SBStrips

Version 0.6.3

Joe Allen (he/him/his)

A discrete model of special biserial algebras, string modules and their syzygies

Abstract

String modules for special biserial (SB) algebras are represented by string graphs. The syzygy of a string module (over an SB algebra) is a direct sum of string modules, by a 2004 result of Liu and Morin, therefore syzygy-taking can be performed at the (combinatorial) level of string graphs rather than the (homological-algebraic) level of modules.

This package represents string graphs in GAP by objects called strips and it implements syzygy-taking as an operation on them. Together with some utilities for book-keeping, it allows for very efficient calculation of kth syzygyies for large k.

Copyright

Acknowledgements

I thank my PhD supervisor Prof Jeremy Rickard, on whom I inflicted multiple early iterations of SBStrips, for his time and his comments. This package was much worse before his feedback. I also thank my fellow members of "Team Splendid" – namely Charley Cummings, Luke Kershaw and Dr Simon Peacock – for countless engaging conversations about representation theory, programming and much else besides.

Additionally, I received help understanding GAPDoc from Prof Max Horn and Dr Frank Lübeck, the latter of whom wrote the `makedocrel.g` file included in this package. I am grateful to them both.

Colophon

This package was created during the author's doctoral studies at the University of Bristol.

Contents

Chapter 1

Introduction

1.1 Aim

The aim of the SBStrips package is to calculate syzygies of string modules over special biserial (SB) algebras in a user-friendly way.

1.2 Some historical context

Special biserial algebras are a combinatorially-defined class of finite-dimensional K-algebras (over some field K, often assumed algebraically closed), which have been the object of much study. Among other results, their indecomposable finite-dimensional modules have been entirely classified into three sorts, one of which are the *string modules*. These are so called because their module structure is characterised by certain decorated graphs. In the literature these graphs are called *strings* but, for our convenience (shortly to be justified), we will call them *string graphs*.

Liu and Morin [LM04] proved that the syzygy $\Omega^1(X)$ of a string module X is a direct sum of indecomposable string modules. Their proof is constructive and elementary: the former, because it explicitly gives the string graphs describing each summand of $\Omega^1(X)$ from that describing X, and the latter, because they cleverly choose a basis of the projective cover $P(X)$ of X which disjointly combines bases of $\Omega^1(X)$ and X. In particular, their proof is valid regardless of the characteristic of the field K or whether it is algebraically closed. Consequently, we can argue that, in a very strong way, "taking the syzygy of a string module" is a (many-valued) combinatorial operation on combinatorial objects, not an algebraic one on algebraic objects.

This package implements that operation effectively. However, instead of the slightly naive notation/formalism used in the above article, SBStrips uses an alternative, more efficient, framework developed by the author during his doctoral studies. More precisely, the author devised a theoretical framework (to prove mathematical theorems)in that this package models. This theoretical framework was created with syzygy-taking in mind.

1.3 Why "strips", not "strings"?

Mantra:

If whenever you read the word "strip" in this package, you imagine that it means the kind of decorated graph that representation theorists call a "string", then you won't go too far wrong.

Liu and Morin's aforementioned paper exploits a kind of alternating behavior, manifesting from one syzygy to the next. Through much trial and error, the author found patterns only apparent over a greater "timescale". It rapidly became impractical to *describe* these greater patterns using the classical notation for strips, let alone to *rigorously prove* statements about them. From this necessity was born the SBStrips package – or, rather, the abstract framework underpinning it.

One crucial aspect in this framework is that string graphs are refined into objects called *strips*. This refinement is technical, does not break any new ground mathematically – it largely amounts to disambiguation and some algorithmic choice-making – and so we keep it behind the scenes.

The SBStrips user may safely assume that *strip* (or `IsStripRep`) simply means the kind of object that GAP uses to represent string graphs for SB algebras. As an added bonus, this name avoids a clash with those objects that GAP already calls "strings"!

1.4 Installation

The SBStrips package was designed for version 4.11 of GAP; the author makes no promises about compatibility with previous versions. It requires version 1.30 of QPA and version 1.6 of GAP-Doc. It is presently distributed in `tar.gz` and `zip` formats. These may be downloaded from `https://github.com/jw-allen/sbstrips/releases` (be sure to download the latest version!), and then unpacked into the user's `pkg` directory.

1.5 `InfoSBStrips`

1.5.1 InfoSBStrips

▷ `InfoSBStrips` (info class)

Returns: nothing.

The InfoClass for the SBStrips package. The default value is 1. Integer values from 0 to 4 inclusive are supported, offering increasingly verbose information about SBStrips' inner working. (When set to 0, no information is printed.)

Chapter 2

Worked example

Many people learn by doing. This chapter is for them. Here, we provide a guided tour of the the SBStrips package together with commentary.

2.1 How to teach a special biserial algebra to GAP using quivers (from QPA)

Before discussing string modules, we have to specify a SB algebra to GAP. We use functionality from the QPA package for this. Indeed, our GAP definition of SB algebra is an object for which `IsSpecialBiserialAlgebra` (**QPA: IsSpecialBiserialAlgebra**) returns `true`.

The first step is to define the presenting quiver `quiv1`. We use just one of the many methods QPA affords; see `Quiver` (**QPA: Quiver no. of vertices, list of arrows**) for details.

Example

```
gap> quiv1 := Quiver( 2,
> [ [ 1, 1, "a" ], [ 1, 2, "b" ], [ 2, 1, "c" ], [ 2, 2, "d" ] ]
> );
<quiver with 2 vertices and 4 arrows>
```

The second step is to create a path algebra `pa1`. We need the quiver `quiv1` that we just created and some field. For convenience, take `Rationals` (the $\mathbb{Q}$ object in GAP).

Example

```
gap> pa1 := PathAlgebra( Rationals, quiv1 );
<Rationals[<quiver with 2 vertices and 4 arrows>]>
```

The third step is to define a list `rels` of relations over that path algebra. Relations are linear combinations of paths in `pa1`. The addition, subtraction and multiplication of elements can be performed using `+`, `-` and `*`, as normal. Paths are `*`-products of arrows in `pa1`. For details on refer to arrows of `pa1`, see `.` (**QPA: . for a path algebra**).

Example

```
gap> rels := [ pa1.a * pa1.a, pa1.b * pa1.d, pa1.c * pa1.b, pa1.d * pa1.c,
>              pa1.c * pa1.a * pa1.b, (pa1.d)^4,
>              pa1.a * pa1.b * pa1.c - pa1.b * pa1.c * pa1.a ];
[ (1)*a^2, (1)*b*d, (1)*c*b, (1)*d*c, (1)*c*a*b, (1)*d^4,
  (1)*a*b*c+(-1)*b*c*a ]
```

We impose a rule on the relations: *they must be monomial relations or commutativity relations*. This imposition causes no mathematical loss of generality, of course.

The fourth step is to create the ideal `ideal` generated by the relations and to tell GAP that the relations form a Gröbner basis of `ideal`.

Example

```
gap> gb := GBNPGroebnerBasis( rels, pa1 );
[ (1)*a^2, (1)*b*d, (1)*c*b, (1)*d*c, (-1)*a*b*c+(1)*b*c*a, (1)*c*a*b,
  (1)*d^4 ]
gap> ideal := Ideal( pa1, gb );
<two-sided ideal in <Rationals[<quiver with 2 vertices and 4 arrows>]>,
  (7 generators)>
gap> GroebnerBasis( ideal, gb );
<complete two-sided Groebner basis containing 7 elements>
```

The final step is to quotient `pa1` by `ideal`, and hence finally obtain the SB algebra object `alg1`

Example

```
gap> alg1 := pa1/ideal;
<Rationals[<quiver with 2 vertices and 4 arrows>]/
<two-sided ideal in <Rationals[<quiver with 2 vertices and 4 arrows>]>,
  (7 generators)>>
```

(This algebra is the same as `SBStripsExampleAlgebra( 1 )`*. See A.2.1 for more information.)*

2.2 How to teach a string (graph) to GAP using strips.

We continue with the example SB algebra `alg1`, created in the previous section.

Consider the following string (graph) for `alg1`.

$$1 \xleftarrow{a} 1 \xrightarrow{b} 2 \xrightarrow{c} 1 \xleftarrow{a} 1 \xrightarrow{b} 2 \xleftarrow{d} 2 \xrightarrow{c} 1 \xleftarrow{a} 1 \xleftarrow{c} 2 \xrightarrow{d} 2$$

Reading from left to right, the first arrow in this string graph is a and it has exponent -1 (which means it points to the left, the negative direction). It's followed by 2 arrows with positive exponent (which we record as the positive integer "2"), then 1 with negative exponent (recorded as negative integer "-1"), then 1 positive ("1"), 1 negative ("-1"), 1 positive ("1"), 2 negative ("-2") and 1 positive ("1").

This is the information used when specifying the string graph to SBStrips. The operation `Stripify` (4.2.1) returns the strip representing this string graph. What gets printed is the formal word associated to the original string graph.

Example

```
gap> s := Stripify( alg1.a, -1, [ 2, -1, 1, -1, 1, -2, 1 ] );
(a)^-1(b*c) (a)^-1(b) (d)^-1(c) (c*a)^-1(d)
gap> IsStripRep( s );
true;
```

The reader will note that reading the string graph "from left to right" a moment ago was only possible of how the graph was written on the page. That same graph may be written equally well in the "reflected" format, as follows.

$$2 \xleftarrow{d} 2 \xrightarrow{c} 1 \xrightarrow{a} 1 \xleftarrow{c} 2 \xrightarrow{d} 2 \xleftarrow{b} 1 \xrightarrow{a} 1 \xleftarrow{c} 2 \xleftarrow{b} 1 \xrightarrow{a} 1$$

This gives us different defining data for the string. However, SBStrips is smart enough to know that these two are representations of the same object.

Example
```
gap> t := Stripify( alg1.d, -1, [ 2, -1, 1, -1, 1, -2, 1 ] );
(d)^-1(c*a) (c)^-1(d) (b)^-1(a) (b*c)^-1(a)
gap> s = t;
true
```

2.3 How to calculate syzygies of string(modules)s using strips

We continue with the strip `s` for the SB algebra `alg1` from the previous sections.

We know that `s` represents some (string) module X for `alg1`. The syzygy of that string module X is a direct sum of indecomposable string modules, each of which may be represented by string graph. Those string graphs, or rather strips representing them, can be calculated directly from `s`. This is the heart of the SBStrips package!

