UNIVERSITEIT GENT

FACULTEIT WETENSCHAPPEN

Computeralgebra: partim computationele groepentheorie

```
   Loading the library. Please be patient, this may take a while.
GAP4, Version: 4.4.6 of 02-Sep-2005, powerpc-apple-darwin8.3.0-gcc
Components:  small 2.1, small2 2.0, small3 2.0, small4 1.0, small5 1.0,
             small6 1.0, small7 1.0, small8 1.0, small9 1.0, small10 0.2,
             id2 3.0, id3 2.1, id4 1.0, id5 1.0, id6 1.0, id9 1.0, id10 0.1,
             trans 1.0, prim 2.1  loaded.
Packages:    TomLib 1.1.2  loaded.
gap> b := (17,18,19,20)(4,8,22,11)(3,7,21,12);
(3,7,21,12)(4,8,22,11)(17,18,19,20)
gap> l := (9,10,12,11)(13,1,17,21)(15,4,20,24);
(1,17,21,13)(4,20,24,15)(9,10,12,11)
gap> a := (21,22,23,24)(7,14,9,20)(6,13,11,19);
(6,13,11,19)(7,14,9,20)(21,22,23,24)
gap> cube := Group([b,l,a]);
Group([ (3,7,21,12)(4,8,22,11)(17,18,19,20), (1,17,21,13)(4,20,24,15)(9,10,12,
    11), (6,13,11,19)(7,14,9,20)(21,22,23,24) ])
gap> f := FreeGroup("bo","li","ac");
<free group on the generators [ bo, li, ac ]>
gap> hom := GroupHomomorphismByImages(f,cube,GeneratorsOfGroup(f),
GeneratorsOfGroup(cube));
[ bo, li, ac ] -> [ (3,7,21,12)(4,8,22,11)(17,18,19,20),
  (1,17,21,13)(4,20,24,15)(9,10,12,11), (6,13,11,19)(7,14,9,20)(21,22,23,24) ]
gap> S := StabChain(cube);
<stabilizer chain record, Base [ 1, 3, 4, 6, 7, 9 ], Orbit length 21, Size:
3674160>
gap> baseim  := [17,11,10,22,23,24];
[ 17, 11, 10, 22, 23, 24 ]
gap> g := permutationbybaseimage(S,baseim);
(1,17,15,4,10,12)(3,11)(6,22)(7,23)(8,20)(9,24,13)(14,19)(18,21)
gap> PreImagesRepresentative(hom,g^-1);
bo*ac^2*bo^-1*li^-1*ac*bo*li*ac^-1*li^-1*bo^-1*ac^-1*bo
```

dr. J. De Beule
Vakgroep Zuivere Wiskunde en Computeralgebra

keuzevak licenties wiskunde/licenties informatica
februari 2007

Voorwoord

Deze cursusnota's horen bij het partim "Computationele groepentheorie" van de cursus "Computeralgebra" uit de licenties wiskunde. Het is in dit partim de bedoeling een *inleiding* te geven tot een vakgebied dat kan omschreven worden als "computationele groepentheorie".

Groepentheorie kan beschouwd worden als een zeer abstract vakgebied binnen de wiskunde. Men kan het zien als een volledig op zichzelf staande tak. Maar de interactie tussen groepentheorie en andere deelgebieden in de wiskunde is vergelijkbaar met de interactie tussen wiskunde en wetenschappen zoals fysica en informatica. Groepentheorie kan zeer nuttig zijn om wiskundige structuren uit andere vakgebieden te onderzoeken. We geven een voorbeeld. Beschouw de kwadriek $\mathrm{Q}(4, q)$ en beschouw twee punten p_1 en p_2 die niet op een rechte van $\mathrm{Q}(4, q)$ liggen. Werkt de stabilisatorgroep van $\mathrm{Q}(4, q)$ transitief op de paren niet-collineaire punten? Het antwoord op deze vraag kan in algemeenheid gegeven worden, maar dat is niet altijd eenvoudig. Soms is ook voeldoende om een aanwijzing te hebben om het onderzoek in de juiste richting verder te zetten. En soms is het voldoende om het antwoord op deze vraag voor welbepaalde q te weten. Een groot aantal computeralgebrasystemen is tegenwoordig in staat om de gebruiker, op een min of meer gebruiksvriendelijke wijze, een antwoord te geven op dergelijke vragen. Daartoe zijn er echter vaak berekeningen noodzakelijk, die berekeningen hebben een zekere complexiteit, en vereisen bijgevolg CPU-tijd en werkgeheugen (algemeen: *resources*). Om de berekeningen zo efficient mogelijk te maken zijn goede algoritmen nodig en om goede algoritmen te ontwikkelen is dikwijls (abstracte) wiskunde nodig, en net groepentheorie leent zich uitstekend tot abstracte redeneringen. (Meer) kennis van wiskundige eigenschappen van groepen stelt ons in staat om (betere) algoritmen te ontwikkelen voor de berekeningen in groepen. Dit feit willen we illustreren in dit opleidingsonderdeel. We vertrekken van een eenvoudig probleem in een concreet voorbeeld, en we gaan, met de computer aan de slag om de berekening te doen. Hoe meer wiskundige kennis we gebruiken om de berekeningen te doen, des te groter zal het probleem zijn dat de computer aankan en des te sneller zullen de berekeningen gedaan worden.

Het spreekt voor zich dat de bespreking van algoritmen slechts zinvol is indien ze geïllustreerd kunnen worden door een implementatie op een computer. Wij zullen hiertoe gebruik maken van de computeralgebrapakketen MAGMA [5] en GAP [2].

Eerdere versies van deze lessen, gedoceerd door Prof. Dr. F. De Clerck en nadien door Prof. Dr. L. Storme, en nota's, waren gebaseerd op een aantal hoofdstukken uit het boek: *Fundamental algorithms for permutation groups*, [1]. Dit (uitstekend) boek levert nog steeds een gedeelte van deze nota's. Het boek bestaat uit drie delen: "Kleine groepen", waarin eenvoudige algoritmen bestudeerd worden die eigenlijk enkel toepasbaar zijn op kleine groepen; "Permutatiegroepen", waarin oudere maar nog steeds fundamentele algoritmen beschreven worden die toepasbaar zijn op permutatiegroepen; en "Homomorfismen", waarin aandacht besteed wordt aan het werken met homomorfismen in een computeralgebrasysteem. De eerste drie hoofdstukken van deze nota's zijn gebaseerd op een gedeelte van het eerste deel van dit boek.

In 2005 verscheen het *Handbook of computational group theory*, [3]. Dit boek is een echt handboek voor het vakgebied. Het bevat een korte geschiedenis van het vakgebied, alle nodige wiskundige achtergrond en belangrijke algoritmen uit de voorbije eeuw. Het boek gaat ook verder door recente ontwikkelingen en nieuwe algoritmen te beschrijven, door niet alleen fundamentele algoritmen te behandelen voor permutatiegroepen, maar ook door aan vrije groepen, polycyclische groepen, matrixgroepen enz. aandacht te besteden; door interessante toepassingen van computationele groepentheorie in bijvoorbeeld automatentheorie en herschrijfsystemen te beschrijven en tenslotte door interessante open problemen en zeer recente ontwikkelingen te beschrijven. Op deze wijze wordt de computationele groepentheorie goed geplaatst in het vakgebied dat we met de term *computeralgebra* kunnen omschrijven.

De wiskundige achtergrond uit hoofdstuk 1 van dit boek vormt de basis van de wiskundige achtergrond die in deze nota's over de eerste drie hoofdstukken verdeeld is. Hoofdstuk 3 behandelt de voorstelling van groepen in een computeralgebrasysteem en vormt de basis van hoofdstuk 4 in deze nota's. Het lijvige hoofdstuk 4 uit het boek vormt de basis voor hoofdstuk 5.

Jan De Beule
februari 2006

Een lange lijst van typografische fouten werd mij overhandigd door de studenten van het academiejaar 2005–2006 die dit vak volgden. Dankzij hun aandacht en ijver was het zeer eenvoudig om de nota's voor dit academiejaar te verbeteren. Ik wens hen bij deze dan ook te bedanken voor het nauwkeurig optekenen van deze fouten. De nota's van dit jaar zijn, op deze correcties en op een kleine toevoeging in Hoofdstuk 5 na, identiek aan de nota's van vorig academiejaar.

Jan De Beule
februari 2007

Inhoudsopgave

Voorwoord **iii**

Inhoudsopgave **v**

Lijst van figuren **ix**

Lijst van algoritmen **xi**

1 Inleiding **1**
- 1.1 Definities . . . 1
  - 1.1.1 Groepen . . . 1
  - 1.1.2 Deelgroepen . . . 2
  - 1.1.3 Cyclische groepen, Diëdergroepen en generatoren . . . 2
  - 1.1.4 Enkele voorbeelden . . . 3
  - 1.1.5 Eerste voorbeelden met GAP en MAGMA . . . 4
  - 1.1.6 Normaaldelers en quotiëntgroepen . . . 7
- 1.2 Bepaling van alle elementen van een groep . . . 7
- 1.3 Het algoritme van Dimino . . . 10
- 1.4 Toepassing: eenvoudige zoekalgoritmen voor kleine groepen . . . 15
- 1.5 Oefeningen . . . 19

2 Acties van groepen, Cayleygraaf en definiërende relaties van een groep **21**
- 2.1 Definities . . . 21
  - 2.1.1 Homomorfismen en isomorfiestellingen . . . 21
  - 2.1.2 Acties van groepen . . . 22
  - 2.1.3 Banen en stabilisatoren . . . 24
  - 2.1.4 Toevoeging, normalisator en centralisator . . . 25
  - 2.1.5 Transitiviteit en primitiviteit . . . 25
  - 2.1.6 Vrije groepen . . . 27
- 2.2 Definiërende relaties van een groep . . . 28
- 2.3 De bepaling van de relaties . . . 31

2.4 Het "Colouring" algoritme . . . 32
2.5 Implementatie en voorbeelden . . . 36
2.6 Oefeningen . . . 37

3 Tralie van deelgroepen **39**
3.1 Definities . . . 39
3.2 De tralie ingedeeld in lagen . . . 40
3.3 Testen wanneer een groep nieuw is . . . 42
3.4 Gebruik van *p*-groepen . . . 44
3.5 Voorbeelden . . . 47

4 Representatie van groepen op een computer **51**
4.1 Representatie van groepen op een computer . . . 51
4.1.1 Fundamentele abstracte datatypes . . . 51
4.1.2 Computationele situaties . . . 52
4.1.3 Straight-line programma's . . . 53
4.1.4 Black-box groepen . . . 54
4.2 Random methoden in Computationele groepentheorie . . . 55
4.2.1 Random algoritmen . . . 55
4.2.2 Random elementen . . . 56
4.3 Enkele elementaire algoritmen . . . 58
4.3.1 Machten en de orde van een element . . . 59
4.3.2 Normale sluiting . . . 60
4.4 Homomorfismen . . . 61
4.5 Oefeningen . . . 63

5 Eindige permutatiegroepen **67**
5.1 Banen en stabilisatoren . . . 67
5.2 Controle op alternerende of symmetrische groep . . . 72
5.3 Bloksystemen . . . 75
5.3.1 Inleiding . . . 75
5.3.2 Het atkinson algoritme . . . 76
5.3.3 Het samenvoegen van klassen . . . 78
5.4 Basis en sterk voortbrengende generatoren . . . 80
5.4.1 Inleiding . . . 80
5.4.2 Het Schreier-Sims algoritme . . . 87
5.4.3 Basisverandering . . . 90
5.5 Rubik's kubus en GAP . . . 93
5.6 Oefeningen . . . 102

Bibliografie **103**

Lijst van figuren

1.1 het Petersen graaf 5

2.1 de hoekpunten van het vierkant gelabelled 29
2.2 Cayleygraaf van de groep D_8 30
2.3 spanning tree 30
2.4 initieel gekleurde bogen 34
2.5 gekleurde bogen na tracering van eerste relatie rond 5 34
2.6 gekleurde bogen na traceringen van twee definiërende relaties 35
2.7 een minimale spanning tree 35

3.1 tralie van deelgroepen van D_8 41

5.1 Rubik's $3 \times 3 \times 3$ kubus, in sleutelhangerformaat 93
5.2 Rubik's $3 \times 3 \times 3$ kubus, vlakjes genummerd 94
5.3 Rubik's $2 \times 2 \times 2$ kubus, vlakjes genummerd 99

Lijst van algoritmen

1.1 bepaling van alle elementen van een groep (1) 8
1.2 bepaling van alle elementen van een groep (2) 9
1.3 inductieve stap in het algoritme van Dimino 11
1.4 het vereenvoudigd algoritme van Dimino 12
1.5 zoeken in een lijst naar 1 element 15
1.6 het verwijderen van elementen die niet voldoen 17
1.7 het verwijderen van nevenklassen die niet voldoen 18
1.8 alle elementen die aan een eigenschap voldoen 18

2.1 definiërende relaties door het Colouring-algoritme 33

3.1 tralie van deelgroepen: inductieve stap (1) 42
3.2 tralie van deelgroepen: inductieve stap (2) 43
3.3 tralie van deelgroepen: inductieve stap (3) 44
3.4 tralie van deelgroepen: inductieve stap (4) 46

4.1 random elementen van een groep: initialisatie 57
4.2 random elementen van een groep 58
4.3 macht van een element 59
4.4 orde van een element 60
4.5 normale sluiting 61

5.1 baan van een element 67
5.2 baan van een element en stabilisator 69
5.3 baan van een element en Schreiervector 70
5.4 trace functie 71
5.5 random stabilizator van een element 72
5.6 controle op alternerende of symmetrische groep 74
5.7 een minimaal bloksysteem 77
5.8 representant van een klasse 79
5.9 samenvoegen van klassen 79
5.10 een minimaal bloksysteem (2) 80

5.11 membership test . . . 83
5.12 membership test (2) . . . 84
5.13 membership test (3) . . . 84
5.14 beeld van een basis . . . 85
5.15 opsomming van alle elementen . . . 86
5.16 opsomming van alle basisbeelden . . . 87
5.17 het Schreier-Sims algoritme . . . 89
5.18 basisverandering . . . 91
5.19 verwijderen van redundante generatoren . . . 93

Hoofdstuk 1

Inleiding

1.1 Definities

1.1.1 Groepen

Een *groep* (**E:** *group*) is een niet-ledige verzameling G, voorzien van een binaire, interne bewerking, genoteerd met $\cdot$, die aan de volgende voorwaarden voldoet:

(G1) $a \cdot (b \cdot c) = (a \cdot b) \cdot c$

(G2) $(\exists e \in G)(\forall g \in G)(e \cdot g = g = g \cdot e)$

(G3) $(\forall g \in G)(\exists g^{-1} \in G)(g \cdot g^{-1} = e = g^{-1} \cdot g)$.

Het element $e \in G$ dat aan de voorwaarde (G2) voldoet wordt het *eenheidselement* (**E:** *identity*) genoemd. De voorwaarden (G1), (G2) en (G3) sluiten niet uit dat G juist 1 element bevat. In dat geval is G de *triviale groep* (**E:** *trivial group*), en bestaat G enkel uit het eenheidselement. Een groep is *abels* (**E:** *abelian*) of *commutatief* (**E:** *abelian*) als en slechts als $G, \cdot$ aan de volgende voorwaarde voldoet:

(G4) $(\forall g, h \in G)(g \cdot h = h \cdot g)$

De *orde* (**E:** *order*) van de groep G is het aantal elementen dat G bevat en wordt genoteerd als $|G|$. Als $|G|$ eindig is, dan noemen we G een *eindige groep* (**E:** *finite group*). Het gebruik van $\cdot$ is slechts een notatie. Indien de bewerking uit de context duidelijk is, dan zullen we de $\cdot$ weglaten: we noteren $G, \cdot$ als G en $g \cdot h$ als gh en we noemen dit de multiplicatieve notatie, G is een multiplicatieve groep; het eenheidselement noteren we met $\mathbf{1}$. Indien we in plaats van $\cdot$ een $+$ gebruiken om de bewerking te noteren, dan spreken we van een additieve notatie of additieve groep; het eenheidselement noteren we met $\mathbf{0}$. De afspraak is dat een additieve groep steeds abels is. Wanneer we het over abstracte groepen hebben, dan zullen we steeds de multiplicatieve notatie gebruiken.

Stel dat G een groep is. Dan definiëren we, voor elke $g \in G$, g^n inductief als $g^1 = g$ en $g^n = gg^{n-1}$, $n > 1$, en g^{-n} als de inverse van g^n. Voor de volledigheid definiëren we $g^0 = \mathbf{1}$. Het is duidelijk dat uit deze definitie volgt dat $g^n \in G$. Voor $g \in G$ definiëren we de *orde* (**E:** *order*) van g als het kleinste getal $n \in \mathbb{N} \setminus \{0\}$ waarvoor $g^n = \mathbf{1}$, we noteren $|g| = n$. Uit deze definitie volgt dat $|g| = 1 \iff g = \mathbf{1}$ en dat $g^k = \mathbf{1} \iff |g| \mid k$.

Stel dat G en H twee groepen zijn. Het *direct product* (**E:** *direct product*) $G \times H$ is de verzameling $\{(g, h)|g \in G, h \in H\}$ van geordende paren elementen uit G en H. De interne bewerking is als volgt gedefinieerd: $(g_1, h_1)(g_2, h_2) = (g_1g_2, h_1h_2)$ voor $g_1, g_2 \in G$ en $h_1, h_2 \in H$. Het is eenvoudig na te gaan dat $G \times H$ samen met de aldus gedefinieerde bewerking een groep is. We kunnen nu G^n, $n > 1$, inductief definiëren als $G \times G^{n-1}$.

1.1.2 Deelgroepen

Stel dat $G, \cdot$ een groep is. Stel dat $H \subset G$. We zeggen dat H een *deelgroep* (**E:** *subgroup*) is van G, genoteerd $H \leqslant G$, als en slechts als $H, \cdot$ een groep is; m.a.w. de beperking van $\cdot$ tot de deelverzameling H voldoet aan de voorwaarden (G1), (G2) en (G3). Het volgende criterium is bekend: $H \subset G$ is een deelgroep van G als en slechts als $H \neq \emptyset$ en $(\forall a, b \in H)(ab^{-1} \in H)$.

Stel nu dat $H \leqslant G$. Stel dat $g \in G$. Definieer $Hg := \{hg|h \in H\}$. We noemen Hg een *rechtse nevenklasse* (**E:** *right coset*) van H. Als $g \in H$, dan volgt uit de definitie dat $Hg = H$. Als $g' \in Hg$, dan volgt uit de definitie dat $Hg' = Hg$, elk element uit Hg bepaalt dus, samen met H de rechtse nevenklasse en wordt een (rechtse nevenklasse) *representant* (**E:** *representative*) genoemd. Tenslotte geldt dat $(\forall g_1, g_2 \in G)(Hg_1 \cap Hg_2 = \emptyset$ of $Hg_1 = Hg_2)$. De rechtse nevenklassen vormen dus een partitie van de verzameling G. Hieruit volgt de stelling van Lagrange:

Stelling 1.1.1. *Stel dat G een eindige groep is en stel dat H een deelgroep is van G. Dan is de orde van H een deler van de orde van G.*

De *index* (**E:** *index*) $|G : H|$ is gedefinieerd als het aantal verschillende rechtse nevenklassen van H in G. Dit getal kan eindig of oneindig zijn, maar is gelijk aan $\frac{|G|}{|H|}$ als G een eindige groep is. Een verzameling T van rechtse nevenklasserepresentanten van H in G wordt een *rechtse transversaal* (**E:** *right transversal*) genoemd. In bovenstaande definities kan natuurlijk "rechts" vervangen worden door "links". Ga als oefening na dat T een rechtse transversaal is als en slechts als T^{-1} een linkse transversaal is, met $T^{-1} := \{g^{-1}|g \in T\}$. Een eenvoudig gevolg van de stelling van Lagrange is dat $|g|$ een deler is van $|G|$, voor elk element g uit de groep G.

1.1.3 Cyclische groepen, Diëdergroepen en generatoren

Een groep G is *cyclisch* (**E:** *cyclic*) als alle elementen geschreven kunnen worden als een gehele macht van juist één element. Met andere woorden, G is een cyclische groep als

er een element $g \in G$ bestaat zodat er voor elke $h \in G$ een $k \in \mathbb{Z}$ bestaat met $h = g^k$. Het element g noemen we de *generator* (**E:** *generator*). Cyclische groepen zijn isomorf als en slechts als ze dezelfde orde hebben. We spreken meestal van de cyclische groep van orde n, genoteerd C_n.

Kies $n \geqslant 2$, $n \in \mathbb{N}$ en beschouw een regelmatige n-hoek P in het Euclidisch vlak. De *diëdergroep* (**E:** *dihedral group*) van orde $2n$, genoteerd D_{2n} is de groep bestaande uit n rotaties rond het middelpunt van P en over een hoek van $\frac{2\pi k}{n}$, $0 \leqslant k < n$, en de n spiegelingen ten op zichte van rechten door het middelpunt van P die ofwel door een (twee) hoekpunt(en) gaan en/of door het (de) middelpunt(en) van een (twee) zijde(n).

Zij $G, \cdot$ een groep en $D \subset G$ een deelverzameling van G. Met $\langle D \rangle$ noteren we de verzameling van alle elementen van G die geschreven kunnen worden als een vermenigvuldiging van een eindig aantal elementen uit D, samen met hun inverse. Is $G = \langle D \rangle$ dan brengt D de groep G voort. Als D eindig is, dan zeggen we dat G eindig voortgebracht wordt.

1.1.4 Enkele voorbeelden

Een *permutatie* (**E:** *permutation*) is een bijectie van een verzameling op zichzelf. Stel dat X een willekeurige verzameling is. De verzameling van alle permutaties van X noteren we met $\mathrm{Sym}(X)$. Onder samenstelling van permutaties is $\mathrm{Sym}(X)$ een groep, we noemen deze groep de *symmetrische groep* (**E:** *symmetric group*) op X. Elke deelgroep van $\mathrm{Sym}(X)$ wordt een *permutatiegroep* (**E:** *permutationgroup*) genoemd. Voor $g \in \mathrm{Sym}(X)$ en $x \in X$ noteren we het beeld van x onder g met x^g. De vermenigvuldiging van twee elementen $g, h \in \mathrm{Sym}(X)$ stelt het element voor dat ontstaat door eerst g uit te voeren en dan h, m.a.w. $x^{(gh)} = (x^g)^h$.

Als X een eindige verzameling is, $|X| = n$, $n \in \mathbb{N}$, dan kunnen we X identificeren met $\{1, \ldots, n\}$. We noteren dan $\mathrm{Sym}(X)$ als S_n of ook $\mathrm{Sym}(n)$. Met deze notaties is S_1 de triviale groep. Stel dat σ een permutatie is van X. Neem een willekeurige $i \in X$. Stel $i_0 = i$, $i_1 = i^\sigma$. Definieer vervolgens op inductieve wijze $i_k = i_{k-1}^\sigma$, zolang $i_{k-1}^\sigma \neq i_0$. We bekomen een eindige rij $(i_0, i_1, \ldots, i_l)$ voor zekere $l \in \mathbb{N}$, met $i_l^\sigma = i_0$. Zo een rij noemen we een *cykel* (**E:** *cycle*). Het is duidelijk dat σ te schrijven is als de samenstelling van disjuncte cykels. Een cykel van lengte 2 noemen we een *transpositie* (**E:** *transposition*). Elke cykel is te schrijven als het product van transposities, bijgevolg elke permutatie. Een permutatie noemen we *even* (**E:** *even*) resp. *oneven* (**E:** *odd*) als ze te schrijven is als de samenstelling van een even aantal transpostities, resp. oneven aantal transposities. Men kan aantonen dat geen permutatie te schrijven is als het product van zowel een even als oneven aantal transposities. De verzameling van alle even permutaties vormt een deelgroep van S_n en noteren we met A_n of $\mathrm{Alt}(n)$ of $\mathrm{Alt}(X)$. we noemen deze groep de *alternerende groep* (**E:** *alternating group*) op X. Het is eenvoudig in te zien dat $|S_n : A_n| = 2$.

Veronderstel dat K een veld is. Kies een vaste $d > 0$, dan vormen de inverteerbare

$d \times d$ matrices over K, samen met de klassieke matrixvermenigvuldiging, een groep. Deze groep noteren we met $GL(d, K)$. Wanneer we ons beperken tot de deelverzameling van inverteerbare matrices met determinant 1, dan bekomen we een deelgroep, genoteerd met $SL(d, k)$.

Beschouw de projectieve ruimte PG(n, q). Een collineatie ϕ is een afbeelding van PG(n, q) naar zichzelf, die k-dimensionale deelruimten afbeeldt op k-dimensionale deelruimten, $-1 \leq k \leq n$, en waarvoor geldt: $\alpha \text{ I } \beta \iff \alpha^\phi \text{ I } \beta^\phi$, met I de incidentie in PG(n, q). We definieren de binaire operatie als de samenstelling van twee collineaties, d.i., $\alpha^{\phi \cdot \psi} = (\alpha^\phi)^\psi$. We noteren deze groep met PΓL$(n + 1, q)$.

