Universidade Federal de Goiás
Instituto de Matemática e Estatística

Angie Tatiana Suárez Romero

Computação em grupos de permutação finitos com GAP

Goiânia
2018

sistema de bibliotecas ufg

PRPG
PRÓ-REITORIA DE PÓS-GRADUAÇÃO

UFG

TERMO DE CIÊNCIA E DE AUTORIZAÇÃO PARA DISPONIBILIZAR VERSÕES ELETRÔNICAS DE TESES E DISSERTAÇÕES NA BIBLIOTECA DIGITAL DA UFG

Na qualidade de titular dos direitos de autor, autorizo a Universidade Federal de Goiás (UFG) a disponibilizar, gratuitamente, por meio da Biblioteca Digital de Teses e Dissertações (BDTD/UFG), regulamentada pela Resolução CEPEC nº 832/2007, sem ressarcimento dos direitos autorais, de acordo com a Lei nº 9610/98, o documento conforme permissões assinaladas abaixo, para fins de leitura, impressão e/ou *download*, a título de divulgação da produção científica brasileira, a partir desta data.

1. Identificação do material bibliográfico: **[x] Dissertação** **[ ] Tese**

2. Identificação da Tese ou Dissertação:

Nome completo do autor: Angie Tatiana Suárez Romero

Título do trabalho: Computação em grupos de permutação finitos com GAP

3. Informações de acesso ao documento:

Concorda com a liberação total do documento [x] SIM [ ] NÃO[1]

Havendo concordância com a disponibilização eletrônica, torna-se imprescindível o envio do(s) arquivo(s) em formato digital PDF da tese ou dissertação.

Assinatura do(a) autor(a)[2]

Ciente e de acordo:

Assinatura do(a) orientador(a)[2]

Data: _13/ 03 / 2018

[1] Neste caso o documento será embargado por até um ano a partir da data de defesa. A extensão deste prazo suscita justificativa junto à coordenação do curso. Os dados do documento não serão disponibilizados durante o período de embargo.

Casos de embargo:
- Solicitação de registro de patente;
- Submissão de artigo em revista científica;
- Publicação como capítulo de livro;
- Publicação da dissertação/tese em livro.

[2] A assinatura deve ser escaneada.

Versão atualizada em setembro de 2017.

Angie Tatiana Suárez Romero

Computação em grupos de permutação finitos com GAP

Dissertação apresentada ao Programa de Pós–Graduação do Instituto de Matemática e Estatística da Universidade Federal de Goiás, como requisito parcial para obtenção do título de Mestre em Matemática.

Área de concentração: Álgebra.

Orientador: Prof. Ricardo Nunes de Oliveira

Goiânia
2018

Ficha de identificação da obra elaborada pelo autor, através do
Programa de Geração Automática do Sistema de Bibliotecas da UFG.

Suarez Romero, Angie Tatiana
Computação em grupos de permutação finitos com GAP [manuscrito] / Angie Tatiana Suarez Romero. - 2018.
CI, 101 f.: il.

Orientador: Prof. Dr. Ricardo Nunes de Oliveira .
Dissertação (Mestrado) - Universidade Federal de Goiás, Instituto de Matemática e Estatística (IME), Programa de Pós-Graduação em Matemática, Goiânia, 2018.
Bibliografia. Apêndice.
Inclui símbolos, gráfico, algoritmos.

1. Teoría de grupos. 2. grupos de permutação finitos. 3. álgebra computacional. I. , Ricardo Nunes de Oliveira, orient. II. Título.

CDU 512.5

UFG

UNIVERSIDADE FEDERAL DE GOIÁS
INSTITUTO DE MATEMÁTICA E ESTATÍSTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM MATEMÁTICA
Campus Samambaia – Caixa Postal 131 – CEP: 74.001-970 – Goiânia-GO.
Fones: (62) 3521-1208 e 3521-1137 www.ime.ufg.br

ATA DA REUNIÃO DA BANCA EXAMINADORA DA DEFESA DE DISSERTAÇÃO DE ANGIE TATIANA SUAREZ ROMERO – Aos cinco do mês de março do ano de dois mil e dezoito (05/03/2018), às 10 horas, reuniram-se os componentes da Banca Examinadora: Prof. Ricardo Nunes de Oliveira - Orientador, Profa. Ana Paula de Araújo Chaves e Prof. Raimundo de Araújo Bastos Júnior (via videoconferência), para, sob a presidência do primeiro, e em sessão pública realizada no Lemat do Instituto de Matemática e Estatística, procederem a avaliação da defesa de dissertação intitulada: **"Computação em grupos de permutação finitps com GAP"**, em nível de Mestrado, área de concentração em Álgebra, de autoria de Angie Tatiana Suarez Romero, discente do Programa de Pós-Graduação em Matemática da Universidade Federal de Goiás. A sessão foi aberta pelo Presidente da Banca, Prof. Ricardo Nunes de Oliveira, que fez a apresentação formal dos membros da Banca. A seguir, a palavra foi concedida ao autor da dissertação que, em 45 minutos procedeu a apresentação de seu trabalho. Terminada a apresentação, cada membro da Banca arguiu o examinando, tendo-se adotado o sistema de diálogo sequencial. Terminada a fase de arguição, procedeu-se a avaliação da defesa. Tendo-se em vista o que consta na Resolução nº. 1513 do Conselho de Ensino, Pesquisa, Extensão e Cultura (CEPEC), que regulamenta o Programa de Pós-Graduação em Matemática e procedidas as correções recomendadas, a dissertação foi **APROVADA** por unanimidade, considerando-se integralmente cumprido este requisito para fins de obtenção do título de **MESTRE EM MATEMÁTICA**, na área de concentração em Álgebra, pela Universidade Federal de Goiás. A conclusão do curso dar-se-á quando da entrega na secretaria do PPGM da versão definitiva da dissertação, com as devidas correções supervisionadas e aprovadas pelo orientador. Cumpridas as formalidades de pauta, às 12 horas a presidência da mesa encerrou esta sessão de defesa de dissertação e para constar eu, Flávia Magalhães Freire, secretária do PPGM, lavrei a presente Ata que, depois de lida e aprovada, será assinada pelos membros da Banca Examinadora em quatro vias de igual teor.

Prof. Dr. Ricardo Nunes de Oliveira
Presidente - IME/UFG

Prof. Dr. Ana Paula de Araújo Chaves
Membro – IME/UFG

Prof. Dr. Raimundo de Araújo Bastos Júnior
Membro – DM/UnB
(via videoconferência)

ANGIE TATIANA SUAREZ ROMERO

COMPUTAÇÃO EM GRUPOS DE PERMUTAÇÃO FINITOS COM GAP

Dissertação defendida no Programa de Pós-Graduação do Instituto de Matemática e Estatística da Universidade Federal de Goiás como requisito parcial para obtenção do título de Mestre em Matemática, aprovada no dia 05 de março de 2018, pela Banca Examinadora constituída pelos professores:

Prof. Dr. Ricardo Nunes de Oliveira
Instituto de Matemática e Estatística - UFG
Presidente da Banca

Ana Paula de A. Chaves

Profa. Dra. Ana Paula de Araújo Chaves
Instituto de Matemática e Estatística - UFG

Raimundo de Araújo Bastos Júnior

Prof. Dr. Raimundo de Araújo Bastos Júnior
Departamento de Matemática – UnB
(Via videoconferência)

Angie Tatiana Suárez Romero

Bacharel em Matemática pela Universidade Distrital Francisco José de Caldas de Bogotá - UD.

A minha irmã Valentina Suárez.

Agradecimentos

Antes de todo, agradeço a minha tia Alicia Romero pela motivação nos meus estudos profissionais e pela força que sempre ofereço para não desistir ao longo do caminho, mesmo assim a meu tio Manuel Romero e meu pai Enrique Suárez.

Agradeço ao meu orientador Prof. Dr. Ricardo Nunes pela orientação e motivação no desenvolvimento do trabalho.

Agradeço à minhã família pela confiança e apoio que sempre teve deles. Em especial à minha irmã Valentina.

Agradeço ao Martín Barajas pela boa companhía, ajuda e motivaço.

Agradeço a meus amigos e colegas, Joan, Tulio, Carlos, Jeferson, Milton, Mayk, Ana Maria e Carolina pela companhia, muitas conversas e ajuda neste tempo.

Agradeço à CAPES e à CNPQ pelo apoio econômico.

A iluminação súbita, é um longo manifesto de um longo trabalho subconsciente anterior.

Henri Poincaré,

.

Resumo

Angie Tatiana Suárez Romero. **Computação em grupos de permutação finitos com GAP**. Goiânia, 2018. 102p. Dissertação de Mestrado. Instituto de Matemática e Estatística, Universidade Federal de Goiás.

O Teorema de Cayley nos permite representar um grupo finito como grupo de permutações de um conjunto finito de pontos. De forma geral, uma ação de um grupo finito G em um conjunto finito Ω, é descrita como uma aplicação do grupo G no grupo simétrico $Sym(\Omega)$. Neste trabalho vamos descrever alguns algoritmos para grupos de permutação e implementa-los no sistema GAP. Começamos descrevendo uma maneira de representar grupos em computadores, calculamos órbitas, estabilizadores na forma básica e por meio de vetores de Schreier. Posteriormente fazemos algoritmos para trabalhar com grupos transitivos e primitivos, chegando assim ao conceito de, base e conjunto gerador forte ($BSGS$) para grupos de permutação finitos com o algoritmo SCHREIER-SIMS. No final trabalhamos com homomorfismos de grupos e encontramos os elementos de um grupo mediante pesquisas backtrack.

Palavras–chave

Teoría de grupos, grupos de permutação finitos, álgebra computacional.

Abstract

Angie Tatiana Suárez Romero. **Computation in finite permutation groups with GAP**. Goiânia, 2018. 102p. MSc. Dissertation. Instituto de Matemática e Estatística, Universidade Federal de Goiás.

Cayley's theorem allows us to represent a finite group as a permutations group of a finite set of points. In general, an action of a finite group G in a finite set, is described as an application of the group G in the symmetric group $Sym(\Omega)$. In this work we will describe some algorithms for permutation groups and implement them in the GAP system. We begin by describing a way of representing groups in computers, we calculate orbits, stabilizers in the basic form and by means of Schreier's vectors. Later we make algorithms to work with primitive and transitive groups, thus arriving at the concept of BSGS, base and strong generator set, for permutation groups with the algorithm SCHREIERSIMS. In the end we work with group homomorphisms, we find the elements of a group through backtrack searches.

Keywords

Theory of groups, finite permutation groups, computational algebra

Introdução

A teoria do grupos computacional, tem as origens no final do século XIX e início do século XX, mas tem estado desenvolvendo-se especialmente nos últimos 30 a 40 anos. As principais áreas da teoria de grupos computacional são os algoritmos para grupos finitamente apresentados, grupos policíclicos e finitos solúveis, grupos de permutação, grupos de matrizes e teoria da representação. No livro [21] pode-se encontrar uma versão mais geral da teoria de grupos computacional na área dos grupos de permutação. Os grupos de permutação são o tipo mais antigo de representações de grupos; de fato, o trabalho de Galois em grupos de permutação, que geralmente é considerado como o início da teoria do grupo como um ramo separado da matemática, precedeu a definição abstrata dos grupos. Cálculo do grupo de Galois, e o problema relacionado de determinar todos os grupos de permutação transitiva de um determinado grau, ainda são áreas ativas da pesquisa, como pode ser visto pelo trabalho do Alexander Hulpke nos últimos anos.

As ideias básicas para trabalhar com grupos de permutação apareceram com o trabalho do Otto Schreier e complementado pelo Charles Sims, encontradas nos artigos [22] e [23]. Otto Schreier foi um matemático austríaco do começo do seculo XX, nascido no ano 1901 na Viena e morreu na Alemanha no ano 1929, neste pouco tempo ele fez uma das maiores contribuições nesta teoria a qual retomou Charles Sims no ano 1970.

Ákos Seress, matemático europeu da Hungria, um dos mais ressentes pesquisadores nesta área, no artigo [2] fala que esta é a área da teoria de grupos computacional onde a análise de complexidade de algoritmos é a mais desenvolvida. A razão para isso é a conexão com o célebre problema de isomorfismo do gráfico. O trabalho mais importante nos últimos anos é feito pelo László Babai que em dezembro de 2015 afirmou ter mostrado que a complexidade algorítmica do problema do isomorfismo gráfico é quasipolinômica, onde um pouco mais tarde o matemático peruano Harald A. Helfgott verificou a demonstração em detalhes e afirma que está correto. Embora o artigo [4] foi publicado pelo Babai no 2016, a

versão mais ressente do trabalho [13] foi sometido pelo Helfgott em outubro do 2017.

O conteúdo desta dissertação esta dividido em três capítulos e um apêndice do que é o sistema GAP. No primeiro capítulo vamos encontrar alguns conceitos importantes da teoria de grupos e da computação. A forma de aplicar o teorema de Cayley é por médio da ação de grupos em conjuntos, pois são as ferramentas das ações as que vai-nos permitir juntar a teoria de grupos com a computação, assim esta seção é uma das mais importantes do capítulo. Outro conceito apresentado neste primeiro capítulo, é a definição de base e conjunto gerador forte ($BSGS$), pois é o fundamento do algoritmo $SCHREIER-SIMS$. No capítulo dois vamos encontrar um procedimento inicial para gerar alguns elementos de um grupo finitamente apresentado de uma maneira pseudo-aleatória. No começo do capítulo mostra-se as diferentes maneiras de trabalhar com grupos em um computador, ali encontramos os primeiros algoritmos do trabalho e sua aplicação no sistema GAP.

No terceiro capítulo vamos encontrar o conteúdo forte do trabalho, isto é, os principais algoritmos dos grupos de permutação finita. Ali começamos com os cálculos das órbitas e os estabilizadores de uma ação, seguindo com os vetores se Schreier para armazenar órbitas e transversais de uma maneira mais eficiente. Logo, vamos encontrar um algoritmo para verificar se uma grupo transitivo de grau maior do que 8 é isomorfo a um grupo simétrico ou alternante. Para verificar se um grupo transitivo também é primitivo, é necessário trabalhar com os sistemas de blocos, estes são os algoritmos que vamos encontrar na seção 3.4. Mais adiante, na seção 3.5 encontramos o algoritmo $SCHREIER-SIMS$ com algumas das muitas aplicações que tem. Para finalizar o capítulo encontramos alguns métodos para trabalhar com homomorfismos de grupos e bem no final uma introdução de como utilizar a pesquisa Backtrack, por exemplo, para gerar os elementos de um grupo.

Sumário

Lista de Algoritmos

Lista de Códigos GAP

Lista de símbolos

α^G	Órbita de $\alpha \in \Omega$ em G.
α^g	Ação de $g \in G$ em $\alpha \in \Omega$.
$\Delta^{(i)}$	i-ésima órbita básica.
A^*	Conjunto das cadeias $x_1x_2...x_r$ com cada $x_i \in A$.
$Alt(\Omega)$	Grupo alternante dos elementos de Ω.
G/H, $\frac{G}{H}$	Grupo quociente de G por um subgrupo normal H.
$G \cong H$	G é isomorfo a H.
$G^{(i)}$	Estabilizador dos primeiros $i-1$ elementos da base.
G^{Ω}	Imagem da ação de G em Ω.
$G_{(\Delta)}$	Estabilizador pontoal dos elementos do conjunto Δ.
G_α	Estabilizador do elemento α.
G_Δ	Estabilizador do conjunto Δ.
$H \leq G$	H subgrupo de G.
$H \trianglelefteq G$	H é um subgrupo normal de G.
Hx	Classe lateral à direita de H em G.
$Im(\varphi)$	Imagem do homomorfismo φ.
$Ker(\varphi)$	Kernel do homomorfismo φ.
Q_n	Grupo dos quatérnions de ordem n.
S_n	Grupo simétrico de n elementos.
$Sym(\Omega)$	Grupo simétrico dos elementos de Ω.

CAPÍTULO 1

Preliminares

Neste capítulo introduziremos a teoria básica que necessitamos para a compreensão do trabalho nos capítulos seguintes. A maior parte desta seção pode ser encontrada no livro guia [15].

1.1 Grupo de permutação

Os resultados expostos a continuação são encontrados em [14]. Sejam Ω um conjunto e $Sym(\Omega)$ o conjunto de permutações de Ω, isto é, o conjunto de todas as bijeções de Ω em Ω. Este conjunto é um grupo sob a composição de aplicações e chamado o **grupo simétrico** em Ω e os subgrupos dele são chamados grupos de permutações em Ω. Se $\Omega = \{1, ..., n\}$, então $Sym(\Omega) := S_n$.

Um dos resultados mais importantes desta teoria é o chamado Teorema de Cayley; todo grupo é isomorfo a um subgrupo de $Sym(\Omega)$ para um conjunto Ω adequado, além disso se o grupo é finito de ordem n, então é isomorfo a um subgrupo de S_n.

Para $x \in \Omega$ e $\pi \in Sym(\Omega)$ denota-se a imagem de x sob π como x^π, além disso uma forma de representar uma permutação π é dada por

$$\begin{pmatrix} x_1 & x_2 & ... & x_r \\ x_{i_1} & x_{i_2} & ... & x_{i_r} \end{pmatrix},$$

onde x_{i_k} é a imagem de x_i sob π, embora a notação acima seja boa, de agora em diante, vamos utilizar uma outra notação um pouco mais simples; um ciclo $\pi = (x_1, x_2, ..., x_r)$ em $Sym(\Omega)$, é uma permutação tal que ${x_i}^\pi = x_{i+1}$ para $i = 1, ..., r-1$, ${x_r}^\pi = x_1$, e $x^\pi = x$ para $x \in \Omega$ tal que $x \notin \{x_1, ..., x_r\}$, neste caso diz-se que o ciclo tem comprimento r, se $r = 2$ o ciclo é chamado transposição, e dois ciclos $(a_1, ..., a_r)$ e $(b_1, ..., b_s)$ são disjuntos se e somente se os conjuntos $\{a_1, ..., a_r\}$ e

$\{b_1, ..., b_s\}$ são disjuntos.

O grau de um grupo de permutação G é definido como a quantidade de pontos em Ω, por exemplo o grau de S_n é n. O grau de uma permutação g é o grau do grupo cíclico $< g >$.

Exemplo 1.1 *A permutação* $\sigma = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 \\ 2 & 3 & 1 & 4 & 6 & 5 \end{pmatrix}$ *do conjunto* $\Omega = \{1,2,3,4,5,6\}$ *pode ser representado como produto de um* 3*-ciclo com uma transposição da seguinte maneira* $\sigma = (1,2,3)(5,6)$*, mas também pode ser representada como* $\sigma = (2,3,1)(6,5)$*, isto quer dizer que a forma de representar uma permutação por meio de ciclos não é única. Além disso observemos que a permutação inversa de* σ*,* $\sigma^{-1} = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 \\ 3 & 1 & 2 & 4 & 6 & 5 \end{pmatrix}$ *pode ser representada como* $\sigma^{-1} = (3,2,1)(5,6)$*, isto é que pode ser obtida escrevendo os elementos dos* 3*-ciclos (e em geral para ciclos de quaisquer comprimento) na ordem inversa e as transposições ficam igual, de fato a permutação inversa de uma transposição é ela mesma.*

As demonstrações das seguintes proposições podem ser encontradas nas paginas 113 e 115 do livro [14], respectivamente.

Proposição 1.2 *Toda permutação em* S_n *é produto de ciclos disjuntos.*

Proposição 1.3 *Toda permutação em* S_n *é produto de transposições.*

Definição 1.4 *Uma permutação* $\pi \in Sym(\Omega)$ *diz-se uma permutação par se pode ser representada como um produto par de transposições, se não é permutação par é permutação impar. O conjunto das permutações pares, é chamado de grupo alternante é denotado por* $Alt(\Omega)$*.*

1.2 Homomorfismos de grupos

Definição 1.5 *Sejam* G *e* H *subgrupos. Um homomorfismo* φ *de* G *em* H *é uma aplicação* $\varphi : G \longrightarrow H$ *tal que* $\varphi(g_1 g_2) = \varphi(g_1)\varphi(g_2)$ *para todo* $g_1, g_2 \in G$*.*

Duas das propriedades importantes dos homomorfismos são: qualquer homomorfismo $\varphi : G \longrightarrow H$ envia 1_G em 1_H e para qualquer $g \in G$ cumpre-se que $\varphi(g)^{-1} = \varphi(g^{-1})$. As provas destas propriedades estão na página 70 do livro [14].

O homomorfismo é chamado de monomorfismo se for injetivo, e se for sobrejetivo de epimorfismo. Se o homomorfismo vai de G em G então é chamado de

endomorfismo, e se além disso é isomorfismo, isto é injetivo e sobrejetivo, então é um automorfismo.

Definição 1.6 *Seja $\varphi : G \longrightarrow H$ um homomorfismo de grupos. O kernel de φ é definido como os elementos de G tais que sua imagem por φ é 1_H; isto é, $Ker(\varphi) = \{g \in G | \varphi(g) = 1_H\}$.*

Entre as propriedades mais importantes dos homomorfismos encontra-se que o conjuntos das imagens $Im(\varphi) = \{\varphi(g) \in H | g \in G\}$ e o $Ker(\varphi)$, os quais são subgrupos de H e G respetivamente; além disso, o $Ker(\varphi)$ é subgrupo normal de G.

Teorema 1.7 (Primeiro teorema de isomorfismo) *Seja $\varphi : G \longrightarrow H$ um homomorfismo de grupos com kernel K. Então $G/K \simeq Im(\varphi)$. Mais precisamente, existe um isomorfismo $\hat{\varphi} : G/K \longrightarrow Im(\varphi)$ definido por $\hat{\varphi}(gK) = \varphi(g)$ para todo $g \in G$.*

1.3 Ação de grupos em conjuntos

Seja $Sym(\Omega)$ o conjunto de permutações de um conjunto Ω. Para $x \in \Omega$ e $g \in Sym(\Omega)$, denotamos o resultado da aplicação de g em x por x^g, e para a composição de permutações $h, g \in Sym(\Omega)$ aplicadas em x temos $(x^g)^h = x^{gh}$.

Definição 1.8 *Sejam G um grupo e Ω um conjunto. Uma ação de G em Ω é definida como um homomorfismo $\varphi : G \to Sym(\Omega)$.*

Assim para cada $g \in G$, $\varphi(g)$ é uma permutação de Ω, denotada por α^g, para a ação de $\varphi(g)$ sobre $\alpha \in \Omega$. Uma forma de ver a ação é

$$\begin{array}{rcl} \varphi : G & \longrightarrow & Sym(\Omega) \\ g & \longmapsto & \pi(g) : \begin{array}{ccc} \Omega & \longrightarrow & \Omega \\ \alpha & \longmapsto & \alpha^g. \end{array} \end{array} \tag{1-1}$$

A imagem $Im(\varphi)$ da ação será denotada por G^Ω.

Exemplo 1.9 *Algumas das ações importantes resultam de tomar o conjunto Ω como sendo o próprio grupo G. Uma delas é dada pela translação à direita que é chamada* ***ação coset****.*

$$\begin{array}{rcl} \varphi : G & \longrightarrow & Sym(G) \\ g & \longmapsto & \pi(g) : \begin{array}{ccc} G & \longrightarrow & G \\ \alpha & \longmapsto & \alpha * g. \end{array} \end{array}$$

O teorema de Cayley garanta que qualquer grupo pode ser representado por um grupo de permutação, essa representação é dada por uma ação de grupos. A seguir veremos como é possível fazer isto com a ação do exemplo 1.9.

Exemplo 1.10 *Seja Q_8 o grupo dos quatérnions de ordem 8, dado pelo conjunto $Q_8 := \{1, -1, i, -i, j, -j, k, -k\}$ munido das operações:*

$$\begin{array}{rcrcl} ij & = & -ji & = & k \\ jk & = & -kj & = & i \\ ki & = & -ik & = & j. \end{array}$$

Onde a ação do exemplo 1.9 fica definida da seguinte forma:

$$\begin{array}{rcccccc} \varphi: & Q_8 & \longrightarrow & Sym(Q_8) & & & \\ & i & \longmapsto & \varphi_i: & Q_8 & \longrightarrow & Q_8 \\ & & & & g & \longmapsto & gi, \\ & j & \longmapsto & \varphi_j: & Q_8 & \longrightarrow & Q_8 \\ & & & & g & \longmapsto & gj, \\ & k & \longmapsto & \varphi_k: & Q_8 & \longrightarrow & Q_8 \\ & & & & g & \longmapsto & gk. \end{array}$$

Agora vamos fazer uma bijeção dos elementos de Q_8 com o conjunto $\{1, 2, 3, 4, 5, 6, 7, 8\}$ da seguinte forma

$$\begin{array}{ccc} 1 & \longleftrightarrow & 1 \\ 2 & \longleftrightarrow & -1 \\ 3 & \longleftrightarrow & i \\ 4 & \longleftrightarrow & -i \\ 5 & \longleftrightarrow & j \\ 6 & \longleftrightarrow & -j \\ 7 & \longleftrightarrow & k \\ 8 & \longleftrightarrow & -k. \end{array}$$

Desta forma o grupo Q_8 pode ser representado pelo gerado das permutações $\varphi_i = (1,3,2,4)(5,8,6,7)$, $\varphi_j = (1,5,2,6)(3,7,4,8)$ e $\varphi_k = (1,7,2,8)(3,6,4,5)$, isto é, $Q_8 =< X >$, onde $X = [\varphi_i, \varphi_j, \varphi_k]$.

Definição 1.11 *O kernel $K = ker(\varphi)$ de uma ação é igual ao subgrupo normal*

$$\{g \in G | \alpha^g = \alpha \text{ para todo } \alpha \in \Omega\}.$$

A ação é considerada fiel se $K = 1_G$. Nesse caso pelo primeiro teorema de isomorfismo temos $G \cong Im(\varphi)$

Exemplo 1.12 *Se G for um subgrupo de $Sym(\Omega)$, podemos definir uma ação de G em Ω simplesmente colocando $\varphi(g) = g$, para $g \in G$. Esta ação é fiel.*

Definição 1.13 *Seja G agindo no conjunto Ω. Defina a relação $\sim$ sobre Ω da seguinte forma $\alpha \sim \beta$ se, e somente se, existe $g(\alpha, \beta) \in G$ tal que $\beta = \alpha^g$. Dessa forma $\sim$ é uma relação de equivalência. As classes de $\sim$ são chamadas órbitas de G sobre Ω. Em particular, a órbita de um elemento específico $\alpha \in \Omega$ fica definida $\alpha^G = \{\alpha^g | g \in G\}$.*

Definição 1.14 *Sejam G agindo no conjunto Ω e $\alpha \in \Omega$. Definimos o estabilizador de α em G, como sendo o conjunto $G_\alpha = \{g \in G | \alpha^g = \alpha\}$.*

Da definição acima podemos ver que o kernel da ação pode ser escrito como $\bigcap_{\alpha \in \Omega} G_\alpha$.

Definição 1.15 *Sejam H qualquer subgrupo de G e Ω o conjunto das classes laterias à direita de H em G, isto é, o conjunto $\{g \in G | g = Hg' \text{ para } g' \in G\}$ definimos a ação de G sobre Ω por $\alpha^g = \alpha g$.*

Teorema 1.16 (Órbita-Estabilizador) *Sejam G um grupo finito agindo no conjunto Ω e $\alpha \in \Omega$. Então*

$$|G| = |\alpha^G||G_\alpha|.$$

Demonstração. Dado $\beta \in \alpha^G$, desde que $\alpha^G = \{\alpha^g | g \in G\}$, existe $g \in G$ tal que $\beta = \alpha^g$, por outro lado se $g' \in G$ com $\alpha^{g'} = \beta$, e como $G_\alpha = \{g \in G | \alpha^g = \alpha\}$ temos que $G_{\alpha^g} = \{g' \in G | \alpha^{g'} = \alpha^g\}$, assim $g'g^{-1} \in G_\alpha$, de fato

$$\alpha^{g'g^{-1}} = \alpha^{g'}\alpha^{-1} = \alpha^g\alpha^{-1}\alpha^{gg^{-1}} = \alpha.$$

Além disso, notemos que os elementos $g' \in G$ com a propriedade $\alpha^{g'} = \beta$ são exatamente os elementos da classe lateral à direita Hg, tendo o subgrupo H como G_α, mas $|H| = |Hg|$, assim para cada $\beta \in \alpha^G$, existem precisamente $|H|$ elementos g' de G tais que $\alpha^{g'} = \beta$, logo o número total de $\beta \in \alpha^G$ é $|G|/|H|$, isto é $|\alpha^G| = |G|/|H|$, portanto $|G| = |\alpha^G||G_\alpha|$. $\square$

Definição 1.17 *Sejam G atuando sobre Ω e $\Delta \subset \Omega$. Então:*

- *O estabilizador do conjunto $\{g \in G | \alpha^g \in \Delta \text{ para todo } \alpha \in \Delta\}$ de Δ em G é denotado por G_Δ.*
- *O estabilizador pontual $\{g \in G | \alpha^g = \alpha \text{ para todo } \alpha \in \Delta\}$ de Δ em G é denotado por $G_{(\Delta)}$.*

Exemplo 1.18 *Sejam* $G =< X >$ *com* $X := \{(1,2,3),(1,2)\}$ *e* $\Delta = \{1,3\}$. *Então os estabilizadores do conjunto* G_Δ *e pontual* $G_{(\Delta)}$, *baixo a ação natural de* G *sobre o conjunto* $\Omega = \{1,2,3\}$ *são:*

$$\begin{aligned} G_\Delta &= \{(),(1,3)\} \\ G_{(\Delta)} &= \{()\}. \end{aligned}$$

1.3.1 Transitividade e primitividade

Além das órbitas e os estabilizadores, as ações de grupos em conjuntos tem outras ferramentas úteis que vamos precisar mais diante para extrair características de um grupo.

