PatternClass

Permutation Pattern Classes

Version 1.1235813213455

August 2015

Michael Albert
Steve Linton
Ruth Hoffmann

Michael Albert Email: malbert at cs.otago.ac.nz

Steve Linton Email: sl4 at st-andrews.ac.uk

Ruth Hoffmann Email: rh347 at icloud.com

Abstract

The `PatternClass` package is build on the idea of token passing networks building permutation pattern classes. Those classes are best determined by their basis. Both sets can be encoded by rank encoding their permutations. Each, the class and its basis, in their encoded form build a rational language [1]. Rational languages can be easily computed by using automata, which also can be build directly from the token passing networks [2]. Both ways will build the same language, i.e. the same automaton.

Copyright

Contents

Chapter 1

Introduction

Token passing networks (TPNs) are directed graphs with nodes that can hold at most one token. Also each graph has a designated input node, which generates an ordered sequence of numbered tokens and a designated output node that collects the tokens in the order they arrive at it. The input node has no incoming edges, whereas the output node has no outgoing edges. A token t travels through the graph, from node to node, if there is an edge connecting the nodes, if the node the token is moving from is either the input node and the tokens $1,\ldots,t-1$ have been released or the node is not the output node, and lastly if the destination node contains no token or it is the output node. [2]

The set of permutations resulting from a TPN is closed under the property of containment. A permutation a contains a permutation b of shorter length if in a there is a subsequence that is isomorphic to b. This class of permutations can be represented by its anti-chain, which in this context is called the basis. [1]

To enhance the computability of permutation pattern classes, each permutation can be encoded, using the so called rank encoding. For a permutation $p_1 \ldots p_n$, it is the sequence $e_1 \ldots e_n$ where e_i is the rank of p_i among $\{p_i, p_{i+1}, \ldots, p_n\}$. It can be shown that the sets of encoded permutations of the class and the basis, both are a rational languages. Rational languages can be represented by automata. [1]

There is another approach to get from TPNs to their corresponding automata. Namely building equivalence classes from TPNs using the different dispositions of tokens within them. These equivalence classes of dispositions and the rank encoding of the permutations allow to build the same rational language as from the process above. [2]

Chapter 2

Token Passing Networks

A token passing network is a directed graph with a designated input node and a designated output node. The input node has no incoming edges whereas the output node has no outgoing edges. Also the input node generates a sequence of tokens, labelled 1, 2, 3, ... These tokens are passed on to the nodes within the graph, where each node, apart from the input and output node, can hold at most one token at any time. The edges do not hold tokens but are there to pass them on. The following must hold if a token t moves from the node x to the node y.

There is an edge from x to y; x is the input node, and the tokens 1, 2, 3, ... , $t-1$ have been moved, or x is any other node but not the output node; lastly either y is the output node or y is not the input node and currently is not occupied by a token. [2]

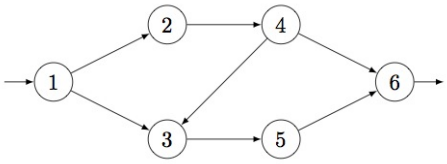
Token passing networks, or TPNs, are represented in GAP as a list. Each entry of the list is the index of the node within the TPN and contains a list of the "destinations", i.e. the end of the edge or arrow where it is directed to.

Here is an example how the input of such a TPN looks in GAP:

Example

```
gap> hex:=[[2,3],[4],[5],[3,6],[6],[]];
[ [ 2, 3 ], [ 4 ], [ 5 ], [ 3, 6 ], [ 6 ], [  ] ]
gap>
```

This list represents the following directed graph:



From this it is visible that the input node is 1, as it has no other node pointing any arrows towards it, and the output node is 6, as it has no destinations (hence the empty list).

2.1 Specific TPN

In `PatternClass` there are several functions that define different kinds of specific token passing networks.

2.1.1 Parstacks

▷ `Parstacks(m, n)` (function)

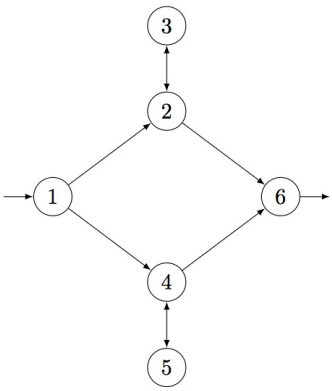
Returns: A list that represents the directed edges of a token passing network.

`Parstacks` constructs a token passing network with 2 different sized stacks `m,n` positioned in parallel.

Example

```
gap> Parstacks(2,2);
[ [ 2, 4 ], [ 3, 6 ], [ 2 ], [ 5, 6 ], [ 4 ], [  ] ]
gap>
```

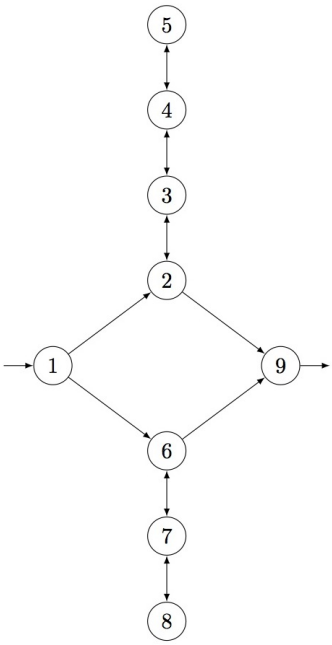
`Parstacks(2,2)` can be visualised as the following directed graph:



Example

```
gap> Parstacks(4,3);
[ [ 2, 6 ], [ 3, 9 ], [ 2, 4 ], [ 3, 5 ], [ 4 ], [ 7, 9 ], [ 6, 8 ], [ 7 ],
  [  ] ]
gap>
```

The token passing network below represents the list that was output by `Parstacks(4,3)`.



2.1.2 Seqstacks

▷ `Seqstacks(`*`n[, m[, o[, p[, ...]]]]`*`)` (function)

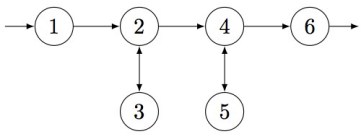
Returns: A list that represents the directed edges of a token passing network.

The token passing network build by `Seqstacks` contains a series of stacks (as many as you have integers in the arguments list) each of different length (each integer in the argument list).

Example

```
gap> Seqstacks(2,2);
[ [ 2 ], [ 3, 4 ], [ 2 ], [ 5, 6 ], [ 4 ], [  ] ]
gap>
```

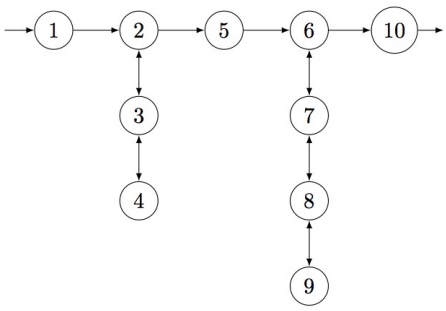
`Seqstacks(2,2)` can be visualised as the following directed graph:



Example

```
gap> Seqstacks(3,1,4);
[ [ 2 ], [ 3, 5 ], [ 2, 4 ], [ 3 ], [ 4 ], [ 7, 10 ], [ 6, 8 ], [ 7, 9 ],
  [ 8 ], [  ] ]
gap>
```

The token passing network containing a series of stacks of length 3, 1 and 4 looks as follows:



2.1.3 BufferAndStack

▷ `BufferAndStack(m, n)` (function)

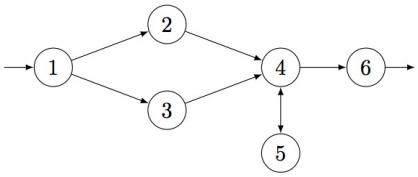
Returns: A list that represents the directed edges of a token passing network.

`BufferAndStack` is a token passing network that consists of a buffer of size `m` which is followed by a single stack of size `n`.

Example

```
gap> BufferAndStack(2,2);
[ [ 2, 3 ], [ 4 ], [ 4 ], [ 5, 6 ], [ 4 ], [  ] ]
gap>
```

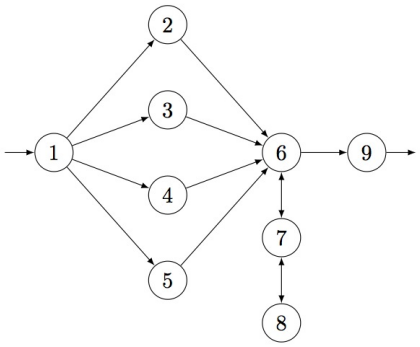
`BufferAndStack(2,2)` is the following directed graph:



Example

```
gap> BufferAndStack(4,3);
[ [ 2 .. 5 ], [ 6 ], [ 6 ], [ 6 ], [ 6 ], [ 7, 9 ], [ 6, 8 ], [ 7 ], [  ] ]
gap>
```

The token passing network correlating to the list output by `BufferAndStack(4,3)` is:



Chapter 3

Permutation Encoding

A permutation $\pi = \pi_1 \dots \pi_n$ has rank encoding $p_1 \dots p_n$ where $p_i = |\{j : j \geq i, \pi_j \leq \pi_i\}|$. In other words the rank encoded permutation is a sequence of p_i with $1 \leq i \leq n$, where p_i is the rank of π_i in $\{\pi_i, \pi_{i+1}, \dots, \pi_n\}$. [1]

The encoding of the permutation 3 2 5 1 6 7 4 8 9 is done as follows:

Permutation	Encoding	Assisting list
325167489	$\emptyset$	123456789
25167489	3	12456789
5167489	32	1456789
167489	323	146789
67489	3231	46789
7489	32312	4789
489	323122	489
89	3231221	89
9	32312211	9
$\emptyset$	323122111	$\emptyset$

Decoding a permutation is done in a similar fashion, taking the sequence $p_1 \dots p_n$ and using the reverse process will lead to the permutation $\pi = \pi_1 \dots \pi_n$, where π_i is determined by finding the number that has rank p_i in $\{\pi_i, \pi_{i+1}, \dots, \pi_n\}$.

The sequence 3 2 3 1 2 2 1 1 1 is decoded as:

Encoding	Permutation	Assisting list
323122111	$\emptyset$	123456789
23122111	3	12456789
3122111	32	1456789
122111	325	146789
22111	3251	46789
2111	32516	4789
111	325167	489
11	3251674	89
1	32516748	9
$\emptyset$	325167489	$\emptyset$

3.1 Encoding and Decoding

3.1.1 RankEncoding

▷ RankEncoding(*p*) (function)

Returns: A list that represents the rank encoding of the permutation p.

Using the algorithm above RankEncoding turns the permutation p into a list of integers.

Example

```
gap> RankEncoding([3, 2, 5, 1, 6, 7, 4, 8, 9]);
[ 3, 2, 3, 1, 2, 2, 1, 1, 1 ]
gap> RankEncoding([ 4, 2, 3, 5, 1 ]);
[ 4, 2, 2, 2, 1 ]
gap>
```

3.1.2 RankDecoding

▷ RankDecoding(*e*) (function)

Returns: A permutation in list form.

