Journal of Symbolic Computation 45 (2010) 629–656

Contents lists available at ScienceDirect

Journal of Symbolic Computation

journal homepage: www.elsevier.com/locate/jsc

Solving the conjugacy problem in Garside groups by cyclic sliding

Volker Gebhardt [a,1], Juan González-Meneses [b]

[a] School of Computing and Mathematics, University of Western Sydney, Locked Bag 1797, Penrith South DC NSW 1797, Australia
[b] Dept. Álgebra, Facultad de Matemáticas, Universidad de Sevilla, Apdo. 1160, 41080 Sevilla, Spain

ARTICLE INFO

Article history:
Received 2 April 2009
Accepted 26 January 2010
Available online 2 February 2010

Keywords:
Garside groups
Conjugacy
Conjugacy problem
Cyclic sliding
Sliding circuits
Complexity analysis

ABSTRACT

We present a solution to the conjugacy decision problem and the conjugacy search problem in Garside groups, which is theoretically simpler than the usual one, with no loss of efficiency. This is done by replacing the well-known cycling and decycling operations by a new one, called cyclic sliding, which appears to be a more natural choice.

We give an analysis of the complexity of our algorithm in terms of fundamental operations with simple elements, so our analysis is valid for every Garside group.

This paper intends to be self-contained, not requiring any previous knowledge of prior algorithms, and includes all the details for the algorithm to be implemented on a computer.

1. Introduction

The **Conjugacy Decision Problem** (CDP) for a group G is the decision problem of determining, given any two elements $a, b \in G$, whether a and b are conjugate in G. The **Conjugacy Search Problem** (CSP), on the other hand, requires to compute for any two given conjugate elements $a, b \in G$ a conjugating element c such that $c^{-1}ac = b$. (We will also write $a^c = b$.)

In this paper we will describe a new algorithm to solve both problems in Garside groups (of finite type). The simplicity of the algorithm will allow us to describe it completely in this introduction in a ready-to-implement manner. The main difference from established algorithms is the use of an operation called *cyclic sliding*, which is a special kind of conjugation introduced in Gebhardt and

E-mail addresses: v.gebhardt@uws.edu.au (V. Gebhardt), meneses@us.es (J. González-Meneses).
URL: http://www.personal.us.es/meneses (J. González-Meneses).
[1] Tel.: +61 2 4736 0688; fax: +61 2 4736 0867.

0747-7171/$ – see front matter © 2010 Elsevier Ltd. All rights reserved.
doi:10.1016/j.jsc.2010.01.013

González-Meneses (in press). Cyclic sliding assumes the role played by *cycling* and *decycling* in previous algorithms.

Cyclic sliding will be motivated and explained in Section 1.2, but it can be defined right now. One just needs to recall the following notions in a Garside group G, which are well known to specialists. Firstly, G admits a partial order $\preccurlyeq$, and there is a special element Δ, called Garside element. Given $x \in G$, $\inf(x)$ and $\sup(x)$ are the maximal and minimal integers, respectively, satisfying $\Delta^{\inf(x)} \preccurlyeq x \preccurlyeq \Delta^{\sup(x)}$. Secondly, given $a, b \in G$, there is a unique greatest common divisor $a \wedge b$ with respect to $\preccurlyeq$. Finally, the elements in the set $[1, \Delta] = \{s \in G \mid 1 \preccurlyeq s \preccurlyeq \Delta\}$, called *simple elements*, generate G. We assume this set to be finite (that is, G is of finite type). It is well known how to compute all the above data in a Garside group G of finite type, as we shall see.

Using the above well-known notions, we can define the following:

Definition 1.1 (*Gebhardt and González-Meneses, in press*). Given $x \in G$, we define the **preferred prefix** $\mathfrak{p}(x)$ of x as the simple element

$$\mathfrak{p}(x) = \left(x\Delta^{-\inf(x)}\right) \wedge \left(x^{-1}\Delta^{\sup(x)}\right) \wedge \Delta,$$

and we define the **cyclic sliding** $\mathfrak{s}(x)$ of x as the conjugate of x by its preferred prefix, that is,

$$\mathfrak{s}(x) = x^{\mathfrak{p}(x)} = \mathfrak{p}(x)^{-1} x\, \mathfrak{p}(x).$$

This is enough to describe a simple algorithm to solve the conjugacy decision problem and the conjugacy search problem in a Garside group of finite type. The algorithm we present now, however, is by far not the best possible one. In Section 1.3 we will give a much better algorithm, which requires some other notions besides the preferred prefix and the cyclic sliding. Nevertheless, the simple version given here for illustration can be useful for theoretical purposes or for applying it to small examples.

ALGORITHM 0:
Solving the conjugacy problem in a Garside group G of finite type

Input: $x, y \in G$.
Output: - Whether x and y are conjugate.
- If x and y are conjugate, an element c such that $x^c = y$.

(1) Set $\widetilde{x} = x$, $c_1 = 1$ and $\mathcal{T} = \emptyset$.
(2) While $\widetilde{x} \notin \mathcal{T}$, set $\mathcal{T} = \mathcal{T} \cup \{\widetilde{x}\}$, $c_1 = c_1 \cdot \mathfrak{p}(\widetilde{x})$ and $\widetilde{x} = \mathfrak{s}(\widetilde{x})$.
(3) Set $\widetilde{y} = y$, $c_2 = 1$ and $\mathcal{T} = \emptyset$.
(4) While $\widetilde{y} \notin \mathcal{T}$, set $\mathcal{T} = \mathcal{T} \cup \{\widetilde{y}\}$, $c_2 = c_2 \cdot \mathfrak{p}(\widetilde{y})$ and $\widetilde{y} = \mathfrak{s}(\widetilde{y})$.
(5) Set $\mathcal{V} = \{\widetilde{x}\}$, $\mathcal{V}' = \{\widetilde{x}\}$ and $c_{\widetilde{x}} = 1$.
(6) While $\mathcal{V}' \neq \emptyset$, do:
  (a) Take $v \in \mathcal{V}'$.
  (b) For every simple element s, do:
    (i) If $v^s = \widetilde{y}$, then:
      (A) Set $c_{\widetilde{y}} = c_v \cdot s$.
      (B) Return 'x and y are conjugate by $c_1 \cdot c_{\widetilde{y}} \cdot c_2^{-1}$'.
    (ii) If $v^s \notin \mathcal{V}$, then:
      (A) Apply iterated cyclic sliding to v^s until the first repetition is encountered, say w.
      (B) If $w = v^s$, then set $c_{v^s} = c_v \cdot s$, $\mathcal{V} = \mathcal{V} \cup \{v^s\}$, and $\mathcal{V}' = \mathcal{V}' \cup \{v^s\}$.
  (c) Remove v from $\mathcal{V}'$.
(7) Return 'x and y are not conjugate'.

The set $\mathcal{V}$ computed by the above algorithm, called the *set of sliding circuits of x* and denoted $\mathrm{SC}(x)$, was introduced in Gebhardt and González-Meneses (in press). It is a finite invariant of the conjugacy class x^G of x, that is, it is a finite subset of x^G and only depends on x^G, not on x itself. This set $\mathrm{SC}(x)$ consists of those conjugates of x which are stabilised by $\mathfrak{s}^k$ for some positive integer k and it is analogous to the ultra summit set $\mathrm{USS}(x)$ introduced by Gebhardt (2005). One has $\mathrm{SC}(x) \subseteq \mathrm{USS}(x)$, and in general $\mathrm{SC}(x)$ is a proper subset of $\mathrm{USS}(x)$.

The first two lines of the algorithm compute an element $\widetilde{x} \in \mathrm{SC}(x)$, by applying iterated cyclic sliding until the first repetition is reached (which is $\widetilde{x}$). A conjugating element c_1 from x to $\widetilde{x}$ is also computed. The following two lines compute $\widetilde{y} \in \mathrm{SC}(y)$ and a conjugating element c_2 from y to $\widetilde{y}$ in the same way. Then, the algorithm starts to compute the whole set $\mathrm{SC}(x)$. If during the computation it finds $\widetilde{y}$ as an element of $\mathrm{SC}(x)$, the algorithm stops and returns a conjugating element from x to y. If this does not occur, that is, if the algorithm computes the whole set $\mathrm{SC}(x)$ without finding $\widetilde{y}$ in it, then it returns the message 'x and y are not conjugate'.

The use of cyclic sliding not only allows to develop a simpler algorithmic solution to the CDP/CSP, but also is of theoretical interest; we refer to Gebhardt and González-Meneses (in press) for details. It is shown there that the set of sliding circuits has all the good properties of the ultra summit set, but is the more natural invariant in many ways. In particular, the properties of the set of sliding circuits fully extend to the case of elements of *summit canonical length* 1, which is not the case for ultra summit sets. Another indication of the naturalness of the cyclic sliding operation is the fact that for *super summit elements* which have a *rigid* conjugate, the (unique) minimal positive element yielding a rigid conjugate is precisely the conjugating element obtained by iterated cyclic sliding.

The structure of this paper is as follows. In the introduction, we present our algorithm solving the conjugacy problems in Garside groups in a ready-to-implement form. This presentation is kept as concise as possible; explanations, motivations and the proof of correctness are postponed to later sections. More precisely, in Section 1.1, we give a basic introduction to the theory of Garside groups; specialists may skip this part. In Section 1.2 we briefly explain the new concepts from Gebhardt and González-Meneses (in press) which are subsequently used for the detailed description of the algorithm in Section 1.3.

The rest of the paper is devoted to the explanation and analysis of the algorithm. Section 2 contains a summary of results from Gebhardt and González-Meneses (in press) which are required in our discussion. In Section 3 the algorithm is explained and shown to be correct. Finally, the complexity of the new algorithm is analysed in Section 4, where Section 4.1 discusses how the operations required for our algorithm can be realised, only assuming knowledge of the lattice of simple elements.

1.1. Basic facts about Garside groups

Garside groups were defined by Dehornoy and Paris (1999). For a detailed introduction to these groups, see (Dehornoy, 2002); a shorter introduction, containing all the details needed for this paper can be found in Birman et al. (2007a) (Section 1.1 and the beginning of Section 1.2).

One of the possible definitions of a Garside group is the following. A group G is said to be a **Garside group** with **Garside structure** (G, P, Δ) if it admits a submonoid P satisfying $P \cap P^{-1} = \{1\}$, called the monoid of **positive elements**, and a special element $\Delta \in P$ called the **Garside element**, such that the following properties hold:

(G1) The partial order $\preccurlyeq$ defined on G by $a \preccurlyeq b \Leftrightarrow a^{-1}b \in P$ (which is invariant under left multiplication by definition) is a lattice order. That is, for every $a, b \in G$ there are a unique least common multiple $a \vee b$ and a unique greatest common divisor $a \wedge b$ with respect to $\preccurlyeq$. (In other words, there exists a unique element $a \vee b$ such that $a \preccurlyeq a \vee b$ and $b \preccurlyeq a \vee b$, and for any $c \in G$ the conditions $a \preccurlyeq c$ and $b \preccurlyeq c$ together imply $a \vee b \preccurlyeq c$. Similarly, there exists a unique element $a \wedge b$, such that $a \wedge b \preccurlyeq a$ and $a \wedge b \preccurlyeq b$, and for any $c \in G$ the conditions $c \preccurlyeq a$ and $c \preccurlyeq b$ together imply $c \preccurlyeq a \wedge b$.)
(G2) The set $[1, \Delta] = \{a \in G \mid 1 \preccurlyeq a \preccurlyeq \Delta\}$, called the set of **simple elements**, generates G.
(G3) Conjugation by Δ preserves P (so it preserves the lattice order $\preccurlyeq$). That is, $\Delta^{-1}P\Delta = P$.
(G4) For all $x \in P\backslash\{1\}$, one has:

$$\|x\| = \sup\{k \mid \exists\, a_1, \ldots, a_k \in P\backslash\{1\} \text{ such that } x = a_1 \cdots a_k\} < \infty.$$

Definition 1.2. A Garside structure (G, P, Δ) is said to be **of finite type** if the set of simple elements $[1, \Delta]$ is finite. A group G is said to be a **Garside group of finite type** if it admits a Garside structure of finite type.

Throughout this paper, let G be a Garside group of finite type with a fixed Garside structure (G, P, Δ) of finite type. We denote by τ the inner automorphism of G corresponding to conjugation by Δ.

Remarks.

(1) By definition, $p \in P \Leftrightarrow 1 \preccurlyeq p$. Given two positive elements $a \preccurlyeq b$, one usually says that a is a **prefix** of b. Hence the simple elements are the positive prefixes of Δ.
(2) The number $\|x\|$ defined above for each $x \in P\backslash\{1\}$, defines a norm in P (setting $\|1\| = 0$). Note that the existence of this norm implies that every element in $P\backslash\{1\}$ can be written as a product of **atoms**, where an atom is an element $a \in P$ that cannot be decomposed in P, that is, $a = bc$ with $b, c \in P$ implies that either $b = 1$ or $c = 1$. In any decomposition of x as a product of $\|x\|$ factors in $P\backslash\{1\}$, all of them are atoms. Notice that the set of atoms generates G and is finite.
(3) It is easy to see that τ induces a permutation of the atoms of G. As this permutation is of finite order and the set of atoms generates G, some power Δ^e of Δ lies in the centre of G.

The main examples of Garside groups of finite type are Artin–Tits groups of spherical type. In particular, braid groups are Garside groups. In the braid group B_n on n strands with the usual Garside structure that we call the **Artin Garside structure** of B_n, one has the following:

- The atoms are the standard generators $\sigma_1, \ldots, \sigma_{n-1}$.
- The positive elements are the braids that can be written as a word which only contains positive powers of the atoms.
- The simple elements are the positive braids in which any two strands cross at most once. One has $|[1, \Delta]| = n!$, so this is a finite type Garside structure.
- The Garside element $\Delta = \sigma_1(\sigma_2\sigma_1)(\sigma_3\sigma_2\sigma_1)\cdots(\sigma_{n-1}\cdots\sigma_1)$ is the positive braid in which any two strands cross exactly once (also called the *half twist*). Δ^2 is central in B_n.

Note that the monoid P induces not only a partial order $\preccurlyeq$ which is invariant under left multiplication, but also a partial order $\succcurlyeq$ which is invariant under right multiplication. The latter is defined by $a \succcurlyeq b \Leftrightarrow ab^{-1} \in P$. It follows from the properties of G that $\succcurlyeq$ is also a lattice order, that P is the set of elements a such that $a \succcurlyeq 1$, and that the simple elements are the positive suffixes of Δ (where we say that a positive element b is a suffix of a if $a \succcurlyeq b$). We will denote by $x \wedge^{\uparrow} y$ (resp. $x \vee^{\uparrow} y$) the greatest common divisor (resp. least common multiple) of $x, y \in G$ with respect to $\succcurlyeq$.

Directly from the definitions, we have the following Lemma.

Lemma 1.3. *For any $a, b \in G$ the following hold:*

(1) $a \preccurlyeq b$ *if and only if* $a^{-1} \succcurlyeq b^{-1}$.
(2) $a \preccurlyeq b$ *if and only if* $\tau(a) \preccurlyeq \tau(b)$.
(3) $(a \wedge b)^{-1} = a^{-1} \vee^{\uparrow} b^{-1}$.
(4) $\tau(a \wedge b) = \tau(a) \wedge \tau(b)$.
(5) $(a \vee b)^{-1} = a^{-1} \wedge^{\uparrow} b^{-1}$.
(6) $\tau(a \vee b) = \tau(a) \vee \tau(b)$.

Proof. For Claim 1 note that $a \preccurlyeq b$ if and only if there exists $c \in P$ such that $ac = b$, that is, $a^{-1} = cb^{-1}$, which is in turn equivalent to $a^{-1} \succcurlyeq b^{-1}$. Claims 3 and 5 then follow from the definitions of the greatest common divisor respectively least common multiple.

Claim 2 holds since $c \in P$ is equivalent to $\tau(c) \in P$ by axiom (G3). Claims 4 and 6 then follow from the definitions of the greatest common divisor respectively least common multiple. □

The following notions are well known to specialists in Garside groups:

Definition 1.4. Given a simple element s, the **right complement** of s is defined by $\partial(s) = s^{-1}\Delta$, and the **left complement** of s is $\partial^{-1}(s) = \Delta\, s^{-1}$.

Notice that the map $\partial : [1, \Delta] \to [1, \Delta]$ is a bijection of the (finite) set $[1, \Delta]$. Notice also that $\partial^2(s) = \Delta^{-1}s\Delta = \tau(s)$.

Definition 1.5. Given two simple elements a and b, we say that the decomposition $a \cdot b$ is **left weighted** if $\partial(a) \wedge b = 1$ or, equivalently, if $ab \wedge \Delta = a$. We say that the decomposition $a \cdot b$ is **right weighted** if $a \wedge^{\uparrow} \partial^{-1}(b) = 1$ or, equivalently, if $ab \wedge^{\uparrow} \Delta = b$.

The process of bringing a product $a \cdot b$ of two simple elements a and b into left weighted form by replacing it with the product $(as) \cdot (s^{-1}b)$, where $s = \partial(a) \wedge b$, is called a **local left sliding** or simply a **local sliding** (Gebhardt and González-Meneses, in press). **Local right sliding** is defined analogously.

Definition 1.6. Given $x \in G$, we say that a decomposition $x = \Delta^p x_1 \cdots x_r$, where $p \in \mathbb{Z}$ and $r \geq 0$, is the **left normal form** of x if $x_i \in [1, \Delta]\backslash\{1, \Delta\}$ for $i = 1, \ldots, r$ and $x_i x_{i+1}$ is a left weighted decomposition for $i = 1, \ldots, r-1$. We say that a decomposition $x = y_r \cdots y_1 \Delta^p$ is the **right normal form** of x if $y_i \in [1, \Delta]\backslash\{1, \Delta\}$ for $i = 1, \ldots, r$ and $y_{i+1} y_i$ is a right weighted decomposition for $i = 1, \ldots, r-1$.

It is well known that left and right normal forms of elements in G exist and are unique. (Proposition 4.3 recalls how to compute them based on local slidings.) Moreover, the numbers p and r do not depend on the normal form (left or right) that we are considering.

Definition 1.7. Given $x \in G$, whose left normal form is $\Delta^p x_1 \cdots x_r$ and whose right normal form is $y_1 \cdots y_r \Delta^p$, we define the **infimum**, **canonical length** and **supremum** of x, respectively, by $\inf(x) = p$, $\ell(x) = r$ and $\sup(x) = p + r$.

