## CONTENTS

1 Path Algebras 1
2 Path algebra elements 7
3 Quiver Homspace 15
4 Quiver Morphisms 19
5 Path Semigroups 29
6 Quiver Paths 39
7 Quiver Representations 45
8 Indices and Tables 71

Python Module Index 73
Index 75
class sage.quivers.algebra.PathAlgebra(k, P, order='negdegrevlex')
Bases: sage.combinat.free_module.CombinatorialFreeModule

Create the path algebra of a quiver over a given field.

Given a quiver $Q$ and a field $k$, the path algebra $kQ$ is defined as follows. As a vector space it has basis the set of all paths in $Q$. Multiplication is defined on this basis and extended bilinearly. If $p$ is a path with terminal vertex $t$ and $q$ is a path with initial vertex $i$ then the product $p * q$ is defined to be the composition of the paths $p$ and $q$ if $t = i$ and 0 otherwise.

INPUT:
- $k$ – field (or commutative ring), the base field of the path algebra
- $P$ – the path semigroup of a quiver $Q$
- $order$ – optional string, one of “negdegrevlex” (default), “degrevlex”, “negdeglex” or “deglex”, defining the monomial order to be used.

OUTPUT:
- the path algebra $kP$ with the given monomial order

NOTE:
Monomial orders that are not degree orders are not supported.

EXAMPLES:

```
sage: P = DiGraph({1:{2:['a']}, 2:{3:['b']}}).path_semigroup()
sage: A = P.algebra(GF(7))
sage: A
Path algebra of Multi-digraph on 3 vertices over Finite Field of size 7
sage: A.variable_names()
('e_1', 'e_2', 'e_3', 'a', 'b')
```

Note that path algebras are uniquely defined by their quiver, field and monomial order:

```
sage: A is P.algebra(GF(7))
True
sage: A is P.algebra(GF(7), order="degrevlex")
False
sage: A is P.algebra(RR)
False
sage: A is DiGraph({1:{2:['a']}}).path_semigroup().algebra(GF(7))
False
```

The path algebra of an acyclic quiver has a finite basis:
The path algebra can create elements from paths or from elements of the base ring:

```
sage: A(5)
5*e_1 + 5*e_2 + 5*e_3
sage: S = A.semigroup()
sage: S
Partial semigroup formed by the directed paths of Multi-digraph on 3 vertices
sage: p = S([(1, 2, 'a')])
sage: r = S([(2, 3, 'b')])
sage: e2 = S([(2, 2)])
sage: x = A(p) + A(e2)
sage: x
a + e_2
sage: y = A(p) + A(r)
sage: y
a + b
```

Path algebras are graded algebras. The grading is given by assigning to each basis element the length of the path corresponding to that basis element:

```
sage: x.is_homogeneous()
False
sage: x.degree()
Traceback (most recent call last):
...  
ValueError: Element is not homogeneous.
sage: y.is_homogeneous()
True
sage: y.degree()
1
sage: A[1]
Free module spanned by [a, b] over Finite Field of size 7
Free module spanned by [a*b] over Finite Field of size 7
```

Element
---
alias of `sage.quivers.algebra_elements.PathAlgebraElement`

arrows() 
Return the arrows of this algebra (corresponding to edges of the underlying quiver).

EXAMPLES:

```
sage: P = DiGraph({1:{2:['a']}, 2:{3:['b', 'c']}, 4:{}}).path_semigroup()
sage: A = P.algebra(GF(5))
sage: A.arrows()
(a, b, c)
```

degree_on_basis(x) 
Return x.degree().

This function is here to make some methods work that are inherited from `CombinatorialFreeModule`.
EXAMPLES:

```python
sage: A = DiGraph({0:{1:['a'], 2:['b']}, 1:{0:['c'], 1:['d']}, 2:{0:['e'], 2:['f']}}).path_semigroup().algebra(ZZ)
sage: A.inject_variables()
Defining e_0, e_1, e_2, a, b, c, d, e, f
sage: X = a+2*b+3*c*e-a*d+5*e_0+3*e_2
sage: X
5*e_0 + a - a*d + 2*b + 3*e_2
sage: X.homogeneous_component(0)  # indirect doctest
5*e_0 + 3*e_2
sage: X.homogeneous_component(1)
a + 2*b
sage: X.homogeneous_component(2)
-a*d
sage: X.homogeneous_component(3)
0
```

`gen(i)`

Return the $i$-th generator of this algebra.

This is an idempotent (corresponding to a trivial path at a vertex) if $i < n$ (where $n$ is the number of vertices of the quiver), and a single-edge path otherwise.

EXAMPLES:

```python
sage: P = DiGraph({1:{2:['a']}, 2:{3:['b', 'c']}, 4:{}}).path_semigroup()
sage: A = P.algebra(GF(5))
sage: A.gens()
(e_1, e_2, e_3, e_4, a, b, c)
sage: A.gen(2)
e_3
sage: A.gen(5)
b
```

`gens()`

Return the generators of this algebra (idempotents and arrows).

EXAMPLES:

```python
sage: P = DiGraph({1:{2:['a']}, 2:{3:['b', 'c']}, 4:{}}).path_semigroup()
sage: A = P.algebra(GF(5))
sage: A.variable_names()
('e_1', 'e_2', 'e_3', 'e_4', 'a', 'b', 'c')
sage: A.gens()
(e_l1, e_2', 'e_3', 'e_4', 'a', 'b', 'c')
sage: A.gens()
(e_1, e_2, e_3, e_4, a, b, c)
```

`homogeneous_component(n)`

Return the $n$-th homogeneous piece of the path algebra.

INPUT:

- $n$ – integer

OUTPUT:

- CombinatorialFreeModule, module spanned by the paths of length $n$ in the quiver

EXAMPLES:
The multiplicative identity of a path algebra is the sum of the basis elements corresponding to the trivial paths at each vertex.

EXAMPLES:
sage: A = DiGraph({1:{2:['a']}, 2:{3:['b']}}).path_semigroup().algebra(QQ)
sage: A.one()
e_1 + e_2 + e_3

order_string()

Return the string that defines the monomial order of this algebra.

EXAMPLES:

sage: P1 = DiGraph({1:{1:['x','y','z']}}).path_semigroup().algebra(GF(25,'t'), order="degrevlex")
sage: P2 = DiGraph({1:{1:['x','y','z']}}).path_semigroup().algebra(GF(25,'t'), order="negdeglex")
sage: P3 = DiGraph({1:{1:['x','y','z']}}).path_semigroup().algebra(GF(25,'t'), order="deglex")
sage: P1.order_string()
'negdegrevlex'
sage: P2.order_string()
'degrevlex'
sage: P3.order_string()
'negdeglex'
sage: P4.order_string()
'deglex'

quiver()

Return the quiver from which the algebra self was formed.

OUTPUT:

• DiGraph, the quiver of the algebra

EXAMPLES:

sage: P = DiGraph({1:{2:['a', 'b']}}).path_semigroup()
sage: A = P.algebra(GF(3))
sage: A.quiver() is P.quiver()
True

semigroup()

Return the (partial) semigroup from which the algebra self was constructed.

Note: The partial semigroup is formed by the paths of a quiver, multiplied by concatenation. If the quiver has more than a single vertex, then multiplication in the path semigroup is not always defined.

OUTPUT:

• the path semigroup from which self was formed (a partial semigroup)

EXAMPLES:

sage: P = DiGraph({1:{2:['a', 'b']}}).path_semigroup()
sage: A = P.algebra(GF(3))
sage: A.semisigroup() is P
True

sum(iter_of_elements)

Returns the sum of all elements in iter_of_elements
INPUT:

- \texttt{iter_of_elements:} iterator of elements of \texttt{self}

NOTE:

It overrides a method inherited from \texttt{CombinatorialFreeModule}, which relies on a private attribute of elements—an implementation detail that is simply not available for \texttt{PathAlgebraElement}.

EXAMPLES:

```
sage: A = DiGraph({0:{1:['a'], 2:['b']}, 1:{0:['c'], 1:['d']}, 2:{0:['e'], 2:['f']}).path_semigroup().algebra(ZZ)
sage: A.inject_variables()
Defining e_0, e_1, e_2, a, b, c, d, e, f
sage: A.sum((a, 2*b, 3*c*e, -a*d, 5*e_0, 3*e_2))
5*e_0 + a - a*d + 2*b + 3*e_2
```
PATH ALGEBRA ELEMENTS

AUTHORS:

- Simon King (2015-08)

class sage.quivers.algebra_elements.PathAlgebraElement

Bases: sage.structure.element.RingElement

Elements of a PathAlgebra.

NOTE:

Upon creation of a path algebra, one can choose among several monomial orders, which are all positive or negative degree orders. Monomial orders that are not degree orders are not supported.

EXAMPLES:

After creating a path algebra and getting hold of its generators, one can create elements just as usual:

```
sage: A = DiGraph({0:{1:['a'], 2:['b']}, 1:{0:['c'], 1:['d']}, 2:{0:['e'], 2:['f']}).path_semigroup().algebra(ZZ)
sage: A.inject_variables()
Defining e_0, e_1, e_2, a, b, c, d, e, f
sage: x = a+2*b+3*c+5*e_0+3*e_2
sage: x
5*e_0 + a + 2*b + 3*c + 3*e_2
```

The path algebra decomposes as a direct sum according to start- and endpoints:

```
sage: x.sort_by_vertices()
[(5*e_0, 0, 0),
 (a, 0, 1),
 (2*b, 0, 2),
 (3*c, 1, 0),
 (3*e_2, 2, 2)]
sage: (x^3+x^2).sort_by_vertices()
[(150*e_0 + 33*a*c, 0, 0),
 (30*a + 3*a*c*a, 0, 1),
 (114*b + 6*a*c*b, 0, 2),
 (90*c + 9*c*a*c, 1, 0),
 (18*c*a, 1, 1),
 (54*c*b, 1, 2),
 (36*e_2, 2, 2)]
```

For a consistency test, we create a path algebra that is isomorphic to a free associative algebra, and compare arithmetic with two other implementations of free algebras (note that the letterplace implementation only allows weighted homogeneous elements):
When the Cython implementation of path algebra elements was introduced, it was faster than both the default implementation and the letterplace implementation of free algebras. The following timings where obtained with a 32-bit operating system; using 64-bit on the same machine, the letterplace implementation has not become faster, but the timing for path algebra elements has improved by about 20%:

```
sage: timeit('pF^5+3*pF^3')  # not tested
1 loops, best of 3: 338 ms per loop
sage: timeit('pP^5+3*pP^3')  # not tested
100 loops, best of 3: 2.55 ms per loop
sage: timeit('pF2^7')       # not tested
10000 loops, best of 3: 513 ms per loop
sage: timeit('pL2^7')       # not tested
125 loops, best of 3: 1.99 ms per loop
sage: timeit('pP2^7')       # not tested
10000 loops, best of 3: 1.54 ms per loop
```

So, if one is merely interested in basic arithmetic operations for free associative algebras, it could make sense to model the free associative algebra as a path algebra. However, standard basis computations are not available for path algebras, yet. Hence, to implement computations in graded quotients of free algebras, the letterplace implementation currently is the only option.

**coefficient** (*P*)

Return the coefficient of a monomial.

**INPUT:**

An element of the underlying partial semigroup.

**OUTPUT:**

The coefficient of the given semigroup element in self, or zero if it does not appear.

**EXAMPLES:**

```
sage: P = DiGraph({1:{1:['x','y','z']}}).path_semigroup().algebra(GF(25,'t'), order="degrevlex")
sage: P.inject_variables()
Defining e_1, x, y, z
sage: p = (x+2*z+1)^3
sage: p
3*z*z*z + 4*x*z*z + 4*z*x*z + 2*x*x*z + 4*z*z*x + 2*x*z*x + 2*z*x*x + x*x*x + 2*z*x + z*x + 3*x*x + z + 3*x + e_1
sage: p.coefficient(sage_eval('x*x*z', P.semigroup().gens_dict()))
2
```
coefficients()  
Returns the list of coefficients.

**Note:** The order in which the coefficients are returned corresponds to the order in which the terms are printed. That is *not* the same as the order given by the monomial order, since the terms are first ordered according to initial and terminal vertices, before applying the monomial order of the path algebra.

**EXAMPLES:**

```sage
sage: P = DiGraph({1:{1:['x','y','z']}}).path_semigroup().algebra(GF(25,'t'), order="degrevlex")
sage: P.inject_variables()
Defining e_1, x, y, z
sage: p = (x+2*z+1)^3
sage: p
3*z*z*z + 4*x*z*z + 4*z*x*z + 2*x*x*z + 4*z*z*x + 2*z*x*z + 2*z*x*x + x*x*x + ...
-2*x*z*x + z*x + 3*x*x + z + 3*x + e_1
sage: p.coefficients()
[3, 4, 2, 4, 2, 1, 2, 1, 3, 1, 3, 1]
```

degree()  
Return the degree, provided the element is homogeneous.  
An error is raised if the element is not homogeneous.

**EXAMPLES:**

```sage
sage: P = DiGraph({1:{1:['x','y','z']}}).path_semigroup().algebra(GF(25,'t'))
sage: P.inject_variables()
Defining e_1, x, y, z
sage: q = (x+y+2*z)^3
sage: q.degree()
3
sage: p = (x+2*z+1)^3
sage: p.degree()
Traceback (most recent call last):
...  
ValueError: Element is not homogeneous.
```

is_homogeneous()  
Tells whether this element is homogeneous.

**EXAMPLES:**

```sage
sage: P = DiGraph({1:{1:['x','y','z']}}).path_semigroup().algebra(GF(25,'t'))
sage: P.inject_variables()
Defining e_1, x, y, z
sage: q = (x+y+2*z)^3
sage: q.is_homogeneous()
True
sage: p = (x+2*z+1)^3
sage: p.is_homogeneous()
False
```
monomial_coefficients()

Return the dictionary keyed by the monomials appearing in this element, the values being the coefficients.

EXAMPLES:

```
sage: P = DiGraph({1:{1:['x','y','z']}}).path_semigroup().algebra(GF(25,'t'), order="degrevlex")
sage: P.inject_variables()
Defining e_1, x, y, z
sage: p = (x+2*z+1)^3
sage: sorted(p.monomial_coefficients().items())
[(x*x*x, 1),
 (z*x*x, 2),
 (x*z*x, 2),
 (z*z*x, 4),
 (x*x*z, 2),
 (z*x*z, 4),
 (x*z*z, 4),
 (z*z*z, 3),
 (x*x, 3),
 (z*x, 1),
 (x*z, 1),
 (z*z, 2),
 (x, 3),
 (z, 1),
 (e_1, 1)]
```

Note that the dictionary can be fed to the algebra, to reconstruct the element:

```
sage: P(p.monomial_coefficients()) == p
True
```

monomials()

Returns the list of monomials appearing in this element.

Note: The order in which the monomials are returned corresponds to the order in which the element’s terms are printed. That is not the same as the order given by the monomial order, since the terms are first ordered according to initial and terminal vertices, before applying the monomial order of the path algebra.

The monomials are not elements of the underlying partial semigroup, but of the algebra.

See also:

support()

EXAMPLES:

```
sage: P = DiGraph({1:{1:['x','y','z']}}).path_semigroup().algebra(GF(25,'t'), order="degrevlex")
sage: P.inject_variables()
Defining e_1, x, y, z
sage: p = (x+2*z+1)^3
sage: p
3*z*z*z + 4*x*z*z + 4*z*x*z + 2*x*x*z + 4*z*z*x + 2*x*z*x + 2*z*x*z + 2*z*z*z +
 2*x*z + x*z + 3*x*x + z + 3*x + e_1
sage: p.monomials()
[z*z*z,
 x*z*z,
 x*x*z,
 x*z*x,
 z*x*z,
 z*x*x,
 x*z*z,
 z*z*x,
 x*x*z,
 x*x*x,
 x*x,
 z,
 e_1]
```
sort_by_vertices()  
Return a list of triples (element, v1, v2), where element is an element whose monomials all have initial vertex v1 and terminal vertex v2, so that the sum of elements is self.

