либо композицию этой композиции необходимо рассчитать композиции для всех $x_i - 1$ и из каждого элемента получившегося списка списков композиций выбрать какую-либо композицию. Очевидно, что для того, чтобы получить все возможные композиции композиций числа $n-1$ необходимо проделать эту операцию с каждой композицией. Этот алгоритм и реализует функция GenerateEdgesSecndLevel(), попутно считая количество полученных вершин.

Для генерации матрицы смежности графа поворотов второго уровня необходима реализация ПКДВК и методов для работы с ним. Класс Node реализует вершину в ПКДВК. На рисунке 4.1 приведена схема этого класса.

Метод GetLeavesCount() возвращает количество листьев в дереве, корнем которого является текущая вершина. Метод вызывается при создании вершины и заносит возвращаемое значение в поле LeavesCount. Поля DoubleLeftDecompozition и DoubleRightDecompozition хранят левое разложение левого разложения и правое разложения правого разложения соответственно. Поля Left и Right хранят ссылки на левого и правого сына вершины. Для вершин, являющихся листьями, в обоих полях хранится null.

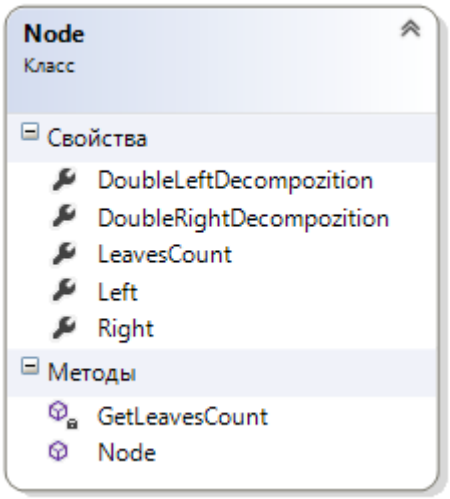


Рисунок 4.1 – Схема класса Node

Статический класс FlatCubicTree, схема которого изображена на рисунке 4.2, предоставляет методы для работы с ПКДВК.

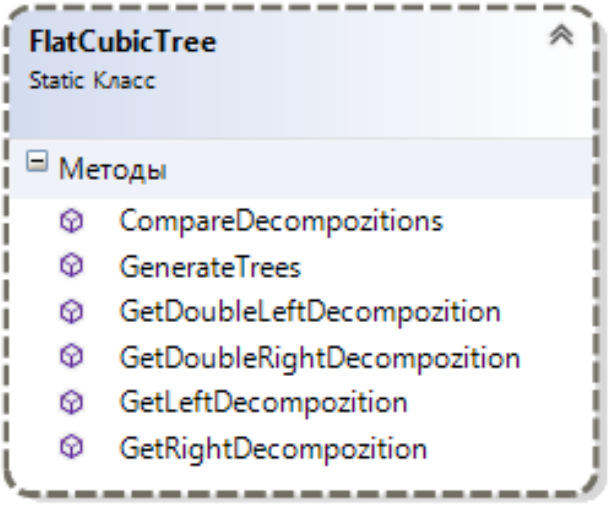


Рисунок 4.2 – Схема класса FlatCubicTree

Метод CompareDecompozitions() служит для поэлементного сравнения двух разложений дерева. Если два разложения не отличаются, то он возвращает true, иначе false.

Метод GenerateTrees(), принимающий на вход число s служит для генерации всевозможных ПКДВК с s листьями или, точнее, всех ПКДВК, у которых ровно $s-1$ вершина не является ни листом, ни корнем. Для этого используется рекурсивный LINQ-запрос, который генерирует все возможные левые и правые поддеревья.

Функции GetLeftDecompozition() и GetRightDecompozition() возвращают левое и правое разложения дерева соответственно. Функция GetLeftDecompozition() проходит от самого левого листа до сына корня и считает количество листьев в ПКДВК, корнями которых являются эти вершины. Функция GetRightDecompozition() работает аналогично, отличаясь лишь тем, что она проходит от сына корня до самого правого листа.

Функции GetDoubleLeftDecompozition() и GetDoubleRightDecompozition() считают левое разложение левого разложения и правое разложение правого разложения. Работают функции аналогично предыдущим, отличаясь лишь тем, что они считают не количество листьев в поддереве, а считают для него левое или правое разложение соответственно.

Генерирует матрицу смежности графа G_n' функция GraphTurnSecndLev(). Для этого сначала генерируются все ПКДВК с $n-1$ листом. Далее, алгоритм пробегает по всем парам вершин графа и по всем деревьям. Если левое разложение дерева совпадает с одной вершиной из пары, а правое с другой, то в матрицу смежности в соответствующую позицию прибавляется единица. В конце производится подсчет суммы элементов в матрице смежности, которая является количеством ребер в графе поворотов второго уровня.

Для подсчета количества компонент связности графа используется поиск в глубину, реализованный в функции Dfs(). Поиск в глубину работает следующим образом: берется первая неиспользованная вершина, добавляется в список вершин текущей компоненты связности. Если у этой вершины есть

соседи, в которых алгоритм еще не побывал, то вызывается функция Dfs() от первого такого соседа. Таким образом рано или поздно алгоритм побывает во всех вершинах текущей компоненты связности. Если в графе еще остались неиспользованные вершины, то запускается алгоритм от первой из таких вершин. Иначе выполняется условие останова алгоритма, и его работа завершается.

Подсчет количества эйлеровых циклов в графе осуществляет функция GetCyclesCount(). Для этого используется ранее упомянутая теорема BEST и матричная теорема о деревьях для ориентированных графов. По теореме BEST количество эйлеровых циклов в орграфе равно

$$C(G) = T_n(G) \prod_{v \in V} (in(v) - 1)! \qquad (17)$$

где $T_n(G)$ – количество остовных деревьев графа G, растущих из v_n.

$T_n(G)$ можно найти из матричной теоремы о деревьях, которая гласит, что количество остовных деревьев, растущих из вершины v_n орграфа G равно n-му диагональному минору матрицы Кирхгофа (определителю матрицы, из которой вычеркнули n-ю строку и n-ый столбец). Остается заметить, что для эйлерова графа количество В функции GetCyclesCount() реализовано вычисление матрицы Кирхгофа и расчет количества эйлеровых циклов по вышеприведенной формуле.

На рисунке 4.3 приведен вид главного окна разработанной программы. Вывод матрицы графа поворотов осуществляется в текстовый файл.

