Martin-Luther-Universität Halle-Wittenberg
Naturwissenschaftliche Fakultät II
Institut für Mathematik

Ausgewählte Varianten des Schreier-Sims-Algorithmus'

Parallelisierung einer deterministischen Verifikation

Bachelorarbeit

zur Erlangung des akademischen Grades

Bachelor of Science

im Studiengang Mathematik

vorgelegt von:

Alexander Sebastian Johann Klemps

Matrikelnummer: 213203168

vorgelegt am:
23.08.2016

Erstgutachterin: Prof. Dr. Rebecca Waldecker
Zweitgutachter: Dr. Helmut Podhaisky

Inhaltsverzeichnis

1. Einführung

Die Untersuchung von Permutationsgruppen, welche lediglich durch ein Erzeugendensystem gegeben sind, wird meist durch die Größe der Gruppe erschwert und notwendige Berechnungen sind mitunter nur sehr umständlich durchführbar. Es ist daher wünschenswert, Eigenschaften wie z. B. die Mächtigkeit solcher Gruppen mittels geeigneter Algorithmen schnell bestimmen zu können.
Die Grundlage für einen solchen Algorithmus bildet das 1970 von Sims erdachte Konzept von Basis und starkem Erzeugendensystem, mit dessen Hilfe u.a. die Größe einer Permutationsgruppe berechnet und Elemente der Symmetrischen Gruppe auf Zugehörigkeit zu dieser Gruppe überprüft werden können. Die Vorstellung dieses Konzeptes sowie des darauf basierenden sogenannten Schreier-Sims-Algorithmus zur Berechnung von Basen und starken Erzeugendensystemen ist Gegenstand des dritten Kapitels dieser Arbeit.
Im Laufe der Zeit wurde der Schreier-Sims-Algorithmus zur Behandlung unterschiedlicher Problemstellungen mehrfach modifiziert. So stellte z. B. Leon 1980 in [3] einen randomisierten Schreier-Sims-Algorithmus vor, welcher in den meisten Fällen schneller ein Ergebnis liefert als die in Kapitel 3 vorgestellte Variante. Der Nachteil der Randomisierung ist jedoch, dass das Ergebnis mit einer gewissen Wahrscheinlichkeit falsch sein kann, wodurch sich wiederum die Notwendigkeit von Verifikationsalgorithmen ergibt. Sowohl der randomisierte Algorithmus als auch eine Methode zur Verifikation werden in Kapitel 4 behandelt.
Die heute zur Verfügung stehenden Prozessoren sind weitaus leistungsfähiger als zu der Zeit, in welcher der Schreier-Sims-Algorithmus entwickelt wurde. So besitzen die meisten Prozessoren heute neben höheren Taktraten zumeist auch die Fähigkeit, mehrere Berechnungen gleichzeitig auszuführen. Die Nutzung möglichst aller durch den Prozessor zur Verfügung gestellten Ressourcen mit dem Ziel der Verkürzung der Laufzeit der Verifikation wird in Kapitel 5 thematisiert. Es werden konkret zwei Ansätze zur Parallelisierung vorgestellt und zur Bewertung dieser hinsichtlich ihres Laufzeitvorteils wurden Laufzeitmessungen vorgenommen und ausgewertet. Dazu in GAP vorgenommene Implementierungen der in dieser Arbeit vorgestellten Algorithmen sowie Informationen zu deren Benutzung befinden sich im Anhang und auf der beiliegenden CD.

2. Grundlagen und allgemeine Bemerkungen

Bevor auf den Schreier-Sims-Algorithmus eingegangen werden kann, müssen zunächst einige Begriffe eingeführt und Notation festgelegt werden. Vorausgesetzt werden lediglich grundlegende Kenntnisse in Gruppentheorie, wie sie üblicherweise in der Linearen Algebra vermittelt werden.
Zunächst wird definiert, was es bedeutet, wenn eine Gruppe auf einer Menge **operiert**.

Definition. *Es seien G eine Gruppe und $\Omega \neq \emptyset$ eine Menge. G **operiert** auf Ω genau dann, wenn jedes Element $g \in G$ eine Permutation σ_g auf Ω bewirkt, welche die für alle $\omega \in \Omega$ und $g, h \in G$ die folgenden Eigenschaften besitzt:*

(1) $\omega^{\sigma_{1_G}} = \omega$ *und*

(2) $(\omega^{\sigma_g})^{\sigma_h} = \omega^{\sigma_{g \cdot h}}$.

Die Gruppe aller Permutationen auf Ω wird mit S_Ω bezeichnet.
Sind $n \in \mathbb{N}$ und $\Omega = \{1, \ldots, n\}$, so wird S_n anstatt S_Ω geschrieben. Weiter wird für alle $g \in G$ und alle $\omega \in \Omega$ von jetzt an kurz $w^g := w^{\sigma_g}$ geschrieben.

Definition. *Es sei G eine Gruppe, welche auf der Menge $\Omega \neq \emptyset$ operiert. Weiter sei $\alpha \in \Omega$. Die Menge $\alpha^G := \{\alpha^g \mid g \in G\}$ wird als **Bahn** oder **Orbit** von α unter G bezeichnet und die Anzahl der Elemente in α^G heißt **Länge der Bahn** α^G oder kurz **Bahnenlänge**. G operiert **transitiv** auf Ω genau dann, wenn für alle $\alpha, \beta \in \Omega$ ein $g \in G$ mitjenes Element an das Ende einer Liste angefügt wird. der Eigenschaft $\alpha^g = \beta$ exisitiert.*

Definition. *Es operiere die Gruppe G auf $\Omega \neq \emptyset$ und es sei $\alpha \in \Omega$. Dann heißt $G_\alpha := \{g \in G \mid \alpha^g = \alpha\}$ der **Stabilisator** von α in G-*

Definition. *Es sei Ω eine Menge mit $n \in \mathbb{N}$ Elementen. Weiter sei G eine Gruppe, welche auf Ω operiert. Ist 1_G das einzige Element aus G, welches alle $\omega \in \Omega$ fixiert, so heißt G eine **Permutationsgruppe** auf Ω.*

Definition. *Seien G eine auf $\Omega \neq \emptyset$ operierende Gruppe und $r \in \mathbb{N}$. G operiert **r-fach transitiv** auf Ω genau dann, wenn für alle $(\omega_1, \ldots, \omega_r), (\mu_1, \ldots, \mu_r) \in \Omega^r$ mit den Eigenschaften $\omega_i \neq \omega_j$ und $\mu_i \neq \mu_j$ für alle $i, j \in \{1, \ldots, r\}$, $i \neq j$ ein $g \in G$ existiert so, dass $\omega_l^g = \omega_l^g$ für jedes $l \in \{1, \ldots, r\}$ gilt.*

Bemerkungen.

(1) Für jedes $n \in \mathbb{N}$ operiert die Symmetrische Gruppe S_n n-fach transitiv auf $\Omega = \{1, \ldots, n\}$.

(2) Sind $n, r \in \mathbb{N}$ und operiert eine Permutationsgruppe G r-fach transitiv auf $\Omega = \{1, \ldots, n\}$, so operiert G für jedes $j \in \{1, \ldots, r-1\}$ j-fach transitiv auf Ω.

Lemma 2.1. Es seien wieder $r \in \mathbb{N}$, $\Omega \neq \emptyset$ und G eine r-fach transitiv auf Ω operierende Gruppe. Weiter seien $j \in \{1, \ldots, r-1\}$ und $\omega_1, \ldots, \omega_j \in \Omega$, wobei $\omega_i \neq \omega_j$ für alle $i, j \in \{1, \ldots, n\}$, $i \neq j$ gelte. Dann operiert der Stabilisator $G_{\omega_1,\ldots,\omega_j}$ transitiv auf $\Omega \backslash \{\omega_1, \ldots, \omega_j\}$.

Beweis. Es seien $\alpha, \beta \in \Omega \backslash \{\omega_1, \ldots, \omega_j\}$. Da G r-fach transitiv auf Ω operiert und $j < k$ ist, operiert G auch $(j+1)$-fach transitiv auf Ω. Somit existiert ein $g \in G$ derart, dass

$$(w_1, \ldots, w_j, \alpha)^g = (w_1^g, \ldots, w_j^g, \alpha^g) = (w_1, \ldots, w_j, \beta)$$

ist. Es gelten also $g \in G_{w_1,\ldots,w_j}$ und $\alpha^g = \beta$. Da insbesondere β beliebig war, operiert $G_{w_1,\ldots,w_j}$ transitiv auf $\Omega \backslash \{\omega_1, \ldots, \omega_j\}$. □

In dieser Arbeit werden verschiedene Algorithmen thematisiert und mittels Pseudocode dargestellt. Dabei gelten bestimmte Konventionen, welche in der nun folgenden Anmerkung zusammengefasst werden.

Bemerkungen zum Pseudocode.

(1) Bei jedem Algorithmus wird stets angegeben, welche Daten ihm als Eingabedaten, kurz **Input**, übergeben werden. Wird ein Ergebnis zurückgegeben, so wird dies als **Output** vermerkt. Dieser ist optional, was bedeutet, dass es auch Algorithmen gibt, welche lediglich Änderungen an den Eingabedaten vornehmen.

(2) Bedingte Anweisungen werden standardmäßig mittels ***if*** und ***else*** gekennzeichnet. Folgt dabei auf ein *if* kein *else*, so wird bei Nichterfüllung der bei *if* gestellten Bedingung nichts getan.

(3) In Anlehnung an die Schreibweise von Listen in GAP werden im Pseudocode eckige Klammern '[...]' verwendet, um Tupel und auch Mengen darzustellen. Betrachtet man eine Liste *list*, so erfolgt der Zugriff auf die Elemente dieser durch *list[i]*, wobei $i \in \mathbb{N}$ und maximal so groß wie die Anzahl der Elemente in *list* ist. Die Anweisung *Add(list, Element)* bedeutet, dass die Liste *list* um den Eintrag *Element* erweitert wird.

(4) Mittels Schleifen, welche durch ***for*** oder ***while*** eingeleitet werden, kann zur Ausführung von Anweisungen über die Elemente einer Liste iteriert werden. Dabei ist es möglich, innerhalb einer Schleife die Liste zu verändern, über welche iteriert wird. [1]

Um die Laufzeit von Algorithmen mit wachsendem $n \in \mathbb{N}$ einschätzen zu können, wird die $\mathcal{O}-$Notation verwendet, welche nun definiert wird.

Definition. *Es seien* $f := (x_i)_{i\in\mathbb{N}}$ *und* $g := (y_j)_{j\in\mathbb{N}}$ *reelle Zahlenfolgen. Dann ist* $f \in \mathcal{O}(g)$ *genau dann, wenn es Konstanten* $c \in \mathbb{R},\ c > 0$ *und* $n_0 \in \mathbb{N}$ *gibt so, dass*

$$x_n \leq c \cdot y_n$$

für alle $n \in \mathbb{N},\ n \geq n_0$ *gilt.*

[1] Dies kann unter anderem in der zugehörigen GAP-Dokumentation auf http://www.gap-system.org/Manuals/doc/ref/chap4.html#X78783E777867638A (Zugriff: 21.08.16) unter Punkt 4.20 nachgelesen werden.

3. Der Schreier-Sims-Algorithmus

In diesem Kapitel soll nun durch Betrachtung der theoretischen Hintergründe die Funktionsweise des Schreier-Sims- Algorithmus' erläutert werden. Dafür werden zunächst einige Begriffe und Bezeichnungen eingeführt und im Anschluss wird dann auf die algorithmische Umsetzung einiger grundlegender Problemstellungen – wie die Berechnung von Bahnen – eingegangen, wobei sich der angegebene Pseudocode an einer Implementierung in GAP orientiert.

Generalvoraussetzung I. Es seien $n \in \mathbb{N}$ und $G \leq S_n$ eine Permutationsgruppe, welche auf $\Omega := \{1, \ldots, n\}$ operiert. Weiter seien $m \in \mathbb{N}$ und $S := \{x_1, \ldots, x_m\} \subseteq G$ so, dass $G = \langle S \rangle$ gilt.

3.1. Berechnung von Bahnen und Transversalen

Bei der Durchführung des Schreier-Sims-Algorithmus' müssen immer wieder Bahnen von Elementen aus Ω unter der Operation von G berechnet werden, weswegen es zweckmäßig ist, sich zuerst mit der Umsetzung eines Verfahrens zur Berechnung dieser zu beschäftigen.

Verfahren 3.1.1 (Praktische Berechnung von Bahnen).
Es gelte Voraussetzung (I) und es sei $\alpha \in \Omega$. Zu Beginn sei $O := \{\alpha\}$. Für alle $\beta \in O$ und alle $i \in \{1, \ldots, m\}$ werden nacheinander die folgenden Schritte durchgeführt:

(1) Berechne β^{x_i}.

(2) (a) Gilt $\beta^{x_i} \in O$, so ist nichts zu zun.

(b) Gilt hingegen $\beta^{x_i} \notin O$, so füge β^{x_i} zu O hinzu.

Das Verfahren wird so lange ausgeführt, bis O vollständig durchlaufen wurde und keine Elemente mehr hinzugefügt werden müssen.

Satz 3.1.2. *Es gelte Voraussetzung (I). Weiter seien $\alpha \in \Omega$ und O die gemäß Verfahren (3.1.1) berechnete Menge. Dann ist $O = \alpha^G$.*

Beweis. Für jedes $i \in \{1, \ldots, m\}$ ist laut Voraussetzung (I) bereits $x_i \in S \subseteq G$. Demnach gilt für alle $\beta \in O$ in Verfahren (3.1.1) $\beta^{x_i} \in \alpha^G$, woraus zunächst $O \subseteq \alpha^G$ folgt. Zum Beweis der Gleichheit sei $x \in G$. Weil $G = \langle S \rangle$ ist, gibt es ein $p \in \mathbb{N}$, Indizes $i_1, \ldots, i_p \in \{1, \ldots, m\}$ und Elemente $x_{i_1}, \ldots, x_{i_p} \in S$ so, dass $x = x_{i_1} \cdot x_{i_2} \cdots x_{i_p}$ gilt.

Betrachtet man nun α^x, so sieht man, dass für alle $j \in \{1, \ldots, p\}$ schon $\alpha^{x_{i_1} \cdots x_{i_j}} \in O$ ist. Daher ergibt sich $\alpha^x \in O$ und somit letztlich $O = \alpha^G$. Da sowohl G als auch Ω endlich sind, terminiert Verfahren (3.1.1), wenn O vollständig durchlaufen wurde und kein weiteres Element mehr hinzugefügt werden muss. □

Definition. *Sei $\alpha \in \Omega$. Eine Teilmenge $\mathfrak{T} \subseteq G$ heißt **Transversale** von α bzgl. G genau dann, wenn $\mathfrak{T}$ für alle $\beta \in \alpha^G$ genau ein $g \in G$ enthält, sodass $\alpha^g = \beta$ gilt. Ein solches g wird als **Transversalelement** von β bzgl. α und G, kurz $T(\beta) := g$, bezeichnet. Weiter wird im Rahmen dieser Arbeit für alle Transversalen gefordert, dass stets $T(\alpha) = 1_G$ gilt.*

Beispiel 3.1.3. *Es seien $n := 3$, $G := S_3$ und $S := \{(1\,2), (1\,3)\}$. Es gelten $2^{id_\Omega} = 2$, $2^{(1\,2)} = 1$ und $2^{(1\,2\,3)} = 3$, also ist $\mathfrak{T} := \{id_\Omega, (1\,2), (1\,2\,3)\}$ eine Transversale von 2 bzgl. G.*

Das Speichern von Transversalelementen in expliziter Form ist bei hinreichend großen Gruppen jedoch alles andere als speichereffizient, da im schlimmsten Fall n Permutationen der Ordnung n gespeichert werden müssen. Eine Möglichkeit zur platzsparenden Repräsentation von Transversalen bietet die Verwendung von **Schreier-Vektoren**.

Definition. *Es gelte Voraussetzung (I). Es seien $\alpha \in \Omega$ und O die gemäß Verfahren (3.1.1) berechnete Bahn von α bzgl. G. Für alle $\gamma \in O \backslash \{\alpha\}$ seien weiter $i_\gamma \in \{1, \ldots, m\}$, $x_{i_\gamma} \in S$ und $\beta \in O$ so, dass γ als $\gamma = \beta^{x_{i_\gamma}}$ bei der Durchführung von Verfahren (3.1.1) zu O hinzugefügt wurde. Ein **Schreier-Vektor** zu α bzgl. G ist ein n-Tupel $v = (v_1, \ldots, v_n)$, welches die folgenden Eigenschaften besitzt:*

(1) Es ist $v_\alpha = -1$.

(2) Für alle $\gamma \in O \backslash \{\alpha\}$ ist $v_\gamma = i_\gamma$.

(3) Für alle $\delta \in \Omega \backslash O$ ist $v_\delta = 0$.

