COMPUTATIONS OF COMBINATORIAL MULTIFILTERED LINK FLOER COMPLEXES

By

Christopher St Clair

A DISSERTATION

Submitted to
Michigan State University
in partial fulfillment of the requirements
for the degree of

Mathematics—Doctor of Philosophy

2024

ABSTRACT

We consider how to effectively compute the combinatorial multigraded link Floer homology of knots and links, then implement it. The goal being to compute the invariant for knots and links for which it is unknown, and further develop the algorithm for its production.

We review a traditional description of knot Floer homology and motivate it. Then we move to the combinatorial description of the theory known as grid homology. From there we consider and implement an algorithm leveraging some changes in perspective and tools afforded by modern computing.

Finally, included at the end is a compilation of some standard properties of the invariant for examples computed using the resulting program. From the resulting directed graphs we extract the Poincaré polynomial along with some properties of the invariant for each sample knot.

Dedicated to my family, including my cats.

ACKNOWLEDGEMENTS

My sincerest thanks and gratitude go to my advisor Matthew Hedden. He guided and taught me not just mathematics and research skills but embodies who I want to be as a mentor. His support, unwavering patience and openness to my ideas and questions go above and beyond.

I want to thank my committee members Matthew Stoffregen, Efstratia Kalfagianni and Elizabeth Munch. Each of them has been so generous with their time to meet with me whenever I have wanted their help or perspective.

My fellow students and researchers also deserve credit for their help. Across cohorts, my colleagues and friends have always been there to talk, teach and support eachother. Particular thanks to Tristan Wells, Chen Zhang, Christopher Potvin and Joseph Melby for our frequent talks.

Across six difficult years my family has always been there for me. Their love and support made the journey possible. Thanks to my mom Paula, my dad Mark, and my brothers Alex and Lester along with their families. They brought joy, laughter and stability that was sorely needed.

Finally, my cats Galois and Noether were by my side through it all. Without them I would have spent a lot more time alone. They always listened and distracted me when I needed it most.

TABLE OF CONTENTS

CHAPTER 1

INTRODUCTION

Knot theory is an active field of interest sporting a quirky flavor and array of applications. These range from literal knots people tie, to knots that proteins and DNA tie to knots that topologists "tie". Being a topic that has existed longer than rope, it has been approached in a multitude of ways. We will help the effort by exploring and leveraging a modern tool developed by low dimensional topologists in pursuit of a tool for efficiently studying 3 and 4 manifolds.

That tool is knot Floer homology (HFK), and its invariants. We will touch on the original formulation of the invariant but by the end of our exploration we will have a program that computes a combinatorial version of HFK defined by Manolescu et al. in [11] known as grid homology. Prior work has been done in this direction exists computing whats known as the (dramatically) simplified, unblocked complex in [1]. We will approach the problem differently and end up with the more robust "infinity" complex.

Knot homology has proven to be a strong tool thus far, associating a a chain complex, or more generally a multi-graded module with an endomorphism, to a knot or link. It tells us knot genus (by extension, it detects the unknot), sliceness invariants and invariants of manifolds obtained by surgery [10]. The invariant is powerful but finicky; computing it relies heavily on how a knot is presented and is rarely straightforward. Grid homology solves the problem in part by restricting to a rigid class of diagrams. However, we have to pay for the solution by having a very complex presentation of our complex.

Knots and links will be represented on a sort of $n \times n$ tic-tac-toe style grid. The complexity issue comes down to the complex having $n!$ generators and, naively, potentially $n!(n-1)!$ entries in the matrix encoding the differential. By the end of chapter 2 we will have simplified the problem some and implemented some tricks to have a program to compute and simplify the invariant for knots and, notably, links through grid size 8 and some examples of size 9.

1.1 Knot Theory

Knot theory is the study of embeddings of one (or more in the case of links) copies of S^1 into different manifolds, up to different notions of equivalence. The simplest and most common formulation is an embedding $K : S^1 \hookrightarrow S^3$ up to ambient isotopy. Unless otherwise specified, we will assume this notion of knots and equivalence. If we are considering embedding several copies of S^1 simultaneously we will refer to these as links with n components $L : \bigsqcup_{i=1}^{n} S^1 \hookrightarrow S^3$

These definitions, although natural, are not very practical. Thankfully there is an equivalent description of knots which is more useful to actually work with. Given any knot in S^3, we will project down to some plane, indicating over and under crossings. Additionally we require the choice of projection to avoid triple points.

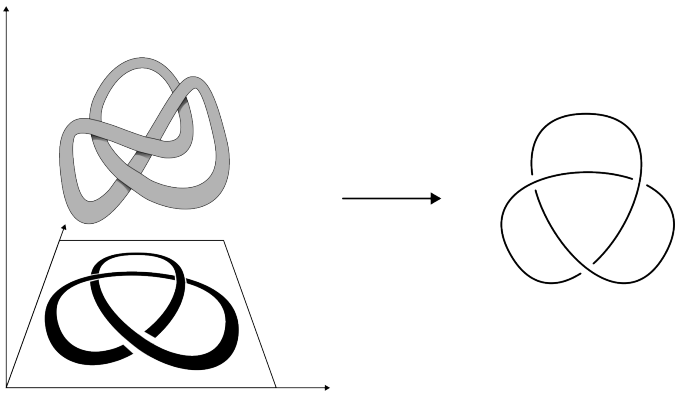


Figure 1.1 Projecting a knot down onto a plane to get a diagram

We will take this projection as defining a knot, and say that two knots are equivalent when they differ by a finite set of Reidemeister moves as in 1.2.

Theorem 1.1.1 (Reidemeister)**.** *Two knots are equivalent if and only if they can be represented by diagrams which differ by a finite sequence of the moves from* ***R0 - R4***

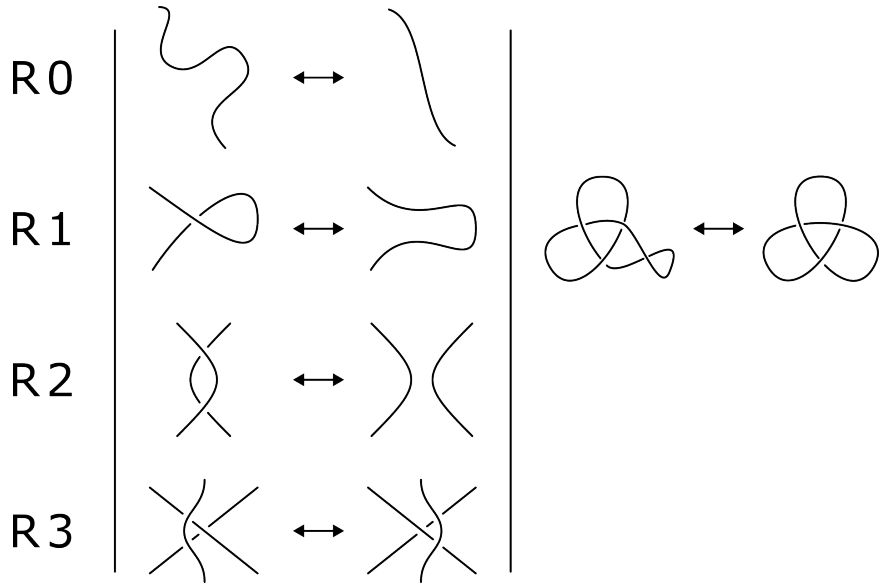


Figure 1.2 The Reidemeister moves

An overarching goal of knot theory is to be able to distinguish knots and links from one another. In pursuit of this we search for properties of knots that do not depend on a given presentation of a knot. In terms of projections, these are properties that do not change under the Reidemeister moves.

A classic example of a knot invariant is the Alexander polynomial Δ which associates a polynomial to a knot.

$$\Delta(K) = \Delta_K(t) = f(t)$$

The heart of this thesis and the associated program is a combinatorial description of knot Floer homology, which can be thought of as a generalization of the Alexander Polynomial. The invariant was discovered by Ozsváth and Szabó [13], and independently by Rasmussen [15]. It is a robust and well studied set of tools, with excellent introductions and surveys available, [10], [7], and [12], to name a few. We will review it briefly here as well.

The Heegaard Floer knot package has a variety of flavors. These are all similar invariants that associate a a multigraded module with an endomorphism to a knot or link up to equivalence. (In general, it is safe to think of each flavor as associating a chain complex to a knot or link. In the simplest cases this is exactly true, and in the more complex ones we

will relax the boundary map condition). To recover the Alexander Polynomial we will take a graded Euler characteristic of our module, see Theorem 1.1.2 later.

The invariant is defined using Heegaard diagrams for knots. A Heegaard diagram is a handle description of a 3-manifold; it consists of a genus g surface called Σ with sets of g disjoint, homologically linearly independent closed curves called alpha curves and another such set of g curves called beta curves which intersect the first transversely.

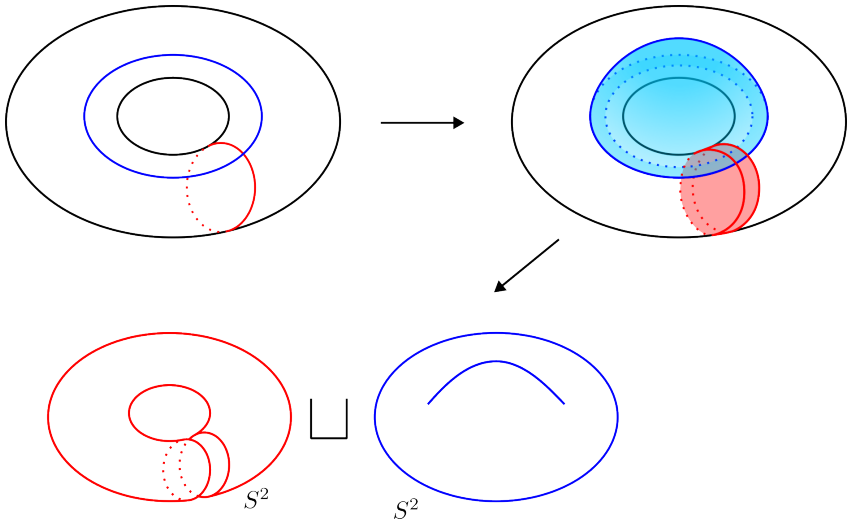


Figure 1.3 Heegaard diagram for S^3

To build a three manifold given such a diagram H we thicken Σ to $\Sigma \times [-1, 1]$. Then we attach 2 handles along the alpha curves at $[\alpha_i, -1]$ and also along the betas at $[\beta_i, 1]$. This will necessarily leave us with a manifold with boundary $S^2 \sqcup S^2$. Without thickening we can see this in figure 1.3. To this manifold we attach two 3-balls to each boundary component. We will call this manifold M and H is a Heegaard diagram describing it. Note that, strictly speaking, we also must specify a basepoint for our Heegaard diagram. It is also worth noting that there are infinitely many Heegaard diagrams for any given 3 manifold.

The reason for looking at these diagrams is because a Heegaard diagram for a knot in a 3-manifold is exactly a Heegaard diagram for that manifold with two basepoints chosen. To recover a knot from a diagram like this you connect the basepoints with two arcs. One arc entirely inside of the alpha handles and one entirely inside of the beta handles we attached. A shorthand for this is to connect the points via two curves along the surface where one

curve avoids the alpha curves and the other avoids the beta. An example of this process can be seen on the left side of figure 1.4.

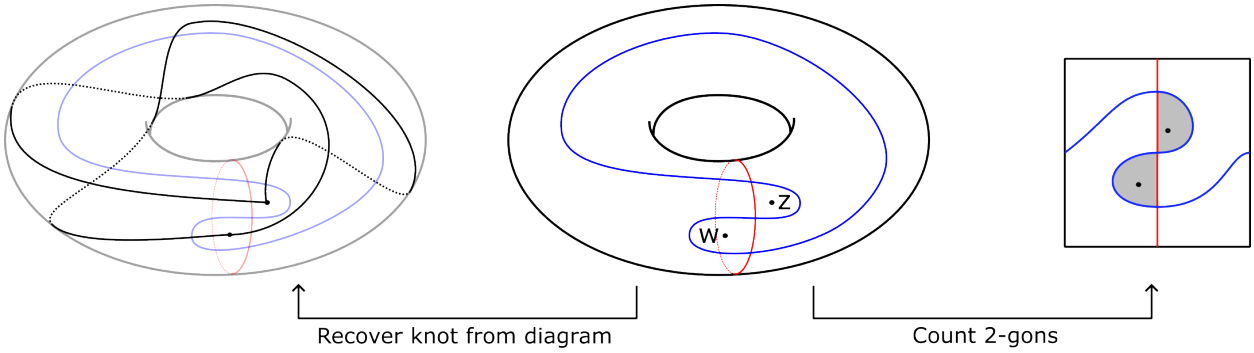


Figure 1.4 Recovering the trefoil from a Heegaard diagram, and the relevant disks for CFK

With such diagrams, we can compute knot Floer homology. Knot Floer homology comes in many flavors $CFK^\infty, CFK^-, \widehat{CFK}$ and more. To refer to all the flavors at once we will use the notation CFK°. Each of the flavors carries a bigraded module structure, along with an endomorphism typically referred to as the boundary map (though again this is a bit of a misnomer as some flavors are not chain complexes; similarly the module may be referred to as a complex as well).

Assuming we have a proper diagram, then the generators of our module are g tuples of intersections between the alpha and beta curves, such that each curve contains exactly one intersection point. In the case of a torus, the generators then are simply the intersections of a single alpha and beta curve. The endomorphism is defined by counting maps of disks (as 2-gons) into the branched cover of the Heegaard diagram with corners at generators. An illustrative example is for $CFK^\infty(3_1)$. First, we will define the boundary map of CFK^∞:

$$\partial x = \sum_{y \in \alpha \cap \beta} \sum_{\substack{\phi \in \pi_2(x,y) \\ \text{valid } \phi}} U^{n_w(\phi)} V^{n_z(\phi)} \cdot y$$

Where U and V are formal variables and and ϕ is a homotopy class of maps of a disk parameterized as in figure 1.5. We denote the intersection number of a disk with a basepoint z or w by n_z and n_w. The details for which disks are valid are outlined in the references.

We are glossing over it here because in the combinatorial description we will be leaning on, the disks in the differential are always valid.

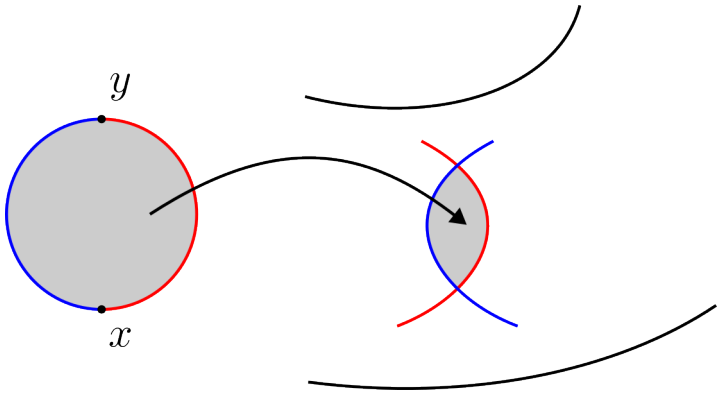


Figure 1.5 Image of map of a disk into a Heegaard diagram

So to put this into practice for the trefoil, we count the disks in figure 1.6. We find the complex is generated then by a, b and c, and the differential of each generator is:

$$\begin{aligned} \partial a &= Vb \\ \partial b &= 0 \\ \partial c &= Ub \end{aligned}$$

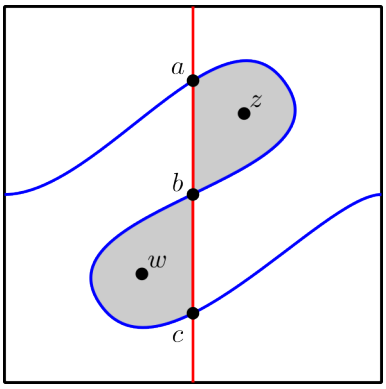


Figure 1.6 Heegaard diagram for the trefoil with its two relevant disks shaded

The gradings are referred to as the Maslov, or homological, gradings and Alexander gradings, the latter of which is a linear combination of the two. There is a Maslov grading for each basepoint, we will refer to them as Maslov U and V gradings and write M_U and M_V. The boundary map lowers these Maslov degrees by 1. The Alexander grading of an element x will be defined as:

$$A(x) = \frac{1}{2}(M_U(x) - M_V(x))$$

With the Alexander and Maslov gradings defined we can see how the invariant contains the Alexander polynomial. From CFK^∞ we convert to $\widehat{CFK}$ by setting $U = V = 0$. Taking homology of resulting complex we obtain $\widehat{HFK}$. To refer to a specific grading it is traditionally written as $\widehat{HFK}_{M_U}(K, A)$. Then the following theorem gives the relationship with $\Delta_K(t)$.

Theorem 1.1.2 ([13])**.** *The Euler characteristic of $\widehat{HFK}$ returns the Alexander polynomial where $|-|$ denotes rank.*

$$\Delta_K(t) = \sum_{M_U, A} (-1)^{M_U} |\widehat{HFK}_{M_U}(K, A)| \cdot t^A$$

1.2 Grid Presentations

Although CFK° are a powerful set of invariants, it is difficult to find diagrams for links that result in straightforward computations. There is an alternative way to present knots which helps here. They are known as grid presentations, or equivalently arc presentations.

Definition 1.2.1. *A **grid diagram** of size n is an $n \times n$ square grid, with one X and one O in every row and column, and with no two symbols in the same position.*

The placements of these X and O's will be referred to as a pair of permutations $\sigma_X, \sigma_O \in S_n$, where $\sigma_X(i)$ is the row that the X is placed in the ith column, and similarly for σ_O. A grid diagram specifies knot or link by connecting the X's to O's vertically, and O's to X's horizontally, and at any intersection making the vertical strand the over strand. Figure 1.7 shows this for the Hopf link given by $\sigma_X = [1, 2, 3, 4]$ and $\sigma_O = [3, 4, 1, 2]$. Note that generally we will be using one line notation for permutations since they easily encode these diagrams.

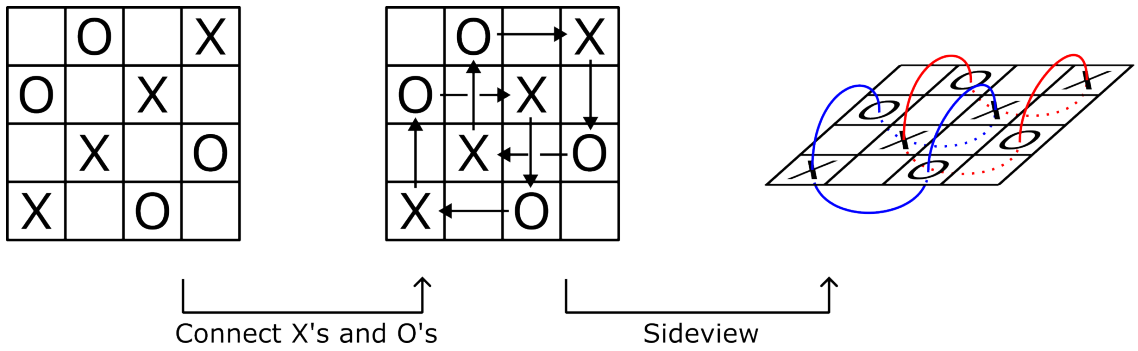


Figure 1.7 A grid diagram for the hopf link

Grid diagrams were introduced by Cromwell in [3], but a thorough treatment of grid diagrams can be found in [14] and our conventions will match theirs as much as possible. We will also mimic their proof of the following Lemma 1.2.2.

Lemma 1.2.2. *Every link can be represented as a grid diagram*

Proof. For a link L choose a planar presentation. Approximate this presentation with a piecewise linear one with only horizontal and linear segments. Then at any crossing where a horizontal strand goes over a vertical one, replace it following the procedure in figure 1.8 (imagine grabbing the strands of the crossing and twisting them 90°). Perturb any strands that are horizontal and at the same height or similarly vertically aligned. Finally label the corners alternating X and O respecting the orientation. □

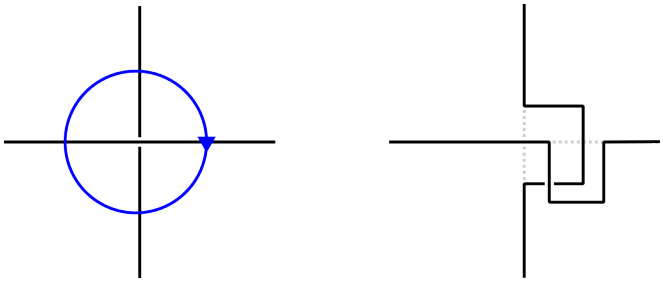


Figure 1.8 Switching a horizontal over crossing to a vertical one

With the description of link presentations as grid diagrams we have a collection of moves that function like the Reidemeister moves. We will refer to these as grid moves. The proof of 1.2.5 can be found in [3].

Definition 1.2.3 (Grid Move 1: Commutation)**.** *Two diagrams differ by a row (column) commutation if exchanging two adjacent rows (columns) transforms one into the other and the intervals defined by the symbols in those rows (columns) are entirely disjoint, or one is a subset of the other.*

This first move can be thought of as $R0$ when the strands are completely disjoint, or as $R2$ when one contains the other; see figure 1.9. Note that if the intervals only partially overlapped we would introduce linking between the strands. In that case this is referred to as a *cross commutation.*

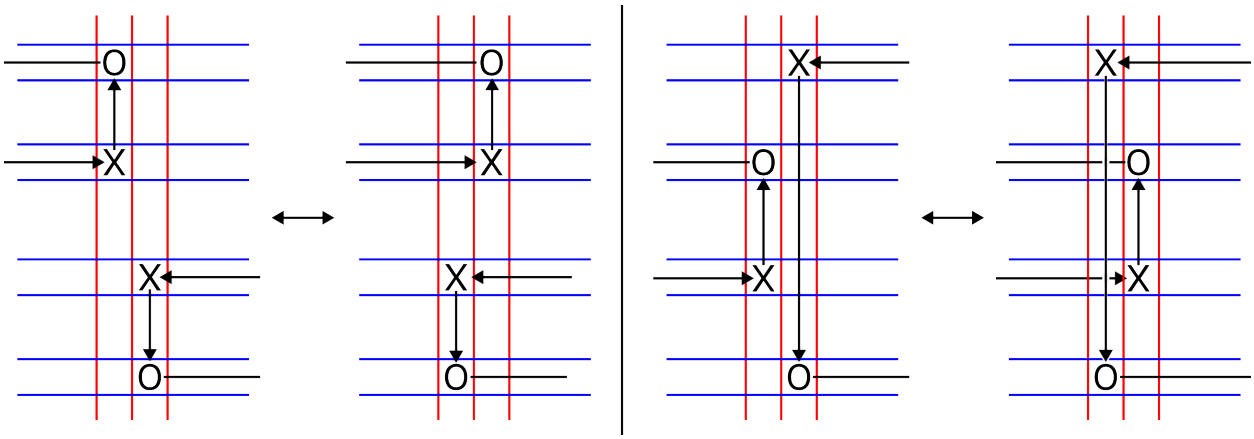


Figure 1.9 The two types of column commutation for grid moves

Definition 1.2.4 (Grid Move 2: Stabilization)**.** *A diagram can be stabilized to a larger diagram at an X or an O in four ways. In the case of X (or equivalently O) we refer to these as X:NE, X:NW, X:SE, X:SW. At the selected symbol split its row and column into two; in the resulting 2×2 s require the compass direction specified to be empty. The inverse of this move is Destabilization.*

The requirement of the empty square in 1.2.4 is sufficient to determine the placement of the remaining symbols in the resulting diagram. See figure 1.10

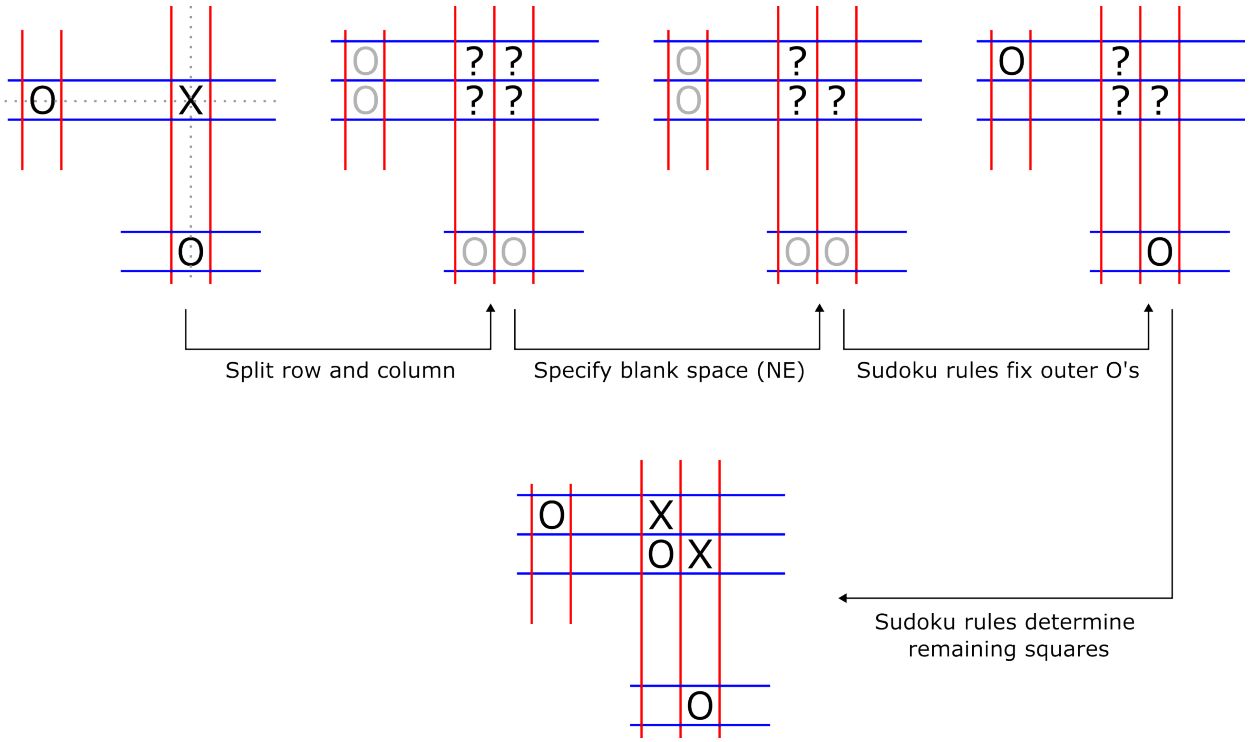


Figure 1.10 Process of producing X^{NE} stabilization. One of four possible X stabilizations and how a direction specifies the placement

Theorem 1.2.5 ([3]). *Two grid diagrams represent the same link if and only if they differ by a finite sequence of grid moves.*

1.3 Grid Homology

We discussed knot Floer homology, and with the aid of grid diagrams we can outline its combinatorial description, grid homology. It was defined in [11] and there is an excellent book entirely on the subject [14]. We will recite the details relevant to the program here. We are going to view the grid diagrams as highly pointed Heegaard diagrams. A $n \times n$ grid diagram will be a torus with n alpha and beta curves, and $2n$ basepoints. The alpha curves are the vertical dividing lines of the diagram, the betas are horizontal, and the basepoints are the X and O symbols.

When a diagram has multiple alpha and beta curves, CFK° is defined using a symmetric product. Without going into detail this shakes out to mean that the complex for grid diagrams will be generated by complete pairings of alpha and beta curves called grid states.

Each grid state consists of n intersections of alpha and beta curves, such that each alpha and beta curve only a single intersection. See figure 1.11 for an example of a pair of such grid states.

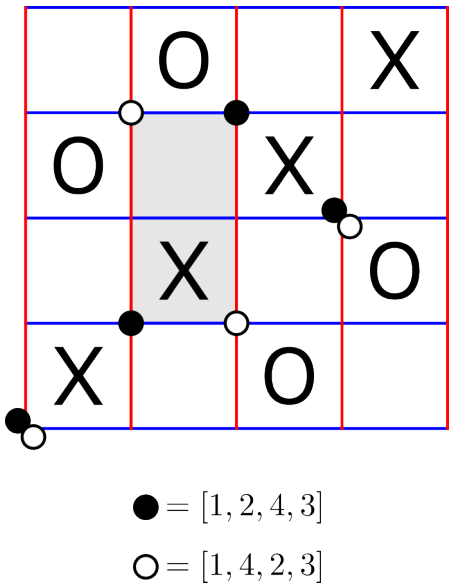


Figure 1.11 A pair of grid states and a rectangle connecting them

We specify the circle pairing by naming which horizontal circle is paired for each vertical circle, left to right. So for example, if we call the state specified by the black circles in figure 1.11 x then we can say the third alpha circle is paired with the fourth beta circle; or rather that $x(3) = 4$ and we can see that these grid states are elements of S_n.

To a grid diagram of a knot, we associate a bigraded module and endomorphism to it. The module is generated by grid states and the endomorphism is defined on the generators and extended to the full module. For a grid state x the boundary map is defined in terms of rectangles in the diagram. We say that a rectangle connects two grid states if those grid states differ by a transposition, or equivalently differ in two pairings, as seen in figure 1.11. We will also consider such rectangles as going from the lower left generator to the upper right.

We will refer to these rectangles as *empty* when the interior of the rectangle does not intersect any of the points of the grid states and the set of such empty rectangles from x to y will be denoted $\mathrm{rect}^\circ(x, y)$. Figure 1.11 shows an example of such an empty rectangle. Note

that each transposition represents two potential rectangles and that some of these rectangles are not empty, for instance the second rectangle in the above example is not empty as can be seen in figure 1.12.

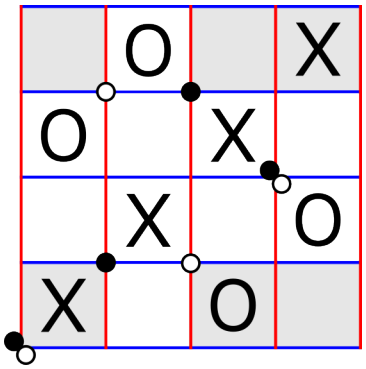


Figure 1.12 The states from figure 1.11 and a non-empty rectangle connecting them, note the basepoint at the lower left which is inside the rectangle wrapping around the torus

With rectangles and empty rectangles defined we can describe the differential. This map is dependent, like CFK, on the flavor but we will consider the most general infinity flavor. For a gridstate x the map ∂ is defined as:

$$\partial x = \sum_{y \in S_n} \sum_{r \in \mathrm{rect}^\circ(x,y)} \left(\prod_{i \in 1 \ldots n} U_i^{X_i(r)} \prod_{i \in 1 \ldots n} V_i^{O_i(r)} \right) y$$

More simply stated, each empty rectangle connecting x to y contributes a term to the differential with a coefficient for each X and O contained within it. When this is computed we can recover the various flavors by substituting values for various U_i and V_i, typically 1 or 0. In the case when we set $U_i = 1$ for all i we get the filtered grid complex described as $\mathcal{GC}^-$ in the following theorem.

Theorem 1.3.1 ([14]). *The quasi-isomorphism type of $\mathcal{GC}^-$ for a grid diagram $\mathcal{GC}^-(G)$ depends only on its associated unoriented knot K.*

CHAPTER 2

PROGRAM IMPLEMENTATION

2.1 Overview

A large portion of the work, and a product of, this dissertation is a program for computing the bigraded module CFK^-. As noted, the program is publicly available at https://github.com/CStClairMath/GridTools. The goal of this chapter is to show the validity of the algorithm and communicate the roles of the different component functions it uses. To aid in that, we will consider the different functions as well as a running example for context. The chapter itself will fall into two broad sections: the first to produce the module associated to a given grid, and the second to reduce the complex and convert flavors.

It is worth noting the program is written in Python and Sage, which can be thought of as an extension of Python. We will not be assuming any background in either, but at the same time we will not give a rigorous introduction to them. With that in mind some basics of the languages will inevitably be missed but with the hope of communicating exactly enough of the languages to understand the algorithm.

Additionally, in the following sections we may omit some code that is present for debugging or catching errors. The code without these omissions is available in the appendix as well as the github repository.

2.2 Generating the Module

Figure 2.1 is a roadmap of the program functions called in producing the complex. Functions separated by a vertical bar indicate that the function on the right is a supporting function of the one on the left.

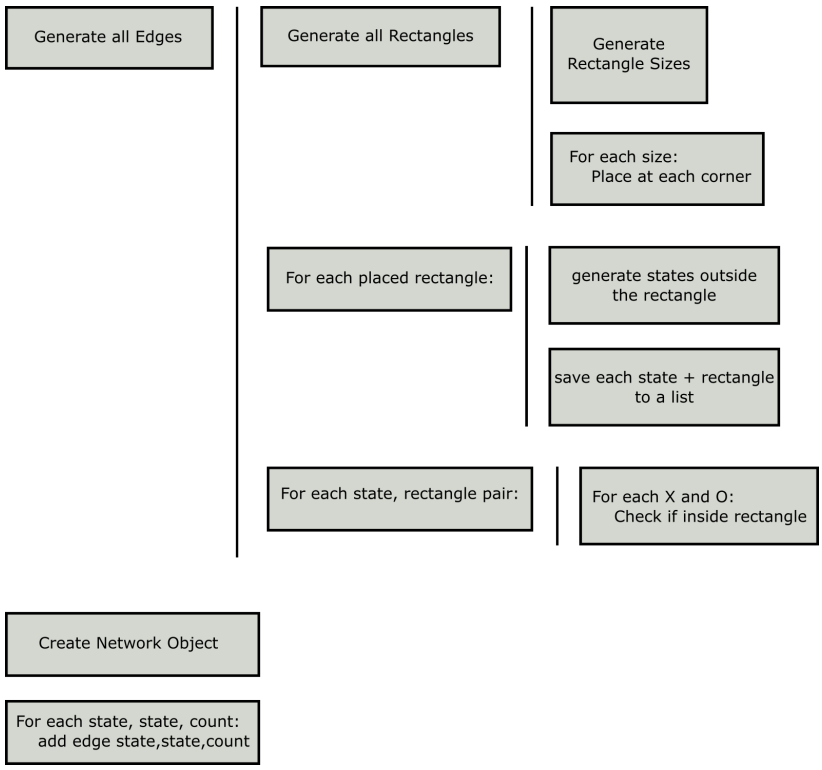


Figure 2.1 Outline of module generation

With roadmap in hand, we will start at the beginning of the module generation code. As with all modern programming languages, we will be borrowing from functions and data types made by others. These are brought into Python and Sage by import statements. These include the following import statements.

```
import networkx as nx #Package for handling mathematical graphs or networks
import numpy as np #General math package
import CodeModules.GFKTools as gfk #Part of the backend of the program
import copy
import math
```

These are not all of the import statements that are called between the different code modules, but these make up the functional pieces. Any remaining imports are reserved for debugging, and performance testing. The last two are basic Python packages. Copy allows for more control copying variables, and math adds to the default functions.

As the comments in green note, networkx and numpy are publicly available Python packages that we will be using. The third import statement is referencing a package stored locally in the program which we will see the details of in this chapter.

The phrasing import X as Y means when we want a function or object from a package X we can refer to it by a shorter name Y. So the following two lines are equivalent to Python for example:

```
MyGraph = networkx.DiGraph
MyGraph = nx.Digraph
```

Similar shorthands for numpy and GFKTools are added by the import lines.

2.3 Lemma-esque Code

The program is made up of a series of helper functions that break the problem into manageable pieces. Some of the functions need dissecting but, like lemmas, will be considered now, and applied later.

2.3.1 generate_all_rectangles

Part of the algorithm to construct the grid complex is to generate all connecting rectangles between grid states. In the approach given here, we will do this by considering all potential rectangles and fitting states to them, rather than the other way around. To that end we need to know all potential sizes of connecting rectangles we can encounter.

Lemma 2.3.1. *Valid connecting rectangles for a grid diagram of grid size n must have dimensions $l \times w$ such that $l + w \leq n$*

Proof. Suppose we have a rectangle r from some generator x to another generator y. For such a rectangle to be valid it must miss all the basepoints of x and y. Thus all the alpha curves intersecting the rectangle must have their basepoints placed above it.

This region is made up of $w - 2$ alpha circles with $n - l - 2$ beta circles which they can intersect. If a rectangle is valid then we were able to find a pairing of $w - 2$ alpha and beta circles in that region. So we know that $w - 2 \leq n - l - 2$ and therefore $w + l \leq n$.

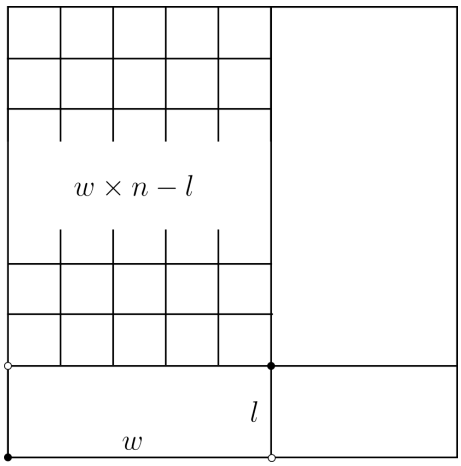


Figure 2.2 The region above a connecting rectangle with labeled dimensions

□

Following the argument backwards, for any rectangle of these sizes we can see that the region above the rectangle will have sufficient room to place basepoints among the intersections. Thus these are exactly all the rectangle sizes we will need to consider to generate the grid complex. We have a function to generate all these sizes named `generate_all_rectangle_sizes`. It works by considering all pairs of width and height that sum to less than n, and saving those dimensions as a list of tuples. Actual code below

```
def generate_all_rectangle_sizes(n):
#Input: n integer
#Output: All possible sizes of allowable rectangles
result = []
for width in range(1,n):
    for height in range(1,n+1-width):
        result.append((width,height))
return result
```

Then using this function we can build a function that finds every rectangle that will contribute to the differential.

```
def generate_all_rectangles(n):
#Input: n integer
#Output: All possible connecting rectangles for a grid of size n in a list format
sizes = generate_all_rectangle_sizes(n)
rects = []
```

```
    for size in sizes:
        for i in range(0,n):
            for j in range(0,n):
                x = (i % n + 1,j % n + 1)
                y = ((i+size[0]) % n + 1,(j+size[1]) % n + 1)
                rects.append((x,y))
    return rects
```

The first line of this code calls the function to generate all rectangle sizes, as was described previously, and saves them in a list named sizes. After this we create a list to hold the collection of rectangles we are about to generate.

The nested loops take every size, and iterates through all the locations for the lower left hand corner of the rectangle. Then it notes the location of the upper right corner of the rectangle according to the size. The modulo operation accounts for rectangles which wrap around the toroidal diagram. Then these pairs of lower left and upper right are added to the `rects` variable.

