SmallOverlap

A GAP 4 Package for small overlap monoids and semigroups

Version 0.1

February 2008

Mark Kambites

Mark Kambites
— Email: Mark.Kambites@manchester.ac.uk
— Homepage: http://www.maths.manchester.ac.uk/~mkambites/
— Address: School of Mathematics,
University of Manchester,
Manchester M13 9PL,
England.

Copyright

Contents

Chapter 1

Introduction

SmallOverlap is a GAP package for computing with monoids and semigroups given by finite presentations satisfying small overlap conditions of the kind introduced by Remmers [Rem71]. It includes routines for testing whether presentations satisfy small overlap conditions, and an implementation of a new, highly efficient algorithm to solve the word problem for monoid and semigroup presentations satisfying the condition $C(4)$ [Kam07].

The purposes of the package are twofold. First, it can be transparently employed by users wishing to make use of faster computations in small overlap monoids and semigroups. And secondly, it will be of use to researchers studying small overlap monoids and semigroups for their own sake.

The package is still at an early development stage, and may well contain bugs. If you find one, or have any comments on the package, the author will be very pleased to hear from you.

This manual comprises five chapters. This introduction contains technical information on installing and loading the package. Chapter 2 contains a very brief introduction to the history and theory of small overlap conditions. Chapter 3 is a short example session to illustrate the most important features of the package. Chapter 4 documents all the functionality of the package in detail. Finally, Chapter 5 discusses some performance issues.

1.1 System Requirements and Dependencies

SmallOverlap is written in pure GAP with no use of external libraries, and so should run happily on any platform supported by GAP. The algorithms implemented are highly efficient, so unless you are working with enormous presentations, you should need no more memory or processor power than is usually required by GAP. The package has been developed and tested with GAP 4.4.10, but is likely to work with any GAP 4 release. The online help requires GAPDoc 1.1 or later.

1.2 Obtaining and Installing SmallOverlap

To install SmallOverlap, download the file smalloverlap.tar from Mark's Homepage and unpack it into the `pkg` directory of your GAP hierarchy.

1.3 Loading SmallOverlap

This section describes how to load the SmallOverlap package, and how to make GAP use it implicitly and transparently for computations. First load the package in the usual way:

Example

```
gap> LoadPackage("smalloverlap");
Loading package SmallOverlap (version 0.1)
Mark Kambites, School of Mathematics, University of Manchester
This is an experimental development version and may contain bugs.
All usage is at your own risk.
Please send comments and bug reports to Mark.Kambites@manchester.ac.uk

SmallOverlap routines are now installed for explicit use only; to
enable implicit use call PreferSmallOverlapEqualityTest()

true
```

Now if you want to make GAP use the new small overlap routines whenever it can, just call the function `PreferSmallOverlapEqualityTest`:

Example

```
gap> PreferSmallOverlapEqualityTest();
SmallOverlap routines will now be used (where applicable) for
testing equality in finitely presented semigroups and monoids.
```

You can now use GAP entirely as normal, forgetting that the package is installed. However, computations with finitely presented semigroups and monoids will now use the small overlap equality tester where applicable, which means that many computations will run faster, or succeed where they would otherwise have failed or not terminated. (In the event that you are performing computations with very large numbers of different finitely presented monoids or semigroups which do *not* satisfy $C(4)$, you may experience a slight slowdown caused by the process of checking the $C(4)$ condition for each monoid - see 5.2 for a full discussion.) If you just wish to enjoy the fast computation abilities of the small overlap routines, you need read no further!

Chapter 2

Theory of Small Overlap Presentations

The aim of this chapter is very briefly to give some historical background and some basic definitions needed to make the rest of this manual intelligible to the non-specialist. A detailed introduction to the theory of small overlap conditions is well beyond the scope of this manual, but we provide some references which can be followed up by the reader seeking a more detailed understanding.

2.1 Historical Background

Small overlap conditions are a natural semigroup- and monoid-theoretic analogue of the small cancellation conditions studied in combinatorial group theory. They were introduced and extensively studied in the thesis of Remmers [Rem71]; he developed a geometric theory of small overlap semigroups and monoids (see also [Rem80]) based on a semigroup-theoretic analogue of the van Kampen diagrams employed by group theorists [LS77].

