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 to established algorithms is the use of an operation called cyclic sliding, which is a special kind of conjugation introduced in [11]. Cyclic sliding assumes the role played by cycling and decycling in previous algorithms.

Cyclic sliding will be motivated and explained in §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 $\preceq$, and there is a special element $\Delta$, called Garside element. Given $x \in G$, $\inf(x)$ and $\sup(x)$ are the maximal and minimal integers, respectively, satisfying $\Delta^\inf(x) \preceq x \preceq \Delta^\sup(x)$. Secondly, given $a, b \in G$, there is a unique greatest common divisor $a \land b$ with respect to $\preceq$. Finally, the elements in the set $[1, \Delta] = \{ s \in G | 1 \preceq s \preceq \Delta \}$, 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.

*Both authors 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.
Using the above well known notions, we can define the following:

**Definition 1.1 ([11]).** Given $x \in G$, we define the **preferred prefix** $p(x)$ of $x$ as the simple element

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

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

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

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 §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 $\bar{x} = x$, $c_1 = 1$ and $\mathcal{T} = \emptyset$.
2. While $\bar{x} \notin \mathcal{T}$, set $\mathcal{T} = \mathcal{T} \cup \{\bar{x}\}$, $c_1 = c_1 \cdot p(\bar{x})$ and $\bar{x} = s(\bar{x})$.
3. Set $\bar{y} = y$, $c_2 = 1$ and $\mathcal{T} = \emptyset$.
4. While $\bar{y} \notin \mathcal{T}$, set $\mathcal{T} = \mathcal{T} \cup \{\bar{y}\}$, $c_2 = c_2 \cdot p(\bar{y})$ and $\bar{y} = s(\bar{y})$.
5. Set $\mathcal{V} = \{\bar{x}\}$, $\mathcal{V}' = \{\bar{x}\}$ and $c_2 = 1$.
6. While $\mathcal{V}' \neq \emptyset$, do:
   (a) Take $v \in \mathcal{V}'$.
   (b) For every simple element $s$, do:
      i. If $v^s = \bar{y}$, then set $c_{\bar{y}} = c_{v} \cdot s$. Return ‘$x$ and $y$ are conjugate by $c_1 \cdot c_{\bar{y}} \cdot c_2^{-1}’. STOP.
      ii. If $v^s \notin \mathcal{V}$, then:
         A. Apply iterated cyclic sliding to $v^s$ until the first repetition, 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 $\text{SC}(x)$, was introduced in [11]. 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 $\text{SC}(x)$ consists of those conjugates of $x$ which are stabilised by $s^k$ for some positive integer $k$ and it is analogous to the ultra summit set $\text{USS}(x)$ from [10]. One has $\text{SC}(x) \subseteq \text{USS}(x)$, and in general $\text{SC}(x)$ is a proper subset of $\text{USS}(x)$. 

2
The first two lines of the algorithm compute an element \( \tilde{x} \in SC(x) \), by applying iterated cyclic sliding until the first repetition is reached (which is \( \tilde{x} \)). A conjugating element \( c_1 \) from \( x \) to \( \tilde{x} \) is also computed. The following two lines compute \( \tilde{y} \in SC(y) \) and a conjugating element \( c_2 \) from \( y \) to \( \tilde{y} \) in the same way. Then, the algorithm starts to compute the whole set \( SC(x) \). If during the computation it finds \( \tilde{y} \) as an element of \( SC(x) \), 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 \( SC(x) \) without finding \( \tilde{y} \) 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 [11] 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 §1.1, we give a basic introduction to the theory of Garside groups; specialists may skip this part. In §1.2 we briefly explain the new concepts from [11] which are subsequently used for the detailed description of the algorithm in §1.3.

The rest of the paper is devoted to the explanation and analysis of the algorithm. §2 contains a summary of results from [11] which are required in our discussion. In §3 the algorithm is explained and shown to be correct. Finally, the complexity of the new algorithm is analysed in §4, where §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 [5]. For a detailed introduction to these groups, see [6]; a shorter introduction, containing all the details needed for this paper can be found in [1] (§1.1 and the beginning of §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** \((G, P, \Delta)\) 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 \( \preceq \) defined on \( G \) by \( a \preceq b \iff a^{-1}b \in P \) (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 \( \preceq \).

(G2) The set \([1, \Delta] = \{a \in G \mid 1 \preceq a \preceq \Delta\}\), called the set of **simple elements**, generates \( G \).

(G3) Conjugation by \( \Delta \) preserves \( P \) (so it preserves the lattice order \( \preceq \)). That is, \( \Delta^{-1}P\Delta = P \).

(G4) For all \( x \in P\setminus\{1\} \), one has:

\[
||x|| = \sup\{k \mid \exists a_1, \ldots, a_k \in P\setminus\{1\} \text{ such that } x = a_1 \cdots a_k < \infty.\]
Definition 1.2. A Garside structure \((G, P, \Delta)\) 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 \((G, P, \Delta)\) of finite type.

Remarks:

1. By definition, \(p \in P \iff 1 \leq p\). Given two positive elements \(a \preceq b\), one usually says that \(a\) is a prefix of \(b\). Hence the simple elements are the positive prefixes of \(\Delta\).

2. The number \(||x||\) defined above for each \(x \in P\setminus\{1\}\), defines a norm in \(P\) (setting \(||1|| = 0\)).

Note that the existence of this norm implies that every element in \(P\setminus\{1\}\) 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 \(P\setminus\{1\}\), all of them are atoms. Notice that the set of atoms generates \(G\). Moreover, the set of atoms is finite if \(G\) is of finite type.

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 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 \(\Delta\) is the positive braid in which any two strands cross exactly once (also called the half twist). That is, \(\Delta = \sigma_1(\sigma_2\sigma_1)(\sigma_3\sigma_2\sigma_1)\cdots(\sigma_{n-1}\cdots\sigma_1)\).

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

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

**Definition 1.3.** Given a simple element \(s\), the right complement of \(s\) is defined by \(\vartheta(s) = s^{-1}\Delta\), and the left complement of \(s\) is \(\vartheta^{-1}(s) = s \Delta^{-1}\).

Notice that the map \(\vartheta : [1, \Delta] \to [1, \Delta]\) is a bijection of the (finite) set \([1, \Delta]\). Notice also that \(\vartheta^2(s) = \Delta^{-1}s\Delta\). We denote by \(\tau\) the inner automorphism of \(G\) corresponding to conjugation by \(\Delta\). Hence \(\vartheta^2(s) = \tau(s)\).

**Definition 1.4.** Given two simple elements \(a\) and \(b\), we say that the decomposition \(a \cdot b\) is left weighted if \(\vartheta(a) \wedge b = 1\) or, equivalently, if \(ab \wedge \Delta = a\). We say that the decomposition \(a \cdot b\) is right weighted if \(a \wedge^n \vartheta^{-1}(b) = 1\) or, equivalently, if \(ab \wedge^n \Delta = b\).

The process of bringing a product \(a \cdot b\) of two two simple elements \(a\) and \(b\) into left weighted form by replacing it with the product \((as) \cdot (s^{-1}b)\), where \(s = \vartheta(a) \wedge b\), is called a local left sliding or simply a local sliding [11]. Local right sliding is defined analogously.
Definition 1.5. 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 $x_i \in \{1, \Delta\} \setminus \{1, \Delta\}$ for $i = 1, \ldots, r$ and $x_ix_{i+1}$ is a left weighted decomposition for $i = 1, \ldots, r - 1$. We say that a decomposition $x = y_1 \cdots y_i \Delta^p$ is the right normal form of $x$ if $y_i \in \{1, \Delta\} \setminus \{1, \Delta\}$ for $i = 1, \ldots, r$ and $y_iy_{i+1}$ 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 4.2 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 1.6. Given $x \in G$, whose left normal form is $\Delta^p x_1 \cdots x_r$, and whose right normal form is $y_1 \cdots y_i \Delta^p$, 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 [7] that $\inf(x)$ and $\sup(x)$ are precisely the maximal and minimal integers, respectively, such that $\Delta^\inf(x) \leq x \leq \Delta^\sup(x)$ (or, equivalently, $\Delta^\sup(x) \geq x \geq \Delta^\inf(x)$). Moreover, if $x = \Delta^p x_1 \cdots x_r$ is in left normal form, then the left normal form of $x^{-1}$ is precisely $x^{-1} = \Delta^{-p} \cdots x_{r-1} \Delta x_r \Delta^{-2p} \cdots x_1$. 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 $\ell(x^{-1}) = \ell(x)$.

Definition 1.7. Given $x \in G$, its (left) initial factor $\iota(x)$ is defined as $\iota(x) = x\Delta^{-\inf(x)} \land \Delta$ and its (left) final factor $\varphi(x)$ is defined as $\varphi(x) = (\Delta^\sup(x) \land x)^{-1} x$.

