SBStrips

Version 0.6.0

Joe Allen

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

Abstract

We implement a discrete model of special biserial algebras and, more to the point, their string modules. We represent string modules using new objects that we call *strips*. Using these, we efficiently calculate syzygies of string modules in terms of the strips that represent them.

This package builds on, an interfaces with, the QPA package. This package was created as part of the author's PhD thesis.

Copyright

Acknowledgements

I thank my PhD supervisor Professor 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.

Contents

Chapter 1

Introduction

1.1 Why "strips", not "strings"?

First, some context. Representation theorists use the word *string* to mean a decorated graph that describes a module; fittingly, this module is then dubbed a *string module*. Liu and Morin [LM04] showed that the syzygy of a string module over a special biserial (SB) algebra is a direct sum of string modules. This means that, in essence, we can forget about the modules *per se* and compute syzygies just at the level of strings. Liu and Morin's proof is constructive, specifying how to obtain the strings indexing the indecompsoable direct summands of syzygy from thae string indexing the original module. Their language sketches how to spot patterns appearing "from one syzygy to the next", but it does not scale in a particularly transparent way. For example, I believe it does not lend itself to clearly seeing asymptotic behaviour of syzygies of string modules. My research has aimed, in part, to provide a more robust language: one which lays bare more patterns in the syzygies of string modules over SB algebras in a manner amenable to computer calculation.

One key ingredient is a slight refinement of the definition of a string. Really, this differs from the established definition only in technical ways, the effect being to disambiguate how the graph is decorated so that the syzygy calculation is streamlined. In my thesis, I propose the term *strip* for this refined notion of a string. A happy side-effect of this name change is that it avoids the clash with what GAP already thinks "string" means.

In brief: if whenever you read the word "strip" here, you imagine that it means the kind of decorated graph that representation theorists call a "string", then you won't go too far wrong.

1.2 Aims

Some text to go here!

1.3 Installation

Some text to go here!

Chapter 2

Worked example

2.1 Strips, aka "strings for special biserial algebras"

This package is principally for "strings and their syzygies". Strings are defined over special biserial (SB) algebras. Our first job is to tell GAP about a SB algebra. We'll do this using tools from QPA. If the following doesn't make sense to you, then see the QPA documentation [QPA18].

An important rule is that the SB algebra be presented by *monomial relations* and *skew commutativity relations*. A monomial relation is just a path p (or more generally λp for some coefficient in the ambient field, here `Rationals`). A (skew) commutativity relation is a linear difference $\lambda p - \mu q$ of paths p, q having common source and target, where λ, μ are nonzero coefficients. It is well-known that all SB algebras admit such a presentation; SBStrips takes it for granted you have used one.

Example

```
gap> q2 := Quiver( 2, [ [1,1,"a"], [1,2,"b"], [2,1,"c"], [2,2,"d"] ] );;
gap> kq2 := PathAlgebra( Rationals, q2 );;
gap> rels2 := [ kq2.a * kq2.a, kq2.b * kq2.d, kq2.c * kq2.b,
> kq2.d * kq2.c, kq2.c * kq2.a * kq2.b, (kq2.d)^4,
> kq2.a * kq2.b * kq2.c - kq2.b * kq2.c * kq2.a ];;
gap> gb2 := GBNPGroebnerBasis( rels2, kq2 );;
gap> ideal2 := Ideal( kq2, gb2 );;
gap> GroebnerBasis( ideal2, gb2 );;
gap> alg2 := kq2/ideal2;
<Rationals[<quiver with 2 vertices and 4 arrows>]/
<two-sided ideal in <Rationals[<quiver with 2 vertices and 4 arrows>]>
, (7 generators)>>
```

This defines a special biserial algebra `alg2`. The following is what representation theorists call a string over `alg2`, but which we'll prefer to call a *strip*. (For reasons why, see Section 1.1.)

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

Note in particular that the "first arrow" in this strip is a and it has exponent -1 (which means it points to the left). This information (plus a bit extra) gets used when creating the strip in GAP via the operation `Stripify` (3.2.1).

Example

```
gap> s := Stripify( alg2.a, -1, [2, -1, 1, -1, 1, -2, 1] );
(a)^-1(b*c) (a)^-1(b) (d)^-1(c) (c*a)^-1(d)
```

2.2 Modules from strips

Representation theorists will know that this strip `s` corresponds to an indecomposable module over `alg2`. In the literature they're 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).

A technicality to bear in mind is that that module is implemented as a representation of the quiver over which `alg2` was defined. You'll find details about quiver representations in the QPA documentation.

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` (3.6.1). You can turn a collected list (see below *ADD REFERENCE*) 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, it is better to call `DirectSumModuleOfStrip`.

2.3 Syzygies of strips

Now, you *can* calculate the syzygy of `X` using QPA's function `1stSyzygy` (**QPA: 1stSyzygy**) on `X` if you really want. It'll give you the syzygy module `\Omega^1(X)` as another quiver representation.