So, let's start calculating the "syzygy strips" of `s`. This calls for the attribute `SyzygyOfStrip` (4.5.1), which returns a list of strips, one for each indecomposable direct summand of the syzygy of its input.

Example
```
gap> SyzygyOfStrip( s );
[ (v2)^-1(c) (a)^-1(b*c) (c*a)^-1(d^2), (a)^-1(v1), (d)^-1(v2) ]
gap> Length( last );
3
```

The call to `Length` (**Reference: Length**) reveals that the the syzygy of `s` has 3 indecomposable summands.

Of course, there's no reason to stop at 1st syzygies. SBStrips is able to take Nth syzygies very easily for $N \geq 0$. For example, we can calculate the 4th syzygy of `s` as follows.

Example
```
gap> 4th_syz := NthSyzygyOfStrip( s, 4 );
[ (v2)^-1(c*a) (c)^-1(v2), (v2)^-1(d^2), (a)^-1(v1), (v2)^-1(d^2),
  (a)^-1(b*c) (a)^-1(v1), (d^2)^-1(v2), (v1)^-1(a), (v2)^-1(v2),
  (v2)^-1(c) (c*a)^-1(v2), (v1)^-1(a), (v2)^-1(v2),
  (v2)^-1(c) (c*a)^-1(v2), (v2)^-1(v2), (a)^-1(v1),
  (v2)^-1(c*a) (c)^-1(v2), (v2)^-1(d), (a)^-1(v1), (v2)^-1(d^2),
  (a)^-1(v1), (d^2)^-1(v2) ]
gap> Length( 4th_syz );
20
```

We find that the 4th syzygy of `s` has 20 indecomposable direct summands.

The reader will spot that many strips appear multiple times in `4th_syz`. If you want to remove duplicates, then the most efficient way is with `Set` (**Reference: Set**). (Mathematically, this is like asking for just the isomorphism types of modules in the 4th syzygy of the module represented by `s`, ignoring how often that type is witnessed.)

Alternatively, you can use `Collected` (**Reference: Collected**), which turns the list into something that a mathematician might call a multiset. This means that the distinct strips are recorded along with their frequency in the list. For example, the second output below means that `(v2)^-1(v2)` occurs 3 times in `4th_syz` while `(v1)^-1(a)` occurs 6 times.

Example
```
gap> Set( 4th_syz );
[ (v2)^-1(v2), (v1)^-1(a), (v2)^-1(d), (v2)^-1(d^2),
```

```
    (v2)^-1(c*a) (c)^-1(v2), (a)^-1(b*c) (a)^-1(v1) ]
gap> Collected( 4th_syz );
[ [ (v2)^-1(v2), 3 ], [ (v1)^-1(a), 6 ], [ (v2)^-1(d), 1 ],
  [ (v2)^-1(d^2), 5 ], [ (v2)^-1(c*a) (c)^-1(v2), 4 ],
  [ (a)^-1(b*c) (a)^-1(v1), 1 ] ]
```

This package uses the term *collected lists* for these multisets, and it offers several built-in functionalities for calculating collected lists of syzygies. Principal among these are `CollectedSyzygyOfStrip` (4.5.3) and `CollectedNthSyzygyOfStrip` (4.5.4).

Example

```
gap> CollectedSyzygyOfStrip( s );
[ [ (a)^-1(v1), 1 ], [ (d)^-1(v2), 1 ],
  [ (v2)^-1(c) (a)^-1(b*c) (c*a)^-1(d^2), 1 ] ]
gap> CollectedNthSyzygyOfStrip( s, 4 );
[ [ (v2)^-1(c) (c*a)^-1(v2), 4 ], [ (v2)^-1(v2), 3 ],
  [ (v1)^-1(a), 6 ], [ (v2)^-1(d^2), 5 ], [ (v2)^-1(d), 1 ],
  [ (a)^-1(b*c) (a)^-1(v1), 1 ] ]
```

The author recommends that, if calculating kth syzygies for large k (say $k \geq 10$), you use a Collected method. Details can be found in 6.1.

2.4 Aside: How to calculate Nth syzygies efficiently for large N

This section is a short digression, more philosophical than computational.

A central point of the author's doctoral studies (refer [All]) is that the syzygies of a string module should be arranged in a particular format. (A little more specifically, they should be written into a certain kind of array.) Most of the time this format does not print nicely onto the Euclidean plane so, sadly, there is little hope of GAP displaying syzygies in the most "optimal" way. The closest it can get – which is not very close at all, frankly – is the list format returned by `SyzygyOfStrip` or `NthSyzygyOfStrip`. However, this format compresses lots into a single line. This loses information and becomes a very inefficient way to store data (let along compute with them). By using functions like `Collected`, `CollectedSyzygyOfStrip` and `CollectedNthSyzygyOfStrip`, we lose what little information the list presentation holds onto, but we streamline out calculations greatly.

To see this, let `s` be the example strip from above and consider the 20th syzygy of `s`. The following calculation shows that it has 344732 distinct summands (many of which will be isomorphic). On the author's device, this took over 2 minutes to perform.

Example

```
gap> NthSyzygyOfStrip( s, 20 );;
gap> time;
130250
gap> Length( last2 );
344732
```

Compare this with a `Collected` approach, wherein the 20th syzygy was calculated in a heartbeat and the 200th syzygy is not much more. For comparison, and as a small boast, we also include times for the 2000th, 20000th and 200000th syzygies for comparison.

Example

```
gap> CollectedNthSyzygyOfStrip( s, 20 );
[ [ (v2)^-1(c) (c*a)^-1(v2), 66012 ], [ (v2)^-1(v2), 55403 ],
```

```
  [ (v1)^-1(a), 121414 ], [ (v2)^-1(d^2), 101901 ], [ (v2)^-1(d), 1 ],
  [ (a)^-1(b*c) (a)^-1(v1), 1 ] ]
gap> time;
62
gap> CollectedNthSyzygyOfStrip( s, 200 );
[ [ (v2)^-1(c) (c*a)^-1(v2),
      2861032065381047716503208868500150020186506750308366O ],
  [ (v2)^-1(v2), 2401226318717329243873309191478875651421941305244698 1 ]
    ,
  [ (v1)^-1(a), 52622583840983769603765180599790256716084480555530640 ],
  [ (v2)^-1(d^2), 44165437642884416151601614150885951220530708429827491
    ], [ (v2)^-1(d), 1 ], [ (a)^-1(b*c) (a)^-1(v1), 1 ] ]
gap> time;
547
gap> CollectedNthSyzygyOfStrip( s, 2000 );; time;
5422
gap> CollectedNthSyzygyOfStrip( s, 20000 );; time;
54172
gap> CollectedNthSyzygyOfStrip( s, 200000 );; time;
548922
```

We warn the reader that, even in this easier-to-store collected form, the integers involved may become too big for GAP to handle. Effective book-keeping only *increases* the upper bound on information we can store; it doesn't *remove* it!

2.5 How to call the strips representing simple, projective and injective (string) modules

We continue with the SB algebra `alg1` from before which, we remind the reader, was defined in terms of the quiver `quiv1`. There is nothing very special about the running example strip `s` that previous sections have focussed on. The associated string module is certainly not canonical in any way.

However, there are some string modules which really *are* canonical for one reason or another, and which SBStrips has methods to call. This includes the simple modules and, more generally, uniserial modules. It also includes the indecomposable projective or injective modules, provided that they are string modules (which is not guaranteeed).

To obtain the list of strips that describe the simple modules, use `SimpleStripsOfSBAlg` (4.3.1).

Example

```
gap> SimpleStripsOfSBAlg( alg1 );
[ (v1)^-1(v1), (v2)^-1(v2) ]
```

The `i`th entry is the strip describing the `i`th simple module S_i.

The uniserial modules are also string modules. They correspond to paths in the SB algebra. There is a method for `Stripify` (4.2.1) that turns a path for the SB algebra into the corresponding strip.

Example

```
gap> Stripify( alg1.a * alg1.b );
(a*b)^-1(v1)
gap> Stripify( alg1.c );
(c)^-1(v2)
gap> Stripify( alg1.d^3 );
```

```
(d^3)^-1(v2)
gap> Stripify( alg1.v1 );
(v1)^-1(v1)
```

(*A quick reminder on QPA syntax.* Here, `a`, `b`, `c` and `d` are the names of arrows in `quiv1`. The corresponding elements of `alg1` are called by `alg1.a` and `alg1.b` and so on. As in `quiv1`, paths are products of arrows. Thus, `alg1.a * alg1.b` is the element of `alg1` corresponding to the path `a * b` ("`a` then `b`") in `quiv1`. We see that vertices or arrows of the SB algebra (such as `alg1.v1` and `alg1.c`) are paths too. We also see an example of the `^` operation: `alg2.d^3` is equivalent to `alg2.d * alg2.d * alg2.d`.)

Since vertices are still paths (trivially) and simple modules are uniserial (trivially), we therefore have a second way to access the simple modules of a SB algebra.

Example

```
gap> s1 := Stripify( alg2.v1 );;
gap> s2 := Stripify( alg2.v2 );;
gap> SimpleStripsOfSBAlg( alg1 );
[ (v1)^-1(v1), (v2)^-1(v2) ]
gap> [ s1, s2 ] = SimpleStripsOfSbAlg( alg2 );
true
```

Some of the indecomposable projective modules are string modules. The attribute `ProjectiveStripsOfSBAlg` (4.3.3) returns a list, whose `r`th entry is the strip describing the module P_r (if P_r is indeed a string module) or the boolean `fail` (if not). The attribute `InjectiveStripsOfSBAlg` (4.3.4) is similar.

Example

```
gap> ProjectiveStripsOfSbAlg( alg1 );
[ fail, (c*a)^-1(d^3) ]
gap> InjectiveStripsOfSbAlg( alg1 );
[ fail, (v1)^-1(a*b) (d^3)^-1(v2) ]
```

2.6 Some inbuilt tests for string modules using strips

The objective of the SBStrips package is to investigate the syzygies of string modules over SB algebras for patterns. There are some patterns, described in 3.3, that it already knows to look for. This section describes those functionalities.

We keep the algebra `alg1` defined in previous sections. For comparison, we introduce an additional algebra using `SBStripsExampleAlgebra` (A.1.1), giving it the very imaginative name `alg2`.