A rank encoded permutation is decoded by using the reversed process from encoding, which is also explained above.

Example

```
gap> RankDecoding([ 3, 2, 3, 1, 2, 2, 1, 1, 1 ]);
[ 3, 2, 5, 1, 6, 7, 4, 8, 9 ]
gap> RankDecoding([ 4, 2, 2, 2, 1 ]);
[ 4, 2, 3, 5, 1 ]
gap>
```

3.1.3 SequencesToRatExp

▷ SequencesToRatExp(*list*) (function)

Returns: A rational expression that describes all the words in list.

A list of sequences is turned into a rational expression by concatenating each sequence and unifying all of them.

Example

```
gap> SequencesToRatExp([[ 1, 1, 1, 1, 1 ],[ 2, 1, 2, 2, 1 ],[ 3, 2, 1, 2, 1 ],
> [ 4, 2, 3, 2, 1 ]]);
11111U21221U32121U42321
gap>
```

Chapter 4

From Networks to Automata

It is known that the language of all encoded permutations of a TPN is rational, and thus is it indeed possible to turn a TPN into an automaton. The idea is to inspect all dispositions of tokens within the TPN and find equivalence classes of these dispositions, for more details consult [2]. Adding a constraint, which limits the number of tokens at any time within the TPN, is also considered in this chapter.

The algorithms featured in this chapter do not return the minimal automata representing the input TPN as they are exactly visualising the equivalence classes of the dispositions of the tokens in the TPN. The automaton can be minimalised by choice, as it simplifies future computations involving the automaton also is makes the automata more legible.

4.1 Functions

4.1.1 GraphToAut

▷ `GraphToAut(`*g, innode, outnode*`)` (function)

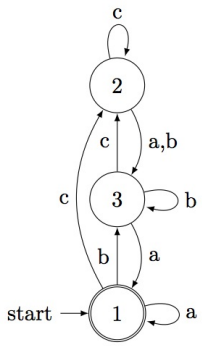
Returns: An automaton representing the input TPN.

`GraphToAut` turns an array represented directed graph, with a distinct input node and a distinct output node, into the corresponding automaton, that accepts the language build by the resulting rank encoded permutations of the directed graph.

Example

```
gap> x:=Seqstacks(2,2);
[ [ 2 ], [ 3, 4 ], [ 2 ], [ 5, 6 ], [ 4 ], [  ] ]
gap> aut:=GraphToAut(x,1,6);
< epsilon automaton on 4 letters with 64 states >
gap> aut:=MinimalAutomaton(aut);
< deterministic automaton on 3 letters with 3 states >
gap> Display(aut);
   |  1  2  3
--------------
 a |  1  3  1
 b |  3  3  3
 c |  2  2  2
Initial state:   [ 1 ]
Accepting state: [ 1 ]
gap>
```

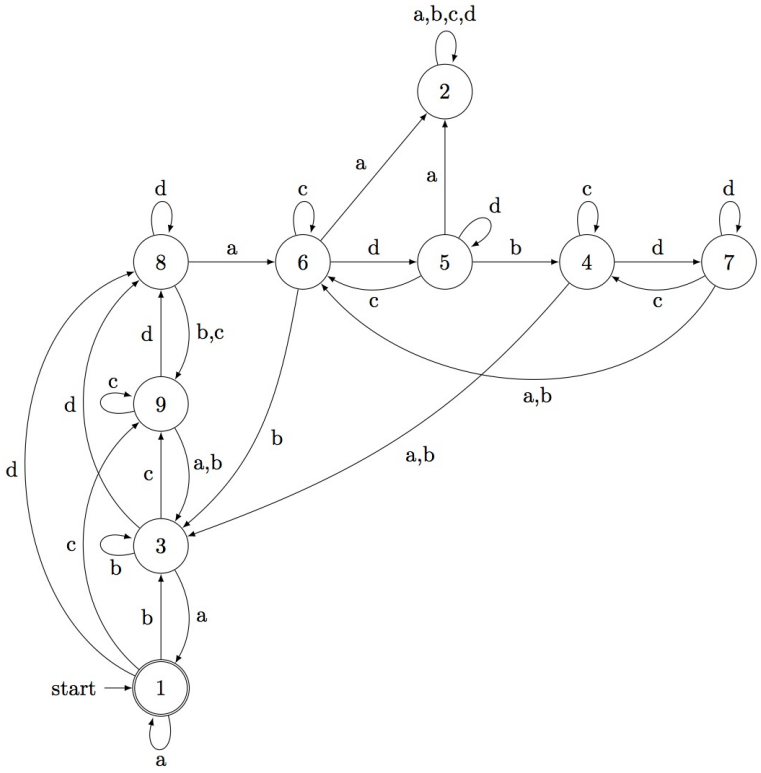
The minimal automaton corresponding to `Seqstacks(2,2)` is:



Example

```
gap> x:=Parstacks(2,2);
[ [ 2, 4 ], [ 3, 6 ], [ 2 ], [ 5, 6 ], [ 4 ], [  ] ]
gap> aut:=GraphToAut(x,1,6);
< epsilon automaton on 5 letters with 66 states >
gap> aut:=MinimalAutomaton(aut);
< deterministic automaton on 4 letters with 9 states >
gap> Display(aut);
   |  1  2  3  4  5  6  7  8  9
--------------------------------
 a |  1  2  1  3  2  2  6  6  3
 b |  3  2  3  3  4  3  6  9  3
 c |  9  2  9  4  6  6  4  9  9
 d |  8  2  8  7  5  5  7  8  8
Initial state:   [ 1 ]
Accepting state: [ 1 ]
gap>
```

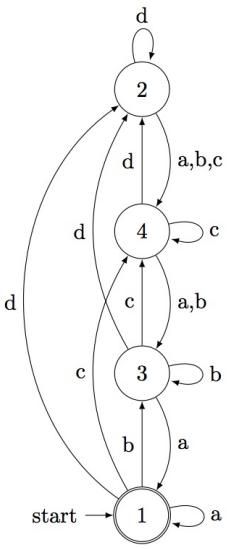
The rank encoded permutations of `Parstacks(2,2)` are accepted by the following automaton:



Example

```
gap> x:=BufferAndStack(3,2);
[ [ 2 .. 4 ], [ 5 ], [ 5 ], [ 5 ], [ 6, 7 ], [ 5 ], [  ] ]
gap> aut:=GraphToAut(x,1,7);
< epsilon automaton on 5 letters with 460 states >
gap> aut:=MinimalAutomaton(aut);
< deterministic automaton on 4 letters with 4 states >
gap> Display(aut);
   |  1  2  3  4
-----------------
 a |  1  4  1  3
 b |  3  4  3  3
 c |  4  4  4  4
 d |  2  2  2  2
Initial state:   [ 1 ]
Accepting state: [ 1 ]
gap>
```

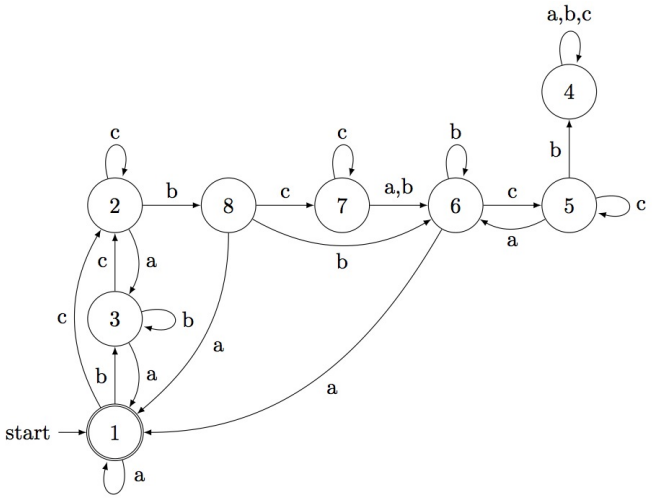
The following automaton represents the language of rank encoded permutations that are output by `BufferAndStack(3,2)`:



Example

```
gap> x:=[[2,3],[4],[5],[3,6],[6],[]];
[ [ 2, 3 ], [ 4 ], [ 5 ], [ 3, 6 ], [ 6 ], [  ] ]
gap> aut:=GraphToAut(x,1,6);
< epsilon automaton on 4 letters with 80 states >
gap> aut:=MinimalAutomaton(aut);
< deterministic automaton on 3 letters with 8 states >
gap> Display(aut);
   |  1  2  3  4  5  6  7  8
-----------------------------
 a |  1  3  1  4  6  1  6  1
 b |  3  8  3  4  4  6  6  6
 c |  2  2  2  4  5  5  7  7
Initial state:   [ 1 ]
Accepting state: [ 1 ]
gap>
```

The input TPN here is the first network described in Chapter 2. The language of rank encoded permutations of this token passing network is accepted by the following automaton:



4.1.2 ConstrainedGraphToAut

▷ `ConstrainedGraphToAut(`*g, innode, outnode, capacity*`)` (function)

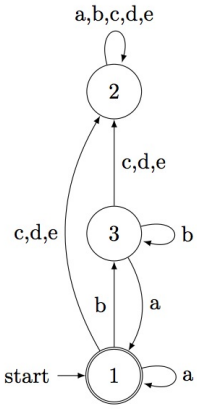
Returns: An automaton representing the input TPN with bounded capacity.

`ConstrainedGraphToAut` builds an epsilon automaton based on the same idea as `GraphToAut`, so it takes a list representation of a directed graph, a designated input node and a distinct output node, but additionally there is the constraint that there can be at most `capacity` tokens in the graph, at any time.