2.3.2 generate_all_states_outside_rectangle

Recall that the algorithm to compute GFC° in section 1.3 starts by considering each of the generators $x \in S_n$ and considers all rectangles that start at x and end at another generator y, while counting only such rectangles that do not contain points from the pair of states. We can avoid considering the invalid rectangles entirely by looking at all the proper rectangles that exist, and constructing all the states they could possibly connect.

By considering these we will have computed every element of the differential as well as generator of the complex.

Proposition 2.3.2. *Every grid state x has a rectangle that starts at it, and contributes to the unreduced complex*

Proof. Let $x \in S_n$ be a generator of the unreduced GFC°. Then there is a rectangle connecting x to $x \circ (1, 2)$ since the rectangle in the first column will be too narrow to contain a basepoint of either generator. □

Recall that since the diagram is on a torus, the rectangle mentioned in the above proof may wrap around vertically. This happens when $x(1) > x(2)$ as in the right hand side of figure 2.3

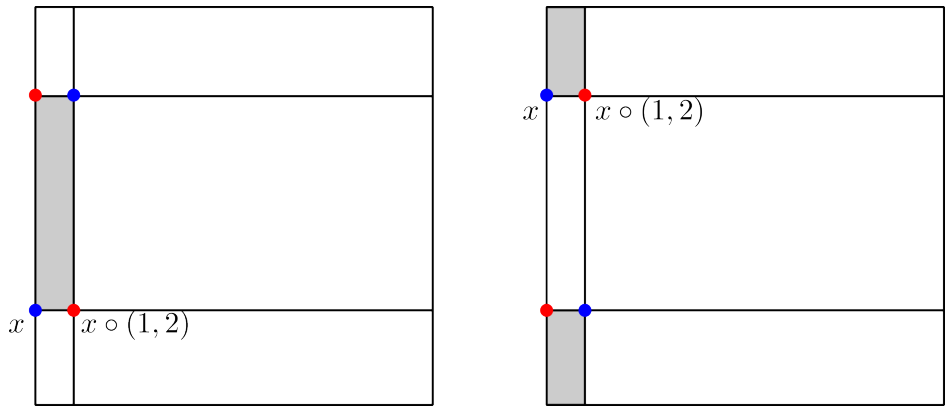


Figure 2.3 The possible vertical rectangles in the differential of every grid state mentioned in the proof of proposition 2.3.2.

Therefore, if we can find every contributing rectangle along with the states that have them as connecting rectangles we will have every component for the differential, and every generator of the complex. This is precisely the strategy we will employ to compute the complex. Recall that the code from section 2.3.1 will give us all possible rectangles, and the next function we will unpack will give us all the states that a given rectangle connects.

That function is appropriately named `generate_all_states_outside_rectangle`. It requires two inputs, a rectangle given by its lower left coordinate, and upper right along with the size of the grid diagram. We will generate the states as if the rectangle is placed in the lower left corner as in figure 2.4 and then shift the results afterwards.

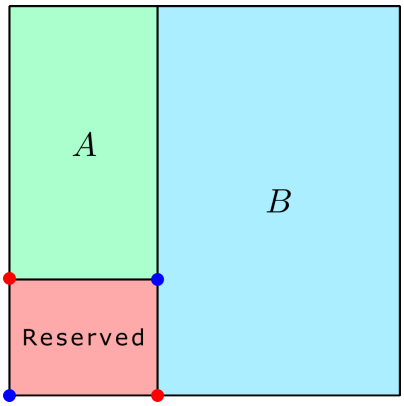


Figure 2.4 Regions in `generate_all_states_outside_rectangle`

```
def generate_all_states_outside_rectangle(rectangle, n):

    #Input: rectangle = ((a0, b0), (a1, b1)), n integer
    #
    #Output: List
    #
    #This function computes the states as if the rectangle had lower corner
    #(1,1) then shifts the result - keep in mind when seeing lists appending 1's

    #Compute the dimensions of the rectangle - mod n since we're working on a torus
    r_width = (rectangle[1][0] - rectangle[0][0]) % n
    r_height = (rectangle[1][1] - rectangle[0][1]) % n

    if (r_width + r_height) > n:

        print("rectangle dimensions too large to have non-empty set of states")
        return []

    pre_result = []
    .
    .
```

This first block of the function computes the size of the rectangle, % here is the mod operation. We store the dimensions into `r_width` and `r_height`. We also do a check to ensure that the dimensions are in line with Lemma 2.3.1.

We are going to generate the states we are searching for by placing the basepoints of the generators outside of the rectangle. We will have 3 rectangular regions available to place these basepoints. We will call them `reserved`, A and B as in figure 2.4.The next section of the code calls a function called `truncated_sn(n, trunc_length)` the code for this function can be found in the appendix B line 175. It returns all of the possible beginnings to an element of S_n of length `trunc_length`.

```
.
.
    #region a is the one above the rectangle and b is the columns after that
    #then s_a is the collection of intersection choices for a grid state x
    #that keeps the rectangle empty
```

```
    sa = truncated_sn(n - r_height - 1, r_width - 1)
    sb = generate_sn(n - abs(r_width) - 1)
.
.
```

We are calling these to give us alphabets for where to place basepoints in the rectangles A and B. We place the points in A first, then place those for B on the horizontal lines that remain. We will handle the case for when A is too narrow to place any points first.

```
.
.
    if sa == []:

        #whats left holds onto the symbols yet to be used
        whats_left = []
        for i in range(2,r_height+1):

            whats_left.append(i)

        for i in range(r_height+1,n+1):

            whats_left.append(i)

        for psi in sb:

            curr_state=[]
            curr_state.append(1)
            curr_state = [1+r_height] + curr_state

            for i in range(n-abs(r_width) - 1):

                curr_state.append(whats_left[psi[i] - 1])

            pcurr_state = perm(curr_state.copy())
            pre_result.append(pcurr_state)
 .
 .
```

The if statement signals that we are in the case when A is too narrow to admit any points. After this we have 2 for loops that record all the coordinates, or heights, that we

need to place in region B. They populate a list with everything except 1 and the height of the rectangle since those are fixed by the rectangle we are building states around.

Each state we build will be a list that starts out named `curr_state`. Since we are guaranteed here to have a rectangle of width 1 we know the first two coordinates of any state associated to this rectangle must start with $[1, r\_height, ...]$ so we start our state with those using the append function and adding to the list.

After this, we place the elements of `whats_left` according to each element of S_{n-2}. Note that the element of the list called is `[psi[i] - 1]`, this is because Python indexes lists starting from 0 and the permutations index from 1. The last two lines convert the state we produced to a permutation and saves it to our running list of states, `pre_result`.

The second case is where region A is large enough to admit points in the generators.

```
for sig in sa:

    #we're lifting this as a list rather than a permutation because
    #we need to shift them all up by the height - making it no longer a perm
    x = sig.copy()

    #loop lifts the symbols above the rectangle into region a
    for count, position in enumerate(x):

        x[count] = position + r_height + 1

    #whats left holds onto the symbols yet to be used
    whats_left = []

    for i in range(2,abs(r_height)+1):

        whats_left.append(i)

    for i in range(abs(r_height) + 2, n + 1):

        if i not in x:

            whats_left.append(i)
```

When A admits points, then either B does or does not. We catch the case when B is too

narrow next.

```
.
.
        if sb == []:

            curr_state = x.copy()
            curr_state.append(1)
            curr_state = [1+r_height] + curr_state
            pcurr_state = perm(curr_state.copy())
            pre_result.append(pcurr_state)
.
.
```

The only way that B can be too narrow to place points is if it is a single column. When this is the case, we just need to add in the end points of our rectangle which we do above.

Then when we have to consider points placed inside B as well we enter the following loop:

```
.
.
        for psi in sb:

            curr_state=x.copy()
            curr_state.append(1)
            curr_state = [1+r_height] + curr_state

            for i in range(n-abs(r_width) - 1):

                curr_state.append(whats_left[psi[i] - 1])

            pcurr_state = perm(curr_state.copy())
            pre_result.append(pcurr_state)
.
.
```

In this case we have placed points in A and now we need to map each element of S_{n-2-w} where w is the width of A, to the remaining horizontal arcs as in figure 2.5

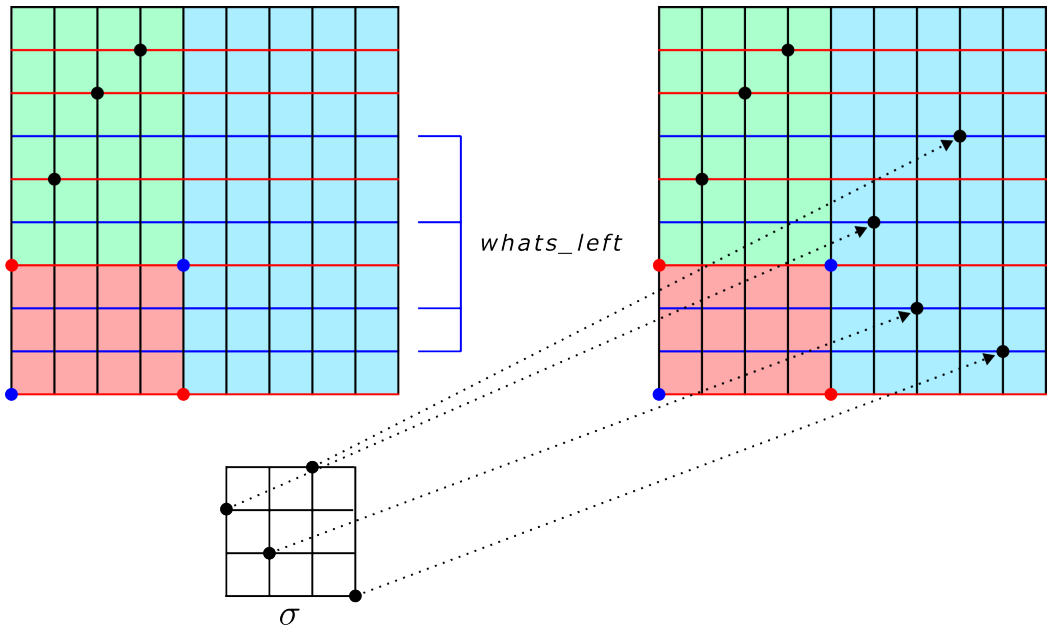


Figure 2.5 Mapping an element of an S_n to remaining rows

After this we have all of the states that correspond to a rectangle of the given size if it were positioned at the lower left corner of the diagram. The final step then is to pair off the source and targets of the rectangles and shift the results to where the rectangle is actually located.

```
    if not ((rectangle[0][0] == 1) and (rectangle[0][1] == 1)):

        raw_result = hv_set_shift(rectangle[0][0] - 1, rectangle[0][1] - 1, pre_result)

    else:

        raw_result = pre_result

    result = []

    rect_pairer = transposition(rectangle[0][0],rectangle[1][0],n)

    #Up to now the raw_results holds the states associated with base of the rectangle
    #this loop takes all those and pairs them with the connected state
    for sig in raw_result:
        result.append([sig*rect_pairer,sig])

    if result == []:

        return None
```

```
    return result
```

The function `hv_set_shift` moves each state horizontally and by composing the permutations of them with $(1, 2, 3...n)^h$ where h, representing the horizontal shift distance, is the vertical arc of the lower left corner of the rectangle, and then precomposing with $(1, 2, 3...n)^v$ where v is the coordinate of the horizontal arc. Figure 2.6 is a diagram showing the effects of the shift.

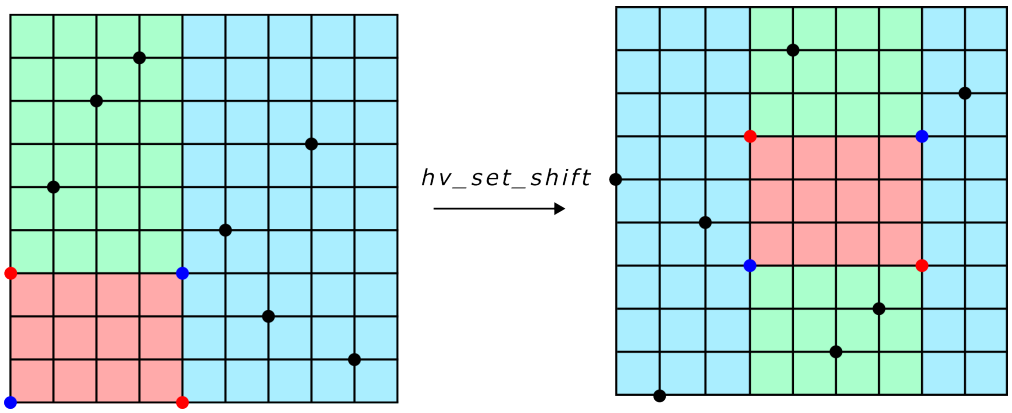


Figure 2.6 How `hv_set_shift` shifts the states to return a rectangle to its intended position. In this case shifting horizontally and vertically 3 steps

The final loop before returning the result is to append each state and the state it connects to as a list to the result variable. The pair is found by composing with the transposition defined by the x coordinates of the rectangle corners.

2.3.3 count_symbols

In generating the complex GFC we will need to check which Xs and Os are inside of a rectangle connecting two states x and y. The `count_symbols` function, and its component functions, handle this.

```
def count_symbols(n, rect, symbol_perm):

    #Input: n integer, rect rectangle ((a0, b0), (a1, b1)), symbol_perm a permutation
    ↪  or list
    #
    #Output: List with 1/0 for symbol in/out of connecting rectangle
```

```
    #
    #Iterates through each symbol symbol_perm(i) and marking a corresponding list
    ↪  entry
    #to 1 if the symbol is present

    temp = zero_list(n)
    usable_sym = symbol_coordinates(symbol_perm)
    #Moving the coordinates to the actual heights rather than the discrete style
    #lists

    for i in range(n):

        if parent_check(rect, usable_sym[i]):

            temp[i] = 1

    return temp
```

Not counting the comments, the core of this function is very short. First we make a container for the result called `temp`. Because the rectangle and symbol coordinates both begin indexing at 1, despite being staggered in the diagram, we need to remap the coordinates. The `symbol_coordinates` function takes a list or permutation representing σ_X or σ_O and returns a list of coordinates - for example `symbol_coordinates([3,2,1,4])` returns the list `[(1.5, 3.5), (2.5, 2.5), (3.5, 1.5), (4.5, 4.5)]`. The supporting code for this function can be found in section B on line 94.

After that is a loop that goes through each column and calls `parent_check` which we will unpack here.

```
def parent_check(rect, target):

    #Input: rect = ((ax, ay), (bx, by)) and target (tx, ty)
    #
    #Output: Returns True if target is inside the rectangle, False otherwise.

    ax = rect[0][0]
    ay = rect[0][1]
    bx = rect[1][0]
    by = rect[1][1]
```

```
    #This is a belabored switch statement. All of these are possible since grids exist
    ↪  on a
    #torus. There's two possibilities for the x coordinates and two for the y as to
    ↪  which one
    #comes first. This gives us the following 4 cases.
    if ( (ax < bx) and (ay < by) ):

        return check_case_{1}(rect, target)

    if ( (ax > bx) and (ay < by) ):

        return check_case_{2}(rect, target)

    if ( (ax < bx) and (ay > by) ):

        return check_case_{3}(rect, target)

    if ( (ax > bx) and (ay > by) ):

        return check_case_{4}(rect, target)

    print("invalid rectangle given")
```

This function's purpose is to sort out which type of rectangle is being checked in line with the cases shown in figure 2.7 and see if the provided coordinate is inside the rectangle. It decides which case is present by comparing the rectangle's provided corner x and y coordinates. Once the case is determined then it calls the corresponding `check_case_i`. These all function very similarly as can be seen in their definitions.

```
def check_case_{1}(rect, target):

    if (is_between(target[0], rect[0][0], rect[1][0]) and is_between(target[1],
    ↪  rect[0][1], rect[1][1])):
        return True
    return False


def check_case_{2}(rect, target):
```

```
    if ((not is_between(target[0], rect[1][0], rect[0][0])) and is_between(target[1],
    ↪  rect[0][1], rect[1][1])):
        return True
    return False


def check_case_{3}(rect, target):

    if (is_between(target[0], rect[0][0], rect[1][0]) and (not is_between(target[1],
    ↪  rect[1][1], rect[0][1]))):
        return True
    return False


def check_case_{4}(rect, target):

    if ((not is_between(target[0], rect[1][0], rect[0][0])) and (not
    ↪  is_between(target[1], rect[1][1], rect[0][1]))):
        return True
    return False
```

Each of these use a function `is_between(target, a, b)` which checks if `a` $<$ `target` $<$ `b` and returns a boolean accordingly. As an example consider `check_case_3`. When it receives `rect` this is a list containing the rectangles corners so `rect` $= ((x_0, y_0), (x_1, y_1))$. Then to access x_1 for example we would call `rect[1][0]`.

So in case 3 the function is checking if `target` has its x coordinate between the x coordinates of the rectangle corners, and that the y coordinate is *not* between the rectangles y coordinates.

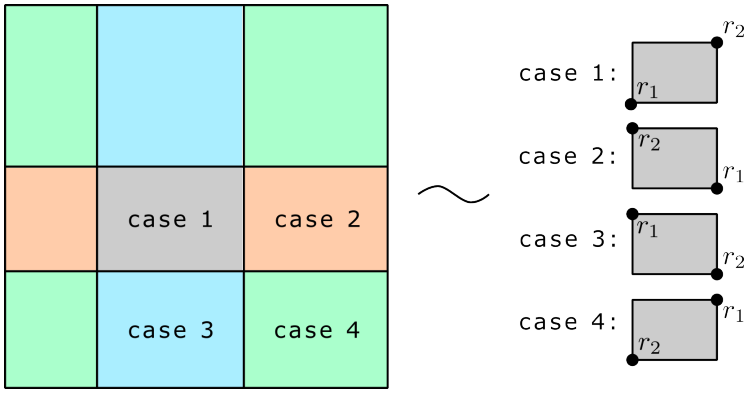


Figure 2.7 The regions that a rectangle's corners could be describing. Since rectangles here are stored by opposite corners they are indicated by (r_1, r_2)

Recall that we are calling this check function inside a loop to count symbols.

```
.
.
    for i in range(n):

        if parent_check(rect, usable_sym[i]):

            temp[i] = 1

    return temp
```

In this loop `i` corresponds to the columns and we can see then that we are recording a 1 in `temp[i]` when it does contain the symbol, or we leave it as 0 otherwise. Then finally we return the list of 0's and 1's. This data is accessed later, figure 2.9 shows 4 examples of what this function could return.

2.4 Calling build_cinf

The function called to start the whole process of building the complex is `build_cinf`. The example that we will accompany on its way through the program is the left-handed trefoil given by the permutations $\sigma_X = [5, 1, 2, 3, 4]$ and $\sigma_O = [2, 3, 4, 5, 1]$ as in figure 2.8.

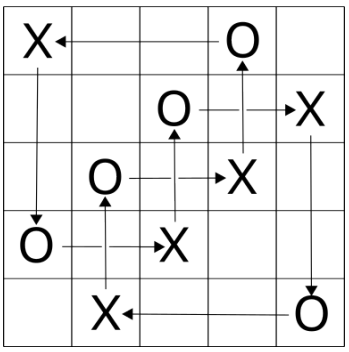


Figure 2.8 Grid diagram of left-handed trefoil $\sigma_X = [5, 1, 2, 3, 4]$ and $\sigma_O = [2, 3, 4, 5, 1]$

To hand this data off to the computer we will be giving it to Python as a list containing two other lists, namely the permutations for the X and O data.

```
trefoil = [[5, 1, 2, 3, 4], [2, 3, 4, 5, 1]]
```

Having the trefoil data stored in the so named variable, we can pass this variable to the first domino in a long series by calling the function `build_cinf` from the `GFKTools` package we recently imported. We will store the result afterward to `UnreducedComplex`.

```
UnreducedComplex = gfk.build_cinf(trefoil)
```

Unpacking this function we will look at `build_cinf`'s definition from the imported module.

```
def build_cinf(symbols):
    #Input: symbols a list of two lists, [sigx, sigo]
    #Output: g a networkx directed graph
    xlist = symbols[0]
    olist = symbols[1]
    size = len(xlist)

    comp = generate_all_edges(size, [xlist,olist])
    g = nx.DiGraph()

    for ele in comp:

        if not g.has_edge(str(ele[0][0]),str(ele[0][1])):
```

```
            g.add_edge(str(ele[0][0]),str(ele[0][1]), diffweight = [])

        g[str(ele[0][0])][str(ele[0][1])]['diffweight'].append((ele[2][0] +
        ↪  ele[2][1]))

    return g
```

The first lines of this function unpack and label the components of the grid given to the function. In the case of our trefoil, this loads $\sigma_X = [5, 1, 2, 3, 4]$ into the variable `xlist` and $\sigma_O = [2, 3, 4, 5, 1]$ into the variable `olist`. The next piece checks the grid size of the given diagram by checking the number of elements in `xlist`.

The following line `comp = generate_all_edges(size, [xlist,olist])` calls the function which generates all the differential maps in the grid complex.

```
def generate_all_edges(n, symbols):
#Input: n an integer, symbols, a collection of lists of length n denoting the
↪  placement of the symbols. For knots and links this should be a pair of two
↪  lists, or permutations.
#Output: Returns a list 3 layers deep containing all the edge information of
↪  CFKinf
pre_diff = generate_all_rectangles(n)
symbol_count = len(symbols) #This should always be 2 for knots and links.
place_holder = []
z_list = zero_list(n)
for i in range(symbol_count):
    place_holder.append(z_list.copy())
unweighted_diff = []
.
.
```

The first step is to generate all the rectangles following section 2.3.1. As per 2.3.2 every generator of the complex, and element of a differential will be caught by one of these rectangles. The second step is to produce a container for all the coefficient data. For X and O we will add n slots in a list; one for each symbol/column.

Next, for each rectangle we are going to call `generate_all_states_outside_rectangle` which will build a list of generators of our module that use the rectangle in the differential as in 2.3.2. To keep track of this data, we then pair it with the rectangle and a copy of our data container.

```
.
.
for rect in pre_diff:

    candidates = generate_all_states_outside_rectangle(rect, n)
    #each candidate is a collection of points differing on the rectangle given
    #so each candidate represents an edge from rect[0] = (a0, b0) to rect[1] = (a1,
    ↪  b1)
    #(filling in the rest of the necessary coordinates with the candidate)


    if candidates is not None:

        for count, candidate in enumerate(candidates):

            candidates[count] = [candidate.copy(), rect, place_holder.copy()]
            #^Replacing the candidate with a clean copy, a note of its connecting
            #rectangle and initialize the edge weight to zero

        unweighted_diff = unweighted_diff + candidates
        #appending the list of these candidates to our unweighted differential
.
.
```

Since that operation is done for each rectangle we add the results to a running list called `unweighted_diff`.

The final step is for X and O to iterate through the `unweighted_diff` elements and record which symbols are inside the rectangle using the `count_symbols` function explained in section 2.3.3,

```
.
.
for count, symbol in enumerate(symbols):
#This should again always be a pair in case of knots and links
```

```
        for i in range(len(unweighted_diff)):
        #here we replace the previous zero weight after counting the symbols

            unweighted_diff[i][2][count] = count_symbols(n, unweighted_diff[i][1],
            ↪  symbol)

    return unweighted_diff
```

This resulting output contains all the information of the module but we are going to take several steps and repackage it into a graph. Continuing along `build_cinf`, we will build the module as a directed graph. The first step is to initialize a directed graph object using networkx. Then we will loop through each edge and add it to a directed graph.

```
.
.
    comp = generate_all_edges(size, [xlist,olist])
    g = nx.DiGraph()

    for ele in comp:

        if not g.has_edge(str(ele[0][0]),str(ele[0][1])):

            g.add_edge(str(ele[0][0]),str(ele[0][1]), diffweight = [])

        g[str(ele[0][0])][str(ele[0][1])]['diffweight'].append((ele[2][0] +
        ↪  ele[2][1]))

    return g
```

It is worth noting that if the edge is already in the graph that the function stores both edge values. This happens when there are two valid rectangles connecting the states x and y. At this point we have computed and stored all the information of the complex in a graph. From here our goal is to make the data more useful and recognizable.

2.5 Initializing the Module

Up to this point all of the computations have been done using Python. Now because we want to handle polynomials over $\mathbb{F}_2$ we will be using Sage. Sage is an extension of Python so the syntax will be nearly identical but the code is broken into a second part to run with the Sage program. The first portion of the code is kept as separate Python file for maximum modularity.

As with the first portion we need to import some modules, in this case some of them will be our own.

```
import itertools as itools
import networkx as nx
import CodeModules.GFKTools as gfk
from CodeModules.GridPermutations import *
import CodeModules.perm as pr
import multiprocessing as mp
from multiprocessing.managers import BaseManager
import random as rd
```

Before we can convert the edge weights of the graph to polynomials we need to define the specific Laurent polynomial ring for Sage. This is handled by our function `cinf_coeff(n)`.

```
def cinf_coeff(size):

    # Takes size as an argument and returns the Laurent polynomial ring over Z2 with
    ↪  coefficients U0,...Usize-1,V0,...Vsize-1

    n = size
    varis = name_some_vars(['U','V'],n)
    F = LaurentPolynomialRing(GF(2), varis)
    F.inject_variables()

    return F,list(F.gens())
```

We name the necessary variables, one for each X and O, using `name_some_vars`. The result will look like `["U0", "U1", ... "Un-1", "V0", ... "Vn-1"]`. This is saved to `varis`. We then pass all of this to Sage's built-in function `LaurentPolynomialRing` along with `GF(2)` $= \mathbb{F}_2$ which defines $\mathbb{F}_2[U_i, U_i^{-1}, V_i, V_i^{-1}]_{i=0\ldots n}$. The next line's `inject_variables` declares each of those variable names as objects we can reference later, so if we execute `U1+U2` Sage will recognize it as an element of the polynomial ring and not a python variable name. Then the function returns the ring and the $2n$ variables we declared.

The function that reassembles the graph edges from section 2.2 is `construct_cinf`. The first couple lines extract the size of the grid if it is not provided explicitly. After that we generate the Laurent polynomial for the module and save the associated variables to `F` and `Vars`.

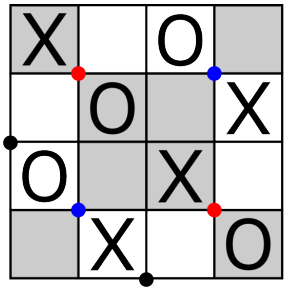
```
def construct_cinf(g, sigx, sigo, size = -1):

    if size == -1:
        size = len(g.get_edge_data(list(g.edges())[0][0],
        ↪  list(g.edges())[0][1])['diffweight'][0])
        n = size/2

    else:
        n = size

    F,Vars = cinf_coeff(n)
    resG = nx.DiGraph()
.
.
```

We initialize a variable `resG` for the resulting graph we are converting to. To do this we are going to loop over all of the edges of `g` which will be the resulting graph from generating the module in section 2.4.

```
.
.
    for edge in g.edges:

        start = edge[0]
```

```
        end = edge[1]
        poly = F(0)

        for subweight in g[edge[0]][edge[1]]['diffweight']:

            i = 0
            polychange = F(1)

            for entry in subweight:

                polychange = polychange*(Vars[i])**entry
                i = i + 1

            poly += polychange

        resG.add_edge(start,end,diffweight = poly)

    return grid_complex(resG, F, sigx, sigo)
```

In each step of this loop we will be accessing the edge data that we stored previously. The format of edge data from 2.4 is a list of lists. The structure of that data is `diffweight[rectangle][X column, O column]`. For each rectangle (at most 2) we will have a polynomial term that will summed. As an idea for the data present see figure 2.9, it illustrates a case where a pair of states are connected by two distinct rectangles. The data stored in the edge data for this example will be in `g[3,2,1,4][3,4,1,2]['diffweight']` and is of the form `[[0,0,1,0,0,1,0,0],[1,0,0,0,0,0,0,1]]`.



X data rectangle 1:	0	0	1	0
O data rectangle 1:	0	1	0	0
X data rectangle 2:	1	0	0	0
O data rectangle 2:	0	0	0	1

Figure 2.9 Rectangles connecting two states and the edge data recorded from symbols as lists

In the loop then we initialize the resulting edgeweight as 0 and in our module base ring by calling `poly = F(0)`. Then for each rectangle we set a `polychange` variable to 1 and iterate through the entries multiplying by `Var[i]`^{k} where $k \in 0, 1$ depending on list data. This is exactly the differential described in section 1.3. We add these to `poly` and save the result to our graph as the edge weight connecting the same vertices that we were given. The final result is not however returned as a graph, it is returned as a `grid_complex` object.

2.6 Class Definition for grid_complex

The `construct_cinf` function of the previous section returns a `grid_complex` object. The idea being that our functions are going to reference lots of internal data of the graph we are constructing as well as auxiliary data like the base ring, σ_X and σ_O and some other running data. Packaging these as one custom object will allow us to write functions that can much more conveniently access this data.

We define classes in Python and Sage very similarly to how we define functions. We define it by using `class` then defining a function that always runs and any supporting functions.

```
class grid_complex:

    def __init__(self, ... )
    .
    .

    def supporting_functions
    .
    .
```

In the rest of this thesis some of the functions we unpack will be supporting functions of this `grid_complex` object and some will not. We will specify when this is the case, and additionally whenever we call such functions it will appear as `example_complex.supporting_function` with the period indicating that the function is a child function of that specific object.

With that in mind, we will unpack the `__init__` function that is always called when we create a `grid_complex` object.

```
class grid_complex:

    def __init__(self, directed_graph, rng, sigx = None, sigo = None):

        self.comp = directed_graph
        self.ring = rng
        self.min_gradings = {}
        self.max_gradings = {}
        self.max_grading_changes = {}
        self.sigx = sigx
        self.sigo = sigo
        self.set_to_minus = False
        self.set_to_tilde = False
        .
        .
```

Each of these `self.xyz` lines is setting internal variables that we can reference at any time. The first saves the directed graph associated to our complex. We also save the ring for the module, and defining permutations. The final two above set boolean flags that we will flip when the appropriate functions have been called. For instance, when computing the surgered complex it is necessary to convert to the minus complex and this variable lets us know if this has already been done.

The variables saved as {} are dictionaries where we can save data much like how lists function but instead of being indexed by integers the indexing is more arbitrary. For example the `min_gradings` variable we will use to save the minimum values of the various gradings and the other dictionaries are similar. We will return to these as they come up.

The final section of the `__init__` function handles setting up some variables for the surgered complex.

```
        .
        .
        # From here the values necessary for the surgered manifold gradings are mapped
        ↪  out
```

```
        if (sigx != None) and (sigo != None):
            self.size = len(sigx)
            self.components = link_components(sigx, sigo)
            for i in range(len(self.components)):

                key = f'AGrading{i}'
                self.min_gradings[key] = 0
                self.max_gradings[key] = 0
                self.max_grading_changes[key] = 0
                key = f'UGrading{i}'
                self.min_gradings[key] = 0
                self.max_gradings[key] = 0
                self.max_grading_changes[key] = 0
                key = f'VGrading{i}'
                self.min_gradings[key] = 0
                self.max_gradings[key] = 0
                self.max_grading_changes[key] = 0
        else:

        # This is included in case the methods in the class are useful to another
        ↪  complex being loaded in

            self.components = None
```

What this code does (assuming the defining permutations are provided) is creates variables for each component in each of our dictionaries. So for instance, in the case of the Hopf link, this block will find that the link has 2 components. It will then make keys `'AGrading0'` and `'AGrading1'`, and for each dictionary it will save a value of 0 to each of those keys. So it will save that the minimum Alexander grading for the first component is 0, as well as the maximum and the largest change between vertices. It does the same for the Maslov gradings.

This is almost certainly not going to be what these values will end up being. But, we know that they are *at least (most)* these values since we know from section 1.3 that X^{SW} and O^{SW} have these values and we will compare against these values and adjust accordingly later.

2.7 Grading the Complex

Having called `construct_cinf` we have our complex with edgeweights represented as polynomials. From 1.3 recall that this is a multi filtered complex but the filtration data of our object as it stands is not clear. Our plan to fix that is going to be to set the elements that are in grading 0 for the various gradings. We know these are x^{NWO} and x^{NWX} for the Maslov gradings.Then, when those are set, we can step through a spanning tree computing relative gradings to find the values for the rest of the generators. Figure 2.10 visualizes what we are trying to find.

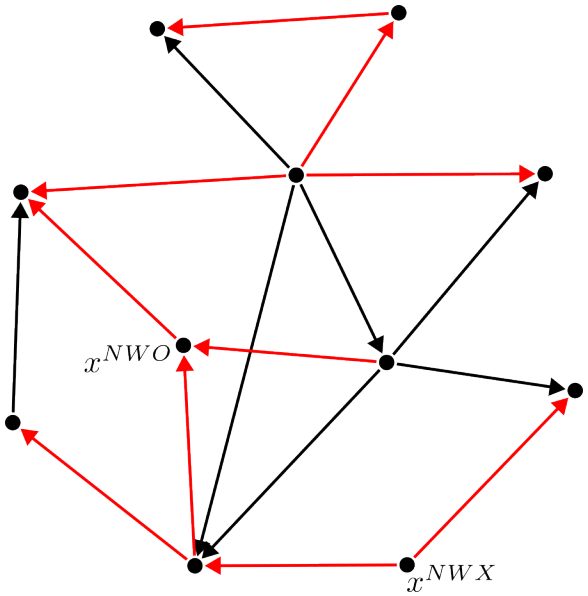


Figure 2.10 Schematic example of a spanning tree highlighted with x^{NWO} and x^{NWX} indicated. Note this is illustrative but is *not* from a real grid complex

When these Maslov gradings are all computed, we can then compute the Alexander grading by $A(x) = \frac{1}{2}(M_U(x) - M_V(x)) - (\frac{n-1}{2})$. This is only technically true for knots, the case of links is more delicate. For links, we will keep track of the Maslov grading but also track what we will dub *virtual Maslov gradings*. This will be exactly the Maslov grading for knots, but in the case of links we will keep track of a Maslov grading for each component rather than the entire link.

That is to say, when we compute relative virtual gradings, we will use $\mathcal{V}M_{U_i}(x) - \mathcal{V}M_{U_i}(y) = 1 - 2\sum_{j\in C_i} \#(r \bigcap X_j)$ where C_i indicates the set of indices that correspond

to the ith component of the link. We make similar virtual gradings for the O_i and V_i. Then recalling the equation $A_i(x) - A_i(y) = \sum_{j \in C_i} \#(r \cap X_j) - \sum_{j \in C_i} \#(r \cap O_j)$ up to an added constant, we can see the following:

$$\begin{array}{rcc} \mathcal{V}M_{V_i}(y) - \mathcal{V}M_{V_i}(x) & = & 1 - 2\sum_{j \in C_i} \#(r \bigcap O_j) \\ - \quad \mathcal{V}M_{U_i}(y) - \mathcal{V}M_{U_i}(x) & = & 1 - 2\sum_{j \in C_i} \#(r \bigcap X_j) \\ \hline (\mathcal{V}M_{U_i}(x) - \mathcal{V}M_{V_i}(x)) - (\mathcal{V}M_{U_i}(y) - \mathcal{V}M_{V_i}(y)) & = & 2\sum_{j \in C_i} (\#(r \bigcap X_j) - \#(r \bigcap O_j)) \end{array}$$

Where the right side is exactly twice the ith Alexander grading and the left side is a function of x minus a function of y. Therefore $A_i = \frac{1}{2}(\mathcal{V}M_{U_i}(x) - \mathcal{V}M_{V_i}(x))$, up to that same constant which, following [14], is $\frac{n_i - 1}{2}$

With the strategy in mind, we will start unpacking the code. The function we call to grade the complex is a method of `grid_complex` and is `grid_complex.grade_link_complex`.

```
    def grade_link_complex(self):

        if self.sigx == None:

            gridX = list(self.comp.nodes())[0]

        else:

            gridX = self.sigx
            cycle = pr.full_cycle(self.size)
            gridX = pr.perm(gridX)
            gridX = cycle*gridX
            gridX = gridX.value

        if self.sigo == None:

            gridO = list(self.comp.nodes())[0]

        else:

            gridO = self.sigo
            cycle = pr.full_cycle(self.size)
            gridO = pr.perm(gridO)
            gridO = cycle*gridO
```

```
            gridO = gridO.value
        .
        .
```

This first block of code finds the states x^{NWX} and x^{NWO}. There is additional code for the sake of modularity that picks the base states arbitrarily if the object was made somehow without X and O permutations. There is still some more setup before grading can begin.

```
        gens = self.ring.gens()
        size = len(gens)/2

        comp_set = len(self.components)

        for i in range(comp_set):
            nx.set_node_attributes(self.comp, False, f"HasBeenGraded{i}")
```

We will need to do loops in this algorithm over each component, so we save how many components we have into the `comp_set` variable. After this we set a variable on each vertex called `HasBeenGradedi` to False where there is an `i` for each component. The next step is to find a spanning tree in our graph if there is one.

Lemma 2.7.1. *The unreduced grid complex is connected as a graph.*

This follows from the fact that we know each grid state connects to each state differing by adjacent transpositions as we saw in the proof of Proposition 2.3.2, and that S_n is generated by adjacent transpositions.

So knowing that we can always find a spanning tree we will outsource the job of finding one to NetworkX, the package we have been using to handle our graphs.

```
        tree = nx.algorithms.minimum_spanning_tree( self.comp.to_undirected()  )
        eds = set(tree.edges())  # optimization
        spanset = []

        for edge in eds:
```

```
            if edge in self.comp.edges():
                spanset.append(edge)

            else:
                spanset.append((edge[1],edge[0]))

        span = self.comp.edge_subgraph(spanset)
```

As a note we technically will not be finding a directed tree since we do not mind which direction an edge is pointed. We are able to find the relative grading we want regardless of edge direction.

The spanning tree algorithm in NetworkX (at the time of writing) is only implemented for undirected graphs. So this code converts to an undirected graph and finds the tree in the first line. After that it saves the edges of the tree and pieces the directed version back together as a directed subgraph in the following loop. The if-else statement just checks to see which direction the edge should have and adds it to a running list `spanset`. Finally, we get the subgraph generated by the edges we saved and this is our spanning subgraph that we save to `span`.