However, you should know that the syzygy of a string module X is a direct sum of string modules. Suppose we write this as $\Omega^1(X) = X_1 \oplus \cdots \oplus X_m$. It turns out that syzygies may be computed in an algorithmic fashion, just at the level of "strings". In other words, you give me the "string" describing X and I give you the "strings" describing each summand X_j of its syzygy in turn. This result was proved constructively by Liu and Morin in their 2004 paper *ADD REFERENCE* using a pseudoalgorithm. The purpose of the SBStrips package is to formalize this pseudoalgorithm and implement it in GAP. Instead of "strings", which can be ambiguous, we use strips. Our added care pays off when examining asymoptotic syzygy behavior of strings.

Recall our strip `s` from above. Let's start calculating its syzygies. The operation `SyzygyOfStrip` 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
```

This example shows that the syzygy of `s` has 3 indecomposable summands.

Of course, there's no reason to stop at 1st syzygies. SBStrips is able to take higher syzygies very easily (but refer to the next section for a discussion of an efficient approach). 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 occur multiple times in this `4th_syz`. (What this means mathematically is that many of those summands are isomorphic.) If you want to remove duplicates from the above list (which is like looking at just the isomorphism types of modules in the direct sum), then the most efficient way is with `Set` (**Reference: Set**). Alternatively, you can use `Collected` (**Reference: Collected**). This turn the list into something that a mathematician might call a multiset. That is, the distinct strips are recorded along with their frequency in the list.

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 ] ]
```

For example, the second output means that `(v2)^-1(v2)` occurs 3 times in `4th_syz` while `(v1)^-1(a)` occurs 6 times.

We call these "multisets" *collected lists*. The SBStrips package has several built-in functionalities for taking such "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 ] ]
```

(In the next section, we pause to discuss the efficiency of these "collected" methods.)

We reiterate that the lists (resp. collected lists) of strips returned by `SyzygyOfStrip` (resp. `CollectedSyzygyOfStrip`) and its `Nth` variants may be turned into lists (resp. collected lists) of quiver representations using `ModuleOfStrip`. These may alternatively be turned into a single direct sum module using `DirectSumModuleOfStrips`.

2.4 Computing higher syzygies efficiently

A central point in my thesis *(reference!)* 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.) 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, consider the 20th syzygy of `s`. The following calculation shows that it has 344732 distinct summands (many of which will be isomorphic); this took over 2 minutes to perform on my device.

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 100th syzygy in not much more).

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, 100 );
[ [ (v2)^-1(c) (c*a)^-1(v2), 980795301785860345365O0564 ],
  [ (v2)^-1(v2), 823168506365148666776570 75 ],
  [ (v1)^-1(a), 1803963808151009012141576 38 ],
  [ (v2)^-1(d^2), 1514042931066841836012232 21 ], [ (v2)^-1(d), 1 ],
  [ (a)^-1(b*c) (a)^-1(v1), 1 ] ]
gap> time;
297
```

Be advised that, even in this easier-to-store form, the integers involved may become to big for GAP to handle. Efficient storage only increases the upper bound on information we can store; it doesn't remove it!

2.5 Other important strips

First, some general theory about finite-dimensional algebras. Recall that (the isomorphism classes of) the simple modules over an SB algebra are in one-to-one correspondence with the vertices of its ordinary quiver. The same is true for the indecomposable projective modules and the indecomposable injective modules. (It is also true for any finite-dimensional quiver algebra, more generally.) In this section, suppose that the vertices are $i_1,\dots,i_n$, and that the simple, indecomposable projective and indecomposable injective modules associated to vertex i_r are respectively S_r, P_r and I_r.

All of the simple modules over an SB algebra are string modules. The list of strips that describe them can be obtained using the following command.

Example

```
gap> SimpleStripsOfSBAlg( alg2 );
[ (v1)^-1(v1), (v2)^-1(v2) ]
```

The rth entry is the strip describing the rth simple module S_r.

Additionally, some of the indecomposable projective modules are string modules. The attribute `ProjectiveStripsOfSBAlg` (5.5.3) returns a list, whose rth 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` (5.5.4) is similar.

Example

```
gap> ProjectiveStripsOfSbAlg( alg2 );
[ fail, (c*a)^-1(d^3) ]
gap> InjectiveStripsOfSbAlg( alg2 );
[ 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. Such modules, which we call *pin* modules (can you see why?) are not implemented in SBStrips. However, from time to time our notation refers to them obliquely, for instance `IsPinBoundarySyllable` (**??**).)

The uniserial modules are also, in particular, string modules. They are in one-to-one correspondence with the 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. Paths in the SB algebra are created using QPA syntax. Perhaps it is clearest to see an example.

Example

```
gap> Stripify( alg2.a * alg2.b );
(a*b)^-1(v1)
gap> Stripify( alg2.c );
(c)^-1(v2)
gap> Stripify( alg2.d^3 );
(d^3)^-1(v2)
gap> Stripify( alg2.v1 );
(v1)^-1(v1)
```

In the first example, `a` and `b` are the names of arrows in the quiver with which `alg2` was presented. The residue of the arrow `a` in `alg2` is `alg2.a`; similarly, `alg2.b`. Their product `alg2.a * alg2.b` is the residue of the path `a * b` in the quiver (where `a * b` means "a then b"). This is what we mean by a path in the SB algebra: products of (residues of) arrows and vertices in the algebra.

We see that vertices or arrows of the SB algebra (such as `alg2.v1` and `alg2.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> [ s1, s2 ] = SimpleStripsOfSbAlg( alg2 );
true
```

2.6 Tests with strips

Now that we've implement strips, we should play around with them. Let's see some of the fun tests built into SBStrips! For this, we'll introduce the algebra `alg1`. It is 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 $K = \mathbb{Q}$. (Really K could be any field.)

