



Posets of twisted involutions in Coxeter groups

Contents

1	Coxeter groups	3
1.1	Introduction to Coxeter groups	3
1.2	Exchange and Deletion Condition	4
1.3	Finite Coxeter groups	6
1.4	Bruhat ordering	6
1.5	Compact hyperbolic Coxeter groups	10
2	Twisted involutions in Coxeter groups	11
2.1	Introduction to twisted involutions	11
2.2	Twisted weak ordering	15
2.3	Residuums	15
2.4	Twisted weak ordering algorithms	19
3	Main Thesis	25
A	Source codes	26
B	References	42

1 Coxeter groups

A Coxeter group, named after Harold Scott MacDonald Coxeter, is an abstract group generated by involutions with specific relations between these generators. A simple class of Coxeter groups are the symmetry groups of regular polyhedras in the Euclidean space.

The symmetry group of the square for example can be generated by two reflections s, t , whose stabilized hyperplanes enclose an angle of $\pi/4$. In this case the map st is a rotation in the plane by $\pi/2$. So we have $s^2 = t^2 = (st)^4 = \text{id}$. In fact, this reflection group is determined up to isomorphy by s, t and these three relations [4, Theorem 1.9]. Furthermore it turns out, that the finite reflection groups in the Euclidean space are precisely the finite Coxeter groups [4, Theorem 6.4].

In this chapter we compile some basic facts on Coxeter groups, based on [4].

1.1 Introduction to Coxeter groups

Definition 1.1. Let $S = \{s_1, \dots, s_n\}$ be a finite set of symbols and

$$R = \{m_{ij} \in \mathbb{N} \cup \infty : 1 \leq i, j \leq n\}$$

a set numbers (or ∞) with $m_{ii} = 1$, $m_{ij} > 1$ for $i \neq j$ and $m_{ij} = m_{ji}$. Then the free represented group

$$W = \langle S \mid (s_i s_j)^{m_{ij}} \rangle$$

is called a **Coxeter group** and (W, S) the corresponding **Coxeter system**. The cardinality of S is called the **rank** of the Coxeter system (and the Coxeter group).

From the definition we see, that Coxeter groups only depend on the cardinality of S and the relations between the generators in S . A common way to visualize this information are Coxeter graphs.

Definition 1.2. Let (W, S) be a Coxeter system. Create a graph by adding a vertex for each generator in S . Let $(s_i s_j)^m = 1$. In case $m = 2$ the two corresponding vertices have no connecting edge. In case $m = 3$ they are connected by an unlabeled edge. For $m > 3$ they have an connecting edge with label m . We call this graph the **Coxeter graph** of our Coxeter system (W, S) .

Definition 1.3. Let (W, S) be a Coxeter system. For an arbitrary element $w \in W$ we call a product $s_{i_1} \cdots s_{i_n} = w$ of generators $s_{i_1} \dots s_{i_n} \in S$ an **expression** of w . Any expression that can be obtained from $s_{i_1} \cdots s_{i_n}$ by omitting some (or all) factors, is called a **subexpression** of w .

The present relations between the generators of a Coxeter group allow us to rewrite expressions. Hence an element $w \in W$ can have more than one expression. Obviously any element $w \in W$ has infinitely many expressions, since any expression $s_{i_1} \cdots s_{i_n} = w$ can be extended by applying $s_i^2 = 1$ from the right. But there must be a smallest number of generators needed to receive w . For example the neutral element e can be expressed by the empty expression. Or each generator $s_i \in S$ can be expressed by itself, but any expression with less factors (i.e. the empty expression) is unequal to s_i .

Definition 1.4. Let (W, S) be a Coxeter system and $w \in W$ an element. Then there are some (not necessarily distinct) generators $s_i \in S$ with $s_1 \cdots s_r = w$. We call r the **expression length**. The smallest number $r \in \mathbb{N}_0$ for that w has an expression of length r is called the **length** of w and each expression of w , that is of minimal length, is called **reduced expression**. The map

$$l : W \rightarrow \mathbb{N}_0$$

that maps each element in W to its length is called **length function**.

Definition 1.5. Let (W, S) be a Coxeter system. We define

$$D_R(w) := \{s \in S : l(ws) < l(w)\}$$

as the **right descending set** of w . The analogue left version

$$D_L(w) := \{s \in S : l(sw) < l(w)\}$$

is called **left descending set** of w . Since the left descending set is not need in this paper, we will often call the right descending just **descending set** of w .

The next lemma yields some useful identities and relations for the length function.

Lemma 1.6. Let (W, S) be a Coxeter system, $s \in S$, $u, w \in W$ and $l : W \rightarrow \mathbb{N}$ the length function. Then

1. $l(w) = l(w^{-1})$,
2. $l(w) = 0$ iff $w = e$,
3. $l(w) = 1$ iff $w \in S$,
4. $l(uw) \leq l(u) + l(w)$,
5. $l(uw) \geq l(u) - l(w)$ and
6. $l(ws) = l(w) \pm 1$.

Proof. See [4, Section 5.2]. □

1.2 Exchange and Deletion Condition

We now obtain a way to get a reduced expression of an arbitrary element $s_1 \cdots s_r = w \in W$.

Definition 1.7. Let (W, S) be a Coxeter system. Any element $w \in W$ that is conjugated to an generator $s \in S$ is called **reflection**. Hence the set of all reflections in W is

$$T = \bigcup_{w \in W} wSw^{-1}.$$

Theorem 1.8 (Strong Exchange Condition). Let (W, S) be a Coxeter system, $w \in W$ an arbitrary element and $s_1 \cdots s_r = w$ with $s_i \in S$ a not necessarily reduced expression for w . For each reflection $t \in T$ with $l(wt) < l(w)$ there exists an index i for which $wt = s_1 \cdots \hat{s}_i \cdots s_r$, where \hat{s}_i means omission. In case we start from a reduced expression, then i is unique.

Proof. See [4, Theorem 5.8]. □

The Strong Exchange Condition can be weakened when insisting on $t \in S$ to receive the following corollary.

Corollary 1.9 (Exchange Condition). *Let (W, S) be a Coxeter system, $w \in W$ an arbitrary element and $s_1 \cdots s_r = w$ with $s_i \in S$ a not necessarily reduced expression for w . For each generator $s \in S$ with $l(ws) < l(w)$ there exists an index i for which $ws = s_1 \cdots \hat{s}_i \cdots s_r$, where \hat{s}_i means omission.*

Proof. Directly from Strong Exchange Condition. □

The Exchange Condition immediately yields another corollary for Coxeter groups:

Corollary 1.10 (Deletion Condition). *Let (W, S) be a Coxeter system, $w \in W$ and $w = s_1 \cdots s_r$ with $s_i \in S$ an unreduced expression of w . Then there exist two indices $i, j \in \{1, \dots, r\}$ with $i < j$, such that $w = s_1 \cdots \hat{s}_i \cdots \hat{s}_j \cdots s_r$, where \hat{s}_i and \hat{s}_j mean omission.*

Proof. Since the expression is unreduced there must be an index j for that the twisted length shrinks. That means for $w' = s_1 \cdots s_{j-1}$ is $l(w's_j) < l(w')$. Using the Exchange Condition we get $w's_j = s_1 \cdots \hat{s}_i \cdots s_{j-1}$ yielding $w = s_1 \cdots \hat{s}_i \cdots \hat{s}_j \cdots s_r$. □

This corollary is called **Deletion Condition** and allows us to reduce expressions, i.e. to find a subexpression that is reduced. Due to the Deletion Condition any unreduced expression can be reduced by omitting an even number of generators (we just have to apply the Deletion Condition inductively).

The Strong Exchange Condition, the Exchange Condition and the Deletion Condition, are some of the most powerful tools when investigating properties of Coxeter groups. We can use the second to prove a very handy property of Coxeter groups. The intersection of two parabolic subgroups is again a parabolic subgroup.

Definition 1.11. Let (W, S) be a Coxeter system. For a subset of generators $I \subset S$ we call the subgroup $W_I \leq W$, that is generated by the elements in I with the corresponding relations, a **parabolic subgroup** of W .

Lemma 1.12. Let (W, S) be a Coxeter system and $I, J \subset S$ two subsets of generators. Then $W_I \cap W_J = W_{I \cap J}$.

Proof. Let $w \in W_{I \cap J}$. Then $w \in W_I$ and $w \in W_J$. To show the other inclusion we induce over the length r . For $r = 0$ we have $w = e$ and so $w \in W_{S'}$ for any $S' \subset S$. Suppose we have proven the assumption for all lengths up to $r - 1$. Let $w \in W_I \cap W_J$ with $l(w) = r$. Then we have two reduced expressions $w = s_1 \cdots s_r = t_1 \cdots t_r$ with $s_i \in I$ and $t_i \in J$. By applying s_r from the right we get $ws_r = s_1 \cdots s_{r-1} = t_1 \cdots t_r s_r$. The expression $t_1 \cdots t_r s_r$ is of length $r - 1$, so Exchange Condition yields $ws_r = s_1 \cdots s_{r-1} = t_1 \cdots \hat{t}_i \cdots t_r$, hence $ws_r \in W_I \cap W_J$. Due to induction we know that $ws_r \in W_{I \cap J}$. **TODO** □

1.3 Finite Coxeter groups

Coxeter groups can be finite and infinite. A simple example for the former category is the following. Let $S = \{s\}$. Due to definition it must be $s^2 = e$. So W is isomorph to \mathbb{Z}_2 and finite. An example for an infinite Coxeter group can be obtained from $S = \{s, t\}$ with $s^2 = t^2 = e$ and $(st)^\infty = e$ (so we have no relation between s and t). Obviously the element st has infinite order forcing W to be infinite. But there are infinite Coxeter groups without an ∞ -relation between two generators, as well. An example for this is W obtained from $S = \{s_1, s_2, s_3\}$ with $s_1^2 = s_2^2 = s_3^2 = (s_1 s_2)^3 = (s_2 s_3)^3 = (s_3 s_1)^3 = e$. But how can one decide weather W is finite or not?

To provide a general answer to this question we fallback to a certain class of Coxeter groups, the irreducible ones.

Definition 1.13. A Coxeter system is called **irreducible**, if the corresponding Coxeter graph is connected. Else, it is called **reducible**.

If a Coxeter system is reducible, then its graph has more than one connection component and each connection component corosponds to a parabolic subgroup of W .

Proposition 1.14. Let (W, S) be a reducible Coxeter system. Then there exists a partition of S into I, J with $(s_i s_j)^2 = e$ whenever $s_i \in I, s_j \in J$ and W is isomorph to the direct product of the two parabolic subgroups W_I and W_J .

Proof. See [4, Proposition 6.1]. □

This proposition tells us, that an arbitray Coxeter system is finite iff its irreducible parabolic subgroups are finite. Therefore we can indeed fallback to irreducible Coxeter systems without loss of generality. If we could categorize all irreducible finite Coxeter systems, we could categorize all finite Coxeter systems. This is done by the following theorem:

Theorem 1.15. The irreducible finite Coxeter systems are exactly the ones in Figure 1.1.

Proof. [4, Theorem 6.4] □

Finally we can decide with ease, if a given Coxeter system is finite. Take its irreducible parabolic subgroups and check, if each is one of $A_n, B_n, D_n, E_6, E_7, E_8, F_4, H_3, H_4$ or $I_2(m)$.

1.4 Bruhat ordering

We now investigate ways to partially order the elements of a Coxeter group. Futhermore, this ordering should be compatible with the length function. The most useful way to achieve this is the Bruhat ordering [4, Section 5.9].

Definition 1.16. Let M be a set. A binary relation, in this case often denoted as “ \leq ”, is called a **partial order** over M , if fullfills the following conditions for all $a, b, c \in M$:

1. $a \leq a$, called **reflexivity**

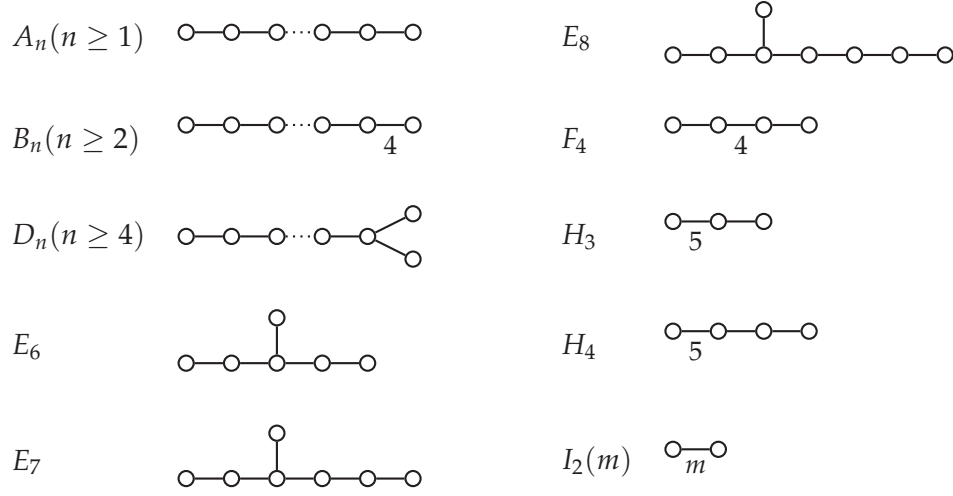


Figure 1.1: All types of irreducible finite Coxeter systems

2. if $a \leq b$ and $b \leq a$ then $a = b$, called **antisymmetry**
3. if $a \leq b$ and $b \leq c$ then $a \leq c$, called **transitivity**

In this case (M, \leq) is called a **poset**. If two elements $a \leq b \in M$ are immediate neighbours, i.e. there is no third element $c \in M$ with $a \leq c \leq b$ we say that b **covers** a . A poset is called **graded poset** if there is a map $\rho : M \rightarrow \mathbb{N}$ so that $\rho(b) - 1 = \rho(a)$ whenever b covers a . In this case ρ is called the **rank function** of the graded poset.

