COMPUTING GRÖBNER BASES FOR PATH ALGEBRAS IN C++

BY

BRYANT COLLINS

A Thesis Submitted to the Graduate Faculty of

WAKE FOREST UNIVERSITY GRADUATE SCHOOL OF ARTS AND SCIENCES

in Partial Fulfillment of the Requirements

for the Degree of

MASTER OF SCIENCE

Mathematics

May 2025

Winston-Salem, North Carolina

Approved By:

Frank Moore, Ph.D., Advisor

Hugh Howards, Ph.D., Chair

Jacob Mayle, Ph.D.

Table of Contents

Abstract

Given some path algebra P and an ideal of this path algebra I, generated by some set of polynomials $f_1, \ldots, f_l$, it is very common to want to know, given some new g, if $g \in I$. Normally, to do this you would have to find some combination of elements of I that gives g, but you could also compute a Gröbner basis G of I, which is characterized as a collection of $g_1, \ldots, g_k \in I$ such that $I = \langle G \rangle$ and $\langle \mathrm{LT}(I) \rangle = \langle \mathrm{LT}(G) \rangle$, where $\mathrm{LT}(S)$ is the set of lead terms of a set of polynomials S. Computing these using Buchberger's Algorithm can be time consuming by hand and occasionally is actually non-terminating, so it would be convenient if we had an immediate way easily generate ideals and compute their corresponding Gröbner bases. This paper will follow our approach to this problem using C++ and serves as documentation to the project found here (`https://github.com/xZecora/path-algebras`) on Github.

Chapter 1: Introduction

When discussing rings in general, dealing with quotients involving ideals is a major part of the theory. However, when working with specific examples it is common to want to know if a given element is inside of an ideal, or which coset the element is a part of, but this can be rather difficult without clever expressions of that element. The idea of a Gröbner basis was introduced to solve this problem; it is a generating set of an ideal I that provides an algorithmic way to determine if an element is inside of I. So, given some ideal I and it's set of generators $\{f_1, \ldots, f_l\}$ we can generate a Gröbner basis G for that ideal, making the process of finding out if an element is in I much more straightforward and algorithmic. Our motivation for this implementation is to try to create a more memory-efficient version of this process, storing common objects only once and passing around object identifiers instead of creating copies of these objects every time we want to use them.

Our goal – aside from memory efficiency – is to have a program that can be given input in some form that describes a graph and then generates that graph's structure and allows the user to perform operations in the path algebra described by that graph. Specific to our issue of Gröbner bases, we would like to be able to generate some ideal and ask what its Gröbner basis is and if an element is contained in our ideal based off of that information. Feeling that this has been accomplished, the purpose of this thesis is to outline the thought process behind decisions we made in creating this program and to function as a sort of documentation for it.

Chapter 2: Background

2.1 Path Algebras

To begin, there are a few basic definitions to get out of the way before we can define explicitly what a path algebra is. First,

Definition 2.1.1. A *multidigraph* Γ is a directed graph that allows non-trivial loops about a single vertex, and that allows multiple edges between two vertices. We will also refer to this as just a *graph*.

Naturally, now that we have a graph, consider a path through that graph from one vertex to another.

Definition 2.1.2. A *path* is a sequence of edges $\{e_1, \ldots, e_k\}$ through our graph Γ such that e_i ends where e_{i+1} begins, and e_1's start vertex is the paths start vertex, and e_k's end vertex is the paths end vertex.

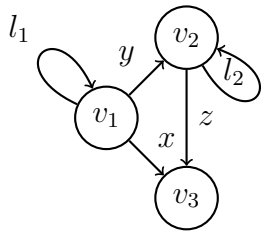
Notably, what we refer to as a path for these purposes is what graph theorists would more strictly refer to as a *walk* because we can visit an edge multiple times.

Definition 2.1.3. A *trivial path* is a path connecting a vertex to itself that functions as a local identity about a vertex.

For the purposes of this thesis, we can use the same labeling as our vertices to denote these trivial paths. Now that we have all of its components defined, we can formally define what a path algebra is.

Definition 2.1.4. Given some multidigraph Γ and some field K we can define a path algebra P – also denoted as $K\Gamma$ – whose basis consists of every finite path through the graph and whose coefficients are in K.

Because we must have some notion of multiplying paths, consider the obvious choice of path concatenation. This clearly works well if the start and end of our two paths match up, but what do we do if the right path does not start where the left path ends? In this case, we define our product to be 0. For example, consider the following graph:



Before multiplying anything, recognize this graph contains loops, specifically the edges l_1 and l_2. These are distinct from the trivial paths represented v_1, v_2, and v_3 because they can have weights assigned to them, unlike the trivial paths. Three examples of paths through this graph are l_2z, l_1y, and x. To multiply these, consider $l_1y \cdot x$. We first consider if their start and end points match. l_1y ends at v_2, while x starts at v_1, so these are not compatible and we get $l_1y \cdot x = 0$. We can also consider, $l_2z \cdot l_1y$. We run into a similar issue where l_2z ends at v_2 but l_1y starts at v_1, and so $l_2z \cdot l_1y = 0$. However, if we consider the other multiplicaiton of $l_1y \cdot l_2z$, they start and end at v_2 respectively, and so they can be multiplied and we get $l_1y \cdot l_2z = l_1yl_2z$, the concatenated paths. These paths are the equivalent of vectors in our path algebra P.

However, paths are not elements of the path algebra. An element of P is linear combination of paths through the underlying graph Γ with coefficients in K and, for the rest of this thesis, these are analagous to polynomials in a polynomial ring. Now that we have a notion of what an element of P looks like and of an identity in our graph Γ, we will want to know what the identity of P. Consider the sum of trivial paths about every vertex, $e = v_1 + \cdots + v_c$. Any path multiplied by this

identity will just be the original path because distributing it to any vertex that it either doesn't start at for left multiplication or doesn't end at for right multiplication will just give us a zero, so we're left with exactly one copy of our original path. This follows similarly for multiplying any polynomial by our identity. Therefore, we are always left with our original path or linear combination of paths and this gives us a multiplicative identity for P.

2.2 Orderings on Paths

When considering the polynomials described in the last section, we need to give the set of paths in a path algebra a well-ordering that is compatible with multiplication. Indeed:

Definition 2.2.1. Let $<$ be a well-ordering on the set of paths. We say that $<$ is admissible if given paths p, q, r, and s such that $p < q$ one has that $rp < rq$ and $ps < qs$, assuming these products are defined.

From here on out, we will denote an arbitrary admissible ordering as $<$. Importantly for defining such an ordering, our edge's weights are represented by n dimensional vectors, and so two immediately appealing definitions for $<$ would be the euclidean metric of our path's weight vector and the component wise comparison of our vector's elements. While the second option has more average comparisons per check compared to the first option, it has no computations, which are much more expensive to compute. Therefore we will use the component wise comparison. Though this raises an issue: one requirement is that we need to have a strict ordering, meaning any two elements must have a smaller one, and if we use just the weights, it is possible that two paths have the same weight. The easiest solution, and the one we ended up using, is combining this weight comparison with a lexicographic comparison.

To formally define this admissible ordering: Consider two paths $p = e_1e_2 \dots e_k$ and $q = f_1f_2 \dots f_l$. Because we allow weights to be multidimensional, we consider the sum of the weight vectors to be the weight of our path, and initially compare p's weight and q's weight component wise. The first path to have a smaller component than the other is considered to be the smaller, and we would say $p < q$ if p is smaller than q. This will quickly give a result in most scenarios, but in the case where they have identical weight vectors, we will fall back to our length-lexicographic comparison. So in the case of p and q, we start by checking which of k or l is larger. If they are non equal, we consider p to be smaller in the case that $k < l$, and q to be smaller in the case that $l < k$. However, in the worst case scenario where they have equal weights and lengths, we iterate over our e_is and f_is simultaneously, comparing the rankings of our edges, where each edge is given some unique value from 1 to n, where n is the number of edges, and 1 being the heaviest rank. The first two edges that are not identical will determine which is smaller, if e_i comes earlier in our list of edges that f_i, p is the smaller path, and similarly for q.