It is shown in ElRifai and Morton (1994) that $\inf(x)$ and $\sup(x)$ are precisely the maximal and minimal integers, respectively, such that $\Delta^{\inf(x)} \preccurlyeq x \preccurlyeq \Delta^{\sup(x)}$ (or, equivalently, $\Delta^{\sup(x)} \succcurlyeq x \succcurlyeq \Delta^{\inf(x)}$). Moreover, if $x = \Delta^p x_1 \cdots x_r$ is in left normal form as written, then $x^{-1} = \Delta^{-(p+r)}\ \partial^{-2(p+r)+1}(x_r)\ \partial^{-2(p+r-1)+1}(x_{r-1}) \cdots \partial^{-2(p+1)+1}(x_1)$ is in left normal form as written. An analogous relation holds for the right normal forms of x and x^{-1}. This implies in particular that $\inf(x^{-1}) = -\sup(x)$, $\sup(x^{-1}) = -\inf(x)$ and $\ell(x^{-1}) = \ell(x)$. From Lemma 1.3 it is obvious that $\inf(\tau(x)) = \inf(x)$, $\sup(\tau(x)) = \sup(x)$ and $\ell(\tau(x)) = \ell(x)$. Moreover, the factors in the left (resp. right) normal form of $\tau(x)$ are precisely the images under τ of the factors in the left (resp. right) normal form of x.

The first factor and the last factor in the left normal form respectively the right normal form are of special importance.

Definition 1.8. Given $x \in G$, the **(left) initial factor** $\iota(x)$ of x is defined as $\iota(x) = x\Delta^{-\inf(x)} \wedge \Delta$ and the **(left) final factor** of x is $\varphi(x) = (\Delta^{\sup(x)-1} \wedge x)^{-1} x$. Similarly, the **right initial factor** of x is $\iota^{\urcorner}(x) = \Delta^{-\inf(x)} x \wedge^{\urcorner} \Delta$ and the **right final factor** of x is $\varphi^{\urcorner}(x) = x\,(\Delta^{\sup(x)-1} \wedge^{\urcorner} x)^{-1}$.

We remark that if $\ell(x) = r > 0$, and $\Delta^p x_1 \cdots x_r$ is the left normal form of x, then $\iota(x) = \tau^{-p}(x_1)$ and $\varphi(x) = x_r$. This explains the names given to these simple elements. Notice also that if $r = 0$, that is, if $x = \Delta^p$, then $\iota(x) = 1$ and $\varphi(x) = \Delta$. From the relation between the normal forms of x and x^{-1}, we see that $\iota(x^{-1}) = \partial(\varphi(x))$. Similarly, $\iota^{\urcorner}(x^{-1}) = \partial^{-1}(\varphi^{\urcorner}(x))$.

Definition 1.9. Let x^G denote the conjugacy class of x in G and define the **summit infimum** $\inf_s(x) = \max\{\inf(y) \mid y \in x^G\}$ and the **summit supremum** $\sup_s(x) = \min\{\sup(y) \mid y \in x^G\}$. The set $\mathrm{SSS}(x) = \{y \in x^G \mid \inf(y) = \inf_s(x),\ \sup(y) = \sup_s(x)\}$ is called the **super summit set** of x; the elements of $\mathrm{SSS}(x)$ are called **super summit elements**. The canonical length of super summit elements is called the **summit canonical length** $\ell_s(x)$. One obviously has $\ell_s(x) = \sup_s(x) - \inf_s(x)$.

It is well known that $\mathrm{SSS}(x) \subset x^G$ is non-empty and finite (ElRifai and Morton, 1994) and it is clear from the definition that $\mathrm{SSS}(x)$ only depends on the conjugacy class of x. By the above remark, $\inf(y^{-1}) = -\sup(y)$ and $\sup(y^{-1}) = -\inf(y)$ for all $y \in G$, and thus $y \in \mathrm{SSS}(x)$ if and only if $y^{-1} \in \mathrm{SSS}(x^{-1})$. Similarly, $\inf(\tau(x)) = \inf(x)$, $\sup(\tau(x)) = \sup(x)$ and $\ell(\tau(x)) = \ell(x)$, whence $y \in \mathrm{SSS}(x)$ if and only if $\tau(y) \in \mathrm{SSS}(\tau(x)) = \mathrm{SSS}(x)$.

We summarise the discussion in this section in the following Lemma.

Lemma 1.10 (*ElRifai and Morton, 1994*). *For any $x \in G$ one has the following:*

(1) $\inf(x) = \max\{i \in \mathbb{Z} \mid \Delta^i \preccurlyeq x\} = \max\{i \in \mathbb{Z} \mid x \succcurlyeq \Delta^i\}$.
(2) $\sup(x) = \min\{i \in \mathbb{Z} \mid x \preccurlyeq \Delta^i\} = \min\{i \in \mathbb{Z} \mid \Delta^i \succcurlyeq x\}$.
(3) *If $x = \Delta^p x_1 \cdots x_r$ is in left normal form, then the left normal form of x^{-1} is*

$$x^{-1} = \Delta^{-(p+r)}\ \partial^{-2(p+r)+1}(x_r)\ \partial^{-2(p+r-1)+1}(x_{r-1}) \cdots \partial^{-2(p+1)+1}(x_1).$$

If $x = x_r \cdots x_1 \Delta^p$ is in right normal form, then the right normal form of x^{-1} is

$$x^{-1} = \partial^{2(p+1)-1}(x_1)\ \partial^{2(p+2)-1}(x_2) \cdots \partial^{2(p+r)-1}(x_r)\ \Delta^{-(p+r)}.$$

(4) $\inf(x^{-1}) = -\sup(x)$, $\sup(x^{-1}) = -\inf(x)$ *and* $\ell(x^{-1}) = \ell(x)$.
(5) $\iota(x^{-1}) = \partial(\varphi(x))$ *and* $\iota^{\gamma}(x^{-1}) = \partial^{-1}(\varphi^{\gamma}(x))$.
(6) $\iota(\tau(x)) = \tau(\iota(x))$, $\varphi(\tau(x)) = \tau(\varphi(x))$, $\iota^{\gamma}(\tau(x)) = \tau(\iota^{\gamma}(x))$, $\varphi^{\gamma}(\tau(x)) = \tau(\varphi^{\gamma}(x))$.
(7) $\inf_s(x^{-1}) = -\sup_s(x)$, $\sup_s(x^{-1}) = -\inf_s(x)$ *and* $\ell_s(x^{-1}) = \ell_s(x)$.
(8) $\mathrm{SSS}(x^{-1}) = \{y^{-1} \mid y \in \mathrm{SSS}(x)\}$.
(9) $y \in \mathrm{SSS}(x)$ *if and only if* $\tau(y) \in \mathrm{SSS}(x)$.

1.2. Cyclic sliding

Before explaining our algorithm, we need to describe the underlying operation called *cyclic sliding* introduced in Gebhardt and González-Meneses (in press). The use of cyclic sliding (instead of the well-known *cycling* and *decycling* operations) is what distinguishes the new algorithm from previously known ones. The cyclic sliding operation will be motivated and explained in more detail in the following section. Here we just give the technical definitions, so that they can be used in the algorithm. Recall that G is a Garside group of finite type with a fixed finite type Garside structure (G, P, Δ).

Definition 1.11. Given $x \in G$, the **preferred prefix** $\mathfrak{p}(x)$ of x is the simple element

$$\mathfrak{p}(x) = \left(x\Delta^{-\inf(x)}\right) \wedge \left(x^{-1}\Delta^{\sup(x)}\right) \wedge \Delta = \iota(x) \wedge \iota(x^{-1}) = \iota(x) \wedge \partial(\varphi(x)),$$

and the **preferred suffix** $\mathfrak{p}^{\gamma}(x)$ of x is the simple element

$$\mathfrak{p}^{\gamma}(x) = \left(\Delta^{-\inf(x)}x\right) \wedge^{\gamma} \left(\Delta^{\sup(x)}x^{-1}\right) \wedge^{\gamma} \Delta = \iota^{\gamma}(x) \wedge^{\gamma} \iota^{\gamma}(x^{-1}) = \iota^{\gamma}(x) \wedge^{\gamma} \partial^{-1}(\varphi^{\gamma}(x)).$$

Definition 1.12. Given $x \in G$, the **cyclic left sliding** $\mathfrak{s}(x)$ of x is the conjugate of x by its preferred prefix, that is,

$$\mathfrak{s}(x) = x^{\mathfrak{p}(x)} = \mathfrak{p}(x)^{-1}x\,\mathfrak{p}(x),$$

and the **cyclic right sliding** $\mathfrak{s}^{\gamma}(x)$ of x is the conjugate of x by the inverse of its preferred suffix:

$$\mathfrak{s}^{\gamma}(x) = x^{\mathfrak{p}^{\gamma}(x)^{-1}} = \mathfrak{p}^{\gamma}(x)\; x\; \mathfrak{p}^{\gamma}(x)^{-1}.$$

If there is no possible confusion, we will call $\mathfrak{s}(x)$ the **cyclic sliding**, or just the **sliding** of x.

It will be convenient to display conjugations in a graph-theoretical style. In this way, we shall write $u \xrightarrow{s} v$ if $u^s = v$ for some $u, s, v \in G$. Hence we have:

$$x \xrightarrow{\mathfrak{p}(x)} \mathfrak{s}(x) \qquad \text{and} \qquad x \xleftarrow{\mathfrak{p}^{\gamma}(x)} \mathfrak{s}^{\gamma}(x).$$

Elements for which the preferred prefix (or the preferred suffix) is trivial behave particularly nicely in may ways.

Definition 1.13. An element $x \in G$ is called **left rigid** or just **rigid** if $\mathfrak{p}(x) = 1$. Similarly, x is called **right rigid** if $\mathfrak{p}^{\gamma}(x) = 1$.

In Birman et al. (2007a), the concept of rigidity was introduced and some of the properties of rigid elements were analysed. It is obvious from the definition that left (respectively right) rigid elements are fixed points for left (respectively right) cyclic sliding. The converse clearly is not true.

The main idea of our algorithm is the following: Iterated application of cyclic sliding sends any element $x \in G$ to a finite subset of its conjugacy class x^G. This subset only depends on x^G and is, in general, small. Hence, it can be used to solve the CDP and the CSP efficiently. This set is defined as follows:

Definition 1.14. We say that $y \in G$ belongs to a **sliding circuit** if $\mathfrak{s}^m(y) = y$ for some $m \geq 1$. Given $x \in G$, we define the **set of sliding circuits of** x, denoted by $\mathrm{SC}(x)$, as the set of all conjugates of x which belong to a sliding circuit.

Lemma 1.15. *The maps* τ *and* $\mathfrak{s}$ *commute. In particular, one has* $y \in \mathrm{SC}(x)$ *if and only if* $y^{\Delta^k} = \tau^k(y) \in \mathrm{SC}(x)$ *for all* $k \in \mathbb{Z}$.

Proof. By Lemmas 1.3 and 1.10, one has $\mathfrak{p}(\tau(y)) = \tau(\mathfrak{p}(y))$, which yields the claim. □

Our algorithm will compute not only the set $\mathrm{SC}(x)$, but also conjugating elements connecting the elements of $\mathrm{SC}(x)$. Basically, it constructs a connected directed graph, whose vertices correspond to the elements of $\mathrm{SC}(x)$ and whose arrows correspond to conjugating elements sending one given element in $\mathrm{SC}(x)$ to another.

Definition 1.16. Given $x \in G$, the **sliding circuits graph** $\mathrm{SCG}(x)$ of x is the directed graph whose set of vertices is $\mathrm{SC}(x)$ and whose arrows correspond to conjugating elements as follows: There is an arrow which starts at $u \in \mathrm{SC}(x)$, ends at $v \in \mathrm{SC}(x)$ and is labelled by $s \in P \setminus \{1\}$ if and only if:

(1) $u^s = v$.
(2) s is an **indecomposable conjugator**, that is, $s \neq 1$ and there is no element t, such that $1 \prec t \prec s$ and $u^t \in \mathrm{SC}(x)$.

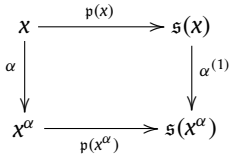
We remark that the label of each arrow is a simple element (see Corollary 2.11 or Corollary 3.3).

Finally, we need to define two operations that will be applied to the conjugating elements. They are analogous to the ones defined in Gebhardt (2005), and we use the same names.

Definition 1.17. Given $x, \alpha \in G$, we define the **transport** of α at x under cyclic sliding as

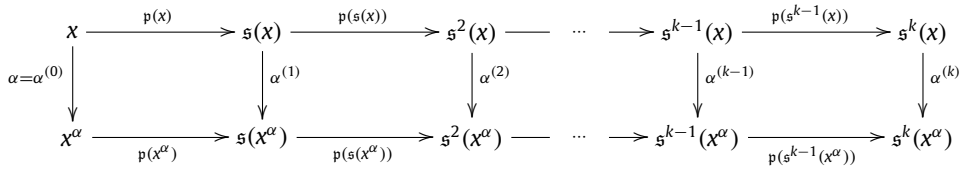
$$\alpha^{(1)} = \mathfrak{p}(x)^{-1}\, \alpha\, \mathfrak{p}(x^\alpha).$$

That is, $\alpha^{(1)}$ is the conjugating element that makes the following diagram commutative, in the sense that the conjugating element along any closed path is trivial:



Note that the horizontal rows in this diagram correspond to applications of cyclic sliding.

For an integer $i > 1$ we define recursively $\alpha^{(i)} = (\alpha^{(i-1)})^{(1)}$. Note that $(\alpha^{(i-1)})^{(1)}$ indicates the transport of $\alpha^{(i-1)}$ at $\mathfrak{s}^{i-1}(x)$. We also define $\alpha^{(0)} = \alpha$.

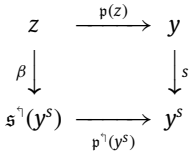


The above operation is a way to transport a conjugating element along a sliding path. However, occasionally we will need to go backwards, in some sense, although the obtained element will not necessarily be a pre-image under transport. In Section 3.3 we will define the *pullback* $s_{(1)}$ of a positive element s at an element $y = \mathfrak{s}(z) \in \mathrm{SC}(x)$ via the properties of its transport at z and define recursively $s_{(i)} = (s_{(i-1)})_{(1)}$ for any integer $i > 1$ and $s_{(0)} = s$ (Definition 3.14). The details are somewhat technical and require some prior work, so we postpone them at this stage. At the moment, we just need to know how to compute pullbacks in a certain special case; this is the content of the following proposition which will be shown in Section 3.3:

Proposition 3.19. *Let $x \in G$, $z \in \mathrm{SC}(x)$, $y = \mathfrak{s}(z)$ and let $s \in G$ be positive such that y^s is super summit. Then the pullback of s at y, as given in Definition 3.14, is*

$$s_{(1)} = \left(\mathfrak{p}(z)\, s\, \mathfrak{p}^{\gamma}(y^s)^{-1}\right) \vee 1\,.$$

Hence, $s_{(1)} = \beta \vee 1$, where $\beta \in G$ is the element that makes the following diagram commutative, in the sense that the conjugating element along any closed path is trivial:



1.3. The algorithm

In this subsection we will describe in detail our algorithm to solve the CDP and the CSP in a Garside group G. The only requirement needed to implement it, which we assume to be fulfilled for the given Garside group G, is to know the structure of the lattices of simple elements, with respect to both $\preccurlyeq$ and $\succcurlyeq$. More precisely, one should know the Garside element Δ and have:

(1) A list containing the atoms, $\mathcal{A} = \{a_1, \ldots, a_\lambda\}$.
(2) A function that, given $a \in \mathcal{A}$ and $s \in [1, \Delta]$, determines whether $a \preccurlyeq s$ and, in that case, computes the simple element $a^{-1}s$.
(3) A function that, given $a \in \mathcal{A}$ and $s \in [1, \Delta]$, determines whether $s \succcurlyeq a$ and, in that case, computes the simple element $s\,a^{-1}$.

In Section 4.1 we will see how, provided the above requirements are fulfilled, one can compute right and left complements, gcds and lcms, normal forms, preferred prefixes and suffixes, cyclic slidings, transports and pullbacks.

The whole algorithm is divided into three parts, called Algorithms 1, 2 and 3. Algorithm 1 computes one element $\widetilde{x}$ in the set $\mathrm{SC}(x)$, starting from an arbitrary element $x \in G$. The algorithm also computes a conjugating element from x to $\widetilde{x}$. Algorithm 2 computes the arrows in the graph $\mathrm{SCG}(x)$ which start at a given vertex; this is necessary for computing the entire set $\mathrm{SC}(x)$. Moreover, knowing all arrows of the graph will allow us to compute a conjugating element for every pair of elements in $\mathrm{SC}(x)$. Finally, Algorithm 3 solves the CDP and the CSP in G using Algorithms 1 and 2.

We remark that Algorithm 1 is a refinement of the algorithm in ElRifai and Morton (1994) to compute an element in the so-called super summit set of x. Here we replace two kinds of conjugation, called *cycling* and *decycling*, by a single kind of conjugation: cyclic sliding. This is one of the reasons that make our algorithm simpler. Algorithm 2 is a modification of the analogous one given in Gebhardt (2005), applied to cyclic sliding instead of cycling. Algorithm 3 is not new, since it is implicitly or explicitly described in ElRifai and Morton (1994), Franco and González-Meneses (2003) and Gebhardt (2005) in the context of other invariant subsets of the conjugacy class, namely super summit sets, super summit sets with minimal simple elements, respectively ultra summit sets. The set $\mathrm{SC}(x)$ is a subset of all of these sets (Gebhardt and González-Meneses, in press).

We recommend that the reader not try to understand the algorithms at a first reading. They will be clarified in the following sections, where each particular step of the algorithms will be explained in a more humane way. See Section 4.2 for remarks concerning efficient implementation of the algorithms.

ALGORITHM 1:
Computing one element in SC(x)

Input: $x \in G$.
Output: $\widetilde{x} \in \mathrm{SC}(x)$ and $c \in G$ such that $x^c = \widetilde{x}$.

(1) Set $\widetilde{x} = x$, $c = 1$ and $\mathcal{T} = \emptyset$.
(2) While $\widetilde{x} \notin \mathcal{T}$, set $\mathcal{T} = \mathcal{T} \cup \{\widetilde{x}\}$, $c = c \cdot \mathfrak{p}(\widetilde{x})$ and $\widetilde{x} = \mathfrak{s}(\widetilde{x})$.
(3) Set $y = \mathfrak{s}(\widetilde{x})$ and $d = \mathfrak{p}(\widetilde{x})$.
(4) While $y \neq \widetilde{x}$, set $d = d \cdot \mathfrak{p}(y)$ and $y = \mathfrak{s}(y)$.
(5) Return $\widetilde{x}$ and $c = c\,d^{-1}$.