Definition 1.17. Let (M, \leq) be a poset. The **Hasse diagram** of the poset is the graph obtained in the following way: Add a vertex for each element in M . Then add a directed edge from vertex a to b whenever b covers a .

Example 1.18. Suppose we have an arbitrary set M . Then the powerset $\mathcal{P}(M)$ can be partially ordered by the subset relation, so $(\mathcal{P}(M), \subseteq)$ is a poset. Indeed this poset is always graded with the cardinality function as rank function. In Figure 1.2 we see the Hasse diagram of this poset with $M = \{x, y, z\}$.

Definition 1.19. Let (W, S) be a Coxeter system and $T = \cup_{w \in W} wSw^{-1}$ the set of all reflections in W . We write $w' \rightarrow w$ if there is a $t \in T$ with $w't = w$ and $l(w') < l(w)$. If there is a sequence $w' = w_0 \rightarrow w_1 \rightarrow \dots \rightarrow w_m = w$ we say $w' < w$. The resulting relation $w' \leq w$ is called **Bruhat ordering**, denoted as $\text{Br}(W)$.

Lemma 1.20. Let (W, S) be a Coxeter system. Then $\text{Br}(W)$ is a poset.

Proof. The Bruhat ordering is reflexive by definition. Since the elements in sequences $e \rightarrow w_1 \rightarrow w_2 \rightarrow \dots$ are strictly ascending in length, it must be antisymmetric. By concatenation of sequences we get the transitivity. \square

What we really want is the Bruhat ordering to be graded with the length function as rank function. By definition we already have $v < w$ iff $l(v) < l(w)$, but its not that obvious



Figure 1.2: Hasse diagram of the set of all subsets of $\{x, y, z\}$ order by the subset relation

that two immediately adjacent elements differ in length by exactly 1. Beforehand let us just mention two other partial orderings, where this property is obvious by definition:

Definition 1.21. Let (W, S) be a Coxeter system. The ordering \leq_R defined by $u \leq_R w$ iff $uv = w$ for some $v \in W$ with $l(u) + l(v) = l(w)$ is called the **right weak ordering**. The left-sided version $u \leq_L w$ iff $vu = w$ is called the **left weak ordering**.

To ensure the Bruhat ordering is graded as well, we need another characterization of the Bruhat ordering in terms of subexpressions.

Proposition 1.22. Let (W, S) be a Coxeter system, $u, w \in W$ with $u \leq w$ and $s \in S$. Then $us \leq w$ or $us \leq ws$ or both.

Proof. We can reduce the proof (TODO why?) to the case $u \rightarrow w$, i.e. $ut = w$ for a $t \in T$ with $l(v) < l(u)$. Let $s = t$. Then $us \leq w$ and we are done. In case $s \neq t$ there are two alternatives for the lengths. We can have $l(us) = l(u) - 1$ which would mean $us \rightarrow u \rightarrow w$, so $us \leq w$.

Assume $l(us) = l(u) + 1$. For the reflection $t' = sts$ we get $(us)t' = ussts = uts = ws$. So it is $us \leq ws$ iff $l(us) < l(ws)$. Suppose this is not the case. Since we have assumed $l(us) = l(u) + 1$ any reduced expression $u = s_1 \cdots s_r$ for u yields a reduced expression $us = s_1 \cdots s_r s$ for us . With the Strong Exchange Condition we can obtain $ws = ust'$ from us by omitting one factor. This omitted factor cannot be s since $s \neq t$. This means $ws = s_1 \cdots \hat{s}_i \cdots s_r s$ and so $ws = s_1 \cdots \hat{s}_i \cdots s_r$, contradicting to our assumption $l(u) < l(w)$ \square

Theorem 1.23. Let (W, S) be a Coxeter system and $w \in W$ with any reduced expression $w = s_1 \cdots s_r$ and $s_i \in S$. Then $u \leq w$ (in the Bruhat ordering) iff u can be obtained as a subexpression of this reduced expression.

Proof. TODO \square

This characterization of the Bruhat ordering is very handy. With it and the following short lemma we will be in the position to show, that $\text{Br}(W)$ is graded with rank function l .

Lemma 1.24. *Let (W, S) be a Coxeter system, $u, w \in W$ with $u < w$ and $l(w) = l(u) + 1$. In case there is a generator $s \in S$ with $u < us$ but $us \neq w$, then both $w < ws$ and $us < ws$.*

Proof. Due to Proposition 1.22 we have $us \leq w$ or $us \leq ws$. Since $l(us) = l(w)$ and $us \neq w$ the first case is impossible. So $us \leq ws$ and because of $u \neq w$ already $us < ws$. In turn, $l(w) = l(us) < l(ws)$, forcing $w < ws$. \square

Proposition 1.25. *Let (W, S) be a Coxeter system and $u < w$. Then there are elements $w_0, \dots, w_m \in W$ such that $u = w_0 < w_1 < \dots < w_m = w$ with $l(w_i) = l(w_{i-1}) + 1$ for $1 \leq i \leq m$.*

Proof. We induce on $r = l(u) + l(w)$. In case $r = 1$ we have $u = e$ and $w = s$ for an $s \in S$ and are done. Conversely suppose $r > 1$. Then there is a reduced expression $w = s_1 \cdots s_r$ for w . Lets fix this expression. Then $l(ws_r) < l(w)$. Thanks to Theorem 1.23 there must be a subexpression of w with $u = s_{i_1} \cdots s_{i_q}$ for some $i_1 < \dots < i_q$. We distinguish between two cases:

$u < us$ If $i_q = r$, then $us = s_{i_1} \cdots s_{i_q} s = s_{i_1} \cdots s_{i_{q-1}}$ which is also a subexpression of ws . This yields $u < us \leq ws < w$. Since $l(ws) < r$ there is, by induction, a sequence of the desired form. The last step from ws to w also differs in length by exactly 1, so we are done. If $i_q < r$ then u is itself already a subexpression of ws and we can again find a sequence from u to ws strictly ascending length by 1 in each step and have one last step from ws to w also increasing length by 1.

$us < u$ Then by induction we can find a sequence from us to w , say $us = w_0 < \dots < w_m = w$, where the lengths of neighboured elements differ by exactly 1. Since $w_0 s = u > us = w_0$ and $w_m s = ws < w = w_m$ there must be a smallest index $i \geq 1$, such that $w_i s < w_i$, which we choose. Suppose $w_i \neq w_{i-1} s$. It is $w_{i-1} < w_{i-1} s \neq w_i$ and due to Lemma 1.24 we get $w_i < w_i s$. This contradicts to the minimality of i . So $w_i = w_{i-1} s$. For all $1 \leq j < i$ we have $w_j \neq w_{j-1} s$, because of $w_j < w_j s$. Again we apply Lemma 1.24 to receive $w_{j-1} s < w_j s$. Alltogether we can construct a sequence

$$u = w_0 s < w_1 s < \dots < w_{i-1} s = w_i < w_{i+1} < \dots < w_m = w,$$

which matches our assumption. \square

Corollary 1.26. *Let (W, S) be a Coxeter system and $\text{Br}(W)$ the Bruhat ordering poset of W . Then $\text{Br}(W)$ is graded with $l : W \rightarrow \mathbb{N}$ as rank function.*

Proof. Let $u, w \in W$ with w covering u . Then Proposition 1.25 says there is a sequence $u = w_0 < \dots < w_m = w$ with $l(w_i) = l(w_{i-1}) + 1$ for $1 \leq i \leq m$. Since w covers u it must be $m = 1$ and so $u < w$ with $l(w) = l(u) + 1$. \square

Theorem 1.27 (Lifting Property). *Let (W, S) be a Coxeter system and $v, w \in W$ with $v \leq w$. Suppose $s \in S$ with $s \in D_R(w)$. Then*

1. $vs \leq w$,
2. $s \in D_R(v) \Rightarrow vs \leq ws$.

Proof. We use the alternative subexpression characterization of the Bruhat ordering from Theorem 1.23.

1. Since $s \in D_R(w)$ there exists a reduced expression $w = s_1 \cdots s_r$ with $s_r = s$. Due to $v \leq w$ we can obtain v as a subexpression $v = s_{i_1} \cdots s_{i_q}$ from w . If $i_q = r$ then $vs = s_{i_1} \cdots s_{i_q} s = s_{i_1} \cdots s_{i_{q-1}}$ is also a subexpression of w . Else, if $i_q \neq r$ then v is a subexpression of $ws = s_1 \cdots s_{r-1} s$ and so vs is again a subexpression of $w = s_1 \cdots s_{r-1} s$. In both cases we get $vs \leq w$.
2. If we additionally assume $s \in D_R(v)$ then we can always find a reduced expression $w = s_1 \cdots s_r$ with $s_r = s$ having $u = s_{i_1} \cdots s_{i_q}$ as subexpression with $s_{i_q} = s$. This yields $vs = s_{i_1} \cdots s_{i_{q-1}} \leq s_1 \cdots s_{r-1} = ws$. \square

The Lifting Property seems quite innocent, but when trying to investigate facts around the Bruhat ordering it proves to be one of the key tools in many cases.

1.5 Compact hyperbolic Coxeter groups

TODO

2 Twisted involutions in Coxeter groups

In this section we focus on a certain subset of elements in Coxeter groups, the so called twisted involutions. From now on (and in the next sections) we fix some symbols to have always the same meaning (some definitions follow later):

S A set of generators.

s A generator in S .

W A Coxeter group with generators S .

u, v, w A element in the Coxeter group W .

m_{ij} The order of the element $(s_i s_j)$ with s_i the i -th generator of W .

(W, S) The Coxeter system obtained from W and S .

θ A Coxeter system automorphism of (W, S) with $\theta^2 = \text{id}$.

\mathcal{I}_θ The set of twisted involutions of W regarding θ .

\underline{S} A set of symbols, $\underline{S} = \{\underline{s} : s \in S\}$.

2.1 Introduction to twisted involutions

Definition 2.1. An automorphism $\theta : W \rightarrow W$ with $\theta(S) = S$ is called a **Coxeter system automorphism** of (W, S) . We always assume $\theta^2 = \text{id}$.

Definition 2.2. Each $w \in W$ with $\theta(w) = w^{-1}$ is called a θ -**twisted involution** or just **twisted involution**, if θ is clear from the context. The set of all twisted involutions in W regarding θ is denoted with $\mathcal{I}_\theta(W)$. Often we just omit the Coxeter group and write \mathcal{I}_θ , when it is clear from the context which W is meant.

Example 2.3. Let $\theta = \text{id}_W$. Then θ is an Coxeter system automorphism and

$$\mathcal{I}_\theta = \{w \in W : w = w^{-1}\}.$$

The next example is more helpfull, since it reveals a way to think of \mathcal{I}_θ as a generalization of ordinary Coxeter groups.

Example 2.4. Let θ be a automorphism of $W \times W$ with

$$\theta : W \times W \rightarrow W \times W : (u, w) \mapsto (w, u).$$

Note that θ is no Coxeter system automorphism, but we can think of it as one if we identify $S \subset W$ with $S \times S \subset W \times W$. Then the set of twisted involutions is

$$\mathcal{I}_\theta = \{(w, w^{-1}) \in W \times W : w \in W\}.$$

This yields a canonical bijection between \mathcal{I}_θ and W .

The map we define right now is of superior importance to this whole paper, since it is needed to define the poset, the main thesis is about.

Definition 2.5. Let $\underline{S} := \{\underline{s} : s \in S\}$ be a set of symbols. Each element in \underline{S} acts from the right on W by the following definition:

$$w\underline{s} = \begin{cases} ws & \text{if } \theta(s)ws = w, \\ \theta(s)ws & \text{else.} \end{cases}$$

This action can be extended on the whole free monoid over \underline{S} by

$$w\underline{s}_1\underline{s}_2 \dots \underline{s}_k = (\dots ((w\underline{s}_1)\underline{s}_2) \dots)\underline{s}_k.$$

If $w\underline{s} = \theta(s)ws$, then we say \underline{s} **acts by twisted conjugation** on w . Else we say \underline{s} **acts by multiplication** on w .

Definition 2.6. Let $k \in \mathbb{N}$ and $s_{i_j} \in S$ for all $1 \leq j \leq k$. Then an expression $w\underline{s}_{i_1} \dots \underline{s}_{i_k}$ is called **twisted w -expression**. In case $w = e$ we omit w , just write $\underline{s}_{i_1} \dots \underline{s}_{i_k}$ and call it **twisted expression**.