Example

```
gap> x:=Seqstacks(2,2);
[ [ 2 ], [ 3, 4 ], [ 2 ], [ 5, 6 ], [ 4 ], [  ] ]
gap> aut:=ConstrainedGraphToAut(x,1,6,2);
< epsilon automaton on 6 letters with 21 states >
gap> aut:=MinimalAutomaton(aut);
< deterministic automaton on 5 letters with 3 states >
gap> Display(aut);
   |  1  2  3
--------------
 a |  1  2  1
 b |  3  2  3
 c |  2  2  2
 d |  2  2  2
 e |  2  2  2
Initial state:   [ 1 ]
Accepting state: [ 1 ]
gap>
```

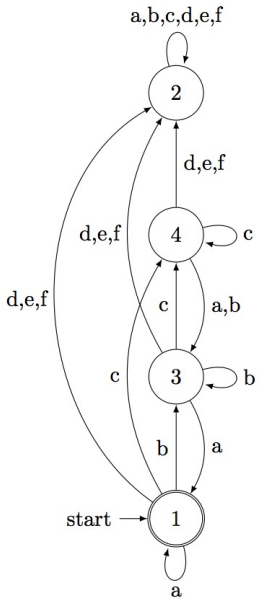
The rank encoded permutations of `Seqstacks(2,2)`, where at any time there are at most 2 tokens within the network, are accepted by the following automaton:



Example

```
gap> x:=BufferAndStack(3,2);
[ [ 2 .. 4 ], [ 5 ], [ 5 ], [ 5 ], [ 6, 7 ], [ 5 ], [  ] ]
gap> aut:=ConstrainedGraphToAut(x,1,6,3);
< epsilon automaton on 7 letters with 112 states >
gap> aut:=MinimalAutomaton(aut);
< deterministic automaton on 6 letters with 4 states >
gap> Display(aut);
   |  1  2  3  4
-----------------
 a |  1  2  1  3
 b |  3  2  3  3
 c |  4  2  4  4
 d |  2  2  2  2
 e |  2  2  2  2
 f |  2  2  2  2
Initial state:   [ 1 ]
Accepting state: [ 1 ]
gap>
```

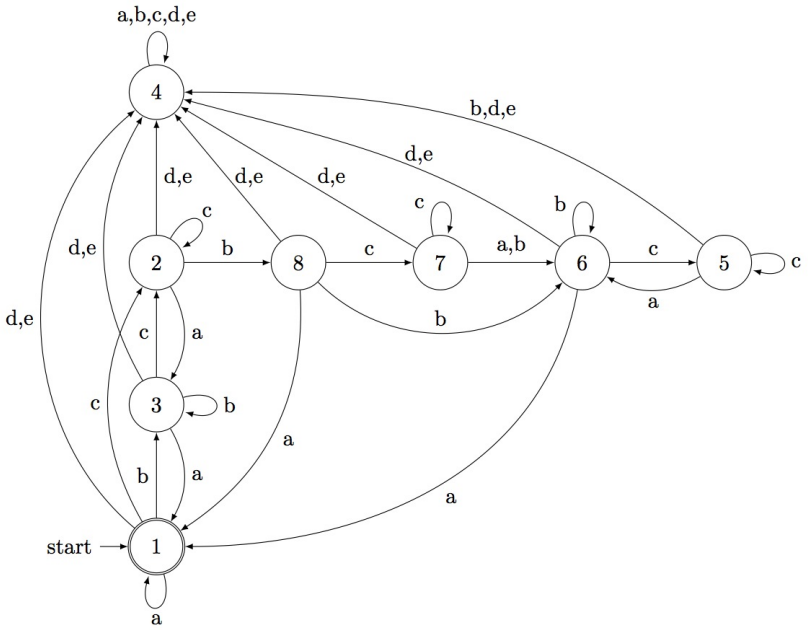
The automaton corresponding to `BufferAndStack(3,2)` with limited capacity of 3 tokens is:



Example

```
gap> x:=[[2,3],[4],[5],[3,6],[6],[]];
[ [ 2, 3 ], [ 4 ], [ 5 ], [ 3, 6 ], [ 6 ], [  ] ]
gap> aut:=ConstrainedGraphToAut(x,1,6,3);
< epsilon automaton on 6 letters with 48 states >
gap> aut:=MinimalAutomaton(aut);
< deterministic automaton on 5 letters with 8 states >
gap> Display(aut);
   |  1  2  3  4  5  6  7  8
------------------------------
 a |  1  3  1  4  6  1  6  1
 b |  3  8  3  4  4  6  6  6
 c |  2  2  2  4  5  5  7  7
 d |  4  4  4  4  4  4  4  4
 e |  4  4  4  4  4  4  4  4
Initial state:   [ 1 ]
Accepting state: [ 1 ]
gap>
```

The automaton that accepts the language of rank encoded permutations of the token passing network, from Chapter 2, with at most 3 tokens in the network at anytime, is:


a,b,c,d,e
4
b,d,e
d,e
d,e
d,e
d,e
c
c
b
d,e
2
b
8
c
7
a,b
6
c
5
c
a
b
d,e
c
a
c
3
b
a
b
a
a
start
1
a

Chapter 5

From Automata to Networks

It is not only important to see how a TPN can be interpreted as a finite state automaton. But also if an arbitrary automaton could represent the language of rank encoded permutations of a TPN. Currently within `PatternClass` it is only possible to check whether deterministic automata are possible representatives of a TPN.

5.1 Functions

5.1.1 IsStarClosed

▷ `IsStarClosed(`*aut*`)` (function)

Returns: `true` if the start and accept states in `aut` are one and the same state.

This has the consequence that the whole rational expression accepted by `aut` is always enclosed by the Kleene star.

Example

```
gap> x:=Automaton("det",4,2,[[3,4,2,4],[2,2,2,4]],[1],[2]);
< deterministic automaton on 2 letters with 4 states >
gap> IsStarClosed(x);
false
gap> AutToRatExp(x);
(a(aUb)Ub)b*
gap> y:=Automaton("det",3,2,[[3,2,1],[2,3,1]],[1],[1]);
< deterministic automaton on 2 letters with 3 states >
gap> IsStarClosed(y);
true
gap> AutToRatExp(y);
((ba*bUa)(aUb))*
gap>
```

5.1.2 Is2StarReplaceable

▷ `Is2StarReplaceable(`*aut*`)` (function)

Returns: `true` if none of the states in the automaton `aut`, which are not the initial state, have a transition to the initial state labelled with the first letter of the alphabet. It also returns `true` if there is at least one state $i \in Q$ with the first two transitions from i being $f(i,1) = 1$ and $f(i,2) = x$, where $x \in Q \setminus \{1\}$ and $f(x,1) = 1$.

Example

```
gap> x:=Automaton("det",3,2,[[1,2,3],[2,2,3]],[1],[2]);
< deterministic automaton on 2 letters with 3 states >
gap> Is2StarReplaceable(x);
true
gap> y:=Automaton("det",4,2,[[4,1,1,2],[1,3,3,2]],[1],[1]);
< deterministic automaton on 2 letters with 4 states >
gap> Is2StarReplaceable(y);
true
gap> z:=Automaton("det",4,2,[[4,1,1,2],[1,4,2,2]],[1],[4]);
< deterministic automaton on 2 letters with 4 states >
gap> Is2StarReplaceable(z);
false
gap>
```

5.1.3 IsStratified

▷ `IsStratified(aut)` (function)

Returns: `true` if `aut` has a specific "layered" form.

A formal description of the most basic stratified automaton is:

(S,Q,f,q_0,A) with $S := \{1,\ldots,n\}$, $Q := \{1,\ldots,m\}$, $n < m$, $q_0 := 1$, $A := Q \setminus \{n+1\}$, $f : Q \times S \to Q$ and $n+1$ is a sink state.

If $i,j \in Q \setminus \{n+1\}$,with $i < j$, and $i',j' \in S$, $i = i'$, $j = j'$ then

$$f(i,i') = i,\ f(i,j') = j,\ f(j,j') = j,\ f(j,i') = j-1 \text{ or } n+1.$$

Example

```
gap> x:=Automaton("det",4,2,[[1,3,1,4],[2,2,4,4]],[1],[2]);
< deterministic automaton on 2 letters with 4 states >
gap> IsStratified(x);
true
gap> y:=Automaton("det",4,2,[[1,3,2,4],[2,4,1,4]],[1],[2]);
< deterministic automaton on 2 letters with 4 states >
gap> IsStratified(y);
false
gap>
```

5.1.4 IsPossibleGraphAut

▷ `IsPossibleGraphAut(aut)` (function)

Returns: `true` if `aut` returns `true` in `IsStratified`, `Is2StarReplaceable` and `IsStarClosed`.

An automaton that fulfils the three functions above has the right form to be an automaton representing rank encoded permutations, which are output from a TPN.

Example

```
gap> x:=Automaton("det",2,2,[[1,2],[2,2]],[1],[1]);
< deterministic automaton on 2 letters with 2 states >
gap> IsPossibleGraphAut(x);
true
gap> y:=Automaton("det",2,2,[[1,2],[1,2]],[1],[1]);
< deterministic automaton on 2 letters with 2 states >
```

```
gap> IsPossibleGraphAut(y);
false
gap> IsStarClosed(y);
true
gap> Is2StarReplaceable(y);
true
gap> IsStratified(y);
false
gap>
```

Chapter 6

Pattern Classes

Permutation pattern classes can be determined using their corresponding basis. The basis of a pattern class is the anti-chain of the class, under the order of containment. A permutation π contains another permutation σ (of shorter length) if there is a subsequence in π, which is isomorphic to σ.

With the rational language of the rank encoded class, it is also possible to find the rational language of the basis and vice versa. Several specific kinds of transducers are used in this process. [1]

6.1 Transducers

6.1.1 Transducer

▷ `Transducer(`*`states, init, transitions, accepting`*`)` (function)

Returns: A record that represents a transducer.

A transducer is essentially an automaton, where running through the process does not determine whether the input is accepted, but the input is translated to another language, over a different alphabet.

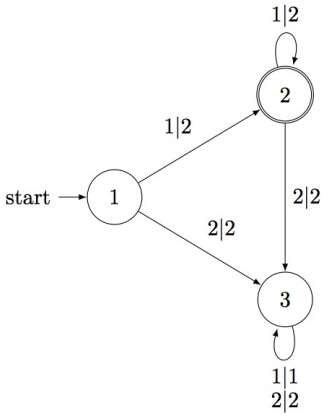
Formally a transducer is a six-tuple with: Q being the set of states, S the input alphabet, G the output alphabet, $I \in Q$ the start state, $A \subseteq Q$ the set of accept states, and with the transition function $f : Q \times (S \cup \{e\}) \to Q \times (G \cup \{e\})$, where e is the empty word.

In this function the transducer is stored by defining how many states there are, which one (by index) is the start or initial state, the transitions are of the form `[inputletter,outputletter,fromstate,tostate]` and a list of accept states. The input and output alphabet are determined by the input and output letters on the transitions.

Example

```
gap> trans:=Transducer(3,1,[[1,2,1,2],[1,2,2,2],[2,2,1,3],[2,2,2,3],
> [1,1,3,3],[2,2,3,3]],[2]);
rec( accepting := [ 2 ], initial := 1, states := 3,
  transitions := [ [ 1, 2, 1, 2 ], [ 1, 2, 2, 2 ], [ 2, 2, 1, 3 ],
      [ 2, 2, 2, 3 ], [ 1, 1, 3, 3 ], [ 2, 2, 3, 3 ] ] )
gap>
```

This transducer can be visualised as the following graph:



6.1.2 DeletionTransducer

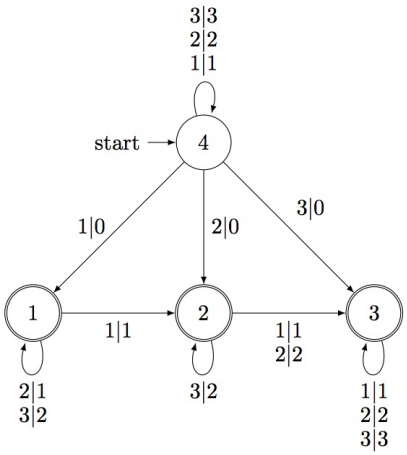
▷ `DeletionTransducer(k)` (function)

Returns: A transducer over k+1 states.