ALGORITHM 2:
Computing the arrows in SCG(x) starting at a given vertex

Input: $v \in \mathrm{SC}(x)$.
Output: The set $\mathcal{A}_v$ of arrows in the graph $\mathrm{SCG}(x)$ starting at v.

(1) Compute the minimal integer $N > 0$ such that $\mathfrak{s}^N(v) = v$.
(2) List the atoms of G, say $a_1, \ldots, a_\lambda$. Set $\mathcal{A}_v = \emptyset$ and $Atoms = \emptyset$.
(3) For $t = 1, \ldots, \lambda$ do:
  (a) Set $s = a_t$.
  (b) While $\ell(v^s) > \ell(v)$, set $s = s \cdot \left(1 \vee (v^s)^{-1}\Delta^{\inf(v)} \vee v^s\Delta^{-\sup(v)}\right)$.
  (c) If $a_t \preccurlyeq \mathfrak{p}(v)$, then compute the iterated N-pullbacks $s, s_{(N)}, s_{(2N)}, \ldots$ until the first repetition is encountered, say $s_{(rN)}$, and set $s = s_{(rN)}$.
  (d) Compute the iterated N-transports $s, s^{(N)}, s^{(2N)}, \ldots$ until the first repetition is encountered, say $s^{(jN)}$. Let $i < j$ be such that $s^{(iN)} = s^{(jN)}$.
  (e) If $a_t \preccurlyeq s^{(mN)}$ for some m with $i \leq m < j$, then do:
    (i) If $a_k \not\preccurlyeq s^{(mN)}$ for all $k = 1, \ldots, \lambda$ such that either $a_k \in Atoms$ or $k > t$, then set
    $$\mathcal{A}_v = \mathcal{A}_v \cup \{s^{(mN)}\} \quad and \quad Atoms = Atoms \cup \{a_t\}.$$
(4) Return $\mathcal{A}_v$.

ALGORITHM 3:
Solving the conjugacy problems in G

Input: $x, y \in G$.
Output: - Whether x and y are conjugate.
- If x and y are conjugate, an element c such that $x^c = y$.

(1) Use Algorithm 1 to compute $\widetilde{x} \in \mathrm{SC}(x)$ and $\widetilde{y} \in \mathrm{SC}(y)$, together with conjugating elements c_1 and c_2 such that $x^{c_1} = \widetilde{x}$ and $y^{c_2} = \widetilde{y}$.
(2) Set $\mathcal{V} = \{\widetilde{x}\}$, $\mathcal{V}' = \{\widetilde{x}\}$ and $c_{\widetilde{x}} = 1$.
(3) While $\mathcal{V}' \neq \emptyset$, do:
  (a) Take $v \in \mathcal{V}'$.
  (b) Use Algorithm 2 to compute $\mathcal{A}_v$.
  (c) For every $s \in \mathcal{A}_v$, do:
    (i) If $v^s = \widetilde{y}$, then set $c_{\widetilde{y}} = c_v \cdot s$. Return '$x$ and y are conjugate by $c_1 \cdot c_{\widetilde{y}} \cdot c_2^{-1}$'.
    (ii) If $v^s \notin \mathcal{V}$, then set $c_{v^s} = c_v \cdot s$, $\mathcal{V} = \mathcal{V} \cup \{v^s\}$, and $\mathcal{V}' = \mathcal{V}' \cup \{v^s\}$.
  (d) Remove v from $\mathcal{V}'$.
(4) Return 'x and y are not conjugate'.

2. Cyclic sliding and the set of sliding circuits

This section summarises some properties of the cyclic sliding operation, the transport map, and the set of sliding circuits, which we require for proving the correctness of the algorithm from Section 1.3 and for analysing its complexity. Most of these results were obtained in Gebhardt and González-Meneses (in press) and we refer to there for further details.

Properties of cyclic sliding

Cyclic sliding does not increase the canonical length. As G is of finite type, this implies that iterated cyclic sliding starting from any $x \in G$ eventually reaches a period, that is, produces an element of $\mathrm{SC}(x)$. Moreover, iterated cyclic sliding achieves the minimal canonical length in the conjugacy class, that is, $\mathrm{SC}(x) \subseteq \mathrm{SSS}(x)$. More precisely, one has the following.

Lemma 2.1 (*Gebhardt and González-Meneses, in press, Lemma 1*). *For every $x \in G$, one has the inequalities $\inf(\mathfrak{s}(x)) \geq \inf(x)$, $\sup(\mathfrak{s}(x)) \leq \sup(x)$, and $\ell(\mathfrak{s}(x)) \leq \ell(x)$. In particular, if x is a super summit element then so is $\mathfrak{s}(x)$.*

Corollary 2.2 (*Gebhardt and González-Meneses, in press, Corollary 1*). *For any element $x \in G$, iterated application of cyclic sliding eventually reaches a period, that is, there are integers $0 \leq i < j$ such that $\mathfrak{s}^i(x) = \mathfrak{s}^j(x)$. In particular, one has $\mathfrak{s}^k(x) \in \mathrm{SC}(x)$ and $\mathfrak{s}^{j-i}(\mathfrak{s}^k(x)) = \mathfrak{s}^k(x)$ for all $k \geq i$.*

Proposition 2.3 (*Gebhardt and González-Meneses, in press, Corollary 2*). *For any $x \in G$, if $\ell(x)$ is not minimal in the conjugacy class of x, then $\ell(x) > \ell(\mathfrak{s}^m(x))$ for some positive integer $m < \|\Delta\|$. In particular, one has $\mathrm{SC}(x) \subseteq \mathrm{SSS}(x)$.*

Properties of the transport map

Under certain (mild) assumptions, the transport map respects many aspects of the Garside structure of G. In particular, transport at super summit elements preserves positive elements and powers of Δ, and it respects the partial order $\preccurlyeq$ as well as gcds with respect to $\preccurlyeq$. One has:

Proposition 2.4. *Let $x \in G$ and let $\alpha, \beta \in G$ such that $x, x^\alpha, x^\beta \in \mathrm{SSS}(x)$ and consider transports at x. Then the following hold.*

(1) *If α is positive then $\alpha^{(1)}$ is positive.*
(2) *If α is positive then $\mathfrak{p}(x) \preccurlyeq \alpha\, \mathfrak{p}(x^\alpha)$.*
(3) *If $\alpha = \Delta^k$ for $k \in \mathbb{Z}$ then $\alpha^{(1)} = \Delta^k$.*
(4) *If $\alpha \preccurlyeq \beta$ then $\alpha^{(1)} \preccurlyeq \beta^{(1)}$.*
(5) *If α is simple then $\alpha^{(1)}$ is simple.*
(6) *$(\alpha \wedge \beta)^{(1)} = \alpha^{(1)} \wedge \beta^{(1)}$.*

Proof. Claim 1 follows from (Gebhardt and González-Meneses, in press, Lemma 5) and is equivalent to Claim 2, as $\alpha^{(1)} = \mathfrak{p}(x)^{-1}\alpha\mathfrak{p}(x^\alpha)$. Claims 3, 4, 5 and 6 are special cases of (Gebhardt and González-Meneses, in press, Lemma 6, Corollary 4, Corollary 5, and Proposition 3). □

Applying iterated cyclic sliding to a conjugate y^s of $y \in \mathrm{SC}(x)$ will eventually produce another element of $\mathrm{SC}(x)$ by Corollary 2.2. The following Lemma makes this more precise: iterated transport of s along the sliding circuit of y eventually becomes periodic and this happens exactly when $\mathrm{SC}(x)$ has been reached.

Lemma 2.5 (*Gebhardt and González-Meneses, in press, Lemma 8*). *Let $x \in G$, $y \in \mathrm{SC}(x)$ and $s \in G$ such that $y^s \in \mathrm{SSS}(x)$. Let N be a positive integer such that $\mathfrak{s}^N(y) = y$ and for integers $i \geq 0$ consider the transports $s^{(iN)}$ at y. Then the following hold.*

(1) *There are integers $i_2 > i_1 \geq 0$ such that $s^{(i_1 N)} = s^{(i_2 N)}$.*
(2) *$y^s \in \mathrm{SC}(x)$ if and only if there is a positive integer k such that $s^{(kN)} = s$.*

Convexity properties and connectedness of the sliding circuits graph

It is well known that for any $x \in G$, the set of elements conjugating x to an element in $\mathrm{SSS}(x)$ is closed under $\wedge$. This has become known as *convexity* and in particular implies the existence of a minimal positive element conjugating x to an element in $\mathrm{SSS}(x)$.

Proposition 2.6. (*Franco and González-Meneses, 2003, Proposition 4.12 or Gebhardt and González-Meneses, in press, Proposition 6*). *Let $x, \alpha, \beta \in G$. If $x^\alpha, x^\beta \in \mathrm{SSS}(x)$, then $x^{\alpha\wedge\beta} \in \mathrm{SSS}(x)$.*

Corollary 2.7. (*Lee and Lee, 2008, Theorem 2.4 or Gebhardt and González-Meneses, in press, Corollary 7*). *Let $x, \alpha, \beta \in G$. If $x^\alpha, x^\beta \in \mathrm{SSS}(x)$, then $x^{\alpha\vee\beta} \in \mathrm{SSS}(x)$.*

Corollary 2.8 (*Gebhardt and González-Meneses, in press, Corollary 8*). *Let $x \in G$. There is a unique positive element $\rho(x)$ (possibly trivial) satisfying the following.*

(1) *$x^{\rho(x)} \in \mathrm{SSS}(x)$.*
(2) *$\rho(x) \preccurlyeq \alpha$ for every positive $\alpha \in G$ satisfying $x^\alpha \in \mathrm{SSS}(x)$.*

The analogous properties for $\mathrm{SC}(x)$ were shown in Gebhardt and González-Meneses (in press). They in particular imply that $\mathrm{SCG}(x)$ is a finite and connected directed graph.

Proposition 2.9 (*Gebhardt and González-Meneses, in press, Proposition 7*). *Let $x \in G$. If $x^\alpha, x^\beta \in \mathrm{SC}(x)$ for elements $\alpha, \beta \in G$, then $x^{\alpha\wedge\beta} \in \mathrm{SC}(x)$.*

Corollary 2.10 (*Gebhardt and González-Meneses, in press, Corollary 9*). *Let $x \in G$. There is a unique positive element $c(x)$ (possibly trivial) satisfying the following.*

(1) $x^{c(x)} \in \mathrm{SC}(x)$.
(2) $c(x) \preccurlyeq \alpha$ *for every positive* $\alpha \in G$ *satisfying* $x^\alpha \in \mathrm{SC}(x)$.

Corollary 2.11 (*Gebhardt and González-Meneses, in press, Corollary 10*). *For any $x \in G$, the graph $\mathrm{SCG}(x)$ is finite and connected.*

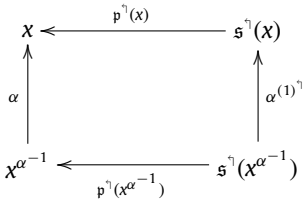
We remark that in the situation of Proposition 2.9 it is not necessarily true that $x^{\alpha\vee\beta} \in \mathrm{SC}(x)$. As an example consider the braid $x = \sigma_2\sigma_1\sigma_2\sigma_3\sigma_1\sigma_2 \in B_4$ and let $\alpha = \sigma_1$ and $\beta = \sigma_2$, whence $\alpha \vee \beta = \sigma_1\sigma_2\sigma_1$. It is easy to check that $\mathfrak{s}^3(x) = x$, $\mathfrak{s}^3(x^\alpha) = x^\alpha$ and $\mathfrak{s}^3(x^\beta) = x^\beta$. However, $\mathfrak{s}^3(x^{\alpha\vee\beta}) \neq x^{\alpha\vee\beta}$ but $\mathfrak{s}^4(x^{\alpha\vee\beta}) = \mathfrak{s}(x^{\alpha\vee\beta})$, that is, $x^{\alpha\vee\beta} \notin \mathrm{SC}(x)$.

Cyclic right sliding and right transport

Recall that in a Garside group G with Garside structure (G, P, Δ), apart from the prefix order $\preccurlyeq$, one also has the suffix order $\succcurlyeq$, defined by $a \succcurlyeq b$ if and only if $ab^{-1} \in P$. With respect to the latter, one can consider the notions of preferred suffix, cyclic right sliding and set of right sliding circuits (denoted $\mathrm{SC}^\urcorner(x)$), which are analogous to those of preferred prefix, cyclic sliding and set of sliding circuits, but refer to the partial order $\succcurlyeq$ instead of $\preccurlyeq$ (cf. Definitions 1.11, 1.12 and 1.14).

Consequently, one can also define a transport map for cyclic right sliding, as follows. We remark that, when one considers these notions with respect to $\succcurlyeq$, and tries to relate them to the analogous notions with respect to $\preccurlyeq$, one must consider conjugating elements *on the left*, meaning that a (left) conjugating element α relates x to $\alpha x \alpha^{-1} = x^{\alpha^{-1}}$.

Definition 2.12. Given $x, \alpha \in G$, we define the **right transport** of α at x under cyclic right sliding as $\alpha^{(1)^\urcorner} = \mathfrak{p}^\urcorner(x^{\alpha^{-1}})\,\alpha\,\mathfrak{p}^\urcorner(x)^{-1}$. That is, $\alpha^{(1)^\urcorner}$ is the conjugating element that makes the following diagram commutative, in the sense that the conjugating element along any closed path is trivial:



All results for cyclic (left) sliding and (left) transport hold in analogous form for cyclic right sliding and right transport; the proofs can be translated in a straightforward way. Alternatively, one can consider a different Garside structure. As shown in Gebhardt and González-Meneses (in press), (G, P^{-1}, Δ^{-1}) also is a Garside structure for G, called the **reverse Garside structure**, and cyclic right sliding and right transport with respect to (G, P, Δ) are just cyclic (left) sliding and (left) transport with respect to (G, P^{-1}, Δ^{-1}). We refer to (Gebhardt and González-Meneses, in press, Section 3.3.2) for details. In particular, we have the following right versions of Lemma 2.1 and Proposition 2.4 (1).

Lemma 2.13. *For $x \in G$, one has $\inf(\mathfrak{s}^\urcorner(x)) \geq \inf(x)$, $\sup(\mathfrak{s}^\urcorner(x)) \leq \sup(x)$, and $\ell(\mathfrak{s}^\urcorner(x)) \leq \ell(x)$. In particular, if x is a super summit element then so is $\mathfrak{s}^\urcorner(x)$.*

Proposition 2.14. *Let $x \in G$ and let $\alpha \in G$ be positive such that $x, x^{\alpha^{-1}} \in \mathrm{SSS}(x)$. Then, the right transport $\alpha^{(1)^\urcorner}$ of α at x is positive.*

A relation between cyclic (left) sliding and cyclic right sliding is given by the following result.

Proposition 2.15 (*Gebhardt and González-Meneses, in press, Proposition 5*). *Let $x \in G$. Then for any $z \in \mathrm{SSS}(x)$ one has $\mathfrak{p}^\urcorner(\mathfrak{s}(z)) \succcurlyeq \mathfrak{p}(z)$ and $\mathfrak{p}^\urcorner(z) \preccurlyeq \mathfrak{p}(\mathfrak{s}^\urcorner(z))$.*

3. Description of the algorithm

In this section we will explain the algorithms from Section 1.3 and prove their correctness. The main idea of these algorithms, as for the previous solutions to the conjugacy problem given in ElRifai and Morton (1994), Franco and González-Meneses (2003) and Gebhardt (2005), is the computation of a finite subset of the conjugacy class, which is an invariant of the conjugacy class, together with conjugating elements connecting each pair of elements of this subset. In our case, the finite set is $\mathrm{SC}(x)$, the vertex set of the connected graph $\mathrm{SCG}(x)$, and the conjugating elements will be paths in $\mathrm{SCG}(x)$.

3.1. Algorithm 3

We start by explaining Algorithm 3 from Section 1.3. We remark that analogues of this algorithm, which use other sets instead of $\mathrm{SC}(x)$, are already given in ElRifai and Morton (1994), Franco and González-Meneses (2003) and Gebhardt (2005). We explain the version given in this paper which uses the invariant $\mathrm{SC}(x)$.

It is clear from the definition that $\mathrm{SC}(x)$ is an invariant subset of the conjugacy class of x. Moreover, we will see that Algorithm 1 computes, given $x \in G$, an element $\widetilde{x} \in \mathrm{SC}(x)$, that is, $\mathrm{SC}(x)$ is non-empty. Hence, two elements x and y are conjugate if and only if $\mathrm{SC}(x) = \mathrm{SC}(y)$ or, equivalently, $\mathrm{SC}(x) \cap \mathrm{SC}(y) \neq \emptyset$. Thus, knowing how to compute $\mathrm{SC}(x)$, starting from a given element x, is sufficient to solve the conjugacy decision problem.

If we also want to solve the conjugacy search problem, that is, we want to find a conjugating element form x to y in case they are conjugate, then we can do the following. Since $\mathrm{SC}(x) = \mathrm{SC}(y)$, we just need to find an element $z \in \mathrm{SC}(x)$, a conjugating element c from x to z, and a conjugating element c_2 from y to z. Then $c\, c_2^{-1}$ conjugates x to y. In order to obtain these conjugating elements, we proceed as follows.

Suppose that $x, y \in G$ are conjugate. As we shall see, Algorithm 1 computes, given $x \in G$, an element $\widetilde{x} \in \mathrm{SC}(x)$ and a conjugating element c_1 from x to $\widetilde{x}$. Applying the same algorithm to y, we obtain an element $\widetilde{y} \in \mathrm{SC}(y) = \mathrm{SC}(x)$ and a conjugating element c_2 from y to $\widetilde{y}$. Hence, in order to obtain a conjugating element from x to y, we just need to find a conjugating element from $\widetilde{x}$ to $\widetilde{y}$. In other words, we need to know how to relate, through a conjugation, any pair of elements of $\mathrm{SC}(x)$. This is achieved thanks to the connected graph $\mathrm{SCG}(x)$, since the vertices of this graph correspond to the elements in $\mathrm{SC}(x)$, and a path between two vertices corresponds to a conjugating element from one vertex to the other.

Algorithm 3 computes a conjugating element from $\widetilde{x}$ to any other element in $\mathrm{SC}(x)$, by computing a maximal tree of the graph $\mathrm{SCG}(x)$. More precisely, the algorithm starts in step 2 by considering $\mathcal{V} = \mathcal{V}' = \{\widetilde{x}\}$ and $c_{\widetilde{x}} = 1$. The set $\mathcal{V}$ contains the elements which we know belong to $\mathrm{SC}(x)$, so at the beginning it only contains $\widetilde{x}$. The set $\mathcal{V}'$ contains the elements of $\mathcal{V}$ that have not yet been used in step 3 of the algorithm, so at the beginning $\mathcal{V}' = \mathcal{V}$. Finally, whenever a new element v is added to $\mathcal{V}$ (and also to $\mathcal{V}'$), we compute an element c_v, which is a conjugating element from $\widetilde{x}$ to v. Of course, in step 2 of the algorithm, the conjugating element from $\widetilde{x}$ to $\widetilde{x} \in \mathcal{V}$ is $c_{\widetilde{x}} = 1$.

