Double Triangle Descendants of K_5

by

Mohamed Laradji

B.Sc. in Mechanical Engineering, King Fahd University of Petroleum and Minerals, 2015

Thesis Submitted in Partial Fulfillment of the
Requirements for the Degree of
Master of Science

in the
Department of Mathematics
Faculty of Science

SIMON FRASER UNIVERSITY
Fall 2017

Approval

Name:	**Mohamed Laradji**
Degree:	**Master of Science (Mathematics)**
Title:	**Double Triangle Descendants of K_5**
Examining Committee:	**Chair:** Tom Archibald Professor
Karen Yeats Senior Supervisor Associate Professor	_____
Matthew DeVos Supervisor Associate Professor	_____
Marni Mishna Supervisor Associate Professor	_____
Luis Goddyn Internal Examiner Professor	_____
Date Defended:	**December 5, 2017**

Abstract

Feynman diagrams in ϕ^4 theory can be represented as 4-regular graphs. The Feynman integral, or even the Feynman period, is very hard to calculate. A graph invariant, called the c_2-invariant, is conjecturally thought to be equal for two graphs when their periods are equal. Double triangle reduction of 4-regular graphs is known to preserve the c_2-invariant. Double triangle descendants of K_5 all have a c_2-invariant that is a constant -1, and conjecturally, are the only graphs with this c_2-invariant. This thesis studies the structure of K_5-descendants to gain insight on the c_2-invariant, get closer to solving the conjecture, and to study what is an interesting combinatorial operation in its own right. It will be shown that the minimum number of triangles in a descendant is 4. Closed-form generating functions are found for three families of K_5-descendants. Two encodings, one for n-zigzags, and a general one for all K_5-descendants, are found.

Keywords: ϕ^4 theory; Feynman diagram; Feynman period; c_2-invariant; constant c_2-invariant; double triangle reduction; double triangle expansion; K_5-descendants; classes of K_5-descendants; zigzag graphs; n-zigzag graphs; enumeration

Acknowledgements

I would like to thank Karen Yeats for introducing me to the problem of characterizing K_5-descendants, and for her constant support. I would also like to thank Marni Mishna for her useful comments throughout the editing process, Matt Devos for our insightful meetings, and Dana and Sam for their friendship and support.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

The broad objective of this thesis is to uncover combinatorial structure in order to simplify calculations of a specific graph-defined integral. In particular, the thesis aims to characterize the effect of repeated applications of a certain graph operation combinatorially, in the hope of getting closer to solving a related conjecture. In addition, several interesting sequences arise when enumerating these graphs that are produced by the operation.

The aforementioned graph operation is called double triangle expansion. One of the main reasons it is of interest is that, for a large number of graphs, it preserves a graph invariant known as the c_2-invariant. Broadly speaking, the c_2-invariant is related to the number of solutions to a graph polynomial, called the Kirchhoff polynomial. The following paragraphs aim to explain the motivation for the problem. The terms used here are defined more rigorously in the background chapter, Chapter 2.

Kontsevich conjectured in 1997 that the point-counting function of the zeroes of the Kirchhoff polynomial ($\Psi(G)$) of a graph G over the finite field with q elements, $[\Psi(G)]_q$, is a polynomial in q. If this were true then the quadratic coefficient of this purported polynomial, known as the c_2-invariant,

$$c_2^{(q)}(G) = \frac{[\Psi(G)]_q}{q^2} \mod q,$$

would be a constant. Indeed, this was verified for all graphs with at most 11 edges [15, p380]. Surprisingly, however, this conjecture was disproved in 2003 by Belkale and Brosnan [2, p171-2].

There exist infinite families of graphs with a constant c_2-invariant. For instance, Brown and Schnetz showed that if a primitive-divergent ϕ^4 graph G is two-vertex reducible, or has weight drop, then G has a zero c_2-invariant. If G has vertex width ≤ 3, then G has a constant c_2-invariant [4, p4]. Double triangle reduction and expansion of a 4-regular graph G is known to preserve the c_2-invariant [4, Corollary 35, p16], and so if G has a constant c_2-invariant and it has at least one triangle, an infinite family with constant c_2-invariants could then be constructed by repeated double triangle expansions of G. One such example is K_5, whose (decompleted) c_2-invariant is equal to a constant -1 [17, p6].

Since K_5 has at least one triangle, we can double triangle expand and construct an infinite family with the same (decompleted) c_2-invariant, which we call the family of K_5-descendants. Interestingly, every completed primitive graph G whose decompletions have a constant -1 c_2-invariant turned out to be a descendant of K_5. Brown and Schnetz conjectured that this is always the case [5, Conjecture 25, p16]. This is a strong conjecture, and if it is true, it suggests that there is some significance in the structure of K_5-descendants. Studying the structure of K_5-descendants can shed some light on the c_2-invariant and on the double triangle operations. It is hoped that it will get us closer to solving the aforementioned conjecture, and to understanding the double triangle operations better.

The next chapter aims to introduce the reader to background material that is used later on. The main topics covered include partitions of integers in Section 2.1, and the c_2-invariant in Section 2.3. Section 2.2 aims to familiarize the reader with the definitions and notations used here for some concepts in graph theory. In Chapter 3, the double triangle operations are introduced, followed by some known results on them.

Chapter 4 presents new results on the family of K_5-descendants. A list of K_5-descendants up to order 10 appears in Section 4.1. A notation for common subgraphs of K_5-descendants is introduced in Section 4.2, along with clarifying examples. In Section 4.3, an encoding for graphs with similar structure to K_5-descendants is introduced. This encoding is used in Section 4.5 to derive the closed-form generating functions for several order and triangle count combinations. The encoding is used again in Section 4.6, where it is proved that any K_5-descendant has at least 4 triangles.

Chapter 2

Background

This chapter aims to familiarize the reader with some definitions and known results that are used later in the document. Section 2.1 defines partitions of integers, and contains proofs for two known results that are later used in Section 4.5. Section 2.2 defines several graph classes and properties that are used throughout this document. Section 2.3 goes through some of the definitions in algebraic geometry, which are used to define the c_2-invariant. Section 2.4 is a short introduction to quantum field theory and ϕ^4-theory, from which the motivation of studying K_5-descendants ultimately comes.

2.1 Partitions of Integers

For an introduction to enumerative combinatorics, refer to Erickson's "Introduction to Combinatorics" [11]. The main results of this section are Lemma 2.1 and Lemma 2.2 on enumerating partitions with certain restrictions.

Let $\mathcal{C}$ be a set, and $|\cdot|_{\mathcal{C}} : \mathcal{C} \to \mathbb{N}$ be a *size function*, with the property that $\mathcal{C}_n := \{\gamma : |\gamma|_{\mathcal{C}} = n\}$ is finite for all $n \in \mathbb{N}$. We call the pair $(\mathcal{C}, |\cdot|_{\mathcal{C}})$ a *combinatorial class*. Defining $C_n := |\mathcal{C}_n|_{\mathcal{C}}$, the *ordinary generating function* of $\mathcal{C}$ is the formal sum

$$C(x) = \sum_{n \geq 0} C_n x^n.$$

In Section 4.5, the generating functions for several sequences are found by counting partitions. Recall that a *partition* of a positive integer n is an ordered list of positive integers $n_1 \geq n_2 \geq \cdots \geq n_k$ such that $n_1 + \cdots + n_k = n$. We present the following two known results, which are used in the proofs of Proposition 4.27 and Proposition 4.30 of Section 4.5.

Lemma 2.1. *The number of partitions of n into exactly k parts, each part ≥ 1, is generated by*

$$p_k(x) = x^k \prod_{i=1}^{k} \frac{1}{1-x^i}.$$

Proof. Observe that the number of partitions of n into at least k non-negative parts is the same as the number of partitions of n where the largest part is at least k. The number of partitions of n where the largest part is at least k is generated by

$$p_{\geq k}(x) = \prod_{i=k}^{\infty} \frac{1}{1-x^i}.$$

Thus, the number of partitions of n into exactly k non-negative parts is generated by

$$\begin{aligned} p_k(x) = p_{\geq k}(x) - p_{\geq k-1}(x) &= \prod_{i=k}^{\infty} \frac{1}{1-x^i} - \prod_{i=k-1}^{\infty} \frac{1}{1-x^i} \\ &= \prod_{i=k}^{\infty} \frac{1}{1-x^i} - (1-x^k) \prod_{i=k}^{\infty} \frac{1}{1-x^i} \\ &= x^k \prod_{i=1}^{k} \frac{1}{1-x^i}, \end{aligned}$$

as desired. □

Lemma 2.2. *The number of partitions of n into exactly k* distinct *parts, each part ≥ 1, is generated by*

$$q_k(x) = x^{k+\binom{k}{2}} \prod_{i=1}^{k} \frac{1}{1-x^i}.$$

Proof. Fix positive integers $n \geq k$. We prove the result by constructing a bijection between the set S_1 of partitions of $n - \binom{k}{2}$ with exactly k parts, and the set S_2 of partitions of n with exactly k *distinct* parts. Define $\phi : S_1 \to S_2$, with

$$\phi(m_0, m_1, \cdots, m_{k-1}) = (m_0 + 0, m_1 + 1, \cdots, m_i + i, \cdots, m_{k-1} + k - 1).$$

We first want to show that ϕ is well-defined. Suppose that

$$n - \binom{k}{2} = \sum_{i=0}^{k-1} m_i,$$

for some $1 \leq m_0 \leq m_1 \leq \cdots \leq m_{k-1}$. Let $n_i = m_i + i$. Since $m_i \leq m_{i+1}$,

$$n_i = m_i + i < m_{i+1} + i + 1 = n_{i+1}$$

Hence, $n_0, n_1, \cdots, n_{k-1}$ are distinct. Furthermore,

$$\sum_{i=0}^{k-1} n_i = \sum_{i=0}^{k-1} (m_i + i) = \sum_{i=0}^{k-1} m_i + \sum_{i=0}^{k-1} i = n - \binom{k}{2} + \binom{k}{2} = n,$$

and so ϕ is well-defined, as desired. It remains to show that ϕ is a bijection. This follows from the fact that ϕ has a well-defined inverse, $\phi^{-1} : S_2 \to S_1$, with

$$\phi^{-1}(n_0, n_1, \cdots, n_{k-1}) = (n_0 - 0, n_1 - 1, \cdots, n_i - 2, \cdots, n_{k-1} - (k-1)).$$

This "inverse" is well-defined since $n_0 \geq 1, n_1 \geq 2, \cdots, n_{k-1} \geq k$, and so

$$(n_0 - 0, n_1 - 1, \cdots, n_i - 2, \cdots, n_{k-1} - (k-1)) \in S_1,$$

as desired. The result then follows from Lemma 2.1, where we get

$$q_k(x) = x^{\binom{k}{2}} p_k(x) = x^{k+\binom{k}{2}} \prod_{i=1}^{k} \frac{1}{1 - x^i},$$

as desired. □

2.2 Graph Theory

In this section, we recall some graph theory notation and results, and introduce some definitions that will be used later in the document. For a treatment of basic graph theory definitions, refer to Pete Clark's "Graph Theory" [10]. In this section, we only consider simple graphs.

A graph central in this work is K_5, the complete graph on 5 vertices. If a graph G has a subgraph isomorphic to K_n for some n, then we say that G has an *n-clique*. We will study the operation of double triangle expansion, which is done on a triangle of a graph. A triangle refers to C_3, where C_n denotes the cycle on n vertices. The graphs K_5 and C_3 are shown in Figure 2.1.

In ϕ^4-theory, introduced in Section 2.4, graphs have a property known as completed primitiveness if they are internally 6-edge-connected. We present a definition of connectivity, followed by two examples.

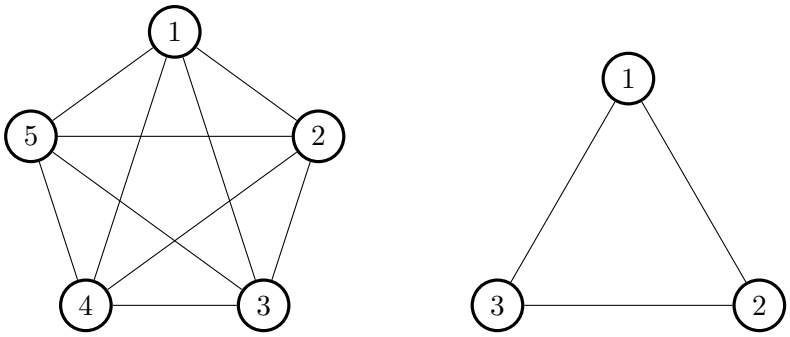


Figure 2.1: The complete graph K_5, and the triangle C_3.

Definition 2.3 (Connectivity)**.** Let G be a graph. We say G is *connected* if there exists a u, v-path for any $u, v \in V(G)$. G is *k-connected* if $G \setminus S$ is non-empty and connected for any $S \subseteq V(G)$ with $|S| \leq k-1$. G is *k-edge-connected* if $G \setminus S$ is connected for any $S \subseteq E(G)$ with $|S| \leq k-1$. The *connectivity* of G is denoted by κ, and $\kappa = k$ if G is k-connected but not $(k+1)$-connected. The *edge-connectivity* of G is denoted by λ, and $\lambda = k$ if G is k-edge-connected but not $(k+1)$-edge-connected. A graph G is said to be *internally k-edge-connected* if the only way to disconnect the graph by removing $k-1$ or fewer edges is to separate off a single vertex.

Example 2.4. For $n \geq 1$, $\kappa(K_n) = \lambda(K_n) = n-1$. For $n \geq 3$, $\kappa(C_n) = \lambda(C_n) = 2$. If T is a tree, $\kappa(T) = \lambda(T) = 1$.

Example 2.5. The complete graph K_n is internally n-edge-connected. The cycle C_n is internally 2-edge-connected.

Definition 2.6 (Induced subgraph)**.** A subgraph H of a graph G is said to be an *induced subgraph on $V(H)$* if for any $u, v \in V(H)$, $(u, v) \in E(G) \implies (u, v) \in E(H)$. In this case, we write $H = G[V(H)]$.

In Section 4.5, we study the triangle count of K_5-descendants. We will introduce some relevant definitions, followed by an illustrating figure (Figure 2.2).

Definition 2.7 (Triangle Count)**.** The *triangle count* of a graph G is the number of distinct triangles in G. We denote this number as $\text{tri}(G)$.

Definition 2.8 (Double Triangle)**.** Suppose that a graph G contains distinct vertices $0, 1, 2, 3$ such that $(0, 1, 2), (1, 2, 3)$ are triangles. We call the ordered 4-tuple $(0, 1, 2, 3)$ a *double triangle*, denoted as T^2.

Definition 2.9 (Triple Triangle)**.** Let G be a graph with distinct vertices $0, 1, 2, 3, 4$ such that $(0, 1, 2), (0, 1, 3), (0, 1, 4)$ are triangles. We call the ordered 5-tuple $(0, 1, 2, 3, 4)$ a *triple triangle*, denoted as T^3.

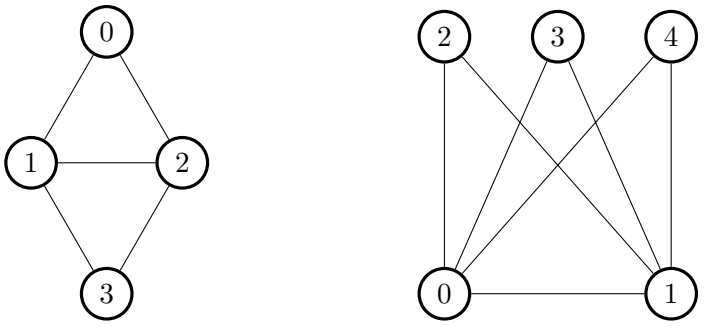


Figure 2.2: The double triangle $(0, 1, 2, 3)$, and the triple triangle $(0, 1, 2, 3, 4)$.

A lot of the graphs considered here, and all descendants of K_5, are 4-regular graphs. A graph G is *n-regular* if $\deg(v) = n$ for all $v \in V(G)$. A graph G is *regular* if it is n-regular for some n. For example, K_5 is 4-regular and C_3 is 2-regular, and they are both regular graphs. We will often want to group vertices of like degree together, and so we present the following definition.

Definition 2.10 (Degree set)**.** Let G be a graph. We define the *degree set*

$$D_i(G) := \{v \in V(G) : \deg v = i\}.$$

The c_2-invariant of a graph G, defined in Section 2.3, depends on the Kirchhoff polynomial of G, which can be calculated using the spanning trees of G. A *tree* is a connected acyclic graph. We present the definitions of a spanning tree and the Kirchhoff polynomial, followed by some examples.

Definition 2.11 (Spanning Tree)**.** For a graph G, a subgraph T is a *spanning tree* (of G) if $V(T) = V(G)$ and T is a tree.

Definition 2.12 (Kirchhoff Polynomial)**.** Let G be a graph. The Kirchhoff Polynomial is defined as

$$\Psi_G = \sum_T \prod_{e \notin E(T)} a_e$$

where the sum runs over the spanning trees of G, and the a_e's are variables indexed by the edges of G.

Example 2.13 (Kirchhoff polynomial of C_3)**.** We will derive the Kirchhoff polynomial of the triangle, C_3. Referring to Figure 2.3, we obtain

$$\Psi_{C_3} = \sum_T \prod_{e \notin E(T)} a_e = a_1 + a_2 + a_3.$$

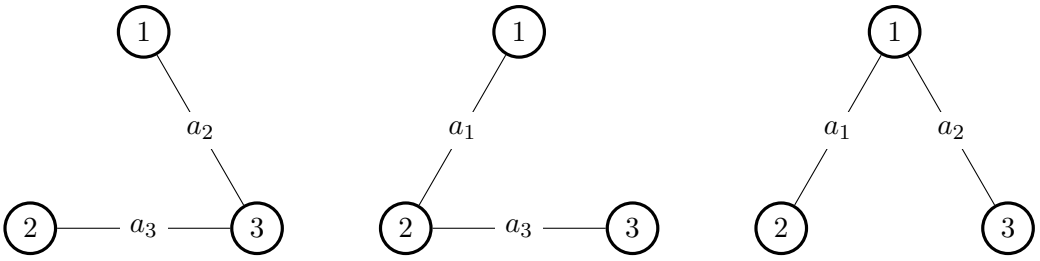


Figure 2.3: Edge-labelled spanning trees of C_3.

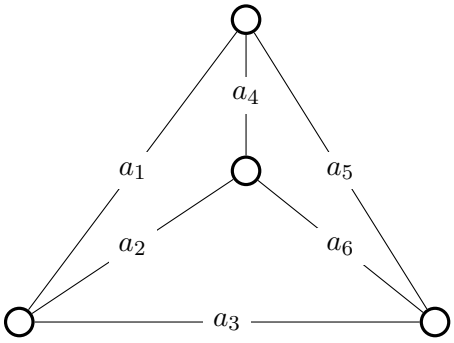


Figure 2.4: The complete graph K_4, with edges labelled.

Example 2.14 (Kirchhoff polynomial of K_4)**.** The graph of K_4 with edges labelled is shown in Figure 2.4. Consider that any triple of edges adjacent to the same vertex, such as a_1, a_2, a_3, will not appear in the Kirchhoff polynomial. That is because the compliment, which in our example will be a_4, a_5, a_6, would be a cycle. On the other hand, we cannot pick a triple of edges that is disconnected, since 3 distinct edges will be adjacent to a total of at least 4 vertices. Therefore, any triple of edges whose compliment is not a cycle will appear in the Kirchhoff polynomial, since the compliment will be a spanning tree. Thus, the Kirchhoff polynomial is given by

$$\Psi_{K_4} = \sum_{T} \prod_{e \notin E(T)} a_e = \sum_{1 \le i < j < k \le 6} a_i a_j a_k - (a_1 a_2 a_3 + a_1 a_4 a_5 + a_2 a_4 a_6 + a_3 a_5 a_6).$$

Another important property of a graph is vertex-transitivity and similar vertices. We present both definitions below.

Definition 2.15 (Similar vertices)**.** Let G be a graph with (at least) 2 distinct vertices u, v. We say that u and v are *similar* if there exists an automorphism $f : G \to G$ with $f(u) = v$.

Definition 2.16 (Vertex-transitivity)**.** Let G be a graph. We say that G is *vertex-transitive* if its vertices are pairwise similar. That is, for any $u, v \in V(G)$, there exists an automorphism $f : G \to G$ with $f(u) = v$.

2.3 The c_2-invariant

We have referred to Daniel Bump's "Algebraic Geometry", [8], in the write-up of this section. The definitions presented here will be helpful in defining the c_2-invariant.

Definition 2.17 (Finite Field $\mathbb{F}_q$)**.** Let q be a prime power. We let $\mathbb{F}_q$ denote the unique finite field with q elements.

Definition 2.18 (Affine Variety)**.** Let $f_1, \cdots, f_m \in k[x_1, \cdots, x_n]$. The *affine variety* of $f_1, \cdots, f_m$ is defined as

$$V(f_1, \cdots, f_m) := \{x = (x_1, \cdots, x_n) : f_1(x) = f_2(x) = \cdots = f_m(x) = 0\}.$$

Definition 2.19 (Point Count)**.** Let $f_1, \cdots, f_m \in \mathbb{F}_q[x_1, \cdots, x_n]$ for some prime power q. We define the *point count* of the variety of $f_1, \cdots, f_m$ over $\mathbb{F}_q$ to be the number

$$[f_1, \cdots, f_m]_q := |V(f_1, \cdots, f_m)|.$$

We now define the c_2-invariant. The Kirchhoff polynomial Ψ_G is defined in Definition 2.12 in page 7. The c_2-invariant as presented below is well-defined as long as the graph G has at least 3 vertices. This is ensured by Proposition 2 in [4, p3], which roughly states that p^2 divides $[\Psi_G]_p$.

Definition 2.20 (c_2-invariant)**.** Let G be a graph with at least 3 vertices. The *c_2-invariant* is the sequence over the primes, where the term at prime p is given by

$$c_2^{(p)}(G) = \frac{[\Psi_G]_p}{p^2} \mod p.$$

The Chevalley-Warning Theorem is helpful for exploiting some of the combinatorial structure of the c_2-invariant [4, p5]. We present Ax's proof in [1], using some arguments found in lecture notes of Clark [9].

Theorem 2.21 (Chevalley-Warning. [9])**.** *Let $\mathbb{F}_q$ be the finite field of q elements, and let $f_1, \cdots, f_r \in \mathbb{F}[x_1, \cdots, x_n]$ be polynomials with zero constant terms, with corresponding degrees $d_1, \cdots, d_r \in \mathbb{N}$ such that $d_1 + \cdots + d_r < n$. Then,*

$$[f_1, \cdots, f_r]_q \equiv 0 \mod q.$$

Proof. Define the following function f for $x \in \mathbb{Z}_q$:

$$f(x) := \prod_{i=1}^{r} (1 - f_i(x)^{q-1}).$$

Consider that $f(x) = 0$ if $f_i(x) \neq 0$ for some $1 \leq i \leq r$, and $f(x) = 1$ if $f_i(x) = 0$ for all $1 \leq i \leq r$. Hence, the function

$$N(f) := \sum_{x \in \mathbb{F}_q^n} f(x) = \sum_{x \in \mathbb{F}_q^n} \prod_{i=1}^{r} (1 - f_i(x)^{q-1}).$$

counts the number of zeroes of $f_1, \cdots, f_r$ in $\mathbb{F}_q^n$. We want to show that $N(f) \equiv 0 \mod q$.