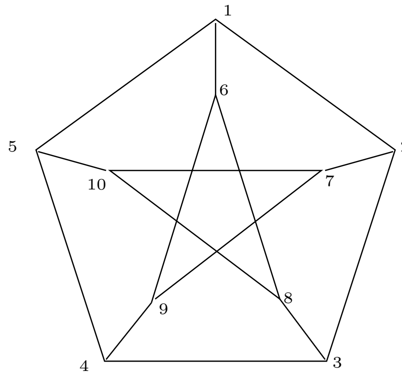
1.1.5 Eerste voorbeelden met GAP en MAGMA

We zullen in GAP en MAGMA de bovenstaande voorbeelden construeren. De bijhorende files zijn `magma1.1` en `gap1.1` respectievelijk, we starten met het MAGMA voorbeeld.

```
"VIERGROEP VAN KLEIN";
G<a,b>:=PermutationGroup<4| (1,2)(3,4), (1,4)(2,3)>;
print G;
ElementSet(G,G);

"CYCLISCHE GROEP VAN DE ORDE 4";
G<a>:=PermutationGroup<4|(2,3,4,1)>;
print G;
ElementSet(G,G);

"AUTOMORPHISMGROUP OF THE PETERSEN GRAPH";
G<a,b,c>:=PermutationGroup<10|(4,8)(5,6)(9,10),(2,5)(3,4)(7,10)(8,9),
                              (1,3,5,2,4)(6,8,10,7,9)>;
Order(G);

"COLLINEATIEGROEP VAN PG(2,2)";
G<a,b>:=PermutationGroup<7| (1,2)(3,5), (2,4,3)(5,7,6)>;
print G;
Order(G);

"GL(d,q)";
G := GL(4,9);
gens := Generators(G);
print gens;
Order(G);

"SL(d,q)";
G := SL(4,9);
gens := Generators(G);
print gens;
Order(G);
```

Het commando `magma < magma1.1 >magma1.1.o` zal de inhoud van `magma1.1` inlezen in magma (hier versie 2.11) en de uitvoer zal geschreven worden in het bestand `magma1.1.o`. Hetzelfde geldt uiteraard voor `gap1.1` en `gap1.1.o`. Hier geven we een listing van magma1.1.o

```
Magma V2.11-13    Fri Aug  5 2005 19:37:54 on Computer-va [Seed = 2904312860]
Type ? for help.  Type <Ctrl>-D to quit.
VIERGROEP VAN KLEIN
Permutation group G acting on a set of cardinality 4
    (1, 2)(3, 4)
    (1, 4)(2, 3)
{
    (1, 4)(2, 3),
    (1, 2)(3, 4),
    (1, 3)(2, 4),
    Id(G)
```



Figuur 1.1: het Petersen graaf

```
}
CYCLISCHE GROEP VAN DE ORDE 4
Permutation group G acting on a set of cardinality 4
    (1, 2, 3, 4)
{
    (1, 3)(2, 4),
    Id(G),
    (1, 4, 3, 2),
    (1, 2, 3, 4)
}
AUTOMORPHISMGROUP OF THE PETERSEN GRAPH
120
COLLINEATIEGROEP VAN PG(2,2)
Permutation group G acting on a set of cardinality 7
    (1, 2)(3, 5)
    (2, 4, 3)(5, 7, 6)
168
GL(d,q)
{
    [  $.1     0     0     0]
    [    0     1     0     0]
    [    0     0     1     0]
    [    0     0     0     1],

    [    2     0     0     1]
    [    2     0     0     0]
    [    0     2     0     0]
    [    0     0     2     0]
}
1624314979123200
SL(d,q)
{
    [    2     0     0     1]
    [    2     0     0     0]
    [    0     2     0     0]
    [    0     0     2     0],

    [  $.1     0     0     0]
    [    0 $.1^7     0     0]
    [    0     0     1     0]
    [    0     0     0     1]
}
203039372390400

Total time: 0.870 seconds, Total memory usage: 10.99MB
```

We geven ter afwisseling een illustratie van een interactieve GAP-sessie (`gap1.1.i`).

```
            #########           ######         ###########           ###
         #############          ######         ############         ####
        ##############         ########        #############       #####
       ###############         ########        #####   ######      #####
      ######         #         #########       #####    #####     ######
     ######                   ##########       #####    #####    #######
     #####                    ##### ####       #####   ######   ########
     ####                    #####  #####      #############   ###  ####
     #####     #######       ####    ####      ###########    ####  ####
     #####     #######      #####    #####     ######        ####   ####
     #####     #######      #####    #####     #####         #############
      #####      #####     ################    #####         #############
      ######     #####     ################    #####         #############
      ################    ##################   #####                ####
       ###############    #####        #####   #####                ####
         #############    #####        #####   #####                ####
          #########      #####          #####  #####                ####

     Information at:  http://www.gap-system.org
     Try '?help' for help. See also  '?copyright' and  '?authors'

   Loading the library. Please be patient, this may take a while.
GAP4, Version: 4.4.3 of 3-May-2004, powerpc-apple-darwin7.5.0-gcc
Components:  small 2.0, small2 2.0, small3 2.0, small4 1.0, small5 1.0,
             small6 1.0, small7 1.0, small8 1.0, id2 3.0, id3 2.1, id4 1.0,
             id5 1.0, id6 1.0, trans 1.0, prim 2.1  loaded.
Packages:    TomLib 1.1.1  loaded.
gap> "VIERGROEP VAN KLEIN";
"VIERGROEP VAN KLEIN"
gap> G := Group([(1,2)(3,4),(1,4)(2,3)]);
Group([ (1,2)(3,4), (1,4)(2,3) ])
gap> G;
Group([ (1,2)(3,4), (1,4)(2,3) ])
gap> Print(G,"\n");
Group( [ (1,2)(3,4), (1,4)(2,3) ] )
gap> List(G);
[ (), (1,2)(3,4), (1,4)(2,3), (1,3)(2,4) ]
gap> "CYCLISCHE GROEP VAN DE ORDE 4";
"CYCLISCHE GROEP VAN DE ORDE 4"
gap> G := Group([(2,3,4,1)]);
Group([ (1,2,3,4) ])
gap> List(G);
[ (), (1,3)(2,4), (1,4,3,2), (1,2,3,4) ]
gap> "GROUP OF THE PETERSEN GRAPH";
"GROUP OF THE PETERSEN GRAPH"
gap> G := Group([(4,8)(5,6)(9,10),(2,5)(3,4)(7,10)(8,9),
> (1,3,5,2,4)(6,8,10,7,9)]);
Group([ (4,8)(5,6)(9,10), (2,5)(3,4)(7,10)(8,9), (1,3,5,2,4)(6,8,10,7,9) ])
gap> Size(G);
120
gap> Order(G);
120
gap> "COLLINEATIEGROEP VAN PG(2,2)";
"COLLINEATIEGROEP VAN PG(2,2)"
gap> G := Group([(1,2)(3,5),(2,4,3)(5,7,6)]);
Group([ (1,2)(3,5), (2,4,3)(5,7,6) ])
gap> Size(G);
168
gap> "GL(d,q)";
"GL(d,q)"
gap> G := GL(4,9);
GL(4,9)
gap> gens := GeneratorsOfGroup(G);
[ [ [ Z(3^2), 0*Z(3), 0*Z(3), 0*Z(3) ], [ 0*Z(3), Z(3)^0, 0*Z(3), 0*Z(3) ],
      [ 0*Z(3), 0*Z(3), Z(3)^0, 0*Z(3) ], [ 0*Z(3), 0*Z(3), 0*Z(3), Z(3)^0 ] ]
    , [ [ Z(3), 0*Z(3), 0*Z(3), Z(3)^0 ], [ Z(3), 0*Z(3), 0*Z(3), 0*Z(3) ],
      [ 0*Z(3), Z(3), 0*Z(3), 0*Z(3) ], [ 0*Z(3), 0*Z(3), Z(3), 0*Z(3) ] ] ]
gap> Order(G);
1624314979123200
gap> "SL(d,q)";
"SL(d,q)"
gap> G := SL(4,9);
SL(4,9)
gap> gens := GeneratorsOfGroup(G);
[ [ [ Z(3^2), 0*Z(3), 0*Z(3), 0*Z(3) ], [ 0*Z(3), Z(3^2)^7, 0*Z(3), 0*Z(3) ],
      [ 0*Z(3), 0*Z(3), Z(3)^0, 0*Z(3) ], [ 0*Z(3), 0*Z(3), 0*Z(3), Z(3)^0 ] ]
    , [ [ Z(3), 0*Z(3), 0*Z(3), Z(3)^0 ], [ Z(3), 0*Z(3), 0*Z(3), 0*Z(3) ],
      [ 0*Z(3), Z(3), 0*Z(3), 0*Z(3) ], [ 0*Z(3), 0*Z(3), Z(3), 0*Z(3) ] ] ]
gap> Order(G);
203039372390400
gap> quit;
```

1.1.6 Normaaldelers en quotiëntgroepen

Stel dat G een groep is. Een deelgroep $H \leqslant G$ wordt een *normaaldeler* (**E:** *normal subgroup*) genoemd als en slechts als $g^{-1}hg \in H$ voor alle $h \in H$ en alle $g \in G$. Wanneer we H^g definiëren als de verzameling $\{g^{-1}hg | h \in H\}$ dan is H een normaaldeler als en slechts als $H^g = H$ voor alle $g \in G$. Als H een normaaldeler is van G dan noteren we $H \trianglelefteq G$. Stel dat $N \trianglelefteq G$ en beschouw twee nevenklassen Ng en Nh. Omdat N een normaaldeler is, geldt voor elke twee $n_1, n_2 \in N$ dat $n_1 g n_2 h$ een element is uit de nevenklasse Ngh. Hierdoor kunnen we het product van twee nevenklassen van N definiëren als $(Ng)(Nh) = Ngh$ en de verzameling van nevenklassen van G is een groep met het gedefinieerde product: de *quotiëntgroep* (**E:** *quotient group*), genoteerd als G/N. Een groep G noemen we *enkelvoudig* (**E:** *simple*) als de enige normaaldelers van G, $\{\mathbf{1}\}$ en G zelf zijn. Beschouw tenslotte een deelverzameling $A \subset G$. De *normale sluiting* (**E:** *normal closure*) van A in G, genoteerd als $\langle A^G \rangle$, is de doornsede van alle normaaldelers van G die A als verzameling bevatten.

1.2 Bepaling van alle elementen van een groep

Stel dat G een groep is en dat we een verzameling S van generatoren voor de groep G kennen. Hoe kunnen we alle elementen van G bepalen en opslaan in een lijst? We veronderstellen uiteraard dat de lijst in het geheugen van de computer past, we maken ons geen zorgen over technische aspecten. We weten dat:

(a) G heeft een eenheidselement (en we kennen dat)

(b) $S \subset G$

(c) de groep is gesloten onder vermenigvuldiging (de bewerking is intern!). Dus als $g, h \in G$, dan weten we dat $g \cdot h \in G$.

(d) G is dus de kleinste verzamling die S bevat en is gesloten onder vermenigvuldiging.

Aan de hand van deze, weliswaar eenvoudige, structurele informatie over de groep G, kunnen we een algoritme (Algoritme 1.1) opstellen om alle elementen van G op te sommen. In pseudocode ziet het er als volgt uit (waarbij we de notatie uit [1] volgen)

Algoritme 1.1 bepaling van alle elementen van een groep (1)

List(S)
 Invoer: een verzameling generatoren S voor G
 Uitvoer: alle elementen van G
1 $list := \{\mathbf{1}\} \cup S$
2 **for** alle paren $(g, h) \in list \times list$
3 **do if** $gh \notin list$
4   **then** voeg gh toe aan $list$
5 **return** $list$

We merken op dat deze pseudocode kan aanleiding geven tot verwarring. De variabele *list*, waar de for-lus afhankelijk van is, wordt in de for-lus zelf gewijzigd (lijn 4). Technisch gezien zullen de meeste compilers ofwel een error geven ofwel deze wijziging negeren. De implementatie (files: `magma1.2.1` en `gap1.2.1`) in MAGMA en GAP wordt, met initialisatie van een voorbeeld groep G:

```
G<a,b>:=PermutationGroup<6|(2,3,4,5,6,1),(3,4,5,6,2)>;

elements:={Id(G)} join Generators(G);
orde:=#elements;

repeat
  vorigeorde:=orde;

  for x in elements do
    for y in elements do

      prod:=x*y;

      if not (prod in elements) then
         elements:=Include(elements,prod);
      end if;

    end for;
  end for;

  orde:=#elements;
until vorigeorde eq orde;
#elements;

G := Group([(2,3,4,5,6,1),(3,4,5,6,2)]);
elements := Union([One(G)],GeneratorsOfGroup(G));
orde := Length(elements);
repeat
  vorigeorde := orde;
  for x in elements do
    for y in elements do
      prod := x*y;
      if not (prod in elements) then
        Add(elements,prod);
      fi;
    od;
  od;
  orde := Length(elements);
until (vorigeorde=orde);
Length(elements);
```

Ga zelf na dat de uitvoering van dit algoritme met de gegeven groep als voorbeeld reeds enige ogenblikken in beslag neemt, ondanks het feit dat de gegeven groep in feite een zeer kleine groep is.

Er is een onmiddellijke verbetering mogelijk van Algoritme 1.1. We hebben namelijk niet ten volle benut dat elk element van $g \in G$ kan geschreven worden als een product van uitsluitend generatoren. We weten dus dat $g = s_{i_1} \cdot s_{i_2} \cdot \ldots \cdot s_{i_m}$. Elk element dat verkregen kan worden door een product van m factoren kunnen we beschouwen als een product van een element g' dat verkregen kan worden door een product van $m-1$ factoren, vermenigvuldigd met een generator s_{i_m}. We hoeven dus enkel rekening te houden met de paren (g, h), waarbij h een generator is. Deze aanpassing laat zich onmiddellijk uitvoeren:

Algoritme 1.2 bepaling van alle elementen van een groep (2)

LIST2(S)

Invoer: een verzameling generatoren S voor G

Uitvoer: alle elementen van G

1 $list := \{\mathbf{1}\} \cup S$
2 **for** alle $g \in list$
3 **do for** alls $s \in S$
4 **do if** $gs \notin list$
5 **then** voeg gs toe aan $list$
6 **return** $list$

De aangepaste implementatie (files: `magma1.3.2` en `gap1.3.2`) in MAGMA en GAP wordt, met initialisatie van een voorbeeld groep G:

```
G<a,b>:=PermutationGroup<6|(2,3,4,5,6,1),(3,4,5,6,2)>;

gens := Generators(G);
elements:={Id(G)} join gens;
orde:=#elements;

repeat
  vorigeorde := orde;

  for x in elements do
    for y in gens do
      prod := x*y;

      if not (prod in elements) then
         elements:=Include(elements,prod);
      end if;

    end for;
  end for;

  orde := #elements;
until vorigeorde eq orde;
#elements;


G := Group([(2,3,4,5,6,1),(3,4,5,6,2)]);
gens := GeneratorsOfGroup(G);
elements := Union([One(G)],gens);
orde := Length(elements);
repeat
  vorigeorde := orde;
  for x in elements do
```

```
    for y in gens do
      prod := x*y;
      if not (prod in elements) then
        Add(elements,prod);
      fi;
    od;
  od;
  orde := Length(elements);
until (vorigeorde=orde);
Length(elements);
```

Ga zelf na dat deze schijnbaar eenvoudige aanpassing reeds een significante versnelling van de uitvoering teweegbrengt. We zien dus duidelijk dat het gebruik van de structurele kennis de algoritmen kan verbeteren.

1.3 Het algoritme van Dimino

In algoritme 2 werden de elementen van een groep gevonden, die geschreven werden als een product van m factoren. In feite is er een inductie op de lengte van het product $s_{i_1} \cdot s_{i_2} \cdot \ldots \cdot s_{i_m}$. In plaats daarvan kunnen we ook, op inductieve wijze, alle elementen gaan opbouwen die gevormd worden door het product van een aantal vast gekozen generatoren. Wanneer we al deze elementen bepaald hebben, dan voegen we de volgende generator toe. Op deze wijze maken we gebruik van de deelgroepen van de groep G. Voorts kunnen we dan ook de nevenklassen in beschouwing nemen.

Stel dat de verzameling generatoren gegeven wordt door $S = \{s_1, s_2, \ldots, s_t\}$. We definiëren $S_i := \{s_1, s_2, \ldots, s_i\}$, en $H_i := \langle S_i \rangle$ de deelgroep voortgebracht door S_i. We definiëren $S_0 := \emptyset$ en H_0 als de triviale groep. De inductieve stap in het nieuwe algoritme, dat het *algoritme van Dimino* heet, is het bepalen van de elementen van H_i als de elementen van H_{i-1} gegeven zijn. In principe kunnen we algoritme 2 reeds aanpassen, door tijdens deze inductieve stap alleen producten van de vorm $g \cdot s$ te beschouwen met ofwel $g \in H_{i-1}$ en $s = s_i$ ofwel $g \notin H_{i-1}$ en $s \in S_i$. Nog steeds hebben we dan echter geen gebruik gemaakt van de volgende feiten over nevenklassen:

(e) De nevenklassen van een deelgroep partitioneren de groep

(f) Alle nevenklassen bevatten evenveel elementen als de deelgroep. Alle elementen van een nevenklasse vinden we door de elementen van de deelgroep te vermenigvuldigen met de nevenklasserepresentant.

Tijdens de inductieve stap (van H_{i-1} naar H_i) kunnen we dus alle elementen van de nevenklasse $H_{i-1}g$ toevoegen aan de lijst van zodra we een element g vinden dat nog niet in de lijst zat, zodat we, bij het vinden van een nieuw element onmiddellijk $|H_{i-1}|$ elementen kunnen toevoegen. Anderzijds zal dit niet noodzakelijk een zeer grote verbetering inhouden. We moeten immers nog steeds veel producten van de vorm $g \cdot s$ onderzoeken om een nieuw element te vinden, dus in feite hebben we niet noodzakelijk het zoekwerk verminderd. Indien we zeker willen zijn dat het toevoegen van een hele

nevenklasse nuttig wordt, dan moeten we in feite een efficiënte manier vinden om nieuwe nevenklasserepresentanten te vinden en het volgende lemma stelt ons daartoe in staat.

Lemma 1.3.1. *Stel dat H een deelgroep is van G. Stel dat Hg een nevenklasse is van H in G en stel dat $s \in G$. Dan behoren alle elementen van $(Hg)s$ tot de nevenklasse van H met nevenklasserepresentant gs.*

Bewijs. bewijs als oefening ■

Het onmiddellijk gevolg van dit lemma is dat we nevenklassrepresentanten van H_{i-1} kunnen vinden door producten van de vorm $g{\cdot}s$ te beschouwen met g een eerder gevonden nevenklasserepresentant en $s \in S_i$. Het is duidelijk dat we dan veel minder producten zullen moeten berekenen om nieuwe nevenklasserepresentanten te vinden. We kunnen deze inductieve stap omzetten in een algoritme:

Algoritme 1.3 inductieve stap in het algoritme van Dimino

```
INDUCTIVESTEP(S, H_{i-1})
    Invoer: een verzameling generatoren S voor G
    een lijst met elementen van H_{i-1}
    Uitvoer: een lijst met alle elementen van H_i
1   coset_reps := {id}
2   for alle g ∈ coset_reps
3        do for alle generatoren s ∈ S_i
4               do if g · s behoort niet tot list
5                     then voeg g · s toe aan coset_reps
6                          voeg alle elementen van H_{i-1}(g · s) toe aan list
```

Om het algoritme volledig op te stellen moeten we enkel nog de basisstap uitvoeren: bepaal alle elementen van een groep die voortgebracht worden door 1 generator (m.a.w., bepaal alle elementen van een cyclische groep). Het is duidelijk dat deze stap in 1 lus kan uitgevoerd worden. We kunnen nu het volledig algoritme opstellen, dit wordt het vereenvoudigd algoritme van Dimino genoemd (Algoritme 1.4).

Algoritme 1.4 het vereenvoudigd algoritme van Dimino

```
DIMINOSIMPLIFIED(S)
     Invoer: een verzameling generatoren S = {s_1, s_2, ..., s_t} voor G
     Uitvoer: een lijst met alle elementen van G
     (de eerste stap is de cyclische groep H_1)
 1   orde := 1; elements[1] := id;
 2   g := s_1
 3   while not (g = id)
 4         do orde := orde + 1;
 5            elements[orde] := g;
 6            g := g · s_1;
     (nu start de behandeling van de volgende, inductieve stappen.)
 7   for i := 2 to t
 8         do if not s_i ∈ elements
 9               then (de volgende generator zou overbodig kunnen zijn)
10                     vorige_orde := orde; (i.e. |H_{i−1}|)
                       (de eerste nevenklasserepresentant is zeker s_i, voeg meteen de nevenklasse toe)
11                     orde := orde + 1;
12                     elements[orde] := s_i;
13                     for j := 2 to vorige_orde
14                           do orde := orde + 1;
15                              elements[orde] := elements[j] · s_i;
                       (houd de positie van de nevenklasserepresentant bij)
16                     rep_pos := vorige_orde + 1;
17                     repeat for alle s ∈ S_i
18                           do elt := elements[rep_pos] · s;
19                              if not elt ∈ elements
20                                 then (voeg nevenklasse toe)
21                                       orde := orde + 1;
22                                       elements[orde] := elt;
23                                       for j := 2 to vorige_orde
24                                             do orde := orde + 1;
25                                                elements[orde] := elements[j] · elt;
                              (houd de positie van de volgende nevenklasserepresentant bij)
26                     rep_pos := rep_pos + vorige_orde;
27                     until rep_pos > orde;
```

We geven de implementatie van dit algoritme in MAGMA en GAP (files: `magma1.3`

en `gap1.3`) en de bijhorende sessies (files: `magma1.3.i` en `gap1.3.i`) waarin het algoritme wordt uitgevoerd. Bemerk hoe we zowel in MAGMA als GAP files kunnen inlezen (`load` en `Read`).

De implementatie in MAGMA:

```
t:=Ngens(G);
orde:=1;
elements:=[Id(G)];
g:=G.1;

while not (g eq Id(G)) do
  orde:=orde+1;
  elements[orde]:=g;
  g:=g*G.1;
end while;

for i:=2 to t do
  if Position(elements,G.i) eq 0 then
    vorigeorde:=orde;
    orde:=orde+1;
    elements[orde]:=G.i;

    for j:=2 to vorigeorde do
      orde:=orde+1;
      elements[orde]:=elements[j]*G.i;
    end for;

    rep_pos:=vorigeorde+1;

    repeat

      for s:=1 to i  do
        elt:=elements[rep_pos]*G.s;

        if Position(elements,elt) eq 0 then
           orde:=orde+1;
           elements[orde]:=elt;

           for j:=2 to vorigeorde do

             orde:=orde+1;

             elements[orde]:=elements[j]*elt;

           end for;
        end if;
      end for;

      rep_pos:=rep_pos+vorigeorde;

     until rep_pos gt orde;
   end if;
end for;
```

```
Magma V2.11-13    Tue Sep 27 2005 19:37:01 on Computer-va [Seed = 1295486258]
Type ? for help.  Type <Ctrl>-D to quit.
> G<a,b>:=PermutationGroup<6|(2,3,4,5,6,1),(3,4,5,6,2)>;
> load "magma1.3";
Loading "magma1.3"
> #elements;
720
> elements[1];
Id(G)
> elements[2];
(1, 2, 3, 4, 5, 6)
> elements[720];
(1, 5)(2, 4)
> quit;

Total time: 0.899 seconds, Total memory usage: 12.98MB
```

De implementatie in GAP (met weglating van de “GAP4” header vanaf nu:

```
orde := 1;
elementen := [One(G)];
```

```
gens := GeneratorsOfGroup(G);
g := gens[1];

while not g = One(G) do
  orde := orde + 1;
  elementen[orde] := g;
  g := g*gens[1];
od;

for i in [2..Length(gens)] do
  if not gens[i] in elementen then
    vorde := orde;
    orde := orde + 1;
    elementen[orde] := gens[i];

    for j in [2..vorde] do
      orde := orde + 1;
      elementen[orde] := elementen[j]*gens[i];
    od;

    pos := vorde + 1;

    repeat

      for j in [1..i] do
        elt := elementen[pos] * gens[j];

        if not elt in elementen then
          orde := orde + 1;
          elementen[orde] := elt;

          for k in [2..vorde] do
            orde := orde + 1;
            elementen[orde] := elementen[k]*elt;
          od;

        fi;

      od;

      pos := pos + vorde;

    until pos > orde;

  fi;

od;
```

```
     Information at:  http://www.gap-system.org
     Try '?help' for help. See also  '?copyright' and  '?authors'

   Loading the library. Please be patient, this may take a while.
GAP4, Version: 4.4.6 of 02-Sep-2005, powerpc-apple-darwin8.2.0-gcc
Components:  small 2.1, small2 2.0, small3 2.0, small4 1.0, small5 1.0,
             small6 1.0, small7 1.0, small8 1.0, small9 1.0, small10 0.2,
             id2 3.0, id3 2.1, id4 1.0, id5 1.0, id6 1.0, id9 1.0, id10 0.1,
             trans 1.0, prim 2.1  loaded.
Packages:    TomLib 1.1.2  loaded.
gap> G := Group([(2,3,4,5,6,1),(3,4,5,6,2)]);
Group([ (1,2,3,4,5,6), (2,3,4,5,6) ])
gap> Read("gap1.3");
gap> time;
9
gap> Length(elementen);
720
gap> elementen[1];
()
gap> elementen[2];
(1,2,3,4,5,6)
gap> elementen[720];
(1,5)(2,4)
gap> quit;
```

Het volledig algoritme van Dimino

Stel dat $H \leqslant G$ een normaaldeler is van G. Als $G = \langle H, g \rangle$, dan worden de nevenklasserepresentanten van H in G gegeven door $\{id, g, g^2, \ldots, g^m\}$, met m het kleinste

natuurlijk getal waarvoor $g^{m+1} \in H$. Hierdoor kan het vereenvoudigd algoritme van Dimino aangepast worden, we hebben nog meer informatie om nieuwe nevenklasserepresentanten te bepalen. Anderzijds moeten we, in elke inductieve stap testen of H_{i-1} een normaaldeler is van H_i. Dit leidt dan weer tot extra berekeningen. Een vergelijkende analyse tussen het vereenvoudigd algoritme en het volledig algoritme kan men vinden in [1].