EXAMPLES:

```
sage: A1 = DiGraph({0:{1:['a'], 2:['b'], 3:['c'], 4:['d']}, 1:{0:['e'], 2:['f'], 3:['g']}, 2:{0:['h'], 1:['i'], 3:['j']}, 3:{0:['k'], 1:['l'], 2:['m'], 4:['n']}, 4:{0:['o'], 1:['p'], 2:['q'], 3:['r']}).path_semigroup().algebra(ZZ.quo(15))
sage: A1.inject_variables()
Defining e_0, e_1, e_2, a, b, c, d, e, f
sage: x = (b*e*b*e+4*b+e_0)^2
sage: y = (a*c*b+1)^3
sage: x.sort_by_vertices()  
[(e_0 + 2*b*e*b*e + b*e*b*e*b*e*b*e + b*e*b*e + b*e*b*e + b*e*b, 0, 2), (4*b + 4*b*e*b*e*b, 0, 2)]
sage: sum(c[0] for c in x.sort_by_vertices()) == x
True
sage: y.sort_by_vertices()  
[(e_0, 0, 0), (3*a*c*b, 0, 2), (e_1, 1, 1), (e_2, 2, 2)]
sage: sum(c[0] for c in y.sort_by_vertices()) == y
True
```

support()  
Returns the list of monomials, as elements of the underlying partial semigroup.

Note: The order in which the monomials are returned corresponds to the order in which the element’s terms are printed. That is not the same as the order given by the monomial order, since the terms are first ordered according to initial and terminal vertices, before applying the monomial order of the path algebra.

See also:

monomials()

EXAMPLES:

```
sage: P = DiGraph({1:{l:1:["x","y","z"]}}).path_semigroup().algebra(GF(25,'t'), order="degrevlex")
sage: P.inject_variables()
Defining e_1, x, y, z
sage: p = (x+2*z+1)^3
sage: p
```
support_of_term()
If self consists of a single term, return the corresponding element of the underlying path semigroup.

EXAMPLES:

```
sage: A = DiGraph({0:{1:['a'], 2:['b']}, 1:{0:['c'], 1:['d']}, 2:{0:['e'], 2:['f']}).path_semigroup().algebra(ZZ)
sage: A.inject_variables()
Defining e_0, e_1, e_2, a, b, c, d, e, f
sage: x = 4*a*d*c*b*e
sage: x.support_of_term()
a*d*c*b*e
sage: x.support_of_term().parent() is A.semigroup()
True
sage: (x + f).support_of_term()
Traceback (most recent call last):
  ... ValueError: 4*a*d*c*b*e + f is not a single term
```

terms()
Returns the list of terms.

Note: The order in which the terms are returned corresponds to the order in which they are printed. That is not the same as the order given by the monomial order, since the terms are first ordered according to initial and terminal vertices, before applying the monomial order of the path algebra.

EXAMPLES:

```
sage: P = DiGraph({1:{1:['x','y','z']}}).path_semigroup().algebra(GF(25,'t'), order="degrevlex")
sage: P.inject_variables()
Defining e_1, x, y, z
sage: p = (x+2*z+1)^3
sage: p
(continues on next page)```
3*z*z*z + 4*x*z*z + 4*z*x*z + 2*x*x*z + 4*z*z*x + 2*z*z*x + x*x*x + x
2*z*z + x*z + z*x + z + 3*x + e_1

sage: p.terms()
[3*z*z*z,
 4*x*z*z,
 4*z*x*z,
 2*x*x*z,
 4*z*z*x,
 2*z*x*x,
 2*z*z*x,
 x*x*x,
 2*z*z,
 x*z,
 z*x,
 3*x*x,
 z,
 3*x,
 e_1]

sage.quivers.algebra_elements.path_algebra_element_unpickle(P, data)

Auxiliary function for unpickling.

EXAMPLES:

sage: A = DiGraph({0:{1:['a'], 2:['b'], 1:{0:['c'], 1:['d'], 2:{0:['e'], 2:['f']}
              }}).path_semigroup().algebra(ZZ.quo(15), order='negdeglex')
sage: A.inject_variables()
Defining e_0, e_1, e_2, a, b, c, d, e, f
sage: X = a+2*b+3*c+5*e_0+3*e_2
sage: loads(dumps(X)) == X # indirect doctest
True
CHAPTER
THREE

QUIVER HOMSPACE

class sage.quivers.homspace.QuiverHomSpace(domain, codomain, category=None)
Bases: sage.categories.homset.Homset

A homomorphism of quiver representations (of one and the same quiver) is given by specifying, for each vertex of the quiver, a homomorphism of the spaces assigned to this vertex such that these homomorphisms commute with the edge maps. This class handles the set of all such maps, \( \text{Hom}_Q(M, N) \).

INPUT:

- domain – the domain of the homomorphism space
- codomain – the codomain of the homomorphism space

OUTPUT:

- QuiverHomSpace, the homomorphism space \( \text{Hom}_Q(\text{domain}, \text{codomain}) \)

Note: The quivers of the domain and codomain must be equal or a ValueError is raised.

EXAMPLES:

```
sage: Q = DiGraph({1:{2:['a', 'b']}}).path_semigroup()
sage: H = Q.S(QQ, 2).Hom(Q.P(QQ, 1))
sage: H.dimension()
2
sage: H.gens()
[Homomorphism of representations of Multi-digraph on 2 vertices, Homomorphism of representations of Multi-digraph on 2 vertices]
```

Element

alias of sage.quivers.morphism.QuiverRepHom

base_ring()

Return the base ring of the representations.

EXAMPLES:

```
sage: Q = DiGraph({1:{2:['a', 'b']}}).path_semigroup()
sage: H = Q.S(QQ, 2).Hom(Q.P(QQ, 1))
sage: H.base_ring()
Rational Field
```

codomain()

Return the codomain of the hom space.

OUTPUT:
• QuiverRep, the codomain of the Hom space

EXAMPLES:

```
sage: Q = DiGraph({1:{2:['a', 'b']}}).path_semigroup()
sage: P = Q.P(QQ, 1)
sage: H = Q.S(QQ, 2).Hom(P)
sage: H.codomain() is P
True
```

coordinates (hom)

Return the coordinates of the map when expressed in terms of the generators (i.e., the output of the \texttt{gens} method) of the hom space.

INPUT:

• \texttt{hom} – \texttt{QuiverRepHom}

OUTPUT:

• \text{list}, the coordinates of the given map when written in terms of the generators of the \texttt{QuiverHomSpace}

EXAMPLES:

```
sage: Q = DiGraph({1:{2:['a', 'b']}}).path_semigroup()
sage: S = Q.S(QQ, 2)
sage: P = Q.P(QQ, 1)
sage: H = S.Hom(P)
sage: f = S.hom({2: [[1,-1]]}, P)
sage: H.coordinates(f)
[1, -1]
```

dimension()

Return the dimension of the hom space.

OUTPUT:

• \text{integer}, the dimension

EXAMPLES:

```
sage: Q = DiGraph({1:{2:['a', 'b']}}).path_semigroup()
sage: H = Q.S(QQ, 2).Hom(Q.P(QQ, 1))
sage: H.dimension()
2
```

domain()

Return the domain of the hom space.

OUTPUT:

• \text{QuiverRep, the domain of the Hom space}

EXAMPLES:

```
sage: Q = DiGraph({1:{2:['a', 'b']}}).path_semigroup()
sage: S = Q.S(QQ, 2)
sage: H = S.Hom(Q.P(QQ, 1))
sage: H.domain() is S
True
```
**gens()**
Return a list of generators of the hom space (as a $k$-vector space).

**OUTPUT:**
- list of `QuiverRepHom` objects, the generators

**EXAMPLES:**

```python
sage: Q = DiGraph({1:{2:['a', 'b']}}).path_semigroup()
sage: H = Q.S(QQ, 2).Hom(Q.P(QQ, 1))
sage: H.gens()
[Homomorphism of representations of Multi-digraph on 2 vertices, Homomorphism of representations of Multi-digraph on 2 vertices]
```

**left_module** (basis=False)
Create the QuiverRep of `self` as a module over the opposite quiver.

**INPUT:**
- `basis` - bool. If False, then only the module is returned. If True, then a tuple is returned. The first element is the QuiverRep and the second element is a dictionary which associates to each vertex a list. The elements of this list are the homomorphisms which correspond to the basis elements of that vertex in the module.

**OUTPUT:**
- QuiverRep or tuple

**Warning:** The codomain of the Hom space must be a left module.

**Note:** The left action of a path $e$ on a map $f$ is given by $(ef)(m) = ef(m)$. This gives the Hom space its structure as a left module over the path algebra. This is then converted to a right module over the path algebra of the opposite quiver $Q.reverse()$ and returned.

**EXAMPLES:**

```python
sage: Q = DiGraph({1:{2:['a', 'b'], 3: ['c', 'd']}, 2:{3:['e']}}).path_semigroup()
sage: P = Q.P(GF(3), 3)
sage: A = Q.free_module(GF(3))
sage: H = P.Hom(A)
sage: H.dimension() 6
sage: M, basis_dict = H.left_module(True)
sage: M.dimension_vector() (4, 1, 1)
sage: Q.reverse().P(GF(3), 3).dimension_vector() (4, 1, 1)
```

As lists start indexing at 0 the $i$-th vertex corresponds to the $(i-1)$-th entry of the dimension vector:

```python
sage: len(basis_dict[2]) == M.dimension_vector()[1] True
```

**natural_map()**
The natural map from domain to codomain.
This is the zero map.

EXAMPLES:

```python
sage: Q = DiGraph({1:{2:['a', 'b']}, 2:{3:['c']}}).path_semigroup()
sage: spaces = {1: QQ^2, 2: QQ^2, 3: QQ^1}
sage: maps = {(1, 2, 'a'): [[[1, 0]], [[0, 0]]], (1, 2, 'b'): [[[0, 0]], [[0, 1]]],
            (2, 3, 'c'): [[[1]], [[1]]]}
sage: M = Q.representation(QQ, spaces, maps)
sage: spaces2 = {2: QQ^1, 3: QQ^1}
sage: S = Q.representation(QQ, spaces2)
sage: S.hom(M)  # indirect doctest
Homomorphism of representations of Multi-digraph on 3 vertices
sage: S.hom(M) == S.Hom(M).natural_map()
True
```

`quiver()`

Return the quiver of the representations.

OUTPUT:

- `DiGraph`, the quiver of the representations

EXAMPLES:

```python
sage: P = DiGraph({1:{2:['a', 'b']}}).path_semigroup()
sage: H = P.S(QQ, 2).Hom(P.P(QQ, 1))
sage: H.quiver()  # P.quiver()
True
```

`zero()`

Return the zero morphism.

**Note:** It is needed to override the method inherited from the category of modules, because it would create a morphism that is of the wrong type and does not comply with `QuiverRepHom`.

EXAMPLES:

```python
sage: Q = DiGraph({1:{2:['a', 'b']}}).path_semigroup()
sage: H = Q.S(QQ, 2).Hom(Q.P(QQ, 1))
sage: H.zero() + H.an_element() == H.an_element()
True
sage: isinstance(H.zero(), H.element_class)
True
```
class sage.quivers.morphism.QuiverRepHom(domain, codomain, data={})

Bases: sage.categories.morphism.CallMorphism

A homomorphism of quiver representations (of one and the same quiver) is given by specifying, for each vertex
of the quiver, a homomorphism of the spaces assigned to this vertex such that these homomorphisms commute
with the edge maps. The domain and codomain of the homomorphism are required to be representations of the
same quiver over the same base ring.

INPUT:

• domain – QuiverRep, the domain of the homomorphism
• codomain – QuiverRep, the codomain of the homomorphism
• data - dict, list, or QuiverRepElement (default: empty dict), with the following meaning:
  – list: data can be a list of images for the generators of the domain. “Generators” means the output of
    the gens() method. An error will be generated if the map so defined is not equivariant with respect
to the action of the quiver.
  – dictionary: data can be a dictionary associating to each vertex of the quiver either a homomorphism
    with domain and codomain the spaces associated to this vertex in the domain and codomain modules
    respectively, or a matrix defining such a homomorphism, or an object that sage can construct such a
    matrix from. Not all vertices must be specified, unspecified vertices are assigned the zero map, and
    keys not corresponding to vertices of the quiver are ignored. An error will be generated if these maps
do not commute with the edge maps of the domain and codomain.
  – QuiverRepElement: if the domain is a QuiverRep_with_path_basis then data can be a
    single QuiverRepElement belonging to the codomain. The map is then defined by sending each
    path, p, in the basis to data*p. If data is not an element of the codomain or the domain is not a
    QuiverRep_with_path_basis then an error will be generated.
  – QuiverRepHom: the input can also be a map f : D → C such that there is a coercion from the
domain of self to D and from C to the codomain of self. The composition of these maps is the
result.

OUTPUT:

• QuiverRepHom

EXAMPLES:

```
sage: Q = DiGraph({1:{2:['a', 'b']}, 2:{3:['c']}}).path_semigroup()
sage: spaces = {1: QQ^2, 2: QQ^2, 3:QQ^1}
sage: maps = {(1, 2, 'a'): [[1, 0], [0, 0]], (1, 2, 'b'): [[0, 0], [0, 1]], (2, 3, ˓→ 'c'): [[1], [1]]}
sage: M = Q.representation(QQ, spaces, maps)
```

(continues on next page)
With no additional data this creates the zero map:

```
sage: f = S.hom(M)
sage: f.is_zero()
True
```

We must specify maps at the vertices to get a nonzero homomorphism. Note that if the dimensions of the spaces assigned to the domain and codomain of a vertex are equal then Sage will construct the identity matrix from 1:

```
sage: maps2 = {2:[1, -1], 3:1}
sage: g = S.hom(maps2, M)
```

Here we create the same map by specifying images for the generators:

```
sage: x = M({2: (1, -1)})
sage: y = M({3: (1,)})
sage: h = S.hom([x, y], M)
sage: g == h
True
```

If the domain is a module of type QuiverRep_with_path_basis (for example, the indecomposable projectives) we can create maps by specifying a single image:

```
sage: Proj = Q.P(GF(7), 3)
sage: Simp = Q.S(GF(7), 3)
sage: im = Simp({3: (1,)})
sage: Proj.hom(im, Simp).is_surjective()
True
```

## algebraic_dual

```
algebraic_dual()
Compute the algebraic dual \( f^t : N^t \to M^t \) of self = \( f : M \to N \) where \((-)^t = \text{Hom}_Q(-, kQ)\).

OUTPUT:
• QuiverRepHom, the map \( f^t : N^t \to M^t \)

Note: If \( e \) is an edge of the quiver \( Q \) and \( g \) is an element of \( \text{Hom}_Q(N, kQ) \) then we let \((ge)(m) = eg(m)\). This gives \( \text{Hom}_Q(N, kQ) \) its structure as a module over the opposite quiver \( Q.reverse() \). The map \( \text{Hom}_Q(N, kQ) \to \text{Hom}_Q(M, kQ) \) returned sends \( g \) to \( gf \).