The actual grading is passed off to helper functions which we do next.

```
        .
        .
        self.componentwise_relative_grading_loop("UGrading", gridX,
        ↪  self.virtual_U_gradings_succ, self.virtual_U_gradings_pred, span,
        ↪  comp_count)
        self.componentwise_relative_grading_loop("VGrading", gridO,
        ↪  self.virtual_V_gradings_succ, self.virtual_V_gradings_pred, span,
        ↪  comp_count)
        self.relative_grading_loop("UGrading", gridX, self.maslov_U_succ,
        ↪  self.maslov_U_pred, span, comp_count)
        self.relative_grading_loop("VGrading", gridO, self.maslov_V_succ,
        ↪  self.maslov_V_pred, span, comp_count)
        .
        .
```

We will unpack `relative_grading_loop` and `componentwise_relative_grading_loop` in a moment, but after those gradings are found there is a final loop in the function to find the Alexander gradings.

```
        .
        .
        for vert in self.comp.nodes():
            self.comp.nodes()[vert]['AGrading'] = 0
            for i in range(len(self.components)):
                stab_count = len(self.components[i])
                self.comp.nodes()[vert][f'AGrading{i}'] =
                ↪  (1/2)*(self.comp.nodes()[vert][f'VGrading{i}'] -
                ↪  self.comp.nodes()[vert][f'UGrading{i}']) - (1/2)*(stab_count - 1)
                self.comp.nodes()[vert]['AGrading'] +=
                ↪  self.comp.nodes()[vert][f'AGrading{i}']

        return
```

In this loop we compute the ith Alexander grading using our virtual Maslov gradings, indicated by `UGradingi` and `VGradingi`. We also need the constant offset for the grading which we find by $\frac{1}{2}(\texttt{stab\_count} - 1)$ where `stab_count` is the number of symbols in the ith component. We add each of these multigradings together to find the total Alexander grading.

2.7.1 relative_grading_loop

We need to compute two gradings per component, so we will be finding $2i+2$ relative gradings across our graph each with unique functions. We will build a structural function to hold these. We have `relative_grading_loop` and `componentewise_relative_grading_loop` for the virtual gradings. They are structured very similarly so we will unpack `relative_grading_loop` first and then look at where the componentwise variant differs.

```
    def relative_grading_loop(self, grading_key, base_vertex, fn1, fn2, span = None,
    ↪  grading_multiplicity = 1):

        # Loop structure around a vertex's neighbors to set gradings based on the
        ↪  functions fn1 and fn2.
```

```
        if span == None:

            span = self.comp


        nx.set_node_attributes(self.comp, False, "HasBeenGraded")
        self.comp.nodes()[str(base_vertex)][f'HasBeenGraded'] = True
        self.comp.nodes()[str(base_vertex)][f'{grading_key}'] = 0
        .
        .
```

The first step checks to see if the loop was provided a spanning tree; if not, it is going to work over the entire complex. The algorithm we implement will work in this case, but in our applications this should not happen. Then we initialize our starting variables and base vertex.

We set a flag for each vertex named `HasBeenGraded` to False. For whichever grading we are computing we switch that flag to True and set the grading to 0 for `base_vertex` which we have to provide to the function when we call it along with the name of what grading we are computing.

```
        on_deck = [str(base_vertex)]

        in_the_hole =[]

        while len(on_deck) > 0:

            for vert in on_deck:

                for i, component_columns in enumerate(self.components):
                    for succ in span.successors(vert):

                        if self.comp.nodes()[succ][f'HasBeenGraded{i}']: continue

                        in_the_hole.append(succ)

                        fn1(i, succ, vert, component_columns)
```

```
                self.comp.nodes()[succ][f'HasBeenGraded{i}'] = True
        .
        .
```

The terminology of the variables and loop here follows sports terminology for taking turns. The vertex being used to grade its neighbors is "at bat" then the vertices that are about to be used the same way are "on deck". Finally the vertices that will follow the on deck ones are "in the hole".

The loop is depicted in 2.11. The first two horizontal arrows grade and reserve vertices for the next step in the loop, saving the vertices but are adjacent to `on_deck`. The diagonal arrows represent flushing `on_deck` and moving `in_the_hole` to the newly emptied `on_deck`. The final dotted arrow just means to iterate until all vertices are graded. The naming scheme is a bit of a misnomer since the "batter" remains in "in the hole" until the rest of the waiting batters have been processed.

Only half of the code for the loop is shown above. In particular it is the code for computing the grading of a vertex that is the target of the active vertex. This can be seen since the outermost loop is over `span.successors(vert)`. It is a good time to note that if the active vertex is a successor or a predecessor we will have to use a different function to assign gradings. This just comes down to whether the active vertex is x or y in the relative grading equations

$$M_U(x) - M_U(y) = 1 - 2\#(r \bigcap X)$$

$$\mathcal{V}M_{U_i}(x) - \mathcal{V}M_{U_i}(y) = 1 - 2\#(r \bigcap X_i)$$

The vertices that are predecessors of the active vertex are handled in the following loop.

```
        .
        .
        for pred in span.predecessors(vert):

            if self.comp.nodes()[pred][f'HasBeenGraded{i}']: continue
```

```
                    in_the_hole.append(pred)

                    fn2(i, pred, vert, component_columns)

                    self.comp.nodes()[pred][f'HasBeenGraded{i}'] = True

            on_deck = in_the_hole
            in_the_hole = []

        return
```

We will consider an example of calling this function, to grade the Maslov U grading for example. To call it we will assume we already created a `grid_complex` type object called `comp=grid_complex(sigx, sigo)`. Additionally assume we found a spanning tree named `tree`. Then the function call would look like the following.

```
comp.relative_grading_loop("UGrading", sigx, comp.maslov_U_succ, comp.maslov_U_pred,
↪  span)
```

So we are passing the function the necessary supplemental functions to compute successor gradings and predecessor gradings as `fn1` and `fn2`.

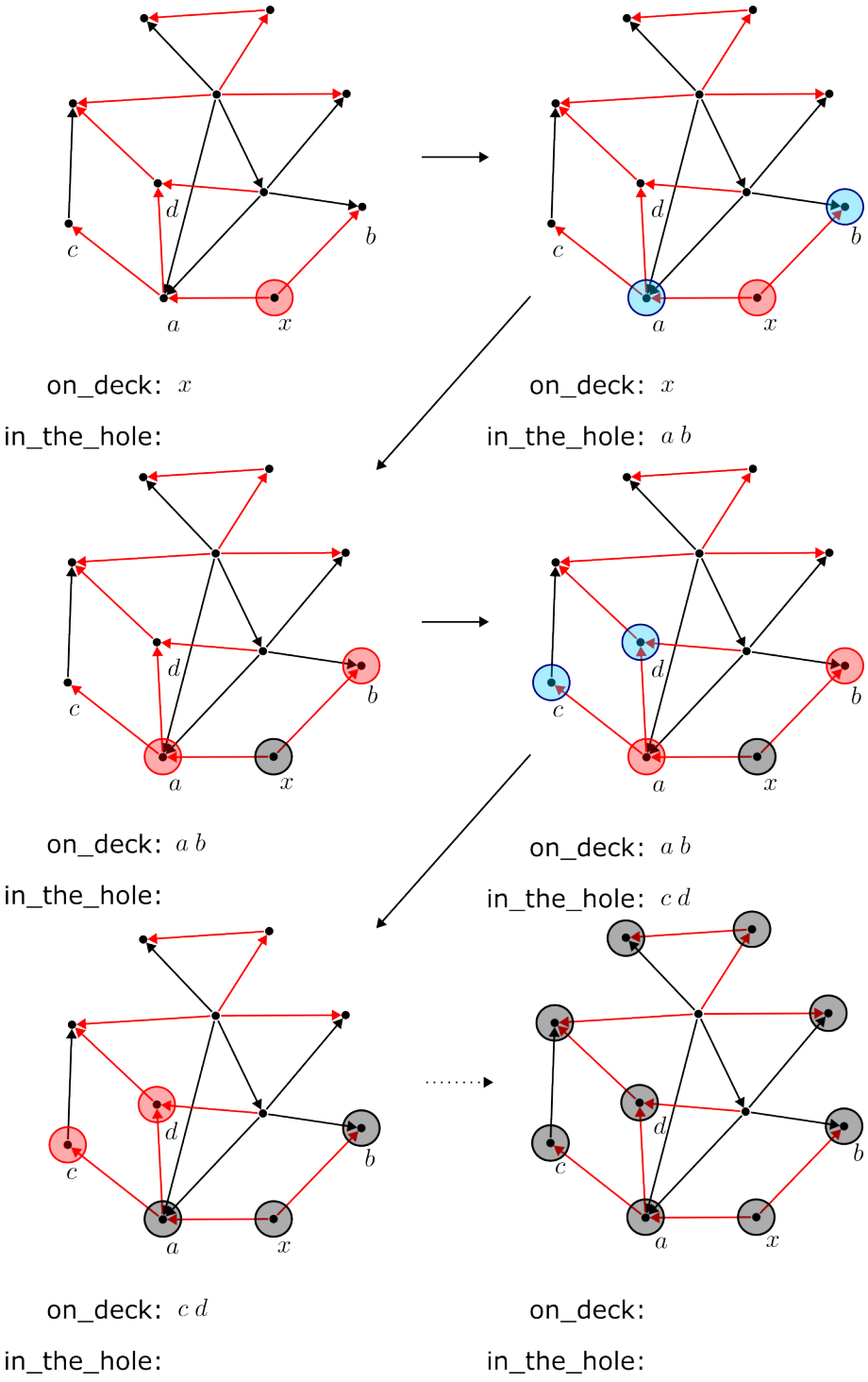


Figure 2.11 Visualization of the grading loop functions.

Before we cover the predecessor and successor grading functions in detail, we will look at the differences in the componentwise version of the grading loop. The first difference is in the setup portion. We will be grading each vertex once for each component of the link so

we will set flags for each of these.

```
.
.
for i in range(grading_multiplicity):

    nx.set_node_attributes(self.comp, False, f"HasBeenGraded{i}")
    self.comp.nodes()[str(base_vertex)][f'HasBeenGraded{i}'] = True
    self.comp.nodes()[str(base_vertex)][f'{grading_key}{i}'] = 0
.
.
```

Then, the only other difference is our actual loop structure. It is one loop deeper so that we can iterate over the components, and the predecessor and successor functions will also use this loop information. We will only look at the successor loop here, the predecessor is similar.

```
            .
            .
            for i, component_columns in enumerate(self.components):
                for succ in span.successors(vert):

                    if self.comp.nodes()[succ][f'HasBeenGraded{i}']: continue

                    in_the_hole.append(succ)

                    fn1(i, succ, vert, component_columns)

                    self.comp.nodes()[succ][f'HasBeenGraded{i}'] = True
            .
            .
```

At this point we have inspected each part of the grading portion of the program except for the innermost grading functions. There are 8 total such functions as combinations of: successor or predecessor, U or V, and componentwise or non-componentwise.

Since they all are very similar we will unpack `maslov_U_pred` and `virtual_V_gradings_succ` to catch each flavor. All the functions are in section C starting on line 707.

```
    def maslov_U_pred(self, pred, vert):

        ed_weight = self.comp[pred][vert]['diffweight']

        component_columns = self.sigx

        Upows = link_U_deg(ed_weight, self.ring, component_columns)
        self.comp.nodes()[pred]['UGrading'] = self.comp.nodes()[vert]['UGrading'] + 1
        ↪  - 2*Upows

        return
```

The function is relatively lightweight. We extract the edgeweight from the vertex, and then we take a sum of all the U_i powers in the polynomial edgeweight. We have not unpacked `link_U_deg` but it is also available in the appendix. The code looks at the first term of a polynomial and adds up the powers of the U variables.

Once this is found and saved to `Upows` we rearrange $M_U(x) - M_U(y) = 1 - 2\#(r \bigcap X)$ to solve for our desired variable. In our case we are setting the value of x which is `pred` using the value of the active vertex. The other function we will look at behaves very similarly.

```
    def virtual_V_gradings_succ(self, i, succ, vert, component_columns):

        ed_weight = self.comp[vert][succ]['diffweight']

        Vpows = link_V_deg(ed_weight, self.ring, component_columns)
        self.comp.nodes()[succ][f'VGrading{i}'] =
        ↪  self.comp.nodes()[vert][f'VGrading{i}'] - 1 + 2*Vpows

        return
```

In this case, the function accepts an argument `component_columns` which indicates the columns that the relevent symbols/variables are associated with. Passing that info along to `link_V_deg` indicates that it will only sum the V_i that come from those columns.

2.8 Reducing the Complex

With the complex generated at this point it will have $n!$ generators and a cumbersome number of edges. The first step will be simplifying the complex from here using a common reduction algorithm. It works by producing a complex with two fewer generators that is chain homotopy equivalent to the original.

The reduction can be thought of graphically as deleting an edge xy in the complex, and extending any of the edges that previously ended at y to ∂x, as in figure 2.12.

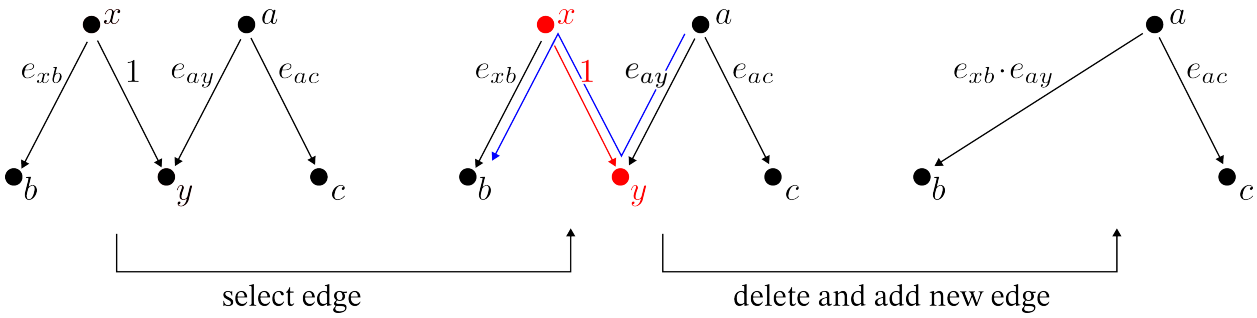


Figure 2.12 Steps in the reduction algorithm

This algorithm is known broadly in the literature and is written down in [6] Lemma 4.1. The proof and standard version of the reduction is for chain complexes but our case is a curved complex as noted in section 1.3. So we will largely mimic their proof but for curved complexes.

Proposition 2.8.1. *Given a curved complex (C, d_M) a freely generated module with potential $d_M^2 = M$, and $d_M(x, y) = 1$ where $d_M(a, b)$ is the coefficient of b in $d_M(a)$, then the complex defined by the reduction above is a curved complex (C', d_N) which is curved homotopy equivalent to C*

Proof. We define a homomorphism $h : C \to C$ by:

$$h(v) = \begin{cases} x & v = y \\ 0 & else \end{cases}$$

We also define $\pi : C \to C'$ and $\iota : C' \to C$ to be the projection and inclusion maps respectively.

Then the boundary map of C' can be described equivalently as $d_N = \pi \circ (d_M - d_M h d_M) \circ \iota$ and $d_N^2(v)$ then can be simplified as follows:

$$\begin{aligned} d_N^2 &= (\pi \circ (d_M - d_M h d_M) \circ \iota)^2 \\ &= \pi \circ (d_M - d_M h d_M)^2 \circ \iota \\ &= \pi \circ (d_M^2 - d_M^2 h d_M - d_M h d_M^2 + d_M h d_M^2 h d_M) \circ \iota \\ &= \pi \circ (M - M(h d_M + d_M h) + M d_M h^2 d_M) \circ \iota \\ &= \pi \circ (M - M(h d_M + d_M h) + M d_M h^2 d_M) \circ \iota \\ &= M \cdot \mathbb{I} \end{aligned}$$

The final simplification follows because $h^2 = 0$, and that $(h d_M + d_M h)(v) = 0$ for $v \in C'$. We can see this, as when we distribute through the second term is 0 and the first term is 0 or x which we lose during projection. Therefore (C', d_N) is a curved complex.

Then defining $f : C \to C'$ as $f = \pi \circ (\mathbb{I} - d_M \circ h)$ and $g : (\mathbb{I} - h \circ d_M) \circ \iota$ we get right away that $f \circ g = \mathbb{I}$. Then, because homotopies of maps for curved complexes are defined nearly identically as for chain complexes we only need to verify $g \circ f - \mathbb{I} = h d_M + d_N h$. Thus we obtain as in [6] that $g \circ f \sim \mathbb{I}$.

□

2.8.1 Reduction Algorithm

The function we call to search the complex for viable edges to reduce using the algorithm is named `graph_red_search`. It will call a helper function to actually do the reduction described which we will unpack afterwards.

```
def graph_red_search(self, started = False, timerstart = None):

    while True:

        try:
            red_target = next((source, target) for source, target, weight in
            ↪  self.comp.edges(data = 'diffweight') if weight == 1)
```

```
                self.graph_reduction(red_target[0], red_target[1])
                continue

            except:
                ("StopIteration")

            break
        return
```

The code here finds the first edge it can where the edge weight is 1, which is the required condition for our algorithm. It does this using the `next` function. From there it calls the helper function, passing it that edge it found and then continues the loop.

If there are no edges with weight 1 then the `next` function will throw a `StopIteration` error. If it gives this specific error we know we have run out of those reducible edges and the function ceases. From here we will go back and look at the helper function `grid_complex.graph_reduction`.

```
    def graph_reduction(self, key, target):

        for x in self.comp.predecessors(target):

            if x == key: continue
            for y in self.comp.successors(key):

                if y == target: continue
                x_weight = self.comp[x][target]['diffweight']
                y_weight = self.comp[key][y]['diffweight']
                red_weight = x_weight * y_weight
                if self.comp.has_edge(x,y):
                    old_weight = self.comp[x][y]['diffweight']
                    red_weight = red_weight + old_weight
                self.comp.add_edge(x,y,diffweight=red_weight)

        self.comp.remove_node(key)
        self.comp.remove_node(target)
        return
```

The loop is nested so that for each vertex that comes into the target we consider its

combination with each vertex the source edge goes to. See figure 2.13 to see these sets highlighted. Then for each of these combinations we save the weights associated and take their product. Then we add an edge between the pair we are considering.

It can happen that an edge is already present, if this is the case then we add the current weight and the weight associated with the reduction algorithm.

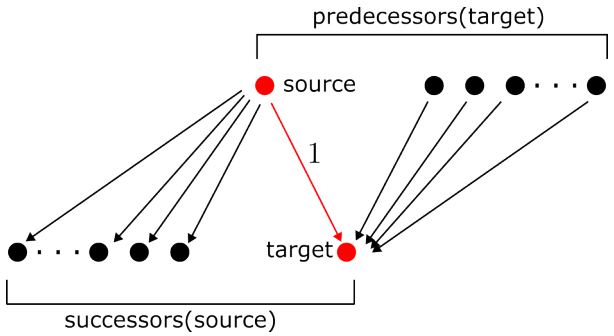


Figure 2.13 The vertices that constitute the two loops in `graph_reduction`

2.8.2 Parallelization Ingredients

In observing the program running the reduction step is the first bottleneck we run into. We will not solve this completely here but we will improve the performance significantly using parallelization.

The observation we make is that when we are reducing an edge in the above algorithm, we are only affecting vertices within a certain neighborhood of that edge. That means that if we can find edges sufficiently far away that are reducible, we can run a second task to reduce those edges at the same time without their algorithms needing to access any of the same vertices/edges.

As an idealized example see figure 2.14. Note that when the reduction algorithm runs inside the green rectangle, it will not affect any of the data in the blue rectangle, and vice versa. We will come back to the top vertex in the figure in a moment.

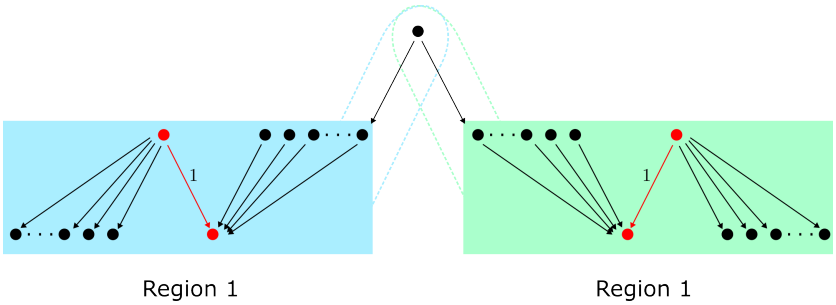


Figure 2.14 Two regions where the reduction algorithm will not impact one another. The top vertex is grouped with them as information to reconstruct the graph afterward.

In order to take advantage of this observation we need to lay groundwork because the type of data we are working with can only be accessed by one process at a time. So we cannot directly reduce two edges simultaneously. Additionally taking the time to break our data into pieces to reduce one edge and reconstruct afterwards will not speed up our problem.

So rather than trying to reduce all the reducible edges simultaneously we will break our graph into regions and do the entire search and reduction loop over each of the regions. The goal is to split the graph into regions as in figure 2.15. If we partition our graph into regions like this that glue up along the grey regions, and the vertices in the blue regions only have edges that extend to the grey region at most, then we can reduce any edge inside the search region without affecting another region.

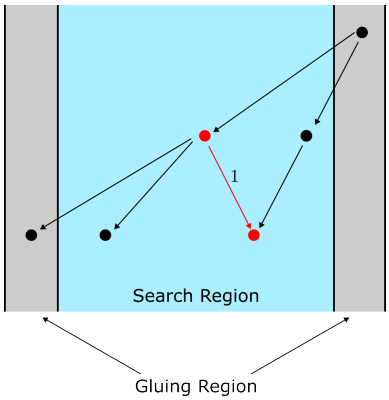


Figure 2.15 Search and glue region schematic for each component of the partition.

We will build our regions by taking subgraphs generated by all vertices a given distance from a selected vertex. This way we can take a sequence of subgraphs of annuli formed in the graph circles as seen in figure 2.16. This allows us to easily get a partition with definite

search and glue regions. This is done just by making sure the outer radius of one region overlaps the inner radius of the next by 1.

We can think of these subgraphs as sort of discs of radius r. We will refer to them as *ego graphs* of radius r keeping in line with the notation of Networkx as it has a built in function to find these subgraphs. It is called `networkx.ego_graph` and we will see it in use when we unpack our similarly named `ego_split` function.

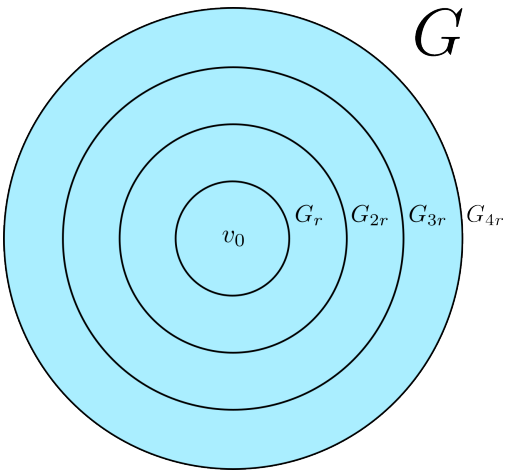


Figure 2.16 Ego graphs of varius radii multiples, denoted G_{cr}

As we noted we need our inner radius and outer radius of the annuli we are going to construct to overlap by 1. However we additionally need to make sure the search region is wide enough to actually have edges that we can reduce. To ensure the regions are wide enough we must ensure the search region's ego distance to be at a minimum 2 units wide. Then if i is the inner radius that reaches the search region, then the edges between $\text{ego}_v(i)$ and $\text{ego}_v(i+1)$ are eligible to be reduced.

So we will split our graph into the annuli defined by $\text{ego}_v(3i+2) \setminus \text{ego}_v(3i-1)$ Graphically the resulting sets can be seen in figure 2.17

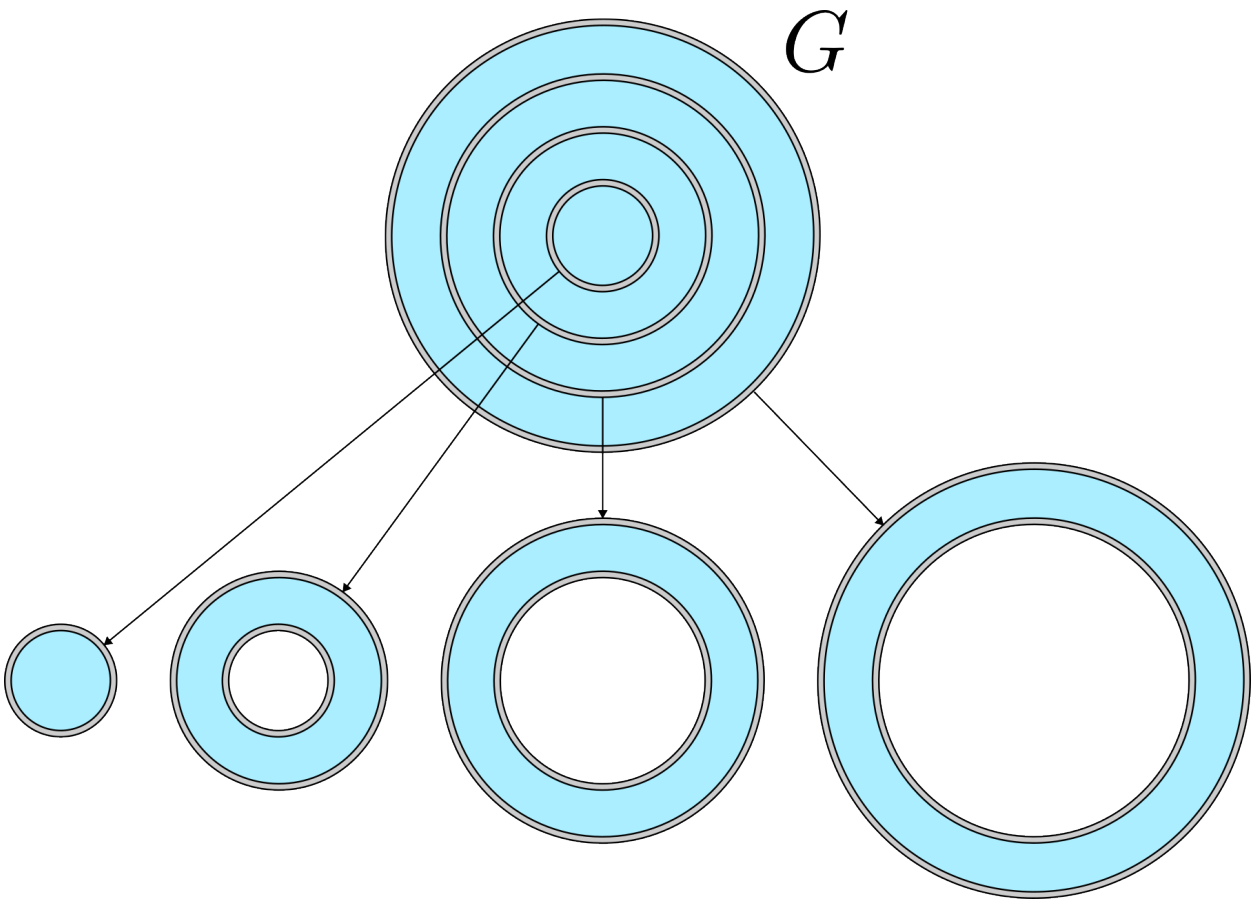


Figure 2.17 Splitting the graph into the annuli use to parallelize our reduction search

2.8.3 Parallelizing the Algorithm

We have all the ingredients necessary now to separate our module and process the pieces in parallel. We will work through the functions in the order they are called. To run the parallel reduction we call `grid_diagram.ego_parallel_red_search` and provide it a cutoff and the number of processors we will let it use. It will split the graph and search the regions in parallel as long as there are at least `cutoff` reducible edges remaining. Once this threshold is met, it will run the non-parallel reduction function from section 2.8.1.

```
def ego_parallel_red_search(self, cutoff = 100, proc_count = 2):

    if len([source for source, target, weight in self.comp.edges(data =
    ↪  'diffweight') if weight == 1]) > cutoff:

    while len([source for source, target, weight in self.comp.edges(data =
    ↪  'diffweight') if weight == 1]) > cutoff:
```

```
            reducible_edge = rd.sample([source for source, target, weight in
            ↪  self.comp.edges(data = 'diffweight') if weight == 1], 1)

            self.ego_parallel_sweep(reducible_edge[0], proc_count)
        self.graph_red_search()

        return
```

This code is essentially just a loop that continuously calls the function `grid_complex.ego_parallel_sweep`, until we have eliminated enough reducible edges to dip below our provided cutoff. Each time it calls the function it provides it a vertex to center the ego radii from. This center vertex is chosen at random from the remaining reducible edges.

If we try to unpack `grid_complex.ego_parallel_sweep`, we will immediately run into a function named `ego_split`. This function finds each subgraph band $\text{ego}_v(i) \setminus \text{ego}_v(i-1)$ for $i \in \{0...n\}$. If the graph is connected this will span the entire graph. This is the case if we are working with the minus or infinity variants since the generators of GFK are the elements of S_n and adjacent transpositions are among the relations. If, however, we convert to another flavor this may not be the case so we will save $G \setminus \text{ego}_v(n)$ to a variable named `safety` and return it along with the bands.

```
def ego_split(graph, vertex, n):

    result = []

    for i in range(n):

        result.append(nx.ego_graph(graph, vertex, i))

    safety = graph.copy()

    safety.remove_nodes_from(result[n - 1].nodes())

    for i in range(n-1, 0, -1):

        result[i].remove_nodes_from(result[i-1].nodes())
```

```
    return result, safety
```

With ego_split understood and out of the way as the first supporting function of grid_complex.ego_parallel_sweep, we can continue to start unpacking it. We will quickly run into more supporting functions.

```
    def ego_parallel_sweep(self, start_vert = None, proc_count = 2):

        if start_vert == None:

            start_vert = self.comp.nodes()[0]

        size = len(list(self.comp.nodes)[0])

        ego_bands, safety = ego_split(self.comp, start_vert, size)

        partition_data = ego_region_partition(size)

        parallel_subgraph_packer(self.comp, ego_bands, partition_data, self.ring)
        .
        .
        .
```

The start of the function has a catch in case a start vertex is not provided, it will pick one. After this it gets the outputs of ego_split we just described and saves the results to ego_bands and safety matching the naming scheme from before. After this we call ego_region_partition. This function defines the radii for the search regions and reserved regions from section 2.8.2. It is a very short function so we will look at it here in the middle of grid_complex.ego_parallel_sweep.

```
def ego_region_partition(n):
    result = {}
    split_count = math.ceil(n/4)
    result["block0"] = {"search_region": [0,1] , "reserved_region": [2]}
    for i in range(1,split_count):
```

```
        result[f"block{i}"] = {"search_region" : [3*i, 3*i+1], "reserved_region" :
        ↪  [3*i-1,3*i+2]}
    return result
```

We are going to make our regions 4 steps wide so the minimum width of 2 in the search region is satisfied. The number of regions then needs to be $\lceil \frac{n}{4} \rceil$ which we store to `split_count`. The regions will be named `blocki` and the initial `block0` is distinct since this "band" only has one edge so we define it separately. Then for each block after that we save it to a dictionary of dictionaries so if we call `result[blocki]` we will get back {`"search_region" : [3*i, 3*i+1], "reserved_region" : [3*i-1,3*i+2]`}.

At this point we have the regions defined and the rings defined by our ego functions. What is left then is to extract the actual subgraphs we will be searching over which is what the function `grid_complex.parallel_subgraph_packer` does and is the last line we have seen thus far from `grid_complex.ego_parallel_sweep`. We will unpack that function now.

```
def parallel_subgraph_packer(graph, subgraphs, region_data, ring):
    for data in region_data:
        region_nodes = []
        for region in region_data[data]:
            for i in region_data[data][region]:
                region_nodes += (list(subgraphs[i].nodes()))
        packed_subgraph = graph.subgraph(region_nodes)
        region_data[data]['total_region'] = grid_complex(packed_subgraph, ring)
    for data in region_data:
        region_nodes = []
        for i in region_data[data]['search_region']:
            region_nodes += list(subgraphs[i].nodes())
        packed_subgraph = graph.subgraph(region_nodes)
        region_data[data]['search_region'] = grid_complex(packed_subgraph, ring)
    return region_data
```

This function is constructed with more generality so we can split our graph differently for parallelization. It expects a list of subgraphs and a dictionary of `region_data` specifying which of those subgraphs are part of each search region and which make up the gluing/reserved regions.

The first loop is over each block and records all the nodes from its regions. It then finds the subgraph they generate in the parent graph and saves this as a `grid_complex` object so that we will have access to our custom methods. The second loop functions similarly but instead we loop over blocks and only the specified search regions. These results overwrite our previous dictionary values of `search_region` and `total_region`.

Then we can end this aside and get back to our explanation of `ego_parallel_sweep`. The next bit of code initializes the parallelization loop.

```
.
.
.
region_count = len(partition_data)
count = 0
MyManager.register('list', list)
with MyManager() as manager:
    processed_subgraphs = manager.list()
    .
    .
    .
```

First we check how many regions we need to assign, saving this to `region_count`. We will use `count` to keep track of how far our loop has progressed. The `MyManager` lines are required for parallel processing, the object it initializes will be how we access and assign jobs to separate processors. Registering it as a list means we can make list like variables that multiple processors can access.

Then the `with` command starts the parallel process and we create one of those list objects. With that available we will start the actual loop.

```
    .
    .
    .
    while count < region_count:

        process_dict = {}
        for i in range(proc_count):
```

```
            if count < region_count:

                process_dict[count] = mp.Process(target = subgraph_red_search,
                ↪  args = (partition_data[f"block{count}"]['total_region'],
                ↪  partition_data[f"block{count}"]['search_region'],
                ↪  processed_subgraphs))
                process_dict[count].start()
                count += 1

        for proc in process_dict:
            process_dict[proc].join()
    .
    .
    .
```

We will run this as while loop, iterating our `count` each time we finish assigning a region to a processor. We will save each process we assign to a dictionary named `process_dict`. We cycle through a for loop for each processor we are allowed in this function. For each of those if there are regions unassigned we will satisfy `count < region_count`. When that is the case we make a process, telling it the target function to run and its arguments. Here it is `subgraph_red_search`. This function can be found on line 995 in the appendix C. It functions similarly to `graph_red_search` from section 2.8.1 but the edges are only reduced from those inside the specified search region.

Ending the first for loop we start the process we just set up and iterate the count. The following loop calls `join` for each of the processes running. This tells our function to wait until each of the processes has concluded before continuing. This repeats until each region is reduced completely.

The function continues after the while loop.

```
    .
    .
    .
    processed_subgraphs = processed_subgraphs._getvalue()
    result = processed_subgraphs[0].comp
```

```
        for element in processed_subgraphs:

            result = nx.compose(result, element.comp)

        result = nx.compose(result, safety)

        self.comp = result

        self.remove_zeros()

        return
```

The results of our processes were saved to `processed_subgraphs`. This was indicated when we started the processes originally. Since this is a special list from the multiprocessing package, we access its contents with `._getvalue()` and save the more accessible list over it.

We are going to take a result graph and keep gluing the reduced subgraphs iteratively until we have our full complex as a result. To start the process we initialize `result` as the first subgraph in the list. From there we call the `networkx.compose(x,y)` function which creates a graph with the vertices of its two arguments and all their edges. We do this for each subgraph we have and then overwrite the overall complex with our new one. The final step removes any edges with weight 0 that may have arisen during the reduction.

CHAPTER 3

RESULTS

The content of this section will be a compilation of some sample results of the program. Note that the number of generators and edges is an upper bound. The program reduces using only the edge reduction algorithm discussed in section 2.8.1, and furthermore it can happen that the order in which the edges are reduced may have an impact on the final count and gradings as well.

The data was produced by generating and reducing the infinity complex using the discussed program, recording the dimensions, Poincaré polynomial, and grading ranges which were then written out. After the complex was converted to the minus flavor and the same data was recorded and written out.

The Poincaré polynomial is computed by starting with 0 and for each vertex adding $u^{M_U(v)} \prod_i t_i^{A_i(v)}$. This encodes all the ranks of the module by Maslov and Alexander Multigrading. The resulting polynomial is factored, often only partially, using Sage's built in factoring algorithm for symbolic expressions.