2.3 Long Division in the Path Algebra

A major operation we will perform throughout this process is division of a polynomial f by some set of polynomials G, the remainder of which we will denote as $\overline{f}^G$. This process is the backbone of Buchberger's Algorithm for generating Gröbner bases and will be used constantly. This notion of dividing a polynomial by an entire set is not immediately obvious and so the algorithm is described in Algorithm 1. Note that $\mathrm{LT}(f)$ is the first term of a polynomial f and $\mathrm{LP}(f)$ is the path associated with $\mathrm{LT}(f)$.

The general process involves taking our dividend, initially f, and searching through the set of lead paths to find a subword of the lead path of our dividend. After we have

Algorithm 1 Polynomial Division by a Set

Input: a polynomial f and a set of polynomials $G = \{g_1, \ldots, g_c\}$
Output: the remainder r

$d \leftarrow f$
$r \leftarrow 0$
while d is non-zero **do**
  $g \leftarrow$ first polynomial in G such that $\mathrm{LP}(g)$ is a subword of $\mathrm{LP}(d)$, 0 otherwise
  if g is non-zero **then**
    $p \leftarrow$ everything in $\mathrm{LP}(d)$ up to $\mathrm{LP}(g)$ ▷ Prefix of d
    $s \leftarrow$ everything in $\mathrm{LP}(d)$ after $\mathrm{LP}(g)$ ▷ Suffix of d
    $d \leftarrow d - p * g * s$
  else ▷ No subwords of the lead term
    $r \leftarrow r + \mathrm{LT}(d)$
    $d \leftarrow d - \mathrm{LT}(d)$
  end if
end while

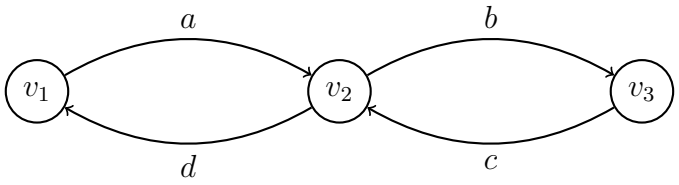


Figure 2.1: Graph corresponding to Example 2.3.1

a polynomial whose lead path is a subword of our dividend's lead path, we multiply it by whatever we need to make their lead paths equal – specifically on the left and right by whatever is on the left and right of the subword inside of the dividend's lead path – and subtract that product from the dividend. If we cannot find anything that is a subword, we take off the lead term of our dividend and put it into our remainder. Because we have an admissible ordering $<$ on our paths and our polynomials are sorted, there is some smallest term, and therefore this will always terminate, and so we can blindly repeat until our dividend is 0.

Example 2.3.1. Let us consider an example of this using the set $G = \{cdab - bc, bc - da\}$ and the ordering $v_1 < v_2 < v_3 < d < c < b < a$. We will reduce the polynomial

$d = -dadab + bcb$ with respect to G. First, set our remainder r to $r = 0$, then check the lead terms of G for any subwords in $-dadab$, noticing that there are none. Because there is no subword, $-dadab$ goes into our remainder r and is removed from d, so we get

$$r = -dadab, \text{ and } d = bcb.$$

Now check our lead terms of G for subwords of bcb, and notice that we have $bc \in bc-da$ as a subword of bcb. So now $p = v_2$ and $s = b$, and we set $d = bcb - v_2(bc - da)b = bcb-bcb-dab = -dab$, because b and d start at the same vertex. We are now left with $d = -dab$, and we can find no for it in the lead terms of G, and so $r = r - dab$ and $d = d + dab = 0$. $d = 0$ and so we are done and our remainder is $r = -dadab + dab$.

2.4 Lead Term Ideals and Gröbner Bases

To begin, we want to restate some definitions from the last section with more specificity. Define $\mathrm{LT}(f_i)$ to be the lead term of the polynomial f_i, which is the product of a coefficient c_i from the field K and a path p_i from the path algebra P, also define $\mathrm{LP}(f_i)$ to be p_i. Next we define the $\mathrm{LT}(S) = \{\mathrm{LT}(f) | f \in S\}$, where S is some set of polynomials in P. Now, let $I = \langle f_1, f_2, \ldots, f_c \rangle$ be an ideal of P generated by the set $\{f_1, \ldots, f_i\}$. Consider the lead term ideal, $\langle \mathrm{LT}(I) \rangle$, defining it as ideal generated by the lead terms of each element of I. If we consider the ideal generated by the lead terms of our f_is, $\langle \mathrm{LT}(f_1), \mathrm{LT}(f_2), \ldots, \mathrm{LT}(f_c) \rangle$, then this is guaranteed to at least be a subset of $\langle \mathrm{LT}(I) \rangle$.

Definition 2.4.1. A set of generators $g_1, \ldots, g_c$ is said to be a Gröbner basis if $\langle g_1, \ldots, g_c \rangle = I$ and $\langle \mathrm{LT}(g_1), \ldots, \mathrm{LT}(g_c) \rangle = \langle \mathrm{LT}(I) \rangle$.

Conveniently, given some set of generators $f_1, \ldots, f_c$ of an ideal I, we can use Buchberger's algorithm to generate a Gröbner basis for it. The algorithm is described

in Algorithm 2, and is based heavily on the algorithm described in [3], having been slightly modified for the case of a path algebra, and to more closely match our code implementation that will be discussed later.

Algorithm 2 Buchberger's Algorithm

Input: a set of generators G
Precondition: all elements of G are monic
Output: A Gröbner Basis G if the ideal generated by the input

1: $S \leftarrow$ set of SPols generated using Algorithm 3 ▷ SPols will be explained shortly
2: **while** S is non-empty **do**
3: $f \leftarrow$ optimal element in S ▷ lowest degree element
4: $S \leftarrow S \setminus \{f\}$
5: $r \leftarrow$ remainder of $\overline{f}^G$ ▷ division by the set G
6: **if** $r \neq 0$ **then**
7: $r \leftarrow r$ divided by its leading coefficient ▷ make r monic
8: $G \leftarrow G \cup \{r\}$
9: $S \leftarrow S \cup \{$new SPols generated with $r\}$
10: **end if**
11: **end while**

Theorem 2.3 in [4] shows that when S is empty, the set G is indeed a Gröbner basis of the ideal generated by the input. However, for some inputs, this condition may never be met. But if there are no cycles in the graph, the path algebra is finite dimensional and hence this algorithm is guaranteed to terminate.

In line 1 of Algorithm 2, we refer to something called an SPol. To generate an SPol, we take two monic polynomials f and g and find any overlap between $\mathrm{LT}(f)$ and $\mathrm{LT}(g)$'s paths, if there is no overlap we cannot generate an SPol using these. Without loss of generality, say $\mathrm{LT}(f)$ has a suffix that overlaps with $\mathrm{LT}(g)$, and $\mathrm{LT}(g)$ has the same subpath as a prefix. Define p as the subpath of $\mathrm{LP}(f)$ up to that overlap, and define s as the subpath of $\mathrm{LP}(g)$ after that overlap. More specifically, say $\mathrm{LP}(f) = f_1 \dots f_k o_1 \dots o_n$ and $\mathrm{LP}(g) = o_1 \dots o_n g_1 \dots g_l$, then the overlap is $o_1 \dots o_n$, $s = g_1 \dots g_l$ and $p = f_1 \dots f_k$. Then our SPol is equal to $fs - pg$.

Our motivation for constructing these bases stems from the process of checking if

Algorithm 3 Generation of SPols

Input: two polynomials f and g of a path algebra P
Precondition: f and g must be monic
Output: every SPol associated with f and g

$S \leftarrow \{\}$
$O \leftarrow$ set of all overlaps between $\mathrm{LP}(f)$ and $\mathrm{LP}(g)$ such that $\mathrm{LP}(f)$ is on the left.
while O is non-empty **do**
    $o \leftarrow$ any element in O
    $p \leftarrow$ everything in $\mathrm{LP}(f)$ up to o
    $s \leftarrow$ everything in $\mathrm{LP}(g)$ after o
    $S \leftarrow S \cup \{f \cdot s - p \cdot g\}$
end while
$O \leftarrow$ set of all overlaps between $\mathrm{LP}(f)$ and $\mathrm{LP}(g)$ such that $\mathrm{LP}(f)$ is on the right.
while O is non-empty **do**
    $o \leftarrow$ any element in O
    $p \leftarrow$ everything in $\mathrm{LP}(g)$ up to o
    $s \leftarrow$ everything in $\mathrm{LP}(f)$ after o
    $S \leftarrow S \cup \{p \cdot f - g \cdot s\}$
end while

some element is inside an ideal. Given a set of generators for an ideal I and some potential element f, it can be impractical to see if f is contained in I because we have to find out what combination of generators can generate it and how. However, we can write f as $f = (\sum_{i=1}^{n} l_i g_i r_i) + \overline{f}^G$ using our division algorithm. If G is a Gröbner basis of I and $\overline{f}^G \neq 0$, no term in $\overline{f}^G$ is the lead term of any element of I by the construction of G, in particular $\mathrm{LT}(\overline{f}^G)$ is not. But, $f \in I \Rightarrow \overline{f}^G \in I$, so if $f \in I$ then $\overline{f}^G$ must be 0, so this gives an easy way of checking if any given f is in our ideal I.