Definição 1.19 *Uma ação de* G *sobre um conjunto* Ω *é chamada transitiva se houver uma única órbita sob a ação, isto é, para qualquer* $\alpha, \beta \in \Omega$, *existe* $g \in G$ *com* $\alpha^g = \beta$. *Se para* $\{\alpha_1, ..., \alpha_n\}$ *e* $\{\beta_1, ..., \beta_n\}$ *existe* $g \in G$ *com* ${\alpha_i}^g = \beta_i$ *dizemos que a ação é n-dobra transitiva.*

Definição 1.20 *Sejam* G *agendo sobre* Ω *e* Δ *subconjunto não vazio de* Ω. *Se para todo* $g \in G$, *temos* $\Delta^g = \Delta$ *ou* $\Delta^g \cap \Delta = \varnothing$ *dizemos que* Δ *é um bloco sobre a ação. Este bloco é chamado não trivial se* $|\Delta| > 1$ *e* $\Delta \neq \Omega$. *As órbitas de qualquer ação são blocos.*

Definição 1.21 *Uma ação transitiva de* G *sobre um conjunto* Ω *é chamada primitiva se só tem blocos triviais.*

As demonstrações dos seguintes teoremas podem ser encontradas na página 16 de [27].

Teorema 1.22 *Um grupo transitivo de grau primo é primitivo.*

Teorema 1.23 *Sejam* G *transitivo sobre* Ω, U *subgrupo de* G *e* Δ *uma órbita de* U. *Se* U^Δ *é primitivo sobre* Δ *e* $|\Omega| < 2|\Delta|$, *então* G *é primitivo sobre* Ω.

Teorema 1.24 *Sejam* p *um primo e* G *um grupo primitivo de grau* $n = p + k$ *com* $k \geq 3$. *Se* G *contém um elemento de grau e ordem* p, *então* G *é ou* $Alt(\Omega)$ *ou* $Sym(\Omega)$.

Demonstração. Ver [27] página 39. □

Definição 1.25 *Se* Δ *é um bloco sob uma ação, então as diferentes traslações* Δ^g *de* Δ *formam uma partição de* Ω. *O conjunto de translações é conhecido como um sistema de blocos.*

Observação 1.26 *Da definição anterior temos que uma ação transitiva é primitiva se, e somente se, não preserva uma partição não trivial de Ω. Observemos também que $|\Delta| = |\Delta^g|$, então todos os blocos de tal partição têm o mesmo tamanho. Portanto, se G^Ω é transitivo e $|\Omega|$ é primo, então G^Ω é primitivo.*

Definição 1.27 *Se G age sobre Ω, então uma relação de equivalência $\sim$ sobre Ω é chamada uma G-congruência se $\alpha \sim \beta$ implica que $\alpha^g \sim \beta^g$ para todo α, $\beta \in \Omega$ e $g \in G$.*

Lembremos que uma relação de equivalência $\sim$ em um conjunto Ω é um subconjunto R de $\Omega \times \Omega$, em que $\alpha \sim \beta$ significa o mesmo que $(\alpha, \beta) \in R$.

Proposição 1.28 *Se G^Ω é transitiva então as G-congruências e os sistema de blocos coincidem.*

Demonstração. Seja $\{\Delta^g | g \in G\}$ um sistema de blocos de G^Ω, então a relação de equivalência $\sim$ sobre Ω é definida por $\alpha \sim \beta$ se e somente se α e β pertencem ao mesmo bloco è uma G-congruência; de fato, se $\alpha \sim \beta$ implica que $\alpha^g \sim \beta^g$ para todo $g \in G$. Por outro lado, qualquer classe de equivalência que define uma G-congruência sobre Ω forma os blocos de um sistema de blocos. $\square$

Definição 1.29 *Se $S \subseteq \Omega \times \Omega$, então a G-congruência gerada por S é definida como a interseção de todas as G-congruências que contém S.*

1.3.2 Conjugação, normalizadores e centralizadores

A **conjugação** é a ação definida sobre o mesmo grupo G dada por

$$\begin{array}{rcl} \varphi : G & \longrightarrow & Sym(G) \\ g & \longmapsto & \pi(g) : \begin{array}{rcl} G & \longrightarrow & G \\ \alpha & \longmapsto & \alpha^g := g^{-1}\alpha g. \end{array} \end{array}$$

As órbitas da ação anterior são chamadas **classes de conjugação** e os elementos da mesma classe são chamados **conjugados em** G, escrevemos $Cl_G(g)$ para notar a órbita de g, isto é, as classes de conjugação que contém g. O estabilizador G_g consiste dos elementos $f \in G$ para o qual $g^f = g$, isto é $f^{-1}gf = g$ o que é igual a $gf = fg$, é chamado **centralizador** de g em G é denotado por $C_G(g)$. O kernel da ação é chamado o **centro** de G e denotado por $Z(G)$, o qual é dado por

$$\{g \in G | \alpha g = g\alpha \text{ para todo } \alpha \in G\}.$$

Observação 1.30 *Aplicando o Teorema da* ***Órbita-Estabilizador*** *teríamos que para todo* $g \in G$

$$Cl_G(g) = \frac{|G|}{C_G(g)}.$$

Agora vamos tomar Ω como sendo o conjunto dos subgrupos H de G e a ação dada por

$$\begin{array}{rcl} \varphi : G & \longrightarrow & Sym(\Omega) \\ g & \longmapsto & \pi(g) : \begin{array}{rcl} \Omega & \longrightarrow & \Omega \\ \alpha & \longmapsto & H^g := g^{-1}Hg. \end{array} \end{array}$$

Novamente, subgrupos na mesma órbita desta ação são chamados **subgrupos conjugados**. O estabilizador do subgrupo H que é igual a $\{g \in G | g^{-1}Hg = H\}$ é chamado **normalizador** de H em G e denotado por $N_G(H)$.

Observação 1.31 *Note que* $H \trianglelefteq N_G(H)$ *e pelo teorema de* ***Órbita-Estabilizador*** *temos que o número de conjugados de* H *em* G *é igual a* $|G : N_G(H)|$.

1.4 Apresentação de grupos

Grupos livres

Definição 1.32 *Um grupo* F *é livre sobre o subconjunto* X *de* F *se, para qualquer grupo* G *e qualquer aplicação* $\theta : X \longrightarrow G$*, existe um único homomorfismo de grupos* $\theta' : F \longrightarrow G$ *com* $\theta'(x) = \theta(x)$ *para todo* $x \in X$*. Isso significa que o diagrama*

$$\begin{tikzcd} X \arrow[r, hook, "i"] \arrow[d, dashed, "\theta"'] & F \arrow[dl, "\exists!\theta'"] \\ G & \end{tikzcd}$$

é comutativo.

A grosso modo podemos dizer que um grupo livre sobre um conjunto X é o maior grupo gerado por X.

Proposição 1.33 *Dois grupos livres finitos sobre o mesmo conjuntos são isomorfos, e mais geralmente, dois grupos livres finitos sobre* X_1 *e* X_2 *são isomorfos se e somente se* $|X_1| = |X_2|$.

Demonstração. Vamos mostrar a parte geral da proposição, pois a primeira parte é consequência imediata desta. Primeiro suponhamos que os dois conjuntos tem a

mesma quantidade de elementos e que F_1 e F_2 são grupos livres sobre X_1 e X_2, respetivamente, definamos as aplicações inclusão $i_1 : X_1 \longrightarrow F_1$, $i_2 : X_2 \longrightarrow F_2$ e a aplicação bijetiva $\tau : X_1 \longrightarrow X_2$, assim, $i_2 \circ \tau : X_1 \longrightarrow F_2$ e $i_1 \circ \tau^{-1} : X_2 \longrightarrow F_1$, logo fazendo a restrição de $i_2 \circ \tau$ e $i_1 \circ \tau^{-1}$ a X_1 e X_2 respetivamente temos os homomorfismos de grupos

$$\theta_1 := i_2 \circ \tau : F_1 \longrightarrow F_2 \qquad \theta_2 := i_1 \circ \tau^{-1} : F_2 \longrightarrow F_1,$$

portanto $\theta_2 \circ \theta_1 : F_1 \longrightarrow F_1$ restringe a identidade sobre X_1, e pela unicidade na definição $\theta_2 \circ \theta_1$ é a identidade em F_1, e da mesma forma $\theta_1 \circ \theta_2$ é a identidade em F_2, logo elas são mutuamente inversas.

Reciprocamente suponha que F_1 e F_2 são livres sobre X_1 e X_2, respetivamente, com F_1 e F_2 isomorfos. Seja G um grupo de ordem 2. Então existem $2^{|X_i|}$ aplicações distintas de X_i a G para $i = 1, 2$, e assim por definição existem também $2^{|X_i|}$ homomorfismos de F_i a G, então para X_1 e X_2 finitos, como F_1 e F_2 são isomorfos, implica que $|X_1| = |X_2|$.

□

Observação 1.34 *O resultado anterior é valido também para grupos livres infinitos, para mais detalhes veja [15], no entanto nesta dissertação vamos trabalhar somente com grupos livres finitos.*

A cardinalidade $|X|$ de X é conhecida como o grau de um grupo livre F sobre X. Agora vamos fazer a construção de um grupo livre sobre um conjunto X e assim provar a existência do grupo livre.

Para qualquer conjunto X, definamos o conjunto $X^{-1} := \{(x, -1) | x \in X\}$, mas vamos escrever x^{-1} em vez de $(x, -1)$. Para $y = x^{-1} \in X^{-1}$, definamos $y^{-1} = x$ e o conjunto A_X como $X \cup X^{-1}$.

Definição 1.35 *Para um conjunto A (que pelo geral é o conjunto $A = A_X$), definamos A^* como o conjunto de todas as cadeias $x_1 x_2 ... x_r$ com cada $x_i \in A$. O número r é o comprimento da cadeia, e para $\sigma \in A^*$, denotamos o comprimento de σ como $|\sigma|$. A cadeia vazia sobre A de comprimento 0 é denotado por ϵ_A, ou somente ϵ.*

Vamos definir os elementos de A^* como **palavras** sobre A e o elemento ϵ denotará a palavra vazia. Uma sub-palavra de $w = x_1 ... x_r \in A^*$ é a palavra vazia,

ou qualquer palavra $x_i x_{i+1} ... x_j$, com $1 \leq i \leq j \leq r$. É chamado de prefixo de w se ele está vazio ou $i = 1$, e é chamado de sufixo de w se ele está vazio ou $j = r$. Vamos fixar nosso conjunto X e definir $A = A_X$.

Dizemos que duas palavras v e w em A são diretamente equivalentes se uma pode ser obtida da outra por interseção ou omissão de uma sub-palavra xx^{-1}, onde $x \in A$. Seja $\sim$ a relação de equivalência sobre A^* gerada pela equivalência direita, isto é, para $v, w \in A^*$, $v \sim w$ se e somente se existe uma sequência $v = v_0, v_1, ..., v_r = w$ de elementos de A^*, com $r \geq 0$, tais que v_i e v_{i+1} são diretamente equivalentes para $0 \leq i \leq r$. Denotamos as classes de equivalência de $w \in A^*$ sob $\sim$ por $[w]$. Seja F_X o conjunto de classes de equivalência de $\sim$.

Para definir uma operação em F_X, vemos que se $u_1 \sim v_1$ e $u_2 \sim v_2$, então $u_1 u_2 \sim v_1 v_2$, assim a operação definida como $[u_1][u_2] = [u_1 u_2]$ em F_X está bem definida, além disso é associativa, tem como elemento identidade o elemento $[\epsilon]$ e se operamos os elementos $[x_1 x_2 ... x_r]$ e $[{x_1}^{-1} {x_2}^{-1} ... {x_r}^{-1}]$ obtemos o elemento identidade $[\epsilon]$, portanto F_X é um grupo sob a multiplicação assim definida.

Teorema 1.36 *Para qualquer conjunto X, F_X é um grupo livre sobre o conjunto $[X] := \{[x] | x \in X\}$, e a aplicação $x \longmapsto [x]$ define uma bijeção de X em $[X]$.*

Demonstração.

Para provar que F_X é livre sob $[X]$, temos que mostrar que para qualquer grupo G e qualquer homomorfismo $\phi : [X] \longrightarrow G$ existe um único homomorfismo de grupos $\phi' : F_X \longrightarrow G$ tal que $\phi'([x]) = \phi([x])$, para todo $[x] \in [X]$. Seja $F = F_X$ e $A = A_X$. Sejam G um grupo e $\theta : X \longrightarrow G$ uma aplicação que vamos estender a $\theta' : A^* \longrightarrow G$ da seguinte maneira: Seja $w := {x_1}^{\epsilon_1} x_2^{\epsilon_2} ... {x_r}^{\epsilon_r}$ onde cada $x_i \in X$ e $\epsilon_i = \pm 1$, e seja $g_i = \theta(x_i)$ para $1 \leq i \leq r$. Então definimos

$$\begin{array}{rccc} \theta' : & A^* & \longrightarrow & G \\ & \epsilon & \longmapsto & 1_G \\ & w & \longmapsto & {g_1}^{\epsilon_1} {g_2}^{\epsilon_2} ... {g_r}^{\epsilon_r}. \end{array}$$

Além disso, como $v \sim w$ implica que $\theta'(v) = \theta'(w)$, temos que θ' induz uma aplicação bem definida $\phi' : F_X \longrightarrow G$ que é um homomorfismo de grupos. Assim para uma aplicação $\phi : [X] \longrightarrow G$ dada, definimos a aplicação

$$\begin{array}{rccc} \theta : & X & \longrightarrow & G \\ & x & \longmapsto & \phi([x]). \end{array} \tag{1-2}$$

Assim a aplicação ϕ', definida em (1-2), é um homomorfismo estendido de ϕ, que é único.

Agora veremos que a aplicação $x \longmapsto [x]$ define uma bijeção de X a $[X]$, seja G qualquer grupo tal que $|G| \geq |X|$, e seja $\theta : X \to G$ uma injeção. Então, como $\phi'([x]) = \theta(x)$ para $x \in X$, temos que $x \longmapsto [x]$ define uma bijeção

$$\begin{array}{rccc} \theta' : & A^* & \longrightarrow & G \\ & w & \longmapsto & \phi([x_1])^{\epsilon_2}\phi([x_2])^{\epsilon_2}...\phi([x_r])^{\epsilon_r}. \end{array}$$

□

De agora em diante vamos escrever w em vez de $[w]$ para os elementos $F := F_X$, e denotaremos $w_1 =_F w_2$ quando $[w_1] = [w_2]$, mas $w_1 = w_2$ denotarão duas palavras iguais, assim pelo Teorema 1.36 denotaremos o F_X como o grupo livre sobre X.

Uma palavra em A^* é chamada reduzida, se não contém xx^{-1} como subcadeias para qualquer $x \in A$.

Proposição 1.37 *Se X é qualquer conjunto, então cada classe de equivalência de F_X contém exatamente uma palavra reduzida.*

Demonstração. Veja [15] pagina 36. □

Apresentação de grupos

Definição 1.38 *Seja A um subconjunto de um grupo G. O fecho normal de A em G, que é denotado por $\left\langle A^G \right\rangle$, é definida como a interseção de todos os subgrupos normais de G que contém A, neste caso $A^G := \{g^{-1}ag | g \in G, a \in A\}$.*

Definição 1.39 *Sejam X um conjunto, $A = A_x$ definida como na seção anterior e R um subconjunto de A^*. Definimos a apresentação de grupo $\langle X|R\rangle$ como o grupo quociente F/N, onde $F = F_X$ é o grupo livre sobre X, e N é o fecho normal $\left\langle R^F \right\rangle$ de R em F.*

Intuitivamente, a definição acima nos fala que $\langle X|R\rangle$ é o maior grupo G gerado por X que em que todas as cadeias $v \in R$ representam o elemento identidade.

Exemplo 1.40 *Um exemplo simples pode ser o grupo cíclico de ordem n, ele é apresentado como $G =< a|a^n = 1_G >=< a|a^n >$, onde 1_G é o elemento identidade do grupo.*

Na página 52 do livro [20], encontra-se o seguinte teorema como exemplo de uma apresentação para o grupo simétrico S_n.

Teorema 1.41 *Se $n > 1$, existe uma apresentação do grupo simétrico S_n com geradores $x_1, x_2, ..., x_{n-1}$ e relações*

$$1 = x_i^2 = (x_j x_{j+1})^3 = (x_k x_l)^2,$$

onde $1 \leq i \leq n-1$, $1 \leq j \leq n-1$ e $1 \leq l < k-1 < n-1$.

O livro guia [15] apresenta o seguinte teorema muito importante para saber se uma aplicação de grupos define um homomorfismo de grupos.

Teorema 1.42 *Sejam $\langle X|R\rangle$ uma apresentação de um grupo G e $\theta : X \longrightarrow H$ uma aplicação. Estenda θ à $\theta : X^{-1} \longrightarrow H$ fazendo $\theta(x^{-1}) = \theta(x)^{-1}$ para todo $x \in X$, então θ estende ao homomorfismo $\theta' : G \longrightarrow H$ se e só se $\theta(x_1) \cdots \theta(x_r) = 1_H$ para todo $w = x_1 ... x_r \in R$. Esta extensão é única se existir.*

Apresentação de subgrupos e teorema de Nielsen-Schrerier

Definição 1.43 *Seja H um subgrupo de um grupo G. O subconjunto S de G é chamado transversal à direita de H em G se S intercepta cada classe lateral à direita de H em exatamente um elemento. Mais precisamente, uma transversal à direita é um sistema de representantes em G das classes laterais à direita de H em G.*

As seguintes condições serão usadas para os resultados desta seção. Sejam G o grupo gerado pelo conjunto X e $A = X \cup X^{-1}$. Pela definição 1.39 tem-se que $G \cong F/N$, para algum N, onde $F = F_X$ é o grupo livre sobre X. Vamos supor que $G = F/N$ e seja $H = E/N$ um subgrupo de G, observe que E é subgrupo de F. Seja T o conjunto das palavras reduzidas de A^* que formam uma transversal à direita de E em F, vamos assumir que T contém a palavra reduzida ϵ como o representante da classe lateral à direita de E, logo as imagens de T em G formam uma transversal à direita de H em G.

Definamos a aplicação

$$\begin{array}{rcl} A^* & \longrightarrow & T \\ w & \longmapsto & \bar{w} := T \cap Ew. \end{array} \tag{1-3}$$

Teorema 1.44 (Schreier) *Com a notação de acima, o subgrupo E de F é gerado pelo subconjunto $\hat{Y}$ de E, com*

$$\hat{Y} := \{tx\overline{tx}^{-1} | t \in T, x \in X, tx \neq_F \overline{tx}\}.$$

Portanto o subgrupo H de G é gerado pelas imagens de $\hat{Y}$ em G.

Para a prova do teorema vamos fazer uso dos seguintes fatos:

Lema 1.45 *Para cada $w \in A^*$ a aplicação (1-3) satisfaz as seguintes propriedades:*

i. $Ew = E\bar{w}$;

ii. $\bar{\bar{w}} = \bar{w}$;

iii. $w \in T$ *se, e só se,* $w = \bar{w}$.

Além disso, para $t \in T$ e $x \in A$ temos $\overline{\overline{tx}x^{-1}} = t$.

Demonstração. A parte $i.$ segue de observar o elemento $\bar{w}$ como um representante da classe Ew. A prova da parte $ii.$, vem de aplicar a definição da aplicação (1-3) e o item $i.$, isto é, $\bar{\bar{w}} = T \cap E\bar{w} = T \cap Ew = \bar{w}$. Para a parte $iii.$ observe que $w \in T$ se, e somente se $w \in T \cap Ew$ e isto é se, somente se $w = \bar{w}$. Por outro lado, para $t \in T$ e $x \in A$, pelo item $i.$ temos que $Etx = E\overline{tx}$, isto é $Et = Etxx^{-1} = E\overline{tx}x^{-1}$, portanto como $t \in T$ temos que $t = \bar{t} = T \cap Et = T \cap E\overline{tx}x^{-1} = \overline{\overline{tx}x^{-1}}$

□

Lema 1.46 *Para $t \in T$ e $x \in A$ temos que $tx^{-1}\overline{tx^{-1}}^{-1} \in \hat{Y} \cap \hat{Y}^{-1}$.*

Demonstração. É claro que $tx^{-1}\overline{tx^{-1}}^{-1} \in \hat{Y}$, só resta mostrar que também pertence a $\hat{Y}^{-1}$, observemos que para $u = \overline{tx^{-1}}$, $u \in T$, logo aplicando o lema 1.45 temos que $t = \overline{ux}$, assim $tx^{-1}\overline{tx^{-1}}^{-1} = tx^{-1}u^{-1} = (uxt^{-1})^{-1} = (ux\overline{ux}^{-1})^{-1} \in \hat{Y}^{-1}$.

□

Demonstração.[Teorema 1.44] Primeiro provaremos que o subgrupo E de F é gerado por $\hat{Y}$. Seja $w = x_1x_2...x_r \in A^*$ e para $0 \leq i \leq r$ seja $t_i := \overline{x_1x_2...x_r}$ onde $t_0 = \epsilon$, observemos que o elemento w pode ser visto como

$$w =_F (t_0x_1t_1^{-1})(t_1x_2t_2^{-1})...(t_{r-1}x_rt_r^{-1})t_r. \tag{1-4}$$

Observe que $t_i = \overline{t_{i-1}x_i}$, para $0 \leq i \leq r$, logo pelo Lema 1.46 cada elemento em parêntese $t_ix_{i+1}t_{i=1}{}^{-1}$ da equação (1-4) é igual a 1_E ou pertence a $\hat{Y} \cap \hat{Y}^{-1}$, assim

provamos que cada elemento w de E está contido no gerado de $\hat{Y}$, e como $\hat{Y}$ é subconjunto de E, temos que E é gerado por $\hat{Y}$. Por outro lado, se w representa um elemento de H , então $t_r = \bar{w} = \epsilon$, assim o subgrupo H de G é gerado pelas imagens de $\hat{Y}$ em G.

□

Observação 1.47 *O conjunto $\hat{Y}$ é chamado o conjunto dos geradores de Schreier de E com respeito ao grupo F.*

Seja Y o conjunto dos novos símbolos correspondentes a $\hat{Y}$, onde denotamos o elemento de Y correspondente a $tx\overline{tx}^{-1} \in \hat{Y}$ por y_{tx}. Definamos a aplicação

$$\begin{array}{rcl} \phi : Y & \longrightarrow & F \\ y_{tx} & \longmapsto & tx\overline{tx}^{-1}. \end{array} \tag{1-5}$$

A aplicação (1-5) pode ser estendida a um homomorfismo (que denotamos igual) do grupo livre F_Y de Y a F, assim o teorema 1.44 garante que $Im(\phi) = E$. Para $t \in T$, $x \in X$, denotamos o elemento de Y^{-1} que aplica ϕ a $tx^{-1}\overline{tx^{-1}}^{-1}$ por $y_{tx^{-1}}$. Definimos a aplicação $\rho : A^* \longrightarrow F_Y$ por $\rho(w) = y_{t_o x_1} y_{t_1 x_2} ... y_{t_{r-1} x_r}$, onde $w = x_1 x_2 ... x_r$ com $w =_F (t_0 x_1 {t_1}^{-1})(t_1 x_2 {t_2}^{-1})...(t_{r-1} x_r {t_r}^{-1}) t_r$, observemos que ϕ e ρ independem da escolha da T, o fato segue-se da definição de grupo livre, além disso, temos que $\rho(uxx^{-1}v) =_F \rho(uv)$, para qualquer $u, v \in A^*$ e $x \in A$, assim $xx^{-1} =_F 1_F$.

Pelo argumento anterior temos que ρ induz uma aplicação bem definida denotada por $\rho : F \longrightarrow F_Y$ que chamamos a aplicação reescrita.

$$\begin{tikzcd} Y \arrow[r, "\phi"] \arrow[d, "i"'] & F \\ F_Y \arrow[ur, "\phi"'] & \end{tikzcd} \qquad \begin{tikzcd} A^* \arrow[r, "\rho"] \arrow[d, "i"'] & F_Y \\ F \arrow[ur, "\rho"'] & \end{tikzcd}$$

Mais geralmente, para $t \in T$ e $w = x_1 x_2 ... x_r \in A^*$, definimos $\rho(t, w)$ da mesma forma que $\rho(w)$, mas com $t_0 = t$, isto é, $\rho(t, w) := y_{t_0 x_1} y_{t_1 x_2} ... y_{t_{r-1} x_r}$ onde $t_i = \overline{t x_1 x_2 ... x_i}$ para $0 \leq i \leq r$ e $\rho(w) = \rho(\epsilon, w)$.

Lema 1.48 *i. $\rho(u, v) = \rho(u)\rho(\overline{u}, v)$, para todo $u, v \in A^*$;*

ii. $\rho(u)^{-1} =_{F_Y} \rho(\overline{u}, u^{-1})$, para todo $u \in A^*$.

Demonstração.

i. Sejam $u = u_1u_2...u_r$ e $v = v_1v_2...v_s$, podemos fazer

$$u =_F (t_0u_1{t_1}^{-1})(t_1u_2{t_2}^{-1})...(t_{r-1}u_r{t_r}^{-1})t_r$$

com $t_0 = t$ e $t_i = \overline{tu_1...u_i}$, e $v =_F (t'_0v_1{t'_1}^{-1})(t'_1v_2{t'_2}^{-1})...(t'_{r-1}v_r{t'_r}^{-1})t'_r$ com $t'_0 = t_r$ e $t'_i = \overline{t_rv_1...v_r}$, assim temos

$$\begin{aligned}\rho(uv) &= \rho(t, uv)\\ &= (y_{t_0u_1}y_{t_1u_2}...y_{t_{r-1}u_r})(y_{t'_0v_1}y_{t'_1v_2}...y_{t'_{s-1}v_s})\\ &= \rho(t,u)\rho(\overline{u}, v)\\ &= \rho(u)\rho(\overline{u}, v).\end{aligned}$$

ii. Observe que $uu^{-1} =_F 1_F$, assim $\rho(uu^{-1}) =_{F_Y} 1_{F_Y}$, então $1_{F_Y} = \rho(uu^{-1}) = \rho(u)\rho(\overline{u}, u^{-1})$, portanto $\rho(u)^{-1} =_{F_Y} \rho(\overline{u}, u^{-1})$.

□

Observação 1.49 *Observe que para $u \in E$ temos que $\overline{u} = \epsilon$ e assim a parte i do lema 1.48 garante que ρ restringido a E é um homomorfismo, portanto, $\rho(u, v) = \rho(u)\rho(\overline{u}, v) = \rho(u)\rho(v)$.*

A transversal T é chamada Transversal de Schreier se é prefixo-fechado, isto é, se todos os prefixos de todos os elementos de T estão em T. Para a prova da existência dos Transversais Schreier precisamos das seguintes definições.