Suppose u is some monomial vector, and assume first that $u_i = m_i(q-1)$ for some $m_i \in \mathbb{N}\ \forall 1 \leq i \leq n$. Then,

$$\sum_{x \in \mathbb{F}_q^n} x^u = \prod_{i=1}^{n} \sum_{x_i \in \mathbb{F}_q} x_i^{u_i} = \prod_{i=1}^{n} (-1) = (-1)^n.$$

Now, assume that for some $1 \leq m \leq n$, u_m is not a positive multiple of $q-1$, and let k be a generator of the cyclic group $\mathbb{F}_q^* := \mathbb{F}_q \setminus \{0\}$. Let $a = k^{u_m}$. Then,

$$\sum_{x_m \in \mathbb{F}_q} x^{u_m} = 0 + \sum_{x_m \in \mathbb{F}_q^*} x^{u_m} = \sum_{i=0}^{q-2} (k^i)^{u_m} = \sum_{i=0}^{q-2} a^i = \frac{1 - a^{q-1}}{1 - a} = 0,$$

and so

$$\sum_{x \in \mathbb{F}_q^n} x^u = \prod_{i=1}^{n} \sum_{x_i \in \mathbb{F}_q} x_i^{u_i} = 0.$$

Now, consider that

$$\deg f(x) \leq \sum_{i=1}^{r} (q-1)d_i < (q-1)n,$$

which implies that for every monomial vector u in f, $u_i < q-1$ for some $1 \leq i \leq n$, and so

$$\sum_{x \in \mathbb{F}_q^n} x^u = 0.$$

for every monomial vector. Hence, $N(f) \equiv 0 \mod q$, as desired. □

The following result is a known corollary of Theorem 2.21, a short proof of which can be found in [17].

Corollary 2.22. *Suppose $f \in \mathbb{Z}[x_1, \cdots, x_n]$ is of degree n. For any prime p, the coefficient of $\prod_{1 \leq i \leq n} {x_i}^{p-1}$ in f^{p-1} is $[f]_p \mod p$.*

2.4 ϕ^4-Theory

For an introduction to quantum field theory, we refer the reader to Ryder's "Quantum Field Theory" [13].

Suppose we conduct an experiment, where we input some known particles into some playground in which they might interact, and then measure the output particles, if any. Even with the input and output particles being known, we do not know the collisions that might have occurred. We could, however, take a sum over all the ways that the collisions could have happened. If we were to draw the propagating particles as half-edges, paired up into edges within the process, and the interactions between particles as vertices, then a particular set of collisions could be interpreted as a time-dependent graph. Forgetting this time-dependence produces the graphs known as Feynman graphs. [18, p5]

Quantum field theory (QFT) is the combination of quantum theory and field theory. A QFT describes interactions between particles, and the types of particles being considered are determined by the QFT in use. The Feynman graphs of a particular QFT are determined by the allowable half-edge types. For instance, QED has 3 half-edge types, a half-proton, a forward half-electron and a backward half-electron. In ϕ^4 theory, an example of a scalar QFT, there is only one half-edge type, resulting in one edge type and one vertex type.

Quantum electrodynamics (QED), first conceived in 1950, is an attempt to describe and explain electrodynamic phenomena, and as such it has been hugely successful. For example, QED predicted the magnetic moment of an electron quite accurately [13, 1.1]. In QED, the interacting particles are photons, electrons and positrons [16, p12].

The scalar ϕ^4-theory is the theory that is relevant in the rest of this document. The most important Feynman graphs of ϕ^4-theory are 4-regular graphs with a vertex removed, say v, with half-edges, known as *external edges*, incident to the vertices that were adjacent to v. These graphs are called 4-point graphs in ϕ^4-theory, referring to the four external edges. This justifies the following definition, which is illustrated in Figure 2.5.

Definition 2.23 (Completion/Decompletion)**.** Let G be a 4-regular graph. A *decompletion* H of G is formed by removing a vertex and the edges adjacent to it from G. In this case, G is called the *completion* of H.

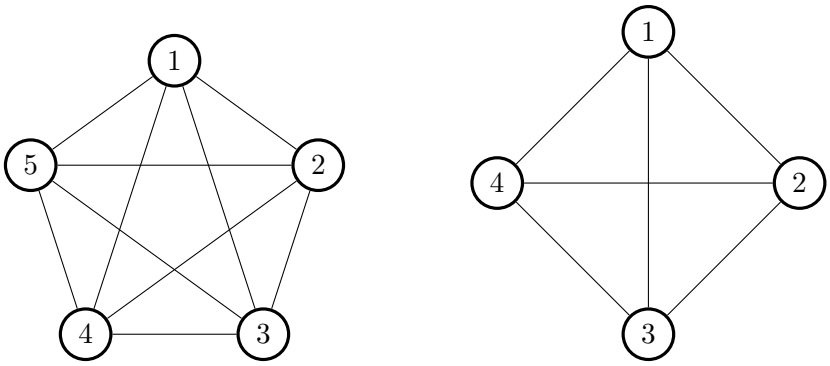


Figure 2.5: The complete graph K_5, and its decompletion, K_4. In general, a 4-regular graph might have several non-isomorphic decompletions.

When predicting the output particles of a collision experiment, given a certain input, we take a weighted sum over all the possible ways that these particles could have interacted, the Feynman graphs. The weight of a Feynman graph is its *Feynman integral*, which is determined by the *Feynman rules* given by the QFT in use [18, p5-6]. We say that the Feynman graph γ is *divergent* if its Feynman integral is divergent, and we further say that γ is *primitive divergent* if no proper subset of its integration variables is divergent. Although the *superficial degree of divergence* of a Feynman graph has a more general definition (for example, see [18, p39]), in the case of graphs in ϕ^4 theory, a graph γ is divergent if

$$4\ell(\gamma) - 2|\mathrm{E}(\gamma)| \geq 0,$$

where $\ell(\gamma)$ is the first Betti number, defined by

$$\ell(\gamma) := |\mathrm{E}(\gamma)| - |\mathrm{V}(\gamma)| + \text{number of components},$$

This is equivalent to the graph condition that γ's completion G is *internally 6-edge-connected*, the requirement that any 5-edge cut of G separates at most a single vertex from the rest of the graph [18, p87]. This is stated in the following theorem, a proof of which can be found in [18, p87].

Theorem 2.24. *Let γ be a graph in ϕ^4 theory. The graph γ is divergent if and only if it has at most 4 external edges.*

We would like to define a quantity on a 4-regular graph G known as the *Feynman period*. The convergence of this quantity, which is an integral, is guaranteed if G is *completed primitive*, which is equivalent to G being internally 6-edge-connected [14, p10]. Schnetz proved in [14, p10] that G is completed primitive if and only if $G - v$ is primitive, where v is any vertex in G. We can then define the Feynman period, justifed by the following theorem.

Definition 2.25 (Feynman period)**.** Let G be a 4-regular graph with loop number ℓ. We define the period P_G of G as the multiple integral

$$P_G = \int_{\alpha_i \geq 0} \frac{\Omega(\alpha)}{\Psi^2_{G-v}(\alpha)},$$

where v is some vertex of G, Ψ_{G-v} is the Kirchhoff polynomial of $G - v$ and

$$\Omega(\alpha) = \sum_{i=1}^{|E(G)|} \prod_{j \neq i} \mathrm{d}\alpha_j.$$

Theorem 2.26 (Schnetz. [14, p11])**.** *Let G be a 4-regular graph. Then P_G exists (that is, the integral converges) if and only if G is completed primitive.*

The period has many symmetries. The period of a primitive 4-regular graph is completion/decompletion invariant. The period is equal for planar decompleted duals, and is invariant under an operation on 4-regular graphs with 4-vertex cuts known as the Schnetz twist. [18, p93-5]

We end this section by re-presenting the definition of the c_2-invariant (Definition 2.20), followed by a proposition stating some trivial cases for the c_2-invariant. A graph G is said to be 2-*vertex-reducible* if $G - \{u, v\}$ is disconnected for some distinct vertices u, v [4, p15].

Definition 2.27 (c_2-invariant)**.** Let p be a prime. Let $[f]_p$ denote the point count of the affine variety of f over $\mathbb{F}_p$, the finite field with characteristic p. The c_2-invariant is the sequence over the primes, where the term at prime p is given by

$$c_2^{(p)}(G) = \frac{[\Psi_G]_p}{p^2} \mod p.$$

Proposition 2.28 (Trivial c_2-invariant. [4, p15])**.** *Let γ be a decompleted graph. If γ is 2-vertex-reducible or contains a doubled edge, then $c_2^{(p)}(G) \equiv 0 \mod p$ for all primes p.*

Chapter 3

Double Triangle Expansion

In this section, double triangle expansion (DTE), and its inverse operation, double triangle reduction (DTR), are discussed. Throughout this section, unless otherwise stated, G is a 4-regular graph with loop number $\ell \leq |\mathrm{V}(G)| \geq 5$. We begin by defining DTE and DTR. The operations are illustrated in Figure 3.1 and Figure 3.6, and an example DTE is shown in Figure 3.2. Although DTR can be defined more generally, we restrict it to the case of proper double triangles (Definition 3.2). This was done in order to simplify the expressions for Proposition 3.9 and Theorem 3.10.

Definition 3.1 (Double-Triangle Expansion)**.** Let $T = (v_1, v_2, v_3)$ be a triangle in a graph G. Let $v_4 \in \mathrm{N}_G(v_2) \setminus \{v_1, v_3\}$, and let $e = (v_2, v_4)$. In *double triangle expansion*, a new graph $H := \mathrm{DTE}_G(T, e)$ is created, by adding a new vertex v_5, removing the edges (v_1, v_3) and (v_2, v_4), and adding the edges (v_1, v_5), (v_2, v_5), (v_3, v_5) and (v_4, v_5).

Definition 3.2 (Proper double triangle)**.** A double triangle D in a graph G is said to be *proper* if D is not part of any triple triangles in G. If G has no proper double triangles, then we say that G is *double-triangle-free*.

Definition 3.3 (Double Triangle Reduction)**.** Let $D = (v_1, v_2, v_3, v_4)$ be a proper double triangle ((v_1, v_2, v_3) and (v_2, v_3, v_4) are triangles) in a graph G. In a new graph $H := \mathrm{DTR}_G(D)$, we identify the vertices v_2 and v_3, discarding any repeated edges, and add the edge (v_1, v_4), producing a new triangle (v_1, v_2, v_4). This process is called a *double triangle reduction*.

Definition 3.4 (Parent/Child)**.** Let G be a graph, and H be G after one DTR is done. We call H a *parent* of G, and G a *child* of H.

The graph H that results from expanding a triangle in a graph G depends on the choice of triangle, the choice of special vertex, and the choice of a neighbour of the special vertex. In case of dealing with unlabelled 4-regular graphs, the choice of neighbour turns out to be inconsequential, as the next proposition shows.

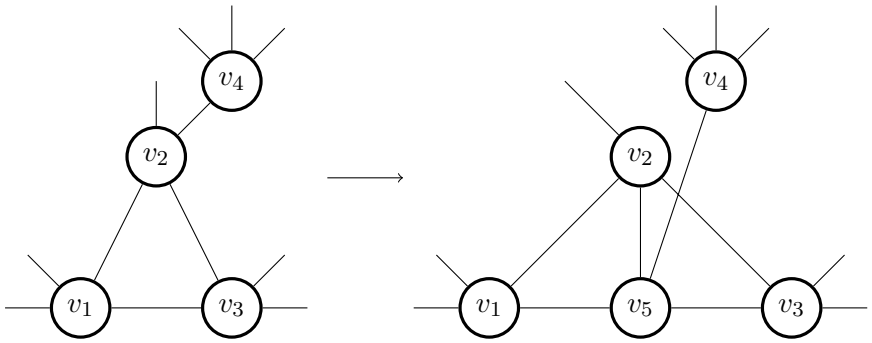


Figure 3.1: The operation DTE$((v_1, v_2, v_3), (v_2, v_4))$ is shown here.

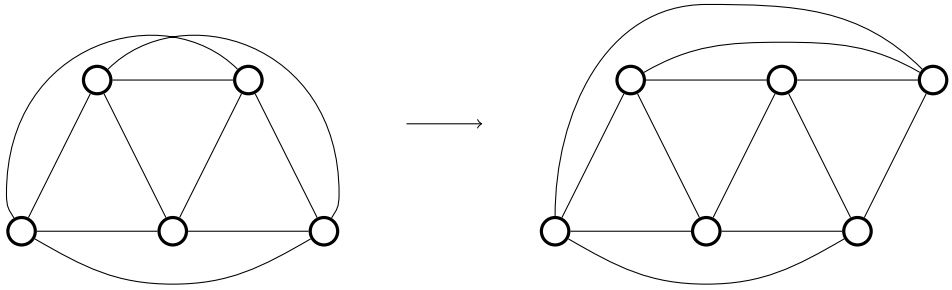


Figure 3.2: The graph $Z_3 \cong K_5$ and Z_4 are shown here. Z_4 can be obtained from Z_3 by DTE of any triangle.

Proposition 3.5. *Let (v_1, v_2, v_3) be a triangle in a 4-regular graph G. Let $\{v_4, v_5\} = N_G(v_2) \setminus \{v_1, v_3\}$. Let H_v denote the graph produced by expanding the triangle (v_1, v_2, v_3), with v_2 as the special vertex, and v the neighbour of v_2. Then,*

$$H_{v_4} \cong H_{v_5}.$$

Proof. Let v_6 be the vertex created by the DTE. Define $\phi : V(H_{v_4}) \to V(H_{v_5})$ with $\phi(v_2) = v_6$, $\phi(v_6) = v_2$, and $\phi(v) = v$ otherwise. We claim that ϕ is an isomorphism. By construction, ϕ is a bijection. To show that ϕ preserves adjacencies, we only have to consider the vertices $v_1, \cdots, v_6$, and the result immediately follows from Definition 3.1. See Figure 3.3. □

DTE can also be thought of in other ways. We can define DTE as taking a triangle T and an edge e not in T but incident to a vertex of T, subdividing e and the edge in the triangle that is opposite to e, and then identifying the resultant degree 2 vertices. This is shown in Figure 3.4. Another definition is taking a triangle and a crossing at some edge of the triangle by an edge, that is not in the triangle and is adjacent to the vertex opposite to the triangle edge, and turning the crossing into a degree 4 vertex. This is shown in Figure 3.5.

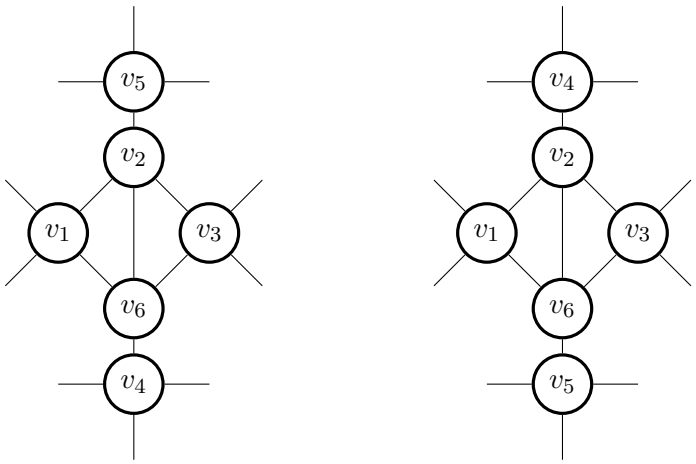


Figure 3.3: This figure illustrates the proof of Proposition 3.5. The graphs shown here, from left to right, are H_{v_4} and H_{v_5}.

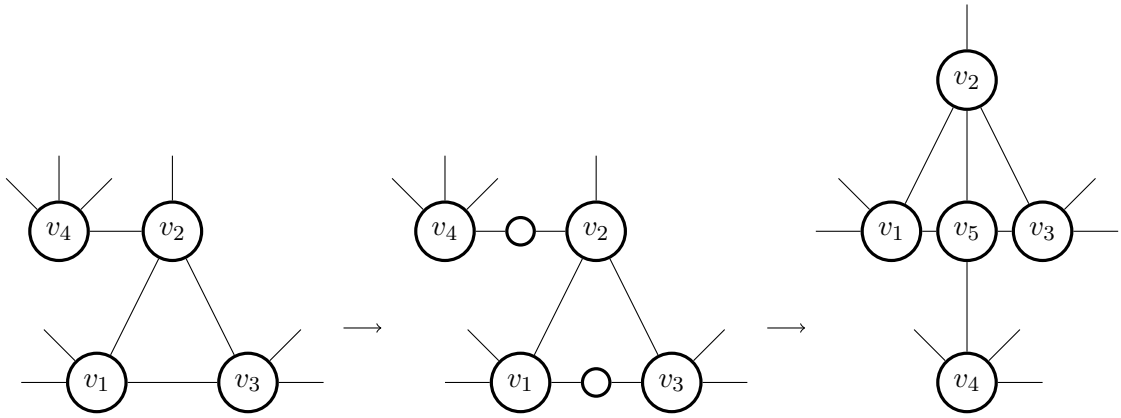


Figure 3.4: A DTE can be done by subdividing an edge not in the triangle and the opposite edge in the triangle, and then identifying the degree 2 vertices.

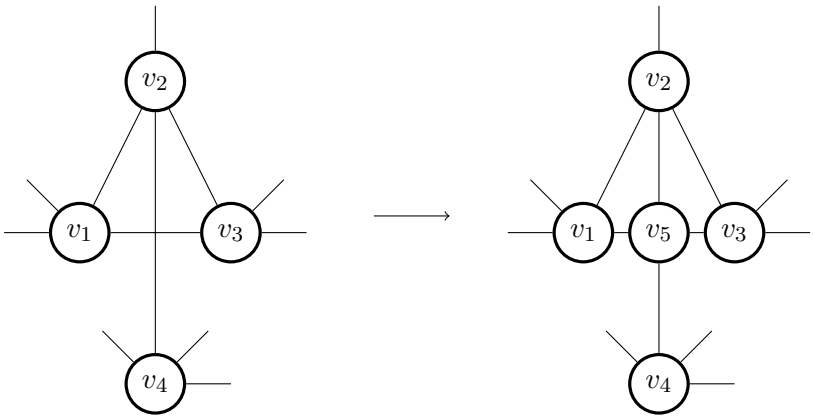


Figure 3.5: A DTE can be done by turning a crossing at a triangle edge into a new vertex.

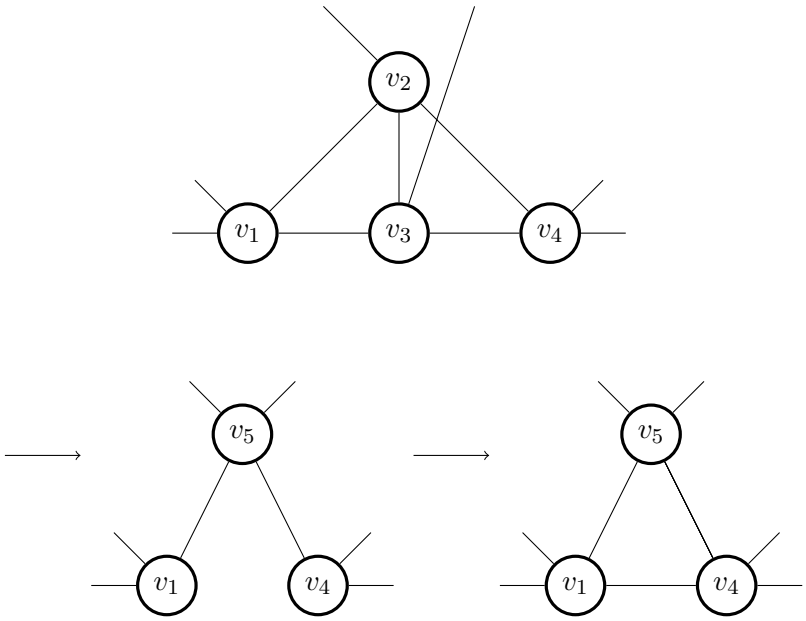


Figure 3.6: The operation DTR$((v_1, v_2, v_3, v_4))$ is shown here. In the intermediate step, vertices v_2, v_3 are identified to create v_5.

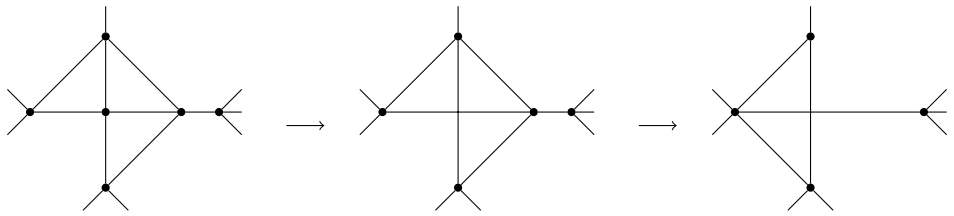


Figure 3.7: This figure shows 2 DTR's done on two double triangles that share a triangle. Notice that the order of the DTR's doesn't matter in this case. In general, DTR's commute in the sense that if a completed primitive graph G has distinct double triangles D_1 and D_2, then DTR of D_1 and D_2 in either order results in the same graph.

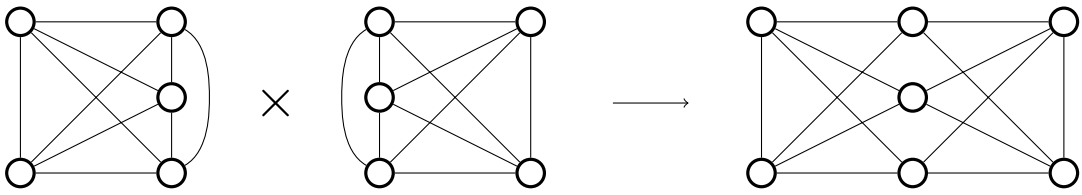


Figure 3.8: If G is the graph on the right, then $G = K_5 \times K_5$. In this case, K_5 is a product split of G.

DTR and DTE have other interesting properties. In [14], Schnetz proved that DTR's preserve completed primitiveness, and are commutative (see Figure 3.7) if the graph is completed primitive. DTR's also commute with an operation called the product split, which is defined in Definition 3.6. Schnetz also proved that if a completed primitive graph G has vertex-connectivity 3, then it is the product of two completed primitive graphs. Finally, it follows that, since DTR's and product splits commute, any sequence of DTR's and product splits terminate at a unique ancestor. An example of a product split is shown in Figure 3.8. This product is also known as a 3-sum.

Definition 3.6 (Product split. [14, Theorem 2.10, p14])**.** Suppose a completed primitive graph G can be obtained by identifying two triangles from two completed primitive graphs G_1, G_2, respectively, and removing the triangle edges. Then, either of G_1, G_2 is a *product split* of G, and G is a *product* of G_1, G_2.

Definition 3.7 (Reducible graph. [14, Definition 2.9, p14])**.** A completed primitive graph is *reducible* if it has vertex-connectivity 3, and is *irreducible* if otherwise.

Theorem 3.8 (Schnetz. [14, Theorem 2.10, p14])**.** *Let G be a reducible completed primitive graph. Then, G is the product of two completed primitive graphs G_1, G_2. Furthermore, the period of G is the product of the periods of G_1, G_2.*

Proposition 3.9 (Schnetz. [14, Proposition 2.19, p20-1])**.** *Suppose that G is a child of H. Then G is completed primitive if and only if H is completed primitive.*

Theorem 3.10 (Schnetz. [14, Definition 2.23, p22])**.** *Let G be a completed primitive graph. DTR's and product splits of G commute, and any sequence of DTR's and product splits terminates at a product of double-triangle-free irreducible graphs, A.*

Definition 3.11 (Descendant/Ancestor/Family)**.** Let G be a completed primitive graph. Let A be the graph(s) that is the termination of a sequence of DTR's and product splits on G as in Theorem 3.10. We call G a *descendant* of A, and A the *ancestor* of G. The set of descendants of the ancestor A is called the *family* of A.