Chapter 3: Data structures for Path Algebras

In this chapter we will lay out most of our data structures and objects used in our code. Several choices were made for the purposes of simplicity and ease of use. Each minor discrepancy will be explained with its corresponding description, while major ones will be addressed upfront.

The most common change from the underlying literature is the fact that we use `PathID`s, `EdgeID`s, and `VertexID`s instead of paths, edges, and vertices. While each of those specific objects exist in our program, we choose to pass around what we chose to call their IDs. These IDs are the index of our desired object inside of the `vector` that stores all objects of that type. This means we have to spend less space storing objects when they appear multiple times.

Besides the use of IDs, you will notice that our `PathTable` is an entire object instead of just a `vector` of `Path`s like our `edgeList` and `vertexList`. This is because we often need to check if a path already exists when we generate it, unlike edges or vertices which are fixed when the `PathAlgebra` is defined. Because of this, it consists of a `vector` of `Path`s and an `unordered_map`[1] with values of `PathID`s and keys of `Path`s. Using the `vector`, we can use a `PathID` to get a desired `Path`, and using the `unordered_map` we can use a generated `Path` to get its corresponding `PathID`. Notably, if no `PathID` is found this means it is a new path and should be added to our `unordered_map`.

3.1 Graph

- vertexList
- edgeList

Graph is a simple class with just two vectors that hold our Vertexs and Edges respectively. Edges contain their connection data and labels while Vertexs store their label, so there isn't much else to store here. This object is generated with an adjacency matrix, and also allows optional lists of labels for the Vertexs and Edges. When not given labels, it automatically labels Edges as "e" concatenated with their EdgeID, and Vertexs are "v" concatenated with their VertexID.

3.2 Path

- mStartVertex, mEndVertex, integers that are VertexIDs
- mPathID, the ID of this Path
- mPath, vector containing the ID's of its edges in order
- mIsZero, a boolean denoting the zero path.
- mIsVertex, a boolean denoting a trivial path about vertex

Path is a representation of a path through our underlying graph. We chose keep Path separate from Graph because Path never needs to access any members or functions from Graph directly and only stores the IDs of Edges and Vertexs. The mIsVertex in our Path is used as a flag to short-circuit various functions that might break or have unwanted behavior if called on one of our Paths representing a trivial path. The last notable thing that makes this different from the graph theoretic notion of a path

is the `mIsZero` boolean. In a path algebra there is no notion of a "zero path", the closest thing being the trivial path about a vertex but that is still a very distinct idea. This is simply used to represent the product of two `Path`s where the right doesn't begin where the left ends, and is necessary because this occurs often when we multiply `PAElements` and we need a way to tell the program not to add that term to our new `PAElement`

3.3 PathTable

- `mPathDictionary`, a `vector` of `Path`s
- `mReversePathDictionary`, an unordered map with Path's as keys and `PathID`'s as values

`PathTable` is the structure we use for storage and fast reference of our `Path`s. When we construct a new path we always add it to the dictionary, and so we need to keep `mReversePathDictionary` in order to quickly check if a path exists already, otherwise we would have to check every single path for equality. Notably, `mPathID` is always set to the `Path`'s index in our vector, and so `mReversePathDictionary` gives us the index in `mPathDictionary`.

3.4 PathOrder

- `mEdgeWeights`, a `vector` of weights
- `mEdgeOrder`, a `vector` of ranks for edges, each index is a unique integer from 1 to n where n is the number of edges, and the index corresponds to `EdgeID`s.
- `mWeightLength`, the dimension of the edge weights

- `mHasWeights`, just a boolean telling us if there are even weights, if not we want to just to lexicographic compares.

`PathOrder` is what we use to enforce the desired ordering on our `Path`s. All of our weights are held here because it is the only object that will ever need them, and similarly for our edge ranks. Besides our weights and ranks, this object contains functions that take in any two paths and tell us which is larger or if they are equal.

3.5 PAElement

- `polynomial`, a `vector` of `Term`s, which are pairs of field element coefficients and `Path`s.

`PAElement` represents a polynomial in our path algebra and is basically synonymous with it, it is a wrapper for a `vector` with some very basic functions on top of it. It notably has constructors for making `PAElement`s out of sections of others, making a single term polynomial with a given `Path` and `FieldElement`, as well as for preconstructed lists of `Path`s and `FieldElement`s. Importantly, this `vector` is sorted using `PathOrder`.

3.6 PathAlgebra

- `mGraph`, the graph that our path algebra is based on
- `mField`, the field our coefficients come from
- `mPathOrder`, the structure that defines the ordering of our polynomials
- `mPathTable`

This is the primary object in our program. We generate a `PathAlgebra` with everything necessary to construct the needed members, like an adjacency matrix for `mGraph` and a characteristic for `mField`. Every major interaction between `PAElement`s occurs inside of this object.

3.7 `OverlapInfo`

- `leftIndex, rightIndex`, two integers that represent the indexes of the corresponding polynomials in the `vector` they are in. `leftIndex` is the polynomial who has the overlap as a suffix, and `rightIndex` has the overlap as a prefix.
- `overlapLocation`, the location of the beginning of the overlap in `leftIndex`s corresponding polynomial.
- `lcmSize`, the size of the least common multiple of both lead paths.
- `overlapSize`, the size of the overlap itself.

Instead of storing our SPols as `PAElement`s, it is more efficient to store them as `OverlapInfo`s, which essentially a collection of integers that describes an SPol. We use this to avoid creating `PAElement`s we may never end up using.

Chapter 4: Functions and some Corresponding Algorithms

4.1 Path Ordering Comparisons

Like we talked about in Section 2.2, Orderings on Path Algebras, every time we need to consider a polynomial it should be sorted for algorithmic purposes. For that to happen, we need to define some functions for comparing two paths to each other with respect to our chosen (admissible) ordering. Very importantly however, we only need to compare `Path` elements because the `PAElements` constructor correctly adds `Terms` based on the results of each comparison.

4.1.1 General Comparison

```
Compare PathOrder::comparePaths(const Path& path1, const Path& path2
    ) const {
    if(path1.mPathID != -1 && path2.mPathID != -1 && path1.mPathID ==
     path2.mPathID)
      return Compare::EQ;

    if (mHasWeights)
    {
       return weightCompare(path1, path2);
    }
    return lengthLexCompare(path1,path2);
}
```

This comparison takes in two `Paths` and decides which version of the comparison to use. When the two paths have valid `PathIDs` that are equivalent, it returns the equality flag. If not, it checks if the ordering uses weights and if so, returns the weight comparison between to. If they have no weights, it simply does a length-lexicographic comparison.

4.1.2 Weight Comparison

```
Compare PathOrder::weightCompare(const Path& path1, const Path&
    path2) const {
  WeightVector weightVector1 = pathWeight(path1);

  WeightVector weightVector2 = pathWeight(path2);

  for(int i = 0; i < weightVector1.size(); ++i)
  {
    if (weightVector1[i] > weightVector2[i])
      return Compare::GT;
    else if (weightVector1[i] < weightVector2[i])
      return Compare::LT;
  }
  return lengthLexCompare(path1, path2);
```

This function is an implementation of the component wise comparison decided upon in Section 2.2. We go through each element of our weight `vectors` until they are non-equal, then return `GT` or `LT` based on if the first or second `Path` had the larger component. In the event we make it to the end of our comparisons and they are still equal, we just go to a length-lexicographic comparison as a fallback.