1.4 Toepassing: eenvoudige zoekalgoritmen voor kleine groepen

In deze paragraaf geven we enkele eenvoudige algoritmen om elementen of deelgroepen van kleine groepen te zoeken. Vaak worden er in de computationele groepentheorie algoritmen ontworpen om op zoek te gaan naar elementen van een groep die een bepaalde eigenschap hebben. Een triviale manier om dit te doen is alle elementen van de groep overlopen en die elementen opslaan in een lijst die aan een gegeven eigenschap voldoen. We kunnen ons echter vragen stellen bij de efficiëntie van deze aanpak. Toch geven we hiervan een illustratie. We zullen nadien enkele voorbeelden geven toegepast op kleine groepen. Uiteindelijk besluiten deze voorbeelden op het inleidend hoofdstuk, waarin we aantonen dat minimale abstracte kennis ons reeds in staat stelt algoritmen te ontwerpen.

Gegeven een groep G, dan willen we een element (en uitgebreider: alle elementen) vinden dat aan een gegeven eigenschap P voldoet. Indien een element $g \in G$ aan die eigenschap voldoet, dan noteren we dat met $P(g)$. We gaan ervan uit dat we in staat zijn om te controleren of g aan P voldoet. We beschrijven een algoritme dat 1 element van een groep vindt dat een aan een specifieke eigenschap voldoet.

Algoritme 1.5 zoeken in een lijst naar 1 element

SEARCH1($list, P$)

Invoer: een lijst die alle elementen van een groep G bevat, een eigenschap P
Uitvoer: een element $g \in G$ waarvoor $P(g)$

1 **for** alle elementen $g \in list$
2   **do if** $P(g)$
3   **then return** g

Een eenvoudig voorbeeld illustreert het gebruik van Algoritme 1.5. Stel dat G een groep is en $h \in G$ een willekeurig element. We zeggen dat een element $g \in G$ het element h *centraliseert* als en slechts als $gh = hg$. De verzameling van alle elementen $g \in G$ die een gegeven element h centraliseren vormt een deelgroep van G, genoteerd met

$C_h(G)$. We noemen deze deelgroep de *centralisator* van h in G. Het volgende voorbeeld is een eenvoudige implementatie van Algoritme 1.5 in GAP (file: `gap1.4.1.i`). Merk wel op dat we de elementen van de cyclische groep voortgebracht door het element h zelf, uitsluiten. Dit is in principe ook een slechte aanpak, want het is mogelijk dat de centralisator van een element h net deze groep zelf is.

```
    Information at:  http://www.gap-system.org
    Try '?help' for help. See also  '?copyright' and  '?authors'

   Loading the library. Please be patient, this may take a while.
GAP4, Version: 4.3fix3 of September 12, 2002, i686-pc-linux-gnu-gcc
Components:  small, small2, small3, small4, small5, small6, small7,
             small8, id2, id3, id4, id5, id6, trans, prim  loaded.
Packages:    tomlib, ctbllib  loaded.
gap> G := Group([(2,3,4,5,6,1),(3,4,5,6,2)]);
Group([ (1,2,3,4,5,6), (2,3,4,5,6) ])
gap> h := (1,2,5,3);
(1,2,5,3)
gap> h in G;
true
gap> H := Group([h]);
Group([ (1,2,5,3) ])
gap> list := Difference(List(G),List(H));;
gap> found := false;
false
gap> i := 1;
1
gap> while not found and i < Length(list) do
  g := list[i];
  if g*h = h*g then
    found := true;
  fi;
  i := i+1;
od;
gap> found;
true
gap> g;
(4,6)
gap> g*h;
(1,2,5,3)(4,6)
gap> h*g;
(1,2,5,3)(4,6)
gap> quit;
```

We kunnen het zoeken van een element dat aan een bepaalde eigenschap voldoet ook andersom bekijken. Bij de start van de zoektocht zijn alle elementen van G een potentiële oplossing. Indien een element niet voldoet, dan wordt het uit de (lange) lijst verwijderd. Indien we een element vinden, dan wordt de lus gestopt en het element teruggegeven, zoals in Algoritme 1.5. Deze redenering vertaalt zich onmiddellijk in het volgende algoritme:

Algoritme 1.6 het verwijderen van elementen die niet voldoen

```
SEARCH2(list, P)
    Invoer: een lijst die alle elementen van een groep G bevat, een eigenschap P
    Uitvoer: een element g ∈ G waarvoor P(g)
1   Γ := G;
2   while Γ ≠ ∅
3         do g := Random(Γ)
4            if P(g)
5               then return g
6               else  Γ := Γ \ {g}
```

In deze vorm is er weinig verschil tussen Algoritme 1.6 en Algoritme 1.5. Het wordt echter interessant als we, bij het vinden van een element dat niet aan de eigenschap voldoet, meerdere elementen tegelijk uit de lijst kunnen verwijderen, zodat het testen van de eigenschap voor al deze elementen vermeden kan worden. Dit zal enkel winst opleveren als we uit het vinden van een element dat niet aan de eigenschap voldoet "gemakkelijk" andere elementen kunnen vinden die niet voldoen. Gemakkelijk is hierbij relatief ten opzichte van het testen van de eigenschap voor elementen. Dit zal natuurlijk het gemakkelijkst gaan als we uit het niet voldoen van een element g aan eigenschap P, zonder globale kennis van G, de elementen $D_P := \{g \in G \| \sim P(g)\}$ kunnen bepalen. We illustreren dit aan de hand van een voorbeeld. We beschouwen nog steeds het voorbeeld van de centralisator van een element $h \in G$. Stel dat $g \in G$ het element h niet centraliseert, dus $P(g)$ geldt niet. Dan kunnen we eenvoudig inzien dat $P(g^{-1})$, $P(hg)$, $P(gh)$, $P(h^m g)$ en $P(gh^m)$ (voor alle $m \in \mathbb{N}$) en $P(h^m \cdot g \cdot h^n)$, voor alle $n, m \in \mathbb{N}$ niet gelden. Er is echter geen eenvoudige manier om na te gaan of $P(y)$ geldt, waarbij $y^m = g$, voor een bepaalde $m \in \mathbb{N}$. We kunnen in dit voorbeeld voor $D_p(g)$ de nevenklasse $\langle h \rangle g$ nemen. Het algoritme wordt dan:

Algoritme 1.7 het verwijderen van nevenklassen die niet voldoen

SEARCH3($list, h$)

Invoer: een lijst die alle elementen van een groep G bevat, een element $h \in G$
Uitvoer: een element $g \in G$ dat h centraliseert.

```
1  Γ := G \ ⟨h⟩;
2  while Γ ≠ ∅
3        do g := Random(Γ)
4           if P(g)
5              then return g
6              else  D_P(g) := ⟨h⟩g;
7                    Γ := Γ \ D_P(g);
```

Tenslotte kunnen we een algoritme ontwerpen dat ons *alle* elementen die aan een gegeven eigenschap voldoen, teruggeeft. Ook hier zien we dat, als de verzameling van die elementen een deelgroep is, de efficiëntie van het algoritme verhoogt. Alle elementen die commuteren met een gegeven element z vormen een deelgroep (de centralisator van z). Indien we een element g vinden dat commuteert met z, dan kunnen we meteen $\langle g \rangle$ toevoegen aan de verzameling van oplossingen, of, nog beter, we kunnen de bestaande deelgroep H van oplossingen vervangen door $\langle H, g \rangle$. Deze redenering vertaalt zich onmiddellijk:

Algoritme 1.8 alle elementen die aan een eigenschap voldoen

SEARCH4($list, h$)

Invoer: een lijst die alle elementen van een groep G bevat, een element $h \in G$
Uitvoer: De groep $C_h(G)$.

```
 1  H := {id};
 2  Γ := G \ H;
 3  while Γ ≠ ∅
 4        do g := Random(Γ)
 5           if P(g)
 6              then H := ⟨H, g⟩;
 7                   Γ := Γ \ H;
 8              else  D_P(g) := ⟨h⟩g;
 9                    Γ := Γ \ D_P(g);
10  return H;
```

We geven een implementatie in GAP van een functie `centralizer`: (file: `gap1.4.2`).

```
G := Group([(2,3,4,5,6,1),(3,4,5,6,2)]);
h := (1,2,5,3);

centralizer := function(G,h)
local gamma,H,gens,dpg,cyclic,g;
H := Group([()]);
cyclic := Group([h]);
gamma := Difference(List(G),List(H));;
while Length(gamma) > 0 do
  g := Random(gamma);
  if g*h = h*g then
    gens := ShallowCopy(GeneratorsOfGroup(H));
    Add(gens,g);
    H := Group(gens);
    gamma := Difference(List(G),List(H));;
  else
    dpg := RightCoset(cyclic,g);
    gamma := Difference(gamma,dpg);
  fi;
od;
return H;
end;
```

We testen de implementatie: (file: `gap1.4.2.i`).

```
     Information at:  http://www.gap-system.org
     Try '?help' for help. See also  '?copyright' and  '?authors'

   Loading the library. Please be patient, this may take a while.
GAP4, Version: 4.4.6 of 02-Sep-2005, powerpc-apple-darwin8.2.0-gcc
Components:  small 2.1, small2 2.0, small3 2.0, small4 1.0, small5 1.0,
             small6 1.0, small7 1.0, small8 1.0, small9 1.0, small10 0.2,
             id2 3.0, id3 2.1, id4 1.0, id5 1.0, id6 1.0, id9 1.0, id10 0.1,
             trans 1.0, prim 2.1  loaded.
Packages:    TomLib 1.1.2  loaded.
gap> G := Group([(2,3,4,5,6,1),(3,4,5,6,2)]);
Group([ (1,2,3,4,5,6), (2,3,4,5,6) ])
gap> h := (1,2,5,3);
(1,2,5,3)
gap> Read("gap1.4.2");
gap> Cgh := centralizer(G,h);
Group([ (), (1,2,5,3), (1,5)(2,3)(4,6) ])
gap> C := Centralizer(G,h);
Group([ (1,2,5,3), (4,6) ])
gap> C = Cgh;
true
gap> quit;
```

1.5 Oefeningen

1. Definieer enkele groepen in GAP en MAGMA:

   - de cyclische groep van orde 4.
   - viergroep van Klein: werkt in op 4 elementen, heeft generatoren $(1,2)(3,4)$ en $(1,4)(2,3)$.
   - De automorfismegroep van het Petersen graaf: werkt in op 10 elementen, heeft generatoren $(4,8)(5,6)(9,10)$, $(2,5)(3,4)(7,10)(8,9)$, $(1,3,5,2,4)(6,8,10,7,9)$
   - De groep PGL$(2,2)$: werkt in op 7 elementen, heeft generatoren $(1,4)(6,7)$, $(1,3,2)(4,7,5)$.

Onderzoek of C_4 een normaaldeler is van de viergroep van Klein. Onderzoek of PGL$(2, 2)$ enkelvoudig is. Een groep is enkelvoudig als en slechts als hij geen niet-triviale normaaldelers heeft.

2. Bekijk de files `magma1.2.1` en `magma1.2.2` en `gap1.2.1` en `gap1.2.2`. Gebruik de twee algoritmes in de files om alle elementen van een groep te berekenen. Gebruik de groepen uit de vorige opgave. Gebruik ook als voorbeeld de alternerende groep A_7. Ga na dat het tweede algoritme veel sneller alle elementen oplevert dan het eerste. Gebruik tenslotte het algoritme van Dimino `magma1.3` en `gap1.3`.

3. Voer het vereenvoudigd algoritme van Dimino met verschillende kleine groepen als voorbeeld uit in MAGMA en GAP. Bekijk de rekentijden.

4. Implementeer algoritme 5 en algoritme 8 in MAGMA. Gebruik hetzelfde voorbeeld als in 1.4.

Hoofdstuk 2

Acties van groepen, Cayleygraaf en definiërende relaties van een groep

In dit hoofdstuk beschrijven we een representatie van (kleine) groepen aan de hand van grafen. Gebruik makend van deze representatie kunnen we, voor een groep voortgebracht door een gegeven verzameling van generatoren, de zogenaamde definiërende relaties bepalen.

2.1 Definities

2.1.1 Homomorfismen en isomorfiestellingen

Stel dat G en H twee groepen zijn. Een *homomorfisme* (**E:** *homomorphism*) ϕ van G naar H is een afbeelding $\phi : G \mapsto H$ zo dat $\phi(g_1g_2) = \phi(g_1)\phi(g_2)$ voor alle elementen $g_1, g_2 \in G$. Daaruit volgt onmiddellijk dat een homomorfisme ϕ $\mathbf{1_G}$ afbeeldt op $\mathbf{1_H}$ en dat $\phi(g^{-1}) = \phi(g)^{-1}$ voor alle elementen $g \in G$. Een homomorfisme noemen we repectievelijk een *monomorfisme* (**E:** *monomorphism*), een *epimorfisme* (**E:** *epimorphism*) of een *isomorfisme* (**E:** *isomorphism*) als, respectievelijk, een injectie, een surjectie of een bijectie is. Een homomorfisme van G naar G wordt een *endomorfisme* (**E:** *endomorphism*) van G genoemd, een isomorfisme van G naar G wordt een *automorfisme* (**E:** *automorphism*) genoemd. De automorfismen van G, met de samenstelling ervan, vormen een groep, genoteerd met $\mathrm{Aut}(G)$. Het is duidelijk dat $\mathrm{Aut}(G) \leqslant \mathrm{Sym}(G)$ en daarom noteren we de werking van een element $\alpha \in \mathrm{Aut}(G)$ op een element $g \in G$ met g^α. Voor elk element $k \in G$, is de afbeelding $\alpha_k : G \to G$, gedefinieerd als $g^{\alpha_k} = k^{-1}gk$, een automorfisme van G. Een automorfisme van dit type wordt een *inwendig automorfisme* (**E:** *inner automorphism*) genoemd. De verzameling van alle inwendige automorfismen vormt een normaaldeler van $\mathrm{Aut}(G)$ en wordt genoteerd met $\mathrm{Inn}(G)$. Een automorfisme dat niet inwendig is, wordt *uitwendig* (**E:** *outer*) genoemd. De groep van uitwendige

automorfismen is bij definitie de quotiëntgroep $\mathrm{Aut}(G)/\mathrm{Inn}(G)$. Twee groepen worden *isomorf* (**E:** *isomorphic*) genoemd, genoteerd $G \cong H$, als en slechts als er een isomorfisme $\phi : G \to H$ bestaat. De inverse van een isomorfisme is opnieuw een isomorfisme, dus $G \cong H \iff H \cong G$. Isomorfe groepen worden in de meeste omstandigheden als identiek beschouwd. We spreken bijvoorbeeld over *de* cyclische groep van orde n, of *de* oneindige cyclische groep. Analoog spreken we over *de* symmetrische groep S_n als we het over de groep $\mathrm{Sym}(\Omega)$ hebben, met $|\Omega| = n$.

Stel dat $\phi : G \to H$ een groepshomomorphisme is. De *kern* (**E:** *kernel*) van ϕ, genoteerd $\ker(\phi)$, is gedefinieerd als de verzameling elementen van G die door ϕ op $\mathbf{1}_H$ afgebeeld worden. Bewijs als oefening dat $\ker(\phi)$ een normaaldeler is van G. We vermelden nu drie isomorfiestellingen, waarvan de eerste als de belangrijkste beschouwd wordt.

Stelling 2.1.1. *Stel dat $\phi : G \to H$ een groepshomomorfisme is met kern K. Dan geldt: $G/K \cong \mathrm{im}(\phi)$. Er bestaat dus een isomorfisme $\bar{\phi} : G/K \to \mathrm{im}(\phi) : \bar{\phi}(Kg) = \phi(g)$ voor alle elementen $g \in G$*

Stelling 2.1.2. *Beschouw een willekeurige groep G en stel dat $K \trianglelefteq G$ en $H \leqslant G$. Dan is $H \cap K \trianglelefteq H$ en $H/(H \cap K) \cong HK/K$.*

Stelling 2.1.3. *Beschouw een willekeurige groep G, stel dat $K \subseteq H \subseteq G$ en $H \trianglelefteq G$ en $K \trianglelefteq G$. Dan geldt: $(G/K)/(H/K) \cong G/H$.*

Veronderstel nu dat α_k een inwendig automorfisme van G is, zoals hoger gedefinieerd. Beschouw de afbeelding $\phi : G \to \mathrm{Aut}(G) : \phi(g) \mapsto \alpha_g$. Dan is ϕ een homomorfisme van G naar $\mathrm{Aut}(G)$ en $\ker(\phi) = \{k \in G | k^{-1}gk = g, \forall g \in G\}$, een verzameling die ook wel het *centrum* (**E:** *centre*) van de groep G genoemd wordt, genoteerd als $Z(G)$[1]. Door de eerste isomorfiestelling kunnen we besluiten dat $G/Z(G) \cong \mathrm{Inn}(G)$.

Stel dat G en M twee groepen zijn, en dat er een homomorfisme $\varphi : G \to \mathrm{Aut}(M)$ bestaat. Het *semidirect produkt* (**E:** *semidirect product*) is de verzameling $G \times M$, voorzien van de volgende vermenigvuldiging: $(g, m)(h, n) := (gh, m^{\varphi(h)}n)$, voor alle $g, h \in G$ en alle $m, n \in M$. Deze groep wordt genoteerd met $G \ltimes M$ of $G \ltimes_\varphi M$.

2.1.2 Acties van groepen

Definitie 2.1.4. *Stel dat G een groep is en Ω een verzameling. Een* actie *(***E:*** action) van G op Ω is een homomorfisme $\phi : G \to \mathrm{Sym}(\Omega)$.*

Als er een actie van G op Ω bestaat, dan spreken we ook soms van een *permutatievoorstelling* (**E:** *permutation representation*) (G, Ω). Uit de definitie volgt dat $\phi(g)$ een permutatie is van Ω, en dit voor elk element $g \in G$. We hebben reeds de notatie

[1]De "Z" is afkomstig van het Duitse "Zentrum"

voor de werking van elementen uit $\mathrm{Sym}(\Omega)$ op elementen uit Ω ingevoerd, en we blijven die gebruiken. Dus $\phi(g)$ is een permutatie, en het beeld van $\alpha \in \Omega$ onder $\phi(g)$ noteren we met $\alpha^{\phi(g)}$. Wanneer de actie vastligt, kunnen we $\alpha^{\phi(g)}$ ook noteren als α^g. De actie van G op Ω is dus een "rechtse" actie. Vanuit theoretisch standpunt is dit misschien ongebruikelijk, omdat functies die op objecten inwerken meestal links van het object genoteerd worden. Anderzijds komt dit wel overeen met de acties uit MAGMA en GAP, die standaard steeds "rechtse" acties zijn. Omwille van die overeenkomst zullen we onze notatie behouden. Analoog zullen we de actie van een matrix op een vector als een rechtse actie beschouwen. Een vector is dus steeds gerepresenteerd als een rijvector, een actie van een matrix op een vector is dan automatisch een rechtse actie. Opnieuw komt dit overeen met de situatie in MAGMA en GAP.

We definiëren de *kern* (**E:** *kernel*) van de actie ϕ als de kern van het homomorfisme ϕ, m.a.w. de kern van de actie is de verzameling $K = \{g \in G | \alpha^g = \alpha, \forall \alpha \in \Omega\}$, dit is een normaaldeler van G. We noemen de actie *getrouw* (**E:** *faithful*) als en slechts als K de triviale groep is. Stel dat er een getrouwe actie is van G op Ω, er bestaat dus een homomorfisme van G naar $\mathrm{Sym}(\Omega)$, dan definiëren we de *graad van de permutatievoorstelling* (**E:** *degree of permutation representation*) als $|\Omega|$, in de veronderstelling dat Ω een eindige verzameling is.

We geven nu een aantal voorbeelden.

Voorbeeld 1

Beschouw een willekeurige groep G, en stel dat Ω de verzameling van elementen van de groep G is. Voor een element $\alpha \in \Omega$ definiëren we $\alpha^g := \alpha g$ voor alle $g \in G$. Deze actie is getrouw en wordt de *rechts reguliere actie* (**E:** *right regular action*) genoemd. Dit principe kan veralgemeend worden als volgt. Stel dat $H \leqslant G$ een willekeurige deegroep is van G. Stel dat Ω de verzameling is van de rechtse nevenklassen van H in G. We definiëren opnieuw, voor een willekeurige $\alpha \in \Omega$, $\alpha^g := \alpha g$ voor alle $g \in G$. Een dergelijke actie noemen we een *nevenklasse actie* (**E:** *coset action*). De rechts reguliere actie correspondeert dan met de triviale deelgroep van G.

De rechts reguliere actie is nauw verwant met het zogenaamde *Cayleygraaf* van een groep.

Definitie 2.1.5. *Stel dat G een groep is die voortgebracht wordt door een verzameling van generatoren X. Het* Cayleygraaf *(***E:** Cayley graph*) $\Gamma_X(G)$ is een gericht gelabelled graaf met als toppenverzameling V de verzameling van alle elementen van G. Twee elementen $g, h \in V$ vormen een gerichte boog $[g, h]$, met label x, genoteerd $[g, h]_x$ als en slechts als er een element $x \in X$ bestaat waarvoor $gx = h$. Aldus is een verzameling van bogen eenduidig bepaald aan de hand van de verzameling van generatoren X. Het is gebruikelijk om, als $[g, h]_x$ een boog van $\Gamma_X(G)$ is, deze te identificeren met de boog $[h, g]_{x^{-1}}$.*

2.1.3 Banen en stabilisatoren

Stel dat de groep G werkt op een verzameling Ω. We definiëren een relatie $\sim$ op Ω als volgt: $\alpha \sim \beta$ als en slechts als er een element $g \in G$ bestaat zodat $\alpha^g = \beta$. Het is eenvoudig om in te zien dat $\sim$ een equivalentierelatie is. De equivalentieklassen van $\sim$ noemen we de *banen* (**E:** *orbits*) van G op Ω. In het bijzonder is de baan van een element $\alpha \in \Omega$ de verzameling $\{\alpha^g | g \in G\}$. We noteren deze baan met α^G.

De *stabilisator* (**E:** *stabilizer*) van α in G, genoteerd G_α of $\mathrm{Stab}_G(\alpha)$ is de verzameling $\{g \in G | \alpha^g = \alpha\}$. Bewijs als oefening dat G_α een deelgroep van G is.

Voorbeeld 2

Beschouw de rechtse nevenklasse actie van G op een deelgroep $H \leqslant G$. Het is duidelijk dat er maar één baan is voor deze actie. De stabilisator van de nevenklasse Hg is de deelgroep $g^{-1}Hg$, en de kern van de actie is gelijk aan de groep $\cap_{g \in G} g^{-1}Hg$. Deze deelgroep is een normaaldeler van G en wordt de *kern* (**E:** *core*) van H in G genoemd, genoteerd met H_G en is de grootste normaaldeler van G bevat in H.

Stelling 2.1.6. *(*orbit-stabilizer stelling*) Stel dat de eindige groep G werkt op de verzameling Ω, en stel dat $\alpha \in \Omega$. Dan geldt $|G| = |\alpha^G||G_\alpha|$.*

Bewijs. Stel $\beta \in \alpha^G$, m.a.w., er bestaat een $g \in G$, zodat $\beta = \alpha^g$. Stel dat $g' \in G$ en $\alpha^{g'} = \beta$, waaruit $g'g^{-1} \in G_\alpha$. Dus $g' \in G_\alpha g$ en hieruit volgt dat de elementen $g' \in G$ waarvoor $\alpha^{g'} = \beta$ juist de elementen van $G_\alpha g$ zijn. We weten dat $|G_\alpha g| = |G_\alpha|$, dus voor elke $\beta \in \alpha^G$ zijn er juist $|G_\alpha|$ elementen $g' \in G$ waarvoor $\alpha^g = \beta$, dus $|\alpha^G| = \frac{|G|}{|G_\alpha|}$. Indien we voor alle $\beta \in \alpha^G$ juist één $g \in G$ kiezen waarvoor $\alpha^g = \beta$, dan vormt de verzameling van deze elementen g een rechtse transversaal voor G_α. ■

Kiezen we een element $g \in G$, dan definiëren we $\Omega_g := \{\alpha \in \Omega | \alpha^g = \alpha\}$ als de verzameling van elementen van Ω die gefixeerd worden door g. Mogelijks is deze verzameling ledig! Het volgende lemma is gekend als het “Lemma van Burnside”, maar wordt ook toegeschreven aan Cauchy en Frobenius.