Definição 1.50 *Uma ordem parcial sobre um conjunto A é uma ralação binaria $<$ sobre os elementos de A que é irreflexiva e transitiva, se além disso cumpre que para todo $x, y \in A, x < y$ ou $x = y$ ou $y < x$, diz-se que a ordem é total e uma boa ordenação se cada subconjunto não vazio contém um elemento mínimo.*

Definição 1.51 *Seja $<$ definindo uma boa ordenação do conjunto A. Então a ordenação lexicográfica (dicionario) $<_L$, é definida por $a_1a_2...a_m < b_1b_2...b_n$, com $a_i, b_j \in A$ se, para algum $k \geq 0$, $a_i = b_i$ para $1 \leq i \leq k$ e ou $k = m < n$ ou $a_{k+1} = b_{k+1}$.*

Definição 1.52 *Seja $<$ uma boa ordenação do conjunto A. Então, a ordenação Shortlex $<_S$ de A^* é definida por $u <_S v$ se $|u| < |v|$ ou se $|u| = |v|$ e $u <_L v$.*

Seja $<$ uma boa ordenação de A^* com a propriedade de que: se $u < v$, então $uw < vw$ para todo $u, v, w \in A^*$ e escolhemos os elementos de T para que sejam os elementos mínimos de suas classes laterais sob esta ordenação, exemplos de ordenações para A com esta propriedade são ordenações Shortlex antes definidos.

Teorema 1.53 (Nielsen-Schreier) *Se T é uma transversal de Schreier, então o subgrupo E de F é um grupo livre sobre o conjunto gerador $\hat{Y}$.*

Demonstração.

Seja Y o conjunto gerador de $\hat{Y}$. Para F_Y o grupo livre sobre Y o seguinte diagrama comuta para quaisquer grupo G e homomorfismo φ.

$$\begin{tikzcd} Y \arrow[r, "\varphi"] \arrow[d, "i"'] & G \\ F_Y \arrow[ur, "\varphi"'] & \end{tikzcd}$$

Como E é subgrupo de F, é suficiente provar que a aplicação (1-5) é um monomorfismo.

Se $w = t \in T$, então cada elemento em parêntese da equação (1-4) são iguais a 1_E, e assim $\rho(t) = \epsilon$. Por outro lado, se $w = tx$ com $t \in T$, $x \in X$ e $tx \neq_F \overline{tx}$, então cada elemento em parêntese é trivial exceto para o mínimo, assim $\rho(tx) = y_{tx}$ e, usando o lema 1.48, temos

$$\begin{aligned} \rho(tx\overline{tx}^{-1}) &= \rho(tx)\rho(\overline{tx}, \overline{tx}^{-1}) \\ &= y_{tx}\rho(\overline{tx}, \overline{tx}^{-1}) \\ &=_F y_{tx}\rho(\overline{tx})^{-1} \\ &= y_{tx}\epsilon \\ &= y_{tx}. \end{aligned}$$

Então a composição $\rho \circ \phi$ é igual ao homomorfismo identidade sobre F_Y e isto implica que ϕ é um monomorfismo.

□

1.5 Bases e conjuntos geradores fortes - BSGS

Em 1970 Sims introduziu o conceito de base para fazer cálculos com grupos de permutação, o artigo tem como nome "Computational methods in the study of permutation groups" que pode ser encontrado desde a página 169 em [22]. O Sims afirma que similarmente a uma base em espaços vetoriais os elementos de grupo têm uma representação única relativa a ele.

Nesta seção vamos assumir G sendo um grupo de permutação finita agindo

sobre $\Omega = \{1,..,n\}$ e gerado por uma sequência finita S de elementos de $Sym(n)$. Seja $B = [\beta_1, ..., \beta_k]$ uma sequência de elementos distintos em Ω. Para $1 \leq i \leq k+1$, Definamos:

1. $G^{(i)} := G_{\beta_1,...,\beta_{i-1}}$, isto é, o subgrupo de G de elementos que deixam fixos os $\beta_1, ..., \beta_{i-1}$. Observe que $G^1 = G$.
2. $S^{(i)} := S \cap G^{(i)}$.
3. $H^{(i)} :=< S^{(i)} >$.
4. $\Delta^{(i)} := \beta_i^{H^{(i)}}$.

Seja $U^{(i)}$ um transversal à direita do estabilizador $H^{(i)}_{\beta_i}$ em $H^{(i)}$, então pelo Teorema da Órbita-estabilizador 1.16 temos que

$$U^{(i)} = \{u^i_\beta \in H^{(i)} | \beta \in \Delta(i), \beta_i^{u^i_\beta} = \beta\}.$$

A sequência $U^{(i)}$ é chamada uma cadeia estabilizadora. A lista $B = [\beta_1, ..., \beta_k]$ é dita uma base para G se o único elemento em G que fixa cada um dos $\beta_1, ..., \beta_k$ é a identidade, isto é, se $G^{(k+1)} = 1_G$ e portanto

$$1_G \leq G^{(k+1)} \leq G^{(k)} \leq ... \leq G^{(2)} \leq G^{(1)} = G. \tag{1-6}$$

A sequência S é chamada de conjunto gerador forte de G relativo à base B se inclui geradores para cada estabilizador $G^{(i)}$ na cadeia (1-6); isto é, se para $i = 1, ..., k+1$ temos que $G^{(i)} =< S^{(i)} >= H^{(i)}$, note que por definição isto sempre é verdadeiro para $i = 1$. Dizemos que a dupla (B, S) é um BSGS para G se B é uma base e S um conjunto gerador forte relativo a B. Neste caso, $G^{(i)}$ é chamado o i-ésimo estabilizador básico e $\Delta^{(i)} = \beta_i^{G^{(i)}}$ é a i-ésima órbita básica.

Se (B, S) é um $BSGS$ para o grupo G, então a sequência de transversais $U^{(1)}, ..., U^{(k)}$ nos da uma forma normal conveniente para os elementos de G, onde cada elemento $g \in G$ tem uma única representação $g = u_k u_{k-1} ... u_1$ com $u_i \in U^{(i)}$. Note que a ordem do grupo pode ser obtida a partir das transversais, de fato, $|G| = |U^{(k)}||U^{(k-1)}|...|U^{(1)}|$. Se $B = [\beta_1, ..., \beta_k]$ é uma base e $g \in G$, então $B^g = [\beta_1^g, ..., \beta_k^g]$ é chamada a imagem base de g relativa a B. O seguinte exemplo é uma generalização do exemplo 2.4 do trabalho de tésis de Thomas Rehn, (Ver [19]).

Exemplo 1.54 *Uma base para o grupo simétrico S_n é $B = [1, 2, ..., n-1]$ e um conjunto gerador para ele é $S = \{(1,2), (2,3), ..., (n-1, n)\}$. De fato, uma sequência com menos de $n-1$ pontos não pode ser uma base, pois suponhamos que a base fosse*

$B = [1,2,...,n-2]$ então a permutação $(n-1,n)$ estabiliza todos os elementos de B. Assim, a cadeia estabilizadora para S_n é feita assim:

$$\begin{aligned}
G^{(1)} &= G = S_n \\
G^{(2)} &= G_1 =< [(2,3),...,(n-1,n)] >\simeq S_{n-1} \\
G^{(3)} &= G_{1,2} =< [(3,4),...,(n-1,n)] >\simeq S_{n-2} \\
&\vdots \\
G^{(n-1)} &= G_{1,2,...,n-2} =< [(n-1,n)] >\simeq S_2 \\
G^{(n)} &= G_{1,2,...,n-1} =< [()] > .
\end{aligned}$$

Neste caso podemos ver que $S^{(i)} = S \cap G^{(i)} = \{(1,2),(2,3),...,(i-2,i-1)\}$ e assim $H^{(i)} = G^{(i)} =< S_{i-1} >$. Portanto a dupla (B,S) é um BSGS para S_n.

Se tomamos a sequência geradora para S_n sendo $T = \{(1,...,n),(n-1,n)\}$ e a mesma base $B = [n,...,2]$. Para cada $t \in T$ temos que $\beta_1^t = n^t \neq n = \beta_1$, assim $S^{(2)} = T \cap G^{(2)} = \emptyset$ e portanto $H^{(2)} =< S^{(2)} >=< () >$ que claramente não é igual a $G^{(2)} =< [(1,2),(2,3),...,(n-2,n-1)] >$. De onde (B,T) não é um BSGS para S_n.

Lema 1.55 *A imagem base de g relativa à base B de G é unicamente determinada pelo elemento $g \in G$.*

Demonstração. Suponhamos que $B^g = B^h$ para $g,h \in G$, então $(B^g)^{h^{-1}} = (B^h)^{h^{-1}}$, assim $B^{gh^{-1}} = B$ e portanto $gh^{-1} = 1_G$, pois o único elemento que fixa cada u dos $\beta_1,...,\beta_k$ é a identidade, logo $g = h$. □

1.6 Complexidade de um algoritmo

Iniciaremos com o que entendemos por algoritmo; na linguagem da computação e da matemática entendemos como algoritmo uma sequência de instruções necessárias para a resolução de um problema. Um problema pode ser resolvido de diferentes maneiras e assim através de diversos algoritmos, então: Como escolher entre um algoritmo e outro?. Para escolher o melhor algoritmo que solucione o problema temos que ter em conta se queremos menor o tempo de processamento ou menor a quantidade de memoria da máquina, neste trabalho vamos optar pelo tempo de processamento e assim nesta seção vamos ver os conceitos principais da complexidade de tempo que é, também chamado, esforço requerido ou quantidade de trabalho, sempre referindo-se ao tempo neste caso.

Para começar vamos dar a definição de uma máquina de Turing. A máquina de Turing é proposta pelo matemático inglês Alan M. Turing em 1936. Basicamente é um modelo abstrato de um computador, que se restringe apenas ao seu funcionamento e não à sua implementação física A seguintes definições podem ser encontradas em [26].

Definição 1.56 *Uma Máquina de Turing é uma 7-upla* $(Q, \Sigma, \Gamma, \delta, q_0, q_{accept}, q_{reject})$*, onde* Q, Σ, Γ *são conjuntos finitos e*

1. Q *é o conjunto de estados,*
2. Σ *É o alfabeto de entrada que não contém o símbolo em branco* $\sqcup$*,*
3. Γ *é o alfabeto da fita, onde* $\sqcup \in \Gamma$ *e* $\Sigma \subset \Gamma$*,*
4. $\delta : Q \times \Gamma \longrightarrow Q \times \Gamma \times \{L, R\}$ *é a função transição,*
5. $q_0 \in Q$ *é o estado inicial,*
6. q_{accept} *é o estado de aceitação,*
7. $q_{reject} \in Q$ *é o estado de rejeição onde* $q_{reject} \neq q_{accept}$*.*

Vamos pensar uma Máquina de Turing como um dispositivo que pode adotar um estado de Q que está ligado a uma cabeça de leitura/escritura onde pode ler e escrever símbolos de uma fita infinita. Em cada movimento a máquina lê o símbolo da fita na posição onde a cabeça de leitura está e de acordo ao símbolo encontrado e ao estado que ela encontra-se vai fazer três coisas: Dependendo da função de transição passa a um novo estado, imprime um novo símbolo na fita na mesma posição onde acaba de ler o símbolo atual e no final move a cabeça de leitura/escritura uma nova posição à esquerda (L) ou à direita (R).

Exemplo 1.57 *Uma Maquina de Turing utiliza um sistema de codificação, neste exemplo vamos fazer uso do sistema unário para os números naturais. Vamos representar cada número natural* x *por uma palavra* $\omega(x) \in \{1, 1, 1, ...\}$ *tal que* $|\omega(x)| - 1 = x$*, por exemplo* $1 \mapsto \omega(1) = 11$*,* $2 \mapsto \omega(2) = 111$ *e assim por adiante. No começo e ao final a cabeça encontra-se no primeiro* 1 *à esquerda e não tem transições definidas para estados finais.*

Seja M *a máquina de Turing dada por*

$$M = \{\{q_0, q_1, q_2, q_3, q_4, q_5\}, \{1\}, \{1, \sqcup\}, \delta, q_0, \{q_5\}\},$$

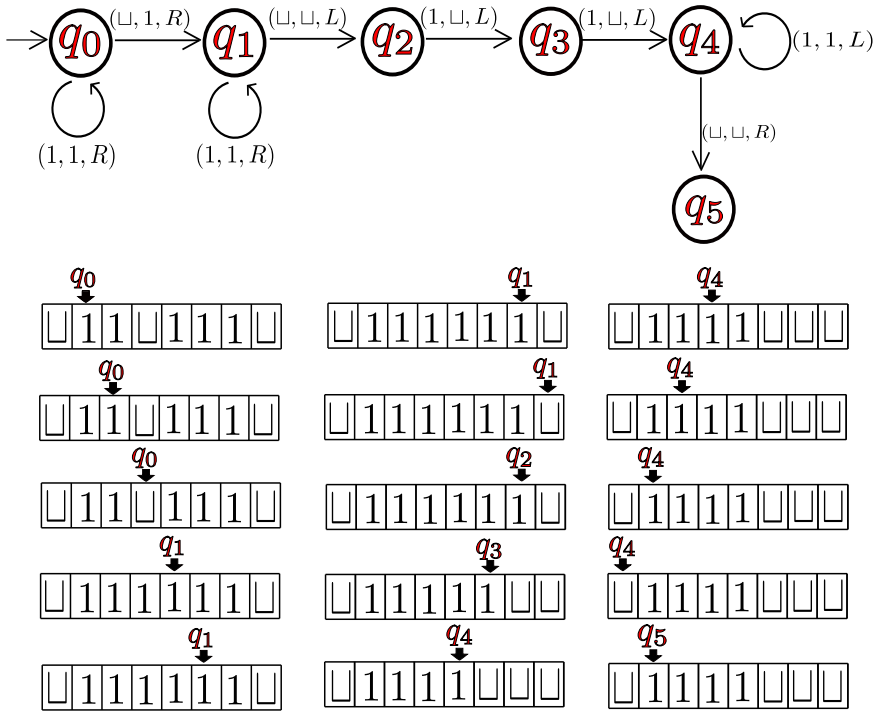


Figura 1.1: *Calcula a soma de dois inteiros positivos.*

onde a função de transição δ é dada por

$$\begin{array}{lll} \delta(q_0,1) = (q_0,1,R) & \delta(q_0,\sqcup) = (q_1,1,R) & \delta(q_1,1) = (q_1,1,R) \\ \delta(q_1,\sqcup) = (q_2,\sqcup,L) & \delta(q_2,1) = (q_3,\sqcup,L) & \delta(q_3,1) = (q_4,\sqcup,L) \\ \delta(q_4,1) = (q_4,1,L) & \delta(q_4,\sqcup) = (q_5,0,R). & \end{array}$$

Na figura 1.1 encontra-se primeiro o desenho da função de transição e depois um exemplo de como M faz a soma dos números 1 e 2.

Definição 1.58 *Seja M uma Máquina de Turing determinista que para em todas as entradas. O tempo de execução ou a complexidade do tempo de M é a função $f : \mathbb{N} \longrightarrow \mathbb{N}$, onde $f(n)$ é o número máximo de passos que M usa em qualquer entrada de comprimento n. Se $f(n)$ é o tempo de execução de M, dizemos que M é executado no tempo $f(n)$ e que M é uma Máquina de Turing do tempo $f(n)$. Usualmente usamos n para representar o comprimento da entrada.*

O livro [26] faz a introdução e definição da notação Big-O da seguinte maneira: *Como o tempo exato de execução de um algoritmo muitas vezes é uma expressão complexa, geralmente nós apenas estimamos isso. Em uma forma conveniente de estimativa, chamada de análise assintótica, procuramos entender o tempo de execução do algoritmo quando é executado em grandes entradas. Fazemos isso considerando apenas o termo de ordem mais alta da expressão para o tempo de execução do algoritmo, desconsiderando tanto o coeficiente desse termo quanto os termos de*

ordem inferior, porque o termo de ordem mais elevado domina os outros termos em grandes entradas. Por exemplo, a função $f(n) = 6n^3 + 2n^2 + 20n + 45$ *tem quatro termos e o termo de ordem mais alta é* $6n^3$*. Desconsiderando o coeficiente 6, dizemos que* f *é assintoticamente no máximo* n^3*. A notação assintótica ou a notação Big-O para descrever esta relação é* $f(n) = O(n^3)$[1]. Para a seguinte definição deixe $\mathbb{R}^+$ ser o conjunto de números reais não negativos.

Definição 1.59 *Sejam* f *e* g *funções* $f, g : N \longrightarrow \mathbb{R}^+$*. Se diz que* $f(n) = O(g(n))$*, se existem inteiros positivos* c *e* n_0 *de tal forma que, para cada número inteiro* $n \geq n_0$*,*

$$f(n) \leq cg(n).$$

Quando $f(n) = O(g(n))$*, dizemos que* $g(n)$ *é um limite superior para* $f(n)$*, ou mais precisamente, que* $g(n)$ *é um limite superior assintótico para* $f(n)$*, para enfatizar que estamos reprimendo fatores constantes.*

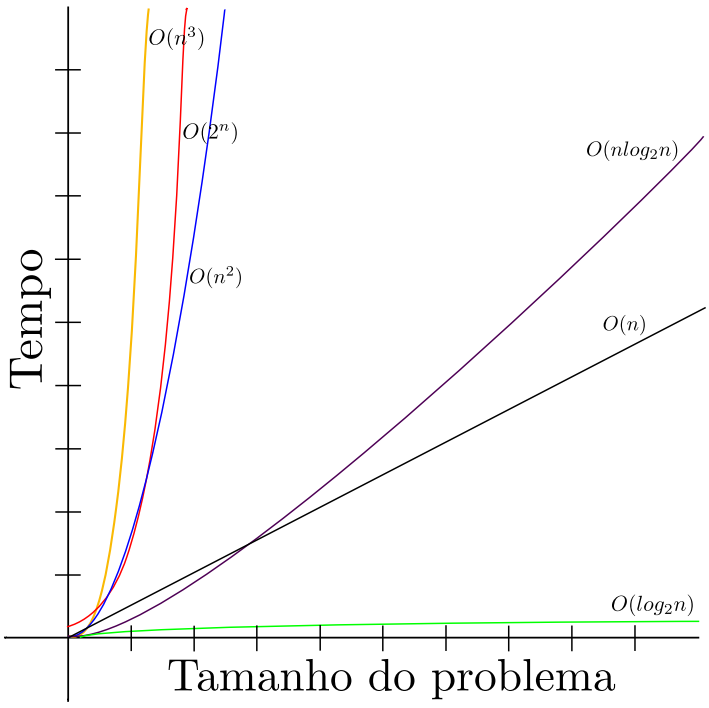


Figura 1.2: *Complexidade de Algoritmos.*

Na figura 1.2 podemos ver que se o tamanho do problema é pequeno não tem muita diferença entre as complexidades, mas entre elas a que faz o trabalho mais rápido é a complexidade constante $O(1)$, seguida de $O(log_2 n)$ que também faz o trabalho bem rápido, logo encontramos a complexidade linear $O(n)$ que é a melhor

[1]Tomado da pagina 277, seção 7.1 do livro [26].

opção se algo não pode ser determinado sem examinar toda a entrada. Sempre é bom evitar os algoritmos com complexidades polinomiais (só é boa para problemas de tamanho pequeno) e exponenciais, pois são as que tem o maior tempo de execução.

Exemplo 1.60 *Os seguintes algoritmos representam como calcular o somatório dos n primeiros inteiros.*

```
soma1:=function(n)
local r;
r:=(n*(n+1))/2;
return r;
end;
```

```
soma2:=function(n)
local r, i;
r:=0;
for i in [1..n] do
 r:=r+i;
od;
return r;
end;
```

```
soma3:=function(n)
local r, i, j;
r:=0;
for i in [1..n] do
 for j in [1..i] do
  r:=r+1;
 od;
od;
return r;
end;
```

Para determinar o tempo de complexidade de cada algoritmo vamos contar o número de operações em cada um deles. O primeiro algoritmo "soma1" tem 5 linhas, mas somente a linha 3 tem operações, de fato, contando a atribuição da variável tem 4 operações, e assim sua complexidade sera constante $O(4)$; o segundo algoritmo "soma2" começa na linha 3 onde tem uma operação, logo na linha 4 faz o trabalho n-vezes, isto é, n operações e na seguinte linha faz essa soma outras n-vezes, o que produz $1+n+n=2n+1$ operações e portanto tem uma complexidade linear $O(n)$; o último algoritmo soma3 tem dois "for", de fato um dentro do outro e assim tem uma complexidade quadrática $O(n^2)$. Portanto, embora os três algoritmos produzam o mesmo resultado "soma3" é a opção menos recomendável, pois tem um maior gasto de tempo na execução.

CAPÍTULO 2

Processo pseudoaleatório para gerar elementos de um grupo

2.1 Representando grupos em computadores

Na prática existem três situações básicas que ocorrem, e que afetam nossa capacidade de extrair informações sobre um grupo G computacionalmente.

A. Podemos representar todos os elementos de G no computador, e temos métodos para calcular representações de inversos e produtos de elementos do grupo. Um exemplo disto pode ser definir um grupo G como uma apresentação finita de $< X|R >$.

B. Como na Situação A e, além disso, podemos decidir se duas representações de elementos no grupo definem o mesmo elemento. Um exemplo é um grupo de permutação G em um conjunto finito definido por um conjunto gerador arbitrário.

C. Como na situação B e, além disso, temos um conjunto gerador especial $g_1, ..., g_r$ de G e um algoritmo que pode computar para cada elemento $g \in G$ uma palavra única w_g com $g = w_g(g_1, ..., g_r)$. Dito algoritmo é conhecido como um **algoritmo de reescrita**. Como exemplo está um grupo de permutação G com um chamado "grupo gerador forte". Na seção 3.5 vamos encontrar dito algoritmo, ele é chamado $SCHREIER-SIMS$.

Definição 2.1 *Dado $G =< S >$, um SLP (straight-line program) de comprimento m calculado para algum $g \in G$ é uma sequência de palavras $(w_1, ..., w_m)$, de modo que, para cada i, w_i é um símbolo para algum elemento de S, ou $w_i =< \{w_j, -1\} >$ para algum $j < i$, ou $w_i =< \{w_j, w_k\} >$ para algum $j, k < i$, de modo que se as palavras forem avaliadas em G da maneira óbvia, então o valor de w_m é g.*

Exemplo 2.2 *Se tivermos* $G =< S >$, *com* $S = \{x_1, x_2\}$, *então*

$$w_1 := x_1, \qquad w_2 := x_2, \qquad w_3 := (w_1 w_2)^2, \qquad w_4 := {w_2}^2 {w_3}^{-1} w_2$$

definem uma sequência de elementos SLP. Considerando $G = Alt(6)$, *com* $x_1 = (1,2,3)$ *e* $x_2 = (2,3,4,5,6)$, *teremos que os elementos*

$$w_1 := (1,2,3), \quad w_2 := (2,3,4,5,6), \quad w_3 := (2,5)(4,6), \quad w_4 := (4,5,6)$$

definem uma sequência de elementos SLP para $g = (4,5,6)$.

Na teoria de grupos computacional existem diferentes maneiras de descrever ou especificar um grupo G, numa pesquisa feita por Charles Sims em [25] especifica que são 4 maneiras diferentes, neste texto vamos trabalhar com uma delas, chamada black box. As descrições de black box são uma abstração da mínima informação que seria necessária para começar a fazer cálculos num grupo finito.

Segundo [21] um grupo de black box G é um grupo cujos elementos são codificados como palavras em A^* de comprimento N sobre um alfabeto A, para algum inteiro positivo N. Não exigimos que os elementos do grupo tenham uma representação única como uma palavra ou cadeia, e nem todas as palavras precisam corresponder aos elementos do grupo. Dadas palavras que representam elementos $h, g \in G$, podemos calcular a palavra que representa o elemento hg, calcular a palavra que representa g^{-1} e decidir quando $h = 1_G$. Assim um dos exemplos clássicos representados por black box são os grupos de permutações com os que estamos trabalhando, onde o alfabeto é o conjunto $A = \{1,2,3,...,n\}$.

2.2 Algoritmo

Muitos dos algoritmos da teoria de grupos computacional, dependem, em grande medida, de gerar elementos aleatórios uniformemente distribuídos de um grupo G. Infelizmente, mesmo para grupos de black box, não existe um método conhecido que gere elementos aleatórios que sejam completamente satisfatórios. Para os grupos de permutação finita isto é mais fácil do que para muitas outras apresentações de grupos, no capítulo 3 apresentaremos como pode ser possível isto.

O algoritmo que apresentamos aqui é o proposto em [8], que gera uma sequência de elementos que não está garantida para estar uniformemente distribuído, é bem rápido e parece comportar-se satisfatoriamente nos algoritmos em que

foi usado até agora.

Começamos com uma lista $x = [x_1, ..., x_k]$ de elementos tais que $G =< x >$. Primeiramente vamos replicar a lista, se é necessário, fazendo que a lista inicial tenha r elementos, onde o valor adequado para r esta em torno de 10, de fato, no artigo [8] encontramos que é suficiente tomar o valor de r sendo $M\acute{a}x(k, 10)$. Nós escolhemos aleatoriamente inteiros diferentes $s, t \in \{1, ..., r\}$ e substituímos x_s por $x_s x_t^{\pm 1}$ ou $x_t^{\pm 1} x_s$. Depois de fazer isto muitas vezes, começamos a retornar os novos valores de x_s como nossos elementos aleatórios.

Em [8] mostra que em efeito o algoritmo retorna elementos aleatoriamente distribuídos uniformemente, mas não há uma estimativa razoável conhecida sobre quanto tempo vamos ter que esperar por isso, e é provável que 50 mudanças não sejam suficientes. A versão que apresenta o livro base [15] é uma modificação devido ao Leedham-Green, que envolve o uso de um acumulador x_0 que é usado como valor de retorno.

A seguir vamos encontrar o pseudo algoritmo das funções $PRINITIALIZE$ e $PRRANDOM$. A função $PRINITIALIZE$ permite inicializar a lista dos $\{x_1, ..., x_r\}$, ela executa o processo de substituição da função $PRRANDOM$ n vezes e retorna a lista desejada. Em algumas aplicações, precisamos saber exatamente como os elementos aleatórios retornados foram derivados dos geradores originais, para isto vamos usar os elementos SLP, neste caso os w_i expressam cada elemento x_i como uma palavra nos geradores originais $x_1, ..., x_k$ de G. Denotamos os geradores do grupo SLP como $\hat{x}_i$.

Algoritmo 2.1: $PRINITIALIZE(X,r,n)$

Input: $X = [x_1,...,x_k]$ gerando um grupo de caixa preta, $r,n \in \mathbb{N}$

Output: Lista χ e uma lista ω com os elementos SLP associados

```
Sejam x̂_i (1 ≤ i ≤ k) os geradores de um grupo SLP;
for i ∈ [1..k] do
    w_i := x̂_i;
end
for i ∈ [k+1..r] do
    x_i := x_{i-k}; w_i := w_{i-k};
end
x_0 := 1_G; w_0 := ε;
χ := [x_0, x_1, ..., x_r]; ω := [w_0, w_1, ..., w_r];
for i ∈ [1..n] do
    PRRANDOM(χ,ω);
end
return χ, ω;
```

Algoritmo 2.2: $PRRANDOM(\chi,\omega)$

Input: Listas χ e ω

Output: Um elemento de G e seu correspondente elemento SLP associado

```
s := RANDOM([1..r]);
t := RANDOM([1..r]/[s]);
x := RANDOM([1..2]);
e := RANDOM(-1,1);
if x = 1 then
    x_s := x_s x_t^e; x_0 := x_0 x_s;
    w_s := w_s w_t^e; w_0 := w_0 w_s;
else
    x_s := x_t^e x_s; x_0 := x_s x_0;
    w_s := w_t^e w_s; w_0 := w_s w_0;
end
return a;
```

Para fazer a implementação no GAP vamos omitir pelo momento os elementos SLP e assim nossa função *prrandom* vai pegar a lista x de geradores do grupo G tal que $G =< x >$ e gerar aleatoriamente um elemento do grupo. Na verdade este é um algoritmo que gera elementos pseudo aleatoriamente, pois para que eles fossem gerados

aleatoriamente teríamos que tomar, por exemplo, x sendo um conjunto de geradores fortes para G, isto pode ser feito mediante o algoritmo $SCHREIER-SIMS$ que será exibido no capítulo 3.