To find the ancestor of a family, one could start with any member of that family, and perform DTR's and product splits until unable to do so. The remaining graph(s) is the ancestor of the family. Note that the ancestor is not necessarily one graph, but it is, in the most general case, a multiset. For instance, the ancestor of $K_5 \times K_5$, shown in Figure 3.8, is the multiset $\{K_5, K_5\}$.

Since K_5 has no proper double triangles, no further DTR's can be done. Since it has no non-trivial 3-vertex cut, it cannot be (non-trivially) product split. Thus, K_5 is the ancestor of its family. K_5-descendants are studied in more detail in Chapter 4.

Roughly speaking, a Feynman integral is said to be *linearly reducible* if, for some ordering of the edges of the Feynman graph, the integral in parametric form can be evaluated iteratively using a class of functions known as multiple polylogarithms [12]. The evaluation of a Feynman integral can be done through an algorithm that Brown constructed in [3]. Linear reducibility was proven by Brown in [3] to be minor-closed, and in [12], Moore and Yeats study the structure of linearly reducible Feynman graphs.

The *transcendental weight* of a number is very roughly the minimum number of nested integrals needed to write an integral expression for that number, with an argument that is a rational expression with rational coefficients, and rational integral bounds. The maximal (transcendental) weight (of the period) of a graph G in ϕ^4-theory with loop number ℓ is $2\ell - 3$. The graph G is said to have *weight drop* if the weight of its period is less than its maximal weight. [7]

One of the earliest occurrences of DTE is Theorem 35 of a paper by Brown and Yeats ([7, p23]), which states that DTE's preserve linear reducibility and weight drop.

Theorem 3.12 (Brown and Yeats. [7, Theorem 35, p23])**.** *Let G' be the graph G after one DTE. Then G is linearly reducible if and only if G' is linearly reducible, and G has weight drop if and only if G' has weight drop.*

It can also be shown that DTE's preserve the c_2-invariant, and this result appears as Corollary 34 of a paper by Brown and Schnetz ([4, p16]).

Theorem 3.13 ([4, Corollary 35, p16])**.** *Let G_2 be the graph G_1 after one DTE. Let H_1 and H_2 be any decompletions of G_1 and G_2, respectively. Then, for all primes p,*

$$c_2^{(p)}(H_1) \equiv c_2^{(p)}(H_2) \mod p.$$

It is known that $c_2(K_4) = (-1, -1, \cdots)$ (See, for example, Yeats' proof in [17, Proposition 3.1, p6].). By Theorem 3.13, if G is a decompletion of some double-triangle descendant of K_5, then $c_2(G) = (-1, -1, \cdots)$. Interestingly, all currently known graphs that have this c_2-invariant happen to be double-triangle descendants of K_5. In a conjecture due to Brown and Schnetz, that is always the case.

Conjecture 3.14 (Brown and Schnetz. [5, Conjecture 25, p16])**.** Let G be a completed primitive graph and let $\tilde{G}$ be any decompletion of G. Then,

$$c_2(\tilde{G}) = (-1, -1, \cdots) \iff G \cong K_5 \text{ or } G \text{ is a double-triangle descendant of } K_5.$$

Conjecture 3.14 suggests that there might be something unique to K_5-descendants. Studying their structure, then, would hopefully be a step in solving the conjecture, and might also shed some light on the c_2-invariant. In the next section, known results and new results about some K_5-descendants are presented. An encoding for K_5-descendants is constructed to aid in understanding their structure. A list of K_5-descendants up to order 10 is provided.

Chapter 4

K_5-descendants

In this chapter, we introduce K_5-descendants. We begin in Section 4.1 by listing all K_5-descendants of order ≤ 10. In Section 4.2, we introduce several classes of graphs that reappear throughout this chapter. In Section 4.3, we introduce an encoding of K_5-descendants, and we prove a small result about pseudo-descendants in Section 4.4. In Section 4.5, an enumeration of K_5-descendants based on their order and triangle counts is listed, and closed forms are found for several sequences from this enumeration. In Section 4.6, we prove that the number of triangles in a K_5-descendants is at least 4. For clarity, we present Definition 4.1 and Figure 4.1.

Definition 4.1 (K_5-descendant)**.** A graph G is a K_5-descendant if it is in the family of K_5. That is, G is a K_5-descendant if it can be double triangle reduced to K_5.

4.1 List of some K_5-descendants

In this section, we list all double triangle descendants of K_5 of order ≤ 10 in Figure 4.2. The graphs were generated using `createfigures.py`.

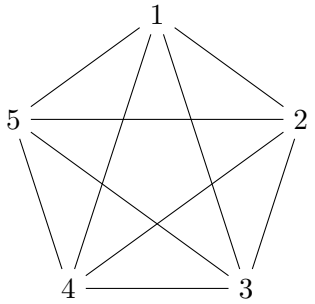
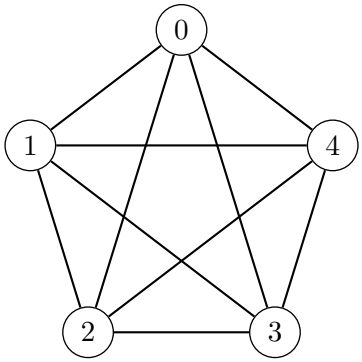
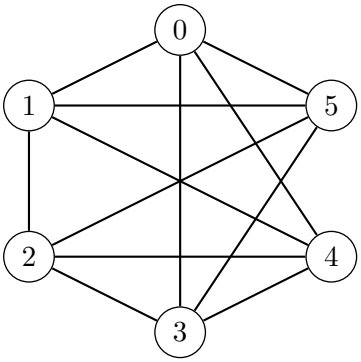
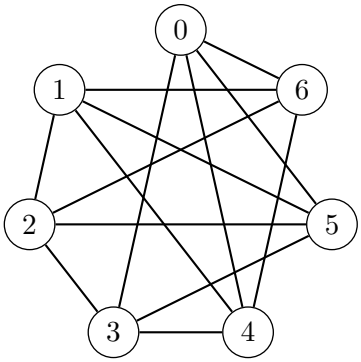
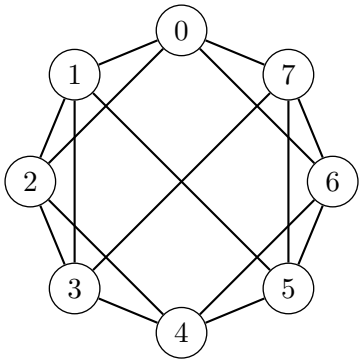


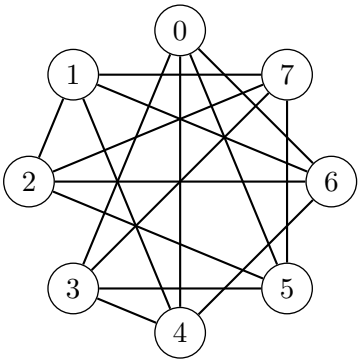
Figure 4.1: The complete graph on 5 vertices, K_5, is shown here.

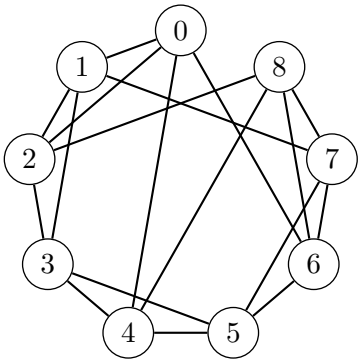

0
1
4
2
3

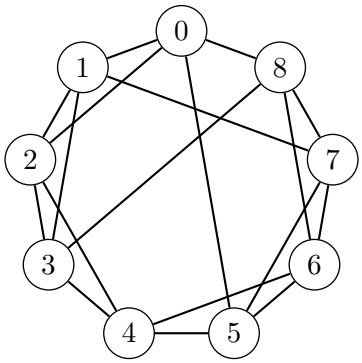

0
1
5
2
4
3

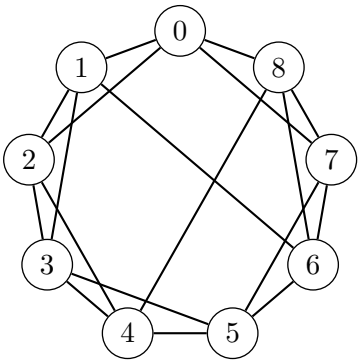

0
1
6
2
5
3
4

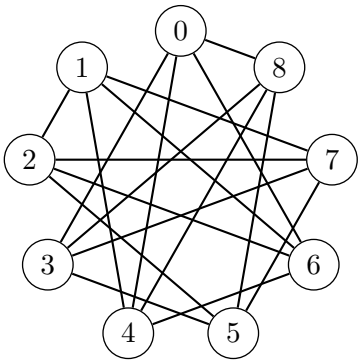

0
1
7
2
6
3
5
4

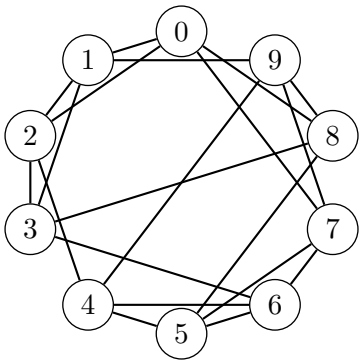

0
1
7
2
6
3
5
4

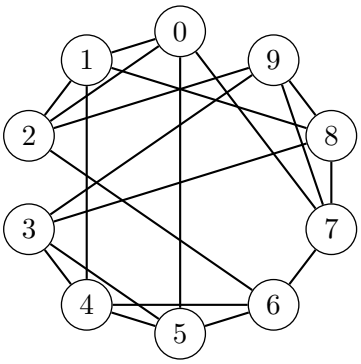

0
1
8
2
7
3
6
4
5

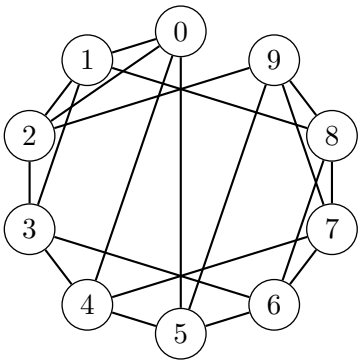

0
1
8
2
7
3
6
4
5


0
1
8
2
7
3
6
4
5


0
1
8
2
7
3
6
4
5


0
1
9
2
8
3
7
4
6
5


0
1
9
2
8
3
7
4
6
5


0
1
9
2
8
3
7
4
6
5

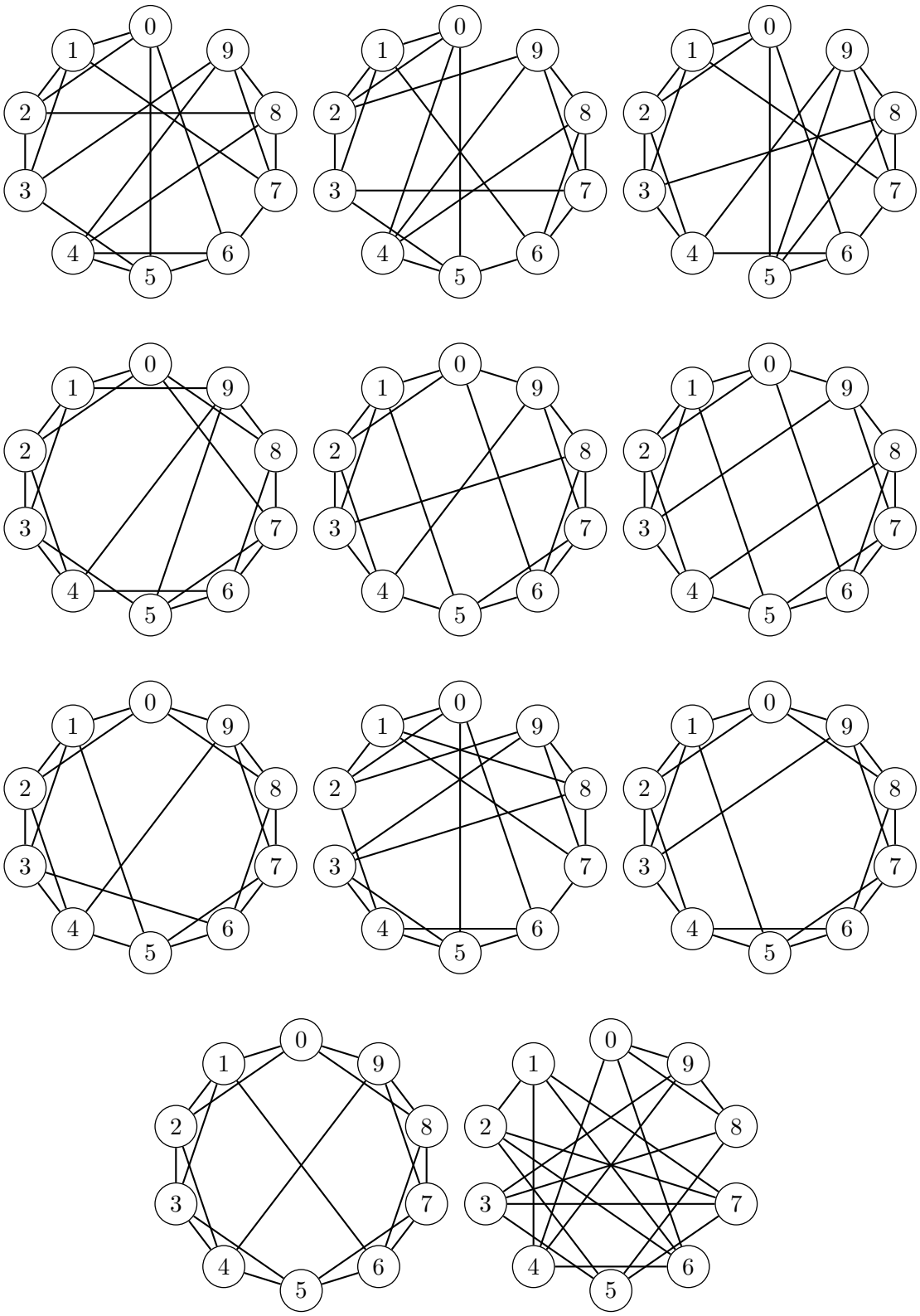


Figure 4.2: This figure lists all K_5-descendants up to order 10. The graphs are ordered first by the number of vertices, and then by descending level L.

4.2 Zigzags and Chains

We now introduce graphs that will appear in the later sections of this chapter. The zigzag graphs are a class of K_5 descendants that have been previously studied, such as in [17] and [6]. We make the generalization here from zigzags to n-zigzag graphs, as many of these graphs are K_5-descendants. In view of the generalization, we will refer to the zigzag graphs as 1-zigzag graphs and reserve the term zigzag alone to refer to the zigzag pieces making up the different n-zigzag classes. We begin with definitions of these classes of graphs, and we present some illustrating figures in 4.3, 4.4 and 4.5.

Definition 4.2 (Zigzag)**.** A *zigzag graph* Z_n^* is a graph whose vertices can be labelled $V(Z_n^*) = \{1, \cdots, n+2\}$ such that

$$\mathrm{E}(Z_n^*) = \{(i, i+1) : 1 \leq i \leq n+1\} \cup \{(i, i+2) : 1 \leq i \leq n\}.$$

Definition 4.3 (n-zigzag)**.** A graph G is a 1-*zigzag* if it is a zigzag with additional edges to make it 4-regular. G is a 2-*zigzag* if G consists of 2 zigzags sharing exactly 2 vertices, along with additional edges to make it 4-regular. For $n \geq 2$, G is an n-*zigzag* if G is 4-regular, and consists of n zigzags arranged in a circle, where each zigzag shares exactly one vertex with each of the previous zigzag and the next zigzag. G is a *proper* n-*zigzag* if G is an n-zigzag, but not an $(n-1)$-zigzag, or if G is a 1-zigzag.

We refer to the unique 1-zigzag with $n+2$ vertices as Z_n. The uniqueness is justified by the following proposition.

Proposition 4.4. *For $0 \leq n < 3$, there exist no 1-zigzag of order $n+2$. For $n \geq 3$, there is exactly one 1-zigzag of order $n+2$.*

Proof. Let G be a 1-zigzag of order $n+2$. By definition, G must be 4-regular, and hence $n \geq 3$. Z_n is produced by adding edges to Z_n^*. Consider that, using the labelling in Definition 4.2, all of the vertices in $V(Z_n^*)$ are of degree 4, except for the vertices $1, 2, n+1, n+2$. Since $\deg_{Z_n^*}(n+2) = 2$ and $(n+1, n+2) \in E(G)$, we must have that $(1, n+2), (2, n+2) \in E(G)$, This leaves vertex 1 and $n+1$ to be of degree 3, and every other vertex to be of degree 4. Hence, $(1, n+1) \in E(G)$, and now G is 4-regular. Since there was one way to do this, we have proved the proposition. □

It will be shown in Section 4.5 that K_5-descendants are either n-zigzags, or consist of one or more chains of zigzags, with 0 or more lone vertices that are not part of any triangle. We present the definition of zigzag chains and lone vertices. This is illustrated in Figure 4.6.

Definition 4.5 (Zigzag chain)**.** Let G be a graph. Suppose there is a sequence of pairwise disjoint sets $S_1, \cdots, S_n \subseteq V(G)$ satisfying the following:

- For all $1 \leq i \leq n$, $G[S_i] \cong Z^*_{m_i}$ for some $m_i \in \mathbb{N}$;
- For each i, S_i is maximal in the sense that for any $D \supsetneq S_i$, $G[D] \ncong Z^*_m$ for any $m \in \mathbb{N}$;
- For $i < j$ with $j - i = 1$, we have $S_i \cap S_j = \{v\}$ for some $v \in \mathrm{V}(G)$. Furthermore,

$$\deg_{G[S_i]} v = \deg_{G[S_j]} v = 2;$$

- The sequence of sets is maximal. That is, if S is the union of the S_i's, then, for any non-empty set S_{n+1} with $S_{n+1} \cap S = \emptyset$, the sequence of sets $S_1, \cdots, S_n, S_{n+1}$ does not satisfy all of the above conditions.

In this case, we call the sequence $S_1, \cdots, S_n$ a *chain*. Defining $s_i = |S_i| - 2$, we also sometimes call the sequence an $(s_1, \cdots, s_n)$-*chain*. The chain is *open* if there exist $v_1 \in S_1$ and $v_2 \in S_n$ such that

$$\deg_{G[S]} v_1 = \deg_{G[S]} v_2 = 2,$$

where S is the union of the S_i's. If $n = 2$, the chain is *closed* if $S_1 \cap S_2 = \{u, v\}$ with

$$\deg_{G[S_1]} u = \deg_{G[S_1]} v = \deg_{G[S_2]} u = \deg_{G[S_2]} v = 2.$$

For $n > 2$, the chain is *closed* if $S_1 \cap S_n = \{v\}$ where

$$\deg_{G[S_1]} v = \deg_{G[S_n]} v = 2.$$

In view of the above definition, an n-zigzag is a closed $(s_1, \cdots, s_n)$-chain for some $s_1, \cdots, s_n \in \mathbb{N}$, with additional edges to make it 4-regular.

Definition 4.6 (Lone vertex)**.** A vertex v in a graph G is said to be a *lone vertex* if it is not part of any triangles.

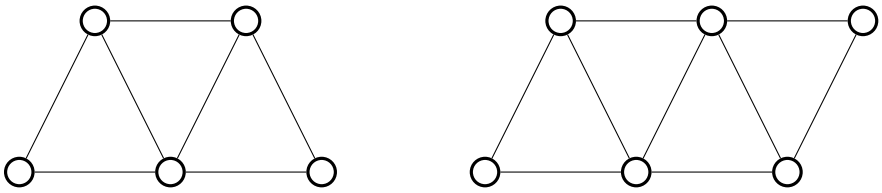


Figure 4.3: The zigzag graphs Z_3^* and Z_4^* are shown here.

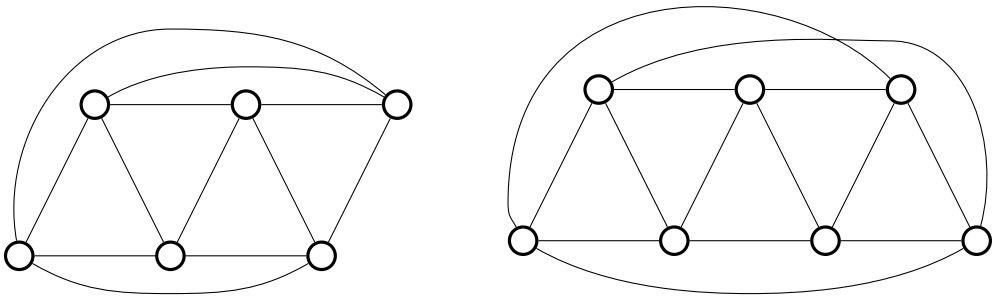


Figure 4.4: The 1-zigzag graphs Z_4 and Z_5 are shown here.

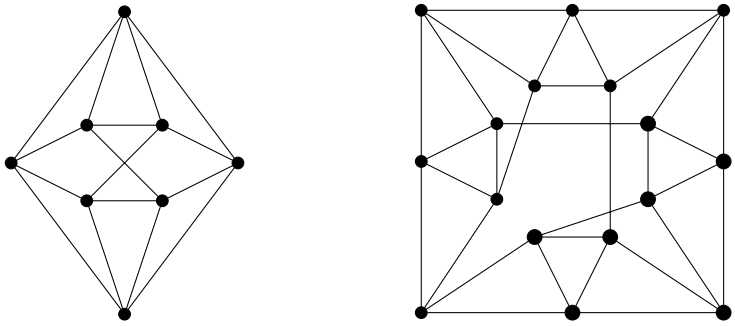


Figure 4.5: A proper 2-zigzag graph and a proper 4-zigzag graph are shown here.

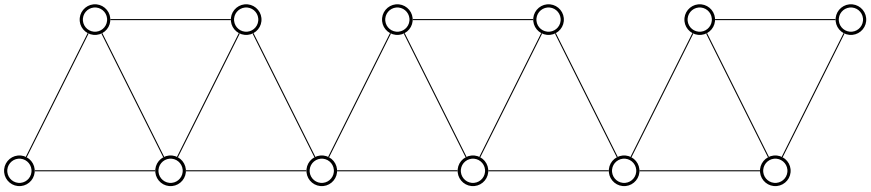


Figure 4.6: A $(3, 3, 2)$-chain is shown here.

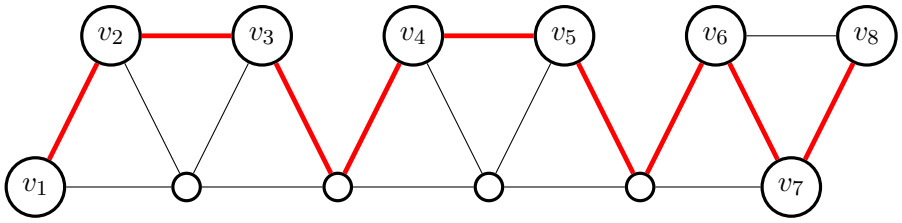


Figure 4.7: A $(3, 3, 2)$-chain is shown here. The outer vertices of the chain are ordered along some direction of the path indicated by the red/bold line.

4.3 Encoding of K_5-descendants

At first glance, K_5-descendants do not seem to have any obvious structure. As a first step in characterizing them, we introduce a set of two vectors that will represent their structure, which is, to the author's knowledge, have not been done for this class of graphs before. We will define a map that takes a K_5-descendant and outputs a set of vectors from which the graph can be reproduced. This representation has several advantages. It will allow easier tabulation of K_5-descendants, and quick calculations of some properties, such as the order, triangle count, and zigzag counts. The main potential benefit of the representation, however, is to obtain clues about the structure of the graphs through a different perspective.