Lemma 2.1.7. *Stel dat de eindige groep G werkt op de verzameling Ω. Dan is het aantal banen van G op Ω gelijk aan $\frac{1}{|G|}\sum_{g \in G} |\Omega_g|$.*

We eindigen deze paragraaf met de volgende definities.

Definitie 2.1.8. *Stel dat de groep G werkt op de verzamling Ω en stel dat $\Delta \subseteq \Omega$.*

(i) De stabilisator *(***E:*** setwise stabilizer*) wordt gedefinieerd als de verzameling $\{g \in G | \alpha^g \in \Delta, \forall \alpha \in \Delta\}$ van Δ in G noteren we met G_Δ.*

(ii) De fixator *(***E:*** pointwise stabilizer*) wordt gedefinieerd als de verzameling $\{g \in G | \alpha^g = \alpha, \forall \alpha \in \Delta\}$ van Δ in G noteren we met $G_{(\Delta)}$.*

Het is eenvoudig in te zien dat $G_{(\Delta)} \leqslant G_\Delta \leqslant G$.

2.1.4 Toevoeging, normalisator en centralisator

De actie van een groep G op zichzelf gedefinieerd als $\alpha^g = g^{-1}\alpha g$, voor $g \in G$ en $\alpha \in \Omega = G$ wordt *toevoeging* (**E:** *conjugation*) genoemd. De banen van deze actie worden de *toevoegingsklassen* (**E:** *conjugacy classes*) van G genoemd. Twee elementen in de zelfde toevoegingsklasse worden *toegevoegd* (**E:** *conjugate*) genoemd. Twee elementen $g, h \in G$ zijn dus toegevoegd als en slechts als er een $f \in G$ bestaat waarvoor $h = f^{-1}gf$. We noteren de baan van g onder deze actie met $\mathrm{Cl}_\mathrm{G}(g)$, dit is dus de toevoeginsklasse die g bevat. Omdat $f^{-1}gf$ het beeld is van g onder het inwendig automorfisme van G bepaald door f, hebben de elementen uit een toevoegingsklasse dezelfde orde.

Kies een element $g \in G$. De stabilisator G_g, met betrekking tot de toevoeging, bestaat uit elementen $f \in G$ waarvoor $g^f = g$, wat gelijkwaardig is met $fg = gf$. Dus G_g is de verzameling van elementen $f \in G$ die commuteren met g. Deze verzameling wordt de *centralisator* (**E:** *centralizer*) van g in G genoemd, genoteerd als $C_G(g)$. Passen we de orbit-stabilizer stelling toe dan vinden we $|\mathrm{Cl}(g)| = \frac{|G|}{|C_G(g)|}$ voor alle $g \in G$. De kern van de toevoeging als actie is het centrum $Z(G)$.

Stellen we $\Omega = \{H | H \leqslant G\}$ dan kunnen we de definitie van toevoeging uitbreiden: $H^g := g^{-1}Hg$ voor alle $g \in G$ en alle $H \in \Omega$. Deelgroepen van G die tot de dezelfde baan behoren worden *toegevoegde deelgroepen* (**E:** *conjugate subgroups*) genoemd. De stabilisator van een deelgroep $H \leqslant G$ is de deelgroep $\{g \in G | g^{-1}Hg = H\}$. Deze deelgroep wordt de *normalisator* (**E:** *normalizer*) van H in G genoemd en genoteerd $N_G(H)$. Het is duidelijk dat $N_G(H) \trianglelefteq G$. Door Stelling 2.1.6 (orbit-stabilizer stelling) is het aantal toegevoegde deelgroepen van H in G gelijk aan $|G : N_G(H)|$.

Tenslotte definiëren we de centralisator van een deelgroep $H \leqslant G$ als de deelgroep $\{g \in G | gh = hg\}$ en noteren we deze groep als $C_G(H)$. Het is duidelijk dat $C_G(G) = Z(G)$.

2.1.5 Transitiviteit en primitiviteit

Een actie van een groep G op een verzameling Ω wordt *transitief* (**E:** *transitive*) genoemd als er juist één baan is onder deze actie. Bijgevolg is de definite gelijkwaardig met het bestaan van minstens één element $g \in G$, zodat $\alpha^g = \beta$, voor elke twee willekeurig gekozen elementen $\alpha, \beta \in \Omega$. Een actie niet niet transitief is wordt *intransitief* (**E:** *intransitive*) genoemd. Een actie wordt *n-voudig transitief* (**E:** *n-fold transitive*) genoemd (ook wel n-transitief), voor een $n > 0$, $n \in \mathbb{N}$ en $|\Omega| > n$, als voor elke twee **geordende** n-tupels $(\alpha_1, \ldots, \alpha_n)$ en $(\beta_1, \ldots, \beta_n)$ er minstens èèn $g \in G$ bestaat zodat $(\alpha_1^g, \ldots, \alpha_n^g) = (\beta_1, \ldots, \beta_n)$.

Beschouw als voorbeeld de groepen $\mathrm{Sym}(\Omega)$ en $\mathrm{Alt}(\Omega)$, $|\Omega| = n$, dewelke l-transitief $(1 \leq l \leq n)$, respectievelijk $(l-2)$-transitief $(3 \leq l \leq n)$ zijn.

Beschouw nu terug de rechtse nevenklasse actie van een groep G. De volgende stelling toont aan dat alle transitieve acties gelijkwaardig zijn met de rechtse nevenklasse actie

voor een bepaalde deelgroep $H \leqslant G$. Bewijs deze stelling als oefening.

Stelling 2.1.9. *(i) Beschouw een willekeurige deelgroep H van een groep G, en noteer met ϕ_H de actie van G op de rechtse nevenklassen van H in G door rechtse vermenigvuldiging. Dan is ϕ_H transitief.*

(ii) Stel dat $\phi : G \to \mathrm{Sym}(\Omega)$ een willekeurige transitieve actie van G op Ω is en kies $\alpha \in \Omega$. Dan is ϕ gelijkwaardig met ϕ_H, met $H = G_\alpha$.

(iii) Stel $H_1, H_2 \leqslant G$, dan zijn ϕ_{H_1} en ϕ_{H_2} gelijkwaardig als en slechts als H_1 en H_2 toegevoegd zijn in G.

Stel dat G transitief werkt op Ω. We zeggen dat G *regulier* (**E:** *regular*) werkt op Ω als $G_\alpha = \{\mathbf{1}\}$ voor één (en bijgevolg voor alle) $\alpha \in \Omega$.

De rechts reguliere actie van G op zichzelf uit Voorbeeld 1 is inderdaad regulier volgens bovenstaande definitie. Daarenboven volgt uit de orbit-stabilizer stelling dat, voor een eindige groep G, die regulier werkt op Ω, $|G| = |\Omega|$.

Stel nu dat G op Ω werkt. Een niet ledige deelverzameling $\Delta \subseteq \Omega$ wordt een *blok* (**E:** *block*) genoemd onder de actie als voor alle elementen $g \in G$ geldt dat $\Delta^g = \Delta$ of $\Delta^g \cap \Delta = \emptyset$. Een blok wordt niet-triviaal genoemd als $|\Delta| > 1$ en $\Delta \neq \Omega$. Het is duidelijk dat de banen van een actie blokken zijn volgens deze definitie. Het wordt echter interessanter om blokken te zoeken wanneer de actie transitief is. Een transitieve actie van een groep G op een verzameling Ω wordt *primitief* (**E:** *primitive*) genoemd als er geen niet-triviale blokken zijn onder de actie. Een actie die niet primitief is wordt *imprimitief* (**E:** *imprimitve*) genoemd.

Als Δ een blok is onder de actie dan vormen de verschillende verzamelingen Δ^g een partitie van Ω. De verzameling van deze beelden wordt een *bloksysteem* (**E:** *block system*) genoemd. Een transitieve actie is dus primitief als en slechts als ze geen niet-triviale partitie van Ω invariant laat. Het is duidelijk dat $|\Delta| = |\Delta^g|$, dus als $|\Omega|$ priem is, dan is een transitieve actie van G op Ω primitief.

We eindigen deze paragraaf met de definitie van het wreath produkt. Stel dat G een groep is en dat $P \leqslant \mathrm{Sym}(n)$. Het *wreath produkt* (**E:** *wreath product*) van P met G, genoteerd $G \wr P$, is gedefinieerd als het semidirect produkt $P \ltimes_\varphi G^n$, waarbij $\varphi : P \to \mathrm{Aut}(G^n)$ een element uit P op de natuurlijke manier laat werken op de n componenten van het cartesisch produkt G^n. Ga na dat φ volgens deze definitie een homomorfisme is van $P \to \mathrm{G^n}$ en bewijs als oefening dat $G^n \trianglelefteq G \wr P$.

Veronderstel dat Ω de disjuncte unie is van de verzamelingen $\Delta_1, \ldots, \Delta_m$, waarbij elke Δ_i l elementen bevat. Het is nu duidelijk dat de groep $G = \mathrm{Sym}(l) \wr \mathrm{Sym}(m)$ deze partitie van Ω invariant laat. Het is ook duidelijk dat $H = \mathrm{Sym}(l)^m$ elke Δ_i stabilizeert. Tenslotte geldt dat $G/H = \mathrm{Sym}(m)$.

2.1.6 Vrije groepen

We definiëren in deze paragraaf de zogenaamde vrije groepen. Stel dat F een groep is. We noemen F een *vrije groep* (**E:** *free group*) over de verzameling $X \subseteq F$ als voor elke groep G en elke afbeelding $\theta : X \to G$, er een uniek groepshomomorfisme $\theta' : F \to G$, met $\theta'(x) = \theta(x)$ voor alle $x \in X$. Het volgt uit de definitie dat elke groep G, voortgebracht door een verzameling X, het epimorfe beeld is van een vrije groep over X. Daaruit volgt dat G isomorf is met een quotiënt van een vrije groep over X. De kardinaliteit van X wordt de *rang* (**E:** *rank*) van de vrije groep F over X genoemd.

We bewijzen het bestaan van de vrije groep over een verzameling X als volgt. Beschouw een willekeurige verzameling X. Definieer $X^{-1} := \{(x, -1) | x \in X\}$. Noteer een element uit X^{-1} als x^{-1}. Voor $y = x^{-1} \in X^{-1}$ definiëren we $y^{-1} := x$. Definieer $A_X := X \cup X^{-1}$.

Voor een willekeurige verzameling A definiëren we A^* als de verzameling van *woorden* $x_1 x_2 \ldots x_r$, $x_i \in A$. Het getal r is de lengte van het woord. We noteren de lengte van $\sigma \in A^*$ met $|\sigma|$. We noteren het *ledig woord* als ϵ_A (of als ϵ als A duidelijk is). Een *deelwoord* van een woord $w = x_1 x_2 \ldots x_r$ is elk woord van de vorm $x_i x_{i+1} \ldots x_j$, $1 \le i \le j \le r$. Als $i = 1$ (respectievelijk $j = r$), dan wordt het deelwoord een *prefix* (respectievelijk *suffix*) genoemd. Het ledige woord wordt een prefix genoemd.

Beschouw nu een verzameling X en beschouw $A := A_X$. Noem twee woorden $v, w \in A^*$ *direct equivalent* als één woord uit het andere kan bekomen worden door invoeging of verwijdering van een deelwoord xx^{-1}, $x \in A$. Stel bijvoorbeeld $X = \{x, y\}$, dan is $xxx^{-1}y$ direct equivalent met xy. Twee woorden $v, w \in A^*$ zijn *equivalent* als er een rij bestaat $v = v_0, v_1, \ldots, v_r = w$, $r \ge 0$, zodat v_i en v_{i+1} direct equivalent zijn. Noteer de bekomen equivalentierelatie op A^* met $\sim$. De equivalentieklasse van een woord w noteren we met $[w]$. Noteer met F_X de verzameling van equivalentieklassen van $\sim$. Het is duidelijk dat $v_1 \sim w_1$ en $v_2 \sim w_2$ impliceert dat $v_1 v_2 \sim w_1 w_2$. Daarmee kunnen we $[u_1][u_2] := [u_1 u_2]$ definiëren met $u_1 u_2$ de aaneenschakeling van de woorden u_1 en u_2. Aangezien deze aaneenschakeling associatief is, is deze vermenigvuldiging eveneens associatief. Verder is $[\epsilon]$ het eenheidselement en $[x_1 x_2 \ldots x_r]$ vermenigvuldigd met $[x_r^{-1} x_{r-1}^{-1} \ldots x_1^{-1}]$ is gelijk aan $[\epsilon]$. We besluiten dat F_X een groep is. Men bewijst:

Stelling 2.1.10. *Voor een willekeurige verzameling X is F_X een vrije groep over de verzameling $[X] := \{[x] | x \in X\}$, en de afbeelding $x \mapsto [x]$ is een bijectie van X naar $[X]$.*

We vermelden enkele basisresultaten:

Stelling 2.1.11. *(i) Twee vrije groepen over eenzelfde verzameling X zijn isomorf.*

(ii) Vrije groepen over twee verzamelingen X_1 en X_2 zijn isomorf als en slechts als $|X_1| = |X_2|$.

Elke verzameling X geeft dus aanleiding tot een vrije groep F over de verzameling $[X]$. We zullen deze groep nu beschouwen als de vrije groep over X. Dat wil zeggen dat we X met $[X]$ identificeren. We zullen een element ook als w noteren en niet als $[w]$. Om alle dubbelzinnigheden uit te sluiten noteren we $w_1 =_F w_2$ voor de gelijkheid $[w_1] = [w_2]$. Met $w_1 = w_2$ bedoelen we nog steeds de gelijkheid in de verzameling X.

Een woord uit A^* wordt *gereduceerd* genoemd als het xx^{-1} niet als deelwoord bevat voor elke $x \in A$. Men bewijst het volgende resultaat.

Stelling 2.1.12. *Beschouw een willekeurige verzameling X. Dan bevat elke equivalentieklasse in F_X juist één gereduceerd woord.*

Beschouw weer een willekeurig verzameling X en stel $A = A_X$ zoals hierboven gedefinieerd. Stel dat R een deelverzameling is van A^*. We definiëren de *groep voorstelling* (**E:** *groep presentation*) $\langle X|R\rangle := F/N$, met N de normale sluiting van R in F.

Men bewijst het volgende resultaat.

Stelling 2.1.13. *Beschouw een verzameling X, stel $A = A_X$ en beschouw een deelverzameling $R \subset A^*$ en stel dat G een groep is. Beschouw een willekeurige afbeelding $\theta : X \to G$. We breiden θ uit tot een afbeelding $\theta : A \to G$, door $\theta(x^{-1}) := \theta(x)^{-1}$, voor alle $x \in X$. Veronderstel nu dat de afbeelding $\theta : X \to G$ de eigenschap heeft dat, voor alle $w = x_1x_2 \ldots x_r \in R$, $\theta(x_1)\theta(x_2)\ldots\theta(x_r) = \mathbf{1}_G$. Dan bestaat er een uniek groepshomorfisme $\theta' : \langle X|R\rangle \to G$ waarvoor $\theta'(xN) = \theta(x)$ voor alle $x \in X$.*

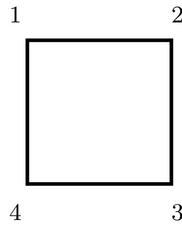
Omgekeerd, als er een groepshomorfimse $\theta' : \langle X|R\rangle \to G$ waarvoor $\theta'(xN) = \theta(x)$ voor alle $x \in X$, dan noodzakelijk $\theta(x_1)\ldots\theta(x_r) = \mathbf{1}$ voor alle $w = x_1 \ldots x_r \in R$.

Beschouw nu de groep $G = \langle X|R\rangle = F/N$. Een element $vN \in G$ noteren we als v. De notatie $v =_G u$ betekent dat de twee elementen $u, v \in A^*$ het zelfde element van G voorstellen.

Stel nu dat een groep G voortgebracht wordt door X. Zoals gebruikelijk noteren we $A = A_X$. Een woord $w \in A^*$ is een *relator* (**E:** *relator*) als en slechts als $w =_G \mathbf{1}$. Het is duidelijk dat de elementen van R in de groep $\langle X|R\rangle$ relators zijn, deze worden *definiërende relators* (**E:** *defining relators*) genoemd. Als $w_1, w_2 \in A^*$, twee elementen zijn waarvoor $w_1 =_G w_2$, dan noemen we $w_1 =_G w_2$ een *relatie* (**E:** *relation*) in G. Zeggen dat $w_1 =_G w_2$ een relatie is in G is equivalent met zeggen dat $w_1w_2^{-1}$ een relator is in G. Tenslotte voeren we een alternatieve notatie in. Stel dat $\mathcal{R} \subseteq A^* \times A^*$. Dan defeniëren we $\langle X|\mathcal{R}\rangle := \langle X|R\rangle$ met $R := \{w_1w_2^{-1} \| (w_1, w_2) \in \mathcal{R}\}$.

2.2 Definiërende relaties van een groep

We beschouwen nu een groep G voortgebracht door de verzameling van generatoren S. Een relatie in G is *triviaal* als ze geëvalueerd kan worden door enkel gebruik te maken van de groepsaxioma's. Dergelijke relaties gelden natuurlijk in elke groep. Het is duidelijk



Figuur 2.1: de hoekpunten van het vierkant gelabelled

dat een relatie van een groep correspondeert met een lus in het Cayleygraaf die begint (en eindigt) in de knoop die correspondeert met het eenheidselement van de groep.

Als voorbeeld gebruiken we de symmetriegroep van het vierkant, genoteerd D_8. Deze groep wordt voortbegracht door $\{a, b\}$ met $a = (1\,2\,3\,4)$ en $b = (2\,4)$. De elementen van D_8 zijn (in een geordende lijst): $[\mathbf{1}, (1\,2\,3\,4), (1\,3)(2\,4), (1\,4\,3\,2), (2\,4), (1\,2)(3\,4), (1\,3), (1\,4)(2\,3),]$. De toppen van het Cayleygraaf zijn de elementen van de groep D_8, we geven ze het nummer dat ze in de bovenstaande lijst hebben. Wanneer we uit element $g_1 \in D_8$ element $g_2 \in D_8$ bekomen door g_1 te vermenigvuldigen met generator s, dan is er een boog van g_1 naar g_2 met als kleur (of label) s. De volgende bogen behoren dus tot het Cayleygraaf van D_8.

$$5 \xrightarrow[a]{} 8$$

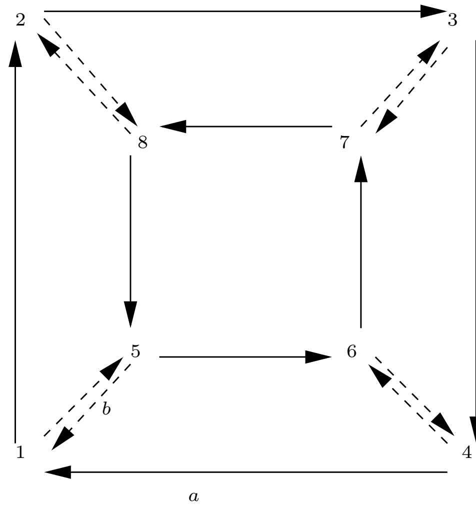
$$5 \xrightarrow[b]{} 1$$

Aan de hand van het Cayleygraaf (Figuur 2.2) van D_8 vinden we onmiddellijk dat

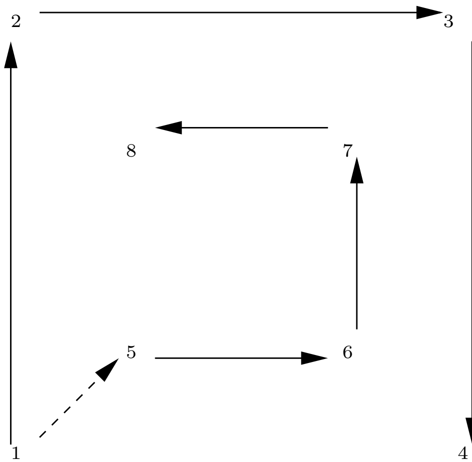
$$\begin{aligned} a &= 2 \\ b &= 5 \\ b^2 &= 1 \\ a \cdot b &= 6 \\ a^6 &= 3 \\ a \cdot b \cdot a \cdot b \cdot a^{-1} &= 4 \end{aligned}$$

Uiteraard zijn er oneindig veel woorden die evalueren naar een gegeven element $g \in G$. Om zo een woord te vinden volstaat het nu in het graaf te vertrekken in knoop 1, en een pad te volgen dat het element g bereikt. Het volgen van een boog in tegengestelde richting (m.a.w. het gebruiken van een inverse van een generator) is toeglaten.

Het is duidelijk dat er meerdere paden vanuit 1 naar een gegeven element leiden. Een *spanning tree* is een deelgraaf van het Cayleygraaf $\Gamma_X(G)$ met de eigenschap dat er voor elke knoop j juist één pad bestaat dat vertrekt in 1 en j bevat.



Figuur 2.2: Cayleygraaf van de groep D_8



Figuur 2.3: spanning tree

Aan de hand van het Cayleygraaf kunnen we nu op zoek gaan naar de relaties. Enkele voorbeelden in D_8 zijn a^4,a^8,a^{12},b^2,$a^4 \cdot b^2$, b^4 en $b \cdot a \cdot b \cdot a$. Deze relaties zijn niet onafhankelijk van elkaar. Een verzameling relaties is een verzameling *definiërende relates* voor de groep als elke niet-triviale relatie hieruit kan afgeleid worden.

2.3 De bepaling van de relaties

Beschouw de groep $G = D_8$ en veronderstel dat $\Gamma_X(G)$ het Cayleygraaf van G is met bijhorende spanning tree T. Stel dat e een boog van Γ is. We kunnen e identificeren met het geordend koppel $[i, j]$. Uit de boog e kunnen we eenvoudig een relatie $R(e)$ afleiden, nl. $R(e) = w_i \cdot s \cdot w_j^{-1}$ met w_i (respectievelijk w_j) het woord uit de spanning tree dat evalueert naar i (respectievelijk j), en s het label van de boog e.

Stelling 2.3.1. *Stel dat G een groep is met bijhorend Cayleygraaf $\Gamma_X(G)$ en bijhorende spanning tree T. De verzameling definiërende relaties voor G wordt gegeven door* $R(T) = \{R(e)\|e \text{ is een boog van } \Gamma \text{ maar niet van } T\}$.

We zullen deze stelling aantonen voor D_8. We bekomen de volgende lijst van relaties, gebruikmakend van de spanning tree uit Figuur 2.2[2].

$R([2, 8]_b) = a \cdot b \cdot a^{-3} \cdot b^{-1}$
$R([3, 7]_b) = a^2 \cdot b \cdot a^{-2} \cdot b^{-1}$
$R([4, 1]_a) = a^3 \cdot a = a^4$
$R([4, 6]_b) = a^3 \cdot b \cdot a^{-1} \cdot b^{-1}$
$R([5, 1]_b) = b \cdot b = b^2$
$R([6, 4]_b) = b \cdot a \cdot b \cdot a^{-3}$
$R([8, 5]_a) = b \cdot a^3 \cdot a \cdot b^{-1}$
$R([7, 3]_b) = b \cdot a^2 \cdot b \cdot a^{-2}$
$R([8, 2]_b) = b \cdot a^3 \cdot b \cdot a^{-1}$

Merk op dat een relatie $R(e)$ ook gedefinieerd is als e een boog uit de spanning tree is. Maar in dat geval ligt de lus volledig in de spanning tree en bekomen we een triviale relatie.

Om de stelling te bewijzen moeten we aantonen dat elke relatie afleidbaar is uit de verzamling $R(T)$. Beschouw een willekeurige lus

$$\mathbf{1} = [j_0, j_1]_{s_{i_1}}, [j_1, j_2]_{s_{i_2}}, \ldots, [j_{m-1}, j_m]_{s_{i_m}} = \mathbf{1}$$

[2]We noteren een boog in het Cayleygraaf nog steeds als volgt: $[i, j]_s$ duidt de boog van i naar j aan, waarbij j uit i bekomen wordt door te vermenigvuldigen met s

rond het eenheidselement, die correspondeert met de relatie

$$s_{i_1} \cdot s_{i_2} \cdot \ldots \cdot s_{i_m} = \mathbf{1}$$

In elke knoop die in deze lus ligt kunnen we een pad uit de spanningtree naar het eenheidselement en terug, invoegen. Elk invoegen van zo een pad is het invoegen van een triviale relatie. De nieuwe lus is de aaneeschakeling van van de lussen $R([j_{k-1}, j_k]_{s_{i_k}})$. Ofwel behoren deze tot $R(T)$ ofwel zijn het triviale relaties. Bijgevolg zijn alle relaties afleidbaar uit $R(T)$. De grootte van $R(T)$ is het aantal bogen die niet tot de spanning tree behoren. We kunnen eenvoudig berekenen dat $|R(T)| = 1 + |G|\,(|S| - 1)$. De natuurlijke vraag stelt zich of er een veel kleinere verzameling van relaties bestaat waaruit we alle relaties uit $R(T)$ kunnen afleiden.