Example

```
gap> SetInfoLevel( InfoSBStrips, 3 );
gap> alg2 := SBStripsExampleAlgebra( 2 );
#I  The quiver of this algebra has 3 vertices
#I    v1
#I    v2
#I    v3
#I  and 3 arrows
#I    a: v1 --> v2
#I    b: v2 --> v3
#I    c: v3 --> v1
<Rationals[<quiver with 3 vertices and 3 arrows>]/
```

```
<two-sided ideal in <Rationals[<quiver with 3 vertices and 3 arrows>]>,
  (3 generators)>>
gap> SetInfoLevel( InfoSBStrips, 2 );
```

By raising the level of `InfoSBStrips` (1.5.1) to 3, we make SBStrips provide a bit more detail about this example algebra. Full details can be found in A.2.2 but, for now, it suffices to say that `alg2` is a Nakayama algebra. This means an algebra for which all indecomposable modules are uniserial. It follows that `alg2` is representation finite and, *a fortiori*, syzygy finite.

Let's pick any old uniserial module U for `alg2` and then call the associated strip `u`.

Example

```
gap> UniserialStripsOfSBAlg( alg2 );
[ (v1)^-1(v1), (v2)^-1(v2), (v3)^-1(v3), (a)^-1(v1), (b)^-1(v2),
  (c)^-1(v3), (a*b)^-1(v1), (b*c)^-1(v2), (c*a)^-1(v3), (a*b*c)^-1(v1),
  (b*c*a)^-1(v2), (c*a*b)^-1(v3) ]
gap> u := last[8];
(a*b)^-1(v1)
```

Is U weakly periodic? Recall that U is *weakly periodic* if there is some `k>0` such that U is a direct summand of $\Omega^k(U)$. We'll test this by hunting for `u` amongst the syzygy strips of `u`. Initially, let's look amongst the first 4 syzygies using the operation `IsWeaklyPeriodicStripByNthSyzygy` (4.5.14).

Example

```
gap> IsWeaklyPeriodicStripByNthSyzygy( u, 4 );
#I  Examining strip: (a*b)^-1(v1)
#I  This strip does not occur as a summand of its first 4 syzygies
false
```

No luck so far but observe that, since `InfoSBStrips` is currently at level 2, SBStrips has also provided some commentary alongside our non-result. Let us raise the threshold from 4 to 10 and try again.

Example

```
gap> IsWeaklyPeriodicStripByNthSyzygy( u, 10 );
#I  Examining strip: (a*b)^-1(v1)
#I  This strip first appears as a direct summand of its 6th syzygy
true
```

Fantastic! We learn that strip `u` does appear among its first 10 syzygies. In fact, SBStrips is even kind enough to tell us the index of the earliest recurrence of `u`: at the 6th syzygy. (This information would not be printed if `InfoSBStrips` had value 1 or 0.)

Now let us take a uniserial module U – for `alg1`, this time – and conduct a similar investigation.

Example

```
gap> UniserialStripsOfSBAlg( alg1 );
[ (v1)^-1(v1), (v2)^-1(v2), (a)^-1(v1), (b)^-1(v1), (c)^-1(v2),
  (d)^-1(v2), (a*b)^-1(v1), (b*c)^-1(v1), (c*a)^-1(v2), (d^2)^-1(v2),
  (d^3)^-1(v2) ]
gap> uu := last[7];
(a*b)^-1(v1)
gap> IsWeaklyPeriodicStripByNthSyzygy( uu, 10 );
#I  Examining strip: (a*b)^-1(v1)
#I  This strip does not occur as a summand of its first 10 syzygies
false
```

```
gap> IsWeaklyPeriodicStripByNthSyzygy( uu, 100 );
#I  Examining strip: (a*b)^-1(v1)
#I  This strip does not occur as a summand of its first 100 syzygies
false
gap> IsWeaklyPeriodicStripByNthSyzygy( uu, 10000 ); time;
#I  Examining strip: (a*b)^-1(v1)
#I  This strip does not occur as a summand of its first 10000 syzygies
false
6703
```

We find that uu does not occur amongst its first 10000 syzygies. This certainly *suggests* that uu is not weakly periodic, but "absence of evidence is not evidence of absence" so this does not constitute a proof. On the basis of this test alone, we cannot rule out its first reccurence being at index 10001. Hunting among syzygy strips of uu for uu itself has been fruitless, therefore we should change tactic. We will now use the test IsFiniteSyzygyTypeStripByNthSyzygy (4.5.13) to calculate the set of syzygy strips of uu appearing at or before index 0, then the strips at or before index 1, then index 2 and so on up until a specified index N. If this ascending sequence of finite sets stabilizes at or before index N, then we may conclude that uu has finite syzygy type.

Example

```
gap> IsFiniteSyzygyTypeStripByNthSyzygy( uu, 10000 );
#I  Examining strip: (a*b)^-1(v1)
#I  This strip has finite syzygy type.
#I  The set of strings appearing as summands of its first N syzygies st\
abilizes at N=4, at which point it has cardinality 6
true
```

This is indeed what happens. Handily, SBStrips also tells us the index at which stabilization occurs (namely 4) and how many distinct strips were seen at or before this index. We deduce that any strips that are going to occur would do so by the 4th syzygy. Since uu did not recur by then, we know that it never will.

If we seek modules of infinite syzygy type, we should not test any strips over alg2. We know the answer will be negative in that case, since alg2 is a representation finite algebra and so all of its modules are trivially of finite syzygy type. Instead, we turn again to strips over alg1.

We need look no further than the simple module at vertex 1.

Example

```
gap> s1 := SimpleStripsOfSBAlg( alg1 )[1];
(v1)^-1(v1)
gap> IsFiniteSyzygyTypeStripByNthSyzygy( s1, 10 );
#I  Examining strip: (v1)^-1(v1)
#I  The set of strings appearing as summands of its first 10 syzygies h\
as cardinality 15
false
gap> IsFiniteSyzygyTypeStripByNthSyzygy( s1, 100 );
#I  Examining strip: (v1)^-1(v1)
#I  The set of strings appearing as summands of its first 100 syzygies \
has cardinality 105
false
gap> IsFiniteSyzygyTypeStripByNthSyzygy( s1, 1000 );
#I  Examining strip: (v1)^-1(v1)
#I  The set of strings appearing as summands of its first 1000 syzygies\
```

```
 has cardinality 1005
false
gap> time;
284860
```

These tests are enough to pique our interest but, sadly, not enough to base rigorous conclusions on.

We round off this section with a very interesting calculation which we hope piques the reader's interest as it has piqued the author's. For each of algebra created with `SBStripsExampleAlgebra` (A.1.1), we will test whether the injective string modules have finite syzygy type. If they do, then we say that algebra *passes* the test.

Example

```
gap> SetInfoLevel( InfoSBStrips, 1 );
gap> alg_list := List( [ 1..5 ], SBStripsExampleAlgebra );;
gap> SetInfoLevel( InfoSBStrips, 2 );
gap> for A in alg_list do
> TestInjectiveStripsUpToNthSyzygy( A, 200 );
> Print( "\n" );
> od;
#I  Examining strip: (v1)^-1(a*b) (d*d*d)^-1(v2)
#I  This strip has finite syzygy type.
#I  The set of strings appearing as summands of its first N syzygies st\
abilizes at N=3, at which point it has cardinality 5
The given SB algebra has passed the test!

#I  Examining strip: (v1)^-1(a*b*c)
#I  This strip has finite syzygy type.
#I  The set of strings appearing as summands of its first N syzygies st\
abilizes at N=0, at which point it has cardinality 1
[...lengthy output omitted from documentation for space reasons...]
The given SB algebra has passed the test!

#I  Examining strip: (v1)^-1(a*b*c*d) (f*g*h*f*g*h)^-1(v1)
#I  This strip has finite syzygy type.
#I  The set of strings appearing as summands of its first N syzygies st\
abilizes at N=7, at which point it has cardinality 14
[...omitted...]
The given SB algebra has passed the test!

#I  Examining strip: (v7)^-1(n*o*p*a) (n*o*p)^-1(v7)
#I  This strip has finite syzygy type.
#I  The set of strings appearing as summands of its first N syzygies st\
abilizes at N=8, at which point it has cardinality 21
[...omitted...]
The given SB algebra has passed the test!

#I  Examining strip: (v1)^-1(a*b*c*d*a) (e*f*g*e*f*g*e)^-1(v1)
#I  This strip has finite syzygy type.
#I  The set of strings appearing as summands of its first N syzygies st\
abilizes at N=6, at which point it has cardinality 13
[...omitted...]
The given SB algebra has passed the test!
```

(We changed the level of InfoSBStrips to give the most informative output, subject to reasonable space constraints of this documentation.)

This calculation suggests that the injective string modules of a SB algebra have finite syzygy type. Indeed, all SB algebras that the author has tested have passed (for sufficiently large N). The author conjectures that this statement – all injective modules over a SB algebra have finite syzygy type – is true in general. It is known to imply several other important homological properties, such as the big finitistic dimension conjecture for SB algebras (currently open). The statement has been the principal focus of the author's doctoral studies [All] but has not been attained... yet.

2.7 How to turn a strip into a quiver representation

A combinatorial approach is all very well and good but perhaps you really do want a module for your SB algebra. SBStrips can provide!

For the following, recall from preceding sections the strip s for the algebra alg1. Further recall that alg1 was defined in terms of the quiver quiv1.

The strip s stands for a string module for alg1. That string module can be modelled as a representation of quiv1. To obtain that quiver representation from s, use ModuleOfStrip (4.5.5).

Example

```
gap> module := ModuleOfStrip( s );
<[ 6, 5 ]>
gap> Print( module );
<Module over <Rationals[<quiver with 2 vertices and 4 arrows>]/
<two-sided ideal in <Rationals[<quiver with 2 vertices and 4 arrows>]>
, (7 generators)>> with dimension vector [ 6, 5 ]>
```

You can turn a list of strips into a list of modules using ModuleOfStrip (4.5.5).

Example

```
gap> 4th_syz := NthSyzygyOfStrip( s, 4 );;
gap> ModuleOfStrip( 4th_syz );
[ <[ 2, 2 ]>, <[ 0, 3 ]>, <[ 2, 0 ]>, <[ 0, 3 ]>, <[ 4, 1 ]>,
  <[ 0, 3 ]>, <[ 2, 0 ]>, <[ 0, 1 ]>, <[ 2, 2 ]>, <[ 2, 0 ]>,
  <[ 0, 1 ]>, <[ 2, 2 ]>, <[ 0, 1 ]>, <[ 2, 0 ]>, <[ 2, 2 ]>,
  <[ 0, 2 ]>, <[ 2, 0 ]>, <[ 0, 3 ]>, <[ 2, 0 ]>, <[ 0, 3 ]> ]
```

You can turn a collected list (see 6.1) of strips into a collected list of modules using ModuleOfStrip (4.5.5).