Beispiel 3.1.4. *Ein zur Transversale $\mathfrak{T}$ von 2 bzgl. G aus Beispiel (3.1.3) gehöriger Schreier-Vektor ist $(1, -1, 2)$.*

Auf diese Weise wird eine Transversale lediglich durch n Einträge im Bereich zwischen -1 und m repräsentiert. Der Nachteil dieser Methode ist, dass Transversalelemente bei Bedarf aus v berechnet werden müssen, so dass mit einer längeren Laufzeit gerechnet werden muss. Ausgehend von Verfahren (3.1.1) ist es nun möglich, zwei Algorithmen

ORBIT und TRANSVERSAL zur Berechnung von Bahnen sowie zugehörigen Schreier-Vektoren und Transversalen anzugeben.

Algorithmus 3.1.5: ORBIT[2]– Implementierung in (A.1.2)

Input: $\alpha \in \Omega$, Erzeugendensystem S und $n = |\Omega|$
Output: Bahn $O = \alpha^G$, Schreier-Vektor v zu α

```
1 O = [α];   v := [0, ..., 0];   v[α] := −1;
2 for β in O do
3     for i in [1, ..., |S|] do
4         if β^{x_i} ∉ O then
5             Add(O, β^{x_i}); v[β^{x_i}] := i;
6 return O, v;
```

In ORBIT besteht die Bahn O im schlimmsten Fall aus $|\Omega| = n$ Elementen, wobei nacheinander auf jedes von diesen die m Erzeuger aus S angewendet werden müssen. Dies führt insgesamt zu einer Laufzeitkomplexität von $\mathcal{O}(n \cdot m)$.
Wie bereits erwähnt, ist es hinsichtlich des benötigten Speicherplatzes nicht praktikabel, die Transversale zu einem Element aus Ω einmalig zu berechnen und als Menge von Permutationen zu speichern. Stattdessen werden die Transversalelemente bei Bedarf von TRANSVERSAL aus den Schreier-Vektoren berechnet. Unter der Annahme[3], dass die Multiplikation zweier Zyklen im schlimmsten Fall eine asymptotische Laufzeit von $\mathcal{O}(n^2)$ besitzt, ergibt sich für TRANSVERSAL eine Komplexität von $\mathcal{O}(n^3)$.

Algorithmus 3.1.6: TRANSVERSAL[2] – Implementierung in (A.1.3)

Input: $\beta \in \Omega$, Schreier-Vektor v zu $\alpha \in \Omega$ und Erzeugendensystem S
Output: Transversalelement $T(\beta)$, falls $v[\beta] \neq 0$ ist. Sonst `false`.

```
1 if v[β] = 0 then
2     return false;
3 else
4     T(β) := 1_G; k := v[β];
5     while k ≠ −1 do
6         T(β) := x_k · T(β);   β := β^{x_k^{−1}};   k := v[β];
7     return T(β)
```

[3]Es konnte nicht in Erfahrung gebracht werden, wie die Multiplikation zweier Zyklen in GAP realisiert wird. Da dies in maximal quadratischer Laufzeit möglich ist, wird dies hier angenommen.
[3]Dieser Pseudocode wurde aus [1], S.80 übernommen.

3.2. Basen und starke Erzeugendensysteme

Grundlegend für den Schreier-Sims-Algorithmus ist das von Sims erdachte Konzept von Basis und starkem Erzeugendensystem, auf welches im Folgenden eingegangen werden soll. Die dabei verwendete Notation wird wieder in einer Generalvoraussetzung festgelegt.

Generalvoraussetzung II. Es gelte Voraussetzung (I). Zusätzlich seien $k \in \mathbb{N}$, $k \leq n$ und $B := (b_1, \ldots, b_k) \in \Omega^k$. Für alle $i \in \{2, \ldots, k+1\}$ seien Stabilisatoren $G^{(i)}$ definiert durch

$$G^{(i)} := \{g \in G \mid \text{für alle } j \in \{1, \ldots, i-1\} \text{ ist } b_j^g = b_j\}$$

und es sei $G^{(1)} := G$. Die Anzahl der Einträge von B soll kurz mit $|B| := k$ bezeichnet werden.

Definition. *Es gelte Voraussetzung (II). B heißt **Basis** für G genau dann, wenn $G^{(k+1)} = \{1_G\}$ gilt, also wenn 1_G das einzige Element in G ist, welches B fixiert.*
*Ist B eine Basis, so heißt ein Erzeugendensystem S von G **stark** bzgl. B genau dann, wenn S Erzeugendensysteme für jeden Stabilisator $G^{(i)}$ enthält, also*

$$G^{(i)} = \langle S \cap G^{(i)} \rangle$$

für alle $i \in \{1, \ldots, k+1\}$ gilt.

Beispiele 3.2.1. *Es gelte Voraussetzung (II).*

(1) Es sei $G := S_n$. Dann muss eine Basis B für G mindestens in Ω^{n-1} liegen, denn anderenfalls gäbe es $p, q \in \Omega \backslash \{b_1, \ldots, b_k\}$ und eine Transposition $(p\ q) \in S_n = G$, welche B fixiert. Umgekehrt ist jede $(n-1)$-elementige Teilmenge von Ω bereits eine Basis für S_n. Insbesondere sieht man hier, dass Basen von Gruppen nicht eindeutig bestimmt sein müssen.

(2) Nun sei $G := D_{2n}$ die Diedergruppe auf $n \geq 3$ Elementen. Dann ist $B = (1, 2)$ eine Basis für G.

Zur Begründung dessen ist es sinnvoll, G als Symmetriegruppe eines regelmäßigen n-Ecks zu betrachten, bestehend aus jeweils n Drehungen und Spiegelungen, wobei die Ecken mit $1, \ldots, n$ nummeriert seien. Jede Drehung, welche das Element 1 fixiert, ist bereits trivial. Also wird 1 nur von der Spiegelung aus G fixiert, deren Spiegelachse durch Ecke 1 verläuft. Da $n \geq 3$ vorausgesetzt wurde, existiert keine Spiegelachse durch Ecke 1 und die benachbarte Ecke 2. Daher ist 1_G das einzige Element in G, welches 1 und 2 fixiert.

(3) Es soll nun die Symmetriegruppe $\mathcal{R}$ eines Zauberwürfels der Größe $3 \times 3 \times 3$ betrachtet werden. Dabei werden die bewegbaren Teilflächen des Würfels wie in Abbildung (1) nummeriert und durch Drehung der Seitenflächen top, left, front, right, back und bottom permutiert.

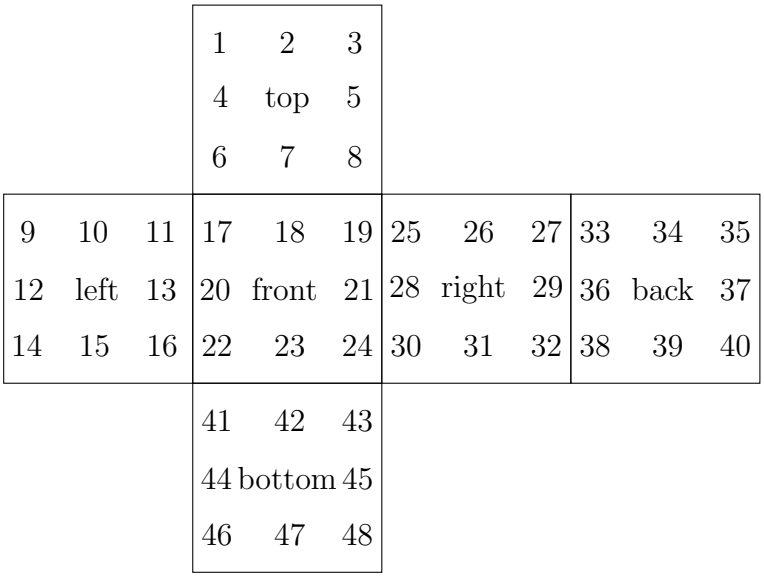

Abbildung 1: Nummerierung der Teilflächen des Zauberwürfels

Diese insgesamt sechs Drehungen bilden ein Erzeugendensystem S für $\mathcal{R} \leq S_{48}$, welches gemäß Abbildung (1) in Zyklenschreibweise folgendermaßen angegeben werden kann:

$$\begin{aligned} S \ = \{&(1\ 3\ 8\ 6)(2\ 5\ 7\ 4)(9\ 33\ 25\ 17)(10\ 34\ 26\ 18)(11\ 35\ 27\ 19),\\ &(9\ 11\ 16\ 14)(10\ 13\ 15\ 12)(1\ 17\ 41\ 40)(4\ 20\ 44\ 37)(6\ 22\ 46\ 35),\\ &(17\ 19\ 24\ 22)(18\ 21\ 23\ 20)(6\ 25\ 43\ 16)(7\ 28\ 42\ 13)(8\ 30\ 41\ 11),\\ &(25\ 27\ 32\ 30)(26\ 29\ 31\ 28)(3\ 38\ 43\ 19)(5\ 36\ 45\ 21)(8\ 33\ 48\ 24),\\ &(33\ 35\ 40\ 38)(34\ 37\ 39\ 36)(3\ 9\ 46\ 32)(2\ 12\ 47\ 29)(1\ 14\ 48\ 27),\\ &(41\ 43\ 48\ 46)(42\ 45\ 47\ 44)(14\ 22\ 30\ 38)(15\ 23\ 31\ 39)(16\ 24\ 32\ 40)\}. \end{aligned}$$

Man kann zeigen, dass

$$B = (14, 6, 3, 1, 2, 4, 5, 7, 12, 8, 13, 15, 16, 21, 23, 24, 29, 31)$$

eine Basis für $\mathcal{R}$ ist, wobei dies zugegeben nicht offensichtlich und der Nachweis dessen mit großem Rechenaufwand verbunden ist. Um eben jenen Rechenaufwand zu umgehen, bedarf es entsprechender Algorithmen zur Berechnung von Basen.

Die Stabilisatoren $G^{(i)}$ bilden per Definition eine absteigende Kette von Untergruppen

$$G = G^{(1)} \geq G^{(2)} \geq \cdots \geq G^{(k)} \geq G^{(k+1)}$$

und für jedes $i \in \{1, \ldots, k\}$ gilt $G^{(i)}_{b_i} = \{g \in G^{(i)} \mid b_i^g = b_i\} = G^{(i+1)}$. Die Kenntnis einer Basis für G mitsamt zugehöriger Stabilisatorkette liefert einige nützliche Informationen über die Gruppe G, wie die folgenden Lemmata zeigen.

Lemma 3.2.2. *Es gelte Voraussetzung (II) und B sei eine Basis für G. Für jedes $i \in \{1, \ldots, k\}$ sei mit $\mathfrak{T}_i$ ein Repräsentantensystem für die Rechtsnebenklassen von $G^{(i+1)}$ in $G^{(i)}$ bezeichnet. Dann existieren zu jedem $g \in G$ eindeutig bestimmte Transversalelemente $t_1 \in T_1, \ldots, t_k \in T_k$ so, dass gilt:*

$$g = t_k \cdot t_{k-1} \cdots t_2 \cdot t_1.$$

Beweis. Es seien $i \in \{1, \ldots, k\}$ und $g \in G^{(i)}$. Da $\mathfrak{T}_i$ ein Repräsentantensystem der Rechtsnebenklassen von $G^{(i+1)}$ in $G^{(i)}$ ist, gibt es ein eindeutig bestimmtes $t_i \in \mathfrak{T}_i$ so, dass $g \in G^{(i+1)} \cdot t_i$ gilt. Das bedeutet wiederum, dass ein $h_i \in G^{(i+1)}$ mit der Eigenschaft $g = h_i \cdot t_i$ existiert. Ist $\tilde{h}_i \in G^{(i+1)}$ ein weiteres Element mit $g = \tilde{h}_i \cdot t_i$, so folgt schon $h_i = \tilde{h}_i$.
Sei nun $g \in G = G^{(1)}$. Es existieren also $h_1 \in G^{(2)}$ und $t_1 \in \mathfrak{T}_1$ so, dass man $g = h_1 \cdot t_1$ erhält. Die wiederholte Anwendung obigen Arguments auf h_1 und alle weiteren h_i liefert, dass es Elemente $t_i \in \mathfrak{T}_i$ und $h_k \in G^{(k+1)}$ gibt derart, dass

$$g = h_k \cdot t_k \cdot t_{k-1} \cdots t_2 \cdot t_1$$

ist. Da B eine Basis für G ist, gilt $G^{(k+1)} = \{1_G\}$ und damit $h_k = 1_G$. Somit wurde eine Zerlegung von g in Elemente $t_i \in \mathfrak{T}_i$ gefunden. □

Zur Berechnung einer solchen Zerlegung ist es nun von Interesse, solche Repräsentantensysteme $\mathfrak{T}_i$ zu bestimmen. Dass dies bereits mit den Mitteln aus Abschnitt (3.1) möglich ist, soll nun gezeigt werden.

Lemma 3.2.3. *Es gelte Voraussetzung (I). Für jedes $\alpha \in \Omega$ seien $R_\alpha := \{G_\alpha \cdot g \mid g \in G\}$ und $\varphi : R_\alpha \to \alpha^G$ eine Abbildung, die wie folgt definiert ist: für alle $g \in G$ sei $(G_\alpha \cdot g)^\varphi := \alpha^g$. Dann ist φ eine Bijektion von R_α auf die Bahn α^G.*

Beweis. Sei $\alpha \in \Omega$. Es wird nun nacheinander gezeigt, dass φ eine wohldefinierte und bijektive Abbildung ist.

Für den Beweis der Wohldefiniertheit seien $g, h \in G$ so gewählt, dass $G_\alpha \cdot g = G_\alpha \cdot h$ gilt. Das bedeutet $g \cdot h^{-1} \in G_\alpha$ und es folgen

$$\alpha^{g \cdot h^{-1}} = (G_\alpha \cdot (g \cdot h^{-1}))^\varphi = (G_\alpha \cdot 1_G)^\varphi = \alpha$$

sowie $(G_\alpha \cdot g)^\varphi = \alpha^g = \alpha^h = (G_\alpha \cdot h)^\varphi$. Also ist φ wohldefiniert.
Seien nun $x, y \in G$ so, dass $\alpha^x = \alpha^y$ gilt. Daraus folgen wiederum $\alpha^{x \cdot y^{-1}} = \alpha$ und $x \cdot y^{-1} \in G_\alpha$. Man erhält daher $G_\alpha \cdot x = G_\alpha \cdot y$ und das bedeutet, dass φ injektiv ist.
Betrachtet man $\beta \in \alpha^G$, so gibt es ein $z \in G$ mit der Eigenschaft $\beta = \alpha^z = (G_\alpha \cdot z)^\varphi$ und damit ist φ surjektiv. □

Mittels des vorangegangenen Lemmas folgt nun sofort, dass die Repräsentantensysteme $\mathfrak{T}_i$ für die Rechtsnebenklassen der Stabilisatoren $G^{(i+1)} = G^{(i)}_{b_i}$ aus Lemma (3.2.2) ebenfalls Transversalen von b_i bzgl. $G^{(i)}$ sind und Algorithmen zur Berechnung dieser wurden bereits im vorangegangenen Abschnitt (3.1) behandelt.
Bevor auf die praktische Bestimmung der Zerlegung aus Lemma (3.2.2) eingegangen wird, werden zunächst noch die zwei folgenden Resultate bewiesen.

Lemma 3.2.4. *Es gelte wieder Voraussetzung (II), wobei B eine Basis für G sei. Weiter sei für jedes $i \in \{1, \ldots, k\}$ mit $\mathfrak{T}_i$ eine Transversale von b_i bzgl. $G^{(i)}$ bezeichnet. Dann gelten:*

(a) $|G| = |\mathfrak{T}_1| \cdot |\mathfrak{T}_2| \cdots |\mathfrak{T}_{k-1}| \cdot |\mathfrak{T}_k|$

(b) Jedes $g \in G$ ist eindeutig durch $B^g := (b_1^g, \ldots, b_k^g)$ bestimmt.

Beweis. Die Aussage in (a) erhält man sofort mittels der aus Lemma (3.2.3) folgenden Übereinstimmung von Repräsentantensystemen und Transversalen und der eindeutigen Zerlegung von Elementen aus G in Transversalelemente gemäß Lemma (3.2.2).
Für (b) seien $g, h \in G$ so gewählt, dass für alle $i \in \{1, \ldots, k\}$ $b_i^g = b_i^h$ gilt. Daraus folgt $b_i^{gh^{-1}} = b_i$ für alle $i \in \{1, \ldots, k\}$ und $g \cdot h^{-1} \in G^{(k+1)}$. Da B eine Basis für G ist, gilt nun $g \cdot h^{-1} = 1_G$ und somit $g = h$. □

Die erste Aussage des vorherigen Lemmas beantwortet also die Frage, wie mit Hilfe einer Basis B für G die Größe von G berechnet werden kann. Teil (b) liefert zudem eine speichereffiziente Möglichkeit der Repräsentation von Elementen $g \in G$ durch die Bilder B^g, sofern $|B|$ wesentlich kleiner als $|\Omega|$ ist. Es ist außerdem möglich, für Elemente $h \in S_n$ ein Verfahren zur Überprüfung der Existenz der in Lemma (3.2.2) angegebenen Zerlegung unter Verwendung von B^h anzugeben.