2.4 Het "Colouring" algoritme

Het colouring-algoritme bepaalt een deelverzameling van $R(T)$ die eveneens een verzameling definiërende relaties is voor G. Het basisprincipe is het kleuren van bogen e, waarvoor de relatie $R(e)$ reeds kan afgeleid worden uit de voorhanden zijnde verzameling relaties. Initieel zijn de bogen van de spanningtree gekleurd, omdat ze met triviale relaties corresponderen. Telkens als er een nieuwe relatie toegevoegd wordt aan de (initieel lege) verzameling $R(T)$, worden alle afgeleide relaties van de deelverzameling bepaald en worden de corresponderende bogen gekleurd. Nadien wordt er een niet gekleurde boog gekozen en wordt de corresponderende relatie toegevoegd, en worden opnieuw bogen corresponderend met afleidbare relaties gekleurd. Het proces stopt als alle bogen gekleurd zijn.

De kern van het algoritme is de bepaling van de afleidingen. Stel dat r een relatie is. Dan kunnen we alle relaties van de vorm $w_i \cdot r \cdot w_i^{-1}$ hieruit bepalen, voor alle knopen i uit het graaf. Dit is een lus (met pad r) rond knoop i. Als deze lus juist één ongekleurde boog e bevat, dan kleuren we deze. De lus kan beschouwd worden als de samenstelling van gekleurde bogen f_i, samen met de ongekleurde boog e. Alle gekleurde bogen corresponderen met afleidbare relaties, dus $R(e)$ kan afgeleid worden uit de relaties corresponderend met de bogen f_i, samen met $w_i \cdot r \cdot w_i^{-1}$. Bijgevolg is $R(e)$ afleidbaar, en mag e gekleurd worden.

Om alle afleidingen van de verzameling $R(T)$ te bepalen, zullen we herhaaldelijk een relatie traceren rond alle knopen van het graaf, bogen kleuren waar mogelijk, totdat geen bogen meer gekleurd kunnen worden. Dit doen we voor alle relaties.

Algoritme 2.1 definiërende relaties door het Colouring-algoritme

```
COLOUR1(S)
     Invoer: een verzameling generatoren X voor G, caleygraaf Γ_X(G),
     Uitvoer: een verzameling definiërende relaties.
 1   construeer een spanning tree T voor Γ_X(G);
 2   kleur de bogen van T;
 3   R := ∅;
 4   while er zijn ongekleurde bogen
 5       do kies een ongekleurde boog e;
 6          voeg R(e) toe aan R;
 7          kleur e
 8          repeat
 9                for alle relaties r ∈ R
10                    do for alle knopen i
11                           do traceer r rond i;
12                              if lus bevat juist 1 ongekleurde boog f
13                                 then kleur f;
14            until er kunnen geen bogen meer gekleurd worden
15   return R
```

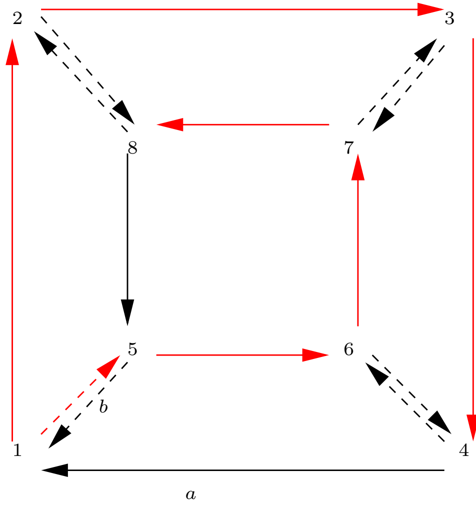
We illustreren dit algoritme aan de hand van de groep D_8. Initieel zijn de bogen van de spanning tree gekleurd.

We kiezen de niet gekleurde boog $[4, 1]_a$. Dit levert de eerste definiërende relatie a^4 op. We traceren nu a^4 rond knoop 5, hetgeen de boog $[8, 5]_a$ kleurt.

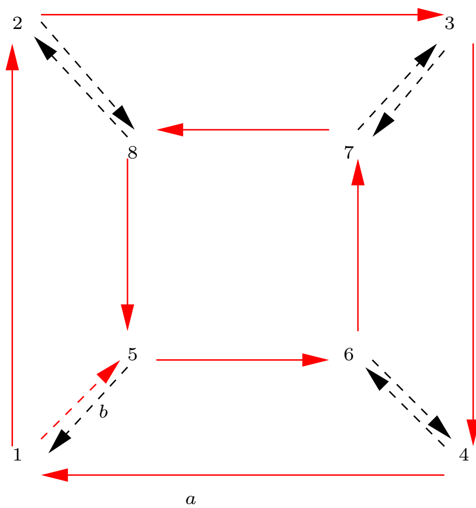
We kiezen nu de boog $[5, 1]_b$. Dit levert de definiërende relatie b^2 op. Traceren van de relaties rond alle knopen levert geen extra gekleurde bogen meer op.

We kiezen de boog $[2, 8]_b$. Dat levert de relatie $a \cdot b \cdot a^{-3} \cdot b^{-1}$ op. Deze wordt toegevoegd aan de verzameling definiërende relaties. b^2 traceren rond knoop 2 kleurt $[8, 2]_b$, $a \cdot b \cdot a^{-3} \cdot b^{-1}$ traceren rond 2 kleurt $[3, 7]_b$, b^2 traceren rond knoop 3 kleurt $[7, 3]_b$, $a \cdot b \cdot a^{-3} \cdot b^{-1}$ traceren rond knoop 3 kleurt $[4, 6]_b$ en tenslotte b^2 traceren rond knoop 4 kleurt $[6, 4]_b$. Alle bogen zijn gekleurd en de verzameling definiërende relaties is $\{a^4, b^2, a \cdot b \cdot a^{-3} \cdot b^{-1}\}$.

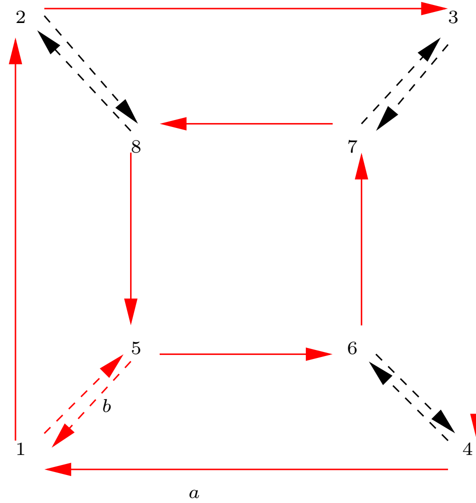
Indien we de spanning tree uit Figuur 2.7 hadden gebruikt, dan hadden we als verzameling definiërende relates $\{a^4, b^2, a \cdot b \cdot a \cdot b^{-1}\}$ gevonden.



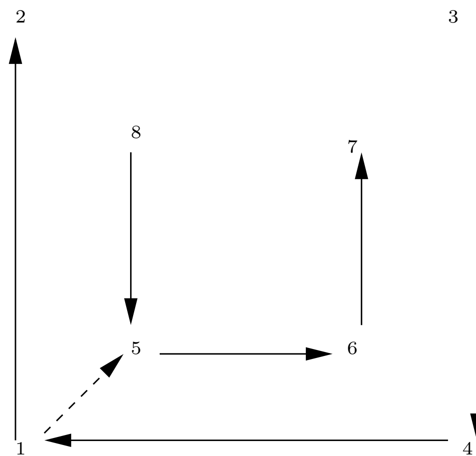
Figuur 2.4: initieel gekleurde bogen



Figuur 2.5: gekleurde bogen na tracering van eerste relatie rond 5



Figuur 2.6: gekleurde bogen na traceringen van twee definiërende relaties



Figuur 2.7: een minimale spanning tree

2.5 Implementatie en voorbeelden

Een aantal praktische aspecten van Algoritme 2.1 komen slechts aan het licht tijdens de implementatie. Zo leiden verschillende keuzes van spanningtree tot verschillende relaties. Daarenboven wordt uit de verzameling niet gekleurde bogen een willekeurige boog gekozen met een random functie. Dit verklaart waarom niet altijd dezelfde definiërende relaties teruggegeven worden. De implementatie is opgenomen in de file `relaties`. In de volgende interactieve magma sessie illustreren we de resultaten. We definiëren telkens een groep, en voeren de implementatie uit met het commando `load relaties`.

```
jdebeule@euclides:~ > magma2.8
Magma V2.8-10     Wed Feb 23 2005 14:22:26 on euclides [Seed = 1543675269]
Type ? for help.  Type <Ctrl>-D to quit.
> G := PermutationGroup<4|(1,2,3,4),(2,4)>;
> load relaties;
Loading "relaties"
Permutation group G acting on a set of cardinality 4
Order = 8 = 2^3
    (1, 2, 3, 4)
    (2, 4)
a =  (1, 2, 3, 4)
b =  (2, 4)
Relaties:
a * b * a * b^-1 = Id(G)
a * b^2 * a^-1 = Id(G)
b * a * b * a^-3 = Id(G)
> load relaties;
Loading "relaties"
Permutation group G acting on a set of cardinality 4
Order = 8 = 2^3
    (1, 2, 3, 4)
    (2, 4)
a =  (1, 2, 3, 4)
b =  (2, 4)
Relaties:
b^2 = Id(G)
a^4 = Id(G)
a * b * a * b^-1 = Id(G)
```

We vinden telkens drie definiërende relaties. We willen ook controleren dat de definiërende relaties wel degelijk de groep bepalen. Daartoe construeren we in MAGMA een vrije groep, en eisen nadien dat de generatoren aan de relaties voldoen. Dit moet ons een groep opleveren isomorf met de groep waar we van startten.

```
> G := PermutationGroup<6|(1,2,3,4,5,6),(1,5)(2,4)>;
> load relaties;
Loading "relaties"
Permutation group G acting on a set of cardinality 6
Order = 12 = 2^2 * 3
    (1, 2, 3, 4, 5, 6)
    (1, 5)(2, 4)
a =  (1, 5)(2, 4)
b =  (1, 2, 3, 4, 5, 6)
Relaties:
a * b^3 * a * b^-3 = Id(G)
a * b^4 * a^-1 * b^-2 = Id(G)
b^3 * a * b^-3 * a^-1 = Id(G)
> G := Sym(3);
> load relaties;
Loading "relaties"
Symmetric group G acting on a set of cardinality 3
Order = 6 = 2 * 3
    (1, 2, 3)
    (1, 2)
a =  (1, 2, 3)
b =  (1, 2)
Relaties:
b * a^2 * b^-1 * a^-1 = Id(G)
b^2 = Id(G)
```

```
> F<x,y> := FreeGroup(2);
> H<x,y>,phi := quo<F|y * x^2 * y^-1 * x^-1 = Id(F),y^2 = Id(F)>;
> print H;
Finitely presented group H on 2 generators
Relations
    y * x^2 * y^-1 * x^-1 = Id(H)
    y^2 = Id(H)
> Order(H);
6
```

2.6 Oefeningen

1. Definiërende relaties van een groep: de implementatie is enkel beschikbaar voor MAGMA, file: relaties. Start MAGMA op, definieer en ken hem toe aan de variabele G. Voer de implementatie uit met het commando 'load relaties;' Een aantal kleine groepen:

   - D_8: werkt in op 4 elementen, heeft generatoren $(1, 2, 3, 4)$ en $(2, 4)$.
   - D_{12}: werkt in op 6 elementen, heeft generatoren $(1, 2, 3, 4, 5, 6)$ en $(1, 5)(2, 4)$.
   - D_{24}: werkt in op 12 elementen, heeft generatoren $(3, 6, 5, 4)$, $(1, 2)(3, 6, 5)$ en $(1, 2)$.
   - S_3: heeft generatoren $(1, 2, 3)$, $(1, 2)$. Ga ook na dat de groepen S_n onmiddellijk kunnen opgeroepen worden.
   - S_4: heeft generatoren $(1, 2, 3, 4)$ en $(1, 2)$
   - S_5, S_6 en S_7
   - A_5: heeft generatoren $(1, 2, 3)$ en $(3, 4, 5)$

   Definieer de (kleine) groepen opnieuw aan de hand van de bekomen relaties.

Hoofdstuk 3

Tralie van deelgroepen

Een volledige beschrijving van een groep G bevat bijvoorbeeld alle deelgroepen en hun onderling verband. De verzameling van alle deelgroepen wordt met de relatie "is een deelgroep van" een partieel geordende verzameling of *tralie*. We zullen een algoritme bespreken dat deze tralie van deelgroepen van oplosbare groepen bepaalt, de *cyclic extension method*. Dit algoritme is toepasbaar op willekeurige groepen, maar bepaalt in dat geval niet alle deelgroepen: de perfecte deelgroepen kunnen niet geconstrueerd worden.

3.1 Definities

Beschouw een willekeurige groep G. We definiëren de *commutator* (**E:** *commutator*) van twee elementen $g, h \in G$ als $[g, h] := g^{-1}h^{-1}gh$. We noteren de deelgroep van G, voortgebracht door **alle** commutators $[g, h]$ (voor alle elementen $g, h \in G$), met $[G, G]$. We noemen de deelgroep $[G, G]$ de *afgeleide deelgroep* (**E:** *derived subgroup*) (of *commutatordeelgroep* (**E:** *commutator subgroup*)) van G. We noteren $[G, G]$ ook met G' of met δG. De afgeleide deelgroep van G' noemen we de *tweede normale afgeleide* (**E:** *second normal derived subgroup*), en noteren we met G'', $G^{(2)}$ of $\delta^{(2)}G$. Per definitie geldt $G'' = [G', G']$, of, $G^{(n)} = [G^{(n-1)}, G^{(n-1)}]$. Een groep G waarvoor $G^{(n)} = \{1\}$ voor een $n \in \mathbb{N}$ noemen we *oplosbaar* (**E:** *solvable or soluble*).

De deelgroep van G, voorgebracht door alle commutators $[g, h]$, met $g \in G$ en $\mathbf{h} \in \mathbf{G}'$, noteren we met $[G, G']$ en noemen we de *tweede centrale afgeleide* (**E:** *second central derived subgroup*) van G. We noteren ook $G^{[2]} = [G, G']$ en $G^{[n]} = [G, G^{[n-1]}]$. Een groep G waarvoor $G^{[n]} = \{1\}$ voor een $n \in \mathbb{N}$ noemen we een *nilpotente* (**E:** *nilpotent*) groep.

Stelling 3.1.1. *Een groep G is oplosbaar als er een ketting van deelgroepen van G bestaat:*

$$\mathbf{1} = U_0 \leqslant U_1 \leqslant \ldots \leqslant U_i = G$$

met de eigenschap dat $U_i \trianglelefteq U_j$ en dat $[U_j : U_{j-1}] = p$. Als G een oplosbare groep is dan is elke deelgroep $H \leqslant G$ een oplosbare groep. Omdat de index $[U_j : U_{j-1}]$ een priemgetal is wordt U_j voorgebracht door U_{j-1} en een extra element.

Een groep G is *perfect* (**E:** *perfect*) als en slechts als $G = [G, G]$. Aangezien $[G, G] \trianglelefteq G$, is een enkelvoudige groep altijd perfect. Het omgekeerde is echter niet zo, beschouw bijvoorbeeld de groep $SL_2(9)$. Tenslotte vermelden we dat elke perfecte deelgroep van een groep G bevat is in de unieke maximale perfecte deelgroep van G.

3.2 De tralie ingedeeld in lagen

We beschrijven de zogenaamde "cylic extension methode", die op inductieve wijze de tralie van deelgroepen van een oplosbare groep G constueert. We zullen eerst de tralie van deelgroepen indelen in lagen. Een deelgroep $U \leqslant G$ behoort tot laag i als en slechts als er een deelgroep U_{i-1} bestaat die tot laag $i - 1$ behoort en waarvoor $U_{i-1} \trianglelefteq U_i$ en $[U_i : U_{i-1}] = p$, een priemgetal. Omdat U oplosbaar is (als deelgroep van een oplosbare groep), behoort U tot juist 1 laag, immers, de oplosbaarheid garandeert het bestaan van een ketting van normaaldelers van U en de orde van U kan op juist 1 manier als het product van priemgetallen geschreven worden. (waarbij deze priemgetallen niet noodzakelijk moeten verschillen).

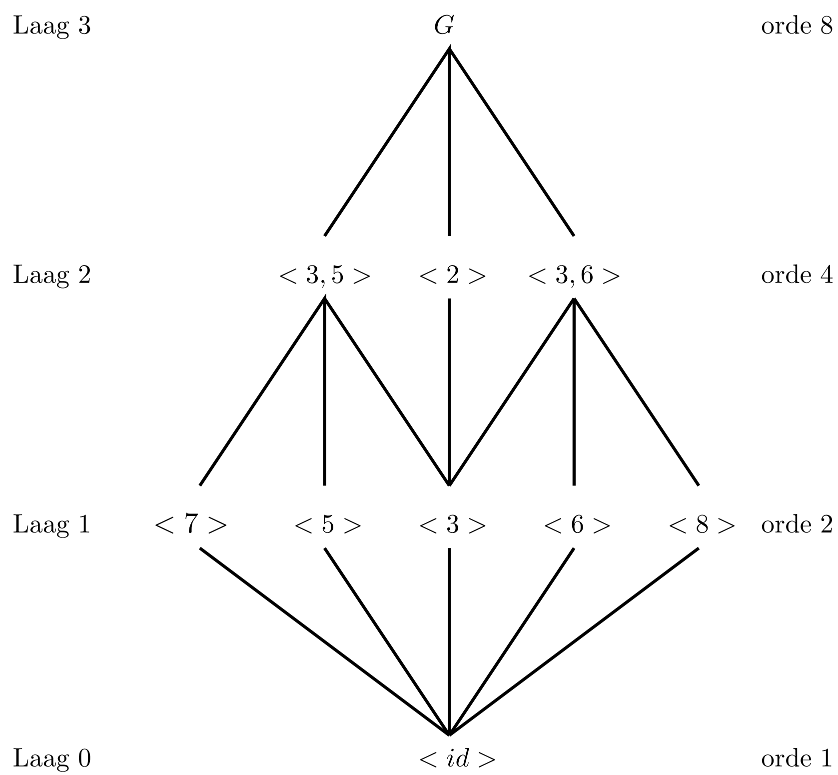
Merk op dat in een tralie elk koppel elementen een kleinste bovengrens heeft en een grootste ondergrens. In een tralie van deelgroepen is is de kleinste bovengrens van het koppel H, K de deelgroep $< H, K >$, en de grootste ondergrens de deelgroep $H \cap K$.

We beschouwen als voorbeeld (zie Fig. 3.1) de groep $D_8 = \{id, (1\,4\,3\,2), (1\,3)(2\,4), (1\,2\,3\,4), (2\,4), (1\,2)(3\,4), (1\,3), (1\,4)(2\,3)\}$ (zie Fig. 2.1 voor de labelling van de hoekpunten van het vierkant). Met een notatie als $< 2 >$ bedoelen we de groep voortgebracht door het tweede element in de bovenstaande lijst, met $< 3, 6 >$ bedoelen we de groep voortgebracht door het derde en zesde element in de lijst. De orde van elke groep is rechts genoteerd. Merk op dat al deze ordes, bovenaan beginnend, telkens het product zijn van $3, 2, 1$ en 0 priemgetallen, corresponderend met laag $3, 2, 1$ en 0 respectievelijk. Deze priemgetallen zijn in dit geval allemaal hetzelfde. De orde van een deelgroep in Laag i is het product van i priemgetallen. We definiëren dan ook de lagen van de tralie aan de hand van de orde van een deelgroep: Laag i bevat alle deelgroepen waarvoor de orde het product van i priemgetallen is.

Hiermee kunnen we de inductieve stap in het algoritme bespreken. Stel dat we over laag $i - 1$ van de tralie beschikken. We willen laag i construeren. De karakterisering van een deelgroep U in laag i kunnen we als volgt samenvatten: $U = U_i$ behoort tot laag i als en slechts als er een groep U_{i-1} uit laag $i - 1$ en een element $g \in G$ bestaat zodat:

1 $U_i =< U_{i-1}, g >$

2 $U_{i-1} \trianglelefteq U_i$



Figuur 3.1: tralie van deelgroepen van D_8

3 de index $[U_i : U_{i-1}] = p_i$, een priemgetal

We gaan dus op zoek naar een element $g \in G \setminus U_{i-1}$, dat U_i normaliseert en waarvoor $g^{p_i} \in U_{i_1}$

Als we laag $i-1$ volledig geconstrueerd hebben, dan beschouwen we elke deelgroep U_{i-1} uit die laag, we zoeken elementen $g \in U_{i-1}$ die aan bovenstaande voorwaarden voldoen, en we voegen de groep $< U_{i-1}, g >$ toe aan laag i, op voorwaarde dat die groep nog niet gekend was. Dit geeft aanleiding tot het volgende algoritme

---

Algoritme 3.1 tralie van deelgroepen: inductieve stap (1)

LatticeInductive(G,*list*,$laag_{i-1}$)

Invoer: oplosbare groep G, *list*: lijst met alle elementen van G
laag $i-1$ uit de tralie van deelgroepen.
Uitvoer: laag i uit de tralie van deelgroepen.

1 **for** alle deelgroepen U_{i-1} uit laag $i-1$
2 **do for** alle elementen $g \in G$
3 **do if** $g \notin U_{i-1}$ and $U_{i-1}^g = U_{i-1}$
4 **then if** $< U_{i-1}, g >$ is een nieuwe deelgroep
5 **then** voeg $< U_{i-1}, g >$ toe aan laag i;

3.3 Testen wanneer een groep nieuw is

Om te controleren dat $W :=< U_{i-1}, g >$ reeds tot laag i behoort, is het nodig en voldoende te controleren dat $U_{i-1} \subset W$ en dat $g \in W$. Deze controle kan gebeuren zonder dat alle elementen van $< U_{i-1}, g >$ gegenereert worden.

Als $< U_{i-1}, g >$ nieuw is, dan weten we dat $U = U_{i-1} \cup U_{i-1} \cdot g \cup U_{i-1} \cdot g^2 \cup \ldots U_{i-1} \cdot g^{p_i - 1}$.

Het aantal testen om te zien of deelgroepen nieuw zijn, kan verminderd worden door nooit elementen g te testen die geen nieuwe deelgroepen opleveren. Hiervoor kunnen we de verzameling Γ definiëren:

(1) $\Gamma = G \setminus U_{i-1}$

(2) $\Gamma := \Gamma \setminus W$ voor alle deelgroepen W die reeds op laag i gekend zijn en waarvoor $U_{i-1} \subset W$

Dit levert ons het volgende algoritme:

Algoritme 3.2 tralie van deelgroepen: inductieve stap (2)

LATTICEINDUCTIVE2(G,$list$,$laag_{i-1}$)

Invoer: oplosbare groep G, $list$: lijst met alle elementen van G
laag $i-1$ uit de tralie van deelgroepen.
Uitvoer: laag i uit de tralie van deelgroepen.

```
1   for alle deelgroepen U_{i-1} uit laag i − 1
2   Γ := G \ U_{i-1}
3   for alle deelgroepen W in laag i
4         do if U_{i-1} ⊂ W
5               then Γ := Γ \ W
6           while Γ ≠ ∅
7                 do kies g ∈ Γ;
8                    if U^g_{i-1} = U_i and p^p ∈ U_{i-1} voor een priemgetal p
9                       then voeg U := ⟨U_{i-1}, g⟩ toe aan laag i;
10                           Γ := Γ \ U;
11                      else  Γ := Γ \ {g};
```

Merk op dat voorwaarden uit de binnenste lus van Algoritme 3.1 hier weggevallen zijn door de nieuwe aanpak.

Algoritme 3.1 kan opnieuw verbeterd worden als we de controles $U_{i-1}^g = U_{i-1}$ kunnen vermijden. Dit kunnen we bereiken door $N_G(U_{i-1}$ te bepalen. De algoritmen uit Hoofdstuk 1 (Paragraaf 1.4). Dit is in elk geval efficiënter dan de controle $U_{i-1}^g = U_{i-1}$ voor alle elementen $g \in \Gamma$. We bekomen Algoritme 3.3.

Algoritme 3.3 tralie van deelgroepen: inductieve stap (3)

LatticeInductiveN(G,$list$,$laag_{i-1}$)

Invoer: oplosbare groep G, $list$: lijst met alle elementen van G
laag $i-1$ uit de tralie van deelgroepen.
Uitvoer: laag i uit de tralie van deelgroepen.

1 **for** alle deelgroepen U_{i-1} uit laag $i-1$
(initialiseer Γ om gekende deelgroepen te vermijden)
(en om enkel elementen uit $N_G(U_{i-1})$ te gebruiken
2 Bepaal $N_G(U_{i-1}$;
3 $\Gamma := N_G(U_{i-1}) \setminus U_{i-1}$;
4 **for** alle deelgroepen W in laag i
5 **do if** $U_{i-1} \subset W$
6 **then** $\Gamma := \Gamma \setminus W$;
(construeer nieuwe deelgroep)
7 **while** $\Gamma \neq \emptyset$
8 **do** kies $g \in \Gamma$;
9 **if** $g^p \in U_{i-1}$ voor een priemgetal p
10 **then** voeg $U =< U_{i-1}, g >$ toe aan laag i;
11 $\Gamma := \Gamma \setminus U$;
12 **else** $\Gamma := \Gamma \setminus \{g\}$;

3.4 Gebruik van p-groepen

Een verdere verbetering aan Algoritme 3.3 is mogelijk door gebruik te maken van de volgende stelling

Stelling 3.4.1. *Stel dat $U =< U_{i-1}, g >$ bevat is in laag i van de tralie van deelgroepen van G, met U_{i-1} een deelgroep bevat in laag $i-1$; en stel dat*

- *1. $U_{i-1} \trianglelefteq U$*
- *2. $g^p \in U_{i-1}$, voor een zeker priemgetal p*

Stel dat de orde van g gelijk is aan $p^n \cdot r$, met p en r relatief priem. Dan heeft g^r orde p^n, $(g^r)^p \in U_{i-1}$ en g^r breidt U_{i-1} uit tot U.