Now step 3 does the following: For a known element of $\mathrm{SC}(x)$ which has not been processed before, that is, for some $v \in \mathcal{V}'$, it calls Algorithm 2 to compute the arrows of $\mathrm{SCG}(x)$ starting at v. For each such arrow s, it computes the endpoint v^s of the arrow. If v^s is not in $\mathcal{V}$, this means that we encountered a new element of $\mathrm{SC}(x)$, so we add it to both $\mathcal{V}$ and $\mathcal{V}'$, and at the same time compute a conjugating element from $\widetilde{x}$ to v^s: Since we know a conjugating element c_v from $\widetilde{x}$ to v and a conjugating element s from v to v^s, we can store $c_{v^s} = c_v \cdot s$ as conjugating element from $\widetilde{x}$ to v^s. Notice that the procedure checks whether $v^s = \widetilde{y}$, since in this case we have already found a conjugating element $c_{\widetilde{y}}$ from $\widetilde{x}$ to $\widetilde{y}$ as desired. Concatenating it from the left with the conjugating element from x to $\widetilde{x}$ and from the right with the conjugating element from $\widetilde{y}$ to y, this produces a conjugating element from x to y which becomes the output of the algorithm. If $\widetilde{y}$ is not encountered, we remove v from $\mathcal{V}'$ at the end of step 3 in order to record the fact that the arrows starting at v have been processed.

Notice that the procedure in step 3 is repeated while $\mathcal{V}' \neq \emptyset$. Since $\mathcal{V} \subseteq \mathrm{SC}(x)$, where $\mathrm{SC}(x)$ is a finite set, since every element of $\mathcal{V}$ is added to $\mathcal{V}'$ exactly once, and since the procedure removes one element from $\mathcal{V}'$ each time it is executed, this means that at some point we will have $\mathcal{V}' = \emptyset$ and

the procedure will stop. At this point, the arrows starting at every element of $\mathcal{V}$ have been processed (exactly once). Moreover, one has $\mathcal{V} = \mathrm{SC}(x)$: Otherwise, since the graph $\mathrm{SCG}(x)$ is connected by Corollary 2.11, there would exist some element $v \in \mathcal{V}$ and some element $w \in \mathrm{SC}(x) \setminus \mathcal{V}$ such that there is an arrow in $\mathrm{SCG}(x)$ from v to w. But since $v \in \mathcal{V}$ and $\mathcal{V}' = \emptyset$, step 3 has been applied to v, which means that w has been added to the set $\mathcal{V}$, a contradiction. Therefore, when the procedure stops, one has $\mathcal{V} = \mathrm{SC}(x)$. If $\widetilde{y}$ was not found in $\mathcal{V}$, this means that $\widetilde{y} \notin \mathrm{SC}(x)$, whence x and y are not conjugate.

Therefore, Algorithm 3 solves the conjugacy decision problem and the conjugacy search problem in Garside groups of finite type, provided that Algorithms 1 and 2 are correct.

3.2. Algorithm 1

Given $x \in G$, Algorithm 1 finds one element $\widetilde{x} \in \mathrm{SC}(x)$ and a conjugating element c such that $x^c = \widetilde{x}$. This is achieved by iterated applications of cyclic sliding to x. By Corollary 2.2, there must exist two positive integers $0 \leq i < j$ such that $\mathfrak{s}^i(x) = \mathfrak{s}^j(x)$, that is, $\mathfrak{s}^i(x) \in \mathrm{SC}(x)$. Algorithm 1 computes this element $\mathfrak{s}^i(x)$, where i is minimal. This is done by storing all the elements $\{\mathfrak{s}^m(x) \mid m \geq 0\}$, the *trajectory* of x under cyclic sliding, in a set called $\mathcal{T}$. Initially, one has $\mathcal{T} = \emptyset$ and $\widetilde{x} = x$. At the beginning of the kth iteration of the loop in step 2, one has $\mathcal{T} = \{\mathfrak{s}^0(x), \mathfrak{s}^1(x), \ldots, \mathfrak{s}^{k-2}(x)\}$ and $\widetilde{x} = \mathfrak{s}^{k-1}(x)$. If $\widetilde{x} \notin \mathcal{T}$, then $\widetilde{x}$ is added to $\mathcal{T}$ and cyclic sliding is applied to $\widetilde{x}$ before the next iteration of the loop. Otherwise, a repetition (the first one) has been found and the loop terminates.

Moreover, c is at every time a conjugating element from x to $\widetilde{x}$: At the beginning of the first iteration of the loop in step 2, $c = 1$ is a conjugating element from x to $\widetilde{x} = x$. In each iteration of the loop, the element c, which is a conjugating element from x to $\widetilde{x}$, is multiplied on the right by $\mathfrak{p}(\widetilde{x})$, yielding a conjugating element from x to $\mathfrak{s}(\widetilde{x})$, and $\widetilde{x}$ is replaced by $\mathfrak{s}(\widetilde{x})$.

Therefore, when the loop of step 2 stops, $\widetilde{x} = \mathfrak{s}^i(x) \in \mathrm{SC}(x)$ (with i minimal) and c is a conjugating element from x to $\widetilde{x}$, as desired. But notice that the conjugating element c is unnecessarily long, as it contains, as a suffix, the product of all conjugating elements along the sliding circuit containing $\widetilde{x}$. Steps 3 and 4 remove this suffix from c.

Step 3 initialises $y = \mathfrak{s}(\widetilde{x})$ and $d = \mathfrak{p}(\widetilde{x})$. The loop in step 4 checks whether $y = \widetilde{x}$, otherwise applies cyclic siding to y and multiplies d by the corresponding conjugating element, $\mathfrak{p}(y)$, in such a way that when the loop terminates, the element d equals the product of all conjugating elements along the sliding circuit containing $\widetilde{x}$. The algorithm then returns $\widetilde{x} \in \mathrm{SC}(x)$ and cd^{-1} as the conjugating element from x to $\widetilde{x}$.

3.3. Algorithm 2

Algorithm 2 is the most involved among all the procedures in this paper. It takes an element $v \in \mathrm{SC}(x)$, that is, a vertex of the graph $\mathrm{SCG}(x)$, and computes the arrows of $\mathrm{SCG}(x)$ starting at v. In other words, Algorithm 2 computes the indecomposable conjugators from v to other elements of $\mathrm{SC}(x)$. To show the correctness of each step of the algorithm, we first need to prove some theoretical results.

Recall the definition of $\rho(y)$ for $y \in G$ in Corollary 2.8 and the definition of $c(y)$ for $y \in G$ in Corollary 2.10; the existence of these elements is crucial for computing the sliding circuits graph of an element $x \in G$.

Corollary 3.1. *Let $x \in G$. Given $y \in \mathrm{SSS}(x)$ and $s \in G$, define $\rho_s = \rho_s(y) = s \cdot \rho(y^s)$. Then ρ_s is the unique $\preccurlyeq$-minimal element satisfying $s \preccurlyeq \rho_s$ and $y^{\rho_s} \in \mathrm{SSS}(x)$. Moreover, $\rho_s(y) \preccurlyeq \Delta^{\sup(s)}$.*

Proof. The first claim follows directly from Proposition 2.6 and Corollary 2.8. The second claim holds since $s \preccurlyeq \Delta^{\sup(s)}$ and $y^{\Delta^{\sup(s)}} \in \mathrm{SSS}(x)$ by Lemma 1.10. □

Corollary 3.2. *Let $x \in G$. Given $y \in \mathrm{SC}(x)$ and $s \in G$, define $c_s = c_s(y) = s \cdot c(y^s)$. Then c_s is the unique $\preccurlyeq$-minimal element satisfying $s \preccurlyeq c_s$ and $y^{c_s} \in \mathrm{SC}(x)$. Moreover, $c_s(y) \preccurlyeq \Delta^{\sup(s)}$.*

Proof. The first claim follows directly from Proposition 2.9 and Corollary 2.10. The second claim holds since $s \preccurlyeq \Delta^{\sup(s)}$ and $y^{\Delta^{\sup(s)}} \in \mathrm{SC}(x)$ by Lemma 1.15. □

Corollary 3.3. *Let $x \in G$ and let $y \in \mathrm{SC}(x)$ be a vertex of $\mathrm{SCG}(x)$. Then the following hold.*

(1) *The label of each arrow in $\mathrm{SCG}(x)$ is a simple element.*
(2) *The number of arrows in $\mathrm{SCG}(x)$ starting at y is bounded by the number of atoms of G.*

Proof. Let s be an arrow starting at y. Since $y^{\Delta} \in \mathrm{SC}(x)$ by Lemma 1.15, we have $y^{s \wedge \Delta} \in \mathrm{SC}(x)$ by Proposition 2.9. As arrows are indecomposable by definition, this implies $s = s \wedge \Delta$ proving Claim 1. For Claim 2 note that for any arrow s starting at y and any atom $a \preccurlyeq s$ we have $c_a(y) = s$ by Corollary 3.2 and the indecomposability of s. □

In order to find the arrows starting at y it is hence sufficient to consider the set of simple elements $\{c_a(y) \mid a \text{ is an atom of } G\}$. Let us then see how to compute $c_s(y)$ given $y \in \mathrm{SC}(x)$ and $s \in G$.

By Lemma 2.5, the element c_s we are looking for is a fixed point under some power of transport along the sliding circuit containing y, and we know by Proposition 2.4 (4) that transport of conjugating elements between super summit elements respects the partial order $\preccurlyeq$. The basic idea is to apply iterated transport to a suitable element p_s, which is derived from s and satisfies $s \preccurlyeq p_s \preccurlyeq c_s$, until that fixed point is reached. All we need to do is to ensure that y^{p_s} is super summit (so that $\preccurlyeq$ is respected) and that $s \preccurlyeq p_s^{(kN)}$ for a sufficiently large multiple kN of the length N of the sliding circuit containing y (so that we can be sure that we obtain the "right" fixed point, that is, one which has s as a prefix).

The first step in the computation of p_s is to find the $\preccurlyeq$-minimal element ρ_s satisfying $s \preccurlyeq \rho_s$ and $y^{\rho_s} \in \mathrm{SSS}(x)$ (cf. Corollary 3.1); this is due to Franco and González-Meneses (2003). Note that $\rho_s \preccurlyeq c_s$ since $\mathrm{SC}(x) \subseteq \mathrm{SSS}(x)$. By Corollary 3.1, we have $\rho_s = s \cdot \rho(y^s)$, so we just need to be able to compute $\rho(y^s)$. This is achieved by the following result.

Proposition 3.4. *For $x \in G$, the following algorithm computes $\rho(x)$ as in Corollary 2.8.*

(1) *Set $\rho = 1$.*
(2) *While $\inf(x^\rho) < \inf_s(x)$ or $\sup(x^\rho) > \sup_s(x)$ do:*
  (a) *Set $\rho = \rho \cdot (1 \vee (x^\rho)^{-1}\Delta^{\inf_s(x)} \vee x^\rho \Delta^{-\sup_s(x)})$.*
(3) *Return $\rho(x) = \rho$.*

If $x = y^s$ with $y \in \mathrm{SSS}(x)$, then the algorithm terminates after at most $\ell(s)\cdot\|\Delta\|$ passes through the loop.

Proof. Since some power Δ^e of Δ is central in G, we can choose a positive element α such that $x^{\rho\alpha} \in \mathrm{SSS}(x)$. Then, by Lemma 1.10, $(x^{\rho\alpha})^{-1} \in \mathrm{SSS}(x^{-1})$ and we have $\sup((x^{\rho\alpha})^{-1}) = -\inf_s(x)$ and $\sup(x^{\rho\alpha}) = \sup_s(x)$. Thus $x^\rho \preccurlyeq x^\rho\alpha = \alpha x^{\rho\alpha} \preccurlyeq \alpha\Delta^{\sup_s(x)}$, whence $x^\rho\Delta^{-\sup_s(x)} \preccurlyeq \alpha$ and, similarly, $(x^\rho)^{-1}\Delta^{\inf_s(x)} \preccurlyeq \alpha$. As $1 \preccurlyeq \alpha$, the above implies $1 \vee (x^\rho)^{-1}\Delta^{\inf_s(x)} \vee x^\rho\Delta^{-\sup_s(x)} \preccurlyeq \alpha$. Moreover, $1 \vee (x^\rho)^{-1}\Delta^{\inf_s(x)} \vee x^\rho\Delta^{-\sup_s(x)} = 1$ if and only if $\sup(x^\rho) \le \sup_s(x)$ and $\inf(x^\rho) = -\sup((x^\rho)^{-1}) \ge \inf_s(x)$, that is, if and only if $x^\rho \in \mathrm{SSS}(x)$.

Hence, at any stage of the above algorithm, the element ρ satisfies $\rho \preccurlyeq c$ for every positive element $c \in G$ such that $x^c \in \mathrm{SSS}(x)$. In particular, $\|\rho\|$ is bounded. As $\|\rho\|$ is strictly increasing at every step of the algorithm, the algorithm terminates and outputs $\rho(x)$ as claimed. Finally, if $x = y^s$ with $y \in \mathrm{SSS}(x)$, then $\rho(x) \preccurlyeq s^{-1}\Delta^{\sup(s)} \preccurlyeq \Delta^{\ell(s)}$, whence the algorithm terminates after at most $\ell(s)\cdot\|\Delta\|$ steps. □

Corollary 3.5. *Steps* 3(a) *and* 3(b) *in Algorithm* 2 *compute the element ρ_{a_t}. The body of the while loop is executed at most $\|\Delta\|$ times.*

Proof. Note that in steps 3(a) and 3(b) of Algorithm 2 we have $v \in \mathrm{SC}(x) \subseteq \mathrm{SSS}(x)$, that is, $\ell(v) = \sup_s(x) - \inf_s(x)$, $\sup(v^s) \ge \sup_s(x)$, and $\inf(v^s) \le \inf_s(x)$. In particular, $\ell(v^s) > \ell(v)$ if and only if $\inf(v^s) < \inf_s(x)$ or $\sup(v^s) > \sup_s(x)$. Hence, by Proposition 3.4, steps 3(a) and 3(b) in Algorithm 2 compute exactly $a_t \cdot \rho(v^{a_t}) = \rho_{a_t}$. Since $\ell(a_t) = 1$, the algorithm terminates after at most $\|\Delta\|$ passes through the while loop. □

As $1 \vee (x^\rho)^{-1}\Delta^{\inf_s(x)} \vee x^\rho\Delta^{-\sup_s(x)} = \left(1 \vee (x^{-1})^\rho\Delta^{-\sup_s(x^{-1})}\right) \vee \left(1 \vee x^\rho\Delta^{-\sup_s(x)}\right)$, the computation in step 2(a) of the algorithm in Proposition 3.4 can be performed efficiently using the following result.

Proposition 3.6. *If $x \in G$ such that $\sup(x) = q + r$ with $0 \le r \le \ell(x)$, then $1 \vee x\Delta^{-q}$ is the product of the leftmost r factors of the right normal form of x.*

Proof. By Lemma 1.3, we have $1 \vee x\Delta^{-q} = (1 \wedge^{\gamma} \Delta^q x^{-1})^{-1} = \left((x \wedge^{\gamma} \Delta^q)x^{-1}\right)^{-1} = x(x \wedge^{\gamma} \Delta^q)^{-1}$. As $x \wedge^{\gamma} \Delta^q$ contains all but the leftmost r factors of the right normal form of x, the claim follows. □

Next we consider the sequence of iterated transports along the sliding circuit which contains the element y. This sequence will eventually become periodic; we are interested in the periodic part.

Definition 3.7. Let $x \in G$, $y \in \mathrm{SC}(x)$ and $u \in G$ such that $y^u \in \mathrm{SSS}(x)$ and let N be the length of the sliding circuit containing y, that is, let N be the smallest positive integer such that $\mathfrak{s}^N(y) = y$. For integers $i \geq 0$ consider the transports $u^{(iN)}$ at y. By Lemma 2.5, there are integers $i_2 > i_1 \geq 0$ such that $u^{(i_1 N)} = u^{(i_2 N)}$. Let i_1 and i_2 be minimal subject to this condition and define $l(u) = i_2 - i_1$ and $F(u) = \{u^{(iN)} \mid i_1 \leq i < i_2\}$.

Lemma 3.8. *In the situation of Definition 3.7, the following hold.*

(1) *For all $k \geq i_1$, one has $u^{((k+l(u))N)} = u^{(kN)}$ and $l(u)$ is the minimal positive integer satisfying this condition.*
In particular, for all $v \in F(u)$ and all $i \in \mathbb{N}$ one has $v^{(il(u)N)} = v$, whence $y^v \in \mathrm{SC}(x)$.
(2) *$1 \in F(u)$ if and only if $F(u) = \{1\}$.*
(3) *$y^u \in \mathrm{SC}(x)$ if and only if $u \in F(u)$.*
(4) *$F(u) = \{u^{(iN)} \mid i \in \mathbb{N}$ and $y^{u^{(iN)}} \in \mathrm{SC}(x)\}$.*

Proof. Claim 1 follows by induction on k and Lemma 2.5. For Claim 2, assume $1 \in F(u)$ and choose k minimal such that $u^{(kN)} = 1$. Then, $u^{(k'N)} = 1$ for all $k' \geq k$, that is, $i_1 = k$ and $i_2 - i_1 = 1$. In particular, $F(u) = \{1\}$. The converse is trivial, so Claim 2 is shown. Claim 3 follows with Lemma 2.5. Claim 4 follows from Claim 1 and Lemma 2.5 together with the minimality of i_1. □

In other words, the periodic part $F(u)$ of the sequence of iterated transports contains those iterated transports $u^{(iN)}$ of u along the sliding circuit of y, which are fixed by repeated transport along the sliding circuit; these iterated transports $u^{(iN)}$ are precisely those satisfying $y^{u^{(iN)}} \in \mathrm{SC}(x)$.

Under certain conditions, we can use the set $F(u)$ to draw conclusions about the element c_s (cf. Corollary 3.2) we are interested in; the following two lemmata make this statement precise.