EXAMPLES:
```
sage: Q = DiGraph({1:{2:['a'], 3:['b','c','d']}, 2:{4:['e','f']}, 3:{4:['g']}, ˓→ 5:{2:['h','i']}}).path_semigroup()
sage: P = Q.P(QQ, 4)
sage: P.algebraic_dual()
Representation with dimension vector (5, 2, 1, 1, 4)
```

The algebraic dual of an indecomposable projective is the indecomposable projective of the same vertex in the opposite quiver.

```
sage: Q.reverse().P(QQ, 4) Representation with dimension vector (5, 2, 1, 1, 4)
```
base_ring()
Return the base ring of the representation in the codomain.
OUTPUT:
  • ring, the base ring of the codomain
EXAMPLES:

```sage
Q = DiGraph({1:{2:['a', 'b']}, 2:{3:['c']}}).path_semigroup()
P = Q.P(QQ, 1)
f = P.hom({1: 1, 2: 1, 3: 1}, P)
f.base_ring() is QQ
True
```

codomain()
Return the codomain of the homomorphism.
OUTPUT:
  • QuiverRep, the codomain
EXAMPLES:

```sage
Q = DiGraph({1:{2:['a', 'b']}, 2:{3:['c']}}).path_semigroup()
spaces = {1: QQ^2, 2: QQ^2, 3: QQ^1}
maps = {(1, 2, 'a'): [[1, 0], [0, 0]], (1, 2, 'b'): [[0, 0], [0, 1]], ...
        (2, 3, 'c'): [[1], [1]]}
M = Q.representation(QQ, spaces, maps)
S = Q.representation(QQ)
g = S.hom(M)
g.codomain() is M
True
```

cokernel()
Return the cokernel of self.
OUTPUT:
  • QuiverRep, the cokernel

Note: To get the factor map of the codomain, D, onto the cokernel, C, use C.coerce_map_from(D).

EXAMPLES:

```sage
Q = DiGraph({1:{2:['a', 'b']}, 2:{3:['c']}}).path_semigroup()
spaces = {1: QQ^2, 2: QQ^2, 3: QQ^1}
maps = {(1, 2, 'a'): [[1, 0], [0, 0]], (1, 2, 'b'): [[0, 0], [0, 1]], ...
        (2, 3, 'c'): [[1], [1]]}
M = Q.representation(QQ, spaces, maps)
spaces2 = {2: QQ^2, 3: QQ^1}
N = Q.representation(QQ, spaces2, {(2, 3, 'c'): [[1], [0]])
maps2 = {2:[[1, 0], [0, 0]], 3:1}
g = N.hom(maps2, M)
g.cokernel().dimension_vector()
(2, 1, 0)
```

direct_sum(maps, return_maps=False, pinch=None)
Return the direct sum of self with the maps in the list maps.
INPUT:

• maps – `QuiverRepHom` or list of `QuiverRepHom`'s
• return_maps – bool (default: False). If False, then the return value is a `QuiverRepHom` which is the direct sum of self with the `QuiverRepHoms` in maps. If True, then the return value is a tuple of length either 3 or 5. The first entry of the tuple is the `QuiverRepHom` giving the direct sum. If pinch is either None or 'codomain' then the next two entries in the tuple are lists giving respectively the inclusion and the projection maps for the factors of the direct sum. Summands are ordered as given in maps with self as the zeroth summand. If pinch is either None or 'domain' then the next two entries in the tuple are the inclusion and projection maps for the codomain. Thus if pinch is None then the tuple will have length 5. If pinch is either 'domain' or 'codomain' then the tuple will have length 3.
• pinch – string or None (default: None). If this is equal to 'domain', then the domains of self and the given maps must be equal. The direct sum of \( f: A \to B \) and \( g: A \to C \) returned is then the map \( A \to B \oplus C \) defined by sending \( x \) to \( (f(x), g(x)) \). If pinch equals 'codomain', then the codomains of self and the given maps must be equal. The direct sum of \( f: A \to C \) and \( g: B \to C \) returned is then the map \( A \oplus B \to C \) defined by sending \( (x, y) \) to \( f(x) + g(y) \). Finally, if pinch is anything other than 'domain' or 'codomain', then the direct sum of \( f: A \to B \) and \( g: C \to D \) returned is the map \( A \oplus C \to B \oplus D \) defined by sending \( (x, y) \) to \( (f(x), g(y)) \).

OUTPUT:

• `QuiverRepHom` or tuple

EXAMPLES:

```python
sage: Q = DiGraph({1:{2:['a', 'b']}}).path_semigroup()
sage: P1 = Q.P(GF(3), 1)
sage: P2 = Q.P(GF(3), 2)
sage: S1 = P1/P1.radical()
sage: S2 = P2/P2.radical()
sage: S1.coerce_map_from(P1)
sage: S2.coerce_map_from(P2)
sage: f = S1.direct_sum(S2)
sage: f.domain().dimension_vector() == Q.free_module(GF(3)).dimension_vector()
True
sage: f.is_surjective()
True
sage: id = P1.Hom(P1).identity()
sage: g = id.direct_sum(id, pinch='domain')
sage: g.is_surjective()
False
```

`domain()`

Return the domain of the homomorphism.

OUTPUT:

• `QuiverRep`, the domain

EXAMPLES:

```python
sage: Q = DiGraph({1:{2:['a', 'b']}, 2:{3:['c']}}).path_semigroup()
sage: spaces = {1: QQ^2, 2: QQ^2, 3:QQ^1}
sage: maps = {(1, 2, 'a'): [[1], [0]], (1, 2, 'b'): [[0], [1]],
(2, 3, 'c'): [[1], [1]]}
sage: M = Q.representation(QQ, spaces, maps)
sage: S = Q.representation(QQ)
```

(continues on next page)
sage: g = M.hom(S)
sage: g.domain() is M
True

get_map(vertex)
Return the homomorphism at the given vertex vertex.

INPUT:
• vertex – integer, a vertex of the quiver

OUTPUT:
• homomorphism, the homomorphism associated to the given vertex

EXAMPLES:

```
sage: Q = DiGraph({1:{2:['a', 'b']}, 2:{3:['c']}}).path_semigroup()
sage: P = Q.P(QQ, 1)
sage: S = P/P.radical()
sage: f = S.coerce_map_from(P)
sage: f.get_map(1).is_bijective()
True
```

get_matrix(vertex)
Return the matrix of the homomorphism attached to vertex vertex.

INPUT:
• vertex – integer, a vertex of the quiver

OUTPUT:
• matrix, the matrix representing the homomorphism associated to the given vertex

EXAMPLES:

```
sage: Q = DiGraph({1:{2:['a', 'b']}, 2:{3:['c']}}).path_semigroup()
sage: I = Q.I(QQ, 3)
sage: M = I/I.radical()
sage: f = M.coerce_map_from(I)
sage: f.get_matrix(1)
[1 0]
[0 1]
```

image()
Return the image of self.

OUTPUT:
• QuiverRep, the image

Note: To get the inclusion map of the image, I, into the codomain, C, use C.coerce_map_from(I).

EXAMPLES:

```
sage: Q = DiGraph({1:{2:['a', 'b']}, 2:{3:['c']}}).path_semigroup()
sage: spaces = {1: QQ^2, 2: QQ^2, 3:QQ^1}
sage: maps = {(1, 2, 'a'): [[1, 0], [0, 0]], (1, 2, 'b'): [[0, 0], [0, 1]],
          (2, 3, 'c'): [[1], [1]]}
```
is_endomorphism()  
Test whether the homomorphism is an endomorphism.  

OUTPUT:  

• bool, True if the domain equals the codomain, False otherwise  

EXAMPLES:  

```python  
sage: Q = DiGraph({1:{2:['a', 'b']}, 2:{3:['c']}}).path_semigroup()  
sage: P = Q.P(QQ, 1)  
sage: f = P.hom({1: 1, 2: 1, 3: 1}, P)  
sage: f.is_endomorphism()  
True  
sage: S = P/P.radical()  
sage: g = S.coerce_map_from(P)  
sage: g.is_endomorphism()  
False  
```

is_injective()  
Test whether the homomorphism is injective.  

OUTPUT:  

• bool, True if the homomorphism is injective, False otherwise  

EXAMPLES:  

```python  
sage: Q = DiGraph({1:{2:['a', 'b']}, 2:{3:['c']}}).path_semigroup()  
sage: P = Q.P(QQ, 1)  
sage: f = P.hom({1: 1, 2: 1, 3: 1}, P)  
sage: f.is_injective()  
True  
sage: g = P.hom(P)  
sage: g.is_injective()  
False  
```

is_isomorphism()  
Test whether the homomorphism is an isomorphism.  

OUTPUT:  

• bool, True if the homomorphism is bijective, False otherwise  

EXAMPLES:  

```python  
sage: Q = DiGraph({1:{2:['a', 'b']}, 2:{3:['c']}}).path_semigroup()  
sage: P = Q.P(QQ, 1)  
sage: f = P.hom({1: 1, 2: 1, 3: 1}, P)  
sage: f.is_isomorphism()  
True  
```
sage: g = P.hom(P)
sage: g.is_isomorphism()
False

is_surjective()
Test whether the homomorphism is surjective.

OUTPUT:
• bool, True if the homomorphism is surjective, False otherwise

EXAMPLES:

```python
sage: Q = DiGraph({1:{2:['a', 'b']}, 2:{3:['c']}}).path_semigroup()
sage: P = Q.P(QQ, 1)
sage: f = P.hom({1: 1, 2: 1, 3: 1}, P)
sage: f.is_surjective()
True
sage: g = P.hom(P)
sage: g.is_surjective()
False
```

is_zero()
Test whether the homomorphism is the zero homomorphism.

OUTPUT:
• bool, True if the homomorphism is zero, False otherwise

EXAMPLES:

```python
sage: Q = DiGraph({1:{2:['a','b']}}).path_semigroup()
sage: P = Q.P(QQ, 1)
sage: f = P.hom({1: 1, 2: 1, 3: 1}, P)
sage: f.is_zero()
False
sage: g = P.hom(P)
sage: g.is_zero()
True
```

iscalar_mult(scalar)
Multiply self by scalar in place.

EXAMPLES:

```python
sage: Q = DiGraph({1:{2:['a','b']}}).path_semigroup()
sage: M = Q.P(QQ, 1)
sage: f = M.Hom(M).an_element()
sage: x = M.an_element()
sage: y = f(x)
sage: f.iscalar_mult(6)
sage: f(x) == 6*y
True
```

kernel()
Return the kernel of self.

OUTPUT:
• QuiverRep, the kernel
Note: To get the inclusion map of the kernel, \( K \), into the domain, \( D \), use \( \text{D.coerce_map_from}(K) \).

**EXAMPLES:**

```python
sage: Q = DiGraph({1:{2:['a', 'b']}, 2:{3:['c']}}).path_semigroup()
sage: spaces = {1: QQ^2, 2: QQ^2, 3:QQ^1}
sage: maps = {(1, 2, 'a'): [[1, 0], [0, 0]], (1, 2, 'b'): [[0, 0], [0, 1]], (2, 3, 'c'): [[1], [1]]}
sage: M = Q.representation(QQ, spaces, maps)
sage: spaces2 = {2: QQ^2, 3: QQ^1}
sage: N = Q.representation(QQ, spaces2, {(2, 3, 'c'): [[1], [0]])}
sage: maps2 = {2:[[1, 0], [0, 0]], 3:1}
sage: g = N.hom(maps2, M)
sage: g.kernel().dimension_vector()
(0, 1, 0)
```

**lift**(\( x \))
Given an element \( x \) of the image, return an element of the domain that maps onto it under \( \text{self} \).

**INPUT:**

- \( x \) – QuiverRepElement

**OUTPUT:**

- QuiverRepElement

**EXAMPLES:**

```python
sage: Q = DiGraph({1:{2:['a','b']}, 2:{3:['c','d']}}).path_semigroup()
sage: P = Q.P(RR, 3)
sage: S = P/P.radical()
sage: proj = S.coerce_map_from(P)
sage: x = S.an_element()
sage: y = proj.lift(x)
sage: proj(y) == x
True
```

**linear_dual()**

Compute the linear dual \( Df : DN \rightarrow DM \) of \( \text{self} = f : M \rightarrow N \) where \( D(-) = \text{Hom}_k(-, k) \).

**OUTPUT:**

- \( \text{QuiverRepHom} \), the map \( Df : DN \rightarrow DM \)

**Note:** If \( e \) is an edge of the quiver \( Q \) and \( g \) is an element of \( \text{Hom}_k(N, k) \) then we let \((ge)(m) = g(ema)\). This gives \( \text{Hom}_k(N, k) \) its structure as a module over the opposite quiver \( Q.\text{reverse()} \). The map \( \text{Hom}_k(N, k) \rightarrow \text{Hom}_k(M, k) \) returned sends \( g \) to \( gf \).

**EXAMPLES:**
sage: Q = DiGraph({1:{2:['a', 'b'], 2:{3:['c']}}}).path_semigroup()
sage: P = Q.P(QQ, 1)
sage: S = P/P.radical()
sage: f = S.coerce_map_from(P)

The dual of a surjective map is injective and vice versa:

sage: f.is_surjective()
True
sage: g = f.linear_dual()
sage: g.is_injective()
True

The dual of a right module is a left module for the same quiver, Sage represents this as a right module for the opposite quiver:

sage: g.quiver().path_semigroup() is Q.reverse()
True

The double dual of a map is the original representation:

sage: g.linear_dual() == f
True

quiver()

Return the quiver of the representations in the domain/codomain.

OUTPUT:

- DiGraph, the quiver of the representations in the domain and codomain

EXAMPLES:

sage: Q = DiGraph({1:{2:['a', 'b'], 2:{3:['c']}}}).path_semigroup()
sage: P = Q.P(QQ, 1)
sage: f = P.hom({1: 1, 2: 1, 3: 1}, P)
sage: f.quiver() is Q.quiver()
True

rank()

Return the rank of the homomorphism self (as a \(k\)-linear map).

OUTPUT:

- integer, the rank

EXAMPLES:

sage: Q = DiGraph({1:{2:['a', 'b'], 2:{3:['c']}}}).path_semigroup()
sage: P = Q.P(QQ, 1)
sage: S = P/P.radical()
sage: f = S.coerce_map_from(P)
sage: assert(f.rank() == 1)

scalar_mult(scalar)

Return the result of the scalar multiplication scalar * self, where scalar is an element of the base ring \(k\).

EXAMPLES:
sage: Q = DiGraph({1:{2:['a','b']}}).path_semigroup()
sage: M = Q.P(QQ, 1)
sage: f = M.Hom(M).an_element()
sage: x = M.an_element()
sage: g = f.scalar_mult(6)
sage: g(x) == 6*f(x)
True
class sage.quivers.path_semigroup.PathSemigroup(Q)
Bases: sage.structure.unique_representation.UniqueRepresentation, sage.structure.parent.Parent

The partial semigroup that is given by the directed paths of a quiver, subject to concatenation.

See representation for a definition of this semigroup and of the notion of a path in a quiver.

Note that a partial semigroup here is defined as a set $G$ with a partial binary operation $G \times G \to G \cup \{\text{None}\}$, which is written infix as a $*$ sign and satisfies associativity in the following sense: If $a, b$ and $c$ are three elements of $G$, and if one of the products $(a * b) * c$ and $a * (b * c)$ exists, then so does the other and the two products are equal. A partial semigroup is not required to have a neutral element (and this one usually has no such element).

EXAMPLES:

```python
sage: Q = DiGraph({1:{2:['a','b'], 3:['c']}, 2:{3:['d']}})
sage: S = Q.path_semigroup()
sage: S
Partial semigroup formed by the directed paths of Multi-digraph on 3 vertices
sage: S.variable_names()
('e_1', 'e_2', 'e_3', 'a', 'b', 'c', 'd')
sage: S.gens()
(e_1, e_2, e_3, a, b, c, d)
sage: S.category()
Category of finite enumerated semigroups
```

In the test suite, we skip the associativity test, as in this example the paths used for testing can’t be concatenated:

```python
sage: TestSuite(S).run(skip=['_test_associativity'])
```

If there is only a single vertex, the partial semigroup is a monoid. If the underlying quiver has cycles or loops, then the (partial) semigroup only is an infinite enumerated set. This time, there is no need to skip tests:

```python
sage: Q = DiGraph({1:{1:['a', 'b', 'c', 'd']}})
sage: M = Q.path_semigroup()
sage: M
Monoid formed by the directed paths of Looped multi-digraph on 1 vertex
sage: M.category()
Category of infinite enumerated monoids
sage: TestSuite(M).run()
```

Element

alias of sage.quivers.paths.QuiverPath

$I(k, \text{vertex})$

Return the indecomposable injective module over $k$ at the given vertex $\text{vertex}$.  