Lemma 2.7. Let $w \in \mathcal{I}_\theta$ and $s \in S$. Then

$$w\underline{s} = \begin{cases} ws & \text{if } l(\theta(s)ws) = l(w), \\ \theta(s)ws & \text{else.} \end{cases}$$

Proof. Suppose \underline{s} acts by multiplication on w . Then $\theta(s)ws = w$ and so $l(\theta(s)ws) = l(w)$. So let the other way around $l(\theta(s)ws) = l(w)$. **TODO** □

Lemma 2.8. It is $l(ws) < l(w)$ iff $l(w\underline{s}) < l(w)$.

Proof. Suppose \underline{s} acts by multiplication on w . Then $w\underline{s} = ws$ and there is nothing to prove. So suppose \underline{s} acts by twisted conjugation on w . If $l(ws) < l(w)$, then Lemma 1.6 yields $l(ws) + 1 = l(w)$. Assuming $l(w\underline{s}) = l(\theta(s)ws) = l(w)$ would imply, that \underline{s} acts by multiplication on w due to Lemma 2.7, which is a contradiction. So $l(w\underline{s}) = l(\theta(s)ws) < l(w)$. The other way around suppose $l(w\underline{s}) < l(w)$. Then Lemma 1.6 says $l(w\underline{s}) = l(\theta(s)ws) = l(w) - 2$ and so $l(ws) = l(w) - 1$. □

Lemma 2.9. For all $w \in W$ and $s \in S$ it is $w\underline{ss} = w$.

Proof. For $w\underline{s}$ there are two cases. Suppose \underline{s} acts by multiplication on w , i.e. $\theta(s)ws = w$. For $w\underline{ss}$ there are again two possible options:

$$w\underline{ss} = \begin{cases} wss = w & \text{if } \theta(s)wss = ws, \\ \theta(s)wss = ws & \text{else.} \end{cases}$$

The second option contradicts itself.

Now suppose \underline{s} acts by twisted conjugation on w . This means $\theta(s)ws \neq w$ and for $(\theta(s)ws)\underline{s}$ there are again two possible options:

$$(\theta(s)ws)\underline{s} = \begin{cases} \theta(s)wss = \theta(s)w & \text{if } \theta(s)\theta(s)wss = \theta(s)ws, \\ \theta(s)\theta(s)wss = w & \text{else.} \end{cases}$$

The first option is impossible since $\theta(s)\theta(s)wss = w$ and we have assumed $\theta(s)ws \neq w$. Hence the only possible cases yield $wss = w$. \square

Remark 2.10. This lemma allows us to rewrite equations of twisted expressions. For example

$$u = w\underline{s} \iff u\underline{s} = wss = w.$$

This can be iterated to get

$$u = w\underline{s}_1 \dots \underline{s}_k \iff u\underline{s}_k \dots \underline{s}_1 = w.$$

Lemma 2.11. For all $\theta, w \in W$ and $s \in S$ it is $w \in \mathcal{I}_\theta$ iff $w\underline{s} \in \mathcal{I}_\theta$.

Proof. Let $w \in \mathcal{I}_\theta$. For $w\underline{s}$ there are two cases. Suppose \underline{s} acts by multiplication on w . Then we get

$$\theta(ws) = \theta(\theta(s)wss) = \theta^2(s)\theta(w) = sw^{-1} = (ws^{-1})^{-1} = (ws)^{-1}.$$

Suppose \underline{s} acts by twisted conjugation on w . Then we get

$$\theta(\theta(s)ws) = \theta^2(s)\theta(w)\theta(s) = sw^{-1}\theta(s) = (\theta^{-1}(s)ws^{-1})^{-1} = (\theta(s)ws)^{-1}.$$

In both cases $w\underline{s} \in \mathcal{I}_\theta$.

Now let $w\underline{s} \in \mathcal{I}_\theta$. Suppose \underline{s} acts by multiplication on w . Then

$$\theta(w) = \theta(\theta(s)ws) = \theta^2(s)\theta(ws) = s(ws)^{-1} = ss^{-1}w^{-1} = w^{-1}.$$

Suppose \underline{s} acts by twisted conjugation on w . Then

$$\begin{aligned} \theta(w) &= \theta(\theta(s)\theta(s)wss) = \theta^2(s)\theta(\theta(s)ws)\theta(s) \\ &= s(\theta(s)ws)^{-1}\theta(s) = s(s^{-1}w^{-1}\theta(s^{-1})\theta(s)) = w^{-1}. \end{aligned}$$

In both cases $w \in \mathcal{I}_\theta$. \square

A remarkable property of the action from Definition 2.5 is its e -orbit. As the following lemma shows, it coincides with \mathcal{I}_θ .

Lemma 2.12. Fix θ . Then the set of twisted involutions regarding θ coincides with the set of all twisted expressions regarding θ .

Proof. As already seen in Lemma 2.11, each twisted expression is in \mathcal{I}_θ , since $e \in \mathcal{I}_\theta$. So let $w \in \mathcal{I}_\theta$. If $l(w) = 0$, then $w = e \in \mathcal{I}_\theta$. Induce on the length of w and let $l(w) = r > 0$. Suppose w has a twisted expression ending with \underline{s} . Then w also has a reduced expression (in S) ending with s and so $l(ws) < l(w)$. With Lemma 2.8 we get $l(w\underline{s}) < l(w)$. By induction $w\underline{s}$ has a twisted expression and hence $w = (w\underline{s})\underline{s}$ has one, too. \square

In the same way, we can use regular expressions to define the length of an element $w \in W$, we can use the twisted expressions to define the twisted length of an element $w \in \mathcal{I}_\theta$.

Definition 2.13. Let \mathcal{I}_θ be the set of twisted involutions. Then we define $\rho(w)$ as the smallest $k \in \mathbb{N}$ for that a twisted expression $w = \underline{s}_1 \dots \underline{s}_k$ exists. This is called the **twisted length** of w .

Lemma 2.14. The Bruhat ordering, restricted to the set of twisted involutions \mathcal{I}_θ , is a graded poset with ρ as rank function. We denote this poset with $\text{Br}(\mathcal{I}_\theta)$.

Proof. See [2, Theorem 4.8]. □

We now establish many properties from Section 1 for twisted expressions and $\text{Br}(\mathcal{I}_\theta)$. As seen in Example 2.4 it is $\text{Br}(W) \cong \text{Br}(\mathcal{I}_\theta)$. So the hope, that many properties can be transfered, is eligible.

Lemma 2.15. Let $w \in \mathcal{I}_\theta$ and $s \in S$. Then $\rho(w\underline{s}) = \rho(w) \pm 1$. In fact it is $\rho(w\underline{s}) = \rho(w) - 1$ iff $s \in D_R(w)$.

Proof. Since $\text{Br}(\mathcal{I}_\theta)$ is graded with rank function ρ and either $w\underline{s}$ covers w or w covers $w\underline{s}$ it is $\rho(w\underline{s}) = \rho(w) \pm 1$. Now suppose $w\underline{s} < w$. Then $l(w\underline{s}) < l(w)$ and with Lemma 2.8 we have $l(ws) < l(w)$ yielding $s \in D_R(w)$. The other way around suppose $w\underline{s} > w$. Then $l(w\underline{s}) > l(w)$ and again with Lemma 2.8 we have $l(ws) > l(w)$ yielding $s \notin D_R(w)$. □

Lemma 2.16 (Lifting property for \underline{S}). Let $v, w \in W$ with $v \leq w$. Suppose $s \in S$ with $s \in D_R(w)$. Then

1. $v\underline{s} \leq w$,
2. $s \in D_R(v) \Rightarrow v\underline{s} \leq w\underline{s}$.

Proof. Whenever a relation comes from the ordinary Lifting Property, we denote it with $<_{LP}$ in this proof.

$v\underline{s} = v\underline{s} \wedge w\underline{s} = w\underline{s}$ Same situation as in Lifting Property.

$v\underline{s} = v\underline{s} \wedge w\underline{s} = \theta(s)ws$ The first part $v\underline{s} = v\underline{s} \leq_{LP} w$ is immediate. Suppose $s \in D_R(v)$.

Then $v\underline{s} \leq_{LP} w\underline{s} \Rightarrow v = \theta(s)v\underline{s} \leq w\underline{s} \Rightarrow v\underline{s} = v\underline{s} \leq \theta(s)ws = w\underline{s}$.

$v\underline{s} = \theta(s)v\underline{s} \wedge w\underline{s} = w\underline{s}$ **TODO**

$v\underline{s} = \theta(s)v\underline{s} \wedge w\underline{s} = \theta(s)ws$ **TODO** □

Proposition 2.17 (Exchange property for twisted expressions). **TODO**

Proposition 2.18 (Deletion property for twisted expressions). **TODO**

TODO

2.2 Twisted weak ordering

In this section we introduce the twisted weak ordering $Wk(\theta)$ on the set \mathcal{I}_θ of θ -twisted involutions.

Definition 2.19. Let (W, S) be a Coxeter system and let \mathcal{I}_θ be the set of θ -twisted involutions in W . For $v, w \in \mathcal{I}_\theta$ we define $v \preceq w$ iff there are $\underline{s}_1, \dots, \underline{s}_k \in \underline{S}$ with $w = v\underline{s}_1 \dots \underline{s}_k$ and $\rho(v) = \rho(w) - k$. We call the poset $(\mathcal{I}_\theta, \preceq)$ **twisted weak ordering**, denoted with $Wk(W, \theta)$. When the Coxeter group W is clear from the context, we just write $Wk(\theta)$.

Lemma 2.20. *The poset $Wk(\theta)$ is a graded poset with rank function ρ .*

Proof. Follows immediately from the definition of \preceq . □

Example 2.21. In Figure 2.1 we see Hasse diagram of $Wk(A_4, \text{id})$. Solid edges represent twisted conjugations and dashed edges represent multiplications.

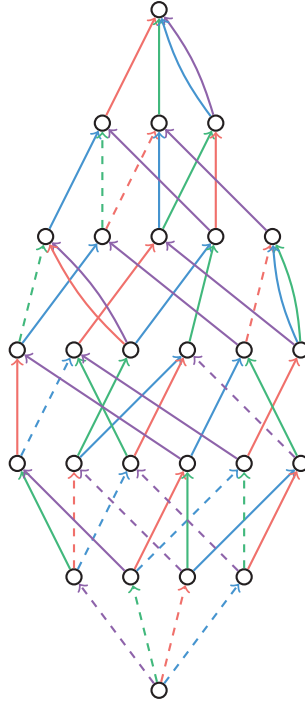


Figure 2.1: Hasse diagram of $Wk(A_4, \text{id})$

Definition 2.22. Let $v, w \in W$ with $\rho(w) - \rho(v) = n$. A sequence $v = w_0 \prec w_1 \prec \dots \prec w_n = w$ is called a **geodesic** from v to w .

TODO

2.3 Residuums

Definition 2.23. Let $w \in W$ and $I \subseteq S$ be a subset of generators. Then we define

$$wC_I := \{w\underline{s}_1 \dots \underline{s}_k : k \in \mathbb{N}_0, s_i \in I\}$$

as the *I-residuum* of w or just *residuum*. To emphasize the size of I , say $|I| = n$, we also speak of a *rank- n -residuum*.

Example 2.24. Let $w \in W$. Then $wC_\emptyset = \{w\}$ and $wC_S = \mathcal{I}_\theta$.

Lemma 2.25. Let $w \in W$ and $I \subset S$. If $v \in wC_I$, then $vC_I = wC_I$.

Proof. Suppose $v \in wC_I$. Then $v = w\underline{s}_1 \dots \underline{s}_n$ for some $s_i \in I$. Suppose $u = w\underline{t}_1 \dots \underline{t}_m \in wC_I$ is any other element in wC_I with $t_i \in I$. Then

$$u = w\underline{t}_1 \dots \underline{t}_m = (v\underline{s}_n \dots \underline{s}_1)\underline{t}_1 \dots \underline{t}_m$$

and so $u \in vC_I$. This yields $wC_I \subset vC_I$. Since $w \in vC_I$ we can swap v and w to get the other inclusion. \square

Corollary 2.26. Let $v, w \in W$ and $I \subset S$. Then either $vC_I \cap wC_I = \emptyset$ or $vC_I = wC_I$.

Proof. Immediately follows from Lemma 2.25. \square

Proposition 2.27. Let $w \in \mathcal{I}_\theta$, $I \subseteq S$ be a set of generators. Then there exists a unique element $w_0 \in wC_I$ with $w_0 \preceq w_0\underline{s}$ for all $s \in I$.