Lemma 3.9. *Let $x \in G$, $y \in \mathrm{SC}(x)$, $s \in G$ and denote $c_s = c_s(y)$. Let N be the length of the sliding circuit containing y, that is, let N be the smallest positive integer such that $\mathfrak{s}^N(y) = y$. If $c_s \preccurlyeq c_s^{(iN)}$ for some $i > 0$ then $c_s^{(iN)} = c_s$.*

Proof. First notice that $c_s \in F(c_s)$ by its definition (cf. Corollary 3.2) and Lemma 3.8. Now assume that $c_s^{(iN)} = c_s\gamma$ with a positive element γ. By induction, $c_s\gamma \preccurlyeq c_s^{(kiN)}$ for all $k \geq 1$ by Proposition 2.4 (4). Again using Lemma 3.8, we have $c_s \preccurlyeq c_s\gamma \preccurlyeq c_s^{(l(c_s)iN)} = c_s$, that is, $\gamma = 1$. □

Lemma 3.10. *Let $x \in G$, $y \in \mathrm{SC}(x)$, $s \in P$ and denote $c_s = c_s(y)$. Assume that u is a positive element satisfying $u \preccurlyeq c_s$ and $y^u \in \mathrm{SSS}(y)$ and assume further that $F = F(u) \neq \{1\}$.*

(1) *If there exists $v \in F$ such that $s \preccurlyeq v$ then $c_s = v$.*
(2) *If $s \not\preccurlyeq v$ for all $v \in F$, then c_s is not an indecomposable conjugator starting at y.*

Proof. First note that by Proposition 2.4 (4), we have $u^{(i)} \preccurlyeq c_s^{(i)}$ for all $i > 0$.

Assume first that there exists an element $v \in F$ such that $s \preccurlyeq v$. We have $y^v \in \mathrm{SC}(x)$ by Lemma 3.8. The minimality of c_s (Corollary 3.2) then implies $c_s \preccurlyeq v$. Let N be the smallest positive integer such that $\mathfrak{s}^N(y) = y$. Now $v = u^{(iN)}$ for some i, whence $c_s \preccurlyeq v = u^{(iN)} \preccurlyeq c_s^{(iN)}$. Lemma 3.9 then yields $v = c_s$ and Claim 1 is shown.

Now assume that $s \not\preccurlyeq v$ for all $v \in F$ and let i be a multiple of $l(c_s)$ sufficiently large so that $v = u^{(iN)} \in F$. Since $1 \notin F$, we have $v \neq 1$ and $y^v \in \mathrm{SC}(x)$ by Lemma 3.8. Moreover, $v = u^{(iN)} \preccurlyeq c_s^{(iN)} = c_s$ and $v \neq c_s$, since $s \not\preccurlyeq v$ but $s \preccurlyeq c_s$. Hence, c_s is not an indecomposable conjugator starting at y and Claim 2 is shown. □

Recall that we are trying to compute the arrows of $\mathrm{SCG}(x)$ starting at y. In Algorithm 2, we start with an atom a and we try to see if there is an arrow c starting at y such that $a \preccurlyeq c$ or, equivalently, such that $\rho_a \preccurlyeq c_a \preccurlyeq c$. ($\rho_a$ and c_a were defined in Corollaries 3.1 and 3.2) As arrows are indecomposable,

$c_a \preccurlyeq c$ would imply $c_a = c$, that is, c_a is the only candidate for the arrow c. Lemma 3.10 says that if $F(\rho_a) \neq \{1\}$ then we will have no problem, since either c_a can be computed by iterated transport of ρ_a along the sliding circuit containing y, or we can be sure that there is no such arrow c, since c_a is decomposable. Unfortunately, it may occur that $F(\rho_a) = \{1\}$, as we can see in the following example:

Example 3.11. Consider in the Artin braid group B_5 the elements $y = x = \Delta \cdot \sigma_2\sigma_1\sigma_4\sigma_3\sigma_4 \cdot \sigma_1$, in left normal form as written, and $s = \sigma_3\sigma_2\sigma_1$. It is easy to check that $\mathfrak{s}^6(y) = y$, that is, $y \in \mathrm{SC}(x)$. Since $y^s = \Delta \cdot \sigma_1\sigma_3 \cdot \sigma_3\sigma_2\sigma_1\sigma_2$ is in left normal form as written, $y^s \in \mathrm{SSS}(x)$, that is, $\rho_s = s$.

However, $s^{(1)} = \mathfrak{p}(y)^{-1}s\mathfrak{p}(y^s) = 1$ and hence $F(s) = \{1\}$, that is, the requirements of Lemma 3.10 are not satisfied.

The above example shows that one could possibly have $F(\rho_a) = \{1\}$ for some atom a in the situation of Algorithm 2. In this case, Lemma 3.10 would not guarantee that iterated transport is sufficient to find c_a or to be sure that c_a is decomposable. Let us now see that there is another condition which also ensures that either c_a can be computed by iterated transport, or that it is decomposable; it is given by the corollary to the following result.

Lemma 3.12. *Let $x \in G$ and $v \in \mathrm{SC}(x)$. Let $s \neq 1$ be a positive element such that $v^s \in \mathrm{SSS}(x)$. If $s^{(k)} = 1$ for some $k \geq 1$, then $s \wedge \mathfrak{p}(v) \neq 1$.*

Proof. This proof parallels the one of (Gebhardt, 2005, Lemma 4.11). Denote $w = v^s$. By hypothesis

$$s^{(k)} = \left(\mathfrak{p}(v)\mathfrak{p}(\mathfrak{s}(v)) \cdots \mathfrak{p}(\mathfrak{s}^{k-1}(v))\right)^{-1} s \left(\mathfrak{p}(w)\mathfrak{p}(\mathfrak{s}(w)) \cdots \mathfrak{p}(\mathfrak{s}^{k-1}(w))\right) = 1,$$

that is,

$$s \left(\mathfrak{p}(w)\mathfrak{p}(\mathfrak{s}(w)) \cdots \mathfrak{p}(\mathfrak{s}^{k-1}(w))\right) = \mathfrak{p}(v)\mathfrak{p}(\mathfrak{s}(v)) \cdots \mathfrak{p}(\mathfrak{s}^{k-1}(v)).$$

We will show the result by induction on k. If $k = 1$, one has $s\mathfrak{p}(w) = \mathfrak{p}(v)$, hence $s \wedge \mathfrak{p}(v) = s \neq 1$. Suppose the result is true for $k-1$, and consider $s^{(1)}$. We can assume that $s^{(1)} \neq 1$, otherwise the result would hold by applying the case $k = 1$. But we have $(s^{(1)})^{(k-1)} = 1$, so by induction hypothesis $s^{(1)} \wedge \mathfrak{p}(\mathfrak{s}(v)) \neq 1$.

Recall that the transport $t^{(1)}$ of an element t at v satisfies $t^{(1)} = \mathfrak{p}(v)^{-1}t\mathfrak{p}(v^t)$. For $t = \mathfrak{p}(v)$ this yields $\mathfrak{p}(v)^{(1)} = \mathfrak{p}(v^{\mathfrak{p}(v)}) = \mathfrak{p}(\mathfrak{s}(v))$. As the transport preserves $\wedge$ by Proposition 2.4 (6), one hence has $(s \wedge \mathfrak{p}(v))^{(1)} = s^{(1)} \wedge \mathfrak{p}(v)^{(1)} = s^{(1)} \wedge \mathfrak{p}(\mathfrak{s}(v)) \neq 1$, which implies $s \wedge \mathfrak{p}(v) \neq 1$ by Proposition 2.4 (3). □

Corollary 3.13. *Let $x \in G$ and $v \in \mathrm{SC}(x)$. Let a be an atom such that $a \not\preccurlyeq \mathfrak{p}(v)$. Then either $F(\rho_a) \neq \{1\}$ or c_a is not an indecomposable conjugator starting at v.*

Proof. Suppose that $F(\rho_a) = \{1\}$. This means that some iterated transport $(\rho_a)^{(k)} = 1$ for some $k \geq 1$. By Lemma 3.12 we have $\rho_a \wedge \mathfrak{p}(v) \neq 1$. Hence there must exist an atom b such that $b \preccurlyeq \rho_a \wedge \mathfrak{p}(v)$. Since $b \preccurlyeq \mathfrak{p}(v)$ and $v^{\mathfrak{p}(v)} \in \mathrm{SC}(x)$, it follows that $c_b \preccurlyeq \mathfrak{p}(v)$. On the other hand, since $b \preccurlyeq \rho_a \preccurlyeq c_a$, it follows that $c_b \preccurlyeq c_a$. But one cannot have $c_b = c_a$, otherwise $a \preccurlyeq c_a = c_b \preccurlyeq \mathfrak{p}(v)$, which is not possible by hypothesis. Therefore, c_b is a proper prefix of c_a, which means that c_a is not an indecomposable conjugator starting at v. □

Recall that if $F(\rho_a) \neq \{1\}$ then either c_a can be found by iterated transport or c_a is not indecomposable by Lemma 3.10. Hence, if $a \not\preccurlyeq \mathfrak{p}(v)$, we just need iterated transport in order to compute or to discard c_a. The case that remains to be dealt with is the case $a \preccurlyeq \mathfrak{p}(v)$ and $F(\rho_a) = \{1\}$.

We will now consider the more general situation that $F(\rho_s) = \{1\}$ for some element $s \in G$. Iterated transport of ρ_s reaches the "wrong" fixed point in this situation. The solution is to apply iterated transport not to ρ_s itself, but to a related element p satisfying $\rho_s \preccurlyeq p \preccurlyeq c_s$ for which the existence of $v \in F(p)$ with $s \preccurlyeq v$ is guaranteed. To this end we introduce the notion of the "pullback" of an element s, defined as the $\preccurlyeq$-minimal among the elements whose transport has s as a prefix.

Definition 3.14. Let $x \in G, z \in \mathrm{SC}(x), y = \mathfrak{s}(z)$ and let $s \in G$ be positive. By Propositions 2.6 and 2.4(6), there exists a unique $\preccurlyeq$-minimal positive element $s_{(1)} \in G$ satisfying $z^{s_{(1)}} \in \mathrm{SSS}(x)$ and $s \preccurlyeq (s_{(1)})^{(1)}$, where $(s_{(1)})^{(1)}$ indicates the transport of $s_{(1)}$ at z. We call $s_{(1)}$ the **pullback** of s at y.

For any integer $k > 1$ we define recursively the k-fold pullback $s_{(k)} = (s_{(k-1)})_{(1)}$ of s at y. Note that $(s_{(k-1)})_{(1)}$ indicates the pullback of $s_{(k-1)}$ at the unique element w in the sliding circuit of y satisfying $\mathfrak{s}^{k-1}(w) = y$. We also define $s_{(0)} = s$.

Lemma 3.15. *Let $x \in G$, $z \in \mathrm{SC}(x)$, $y = \mathfrak{s}^k(z)$ for a positive integer k and let $s \in G$ be positive. Then, the k-fold pullback $s_{(k)}$ of s at y is the $\preccurlyeq$-minimal positive element satisfying $s \preccurlyeq (s_{(k)})^{(k)}$ and $z^{s_{(k)}} \in \mathrm{SSS}(x)$.*

Proof. The claim holds for $k = 1$ by definition of the pullback. Suppose the claim is true for $k-1$. By Proposition 2.4 (4), one then has $s \preccurlyeq (s_{(k-1)})^{(k-1)} \preccurlyeq (((s_{(k-1)})_{(1)})^{(1)})^{(k-1)} = (s_{(k)})^{(k)}$. Moreover, if α is a positive element such that $s \preccurlyeq \alpha^{(k)}$ and $z^{\alpha} \in \mathrm{SSS}(x)$, then by Proposition 2.4 and Lemma 2.1, $\alpha^{(1)}$ is a positive element satisfying $s \preccurlyeq (\alpha^{(1)})^{(k-1)}$ and $\mathfrak{s}(z)^{\alpha^{(1)}} = \mathfrak{s}(z^{\alpha}) \in \mathrm{SSS}(x)$. Hence, $s_{(k-1)} \preccurlyeq \alpha^{(1)}$ by induction. By the definition of the pullback of $s_{(k-1)}$, we then have $s_{(k)} = (s_{(k-1)})_{(1)} \preccurlyeq \alpha$, as we wanted to show. □

Lemma 3.16. *Let $x \in G$, $z \in \mathrm{SC}(x)$, $y = \mathfrak{s}^k(z)$ for a positive integer k and let $s, t \in G$ such that $1 \preccurlyeq s \preccurlyeq t$. Then, the k-fold pullbacks $s_{(k)}$ of s and $t_{(k)}$ of t at y satisfy $s_{(k)} \preccurlyeq t_{(k)}$.*

Proof. By Lemma 3.15, we have $t \preccurlyeq (t_{(k)})^{(k)}$ and $z^{t_{(k)}} \in \mathrm{SSS}(x)$. Hence $s \preccurlyeq t \preccurlyeq (t_{(k)})^{(k)}$ and, again using Lemma 3.15, we obtain $s_{(k)} \preccurlyeq t_{(k)}$ as we wanted to show. □

Lemma 3.17. *Let $x \in G$, $z \in \mathrm{SC}(x)$, $y = \mathfrak{s}(z)$ and let $s \in G$ be positive. Then the pullback $s_{(1)}$ of s at y satisfies $s_{(1)} \preccurlyeq \Delta^{\sup(s)}$. In particular, the pullback of a simple element is simple.*

Proof. Let $q = \sup(s) \geq 0$ and consider transport at z. We have $s \preccurlyeq \Delta^q = (\Delta^q)^{(1)}$ by Proposition 2.4 (3). Moreover, Δ^q is positive and $z^{\Delta^q} \in \mathrm{SSS}(x)$ by Lemma 1.10. By $\preccurlyeq$-minimality of $s_{(1)}$, we obtain $s_{(1)} \preccurlyeq \Delta^q$ as claimed. □

We remark that in general $\Delta^q_{(1)} \neq \Delta^q$. (This is no surprise, as $s^{(1)} = \Delta^q$ does not imply $s = \Delta^q$.) Consider, for instance, the braid $y = \sigma_3\sigma_2\sigma_1\sigma_3\sigma_2\sigma_4 \in B_5$. It is easy to check that $\mathfrak{s}^4(y) = y$. Denoting $z = \mathfrak{s}^3(y) = \sigma_2\sigma_1\sigma_3\sigma_2\sigma_3\sigma_4$ and $s = \sigma_3\sigma_2\sigma_1\sigma_4\sigma_3\sigma_2\sigma_4\sigma_3\sigma_4$ one easily verifies that the transport $s^{(1)}$ of s at z is Δ, that is, the pullback $\Delta_{(1)}$ of Δ at y satisfies $\Delta_{(1)} \preccurlyeq s \neq \Delta$. (In fact, one has $\Delta_{(1)} = s$.)

The next result shows how one can use pullbacks to compute c_s in the case in which $F(\rho_s) = \{1\}$ may occur.

Proposition 3.18. *Let $x \in G$, $v \in \mathrm{SC}(x)$ and let N be the length of the sliding circuit of v, that is, let N be the smallest positive integer such that $\mathfrak{s}^N(v) = v$. Let $s \in P \setminus \{1\}$ such that $v^s \in \mathrm{SSS}(x)$ and for integers $k \geq 0$ consider the iterated pullbacks $s_{(kN)}$ at v. Let $i \geq 0$ be such that $s_{(iN)} = s_{(jN)}$ for some $j > i$. Then c_s is the only element in $F(s_{(iN)})$ which admits s as a prefix. In particular, $F(s_{(iN)}) \neq \{1\}$.*

Proof. First note that by Lemma 3.17, we have $1 \preccurlyeq s_{(kN)} \preccurlyeq \Delta^{\sup(s)}$ for all $k \geq 0$. As G is of finite type, the number of such elements is finite, whence there exist integers $i \geq 0$ and $j > i$ such that $s_{(iN)} = s_{(jN)}$.

Let $m = i(j-i) \geq i$ and denote $p = s_{(mN)}$. Notice that iterated N-fold pullback becomes periodic of period $j-i$ starting from the ith term, hence $p_{(k(j-i)N)} = p$ for all $k \geq 0$, that is, $p = s_{(k(j-i)N)}$ for all $k \geq i$. Now recall from Lemma 2.5 that, since $v^{c_s} \in \mathrm{SC}(x)$, we have $(c_s)^{(tN)} = c_s$ for some $t \geq 1$. Consider then $M > i$ to be a multiple of t, big enough so that $p^{(M(j-i)N)} \in F(p)$. According to Lemma 3.15, $p = s_{(M(j-i)N)}$ is the $\preccurlyeq$-minimal positive element such that $s \preccurlyeq p^{(M(j-i)N)}$. This implies that $F(p) \neq \{1\}$ and that $F(p)$ contains an element admitting s as a prefix. Moreover, $s \preccurlyeq c_s = (c_s)^{(M(j-i)N)}$, where the equality in the last step holds since M is a multiple of t. By the minimality of p one finally has $p \preccurlyeq c_s$. We can then apply Lemma 3.10 to p, and conclude that $c_s = p^{(M(j-i)N)} \in F(p)$. Uniqueness also follows from Lemma 3.10.

It only remains to be shown that $F(p) = F(s_{(iN)})$, that is $F(s_{(mN)}) = F(s_{(iN)})$ for m as above; indeed, we will show that $F(s_{(kN)}) = F(s_{(iN)})$ for all $k \geq i$. Since iterated N-fold pullback is periodic of period $j-i$ from the ith term, we can assume $i < k < j$.

We have $s_{(iN)} \preccurlyeq (s_{(kN)})^{((k-i)N)}$ and also $s_{(kN)} \preccurlyeq (s_{(jN)})^{((j-k)N)} = (s_{(iN)})^{((j-k)N)}$ by Lemma 3.15. Applying $(k-i)N$-fold transport to the second expression and using Proposition 2.4 (4), one obtains $(s_{(kN)})^{((k-i)N)} \preccurlyeq (s_{(iN)})^{((j-i)N)}$. Together with the first expression, this yields $s_{(iN)} \preccurlyeq (s_{(kN)})^{((k-i)N)} \preccurlyeq (s_{(iN)})^{((j-i)N)}$.