Remmers showed that if a presentation satisfies the condition $C(3)$ then the ratio of the lengths of equivalent words is bounded by a constant dependent only on the presentation. In particular, equivalence classes for such presentations are finite, which immediately gives a theoretical decision algorithm for the word problem by exhaustive enumeration. An accessible exposition of some of Remmers' results can be found in the book of Higgins [Hig92].

For practical purposes, exhaustive enumeration of equivalence classes is not a feasible approach to the word problem, since these classes can be exponential in the size of their elements, and it is not clear from Remmers' work whether there is a more tractable solution. However, recent research of Kambites [Kam07] has yielded a highly practical (linear time) algorithm for the word problem of any presentation satisfying the slightly stronger condition $C(4)$. An implementation of this algorithm forms the core of the SmallOverlap package.

2.2 Basic Definitions

We assume the reader to be familiar with finitely presented monoids and semigroups. Suppose we have a fixed monoid or semigroup presentation. A *relation word* is a word over the generators which forms one side of a relation in the presentation. A *piece* of the presentation is a word which occurs as a factor of two different relation words, or as a factor of one relation word in two different places. (We make the convention that the empty word is always a piece; this is necessary to facilitate a uniform treatment of the free monoid as a small overlap monoid.)

Let n be a positive integer. The presentation is said to *satisfy* $C(n)$ if no relation word can be written as a product of *strictly fewer than n* pieces. The *small overlap class* of the presentation is the smallest positive integer n such that the presentation satisfies $C(n)$, or infinity if the presentation satisfies $C(n)$ for all n, or 0 if the presentation does not satisfy $C(n)$ for any n.

Now let R be a relation word. If the presentation satisfies the condition $C(1)$ then there is a unique relation word T such that $R = T$ or $T = R$ is a relation in the presentation. T is called the *complement relation word* of R.

The *maximal piece prefix* of the relation word R is the (possibly empty) longest prefix of R which is a piece. Similarly, the *maximal piece suffix* of R is the longest suffix of R which is a piece. If the presentation satisfies $C(3)$ then the maximal piece prefix and maximal piece suffix clearly cannot meet, so r admits a factorisation

$$r = XYZ$$

where X is the maximal piece prefix, Z is the maximal piece suffix, and Y is a non-empty word, called the *middle word* of R. If moreover the presentation satisfies the stronger condition $C(4)$, then the middle word Y is not a piece.

Chapter 3

Example Session

This chapter contains a short example session to illustrate some basic features of the SmallOverlap package. We begin, naturally enough, by loading the package:

Example

```
gap> LoadPackage("smalloverlap");
Loading package SmallOverlap (version 0.1)
Mark Kambites, School of Mathematics, University of Manchester
This is an experimental development version and may contain bugs.
All usage is at your own risk.
Please send comments and bug reports to Mark.Kambites@manchester.ac.uk

SmallOverlap routines are now installed for explicit use only; to
enable implicit use call PreferSmallOverlapEqualityTest()

true
```

Next we define a finitely presented monoid to play with, and give some names to its generators. (See [Gro, Chapter 51] for information on defining finitely presented monoids and semigroups in GAP.)

Example

```
gap> f := FreeMonoid(["a", "b", "c"]);
<free monoid on the generators [ a, b, c ]>
gap> gens := GeneratorsOfMonoid(f);; A := gens[1];; B := gens[2];; C := gens[3];;
gap> s := f / [[A^2 * B * C, A * C * B * A]];
<fp monoid on the generators [ a, b, c ]>
gap> a:=GeneratorsOfMonoid(s)[1];;
gap> b:=GeneratorsOfMonoid(s)[2];;
gap> c:=GeneratorsOfMonoid(s)[3];;
```

Now we make our first use of the SmallOverlap package, to compute the small overlap class of our newly defined monoid.

