SBStrips

Version 0.6.0

Joe Allen

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" – 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 (it explicitly gives the string graphs describing each summand of $\Omega^1(X)$ from that describing X) and elementary (they cleverly choose a basis of the projective cover $P(X)$ of X which disjointly combines bases of $\Omega^1(X)$ and X), being valid regardless of the characteristic of K or whether it is algebraically closed. It follows 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 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 GAPDoc. 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 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 those people, as are the exercises in Appendix [ADD REFERENCE] (and their solutions in [ADD REFERENCE]).

In this chapter, we demonstrate the SBStrips package. We also give some commentary.

2.1 How to teach a special biserial algebra to GAP

Before discussing string modules, we have to specify a SB algebra to GAP. We use functionality from the QPA package for this.

The first 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 (record this as the integer "`2`"), then 1 with negative (record this as "`-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` (3.2.1) 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 "reversed" 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(graph)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` (3.4.1), 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 N syzygies very easily (but refer to the 2.4 for a discussion of an efficient approach when N is big). 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.

Note 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` (3.4.3) and `CollectedNthSyzygyOfStrip` (3.4.4).

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 5.1.

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 [All21]) 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),
      28610320653810477165032088685001500201865067503083660 ],
  [ (v2)^-1(v2), 24012263187173292438733091914788756514219413052446981 ]
    ,
  [ (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` (3.3.1).

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 are in one-to-one correspondence with paths in the SB algebra. There is a method for `Stripify` (3.2.1) 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` (3.3.3) 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` (3.3.4) is similar.

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

If a projective or injective module over an SB algebra is not a string module, then it must be *p*rojective, *i*njective and *n*onuniserial The SBStrips package calls these *pin* modules. No tools for pin modules are implemented in SBStrips. We only mention the terminology in passing because the inner workings of SBStrips occasionally make oblique references to them.

2.6 Tests with strips

Let `alg1` be the Nakayama algebra KQ/J^4, where Q is the 3-cycle quiver, where J is the 4th power of the arrow ideal of KQ, and where K is the field of rationals (not that that matters at all).

This algebra is chosen because it is much more boring than `alg2`. It's certainly a monomial algebra, and therefore *syzygy finite*: this means there exists some integer m and some finite-dimensional module M such that the mth syzygy $\Omega^m(X)$ of *any* `alg1`-module X is a direct summand of some (finite) direct sum of copies of M. (This is proven explicitly in [ZH91, Thm I] for $m = 2$ and M the direct sum of right ideals of `alg1`.) But we can say actually say something stronger. It is a Nakayama algebra, and therefore *representation finite*: this means there are only finitely many isomorphism classes of alg1-modules. (For a proof, see [ASS06, Thm V.3.5].) From this it is trivial to see that

$$\{M \text{ an indecomposable module}\colon M \text{ is a direct summand of } \Omega^m(X) \text{ for some } m\}$$

contains finitely many isomorphism types, for any alg1-module X. This is the condition that X have *finite syzygy type*.

2.7 Modules from strips

This strip s represents a module over alg2. In the literature such modules are called string modules, but maybe here we could call them strip modules? Whatever you want to call it, that module can be made in GAP using ModuleOfStrip (3.6.1).