Example

```
gap> coll_4th_syz := CollectedNthSyzygyOfStrip( s, 4 );
[ [ (v2)^-1(c) (c*a)^-1(v2), 4 ], [ (v2)^-1(v2), 3 ], [ (v1)^-1(a), 6 ],
  [ (v2)^-1(d^2), 5 ], [ (v2)^-1(d), 1 ], [ (a)^-1(b*c) (a)^-1(v1), 1 ]
 ]
gap> ModuleOfStrip( coll_4th_syz );
[ [ <[ 2, 2 ]>, 4 ], [ <[ 0, 1 ]>, 3 ], [ <[ 2, 0 ]>, 6 ],
  [ <[ 0, 3 ]>, 5 ], [ <[ 0, 2 ]>, 1 ], [ <[ 4, 1 ]>, 1 ] ]
```

The latter two methods take a list (or collected list) of strips and return a list of modules (or a collected list, as appropriate). Perhaps you want the direct sum of all the modules in that list or collected list. Naive calls to QPA's inbuilt functionality turn out to be resource intensive. In its place, SBStrips offers the operation DirectSumModuleOfListOfStrips (4.5.6) or DirectSumModuleOfListOfStrips (4.5.6).

Example

```
gap> 4th_syz := NthSyzygyOfStrip( s, 4 );;
gap> coll_4th_syz := CollectedNthSyzygyOfStrip( s, 4 );;
gap> DirectSumModuleOfListOfStrips( 4th_syz );
<[ 24, 29 ]>
gap> DirectSumModuleOfListOfStrips( coll_4th_syz );
<[ 24, 29 ]>
```

Chapter 3

Mathematical background

3.1 Finite-dimensional algebras

Here and throughout, K is some field. By a *K-algebra* A, we mean an associative and unital (but not necessarily commutative) ring with a compatible K-vector space structure.

Suppose Q is a finite quiver: that is, a directed graph with finitely many vertices and finitely many arrows, where loops and/or multiple edges are permitted. The paths of Q (also called *Q-paths* to emphasis their parent quiver), including the "stationary" paths at each vertex, form the basis of a vector space. multiplication can be defined on basis vectors p and q by "concatenation extended by zero"; more precisely, $p \cdot q = pq$ ("p then q") if pq is a path in Q, and $p \cdot q = 0$ otherwise. This defines the *path algebra* KQ. Its multiplicative unit is the sum of stationary paths. It has finite K-dimension iff Q contains no (directed) cycles.

Let $J \trianglelefteq KQ$ denote the *arrow ideal* of KQ: the smallest two-sided ideal of KQ containing the arrows of Q. An ideal $I \trianglelefteq KQ$ is *admissible* iff there is an integer $N \geq 2$ with $J^N \subseteq I \subseteq J^2$.

By a *(bound) quiver algebra*, we mean a quotient KQ/I of a path algebra KQ by an admissible ideal I. Quiver algebras are always finite-dimensional [ASS06, Sec II.2]. Indeed, at least when K is algebraically closed, any finite-dimensional algebra is a direct product of connected ones (trivially), any connected finite-dimensional algebra is Morita equivalent to a basic one [ASS06, Sec I.6] and any basic, connected algebra is isomorphic to a quiver algebra [ASS06, Sec II.3].

In this work, we assume that K is algebraically closed and A is a quiver algebra KQ/I. By the above, this is no loss of generality. We also use the term A-path to mean a nonzero element $p+I$ represented by a path of the quiver.

3.2 Modules and bound quiver representations

A representation of A is a homomorphism of algebras $\phi : A \to \operatorname{End} X$ whose target is the endomorphism algebra $\operatorname{End} X$ of a K-vector space X; for convenience, write ϕ_a for the image of a in ϕ. In this case, we call X a *(right) A-module*, with associated action $x \cdot a = x\phi_a$. The module is *finite-dimensional* iff X is.

Since $A = KQ/I$, we can work in terms of *(bound) representations of quivers*. These are assignments of a vector space X_i to each vertex i of Q and a linear map $\theta_\alpha : X_i \to X_j$ to each arrow $\alpha : i \to j$ of Q such that for any $\rho = \sum_{k=1}^{m} p_k \lambda_k \in I$, the associated map $\sum_{k=1}^{m} \theta_{p_k} : \bigoplus_i X_i \to \bigoplus_i X_i$ is zero. Here, $\theta_{p_k} = \theta_{\alpha_1} \cdots \theta_{\alpha_r}$ for a decomposition of a nonstationary p_k into a product $\alpha_1 \cdots \alpha_r$ of arrows, and $\theta_i = \mathrm{id}_{X_i}$ for any stationary path at i.

As is well-known [ASS06, III.1], representations of quivers are equivalent to modules. More specifically, the categories $\mathrm{Rep}(Q,I)$ of bound quiver representations and $\mathrm{Mod}\text{-}A$ of A-modules are equivalent, and this equivalence restricts to their respective full subcategories $\mathrm{rep}(Q,I)$ and $\mathrm{mod}\text{-}A$ of finite-dimensional objects. In keeping with the quiver-minded approach from above, whenever we say module, we really mean the equivalent bound quiver representation.

We note in particular that all of the categories in the previous paragraph are abelian: thus, we can speak of the direct sum of modules (denoted with $\oplus$) . We call a module X *indecomposable* if $X = Y \oplus Z$ implies Y or Z is zero. Further, if $\mathscr{U}$ is any set of A-modules, we define the additive closure $\mathrm{add}\,\mathscr{U}$ of $\mathscr{U}$ as the full subcategory of $\mathrm{mod}\text{-}A$ whose objects are isomorphic to direct summands of finite direct sums of members of $\mathscr{U}$.

There are certain canonical classes of module. A module X is: *simple* if it has no proper, nonzero submodules; *projective* if the covariant functor $\mathrm{Hom}_A(X,-)\colon \mathrm{Mod}\text{-}A \to \mathrm{Mod}\text{-}Z$ is exact, or; *injective* if the functor $\mathrm{Hom}_A(-,X)\colon \mathrm{Mod}\text{-}A \to \mathrm{Mod}\text{-}Z$ is exact. The simple A-modules (necessarily indecomposable) are in one-to-one correspondence with the vertices of Q, as are the indecomposable projective and injective modules. We respectively write S_i, P_i and I_i for the simple, indecomposable projective and indecomposable injective module corresponding to the vertex i.

A *composition series* for a module X is a strictly ascending chain of submodules

$$0 = X_0 < X_1 < X_2 < \cdots < X_{l-1} < X_l = X$$

of X such that each consecutive quotient X_{k+1}/X_k is simple. A module is *uniserial* if it has a unique composition series; equivalently, if its submodules form a chain.

Write $[X]$ for the isomorphism type of X. One can seek to classify the isomorphism classes of indecomposable (finite-dimensional) modules of an algebra. A deep theorem of Drozd [Dro80] establishes that all finite-dimensional algebras fall into exactly one of three *representation types*. In increasing order of difficulty, the options are *representation finite*, *tame* or *wild*. The first simply means the algebra has only finitely many isoclasses of indecomposables. Speaking informally, tame algebras are those for which, in each dimension, almost all modules lie in one of finitely many classes each parameterized by the field. Speaking even more informally, wild algebras are those for which the classification problem is intractible in a very strong way. Discussion and formal definitions of representation type can be found in [Ben95, Sec 4.4].

3.3 Syzygies and patterns

A *projective cover* of a finite-dimensional module X is a surjective module homomorphism $\pi\colon P \twoheadrightarrow X$ such that P is projective and $\ker\pi$ is a superfluous submodule of P: refer [ASS06, Sec I.5]. The isomorphism class of P is unique: we denote it by $\mathbb{P}M$ but, in abuse of notation, treat $\mathbb{P}M$ as if it were a module rather than an isomorphism class. It is easy to see that $\mathbb{P}(X \oplus Y) = (\mathbb{P}X) \oplus (\mathbb{P}Y)$ and that, if X is itself projective, then the identity map on X is a projective cover.

By convention the *0th syzygy* $\Omega^0 X$ of X is X/P, for P the largest projective direct summand of X. For $k \geq 0$, we iteratively define the $(k+1)$th syzygy $\Omega^{k+1}X$ to be the kernel of the projective cover of $\Omega^k X$.

For $k \geq 0$ and some fixed module X, let

$$\mathscr{A}_k = \big\{[M]\colon M \in \mathrm{add}\{\Omega^t X\} \text{ for some } t \geq k\big\}, \quad \mathscr{B}_k = \big\{[M]\colon M \in \mathrm{add}\{\Omega^t X\} \text{ for some } t \leq k\big\}.$$

(These letters were chosen so that $\mathscr{A}_k$ contains the isoclasses of indecomposables appearing at or $\mathscr{A}$fter the kth syzygy, while $\mathscr{B}_k$ contains those appearing at or $\mathscr{B}$efore.) The $\mathscr{A}_k$ and $\mathscr{B}_k$ relate in the

following way:

$$\mathscr{B}_0 \subseteq \mathscr{B}_1 \subseteq \mathscr{B}_2 \subseteq \cdots \subseteq \bigcup_{k\geq 0} \mathscr{B}_k = \mathscr{A}_0 \supseteq \mathscr{A}_1 \supseteq \mathscr{A}_2 \supseteq \cdots \supseteq \bigcap_{k\geq 0} \mathscr{A}_k.$$

We comment that, for each successive inclusion $\mathscr{B}_k \supseteq \mathscr{B}_{k+1}$ or $\mathscr{A}_k \supseteq \mathscr{A}_{k+1}$, the appropriate set difference between them is finite. Note also that $\cap_{k\geq 0}\mathscr{A}_k$ contains exactly those isoclasses witnessed at $\Omega^k X$ for infinitely many indices k. Below, we use this sequence of inclusions to define some terminology for patterns in syzygies. Our definitions are inspired by comparable work in [GHZ96, Sec 2] and [Ric19, Sec 7].