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 $\iota(x^{-1}) = \partial(\varphi(x))$. Right versions $\iota^r(x) = \Delta^{-\inf(x)} x \land \Delta$ and $\varphi^r(x) = x(\Delta^{\sup(x) - 1} \land x)^{-1}$ can be defined analogously.

Definition 1.8. Let $x^G$ denote the conjugacy class of $x$ in $G$ and define the summit infimum $\inf_s(x)$ respectively the summit supremum $\sup_s(x)$ of $x$ by $\inf_s(x) = \max\{\inf(y) \mid y \in x^G\}$ and $\sup_s(x) = \min\{\sup(y) \mid y \in x^G\}$. The set $\text{SSS}(x) = \{y \in x^G \mid \inf(x) = \inf_s(x), \sup(y) = \sup_s(x)\}$ is called the super summit set of $x$; the elements of $\text{SSS}(x)$ are called super summit elements.

It is well known that $\text{SSS}(x) \subset x^G$ is non-empty and finite [7] and it is clear from the definition that $\text{SSS}(x)$ only depends on the conjugacy class of $x$. Since, by the above remark, $\inf(y^{-1}) = -\sup(y)$ and $\sup(y^{-1}) = -\inf(y)$ for all $y \in G$, one has $y \in \text{SSS}(x)$ if and only if $y^{-1} \in \text{SSS}(x^{-1})$.

1.2 Cyclic sliding

Before explaining our algorithm, we need to describe the underlying operation called cyclic sliding introduced in [11]. 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 $(G, P, \Delta)$.

Definition 1.9. Given $x \in G$, the preferred prefix $p(x)$ of $x$ is the simple element
\[
p(x) = x \Delta^{-\inf(x)} \land (x^{-1} \Delta^\sup(x)) \land \Delta = \iota(x) \land \iota(x^{-1}) = \iota(x) \land \partial(\varphi(x)),
\]
and the preferred suffix $p^\gamma(x)$ of $x$ is the simple element
\[
p^\gamma(x) = (\Delta^{-\inf(x)} x) \land (\Delta^\sup(x) x^{-1}) \land \Delta = \iota^\gamma(x) \land \iota^\gamma(x^{-1}) = \iota^\gamma(x) \land \partial^{-1}(\varphi^\gamma(x)).
\]
Definition 1.10. Given $x \in G$, the cyclic left sliding $s(x)$ of $x$ is the conjugate of $x$ by its preferred prefix, that is,
\[ s(x) = x^p(x) = p(x)^{-1}x p(x), \]
and the cyclic right sliding $s^\gamma(x)$ of $x$ is the conjugate of $x$ by the inverse of its preferred suffix:
\[ s^\gamma(x) = x^{p^{\gamma}(x)^{-1}} = p^{\gamma}(x)^{-1}x p^{\gamma}(x). \]
If there is no possible confusion, we will call $s(x)$ 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 $u \rightarrow v$ if $u^s = v$ for some $u, s, v \in G$. Hence we have:
\[ x \xrightarrow{p(x)} s(x) \quad \text{and} \quad x \xleftarrow{p^\gamma(x)} s^\gamma(x). \]

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

Definition 1.11. An element $x \in G$ is called left rigid or just rigid if $p(x) = 1$. Similarly, $x$ is called right rigid if $p^\gamma(x) = 1$.

The concept of rigidity was introduced in [1] and some of the properties of rigid elements were analysed there. 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 1.12. We say that $y \in G$ belongs to a sliding circuit if $s^m(y) = y$ for some $m \geq 1$. Given $x \in G$, we define the set of sliding circuits of $x$, denoted by $SC(x)$, as the set of all conjugates of $x$ which belong to a sliding circuit.

Since the partial order $\preceq$ is invariant under $\tau$, one has $p(\tau(y)) = \tau(p(y))$, whence $\tau$ and $s$ commute. In particular, one has $y \in SC(x)$ if and only if $p^\Delta y = \tau^k y \in SC(x)$ for all $k \in \mathbb{Z}$.

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

Definition 1.13. Given $x \in G$, the sliding circuits graph $SCG(x)$ of $x$ is the directed graph whose set of vertices is $SC(x)$ and whose arrows correspond to conjugating elements as follows: There is an arrow which starts at $u \in SC(x)$, ends at $v \in SC(x)$ 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 $1 < t < s$ and $u^t \in SC(x)$.

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

Finally, we need to define two operations that will be applied to the conjugating elements. They are analogous to the ones defined in [10], and we use the same names.
Definition 1.14. Given \(x, \alpha \in G\), we define the transport of \(\alpha\) at \(x\) under cyclic sliding as

\[
\alpha^{(1)} = p(x)^{-1} \alpha p(x^{\alpha}).
\]

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:

\[
\begin{array}{ccc}
x & \xrightarrow{p(x)} & s(x) \\
\alpha & \downarrow & \alpha^{(1)} \\
x^{\alpha} & \xrightarrow{p(x^{\alpha})} & s(x^{\alpha})
\end{array}
\]

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 \(s^{-1}(x)\). 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 \(y = s(z) \in SC(x)\) 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 3.10). 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 3.15.** Let \(x \in G\), \(z \in SC(x)\), \(y = s(z)\) 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 3.10, is

\[
s^{(1)} = (p(z) \ s \ p^{\circ}(y^{s})^{-1}) \vee 1.
\]

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:

\[
\begin{array}{ccc}
z & \xrightarrow{p(z)} & y \\
\beta & \downarrow & \downarrow s \\
s(y^{s}) & \xrightarrow{p^{\circ}(y^{s})} & y^{s}
\end{array}
\]

### 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 \(\preceq\) and \(\succeq\). More precisely, one should have the following:

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 \(a \preceq s\) 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 \(s \succeq a\) and, in that case, computes the simple element \(sa^{-1}\).
In Section 4.1 we will see how, provided the above requirements are fulfilled, one can compute right and left complements, gcds and lcm, normal forms, preferred prefix 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 $\tilde{x}$ in the set $SC(x)$, starting from an arbitrary element $x \in G$. The algorithm also computes a conjugating element from $x$ to $\tilde{x}$. Algorithm 2 computes the arrows in the graph $SCG(x)$ which start at a given vertex; this is necessary for computing the entire set $SC(x)$. Moreover, knowing all arrows of the graph will allow us to compute a conjugating element for every pair of elements in $SC(x)$. 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 [7] 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 [10], applied to cyclic sliding instead of cycling. Algorithm 3 is not new, since it is implicitly or explicitly described in [7, 9, 10] 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 $SC(x)$ is a subset of all of these sets [11].

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 §4.2 for remarks concerning efficient implementation of the algorithms.

**Algorithm 1: Computing one element in $SC(x)$**

**Input:** $x \in G$.

**Output:** $\tilde{x} \in SC(x)$ and $c \in G$ such that $x^c = \tilde{x}$.

1. Set $\tilde{x} = x$, $c = 1$ and $T = \emptyset$.
2. While $\tilde{x} \notin T$, set $T = T \cup \{\tilde{x}\}$, $c = c \cdot p(\tilde{x})$ and $\tilde{x} = s(\tilde{x})$.
3. Set $y = s(\tilde{x})$ and $d = p(\tilde{x})$.
4. While $y \neq \tilde{x}$, set $d = d \cdot p(y)$ and $y = s(y)$.
5. Return $\tilde{x}$ and $c = c \cdot d^{-1}$. 
**Algorithm 2: Computing the arrows in SCG(x) starting at a given vertex**

**Input:** $v \in SC(x)$.  
**Output:** The set $A_v$ of arrows in the graph SCG(x) starting at $v$.

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

**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 $\tilde{x} \in SC(x)$ and $\tilde{y} \in SC(y)$, together with conjugating elements $c_1$ and $c_2$ such that $x^{c_1} = \tilde{x}$ and $y^{c_2} = \tilde{y}$.
2. Set $V = \{\tilde{x}\}$, $V' = \{\tilde{x}\}$ and $c_\tilde{x} = 1$.
3. While $V' \neq \emptyset$, do:
   (a) Take $v \in V'$.
   (b) Use Algorithm 2 to compute $A_v$.
   (c) For every $s \in A_v$, do:
      i. If $v^s = \tilde{y}$, then set $c_{\tilde{y}} = c_v \cdot s$. Return ‘$x$ and $y$ are conjugate by $c_1 \cdot c_{\tilde{y}} \cdot c_2^{-1}$’.
      STOP.
      ii. If $v^s \notin V$, then set $c_{v^s} = c_v \cdot s$, $V = V \cup \{v^s\}$, and $V' = V' \cup \{v^s\}$.
   (d) Remove $v$ from $V'$.
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 \([11]\) 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 \(\text{SC}(x)\). Moreover, iterated cyclic sliding achieves the minimal canonical length in the conjugacy class, that is, \(\text{SC}(x) \subseteq \text{SSS}(x)\). More precisely, one has the following.

**Lemma 2.1** ([11, Lemma 3.4]). For every \(x \in G\), one has the inequalities \(\inf(s(x)) \geq \inf(x)\), \(\sup(s(x)) \leq \sup(x)\), and \(\ell(s(x)) \leq \ell(x)\).