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 ]>
```

(This creates a quiver representation, using QPA.)

You can turn a list of strips into a list of modules using ModuleOfStrip (3.6.1). You can turn a collected list (see 5.1) of strips into a collected list of modules using ModuleOfStrip (3.6.1). If you want to turn a list or collected list of strips into a single module, namely the direct sum of all the modules represented by strips in your list, then you want to call DirectSumModuleOfStrip.

Chapter 3

Strips

3.1 Introduction

Some introductory text here

3.2 Constructing strips

3.2.1 Stripify

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

Arguments: *arr*, the residue of an arrow in a special biserial algebra (see 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).

(Remember that residues of arrows in an quiver algebra can be easily accessed using the `\.` operation. See . (**QPA: . for a path algebra**) for details and see below for examples.)

Returns: the strip specified by this data

Recall that SBStrips uses strip objects to represent the kind of decorated graph that representation theorists call "strings". Now, suppose you draw that string 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.

(Of course, this method assumes that the string contains at least one arrow. There is a different, easier, method for strings comprising only a single vertex. Namely `Stripify` (3.2.1) called with the residue of a vertex.)

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.

▷ `Stripify(path)` (method)

Arguments: *path*, the residue (in a special biserial algebra) of some path.
(Remember that residues of vertices and arrows can be easily accessed using . (**QPA: . for a path algebra**), and that these can be multiplied together using * (**Reference: ***) to make a path.)

Returns: The strip corresponding to *path*

Recall that uniserial modules are string modules. The uniserial modules of a SB algebra are in 1-to-1 correspondence with the paths p linearly independent from all other paths. Therefore, this path is all you need to specify the strip.

Example
```
gap> # Include an example here!
```

3.3 Particular strips

3.3.1 SimpleStripsOfSBAlg

▷ `SimpleStripsOfSBAlg(sba)` (attribute)

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

Returns: a list `simple_list`, whose jth entry is the simple strip corresponding to the jth 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.

3.3.2 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` (3.3.1) will occur in this list too.

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

3.3.4 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`.

3.4 Calculating syzygies of strips

3.4.1 SyzygyOfStrip

(This attribute takes 1st syzygies of strips. For higher syzygies, `NthSyzygyOfStrip` (3.4.2) may be more convenient while `CollectedNthSyzygyOfStrip` (3.4.4) may be more efficient.)

▷ `SyzygyOfStrip(strip)` (attribute)

Argument: `strip`, a strip

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

▷ `SyzygyOfStrip(list)` (attribute)

Argument: `list`, a list of strips

Returns: a list of strips, corresponding to the indecomposable direct summands of the syzygy of the strips in `list`

The syzygy of each strip in `list` is calculated. This gives several lists of strips, which are then concatenated.

3.4.2 NthSyzygyOfStrip

(This operation calculates Nth syzygies of strips. For large N – say, $N \geq 10$ – consider using `CollectedNthSyzygyOfStrip` (3.4.4) instead, since it is much more efficient.)

▷ `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`

▷ `NthSyzygyOfStrip(list, N)` (method)

Argument: `list`, a list of strips; `N`, a positive integer

Returns: the `N`th syzygy strips of each strip in `list` in turn, all in a single list

3.4.3 CollectedSyzygyOfStrip

*(This operation calculates 1st syzygies and then collects the result into a collected list, using `Collected` (**Reference: Collected**). It has different methods, depending on whether its input is a*

single strip, a (flat) list of strips or a collected list of strips.)

▷ `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 ) )`.

▷ `CollectedSyzygyOfStrip(list)` (method)

Argument: `list`, a (flat) list of strips
Returns: a collected list, whose elements are the syzygy strips of the strips in `list`
This is equivalent to calling `Collected( SyzygyOfStrip( list ) )`.

▷ `CollectedSyzygyOfStrip(clist)` (method)

Argument: `clist`, a collected list of strips
Returns: a collected list, whose elements are the syzygy strips of the strips in `clist`

3.4.4 CollectedNthSyzygyOfStrip

(This operation calculates Nth syzygies of strips and collects the result into a collected list. It has different methods, depending on whether its input is a single strip, a (flat) list of strips or a collected of strips.)

▷ `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`

▷ `CollectedNthSyzygyOfStrip(list, N)` (method)

Arguments: `list`, a (flat) list of strips; `N`, a positive integer
Returns: a collected list, whose entries are the `N`th syzygies of the strips in `list`

▷ `CollectedNthSyzygyOfStrip(list, N)` (method)

Arguments: `clist`, a collected list of strips; `N`, a positive integer
Returns: a collected list, whose entries are the `N`th syzygies of the strips in `clist`

3.5 Attributes and properties of strips

3.5.1 WidthOfStrip

▷ `WidthOfStrip(strip)` (operation)

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

3.5.2 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.

3.6 Operation on strips

3.6.1 ModuleOfStrip (for a strip)

