aaa

Version 0.0.0

Collin Bleak
Fernando Flores Brito
Luke Elliott
Feyisayo Olukoya

Collin Bleak Email: cb211@st-andrews.ac.uk
Homepage: www-groups.mcs.st-and.ac.uk/~collin/

Abstract

The aaa package is a GAP package containing methods for transducers that can be represented as asynchronous automata. It implements the processes described in the paper by Grigorchuk, Nekrashevich, and Sushchanskii [GNS00] for working with finite transducers and the rational group R.

Copyright

Acknowledgements

The current package is a complete rewrite of the original AAA package that was developed by Plamena Mineva-Sholz, Angela Richardson, and Ieva Vasiljeva under the supervision of Collin Bleak. This version has been structured following the Semigroups [M+18] package, a very useful recommendation of J. D. Mitchell, and contains all the functionality of the previous version, and more.

Special thanks to Michael Torpey and Wilf Wilson for their continuous support. Their ample knowledge about GAP and programming has made the development of this package run smoothly.

Manuel Delgado and Attila Egri-Nagy contributed to the function `Splash` (6.1.1). The function `DotTransducer` (6.2.1), written by Michael Torpey, is based on the Digraphs [DBJM+18] package function they wrote called DotString.

Contents

1 aaa package **4**
- 1.1 The aaa package and the rational group . . . 4
- 1.2 Examples . . . 4
- 1.3 Main Functionality . . . 5

2 Installing aaa **6**
- 2.1 Installation . . . 6
- 2.2 Rebuilding the documentation . . . 6

3 Transducers **7**
- 3.1 Creating a transducer . . . 7
- 3.2 Attributes of Transducers . . . 9

4 Words accepted by transducers and word operations **11**
- 4.1 Words and operations . . . 11

5 Transducer operations **13**
- 5.1 Transducer operations . . . 13

6 Visualising Transducers **16**
- 6.1 Automatic viewing . . . 16
- 6.2 `dot` pictures . . . 16

References **18**

Index **19**

Chapter 1

aaa package

1.1 The aaa package and the rational group

This package grew out of a desire to be able to work efficiently with elements of the asychronous rational group as described in the paper [GNS00].

Note that there are already two related packages in GAP. The first package is called "fr" by Laurent Bartholdi and the second package is AutomGrp by Yevgen Muntyan and Dmytro Savchuk. The package "fr" is meant to work with groups geneated by synchonous automata and more generally functionally recursive groups. The AutomGrp package works with automata as described in [GNS00] but these automata are also constrained to be synchronous (one letter in, one letter out).

The paper [GNS00] actually describes the group R of homeomorphisms of the binary Cantor space which are describable by finite initial asynchronous automata (the fact that the Cantor space is binary is not relevent). Groups generated by such homeomorphisms form a much broader class of groups; the group R contains many of the groups of homeomorphisms of Cantor space which are studied in the literature, such as the Thompson groups F, T, V, Grigorchuk's group, and hybrids such as Roever's group and other Nekrashevich type groups.

The paper [GNS00] gives several algorithms for performing calculations in these groups and this package implements all of that functionality although occasionally using different underlying algorithms. The names of various algorithms are natural from the text of the paper as one can see in this manual.

This package is built over the framework provided by the automata package in GAP. TODO insert link

1.2 Examples

TODO add picture

This is an example of a transducer representing a homeomorphism on the two letter alphabet Cantor space. One reads singley infinite strings starting from state 1 taking appropriate transitions on the transducer pictured. Each transitional arrow has a label of the form "digit/(finite word)" one uses the digits to discern which transitions to follows and replaces each such digit with the word under the slash. All transducers representing homeomorphisms have yhe property that cycles always write non-empty input. Thus the total output is the infinite string of digits written by the infinite concatenation of these substitution words.

For example the string "101011..." becomes "".