The aforementioned vector representation does have several weaknesses. Perhaps the most glaring weakness is that not all graphs characterized by these vectors are descendants of K_5, and some collections of vectors are not even graphic. Another flaw is that it is not obvious or simple to perform double-triangle reductions or expansions on the vectors directly, since, from the way it's constructed, these operations can affect the vectors globally, even though the operations are local phenomena on the graph.

The problem of characterizing the descendants of K_5 thus becomes the problem of characterizing which collections of vectors are realizable as K_5-descendants. Another problem is discerning whether two collections of vectors are representations of isomorphic graphs, more efficiently than directly analysing the two graphs. In this section, we will introduce this vector representation, which is a set of two vectors, the chain vector and the chord vector.

Definition 4.7 (Chain vector of n-zigzags)**.** Let G be an n-zigzag graph. Then G is a closed $(z_1, \cdots, z_n)$-chain for some $z_1, \cdots, z_n \in \mathbb{N}$. The *chain vector* of G is defined as

$$\mathrm{C}(G) := [z_1, z_2, \cdots, z_n].$$

We call $V(G), \emptyset$ a *chain partition* of G, where $\emptyset$ is the empty set.

Definition 4.8 (Chain vector of general graphs)**.** Let G be a graph that is not an n-zigzag. Suppose that the vertices of G can be partitioned into sets $C_1, C_2, \cdots, C_r, L$ such that:

- For all i, $G[C_i]$ is an open c_i-chain, where $c_i = (z_1, \cdots, z_{k_i})$ for some $z_1, \cdots, z_{k_i} \geq 1$;
- L is the set of lone vertices of G. That is, for any $v \in L$, v is not part of any triangle in G. We let $l := |L|$.

The *chain vector* of G is defined as

$$\mathrm{C}(G) := [c_1, 0, c_2, 0, \cdots, c_r, 0, -l].$$

We call the sequence of sets $C_1, C_2, \cdots, C_r, L$ a *chain partition* of G.

Definition 4.9 (Special vertices of a chain)**.** Let C be a chain in a graph G. We define the following sets

- The set of *middle vertices*

$$M := \{v : v \text{ is part of exactly two zigzags}\}.$$

- The set of *internal vertices*

$$I := \{v : v \text{ is part of exactly one zigzag and } \deg_{G[C]}(v) = 4\}.$$

- The set of *chord vertices*

$$B = \{v : \deg_{G[C]}(v) = 3, \text{ or } \deg_{G[C]}(v) = 2 \text{ and } v \text{ is not an end vertex}\}.$$

- The set of *end vertices*

$$E = \{v : \deg_{G[C]}(v) = 2 \text{ and } v \text{ is adjacent to at most one middle vertex}\}.$$

- The set of *outer vertices*

$$U = B \cup E.$$

We define an *outer path* of C, $\mathrm{P}(C)$, as a path in C that starts on an end vertex of the first zigzag, ends on an end vertex of the last zigzag, and passes through all the vertices in B, E. If $U = \{v_1, \cdots, v_k\}$, we say that U is *ordered w.r.t* $P(C)$ if, for any $i < j$, v_i appears before v_j in $\mathrm{P}(C)$, with $\mathrm{P}(C)$ traversed in either direction. We note that although the outer path is not necessarily unique, the outer vertices are always in the same order (Figure 4.7).

Definition 4.10 (Chord vector)**.** Let G be a graph with a chain partition $C_1, C_2, \cdots, C_r, L$. Let $U_1, \cdots, U_r$ be the outer vertices of $C_1, \cdots, C_r$, respectively. We order the vertices of $U_i := \{v_1^i, \cdots, v_{m_i}^i\}$ along $\mathrm{P}(C_i)$. Let A be a set defined by

$$A := E(G) \setminus \bigcup_{i=1}^{r} E(C_i).$$

Label the edges in A in some order as $1, 2, \cdots, k$. Let $e_1^j, \cdots, e_{t_j}^j$ be the (labels of the) edges in A adjacent to v_j^i. The *partial chord vector* of C_i is given by

$$\mathrm{PCh}(C_i) := [e_1^1, \cdots, e_{t_1}^1, e_1^2, \cdots, e_{t_2}^2, \cdots, e_1^{m_i}, \cdots, e_{t_{m_i}}^{m_i}].$$

Order the vertices in L arbitrarily as $l_1, \cdots, l_n$. Let $f_1^j, \cdots, f_{n_j}^j$ be the edges adjacent to l_j. The *lone chord vector* is given by

$$\mathrm{PCh}(L) := [f_1^1, \cdots, f_{n_1}^1, f_1^2, \cdots, f_{n_2}^2, \cdots, f_1^n, \cdots, f_{n_n}^n].$$

The *chord vector* of G is given by

$$\mathrm{Ch}(G) := [\mathrm{PCh}(C_1), \cdots, \mathrm{PCh}(C_r), \mathrm{PCh}(L)].$$

Definition 4.11 (Vector representation)**.** Suppose a graph G has a chain vector $\mathrm{C}(G)$ and a corresponding chord vector $\mathrm{Ch}(G)$. A *vector representation* of G is given by the pair $\mathrm{C}(G), \mathrm{Ch}(G)$. An example is given in Figure 4.8.

As was previously mentioned, this vector representation is not unique to K_5-descendants. This justifies the following definition. An example is shown in Figure 4.9.

Definition 4.12 (Pseudo-descendant)**.** A graph G is a *pseudo-descendant* if it admits a vector representation. A *non-descendant* is a pseudo-descendant that is not a K_5-descendant.

We note that since the chain and chord vectors depend on the particular orderings on the vertices and edges of G, a graph G might have several distinct vector representations. We say that two vector representations are *equivalent* if they arise from isomorphic graphs. We would like to remove as many redundancies as possible from the definition of the vector representation, and so we will define a standard vector, based on the vector representation. We begin with preliminary definitions.

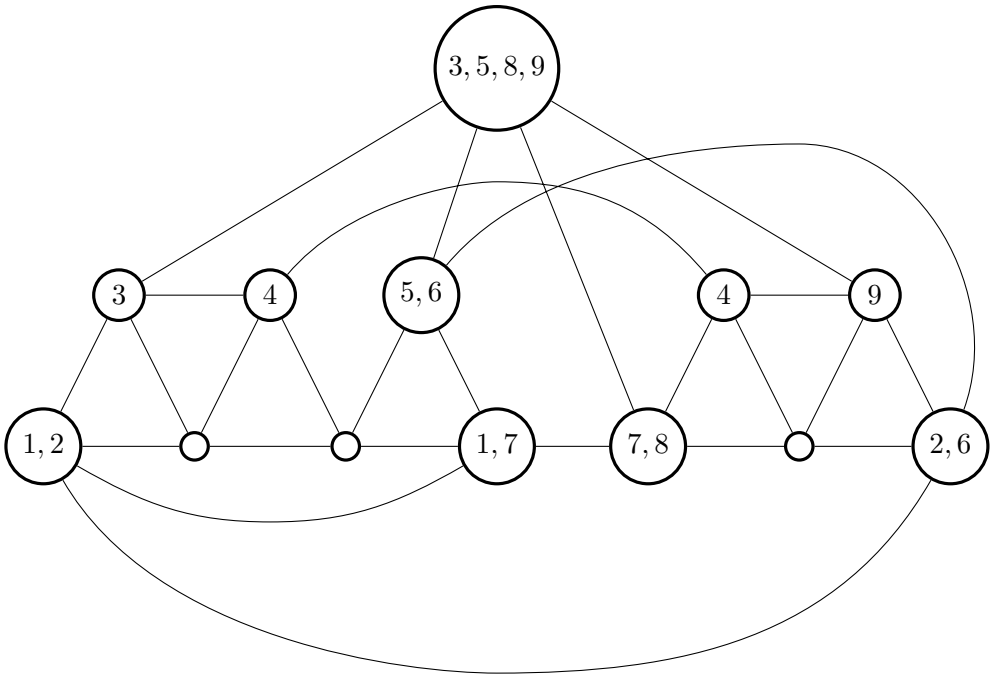


Figure 4.8: An example of a pseudo-descendant. Vertices that share the same label are adjacent. This graph has a vector representation $[(3,1,0,3,0,-1),(1,2,3,4,5,6,1,7,7,8,4,9,2,6,3,5,8,9)]$ and a standard vector $((3,1,0,3,0,-1),(4,8,8,4,6,5,1,4,2))$.

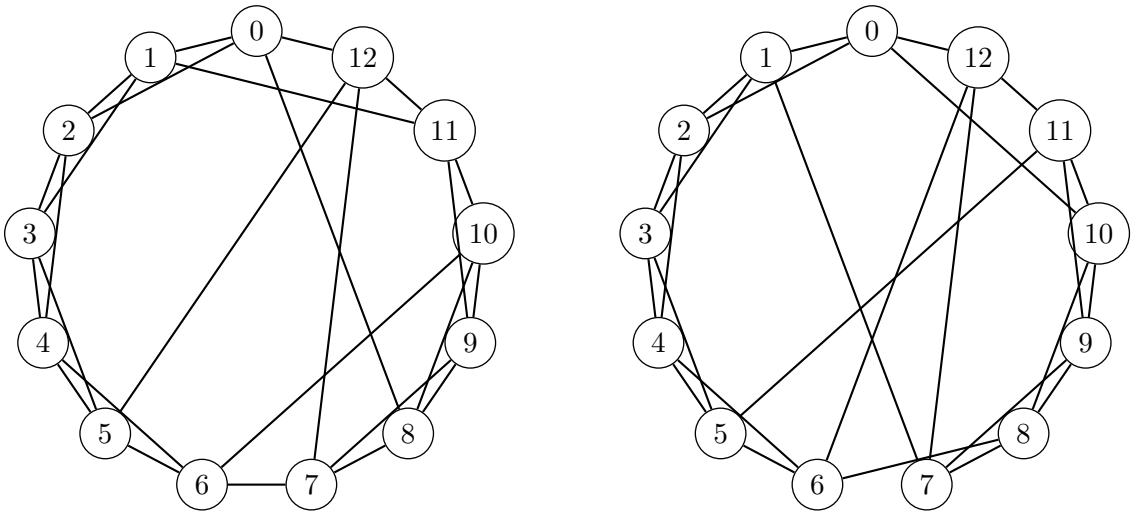


Figure 4.9: The graph on the left has a standard vector $((5,0,3,0,-1),(5,8,6,6,1,3,4,1))$ and is a descendant of K_5. The graph on the right has a standard vector $((5,0,3,0,-1),(6,8,3,5,2,5,4,1))$ and is a non-descendant. We call both graphs pseudo-descendants. In both graphs, the two zigzags are the vertices 0 through 6 and 7 through 11, respectively. The plots were generated using `pdplot(G,1,filename,graphicsize)`.

Definition 4.13 (Standard chain ordering)**.** Let G be a pseudo-descendant. Let $C_1, \cdots, C_r$ be the chains of G with corresponding chain sizes $c_1, \cdots, c_r$, with $c_i = z_1, \cdots, z_{n_i}$. Suppose that the following two conditions are satisfied

- For each i, $(z_1, \cdots, z_{n_i}) \geq_{\text{lex}} C^*$, for any cyclic permutation C^* of C_i or C_i^{-1}, where C_i^{-1} is the reverse of C_i;
- For $1 \leq i \leq r-1$, $c_i \geq_{\text{lex}} c_{i+1}$;

where $\geq_{\text{lex}}$ is the lexicographical orderding. In this case, we say that $C_1, \cdots, C_r$ are in *standard chain ordering.*

Definition 4.14 (Chord length vector)**.** Let G be a pseudo-descendant and let $C_1, \cdots, C_r$ be the chains of G in standard chain ordering. Let $l_1, \cdots, l_k$ be the lone vertices of G. We label the outer vertices of G from $1, \cdots, m$ in order of their appearance in the sequence $\mathrm{P}(C_1), \cdots, \mathrm{P}(C_r)$, where $\mathrm{P}(C_i)$ is the outer path, traversed from left to right on C_i. We label the lone vertices in some order as $m+1, \cdots, m+k$. We start with an empty list, and construct a *position list* PL iteratively as follows:

- If i, j are chord vertices in a 2-triangle zigzag, they will be placed together in a new position;
- If i is not a chord vertex of a 2-triangle zigzag, it is placed on its own in a new position.

Define the *position function*

$$\mathrm{Pos}(v) := \text{the position in PL where } v \text{ appears.}$$

Let H be G after all edges whose vertices are contained in a single chain are removed. Label the edge list as $e_1, \cdots, e_n$ ordered by the smaller Pos of their endpoints (and arbitrarily for edges with the same smaller endpoint). We define a *chord length vector* CLV of length n where, if $e_i = (u, v)$, the ith component of CLV is given by

$$\mathrm{CLV}_i = \mathrm{Pos}(v) - \mathrm{Pos}(u).$$

Definition 4.15 (Standard vector)**.** Let G be a pseudo-descendant and let $C_1, \cdots, C_r$ be the chains of G in standard chain ordering, with corresponding chain sizes $c_1, \cdots, c_r$. If G is a 1-zigzag, a *standard vector* of G is given by the pair $((c_1), (,))$. Let k be the number of lone vertices of G. Let V be a chord length vector of G. A *standard vector* of G is given by the pair

$$((c_1, 0, c_2, 0, \cdots, c_r, 0, -k), V);$$

unless G is an n-zigzag with $n > 1$, then a standard vector of G is given by the pair $((c_1), V)$.

Although the standard vector is a more condensed form of the vector representation, it still contains some redundancies. For a pseudo-descendant G, the chain vector part of any standard vector is the same, but the chord length vector is not necessarily unique. For instance, for the K_5-descendant in Figure 4.9, $((5,0,3,0,-1),(5,8,6,6,1,3,4,1))$ and $((5,0,3,0,-1),(8,5,6,6,1,3,4,1))$ are equivalent standard vectors. Factors that can affect the chord length vector are the ordering of chains with identical chain sizes, the ordering of the lone vertices, and the direction in which the outer path of the chains is taken. We could construct a unique representation of a pseudo-descendant if we further require that the chord length vector part is maximal in some ordering, and in the next definition, we do so with the lexicographical ordering.

Definition 4.16 (Ordered standard vector)**.** Let G be a pseudo-descendant and let $C_1, \cdots, C_r$ and $l_1, \cdots, l_k$ be the chains in standard chain ordering and lone vertices, respectively, of G. Let the corresponding chain sizes be $c_1, \cdots, c_r$. If G is a 1-zigzag, the *ordered standard vector* of G is given by $((c_1),(,))$. Otherwise, we reorder the chains and lone vertices such that:

- The chains are in standard chain ordering.
- The order produces the maximum chord length vector CLV in the lexicographical ordering.

If G is not an n-zigzag, then the ordered standard vector of G is given by the pair

$$((c_1, 0, c_2, 0, \cdots, c_r, 0, -k), \text{CLV});$$

If G is an n-zigzag with $n > 1$, then the ordered standard vector of G is given by the pair $((c_1), \text{CLV})$.

Although the ordered standard vector might have some theoretical potential, it is inefficient for practical calculations. If G has m distinct chain, each appearing n_m times, finding the standard chord vector will require $\prod_{i=1}^{n} n_i!$ standard chord vector calculations. As it turns out, for large orders, it is much slower to compare two graphs by calculating their ordered standard vectors and comparing them, than it is to simply check if the graphs are isomorphic. Thus, in searching for K_5-descendants by expanding all triangles and removing the isomorphic ones (as is done in `K5dsearch.py`), checking isomorphisms is much faster than calculating ordered standard vectors and checking equality, due to the large number of isomorphic graphs that are produced. The ordered standard vector representation is useful in that it turns the question of isomorphism of two pseudo-descendants into simply calculating their ordered standard vector representations, and comparing them, as the following proposition states.

Proposition 4.17. *Let G, H be pseudo-descendents, and let V_G, V_H denote the ordered standard vector representations of G and H, respectively. Then,*

$$G \cong H \iff V_G = V_H$$

Proof. ($V_G = V_H \implies G \cong H$) Consider that from the first part of the V_G, the chains of G can be constructed, and the outer vertices of each chain can be found and uniquely ordered. The second part of V_G constructs all the remaining edges and the lone vertices. Thus, V_G uniquely determines G, as desired.
($G \cong H \implies V_G = V_H$) Let $V_G^{(1)}$ and $V_G^{(2)}$ denote the first and second part of V_G, respectively. Consider that $V_G^{(1)}$ is constructed by taking the ordering and orientation of the chains of G that give the maximum in lexicographical ordering. Hence, the chain structure of G uniquely determines $V_G^{(1)}$. In constructing $V_G^{(2)}$, a finer ordering of the identical chains and of the lone vertices of G is chosen that produces the maximum chord length vector. Hence, $V_G^{(2)}$ is uniquely determined. □

Observe that if a graph is triangle-free, then it is a pseudo-descendant, but certainly not a K_5-descendant. A standard vector of that graph is simply a list of edges adjacent to each vertex.

4.4 Pseudo-descendant Parameters

We begin by defining some parameters that are well defined for pseudo-descendants.

Definition 4.18. Let G be a pseudo-descendant. By definition, G will consist of either a single closed chain, or no closed chains and some positive number of open chains. We let c, z, t, l, n denote the number of open chains, zigzags, triangles, lone vertices, and vertices, in G, respectively. We let z_k denote the number of zigzags containing exactly k triangles. We denote the number of middle, internal, boundary, and end vertices, respectively, by n_m, n_i, n_b, n_e, respectively. These quantities are defined in Definition 4.9 on page 28. A summary of these quantities is provided in Table 4.1.

While we don't expect to obtain a full characterization of K_5 descendants using the aforementioned quantities, relations between those quantities that are satisfied for pseudo-descendants can help in determining whether a graph is not a pseudo-descendant. A simple result in that direction is the following proposition.

Proposition 4.19. *If G is a pseudo-descendant, then $n = c + z + t + l$.*

Table 4.1: Summary of the psuedo-descendant quantities defined in Definition 4.18.

Quantity (number of)	**Symbol**
triangles	t
zigzags	z
open chains	c
k-triangle zigzags	z_k
vertices	n
middle vertices	n_m
internal vertices	n_i
boundary vertices	n_b
end vertices	n_e
lone vertices	l

Proof. Let G be a pseudo-descendant. We have that

$$n = n_m + n_i + n_b + n_e + l \tag{4.1}$$

Each open chain contributes two end vertices. Each zigzag of $k \geq 2$ triangles contributes 2 boundary vertices and $k-2$ internal vertices. Each 1-triangle zigzag contributes 1 boundary vertex. Thus,

$$n_e = 2c \tag{4.2}$$

$$n_b = \sum_{k\geq 2} 2 + z_1 \tag{4.3}$$

$$n_i = \sum_{k\geq 2} (k-2)z_k \tag{4.4}$$

Let the number of zigzags in a chain be denoted as $\mathrm{z}(C)$. A chain C contributes $\mathrm{z}(C) - 1$ middle vertices. Thus,

$$n_m = \sum_C (\mathrm{z}(C) - 1) = \sum_C \mathrm{z}(C) - \sum_C 1 = z - c \tag{4.5}$$

Combining the equations, we get

$$n = z - c + \sum_{k\geq 2} (k-2)z_k + \sum_{k\geq 2} 2 + z_1 + 2c + l = z + c + l + \sum_{k\geq 1} kz_k = z + c + l + t$$

□

Table 4.2: This table lists the number of K_5-descendants of a of order n and triangle count t. We start the count from $\text{tri}(G) = 4$, justified by the fact that any K_5-descendant has at least 4 triangles, a fact proved in Section 4.6. The enumeration was produced through the code in `K5dordertritable.py`.

n/t	4	5	6	7	8	9	10	11	12	13	14
5	0	0	0	0	0	0	1	0	0	0	0
6	0	0	0	0	1	0	0	0	0	0	0
7	0	0	0	1	0	0	0	0	0	0	0
8	0	0	1	0	1	0	0	0	0	0	0
9	0	1	1	1	0	1	0	0	0	0	0
10	1	2	6	2	2	0	1	0	0	0	0
11	3	8	19	15	4	2	0	1	0	0	0
12	8	37	88	76	34	7	3	0	1	0	0
13	21	147	390	435	218	61	10	3	0	1	0
14	67	550	1758	2405	1576	505	106	14	4	0	1

Table 4.3: This table lists a few terms of the counting sequences for K_5-descendants of levels 0 to 4. The counts start at $|\text{V}(G)| = 7$. The table also lists the denominator and numerator of the generating functions for each sequence. The generating functions for levels 0 to 3 were derived in propositions 4.25, 4.26, 4.27 and 4.30. The conjectured generating function for level 4 is also shown here.
In the row for $L = 4$, $f(x) = x^{20}-x^{19}+3x^{18}-3x^{17}+4x^{16}+4x^{14}+3x^{13}+6x^{12}+3x^{11}+4x^{10}+x^9$.

Level	**Sequence**	**Numerator**	**Denominator**
0	$1, 1, 1, \ldots$	x^7	$(1-x)$
1	$0, 0, 0, \ldots$	0	1
2	$0, 1, 1, 2, 2, 3, 3, \ldots$	x^8	$(1-x)^2(1+x)$
3	$0, 0, 1, 2, 4, 7, 10, 14, \ldots$	$(x^9 + x^{11})$	$(1-x)^3(1+x+x^2)$
4	$0, 0, 1, 6, 15, 34, 61, 106, \ldots$	$\stackrel{?}{=} f(x)$	$\stackrel{?}{=} (1-x)^4(1+x)(1+x+x^2+x^3)$

4.5 Order and Triangle Count

In this section, we enumerate K_5-descendants up to order 14, grouping them by their level, defined in Definition 4.20. Some interesting sequences arise when looking at the number of descendants of K_5 of certain levels. Table 4.2 lists the number of K_5-descendants grouped by their order and triangle counts, and Table 4.3 lists the first 5 level sequences. We derive the closed form of the sequences for $\mathrm{L}(G) = 0, 1, 2, 3$ in propositions 4.25, 4.26, 4.27, and 4.30. We begin by proving Lemma 4.24.

Definition 4.20 (Level)**.** Let G be a graph. We define the level of G to be

$$\mathrm{L}(G) = \mathrm{order}(G) - \mathrm{tri}(G).$$

Lemma 4.21. *If G is a K_5-descendant of order > 5, then neither K_4 nor the triple triangle T^3 are subgraphs of G.*

Proof. Observe that since K_5 is completed primitive, by Proposition 3.9, all K_5-descendants are completed primitive. Let G be a K_5-descendant of order > 5. If K_4 is a subgraph of G, then G has a non-trivial internal 4-edge-cut, and so it is not internally 6-edge-connected, and, by Theorem 2.24, G is not completed primitive, a contradiction.
Suppose now that T^3 is a subgraph of G. Then G has a 3-vertex-cut. By Theorem 3.10, we can find the ancestor of G by performing all double triangle reductions first. Since K_5 has no 3-vertex-cut, it cannot be the ancestor of G, a contradiction.