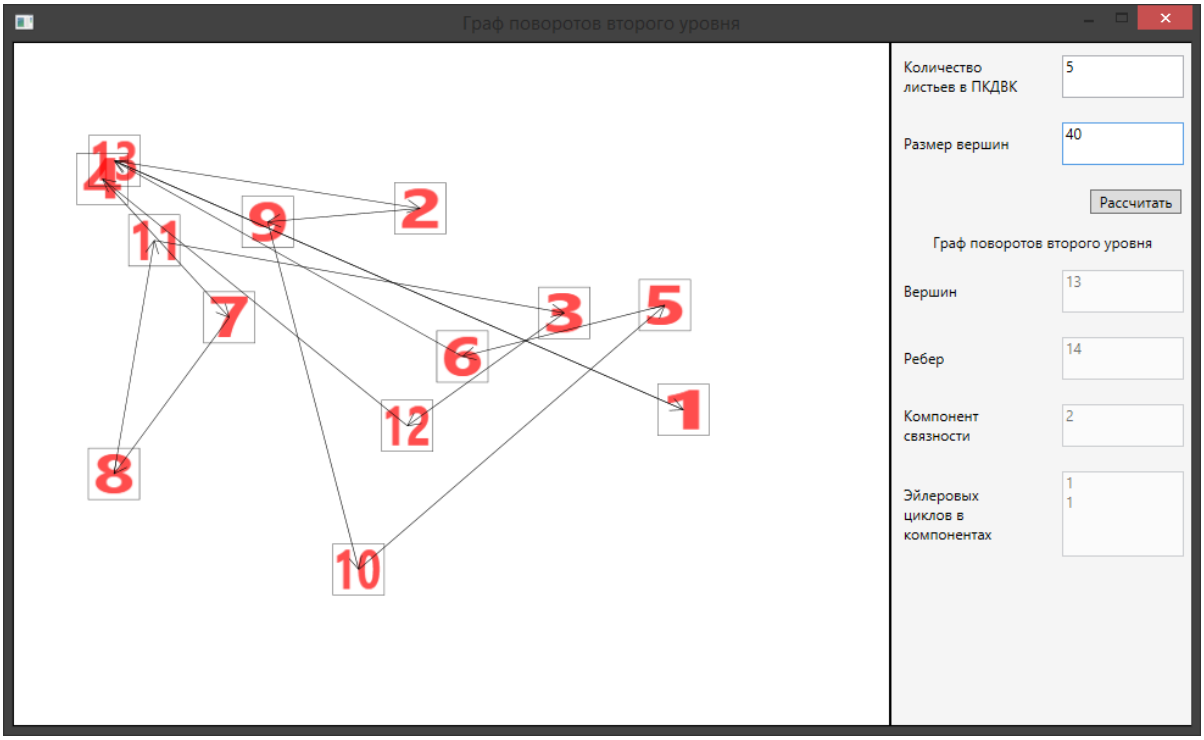


Рисунок 4.3 – Главное окно реализованного приложения

4.3 Анализ полученных результатов

В следующей таблице приведены значения рассчитываемых параметров для графов поворотов второго уровня.

Таблица 1 – Параметры графа поворотов второго уровня при различных n

Количество листьев в ПКДВК	Количество ребер	Количество вершин	Количество компонент связности	Количество эйлеровых циклов по компонентам
2	0	1	1	0
3	2	2	1	1
4	5	5	2	1, 1
5	14	13	2	1, 1
6	42	34	3	48, 1, 1

Продолжение таблицы 1

7	132	89	1	616381269626018 8569600
8	429	233	1	263073878881764 221315902044659 914132710649718 795159188951325 302996036887159 291039074630182 307828842850313 375981722664960 000000000000000 00
9	1430	610	1	Очень много

Как и предполагалось количество ребер в графе поворотов второго уровня не изменилось относительно графа поворотов первого уровня – их количество равно C_{n-1}.

Гораздо более интересными оказались результаты вычисления количества вершин графа G'_n.

Таблица 2 – Количество вершин графа поворотов второго уровня

Количество листьев в ПКДВК	2	3	4	5	6	7	8
Количество вершин	1	2	5	13	34	89	233

Продолжение таблицы 2

Количество листьев в ПКДВК	9	10	11	12	13	14	15
Количество вершин	610	1597	4181	10946	28657	75025	196418

Первые пятнадцать вычисленных элементов последовательности количеств вершин в графах поворотов второго уровня совпадают с элементами последовательности чисел Фибоначчи с четными номерами.

Полученный результат позволяет выдвинуть следующую гипотезу.

Гипотеза 3.1.

Пусть G'_n, граф поворотов второго уровня для ПКДВК с n листьями. Тогда количество вершин в этом графе равно числу Фибоначчи с четным номером.

Для проверки данной гипотезы необходимо большее количество экспериментальных данных, что требует улучшения существующих алгоритмов построения графа поворотов.

Интересным наблюдением является то, что граф поворотов второго уровня для ПКДВК с малым количеством листьев имеет несколько компонент связности, а для количества листьев больше 6 становится связным. Причина, по которой это происходит неясна и требует дальнейшего исследования. На основании этого результата можно сформулировать еще одну гипотезу.

Гипотеза 3.2.

G'_n является сильно связанным (то есть всего его вершины взаимно достижимы) при $n > 6$.

Заключение

В результате выполнения дипломной работы были получены следующие результаты:

- рассмотрена и проанализирована научная литература, связанная с проблематикой работы;
- определены и исследованы графы поворотов второго уровня, сформулировано и доказано несколько теорем, описывающих их свойства;
- определены количественные характеристики некоторых графов поворотов второго уровня, на основе этого выдвинуты две гипотезы: о количестве вершин в графе поворотов второго уровня и о связности графа поворотов второго уровня при больших n.

Дальнейшее изучение графов поворотов второго уровня может быть направлено на:

- дальнейшее изучение количественных характеристик, совершенствование алгоритмов построения графов;
- доказательство гипотезы о количестве вершин в графе поворотов второго уровня, выявление причин, по которым количество вершин выражается именно числами Фибоначчи;
- изучение причин, по которым лишь некоторые графы поворотов второго не являются связными.