$k3_1$	
σ_X	$[5, 1, 2, 3, 4]$
σ_O	$[2, 3, 4, 5, 1]$
GFC^∞	
Number of Generators	48
Number of Edges	261

Poincaré Polynomial $11/t_0^3 + t_0/u^4 + 5/u^3 + 11/(t_0u^2) + 14/(t_0^2u) + 5u/t_0^4 + u^2/t_0^5$

Poincaré Polynomial (factored) $(t_0^2 + t_0u + u^2)(t_0 + u)^4/(t_0^5u^4)$

Grading Ranges

$$-4 < UGrading < 0$$

$-2 < VGrading < 2$

$-3 < AGrading < 1$

$-3 < AGrading0 < 1$

GFC^-

Number of Generators	16
Number of Edges	74

Poincaré Polynomial $1/t_0^3 + t_0/u^4 + 4/u^3 + 6/(t_0u^2) + 4/(t_0^2u)$

Poincaré Polynomial (factored) $(t_0 + u)^4/(t_0^3u^4)$

Grading Ranges

$-4 < UGrading < 0$

$-2 < VGrading < 2$

$-3 < AGrading < 1$

$-3 < AGrading0 < 1$

$mk3_1$

σ_X	$[5, 4, 3, 2, 1]$
σ_O	$[2, 1, 5, 4, 3]$

GFC^∞

Number of Generators	48
Number of Edges	310

Poincaré Polynomial $1/t_0^5 + t_0/u^6 + 5/u^5 + 11/(t_0u^4) + 14/(t_0^2u^3) + 11/(t_0^3u^2) + 5/(t_0^4u)$

Poincaré Polynomial (factored) $(t_0^2 + t_0u + u^2)(t_0 + u)^4/(t_0^5u^6)$

Grading Ranges

$-4 < UGrading < 0$

$-6 < VGrading < 0$

$-5 < AGrading < 0$

$-5 < AGrading0 < 0$

GFC^-

Number of Generators	16
Number of Edges	76

Poincaré Polynomial $1/t_0^5 + 1/(t_0u^4) + 4/(t_0^2u^3) + 6/(t_0^3u^2) + 4/(t_0^4u)$

Poincaré Polynomial (factored) $(t_0 + u)^4/(t_0^5u^4)$

Grading Ranges

$-4 < UGrading < 0$

$-6 < VGrading < 0$

$-5 < AGrading < 0$

$-5 < AGrading0 < 0$

$k4_1$

σ_X	$[6, 1, 4, 5, 3, 2]$
σ_O	$[3, 5, 6, 2, 1, 4]$

GFC^∞

Number of Generators	160
Number of Edges	1570

Poincaré Polynomial $8/t_0^5+t_0/u^6+8/u^5+26/(t_0u^4)+45/(t_0^2u^3)+45/(t_0^3u^2)+26/(t_0^4u)+u/t_0^6$

Poincaré Polynomial (factored) $(t_0^2 + 3t_0u + u^2)(t_0 + u)^5/(t_0^6u^6)$

Grading Ranges

$$-5 < UGrading < 0$$

$$-5 < VGrading < 0$$

$$-5 < AGrading < 0$$

$$-5 < AGrading0 < 0$$

GFC^-

Number of Generators	28
Number of Edges	193

Poincaré Polynomial $1/t_0^5 + 1/u^5 + 5/(t_0u^4) + 9/(t_0^2u^3) + 8/(t_0^3u^2) + 4/(t_0^4u)$

Poincaré Polynomial (factored) $(t_0^3 + 3t_0^2u + 2t_0u^2 + u^3)(t_0 + u)^2/(t_0^5u^5)$

Grading Ranges

$$-5 < UGrading < 0$$

$$-5 < VGrading < 0$$

$$-5 < AGrading < 0$$

$$-5 < AGrading0 < 0$$

$k5_1$

σ_X	$[5, 6, 7, 1, 2, 3, 4]$
σ_O	$[7, 3, 4, 5, 6, 1, 2]$
GFC^∞	
Number of Generators	320
Number of Edges	4767

Poincaré Polynomial $57/t_0^4 + t_0^2/u^6 + 7t_0/u^5 + 22/u^4 + 42/(t_0u^3) + 57/(t_0^2u^2) + 62/(t_0^3u) + 41u/t_0^5 + 22u^2/t_0^6 + 7u^3/t_0^7 + u^4/t_0^8 + 1/(t_0^5u)$

Poincaré Polynomial $5/t_0^4 + t_0^2/u^6 + 6t_0/u^5 + 13/u^4 + 30/(t_0u^3) + 21/(t_0^2u^2) + 2/(t_0^3u) + u/t_0^5 + u^3/t_0^7 + u^4/t_0^8 + 1/(t_0^5u)$

Poincaré Polynomial (factored) $(t_0^{10} + 6t_0^9u + 13t_0^8u^2 + 30t_0^7u^3 + 21t_0^6u^4 + 2t_0^5u^5 + 5t_0^4u^6 + t_0^3u^7 + t_0u^9 + u^{10} + t_0^3u^5)/(t_0^8u^6)$

Grading Ranges

$$-6 < UGrading < 4$$

$$-6 < VGrading < 4$$

$$-8 < AGrading < 2$$

$$-8 < AGrading0 < 2$$

GFC^-

Number of Generators	82
Number of Edges	65

Poincaré Polynomial $5/t_0^4 + t_0^2/u^6 + 6t_0/u^5 + 13/u^4 + 30/(t_0u^3) + 21/(t_0^2u^2) + 2/(t_0^3u) + u/t_0^5 + u^3/t_0^7 + u^4/t_0^8 + 1/(t_0^5u)$

Poincaré Polynomial (factored) $(t_0^{10} + 6t_0^9u + 13t_0^8u^2 + 30t_0^7u^3 + 21t_0^6u^4 + 2t_0^5u^5 + 5t_0^4u^6 + t_0^3u^7 + t_0u^9 + u^{10} + t_0^3u^5)/(t_0^8u^6)$

Grading Ranges

$$-6 < UGrading < 4$$

$$-6 < VGrading < 4$$

$$-8 < AGrading < 2$$

$$-8 < AGrading0 < 2$$

$mk5_1$

σ_X	$[2, 1, 7, 6, 5, 4, 3]$
σ_O	$[7, 6, 5, 4, 3, 2, 1]$

GFC^∞

Number of Generators	328
Number of Edges	6062

Poincaré Polynomial $1/t_0^8 + t_0^2/u^{10} + 7t_0/u^9 + 22/u^8 + 43/(t_0u^7) + 56/(t_0^2u^6) + 63/(t_0^3u^5) + 57/(t_0^4u^4) + 41/(t_0^5u^3) + 21/(t_0^6u^2) + 7/(t_0^7u) + 1/(t_0u^8) + 2/(t_0^2u^8) + 1/(t_0^4u^6) + 1/(t_0^5u^5) + 1/(t_0^6u^4) + 1/(t_0^2u^9) + 1/(t_0^3u^8) + 1/(t_0^4u^7)$

Poincaré Polynomial (factored) $(t_0^{10} + 7t_0^9u + 22t_0^8u^2 + 43t_0^7u^3 + 56t_0^6u^4 + 63t_0^5u^5 + 57t_0^4u^6 + 41t_0^3u^7 + 21t_0^2u^8 + 7t_0u^9 + u^{10} + t_0^7u^2 + 2t_0^6u^2 + t_0^4u^4 + t_0^3u^5 + t_0^2u^6 + t_0^6u + t_0^5u^2 + t_0^4u^3)/(t_0^8u^{10})$

Grading Ranges

$$-9 < UGrading < 0$$

$$-10 < VGrading < 0$$

$$-8 < AGrading < 0$$

$$-8 < AGrading0 < 0$$

GFC^-

Number of Generators	44
Number of Edges	193

Poincaré Polynomial $1/t_0^8 + 3/u^8 + 14/(t_0^3u^5) + 1/(t_0^5u^3) + 13/(t_0^6u^2) + 5/(t_0^7u) + 1/(t_0u^8) + 2/(t_0^2u^8) + 1/(t_0^4u^6) + 1/(t_0^5u^5) + 1/(t_0^6u^4) + 1/(t_0^2u^9)$

Poincaré Polynomial (factored) $(3t_0^8u + 14t_0^5u^4 + t_0^3u^6 + 13t_0^2u^7 + 5t_0u^8 + u^9 + t_0^7u + 2t_0^6u + t_0^4u^3 + t_0^3u^4 + t_0^2u^5 + t_0^6)/(t_0^8u^9)$

Grading Ranges

$$-9 < UGrading < 0$$

$$-10 < VGrading < 0$$

$$-8 < AGrading < 0$$

$$-8 < AGrading0 < 0$$

$k5_2$	
σ_X	$[2, 6, 7, 3, 4, 5, 1]$
σ_O	$[7, 1, 5, 6, 2, 3, 4]$
GFC^∞	
Number of Generators	458
Number of Edges	8318

Poincaré Polynomial $1/t_0^3 + u/t_0^4 + 2/t_0^4 + 50/t_0^5 + 2t_0/u^6 + 15/u^5 + 50/(t_0u^4) + 95/(t_0^2u^3) + 122/(t_0^3u^2) + 95/(t_0^4u) + 15u/t_0^6 + 2u^2/t_0^7 + 3/(t_0^3u^3) + 2/(t_0^2u^5) + 3/(t_0^4u^3)$

Poincaré Polynomial (factored) $(t_0^4u^6 + t_0^3u^7 + 2t_0^3u^6 + 2t_0^8 + 15t_0^7u + 50t_0^6u^2 + 95t_0^5u^3 + 122t_0^4u^4 + 95t_0^3u^5 + 50t_0^2u^6 + 15t_0u^7 + 2u^8 + 3t_0^4u^3 + 2t_0^5u + 3t_0^3u^3)/(t_0^7u^6)$

Grading Ranges

$$-6 < UGrading < 2$$
$$-6 < VGrading < 2$$
$$-7 < AGrading < 1$$
$$-7 < AGrading0 < 1$$

GFC^-	
Number of Generators	52
Number of Edges	30

Poincaré Polynomial $u/t_0^4 + 2/t_0^4 + t_0/u^6 + 6/u^5 + 19/(t_0u^4) + 7/(t_0^2u^3) + 2/(t_0^3u^2) + 4/(t_0^4u) + 2u^2/t_0^7 + 3/(t_0^3u^3) + 2/(t_0^2u^5) + 3/(t_0^4u^3)$

Poincaré Polynomial (factored) $(t_0^3u^7 + 2t_0^3u^6 + t_0^8 + 6t_0^7u + 19t_0^6u^2 + 7t_0^5u^3 + 2t_0^4u^4 + 4t_0^3u^5 + 2u^8 + 3t_0^4u^3 + 2t_0^5u + 3t_0^3u^3)/(t_0^7u^6)$

Grading Ranges

$$-6 < UGrading < 2$$

$$-6 < VGrading < 2$$

$$-7 < AGrading < 1$$

$$-7 < AGrading0 < 1$$

$mk5_2$

σ_X	$[4, 1, 7, 6, 3, 2, 5]$
σ_O	$[7, 6, 5, 2, 1, 4, 3]$
GFC^∞	
Number of Generators	450
Number of Edges	8253

Poincaré Polynomial $1/(t_0^3u^2)+2/t_0^7+2t_0/u^8+15/u^7+49/(t_0u^6)+97/(t_0^2u^5)+120/(t_0^3u^4)+97/(t_0^4u^3)+50/(t_0^5u^2)+15/(t_0^6u)+1/(t_0^3u^5)+1/(t_0u^8)$

Poincaré Polynomial (factored) $(t_0^4u^6+2t_0^8+15t_0^7u+49t_0^6u^2+97t_0^5u^3+120t_0^4u^4+97t_0^3u^5+50t_0^2u^6+15t_0u^7+2u^8+t_0^4u^3+t_0^6)/(t_0^7u^8)$

Grading Ranges

$$-8 < UGrading < 0$$

$$-8 < VGrading < 0$$

$$-7 < AGrading < 0$$

$$-7 < AGrading0 < 0$$

GFC^-

Number of Generators	30
Number of Edges	84

Poincaré Polynomial $1/t_0^7+1/(t_0u^6)+3/(t_0^2u^5)+7/(t_0^3u^4)+5/(t_0^4u^3)+5/(t_0^5u^2)+6/(t_0^6u)+1/(t_0^3u^5)+1/(t_0u^8)$

Poincaré Polynomial (factored) $(t_0^6u^2+3t_0^5u^3+7t_0^4u^4+5t_0^3u^5+5t_0^2u^6+6t_0u^7+u^8+t_0^4u^3+t_0^6)/(t_0^7u^8)$

Grading Ranges

$$-8 < UGrading < 0$$

$$-8 < VGrading < 0$$

$$-7 < AGrading < 0$$

$$-7 < AGrading0 < 0$$

$k8_{19}$

σ_X	$[3,2,1,7,6,5,4]$
σ_O	$[7,6,5,4,3,2,1]$
GFC^∞	
Number of Generators	322
Number of Edges	1863

Poincaré Polynomial $1/(t_0^5u^2)+1/u^8+5/(t_0u^7)+15/(t_0^2u^6)+19/(t_0^3u^5)+15/(t_0^4u^4)+6/(t_0^5u^3)+1/t_0^9+t_0^3/u^{12}+7t_0^2/u^{11}+21t_0/u^{10}+35/u^9+35/(t_0u^8)+22/(t_0^2u^7)+14/(t_0^3u^6)+21/(t_0^4u^5)+35/(t_0^5u^4)+34/(t_0^6u^3)+21/(t_0^7u^2)+7/(t_0^8u)+1/(t_0u^9)+1/(t_0^3u^7)+1/(t_0^5u^5)+1/(t_0^6u^4)+1/(t_0^4u^7)+1/(t_0^6u^5)$

Poincaré Polynomial (factored) $(t_0^4u^{10}+t_0^9u^4+5t_0^8u^5+15t_0^7u^6+19t_0^6u^7+15t_0^5u^8+6t_0^4u^9+t_0^{12}+7t_0^{11}u+21t_0^{10}u^2+35t_0^9u^3+35t_0^8u^4+22t_0^7u^5+14t_0^6u^6+21t_0^5u^7+35t_0^4u^8+34t_0^3u^9+21t_0^2u^{10}+7t_0u^{11}+u^{12}+t_0^8u^3+t_0^6u^5+t_0^4u^7+t_0^3u^8+t_0^5u^5+t_0^3u^7)/(t_0^9u^{12})$

Grading Ranges

$-9 < UGrading < 0$

$-12 < VGrading < 0$

$-9 < AGrading < 0$

$-9 < AGrading0 < 0$

GFC^-

Number of Generators	130
Number of Edges	524

Poincaré Polynomial $1/(t_0^5u^2)+1/u^8+2/(t_0u^7)+5/(t_0^2u^6)+5/(t_0^3u^5)+14/(t_0^4u^4)+6/(t_0^5u^3)+1/t_0^9+2/u^9+5/(t_0^3u^6)+8/(t_0^4u^5)+18/(t_0^5u^4)+30/(t_0^6u^3)+19/(t_0^7u^2)+7/(t_0^8u)+1/(t_0u^9)+1/(t_0^3u^7)+1/(t_0^5u^5)+1/(t_0^6u^4)+1/(t_0^4u^7)+1/(t_0^6u^5)$

Poincaré Polynomial (factored) $(t_0^4u^7+t_0^9u+2t_0^8u^2+5t_0^7u^3+5t_0^6u^4+14t_0^5u^5+6t_0^4u^6+2t_0^9+5t_0^6u^3+8t_0^5u^4+18t_0^4u^5+30t_0^3u^6+19t_0^2u^7+7t_0u^8+u^9+t_0^8+t_0^6u^2+t_0^4u^4+t_0^3u^5+t_0^5u^2+t_0^3u^4)/(t_0^9u^9)$

Grading Ranges

$-9 < UGrading < 0$

$-12 < VGrading < 0$

$-9 < AGrading < 0$

$-9 < AGrading0 < 0$

$mk8_{19}$

σ_X	$[3,4,5,6,7,1,2]$
σ_O	$[7,1,2,3,4,5,6]$

GFC^∞

Number of Generators	320
Number of Edges	2422

Poincaré Polynomial $14/t_0^3 + t_0^3/u^6 + 7t_0^2/u^5 + 21t_0/u^4 + 35/u^3 + 35/(t_0u^2) + 22/(t_0^2u) + 22u/t_0^4 + 35u^2/t_0^5 + 35u^3/t_0^6 + 21u^4/t_0^7 + 7u^5/t_0^8 + u^6/t_0^9 + 15/t_0^4 + 1/u^4 + 6/(t_0u^3) + 15/(t_0^2u^2) + 20/(t_0^3u) + 6u/t_0^5 + u^2/t_0^6$

Poincaré Polynomial (factored) $(t_0^6 + t_0^5u + t_0u^5 + u^6 + t_0^3u^2)(t_0 + u)^6/(t_0^9u^6)$

Grading Ranges

$$-6 < UGrading < 3$$

$$-4 < VGrading < 6$$

$$-6 < AGrading < 3$$

$$-6 < AGrading0 < 3$$

GFC^-

Number of Generators	132
Number of Edges	431

Poincaré Polynomial $4/t_0^3 + t_0^3/u^6 + 6t_0^2/u^5 + 15t_0/u^4 + 22/u^3 + 25/(t_0u^2) + 20/(t_0^2u) + 3u/t_0^4 + 6u^2/t_0^5 + u^3/t_0^6 + 14/t_0^4 + 8/(t_0^3u) + 6u/t_0^5 + u^2/t_0^6$

Poincaré Polynomial (factored) $(t_0^9 + 6t_0^8u + 15t_0^7u^2 + 22t_0^6u^3 + 25t_0^5u^4 + 20t_0^4u^5 + 4t_0^3u^6 + 3t_0^2u^7 + 6t_0u^8 + u^9 + 8t_0^3u^5 + 14t_0^2u^6 + 6t_0u^7 + u^8)/(t_0^6u^6)$

Grading Ranges

$$-6 < UGrading < 3$$

$$-4 < VGrading < 6$$

$$-6 < AGrading < 3$$

$$-6 < AGrading0 < 3$$

$L2a1_0$

σ_X	$[4,3,2,1]$
σ_O	$[2,1,4,3]$
GFC^∞	
Number of Generators	16
Number of Edges	56

Poincaré Polynomial $1/(t_0^{(3/2)}t_1^{(3/2)})+t_0^{(1/2)}t_1^{(1/2)}/u^4+2t_0^{(1/2)}/(t_1^{(1/2)}u^3)+2t_1^{(1/2)}/(t_0^{(1/2)}u^3)+t_0^{(1/2)}/(t_1^{(3/2)}u^2)+4/(t_0^{(1/2)}t_1^{(1/2)}u^2)+t_1^{(1/2)}/(t_0^{(3/2)}u^2)+2/(t_0^{(1/2)}t_1^{(3/2)}u)+2/(t_0^{(3/2)}t_1^{(1/2)}u)$

Poincaré Polynomial (factored) $(t_0+u)^2(t_1+u)^2/(t_0^{(3/2)}t_1^{(3/2)}u^4)$

Grading Ranges

$$-3 < UGrading < 0$$

$$-4 < VGrading < 0$$

$$-3 < AGrading < 0$$

$$-3/2 < AGrading0 < 1/2$$

$$-3/2 < AGrading1 < 1/2$$

GFC^-	
Number of Generators	8
Number of Edges	15

Poincaré Polynomial $1/(t_0^{(3/2)}t_1^{(3/2)})+t_0^{(1/2)}/(t_1^{(1/2)}u^3)+t_0^{(1/2)}/(t_1^{(3/2)}u^2)+1/(t_0^{(1/2)}t_1^{(1/2)}u^2)+t_1^{(1/2)}/(t_0^{(3/2)}u^2)+1/(t_0^{(1/2)}t_1^{(3/2)}u)+2/(t_0^{(3/2)}t_1^{(1/2)}u)$

Poincaré Polynomial (factored) $(t_0^2+t_0u+t_1u+u^2)(t_1+u)/(t_0^{(3/2)}t_1^{(3/2)}u^3)$

Grading Ranges

$$-3 < UGrading < 0$$

$$-4 < VGrading < 0$$

$$-3 < AGrading < 0$$

$$-3/2 < AGrading0 < 1/2$$

$$-3/2 < AGrading1 < 1/2$$

$L2a1_1$

σ_X	$[4,1,2,3]$
σ_O	$[2,3,4,1]$
GFC^∞	
Number of Generators	16
Number of Edges	56

Poincaré Polynomial $2t_0^{(1/2)}/t_1^{(1/2)} + 2t_1^{(1/2)}/t_0^{(1/2)} + t_0^{(3/2)}t_1^{(3/2)}/u^3 + 2t_0^{(3/2)}t_1^{(1/2)}/u^2 + 2t_0^{(1/2)}t_1^{(3/2)}/u^2 + t_0^{(3/2)}/(t_1^{(1/2)}u) + 4t_0^{(1/2)}t_1^{(1/2)}/u + t_1^{(3/2)}/(t_0^{(1/2)}u) + u/(t_0^{(1/2)}t_1^{(1/2)})$

Poincaré Polynomial (factored) $(t_0+u)^2(t_1+u)^2/(t_0^{(1/2)}t_1^{(1/2)}u^3)$

Grading Ranges

$$-3 < UGrading < 0$$

$$-2 < VGrading < 1$$

$$0 < AGrading < 3$$

$$-1/2 < AGrading0 < 3/2$$

$$0 < AGrading1 < 3/2$$

GFC^-

Number of Generators	8
Number of Edges	18

Poincaré Polynomial $t_1^{(1/2)}/t_0^{(1/2)} + t_0^{(3/2)}t_1^{(3/2)}/u^3 + t_0^{(3/2)}t_1^{(1/2)}/u^2 + 2t_0^{(1/2)}t_1^{(3/2)}/u^2 + 3t_0^{(1/2)}t_1^{(1/2)}/u$

Poincaré Polynomial (factored) $(t_0^2t_1 + t_0^2u + 2t_0t_1u + 3t_0u^2 + u^3)t_1^{(1/2)}/(t_0^{(1/2)}u^3)$

Grading Ranges

$$-3 < UGrading < 0$$

$$-2 < VGrading < 1$$

$$0 < AGrading < 3$$

$$-1/2 < AGrading0 < 3/2$$

$$0 < AGrading1 < 3/2$$

$L4a1_0$

σ_X	$[6, 3, 4, 1, 2, 5]$
σ_O	$[4, 5, 2, 3, 6, 1]$
GFC^∞	
Number of Generators	128
Number of Edges	1203

Poincaré Polynomial $3/t_0^2+3/t_1^2+6/(t_0t_1)+t_0^2t_1/u^5+t_0t_1^2/u^5+3t_0^2/u^4+6t_0t_1/u^4+3t_1^2/u^4+12t_0/u^3 + 3t_0^2/(t_1u^3) + 12t_1/u^3 + 3t_1^2/(t_0u^3) + t_0^2/(t_1^2u^2) + 10t_0/(t_1u^2) + 10t_1/(t_0u^2) + t_1^2/(t_0^2u^2) + 18/u^2 + 12/(t_0u) + 3t_0/(t_1^2u) + 12/(t_1u) + 3t_1/(t_0^2u) + u/(t_0t_1^2) + u/(t_0^2t_1)$

Poincaré Polynomial (factored) $(t_0 + t_1)(t_0 + u)^3(t_1 + u)^3/(t_0^2t_1^2u^5)$

Grading Ranges

$$-5 < UGrading < 0$$

$$-4 < VGrading < 1$$

$$-2 < AGrading < 3$$

$$-1 < AGrading0 < 2$$

$$-1 < AGrading1 < 2$$

GFC^{-}

Number of Generators	30
Number of Edges	239

Poincaré Polynomial $1/(t_0t_1) + t_0^2t_1/u^5 + t_0^2/u^4 + 3t_0t_1/u^4 + t_1^2/u^4 + 6t_0/u^3 + 3t_1/u^3 + 5t_0/(t_1u^2) + 3t_1/(t_0u^2) + 1/u^2 + 3/(t_0u) + 2/(t_1u)$

Poincaré Polynomial (factored) $(t_0^3t_1^2+t_0^3t_1u+3t_0^2t_1^2u+t_0t_1^3u+6t_0^2t_1u^2+3t_0t_1^2u^2+5t_0^2u^3+t_0t_1u^3+3t_1^2u^3+2t_0u^4+3t_1u^4+u^5)/(t_0t_1u^5)$

Grading Ranges

$$-5 < UGrading < 0$$
$$-4 < VGrading < 1$$
$$-2 < AGrading < 3$$
$$-1 < AGrading0 < 2$$
$$-1 < AGrading1 < 2$$

$L4a1_1$

σ_X	$[3, 2, 6, 1, 5, 4]$
σ_O	$[6, 5, 4, 3, 2, 1]$

GFC^{∞}

Number of Generators	128
Number of Edges	1119

Poincaré Polynomial $1/(t_0^{(11/2)}t_1^{(5/2)}) + t_1^{(1/2)}/(t_0^{(1/2)}u^8) + 2/(t_0^{(1/2)}t_1^{(1/2)}u^7) + 4t_1^{(1/2)}/(t_0^{(3/2)}u^7) + 1/(t_0^{(1/2)}t_1^{(3/2)}u^6) + 9/(t_0^{(3/2)}t_1^{(1/2)}u^6) + 6t_1^{(1/2)}/(t_0^{(5/2)}u^6) + 6/(t_0^{(3/2)}t_1^{(3/2)}u^5) + 16/(t_0^{(5/2)}t_1^{(1/2)}u^5) + 4t_1^{(1/2)}/(t_0^{(7/2)}u^5) + 1/(t_0^{(3/2)}t_1^{(5/2)}u^4) + 14/(t_0^{(5/2)}t_1^{(3/2)}u^4) + 14/(t_0^{(7/2)}t_1^{(1/2)}u^4) + t_1^{(1/2)}/(t_0^{(9/2)}u^4) + 4/(t_0^{(5/2)}t_1^{(5/2)}u^3) +$

$16/(t_0^{(7/2)}t_1^{(3/2)}u^3) + 6/(t_0^{(9/2)}t_1^{(1/2)}u^3) + 6/(t_0^{(7/2)}t_1^{(5/2)}u^2) + 9/(t_0^{(9/2)}t_1^{(3/2)}u^2) + 1/(t_0^{(11/2)}t_1^{(1/2)}u^2) + 4/(t_0^{(9/2)}t_1^{(5/2)}u) + 2/(t_0^{(11/2)}t_1^{(3/2)}u)$

Poincaré Polynomial (factored) $(t_0t_1+u^2)(t_0+u)^4(t_1+u)^2/(t_0^{(11/2)}t_1^{(5/2)}u^8)$

Grading Ranges

$-5 < UGrading < 0$

$-8 < VGrading < 0$

$-8 < AGrading < 0$

$-11/2 < AGrading0 < 0$

$-5/2 < AGrading1 < 0$

GFC^-

Number of Generators 28

Number of Edges 134

Poincaré Polynomial $1/(t_0^{(11/2)}t_1^{(5/2)}) + 1/(t_0^{(5/2)}t_1^{(1/2)}u^5) + 2/(t_0^{(5/2)}t_1^{(3/2)}u^4) + 1/(t_0^{(5/2)}t_1^{(5/2)}u^3) + 4/(t_0^{(7/2)}t_1^{(3/2)}u^3) + 3/(t_0^{(9/2)}t_1^{(1/2)}u^3) + 3/(t_0^{(7/2)}t_1^{(5/2)}u^2) + 7/(t_0^{(9/2)}t_1^{(3/2)}u^2) + 1/(t_0^{(11/2)}t_1^{(1/2)}u^2) + 4/(t_0^{(9/2)}t_1^{(5/2)}u) + 1/(t_0^{(11/2)}t_1^{(3/2)}u)$

Poincaré Polynomial (factored) $(t_0^3t_1^2+2t_0^3t_1u+t_0^3u^2+4t_0^2t_1u^2+3t_0t_1^2u^2+3t_0^2u^3+7t_0t_1u^3+t_1^2u^3+4t_0u^4+t_1u^4+u^5)/(t_0^{(11/2)}t_1^{(5/2)}u^5)$

Grading Ranges

$-5 < UGrading < 0$

$-8 < VGrading < 0$

$-8 < AGrading < 0$

$-11/2 < AGrading0 < 0$

$-5/2 < AGrading1 < 0$

$L5a1_0$

σ_X	$[6,1,3,4,2,7,5]$
σ_O	$[2,4,5,7,6,3,1]$
GFC^∞	
Number of Generators	512
Number of Edges	10240

Poincaré Polynomial $13/t_0^{(7/2)}+6/(t_0^{(5/2)}t_1)+16t_1/t_0^{(9/2)}+t_1^2/t_0^{(11/2)}+t_0^{(3/2)}t_1^2/u^7+t_0^{(5/2)}/u^6+2t_0^{(3/2)}t_1/u^6+6t_0^{(1/2)}t_1^2/u^6+3t_0^{(3/2)}/u^5+13t_0^{(1/2)}t_1/u^5+18t_1^2/(t_0^{(1/2)}u^5)+2t_1^3/(t_0^{(3/2)}u^5)+14t_0^{(1/2)}/u^4+t_0^{(3/2)}/(t_1u^4)+35t_1/(t_0^{(1/2)}u^4)+26t_1^2/(t_0^{(3/2)}u^4)+8t_1^3/(t_0^{(5/2)}u^4)+29/(t_0^{(1/2)}u^3)+2t_0^{(1/2)}/(t_1u^3)+53t_1/(t_0^{(3/2)}u^3)+33t_1^2/(t_0^{(5/2)}u^3)+9t_1^3/(t_0^{(7/2)}u^3)+35/(t_0^{(3/2)}u^2)+1/(t_0^{(1/2)}t_1u^2)+62t_1/(t_0^{(5/2)}u^2)+26t_1^2/(t_0^{(7/2)}u^2)+2t_1^3/(t_0^{(9/2)}u^2)+34/(t_0^{(5/2)}u)+5/(t_0^{(3/2)}t_1u)+33t_1/(t_0^{(7/2)}u)+12t_1^2/(t_0^{(9/2)}u)+6u/t_0^{(9/2)}+u/(t_0^{(5/2)}t_1^2)+u/(t_0^{(7/2)}t_1)+t_1u/t_0^{(11/2)}+u^2/t_0^{(11/2)}$

Poincaré Polynomial (factored) $(t_0^7t_1^4+t_0^8t_1^2u+2t_0^7t_1^3u+6t_0^6t_1^4u+3t_0^7t_1^2u^2+13t_0^6t_1^3u^2+18t_0^5t_1^4u^2+2t_0^4t_1^5u^2+t_0^7t_1u^3+14t_0^6t_1^2u^3+35t_0^5t_1^3u^3+26t_0^4t_1^4u^3+8t_0^3t_1^5u^3+2t_0^6t_1u^4+29t_0^5t_1^2u^4+53t_0^4t_1^3u^4+33t_0^3t_1^4u^4+9t_0^2t_1^5u^4+t_0^5t_1u^5+35t_0^4t_1^2u^5+62t_0^3t_1^3u^5+26t_0^2t_1^4u^5+2t_0t_1^5u^5+5t_0^4t_1u^6+34t_0^3t_1^2u^6+33t_0^2t_1^3u^6+12t_0t_1^4u^6+6t_0^3t_1u^7+13t_0^2t_1^2u^7+16t_0t_1^3u^7+t_1^4u^7+t_0^3u^8+t_0^2t_1u^8+6t_0t_1^2u^8+t_1^3u^8+t_1^2u^9)/(t_0^{(11/2)}t_1^2u^7)$

Grading Ranges

$$-6 < UGrading < 0$$

$$-5 < VGrading < 1$$

$$-7/2 < AGrading < 5/2$$

$$-9/2 < AGrading0 < 1/2$$

$$0 < AGrading1 < 3$$

GFC^-

Number of Generators	30
Number of Edges	78

Poincaré Polynomial $2/t_0^{(7/2)} + 2t_1/t_0^{(9/2)} + t_0^{(1/2)}t_1^2/u^6 + t_0^{(1/2)}t_1/u^5 + 3t_1^2/(t_0^{(1/2)}u^5) + t_1^3/(t_0^{(3/2)}u^5) + t_0^{(1/2)}/u^4 + 4t_1/(t_0^{(1/2)}u^4) + 4t_1^2/(t_0^{(3/2)}u^4) + t_1^3/(t_0^{(5/2)}u^4) + 5t_1/(t_0^{(3/2)}u^3) + 4t_1^2/(t_0^{(5/2)}u^3) + t_1/(t_0^{(7/2)}u)$

Poincaré Polynomial (factored) $(t_0^5t_1^2 + t_0^5t_1u + 3t_0^4t_1^2u + t_0^3t_1^3u + t_0^5u^2 + 4t_0^4t_1u^2 + 4t_0^3t_1^2u^2 + t_0^2t_1^3u^2 + 5t_0^3t_1u^3 + 4t_0^2t_1^2u^3 + t_0t_1u^5 + 2t_0u^6 + 2t_1u^6)/(t_0^{(9/2)}u^6)$

Grading Ranges

$$-6 < UGrading < 0$$

$$-5 < VGrading < 1$$

$$-7/2 < AGrading < 5/2$$

$$-9/2 < AGrading0 < 1/2$$

$$0 < AGrading1 < 3$$

$L5a1_1$

σ_X	$[3, 6, 5, 7, 1, 2, 4]$
σ_O	$[5, 4, 2, 3, 6, 7, 1]$
GFC^∞	
Number of Generators	516
Number of Edges	10504

Poincaré Polynomial $1/(t_0^{(1/2)}t_1^4) + 1/(t_0^{(3/2)}t_1^2u) + u/(t_0^{(5/2)}t_1^3) + u/(t_0^{(7/2)}t_1^2) + 3/(t_0^{(1/2)}t_1^6) + 7/(t_0^{(3/2)}t_1^5) + 13/(t_0^{(5/2)}t_1^4) + 13/(t_0^{(7/2)}t_1^3) + t_1/(t_0^{(1/2)}u^7) + 7/(t_0^{(1/2)}u^6) + t_0^{(3/2)}/(t_1^2u^6) + t_0^{(1/2)}/(t_1u^6) + 12/(t_0^{(3/2)}u^5) + t_0^{(3/2)}/(t_1^3u^5) + 5t_0^{(1/2)}/(t_1^2u^5) + 18/(t_0^{(1/2)}t_1u^5) + 9/(t_0^{(5/2)}u^4) + 7t_0^{(1/2)}/(t_1^3u^4) + 33/(t_0^{(1/2)}t_1^2u^4) + 35/(t_0^{(3/2)}t_1u^4) + 1/(t_0^{(7/2)}u^3) + 5t_0^{(1/2)}/(t_1^4u^3) + 30/(t_0^{(1/2)}t_1^3u^3) + 63/(t_0^{(3/2)}t_1^2u^3) + 28/(t_0^{(5/2)}t_1u^3) + 3t_0^{(1/2)}/(t_1^5u^2) + 21/(t_0^{(1/2)}t_1^4u^2) +$

$54/(t_0^{(3/2)}t_1^3u^2) + 41/(t_0^{(5/2)}t_1^2u^2) + 6/(t_0^{(7/2)}t_1u^2) + 7/(t_0^{(1/2)}t_1^5u) + 34/(t_0^{(3/2)}t_1^4u) + 29/(t_0^{(5/2)}t_1^3u)+12/(t_0^{(7/2)}t_1^2u)+5u/(t_0^{(5/2)}t_1^5)+4u/(t_0^{(7/2)}t_1^4)+u^2/(t_0^{(7/2)}t_1^5)+1/(t_0^{(1/2)}t_1^3u^4)+1/(t_0^{(5/2)}t_1u^4)$

Poincaré Polynomial (factored) $(t_0^2t_1^4u^6 + t_0^3t_1^2u^7 + t_0t_1^3u^8 + t_1^4u^8 + t_0^3t_1^7 + t_0^5t_1^4u + t_0^4t_1^5u + 7t_0^3t_1^6u + t_0^5t_1^3u^2 + 5t_0^4t_1^4u^2 + 18t_0^3t_1^5u^2 + 12t_0^2t_1^6u^2 + 7t_0^4t_1^3u^3 + 33t_0^3t_1^4u^3 + 35t_0^2t_1^5u^3 + 9t_0t_1^6u^3 + 5t_0^4t_1^2u^4 + 30t_0^3t_1^3u^4 + 63t_0^2t_1^4u^4 + 28t_0t_1^5u^4 + t_1^6u^4 + 3t_0^4t_1u^5 + 21t_0^3t_1^2u^5 + 54t_0^2t_1^3u^5 + 41t_0t_1^4u^5 + 6t_1^5u^5 + 7t_0^3t_1u^6 + 34t_0^2t_1^2u^6 + 29t_0t_1^3u^6 + 12t_1^4u^6 + 3t_0^3u^7 + 7t_0^2t_1u^7 + 13t_0t_1^2u^7 + 13t_1^3u^7 + 5t_0t_1u^8 + 4t_1^2u^8 + t_1u^9 + t_0^3t_1^3u^3 + t_0t_1^5u^3)/(t_0^{(7/2)}t_1^6u^7)$

Grading Ranges

$$-6 < UGrading < 1$$

$$-5 < VGrading < 1$$

$$-13/2 < AGrading < 0$$

$$-7/2 < AGrading0 < 1/2$$

$$-4 < AGrading1 < 0$$

GFC^-

Number of Generators	28
Number of Edges	28

Poincaré Polynomial $1/(t_0^{(1/2)}t_1^4) + u/(t_0^{(5/2)}t_1^3) + u/(t_0^{(7/2)}t_1^2) + 1/(t_0^{(7/2)}t_1^3) + 1/(t_0^{(1/2)}u^6) + 1/(t_0^{(3/2)}u^5)+1/(t_0^{(1/2)}t_1u^5)+1/(t_0^{(5/2)}u^4)+1/(t_0^{(1/2)}t_1^2u^4)+4/(t_0^{(3/2)}t_1u^4)+t_0^{(1/2)}/(t_1^4u^3)+1/(t_0^{(1/2)}t_1^3u^3) + 4/(t_0^{(3/2)}t_1^2u^3) + 1/(t_0^{(5/2)}t_1u^3) + 3/(t_0^{(3/2)}t_1^3u^2) + 3/(t_0^{(5/2)}t_1^3u) + 1/(t_0^{(1/2)}t_1^3u^4) + 1/(t_0^{(5/2)}t_1u^4)$

Poincaré Polynomial (factored) $(t_0^3u^6 + t_0t_1u^7 + t_1^2u^7 + t_0^3t_1^4 + t_0^3t_1^3u + t_0^2t_1^4u + t_0^3t_1^2u^2 + 4t_0^2t_1^3u^2 + t_0t_1^4u^2 + t_0^4u^3 + t_0^3t_1u^3 + 4t_0^2t_1^2u^3 + t_0t_1^3u^3 + 3t_0^2t_1u^4 + 3t_0t_1u^5 + t_1u^6 + t_0^3t_1u^2 + t_0t_1^3u^2)/(t_0^{(7/2)}t_1^4u^6)$

Grading Ranges

$$-6 < UGrading < 1$$

$$-5 < VGrading < 1$$

$$-13/2 < AGrading < 0$$

$$-7/2 < AGrading0 < 1/2$$

$$-4 < AGrading1 < 0$$

$L6n1_{00}$

σ_X	$[6, 1, 5, 3, 4, 2]$
σ_O	$[3, 4, 2, 6, 1, 5]$
GFC^∞	
Number of Generators	142
Number of Edges	1182

Poincaré Polynomial $2/(t_0^2t_1^3) + 2/(t_0^3t_1^2) + 1/(t_0^2t_1^2t_2) + 2t_2/(t_0^3t_1^3) + t_2/u^6 + 2t_2/(t_0u^5) + 2t_2/(t_1u^5) + t_2^2/(t_0t_1u^5) + 2/u^5 + 4/(t_0u^4) + 4/(t_1u^4) + 1/(t_2u^4) + t_2/(t_0^2u^4) + t_2/(t_1^2u^4) + 6t_2/(t_0t_1u^4) + 2t_2^2/(t_0t_1^2u^4) + 2t_2^2/(t_0^2t_1u^4) + 2/(t_0^2u^3) + 2/(t_1^2u^3) + 10/(t_0t_1u^3) + 2/(t_0t_2u^3) + 2/(t_1t_2u^3) + 7t_2/(t_0t_1^2u^3) + 7t_2/(t_0^2t_1u^3) + t_2^2/(t_0t_1^3u^3) + 4t_2^2/(t_0^2t_1^2u^3) + t_2^2/(t_0^3t_1u^3) + 7/(t_0t_1^2u^2) + 7/(t_0^2t_1u^2) + 1/(t_0^2t_2u^2) + 1/(t_1^2t_2u^2) + 4/(t_0t_1t_2u^2) + 2t_2/(t_0t_1^3u^2) + 10t_2/(t_0^2t_1^2u^2) + 2t_2/(t_0^3t_1u^2) + 2t_2^2/(t_0^2t_1^3u^2) + 2t_2^2/(t_0^3t_1^2u^2) + 1/(t_0t_1^3u) + 7/(t_0^2t_1^2u) + 1/(t_0^3t_1u) + 2/(t_0t_1^2t_2u) + 2/(t_0^2t_1t_2u) + 4t_2/(t_0^2t_1^3u) + 4t_2/(t_0^3t_1^2u) + t_2^2/(t_0^3t_1^3u) + u/(t_0^3t_1^3) + 1/(t_0t_1u^4) + t_2/(t_0t_1^2u^4) + t_2/(t_0^2t_1u^4) + 1/(t_0t_1^2u^3) + 1/(t_0^2t_1u^3) + t_2/(t_0^2t_1^2u^3) + 1/(t_0^2t_1^2u^2)$