Verfahren 3.2.5 (Praktische Bestimmung der Zerlegung in Transversalelemente).
Es gelte Voraussetzung (II), wobei B eine Basis für G sei. Für alle $i \in \{1, \dots, k\}$ sei mit $\mathfrak{T}_i$ eine Transversale von b_i bzgl. $G^{(i)}$ bezeichnet. Ferner sei $g \in G$. Dann existiert laut Lemma (3.2.2) eine eindeutige Zerlegung $g = t_k \cdot t_{k-1} \cdots t_2 \cdot t_1$ von g in Transversalelemente $t_i \in \mathfrak{T}_i$.
Setze $h_1 := g$. Es gilt dann $t_2, \dots, t_k \in G^{(2)}$ und man erhält somit

$$b_1^{h_1} = b_1^g = b_1^{t_k \cdots t_2 \cdot t_1} = b_1^{t_1}.$$

Daher ist t_1 das Transversalelement von $b_1^{h_1}$ bzgl. b_1 und $G^{(1)}$, mit der Notation aus Abschnitt (3.1) ist also $t_1 = T_1(b_1^{h_1})$. Setze nun $h_2 := g \cdot t_1^{-1}$. Damit folgt $b_1^{h_2} = b_1$ und es gilt $h_2 \in G^{(2)}$. Weiter sind $t_3, \dots, t_k \in G^{(3)}$. Es ergibt sich wiederum $b_2^{h_2} = b_2^{t_k \cdots t_2} = b_2^{t_2}$ und $b_2^{h_3} = b_2$ mit $t_2 = T_2(b_2^{h_1})$ und $h_3 := g \cdot t_1^{-1} \cdot t_2^{-1} \in G^{(3)}$.
Die Wiederholung dieses Argumentes liefert Elemente mit $h_1 := g$ und $h_i := g \cdot t_1^{-1} \cdots t_{i-1}^{-1}$ für $i \in \{2, \dots, k\}$ und für jedes $p \in \{1, \dots, k\}$ die Vorschrift

$$t_p = T_p(b_p^{h_p}).$$

zur rekursiven Berechnung der Transversalelemente t_p. Insbesondere gilt $h_p \in G^{(p)}$.

Sind also eine Basis für G und ein starkes Erzeugendensystem bekannt, so kann mittels Verfahren (3.2.5) für jedes $g \in S_n$ die Existenz einer Zerlegung gemäß Lemma (3.2.2) nachgeprüft werden. Darauf aufbauend kann nun mit Hilfe von Orbit und Transversal aus Abschnitt (3.1) der folgende Algorithmus Strip angegeben werden.

Algorithmus 3.2.6: Strip[4]– Implementierung in (A.3.3)

Input: $g \in S_n$, k-Tupel $B = (b_1, \dots, b_k) \in \Omega^k$, Erzeugendensystem $\tilde{S}$ von G, Bahnen $O_i := b_i^{G^{(i)}}$ und Schreier-Vektoren v_i zu b_i bzgl. $G^{(i)}$, $i \in \{1, \dots, |B|\}$
Output: $h \in S_n$ und $j \in \{1, \dots, |B| + 1\}$

```
1  h := g;
2  for j in [1,...,|B|] do
3      b := b_j^h;
4      if b ∉ O_j then
5          return h, j;
6      else
7          t_j := Transversal(b, v_j, S̃ ∩ G^(j));   h := h · t_j^{-1};
8  return h, |B| + 1;
```

[4]Dieser Pseudocode wurde aus [1], S.89 übernommen.

Bemerkungen.

(1) Sind die Voraussetzungen von Lemma (3.2.2) verletzt, ist also B keine Basis oder ist $\tilde{S}$ kein starkes Erzeugendensystem, so kann es passieren, dass für ein $g \in G$ keine vollstädnige Zerlegung in Transversalelemente berechnet werden kann. In diesem Fall konnten nur bricht Algorithmus (3.2.6) in Zeile 4 vorzeitig ab.

(2) Das von Algorithmus (3.2.6) zurückgegebene $h \in S_n$ ist der bei der Zerlegung von g in Transversalelemente verbleibende Rest, es gilt

$$g = h \cdot t_{j-1} \cdots t_1.$$

Diese Rest ist vor allem dann interessant, wenn die Voraussetzungen von Lemma (3.2.2) verletzt sind, siehe Bemerkung (1). Dabei liefert der zweite Rückgabewert $j \in \{1, \ldots, k+1\}$ die Information $h \in G^{(j)}$.

Definition. *Es gelte Voraussetzung (II) und es sei $g \in S_n$. Weiter seien $h \in S_n$ und $j \in \{1, \ldots, k+1\}$ die Ergebnisse der Anwendung von Algorithmus (3.2.6) auf g, B und S. Dann wird ab nun h als das **Residuum von g vom Typ j** bzgl. B und S bezeichnet.*

Zusätzlich zur Berechnung der Zerlegung von Elementen aus G in Transversalelemente liefert Algorithmus (3.2.6) die Möglichkeit, Elemente $g \in S_n$ auf Zugehörigkeit zu G zu überprüfen.

Lemma 3.2.7. *Es gelte Voraussetzung (II) und es seien $g \in S_n$ und h das Residuum von g vom Typ j bzgl. B und $\tilde{S}$. Gelten dann $h = 1_G$ und $j = k+1$, so ist $g \in G$. Ist zusätzlich B eine Basis und ist $\tilde{S}$ ein starkes Erzeugendensystem, so folgen aus $g \in G$ schon $h = 1_G$ und $j = k+1$.*

Beweis. Zuerst seien $h = 1_G$ und $j = k+1$. Gemäß Verfahren (3.2.5) wurden also eindeutig bestimmte Transversalelemente $t_1, \ldots, t_k \in G$ berechnet so, dass gilt

$$g = h \cdot t_k \cdots t_1 = 1_G \cdot t_k \cdots t_1 \in G.$$

Nun seien $g \in G$, B eine Basis für G und $\tilde{S}$ ein starkes Erzeugendensystem bzgl B. Laut Lemma (3.2.2) existieren wieder $t_1, \ldots, t_k \in G$ so, dass $g = t_k \cdots t_1$ ist. Die Durchführung von Algorithmus (3.2.6) liefert letztlich $h = g \cdot t_1^{-1} \cdots t_k^{-1} = 1_G$ und $j = k+1$, wie gewünscht. □

Bemerkung. Im Falle $g \in S_n \backslash G$ liefert Algorithmus (3.2.6) laut vorangegangenem Lemma also entweder $j \leq k$ oder $h \neq 1_G$ und $j = k+1$ zurück.

Im nächsten Abschnitt wird sich zeigen, dass STRIP auch dazu verwendet werden kann, eine Basis für G und ein zugehöriges starkes Erzeugendensystem zu konstruieren.

3.3. Schreiers Untergruppensatz und der Schreier-Sims-Algorithmus

Interessant ist nun der Fall, in dem nicht notwendigerweise B eine Basis für G und S ein zugehöriges starkes Erzeugendensystem bilden. Wünschenswert wäre es, B und S so anzupassen, dass sie den Anforderungen genügen. Ob und wie dies funktioniert, soll im Folgenden geklärt werden.

Generalvoraussetzung III. Es gelte Voraussetzung (II). Weiter sei $\tilde{S} \subseteq G$ ein Erzeugendensystem von G mit der Eigenschaft $S \subseteq \tilde{S}$. Für alle $i \in \{1, \ldots, k+1\}$ sei $\tilde{S}^{(i)} := \tilde{S} \cap G^{(i)}$. Zur Unterscheidung von den tatsächlichen Stabilisatoren $G^{(i)}$ seien die von den $\tilde{S}^{(i)}$ erzeugten Untergruppen mit $H^{(i)} := \langle \tilde{S}^{(i)} \rangle$ bezeichnet.

Es werden zunächst Bedingungen an $\tilde{S}^{(i)}$ und $H^{(i)}$ benötigt derart, dass B eine Basis für G und $\tilde{S}$ ein zugehöriges starkes Erzeugendensystem ist. Solche Bedingungen liefert das nächste Lemma.

Lemma 3.3.1. *Es gelte Voraussetzung (III). B ist eine Basis für G mit zugehörigem starkem Erzeugendensystem $\tilde{S}$ genau dann, wenn*

$$H^{(k+1)} = \{1_G\} \text{ und } H_{b_i}^{(i)} = H^{(i+1)}$$

für alle $i \in \{1, \ldots, k\}$ gelten.

BEWEIS. Zuerst seien B eine Basis für G und $\tilde{S}$ ein zugehöriges starkes Erzeugendensystem. Dann gilt per Definition $H^{(i)} = G^{(i)}$ für alle $i \in \{1, \ldots, k+1\}$ und insbesondere ist $H^{(k+1)} = \{1_G\}$. Weiterhin folgt mit der Eigenschaft $G_{b_i}^{(i)} = G^{(i+1)}$, dass $H_{b_i}^{(i)} = H^{(i+1)}$ ist.
Umgekehrt seien nun $H_{b_i}^{(i)} = H^{(i+1)}$ und $H^{(k+1)} = \{1_G\}$. Aus

$$H^{(1)} = \langle \tilde{S}^{(1)} \rangle = \langle \tilde{S} \cap G^{(1)} \rangle = \langle \tilde{S} \cap G \rangle = G = G^{(1)}$$

folgt induktiv wegen $H_{b_i}^{(i)} = H^{(i+1)}$, dass für alle $i \in \{1, \ldots, k+1\}$ $H^{(i)} = G^{(i)}$ gilt, also auch $G^{(k+1)} = \{1_G\}$. □

Die Bedingung $H^{(k+1)} = \{1_G\}$ wird dadurch gesichert, dass kein Element aus S alle EInträge in B als Fixpunkte hat. Ist dem nicht so, dann muss B um ein oder mehrere Elemente aus Ω so erweitert werden, dass die Bedingung erfüllt ist. Die Sicherstellung von $H_{b_i}^{(i)} = H^{(i+1)}$ erfordert etwas mehr Überlegung und bildet den Hauptbestandteil des Schreier-Sims-Algorithmus'. Aus der Konstruktion der $H^{(i)}$ folgt zunächst, dass $H^{(i+1)} \leq H_{b_i}^{(i)}$ gilt. Erforderlich ist jedoch die Gleichheit, deren Überprüfung durch die Unkenntnis von $H_{b_i}^{(i)}$ erschwert wird. Abhilfe schafft hier der folgende Satz, welcher mit Beweis in [2], S. 8 zu finden ist.

Satz 3.3.2 (Schreiers Untergruppensatz). *Es gelte Voraussetzung (II) und es seien $U \leq G = \langle S \rangle$ und $\mathfrak{T}$ ein Repräsentantensystem der Rechtsnebenklassen von U in G, wobei $1_G \in \mathfrak{T}$ sei. Für jedes $g \in G$ sei mit $\bar{g} \in \mathfrak{T}$ der Repräsentant mit der Eigenschaft $U \cdot g = U \cdot \bar{g}$ bezeichnet. Dann gilt*

$$U = \langle t \cdot s \cdot (\overline{t \cdot s})^{-1} \mid t \in \mathfrak{T},\ s \in S \rangle.$$

Beweis. Setze $V := \{t \cdot s \cdot (\overline{t \cdot s})^{-1} \mid t \in \mathfrak{T},\ s \in S\}$. Per Definition gilt für alle $t \in \mathfrak{T}$ und alle $s \in S$ die Gleichung $U \cdot (t \cdot s) = U \cdot (\overline{t \cdot s})$, also ist $t \cdot s \cdot (\overline{t \cdot s})^{-1} \in U$ und daher $V \subseteq U$. Es genügt daher zu zeigen, dass sich jedes Element aus U als Produkt von Elementen aus V darstellen lässt. Sei dazu $u \in U$. Insbesondere ist $u \in G$, weswegen es ein $p \in \mathbb{N}$, Indizes $i_1 \ldots, i_p \in \{1, \ldots, m\}$ und Elemente $x_{i_1}, \ldots, x_{i_p} \in S$ gibt derart, dass $u = x_{i_1} \cdot x_{i_2} \cdots x_{i_p}$ ist.
Ausgehend von der Darstellung in den Erzeugern von G wird u nun als Produkt von Elementen aus V geschrieben mit der Hilfe geeignet gewählter Repräsentanten $t_i \in \mathfrak{T}$. Beginnend mit $t_1 := 1_G$ ergibt sich

$$\begin{aligned} u &= x_{i_1} \cdot x_{i_2} \cdots x_{i_p} = t_1 \cdot x_{i_1} \cdot x_{i_2} \cdots x_{i_p} \\ &= t_1 \cdot x_{i_1} \cdot \underbrace{((\overline{t_1 \cdot x_{i_1}})^{-1} \cdot \overline{t_1 \cdot x_{i_1}})}_{=1_G} \cdot x_{i_2} \cdots x_{i_p}. \end{aligned}$$

Mit der Wahl $t_2 := \overline{t_1 \cdot x_{i_1}}$ folgt weiter

$$\begin{aligned} u &= \underbrace{t_1 \cdot x_{i_1} \cdot (\overline{t_1 \cdot x_{i_1}})^{-1}}_{=:v_1} \cdot t_2 \cdot x_{i_2} \cdots x_{i_p} = v_1 \cdot t_2 \cdot x_{i_2} \cdots x_{i_p} \\ &= v_1 \cdot t_2 \cdot x_{i_2} \cdot ((\overline{t_2 \cdot x_{i_2}})^{-1} \cdot \overline{t_2 \cdot x_{i_2}}) \cdot x_{i_3} \cdots x_{i_p}. \end{aligned}$$

Geht man weiter nach diesem Prinzip vor, so erhält man letztlich

$$u = v_1 \cdots v_{p-1} \cdot t_p \cdot x_{i_p},$$

wobei $t_1 = 1_G$, $t_j = \overline{t_{j-1} \cdot x_{i_{j-1}}}$ und $v_{j-1} = t_{j-1} \cdot x_{i_{j-1}} \cdot (\overline{t_{j-1} \cdot x_{i_{j-1}}})^{-1}$ für alle $j \in \{2, \ldots, p\}$ ist. Mittels Induktion wird nun gezeigt, dass $t_j = \overline{x_{i_1} \cdots x_{i_{j-1}}}$ gilt.

IA: Es gilt $U \cdot t_2 = U \cdot (t_1 \cdot x_{i_1}) = U \cdot x_{i_1}$, also ist $t_2 = \overline{x_{i_1}}$.

IV: Die Behauptung gelte für ein $j \in \{2, \ldots, p-1\}$.

IS: Es ist $t_{j+1} = \overline{t_j \cdot x_{i_j}}$ und daher

$$U \cdot t_{j+1} = U \cdot (t_j \cdot x_{i_j}) = (U \cdot t_j) \cdot x_{i_j} \overset{IV}{=} U \cdot (x_{i_1} \cdots x_{i_{j-1}} \cdot x_{i_j}),$$

also $t_{j+1} = \overline{x_{i_1} \cdots x_{i_j}}$, wie behauptet.

Daraus folgt insbesondere, dass $\overline{t_p \cdot x_{i_p}} = \overline{x_{i_1} \cdots x_{i_p}} = \bar{u} = 1_G$ gewählt werden kann, da $u \in U$ ist. Deswegen ist $v_p := t_p \cdot x_{i_p} = t_p \cdot x_{i_p} \cdot (\overline{t_p \cdot x_{i_p}})^{-1} \in V$ und damit $u = v_1 \cdots v_p$ mit Elementen $v_1, \ldots, v_p \in V$. □

Definition. *Die Elemente des in Satz (3.3.2) angegebenen Erzeugendensystems V werden als **Schreier-Erzeuger** von U in G bezeichnet.*

Bemerkungen.

(1) Mit Hilfe von Satz (3.3.2) kann für jedes $i \in \{1, \ldots, k\}$ ein Erzeugendensystem von $H^{(i)}_{b_i} \leq H^{(i)} = \langle \tilde{S}^{(i)} \rangle$ berechnet werden. Das für die Anwendung des Satzes notwendige Repräsentantensystem der Rechtsnebenklassen von $H^{(i)}_{b_i}$ in $H^{(i)}$ erhält man wieder mit Lemma (3.2.3) als eine Transversale $\mathfrak{T}_i \subseteq H^{(i)}$ von b_i bzgl. $H^{(i)}$. Gemäß der in Satz (3.3.2) verwendeten Notation wird somit nun für ein Element $h \in H^{(i)}$ mit $\overline{h} \in \mathfrak{T}_i$ das Transversalelement bezeichnet, für welches $b_i^h = b_i^{\overline{h}}$ gilt.