29
This module is literally indecomposable only when $k$ is a field.

**INPUT:**
- $k$ – ring, the base ring of the representation
- vertex – integer, a vertex of the quiver

**OUTPUT:**
- QuiverRep, the indecomposable injective module at vertex vertex with base ring $k$

**EXAMPLES:**

```sage
sage: Q = DiGraph({1:{2:['a','b']}, 2:{3:['c','d']}}).path_semigroup()
sage: I2 = Q.I(GF(3), 2)
sage: I2.dimension_vector() (4, 2, 1)
sage: Q.I(ZZ, 3).dimension_vector() (0, 1, 1)
sage: Q.I(ZZ, 1).dimension_vector() (1, 0, 0)
```

The vertex given must be a vertex of the quiver:

```sage
sage: Q.I(QQ, 4)
Traceback (most recent call last):
  ... ValueError: must specify a valid vertex of the quiver
```

**$P(k, \text{vertex})$**

Return the indecomposable projective module over $k$ at the given vertex vertex.

This module is literally indecomposable only when $k$ is a field.

**INPUT:**
- $k$ – ring, the base ring of the representation
- vertex – integer, a vertex of the quiver

**OUTPUT:**
- QuiverRep, the indecomposable projective module at vertex vertex with base ring $k$

**EXAMPLES:**

```sage
sage: Q = DiGraph({1:{2:['a','b']}, 2:{3:['c','d']}}).path_semigroup()
sage: P2 = Q.P(GF(3), 2)
sage: P2.dimension_vector() (0, 1, 1)
sage: Q.P(ZZ, 3).dimension_vector() (1, 0, 0)
sage: Q.P(ZZ, 1).dimension_vector() (1, 2, 4)
```

The vertex given must be a vertex of the quiver:

```sage
sage: Q.P(QQ, 4)
Traceback (most recent call last):
  ... ValueError: must specify a valid vertex of the quiver
```

**$S(k, \text{vertex})$**

Return the simple module over $k$ at the given vertex vertex.

This module is literally simple only when $k$ is a field.
INPUT:

- $k$ – ring, the base ring of the representation
- vertex – integer, a vertex of the quiver

OUTPUT:

- QuiverRep, the simple module at vertex with base ring $k$

EXAMPLES:

```python
sage: P = DiGraph({1:{2:['a','b']}, 2:{3:['c','d']}}).path_semigroup()
sage: S1 = P.S(GF(3), 1)
sage: P.S(ZZ, 3).dimension_vector()
(0, 0, 1)
sage: P.S(ZZ, 1).dimension_vector()
(1, 0, 0)
```

The vertex given must be a vertex of the quiver:

```python
sage: P.S(QQ, 4)
Traceback (most recent call last):
  ...
ValueError: must specify a valid vertex of the quiver
```

```python
algebra ($k$, order='negdegrevlex')
```

Return the path algebra of the underlying quiver.

INPUT:

- $k$ – a commutative ring
- order – optional string, one of “negdegrevlex” (default), “degrevlex”, “negdeglex” or “deglex”, defining the monomial order to be used.

NOTE:

Monomial orders that are not degree orders are not supported.

EXAMPLES:

```python
sage: Q = DiGraph({1:{2:['a','b']}, 2:{3:['d']}, 3:{1:['c']}})
sage: P = Q.path_semigroup()
sage: P.algebra(GF(3))
Path algebra of Multi-digraph on 3 vertices over Finite Field of size 3
```

Now some example with different monomial orderings:

```python
sage: P1 = DiGraph({1:{1:['x','y','z']}}).path_semigroup().algebra(GF(25,'t'))
sage: P2 = DiGraph({1:{1:['x','y','z']}}).path_semigroup().algebra(GF(25,'t'), → order='degrevlex')
sage: P3 = DiGraph({1:{1:['x','y','z']}}).path_semigroup().algebra(GF(25,'t'), → order='negdeglex')
sage: P4 = DiGraph({1:{1:['x','y','z']}}).path_semigroup().algebra(GF(25,'t'), → order='deglex')
sage: P1.order_string()
'negdegrevlex'
sage: sage_eval('(x+2*z+1)^3', P1.gens_dict())
e_1 + z + 3*x + 2*z*z + z*x + 3*x*x + 3*z*z*z + 4*x*z*z + 4*z*x*z + 2*x*x*z + 4*z*z*x + 2*x*z*x + 2*z*x*x + x*x*x
sage: sage_eval('(x+2*z+1)^3', P2.gens_dict())
(continues on next page)
\[3z^3 + 4zx^2z + 2xxz^2 + 4zx^2x + 2z^2xx + xxz^2 + \text{e}_1\]

\[= 2z^2 + xz + zx + 3xx + z + 3x + \text{e}_1\]

\[\text{sage: sage} \text{eval}('(x+2z+1)^3', P3.gens_dict())\]
\[e_1 + z + 3x + 2z^2 + zx + xz + 3xx + 4z^2 + 4zx + 4z + \text{e}_1\]
\[= 2z^2 + xz + 3xx + 2xx + z + xx + \text{e}_1\]

\[\text{sage: sage} \text{eval}('(x+2z+1)^3', P4.gens_dict())\]
\[3z^3 + 4zx^2z + 2xxz^2 + 4zx^2x + 2z^2xx + xxz^2 + \text{e}_1\]
\[= 2z^2 + xz + zx + 3xx + z + 3x + \text{e}_1\]

\section*{all\_paths \(\text{start}=\text{None}, \text{end}=\text{None}\)}

List of all paths between a pair of vertices \((\text{start}, \text{end})\).

\textbf{INPUT:}

\begin{itemize}
  \item \texttt{start} – integer or \texttt{None} (default: \texttt{None}); the initial vertex of the paths in the output; if \texttt{None} is given then the initial vertex is arbitrary.
  \item \texttt{end} – integer or \texttt{None} (default: \texttt{None}); the terminal vertex of the paths in the output; if \texttt{None} is given then the terminal vertex is arbitrary
\end{itemize}

\textbf{OUTPUT:}

\begin{itemize}
  \item list of paths, excluding the invalid path
\end{itemize}

\textbf{Todo:} This currently does not work for quivers with cycles, even if there are only finitely many paths from \texttt{start} to \texttt{end}.

\textbf{Note:} If there are multiple edges between two vertices, the method \texttt{sage.graphs.digraph.all\_paths()} will not differentiate between them. But this method, which is not for digraphs but for their path semigroup associated with them, will.

\textbf{EXAMPLES:}

\begin{verbatim}
\text{sage: Q = DiGraph(({1:{2:['a','b'], 3:['c']}, 2:{3:['d']}}))
\text{sage: F = Q.path_semigroup()}
\text{sage: F.all_paths(1, 3)}
[a*d, b*d, c]
\end{verbatim}

If \texttt{start=\text{end}} then we expect only the trivial path at that vertex:

\begin{verbatim}
\text{sage: F.all_paths(1, 1)}
[e_1]
\end{verbatim}

The empty list is returned if there are no paths between the given vertices:

\begin{verbatim}
\text{sage: F.all_paths(3, 1)}
[]
\end{verbatim}

If \texttt{end=\text{None}} then all edge paths beginning at \texttt{start} are returned, including the trivial path:

\begin{verbatim}
\text{sage: F.all_paths(3)}
[e_2, d]
\end{verbatim}

If \texttt{start=\text{None}} then all edge paths ending at \texttt{end} are returned, including the trivial path. Note that the two edges from vertex 1 to vertex 2 count as two different edge paths:
sage: F.all_paths(None, 2)
[a, b, e_2]
sage: F.all_paths(end=2)
[a, b, e_2]

If `start=end=None` then all edge paths are returned, including trivial paths:

sage: F.all_paths()
[e_1, a, b, a*d, b*d, c, e_2, d, e_3]

The vertex given must be a vertex of the quiver:

sage: F.all_paths(1, 4)
Traceback (most recent call last):
...
ValueError: the end vertex 4 is not a vertex of the quiver

If the underlying quiver is cyclic, a `ValueError` is raised:

sage: Q = DiGraph({1:{2:['a','b'], 3:['c']}, 3:{1:['d']}})
sage: F = Q.path_semigroup()
sage: F.all_paths()
Traceback (most recent call last):
...
ValueError: the underlying quiver has cycles, thus, there may be an infinity
→ of directed paths

**arrows()**

Return the elements corresponding to edges of the underlying quiver.

EXAMPLES:

sage: P = DiGraph({1:{2:['a','b'], 3:['c']}, 2:{3:['d']}}).path_semigroup()
sage: P.arrows()
(a, b, c, d)

**cardinality()**

EXAMPLES:

sage: Q = DiGraph({1:{2:['a','b'], 3:['c']}, 3:{1:['d']}})
sage: F = Q.path_semigroup()
sage: F.cardinality()
9
sage: A = F.algebra(QQ)
sage: list(A.basis())
[e_1, e_2, e_3, a, b, c, d, a*d, b*d]
sage: Q = DiGraph({1:{2:['a','b'], 3:['c']}, 3:{1:['d']}})
sage: F = Q.path_semigroup()
sage: F.cardinality()
+Infinity
sage: A = F.algebra(QQ)
sage: list(A.basis())
Traceback (most recent call last):
...
ValueError: the underlying quiver has cycles, thus, there may be an infinity
→ of directed paths
free_module$(k)$
Return a free module of rank 1 over $kP$, where $P$ is `self`. (In other words, the regular representation.)

**INPUT:**
- $k$ – ring, the base ring of the representation.

**OUTPUT:**
- `QuiverRep_with_path_basis`, the path algebra considered as a right module over itself.

**EXAMPLES:**
```
sage: Q = DiGraph({1:{2:['a', 'b'], 3: ['c', 'd']}, 2:{3:['e']}}).path_semigroup()
sage: Q.free_module(GF(3)).dimension_vector()
(1, 3, 6)
```

`gen`(i)
Return generator number $i$.

**INPUT:**
- $i$ – integer

**OUTPUT:**
An idempotent, if $i$ is smaller than the number of vertices, or an arrow otherwise.

**EXAMPLES:**
```
sage: P = DiGraph({1:{2:['a','b'], 3:['c']}, 3:{1:['d']}}).path_semigroup()
sage: P.1  # indirect doctest
e_2
e_2
sage: P.idempotents()[1]
e_2
sage: P.5
c
sage: P.gens()[5]
c
```

gens()
Return the tuple of generators.

**Note:** This coincides with the sum of the output of `idempotents()` and `arrows()`.

**EXAMPLES:**
```
sage: P = DiGraph({1:{2:['a','b'], 3:['c']}, 3:{1:['d']}}).path_semigroup()
sage: P.gens()
(e_1, e_2, e_3, a, b, c, d)
sage: P.gens() == P.idempotents() + P.arrows()
True
```

`idempotents()`
Return the idempotents corresponding to the vertices of the underlying quiver.

**EXAMPLES:**
```python
sage: P = DiGraph({1:{2:['a','b'], 3:['c']}, 3:{1:['d']}}).path_semigroup()
sage: P.idempotents()
(e_1, e_2, e_3)
```

**injective**\((k, \text{vertex})\)

Return the indecomposable injective module over \(k\) at the given vertex \(\text{vertex}\).

This module is literally indecomposable only when \(k\) is a field.

**INPUT:**

- \(k\) – ring, the base ring of the representation
- \(\text{vertex}\) – integer, a vertex of the quiver

**OUTPUT:**

- QuiverRep, the indecomposable injective module at vertex \(\text{vertex}\) with base ring \(k\)

**EXAMPLES:**

```python
sage: Q = DiGraph({1:{2:['a','b']}, 2:{3:['c','d']}}).path_semigroup()
sage: I2 = Q.I(GF(3), 2)
sage: Q.I(ZZ, 3).dimension_vector()
(4, 2, 1)
sage: Q.I(ZZ, 1).dimension_vector()
(1, 0, 0)
```

The vertex given must be a vertex of the quiver:

```python
sage: Q.I(QQ, 4)
Traceback (most recent call last):
... ValueError: must specify a valid vertex of the quiver
```

**is_finite()**

This partial semigroup is finite if and only if the underlying quiver is acyclic.

**EXAMPLES:**

```python
sage: Q = DiGraph({1:{2:['a','b'], 3:['c']}, 2:{3:['d']}})
sage: Q.path_semigroup().is_finite()
True
sage: Q = DiGraph({1:{2:['a','b'], 3:['c']}, 3:{1:['d']}})
sage: Q.path_semigroup().is_finite()
False
```

**iter_paths_by_length_and_endpoint**\((d, v)\)

An iterator over quiver paths with a fixed length and end point.

**INPUT:**

- \(d\) – an integer, the path length
- \(v\) – a vertex, end point of the paths

**EXAMPLES:**

```python
sage: Q = DiGraph({1:{2:['a','b'], 3:['c']}, 2:{3:['d']}})
sage: F = Q.path_semigroup()
sage: F.is_finite()
```

(continues on next page)
False

```python
sage: list(F.iter_paths_by_length_and_endpoint(4,1))
[c*a*d*c, c*b*d*c]
```

```python
sage: list(F.iter_paths_by_length_and_endpoint(5,1))
[d*c*a*d*c, d*c*b*d*c]
```

```python
sage: list(F.iter_paths_by_length_and_endpoint(5,2))
[c*a*d*c*a, c*b*d*c*a, c*a*d*c*b, c*b*d*c*b]
```

**iter_paths_by_length_and_startpoint** \( (d, v) \)

An iterator over quiver paths with a fixed length and start point.

**INPUT:**

- \( d \) – an integer, the path length
- \( v \) – a vertex, start point of the paths

**EXAMPLES:**

```python
sage: Q = DiGraph({1:{2:['a','b']}, 2:{3:['d']}, 3:{1:['c']}})
sage: P = Q.path_semigroup()
sage: P.is_finite()
False
sage: list(P.iter_paths_by_length_and_startpoint(4,1))
[a*d*c*a, a*d*c*b, b*d*c*a, b*d*c*b]
```

```python
sage: list(P.iter_paths_by_length_and_startpoint(5,1))
[a*d*c*a*d, a*d*c*b*d, b*d*c*a*d, b*d*c*b*d]
```

```python
sage: list(P.iter_paths_by_length_and_startpoint(5,2))
[d*c*a*d*c, d*c*b*d*c]
```

**ngens**

Return the number of generators (**arrows()** and **idempotents()**).

**EXAMPLES:**

```python
sage: F = DiGraph({1:{2:['a','b']}, 2:{3:['c']}, 3:{1:['d']}}).path_semigroup()
sage: F.ngens()
7
```

**projective** \( (k, \text{vertex}) \)

Return the indecomposable projective module over \( k \) at the given vertex \( \text{vertex} \).

This module is literally indecomposable only when \( k \) is a field.

**INPUT:**

- \( k \) – ring, the base ring of the representation
- \( \text{vertex} \) – integer, a vertex of the quiver

**OUTPUT:**

- **QuiverRep**, the indecomposable projective module at \( \text{vertex} \) with base ring \( k \)

**EXAMPLES:**

```python
sage: Q = DiGraph({1:{2:['a','b']}, 2:{3:['c','d']}}).path_semigroup()
sage: P2 = Q.P(GF(3), 2)
sage: Q.P(ZZ, 3).dimension_vector()
(0, 0, 1)
```
The vertex given must be a vertex of the quiver:

```python
sage: Q.P(QQ, 4)
Traceback (most recent call last):
  ... ValueError: must specify a valid vertex of the quiver
```

### quiver()

Return the underlying quiver (i.e., digraph) of this path semigroup.

**Note:** The returned digraph always is an immutable copy of the originally given digraph that is made weighted.

**EXAMPLES:**