Código 2.1 prinitialize(g,r,n)

```
prinitialize:=function(g,r,n)
local S, x, k, i, a, kk;
x:=[]; k:=Length(g); x[1]:=();
for i in [2..k+1] do
  x[i]:=g[i-1];
od;
kk:=k+2;
for i in [kk..r] do
  x[i]:=x[i-(k+1)];
od;
for i in [1..n] do
  prrandom(x);
od;
return x;
end;
```

Código 2.2 prrandom(x)

```
prrandom:=function(x)
local s, t, e, a, r, xx;
a:=(); r:=Length(x);
s:=Random([1..r]);
t:=Random(Difference([1..r],[s]));
xx:=Random([1,2]);
e:=Random([-1,1]);
if xx=1 then
  x[s]:=x[s]*x[t]^e;
  a:=a*x[s];
else
  x[s]:=x[t]^e*x[s];
  a:=x[s]*a;
fi;
return a;
end;
```

Exemplo 2.3 *Vamos gerar alguns elementos do grupo dos quatérnions aplicando o exemplo 1.10 e para grupo simétrico S_7. Nos dois casos vamos tomar $r = 10$ e $n = 20$.*

```
gap> Read("orbit");
```

```
gap> Read("prrandom");
gap> Read("prinitialize");
gap> x:=[(1,3,2,4)(5,8,6,7),(1,5,2,6)(3,7,4,8),(1,7,2,8)(3,6,4,5)];;
gap> x1:=prinitialize(x,10,20);;
gap> prrandom(x1);
(1,2)(3,4)(5,6)(7,8)
gap> prrandom(x1);
(1,7,2,8)(3,6,4,5)
gap> s7:=[(1,2,3,4,5,6,7),(1,2)];;
gap> s71:=prinitialize(s7,10,20);;
gap> prrandom(s71);
(1,4,7,3,6,2,5)
```

CAPÍTULO 3

Computação em grupos de permutação de ordem finita

3.1 Cálculo de órbitas e estabilizadores

Nesta seção vamos trabalhar com um grupo G finito que é gerado por uma sequência finita $X = \{x_1, ..., x_r\}$, isto é, $G =< X >$, atuando sobre um conjunto finito Ω. Suponha que para todo $\alpha \in \Omega$ e todo $x \in X$ a imagem α^x sob ação pode ser calculada e podemos determinar quando dois elementos de Ω são iguais. Primeiro, faremos um algoritmo para o cálculo da órbita α^G de um ponto $\alpha \in \Omega$ sobre G.

Algoritmo 3.1: $ORBIT(\alpha, X)$

Input: $\alpha \in \Omega$, $X = [x_1, ..., x_r]$, $x_i \in Sym(\Omega)$ com $< X >= G$
Output: α^G

```
Δ := [α]
for β ∈ Δ, x ∈ X do
    if β^x ∉ Δ then
        Add(~Δ, β^x);
    end
end
return Δ;
```

Neste algoritmo a condição $x_i \in Sym(\Omega)$ corresponde à bijeção representada pelo elemento x_i de X. De fato, $Sym(\Omega)$ tem a mesma quantidade de elementos que X. Para ver que o algoritmo acima está correto, observe que todos os elementos armazenados em Δ são imagens sob um elemento de X de um elemento que já está em Δ. Se $\beta \in \Delta$, então $\beta = \alpha^g$ para algum $g \in G$ e como X gera G, então $g = x_{i_1} x_{i_2} ... x_{i_l}$ onde cada $x_{i_j} \in A_x = X \cup X^{-1}$. Agora, pelo fato de G ser finito os x_{i_j} tem ordem finita. Sem perda de generalidade assumamos que cada $x_{i_j} \in X$ e vamos mostrar por indução sobre um tempo constante t assim: para o tempo $t = 0$, $\beta = \alpha$ e assim $\Delta = [\alpha]$. Para o tempo $t = 1$, se $\alpha^{x_{i_1}} \notin \Delta$, se teria $\Delta = [\alpha, \alpha^{x_{i_1}}]$. Para o tempo $t = 2$, se $\alpha^{x_{i_1} x_{i_2}} \notin \Delta$ se teria

$\Delta = [\alpha, \alpha^{x_{i_1}}, \alpha^{x_{i_1}x_{i_2}}]$ (lembremos que $(\alpha^{x_{i_1}})^{x_{i_2}} = \alpha^{x_{i_1}x_{i_2}}$). Assim por adiante suponha que no tempo $t-1$ o resultado é valido, então $\gamma := \alpha^{g'} \in \Delta$, onde $g' = x_{i_1}x_{i_2}...x_{i_{t-1}}$ assim no tempo t temos que $\beta = \gamma^{x_{i_t}}$ é armazenado em Δ.

Com base no pressuposto de que as imagens α^x podem ser calculadas em tempo constante, e que a adesão de elementos de Ω em Δ, pode ser provado em tempo constante, assim a complexidade do algoritmo 3.1 é o produto da quantidade de elementos armazenados em Δ com r, isto é, $O(|\Delta|r)$.

A seguir exibiremos a implementação do algoritmo 3.1 no sistema GAP.

Exemplo 3.1 *Se consideramos o grupo gerado pelo conjunto de permutações* $x = \{x_1, x_2, ..., x_r\}$ *e o conjunto* $\Omega = \{1, ..., n\}$*, temos a ação natural dada por*

$$\begin{array}{rcccl} \varphi: & G =< x > & \longrightarrow & Sym(\Omega) & \\ & g & \longmapsto & \pi: & \begin{array}{ccc} \Omega & \longrightarrow & \Omega \\ \alpha & \longmapsto & \alpha^g. \end{array} \end{array}$$

Nesse caso, o algoritmo para calcular a órbita de um elemento $a \in \Omega$ *é dado por.*

Código 3.1 orbit(a, x)

```
orbit:= function(a, x)
local delta, r, b, i;
r:=Length(x);
delta:= [a];
for b in delta do
  for i in [1..r] do
    if not b^x[i] in delta then
      Add(delta,b^x[i]);
    fi;
  od;
od;
return delta;
end;
```

Uma das possíveis aplicações do algoritmo 3.1, é saber se um grupo é transitivo, isto é, se uma ação natural de um grupo de permutação num conjunto $[1..n]$ é transitiva. Isto pode ser feito segundo a definição 1.19 sabendo se existe somente uma órbita no grupo $G =< X >$ baixo a ação natural.

Código 3.2 istransitive(n,x)

```
istransitive:= function(n,x)
local delta, b, i, j;
delta:= [];
for i in [1..n] do
  b:=orbit(i,x);
    if not b in delta then
      Add(delta,b);
    fi;
od;
for j in [1..n] do
  if Length(delta[j]) < n then
    return "falso";
  else
    return "verdadeiro";
  fi;
od;
end;
```

Exemplo 3.2 *Vamos ver para alguns grupos se ação natural é transitiva.*

```
gap> Read("orbit");
gap> Read("istransitive");
gap> istransitive(4,[(1,2,3,4),(1,2)]);
"verdadeiro"
gap> istransitive(5,[(1,2,3,4),(1,2)]);
"falso"
gap> istransitive(6,[(1,2,3,4,5,6),(2,6)(3,5)]);
"verdadeiro"
gap> istransitive(8,[(1,2,3,4,5,6),(2,6)(3,5)]);
"falso"
```

Lembrando que as órbitas da ação conjugação sobre o mesmo grupo G produz as classes de conjugação. A seguinte função vai retornar a classe de conjugação que contém o elemento g, isto é $Cl_G(g)$.

Exemplo 3.3 *Vamos considerar o grupo simétrico de* 4 *elementos e encontrar a classe de conjugação que tem, por exemplo, o elemento* $(1,3,4)$. *O GAP tem a ferramenta "*$ConjugacyClasses(< G >)$*"; que gera todas as classes conjugadas de um grupo* G *e também "*$ConjugacyClass(< G >, elem)$*" que retorna a classe conjugada de* G *que contém o elemento "elem".*

```
gap> x:=[(1,2,3,4),(1,2)];
[ (1,2,3,4), (1,2) ]
```

Código 3.3 orbitconjugation(a,x)

```
orbitconjugation:= function(a,x)
local delta, i, j, h;
delta:=[a];
for i in delta do
 for j in x do
  h:=Inverse(j)*i*j;
  if not h in delta then
   Add(delta,h);
  fi;
 od;
od;
return delta;
end;
```

```
gap> ConjugacyClasses(Group(x));
[ ()^G, (3,4)^G, (2,3,4)^G, (1,2)(3,4)^G, (1,2,3,4)^G ]
gap> h:=ConjugacyClass(Group(x),(1,3,4));
(1,3,4)^G
gap> Read("orbitconjugation");
gap> h1:=orbitconjugation((1,3,4),x);
[ (1,3,4), (1,2,4), (2,3,4), (1,2,3),
 (1,4,2), (1,3,2), (2,4,3), (1,4,3) ]
gap> Elements(h1)=Elements(h);
true
```

Do algoritmo 3.1 temos que para cada $\beta \in \alpha^G$ existe $u_\beta \in G$ tal que $\alpha^{u_\beta} = \beta$. Aplicando o Teorema da Orbita-estabilizador 1.16 obtemos que o conjunto $T := \{u_\beta | \beta \in \alpha^G, \alpha^{u_\beta} = \beta\}$ é um transversal à direita do estabilizador de α em G. Estendendo o algoritmo 3.1 para a ação natural também podemos encontrar o transversal à direita associado T. Esta função será chamada *ortransversal*.

Considerando o transversal à direita T e fazendo uso do teorema 1.44 para o subgrupo G_α de G, obtemos que

$$G_\alpha = \left\langle \{u_\beta x \overline{u_\beta x}^{-1} | u_\beta \in T, x \in X\} \right\rangle.$$

Mas lembremos que o elemento $\overline{u_\beta x} = T \cap G_\alpha u_\beta x$, onde a primeira condição dada por T diz que $\alpha^{u_\beta} = \beta$ e a segunda que $\alpha^{g u_\beta x} = \alpha^{u_\beta x}$. Pois, $G_\alpha u_\beta x = \{g \in G | \alpha^g = \alpha^{u_\beta x}\} = \{g u_\beta x \in G | \alpha^{g u_\beta x} = \alpha^{u_\beta x}\}$, assim na interseção das duas condições teremos o elemento $u_{\beta^x} \in G$ tal que $\alpha^{u_{\beta^x}} = \beta^x$, logo o estabilizador do elemento α, G_α é dado por

Código 3.4 ortransversal(a,x)

```
ortransversal:= function(a,x)
local delta, t, i, b, r;
r:=Length(x); delta:= [a]; t:=[()];
for b in delta do
  for i in [1..r] do
    if not b^x[i] in delta then
      Add(delta,b^x[i]);
      Add(t,t[Position(delta,b)]*x[i]);
    fi;
  od;
od;
return [delta, t];
end;
```

$$G_\alpha = \left\langle \{u_\beta x {u_\beta x}^{-1} | u_\beta \in T, x \in X\} \right\rangle.$$

Logo o seguinte algoritmo retorna a lista Δ dos elementos (β, u_β) para $\beta \in \alpha^G$ e a sequência geradora Y de G_α.

Algoritmo 3.2: $ORBITSTABILIZER(\alpha, X)$

Input: $\alpha \in \Omega$, $X = [x_1, ..., x_r]$, $x_i \in Sym(\Omega)$ com $< X >= G$

Output: Δ, Y (com as descrições acima)

$\Delta := [(\alpha, 1_G)]$

$Y := []$

for $(\beta, u_\beta) \in \Delta$, $x \in X$ **do**

  if $\beta^x \notin \Delta$ **then**

    Add$(\sim \Delta, (\beta^x, u_\beta x))$;

  else

    Add$(\sim Y, u_\beta x (u_{\beta^x})^{-1})$;

  end

end

return Δ, Y;

Agora veremos a implementação no GAP do algoritmo 3.2 para a ação natural, esta será a função que calcula o estabilizador de um elemento a num grupo $G =< x >$.

Na seção 1.2.1 vimos que para a ação da conjugação sobre o mesmo grupo G o estabilizador de um elemento $g \in G$, G_g, é igual ao centralizador de g em G, assim aplicando o algoritmo 3.2 para a ação podemos obter o centralizador de um elemento g de $G =< X >$.

Código 3.5 stabilizer(a, x)

```
stabilizer:= function(a, x)
local delta, t, i, j, y, h, r, b, m;
r:=Length(x); delta:= [a]; t:=[()]; y:=[];
for b in delta do
  for i in [1..r] do
    h:=b^x[i];
    if not h in delta then
      Add(delta,h);
      Add(t,t[Position(delta,b)]*x[i]);
    else
      m:=t[Position(delta,b)]*x[i]*Inverse(t[Position(delta,h)]);
          if not m in y then
        Add(y,m);
      fi;
    fi;
  od;
od;
return Group(y);
end;
```

Código 3.6 centralizer(a, x)

```
centralizer:= function(a, x)
local delta, t, i, j, y, h, m;
delta:= [a]; t:=[()]; y:=[];
for i in delta do
  for j in x do
      h:=Inverse(j)*i*j;
    if not h in delta then
      Add(delta,h);
      Add(t,t[Position(delta,i)]*j);
    else
      m:=t[Position(delta,i)]*j*Inverse(t[Position(delta,h)]);
      if not m in y then
        Add(y,m);
      fi;
    fi;
  od;
od;
return Group(y);
end;
```

Exemplo 3.4 *Seja S_4 o grupo simétrico de ordem 4, $S_4 =< (1,2,3,4),(1,2) >$, primeiro faremos uso da ferramenta "Centralizer($< G >$,elem)" do GAP para encontrar o centralizador de um elemento qualquer de S_4. Por exemplo para o elemento $(1,3)(2,4)$ temos que;*

```
gap> x:=[(1,2,3,4),(1,2)];;
gap> g:=Centralizer(Group(x),(1,3)(2,4));
Group([ (1,3)(2,4), (1,4)(2,3), (2,4) ])
```

Com a ferramenta do GAP o centralizador de $(1,3)(2,4)$ em S_4 é armazenado em g, isto é $g = C_{S_4}((1,3)(2,4))$; agora vamos chamar nosso algoritmo e armazenar em gg o centralizador $C_{S_4}((1,3)(2,4))$ dado pelo algoritmo, assim o GAP estabelece se os dois grupos gerados são iguais.

```
gap> Read("centralizer");
gap> gg:=centralizer((1,3)(2,4),x);
Group([ (1,2,3,4), (), (1,4)(2,3), (2,4) ])
gap> g=gg;
true
```

Considerando ação da conjugação sobre o conjunto Ω dos subgrupos de G temos que o estabilizador de um elemento $H \in \Omega$, G_H, é igual ao normalizador de H em G, assim aplicando o algoritmo 3.2 para esta ação, obtemos que o normalizador do subgrupo H de $G =< X >$.

Código 3.7 normalizer(h,x)

```
normalizer:= function(h, x)
local delta, t, i, j, y, k, m;
delta:= [h]; t:=[()]; y:=[];
for i in delta do
  for j in x do
    k:=Inverse(j)*Elements(i)*j;
    if not Group(k) in delta then
      Add(delta,Group(k)); Add(t,t[Position(delta,i)]*j);
    else
      m:=t[Position(delta,i)]*j*Inverse(t[Position(delta,Group(k))]);
      if not m in y then
        Add(y,m);
      fi;
    fi;
  od;
od;
return Group(y);
end;
```

3.2 Vetores de Schreier

Nesta seção vamos abordar os elementos u_β do transversal quando $\Omega = \{1,...,n\}$, para algum $n \in \mathbb{N}$.

Definição 3.5 *Um vetor de Schreier para $\alpha \in \Omega$ é uma lista v de comprimento n com as seguintes propriedades:*

1. $v[\alpha] = -1$,
2. *para $\gamma \in \alpha^G - \{\alpha\}$, $v[\gamma] \in \{1,...,r\}$; mais ainda, $v[\gamma] = i$ onde γ é armazenado em Δ como β^{x_i} no algoritmo 3.1,*
3. $v[\beta] = 0$ *para* $\beta \notin \alpha^G$.

Algoritmo 3.3: $ORBITSV(\alpha, X)$

Input: $\alpha \in \{1,...,n\}$, $X = [x_1,...,x_r]$, $x_i \in S_n$ com $< X >= G$
Output: α^G e o vetor de Schreier de α

```
for i ∈ {1,...,n} do
    v[i]:=0
end
for β^x ∈ Δ, i ∈ {1,...,r} do
    if β^{x_i} ∉ Δ then
        Add(∼ Δ, β^x), v[β^{x_i}] := i
    end
end
return Δ, v;
```

O algoritmo 3.3 tem complexidade constante $O(nr)$, pois é o resultado de r vezes fazer o algoritmo nas n entradas do vetor.

Exemplo 3.6 *Vamos implementar o algoritmo anterior para $n = 6$, $X = [x_1, x_2, x_3]$, $x_1 = (1,2,3)$, $x_2 = (1,5)(2,3)$, $x_3 = (5,6)$ e $\alpha = 2$. Começamos com $\Delta = [2]$ e $v = [0,-1,0,0,0,0,0]$.*

1. *Para $\beta = 2$ temos que:*
   - 1.1. *Para $x_1 \in X$, $\beta^{x_1} = 2^{(1,2,3)} = 3 \notin \Delta$, adicionamos 3 a Δ e fazemos $v[3] = 1$.*
   - 1.2. *Para $x_2 \in X$, $\beta^{x_2} = 2^{(1,5)(2,3)} = 3 \in \Delta$.*
   - 1.3. *Para $x_3 \in X$, $\beta^{x_3} = 2^{(5,6)} = 2 \in \Delta$.*

   Agora nossos dados são $\Delta = [2,3]$ e $v = [0,-1,1,0,0,0]$.
2. *Seja $\beta = 3$:*

2.1. Para $x_1 \in X$, $\beta^{x_1} = 3^{(1,2,3)} = 1 \notin \Delta$, *adicionamos* 1 *a* Δ *e fazemos* $v[1] = 1$.

2.2. Para $x_2 \in X$, $\beta^{x_2} = 3^{(1,5)(2,3)} = 2 \in \Delta$.

2.3. Para $x_3 \in X$, $\beta^{x_3} = 3^{(5,6)} = 3 \in \Delta$.

Neste momento teremos $\Delta = [2,3,1]$ *e* $v = [1,-1,1,0,0,0]$.

3. *Para* $\beta = 1$, *somente* $x_2 \in X$, *pois* $\beta^{x_1} = 1^{(1,5)(2,3)} = 5 \notin \Delta$, *então adicionamos* 5 *a* Δ *e fazemos* $v[5] = 2$ *e assim obtemos* $\Delta = [2,3,1,5]$ *e* $v = [1,-1,1,0,2,0]$.

4. *Para* $\beta = 5$ *temos* $x_3 \in X$, $\beta^{x_3} = 5^{(5,6)} = 6 \notin \Delta$, *assim obtemos finalmente que* $\Delta = [2,3,1,5,6]$ *e* $v = [1,-1,1,0,2,3]$ *pois para* $\beta = 6$, $\beta^{x_i} \in \Delta$ *para* $i = 1,2,3$.

Agora veremos a implementação do algoritmo 3.3 junto com o exemplo 3.6 no GAP.

Código 3.8 orbitsv(a,x)

```
orbitsv:=function(a,x)
local delta, n, r, v, j, i, k;
n:=LargestMovedPoint(x);
v:=[1..n];
for i in [1..n] do v[i]:=0;
od;
r:=Length(x);
delta:=[a];
v[a]:=-1;
for j in delta do
  for k in [1..r] do
    if not j^x[k] in delta then
      Add(delta,j^x[k]);
      v[j^x[k]]:=k;
    fi;
  od;
od;
return [delta, v];
end;
```

```
gap> Read("orbitsv");
gap> x:=[(1,2,3),(1,5)(2,3),(5,6)];;
gap> orbitsv(2,x);
[ [ 2, 3, 1, 5, 6 ], [ 1, -1, 1, 0, 2, 3 ] ]
```

O seguinte algoritmo faz a função $UBETA$, que podemos usar para encontrar o u_β associado a um elemento $\beta \in \Omega$ tendo o vetor de Schreier do elemento $\alpha \in \Omega$. Se $\beta \notin \alpha^\Omega$, a função $UBETA$ retorna "falso".

Algoritmo 3.4: $UBETA(v, \alpha, \beta)$

Input: O vetor de Schreier v para algum $\alpha \in \Omega$, $\beta \in \Omega$

Output: u_β com $\alpha^{u_\beta} = \beta$ ou falso se $\beta \notin \alpha^G$

```
if v[β] = 0 then
    return falso
end
u := 1_G; k := v[β];
while k ≠ −1 do
    u := x_k u; β := β^(x_k^−1); k := v[β];
end
return u;
```

O primeiro que o algoritmo 3.4 faz é determinar se $\beta \in \alpha^\Omega$. Logo ele vai criar duas variáveis u e k, onde u começa no elemento identidade do grupo e k começa no elemento que encontra-se na posição β no vetor de Schreier. Assim, se $k = -1$ teremos $\beta = \alpha$, pois, se isto acontece no começo teremos $u = 1_G = ()$ e se $k \neq -1$ o algoritmo vai fazer o seguinte: Ao ser $k \neq -1$ existe um elemento $m \in \Omega$ que deixamos ser nosso novo β, tal que $m^{x_k} = \beta$ (é por isto que $m = \beta := \beta^{x_k^{-1}}$), assim em u vamos operar os novos x_k que resultam de fazer $k = v(m)$ até chegar ao $k = -1$, onde o algoritmo vai parar e retornar ao último elemento armazenado em u.

A seguir apresentaremos a implementação do algoritmo 3.4 no sistema GAP junto com a aplicação para o vetor do exemplo 3.6 e alguns valores de β.

Código 3.9 ubeta(b,v,x)

```
ubeta:=function(b,v,x)
local u, k;
if v[b]=0 then
  return ("falso");
  else
  u:=();
  k:=v[b];
  while k<>-1 do
    u:=x[k]*u;
    b:=b^Inverse(x[k]);
    k:=v[b];
  od;
fi;
return u;
end;
```

```
gap> Read("ubeta");
```

```
gap> ubeta(4,[1,-1,1,0,2,3],[(1,2,3),(1,5)(2,3),(5,6)]);
"falso"
gap> ubeta(5,[1,-1,1,0,2,3],[(1,2,3),(1,5)(2,3),(5,6)]);
(1,2,5)
gap> ubeta(6,[1,-1,1,0,2,3],[(1,2,3),(1,5)(2,3),(5,6)]);
(1,2,6,5)
```

Depois de ter o vetor de Schreier associado ao elemento α do grupo $G =< x >$, podemos gerar aleatoriamente um elemento $g \in G$ e fazer $\beta := \alpha^g$ donde temos que $gu_\beta{}^{-1}$ é um elemento aleatório do estabilizador G_α. O seguinte algoritmo implementa esse processo usando um vetor de Schreier para a órbita e a função $PRRANDOM$ da seção 2.2.

Algoritmo 3.5: $RANDOMSTAB(\alpha, v, x, xx)$

Input: $G =< x > \leq Sym(n)$, o vetor de Schreier v para algum $\alpha \in \Omega$ e xx é a lista inicializada com a função $PRINITIALIZE$

Output: Um elemento aleátorio de G_α

1 $g := PRRANDOM(xx)$;

2 $h := UBETA(\alpha^g, v, x)$;

3 **return** gh^{-1};

Código 3.10 randomstab(a,x)

```
randomstab:=function(a,x)
local g, h, xx, v;
xx:=prinitialize(x,10,20);
g:=prrandom(xx);
v:=orbitsv(a,x)[2];
h:=ubeta(a^g,v,x);
return g*Inverse(h);
end;
```

Exemplo 3.7 *Neste exemplo apresentamos a implementação do código 3.10 para gerar elementos aleatórios do estabilizador G_2, para um grupo qualquer dado, vamos utilizar a ferramenta do GAP chamada StructureDescription($< G >$), para saber com qual grupo estamos trabalhando.*

```
gap> Read("orbitsv");
gap> Read("ubeta");
gap> Read("prrandom");
gap> Read("prinitialize");
gap> Read("orbitsv");
```

```
gap> Read("randomstab");
gap> x:=[(5,6,7),(2,3)(5,6),(2,3,4),(1,2)(3,4)];;
gap> StructureDescription(Group(x));
"(C3␣x␣A4)␣:␣C2"
gap> randomstab(2,x);
(1,3,4)(5,7,6)
gap> randomstab(2,x);
(3,4)(5,7)
gap> randomstab(2,x);
(1,4)(5,6)
```

3.3 Verificando para $Alt(\Omega)$ e $Sym(\Omega)$

Um algoritmo de Monte Carlo depende de amostragens para obtenção de resultados numéricos, o Método foi formalizado em 1949, por meio do artigo titulado *Monte Carlo Method*, publicado por John Von Neumann e Stanislav Ulam. Um algoritmo de Monte Carlo, as vezes, pode produzir uma resposta errada, por isso um requisito do método é que um dos parâmetros de entrada desse algoritmo deve ser um número real ϵ com $0 < \epsilon < 1$, assim a probabilidade da resposta ser errada deve ser inferior a ϵ, para qualquer valor dos dados de entrada. O objetivo desta seção é descrever um algoritmo de Monte Carlo para testar se um grupo de permutação transitiva num conjunto Ω é $Alt(\Omega)$ ou $Sym(\Omega)$ quando $|\Omega| \geq 8$. A resposta 'sim' é garantida corretamente, enquanto a resposta 'não' tem pouca probabilidade de estar errada. Se a resposta é positiva, é fácil verificar se o grupo é $Alt(\Omega)$ ou $Sym(\Omega)$ olhando se os elementos geradores são permutações pares.

Teorema 3.8 *Sejam $G \leq Sym(\Omega)$ transitivo sobre Ω com $|\Omega| = n$ e p um número primo com $n/2 < p < n-2$. Se G contém um elemento com um ciclo de comprimento p, então $G = Alt(\Omega)$ ou $G = Sym(\Omega)$.*

Demonstração. Seja $\sigma = (u_1, u_2, ..., u_p)$ o ciclo de comprimento p em G. Tomando $\Delta = \{u_1, ..., u_p\}$ como órbita de $U =< \sigma >$ temos, pelo teorema 1.22, que U^{Δ} é primitivo sobre Δ e como $|\Omega| = n < 2p = 2|\Delta|$, pelo teorema 1.23, obtemos que G é primitivo sobre Ω. Agora a condição $p < n-2$ garante que $n = p+k$ com $k \geq 3$ e portanto o teorema 1.24 diz que $G = Alt(\Omega)$ ou $G = Sym(\Omega)$. $\square$

O algoritmo desta seção, que chamaremos $ISALTSYM$, começará com um grupo $G =< x >$, onde, mediante a função $PRRANDOM$, escolhe aleatoriamente um elemento e examina se tem um ciclo de comprimento p, com $n/2 < p < n-2$, se isto acontece, então o algoritmo vai parar retornando "verdadeiro". Observe que só funciona para $|\Omega| \geq 8$, pois não tem primos com $n/2 < p < n-2$ quando $n < 8$.

Antes de fazer o algoritmo, temos que saber qual é a quantidade de elementos necessários para obter um resultado favorável com o mínimo erro possível, para isto vamos fazer uso da parte (b) do seguinte lema extraído da pagina 226 do livro [21].

Lema 3.9 **(a)** *Seja q um inteiro no intervalo $n/2 < q \leq n$. A proporção de elementos de S_n contendo um ciclo de comprimento q é $1/q$. Em A_n, esta proporção é $1/q$ se $q \leq n-2$, é 0 ou $2/q$ se $q \in \{n-1, n\}$.*

(b) *A proporção de elementos de $Sym(\Omega)$ e $Alt(\Omega)$ que contém um ciclo de comprimento p, para algum primo p no intervalo $n/2 < q < n-2$, é assintoticamente $\frac{Ln(2)}{Ln(n)}$.*

A seguir apresentamos a aplicação do algoritmo procurado no sistema GAP.