Приложение А
(справочное).
Схемы алгоритмов основных функций

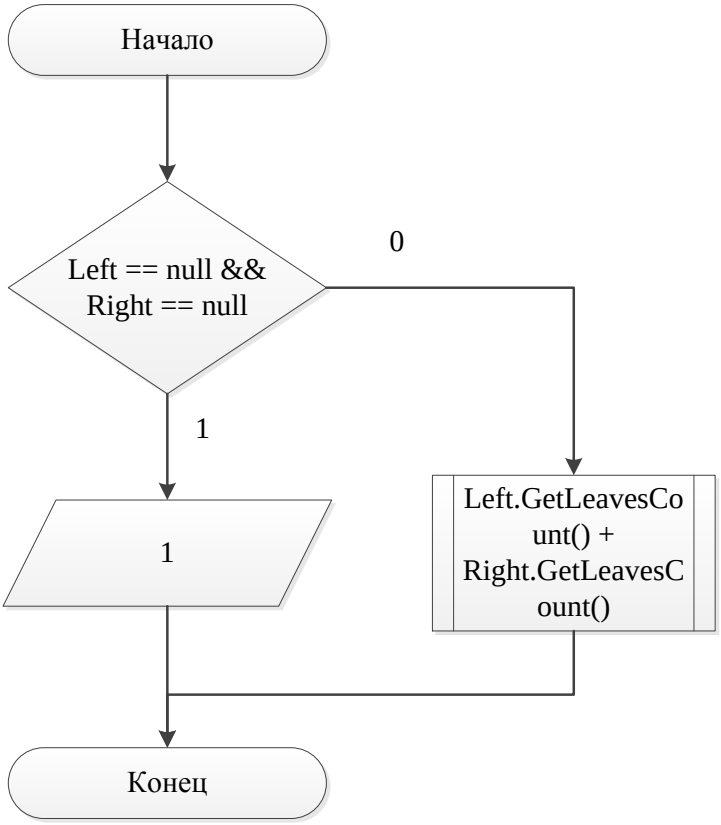


Рисунок А.1 – Схема алгоритма функции Node.GetLeavesCount()

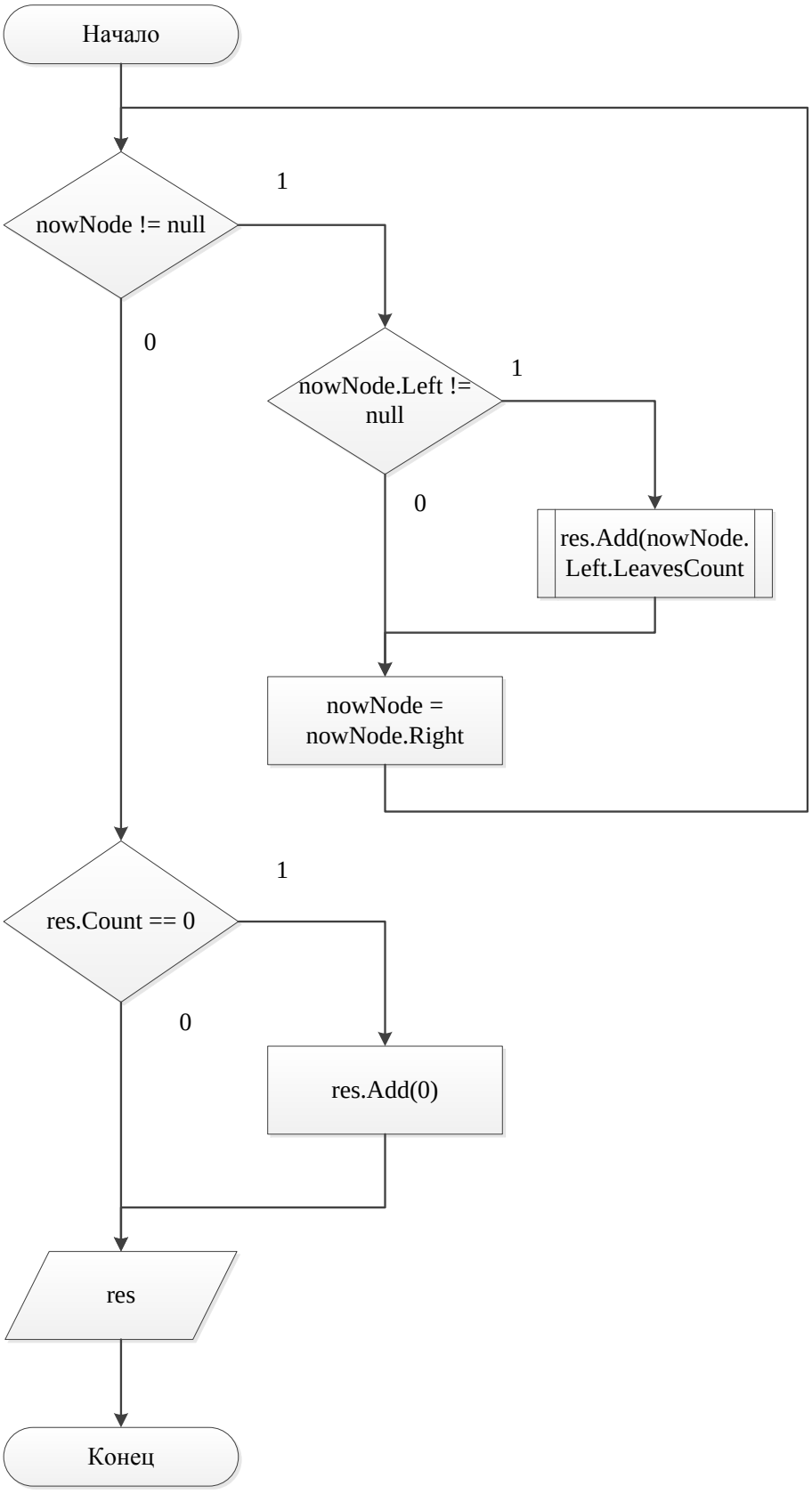


Рисунок А.2 – Схема алгоритма функции FlatCubicTree.GetRightDecompozition(Node node)