If there is an index t for which $\mathscr{A}_t = \mathscr{A}_{t+1} (= \mathscr{A}_{t+2} = \cdots = \cap_{k\geq 0}\mathscr{A}_k)$, then there is a minimal one $t_\star$, in which case we say that the *syzygy repetition index* of X is $t - \star$. This holds exactly when $t_\star$ satisfies $\mathscr{A}_{t_\star} = \cap_{k\geq 0}\mathscr{A}_k$ (and is the minimal index to enjoy this property). If no such t exists, the syzygy repetition of index of X is $+\infty$.

If $\mathscr{A}_0$ is finite, then we say X has *syzygy type* $|\mathscr{A}_0|$ *of index* $s_\star$, for $s_\star$ the minimal index k such that $\mathscr{B}_k = \mathscr{B}_{k+1} (\mathscr{B}_{k+2} = \cdots = \mathscr{A}_0)$; the existence of $s_\star$ in this case follows from an easy finiteness argument. Just as immediately, we see that if X has finite syzygy type $|\mathscr{A}_0|$ then it has finite syzygy repetition index at most $|\mathscr{A}_0|$.

If $[X] \in \cap_{k\geq 0}\mathscr{A}_k$, then we call X *weakly periodic*.

3.4 Special biserial algebras

A *special biserial (SB) algebra* is a quiver algebra KQ/I such that

1. every vertex of Q is the source of at most 2 arrows,
2. every vertex of Q is the target of at most 2 arrows,
3. for every arrow a of Q, there is at most one arrow b with $ab \notin I$ and
4. for every arrow a of Q, there is at most one arrow c with $ca \notin I$.

These algebras emerged from the modular representation theory of finite groups. A key text on them is [WW85], which establishes in particular that they are tame algebras. Their indecomposable modules fall into three classes: band modules, string modules and a finite class of projective-injective nonuniserial ("pin") modules.

3.5 String modules for special biserial algebras

String modules earn their name from the string graphs that describe them so well. A string graph for $A = KQ/I$ is a quiver homomorphism $w\colon \Gamma \to Q$ from a quiver Γ whose underlying (undirected) graph is linear, and where the image in w of any Γ-path p is linearly independent of all other A-paths. More commonly, the string graph w is depicted by labelling each vertex and arrow of Γ by its respective images in Q. Then vertices of w provide a basis of the associated string module, and the labels describe the A action. We can identify a string graph with the string module it represents.

One subtle point: we do not require string graphs to be connected; accordingly we do require string modules to be indecomposable.

Liu and Morin [LM04] proved that the syzygy of a string module is a string module. In as-yet-unpublished work of Galstad [Gal], advertised by Huisgen-Zimmermann without proof [HZ16], the

syzygy of a band module X is also a string module provided that at least one indecomposable direct summand of $\mathbb{P}X$ is a string module.

Chapter 4

Strips

4.1 Introduction

Strips are the principal objects of the SBStrips package. They are syzygy-minded representations of string graphs.

4.2 Constructing strips

4.2.1 Stripify

▷ `Stripify(arr, N, int_list)` (method)
▷ `Stripify(path)` (method)

Arguments (first method): `arr`, an arrow in a SB algebra (see note below); `N`, an integer which is either `1` or `-1`; `int_list`, a (possibly empty) list of nonzero integers whose entries are alternately positive and negative).

Argument (second method): `path`, a path in a SB algebra.

(*Note.* Remember that vertices and arrows in a SB algebra, which is to say the elements in the algebra corresponding to the vertices and arrows of the quiver, can be easily accessed using `.` (**QPA: . for a path algebra**), and that these can be multiplied together using `\*` (**Reference: ***) to make a path in the SB algebra.)

Returns: the strip specified by these data

The first method is intended for specifying arbitrary string(graphs) over a SB algebra to GAP. The second method is more specialized, being intended for specifying those string(graph)s where all arrows point in the same direction. This includes the vacuous case where the string (graph) has no arrows.

For the first method, suppose you draw your string graph on the page as a linear graph with some arrows pointing to the right (the "positive" direction) and some to the left (the "negative" direction). See further below for examples.

The first arrow (ie, the leftmost one drawn on the page) is `arr`. If it points to the right (the "positive" direction), then set `N` to be `1`. If it points to the left (the "negative" direction), then set `N` to be `-1`.

Now, ignore that first arrow `arr` and look at the rest of the graph. It is made up of several paths that alternately point rightward and leftward. Each path has a *length*; that is, the total number of arrows in

it. Enter the lengths of these paths to *int_list* in the order you read them, using positive numbers for paths pointing rightwards and negative numbers for paths pointing leftwards.

SBStrips will check that your data validily specify a strip. If it doesn't think they do, then it will throw up an Error message.

For the second method, SBStrips directly infers the string (graph) and the SB algebra directly from *path*.

4.3 Canonical strips

4.3.1 SimpleStripsOfSBAlg

▷ SimpleStripsOfSBAlg(*sba*) (attribute)

Argument: *sba*, a special biserial algebra (ie, IsSpecialBiserialAlgebra (**QPA: IsSpecialBiserialAlgebra**) returs true)

Returns: a list simple_list, whose *j*th entry is the simple strip corresponding to the *j*th vertex of *sba*.

You will have specified *sba* to GAP via some quiver. The vertices of that quiver are ordered; SimpleStripsOfSBAlg adopts that order for strips of simple modules.

4.3.2 UniserialStripsOfSBAlg

▷ UniserialStripsOfSBAlg(*sba*) (attribute)

Argument: *sba*, a special biserial algebra

Returns: a list of the strips that correspond to uniserial modules for *sba*

Simple modules are uniserial, therefore every element of SimpleStripsOfSBAlg (4.3.1) will occur in this list too.

4.3.3 ProjectiveStripsOfSBAlg

▷ ProjectiveStripsOfSBAlg(*sba*) (attribute)

Argument: *sba*, a special biserial algebra (ie, IsSpecialBiserialAlgebra (**QPA: IsSpecialBiserialAlgebra**) returs true)

Returns: a list proj_list, whose entry are either strips or the boolean fail.

You will have specified *sba* to GAP via some quiver. The vertices of that quiver are ordered; ProjectiveStripsOfSBAlg adopts that order for strips of projective modules.

If the projective module corresponding to the jth vertex of *sba* is a string module, then ProjectiveStripsOfSBAlg(sba)[j] returns the strip describing that string module. If not, then it returns fail.

4.3.4 InjectiveStripsOfSBAlg

▷ InjectiveStripsOfSBAlg(*sba*) (attribute)

Argument: *sba*, a special biserial algebra

Returns: a list `inj_list`, whose entries are either strips or the boolean `fail`.

You will have specified *sba* to GAP via some quiver. The vertices of that quiver are ordered; `InjectiveStripsOfSBAlg` adopts that order for strips of projective modules.

If the injective module corresponding to the `j`th vertex of *sba* is a string module, then `InjectiveStripsOfSBAlg( sba )[j]` returns the strip describing that string module. If not, then it returns `fail`.

4.4 Attributes and properties of strips

4.4.1 WidthOfStrip

▷ `WidthOfStrip(`*strip*`)` (operation)

Argument: *strip*, a strip

Returns: a nonnegative integer, counting the number (with multiplicity) of syllables of *strip* are nonstationary.

4.4.2 IsZeroStrip

▷ `IsZeroStrip(`*strip*`)` (property)

Argument: *strip*, a strip

Returns: `true` if *strip* is the zero strip of some SB algebra, and `false` otherwise.

Note that SBStrips knows which SB algebra *strip* belongs to.

4.5 Operation on strips

The following attributes and operations usually take strips as input. However, many of them are clever enough to recognise a list or collected list of strips. They will then resort to special methods that apply the strip method entrywise, combine the outputs and return a list or collected list (as appropriate).

4.5.1 SyzygyOfStrip

▷ `SyzygyOfStrip(`*strip*`)` (attribute)

Argument: *strip*, a strip

Returns: a list of strips, corresponding to the indecomposable direct summands of the syzygy of *strip*.

For higher syzygies, `NthSyzygyOfStrip` (4.5.2) is probably more convenient and `CollectedNthSyzygyOfStrip` (4.5.4) probably more efficient.

4.5.2 NthSyzygyOfStrip

▷ `NthSyzygyOfStrip(`*strip*`, `*N*`)` (method)

Arguments: *strip*, a strip; *N*, a positive integer

Returns: a list of strips containing the indecomposable *N*th syzygy strips of *strip*

For large N – say, $N \geq 10$ – consider using `CollectedNthSyzygyOfStrip` (4.5.4) instead, since it is much more efficient.

4.5.3 CollectedSyzygyOfStrip

▷ `CollectedSyzygyOfStrip(`*strip*`)` (method)

Argument: *strip*, a strip

Returns: a collected list, whose elements are the syzygy strips of *strip*

This is equivalent to calling `Collected( SyzygyOfStrip( strip ) );`.

4.5.4 CollectedNthSyzygyOfStrip

▷ `CollectedNthSyzygyOfStrip(`*strip, N*`)` (method)

Arguments: *strip*, a strip; *N*, a positive integer.

Returns: a collected list, whose entries are the *N*th syzygies of *strip*.

4.5.5 ModuleOfStrip

▷ `ModuleOfStrip(`*strip*`)` (method)

Argument: a strip *strip*.

Returns: a right module for the SB algebra over which *strip* is defined, or a list or collected list of the modules associated to the strips in *list* or *clist* respectively.

This operation returns the string module corresponding to the strip *strip*. More specifically, it gives that module as a quiver, ultimately using `RightModuleOverPathAlgebra` (**QPA: RightModuleOverPathAlgebra with dimension vector**).

4.5.6 DirectSumModuleOfListOfStrips (for a (flat) list of strips)

▷ `DirectSumModuleOfListOfStrips(`*list*`)` (method)
▷ `DirectSumModuleOfListOfStrips(`*clist*`)` (method)

Argument (first method): *list*, a list of strips

Argument (second method): *clist*, a collected list of strips

Returns: the quiver representation corresponding to the direct sum of A-modules whose indecomposable direct summands are specified by *list* or *clist*.

The methods for this operation make the obvious requirement that all strips present belong to the the same SB algebra.

4.5.7 IsStripDirectSummand

▷ `IsStripDirectSummand(`*strip_or_strips, list*`)` (operation)

Arguments: *strip_or_strips*, a strip or list of strips or collected list of strips; *list*, a list or collected list of strips.