A deletion transducer is a transducer that deletes one letter of a rank encoded permutation and returns the correct rank encoding of the new permutation. The deletion transducer over k deletes letters over the set of all rank encoded permutations with highest rank k. Specifically, running through a deletion transducer with a rank encoded permutation, of highest rank k, will lead to the set of all rank encoded permutations that have one letter of the initial permutation removed. It is important to note that computing these shorter permutations with the transducer, is done by reading the input permutation from right to left. For example the deletion transducer with k=3, looks as follows:

Example

```
gap> DeletionTransducer(3);
rec( accepting := [ 1 .. 3 ], initial := 4, states := 4,
  transitions := [ [ 1, 1, 4, 4 ], [ 1, 0, 4, 1 ], [ 2, 1, 1, 1 ],
      [ 1, 1, 1, 2 ], [ 3, 2, 1, 1 ], [ 1, 1, 2, 3 ], [ 1, 1, 3, 3 ],
      [ 2, 2, 4, 4 ], [ 2, 0, 4, 2 ], [ 3, 2, 2, 2 ], [ 2, 2, 2, 3 ],
      [ 2, 2, 3, 3 ], [ 3, 3, 4, 4 ], [ 3, 0, 4, 3 ], [ 3, 3, 3, 3 ] ] )
gap>
```



6.1.3 TransposedTransducer

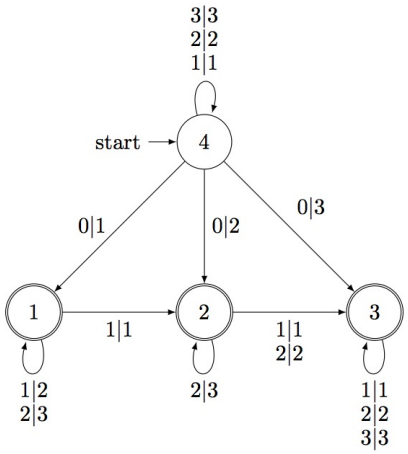
▷ `TransposedTransducer(t)` (function)

Returns: A new transducer with interchanged input and output letters on each transition.

A transducer is transposed when all origins and destinations of transitions are left the same as before but the input and output letters on each transition are interchanged. Taking the deletion transducer from above, its transpose looks as follows:

Example

```
gap> TransposedTransducer(DeletionTransducer(3));
rec( accepting := [ 1 .. 3 ], initial := 4, states := 4,
  transitions := [ [ 1, 1, 4, 4 ], [ 0, 1, 4, 1 ], [ 1, 2, 1, 1 ],
      [ 1, 1, 1, 2 ], [ 2, 3, 1, 1 ], [ 1, 1, 2, 3 ], [ 1, 1, 3, 3 ],
      [ 2, 2, 4, 4 ], [ 0, 2, 4, 2 ], [ 2, 3, 2, 2 ], [ 2, 2, 2, 3 ],
      [ 2, 2, 3, 3 ], [ 3, 3, 4, 4 ], [ 0, 3, 4, 3 ], [ 3, 3, 3, 3 ] ] )
gap>
```



6.1.4 InvolvementTransducer

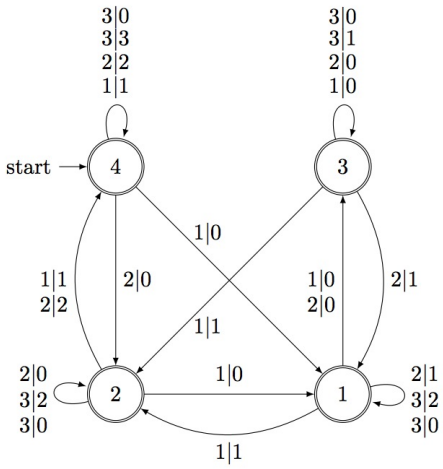
▷ `InvolvementTransducer(k)` (function)

Returns: A transducer over k+1 states, with a k sized alphabet.

An involvement transducer is a transducer over k+1 states, and deletes any number of letters in a rank encoded permutation, of rank at most k.

Example

```
gap> InvolvementTransducer(3);
rec( accepting := [ 1 .. 4 ], initial := 4, states := 4,
  transitions := [ [ 1, 1, 1, 2 ], [ 1, 0, 1, 3 ], [ 2, 1, 1, 1 ],
      [ 2, 0, 1, 3 ], [ 3, 2, 1, 1 ], [ 3, 0, 1, 1 ], [ 1, 1, 2, 4 ],
      [ 1, 0, 2, 1 ], [ 2, 2, 2, 4 ], [ 2, 0, 2, 2 ], [ 3, 2, 2, 2 ],
      [ 3, 0, 2, 2 ], [ 1, 1, 3, 2 ], [ 1, 0, 3, 3 ], [ 2, 1, 3, 1 ],
      [ 2, 0, 3, 3 ], [ 3, 1, 3, 3 ], [ 3, 0, 3, 3 ], [ 1, 1, 4, 4 ],
      [ 1, 0, 4, 1 ], [ 2, 2, 4, 4 ], [ 2, 0, 4, 2 ], [ 3, 3, 4, 4 ],
      [ 3, 0, 4, 4 ] ] )
gap>
```



6.1.5 CombineAutTransducer

▷ `CombineAutTransducer(`*aut, trans*`)` (function)

Returns: An automaton consisting of a combination of `aut` and `trans`.

Combining automata and transducers is done over the natural "translation" of the by the automaton accepted language through the transducer and then building a new automaton that accepts the new language. The function `CombineAutTransducer` does this process and returns the new non-deterministic automaton.

Example

```
gap> a:=Automaton("det",1,1,[[1]],[1],[1]);
< deterministic automaton on 1 letters with 1 states >
gap> AutToRatExp(a);
a*
gap> t:=Transducer(2,1,[[1,2,1,2],[2,1,1,2],
> [1,1,2,1],[2,2,2,1]],[1]);
rec( accepting := [ 1 ], initial := 1, states := 2,
  transitions := [ [ 1, 2, 1, 2 ], [ 2, 1, 1, 2 ], [ 1, 1, 2, 1 ],
      [ 2, 2, 2, 1 ] ] )
gap> res:=CombineAutTransducer(a,t);
< non deterministic automaton on 2 letters with 2 states >
gap> AutToRatExp(res);
(ba)*
gap> Display(res);
   |  1       2
-------------------
 a |         [ 1 ]
 b | [ 2 ]
Initial state:   [ 1 ]
Accepting state: [ 1 ]
gap>
```

6.2 From Class to Basis and vice versa

6.2.1 BasisAutomaton

▷ `BasisAutomaton(a)` (function)

Returns: An automaton that accepts the rank encoded permutations of the basis of the input automaton `a`.

Every pattern class has a basis that consists of the smallest set of permutations which do not belong to the class. Using

$$B(L) = (L^r D^t)^r \cap L^c$$

it is possible using the deletion transducer D and the language of rank encoded permutations L to find the language of the rank encoded permutations of the basis $B(L)$, and thus the basis.

Example

```
gap> x:=Parstacks(2,2);
[ [ 2, 4 ], [ 3, 6 ], [ 2 ], [ 5, 6 ], [ 4 ], [  ] ]
gap> xa:=GraphToAut(x,1,6);
< epsilon automaton on 5 letters with 66 states >
gap> ma:=MinimalAutomaton(xa);
< deterministic automaton on 4 letters with 9 states >
gap> Display(ma);
   |  1  2  3  4  5  6  7  8  9
--------------------------------
 a |  1  2  1  3  2  2  6  6  3
 b |  3  2  3  3  4  3  6  9  3
 c |  9  2  9  4  6  6  4  9  9
 d |  8  2  8  7  5  5  7  8  8
Initial state:   [ 1 ]
Accepting state: [ 1 ]
gap> ba:=BasisAutomaton(ma);
< deterministic automaton on 4 letters with 9 states >
gap> Display(ba);
   |  1  2  3  4  5  6  7  8  9
--------------------------------
 a |  2  2  1  3  4  2  2  2  2
 b |  2  2  2  2  2  4  1  2  8
 c |  2  2  2  2  2  2  1  2  2
 d |  2  2  2  9  2  2  5  6  2
Initial state:    [ 7 ]
Accepting states: [ 1, 5 ]
gap> AutToRatExp(ba);
d(a(dbdb)*aaU@)UbUc
gap>
```

Ignoring the trailing `UbUc` which essentially are noise, the basis of the pattern class indicates which permutations are avoided in this particular class. The shortest permutation in the basis, looking at the rational expression, is *daaa*, which can be translated to 4111 and decoded to the permutation 4123.

6.2.2 ClassAutomaton

▷ `ClassAutomaton(a)` (function)

Returns: The automaton that accepts permutations of the class in their rank encoding.

The function ClassAutomaton does the reverse process of BasisAutomaton. Namely it takes the automaton that represents the language of the rank encoded basis of a permutation class, and using the formula

$$L = B_k \cap ((B(L)^r I^t)^c)^r$$

returns the automaton that accepts the rank encoded permutations of the class. In the formula, B_k is the automaton that accepts all permutations of any length with highest rank k, $B(L)$ is the automaton that represents the basis and I is the involement transducer.

Example

```
gap> xa:=Automaton("det",9,4,[[2,2,1,3,4,2,2,2,2],[2,2,2,2,2,4,1,2,8],
> [2,2,2,2,2,2,1,2,2],[2,2,2,9,2,2,5,6,2]],[7],[1,5]);
< deterministic automaton on 4 letters with 9 states >
gap> ca:=ClassAutomaton(xa);
< deterministic automaton on 4 letters with 9 states >
gap> Display(ca);
   |  1  2  3  4  5  6  7  8  9
--------------------------------
 a |  1  2  1  3  2  2  6  6  3
 b |  3  2  3  3  4  3  6  9  3
 c |  9  2  9  4  6  6  4  9  9
 d |  8  2  8  7  5  5  7  8  8
Initial state:   [ 1 ]
Accepting state: [ 1 ]
gap> IsPossibleGraphAut(ca);
true
gap>
```

6.2.3 BoundedClassAutomaton

▷ BoundedClassAutomaton(*k*) (function)

Returns: An automaton that accepts all rank encoded permutations, with highest rank being k.

The bounded class automaton, is an automaton that accepts all rank encoded permutations, of any length, with highest rank k.

Example

```
gap> BoundedClassAutomaton(3);
< deterministic automaton on 3 letters with 3 states >
gap> Display(last);
   |  1  2  3
--------------
 a |  1  1  2
 b |  2  2  2
 c |  3  3  3
Initial state:   [ 1 ]
Accepting state: [ 1 ]
gap>
```

6.2.4 ClassAutFromBaseEncoding

▷ ClassAutFromBaseEncoding(*base, k*) (function)

Returns: The class automaton from a list of rank encoded basis permutations.