Proof. Suppose there is no such element. Then for each $w \in wC_I$ we can find a $s \in I$ with $w' = w\underline{s} \preceq w$ and $w' \in wC_I$. By repetition of Deletion property for twisted expressions we get, that $e \in wC_I$, but e has the property, which we assumed, that no element in wC_I has. Hence there must be at least one such element. Now suppose there are two distinct elements u, v with the desired property. Note that this means, that u and w have no reduced twisted expression ending with some $\underline{s} \in I$. Let v have a reduced twisted expression $v = \underline{s}_1 \dots \underline{s}_k$. Since u and v are both in wC_I there must be a twisted v -expression for u

$$u = v\underline{s}_{k+1} \dots \underline{s}_{k+l} = \underline{s}_1 \dots \underline{s}_{k+l}$$

with $s_n \in I$ for $k+1 \leq n \leq k+l$. This twisted expression cannot be reduced, since it ends with $\underline{s}_{k+l} \in I$. Then Deletion property for twisted expressions yields that this twisted expression contains a reduced twisted subexpression for u . It cannot end with \underline{s}_n for $k+1 \leq n \leq k+l$. Hence, it is a twisted subexpression of $\underline{s}_1 \dots \underline{s}_k = v$, too. So $u \leq v$ by Theorem 1.23. Because of symmetry it is also $v \leq u$ and so $u = v$, contradicting to our assumption $u \neq v$. \square

Corollary 2.28. Let $w \in \mathcal{I}_\theta$, $I \subseteq S$ be a set of generators and let $\rho_{\min} := \{\rho(v) : v \in wC_I\}$ be the minimal twisted length within the residuum wC_I . Then there is a unique element $w_{\min} \in wC_I$ with $\rho(w_{\min}) = \rho_{\min}$. We denote this element with $\min(w, I)$.

Proof. The minimal rank ρ_{\min} exists, since $\rho : \mathcal{I}_\theta \rightarrow \mathbb{N}_0$, \mathbb{N}_0 is well-ordered and $wC_I \neq \emptyset$. Suppose we have an element w_{\min} with $\rho(w_{\min}) = \rho_{\min}$. This means, that in particular all $w_{\min}\underline{s}$ with $s \in I$ must be of larger twisted length, i.e. $w_{\min} < w_{\min}\underline{s}$ for all $s \in I$. With Proposition 2.27 this element must be unique. \square

We proceed with some properties of rank-2-residuums. Our interest in these residuums stems from the fact, that their properties are needed later in Section 2.4 to construct an effective algorithm for calculating the twisted weak ordering, i.e. calculating the Hasse diagram of $Wk(\theta)$ for arbitrary Coxeter systems (W, S) and Coxeter system automorphisms θ .

Definition 2.29. Let $s, t \in S$ be two distinct generators. We define:

$$[\underline{st}]^n := \begin{cases} (\underline{st})^{\frac{n}{2}} & n \text{ even,} \\ (\underline{st})^{\frac{n-1}{2}} \underline{s} & n \text{ odd.} \end{cases}$$

This definition allows us rewrite rank-2-residuums. Suppose we have an element $w \in \mathcal{I}_\theta$ and two distinct generators $s, t \in S$. Thanks to Lemma 2.25 and Corollary 2.28 we can assume, that $w = \min(w, \{s, t\})$. Then

$$wC_{\{s,t\}} = \{w\} \cup \{w[\underline{st}]^n : n \in \mathbb{N}\} \cup \{w[\underline{ts}]^n : n \in \mathbb{N}\}.$$

This encourages the following definition.

Definition 2.30. Let $w \in \mathcal{I}_\theta$ and let $s, t \in S$ be two distinct generators. Suppose $w = \min(w, \{s, t\})$. Then we call $\{w[\underline{st}]^n : n \in \mathbb{N}\}$ the ***s-branch*** and $\{w[\underline{ts}]^n : n \in \mathbb{N}\}$ the ***t-branch*** of $wC_{\{s,t\}}$.

One question arises immediately: Are the *s*- and the *t*-branch disjoint? With the following propositions, corollaries and lemmas we will get a much better idea of the structure of rank-2-residuums and answer this question.

Proposition 2.31. Let $w \in W$ and let $s, t \in S$ be two distinct generators. Without loss of generality suppose $w = \min(w, \{s, t\})$. If there is a $v \in wC_{\{s,t\}}$ with $v\underline{s} \prec v$ and $v\underline{t} \prec v$, then it is the unique element with this property forcing $wC_{\{s,t\}}$ to consist of two geodesics from w to v intersecting only in these two elements. Else, the *s*- and *t*-branch are disjoint, strictly ascending in twisted length and of infinite size.

Proof. Suppose there is a v in the *s*-branch with $v\underline{s} \prec v$ and $v\underline{t} \prec v$, say $v = w[\underline{st}]^n$. Then $w[\underline{st}]^{m+1} \prec w[\underline{st}]^m$ for $n \leq m \leq 2n-1$ and $w[\underline{st}]^{2n} = w$. This is because assuming the opposite would yield an element $u \neq w$ with $u \prec u\underline{s}$ and $u \prec u\underline{t}$ contradicting to the uniqueness from Proposition 2.27. If no such v exists, then the *s*- and *t*-branch must be disjoint, strictly ascending in twisted length and so of infinite size. \square

Proposition 2.32. Let $w \in S$ and $s, t \in S$ be two distinct generators. If \underline{s} acts by multiplications on w and $w\underline{s} \prec w$, then either $w\underline{st} \prec w\underline{s}$ or $w\underline{t} \succ w$.

Proof. We have $\theta(s)ws = w$ and $s \in D_R(w)$. If $t \notin D_R(w)$, then we are done. In return suppose $t \in D_R(w)$. This means $w\underline{s} \leq w$ and $w\underline{t} \leq w$ and [3, Lemma 3.9] yields $w\underline{st} < w$ and $w\underline{ts} < w$. If $t \in D_R(w\underline{s})$, then we are done. Conservely suppose $t \notin D_R(w\underline{s})$. Then $t \in D_R(w\underline{st})$. Together with $w\underline{st} \leq w$ [3, Lemma 3.9(2)] says $(w\underline{st})\underline{t} \leq w\underline{t}$. Finally we get

$$ws = w\underline{s} = (w\underline{st})\underline{t} \leq w\underline{t} = wt.$$

Since $w_{\underline{s}}$ and $w_{\underline{t}}$ are of same twisted length they have to be equal and therefore $s = t$ which contradicts to our assumption of two distinct generators s and t . Now we have $w_{\underline{s}} < w_{\underline{t}}$ or $w_{\underline{t}} > w$ (in $\text{Br}(\mathcal{I}_\theta)$). Both cases can be transferred to the twisted weak ordering. If $w_{\underline{s}} < w_{\underline{t}}$, then $\rho(w_{\underline{s}}) = \rho(w_{\underline{t}}) - 1$ and $w_{\underline{s}} = w_{\underline{t}}$, yielding $w_{\underline{s}} \prec w_{\underline{t}}$. If $w_{\underline{t}} > w$ then $\rho(w_{\underline{t}}) = \rho(w) + 1$, yielding $w_{\underline{t}} \succ w$. \square

Corollary 2.33. *Let $w \in S$ and let $s, t \in S$ be two distinct generators. If w is neither the unique element in $wC_{\{s,t\}}$ of smallest twisted length, i.e. $\min(w, \{s, t\})$, nor the unique (but not necessarily existing) element of largest twisted length, then both \underline{s} and \underline{t} act by twisted conjugation on w .*

Proof. Follows immediately from Proposition 2.32. \square

Lemma 2.34. *Let $s, t \in S$ be two distinct generators and $w \in S$ with $w = \min(w, \{s, t\})$. Suppose $v \in wC_{\{s,t\}}$ with $v_{\underline{s}} \prec v$ and $v_{\underline{t}} \prec v$. Then the twisted conjugations and multiplications are distributed axisymmetrically or pointsymmetrically like in Figure 2.2.*

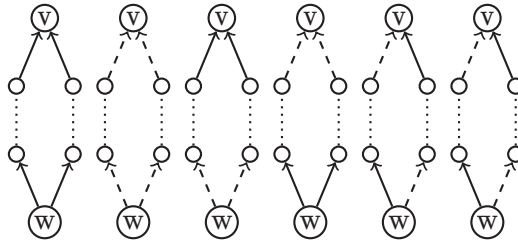


Figure 2.2: Possible distributions of twisted conjugations and multiplications in a rank-2-residuum

Proof. If u covers w , then there are only two edges and the assumption holds. So suppose $wC_{\{s,t\}}$ contains at least four edges. Due to Corollary 2.33 the actions by multiplication can only occur next to w and v . Hence there are $2^4 = 16$ configurations possible. There are two geodesics from w to v . Suppose they have a different count of actions by multiplication, say the first has n and the second m actions by multiplication, and $\rho(v) - \rho(w) = k$. Then

$$l(v) = l(w) + n + 2(k - n) \neq l(w) + m + 2(k - m) = l(v),$$

which is obviously a contradiction. This wipes out ten out of the 16 configurations. The remaining are those from Figure 2.2. \square

Lemma 2.35. *Let $w \in S$, $s, t \in S$ be two distinct generators and $m = \text{ord}(st) < \infty$. Then $|wC_{\{s,t\}}| \leq 2m$.*

Proof. **TODO** \square

Example 2.36. In Figure 2.3 we see two Hasse diagrams of $\text{Wk}(A_4, \text{id})$. The first only contains edges with label s_1, s_3 and the second only edges with label s_2, s_4 .

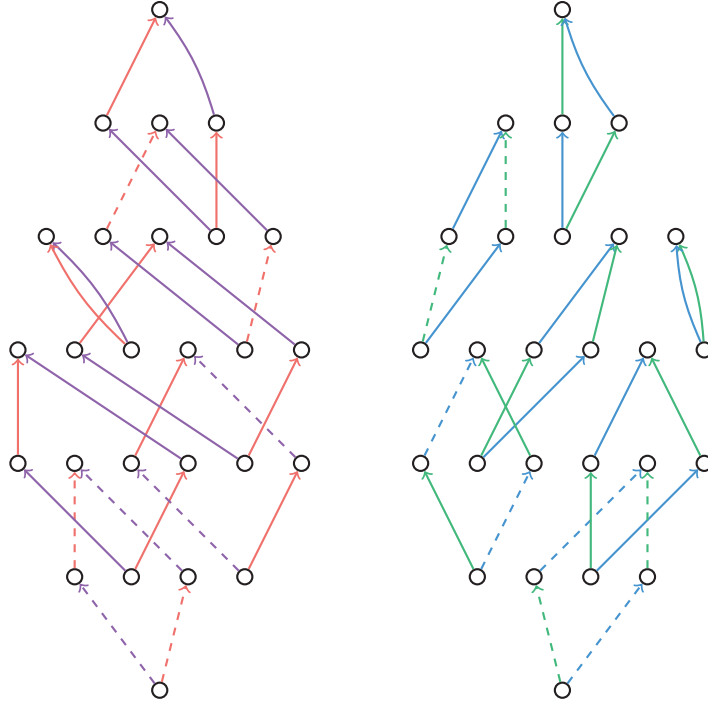


Figure 2.3: Hasse diagrams of $Wk(A_4, \text{id})$ with only s_1, s_3 edges on the left and only s_2, s_4 edges on the right side

2.4 Twisted weak ordering algorithms

Now we address the problem of calculating $Wk(\theta)$ for an arbitrary Coxeter group W , given in form of a set of generating symbols $S = \{s_1, \dots, s_n\}$ and the relations in form of $m_{ij} = \text{ord}(s_i s_j)$. From this input we want to calculate the Hasse diagram, i.e. the vertex set \mathcal{I}_θ and the edges labeled with \underline{s} . Thanks to Lemma 2.12 the vertex set can be received by walking the e -orbit of the action from Definition 2.5. The only element of twisted length 0 is e . Suppose we have already calculated the Hasse diagram until the twisted length k , i.e. we know all vertices $w \in \mathcal{I}_\theta$ with $\rho(w) \leq k$ and all edges connecting two vertices u, v with $\rho(u) + 1 = \rho(v) \leq k$. Let $\rho_k := \{w \in \mathcal{I}_\theta : \rho(w) = k\}$. Then all vertices in ρ_{k+1} are of the form $w\underline{s}$ for some $w \in \rho_k, s \in S$. For each $(w, s) \in \rho_k \times S$, we calculate $w\underline{s}$. If $\rho(w\underline{s}) = k + 1$ then $w \prec w\underline{s}$. To avoid having to check the twisted length we use Lemma 2.15. We already know the set $S_w \subseteq S$ of all generators yielding an edge into w . Due to the lemma it is $\rho(w\underline{s}) = k - 1$ for all $s \in S_w$ and $\rho(w\underline{s}) = k + 1$ for all $s \in S \setminus S_w$. Hence we only calculate $w\underline{s}$ for $s \in S \setminus S_w$ and know $w \prec w\underline{s}$ without checking the twisted length explicitly. The last problem to solve is the possibility of two different $(w, s), (v, t) \in \rho_k \times S$ with $w\underline{s} = v\underline{t}$. To deal with this, we have to compare a potential new twisted involution $w\underline{s}$ with each element of twisted length $k + 1$, already calculated. The concrete problem of comparing two elements in a free presented group, called **wordproblem for groups**, will not be addressed here. We suppose, that whatever computer system is used to implement our algorithm, supplies a suitable way to do that. The only thing to note is, that solving

the wordproblem is not a cheap operation. Reducing the count of element comparisons is a major demand to any algorithm, calculating $Wk(\theta)$.