Example

```
gap> SmallOverlapClass(s);
4
```

So the presentation we gave for `s` satisfies the small overlap condition $C(4)$. Note that `SmallOverlapClass` (4.1.2) is implemented as an attribute of finitely presented monoids and semigroups, which means that when once we have computed it, GAP knows it for the rest of the session:

Example
```
gap> s;
<C(4) small overlap fp monoid on the generators [ a, b, c ]>
```

For this presentation, GAP's standard Knuth-Bendix completion algorithm does not terminate, so it does not know how to solve the word problem:

Example
```
gap> a = b;
```

At this point, GAP will enter an infinite loop, and nothing further will happen! (If you are following along with the example in GAP, you will need to interrupt this computation - on most platforms hold down Ctrl and press C to break into the computation, and then type "`quit;`" at the break prompt to return to the normal GAP prompt.)

Now we try the same test, making explicit use of the SmallOverlap routines:

Example
```
gap> C4WordProblem(a, b);
false
```

To enable implicit use of the SmallOverlap routines for equality testing, we use the function `PreferSmallOverlapEqualityTest` (4.3.1):

Example
```
gap> PreferSmallOverlapEqualityTest(true);
SmallOverlap routines will now be used (where applicable) for
testing equality in finitely presented semigroups and monoids.
And you'll know about it.
```

We can now just use the equality test operator (=) and GAP will automatically use the small overlap routines if applicable...

Example
```

gap> a = b;
(using SmallOverlap equality test)
false
gap> a*a*b*c*a*b*c = a*a*b*c*c*b*a;
(using SmallOverlap equality test)
true
```

The output "`(using SmallOverlap equality test)`" is because we passed the argument `true` to `PreferSmallOverlapEqualityTest` (4.3.1). If we had instead used the argument `false` (or no argument), the selection of small overlap routines would have been entirely transparent.

Note that the selection of the small overlap equality tester is *not* dependent on the fact that we had previously called `SmallOverlapClass` (4.1.2) to check the small overlap class of the presentation. Had we not done so, it would have been checked automatically.

For semigroups and monoids not satisfying C(4), the equality tester reverts to whatever GAP would have used had the package not been installed (usually Knuth-Bendix completion, although it could be something different if you have other packages installed or the semigroup or monoid has other special properties).

Example

```
gap> t := f / [[A^2, B]];
<fp monoid on the generators [ a, b, c ]>
gap> GeneratorsOfMonoid(t)[1] = GeneratorsOfMonoid(t)[2];
(using GAP's default (Knuth-Bendix?) equality test)
false
```

In this case GAP reverts to the usual Knuth-Bendix method (which, fortunately, works this time!). This is because....

Example

```
gap> IsC4(t);
false
gap> SmallOverlapClass(t);
2
```

....the presentation we have given for `t` does not have sufficiently good small overlap properties for the new routines to be applicable.

Chapter 4

Reference

This chapter documents in detail the new attributes and functions provided by the SmallOverlap; examples of the use of the most important functions can be found in the example session in Chapter 3.

4.1 Attributes for Monoids and Semigroups

This section documents new attributes and filters defined by the package for finitely presented monoids and semigroups.

4.1.1 IsFpMonoidOrSemigroup

◊ `IsFpMonoidOrSemigroup(`*S*`)` (filter)

The filter `IsFpMonoidOrSemigroup` is `true` if the argument *S* is either a finitely presented monoid or a finitely presented semigroup, and `false` otherwise. It is the disjunction of the standard GAP filters `IsFpMonoid` and `IsFpSemigroup` (see [Gro, Chapter 51]).

4.1.2 SmallOverlapClass

◊ `SmallOverlapClass(`*S*`)` (attribute)

The `SmallOverlapClass` of a finitely presented monoid or semigroup *S*, is the greatest positive integer *n* such that the presentation defining *S* satisfies the small overlap condition $C(n)$. It returns `infinity` if the presentation satisfies $C(n)$ for all *n*, and 0 if the presentation does not satisfy $C(n)$ for any *n*.

4.1.3 IsC4, IsC3, IsC2, IsC1

◊ `IsC4(`*S*`)` (filter)
◊ `IsC3(`*S*`)` (filter)
◊ `IsC2(`*S*`)` (filter)
◊ `IsC1(`*S*`)` (filter)

The filters IsC4, IsC3, IsC2, IsC1 indicate whether the defining presentation for the finitely presented monoid or semigroup S satisfies the small overlap conditions $C(4)$, $C(3)$, $C(2)$ and $C(1)$ respectively. (They are currently implemented using the attribute SmallOverlapClass (4.1.2); they could in principle be made more efficient by using dedicated code, but for "everyday"-sized presentations SmallOverlapClass (4.1.2) is so fast that it makes no difference.)

