DeepThought

This package provides functions for computations in finitely generated nilpotent groups based on the Deep Thought algorithm.

1.0.2

13 September 2018

Nina Wagner

Max Horn

Nina Wagner
Email: nina.wagner@math.uni-giessen.de
Address: AG Algebra
Mathematisches Institut
Justus-Liebig-Universität Gießen
Arndtstraße 2
35392 Gießen
Germany

Max Horn
Email: max.horn@math.uni-giessen.de
Homepage: http://www.quendi.de/math
Address: AG Algebra
Mathematisches Institut
Justus-Liebig-Universität Gießen
Arndtstraße 2
35392 Gießen
Germany

Contents

Chapter 1

The Deep Thought algorithm

Polycyclic and, especially, finitely generated nilpotent groups exhibit a rich structure allowing a special approach towards multiplication using polynomials. The so-called Deep Thought algorithm introduced in [LGS98] computes these polynomials in practice for a suitable presentation of a finitely generated nilpotent group. Such a presentation is of the following form

$$\langle a_1,\ldots,a_n \mid a_i^{s_i} = a_{i+1}^{c_{i,i,i+1}}\cdots a_n^{c_{i,i,n}}, 1\le i\le n, a_j a_i = a_i a_j a_{j+1}^{c_{i,j,j+1}}\cdots a_n^{c_{i,j,n}}, 1\le i<j\le n\rangle$$

with $s_i \in \mathbb{N}\cup\{\infty\}$ and $c_{i,j,k}\in\mathbb{Z}$. If $s_i=\infty$, then the power relation $a_i^{s_i}$ is skipped.

Let G denote the group presented by the above presentation. Then every element in G can be written uniquely in a so-called normal form. That is, if $G_i := \langle a_i,\ldots,a_n\rangle$ and $r_i := |G_i : G_{i+1}|$, $1\le i\le n$, are the relative orders, then for certain $e_i\in\mathbb{Z}$ each element can be written as

$$a_1^{e_1}\cdots a_n^{e_n}$$

with $0\le e_i<r_i$ if $r_i<\infty$. A presentation is called confluent if and only if $s_i=r_i$ for all $1\le i\le n$. If a presentation is not confluent, not all functions provided in this package are applicable, see function `DTP_DTapplicability`. For more theoretical background see for example the documentation of the GAP package Polycyclic.

The Deep Thought algorithm computes rational polynomials $f_1,\ldots,f_n$ in $2n$ indeterminates such that if $x := a_1^{x_1}\cdots a_n^{x_n}$ and $y := a_1^{y_1}\cdots a_n^{y_n}$ are two elements (in normal form or not with $x_1,\ldots,x_n,y_1,\ldots,y_n\in\mathbb{Z}$), then their product xy is given by

$$a_1^{f_1(x_1,\ldots,x_n,y_1,\ldots,y_n)}\cdots a_n^{f_n(x_1,\ldots,x_n,y_1,\ldots,y_n)}.$$

If the collector is confluent, also the normal form of the product can be computed. Otherwise this is not possible. Moreover, there exists a second version of the Deep Thought algorithms which computes n^2 polynomials f_{rs}, $1\le r,s\le n$, suitable for multiplications of the form

$$(a_1^{x_1}\cdots a_n^{x_n})\cdot a_s^{y_s} = a_1^{f_{1s}(x_1,\ldots,x_n,y_s)}\cdots a_n^{f_{ns}(x_1,\ldots,x_n,y_s)}$$

for $1\le s\le n$. Each general multiplication can be expressed using these special multiplications. Depending on the presentation, polynomials of one version may be more efficient for computations than the polynomials of the other version. For a suggestion on which polynomials to use for a given presentation, see `DTP_DTapplicability`. In the following, Deep Thought type f_r refers to the Deep Thought algorithm which uses n polynomials and type f_{rs} refers to the Deep Thought algorithm using n^2 polynomials.

In order to work with the Deep Thought functions, the group presentation is expected to be given as a collector `coll`, as defined in the GAP package Polycyclic. Moreover, the Polycyclic package introduces the structure of exponent vectors `expvec`, which will be used also in this package to represent group elements. In the following text, a group element $a_1^{x_1} \cdots a_n^{x_n}$ is identified with its exponent vector in form of the list `[x_1, ..., x_n]`. For example, if `expvec1, expvec2` are exponent vectors of elements in the same group, then `expvec1 * expvec2` describes the multiplication of the corresponding group elements, and so on. Note that generally exponent vectors are not assumed to represent normal forms.