The steps discussed have been compiled in to an algorithm by Haas [1, Algorithm 3.1.1]. We take this as our starting point. Since the runtime is far from being optimal, we use the structural properties of rank-2-residuums from Section 2.3 to improve the algorithm. As we will show, these optimizations yield an algorithm with an asymptotical perfect runtime behavior. TWOA1 shows the algorithm of Haas.

Algorithm 2.37 (TWOA1).

```

1: procedure TWISTEDWEAKORDERINGALGORITHM1( $(W, S), k_{max}$ )
2:    $V \leftarrow \{(e, 0)\}$ 
3:    $E \leftarrow \{\}$ 
4:   for  $k \leftarrow 0$  to  $k_{max}$  do
5:     for all  $(w, k_w) \in V$  with  $k_w = k$  do
6:       for all  $s \in S$  with  $\nexists(\cdot, w, s) \in E$  do ▷ Only for  $s \notin D_R(w)$ 
7:          $y \leftarrow ws$ 
8:          $z \leftarrow \theta(s)y$ 
9:         if  $z = w$  then
10:           $x \leftarrow y$ 
11:           $t \leftarrow s$ 
12:         else
13:           $x \leftarrow z$ 
14:           $t \leftarrow \underline{s}$ 
15:         end if
16:          $isNew \leftarrow \mathbf{true}$ 
17:         for all  $(w', k_{w'}) \in V$  with  $k_{w'} = k + 1$  do ▷ Check if  $x$  already known
18:           if  $x = w'$  then
19:              $isNew \leftarrow \mathbf{false}$ 
20:           end if
21:         end for
22:         if  $isNew = \mathbf{true}$  then
23:            $V \leftarrow V \cup \{(x, k + 1)\}$ 
24:         end if
25:          $E \leftarrow E \cup \{(w, x, t)\}$ 
26:       end for
27:     end for
28:      $k \leftarrow k + 1$ 
29:   end for
30:   return  $(V, E)$  ▷ The poset graph
31: end procedure

```

Note, that if W is finite, k_{max} does not have to be evaluated explicitly. When k reaches the maximal twisted length in $Wk(\theta)$, then the only vertex of twisted length k is the unique

element $w_0 \in W$ of maximal ordinary length. Since $s \in D_R(w_0)$ for all $s \in S$, there is no $s' \in S$ remaining to calculate $w_0 s'$ for. This condition can be checked to terminate the algorithm without knowing k_{max} before. When W is infinite, there is no maximal element and \mathcal{I}_θ is infinite, too. In this case k_{max} is used to terminate after having calculated a finite part of $Wk(\theta)$.

Lemma 2.38. *TWOA1 is a deterministic algorithm.*

Proof. The outer loop (line 4) is strictly ascending in $k \in \{0, \dots, k_{max}\}$ and so finite. The innermost loop (line 6) is finite since S is finite. The inner loop (line 5) is finite, since V starts as finite set and in each step there are added at most $|V| \cdot |S|$ many new vertices. Therefore the algorithm terminates. The soundness is due to the arguments at the beginning of Section 2.4. \square

Lemma 2.39. *Let $k \in \mathbb{N}$, $n = |\{w \in \mathcal{I}_\theta : \rho(w) \leq k\}|$ and for $0 \leq i \leq k$ let $\rho_i = |\{w \in \mathcal{I}_\theta : \rho(w) = i\}|$. Then $TWOA1 \in \mathcal{O}(n^2/k)$.*

Proof. Our algorithm has to do at least $\rho_i(\rho_i - 1)/2$ many element comparisons (line 17) for each $0 \leq i \leq k$. Set $m = \lfloor \frac{n}{k} \rfloor$. In the most optimistic case it is $\rho_i \geq m$ for all i . In practice the situation will be worse, since some ρ_i will be smaller than m (for example $\rho_0 = 1$) and so some ρ_i will be much larger than m . This optimistic case yields at least $m(m - 1)/2 \cdot k$ many element comparisons. Hence regarding the most delimiting operation, the element comparison, our algorithm is in $\Omega(m^2 k) = \Omega(n^2/k)$. The element comparison at line 9 done at most $n \cdot |S|$. Other operations, like for example insertion into or searching in sets can be considered super linear, if for example sets are ordered immediately at insertion and then searching is done with binary search. So the algorithm is in $\mathcal{O}(n^2/k)$. \square

Corollary 2.40. *Let $k \in \mathbb{N}$ and $n = |\{w \in \mathcal{I}_\theta : \rho(w) \leq k\}|$. Then any algorithm calculating $Wk(\theta)$ is in $\Omega(n)$.*

Proof. Any algorithm must at least return $\{w \in \mathcal{I}_\theta : \rho(w) \leq k\}$ and this set is of size n . \square

Our goal is to improve TWOA1 so that we get an algorithm in $\mathcal{O}(n)$, i.e. an asymptotical perfect algorithm for calculating $Wk(\theta)$. As already seen the element comparison of a potential new element with all already known elements of same twisted length (line 17) is the bottleneck. Here the rank-2-residuums become key. Suppose we have a $w \in \mathcal{I}_\theta$ with $\rho(w) = k$ and $s \in S$. In TWOA1 we would now check, if w_s is a new vertex, or if we already calculated it by comparing it with all already known vertices of twisted length $k + 1$. Assume it is already known. This means there is another twisted involution v with $\rho(v) = k$ and another generator $t \in S$ with $vt = w_s$. With Proposition 2.31 w_s is the unique element of maximal twisted length in the rank-2-residuum $wC_{\{s,t\}}$. This yields a necessary condition for w_s to be equal to a already known vertex, allowing us to replace the ineffective search all method in TWOA1, line 17.

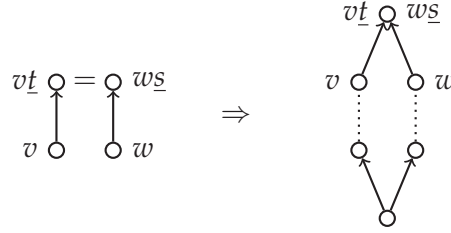


Figure 2.4: Optimization of TWOA1

Lemma 2.41. *Let $k \in \mathbb{N}$ and suppose we are in the situation described at the beginning of Section 2.4. Let $\rho_i := \{w \in \mathcal{I}_\theta : \rho(w) = i\}$ and ρ'_{k+1} the set of the already calculated vertices with twisted length $k + 1$. If $w_s \in \rho'_{k+1}$ for some $w \in \rho_k, s \in S$, say $w_s = v_t$ with $v \in \rho_k$ and $t \in S \setminus \{s\}$, then $w_s = w[\underline{ts}]^n$ for some $n \in \mathbb{N}$ with $w[\underline{ts}]^j \in \rho_0 \cup \dots \cup \rho_k \cup \rho'_{k+1}$ for $1 \leq j \leq n$.*

Proof. The equality $w_s = w[\underline{ts}]^n$ for some $n \in \mathbb{N}$ is due to Proposition 2.31. All vertices in this rank-2-residuum except v_t have a twisted length of k or lower. For v_t we supposed it is already known, hence $v_t \in \rho'_{k+1}$. Therefore all vertices $w[\underline{ts}]^j, 1 \leq j \leq n$ are in $\rho_0 \cup \dots \cup \rho_k \cup \rho'_{k+1}$. \square

This can be checked effectively. Both, w and s are fixed. Start with $M = \emptyset$. For all already known edges from or to w being labeled with $\underline{t} \in \underline{S} \setminus \{s\}$ we do the following: Walk $w[\underline{ts}]^i$ for $i = 0, 1, \dots$ until $\rho(w[\underline{ts}]^i) = k + 1$. Note that walking in this case really means walking the graph. All involved vertices and edges have already been calculated. So there is no need for more calculations in W to find $w[\underline{ts}]^i$. By Proposition 2.31 such a path must exist (in a completely calculated graph). But we could be in the case, where the last step from $w[\underline{ts}]^{i-1}$ to $w[\underline{ts}]^i$ has not been calculated yet. If it is already calculated, then add this element to M by setting $M = M \cup \{w[\underline{ts}]^i\}$. If not, do not add it to M .

Now M contains all already known elements of twisted length $k + 1$, satisfying the necessary condition from Lemma 2.41. Furthermore $|M| < |S|$. So for each pair (w, s) we have to do at most $|S| - 1$ many element comparisons to determine, if w_s is new or already known, no matter how many elements of twisted length $k + 1$ are already known. We can get even more information from the rank-2-residuums:

Lemma 2.42. *Let $w \in \mathcal{I}_\theta$ with $\rho(w) = k, s, t$ be two distinct generators and $s \notin D_R(w)$. Suppose $n \in \mathbb{N}$ to be the smallest number for that $\rho(w[\underline{ts}]^{2n-1}) = k + 1$ holds. Then:*

1. *If $n = \text{ord}(st)$, then $w[\underline{ts}]^{2n-1} = w_s$.*
2. *If $n \geq 2$ and $l(w[\underline{ts}]^{2n-1}) - l(w[\underline{ts}]^{2n-2}) = 1$, then $w[\underline{ts}]^{2n-1} = w_s$.*

Proof. 1. Follows immediately from Lemma 2.35.

2. Because of the length difference the step from $w[\underline{ts}]^{2n-2}$ to $w[\underline{ts}]^{2n-1}$ is a multiplication, not a twisted conjugation, and because of $n \geq 1$ this step cannot be next to the smallest element in $wC_{\{s,t\}}$. Hence $w[\underline{ts}]^{2n-1} = w_s$ by Corollary 2.33. \square

Lemma 2.43. *Let $w \in \mathcal{I}_\theta$ with $\rho(w) = k$, s, t be two distinct generators and $s \notin D_R(w)$. Suppose $w[\underline{ts}]^{2n-1} = w\underline{s}$ and suppose n to be the smallest number with this property. Then $w[\underline{ts}]^{n-1}$ is the minimal element $\min(w, \{s, t\})$ and $w[\underline{ts}]^{2n-1}$ is the maximal element. Define*

$$\begin{aligned} a &= l(w\underline{s}) - l(w), \\ b &= l(w[\underline{ts}]^{n-1}) - l(w[\underline{ts}]^{n-2}), \\ c &= l(w[\underline{ts}]^n) - l(w[\underline{ts}]^{n-1}) \text{ and} \\ d &= l(w[\underline{ts}]^{2n-1}) - l(w[\underline{ts}]^{2n-2}). \end{aligned}$$

Note that $a, b, c, d \in \{1, 2\}$ contain the information, if edges next to the minimal and the maximal element of $wC_{\{s,t\}}$ are twisted conjugations or multiplications. Then each can be deduced from the three remaining ones with the equation $(a + b + c + d) \equiv 0 \pmod{2}$.

Proof. The minimality of $w[\underline{ts}]^{n-1}$ and the maximality of $w[\underline{ts}]^{2n-1}$ is due to Proposition 2.31. The soundness of the equation follows from the symmetric distribution of twisted conjugations and mutipliations from Lemma 2.34. \square

Algorithm 2.44 (TWOA2).

```

1: procedure TWISTEDWEAKORDERINGALGORITHM1( $(W, S), k_{max}$ )
2:    $V \leftarrow \{(e, 0)\}$ 
3:    $E \leftarrow \{\}$ 
4:   for  $k \leftarrow 0$  to  $k_{max}$  do
5:     for all  $(w, k_w) \in V$  with  $k_w = k$  do
6:       for all  $s \in S$  with  $\nexists (\cdot, w, s) \in E$  do       $\triangleright$  Only for  $s \notin D_R(w)$        $\triangleright$  TODO
7:         end for
8:       end for
9:        $k \leftarrow k + 1$ 
10:    end for
11:    return  $(V, E)$                                  $\triangleright$  The poset graph
12: end procedure

```

			Timings		Element comparisons	
W	$ Wk(W, \text{id}) $	$\rho(w_0)$	TWOA1	TWOA2	TWOA1	TWOA2
A_9	9496	25	00:02.180	00:01.372	13,531,414	42,156
A_{10}	35696	30	00:31.442	00:06.276	185,791,174	173,356
A_{11}	140152	36	11:04.241	00:29.830	2,778,111,763	737,313
E_6	892	20	00:03.044	00:00.268	85,857	2,347
E_7	10208	35	06:11.728	00:02.840	7,785,186	29,687
E_8	199952	64	–	11:03.278	–	682,227

Table 2.1: Benchmark

3 Main Thesis

Question 3.1. Let (W, S) be a Coxeter system, $\theta : W \rightarrow W$ an automorphism of W with $\theta^2 = \text{id}$ and $\theta(S) = S$, and $K \subset S$ a subset of S generating a finite subgroup of W with $\theta(K) = K$. Furthermore let $T, S_1, S_2, S_3 \subset S$ be four pairwise disjoint sets of generators. For which Coxeter groups W does the implication

$$w \in w_K C_{T \cup S_i}, i = 1, 2, 3 \Rightarrow w \in w_K C_T \quad (3.1.1)$$

hold for any possible $K, \theta, T, S_1, S_2, S_3$ and w ?