1.3 Main Functionality

Our transducer package allows users to write transducers which change alphabet sizes, and do generic transducer-type transformations to finite strings. Given a base initial transducer T with identital input and output alphabets, the package can verify that T represents an invertible homeomorphism h of Cantor space. If h is invertible the package supports inversion (by producing a transducer S representing the inverse of h). It also supports taking products and supports the minimisation and checking for incomplete response procedures as described in the GNS paper. More detailed technical processes are also supported as one can see by reading further into this manual. Here is a list of the functions for transducers present in the package:

"Transducer", "TransducerFunction","OutputFunction",
"TransitionFunction", "InputAlphabet", "OutputAlphabet",
"States", "NrStates", "NrOutputSymbols",
"NrInputSymbols", "IdentityTransducer", "AlphabetChangeTransducer",
"RandomTransducer", "InverseTransducer", "TransducerProduct",
"RemoveStatesWithIncompleteResponse","RemoveInaccessibleStates",
"CopyTransducerWithInitialState",
"IsInjectiveTransducer", "IsSurjectiveTransducer", "IsBijectiveTransducer",
"TransducerImageAutomaton", "TransducerConstantStateOutputs", "IsDegenerateTransducer",
"IsMinimalTransducer", "CombineEquivalentStates", "MinimalTransducer",
"IsSynchronousTransducer", "TransducerOrder", "IsomorphicInitialTransducers",
"OmegaEquivalentTransducers", "EqualTransducers", "FixedOutputDigraph". ~
~

Chapter 2

Installing aaa

2.1 Installation

All you need to do is place the aaa package in the `pkg` directory of your GAP installation and load the package by typing `LoadPackage("aaa")` in your GAP session.

2.2 Rebuilding the documentation

Should you need to rebuild the documentation, use the `AaaMakeDoc` (2.2.1) function.

2.2.1 AaaMakeDoc

▷ `AaaMakeDoc()` (function)

Returns: Nothing.

This function should be called with no argument to compile the aaa documentation.

Example

```
gap> AaaMakeDoc();
```

Chapter 3

Transducers

In this chapter we decribe how to create an aaa transducer.

3.1 Creating a transducer

The aaa package provides a method to represent transducers in GAP.

3.1.1 Transducer

▷ `Transducer(`*m*`, `*n*`, `*P*`, `*L*`)` (operation)

Returns: A transducer.

For two positive integers *m*, and *n*, a dense list of dense lists of integers *P*, and a dense list of dense lists of dense lists of integers *L*, such that *P* and *L* have the same size, and each of their elements have size equal to *m*, the operation `Transducer(`*m*`, `*n*`, `*P*`, `*L*`)` returns a transducer with input alphabet `[0 .. m - 1]`, output alphabet `[0 .. n - 1]`, transition function *P*, and output function *L*. If `p` abstractly represents the transition function of the transducer, then *P* has the property that `p(currentstate, inputletter - 1) = `*P*`[currentstate][inputletter]`, where `currentstate` is an integer from `[1 .. Size(`*P*`)]`, and `inputletter` is a positive integer representing the word `[inputletter - 1]`. Similarly, if `l` abstractly represents the output function of the transducer, then *L* has the property that `l(currentstate, inputletter - 1) = `*L*`[currentstate][inputletter]`, where `currentstate`, and `inputletter` are as before.

Example

```
gap> T := Transducer(2, 2, [[2, 3], [2, 3], [2, 3]],
>                          [[[], []], [[0], [0]], [[1], [1]]]);
<transducer with input alphabet on 2 symbols, output alphabet on 2 symbols,
  and 3 states.>
```

3.1.2 TransducerFunction

▷ `TransducerFunction(`*T*`, `*word*`, `*m*`)` (operation)

Returns: A list.

For a transducer *T*, a dense list *word*, and a positive integer *m* such that it is a state of the transducer, the operation `TransducerFunction(`*T*`, `*word*`, `*m*`)` returns a list containing the word obtained when *word* is read by *T* from state *m*, and the state that is reached after reading *word* from state *m*.