(2) Um $H^{(i)}_{b_i} = H^{(i+1)}$ zu überprüfen, muss wegen $H^{(i+1)} \leq H^{(i)}_{b_i}$ lediglich die Inklusion

$$V_i := \{t \cdot s \cdot (\overline{t \cdot s})^{-1} \mid t \in \mathfrak{T}_i,\ s \in \tilde{S}^{(i)}\} \subseteq H^{(i+1)}$$

betrachtet werden. Für alle $i \in \{1, \ldots, k\}$ und jeden Schreier-Erzeuger $v \in V_i$ muss also $v \in H^{(i+1)}$ getestet werden. Dabei kann es vorkommen, dass für gewisse $t \in \mathfrak{T}_i$ und $s \in \tilde{S}^{(i)}$ schon $t \cdot s \in \mathfrak{T}_i$ ist. In diesem Fall gilt dann $t \cdot s = \overline{t \cdot s}$, der Erzeuger $t \cdot s \cdot (\overline{t \cdot s})^{-1} \in V_i$ ist also trivial und ein Enthaltenstest erübrigt sich.

Es soll nun thematisiert werden, wie die Überprüfung der Schreier-Erzeuger unter Verwendung von Algorithmus (3.2.6) Strip realisiert werden kann. Eine dazu benötigte Aussage über Schreier-Erzeuger beinhaltet das folgende Lemma.

Lemma 3.3.3. *Es gelte Voraussetzung (III), es seien $i \in \{1, \dots, k\}$ und $v \in H^{(i)}_{b_i}$ ein Schreier-Erzeuger von $H^{(i)}_{b_i}$ in $H^{(i)}$. Für jedes $p \in \{1, \dots, k\}$ sei mit $\mathfrak{T}_p \subseteq H^{(p)}$ eine Transversale von b_p bzgl. $H^{(p)}$ bezeichnet. Existieren dann Elemente $t_1 \in \mathfrak{T}_1, \dots, t_k \in \mathfrak{T}_k$ so, dass $v = t_k \cdots t_1$ ist, so gilt $t_i = t_{i-1} = \cdots = 1_G$ und damit $v = t_k \cdots t_{i+1} \in H^{(i+1)}$.*

Beweis. Da $v \in H^{(i)}_{b_i}$ ist, fixiert v für jedes $q \in \{1, \dots, i\}$ das Element b_q. Bestimmt man nun die Transversalelemente t_q wie in Verfahren (3.2.5), so erhält man gemäß der Definition von Transversalen

$$t_q = T_q(b_q^v) = T_q(b_q) = 1_G$$

und somit letztlich

$$v = t_k \cdots t_{i+1} \cdot t_i \cdots t_1 = t_k \cdots t_{i+1}.$$

Per Definition gilt $H^{(k)} \leq \cdots \leq H^{(i+1)}$ und damit folgt $v \in H^{(i+1)}$. □

Es folgt nun ein Verfahren zur Überprüfung und Anpassung von B und $\tilde{S}$ durch die Betrachtung von Schreier-Erzeugern, um letztlich eine Basis für G und ein starkes Erzeugendensystem zu erhalten.

Verfahren 3.3.4 (Überprüfung der Schreier-Erzeuger).
Es gelte Voraussetzung (III). Weiter sei für jedes $i \in \{1, \dots, k\}$ mit $\mathfrak{T}_i$ eine Transversale von b_i bzgl. $H^{(i)}$ bezeichnet. Setze $p := k$.

(1) Für alle $\beta \in b_p^{H^{(p)}}$ und $s \in \tilde{S}^{(p)}$ führe man nun folgende Schritte aus:

(1.1) Berechne $t := T_p(\beta)$ und $v := t \cdot s \cdot T_p(b_p^{t \cdot s})^{-1} = t \cdot s \cdot T_p(\beta^s)^{-1} \in H^{(p)}_{b_p}$.

(1.2) Ist $v \neq 1_G$, so wende Algorithmus (3.2.6) Strip auf v, und $\tilde{S}$ an, um $v \in H^{(p+1)}$ zu überprüfen. Es sei h das Residuum von g vom Typ j bzgl. B und $\tilde{S}$.

(1.3) (a) Gilt $h = 1_G$ und $j = k + 1$, so ist nichts zu tun.

(b) Ist $j \leq k$, so erweitere $\tilde{S}$ um h.

(c) Im Fall $h \neq 1_G$ und $j = k + 1$ erweitere B um ein nicht von h fixiertes Element aus Ω und $\tilde{S}$ um h.

In den Fällen (b) und (c) wird $p := j$ gesetzt und das Verfahren in Schritt (1) neu gestartet.

(2) Ist $p > 1$, so starte das Verfahren in Schritt (1) mit $p - 1$ anstatt p neu.

Das Verfahren terminiert, wenn $p = 1$ ist und keine weiteren Erzeuger hinzugefügt werden müssen.

Satz 3.3.5. *Es gelte Voraussetzung (III), wobei zu Beginn $\tilde{S} = S$ gelte. Weiter seien B und S so, dass es für alle $x \in S$ bereits $B^x \neq B$ gilt. Dann liefert die Durchführung von Verfahren (3.3.4) eine Basis B für G und ein starkes Erzeugendensystem $\tilde{S}$ bzgl. B.*

Beweis. Das p in Verfahren (3.3.4) durchläuft $\{1, \ldots, k\}$ in umgekehrter Reihenfolge und gibt an, welcher Teil der zu B und $\tilde{S}$ gehörigen Stabilisatorkette überprüft wird. Es werden also nacheinander die Schreier-Erzeuger von $H^{(p)}_{b_p}$ in $H^{(p)}$ in Schritt (1) berechnet und im weiteren Verlauf auf Zugehörigkeit zu $H^{(p+1)}$ getestet. Besondere Beachtung soll nun den Residuen h der Schreier-Erzeuger zukommen, welche in Schritt (2) berechnet und anhand derer in Schritt (3) B und $\tilde{S}$ angepasst werden.
Seien also $v \in H^{(p)}_{b_p}$ ein Schreier-Erzeuger von $H^{(p)}_{b_p}$ in $H^{(p)}$ und h das Residuum von v vom Typ j bzgl. B und $\tilde{S}$. Laut Lemma (3.2.7) müssen nun die folgenden Fälle unterschieden werden:

1. Fall: Es seien $h = 1_G$ und $j = k + 1$. Dann liefern die Lemmata (3.2.7) und (3.3.3), dass $v \in H^{(p+1)}$ gilt. Das Verfahren kann also mit der Überprüfung des nächsten Schreier-Erzeugers fortgesetzt werden.

2. Fall: Es sei $j \leq k$. Für das Residuum gilt also $b_j^h \notin b_j^{H^{(j)}}$, es konnte somit keine vollständige Zerlegung von v in Transversalelemente berechnet werden. Insbesondere wurde in diesem Fall $h \notin H^{(j)} = \langle \tilde{S}^{(j)} \rangle$ festgestellt. Da h somit noch nicht in $H^{(j)}$ liegt, muss h als weiterer Erzeuger von $G^{(j)}$ zu $\tilde{S}$ hinzugefügt werden.

3. Fall: Es seien $h \neq 1_G$ und $j = k + 1$. Es wurde also ein nicht-triviales Element aus G gefunden, welches B elementweise fixiert. Sei $a \in \Omega$ so, dass $a^h \neq a$ ist. Dann lassen weder h noch die Elemente aus $\tilde{S}$ die Menge $\{b_1, \ldots, b_k\} \cup \{a\}$ elementweise fest. Es bietet sich also an, mit $(b_1, \ldots, b_k, a)$ als neuen Kandidaten für eine Basis von G fortzufahren und h wie im 2. Fall zu $\tilde{S}$ als einen Erzeuger von $G^{(k+1)}$ hinzuzufügen.

Da im 2. und 3. Fall B oder $\tilde{S}$ erweitert wurden, kann es passieren, dass bisher nicht überprüfte Schreier-Erzeuger von $H^{(q)}_{b_q}$ in $H^{(q)}$ für $q \in \{p, \ldots, j\}$ berechnet werden können. Das Verfahren muss also die Schreier-Erzeuger von $H^{(q)}_{b_q}$ in $H^{(q)}$ erneut überprüfen. Trat ausschließlich Fall 1 bei der Überprüfung ein, so gilt mit Satz (3.3.2) $H^{(p)}_{b_p} = H^{(p+1)}$ und man kann mit der Überprüfung von $H^{(p-1)}_{b_{p-1}} = H^{(p)}$ fortfahren, sofern $p > 1$ ist.

Da sowohl Ω als auch G endlich sind, sind die Größen von B und $\tilde{S}$ ebenfalls endlich und es müssen lediglich endlich viele Schreier-Erzeuger überprüft werden. Das Verfahren endet also nach endlich vielen durchgeführten Schritten, wenn alle Erzeuger erfolgreich überprüft wurden. Gilt letztlich $p = 1$, so ist $H^{(k+1)} = 1_G$, da alle Elemente von S laut Voraussetzung und alle hinzugefügten Elemente per Kontruktion B nicht fixieren. Weiter wurde für jedes $p \in \{1, \ldots, k\}$ die Gleichheit $H^{(p)}_{b_p} = H^{(p+1)}$ verifiziert und mit Lemma (3.3.1) folgt somit, dass B eine Basis für G und $\tilde{S}$ ein starkes Erzeugendensystem bzgl. B ist. □

Bemerkung. Für konkrete Berechnungen ist es sinnvoll, die Erzeugendensysteme der Stabilisatoren sukzessive zu erweitern, anstatt diese immer wieder neu aus $\tilde{S}$ zu berechnen, wenn ein neuer Erzeuger gefunden wurde. Wurde in Verfahren (3.3.4)(3) festgestellt, dass ein Schreier-Erzeuger nicht in $H^{(p+1)}$ liegt, so werden lediglich alle $S^{(q)}$, $q \in \{p+1, \ldots, j\}$, um h erweitert. Die Begründung dafür ist, dass $h \in G^{(j)}$ als Produkt von Elementen aus $S^{(p)}$ berechnet wurde und ein Hinzufügen von h zu den anderen $S^{(l)}$, $l \in \{1, \ldots, p\}$, somit keine neuen Informationen liefern würde.

Die Durchführung von Verfahren (3.3.4) bildet den Hauptbestandteil des Schreier-Sims-Algorithmus' und unter Verwendung der Algorithmen (3.1.5), (3.1.6) sowie (3.2.6) kann nun dessen Implementierung vorgenommen werden. Der zugehörige Pseudocode folgt auf der nächsten Seite.

Algorithmus 3.3.6: SchreierSims[5]– Implementierung in (A.3.4)

Input: k-Tupel $B = (b_1, \ldots, b_k) \in \Omega^k$, Erzeugendensystem S von G, $n := |\Omega|$
Output: Basis B für G und starkes Erzeugendensystem $\tilde{S}$ bzgl. B

```
1  k := |B|;
2  for x in S do
3      if x fixes B then
4          let a ∈ Ω be a point with a^x ≠ a;    Add(B, a);
5  for i in [1,...,k] do
6      S̃^(i) := S ∩ G^(i);    O^(i), v^(i) := Orbit(b_i, S̃^(i), n);
7  p:=k;  𝒮 := [S̃^(1), ..., S̃^(k)];  𝒪 := [O^(1), ..., O^(k)];  𝒱 := [v^(1), ..., v^(k)];
8  while p ≥ 1 do
9      for β in O^(p) do
10         t := Transversal(β, v^(p), S̃^(p));
11         for x in S̃^(p) do
12             t̃ := Transversal(β^x, v^(p), S̃^(p));    v := t · x · t̃^(-1);
13             if v ≠ 1_G then
14                 error := false;    h, j := Strip(v, B, 𝒮, 𝒪, 𝒱);
15                 if j ≤ k then
16                     error := true;
17                 elseif h ≠ 1_G then
18                     error := true
19                     let a ∈ Ω be a point with a^h ≠ a;
20                     Add(B, a);    k := k + 1;    S^(k) := [ ];
21                 if error = true then
22                     for q in [i + 1, ..., j] do
23                         Add(S̃^(q), h);
24                         O^(q), v^(q) := Orbit(b_q, S̃^(q), n);
25                     p := j;    break;
26         if error = true then
27             break;
28     if error = true then
29         continue;
30     p := p − 1;
31 return S̃ := ⋃_{i=1}^{k} S̃^(i);
```

[5]Dieser Pseudocode wurde aus [1], S.91 übernommen.

4. Randomisierung des Schreier-Sims-Algorithmus

4.1. Erzeugung zufälliger Permutationen

Die Verwendung zufällig ausgewählter Permutationen ist der wesentliche Unterschied zwischen der deterministischen Variante des Schreier-Sims-Algorithmus' und dem in Abschnitt (4.2) thematisierten Algorithmus. Ein entsprechender Algorithmus zur Erzeugung solcher Permutationen, welcher u.a. in [1] und [2] nachgeschlagen werden kann, soll hier kurz vorgestellt werden.
Wieder gelte Voraussetzung (III). Zu Beginn wird eine Liste X durch Algorithmus (4.1.1) initialisiert, in welche wiederholt Erzeuger aus S eingetragen werden und welche mindestens 11, im Fall $|S| \geq 11$ maximal $|S|$ Elemente enthält. Laut [1], S. 70 hat sich die Wahl der Mindestanzahl von 11 Elementen auf Grund experimenteller Ergebnisse als sinnvoll erwiesen.

Algorithmus 4.1.1: InitializeGenerator – Implementierung in (A.2.1)

Input: Erzeugendensystem $S = \{x_1, \ldots, x_m\}$ von G
Output: Liste X von Elementen aus S

```
1  X := [x_1, ..., x_m];   p := |S|;   r := max{11, |S|};
2  for i in [p+1, ..., r] do
3      X[i] := X[i-p];
4  return X;
```

Die Idee ist nun, die Elemente in X solange in möglichst zufälliger Art und Weise miteinander zu verknüpfen, bis die als Ergebnis entstehende Permutation in etwa einem gleichverteilt zufällig gewählten Gruppenelement entspricht. Konkret werden zunächst mittels eines Zufallsgenerators Random voneinander verschiedene Indizes $s, t \in \{1, \ldots, |X|\}$ ermittelt. Diese legen fest, dass die beiden Elemente $X[s]$ und $X[t]$ miteinander multipliziert werden sollen. Weiter wird mittels Random ein $e \in \{1, -1\}$ bestimmt, welches festlegt, ob $X[t]$ invertiert werden soll oder nicht. Es wird also $X[s] \cdot X[t]^e$ gerechnet. Zuletzt kann noch die Reihenfolge dieser Multiplikation zufallsbasiert vertauscht werden. Danach erfolgt die Zuweisung $X[s] := X[s] \cdot X[t]^e$, was dazu führen soll, dass die Elemente in X nach hinreichend vielen Iterationen dieses Vorgehens zufällige Elemente aus G sind, mit deren Hilfe wiederum annähernd gleichverteilt zufällig gewählte Permutationen erzeugt werden können. In [1], S.70 wird darauf verwiesen, dass standardmäßig eine heuristische Anzahl von 50 Iterationen verwendet wird.

Algorithmus 4.1.2: RandomPerm – Implementierung in (A.2.2)

Input: Erzeugendensystem $S = \{x_1, \ldots, x_m\}$ von G und durch Algorithmus (4.1.1) erzeugte Liste X von Elementen aus S

Output: möglichst zufällig gewählte Permutation $a \in G$

```
1  a := 1_G ;   r := max{11, |S|} ;
2  for i in [1, ..., 50] do
3        s := Random([1, ..., r]) ;   t := Random([1, ..., r] \ [s]) ;
4        e := Random([1, −1]) ;
5        if Random([0, 1]) = 1 then
6              X[s] := X[s] · X[t]^e ;   a := a · X[s] ;
7        else
8              X[s] := X[t]^e · X[s] ;   a := X[s] · a ;
9  return a ;
```

4.2. Randomisierter Schreier-Sims-Algorithmus

Jeffrey S. Leon stellte 1980 in [3] eine Version des Schreier-Sims-Algorithmus' vor, welche anstelle von Schreier-Erzeugern zufällig gewählte Elemente aus G in Transversalelemente zerlegt. Entstanden ist ein randomisierter Algorithmus, welcher bis zum Eintreten einer bestimmten Stoppbedingung B und $\tilde{S}$ anpasst. Wesentlich ist letztlich die Frage, ob das nach Eintreten dieser Stoppbedingung vom Algorithmus zurückgegebene Ergebnis auch wirklich richtig ist.
Doch bevor man diese Frage stellt, sollte man sich mit möglichen Stoppbedingungen beschäftigen. Es gelte also wieder Voraussetzung (III) des vorherigen Kapitels und für alle $i \in \{1, \ldots, k\}$ sei $\mathfrak{T}_i$ eine Transversale von b_i bzgl. $H^{(i)}$. Dann erhält man z. B. eine sinnvolle Stoppbedingung, wenn die Größe der Gruppe G bereits vorab bekannt ist. Wird bei der Durchführung des randomisierten Algorithmus' nämlich festgestellt, dass

$$|G| = |\mathfrak{T}_1| \cdot |\mathfrak{T}_k| \cdots |\mathfrak{T}_k|$$

gilt, so kann man wegen Lemma (3.2.4)(a) darauf schließen, dass B eine Basis und $\tilde{S}$ ein starkes Erzeugendensystem ist und den Algorithmus beenden.
Da $|G|$ in den meisten Fällen jedoch unbekannt ist und vom Schreier-Sims-Algorithmus bestimmt werden soll, benötigt man eine alternative Stoppbedingung. Eine solche liefert das folgende Resultat, welches mit Beweis in [4] zu finden ist.