**Corollary 2.2** ([11, Corollary 3.5]). For every \(x \in G\), iterated application of cyclic sliding eventually reaches a period, that is, there are integers \(0 \leq i < j\) such that \(s^i(x) = s^j(x)\). In particular, one has \(s^n(x) \in \text{SC}(x)\) and \(s^{j-i}(s^i(x)) = s^j(x)\) for all \(k \geq i\).

**Proposition 2.3** ([11, Corollary 3.9]). For every \(x \in G\), if \(\ell(x)\) is not minimal in the conjugacy class of \(x\), then \(\ell(x) > \ell(s^m(x))\) for some positive integer \(m < ||\Delta||\). In particular, one has \(\text{SC}(x) \subseteq \text{SSS}(x)\).

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 \(\Delta\), and it respects the partial order \(\preceq\) as well as gcds with respect to \(\preceq\). One has:

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

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

*Proof.* Claim 1 follows from [11, Lemma 3.15] and is equivalent to Claim 2, as \(\alpha^{(1)} = p(x)^{-1} \alpha p(x^\alpha)\). Claims 3, 4, 5 and 6 are special cases of [11, Lemma 3.16], [11, Corollary 3.18], [11, Corollary 3.19] and [11, Proposition 3.20], respectively.

Applying iterated cyclic sliding to a conjugate \(y^s\) of \(y \in \text{SC}(x)\) will eventually produce another element of \(\text{SC}(x)\) 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 \(\text{SC}(x)\) has been reached.
Lemma 2.5 ([11, Lemma 3.32]). Let $x \in G$, $y \in \text{SC}(x)$ and $s \in G$ such that $y^s \in \text{SSS}(x)$. Let $N$ be a positive integer such that $s^N(y) = y$ 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 $s^{(i_1N)} = s^{(i_2N)}$.
2. $y^s \in \text{SC}(x)$ 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 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 SSS$(x)$.

Proposition 2.6 ([9, Proposition 4.12] or [11, Proposition 3.29]). Let $x \in G$. If $x^\alpha, x^\beta \in \text{SSS}(x)$ for elements $\alpha, \beta \in G$, then $x^\alpha \wedge \beta \in \text{SSS}(x)$.

Corollary 2.7 ([12, Theorem 2.4] or [11, Corollary 3.30]). Let $x \in G$. If $x^\alpha, x^\beta \in \text{SSS}(x)$ for elements $\alpha, \beta \in G$, then $x^\alpha \vee \beta \in \text{SSS}(x)$.

Corollary 2.8 ([11, Corollary 3.31]). Let $x \in G$. There is a unique positive element $\rho(x)$ (possibly trivial) satisfying the following.

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

The analogous properties for SC$(x)$ were shown in [11]. They in particular imply that SCG$(x)$ is a finite and connected directed graph.

Proposition 2.9 ([11, Proposition 3.33]). Let $x \in G$. If $x^\alpha, x^\beta \in \text{SC}(x)$ for elements $\alpha, \beta \in G$, then $x^\alpha \wedge \beta \in \text{SC}(x)$.

Corollary 2.10 ([11, Corollary 3.34]). Let $x \in G$. There is a unique positive element $c(x)$ (possibly trivial) satisfying the following.

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

Corollary 2.11 ([11, Corollary 3.35]). For every $x \in G$, the graph SCG$(x)$ is finite and connected. Moreover, the arrows of SCG$(x)$ correspond to simple elements, and the number of arrows starting at a given vertex is bounded above by the number of atoms of $G$.

Cyclic right sliding and right transport

Recall that in a Garside group $G$ with Garside structure $(G, P, \Delta)$, apart from the prefix order $\preceq$, one also has the suffix order $\succ$, defined by $\alpha \succ b$ if and only if $ab^{-1} \in P$. With respect to the latter, one can consider the the notions of preferred suffix, cyclic right sliding and set of right sliding circuits (denoted SC$^r(x)$), which are analogous to those of preferred prefix, cyclic sliding and set of sliding circuits, but refer to the partial order $\succ$ instead of $\preceq$ (cf. Definitions 1.9, 1.10 and 1.12).

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 $\succ$, and tries to relate them to the analogous notions with respect to $\preceq$, one must consider conjugating elements on the left, meaning that a (left) conjugating element $\alpha$ relates $x$ to $x^{\alpha^{-1}} = \alpha x \alpha^{-1}$.
Definition 2.12. Given $x, \alpha \in G$, we define the right transport of $\alpha$ at $x$ under cyclic right sliding as $\alpha^{(1)} = p^n(x^{\alpha^{-1}}) \alpha p^n(x)^{-1}$. 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:

$$
\begin{array}{ccc}
  x & \xleftarrow{\alpha} & p^n(x) \\
  \downarrow & & \downarrow \\
  x^{\alpha^{-1}} & \xleftarrow{p^n(x^{\alpha^{-1}})} & s^n(x^{\alpha^{-1}})
\end{array}
$$

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 [11], $(G, P^{-1}, \Delta^{-1})$ also is a Garside structure for $G$, called the reverse Garside structure, and cyclic right sliding and right transport with respect to $(G, P, \Delta)$ are just cyclic (left) sliding and (left) transport with respect to $(G, P^{-1}, \Delta^{-1})$. We refer to [11, §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 $\inf(s^n(x)) \geq \inf(x)$, $\sup(s^n(x)) \leq \sup(x)$, and $\ell(s^n(x)) \leq \ell(x)$.

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

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

Proposition 2.15 ([11, Proposition 3.26]). Let $x \in G$. Then for any $z \in \text{SSS}(x)$ one has $p^n(s(z)) \geq p(z)$ and $p^n(z) \leq p(s^n(z))$.

3 Description of the algorithm

In this section we will explain the algorithms from §1.3 and prove their correctness. The main idea of these algorithms, as for the previous solutions to the conjugacy problem given in [7, 9, 10], 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 $\text{SC}(x)$, the vertex set of the connected graph $\text{SCG}(x)$, and the conjugating elements will be paths in $\text{SCG}(x)$.

3.1 Algorithm 3

We start by explaining Algorithm 3 from §1.3. We remark that analogues of this algorithm, which use other sets instead of $\text{SC}(x)$, are already given in [7, 9, 10]. We explain the version given in this paper which uses the invariant $\text{SC}(x)$.

It is clear from the definition that $\text{SC}(x)$ is an invariant subset of the conjugacy class of $x$. Moreover, we will see that Algorithm 1 computes, given $x \in G$, an element $\bar{x} \in \text{SC}(x)$, that is, $\text{SC}(x)$ is non-empty. Hence, two elements $x$ and $y$ are conjugate if and only if $\text{SC}(x) = \text{SC}(y)$ or, equivalently, $\text{SC}(x) \cap \text{SC}(y) \neq \emptyset$. Thus, knowing how to compute $\text{SC}(x)$, 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 $SC(x) = SC(y)$, we just need to find an element $z \in SC(x)$, a conjugating element $c$ form $x$ to $z$, and a conjugating element $c_2$ from $y$ to $z$. Then $c^{-1}c_2$ 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 $\tilde{x} \in SC(x)$ and a conjugating element $c_1$ from $x$ to $\tilde{x}$. Applying the same algorithm to $y$, we obtain an element $\tilde{y} \in SC(y) = SC(x)$ and a conjugating element $c_2$ from $y$ to $\tilde{y}$. Hence, in order to obtain a conjugating element from $x$ to $y$, we just need to find a conjugating element from $\tilde{x}$ to $\tilde{y}$. In other words, we need to know how to relate, through a conjugation, any pair of elements of $SC(x)$. This is achieved thanks to the connected graph $SCG(x)$, since the vertices of this graph correspond to the elements in $SC(x)$, and a path between two vertices corresponds to a conjugating element from one vertex to the other.

Algorithm 3 computes a conjugating element from $\tilde{x}$ to any other element in $SC(x)$, by computing a maximal tree of the graph $SCG(x)$. More precisely, the algorithm starts in step 2 by considering $V = V' = \{\tilde{x}\}$ and $c_2 = 1$. The set $V$ contains the elements which we know belong to $SC(x)$, so at the beginning it only contains $\tilde{x}$. The set $V'$ contains the elements of $V$ that have not yet been used in step 3 of the algorithm, so at the beginning $V' = \emptyset$. Finally, whenever a new element $v$ is added to $V$ (and also to $V'$), we compute an element $c_v$, which is a conjugating element from $\tilde{x}$ to $v$. Of course, in step 2 of the algorithm, the conjugating element from $\tilde{x}$ to $\tilde{y} \in V$ is $c_2 = 1$.