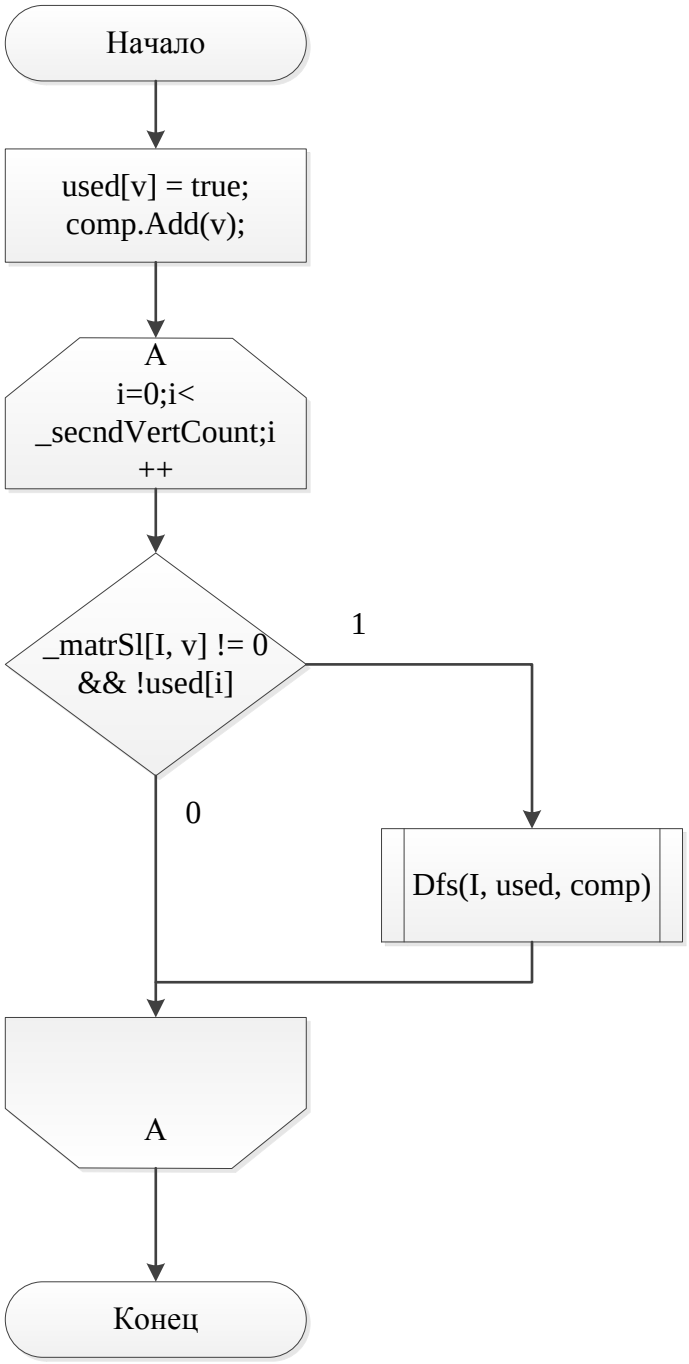


Рисунок А.3 – Схема алгоритма функции TurnGraph.Dfs(int v, List<bool> used, List<int> comp)

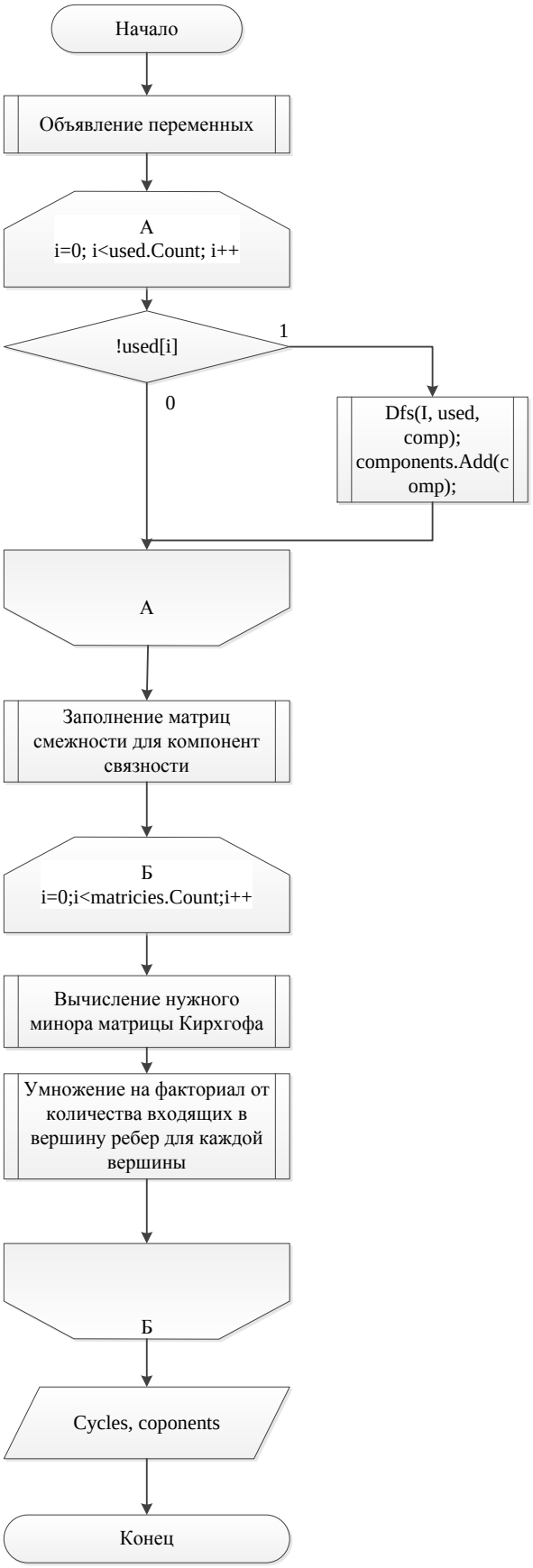


Рисунок А.4 – Схема алгоритма функции TurnGraph.GetCyclesCount()