Satz 4.2.1 (aus [4], S. 63, Lemma 4.3.1). *Es gelte Voraussetzung (II). Ist B keine Basis für G oder S kein starkes Erzeugendensystem, so beträgt die Wahrscheinlichkeit, dass ein gleichverteilt zufällig gewähltes Element $g \in G$ nicht erfolgreich[6] in Transversalelemente zerlegt werden kann, mindestens $\frac{1}{2}$.*

Korollar 4.2.2. *Es gelte Voraussetzung (II). Wurden nacheinander $l \in \mathbb{N}$ gleichverteilt zufällig gewählte Elemente aus G erfolgreich in Transversalelemente zerlegt, so ist die Wahrscheinlichkeit, dass B keine Basis oder S kein starkes Erzeugendensystem ist, höchstens 2^{-l}.*

Unter Angabe einer Toleranzgrenze für die Fehlerwahrscheinlichkeit bei einem Abbruch des Algorithmus liefert Korollar (4.2.2) eine akzeptable Stoppbedinung. Mit Hilfe dieser und der Betrachtungen aus Abschnitt (4.1) kann nun die randomisierte Variante des Schreier-Sims-Algorithmus' nach Leon implementiert werden. Der zugehörige Pseudocode ist auf der nächsten Seite zu finden.

[6] 'Erfolgreich' bedeutet hier, dass das Residuum von g bzgl. B und S trivial und vom Typ $k+1$ ist.

Algorithmus 4.2.3: RandomSchreierSims[7]– Implementierung in (A.3.5)

Input: k-Tupel $B = (b_1, \ldots, b_k) \in \Omega^k$, Erzeugendensystem S von G, $n := |\Omega|$ und $\varepsilon \in \mathbb{N}$.

Output: Basis B für G und starkes Erzeugendensystem $\tilde{S}$ bzgl. B

```
1  for x in S do
2      if x fixes B then
3          let a ∈ Ω be a point with a^x ≠ a;
4          Add(B, a);
5  for i in [1,...,k] do
6      S̃^(i) := S ∩ G^(i);    O^(i), v^(i) := Orbit(b_i, S̃^(i), n);
7  𝒮 := [S̃^(1), ..., S̃^(k)];  𝒪 := [O^(1), ..., O^(k)];  𝒱 := [v^(1), ..., v^(k)];
8  c := 0;
9  while c < ε do
10     let g be a randomly chosen element of G;
11     h, j := Strip(v, B, 𝒮, 𝒪, 𝒱);    error := false;
12     if j ≤ k then
13         error := true;
14     elseif h ≠ 1_G then
15         error := true
16         let a ∈ Ω be a point with a^h ≠ a;
17         Add(B, a);    k := k + 1;    S̃^(k) := [ ];
18     if error = true then
19         for q in [2, ..., j] do
20             Add(S̃^(q), h);
21             O^(q), v^(q) := Orbit(b_q, S̃^(q), n);
22         c := 0;
23     else
24         c := c + 1;
25 return S̃ := ⋃_{i=1}^{k} S̃^(i);
```

[7] Dieser Pseudocode wurde aus [1], S.98 übernommen.

Bemerkungen.

(1) Anders als beim deterministischen Algorithmus (3.3.6) muss das Residuum der zufällig gewählten Permutation in Zeile 19 zu allen Teilerzeugendensystemen $S^{(q)}$ für $q \in \{2, \ldots, j\}$ hinzugefügt werden, da das Residuum nicht zwangsläufig als Produkt von Elementen aus $S^{(p)}$ geschrieben werden kann. Grund dafür ist die zufällige Auswahl der Permutation, deren Residuum betrachtet wird.

(2) Durch das Erweitern aller Teilerzeugendensysteme um entsprechende Residuen kann es passieren, dass auch nicht benötigte Erzeuger hinzugefügt werden und einige dieser Teilerzeugendensysteme somit unnötig groß werden.

(3) Sowohl in [1] als auch in [3] wird darauf hingewiesen, dass die randomisierte Variante in den meisten Fällen wesentlich schneller ein Ergebnis liefert als die im vorigen Kapitel vorgestellte deterministische.

(4) Die Wahrscheinlichkeit, dass Algorithmus (4.2.3) keine Basis B und kein starkes Erzeugendensystem $\tilde{S}$ liefert, ist zwar gering, aber größer Null. Die Überprüfung der Korrektheit der Ergebnisse ist Gegenstand des nächsten Abschnitts (4.3) und des 5. Kapitels.

Bevor die Verifikation der Ergebnisse von (4.2.3) thematisiert wird, soll zunächst noch einmal auf Bemerkung (2) eingegangen werden, da die Größe des Erzeugendensystems u.a. maßgeblich für die Geschwindigkeit einer Verifikation ist. Leon empfiehlt in [3], überflüssige Erzeuger in $\tilde{S}$ nach der Durchführung von (4.2.3) und vor einer Verifikation der Ergebnisse zu entfernen. Dabei bedeutet 'überflüssig', dass für ein $i \in \{1, \ldots, k\}$ und ein $g \in \tilde{S}^{(i)} \backslash \tilde{S}^{(i+1)}$ schon $b_i^{\langle \tilde{S}^{(i)} \backslash \{g\} \rangle} = b_i^{\langle \tilde{S}^{(i)} \rangle}$ gilt. In diesem Fall kann das Element g aus $\tilde{S}$ entfernt werden, anderenfalls wird es benötigt und muss beibehalten werden. Dies wird durch folgenden Algorithmus realisiert.

Algorithmus 4.2.4: RemoveRedundantGens[8]– Implementierung in (A.3.6)

Input: Von Algorithmus (4.2.3) berechnetes $B = (b_1, \ldots, b_k)$ und $\tilde{S}$, für jedes $i \in \{1, \ldots, k\}$ Bahnen $O^{(i)} := b_i^{\langle \tilde{S}^{(i)} \rangle}$

```
1  for i in [k, k − 1, ..., 1] do
2      for g ∈ S̃^(i) \ S̃^(i+1) do
3          if b_i^⟨S̃^(i) \ {g}⟩ = O^(i) then
4              remove g from S̃;
```

[8] Dieser Pseudocode wurde aus [1], S.95 übernommen.

4.3. Deterministische Verifikation

Nach der Durchführung von Algorithmus (4.2.3) ist zunächst noch ungewiss, ob die gelieferten Ergebnisse wirklich eine Basis und ein starkes Erzeugendensystem sind. In [1], S. 98 wird unter anderem vorgeschlagen, den deterministischen Algorithmus (3.3.6) auf die Ergebnisse von (4.2.3) anzuwenden. Diesem Hinweis folgend kann nun der Hauptbestandteil (Zeilen 9 bis 31) von (3.3.6) zur Durchführung von Verfahren (3.3.4) verwendet werden, um B und $\tilde{S}$ auf Korrektheit zu überprüfen. Das Resultat ist der nun folgende Pseudocode zur Verifikation von $H_{b_p}^{(p)} = H^{(p+1)}$ für jedes $p \in \{1, \ldots, k\}$.

Algorithmus 4.3.1: DeterministicVerification – Implementierung in (A.3.7)

Input: Von (4.2.3) berechnete mögliche Basis $B = (b_1, \ldots, b_k) \in \Omega^k$, für jedes $i \in \{1, \ldots, k\}$ Erzeugendensysteme $\tilde{S}^{(i)}$, zugehörige Bahnen $O^{(i)} := b_i^{\langle S^{(i)} \rangle}$, Schreier-Vektoren $v^{(i)}$, $n := |\Omega|$ und $p \in \{1, \ldots, k\}$.

Output: `true`, falls alle Schreier-Erzeuger von $H_{b_p}^{(p)}$ in $H^{(p+1)}$ liegen. Sonst `false`.

```
1  S := [S̃^(1), ..., S̃^(k)];  O := [O^(1), ..., O^(k)];  V := [v^(1), ..., v^(k)];
2  for β in O^(p) do
3      t := Transversal(β, v^(p), S^(p));
4      for x in S̃^(p) do
5          if x ∉ S^(p+1) or β ∉ O^(p+1) then
6              t̃ := Transversal(β^x, v^(p), S̃^(p));
7              v := t · x · t̃^-1;
8              if v ≠ 1_G then
9                  h, j := Strip(v, B, S, O, V);
10                 if j ≤ k then
11                     return false;
12                 elseif h ≠ 1_G then
13                     return false;
14 return true;
```

Lemma 4.3.2. *Es gelte Voraussetzung (III). Sei T die Laufzeit von Algorithmus (4.3.1). Dann gilt*

$$T \in \mathcal{O}(n^4 \cdot |\tilde{S}| \cdot (|B| + 1)).$$

BEWEIS. Es muss nun betrachtet werden, über welche Mengen in Algorithmus (4.3.1) iteriert wird und welche Operationen dabei ausgeführt werden. Weiter wird wieder angenommen, dass die Multiplikation zweier Zyklen in quadratischer Laufzeit $\mathcal{O}(n^2)$ erfolgen kann. In Zeile 2 wird über alle jeweiligen Bahnenelemente iteriert, was im schlimmsten Fall n viele sind. Für jedes Bahnelement wird ein Transversalelement t mit einem Aufwand von $\mathcal{O}(n^3)$ berechnet und anschließend über alle Elemente eines Teilerzeugendensystems in Zeile 4 iteriert, wobei die Anzahl der Erzeuger durch $|\tilde{S}|$ beschränkt wird. Die Operationen von Zeile 6 bis 13 bestitzen insgesamt eine Laufzeitkomplexität von $\mathcal{O}(n^3) + \mathcal{O}(n^2) + \mathcal{O}(n^3 \cdot |B|)$, wobei die letzten beiden Terme die Laufzeiten der Multiplikation in Zeile 7 und der Methode STRIP in Zeile 9 beschreiben. Insgesamt ergibt sich

$$\begin{aligned} T &\in \mathcal{O}(n) \cdot \Big[\mathcal{O}(n^3) + \mathcal{O}(|\tilde{S}|) \cdot \big(\mathcal{O}(n^3) + \mathcal{O}(n^2) + \mathcal{O}(n^3 \cdot |B|)\big)\Big] \\ &= \mathcal{O}(n^4) + \mathcal{O}(n^4 \cdot |\tilde{S}|) + \mathcal{O}(n^3 \cdot |\tilde{S}|) + \mathcal{O}(n^4 \cdot |\tilde{S}| \cdot |B|) \\ &= \mathcal{O}(n^4 \cdot |S| \cdot (|B| + 1)). \end{aligned}$$

□

Bemerkungen.

(1) Die im Vergleich zu (3.3.6) zusätzliche Bedingung in Zeile 5 von (4.3.1) sichert, dass bereits überprüfte Schreier-Erzeuger keinem erneuten Test unterzogen werden.

(2) Weitere Verifikationsmethoden sind z. B.

- die Verifikation mittels des Todd-Coxeter-Algorithmus und
- die Verifikationsmethode VERIFY nach Sims.

Eine ausführliche Behandlung dieser beiden Methoden findet man jeweils in [1] und in [4].

Die Berechnung von Basis und starkem Erzeugendensystem mit den in diesem Kapitel vorgestellten Algorithmen erfolgt gemäß dem Vorschlag von Leon in [3] in folgender Reihenfolge:

(a) Berechnung einer vermeintlichen Basis und eines vermeintlichen starken Erzeugendensystems mit Algorithmus (4.2.3).

(b) Entfernen überflüssiger Erzeuger mittels Algorithmus (4.2.4) zur Minimierung der Rechenzeit der Verifikation in (c).

(c) Anwendung einer Verifikationsmethode wie z. B. Algorithmus (4.3.1) auf die Ergebnisse aus (a).

5. Parallelisierung einer deterministischen Verifikation

Der zeitaufwändigste Teil bei der Konstruktion einer Basis und eines starken Erzeugendensystems mittels des in Kapitel zwei vorgestellten randomisierten Schreier-Sims-Algorithmus (4.2.3) ist die darauf folgende deterministische Verifikation der Ergebnisse, z. B. mit Algorithmus (4.3.1). Das Ziel ist nun, die Laufzeit der Verifikation durch eine effizientere Nutzung der zur Verfügung stehenden Prozessorleistung zu verringern. Zu diesem Zweck werden in diesem Kapitel zwei Möglichkeiten der Parallelisierung der Verifikationsmethode aufgezeigt. Dabei häufig verwendete Begriffe werden zunächst definiert.

Definition. *Als **Programm** wird eine Folge von Anweisungen bezeichnet, welche den Regeln einer bestimmten Programmiersprache genügen. Ein **Prozess** ist die Ausführung eines Programms auf einem Prozessor. Verwendet ein Algorithmus mehrere Prozesse gleichzeitig, so heißt dieser **parallel**. Wird hingegen nur genau ein Prozess gestartet, heißt er **sequentiell**.*

Bemerkungen.

(1) Das Ziel ist also nun, die Verifikation der Ergebnisse von (4.2.3) in Teilschritte zu unterteilen, möglichst viele von diesen gleichzeitig in jeweils einem Prozess auszuführen und letztlich die einzelnen Zwischenergebnisse auszuwerten.

(2) Beim Anlegen eines Prozesses müssen die entsprechenden Daten in der Regel in den ihm zugewiesenen Speicher kopiert werden, bevor Berechnungen mit diesen Daten durchgeführt werden können. Weiterhin müssen die Ergebnisse der einzelnen Prozesse an einen Hauptprozess übermittelt werden, in welchem diese zum Zwecke der Verifikation ausgewertet werden. Diese „Parallelisierungskosten“ wirken sich zusätzlich zu den eigentlichen Berechnungen auf die Laufzeit der Verifikation aus. Der Laufzeitvorteil einer Parallelisierung muss also größer als der Laufzeitnachteil durch die Parallelisierungskosten sein.

(3) Die maximale Anzahl gleichzeitig ausführbarer Prozesse hängt vom jeweils zur Verfügung stehenden Computer ab, auf welchem die Berechnungen erfolgen.

5.1. Zwei Ansätze zur Parallelisierung

Es gelte wieder Voraussetzung (III). Bevor auf besagte Parallelisierungsansätze eingegangen wird, soll zunächst erwähnt werden, dass die Überprüfungen der Gleichheiten $H_{b_q}^{(q)} = H^{(q+1)}$ mittels Algorithmus (4.3.1) für alle $q \in \{1, \ldots, k\}$ unabhängig voneinander und beginned mit $q = k$ der Reihe nach bis $q = 1$ erfolgen.

Methode (I): Parallelisierung nach Basisgröße

Bei dieser Methode wird jede Relation $H_{b_q}^{(q)} = H^{(q+1)},\ q \in \{1, \ldots, k\}$, in einem separaten Prozess mittels Algorithmus (4.3.1) überprüft. Sie wurde mit „Parallelisierung nach Basisgröße“ benannt, da insgesamt $|B| = k$ Prozesse gestartet werden. Liefert jeder dieser k Prozesse am Ende ihrer Durchführung `true`, so wurden die Ergebnisse von (4.2.3) als korrekt verifiziert. Anderenfalls wurde festgestellt, dass B keine Basis oder $\tilde{S}$ kein starkes Erzeugendensystem ist.
Die Vorgehensweise bei dieser Methode wird in der folgenden Abbildung (2) schematisiert dargestellt und eine Implementierung dieser findet sich in (A.4.1).

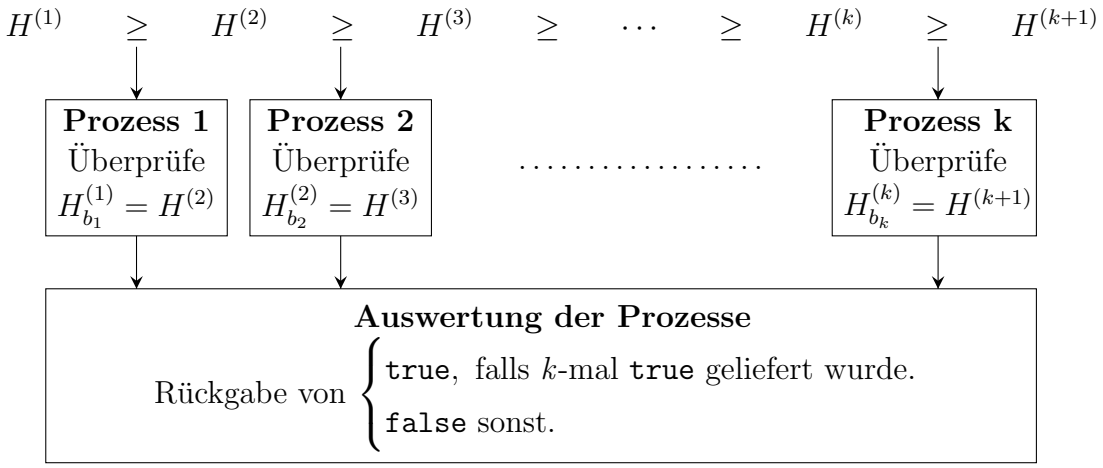

Abbildung 2: Parallelisierung nach Basisgröße

In welchen Fällen ergibt sich nun durch diesen Parallelisierungsansatz ein Laufzeitvorteil gegenüber der sequentiellen Verifikation? Dazu soll die nun folgende Prognose abgegeben werden.