□

Lemma 4.22. *Let G be a K_5-descendant of order > 5, and let H be a child of G. Then,* $\mathrm{tri}(H) = \mathrm{tri}(G) + c$, *where $c \in \{-1, 0, 1\}$.*

Proof. Let v_1, v_2, v_3 be a triangle in G, and let $v_4 \in N(v_1) \setminus \{v_2, v_3\}$. In a double triangle expansion, WLOG, the edges $(v_1, v_4), (v_2, v_3)$ are removed, a new vertex v_5 is added, and the edges $(v_5, v_1), (v_5, v_2), (v_5, v_3), (v_5, v_4)$ are added. Let H denote G after this double triangle expansion. If any triangles not containing v_5 have been removed, they necessarily must contain both v_2, v_3, or both v_1, v_4. Consider that in $G[v_1, v_2, v_3, v_4]$ (the induced subgraph of G on v_1, v_2, v_3, v_4), $\deg(v_4) = 1$, $\deg v_2 = \deg v_3 = 2$, and $\deg v_1 = 3$. Hence, using Lemma 4.21, there is at most 1 additional triangle containing v_2, v_3 (other than v_1, v_2, v_3) in G, and at most one triangle containing v_1, v_4.

On the other hand, if new triangles are created, then these triangles must each contain one of the edges $(v_5, v_1), (v_5, v_2), (v_5, v_3), (v_5, v_4)$. Other than the double triangle (v_1, v_2, v_5, v_3), triangles would be created if G contains the edge (v_2, v_4) or (v_3, v_4) (G cannot contain both these edges as that would make a 4-clique, which is impossible due to Lemma 4.21). Suppose then that G contains (v_2, v_4). Double triangle expansion removes the triangle (v_2, v_3, v_4) by deleting (v_2, v_4), and creates the triangle (v_3, v_4, v_5). Thus, the net contribution of having the edge (v_2, v_4), and similarly for (v_3, v_4), is 0. It thus suffices to consider three cases.

Case 1: G has the triangles (v_2, v_3, a), (v_1, v_4, b), for some vertices $a, b \notin \{v_1, v_2, v_3, v_4\}$ and $a \neq b$, since $a = b$ implies G contains a 4-clique, which is not possible. Double triangle expansion thus removes a total of 3 triangles in G, and adds a double triangle. Hence, $\text{tri}(H) = \text{tri}(G) - 3 + 2 = \text{tri}(G) - 1$.

Case 2: G has the triangle (v_2, v_3, a) for some vertex $a \notin \{v_1, v_2, v_3, v_4\}$, and does not contain a triangle (v_1, v_4, b) for any $b \notin \{v_1, v_2, v_3, v_4\}$. Double triangle expansion removes a total of 2 triangles in G, and adds a double triangle. Hence, $\text{tri}(H) = \text{tri}(G)$.

Case 3: G has the triangle (v_1, v_4, a) for some vertex $a \notin \{v_1, v_2, v_3, v_4\}$, and does not contain a triangle v_2, v_3, b for any $b \notin \{v_1, v_2, v_3, v_4\}$. Double triangle expansion removes 2 triangles, and adds a double triangle. Hence, $\text{tri}(H) = \text{tri}(G)$.

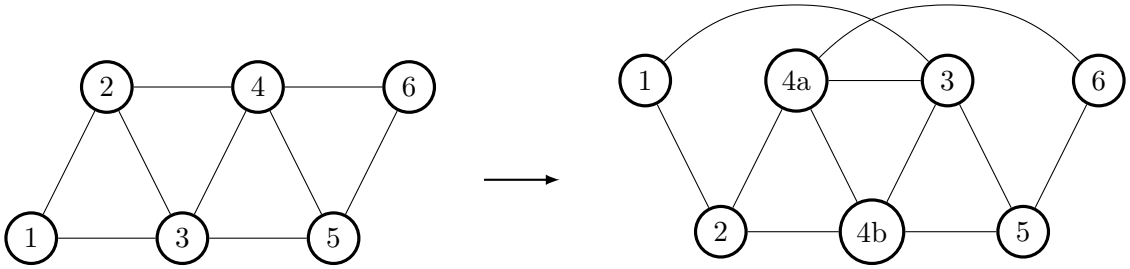
Case 4: G does not have a triangle (v_2, v_3, a) or (v_1, v_4, a) for any $a \in \{v_1, v_2, v_3, v_4\}$. Double triangle expansion removes 1 triangle, and adds a double triangle. Hence, $\text{tri}(H) = \text{tri}(G) + 1$.

□

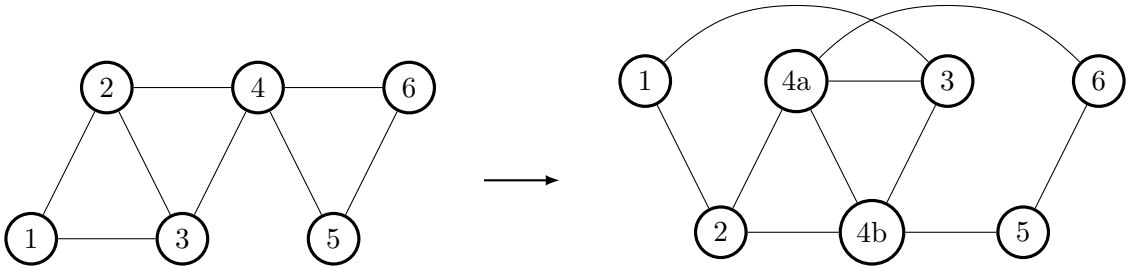
In light of the above lemma, we can define triangle types depending on which of the four cases the triangle is in. This is illustrated in Figure 4.10.

Definition 4.23 (Triangle Types)**.** Suppose a graph G contains a triangle (v_1, v_2, v_3). If there exist vertices $v_4, v_5, v_6 \notin \{v_1, v_2, v_3\}$ with (v_1, v_4, v_5) and (v_2, v_3, v_6) triangles, we call double triangle expansion of this triangle with v_1 as the special vertex a Type I operation. If there exist vertices $v_4, v_5 \notin \{v_1, v_2, v_3\}$ with (v_1, v_4, v_5) a triangle, and the edge (v_2, v_3) is part of exactly one triangle, we call double triangle expansion of this triangle with v_1 as the special vertex a Type II operation. If there exist a vertex $v_4 \notin \{v_1, v_2, v_3\}$ so that (v_2, v_3, v_4) is a triangle, and v_1, a, b is not a triangle for any $a, b \notin \{v_2, v_3\}$, we call double triangle expansion of this triangle with v_1 as the special vertex a type III operation. Finally, if (v_2, v_3, a) and (v_1, b, c) is not a triangle for any $a \neq v_1$ and $b, c \notin \{v_2, v_3\}$, we call double triangle expansion of this triangle with v_1 as the special vertex a type IV operation.

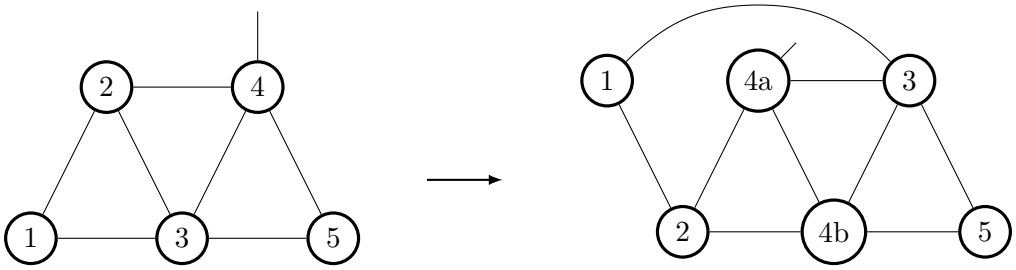
Type Ia



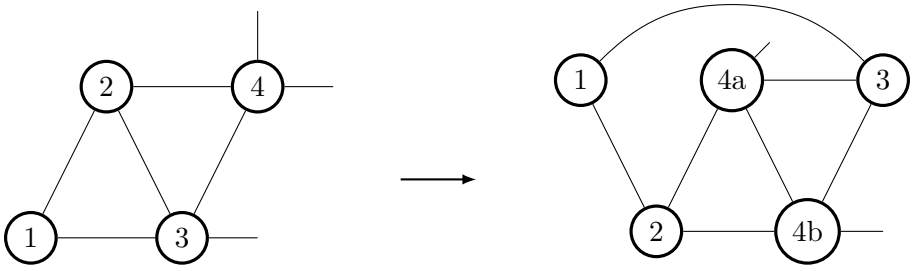
Type Ib



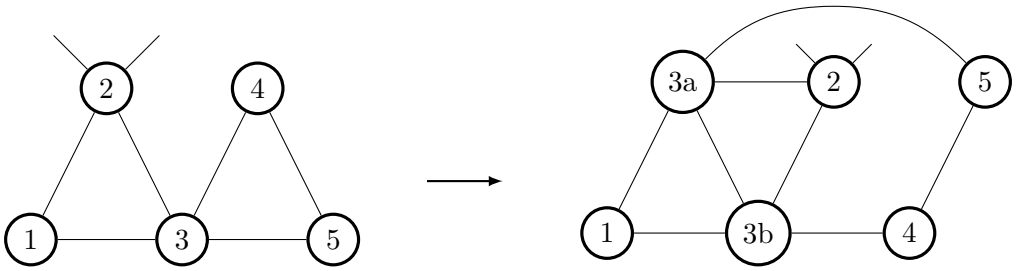
Type IIa



Type IIb



Type III



Type IV

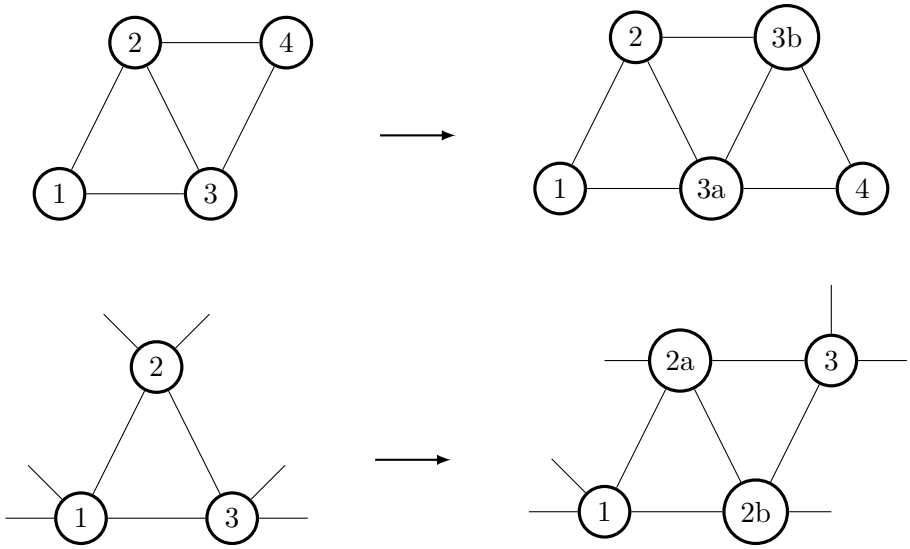


Figure 4.10: This figure shows a double triangle expansion of a triangle in various configurations. The vertices suffixed with a letter, such as $3a$ and $3b$ in the first right figure, indicate which triangle and special vertex were chosen for each vector expansion. Each vertex is of degree 4 in the graph. Edges that are not part of any triangle in the graph are drawn as half-edges here. Missing edges are edges that can be, but are not necessarily, part of any triangle. For instance, in the third figure on the left, vertex 2 is missing one edge, indicating that vertices 1 and 2 might be adjacent to a common vertex v, thus making a triangle. Vertex 4, on the other hand, is adjacent to a half-edge, which means that the fourth edge adjacent to vertex 4 cannot be part of any triangle. The effect of the double triangle expansions on the chain vector is summarized in Table 4.4 in page 49.

Lemma 4.24. *Let G be a K_5-descendant, and let H be a child of G. Then, $L(H) = L(G)+c$, where $c \in \{0, 1, 2\}$.*

Proof of Lemma 4.24. We have $\text{order}(H) = \text{order}(G)+1$. By Lemma 4.32, $\text{tri}(H) = \text{tri}(G)+c$, where $c \in \{-1, 0, 1\}$. Hence,

$$\begin{aligned}\text{L}(H) - \text{L}(G) &= (\text{order}(H) - \text{tri}(H)) - (\text{order}(G) - \text{tri}(G)) \\ &= (\text{order}(H) - \text{order}(G)) - (\text{tri}(H) - \text{tri}(G)) = 1 - c \in \{0, 1, 2\},\end{aligned}$$

as desired.

□

Proposition 4.25 (1-zigzags)**.** *If G of order $n \geq 7$ is a K_5-descendant with $L(G) = 0$, then G is a 1-zigzag graph. Furthermore, there is only one 1-zigzag at each order, up to isomorphism.*

Proof. We proceed by induction on the order of G, n. If $n = 7$, then there is indeed one K_5-descendant with $\text{L} = 0$, and it is the 1-zigzag graph Z_5. Now assume, for some $k \geq 7$, that $n = k, \text{L}(G) = 0 \implies G$ is a 1-zigzag. Suppose G is a K_5-descendant of order k. By Lemma 4.24, any child H of G has $\text{L}(H) = \text{L}(G) + a$, where $a \in \{0, 1, 2\}$. In particular, a graph H with $\text{L}(H) = 0$ can only be a child of G if $\text{L}(G) = 0$, implying that G is a 1-zigzag graph. In order for $\text{L}(H) = 0$, the operation that produces H from G needs to be a Type I double triangle expansion. Consider that a 1-zigzag graph is vertex-transitive, implying that the neighbourhood of any vertex is the same up to symmetry. There is one Type I triangle at each vertex, and so all Type I triangles are the same, up to isomorphism. Hence, there is only one graph H with $\text{L}(H) = 0$, and the Type I operation will result in a zigzag graph of the next order. By induction, there is only one 1-zigzag graph at each order, and those are precisely the graphs with $\text{L}(G) = 0$. □

Proposition 4.26. *There is no K_5-descendant G with $L(G) = 1$.*

Proof. We proceed by induction on the order of G, n. At $n = 7$, there are no K_5-descendants with $\text{L} = 1$. Suppose that for some $n = k \geq 7$, it is true that there are no K_5-descendants with $\text{L} = 1$. Let G be a K_5-descendant with $\text{order}(G) = k$. By Lemma 4.24, in order to obtain a graph H from G with $\text{L}(H) = 1$, either $\text{L}(G) = 0$ or $\text{L}(G) = 1$. Hence, we need $\text{L}(G) = 0$, and so by Proposition 4.25, G is a 1-zigzag. We need to perform a DTE on a Type II or a Type III triangle in G, but 1-zigzags, and in particular G, contain no such triangles. Thus, no graph H with $\text{L}(H) = 1$ can be produced. By induction, there is no K_5-descendant with $\text{L} = 1$. □

We note that there exists 4-regular graphs with $\text{L}(G) = 1$ that are therefore not K_5-descendants. One such example is the graph in Figure 4.11.

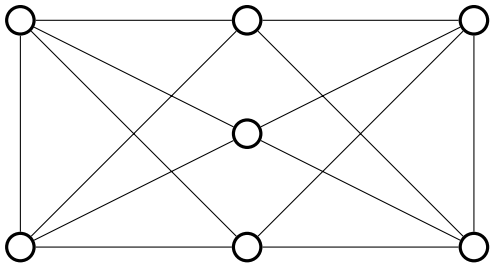


Figure 4.11: The graph shown here has 7 vertices and 6 triangles, and so L = 1. The ancestor of this graph is the multiset $\{K_5, K_5\}$.

Proposition 4.27 (2-zigzags)**.** *Let G of order $n \geq 7$ be a K_5-descendant with $L(G) = 2$. Then, G is a 2-zigzag graph. The number of such graphs, that are descendants of K_5, is given by the sequence $(0, 0, 1, 1, 2, 2, 3, 3, \dots)$, generated by the function*

$$F_2(x) = \frac{x^8}{(1-x)(1-x^2)}.$$

Proof. Recall the definition of 2-zigzags, defined in Definition 4.3 in page 24, which is a graph consisting of two zigzags sharing a vertex at each end. A standard vector, defined in Definition 4.15 in page 31, of a 2-zigzag graph is $((z_1, z_2), (a, b))$, where z_1, z_2 are the number of triangles in each zigzag, respectively, and (a, b) is the chord length vector. We want to show that the K_5-descendants with L = 2 are precisely the 2-zigzags with $z_1, z_2 \geq 3$ and $(a, b) = (2, 2)$. The result then follows, and $F_2(x)$ is given by the generating function for partitions of $i + j$ with $i, j \geq 3$, which is the same as the generating function $F_2(x)$ for partitions of $i + j - 6$ with $i, j \geq 0$. Using Lemma 2.1 in page 4, we obtain

$$F_2(x) = x^2 x^4 p_2(x) = \frac{x^8}{(1-x)(1-x^2)},$$

where we multiplied by x^2 to index by order instead of triangle count, and by x^4 since $i, j \geq 3$ instead of $i, j \geq 1$.

Observe that a Type I DTE of a 1-zigzag Z_n, with $n \geq 7$, produces a 2-zigzag graph with a standard vector $((n_1, n_2), (2, 2))$, where $n_1, n_2 \geq 3$ and $n_1 + n_2 = n$. A Type IV DTE of a 2-zigzag with a standard vector $((z_1, z_2), (2, 2))$ produces a 2-zigzag with a vector representation $((z_1 + 1, z_2), (2, 2))$ or $((z_1, z_2 + 1), (2, 2))$, with the observation that $((a, b), (2, 2))$ and $((b, a), (2, 2))$ describe the same graph. We prove the result by induction on the order.

There are no K_5-descendants of order ≤ 7 with L = 2. Now, assume that, for some $k \geq 7$, that any graph of order k and level L = 2 is a 2-zigzag with a standard vector $((z_1 \geq 3, z_2 \geq 3), (2, 2))$. Let G be a K_5-descendant with $\mathrm{L}(G) = 2$ and order $k + 1$. Consider that, by Lemma 4.24 and propositions 4.25 and 4.26, G can only be obtained by DTE of a Type I triangle of a 1-zigzag, or a Type IV triangle of an L = 2 graph. But, given the induction hypothesis, either of these operations will result in a 2-zigzag with a standard vector $((z_1 \geq 3, z_2 \geq 3), (2, 2))$. By induction, the result is true for all orders.

□

Before proceeding to prove Proposition 4.30 for L = 3, we prove some preliminary lemmas.

Lemma 4.28. *Suppose that a pseudo-descendant G has a standard vector $((z_1, z_2, z_3), C)$ where $z_1 \geq z_2 \geq z_3 \geq 3$ and $C \in \{(2, 3, 2), (3, 4, 2), (4, 2, 3)\}$. Then, G admits exactly* 1, 2, *or* 3 *chord length vectors from $\{(2, 3, 2), (3, 4, 2), (4, 2, 3)\}$, if the z_i's are distinct, exactly two of the z_i's are equal, or the z_i's are all equal, respectively. Furthermore, G, in all cases, does not admit the chord length vector $(3, 3, 3)$.*

Proof. Let $z_{11}, z_{12}, z_{21}, z_{22}, z_{31}, z_{32}$ be the chord vertices, in order along an outer path of the closed chain (z_1, z_2, z_3). The chords in G if $C = (2, 3, 2), (3, 4, 2)$, or $(4, 2, 3)$ are respectively given by $\{(z_{11}, z_{21}), (z_{12}, z_{31}), (z_{22}, z_{32})\}$, $\{(z_{11}, z_{22}), (z_{12}, z_{32}), (z_{21}, z_{31})\}$, or $\{(z_{11}, z_{31}), (z_{12}, z_{22}), (z_{21}, z_{32})\}$. Define the length of a chord (u, v) as the smaller of $\mathrm{Pos}(u) - \mathrm{Pos}(v) \mod 6$ and $\mathrm{Pos}(v) - \mathrm{Pos}(u) \mod 6$. The lengths of chords are respectively given by $(2, 3, 2), (3, 2, 2)$ and $(2, 2, 3)$. The lengths of chords adjacent to chord vertices in the first zigzag are respectively given by $(2, 3), (3, 2)$ and $(2, 2)$. The lengths of chords adjacent to chord vertices in the second zigzag are respectively given by $(2, 2)$, $(3, 2)$ and $(2, 3)$. The lengths of chords adjacent to chord vertices in the third zigzag are respectively given by $(3, 2)$, $(2, 2)$ and $(2, 3)$.

Suppose that $z_1 > z_2 > z_3$. Then, each of the chord length vectors $(2, 3, 2), (3, 4, 2)$, or $(4, 2, 3)$ represent a different graph, since the lengths of chords adjacent to each zigzag are different for each case. Suppose that $z_1 = z_2 > z_3$. Then, $(2, 3, 2)$ and $(4, 2, 3)$ represent the same graph, and $(3, 4, 2)$ represents a different graph. If $z_1 > z_2 = z_3$, then $(2, 3, 2)$ and $(3, 4, 2)$ represent the same graph, and $(4, 2, 3)$ represents a different graph. If $z_1 = z_2 = z_3$, then $(2, 3, 2), (3, 4, 2)$, and $(4, 2, 3)$ all represent the same graph.

Finally, G does not admit the chord length vector $(3, 3, 3)$, since, if it did, all of its chords will have length 3, but since G admits one of $\{(2, 3, 2), (3, 4, 2), (4, 2, 3)\}$, it has two chords of length 2 and one chord of length 3, a contradiction.

□

Lemma 4.29. *A K_5-descendant G has $L = 3$ if and only if G has one of the following standard vectors:*

- $((z_1, z_2, 0), (3, 5, 3, 3))$ *where* $z_1 \geq z_2 \geq 3$*;*
- $((z_1, z_2, z_3), (3, 3, 2))$ *where* $z_1 \geq z_2 \geq 3$ *and* $1 \leq z_3 \leq 2$*;*
- $((z_1, z_2, z_3), (3, 3, 3)$ *where* $z_1 \geq z_2 \geq z_3 \geq 3$*.*
- $((z_1, z_2, z_3), C)$ *where* $z_1 \geq z_2 \geq z_3 \geq 3$ *and* $C \in \{(2, 3, 2), (3, 4, 2), (4, 2, 3)\}$*.*

Proof. ($\Longleftarrow$) Using Proposition 4.19 in page 33, we have that $\mathrm{L}(G) = n - t = c + z + l$. In the first case, $c = 1, z = 2$ and $l = 0$, and so $\mathrm{L}(G) = 3$. In the remaining three cases, $c = 0, z = 3$ and $l = 0$, and so $\mathrm{L}(G) = 3$, as desired.

($\Longrightarrow$) We prove this by induction on the order of G. The result is true for order = 9. Suppose that the result is true for some order $n \geq 9$, and let G be a K_5-descendant of order n. By Lemma 4.24, to obtain a graph H with $\mathrm{L}(H) = 3$, we need to DTE a Type I triangle of G if $\mathrm{L}(G) = 1$, a Type II or III triangle of G if $\mathrm{L}(G) = 2$, and a Type IV triangle of G if $\mathrm{L}(G) = 3$.

There are no graphs with $\mathrm{L}(G) = 1$ by Proposition 4.26, and so we assume first that $\mathrm{L}(G) = 2$, which implies that G is a 2-zigzag with a standard vector $((z_1, z_2), (2, 2))$ with $z_1, z_2 \geq 3$, by Proposition 4.27. We can only DTE Type II triangles since G has no Type III triangles. Observe that we have a total of 4 Type II triangles in G, 2 for each zigzag. Since the chord vertices of a specific zigzag in G are vertex-transitive, we have a total of 2 possibly different Type II triangles in G. Suppose we DTE a Type II triangle in the zigzag with z_i triangles. Let z_j be the number of triangles in the other zigzag. Then, the graph H that is produced will have a chain vector $(z_i - 3, 3, z_j)$, with $z_i - 3$ possibly zero. We have three cases: $z_i - 3 \geq 3$, $1 \leq z_i - 3 \leq 2$, or $z_i - 3 = 0$. These cases result, respectively, in the following standard vectors

- $((k_1, k_2, 0), (3, 5, 3, 3))$ where $k_1 \geq k_2 \geq 3$;
- $((k_1, k_2, k_3), (3, 3, 2))$ where $k_1 \geq k_2 \geq 3$ and $1 \leq k_3 \leq 2$;
- $((k_1, k_2, k_3), (3, 3, 3)$ where $k_1 \geq k_2 \geq k_3 \geq 3$;

as desired.