4.1.3 Length-Lexicographic Comparison

```
Compare PathOrder::lengthLexCompare(const Path& path1, const Path&
    path2) const {
  if (path1.length() > path2.length()) return Compare::GT;
  if (path1.length() < path2.length()) return Compare::LT;
  std::vector<EdgeID> path1List = path1.getEdgeList();
  std::vector<EdgeID> path2List = path2.getEdgeList();

  for(int i = 0; i < path1.length(); i++)
  {
    if(path1List[i] < path2List[i])
      return Compare::GT;
    if(path1List[i] > path2List[i])
      return Compare::LT;
  }

  return Compare::EQ;
}
```

The first thing we do is compare the lengths of our paths and then return the less

than or greater than flag depending on if one is larger than the other. Otherwise, we fall back to the lexicographic comparison. For this lexicographic comparisons, we iterate over our `EdgeID vectors` simultaneously and compare the components, we return the first one with a smaller `EdgeID` as larger because its edge comes first. If two paths have no lexicographic differences, they must be the same `Path`.

4.2 Functions for Handling `Paths`

This section is about functions whose primary function involves creating or managing `Paths`. There are later functions that operate on `Paths` as well – like the ones from Sections 4.4.2 and 4.5.2 – but they do not directly create or place them and their discussion is more relevant in their given sections.

4.2.1 Multiplication of `Paths`

The only operation that happens with paths is the multiplication of two paths together. Though it is the only thing we do with them, we do it constantly. Almost everything we do uses the multiplication of `Paths` in some way, and so it is very important that this is done efficiently, and so we sacrificed some level of memory to have everything accessible to do simple checks and then immediately concatenate the paths. This function takes in our two paths and returns the `PathID` of the product. There are 3 wrappers for this function that take `PathIDs` as the left or right input or as both of the inputs and simply call this function after accessing the corresponding path.

```
1 PathID PathAlgebra::multiplyPaths(const Path& path1,
2                                   const Path& path2){
3   if (path1.mIsVertex){
4     if(path1.mEndVertex == path2.mStartVertex){
5       return mPathTable.findOrAdd(path2);
6     } else {
7       return PathID(0); // ID of the empty (i.e. zero) path.
8     }
```

```
9   }
10
11  if (path2.mIsVertex){
12    if(path1.mEndVertex == path2.mStartVertex){
13      return mPathTable.findOrAdd(path1);
14    } else {
15      return PathID(0); // ID of the empty path.
16    }
17  }
18
19  if (!(path1.mStartVertex == -1 || path2.mStartVertex == -1) &&
20      path1.mEndVertex == path2.mStartVertex){
21    Path newPath = Path(path1, path2); // concatenation constructor
22
23    return mPathTable.findOrAdd(newPath);
24  } else {
25    return PathID(0); // ID of the empty path.
26  }
27 }
```

The first important part is our checks on lines 3 and 11, which ask if either of our inputs is a trivial path. If either path is trivial and about the corresponding start or end vertex – depending if it is on the left or right respectively – then we just want to return the other `PathID`. After this, we first check if either of these is the "zero path". While this notion is more or less meaningless in a real path algebra, we need a way to represent an empty path. If neither path is the zero path, we check if the right path starts where the left path ends, and if so we return the two paths concatenated. If either one of our paths is the zero path or the respective start and end vertices don't agree, we return the zero path. As commented in the last line, `0` is the `PathID` of the zero path, as it is always the first thing generated in our `PathTable`.

4.2.2 `findOrAdd`

Most times we generate a new `Path`, primarily through multiplying but occasionally through raw `vector` creation, we want to assign a `PathID` to that `Path` for ease of passing around or later reference. To do that, we need a function that searches the `PathTable` and either finds the `Path` we just created or adds the new `Path` to the

table.

```
PathID PathTable::findOrAdd(const Path& path) {
  auto lookup = this->mReversePathDictionary.find(path);
  PathID newPathID;
  if(lookup == this->mReversePathDictionary.end()) {
    newPathID = addToTable(path);
  } else {
    newPathID = lookup->second;
  }
  return newPathID;
}
```

We start this function by searching the `unordered_map` in `PathTable` for the value corresponding to our input `Path`. If nothing is found we know that this is a new `Path` we have not seen before and so we can add it to the `PathTable`. Finally, we return the `PathID` of our path, which is either the ID of the one in the table or a new one generated by `addToTable`, which just puts it on the end with the first unused integer and sets the ID inside the `Path` as well.

4.3 Operations for `PAElements`

`PAElements` are the primary objects we use, as they represent the polynomial elements of our path algebra. Because of that, the functions we have to perform operations on them are some of the most used functions in our project. This includes addition, subtraction, and multiplication. This also involves a notion discussed previously in Section 2.3 which is division of a polynomial by a set, and specifically getting the remainder from this division. Each of these operations will be discussed in depth and algorithmically below.

4.3.1 Addition and Subtraction

For addition and subtraction, we have two functions, `add` and `subtract`, with the only difference between the two being a negation on the terms in the right `PAElement` every time we add an element from it. Because the two functions are so similar algorithmically, we will only go over the addition function.

The `add` method takes in two `PAElements` and algorithmically adds them together `Term` by `Term`, combining `Terms` with the same `Path` and maintaining the ordering.

```
1  void PathAlgebra::add(PAElement &result, const PAElement &f, const
       PAElement &g) {
2
3    assert(result.polynomial.size() == 0);
4
5    // this should compare path weights at each step and add which is
       heavier
6    auto fit = f.polynomial.begin();
7    auto git = g.polynomial.begin();
8
9    while (fit != f.polynomial.end() && git != g.polynomial.end()) {
10     const Path& fPath = this->mPathTable.mPathDictionary[int(fit->
       pathID)];
11     const Path& gPath = this->mPathTable.mPathDictionary[int(git->
       pathID)];
12
13     // if fPath heavier, add from f
14     Compare comp = mPathOrder.comparePaths(fPath,gPath);
15
16     switch (comp) {
17       case Compare::GT:
18         result.polynomial.push_back({fit->coeff,fit->pathID});
19         ++fit;
20         break;
21       case Compare::LT:
22         result.polynomial.push_back({git->coeff,git->pathID});
23         ++git;
24         break;
25       case Compare::EQ:
26         FieldElement sum = mField.add(fit->coeff,git->coeff);
27         // if our sum is 0, then we dont want to add it and just
       want to move on.
28         if(!sum.isZero())
29           result.polynomial.push_back({sum,fit->pathID});
30         ++fit;
31         ++git;
32         break;
```

```
    }
  }

  // add everything from f not already used
  while (fit != f.polynomial.end()) {
    result.polynomial.push_back({fit->coeff,fit->pathID});
    ++fit;
  }
  // add everything from g not already used
  while (git != g.polynomial.end()) {
    result.polynomial.push_back({git->coeff,git->pathID});
    ++git;
  }
}
```

Algorithm 4 Polynomial Addition

Input: two sorted polynomials F and G
Output: the sum s

$f \leftarrow \mathrm{LP}(F)$
$g \leftarrow \mathrm{LP}(G)$
$s \leftarrow 0$
while F and G are non-zero **do**
    if $g < f$ **then**
        $s \leftarrow s + \mathrm{LT}(F)$
        $F \leftarrow F - \mathrm{LT}(F)$
    else if $f < g$ **then**
        $s \leftarrow s + \mathrm{LT}(G)$
        $G \leftarrow G - \mathrm{LT}(G)$
    else
        $s \leftarrow s + \mathrm{LT}(F) + \mathrm{LT}(G)$
        $G \leftarrow G - \mathrm{LT}(G)$
        $F \leftarrow F - \mathrm{LT}(F)$
    end if
end while
while F is non-zero **do**
    $s \leftarrow s + \mathrm{LT}(F)$
    $F \leftarrow F - \mathrm{LT}(F)$
end while
while G is non-zero **do**
    $s \leftarrow s + \mathrm{LT}(G)$
    $G \leftarrow G - \mathrm{LT}(G)$
end while

The main difference between our algorithm and code is the way we iterate over

our polynomials. In the algorithm, we simply remove the lead term each time so that our new lead term is what we want. This option is chosen for the algorithm for the sake of notational simplicity.