```python
sage: Q = DiGraph({1:{2:['a','b']}, 2:{3:['d']}, 3:{1:['c']}}, weighted=False)
sage: F = Q.path_semigroup()
sage: F.quiver() == Q
False
sage: Q.weighted(True)
sage: F.quiver() == Q
True
```

### representation(k, *args, **kwds)

Return a representation of the quiver. For more information see the QuiverRep documentation.

### reverse()

The path semigroup of the reverse quiver.

**EXAMPLES:**

```python
sage: Q = DiGraph({1:{2:['a','b']}, 2:{3:['d']}, 3:{1:['c']}})
sage: F = Q.path_semigroup()
sage: F.reverse() is Q.reverse().path_semigroup()
True
```

### simple(k, vertex)

Return the simple module over $k$ at the given vertex vertex. This module is literally simple only when $k$ is a field.

**INPUT:**

- $k$ – ring, the base ring of the representation
- vertex – integer, a vertex of the quiver

**OUTPUT:**

- QuiverRep, the simple module at vertex with base ring $k$

**EXAMPLES:**
The vertex given must be a vertex of the quiver:
sage.quivers.paths.NewQuiverPath(Q, start, end, biseq_data)

Return a new quiver path for given defining data.

INPUT:

- Q, the path semigroup of a quiver
- start, an integer, the label of the startpoint
- end, an integer, the label of the endpoint
- biseq_data, a tuple formed by
  - A string, encoding a bitmap representing the path as integer at base 32,
  - the number of bits used to store the path,
  - the number of bits used to store a single item
  - the number of items in the path.

class sage.quivers.paths.QuiverPath

Bases: sage.structure.element.MonoidElement

Class for paths in a quiver.

A path is given by two vertices, start and end, and a finite (possibly empty) list of edges \( e_1, e_2, \ldots, e_n \) such that the initial vertex of \( e_1 \) is \( \text{start} \), the final vertex of \( e_i \) is the initial vertex of \( e_{i+1} \), and the final vertex of \( e_n \) is \( \text{end} \). In the case where no edges are specified, we must have \( \text{start} = \text{end} \) and the path is called the trivial path at the given vertex.

Note: Do not use this constructor directly! Instead, pass the input to the path semigroup that shall be the parent of this path.

EXAMPLES:

Specify a path by giving a list of edges:

```
sage: Q = DiGraph({1:{2:['a','d'], 3:['e']}, 2:{3:['b']}, 3:{1:['f'], 4:['c']}})
sage: F = Q.path_semigroup()
sage: p = F([(1, 2, 'a'), (2, 3, 'b')])
sage: p
a*b
```

Paths are not unique, but different representations of “the same” path yield equal paths:
The `*` operator is concatenation of paths. If the two paths do not compose, its result is `None`:

```python
sage: print(p*q)
None
sage: p*F([(3, 4, 'c')])
a*b*c
sage: F([(2,3,'b'), (3,1,'f')])*p
b*f*a*b
```

The length of a path is the number of edges in that path. Trivial paths are therefore length-0:

```python
sage: len(p)
2
sage: triv = F([(1, 1)])
sage: len(triv)
0
```

List index and slice notation can be used to access the edges in a path. QuiverPaths can also be iterated over. Trivial paths have no elements:

```python
sage: for x in p: print(x)
(1, 2, 'a')
(2, 3, 'b')
sage: list(triv)
[]
```

There are methods giving the initial and terminal vertex of a path:

```python
sage: p.initial_vertex()
1
sage: p.terminal_vertex()
3
```

**complement**(subpath)

Return a pair \((a,b)\) of paths s.t. \(self==a\cdot subpath\cdot b\), or \((None, None)\) if subpath is not a subpath of this path.

**NOTE:**

\(a\) is chosen of minimal length.

**EXAMPLES:**

```python
sage: S = DiGraph({1:{1:['a','b','c','d']}}).path_semigroup()
sage: S.inject_variables()
Defining e_1, a, b, c, d
sage: (b*c*a*d*b+a*d*d).complement(a*d)
(None, None)
sage: (b*c+a*d+b+a*d+d).complement(a*d)
(b*c, b*a*d*d)
sage: (b*c+a*d+b).complement(a*c)
(b*c+a*d+b, None)
```

**degree**

Return the length of the path.
\texttt{length()} and \texttt{degree()} are aliases \texttt{gcd}(P)

Greatest common divisor of two quiver paths, with co-factors.

For paths, by “greatest common divisor”, we mean the largest terminal segment of the first path that is an initial segment of the second path.

INPUT:

A \texttt{QuiverPath} P

OUTPUT:

\begin{itemize}
  \item \texttt{QuiverPath}`\ `(C1,G,C2)` such that self==C1*G and P=G*C2, or
  \item (None, None, None), if the paths do not overlap (or belong to different quivers).
\end{itemize}

EXAMPLES:

\begin{Verbatim}
sage: Q = DiGraph({1:{2:['a']}, 2:{1:['b'], 3:['c']}, 3:{1:['d']}}).path_semigroup()
sage: p1 = Q(['c','d','a','b','a','c','d'])
sage: p1
c*d*a*b*a*c*d
sage: p2 = Q(['a','b','a','c','d','a','c','d','a','b'])
sage: p2
a*b*a*c*d*a*c*d*a*b
sage: S1, G, S2 = p1.gcd(p2)
sage: S1, G, S2
(c*d, a*b*a*c*d, a*c*d*a*b)
\end{Verbatim}

We test that a full overlap is detected:

\begin{Verbatim}
sage: p2.gcd(p2)
(e_1, a*b*a*c*d*a*c*d*a*b, e_1)
\end{Verbatim}

The absence of an overlap is detected:

\begin{Verbatim}
sage: p2[2:-1].has_prefix(p2[1:])
(False, None, None)
\end{Verbatim}

\texttt{has\_prefix} (\texttt{subpath})

Tells whether this path starts with a given sub-path.

INPUT:

\texttt{subpath}, a path in the same path semigroup as this path.

OUTPUT:

0 or 1, which stands for False resp. True.
EXAMPLES:

```python
sage: S = DiGraph({0:{1:['a'], 2:['b']}, 1:{0:['c'], 1:['d']}, 2:{0:['e'], 2:['f']}).path_semigroup()
sage: S.inject_variables()
Defining e_0, e_1, e_2, a, b, c, d, e, f
sage: (c+b+e+a).has_prefix(b+e)
0
sage: (c+b+e+a).has_prefix(c+b)
1
sage: (c+b+e+a).has_prefix(e_1)
1
sage: (c+b+e+a).has_prefix(e_2)
0
```

\textbf{has\_subpath} \texttt{(subpath)}

Tells whether this path contains a given sub-path.

**INPUT:**

subpath, a path of positive length in the same path semigroup as this path.

**EXAMPLES:**

```python
sage: S = DiGraph({0:{1:['a'], 2:['b']}, 1:{0:['c'], 1:['d']}, 2:{0:['e'], 2:['f']}).path_semigroup()
sage: S.inject_variables()
Defining e_0, e_1, e_2, a, b, c, d, e, f
sage: (c+b+e+a).has_subpath(b+e)
1
sage: (c+b+e+a).has_subpath(b+f)
0
sage: (c+b+e+a).has_subpath(e_1)
Traceback (most recent call last):
... ValueError: We only consider sub-paths of positive length
sage: (c+b+e+a).has_subpath(None)
Traceback (most recent call last):
... ValueError: The given sub-path is empty
```

\textbf{initial\_vertex}()

Return the initial vertex of the path.

**OUTPUT:**

• integer, the label of the initial vertex

**EXAMPLES:**

```python
sage: Q = DiGraph({1:{2:['a']}, 2:{3:['b']}}).path_semigroup()
sage: y = Q([(1, 2, 'a'), (2, 3, 'b')])
sage: y.initial_vertex()
1
```

\textbf{length}()

Return the length of the path.

length() and degree() are aliases
**reversal()**

Return the path along the same edges in reverse order in the opposite quiver.

**EXAMPLES:**

```python
sage: Q = DiGraph({1:{2:['a']}, 2:{3:['b']}, 3:{4:['c'], 1:['d']}}).path_semigroup()
sage: p = Q([(1, 2, 'a'), (2, 3, 'b'), (3, 1, 'd'), (1, 2, 'a'), (2, 3, 'b'), (3, 4, 'c')])
sage: p
a*b*d*a*b*c
sage: p.reversal()
c*b*a*d*b*a
sage: e = Q.idempotents()[0]
sage: e
e_1
sage: e.reversal()
e_1
```

**terminal_vertex()**

Return the terminal vertex of the path.

**OUTPUT:**

- integer, the label of the terminal vertex

**EXAMPLES:**

```python
sage: Q = DiGraph({1:{2:['a']}, 2:{3:['b']}}).path_semigroup()
sage: y = Q([(1, 2, 'a'), (2, 3, 'b')])
sage: y.terminal_vertex()
3
```
A quiver is a directed graph used for representation theory. In our representation theoretic code, it is assumed that

- the vertices of the quiver are labelled by integers, and
- each edge of the quiver is labelled with a nonempty string. The label cannot begin with 'e_' or contain '*' and distinct edges must have distinct labels.

As far as the DiGraph class is concerned, a path is a finite list of pairwise distinct vertices \(v_1, \ldots, v_n\) such that there exists an edge from \(v_i\) to \(v_{i+1}\). If there are multiple edges between the same two vertices this does not contribute additional paths as listed by the DiGraph class; for example only two paths are listed from 1 to 3 in \(Q\):

```python
sage: Q = DiGraph({1:{2:['a','b'], 3:['c']}, 2:{3:['d']}})
sage: Q.edges()
[(1, 2, 'a'), (1, 2, 'b'), (1, 3, 'c'), (2, 3, 'd')]
sage: Q.all_paths(1, 3)
[[1, 2, 3], [1, 3]]
```

The notion of a path in a quiver (in representation theory) is fundamentally different in several aspects. First, paths are no longer required to have distinct vertices, or even distinct edges; thus, “path” in quiver theory is closer to the notion of “walk” in graph theory. Furthermore, paths in quiver theory “know” their edges, so parallel edges between the same two vertices of a Quiver make different paths. But paths in quiver theory also “know” their vertices, so that a length-0 path from \(a\) to \(a\) is not the same as a length-0 path from \(b\) to \(b\) for \(a \neq b\). Formally, we say that a path is given by two vertices, \(\text{start}\) and \(\text{end}\), and a finite (possibly empty) list of edges \(e_1, e_2, \ldots, e_n\) such that the initial vertex of \(e_1\) is \(\text{start}\), the final vertex of \(e_i\) is the initial vertex of \(e_{i+1}\), and the final vertex of \(e_n\) is \(\text{end}\). In the case where no edges are specified, we must have \(\text{start} = \text{end}\) and the path is called the trivial path at the given vertex.

Quiver paths in the sense stated above correspond to the elements of a partial semigroup, with multiplication of paths given by concatenation. Hence, rather than overloading the method name inherited from DiGraph or inventing a new method name, we move this functionality to this so-called \textit{path semigroup}. Note that with this definition there are three paths from 1 to 3 in our example:

```python
sage: Q.path_semigroup().all_paths(1, 3)
[a*d, b*d, c]
```

The returned paths are of type \textit{QuiverPath}, which are elements in the path semigroup that is associated with the quiver (a partial semigroup, which does not generally have a neutral element). You can specify a \textit{QuiverPath} by giving an edge or a list of edges, passed as arguments to the path semigroup containing this path. Here an edge is a tuple of the form \((i, j, l)\), where \(i\) and \(j\) are vertices and \(l\) is the label of an edge from \(i\) to \(j\):
Trivial paths are indicated by passing a list containing the tuple \((\text{vertex}, \text{vertex})\):

```python
sage: Q.path_semigroup()([[3, 3]])
e_3
```

Here is an alternative way to define a path:

```python
sage: PQ = Q.path_semigroup()
sage: q = PQ(['a', 'd'])
sage: p == q
True
```

If the vertices along the path do not match, a value error is raised:

```python
sage: inv1 = PQ([[2, 3, 'd'], [1, 2, 'a']])
Traceback (most recent call last):
  ...
ValueError: Edge d ends at 3, but edge a starts at 1
sage: inv2 = PQ([[1, 2, 'a'], [1, 2, 'a']])
Traceback (most recent call last):
  ...
ValueError: Edge a ends at 2, but edge a starts at 1
sage: inv3 = PQ([[1, 2, 'x']])
Traceback (most recent call last):
  ...
ValueError: (1, 2, 'x') is not an edge
```

The \(*\) operator is concatenation of paths. If the two paths do not compose, then the result is \texttt{None} (whence the “partial” in “partial semigroup”).

```python
sage: print(p*q)
None
```

Let us now construct a larger quiver:

```python
sage: Qbig = DiGraph({1:{2:['a','b'], 3:['c']}, 2:{3:['d']}, 3:{4:['e']}, 4:{5:['f']},
                   5:{1:['g']}})
sage: Pbig = Qbig.path_semigroup()
```

Since \(Q\) is a sub-digraph of \(Q\_\text{big}\), we have a coercion of the associated path semigroups:

```python
sage: Pbig.has_coerce_map_from(PQ)
True
```

In particular, \(p\) is considered to be an element of \(P\_\text{big}\), and can be composed with paths that were defined for the larger quiver:

```python
sage: p in Pbig
True
sage: p*Pbig([(3, 4, 'e')])
a*d*e
sage: Pbig([(4, 5, 'f'), (5, 1, 'g')])*p
f*g*a*d
```
The length of a path is the number of edges in that path:

```
sage: len(p)
2
sage: triv = PQ([(1, 1)])
sage: len(triv)
0
```

List index and slice notation can be used to access the edges in a path. QuiverPaths can also be iterated over. Trivial paths have no elements:

```
sage: for x in p: print(x)
(1, 2, 'a')
(2, 3, 'd')
sage: triv[:]
e_1
```

There are methods giving the initial and terminal vertex of a path:

```
sage: p.initial_vertex()
1
sage: p.terminal_vertex()
3
```

**QuiverPath** form the basis of the quiver algebra of a quiver. Given a field \( k \) and a quiver \( Q \), the quiver algebra \( kQ \) is, as a vector space, the free \( k \)-vector space whose basis is the set of all paths in \( Q \). Multiplication is defined on this basis and extended bilinearly. The product of two basis elements is given by path composition when it makes sense and is set to be zero otherwise. Specifically, if the terminal vertex of the left path equals the initial vertex of the right path, then their product is the concatenation of the two paths, and otherwise their product is zero. In sage, quiver algebras are handled by the **QuiverAlgebra** class:

```
sage: A = PQ.algebra(GF(7))
sage: A
Path algebra of Multi-digraph on 3 vertices over Finite Field of size 7
```

Quivers have a method that creates their algebra over a given field (or, more generally, commutative ring). Note that **QuiverAlgebras** are uniquely defined by their quiver and field, and play nicely with coercions of the underlying path semigroups:

```
sage: A is PQ.algebra(GF(7))
True
sage: A is PQ.algebra(RR)
False
sage: Q1 = Q.copy()
sage: Q1.add_vertex(4)
sage: PQ1 = Q1.path_semigroup()
sage: A is PQ1.algebra(GF(7))
False
sage: Pbig.algebra(GF(7)).has_coerce_map_from(A)
True
```

The **QuiverAlgebra** can create elements from **QuiverPaths** or from elements of the base ring:

```
sage: A(5)
5*e_1 + 5*e_2 + 5*e_3
sage: r = PQ([(1, 2, 'b'), (2, 3, 'd')])
sage: e2 = PQ([(2, 2)])
```

(continues on next page)
QuiverAlgebras are $\mathbb{N}$-graded algebras. The grading is given by assigning to each basis element the length of the path corresponding to that basis element:

```python
sage: x.is_homogeneous()
False
sage: x.degree()
Traceback (most recent call last):
  ...ValueError: Element is not homogeneous.
sage: y.is_homogeneous()
True
sage: y.degree()
2
```

The category of right modules over a given quiver algebra is equivalent to the category of representations of that quiver. A quiver representation is a diagram in the category of vector spaces whose underlying graph is the quiver. So to each vertex of the quiver we assign a vector space and to each edge of the quiver a linear map between the vector spaces assigned to the start and end vertices of that edge. To create the zero representation we just specify the base ring and the path semigroup:

```python
sage: Z = Q1.path_semigroup().representation(GF(5))
sage: Z.is_zero()
True
```

To each vertex of a quiver there is associated a simple module, an indecomposable projective, and an indecomposable injective, and these can be created from the qQuiver:

```python
sage: S = PQ.S(GF(3), 1)
sage: I = PQ.I(QQ, 2)
sage: P = PQ.P(GF(3), 1)
```

Radicals, socles, tops, and quotients can all be computed and we can test if modules are simple or semisimple, get their dimension, and test for equality. Like quivers, QuiverRep objects are unique and therefore equal if and only if they are identical:

```python
sage: P.is_simple()
False
sage: P.dimension()
6
sage: R = P.radical()
sage: P.socle()
Representation with dimension vector (0, 0, 3)
sage: (P/R).is_simple()
True
sage: P == R
```

(continues on next page)
There are special methods to deal with modules that are given as right ideals in the quiver algebra. To create such a module pass the keyword `option='paths'` along with a path or list of paths that generate the desired ideal:

```
sage: M = PQ.representation(QQ, \[(1, 1), (1, 2, 'a')\], option='paths')
sage: M.dimension_vector()
(1, 2, 3)
```

There are also special methods to deal with modules that are given as the linear dual of a right ideal in the quiver algebra. To create such a module, pass the keyword `option='dual paths'` to the constructor along with a path or list of paths. The module returned is the dual of the ideal created in the opposite quiver by the reverses of the given paths:

```
sage: D = PQ.representation(QQ, \[(1, 1), (1, 2, 'a')\], option='dual paths')
sage: D.dimension_vector()
(2, 0, 0)
```

For modules that are not a standard module or an ideal of the quiver algebra `QuiverRep` can take as input two dictionaries. The first associates to each vertex a vector space or an integer (the desired dimension of the vector space), the second associates to each edge a map or a matrix or something from which Sage can construct a map:

```
sage: PQ2 = DiGraph({1:{2:['a', 'b']}}).path_semigroup()
sage: M2 = PQ2.representation(QQ, {1: QQ^2, 2: QQ^1}, {(1, 2, 'a'): [1, 0], (1, 2, 'b -->'): [0, 1]})
sage: M2.get_space(2)
Vector space of dimension 2 over Rational Field
sage: M2.get_space(2)
Vector space of dimension 1 over Rational Field
sage: M2.get_map((1, 2, 'a'))
Vector space morphism represented by the matrix:
[1 0]
Domain: Vector space of dimension 1 over Rational Field
Codomain: Vector space of dimension 2 over Rational Field
```

A homomorphism between two quiver representations is given by homomorphisms between the spaces assigned to the vertices of those representations such that those homomorphisms commute with the edge maps of the representations. The homomorphisms are created in the usual Sage syntax, the defining data given by a dictionary associating maps to vertices:

```
sage: P2 = PQ2.P(QQ, 1)
sage: f = P2.hom({1:[1, 1], 2:[[1], [1]]}, M2)
```

When the domain is given as a right ideal in the quiver algebra we can also create a homomorphism by just giving a single element in the codomain. The map is then induced by acting on that element:

```
sage: x = P2.gens('x')[0]
sage: x
x_0
sage: f == P2.hom(f(x), M2)
True
```

As you can see, the above homomorphisms can be applied to elements. Just like elements, addition is defined via the `+` operator. On elements scalar multiplication is defined via the `*` operator but on homomorphisms `*` defines...
composition, so scalar multiplication is done using a method:

```python
sage: g = f + f
sage: g == f.scalar_mult(2)
True
sage: g == 2*f  # This multiplies the map with the scalar 2
True
sage: g(x) == 2*f(x)  # This applies the map, then multiplies by the scalar
True
```

The `direct_sum` method for modules returns only the resulting module by default. But can also return the projection and inclusion homomorphisms into the various factors:

```python
sage: N2, inclusions, projections = M2.direct_sum([P2], return_maps=True)
\[\text{inclusions[0].domain()} \text{ is M2}
\text{True}
\text{projections[0].codomain()} \text{ is M2}
\text{True}
\text{(projections[0]*inclusions[0]).is_isomorphism()}
\text{True}
```

As you see above we can determine if a given map is an isomorphism. Testing for injectivity and surjectivity works as well:

```python
sage: f.is_injective()
False
sage: f.is_surjective()
False
```

We can create all the standard modules associated to maps:

```python
sage: f.kernel()
Representation with dimension vector (0, 1)
\text{sage: f.cokernel()}
Representation with dimension vector (1, 0)
\text{sage: im = f.image()}
\text{sage: im}
Representation with dimension vector (1, 1)
```

These methods, as well as the `submodule` and `quotient` methods that are defined for representations, return only the resulting representation. To get the inclusion map of a submodule or the factor homomorphism of a quotient use `coerce_map_from`:

```python
\text{sage: incl = M2.coerce_map_from(im)}
\text{sage: incl.domain()} \text{ is im}
\text{True}
\text{sage: incl.codomain()} \text{ is M2}
\text{True}
\text{sage: incl.is_injective()}
\text{True}
```

Both `QuiverRep` objects and `QuiverRepHom` objects have `linear_dual` and `algebraic_dual` methods. The `linear_dual` method applies the functor $\text{Hom}_k(\cdots, k)$ where $k$ is the base ring of the representation, and the `algebraic_dual` method applies the functor $\text{Hom}_Q(\cdots, kQ)$ where $kQ$ is the quiver algebra. Both these functors yield left modules. A left module is equivalent to a right module over the opposite algebra, and the opposite of a quiver algebra is the algebra of the opposite quiver, so both these methods yield modules and representations of the opposite quiver:
There is a method returning the projective cover of any module. Note that this method returns the homomorphism; to get the module take the domain of the homomorphism:

```
sage: cov = M2.projective_cover()
sage: cov
Homomorphism of representations of Multi-digraph on 2 vertices
sage: cov.domain()
Representation with dimension vector (2, 4)
```

As projective covers are computable, so are the transpose and Auslander-Reiten translates of modules:

```
sage: M2.transpose()
Representation with dimension vector (4, 3)
sage: PQ2.I(QQ, 1).AR_translate()
Representation with dimension vector (3, 2)
```

We have already used the `gens` method above to get an element of a quiver representation. An element of a quiver representation is simply a choice of element from each of the spaces assigned to the vertices of the quiver. Addition, subtraction, and scalar multiplication are performed pointwise and implemented by the usual operators:

```
sage: M2.dimension_vector()
(2, 1)
sage: x, y, z = M2.gens('xyz')
sage: 2*x + y != x + 2*y
True
```

To create a specific element of a given representation we just specify the representation and a dictionary associating to each vertex an element of the space associated to that vertex in the representation:

```
sage: w = M2({1:(1, -1), 2:(3,)})
sage: w.get_element(1)
(1, -1)
```

The right action of a quiver algebra on an element is implemented via the `*` operator:

```
sage: A2 = x.quiver().path_semigroup().algebra(QQ)
sage: a = A2('a')
sage: x*a == z
True
```

```
class sage.quivers.representation.QuiverRepElement (parent, elements=None, name=None)

    Bases: sage.structure.element.ModuleElement

    An element of a quiver representation is a choice of element from each of the spaces assigned to the vertices of the quiver. Addition, subtraction, and scalar multiplication of these elements is done pointwise within these spaces.

    INPUT:
```
• module – QuiverRep (default: None), the module to which the element belongs

• elements - dict (default: empty), a dictionary associating to each vertex a vector or an object from which sage can create a vector. Not all vertices must be specified, unspecified vertices will be assigned the zero vector of the space associated to that vertex in the given module. Keys that do not correspond to a vertex are ignored.

• name – string (default: None), the name of the element

OUTPUT:

• QuiverRepElement

Note: The constructor needs to know the quiver in order to create an element of a representation over that quiver. The default is to read this information from module as well as to fill in unspecified vectors with the zeros of the spaces in module. If module is None then quiver MUST be a quiver and each vertex MUST be specified or an error will result. If both module and quiver are given then quiver is ignored.

EXAMPLES:

```python
sage: Q = DiGraph({1:{2:['a'], 3:['b']}, 2:{3:['c']}}).path_semigroup()
sage: spaces = dict((v, GF(3)^2) for v in Q.quiver())
sage: M = Q.representation(GF(3), spaces)
sage: elems = {1: (1, 0), 2: (0, 1), 3: (2, 1)}
sage: M(elems)
Element of quiver representation
sage: v = M(elems, 'v')
sage: v
v
sage: (v + v + v).is_zero()
True
```

copy()  
Return a copy of self.

EXAMPLES:

```python
sage: Q = DiGraph({1:{2:['a'], 3:['b']}, 2:{3:['c']}}).path_semigroup()
sage: spaces = dict((v, GF(3)^2) for v in Q.quiver())
sage: M = Q.representation(GF(3), spaces)
sage: elems = {1: (1, 0), 2: (0, 1), 3: (2, 1)}
sage: v = M(elems)
sage: w = v.copy()
sage: w._set_element((0, 0), 1)
sage: w.get_element(1)
(0, 0)
sage: v.get_element(1)
(1, 0)
```

get_element(vertex)  
Return the element at the given vertex.

INPUT:

• vertex – integer, a vertex of the quiver

OUTPUT:

• vector, the vector assigned to the given vertex
EXAMPLES:

```python
sage: Q = DiGraph({1:{2:['a'], 3:['b']}, 2:{3:['c']}}).path_semigroup()
sage: spaces = dict((v, GF(3)^2) for v in Q.quiver())
sage: M = Q.representation(GF(3), spaces)
sage: elems = {1: (1, 0), 2: (0, 1), 3: (2, 1)}
sage: v = M(elems)
sage: v.get_element(1)
(1, 0)
sage: v.get_element(3)
(2, 1)
```

**is_zero()**

Test whether `self` is zero.

**OUTPUT:**

- bool, `True` if the element is the zero element, `False` otherwise

**EXAMPLES:**

```python
sage: Q = DiGraph({1:{2:['a'], 3:['b']}, 2:{3:['c']}}).path_semigroup()
sage: spaces = dict((v, GF(3)^2) for v in Q.quiver())
sage: M = Q.representation(GF(3), spaces)
sage: elems = {1: (1, 0), 2: (0, 1), 3: (2, 1)}
sage: v = M(elems)
sage: v.is_zero()
False
sage: w = M()
sage: w.is_zero()
True
```

**quiver()**

Return the quiver of the representation.

**OUTPUT:**

- `DiGraph`, the quiver of the representation

**EXAMPLES:**

```python
sage: Q = DiGraph({1:{2:['a'], 3:['b']}, 2:{3:['c']}}).path_semigroup()
sage: P = Q.P(QQ, 1)
sage: v = P.an_element()
sage: v.quiver() == Q.quiver()
True
```

**support()**

Return the support of `self` as a list.

The support is the set of vertices to which a nonzero vector is associated.

**OUTPUT:**

- list, the support

**EXAMPLES:**

```python
sage: Q = DiGraph({1:{2:['a'], 3:['b']}, 2:{3:['c']}}).path_semigroup()
sage: spaces = dict((v, GF(3)^2) for v in Q.quiver())
sage: M = Q.representation(GF(3), spaces)
```

(continues on next page)
A quiver representation is a diagram in the category of vector spaces whose underlying graph is the quiver. Giving a finite dimensional representation is equivalent to giving a finite dimensional right module for the path algebra of the quiver.

**INPUT:**

The first two arguments specify the base ring and the quiver, and they are always required:

- \( k \) – ring, the base ring of the representation
- \( P \) – the partial semigroup formed by the paths of the quiver of the representation

Then to specify the spaces and maps associated to the quiver there are three possible options. The first is the 'values' option, where the next two arguments give the data to be assigned. The following can either be the next two entries in the argument list or they can be passed by keyword. If the argument list is long enough the keywords are ignored; the keywords are only checked in the event that the argument list does not have enough entries after \( P \).

- \( \text{spaces} \) – dict (default: empty); a dictionary associating to each vertex a free module over the base ring \( k \). Not all vertices must be specified; unspecified vertices are automatically set to \( k^0 \). Keys of the dictionary that don’t correspond to vertices are ignored.
- \( \text{maps} \) – dict (default: empty); a dictionary associating to each edge a map whose domain and codomain are the spaces associated to the initial and terminal vertex of the edge respectively. Not all edges must be specified; unspecified edges are automatically set to the zero map. Keys of the dictionary that don’t correspond to edges are ignored.

The second option is the paths option which creates a module by generating a right ideal from a list of paths. Thus the basis elements of this module correspond to paths of the quiver and the maps are given by right multiplication by the corresponding edge. As above this can be passed either as the next entry in the argument list or as a keyword. The keyword is only checked if there is no entry in the argument list after \( P \).

- \( \text{basis} \) – list; a nonempty list of paths in the quiver \( Q \). Entries that do not represent valid paths are ignored and duplicate paths are deleted. There must be at least one valid path in the list or a ValueError is raised. The closure of this list under right multiplication forms the basis of the resulting representation.

The third option is the dual paths option which creates the dual of a left ideal in the quiver algebra. Thus the basis elements of this module correspond to paths of the quiver and the maps are given by deleting the corresponding edge from the start of the path (the edge map is zero on a path if that edge is not the initial edge of the path). As above this can be passed either as the next entry in the argument list or as a keyword.

- \( \text{basis} \) – list; a nonempty list of paths in the quiver \( Q \). Entries that do not represent valid paths are ignored and duplicate paths are deleted. There must be at least one valid path in the list or a ValueError is raised. The closure of this list under left multiplication of edges forms the basis of the resulting representation.

Using the second and third options requires that the following keyword be passed to the constructor. This must be passed as a keyword.

- \( \text{option} \) – string (default: None), either 'values' or 'paths' or 'dual paths'. None is equivalent to 'values'.

**OUTPUT:**
• QuiverRep

EXAMPLES:

```
sage: Q1 = DiGraph({1:(2:['a'])}).path_semigroup()
```

When the `option` keyword is not supplied the constructor uses the 'values' option and expects the spaces and maps to be specified. If no maps or spaces are given the zero module is created:

```
sage: M = Q1.representation(GF(5))
sage: M.is_zero()
True
```

The simple modules, indecomposable projectives, and indecomposable injectives are examples of quiver representations:

```
sage: S = Q1.S(GF(3), 1)
sage: I = Q1.I(QQ, 2)
sage: P = Q1.P(GF(3), 1)
```

Various standard submodules can be computed, such as radicals and socles. We can also form quotients and test for certain attributes such as semisimplicity:

```
sage: R = P.radical()
sage: R.is_zero()
False
sage: (P/R).is_simple()
True
sage: P == R
False
```

With the option 'paths' the input data should be a list of `QuiverPaths` or things that `QuiverPaths` can be constructed from. The resulting module is the submodule generated by these paths in the quiver algebra, when considered as a right module over itself:

```
sage: P1 = Q1.representation(QQ, [[[1, 1]]], option='paths')
sage: P1.dimension()
2
```

In the following example, the 3rd and 4th paths are actually the same, so the duplicate is removed:

```
sage: N = Q1.representation(QQ, [[[1, 1]], [[2, 2]], [[1, 2, 'a']], [[1, 2, 'a ↷']]], option='paths')
sage: N.dimension()
3
```

The dimension at each vertex equals the number of paths in the closed basis whose terminal point is that vertex:

```
sage: Q2 = DiGraph({1:(2:['a'], 3:['b', 'c']), 2:{3:['d']}}).path_semigroup()
sage: M = Q2.representation(QQ, [[[2, 2]], [[1, 2, 'a']]], option='paths')
sage: M.dimension_vector()
(0, 2, 2)
sage: N = Q2.representation(QQ, [[[2, 2]], [[1, 2, 'a'], (2, 3, 'd')]], option='paths')
sage: N.dimension_vector()
(0, 1, 2)
```

`create_key (k, P, *args, **kwds)`

Return a key for the specified module.
The key is a tuple. The first and second entries are the base ring $k$ and the partial semigroup $P$ formed by the paths of a quiver. The third entry is the option and the remaining entries depend on that option. If the option is 'values' and the quiver has $n$ vertices then the next $n$ entries are the vector spaces to be assigned to those vertices. After that are the matrices of the maps assigned to edges, listed in the same order that $Q\text{.edges()}$ uses. If the option is 'paths' or 'dual paths' then the next entry is a tuple containing a sorted list of the paths that form a basis of the quiver.

INPUT:

See the class documentation.

OUTPUT:

• tuple

EXAMPLES:

```python
sage: P = DiGraph({1:{2:['a']}}).path_semigroup()
sage: from sage.quivers.representation import QuiverRep
sage: QuiverRep.create_key(GF(5), P)
(Finite Field of size 5,
Partial semigroup formed by the directed paths of Multi-digraph on 2
vertices,
'values',
Vector space of dimension 0 over Finite Field of size 5,
Vector space of dimension 0 over Finite Field of size 5,
[])
```

`create_object` *(version, key, **extra_args)*

Create a `QuiverRep_generic` or `QuiverRep_with_path_basis` object from the key.

The key is a tuple. The first and second entries are the base ring $k$ and the quiver $Q$. The third entry is the 'option' and the remaining entries depend on that option. If the option is 'values' and the quiver has $n$ vertices then the next $n$ entries are the vector spaces to be assigned to those vertices. After that are the matrices of the maps assigned to edges, listed in the same order that $Q\text{.edges()}$ uses. If the option is 'paths' or 'dual paths' then the next entry is a tuple containing a sorted list of the paths that form a basis of the quiver.

INPUT:

• version – the version of sage, this is currently ignored
• key – tuple

OUTPUT:

• `QuiverRep_generic` or `QuiverRep_with_path_basis`

EXAMPLES:

```python
sage: Q = DiGraph({1:{2:['a']}}).path_semigroup()
sage: from sage.quivers.representation import QuiverRep
sage: key = QuiverRep.create_key(GF(5), Q)
sage: QuiverRep.create_object(0, key)
Representation with dimension vector (0, 0)
```

`class` `sage.quivers.representation.QuiverRep_generic(k, P, spaces, maps)`

Bases: `sage.misc.fast_methods.WithEqualityById, sage.modules.module.Module`

A generic quiver representation.

This class should not be called by the user.
Call `QuiverRep` with `option='values'` (which is the default) instead.

**INPUT:**

- `k` – ring, the base ring of the representation
- `P` – the path semigroup of the quiver `Q` of the representation
- `spaces` – dict (default: empty), a dictionary associating to each vertex a free module over the base ring `k`. Not all vertices need to be specified, unspecified vertices are automatically set to `k^0`. Keys of the dictionary that don’t correspond to vertices are ignored.
- `maps` – dict (default: empty), a dictionary associating to each edge a map whose domain and codomain are the spaces associated to the initial and terminal vertex of the edge respectively. Not all edges need to be specified, unspecified edges are automatically set to the zero map. Keys of the dictionary that don’t correspond to edges are ignored.

**OUTPUT:**

- `QuiverRep`

**EXAMPLES:**

```python
sage: Q = DiGraph({1:{3:['a']}, 2:{3:['b']}}).path_semigroup()
sage: spaces = {1: QQ^2, 2: QQ^3, 3: QQ^2}
sage: maps = {(1, 3, 'a'): (QQ^2).Hom(QQ^2).identity(), (2, 3, 'b'): [[1, 0], [0, 1]], (0, 0)}
sage: M = Q.representation(QQ, spaces, maps)
```

**AR_translate()**

Return the Auslander-Reiten translate of `self`.

**EXAMPLES:**

```python
sage: Q = DiGraph({1:{2:['a', 'b']}}).path_semigroup()
sage: M = Q.representation(GF(3), {1: 1, 2: 1}, {(1, 2, 'a'): 1})
sage: tauM = M.AR_translate()
sage: tauM
Representation with dimension vector (1, 1)
sage: tauM.get_map((1, 2, 'a')).matrix()
[1]
sage: tauM.get_map((1, 2, 'b')).matrix()
[0]
```

The module `M` above is its own AR translate. This is not always true:

```python
sage: Q2 = DiGraph({3:{1:['b']}, 5:{3:['a']}}).path_semigroup()
sage: Q2.S(QQ, 5).AR_translate()
Representation with dimension vector (0, 1, 0)
```

**Element**

alias of `QuiverRepElement`
Hom(*codomain*)
Return the hom space from *self* to *codomain*.
For more information see the QuiverHomSpace documentation.

actor()
Return the quiver path algebra acting on this representation.

OUTPUT:
• a quiver path algebra

EXAMPLES:
```
sage: Q = DiGraph({1:{2:['a']}}).path_semigroup()
sage: M = Q.representation(GF(5))
sage: M.base_ring()
Finite Field of size 5
sage: M.actor()
Path algebra of Multi-digraph on 2 vertices over Finite Field of size 5
```

algebraic_dual(*basis=False*)
Compute the algebraic dual $\text{Hom}_Q(M, kQ)$ of the module $M = \text{self}$.

INPUT:
• *basis* – bool; if False, then only the module is returned. If True, then a tuple is returned. The first element is the QuiverRep and the second element is a dictionary which associates to each vertex a list. The elements of this list are the homomorphisms which correspond to the basis elements of that vertex in the module.

OUTPUT:
• QuiverRep or tuple

Note: Here $kQ$ is the path algebra considered as a right module over itself. If $e$ is an edge of the quiver $Q$ then we let $(fe)(m) = ef(m)$. This gives $\text{Hom}_Q(M, kQ)$ a module structure over the opposite quiver $Q.reverse()$.

EXAMPLES:
```
sage: Q = DiGraph({1:{2:['a', 'b'], 3: ['c', 'd']}, 2:{3:['e']}}).path_semigroup()
sage: Q.free_module(GF(7)).algebraic_dual().dimension_vector()
(7, 2, 1)
```

an_element()
Return an element of *self*.

OUTPUT:
• QuiverRepElement

EXAMPLES:
```
sage: Q = DiGraph({1:{2:['a', 'b']}}).path_semigroup()
sage: M = Q.P(QQ, 1)
sage: M.an_element()
Element of quiver representation
```
coordinates(vector)
Return the coordinates when vector is expressed in terms of the gens.

INPUT:

• vector – QuiverRepElement

OUTPUT:

• list, the coefficients when the vector is expressed as a linear combination of the generators of the module

EXAMPLES:

```python
sage: Q = DiGraph({1:{2:['a', 'b']}}).path_semigroup()
sage: M = Q.P(QQ, 1)
sage: x, y, z = M.gens('xyz')
sage: M.coordinates(x - y + z)
[1, -1, 1]
sage: M.coordinates(M.an_element())
[1, 1, 0]
sage: M.an_element() == x + y
True
```

dimension(vertex=None)
Return the dimension of the space associated to the given vertex vertex.

INPUT:

• vertex – integer or None (default: None), the given vertex

OUTPUT:

• integer, the dimension over the base ring of the space associated to the given vertex. If vertex=None then the dimension over the base ring of the module is returned

EXAMPLES:

```python
sage: Q = DiGraph({1:{2:['a', 'b']}}).path_semigroup()
sage: P = Q.P(GF(2), 1)
sage: P.dimension(1)
1
sage: P.dimension(2)
2
```

The total dimension of the module is the sum of the dimensions at each vertex:

```python
sage: P.dimension()
3
```

dimension_vector()
Return the dimension vector of the representation.

OUTPUT:

• tuple

Note: The order of the entries in the tuple matches the order given by calling the vertices() method on the quiver.

EXAMPLES:
Each coordinate of the dimension vector is the dimension of the space associated to that coordinate:

```python
sage: P.get_space(2).dimension()
2
```

**direct_sum** *(modules, return_maps=False)*

Return the direct sum of self with the given modules modules.

The modules must be modules over the same quiver and base ring.

**INPUT:**

- modules – QuiverRep or list of QuiverRep’s
- return_maps – Boolean (default: False); if False, then the output is a single QuiverRep object which is the direct sum of self with the given module or modules. If True, then the output is a list [sum, iota, pi]. The first entry sum is the direct sum of self with the given module or modules. Both iota and pi are lists of QuiverRepHoms with one entry for each summand; iota[i] is the inclusion map and pi[i] is the projection map of the i-th summand. The summands are ordered as given with self being the zeroth summand.

**OUTPUT:**

- QuiverRep or tuple

**EXAMPLES:**

```python
sage: Q = DiGraph({1:{2:['a', 'b']}, 2:{4:['c']}, 3:{4:['d']}}).path_semigroup()
sage: P1 = Q.P(QQ, 1)
sage: P2 = Q.P(QQ, 2)
sage: S = P1.direct_sum(P2)
sage: P1.dimension_vector()
(1, 1, 1, 2)
sage: P2.dimension_vector()
(0, 1, 0, 1)
sage: S.dimension_vector()
(1, 2, 1, 3)
sage: S, iota, pi = P1.direct_sum(P2, return_maps=True)
sage: iota[0].domain() is P1
True
sage: iota[1].domain() is P2
True
sage: pi[0].codomain() is P1
True
sage: pi[1].codomain() is P2
True
sage: iota[0].codomain() is S
True
sage: iota[1].codomain() is S
True
sage: pi[0].domain() is S
True
sage: pi[1].domain() is S
True
```
```python
sage: iota[0].get_matrix(4)
[1 0 0]
[0 1 0]
sage: pi[0].get_matrix(4)
[1 0]
[0 1]
[0 0]
sage: P1prime = S/iota[1].image()
sage: f = P1prime.coerce_map_from(S)
sage: g = f*iota[0]
sage: g.is_isomorphism()
True
```

**gens** *(names='v')*

Return a list of generators of self as a $k$-module.

**INPUT:**

- names – an iterable variable of length equal to the number of generators, or a string (default: 'v');
  gives the names of the generators either by giving a name to each generator or by giving a name to
  which an index will be appended

**OUTPUT:**

- list of QuiverRepElement objects, the linear generators of the module (over the base ring)

**Note:** The generators are ordered first by vertex and then by the order given by the `gens()` method of
the space associated to that vertex.

**EXAMPLES:**

```python
sage: Q = DiGraph({1:{2:['a', 'b']}}).path_semigroup()
sage: M = Q.P(QQ, 1)
sage: M.gens()
[v_0, v_1, v_2]
```

If a string is given then it is used as the name of each generator, with the index of the generator appended
in order to differentiate them:

```python
sage: M.gens('generator')
[generator_0, generator_1, generator_2]
```

If a list or other iterable variable is given then each generator is named using the appropriate entry. The
length of the variable must equal the number of generators (the dimension of the module):

```python
sage: M.gens(['w', 'x', 'y', 'z'])
Traceback (most recent call last):
...
TypeError: can only concatenate list (not "str") to list
```

Strings are iterable, so if the length of the string is equal to the number of generators then the characters of
the string will be used as the names:
sage: M.gens('xyz')
[x, y, z]

def get_map(edge):
    """Return the map associated to the given edge edge."
    ""
    INPUT:
    - edge -- tuple of the form (initial vertex, terminal vertex, label) specifying the edge whose map is returned
    EXAMPLES:

    sage: Q = DiGraph({1:{2:['a', 'b']}}).path_semigroup()
sage: Q.P(ZZ, 1).get_map((1, 2, 'a'))
Free module morphism defined by the matrix
[1 0]
Domain: Ambient free module of rank 1 over the principal ideal domain Integer Ring
Codomain: Ambient free module of rank 2 over the principal ideal domain Integer Ring

def get_space(vertex):
    """Return the module associated to the given vertex vertex."
    ""
    INPUT:
    - vertex -- integer, a vertex of the quiver of the module
    EXAMPLES:

    sage: Q = DiGraph({1:{2:['a'], 3:['b']}}).path_semigroup()
sage: Q.P(QQ, 1).get_space(1)
Vector space of dimension 1 over Rational Field

def is_semisimple():
    """Test whether the representation is semisimple."
    ""
    OUTPUT:
    - bool
    EXAMPLES:

    sage: Q = DiGraph({1:{2:['a', 'b']}}).path_semigroup()
sage: M = Q.P(QQ, 1)
sage: (M/M.radical()).is_semisimple()
True

def is_simple():
    """Test whether the representation is simple."
    ""
    OUTPUT:
    - bool
    EXAMPLES:

    sage: Q = DiGraph({1:{2:['a', 'b']}}).path_semigroup()
sage: Q.P(RR, 1).is_simple()
False

(continues on next page)
is_zero()
Test whether the representation is zero.

OUTPUT:
• bool

EXAMPLES:
```
sage: Q = DiGraph({1:{2:['a', 'b']}}).path_semigroup()
sage: M = Q.representation(ZZ)
sage: N = Q.representation(ZZ, {1: 1})
sage: M
Representation with dimension vector (0, 0)
sage: N
Representation with dimension vector (1, 0)
sage: M.is_zero()  # True
sage: N.is_zero()  # False
```

linear_combination_of_basis(coordinates)
Return the linear combination of the basis for self given by coordinates.

INPUT:
• coordinates – list; a list whose length is the dimension of self. The i-th element of this list defines the coefficient of the i-th basis vector in the linear combination.

OUTPUT:
• QuiverRepElement

EXAMPLES:
```
sage: Q = DiGraph({1:{2:['a', 'b']}}).path_semigroup()
sage: M = Q.P(QQ, 1)
sage: x, y, z = M.gens('xyz')
sage: e = x - y + z
sage: M.coordinates(e)
[1, -1, 1]
sage: M.linear_combination_of_basis([1, -1, 1]) == e
True
```

linear_dual()
Compute the linear dual $Hom_k(M,k)$ of the module $M = self$ over the base ring $k$.

OUTPUT:
• QuiverRep, the dual representation

Note: If $e$ is an edge of the quiver $Q$ then we let $(fe)(m) = f(me)$. This gives $Hom_k(M,k)$ a module structure over the opposite quiver $Q.reverse()$.

EXAMPLES:
sage: Q = DiGraph({1:[2:('a','b')], 2:[3:['c']]}).path_semigroup()
sage: M = Q.P(QQ, 1)
sage: M.linear_dual()
Representation with dimension vector (1, 2, 2)
sage: M.linear_dual().quiver() is Q.reverse().quiver()
True

**projective_cover** *(return_maps=**False**)*

Return the projective cover of *self*.

**EXAMPLES:**

```sage
sage: Q = DiGraph({1:[2:('a','b')], 2:[3:['c','d']]}).path_semigroup()
sage: S1 = Q.S(GF(3), 1)
sage: f1 = S1.projective_cover()
sage: f1.is_surjective()
True
```

**quiver**()

Return the quiver of the representation.

**OUTPUT:**

- DiGraph

**EXAMPLES:**

```sage
sage: Q = DiGraph({1:[2:['a']]}) .path_semigroup()
sage: M = Q.representation(GF(5))
sage: M.quiver() is Q.quiver()
True
```

**quotient** *(sub, check=True)*

Return the quotient of *self* by the submodule *sub*.

**INPUT:**

- *sub* – QuiverRep; this must be a submodule of *self*, meaning the space associated to each vertex *v* of *sub* is a subspace of the space associated to *v* in *self* and the map associated to each edge *e* of *sub* is the restriction of the map associated to *e* in *self*
- *check* – bool; if True then *sub* is checked to verify that it is indeed a submodule of *self* and an error is raised if it is not

**OUTPUT:**

- QuiverRep, the quotient module *self* / *sub*

**Note:** This function returns only a QuiverRep object *quot*. The projection map from *self* to *quot* can be obtained by calling *quot.coerce_map_from(*self)*.

**EXAMPLES:**
radical()
Return the Jacobson radical of self.

OUTPUT:
• QuiverRep, the Jacobson radical

EXAMPLES:

```python
sage: Q = DiGraph({1:{2:['a','b']}, 2:{3:['c']}}).path_semigroup()
sage: M = Q.P(QQ, 1)
sage: M.radical()
```

right_edge_action(element, path)
Return the result of element*path.

INPUT:
• element – QuiverRepElement, an element of self
• path – QuiverPath or list of tuples

OUTPUT:
• QuiverRepElement, the result of element*path when path is considered an element of the path algebra of the quiver

EXAMPLES:

```python
sage: Q = DiGraph({1:{2:['a','b']}, 2:{3:['c']}}).path_semigroup()
sage: M = Q.P(QQ, 1)
sage: v = M.an_element()
sage: v.support()
[1, 2, 3]
sage: M.right_edge_action(v, [(1, 1)]).support()
[1]
sage: M.right_edge_action(v, [(1, 1)]).support()
[1]
sage: M.right_edge_action(v, [(1, 2, 'a')]).support()
[2]
sage: M.right_edge_action(v, 'a') == M.right_edge_action(v, [(1, 2, 'a')])
True
```

socle()
The socle of self.

OUTPUT:
• QuiverRep, the socle

EXAMPLES:
sage: Q = DiGraph({1:{2:['a','b']}, 2:{3:['c']}}).path_semigroup()
sage: M = Q.P(QQ, 1)
sage: M.socle()
Representation with dimension vector (0, 0, 2)

submodule (elements=[], spaces=None)
Returns the submodule generated by the data.

INPUT:
• elements – a collection of QuiverRepElements (default: empty list), each should be an element of self
• spaces – dictionary (default: empty), this dictionary should contain entries of the form (v: S) where v is a vertex of the quiver and S is a subspace of the vector space associated to v

OUTPUT:
• QuiverRep, the smallest subrepresentation of self containing the given elements and the given subspaces

Note: This function returns only a QuiverRep object sub. The inclusion map of sub into M = self can be obtained by calling M.coerce_map_from(sub).

EXAMPLES:

sage: Q = DiGraph({1:{3:['a']}, 2:{3:['b']}}).path_semigroup()
sage: spaces = {1: QQ^2, 2: QQ^3, 3: QQ^2}
sage: maps = {(1, 3, 'a'): (QQ^2).Hom(QQ^2).identity(), (2, 3, 'b'): [[1, 0], \rightarrow [0, 0], [0, 0]]}
sage: M = Q.representation(QQ, spaces, maps)
sage: v = M.an_element()
sage: M.submodule([v])
Representation with dimension vector (1, 1, 1)

The smallest submodule containing the vector space at vertex 1 also contains the entire vector space associated to vertex 3 because there is an isomorphism associated to the edge (1, 3, 'a'):

sage: M.submodule(spaces={1: QQ^2})
Representation with dimension vector (2, 0, 2)

The smallest submodule containing the vector space at vertex 2 also contains the image of the rank 1 homomorphism associated to the edge (2, 3, 'b'):

sage: M.submodule(spaces={2: QQ^3})
Representation with dimension vector (0, 3, 1)

As v is not already contained in this submodule, adding it as a generator yields a larger submodule:

sage: v.support()
[1, 2, 3]
sage: M.submodule([v], {2: QQ^3})
Representation with dimension vector (1, 3, 1)

Giving no generating data yields the zero submodule:
sage: M.submodule().is_zero()
True

If the given data generates all of $M$ then the result is $M$:

sage: M.submodule(M.gens()) is M
True

**support()**

Return the support of `self` as a list.

**OUTPUT:**

- list, the vertices of the representation that have nonzero spaces associated to them

**EXAMPLES:**

```python
sage: Q = DiGraph({1:{2: ['a']}, 3:{2: ['b'], 4: ['c']}}).path_semigroup()
sage: M = Q.P(QQ, 3)
sage: M
Representation with dimension vector (0, 1, 1, 1)
sage: M.support()
[2, 3, 4]
```

**top()**

Return the top of `self`.

**OUTPUT:**

- QuiverRep, the quotient of `self` by its radical

**EXAMPLES:**

```python
sage: Q = DiGraph({1:{2: ['a', 'b']}, 2:{3: ['c']}}).path_semigroup()
sage: M = Q.P(QQ, 1)
sage: M
```

**transpose()**

Return the transpose of `self`.

The transpose, $\text{Tr} M$, of a module $M$ is defined as follows. Let $p : P_1 \to P_2$ be the second map in a minimal projective presentation $P_1 \to P_2 \to M \to 0$ of $M$. If $p^t$ is the algebraic dual of $p$ then define $\text{Tr} M = \text{coker} p^t$.

**OUTPUT:**

- QuiverRep

**EXAMPLES:**

```python
sage: Q = DiGraph({1:{2: ['a', 'b']}}).path_semigroup()
sage: M = Q.representation(GF(3), {1: 1, 2: 1}, {(1, 2, 'a'): 1})
sage: M.transpose()
```

**zero submodule()**

Return the zero submodule of `self`.
• QuiverRep, the zero submodule of self.

**EXAMPLES:**

```python
sage: Q = DiGraph({1:{2:['a','b']}, 2:{3:['c']}}).path_semigroup()
sage: M = Q.P(QQ, 1)
sage: M.zero_submodule()
Representation with dimension vector (0, 0, 0)
sage: M.zero_submodule().is_zero()
True
```

class sage.quivers.representation.QuiverRep_with_dual_path_basis(k, P, basis)

Bases: sage.quivers.representation.QuiverRep_generic

The basis of the module must be closed under left deletion of an edge: that is, deleting any edge from the beginning of any path in the basis must result in a path also contained in the basis of the module.

**INPUT:**

- k – ring; the base ring of the representation
- P – the path semigroup of the quiver Q of the representation
- basis – list (default: empty); should be a list of paths (also lists) in the quiver Q. Entries that do not represent valid paths are ignored and duplicate paths are deleted. The closure of this list under left deletion forms the basis of the resulting representation.

```python
class sage.quivers.representation.QuiverRep_with_path_basis(k, P, basis)

Bases: sage.quivers.representation.QuiverRep_generic

The basis of the module must be closed under right multiplication by an edge; that is, appending any edge to the end of any path in the basis must result in either an invalid path or a valid path also contained in the basis of the module.

**INPUT:**

- k – ring, the base ring of the representation
- P – the path semigroup of the quiver Q of the representation
- basis – list (default: empty); should be a list of paths (also lists) in the quiver Q. Entries that do not represent valid paths are ignored and duplicate paths are deleted. The closure of this list under right multiplication forms the basis of the resulting representation.

is_left_module()

Test whether the basis is closed under left multiplication.

**EXAMPLES:**

```python
sage: Q1 = DiGraph({1:{2:['a']}}).path_semigroup()
sage: P2 = Q1.representation(QQ, [[(2, 2)]]), option='paths')
sage: P2.is_left_module()
False
```

The supplied basis is not closed under left multiplication, but it’s not closed under right multiplication either. When the closure under right multiplication is taken the result is also closed under left multiplication and therefore produces a left module structure:

```python
sage: kQ = Q1.representation(QQ, [[(1, 1)], [(2, 2)]], option='paths')
sage: kQ.is_left_module()
True
```
Taking the right closure of a left closed set produces another left closed set:

```python
sage: Q2 = DiGraph({1:{2:['a'], 3:['b', 'c']}, 2:{3:['d']}}).path_semigroup()
sage: M = Q2.representation(QQ, [[(2, 2)], [(1, 2, 'a')]], option='paths')
sage: M.is_left_module()
True
```

Note that the second path is length 2, so even though the edge (1, 2, 'a') appears in the input the path [(1, 2, 'a')] is not in the right closure:

```python
sage: N = Q2.representation(QQ, [[(2, 2)], [(1, 2, 'a'), (2, 3, 'd')]], option='paths')
sage: N.is_left_module()
False
```
CHAPTER

EIGHT

INDICES AND TABLES

• Index
• Module Index
• Search Page
q
sage.quivers.algebra, 1
sage.quivers.algebra_elements, 7
sage.quivers.homspace, 15
sage.quivers.morphism, 19
sage.quivers.path_semigroup, 29
sage.quivers.paths, 39
sage.quivers.representation, 45
INDEX

A
actor() (sage.quivers.representation.QuiverRep_generic method), 58
algebra() (sage.quivers.path_semigroup.PathSemigroup method), 31
algebraic_dual() (sage.quivers.morphism.QuiverRepHom method), 20
algebraic_dual() (sage.quivers.representation.QuiverRep_generic method), 58
all_paths() (sage.quivers.path_semigroup.PathSemigroup method), 32
an_element() (sage.quivers.representation.QuiverRep_generic method), 58
AR_translate() (sage.quivers.representation.QuiverRep_generic method), 57
arrows() (sage.quivers.algebra.PathAlgebra method), 2
arrows() (sage.quivers.path_semigroup.PathSemigroup method), 33

B
base_ring() (sage.quivers.homspace.QuiverHomSpace method), 15
base_ring() (sage.quivers.morphism.QuiverRepHom method), 20

cardinality() (sage.quivers.path_semigroup.PathSemigroup method), 33
codomain() (sage.quivers.homspace.QuiverHomSpace method), 15
codomain() (sage.quivers.morphism.QuiverRepHom method), 21
coefficient() (sage.quivers.algebra_elements.PathAlgebraElement method), 8
coefficients() (sage.quivers.algebra_elements.PathAlgebraElement method), 9
cokernel() (sage.quivers.morphism.QuiverRepHom method), 21
complement() (sage.quivers.paths.QuiverPath method), 40
coordinates() (sage.quivers.homspace.QuiverHomSpace method), 16
coordinates() (sage.quivers.representation.QuiverRep_generic method), 58
copy() (sage.quivers.representation.QuiverRepElement method), 52
create_key() (sage.quivers.representation.QuiverRepFactory method), 55
create_object() (sage.quivers.representation.QuiverRepFactory method), 56

D
degree() (sage.quivers.algebra_elements.PathAlgebraElement method), 9
degree() (sage.quivers.paths.QuiverPath method), 40
degree_on_basis() (sage.quivers.algebra.PathAlgebra method), 2
dimension() (sage.quivers.homspace.QuiverHomSpace method), 16
dimension() (sage.quivers.representation.QuiverRep_generic method), 59
dimension_vector() (sage.quivers.representation.QuiverRep_generic method), 59
direct_sum() (sage.quivers.morphism.QuiverRepHom method), 21
direct_sum() (sage.quivers.representation.QuiverRep_generic method), 60
domain() (sage.quivers.homspace.QuiverHomSpace method), 16
domain() (sage.quivers.morphism.QuiverRepHom method), 22

E
Element (sage.quivers.algebra.PathAlgebra attribute), 2
Element (sage.quivers.homspace.QuiverHomSpace attribute), 15
Element (sage.quivers.path_semigroup.PathSemigroup attribute), 29
Element (sage.quivers.representation.QuiverRep_generic attribute), 57

F
free_module() (sage.quivers.path_semigroup.PathSemigroup method), 33

G
gcd() (sage.quivers.paths.QuiverPath method), 41
gen() (sage.quivers.algebra.PathAlgebra method), 3
gen() (sage.quivers.path_semigroup.PathSemigroup method), 34
gens() (sage.quivers.algebra.PathAlgebra method), 3
gens() (sage.quivers.homspace.QuiverHomSpace method), 16
gens() (sage.quivers.path_semigroup.PathSemigroup method), 34
gens() (sage.quivers.representation.QuiverRep_generic method), 61
get_element() (sage.quivers.representation.QuiverRepElement method), 52
get_map() (sage.quivers.morphism.QuiverRepHom method), 23
get_map() (sage.quivers.representation.QuiverRep_generic method), 62
get_matrix() (sage.quivers.morphism.QuiverRepHom method), 23
get_space() (sage.quivers.representation.QuiverRep_generic method), 62

H
has_prefix() (sage.quivers.paths.QuiverPath method), 41
has_subpath() (sage.quivers.paths.QuiverPath method), 42
Hom() (sage.quivers.representation.QuiverRep_generic method), 57
homogeneous_component() (sage.quivers.algebra_elements.PathAlgebraElement method), 3
homogeneous_components() (sage.quivers.algebra.PathAlgebra method), 4

I
I() (sage.quivers.path_semigroup.PathSemigroup method), 29
idempotents() (sage.quivers.algebra.PathAlgebra method), 4
idempotents() (sage.quivers.path_semigroup.PathSemigroup method), 34
image() (sage.quivers.morphism.QuiverRepHom method), 23
initial_vertex() (sage.quivers.paths.QuiverPath method), 42
injective() (sage.quivers.path_semigroup.PathSemigroup method), 35
is_endomorphism() (sage.quivers.morphism.QuiverRepHom method), 24
is_finite() (sage.quivers.path_semigroup.PathSemigroup method), 35
is_homogeneous() (sage.quivers.algebra_elements.PathAlgebraElement method), 9
is_injective() (sage.quivers.morphism.QuiverRepHom method), 24
is_isomorphism() (sage.quivers.morphism.QuiverRepHom method), 24
is_left_module() (sage.quivers.representation.QuiverRep_with_path_basis method), 68
is_semisimple() (sage.quivers.representation.QuiverRep_generic method), 62
is_simple() (sage.quivers.representation.QuiverRep_generic method), 62
is_surjective() (sage.quivers.morphism.QuiverRepHom method), 25
QuiverPath (class in sage.quivers.paths), 39
QuiverRep_generic (class in sage.quivers.representation), 56
QuiverRep_with_dual_path_basis (class in sage.quivers.representation), 68
QuiverRep_with_path_basis (class in sage.quivers.representation), 68
QuiverRepElement (class in sage.quivers.representation), 51
QuiverRepFactory (class in sage.quivers.representation), 54
QuiverRepHom (class in sage.quivers.morphism), 19
quotient() (sage.quivers.representation.QuiverRep_generic method), 64

R
radical() (sage.quivers.representation.QuiverRep_generic method), 65
rank() (sage.quivers.morphism.QuiverRepHom method), 27
representation() (sage.quivers.path_semigroup.PathSemigroup method), 37
reversal() (sage.quivers.paths.QuiverPath method), 42
reverse() (sage.quivers.path_semigroup.PathSemigroup method), 37
right_edge_action() (sage.quivers.representation.QuiverRep_generic method), 65

S
S() (sage.quivers.path_semigroup.PathSemigroup method), 30
sage.quivers.algebra (module), 1
sage.quivers.algebra_elements (module), 7
sage.quivers.homspace (module), 15
sage.quivers.morphism (module), 19
sage.quivers.path_semigroup (module), 29
sage.quivers.paths (module), 39
sage.quivers.representation (module), 45
scalar_mult() (sage.quivers.morphism.QuiverRepHom method), 27
semigroup() (sage.quivers.algebra.PathAlgebra method), 5
simple() (sage.quivers.path_semigroup.PathSemigroup method), 37
socle() (sage.quivers.representation.QuiverRep_generic method), 65
sort_by_vertices() (sage.quivers.algebra_elements.PathAlgebraElement method), 11
submodule() (sage.quivers.representation.QuiverRepgeneric method), 66
sum() (sage.quivers.algebra.PathAlgebra method), 5
support() (sage.quivers.algebra_elements.PathAlgebraElement method), 11
support() (sage.quivers.representation.QuiverRepgeneric method), 67
support() (sage.quivers.representation.QuiverRepElement method), 53
support_of_term() (sage.quivers.algebra_elements.PathAlgebraElement method), 12

T
terminal_vertex() (sage.quivers.paths.QuiverPath method), 43
terms() (sage.quivers.algebra_elements.PathAlgebraElement method), 12
top() (sage.quivers.representation.QuiverRep_generic method), 67
transpose() (sage.quivers.representation.QuiverRep_generic method), 67

Z
zero() (sage.quivers.homspace.QuiverHomSpace method), 18
zero_submodule() (sage.quivers.representation.QuiverRep_generic method), 67