Poincaré Polynomial (factored) $(t_0^3t_1^3t_2^2 + 2t_0^3t_1^3t_2u + 2t_0^3t_1^2t_2^2u + 2t_0^2t_1^3t_2^2u + t_0^2t_1^2t_2^3u + t_0^3t_1^3u^2 + 4t_0^3t_1^2t_2u^2 + 4t_0^2t_1^3t_2u^2 + t_0^3t_1t_2^2u^2 + 6t_0^2t_1^2t_2^2u^2 + t_0t_1^3t_2^2u^2 + 2t_0^2t_1t_2^3u^2 + 2t_0t_1^2t_2^3u^2 + 2t_0^3t_1^2u^3 + 2t_0^2t_1^3u^3 + 2t_0^3t_1t_2u^3 + 10t_0^2t_1^2t_2u^3 + 2t_0t_1^3t_2u^3 + 7t_0^2t_1t_2^2u^3 + 7t_0t_1^2t_2^2u^3 + t_0^2t_2^3u^3 + 4t_0t_1t_2^3u^3 + t_1^2t_2^3u^3 + t_0^3t_1u^4 + 4t_0^2t_1^2u^4 + t_0t_1^3u^4 + 7t_0^2t_1t_2u^4 + 7t_0t_1^2t_2u^4 + 2t_0^2t_2^2u^4 + 10t_0t_1t_2^2u^4 + 2t_1^2t_2^2u^4 + 2t_0t_2^3u^4 + 2t_1t_2^3u^4 + 2t_0^2t_1u^5 + 2t_0t_1^2u^5 + t_0^2t_2u^5 + 7t_0t_1t_2u^5 + t_1^2t_2u^5 + 4t_0t_2^2u^5 + 4t_1t_2^2u^5 +$

$t_2^3u^5+t_0t_1u^6+2t_0t_2u^6+2t_1t_2u^6+2t_2^2u^6+t_2u^7+t_0^2t_1^2t_2u^2+t_0^2t_1t_2^2u^2+t_0t_1^2t_2^2u^2+t_0^2t_1t_2u^3+$
$t_0t_1^2t_2u^3+t_0t_1t_2^2u^3+t_0t_1t_2u^4)/(t_0^3t_1^3t_2u^6)$

Grading Ranges

$$-4 < UGrading < 1$$
$$-6 < VGrading < 0$$
$$-6 < AGrading < 0$$
$$-3 < AGrading0 < 0$$
$$-3 < AGrading1 < 0$$
$$-1 < AGrading2 < 2$$

GFC^-

Number of Generators	42
Number of Edges	168

Poincaré Polynomial $1/(t_0^2t_1^3)+1/(t_0^3t_1^2)+2t_2/(t_0^3t_1^3)+2/(t_0u^4)+2/(t_1u^4)+1/(t_2u^4)+$
$1/(t_1^2u^3)+3/(t_0t_1u^3)+2/(t_0t_2u^3)+2/(t_1t_2u^3)+t_2/(t_0t_1^2u^3)+t_2^2/(t_0^2t_1^2u^3)+3/(t_0t_1^2u^2)+$
$2/(t_0^2t_1u^2)+1/(t_0^2t_2u^2)+1/(t_1^2t_2u^2)+t_2/(t_0^2t_1^2u^2)+t_2^2/(t_0^3t_1^2u^2)+1/(t_0t_1^2t_2u)+2/(t_0^2t_1t_2u)+$
$2t_2/(t_0^2t_1^3u)+2t_2/(t_0^3t_1^2u)+u/(t_0^3t_1^3)+t_2/(t_0t_1^2u^4)+t_2/(t_0^2t_1u^4)+1/(t_0t_1^2u^3)+1/(t_0^2t_1u^3)+$
$t_2/(t_0^2t_1^2u^3)+1/(t_0^2t_1^2u^2)$

Poincaré Polynomial (factored) $(t_0^3t_1^3+2t_0^3t_1^2t_2+2t_0^2t_1^3t_2+2t_0^3t_1^2u+2t_0^2t_1^3u+t_0^3t_1t_2u+$
$3t_0^2t_1^2t_2u+t_0^2t_1t_2^2u+t_0t_1t_2^3u+t_0^3t_1u^2+t_0t_1^3u^2+3t_0^2t_1t_2u^2+2t_0t_1^2t_2u^2+t_0t_1t_2^2u^2+t_1t_2^3u^2+$
$t_0^2t_1u^3+2t_0t_1^2u^3+2t_0t_2^2u^3+2t_1t_2^2u^3+t_0t_2u^4+t_1t_2u^4+2t_2^2u^4+t_2u^5+t_0^2t_1t_2^2+t_0t_1^2t_2^2+$
$t_0^2t_1t_2u+t_0t_1^2t_2u+t_0t_1t_2^2u+t_0t_1t_2u^2)/(t_0^3t_1^3t_2u^4)$

Grading Ranges

$$-4 < UGrading < 1$$
$$-6 < VGrading < 0$$

$$-6 < AGrading < 0$$

$$-3 < AGrading0 < 0$$

$$-3 < AGrading1 < 0$$

$$-1 < AGrading2 < 2$$

$L6n1_{10}$

σ_X	$[3, 4, 2, 6, 1, 5]$
σ_O	$[6, 1, 5, 3, 4, 2]$
GFC^∞	
Number of Generators	144
Number of Edges	1240

Poincaré Polynomial $2/(t_0^2t_1) + 1/(t_0t_2^2) + 2/(t_0^2t_2) + 2/(t_0t_1t_2) + t_0t_1^2/u^6 + 2t_0t_1/u^5 + 2t_1^2/u^5 + 2t_0t_1^2/(t_2u^5) + t_1t_2/u^5 + t_0/u^4 + 7t_1/u^4 + t_1^2/(t_0u^4) + t_0t_1^2/(t_2^2u^4) + 4t_0t_1/(t_2u^4) + 4t_1^2/(t_2u^4) + 2t_1t_2/(t_0u^4) + 2t_2/u^4 + 7t_1/(t_0u^3) + 2t_0t_1/(t_2^2u^3) + 2t_1^2/(t_2^2u^3) + 2t_0/(t_2u^3) + 10t_1/(t_2u^3) + 2t_1^2/(t_0t_2u^3) + 4t_2/(t_0u^3) + t_2/(t_1u^3) + t_1t_2/(t_0^2u^3) + 7/u^3 + 10/(t_0u^2) + 2/(t_1u^2) + 2t_1/(t_0^2u^2) + t_0/(t_2^2u^2) + 4t_1/(t_2^2u^2) + t_1^2/(t_0t_2^2u^2) + 7t_1/(t_0t_2u^2) + 7/(t_2u^2) + 2t_2/(t_0^2u^2) + 2t_2/(t_0t_1u^2) + 4/(t_0^2u) + 4/(t_0t_1u) + 2t_1/(t_0t_2^2u) + 2/(t_2^2u) + 7/(t_0t_2u) + 1/(t_1t_2u) + t_1/(t_0^2t_2u) + t_2/(t_0^2t_1u) + u/(t_0^2t_1t_2) + t_1/u^5 + t_1/(t_0u^4) + t_1/(t_2u^4) + 1/u^4 + 1/(t_0u^3) + t_1/(t_0t_2u^3) + 1/(t_2u^3) + 1/(t_0t_2u^2)$

Poincaré Polynomial (factored) $(t_0^2t_1^2t_2 + t_0^2t_1^2u + t_0^2t_1t_2u + t_0t_1^2t_2u + t_0t_1t_2^2u + t_0^2t_1u^2 + t_0t_1^2u^2 + 3t_0t_1t_2u^2 + t_0t_2^2u^2 + t_1t_2^2u^2 + t_0t_1u^3 + t_0t_2u^3 + t_1t_2u^3 + t_2^2u^3 + t_2u^4 + t_0t_1t_2u)(t_0 + u)(t_1 + u)(t_2 + u)/(t_0^2t_1t_2^2u^6)$

Grading Ranges

$$-5 < UGrading < 1$$

$$-6 < VGrading < 0$$

$$-4 < AGrading < 2$$

$$-2 < AGrading0 < 1$$

$$-1 < AGrading1 < 2$$

$$-2 < AGrading2 < 1$$

GFC^-

Number of Generators	40
Number of Edges	188

Poincaré Polynomial $2/(t_0^2t_1)+1/(t_0t_2^2)+1/(t_0^2t_2)+t_0t_1^2/(t_2u^5)+2t_0t_1/(t_2u^4)+2t_1^2/(t_2u^4)+t_1/(t_0u^3)+2t_0t_1/(t_2^2u^3)+2t_1^2/(t_2^2u^3)+2t_1/(t_2u^3)+t_2/(t_0u^3)+1/u^3+1/(t_0u^2)+t_0/(t_2^2u^2)+t_1/(t_2^2u^2)+t_1^2/(t_0t_2^2u^2)+t_1/(t_0t_2u^2)+2/(t_2u^2)+t_2/(t_0^2u^2)+1/(t_0^2u)+1/(t_0t_1u)+t_1/(t_0t_2^2u)+3/(t_0t_2u)+u/(t_0^2t_1t_2)+t_1/(t_0u^4)+t_1/(t_2u^4)+1/u^4+1/(t_0u^3)+t_1/(t_0t_2u^3)+1/(t_2u^3)+1/(t_0t_2u^2)$

Poincaré Polynomial (factored) $(t_0^3t_1^3t_2+2t_0^3t_1^2t_2u+2t_0^2t_1^3t_2u+2t_0^3t_1^2u^2+2t_0^2t_1^3u^2+2t_0^2t_1^2t_2u^2+t_0^2t_1t_2^2u^2+t_0t_1^2t_2^2u^2+t_0t_1t_2^3u^2+t_0^3t_1u^3+t_0^2t_1^2u^3+t_0t_1^3u^3+2t_0^2t_1t_2u^3+t_0t_1^2t_2u^3+t_0t_1t_2^2u^3+t_1t_2^3u^3+t_0t_1^2u^4+3t_0t_1t_2u^4+t_0t_2^2u^4+t_1t_2^2u^4+t_0t_1u^5+t_1t_2u^5+2t_2^2u^5+t_2u^6+t_0^2t_1^2t_2u+t_0^2t_1t_2^2u+t_0t_1^2t_2^2u+t_0^2t_1t_2u^2+t_0t_1^2t_2u^2+t_0t_1t_2^2u^2+t_0t_1t_2u^3)/(t_0^2t_1t_2^2u^5)$

Grading Ranges

$$-5 < UGrading < 1$$

$$-6 < VGrading < 0$$

$$-4 < AGrading < 2$$

$$-2 < AGrading0 < 1$$

$$-1 < AGrading1 < 2$$

$$-2 < AGrading2 < 1$$

$L6n1_{01}$

σ_X	$[7,1,2,4,3,5,6]$
σ_O	$[3,4,5,6,7,2,1]$
GFC^∞	
Number of Generators	288
Number of Edges	5194

Poincaré Polynomial $t_0t_1^{(1/2)}/t_2^{(5/2)} + 5t_1^{(3/2)}/t_2^{(5/2)} + 3t_1^{(5/2)}/(t_0t_2^{(5/2)}) + t_1^{(7/2)}/(t_0^2t_2^{(5/2)}) + 2t_0/(t_1^{(1/2)}t_2^{(3/2)}) + 9t_1^{(1/2)}/t_2^{(3/2)} + 13t_1^{(3/2)}/(t_0t_2^{(3/2)}) + 4t_1^{(5/2)}/(t_0^2t_2^{(3/2)}) + 2/(t_1^{(1/2)}t_2^{(1/2)}) + 5t_1^{(1/2)}/(t_0t_2^{(1/2)}) + t_1^{(3/2)}/(t_0^2t_2^{(1/2)}) + t_0^2t_1^{(5/2)}t_2^{(1/2)}/u^6 + t_0t_1^{(7/2)}/(t_2^{(1/2)}u^5) + 2t_0t_1^{(5/2)}t_2^{(1/2)}/u^5 + 4t_1^{(7/2)}t_2^{(1/2)}/u^5 + t_0^3t_1^{(3/2)}/(t_2^{(3/2)}u^4) + t_0^2t_1^{(5/2)}/(t_2^{(3/2)}u^4) + t_0t_1^{(7/2)}/(t_2^{(3/2)}u^4) + 2t_0^2t_1^{(3/2)}/(t_2^{(1/2)}u^4) + 2t_0t_1^{(5/2)}/(t_2^{(1/2)}u^4) + 4t_1^{(7/2)}/(t_2^{(1/2)}u^4) + 3t_0t_1^{(3/2)}t_2^{(1/2)}/u^4 + 5t_1^{(5/2)}t_2^{(1/2)}/u^4 + 2t_1^{(7/2)}t_2^{(1/2)}/(t_0u^4) + 3t_0^2t_1^{(3/2)}/(t_2^{(3/2)}u^3) + 4t_1^{(7/2)}/(t_2^{(3/2)}u^3) + 7t_0t_1^{(3/2)}/(t_2^{(1/2)}u^3) + 11t_1^{(5/2)}/(t_2^{(1/2)}u^3) + 4t_1^{(7/2)}/(t_0t_2^{(1/2)}u^3) + 3t_1^{(3/2)}t_2^{(1/2)}/u^3 + 4t_1^{(5/2)}t_2^{(1/2)}/(t_0u^3) + t_1^{(7/2)}t_2^{(1/2)}/(t_0^2u^3) + t_0^2t_1^{(1/2)}/(t_2^{(3/2)}u^2) + 6t_0t_1^{(3/2)}/(t_2^{(3/2)}u^2) + 8t_1^{(5/2)}/(t_2^{(3/2)}u^2) + 2t_1^{(7/2)}/(t_0t_2^{(3/2)}u^2) + 4t_0t_1^{(1/2)}/(t_2^{(1/2)}u^2) + 10t_1^{(3/2)}/(t_2^{(1/2)}u^2) + 9t_1^{(5/2)}/(t_0t_2^{(1/2)}u^2) + 3t_1^{(7/2)}/(t_0^2t_2^{(1/2)}u^2) + t_1^{(1/2)}t_2^{(1/2)}/u^2 + t_1^{(3/2)}t_2^{(1/2)}/(t_0u^2) + t_1^{(5/2)}t_2^{(1/2)}/(t_0^2u^2) + 3t_0t_1^{(3/2)}/(t_2^{(5/2)}u) + t_1^{(5/2)}/(t_2^{(5/2)}u) + 3t_0t_1^{(1/2)}/(t_2^{(3/2)}u) + 8t_1^{(3/2)}/(t_2^{(3/2)}u) + 12t_1^{(5/2)}/(t_0t_2^{(3/2)}u) + 7t_1^{(1/2)}/(t_2^{(1/2)}u) + 8t_1^{(3/2)}/(t_0t_2^{(1/2)}u) + 5t_1^{(5/2)}/(t_0^2t_2^{(1/2)}u) + t_1^{(1/2)}t_2^{(1/2)}/(t_0u) + 4t_1^{(1/2)}u/t_2^{(5/2)} + 11t_1^{(3/2)}u/(t_0t_2^{(5/2)}) + 2t_1^{(5/2)}u/(t_0^2t_2^{(5/2)}) + 7u/(t_1^{(1/2)}t_2^{(3/2)}) + 7t_1^{(1/2)}u/(t_0t_2^{(3/2)}) + 3t_1^{(3/2)}u/(t_0^2t_2^{(3/2)}) + t_1^{(5/2)}u/(t_0^3t_2^{(3/2)}) + u/(t_0t_1^{(1/2)}t_2^{(1/2)}) + t_1^{(1/2)}u/(t_0^2t_2^{(1/2)}) + t_1^{(1/2)}u^2/t_2^{(7/2)} + t_1^{(3/2)}u^2/(t_0t_2^{(7/2)}) + t_1^{(5/2)}u^2/(t_0^2t_2^{(7/2)}) + 2u^2/(t_1^{(1/2)}t_2^{(5/2)}) + 5t_1^{(1/2)}u^2/(t_0t_2^{(5/2)}) + 3t_1^{(3/2)}u^2/(t_0^2t_2^{(5/2)}) + 4u^2/(t_0t_1^{(1/2)}t_2^{(3/2)}) + 3t_1^{(1/2)}u^2/(t_0^2t_2^{(3/2)}) + t_1^{(3/2)}u^2/(t_0^3t_2^{(3/2)}) + u^3/(t_1^{(1/2)}t_2^{(7/2)}) + u^3/(t_1^{(3/2)}t_2^{(5/2)}) + 2u^3/(t_0t_1^{(1/2)}t_2^{(5/2)}) + 2t_1^{(1/2)}u^3/(t_0^2t_2^{(5/2)}) + u^3/(t_0^2t_1^{(1/2)}t_2^{(3/2)}) + u^4/(t_0t_1^{(3/2)}t_2^{(5/2)}) + t_1^{(5/2)}/(t_0^3t_2^{(3/2)}) + t_1^{(7/2)}/(t_2^{(3/2)}u^4) + t_0t_1^{(3/2)}/(t_2^{(3/2)}u^3) + t_1^{(7/2)}/(t_0t_2^{(3/2)}u^3) + t_1^{(5/2)}/(t_0t_2^{(1/2)}u^3) + t_1^{(3/2)}t_2^{(1/2)}/(t_0u^3) + t_0t_1^{(1/2)}/(t_2^{(3/2)}u^2) +$

$t_1^{(3/2)}/(t_2^{(3/2)}u^2)+3t_1^{(5/2)}/(t_0t_2^{(3/2)}u^2)+t_1^{(3/2)}/(t_0t_2^{(1/2)}u^2)+t_1^{(3/2)}/(t_2^{(5/2)}u)+t_1^{(1/2)}/(t_2^{(3/2)}u)+2t_1^{(3/2)}/(t_0t_2^{(3/2)}u)$

Poincaré Polynomial (factored) $(t_0^5t_1^4t_2^4+t_0^4t_1^5t_2^3u+2t_0^4t_1^4t_2^4u+4t_0^3t_1^5t_2^4u+t_0^6t_1^3t_2^2u^2+t_0^5t_1^4t_2^2u^2+t_0^4t_1^5t_2^2u^2+2t_0^5t_1^3t_2^3u^2+2t_0^4t_1^4t_2^3u^2+4t_0^3t_1^5t_2^3u^2+3t_0^4t_1^3t_2^4u^2+5t_0^3t_1^4t_2^4u^2+2t_0^2t_1^5t_2^4u^2+3t_0^5t_1^3t_2^2u^3+4t_0^3t_1^5t_2^2u^3+7t_0^4t_1^3t_2^3u^3+11t_0^3t_1^4t_2^3u^3+4t_0^2t_1^5t_2^3u^3+3t_0^3t_1^3t_2^4u^3+4t_0^2t_1^4t_2^4u^3+t_0t_1^5t_2^4u^3+t_0^5t_1^2t_2^2u^4+6t_0^4t_1^3t_2^2u^4+8t_0^3t_1^4t_2^2u^4+2t_0^2t_1^5t_2^2u^4+4t_0^4t_1^2t_2^3u^4+10t_0^3t_1^3t_2^3u^4+9t_0^2t_1^4t_2^3u^4+3t_0t_1^5t_2^3u^4+t_0^3t_1^2t_2^4u^4+t_0^2t_1^3t_2^4u^4+t_0t_1^4t_2^4u^4+3t_0^4t_1^3t_2u^5+t_0^3t_1^4t_2u^5+3t_0^4t_1^2t_2^2u^5+8t_0^3t_1^3t_2^2u^5+12t_0^2t_1^4t_2^2u^5+7t_0^3t_1^2t_2^3u^5+8t_0^2t_1^3t_2^3u^5+5t_0t_1^4t_2^3u^5+t_0^2t_1^2t_2^4u^5+t_0^4t_1^2t_2u^6+5t_0^3t_1^3t_2u^6+3t_0^2t_1^4t_2u^6+t_0t_1^5t_2u^6+2t_0^4t_1t_2^2u^6+9t_0^3t_1^2t_2^2u^6+13t_0^2t_1^3t_2^2u^6+4t_0t_1^4t_2^2u^6+2t_0^3t_1t_2^3u^6+5t_0^2t_1^2t_2^3u^6+t_0t_1^3t_2^3u^6+4t_0^3t_1^2t_2u^7+11t_0^2t_1^3t_2u^7+2t_0t_1^4t_2u^7+7t_0^3t_1t_2^2u^7+7t_0^2t_1^2t_2^2u^7+3t_0t_1^3t_2^2u^7+t_1^4t_2^2u^7+t_0^2t_1t_2^3u^7+t_0t_1^2t_2^3u^7+t_0^3t_1^2u^8+t_0^2t_1^3u^8+t_0t_1^4u^8+2t_0^3t_1t_2u^8+5t_0^2t_1^2t_2u^8+3t_0t_1^3t_2u^8+4t_0^2t_1t_2^2u^8+3t_0t_1^2t_2^2u^8+t_1^3t_2^2u^8+t_0^3t_1u^9+t_0^3t_2u^9+2t_0^2t_1t_2u^9+2t_0t_1^2t_2u^9+t_0t_1t_2^2u^9+t_0^2t_2u^{10}+t_0^3t_1^5t_2^2u^2+t_0^4t_1^3t_2^2u^3+t_0^2t_1^5t_2^2u^3+t_0^2t_1^4t_2^3u^3+t_0^2t_1^3t_2^4u^3+t_0^4t_1^2t_2^2u^4+t_0^3t_1^3t_2^2u^4+3t_0^2t_1^4t_2^2u^4+t_0^2t_1^3t_2^3u^4+t_0^3t_1^3t_2u^5+t_0^3t_1^2t_2^2u^5+2t_0^2t_1^3t_2^2u^5+t_1^4t_2^2u^6)/(t_0^3t_1^{(3/2)}t_2^{(7/2)}u^6)$

Grading Ranges

$$-6 < UGrading < 1$$

$$-4 < VGrading < 4$$

$$-2 < AGrading < 5$$

$$-3 < AGrading0 < 3$$

$$0 < AGrading1 < 7/2$$

$$-5/2 < AGrading2 < 1/2$$

GFC^-

Number of Generators	82
Number of Edges	194

Poincaré Polynomial $t_0^2t_1^{(5/2)}t_2^{(1/2)}/u^6+t_0t_1^{(7/2)}/(t_2^{(1/2)}u^5)+2t_0t_1^{(5/2)}t_2^{(1/2)}/u^5+3t_1^{(7/2)}t_2^{(1/2)}/u^5+t_0^3t_1^{(3/2)}/(t_2^{(3/2)}u^4)+t_0^2t_1^{(5/2)}/(t_2^{(3/2)}u^4)+t_0t_1^{(7/2)}/(t_2^{(3/2)}u^4)+$

$t_0^2 t_1^{(3/2)}/(t_2^{(1/2)}u^4) + 2t_0 t_1^{(5/2)}/(t_2^{(1/2)}u^4) + 4t_1^{(7/2)}/(t_2^{(1/2)}u^4) + t_0 t_1^{(3/2)} t_2^{(1/2)}/u^4 + 4t_1^{(5/2)} t_2^{(1/2)}/u^4 + 2t_1^{(7/2)} t_2^{(1/2)}/(t_0 u^4) + 2t_0^2 t_1^{(3/2)}/(t_2^{(3/2)}u^3) + t_1^{(7/2)}/(t_2^{(3/2)}u^3) + 4t_0 t_1^{(3/2)}/(t_2^{(1/2)}u^3) + 5t_1^{(5/2)}/(t_2^{(1/2)}u^3) + 3t_1^{(7/2)}/(t_0 t_2^{(1/2)}u^3) + 2t_1^{(3/2)} t_2^{(1/2)}/u^3 + 3t_1^{(5/2)} t_2^{(1/2)}/(t_0 u^3) + t_1^{(7/2)} t_2^{(1/2)}/(t_0^2 u^3) + t_0^2 t_1^{(1/2)}/(t_2^{(3/2)}u^2) + 4t_0 t_1^{(3/2)}/(t_2^{(3/2)}u^2) + t_1^{(5/2)}/(t_2^{(3/2)}u^2) + t_1^{(7/2)}/(t_0 t_2^{(3/2)}u^2) + t_0 t_1^{(1/2)}/(t_2^{(1/2)}u^2) + 3t_1^{(3/2)}/(t_2^{(1/2)}u^2) + 4t_1^{(5/2)}/(t_0 t_2^{(1/2)}u^2) + 2t_1^{(7/2)}/(t_0^2 t_2^{(1/2)}u^2) + t_1^{(3/2)}/(t_2^{(3/2)}u) + t_1^{(5/2)}/(t_0 t_2^{(3/2)}u) + 2t_1^{(3/2)}/(t_0 t_2^{(1/2)}u) + 3t_1^{(3/2)}u/(t_0 t_2^{(5/2)}) + t_1^{(1/2)}u/(t_0 t_2^{(3/2)}) + t_1^{(5/2)}/(t_0^3 t_2^{(3/2)}) + t_0 t_1^{(3/2)}/(t_2^{(3/2)}u^3) + t_1^{(7/2)}/(t_0 t_2^{(3/2)}u^3) + t_1^{(3/2)}/(t_2^{(3/2)}u^2) + 3t_1^{(5/2)}/(t_0 t_2^{(3/2)}u^2) + t_1^{(3/2)}/(t_0 t_2^{(1/2)}u^2) + t_1^{(3/2)}/(t_2^{(5/2)}u) + t_1^{(1/2)}/(t_2^{(3/2)}u) + 2t_1^{(3/2)}/(t_0 t_2^{(3/2)}u)$

Poincaré Polynomial (factored) $(t_0^5 t_1^2 t_2^3 + t_0^4 t_1^3 t_2^2 u + 2t_0^4 t_1^2 t_2^3 u + 3t_0^3 t_1^3 t_2^3 u + t_0^6 t_1 t_2 u^2 + t_0^5 t_1^2 t_2 u^2 + t_0^4 t_1^3 t_2 u^2 + t_0^5 t_1 t_2^2 u^2 + 2t_0^4 t_1^2 t_2^2 u^2 + 4t_0^3 t_1^3 t_2^2 u^2 + t_0^4 t_1 t_2^3 u^2 + 4t_0^3 t_1^2 t_2^3 u^2 + 2t_0^2 t_1^3 t_2^3 u^2 + 2t_0^5 t_1 t_2 u^3 + t_0^3 t_1^3 t_2 u^3 + 4t_0^4 t_1 t_2^2 u^3 + 5t_0^3 t_1^2 t_2^2 u^3 + 3t_0^2 t_1^3 t_2^2 u^3 + 2t_0^3 t_1 t_2^3 u^3 + 3t_0^2 t_1^2 t_2^3 u^3 + t_0 t_1^3 t_2^3 u^3 + t_0^5 t_2 u^4 + 4t_0^4 t_1 t_2 u^4 + t_0^3 t_1^2 t_2 u^4 + t_0^2 t_1^3 t_2 u^4 + t_0^4 t_2^2 u^4 + 3t_0^3 t_1 t_2^2 u^4 + 4t_0^2 t_1^2 t_2^2 u^4 + 2t_0 t_1^3 t_2^2 u^4 + t_0^3 t_1 t_2 u^5 + t_0^2 t_1^2 t_2 u^5 + 2t_0^2 t_1 t_2^2 u^5 + 3t_0^2 t_1 u^7 + t_0^2 t_2 u^7 + t_0^4 t_1 t_2 u^3 + t_0^2 t_1^3 t_2 u^3 + t_0^3 t_1 t_2 u^4 + 3t_0^2 t_1^2 t_2 u^4 + t_0^2 t_1 t_2^2 u^4 + t_0^3 t_1 u^5 + t_0^3 t_2 u^5 + 2t_0^2 t_1 t_2 u^5 + t_1^2 t_2 u^6) t_1^{(1/2)}/(t_0^3 t_2^{(5/2)} u^6)$

Grading Ranges

$$-6 < UGrading < 1$$

$$-4 < VGrading < 4$$

$$-2 < AGrading < 5$$

$$-3 < AGrading0 < 3$$

$$0 < AGrading1 < 7/2$$

$$-5/2 < AGrading2 < 1/2$$

$L6n1_{11}$

σ_X	$[2, 6, 1, 5, 3, 4]$
σ_O	$[5, 3, 4, 2, 6, 1]$

GFC^{∞}

Number of Generators	142
Number of Edges	1139

Poincaré Polynomial $2/(t_1^3t_2^2) + 2/(t_0t_1^2t_2^2) + 2/(t_0t_1^3t_2) + 1/(t_0^2t_1^2t_2) + t_2/u^6 + t_0/(t_1u^5) + 2t_2/(t_0u^5)+2t_2/(t_1u^5)+2/u^5+4/(t_0u^4)+2t_0/(t_1^2u^4)+7/(t_1u^4)+2t_0/(t_1t_2u^4)+1/(t_2u^4)+t_2/(t_0^2u^4) + t_2/(t_1^2u^4) + 4t_2/(t_0t_1u^4) + 2/(t_0^2u^3) + t_0/(t_1^3u^3) + 7/(t_1^2u^3) + 9/(t_0t_1u^3) + t_0/(t_1t_2^2u^3)+2/(t_0t_2u^3)+4t_0/(t_1^2t_2u^3)+7/(t_1t_2u^3)+2t_2/(t_0t_1^2u^3)+2t_2/(t_0^2t_1u^3)+2/(t_1^3u^2)+7/(t_0t_1^2u^2)+4/(t_0^2t_1u^2)+2t_0/(t_1^2t_2^2u^2)+2/(t_1t_2^2u^2)+1/(t_0^2t_2u^2)+2t_0/(t_1^3t_2u^2)+10/(t_1^2t_2u^2)+7/(t_0t_1t_2u^2)+t_2/(t_0^2t_1^2u^2)+1/(t_0t_1^3u)+2/(t_0^2t_1^2u)+t_0/(t_1^3t_2^2u)+4/(t_1^2t_2^2u)+1/(t_0t_1t_2^2u)+4/(t_1^3t_2u)+7/(t_0t_1^2t_2u)+2/(t_0^2t_1t_2u)+u/(t_0t_1^3t_2^2)+1/(t_1u^5)+1/(t_1^2u^4)+1/(t_1t_2u^4)+1/(t_0t_1^2u^3)+1/(t_1^2t_2u^3)+1/(t_0t_1t_2u^3)+1/(t_0t_1^2t_2u^2)$

Poincaré Polynomial (factored) $(t_0^2t_1^3t_2^3 + t_0^3t_1^2t_2^2u + 2t_0^2t_1^3t_2^2u + 2t_0^2t_1^2t_2^3u + 2t_0t_1^3t_2^3u + 2t_0^3t_1^2t_2u^2 + t_0^2t_1^3t_2u^2 + 2t_0^3t_1t_2^2u^2 + 7t_0^2t_1^2t_2^2u^2 + 4t_0t_1^3t_2^2u^2 + t_0^2t_1t_2^3u^2 + 4t_0t_1^2t_2^3u^2 + t_1^3t_2^3u^2 + t_0^3t_1^2u^3 + 4t_0^3t_1t_2u^3 + 7t_0^2t_1^2t_2u^3 + 2t_0t_1^3t_2u^3 + t_0^3t_2^2u^3 + 7t_0^2t_1t_2^2u^3 + 9t_0t_1^2t_2^2u^3 + 2t_1^3t_2^2u^3 + 2t_0t_1t_2^3u^3 + 2t_1^2t_2^3u^3 + 2t_0^3t_1u^4 + 2t_0^2t_1^2u^4 + 2t_0^3t_2u^4 + 10t_0^2t_1t_2u^4 + 7t_0t_1^2t_2u^4 + t_1^3t_2u^4 + 2t_0^2t_2^2u^4 + 7t_0t_1t_2^2u^4 + 4t_1^2t_2^2u^4 + t_1t_2^3u^4 + t_0^3u^5 + 4t_0^2t_1u^5 + t_0t_1^2u^5 + 4t_0^2t_2u^5 + 7t_0t_1t_2u^5 + 2t_1^2t_2u^5 + t_0t_2^2u^5 + 2t_1t_2^2u^5 + 2t_0^2u^6 + 2t_0t_1u^6 + 2t_0t_2u^6 + t_1t_2u^6 + t_0u^7 + t_0^2t_1^2t_2^2u + t_0^2t_1^2t_2u^2 + t_0^2t_1t_2^2u^2 + t_0^2t_1t_2u^3 + t_0t_1^2t_2u^3 + t_0t_1t_2^2u^3 + t_0t_1t_2u^4)/(t_0^2t_1^3t_2^2u^6)$

Grading Ranges

$$-5 < UGrading < 1$$

$$-6 < VGrading < 0$$

$$-6 < AGrading < 0$$

$$-2 < AGrading0 < 1$$

$$-3 < AGrading1 < 0$$

$$-2 < AGrading2 < 1$$

GFC^-

Number of Generators	36
Number of Edges	91

Poincaré Polynomial $2/(t_1^3t_2^2)+1/(t_0t_1^3t_2)+1/(t_0u^4)+1/(t_1u^4)+t_0/(t_1t_2u^4)+t_2/(t_0t_1u^4)+2/(t_0^2u^3)+1/(t_0t_1u^3)+2t_2/(t_0^2t_1u^3)+3/(t_0t_1^2u^2)+1/(t_0^2t_2u^2)+t_0/(t_1^3t_2u^2)+1/(t_1^2t_2u^2)+1/(t_0t_1t_2u^2)+t_2/(t_0^2t_1^2u^2)+2/(t_0^2t_1^2u)+2/(t_1^2t_2^2u)+3/(t_1^3t_2u)+1/(t_0^2t_1t_2u)+u/(t_0t_1^3t_2^2)+1/(t_1u^5)+1/(t_1^2u^4)+1/(t_1t_2u^4)+1/(t_0t_1^2u^3)+1/(t_1^2t_2u^3)+1/(t_0t_1t_2u^3)+1/(t_0t_1^2t_2u^2)$

Poincaré Polynomial (factored) $(t_0^3t_1^2t_2u+t_0^2t_1^2t_2^2u+t_0t_1^3t_2^2u+t_0t_1^2t_2^3u+t_0t_1^2t_2^2u^2+2t_1^3t_2^2u^2+2t_1^2t_2^3u^2+t_0^3t_2u^3+t_0^2t_1t_2u^3+t_0t_1^2t_2u^3+t_1^3t_2u^3+3t_0t_1t_2^2u^3+t_1t_2^3u^3+2t_0^2t_1u^4+3t_0^2t_2u^4+t_1^2t_2u^4+2t_1t_2^2u^4+2t_0^2u^5+t_0t_2u^5+t_0u^6+t_0^2t_1^2t_2^2+t_0^2t_1^2t_2u+t_0^2t_1t_2^2u+t_0^2t_1t_2u^2+t_0t_1^2t_2u^2+t_0t_1t_2^2u^2+t_0t_1t_2u^3)/(t_0^2t_1^3t_2^2u^5)$

Grading Ranges

$$-5 < UGrading < 1$$
$$-6 < VGrading < 0$$
$$-6 < AGrading < 0$$
$$-2 < AGrading0 < 1$$
$$-3 < AGrading1 < 0$$
$$-2 < AGrading2 < 1$$

$L7n1_1$

σ_X	$[3, 4, 2, 5, 7, 1, 6]$
σ_O	$[7, 1, 6, 3, 4, 5, 2]$

GFC^∞

Number of Generators	320
Number of Edges	4480

Poincaré Polynomial $15/(t_0^{(7/2)}t_1^2) + 8/(t_0^{(9/2)}t_1) + t_0^{(1/2)}t_1/u^7 + t_0^{(1/2)}/u^6 + 5t_1/(t_0^{(1/2)}u^6) + t_1^2/(t_0^{(3/2)}u^6) + 7/(t_0^{(1/2)}u^5) + t_0^{(1/2)}/(t_1u^5) + 14t_1/(t_0^{(3/2)}u^5) + t_1^2/(t_0^{(5/2)}u^5) + 22/(t_0^{(3/2)}u^4) + 5/(t_0^{(1/2)}t_1u^4) + 18t_1/(t_0^{(5/2)}u^4) + 2t_1^2/(t_0^{(7/2)}u^4) + 34/(t_0^{(5/2)}u^3) + 5/(t_0^{(1/2)}t_1^2u^3) + 20/(t_0^{(3/2)}t_1u^3) + 6t_1/(t_0^{(7/2)}u^3) + t_1^2/(t_0^{(9/2)}u^3) + 18/(t_0^{(7/2)}u^2) + 11/(t_0^{(3/2)}t_1^2u^2) + 35/(t_0^{(5/2)}t_1u^2) + 2t_1/(t_0^{(9/2)}u^2) + 8/(t_0^{(9/2)}u) + 11/(t_0^{(5/2)}t_1^2u) + 28/(t_0^{(7/2)}t_1u) + 6u/(t_0^{(9/2)}t_1^2) + u/(t_0^{(11/2)}t_1) + u^2/(t_0^{(11/2)}t_1^2) + t_1/(t_0^{(3/2)}u^6) + 3/(t_0^{(3/2)}u^5) + 1/(t_0^{(1/2)}t_1u^5) + t_1/(t_0^{(5/2)}u^5) + 5/(t_0^{(5/2)}u^4) + 4/(t_0^{(3/2)}t_1u^4) + t_1/(t_0^{(7/2)}u^4) + 3/(t_0^{(7/2)}u^3) + 7/(t_0^{(5/2)}t_1u^3) + 2/(t_0^{(9/2)}u^2) + 3/(t_0^{(7/2)}t_1u^2) + 1/(t_0^{(7/2)}t_1^2u)$

Poincaré Polynomial (factored) $(t_0^6t_1^3+t_0^6t_1^2u+5t_0^5t_1^3u+t_0^4t_1^4u+t_0^6t_1u^2+7t_0^5t_1^2u^2+14t_0^4t_1^3u^2+t_0^3t_1^4u^2+5t_0^5t_1u^3+22t_0^4t_1^2u^3+18t_0^3t_1^3u^3+2t_0^2t_1^4u^3+5t_0^5u^4+20t_0^4t_1u^4+34t_0^3t_1^2u^4+6t_0^2t_1^3u^4+t_0t_1^4u^4+11t_0^4u^5+35t_0^3t_1u^5+18t_0^2t_1^2u^5+2t_0t_1^3u^5+11t_0^3u^6+28t_0^2t_1u^6+8t_0t_1^2u^6+15t_0^2u^7+8t_0t_1u^7+6t_0u^8+t_1u^8+u^9+t_0^4t_1^3u+t_0^5t_1u^2+3t_0^4t_1^2u^2+t_0^3t_1^3u^2+4t_0^4t_1u^3+5t_0^3t_1^2u^3+t_0^2t_1^3u^3+7t_0^3t_1u^4+3t_0^2t_1^2u^4+3t_0^2t_1u^5+2t_0t_1^2u^5+t_0^2u^6)/(t_0^{(11/2)}t_1^2u^7)$

Grading Ranges

$$-6 < UGrading < 0$$

$$-6 < VGrading < 0$$

$$-11/2 < AGrading < 0$$

$$-9/2 < AGrading0 < 0$$

$$-2 < AGrading1 < 2$$

GFC^-

Number of Generators	84
Number of Edges	325

Poincaré Polynomial $3/(t_0^{(7/2)}t_1^2)+1/(t_0^{(9/2)}t_1)+2/(t_0^{(1/2)}u^5)+3t_1/(t_0^{(3/2)}u^5)+2/(t_0^{(3/2)}u^4)+10t_1/(t_0^{(5/2)}u^4)+t_1^2/(t_0^{(7/2)}u^4)+10/(t_0^{(5/2)}u^3)+3/(t_0^{(3/2)}t_1u^3)+4t_1/(t_0^{(7/2)}u^3)+t_1^2/(t_0^{(9/2)}u^3)+5/(t_0^{(7/2)}u^2)+5/(t_0^{(5/2)}t_1u^2)+t_1/(t_0^{(9/2)}u^2)+3/(t_0^{(9/2)}u)+2/(t_0^{(7/2)}t_1u)+t_1/(t_0^{(3/2)}u^6)+$