Now step 3 does the following: For a known element of $SC(x)$ which has not been processed before, that is, for some $v \in V'$, it calls Algorithm 2 to compute the arrows of $SCG(x)$ starting at $v$. For each such arrow $s$, it computes the endpoint $v^s$ of the arrow. If $v^s$ is not in $V$, this means that we encountered a new element of $SC(x)$, so we add it to both $V$ and $V'$, and at the same time compute a conjugating element from $\tilde{x}$ to $v^s$: Since we know a conjugating element $c_v$ from $\tilde{x}$ to $v$ and a conjugating element $s$ from $v$ to $v^s$, we can store $c_v \cdot c_s$ as conjugating element from $\tilde{x}$ to $v^s$. Notice that the procedure checks whether $v^s = \tilde{y}$, since in this case we have already found a conjugating element $c_{\tilde{y}}$ from $\tilde{x}$ to $\tilde{y}$ as desired. Concatenating it from the left with the conjugating element from $x$ to $\tilde{x}$ and from the right with the conjugating element from $\tilde{y}$ to $y$, this produces a conjugating element from $x$ to $y$ which becomes the output of the algorithm. If $\tilde{y}$ is not encountered, we remove $v$ from $V'$ 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 $V' \neq \emptyset$. Since $V \subseteq SC(x)$, where $SC(x)$ is a finite set, since every element of $V$ is added to $V'$ exactly once, and since the procedure removes one element from $V'$ each time it is executed, this means that at some point we will have $V' = \emptyset$ and the procedure will stop. At this point, the arrows starting at every element of $V$ have been processed (exactly once). Moreover, one has $V = SC(x)$, since otherwise there would exist some element $v \in V$ and some element $w \in SC(x) \setminus V$ such that there is an arrow in $SCG(x)$ from $v$ to $w$. (This follows, since the graph $SCG(x)$ is connected by 2.11.) But since $v \in V$ and $V' = \emptyset$, step 3 has been applied to $v$, which means that $w$ has been added to the set $V$, a contradiction. Therefore, when the procedure stops, one has $V = SC(x)$. If $\tilde{y}$ was not found in $V$, this means that $\tilde{y} \notin SC(x)$, 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 $\tilde{x} \in SC(x)$ and a conjugating element $c$ such that $x^c = \tilde{x}$. 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 $s^i(x) = s^j(x)$, that is, $s^i(x) \in \text{SC}(x)$. Algorithm 1 computes this element $s^i(x)$, where $i$ is minimal. This is done by storing all the elements $\{s^m(x) \mid m \geq 0\}$, the trajectory of $x$ under cyclic sliding, in a set called $T$. Initially, one has $T = \emptyset$ and $\tilde{x} = x$. At the beginning of the $k$-th iteration of the loop in step 2, one has $T = \{s^0(x), s^1(x), \ldots, s^{k-2}(x)\}$ and $\tilde{x} = s^{k-1}(x)$. If $\tilde{x} \notin T$, then $\tilde{x}$ is added to $T$ and cyclic sliding is applied to $\tilde{x}$ 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 $\tilde{x}$: At the beginning of the first iteration of the loop in step 2, $c = 1$ is a conjugating element from $x$ to $\tilde{x} = x$. In each iteration of the loop, the element $c$, which is a conjugating element from $x$ to $\tilde{x}$, is multiplied on the right by $p(\tilde{x})$, yielding a conjugating element from $x$ to $\tilde{s}(\tilde{x})$, and $\tilde{x}$ is replaced by $s(\tilde{x})$.

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

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

### 3.3 Algorithm 2

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

Proposition 2.9 and Corollary 2.10 have a consequence which is crucial for computing the sliding circuits graph of an element $x \in G$: Given $y \in \text{SC}(x)$ and $s \in G$, there is a unique $\prec$-minimal element $c_s = c_s(y)$ satisfying $s \preceq c_s$ and $y^{c_s} \in \text{SC}(x)$; specifically, $c_s(y) = s \cdot c(y^s)$. Moreover, as $s \preceq \Delta_{\text{sup}}(s)$ and $x^{\Delta_{\text{sup}}(s)} \in \text{SC}(x)$, we have $c_s(y) \preceq \Delta_{\text{sup}}(s)$. In particular, the number of arrows starting at a given vertex $y$ of SCG($x$) is bounded by the number of atoms of $G$ and the label of each arrow is a simple element. 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 $y \in \text{SC}(x)$ 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 $\preceq$. The basic idea is to apply iterated transport to a suitable element $p_s$, which is derived from $s$ and satisfies $s \preceq p_s \preceq c_s$, until that fixed point is reached. All we need to do is to ensure that $y^{p_s}$ is super summit (so that $\preceq$ is respected) and that $s \preceq p_s^{(kN)}$ 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 an element $\rho_s$ which satisfies $s \preceq \rho_s$ and $y^{\rho_s} \in \text{SSS}(x)$, and which is $\preceq$-minimal among all elements doing so; this is due to [9]. Note that $\rho_s \preceq c_s$ since SC($x$) $\subseteq$ SSS($x$). By Corollary 2.8, 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 3.1. 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 $y \in \text{SSS}(x)$, then the algorithm terminates after at most $\ell(s) \cdot \|\Delta\|$ passes through the loop.

Proof. Let $\alpha$ be a positive element such that $x^\rho \in \text{SSS}(x)$. Then $(x^\rho)^{-1} \in \text{SSS}(x^{-1})$ and we have $\inf((x^\rho)^{-1}) = -\inf_s(x)$ and $\sup((x^\rho)^{-1}) = -\sup_s(x)$. Thus $x^\rho \leq x^\rho\alpha = \alpha x^\rho \leq \alpha \Delta^{\sup_s(x)}$, whence $x^\rho \Delta^{-\sup_s(x)} \leq \alpha$ and, analogously, $(x^\rho)^{-1}\Delta^{\inf_s(x)} \leq \alpha$. As $\alpha$ is positive, the above implies $1 \vee (x^\rho)^{-1}\Delta^{\inf_s(x)} \vee x^\rho\Delta^{-\sup_s(x)} \leq \alpha$. Moreover, $1 \vee (x^\rho)^{-1}\Delta^{\inf_s(x)} \vee x^\rho\Delta^{-\sup_s(x)} = 1$ if and only if $\inf(x^\rho) \leq \sup_s(x)$ and $\inf(x^\rho) = \sup((x^\rho)^{-1}) \geq \inf_s(x)$, that is, if and only if $x^\rho \in \text{SSS}(x)$.

Hence, at any stage of the above algorithm, the element $\rho$ satisfies $\rho \leq c$ for every positive element $c \in G$ such that $x^c \in \text{SSS}(x)$. 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 $y \in \text{SSS}(x)$, then $\rho(x) \leq s^{-1}\Delta^{\sup_s(x)} \leq \Delta^{\ell(s)}$, whence the algorithm terminates after at most $\ell(s) \cdot \|\Delta\|$ steps.

Corollary 3.2. Steps 3 (a) and 3 (b) in Algorithm 2 compute the element $\rho_a$.

Proof. Note that in steps 3 (a) and 3 (b) of Algorithm 2 we have $v \in \text{SC}(x) \subseteq \text{SSS}(x)$ and $\ell(a_t) = 1$. Hence, by Proposition 3.1, steps 3 (a) and 3 (b) in Algorithm 2 compute exactly $a_t \cdot \rho(x^{a_t}) = \rho_a$ with at most $\|\Delta\|$ passes through the while loop.

The element $1 \vee (x^\rho)^{-1}\Delta^{\inf_s(x)} \vee x^\rho\Delta^{-\sup_s(x)} = (1 \vee (x^\rho)^{-1}\rho\Delta^{-\sup_s(x)}) \vee (1 \vee x^\rho\Delta^{-\sup_s(x)})$ in step 2 (a) of the algorithm in Proposition 3.1 can be computed efficiently using the following result.

Proposition 3.3. If $x \in G$ such that $\sup(x) = q + r$ with $0 \leq r \leq \ell(x)$, then $1 \vee x\Delta^{-q}$ is the product of the leftmost $r$ factors of the right normal form of $x$.

Proof. Observe that $a \leq b$ is equivalent to $b^{-1} \geq a^{-1}$ for all $a, b \in G$ by the definitions of $\preceq$ and $\succeq$. This implies that one has $a \vee b = (a^{-1} \wedge b^{-1})^{-1}$ for all $a, b \in G$. Hence, in particular, $1 \vee x\Delta^{-q} = (1 \wedge x\Delta^q)^{-1} = ((x \wedge x\Delta^q)^{-1})^q = x(x \wedge x\Delta^q)^{-1}$. Since $x \wedge x\Delta^q$ 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 3.4. Let $x \in G$, $y \in \text{SC}(x)$ and $u \in G$ such that $y^u \in \text{SSS}(x)$ and let $N$ be the length of the sliding circuit containing $y$, that is, let $N$ be the smallest positive integer such that $s^{N}(y) = y$. For integers $i \geq 0$ consider the transports $w^{(iN)}$ at $y$. By Lemma 2.5, there are integers $i_2 > i_1 \geq 0$ such that $w^{(i_1N)} = w^{(i_2N)}$. Let $i_1$ and $i_2$ be minimal subject to this condition and define $l(u) = i_2 - i_1$ and $F(u) = \{w^{(iN)} \mid i_1 \leq i < i_2\}$.
Note that $1 \in F(u)$ if and only if $F(u) = \{1\}$. Moreover by Lemma 2.5, $y^u \in SC(x)$ if and only if $i_1 = 0$, that is, if and only if $u \in F(u)$. Finally, for all $v \in F(u)$ and all $i \in \mathbb{N}$ we have $u^{(i)(u)(N)} = v$, in particular, $y^u \in SC(x)$. In other words, the set $F(u)$ contains those iterated transports $u^{(i)(N)}$ of $u$ along the sliding circuit of $y$ which are fixed by some iterated transport along the sliding circuit, that is, those which satisfy $y^{u^{(i)(N)}} \in SC(x)$.