1.1 Category DTObj

This package uses the category `DTObj`. A `DTObj` is a `IsFromTheLeftCollectorRep` with certain further list entries to store the Deep Thought polynomials of a collector together with some additional information. That is, the functions `DTP_DTpols_r` and `DTP_DTpols_rs` return a `DTObj` which has entries as `IsFromTheLeftCollectorRep` and additionally:

- `DTObj![PC_DTPPolynomials]`: Deep Thought polynomials in form of (nested) lists
- `DTObj![PC_DTPOrders]`: list containing orders of group generators if the collector is confluent
- `DTObj![PC_DTPConfluent]`: boolean value indicating whether the collector is confluent or not

Chapter 2

Using Deep Thought functions

In the following sections, functions provided for computing Deep Thought polynomials and using them for calculations are listed.

2.1 Computing Deep Thought polynomials

2.1.1 DTP_DTapplicability

▷ `DTP_DTapplicability(`*`coll`*`)` (function)

Returns: boolean

Checks the collector `coll` for applicability of Deep Thought functions. Note that depending on confluency some functions may be applicable, while others are not. Information on the applicability and which type of Deep Thought polynomials are suggested is printed to the terminal. Here, "+" means that the following property is fulfilled, otherwise there is a "-". The function returns `false` if Deep Thought is not applicable to the collector `coll` and `true` otherwise. Anyway, even if `true` is returned, *not all functions need to be applicable* (in case of inconfluencies).

2.1.2 DTP_DTObjFromCollector

▷ `DTP_DTObjFromCollector(`*`coll[, rs_flag]`*`)` (function)

Returns: a DTObj

Computes a DTObj for the collector coll, either with polynomials of type f_{rs} (if `rs_flag = true`) or with polynomials of type f_r, if `rs_flag = false`. If the optional argument `rs_flag` is not provided, polynomials of type f_{rs} are computed. The function checks whether the collector `coll` is confluent. If not, a warning is displayed. Note that the function assumes the collector `coll` to be suitable for Deep Thought, see function `DTP_DTapplicability`.

Example

```
gap> G := UnitriangularPcpGroup(10, 0);;
gap> coll := Collector(G);;
gap> DTP_DTapplicability(coll);
Checking collector for DT-applicability. "+" means the following property
is fulfilled.
+   conjugacy relations
+   power relations
+   confluent
Suggestion: Call DTP_DTObjFromColl with rs_flag = true.
```

```
true
# calling DTP_DTObjFromCollector without rs_flag implies rs_flag = true:
gap> DTObj := DTP_DTObjFromCollector(coll);
<DTObj>
```

2.2 Computations with Deep Thought polynomials

2.2.1 DTP_Exp

▷ DTP_Exp(*expvec, int, DTObj*) (function)

Returns: an exponent vector

Computes the exponent vector of $\texttt{expvec}^{int}$. If IsConfluent(DTObj) = true, then the result is in normal form.

2.2.2 DTP_Inverse

▷ DTP_Inverse(*expvec, DTObj*) (function)

Returns: an exponent vector

Computes the exponent vector of the inverse of the element corresponding to expvec. If IsConfluent(DTObj) = true, then the result is in normal form.

2.2.3 DTP_IsInNormalForm

▷ DTP_IsInNormalForm(*expvec, coll*) (function)

Returns: boolean or positive integer

Checks whether expvec is in normal form or not. If yes, the return value is true. Otherwise the return value is the smallest generator index for which the normal form condition is violated, i.e. for which the relative order RelativeOrder(coll)[i] is non-zero, and expvec[i] < 0 or expvec[i] $\geq$ RelativeOrder(coll)[i].

2.2.4 DTP_Multiply

▷ DTP_Multiply(*expvec1, expvec2, DTObj*) (function)

Returns: an exponent vector

Computes the exponent vector of the product expvec1 * expvec2 using the Deep Thought polynomials. If IsConfluent(DTObj) = true, then the result is returned in normal form. DTP_Multiply either calls DTP_Multiply_r or DTP_Multiply_rs depending on which type of polynomials are stored in DTObj.