Prognose. Da die Anzahl der Prozesse von der Größe der betrachteten Basis abhängig ist, kann bei Gruppen mit hinreichend großer Basis mit einem Laufzeitvorteil gerechnet werden.

Methode (II): Parallelisierung nach Bahnenlänge

Bei der nun vorzustellenden zweiten Methode wird ausgenutzt, dass in Algorithmus (4.3.1) in Zeile 2 bei der Berechnung entsprechender Schreier-Erzeuger zur Überprüfung von $H_{b_q}^{(q)} = H^{(q+1)}$ für fest gewähltes $q \in \{1, \ldots, k\}$ über alle Elemente der Bahn $b_q^{H^{(q)}}$ iteriert wird. Es wird $b_q^{H^{(q)}}$ in eine vorgegebene Anzahl $y \in \mathbb{N}$ Teilmengen $P_1, \ldots, P_y$

partitioniert, mittels derer in mehreren Prozessen Schreier-Erzeuger von $H_{b_q}^{(q)}$ berechnet und auf Zugehörigkeit zu $H^{(q+1)}$ getestet werden können. Damit die Anzahl der durchzuführenden Berechnungen von Prozess zu Prozess in etwa gleich ist, sollten die Teilmengen $P_1, \ldots, P_y$ möglichst gleichmächtig sein. Da aber y nicht notwendigerweise ein Teiler von $|b_q^{H^{(q)}}|$ ist, wird dies im Allgemeinen nicht möglich sein. Es genügt jedoch auch schon, dass $|P_1| = \cdots = |P_{y-1}|$ und $b_q^{H^{(q)}} = |P_1| \cdot y + |P_y|$ gelten, um die Arbeit möglichst gleichmäßig auf die Prozesse zu verteilen.
Nach der Durchführung der insgesamt y Prozesse werden deren Ergebnisse wieder ausgewertet. Wurde von jedem Prozess `true` geliefert, so gilt $H_{b_q}^{(q)} = H^{(q+1)}$ und es kann die Überprüfung von $H_{b_{q-1}}^{(q-1)} = H^{(q)}$ in gleicher Weise begonnen werden, sofern $q > 1$ ist. Im Fall, dass mindestens einmal `false` geliefert wurde, hat man die Unvollständigkeit von B oder $\tilde{S}$ verifiziert und der Verifikationsalgorithmus kann beendet werden. Wenn letzlich für jedes $q \in \{1, \ldots, k\}$ erfolgreich $H_{b_q}^{(q)} = H^{(q+1)}$ überprüft wurde, so sind B eine Basis und $\tilde{S}$ ein starkes Erzeugendensystem gemäß Lemma (3.3.1).
Eine Implementierung dieser Methode findet man in (A.4.2) und in der folgenden Abbildung (3) wird diese schematisch dargestellt.

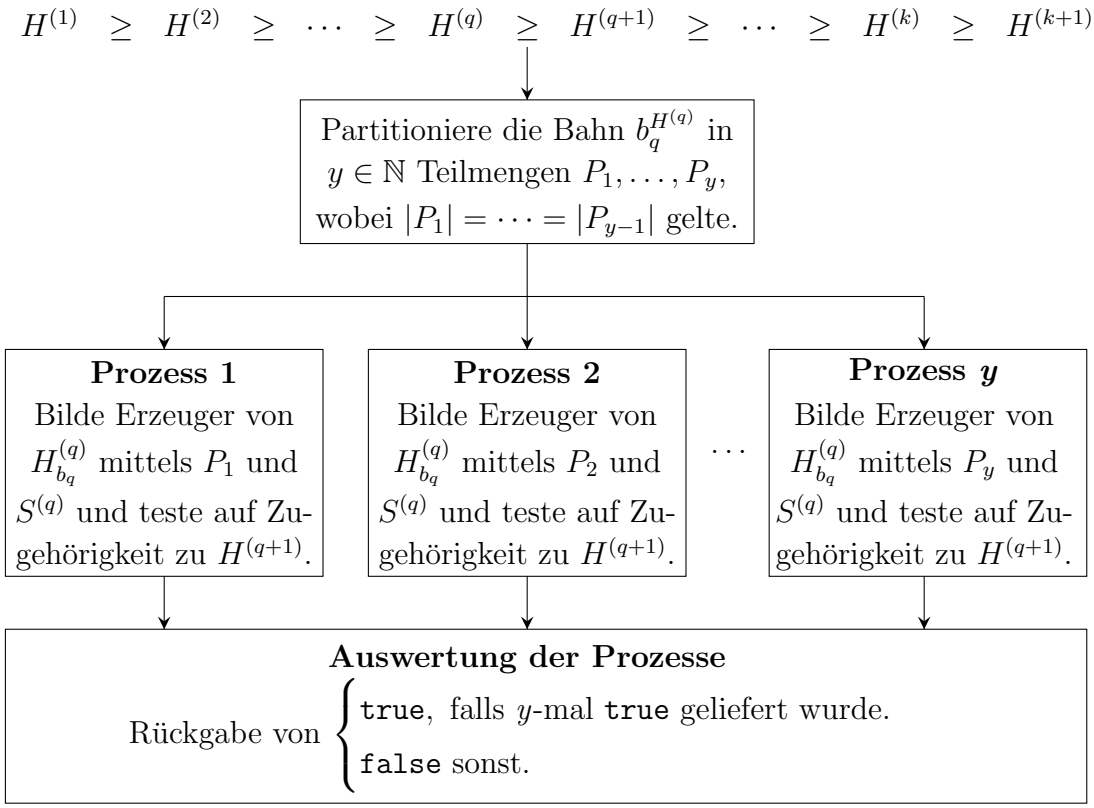

Abbildung 3: Parallelisierung nach Bahnenlänge - Verifikation von $H_{b_q}^{(q)} = H^{(q+1)}$

Wie schon zuvor bei Methode (I) soll nun eine Prognose abgegeben werden, in welchem Fall man bei der Verwendung dieser Methode einen Laufzeitvorteil gegenüber der sequentiellen Verifikation erwarten kann.

Prognose. Die Verwendung des zweiten Parallelisierungsansatzes wird bei Gruppen zu einem Laufzeitvorteil führen, bei denen mindestens eine der Bahnen $b_q^{H^{(q)}}$, $q \in \{1, \ldots, k\}$, hinreichend groß ist.

Beispiele für solche Gruppen wären u.a. transitiv operierende Gruppen bei genügend großem $n = |\Omega| \in \mathbb{N}$.

5.2. Laufzeitmessungen

Zum Vergleich der Laufzeiten der beiden parallelisierten Verifikationen aus Abschnitt (5.1) und der zugrundeliegenden sequentiellen Verifikation wurden Laufzeitmessungen für die Gruppen S_{80}, S_{120}, S_{160}, S_{200}, M_{11}, M_{24}, J_1, J_2, J_3, Suz, Co_2, Fi_{22}, Fi_{23} und die Gruppe $\mathcal{R}$ aus Beispiel (3.2.1) vorgenommen. Diese sind bis auf $\mathcal{R}$ allesamt transitiv und der Großteil von ihnen gehört zu den sporadischen Gruppen. Ihre Auswahl erfolgte unter dem Gesichtspunkt, dass jene sporadische Gruppen besonders kleine Basen besitzen im Vergleich zur Mächtigkeit der Mengen, auf denen sie jeweils operieren. Die Berechnung von kleinen Basen für Gruppen ist besonders ergiebig, da deren Elemente laut Lemma (3.2.4)(b) eindeutig durch die zugehörigen Bilder einer Basis bestimmt werden können. Permutationen können somit anhand ihrer Wirkung auf $|B|$ Elementen ansatt auf $|\Omega|$ vielen charakterisiert werden, was sich als besonders effizient beim Speichern von Permuationen bei Berechnungen mittels Computern erweist.
Die Daten zu den Ergebnissen von Algorithmus (4.2.3), welche wesentlich für die Laufzeit der im Anschluss durchzuführenden Verifikation sind, werden auf der nächsten Seite in Tabelle (1) gelistet. Ebenfalls sind dort die jeweiligen Laufzeiten von (4.2.3) zu finden. Das n bei den sporadischen Gruppen gibt jeweils an, dass diese Gruppen als Untergruppen der jeweiligen S_n aufgefasst werden. In Tabelle (2) werden schließlich die Laufzeiten der Verifikationen aufgelistet. Sämtliche Laufzeiten wurden als Mittelwert aus jeweils 5 Laufzeitmessungen errechnet.
Alle Berechnungen wurden auf einem Intel Core i5-4570 mit 4 Prozessorkernen und einer Taktrate von 3.2 GHz mittels HPC-GAP unter Ubuntu 14.04 ausgeführt. Dabei wurden ausschließlich die in dieser Arbeit vorgestellten Algorithmen verwendet. Methode (II) wurde mit dem Parameter $y = 20$ durchgeführt.

Tabelle 1: Daten zu den von (4.2.3) berechneten Ergebnissen für ausgewählte Gruppen

Gruppe	$n = \|\Omega\|$	$\|B\|$	$\|\tilde{S}\|$	Rechenzeit von (4.2.3) in s
M_{11}	11	4	6	0.0019
M_{24}	24	7	9	0.0026
J_1	266	3	5	0.0025
J_2	100	4	5	0.0016
J_3	6 156	3	6	0.0860
Suz	1 782	6	7	0.0244
Co_2	2 300	6	9	0.0874
Fi_{22}	3 510	5	9	0.3824
Fi_{23}	31 671	10	11	70.542
$\mathcal{R}$	48	18	22	0.0141
S_{80}	80	79	80	1.0107
S_{120}	120	119	120	6.9393
S_{160}	160	159	160	27.808
S_{200}	200	199	200	77.641

Tabelle 2: Laufzeiten der Verifikationen, jeweils in Sekunden

Gruppe	Methode (I)	Methode (II)	sequentielle Verifikation
M_{11}	0.0003	0.0005	0.0002
M_{24}	0.0004	0.0012	0.0008
J_1	0.0117	0.0059	0.0131
J_2	0.0020	0.0026	0.0026
J_3	7.4268	5.3407	7.6078
Suz	1.1767	0.9387	1.2557
Co_2	2.7288	1.9744	3.3079
Fi_{22}	3.5566	3.0761	5.5097
Fi_{23}	470.48	334.18	646.98
$\mathcal{R}$	0.0074	0.0175	0.0164
S_{80}	0.1253	0.1304	0.3409
S_{120}	0.5356	0.5146	1.4332
S_{160}	1.5529	1.4475	4.1966
S_{200}	3.5365	3.1563	9.1560

5.3. Auswertung der Laufzeitmessungen

Es soll nun untersucht werden, ob die Prognosen aus Abschnitt (5.1) hinsichtlich der Laufzeitvorteile der Parallelisierungsansätze durch die Ergebnisse der Laufzeitmessungen im vorherigen Abschnitt bestätigt werden.

Auswertung der Laufzeiten der Verifikation mit Methode eins

Betrachtet man die Laufzeiten der Verifikation unter Verwendung des ersten Parallelisierungsansatzes in Tabelle (2), so stellt man fest, dass die relativen Laufzeitvorteile – sofern vorhanden – gegenüber der sequentiellen Verifikation bei den getesteten sporadischen Gruppen deutlich geringer ausfallen als bei der Gruppe $\mathcal{R}$ oder der Symmetrischen Gruppe für verschiedene n. Besonders auffällig sind hier die Laufzeiten bei den Gruppen J_1 und J_3, welche nicht wesentlich kleiner als die der sequentiellen Verifikation sind. Beide Gruppen besitzen jeweils eine Basis der Größe drei, die Anzahl der bei der Laufzeitmessung zur Verfügung stehenden Prozessorkerne betrug aber vier. Somit wurde in diesen Fällen die verfügbare Prozessorleistung nicht effizient ausgenutzt.
Es soll nun die J_3 genauer betrachtet werden. Algorithmus (4.2.3) berechnet für diese Gruppe eine Basis $B = (1, 2, 3)$, ein zugehöriges starkes Erzeugendensystem $\tilde{S}$ und die Bahnenlängen

$$|1^{H^{(1)}}| = 6\,156, \quad |2^{H^{(2)}}| = 510 \quad \text{und } |3^{H^{(3)}}| = 8,$$

wobei wieder $H^{(i)} = \langle \tilde{S}^{(i)} \rangle$, $i \in \{1, 2, 3\}$, gelte. Die erste Bahn besitzt auf Grund der Transitivität der J_3 die Mächtigkeit $|\Omega|$ und ist damit in Relation zu den anderen Bahnen sehr groß. Daraus ergibt sich das Problem, dass bei der Überprüfung von $H_1^{(1)} = H^{(2)}$ wesentlich mehr Schreier-Erzeuger berechnet werden müssen als bei $H_2^{(2)}$ und $H_3^{(3)}$. Dies führt dazu, dass die letzten beiden Überprüfungen weitaus eher fertig sind als die erste, und insgesamt drei Prozessorkerne ungenutzt bleiben, bis $H_1^{(1)} = H^{(2)}$ überprüft wurde. Insgesamt ergibt sich aus genannten Gründen im Falle der J_3 lediglich eine Laufzeit, welche gleich der des sequentiellen Algorithmus' ist.

Anders verhält es sich bei der Gruppe $\mathcal{R}$ aus Beispiel (3.2.1)(3) und der Symmetrischen Gruppe S_n auf $n \in \{80, 120, 160, 200\}$ Elementen. Bei beiden sind die Basen relativ zum jeweiligen n größer als bei den sporadischen Gruppen, bei der S_n gilt sogar stets $|B| = n - 1$, wie man Tabelle (2) oder Beispiel (3.2.1)(1) entnehmen kann. Im Fall von $\mathcal{R}$ benötigt Methode eins lediglich halb so viel Zeit wie die sequentielle Verifikation, bei der S_n sogar nur etwas mehr als ein Drittel.

Aus den Ergebnissen der Laufzeitmessungen in den Tabellen (1) und (2) ist somit ein Zusammenhang zwischen der Laufzeit der Verifikation mittels Methode eins und der Größe der jeweiligen Basis erkennbar, welcher der ersten Prognose aus Abschnitt (5.1) entspricht. Schlechtere Ergebnisse ergeben sich bei Gruppen mit kleinen Basen relativ zu n und bei transitiv operierenden Gruppen bei gleichzeitig hinreichend großem n.

Auswertung der Laufzeiten der Verifikation mit Methode (II)

Es werden zunächst wieder die Laufzeiten der Verifikation mittels Methode (II) bei den sporadischen Gruppen betrachtet. Während diese bei $M_{11},\ M_{24}$ und J_2 nicht wesentlich von den Laufzeiten des sequentiellen Alogrithmus' abweichen, zeigen sich mit wachsendem n immer größer werdende Laufzeitvorteile. Diese betragen etwa 2.3 Sekunden bei J_3 und $n = 6\,156$ bis hin zu 312 Sekunden bei Fi_{23} und $n = 31\,671$. Dabei unterscheiden sich die Größen der jeweils zu diesen Gruppen berechneten Basen und starken Erzeugendensysteme kaum voneinander.
Der Grund für die mit wachsendem n immer geringer werdenden Laufzeiten liegt in der Transitivität der ausgewählten sporadischen Gruppen. Um dies zu verdeutlichen, soll nun Fi_{23} genauer betrachtet werden. Von Algorithmus (4.2.3) wurden eine Basis

$$B = (1, 2, 4, 7, 11, 37, 16, 29, 22, 46)$$

und ein zugehöriges starkes Erzeugendensystem $\tilde{S}$ berechnet. Interessant sind nun die Bahnenlängen

$$|1^{H^{(1)}}| = 31\,671, \quad |2^{H^{(2)}}| = 28\,160, \quad |4^{H^{(3)}}| = 3\,159, \quad |7^{H^{(4)}}| = 288. \quad |11^{H^{(5)}}| = 7,$$
$$|37^{H^{(6)}}| = 6, \quad |16^{H^{(7)}}| = 5, \quad |29^{H^{(8)}}| = 4, \quad |22^{H^{(9)}}| = 3 \quad \text{und} \quad |46^{H^{(10)}}| = 2.$$

Besonders auffällig sind die ersten drei, welche sehr groß im Vergleich zu den Restlichen sind. Es müssen somit deutlich mehr Schreier-Erzeuger bei den Überprüfungen von $H_1^{(1)} = H^{(2)},\ H_2^{(2)} = H^{(3)}$ und $H_4^{(3)} = H^{(4)}$ berechnet werden als bei den anderen sechs druchzuführenden Überprüfungen. Daher ist es zweckmäßig, diese Bahnen gemäß Methode (II) zu partitionieren und den Rechenaufwand auf mehrere Prozesse aufzuteilen. Die Anwendung von Methode (II) auf die restlichen Bahnen lohnt sich jedoch kaum, da die Bahnenlängen in diesem Fall kleiner sind als $y = 20$, welches angibt, in wie viele Teilmengen die jeweilige Bahn partitioniert werden soll. In diesen Fällen werden die Rechnungen jeweils nur in einem Prozess durchgeführt, um Parallelisierungskosten zu vermeiden. Letztlich ergeben sich Laufzeitvorteile von 312 Sekunden gegenüber dem sequentiellen Algorithmus und sogar 136 Sekunden gegenüber Methode (I). Diese Betrachtungen zu Fi_{23} können so auch für die anderen getesteten sporadischen Gruppen

mit jeweils ähnlichem Ergebnis durchgeführt werden.