Proposition 3.2. Let (W, S) be a Coxeter system and K, T, S_1, S_2, S_3 be like in Question 3.1. Suppose we have $w \in W$ and $a_1, \dots, a_n \in T \cup S_1$, $b_1, \dots, b_n \in T \cup S_2$, $c_1, \dots, c_n \in T \cup S_3$ with

$$\begin{aligned} w &= w_K a_1 \dots a_n \\ &= w_K b_1 \dots b_n \\ &= w_K c_1 \dots c_n \end{aligned}$$

and (3.1.1) does not hold for these three expressions, i.e. $w \notin w_K C_T$. Then there exist $t_1, \dots, t_m \in T$ and $a'_1, \dots, a'_{n-m} \in T \cup S_1$, $b'_1, \dots, b'_{n-m} \in T \cup S_2$, $c'_1, \dots, c'_{n-m} \in T \cup S_3$ such that

$$\begin{aligned} w t_1 \dots t_m &= w_K a'_1 \dots a'_{n-m} \\ &= w_K b'_1 \dots b'_{n-m} \\ &= w_K c'_1 \dots c'_{n-m} \end{aligned}$$

with $a'_{n-m}, b'_{n-m}, c'_{n-m} \notin T$.

Proof. Suppose at least one element of a_n, b_n, c_n to be in T , for example $a_n \in T$. Then we can apply a_n to all three expressions. Since $\rho(w a_n) < \rho(w)$ the Exchange property for twisted expressions yields

$$\begin{aligned} w a_n &= w_K a_1 \dots a_n a_n = w_K a_1 \dots a_{n-1} \\ &= w_K b_1 \dots b_n a_n = w_K b_1 \dots \hat{b}_i \dots b_n \\ &= w_K c_1 \dots c_n a_n = w_K c_1 \dots \hat{c}_j \dots c_n \end{aligned}$$

where $\hat{}$ means omission. The omission cannot occur within w_K since all three expressions are still of same twisted length and in the first expression we can see, that $w_K \preceq w a_n$ still holds. This step can be repeated until $w = w_K$ or $a_n, b_n, c_n \notin T$. \square

Lemma 3.3. A counterexample to Question 3.1 can only exist, if there is an element $u \in w_K C_T$ and three distinct generators $s_1, s_2, s_3 \in D_R(u)$ such that $u s_i \notin w_K C_T$ for $i = 1, 2, 3$.

Proof. According to Proposition 3.2. \square

A Source codes

```

1 Read("twistedinvolutionweakordering.gap");
2
3 CalculateTwistedWeakOrderings := function()
4     local tasks, task, theta, W, matrix;
5
6     tasks := [
7         rec(system := CoxeterGroup_An(1), thetas := [ "id" ]),
8         rec(system := CoxeterGroup_An(2), thetas := [ "id", "-id" ]),
9         rec(system := CoxeterGroup_An(3), thetas := [ "id", "-id" ]),
10        rec(system := CoxeterGroup_An(4), thetas := [ "id", "-id" ]),
11        rec(system := CoxeterGroup_An(5), thetas := [ "id", "-id" ]),
12        rec(system := CoxeterGroup_An(6), thetas := [ "id", "-id" ]),
13        rec(system := CoxeterGroup_An(7), thetas := [ "id", "-id" ]),
14        rec(system := CoxeterGroup_An(8), thetas := [ "id", "-id" ]),
15        rec(system := CoxeterGroup_An(9), thetas := [ "id", "-id" ]),
16        rec(system := CoxeterGroup_An(10), thetas := [ "id", "-id" ]),
17        # rec(system := CoxeterGroup_An(11), thetas := [ "id", "-id" ]),
18        # rec(system := CoxeterGroup_An(10), thetas := [ "id" ]),
19        # rec(system := CoxeterGroup_An(12), thetas := [ "id", "-id" ]),
20        # rec(system := CoxeterGroup_An(13), thetas := [ "id", "-id" ]),
21        rec(system := CoxeterGroup_BCn(2), thetas := [ "id", "-id" ]),
22        rec(system := CoxeterGroup_BCn(3), thetas := [ "id" ]),
23        # rec(system := CoxeterGroup_BCn(4), thetas := [ "id" ]),
24        # rec(system := CoxeterGroup_BCn(5), thetas := [ "id" ]),
25        # rec(system := CoxeterGroup_BCn(6), thetas := [ "id" ]),
26        # rec(system := CoxeterGroup_BCn(7), thetas := [ "id" ]),
27        rec(system := CoxeterGroup_Dn(4), thetas := [ "id" ]),
28        rec(system := CoxeterGroup_Dn(5), thetas := [ "id" ]),
29        rec(system := CoxeterGroup_Dn(6), thetas := [ "id" ]),
30        rec(system := CoxeterGroup_E6(), thetas := [ "id", [6,5,3,4,2,1] ]),
31        rec(system := CoxeterGroup_E7(), thetas := [ "id" ]),
32        # rec(system := CoxeterGroup_E8(), thetas := [ "id" ]),
33        rec(system := CoxeterGroup_F4(), thetas := [ "id" ]),
34        rec(system := CoxeterGroup_H3(), thetas := [ "id" ]),
35        rec(system := CoxeterGroup_H4(), thetas := [ "id" ]),
36        # rec(system := CoxeterGroup_I2m(3), thetas := [ "id", "-id" ]),
37        # rec(system := CoxeterGroup_I2m(4), thetas := [ "id", "-id" ]),
38        # rec(system := CoxeterGroup_I2m(5), thetas := [ "id", "-id" ]),
39        # rec(system := CoxeterGroup_I2m(6), thetas := [ "id", "-id" ]),
40    ];
41
42    for task in tasks do
43        W := task.system.group;
44        matrix := task.system.matrix;
45
46        for theta in List(task.thetas, t -> GroupAutomorphismByPermutation(W, t)) do
47            Print(Name(W), " ", Name(theta), "\n");
48            TwistedInvolutionWeakOrdering(StringToFilename(Concatenation(Name(W), "-",
49                Name(theta))), W, matrix, theta);
50        od;
51    od;
52
53    Benchmark := function ()
54        local tasks, task, theta, W, matrix, benchmarks, b, result, startTime, endTime;
55
56        tasks := [
57            rec(system := CoxeterGroup_An(9), thetas := [ "id" ]),

```

```

58     #rec(system := CoxeterGroup_An(10), thetas := [ "id" ]),
59     #rec(system := CoxeterGroup_An(11), thetas := [ "id" ]),
60     rec(system := CoxeterGroup_E6(), thetas := [ "id" ]),
61     #rec(system := CoxeterGroup_E7(), thetas := [ "id" ]),
62 ];
63
64 benchmarks := [];
65
66 Print("Benchmark algo 1\n");
67
68 for task in tasks do
69     W := task.system.group;
70     matrix := task.system.matrix;
71
72     for theta in List(task.thetas, t -> GroupAutomorphismByPermutation(W, t)) do
73         Print(Name(W), " ", Name(theta), "\n");
74
75         startTime := Runtime();
76         result := TwistedInvolutionWeakOrdering1(fail, W, matrix, theta);
77         endTime := Runtime();
78
79         Add(benchmarks, rec(name := Name(W), algo := "TWOA1", time := StringTime(
80             endTime - startTime), result := result));
81     od;
82
83 Print("Benchmark algo 2\n");
84
85 for task in tasks do
86     W := task.system.group;
87     matrix := task.system.matrix;
88
89     for theta in List(task.thetas, t -> GroupAutomorphismByPermutation(W, t)) do
90         Print(Name(W), " ", Name(theta), "\n");
91
92         startTime := Runtime();
93         result := TwistedInvolutionWeakOrdering(fail, W, matrix, theta);
94         endTime := Runtime();
95
96         Add(benchmarks, rec(name := Name(W), algo := "TWOA2", time := StringTime(
97             endTime - startTime), result := result));
98     od;
99
100    for b in benchmarks do
101        Display(b);
102    od;
103 end;
104
105 TestCondition := function ()
106     local tasks, task, S, K, _T, T, K12, K23, K31, wK, parts, part, graph,
107         resS12, resS23, resS31, resT, resDiff, i, j;
108
109     tasks := [
110     #         "H_3-id",
111     #         "H_4-id",
112     #         "F_4-id",
113     #         "D__4-id",
114     #         "D__5-id",
115     #         "D__6-id",
116     #         "E_6-id",

```

```

117 #      "E_7-id",
118 #      "E_8-id",
119 #      "A__4-id",
120 #      "A__5-id",
121 #      "A__6-id",
122 #      "A__7-id",
123 #      "BC__4-id",
124 #      "BC__5-id",
125 #      "BC__6-id",
126 #      "A__8-id",
127 #      "A__9-id",
128 #      "A__10-id",
129 #      "BC__7-id",
130 ];
131
132 for task in tasks do
133   graph := TwistedInvolutionWeakOrderingPersistReadResults(task);
134   S := [1..graph.data.rank];
135   Print(graph.data.name, " ", graph.data.automorphism, "\n");
136   i := 0;
137
138   for K in IteratorOfCombinations(S) do
139     j := 0;
140     i := i + 1;
141
142     if Length(K) <= 2 or Length(K) = Length(S) then continue; fi;
143     parts := PartitionsSet(K, 3);
144     wK := TwistedInvolutionWeakOrderungLongestWord(graph.vertices[1], K);
145
146     for _T in IteratorOfCombinations(K) do
147       j := j + 1;
148       Print(i, " ", j, "                                \r");
149
150       T := Union(_T, Difference(S, K));
151
152       for part in parts do
153         K12 := part[1];
154         K23 := part[2];
155         K31 := part[3];
156         #Print("K=", K, " T=", T, " K12=", K12, " K23=", K23, " K31=", K31,
157             "\n");
158
159         resS12 := TwistedInvolutionWeakOrderungResiduum(wK, Union(K12, T));
160         resS23 := TwistedInvolutionWeakOrderungResiduum(wK, Union(K23, T));
161         resS31 := TwistedInvolutionWeakOrderungResiduum(wK, Union(K31, T));
162         resT := TwistedInvolutionWeakOrderungResiduum(wK, T);
163
164         resDiff := Difference(Intersection(resS12, resS23, resS31), resT);
165
166         if Length(resDiff) > 0 then
167           Print("*** FOUND COUNTEREXAMPLE ***\n",
168               "W = ", graph.name, "\n",
169               "theta = ", graph.automorphis, "\n",
170               "S = ", S, "\n",
171               "K = ", K, "\n",
172               "wK = ", wK, "\n",
173               "T = ", T, "\n",
174               "K12 = ", K12, "\n",
175               "K23 = ", K23, "\n",
176               "K31 = ", K31, "\n",
177               "w = ", resDiff, "\n\n");

```

```

177         fi;
178     od;
179 od;
180 od;
181 od;
182 end;

1 LoadPackage("io");
2
3 Read("misc.gap");
4 Read("coxeter.gap");
5 Read("twistedinvolutionsweakordering-persist.gap");
6
7 TwistedInvolutionsDeduceNodeAndEdgeFromGraph := function(matrix, startNode, startLabel,
8     labels)
9     local rank, comb, trace, possibleEqualNodes, e, k, n;
10
11     rank := -1/2 + Sqrt(1/4 + 2*Length(matrix)) + 1;
12     possibleEqualNodes := [];
13
14     for comb in List(Filtered(labels, label -> label <> startLabel), label -> rec(
15         startNode := startNode, s := [startLabel, label], m := CoxeterMatrixEntry(
16         matrix, rank, startLabel, label))) do
17         trace := [];
18         k := 1;
19         n := comb.startNode;
20
21         Add(trace, rec(node := n, edge := rec(label := comb.s[1], type := -1)));
22
23         while k < comb.m do
24             e := FindElement(n.inEdges, e -> e.label = comb.s[k mod 2 + 1]);
25             if e = fail then break; fi;
26             n := e.source;
27
28             Add(trace, rec(node := n, edge := e));
29             k := k + 1;
30         od;
31
32         while k > 0 do
33             e := FindElement(n.outEdges, e -> e.label = comb.s[k mod 2 + 1]);
34             if e = fail then break; fi;
35             n := e.target;
36
37             Add(trace, rec(node := n, edge := e));
38             k := k - 1;
39         od;
40
41         if k <> 0 then continue; fi;
42
43         if Length(trace) = 2*comb.m then
44             return rec(result := 0, node := trace[Length(trace)].node, type := trace[
45                 comb.m + 1].edge.type, trace := trace);
46         fi;
47
48         if Length(trace) >= 4 then
49             if trace[Length(trace) / 2 + 1].edge.type <> trace[Length(trace) / 2].edge.
50                 type then
51                 # cannot be equal
52             else
53                 if trace[Length(trace)].edge.type = 0 then
54                     return rec(result := 0, node := trace[Length(trace)].node, type :=