4.1.4 RelationWords

◇ RelationWords(*S*) (attribute)

The attribute RelationWords of the finitely presented semigroup or monoid S is a list of the relation words in the defining presentation in their letter representations (see [Gro, Section 35.6]). Relation words paired in the presentation occur as adjacent elements of the list; given the index of a relation word in the list, the index of complement relation word can be retrieved with ComplementRelationWord (4.3.4).

4.1.5 RelationWordLengths

◇ RelationWordLengths(*S*) (attribute)

The attribute RelationWordLengths of the finitely presented semigroup or monoid S is a list of the lengths of the relation words in the defining presentation, in an order corresponding to RelationWords (4.1.4).

4.1.6 MaxPiecePrefixLengths, MiddleWordLengths, MaxPieceSuffixLengths

◇ MaxPiecePrefixLengths(*S*) (attribute)
◇ MiddleWordLengths(*S*) (attribute)
◇ MaxPieceSuffixLengths(*S*) (attribute)

These attributes of a finitely presented monoid or semigroup S are lists containing respectively the length of the longest piece prefix, length of the middle word, and length of the longest piece suffix of each relation word, in an order corresponding to RelationWords (4.1.4).

4.1.7 MaxPieceSuffixLengthBound

◇ MaxPieceSuffixLengthBound(*S*) (attribute)

This is the length of the longest maximal piece suffix of any relation word in the defining presentation for the finitely presented monoid or semigroup S. It is useful for estimating buffer sizes required for certain computations.

4.2 Attributes for Elements of Monoids and Semigroups

This section documents new attributes and filters defined by the package for elements of monoids and semigroups.

4.2.1 IsElementOfFpMonoidOrSemigroup

◊ IsElementOfFpMonoidOrSemigroup(*u*) (filter)

The filter IsElementOfFpMonoidOrSemigroup is true if the argument *u* is an element of a finitely presented monoid or semigroup and false otherwise. It is the disjunction of the standard GAP filters IsElementOfFpMonoid and IsElementOfFpSemigroup (see [Gro, Chapter 51]).

4.2.2 IsElementOfC4, IsElementOfC3, IsElementOfC2, IsElementOfC1

◊ IsElementOfC4(*u*) (filter)
◊ IsElementOfC3(*u*) (filter)
◊ IsElementOfC2(*u*) (filter)
◊ IsElementOfC1(*u*) (filter)

The filters IsElementOfC4, IsElementOfC3, IsElementOfC2, IsElementOfC1 indicate whether the defining presentation for the containing monoid or semigroup of *u* satisfies the small overlap conditions $C(4)$, $C(3)$, $C(2)$ and $C(1)$ respectively.

4.3 Functions

This section documents functions defined by the package.

4.3.1 PreferSmallOverlapEqualityTest

◊ PreferSmallOverlapEqualityTest(*[verbose]*) (function)

A call to the function to PreferSmallOverlapEqualityTest installs new methods for the equality test operator for elements of finitely presented monoids and semigroups respectively, which check if the defining presentation satisfies $C(4)$ and apply C4WordProblemReckless (4.3.3) if applicable. This is not installed by default since checking the $C(4)$ condition imposes a small overhead on equality tests in non-$C(4)$ semigroups (see 5.2).

The optional argument *verbose* is a boolean value indicating whether the equality tester should write to the standard output each time it performs an equality test, to tell the user which equality test routine is being used; this can be useful in interactive mode but is clearly undesirable when the equality tester is being employed in a program. The default behaviour is to produce no such output (equivalent to the argument false).

4.3.2 C4WordProblem

◊ C4WordProblem(*u*, *v*) (function)

C4WordProblem decides very efficiently whether the arguments *u* and *v* represent the same element of a finitely presented semigroup or monoid with defining presentation satisfying the small overlap condition $C(4)$. It returns true if they do represent the same element, false if they do not, and fail if *u* and *v* are not elements of the same finitely presented semigroup or monoid, or if the defining presentation does not satisfy $C(4)$.