Приложение Б
(справочное).
Часть листинга программы

```
//представляет вершину дерева
class Node
{
    public List<List<int>> DoubleLeftDecompozition { get; set; }
    public List<List<int>> DoubleRightDecompozition { get; set; }
    //ссылка налевого сына
    public Node Left { get; private set; }
    //ссылка на правого сына
    public Node Right { get; private set; }
    public int LeavesCount { get; private set; }
    public Node(Node left, Node right)
    {
        Left = left;
        Right = right;
        LeavesCount = (Left != null && Right != null ? Left.GetLeavesCount() +
Right.GetLeavesCount() : 1);
    }
    /// <summary>
    /// возвращает количество листьев в дереве
    /// </summary>
    public int GetLeavesCount()
    {
        if (Left == null && Right == null)
            return 1;
        return (Left != null ? Left.GetLeavesCount() : 0) + (Right != null ?
Right.GetLeavesCount() : 0);
    }
}
static class FlatCubicTree
{
    //возвращает левое разложение левого разложения дерева
    public static List<List<int>> GetDoubleLeftDecompozition(Node node)
    {
        var res = new List<List<int>>();
        var nowNode = node;
        while (nowNode != null)
        {
            var right = nowNode.Right;
            if (right != null)
                res.Insert(0, GetLeftDecompozition(right));
            nowNode = nowNode.Left;
        }
        return res;
    }
    //возвращает правое разложение правого разложения дерева
    public static List<List<int>> GetDoubleRightDecompozition(Node node)
    {
        var res = new List<List<int>>();
        var nowNode = node;
        while (nowNode != null)
        {
            var left = nowNode.Left;
            if (left != null)
                res.Add(GetRightDecompozition(left));
```

```
            nowNode = nowNode.Right;
        }
        return res;
    }
    //возвращает правое разложение дерева
    public static List<int> GetRightDecompozition(Node node)
    {
        var res = new List<int>();
        var nowNode = node;
        while (nowNode != null)
        {
            if (nowNode.Left != null)
                res.Add(nowNode.Left.LeavesCount);
            nowNode = nowNode.Right;
        }
        if (res.Count == 0)
            res.Add(0);
        return res;
    }
    //возвращает левое разложение дерева
    public static List<int> GetLeftDecompozition(Node node)
    {
        var res = new List<int>();
        var nowNode = node;
        while (nowNode != null)
        {
            if (nowNode.Right != null)
                res.Insert(0, nowNode.Right.LeavesCount);
            nowNode = nowNode.Left;
        }
        if (res.Count == 0)
            res.Add(0);
        return res;
    }
    //генерирует все ПКДВК с size листьями
    public static IEnumerable<Node> GenerateTrees(int size)
    {
        if (size == 0)
            return new Node[] { new Node(null, null) };
        return from i in Enumerable.Range(0, size)
               from left in GenerateTrees(i)
               from right in GenerateTrees(size - 1 - i)
               select new Node(left, right);
    }
    //сравнивает два разложения
    public static bool CompareDecompozitions(List<List<int>> l1, List<List<int>> l2)
    {
        var res = true;
        if (l1.Count == l2.Count)
        {
            for (int i = 0; i < l1.Count && res; i++)
            {
                if (l1[i].Count == l2[i].Count)
                {
                    for (int j = 0; j < l1[i].Count && res; j++)
                    {
                        if (l1[i][j] != l2[i][j])
                            res = false;
                    }
                }
                else
                    res = false;
```

```
            }
        }
        else
            res = false;
        return res;
    }
}
class TurnGraph
{
    //генерирует вершины графа поворотов 1 уровня
    private List<List<int>> GenerateEdgesFirstLevel(int n)
    {
        Step(n - 1, n - 1);
        var fCompoz = new List<List<int>>();
        foreach (var item in _compoz)
        {
            fCompoz.Add(new List<int>());
            fCompoz.Last().AddRange(item);
            item.Clear();
        }
        _compoz.Clear();
        _firstVertCount = fCompoz.Count;
        _firstEdgeCount = Cat(n - 1);
        return fCompoz;
    }
    //генерирует вершины графа поворотов 2 уровня
    private List<List<List<int>>> GenerateEdgesSecndLevel(List<List<int>> fCompoz, ref
Dictionary<int, int> numsVertex)
    {
        var sCompoz = new List<List<List<int>>>();
        numsVertex = new Dictionary<int, int>();
        foreach (var itemList in fCompoz)
        {
            var tmpList = new List<List<List<int>>>();
            foreach (var item in itemList)
            {
                var k = item;
                Step(k - 1, k - 1);
                tmpList.Add(new List<List<int>>());
                foreach (var itemC in _compoz)
                {
                    tmpList.Last().Add(new List<int>());
                    tmpList.Last().Last().AddRange(itemC);
                }
                _compoz.Clear();
            }
            var iterCount = tmpList.Aggregate(1, (current, item) => current *
item.Count);
            var iter = 0;
            var indicies = new int[tmpList.Count];
            var one = new int[tmpList.Count];
            one[0] = 1;
            do
            {
                var vertex = new List<List<int>>();
                for (int i = 0; i < tmpList.Count; i++)
                {
                    var list = tmpList[i][indicies[i]];
                    vertex.Add(new List<int>());
                    vertex.Last().AddRange(list);
                }
                sCompoz.Add(vertex);
```

```
                numsVertex.Add(sCompoz.Count - 1, fCompoz.IndexOf(itemList));
                _secndVertCount++;
                for (int i = 0; i < indicies.Count(); i++)
                {
                    indicies[i] += one[i];
                    if (i < indicies.Count() - 1)
                        indicies[i + 1] += (indicies[i] / tmpList[i].Count);