Given the permutations in the basis, in their rank encoded form, and the bound of the rank of the permutations of the class, ClassAutFromBaseEncoding builds the automaton that accepts rank encoded permutations of the class.

Example

```
gap> ClassAutFromBaseEncoding([[4,1,1,1]],4);
< deterministic automaton on 4 letters with 7 states >
gap> Display(last);
   |  1  2  3  4  5  6  7
--------------------------
 a |  1  2  1  3  2  2  6
 b |  3  2  3  3  4  3  4
 c |  4  2  4  4  6  6  4
 d |  7  2  7  7  5  5  7
Initial state:   [ 1 ]
Accepting state: [ 1 ]
gap>
```

6.2.5 ClassAutFromBase

▷ ClassAutFromBase(*perms, k*) (function)

Returns: The class automaton from a list of permutations of the basis.

Taking perms which is the list of permutations in the basis and k an integer which indicates the highest rank in the encoded permutations of the class, ClassAutFromBase constructs the automaton that accepts the language of rank encoded permutations of the class.

Example

```
gap> ClassAutFromBase([[4,1,2,3]],4);
< deterministic automaton on 4 letters with 7 states >
gap> Display(last);
   |  1  2  3  4  5  6  7
--------------------------
 a |  1  2  1  3  2  2  6
 b |  3  2  3  3  4  3  4
 c |  4  2  4  4  6  6  4
 d |  7  2  7  7  5  5  7
Initial state:   [ 1 ]
Accepting state: [ 1 ]
gap>
```

6.2.6 ExpandAlphabet

▷ ExpandAlphabet(*a, newAlphabet*) (function)

Returns: The automaton a, over the alphabet of size newAlphabet.

Given an automaton and the size of the new alphabet, which has to be bigger than the size of the alphabet in a , ExpandAlphabet changes the automaton a to contain an alphabet of size newAlphabet. The new letters have no transitions within the automaton.

Example

```
gap> aut:=Automaton("det",3,2,[[2,2,3],[3,3,3]],[1],[3]);
< deterministic automaton on 2 letters with 3 states >
gap> Display(aut);
   |  1  2  3
```

```
--------------
 a |  2  2  3
 b |  3  3  3
Initial state:   [ 1 ]
Accepting state: [ 3 ]
gap> ExpandAlphabet(aut,4);
< deterministic automaton on 4 letters with 3 states >
gap> Display(last);
   |  1  2  3
--------------
 a |  2  2  3
 b |  3  3  3
 c |
 d |
Initial state:   [ 1 ]
Accepting state: [ 3 ]
gap>
```

6.3 Direct Sum of Regular Classes

It is obvious that the direct sum of two rational pattern classes is also rational. But the skew sum of two rational pattern classes does not imply that the resulting pattern class is rational, because if the second class in the sum has infinitely many permutations, the alphabet of the skew sum class will be infinite and thusly the resulting class will not be rational.

6.3.1 ClassDirectSum

▷ `ClassDirectSum(`*aut1, aut2*`)` (function)

Returns: An automaton that corresponds to the direct sum of `aut1` with `aut2`.

`ClassDirectSum` builds the concatenation automaton of `aut1` with `aut2`, which corresponds to the pattern class `aut1` $\oplus$ `aut2`.

Example

```
gap> a:=BasisAutomaton(GraphToAut(Parstacks(2,2),1,6));
< deterministic automaton on 4 letters with 9 states >
gap> AutToRatExp(a);
d(a(dbdb)*aaU@)UbUc
gap> b:=MinimalAutomaton(GraphToAut(Seqstacks(2,2),1,6));
< deterministic automaton on 3 letters with 3 states >
gap> AutToRatExp(b);
((cc*(aUb)Ub)(cc*(aUb)Ub)*aUa)*
gap> ab:=ClassDirectSum(a,b);
< deterministic automaton on 4 letters with 11 states >
gap> AutToRatExp(ab);
((d(acUc)c*(aUb)Ud(abUb))(cc*(aUb)Ub)*aUda(d(bdbd)*bdbaaUa)UbUc)((cc*(aUb)U\
b)(cc*(aUb)Ub)*aUa)*Ud(aU@)
gap>
```

6.4 Statistical Inspections

It is of interest to see what permutations and how many of different length are accepted by the class, respectively the basis.

In this section, the examples will be inspecting the basis automaton of the token passing network containing 2 stacks of capacity 2, which are situated in parallel to each other.

Example

```
gap> x:=Parstacks(2,2);
[ [ 2, 4 ], [ 3, 6 ], [ 2 ], [ 5, 6 ], [ 4 ], [  ] ]
gap> xa:=GraphToAut(x,1,6);
< epsilon automaton on 5 letters with 66 states >
gap> ma:=MinimalAutomaton(xa);
< deterministic automaton on 4 letters with 9 states >
gap> ba:=BasisAutomaton(ma);
< deterministic automaton on 4 letters with 9 states >
gap> AutToRatExp(ba);
d(a(dbdb)*aaU@)UbUc
gap>
```

6.4.1 Spectrum

▷ `Spectrum(`*aut[, int]*`)` (function)

Returns: A list indicating how many words of each length from 1 to `int` or 15 (default) are accepted by the automaton.

Each entry in the returned list indicates how many words of length the index of the entry are accepted by the automaton `aut`. The length of the list is by default 15, but if this is too much or too little the second optional argument regulates the length of the list.

Example

```
gap> Spectrum(ba);
[ 3, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0 ]
gap> Spectrum(ba,20);
[ 3, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 1 ]
gap>
```

6.4.2 NumberAcceptedWords

▷ `NumberAcceptedWords(`*aut, len*`)` (function)

Returns: The number of words of length `len` accepted by the automaton `aut`.

Given the automaton `aut` and the integer `len`, `NumberAcceptedWords` determines how many words of length `len` are accepted by the automaton.

Example

```
gap> NumberAcceptedWords(ba,1);
3
gap> NumberAcceptedWords(ba,16);
1
gap>
```

6.4.3 AutStateTransitionMatrix

▷ `AutStateTransitionMatrix(aut)` (function)

Returns: A matrix containing the number of transitions between states of the automaton `aut`.

In the matrix computed by `AutStateTransitionMatrix` the rows are indexed by the state the transitions are originating from, the columnns are indexed by the states the transitions are ending at. Each entry $a_{i,j}$ of the matrix represents the number of transitions from the state i to the state j.

Example

```
gap> AutStateTransitionMatrix(ba);
[ [ 0, 4, 0, 0, 0, 0, 0, 0, 0 ], [ 0, 4, 0, 0, 0, 0, 0, 0, 0 ],
  [ 1, 3, 0, 0, 0, 0, 0, 0, 0 ], [ 0, 2, 1, 0, 0, 0, 0, 0, 1 ],
  [ 0, 3, 0, 1, 0, 0, 0, 0, 0 ], [ 0, 3, 0, 1, 0, 0, 0, 0, 0 ],
  [ 2, 1, 0, 0, 1, 0, 0, 0, 0 ], [ 0, 3, 0, 0, 0, 1, 0, 0, 0 ],
  [ 0, 3, 0, 0, 0, 0, 0, 1, 0 ] ]
gap>
```

6.4.4 AcceptedWords

▷ `AcceptedWords(aut, int)` (function)

Returns: All words of length `int` that are accepted by the automaton `aut`.

`AcceptedWords` outputs all permutations accepted by the automaton `aut`, which have length `int`, in a list. The permutations are output in their rank encoding.

Example

```
gap> AcceptedWords(ba,1);
[ [ 2 ], [ 3 ], [ 4 ] ]
gap> AcceptedWords(ba,16);
[ [ 4, 1, 4, 2, 4, 2, 4, 2, 4, 2, 4, 2, 4, 2, 1, 1 ] ]
gap>
```

6.4.5 AcceptedWordsR

▷ `AcceptedWordsR(aut, int1[, int2])` (function)

▷ `AcceptedWordsReversed(aut, int1[, int2])` (function)

Returns: The list of all by the automaton accepted words of length `int1`, where each word is written in reverse.

The functions `AcceptedWordsR` and `AcceptedWordsReversed` are synonymous and take the following arguments; an automaton `aut`, an integer `int1` which indicates the length of the words that are accepted by the `aut` and another integer `int2` which is optional and represents the initial state of `aut`. The return value of these functions is the list containing all permutations of length `int1` that are accepted by `aut`. The permutations are rank encoded and written in reverse.

Example

```
gap> AcceptedWordsR(ba,1);
[ [ 2 ], [ 3 ], [ 4 ] ]
gap> AcceptedWordsReversed(ba,16);
[ [ 1, 1, 2, 4, 2, 4, 2, 4, 2, 4, 2, 4, 2, 4, 1, 4 ] ]
gap>
```

Chapter 7

Some Permutation Essentials

In this chapter we mention a couple functions that are fairly basic but useful tools to work with.

7.1 Complement

7.1.1 PermComplement

▷ `PermComplement(`*perm*`)` (function)

Returns: The permutation that is the complement of `perm`.

The complement of a permutation $\tau = \tau_1 \dots \tau_n$ is the permutation

$$\tau^C = (n+1) - \tau_1 \;\; (n+1) - \tau_2 \dots (n+1) - \tau_n$$

.

Example

```
gap> PermComplement([3,2,8,6,7,1,5,4]);
[ 6, 7, 1, 3, 2, 8, 4, 5 ]
gap>
```

7.2 Rank Encoding

7.2.1 IsRankEncoding

▷ `IsRankEncoding(`*perm*`)` (function)

Returns: `true` if `perm` is a valid rank encoding of a permutation.

`IsRankEncoding` checkes whether the input list `perm` is a valid rank encoding by checking whether it is accepted by the bounded class automaton, with the highest rank being set by the highest element in `perm`.

Example

```
gap> IsRankEncoding([3,2,6,4,4,1,2,1]);
true
gap> IsRankEncoding([3,2,6,4,5,1,2,1]);
false
gap>
```

Chapter 8

Properties of Permutations

It has been of interest to the authors to compute different properties of permutations. Inspections include plus- and minus-decomposable permutations, block-decompositions of permutations, as well as the computation of the direct and skew sum of permutations. First a couple of definitions on which some of the properties are based on:

An interval of a permutation σ is a set of contiguous values which in σ have consecutive indices.

A permutation of length n is called simple if it only contains the intervals of length 0, 1 and n.

8.1 Intervals in Permutations

As mentioned above, an interval of a permutation σ is a set of contiguous numbers which in σ have consecutive indices. For example in $\sigma = 46871523$ the following is an interval $\sigma(2)\sigma(3)\sigma(4) = 687$ whereas $\sigma(4)\sigma(5)\sigma(6) = 715$ is not.