Código 3.11 isaltsym(n,x)

```
isaltsym:=function(n,x)
local a, i, c, d, j, ii, t, xx;
c:=Log(2^100,n);
d:=[];
a:=[];
xx:=prinitialize(x,10,20);
t:=false;
for i in [1..c] do
  a[i]:=prrandom(xx);
  d[i]:=CycleLengths(a[i],[1..n]);
od;
for ii in [1..c] do
  for j in [1..Length(d[ii])] do
    if IsPrime(d[ii][j]) and n/2 < d[ii][j] and d[ii][j] < (n-2) then
      t:=true;
    fi;
  od;
od;
if t=true then
  return ("verdadeiro");
else
  return ("falso");
fi;
end;
```

Exemplo 3.10 *Aqui apresentamos primeiramente o grupo dos quatérnions do exemplo 1.10, logo o grupo simétrico de grau 10 e o grupo alternado de grau 9. Lembremos que para um grupo, em geral, temos que verificar primeiro se ele é transitivo.*

```
gap> Read("orbit");
gap> Read("istransitive");
```

```
gap> Read("prrandom");
gap> Read("prinitialize");
gap> Read("isaltsym");
gap> q:=[(1,3,2,4)(5,8,6,7),(1,5,2,6)(3,7,4,8),(1,7,2,8)(3,6,4,5)];;
gap> istransitive(8,q);
"verdadeiro"
gap> isaltsym(8,q);
"falso"
gap> s10:=[(1,2,3,4,5,6,7,8,9,10),(1,2)];;
gap> isaltsym(10,s10);
"verdadeiro"
gap> a:=AlternatingGroup(9);;
gap> a9:=GeneratorsOfGroup(a);
[ (1,2,3,4,5,6,7,8,9), (7,8,9) ]
gap> isaltsym(9,a9);
"verdadeiro"
```

3.4 Grupos transitivos e primitivos. Sistemas de blocos

Nesta seção vamos trabalhar com um grupo $G \leq Sym(n)$ gerado por uma sequência finita $x = \{x_1, ..., x_r\}$. Lembrando que uma ação de G sobre um conjunto Ω é chamada transitiva se houver uma única órbita sob a ação então com ajuda da função "$Transitive$" criada na seção 3.1, podemos saber quando G é transitivo, isto porque nesta seção vamos trabalhar somente com grupos transitivos. O objetivo desta seção é testar se o grupo G que é transitivo e também primitivo, este último é, se a ação de G no conjunto $\Omega = \{1, ..., n\}$ só tem blocos triviais.

O algoritmo que descreveremos aqui foi feito pelo Michael D. Atkinson em [3], com ele saberemos se um sistema de blocos é minimal. O Robert M.Beals em [5] fez o algoritmo em tempo quase-linear e o Ákos Seress mostra no livro [21] um resultado dele e do M. Schönert do ano 1994, onde descreve uma versão alternativa de tempo $O(nLog^3(|G|) + nkLog(|G|))$. Embora os dois anteriores são muito melhores para o tempo de processamento do computador, aqui descreveremos o algoritmo inicial do Atkinson.

Para continuar vamos lembrar alguns resultados da seção 1.3.1:

1. Se Δ é um bloco sob uma ação, então as diferentes traslações Δ^g de Δ formam uma partição de Ω. O conjunto de translações é conhecido como um sistema de blocos.

2. Se G age sobre Ω, então uma relação de equivalência $\sim$ sobre Ω é chamada uma G-congruência se $\alpha \sim \beta$ implica que $\alpha^g \sim \beta^g$, para todo α, $\beta \in \Omega$ e $g \in G$.

3. Se G^Ω é transitiva, então as G-congruências e os sistema de blocos coincidem.

4. Se $S \subseteq \Omega \times \Omega$, então a G-congruência gerada por S é definida como a interseção de todas as G-congruências que contêm S.

Portanto, encontrar o mais pequeno sistema de blocos que contem os $k > 1$ pontos $\{\alpha_1, ..., \alpha_k\}$ de $\Omega = \{1, ..., n\}$ é equivalente a encontrar a G-congruência $\sim$ gerada pelo conjunto $\{(\alpha_1, \alpha_i) | 2 \leq i \leq k\}$. A função que exibiremos, de nome $MINIMALBLOCK$, manipula uma relação de equivalência $\sim$ sobre $\{1, ..., n\}$. Começamos colocando os k pontos na mesma classe de equivalência, e os outros pontos em classes separadas. Durante o desenvolvimento da função algumas dessas classes são mescladas; em qualquer fase do algoritmo, cada classe terá um representante armazenado e o valor final retornado de $\sim$ é a G-congruência $\sim$ gerada por $\{(\alpha_1, \alpha_i) | 2 \leq i \leq k\}$. Primeiro apresentaremos o algoritmo na sua forma básica, depois como é que ele funciona e por fim, como fazemos para programar ele.

Algoritmo 3.6: $MINIMALBLOCK1(G, \{\alpha_1, ..., \alpha_k\})$

Input: $G =< x >\leq Sym(n)$, G transitivo, $\alpha_1, ..., \alpha_k \in [1..n]$
Output: G-congruência $\sim$ gerada por $\{(\alpha_1, \alpha_i) | 2 \leq i \leq k\}$

```
Inicializo ~ com classes {α_i | 1 ≤ i ≤ k} com representante α_1 e
{{γ} | γ ≠ α_i (1 ≤ i ≤ k)};
for i ∈ [1..k−1] do
    q[i] := α_{i+1}; l := k−1; i := 1;
end
while i ≤ l do
    γ := q[i]; i := i+1;
    for x ∈ X do
        δ := REP(γ); k := REP(γ^x); λ := REP(δ^x);
        if k ≠ λ then
            Mescle as classes CLASS(k) e CLASS(λ), e faça k o representante da
            classe combinada.
            l := l+1; q[l] := REP(λ);
        end
    end
end
return ~;
```

No algoritmo 3.6 a função $REP(a)$ quer dizer o representante da classe onde esta o elemento a. Agora veremos com um exemplo como é que ele funciona.

Exemplo 3.11 *Pelo exemplo 3.2 temos que para $n = 6$ o grupo $g :=< [x_1, x_2] >$ com $x_1 = (1,2,3,4,5,6)$ e $x_2 = (2,6)(3,5)$ é transitivo, agora assumiremos que $\alpha_1 = 1$ e $\alpha_2 = 4$.*

1. *Começamos inicializando as classes com representantes assim: a classe $\{1,4\}$ com 1 como representante e as classes $\{i\}$ com i como representante para $i := 2,3,5,6$..*

2. *Façamos $q[1] := 4$, $i := 1$ e $l := 1$. Como $i = 1 \leq 1 = l$, fazemos $\gamma := q[1] = 4$ e assim:*

   2.1. *Para x_1 temos: $\delta = REP(\gamma) = REP(4) = 1$, $k = REP(\gamma^{x_1}) = REP(4^{x_1}) = REP(5) = 5$ e $\lambda = REP(\delta^{x_1}) = REP(1^{x_1}) = REP(2) = 2$. Como $k = 5 \neq 2 = \lambda$, então as classes $\{5\}$ e $\{2\}$ são mescladas sendo agora 2 o representante da nova classe, fazemos $l = 2$ e $q[2] = REP(2) = 2$.*

   2.2. *Para x_2 temos: $\delta = REP(\gamma) = 1$, $k = REP(\gamma^{x_2}) = REP(4^{x_2}) = REP(4) = 1$ e $\lambda = REP(\delta^{x_2}) = REP(1^{x_2}) = REP(1) = 1$. Como $k = 1 = \lambda$, todo fica igual.*

3. *Agora $i = 2$ e $l = 2$. Como $i = 2 \leq 2 = l$, fazemos $\gamma := q[2] = 2$ e assim:*

   3.1. *Para x_1 temos: $\delta = REP(2) = 5$, $k = REP(2^{x_1}) = REP(3) = 3$ e $\lambda = REP(5^{x_1}) = REP(6) = 6$. Como $k = 3 \neq 6 = \lambda$, então as classes $\{3\}$ e $\{6\}$ são mescladas sendo agora 6 o representante da nova classe, fazemos $l = 3$ e $q[3] = REP(6) = 6$.*

   3.2. *Para x_2 temos: $\delta = REP(2) = 5$, $k = REP(2^{x_2}) = REP(6) = 3$ e $\lambda = REP(5^{x_2}) = REP(3) = 3$. Como $k = 3 = \lambda$, todo fica igual.*

4. *Para $i = 3$ e $l = 3$. Como $i = 3 \leq 3 = l$, fazemos $\gamma := q[3] = 6$ e assim:*

   4.1. *Para x_1 temos: $\delta = REP(6) = 3$, $k = REP(6^{x_1}) = REP(1) = 1$ e $\lambda = REP(3^{x_1}) = REP(4) = 1$. Como $k = 1 = \lambda$, todo fica igual.*

   4.2. *Para x_2 temos: $\delta = REP(6) = 3$, $k = REP(6^{x_2}) = REP(2) = 5$ e $\lambda = REP(3^{x_2}) = REP(5) = 5$. Como $k = 5 = \lambda$, todo fica igual.*

5. *Como o $i = 4$ e $l = 3$, então a condição $i \leq l$ não se cumpre então as classes de equivalência que definem a G-congruência são: $\{1,4\}, \{2,5\}, \{3,6\}$.*

Agora vamos mostrar que efetivamente o algoritmo 3.6 produz a G-congruência gerada por $\{(\alpha_1, \alpha_i) | 2 \leq i \leq k\}$.

Teorema 3.12 *A relação de equivalência R retornada pela função $MINIMALBLOCK$ é a G-congruência gerada por $\{(\alpha_1, \alpha_i) | 2 \leq i \leq k\}$.*

Demonstração. Denotemos o valor final retornado da relação de equivalência $\sim$ por $\sim_F$, e a G-congruência gerada por $\{(\alpha_1,\alpha_i)|2 \leq i \leq k\}$ com $\equiv$. Lembremos primeiro que $R \subseteq \Omega \times \Omega$, tal que $\alpha \sim \beta$, é equivalente a dizer que $(\alpha,\beta) \in R$. Mostraremos que $\sim_F$ e $\equiv$ são equivalentes, primeiro vemos que em qualquer parte do algoritmo a relação $\sim$ implica $\equiv$, isto é que $u \sim v$ implica que $u \equiv v$ para qualquer $u,v \in \Omega$. Quando o algoritmo começa temos a única classe não trivial sendo $\{\alpha_i|1 \leq i \leq k\}$, assim isto é verdadeiro em cada uma das classes. Sempre que as duas classes são mescladas durante o curso do algoritmo, elas contém γ^x e δ^x sobre o elemento do grupo x de dois pontos γ e δ para os quais $\gamma \sim \delta$ antes de mesclar eles. Assumiremos indutivamente que $\gamma \equiv \delta$. Como $\equiv$ é a G-congruência, então $\gamma^x \equiv \delta^x$, e assim a propriedade "$\gamma \sim \delta \longrightarrow \gamma \equiv \delta$" permanece verdadeira após da fusão e portanto temos $\sim_F \subseteq \equiv$ verdadeiro o tempo todo.

Reciprocamente, para mostrar que $\equiv \subseteq \sim_F$, é suficiente ver que $\sim_F$ é uma G-congruência, porque o fato de conter os elementos da forma $\{(\alpha_1,\alpha_i)|2 \leq i \leq k\}$ já está implícito. Assim, vamos mostrar que se $u \sim_F v$ então $u^g \sim_F v^g$, para qualquer $g \in G$. Ao ser G um grupo finito, basta mostrar o fato para os geradores, isto é para os $x \in X$. Observe que sempre que um ponto λ é adicionado à fila (isto é sempre que um ponto λ é adicionado na lista q), nós temos que $REP(\lambda) = \lambda$ antes dele ser adicionado, e $REP(\lambda) \neq \lambda$ imediatante depois. É possível que $REP(\lambda)$ seja novamente redefinido mais tarde no procedimento, mas sempre será igual a um representante da classe existente, então nunca mais será igual a λ. Portanto, em todos os momentos λ estará na fila (na lista q) se e somente se $REP(\lambda) \neq \lambda$. O anterior argumento mostra que cada elemento é adicionado na lista q no máximo uma vez e, portanto, o procedimento deve terminar.

Seja l_F o comprimento da lista q no final do algoritmo. Para $\lambda \in [1..n]$, defina:

$$w(\lambda) = \begin{cases} k & \text{se } q[k] = \lambda \\ l_F + 1 & \text{se } \lambda \text{ não esta na lista } q. \end{cases}$$

Agora faremos indução sobre $z := 2l_F + 2 - w(u) - w(v)$. Se $z = 0$, então nem u nem v estão na lista q, e assim $REP(u) = u$ e $REP(v) = v$, portanto se $u \sim_F v$ então $u^x \sim_F v^x$, para todo $x \in X$. Por outro lado se $z > 0$, então pelo menos um dos u,v está na lista q, suponha que $u = q[k]$. Assim, ao chegar $i = k$ no ciclo principal do algoritmo e considerando a ação do elemento gerador x, temos $\gamma = u$ e $\delta = REP(\gamma) \neq \gamma$. Neste ponto, δ é um representante de uma classe e assim o δ não esta na lista q, isto é, $w(\delta) = l_F + 1$ e portanto $w(\delta) > w(\gamma)$. As classes de γ^x e δ^x são agora mescladas, e assim, $\gamma^x \sim_F \delta^x$. Como $w(\delta) > w(\gamma)$ temos $\delta^x \sim_F v^x$ pela hipótese de indução, e como $\gamma^x \sim_F \delta^x$, então $u^x \sim_F v^x$ e assim o teorema esta provado. $\square$

Agora vamos fazer a implementação do algoritmo 3.6 no sistema GAP. A função $MinimalBlock$ será um algoritmo $UNION-FIND$. Um algoritmo $UNION-FIND$ é

um processo que realiza duas operações na estrutura de dados:

Find : Determina a qual conjunto pertence um determinado elemento.

Union : Combina ou agrupa dois conjuntos em um único conjunto.

Nas seções 4.6 e 4.7 do livro [1] encontra-se uma análise detalhada da complexidade de tempo desta classe de algoritmos, por enquanto assumiremos, como no livro guia [15], que a complexidade de tempo do algoritmo $MinimalBlock$ é microscopicamente maior do que $O(kn)$. Para fazer a aplicação definimos primeiro as funções REP e $MERGE$ que serão as funções $UNION-FIND$.

Algoritmo 3.7: $REP(k,p)$

Input: $k \in [1..n]$, uma lista p

Output: O representante da classe que contém k

```
λ := k; ρ := p[λ];
while ρ ≠ λ do
    γ := rho; ρ := p[λ];
end
μ := k; ρ := p[μ];
while ρ ≠ λ do
    p[μ] := λ; μ := ρ; ρ := p[μ]
end
return λ;
```

Algoritmo 3.8: $MERGE(k,\lambda,c,p,q,l)$

Input: $k \in [1..n]$,$\lambda \in \Omega$, listas c,p,q e $l \in \mathbb{N}$

```
φ := REP(k,p); ψ := REP(λ,p);
if φ ≠ ψ then
    if c[φ] > c[ψ] then
        μ := φ; v := ψ
    else
        μ := ψ; v := φ
    end
    p[v] := μ; c[μ] := c[μ] + c[v];
    l := l + 1; q[l] := v;
end
```

Código 3.12 rep(k,p)

```
rep:=function(k,p)
local lambda, rho, mi;
lambda:=k; rho:=p[lambda];
while rho <> lambda do
  lambda := rho;
  rho := p[lambda];
od;
  mi := k;
  rho := p[mi];
while rho <> lambda do
  p[mi] := lambda;
  mi := rho;
  rho := p[mi];
od;
return lambda;
end;
```

Código 3.13 merge(k,λ,c,p,q,l)

```
merge:=function(k,lambda,c,p,q,l)
local f, psi, mi, ni;
f:=rep(k,p); psi:=rep(lambda,p);
if f<>psi then
  if c[f]>=c[psi] then
    mi := f;
    ni :=psi;
  else mi := psi;
    ni:= f;
  fi;
  p[ni]:=mi;
  c[mi]:=c[mi]+c[ni];
  l:=l+1;
  q[l]:=ni;
fi;
return [p,c,q,l];
end;
```

Algoritmo 3.9: $MINIMALBLOCK(G, \{\alpha_1, ..., \alpha_k\})$

Input: $G =< x >\leq Sym(n)$, G transitivo, $\alpha_1, ..., \alpha_k \in [1..n]$

Output: G-congruência $\sim$ gerada por $\{(\alpha_1, \alpha_i) | 2 \leq i \leq k\}$

```
1  for i ∈ [1..n] do
2  |   p[i] := i; c[i] := 1;
3  end
4  for i ∈ [1..k−1] do
5  |   p[α_{i+1} := α_1;
6  end
7  for i ∈ [1..k−1] do
8  |   q[i] := α_{i+1};
9  end
10 c[α_1] := k; i := 1; l := k−1;
11 while i ≤ l do
12 |   γ := q[i]; i := i+1;
13 |   for x ∈ X do
14 |   |   δ := REP(γ,p); MERGE(γ^x,δ^x,c,p,q,l);
15 |   end
16 end
17 for i ∈ [1..n] do
18 |   REP(i,p)
19 end
20 return p;
```

Analisaremos o que algoritmo 3.9 faz. Na linha 1 criamos as listas p e c. A lista p é aquela que vai retornar no final com a G-congruência gerada por $\{(\alpha_1, \alpha_i) | 2 \leq i \leq k\}$, ela vai começar com n elementos, onde na posição i está o elemento i, enquanto a lista c começa contendo o número 1 em todas suas entradas e vai contar quantos elementos tem cada classe. Na linha 4 criamos a classe que contém os elementos $\alpha_1, ..., \alpha_k$ na lista p, isto é feito colocando o elemento α_1 que é o representante nas posições α_i, para $i = 1, ..., k$. Na linha 7 criamos a lista q que irá conter os elementos, para os quais, vamos verificar o algoritmo, ela começa com o elemento α_2 na posição $q[1]$ e vai até o elemento α_k na posição $q[k-1]$ no momento da criação. Seguindo com o algoritmo vamos fazer $c[\alpha_1] = k$, segue que o α_1 é o representante da primeira classe com k elementos e assim começamos a executar a função para $i = 1, .., k-1$. Começamos deixando γ o elemento da posição i na lista q; logo mediante a função REP armazenamos em δ o representante da classe que contém γ e assim para cada $x \in X$ verificamos se os elementos γ^x e δ^x estão na mesma classe, caso não esteja, a função $MERGE$ mistura as classes deixando ser o representante da nova classe o representante da classe que contém mais elementos, para isso que foi

Código 3.14 minimalblock(x,α)

```
minimalblock:=function(x,alpha)
local i, i1, i2, i3, c, r, j,
k, n, p, q, l, ga, de, M;
n:=LargestMovedPoint(x);
r:=Length(x);
k:=Length(alpha);
p:=[]; c:=[]; q:=[]; M:=[];
for i1 in [1..n] do
 p[i1]:=i1;
 c[i1]:=1;
od;
for i2 in [1..k-1] do
 p[alpha[i2+1]]:=alpha[1];
od;
for i3 in [1..k-1] do
 q[i3]:=alpha[i3+1];
od;
c[alpha[1]]:=k;
i:=1;
l:=k-1;
while i<=l do
 ga:=q[i];
 i:=i+1;
 for j in [1..r] do
  de:=rep(ga,p);
  M:=merge(ga^x[j], de^x[j],c,p,q,l);
  p:=M[1];
  c:=M[2];
  q:=M[3];
  l:=M[4];
 od;
od;
for i in [1..n] do
 rep(i,p);
od;
return p;
end;
```

criada a lista c. O objetivo da linha 17 é garantir que a lista p cumpra $p[\alpha] = REP(\alpha)$ para todo α.

Exemplo 3.13 *No exemplo 3.11 encontramos que para* $x := \{(1,2,3,4,5,6),(2,6)(3,5)\}$ *e* $\alpha_1 = 1, \alpha_2 = 4$ *as classes de equivalência que definem a G-congruência são:* $\{1,4\},\{2,5\},\{3,6\}$. *Agora veremos como o GAP mostra este resultado numa lista de* 6 *elementos.*

```
gap> x:=[(1,2,3,4,5,6),(2,6)(3,5)];
[ (1,2,3,4,5,6), (2,6)(3,5) ]
gap> Read("rep");
gap> Read("merge");
gap> Read("minimalblock");
gap> minimalblock(x,[1,4]);
[ 1, 5, 3, 1, 5, 3 ]
```

Agora voltando para o objetivo desta seção que é verificar quando um grupo transitivo é também primitivo, o melhor, quando a ação natural do conjunto Ω no grupo $< X >$ sendo transitiva é também primitiva. Isto acontece quando a ação só admite sistemas de blocos triviais. Para fazer que o algoritmo verifique isso, executaremos a função $MINIMALBLOCK$ para as duplas $\{1,i\}$ com $2 \leq i \leq n$, isto quer dizer que nosso seguinte algoritmo vai repetir a função $MINIMALBLOCK$ $n-1$-vezes e assim o código *isprimitive* fica com complexidade $O(n^2)$.

Código 3.15 isprimitive(x)

```
isprimitive:=function(x)
local n, r, i, t, j, k;
n:=LargestMovedPoint(x);
r:=[];
for i in [2..n] do
  r[i-1]:=minimalblock(x,[1,i]);
od;
for k in [1..n-1] do
  for i in [1..n] do
    for j in [1..n] do
      if r[k][i]<>r[k][j] then
        return "falso";
      fi;
    od;
  od;
od;
for k in [1..n-1] do
  for i in [1..n] do
    for j in [1..n] do
      if r[k][i]=r[k][j] then
        return "verdadeiro";
      fi;
    od;
  od;
od;
end;
```

Exemplo 3.14 *Neste exemplo aplicaremos a função isprimitive para o grupo* $G =< X >$ *com* $X := \{(1,2,3,4,5,6),(2,6)(3,5)\}$ *do exemplo 3.11, que claramente não é primitivo, pois produz sistema de blocos não triviais. Também para o grupo simétrico gerado por* 4 *elementos e para o grupo alternante com* $n = 10$. *O GAP tem a ferramenta* $IsPrimitive(< G >)$ *que faz a mesma verificação.*

```
gap> Read("rep");
gap> Read("merge");
gap> Read("minimalblock");
gap> Read("isprimitive");
gap> x:=[(1,2,3,4,5,6),(2,6)(3,5)];;
gap> A:=AlternatingGroup(10);;
gap> A10:=GeneratorsOfGroup(A);
[ (1,2,3,4,5,6,7,8,9), (8,9,10) ]
gap> S4:=[(1,2,3,4),(1,2)];;
gap> isprimitive(x);
"falso"
gap> isprimitive(A10);
"verdadeiro"
gap> isprimitive(S4);
"verdadeiro"
gap> IsPrimitive(Group(x));
false
gap> IsPrimitive(Group(A10));
true
gap> IsPrimitive(Group(S4));
true
```

3.5 Algoritmo SCHREIER-SIMS

O nome do algoritmo $SCHREIER-SIMS$ é dado em homenagem aos matemáticos Otto Schreier e Charles Sims. Em 1970, Charles Sims introduziu o conceito de base para cálculos com grupos de permutação (ver [23, 24]). Da mesma forma que uma base em espaços vetoriais, os elementos do grupo possuem uma representação única relativa a ele. Uma vez executado, ele permite encontrar uma $BSGS$ para um grupo de permutações de ordem finita e com ela podemos fazer muitas tarefas, entre elas está, verificar se uma permutação dada está contida no grupo e encontrar a ordem do grupo.

Primeiramente, lembrando os conceitos dados na seção 1.5 vamos criar uma função que armazene os estabilizadores básicos, as órbitas básicas, e também os vetores de Schreier associados. Posteriormente, vamos supor que temos uma $BSGS$ e ver como o processo chamado $STRIP$ ajuda na adesão de permutações na construção do conjunto gerador forte, função que ajuda também para saber se uma permutação está contida em um grupo dado. Logo, encontraremos o algoritmo $SCHREIER-SIMS$, seu funcionamento e algumas aplicações dele. Finalmente, veremos como é possível fazer uma mudança de base.

Consideremos o grupo de permutação $G =< S >$, a lista $B := \{\beta_1, ..., \beta_k\}$ e para $1 \leq i \leq k+1$ sejam $G^{(i)}$, $S^{(i)}$, $H^{(i)}$, $\Delta^{(i)}$ e $U^{(i)}$ como foram definidos na seção 1.5. A seguinte função, que chamaremos *simsini*, vai construir a cadeia estabilizadora e retornar os seguintes dados: $S^{(i)}$, $H^{(i)}$, $\Delta^{(i)}$ e o vetor de Schreier para assim calcular os valores dos transversais $U^{(i)}$ para $1 \leq i \leq k+1$.

Código 3.16 simsini(B,S)

```
simsini:=function(B,S)
local SS,H,delta, k, i, G, j, U;
k:=Length(B); G:=[]; SS:=[];
H:=[]; delta:=[]; U:=[];
G[1]:=Group(S); SS[1]:=S;
H[1]:=Group(SS[1]);
delta[1]:=orbit(B[1],SS[1]);
U[1]:=ortransversal(B[1],SS[1])[2];
if k>=2 then
 G[2]:=stabilizer(B[1],S);
 SS[2]:=Intersection(S,stabilizer(B[1],S));
 H[2]:=Group(Union([()],SS[2]));
 delta[2]:=orbit(B[2],SS[2]);
 U[2]:=ortransversal(B[2],SS[2])[2];
fi;
if k>=3 then
 for i in [3..k] do
  G[i]:=stabilizer(B[i-1],
        GeneratorsOfGroup(G[i-1]));
  SS[i]:=Intersection(S,G[i]);
  H[i]:=Group(Union([()],SS[i]));
  delta[i]:=orbit(B[i],SS[i]);
  U[i]:=ortransversal(B[i],SS[i])[2];
 od;
fi;
G[k+1]:=stabilizer(B[k],
       GeneratorsOfGroup(G[k]));
SS[k+1]:=Intersection(S,G[k+1]);
H[k+1]:=Group(Union([()],SS[k+1]));
return [G,SS,H,delta,U];
end;
```

A função *simsini* nos permite saber se uma base e um conjunto de geradores de

um grupo G formam uma $BSGS$ para o grupo, a está aplicação vamos chamar de *isbsgs* e simplesmente verifica que $H^{(i)} = G^{(i)}$ para $1 \leq i \leq k+1$ e $G^{(k+1)} = 1_G = ()$.

Código 3.17 isbsgs(B,S)

```
isbsgs:=function(B,S)
local G, H, k;
k:=Length(B);
G:=simsini(B,S)[1];
H:=simsini(B,S)[3];
if G=H and G[k+1]=Group(()) then
  return "verdadeiro";
else
  return "falso";
fi;
end;
```

Exemplo 3.15 *No exemplo 1.54 vimos que uma possível $BSGS$ para os grupo simétrico S_n é dada por, $B := \{1,2,...,n-1\}$ e $S := \{(1,2),(2,3),...(n-1,n)\}$.*

```
gap> Read("orbit");
gap> Read("stabilizer");
gap> Read("orbitsv");
gap> Read("ubeta");
gap> Read("simsini");
gap> Read("isbsgs");
gap> B4:=[1,2,3];;
gap> S4:=[(1,2),(2,3),(3,4)];;
gap> B7:=[1,2,3,4,5,6];;
gap> S7:=[(1,2),(2,3),(3,4),(4,5),(5,6),(6,7)];;
gap> isbsgs(B4,S4);
"verdadeiro"
gap> isbsgs(B7,S7);
"verdadeiro"
```

A seguir, assumiremos que já temos uma $BSGS$ e seja Δ^* a lista que contém todos os $\Delta^{(i)}$. Verificaremos como usar os dados anteriores para verificar a adesão de uma permutação $g \in Sym(\Omega)$ a nossa lista S de geradores fortes do grupo de permutações G. Isto é realizado pela função $STRIP$.

Algoritmo 3.10: $STRIP(g, B, S, \Delta^*)$

Input: $g \in Sym(\Omega), B, S, \Delta^*$, como foram descritas acima
Output: $h \in Sym(\Omega)$ e $i \in [1..k+1]$

```
h := g;
for i ∈ [1..k] do
    β := β_i^h;
    if β ∉ Δ^(i) then
        return h, i;
    end
    Seja u_i ∈ U^(i) com β_i^{u_i} = β;
    h := h u_i^{-1};
end
return h,k+1;
```

Lema 3.16 *Suponha que (B, S) é uma BSGS para G. Aplicando a função $STRIP$ ao elemento g vai retornar 1_G se e somente se $g \in G$.*

Demonstração. Se o elemento h retornado da função $STRIP$ é o elemento identidade então $g = u_k u_{k-1} ... u_1$, donde $g \in G$. Reciprocamente, como tomamos (B, S) sendo uma $BSGS$ para G, então g terá uma única representação $g = u_k u_{k-1} ... u_1$ com $u_i \in U^{(i)}$ e assim a função $STRIP$ encontra os $u_i's$ e retornara no final o elemento 1_G. □

Se um inteiro $i \leq k$ é retornado, isto quer dizer que a função foi interrompida no momento de verificar se $\beta \in \Delta^{(i)}$, isto que vamos usar mais adiante para a construção do algoritmo $SCHREIER-SIMS$. Agora veremos a criação da função $STRIP$ no sistema GAP.