Example

```
gap> T := Transducer(2, 2, [[1, 2], [2, 1]], [[[0], []], [[1], [0, 1]]]);;
gap> TransducerFunction(T, [0, 1, 0], 1);
[ [ 0, 1 ], 2 ]
```

3.1.3 XFunction

▷ `TransitionFunction(`*T*`)` (operation)

▷ `OutputFunction(`*T*`)` (operation)

Returns: A list.

For a transducer *T*, the operation `TransitionFunction(`*T*`)` displays the list representing the transition function of *T*, and the operation `OutputFunction(`*T*`)` displays the list representing the output function of *T*.

Example

```
gap> T := Transducer(2, 2, [[1, 2], [2, 1]], [[[0], []], [[1], [0, 1]]]);;
gap> TransitionFunction(T);
[ [ 1, 2 ], [ 2, 1 ] ]
gap> OutputFunction(T);
[ [ [ 0 ], [  ] ], [ [ 1 ], [ 0, 1 ] ] ]
```

3.1.4 XAlphabet

▷ `InputAlphabet(`*T*`)` (operation)

▷ `OutputAlphabet(`*T*`)` (operation)

Returns: A list.

For a transducer *T*, the operation `InputAlphabet(`*T*`)` returns the list representing the input alphabet of *T*, and the operation `OutputAlphabet(`*T*`)` returns the list representing the output alphabet of *T*.

Example

```
gap> P := Transducer(2, 3, [[1, 2], [2, 1]], [[[0], [2]], [[1], [0, 1]]]);;
gap> InputAlphabet(P);
[ 0, 1 ]
gap> OutputAlphabet(P);
[ 0 .. 2 ]
```

3.1.5 NrXSymbols

▷ `NrInputSymbols(`*T*`)` (operation)

▷ `NrOutputSymbols(`*T*`)` (operation)

Returns: A positive integer.

For a transducer *T*, the operation `NrInputSymbols(`*T*`)` returns the number of symbols of the input alphabet of *T*, and the operation `NrOutputSymbols(`*T*`)` returns the number of symbols of the output alphabet of *T*.

Example

```
gap> T := Transducer(2, 3, [[1, 2], [2, 1]], [[[0], []], [[1], [2, 1]]]);;
gap> NrInputSymbols(T); NrOutputSymbols(T);
2
3
```

3.1.6 States

▷ `States(`*T*`)` (operation)

Returns: A list of integers.

For a transducer *T*, the operation `States(`*T*`)` returns a list representing the set of states of the transducer *T*.

Example

```
gap> T := Transducer(2, 2, [[1, 2], [2, 1]], [[[0], []], [[1], [0, 1]]]);;
gap> States(T);
[ 1, 2 ]
```

3.1.7 NrStates

▷ `NrStates(`*T*`)` (operation)

Returns: A positive integer.

For a transducer *T*, the operation `NrStates(`*T*`)` returns the number of states of the transducer *T*.

Example

```
gap> T := Transducer(2, 2, [[1, 2], [2, 1]], [[[0], []], [[1], [0, 1]]]);;
gap> NrStates(T);
2
```

3.2 Attributes of Transducers

In this section we explain the different transducer attributes that the aaa package can compute.

3.2.1 IsInjectiveTransducer

▷ `IsInjectiveTransducer(`*T*`)` (attribute)

Returns: `true` or `false`.

For a transducer *T*, with initial state 1, the attribute `IsInjectiveTransducer(`*T*`)` returns `true` if *T* is an injective transducer, and `false` otherwise.