4.3.2 Multiplication

```
1 void PathAlgebra::multiply(PAElement &result, const PAElement &f,
    const PAElement &g)
2 {
3   assert(result.polynomial.size() == 0);
4   if (f.polynomial.size() < g.polynomial.size())
5      multiplyShortLeft(result, f, g);
6   else
7      multiplyShortRight(result, f, g);
8 }
```

Our general multiplication function just takes two polynomials and determines which actual multiplication function is correct and calls it. As `multiplyShortLeft` and `multiplyShortRight` are basically the same, we will explore `multiplyShortLeft` from line 5 to explain our multiplication.

```
1 void PathAlgebra::multiplyShortLeft(PAElement &result, const
    PAElement &shortPoly, const PAElement &longPoly)
2 {
3    assert(result.polynomial.size() == 0);
4    std::vector<PAElement> tempVec;
5    for (const auto &t : shortPoly.polynomial)
6    {
7       PAElement tempElt;
8       leftMultiply(tempElt, t.pathID, t.coeff, longPoly);
9       tempVec.push_back(tempElt);
10   }
11   add(result,tempVec);
12 }
```

The purpose of having two separate multiply functions comes from a problem we will encounter in Section 6.5, where it is faster to distribute a smaller polynomial into a larger one because we have to sort the result, and less polynomials that are longer is faster than more short polynomials. Because our `add` function allows us to pass a `vector` of `PAElements` to add together, the easiest way to multiply is to use the

distributive property for each `Term` of `shortPoly` and add the resulting product to a `vector`, and then pass that `vector` and our empty `result` into the `add` function.

4.3.3 Exponentiation

```
void PathAlgebra::exponent(PAElement &result, const PAElement &f,
    long n) {
  // exponents must be nonnegative
  assert(n >= 0);
  switch (n) {
    case 0:
      result = one();
      break;
    case 1:
      result = f;
      break;
    default:
      result = f;
      for (auto i = 1; i < n; ++i){
        PAElement tmp;
        multiply(tmp, result, f);
        result = tmp;
      }
  }
}
```

Our exponentiation function takes in just a `PAElement` and an integer n to raise the exponent to. If $n = 0$, we can just return the identity polynomial, and if $n = 1$ we similarly just return our initial polynomial. Finally, if we have any non-trivial exponent, we multiply our initial polynomial by an intermediary value n times and return the final result of that. This method is very straightforward, and there may be faster ways to handle exponentiation, but we will talk about that in Section 6.5.

4.4 Long Division for `PAElements`

To implement the function corresponding to Algorithm 1 we had to implement several functions that were just for its use, like the one discussed in Section 4.4.2. However we also needed to reuse several functions written for general operations in `PathAlgebra`.

While some of these were fine to use as is, there was one we could optimize using assumptions that can be made about the inputs given in `dividePAElement`. We multiply on both sides often in this division algorithm and we could use the normal `multiplyPaths` function twice for this, but because of assumptions we know our `Paths` start and end at the same place in a polynomial, we can skip the checking of compatibility and immediately concatenate our `Paths`. `divisionAlgorithm_twoSidedMultiplyPaths` does this for us, it's not called directly but we use `divisionAlgorithm_twoSidedMultiply` to iterate over our polynomial and call it on every term.

4.4.1 `dividePAElement`

This is the function that performs division for `PAElements`. It returns a `PAElement` that is the remainder of our division, and takes as input a collection of `PAElements` that are our potential divisors and a single `PAElement` that is our dividend.

```
PAElement PathAlgebra::dividePAElement(const std::vector<PAElement>
    divisors, const PAElement dividend){
  PAElement curDividend = dividend;
  PAElement remainder;
  std::vector<PathID> divisorLTs = {};
  for(auto itr : divisors)
    divisorLTs.push_back(itr.polynomial[0].pathID);

  while(curDividend.polynomial.size() != 0){
    std::pair<int, PathID> subword = isAnySubword(divisorLTs,
   curDividend.polynomial[0].pathID);
    if(subword != (std::pair<int,int>){-1,-1}){
      // build prefix + suffix from curDividend.polynomial[0].pathID
      auto curLTID = curDividend.leadTermID();
      auto curLeadTerm = mPathTable.mPathDictionary[curLTID];
      auto foundPath = mPathTable.mPathDictionary[divisorLTs[subword
   .second]];

      auto preEnd = curLeadTerm.begin() + subword.first;
      std::vector<EdgeID> prefix(curLeadTerm.begin(), preEnd);

      auto sufBeg = curLeadTerm.begin() + subword.first + foundPath.
   length();
      std::vector<EdgeID> suffix(sufBeg, curLeadTerm.end());

      curDividend = divisionAlgorithm_subtract(curDividend,
```

```
23                    divisionAlgorithm_twoSidedMultiply(prefix,
24                         curDividend.polynomial[0].coeff,
25                         divisors[subword.second],
26                         suffix,
27                         curLeadTerm.mStartVertex,
28                         curLeadTerm.mEndVertex));
29       }
30       else{
31         remainder.polynomial.push_back(curDividend.polynomial[0]);
32         curDividend.polynomial.erase(curDividend.polynomial.begin());
33       }
34     }
35     return remainder;
36   }
```

This function closely follows Algorithm 1. Two notable differences are that, on line 2, we make a copy of our dividend because we are going to want to modify it all the way through, and, on lines 4 to 6, we make a new list of divisors that are just the lead terms for ease of access. The section on lines 16 and 17 is the creation of p from Algorithm 1, line 6 and lines 19 and 20 are the creation of s from Algorithm 1, line 7. Also note that on line 9 we return a `pair`, the first value is the subwords location in the superword, while the second value is the ID of the `Path` that is a subword. Finally, the long function call starting at line 22 is equivalent to the $d \leftarrow d - p * g * s$ on line 8 of Algorithm 1.

4.4.2 `isAnySubword`

Our subword detection function takes a `PathID` for our superword that it searches through and a `vector` of `PathID`s for our potential subwords. If a subword is found, we return a pair whose first element is the index in the superword the subword starts, and the second element being the index into our input `vector` that the subword is found at. If nothing is found, we return the pair $\{-1, -1\}$ to signify that nothing was found.

```
1  std::pair<int,int> PathAlgebra::isAnySubword(const std::vector<
       PathID>& subIDDict, const PathID& superPathID){
```

```
  std::vector<EdgeID> word = mPathTable.mPathDictionary[superPathID
   ].getEdgeList();
  size_t wordLen = mPathTable.mPathDictionary[superPathID].length();
  for (int i = 0; i < wordLen; i++){
    for (int j = 0; j < subIDDict.size(); ++j){
      if(wordLen-i >= mPathTable.mPathDictionary[subIDDict[j]].
   length() &&
         word[i] == mPathTable.mPathDictionary[subIDDict[j]].mPath
   [0] &&
         std::equal(word.begin()+i,
                    word.begin()+i+mPathTable.mPathDictionary[
   subIDDict[j]].length(),
                    mPathTable.mPathDictionary[subIDDict[j]].mPath.
   begin()))
        return {i,j};
    }
  }
  return {-1,-1};
}
```

In the process of checking for our subwords, we iterate over our superword exactly once, and at every index we check through our vector to see if any of the paths we have start with the value at our current index. If they do, we do a memory compare between the memory regions containing our vectors, and if they are equal we return the index where our subword starts and that subwords index in our `vector` of `PathID`s. Otherwise, we go through the rest of our paths until we reach the end, then go to the next entry of our superword. While there are many subword detection algorithms, nothing is particularly practical for checking a bank of subwords against a single superword, they tend to search banks of superwords for a single subword.