Lema 3.17 *Sejam B, S, $S^{(i)}$ e $H^{(i)}$ como definidas antes, então (B, S) é um BSGS para G se e somente se $H^{(k+1)} = 1_G$ e $H^{(i)}_{\beta_i} = H^{i+1}$ para $1 \leq i \leq k$.*

Demonstração. Suponha primeiro que (B, S) é uma $BSGS$ para G, por definição temos que $G^{(k+1)} = 1_G$ e $H^{(i)} = G^{(i)}$ para $1 \leq i \leq k$. Portanto temos $H^{(k+1)} = G^{(k+1)} = 1_G$ e $H^{(i+1)} = G^{(i+1)} = G^{(i)}_{\beta_i} = H^{(i)}_{\beta_i}$. Reciprocamente, suponha que $H^{(k+1)} = 1_G$ e $H^{(i)}_{\beta_i} = H^{i+1}$ para $1 \leq i \leq k$ e mostramos por indução sobre i que $H^{(i)} = G^{(i)}$ para $1 \leq i \leq k+1$. Para $i = 1$ isto é verdadeiro pela definição. Se for certo para i vemos que $H^{(i+1)} = H^{(i)}_{\beta_i} = G^{(i)}_{\beta_i} = G^{(i+1)}$. Além disso $G^{(k+1)} = H^{(k+1)} = 1_G$. □

O algoritmo $SCHREIER-SIMS$ toma uma sequência inicial B (que pode ser vazia) e uma sequência geradora S com $G =< S >$, e as estende até formar uma $BSGS$ para G. Começa estendendo B se é necessário para garantir que nenhum elemento fixa cada

Código 3.18 strip(g,B,S,Δ,U)

```
strip:= function(g,B,S,delta,U)
local h, k, beta, i, j, G;
k:=Length(B); G:=Group(S); h:=g;
for i in [1..k] do
  beta:=B[i]^h;
  if not beta in delta[i] then
    return [h,i];
  fi;
  for j in U[i] do
    if B[i]^j=beta then
      h:=h*Inverse(j);
    fi;
  od;
od;
return [h, k+1];,SS,vs
end;
```

ponto em B, e assim estabelece as órbitas básicas e os transversais associados mediante a função $SIMSINI$. Nesta primeira parte é claro que já temos a condição $H^{(k+1)} = 1_G$ e o algoritmo segue verificando as condições $H^{(i)}_{\beta_i} = H^{(i+1)}$ para $i = k, k-1, ..., 1$. Isto é feito na parte principal do algoritmo começando na linha 11. Como por definição já temos $H^{(i+1)} =< S \cap G^{(i+1)} > \subseteq < S \cap G^{(i)} >_{\beta_i} = H^{(i)}_{\beta_i}$, então só resta provar que $H^{(i)}_{\beta_i} \subseteq H^{(i+1)}$ o que fazemos verificando se os geradores de $H^{(i)}_{\beta_i}$ estão em $H^{(i+1)}$. Esses geradores de $H^{(i)}_{\beta_i}$ são encontrados usando o teorema de Schreier 1.44, como é descrito no algoritmo 3.2. Na implementação devemos ter cuidado de ao considerar aqueles geradores que são não triviais. Por isso é necessário fazer a verificação na ordem $i = k, k-1, ..., 1$, pois ao fazer isto, sabemos quando estamos verificando para um i em particular, se a condição é necessariamente satisfeita para o i maior, e assim podemos usar o algoritmo $STRIP$ para verificar se os geradores de $H^{(i)}_{\beta_i}$ estão em $H^{(i+1)}$.

Se o processo falhar para algum gerador de Schreier g, então tornaremos a variável y para falso, e tomamos um novo gerador h para $S^{(i+1)}$, substituindo assim $H^{(i+1)}$ para um subgrupo maior de $G^{(i+1)}$, o novo gerador h não é necessariamente o mesmo g, na realidade é o resultado de aplicar a função $STRIP$ ao elemento g. Se h fixa todos os pontos existentes na base, então anexamos a lista B um novo ponto dado por h, mantendo assim a condição $H^{(k+1)} = 1_G$ do lema 3.17. Para um i em particular no processo principal do algoritmo $SCHREIER-SIMS$, cada novo gerador que armazenamos em $S^{(i)}$ aumenta estritamente o grupo $H^{(i)}$, e como o grupo G é de ordem finita, então só podemos anexar geradores um número finito de vezes, e assim o processo deve terminar. Quando o algoritmo termina temos que todas as condições do lema 3.17 foram verificadas e assim (B, S) torna-se uma $BSGS$ para G.

Algoritmo 3.11: $SCHREIERSIMS(B,S)$

Input: B e S, como foram descritas acima

```
for x ∈ S do
    if x ∈ G_{β_1,...,β_k} then
        Seja γ ∈ Ω com γ^x ≠ γ; Add(B,γ); k = k+1;
    end
end
for i ∈ [1..k] do
    S^(i) := S ∩ G_{β_1,...,β_{i-1}}; H^(i) :=< S^(i) >; Δ(i) := β_i^{H^(i)};
end
i := k;
while i ≥ 1 do
    for β ∈ Δ^(i) do
        Encontra u_β ∈ H^(i) com β_i^{u_β} = β
        for x ∈ S^(i) do
            if u_β x ≢ u_{β^x} then
                y := true; h,j := STRIP(u_β x (u_{β^x})^{-1}, B, S, Δ*);
                if j ≤ k then
                    y := falso
                end
                else if h ≠ () then
                    y := falso;
                    Seja γ ∈ Ω com γ^h ≠ γ; Add(B,γ); k := k+1; S^(k) := [];
                if y=falso then
                    for l ∈ [i+1..j] do
                        Add(S^(l),h); H^(l) :=< S^(l) >; Δ^(l) := β_l^{H^(l)};
                    end
                    i := j; continue para i;
                end
            end
        end
    end
    i := i-1;
end
S := ⋃_{i=1}^k S^(i)
```

Exemplo 3.18 *Seja $n = 5$, $S = \{a,b\}$, com $a = (1,2,4,3)$ e $b = (1,2,5,4)$. Começamos com a base B sendo o conjunto vazio.*

1. *Primeiramente acrescentamos a nossa base B. Considerando o elemento a, temos que para $\gamma = 1$ cumpre-se que $\gamma^a \neq \gamma$, e assim $B = [1]$.*

2. *Considerando o único elemento de B fazemos $S^{(1)} = S$, $H^{(1)} = G$, $\Delta^{(1)} = \beta_1^G = [1,2,4,5,3]$ com o transversal sendo $U^{(1)} = [(),a,a^2,ab,a^3]$. Isto quer dizer que $\beta_1^{U^{(1)}[j]} = \Delta[j]$ para $j = 1,...,5$.*

3. *Começamos o processo maior para $i = 1$. Fazendo $\beta = 1$ que é nosso primeiro elemento de $\Delta^{(1)}$ temos que $u_\beta = u_1 = ()$, pois, $\beta_1^{()} = 1^{()} = 1 = \beta$. Assim, para:*

   3.1. *$x = a \in S^{(1)}$. Temos que $u_{\beta^x} = u_{1^a} = u_2 = a$ e assim $u_\beta * a = () * a = a = u_{\beta^x}$;*

   3.2. *$x = b \in S^{(2)}$. Temos que $u_{\beta^x} = u_{1^b} = u_2 = a$ e assim $u_\beta * b = () * b = b \neq a = u_{\beta^x}$. Logo fazemos $y := true$ e $h,j := STRIP(b * a^{-1},[1],[a,b],[\Delta^{(1)}])$; isto é, $h := (2,5)(3,4) = c$ e $j = 2$. Temos que $j = 2$ não é menor ou igual do que $i = 1$, mas se h é diferente do elemento neutro $()$, então tornamos a variável $y := false$ e fazemos $\gamma = 2$ (pois é o primeiro elemento de Ω tal que $\gamma^h \neq \gamma$). Então $B := [1,2]$, $k = 2$ e $S^{(2)} := []$. Como neste passo a variável y tornou-se falsa, temos $S^{(2)} := [c] = [(2,5)(3,4)]$, $H^{(2)} :=< S^{(2)} >$, $\Delta^{(2)} := \beta_2^{H^{(2)}} = 2^{H^{(2)}} = [2,5]$ e $U^{(2)} = [(),c]$. Agora começamos novamente o processo para $i = j = 2$.*

4. *Agora verificamos para $i = 2$. Para $\beta = 2 \in \Delta^{(2)}$ temos $u_\beta = u_2 = ()$ e assim para $x = c \in S^{(2)}$ obtemos $u_{\beta^x} = u_{2^c} = u_5 = c = () * c = u_\beta * x$. Portanto continuamos o processo para $i = i - 1 = 2 - 1 = 1$.*

5. *Lembremos que os nossos dados, até agora, são: $B = [1,2]$, $S^{(1)} = [a,b]$, $S^{(2)} = [c]$, $\Delta^{(1)} = [1,2,4,5,3]$, $\Delta^{(2)} = [2,5]$, $U^{(1)} = [(),a,a^2,ab,a^3]$ e $U^{(2)} = [(),c]$. Verificamos de novo para $i = 1$. Se tomamos $\beta = 2 \in \Delta^{(1)}$ vemos que o processo vai parar. De fato, pois teríamos $u_2 = a$ e assim para $x = a \in S^{(1)}$ obtemos $u_{\beta^x} = u_{2^a} = u_4 = a^2 = a * a = u_\beta * x$ o mesmo ocorre com $x = b \in S^{(1)}$, pois $u_{\beta^x} = u_{2^b} = u_5 = ab = a * b = u_\beta * x$. Agora para $\beta = 4 \in \Delta^{(1)}$ temos $u_4 = a^2$ e assim:*

   5.1. *Para $x = a \in S^{(1)}$. Temos que $u_{\beta^x} = u_{4^a} = u_3 = a^3 = a^2 * a = u_\beta * x$;*

   5.2. *Para $x = b \in S^{(1)}$. Temos que $u_{\beta^x} = u_{4^b} = u_1 = () \neq a^2 * b = u_\beta * x$. Logo fazemos $y := true$ e $h,j = STRIP(a^2b,[1,2],[a,b],[\Delta^{(1)},\Delta^{(2)}])$. $h = (2,3,5,4) = a^2b$ e $j = 2$. Como $j \leq k$ tornamos a variável y para falso e fazemos para $l = 2$: $S^{(2)} = [c,d] = [(2,5)(3,4),(2,3,5,4)]$, $H^{(2)} =< S^{(2)} >$, $\Delta^{(2)} = [2,5,3,4]$ e $U^{(2)} = [(),c,d,cd]$ com $cd = (2,4,5,3)$. O processo continua para $i = j = 2$.*

6. *Verificando para $i = 2$ o processo não vai mudar os dados, pois podemos comprovar que em todos os casos $u_{\beta^x} = u_\beta * x$. Assim o processo continuará para $i = i - 1 = 1$.*

7. *Para $i = 1$. Somente resta comprovar para os elementos 5 e 3. Seja $\beta = 5$, então $u_5 = ab$ e assim:*

7.1. Para $x = a \in S^{(1)}$. Como $u_{\beta^x} = u_{5^a} = u_5 = ab \neq ab * a = u_\beta * x$ então fazemos $y := true$ e assim $h, j = STRIP(aba * (ab)^{-1}, [1,2], [a,b], [\Delta^{(1)}, \Delta^{(2)}])$. Isto é, $h = (2,3,5,4) = d$ e $j = 2$, mas este elemento já está em $S^{(2)}$.

7.2. Para $x = b \in S^{(1)}$ podemos verificar que acontece o mesmo que no caso anterior, isto é que o elemento h retornado da função $STRIP$ já encontra-se na lista $S^{(2)}$.

Por último verificamos para $\beta = 3 \in \Delta^{(1)}$. Neste caso os geradores de schreier retornados da função STRIP, nos dois casos, é o elemento identidade $()$ e portanto o processo termina. Assim o algoritmo retornará a dupla (B,S), onde $B = [1,2]$ e $S = [a,b,c,d]$, formando uma BSGS para o grupo $G =< (a,b) >$.

Código 3.19 schreiersims(B,S)

```
schreiersims:=function(B,S)
local n, k, gam, gamma, SS, H, delta, U, i, beta, ub,
                    x, ubx, y, r, h, j, gamma2, l;
n:=LargestMovedPoint(S); k:=Length(B); gam:=[];
if Length(B)=0 then
 for x in S do
  for gamma in [1..n] do
   if gamma^x <> gamma then;
    Add(gam,gamma);
   fi;
  od;
 od;
 Add(B,gam[1]);
 k:=k+1;
fi;
SS:=simsini(B,S)[2]; H:=simsini(B,S)[3];
delta:=simsini(B,S)[4]; U:=simsini(B,S)[5];
i:=k;
while i>=1 do
 for beta in delta[i] do
  for ub in U[i] do
   for x in SS[i] do
    for ubx in U[i] do
     if B[i]^ub=beta and B[i]^ubx=beta^x then
      if ub*x<>ubx then
       y:=true; r:=ub*x*Inverse(ubx);
       h:=strip(r,B,S,delta,U)[1];
       j:=strip(r,B,S,delta,U)[2];
       if j<=k then
        y:=false;
       elif h<>() then
        y:=false;
        for gamma2 in [1..n] do
         if gamma2^h <> gamma2 then;
          if not gamma2 in B then
           Add(B,gamma2);
           k:=k+1; SS[k]:=[];
           break;
          fi;
         fi;
        od;
       fi;
       if y=false then
        for l in [i+1..j] do
         if not h in SS[l] then
          Add(SS[l],h);
          H[l]:=Group(SS[l]);
          delta[l]:=orbit(B[l],SS[l]);
          U[l]:=ortransversal(B[l],SS[l])[2];
         fi;
        od;
        i:=j;
        continue;
        break;
       fi;
      fi;
     fi;
    od;
   od;
  od;
 od;
 i:=i-1;
od;
S:=Union(SS);
return [B,S];
end;
```

Exemplo 3.19 *O resultado do GAP para o exemplo 3.18 é:*

```
gap> Read("orbit");
```

```
gap> Read("stabilizer");
gap> Read("ortransversal");
gap> Read("strip");
gap> Read("simsini");
gap> Read("schreiersims");
gap> schreiersims([],[(1,2,4,3),(1,2,5,4)]);
[ [ 1, 2 ], [ (2,3,5,4), (2,5)(3,4), (1,2,4,3), (1,2,5,4) ] ]
```

A complexidade do tempo para o algoritmo está dividida em dois casos, o primeiro é calculando direito nos elementos dos transversais $U^{(i)}$ e o segundo é fazendo uso dos vetores de Schreier. O tempo de execução de um algoritmo que armazena os transversais $U^{(i)}$ foi mostrado por Furst, Hopcroft Luks no artigo [9] do ano 1980, para ser $O(n^6+kn^2)$. No ano 1982 o Mark Jerrum [17] fez uma variante do algoritmo, dando como resultado um tempo de execução $O(n^5+kn^2)$, outra versão do algoritmo com o mesmo tempo de complexidade foi feita pelo Knuth no artigo [18]. Todos os resultados anteriores podem ser encontrados no capítulo 14 de [7] e o capítulo 4 de [21] contém os detalhes das diferentes variações do algoritmo.

3.5.1 Algumas aplicações

Pertença de um elemento em um grupo

Vamos fazer uso da função 3.10 junto com o algoritmo 3.11 para saber se um elemento pertence a um grupo dado. Tendo uma $BSGS$ é só verificar se o elemento h que retorna da função $STRIP$ é a permutação identidade.

Código 3.20 belong(g,x)

```
belong:=function(g,x)
local B, S, delta, U, t;
B:=schreiersims([],x)[1];
S:=schreiersims([],x)[2];
delta:=simsini(B,S)[4];
U:=simsini(B,S)[5];
t:=strip(g,B,S,delta,U)[1];
if t=() then
        return "verdadeiro";
else
        return "falso";
fi;
end;
```

Representação única de um elemento

Na seção 1.5 vimos que se (B, S) é uma $BSGS$ então cada elemento de G tem uma única representação nos elementos dos transversais $U^{(i)}$. A seguinte função que chamamos de $factor$ utiliza o procedimento do algoritmo $STRIP$ para criar essa representação única de cada elemento. Ela vai tomar um elemento $g \in < x >$ e retornar um vetor $T := [u_1..u_k]$ com cada um dos fatores em que o elemento é descomposto, assim $g := u_k * u_{k-1} * ... * u_1$.

Código 3.21 factor(g,x)

```
factor:= function(g,x)
local B, S, h, k, beta, i, j, s, delta, U, T;
B:=schreiersims([],x)[1];
S:=schreiersims([],x)[2];
k:=Length(B);
delta:=simsini(B,S)[4];
U:=simsini(B,S)[5];
T:=[];
h:=g;
for i in [1..k] do
  beta:=B[i]^h;
  if not beta in delta[i] then
    return [h,i];
  fi;
  for j in U[i] do
    if B[i]^j=beta then
      Add(T,j);
      h:=h*Inverse(j);
    fi;
  od;
od;
return T;
end;
```

Ordem de um grupo

Uma das aplicações mais importantes do algoritmo $SCHREIER-SIMS$ resulta em encontrar a ordem de um grupo de permutação finita. Como cada elemento $g \in G$ pode ser representado de maneira única como o produto $g := u_k u_{k-1} ... u_1$, onde $u_i \in U^{(i)}$ e $|U^{(i)}| = |\Delta(i)|$, então $|G| = \prod_{i=n}^{k} |\Delta^{(i)}|$.

Código 3.22 order(x)

```
order:=function(x)
local B, S, delta, ord, i;
B:=schreiersims([],x)[1];
S:=schreiersims([],x)[2];
delta:=simsini(B,S)[4];
ord:=1;
for i in [1..Length(B)] do
  ord:=ord*Length(delta[i]);
od;
return ord;
end;
```

3.5.2 Mudança de base

Uma das operações que podemos fazer num espaço vetorial quando temos uma base é fazer uma mudança de base, isto geralmente é feito para simplificar alguns problemas. Assim como pode ser feito para os espaços vetoriais, em [23] e [24], Charles Sims fala de algumas maneiras para encontrar uma nova base a partir de uma $BSGS$. Um dos problemas que pode ser resolvido mudando a base é o cálculo do estabilizador pontual (ver 1.17) de alguns pontos $\alpha_1, ...\alpha_r \in \{1, ..., n\}$, então a melhor maneira é mudar a base de G para começar com a seqüencia $\alpha_1, ..., \alpha_r$ e então o estabilizador necessário é justamente $G^{(r+1)}$.

Para fazer uma mudança de base arbitraria, segundo [16] uma maneira fácil é a seguinte, por exemplo queremos trocar o elemento β_i da base pelo elemento α, então procedemos assim:

- Vamos buscar um elemento $g \in G$ tal que $\beta_i = \alpha$, então a imagem base de g relativa a B será também uma base, isto é, $B^g = \{\beta_1^g, ..., \beta_k^g\}$.
- Vamos conjugar a cadeia de geradores fortes pelo elemento g, isto é, $g^{-1}Sg$.

Código 3.23 baseconjugation(B,S,g)

```
baseconjugation:=function(B,S,g)
local Bc, Sc, i, k;
k:=Length(B);
Bc:=[];
Sc:=[];
for i in [1..k] do
        Bc[i]:=B[i]^g;
od;
Sc:=Inverse(g)*S*g;
return [Bc,Sc];
end;
```

O processo "$BASESWAP$" feito por Charles Sims nos artigos [[23],[23]] faz uma mudança de base trocando dois pontos adjacentes β_i, β_{i+1} de uma base existente (B, S).

Seja $B = \{\beta_1, ..., \beta_k\}$, quando trocamos dois pontos adjacentes β_i, β_{i+1}, a nova base B passa a ser B', onde $B' = \{\beta_1, ..., \beta_{i-1}, \beta_{i+1}, \beta_i, ..., \beta_k\}$ e na cadeia estabilizadora veremos que o único estabilizador que muda é $G^{(i+1)} = G_{\beta_1, ...\beta_i}$, que passa agora a ser $G_{\beta_1, ...\beta_{i-1}\beta_{i+1}} = G^{(i)}_{\beta_{i+1}}$. A ideia do algoritmo é adicionar na lista S uma nova lista T, tal que $(B', S \cap T)$ forma uma $BSGS$ para o grupo. Assim é suficiente mostrar que $< T >= G^{(i)}_{\beta_{i+1}} := H$. Lembremos que (B, S) é uma $BSGS$ e assim $H^{(i)} = G^{(i)} =< S^{(i)} >$ para $1 \leq i \leq k+1$, daí começamos com $T := S^{(i+2)}$, pois $< S^{(i+2)} >$ estabiliza todos os β_j

com $1 \leq j \leq i+1$, e assim nesta lista somente vamos adicionar um elemento yx que fixe o β_{i+1}, tal que $x, y \in G^{(i)}$.

Aplicando o teorema 1.16 da **Órbita-Estabilizador** temos que

$$|G^{(i)}| = |\beta_{i+1}^{G^{(i)}}||G^{(i)}_{\beta_{i+1}}|. \tag{3-1}$$

$$\begin{aligned} |G^{(i)}| &= |\beta_i^{G^{(i)}}||G^{(i)}_{\beta_i}| = |\beta_i^{G^{(i)}}||G_{\beta^{(i+1)}}| \\ &= |\beta_i^{G^{(i)}}||\beta_{i+1}^{G^{(i+1)}}||G^{(i+1)}_{\beta_{i+1}}| \\ &= |\beta_i^{G^{(i)}}||\beta_{i+1}^{G^{(i+1)}}||G^{(i+2)}|. \end{aligned} \tag{3-2}$$

Juntando as equações (3-1) e (3-2) obtemos:

$$|G^{(i)}_{\beta_{i+1}}| = \frac{|\beta_i^{G^{(i)}}||\beta_{i+1}^{G^{(i+1)}}|}{|G^{(i)}_{\beta_{i+1}}|}|G^{(i+2)}|. \tag{3-3}$$

Assim $|H| = s|G^{(i+2)}|$, onde $s := \frac{|\beta_i^{G^{(i)}}||\beta_{i+1}^{G^{(i+1)}}|}{|G^{(i)}_{\beta_{i+1}}|}$. Agora, como $H = G^{(i)}_{\beta_{i+1}}$ e $G^{(i+2)} = G^{i+1}_{\beta_{i+1}} = G^{(i)}_{\beta_i,\beta_{i+1}} = H_{\beta_i}$, então aplicando, novamente o Teorema da **Órbita-estabilizador** temos que $s = |\beta_i^H|$. Note que s é o comprimento da $(i+1)$-ésima órbita básica antes da mudança de base.

O algoritmo

O algoritmo $BASESWAP$ começa definindo o valor de s, inicializando a lista T e na lista Γ colocamos os elementos da i-ésima órbita básica sem os pontos β_i, β_{i+1}. O "while"principal continuará até que tenhamos armazenado os elementos suficientes em T para que $< T >= H$, e para que isto ocorra tem-se que cumprir $|\beta_i^{<T>}| = s$.

Pelo teorema da **Órbita-Estabilizador** $|G^{(i)}| = |\beta_i^{G^{(i)}}||G^{(i)}_{beta_i}|$, assim um elemento z de $G^{(i)}$ pode ser escrito como $z = yx$, onde $g \in G^{(i+1)}$ e $x \in U^{(i)}$. Logo, dado um elemento $x \in U^{(i)}$, existirá tal elemento $yx \in H$ se e somente se existe $y \in G^{(i+1)}$ com $\beta_{i+1}^{yx} = \beta_{i+1}$, isto é, se e somente se, $\beta_{i+1}^{x^{-1}} \in \beta_{i+1}^{G^{(i+1)}} = \Delta^{(i+1)}$. Vemos que somente um elemento cumpre esta condição, pois, se y_1x e y_2x são dois elementos de H desta forma, então, $(\beta_{i+1}^{y_ix})^{(y_2x)^{-1}} = \beta_{i+1}$ e assim $(y_1x)(y_2x)^{-1} = y_1xx^{-1}y_2^{-1} = y_1y_2^{-1} \in G^{(i+2)} \subseteq < T >$. Então só precisamos colocar um desses elementos. Agora vamos remover os elementos de $\gamma^{<T>}$ da lista Γ, pois se não houver nenhum elemento $yx \in H$ para algum x, então também não pode haver elemento da forma yxt, para qualquer $t \in < T >$. Por outro lado, também vamos remover de Γ os elementos de $\beta_i^{<T>}$, pois não vamos anexar um elemento yx para o qual $\beta_i^{yx} = \beta_i^t$ para algum t que já existe em $< T >$.

Algoritmo 3.12: $BASESWAP(B, S, \Delta^*, i)$

Input: Uma $BSGS$ (B, S) para $G \leq Sym(n)$, $i \in \{1, ..., |B| - 1\}$

1 $s := \frac{|\beta_i^{G^{(i)}}||\beta_{i+1}^{G^{(i+1)}}|}{|G^{(i)}_{\beta_{i+1}}|}$; $T := S^{(i+2)}$; $\Gamma := \Delta(i) - \{\beta_i, \beta_{i+1}\}$;

2 **while** $|\beta_{i+1}^{<T>}| \neq s$ **do**

3 Escolha $\gamma \in \Gamma$, $x \in U^{(i)}$ tal que $\gamma = \beta_i^x$;

4 **if** $\beta_{i+1}^{x^{-1}} \notin \Delta^{(i+1)}$ **then**

5 $\Gamma := \Gamma - \gamma^{<T>}$;

6 **else**

7 Encontra $y \in G^{(i+1)}$ com $\beta_{i+1}^y = \beta_{i+1}^{x^{-1}}$;

8 **if** $\beta_i^{yx} \notin \beta_i^{<T>}$ **then**

9 $T := T \cup \{yx\}$; $\Gamma := \Gamma - \beta_i^{<T>}$;

10 **end**

11 **end**

12 **end**

13 $S := S \cup T$;

14 Troca β_i e β_{i+1} e Recalcula $\Delta^{(i)}$, $\Delta^{(i+1)}$ e os vetores de Schreier;

Código 3.24 baseswap(B,S,i)

```
baseswap:=function(B,S,i)
local G, SS, delta, U, s, gam, gamma,
        x, y, T, sw1,sw2;
G:=simsini(B,S)[1];
SS:=simsini(B,S)[2];
delta:=simsini(B,S)[4];
U:=simsini(B,S)[5];
s:=Length(delta[i])*Length(delta[i+1])/
Length(orbit(B[i+1],GeneratorsOfGroup(G[i])));
T:=Union([()],SS[i+2]);
gamma:=Difference(delta[i],[B[i],B[i+1]]);
while Length(orbit(B[i],T))<>s do
 for gam in gamma do
  for x in U[i] do
   if gam=B[i]^x then
    if not B[i+1]^Inverse(x) in delta[i+1] then
     gamma:=Difference(gamma,orbit(gam,T));
    else
     for y in G[i+1] do
      if B[i+1]^y=B[i+1]^Inverse(x) then
       if not B[i]^(y*x) in orbit(B[i],T) then
        T:=Union(T,[y*x]);
        gamma:=Difference(gamma,orbit(B[i],T));
       fi;
      fi;
     od;
    fi;
   od;
  od;
 od;
S:=Union(S,T);
sw1:=B[i];
sw2:=B[i+1];
B[i]:=sw2;
B[i+1]:=sw1;
return [B,S];
end;
```

Exemplo 3.20 *Neste exemplo utilizaremos primeiro o algoritmo $SCHREIER-SIMS$ para encontrar uma $BSGS$ para o grupo simétrico de grau 6, logo mudaremos a base obtida, pelo algoritmo $BASESWAP$, e uma outra base pela conjugação, assim, vemos que as três bases, embora são $BSGS$ para o grupo simétrico, elas são bem diferentes. Depois vemos como é que funciona para um outro grupo qualquer.*

```
gap> Read("orbit");
gap> Read("stabilizer");
gap> Read("ortransversal");
gap> Read("strip");
gap> Read("simsini");
gap> Read("schreiersims");
gap> Read("baseswap");
gap> x1:=[(1,2,3,4,5,6),(1,2)];;
gap> s1:=schreiersims([],x1);
[ [ 1, 2, 5, 3, 4 ], [ (5,6), (4,6), (3,4,6), (2,6,5,4,3), (1,2), (1,2,3,4,5,6) ] ]
gap> b1:=baseswap(s1[1],s1[2],3);
[ [ 1, 2, 3, 5, 4 ], [ (), (5,6), (4,6,5), (4,6), (3,4,6), (2,6,5,4,3), (1,2), (1,2,3,4,5,6) ] ]
gap> bc:=baseconjugation(s1[1],s1[2],(1,6)(2,4,5,3));
 [ [ 6, 4, 3, 2, 5 ], [ (1,3), (1,5), (1,2,5), (1,3,5,2,4), (4,6), (1,6,4,2,5,3) ] ]
gap> x2:=[(1,2,6,7),(1,2)(5,6)];;
gap> s2:=schreiersims([],x2);
[ [ 1, 2 ], [ (2,7,6,5), (1,2)(5,6), (1,2,6,7) ] ]
gap> b2:=baseswap(s2[1],s2[2],1);
[ [ 2, 1 ], [ (), (2,7,6,5), (1,2)(5,6), (1,2,6,7), (1,5)(6,7), (1,6,5,7) ] ]
gap> Read("isbsgs");
gap> isbsgs(bc[1],bc[2]);
 "verdadeiro"
gap> isbsgs(b1[1],b1[2]);
"verdadeiro"
gap> isbsgs(b2[1],b2[2]);
"verdadeiro"
```

3.6 Homomorfismos de grupos

Um dos aspectos importantes da teoria de grupos computacional é a capacidade de fazer cálculos eficientemente com homomorfismos de grupos, entre outras coisas, esses cálculos permite-nos transladar de uma apresentação de um grupo para uma outra, possivelmente mais conveniente. Cada um desses homomorfismos é induzido pela ação de um grupo G num conjunto finito X, pois como vimos no exemplo 1.10, a ação da translação à direita nos permitiu passar de um grupo finitamente apresentado para um

grupo de permutação.