Bewijs. Omdat p en r relatief priem zijn, bestaat er een $a, b \in \mathbb{Z}$ zodat $ap + br = 1$. De orde van g^r is duidelijk p^n. Daar $g^p \in U_{i-1}$ en daar $(g^r)^p = (g^p)^r$, behoort de p-de

macht van g^r tot U_{i-1}. De deelgroep $< U_{i-1}, g^r >$ is gelijk aan U daar $g \in < U_{i-1}, g^r >$ wegens $g = g^{ap+br} = (g^p)^a \cdot (g^r)^b$, wat duidelijk behoort tot $< U_{i-1}, g^r >$. ■

Deze stelling heeft als effect dat enkel elementen met een priemmacht orde beschouwd moeten worden wanneer we de deelgroepen uit laag $i-1$ uitbreiden. Er is echter nog heel wat redundantie aanwezig bij deze elementen. Veronderstel dat g orde p^n heeft. De cyclische groep $< g >$ bevat de elementen van $< g^p >$. Elementen met orde p^n zijn elementen van de vorm g^r, met p en r relatief priem. Omdat voor zo een r $< g > = < g^r >$, geldt $< U_{i-1}, g > = < U_{i-1}, g^r >$ en dus g zal U_{i-1} uitbreiden tot een deelgroep U in laag i als en slechts als g^r U_{i-1} uitbreidt tot dezelfde deelgroep. Daardoor volstaat het om enkel een generator te beschouwen van elke cyclische groep van priemmacht orde.

Definieer Z als de verzameling die alle cyclische deelgroepen van priemmacht orde van G bevat. Voor D_8 wordt Z gegeven door

$$Z = \{< 2 >, < 3 >, < 5 >, < 6 >, < 7 >, < 8 >\}$$

Definieer Z_g als de verzameling van generatoren van alle cyclische deelgroepen in Z, voor elke cyclische groep moeten we dus 1 generator kiezen, met de nummering van de elementen van D_8 wordt dit dus:

$$Z_g = \{2, 3, 4, 6, 7, 8\}$$

Deze verzameling zal veel kleiner zijn dan de groep G zelf (in dit geval is er weinig verschil omdat D_8 een zeere kleine groep is).

Algoritme 3.4 tralie van deelgroepen: inductieve stap (4)

LATTICEINDUCTIVEC(G,$list$,$laag_{i-1}$)

Invoer: oplosbare groep G, $list$: lijst met alle elementen van G
laag $i-1$ uit de tralie van deelgroepen.
de verzameling Z_g.
Uitvoer: laag i uit de tralie van deelgroepen.

1 **for** alle deelgroepen U_{i-1} uit laag $i-1$
(initialiseer Γ om gekende deelgroepen te vermijden)
(en om enkel elementen uit $N_G(U_{i-1})$ te gebruiken
(en om enkel de representanten van priemmacht orde te gebruiken)
2 Bepaal $N_G(U_{i-1};$
3 $\Gamma := Z_g \cap (N_G(U_{i-1}) \setminus U_{i-1});$
4 **for** alle deelgroepen W in laag i
5   **do if** $U_{i-1} \subset W$
6   **then** $\Gamma := \Gamma \setminus W;$
(construeer nieuwe deelgroepen)
7 **while** $\Gamma \neq \emptyset$
8   **do** kies $g \in \Gamma;$
9   **if** $g^p \in U_{i-1}$ voor een priemgetal p
10   **then** voeg $U =< U_{i-1}, g >$ toe aan laag i;
11             $\Gamma := \Gamma \setminus U;$
12   **else** $\Gamma := \Gamma \setminus \{g\};$

Het volledige algoritme zal starten met de bepaling van alle cyclische deelgroepen van priemorde van de groep G, en met het kiezen van een verzameling generatoren. Daartoe kunnen we alle elementen van G overlopen, en alle elementen van priemorde beschouwen. De eerste laag zal bestaan uit de deelgroepen van orde p. De cyclische deelgroepen van orde p^n kan men in principe als eerste toevoegen in laag n.

Tenslotte vermelden we nog hoe we dit algoritme kunnen aanpassen voor willekeurige groepen. De bovenstaande methode kan deelgroepen U construeren waarvoor een keten van de vorm

$$id \neq U_0 \leqslant U_1 \leqslant \ldots \leqslant U_i = U$$

bestaat. (U_0 is dus een perfecte deelgroep). Indien we echter in het begin de perfecte deelgroepen van G toevoegen aan de tralie, dan kan de cyclic extension method deelgroepen zoals U wel toevoegen. Dus voor willekeurige groepen is het probleem opgelost als we alle perfecte deelgroepen kunnen bepalen. We hebben reeds vermeld dat alle perfecte deelgroepen bevat zijn in de unieke maximale perfecte deelgroep van G. Deze

unieke maximale perfecte deelgroep van G is de kleinste deelgroep in de ketting van de normale afgeleiden van G.

Er zijn databanken beschikbaar van alle perfecte groepen met hoogstens 10000 elementen, waarbij er ook informatie beschikbaar is over een aantal eigenschappen van deze groepen. Daarvan kan men gebruik maken om de perfecte deelgroepen van een groep toe te voegen aan de tralie.

3.5 Voorbeelden

De file `magma3.5.1.i` toont een korte Magma-sessie waarin de tralie van deelgroepen van D_8 bepaald wordt.

```
Magma V2.8-10     Fri Feb 10 2006 13:26:47 on colossus [Seed = 111789000]
Type ? for help.  Type <Ctrl>-D to quit.
> G:=PermutationGroup<4|(1,2,3,4),(2,4)>;
> print G;
Permutation group G acting on a set of cardinality 4
    (1, 2, 3, 4)
    (2, 4)
> IsSoluble(G);
true
> s := SubgroupLattice(G);
> s;

Partially ordered set of subgroup classes
-----------------------------------------

[1]  Order 1  Length 1  Maximal Subgroups:
---
[2]  Order 2  Length 1  Maximal Subgroups: 1
[3]  Order 2  Length 2  Maximal Subgroups: 1
[4]  Order 2  Length 2  Maximal Subgroups: 1
---
[5]  Order 4  Length 1  Maximal Subgroups: 2
[6]  Order 4  Length 1  Maximal Subgroups: 2 3
[7]  Order 4  Length 1  Maximal Subgroups: 2 4
---
[8]  Order 8  Length 1  Maximal Subgroups: 5 6 7

> H := PermutationGroup<4|(1,2,3,4)>;
> s!H;
5
```

De file `gap3.5.1` toont de volledige implementatie[1] gebaseerd op Paragraaf 3.4. Voor deze file ingelezen wordt in een GAP-sessie, moet een groep G gedefinieerd zijn.

```
if (IsSolvableGroup(G)) then

   P:=Set(Factors(Order(G)));
   elementen := Elements(G);
   laag_0_verz:=[[One(G)]];
   laag_0_groep:=[Subgroup(G,[One(G)])];

   Print(" \n" );
   Print(" De resultaten voor laag 0 \n");
   Print("***************************\n");
   Print(" \n");
   Print(" Aantal deelgroepen in de laag : ",Size(laag_0_verz),"\n");
   Print(" Deze deelgroepen zijn : \n");
   Print( laag_0_verz[1],"\n");
   Print("orde = 1\n");
```

[1]met dank aan Anja Hallez en Tom Mélange

```
   vorige_laag_verz:=laag_0_verz;
   vorige_laag_groep:=laag_0_groep;
   laag:=0;

   repeat
      huidige_laag_verz:=[];
      huidige_laag_groep:=[];
      i:=1;
         for k in [1..Size(vorige_laag_groep)] do
            U_verz:=vorige_laag_verz[k];
    U_groep:=vorige_laag_groep[k];
    Gamma:=Difference(G,U_verz);


         if not (huidige_laag_verz = []) then

            for s in [1..Size(huidige_laag_verz)] do
                W:= huidige_laag_verz[s];
                if (Intersection (U_verz,W)) = U_verz then
                   Gamma:=Difference(Gamma,W);

                fi;
            od;
         fi;

r:=Size(Gamma);

if not (r = 0) then
         repeat
            g:=Gamma[r];
    test:=0;
    k:=1;
    while  ((test=0) and (k<=Size(P))) do
if (g^P[k] in U_verz) then
   test:=1;
fi;
k:=k+1;
    od;

if ((U_groep^g)=U_groep and test=1) then
   huidige_laag_groep[i]:=Subgroup(G,Elements(Union(GeneratorsOfGroup(U_groep),[g])));
           huidige_laag_verz[i]:=Elements(huidige_laag_groep[i]);
           Gamma:=Difference(Gamma,huidige_laag_verz[i]);
           i:=i+1;
  else

            Gamma:=Difference(Gamma,[g]);
        fi;
        r:=Size(Gamma);

        until r = 0;
      fi;
   od;

   vorige_laag_verz:=huidige_laag_verz;
   vorige_laag_groep:=huidige_laag_groep;
   laag:=laag+1;

   Print(" \n" );
   Print(" De resultaten voor laag ", laag ,"\n");
   Print("**************************\n");
   Print(" \n");
   Print(" Aantal deelgroepen in de laag : ",Size(huidige_laag_groep),"\n");
   Print(" Deze deelgroepen zijn : \n");
   for s in [1..Size(huidige_laag_verz)] do
       Print( huidige_laag_verz[s],"\n");
       Print("orde = ",Order(huidige_laag_groep[s]),"\n");
   od;

   until huidige_laag_groep[1] = G ;

else
   Print(" Deze groep is niet oplosbaar!\n");
fi;
```

Tenslotte de interactieve sessie met enkele voorbeelden

```
   Loading the library. Please be patient, this may take a while.
GAP4, Version: 4.4.6 of 02-Sep-2005, powerpc-apple-darwin8.3.0-gcc
Components:  small 2.1, small2 2.0, small3 2.0, small4 1.0, small5 1.0,
             small6 1.0, small7 1.0, small8 1.0, small9 1.0, small10 0.2,
             id2 3.0, id3 2.1, id4 1.0, id5 1.0, id6 1.0, id9 1.0, id10 0.1,
             trans 1.0, prim 2.1  loaded.
Packages:    TomLib 1.1.2  loaded.
gap> G:=Group([(1,2,3,4),(2,4)]);
Group([ (1,2,3,4), (2,4) ])
gap> Read("./Courses/compalg2006/tex/gap/gap3.5.1");

 De resultaten voor laag 0
***************************

 Aantal deelgroepen in de laag : 1
 Deze deelgroepen zijn :
[ () ]
orde = 1

 De resultaten voor laag 1
***************************

 Aantal deelgroepen in de laag : 5
 Deze deelgroepen zijn :
[ (), (1,4)(2,3) ]
orde = 2
[ (), (1,3)(2,4) ]
orde = 2
[ (), (1,3) ]
orde = 2
[ (), (1,2)(3,4) ]
orde = 2
[ (), (2,4) ]
orde = 2

 De resultaten voor laag 2
***************************

 Aantal deelgroepen in de laag : 3
 Deze deelgroepen zijn :
[ (), (1,2)(3,4), (1,3)(2,4), (1,4)(2,3) ]
orde = 4
[ (), (1,2,3,4), (1,3)(2,4), (1,4,3,2) ]
orde = 4
[ (), (2,4), (1,3), (1,3)(2,4) ]
orde = 4

 De resultaten voor laag 3
***************************

 Aantal deelgroepen in de laag : 1
 Deze deelgroepen zijn :
[ (), (2,4), (1,2)(3,4), (1,2,3,4), (1,3), (1,3)(2,4), (1,4,3,2), (1,4)(2,3) ]
orde = 8
gap> G := SymmetricGroup(5);
Sym( [ 1 .. 5 ] )
gap> Read("./Courses/compalg2006/tex/gap/gap3.5.1");
 Deze groep is niet oplosbaar!
gap> G1 := Group([(1,2,3,4)]);
Group([ (1,2,3,4) ])
gap> G2 := Group([(1,2,3,4,5)]);
Group([ (1,2,3,4,5) ])
gap> G := DirectProduct(G1,G2);
Group([ (1,2,3,4), (5,6,7,8,9) ])
gap> Read("./Courses/compalg2006/tex/gap/gap3.5.1");

 De resultaten voor laag 0
***************************

 Aantal deelgroepen in de laag : 1
 Deze deelgroepen zijn :
[ () ]
orde = 1

 De resultaten voor laag 1
***************************

 Aantal deelgroepen in de laag : 2
 Deze deelgroepen zijn :
[ (), (1,3)(2,4) ]
orde = 2
[ (), (5,6,7,8,9), (5,7,9,6,8), (5,8,6,9,7), (5,9,8,7,6) ]
orde = 5
```

```
 De resultaten voor laag 2
***************************

 Aantal deelgroepen in de laag : 2
 Deze deelgroepen zijn :
[ (), (1,2,3,4), (1,3)(2,4), (1,4,3,2) ]
orde = 4
[ (), (5,6,7,8,9), (5,7,9,6,8), (5,8,6,9,7), (5,9,8,7,6), (1,3)(2,4),
  (1,3)(2,4)(5,6,7,8,9), (1,3)(2,4)(5,7,9,6,8), (1,3)(2,4)(5,8,6,9,7),
  (1,3)(2,4)(5,9,8,7,6) ]
orde = 10

 De resultaten voor laag 3
***************************

 Aantal deelgroepen in de laag : 1
 Deze deelgroepen zijn :
[ (), (5,6,7,8,9), (5,7,9,6,8), (5,8,6,9,7), (5,9,8,7,6), (1,2,3,4),
  (1,2,3,4)(5,6,7,8,9), (1,2,3,4)(5,7,9,6,8), (1,2,3,4)(5,8,6,9,7),
  (1,2,3,4)(5,9,8,7,6), (1,3)(2,4), (1,3)(2,4)(5,6,7,8,9),
  (1,3)(2,4)(5,7,9,6,8), (1,3)(2,4)(5,8,6,9,7), (1,3)(2,4)(5,9,8,7,6),
  (1,4,3,2), (1,4,3,2)(5,6,7,8,9), (1,4,3,2)(5,7,9,6,8),
  (1,4,3,2)(5,8,6,9,7), (1,4,3,2)(5,9,8,7,6) ]
orde = 20
```

Hoofdstuk 4

Representatie van groepen op een computer

4.1 Representatie van groepen op een computer

4.1.1 Fundamentele abstracte datatypes

Er zijn drie veel gebruikte methodes om groepen te representeren in een computeralgebrasysteem: als groepen van permutaties die werken op een eindige verzameling; als groepen van matrices over een ring; en als eindig voorgestelde groepen.

Wat permutatiegroepen betreft verkiezen we meestal om te werken met een groep die werkt op de verzameling $\{1, \ldots, n\}$, m.a.w. een deelgroep van $\mathrm{Sym}(n)$. Indien de permutaties werken op een andere verzameling, dan kunnen we de elementen altijd identificeren met een verzameling $\{1, \ldots, r\}$. Als we matrixgroepen beschouwen, dan veronderstellen we in feite dat we elementaire berekeningen kunnen uit voeren in de onderliggende ring die zelf voorgesteld kan worden in een computeralgebrasysteem. Gelet op de beperkingen met betrekking tot de voorstelling van reële en complexe getallen in elk computeralgebrasysteem (zie [4]), zullen we groepen van matrices over deze velden buiten beschouwing moeten laten. Als ring zijn bijvoorbeeld eindige velden, gehele getallen (en sommige ringen van gehelen) toegelaten, evenals het veld der rationale getallen en vele andere.

Een specifiek probleem is de constructie van bijvoorbeeld quotiëntgroepen van permutatiegroepen of matrixgroepen. Een quotëntgroep van een permutatiegroep is niet noodzakelijk eenvoudig als een permutatiegroep voor te stellen. Met eenvoudig voorstellen bedoelen we dat berekeningen de capaciteiten van computers snel kunnen overstijgen. Er zijn voorbeelden gekend van permutatiegroepen van graad $4n$ en orde 8^n, $n \geq 1$, waarvan er quotiëntgroepen bestaan waarvoor de kleinste graad van een getrouwe permutatievoorstelling 2^{2n+1} is.

Een belangrijk aspect bij het ontwerp van technieken en algoritmes is dat we meestal

veronderstellen dat een groep gegeven is aan de hand van zijn generatoren. Wanneer we algoritmes ontwikkelen die bijvoorbeeld een Sylow p-deelgroep van een groep berekenen, dan zullen we een verzameling generatoren bepalen van deze groep.

Wanneer we willen werken met grote eindige groepen, dan is het uiteraard niet wenselijk om alle elementen van een dergelijke groep te berekenen, lussen uit te voeren over de ganse elementenverzameling en alle elementen op te slaan in een lijst. Toch is het niet altijd te vermijden. We weten bijvoorbeeld dat elke eindige groep, via de rechts reguliere actie op zichzelf, een reguliere permutatievoorstelling heeft. Het gebruik van deze representatie is op zich gelijkwaardig met het gebruik van alle elementen van de groep, maar deze worden nu opgeslagen als natuurlijke getallen.

4.1.2 Computationele situaties

Een blauwdruk om alle mogelijke situaties in te delen is als volgt.

1. Elk element van G kan voorgesteld worden in het CAS, daarenboven kunnen inverses en producten van elementen berekend worden, maar het kan mogelijk zijn dat we niet kunnen beslissen of twee voorstellingen van een element hetzelfde element van de groep voorstellen. Een voorbeeld is een eindige voorgestelde groep $\langle X|R \rangle$.

2. Analoog als de vorige situatie, en we kunnen beslissen of twee voorstellingen van een element het zelfde groepselement voorstellen. Voorbeelden zijn permutatiegroepen, gedefinieerd door een verzameling generatoren en matrixgroepen over eindige velden

3. Analoog als vorige situatie, en we beschikken over een speciale verzameling van generatoren $\{g_1, \ldots, g_r\}$ en een algoritme dat voor elk element g van de groep een uniek woord w_g kan berekenen zodat $g = w_g(g_1, \ldots, g_r)$. Een voorbeeld is een permutatiegroep waarbij we beschikken over een zogenaamde verzameling sterk voorbrengende generatoren.

Als een bepaalde groep in beschouwing genomen wordt in een CAS, dan zal de eerste doelstelling zijn om zo snel mogelijk een gunstige computationele situatie op te stellen voor er "iets" gedaan wordt. Uiteraard zullen de methodes om deze situatie te bereiken afhankelijk zijn van de voorstelling van de gegeven groep in het CAS:

1. Stel dat we een eindig voorgestelde groep $\langle X|R \rangle$ beschouwen en dat het geweten is dat deze groep eindig is. Dan zal het in de meeste gevallen nuttig zijn om eerst een getrouwe permutatievoorstelling te bepalen. In zekere zin correspondeert dit met de evolutie van situatie 1 naar situatie 2.

2. Stel dat een groep gegeven is aan de hand van een verzameling voortbrengende permutaties. In de meeste gevallen zal het nuttig zijn om eerst een verzameling

sterk voortbrengende generatoren te berekenen. Dit komt overeen met de evolutie van situatie 2 naar situatie 3.

Situatie 3 is de voorkeur situatie voor computationele doeleinden, deze situatie heeft vele voordelen. Zo spelen bijvoorbeeld de woorden w_g de rol van canonische vorm van de groepselementen. Ze laten toe om de orde van de groep snel te bepalen door het aantal mogelijke canonische woorden te berekenen; dit aantal kan dan eindig of oneindig zijn. Ook kan een lijst opgeslagen worden met alle elementen en kunnen elementen met elkaar vergeleken worden aan de hand van hun canonische vorm. Ook kunnen we snelle algoritmen ontwikkelen om te controleren of een element tot een groep behoort. Op dit soort algoritmen zullen we dieper ingaan in de volgende hoofdstukken.

4.1.3 Straight-line programma's

Stel dat een gegeven groep beschouwd wordt, in situatie 3. Meestal is de speciale verzameling voortbrengers voor de canonische vormen niet dezelfde verzameling als diegene die de gebruiker meegegeven heeft aan het CAS om de groep te definiëren. We zullen in Hoofdstuk 5 zien dat een verzameling sterk voortbrengende generatoren voor een permutatiegroep meestal een verzaling is die de gegeven verzameling generatoren bevat. Het is niet onvoorstelbaar dat we op een gegeven ogenblik deze verzameling sterk voortbrengende generatoren willen relateren aan de originele verzameling generatoren. Om dit te bereiken zullen we een data-structuur definiëren die gekend is als een *streight-line programma* (**E:** *straight-line program*) (SLP). Deze datastructuur levert ook een efficiënte manier om elementen van een groep op te slaan in een lijst. De terminologie komt uit de informatica. Wij zullen het in de volgende vorm gebruiken. Een *SLP groep* (**E:** *SLP group*) is een vrije groep van rang k met generatoren $\hat{x}_1, \ldots, \hat{x}_k$. De elementen van een SLP groep noemen we *SLP elementen* (**E:** *SLP elements*) of *SLPs*. Wiskundig gezien is er natuulijk geen verschil tussen een vrije groep en een SLP groep. Het verschil zit in de manier waarop de elementen van een SLP groep opgeslagen worden. Over het algemeen wordt een SLP element niet opgeslagen als een woord in de originele generatoren, maar als een woord in generatoren die vooraf gedefinieerd zijn. Beschouw een SLP groep van rang 2 en beschouw de volgende elementen:

$$w_1 := \hat{x}_1, w_2 := \hat{x}_2, w_3 := (w_1 w_2)^2, w_4 := w_2^2 w_3^{-1} w_2$$

Deze elementen zullen niet geëvalueerd worden in de oorspronkelijke generatoren, maar opgeslagen worden zoals ze hier gedefinieerd zijn. In het bijzonder wordt het product van twee elementen opgeslagen als dat product, zonder dat dat product uitgewerkt wordt. Dit levert op een natuurlijke wijze de volgende definitie.

Een *evaluatie* (**E:** *evaluation*) φ van een SLP groep S naar een groep G is een toekenning van elementen $x_i \in G$ aan de generatoren $\hat{x}_i$. Omdat S een vrije groep is, bepaalt deze toekenning een uniek homorfisme $\varphi S \to G$ dat $\hat{x}_i$ afbeeldt op x_i. Elk SLP

element is opgeslagen als een (meestal kort) product van woorden, dus de evaluatie van een SLP element kan gebeuren door alle woorden te evalueren.

Nemen we als voorbeeld voor G de groep $\text{Alt}(6)$, $x_1 = (1,2,3), x_2 = (2,3,4,5,6)$, dan evalueren w_1, w_2, w_3 en w_4 respectievelijk naar x_1, x_2, $x_3 := (2,5)(4,6)$ en $x_4 := (4,5,6)$. Het blijkt nu veel eenvoudiger (dus gunstiger vanuit computationeel standpunt) te zijn om elk element van G te schrijven als een product van x_1, x_2, x_3 en x_4, dan enkel van x_1 en x_2. Beschouw nu bijvoorbeeld $g := (1,5)(2,6) \in G$. We vinden $g = x_4 x_3 x_2^{-1} x_1 x_3$. Nu kunnen we de definities van x_3 en x_4 (komende van de corresponderende SLP elementen) gebruiken om g uit te drukken als een product in de oorspronkelijke generatoren. In dit voorbeeld is $x_3 := (x_1 x_2)^2$ en

$$x_4 = x_2^2 x_3^{-1} x_2 = x_2^2 (x_1 x_2)^{-2} x_2 = x_2 x_1^{-1} x_2^{-1} x_1^{-1 x_2}$$

waaruit

$$(1,5)(2,6) = x_2 x_1^{-1} x_2^{-1} x_1^{-1} x_2 x_1 x_2 x_1^2 (x_1 x_2)^2$$

Merk op dat dit niet het kortste woord is voor g in de oorspronkelijke generatoren, maar deze werkwijze geeft ook geen informatie hieromtrent. Het kortste woord voor g is $x_2 x_1 x_2 x_1^{-1} x_2^{-1}$, maar het vinden van het korste woord (op een efficiënte wijze) is in feite een fundamenteel probleem.

De algemene werkwijze start met de definitie van een SLP groep in generatoren $\hat{x}_i$ die corresponderen met initiële generatoren x_i $(1 \leq i \leq k)$ van een groep G, en de evaluatie die $\hat{x}_i$ afbeeldt op x_i. Dan definiëren we SLP elementen $w_1, \ldots, w_r$, (waarbij gebruikelijk $w_i = \hat{x}_i$ met $1 \leq i \leq k$ samen met hun evaluaties in G, waarbij deze gekozen zijn zodanig dat elk element van G als een redelijk kort woord in de generatoren $\{x_1, \ldots, x_r\}$ geschreven wordt. Deze definities in de SLP groep kunnen nu in principe gebruikt worden om elementen van G te gaan schrijven als een woord in de oorspronkelijke generatoren $\{x_1, \ldots, x_k\}$.