$3/(t_0^{(3/2)}u^5) + t_1/(t_0^{(5/2)}u^5) + 4/(t_0^{(5/2)}u^4) + 3/(t_0^{(3/2)}t_1u^4) + t_1/(t_0^{(7/2)}u^4) + 2/(t_0^{(7/2)}u^3) + 7/(t_0^{(5/2)}t_1u^3) + 2/(t_0^{(9/2)}u^2) + 3/(t_0^{(7/2)}t_1u^2) + 1/(t_0^{(7/2)}t_1^2u)$

Poincaré Polynomial (factored) $(2t_0^4t_1^2u + 3t_0^3t_1^3u + 2t_0^3t_1^2u^2 + 10t_0^2t_1^3u^2 + t_0t_1^4u^2 + 3t_0^3t_1u^3 + 10t_0^2t_1^2u^3 + 4t_0t_1^3u^3 + t_1^4u^3 + 5t_0^2t_1u^4 + 5t_0t_1^2u^4 + t_1^3u^4 + 2t_0t_1u^5 + 3t_1^2u^5 + 3t_0u^6 + t_1u^6 + t_0^3t_1^3 + 3t_0^3t_1^2u + t_0^2t_1^3u + 3t_0^3t_1u^2 + 4t_0^2t_1^2u^2 + t_0t_1^3u^2 + 7t_0^2t_1u^3 + 2t_0t_1^2u^3 + 3t_0t_1u^4 + 2t_1^2u^4 + t_0u^5)/(t_0^{(9/2)}t_1^2u^6)$

Grading Ranges

$$-6 < UGrading < 0$$
$$-6 < VGrading < 0$$
$$-11/2 < AGrading < 0$$
$$-9/2 < AGrading0 < 0$$
$$-2 < AGrading1 < 2$$

$L7n2_0$

σ_X	$[4, 7, 2, 1, 3, 5, 6]$
σ_O	$[1, 3, 5, 4, 6, 7, 2]$
GFC^∞	
Number of Generators	516
Number of Edges	8353

Poincaré Polynomial $t_0^2/(t_1^{(5/2)}u) + 2t_0/t_1^{(7/2)} + 3t_0/t_1^{(11/2)} + 18/t_1^{(9/2)} + 15/(t_0t_1^{(7/2)}) + t_0^2t_1^{(1/2)}/u^7 + 6t_0^2/(t_1^{(1/2)}u^6) + 3t_0t_1^{(1/2)}/u^6 + 15t_0^2/(t_1^{(3/2)}u^5) + 18t_0/(t_1^{(1/2)}u^5) + 3t_1^{(1/2)}/u^5 + 21t_0^2/(t_1^{(5/2)}u^4) + 45t_0/(t_1^{(3/2)}u^4) + 18/(t_1^{(1/2)}u^4) + t_1^{(1/2)}/(t_0u^4) + 15t_0^2/(t_1^{(7/2)}u^3) + 61t_0/(t_1^{(5/2)}u^3) + 45/(t_1^{(3/2)}u^3) + 6/(t_0t_1^{(1/2)}u^3) + 6t_0^2/(t_1^{(9/2)}u^2) + 43t_0/(t_1^{(7/2)}u^2) + 60/(t_1^{(5/2)}u^2) + 15/(t_0t_1^{(3/2)}u^2) + t_0^2/(t_1^{(11/2)}u) + 18t_0/(t_1^{(9/2)}u) + 45/(t_1^{(7/2)}u) + 20/(t_0t_1^{(5/2)}u) + 3u/t_1^{(11/2)} + 6u/(t_0t_1^{(9/2)}) + u^2/(t_0t_1^{(11/2)}) + t_0/(t_1^{(5/2)}u^4)$

Poincaré Polynomial (factored) $(t_0^3t_1^3u^6 + 2t_0^2t_1^2u^7 + t_0^3t_1^6 + 6t_0^3t_1^5u + 3t_0^2t_1^6u + 15t_0^3t_1^4u^2 + 18t_0^2t_1^5u^2 + 3t_0t_1^6u^2 + 21t_0^3t_1^3u^3 + 45t_0^2t_1^4u^3 + 18t_0t_1^5u^3 + t_1^6u^3 + 15t_0^3t_1^2u^4 + 61t_0^2t_1^3u^4 + 45t_0t_1^4u^4 + 6t_1^5u^4 + 6t_0^3t_1u^5 + 43t_0^2t_1^2u^5 + 60t_0t_1^3u^5 + 15t_1^4u^5 + t_0^3u^6 + 18t_0^2t_1u^6 + 45t_0t_1^2u^6 + 20t_1^3u^6 + 3t_0^2u^7 + 18t_0t_1u^7 + 15t_1^2u^7 + 3t_0u^8 + 6t_1u^8 + u^9 + t_0^2t_1^3u^3)/(t_0t_1^{(11/2)}u^7)$

Grading Ranges

$$-6 < UGrading < 1$$

$$-6 < VGrading < 1$$

$$-11/2 < AGrading < 3/2$$

$$-1 < AGrading0 < 2$$

$$-11/2 < AGrading1 < 1/2$$

GFC^-

Number of Generators	34
Number of Edges	46

Poincaré Polynomial $2t_0/t_1^{(7/2)} + t_0^2/(t_1^{(1/2)}u^6) + 4t_0^2/(t_1^{(3/2)}u^5) + t_0/(t_1^{(1/2)}u^5) + t_1^{(1/2)}/u^5 + t_0^2/(t_1^{(5/2)}u^4) + 5t_0/(t_1^{(3/2)}u^4) + 3/(t_1^{(1/2)}u^4) + t_0^2/(t_1^{(7/2)}u^3) + 3t_0/(t_1^{(5/2)}u^3) + 3/(t_1^{(3/2)}u^3) + 1/(t_0t_1^{(1/2)}u^3) + 4/(t_1^{(5/2)}u^2) + 1/(t_1^{(7/2)}u) + 1/(t_0t_1^{(5/2)}u) + u/t_1^{(11/2)} + t_0/(t_1^{(5/2)}u^4)$

Poincaré Polynomial (factored) $(2t_0^2t_1^2u^6 + t_0^3t_1^5 + 4t_0^3t_1^4u + t_0^2t_1^5u + t_0t_1^6u + t_0^3t_1^3u^2 + 5t_0^2t_1^4u^2 + 3t_0t_1^5u^2 + t_0^3t_1^2u^3 + 3t_0^2t_1^3u^3 + 3t_0t_1^4u^3 + t_1^5u^3 + 4t_0t_1^3u^4 + t_0t_1^2u^5 + t_1^3u^5 + t_0u^7 + t_0^2t_1^3u^2)/(t_0t_1^{(11/2)}u^6)$

Grading Ranges

$$-6 < UGrading < 1$$

$$-6 < VGrading < 1$$

$$-11/2 < AGrading < 3/2$$

$$-1 < AGrading0 < 2$$

$$-11/2 < AGrading1 < 1/2$$

$L7n2_1$

σ_X	$[3, 5, 1, 6, 7, 2, 4]$
σ_O	$[7, 2, 4, 3, 5, 6, 1]$
GFC^∞	
Number of Generators	512
Number of Edges	9731

Poincaré Polynomial $3t_0/t_1^{(13/2)} + 11/t_1^{(11/2)} + 19/(t_0t_1^{(9/2)}) + 3/(t_0^2t_1^{(7/2)}) + t_0^2/(t_1^{(1/2)}u^7) + 2t_0^2/(t_1^{(3/2)}u^6) + 5t_0/(t_1^{(1/2)}u^6) + 2t_1^{(1/2)}/u^6 + 8t_0^2/(t_1^{(5/2)}u^5) + 19t_0/(t_1^{(3/2)}u^5) + 6/(t_1^{(1/2)}u^5) + 3t_1^{(1/2)}/(t_0u^5) + 10t_0^2/(t_1^{(7/2)}u^4) + 36t_0/(t_1^{(5/2)}u^4) + 33/(t_1^{(3/2)}u^4) + 5/(t_0t_1^{(1/2)}u^4) + 7t_0^2/(t_1^{(9/2)}u^3) + 43t_0/(t_1^{(7/2)}u^3) + 61/(t_1^{(5/2)}u^3) + 13/(t_0t_1^{(3/2)}u^3) + 2/(t_0^2t_1^{(1/2)}u^3) + t_0^2/(t_1^{(11/2)}u^2) + 33t_0/(t_1^{(9/2)}u^2) + 63/(t_1^{(7/2)}u^2) + 29/(t_0t_1^{(5/2)}u^2) + t_0^2/(t_1^{(13/2)}u) + 10t_0/(t_1^{(11/2)}u) + 35/(t_1^{(9/2)}u) + 36/(t_0t_1^{(7/2)}u) + 2/(t_0^2t_1^{(5/2)}u) + 3u/t_1^{(13/2)} + 5u/(t_0t_1^{(11/2)}) + u/(t_0^2t_1^{(9/2)}) + u^2/(t_0t_1^{(13/2)})$

Poincaré Polynomial (factored) $(t_0^4t_1^6 + 2t_0^4t_1^5u + 5t_0^3t_1^6u + 2t_0^2t_1^7u + 8t_0^4t_1^4u^2 + 19t_0^3t_1^5u^2 + 6t_0^2t_1^6u^2 + 3t_0t_1^7u^2 + 10t_0^4t_1^3u^3 + 36t_0^3t_1^4u^3 + 33t_0^2t_1^5u^3 + 5t_0t_1^6u^3 + 7t_0^4t_1^2u^4 + 43t_0^3t_1^3u^4 + 61t_0^2t_1^4u^4 + 13t_0t_1^5u^4 + 2t_1^6u^4 + t_0^4t_1u^5 + 33t_0^3t_1^2u^5 + 63t_0^2t_1^3u^5 + 29t_0t_1^4u^5 + t_0^4u^6 + 10t_0^3t_1u^6 + 35t_0^2t_1^2u^6 + 36t_0t_1^3u^6 + 2t_1^4u^6 + 3t_0^3u^7 + 11t_0^2t_1u^7 + 19t_0t_1^2u^7 + 3t_1^3u^7 + 3t_0^2u^8 + 5t_0t_1u^8 + t_1^2u^8 + t_0u^9)/(t_0^2t_1^{(13/2)}u^7)$

Grading Ranges

$$-6 < UGrading < 1$$

$$-6 < VGrading < 1$$

$$-13/2 < AGrading < 1/2$$

$$-2 < AGrading0 < 2$$

$$-13/2 < AGrading1 < 1/2$$

GFC^-

Number of Generators	30
Number of Edges	0

Poincaré Polynomial $t_1^{(1/2)}/u^6+2t_0^2/(t_1^{(5/2)}u^5)+t_0/(t_1^{(3/2)}u^5)+2/(t_1^{(1/2)}u^5)+2t_1^{(1/2)}/(t_0u^5)+t_0^2/(t_1^{(7/2)}u^4)+7t_0/(t_1^{(5/2)}u^4)+5/(t_1^{(3/2)}u^4)+1/(t_0t_1^{(1/2)}u^4)+t_0/(t_1^{(7/2)}u^3)+1/(t_1^{(5/2)}u^3)+1/(t_0t_1^{(3/2)}u^3)+1/(t_1^{(9/2)}u)+2/(t_0t_1^{(7/2)}u)+1/(t_0^2t_1^{(5/2)}u)+u/t_1^{(13/2)}$

Poincaré Polynomial (factored) $(t_0^2t_1^7+2t_0^4t_1^4u+t_0^3t_1^5u+2t_0^2t_1^6u+2t_0t_1^7u+t_0^4t_1^3u^2+7t_0^3t_1^4u^2+5t_0^2t_1^5u^2+t_0t_1^6u^2+t_0^3t_1^3u^3+t_0^2t_1^4u^3+t_0t_1^5u^3+t_0^2t_1^2u^5+2t_0t_1^3u^5+t_1^4u^5+t_0^2u^7)/(t_0^2t_1^{(13/2)}u^6)$

Grading Ranges

$$-6 < UGrading < 1$$

$$-6 < VGrading < 1$$

$$-13/2 < AGrading < 1/2$$

$$-2 < AGrading0 < 2$$

$$-13/2 < AGrading1 < 1/2$$

BIBLIOGRAPHY

[1] John A. Baldwin and William D. Gillam. "Computations of Heegaard-Floer Knot Homology". In: *Journal of Knot Theory and Its Ramifications* 21.08 (2012), p. 1250075. DOI: 10.1142/S0218216512500757. eprint: https://doi.org/10.1142/S0218216512500757. URL: https://doi.org/10.1142/S0218216512500757.

[2] Wutichai Chongchitmate and Lenhard Ng. "An Atlas of Legendrian Knots". In: *Experimental Mathematics* 22 (Oct. 2010). DOI: 10.1080/10586458.2013.750221.

[3] Peter R. Cromwell. "Embedding knots and links in an open book I: Basic properties". In: *Topology and its Applications* 64.1 (1995), pp. 37–58. ISSN: 0166-8641. DOI: https://doi.org/10.1016/0166-8641(94)00087-J. URL: https://www.sciencedirect.com/science/article/pii/016686419400087J.

[4] Aric Hagberg, Pieter Swart, and Daniel S Chult. *Exploring network structure, dynamics, and function using NetworkX*. tech. rep. Los Alamos National Lab.(LANL), Los Alamos, NM (United States), 2008.

[5] Shelly Harvey and Danielle O'Donnol. "Heegaard Floer homology of spatial graphs". In: *Algebraic & Geometric Topology* 17.3 (July 2017), pp. 1445–1525. DOI: 10.2140/agt.2017.17.1445. URL: https://doi.org/10.2140%2Fagt.2017.17.1445.

[6] Matthew Hedden and Yi Ni. "Khovanov module and the detection of unlinks". In: *Geometry & Topology* 17.5 (Oct. 2013), pp. 3027–3076. DOI: 10.2140/gt.2013.17.3027. URL: https://doi.org/10.2140%2Fgt.2013.17.3027.

[7] Jennifer Hom. *A survey on Heegaard Floer homology and concordance*. 2017. arXiv: 1512.00383 [math.GT]. URL: https://arxiv.org/abs/1512.00383.

[8] Charles Livingston and Allison H. Moore. *KnotInfo: Table of Knot Invariants*. URL: knotinfo.math.indiana.edu. July 2024.

[9] Charles Livingston and Allison H. Moore. *LinkInfo: Table of Link Invariants*. URL: linkinfo.math.indiana.edu. July 2024.

[10] Ciprian Manolescu. *An introduction to knot Floer homology*. 2016. arXiv: 1401.7107 [math.GT].

[11] Ciprian Manolescu, Peter Ozsváth, and Sucharit Sarkar. "A Combinatorial Description of Knot Floer Homology". In: *Annals of Mathematics* 169.2 (2009), pp. 633–660. ISSN: 0003486X. URL: http://www.jstor.org/stable/40345454 (visited on 06/24/2024).

[12] Peter Ozsvath and Zoltan Szabo. "An overview of knot Floer homology". In: (June 2017).

[13] Peter Ozsváth and Zoltán Szabó. "Holomorphic disks and knot invariants". In: *Advances in Mathematics* 186.1 (2004), pp. 58–116. ISSN: 0001-8708. DOI: https://doi.org/10.1016/j.aim.2003.05.001. URL: https://www.sciencedirect.com/science/article/pii/S0001870803002330.

[14] Peter S. Ozsváth, Andr'as I. Stipsicz, and Zoltán Imre Szabó. *Grid Homology for Knots and Links*. 2015.

[15] Jacob Rasmussen. "Floer homology and knot complements". In: *arXiv: Geometric Topology* (2003). URL: https://api.semanticscholar.org/CorpusID:118240720.

[16] Ian Michael Zemke. "TQFT structures in Heegaard Floer homology". PhD thesis. University of California Los Angeles, 2017.

APPENDIX A

PERMUTATION CODE

The following Python code is my implementation of the symmetric group that is used occasionally to support the other programs. The code is included here without exposition and explanation as in the primary sections.

```
import copy
import sys
import math

class perm: #takes a list of integers and makes it a permutation type with standard
↪  permutation
            #group operation. Will copy argument if given a perm instead of a list
            ↪  datatype.

    def __init__(self, lst = [1]): #initialization just loads the list into the
    ↪  self.value
        check_if_valid(lst)
        if type(lst) == perm:
            self.value = lst.value.copy()
        else:
            self.value = lst.copy()

    def __str__(self):
        return str(self.value)

    def __repr__(self):
        return str(self.value)

    def __getitem__(self, i):
        return self.value[i- 1]

    def __setitem__(self, i, val):
        self.value[i- 1] = val

    def __mul__(self, other): #self o other as permutation composition, also loads this
    ↪  into the * notation so sig*phi works
        temp_sig = self.value.copy()
        temp_phi = other.value.copy()
        temp_sig, temp_phi = match_sizes(temp_sig, temp_phi)
        n = len(temp_phi)
```

```
        result = temp_sig.copy()
        for i in range(n):
            result[i] = temp_sig[temp_phi[i] - 1]
        return perm(result)

    def __len__(self):
        return len(self.value)

    def __eq__(self, other): #lets us compare permutations
        if type(other) != perm:
            return False
        if self.value == other.value:
            return True
        else:
            return False

    def copy(self):
        newCopy = perm(self.value)
        return newCopy

    def inverse(self): #Finding sig^- 1
        result = identity_perm(len(self.value))
        for i in range(1,len(self.value)+1):
            result[self[i]] = i
        return result

    def __pow__(self, n): #computes sig**n and loads it into the ** notation
        temp_sig, temp_id = match_sizes(self.value.copy(), [1])
        result = perm(temp_id)
        if n >= 0:
            for i in range(n):
                result = self * result
            return result
        else:
            inv = self.inverse()
            for i in range(-n):
                result = inv * result
            return result

    def size(self):
        return len(self.value)

    def cycles(self):
        #returns a tuple of tuples representing the cycle notation for
```

```
    cycle_collection = [] #the given permutation for example [2, 1, 3, 5, 6, 4] -->
    ↪  ((1, 2)(3)(4,5,6)
    counted = []
    for x in range(self.size()):
        current = x + 1
        current_cycle = []
        while current not in counted:
            current_cycle.append(current)
            counted.append(current)
            current = self.value[current - 1]
        if len(current_cycle) > 0:
            cycle_collection.append(tuple(current_cycle))
    return tuple(cycle_collection)

def reduce_at(self, position): #removes an element that maps to itself and
↪  renumbers the permutation so its consistent
    if self.value[position - 1] == position:
        result = []
        for x in self.value:
            if x < position:
                result.append(x)
            elif x > position:
                result.append(x- 1)
        return perm(result)
    else:
        raise ValueError("Cannot reduce at given value - given value is not fixed
        ↪  under the permutation")

def collapse_at(self, position): #pass this function the i s.t. you want to remove
↪  sigma(i)
    original_value = self.value[position - 1]
    result = []
    reference = self.value.copy()
    for x in reference:
        if x < original_value:
            result.append(x)
        elif x > original_value:
            result.append(x - 1)
    return perm(result)

def widen_at(self, position = 1, value = 1):  #inverse of the reduce function -
↪  widens a permutation by adding i -> i in
    reference = self.value.copy()  #the middle of a permutation and renumbers the
    ↪  inputs and outputs that are
    size = len(reference)+1        # larger than i accordingly.
```

```
        reference.append(size)          # Ex: [1, 3, 4, 6, 2, 5] widen at 4 --> [1, 3,
        ↪  5, 4, 7, 2, 6]
        result = []                     #This operation isn't very natural in
        ↪  permutations but it is in the context of grids
        for i in range(size):
            if i+1 < position:
                if reference[i] < value:
                    result.append(reference[i])
                else:
                    result.append(reference[i] +1)
            elif i+1 > position:
                if reference[i- 1] < value:
                    result.append(reference[i- 1])
                else:
                    result.append(reference[i- 1] +1)
            else:
                result.append(value)
        return perm(result)

def identity_perm(n): #produces perm data type of [1, 2, 3 ... n]
    temp_list = []
    for i in range(n):
        temp_list.append(i+1)
    return perm(temp_list)

def extend_perm(sig, n, isPerm = False): #Extends a permutation to length n - will not
↪  shorten a given permutation
    check_if_valid(sig)                 #accepts lists and perms returning the same
    ↪  type as given
    if type(sig) == list:
        size = len(sig)
        if n <= size:
            if isPerm:
                return perm(sig)
            return sig
        extended_perm = sig
        for i in range(size+1, n+1):
            extended_perm.append(i)
        if isPerm:
            return perm(extended_perm)
        return extended_perm
    else:
        return extend_perm(sig.value, n, True)

def match_sizes(sigma, phi):
```

```
    #extends the sigma and phi as necessary, adding elements mapping to themselves
    #accepts lists and perms returning the same type as given
    check_if_valid(sigma)
    check_if_valid(phi)
    new_sigma = perm(sigma)
    new_phi = perm(phi)
    fin_sigma = extend_perm(new_sigma, len(phi))
    fin_phi = extend_perm(new_phi, len(sigma))

    return (type(sigma)(fin_sigma.value), type(phi)(fin_phi.value))

def check_if_valid(sig): #Running collection of possible errors for given permutation
↪  arguments
    if not ((type(sig) == list) or (type(sig) == perm)): #checks if sig is a list or a
    ↪  perm type and raises a warning if not
        raise ValueError("Neither list nor perm passed to function")
    return


def transposition(x, y, n = 2): #Permutation of [1,2 ... y ... x ... n] just swapping x
↪  and y
    temp_list = []
    for i in range(n):
        temp_list.append(i+1)
    temp_list[x- 1] = y
    temp_list[y- 1] = x
    return perm(temp_list)

def generate_all_transpositions(n):
    temp_list = []
    for x in range(1,n):
        for y in range(x+1,n+1):
            temp_list.append(transposition(x,y,n))
#            print("transposing x = " + str(x) + " and y = " + str(y))
    return temp_list

def generate_all_labeled_transpositions(n):
    temp_list = []
    for x in range(1,n):
        for y in range(x+1,n+1):
            temp_list.append([transposition(x,y,n), [x,y]])
    return temp_list

def generate_sn(n):
    ref = list(range(1,n+1))
```

```
    temp_list = []
    working_x = []
    hold = []
    for x in range(1,n+1):
        temp_list.append([x])
    for i in range(n- 1):
#        print(i)
        for z in range(1,n+1):
            for x in temp_list:
                if z not in x:
                    working_x = x.copy()
                    working_x.append(z)
                    hold.append(working_x)
        temp_list = hold.copy()
        hold = []
    result = []
    for entry in temp_list:
        result.append(perm(entry))
    return result


def full_cycle(n): #permutation of [1, 2, 3, ... n]
    result = []
    for i in range(n- 1):
        result.append(i+2)
    result.append(1)
    return perm(result)

def perm_from_cycle(cycle, given_size = - 1): #takes a TUPLE of TUPLES so make sure to
↪  include commas if either of those tuples has a single element
    size = 0              #currently broken - doesn't handled duplicate instances of an
    ↪  element -- requires reduced cycles
    if given_size == - 1:
        for sub_cycle in cycle:
            for x in sub_cycle:
                if x > size:
                    size = x
    else:
        size = given_size
    temp_sig = []
    for i in range(size):
        temp_sig.append(i+1)
    for sub_cycle in cycle:
        for i in range(len(sub_cycle)):
            temp_sig[sub_cycle[i- 1] - 1] = sub_cycle[i]
```

```
    return perm(temp_sig)
```

APPENDIX B

CODE FOR GFK GENERATION

The following code is discussed in section 2.2. It is used to generate the data for the module.

```
import sys
import pickle
import time
try:
    from .perm import *
except:
    from perm import *
import networkx as nx


#########################################################
#First chunk is code for actually computing the complex
#########################################################

def is_between(target, a, b):

    #Input: integers target, a, b
    #
    #Output: Returns True if target is between a and b False otherwise

    if target > a:
        if target < b:
            return True
    return False

#Note: The next 4 cases and the parent_check function use the following conventions,
↪  where the shaded
#       shaded region is the parent rectangle being checked if it contains the target
#
#Input: Coordinates rect = ((ax, ay), (bx, by)) target = (tx, ty)
#
#Output: True if target is inside shaded region, False otherwise
#

def check_case_{1}(rect, target):

    if (is_between(target[0],  rect[0][0],  rect[1][0]) and is_between(target[1],
    ↪  rect[0][1],  rect[1][1])):
        return True
```

```
    return False


def check_case_{2}(rect, target):

    if ((not is_between(target[0], rect[1][0], rect[0][0])) and is_between(target[1],
    ↪ rect[0][1], rect[1][1])):
        return True
    return False


def check_case_{3}(rect, target):

    if (is_between(target[0], rect[0][0], rect[1][0]) and (not is_between(target[1],
    ↪ rect[1][1], rect[0][1]))):
        return True
    return False


def check_case_{4}(rect, target):

    if ((not is_between(target[0], rect[1][0], rect[0][0])) and (not
    ↪ is_between(target[1], rect[1][1], rect[0][1]))):
        return True
    return False

def parent_check(rect, target):

    #Input: rect = ((ax, ay), (bx, by)) and target (tx, ty)
    #
    #Output: Returns True if target is inside the rectangle, False otherwise.

    ax = rect[0][0]
    ay = rect[0][1]
    bx = rect[1][0]
    by = rect[1][1]

    #This is a belabored switch statement. All of these are possible since grids exist
    ↪ on a
    #torus. There's two possibilities for the x coordinates and two for the y as to
    ↪ which one
    #comes first. This gives us the following 4 cases.
    if ( (ax < bx) and (ay < by) ):

        return check_case_{1}(rect, target)
```

```
    if ( (ax > bx) and (ay < by) ):

        return check_case_{2}(rect, target)

    if ( (ax < bx) and (ay > by) ):

        return check_case_{3}(rect, target)

    if ( (ax > bx) and (ay > by) ):

        return check_case_{4}(rect, target)

    print("invalid rectangle given")


def symbol_coordinates(symbol_perm):

    #Input: List or permutation of symbols (typically X and O) coordinates
    #
    #Output: List of the same size with the cartesian (x,y) coordinates of the symbols
    ↪  if
    #        the lower left corner of the grid is at (1,1) and the alpha and beta
    ↪  curves are
    #        1 unit apart
    #
    #Note: This could be approached centering the corner at (0,0) the choice of putting
    ↪  it at
    #      (1,1) is so that the generator coordinates are exactly at the height of
    ↪  their given
    #      permutation. That is the state [2,1,3...] has the first intersection at
    ↪  (1,2)
    #
    #Example: call symbol_coordinates([1,3,2])


    if type(symbol_perm) == perm:

        temp = symbol_perm.value.copy()

    else:

        temp = symbol_perm.copy()

    for count, element in enumerate(temp):
```

```
        temp[count] = (count + 1.5, element + 0.5)

    return temp


def hv_set_shift(h,v,sigs):

    #Input: h integer, v integer, sigs a collection of permutations
    #
    #Output: The list of all permutations shifted horizontally by h and vertically by
    ↪  v.
    #       this amounts to setting phi = [1, 2, 3, ... , n] and taking each sig in
    ↪  the
    #       list and replacing it with (phi^h) o sig o (phi^v)

    result = []

    if len(sigs) == 0:

        n = 0

    else:

        n = len(sigs[0])

    hshift = full_cycle(n)**(-h)
    vshift = full_cycle(n)**v

    for sig in sigs:

        result.append((vshift*sig)*hshift)

    return result

def deprecated_truncated_sn(n, trunc_length):

    #Input: n integer, trunc_length integer
    #
    #Output: List of lists, each on length trunc_length. One for each sub-sequence
    #       that appears in sn. For example [5,1,6] would be an element for n = 6
    #       and trunc_length = 3, from possibly [5,1,6,3,2,4]

    #We'll make sn then chop off the first trunc_length and save it to result
    pre_result = generate_sn(n)
```

```
    result = []

    for sig in pre_result:

        temp = sig.value.copy()
        trunc_sig = temp[0:trunc_length]

        if trunc_sig not in result:

            result.append(trunc_sig.copy())

    return result


def truncated_sn(n, trunc_length):

    result = [[]]
    hold = []
    symset = list(range(1,n+1))
    while len(result[0]) < trunc_length:
        for sym in symset:
            for res in result:
                if not (sym in res):
                    hold.append(res + [sym])
        result = hold
        hold = []

    return result


def generate_all_states_outside_rectangle(rectangle, n):

    #Input: rectangle = ((a0, b0), (a1, b1)), n integer
    #
    #Output: List
    #
    #This function computes the states as if the rectangle had lower corner
    #(1,1) then shifts the result - keep in mind when seeing lists appending 1's

    #Compute the dimensions of the rectangle - mod n since we're working on a torus
    r_width = (rectangle[1][0] - rectangle[0][0]) % n
    r_height = (rectangle[1][1] - rectangle[0][1]) % n

    if (r_width + r_height) > n:
```

```
    print("rectangle dimensions too large to have non-empty set of states")
    return []

pre_result = []

#region a is the one above the rectangle and b is the columns after that
#then s_a is the collection of intersection choices for a grid state x
#that keeps the rectangle empty
sa = truncated_sn(n - r_height - 1, r_width - 1)
sb = generate_sn(n - abs(r_width) - 1)
#sanity check: 2 intersections used by corners, rw - 1 for a and n - rw - 1 for b
↪  means 2 + a + b = n

#This case should only happen when the rectangle prevents placing any points in
#region a (above it)
if sa == []:

    #whats left holds onto the symbols yet to be used
    whats_left = []
    for i in range(2,r_height+1):

        whats_left.append(i)

    for i in range(r_height+1,n+1):

        whats_left.append(i)

    for psi in sb:

        curr_state=[]
        curr_state.append(1)
        curr_state = [1+r_height] + curr_state

        for i in range(n-abs(r_width) - 1):

            curr_state.append(whats_left[psi[i] - 1])

        pcurr_state = perm(curr_state.copy())
        pre_result.append(pcurr_state)

#Case for the rest of the rectangles
for sig in sa:

    #we're lifting this as a list rather than a permutation because
    #we need to shift them all up by the height - making it no longer a perm
```

```
    x = sig.copy()

    #loop lifts the symbols above the rectangle into region a
    for count, position in enumerate(x):

        x[count] = position + r_height + 1

    #whats left holds onto the symbols yet to be used
    whats_left = []

    for i in range(2,abs(r_height)+1):

        whats_left.append(i)

    for i in range(abs(r_height) + 2, n + 1):

        if i not in x:

            whats_left.append(i)

    if sb == []:

        curr_state = x.copy()
        curr_state.append(1)
        curr_state = [1+r_height] + curr_state
        pcurr_state = perm(curr_state.copy())
        pre_result.append(pcurr_state)

    for psi in sb:

        curr_state=x.copy()
        curr_state.append(1)
        curr_state = [1+r_height] + curr_state

        for i in range(n-abs(r_width) - 1):

            curr_state.append(whats_left[psi[i] - 1])

        pcurr_state = perm(curr_state.copy())
        pre_result.append(pcurr_state)

#States need to all be shifted unless the rectangle is actually based at (1, 1)
if not ((rectangle[0][0] == 1) and (rectangle[0][1] == 1)):

    raw_result = hv_set_shift(rectangle[0][0] - 1, rectangle[0][1] - 1, pre_result)
```

```
    else:

        raw_result = pre_result

    result = []

    rect_pairer = transposition(rectangle[0][0], rectangle[1][0], n)

    #Up to now the raw_results holds the states associated with base of the rectangle
    #this loop takes all those and pairs them with the connected state
    for sig in raw_result:
        result.append([sig*rect_pairer,sig])

    if result == []:

        return None

    return result


def zero_list(n):

    #Input: Integer n
    #
    #Output: List of length n with all entries zero --- ex: zero_list(5) = [0, 0, 0, 0,
    ↪  0]

    result = []

    for i in range(n): result.append(0)

    return result

def count_symbols(n, rect, symbol_perm):

    #Input: n integer, rect rectangle ((a0, b0), (a1, b1)), symbol_perm a permutation
    ↪  or list
    #
    #Output: List with 1/0 for symbol in/out of connecting rectangle
    #
    #Iterates through each symbol symbol_perm(i) and marking a corresponding list
    ↪  entry
    #to 1 if the symbol is present
```

```
    temp = zero_list(n)
    usable_sym = symbol_coordinates(symbol_perm)
    #Moving the coordinates to the actual heights rather than the discrete style
    #lists

    for i in range(n):

        if parent_check(rect, usable_sym[i]):

            temp[i] = 1

    return temp

def generate_all_rectangle_sizes(n):

    #Input: n integer
    #
    #Output: All possible sizes of allowable rectangles
    #
    #Any sizes outside of this are too large to not contain a generator's basepoint
    #not to be confused with z/w's.

    result = []

    for width in range(1,n):

        for height in range(1,n+1-width):

            result.append((width,height))

    return result

def generate_all_rectangles(n):

    #Input: n integer
    #
    #Output: All possible connecting rectangles for a grid of size n in a list
    #        format

    sizes = generate_all_rectangle_sizes(n)
    rects = []

    for size in sizes:

        for i in range(0,n):
```

```
            for j in range(0,n):

                x = (i % n + 1,j % n + 1)
                y = ((i+size[0]) % n + 1,(j+size[1]) % n + 1)
                rects.append((x,y))

    return rects

def generate_all_edges(n, symbols):

    #Input: n an integer, symbols, a collection of lists of length n denoting the
    ↪  placement of the symbols. For knots and
    #       links this should be a pair of two lists, or permutations.
    #
    #Output: Returns a list 3 layers deep containing all the edge information of
    ↪  CFKinf
    #
    #Call generate_all_edges(5,[[5, 1, 2, 3, 4],  [2, 3, 4, 5, 1]]) as an example

    pre_diff = generate_all_rectangles(n)
    symbol_count = len(symbols) #This should always be 2 for knots and links.
    place_holder = []
    z_list = zero_list(n)

    for i in range(symbol_count):

        place_holder.append(z_list.copy())

    unweighted_diff = []

    for rect in pre_diff:

        candidates = generate_all_states_outside_rectangle(rect, n)
        #each candidate is a collection of points differing on the rectangle given
        #so each candidate represents an edge from rect[0] = (a0, b0) to rect[1] = (a1,
        ↪  b1)
        #(filling in the rest of the necessary coordinates with the candidate)

        if candidates is not None:

            for count, candidate in enumerate(candidates):

                candidates[count] = [candidate.copy(), rect, place_holder.copy()]
                #^Replacing the candidate with a clean copy, a note of its connecting
```

```
                #rectangle and initialize the edge weight to zero

            unweighted_diff = unweighted_diff + candidates
            #appending the list of these candidates to our unweighted differential

    for count, symbol in enumerate(symbols):
    #This should again always be a pair in case of knots and links

        for i in range(len(unweighted_diff)):
        #here we replace the previous zero weight after counting the symbols

            unweighted_diff[i][2][count] = count_symbols(n, unweighted_diff[i][1],
            ↪  symbol)

    return unweighted_diff

def imp_from_pickle(filename = 'PickleDefault.pickle'):

# Imports pickle file and returns the object. Will import from DefaultPickleComp if no
↪  name is provided

    if filename == 'DefaultPickleComp':
        print('No name provided for import - importing from DefaultPickleComp')


    try:
        file = open(filename,'rb')
        print("file opened")
    except:
        print('Ran into an error: Make sure you\'ve exported to the file you\'re
        ↪  trying to import from')
    stuff = pickle.load(file)
    file.close()
    print('file closed')
    return stuff


def pickle_it(comp, filename = "PickleDefault.pickle"):

    #Input: comp is any data (Usually a complex as a networkx type graph) and str
    ↪  filename
    #Exports the complex as a binary file using the pickle module, just shorthand

    file = open(filename, 'wb')
    pickle.dump(comp, file)
```

```
    file.close()

    return

def build_cinf(symbols):

    #Input: symbols a list of two lists, [sigx, sigo]
    #
    #Output: g a networkx directed graph

    xlist = symbols[0]
    olist = symbols[1]
    size = len(xlist)

#     if type(symbols) == grid:

#         xlist = symbols.sig_x
#         olist = symbols.sig_o

    comp = generate_all_edges(size, [xlist,olist])
    g = nx.DiGraph()
#     nx.set_edge_attributes(g, {'diffweight':[]})
    for ele in comp:

        if not g.has_edge(str(ele[0][0]),str(ele[0][1])):

            g.add_edge(str(ele[0][0]),str(ele[0][1]), diffweight = [])

        g[str(ele[0][0])][str(ele[0][1])]['diffweight'].append((ele[2][0] +
        ↪  ele[2][1]))

    return g


def pickle_cinf(gknot, filename = 'DefaultPickleComp'):

    #Input: Grid or pair of x and o coordinates as list or tuple
    #
    #The function takes the pair calls for the CFK complex to be constructed, loads
    ↪  the
    #result into a networkx directed-graph then calls the function to pickle the
    ↪  resulting
    #graph structure. This is intended to pass to the sage processing from here.

    if filename == 'DefaultPickleComp':
```

```
        print('No name provided for pickling - saving to DefaultPickleComp')

#     if type(gknot) == grid:

#         size = gknot.size
#         xlist = list(gknot.sig_x)
#         olist = list(gknot.sig_o)

    if (type(gknot) == list) or (type(gknot) == tuple):

        size = len(gknot[0])
        xlist = gknot[0]
        olist = gknot[1]


    comp = generate_all_edges(size, [xlist,olist])
    g = nx.DiGraph()
    for ele in comp:

        g.add_edge(str(ele[0][0]),str(ele[0][1]), diffweight = (ele[2][0] + ele[2][1]))

    pickle_it((g, size, gknot), filename)

    return


##########################################
#Code in this section is for getting the user input
##########################################

def get_integer_bounded(n):

    #Input: integer n
    #
    #Output: An integer input from the user between 1 and n

    while True:

        num = input()

        try:

            val = int(num)
            if (val < n + 1) and (val > 0):
                return val
```

```
            print("Your integer isn't between 1 and " + str(n))

        except ValueError:

            print("Please only enter an integer.")

def user_sig(n):

    #Input: integer n
    #
    #Output: List version of the n entries from the user. Check if valid can be
    ↪  strengthened
    #        in perm.py, as it stands it won't catch any errors in this function

    result = []
    print("input the coordinates, returning after each entry")

    for i in range(n):

        val = get_integer_bounded(n)
        result.append(int(val))

    if check_if_valid(result):

        return result

    else:

        print("Invalid sig entered")
        user_sig(n)


def correct_prompt():

    #Input: None
    #
    #Output: True if user confirms, False if user specifies, loops until one of the two
    ↪  is given

    while True:

        ans = input("Is this correct? (Y/N)")
        if (ans == 'Y') or (ans == 'y'): return True
        elif (ans == 'N') or (ans == 'n'): return False
```

```
        else: print("You didn't enter Y or N, please only enter one of these
        ↪  options.")


def prompt_for_link():

    #Input: None
    #
    #Output: List of two lists representing sigx and sigo

    sigx = []
    check = False
    sigo = []
    n = int(input("Enter grid size: "))

    while True:

        print('Grid size = ' + str(n))
        ans = correct_prompt()
        if ans: break
        else: n = int(input('Enter grid size: '))

    print('First enter sigx.')
    sigx = user_sig(n)

    while check != True:

        print("sigx = " + str(sigx))
        ans = correct_prompt()
        if ans: break

        else: sigx = user_sig(n)

    print('Now enter sigo')
    sigo = user_sig(n)

    while check != True:

        print("sigo= " + str(sigo))
        ans = correct_prompt()
        if ans: break

        else: sigo = user_sig(n)

    return [sigx, sigo]
```

```
########################################
#Function call if ran as script
########################################

# def main():

#     user_syms = prompt_for_link()
#     fname = str(input('What would you like the pickle (output) to be named: '))
#     pickle_cinf(user_syms, fname)

# if __name__ == '__main__':

#     main()
```