Die Laufzeit von Methode (II) im Fall der Gruppe $\mathcal{R}$ ist hingegen sogar schlechter als die der sequentiellen Verifikation und damit insbesondere weitaus schlechter als die Laufzeit von Methode (I). Dies kann damit begründet werden, dass die Basis von $\mathcal{R}$ im Vergleich zu n größer ist als bei den sporadischen Gruppen, während die zur Basis und zum starken Erzeugendensystem gehörigen Bahnen wegen der Nicht-Transitivität der Gruppe relativ klein sind. Berechnungen mittels Algorithmus (4.2.3) ergeben, dass die Bahnenlängen allesamt kleiner als 24 und die meisten sogar kleiner als $y = 20$ sind. Es entsteht somit kein Vorteil durch die Parallelisierung gemäß Methode (II), im Gegenteil ergibt sich sogar eine höhere Laufzeit gegenüber der sequentiellen Verifikation.

Besonders gute Laufzeiten werden bei der Symmetrischen Gruppe erzielt, obwohl die berechneten Basen stets $n-1$ Einträge besitzen. Zur Begründung dessen seien nun mit $B = (b_1, \ldots, b_{n-1})$ eine Basis für die S_n, mit $\tilde{S}$ ein zugehöriges starkes Erzeugendensystem und für jedes $i \in \{1, \ldots, n-1\}$ mit $H^{(i)} = S^{(i)}$ die jeweiligen Stabilisatoren bezeichnet. Es sollen nun wieder die Bahnen $b_1^{H^{(1)}}, \ldots, b_k^{H^{(n-1)}}$ betrachtet werden. Wie bereits in Kapitel 2 erwähnt, ist die S_n für jedes $n \in \mathbb{N}$ n-fach transitiv. Mit Lemma (2.1) folgt nun, dass für jedes $i \in \{1, \ldots, n-1\}$ die Stabilisatoren $H^{(i)}$ jeweils transitiv auf $\{1, \ldots, n\} \backslash \{b_1, \ldots, b_{i-1}\}$ operieren. Für die Bahnenlängen gilt somit

$$|b_i^{H^{(i)}}| = n - (i-1),$$

was bei hinreichend großem n besonders vorteilhaft für die Verifikation mittels Methode (II) ist. So beträgt die Laufzeit von Methode (II) bei allen getesteten Symmetrischen Gruppen lediglich ein Drittel der Laufzeit der sequentiellen Verifikation.

Insgesamt hat sich Methode (II) bei Transitivität der betrachteten Gruppe und hinreichend großem n als vorteilhaft erwiesen, was sich in besonders deutlichen Laufzeitvorteilen bei Fi_{23} und der Symmetrischen Gruppe widerspiegelt. Dieses Ergebnis entspricht der zweiten Prognose aus Abschnitt (5.1).

Auswirkung von Parallelisierungskosten auf die Laufzeiten

Wie schon zu Beginn des Kapitels erwähnt, entstehen bei der Verwendung der beiden Parallelisierungsansätze gewisse Parallelisierungskosten, welche sich zusätzlich zur eigentlichen Rechenzeit negativ auf die Laufzeit der Verifikation auswirken. Hauptgrund für diese Kosten ist das Kopieren der zu bearbeitenden Daten in den dem jeweiligen Prozess zugewiesenen Speicher und je größer die Datenmengen sowie die Zahl der Prozesse

sind, desto höher sind die Parallelisierungskosten. Diese können jedoch durch einmaliges Kopieren der Daten in einen global zugänglichen Speicherbereich verringert werden, was auch bereits bei der Durchführung der Laufzeitmessungen in Abschnitt (5.2) getan wurde. Um die Auswirkungen der Parallelisierungskosten zu verdeutlichen, werden nun noch einmal gemessene Laufzeiten bei der Symmetrischen Gruppe angegeben, wobei die Daten diesmal nicht in den globalen Speicher kopiert wurden.

Tabelle 3: Auswirkung von Parallelisierungskosten auf die Laufzeiten

Gruppe	Methode (I)		Methode (II)		sequentiell	
	vorher	nachher	vorher	nachher	vorher	nachher
S_{80}	0.1253	0.1376	0.1304	0.4516	0.3409	0.3483
S_{120}	0.5356	0.5412	0.5146	1.5220	1.4332	1.4310
S_{160}	1.5529	1.6286	1.4475	3.8570	4.1966	4.1594
S_{200}	3.5365	3.5458	3.1563	7.3848	9.1560	9.1202

Wie erwartet nimmt die Änderung keinen Einfluss auf die Laufzeit der sequentiellen Verifikation und die Unterschiede zwischen den Laufzeiten bei Methode eins sind äußerst gering. Die Auswirkungen auf Methode (II) sind jedoch deutlich. Grund hierfür ist die größere Anzahl an Prozessen, welche 20-mal größer ist als bei Methode (I). Dementsprechend gibt es auch weitaus mehr Kopiervorgänge, welche sich negativ auf die Laufzeit auswirken. Zudem sind die Datenmengen bei der Symmetrischen Gruppe hinreichend groß, was ebenfalls zu erhöhten Parallelisierungskosten beiträgt.

6. Offene Fragen und Ausblick

Die in Kapitel fünf vorgestellten Parallelisierungsansätze basieren im Wesentlichen auf der gewählten Verifikationsmethode (4.3.1). Es gibt jedoch noch weitere Verifikationsmethoden, wie z. B. die Verifikation mittels des Todd-Coxeter-Verfahrens. Interessant ist nun, ob sich diese Todd-Coxeter- Verifikation ebenfalls parallelisieren lässt, eventuell sogar unter Verwendung der Ansätze aus Kapitel fünf, oder ob eine Parallelisierung des Todd-Coxeter-Verfahrens an sich in diesem Fall zielführender wäre. Zur Untersuchung dessen wurden von mir bereits einige Implementierungen zum Rechnen mit Wörtern in freien Gruppen und eine Verifikation mittels des von GAP bereitgestellten Algorithmus zur Durchführung des Todd-Coxeter-Verfahrens vorgenommen. Jedoch konnte ich meine Arbeit in diesem Bereich bisher nicht zu Ende führen.
Weiterhin wurde festgestellt, dass die in Kapitel 5 vorgestellten Parallelisierungsansätze unter bestimmten Voraussetzungen – wie z. B. bei Transitivität der Gruppe – gute bis sehr gute Ergebnisse liefern. Inwieweit wäre es also möglich und insbesondere effizient, beide Ansätze in einem Verifikationsalgorithmus zu verwenden, basierend auf den vom randomisierten Schreier-Sims-Algorithmus (4.2.3) gelieferten Ergebnissen wie z. B. der Größe der Basis oder den Bahnenlängen $|b_q^{H^{(q)}}|,\ q \in \{1, \ldots, k\}$? Die Kombination beider Ansätze könnte z. B. so erfolgen, dass bei großen Bahnen der zweite und ansonsten der erste verwendet wird. Die Untersuchung eines solchen, aus beiden Parallelisierungsansätzen zusammengesetzten Verifikationsalgorithmus steht noch aus.
Außerdem wurde bisher nicht untersucht, wie sich die Wahl des Parameters $y \in \mathbb{N}$ bei Methode (II) auf die Laufzeit der parallelisierten Verifikation auswirkt und für welche Werte von y diese minimiert werden kann.
Zusätzlich zu den bisher genannten Problemen können noch weitere Verbesserungen an den von mir vorgenommenen Implementierungen vorgenommen werden. So empfiehlt es sich, Permutationen als Produkt der jeweiligen Erzeuger mittels Listen darzustellen und mit diesen anstatt Zyklen zu rechnen. Die Multiplikation zweier Permutationen kann dann durch einfaches Aneinanderhängen entsprechender Listen realisiert werden. Weiterhin lässt sich die Berechnung von Transversalen mittels sogenannter *Shallow Schreier Trees* effizienter gestalten. Ausführungen dazu findet man sowohl in [1] als auch in [4].

Literaturverzeichnis

[1] Derek F. Holt, Bettina Eick und Eamonn A. O'Brien. *Handbook of Computational Group Theory*. Discrete Mathematics and its Applications. Boca Raton: Chapman & Hall/CRC, 2005.

[2] Alexander Hulpke. *Notes on Computational Group Theory*. `http://www.math.colostate.edu/~hulpke/CGT/cgtnotes.pdf`, accessed 31-July-2016. 2010.

[3] Jeffrey S. Leon. „On an Algorithm for Finding a Base and a Strong Generating Set for a Group Given by Generating Permutations". In: *Mathematics of Computation* 35.151 (Juli 1980). `http://www.jstor.org/stable/2006206`, accessed 28-July-2016, S. 941–974.

[4] Ákos Seress. *Permutation Group Algorithms*. New York: Cambridge University Press, 2003.

A. Anhang - Implementierungen in GAP

A.1. Datei - Orbit.gi

A.1.1. Funktion - OrbitSimple

```
1  ##########################################################################
2  # Description: A function to calculate orbits, described in             #
3  #              section 3.1.                                              #
4  # Input: - integer 0 < a <= n + 1                                        #
5  #        - list of generators S                                          #
6  # Output: - list of integers 'orbit'                                     #
7  ##########################################################################
8
9  OrbitSimple := function(a, S)
10     local orbit, b, x;
11     orbit := [a];
12     for b in orbit do
13         for x in S do
14             if not b^x in orbit then
15                 Add(orbit, b^x);
16             fi;
17         od;
18     od;
19     return orbit;
20 end;
```

A.1.2. Funktion - OrbitSV

```
21 ##########################################################################
22 # Description: A function to calculate orbits and belonging Schreier-   #
23 #              vectors, described in section 3.1.                        #
24 # Input: - integer 0 < a <= n                                            #
25 #        - list of generators S                                          #
26 # Output: - list of integers 'orbit'                                     #
27 #         - belonging Schreier-vector 'sv' as list of integers           #
28 ##########################################################################
29
```

```
30 OrbitSV := function(a, S, n)
31     local orbit, b, x, i, sv;
32     orbit := [a];  sv := [1..n]*0;  sv[a] := -1;
33     for b in orbit do
34         for i in [1..Length(S)] do
35             if not b^S[i] in orbit then
36                 Add(orbit, b^S[i]);  sv[b^S[i]] := i;
37             fi;
38         od;
39     od;
40     return [orbit, sv];
41 end;
```

A.1.3. Funktion - Transversal

```
42 ##########################################################################
43 # Description: A function to calculate transversal elements, described#
44 #              in section 3.1.                                          #
45 # Input: - orbit element b of type integer 0 < b <= n                   #
46 #        - Schreier-vector 'sv' as list of integers in range [1..n]     #
47 #        - list of generators S                                         #
48 # Output: - transversal element of b w.r.t. a and <S>                   #
49 ##########################################################################
50
51 Transversal := function(b, sv, S)
52     local u, k;
53     if sv[b] = 0 then
54         return false;
55     fi;
56     u := ();  k := sv[b];
57     while not k = -1 do
58         u := S[k]*u;  b := b^(S[k]^-1);  k := sv[b];
59     od;
60     return u;
61 end;
```

A.2. Datei - RandPerm.gi

A.2.1. Funktion - InitGenerator

```
1  ##############################################################################
2  # Description: A function to initialize a generator for random              #
3  #              permutations, described in section 4.1.                       #
4  # Input: - list of generators S                                              #
5  # Output: - list of permutations X including repetitions of generators#
6  #           from S                                                           #
7  ##############################################################################
8
9  InitGenerator := function(S)
10     local i, X, k;
11     X := List(S); k := Length(X);
12     for i in [k+1..Maximum(11, Length(S))] do
13         Add(X, X[i-k]);
14     od;
15     if IsReadOnlyGlobal("X") = true then
16         MakeReadWriteGlobal("X");
17     fi;
18     return X;
19 end;
```

A.2.2. Funktion - RandomPerm

```
20 ##############################################################################
21 # Description: A function to calculate a random permutation,                #
22 #              described in section 4.1.                                     #
23 # Input: - list of generators S, generator X returned by InitGenerator#
24 # Output: - random permutation a as cycle                                    #
25 ##############################################################################
26
27 RandomPerm := function(S, X)
28     local Randomize, a, r, i;
29
30     r := Maximum(11, Length(S));
31     Randomize := function()
32         local s, t, e, range;
```

```
33
34          # indices s and t specify which permutations X[s], X[t]
35          # will be multiplied with each other
36          # e specifies whether X[t] or its inverse will be used
37          range := [1..r];  s := Random(range);  e := Random([-1,1]);
38          t := Random(range);  Remove(range, s);
39
40          # alternate the order of multiplication
41          # of X[t] and X[s] randomly
42          # update generator X
43
44          if Random([1,2]) = 1 then
45             X[s] := X[s]*X[t]^e;
46             a := a*X[s];
47          else
48             X[s] := X[t]^e*X[s];
49             a := X[s]*a;
50          fi;
51          return a;
52      end;
53
54      # calculate random permutation a as
55      # product of random permutations from X
56      a := ();
57      for i in [1..50] do
58          Randomize();
59      od;
60      return a;
61  end;
```

A.3. Datei - SchreierSims.gi

```
# Using functions from 'RandPerm.gi' and 'Orbit.gi'
Read("RandomPerm.gi"); Read("Orbit.gi");
```

A.3.1. Funktion - BaseImage

```
##############################################################################
# Description: A function to calculate the pointwise image of a list.       #
# Input: - list of integers B = [b_1,...,b_k]                                #
#        - permutation x as cycle                                            #
# Output: - list of integers [b_1^x,...,b_k^x]                               #
##############################################################################

BaseImage := function(B, x)
    local i, image;
    image := [];
    for i in [1..Length(B)] do
        Add(image, B[i]^x);
    od;
    return image;
end;
```

A.3.2. Funktion - GeneratingSets

```
##############################################################################
# Description: A function to setup partial generating sets of                #
#              stabilizeres.                                                 #
# Input: - list of integers B = [b_1,...,b_k]                                #
#        - generating system S as list of permutations                       #
# Output: - list of generating system (lists of permutations),               #
#           one generating system for each stabilizer in chain               #
##############################################################################

GeneratingSets := function(B, S)
    local s, x, i, partialBase, sets;
    sets := [S];
    for i in [1..Length(B)-1] do
        Add(sets, []);
```

```
32      od;
33      for x in S do
34          for i in [1..Length(B)-1] do
35              partialBase := B{[1..i]};
36              if BaseImage(partialBase, x) = partialBase then
37                  Add(sets[i+1], x);
38              else
39                  break;
40              fi;
41          od;
42      od;
43      return sets;
44  end;
```

A.3.3. Funktion - Strip

```
45  ##############################################################################
46  # Description: A function to strip a permutation w.r.t. a base,             #
47  #              described in section 3.2.                                     #
48  # Input: - permutation g as cycle                                            #
49  #        - list of integers B = [b_1,...,b_k]                                #
50  #        - generating system S as list of permutations                       #
51  #        - data structure returned by 'GeneratingSets' function              #
52  # Output: - residue and dropout level of g                                   #
53  ##############################################################################
54
55  Strip := function(g, B, orbits, sgs)
56      local m, i, u, b;
57      m := g;
58      for i in [1..Length(B)] do
59          b := B[i]^m;
60          if orbits[i][2][b] = 0 then
61              return [m, i];
62          fi;
63          u := Transversal(b, orbits[i][2], sgs[i]);  m := m*(u^(-1));
64      od;
65      return [m, Length(B)+1];
66  end;
```