**Lemma 3.5.** Let $x \in G$, $y \in SC(x)$, $s \in G$ 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 $s^N(y) = y$. If $c_s \leq c_s^{(i)(N)}$ for some $i > 0$ then $c_s^{(i)(N)} = c_s$.

**Proof.** Let $c_s^{(i)(N)} = c_s \gamma$ with a positive element $\gamma$. By induction, $c_s \gamma \leq c_s^{(ki)(N)}$ for all $k \geq 1$ from Proposition 2.4 (4). By the preceding remark, this in particular implies $c_s \leq c_s \gamma \leq c_s^{(ci)(N)} = c_s$, that is, $\gamma = 1$.

**Lemma 3.6.** Let $x \in G$, $y \in SC(x)$, $s \in P$ and denote $c_s = c_s(y)$. Assume that $p \in P$ satisfies $p \leq c_s$ and $y^p \in SSS(y)$ and that $F = F(p) \neq \{1\}$.

1. If there exists $v \in F$ such that $s \leq v$ then $c_s = v$.
2. If $s \not\leq v$ 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 $p^{(i)} \leq c_s^{(i)}$ for all $i > 0$.

Assume first that $v \in F$ such that $s \leq v$. Then $y^v \in SC(x)$ and the minimality of $c_s$ implies $c_s \leq v$. Now $v = p^{(i)(N)}$ for some $i$, whence $c_s \leq v = p^{(i)(N)} \leq c_s^{(i)(N)}$. Lemma 3.5 then yields $v = c_s$ and Claim 1 is shown.

Now assume that $s \not\leq v$ for all $v \in F$ and let $i$ be a multiple of $l(c_s)$ sufficiently large so that $v = p^{(i)(N)} \in F$. Since $1 \not\in F$, we have $v \neq 1$ and $y^v \in SC(x)$. Moreover, $v = p^{(i)(N)} \leq c_s^{(i)(N)} = c_s$ and $v \neq c_s$, since $s \not\leq v$ but $s \leq c_s$. 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 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 $a \leq c$ or, equivalently, such that $ho_a \leq c$. The above result says that if $F(\rho_a) \neq \{1\}$ then we will have no problem, since either $c_a$ can be computed by iterated transport (where $c_a$ is the only possible candidate for being such an arrow), or we can be sure that $c_a$ is not an arrow, since it is decomposable. Unfortunately, it may occur that $F(\rho_a) = \{1\}$, as we can see in the following example:

**Example 3.7.** Consider in the Artin braid group $B_5$ the elements $y = x = \Delta \cdot \sigma_3 \sigma_1 \sigma_4 \sigma_3 \sigma_4 \cdot \sigma_1$, in left normal form as written, and $s = \sigma_3 \sigma_2 \sigma_1$. It is easy to check that $s^6(y) = y$, that is, $y \in SC(x)$. Since $y^* = \Delta \cdot \sigma_1 \sigma_3 \cdot \sigma_3 \sigma_2 \sigma_1 \sigma_2$ is in left normal form as written, $y^* \in SSS(x)$, that is, $\rho_s = s$.

However, $s^{(1)} = p(y)^{-1} sp(y^*) = 1$ and hence $F(s) = \{1\}$, that is, the requirements of Lemma 3.6 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 3.6 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 3.8.** Let $x \in G$ and $v \in SC(x)$. Let $s \neq 1$ be a positive element such that $v^s \in SSS(x)$. If $s^{(k)} = 1$ for some $k \geq 1$, then $s \wedge p(v) \neq 1$. 

16
Proposition 2.4 (3).

Recall that the transport $F$ of Definition 3.10.

Let $\rho_k$ be such that $\rho_k > \rho$ be an atom such that $\rho_k \notin \mathrm{SC}(x)$. We will show the result by induction on $k$. By Proposition 2.4 (4), one then has

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

that is,

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

We will show the result by induction on $k$. If $k = 1$, one has $s \land p(v) = s \neq 1$. 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 $s^{(1)} \land p(s(v)) \neq 1$.

Recall that the transport $t^{(1)}$ of an element $t$ at $v$ satisfies $t^{(1)} = p(v)^{-1} t p(v')$. For $t = p(v)$ this yields $p(v^{(1)} = p(s^{1}(v)) = p(s(v))$. As the transport preserves $\land$ by Proposition 2.4 (6), one hence has $(s \land p(v))^{(1)} = s^{(1)} \land p(v^{(1)}) = s^{(1)} \land (s^{1} \land p(s(v)) \neq 1$, which implies $s \land p(v) \neq 1$ by Proposition 2.4 (3).

Corollary 3.9. Let $x \in G$ and $v \in \mathrm{SC}(x)$. Let $a$ be an atom such that $a \neq p(v)$. Then either $F(\rho_\alpha) \neq \{1\}$ or $c_a$ is not an indecomposable conjugator starting at $v$.

Proof. Suppose that $F(\rho_\alpha) = \{1\}$. This means that some iterated transport $(\rho_\alpha)^{(k)} = 1$ for some $k \geq 1$. By the above lemma, it follows that $\rho_\alpha \land p(v) \neq 1$. Hence there must exist an atom $b$ such that $b \neq \rho_\alpha \land p(v)$. Since $b \neq p(v)$ and $p(v') \in \mathrm{SC}(x)$, it follows that $c_b \neq p(v)$. On the other hand, since $b \neq \rho_\alpha \neq c_a$, it follows that $c_b \nsucc c_a$. But one cannot have $c_b = c_a$, otherwise $a \neq c_a = a \neq p(v)$, 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 $F(\rho_\alpha) \neq \{1\}$ then either $c_a$ can be found by iterated transport or $c_a$ is not indecomposable. Hence, if $a \neq p(v)$, 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 $a \neq p(v)$ and $F(\rho_\alpha) = \{1\}$.

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

Definition 3.10. Let $x \in G$, $z \in \mathrm{SC}(x)$, $y = s(z)$ and let $s \in G$ be positive. By Propositions 2.6 and Proposition 2.4 (6), there exists a unique positive $\nsucc$-minimal element $s_{(1)} \in G$ satisfying $z^{(1)} = s_{(1)}$ in $\mathrm{SSS}(x)$ and $s \nsucc (s_{(1)})^{(1)}$, 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 $s_{(k)} = s_{(k-1)}^{(k)}$ 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 $s^{k-1}(w) = y$. We also define $s_{(0)} = s$.

Lemma 3.11. Let $x \in G$, $z \in \mathrm{SC}(x)$, $y = s^k(z)$ 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 minimal positive element such that $s \nsucc (s_{(k)})^{(k)}$ and $z^{(k)} \in \mathrm{SSS}(x)$.

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 $s \nsucc (s_{(k-1)})^{(k-1)} \nsucc ((s_{(k-1)})^{(1)})^{(k-1)} = (s_{(k)})^{(k)}$. Moreover, if $\alpha$ is a positive element such that $s \nsucc \alpha^{(k)}$ and $z^{\alpha} \in \mathrm{SSS}(x)$, then $\alpha^{(1)}$ is a positive element such that $s \nsucc (\alpha^{(1)})^{(k-1)}$ and $s(z)^{\alpha^{(1)}} = s(z)^{\alpha} \in \mathrm{SSS}(x)$. Hence, $s_{(k-1)} \nsucc \alpha^{(1)}$ by induction. By definition of the pullback of $s_{(k-1)}$, we then have $s_{(k)} = (s_{(k-1)})^{(1)} \nsucc \alpha$, as we wanted to show.
Lemma 3.12. Let \( x \in G, z \in SC(x) \), \( y = s^k(z) \) for a positive integer \( k \) and let \( s, t \in G \) such that \( 1 \leq s \leq t \). Then, \( s(k) \leq t(k) \).

Proof. By Lemma 3.11, we have \( t \leq (t(k))^{(k)} \) and \( z^{(k)} \in SSS(x) \). But then \( s \leq t \leq (t(k))^{(k)} \) and, again using Lemma 3.11, we obtain \( s(k) \leq t(k) \) as we wanted to show.