Example

```
gap> T := Transducer(2, 2, [[2, 4], [3, 6], [3, 2], [5, 7], [5, 4],
>      [6, 6], [7, 7]], [[[0], []], [[0, 1], [1, 0, 1]], [[1, 1, 1],
>      [1, 0]], [[0, 0], [0, 1, 0]], [[0, 0, 0], [1, 1]], [[0], [1]],
>      [[0], [1]]]);;
gap> IsInjectiveTransducer(T);
false
gap> f := Transducer(3, 3, [[1, 1, 2], [1, 3, 2], [1, 1, 2]], [[[2],
>      [0], [1]], [[0, 0], [], [1]], [[0, 2], [2], [0, 1]]]);;
gap> IsInjectiveTransducer(f);
true
```

3.2.2 IsSurjectiveTransducer

▷ `IsSurjectiveTransducer(`*T*`)` (attribute)

Returns: `true` or `false`.

For a transducer *T*, with initial state 1, the attribute `IsSurjectiveTransducer(`*T*`)` returns `true` if *T* is a surjective transducer, and `false` otherwise.

Example

```
gap> f := Transducer(3, 3, [[1, 1, 2], [1, 3, 2], [1, 1, 2]], [[[2],
>      [0], [1]], [[0, 0], [], [1]], [[0, 2], [2], [0, 1]]]);;
gap> IsSurjectiveTransducer(f);
true
```

Chapter 4

Words accepted by transducers and word operations

In this chapter we decribe the words that are accepted by an aaa transducer, as well as operations that can be performed to them.

4.1 Words and operations

An aaa word is a dense list of integers.

4.1.1 IsPrefix

▷ `IsPrefix(u, v)` (operation)

Returns: `true` or `false`.

A word `p` is a prefix of the word `w` if `w = pq` for some word `q`. The operation `IsPrefix(u, v)` returns `true` if the word v is a prefix of the word u, and `false` otherwise.

Example

```
gap> u := [0, 1];; v := [0, 1, 0];; empty := [];;
gap> IsPrefix(u, v); IsPrefix(v, u); IsPrefix(u, empty);
false
true
true
```

4.1.2 Minus

▷ `Minus(u, v)` (operation)

Returns: A dense list or `fail`.

If v is a prefix of u, then $u = vm$ for some word `m`, and the operation `Minus(u, v)` returns `m`. Otherwise, it returns `fail`.

Example

```
gap> u := [0, 1];; v := [0, 1, 1];;
gap> Minus(v, u);
[ 1 ]
gap> Minus(u, v);
fail
```

4.1.3 GreatestCommonPrefix

▷ GreatestCommonPrefix(*list*) (operation)

Returns: A dense list.

If *list* is a list of words, the operation GreatestCommonPrefix(*list*) returns the greatest common prefix of all the words in *list*.

Example

```
gap> u := [0, 0];; v := [0, 0, 1, 2];; w := [0, 1, 2];;
gap> GreatestCommonPrefix([u, v, w]);
[ 0 ]
gap> empty := [];; z := [1];;
gap> GreatestCommonPrefix([u, v, w, empty]);
[  ]
gap> GreatestCommonPrefix([u, v, w, z]);
[  ]
```

4.1.4 PreimageConePrefixes

▷ PreimageConePrefixes(*u*, *s*, *T*) (operation)

Returns: A list.

If *u* is a word with letters from the output alphabet of the transducer *T*, and *s* is a positive integer, the operation PreimageConePrefixes(*u*, *s*, *T*) returns the list P of prefixes such that if w = pq for some p in P, and some other word q, then the image of w when read by *T* from state *s* is *u*v for some word v. This operation is meant to emulate the inverses of the functions f_q discussed in the last two paragraphs of p.142 in [GNS00].

Example

```
gap> f := Transducer(3, 3, [[1, 1, 2], [1, 3, 2], [1, 1, 2]], [[[2], [0], [1]],
>                      [[0, 0], [], [1]], [[0, 2], [2], [0, 1]]]);;
gap> PreimageConePrefixes([0], 2, f);
[ [ 0 ], [ 1, 0 ], [ 1, 2 ] ]
gap> TransducerFunction(f, [0], 2); TransducerFunction(f, [1, 0], 2);
[ [ 0, 0 ], 1 ]
[ [ 0, 2 ], 1 ]
gap> TransducerFunction(f, [1, 2], 2);
[ [ 0, 1 ], 2 ]
```

4.1.5 ImageConeLongestPrefix

▷ ImageConeLongestPrefix(*u*, *s*, *T*) (operation)

Returns: A list.