Para um homomorfismo de grupos $\varphi : G \longrightarrow H$, gostaríamos de executar as seguintes operações:

1. Calcular a imagem $\varphi(g)$ de qualquer $g \in G$.
2. para $h \in H$, verificar quando $h \in Im(\varphi)$ e, se possível, encontrar um $g \in G$ com $\varphi(g) = h$.
3. Encontrar $\varphi(K)$, para algum $K \leq G$.
4. Encontrar $\varphi^{-1}(K)$, para algum $K \leq H$.
5. Achar o $Ker(\varphi)$.

Com ajuda do livro guia [15] e do artigo [6] apresentaremos alguns métodos computacionais que nos ajudará no calculo das tarefas anteriores. Na pratica tem dois métodos comuns de definir um homomorfismo, o primeiro deles é usando uma regra para calcular $\varphi(g)$ para $g \in G$ e o segundo é especificar as imagens $\varphi(x) \in H$ para x num conjunto gerador finito X de G.

Primeiramente apresentaremos a maneira de encontrar a imagem e o kernel de uma aplicação que é também um homomorfismo. Se conhecemos uma apresentação $\langle X|R\rangle$ de G sobre X, então o teorema 1.42 da uma forma simples de verificar isto. Tendo em conta que um relator de uma grupo $G =< X >$ é uma palavra $w \in A^*$ com $w =_G 1$, onde $A = X \cup X^{-1}$. Assumindo que a aplicação $\varphi : G \longrightarrow H$ é estendida, e então avaliamos suas imagens em H sobre cada relator do grupo em R. Se esta avaliação da 1, então φ estende para um homomorfismo.

Exemplo 3.21 *Segundo o teorema 1.41, uma forma de apresentar o grupo simétrico S_4 é dada por: $\langle [x_1,x_2,x_3] | [x_1^2,x_2^2,x_3^2,(x_1*x_2)^3,(x_2*x_3)^3,(x_3*x_1)^2] \rangle$, onde as permutações x_i podem ser vistas como transposições $(i,i+1)$. Com ajuda do GAP verificaremos que a função $\varphi : S_4 \longrightarrow S_3$, estende para um homomorfismo, onde $\varphi(x) = (1,2)$, se a permutação x é impar e $\varphi(x) = ()$, se a permutação x é par, isto é, que a aplicação avaliada nos relatores de G produz a permutação identidade.*

```
gap> f:=FreeGroup("x_1","x_2","x_3");
<free group on the generators [ x_1, x_2, x_3 ]>
gap> s4:=f/[f.1^2,f.2^2,f.3^2,(f.1*f.2)^3,(f.2*f.3)^3,(f.3*f.1)^2];
<fp group on the generators [ x_1, x_2, x_3 ]>
gap> RelatorsOfFpGroup(s4);
[ x_1^2, x_2^2, x_3^2, (x_1*x_2)^3, (x_2*x_3)^3, (x_3*x_1)^2 ]
gap> x_1:=(1,2);;
```

```
gap> x_2:=(2,3);;
gap> x_3:=(3,4);;
gap> phi:=function(x)
> if SignPerm(x)=-1 then return (1,2); else return (); fi; end;
function( x ) ... end
gap> [phi( x_1^2),phi( x_2^2),phi( x_3^2),phi( (x_1*x_2)^3),
> phi( (x_2*x_3)^3),phi( (x_3*x_1)^2) ];
[ (), (), (), (), (), () ]
```

A função $IMAGEKERNEL$ é um método geral muito útil que nos permite encontrar a $Im(\varphi)$ e o $Ker(\varphi)$, para um homomorfismo de grupos $\varphi : G \longrightarrow H$. O loop maior(While) dela está fundamentado pelo **primeiro teorema de isomorfismo**, assim ele vai parar somente até ter que o produto das ordens da imagem e do kernel seja igual à ordem do grupo. É claro que ela funciona, pois, se g é um elemento aleatório em G, então $\varphi(g)$ será, novamente, de novo um elemento aleatório da $Im(\varphi)$; além disso, precisamos que gk^{-1} seja um elemento aleatório do $Ker(\varphi)$. Usualmente uma função que armazene as imagens do homomorfismo φ produzirá a mesma imagem inversa de um elemento de $Im(\varphi)$, então k depende só de h, e assim gk^{-1} será aleatório no $Ker(\varphi)$.

Algoritmo 3.13: $IMAGEKERNEL(\varphi, x)$

Input: $G =< x >$, homomorfismo de grupos $\varphi : G \longrightarrow H$

Output: Geradores para $Kernel$ e $Imagem$

```
Y := []; Z := []; χ := PRINITIALIZE(x, 10, 20);
while |<Y>||<Z>| < |G| do
    g := PRRANDOM(χ); h := φ(g);
    if h ∉ <Z> then
        Add(Z, h)
    else
        k := φ^{-1}(h);
        if gk^{-1} ∉ <Y> then
            Add(Y, gk^{-1});
        end
    end
end
return Y, Z
```

Exemplo 3.22 *Na pagina 556 do manual do GAP [11], temos uma ferramenta que cria um homomorfismo de grupos partindo de uma função. Utilizaremos esta ferramenta,*

Código 3.25 imagekernel(ϕ,x)

```
imagekernel:=function(phi,x)
local Y, Z, xx, g, h, k, r;
Y:=[]; Z:=[]; Add(Y,()); Add(Z,());
xx:=prinitialize(x,10,20);
while Order(Group(Y))*Order(Group(Z))<Order(Group(x)) do
  g:=prrandom(xx);
  h:=phi(g);
  if not h in Group(Z) then
    Add(Z,h);
  else
    for r in Group(Z) do
      if phi(r)=h then
        k:=r;
        if not g*Inverse(k) in Group(Y) then
          Add(Y, g*Inverse(k));
        fi;
      fi;
    od;
  fi;
od;
return [Y,Z];
end;
```

juntamente com o exemplo 3.21, para ver que a função imagekernel produz o mesmo resultado que as ferramentas Kernel(hom) e Imagem(hom) do GAP.

```
gap> Read("prrandom");
gap> Read("prinitialize");
gap> Read("imagekernel");
gap> s3:=SymmetricGroup(3);
Sym( [ 1 .. 3 ] )
gap> s4:=SymmetricGroup(4);
Sym( [ 1 .. 4 ] )
gap> phi:=function(x)
> if SignPerm(x)=-1 then return (1,2); else return (); fi; end;
function( x ) ... end
gap> imagekernel(phi,[(1,2),(2,3),(3,4)]);
[ [ (), (1,2)(3,4), (1,3,4) ], [ (), (1,2) ] ]
gap> hom:=GroupHomomorphismByFunction(s4,s3,phi);
MappingByFunction( Sym( [ 1 .. 4 ] ), Sym( [ 1 .. 3 ] ), function( x ) ... end )
gap> Image(hom);
Group([ (1,2), (1,2) ])
gap> Kernel(hom);
```

```
Group([ (2,3,4), (1,2,4) ])
gap> t:=Group([ (), (1,2)(3,4), (1,3,4) ]);
Group([ (), (1,2)(3,4), (1,3,4) ])
gap> tt:=Kernel(hom);
Group([ (2,3,4), (1,2,4) ])
gap> t=tt;
true
```

Quando se tem um grupo $G =< X >$ é possível encontrar uma $BSGS$ (B,S) para ele usando a função $schreiersims$, mas temos que ter cuidado na hora de armazenar os elementos SLP que expressam os geradores fortes em S, em termos dos geradores originais X de G, podemos definir um homomorfismo de grupos $\varphi : G \longrightarrow H$ especificando as imagens $\varphi(x)$, para $x \in X$. Para as imagens $\varphi(x)$, para $x \in S - X$, pode então ser calculado suas expressões SLP (ver 2.1). Agora, nossa função 3.21 utiliza o algoritmo $STRIP$ para expressar um elemento arbitrário $g \in G$ como uma palavra nos geradores fortes, e assim $\varphi(g)$ pode ser calculado. Neste momento não temos um algoritmo de analise de reescrita de membros para subgrupos de H, portanto ainda não temos as ferramentas necessárias para encontrar a imagem ou imagem inversa de um subgrupo de G ou H, respetivamente. Além disso, se tivermos uma apresentação de G sobre S, então poderemos verificar quando as imagens $\varphi(x)$, para $x \in X$, realmente definem um homomorfismo $\varphi : G \longrightarrow H$.

Existem, no entanto, duas situações especificas e frequentes em que podemos realizar algumas dessas tarefas de forma mais eficientes do que podemos pelos métodos gerais. Esses são a ação induzida de G sobre uma união não vazia de órbitas de G^{Ω} e a ação induzida de G sobre um sistema de blocos preservado por G^{Ω}.

3.6.1 Ação induzida sob uma união de órbitas

Nesta seção utilizaremos como referencia, além do livro guia, o artigo [6]. Seja ψ uma união não vazia de órbitas de uma ação sobre o conjunto Ω, isto $\psi \subseteq \Omega$. Para a computação com a ação da órbita induzida $\varphi : G \longrightarrow G^{\psi}$ primeiro renomeamos os elementos de ψ como $1, 2, ...,$, de modo que $Im(\varphi) \leq Sym(|\psi|)$, isto é, se $\bar{\alpha}, \bar{\beta}$ são a renumeração de $\alpha, \beta \in \psi$, então o homomorfismo φ fica definido como segue

$$\begin{array}{rcl} \varphi : G & \longrightarrow & Sym(|\psi|) \\ g & \longmapsto & \pi : \begin{array}{rcl} \psi & \longrightarrow & \psi \\ \bar{\alpha} & \longmapsto & \bar{\alpha}^{g} = \bar{\beta} \Leftrightarrow \alpha^{g} = \beta. \end{array} \end{array}$$

Exemplo 3.23 *Seja* $G =< X >$, *com* $X = \{(1,2,4,3),(1,2,5,4)\}$, *atuando sobre o mesmo grupo* G *baixo a ação da conjugação, e seja* $\psi := (2,3,5,4)^{G} \cup (2,4,5,3)^{G}$,

onde $(2,3,5,4)^G = [(2,3,5,4),(1,4,3,2),(1,2,4,3),(1,5,3,4),(1,3,2,5)]$ *e* $(2,4,5,3)^G = [(2,4,5,3),(1,2,5,4),(1,3,4,2),(1,4,3,5),(1,5,2,3)]$. *Assim, renomeando os elementos de* ψ, *de modo que* $(2,3,5,4) := 1, (1,4,3,2) := 2, ..., (1,5,2,3) := 10$, *teremos, por exemplo, que*

$$\begin{array}{rccc} \varphi: & G & \longrightarrow & Sym(|\psi|) = Sym(10) \\ & (1,2,3,4,5) & \longmapsto & (1,4,2,5,3)(6,9,7,10,8). \end{array}$$

Neste caso temos que $Im(\varphi) \leq Sym(10)$.

Na continuação, mudamos de base para ter tantos pontos da base quanto possível em ψ. Mais precisamente, mudamos para uma base $[\beta_1, ..., \beta_k]$ em que, para alguns $j, \beta_i \in \psi$ se e somente se $1 \leq i \leq j$, e o $(j+1)$-ésimo estabilizador básico $G^{(j+1)}$ fixe todos os pontos de ψ. Então, temos $G^{(j+1)} = ker(\psi)$.

As imagens $\varphi(g)$ dos geradores fortes de G que não estão em $G^{(j+1)}$ formam um grupo gerador forte para $Im(\varphi)$ com respeito à base $[\varphi(\beta_1), ..., \varphi(\beta_j)]$. Podemos, portanto, testar permutações para que pertença a $Im(\varphi)$.

Exemplo 3.24 *No exemplo 3.21 temos que uma BSGS para* G *é dada por* $B = [1,2]$ *com* $S = [(2,3,5,4),(2,5)(3,4),(1,2,4,3),(1,2,5,4)]$, *assim* $Ker(\varphi) = [()]$ *e para calcular a imagem basta somente avaliar o homomorfismo nos geradores fortes de* G, *isto é,*

$$\begin{array}{rccc} \varphi: & G & \longrightarrow & Sym(10) \\ & (2,3,5,4) & \longmapsto & (2,3,5,4)(7,8,10,9). \\ & (2,5)(3,4) & \longmapsto & (2,5)(3,4)(7,10)(9,8). \\ & (1,2,4,3) & \longmapsto & (1,5,3,4)(6,9,10,7). \\ & (1,2,5,4) & \longmapsto & (1,5,2,3)(6,9,8,10). \end{array}$$

E assim, uma BSGS para $Im(\varphi)$ *é dada por* (B', S'), *com* $B' = [2,1]$ *e* $S' = [(2,3,5,4)(7,8,10,9),(2,5)(3,4)(7,10)(8,9),(1,5,3,4)(6,9,10,7),(1,5,2,3)(6,9,8,10)]$.

Para calcular uma imagem inversa $\varphi^{-1}(h)$, para $h \in Im(\varphi)$, podemos usar a função *strip* na imagem de φ para expressar h como uma palavra nos geradores fortes de $Im(\varphi)$, mas como esses geradores fortes são apenas $\varphi(g)$, para $g \in S$, a mesma palavra definida sobre S irá avaliar para $g \in G$ com $\varphi(g) = h$. Podemos, portanto, realizar todos os tipos de cálculos desejados com φ.

3.6.2 Ação induzida sob um sistema de blocos

Agora veremos como funciona a ação induzida de um grupo transitivo G^Ω sob um sistema de blocos Γ preservado por G^Ω, onde $\Omega = \{1, ..., n\}$. Se $\Gamma = \{\Gamma_1, ..., \Gamma_m\}$ é um sistema de blocos para G, o homomorfismo de blocos enviará cada $g \in G$ numa permutação induzida sobre Γ da seguinte maneira:

$$\begin{array}{rcl} \varphi: G & \longrightarrow & Sym(\{1,...,m\}) \\ g & \longmapsto & \pi: \quad \{1,...,m\} \longrightarrow \{1,...,m\} \\ & & \qquad\quad \Gamma_i \longmapsto \Gamma_i^g. \end{array}$$

Utilizando a função *minimalblock* 3.14, podemos definir um sistema de blocos mediante uma função de partição associada p, para a qual, $p[\alpha] \in \Omega$ é o ponto que representa o bloco contendo α, então $p[p[\alpha]] = p[\alpha]$, para todo $\alpha \in \Omega$. Primeiro vamos criar uma função que enumere os blocos, $1,...,m$, onde m é o número de blocos. Isto será realizado mediante a seguinte função $NUMBERBLOCKS$, que retorna o valor de m e as listas b, ρ, onde para, $\alpha \in \Omega$, $b[\alpha]$ é o número do bloco contendo α e, para $i \in \{1,...,m\}$, $\rho[i]$ é o representante em Ω do bloco número i.

Algoritmo 3.14: $NUMBERBLOCKS(p)$

Input: Uma lista p definindo um sistema de blocos

Output: m, b, ρ como foram descritas antes

```
m := 0;
for i ∈ [1..n] do
    if p[i] = i then
        m := m + 1; b[i] := m; ρ[m] := i;
    end
end
for i ∈ [1..n] do
    b[i] := b[p[i]];
end
return m, b, ρ;
```

Tendo a função $NUMBERBLOCKS$ a ação induzida $\varphi : G \longrightarrow G^{\Gamma}$ agora pode ser facilmente calculada pela regra $i^{\varphi(g)} = b[\rho[i]^g]$ para $i \in \{1,...,m\}$ e $g \in G$.

Exemplo 3.25 *No exemplo 3.11 encontramos que para o grupo $G =< x >$, com $x := \{(1,2,3,4,5,6),(2,6)(3,5)\}$ e para $\alpha_1 = 1, \alpha_2 = 4$ as classes de equivalência que definem a G-congruência gerada por α_1 e α_2 são: $\{1,4\},\{2,5\},\{3,6\}$, assim a partição anterior de Ω, define um sistema de blocos, posteriormente com a função minimalblock encontramos representado este resultado na lista $p = [1,5,3,1,5,3]$. Neste caso a função numberblocks vai-nos retornar:*

```
gap> Read("numberblocks");
gap> numberblocks([1,5,3,1,5,3]);
[ 3, [ 1, 3, 2, 1, 3, 2 ], [ 1, 3, 5 ] ]
```

Código 3.26 numberblocks(p)

```
numberblocks:=function(p)
local i, n, m, b, rho;
n:=Length(p); m:=[]; b:=[]; rho:=[];
m:=0;
for i in [1..n] do
  if p[i]=i then
    m:=m+1;
    b[i]:=m;
    rho[m]:=i;
  fi;
od;
for i in [1..n] do
  b[i]:=b[p[i]];
od;
return [m,b,rho];
end;
```

Isto é, temos 3 *blocos, onde o bloco número* 1 *é o bloco* $[1,4]$ *e tem como representante o* 1*, o bloco* 2 *é o bloco* $[3,6]$ *que tem como representante o* 3 *e o bloco número* 3 *com representante o* 5 *é o bloco* $[2,5]$*. Neste caso nos geradores originais* x_1, x_2 *teremos:*

$$\begin{array}{rcl} \varphi: \ G & \longrightarrow & Sym(\{1,2,3\}) \\ x_1 & \longmapsto & (1,3,2) \\ x_2 & \longmapsto & (2,3). \end{array}$$

De fato, por exemplo, $1^{\varphi(x1)} = b[\rho[1]^{x_1}] = b[1^{x_1}] = b[2] = 3$*,* $2^{\varphi(x1)} = b[\rho[2]^{x_1}] = b[3^{x_1}] = b[4] = 1$ *e* $3^{\varphi(x1)} = b[\rho[3]^{x_1}] = b[5^{x_1}] = b[6] = 2$ *e portanto a permutação gerada por* x_1 *no sistema de blocos é* $(1,3,2)$*. Isto é, que o bloco* 1 *vai para o bloco* 3*, o bloco* 3 *no bloco* 2 *e finalmente, o bloco* 2 *no bloco* 1*.*

Nosso objetivo agora será encontrar a maneira de calcular $Im(\varphi)$ e $Ker(\varphi)$. Para isto, lembrando a definição do estabilizador de conjunto G_Δ, veremos uma maneira de calcular o estabilizador do bloco contendo α no subgrupo $H =< y >$ de G.

Teorema 3.26 *Seja* G^Ω *preservando o sistema de blocos* Γ*. Sejam* $H \leq G$*,* $\alpha \in \Omega$ *e* Δ *o bloco de* Γ *contendo* α*. Para cada* $\beta \in \alpha^H$*, seja* μ_β *um elemento de* H *com* $\alpha^{\mu_\beta} = \beta$*. Então o estabilizador* H_Δ *de* Δ *em* H *é gerado por* H_α *junto com* $\Sigma = \{\mu_\beta | \beta \in \alpha^H \cap \Delta\}$*.*

Demonstração. Se $g \in H_\alpha$, então $\alpha^g = \alpha$, e como $\alpha \in \Delta$, então $g \in H_\Delta$; além disso se $\mu_\beta \in \Sigma$, então $\alpha^{\mu_\beta} = \beta \in \Delta$, e assim $\mu_\beta \in H_\Delta$, e portanto, $< (H_\alpha, \Sigma) > \subseteq H_\Delta$. Reciprocamente, se $g \in H_\Delta$, então $\alpha^g = \beta$, para algum $\beta \in \Delta$; além disso, para $\mu_\beta \in \Sigma$ temos que $\alpha^{\mu_\beta} = \beta$, assim $\alpha = \beta^{\mu_\beta^{-1}}$, e portanto, $g\mu_\beta^{-1} \in H_\alpha$. Logo temos a igualdade desejada. $\square$

Começamos usando a função *schreiersims* para encontrar uma base para H, começando com o elemento α. Isto permite que os geradores do estabilizador H_α e so elementos μ_β do conjunto Σ sejam calculados pelas funções estândar $BSGS$.

Algoritmo 3.15: $BLOCKSTABILIZER(Y, \alpha, b)$

Input: Geradores Y de $H \leq G$, $\alpha \in \Omega$ e a lista b da ação de blocos

Output: Geradores de H_Δ para o bloco Δ contendo α

1 Mude a base de H, fazendo α o primeiro ponto da base;
2 Seja Z o conjunto de geradores do estabilizador H_α;
3 **for** $\beta \in \alpha^H$ **do**
4 **if** $b[\beta] = b[\alpha]$ *e* $\beta \notin \alpha^{<Z>}$ **then**
5 $Add(Z, \mu_\beta)$, onde $\mu_\beta \in H$ com $\alpha^{\mu_\beta} = \beta$;
6 **end**
7 **end**
8 **return** Z;

Código 3.27 blockstabilizer(y,α,b)

```
blockstabilizer:=function(y,alpha,b)
local S, Z, beta, U, ub;
S:=schreiersims([alpha],y)[2];
Z:=List(GeneratorsOfGroup(stabilizer(alpha,S)));
U:=ortransversal(alpha,S)[2];
for beta in orbit(alpha,S) do
  if b[beta]=b[alpha] and not beta in orbit(alpha,Z) then
    for ub in U do
      if alpha^ub=beta then
        Add(Z,ub);
      fi;
    od;
  fi;
od;
return Z;
end;
```

Tendo em conta a existência do homomorfismo $\varphi : G \longrightarrow Sym([1..m])$, onde para $g \in G$, o elemento $\varphi(g)$ envia i em j se e somente se $\Gamma_i^g = \Gamma_j$; então a permutação $\varphi(g)$ estabiliza i se e somente se g fixa o bloco Γ_i como conjunto, isto é G_{Γ_i}, então a imagem de G_{Γ_i} é o estabilizador de i na $Im(\varphi)$. Portanto, escolhemos blocos Γ_{i_1}, Γ_{i_2}, ... Γ_{i_r} em Γ tais que qualquer permutação que estabilize cada um dos blocos Γ_{i_1}, Γ_{i_2}, ... Γ_{i_r} estabilize todo bloco de Γ. Assim,

$$Ker(\varphi) = \bigcap_{\Gamma_j \in \Gamma} G_{\Gamma_j} = G_{\Gamma_{i_1}, \Gamma_{i_2}, \ldots, \Gamma_{i_r}}.$$

Portanto, $[i_1..i_r]$ é uma base para $Im(\varphi)$.

Algoritmo 3.16: $BLOCKIMAGEKERNEL(G,b)$

Input: Grupo G e uma lista b para ação de bloco $\varphi : G \longrightarrow G^{\Gamma}$

Output: B_Γ, S_Γ e $Ker(\varphi)$

```
X:= Geradores iniciais de G; B_Γ := []; S_Γ := [φ(x)|x ∈ X, φ(x) ≠ 1];
i := 1; escolha α_i ∈ Ω;
while true do
    Add(B_Γ, b[α_i]);
    Mude a base (B,S) de G para fazer α_i o i-ésimo ponto da base;
    Z := BLOCKSTABILIZER(S^(i), α_i, b); Z̄ := [φ(x)|x ∈ Z, φ(x) ≠ 1];
    if Z̄ = ∅ then
        return B_Γ, S_Γ, < Z >;
    else
        S_Γ := S_Γ ∪ Z̄; i := i+1;
        Escolha α_i ∈ Ω com b[α_i] não fixando todos os φ(x) ∈ Z̄;
    end
end
```

Para fazer a implementação da função $BLOCKIMAGEKERNEL$ no sistema GAP vamos primeiro criar uma função *imageblock* que nos retornará o conjunto imagem inicial $[\varphi(x)|x \in X, \varphi(x) \neq ()]$.

Exemplo 3.27 *No exemplo 3.13 vimos que* $p = [1,5,3,1,5,3]$ *forma um sistema de blocos para o grupos* $G =< x >$, *onde* $x = [(1,2,3,4,5,6),(2,6)(3,5)]$. *Agora encontraremos uma BSGS para a imagem do homomorfismo gerado pela ação de blocos e o kernel para tal ação.*

```
gap> Read("orbit");
gap> Read("stabilizer");
gap> Read("ortransversal");
gap> Read("strip");
gap> Read("simsini");
gap> Read("schreiersims");
```

Código 3.28 imageblock(b,ρ,x)

```
imageblock:=function(b,rho,x)
local m, im, r, img, i, j;
r:=Length(x);
m:=Length(rho);
im:=[];
for j in [1..r] do
  im[j]:=[];
  for i in [1..m] do
    im[j][i]:=b[rho[i]^x[j]];
  od;
od;
img:=[];
for j in [1..r] do
  img[j]:=MappingPermListList([1..m],im[j]);
od;
return img;
end;
```

Código 3.29 blockimagekernel(x,p)

```
blockimagekernel:=function(x,p)
local m, b, rho, B, S, Bg, Sg, i, alpha, Bm, Z, Zl,
temp, j, g, SS, n, Sm, xx, x1;
m:=numberblocks(p)[1]; b:=numberblocks(p)[2];
rho:=numberblocks(p)[3]; n:=Length(b);
Bg:=[]; Sg:=Difference(imageblock(b,rho,x),[()]);
alpha:=[]; i:=1; alpha[i]:=1;
while true do
  Add(Bg,b[alpha[i]]);
  B:=schreiersims([alpha[i]],x)[1];
  S:=schreiersims([alpha[i]],x)[2];
  if b[i]<>alpha[i] then
    xx:=prinitialize(S,10,20); x1:=prrandom(xx);
    if B[i]^x1=alpha[i] then
      Bm:=baseconjugation(B,S,x1)[1];
      Sm:=baseconjugation(B,S,x1)[2];
    fi;
  else
    Bm:=B; Sm:=S;
  fi;
  SS:=simsini(Bm,Sm)[2];
  Z:=blockstabilizer(Union(SS[i],[()]),alpha[i],b);
  Zl:=Difference(imageblock(b,rho,Z), [()]);
  if Zl=[] then
    return [Bg,Sg,Group(Z)];
  else
    Sg:=Union(Sg,Zl); i:=i+1;
    for j in [1..n] do
      for g in Zl do
        if b[j]^g<>b[j] then
          temp:=[]; Add(temp, j);
        fi;
      od;
    od;
    alpha[i]:=temp[1];
  fi;
od;
end;
```

```
gap> Read("baseconjugation");
gap> Read("numberblocks");
gap> Read("blockstabilizer");
gap> Read("imageblock");
gap> Read("blockimagekernel");
gap> Read("prrandom");
gap> Read("prinitialize");
gap> Read("blockimagekernel");
gap> p:=[1,5,3,1,5,3];;
gap> num:=numberblocks(p);
[ 3, [ 1, 3, 2, 1, 3, 2 ], [ 1, 3, 5 ] ]
gap> b:=num[2];
[ 1, 3, 2, 1, 3, 2 ]
gap> x:=[(1,2,3,4,5,6),(2,6)(3,5)];;
gap> blockimagekernel(x,p);
[ [ 1, 2 ], [ (2,3), (1,3,2) ], Group(()) ]
```

3.7 Pesquisas Backtrack

Nesta seção mostraremos uma forma de encontrar os elementos de um grupo de permutação de grau n, e, portanto, sua complexidade é pelo menos $O(|G|)$, no pior dos casos. Durante a seção assumiremos uma $BSGS$ (B,S) com $B := [\beta_1..\beta_k]$ para um grupo G. Para $0 \leq l \leq k$, denotemos o segmento $[\beta_1..\beta_l]$ de B por $B(l)$. Começamos definindo uma ordenação $\prec$ sobre os elementos da lista $\Omega := [1..n]$, onde os elementos da base B vêm primeiro e em ordem. Isto é, $\beta_i \prec \beta_j$ para $i < j$, e $\beta_i \prec \alpha$ se $\alpha \notin B$. Pela definição de base, um elemento $g \in G$ é determinado unicamente pela imagem base $B^g := [\beta_1^g..\beta_k^g]$, e assim podemos ordenar os elementos de G pelas imagens base. Isto é, para $g,h \in G$, definimos $g \prec h$ se B^g precede B^h na ordem lexicográfica (ver: 1.51) induzida por $\prec$. Para ser mais preciso, $g \prec h$ se e somente se, para algum l com $1 \leq l \leq k$, temos que $\beta_i^g = \beta_i^h$ para $1 \leq i \leq l$ e $\beta_l^g \prec \beta_l^h$.