Returns: `true` if the string module represented by *strip_or_strips* is a direct summand of the string module represented by the strips in *list*, and `false` otherwise.

4.5.8 VectorSpaceDualOfStrip

▷ `VectorSpaceDualOfStrip(`*strip*`)` (attribute)
▷ `OppositeStrip(`*strip*`)` (attribute)
▷ `DOfStrip(`*strip*`)` (attribute)

Argument: *strip*, a strip representing some string module X over a K-algebra A.
Returns: a strip representing the vector-space dual module $\mathrm{D}M = \mathrm{Hom}_K(X,K)$ of X.
Recall that $\mathrm{D}X$ is a module for A^{op}, the opposite algebra to A.
`OppositeStrip` and `DOfStrip` are synonyms for `VectorSpaceDualOfStrip`.

4.5.9 TransposeOfStrip

▷ `TransposeOfStrip(`*strip*`)` (attribute)
▷ `TrOfStrip(`*strip*`)` (attribute)

Argument: *strip*, a strip representing some string module X.
Returns: a strip representing the transpose $\mathrm{Tr}X$ of X.
Recall that if X is an A-module, then $\mathrm{Tr}X$ is an A^{op}-module.
`TrOfStrip` is a synonym for `TransposeOfStrip`.

4.5.10 DTrOfStrip

▷ `DTrOfStrip(`*strip*`)` (attribute)
▷ `ARTranslateOfStrip(`*strip*`)` (attribute)

Argument: *strip*, a strip representing some string module X.
Returns: a strip representing the Auslander-Reiten translate $\mathrm{DTr}X$ of X.
Recall that if X is projective then $\mathrm{DTr}X = 0$.
`ARTranslateOfStrip` is a synonym for `DTrOfStrip`.

4.5.11 TrDOfStrip

▷ `TrDOfStrip(`*strip*`)` (attribute)
▷ `ARInverseTranslateOfStrip(`*strip*`)` (attribute)

Argument: *strip*, a strip representing some string module X.
Returns: a strip representing the Auslander-Reiten inverse translate $\mathrm{TrD}X$ of X.
Recall that if X is injective then $\mathrm{TrD}X = 0$.
`ARInverseTranslateOfStrip` is a synonym for `TrDOfStrip`.

4.5.12 SuspensionOfStrip

▷ `SuspensionOfStrip(strip)` (attribute)

Argument: *strip*, a strip representing some string module X

Returns: a list of strips, representing the indecomposable direct summands of the suspension $\Sigma X = \mathrm{Tr}\Omega\mathrm{Tr}X$ of X

4.5.13 IsFiniteSyzygyTypeStripByNthSyzygy

▷ `IsFiniteSyzygyTypeStripByNthSyzygy(strip, N)` (operation)

Arguments: *strip*, a strip; *N*, a positive integer

Returns: `true` if the strips appearing in the *N*th syzygy of *strip* have all appeared among earlier syzygies, and `false` otherwise.

If the call to this function returns `true`, then it will also print the smallest *N* for which it would return `true`.

4.5.14 IsWeaklyPeriodicStripByNthSyzygy

▷ `IsWeaklyPeriodicStripByNthSyzygy(strip, N)` (operation)

Arguments: *strip*, a strip; *N*, a positive integer

Returns: `true` if *strip* is appears among its own first *N* syzygies, and `false` otherwise.

If the call to this function returns `true`, then it will also print the index of the syzygy at which *strip* first appears.

4.6 Tests on an SB algebra that use strips

4.6.1 TestInjectiveStripsUpToNthSyzygy

▷ `TestInjectiveStripsUpToNthSyzygy(sba, N)` (function)

Arguments: *sba*, a special biserial algebra (ie, `IsSpecialBiserialAlgebra` (**QPA: IsSpecialBiserialAlgebra**) returs `true`); *N*, a positive integer

Returns: `true`, if all strips of injective string modules have finite syzygy type by the *N*th syzygy, and `false` otherwise.

This function calls `InjectiveStripsOfSBAlg` (4.3.4) for *sba*, filters out all the `fails`, and then checks each remaining strip individually using `IsFiniteSyzygyTypeStripByNthSyzygy` (4.5.13) (with second argument *N*).

Author's note. For every special biserial algebra the author has tested, this function returns true (for sufficiently large *N*). It suggests that the minimal injective cogenerator of a SB algebra always has finite syzygy type. This condition implies many homological conditions of interest (including the big finitistic dimension conjecture)!

Chapter 5

QPA utilities

5.1 Introduction

In order to do what it does, the SBStrips package includes several utility functions for use on quivers where each vertex has indegree and outdegree at most 2. (The existing term for such quivers is *special biserial, abbreivated SB*.) This class includes the 1-regular quivers: those where each vertex has indegree and outdegree exactly 1.

These quiver utility functions really build on the QPA package. We document them in this standalone chapter, alongside utilities for algebras presented by quivers.

The term *quiver algebra* means an object for which `IsQuiverAlgebra` (**QPA: IsQuiverAlgebra**) returns `true`.

5.2 Utilities for 1-regular quivers

5.2.1 Is1RegQuiver

▷ `Is1RegQuiver(`*`quiver`*`)` (property)

Argument: *`quiver`*, a quiver
Returns: either `true` or `false`, depending on whether or not *`quiver`* is 1-regular.

5.2.2 PathBySourceAndLength

▷ `PathBySourceAndLength(`*`vert, len`*`)` (operation)

Arguments: *`vert`*, a vertex of a 1-regular quiver Q; *`len`*, a nonnegative integer.
Returns: the unique path in Q which has source *`vert`* and length *`len`*.

5.2.3 PathByTargetAndLength

▷ `PathByTargetAndLength(`*`vert, len`*`)` (operation)

Arguments: *`vert`*, a vertex of a 1-regular quiver Q; *`len`*, a nonnegative integer.
Returns: the unique path in Q which has target *`vert`* and length *`len`*.

5.2.4 1RegQuivIntAct

▷ `1RegQuivIntAct(x, k)` (operation)

Arguments: `x`, which is either a vertex or an arrow of a 1-regular quiver; `k`, an integer.

Returns: the path $x+k$, as per the $\mathbb{Z}$-action (see below).

Recall that a quiver is 1-regular iff the source and target functions s,t are bijections from the arrow set to the vertex set (in which case the inverse t^{-1} is well-defined). The generator $1 \in \mathbb{Z}$ acts as "t^{-1} then s" on vertices and "s then t^{-1}" on arrows.

This operation figures out from `x` the quiver to which `x` belongs and applies `1RegQuivIntActionFunction` (5.2.5) of tha quiver. For this reason, it is more user-friendly.

5.2.5 1RegQuivIntActionFunction

▷ `1RegQuivIntActionFunction(quiver)` (attribute)

Argument: *quiver*, a 1-regular quiver (as tested by `Is1RegQuiver` (5.2.1))

Returns: a single function `f` describing the $\mathbb{Z}$-actions on the vertices and the arrows of *quiver*

Recall that a quiver is 1-regular iff the source and target functions s,t are bijections from the arrow set to the vertex set (in which case the inverse t^{-1} is well-defined). The generator $1 \in \mathbb{Z}$ acts as "t^{-1} then s" on vertices and "s then t^{-1}" on arrows.

In practice you will probably want to use `1RegQuivIntAct` (5.2.4), since it saves you having to remind SBStrips which quiver you intend to act on.

5.3 Utilities for SB quivers

5.3.1 Is2RegQuiver

▷ `Is2RegQuiver(quiver)` (property)

Argument: *quiver*, a quiver

Returns: either `true` or `false`, depending on whether or not *quiver* is 2-regular.

5.3.2 2RegAugmentationOfQuiver

▷ `2RegAugmentationOfQuiver(ground_quiv)` (attribute)

Argument: *ground_quiv*, a sub2-regular quiver (as tested by `IsSpecialBiserialQuiver` (**QPA: IsSpecialBiserialQuiver**))

Returns: a 2-regular quiver of which *ground_quiv* may naturally be seen as a subquiver

If *ground_quiv* is itself sub-2-regular, then this attribute returns *ground_quiv* identically. If not, then this attribute constructs a brand new quiver object which has vertices and arrows having the same names as those of *ground_quiv*, but also has arrows with names `augarr1`, `augarr2` and so on.

5.3.3 Is2RegAugmentationOfQuiver

▷ `Is2RegAugmentationOfQuiver(`*`quiver`*`)` (property)

Argument: *quiver*, a quiver
Returns: `true` if *quiver* was constructed by 5.3.2 or if *quiver* was an already 2-regular quiver, and `false` otherwise.

5.3.4 OriginalSBQuiverOf2RegAugmentation

▷ `OriginalSBQuiverOf2RegAugmentation(`*`quiver`*`)` (attribute)

Argument: *quiver*, a quiver
Returns: The sub-2-regular quiver of which *quiver* is the 2-regular augmentation.
Informally speaking, this attribute is the "inverse" to `2RegAugmentationOfQuiver`.

5.3.5 RetractionOf2RegAugmentation

▷ `RetractionOf2RegAugmentation(`*`quiver`*`)` (attribute)

Argument: *quiver*, a quiver constructed using `2RegAugmentationOfQuiver`
Returns: a function `ret`, which accepts paths in *quiver* as input and which outputs paths in `OriginalSBQuiverOf2RegAugmentation( quiver )` 5.3.2.
One can identify `OriginalSBQuiverOf2RegAugmentation( quiver )` with a subquiver of *quiver*. Some paths in *quiver* lie wholly in that subquiver, some do not. This function `ret` takes those that do to the corresponding path of `OriginalSBQuiverOf2RegAugmentation( quiver )`, and those that do not to the zero path of `OriginalSBQuiverOf2RegAugmentation( quiver )`.

5.4 Miscellaneous utilities for QPA

What follows are minor additional utilities for QPA.

5.4.1 String (for paths of length at least 2)

▷ `String(`*`path`*`)` (method)

Argument: *path*, a path of length at least 2 in a quiver (see `IsPath` (**QPA: IsPath**) and `LengthOfPath` (**QPA: LengthOfPath**) for details)
Returns: a string describing *path*
Methods for `String` (**Reference: String**) already exist for vertices and arrows of a quiver; that is to say, paths of length 0 or 1. QPA forgets these for longer paths: at present, only the default answer `"<object>"` is returned.
A path in QPA is products of arrows. Accordingly, we write its string as a `*`-separated sequences of its constituent arrows. This is in-line with how paths are printed using `ViewObj` (**Reference: ViewObj**).