Lemma 3.13. Let \( x \in G, \ z \in SC(x) \), \( y = s(z) \) and let \( s \in G \) be positive. Then the pullback \( s_{(i)} \) of \( s \) at \( y \) satisfies \( s_{(i)} \leq \Delta^\sup(s) \).

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

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

Proof. First note that by Lemma 3.13, we have \( 1 \leq s_{(k)} \leq \Delta^\sup(s) \) 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 \( s_{(i)} = s_{(j)} \).

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 \( i \)-th 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 \( v^s \in SC(x) \), we have \( (c_s)^{(N)} = c_s \) for some \( t \geq 1 \). Consider then \( M > i \) to be a multiple of \( t \), big enough so that \( p_{(M(j - i)N)} \in F(p) \). By Lemma 3.11, \( p = s_{(M(j - i)N)} \) is the minimal positive element such that \( s \leq p_{(M(j - i)N)} \). This implies that \( F(p) \neq \{1\} \) and that \( F(p) \) contains an element which admits \( s \) as a prefix. Moreover, \( s \leq c_s = (c_s)^{(M(j - i)N)} \), where the equality in the last step holds since \( M \) is a multiple of \( t \). By the minimality of \( p \) one finally has \( p \leq c_s \). We can then apply Lemma 3.6 to \( p \), and conclude that \( c_s = p_{(M(j - i)N)} \in F(p) \). Uniqueness also follows from Lemma 3.6.

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

We have \( s_{(iN)} \leq (s_{(kN)})^{((k-i)N)} \) and also \( s_{(kN)} \leq (s_{(iN)})^{((j-k)N)} = (s_{(iN)})^{((j-k)N)} \) by Lemma 3.11. Applying \( (k-i)N \)-fold transport to the second expression and using Proposition 2.4 (4), one obtains \( (s_{(kN)})^{((k-i)N)} \leq (s_{(iN)})^{((j-i)N)} \), whence \( s_{(iN)} \leq (s_{(kN)})^{((k-i)N)} \leq (s_{(iN)})^{((j-i)N)} \).

Using Proposition 2.4 (4) again, we can for any \( K \geq 0 \) apply \( K \)-fold transport to this expression and we see that \( (s_{(iN)})^{(K)} \leq (s_{(kN)})^{((K+k-i)N)} \leq (s_{(iN)})^{((K+j-i)N)} \) 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 \( c_{s'} = s' \leq (s_{(iN)})^{((j-i)N)} \) and hence \( s' = (s_{(iN)})^{((j-i)N)} \) by Lemma 3.5 (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 3.15. Let $x \in G$, $z \in SC(x)$, $y = s(z)$ 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 3.10, is

$$s_{(1)} = (p(z) \ s \ p^{y^s})^{-1} \vee 1.$$  

Proof. Let $u = (p(z) \ s \ p^{y^s})^{-1} \vee 1$. 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: $z^u \in SSS(x)$.

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

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

Proof: By definition, $u$ is positive and $u = p(z) \ s \ p^{y^s}$, where $\alpha = p^y(y^s) \wedge p(z)$. (Note that $\alpha^{-1} = p^y(y^s) \vee (p(z)s)^{-1}$.) By Proposition 2.15, $p^y(y^s) \preceq p(s^y(y^s))$ and since $p^y(y^s)\alpha^{-1}$ is positive and $s\preceq \alpha^{-1}$, we obtain with Proposition 2.4 (2)

$$p^y(y^s) \preceq p(s^y(y^s)) \preceq p(y^s)\alpha^{-1} p(s^y(y^s)) \alpha^{-1} \cdot p(y^s)\alpha^{-1} \preceq p(y^s)\alpha^{-1}$$

which implies $1 \preceq \alpha^{-1} \cdot p(y^s)$, Hence, $s \preceq s \alpha^{-1} \cdot p(y^s) = p(z) - 1 \cdot u(z) = u_{(1)}$.

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

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

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

$\square$
Example 3.16. Consider the situation from Example 3.7. The trajectory of \( y = \Delta \cdot \sigma_3 \sigma_1 \sigma_4 \sigma_3 \sigma_4 \cdot \sigma_1 \) 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 3.14, we have \( i = 1 \) and \( j = 2 \).

Computing iterated transports of \( p = s(1iN) = s(6) = \sigma_3 \sigma_4 \), we obtain \( p(12) = p(6) = \sigma_3 \sigma_2 \sigma_1 \sigma_4 \). Hence, we have \( F(p) = \{ p(6) \} \) and as \( s \not\leq p(6) \), we obtain \( c_s = p(6) = \sigma_3 \sigma_2 \sigma_1 \sigma_4 \).

Note that \( p \not\in F(p) \), 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 \( (\rho_{a_t})_{(1iN)} \), one of whose iterated transports is precisely \( c_{a_t} \). This computation is only done whenever \( a_t \not\leq p(v) \), 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 [11, Theorem 1.1] equivalently, has a rigid conjugate). The algorithm continues in step 3 (d) by applying iterated transport to the corresponding element (either \( \rho_{a_t} \) or \( (\rho_{a_t})_{(1iN)} \)) until the first repetition occurs. Then, step 3 (e) checks whether any of the elements in \( F(\rho_{a_t}) \) respectively \( F((\rho_{a_t})_{(1iN)}) \) admit \( a_t \) as a prefix, in which case it will precisely be \( c_{a_t} \) by Lemma 3.6. If \( a_t \) does not occur as a prefix, then \( c_{a_t} \) is not indecomposable by Lemma 3.6, Corollary 3.9 and Proposition 3.14.

However, even if \( c_{a_t} \) occurs as an element of \( F(\rho_{a_t}) \) respectively \( F((\rho_{a_t})_{(1iN)}) \), 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 \( a_{i_1} \) such that \( c_{a_{i_1}} \) is an indecomposable conjugator starting at \( v \) and \( k = \max \{ i \mid \alpha_i \not\leq c_{a_{i_1}} \} \). 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 \( c_{a_l} \not\leq c_{a_t} \), 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 \( a_l \in Atoms \). Therefore, if \( a_k \not\leq c_{a_t} \) for all \( a_k \in Atoms \) 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 [13] 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 [8] and [3]). Hence one should not use the algorithms below if one just needs to make computations in braid groups.
Assumption 4.1. Let $G$ be a Garside group of finite type. We assume that the list of atoms $A = \{a_1, \ldots, a_\lambda\}$ of $G$ is 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 $a \in A$ and a simple element $s$, test whether $a \preceq s$ (respectively $s \succeq a$) and, if yes, compute the simple element $a^{-1}s$ (respectively $sa^{-1}$).

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 $a \in A$ 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 $a_i \not\preceq s$ for all $i = 1, \ldots, \lambda$, where the latter condition can be tested using the operation (Op) at most $\lambda$ 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 $\Delta$ and the operations (Op) and (Op2) can be realised in terms of (Op3) at a cost of $O(C\lambda \|\Delta\|)$; the map $\partial$ induces a duality between this situation and the situation from Assumption 4.1. Finally, note that (Op1) can be realised in terms of (Op3) at a cost of $O(C\lambda)$, if $\Delta$ is the lcm of the atoms of $G$: in this case, $\partial(s) = \Delta$ is equivalent to $a_i \preceq \partial(s)$ 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 $a \preceq s$, 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 $a \preceq s$ for every $a \in A$, until the answer is positive. This has time complexity $O(C\lambda)$. Therefore, in the algorithms below we will sometimes write ‘Take an atom $a \preceq s$’, 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.

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 $\Delta$ or $\Delta^{-1}$. We will also see a generic way of performing the operations (Op2) and (Op3). The following algorithm underlies all of these:
Algorithm to compute 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 $a \preceq s$.
   (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 $\preceq$ by $\succ$ and multiplying with $a^{-1}$ on the right instead of on the left. The given algorithm can also be used to compute $\partial^{-1}(s)$, 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 $a \preceq \partial(s)$, where the latter step has a cost of $O(C)$ by Assumption 4.1. 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 $\partial^{-1}(s) \succ a$ 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.

Algorithm to compute the greatest common divisor of two simple elements

**Input:** Two simple elements $s$ and $t$.

**Output:** The simple element $s \land t$.

1. Set $i = 1$ and $d = \Delta$.
2. While $i \leq \lambda$ do:
   (a) If $a_i \preceq s$ and $a_i \preceq t$, 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 $\lambda$ 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 $s \land^\succ t$ with respect to $\succ$.  

22
Least common multiples of simple elements with respect to $\preceq$ or $\succeq$ can now be computed using the following formulae, which can easily be seen to hold:

$$s \vee t = \partial^{-1}(\partial(s) \wedge \partial(t)), \quad s \wedge t = \partial((\partial^{-1}(s) \wedge \partial^{-1}(t))$$.

Therefore, computing $s \vee t$ or $s \wedge t$ 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).