8.1.1 IsInterval

▷ `IsInterval(`*list*`)` (function)

Returns: `true`, if `list` is an interval.

IsInterval takes in any list with unique elements, which can be ordered lexicographically and checkes whether it is an interval.

Example

```
gap> IsInterval([3,6,9,2]);
false
gap> IsInterval([2,6,5,3,4]);
true
gap>
```

8.2 Simplicity

As mentioned above a permutation is said to be simple if it only contains intervals of length 0, 1 or the length of the permutation.

8.2.1 IsSimplePerm

▷ `IsSimplePerm(`*perm*`)` (function)

Returns: `true` if `perm` is simple.

To check whether `perm` (length of `perm` = n) is a simple permutation `IsSimplePerm` uses the basic algorithm proposed by Uno and Yagiura in [3] to compare the `perm` against the identity permutation of the same length.

Example

```
gap> IsSimplePerm([2,3,4,5,1,1,1,1]);
true
gap> IsSimplePerm([2,4,6,8,1,3,5,7]);
true
gap> IsSimplePerm([3,2,8,6,7,1,5,4]);
false
gap>
```

8.3 Point Deletion in Simple Permutations

In [4] it is shown how one can get chains of permutations by starting with a simple permutation and then removing either a single point or two points and the resulting permutation would still be simple. We have applied this theory to create functions such that the set of simple permutations of shorter length, by one deletion, can be found.

8.3.1 OnePointDelete

▷ `OnePointDelete(`*perm*`)` (function)

Returns: A list of simple permutations with one point less than `perm`.

`OnePointDelete` removes single points in the simple permutation and returns a list of the resulting simple permutations, in their rank encoding.

Example

```
gap> OnePointDelete([5,2,3,1,2,1]);
[ [ 2, 3, 1, 2, 1 ], [ 4, 1, 2, 2, 1 ] ]
gap> OnePointDelete([5,2,4,1,6,3]);
[ [ 2, 3, 1, 2, 1 ], [ 4, 1, 2, 2, 1 ] ]
gap>
```

8.3.2 TwoPointDelete

▷ `TwoPointDelete(`*perm*`)` (function)

Returns: The exceptional permutation with two point less than `perm`.

`TwoPointDelete` removes two points of the input exceptional permutation and returns the list of the unique resulting permutation, in its rank encoding.

Example

```
gap> TwoPointDelete([2,4,6,8,1,3,5,7]);
[ [ 2, 3, 4, 1, 1, 1 ] ]
gap> TwoPointDelete([2,3,4,5,1,1,1,1]);
[ [ 2, 3, 4, 1, 1, 1 ] ]
gap>
```

8.3.3 PointDeletion

▷ `PointDeletion(perm)` (function)

Returns: A list of simple permutations with of shorter length than `perm`.

`PointDeletion` takes any simple permutation does not matter whether exceptional or not and removes the right number of points.

Example

```
gap> PointDeletion([5,2,3,1,2,1]);
[ [ 2, 3, 1, 2, 1 ], [ 4, 1, 2, 2, 1 ] ]
gap> PointDeletion([5,2,4,1,6,3]);
[ [ 2, 3, 1, 2, 1 ], [ 4, 1, 2, 2, 1 ] ]
gap> PointDeletion([2,4,6,8,1,3,5,7]);
[ [ 2, 3, 4, 1, 1, 1 ] ]
gap> PointDeletion([2,3,4,5,1,1,1,1]);
[ [ 2, 3, 4, 1, 1, 1 ] ]
gap>
```

8.4 Block-Decomposition

Given a permutation π of length m and nonempty permutations $\alpha_1,\ldots,\alpha_m$ the inflation of π by $\alpha_1,\ldots,\alpha_m$, written as $\pi[\alpha_1,\ldots,\alpha_m]$, is the permutation obtained by replacing each entry $\pi(i)$ by an interval that is order isomorphic to α_i [5]. Conversely a block-decomposition of σ is any expression of σ as an inflation $\sigma=\pi[\alpha_1,\ldots,\alpha_m]$. The block decomposition of a permutation is unique if and only if $\sigma,\pi,\alpha_1,\ldots,\alpha_n$ all are in the same pattern class and π is simple and $\pi\neq 12,\ 21$ [6].

For example the inflation of $25413[21,1,1,1,2413]=328915746$, written in GAP this is `[[2,5,4,1,3],[2,1],[1],[1],[1],[2,4,1,3]]`. This decomposition of 328915746 is not unique. The unique block-decomposition, as described above, for $328915746=2413[21,12,1,2413]$ or in GAP notation `[3,2,8,9,1,5,7,4,6]=[[2,4,1,3],[2,1],[1,2],[1],[2,4,1,3]]`.

8.4.1 Inflation

▷ `Inflation(list_of_perms)` (function)

Returns: A permutation that represents the inflation of the list of permutations, taking the first permutation to be π, as described in the definition of inflation.

Inflation takes the list of permutations that stand for a box decomposition of a permutation, and calculates that permutation by replacing each entry i in the first permutation by an interval order isomorphic to the permutation in index $i+1$.

Example

```
gap> Inflation([[3,2,1],[1],[1,2],[1,2,3]]);
[ 6, 4, 5, 1, 2, 3 ]
gap> Inflation([[1,2],[1],[4,2,1,3]]);
[ 1, 5, 3, 2, 4 ]
gap> Inflation([[2,4,1,3],[2,1],[3,1,2],[1],[2,4,1,3]]);
[ 3, 2, 10, 8, 9, 1, 5, 7, 4, 6 ]
gap>
```

8.4.2 BlockDecomposition

▷ `BlockDecomposition(`*perm*`)` (function)

Returns: A list of permutations, representing the block-decomposition of `perm`. In the list the first permutation is π, as described in the definiton of block-decomposition above.

`BlockDecomposition` takes a plus- and minus-indecomposable permutation and decomposes it into its maximal maximal intervals, which are preceded by the simple permutation that represents the positions of the intervals. If a plus- or minus-decomposable permutation is input, then the decomposition will not be the unique decomposition, by the definition of plus- or minus- decomposable permutations, see below.

Example

```
gap> BlockDecomposition([3,2,10,8,9,1,5,7,4,6]);
[ [ 2, 4, 1, 3 ], [ 2, 1 ], [ 3, 1, 2 ], [ 1 ], [ 2, 4, 1, 3 ] ]
gap> BlockDecomposition([1,2,3,4,5]);
[ [ 1, 2 ], [ 1, 2, 3, 4 ], [ 1 ] ]
gap> BlockDecomposition([5,4,3,2,1]);
[ [ 2, 1 ], [ 4, 3, 2, 1 ], [ 1 ] ]
gap>
```

8.5 Plus-Decomposability

A permutation σ is said to be plus-decomposable if it can be written uniquely in the following form,

$$\sigma = 12[\alpha_1, \alpha_2]$$

where α_1 is not plus-decomposable.

The subset of a rational class, containing all permutations that are plus-decomposable and in the class, has been found to be also rational under the rank encoding.

8.5.1 IsPlusDecomposable

▷ `IsPlusDecomposable(`*perm*`)` (function)

Returns: `true` if `perm` is plus-decomposable.

To check whether `perm` is a plus-decomposable permutation `IsPlusDecomposable` uses the fact that there has to be an interval $1..x$ where $x < n$ (n = length of the `perm`) in the rank encoded permutation that is a valid rank encoding.

Example

```
gap> IsPlusDecomposable([3,3,2,3,2,2,1,1]);
true
gap> IsPlusDecomposable([3,4,2,6,5,7,1,8]);
true
gap> IsPlusDecomposable([3,2,8,6,7,1,5,4]);
false
gap>
```

8.6 Minus-Decomposability

Minus-decomposability is essentially the same as plus-decomposability, the difference is that if a permutation σ is minus-decomposable, it can be written uniquely in the following form,

$$\sigma = 21[\alpha_1, \alpha_2]$$

where α_1 is not minus-decomposable.

Here also, the subset of a rational class, containing all permutations that are minus-decomposable and in the class, has been found to be rational under the rank encoding.

8.6.1 IsMinusDecomposable

▷ `IsMinusDecomposable(`*perm*`)` (function)

Returns: `true` if `perm` is minus-decomposable.

To check whether `perm` (length of `perm` = n) is a minus-decomposable permutation `IsMinusDecomposable` uses the fact that the first $n-x$, where $x < n$, letters in the rank encoding of `perm` have to be $> x$ and that the letters from position $x+1$ until the last one have to be $\leq x$.

Example

```
gap> IsMinusDecomposable([3,3,3,3,3,3,2,1]);
true
gap> IsMinusDecomposable([3,4,5,6,7,8,2,1]);
true
gap> IsMinusDecomposable([3,2,8,6,7,1,5,4]);
false
gap>
```

8.7 Sums of Permutations

The direct sum of two permutations $\sigma = \sigma_1 \dots \sigma_k$ and $\tau = \tau_1 \dots \tau_l$ is defined as,

$$\sigma \oplus \tau = \sigma_1 \ \ \sigma_2 \dots \sigma_k \ \ \tau_1 + k \ \ \tau_2 + k \dots \tau_l + k \, .$$

In a similar fashion the skew sum of σ, τ is

$$\sigma \ominus \tau = \sigma_1 + l \ \ \sigma_2 + l \dots \sigma_k + l \ \ \tau_1 \ \tau_2 \dots \tau_l \, .$$

The calculation of the direct and skew sums of permutations using the rank encoding is also straight forward and is used in the functions described below. The direct sum of two permutations σ, τ represented as their rank encoded sequences is the permutation which has the rank encoding that is the concatention of the rank encoding of σ and τ. The skew sum of two permutations σ, τ encoded by the rank encoding is the concatenation of the rank encodings of σ and τ where in the sequence corresponding to σ under the rank encoding each element has been increased by l, with l being the length of τ.

8.7.1 PermDirectSum

▷ `PermDirectSum(`*perm1, perm2*`)` (function)

Returns: A permutation resulting from `perm1` $\oplus$ `perm2`.

`PermDirectSum` returns the permutation corresponding to `perm1` $\oplus$ `perm2` if `perm1` and `perm2` are both not rank encoded. If both `perm1` and `perm2` are rank encoded, then `PermDirectSum` returns a rank encoded sequence.

Example

```
gap> PermDirectSum([2,4,1,3],[2,5,4,1,3]);
[ 2, 4, 1, 3, 6, 9, 8, 5, 7 ]
gap> PermDirectSum([2,3,1,1],[2,4,3,1,1]);
[ 2, 3, 1, 1, 2, 4, 3, 1, 1 ]
gap>
```

8.7.2 PermSkewSum

▷ `PermSkewSum(`*perm1, perm2*`)` (function)

Returns: A permutation resulting from `perm1` $\ominus$ `perm2`.