2.2.5 DTP_Multiply_r

▷ DTP_Multiply_r(*expvec1, expvec2, DTObj*) (function)

Returns: an exponent vector

Computes the exponent vector of the product expvec1 * expvec2 using the Deep Thought polynomials of type f_r stored in DT_Obj. If IsConfluent(DTObj) = true, then the result is returned in normal form.

2.2.6 DTP_Multiply_rs

▷ `DTP_Multiply_rs(`*expvec1, expvec2, DTObj*`)` (function)

Returns: an exponent vector

Computes the exponent vector of the product `expvec1 * expvec2` using the Deep Thought polynomials of type f_{rs} stored in `DT_Obj`. If `IsConfluent(DTObj) = true`, then the result is returned in normal form.

2.2.7 DTP_NormalForm

▷ `DTP_NormalForm(`*expvec, DTObj*`)` (function)

Returns: an exponent vector

Computes the exponent vector of the normal form of `expvec`. For this function to be applicable, we need `IsConfluent(DTObj) = true`.

2.2.8 DTP_Order

▷ `DTP_Order(`*expvec, DTObj*`)` (function)

Returns: positive integer or infinity

Computes the order of the element described by `expvec`. For this function to be applicable, we need `IsConfluent(DTObj) = true`.

2.2.9 DTP_SolveEquation

▷ `DTP_SolveEquation(`*expvec1, expvec2, DTObj*`)` (function)

Returns: an exponent vector

Computes the exponent vector of the element corresponding to `expvec1`$^{-1}$ `* expvec2`, i.e. the result solves the equation `expvec1 * result = expvec2`. If `IsConfluent(DTObj) = true`, then the result describes a normal form.

Example

```
gap> G := PcGroupToPcpGroup(SmallGroup(23^5, 2));
Pcp-group with orders [ 23, 23, 23, 23, 23 ]
gap> coll := Collector(G);
<<from the left collector with 5 generators>>
gap> DTObj := DTP_DTObjFromCollector(coll);
<DTObj>
gap> g := [100, 134, -31, 52, 5235];
[ 100, 134, -31, 52, 5235 ]
gap> DTP_IsInNormalForm(g, DTObj);
1
gap> g := DTP_NormalForm(g, DTObj);
[ 8, 19, 15, 10, 19 ]
gap> DTP_IsInNormalForm(g, DTObj);
true
gap> DTP_Inverse(g, DTObj);
[ 15, 4, 22, 12, 3 ]
gap> DTP_Order(g, DTObj);
529
gap> h := [142, 2, -41, 23, 1];
[ 142, 2, -41, 23, 1 ]
```

```
gap> DTP_Multiply(g, h, DTObj);
[ 12, 21, 4, 16, 20 ]
```

2.3 Computations with pcp-elements

When Deep Thought polynomials are available, certain computations allow different approaches which may be faster than the methods used by default. In this section, computations for which such extra functions taking pcp-elements as input are available are listed. All of these functions expect the collector belonging to the pcp-elements to be a `DTObj`.

2.3.1 DTP_PCP_Exp

▷ `DTP_PCP_Exp(`*`pcp-element, int`*`)` (function)

Returns: pcp-element

Returns the pcp-element `pcp-element`^{int}. If `IsConfluent(DTObj) = true`, then the result is in normal form.

2.3.2 DTP_PCP_Inverse

▷ `DTP_PCP_Inverse(`*`pcp-element`*`)` (function)

Returns: pcp-element

Returns the pcp-elment `pcp-element^-1`. If `IsConfluent(DTObj) = true`, then the result is in normal form.

2.3.3 DTP_PCP_NormalForm

▷ `DTP_PCP_NormalForm(`*`pcp-element`*`)` (function)

Returns: pcp-element

Returns a pcp-element which is the normal form of the input pcp-element. For this function to be applicable, we need `IsConfluent(DTObj) = true`.

2.3.4 DTP_PCP_Order

▷ `DTP_PCP_Order(`*`pcp-element`*`)` (function)

Returns: positive integer or infinity

Computes the order of the pcp-element. For this function to be applicable, we need `IsConfluent(DTObj) = true`.

2.3.5 DTP_PCP_SolveEquation

▷ `DTP_PCP_SolveEquation(`*`pcp-element1, pcp-element2`*`)` (function)

Returns: pcp-element

Returns the pcp-element `pcp-element1`$^{-1}$ `* pcp-element2`, i.e. the result solves the equation `pcp-element1 * pcp-element = pcp-element2`. If `IsConfluent(DTObj) = true`, then the result describes a normal form.