                    indicies[i] %= tmpList[i].Count;
                }
                iter++;
            }
            while (iter < iterCount);
            tmpList.Clear();
        }
        return sCompoz;
    }
    //генерирует матрицу графа поворотов второго уровня
    private void GraphTurnSecndLev(List<List<List<int>>> sCompoz, int n)
    {
        var listTrees = FlatCubicTree.GenerateTrees(n - 1).ToList().ToArray();
        for (int i = 0; i < listTrees.Length; i++)
        {
            listTrees[i].DoubleLeftDecompozition =
FlatCubicTree.GetDoubleLeftDecompozition(listTrees[i]);
            listTrees[i].DoubleRightDecompozition =
FlatCubicTree.GetDoubleRightDecompozition(listTrees[i]);
        }
        _matrSl = new int[_secndVertCount, _secndVertCount];
        for (int i = 0; i < sCompoz.Count; i++)
        {
            for (int j = 0; j < sCompoz.Count; j++)
            {
                if (i != j)
                {
                    for (int k = 0; k < listTrees.Length; k++)
                    {
                        if
(FlatCubicTree.CompareDecompozitions(listTrees[k].DoubleLeftDecompozition, sCompoz[i]) &&

FlatCubicTree.CompareDecompozitions(listTrees[k].DoubleRightDecompozition, sCompoz[j]))
                        {
                            _secndEdgeCount++;
                            _matrSl[j, i]++;
                        }
                    }
                }
            }
        }
    }
    //определяет количество эйлеровых циклов в графе
    private List<object> GetCyclesCount()
    {
        var cycles = new List<BigInteger>();
        var used = new List<bool>();
        for (int i = 0; i < _secndVertCount; i++)
            used.Add(false);
        var components = new List<List<int>>();
        for (int i = 0; i < used.Count; i++)
        {
            if (!used[i])
            {
                var comp = new List<int>();
```

```
                Dfs(i, used, comp);
                comp.Sort();
                components.Add(comp);
            }
        }
        var matricies = new List<int[,]>();
        foreach (var comp in components)
        {
            matricies.Add(new int[comp.Count,comp.Count]);
            for (int i = 0; i < comp.Count; i++)
                for (int j = 0; j < comp.Count; j++)
                    matricies.Last()[i, j] = _matrSl[comp[i], comp[j]];
        }
        var c = 0;
        foreach (var matr in matricies)
        {
            var len = components[c].Count;
            var kirgoff = new int[len, len];
            for (int i = 0; i < len; i++)
                for (int j = 0; j < len; j++)
                    if (i != j)
                        kirgoff[i, j] = (-1)*matr[i, j];
                    else
                        for (int k = 0; k < len; k++)
                            kirgoff[i, j] += matr[k, i];
            var detMatr = new int[len - 1, len - 1];
            for (int i = 1; i < len; i++)
                for (int j = 1; j < len; j++)
                    detMatr[i - 1, j - 1] = kirgoff[i, j];
            var nowCycles = Det(detMatr, len - 1);
            for (int i = 0; i < len; i++)
            {
                var inv = 0;
                for (int k = 0; k < len; k++)
                    inv += matr[i, k]; //matr[k, i];
                nowCycles *= Factorial(inv - 1);
            }
            cycles.Add(nowCycles);
            c++;
        }
        var a = new List<object> { components.Count, cycles, components };
        return a;
    }
    //поиск в глубину
    private void Dfs(int v, List<bool> used, List<int> comp)
    {
        used[v] = true;
        comp.Add(v);
        for (int i = 0; i < _secndVertCount; i++)
            if (_matrSl[i, v] != 0 && !used[i])
                Dfs(i, used, comp);
    }
    private static BigInteger ProdTree(int l, int r)
    {
        if (l > r)
            return 1;
        if (l == r)
            return l;
        if (r - l == 1)
            return (BigInteger)l * r;
        int m = (l + r) / 2;
        return ProdTree(l, m) * ProdTree(m + 1, r);
```

```
        }
        //факториал числа
        private static BigInteger Factorial(int n)
        {
            if (n < 0)
                return 0;
            if (n == 0)
                return 1;
            if (n == 1 || n == 2)
                return n;
            return ProdTree(2, n);
        }
        //определитель матрицы
        private BigInteger Det(int[,] matr, int len)
        {
            var dMatr = new double[len, len];
            double det = 1.0;
            for (int i = 0; i < len; i++)
                for (int j = 0; j < len; j++)
                    dMatr[i, j] = matr[i, j];
            for (int i = 0; i < len - 1; i++)
                for (int j = i + 1; j < len; j++)
                {
                    var mnozh = dMatr[j, i] / dMatr[i, i];
                    for (int l = 0; l < len; l++)
                        dMatr[j, l] = dMatr[j, l] - mnozh * dMatr[i, l];
                }
            for (int i = 0; i < len; i++)
                det *= dMatr[i, i];
            return (BigInteger)det;
        }
        //шаг рекурсии, считающей композиции числа
        private void Step(int n, int beg)
        {
            for (int i = 1; i <= n; i++)
            {
                _prev.Add(i);
                Step(n - i, beg);
            }
            if (_prev.Sum() == beg)
            {
                if (_prev.Count == 0)
                    _prev.Add(0);
                _compoz.Add(new List<int>(_prev));
            }
            if (_prev.Count > 0)
                _prev.RemoveAt(_prev.Count - 1);
        }
        //биномиальные коэффициенты
        private int C(int n, int k)
        {
            double res = 1;
            for (int i = 1; i <= k; ++i)
                res = res * (n - k + i) / i;
            return (int)(res + 0.01);
        }
        //число Каталана
        private int Cat(int n)
        {
            return n >= 0 ? (C(2 * n, n) / (n + 1)) : 1;
        }
    }
```