4.1.4 Black-box groepen

Met een *black-box groep* (**E:** *black-box group*) bedoelen we een groep G die op de volgende manier voorgesteld kan worden in een CAS. Gegeven zijn een eindig alfabet A en een natuurlijk getal $N > 0$. Alle groepselementen worden voorgesteld door strings in A^* met lengte ten hoogste N. We veronderstellen dat we, gegeven twee strings g en h, in constante tijd g^{-1} en gh kunnen berekenen, en dat we kunnen beslissen of $g =_G h$. Merk op dat we niet (moeten) kunnen beslissen of een gegeven string van lengte ten hoogste N tot G behoort. Merk op dat *membership testing* in feite ook een fundamenteel probleem is. Wanneer we een black-box groep beschouwen dan wil dat in feite zeggen de we ons met deze groep in situatie 2 bevinden, met de belangrijke beperking dat strings ten hoogste N karakters bevatten wat impliceert dat de groep eindig is. We beschikken zelfs over een bovengrens op de grootte van de groep: $\sum_{i=0}^{N} |A|^i$. Eindige permutatiegroepen en matrixgroepen zijn voorbeelden van black-box groepen.

4.2 Random methoden in Computationele groepentheorie

Veel algoritmen in Computationele groepentheorie (en computeralgebra in het algemeen) hangen sterk af van het maken van zogenaamde random keuzes, zoals het bepalen van random elementen van een groep.

4.2.1 Random algoritmen

De uitvoer van een *deterministisch algoritme* (**E:** *deterministic algorithm*) is enkel afhankelijk van de input. Een dergelijk algoritme, uitgevoerd met dezelfde input, zal steeds op dezelfde wijze uitgevoerd worden en dezelfde resultaten teruggeven. De analyse van de complexiteit zal een accurate schatting kunnen geven van de rekentijd en de opslagcapaciteit die vereist is.

Een *random algoritme* (**E:** *randomized algorithm*) maakt gebruik van een random generator tijdens de uitvoering, meestal om bijvoorbeeld willekeurige elementen van een groep te bepalen. Uiteraard zal de uitvoering van een dergelijk algoritme niet enkel van de input afhangen. Het is mogelijk dat verschillende uitvoeringen met dezelfde input grote verschillen in uitvoeringstijd hebben. Het is nog steeds mogelijk om een analyse van de complexiteit te doen, maar nu kunnen we enkel schattingen geven die gemiddelde rekentijden opleveren. We zullen daarbij moeten veronderstellen dat we beschikken over een random generator die willekeurige getallen kan genereren in het interval $[a, b]$. In de praktijk weten we dat perfecte random generatoren niet bestaan, maar daarvoor verwijzen we naar andere literatuur. We gaan ervan uit dat we binnen ons CAS beschikken over deze elementaire mogelijkheden. Belangrijker voor dit gebied is het onderscheid dat we moeten maken in de verschillende random algoritmes in computationele groepentheorie.

Een *Monte Carlo algoritme* (**E:** *Monte Carlo algorithm*) is een random algoritme dat mogelijks een fout antwoord oplevert. Maar het is een vereiste dat een van de input data een reëel getal ϵ is, $0 < \epsilon < 1$ en dat de waarschijnlijkheid op een fout antwoord kleiner dan ϵ is voor alle mogelijke waarden van de resterende input data. De performantie van het algoritme zal afhangen van ϵ, een grotere nauwkeurigheid zal resulteren in langere rekentijden.

Een *Las Vegas algoritme* (**E:** *Las Vegas algorithm*) is een algoritme dat *nooit* een fout antwoord oplevert, maar *geen* antwoord zal opleveren met probabiliteit ϵ, waarbij deze laatste parameter weer behoort tot de input data. In de praktijk kunnen we een dergelijk algoritme meerdere keren uitvoeren met dezelfde input data, en wachten tot het een antwoord oplevert, wat, afhankelijk van ϵ, met een aan zekerheid grenzende waarschijnlijkheid kan gebeuren. In vele implementaties van dergelijke algoritmen gebeurt dit automatisch.

Sommige traditionele wiskundigen (die soms nogal terughoudend zijn in het alge-

meen wat betreft het gebruik van computers :-)) vinden algoritmen die mogelijk een fout antwoord opleveren schrikwekkend[1]. Desondanks hebben Monte Carlo algoritmen hun nut bewezen. We denken bijvoorbeeld aan priemtesten voor gehele getallen, waar Monte Carlo algoritmen significant beter presteren dan de best gekende deterministische methodes en voor praktische toepassingen (bijvoorbeeld cryptografie) goed genoeg zijn.

Andere wiskundigen (die inderdaad vertrouwen op computerresultaten) hebben soms de andere (uiterste) gewoonte om een resultaat met een foutwaarschijnlijkheid kleiner dan 10^{-20} als *bewezen* te beschouwen, met als argument dat veel wiskundige artikels met een grotere waarschijnlijkheid een fout bevatten[2].

In het kader van deze cursusnota's is het niet de bedoeling om in filosofische discussies te vervallen. Het is belangrijker om te benadrukken dat computeralgebra een zeer nuttig instrument kan zijn bij het doen van onderzoek, dat computeralgebra een wiskundige branche is, waar het streven naar correctheid niet verschilt van het streven naar correctheid in alle andere mogelijke wiskundige branches en dat het, eveneens zoals in alle andere wiskundige branches, de moeite loont om verkregen resultaten op een kritische maar verstandige wijze verder te onderzoeken op hun correctheid.

In die zin is het zeer belangrijk dat computeralgebrasystemen die random algoritmes implementeren dat aangeven in de documentatie. Een gebruiker weet dan ten minste of de resultaten op een deterministische wijze bekomen werden (en waarbij de correctheid dus enkel nog afhankelijk is van de implementatie, de onderliggende theoretische correctheid en de correcte werking van de computer) of een random algoritme (waarbij de correctheid daarenboven nog afhankelijk is de input).

4.2.2 Random elementen

We gaan ervan uit dat we een goede random generator hebben en een bijhorende functie die ons willekeurige elemeten uit een interval $[a, b]$ kunnen geven, met a en b twee natuurlijke getallen.

Algoritmen zijn dikwijls afhankelijk van de mogelijkheid om uniform verdeelde random elementen te genereren van een groep G. In sommige gevallen is dit gemakkelijk, bijvoorbeeld $G = \mathrm{Sym}(n)$, of algemeen, bij eindige permutatiegroepen, waar we goede methodes kunnen ontwerpen om willekeurige elementen te genereren. Het kan nuttig zijn om een rij van willekeurige elementen van een groep te genereren in situatie 1 of 2. Voor eindig voorgestelde groepen kunnen we een methode vinden die willekeurige woorden genereert waarvan de lengte binnen een bepaald interval ligt.

In deze paragraaf zullen we het *product replacement algorithm* bespreken. We beschikken over een geordende lijst $X = [x_1, x_2, \ldots, x_r]$ van elementen die G voortbrengen.

[1] "to recoil in horror" uit [3] lijkt zelfs veel sterker dan deze vertaling

[2] Nog andere wiskundigen gebruiken dit argument te pas en te onpas om te verklaren dat zowat niets in de wiskunde als bewezen beschouwd kan worden, doch dit geheel terzijde.

Het blijkt dat $r = 10$ goede resultaten oplevert. Als de lijst X initieel meer elementen bevat, dan nemen we $r \geq 10$. Als de lijst minder elementen bevat, dan herhalen we alle elementen totdat $r = 10$. De kern van het algoritme is het vervangen van x_s door $x_s x_t^{\pm 1}$, waarbij we s en t willekeurig kiezen in het interval $[1, r]$. Merk op dat noodzakelijk X nog steeds de groep voortbrengt. Na een aantal vervangingen (typisch 50) wordt telkens het element x_s ook teruggegeven, als random element. Men kan bewijzen dat deze methode een uniform verdeelde reeks random elementen van G oplevert, maar het is waarschijnlijk dat 50 vervangingen niet voldoende is en een betere afschatting van dit aantal is niet gekend. Het voorgestelde algoritme is een kleine aanpassing van het oorspronkelijk algoritme dat dit idee gebruikt. Er wordt gebruik gemaakt van een *accumulator* (de variabele x_0 in de code) die altijd teruggegeven wordt. Er zijn vermoedens dat deze aanpassing voor een snellere convergentie zorgt dan het origineel algoritme.

We beschikken over een verzameling X generatoren. De initialisatie functie zal een lijst $\mathcal{X}$ initialiseren, het vervangingsproces uitvoeren (m.b.v. de functie PrRandom) en tenslotte deze lijst teruggeven.

Algoritme 4.1 random elementen van een groep: initialisatie

PrInitialize(X,r,n)

  Invoer: $X = [x_1, \ldots, x_k]$ een lijst generatoren voor een black-box groep
  $r, n \in \mathbb{N}$
  Uitvoer: lijst $\mathcal{X}$ en bijhorende SLP elementen $\mathcal{W}$

1 **for** $i \in [1 \ldots k]$
2   **do** $w_i := \hat{x}_i$
3 **for** $i \in [k+1 \ldots r]$
4   **do** $x_i := x_{i-k}$; $w_i := w_{i-k}$;
5 $x_0 := \mathbf{1}_G$; $w_0 := \epsilon$;
6 $\mathcal{X} := [x_0, x_1, \ldots, x_r]$; $\mathcal{W} := [w_0, w_1, \ldots, w_r]$;
7 **for** $i \in [1 \ldots n]$ **do** PrRandom($*\mathcal{X}, *\mathcal{W}$);
8 **return** $\mathcal{X}, \mathcal{W}$

Met de notatie $*\mathcal{X}$ bedoelen we het doorgeven van een variabele die kan gewijzigd worden door de functie die die variabele als input krijgt. Een dergelijke constructie kan eenvoudig gerealiseerd worden in de praktijk door een pointer mee te geven.

Voor sommige toepassingen is het nuttig (of noodzakelijk) om de random elementen te kennen als product in de originele generatoren. Daarom zullen we ook gebruik maken van de SLP elementen w_i die elk element x_i uitdrukken als product in de generatoren $x_1, \ldots, x_k$. Zoals gebruikelijk noteren we de elementen van de SLP groep die naar x_i evalueren met $\hat{x}_i$.

De functie PrRandom die random elementen teruggeeft moet worden uitgevoerd met als input de lijsten $\mathcal{X}, \mathcal{W}$ die geïnitialiseerd werden door de functie PrInitialize. De volgende oproepen worden met dezelfde variabelen uitgevoerd, die door PrRandom zelf telkens gewijzigd worden.

We merken op dat we details in verband met de implementatie van de SLP groepen achterwege laten. Elk SLP element w_i is gedefinieerd als een woord in bestaande SLP elementen, bijgevolg moeten al die elementen ergens opgeslagen worden om telkens de woorden w_i te kunnen evalueren indien gewenst. Wanneer het zeker is dat deze evaluatie niet zal opgevraagd worden, dan zullen we enkel $\mathcal{X}$ meegeven als input voor de functie PrRandom.

Algoritme 4.2 random elementen van een groep

PrRandom($*\mathcal{X}$,$*\mathcal{W}$)

Invoer: $X = [x_1, \ldots, x_k]$ een lijst generatoren voor een black-box groep
$r, n \in \mathbb{N}$
Uitvoer: een element van G en het corresponderende SLP element.

1 s :=Random($[1 \ldots r]$);
2 t :=Random($[1 \ldots r] \setminus \{s\}$);
3 x :=Random($[1 \ldots 2]$);
4 e :=Random($\{1, -1\}$); **if** $x = 1$
5 **then** $x_s := x_s x_t^e$; $x_0 := x_0 x_s$;
6 $w_s := w_s w_t^e$; $w_0 := w_0 w_s$;
7 **else** $x_s := x_t^e x_s$; $x_0 := x_s x_0$;
8 $w_s := w_t^e w_s$; $w_0 := w_s w_0$;
9 **return** x_0, w_0;

Stel dat G een willekeurige groep is, dan zullen we het kiezen van een random element noteren als PrRandom(G), waarbij we ervan uit gaan dat alle noodzakelijke initialisaties gebeurd zijn.

4.3 Enkele elementaire algoritmen

In deze paragraaf bespreken we voorbeelden van structurele berekeningen die kunnen uitgevoerd worden op elke eindig voorgestelde groep waarvan we random elementen kunnen genereren en waarvoor een membership test beschikbaar is. Situatie 3 is een voorbeeld van een dergelijke situatie.

4.3.1 Machten en de orde van een element

Veronderstel dat we ons met een groep G in situatie 2 bevinden en dat we, voor een element $g \in G$, het element g^n, $n \in \mathbb{N}$ willen berekenen. Voor kleine n kunnen we eenvoudig g vermenigvuldigen met $n-1$ zichzelf. Voor grote n bestaat er echter een eenvoudig algoritme (met complexiteit $O(\log(n))$, door g^n te berekenen als g^{2^i}, waarbij elke g^{2^i} kan berekend worden door herhaaldelijk g te kwadrateren. Indien we daarenboven op voorhand wisten dat we g^n moeten berekenen voor verschillende waarden van n, dan kunnen we elementen van de vorm g^{2^i} opslaan in plaats van telkens opnieuw te berekenen.

Algoritme 4.3 macht van een element

POWER(g,n)

Invoer: $g \in G$, $n \in \mathbb{N}$
Uitvoer: g^n

1. $x := \mathbf{1}_G$;
2. **if** $n \mod 2 = 1$
3. **then** $x := xg$;
4. $n := n - 1$;
5. **while** $n > 1$
6. **do** $g := g^2$;
7. $n := \frac{n}{2}$;
8. **if** $n \mod 2 = 1$
9. **then** $x := xg$;
10. $n := n - 1$;
11. **return** x;

Een implementatie van de functie POWER in GAP zou als volgt kunnen zijn (`gap4.2.1`):

```
power := function(G,g,n)
local x;
x := One(G);
if n mod 2 = 1 then
  x := x*g;
  n := n-1;
fi;
while n > 1 do
  g := g^2;
  n := n/2;
  if n mod 2 = 1 then
    x := x*g;
    n := n-1;
  fi;
od;
return x;
end;
```

Veronderstel dat G een black-box groep is en veronderstel dat we voor een element $g \in G$ de orde willen berekenen. A priori is er geen enkele informatie beschikbaar,

zodat we enkel kunnen testen of $g^n = \mathbf{1}_G$ voor oplopende waarden van n. In sommige situaties echter (elementen van eindige velden of matrices bijvoorbeeld of wanneer we in staat zijn om de orde van de groep te berekenen, bijvoorbeeld bij permutatiegroepen) kunnen we een natuurlijk getal n vinden zodanig dat de orde van g het getal n deelt. Op voorwaarde dat we n kunnen ontbinden in priemfactoren (wat voor grote n veel rekentijd kan vergen) kunnen we met het volgende algoritme de orde van g bepalen. (De complexiteit is $O(\log(n)^3)$.

Algoritme 4.4 orde van een element

```
OrderBounded(g,n)
   Invoer: g ∈ G, n ∈ ℕ, |g| deelt n
   Uitvoer: |g|
1  if n = 1 then return 1;
2  for p ∈PrimeDivisors(n)
3        do if Power(g, n/p) = 1_G
4              then return OrderBounded(g, n/p);
5  return n;
```

4.3.2 Normale sluiting

Stel dat $G = \langle X \rangle$ een groep is met een deelgroep $H = \langle Y \rangle$, met X en Y eindig. Veronderstel dat we de normale sluiting $N := \langle H^G \rangle$ willen berekenen. Theoretisch kunnen we beginnen met $N = H$, en elk element y^x, met $y \in Y$, $x \in A := X \cup X^{-1}$, toevoegen aan Y, indien $y^x \notin N$.

In het algemeen is het mogelijk dat N niet eindig voortgebracht is. In dat geval zal de beschreven methode niet eindigen. Wanneer we echter veronderstellen dat $|G : H|$ eindig is, dan levert de beschrijving een deterministisch algoritme op. Wanneer echter $|G : H|$ groot is, dan is het meestal sneller om willekeurige toegevoegden van H onder G toe te voegen aan de verzameling voortbrengers van N, en telkens te controleren of N de normale sluiting is. Het volgend algoritme implementeert deze beschrijving. Het algoritme voegt telkens n willekeurige elementen toe. Deze parameter n geven we mee als input

Algoritme 4.5 normale sluiting

NORMALCLOSURE(X,Y,n)

Invoer: Verzamelingen X, Y die groepen G en H voortbrengen, $n \in \mathbb{N} \setminus \{0\}$
Uitvoer: Generatoren van H^G

```
 1  Z := Y; C := false;
 2  𝒳 :=PrInitialize(X, 10, 20);
 3  while not C;
 4        do 𝒵 :=PrInitialize(X, 10, 20);
 5           for i ∈ [1 . . . n]
 6                do g :=PrRandom(*𝒳);
 7                   h :=PrRandom(*𝒵);
 8                   if h^g ∉ ⟨Z⟩ then Append(*Z, h^g);
 9           C := true;
10           for g ∈ X, h ∈ Z
11                do if h^g ∈ ⟨Z⟩
12                      then C := false;
13                           break;
14  return n;
```

4.4 Homomorfismen

De mogelijkheid om met homomorfismen te werken is een zeer krachtig middel in de computationele groepentheorie. In het bijzonder kunnen we van één representatie van een groep overgaan naar een andere (misschien betere representatie om bepaalde algoritmen uit te voeren). Een groep kan bijvoorbeeld gedefinieerd zijn als een matrix groep of een eindig voorgestelde groep, en we zouden een permutatievoorstelling van deze groep kunnen verkiezen om mee te werken.

In het kader van deze nota's opteren we voor een praktische benadering. Een inleidende behandeling tot homomorfismen kan men vinden in [3, Hoofdstuk 3].

We beschouwen de volgende GAP-sessie (file `gap4.4.1.i`).

```
   Loading the library. Please be patient, this may take a while.
GAP4, Version: 4.4.6 of 02-Sep-2005, powerpc-apple-darwin8.3.0-gcc
Components:  small 2.1, small2 2.0, small3 2.0, small4 1.0, small5 1.0,
             small6 1.0, small7 1.0, small8 1.0, small9 1.0, small10 0.2,
             id2 3.0, id3 2.1, id4 1.0, id5 1.0, id6 1.0, id9 1.0, id10 0.1,
             trans 1.0, prim 2.1  loaded.
Packages:    TomLib 1.1.2  loaded.
gap> f := FreeGroup(["a","b"]);
<free group on the generators [ a, b ]>
gap> G := SymmetricGroup(4);
Sym( [ 1 .. 4 ] )
gap> hom := GroupHomomorphismByImages(f,G,GeneratorsOfGroup(f),
GeneratorsOfGroup(G));
[ a, b ] -> [ (1,2,3,4), (1,2) ]
```

```
gap> g := Random(G);
(1,4,3,2)
gap> PreImagesRepresentative(hom,g);
a^-1
gap> h := Random(f);
b*a^-3
gap> f2 := Group([h]);
Group([ b*a^-3 ])
gap> hom;
[ a, b ] -> [ (1,2,3,4), (1,2) ]
gap> Image(hom);
Group([ (1,2,3,4), (1,2) ])
gap> Image(hom,f2);
Group([ (1,3,4) ])
gap> Image(hom,h);
(1,3,4)
```

Indien we over een homomorfisme $\varphi : G \to H$ beschikken, dan zijn de volgende mogelijkheden gewenst:

(a) Bereken $\varphi(g)$ voor een willekeurige $g \in G$

(b) Voor een $h \in H$, controleer of $h \in \mathrm{Im}(\varphi)$ en zo ja, bereken een $g \in G$, zodat $\varphi(g) = h$.

(c) Bereken $\varphi(K)$, voor een deelgroep $K \leqslant G$.

(d) Bereken $\varphi^{-1}(K)$ voor een deelgroep $K \leqslant H$.

(e) Bereken $\ker(\varphi)$.

De volgende GAP-sessie illustreert het bepalen van de kern van een homomorfisme in GAP (`gap4.4.2.i`).

```
   Loading the library. Please be patient, this may take a while.
GAP4, Version: 4.4.6 of 02-Sep-2005, powerpc-apple-darwin8.2.0-gcc
Components:  small 2.1, small2 2.0, small3 2.0, small4 1.0, small5 1.0,
             small6 1.0, small7 1.0, small8 1.0, small9 1.0, small10 0.2,
             id2 3.0, id3 2.1, id4 1.0, id5 1.0, id6 1.0, id9 1.0, id10 0.1,
             trans 1.0, prim 2.1  loaded.
Packages:    TomLib 1.1.2  loaded.
gap> gens:=[(1,2,3,4),(1,2)];
[ (1,2,3,4), (1,2) ]
gap> g:=Group(gens);
Group([ (1,2,3,4), (1,2) ])
gap> h:=Group((1,2,3),(1,2));
Group([ (1,2,3), (1,2) ])
gap> hom:=GroupHomomorphismByImages(g,h,gens,[(1,2),(1,3)]);
[ (1,2,3,4), (1,2) ] -> [ (1,2), (1,3) ]
gap> Image(hom,(1,4));
(2,3)
gap> Kernel(hom);
Group([ (1,4)(2,3), (1,2)(3,4) ])
gap> Image(hom);
Group([ (1,2), (1,3) ])
gap> Size(Image(hom));
6
gap> Size(Kernel(hom));
4
```

De volgende GAP-sessie illustreert hoe de GAP-functie `PreImagesRepresentative` zich gedraagt.We definiëren een woord in de generatoren a en b van een groep. Dit woord evalueert naar een permutatie, dat ook kan voorgesteld worden door een veel korter woord. Nadien bepalen we in de vrije groep, via `PreImagesRepresentative`, een corresponderend woord (`gap4.4.3.i`).

```
   Loading the library. Please be patient, this may take a while.
GAP4, Version: 4.4.6 of 02-Sep-2005, powerpc-apple-darwin8.3.0-gcc
Components:  small 2.1, small2 2.0, small3 2.0, small4 1.0, small5 1.0,
             small6 1.0, small7 1.0, small8 1.0, small9 1.0, small10 0.2,
             id2 3.0, id3 2.1, id4 1.0, id5 1.0, id6 1.0, id9 1.0, id10 0.1,
             trans 1.0, prim 2.1  loaded.
Packages:    TomLib 1.1.2  loaded.
gap> f := FreeGroup(["a","b"]);
<free group on the generators [ a, b ]>
gap> G := SymmetricGroup(4);
Sym( [ 1 .. 4 ] )
gap> hom := GroupHomomorphismByImages(f,G,GeneratorsOfGroup(f),
GeneratorsOfGroup(G));
[ a, b ] -> [ (1,2,3,4), (1,2) ]
gap> gens := GeneratorsOfGroup(G);
[ (1,2,3,4), (1,2) ]
gap> a := gens[1];
(1,2,3,4)
gap> b := gens[2];
gap> g := a*b*a*b*a^-1;
(1,4,2,3)
gap> a^3;
(1,4,3,2)
gap> PreImagesRepresentative(hom,g);
b^-1*a^-2
gap> b^-1*a^-2;
(1,4,2,3)
```

De laatste sessie beschouwen we een homomorfisme van de permutatiegroep S_4 naar $\mathrm{Sym}(\Omega)$, met Ω de verzameling van ongeoordende koppels elementen uit $\{1, 2, 3, 4\}$ (`gap4.4.4.i`).

```
   Loading the library. Please be patient, this may take a while.
GAP4, Version: 4.4.6 of 02-Sep-2005, powerpc-apple-darwin8.2.0-gcc
Components:  small 2.1, small2 2.0, small3 2.0, small4 1.0, small5 1.0,
             small6 1.0, small7 1.0, small8 1.0, small9 1.0, small10 0.2,
             id2 3.0, id3 2.1, id4 1.0, id5 1.0, id6 1.0, id9 1.0, id10 0.1,
             trans 1.0, prim 2.1  loaded.
Packages:    TomLib 1.1.2  loaded.
gap> g := SymmetricGroup(4);
Sym( [ 1 .. 4 ] )
gap> comb := Combinations([1..4],2);
[ [ 1, 2 ], [ 1, 3 ], [ 1, 4 ], [ 2, 3 ], [ 2, 4 ], [ 3, 4 ] ]
gap> hom := ActionHomomorphism(g,comb,OnSets);
<action homomorphism>
gap> h := Image(hom);
Group([ (1,4,6,3)(2,5), (2,4)(3,5) ])
gap> Size(h);
24
```

4.5 Oefeningen

1. beschouw de volgende matrices over $\mathbb{Q}$

$$m_1 = \begin{pmatrix} \frac{1}{2} & -\frac{1}{2} & \frac{1}{2} & \frac{1}{2} & \frac{1}{2} & \frac{1}{2} & \frac{1}{2} \\ -\frac{1}{4} & \frac{3}{4} & \frac{1}{4} & \frac{1}{4} & \frac{1}{4} & \frac{1}{4} & \frac{1}{4} \\ \frac{1}{4} & \frac{1}{4} & \frac{3}{4} & -\frac{1}{4} & -\frac{1}{4} & -\frac{1}{4} & -\frac{1}{4} \\ \frac{1}{4} & \frac{1}{4} & -\frac{1}{4} & \frac{3}{4} & -\frac{1}{4} & -\frac{1}{4} & -\frac{1}{4} \\ \frac{1}{4} & \frac{1}{4} & -\frac{1}{4} & -\frac{1}{4} & \frac{3}{4} & -\frac{1}{4} & -\frac{1}{4} \\ \frac{1}{4} & \frac{1}{4} & -\frac{1}{4} & -\frac{1}{4} & -\frac{1}{4} & \frac{3}{4} & -\frac{1}{4} \\ \frac{1}{4} & \frac{1}{4} & -\frac{1}{4} & -\frac{1}{4} & -\frac{1}{4} & -\frac{1}{4} & \frac{3}{4} \end{pmatrix}, m_7 = \begin{pmatrix} \frac{1}{2} & \frac{1}{2} & \frac{1}{2} & \frac{1}{2} & -\frac{1}{2} & -\frac{1}{2} & -\frac{1}{2} \\ \frac{1}{4} & \frac{3}{4} & -\frac{1}{4} & -\frac{1}{4} & \frac{1}{4} & \frac{1}{4} & \frac{1}{4} \\ \frac{1}{4} & -\frac{1}{4} & \frac{3}{4} & -\frac{1}{4} & \frac{1}{4} & \frac{1}{4} & \frac{1}{4} \\ \frac{1}{4} & -\frac{1}{4} & -\frac{1}{4} & \frac{3}{4} & \frac{1}{4} & \frac{1}{4} & \frac{1}{4} \\ -\frac{1}{4} & \frac{1}{4} & \frac{1}{4} & \frac{1}{4} & \frac{3}{4} & -\frac{1}{4} & -\frac{1}{4} \\ -\frac{1}{4} & \frac{1}{4} & \frac{1}{4} & \frac{1}{4} & -\frac{1}{4} & \frac{3}{4} & -\frac{1}{4} \\ -\frac{1}{4} & \frac{1}{4} & \frac{1}{4} & \frac{1}{4} & -\frac{1}{4} & -\frac{1}{4} & \frac{3}{4} \end{pmatrix}$$

$$m_2 = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix}, m_3 = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix}$$

$$m_4 = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix}, m_5 = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix}$$

$$m_6 = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{pmatrix}$$

Beschouw de groep m voortgebracht door $m_1, m_2, m_3, m_4, m_5, m_6$ en m_7 en de deelgroep $n \leqslant m$ voortgebracht door m_2, m_3, m_4, m_5, m_6 en m_7.