If *u* is a word with letters from the input alphabet of the transducer *T*, and *s* is a positive integer, the operation ImageConeLongestPrefix(*u*, *s*, *T*) returns the greatest common prefix of the set { L(*u*v, *s*) | v is word with letters in I} where L and I abstractly represent the output function and input alphabet of *T*, respectively.

Example

```
gap> t := Transducer(3, 3, [[1, 1, 2], [1, 3, 2], [1, 1, 2]], [[[2], [0], []],
>                           [[1, 0, 0], [1], [1]], [[0, 2], [2], [0]]]);;
gap> ImageConeLongestPrefix([], 2, t);
[ 1 ]
```

Chapter 5

Transducer operations

In this chapter we decribe the methods that are available in the aaa package to perform operations on aaa transducers. These methods are implementations of the algorithms introduced in [GNS00].

5.1 Transducer operations

The following are the methods that can be used to analyze aaa transducers.

5.1.1 InverseTransducer

▷ `InverseTransducer(`*T*`)` (operation)

Returns: A transducer.

For an invertible transducer *T* whose first state is a homeomorphism state, the operation `InverseTransducer(`*T*`)` returns the inverse of *T*. Please note that it is the user's responsibility to ensure that the transducer *T* is both invertible and that its first state is a homeomorphism state.

Example

```
gap> f := Transducer(3, 3, [[1, 1, 2], [1, 3, 2], [1, 1, 2]], [[[2], [0], [1]],
>                      [[0, 0], [], [1]], [[0, 2], [2], [0, 1]]]);;
gap> g := InverseTransducer(f);;
gap> w := TransducerFunction(f, [0, 1], 1)[1];
[ 2, 0 ]
gap> TransducerFunction(g, w, 1)[1];
[ 0, 1 ]
```

5.1.2 TransducerProduct

▷ `TransducerProduct(`*T*`, `*P*`)` (operation)

Returns: A transducer.

For two transducers *T* and *P*, the operation `TransducerProduct(`*T*`, `*P*`)` returns the product of the transducers *T* and *P*.

Example

```
gap> f := Transducer(3, 3, [[1, 1, 2], [1, 3, 2], [1, 1, 2]], [[[2], [0], [1]],
>                      [[0, 0], [], [1]], [[0, 2], [2], [0, 1]]]);
<transducer with input alphabet on 3 symbols, output alphabet on 3 symbols,
  and 3 states.>
gap> ff := TransducerProduct(f, f);
```

```
<transducer with input alphabet on 3 symbols, output alphabet on 3 symbols,
  and 9 states.>
```

5.1.3 RemoveStatesWithIncompleteResponse

▷ RemoveStatesWithIncompleteResponse(*T*) (operation)

Returns: A transducer.

For a transducer *T* that has states with incomplete response, the operation RemoveStatesWithIncompleteResponse(*T*) returns a transducer P that has one more state than *T*(acting as the new initial state) and which has no states with incomplete response. State s of the transducer *T* is state s + 1 of the transducer P.

Example

```
gap> t := Transducer(3, 3, [[1, 1, 2], [1, 3, 2], [1, 1, 2]], [[[2], [0], []],
>                          [[1, 0, 0], [1], [1]], [[0, 2], [2], [0]]]);
<transducer with input alphabet on 3 symbols, output alphabet on 3 symbols,
  and 3 states.>
gap> p := RemoveStatesWithIncompleteResponse(t);
<transducer with input alphabet on 3 symbols, output alphabet on 3 symbols,
  and 4 states.>
gap> TransducerFunction(t, [2], 1)[1]; TransducerFunction(t, [1], 2)[1];
[  ]
[ 1 ]
gap> TransducerFunction(p, [2], 2)[1];
[ 1 ]
```

5.1.4 RemoveInaccessibleStates

▷ RemoveInaccessibleStates(*T*) (operation)

Returns: A transducer.