Приложение В
(справочное).
Графическая часть

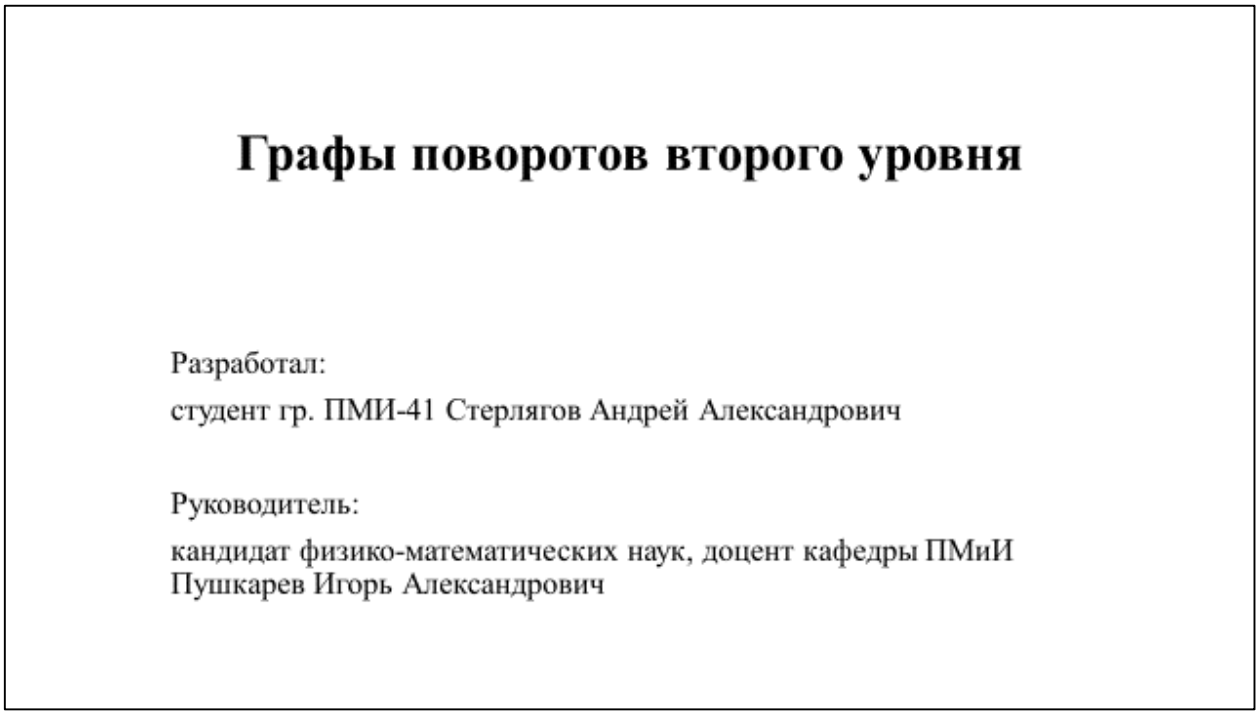


Рисунок В.1 – Титульный слайд

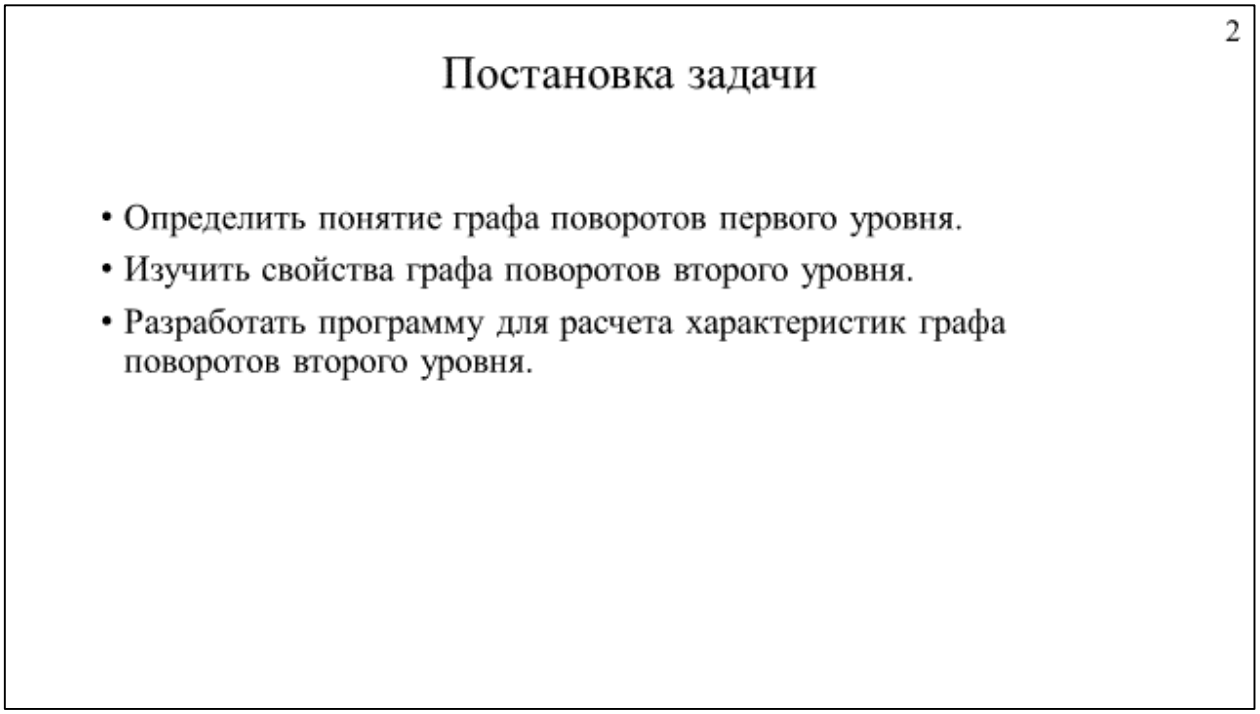


Рисунок В.2 – Слайд «Постановка задачи»

3

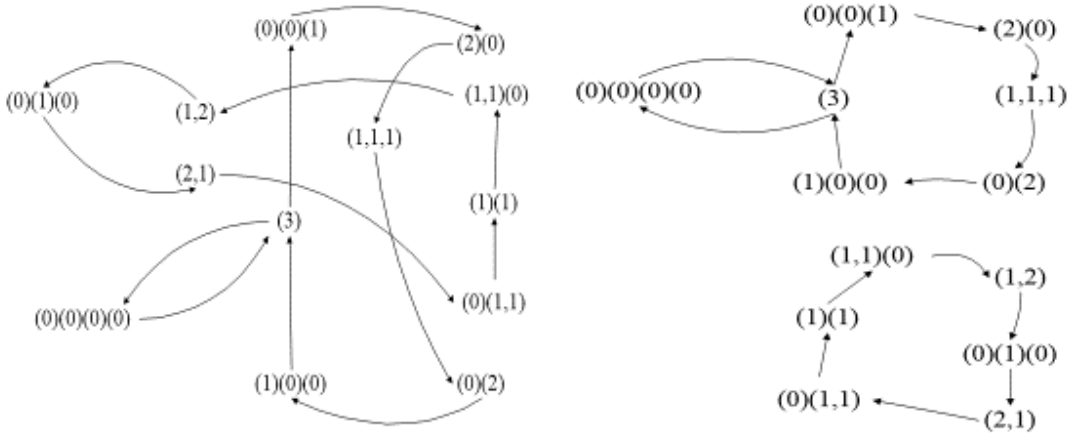
Основные определения

- Композицией числа n называется его разложение в сумму упорядоченных слагаемых.
- Композицией композиции числа n называется совокупность композиций каждого элемента его композиции.

Рисунок В.3 – Слайд «Основные определения»

4

Пример графа поворотов второго уровня



Граф поворотов второго уровня для $n=5$

Рисунок В.4 – Слайд «Пример графа поворотов второго уровня»

5

Свойства графа поворотов второго уровня

- Каждая компонента связности является эйлеровым графом.
- Любой поворот второго уровня является поворотом первого уровня.
- Через вершины, которые являются композициями числа (n-1) проходят все простые циклы.

Рисунок В.5 – Слайд «Свойства графа поворотов второго уровня»

6

Реализация программного обеспечения

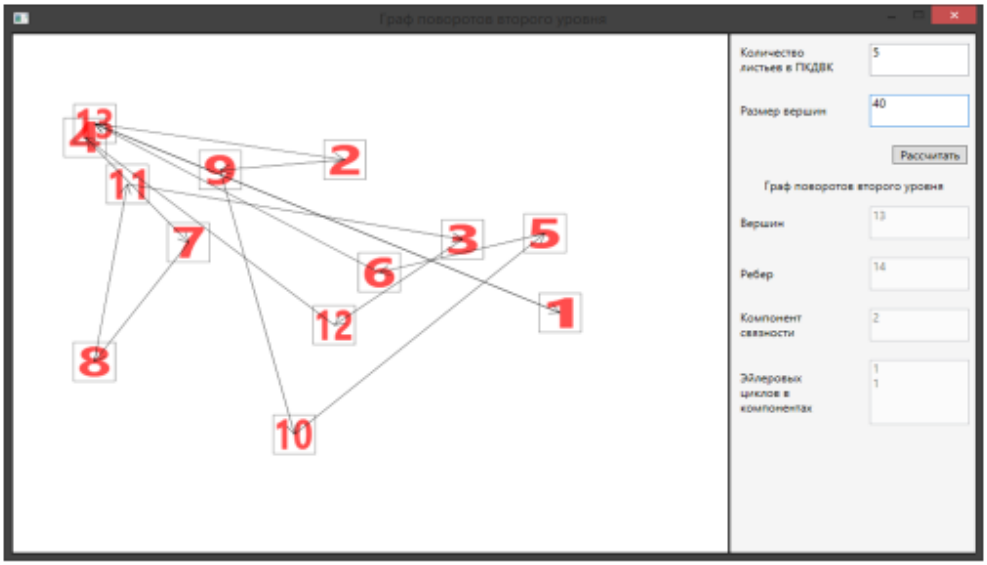


Рисунок В.6 – Слайд «Реализация программного обеспечения»