`PermSkewSum` returns the permutation corresponding to `perm1` $\ominus$ `perm2` if `perm1` and `perm2` are both not rank encoded. If both `perm1` and `perm2` are rank encoded, then `PermSkewSum` returns a rank encoded sequence.

Example

```
gap> PermSkewSum([2,4,1,3],[2,5,4,1,3]);
[ 7, 9, 6, 8, 2, 5, 4, 1, 3 ]
gap> PermSkewSum([2,3,1,1],[2,4,3,1,1]);
[ 7, 8, 6, 6, 2, 4, 3, 1, 1 ]
gap>
```

Chapter 9

Regular Languages of Sets of Permutations

This chapter is dedicated to the different sets of permutations with the same properties.

9.1 Inversions in Permutations

An inversion in a permutation $\tau = \tau_1 \dots \tau_n$ is a pair (i, j) such that $1 \leq i < j \leq n$ and $\tau_i > \tau_j$ [7].

9.1.1 InversionAut

▷ `InversionAut(`*k*`)` (function)

Returns: An automaton that accepts all permutations with exactly `k` inversions.

The rational language of all permutations with a given number , `k`, of inversions is computed by `InversionAut`.

Example

```
gap> a:=InversionAut(1);
< deterministic automaton on 2 letters with 4 states >
gap> AutToRatExp(a);
a*baa*
gap> Spectrum(a);
[ 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14 ]
gap> b:=InversionAut(5);
< deterministic automaton on 6 letters with 14 states >
gap> AutToRatExp(b);
((a*ba*bUa*c)a*bUa*ba*cUa*d)a*(ba*baa*Ucaaa*)U(a*ba*bUa*c)a*(ca*baa*Udaaaa*)U(\
a*ba*daUa*eaa)a*baa*Ua*ba*(dbUeaa)aaa*U(a*eabUa*(ebUfaa)a)aaa*
gap> Spectrum(b);
[ 0, 0, 0, 3, 22, 71, 169, 343, 628, 1068, 1717, 2640, 3914, 5629, 7889 ]
gap>
```

9.1.2 InversionAutOfClass

▷ `InversionAutOfClass(`*aut, inv*`)` (function)

Returns: An automaton accepting all permutations of a class with `inv` inversions.

InversionAutOfClass intersects the rational pattern class with the rational language containing all permutations under the rank encoding that have exactly inv inversions.

Example

```
gap> a:=MinimalAutomaton(GraphToAut(Seqstacks(2,2),1,6));
< deterministic automaton on 3 letters with 3 states >
gap> Spectrum(a);
[ 1, 2, 6, 18, 54, 162, 486, 1458, 4374, 13122, 39366, 118098, 354294,
  1062882, 3188646 ]
gap> b:=InversionAutOfClass(a,4);
< deterministic automaton on 5 letters with 23 states >
gap> Spectrum(b);
[ 0, 0, 0, 3, 13, 35, 75, 140, 238, 378, 570, 825, 1155, 1573, 2093 ]
gap>
```

9.2 Plus- and Minus-(In)Decomposablilty

9.2.1 PlusDecomposableAut

▷ PlusDecomposableAut(*aut*) (function)

Returns: An automaton that accepts the subset of the class aut containing the plus-decomposable permutations of aut.

The PlusDecomposableAut automaton accepts the language of all plus-decomposable permutations of the encoded class accepted by aut.

Example

```
gap> xa:=MinimalAutomaton(GraphToAut(Parstacks(2,2),1,6));
< deterministic automaton on 4 letters with 9 states >
gap> Spectrum(xa);
[ 1, 2, 6, 23, 89, 345, 1338, 5189, 20122, 78024, 302529, 1172993, 4547973,
  17633432, 68368135 ]
gap> a:=PlusDecomposableAut(xa);
< deterministic automaton on 4 letters with 16 states >
gap> Spectrum(a);
[ 0, 1, 3, 11, 47, 196, 808, 3306, 13433, 54265, 218145, 873303, 3483654,
  13853682, 54945158 ]
gap>
```

9.2.2 PlusIndecomposableAut

▷ PlusIndecomposableAut(*aut*) (function)

Returns: An automaton that accepts all permutations of aut that are not plus-decomposable.

The PlusIndecomposableAutomaton automaton accepts the language of all plus-indecomposable permutations of the encoded class accepted by aut, by rejecting every rank encoding that in the original automaton would have entered and left the accept state before the last letter in the rank encodedpermutation.

Example

```
gap> xa:=MinimalAutomaton(GraphToAut(Parstacks(2,2),1,6));
< deterministic automaton on 4 letters with 9 states >
gap> Spectrum(xa);
[ 1, 2, 6, 23, 89, 345, 1338, 5189, 20122, 78024, 302529, 1172993, 4547973,
  17633432, 68368135 ]
```

```
gap> a:=PlusIndecomposableAut(xa);
< deterministic automaton on 4 letters with 11 states >
gap> Spectrum(a);
[ 1, 1, 3, 12, 42, 149, 530, 1883, 6689, 23759, 84384, 299690, 1064319,
  3779750, 13422977 ]
gap>
```

9.2.3 MinusDecomposableAut

▷ MinusDecomposableAut(*aut*) (function)

Returns: An automaton that accepts the subset of the class aut containing the minus-decomposable permutations of aut.

The MinusDecomposableAut automaton accepts the language of all minus-decomposable permutations of the rank encoded class accepted by aut.

Example

```
gap> xa:=MinimalAutomaton(GraphToAut(Parstacks(2,2),1,6));
< deterministic automaton on 4 letters with 9 states >
gap> Spectrum(xa);
[ 1, 2, 6, 23, 89, 345, 1338, 5189, 20122, 78024, 302529, 1172993, 4547973,
  17633432, 68368135 ]
gap> a:=MinusDecomposableAut(xa);
< deterministic automaton on 4 letters with 12 states >
gap> Spectrum(a);
[ 0, 1, 3, 10, 24, 64, 180, 520, 1524, 4504, 13380, 39880, 119124, 356344,
  1066980 ]
gap>
```

9.2.4 MinusIndecomposableAut

▷ MinusIndecomposableAut(*aut*) (function)

Returns: An automaton that accepts all permutations of aut that are not minus-decomposable.

The MinusIndecomposableAut automaton accepts the language of all minus-indecomposable permutations of the encoded class accepted by aut, which is the complement set of the set of minus-decomposable permutations within the class.

Example

```
gap> xa:=MinimalAutomaton(GraphToAut(Parstacks(2,2),1,6));
< deterministic automaton on 4 letters with 9 states >
gap> Spectrum(xa);
[ 1, 2, 6, 23, 89, 345, 1338, 5189, 20122, 78024, 302529, 1172993, 4547973,
  17633432, 68368135 ]
gap> a:=MinusIndecomposableAut(xa);
< deterministic automaton on 4 letters with 17 states >
gap> Spectrum(a);
[ 1, 1, 3, 13, 65, 281, 1158, 4669, 18598, 73520, 289149, 1133113, 4428849,
  17277088, 67301155 ]
gap>
```

9.3 Language of all non-simple permutations

The regular language of all non-simple rank encoded permutations with highest rank k is described by the following equation,

$$E(NS_k) = E(\Omega_k) \cap (\bigcup_{l=1}^{k-1} P_l \bigcup_{m=l}^{k-1} E(\hat{\Omega}_{k-m})^{+m} \cup \bigcup_{j=1}^{k-1} E(\hat{\Omega}_{k-j})^{+j} \cup$$

$$\cup \bigcup_{a=2}^{k-1} \bigcup_{b=0}^{k-1-a} Q_{a,b} \bigcup_{i=0}^{a-2} (E(\hat{\Omega}_{k-(b+i)})^{+b+i})^{(a-i)}) \Sigma^* \cup E(\mathscr{D}_P(\Omega_k))$$

where Σ is the alphabet $\{1,\ldots,k\}$, $k \in \mathbb{N}$, $k \geq 3$.

P_l is the language of prefixes of rank encoded permutations, where the prefix ends with the total sum of gap sizes to be equal to l.

$Q_{i,j}$ is the language of prefixes of rank encoded permutations, where the prefix ends with a gap of size i and the sum of the sizes of gaps below equals to j.

$E(\Omega_{k-i})^{+i}$ is the language of $E(\Omega_{k-i})$ $i \in \mathbb{N}$, with the alphabet shifted upwards by i.

$E(\Omega_k)^{(i)}$ is the sublanguage of $E(\Omega_k)$ containing the words of length $\leq i$, $i \in \mathbb{N}$.

$E(\hat{\Omega}_k)$ is the sublanguage of $E(\Omega_k)$ excluding the words of length ≤ 1.

$E(\mathscr{D}_P(\Omega_k))$ is the language of all plus-decomposable permutations as described in [8].

9.3.1 LengthBoundAut

▷ `LengthBoundAut(aut, min, i, k)` (function)

Returns: The subautomaton of `aut` that accepts words between (and including) the lengths `min` and `i`.

We are taking the automaton `aut` and it's alphabet `k`, and find the automaton that accepts all words of `aut` of length between (and including) `min` and `i`.

Example

```
gap> a:=BoundedClassAutomaton(4);
< deterministic automaton on 4 letters with 4 states >
gap> Spectrum(a);
[ 1, 2, 6, 24, 96, 384, 1536, 6144, 24576, 98304, 393216, 1572864, 6291456,
  25165824, 100663296 ]
gap> LengthBoundAut(a,4,8,4);
< deterministic automaton on 4 letters with 22 states >
gap> Spectrum(last);
[ 0, 0, 0, 24, 96, 384, 1536, 6144, 0, 0, 0, 0, 0, 0, 0 ]
gap>
```

9.3.2 ShiftAut

▷ `ShiftAut(i, k)` (function)

Returns: The automaton Ω_{k-i}^{+i}.

We are shifting the alphabet of Ω_{k-i} in their values by i to expand to the alphabet $\{1,\ldots,k\}$, but keeping the automaton structure of Ω_{k-i}.

Example

```
gap> ShiftAut(2,4);
< non deterministic automaton on 4 letters with 4 states >
gap> Display(last);
   |  1       2       3       4
-----------------------------------
 a |
 b |
 c | [ 2 ]   [ 4 ]   [ 4 ]   [ 4 ]
 d | [ 3 ]   [ 3 ]   [ 3 ]   [ 3 ]
Initial state:   [ 1 ]
Accepting state: [ 4 ]
gap> ShiftAut(1,4);
< non deterministic automaton on 4 letters with 5 states >
gap> Display(last);
   |  1       2       3       4       5
-------------------------------------------
 a |
 b | [ 2 ]   [ 5 ]   [ 5 ]   [ 3 ]   [ 5 ]
 c | [ 3 ]   [ 3 ]   [ 3 ]   [ 3 ]   [ 3 ]
 d | [ 4 ]   [ 4 ]   [ 4 ]   [ 4 ]   [ 4 ]
Initial state:   [ 1 ]
Accepting state: [ 5 ]
gap>
```

9.3.3 NextGap

▷ NextGap(*gap, rank*) (function)

Returns: A list of gap sizes.