### Algorithm to test whether two simple elements are equal

**Input:** Two simple elements $s$ and $t$.

**Output:** The truth value of $s = t$.

1. Set $i = 1$ and $d = \Delta$.
2. While $i \leq \lambda$ do:
   (a) If $a_i \preceq s$ and $a_i \preceq t$, 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. 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 as the one computing the gcd of two simple elements:

### Algorithm to compute 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 $a_i \preceq s'$ and $a_i \preceq t$, 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 $\partial^{-1}(s'), t$.

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 §1.1), based on the following well-known result.
Proposition 4.2 (see, for example, [4, Props. 3.1 and 3.3] or [8]). 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 \( s_i' s_{i+1} \), that is, let \( t_i = \partial(s_i' s_{i+1}) \) and define \( s_i'' = s_i's_{i+1} t_i \) 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 \( t_i = \partial(s_i s_{i+1}) \) and define \( s_i' = s_i t_i \) and \( s_i'' = t_i^{-1} s_i' \). Finally define \( s_1'' = s_1' \). Then, \( s_1'' \cdots s_k'' s_{k+1}' \) is the left normal form of \( s_1 \cdots s_k s_{k+1}' \) (where possibly \( s_1'' = \Delta \) or \( s_k'' = 1 \)).

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 \( \Delta \) or \( \Delta^{-1} \) on the left hand side, applying \( \tau \) or \( \tau^{-1} \) as required, so that the element will be written as \( \Delta s_1 \cdots s_k \), where each \( s_i \) is a simple element; the number of applications of \( \tau \) or \( \tau^{-1} \) is bounded by \( k(k-1)/2 \) and each application has a cost of \( O(CA \|\Delta\|) \). Finally, one applies local slidings to every pair of consecutive simple elements until every pair is left weighted; it follows from Proposition 4.2 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 \( \tau \) or \( \tau^{-1} \) is bounded by \( k \) and only \( O(k) \) local slidings are required by Proposition 4.2.

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 \( \Delta^q b_1 \cdots b_l \). 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 \( \Delta \)).

Algorithm to compute 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 \((a \land b) \land 1 = 1\). It is then easy to see by induction that after the \( i \)-th pass through the while loop one has \( d = (a \land b) \land \Delta^i\). 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 \( k \). 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 4.2. 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 \( \gcd \) \( a \land^i b \) 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 \( d = (a^{-1}\Delta^m) \land^n (b^{-1}\Delta^m) \) using (the right version of) the algorithm above. Then, \( a \lor b = (a^{-1} \land^n b^{-1})^{-1} = \Delta^m((a^{-1}\Delta^m) \land^n (b^{-1}\Delta^m))^{-1} = \Delta^m d^{-1}\). This 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 \( a \lor b \) is \( O(C\lambda k^2 \|\Delta\|)\). Computing the right \( \text{lcm} \) \( a \lor^i b \) 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 1.9, namely \( p(x) = \iota(x) \land \partial(\varphi(x)) \). Since \( \iota(x) = \tau^{-\rho}(x_1) \) with \(|\rho| \leq k\) and \( \varphi(x) = x_r \), the complexity of computing \( p(x) \) from the normal form of \( x \) is \( O(C\lambda k \|\Delta\|)\). The normal form of \( s(x) = x^p(x) \) 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 \( p(x) \) and the normal form of \( s(x) \) can be obtained at a cost of \( O(C\lambda k \|\Delta\|)\).

The transport of an element \( \alpha \) at an element \( x \) is given by the formula \( \alpha^{(1)} = p(x)^{-1} \alpha p(x^\alpha) \). If \( x \) and \( \alpha \) 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 \( \alpha \) is simple, then the normal form of \( x^\alpha \) 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 3.15, is \( s^{(1)} = (p(z) \land s \land (y^s)^{-1}) \lor 1 \); 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 \( p(z) \land s \land (y^s) \) 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 \( \text{lcm} \) 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 \( s^{(1)} \) is analogous and has the same complexity.
Summarising the results obtained in this section, we have:

**Theorem 4.3.** Let \( G \) be a Garside group of finite type with Garside element \( \Delta \) and set of atoms \( \mathcal{A} = \{ a_1, \ldots, a_\lambda \} \) for which Assumption 4.1 is satisfied. Moreover, let \( a \) be an atom of \( G \), let \( s \) and \( t \) be simple elements of \( G \) and let \( x, y \) and \( \alpha \) 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 \( s \wedge t \), \( s \wedge^* t \), \( s \vee t \) or \( s \vee^* t \).
   - 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 \( \{ \text{resp. the right normal form} \} \) of \( xs \), \( sx \), \( xs^{-1} \), \( s^{-1}x \), \( x^s \) or \( x^{s^{-1}} \), if the left normal form \( \{ \text{resp. the right normal form} \} \) of \( x \) is known.
   - Compute \( p(x) \) or \( s(x) \) \( \{ \text{resp. } p^\lambda(x) \text{ or } s^\lambda(x) \} \), if the left normal form \( \{ \text{resp. the right normal form} \} \) of \( x \) is known.
   - Compute the left transport \( s^{(1)} \) \( \{ \text{resp. the right transport } s^{(1)} \} \) of \( s \) at \( x \), if the left normal form \( \{ \text{resp. the right normal form} \} \) of \( x \) is known.
   - Compute the left pullback \( s^{(1)} \) \( \{ \text{resp. the right pullback } s^{(1)} \} \) of \( s \) at \( x \), if it is defined and if the right normal form \( \{ \text{resp. the left normal form} \} \) of \( x \) and the left normal form of the element \( z \in SC(x) \) satisfying \( s(z) = x \) \( \{ \text{resp. the right normal form of the element } z \in SC^\lambda(x) \text{ satisfying } s^\lambda(z) = x \} \) 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 \( x \wedge y \), \( x \wedge^* y \), \( x \vee y \) or \( x \vee^* y \).
   - Compute \( p(x) \), \( p^\lambda(x) \), \( s(x) \) or \( s^\lambda(x) \).
   - Compute the left transport \( \alpha^{(1)} \) of \( \alpha \) at \( x \) or the right transport \( \alpha^{(1)} \) of \( \alpha \) at \( x \).
   - Compute the left pullback \( \alpha^{(1)} \) \( \{ \text{resp. the right pullback } \alpha^{(1)} \} \) of \( \alpha \) at \( x \), if it is defined and if the element \( z \in SC(x) \) satisfying \( s(z) = x \) \{ resp. the element \( z \in SC^\lambda(x) \text{ satisfying } s^\lambda(z) = x \} \) 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 \( SC(x) \) from Section 1.3. Firstly, we define some bounds which will be used in the sequel.
Notation 4.4. 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 $s^i(x) = s^j(x)$.

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

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

Remark 4.5. 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 $s^m(x) \in SSS(x)$ for all $m \geq k \|\Delta\|$. As $|SSS(x)| \leq |[1, \Delta]|^k$, one can choose $T \leq k \|\Delta\| + |[1, \Delta]|^k$. Moreover, as $SC(x) \subseteq SSS(x)$ is finite, $M$ as above exists and one can choose $M \leq |SC(x)|$. (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 4.6. Let $x \in G$, $z \in SC(x)$, and let $s$ be a simple element such that $z^s \in SSS(x)$. If $N$, $i$, $j$ and $K$ are integers such that $s^N(z) = z$, $0 \leq i < j \leq K$ and $(s^{(iN)})(s^{(jN)}) = (s^{(iN)})(s^{(jN)})$, where $\ell^{(m)}$ 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 $s^{(iN)} = s^{(jN)}$.

Proof. By Lemma 3.11 we have $s^{(iN)} = s^{(jN)}$. Again using Lemma 3.11, we obtain $s^{(iN)} \neq s^{(jN)}$, that is, $s^{(iN)} \neq s^{(jN)}$.

Similarly, we have $s^{(iN)} = s^{(jN)}$ and from this obtain $s^{(iN)} \neq s^{(jN)}$, that is, $s^{(iN)} \neq s^{(jN)}$. Applying $(j-i)N$-fold pullback to the last statement yields $s^{(iN)} \neq s^{(jN)}$ using Lemma 3.12.

Hence, $s^{(iN)} = s^{(jN)}$ as we wanted to show. 

Corollary 4.7. Consider for $x \in G$ the bounds from Notation 4.4. For any element $z \in SC(x)$ and any simple element $s$ satisfying $z^s \in SSS(x)$ there exist two integers $0 \leq i < j \leq 2R$ satisfying $s^{(iN)} = s^{(jN)}$, where $s^N(z) = z$ and $s^{(m)}$ denotes $m$-fold pullback at $z$ for $m \in \mathbb{N}$.

Proof. By the choice of $R$ there are integers $0 \leq i' < j' \leq R$ such that $(s^{(iN)})(s^{(jN)}) = (s^{(iN)})(s^{(jN)})$. We then have $s^{(iN)} = s^{(jN)}$ by Lemma 4.6. Setting $i = R$ and $j = R + j' - i'$, we have $0 \leq i < j \leq 2R$ and $s^{(iN)} = s^{(jN)}$ as desired.