Using Proposition 2.4 (4) again, we can for any $K \geq 0$ apply K-fold transport to this expression and we see that $(s_{(iN)})^{(K)} \preccurlyeq (s_{(kN)})^{(K+(k-i)N)} \preccurlyeq (s_{(iN)})^{(K+(j-i)N)}$ for all $K \geq 0$. That is, for any integer K

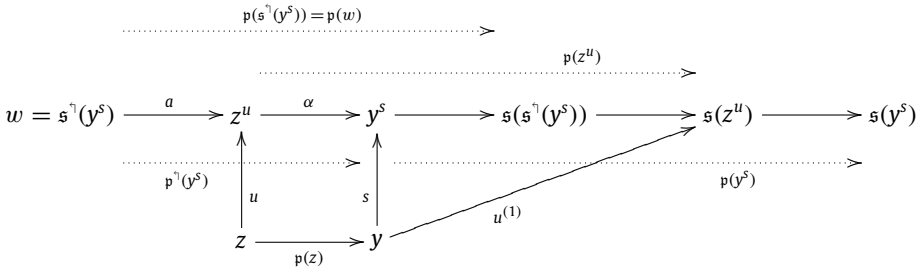
large enough so that $s' = (s_{(iN)})^{(K)} \in F(s_{(iN)})$, we have $c_{s'} = s' \preccurlyeq (s')^{((j-i)N)}$ and hence $s' = (s')^{((j-i)N)}$ by Lemma 3.9 (where $s' = c_{s'}$ is chosen as the element s in the statement of the lemma). Hence, the above inequality implies $s' = (s_{(kN)})^{(K+(k-i)N)}$. As this is true for all sufficiently large K, we have $F(s_{(iN)}) = F(s_{(kN)})$. In particular, $F(p) = F(s_{(iN)})$, whence $c_s \in F(s_{(iN)})$, as we wanted to show. □

The following result allows us to compute pullbacks in the situation of Algorithm 2.

Proposition 3.19. *Let $x \in G$, $z \in \mathrm{SC}(x)$, $y = \mathfrak{s}(z)$ and let $s \in G$ be positive such that y^s is super summit. Then the pullback of s at y, as given in Definition 3.14, is*

$$s_{(1)} = \left(\mathfrak{p}(z)\, s\, \mathfrak{p}^{\curlywedge}(y^s)^{-1}\right) \vee 1 .$$

Proof. Let $u = \left(\mathfrak{p}(z)\, s\, \mathfrak{p}^{\curlywedge}(y^s)^{-1}\right) \vee 1$. We show that u satisfies the defining properties of $s_{(1)}$. The following commutative diagram illustrates the situation; all conjugating elements corresponding to arrows will be shown to be positive.



Claim 1: $z^u \in \mathrm{SSS}(x)$.

Proof: As $y^s \in \mathrm{SSS}(x)$, we have $z^{\mathfrak{p}(z)\, s\, \mathfrak{p}^{\curlywedge}(y^s)^{-1}} = \mathfrak{s}^{\curlywedge}(y^s) \in \mathrm{SSS}(x)$ by Lemma 2.13. Then, Corollary 2.7 implies $z^u \in \mathrm{SSS}(x)$, since $u = \left(\mathfrak{p}(z)\, s\, \mathfrak{p}^{\curlywedge}(y^s)^{-1}\right) \vee 1$.

Claim 2: u is a positive element and $s \preccurlyeq u^{(1)}$.

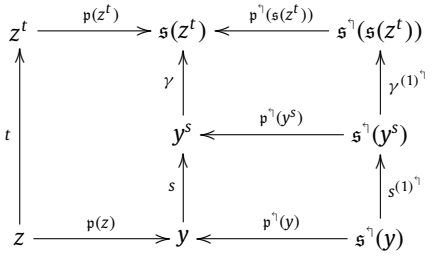
Proof: The element u is positive by definition. Moreover, defining $\alpha = \mathfrak{p}^{\curlywedge}(y^s) \wedge^{\curlywedge} \mathfrak{p}(z)\, s$, we have $u = \left(\mathfrak{p}(z)\, s\, \mathfrak{p}^{\curlywedge}(y^s)^{-1}\right) \vee \left(\mathfrak{p}(z)\, s\, s^{-1}\, \mathfrak{p}(z)^{-1}\right) = \mathfrak{p}(z)\, s \left(\mathfrak{p}^{\curlywedge}(y^s)^{-1} \vee s^{-1}\, \mathfrak{p}(z)^{-1}\right) = \mathfrak{p}(z)\, s\, \alpha^{-1}$ by Lemma 1.3.

Since α is a positive suffix of $\mathfrak{p}^{\curlywedge}(y^s)$, we can write $\mathfrak{p}^{\curlywedge}(y^s) = a\,\alpha$ for some positive a. Denoting $w = \mathfrak{s}^{\curlywedge}(y^s)$, we have $w^a = \mathfrak{s}^{\curlywedge}(y^s)^{\mathfrak{p}^{\curlywedge}(y^s)\alpha^{-1}} = (y^s)^{\alpha^{-1}} = y^{\mathfrak{p}(z)^{-1}u} = z^u$. Hence $w^a = z^u \in \mathrm{SSS}(x)$ by Claim 1.

By Proposition 2.15, $a\,\alpha = \mathfrak{p}^{\curlywedge}(y^s) \preccurlyeq \mathfrak{p}(\mathfrak{s}^{\curlywedge}(y^s)) = \mathfrak{p}(w)$. On the other hand, as a is positive, $w \in \mathrm{SSS}(x)$ and $w^a \in \mathrm{SSS}(x)$, we obtain with Proposition 2.4 (2) that $\mathfrak{p}(w) \preccurlyeq a\, \mathfrak{p}(w^a)$. Therefore $a\,\alpha \preccurlyeq \mathfrak{p}(w) \preccurlyeq a\, \mathfrak{p}(w^a)$, so $\alpha \preccurlyeq \mathfrak{p}(w^a) = \mathfrak{p}(z^u)$. This means $1 \preccurlyeq \alpha^{-1}\mathfrak{p}(z^u)$, whence we finally obtain $s \preccurlyeq s\,\alpha^{-1}\, \mathfrak{p}(z^u) = \mathfrak{p}(z)^{-1}\, u\, \mathfrak{p}(z^u) = u^{(1)}$.

Claim 3: If t is a positive element such that $z^t \in \mathrm{SSS}(x)$ and $s \preccurlyeq t^{(1)}$, then $u \preccurlyeq t$.

Proof: Write $t^{(1)} = s\gamma$ for some positive element γ and apply cyclic right sliding to y, y^s and $y^{t^{(1)}} = \mathfrak{s}(z^t)$, as shown in the following commutative diagram.



We obtain $t = \mathfrak{p}(z)\, s\, \gamma\, \mathfrak{p}(z^t)^{-1} = \mathfrak{p}(z)\, s\, \mathfrak{p}^{\curlywedge}(y^s)^{-1}\, \gamma^{(1)^{\curlywedge}} \left[\mathfrak{p}^{\curlywedge}(\mathfrak{s}(z^t))\, \mathfrak{p}(z^t)^{-1}\right]$, where $\gamma^{(1)^{\curlywedge}}$ is positive by Proposition 2.14 and the factor in brackets is positive by Proposition 2.15. Therefore, we have $\mathfrak{p}(z)\, s\, \mathfrak{p}^{\curlywedge}(y^s)^{-1} \preccurlyeq t$, and since t is positive, one finally obtains $u = \left(\mathfrak{p}(z)\, s\, \mathfrak{p}^{\curlywedge}(y^s)^{-1}\right) \vee 1 \preccurlyeq t$. □

Example 3.20. Consider the situation from Example 3.11. The trajectory of $y = \Delta \cdot \sigma_2\sigma_1\sigma_4\sigma_3\sigma_4 \cdot \sigma_1$ under cyclic sliding has length $N = 6$. Computing iterated pullbacks of $\rho_s = s = \sigma_3\sigma_2\sigma_1$ at y we obtain $s_{(12)} = s_{(6)} = \sigma_3\sigma_4$. Hence, using the notation from Proposition 3.18, we have $i = 1$ and $j = 2$.

Computing iterated transports of $p = s_{(iN)} = s_{(6)} = \sigma_3\sigma_4$, we obtain $p^{(12)} = p^{(6)} = \sigma_3\sigma_2\sigma_1\sigma_4$. Hence, we have $F(p) = \{p^{(6)}\}$ and as $s \preccurlyeq p^{(6)}$, we obtain $c_s = p^{(6)} = \sigma_3\sigma_2\sigma_1\sigma_4$.

Note that $p \notin F(p)$, that is, computing iterated transports is necessary even after reaching a stable loop under iterated N-fold pullback.

The results obtained in this section ensure that step 3(c) of Algorithm 2, if executed, will compute an element $(\rho_{a_t})_{(iN)}$, one of whose iterated transports is precisely c_{a_t}. This computation is only done whenever $a_t \preccurlyeq \mathfrak{p}(v)$, which is the only case, as we saw above, in which we cannot be sure to find c_{a_t} or to be able to discard c_{a_t} as decomposable using $F(\rho_{a_t})$. Note, in particular, that computing pullbacks is not necessary if v is rigid (or, by (Gebhardt and González-Meneses, in press, Theorem 1) equivalently, has a rigid conjugate). The algorithm continues in step 3(d) by applying iterated transport to the corresponding element (either ρ_{a_t} or $(\rho_{a_t})_{(iN)}$) until the first repetition occurs. Then, step 3(e) checks whether any of the elements in $F(\rho_{a_t})$ respectively $F((\rho_{a_t})_{(iN)})$ admits a_t as a prefix, in which case it will precisely be c_{a_t} by Lemma 3.10. If a_t does not occur as a prefix, then c_{a_t} is not indecomposable by Lemma 3.10, Corollary 3.13 and Proposition 3.18.

However, even if c_{a_t} occurs as an element of $F(\rho_{a_t})$ respectively $F((\rho_{a_t})_{(iN)})$, it is not necessarily an indecomposable conjugator. The latter property is checked in step 3(e)i: The set *Atoms* will eventually contain the atoms a_k such that c_{a_k} is an indecomposable conjugator starting at v and $k = \max\{i \mid a_i \preccurlyeq c_{a_k}\}$. Suppose that we have computed c_{a_t} for some atom a_t. If t is not the biggest index among the atoms dividing c_{a_t}, then we can discard c_{a_t} at this step since, if it is indecomposable, it will appear again in a further step of the algorithm, when the mentioned atom is processed. On the other hand, if t is the maximal index among the atoms dividing c_{a_t} but c_{a_t} is decomposable, then there must exist some indecomposable $c_{a_l} \preccurlyeq c_{a_t}$, where $l < t$ is maximal among the atoms dividing c_{a_l}. In particular, a_l has been processed before a_t, and we must have $a_l \in \mathit{Atoms}$. Therefore, if $a_k \not\preccurlyeq c_{a_t}$ for all $a_k \in \mathit{Atoms}$ and also for all $k > t$, we can be sure that c_{a_t} is indecomposable, and we can add a_t to the set *Atoms*. This is what is done in step 3(e)i, hence Algorithm 2 computes the arrows starting at v, as claimed.

4. Complexity of the algorithms

4.1. Computing in Garside groups

In this section, we will describe how one can perform all the computations required by our algorithms in any Garside group of finite type, provided some basic operations on simple elements can be performed. We refer the reader to (Michel, 1997) for a similar approach.

We remark that in a particular Garside group there may be specific algorithms having better complexity than the generic ones we describe below. This is in particular the case for braid groups (see Epstein et al., 1992 and Birman et al., 1998). Hence one should not use the algorithms below if one just needs to make computations in braid groups.

4.1.1. Context of the complexity analyses

We analyse the complexity of algorithms in terms of two numerical invariants of the Garside group G: firstly the number λ of atoms of G, and secondly the maximal length $\|\Delta\|$ of an expression of the Garside element in terms of atoms. Considering the posets of simple elements with respect to the partial orders $\preccurlyeq$ and $\succcurlyeq$, these invariants give the number of minimal elements respectively the height of the poset.

Example 4.1.

(1) Consider the Artin braid group B_n on n strands. There are $n!$ simple elements in B_n. We have $\lambda = n - 1$ and $\|\Delta\| = \frac{1}{2}n(n-1)$.

(2) Consider the braid group BKL_n on n strands in the Birman–Ko–Lee presentation. There are $\frac{(2n)!}{n!(n+1)!}$ simple elements in BKL_n. We have $\lambda = \frac{1}{2}n(n-1)$ and $\|\Delta\| = n - 1$.

Note that in both cases, λ and $\|\Delta\|$ grow polynomially in n, whereas the number of simple elements grows exponentially in n.

Although we are in this section not specifically interested in braid groups, Example 4.1 illustrates that precalculating information for all simple elements, and even producing an explicit list of all simple elements, should be avoided; the number of simple elements even in relatively basic Garside groups can be impractically large. In the case of the braid groups, any such precomputation would result in both the time and the space complexity of the algorithm being exponential in n. The example also shows that the number of simple elements would be a very bad parameter to use for the purpose of a complexity analysis.

We will frequently have to compute a sequence of elements and test for repetitions. In order to do that efficiently, we use *hash tables*.

Hash tables

Let $K \in \mathbb{N}$, let $D \in \mathbb{N}$ be coprime to K, and let $h : M \to \{0, \dots, K-1\}$. A **hash table** of size K for storing elements of M is an array T of size K, whose fields are labelled by $0, \dots, K-1$, which is initially empty. The function h is called the **hash function.**

The **hash positions** for an element $m \in M$ are the fields $h_i(m) = h(m) + iD \pmod K$ of T for $i = 0, 1, \dots$; the position $h_0(m)$ is called the **primary hash position** of m, the positions $h_i(m)$ with $i > 0$ are called the **secondary hash positions** of m.

An element $m \in M$ is inserted into T by storing it in the first of its hash positions which is empty. In order to test whether an element m is contained in T it is sufficient to examine its hash positions in order, until either the element m is found or an empty hash position is encountered. The probability of secondary hash positions being used can be made arbitrarily small by choosing K large compared to the number of elements stored in T. If this probability is negligible, testing whether an element is in T, and inserting it if it is not, on average requires the computation of one hash value and possibly one element comparison.

For more information on hashing we refer to Knuth (1998).

4.1.2. Basic assumptions

Assumption 4.2. Let G be a Garside group of finite type. We assume that the Garside element Δ of G and the list $\mathcal{A} = \{a_1, \dots, a_\lambda\}$ of atoms of G are known and that the following operations can be performed effectively; we consider the cost of these operations to be $O(C)$.

(H) Given a simple element s, compute a hash value for s.
(Op) Given an atom $a \in \mathcal{A}$ and a simple element s, test whether $a \preccurlyeq s$ (respectively $s \succcurlyeq a$) and, if yes, compute the simple element $a^{-1}s$ (respectively $s\,a^{-1}$).

We further assume that elements of G are stored as products (sequences) of simple elements or inverses of simple elements. Then, two elements consisting of at most k such factors can be multiplied at a cost of $O(k)$ simply by concatenating the corresponding sequences.

We remark that we also could have considered the following additional basic operations:

(Op1) Given a simple element s, test whether $s = 1$.
(Op2) Given two simple elements s and t, test whether $s = t$.
(Op3) Given an atom $a \in \mathcal{A}$ and a simple element s, test whether sa (resp. as) is simple and, if yes, compute the simple element sa (resp. as).

However, if s is a simple element, then $s = 1$ is equivalent to $a_i \not\preccurlyeq s$ for all $i = 1, \dots, \lambda$, where the latter condition can be tested using the operation (Op) at most λ times. Hence, (Op1) can be realised in terms of (Op) at a cost of $O(C\lambda)$. We will moreover see below that (Op2) and (Op3) can be realised in terms of (Op) at a cost of $O(C\lambda\ \|\Delta\|)$. While doing so may not yield the most efficient ways of realising (Op1), (Op2) and (Op3), it does not change the complexities of the algorithms we consider.

We remark that the operations (Op) and (Op3) can be realised at equal cost in many Garside groups; this is the case for braid groups, for instance. However, as we are working with a generic Garside group of finite type, we want to keep our assumptions to the minimum. We moreover mention that one could use (Op3) as basic operation instead of (Op): if the cost of (Op3) is $O(C)$, then one can test at a cost

of $O(C\lambda)$ whether a simple element is equal to Δ and the operations (Op) and (Op2) can be realised in terms of (Op3) at a cost of $O(C\lambda \ \| \Delta \|)$; the map ∂ induces a duality between this situation and the situation from Assumption 4.2. Finally, note that (Op1) can be realised in terms of (Op3) at a cost of $O(C\lambda)$, if Δ is the lcm of the atoms of G: in this case, $\partial(s) = \Delta$ is equivalent to $a_i \preccurlyeq \partial(s)$ for all $i = 1, \ldots, \lambda$, that is, $s = 1$ is equivalent to $sa_i \in [1, \Delta]$ for all $i = 1, \ldots, \lambda$.

An important remark concerning the algorithms below is the following: One of the most frequently used operations consists of determining an atom a such that $a \preccurlyeq s$, given a nontrivial simple element s. If the simple elements are stored as products of atoms, this operation has a cost of $O(1)$. However, if the simple elements are stored in a different way, it is possible that the only way to find such an atom is to check whether $a \preccurlyeq s$ for every $a \in \mathcal{A}$, until the answer is positive. This has time complexity $O(C\lambda)$. Therefore, in the algorithms below we will sometimes write 'Take an atom $a \preccurlyeq s$', and we will assume that this operation has a cost of $O(C\lambda)$, although the reader should notice that the actual cost could be only $O(1)$ in some situations.

4.1.3. Algorithms for computing in a generic Garside group

The first computations which we will express in terms of the basic operations are computing left and right complements of simple elements and conjugation of simple elements by Δ or Δ^{-1}. We will also see a generic way of performing the operations (Op2) and (Op3). The following algorithm underlies all of these:

Computing the right complement of a simple element

Input: A simple element s.
Output: The simple element $\partial(s) = s^{-1}\Delta$.

(1) Set $d = \Delta$.
(2) While $s \neq 1$ do:
  (a) Take an atom $a \preccurlyeq s$.
  (b) Set $d = a^{-1}d$ and $s = a^{-1}s$.
(3) Return d.

At most $\| \Delta \|$ passes through the loop are required and the costs of the test $s \neq 1$, step 2(a) and step 2(b) are $O(C\lambda)$, $O(C\lambda)$ and $O(C)$, respectively. Hence, the complexity of this algorithm is $O(C\lambda \ \| \Delta \|)$. Notice that $\partial^{-1}(s) = \Delta\, s^{-1}$ can be computed in the same way, replacing $\preccurlyeq$ by $\succcurlyeq$ and multiplying with a^{-1} on the right instead of on the left. The given algorithm can also be used to compute $\tau(s) = \partial^2(s)$ or $\tau^{-1}(s) = \partial^{-2}(s)$, so all these operations have a cost of $O(C\lambda \, \|\Delta\|)$.

Given a simple element s and an atom a, one can determine whether sa is simple by computing $\partial(s)$ with the above algorithm and checking whether $a \preccurlyeq \partial(s)$, where the latter step has a cost of $O(C)$ by Assumption 4.2. Moreover, if sa is simple, one can compute $sa = \partial^{-1}(a^{-1}\partial(s))$. That is, we can perform operation (Op3) that way. Similarly, one can determine whether as is simple by checking whether $\partial^{-1}(s) \succcurlyeq a$ and, if it is, one can compute $as = \partial(\partial^{-1}(s)a^{-1})$. All these operations have a cost of $O(C\lambda \, \|\Delta\|)$.