APPENDIX C

CODE FOR COMPLEX REDUCTION

The code here is for grading and reducing the complex after it is produced by B. The code is included here without exposition and explanation as in the primary sections.

```
import itertools as itools
import networkx as nx
import csv
# import ast
import numpy as np
from scipy import sparse
import matplotlib.pyplot as plt
import multiprocessing as mp
#from sage.graphs.graph_decompositions.graph_products import is_cartesian_product
import GFKTools as gfk
from GridPermutations import *
import time
import pickle
import perm as pr
from multiprocessing.managers import BaseManager
import random as rd


TIMERS = True
PROCESSOR_COUNT = 12
OUTPUTDIRECTORY = 'Outputs/'
PRINT_PROGRESS = True

class MyManager(BaseManager):

    # Necessary class definition for parallel processing

    pass

class grid_complex:

    # This is the data type that holds all the information and most of the functions
    ↪  and methods necessary
    # to produce and manipulate the graded complex.

    def __init__(self, directed_graph, rng, sigx = None, sigo = None):
```

```
# Initializing and setting default values. sigx and sigo should generally be
↪  provided - fail safes are included
# however if they're ever used then only the relative grading of the final object
↪  will be correct

    if type(directed_graph) != nx.DiGraph:
        raise("!This data type only supports networkx digraphs!")

    self.comp = directed_graph
    self.ring = rng
    self.min_gradings = {}
    self.max_gradings = {}
    self.max_grading_changes = {}
    self.sigx = sigx
    self.sigo = sigo
    self.set_to_minus = False
    self.set_to_tilde = False

    # From here the values necessary for the surgered manifold gradings are mapped
    ↪  out

    if (sigx != None) and (sigo != None):
        self.size = len(sigx)
        self.components = link_components(sigx, sigo)
        for i in range(len(self.components)):

            key = f'AGrading{i}'
            self.min_gradings[key] = 0
            self.max_gradings[key] = 0
            self.max_grading_changes[key] = 0
            key = f'UGrading{i}'
            self.min_gradings[key] = 0
            self.max_gradings[key] = 0
            self.max_grading_changes[key] = 0
            key = f'VGrading{i}'
            self.min_gradings[key] = 0
            self.max_gradings[key] = 0
            self.max_grading_changes[key] = 0
    else:

    # This is included in case the methods in the class are useful to another
    ↪  complex being loaded in

        self.components = None
```

```
def __repr__(self):

    # If the object is called it will return the underlying digraph
    return self.comp

def subcomplex(self, subgraph):

    # This is essentially just the subgraph - may not be an actual subcomplex if
    ↪  poor choice of vertices/edges are made
    sub_copy = subgraph.copy()
    result = grid_complex(sub_copy, self.ring)


def copy(self):

    # Adds copy functionality like the copy module
    if self.sigx == None:
        new_copy = grid_complex(self.comp.copy(), self.ring)
    else:
        new_copy = grid_complex(self.comp.copy(), self.ring, self.sigx.copy(),
        ↪  self.sigo.copy())
    return new_copy

def grid(self):
    # Adding functionality to return the original grid that produced the complex
    return [self.sigx, self.sigo]


def graph(self):
    return self.comp

def ring(self):
    return self.ring



def to_hat(self):

    # Substitutes 0 for all the U and V variables in the complex

    print("setting Ui's and Vi's = 0")
    gens = self.ring.gens()
    size = len(gens)/2
```

```
    for edge in self.comp.edges():

        for i in range(2*size):

            src = edge[0]
            tar = edge[1]
            self.comp[src][tar]['diffweight'] =
            ↪  self.comp[src][tar]['diffweight'].subs({gens[i]:0})

    return


def to_minus(self):

    # Substitutes U0 for all the Ui and 1 for Vi

    self.set_to_minus = True
    print("normalizing Ui's and setting Vi = 1")
    gens = self.ring.gens()
    size = len(gens)/2
    for edge in self.comp.edges():

        for component in self.components:
            for i in component:

                # if i == component[0]: continue
                setting_var = component[0] - 1
                src = edge[0]
                tar = edge[1]
                self.comp[src][tar]['diffweight'] =
                ↪  self.comp[src][tar]['diffweight'].subs({gens[i -
                ↪  1]:gens[setting_var]})
                self.comp[src][tar]['diffweight'] =
                ↪  self.comp[src][tar]['diffweight'].subs({gens[i+size- 1]:1})

    self.remove_zeros()
    return


def to_tilde(self, overwrite = True):

    # Converts the complex to the tilde flavor by setting all the U's and V's to 0
    # overwrite option determines whether to apply it to the complex or to make a
    ↪  copy and apply
    # the changes there.
```

```
    if overwrite == False:

        replacement = self.copy()
        return replacement.to_tilde()

    self.set_to_tilde = True
    print("Setting all U's and V's to 0")
    gens = self.ring.gens()
    size = len(gens)/2

    for edge in self.comp.edges():

        for i in range(2*size):

            src = edge[0]
            tar = edge[1]
            self.comp[src][tar]['diffweight'] =
            ↪  self.comp[src][tar]['diffweight'].subs({gens[i]:0})

    self.remove_zeros()

    return self

def link_normalize(self):

    # Substitutes Ucomp for all the Ui associated to that component

    gens = self.ring.gens()
    size = len(gens)/2
    for edge in self.comp.edges():

        for component in self.components:
            for i in component:

                if i == component[0]: continue
                setting_var = component[0] - 1
                src = edge[0]
                tar = edge[1]
                self.comp[src][tar]['diffweight'] =
                ↪  self.comp[src][tar]['diffweight'].subs({gens[i -
                ↪  1]:gens[setting_var]})
                self.comp[src][tar]['diffweight'] =
                ↪  self.comp[src][tar]['diffweight'].subs({gens[i+size-
                ↪  1]:gens[size + setting_var]})
```

```
        self.remove_zeros()
        return


#     def normalize(self):
#     #Substitutes U0 for all the Ui and V0 for all Vi

#         gens = self.field.gens()
#         size = len(gens)/2
#         for edge in self.comp.edges():

#             for i in range(size):

#                 src = edge[0]
#                 tar = edge[1]
#                 self.comp[src][tar]['diffweight'] =
↪  self.comp[src][tar]['diffweight'].subs({gens[i]:gens[0]})
#                 self.comp[src][tar]['diffweight'] =
↪  self.comp[src][tar]['diffweight'].subs({gens[i+size]:gens[size]})

#         self.remove_zeros()
#         return

    def remove_zeros(self):

    # Searches the complex for edges with weight 0 and removes the edge

        elist = list(self.comp.edges())
        for x,y in elist:

            if self.comp[x][y]['diffweight'] == 0:

                self.comp.remove_edge(x,y)

        return


    def split_by_grading(self, partition_list, key):

        # Unnecessary in current version - different approach to parallelization
        # partition_list is expected to be of the form matching the partition
        ↪  function's output.

        result = []
```

```
        for data in partition_list:
            gens = [vert for vert in self.comp.nodes() if
            ↪  ((self.comp.nodes()[vert][key] >= data[0]) and
            ↪  (self.comp.nodes()[vert][key] <= data[4])) ]
            subg = self.comp.subgraph(gens)
            result.append(subg)

        return result


    def graph_red_search(self, started = False, timerstart = None):

        # searches through a cfk inf complex for reducible edges and calling
        # the reduction function to eliminate the pair according to the reduction
        ↪  algorithm

        if not started:
            timerstart = time.time()
            print("Reducing complex...")

        print(len([source for source, target, weight in self.comp.edges(data =
        ↪  'diffweight') if weight == 1]))
        while True:
#             count = (count + 1)%
            try:
                red_target = next((source, target) for source, target, weight in
                ↪  self.comp.edges(data = 'diffweight') if weight == 1)
#                 print(self.comp.edges[red_target])
                self.graph_reduction(red_target[0],  red_target[1])
                continue
            except:
                ("StopIteration")
            break

        timerstop = time.time()
        # print('Time to reduce complex: ' + str(timerstop - timerstart))

        return

    def graph_reduction(self, key, target):

        # Deletes edge specified from graph_red_search and adds in edges according to
        ↪  the
        # reduction algorithm
```

```
    for x in self.comp.predecessors(target):

        if x == key: continue
        for y in self.comp.successors(key):

            if y == target: continue
            x_weight = self.comp[x][target]['diffweight']
            y_weight = self.comp[key][y]['diffweight']
            red_weight = x_weight * y_weight
            if self.comp.has_edge(x,y):
                old_weight = self.comp[x][y]['diffweight']
                red_weight = red_weight + old_weight
            self.comp.add_edge(x,y,diffweight=red_weight)

    self.comp.remove_node(key)
    self.comp.remove_node(target)
    return


def minus_reduction(self, overwrite = True):

    # Reduces the complex but only reducing edges between vertices that are in the
    ↪  same Alexander gradings
    # Note: This will not overwrite (regardless of argument) if the complex hasn't
    ↪  been converted to the minus flavor. Instead, it
    # will make a copy, do the reduction there, and return the new complex

    if self.set_to_minus == False:
        replacement = self.copy()
        replacement.to_minus()
        replacement.minus_reduction(True)

    if overwrite == False:
        replacement = self.copy()
        replacement.minus_reduction(True)

    while True:

        starting_edge_count = len([source for source, target, weight in
        ↪  self.comp.edges(data = 'diffweight') if (weight == 1 and
        ↪  alexander_grading_equivalent(self.comp, source, target,
        ↪  len(self.components)))])

        try:
```

```
            source, target = next((source, target) for source, target, weight in
            ↪  self.comp.edges(data = 'diffweight') if (weight == 1 and
            ↪  alexander_grading_equivalent(self.comp, source, target,
            ↪  len(self.components))))
            print("my alexander function returned " +
            ↪  str(alexander_grading_equivalent(self.comp, source, target,
            ↪  len(self.components))))
            print(self.comp.nodes()[source]["AGrading0"])
            print(self.comp.nodes()[target]["AGrading0"])
            # print(self.comp.nodes()[source]["AGrading1"])
            self.graph_reduction(source, target)
        except StopIteration:
            pass
        except:
            print("Unexpected Error")

        end_edge_count = len([source for source, target, weight in
        ↪  self.comp.edges(data = 'diffweight') if (weight == 1 and
        ↪  alexander_grading_equivalent(self.comp, source, target,
        ↪  len(self.components)))])

        if starting_edge_count == end_edge_count:

            break

    return self


def grade_link_complex(self):

    # Input: given_graph a networkx directed graph with 'diffweight' attribute on
    ↪  edges
    #        given_field the laurent polynomial field associated to the grid graph
    #        gridX a list representing the vertex to be graded 0 in U V and
    ↪  Alexander gradings
    #
    # Output: given_graph with new attributes on the vertices for U V and Alexander
    ↪  gradings
    #         also an attribute HasBeenGraded as an artifact


    # If the positions of the Xs aren't provided we'll initialize around whatever
    # state happens to appear first in the digraph structure - This will mean the
    ↪  complex's absolute grading will be off
```

```
        if self.sigx == None:

            gridX = list(self.comp.nodes())[0]

        else:

            gridX = self.sigx

        if self.sigo == None:

            gridO = list(self.comp.nodes())[0]

        else:

            gridO = self.sigo

        gens = self.ring.gens()
        size = len(gens)/2

        print("grading complex...")

        comp_set = len(self.components)

        # Adding an attribute to all nodes to keep track of if they've been assigned
        ↪  gradings
        for i in range(comp_set):
            nx.set_node_attributes(self.comp, False, f"HasBeenGraded{i}")

        # The gradings are relative so we're declaring one to be in U, V, and Alexander
        ↪  grading 0
        # this block initializes those balues
        for i in range(comp_set):
#             self.comp.nodes()[str(gridX)][f'HasBeenGraded{i}'] = True
            self.comp.nodes()[str(gridX)][f'AGrading{i}'] = 0
#             self.comp.nodes()[str(gridX)][f'UGrading{i}'] = 0
#             self.comp.nodes()[str(gridX)][f'VGrading{i}'] = 0

        if TIMERS: timerstart = time.time()

        # Built in function to find a spanning tree
        #span = nx.algorithms.tree.branchings.greedy_branching(given_graph)

        tree = nx.algorithms.minimum_spanning_tree( self.comp.to_undirected()  )
        eds = set(tree.edges())  # optimization
        spanset = []
```

```
for edge in eds:

    if edge in self.comp.edges():
        spanset.append(edge)

    else:
        spanset.append((edge[1], edge[0]))

span = self.comp.edge_subgraph(spanset)

if TIMERS:

    timerstop = time.time()
    print("Time to find arborescence:" + str(timerstop - timerstart))

# Bit of baseball terminology for the following nested loops, the active data
↪  is essentially at bat, the list we're working
# through is called on_deck, and then we're building up the follow up as
↪  in_the_hole which will turn into
# on deck on the following loop
#
# On deck holds the edges to be iterated through
on_deck = [str(gridX)]

# In the hole holds the ones to be iterated through once on_deck is cleared
in_the_hole = []

if TIMERS: timerstart = time.time()


comp_count = len(self.components)

# Grading Loops Start:
####################

self.componentwise_relative_grading_loop("UGrading", gridX,
↪  self.virtual_U_gradings_succ, self.virtual_U_gradings_pred, span,
↪  comp_count)
self.componentwise_relative_grading_loop("VGrading", gridX,
↪  self.virtual_V_gradings_succ, self.virtual_V_gradings_pred, span,
↪  comp_count)
self.relative_grading_loop("UGrading", gridX, self.maslov_U_succ,
↪  self.maslov_U_pred, span, comp_count)
```

```
        self.relative_grading_loop("VGrading", gridX, self.maslov_V_succ,
        ↪  self.maslov_V_pred, span, comp_count)

        ####################
        # Grading Loops End

        for vert in self.comp.nodes():
            self.comp.nodes()[vert]['AGrading'] = 0
            for i in range(len(self.components)):
                stab_count = len(self.components[i])
                self.comp.nodes()[vert][f'AGrading{i}'] =
                ↪  (1/2)*(self.comp.nodes()[vert][f'VGrading{i}'] -
                ↪  self.comp.nodes()[vert][f'UGrading{i}']) -(1/2)*(stab_count - 1)
                self.comp.nodes()[vert]['AGrading'] +=
                ↪  self.comp.nodes()[vert][f'AGrading{i}']
        if TIMERS:

            timerstop = time.time()
            print('Time to grade complex (given arborescence): ' + str(timerstop -
            ↪  timerstart))

        return

    def gml_export(self, filename = 'PleaseNameMe.gml'):

        # Exports the graph as a gml file which can be opened in a program like Gephi

#         if component_length == - 1:
#             return("!!! Unknown number of components for export !!!")

        component_length = len(self.components)

        if component_length == 0:
            raise("Error finding number of components")

        nxG = self.comp.copy()

        if filename == 'PleaseNameMe.gml':
            print("You didn't name your output! It's been named PleaseNameMe.gml")

        if filename[-4:] != ".gml":

            return self.gml_export(filename + ".gml")
            # filename += ".gml"
```

```
        for x,y in nxG.edges():

            nxG[x][y]['diffweight'] = str(nxG[x][y]['diffweight'])

        for node in nxG.nodes():

            #print(str((nxG.nodes()[node]['UGrading'],  nxG.nodes()[node]['VGrading'],
            ↪  nxG.nodes()[node]['AGrading'])))
            try:
                nxG.nodes[node]['UGrading'] = int(nxG.nodes[node]['UGrading'])
                nxG.nodes[node]['VGrading'] = int(nxG.nodes[node]['VGrading'])
                nxG.nodes[node]['AGrading'] = int(nxG.nodes[node]['AGrading'])
            except:
                nxG.nodes[node]['UGrading'] = int(-99)
                nxG.nodes[node]['VGrading'] = int(-99)
                nxG.nodes[node]['AGrading'] = int(-99)
            for i in range(component_length):
                try:
                    nxG.nodes[node][f'AGrading{i}'] =
                    ↪  int(nxG.nodes[node][f'AGrading{i}'])
                    nxG.nodes[node][f'UGrading{i}'] =
                    ↪  int(nxG.nodes[node][f'UGrading{i}'])
                    nxG.nodes[node][f'VGrading{i}'] =
                    ↪  int(nxG.nodes[node][f'VGrading{i}'])
                except:
                    nxG.nodes[node][f'AGrading{i}'] = int(-99)
                    nxG.nodes[node][f'UGrading{i}'] = int(-99)
                    nxG.nodes[node][f'VGrading{i}'] = int(-99)

        print("writing to " + OUTPUTDIRECTORY + str(filename))
        nx.write_gml(nxG, OUTPUTDIRECTORY + filename)

        return


    def find_grading_ranges(self, key = "AGrading"):

        # Finds the minimum and maximum gradings among the vertices associated with a
        ↪  grading key

        self.min_gradings[key] = 0
        self.max_gradings[key] = 0

        for vert in self.comp.nodes():
```

```
            if self.comp.nodes[vert][key] < self.min_gradings[key]:

                self.min_gradings[key] = self.comp.nodes[vert][key]

            if self.comp.nodes[vert][key] > self.max_gradings[key]:

                self.max_gradings[key] = self.comp.nodes[vert][key]

        return


    def comp_truncate(self, grading_cutoff):

        # Grading cutoff should be a tuple of values, this function will
        # I've only considered this for calling after converting to minus complex

        generators = self.ring.gens()
        for i in range(len(self.components)):
            for vert in self.comp:

                if self.comp.nodes()[vert][f"AGrading{i}"] >= grading_cutoff[i]:
                    self.comp.nodes()[vert][f"AGrading{i}"] += 1
                    self.comp.nodes()[vert][f"UGrading{i}"] += -2

                    for targ in self.comp.successors(vert):

                        self.comp[vert][targ]['diffweight'] =
                        ↪  self.comp[vert][targ]['diffweight']*generators[i]

                    for pred in self.comp.predecessors(vert):

                        self.comp[pred][vert]['diffweight'] =
                        ↪  self.comp[pred][vert]['diffweight']*(generators[i]^(- 1))

        return

    def surgery(self, grading_list = None, target_grading = None):

        # Creates a copy of the graph and truncates it below every combination of
        ↪  Alexander gradings
        # and reduces the resulting complexes then writes them out

        if grading_list == None:

            grading_list = []
```

```
        for i in range(len(self.components)):
            self.find_grading_ranges(f"AGrading{i}")

        for i in range(len(self.components)):
            grading_list.append(self.max_gradings[f"AGrading{i}"])

    if target_grading == None:

        target_grading = []
        for i in range(len(self.components)):
            target_grading.append(self.min_gradings[f"AGrading{i}"])

    print("target gradings = " + str(target_grading))
    print("max gradings = " + str(grading_list))
    if self.set_to_minus == False:

        print("This complex hasn't been converted to minus. Making a copy of the
        ↪  complex and converting it to the minus complex")
        minus_copy = self.copy()
        minus_copy.to_minus()
        minus_copy.surgery(grading_list, target_grading)
        print("uhh didn't expect to be here...")

    grading_ranges = []

    for i in range(len(target_grading)):

        grading_ranges.append(list(range(target_grading[i],  grading_list[i] +1)))

    sub_gradings = itools.product(*grading_ranges)

    for grading in sub_gradings:

        specimen = self.copy()
        specimen.comp_truncate(grading)
        specimen.graph_red_search()
        specimen.remove_zeros()
        specimen.gml_export(str(self.sigx) + str(self.sigo) + "surgery" +
        ↪  str(grading))

    return
```

```
def relative_grading_loop(self, grading_key, base_vertex, fn1, fn2, span = None,
↪  grading_multiplicity = 1):

    # Loop structure around a vertex's neighbors to set gradings based on the
    ↪  functions fn1 and fn2.

    if span == None:

        span = self.comp


    nx.set_node_attributes(self.comp, False, "HasBeenGraded")
    self.comp.nodes()[str(base_vertex)]['HasBeenGraded'] = True
    self.comp.nodes()[str(base_vertex)][f'{grading_key}'] = 0

    on_deck = [str(base_vertex)]

    in_the_hole = []

    while len(on_deck) > 0:

        for vert in on_deck:

            # Every vertex in on_deck should be graded. The loops iterate through
            ↪  the neighbors of each of these
            # vertices, grading them and then adding them to in_the_hole, ignoring
            ↪  vertices that have already been graded.
            #
            # The loop is broken into two halves since we have two flavors of
            ↪  neighbor in a directed graph, successors and
            # predecessors, named accordingly. These flavors differ in relative
            ↪  grading change by a sign.
            for i, component_columns in enumerate(self.components):
                for succ in span.successors(vert):

                    #skip the vertex if its already been graded
                    if self.comp.nodes()[succ]['HasBeenGraded']: continue

                    in_the_hole.append(succ)

                    fn1(succ, vert)

                    self.comp.nodes()[succ]['HasBeenGraded'] = True

                for pred in span.predecessors(vert):
```

```
                    if self.comp.nodes()[pred]['HasBeenGraded']: continue

                    in_the_hole.append(pred)

                    fn2(pred, vert)

                    self.comp.nodes()[pred][f'HasBeenGraded'] = True

        on_deck = in_the_hole
        in_the_hole = []

    return


def componentwise_relative_grading_loop(self, grading_key, base_vertex, fn1, fn2,
↪  span = None, grading_multiplicity = 1):

    # Loop structure around a vertex's neighbors to set gradings based on the
    ↪  functions fn1 and fn2, passing
    # the functions are passed component information as well

    if span == None:

        span = self.comp

    for i in range(grading_multiplicity):

        nx.set_node_attributes(self.comp, False, f"HasBeenGraded{i}")
        self.comp.nodes()[str(base_vertex)][f'HasBeenGraded{i}'] = True
        self.comp.nodes()[str(base_vertex)][f'{grading_key}{i}'] = 0

    on_deck = [str(base_vertex)]

    in_the_hole =[]

    while len(on_deck) > 0:

        for vert in on_deck:

            # Every vertex in on_deck should be graded. The loops iterate through
            ↪  the neighbors of each of these
            # vertices, grading them and then adding them to in_the_hole, ignoring
            ↪  vertices that have already been graded.
            #
```

```
                # The loop is broken into two halves since we have two flavors of
                ↪  neighbor in a directed graph, successors and
                # predecessors, named accordingly. These flavors differ in relative
                ↪  grading change by a sign.
                for i, component_columns in enumerate(self.components):
                    for succ in span.successors(vert):

                        #skip the vertex if its already been graded
                        if self.comp.nodes()[succ][f'HasBeenGraded{i}']: continue

                        in_the_hole.append(succ)

                        fn1(i, succ, vert, component_columns)

                        self.comp.nodes()[succ][f'HasBeenGraded{i}'] = True

                    for pred in span.predecessors(vert):

                        if self.comp.nodes()[pred][f'HasBeenGraded{i}']: continue

                        in_the_hole.append(pred)

                        fn2(i, pred, vert, component_columns)

                        self.comp.nodes()[pred][f'HasBeenGraded{i}'] = True

            on_deck = in_the_hole
            in_the_hole = []

        return
# The following block of function definitions are the supporting functions for the
↪  grading loops
# These are passed as fn1 and fn2 to the grading loops in the grading function
########################################

    def virtual_U_gradings_pred(self, i, pred, vert, component_columns):

        ed_weight = self.comp[pred][vert]['diffweight']

        Upows = link_U_deg(ed_weight, self.ring, component_columns)
        self.comp.nodes()[pred][f'UGrading{i}'] =
        ↪  self.comp.nodes()[vert][f'UGrading{i}'] + 1 - 2*Upows

        return
```

```
    def virtual_U_gradings_succ(self, i, succ, vert, component_columns):

        ed_weight = self.comp[vert][succ]['diffweight']

        Upows = link_U_deg(ed_weight, self.ring, component_columns)
        self.comp.nodes()[succ][f'UGrading{i}'] =
        ↪  self.comp.nodes()[vert][f'UGrading{i}'] - 1 + 2*Upows

        return

    def virtual_V_gradings_pred(self, i, pred, vert, component_columns):

        ed_weight = self.comp[pred][vert]['diffweight']

        Vpows = link_V_deg(ed_weight, self.ring, component_columns)
        self.comp.nodes()[pred][f'VGrading{i}'] =
        ↪  self.comp.nodes()[vert][f'VGrading{i}'] + 1 - 2*Vpows

        return

    def virtual_V_gradings_succ(self, i, succ, vert, component_columns):

        ed_weight = self.comp[vert][succ]['diffweight']

        Vpows = link_V_deg(ed_weight, self.ring, component_columns)
        self.comp.nodes()[succ][f'VGrading{i}'] =
        ↪  self.comp.nodes()[vert][f'VGrading{i}'] - 1 + 2*Vpows

        return

    def maslov_U_pred(self, pred, vert):

        ed_weight = self.comp[pred][vert]['diffweight']

        component_columns = self.sigx

        Upows = link_U_deg(ed_weight, self.ring, component_columns)
        self.comp.nodes()[pred]['UGrading'] = self.comp.nodes()[vert]['UGrading'] + 1
        ↪  - 2*Upows

        return

    def maslov_U_succ(self, succ, vert):

        ed_weight = self.comp[vert][succ]['diffweight']
```

```
        component_columns = self.sigx

        Upows = link_U_deg(ed_weight, self.ring, component_columns)
        self.comp.nodes()[succ]['UGrading'] = self.comp.nodes()[vert]['UGrading'] - 1
        ↪  + 2*Upows

        return

    def maslov_V_pred(self, pred, vert):

        ed_weight = self.comp[pred][vert]['diffweight']

        component_columns = self.sigo

        Vpows = link_U_deg(ed_weight, self.ring, component_columns)
        self.comp.nodes()[pred]['VGrading'] = self.comp.nodes()[vert]['VGrading'] + 1
        ↪  - 2*Vpows

        return

    def maslov_V_succ(self, succ, vert):

        ed_weight = self.comp[vert][succ]['diffweight']

        component_columns = self.sigo

        Vpows = link_U_deg(ed_weight, self.ring, component_columns)
        self.comp.nodes()[succ]['VGrading'] = self.comp.nodes()[vert]['VGrading'] - 1
        ↪  + 2*Vpows

        return

    ########################################
    # End of relative grading support functions


    def find_max_difference(self, key_set):

        # For a given set of keys this function iterates through the graph and finds
        ↪  the largest difference. This could be improvable
        # speed-wise by considering edges instead but as it stands the grading would
        ↪  have to be recomputed since that data is
        # recorded in  the vertices instead. So in its current state that would be more
        ↪  expensive in processing and this is cheaper
```

```
        # memory wise regardless.

        if type(key_set) == str:
            key_set = [key_set]

        for key in key_set:
            self.max_grading_changes[key] = 0

        result = 0
        for vert in self.comp.nodes():
            for nb in self.comp.neighbors(vert):
                for key in key_set:
                    if (abs(self.comp.nodes()[vert][key] - self.comp.nodes()[nb][key]))
                    ↪  > self.max_grading_changes[key]:
                        print("setting value")
                        self.max_grading_changes[key] =
                        ↪  abs(self.comp.nodes()[vert][key] -
                        ↪  self.comp.nodes()[nb][key])

        return

    def parallel_graph_single_split(self, key, split_count, split_blocks = None):

        # Deprecated by ego split

        # WARNING: !!!split count should be passed at most one lower than the actual
        ↪  number of cores available, this is because of
        # ceilings being a part of the function - it means it can return a set with
        ↪  more blocks than the given split count!!!

        self.find_max_difference(key)

        max_step = self.max_grading_changes[key]

        self.find_grading_ranges(key)

        if split_blocks == None:

            split_blocks = degree_partition(max_step,
            ↪  math.ceil(self.min_gradings[key]), math.ceil(self.max_gradings[key]),
            ↪  split_count)

        if split_blocks == None:

            return None
```

```
    result = []

    for split in split_blocks:

        current_subgraph = []
        vertex_set = []
        vertex_set = [vert for vert in self.comp.nodes() if
        ↪  ((self.comp.nodes()[vert][key] >= split[0]) and
        ↪  (self.comp.nodes()[vert][key] <= split[- 1]))]
        current_subgraph = self.comp.subgraph(vertex_set).copy()
        result.append([current_subgraph, split])

    return result




def parallel_reduction_helper(self, subgraph_set, overwrite = True):

    print("running parallel_reduction_helper")
    process_dict = {}
    for count, subgraph in enumerate(subgraph_set):

        process_dict[count] = mp.Process(target =
        ↪  subgraph[0].range_graph_red_search(), args = subgraph[1] )
        process_dict[count].start()

    for proc in process_dict:

        proc.join()

    result = subgraph_set[0][0]

    for subgraph in subgraph_set:

        result = nx.compose(result, subgraph)

    if overwrite:

        self.comp = result
        return

    else:
```

```
            return result


    def grading_parallel_graph_red_search(self, proc_count = 2, splitting_key =
    ↪  "AGrading"):

        # Deprecated by ego_parallel_red_search

        # graph_red_search with parallel processing by splitting into pieces based on
        ↪  splitting_key


        key = splitting_key

        subgraph_set = self.parallel_graph_single_split(splitting_key, proc_count - 1)

        if subgraph_set == None:

            self.graph_red_search()
            return

        step = parallel_active_range(self.max_grading_changes[key],
        ↪  math.ceil(self.min_gradings[key]), math.ceil(self.max_gradings[key]),
        ↪  proc_count)

        target_loop = math.ceil((1+self.max_gradings[key]
        ↪  -self.min_gradings[key])/step)
        print(str(target_loop) + str(" target number of loops"))

        partition = subgraph_set[:][1]

        for i in range(target_loop):

            self.parallel_reduction_helper(subgraph_set)

            partition_block_iterator(partition, step)

            subgraph_set = self.parallel_graph_single_split(splitting_key, proc_count -
            ↪  1, split_blocks = partition)
#             iterate the split

        return
```

```
    def ego_parallel_red_search(self, cutoff = 100, proc_count = 2):

        # graph_red_search with parallel processing support by using networkx ego graph
        ↪  function
        # to split the graph

        if len([source for source, target, weight in self.comp.edges(data =
        ↪  'diffweight') if weight == 1]) > cutoff:
            print("entering parallel reduction")
        while len([source for source, target, weight in self.comp.edges(data =
        ↪  'diffweight') if weight == 1]) > cutoff:
               print(str(len([source for source, target, weight in self.comp.edges(data
               ↪  = 'diffweight') if weight == 1])) + "reducible edges remaining")
               reducible_edge = rd.sample([source for source, target, weight in
               ↪  self.comp.edges(data = 'diffweight') if weight == 1],  1)

               self.ego_parallel_sweep(reducible_edge[0],  proc_count)
        print("parallel reduction complete")
        self.graph_red_search()

        return


    def ego_parallel_sweep(self, start_vert = None, proc_count = 2):

        if start_vert == None:

            start_vert = self.comp.nodes()[0]

        size = len(list(self.comp.nodes)[0])
#         (self.comp.nodes[0])
        ego_bands, safety = ego_split(self.comp, start_vert, size)

        partition_data = ego_region_partition(size)

        parallel_subgraph_packer(self.comp, ego_bands, partition_data, self.ring)

        region_count = len(partition_data)
        count = 0
        MyManager.register('list', list)
        with MyManager() as manager:
            processed_subgraphs = manager.list()
            while count < region_count:

                process_dict = {}
```

```
                for i in range(proc_count):

                    if count < region_count:

#                         processed_subgraphs = []
                        process_dict[count] = mp.Process(target = subgraph_red_search,
                        ↪  args = (partition_data[f"block{count}"]['total_region'],
                        ↪  partition_data[f"block{count}"]['search_region'],
                        ↪  processed_subgraphs))
                        process_dict[count].start()
                        count += 1

                # print("Assigned parallel jobs, waiting for them to finish")
                for proc in process_dict:
                    # print(proc)
                    process_dict[proc].join()

                # print("count = " + str(count) + "region_count = " +
                ↪  str(region_count))
            processed_subgraphs = processed_subgraphs._getvalue()
            # print("replacing parent graph...")
            result = processed_subgraphs[0].comp

        for element in processed_subgraphs:

            result = nx.compose(result, element.comp)

        result = nx.compose(result, safety)

        # print('reduced total graph from ' +  str(len(self.comp.nodes())), end = "")

        self.comp = result

        self.remove_zeros()
        # print(' to ' +  str(len(self.comp.nodes())))

        return


def subgraph_red_search(subg, search_reg, result_list):

    # graph_red_search limited to the subgraph search_reg, results are appended to
    ↪  result_list
    # which in practice is a proxy list object handled by a multiprocessing manager
```

```
    subgraph = subg.copy()
    search_region = search_reg.copy()
    og_size = len(subgraph.comp.nodes())
    for ed in [edge for edge in search_region.comp.edges() if
    ↪  search_region.comp.edges[edge]['diffweight'] == 1]:
#         print("identified edge " + str(ed) + " for reduction")
#         print("nodes of subg" + str(subgraph.comp.nodes()))
#         print("nodes of search region" + str(search_region.comp.nodes()))

        if ed in subgraph.comp.edges():
#             print("reducing an edge")
            subgraph.graph_reduction(ed[0],  ed[1])

    f_size = len(subgraph.comp.nodes())
    # print("reduced subgraph from size " + str(og_size) + " to " + str(f_size))
    result_list.append(subgraph)
#     print("running change result length = " + str(len(result_list)))

    return


def ego_split(graph, vertex, n):

    # Returns a list of collections of vertices whose index is also their distance from
    ↪  the provided vertex
    # Safety is provided to catch any separate components.

    result = []

    for i in range(n):

        result.append(nx.ego_graph(graph, vertex, i))

    safety = graph.copy()

    safety.remove_nodes_from(result[n - 1].nodes())

    for i in range(n- 1, 0, - 1):

        result[i].remove_nodes_from(result[i- 1].nodes())

    return result, safety

def ego_region_partition(n):
```

```
    # Returns a dictionary of dictionaries which indicate the regions that are going to
    ↪  be reduced and
    # preserved during the parallel reduction

    result = {}

    split_count = math.ceil(n/4)

    result["block0"] = {"search_region": [0,1] , "reserved_region": [2]}

    for i in range(1,split_count):

        result[f"block{i}"] = {"search_region" : [3*i, 3*i+1],  "reserved_region" :
        ↪  [3*i- 1,3*i+2]}

    return result

def parallel_subgraph_packer(graph, subgraphs, region_data, ring):

    # Takes the collections of vertices (intended for those from ego_split) as
    ↪  subgraphs from a parent
    # graph and joins them up as subgraphs based on region_data. ring is provided to
    ↪  construct grid_complex objects
    # from the resulting graphs.

    # region_data should be a dict of dicts with inner dict data labeled
    ↪  "search_region" and "reserved_region"
    # the outer data should be labeled f"block{i}". See ego_region_partition for an
    ↪  example function that works
    # with this

    # subgraphs should be a list of subgraphs corresponding to the region data
    ↪  specified above

    for data in region_data:
#         print(region_data[data])

        region_nodes = []

        # unpacking the indices of the subgraphs we were passed - so we need to unpack
        ↪  3
        # layers deep in total

        for region in region_data[data]:
```

```
            for i in region_data[data][region]:
#                 print(type(subgraphs[i]))

                region_nodes += (list(subgraphs[i].nodes()))

        packed_subgraph = graph.subgraph(region_nodes)

        region_data[data]['total_region'] = grid_complex(packed_subgraph, ring)


    for data in region_data:

        region_nodes = []

        for i in region_data[data]['search_region']:

            region_nodes += list(subgraphs[i].nodes())

        packed_subgraph = graph.subgraph(region_nodes)

        region_data[data]['search_region'] = grid_complex(packed_subgraph, ring)

    return region_data


def subgraph_neighborhood(graph, subgraph):

    # Output: subgraph induced by the given subgraph and any neighbors it has in graph

    result_nodes = set(subgraph.nodes())
    for node in subgraph.nodes():

        for neighbor in graph.neighbors(node):

            result_nodes.add(neighbor)

    result = graph.subgraph(result_nodes)

    return result



def partition_block_iterator(blocks, step_size):

    for count, block in enumerate(blocks):
```

```
        if count == 0:

            for i in range(1, len(block)):

                blocks[i] += step_size

        else:

            for i in range(len(block)):

                blocks[i] += step_size

    return

def name_some_vars(letters, num):

# Accepts a collection of strings, and an integer. Passing "U" and 3 for example
↪  returns "U0, U1, U2"

    result = []
    num = int(num)
    for letter in letters:

        for i in range(num):
            new_var = f"{letter}{i}"
            #print(new_var)
            result.append(new_var)

    return result

def construct_cinf(g, sigx, sigo, size = - 1):

    # Construct CFKinf complex from graph data - essentially just changing weights to
    ↪  polynomials
    # Only works for grid diagrams *not* Latin Squares

    print('constructing cinf...')
    if size == - 1:
        size = len(g.get_edge_data(list(g.edges())[0][0],
        ↪  list(g.edges())[0][1])['diffweight'][0])  #kind of a mess - just turning
        ↪  the edges
        print("Grid size is " + str(size/2))
        n = size/2
        ↪  #into a list and checking the length of#the weight of the first edge
```

```
    else:
        n = size
    timerstart = time.time()
    F,Vars = cinf_coeff(n)
    resG = nx.DiGraph()
    for edge in g.edges:

        start = edge[0]
        end = edge[1]
        poly = F(0)


        for subweight in g[edge[0]][edge[1]]['diffweight']:

            i = 0
            polychange = F(1)
#             print(str(subweight) + str(edge))
            for entry in subweight:

                polychange = polychange*(Vars[i])**entry
                i = i + 1

            poly += polychange
#             print(str(edge) + str(poly))
        resG.add_edge(start,end,diffweight = poly)

    timerend = time.time()
    elap = timerend - timerstart
    print('Time to construct cinf '+ str(elap))
    return grid_complex(resG, F, sigx, sigo)


def cinf_coeff(size):

    # Takes size as an argument and returns the Laurent polynomial ring over Z2 with
    ↪  coefficients U0,...Usize- 1,V0,...Vsize- 1

    n = size
    varis = name_some_vars(['U','V'], n)
    F = LaurentPolynomialRing(GF(2), varis)
    F.inject_variables()

    return F,list(F.gens())
```

```
def range_skip_entry(n, skip):

    # Acts similarly to standard range(n) but omits the "skip"th entry

    u = []
    for i in range(0, skip): u.append(i)
    for j in range(skip+1, n): u.append(j)
    return u


def link_GFC(sigx, sigo, filename = None):

    # Group of usual commands/functions used to produce, simplify and output a
    ↪  grid_complex and
    # its simplification in one function - uses the parallel processing functions

    start_time = time.time()

    if filename == None:
        filename = "X"
        for pos in sigx:
            filename = filename + str(pos)
        filename = filename + "O"
        for pos in sigo:
            filename = filename + str(pos)
        filename = filename + ".gml"

    comp = setup_complex(sigx, sigo)

    print("passing to parallel reducer")
    comp.ego_parallel_red_search(proc_count = PROCESSOR_COUNT)
#     comp.parallel_graph_red_search(PROCESSOR_COUNT, split_key)
    print("completed parallel reducer function")
    comp.gml_export(filename)
    picklefilename = filename + ".p"
    gfk.pickle_it(comp, picklefilename)

    comp.link_normalize()

#     comp.parallel_graph_red_search(PROCESSOR_COUNT)

    filename = "Normalized" + filename
    comp.gml_export(filename)
    picklefilename = filename + ".p"
    gfk.pickle_it(comp, picklefilename)
```

```
    for i in range(len(comp.components)):
        comp.find_grading_ranges(f'AGrading{i}')

    end_time = time.time()

    print("Time to process complex = " + str(end_time - start_time) + " seconds")

    return comp



def setup_complex(sigx, sigo):

    # Takes two lists sigx and sigo, constructs and grades the associated complex then
    ↪  returns the result

    raw_complex = gfk.build_cinf([sigx, sigo])

    comp = construct_cinf(raw_complex, sigx, sigo)

    comp.grade_link_complex()

    return comp



def link_components(sigx, sigo):

    # Returns the number of components in the link defined by sigx and sigo

    xperm = pr.perm(sigx)
    operm = pr.perm(sigo)
    comps = xperm*operm**(- 1)
    result = comps.cycles()

    return result

def link_U_deg(poly, ring, component_columns):

    # Input: poly a laurent polynomial in field a laurent polynomial ring
    #
    # Output: The total sum of powers of Ui in poly

    gens = ring.gens()
```

```
    size = len(gens)/2
    degree = 0

    if type(poly) == sage.rings.finite_rings.integer_mod.IntegerMod_int: return 0

    powers = poly.exponents()

    # len(powers) tells you how many terms the polynomial has
#     if len(powers) > 1:

#         print(poly)

#         raise Exception("Ran into a non-homogoneous degree change - polynomial wasn't
↪  a monomial")

    if len(powers) == 0:

        return 0

    # powers is a list of lists since its intended for more than just monomials, since
    ↪  we are only care about the leading
    # term we pull that one out
    powers = powers[0]

    for i in component_columns:

        degree = degree + powers[i- 1]

    return degree


def link_V_deg(poly, ring, component_columns):

    #Input: poly a laurent polynomial in "ring" a laurent polynomial ring
    #
    #Output: The total sum of powers of Ui in poly

    gens = ring.gens()
    size = len(gens)/2
    degree = 0

    if type(poly) == sage.rings.finite_rings.integer_mod.IntegerMod_int: return 0

    powers = poly.exponents()
```

```
    if len(powers) == 0:

        return 0

    # powers is a list of lists since its intended for more than just monomials, since
    ↪  we are only care about the leading
    # term we pull that one out

    powers = powers[0]

    for i in component_columns:

        degree = degree + powers[size + i- 1]

    return degree

def parallel_active_range(max_grading_step, lower_range, upper_range, split_count):

    # deprecated by ego parallelization

    # Finds and returns the range of gradings that can be reduced without affecting the
    ↪  gluing region
    # or bleeding outside of the reserved regions

    block_size = math.floor((upper_range - lower_range)/split_count)

    result = block_size - 2*max_grading_step

    return result


def degree_partition(max_grading_step, lower_range, upper_range, split_count):

    # deprecated by ego parallelization

    # Returns lists marking degrees for gluing, reducing and preserving when splitting
    ↪  the graph
    # by grading for parallelization

    #output = list of lists

    if split_count == 0:
        raise Exception("Cannot split the graph into 0 pieces - check function
        ↪  arguments")
```

```
    first_round = []

    block_size = math.floor((upper_range - lower_range)/split_count)

    active_range = block_size - 2*max_grading_step
    print("active range " + str(active_range))

    if ((active_range <= 0) and (split_count > 1)) :

        print(str((max_grading_step, lower_range, upper_range, split_count - 1)))
        print("Cannot partition the graph into this many pieces! Parititioning into a
        ↪  smaller number of pieces")
        return degree_partition(max_grading_step, lower_range, upper_range, split_count
        ↪  - 1)

    if split_count == 1:

        return None

    block = []

    max_grading_step += - 1

    trailing_edge = lower_range - 1 - max_grading_step
    leading_edge = trailing_edge

    while trailing_edge < upper_range:

        block = []
        block.append(leading_edge)
        leading_edge += 1
        block.append(leading_edge)
        leading_edge += max_grading_step
        block.append(leading_edge)
        leading_edge += active_range
        block.append(leading_edge)
        leading_edge += max_grading_step
        block.append(leading_edge)
        leading_edge += 1
        block.append(leading_edge)
        first_round.append(block)
        trailing_edge = leading_edge

    print(first_round)
    return first_round
```

```
def alexander_grading_equivalent(comp, source, target, component_count):

    #Returns bool for if two vertices have the same Alexander multigradings

    result = True

    for i in range(component_count):
        if comp.nodes()[source][f'AGrading{i}'] !=
        ↪  comp.nodes()[target][f'AGrading{i}']:
            result = False

    return result


#End of main code block
```