4.5 Buchberger's Algorithms

4.5.1 Buchberger's Algorithm

`Buchbergers` takes in a `vector` of `PAElements` called `generators` that represents the generators of some ideal I, as well as an integer `maximumSize` which is meant to be a safeguard for the common case that our Gröbner basis will be infinite and therefore `Buchbergers` will never terminate. If that happens we will simply exit once we have the maximum basis size desired. Also of note is that the `maximumDegree` parameter is defined in the function declaration to be a default parameter with value 0, and so it may not appear in all function calls.

```
std::vector<PAElement> PathAlgebra::Buchbergers(
            const std::vector<PAElement> &generators,
            int maximumSize,
            int maximumDegree) {
  std::vector<PAElement> newGenerators = generators;

  std::priority_queue<OverlapInfo, std::vector<OverlapInfo>,
   OverlapCompare> overlapQueue;

  for(int i = 0; i < generators.size(); i++){ // iterate over the
   generator list fully
    for(int j = 0; j < generators.size(); j++){ // iterate over the
   generator list up to the outer loops position
      std::vector<OverlapInfo> newOverlaps =
   Buchbergers_processOverlaps(generators, i, j);
      for(auto overlap : newOverlaps){
        overlapQueue.push(overlap);
      }
    }
  }

  while(!overlapQueue.empty() && (newGenerators.size() < maximumSize
    || maximumSize == 0)){
    PAElement remainder = divideOverlap(newGenerators, overlapQueue.
   top());
    overlapQueue.pop();
    if(remainder.polynomial.size() > 0) {
      size_t newIndex = newGenerators.size();
      makeMonic(remainder);
      newGenerators.push_back(remainder);
      for(int i = 0; i < newIndex; i++){
```

```
26          std::vector<OverlapInfo> newOverlaps1 =
    Buchbergers_processOverlaps(newGenerators, i, newIndex);
27          for(auto overlap : newOverlaps1)
28          overlapQueue.push(overlap);
29          std::vector<OverlapInfo> newOverlaps2 =
    Buchbergers_processOverlaps(newGenerators, newIndex, i);
30          for(auto overlap : newOverlaps2)
31          overlapQueue.push(overlap);
32        }
33        std::vector<OverlapInfo> newOverlaps3 =
    Buchbergers_processOverlaps(newGenerators, newIndex, newIndex);
34        for(auto overlap : newOverlaps3)
35          overlapQueue.push(overlap);
36      }
37    }
38
39    return newGenerators;
40  }
```

This function is the entire purpose of our project and is more or less the same as outlined in Algorithm 2. The only key things to note are certain choices made to adequately implement our algorithm. In line 7 we define a priority queue that will go on to hold all of our overlaps, we chose a `priority_queue` because the `OverlapInfos` we add to this list should be in a particular order for optimal generation of the Gröbner basis and a priority queue allows us to always keep the smallest overlap immediately on hand. Smaller overlaps are preferred because the associated SPol is larger and therefore more likely to have subwords found.

Next, the nested for loops on lines 9 to 16 iterate over every combination of polynomials in `generators` and adds their corresponding `OverlapInfos` to the `priority_queue`. The `Buchbergers_processOverlaps` function is important in its own right however, so it will be explained in depth later, the main thing is it just returns a `vector` detailing every possible overlap between two paths. All of this generates our initial set of overlaps to begin working through with Buchberger's Algorithm. It's noteworthy that the actual algorithm generates our SPols and stores them, not just our overlaps. We choose to store just the overlap information because you can

generate the SPol from there, but it's also possible we don't use all of the overlaps so we don't want to waste resources computing them until they are needed.

Now that there are some overlaps to check, we enter the while loop on lines 18 to 37. This while loop goes until our queue is empty which is the designed termination of Buchberger's Algorithm, but as described above it will also terminate once we get to the maximum size of `newGenerators` desired. Once the loop starts, we get the remainder of dividing the first overlap by the `newGenerators` set and remove that overlap from the queue. From here we follow Algorithm 2 exactly. If the remainder is non-zero we add it to `newGenerators` and then generate every possible overlap between it and the entire set including itself, adding them to our queue, then repeating our loop. If our remainder is 0, then we just do nothing and go to the next element of the queue.

4.5.2 `Buchbergers_processOverlaps` and `findOverlaps`

```
1  std::vector<OverlapInfo> PathAlgebra::Buchbergers_processOverlaps(
       const std::vector<PAElement> &list,
2                  const int &leftIndex,
3                  const int &rightIndex){
4    PathID firstLT = list[leftIndex].polynomial[0].pathID;
5    PathID secondLT = list[rightIndex].polynomial[0].pathID;
6
7    size_t firstLen = mPathTable.mPathDictionary[firstLT].mPath.size()
       ;
8    size_t secondLen = mPathTable.mPathDictionary[secondLT].mPath.size
       ();
9
10   std::vector<int> overlapLocations = findOverlaps(firstLT, secondLT
       );
11
12   std::vector<OverlapInfo> overlaps;
13
14   for(auto overlap : overlapLocations){
15     overlaps.push_back({leftIndex, rightIndex, overlap, secondLen +
       overlap, firstLen - overlap});
16   }
17   return overlaps;
18 }
```

`Buchbergers_processOverlaps` is mainly a wrapper for `findOverlaps` at line 10 that takes overlap locations found and puts them into `OverlapInfo` structs. The actually interesting function here is `findOverlaps` below.

```
1  std::vector<int> PathAlgebra::findOverlaps(const PathID& prefix,
       const PathID& suffix) const{
2    std::vector<int> locations = {};
3    // starting at i = 1 so as to only look for proper prefixes
4    for (int i = 1; i < mPathTable.mPathDictionary[prefix].length(); i
       ++){
5      // changed to strict here to only look for proper suffixes
6      if((mPathTable.mPathDictionary[prefix].length()-i) < mPathTable.
       mPathDictionary[suffix].length() &&
7          std::equal(mPathTable.mPathDictionary[suffix].mPath.begin(),
8                     mPathTable.mPathDictionary[suffix].mPath.begin()+(
       mPathTable.mPathDictionary[prefix].length()-i),
9                     mPathTable.mPathDictionary[prefix].mPath.begin()+i
       ))
10        locations.push_back(i);
11   }
12   return locations;
13 }
```

`findOverlaps` takes in two `PathID`s and iterates over the prefix path. As long as there are less elements to the left of our current index than are entirely inside of our suffix, we do an equality check on chunks of memory equal in size to the remainder of our prefixes `edgeList vector` that are at the end of our prefix and beginning of our suffix. If the result is equality, then we push the current location into a `vector` and continue. Once we're at the end of our prefixes `edgeList vector` we return that list of starting locations.

Back in `Buchbergers_processOverlaps`, once we've returned that `vector` of overlap locations, we iterate over it and combine those locations with other information about the overlap and place it in another `vector` that gets returned to `Buchbergers`.

Chapter 5: Usage

Now that we have gone over what the major functions in our code do, let us go through an example of generating a Gröbner basis from downloading the code to reading the finished basis. As provided above, the code can be found here (`https://github.com/xZecora/path-algebras`) on Github, and you can use whatever your preferred method of getting code from Github is to download it. The only assumption we will make is that you have `glibc`, `g++`, and a way to use a `Makefile`, typically `cmake`.

Now that we have a directory with our code, lets go through everything needed to generate a Gröbner basis, for this example we will use the defining ideal of a Sklyanin algebra, an ideal generated by three polynomials:

- $f_1 = \alpha yz + \beta zy + \gamma x^2$,
- $f_2 = \alpha zx + \beta xz + \gamma y^2$,
- $f_3 = \alpha xy + \beta yx + \gamma z^2$.

A Sklyanin algebra is defined specifically by $k\langle x, y, z\rangle/\langle f_1, f_2, f_3\rangle$, see [2]. Here, x, y, and z can be conveniently represented by the edges in the very basic graph from Figure 5.1.