4.3.3 C4WordProblemReckless

◊ `C4WordProblemReckless(`*S*`, `*u*`, `*v*`)` (function)

`C4WordProblemReckless` tests whether the arguments *u* and *v* are equal as elements of the $C(4)$ finitely presented monoid or semigroup *S*, returning `true` if they are and `false` if they are not. Unlike `C4WordProblem` (4.3.2), no checks are performed on the arguments and if they do not have the correct properties the behaviour is undefined - it may crash, not terminate, return the wrong answer, cause your computer to explode or occasionally even return the right answer. This function is available because it has slightly lower constant function call overhead that `C4WordProblem` (4.3.2), which may be significant when performing very large numbers of equality tests in the same semigroup or monoid. For all other purposes, and if in any doubt, use `C4WordProblem` (4.3.2) instead.

4.3.4 ComplementRelationWord

◊ `ComplementRelationWord(`*n*`)` (function)

`ComplementRelationWord` takes as its argument an integer *n*, and simply returns $n-1$ if *n* is even, and $n+1$ if *n* is odd. Hence, if *n* is an index into the list of relation words returned by `RelationWords` (4.1.4), then the valued returned is the index in the list corresponding to the complement relation word.

Chapter 5

Performance Issues

This chapter briefly discusses some performance issues with the use of the SmallOverlap package.

5.1 Time and Space Complexity

Most of the routines implemented require time and space which is quadratic in the presentation lengths, and linear in the element lengths. For a detailed complexity analysis of the algorithms see [Kam07].

5.2 Performance Penalty for non-C(4) Monoids and Semigroups.

We have already remarked that the small overlap equality test method is not automatically installed, since testing whether small overlap methods are applicable inevitably imposes a slight performance overhead when computing with monoids in which they turn out *not* to be applicable. More precisely, when the equality test method *is* installed:

- Testing the $C(4)$ condition is carried out once per presentation. This takes time quadratic (with very small coefficients) in the presentation length, so is only likely to be noticeable if working with very large presentations, or with extremely large numbers of presentations.
- Checking each element to see if the containing fp monoid or semigroup has the $C(4)$ property is carried out once per element. This should be done in constant time (assuming GAP is reasonably efficient in its internal implementation of function calls and attribute tests) but this may still be appreciable if you are carrying out vast numbers of equality tests in the same monoid, and that monoid does not satisfy $C(4)$.

In addition, one could almost certainly construct examples of $C(4)$ monoids where the standard GAP equality test algorithm (reduction to normal form using a confluent rewriting system created by Knuth-Bendix completion) runs even faster than the small overlap algorithm. In practice the latter is so fast that in such a case it is unlikely to matter much which algorithm is used but, once again, performing very large numbers of equality tests in such a monoid could in principle be slower with the small overlap routines.

For most purposes these penalties are negliglible, and since the vast majority of monoids *do* satisfy $C(4)$, likely to be vastly outweighed by the time saved by using small overlap methods where applicable. Hence, unless you are performing very large numbers of equality tests, and moreover have

definite reason to believe that a large proportion of these will be in monoids not satisfying $C(4)$, we strongly recommend that you call `PreferSmallOverlapEqualityTest`.

References

[Gro] GAP Group. The GAP 4 Reference Manual. Available online at http://www.gap-system.org/Doc/manuals.html. 8, 11, 12, 13

[Hig92] P. M. Higgins. *Techniques of semigroup theory*. Oxford Science Publications. The Clarendon Press Oxford University Press, New York, 1992. With a foreword by G. B. Preston. 6

[Kam07] M. Kambites. Small overlap monoids: the word problem. arXiv:0712.0250v1 [math.RA], 2007. 4, 6, 15

[LS77] R. C. Lyndon and P. E. Schupp. *Combinatorial Group Theory*. Springer-Verlag, 1977. 6

[Rem71] J. H. Remmers. *Some algorithmic problems for semigroups: a geometric approach*. PhD thesis, University of Michigan, 1971. 4, 6

[Rem80] J. H. Remmers. On the geometry of semigroup presentations. *Adv. in Math.*, 36(3):283–296, 1980. 6

Index