Suppose now that $\mathrm{L}(G) = 3$. Then, G, by the induction hypothesis, admits one of the following standard vectors:

- $((z_1, z_2, 0), (3, 5, 3, 3))$ where $z_1 \geq z_2 \geq 3$;

- $((z_1, z_2, z_3), (3, 3, 2))$ where $z_1 \geq z_2 \geq 3$ and $1 \leq z_3 \leq 2$;
- $((z_1, z_2, z_3), (3, 3, 3)$ where $z_1 \geq z_2 \geq z_3 \geq 3$.
- $((z_1, z_2, z_3), C)$ where $z_1 \geq z_2 \geq z_3 \geq 3$ and $C \in \{(2, 3, 2), (3, 4, 2), (4, 2, 3)\}$.

To obtain a graph with L = 3, we can only do Type IV DTE's. The effect of a Type IV DTE is to increase the number of triangles in a zigzag by 1. If G has the first, the third, or the fourth standard vector, then a Type IV DTE on G produces a graph with the same form of standard vector. Suppose then that G has the second standard vector. That is, $SV(G) = ((z_1, z_2, z_3), (3, 3, 2))$ where $z_1 \geq z_2 \geq 3$ and $1 \leq z_3 \leq 2$. If $z_3 = 1$, or if we DTE one of the first two zigzags, then a standard vector of the child is one of the following (note that the chain vectors might not be ordered)

- $((z_1 + 1, z_2, z_3), (3, 3, 2))$ where $z_1 \geq z_2 \geq 3$ and $1 \leq z_3 \leq 2$;
- $((z_1, z_2 + 1, z_3), (3, 3, 2))$ where $z_1 \geq z_2 \geq 3$ and $1 \leq z_3 \leq 2$;
- $((z_1, z_2, 2), (3, 3, 2))$ where $z_1 \geq z_2 \geq z_3 \geq 3$,

as desired.

Assume then that $z_3 = 2$, and that we DTE a Type IV triangle in the third zigzag. There are two such triangles, and DTE of each triangle results in exactly one of the following vector representations

- $((z_1, z_2, 3), (3, 3, 3)$ where $z_1 \geq z_2 \geq 3$.
- $((z_1, z_2, 3), C)$ where $z_1 \geq z_2 \geq 3$ and $C \in \{(2, 3, 2), (3, 4, 2), (4, 2, 3)\}$.

as desired.

We have exhausted all possible cases, and in each case, the child graph admitted one of the standard vectors in the lemma. By induction, the lemma is true for all orders.

□

Proposition 4.30 (Level 3)**.** *The number of K_5-descendants G with $L(G) = 3$ is given by the sequence* $(0, 0, 1, 2, 4, 7, 10, 14, \dots)$*, generated by*

$$F_3(x) = \frac{x^9(1 + x^2)}{(1 - x)^3(1 + x + x^2)}.$$

Proof. By Lemma 4.29, we have that a K_5-descendant G has L = 3 if and only if G is an element of at least one of the following sets:

$S_1 := \{G : G$ admits the standard vector $((z_1, z_2, 0), (3, 5, 3, 3)), z_1 \geq z_2 \geq 3\}$,

$S_2 := \{G : G$ admits the standard vector $((z_1, z_2, z_3), (3, 3, 2)), z_1 \geq z_2 \geq 3, 1 \leq z_3 \leq 2\}$,

$S_3 := \{G : G$ admits the standard vector $((z_1, z_2, z_3), (3, 3, 3), z_1 \geq z_2 \geq z_3 \geq 3\}$,

$S_4 := \{G : G$ admits the standard vector $((z_1, z_2, z_3), C), z_1 > z_2 > z_3 \geq 3,$
$C \in \{(2, 3, 2), (3, 4, 2), (4, 2, 3)\}\}$,

$S_5 := \{G : G$ admits the standard vector $((z, z, z_3), C), z > z_3 \geq 3,$
$C \in \{(2, 3, 2), (3, 4, 2), (4, 2, 3)\}\}$,

$S_6 := \{G : G$ admits the standard vector $((z_1, z, z), C), z_1 > z \geq 3,$
$C \in \{(2, 3, 2), (3, 4, 2), (4, 2, 3)\}\}$,

$S_7 := \{G : G$ admits the standard vector $((z, z, z), C), z \geq 3,$
$C \in \{(2, 3, 2), (3, 4, 2), (4, 2, 3)\}\}$.

Consider that, since the chain vector, the first part of a standard vector, is uniquely determined by the graph, and using Lemma 4.28, it follows that the S_i's are pairwise disjoint. Fix a positive integer t, denoting the number of triangles. Define the following counts:

$N_1(t) :=$ number of 3-part partitions of t where two parts are $P_1 \geq P_2 \geq 3$,
and one part is $0 \leq P_3 \leq 2$,

$N_2(t) :=$ number of 3-part partitions of t where the parts are $P_1 > P_2 > P_3 \geq 3$,

$N_3(t) :=$ number of 3-part partitions of t where the parts are $P_1 = P_2 > P_3 \geq 3$,

$N_4(t) :=$ number of 3-part partitions of t where the parts are $P_1 > P_2 = P_3 \geq 3$,

$N_5(t) :=$ number of 3-part partitions of t where the parts are $P_1 = P_2 = P_3 \geq 3$.

Let S_i^t denote the graphs in S_i with exactly t triangles. By Lemma 4.28, we have that

$$\sum_{i=4}^{7} |S_i^t| = 3N_2(t) + 2N_3(t) + 2N_4(t) + N_5(t).$$

We also have that

$$\sum_{i=1}^{3} |S_i^t| = N_1(t) + N_2(t) + N_3(t) + N_4(t) + N_5(t).$$

Thus,

$$\sum_{i=1}^{7} |S_i^t| = N_1(t) + 4N_2(t) + 3N_3(t) + 3N_4(t) + 2N_5(t).$$

$F_3(x)$ is then given by the generating function

$$\begin{aligned} x^{-3}F_3(x) &= \sum_{t\geq 0}\sum_{i=1}^{7} |S_i^t| x^t = \sum_{t\geq 0} (N_1(t) + 4N_2(t) + 3N_3(t) + 3N_4(t) + 2N_5(t))x^t \\ &= f_1(x) + 4f_2(x) + 3f_3(x) + 3f_4(x) + 2f_5(x), \end{aligned}$$

where $f_i(x) := \sum_{t\geq 0} N_i(t)x^t$. We multiplied $F_3(x)$ by x^{-3} to index using order instead of triangle count. Using Lemma 2.1 in page 4, we have that

$$\begin{aligned} f_1(x) = (1 + x + x^2)x^4 p_2(x) &= \frac{x^6(1 + x + x^2)}{(1 - x)(1 - x^2)}, \\ (f_2 + f_3 + f_4 + f_5)(x) = x^6 p_3(x) &= \frac{x^9}{(1 - x)(1 - x^2)(1 - x^3)}, \\ (f_2 + f_3 + f_4)(x) = x^6 p_3(x) - \frac{x^9}{1 - x^3} &= \frac{x^{10} + x^{11} - x^{12}}{(1 - x)(1 - x^2)(1 - x^3)}. \end{aligned}$$

Using Lemma 2.2 in page 4, we have that

$$f_2(x) = x^6 q_3(x) = x^6 \frac{x^{3+\binom{3}{2}}}{(1 - x)(1 - x^2)(1 - x^3)} = \frac{x^{12}}{(1 - x)(1 - x^2)(1 - x^3)}.$$

Thus,

$$\frac{F_3(x)}{x^3} = f_1(x) + f_2(x) + 2(f_2 + f_3 + f_4 + f_5)(x) + (f_2 + f_3 + f_4)(x) = \frac{x^6(1 + x^2)}{(1 - x)^3(1 + x + x^2)},$$

as desired.

□

We end this section with a conjecture on the closed form of the sequence for $\mathrm{L}(G) = 4$ in Table 4.3. The sequence continues as

$$0, 1, 6, 15, 34, 61, 106, 162, 246, 342, 477, 626, 825, 1039, 1314, 1606, 1970, 2352, 2817, 3302, \ldots$$

The candidate closed form of the generating function was found by noting that, starting at order 18, the fourth difference of the terms is the sequence $(10, -9, 9, -10, 10, -9, 9, -10, \ldots)$.

Conjecture 4.31. The number of K_5-descendants G with $\mathrm{L}(G) = 4$ is given by the sequence $(0, 1, 6, 15, 34, 61, 106, 162, 246, \cdots)$, given by

$$F_4(x) = \frac{x^{20} - x^{19} + 3x^{18} - 3x^{17} + 4x^{16} + 4x^{14} + 3x^{13} + 6x^{12} + 3x^{11} + 4x^{10} + x^9}{(1-x)^4(1+x)^2(1+x^2)}.$$

4.6 Minimum Number of Triangles is 4

Double triangle expansion always produces a double triangle, and thus, trivially, any K_5-descendant has at least two triangles. Interestingly, it can be shown that K_5-descendants have at least 4 triangles. To prove this, we will look at the effect of a double triangle expansion (DTE) of a psuedo-descendant on its chain vector. Note that consecutive 0's in a chain vector are considered the same as one 0. For example, $(3, 0, 1, 0, 0, 2, 0) = (3, 0, 1, 0, 2, 0)$. Note also that the chain vectors referred to here are not assumed to be in standard ordering.

Lemma 4.32. *Let G be a pseudo-descendent with a chain vector $(z_1, z_2, \cdots, z_k)$, where $k \geq 1$, $z_1 \geq 1$, and $z_i \geq 0$ for $2 \leq i \leq k$. Let $\tilde{G}$ be G after one double triangle expansion. Then, $\tilde{G}$ has one of the chain vectors in the fourth column of Table 4.4.*

Since double triangle reduction is the inverse of double triangle expansion, we can conclude the following corollary.

Corollary 4.33. *Let G be a pseudo-descendant with a chain vector $(z_1, z_2, \cdots, z_k)$, where $k \geq 1$, $z_1 \geq 1$, and $z_i \geq 0$ for $2 \leq i \leq k$. Let $\tilde{G}$ be G after one double triangle reduction. Then, $\tilde{G}$ has one of the chain vectors in the third column of Table 4.4.*

Proof of Lemma 4.32. We will begin with Type I DTE's. Figure 4.10 shows the two ways a Type I DTE changes a zigzag(s). When expanding a Type Ia triangle, one zigzag of size $m + n + 4$ for some $m, n \geq 0$ is operated on, where m is the number of triangles in the zigzag to the left of the chosen operating four triangle segment, and n is the number of such triangles to the right. This zigzag's contribution to the chain vector is $(\cdots, m+n+4, \cdots)$. The resulting vector representation depends on which four triangle segment of the zigzag is operated on, and how large the zigzag is. WLOG, we have three possible cases. If $m, n = 0$, the resulting vector representation will be $(\cdots, m, 3, n, \cdots)$, since the resulting zigzag is isolated from both left and right. If $m > 0$ and $n = 0$, the resulting vector is $(\cdots, m, 3, n, \cdots)$, since the resulting zigzag shares a vertex with the zigzag on the left, and is isolated from the right. Similarly, if $m, n > 0$, the resulting chain vector is $(\cdots, m, 3, n, \cdots)$. Thus, in all cases, $(\cdots, m + n + 4, \cdots)$ is changed to $(\cdots, m, 3, n, \cdots)$ for all $m, n \geq 0$.

When expanding a Type Ib triangle, two zigzags of sizes $m+2$ and $n+1$, respectively, are operated on, where m indicates the number of triangles to the left of the leftmost zigzag, and n the number of triangles to the right of the rightmost zigzag. The chain vector contribution of the two zigzags is $(\cdots, m+2, n+1, \cdots)$. We have three cases. If $m, n = 0$, that part of the vector becomes $(\cdots, 0, 2, 0, 0, \cdots) = (\cdots, m, 2, 0, n, \cdots)$. If $m > 0, n = 0$, we get $(\cdots, m, 2, 0, 0, \cdots) = (\cdots, m, 2, 0, n, \cdots)$ and if $m, n > 0$, we get $(\cdots, m, 2, 0, n, \cdots)$. In all cases, $(\cdots, m+2, n+1, \cdots)$ is changed to $(\cdots, m, 2, 0, n, \cdots)$ for all $m, n \geq 0$.

We now move on to Type II DTE's. Referring to Figure 4.10, there are two cases: either the special vertex (vertex 4 in the corresponding figure) is part of exactly two triangles, or it is part of exactly one triangle. Suppose first that it is part of exactly two triangles. The part of the chain vector that is being operated on is $(\cdots, m+3, \cdots)$, where $m \geq 0$ is the number of triangles to the left of the three triangle segment. If $m = 0$, then the vector part after DTE becomes $(\cdots, 0, 3, \cdots) = (\cdots, m, 3, \cdots)$. If $m > 0$, then the vector part after DTE becomes $(\cdots, m, 3, \cdots)$. In either case, $(\cdots, m+3, \cdots)$ is changed to $(\cdots, m, 3, \cdots)$.

Suppose now that the special vertex is part of exactly one triangle. The part of the chain vector that is being operated on is $(\cdots, m+2, 0, \cdots)$, where $m \geq 0$ is the number of triangles to the left of the two triangle segment (See Type IIb in Figure 4.10). If $m = 0$, the vector segment becomes $(\cdots, 0, 2, 0, \cdots) = (\cdots, m, 2, 0, \cdots)$, and if $m > 0$, it becomes $(\cdots, m, 2, 0, \cdots)$. In either case, $(\cdots, m+2, 0, \cdots)$ is changed to $(\cdots, m, 2, 0, \cdots)$.

Next, we move on to the Type III DTE. The triangle being expanded will be as shown in Figure 4.10 (Type III). The zigzag containing the triangle being operated on ($(1, 2, 3)$ in Figure 4.10) must contain exactly 1 triangle, as, otherwise, the zigzag containing $(1, 2, 3)$ will be longer and it becomes a Type Ib operation. Let m be the number of triangles to the right of the rightmost triangle (and part of the same zigzag). A DTE of $(1, 2, 3)$ with 3 as the special vertex will result in a 2-triangle zigzag Z, and, if $m > 0$, an m-triangle zigzag that does not share any vertex with Z, or, if $m = 0$, it will add 2 lone vertices (4 and 5). If vertex 1 was part of two zigzags, it will also be part of two zigzags after expansion. In either case, the chain vector $(\cdots, 1, m+1, \cdots)$ will be changed to $(\cdots, 2, 0, m, \cdots)$.

Finally, we move on to Type IV DTE. There are two cases. Either the triangle is part of a larger zigzag, or it's part of a 1-triangle zigzag. In the first case, we operate on $(\cdots, m+1, \cdots)$, where $m \geq 0$ is the number of triangles to the left or right of the triangle being operated on. In either case, a Type IV DTE's only effect is to add one triangle to the zigzag containing the triangle being expanded. Thus, the chain vector $(\cdots, m+1, \cdots)$ is changed to $\cdots, m+2, \cdots$.

□

Table 4.4: This table summarizes the different ways double triangle expansion can change the chain vector of a pseudo-descendant. Refer to Figure 4.10 in page 39.

Triangle Type	**Parameters**	**Vector before expansion**	**Vector after expansion**
Ia	$m, n \geq 0$	$(\cdots, m+n+4, \cdots)$	$(\cdots, m, 3, n, \cdots)$
Ib	$m, n \geq 0$	$(\cdots, m+2, n+1, \cdots)$	$(\cdots, m, 2, 0, n, \cdots)$
IIa	$m \geq 0$	$(\cdots, m+3, \cdots)$	$(\cdots, m, 3, \cdots)$
IIb	$m \geq 0$	$(\cdots, m+2, 0, \cdots)$	$(\cdots, m, 2, 0, \cdots)$
III	$m \geq 0$	$(\cdots, 1, m+1, \cdots)$	$(\cdots, 2, 0, m, \cdots)$
IV	$m \geq 0$	$(\cdots, m+1, \cdots)$	$(\cdots, m+2, \cdots)$
IV	None	$(\cdots, 0, 1, 0, \cdots)$	$(\cdots, 0, 2, 0, \cdots)$

We will now proceed to prove the main result of the section after the following lemma.

Lemma 4.34. *All K_5-descendants of order ≥ 8 that are not 1-zigzags can be double triangle reduced to the K_5-descendant with standard vector $((3,3),(2,2))$.*

Proof. This result immediately follows from Lemma 4.24 and propositions 4.25, 4.26 and 4.27. □

Theorem 4.35. *If G is a K_5-descendant, then tri$(G) \geq 4$.*

Proof. Let G be a K_5-descendant with $\text{tri}(G) \leq 3$. We know that $\text{tri}(G) \geq 4$ if $\text{order}(G) < 8$, so we assume $\text{order}(G) \geq 8$. By Proposition 4.25, we have that if G is a 1-zigzag, then $\text{tri}(G) = \text{order}(G) \geq 8$. We assume then that G is not a 1-zigzag. By Theorem 3.10 and Lemma 4.34, we should be able to reach the chain vector $(3,3)$ from G through double triangle reductions exclusively (that is, without double triangle expansions). Since double triangle expansion always leaves a double triangle, at least one of the zigzags in G has ≥ 2 triangles. Thus, the chain vector of G is one of the following:

$$(3), (2,1), (2), (3,0), (2,1,0), (2,0), (2,0,1,0).$$

The chain vectors (3), $(2,1)$ and (2) are not possible since G will have 5, 6 or 4 vertices, respectively, but $\text{order}(G) \geq 8$. Therefore, there remains four cases to consider. We will refer to Table 4.4 in page 49, and we want to show that for each case, successive double triangle reductions either leads to a chain vector in another case, or to an unrealizable chain vector.

Case 1: $CV_1 = (3, 0)$.

First, observe that the only way to have a DTR strictly increase the number of non-zero entries in the chain vector is through Type Ib with $m = n = 0$, or Type III with $m = 0$. In either case, the chain vector needs to contain a 2.

Using a Type Ia reduction with $m = n = 0$ gives us $CV_2 = (4, 0)$, which will force us to use Type IV reduction to get to $(3, 0)$ again since we need a 2. Using a Type IIa reduction on $(3, 0)$ results in (3), which has already been excluded, or $(3, 0) = CV_1$. The only possible useful step at this point, then, is to use Type IV reduction to obtain $CV_2 = (2, 0)$, which is considered in Case 3.

Case 2: $CV_1 = (2, 1, 0) = (1, 2, 0)$.

Using a Type Ib reduction with $m = 1$ and $n = 0$ results in $CV_2 = (3, 1)$ or $(3, 1, 0)$. Suppose that $CV_2 = (3, 1)$. The graph with chain vector $(3, 1)$, if it exists, has order 6 and 4 triangles, but the only K_5-descendant of order 6 (the zigzag graph Z_4) has 8 triangles, and so this subcase is not possible. So we suppose that $CV_2 = (3, 1, 0)$. A Type Ia reduction with $m = 0$ and $n = 1$ results in $CV_3 = (4)$ or $(4, 0)$. If $CV_3 = (4)$, then the graph, if it exists, will have 6 vertices. The only K_5-descendant with 6 vertices, Z_4, has exactly one child, Z_5, for which the chain vector $(3, 1, 0)$ is not a valid representation. Thus, we assume $CV_3 = (4, 0)$. As was argued in Case 1, this forces us to use Type IV reduction twice to obtain $CV_5 = (2, 0)$.

We start over at $CV_1 = (2, 1, 0)$. The only other possible operation is a Type IIb reduction with $m = 1$, which results in $CV_2 = (3, 0)$, forcing us to do a Type IV reduction, resulting in $CV_3 = (2, 0)$.

Case 3: $CV_1 = (2, 0)$.

We have three options for the next step.

- Subcase 1: We use a Type Ib reduction with $m = n = 0$, giving us $CV_2 = (2, 1)$ or $CV_2 = (2, 1, 0)$. The chain vector $(2, 1)$ is not realizable, while $(2, 1, 0)$, dealt with in Case 2, can only lead us back to $(2, 0)$.
- Subcase 2: We use a Type IIb reduction with $m = 0$, which results in $CV_2 = (2, 0)$.
- Subcase 3: We use a Type III reduction with $m = 0$, which results in $CV_2 = (1, 1)$ or $CV_2 = (0, 1, 1)$. In either case, the chain vector is not possible, since at least one zigzag must contain more than one triangle for any K_5-descendants.

In all subcases, we either get back to $(2,0)$ or reach an unrealizable (for a K_5-descendant) chain vector.

Case 4: $CV_1 = (2,0,1,0)$.

We have three options for the next step.

- Subcase 1: We use a Type Ib reduction with $m = 0$ and $n = 1$, giving us $CV_2 = (2,2)$ or $CV_2 = (2,2,0)$. If a graph has chain vector $(2,2)$, then it has 6 vertices, but the only K_5-descendant of 6 vertices is Z_4, for which $(2,2)$ is not a valid chain vector. We suppose then that $CV_2 = (2,2,0)$. If we do a Type Ib reduction with $m = 2$ and $n = 0$, we obtain $CV_3 = (4,1)$ or $(4,1,0)$. If a graph has chain vector $(4,1)$, then it has 7 vertices, but the only K_5-descendant of 7 vertices is Z_5, which has 7 triangles, not $5 = 4+1$. Thus we assume $CV_3 = (4,1,0)$. We are forced to use a Type IV reduction to obtain $CV_4 = (3,1,0)$, which was already dealt with in Case 2 and will eventually result in $CV = (2,0)$.

  We return to $CV_2 = (2,2,0)$. We can do a Type IIb reduction with $m = 2$ to obtain $CV_3 = (4,0)$, which ultimately lead to $CV = (2,0)$. The only other possible operation on CV_2 is a Type IV operation to obtain $CV_3 = (2,1,0)$, which is dealt with in Case 2, and will eventually lead to $CV = (2,0)$.

- Subcase 2: We use a Type IIb reduction with $m = 0$, which results in $CV_2 = (2,0,1) = (2,1,0)$, which is dealt with in Case 2.

- Subcase 3: We use a Type III reduction with $m = 1$, which results in $CV_2 = (0,1,2)$, which is dealt with in Case 2, or $CV_2 = (1,2)$, which has been already excluded.

In all cases, we either get back to $(2,0)$, or reach an unrealizable chain vector.