5.4.2 ArrowsOfQuiverAlgebra

▷ `ArrowsOfQuiverAlgebra(alg)` (operation)

Argument: `alg`, a quiver algebra
Returns: the residues of the arrows in the defining quiver of `alg`, listed together

5.4.3 VerticesOfQuiverAlgebra

▷ `VerticesOfQuiverAlgebra(alg)` (operation)

Argument: `alg`, a quiver algebra
Returns: the residues of the vertices in the defining quiver of `alg`, listed together

5.4.4 FieldOfQuiverAlgebra

▷ `FieldOfQuiverAlgebra(alg)` (operation)

Argument: `alg`, a quiver algebra
Returns: the field of definition of `alg`

5.4.5 DefiningQuiverOfQuiverAlgebra

▷ `DefiningQuiverOfQuiverAlgebra(alg)` (operation)

Argument: `alg`, a quiver algebra
Returns: the quiver of definition of `alg`
This single operation performs `OriginalPathAlgebra` (**QPA: OriginalPathAlgebra**) and then `QuiverOfPathAlgebra` (**QPA: QuiverOfPathAlgebra**)

5.4.6 Paths obtained by adding/removing an arrow at source/target

▷ `PathOneArrowLongerAtSource(path)` (attribute)
▷ `PathOneArrowLongerAtTarget(path)` (attribute)
▷ `PathOneArrowShorterAtSource(path)` (attribute)
▷ `PathOneArrowShorterAtTarget(path)` (attribute)

Argument: `path`, a path
Returns: a path `new_path` which differs from `path` by one arrow in the appropriate way, or `fail` if no such arrow exists.
Both of the `-Shorter-` attributes require `path` to have length at least 1, as measured by `LengthOfPath` (**QPA: LengthOfPath**).
Both of the `-Longer-` attributes require there to exist a unique arrow to add. So, for example `PathOneArrowLongerAtSource` requires the source of `path` to have indegree exactly 1, as measured by `InDegreeOfVertex` (**QPA: InDegreeOfVertex**). This is always the situation with 1-regular quivers, where these operations are most intended to be used.

Chapter 6

Miscellaneous utilities

In this chapter, we document some additional functionalities that have been implemented in SBStrips but which, really, can stand independently of it. Others may find these useful without caring a jot about SB algebras.

6.1 Collected lists

Sometimes it is important to know *where* in a list an element appears. Sometimes, all that matters is *how often* it does. (In mathematical terms, these two ideas respectively correspond to a *sequence* of elements and the multiset of values it takes.) One can of course move from knowing the positions of elements to just knowing their frequency. This is a strict loss of information, but usually not a loss of very important information.

GAP implements this functionality using `Collected` (**Reference: Collected**). Calls to this operation yield lists that store information in a more economical, if slightly less informative, fashion, of which SBStrips makes great use. Using `Collected` on a list `list` returns another list, detailing the different elements appearing in `list` and their *multiplicity* (ie, number of instances) in `list`.

Example

```
gap> list := [ "s", "b", "s", "t", "r", "i", "p", "s" ];
[ "s", "b", "s", "t", "r", "i", "p", "s" ]
gap> clist := Collected( list );
[ [ "b", 1 ], [ "i", 1 ], [ "p", 1 ], [ "r", 1 ], [ "s", 3 ],
  [ "t", 1 ] ]
gap> entry := clist[5];
[ "s", 3 ]
```

In the above example, the entry `[ "s", 3 ]` in `clist` tells us that the element `"s"` appears `3` times in `list`. In other words, `"s` has *multitplicity* `3` (in `list`).

In this documentation, we will use the terms *elements* and *multiplicities* respectively to mean the first and second entries of entries of a collected list. So, in the above example, the elements of `clist` are `"b"`, `"i"`, `"p"`, `"r"`, `"s"` and `"t"` and their respective multiplicities are `1`, `1`, `1`, `1`, `3` and `1`.

What characterises a collected list is that all of its entries are lists of length 2, the second being a positive integer. Elements may be repeated. This doesn't happen from simple uses of `Collected`, but can result from combining several collected lists, for instance with `Collected` (**Reference: Collected**) or `Append` (**Reference: Append**).

Example

```
gap> hello := Collected( [ "h", "e", "l", "l", "o" ] );
[ [ "e", 1 ], [ "h", 1 ], [ "l", 2 ], [ "o", 1 ] ]
gap> world := Collected( [ "w", "o", "r", "l", "d" ] );
[ [ "d", 1 ], [ "l", 1 ], [ "o", 1 ], [ "r", 1 ], [ "w", 1 ] ]
gap> hello_world := Concatenation( hello, world );
[ [ "e", 1 ], [ "h", 1 ], [ "l", 2 ], [ "o", 1 ], [ "d", 1 ],
  [ "l", 1 ], [ "o", 1 ], [ "r", 1 ], [ "w", 1 ] ]
gap> IsCollectedList( hello_world );
true
```

Here, the element "l" appears twice in hello_world, first with multiplicity 2 and then again with multiplicity 1. The element "o" also appears twice with multiplicity 1 each time. Despite this repetition, hello_world is still a collected list. It may be "tidied up" using Recollected (6.1.6).

Example

```
gap> Recollected( hello_world );
[ [ "e", 1 ], [ "h", 1 ], [ "l", 3 ], [ "o", 2 ], [ "d", 1 ],
  [ "r", 1 ], [ "w", 1 ] ]
```

6.1.1 IsCollectedList

▷ IsCollectedList(*list*) (property)

Argument: *list*, a list

Returns: true if all entries of *list* are lists of length 2 having a positive integer in their second entry, and false otherwise.

This property will return true on lists returned from the GAP operation Collected (**Reference: Collected**), as well as on combinations of such lists using Concatenation (**Reference: concatenation of lists**) or Append (**Reference: Append**). This is the principal intended use of this property.

When this document refers to a *collected list*, it means a list for which IsCollectedList returns true.

6.1.2 IsCollectedDuplicateFreeList

▷ IsCollectedDuplicateFreeList(*clist*) (property)

Argument: *clist*

Returns: true if *clist* is a collected list with no repeated elements

In particular, if *clist* was created by applying Collected (**Reference: Collected**) to a duplicate-free list (see IsDuplicateFreeList (**Reference: IsDuplicateFreeList**)), then this property will return true. This is the principal intended use of this property.

6.1.3 IsCollectedHomogeneousList

▷ IsCollectedHomogeneousList(*clist*) (property)

Argument: *clist*, a collected list

Returns: true if the elements of clist form a homogeneous list, and false otherwise

If `obj` is the result of applying `Collected` (**Reference: Collected**) to a homogeneous list, then this property returns `true`. This is the principal intended use of this property.

6.1.4 ElementsOfCollectedList

▷ `ElementsOfCollectedList(`*`clist`*`)` (operation)

Argument: *`clist`*, a collected list.
Returns: the elements of *`clist`*.

6.1.5 MultiplicityOfElementInCollectedList

▷ `MultiplicityOfElementInCollectedList(`*`obj, clist`*`)` (operation)

Arguments: *`obj`*, an object; *`clist`*, a collected list.
Returns: a nonnegative integer, namely the (total) multiplicity of *`obj`* in *`clist`*.

6.1.6 Recollected

▷ `Recollected(`*`clist`*`)` (operation)

Argument: *`clist`*, a collected list
Returns: a collected list, removing repeated elements in *`clist`* and totalling their multiplicities.
If *`clist`* contains entries with matching first entries, say `[ obj, n ]` and `[ obj, m ]`, then it will combine them into a single entry `[ obj, n+m ]` with totalised multiplicity. This can be necessary when dealing with concatenations (see `Concatenation` (**Reference: concatenation of lists**)) of collected lists.

6.1.7 Uncollected

▷ `Uncollected(`*`clist`*`)` (operation)

Argument: *`clist`*, a collected list
Returns: a (flat) list, where each element in *`clist`* appears with the appropriate multiplicity

6.1.8 CollectedLength

▷ `CollectedLength(`*`clist`*`)` (attribute)

Argument: *`clist`*, a collected list
Returns: the sum of the multiplicities in *`clist`*

6.1.9 IsCollectedSublist

▷ `IsCollectedSublist(`*`sublist, superlist`*`)` (operation)

Arguments: *`sublist`* and *`superlist`*, two collected lists **Returns:** `true` if each element of *`sublist`* occurs in *`superlist`* with multiplicity as least that in *`sublist`*, and `false` otherwise.

6.1.10 CollectedListElementwiseFunction

▷ `CollectedListElementwiseFunction(`*`clist, func`*`)` (operation)

Arguments: *`clist`*, a collected list; *`func`*, a function

Returns: a new collected list, obtained from *`clist`* by applying *`func`* to each element.

If *`func`* returns lists (perhaps because it implements a "many-valued function"), consider using `CollectedListElementwiseListValuedFunction` (6.1.11) instead.

6.1.11 CollectedListElementwiseListValuedFunction

▷ `CollectedListElementwiseListValuedFunction(`*`clist, func`*`)` (operation)

Arguments: *`clist`*, a collected list; *`func`*, a function (presumed to return lists of objects).

Returns: a new collected list.

Imagine *`clist`* were unpacked into a flat list, *`func`* were applied to each element of the flat list in turn and the result concatenated then collected. That is what this operation returns (although it determines the result more efficiently than the procedure just described).

Appendix A

Example algebras

A.1 The function

For your convenience, SBStrips comes bundled with 5 SB algebras built in. We detail these algebras in this appendix. They may be obtained by calling `SBStripsExampleAlgebra` (A.1.1).

A.1.1 SBStripsExampleAlgebra

▷ `SBStripsExampleAlgebra(n)` (function)

Arguments: `n`, an integer between `1` and `5` inclusive
Returns: a SB algebra
Calling this function with argument `1`, `2`, `3`, `4` or `5` respectively returns the algebras described in subsections A.2.1, A.2.2, A.2.3, A.2.4 or A.2.5.

A.2 The algebras

Each algebra is of the form $KQ/\langle\rho\rangle$, where K is the field `Rationals` in GAP and where Q and ρ are respectively a quiver and a set of relations. These change from example to example.
The LaTeX version of this documentation provides pictures of each quiver.