Next, we will describe the lattice operations on simple elements, which are important for computing normal forms of elements.

Computing the greatest common divisor of two simple elements

Input: Two simple elements s and t.
Output: The simple element $s \wedge t$.

(1) Set $i = 1$ and $d = \Delta$.
(2) While $i \leq \lambda$ do:
  (a) If $a_i \preccurlyeq s$ and $a_i \preccurlyeq t$, then
  (b) set $d = a_i^{-1}d$, set $s = a_i^{-1}s$, set $t = a_i^{-1}t$ and set $i = 1$,
  else
  (c) set $i = i + 1$.
(3) Return $\partial^{-1}(d)$.

The tests in step 2(a) and the operations in step 2(b) have a cost of $O(C)$, step 3 has a cost of $O(C\lambda\ \|\Delta\|)$, and all remaining operations have a cost of $O(1)$. As step 2(b) is executed at most $\|\Delta\|$ times, with at most λ passes through the while loop between two consecutive executions, the cost of step 2 is $O(C\lambda\ \|\Delta\|)$, so the complexity of the algorithm is also $O(C\lambda\ \|\Delta\|)$. Note that finding the atoms which are common divisors of s and t is critical for the complexity of the algorithm. Thus, even if step 3 was avoided by making use of a realisation of (Op3) with a cost of $O(C)$, the complexity of the algorithm would not improve.

By symmetry, one can similarly compute the greatest common divisor $s \wedge^{\urcorner} t$ with respect to $\succcurlyeq$.

Least common multiples of simple elements with respect to $\preccurlyeq$ or $\succcurlyeq$ can now be computed using the following formulae, which can easily be seen to hold:

$$s \vee t = \partial^{-1}\left(\partial(s) \wedge^{\urcorner} \partial(t)\right), \qquad s \vee^{\urcorner} t = \partial\left(\partial^{-1}(s) \wedge \partial^{-1}(t)\right).$$

Therefore, computing $s \vee t$ or $s \vee^{\urcorner} t$ also takes time $O(C\lambda\ \|\Delta\|)$.

As $s = t$ is equivalent to $s = s \wedge t = t$, we can use the following modification of the above algorithm to test whether two simple elements are equal, that is, perform operation (Op2).

Testing whether two simple elements are equal

Input: Two simple elements s and t.
Output: The truth value of $s = t$.

(1) Set $i = 1$.
(2) While $i \leq \lambda$ do:
  (a) If $a_i \preccurlyeq s$ and $a_i \preccurlyeq t$, then
  (b) set $s = a_i^{-1}s$, set $t = a_i^{-1}t$ and set $i = 1$,
  else
  (c) set $i = i + 1$.
(3) If $s = 1$ and $t = 1$, then return true, else return false.

The cost of step 3 is $O(C\lambda)$; all other steps are as before. Hence, the complexity of the algorithm is $O(C\lambda\ \|\Delta\|)$. This implies, in particular, that two elements of canonical length at most k whose (left or right) normal forms are known, can be compared at a cost of $O(C\lambda k\ \|\Delta\|)$ by comparing their infima (at a cost of $O(1)$) and at most k pairs of simple elements.

The following algorithm computing the local sliding of a pair of simple elements is also just a small modification of the algorithm computing the gcd of two simple elements:

Computing the local sliding of a pair of simple elements

Input: Two simple elements s and t.
Output: The simple elements $s(\partial(s) \wedge t)$ and $(\partial(s) \wedge t)^{-1}t$.

(1) Set $i = 1$ and $s' = \partial(s)$.
(2) While $i \leq \lambda$ do:
  (a) If $a_i \preccurlyeq s'$ and $a_i \preccurlyeq t$, then
  (b) set $s' = a_i^{-1}s'$, set $t = a_i^{-1}t$ and set $i = 1$,
  else
  (c) set $i = i + 1$.
(3) Return $\partial^{-1}(s'),\ t$.

The cost of step 1 is $O(C\lambda\ \|\Delta\|)$; all other steps are identical. Hence, the local sliding of a pair of simple elements can also be computed at a cost of $O(C\lambda\ \|\Delta\|)$.

Knowing how to compute local slidings, one can use the standard algorithms to compute the left or right normal form of any element (see Section 1.1), based on the following well-known result.

Proposition 4.3 (*see, for example, (Charney, 1992, Props. 3.1 and 3.3) or (Epstein et al., 1992)*). *Let* $s_1, \ldots, s_k$ *and* s'_0, s'_{k+1} *be simple elements such that the product* $s_1 \cdots s_k$ *is in left normal form as written.*

(1) *Consider the product* $s'_0 s_1 \cdots s_k$. *For* $i = 1, \ldots, k$ *apply a local sliding to the pair* $s'_{i-1}s_i$, *that is, let* $t_i = \partial(s'_{i-1}) \wedge s_i$ *and define* $s''_{i-1} = s'_{i-1}t_i$ *and* $s'_i = t_i^{-1}s_i$. *Finally define* $s''_k = s'_k$. *Then,* $s''_0 \cdots s''_k$ *is the left normal form of* $s'_0 s_1 \cdots s_k$ *(where possibly* $s''_0 = \Delta$ *or* $s''_k = 1$*).*

(2) *Consider the product* $s_1 \cdots s_k s'_{k+1}$. *For* $i = k, \ldots, 1$ *apply a local sliding to the pair* $s_i s'_{i+1}$, *that is, let* $t_i = \partial(s_i) \wedge s'_{i+1}$ *and define* $s'_i = s_i t_i$ *and* $s''_{i+1} = t_i^{-1}s'_{i+1}$. *Finally define* $s''_1 = s'_1$. *Then,* $s''_1 \cdots s''_{k+1}$ *is the left normal form of* $s_1 \cdots s_k s'_{k+1}$ *(where possibly* $s''_1 = \Delta$ *or* $s''_{k+1} = 1$*).*

Given an element x written as a product of k simple elements or inverses of simple elements, the left normal form of x can be obtained as follows. First, one replaces each inverse s^{-1} of a simple element with $\Delta^{-1}\partial^{-1}(s)$; at most k replacements are necessary and each replacement has a cost of $O(C\lambda \ \|\Delta\|)$. Then, one collects all appearances of Δ or Δ^{-1} on the left hand side, applying τ or τ^{-1} as required, so that the element will be written as $\Delta^q s_1 \cdots s_k$, where each s_i is a simple element; the number of applications of τ or τ^{-1} is bounded by $k(k-1)/2$ and each application has a cost of $O(C\lambda \ \|\Delta\|)$. Finally, one applies local slidings to every pair of consecutive simple elements until every pair is left weighted; it follows from Proposition 4.3 that at most $k(k-1)/2$ local slidings are required, each at a cost of $O(C\lambda \ \|\Delta\|)$. Therefore, the complexity of computing the left normal form of x is $O(C\lambda k^2 \ \|\Delta\|)$. Computing right normal forms is analogous and has the same complexity.

Note, however, that if the left normal form (resp. the right normal form) of x is known and s is a simple element, then the left normal forms (resp. the right normal forms) of xs, sx, xs^{-1}, $s^{-1}x$, x^s and $x^{s^{-1}}$ can be computed at a cost of $O(C\lambda k \ \|\Delta\|)$, where $k = \ell(x)$: the number of applications of τ or τ^{-1} is bounded by k and only $O(k)$ local slidings are required by Proposition 4.3.

We now show how to compute the gcd of two arbitrary elements a and b, given as products of simple elements and inverses of simple elements with at most k factors. First, we write them in left normal form, say $\Delta^p a_1 \cdots a_r$ and $\Delta^q b_1 \cdots b_t$. If we denote $m = \min\{p, q\}$, we can consider $a' = \Delta^{-m}a$ and $b' = \Delta^{-m}b$. Notice that a' and b' are positive elements, and one of them has infimum zero. Since $a \wedge b = \Delta^m a' \wedge \Delta^m b' = \Delta^m(a' \wedge b')$, it is sufficient to know how to compute gcds of positive elements and we will hence detail the algorithm to compute $a \wedge b$ assuming a and b are positive; the cost of reducing to this case by computing the normal forms of a and b is $O(C\lambda k^2 \ \|\Delta\|)$. We remark that, if the left normal form of a positive element a is known, then $a \wedge \Delta$ is also known, since it is precisely the first factor in its left normal form (which may be Δ).

Computing the greatest common divisor of two positive elements

Input: Two positive elements a and b.
Output: The element $a \wedge b$.

(1) Set $u = \Delta$, $a' = a$, $b' = b$ and $d = 1$.
(2) While $u \neq 1$ do:
  (a) Compute the left normal forms of a' and b'.
  (b) Set $s = a' \wedge \Delta$ and $t = b' \wedge \Delta$.
  (c) Set $u = s \wedge t$.
  (d) Set $d = du$, set $a' = u^{-1}a'$ and $b' = u^{-1}b'$.
(3) Return d.

Since a and b are positive, one has $(a \wedge b) \wedge 1 = 1$. It is then easy to see by induction that after the ith pass through the while loop one has $d = (a \wedge b) \wedge \Delta^i$. Hence, if a and b are given as products of simple elements and inverses of simple elements with at most k factors, the number of repetitions of the while loop is bounded by $k + 1$. The cost of step 2(a) in the first pass through the while loop is $O(C\lambda k^2 \ \|\Delta\|)$, but in all subsequent passes, the cost is $O(C\lambda k \ \|\Delta\|)$ by Proposition 4.3. As the costs of steps 2(b), 2(c) and 2(d) are $O(1)$, $O(C\lambda \ \|\Delta\|)$ and $O(k)$, respectively, the complexity of the algorithm hence is $O(C\lambda k^2 \ \|\Delta\|)$. Computing the right gcd $a \wedge^{\neg} b$ is analogous and has the same complexity.

One can now compute the least common multiple of two elements a and b, given as products of simple elements and inverses of simple elements with at most k factors, as follows. Compute the normal forms of a and b and let $m = \max\{\sup(a), \sup(b)\}$. The elements $a^{-1}\Delta^m$ and $b^{-1}\Delta^m$ are both positive, whence we can compute the element $d = (a^{-1}\Delta^m) \wedge^{\neg} (b^{-1}\Delta^m)$ using (the right version of)

the algorithm above. Then, $a \vee b = (a^{-1} \wedge^{\urcorner} b^{-1})^{-1} = \Delta^m((a^{-1}\Delta^m) \wedge^{\urcorner} (b^{-1}\Delta^m))^{-1} = \Delta^m d^{-1}$. The cost of this computation is dominated by computing d as the right gcd of $a^{-1}\Delta^m$ and $b^{-1}\Delta^m$ which has cost $O(C\lambda k^2 \|\Delta\|)$. Thus, the complexity of computing the lcm $a \vee b$ is $O(C\lambda k^2 \|\Delta\|)$. Computing the right lcm $a \vee^{\urcorner} b$ is analogous and has the same complexity.

The computations of the preferred prefix and the cyclic sliding of an element can now be done just by applying the definitions, since we already know how to perform all operations that occur. For instance, in order to compute the preferred prefix of an element x, given as a product of simple elements and inverses of simple elements with k factors, one first computes the left normal form of $x = \Delta^p x_1 \cdots x_r$, which takes time $O(C\lambda k^2 \|\Delta\|)$. Then one applies the formula given in Definition 1.11, namely $\mathfrak{p}(x) = \iota(x) \wedge \partial(\varphi(x))$. Since $\iota(x) = \tau^{-p}(x_1)$ with $|p| \leq k$ and $\varphi(x) = x_r$, the complexity of computing $\mathfrak{p}(x)$ from the normal form of x is $O(C\lambda k \|\Delta\|)$. The normal form of $\mathfrak{s}(x) = x^{p(x)}$ can then be computed in $O(C\lambda k \|\Delta\|)$. Thus, the cost of applying a cyclic sliding is dominated by the cost of computing the normal form, that is, applying a cyclic sliding has complexity $O(C\lambda k^2 \|\Delta\|)$. Note that if the normal form of x is known, then $\mathfrak{p}(x)$ and the normal form of $\mathfrak{s}(x)$ can be obtained at a cost of $O(C\lambda k \|\Delta\|)$.

The transport of an element α at an element x is given by $\alpha^{(1)} = \mathfrak{p}(x)^{-1}\alpha\mathfrak{p}(x^\alpha)$. If x and α are given as products of simple elements and inverses of simple elements with at most k factors, then $\alpha^{(1)}$ can be computed with the above formula in time $O(C\lambda k^2 \|\Delta\|)$ by the arguments from the previous paragraph. In other words, applying a transport has the same complexity as computing a normal form. Note that if the normal form of x is known and α is simple, then the normal form of x^α can be obtained at a cost of $O(C\lambda k \|\Delta\|)$, whence $\alpha^{(1)}$ can be computed at a cost of $O(C\lambda k \|\Delta\|)$ by the arguments above.

Computing the preferred suffix, applying a cyclic right sliding and applying right transport are analogous and the complexities are the same as for the left versions discussed above.

Finally, the pullback of a positive element s at an element y, with the hypotheses and the notation of Proposition 3.19, is $s_{(1)} = \left(\mathfrak{p}(z)\, s\, \mathfrak{p}^{\urcorner}(y^s)^{-1}\right) \vee 1$; we assume that we also know the element z. If y, z and s are given as products of simple elements and inverses of simple elements with at most k factors, then $s_{(1)}$ can be computed in time $O(C\lambda k^2 \|\Delta\|)$ using the operations described above. If s is simple and if the left normal form of z and the right normal form of y are known, then $\mathfrak{p}(z)\, s\, \mathfrak{p}^{\urcorner}(y^s)^{-1}$ can be computed at a cost of $O(C\lambda k \|\Delta\|)$ and, since this product involves only 3 simple factors, the subsequent computation of the lcm has a cost of $O(C\lambda \|\Delta\|)$, whence in this case $s_{(1)}$ can be obtained at a cost of $O(C\lambda k \|\Delta\|)$. Computing the right pullback $s_{(1)^{\urcorner}}$ is analogous and has the same complexity.

Summarising the results obtained in this section, we have:

Theorem 4.4. *Let G be a Garside group of finite type with Garside element Δ and set of atoms $\mathcal{A} = \{a_1, \ldots, a_\lambda\}$ for which Assumption 4.2 is satisfied. Moreover, let a be an atom of G, let s and t be simple elements of G and let x, y and α be elements of G, given as products of simple elements or inverses of simple elements with at most k factors.*

(1) *The following operation can be performed in $O(C\lambda)$:*
- *Test whether $s = 1$.*

(2) *The following operations can be performed in $O(C\lambda \|\Delta\|)$:*
- *Test whether $s = t$.*
- *Compute $\partial(s)$, $\partial^{-1}(s)$, $\tau(s)$ or $\tau^{-1}(s)$.*
- *Test whether the product as is simple and, if so, compute as.*
- *Test whether the product sa is simple and, if so, compute sa.*
- *Compute $s \wedge t$, $s \wedge^{\urcorner} t$, $s \vee t$ or $s \vee^{\urcorner} t$.*
- *Perform a local (left or right) sliding on the product $s \cdot t$.*

(3) *The following operations can be performed in $O(C\lambda k \|\Delta\|)$:*
- *Test whether $x = y$, if the left normal forms or the right normal forms of x and y are known.*
- *Compute the left normal form [resp. the right normal form] of xs, sx, xs^{-1}, $s^{-1}x$, x^s or $x^{s^{-1}}$, if the left normal form [resp. the right normal form] of x is known.*
- *Compute $\mathfrak{p}(x)$ or $\mathfrak{s}(x)$ [resp. $\mathfrak{p}^{\urcorner}(x)$ or $\mathfrak{s}^{\urcorner}(x)$], if the left normal form [resp. the right normal form] of x is known.*

- *Compute the left transport* $s^{(1)}$ [*resp. the right transport* $s^{(1)^{\urcorner}}$] *of* s *at* x*, if the left normal form* [*resp. the right normal form*] *of* x *is known.*
- *Compute the left pullback* $s_{(1)}$ [*resp. the right pullback* $s_{(1)^{\urcorner}}$] *of* s *at* x*, if it is defined and if the right normal form* [*resp. the left normal form*] *of* x *and the left normal form of the element* $z \in \mathrm{SC}(x)$ *satisfying* $\mathfrak{s}(z) = x$ [*resp. the right normal form of the element* $z \in \mathrm{SC}^{\urcorner}(x)$ *satisfying* $\mathfrak{s}^{\urcorner}(z) = x$] *are known.*

(4) *The following operations can be performed in* $O(C\lambda k^2\ \|\Delta\|)$*:*

- *Compute the left normal form of* x *or the right normal form of* x*.*
- *Compute* $x \wedge y$, $x \wedge^{\urcorner} y$, $x \vee y$ *or* $x \vee^{\urcorner} y$*.*
- *Compute* $\mathfrak{p}(x)$, $\mathfrak{p}^{\urcorner}(x)$, $\mathfrak{s}(x)$ *or* $\mathfrak{s}^{\urcorner}(x)$*.*
- *Compute the left transport* $\alpha^{(1)}$ *of* α *at* x *or the right transport* $\alpha^{(1)^{\urcorner}}$ *of* α *at* x*.*
- *Compute the left pullback* $\alpha_{(1)}$ [*resp. the right pullback* $\alpha_{(1)^{\urcorner}}$] *of* α *at* x*, if it is defined and if the element* $z \in \mathrm{SC}(x)$ *satisfying* $\mathfrak{s}(z) = x$ [*resp. the element* $z \in \mathrm{SC}^{\urcorner}(x)$ *satisfying* $\mathfrak{s}^{\urcorner}(z) = x$] *is known.*

4.2. Complexity of the new algorithms

Knowing the computational cost of the basic operations, we can now analyse the complexity of the algorithms for computing $\mathrm{SC}(x)$ from Section 1.3. Firstly, we define some bounds which will be used in the sequel.

Notation 4.5. Let x be an element of G given as a product of simple elements or inverses of simple elements with at most k factors.

[Distance to cyclic sliding repetition] Let T be an integer such that there exist two integers $0 \leq i < j \leq T$ satisfying $\mathfrak{s}^i(x) = \mathfrak{s}^j(x)$.

[Length of sliding circuits] Let M be an integer such that for any element $z \in \mathrm{SC}(x)$ there exists a positive integer $N \leq M$ with $\mathfrak{s}^N(z) = z$.