For a transducer *T*, the operation RemoveInaccessibleStates(*T*) returns the transducer that is obtained by removing the states that are not accesssible from state 1.

Example

```
gap> f := Transducer(3, 3, [[1, 1, 2], [1, 3, 2], [1, 1, 2]], [[[2], [0], [1]],
>                     [[0, 0], [], [1]], [[0, 2], [2], [0, 1]]]);
<transducer with input alphabet on 3 symbols, output alphabet on 3 symbols,
  and 3 states.>
gap> ff := TransducerProduct(f, f);
<transducer with input alphabet on 3 symbols, output alphabet on 3 symbols,
  and 9 states.>
gap> m := RemoveInaccessibleStates(ff);
<transducer with input alphabet on 3 symbols, output alphabet on 3 symbols,
  and 6 states.>
```

5.1.5 CombineEquivalentStates

▷ CombineEquivalentStates(*T*) (operation)

Returns: A transducer.

For a transducer *T*, the operation CombineEquivalentStates(*T*) returns the transducer that is obtained by identifying states from which all finite words write the same word.

Example

```
gap> T := Transducer(2, 2, [[1, 3], [2, 1], [1, 1], [1, 3]], [[[0], [0]],
>                          [[1, 1], [0]], [[0], [0, 1]], [[0], [0]]]);
<transducer with input alphabet on 2 symbols, output alphabet on 2 symbols,
and 4 states.>
gap> T2 := CombineEquivalentStates(T);
<transducer with input alphabet on 2 symbols, output alphabet on 2 symbols,
and 3 states.>
gap> OutputFunction(T);
[ [ [ 0 ], [ 0 ] ], [ [ 1, 1 ], [ 0 ] ], [ [ 0 ], [ 0, 1 ] ], [ [ 0 ], [ 0 ] ] ]
gap> TransitionFunction(T);
[ [ 1, 3 ], [ 2, 1 ], [ 1, 1 ], [ 1, 3 ] ]
```

5.1.6 MinimalTransducer

▷ `MinimalTransducer(`*T*`)` (operation)

Returns: A transducer.

Every transducer has a minimal omega-equivalent form (this transducer produces the same outputs on all infinite length inputs as the original). One arrives at this form by first removing inaccessible states, then removing incomplete response from all states, and finally combining equivalent states. Those three operations are described above. For a transducer *T*, the operation `MinimalTransducer(`*T*`)` returns the transducer's minimal omega-equivalent form.

Example

```
gap> T := Transducer(2, 2, [[1, 3], [2, 1], [1, 1], [1, 3]], [[[0], [0]],
>                          [[1, 1], [0]], [[0], [0, 1]], [[0], [0]]]);
<transducer with input alphabet on 2 symbols, output alphabet on 2 symbols,
and 4 states.>
gap> M := MinimalTransducer(T);
<transducer with input alphabet on 2 symbols, output alphabet on 2 symbols,
and 3 states.>
gap> OutputFunction(M);
[ [ [ 0, 0, 0 ], [ 0, 0 ] ], [ [ 0 ], [  ] ], [ [ 0, 0 ], [ 1, 0, 0 ] ] ]
gap> TransitionFunction(M);
[ [ 2, 3 ], [ 2, 3 ], [ 2, 2 ] ]
```

5.1.7 CopyTransducerWithInitialState

▷ `CopyTransducerWithInitialState(`*T*`, `*m*`)` (operation)

Returns: A transducer.

For a transducer *T* and a positive integer *m*, the operation `CopyTransducerWithInitialState(`*T*`, `*m*`)` returns the transducer that is obtained by relabeling the states of *T* such that state *m* becomes state 1, but is otherwise equivalent to *T*.