```

```

                                0, trace := trace);
50         else
51             Add(possibleEqualNodes, trace[Length(trace)].node);
52         fi;
53     fi;
54     else
55         Add(possibleEqualNodes, trace[Length(trace)].node);
56     fi;
57 od;
58
59     return rec(result := -1, possibleEqualNodes := possibleEqualNodes);
60 end;
61
62 # Calculates the poset Wk(theta).
63 TwistedInvolutionWeakOrdering := function (filename, W, matrix, theta)
64     local persistInfo, maxOrder, nodes, edges, absNodeIndex, absEdgeIndex, prevNode,
65         currNode, newEdge,
66         label, type, deduction, startTime, endTime, S, k, i, s, x, y, n;
67
68     persistInfo := TwistedInvolutionWeakOrderingPersistResultsInit(filename);
69
70     S := GeneratorsOfGroup(W);
71     maxOrder := Minimum([Maximum(Concatenation(matrix, [1])), 5]);
72     nodes := [ [], [ rec(element := One(W), twistedLength := 0, inEdges := [], outEdges
73         := [], absIndex := 1) ] ];
74     edges := [ [], [] ];
75     absNodeIndex := 2;
76     absEdgeIndex := 1;
77     k := 0;
78
79     while Length(nodes[2]) > 0 do
80         if not IsFinite(W) then
81             if k > 200 or absNodeIndex > 10000 then
82                 break;
83             fi;
84         fi;
85
86         for i in [1..Length(nodes[2])] do
87             Print(k, " ", i, " \r");
88
89             prevNode := nodes[2][i];
90             for label in Filtered([1..Length(S)], n -> Position(List(prevNode.inEdges,
91                 e -> e.label), n) = fail) do
92                 deduction := TwistedInvolutionDeduceNodeAndEdgeFromGraph(matrix,
93                     prevNode, label, [1..Length(S)]);
94
95                 if deduction.result = 0 then
96                     type := deduction.type;
97                     currNode := deduction.node;
98                 elif deduction.result = 1 then
99                     type := deduction.type;
100
101                     currNode := rec(element := y, twistedLength := k + 1, inEdges :=
102                         [], outEdges := [], absIndex := absNodeIndex);
103                     Add(nodes[1], currNode);
104
105                     absNodeIndex := absNodeIndex + 1;
106                 else
107                     x := prevNode.element;
108                     s := S[label];
109

```

```

105         type := 1;
106         y := s^theta*x*s;
107         if (CoxeterElementsCompare(x, y)) then
108             y := x * s;
109             type := 0;
110         fi;
111
112         currNode := FindElement(deduction.possibleEqualNodes, n ->
                                CoxeterElementsCompare(n.element, y));
113
114         if currNode = fail then
115             currNode := rec(element := y, twistedLength := k + 1, inEdges
                             := [], outEdges := [], absIndex := absNodeIndex);
116             Add(nodes[1], currNode);
117
118             absNodeIndex := absNodeIndex + 1;
119         fi;
120     fi;
121
122     newEdge := rec(source := prevNode, target := currNode, label := label,
                    type := type, absIndex := absEdgeIndex);
123
124     Add(edges[1], newEdge);
125     Add(currNode.inEdges, newEdge);
126     Add(prevNode.outEdges, newEdge);
127
128     absEdgeIndex := absEdgeIndex + 1;
129 od;
130 od;
131
132 TwistedInvolutionWeakOrderingPersistResults(persistInfo, nodes[2], edges[2]);
133
134 Add(nodes, [], 1);
135 Add(edges, [], 1);
136 if (Length(nodes) > maxOrder + 1) then
137     for n in nodes[maxOrder + 2] do
138         n.inEdges := [];
139         n.outEdges := [];
140     od;
141     Remove(nodes, maxOrder + 2);
142     Remove(edges, maxOrder + 2);
143 fi;
144 k := k + 1;
145 od;
146
147 TwistedInvolutionWeakOrderingPersistResultsInfo(persistInfo, W, matrix, theta,
148         absNodeIndex - 1, k - 1);
149 TwistedInvolutionWeakOrderingPersistResultsClose(persistInfo);
150
151 return rec(numNodes := absNodeIndex - 1, numEdges := absEdgeIndex - 1,
152         maxTwistedLength := k - 1);
153 end;
154
155 # Calculates the poset Wk(theta).
156 TwistedInvolutionWeakOrdering1 := function (filename, W, matrix, theta)
157     local persistInfo, maxOrder, nodes, edges, absNodeIndex, absEdgeIndex, prevNode,
158         currNode, newEdge,
159         label, type, deduction, startTime, endTime, S, k, i, s, x, y, n;
160
161     persistInfo := TwistedInvolutionWeakOrderingPersistResultsInit(filename);

```

```

160 S := GeneratorsOfGroup(W);
161 maxOrder := Minimum([Maximum(Concatenation(matrix, [1])), 5]);
162 nodes := [ [], [ rec(element := One(W), twistedLength := 0, inEdges := [], outEdges
    := [], absIndex := 1) ] ];
163 edges := [ [], [] ];
164 absNodeIndex := 2;
165 absEdgeIndex := 1;
166 k := 0;
167
168 while Length(nodes[2]) > 0 do
169     if not IsFinite(W) then
170         if k > 200 or absNodeIndex > 10000 then
171             break;
172         fi;
173     fi;
174
175     for i in [1..Length(nodes[2])] do
176         Print(k, " ", i, " \r");
177
178         prevNode := nodes[2][i];
179         for label in Filtered([1..Length(S)], n -> Position(List(prevNode.inEdges,
180             e -> e.label), n) = fail) do
181             x := prevNode.element;
182             s := S[label];
183
184             type := 1;
185             y := s^theta*x*s;
186             if (CoxeterElementsCompare(x, y)) then
187                 y := x * s;
188                 type := 0;
189             fi;
190
191             currNode := FindElement(nodes[1], n -> CoxeterElementsCompare(n.element
192                 , y));
193
194             if currNode = fail then
195                 currNode := rec(element := y, twistedLength := k + 1, inEdges :=
196                     [], outEdges := [], absIndex := absNodeIndex);
197                 Add(nodes[1], currNode);
198
199                 absNodeIndex := absNodeIndex + 1;
200             fi;
201
202             newEdge := rec(source := prevNode, target := currNode, label := label,
203                 type := type, absIndex := absEdgeIndex);
204
205             Add(edges[1], newEdge);
206             Add(currNode.inEdges, newEdge);
207             Add(prevNode.outEdges, newEdge);
208
209             absEdgeIndex := absEdgeIndex + 1;
210         od;
211     od;
212
213     TwistedInvolutionWeakOrderingPersistResults(persistInfo, nodes[2], edges[2]);
214
215     Add(nodes, [], 1);
216     Add(edges, [], 1);
217     if (Length(nodes) > maxOrder + 1) then
218         for n in nodes[maxOrder + 2] do
219             n.inEdges := [];

```



```

216         n.outEdges := [];
217     od;
218     Remove(nodes, maxOrder + 2);
219     Remove(edges, maxOrder + 2);
220 fi;
221 k := k + 1;
222 od;
223
224 TwistedInvolutionWeakOrderingPersistResultsInfo(persistInfo, W, matrix, theta,
    absNodeIndex - 1, k - 1);
225 TwistedInvolutionWeakOrderingPersistResultsClose(persistInfo);
226
227 return rec(numNodes := absNodeIndex - 1, numEdges := absEdgeIndex - 1,
    maxTwistedLength := k - 1);
228 end;
229
230 TwistedInvolutionWeakOrderungResiduum := function (vertex, labels)
231     local visited, queue, residuum, current, edge;
232
233     visited := [ vertex ];
234     queue := [ vertex ];
235     residuum := [];
236
237     while Length(queue) > 0 do
238         current := queue[1];
239         Remove(queue, 1);
240         Add(residuum, current);
241
242         for edge in current.outEdges do
243             if edge.label in labels and not edge.target in visited then
244                 Add(visited, edge.target);
245                 Add(queue, edge.target);
246             fi;
247         od;
248     od;
249
250     return residuum;
251 end;
252
253 TwistedInvolutionWeakOrderungLongestWord := function (vertex, labels)
254     local current;
255
256     current := vertex;
257
258     while Length(Filtered(current.outEdges, e -> e.label in labels)) > 0 do
259         current := Filtered(current.outEdges, e -> e.label in labels)[1].target;
260     od;
261
262     return current;
263 end;

```



```

1 GroupAutomorphismByPermutation := function (G, generatorPermutation)
2     local automorphism, generators;
3
4     generators := GeneratorsOfGroup(G);
5
6     if generatorPermutation = "id" or generatorPermutation = [1..Length(generators)]
7         then
8         automorphism := IdentityMapping(G);
9         SetName(automorphism, "id");

```

```

10     return automorphism;
11     elif generatorPermutation = "-id" then
12         generatorPermutation := Reversed([1..Length(GeneratorsOfGroup(G))]);
13     fi;
14
15     automorphism := GroupHomomorphismByImages(G, G, generators, generators{
16         generatorPermutation});
17     SetName(automorphism, Concatenation("(", JoinStringsWithSeparator(
18         generatorPermutation, ","), ")"));
19
20     return automorphism;
21 end;
22
23 GroupAutomorphismIdNeg := function (G)
24     return GroupAutomorphismByPermutation(G, Reversed([1..Length(GeneratorsOfGroup(G))
25     ])));
26 end;
27
28 GroupAutomorphismId := function (G)
29     return GroupAutomorphismByPermutation(G, [1..Length(GeneratorsOfGroup(G))]);
30 end;
31
32 FindElement := function (list, selector)
33     local i;
34
35     for i in [1..Length(list)] do
36         if (selector(list[i])) then
37             return list[i];
38         fi;
39     od;
40
41     return fail;
42 end;
43
44 StringToFilename := function(str)
45     local result, c;
46
47     result := "";
48
49     for c in str do
50         if IsDigitChar(c) or IsAlphaChar(c) or c = '-' or c = '_' then
51             Add(result, c);
52         else
53             Add(result, '_');
54         fi;
55     od;
56
57     return result;
58 end;
59
60 IO_ReadLinesIterator := function (file)
61     local IsDone, Next, ShallowCopy;
62
63     IsDone := function (iter)
64         return iter!.nextLine = "" or iter!.nextLine = fail;
65     end;
66
67     Next := function (iter)
68         local line;
69
70         line := iter!.nextLine;

```

```

68
69     if line = fail then
70         Error(LastSystemError());
71         return fail;
72     fi;
73
74     iter!.nextLine := IO_ReadLine(iter!.file);
75
76     return Chomp(line);
77 end;
78
79 ShallowCopy := function (iter)
80     return fail;
81 end;
82
83 return IteratorByFunctions(rec(IsDoneIterator := IsDone, NextIterator := Next,
84     ShallowCopy := ShallowCopy, file := file, nextLine := IO_ReadLine(file)));
85 end;
86
87 IO_ReadLinesIteratorCSV := function (file, seperator)
88     local IsDone, Next, ShallowCopy;
89
90     IsDone := function (iter)
91         return iter!.nextLine = "" or iter!.nextLine = fail;
92     end;
93
94     Next := function (iter)
95         local line, lineSplitted, result, i;
96
97         line := iter!.nextLine;
98         if line = fail then
99             Error(LastSystemError());
100            return fail;
101        fi;
102        iter!.nextLine := IO_ReadLine(iter!.file);
103
104        lineSplitted := SplitString(Chomp(line), iter!.seperator);
105        result := rec();
106
107        for i in [1..Minimum(Length(iter!.headers), Length(lineSplitted))] do
108            result.(iter!.headers[i]) := EvalString(lineSplitted[i]);
109        od;
110
111        return result;
112    end;
113
114    ShallowCopy := function (iter)
115        return fail;
116    end;
117
118    return IteratorByFunctions(rec(IsDoneIterator := IsDone, NextIterator := Next,
119        ShallowCopy := ShallowCopy, file := file, seperator := seperator,
120        headers := SplitString(Chomp(IO_ReadLine(file)), seperator),
121        nextLine := IO_ReadLine(file)));
122 end;

```



```

1 Read("coxeter-generators.gap");
2
3 CoxeterElementsCompare := function (w1, w2)
4     return w1 = w2;
5 end;

```

```

6
7 CoxeterMatrixEntry := function(matrix, rank, i, j)
8   local temp;
9
10  if (i = j) then
11    return 1;
12  fi;
13
14  if (i > j) then
15    temp := i;
16    i := j;
17    j := temp;
18  fi;
19
20  return matrix[(rank-1)*(rank)/2 - (rank-i)*(rank-i+1)/2 + (j-i-1) + 1];
21 end;

1 # Generates a coxeter group with given rank and relations. The relations have to
2 # be given in a linear list of the upper right entries (above diagonal) of the
3 # coxeter matrix.
4 #
5 # Example:
6 # To generate the coxeter group A_4 with the following coxeter matrix:
7 #
8 # | 1 3 2 2 |
9 # | 3 1 3 2 |
10 # | 2 3 1 3 |
11 # | 2 2 3 1 |
12 #
13 # A4 := CoxeterGroup(4, [3,2,2, 3,2, 3]);
14 CoxeterGroup := function (rank, upperTriangleOfCoxeterMatrix)
15   local generatorNames, relations, F, S, W, i, j, k;
16
17   generatorNames := List([1..rank], n -> Concatenation("s", String(n)));
18
19   F := FreeGroup(generatorNames);
20   S := GeneratorsOfGroup(F);
21
22   relations := [];
23
24   Append(relations, List([1..rank], n -> S[n]^2));
25
26   k := 1;
27   for i in [1..rank] do
28     for j in [i+1..rank] do
29       Add(relations, (S[i]*S[j])^(upperTriangleOfCoxeterMatrix[k]));
30       k := k + 1;
31     od;
32   od;
33
34   W := F / relations;
35
36   return W;
37 end;
38
39 CoxeterGroup_An := function (n)
40   local upperTriangleOfCoxeterMatrix, W;
41
42   upperTriangleOfCoxeterMatrix := Flat(List(Reversed([1..n-1]), m -> Concatenation
43     ([3], List([1..m-1], o -> 2))));