Example

```
gap> G := HeisenbergPcpGroup(7);;
gap> coll := Collector(G);;
gap> DTObj := DTP_DTObjFromCollector(coll);;
gap> H := PcpGroupByCollector(DTObj);;
gap> g := Random(H);; h := Random(H);;
gap> DTP_PCP_SolveEquation(g, h);
g1^-3*g2^-1*g3^-7*g4*g5^-6*g6*g7*g8^2*g9^3*g11^-4*g12^5*g14^-2*g15^7
g1^-3*g2^-1*g3^-7*g4*g5^-6*g6*g7*g8^2*g9^3*g11^-4*g12^5*g14^-2*g15^7
infinity
gap> g^-1;
g1^-2*g3^-3*g4^-1*g5^-4*g6^2*g7*g8^-3*g10^-3*g11^-1*g12^4*g14^-2*g15^-3
gap> DTP_PCP_Inverse(h);
g1*g2*g3^4*g4^-2*g5^2*g6*g8^-5*g9^-3*g10^-3*g11^3*g12^-1*g15^-33
```

2.4 Accessing Deep Thought polynomials

In this section, functions which can be used to display the content of a `DTObj` are documented. Furthermore, Deep Thought polynomials stored in a `DTObj` can be converted to GAP polynomials.

2.4.1 DTP_Display_DTObj

▷ `DTP_Display_DTObj(`*`DTObj`*`)` (function)

Returns: nothing

Prints information about `DTObj` to the terminal. In particular, the Deep Thought polynomials are printed in human-readable form. This function is also called by the method of `Display` for a `DTObj`.

2.4.2 DTP_pols2GAPpols

▷ `DTP_pols2GAPpols(`*`DTObj`*`)` (function)

Returns: list

Converts the Deep Thought polynomials stored in `DTObj[PC_DTPPolynomials]` to GAP polynomials and returns them in a list together with their polynomial ring.

Example

```
gap> coll := FromTheLeftCollector(4);;
gap> SetConjugate(coll, 2, 1, [2, 1, 3, 2]);
gap> SetConjugate(coll, 3, 1, [3, 1, 4, 1]);
gap> SetConjugate(coll, 3, 2, [3, 1, 4, 5]);
gap> UpdatePolycyclicCollector(coll);
gap> DTObj := DTP_DTObjFromCollector(coll);
<DTObj>
gap> Display(DTObj);
Polynomials f_rs for s = 1:
f_1,s = X_1 + Y_1
f_2,s = X_2
f_3,s = X_3 + 2 * X_2 Y_1
f_4,s = X_4 + X_3 Y_1 + 2 * X_2 Binomial(Y_1, 2) + 10 * Binomial(X_2, 2) Y_1
Polynomials f_rs for s = 2:
f_1,s = X_1
f_2,s = X_2 + Y_2
```

```
f_3,s = X_3
f_4,s = X_4 + 5 * X_3 Y_2
Polynomials f_rs for s = 3:
f_1,s = X_1
f_2,s = X_2
f_3,s = X_3 + Y_3
f_4,s = X_4
Polynomials f_rs for s = 4:
f_1,s = X_1
f_2,s = X_2
f_3,s = X_3
f_4,s = X_4 + Y_4
gap> DTObj := DTP_DTObjFromCollector(coll, false);
<DTObj>
gap> Display(DTObj);
f_1 = X_1 + Y_1
f_2 = X_2 + Y_2
f_3 = X_3 + Y_3 + 2 * X_2 Y_1
f_4 = X_4 + Y_4 + X_3 Y_1 + 2 * X_2 Binomial(Y_1, 2) +
10 * Binomial(X_2, 2) Y_1 + 5 * X_3 Y_2 + 10 * X_2 Y_1 Y_2
gap> DTP_pols2GAPpols(DTObj);
 [ [ x1+y1, x2+y2, 2*x2*y1+x3+y3,
5*x2^2*y1+x2*y1^2+10*x2*y1*y2-6*x2*y1+x3*y1+5*x3*y2+x4+y4 ],
Rationals[x1,x2,x3,x4,y1,y2,y3,y4] ]
```

References

[LGS98] C. Leedham-Green and L. Soicher. Symbolic collection using deep thought. *LMS J. Comp. Math. 1*, pages 9–24, 1998. 3

Index