Example

```
gap> f := Transducer(3, 3, [[1, 1, 2], [1, 3, 2], [1, 1, 2]], [[[2], [0], [1]],
>                       [[0, 0], [], [1]], [[0, 2], [2], [0, 1]]]);;
gap> p := CopyTransducerWithInitialState(f, 3);;
gap> TransducerFunction(f, [0, 1, 0], 3);
[ [ 0, 2, 0, 2 ], 1 ]
gap> TransducerFunction(p, [0, 1, 0], 1);
[ [ 0, 2, 0, 2 ], 2 ]
```

Chapter 6

Visualising Transducers

We provide the Splash (6.1.1) function written by A. Egry-Nay.

6.1 Automatic viewing

The following are the methods that can be used to display aaa transducers.

6.1.1 Splash

▷ Splash(*str*) (function)

Returns: Nothing.

This function attempts to convert the string *str* into a pdf document and open this document, i.e. to splash it all over your monitor.

The string *str* must correspond to a valid dot text file and you must have have GraphViz and pdflatex installed on your computer. For details about these file formats, see http://www.latex-project.org and http://www.graphviz.org.

This function is provided to allow convenient, immediate viewing of the pictures produced by DotTransducer (6.2.1).

This function was written by Attila Egri-Nagy and Manuel Delgado with some minor changes by J. D. Mitchell.

Example

```
gap> T := Transducer(2, 2, [[1, 2], [2, 1]], [[[0], []], [[1], [0, 1]]]);;
gap> Splash(DotTransducer(T));
```

6.2 dot pictures

In this section, we describe the operations in aaa for creating pictures in dot format.

The operations described in this section return strings, which can be viewed using Splash (6.1.1).

6.2.1 DotTransducer

▷ DotTransducer(*T*) (operation)

Returns: A string.

For a transducer T, this operation produces a graphical representation of the transducer as an automaton. The output is in `dot` format (also known as `GraphViz`) format. For details about this file format, and information about how to display or edit this format see http://www.graphviz.org.

Example

```
gap> T := Transducer(2, 2, [[1, 2], [2, 1]], [[[0], []], [[1], [0, 1]]]);;
gap> DotTransducer(T);
"//dot\n
digraph finite_state_machine{\n
rankdir=LR;\n
node [shape=circle]\n
1\n
2\n
1 -> 1 [label=\"0|0\"]\n
1 -> 2 [label=\"1|\"]\n
2 -> 2 [label=\"0|1\"]\n
2 -> 1 [label=\"1|01\"]\n
}\n"
```

References

[DBJM+18] J. De Beule, J. Jonusas, J. D. Mitchell, M. Torpey, and W. Wilson. *Digraphs - GAP package, Version 0.12.1*, Apr 2018. 2

[GNS00] R. I. Grigorchuk, V. V. Nekrashevich, and V. I. Sushchanskii. Automata, Dynamical Systems, and Groups. *Proceedings of the Steklov Institute of Mathematics*, 231:128–203, 2000. 2, 4, 12, 13

[M+18] J. D. Mitchell et al. *Semigroups - GAP package, Version 3.0.15*, Mar 2018. 2

Index

AaaMakeDoc, 6

CombineEquivalentStates, 14
CopyTransducerWithInitialState, 15

DotTransducer, 16

GreatestCommonPrefix, 12

ImageConeLongestPrefix, 12
InputAlphabet, 8
InverseTransducer, 13
IsInjectiveTransducer, 9
IsPrefix, 11
IsSurjectiveTransducer, 9

MinimalTransducer, 15
Minus, 11

NrInputSymbols, 8
NrOutputSymbols, 8
NrStates, 9

OutputAlphabet, 8
OutputFunction, 8

PreimageConePrefixes, 12

RemoveInaccessibleStates, 14
RemoveStatesWithIncompleteResponse, 14

Splash, 16
States, 9

Transducer, 7
TransducerFunction, 7
TransducerProduct, 13
TransitionFunction, 8