[Distance to transport repetition] Let R be an integer such that for any element $z \in \mathrm{SC}(x)$ and any simple element s satisfying $z^s \in \mathrm{SSS}(x)$ there exist two integers $0 \leq i < j \leq R$ satisfying $s^{(iN)} = s^{(jN)}$, where $\mathfrak{s}^N(z) = z$ and $s^{(m)}$ denotes m-fold transport at z for $m \in \mathbb{N}$.

Remark 4.6. It is easy to see that integers T, M and R as above exist and to give some obvious (but very crude) upper bounds for them: By Corollary 2.2, iterated cyclic sliding becomes periodic, so T as above exists. Indeed, it follows from Proposition 2.3 that $\mathfrak{s}^m(x) \in \mathrm{SSS}(x)$ for all $m \geq k\ \|\Delta\|$. Since $|\mathrm{SSS}(x)| \leq |[1, \Delta]|^k$, it is possible to choose $T \leq k\ \|\Delta\| + |[1, \Delta]|^k$. Moreover, as $\mathrm{SC}(x) \subseteq \mathrm{SSS}(x)$ is finite, M as above exists and one can choose $M \leq |\mathrm{SC}(x)|$. Hence, in particular, $M \leq |[1, \Delta]|^k$. Finally, by Proposition 2.4 (5), transports of simple elements are simple. Since G is of finite type, R as above exists and one can choose $R \leq |[1, \Delta]|$.

Lemma 4.7. *Let* $x \in G$, $z \in \mathrm{SC}(x)$*, and let* s *be a simple element such that* $z^s \in \mathrm{SSS}(x)$*. If* N*,* i*,* j *and* K *are integers such that* $\mathfrak{s}^N(z) = z$*,* $0 \leq i < j \leq K$ *and* $\left(s_{(KN)}\right)^{(iN)} = \left(s_{(KN)}\right)^{(jN)}$*, where* $t^{(m)}$ *denotes* m*-fold transport of* t *at* z *and* $t_{(m)}$ *denotes* m*-fold pullback of* t *at* z *for* $m \in \mathbb{N}$*, then* $s_{(KN)} = s_{((K+j-i)N)}$*.*

Proof. By Lemma 3.15 we have $s_{(KN-iN)} \preccurlyeq \left(\left(s_{(KN-iN)}\right)_{(iN)}\right)^{(iN)} = \left(s_{(KN)}\right)^{(iN)} = \left(s_{(KN)}\right)^{(jN)}$. Again using Lemma 3.15, we obtain $\left(s_{(KN-iN)}\right)_{(jN)} \preccurlyeq s_{(KN)}$, that is, $s_{((K+j-i)N)} \preccurlyeq s_{(KN)}$.

Similarly, we have $s_{(KN-jN)} \preccurlyeq \left(\left(s_{(KN-jN)}\right)_{(jN)}\right)^{(jN)} = \left(s_{(KN)}\right)^{(jN)} = \left(s_{(KN)}\right)^{(iN)}$ and from this obtain $\left(s_{(KN-jN)}\right)_{(iN)} \preccurlyeq s_{(KN)}$, that is, $s_{((K+i-j)N)} \preccurlyeq s_{(KN)}$. Applying $(j-i)N$-fold pullback to the last statement yields $s_{(KN)} \preccurlyeq s_{((K+j-i)N)}$ using Lemma 3.16.

Hence, $s_{(KN)} = s_{((K+j-i)N)}$ as we wanted to show. □

Corollary 4.8. *Consider for $x \in G$ the bounds from Notation 4.5. For any element $z \in \mathrm{SC}(x)$ and any simple element s satisfying $z^s \in \mathrm{SSS}(x)$ there exist two integers $0 \leq i < j \leq 2R$ satisfying $s_{(iN)} = s_{(jN)}$, where $\mathfrak{s}^N(z) = z$ and $s_{(m)}$ denotes m-fold pullback of s at z for $m \in \mathbb{N}$.*

Proof. By the choice of R there are integers $0 \leq i' < j' \leq R$ such that $\left(s_{(RN)}\right)^{(i'N)} = \left(s_{(RN)}\right)^{(j'N)}$. We then have $s_{(RN)} = s_{((R+j'-i')N)}$ by Lemma 4.7. Setting $i = R$ and $j = R + j' - i'$, we have $0 \leq i < j \leq 2R$ and $s_{(iN)} = s_{(jN)}$ as desired. □

Proposition 4.9. *Let G be a Garside group of finite type with Garside element Δ and λ atoms, and let x be an element of G given as a product of simple elements or inverses of simple elements with at most k factors. Using the bounds from Notation 4.5, the complexity of Algorithm 1 is $O(C\lambda k(k+T)\ \|\Delta\|)$.*

Proof. Observe that $\ell(\mathfrak{s}^i(x)) \leq k$ for all non-negative integers i. In particular, the normal forms of two such elements can be compared at a cost of $O(C\lambda k\ \|\Delta\|)$ by Theorem 4.4. Note further that a hash function depending on all factors in the normal form can be computed at a cost of $O(Ck)$, if the normal form is known. We use a sufficiently large hash table, together with this hash function, to store the trajectory $\mathcal{T}$ in step 2. If the normal form of an element y with $\ell(y) \leq k$ is known, testing whether $y \in \mathcal{T}$ (and storing it if it is not) then has a cost of $O(C\lambda k\ \|\Delta\|)$.

We initially compute the normal form of x at a cost of $O(C\lambda k^2\ \|\Delta\|)$. Step 1 has a cost of $O(1)$. Step 3 and each pass through the while loops in step 2 and step 4 have a cost of $O(C\lambda k\ \|\Delta\|)$ by Theorem 4.4. The number of passes through the while loops is bounded by T. Step 5 has a cost of $O(T)$. Hence the claim holds. □

Proposition 4.10. *Let G be a Garside group of finite type with Garside element Δ and λ atoms, let $x \in G$, and let $v \in \mathrm{SC}(x)$ be given as a product of simple elements or inverses of simple elements with at most k factors. If the left and right normal forms of v are known, then, using the bounds from Notation 4.5, the complexity of Algorithm 2 is $O\big(C\lambda^2 k\ \|\Delta\|\ (\|\Delta\| + RM)\big)$.*

Proof. In step 1, we perform $N \leq M$ times the following operations: apply a cyclic sliding to an element whose left and right normal forms are known, compute the left normal form and the right normal form of the result and compare it to v; each of these has a cost of $O(C\lambda k\ \|\Delta\|)$ by Theorem 4.4. Hence, the cost of step 1 is $O(C\lambda kM\ \|\Delta\|)$.

Step 2 has a cost of $O(\lambda)$; we store the set $\mathcal{A}_v$ as a list and the set $Atoms \subseteq \{a_1, \ldots, a_\lambda\}$ as an array of λ flags.

Steps 3(a) to (e) are executed λ times. Step 3(a) has a cost of $O(1)$; the costs of the remaining steps are as follows:

For step 3(b) note that at any time we have $a_t \preccurlyeq s \preccurlyeq \rho_{a_t} \preccurlyeq \Delta$, so s is simple. In particular, $\sup(v^s) - \sup(v) \in \{0, 1\}$. As the right normal form of v is known and s is simple, the right normal form of v^s can be computed at a cost of $O(C\lambda k\ \|\Delta\|)$ by Theorem 4.4. By Proposition 3.6, we can obtain the element $1 \vee v^s\Delta^{-\sup(v)}$ from the right normal form of v^s at a cost of $O(1)$: it is the leftmost factor in the right normal form if $\sup(v^s) = \sup(v) + 1$, and it is trivial if $\sup(v^s) = \sup(v)$. In the same way, we can obtain $1 \vee (v^s)^{-1}\Delta^{\inf(v)}$ from the right normal form of $(v^s)^{-1}$. Observe that the right normal form of $(v^s)^{-1}$ is related to the right normal form of v^s: the leftmost factor in the right normal form of $(v^s)^{-1}$ can be obtained from the rightmost non-Δ factor in the right normal form of v^s by applying the map ∂ or ∂^{-1} at most $2k+1$ times, that is, at a cost of $O(C\lambda k\ \|\Delta\|)$ by Theorem 4.4. As both $1 \vee v^s\Delta^{-\sup(v)}$ and $1 \vee (v^s)^{-1}\Delta^{\inf(v)}$ are simple, so is their lcm. In particular, computing the lcm and the final multiplication (which is a local sliding) each have a cost of $O(C\lambda\ \|\Delta\|)$ by Theorem 4.4. Hence, since the number of passes through the while loop is at most $\|\Delta\|$ by Proposition 3.4, step 3(b) has a cost of $O(C\lambda k\ \|\Delta\|^2)$.

In step 3(c) the initial test $a_t \preccurlyeq \mathfrak{p}(v)$ has a cost of $O(C)$. We can store the simple elements $s_{(iN)}$ $(i = 1, 2 \ldots)$ in a sufficiently large hash table, using the hash function from Assumption 4.2. Testing whether $s_{(iN)}$ has already occurred (and storing it if not) then has a cost of $O(C\lambda\ \|\Delta\|)$. Since the left and right normal forms of all elements in the sliding circuit of v are known from step 1, each pullback can be computed at a cost of $O(C\lambda k\ \|\Delta\|)$ by Theorem 4.4. As the number of pullbacks which need to be computed is bounded by $2RM$ by Corollary 4.8, the cost of step 3(c) hence is $O(C\lambda kRM\ \|\Delta\|)$.

By the same arguments, step 3(d) has a cost of $O(C\lambda kRM \ \| \Delta \|)$, since each transport can be computed at a cost of $O(C\lambda k \ \| \Delta \|)$ and the number of transports which need to be computed is bounded by RM.

The test in the outer if statement in step 3(e) has a cost of $O(CR)$, whereas the test in the if statement in step 3(e)i has a cost of $O(C\lambda)$, since testing whether $a_k \in Atoms$ has a cost of $O(1)$. As the remaining operations in step 3(e)i have a cost of $O(1)$, the cost of step 3(e) is $O(C(\lambda + R))$.

Hence, the complexity of Algorithm 2 is $O(C\lambda^2 k \ \|\Delta\| \ (\|\Delta\| + RM))$ as claimed. □

Theorem 4.11. *Let G be a Garside group of finite type with Garside element Δ and λ atoms, and let x and y be elements of G given as products of simple elements or inverses of simple elements with at most k factors. Let T, M and R be the maxima of the bounds from Notation 4.5 for x and y, respectively.*

The complexity of Algorithm 3 is $O\Big(C\lambda k \ \|\Delta\| \cdot \big(k + T + |\mathrm{SC}(x)|\lambda(\|\Delta\| + RM)\big)\Big)$.

Proof. Observe that $\ell(z) \leq k$ for all $z \in \mathrm{SC}(x)$. In particular, the (left) normal forms of two such elements can be compared at a cost of $O(C\lambda k \ \|\Delta\|)$ by Theorem 4.4. Note further that a hash function depending on all factors in the normal form can be computed at a cost of $O(Ck)$, if the normal form is known. We use a sufficiently large hash table, together with this hash function, to store the set $\mathcal{V}$. More precisely, whenever a new element $v^s \in \mathrm{SC}(x)$ is found, where s is an indecomposable conjugator and $v \in \mathcal{V}$, we store the following information in the hash table entry for v^s: the left normal form and the right normal form of v^s, the indecomposable conjugator s, and the position of v in the hash table. If the left normal form and the right normal form of v^s are known, testing whether $v^s \in \mathcal{V}$, and storing all required data if it is not, has a cost of $O(C\lambda k \ \|\Delta\|)$. The set $\mathcal{V}'$ is stored as a list (storing hash table indices instead of actual elements), whence storing or retrieving an element of $\mathcal{V}'$ has a cost of $O(1)$. Observe that the conjugating elements c_v for $v \in \mathcal{V}$ are implicit in the spanning tree structure for $\mathrm{SCG}(x)$ with root $\widetilde{x}$ which is computed: for any $v \in \mathcal{V}$, the conjugating element c_v can be obtained by tracing back the path to the root which is given by the indecomposable conjugators stored for every entry in the hash table. This trace-back has a cost of $O(|\mathrm{SC}(x)|)$, since the length of the path to the root is bounded by $|\mathrm{SC}(x)|$ and each step of the trace-back has a cost of $O(1)$. In particular, there is no actual computation of $c_{v^s} = c_v \cdot s$ in step 3(c)ii; at most $c_{\widetilde{y}}$ is ever explicitly computed (in step 3(c)i).

Step 1 has a cost of $O(C\lambda k(k+T) \ \|\Delta\|)$ by Proposition 4.9; this includes computing the left and right normal forms of $\widetilde{x}$ and $\widetilde{y}$. For step 3(c) note that, since the left normal form and the right normal form of v are known, the left normal form and the right normal form of each conjugate v^s can be computed at a cost of $O(C\lambda k \ \|\Delta\|)$ by Theorem 4.4. Steps 3(a), 3(d) and 4 have a cost of $O(1)$. Steps 2, 3(c)ii, as well as the test of the condition in step 3(c)i have a cost of $O(C\lambda k \ \|\Delta\|)$. By Proposition 4.10, Step 3(b) has a cost of $O(C\lambda^2 k \ \|\Delta\| \ (\|\Delta\| + RM))$. The body of the while loop in step 3 is executed $|\mathrm{SC}(x)|$ times and the body of the for loop in step 3(c) is executed at most λ times. The actual computation of the conjugating element $c_1 \cdot c_{\widetilde{y}} \cdot c_2^{-1}$ in step 3(c)i has a cost of $O(T + |\mathrm{SC}(x)|)$, but is executed at most once.

Thus, the complexity of Algorithm 3 is

$$\begin{aligned} &O\Big(C\lambda k(k+T) \ \|\Delta\| \ \Big) + O\Big(|\mathrm{SC}(x)| \cdot C\lambda^2 k \ \|\Delta\| \ (\|\Delta\| + RM)\Big) \\ &\quad + O\Big(|\mathrm{SC}(x)|\lambda \cdot C\lambda k \ \|\Delta\| \ \Big) + O\Big(T + |\mathrm{SC}(x)|\Big) \\ &= O\Big(C\lambda k \ \|\Delta\| \cdot \big(k + T + |\mathrm{SC}(x)|\lambda(\|\Delta\| + RM)\big)\Big) \end{aligned}$$

as claimed. □

Remark 4.12. Unfortunately, the obvious bounds for T and M given in Remark 4.6 are exponential in k. For the Artin braid groups B_n one has $|[1, \Delta]| = n!$, that is, the above bounds are also exponential in n (or $\|\Delta\|$) for this sequence of Garside groups, as is the bound for R given in Remark 4.6. Moreover, no bound for $|\mathrm{SC}(x)|$ is currently known which is better than the obvious bound $|\mathrm{SC}(x)| \leq |\mathrm{SSS}(x)| \leq |[1, \Delta]|^k$ (cf. Remark 4.6); the latter again is exponential. None of these bounds adequately describes the behaviour observed in computer experiments.

We conjecture that there are bounds for T, M and R which are polynomial in k and $\| \Delta \|$. If the elements of $\mathrm{SC}(x)$ are rigid, then one can choose $R = \| \Delta \|$ by (Gebhardt and González-Meneses, in

press, Proposition 8 and Corollary 11), and obviously $M = 1$. However, even in this case, no realistic bound for T is known.

The situation for $|\mathrm{SC}(x)|$ is more complicated. It is shown in Birman et al. (2007b) that $|\mathrm{USS}(x)|$ grows exponentially in n for periodic elements of the Artin braid groups B_n. By (Gebhardt and González-Meneses, in press, Proposition 9), the same is true for $|\mathrm{SC}(x)|$. Hence, a bound for $|\mathrm{SC}(x)|$ which is polynomial in k and $\| \Delta \|$ cannot be expected in general. However, it may be possible to establish such a bound for certain classes of elements, for example rigid elements.[2] For the situation of Artin braid groups, an attempt to reduce the general case to the special case of rigid elements is sketched in Birman et al. (2007a).

The problem of finding bounds for T, M and R which are polynomial in k and $\|\Delta\|$ and the problem of understanding $|\mathrm{SC}(x)|$ correspond to open problems formulated in Birman et al. (2007a) in the context of Artin braid groups for ultra summit sets and the cycling and decycling operations.

Acknowledgements

Both authors were partially supported by MTM2007-66929 and FEDER. This work was done partially while the second author was visiting the Institute for Mathematical Sciences, National University of Singapore in 2007. The visit was supported by the Institute.

References

Birman, J.S., Gebhardt, V., González-Meneses, J., 2007a. Conjugacy in Garside groups I: Cycling, powers and rigidity. Groups Geom. Dyn. 1, 221–279.
Birman, J.S., Gebhardt, V., González-Meneses, J., 2007b. Conjugacy in Garside groups III: Periodic braids. J. Algebra 316, 746–776.
Birman, J.S., Ko, K.Y., Lee, S.J., 1998. A new approach to the word and conjugacy problems in the braid groups. Adv. Math. 139, 322–353.
Charney, R., 1992. Artin groups of finite type are biautomatic. Math. Ann. 292, 671–683.
Dehornoy, P., Paris, L., 1999. Gaussian groups and Garside groups, two generalizations of Artin groups. Proc. Lond. Math. Soc. (3) 79, 569–604.
Dehornoy, P., 2002. Groupes de Garside. Ann. Sci. Ec. Norm. Super. (4) 35, 267–306.
ElRifai, E., Morton, H., 1994. Algorithms for positive braids. Q. J. Math. Oxf. Ser. (2) 45, 479–497.
Epstein, D.B.A., Cannon, J.W., Holt, D.F., Levy, S.V.F., Paterson, M.S., Thurston, W.P., 1992. Word Processing in Groups. Jones and Bartlett Publishers, Boston.
Franco, N., González-Meneses, J., 2003. Conjugacy problem for braid groups and Garside groups. J. Algebra 266, 112–132.
Gebhardt, V., 2005. A new approach to the conjugacy problem in Garside groups. J. Algebra 292, 282–302.
Gebhardt, V, González-Meneses, J., 2009. The cyclic sliding operation in Garside groups. Math. Z., in press (doi:10.1007/s00209-009-0502-2).
Knuth, D.E., 1998. The Art of Computer Programming, Volume 3: Sorting and Searching. Addison-Wesley, Reading, Massachusetts.
Lee, E.-K., Lee, S.J., 2008. Abelian subgroups of Garside groups. Comm. Algebra 36, 1121–1139.
Michel, J., 1997. Garside and braid monoids and groups. In: The GAP Manual. (Chapter 82). Available at http://www.math.jussieu.fr/~jmichel/htm/CHAP082.htm.

[2] After this paper was accepted, Prasolov [arXiv:0906.0076v1 [math.GT]] found a family of pseudo-Anosov, rigid braids whose sets of sliding circuits have exponential size with respect to the number of strands.