To begin, make a file called `example.cpp` that you can edit and open it in your preferred text editor. Now that we have a file to work from, place the below code exactly as is inside of the file:

```
#include <iostream>
#include "Field.hpp"
#include "Graph.hpp"
#include "PathAlgebra.hpp"
#include "PAElement.hpp"
```

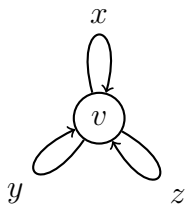


Figure 5.1: Our basic example of a graph

```
6
7 int main(int argc, char** argv){
8 }
```

This is all standard C++ boiler plate, it includes the header files from our code we wrote as well as for printing out things to our screen. Please note that all future code will go inside the bracket of our `main` function. Next, we need to generate a `PathAlgebra` to actually work inside of, for this case our graph will be a single vertex with three loops about it, and no meaningful weights, like the one draw in figure 5.1. Our field is arbitrarily chosen with characteristic 101, any prime number will suffice in this position, it was an arbitrary choice and only effects are coefficients and occasionally a term may disappear because its coefficient becomes 0.

```
8   Field myField { 101 };
9
10  Graph myGraph({ std::vector<int> {3} }, {"v"}, {"x","y","z"});
11
12  std::vector<WeightVector> weights = {{1},{1},{1}};
13
14  PathOrder myPathOrder(weights);
15
16  //PathOrder myPathOrder;
17
18  PathAlgebra myPathAlgebra(myGraph, myField, myPathOrder);
```

Our graph is represented in this case by an adjacency where an entry a_{ij} – or `A[i][j]` in C++ where A is our 2-dimensional vector – is the number of edges going from the ith vertex to the jth. The two `vectors` used are the labels for the `Vertex`s and `Edge`s respectively. As stated above, our weights are meaningless because they

are all the same, and so a lexicographic ordering is exactly equivalent because it will always fall back to that. An example with arbitrary weights is included as an example so the user can modify it if desired. Simply replace the `1`s inside the weight vector with the weights corresponding to `x`, `y`, and `z` in that respective order. The weights can also be multidimensional vectors if so desired. However, if you don't want weights associated, you can also define `myPathOrder` as on line 16. Next, we should generate all of our initial path's.

```
20  Path path_x(0,0,{0});
21  Path path_y(0,0,{1});
22  Path path_z(0,0,{2});
23  Path vert(0);
```

Our first three lines generate the actual paths. Their start and end `VertexID`s are each 0, the only vertex, and the vector contains their respective `EdgeID` as each of them is just a single edge.

```
25  PathID id_x2 = myPathAlgebra.multiplyPaths(id_x,id_x);
26  PathID id_y2 = myPathAlgebra.multiplyPaths(id_y,id_y);
27  PathID id_z2 = myPathAlgebra.multiplyPaths(id_z,id_z);
28  PathID id_xy = myPathAlgebra.multiplyPaths(id_x,id_y);
29  PathID id_yx = myPathAlgebra.multiplyPaths(id_y,id_x);
30  PathID id_xz = myPathAlgebra.multiplyPaths(id_x,id_z);
31  PathID id_zx = myPathAlgebra.multiplyPaths(id_z,id_x);
32  PathID id_yz = myPathAlgebra.multiplyPaths(id_y,id_z);
33  PathID id_zy = myPathAlgebra.multiplyPaths(id_z,id_y);
```

Now, to generate the defining ideal of our Sklyanin algebra, we need multiple individual paths, specifically every path which is a combination of `x`, `y`, and `z` including repeated values. This code uses our `multiplyPaths` function from Section 4.2.1 to multiply our single edge paths generated above to create all 9 monomials that will go into the generators.

```
35  PAElement f1({id_x2,id_yz,id_zy},{FieldElement(1),FieldElement(3),
     FieldElement(7)});
36  PAElement f2({id_xy,id_yx,id_z2},{FieldElement(1),FieldElement(36)
     ,FieldElement(34)});
37  PAElement f3({id_xz,id_y2,id_zx},{FieldElement(1),FieldElement(29)
     ,FieldElement(87)});
```

The `PAElements` created here seem to have somewhat arbitrary field elements for coefficients, but we had to order our polynomials while keeping them monic, and so those are what that process results in.

```
39  std::vector<PAElement> sklyGens = {f1,f2,f3};
40  auto sklyGensGB = myPathAlgebra.Buchbergers(sklyGens, 10);
```

Next, we place our generators into a vector and call our implementation of Buchberger's algorithm from Section 4.5.1. The 10 on line 40 is telling our function to max out at a basis size of 10, we use this because Sklyanin has an infinite Gröbner basis and otherwise our algorithm will never terminate.

```
41  for (auto f : sklyGensGB){
42    myPathAlgebra.printPAElementByLabel(std::cout, f);
43    std::cout << std::endl << std::flush;
44  }
```

Our final chunk of code is just the for loop that prints out our entire basis. It uses our `printPAElementByLabel` function to print each element of our Gröbner basis one at a time. When we put all of this together our file should look like this:

```
1  #include <iostream>
2  #include "Field.hpp"
3  #include "Graph.hpp"
4  #include "PathAlgebra.hpp"
5  #include "PAElement.hpp"
6
7  int main(int argc, char** argv)
8  {
9    Field myField { 101 };
10
11   Graph myGraph({ std::vector<int> {3} }, {"v"}, {"x","y","z"});
12
13   PathOrder myPathOrder;
14
15   PathAlgebra myPathAlgebra(myGraph, myField, myPathOrder);
16
17   Path path_x(0,0,{0});
18   Path path_y(0,0,{1});
19   Path path_z(0,0,{2});
20   Path vert(0);
21
22   PathID id_x2 = myPathAlgebra.multiplyPaths(path_x,path_x);
23   PathID id_y2 = myPathAlgebra.multiplyPaths(path_y,path_y);
24   PathID id_z2 = myPathAlgebra.multiplyPaths(path_z,path_z);
```

```
  PathID id_xy = myPathAlgebra.multiplyPaths(path_x,path_y);
  PathID id_yx = myPathAlgebra.multiplyPaths(path_y,path_x);
  PathID id_xz = myPathAlgebra.multiplyPaths(path_x,path_z);
  PathID id_zx = myPathAlgebra.multiplyPaths(path_z,path_x);
  PathID id_yz = myPathAlgebra.multiplyPaths(path_y,path_z);
  PathID id_zy = myPathAlgebra.multiplyPaths(path_z,path_y);

  PAElement f1({id_x2,id_yz,id_zy},{FieldElement(1),FieldElement(3),
   FieldElement(7)});
  PAElement f2({id_xy,id_yx,id_z2},{FieldElement(1),FieldElement(36)
   ,FieldElement(34)});
  PAElement f3({id_xz,id_y2,id_zx},{FieldElement(1),FieldElement(29)
   ,FieldElement(87)});

  std::vector<PAElement> sklyGens = {f1,f2,f3};
  auto sklyGensGB = myPathAlgebra.Buchbergers(sklyGens, 10);

  for (auto f : sklyGensGB){
    myPathAlgebra.printPAElementByLabel(std::cout, f);
    std::cout << std::endl << std::flush;
  }
}
```

This file is also included in the Github, titled `example_full.cpp`. With all of this in our file, we can run `make example` in our directory – or however you prefer to compile code, ensuring that `example.cpp` is what you compile as main – and are left with a binary file, likely called `example` or `a.out`. Simply run this file however your system allows and you should be greeted with the first 10 generators of the Gröbner basis for the Sklyanin algebras defining ideal. Expected output is as follows:

```
1*x*x + 3*y*z + 7*z*y
1*x*y + 36*y*x + 34*z*z
1*x*z + 29*y*y + 87*z*x
1*y*y*z + 12*y*z*y + 14*z*y*y + 8*z*z*x
1*y*y*x + 3*y*z*z + 92*z*y*z + 42*z*z*y
1*y*y*y*y + 65*y*z*y*x + 80*y*z*z*z + 88*z*y*z*z + 6*z*z*y*z + 13*z*
   z*z*y
1*y*z*y*y + 92*y*z*z*x + 68*z*y*y*y + 3*z*y*z*x + 30*z*z*y*x + 64*z*
   z*z*z
1*y*z*y*z*y + 72*y*z*z*y*y + 28*y*z*z*z*x + 22*z*z*y*y*y + 7*z*z*y*z
   *x + 6*z*z*z*y*x + 62*z*z*z*z*z
1*y*z*y*z*z + 89*y*z*z*y*z + 94*y*z*z*z*y + 94*z*y*z*y*z + 50*z*y*z*
   z*y + 24*z*z*y*z*y + 8*z*z*z*z*x
1*y*z*y*z*x + 47*y*z*z*y*x + 30*y*z*z*z*z + 87*z*y*z*y*x + 19*z*y*z*
   z*z + 6*z*z*y*z*z + 73*z*z*z*y*z + 84*z*z*z*z*y
```

Chapter 6: Further Work

6.1 `SumCollector`

When we add together large polynomials we use the fact that they are already sorted to guarantee that the new polynomial will also be sorted. However, it would be nice if we could quickly construct arbitrary polynomials to add together that might save us some time from sorting things in certain places. In theory, we could use a concept called a `SumCollector` to make this possible. Using a `SumCollector`, we can simply dump all terms in a polynomial into a `priority_queue`, which keeps the largest monomial on top, and then pull them out one at a time. Because two things will be sequential if they are equivalent, we can simply check if a term is the same as our current last term, and if it is, we can add them, otherwise it must be new and can just go on the end as a separate term. This gives us a sorted polynomial without having to sort our two polynomials separately before hand. This process has been written algorithmically in Algorithm 5.