Proposition 4.8. Let $G$ be a Garside group of finite type with Garside element $\Delta$ and $\lambda$ 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 4.4, the complexity of Algorithm 1 is $O(C\lambda k (k + T) \|\Delta\|)$.

Proof. Observe that $\ell(s^i(x)) \leq k$ 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 4.3. 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 $T$ in step 2. If the normal form of an element $y$ with $\ell(y) \leq k$ is known, testing whether $y \in T$ (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 4.3. 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 4.9.** Let $G$ be a Garside group of finite type with Garside element $\Delta$ and $\lambda$ atoms, and let $v$ be an element of $G$ 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 4.4, the complexity of Algorithm 2 is $O(C\lambda^2 k \|\Delta\| (\|\Delta\| + RM))$.

**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 4.3. Hence, the cost of step 1 is $O(C\lambda k M \|\Delta\|)$.

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 $\lambda$ flags.

Steps 3(a) to (e) are executed $\lambda$ 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 $a_t \preceq s \preceq \rho_a \preceq \Delta$, 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 4.3. By Proposition 3.3, we can obtain the element $1 \lor v^s \Delta^{-\sup(v)}$ 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 $1 \lor (v^s)^{-1} \Delta^{-\inf(v)}$ 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-$\Delta$ factor in the right normal form of $v^s$ by applying the map $\partial$ or $\partial^{-1}$ at most $2k + 1$ times, that is, at a cost of $O(C\lambda k \|\Delta\|)$ by Theorem 4.3. As both $1 \lor v^s \Delta^{-\sup(v)}$ and $1 \lor (v^s)^{-1} \Delta^{-\inf(v)}$ 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 4.3. Hence, since the number of passes through the while loop is at most $\|\Delta\|$ by Proposition 3.1, step 3(b) has a cost of $O(C\lambda k \|\Delta\|^2)$.

In step 3(c) the initial test $a_t \preceq \rho(v)$ has a cost of $O(1)$. We can store the simple elements $s_{(1:N)}$ ($i = 1, 2, \ldots$) in a sufficiently large hash table, using the hash function from Assumption 4.1. Testing whether $s_{(1:N)}$ 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 4.3. As the number of pullbacks which need to be computed is bounded by 2RM by Corollary 4.7, the cost of step 3(c) hence is $O(C\lambda k RM \|\Delta\|)$.

By the same arguments, step 3(d) has a cost of $O(C\lambda k RM \|\Delta\|)$, 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 $O(CA)$, 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. $\square$
Theorem 4.10. Let $G$ be a Garside group of finite type with Garside element $\Delta$ and $\lambda$ 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 4.4 for $x$ and $y$, respectively.

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

Proof. Observe that $\ell(z) \leq k$ for all $z \in SC(x)$. In particular, the (left) normal forms of two such elements can be compared at a cost of $O(C\lambda k \|\Delta\|)$ by Theorem 4.3. 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 $V$. More precisely, whenever a new element $v^s \in SC(x)$ is found, where $s$ is an indecomposable conjugator and $v \in V$, we store the left normal form and the right normal form of $v^s$, as well as the indecomposable conjugator $s$, in the hash table entry for $v^s$. If the left normal form and the right normal form of $v^s$ are known, testing whether $v^s \in V$, and storing all required data if it is not, has a cost of $O(C\lambda k \|\Delta\|)$. The set $V^\prime$ is stored as a list (storing hash table indices instead of actual elements), whence storing or retrieving an element of $V^\prime$ has a cost of $O(1)$.

Observe that the conjugating elements $c_v$ for $v \in V$ are implicit in the spanning tree structure for $SCG(x)$ with root $\tilde{x}$ which is computed: for any $v \in V$, 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. In particular, there is no actual computation of $c_{v^s} = c_v \cdot s$ in step 3 (c) ii; at most $c_y$ is ever explicitly computed (in step 3 (c) i).

Step 1 has a cost of $O(C\lambda k (k + T) \|\Delta\|)$ by Proposition 4.8: this includes computing the left and right normal forms of $\tilde{x}$ and $\tilde{y}$. 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 4.3. 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 $O(C\lambda k \|\Delta\|)$. Step 3 (b) has a cost of $O(C\lambda^2 k \|\Delta\| (\|\Delta\| + RM))$ by Proposition 4.9. The body of the while loop in step 3 is executed $|SC(x)|$ times and the body of the for loop in step 3 (c) is executed at most $\lambda$ times. The actual computation of the conjugating element $c_1 \cdot c_y \cdot c_2^{-1}$ in step 3 (c) ii has a cost of $O(T + |SC(x)|)$, but is executed at most once.

Thus, the complexity of Algorithm 3 is

$$O\left(C\lambda k (k + T) \|\Delta\| \cdot \left|SC(x)\right| \cdot C\lambda^2 k \|\Delta\| (\|\Delta\| + RM)\right)$$

$$+ O\left(|SC(x)| \cdot C\lambda k \|\Delta\| (\|\Delta\| + RM)\right)$$

$$+ O\left(|SC(x)| \lambda \cdot C\lambda k \|\Delta\|\right)$$

$$+ O\left(T + |SC(x)|\right)$$

$$= O\left(C\lambda k \|\Delta\| \cdot (k + T + |SC(x)|\lambda(\|\Delta\| + RM))\right)$$

as claimed.

Remark 4.11. Unfortunately, the obvious bounds for $T$ and $M$ given in Remark 4.5 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 4.5. Moreover, no bound for $|SC(x)|$ is currently known which is better than the obvious bound $|SC(x)| \leq |SSS(x)| \leq (\|\Delta\|)^k$ (cf. Remark 4.5); 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 $SC(x)$ are rigid, then one can choose $R = \|\Delta\|$ by [11, Proposition 4.3 and Corollary 4.4], and obviously $M = 1$. However, even in this case, no realistic bound for $T$ is known.

The situation for $|SC(x)|$ is more complicated. It is shown in [2] that $|USS(x)|$ grows exponentially in $n$ for periodic elements of the Artin braid groups $B_n$. By [11, Proposition 5.1], the same is true
for $|\text{SC}(x)|$. Hence, a bound for $|\text{SC}(x)|$ 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. For the situation of Artin braid groups, an attempt to reduce the general case to the special case of rigid elements is sketched in [1].

The problem of finding bounds for $T$, $M$ and $R$ which are polynomial in $k$ and $\|\Delta\|$ and the problem of understanding $|\text{SC}(x)|$ correspond to open problems formulated in [1] in the context of Artin braid groups for ultra summit sets and the cycling and decycling operations.

References

[1] J. S. Birman, V. Gebhardt and J. González-Meneses, Conjugacy in Garside groups I: cycling, powers and rigidity, Groups Geom. Dyn. 1 (2007), 221-279.

[2] J. S. Birman, V. Gebhardt and J. González-Meneses, Conjugacy in Garside groups III: Periodic braids, J. Alg. 316 (2007), 746-776.

[3] J. S. Birman, K. Y. Ko and S. J. Lee, A new approach to the word and conjugacy problems in the braid groups, Adv. Math. 139 (1998), 322-353.

[4] R. Charney, Artin groups of finite type are biautomatic, Math. Ann. 292 (1992), 671-683.

[5] P. Dehornoy and L. Paris, Gaussian groups and Garside groups, two generalizations of Artin groups, Proc. London Math. Soc. 79 (1999), 569-604.

[6] P. Dehornoy, Groupes de Garside, Ann. Scient. Ec. Norm. Sup. 35 (2002), 267-306.

[7] E. ElRifai and H. Morton, Algorithms for positive braids, Quart. J. Math. Oxford Ser. (2), 45 (1994), 479-497.

[8] D. B. A. Epstein, J. W. Cannon, D. F. Holt, S. V. F. Levy, M. S. Patterson and W. P. Thurston, Word Processing in Groups, Jones and Bartlett Publishers, Boston 1992.

[9] N. Franco and J. González-Meneses, Conjugacy problem for braid groups and Garside groups, J. Alg. 266 (2003), 112-132.

[10] V. Gebhardt, A new approach to the conjugacy problem in Garside groups, J. Alg. 292 (2005), 282-302.

[11] V. Gebhardt and J. González-Meneses, The cyclic sliding operation in Garside groups, arXiv:0808.1430v2.

[12] E.-K. Lee and S. J. Lee, Abelian subgroups of Garside groups, Comm. Alg. 36 (2008), 1121-1139.

[13] J. Michel, Garside and braid monoids and groups. Chapter 81 of the GAP manual available at http://www.math.jussieu.fr/~jmichel/htm/CHAP081.htm

Volker Gebhardt:
School of Computing and Mathematics
University of Western Sydney
Locked Bag 1797
Penrith South DC NSW 1797, Australia
E-mail: v.gebhardt@uws.edu.au

Juan González-Meneses:
Dept. Algebra. Facultad de Matemáticas
Universidad de Sevilla
Apdo. 1160
41080 Sevilla (SPAIN)
E-mail: meneses@us.es
URL: www.personal.us.es/meneses

30