This algebra is chosen because it is much more boring than `alg2`. For instance, as a monomial algebra, `alg1` is *syzygy-finite*. Roughly, this means that the class of syzygies stabilizes. More precisely, it means that there is some integer $0 \leq m < \infty$ and a finite set $\mathscr{S}$ of indecomposable modules such that, for any module X, the indecomposable summands of $\Omega^m(X)$ belong to $\mathscr{S}$. (By a result of Zimmermann Huisgen *REFERENCE!*, $m = 2$ works.) But, in fact, we can say something stronger. Because `alg2` is a Nakayama algebra, it is *representation finite* (so, in fact, $m = 0$ works). Things to test: weakly periodic; finite syzygy type

2.7 Additional examples

Give some other sample algebras here, and do things with them.

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` (5.5.2) 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 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 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.

3.6 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.

Chapter 4

Discrete model for SB algebras and their string modules

In this chapter, I spell out my model.

Chapter 5

Quiver utilities and the overquiver of a SB algebra

5.1 Introduction

Quivers are finite directed graphs. Paths in a given quiver Q can be concatenated in an obvious way, and this concatenation can be extended K-linearly (over a field K) to give an associative, unital algebra KQ called a *path algebra*. A path algebra is infinite-dimensional iff its underlying quiver Q is acyclic. Finite-dimensional *quiver algebras* – that is, finite-dimensional quotient algebras KQ/I of a path algebra KQ by some (frequently admissible) ideal I – are a very important class of rings, whose representation theory has been much studied.

The excellent QPA package implements these objects in GAP. The (far more humdrum) SBStrips package extends QPA's functionality. Quivers constructed using the QPA function `Quiver` (**QPA: Quiver no. of vertices, list of arrows**) belong to the filter `IsQuiver` (**QPA: IsQuiver**), and special biserial algebras are those quiver algebras for which the property `IsSpecialBiserialAlgebra` (**QPA: IsSpecialBiserialAlgebra**) returns `true`.

In this section, we explain some added functionality for quivers and special biserial algebras.

5.2 New property of 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 IsOverquiver

▷ `IsOverquiver(`*`quiver`*`)` (property)

Argument: *`quiver`*, a quiver
Returns: `true` if *`quiver`* was constructed by **??**, and `false` otherwise.

5.3 New attributes of quivers

5.3.1 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` (5.3.2) of tha quiver. For this reason, it is more user-friendly.

5.3.2 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` (5.4.1), since it saves you having to remind SBStrips which quiver you intend to act on.

5.3.3 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.

5.4 Operations on vertices and arrows of quivers

5.4.1 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` (5.3.2) of tha quiver. For this reason, it is more user-friendly.

5.4.2 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*.

5.4.3 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*.

5.5 New attributes for special biserial algebras

5.5.1 OverquiverOfSBAlg

▷ `OverquiverOfSBAlg(sba)` (attribute)

Argument: *sba*, a special biserial algebra
Returns: a quiver `oquiv` with which uniserial *sba*-modules can be conveniently (and unambiguously) represented.

5.5.2 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.

5.5.3 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`.

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

5.6 New function for special biserial algebras

5.6.1 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` (5.5.4) 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 6

Permissible data of a SB algebra

In this chapter, I explain the permissible data of a SB algebra. Largely, this means expanding on *independent paths* and *components*.

Chapter 7

Vertex-indexed sequences and encodings of permissible data

In this chapter, I explain about the source and target encodings of the permissible data of a SB algebra. I also describe vertex-indexed sequences more generally.

Chapter 8

Syllables

In this chapter, I describe syllables.

Chapter 9

Patches

In this chapter, I describe patches.

Chapter 10

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 about SB algebras or what-have-you. Among these, we include minor extensions of functionality for QPA

10.1 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 (10.1.4).

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

10.1.1 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.

10.1.2 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.

10.1.3 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.

10.1.4 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.

10.1.5 Uncollected

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

Argument: *clist*, a collected list

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

10.2 Miscellaneous utilities for QPA

What follows are minor additional utilities for QPA.

10.2.1 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**).

10.2.2 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

10.2.3 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

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(
> 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.2.2 Algebra 2

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.

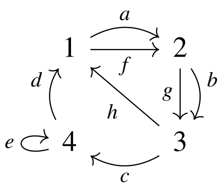
A.2.3 Algebra 3

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

[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

[QPA18] *QPA – Quivers, path algebras and representations, Version 1.29*, 2018. https://folk.ntnu.no/oyvinso/QPA/. 6

Index