A.2.1 Algebra 1

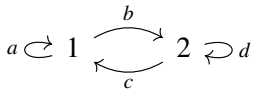
The quiver and relations of this algebra are specified to QPA as follows.

Example

```
gap> quiv := Quiver(
> 2,
> [ [ 1, 1, "a" ], [ 1, 2, "b" ], [ 2, 1, "c" ], [ 2, 2, "d" ] ]
> );
<quiver with 2 vertices and 4 arrows>
pa := PathAlgebra( Rationals, quiv );
<Rationals[<quiver with 2 vertices and 4 arrows>]>
gap> rels := [
> pa.a * pa.a, pa.b * pa.d, pa.c * pa.b, pa.d * pa.c,
> pa.c * pa.a * pa.b, (pa.d)^4,
> pa.a * pa.b * pa.c - pa.b * pa.c * pa.a
```

```
> ];
[ (1)*a^2, (1)*b*d, (1)*c*b, (1)*d*c, (1)*c*a*b,
  (1)*d^4, (1)*a*b*c+(-1)*b*c*a ]
```

Here is a picture of the quiver.



The relations of this algebra are chosen so that the nonzero paths of length 2 are: `a*b`, `b*c`, `c*a`, `d*d`.

The simple module associated to vertex `v2` has infinite syzygy type.

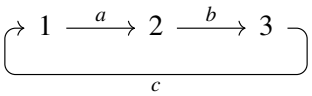
A.2.2 Algebra 2

The quiver and relations of this algebra are specified to QPA as follows.

Example

```
gap> quiv := Quiver(
> 3,
> [ [ 1, 2, "a" ], [ 2, 3, "b" ], [ 3, 1, "c" ] ]
> );
<quiver with 3 vertices and 3 arrows>
gap> pa := PathAlgebra( Rationals, quiv );
<Rationals[<quiver with 3 vertices and 3 arrows>]>
gap> rels := NthPowerOfArrowIdeal( pa, 4 );
[ (1)*a*b*c*a, (1)*b*c*a*b, (1)*c*a*b*c ]
```

Here is a picture of the quiver.



(In other words, this quiver is the 3-cycle quiver, and the relations are the paths of length 4.) The nonzero paths of length 2 are: `a*b`, `b*c`, `c*a`.

This algebra is a Nakayama algebra, and so has finite representation type. *A fortiori*, it is syzygy-finite.

A.2.3 Algebra 3

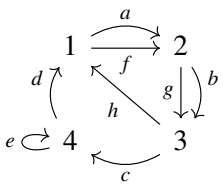
The quiver and relations of this algebra are specified to QPA as follows.

Example

```
gap> quiv := Quiver(
> 4,
> [ [1,2,"a"], [2,3,"b"], [3,4,"c"], [4,1,"d"], [4,4,"e"], [1,2,"f"],
>   [2,3,"g"], [3,1,"h"] ]
> );
<quiver with 4 vertices and 8 arrows>
gap> pa := PathAlgebra( Rationals, quiv );
<Rationals[<quiver with 4 vertices and 8 arrows>]>
gap> rels := [
```

```
> pa.a * pa.g, pa.b * pa.h, pa.c * pa.e, pa.d * pa.f,
> pa.e * pa.d, pa.f * pa.b, pa.g * pa.c, pa.h * pa.a,
> pa.a * pa.b * pa.c * pa.d * pa.a - ( pa.f * pa.g * pa.h )^2 * pa.f,
> pa.d * pa.a * pa.b * pa.c - ( pa.e )^3,
> pa.c * pa.d * pa.a * pa.b * pa.c,
> ( pa.h * pa.f * pa.g )^2 * pa.h
> ];
[ (1)*a*g, (1)*b*h, (1)*c*e, (1)*d*f, (1)*e*d, (1)*f*b, (1)*g*c,
  (1)*h*a, (1)*a*b*c*d*a+(-1)*f*g*h*f*g*h*f, (-1)*e^3+(1)*d*a*b*c,
  (1)*c*d*a*b*c, (1)*h*f*g*h*f*g*h ]
```

Here is a picture of the quiver.



The relations of this algebra are chosen so that the nonzero paths of length 2 are: `a*b`, `b*c`, `c*d`, `d*a`, `e*e`, `f*g`, `g*h` and `h*f`.

A.2.4 Algebra 4

The quiver and relations of this algebra are specified to QPA as follows.

Example

```
gap> quiv := Quiver(
> 8,
> [ [ 1, 1, "a" ], [ 1, 2, "b" ], [ 2, 2, "c" ], [ 2, 3, "d" ],
>   [ 3, 4, "e" ], [ 4, 3, "f" ], [ 3, 4, "g" ], [ 4, 5, "h" ],
>   [ 5, 6, "i" ], [ 6, 5, "j" ], [ 5, 7, "k" ], [ 7, 6, "l" ],
>   [ 6, 7, "m" ], [ 7, 8, "n" ], [ 8, 8, "o" ], [ 8, 1, "p" ] ]
> );
<quiver with 8 vertices and 16 arrows>
gap> pa := PathAlgebra( Rationals, quiv );
<Rationals[<quiver with 8 vertices and 16 arrows>]>
gap> rels := [
> pa.a * pa.a, pa.b * pa.d, pa.c * pa.c, pa.d * pa.g, pa.e * pa.h,
> pa.f * pa.e, pa.g * pa.f, pa.h * pa.k, pa.i * pa.m, pa.j * pa.i,
> pa.k * pa.n, pa.l * pa.j,
> pa.m * pa.l, pa.n * pa.p, pa.o * pa.o, pa.p * pa.b,
> pa.a * pa.b * pa.c * pa.d,
> pa.e * pa.f * pa.g * pa.h,
> pa.g * pa.h * pa.i * pa.j * pa.k,
> pa.c * pa.d * pa.e - pa.d * pa.e * pa.f * pa.g,
> pa.f * pa.g * pa.h * pa.i - pa.h * pa.i * pa.j * pa.k * pa.l,
> pa.j * pa.k * pa.l * pa.m * pa.n - pa.m * pa.n * pa.o,
> pa.o * pa.p * pa.a * pa.b - pa.p * pa.a * pa.b * pa.c
> ];
```

The relations of this algebra are chosen so that the nonzero paths of length 2 are: `a*b`, `b*c`, `c*d`, `d*e`, `e*f`, `f*g`, `g*h`, `h*i`, `i*j`, `j*k`, `k*l`, `l*m`, `m*n`, `n*o`, `o*p` and `p*a`.

A.2.5 Algebra 5

The quiver and relations of this algebra are specified to QPA as follows.

Example

```
gap> quiv := Quiver(
> 4,
> [ [ 1, 2, "a" ], [ 2, 3, "b" ], [ 3, 4, "c" ], [ 4, 1, "d" ],
>   [ 1, 2, "e" ], [ 2, 3, "f" ], [ 3, 1, "g" ], [ 4, 4, "h" ] ]
> );
<quiver with 4 vertices and 8 arrows>
gap> pa := PathAlgebra( Rationals, quiv5 );
<Rationals[<quiver with 4 vertices and 8 arrows>]>
gap> rels := [
> pa.a * pa.f, pa.b * pa.g, pa.c * pa.h, pa.d * pa.e, pa.e * pa.b,
> pa.f * pa.c, pa.g * pa.a, pa.h * pa.d,
> pa.b * pa.c * pa.d * pa.a * pa.b * pa.c,
> pa.d * pa.a * pa.b * pa.c * pa.d * pa.a,
> ( pa.h )^6,
> pa.a * pa.b * pa.c * pa.d * pa.a * pa.b -
>     pa.e * pa.f * pa.g * pa.e * pa.f * pa.g * pa.e * pa.f,
> pa.c * pa.d * pa.a * pa.b * pa.c * pa.d -
>     pa.g * pa.e * pa.f * pa.g * pa.e * pa.f * pa.g
> ];
[ (1)*a*f, (1)*b*g, (1)*c*h, (1)*d*e, (1)*e*b, (1)*f*c, (1)*g*a,
  (1)*h*d, (1)*b*c*d*a*b*c, (1)*d*a*b*c*d*a, (1)*h^6,
  (1)*a*b*c*d*a*b+(-1)*e*f*g*e*f*g*e*f,
  (1)*c*d*a*b*c*d+(-1)*g*e*f*g*e*f*g ]
```

The relations of this algebra are chosen so that the nonzero paths of length 2 are: `a*b`, `b*c`, `c*d`, `d*a`, `e*f`, `f*g`, `g*e`, `h*h`.

References

[All] J. Allen. PhD thesis. Work in progress. 10, 16

[ASS06] I. Assem, D. Simson, and A. Skowronski. *Elements of the Representation Theory of Associative Algebras, volume I: Techniques of Representation Theory*. Number 65 in London Mathematical Society student texts. Cambridge University Press, 2006. 18, 19

[Ben95] D. J. Benson. *Representations and Cohomology, I: Basic representation theory of finite groups and associative algebras*. Number 30 in Cambridge Studies in Advanced Mathematics. Cambridge University Press, 1995. 19

[Dro80] J. A. Drozd. Tame and wild matrix problems. In P. Gabriel and V. Dlab, editors, *Representation Theory II*, volume 832 of *Lecture Notes in Mathematics*, pages 242–258. Springer, Berlin, Heidelberg, 1980. 19

[Gal] J. Galstad. PhD thesis. Work in progress. 20

[GHZ96] K. R. Goodearl and B. Huisgen-Zimmermann. Repetitive resolutions over classical orders and finite dimensional algebras. In I. Reiten, S. O. Smalo, and O. Solberg, editors, *Algebras and modules II: Eighth International Conference on Representations of Algebras, August 4-10, 1996, Geiranger, Norway*, volume 24 of *CMS Conference Proceedings*, pages 205–226. Canadian Mathematical Society, 1996. 20

[HZ16] B. Huisgen-Zimmermann. Representation-tame algebras need not be homologically tame. *Algebras and Representation Theory*, 19:943–956, 2016. 20

[LM04] S. Liu and J.-P. Morin. The strong no loop conjecture for special biserial algebras. *Proceedings of the American Mathematical Society*, 132(12):3513–3523, 2004. 5, 20

[Ric19] J. Rickard. Unbounded derived categories and the finitistic dimension conjecture. *Advances in Mathematics*, 354, 2019. 20

[WW85] B. Wald and J. Waschbüsch. Tame biserial algebras. *Journal of Algebra*, 95:480–500, 1985. 20

Index