▷ `ModuleOfStrip(`*`strip`*`)` (method)
▷ `ModuleOfStrip(`*`list`*`)` (method)
▷ `ModuleOfStrip(`*`clist`*`)` (method)

Argument: a strip *`strip`*, or a list *`list`* of strips, or a collected list *`clist`* of strips
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.

Reminder. The indecomposable modules for a SB algebra come in two kinds (over an algebraically closed field, at least). One of those are *string modules*, so-called because they may be described by the decorated graphs that representation theorists call *strings* and which the SBStrips package calls *strips*.

The first method for 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**).

The second and third methods respectively apply the first method to each strip in *`list`* or in *`clist`*, returning a list or collected list of modules.

3.6.2 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`.

3.6.3 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.

3.7 Tests on an SB algebra that use strips

3.7.1 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` (3.3.4) for *sba*, filters out all the `fail`s, and then checks each remaining strip individually using `IsFiniteSyzygyTypeStripByNthSyzygy` (3.6.2) (with second argument *N*).

Author's note. For every special biserial algebra I test, this function returns true for sufficiently large *N*. It suggests that the 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 4

QPA utilities

4.1 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 (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.

4.2 Utilities for 1-regular quivers

4.2.1 Is1RegQuiver

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

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

4.2.2 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`*.

4.2.3 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`*.

4.2.4 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` (4.2.5) of tha quiver. For this reason, it is more user-friendly.

4.2.5 1RegQuivIntActionFunction

▷ `1RegQuivIntActionFunction(quiver)` (attribute)

Argument: `quiver`, a 1-regular quiver (as tested by `Is1RegQuiver` (4.2.1))

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` (4.2.4), since it saves you having to remind SBStrips which quiver you intend to act on.

4.3 Utilities for SB quivers

4.3.1 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.

4.3.2 Is2RegAugmentationOfQuiver

▷ `Is2RegAugmentationOfQuiver(quiver)` (property)

Argument: `quiver`, a quiver

Returns: `true` if `quiver` was constructed by 4.3.1 or if `quiver` was an already 2-regular quiver, and `false` otherwise.

4.3.3 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`.

4.3.4 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 )` 4.3.1.

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 )`.

4.4 Miscellaneous utilities for QPA

What follows are minor additional utilities for QPA.

4.4.1 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**).

4.4.2 ArrowsOfQuiverAlgebra

▷ `ArrowsOfQuiverAlgebra(`*`alg`*`)` (operation)

Argument: *`alg`*, a quiver algebra (see `IsQuiverAlgebra` (**QPA: IsQuiverAlgebra**))
Returns: the residues of the arrows in the defining quiver of *`alg`*, listed together

4.4.3 VerticesOfQuiverAlgebra

▷ `VerticesOfQuiverAlgebra(`*`alg`*`)` (operation)

Argument: *`alg`*, a quiver algebra (see `IsQuiverAlgebra` (**QPA: IsQuiverAlgebra**))
Returns: the residues of the vertices in the defining quiver of *`alg`*, listed together

Chapter 5

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.

5.1 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> elt := 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`, of course, 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` (5.1.4).

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

5.1.1 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`.

5.1.2 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.

5.1.3 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.

5.1.4 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 (`Concatenation` (**Reference: concatenation of lists**)) of collected lists.

5.1.5 Uncollected

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

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

5.1.6 CollectedLength

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

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

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.

$$\begin{tikzcd}
1 \arrow[loop left, "a"] \arrow[r, bend left, "b"] & 2 \arrow[l, bend left, "c"] \arrow[loop right, "d"]
\end{tikzcd}$$

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.

```
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.

$$\begin{tikzcd}
1 \arrow[r, "a"] & 2 \arrow[r, "b"] & 3 \arrow[ll, bend left, "c"]
\end{tikzcd}$$

(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.

```
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.

$$\begin{tikzcd}
1 \arrow[r, bend left, "a"] \arrow[r, "f"'] & 2 \arrow[d, "g"'] \arrow[d, bend left, "b"] \\
4 \arrow[u, bend left, "d"] \arrow[loop left, "e"] & 3 \arrow[ul, "h"] \arrow[l, bend left, "c"]
\end{tikzcd}$$

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

[All21] J. Allen. *PhD thesis (title TBC)*. PhD thesis, School of Mathematics, University of Bristol, expected 2021. Work in progress. 10

[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. 13

[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

[ZH91] B. Zimmermann-Huisgen. Predicting syzygies over monomial relation algebras. *Manuscripta mathematica*, 70:157–182, 1991. 12

Index