7

Анализ полученных результатов

Количество листьев в ПКДВК	2	3	4	5	6	7	8
Количество вершин	1	2	5	13	34	89	233
Количество листьев в ПКДВК	9	10	11	12	13	14	15
Количество вершин	610	1597	4181	10946	28657	75025	196418

Рисунок В.7 – Слайд «Анализ полученных результатов (1)»

8

Анализ полученных результатов

Количество листьев в ПКДВК	2	3	4	5	6	7	8	9
Количество компонент связности	1	1	2	2	3	1	1	1

Рисунок В.8 – Слайд «Анализ полученных результатов (2)»

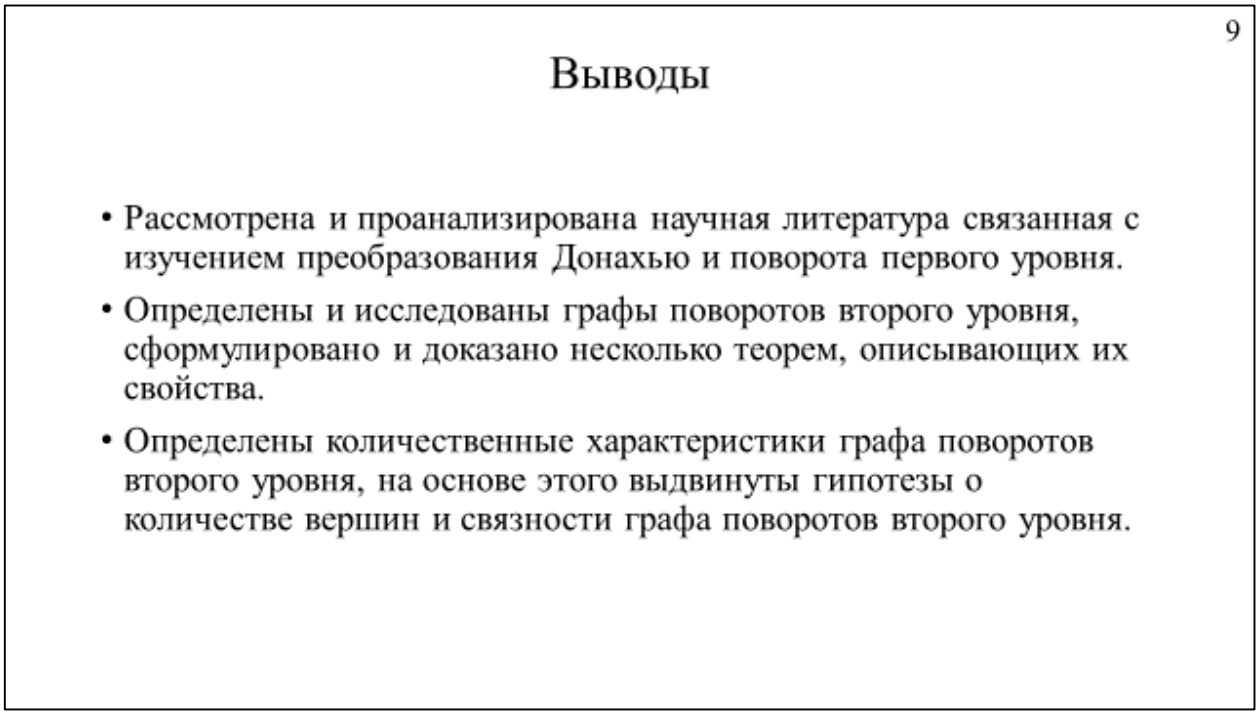


Рисунок В.9 – Слайд «Выводы»

Приложение Г
(обязательное).
Авторская справка

Я, Стерлягов Андрей Александрович, автор дипломной работы, сообщаю, что мне известно о персональной ответственности автора за разглашение сведений, подлежащих защите законами РФ о защите объектов интеллектуальной собственности. Одновременно сообщаю, что:

1) при подготовке к защите дипломной работы не использованы источники (документы, отчеты, диссертации, литература и т.п.), имеющие гриф секретности или «Для служебного пользования» ВятГУ или другой организации;

2) данная работа не связана с незавершенными исследованиями или уже с завершенными, но еще официально не разрешенными к опубликованию ВятГУ или другими организациями;

3) данная работа не содержит коммерческую информацию, способную нанести ущерб интеллектуальной собственности ВятГУ или другой организации;

4) данная работа не является результатом НИР или ОКР, выполняемой по договору с организацией;

5) в предлагаемом к опубликованию тексте нет данных по незащищенным объектам интеллектуальной собственности других авторов;

6) согласен на использование результатов своей работы ВятГУ для учебного процесса;

7) использование моей дипломной работы в научных исследованиях оформляется в соответствии с законодательством РФ о защите интеллектуальной собственности.

«__»_____ 2015 г. Подпись автора _____

Сведения по авторской справке подтверждаю

«__»_____ 2015 г. Зав. кафедрой _____

Приложение Д
(обязательное).
Библиографический список

1. Donaghey R. Automorphisms of Catalan trees and bracketing // J. Combin. Theory. 1980. № 1. С. 75-90.
2. Чулаевский В. А. Преобразование пекаря // Квант, № 4, 1989. С. 19-23.
3. Пушкарев И.А., Бызов В.А. Поворот первого уровня на множестве плоских деревьев // Записки научных семинаров ПОМИ. 2013. Т. 411. С. 178-190.
4. Пушкарев И.А., Бызов В.А. Пребразование Донахью: элементарный подход // Записки научных семинаров ПОМИ. 2013. Т. 411. С. 148-178.
5. Пушкарев И.А. Об одном преобразовании плоских деревьев // Мат. вестник педвузов и университетов Волго-Вятского региона. 2008. № 10. С. 82-89.
6. Chinn P.Z., Colyer G., Flashman M., Migiliore E. Cuisenaire rods go to college // PRIMUS: Problems, Resources, and Issues in Mathematics Undergraduate Studies. 1992. Т. 2. С. 118-130.
7. Сачков В.Н. Комбинаторные методы дискретной математики. Москва: Наука, 1977. 241 с.
8. Татт У. Теория графов: Пер. с англ. Москва: Мир, 1988.