□

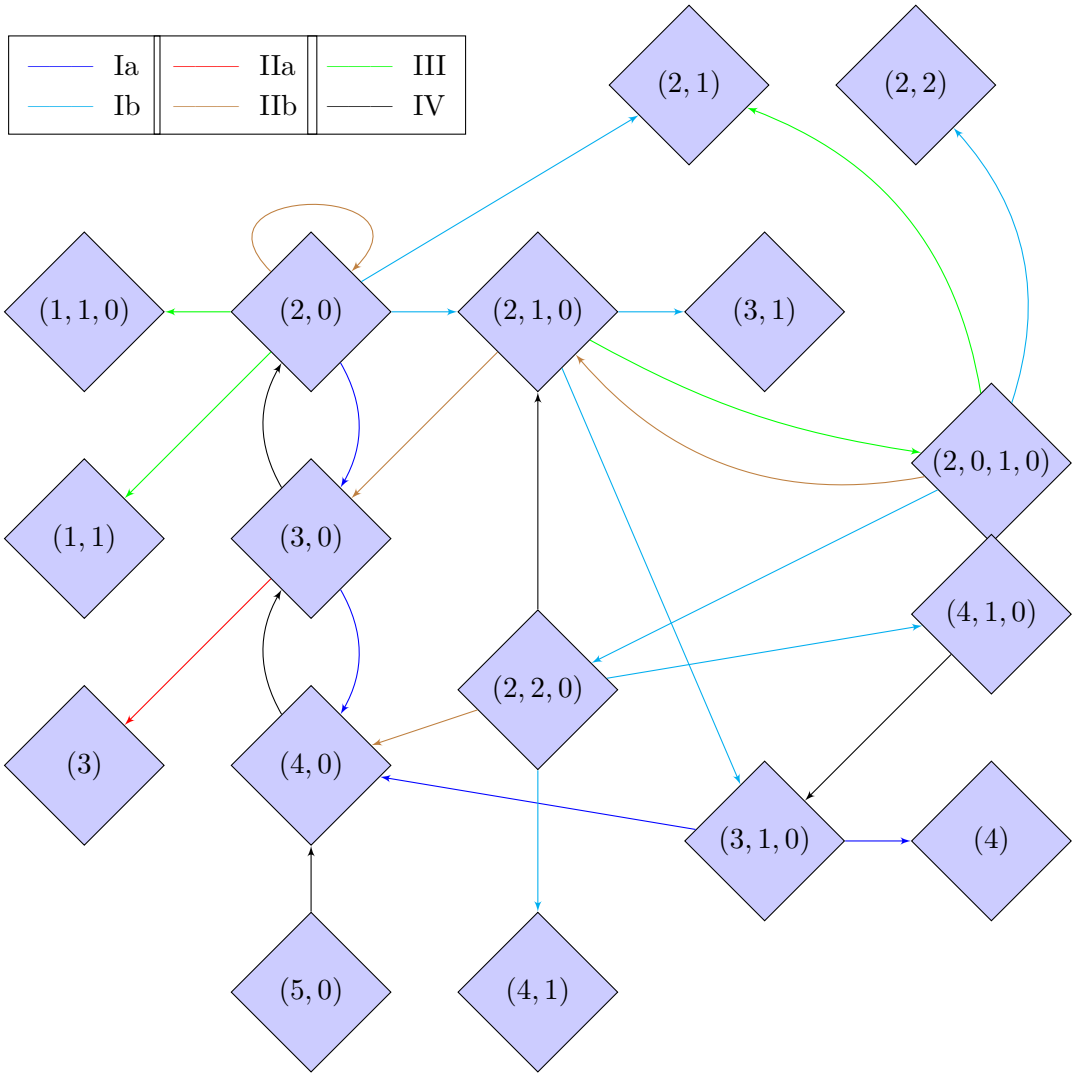


Figure 4.12: This figure illustrates the proof of Theorem 4.35. It shows the various ways DTR can affect the chain vector when starting with < 4 triangles. The key observation is that if you start with a chain vector that has at most 3 triangles, then no sequence of DTR's can result in the chain vector that is admissible for some K_5-descendant.

Chapter 5

Conclusion

The aim of this document was to better understand the structure of K_5-descendants, which are a combinatorially interesting class of graphs, and are interesting in ϕ^4-theory given Brown and Schnetz's conjecture that these graphs are the only graphs in ϕ^4-theory with constant -1 c_2-invariant,

Conjecture 3.14 (Brown and Schnetz. [5, Conjecture 25, p16])**.** Let G be a completed primitive graph and let $\tilde{G}$ be any decompletion of G. Then,

$$c_2(\tilde{G}) = (-1, -1, \cdots) \iff G \cong K_5 \text{ or } G \text{ is a double-triangle descendant of } K_5.$$

A better understanding of the structure of K_5-descendants is our best way to step towards proving this conjecture. Additionally, the double triangle operations are interesting graph operations, both graph theoretically and in ϕ^4-theory, given that they preserve the c_2-invariant, completed primitiveness, and several other graph properties. The main contributions of this thesis are the vector encoding for K_5-descendants presented in Section 4.3, the counting propositions in Section 4.5, and the result that the minimum number of triangles in a K_5-descendant is 4.

The vector encoding is a step towards characterization of these graphs. It is also useful for quickly checking non-isomorphism of graphs that have different chain vectors. The counting propositions are interesting enumerative combinatorics and the technique used in the proofs could potentially be generalized. The minimum triangle count result is interesting because, firstly, it is not obvious that minimum should not just be two, instead of 4 as the result shows. Secondly, it is not as simple as having 2 separate double triangles, but in fact a K_5-descendant with (exactly) 4 triangles could have one 2-triangle zigzag and two 1-triangle zigzags.

Computer programs were heavily used in discovering some of the results. The software that was used for coding is the SageMath software package, an open source programming platform based on Python. SageMath was chosen due to the large number of graph theory libraries available, and the ease at which programs can be coded and debugged. Some of the code used is included in the appendix. This code can be used to enumerate the descendants of K_5 or any other simple 4-regular graph, calculate the double triangle ancestor of a 4-regular graph, calculate a standard vector representation for a pseudo-descendant, and plot pseudo-descendants in an easy to read manner. Given a chain vector, the code can be used to list and separately enumerate K_5-descendants and non-K_5-descendants with that chain vector.

There is potentially more that can be done with the standard vector representation. The effect of double triangle expansion on the chain vector was calculated in Section 4.6, but calculating the effect of DTE's on the chord length vector could prove insightful on the structure of K_5-descendants. Additionally, in the standard vector, there are no restrictions on lone vertices, but it seems likely that only certain lone vertex structures are possible. If so, this could both improve the usefulness of the standard vector representation, and help in full characterization.

Solving the author's conjecture for the closed form of the generating function that counts K_5-descendants with level $\mathrm{L} = 4$ could potentially give some clues as to how to approach the enumeration of K_5-descendants of the next levels, $\mathrm{L} \geq 5$.

Conjecture 4.31. The number of K_5-descendants G with $\mathrm{L}(G) = 4$ is given by the sequence $(0, 1, 6, 15, 34, 61, 106, 162, 246, \cdots)$, given by

$$F_4(x) = \frac{x^{20} - x^{19} + 3x^{18} - 3x^{17} + 4x^{16} + 4x^{14} + 3x^{13} + 6x^{12} + 3x^{11} + 4x^{10} + x^9}{(1-x)^4(1+x)^2(1+x^2)}.$$

The pseudo-descendant quantities introduced in Section 4.4 could be useful to quickly weed out graphs that do not have the right relation between those quantities. The effect of double triangle expansion on the n, c, z, t, l parameters can be determined, resulting in vectors representing how DTE can change these values. The positive integer span of these vectors spans all K_5-descendant graphs, and expectedly many non-K_5-descendants. However, graphs that are not in the span can be ruled out.

The question of the complexity of this problem is also interesting. For instance, could double triangle expansion on a pseudo-descendant create local non-planarities? If not, this suggests that the class of K_5-descendants is somewhat more tangible than thought, giving hope for a full characterization. If the answer is yes, what does this say about the difficulty of the problem? The author thanks supervisor Matthew DeVos for suggesting this interesting problem.

Brown and Schnetz's conjecture that only K_5-descendants have $c_2 = -1$ remains unsolved, and we do not seem to be closer to the answer in either direction.

Bibliography

[1] James Ax. Zeroes of polynomials over finite fields. *Amer. J. Math.*, 86:255–261, 1964.

[2] Prakash Belkale and Patrick Brosnan. Matroids, motives, and a conjecture of Kontsevich. *Duke Math. J.*, 116(1):147–188, 2003.

[3] F. C. S. Brown. On the periods of some Feynman integrals. *ArXiv e-prints*, October 2009, 0910.0114.

[4] Francis Brown and Oliver Schnetz. A K3 in ϕ^4. *Duke Math. J.*, 161(10):1817–1862, 2012.

[5] Francis Brown and Oliver Schnetz. Modular forms in quantum field theory. *Commun. Number Theory Phys.*, 7(2):293–325, 2013.

[6] Francis Brown and Oliver Schnetz. Single-valued multiple polylogarithms and a proof of the zig-zag conjecture. *J. Number Theory*, 148:478–506, 2015.

[7] Francis Brown and Karen Yeats. Spanning forest polynomials and the transcendental weight of Feynman graphs. *Comm. Math. Phys.*, 301(2):357–382, 2011.

[8] D. Bump. *Algebraic Geometry*. World Scientific, 1998.

[9] Pete L. Clark. The Chevalley-Warning Theorem (featuring. . . the Erdos-Ginzburg-Ziv Theorem). from *http://math.uga.edu/ pete/4400ChevalleyWarning.pdf*. Retrieved August 20 2017.

[10] Reinhard Diestel. *Graph theory*, volume 173 of *Graduate Texts in Mathematics*. Springer, Berlin, fifth edition, 2017.

[11] Martin J. Erickson. *Introduction to combinatorics*. Wiley Series in Discrete Mathematics and Optimization. John Wiley & Sons, Inc., Hoboken, NJ, second edition, 2013.

[12] B. Moore and K. Yeats. Graph Minors and the Linear Reducibility of Feynman Diagrams. *ArXiv e-prints*, August 2017, 1708.01691.

[13] Lewis H. Ryder. *Quantum field theory*. Cambridge University Press, Cambridge, second edition, 1996.

[14] Oliver Schnetz. Quantum periods: a census of ϕ^4-transcendentals. *Commun. Number Theory Phys.*, 4(1):1–47, 2010.

[15] John R. Stembridge. Counting points on varieties over finite fields related to a conjecture of Kontsevich. *Ann. Comb.*, 2(4):365–385, 1998.

[16] Karen Yeats. *Growth estimates for Dyson-Schwinger equations*. ProQuest LLC, Ann Arbor, MI, 2008. Thesis (Ph.D.)–Boston University.

[17] Karen Yeats. A few c_2 invariants of circulant graphs. *Commun. Number Theory Phys.*, 10(1):63–86, 2016.

[18] Karen Yeats. *A combinatorial perspective on quantum field theory*, volume 15 of *SpringerBriefs in Mathematical Physics*. Springer, Cham, 2017.

Appendix A

Code

Computer programs were heavily used in discovering some of the results. The software that was used for coding is the SageMath software package, an open source programming platform based on Python. SageMath was chosen due to the large number of graph theory libraries available, and the ease at which programs can be coded and debugged. Some of the code used is included in the appendix. This code can be used to enumerate the descendants of K_5 or any other simple 4-regular graph, calculate the double triangle ancestor of a 4-regular graph, calculate a standard vector representation for a pseudo-descendant, and plot pseudo-descendants in an easy to read manner. Given a chain vector, the code can be used to list and separately enumerate K_5-descendants and non-K_5-descendants with that chain vector.

The code can be found at https://github.com/mlaradji/k5-descendants.

Listings

init.py

```
import itertools
import time
from collections import deque
import sympy as sp
import pickle

execfile('generalfunctions.py')

execfile('DTRchainfunctions.py')
execfile('DTRfunctions.py')
execfile('PDgraphrepresentation.py')
execfile('PDvectorrepresentation.py')

execfile('K5dsearch.py')
execfile('skeletonsearch.py')
```

initializevectorsearch.py

```
##############################################################################
# Executing this file produces the objects required to execute
# K5dsearch(start,target,Gseq,SVseq,parents,expanded). Use start=1.
#
# Note that this erases any previously stored output of K5dsearch from
#  memory.
#

version=0
Gseq=[[] for i in range(0,5)]
SVseq=dict()
expanded=dict()
lst1=[[] for i in range(0,11)]
Gseq.append(lst1)
Gseq[5][10].append([[graphs.CompleteGraph(5)]])
SVseq[tuple([5,10,0,0,0])]=tuple([tuple([tuple([10]),tuple([])])])
expanded[tuple([5,10,0,0,0])]=0
parents=dict()
```

k5dsearch_save.py

```
version+=1

with open("K5dsearch_version.file", "wb") as f:
    pickle.dump(version, f, pickle.HIGHEST_PROTOCOL)

with open("K5dsearch_Gseq_"+str(version)+".file", "wb") as f:
    pickle.dump(Gseq, f, pickle.HIGHEST_PROTOCOL)

with open("K5dsearch_SVseq_"+str(version)+".file", "wb") as f:
    pickle.dump(SVseq, f, pickle.HIGHEST_PROTOCOL)

with open("K5dsearch_expanded_"+str(version)+".file", "wb") as f:
    pickle.dump(expanded, f, pickle.HIGHEST_PROTOCOL)

with open("K5dsearch_parents_"+str(version)+".file", "wb") as f:
    pickle.dump(parents, f, pickle.HIGHEST_PROTOCOL)
```

k5dsearch_load.py

```
with open("K5dsearch_version.file", "rb") as f:
    version = pickle.load(f)

with open("K5dsearch_Gseq_"+str(version)+".file", "rb") as f:
    Gseq = pickle.load(f)

with open("K5dsearch_SVseq_"+str(version)+".file", "rb") as f:
    SVseq = pickle.load(f)

with open("K5dsearch_expanded_"+str(version)+".file", "rb") as f:
    expanded = pickle.load(f)

with open("K5dsearch_parents_"+str(version)+".file", "rb") as f:
    parents = pickle.load(f)
```

K5dsearch.py

```
#############################################################################
# Input: start - The order at which to start the search. Note that
#                Gseq must contain at least one graph of order start-1.
#                If initializevectorsearch.py is executed, use start=1.
#        target - The order at which to stop.
#        Gseq - List of K5 descendants. Gseq[order][tri] is a list
#               containing graphs with order vertices and tri triangles.
#        SVseq - dictionary with keys (order, triangle count, index).
#                SVseq[(order, tri, index)] gives the standard
#                vector of Gseq[order][tri][index].
#        parents - dictionary with keys (order, triangle count, index).
#                  parents[(order, tri, index)] gives the standard
#                  representation of the parents of
#                  Gseq[order][tri][index].
#        expanded - dictionary with keys (order, triangle count, index).
#                   expanded[(order, tri, index)]=1 if all the
#                   descendants of Gseq[order][tri][index] have been
#                   found, 0 otherwise.
#
# Output: Gseq, SVseq, parents, expanded
#
# Note 1: Gseq and SVseq are only saved in memory, and not to disk. To
#  save to disk, use the command execfile('k5dsearch_save.py'). To load
#  from disk, use the command execfile('k5dsearch_load.py').
#
# Note 2: The parents dictionary is currently disabled due to bugs.
#

def K5dsearch(start,target,Gseq,SVseq,parents,expanded):

    start_time = time.time()
    for order in range(start,target+1):

        while len(Gseq)<order+1:
            Gseq.append([[]])

        int_time, nc_time=time.time(),time.time()

        for tricount in range(0,len(Gseq[order-1])):
            for chaincount in range(0,len(Gseq[order-1][tricount])):
                for lonecount in range(0,len(
                    Gseq[order-1][tricount][chaincount])):
                    for index in range(0,len(
                        Gseq[order-1][tricount][chaincount][lonecount])):
                        G=copy(Gseq[order-1][tricount][
```

```
    chaincount][lonecount][index])
triset=set()
if not expanded[tuple([order-1,tricount,
                       chaincount,
                       lonecount,index])]:
    trilist=listtriangles(G)
    for i in range(0,3):
        triset=triset.union(trilist[i])

for tri in triset:
    candidate=copy(G)
    dtri(candidate,tri[1],tri[2],tri[0],0)
    candidate=pdrelabel(candidate)
    candSV=tuple(standardvector(candidate))
    candPDparam=PDparameters(candSV[0])
    candtricount=candPDparam[0]
    candchaincount=candPDparam[1]
    candlonecount=candPDparam[3]

    while len(Gseq[order])<candtricount+1:
        Gseq[order].append([])
    while len(Gseq[order][
        candtricount])<candchaincount+1:
        Gseq[order][candtricount].append([])
    while len(Gseq[order][candtricount][
        candchaincount])<candlonecount+1:
        Gseq[order][candtricount][
            candchaincount].append([])

    iso=0

    for i in range(0,len(Gseq[order][
        candtricount][candchaincount][
        candlonecount])):
        if candSV[0]==SVseq[tuple([
            order,candtricount,
            candchaincount,candlonecount,i])][0]:

        #parents[
        #    tuple([order,candtricount,i])].add(
        #    tuple([order-1,tricount,index]))
            if candidate.is_isomorphic(Gseq[
                order][candtricount][
                candchaincount][candlonecount][i]):
                iso=1
                break
```

```
                    if not iso:
                        Gseq[order][candtricount][
                            candchaincount][
                            candlonecount].append(candidate)
                        tpl=tuple([order,candtricount,
                                   candchaincount,candlonecount,len(
                                       Gseq[order][candtricount][
                                           candchaincount][
                                           candlonecount])-1])
                        SVseq[tpl]=candSV
                        #parents[tpl]=set([tuple([
                        #     order-1,tricount,index])])
                        expanded[tpl]=0

                    expanded[tuple([order-1,tricount,
                                    chaincount,lonecount,index])]=1

        print(str(order)+'-'+str(time.time() - int_time))
    print("--- %s seconds ---" % (time.time() - start_time))
```

##

skeletonsearch.py

```
#############################################################################
# Input: L is a list of nonnegative integers.
# Output: descendants - list of descendants with L as their chain vector.
#         illegals - list of pseudo-descendants with chain vector L.
# Example input: L=[3,3,3], L=[3,3,0] or L=[3,0,3,0,-1].
#
# Note that if there are several lone vertices, then each one is to be
#   represented as -1 in L.

def listgraphs3(L):
    S,V1,V2,V4=skeleton(L)
    Lz=listzigzags(S)
    noftri=0
    for i in iter(L):
        if i>0:
            noftri+=i
    descendants=[]
    illegals=[]
    K5=graphs.CompleteGraph(5)
    possibleneighborsr=dict()
    remainingdegree=dict()
    Vs=set(union(set(V1),union(set(V2),set(V4))))
    for i in iter(V1):
        possibleneighborsr[i]=Vs.difference(set(
            returnzigzagcontainingvertex(Lz,i)))
        possibleneighborsr[i].discard(i)
        remainingdegree[i]=1
    for i in iter(V2):
        possibleneighborsr[i]=Vs.difference(set(
            returnzigzagcontainingvertex(Lz,i)))
        possibleneighborsr[i].discard(i)
        remainingdegree[i]=2
    for i in iter(V4):
        possibleneighborsr[i]=copy(Vs)
        possibleneighborsr[i].discard(i)
        remainingdegree[i]=4
    possibleneighbors=copy(possibleneighborsr)
    v=Vs.pop()
    Vs.add(v)
    possibleedgeslist=deque([deque([(v,i) for i in iter(
        possibleneighbors[v])])])
    edgeslist=deque()
    nofmissingedges=(len(V1)+2*len(V2)+4*len(V4))/2
    print(nofmissingedges)
    length=1
```

```
while length>0:
    if len(possibleedgeslist[-1])>0:
        edge=possibleedgeslist[-1].pop()
        edgeslist.append(edge)
        if len(edgeslist)<nofmissingedges:
            u=edge[0]
            v=edge[1]
            possibleneighbors[u].discard(v)
            possibleneighbors[v].discard(u)
            remainingdegree[u]-=1
            remainingdegree[v]-=1
            if remainingdegree[u]==0:
                Vs.discard(u)
                for i in iter(Vs):
                    possibleneighbors[i].discard(u)
            if remainingdegree[v]==0:
                Vs.discard(v)
                for i in iter(Vs):
                    possibleneighbors[i].discard(v)
            u=Vs.pop()
            Vs.add(u)
            possibleedgeslist.append(
                deque([(u,i) for i in iter(possibleneighbors[u])]))
            length+=1
        else:
            K=copy(S)
            K.add_edges(edgeslist)
            edgeslist.pop()
            if K.is_connected() and K.is_regular(
            ) and K.triangles_count()==noftri:
                if dtriancestor(K).order()==5:
                    iso=0
                    for D in iter(descendants):
                        if K.is_isomorphic(D):
                            iso=1
                            break
                    if not iso:
                        descendants.append(K)
                else:
                    iso=0
                    for I in iter(illegals):
                        if K.is_isomorphic(I):
                            iso=1
                            break
                    if not iso:
                        illegals.append(K)
    else:
```

```
            possibleedgeslist.pop()
            length-=1
            if len(edgeslist)>0:
                edge=edgeslist.pop()
                u=edge[0]
                v=edge[1]
                possibleneighbors[u].add(v)
                possibleneighbors[v].add(u)
                remainingdegree[u]+=1
                remainingdegree[v]+=1
                for i in iter(Vs):
                    Cu=u in possibleneighborsr[i]
                    Cv=v in possibleneighborsr[i]
                    if not conts(edgeslist,(i,u)) and not S.has_edge(i,u):
                        possibleneighbors[i].add(u)
                    if not conts(edgeslist,(i,v)) and not S.has_edge(i,v):
                        possibleneighbors[i].add(v)
                Vs.add(u)
                Vs.add(v)
    return descendants, illegals


##########################################################################

def skeleton(L):
    G=graphs.CompleteGraph(0)
    ind=0
    V1=[]
    V2=[]
    V4=[]
    for i in range(0,len(L)):
        if L[i]>0:
            zigzag(G,[j+ind for j in range(0,L[i]+2)])
            if L[i-1]<1:
                #if L[i-1]==-1:
                #    if i>=1:
                #        G.add_edge(ind-1,ind)
                #    V1.append(ind)
                #else:
                V2.append(ind)
            if L[i]>1:
                V1.append(ind+1)
                V1.append(ind+L[i])
            else:
                V2.append(ind+1)
            l=L[mod(i+1,len(L))]
            if l>0:
```

```
                ind+=L[i]+1
            elif l==0:
                V2.append(ind+L[i]+1)
                ind+=L[i]+2
            else:
                V1.append(ind+L[i]+1)
                ind+=L[i]+2
        elif L[i]==-1:
            G.add_vertex(ind)
            V4.append(ind)
            #if i>=1:
            #    G.add_edge(ind,ind-1)
            ind+=1
    if L[-1]>0 and L[0]>0:
        G.merge_vertices((0,ind))
    #elif L[0]==-1 or L[-1]==-1:
    #    G.add_edge(ind-1,0)
    return G,V1,V2,V4
```

```
##########################################################################
```

```
def returnzigzagcontainingvertex(L,v):
    for l in L:
        if conts(l,v):
            return l
    return []
```

```
##########################################################################
```

DTRfunctions.py

```
##########################################################################
```

```
def dtriancestor(G):
    A=copy(G)
    notri=0
    while not notri:
        dt=finddtri(A)
        if dt!=None:
            dtrired(A,dt[0],dt[1],dt[2],dt[3])
        else:
            notri=1
    return A
```

```
##########################################################################

def dtri(G,v1,v2,v3,c):
    G.delete_edge(v1,v2)
    vn=max(G.vertices())+1
    G.add_edges(((v1,vn),(v2,vn),(v3,vn)))
    S=setminus(G.neighbors(v3),(v1,v2,vn))
    if len(S)==1:
        c=0
    G.delete_edge(S[c],v3)
    G.add_edge(S[c],vn)
    return


##########################################################################

def dtrired(G,v1,v2,v3,v4):
    G.delete_edges([(v1,v4),(v2,v4),(v3,v4)])
    v=G.neighbors(v4)[0]
    G.add_edges([(v1,v3),(v,v2)])
    G.delete_vertex(v4)
    return


##########################################################################

def finddtri(G):
    A=copy(G)
    A1=A.vertex_iterator()
    for i in A1:
        A2=A.neighbor_iterator(i)
        for j in A2:
            A.delete_edge(i,j)
            A3=A.neighbor_iterator(j)
            for k in A3:
                if not A.has_edge(i,k):
                    A.delete_edge(j,k)
                    A4=A.neighbor_iterator(k)
                    for l in A4:
                        if A.has_edge(i,l) and A.has_edge(
                            j,l) and A.has_edge(k,l):
                            A.delete_edges([(i,l),(j,l),(k,l)])
                            m=A.neighbors(l)[0]
                            if not A.has_edge(j,m):
                                return [i,j,k,l]
                            A.add_edges([(i,l),(j,l),(k,l)])
                    A.add_edge(j,k)
```

```
            A.add_edge(i,j)
    return None


##########################################################################

def isk5dsc(G):
    A=copy(G)
    n=A.order()
    A.allow_multiple_edges(1)
    while n>5:
        dt=finddtri(A)
        if dt!=None:
            dtrired(A,dt[0],dt[1],dt[2],dt[3])
            n-=1
            print(n)
        else:
            print('notri')
            return 0
    print('iso')
    return A.is_isomorphic(graphs.CompleteGraph(5))

##########################################################################
```