3.7.1 Pesquisando através dos elementos de um grupo

A seguinte função $PRINTELEMENTS$ imprime cada elemento do grupo. Isto será feito executando todas as possibilidades para $u_l * u_{l-1} * ... * u_1 := g[l]$, para $1 \leq l \leq k$.

Algoritmo 3.17: $PRINTELEMENTS(G)$

Input: Grupo de permuatção G com uma $BSGS$ (B, S, Δ^*)

$l := 1$; $c[l] := 1$; $\Lambda[l] := SORT(\Delta[l], \prec)$; $u_l := 1_G$
while *true* **do**
  while $l < k$ **do**
    $l := l + 1$; $\Lambda[l] := SORT((\Delta^{(l)})^{g[l-1]}, \prec)$;
    $c[l] := 1$; $\gamma := \Lambda[l][c[l]]^{g[l-1]^{-1}}$; $u_l := u_\gamma^l$;
  end
  print $g[l]$;
  while $l > 0$ ***and*** $c[l] := |\Delta^{(l)}|$ **do**
    $l := l - 1$;
  end
  if $l = 0$ **then**
    return;
  end
  $c[l] := c[l] + 1$; $\gamma := \Lambda[l][c[l]]^{g[l-1]^{-1}}$; $u_l := u_\gamma^l$;
end

Para a criação da função anterior no sistema GAP iniciamos criando as funções *imagebase*, *sort* e g, onde a primeira vai retornar a imagem base de um elemento g do grupo G na base B, a segunda função *sort* vai ordenar o elementos da lista l e a última função simplesmente vai retornar $g[k]$, isto é, $u_l * u_{l-1} * ... * u_1$.

```
imagebase:=function(B,g)
local Bc, i, k;
k:=Length(B);
Bc:=[];
for i in [1..k] do
 Bc[i]:=B[i]^g;
od;
return Bc;
end;
```

```
sort:=function(list,B,n)
local i, l1, l2;
l1:=[]; l2:=[];
for i in list do
 if i in B then
  Add(l1,i);
 else
  Add(l2,i);
 fi;
od;
Sort(l1); Sort(l2);
Append(l1,l2);
return l1;
end;
```

```
g:=function(u,l)
local i, t;
t:=();
for i in [1..l] do
 t:=u[i]*t;
od;
return t;
end;
```

Fazendo uso das funções anteriores, criamos a função $printelements(B, S)$ no sistema GAP. Começaremos utilizando a função *simsini* para calcular as órbitas básicas e os transversais a direita associados a elas. A lista C armazenará na posição l a quantidade de palavras com comprimento l e na lista t a posição l corresponderá com a imagem da

l-ésima órbita básica sob o elemento $g[l]$ aplicando a ordenação $\prec$ de forma crescente. Observe que o pseudo-código tem um bucle maior, "While", que só executa quando, ao menos, um dos bucles internos cumpra-se. O primeiro dos bucles só é executado quando $l < k$, isto é, quando a extensão da palavra não seja igual u maior, do que o comprimento da base; neste caso, se isto ocorrer, o programa imprimirá o elemento $g[l]$. Se o bucle anterior não é satisfeito, então o seguinte bucle só garante a quantidade de elementos de comprimentos l seja igual a quantidade de elementos na l-ésima órbita básica, assim ele diminui um grau do comprimento l e continua o processo. Logo encontramos um condicional que faz parar o programa quando o comprimento l seja zero.

Código 3.30 printelements(B,S)

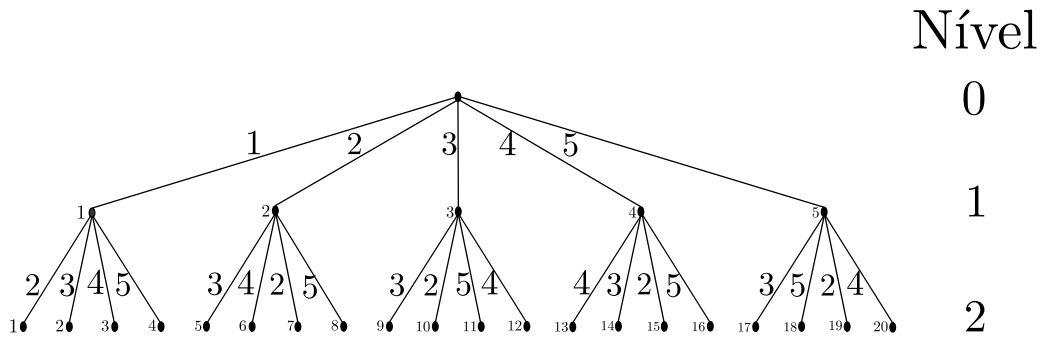
```
printelements:=function(B,S)
local l, c, t, delta, U, u, k, gamma, r;
c:=[]; t:=[]; u:=[];
delta:=simsini(B,S)[4];
U:=simsini(B,S)[5];
k:=Length(B); l:=1; c[l]:=1;
t[l]:=sort(delta[l],B);
u[l]:=();
while true do
 while l<k do
  l:=l+1;
  t[l]:=sort(imagebase(delta[l],g(u,l-1)),B);
  c[l]:=1;
  gamma:=t[l][c[l]]^Inverse(g(u,l-1));
  u[l]:=U[l][Position(delta[l],gamma)];
 od;
 Print(g(u,l), ",␣");
 while l>0 and c[l]=Length(delta[l]) do
  l:=l-1;
 od;
 if l=0 then
  return;
 fi;
 c[l]:=c[l]+1;
 gamma:=t[l][c[l]]^Inverse(g(u,l-1));
 u[l]:=U[l][Position(delta[l], gamma)];
od;
end;
```

Exemplo 3.28 *Neste exemplo veremos como nossa função printelements imprime os elementos do grupo G no exemplo 3.18 na mesma ordem da ferramenta do GAP, Elements(G).*

```
gap> Read("orbit");
gap> Read("stabilizer");
gap> Read("ortransversal");
gap> Read("simsini");
gap> Read("g");
gap> Read("imagebase");
gap> Read("sort");
gap> Read("printelements");
gap> B:=[ 1, 2 ];;
gap> S:=[ (2,3,5,4), (2,5)(3,4), (1,2,4,3), (1,2,5,4) ];;
gap> printelements(B,S);
(), (2,3,5,4), (2,4,5,3), (2,5)(3,4), (1,2)(3,5), (1,2,3,4,5), (1,2,4,3),
```

```
(1,2,5,4), (1,3,4,2), (1,3)(4,5), (1,3,5,2,4), (1,3,2,5), (1,4,5,2),
(1,4,3,5), (1,4)(2,3), (1,4,2,5,3), (1,5,4,3,2), (1,5,3,4), (1,5,2,3), (1,5)
(2,4),
gap> Elements(Group(S));
[ (), (2,3,5,4), (2,4,5,3), (2,5)(3,4), (1,2)(3,5), (1,2,3,4,5), (1,2,4,3),
  (1,2,5,4), (1,3,4,2), (1,3)(4,5), (1,3,5,2,4), (1,3,2,5), (1,4,5,2),
  (1,4,3,5), (1,4)(2,3), (1,4,2,5,3), (1,5,4,3,2), (1,5,3,4), (1,5,2,3),
  (1,5)(2,4) ]
```

O diagrama abaixo mostra a execução da função $PRINTELEMENTS$ no anterior exemplo. Começa no vértice superior com a permutação identidade (), e cada u dos vértices representa uma permutação diferente, sendo assim os 20 vértices no final do árvore o total dos elementos do grupo.



Como a base do grupo tem dois elementos, então nossa árvore só tem nível 2, por exemplo o vértice $(4,1)$, isto é, o vértice número 4 do nível 1 e corresponde a permutação $u_4^{(1)} = (1,4)(2,3)$, isto quer dizer que as permutações baixo deste vértice sempre terão que $1^g = 1^{\mu_1} = 1^{u_4^{(1)}} = 4$, para quaisquer g que esteja baixo deste vértice.

Um caso especial é quando trabalhamos com o grupos simétrico de grau n. Aqui, não precisamos usar os transversais $U^{(i)}$, pois é só executar o conjunto completo de permutações de $[1..n]$ em ordem lexicográfica, começando com a identidade () e terminando com a permutação de reversão de ordem $(n, n-1, ..., 1)$. Isso é o mesmo que a ordem definida pelas imagens de base em relação a base $[1, ..., n-1]$ de G. O problema é encontrar a permutação imediatamente após a permutação atual $g \in Sym(n)$. Uma forma de trabalhar com permutações é na forma vetorial, neste caso a permutação identidade é equivalente a $[1, 2, ..., n]$, vamos utilizar esta notação para a seguinte função $SYMMETRICGROUPELEMENTS$ que vai retornar os elementos do grupos simétrico de grau n. Para encontrar a permutação imediatamente, após a permutação atual $g \in Sym(n)$ faremos o seguinte:

1. olhamos para as imagens $n^g, (n-1)^g, ...$ até que a sequência pare de aumentar em i^g,

2. olhamos novamente as mesmas imagens $n^g, (n-1)^g, \ldots$ até encontrar o elemento j tal que j^g é o elemento imediatamente maior do que i^g,

3. substituímos o elemento i^g por j^g e colocamos as imagens $k^{g'}$ da nova permutação g' para $k > i$ na ordem crescente.

Exemplo 3.29 *Se tomamos a permutação* $g = (1,7,2,6,4) \in Sym(7)$*, na forma vetorial* $[7,6,4,1,5,3,2]$ *vemos que, a sequência das imagens* $7^g = 2$*,* $6^g = 3$*, ..., deixa de crescer para* $i = 4$*, e o* $j = 7$*, pois,* $j^g = 7^g = 2 > 1 = 4^g = i^g$*. Agora, fazendo a substituição do* $i^g = 1$ *por* $j^g = 2$*, temos* $[7,6,4,2,5,3,1]$ *que ao fazer a ordenação das imagens dos* k *maiores do que* $i = 4$*, obtemos* $g' = [7,6,4,2,1,3,5] = (1,7,5)(2,6,3,4)$*.*

O primeiro que a função fara é criar um vetor p com n entradas, onde vai armazenar a permutação g, logo a variável s é usada para trocar os valores de $p[i]$ e $p[j]$.

Algoritmo 3.18: $SYMMETRICGROUPELEMENTS(n)$

```
Input: n ∈ ℕ
for i ∈ [1..n] do
    p[i] := i
end
while true do
    Print p ∈ Sym(n); i := n − 1;
    while i > 0 and p[i] > p[i + 1] do
        i := i − 1;
    end
    if i = 0 then
        break;
    end
    j := n;
    while p[i] > p[j] do
        j := j − 1;
    end
    s := p[i]; p[i] := p[j]; p[j] := s; i := i + 1; j := n;
    while i < j do
        s := p[i]; p[i] := p[j]; p[j] := s; i := i + 1; j := j − 1;
    end
end
```

Código 3.31 symmetricgroupelements(n)

```
symmetricgroupelements:=function(n)
local i, j, p, s;
p:=[];
for i in [1..n] do
 p[i]:=i;
od;
while true do
 Print(PermList(p), ",␣");
 i:=n-1;
 while i>0 and p[i]>p[i+1] do
  i:=i-1;
 od;
 if i=0 then
  break;
 fi;
 j:=n;
 while p[i]>p[j] do
  j:=j-1;
 od;
 s:=p[i]; p[i]:=p[j];
 p[j]:=s; i:=i+1;
 j:=n;
 while i<j do
  s:=p[i]; p[i]:=p[j];
  p[j]:=s; i:=i+1;
  j:=j-1;
 od;
od;
end;
```

Exemplo 3.30 *Neste exemplo encontramos a lista dos elementos do grupo simétrico de grau 4.*

```
gap> Read("symmetricgroupelements");
gap> symmetricgroupelements(4);
(), (3,4), (2,3), (2,3,4), (2,4,3), (2,4), (1,2), (1,2)(3,4), (1,2,3),
(1,2,3,4), (1,2,4,3), (1,2,4), (1,3,2), (1,3,4,2), (1,3), (1,3,4), (1,3)
(2,4), (1,3,2,4), (1,4,3,2), (1,4,2), (1,4,3), (1,4), (1,4,2,3), (1,4)(2,3),
gap> Elements(SymmetricGroup(4));
[ (), (3,4), (2,3), (2,3,4), (2,4,3), (2,4), (1,2), (1,2)(3,4), (1,2,3),
  (1,2,3,4), (1,2,4,3), (1,2,4), (1,3,2), (1,3,4,2), (1,3), (1,3,4),
  (1,3)(2,4), (1,3,2,4), (1,4,3,2), (1,4,2), (1,4,3), (1,4), (1,4,2,3),
  (1,4)(2,3) ]
```

Referências Bibliográficas

[1] AHO, A. V; HOPCROFT, J. E; ULLMAN, J. D. **The Design and Analysis of Computer Algorithms**. Addison-Wesley, 1st edition, 1974.

[2] ÁKOS, S. **An introduction to computational group theory**. Notices Amer. Math. Soc, 44:671–679, 1997.

[3] ATKINSON, M. D. **An algorithm for finding the blocks of a permutation group**. Math. Comput., 29:911–913, 1975.

[4] BABAI, L. **Graph isomorphism in quasipolynomial time [extended abstract]**. In: PROCEEDINGS OF THE FORTY-EIGHTH ANNUAL ACM SYMPOSIUM ON THEORY OF COMPUTING, STOC '16, p. 684–697, New York, NY, USA, 2016. ACM.

[5] BEALS, R. M. **Computing blocks of imprimitivity for small-base groups in nearly linear time**. Larry Finkelstein and William M.Kantor, editors. Groups and Computation, Vol.11,:pp.17–27, 1991.

[6] BUTLER, G. **Effective computation with group homomorphisms**. Journal of Symbolic Computation, 1, 1985.

[7] BUTLER, G. **Fundamental Algorithms for Permutation Groups**. Lecture Notes in Computer Science 559. Springer-Verlag Berlin Heidelberg, 1 edition, 1991.

[8] CELLER, F; LEEDHAM-GREEN, C; MURRAY, S; NIEMEYER, A; O'BRIEN, E. **Generating random elements of a finite group**. Communications in Algebra, 1 January 1995, Vol.23(13), pp.4931-4948, 1995.

[9] FURST, M; HOPCROFT, J; LUKS, E. **[ieee 21st annual symposium on foundations of computer science (sfcs 1980) - syracuse, ny, usa (1980.10.13-1980.10.15)] 21st annual symposium on foundations of computer science (sfcs 1980) - polynomial-time algorithms for permutation groups**. 1980.

[10] The GAP Group. **GAP – Groups, Algorithms, and Programming, Version 4.8.7**, 2017.

[11] The GAP Group. **GAP – Groups, Algorithms, and Programming, Version 4.8.7, Refecence Manual**, 2017.

[12] GERSTING, J. L. **Fundamentos Matemáticos para a Ciência da Computação**. LTC, 3 edition, 2001.

[13] HELFGOTT, H. A. **Graph isomorphisms in quase-polynomial time**. Séminaire BOURBAKI, 2017.

[14] HERSTEIN, I. N. **Abstract algebra**. Prentice-Hall, 3rd ed edition, 1996.

[15] HOLT, D. F; EICK, B; O'BRIEN, E. A. **Handbook of computational group theory**. Discrete Mathematics and its Applications (Boca Raton). Chapman & Hall/CRC, Boca Raton, FL, 2005.

[16] HULPKE, A. **Notes on Computational Group Theory**. Department of Mathematics, Colorado State University, 2010.

[17] JERRUM, M; JERRUM, M; JERRUM, M; JERRUM, M. [**ieee 23rd annual symposium on foundations of computer science - chicago, il, usa (1982.11.3-1982.11.5)**] **23rd annual symposium on foundations of computer science (sfcs 1982) - a compact representation for permutation groups**. 1982.

[18] KNUTH, D. E. **Efficient representation of perm groups**. Combinatorica, 11, 1991.

[19] REHN, T. **Fundamental permutation group algorithms for symmetry computation. diploma thesis in computer science**. Otto-von-Guericke University Magdeburg, 2010.

[20] ROBINSON, D. **A Course in the Theory of Groups**. Springer, 2 edition, 1995.

[21] SERESS, Á. **Permutation group algorithms**. Cambridge tracts in mathematics 152. Cambridge University Press, 1 edition, 2003.

[22] SIMS, C. C. **Computational methods in the study of permutation groups**. In: Leech, J, editor, COMPUTATIONAL PROBLEMS IN ABSTRACT ALGEBRA, p. 169 – 183. Pergamon, 1970.

[23] SIMS, C. C. [**acm press the second acm symposium - los angeles, california, united states (1971.03.23-1971.03.25)**] **proceedings of the second acm symposium on symbolic and algebraic manipulation - symsac '71 - computation with permutation groups**. 1971.

[24] SIMS, C. C. **Determining the conjugacy classes of permutation groups**. Garrett Birkhoff and Marshall Hall Jr., editors, Computers in Algebra and Number Theory,, 4, 1971.

[25] SIMS, C. C. **Computational Group Theory**. Rutgers University, 1998.

[26] SIPSER, M. **Introduction to the Theory of Computation**. Cengage, 3ed. edition, 2012.

[27] WIELANDT, H. **Finite permutation groups**. Academic Press, ap edition, 1964.

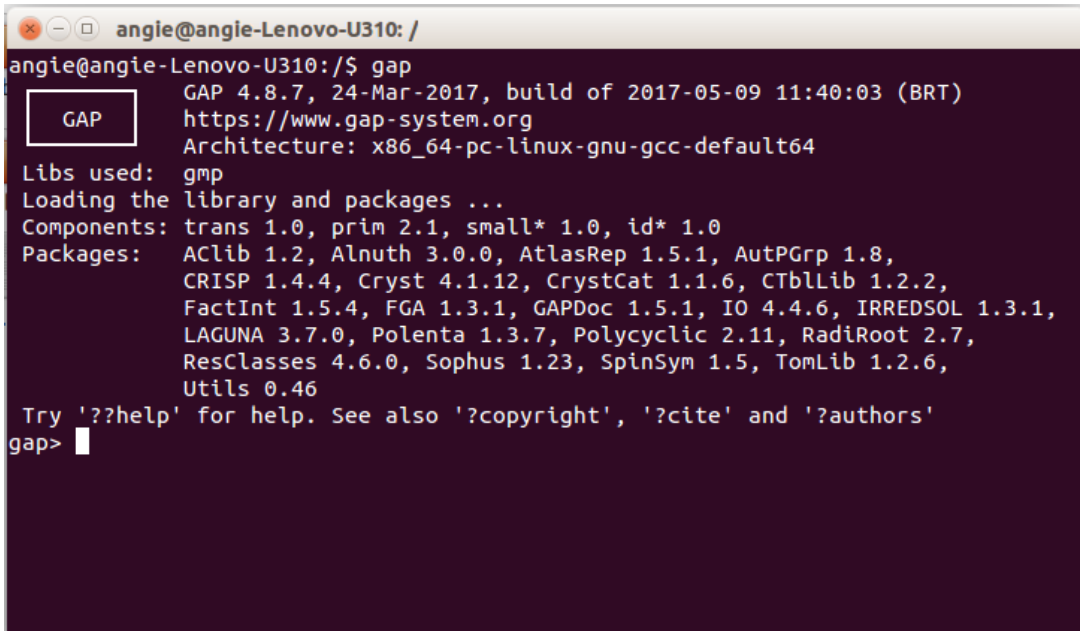
APÊNDICE A

GAP-Groups, Algorithms, Programming

GAP é um sistema de livre aceso utilizado na álgebra discreta computacional, principalmente na teoria de grupos. GAP fornece uma linguagem de programação, uma biblioteca com milhares de funções implementando algoritmos algébricos escritos na linguagem GAP. O GAP foi desenvolvido no departamento de matemática (LDFM), da Universidade Técnica de Aachen na Alemanha, de 1986 a 1997. Mas desde o ano 2005 a coordenação foi transferida uma parceria de quatro Centros GAP, localizados na Universidade de St Andrews; na Universidade Técnica de Aachen; na Universidade Técnica de Braunschweig; e na Universidade do Estado do Colorado em Fort Collins. Todo o relativo ao sistema pode ser encontrado em

https://www.gap-system.org/

Nesta seção vamos expor algumas das ferramentas do GAP utilizadas ao longo do trabalho, baseamos no tutorial [10] e no seu manual de referência [11]. Neste trabalho, trabalhamos o GAP no sistema operacional Linux, assim sempre executamos o programa desde um terminal.



A.1 Listas

As listas são a maneira mais pratica para trabalhar com objetos juntos. Uma lista organiza objetos em uma ordem definitiva. Portanto, cada lista implica uma bijeção dos inteiros para os elementos da lista. Isto é, há um primeiro elemento de uma lista, um segundo, um terceiro e assim por diante. Uma lista pode ser escrita escrevendo os elementos entre colchetes e separando-os com vírgulas. Uma lista vazia, ou seja, uma lista sem elementos, está escrita como [], e uma lista de elementos ordenados, por exemplo, do 3 ao 7 pode ser escrita [3..7]. As listas podem ser nomeadas para trabalhar com elas e com seus elementos usando a função $list[ix]$.

```
gap> l:=[4,6,9];
[ 4, 6, 9 ]
gap> k:=[[],[1,2,3],[9,8]];
[ [ ], [ 1, 2, 3 ], [ 9, 8 ] ]
gap> k[3];
[ 9, 8 ]
gap> k[3][2];
8
```

Podemos adicionar novos elementos à lista usando:

- A função $Add(list,p)$ para adicionar o elemento p ao final da lista.

  ```
  gap> Add(l,8);l;
  [ 4, 6, 9, 8 ]
  ```

- A função $list[ix] := object$ para adicionar ou mudar um objeto na posição dada.

  ```
  gap> l[3]:=10;;l;
  [ 4, 6, 10, 8 ]
  gap> l[7]:=[1,2,3];;l;
  [ 4, 6, 10, 8,,, [ 1, 2, 3 ] ]
  ```

Também podemos remover objetos de uma lista usando a função $Remove(list,pos)$.

```
gap> Remove(l,7);;l;
[ 4, 6, 10, 8 ]
```

A função $Append(list1,list2)$ adiciona a lista 2 ao final da lista 1.

```
gap> Append(l,[1,3,5]);;l;
[ 4, 6, 10, 8, 1, 3, 5 ]
```

Uma outra função muito útil para listas é $in(obj,list)$, permite saber se um objeto encontra-se em uma lista.

```
gap> 4 in l;
true
gap> 2 in l;
false
```

Para encontrar a posição de um objeto em uma lista a função $Position(list, obj)$.

```
gap> Position(l,8);
4
```

Uma forma de fazer uma concatenação de listas é usando a função $Concatenation(list1, list2, ...)$.

```
gap> m:=Concatenation([1,2,5],[5,8],[9,2,3]);
[ 1, 2, 5, 5, 8, 9, 2, 3 ]
```

Além das funções antes dadas também é possível fazer operações aritméticas com listas, uniões, interseções, diferencias, entre muitas outras mais operações.

```
gap> Union(l,m);
[ 1, 2, 3, 4, 5, 6, 8, 9, 10 ]
gap> Intersection(l,m);
[ 1, 3, 5, 8 ]
```

A.2 Funções e linguagem de programação

Além de todas as ferramentas carregadas no GAP, o sistema permite-nos criar novas funções fazendo uso de alguns recursos da programação em geral. É possível criar funções no mesmo programa GAP, é só usar a palavra-chave "function", seguida de uma lista de parâmetros formais delimitados por parênteses, chamados de argumentos da função; logo vai uma série de afirmações que compõe o corpo da função sempre terminando em ";"e ao final dela a palavra "end;". No seguinte exemplo vamos usar a palavra-chave "return"para retornar o resultado da função.

```
gap> soma:=function(a,b)
> return a+b;
> end;
function( a, b ) ... end
gap> soma(-4,5);
1
```

Uma forma de criar funções é colocar o código fonte em outro arquivo de notas e logo ler ele no terminar, onde está executado o GAP. Neste trabalho todas as funções usadas estão na mesma pasta e assim para ler elas mais fácil é só abrir o terminal desde a mesma pasta sempre. Sempre que precisemos de novas variáveis temos que nomear elas ao começo das afirmações colocando antes a palavra "local".

```
soma:=function(a,b)
local c;
c:=a+b;
return c;
end;
gap> Read("soma");
gap> soma(-4,5);
1
```

Na criação dos programas utilizamos algumas das palavras-chave e dos símbolos próprios do GAP. As Palavras-chave são palavras exclusivas que são usadas para designar operações especiais ou fazem parte de declarações. A lista de palavras-chave está contida no GAPInfo.Keywords, e pode ser imprimida da seguinte maneira.

```
gap> SortedList( GAPInfo.Keywords );
[ "Assert", "Info", "IsBound", "QUIT", "TryNextMethod", "Unbind", "and",
  "atomic", "break", "continue", "do", "elif", "else", "end", "false", "fi",
  "for", "function", "if", "in", "local", "mod", "not", "od", "or", "quit",
  "readonly", "readwrite", "rec", "repeat", "return", "then", "true",
  "until", "while" ]
```

Alguns dos símbolos usados são:

```
+    -    *    /    ^    ~
=    <>   <    <=   >    >=
:=   .    ..   ->   ,    ;
[    ]    {    }    (    )
```

A.3 Grupos de permutação

Nesta seção vamos apresentar algumas das ferramentas que tem o GAP para grupos em geral e algumas outras para os grupos de permutação onde mediante a ação natural ele pode calcular órbitas, estabilizadores, transversais a direita entre muitas outras operações descritas no trabalho.

Função	Exemplo
$Group(gens, ...)$	gap> a:=(1,2,4,3);b:=(1,2,5,4);; gap> g:=Group([a,b]); Group([(1,2,4,3), (1,2,5,4)])

$Order(G)$, $Order(elm)$	gap> Order(g);
	20
	gap> Order(a*b);
	5

$Elements(G)$	gap> Elements(g); [(), (2,3,5,4), (2,4,5,3), (2,5)(3,4), (1,2)(3,5), (1,2,3,4,5), (1,2,4,3), (1,2,5,4), (1,3,4,2), (1,3)(4,5), (1,3,5,2,4), (1,3,2,5), (1,4,5,2), (1,4,3,5), (1,4)(2,3), (1,4,2,5,3), (1,5,4,3,2), (1,5,3,4), (1,5,2,3), (1,5)(2,4)]

$Subgroup(G, gens)$	gap> s:=Subgroup(g,[b^2,a*b]);
	Group([(1,5)(2,4), (1,5,4,3,2)])
	gap> Order(s);
	10

$GeneratorsOfGroup(G)$	gap> GeneratorsOfGroup(s); [(1,5)(2,4), (1,5,4,3,2)]
$StructureDescription(G)$	gap> StructureDescription(g); "C5␣:␣C4"

$ConjugacyClasses(G)$,
$ConjugacyClass(G, g)$
	gap> ConjugacyClasses(g);
	[()^G, (2,3,5,4)^G, (2,4,5,3)^G, (2,5)(3,4)^G, (1,2,3,4,5)^G]
	gap> ConjugacyClass(g,a);
	(1,2,4,3)^G

$MovedPoints(G)$,
$LargestMovedPoint(G)$
	gap> MovedPoints(g);
	[1, 2, 3, 4, 5]
	gap> LargestMovedPoint(g);
	5