```

```

44     #W := CoxeterGroup(n, upperTriangleOfCoxeterMatrix);
45     W := GroupWithGenerators(List([1..n], s -> (s,s+1)));
46
47     SetName(W, Concatenation("A_", String(n), "}"));
48     SetSize(W, Factorial(n + 1));
49
50     return rec(group := W, rank := n, matrix := upperTriangleOfCoxeterMatrix);
51 end;
52
53 CoxeterGroup_BcN := function (n)
54     local upperTriangleOfCoxeterMatrix, W;
55
56     upperTriangleOfCoxeterMatrix := Flat(List(Reversed([1..n-1]), m -> Concatenation
57         ([3], List([1..m-1], o -> 2))));
58     upperTriangleOfCoxeterMatrix[Length(upperTriangleOfCoxeterMatrix)] := 4;
59
60     W := CoxeterGroup(n, upperTriangleOfCoxeterMatrix);
61
62     SetName(W, Concatenation("BC_", String(n), "}"));
63     SetSize(W, 2^n * Factorial(n));
64
65     return rec(group := W, rank := n, matrix := upperTriangleOfCoxeterMatrix);
66 end;
67
68 CoxeterGroup_Dn := function (n)
69     local upperTriangleOfCoxeterMatrix, W;
70
71     upperTriangleOfCoxeterMatrix := Flat(List(Reversed([1..n-1]), m -> Concatenation
72         ([3], List([1..m-1], o -> 2))));
73     upperTriangleOfCoxeterMatrix[Length(upperTriangleOfCoxeterMatrix)] := 2;
74     upperTriangleOfCoxeterMatrix[Length(upperTriangleOfCoxeterMatrix) - 1] := 3;
75     upperTriangleOfCoxeterMatrix[Length(upperTriangleOfCoxeterMatrix) - 2] := 3;
76
77     W := CoxeterGroup(n, upperTriangleOfCoxeterMatrix);
78
79     SetName(W, Concatenation("D_", String(n), "}"));
80     SetSize(W, 2^(n-1) * Factorial(n));
81
82     return rec(group := W, rank := n, matrix := upperTriangleOfCoxeterMatrix);
83 end;
84
85 CoxeterGroup_E6 := function ()
86     local upperTriangleOfCoxeterMatrix, W;
87
88     upperTriangleOfCoxeterMatrix := [3, 2, 2, 2, 2, 3, 2, 2, 2, 3, 3, 2, 2, 2, 3];
89
90     W := CoxeterGroup(6, upperTriangleOfCoxeterMatrix);
91
92     SetName(W, "E_6");
93     SetSize(W, 2^7 * 3^4 * 5);
94
95     return rec(group := W, rank := 6, matrix := upperTriangleOfCoxeterMatrix);
96 end;
97
98 CoxeterGroup_E7 := function ()
99     local upperTriangleOfCoxeterMatrix, W;
100
101     upperTriangleOfCoxeterMatrix := [3, 2, 2, 2, 2, 2, 3, 2, 2, 2, 2, 3, 3, 2, 2, 2,
102         2, 2, 3, 2, 3];
103
104     W := CoxeterGroup(7, upperTriangleOfCoxeterMatrix);

```

```

102
103     SetName(W, "E_7");
104     SetSize(W, 2^10 * 3^4 * 5 * 7);
105
106     return rec(group := W, rank := 7, matrix := upperTriangleOfCoxeterMatrix);
107 end;
108
109 CoxeterGroup_E8 := function ()
110     local upperTriangleOfCoxeterMatrix, W;
111
112     upperTriangleOfCoxeterMatrix := [3, 2, 2, 2, 2, 2, 2, 3, 2, 2, 2, 2, 2, 3, 3, 2,
113         2, 2, 2, 2, 2, 2, 3, 2, 2, 3, 2, 3];
114
115     W := CoxeterGroup(8, upperTriangleOfCoxeterMatrix);
116
117     SetName(W, "E_8");
118     SetSize(W, 2^14 * 3^5 * 5^2 * 7);
119
120     return rec(group := W, rank := 8, matrix := upperTriangleOfCoxeterMatrix);
121 end;
122
123 CoxeterGroup_F4 := function ()
124     local upperTriangleOfCoxeterMatrix, W;
125
126     upperTriangleOfCoxeterMatrix := [3, 2, 2, 4, 2, 3];
127
128     W := CoxeterGroup(4, upperTriangleOfCoxeterMatrix);
129
130     SetName(W, "F_4");
131     SetSize(W, 2^7 * 3^2);
132
133     return rec(group := W, rank := 4, matrix := upperTriangleOfCoxeterMatrix);
134 end;
135
136 CoxeterGroup_H3 := function ()
137     local upperTriangleOfCoxeterMatrix, W;
138
139     upperTriangleOfCoxeterMatrix := [5, 2, 3];
140
141     W := CoxeterGroup(3, upperTriangleOfCoxeterMatrix);
142
143     SetName(W, "H_3");
144     SetSize(W, 120);
145
146     return rec(group := W, rank := 3, matrix := upperTriangleOfCoxeterMatrix);
147 end;
148
149 CoxeterGroup_H4 := function ()
150     local upperTriangleOfCoxeterMatrix, W;
151
152     upperTriangleOfCoxeterMatrix := [5, 2, 2, 3, 2, 3];
153
154     W := CoxeterGroup(4, upperTriangleOfCoxeterMatrix);
155
156     SetName(W, "H_4");
157     SetSize(W, 14400);
158
159     return rec(group := W, rank := 4, matrix := upperTriangleOfCoxeterMatrix);
160 end;
161
162 CoxeterGroup_I2m := function (m)

```

```

162     local upperTriangleOfCoxeterMatrix, W;
163
164     upperTriangleOfCoxeterMatrix := [m];
165
166     W := CoxeterGroup(2, upperTriangleOfCoxeterMatrix);
167
168     SetName(W, Concatenation("I_2(", String(m), ")"));
169     SetSize(W, 2*m);
170
171     return rec(group := W, rank := 2, matrix := upperTriangleOfCoxeterMatrix);
172 end;
173
174 CoxeterGroup_TildeAn := function (n)
175     local upperTriangleOfCoxeterMatrix, W;
176
177     upperTriangleOfCoxeterMatrix := Flat(List(Reversed([1..n]), m -> Concatenation([3],
178         List([1..m-1], o -> 2))));
179
180     if n = 1 then
181         upperTriangleOfCoxeterMatrix[1] := 0;
182     else
183         upperTriangleOfCoxeterMatrix[n] := 3;
184     fi;
185
186     W := CoxeterGroup(n + 1, upperTriangleOfCoxeterMatrix);
187
188     SetName(W, Concatenation("\\tilde A_{", String(n), "}"));
189     SetSize(W, infinity);
190
191     return rec(group := W, rank := n + 1, matrix := upperTriangleOfCoxeterMatrix);
192 end;
193
194 TwistedInvolutionWeakOrderingPersistReadResults := function(filename)
195     local fileD, fileV, fileE, csvLine, data, vertices, edges, newEdge, source, target,
196         i;
197
198     fileD := IO_File(Concatenation("results/", filename, "-data"), "r");
199     fileV := IO_File(Concatenation("results/", filename, "-vertices"), "r", 1024*1024);
200     fileE := IO_File(Concatenation("results/", filename, "-edges"), "r", 1024*1024);
201
202     data := NextIterator(IO_ReadLinesIteratorCSV(fileD, ";"));
203     vertices := [];
204     edges := [];
205
206     i := 1;
207     for csvLine in IO_ReadLinesIteratorCSV(fileV, ";") do
208         Add(vertices, rec(absIndex := i, twistedLength := csvLine.twistedLength, name
209             := csvLine.name, inEdges := [], outEdges := []));
210         i := i + 1;
211     od;
212
213     i := 1;
214     for csvLine in IO_ReadLinesIteratorCSV(fileE, ";") do
215         source := vertices[csvLine.sourceIndex + 1];
216         target := vertices[csvLine.targetIndex + 1];
217         newEdge := rec(absIndex := i, source := source, target := target, label :=
218             csvLine.label, type := csvLine.type);
219
220         Add(source.outEdges, newEdge);
221         Add(target.inEdges, newEdge);
222         Add(edges, newEdge);
223     od;
224 end;

```

```

27         i := i + 1;
28     od;
29
30     IO_Close(fileD);
31     IO_Close(fileV);
32     IO_Close(fileE);
33
34     return rec(data := data, vertices := vertices, edges := edges);
35 end;
36
37 TwistedInvolutionWeakOrderingPersistResultsInit := function(filename)
38     local fileD, fileV, fileE;
39
40     if (filename = fail) then return fail; fi;
41
42     fileD := IO_File(Concatenation("results/", filename, "-data"), "w");
43     fileV := IO_File(Concatenation("results/", filename, "-vertices"), "w", 1024*1024);
44     fileE := IO_File(Concatenation("results/", filename, "-edges"), "w", 1024*1024);
45     IO_Write(fileD, "name;rank;size;generators;matrix;automorphism;wk_size;
46         wk_max_length\n");
47     IO_Write(fileV, "twistedLength;name\n");
48     IO_Write(fileE, "sourceIndex;targetIndex;label;type\n");
49
50     return rec(fileD := fileD, fileV := fileV, fileE := fileE);
51 end;
52
53 TwistedInvolutionWeakOrderingPersistResultsClose := function(persistInfo)
54     if (persistInfo = fail) then return; fi;
55
56     IO_Close(persistInfo.fileD);
57     IO_Close(persistInfo.fileV);
58     IO_Close(persistInfo.fileE);
59 end;
60
61 TwistedInvolutionWeakOrderingPersistResultsInfo := function(persistInfo, W, matrix,
62     theta, numNodes, maxTwistedLength)
63     if (persistInfo = fail) then return; fi;
64
65     IO_Write(persistInfo.fileD, "\"", ReplacedString(Name(W), "\"", "\\\""), "\";");
66     IO_Write(persistInfo.fileD, Length(GeneratorsOfGroup(W)), ";");
67     if (Size(W) = infinity) then
68         IO_Write(persistInfo.fileD, "\"infinity\";");
69     else
70         IO_Write(persistInfo.fileD, Size(W), ";");
71     fi;
72     IO_Write(persistInfo.fileD, "[", JoinStringsWithSeparator(List(GeneratorsOfGroup(W)
73         , n -> Concatenation("\"", String(n), "\"")), ",", "]);");
74     IO_Write(persistInfo.fileD, "[", JoinStringsWithSeparator(matrix, ",", "]);");
75     IO_Write(persistInfo.fileD, "\"", Name(theta), "\";");
76
77     if (Size(W) = infinity) then
78         IO_Write(persistInfo.fileD, "\"infinity\";");
79         IO_Write(persistInfo.fileD, "\"infinity\"");
80     else
81         IO_Write(persistInfo.fileD, numNodes, ";");
82         IO_Write(persistInfo.fileD, maxTwistedLength, "");
83     fi;
84 end;
85
86 TwistedInvolutionWeakOrderingPersistResults := function(persistInfo, nodes, edges)
87     local n, e, i, tmp, bubbles;

```



```

85
86   if (persistInfo = fail) then return; fi;
87
88   # bubble sort the edges, to make sure, that double edges are neighbours in the list
89   bubbles := 1;
90   while bubbles > 0 do
91     bubbles := 0;
92     for i in [1..Length(edges)-1] do
93       if edges[i].source.absIndex = edges[i+1].source.absIndex and edges[i].
94         target.absIndex > edges[i+1].target.absIndex then
95         tmp := edges[i];
96         edges[i] := edges[i+1];
97         edges[i+1] := tmp;
98         bubbles := bubbles + 1;
99       fi;
100     od;
101
102   for n in nodes do
103     if n.absIndex = 1 then
104       IO_Write(persistInfo.fileV, n.twistedLength, ";"e"\n");
105     else
106       IO_Write(persistInfo.fileV, n.twistedLength, ";"", String(n.element), "\"\n");
107     fi;
108   od;
109
110   for e in edges do
111     IO_Write(persistInfo.fileE, e.source.absIndex-1, ";", e.target.absIndex-1, ";",
112       e.label, ";", e.type, "\n");
113   od;
114 end;

```

B References

- [1] Ruth Haas and Aloysius G. Helminck. Algorithms for twisted involutions in weyl groups. *Algebra Colloquium* 19, 2012.
- [2] Axel Hultman. Fixed points of involutive automorphisms of the bruhat order. *Adv. Math.* 195, pages 283–296, 2005. Also available at <http://www.ams.org/mathscinet-getitem?mr=2145798>.
- [3] Axel Hultman. The combinatorics of twisted involutions in coxeter groups. *Transactions of the American Mathematical Society, Volume 359*, pages 2787–2798, 2007. Also available at <http://www.ams.org/mathscinet-getitem?mr=2286056>.
- [4] James E. Humphreys. *Reflection groups and Coxeter groups*. Cambridge University Press, 1992.