A.3.4. Funktion - SchreierSims

```
67  ##########################################################################
68  # Description: The standard deterministic Schreier-Sims-Algorithm      #
69  #              described in section 3.3.                                #
70  # Input: - list B of base points (might be empty list [] first)        #
71  #        - list S of given generators of group G                       #
72  #        - integer n specifying the degree of symmetric group the      #
73  #          generated group G is embedded in                            #
74  # Output: - list 'BSGS' with three entries including                   #
75  #           -- the calculated base B as list in BSGS[1]                #
76  #           -- a strong generating set S as list of partial            #
77  #              generating sets for each stabilizer in BSGS[2]          #
78  #           -- a list of the sizes of the orbits belonging to B        #
79  #              and S in BSGS[3]                                        #
80  ##########################################################################
81
82  SchreierSims := function(B, S, n)
83      local x, u, y, h, b, i, j, k, t, orbits, sgs, sizes, stripped;
84
85      # extending partial base so that no point of S
86      # fixes all points
87      for x in S do
88          if BaseImage(B, x) = B then
89              Add(B, MovedPoints(x)[1]);
90          fi;
91      od;
92      # calculating generating sets for each stabilizer
93      sgs := GeneratingSets(B, S);
94
95      # calculate orbits and schreier vectors of base
96      # points under action of generated stabilizers
97      orbits := [];
98      for i in [1..Length(B)] do
99          Add(orbits, OrbitSV(B[i], sgs[i], n));
100     od;
101     i := Length(B);
102     while i >= 1 do
```

```
103         for b in orbits[i][1] do
104             t := Transversal(b, orbits[i][2], sgs[i]);
105             for x in sgs[i] do
106                 u := Transversal(b^x, orbits[i][2], sgs[i]);  y := true;
107                 if not t*x = u then
108                     # check whether Schreier generator
109                     # belongs to H^(i+1)
110                     stripped := Strip(t*x*(u^(-1)), B, orbits, sgs);
111                     j := stripped[2];  h := stripped[1];
112 
113                     # in case the Schreier generator does
114                     # not belong to H^(i+1) do ...
115                     if j <= Length(B) then
116                         y := false;
117                     elif not h = () then
118                         Add(B, MovedPoints(h)[1]);
119                         Add(sgs, []);  y := false;
120                     fi;
121                     if y = false then
122                         # update generating sets, orbits
123                         # and Schreier vectors
124                         for k in [i+1..j] do
125                             Add(sgs[k], h);
126                             orbits[k] := OrbitSV(B[k], sgs[k], n);
127                         od;
128                         i := j;  break;
129                     fi;
130                 fi;
131             od;
132             if y = false then
133                 break;
134             fi;
135         od;
136         if y = false then
137             continue;
138         fi;
139         i := i - 1;
140     od;
```

```
141     sizes := [];
142     for i in [1..Length(B)] do
143        Add(sizes, Length(orbits[i][1]));
144     od;
145     return [B, sgs, sizes];
146 end;
```

A.3.5. Funktion - RandomSchreierSims

```
147 ##########################################################################
148 # Description: The randomized Schreier-Sims-Algorithm,                   #
149 #              described in section 4.2.                                 #
150 # Input: - list B of base points (might be empty list [] first)          #
151 #        - list S of given generators of group G                         #
152 #        - an integer 'bound' capping the number of elements             #
153 #          getting stripped                                              #
154 #          with trivial residue in a row                                 #
155 #        - integer n specifying the degree of symmetric group the        #
156 #          generated group G is embedded in                              #
157 # Output: - list 'BSGS' with three entries including                     #
158 #           -- the calculated base B as list in BSGS[1]                  #
159 #           -- a strong generating set S as list of partial              #
160 #              generating sets for each stabilizer in BSGS[2]            #
161 #           -- a list of the sizes of the orbits belonging to B          #
162 #              and S in BSGS[3]                                          #
163 ##########################################################################
164
165 RandomSchreierSims := function(B, S, bound, n)
166     local x, g, i, h, X, c, j, k, y, sgs, orbits, sizes, SGS, stripped;
167
168     for x in S do
169         if BaseImage(B, x) = B then
170             Add(B, MovedPoints(x)[1]);
171         fi;
172     od;
173     sgs := GeneratingSets(B, S);  orbits := [];  SGS := ShallowCopy(S);
174
175     for i in [1..Length(B)] do
```

```
176          Add(orbits, OrbitSV(B[i], sgs[i], n));
177      od;
178
179      X := InitGenerator(S);  c := 0;
180      while c < bound do
181          g := RandomPerm(S, X);
182          stripped := Strip(g, B, orbits, sgs);
183          j := stripped[2];  h := stripped[1];  y := true;
184          if j <= Length(B) then
185              y := false;
186          elif not h = () then
187              y := false;  Add(B, MovedPoints(h)[1]);  Add(sgs, []);
188          fi;
189          if y = false then
190              Add(SGS, h);
191              for k in [2..j] do
192                  Add(sgs[k], h);
193                  orbits[k] := OrbitSV(B[k], sgs[k], n);
194              od;
195              c := 0;
196          fi;
197          c := c + 1;
198      od;
199
200      sizes := [];  sgs[1] := SGS;
201      for i in [1..Length(B)] do
202          Add(sizes, Length(orbits[i][1]));
203      od;
204      return [B, sgs, sizes];
205  end;
```

A.3.6. Funktion - RemoveRedundantGens

```
206 ##########################################################################
207 # Description: A function to remove redundant generators from            #
208 #              a strong generating set, described in section 4.2.        #
209 # Input: - BSGS data structure returned by RandomSchreierSims            #
210 #        - Generating set which defined the group. NOT the strong        #
211 #          generating set returned by RandomSchreierSims!                #
212 #        - integer n specifying the size of the set the group            #
213 #          operates on                                                   #
214 ##########################################################################
215
216 RemoveRedundantGens := function(BSGS, initialGenSet, n)
217     local orbits, i, g, newGenSys, removeGen;
218     orbits := [];
219
220     removeGen := function(SGS, gen, level)
221         local j;
222         for j in [1..level] do
223             RemoveSet(SGS[j], gen);
224         od;
225     end;
226     for i in [1..Length(BSGS[1])] do
227         BSGS[2][i] := Set(BSGS[2][i]);
228         Add(orbits, Set(OrbitSimple(BSGS[1][i], BSGS[2][i])));
229     od;
230     for i in Reversed([1..Length(BSGS[1])-1]) do
231         for g in BSGS[2][i] do
232             if not g in BSGS[2][i+1]
233                and not g in initialGenSet then
234                 newGenSys := Set(BSGS[2][i]);
235                 RemoveSet(newGenSys, g);
236                 if Set(OrbitSimple(BSGS[1][i], newGenSys)) = orbits[i]
237                         then removeGen(BSGS[2], g, i);
238                 fi;
239             fi;
240         od;
241     od;
```

```
242     orbits := [];
243     for i in [1..Length(BSGS[1])] do
244         Add(orbits, OrbitSV(BSGS[1][i], BSGS[2][i], n));
245     od;
246     Add(BSGS, orbits);  MakeImmutable(BSGS);
247 end;
```

A.3.7. Funktion - SimpleVerify

```
248 ##############################################################################
249 # Description: The sequential deterministic verification,                    #
250 #              described in section 4.3                                      #
251 # Input: - BSGS data structure returned by RandomSchreierSims                #
252 #        - integer m specifying which layer H^{(m)}_{b_m} = H^{(m+1)} holds  #
253 #          has to be verified                                                #
254 # Output: - boolean, true if H^{(m)}_{b_m} = H^{(m+1)} holds,                #
255 #           else false                                                       #
256 ##############################################################################
257
258 SimpleVerify := function(BSGS, m)
259     local x, u, b, t, orbits, gen, stripped, sgs, B;
260
261     B := BSGS[1]; sgs := BSGS[2]; orbits := BSGS[4];
262     for b in orbits[m][1] do
263         t := Transversal(b, orbits[m][2], sgs[m]);
264         for x in sgs[m] do
265             if not x in sgs[m+1] or orbits[m+1][2][b] = 0 then
266                 u := Transversal(b^x, orbits[m][2], sgs[m]);
267                 if not t*x = u then
268                     gen := t*x*(u^(-1));
269                     # check whether Schreier generator 'gen'
270                     # belongs to H^(i+1)
271                     stripped := Strip(gen, B, orbits, sgs);
272                     # if not, return
273                     if stripped[2] <= Length(B) then
274                         return false;
275                     elif not stripped[1] = () then
276                         return false;
```

```
277                     fi;
278                 fi;
279             fi;
280         od;
281     od;
282     return true;
283 end;
```

A.3.8. Funktion - ParallelVerify

```
284 ##########################################################################
285 # Description: The deterministic verification, parallelized w.r.t.       #
286 #              the length of the basic orbits, described as method 2     #
287 #              in section 5.1.                                           #
288 # Input: - BSGS data structure returned by RandomSchreierSims            #
289 #        - integer m specifying which layer H^{(m)}_{b_m} = H^{(m+1)}    #
290 #          has to be verified                                            #
291 # Output: - boolean, true if H^{(m)}_{b_m} = H^{(m+1)} holds,            #
292 #           else false                                                   #
293 ##########################################################################
294
295 ParallelVerify := function(BSGS, m)
296     local b, i, orbits, sgs, B, Verify, tasks, results, numOfTasks,
297           remaining, partition, pointsPerTask;
298     # the well-known deterministic verify routine
299     Verify := function(B, sgs, orbits, m, partition)
300         local x, u, i, stripped, gen, t;
301         for b in partition do
302             t := Transversal(b, orbits[m][2], sgs[m]);
303                 for x in sgs[m] do
304                     if not x in sgs[m+1] or orbits[m+1][2][b] = 0 then
305                         u := Transversal(b^x, orbits[m][2], sgs[m]);
306                         if not t*x = u then
307                             gen := t*x*(u^(-1));
308                             stripped := Strip(gen, B, orbits, sgs);
309                             if stripped[2] <= Length(B) then
310                                 return false;
311                             elif not stripped[1] = () then
```

```
312                                     return false;
313                                 fi;
314                             fi;
315                         fi;
316                     od;
317             od;
318             return true;
319         end;
320         B := BSGS[1]; sgs := BSGS[2];
321         orbits := BSGS[4]; partition := [];
322
323         pointsPerTask := Int(Length(orbits[m][1])*0.05) + 1;
324         numOfTasks := Int(Length(orbits[m][1])/pointsPerTask) + 1;
325         remaining := Length(orbits[m][1]) mod pointsPerTask;
326         tasks := [1..numOfTasks]; results := [1..numOfTasks];
327
328         # slice orbit into 'numOfTask' many nearly equally sized subsets
329         for i in [1..numOfTasks-1] do
330             Add(partition, orbits[m][1]{[(i-1)*pointsPerTask+1..
331                                             i*pointsPerTask]});
332         od;
333         Add(partition, orbits[m][1]{[(numOfTasks-1)*pointsPerTask+1..
334                         ((numOfTasks-1)*pointsPerTask+remaining)]});
335         # start several verifications ...
336         for i in [1..numOfTasks] do
337             tasks[i] := RunTask(Verify, B, sgs, orbits, m, partition[i]);
338         od;
339
340         # ... and evaluate their results afterwards
341         for i in [1..numOfTasks] do
342             results[i] := TaskResult(tasks[i]);
343         od;
344         if false in results then
345             return false;
346         else
347             return true;
348         fi;
349     end;
```

A.4. Datei - Verify.gi

```
Read("SchreierSims.gi");
```

A.4.1. Funktion - Method1

```
##########################################################################
# Description: The deterministic verification, parallelized w.r.t.      #
#              base points, described as method 1 in section 5.1.       #
# Input: - BSGS data structure returned by RandomSchreierSims            #
# Output: - boolean, true if BSGS is correct, else false                 #
##########################################################################

Method1 := function(BSGS)
    local l, tasks, results;

    Print("Start verification.\n");
    Print("Apply parallel computing method w.r.t. base points.\n");

    tasks := [1..Length(BSGS[1])];  results := [1..Length(BSGS[1])];

    for l in [1..Length(BSGS[1])] do
        tasks[l] := RunTask(SimpleVerify, BSGS, l);
    od;
    for l in [1..Length(BSGS[1])] do
        results[l] := TaskResult(tasks[l]);
    od;

    Print("\nVerification done.");

    if false in results then
        Print("\nBSGS is wrong!\n");
    else
        Print("\nCorrectness of BSGS verified!\n");
    fi;
    Print("*******************************************************\n");
end;
```

A.4.2. Funktion - Method2

```
33 ##########################################################################
34 # Description: The deterministic verification, parallelized w.r.t.      #
35 #              orbit points, described as method 2 in section 5.1.      #
36 # Input: - BSGS data structure returned by RandomSchreierSims           #
37 # Output: - boolean, true if BSGS is correct, else false                 #
38 ##########################################################################
39 Method2 := function(BSGS)
40     local l, tasks, results;
41 
42     Print("Start verification.\n");
43     Print("Apply parallel computing method w.r.t. orbit points.\n");
44 
45     tasks := [1..Length(BSGS[1])];  results := [1..Length(BSGS[1])];
46 
47     for l in Reversed([1..Length(BSGS[1])]) do
48         results[l] := ParallelVerify(BSGS, l);
49         if false in results then
50             break;
51         fi;
52     od;
53 
54     Print("\nVerification done.");
55 
56     if false in results then
57         Print("\nBSGS is wrong!\n");
58     else
59         Print("\nCorrectness of BSGS verified!\n");
60     fi;
61     Print("*******************************************************\n");
62 end;
```

B. Anhang - Anleitung zur Verwendung der Implementierungen

In diesem Abschnitt soll kurz auf die Ausführung der Implementierungen aus Anhang A eingegangen werden. Es folgen die zur Ausführung benötigten Programme und Software-Pakete:

- **HPC-GAP**[9] (Version 4.dev vom 09.06.2016 oder eine aktuellere Version) und
- das Package **AtlasGroups**.

Anhand eines Beispiels soll nun gezeigt werden, wie der randomisierte Schreier-Sims-Algorithmus mitsamt Verifikation mittels der Implementierungen aus Anhang A auf eine Gruppe angewendet wird.

Beispiel. Es sei $G := M_{11}$. Nun soll eine Basis für G mit Hilfe des randomisierten Schreier-Sims- Algorithmus' berechnet werden. Dazu müssen zunächst die Dateien 'SchreierSims.gi' sowie 'Verify.gi' und das Paket 'AtlasRep' eingelesen werden.

```
1 gap> LoadPackage("AtlasRep");;
2 gap> Read("SchreierSims.gi");;  Read("Verify.gi");;
```

Weiter müssen einige Bezeichner zur Spezifikation der Gruppe eingeführt werden.

```
3 gap> G := AtlasGroup("M11");;
4 gap> S := Set(GeneratorsOfGroup(G));;
5 gap> n := LargestMovedPoint(G);;
6 gap> B := [];;
```

Auf diese kann nun der randomisierte Schreier-Sims-Algorithmus angewendet werden. Die Ergebnisse werden unter dem Bezeichner 'BSGS' gespeichert.

```
7 gap> BSGS := RandomSchreierSims(B, S, 6, n);;
```

In diesem Beispiel beträgt die Wahrscheinlichkeit dafür, dass die gelieferten Ergebnisse falsch sind, höchstens 2^{-6}. Mittels 'BSGS' kann nun auf die Ergebnisse zugegriffen werden.

[9] Eine Anleitung zum Download und zur Installation von HPC-GAP findet man unter https://github.com/gap-system/gap/wiki/Building-HPC-GAP (Zugriff 21.08.2016)

So wurde z. B. folgende Basis berechnet:

```
gap> BSGS[1];
[ 2, 1, 3, 4 ]
```

Das zugehörige starke Erzeugendensystem erhält man wie folgt:

```
gap> BSGS[2][1];
[ (4,8,5,11)(6,7,10,9), (4,10,5,6)(7,11,9,8), (3,6,11,5)(4,10,9,7),
  (2,10)(4,11)(5,7)(8,9), (1,4,3,8)(2,5,6,9), (1,11)(4,10)(5,9)(6,7) ]
```

Die Größe der zur Basis und dem starken Erzeugendensystem gehörigen Bahnen erhält man mit folgendem Befehl:

```
gap> BSGS[3];
[ 11, 10, 9, 8 ]
```

Man sieht hier, dass G vierfach transitiv auf $\{1, \ldots, 11\}$ operiert.
Zudem gilt $|G| = 7\,920 = 11 \cdot 10 \cdot 9 \cdot 8$, die Ordnung von G ist also das Produkt der Bahnenlängen. Das Ergebnis des randomisierten Algorithmus' wird also wohl richtig sein. Dennoch soll das Ergebnis mittels der vorgestellten Verifikationsmethoden überprüft werden. Allerdings sollen zuvor noch überflüssige Erzeuger aus dem starken Erzeugendensystem entfernt werden.

```
gap> RemoveRedundantGens(BSGS, S, n);
```

Es soll nun die erste Verifikationsmethode verwendet werden, welche in Abschnitt (5.1) vorgestellt wurde. Die Anwendung dieser liefert das folgende Resultat.

```
gap> Method1(BSGS);
Start verification.
Apply parallel computing method w.r.t. base points.

Verification done.
Correctness of BSGS verified!
```

Die vom randomisierten Algorithmus gelieferten Ergebnisse sind also – wie bereits erwartet– richtig.

Selbstständigkeitserklärung

Hiermit versichere ich, dass ich die vorliegende Bachelorarbeit selbstständig verfasst, keine anderen als die angegebenen Quellen und Hilfsmittel benutzt sowie Zitate kenntlich gemacht habe.
Diese Arbeit wurde noch nicht in gleicher oder ähnlicher Fassung in einem anderen Studiengang als Prüfungsleistung vorgelegt.

Halle(Saale), den 23.08.2016 .....

Alexander S. J. Klemps