6.2 Personal Memory Management and `shared_ptrs` Instead of `PathTable`

The way we currently are handling our memory allocations is by letting C++ do everything for us. While this is standard and a lot of time is spent by others making sure C++ handles memory well, if we take the time to allocate everything by hand and keep common things more quickly accessible, this could increase efficiency in a few places.

One major thing that makes the code difficult to write and read is the accessing of paths. Most objects need a `Path`, they only have the `PathID`. While this is great for

Algorithm 5 `SumCollector` Logic

Precondition: $<$ defines two things with the same path but different coefficients as equivalent

Input: two polynomials f and g, whose terms are sorted with respect to an admissible ordering $<$

Output: the sum of $f + g$

1: $C \leftarrow \{\}$, a partially-ordered set with respect to $<$
2: **for all** terms t in f **do**
3: $\quad C \leftarrow C \cup \{t\}$
4: **end for**
5: **for all** terms t in g **do**
6: $\quad C \leftarrow C \cup \{t\}$
7: **end for**
8: $S \leftarrow$ the 0 polynomial
9: **while** C is non-empty **do**
10: $\quad t \leftarrow$ the largest element of C
11: $\quad S \leftarrow S + t$
12: $\quad C \leftarrow C \setminus \{t\}$
13: **end while**
14: **return** S

memory efficiency as we only have to pass around `int`s, it makes accessing anything an extremely long function call and vector index, for example,

```
mPathTable.mPathDictionary[divisorsLTs[subword.second]]
```

is a particularly egregious example from line 14 in `dividePAElement`.

If, instead of passing around `PathID`s, we passed around `shared_ptr` to `Path`s that had been created, we would have immediate access to a `Path` instead of having to go through our `PathTable`. This is doubly enticing because a `shared_ptr` and `PathID` have the same size in memory, so there is no increase in memory usage. This would take the above `Path` access down to

```
divisorsLTs[subword.second]
```

assuming that `divisorsLTs` is now of type `std::vector<shared_ptr<Path>>`, a `vector` of pointers to `Path` objects, instead of a `vector` of `PathID`s.

Though this gets rid of the annoying access calls, we still have to have our `PathTable` object because we need a quick way to find the `PathID`, or in this case `shared_ptr`, of a `Path`. This would still be well worth the effort however when paired with the above potential improvement of personally handling memory management, as the two ideas go hand in hand.

6.3 Reduced Gröbner Bases

When talking about Groöbner bases and their purposes, we mentioned that the goal of a Gröbner basis is the ability to quickly check if an element is in an ideal or not based on its reduction with respect to the elements of the basis. An obvious improvement to this would be the reduction of a Gröbner basis, a set of generators with the same property that is shorter or in some way easier to check. An additional property of reduced Gröbner basis is that they are unique for a given idea, so only using reduced Gröbner basis would make checking the equality of ideals trivial, because reduced Gröbner basis equality is ideal equality. While not necessary because our code gives a usable Gröbner basis, the implementation of a reduction algorithm would potentially make the output more concise or easier to use in some circumstances.

6.4 Dynamic Input

Currently our code is run purely off of what we manually generate in our `main.cpp` file. Ideally, we would be able to take either a text file or hand typed data as input and generate everything dynamically from there, as well as waiting for the user to give type in commands to perform operations on our algebra. While this is the goal in the long run and would surely be done with enough time, it wasn't practical within the time frame of what we had. The hardest part of implementing this would likely be the parsing of user input, as everything else would just be passing that information

to the places they need to go.

6.5 Exponentiation

When handling exponents, both in programming and in general, it can be easier to use only keep track of exponents that are powers of two, for example: $5^{13} = 5^1 \cdot 5^4 \cdot 5^8$, as this can be done with 5 multiplications instead of 12. This works well for things like integers because every multiplication is more or less equally complex. However, we are only multiplying polynomials, and this makes multiplications linearly more costly every time you increase the size of one of the polynomials. While the same amount of multiplications happens between $(x + y)^{13}$ being treated as 12 binomial multiplications and it being treated as $(x + y)^1 \cdot (x + y)^4 \cdot (x + y)^8$ gives me the same amount of multiplications, 2^{13}, the process of sorting the two is vastly different, as every time we sort the binomial multiplication we sort two polynomials of equal length, as opposed to 2 polynomials of equal length and then 32 polynomials of equal length. The sorting of the 32 polynomials simply takes too long with our current method of adding. This might be improved with a proper implementation of the `SumCollector` object discussed earlier or a potentially more intelligent addition using assumptions about the distributive property.

Bibliography

[1] *cppreference unordered_map*, https://en.cppreference.com/w/cpp/container/unordered_map, Accessed: 2025-04-07.

[2] M. Artin, J. Tate, and M. Van den Bergh, *Some algebras associated to automorphisms of elliptic curves*, The Grothendieck Festschrift, Vol. I, Progr. Math., vol. 86, Birkhäuser Boston, Boston, MA, 1990, pp. 33–85. MR 1086882

[3] Bruno Buchberger, *An algorithm for finding the basis elements of the residue class ring of a zero dimensional polynomial ideal*, J. Symbolic Comput. **41** (2006), no. 3-4, 475–511, Translated from the 1965 German original by Michael P. Abramson. MR 2202562

[4] Edward L. Green, *Noncommutative Gröbner bases, and projective resolutions*, Computational methods for representations of groups and algebras (Essen, 1997), Progr. Math., vol. 173, Birkhäuser, Basel, 1999, pp. 29–60. MR 1714602

Curriculum Vitae

BRYANT COLLINS

445 Foster Rd. ◇ Cleveland, North Carolina 27013
(704) · 798 · 2943 ◇ bryanthcollins@gmail.com

EDUCATION

Wake Forest University *May 2025*
M.S. in Mathematics

Western Carolina University *December 2022*
B.S. in Mathematics
Minor in Computer Science

RESEARCH EXPERIENCE

Master's Thesis Research February 2023 - April 2025
Department of Mathematics, Wake Forest University *Winston-Salem, NC*

- Topic: Non-Commutative Algebras, Gröbner Bases, and Path Algebras.
- Advisor: Prof. W. Frank Moore, Department of Mathematics, Wake Forest University

Undergraduate Research February 2021 - November 2022
Department of Mathematics, Western Carolina University *Cullowhee, NC*

- Topic: Number Theory, Schur Numbers, and Rado Numbers.
- Advisor: Prof. Mark Budden, Department of Mathematics, Western Carolina University

TEACHING AND WORK EXPERIENCE

Grader February 2023 - April 2025
Department of Mathematics, Wake Forest University *Winston-Salem, NC*

- Calculus I (MTH 111) *Spring 2024*

Math & Stats Center Tutor February 2023 - April 2025
Department of Mathematics, Wake Forest University *Winston-Salem, NC*

- Courses Tutored: Calculus I/II/III, Linear Algebra, Graduate Abstract Algebra I/II, Discrete Mathematics.

Instructor/Teaching Assistant February 2023 - December 2024
Department of Mathematics, Wake Forest University *Winston-Salem, NC*

- Introduction to Calculus (MTH 104L) *Fall 2023, Fall 2024*

Math & Stats Center Tutor February 2022 - December 2022
Western Carolina University *Cullowhee, NC*

- Courses Tutored: Calculus I/II/III, Linear Algbera, Abstract Algebra, Discrete Structures, Introduction to Programming.

RELEVANT GRADUATE COURSEWORK

Real Analysis, Abstract Algebra, Geometry, Topology, Commutative Algebra, Representation Theory, Theory of Algorithms, and Advanced Linear Algebra

SOFTWARE

PathAlgebras, A C++ implementation of Buchberger's Algorithm for finding Gröbner bases. Written with Dr. W. Frank Moore.

TECHNICAL STRENGTHS

Computer Languages	C, C++, Python, Java, Bash, AWK
Tools	Vim, Valgrind, cmake
Systems	Linux, systemd, runit