DTRchainfunctions.py

```
##########################################################################
#listchains(G) creates two lists from G, the first of which is a list
# of lists (chains) of zigzag lengths. The second list is a list of lists
# of lists of zigzag vertices.
#
# Example output: [[3, 3]], [[[0, 6, 7, 4, 1], [1, 5, 8, 2, 3]]]
#

def listchains(G):
    A=copy(G)
    reversedir=0
    startnewchain=1
    chainlist=[]
    chainvertices=[]

    while A.triangles_count()>0:

        if startnewchain:
            Z=findzigzag(A,-1)
            A.delete_vertices(Z[1:-1])
            if A.subgraph(Z).is_regular(4):
```

```
            tri=len(Z)
        else:
            tri=len(Z)-2
        chainlist.append([tri])
        chainvertices.append([Z])
        startnewchain=0
        currfirstval=Z[0]
        currlastval=Z[-1]

    else:
        if reversedir:
            val=chainvertices[-1][0][0]
            Z=findzigzag(A,val)

            if not Z:
                A.delete_vertex(val)
                startnewchain=1
                reversedir=0
            else:
                if Z[0]==val:
                    Z.reverse()
                tri=len(Z)-2
                chainlist[-1].insert(0,tri)
                chainvertices[-1].insert(0,Z)
                A.delete_vertices(Z[1:-1])

        else:
            val=chainvertices[-1][-1][-1]
            Z=findzigzag(A,val)
            #print(val)
            #print(A.edges())

            if not Z:
                A.delete_vertex(val)
                reversedir=1
            else:
                if Z[-1]==val:
                    Z.reverse()
                tri=len(Z)-2
                chainlist[-1].append(tri)
                chainvertices[-1].append(Z)
                A.delete_vertices(Z[1:-1])

return chainlist, chainvertices
```

##

```
#findtri returns a triangle in G if s is unspecified. If s>=0, it
# returns a triangle containing s. If G containts no triangles,
# it returns 0.

def findtri(G,s=-1):
    A=copy(G)
    if s!=-1 and A.has_vertex(s):
        A1=iter([s])
    else:
        A1=A.vertex_iterator()
    for i in A1:
        for j in A.neighbor_iterator(i):
            A.delete_edge(i,j)
            for k in A.neighbor_iterator(j):
                if A.has_edge(i,k):
                    return [i,j,k]
            A.add_edge(i,j)
    return 0


##########################################################################
#findzigzag starts with a triangle and finds a maximal zigzag containing
# that triangle. If G contains no triangles, it returns 0.
# If s is specified, it returns a zigzag containing the vertex s if it
# exists, and 0 otherwise.

def findzigzag(G,s=-1):
    A=copy(G)
    if A.triangles_count()==0:
        return 0
    Z=findtri(A,s)
    if not Z:
        return 0

    L=[]
    for i in range(0,2):
        for j in range(i+1,3):
            L.append([Z[i],Z[j]])
            A.delete_edge([Z[i],Z[j]])
    k=0
    while len(L)>k:
        V=inter(A.neighbors(L[k][0]),A.neighbors(L[k][1]))
        if len(V)==0:
            k+=1
        else:
            v=V[0]
            Z.append(v)
```

```
            L.append([v,L[k][0]])
            L.append([v,L[k][1]])
            A.delete_edges([(v,L[k][0]),(v,L[k][1])])
            k+=1

    if len(Z)>3:
        H=G.subgraph(Z)
        for i in range(0,len(Z)):
            if H.degree(Z[i])==2:
                Zo=[Z[i]]
                break
        N=H.neighbors(Zo[0])
        for i in range(0,len(N)):
            if H.degree(N[i])==3:
                Zo.append(N[i])
                break
        H.delete_edge(Zo)

        i=0
        while H.size()>0:
            S=inter(H.neighbors(Zo[i]),H.neighbors(Zo[i+1]))
            Zo.append(S[0])
            H.delete_edges([(Zo[i],S[0]),(Zo[i+1],S[0])])
            i+=1
    elif len(Z)==3:
        Zo=Z
        if G.cluster_triangles(Z[1])==2:
            Zo=[Zo[0],Zo[2],Zo[1]]

        if s!=Z[0] and G.cluster_triangles(
            Z[0])==1 and G.cluster_triangles(Z[1])==2:
            Zo=[Zo[1],Zo[0],Zo[2]]

    return Zo


##########################################################################

def listtriangles(G):
    A=copy(G)
    S0=set()
    S1=set()
    S2=set()
    for v1 in A.vertex_iterator():
        for v2 in A.neighbor_iterator(v1):
            S=set([v1,v2])
            for v3 in A.neighbor_iterator(v2):
```

```
                if v3>v2 and v3 not in S and A.has_edge(v1,v3):
                    S=set([v1,v2,v3])
                    S0.add((v1,v2,v3))
                    for v4 in A.neighbor_iterator(v2):
                        if v4 not in S and A.has_edge(v2,v4) and A.has_edge(
                            v3,v4):
                            S1.add((v1,v2,v3))
                    for v4 in A.neighbor_iterator(v1):
                        if v4 not in S:
                            S=set([v1,v2,v3,v4])
                            for v5 in A.neighbor_iterator(v4):
                                if v5>v4 and v5 not in S and A.has_edge(
                                    v1,v5):
                                    S2.add((v1,v2,v3))
    S12=S1.union(S2)
    T2=S1.intersection(S2)
    T1=S12.difference(T2)
    T0=S0.difference(S12)
    return T0,T1,T2

##############################################################################

def listuniquetriangles(G):
    chainlist,chvertexlist=listchains(G)
    trilist=listtriangles(G)
    trilist0=trilist[0]
    newT0=set()
    found=dict()
    for i in range(0,len(chainlist)):
        for j in range(0,len(chainlist[i])):
            if chainlist[i][j]>=1:
                tpl=iter([i,j])
                for k in iter(chvertexlist[i][j][1:-1]):
                    toremove=set()
                    for tri in iter(trilist0):
                        if tri[0]==k:
                            if tpl not in found:
                                newT0.add(tri)
                                found[tpl]=1
                            toremove.add(tri)

                    trilist0=trilist0.difference(toremove)
    return newT0,trilist[1],trilist[2]
```

generalfunctions.py

```
##########################################################################
# Some set functions

def inter(A,B):
    C=[]
    for i in range(0,len(A)):
        for j in range(0,len(B)):
            if A[i]==B[j]:
                C.append(A[i])
    return C

def conts(A,a):
    cont=0
    for i in range(0,len(A)):
        if a==A[i]:
            cont=1
            break
    return cont

def setminus(A,B):
    C=[]
    for i in range(0,len(A)):
        if not conts(B,A[i]):
            C.append(A[i])
    return C


##########################################################################
#cyclicpermutations produces an iterable of a cyclic permutation of
# length "length".

def cyclicpermutations(length):
    S=[i for i in range(0,length)]
    index=0
    while 1:
        yield S
        index+=1
        if index==length:
            raise StopIteration
        S=[S[mod(i+1,length)] for i in range(0,length)]


##########################################################################

def sumvectors(v1,v2):
    return tuple([v1[i]+v2[i] for i in range(0,len(v1))])
```

```
##########################################################################

def modvector(vector,modulus):
    workingvector=list(vector)
    for i in range(len(vector)-1,0,-1):
        v=workingvector[i]
        m=modulus[i]
        quotient=int(v/m)
        remainder=v-m*quotient
        workingvector[i]=remainder
        workingvector[i-1]+=quotient
    workingvector[0]=mod(workingvector[0],modulus[0])
    return tuple(workingvector)


##########################################################################

def itertuples(moduluslist):
    unit=[0 for i in range(0,len(moduluslist)-1)]
    zerovector=copy(unit)
    zerovector.append(0)
    zerovector=tuple(zerovector)
    unit.append(1)
    unit=tuple(unit)
    tpl=zerovector
    while 1:
        yield tpl
        tpl=modvector(sumvectors(tpl,unit),moduluslist)
        if tpl==zerovector:
            raise StopIteration


##########################################################################

def chainsignlist(lengthlist,symmetriclist):
    S=[itertuples([symmetriclist[i]+1 for j in range(
        0,lengthlist[i])]) for i in range(0,len(lengthlist))]
    CS=[S[i].next() for i in range(0,len(S))]
    index=0
    found=0
    while 1:
        yield CS
        while not found:
            try:
                CS[index]=S[index].next()
                found=1
```

```
            except StopIteration:
                S[index]=itertuples([symmetriclist[
                    index]+1 for j in range(
                    0,lengthlist[index])])
                CS[index]=S[index].next()
                index+=1
                if index==len(lengthlist):
                    raise StopIteration
        found=0
        index=0


##########################################################################
#chainpermutations takes a list of positive integers, and produces
# as an output an iterable of a list of permutations of lengths as in
# lengthlist.

def chainpermutations(lengthlist):
    P=[itertools.permutations(range(0,lengthlist[i])) for i in range(
        0,len(lengthlist))]
    CP=[P[i].next() for i in range(0,len(P))]
    index=0
    found=0
    while 1:
        yield CP
        while not found:
            try:
                CP[index]=P[index].next()
                found=1
            except StopIteration:
                P[index]=itertools.permutations(range(0,lengthlist[
                    index]))
                CP[index]=P[index].next()
                index+=1
                if index==len(lengthlist):
                    raise StopIteration
        found=0
        index=0


##########################################################################

def symmetricvector(v):
    return list(v)==list(reversed(v))


##########################################################################
```

PDvectorrepresentation.py

```
#########################################################################

def standardvector(G):

    if iszigzag(G):
        return tuple([tuple([G.triangles_count()]),tuple()])

    K=pdrelabel(G)

    #This piece of code constructs an ordered list of edges that are not
    # in chains. These will be the "chords" in our graph.
    #

    chainlist,chainvertexlist=listchains(K)
    chainlist,chainvertexlist=orderchainlist(chainlist,chainvertexlist)
    chainedgelist=[]
    lonevertices=deepcopy(K.vertices())
    for chain in iter(chainvertexlist):
        for zigzag in iter(chain):
            chainedgelist.extend(K.subgraph(zigzag).edges())
            for i in range(0,len(zigzag)):
                try:
                    lonevertices.remove(zigzag[i])
                except:
                    continue

    H=deepcopy(K)
    H.delete_edges(chainedgelist)
    E=deepcopy(H.edges())

    position=positionsofvertices(H,chainvertexlist,lonevertices)
    #print(position)
    poc=dict() #positions of chords

    for i in range(0,len(E)):
        u,v=E[i][0],E[i][1]
        pos1 = next(key for key, value in position.items() if u in set(value))
        pos2 = next(key for key, value in position.items() if v in set(value))
        poc[i]=[pos1,pos2]

    chordlengthvector=[abs(poc[i][1]-poc[i][0]) for i in range(0,len(E))]
    chainv=chainvector(chainlist,len(lonevertices))

    return chainv,chordlengthvector
```

```
##########################################################################

def chainvector(chainlist,lvcount):
    cv=[]
    cond0=len(chainlist)==1 and lvcount==0

    for chain in iter(chainlist):
        for Z in iter(chain):
            cv.append(Z)
        if not cond0:
            cv.append(0)

    if lvcount>0:
        cv.append(-lvcount)

    return tuple(cv)


##########################################################################

def positionsofvertices(G,chainvertexlist,lvlist):
    positionindex=0
    position=dict()

    for chain in iter(chainvertexlist):
        for zigzg in iter(chain):
            if len(chain)==1:
                zigzag=sorted(zigzg)
            else:
                zigzag=zigzg
            for i in range(0,len(zigzag)):
                if G.degree(zigzag[i])==2:
                    position[positionindex]=[zigzag[i]]
                    positionindex+=1
                elif G.degree(zigzag[i])==1:
                    if len(zigzag)==4:
                        if positionindex not in position:
                            position[positionindex]=[zigzag[i]]
                        else:
                            position[positionindex].append(zigzag[i])
                            positionindex+=1
                    else:
                        position[positionindex]=[zigzag[i]]
                        positionindex+=1

    if len(lvlist)>0:
```

```
        vertex=lvlist[0]
        l=-len(lvlist)
        while l<0:
            position[positionindex]=[vertex]
            positionindex+=1
            vertex+=1
            l+=1

    return position

##########################################################################

def pdrelabel(G,outputcolor=0,colorlist=0):
    if iszigzag(G):
        return G

    chainlist,chainvertexlist=listchains(G)
    chainvertexlist,chainlist=orderchainlist(chainvertexlist,chainlist)

    #H=deepcopy(G)
    K=deepcopy(G)
    lv=K.vertices()
    index=0
    colorindex=0
    label=dict()
    color=dict()
    rcolor=dict()

    if not colorlist:
        colorlist=['blue', 'green', 'red', 'cyan', 'm', 'yellow',
                   'black', 'white']

    for chain in iter(chainvertexlist):
        for zigzag in iter(chain):

            for i in range(0,len(zigzag)):
                if not zigzag[i] in label:
                    label[zigzag[i]]=index
                    rcolor[index]=colorlist[colorindex]
                    if colorlist[colorindex] in color:
                        color[colorlist[colorindex]].append(index)
                    else:
                        color[colorlist[colorindex]]=[index]
                    index+=1
                    lv.remove(zigzag[i])
            colorindex+=1
```

```
    for vertex in iter(lv):
        label[vertex]=index
        rcolor[index]=colorlist[colorindex]
        color[colorlist[colorindex]]=[index]
        index+=1
        colorindex+=1

    K.relabel(label)

    if outputcolor:
        return K,color,rcolor
    else:
        return K


##########################################################################

def orderchainlist(chainlist,chainvertexlist):
    for i in range(0,len(chainlist)):
        C=chainvertexlist[i]
        c=chainlist[i]
        C,c=orderchain(C,c)
        c=list(c)
        chainvertexlist[i]=C
        c.append(-i)
        chainlist[i]=tuple(c)

    chainlist=mergesort(chainlist)
    newchainvertexlist=[]

    for i in range(0,len(chainlist)):
        identifier=-chainlist[i][-1]
        chainlist[i]=tuple(chainlist[i][0:-1])
        newchainvertexlist.append(chainvertexlist[identifier])

    return chainlist,newchainvertexlist

##########################################################################
# obtained from pythonandr.com/2015/07/05/the-merge-sort-python-code/
# Author: Anirudh Jay (pythonandr.com/author/anirudhjay/)

def merge(a,b):
    """ Function to merge two arrays """
    c = []
    while len(a) != 0 and len(b) != 0:
        if a[0] < b[0]:
            c.append(a[0])
```

```
            a.remove(a[0])
        else:
            c.append(b[0])
            b.remove(b[0])
    if len(a) == 0:
        c += b
    else:
        c += a
    return c

# Code for merge sort

def mergesort(x):
    """ Function to sort an array using merge sort algorithm """
    if len(x) == 0 or len(x) == 1:
        return x
    else:
        middle = len(x)/2
        a = mergesort(x[:middle])
        b = mergesort(x[middle:])
    return merge(a,b)


##############################################################################

def orderchain(C,c):
    maxC=C
    maxc=tuple(c)
    n=len(c)

    for P in cyclicpermutations(n):
        for i in range(0,2):
            if i:
                newc=tuple([c[P[n-1-i]] for i in range(0,n)])
            else:
                newc=tuple([c[P[i]] for i in range(0,n)])

            if newc>maxc:
                maxc=newc
                maxC=[C[P[i]] for i in range(0,n)]

    return maxC,maxc


##############################################################################

def chainsize(C,c):
```

```
    size=0
    for i in range(0,len(c)):
        size+=c[i]
    return size


##########################################################################
```

PDgraphrepresentation.py

```
#############################################################################
# This function converts a standard vector representation to a graph.
# If using a chord vector, use
#   convertsvtograph(chainlist,cl=chordvector).
# If using a chord length vector clv, use
#   convertsvtograph(chainlist,cl=clv).


def convertsvtograph(chainlist,cl=0,cll=0):
    newchainlist=list(chainlist)

    if len(chainlist)==1:
        return createZ(chainlist[0])

    #This step divides the lone vertices, so that it can be used as
    # input for the skeleton() function. We convert, e.g., [1,2,0,-3]
    # to [1,2,0,-1,-1,-1].
    l=newchainlist[-1]
    if l<0:
        newchainlist[-1]=-1
        l+=1
        while l<0:
            newchainlist.append(-1)
            l+=1

    #skeleton(chainlist) creates a skeleton that has the corresponding
    # chainlist. There are no chord edges yet, and those will be added
    # later.
    G=skeleton(newchainlist)[0]
    chainvertexlist=listchains(G)[1]
    L=deepcopy(G.vertices())
    #G.allow_multiple_edges(1)

    #The following piece of code calculates the position of each vertex
    # based on the definition of the standard vector, and assigns
    # chords to each position.
    positionindex=0
    if cl:
        chordindex=0
        chords=dict()
    position=dict()
    valency=dict()
    newchainvertexlist=[]
    for chain in iter(chainvertexlist):
        for zigzg in iter(chain):
```

```
            zigzag=sorted(zigzg)
            for i in range(0,len(zigzag)):
                try:
                    L.remove(zigzag[i])
                except:
                    continue

                if G.degree(zigzag[i])==2:
                    position[positionindex]=[zigzag[i]]
                    if cl:
                        chords[positionindex,0]=[
                            chordlist[chordindex],chordlist[chordindex+1]]
                        chordindex+=2
                    valency[positionindex,0]=2
                    positionindex+=1
                elif G.degree(zigzag[i])==3:
                    if len(zigzag)==4:
                        if positionindex not in position:
                            position[positionindex]=[zigzag[i]]
                            valency[positionindex,0]=1
                            if cl:
                                chords[positionindex,0]=[
                                    chordlist[chordindex]]
                                chordindex+=1
                        else:
                            position[positionindex].append(zigzag[i])
                            valency[positionindex,1]=1
                            if cl:
                                chords[positionindex,1]=[
                                    chordlist[chordindex]]
                                chordindex+=1
                            positionindex+=1
                    else:
                        position[positionindex]=[zigzag[i]]
                        valency[positionindex,0]=1
                        if cl:
                            chords[positionindex,0]=[
                                chordlist[chordindex]]
                            chordindex+=1
                        positionindex+=1

        #This piece of code addes the lone vertices to the position and chord
        # dictionaries.
        if len(L)>0:
            vertex=L[0]
            l=-len(L)
            while l<0:
```

```
            position[positionindex]=[vertex]
            valency[positionindex,0]=4
            if cl:
                chords[positionindex,0]=[
                    chordlist[chordindex+i] for i in range(0,4)]
                chordindex+=4
            positionindex+=1
            vertex+=1
            l+=1

    #This piece of code calculates the condensed chordlengthlist.
    if cl:
        fpos,lpos=dict(),dict()
        chordlengthlist=[]
    if cl:
        for pos in range(0,positionindex):
            for subpos in range(0,len(position[pos])):
                for chord in iter(chords[pos,subpos]):
                    if chord not in fpos:
                        fpos[chord]=pos
                    else:
                        lpos[chord]=pos

    #print(chords)
    #print(fpos)
    #print(lpos)

    if cl:
        chordlengthlist=[lpos[i]-fpos[i] for i in range(
            1,len(chordlist)/2+1)]

    #This piece of code adds the remaining edges to the skeleton graph.
    H=deepcopy(G)
    chordindex=1
    posindex=0
    if not cl:
        chordlengthlist=deepcopy(cll)
    #print(valency)
    #print(chainvertexlist)
    #print(position)
    #print(chordlengthlist)

    while chordindex<len(chordlengthlist)+1:
        nextpos=1
        while nextpos:
            for i in range(0,len(position[posindex])):
                if valency[posindex,i]>0:
```

```
                    fv=position[posindex][i]
                    valency[posindex,i]-=1
                    nextpos=0
                    break
            if nextpos:
                posindex+=1
        lpos=posindex+chordlengthlist[chordindex-1]

        nextpos=1
        while nextpos:
            for i in range(0,len(position[lpos])):
                if valency[lpos,i]>0:
                    lv=position[lpos][i]
                    valency[lpos,i]-=1
                    nextpos=0
                    break
            if nextpos:
                lpos+=1
        H.add_edge(fv,lv)
        chordindex+=1

    return H


##########################################################################
#This function creates the graph of Zn, with n as input.
#

def createZ(n):
    G=Graph()
    G.add_vertices([i for i in range(0,n+2)])
    G.add_edges([i,mod(i+1,n+2)] for i in range(0,n+2))
    G.add_edges([i,mod(i+2,n+2)] for i in range(0,n+2))
    return G


##########################################################################

def pdplot(G,returnlatex=0,filename='fig',graphicsize=(8,8),
           vertexlabels=1):
    from sage.graphs.graph_latex import check_tkz_graph
    K,color,colorlist=pdrelabel(G,outputcolor=1)

    if returnlatex:
        check_tkz_graph()
        K.set_pos(K.layout_circular(radius=3))
        plt=plot(K,layout='circular',vertex_colors=color,vertex_size=400)
```

```
            K.set_latex_option(tkz_style,'Normal')
            K.set_latex_options(vertex_label_colors=colorlist,
                                graphic_size=graphicsize)
            if not vertexlabels:
                K.set_latex_option(vertex_labels,0)
            opts=K.latex_options()
            with open(filename+".tex", "wb") as f:
                f.write(opts.tkz_picture())
            return plt

    return plot(K,layout='circular',vertex_colors=color)
```

##

createfigures.py

```
##########################################################################

#filename1='k5d50300'
#filename2='nonk5d50300'
#graphicsize=(6,6)
#G=Gseq[13][8][2][1][0]
#SV=standardvector(G)
#H=convertsvtograph(SV[0], cll=SV[1])
#H.delete_edges([[5,12],[6,7]])
#H.add_edges([[6,12],[5,7]])
#pdplot(G,1,filename1,graphicsize)
#pdplot(H,1,filename2,graphicsize)


##########################################################################
graphicsize=(4,4)
with open("figurefilenames.tex", "wb") as f:
    for n in range(0,len(Gseq)):
        for t in range(0,len(Gseq[n])):
            for c in range(0,len(Gseq[n][t])):
                for l in range(0,len(Gseq[n][t][c])):
                    for i in range(0,len(Gseq[n][t][c][l])):
                        G=deepcopy(Gseq[n][t][c][l][i])
                        filename='k5dlistn'+str(n)+'t'+str(t)+'c'+str(
                            c)+'l'+str(l)
                        pdplot(G,1,filename,graphicsize,vertexlabels=0)
                        f.write('\input{'+filename+'.tex}\n')

##########################################################################
```