APPENDIX D

LINK AND KNOT PERMUTATIONS

The following code is the collection of grid presentations of knots up to grid size 9 and all links from LinkInfo up to size 9 in one line notation. It is stored in Python dictionaries of lists as written. The entire collection of links listed on Linkinfo is available along with the program publicly at https://github.com/CStClairMath/GridTools. The code is included here without exposition and explanation as in the primary sections.

```
#Permutations of knots up to arc index (grid size) 9 transcribed from Lenherd Ng's
↪  Legendrian knot atlas

knot_dict = dict(k3_{1}  = [[5, 1, 2, 3, 4], [2, 3, 4, 5, 1]],
mk3_{1}  = [[5,4,3,2,1],[2,1,5,4,3]],
k4_{1}  = [[6, 1, 4, 5, 3, 2],[3, 5, 6, 2, 1, 4]],
k5_{1}  = [[5, 6, 7, 1, 2, 3, 4],[7, 3, 4, 5, 6, 1, 2]],
mk5_{1}  = [[2, 1, 7, 6, 5, 4, 3],[7, 6, 5, 4, 3, 2, 1]],
k5_{2}  = [[2, 6, 7, 3, 4, 5, 1],[7, 1, 5, 6, 2, 3, 4]],
mk5_{2}  = [[4, 1, 7, 6, 3, 2, 5],[7, 6, 5, 2, 1, 4, 3]],
k6_{1}  = [[8, 1, 6, 7, 4, 5, 3, 2],[3, 7, 8, 5, 6, 2, 1, 4]],
mk6_{1}  = [[8, 1, 6, 7, 3, 2, 5, 4],[5, 7, 8, 2, 1, 4, 3, 6]],
k6_{2}  = [[8, 3, 5, 6, 7, 2, 1, 4],[6, 7, 8, 1, 4, 5, 3, 2]],
mk6_{2}  = [[8, 7, 6, 5, 2, 1, 3, 4],[3, 1, 8, 7, 6, 4, 5, 2]],
k6_{3}  = [[8, 1, 4, 6, 7, 5, 3, 2],[4, 5, 7, 8, 3, 2, 1, 6]],
k7_{1}  = [[2, 7, 8, 9, 1, 3, 4, 5, 6],[9, 1, 2, 5, 6, 7, 8, 3, 4]],
mk7_{1}  = [[2, 1, 9, 8, 7, 6, 5, 4, 3],[9, 8, 7, 6, 5, 4, 3, 2, 1]],
k7_{2}  = [[2, 8, 9, 6, 7, 3, 4, 5, 1], [9, 1, 7, 8, 5, 6, 2, 3, 4]],
mk7_{2}  = [[6, 1, 9, 8, 3, 2, 5, 4, 7], [9, 8, 7, 2, 1, 4, 3, 6, 5]],
k7_{3}  = [[4, 1, 9, 8, 7, 6, 3, 2, 5], [9, 8, 7, 6, 5, 2, 1, 4, 3]],
mk7_{3}  = [[5, 8, 9, 6, 7, 1, 2, 3, 4], [9, 3, 7, 8, 4, 5, 6, 1, 2]],
k7_{4}  = [[6, 1, 9, 3, 2, 8, 5, 4, 7], [9, 8, 2, 1, 7, 4, 3, 6, 5]],
mk7_{4} = [[2, 8, 9, 3, 6, 7, 4, 5, 1], [9, 1, 7, 8, 2, 5, 6, 3, 4]],
k7_{5} = [[3, 8, 9, 1, 4, 5, 6, 7, 2], [9, 1, 2, 7, 8, 3, 4, 5, 6]],
mk7_{5} = [[4, 1, 9, 8, 7, 3, 2, 6, 5], [9, 8, 7, 6, 2, 1, 5, 4, 3]],
k7_{6} = [[2, 8, 9, 4, 3, 5, 6, 7, 1], [9, 1, 3, 2, 7, 8, 4, 5, 6]],
mk7_{6} = [[3, 1, 9, 8, 6, 7, 5, 4, 2], [9, 8, 7, 2, 1, 4, 3, 6, 5]],
k7_{7} = [[2, 5, 8, 9, 7, 4, 3, 6, 1], [9, 1, 3, 6, 2, 8, 5, 4, 7]],
mk77 = [[4, 8, 9, 2, 1, 7, 5, 6, 3], [9, 1, 7, 8, 6, 3, 2, 4, 5]],
k8_{19} = [[3, 2, 1, 7, 6, 5, 4], [7, 6, 5, 4, 3, 2, 1]],
mk8_{19} = [[3, 4, 5, 6, 7, 1, 2], [7, 1, 2, 3, 4, 5, 6]],
k8_{20} = [[8, 2, 1, 4, 5, 7, 6, 3], [4, 7, 5, 6, 8, 3, 2, 1]],
mk8_{20} = [[8, 6, 4, 5, 3, 7, 1, 2], [3, 1, 7, 8, 6, 2, 4, 5]],
```

```
k8_{21}= [[8, 1, 4, 6, 5, 7, 2, 3], [5, 7, 8, 2, 1, 3, 4, 6]],
mk8_{21} = [[8, 7, 4, 6, 5, 3, 1, 2], [3, 1, 8, 2, 7, 6, 4, 5]],
k9_{42} = [[8, 7, 3, 2, 1, 4, 6, 5], [4, 1, 8, 6, 5, 7, 3, 2]],
mk9_{42} = [[8, 1, 5, 6, 7, 4, 2, 3], [4, 7, 8, 2, 3, 1, 5, 6]],
k9_{43} = [[4, 8, 9, 3, 2, 7, 1, 6, 5], [9, 1, 7, 8, 6, 5, 4, 3, 2]],
mk9_{43} = [[7, 1, 6, 9, 2, 3, 4, 5, 8], [9, 8, 4, 5, 6, 7, 1, 2, 3]],
k9_{44} = [[3, 8, 1, 9, 6, 4, 5, 7, 2], [9, 5, 7, 2, 1, 8, 3, 4, 6]],
mk9_{44} = [[5, 8, 9, 3, 1, 7, 2, 6, 4], [9, 2, 7, 8, 6, 4, 5, 3, 1]],
k9_{45} = [[2, 8, 9, 3, 5, 4, 6, 7, 1], [9, 1, 4, 7, 2, 8, 3, 5, 6]],
mk9_{45} = [[4, 2, 9, 3, 8, 1, 7, 5, 6], [9, 8, 5, 7, 4, 6, 2, 1, 3]],
k9_{46} = [[8, 3, 1, 5, 7, 4, 6, 2], [4, 7, 6, 8, 2, 1, 3, 5]],
mk9_{46} = [[8, 1, 7, 5, 6, 3, 4, 2], [3, 6, 4, 8, 2, 7, 1, 5]],
k9_{47} = [[4, 2, 1, 9, 7, 5, 6, 8, 3], [9, 8, 6, 3, 2, 1, 4, 5, 7]],
mk9_{47} = [[6, 1, 9, 2, 8, 3, 4, 5, 7], [9, 8, 4, 7, 5, 6, 1, 2, 3]],
k9_{48} = [[5, 1, 9, 2, 7, 6, 8, 4, 3], [9, 8, 4, 6, 5, 1, 3, 2, 7]],
mk9_{48} = [[2, 5, 8, 9, 4, 6, 7, 3, 1], [9, 1, 3, 7, 8, 2, 5, 6, 4]],
k9_{49} = [[6, 2, 9, 1, 8, 5, 4, 3, 7], [9, 8, 4, 7, 3, 2, 1, 6, 5]],
mk9_{49} = [[2, 4, 8, 9, 5, 6, 7, 1, 3], [9, 1, 3, 7, 8, 2, 4, 5, 6]],
k10_{124} = [[8, 7, 6, 5, 4, 3, 2, 1], [3, 2, 1, 8, 7, 6, 5, 4]],
mk10_{124} = [[8, 1, 2, 3, 4, 5, 6, 7], [3, 4, 5, 6, 7, 8, 1, 2]],
k10_{128} = [[5, 1, 9, 4, 3, 2, 8, 7, 6], [9, 8, 2, 1, 7, 6, 5, 4, 3]],
mk10_{128} = [[2, 8, 9, 3, 4, 5, 6, 7, 1], [9, 1, 6, 7, 8, 2, 3, 4, 5]],
k10_{132} = [[3, 8, 2, 9, 4, 7, 5, 6, 1], [9, 1, 7, 6, 8, 3, 2, 4, 5]],
mk10_{132} = [[6, 2, 9, 8, 4, 5, 3, 7, 1], [9, 8, 7, 3, 1, 2, 6, 4, 5]],
k10_{136} = [[5, 1, 9, 7, 3, 2, 8, 4, 6], [9, 8, 4, 2, 1, 6, 5, 7, 3]],
mk10_{136} = [[2, 6, 8, 9, 4, 5, 3, 7, 1], [9, 1, 3, 7, 8, 2, 6, 4, 5]],
k10_{139} = [[4, 3, 9, 2, 8, 1, 7, 6, 5], [9, 8, 7, 6, 5, 4, 2, 3, 1]],
mk10_{139} = [[5, 4, 6, 8, 9, 7, 1, 3, 2], [9, 1, 2, 3, 4, 5, 6, 7, 8]],
k10_{140} = [[4, 8, 2, 9, 3, 5, 7, 6, 1], [9, 1, 7, 6, 8, 2, 4, 3, 5]],
mk10_{140} = [[7, 2, 9, 5, 6, 4, 8, 3, 1], [9, 8, 3, 1, 2, 7, 5, 6, 4]],
k10_{142} = [[6, 1, 9, 5, 4, 3, 2, 8, 7], [9, 8, 3, 2, 1, 7, 6, 5, 4]],
mk10_{142} = [[3, 4, 8, 9, 5, 6, 7, 1, 2], [9, 1, 2, 7, 8, 3, 4, 5, 6]],
k10_{145} = [[5, 8, 7, 1, 9, 4, 2, 3, 6], [9, 4, 3, 5, 6, 7, 8, 1, 2]],
mk10_{145} = [[5, 2, 9, 3, 8, 1, 6, 7, 4], [9, 8, 7, 6, 4, 5, 2, 3, 1]],
k10_{160} = [[5, 1, 9, 2, 4, 3, 8, 7, 6], [9, 8, 3, 7, 1, 6, 5, 4, 2]],
mk10_{160} = [[3, 7, 8, 9, 2, 4, 5, 1, 6], [9, 1, 5, 6, 7, 8, 3, 4, 2]],
k10_{161} = [[4, 3, 5, 7, 9, 6, 8, 2, 1], [9, 8, 1, 2, 3, 4, 5, 6, 7]],
mk10_{161} = [[5, 3, 9, 2, 8, 1, 7, 6, 4], [9, 8, 7, 6, 4, 5, 2, 3, 1]],
k11n19 = [[4, 7, 8, 9, 2, 3, 1, 5, 6], [9, 1, 5, 6, 7, 8, 4, 2, 3]],
mk11n19 = [[4, 3, 7, 9, 8, 2, 1, 6, 5], [9, 8, 1, 6, 5, 7, 4, 3, 2]],
k11n38 = [[4, 1, 9, 2, 6, 8, 7, 3, 5], [9, 8, 3, 7, 1, 5, 4, 6, 2]],
mk11n38 = [[2, 7, 3, 9, 5, 6, 4, 8, 1], [9, 1, 8, 4, 2, 3, 7, 5, 6]],
k11n95 = [[3, 1, 9, 7, 8, 6, 5, 4, 2], [9, 8, 5, 2, 4, 3, 1, 7, 6]],
mk11n95 = [[4, 6, 7, 9, 2, 8, 1, 3, 5], [9, 1, 3, 5, 6, 4, 7, 8, 2]],
k11n118 = [[5, 3, 9, 1, 8, 2, 7, 4, 6], [9, 8, 7, 6, 4, 5, 3, 1, 2]],
```

```
mk11n118 = [[4, 2, 5, 9, 7, 6, 8, 1, 3], [9, 8, 1, 3, 2, 4, 5, 6, 7]],
k12n242 = [[4, 3, 2, 9, 8, 1, 7, 6, 5], [9, 8, 7, 6, 5, 4, 3, 2, 1]],
mk12n242 = [[6, 7, 8, 9, 1, 2, 3, 4, 5], [9, 3, 4, 5, 6, 7, 8, 1, 2]],
k12n591 = [[5, 3, 2, 9, 8, 1, 7, 6, 4], [9, 8, 7, 6, 4, 5, 3, 2, 1]],
mk12n591 = [[5, 7, 8, 9, 1, 2, 3, 4, 6], [9, 3, 4, 6, 5, 7, 8, 1, 2]],
k15n41185 = [[4, 3, 2, 1, 9, 8, 7, 6, 5], [9, 8, 7, 6, 5, 4, 3, 2, 1]],
mk15n41185 = [[4, 5, 6, 7, 8, 9, 1, 2, 3], [9, 1, 2, 3, 4, 5, 6, 7, 8]])

#Permutations of (prime?) links. Retrieved from linkinfo and processed for their
↪  permutations. Not necessarily minimal.

link_dict = dict( L2a1_{0} = [[4, 3, 2, 1], [2, 1, 4, 3]]
L2a1_{1} = [[4, 1, 2, 3], [2, 3, 4, 1]]
L4a1_{0} = [[6, 3, 4, 1, 2, 5], [4, 5, 2, 3, 6, 1]]
L4a1_{1} = [[3, 2, 6, 1, 5, 4], [6, 5, 4, 3, 2, 1]]
L5a1_{0} = [[6, 1, 3, 4, 2, 7, 5], [2, 4, 5, 7, 6, 3, 1]]
L5a1_{1} = [[3, 6, 5, 7, 1, 2, 4], [5, 4, 2, 3, 6, 7, 1]]
L6a1_{0} = [[8, 3, 4, 6, 5, 1, 2, 7], [4, 5, 2, 3, 7, 6, 8, 1]]
L6a1_{1} = [[2, 1, 8, 5, 4, 3, 7, 6], [8, 7, 4, 3, 2, 6, 5, 1]]
L6a2_{0} = [[8, 2, 3, 4, 5, 6, 7, 1], [3, 7, 1, 8, 2, 4, 5, 6]]
L6a2_{1} = [[3, 2, 8, 1, 5, 4, 7, 6], [8, 7, 4, 3, 2, 6, 5, 1]]
L6a3_{0} = [[8, 1, 2, 3, 4, 5, 6, 7], [2, 3, 4, 5, 6, 7, 8, 1]]
L6a3_{1} = [[2, 5, 4, 7, 6, 1, 8, 3], [4, 3, 6, 5, 8, 7, 2, 1]]
L6a4_{0}_{0} = [[8, 1, 7, 5, 4, 2, 3, 6], [2, 4, 3, 8, 6, 5, 7, 1]]
L6a4_{1}_{0} = [[3, 5, 4, 8, 7, 1, 2, 6], [7, 2, 6, 5, 3, 4, 8, 1]]
L6a4_{0}_{1} = [[8, 1, 3, 5, 4, 2, 7, 6], [2, 4, 7, 8, 6, 5, 3, 1]]
L6a4_{1}_{1} = [[3, 2, 4, 5, 7, 1, 8, 6], [7, 5, 6, 8, 3, 4, 2, 1]]
L6a5_{0}_{0} = [[9, 5, 6, 1, 3, 4, 7, 2, 8], [6, 7, 2, 4, 5, 8, 3, 9, 1]]
L6a5_{1}_{0} = [[5, 1, 9, 2, 8, 7, 3, 4, 6], [9, 8, 3, 7, 6, 4, 5, 2, 1]]
L6a5_{0}_{1} = [[6, 7, 5, 4, 3, 8, 2, 1], [8, 2, 1, 6, 5, 4, 7, 3]]
L6a5_{1}_{1} = [[5, 1, 9, 7, 8, 4, 3, 2, 6], [9, 8, 3, 2, 6, 7, 5, 4, 1]]
L6n1_{0}_{0} = [[6, 1, 5, 3, 4, 2], [3, 4, 2, 6, 1, 5]]
L6n1_{1}_{0} = [[3, 4, 2, 6, 1, 5], [6, 1, 5, 3, 4, 2]]
L6n1_{0}_{1} = [[7, 1, 2, 4, 3, 5, 6], [3, 4, 5, 6, 7, 2, 1]]
L6n1_{1}_{1} = [[2, 6, 1, 5, 3, 4], [5, 3, 4, 2, 6, 1]]
L7a1_{0} = [[9, 1, 3, 5, 4, 8, 2, 7, 6], [2, 4, 8, 9, 7, 6, 5, 3, 1]]
L7a1_{1} = [[2, 7, 3, 6, 5, 4, 9, 8, 1], [9, 1, 8, 2, 7, 6, 5, 3, 4]]
L7a2_{1} = [[2, 1, 9, 7, 8, 4, 3, 5, 6], [9, 8, 4, 3, 5, 2, 6, 7, 1]]
L7a3_{0} = [[7, 8, 6, 5, 4, 3, 9, 2, 1], [9, 2, 1, 7, 6, 5, 4, 8, 3]]
L7a4_{0} = [[9, 3, 4, 6, 5, 8, 7, 2, 1], [4, 5, 2, 3, 7, 6, 1, 9, 8]]
L7a4_{1} = [[2, 1, 9, 3, 4, 6, 5, 8, 7], [9, 8, 4, 5, 2, 3, 7, 6, 1]]
L7a5_{0} = [[9, 4, 5, 6, 8, 2, 3, 1, 7], [5, 8, 1, 9, 3, 4, 7, 6, 2]]
L7a5_{1} = [[3, 2, 9, 1, 7, 8, 4, 5, 6], [9, 8, 4, 3, 2, 5, 6, 7, 1]]
L7a6_{0} = [[9, 5, 6, 8, 7, 4, 3, 2, 1], [6, 7, 4, 5, 3, 2, 1, 9, 8]]
L7a6_{1} = [[2, 1, 9, 5, 6, 3, 4, 8, 7], [9, 8, 6, 7, 4, 5, 2, 3, 1]]
```

```
L7a7_{0}_{0} = [[7, 8, 2, 3, 6, 5, 4, 9, 1], [9, 3, 4, 1, 2, 7, 6, 5, 8]]
L7a7_{1}_{0} = [[2, 1, 9, 3, 4, 7, 8, 5, 6], [9, 8, 4, 5, 2, 3, 6, 7, 1]]
L7a7_{0}_{1} = [[7, 8, 4, 3, 2, 5, 6, 9, 1], [9, 3, 2, 1, 6, 7, 4, 5, 8]]
L7a7_{1}_{1} = [[3, 2, 9, 6, 5, 4, 1, 8, 7], [9, 8, 5, 4, 3, 7, 6, 2, 1]]
L7n1_{0} = [[8, 1, 2, 4, 5, 3, 6, 7], [3, 4, 5, 6, 7, 8, 2, 1]]
L7n1_{1} = [[3, 4, 2, 5, 7, 1, 6], [7, 1, 6, 3, 4, 5, 2]]
L7n2_{0} = [[4, 7, 2, 1, 3, 5, 6], [1, 3, 5, 4, 6, 7, 2]]
L7n2_{1} = [[3, 5, 1, 6, 7, 2, 4], [7, 2, 4, 3, 5, 6, 1]]
L8n1_{0} = [[8, 1, 4, 5, 6, 3, 2, 7], [5, 6, 7, 8, 2, 1, 4, 3]]
L8n1_{1} = [[5, 6, 2, 1, 3, 4, 9, 7, 8], [9, 1, 8, 4, 5, 7, 6, 3, 2]]
L8n2_{0} = [[7, 1, 5, 2, 6, 4, 3, 8], [2, 6, 8, 7, 3, 1, 5, 4]]
L8n2_{1} = [[4, 6, 1, 2, 3, 8, 7, 5], [8, 3, 5, 4, 7, 6, 2, 1]]
L8n3_{0}_{0} = [[8, 1, 2, 4, 5, 3, 6, 7], [5, 4, 6, 7, 8, 1, 2, 3]]
L8n3_{1}_{0} = [[4, 2, 1, 5, 3, 7, 8, 6], [7, 8, 3, 2, 6, 4, 5, 1]]
L8n3_{0}_{1} = [[8, 4, 2, 7, 5, 1, 6, 3], [5, 1, 6, 4, 8, 3, 2, 7]]
L8n3_{1}_{1} = [[4, 8, 1, 2, 3, 7, 5, 6], [7, 2, 3, 5, 6, 4, 8, 1]]
L8n4_{0}_{0} = [[8, 1, 2, 6, 5, 4, 3, 7, 9], [6, 5, 7, 4, 9, 8, 1, 2, 3]]
L8n4_{1}_{0} = [[3, 4, 1, 6, 5, 2, 9, 8, 7], [9, 8, 5, 4, 7, 6, 3, 2, 1]]
L8n4_{0}_{1} = [[8, 5, 2, 6, 9, 4, 1, 7, 3], [6, 1, 7, 4, 5, 8, 3, 2, 9]]
L8n4_{1}_{1} = [[5, 7, 1, 2, 3, 8, 4, 6], [8, 2, 3, 4, 6, 5, 7, 1]]
L8n5_{0}_{0} = [[9, 1, 2, 6, 5, 3, 7, 4, 8], [6, 5, 7, 4, 8, 1, 2, 9, 3]]
L8n5_{0}_{1} = [[9, 5, 2, 6, 8, 1, 7, 4, 3], [6, 1, 7, 4, 5, 3, 2, 9, 8]]
L8n7_{0}_{0}_{0} = [[6, 1, 8, 4, 3, 9, 7, 5, 2], [9, 3, 2, 7, 5, 6, 4, 1, 8]]
L8n7_{1}_{0}_{0} = [[3, 1, 2, 9, 5, 4, 7, 8, 6], [9, 7, 4, 3, 8, 6, 1, 5, 2]]
L8n7_{0}_{1}_{0} = [[6, 3, 8, 4, 5, 9, 7, 1, 2], [9, 1, 2, 7, 3, 6, 4, 5, 8]]
L8n7_{1}_{1}_{0} = [[3, 1, 4, 9, 5, 6, 7, 8, 2], [9, 7, 2, 3, 8, 4, 1, 5, 6]]
L8n7_{0}_{0}_{1} = [[6, 1, 8, 7, 3, 9, 4, 5, 2], [9, 3, 2, 4, 5, 6, 7, 1, 8]]
L8n7_{1}_{0}_{1} = [[3, 1, 2, 9, 8, 4, 7, 5, 6], [9, 7, 4, 3, 5, 6, 1, 8, 2]]
L8n7_{0}_{1}_{1} = [[6, 3, 8, 7, 5, 9, 4, 1, 2], [9, 1, 2, 4, 3, 6, 7, 5, 8]]
L8n7_{1}_{1}_{1} = [[3, 1, 4, 9, 8, 6, 7, 5, 2], [9, 7, 2, 3, 5, 4, 1, 8, 6]]
L8n8_{0}_{0}_{0} = [[5, 1, 7, 3, 8, 4, 6, 2], [8, 4, 2, 6, 5, 1, 3, 7]]
L8n8_{1}_{0}_{0} = [[3, 1, 2, 8, 4, 6, 5, 7], [8, 6, 5, 3, 7, 1, 2, 4]]
L8n8_{0}_{1}_{0} = [[5, 4, 7, 3, 8, 1, 6, 2], [8, 1, 2, 6, 5, 4, 3, 7]]
L8n8_{1}_{1}_{0} = [[3, 1, 5, 8, 4, 6, 2, 7], [8, 6, 2, 3, 7, 1, 5, 4]]
L8n8_{0}_{0}_{1} = [[5, 1, 7, 6, 8, 4, 3, 2], [8, 4, 2, 3, 5, 1, 6, 7]]
L8n8_{1}_{0}_{1} = [[3, 1, 2, 8, 7, 6, 5, 4], [8, 6, 5, 3, 4, 1, 2, 7]]
L8n8_{0}_{1}_{1} = [[5, 4, 7, 6, 8, 1, 3, 2], [8, 1, 2, 3, 5, 4, 6, 7]]
L8n8_{1}_{1}_{1} = [[3, 1, 5, 8, 7, 6, 2, 4], [8, 6, 2, 3, 4, 1, 5, 7]]
L9n1_{0} = [[9, 1, 4, 6, 5, 7, 2, 3, 8], [5, 6, 7, 8, 9, 3, 4, 1, 2]]
L9n1_{1} = [[3, 4, 2, 7, 9, 5, 6, 1, 8], [9, 1, 8, 3, 6, 7, 4, 5, 2]]
L9n2_{0} = [[6, 9, 4, 1, 5, 7, 2, 3, 8], [1, 5, 7, 6, 8, 3, 4, 9, 2]]
L9n2_{1} = [[3, 5, 1, 8, 9, 6, 7, 2, 4], [9, 2, 4, 3, 7, 8, 5, 6, 1]]
L9n3_{0} = [[7, 9, 4, 5, 1, 6, 2, 3, 8], [1, 5, 6, 8, 7, 3, 4, 9, 2]]
L9n3_{1} = [[2, 4, 3, 7, 9, 5, 6, 8, 1], [9, 1, 8, 2, 6, 7, 4, 3, 5]]
L9n4_{1} = [[3, 4, 2, 5, 6, 7, 9, 1, 8], [9, 1, 8, 3, 4, 5, 6, 7, 2]]
```

```
L9n5_{0} = [[4, 9, 2, 1, 3, 5, 6, 7, 8], [1, 3, 5, 4, 6, 7, 8, 9, 2]]
L9n5_{1} = [[3, 5, 1, 6, 7, 8, 9, 2, 4], [9, 2, 4, 3, 5, 6, 7, 8, 1]]
L9n6_{1} = [[2, 4, 3, 5, 6, 7, 9, 8, 1], [9, 1, 8, 2, 4, 5, 6, 3, 7]]
L9n7_{0} = [[9, 1, 4, 5, 6, 2, 3, 7, 8], [5, 6, 7, 9, 3, 4, 8, 1, 2]]
L9n13_{0} = [[9, 2, 5, 6, 4, 7, 3, 8, 1], [4, 6, 7, 3, 8, 1, 9, 2, 5]]
L9n14_{0} = [[2, 8, 5, 6, 4, 7, 3, 9, 1], [6, 4, 7, 3, 9, 1, 8, 2, 5]]
L9n14_{1} = [[2, 1, 6, 4, 5, 3, 8, 9, 7], [8, 3, 9, 7, 2, 6, 5, 4, 1]]
L9n15_{0} = [[8, 1, 2, 4, 3, 5, 6, 7], [3, 4, 5, 6, 7, 8, 1, 2]]
L9n15_{1} = [[2, 1, 5, 8, 3, 4, 9, 6, 7], [9, 4, 2, 6, 7, 8, 5, 3, 1]]
L9n16_{0} = [[2, 8, 3, 6, 5, 4, 7, 9, 1], [6, 5, 7, 4, 9, 8, 1, 2, 3]]
L9n17_{1} = [[2, 1, 5, 7, 8, 3, 9, 4, 6], [8, 3, 2, 9, 4, 6, 5, 7, 1]]
L9n18_{1} = [[2, 1, 4, 8, 6, 7, 3, 9, 5], [7, 3, 6, 5, 2, 9, 8, 4, 1]]
L9n19_{1} = [[4, 5, 2, 3, 8, 1, 7, 6], [8, 1, 7, 6, 5, 4, 3, 2]]
L9n23_{0}_{0} = [[9, 1, 3, 5, 6, 2, 7, 4, 8], [6, 5, 7, 8, 9, 4, 3, 1, 2]]
L9n23_{1}_{0} = [[2, 3, 1, 8, 4, 5, 6, 9, 7], [8, 9, 5, 2, 6, 7, 3, 4, 1]]
L9n23_{0}_{1} = [[9, 5, 3, 8, 6, 4, 7, 1, 2], [6, 1, 7, 5, 9, 2, 3, 4, 8]]
L9n23_{1}_{1} = [[2, 9, 1, 8, 6, 5, 3, 4, 7], [8, 3, 5, 2, 4, 7, 6, 9, 1]]
L10n45_{1} = [[5, 3, 7, 8, 6, 2, 9, 1, 4], [9, 6, 4, 5, 1, 7, 3, 8, 2]]
L10n46_{1} = [[1, 8, 5, 4, 6, 7, 3, 2, 9], [7, 3, 2, 1, 9, 5, 8, 6, 4]]
L10n93_{0}_{0} = [[8, 9, 1, 2, 3, 5, 4, 6, 7], [2, 3, 4, 5, 6, 8, 7, 9, 1]]
L10n93_{0}_{1} = [[8, 3, 1, 2, 6, 5, 4, 9, 7], [2, 9, 4, 5, 3, 8, 7, 6, 1]]
L10n93_{1}_{1} = [[2, 3, 5, 7, 6, 1, 8, 9, 4], [5, 6, 8, 4, 9, 7, 2, 3, 1]]
L10n94_{1}_{0} = [[3, 7, 5, 9, 4, 2, 6, 1, 8], [9, 1, 8, 6, 7, 5, 3, 4, 2]]
L10n94_{1}_{1} = [[2, 9, 4, 8, 6, 1, 5, 3, 7], [8, 6, 7, 5, 3, 4, 2, 9, 1]]
L10n104_{0}_{0}_{0} = [[9, 1, 4, 3, 6, 5, 2, 8, 7], [6, 5, 7, 8, 9, 1, 4, 3, 2]]
L10n104_{1}_{0}_{0} = [[1, 9, 2, 5, 6, 4, 3, 7, 8], [5, 4, 7, 1, 3, 9, 8, 2, 6]]
L10n104_{0}_{1}_{0} = [[9, 5, 4, 3, 6, 1, 2, 8, 7], [6, 1, 7, 8, 9, 5, 4, 3, 2]]
L10n104_{1}_{1}_{0} = [[5, 9, 2, 1, 6, 4, 3, 7, 8], [1, 4, 7, 5, 3, 9, 8, 2, 6]]
L10n104_{0}_{0}_{1} = [[8, 6, 1, 4, 3, 2, 5, 7], [4, 2, 5, 8, 7, 6, 1, 3]]
L10n104_{1}_{0}_{1} = [[1, 9, 7, 5, 6, 4, 3, 2, 8], [5, 4, 2, 1, 3, 9, 8, 7, 6]]
L10n104_{0}_{1}_{1} = [[8, 6, 1, 4, 7, 2, 5, 3], [4, 2, 5, 8, 3, 6, 1, 7]]
L10n104_{1}_{1}_{1} = [[5, 9, 7, 1, 6, 4, 3, 2, 8], [1, 4, 2, 5, 3, 9, 8, 7, 6]]
L10n105_{0}_{0}_{0} = [[1, 3, 9, 2, 5, 4, 8, 6, 7], [4, 8, 5, 6, 7, 1, 3, 2, 9]]
L10n105_{1}_{0}_{0} = [[3, 1, 8, 2, 5, 7, 4, 6], [7, 5, 4, 6, 1, 3, 8, 2]]
L10n105_{0}_{1}_{0} = [[4, 3, 9, 2, 5, 1, 8, 6, 7], [1, 8, 5, 6, 7, 4, 3, 2, 9]]
L10n105_{1}_{1}_{0} = [[3, 5, 8, 2, 1, 7, 4, 6], [7, 1, 4, 6, 5, 3, 8, 2]]
L10n105_{1}_{0}_{1} = [[3, 1, 8, 6, 5, 7, 4, 2], [7, 5, 4, 2, 1, 3, 8, 6]]
L10n105_{1}_{1}_{1} = [[3, 5, 8, 6, 1, 7, 4, 2], [7, 1, 4, 2, 5, 3, 8, 6]]
L11n132_{1} = [[2, 9, 1, 3, 7, 6, 4, 5, 8], [7, 5, 6, 8, 2, 9, 1, 3, 4]]
L11n148_{1} = [[1, 8, 7, 4, 6, 5, 3, 2, 9], [5, 3, 2, 1, 9, 7, 8, 6, 4]]
L11n204_{0} = [[8, 9, 1, 2, 3, 4, 5, 6, 7], [2, 3, 4, 5, 6, 7, 8, 1, 9]]
L11n204_{1} = [[2, 3, 7, 5, 6, 1, 8, 9, 4], [6, 5, 4, 8, 9, 7, 2, 3, 1]]
L11n276_{0}_{0} = [[9, 2, 3, 4, 5, 6, 7, 8, 1], [4, 6, 8, 9, 1, 2, 3, 5, 7]]
L11n276_{0}_{1} = [[9, 2, 8, 4, 1, 6, 3, 5, 7], [4, 6, 3, 9, 5, 2, 7, 8, 1]]
L11n276_{1}_{1} = [[8, 4, 1, 2, 3, 5, 7, 9, 6], [3, 9, 5, 6, 7, 8, 1, 4, 2]]
```

```
L11n277_{0}_{0} = [[8, 2, 4, 3, 5, 6, 7, 9, 1], [3, 6, 9, 8, 1, 2, 4, 5, 7]]
L11n277_{0}_{1} = [[8, 2, 9, 3, 1, 6, 4, 5, 7], [3, 6, 4, 8, 5, 2, 7, 9, 1]])
```