Knowing the current available gap sizes gap, which are the number of available spaces in a permutation plot. These gaps are separated by blocks where there are already points inserted. We determine where the next point (known to us as its rank) is being placed and what the next gap sizes are.

Example

```
gap> NextGap([1,1],2);
[ 1 ]
gap> NextGap([1],3);
[ 1, 1 ]
gap> NextGap([2,1],4);
[ 2, 1 ]
gap>
```

9.3.4 GapAut

▷ GapAut(*k*) (function)

Returns: The non-deterministic automaton accepting the rank encoded language of Ω_k and the list of all possible gap sizes.

The automaton accepts the rank encoded permutations of Ω_k, but the automaton is slightly extended through having each state corresponding to a gap size and the start state being the emptyset of gap sizes. The transitions of the automaton are determined through the knowledge of the available

spaces and the rank. This is calculated in NextGap. Please note that the index of the gap sizes in the list corresponds to the state of the automaton.

Example

```
gap> GapAut(3);
[ < non deterministic automaton on 3 letters with 5 states >,
  [ [  ], [ 0 ], [ 1 ], [ 2 ], [ 1, 1 ] ] ]
gap> Display(last[1]);
   |  1       2       3       4       5
-------------------------------------------
 a | [ 2 ]   [ 2 ]   [ 2 ]   [ 3 ]   [ 3 ]
 b | [ 3 ]   [ 3 ]   [ 3 ]   [ 3 ]   [ 3 ]
 c | [ 4 ]   [ 4 ]   [ 5 ]   [ 4 ]   [ 5 ]
Initial state:    [ 1 ]
Accepting states: [ 1, 2 ]
gap>
```

9.3.5 SumAut

▷ SumAut(*sum, k*) (function)

Returns: The automaton accepting the language P_{sum}.

This automaton is based on the GapAut where the accept states are chosen by their gap sizes, namely if the total sum of gap sizes equal to sum.

Example

```
gap> SumAut(2,3);
< non deterministic automaton on 3 letters with 5 states >
gap> Display(last);
   |  1       2       3       4       5
-------------------------------------------
 a | [ 2 ]   [ 2 ]   [ 2 ]   [ 3 ]   [ 3 ]
 b | [ 3 ]   [ 3 ]   [ 3 ]   [ 3 ]   [ 3 ]
 c | [ 4 ]   [ 4 ]   [ 5 ]   [ 4 ]   [ 5 ]
Initial state:    [ 1 ]
Accepting states: [ 4, 5 ]
gap>
```

9.3.6 GapSumAut

▷ GapSumAut(*gap, sum, k*) (function)

Returns: The automaton accepting the language $Q_{gap,sum}$.

This automaton is based on the GapAut where the accept states are chosen by their gap sizes, namely if there is a gap size gap and the gap sizes before have a total sum of sum.

Example

```
gap> GapSumAut(1,0,3);
< non deterministic automaton on 3 letters with 5 states >
gap> Display(last);
   |  1       2       3       4       5
-------------------------------------------
 a | [ 2 ]   [ 2 ]   [ 2 ]   [ 3 ]   [ 3 ]
 b | [ 3 ]   [ 3 ]   [ 3 ]   [ 3 ]   [ 3 ]
 c | [ 4 ]   [ 4 ]   [ 5 ]   [ 4 ]   [ 5 ]
Initial state:    [ 1 ]
```

```
Accepting states: [ 3, 5 ]
gap>
```

9.3.7 NonSimpleAut

▷ `NonSimpleAut(k)` (function)

Returns: The automaton accepting all rank encoded non-simple permutations with rank encoding `k`.

We find the language of all non-simple permutations of the set of all k rank encoded permutations Ω_k using the above equation.

Example

```
gap> a:=NonSimpleAut(3);
< deterministic automaton on 3 letters with 9 states >
gap> Display(a);
   |  1  2  3  4  5  6  7  8  9
--------------------------------
 a |  1  3  1  5  3  1  6  3  3
 b |  3  3  3  3  9  9  3  9  3
 c |  2  2  2  2  4  4  2  7  4
Initial state:   [ 8 ]
Accepting state: [ 1 ]
gap>
```

9.4 Simplicity

The set of simple permutations of a class is the complement set of the class with the non-simple permutations. We are working in the rank encoding and so in language terms our set of simple permutations S_k will be the subset of Ω_k

$$E(S_k) = E(\Omega_k \setminus NS_k) = E(\Omega_k) \setminus E(NS_k) = E(\Omega_k) \cap E(NS_k)^C$$

9.4.1 SimplePermAut

▷ `SimplePermAut(k)` (function)

Returns: The automaton accepting all rank encoded simple permutations with highest rank encoding `k`.

We find the language of all simple permutations of the set of all k rank encoded permutations Ω_k using the above equation.

Example

```
gap> SimplePermAut(3);
< deterministic automaton on 3 letters with 8 states >
gap> Display(last);
   |  1  2  3  4  5  6  7  8
----------------------------
 a |  2  2  1  1  7  2  1  6
 b |  2  2  4  2  2  4  4  2
 c |  2  2  8  5  2  5  5  2
Initial state:    [ 3 ]
Accepting states: [ 1, 3 ]
gap>
```

9.5 Exceptionality

A permutation is said to be exceptional if it is of one of the following forms,

$$246\ldots(2m)135\ldots(2m-1)$$

$$(2m-1)(2m-3)\ldots1(2m)(2m-2)\ldots2$$

$$(m+1)1(m+2)2(m+3)3\ldots(2m)m$$

$$m(2m)(m-1)(2m-1)\ldots1(m+1)$$

where $m \geq 2$ [4].

9.5.1 IsExceptionalPerm

▷ `IsExceptionalPerm(`*perm*`)` (function)

Returns: `True` if `perm` is exceptional, `False` otherwise.

The functions checks whether `perm` is one of the 4 types of exceptional permutations, that are described above.

Example

```
gap> IsExceptionalPerm([1,2,5,3,4]);
false
gap> IsExceptionalPerm([1,1,3,1,1]);
false
gap> IsExceptionalPerm([2,4,6,1,3,5]);
true
gap> IsExceptionalPerm([2,3,4,1,1,1]);
true
gap>
```

9.5.2 ExceptionalBoundedAutomaton

▷ `ExceptionalBoundedAutomaton(`*k*`)` (function)

Returns: The automaton which accepts all exceptional permutations with highest rank encoding `k`.

The language of `k` rank encoded exceptional permutations will be finite, and so it regular.

Example

```
gap> ExceptionalBoundedAutomaton(8);
< deterministic automaton on 8 letters with 41 states >
gap> Spectrum(last,20);
[ 0, 2, 0, 2, 0, 4, 0, 4, 0, 2, 0, 2, 0, 2, 0, 0, 0, 0, 0, 0 ]
gap> ExceptionalBoundedAutomaton(5);
< deterministic automaton on 5 letters with 21 states >
gap> Spectrum(last);
[ 0, 2, 0, 2, 0, 4, 0, 2, 0, 0, 0, 0, 0, 0, 0 ]
gap>
```

Chapter 10

Miscellaneous functions

This temporary chapter is dedicated to miscellaneous functions that are relevant to some specific ongoing research questions.

10.1 Automaton Manipulation

10.1.1 LoopFreeAut

▷ `LoopFreeAut(aut)` (function)

Returns: An automaton without any loops of length 1.

`LoopFreeAut` builds the subautomaton of `aut` that does not contain any loops of length 1, except for the sink state.

Example

```
gap> a:=Automaton("det",4,3,[[2,4,3,3],[4,4,1,4],[3,1,2,4]],[1],[2]);
< deterministic automaton on 3 letters with 4 states >
gap> Display(a);
   |  1  2  3  4
-----------------
 a |  2  4  3  3
 b |  4  4  1  4
 c |  3  1  2  4
Initial state:   [ 1 ]
Accepting state: [ 2 ]
gap> b:=LoopFreeAut(a);
< deterministic automaton on 3 letters with 5 states >
gap> Display(b);
   |  1  2  3  4  5
--------------------
 a |  2  4  5  3  5
 b |  4  4  1  5  5
 c |  3  1  2  5  5
Initial state:   [ 1 ]
Accepting state: [ 2 ]
gap>
```

10.1.2 LoopVertexFreeAut

▷ `LoopVertexFreeAut(aut)` (function)

Returns: An automaton without any vertices that had loops of length 1.

`LoopVertexFreeAut` builds the subautomaton that does not contain the vertices and transitions of vertices in `aut` that have loops of length 1. The function minimalises and determinises the automaton before returning it, which might change the numbering on the vertices, but the returned automaton is isomorphic to the subautomaton of `aut`.

Example

```
gap> a:=Automaton("det",4,3,[[2,4,3,3],[4,4,1,4],[3,1,2,4]],[1],[2]);
< deterministic automaton on 3 letters with 4 states >
gap> Display(a);
   |  1  2  3  4
-----------------
 a |  2  4  3  3
 b |  4  4  1  4
 c |  3  1  2  4
Initial state:   [ 1 ]
Accepting state: [ 2 ]
gap> b:=LoopVertexFreeAut(a);
< deterministic automaton on 3 letters with 3 states >
gap> Display(b);
   |  1  2  3
--------------
 a |  2  2  1
 b |  2  2  2
 c |  3  2  2
Initial state:   [ 3 ]
Accepting state: [ 1 ]
gap>
```

Index

References

[1] M. Albert, M. Atkinson, and N. Ruškuc, "Regular closed sets of permutations," *Theoretical Computer Science*, vol. 306, pp. 85 – 100, 2003. 2, 5, 10, 23

[2] M. Atkinson, M. Livesey, and D. Tulley, "Permutations generated by token passing in graphs," *Theoretical Computer Science*, vol. 178, pp. 103 – 118, n.d. 2, 5, 6, 12

[3] T. Uno and M. Yagiura, "Fast algorithms to enumerate all common intervals of two permutations," *Algorithmica*, vol. 26, p. 2000, 2000. 35

[4] A. Pierrot and D. Rossin, "Simple permutations poset," *ArXiv e-prints*, pp. 1–15, Jan. 2012. 35, 47

[5] R. Brignall, "A survey of simple permutations," *ArXiv e-prints*, June 2008. 36

[6] M. Albert and M. Atkinson, "Simple permutations and pattern restricted permutations," *Discrete Mathematics*, vol. 300, no. 1–3, pp. 1 – 15, 2005. 36

[7] A. Claesson, V. Jelínek, and E. Steingrímsson, "Upper bounds for the Stanley-Wilf limit of 1324 and other layered patterns," *ArXiv e-prints*, Nov. 2011. 40

[8] R. Hoffmann and S. Linton, "Regular Languages of Plus- and Minus- (In)Decomposable Permutations," *Pure Mathematics and Applications (to appear)*, 2013. 43