- Definieer deze groepen in GAP, maak gebruik van de file `gap4.5.1` om deze matrices in te voern in GAP.
- Bepaal de rechtse nevenklassen van n in m.
- Definieer een actie van $\varphi : m \to \mathrm{Sym}(\Omega)$ waarbij $\Omega = \{1, \ldots, N\}$, met N het aantal rechtse nevenklassen van n in m, waarbij φ de actie van m op de rechtse nevenklassen voorstelt door rechtse vermenigvuldiging.
- Bepaal het aantal banen van m op Γ, met Γ de verzameling van ongeordende tripels van rechtse nevenklassen van n in m, waarbij m op Γ werkt door rechtse vermenigvuldiging. Gebruik daarbij de actie die je in het vorig punt definieerde.

2. De volgende methode genereert random elementen van de groep $\mathrm{Sym}(n)$, in de veronderstelling dat we random elementen uit intervallen kunnen genereren. Start met een random keuze voor 1^g. De keuze voor het beeld 2^g moet in de verzameling $\{1, \ldots, n\} \setminus \{1^g\}$ gebeuren, dit kan weer op een willekeurige manier. Implementeer dit algoritme in GAP en/of MAGMA.

3. Implementeer de procedures PrInitialize en PrRandom in GAP of MAGMA, maar laat alle SLP berekenen achterwege. Beschouw de groep $\mathrm{GL}(d, q)$ en laat deze procedures random elementen van deze groep genereren.

4. Implementeer de functie OrderBounded in GAP en/of MAGMA.

Hoofdstuk 5

Eindige permutatiegroepen

5.1 Banen en stabilisatoren

We veronderstellen in deze sectie dat G een permutatiegroep is voortgebracht door de geordende verzameling $X = [x_1, \ldots, x_r]$ en dat G werkt op een eindige verzameling Ω. Verder veronderstellen we ook dat voor elke $\alpha \in \Omega$ en elke $x \in X$, het beeld α^x kan berekend worden, en dat we eenvoudig kunnen beslissen of twee elementen $x, y \in \Omega$ aan elkaar gelijk zijn. In het geval dat $\Omega = \{1, \ldots, n\} \subset \mathbb{N}$ zijn deze veronderstellingen op een vrij eenvoudige manier te realiseren. Als Ω bijvoorbeeld de verzameling van deelgroepen is van G, waarbij de actie toevoeging is, dan is deze veronderstelling veel moeilijker te realiseren. We denken onder andere aan het testen van de gelijkheid tussen twee deelgroepen. Een eenvoudig algoritme om de baan van $\alpha \in \Omega$ onder G te berekenen is het volgende.

Algoritme 5.1 baan van een element

ORBIT(α,X)

Invoer: een geordende verzameling generatoren $X = [x_1, \ldots, x_r]$ voor G, $\alpha \in \Omega$

Uitvoer: de baan α^G

```
1  Δ = [α];
2  for β ∈ Δ
3       do for x ∈ X
4               do if β^x ∉ Δ
5                     then voeg β^x toe aan Δ;
6  return Δ;
```

Wat betreft de for-lus op lijn 3 hebben we dezelfde opmerking als in Hoofdstuk 1,

pagina 8.

Alle elementen die toegevoegd worden aan Δ zijn het beeld van een element $\beta \in \Delta$ onder een element $x_i \in X$. Bijgevolg is noodzakelijkg $\Delta \subseteq \alpha^G$. Stel nu dat $\beta = \alpha^g$ voor een element $g \in G$. Omdat X de groep G voortbrengt, geldt dat $g = x_{i_1} x_{i_2} \ldots x_{i_k}$ met alle $x_{i_j} \in X$. We tonen door middel van inductie op k dat β tot de geconstrueerde Δ behoort. Dit is waar voor $k = 0$, omdat dan $\beta = \alpha$ en Δ geïnitialiseerd werd als $[\alpha]$. We veronderstellen dat $\gamma := \alpha^{g'} \in \Delta$ voor $g' = x_{i_1} x_{i_2} \ldots x_{i_{k-1}}$, met een vaste $k > 0$. Het volgt onmiddellijk dat $\beta = \gamma^{x_{i_k}}$ zal worden toegevoegd aan Δ. We besluiten dat $\beta \in \Delta$.

In de veronderstelling dat beelden α^x kunnen berekend worden in constante tijd, en dat in constante tijd kan gecontroleerd worden of elementen van Ω in Δ zitten, is het eenvoudig in te zien dat de complexiteit van dit algoritme $O(|\Delta| r)$ is.

In gap en magma kunnen we ORBIT eenvoudig implementeren (files `gap5.1.1` en `gap5.1.1.i`, `magma5.1.1` en `magma5.1.1.i`, hier tonen we enkel de implementatie in magma).

```
orbit := function(delta,S)
 local new,o,s,gamma;
 o := {};
 new := {delta};
 while not ( #new eq 0 ) do
   o := new join o;
   new := {};
   for gamma in o do
     for s in S do
       if not (gamma^s in o) then
         new:=Include(new,gamma^s);
       end if;
     end for;
   end for;
 end while;
 return o;
end function;
```

```
Magma V2.8-10     Wed Feb  1 2006 15:51:25 on colossus [Seed = 431657122]
Type ? for help.  Type <Ctrl>-D to quit.
> load "magma5.1.1";
Loading "magma5.1.1"
> G:=PermutationGroup<11| (1,4,5,11,6,10,3,2)(7,8),(1,4,5,11,6,10,3,2)(8,9),(1,4,2,3)(5,11,10,6) >;
> S:=Generators(G);
> orbit(1,S);
{ 1, 2, 3, 4, 5, 6, 10, 11 }
> orbit(7,S);
{ 7, 8, 9 }
> IsTransitive(G);
false
> Orbit(G,1);
GSet{ 1, 2, 3, 4, 5, 6, 10, 11 }
> Orbit(G,7);
GSet{ 7, 8, 9 }
```

```
   Loading the library. Please be patient, this may take a while.
GAP4, Version: 4.4.6 of 02-Sep-2005, powerpc-apple-darwin8.3.0-gcc
Components:  small 2.1, small2 2.0, small3 2.0, small4 1.0, small5 1.0,
             small6 1.0, small7 1.0, small8 1.0, small9 1.0, small10 0.2,
             id2 3.0, id3 2.1, id4 1.0, id5 1.0, id6 1.0, id9 1.0, id10 0.1,
             trans 1.0, prim 2.1  loaded.
Packages:    TomLib 1.1.2  loaded.
gap> Read("./Courses/compalg2006/tex/gap/gap5.1.1");
gap> G:=Group([ (1,2,3,4)(5,6,7,8),(1,5)(2,8)(3,7)(4,6) ]);
Group([ (1,2,3,4)(5,6,7,8), (1,5)(2,8)(3,7)(4,6) ])
gap> S:=GeneratorsOfGroup(G);
[ (1,2,3,4)(5,6,7,8), (1,5)(2,8)(3,7)(4,6) ]
gap> orbit(1,S);
[ 1, 2, 3, 4, 5, 6, 7, 8 ]
gap> Orbit(G,1);
```

```
[ 1, 2, 5, 3, 8, 6, 4, 7 ]
gap> IsTransitive(G);
true
gap> IsTransitive(G,[1..8]);
true
```

In bepaalde situaties zijn we niet enkel geïnteresseerd in de baan α^G zelf, maar willen we ook voor ieder element $\beta \in \alpha^G$, een element $u_\beta \in G$ kennen, waarvoor $\alpha^{u_\beta} = \beta$. We weten dat de verzameling $\{u_\beta | \beta \in \alpha^G\}$ een rechtse transversaal is van de stabilizator G_α van α in G. Men kan de volgende stelling aantonen (zie [3]).

Stelling 5.1.1. $G_\alpha = \langle \{u_\beta x u_{\beta^x}^{-1} | \beta \in \alpha^G, x \in X\} \rangle$

De elementen van de verzameling in het rechterlid worden de *Schreiergeneratoren* (**E:** *Schreier generators*) van G_α genoemd. De volgende functie, OrbitStabilizer, geeft een lijst Δ terug die paren (β, u_β) bevat, voor alle $\beta \in \alpha^G$, en een lijst generatoren Y van G_α.

Merk op dat het mogelijk is dat een berekende Schreiergenerator de identieke is. Immers, wanneer een element β^x geïntroduceerd wordt in Δ, dan geldt natuurlijk dat $u_{\beta^x} = u_\beta x$, waaruit volgt dat de geassocieerde Schreiergenerator de identieke is. We zeggen dat de Schreiergenerator *per definitie triviaal* is. We noteren in dit geval $u_{\beta^x} \equiv u_\beta x$. Een dergelijke generator voegen we natuurlijk niet toe aan de lijst Y. Het is echter ook mogelijk dat $u_\beta x = u_{\beta^x}$ maar $u_\beta x \not\equiv u_{\beta^x}$. Het voorgestelde algoritme zal in dat geval toch de identieke toevoegen aan Y.

Algoritme 5.2 baan van een element en stabilisator

OrbitStabilizer(α,X)
  Invoer: $\alpha \in \Omega$, $X = [x_1, \ldots, x_r]$, $x_i \in \mathrm{Sym}(\Omega)$ met $\langle X \rangle = G$
  Uitvoer: Δ, Y zoals boven beschreven.
1 $\Delta := [(\alpha, \mathbf{1}_G)]$;
2 $Y := []$;
3 **for** $(\beta, u_\beta) \in \Delta$
4   **do for** $x \in X$ **do if** $\beta^x \notin \Delta$
5   **then** voeg $(\beta^x, u_\beta x)$ aan Δ;
6   **else** voeg $u_\beta x (u_{\beta^x})^{-1}$ toe aan Y;
7 **return** Δ, Y;

Een belangrijk en veel gebruikt hulpmiddel om de elementen van de transversaal te berekenen is de zogenaamde Schreiervector. We veronderstellen dat $\Omega = \{1 \ldots n\}$ voor een bepaalde $n \in \mathbb{N}$. Indien (G, Ω) een permutatievoorstelling is van hoge graad (b.v. $10^6, 10^7$) die transitief werkt, dan willen we het opslaan van de elementen u_β vermijden.

We voeren in de volgende definitie het begrip Scheiervector in. We veronderstellen nog steeds dat de groep wordt voortgebracht door een verzameling $X = [x_1, \ldots, x_k]$.

Definitie 5.1.2. *Een* Schreiervector *(***E:** Schreier vector*) geassocieerd aan een element $\alpha \in \Omega$ is een lijst v van lengte n die de volgende eigenschappen heeft:*

- $v[\alpha] = -1$,
- *voor alle $\gamma \in \alpha^G \setminus \{\alpha\}, v[\gamma] := i \in \{1, \ldots, k\}$, als γ werd toegevoegd aan de lijst Δ als β^{x_i}, gebruikmakend van het algoritme* Orbit*,*
- $v[\beta] = 0$ *voor alle* $\beta \notin \alpha^G$.

Dit houdt ook in dat v kan gebruikt worden als een karakteristieke functe van de baan α^G. De volgende functie OrbitSv, is een aangepaste versie van de functie Orbit die meteen de Schreiervector bepaalt.

Algoritme 5.3 baan van een element en Schreiervector

OrbitSv(α,X)
  Invoer: een geordende verzameling generatoren $X = [x_1, \ldots, x_r]$ voor G, $\alpha \in \Omega$
  Uitvoer: de baan α^G en bijhorende Schreier vector v
1 **for** $i \in [1..n]$ **do** $v[i] := 0$;
2 $\Delta = [\alpha]$; $v[\alpha] := -1$;
3 **for** $\beta \in \Delta$
4   **do for** $x \in X$ **do if** $\beta^x \notin \Delta$
5   **then** voeg β^x toe aan Δ; $v[\beta^{x_i}] := i$;
6 **return** Δ, v;

We voeren het algoritme uit op het volgende voorbeeld. Beschouw een deelgroep G van Sym(n), $n = 7$, voortgebracht door $x_1 = (1, 3, 7)(2, 5)$ en $x_2 = (3, 4, 6, 7)$. We passen OrbitSv toe voor $\alpha = 1$. Δ wordt geïnitialiseerd als [1] en v als $[-1, 0, 0, 0, 0, 0, 0]$. De hoofdlus start met $\beta = 1$. Eerst wordt x_1 gebruikt, wat tot toevoegen van 3 leidt, en $v[3] = 1$. Gebruik van x_2 levert niets op, want $1^{x_2} = 1 \in \Delta$. We gaan verder met $\beta = 3$, $\beta^{x_1} = 7 \notin \Delta$, 7 wordt dus toegevoegd en $v[7] = 1$; $\beta^{x_2} = 4 \notin \Delta$, 4 wordt dus toegevoegd en $v[4] = 2$. We gaan verder met $\beta = 7$, $\beta^{x_1} = 1 \in \Delta$, er wordt niets toegevoegd, hetzelfde geldt voor x_2. We gaan verder met $\beta = 4$, $\beta^{x_1} = 4 \in \Delta$; $\beta^{x_2} = 6 \notin \Delta$, dus 6 wordt toegevoegd en $v[6] = 2$. Tenslotte zijn 6^{x_1} en 6^{x_2} element van Δ, de lus wordt dus beëindigd. We vinden $\Delta = \{1, 3, 7, 4, 6\}$ en $v = [-1, 0, 1, 2, 0, 2, 1]$.

We kunnen nu een Schreiervector gebruiken om een element u_β te berekenen.

Algoritme 5.4 trace functie

TRACE(β,v,X)

Invoer: een geordende verzameling generatoren $X = [x_1, \ldots, x_r]$ voor G
Schreiervector v voor een $\alpha \in \Omega$.
Uitvoer: u_β, zodat $\alpha^{u_\beta} = \beta \iff \beta \in \alpha^G$, `false` als $\beta \notin \alpha^G$.

1 **if** $v[\beta] = 0$
2 **then return** `false`;
3 $u := \mathbf{1}_G$; $k := v[\beta]$;
4 **while** $k \neq -1$
5 **do** $u := x_k \cdot u$;
6 $\beta := \beta^{x_k^{-1}}$;
7 $k := v[\beta]$;
8 **return** u;

Passen we deze functie toe op bovenstaand voorbeeld met $\beta = 6$, dan vinden we achtereenvolgens $v[6] = 2$, dus $u := x_2$ en $\beta := 6^{x_2^{-1}} = 4$, dan vinden we $v[4] = 2$, u wordt x_2^2, $\beta := 4^{x_2^{-1}} = 3$. Dan vinden we $v[3] = 1$, dus u wordt $x_1 x_2^2$ en β wordt $3^{x_1^{-1}} = 1$. Tenslotte wordt de uitvoering beëindigd met $v[1] = -1$, en $u = (1, 6, 3, 4, 7)(2, 5)$ wordt teruggegeven.

De functies ORBITSV en TRACE kunnen eenvdouig in GAP en Magma geïmplementeerd worden. (files: `gap5.1.2`, `gap5.1.2.i`, `magma5.1.2` en `magma5.1.2.i`, enkel de GAP voorbeelden zijn opgenomen.)

```
orbitsv := function(delta,S,omega)
 local new,o,s,gamma,v;
 o:=[];
 v:=[];
 for gamma in omega do
   v[gamma]:=0;
 od;
 new:=[delta];
 v[delta] := -1;
 while not ( Length(new) = 0 ) do
   o := Union(new,o);
   new := [];
   for gamma in o do
     for s in S do
       if not (gamma^s in o) then
         Add(new,gamma^s);
         v[gamma^s]:=Position(S,s);
       fi;
     od;
   od;
 od;
 return [o,v];
end;

trace := function(beta,v,S)
  local u,k;
  if v[beta] = 0 then
    return false;
  else
    u := ();
    k := v[beta];
```

```
    while not (k=-1) do
      u := S[k]*u;
      beta := beta^(S[k]^(-1));
      k := v[beta];
    od;
  fi;
return u;
end;


   Loading the library. Please be patient, this may take a while.
GAP4, Version: 4.4.6 of 02-Sep-2005, powerpc-apple-darwin8.3.0-gcc
Components:  small 2.1, small2 2.0, small3 2.0, small4 1.0, small5 1.0,
             small6 1.0, small7 1.0, small8 1.0, small9 1.0, small10 0.2,
             id2 3.0, id3 2.1, id4 1.0, id5 1.0, id6 1.0, id9 1.0, id10 0.1,
             trans 1.0, prim 2.1  loaded.
Packages:    TomLib 1.1.2  loaded.
gap> Read("./Courses/compalg2006/tex/gap/gap5.1.2");
gap> a := (1,3,7)(2,5);
(1,3,7)(2,5)
gap> b := (3,4,6,7);
(3,4,6,7)
gap> S := [a,b];
gap> r := orbitsv(1,S,[1..7]);
[ [ 1, 3, 4, 6, 7 ], [ -1, 0, 1, 2, 0, 2, 1 ] ]
gap> v := r[2];
[ -1, 0, 1, 2, 0, 2, 1 ]
gap> trace(6,v,S);
(1,6,3,4,7)(2,5)
```

Om deze paragraaf te eindigen beschrijven we nog een eenvoudige toepassing. Stel dat een baan α^G gegeven is. Met behulp van de functie TRACE kunnen we de elementen u_β berekenen, met behulp van de functie PRRANDOM kunnen we random elementen van G berekenen. Stel dat $g \in G$ zo een random element is, en stel $\beta := \alpha^g$, dan is gu_β^{-1} random in G_α.

Algoritme 5.5 random stabilizator van een element

RANDOMSTAB(α,v,X)

Invoer: $G = \langle X \rangle$, Schreiervector v voor baan α^G

Uitvoer: een random element van G_α.

1 $g :=$PRRANDOM(G).

2 $h :=$TRACE(α^g, v, X); **return** gh^{-1};

5.2 Controle op alternerende of symmetrische groep

In deze paragraaf beschrijven we een snel en eenvoudig Monte Carlo algoritme om te testen of een groep die transitief werkt op een verzameling Ω gelijk is aan $\mathrm{Alt}(\Omega)$ of $\mathrm{Sym}(\Omega)$, op voorwaarde dat $|\Omega| \geq 8$. Een positief antwoord is gegarandeerd correct, terwijl een negatief antwoord fout kan zijn met een bepaalde probabiliteit. Merk op dat in het geval van een positief antwoord het eenvoudig is om na te gaan of de groep dan gelijk is aan $\mathrm{Alt}(\Omega)$ of $\mathrm{Sym}(\Omega)$, het volstaat om te controleren of alle generatoren van de group even permutaties zijn.

Wanneer een onbekende permutatiegroep G in beschouwing genomen wordt, die werkt op Ω, dan kunnen we met de algoritmes uit vorige sectie de transitiviteit nagaan. Indien de groep transitief werkt, dan kunnen we nagaan of het $\mathrm{Alt}(\Omega)$ of $\mathrm{Sym}(\Omega)$ betreft met het hier voorgestelde algoritme. Het algoritme is gebaseerd op de volgende stelling

Stelling 5.2.1. *Beschouw een groep $G \leqslant \mathrm{Sym}(\Omega)$ die transitief werkt op Ω, $|\Omega| = n$. Stel dat p een priemgetal is, $\frac{n}{2} < p < n-2$. Als een element van G een cykel van lengte p bevat, dan is G isomorf met $\mathrm{Alt}(\Omega)$ of met $\mathrm{Sym}(\Omega)$.*

Het algoritme IsAltSym gebruikt deze stelling en wordt als volgt opgebouwd. We kiezen $N(n,\epsilon)$ random elementen uit G. Indien een gekozen element een p-cykel bevat, met $\frac{n}{2} < p < n-2$, dan geven we `true` terug. Merk op dat hieruit de voorwaarde $n \geq 8$ volgt, omdat er geen priemgetallen liggen tussen $\frac{n}{2} < p < n-2$ voor $n < 8$. Uit stelling 5.2.1 volgt dat een antwoord `true` correct is. Het getal $N(n,\epsilon)$ moet nu bepaald worden waardoor een antwoord `false` met probabiliteit kleiner dan ϵ fout is. Het is geweten dat de verhouding tussen het aantal elementen dat een p-cykel bevat, voor alle priemgetallen p, $\frac{n}{2} < p < n-2$, en het totaal aantal elementen van de groep, is minstens

$$\sum_p \frac{1}{p}$$

waarbij gesommeerd wordt over alle priemgetallen $\frac{n}{2} < p < n-2$. Uit de getaltheorie kennen we een afschatting voor deze som: $\frac{\ln(2)}{\ln(n)}$, maar voor kleine waarden van n ($n \leq 185$), is deze afschatting (veel) te groot. We kiezen een waarde $c(n)$ als volgt: $c(n) := 0.34$ voor $8 \leq n \leq 16$ en $c(n) := 0.57$ voor $n > 16$. Dan is de afschatting $c(n)\frac{\ln(2)}{\ln(n)}$ steeds een ondergrens voor de som. Stel $d(n) := c(n)\frac{\ln(2)}{\ln(n)}$, dan is dit dus een ondergrens voor de verhouding elementen die een p-cykel bevatten ten opzichte van alle elementen. De kans dat we na het kiezen van k random elementen nog steeds geen element dat een p-cykel bevat, gekozen hebben is dus $(1-d(n))^k < e^{-d(n)k}$, waaruit we $N(n,\epsilon) := -\frac{\ln(\epsilon)}{d(n)}$ stellen, wat garandeert dat de kans op het kiezen van een niet p-cykel uit de groep $\mathrm{Alt}(\Omega)$ of $\mathrm{Sym}(\Omega)$ na $N(n,\epsilon)$ keuzes, kleiner maakt dan ϵ.

Algoritme 5.6 controle op alternerende of symmetrische groep

IsAltSym(G, ϵ)

Invoer: een groep G, probabiliteit ϵ.
Uitvoer: `true` of `false`

1 Initialiseer $N(n, \epsilon)$.
2 **for** k in $[1..N(n, \epsilon)]$
3 **do** g := PrRandom(G);
4 **if** g bevat p-cycel
5 **then return** `true`;
6 **return** `false`;

We illustreren deze functie in GAP (files: `gap5.2.1` en `gap5.2.1.i`)

```
isaltsym := function(G,s)
local N,g,i,bool,lengths,n,primes;
n := NrMovedPoints(G);
if n > 16 then
  N := QuoInt(100*(LogInt(s,2)+1)*(LogInt(n,2)+1),48)+1;
else
  N := QuoInt(100*(LogInt(s,2)+1)*(LogInt(n,2)+1),82)+1;
fi;
bool := false;
primes := [];
i := QuoInt(n,2);
i := NextPrimeInt(i);
while i < n-2 do
  Add(primes,i);
  i := NextPrimeInt(i);
od;
i := 1;
while ((not bool) and (i < N)) do
  g := Random(G);
  lengths := CycleLengths(g,[1..n]);
  if Length(Intersection(lengths,primes)) > 0 then
    bool := true;
  fi;
  i := i+1;
od;
return bool;
end;
```

```
gap> n := 49;
49
gap> perm := [];
[  ]
gap> perm[49] := 1;
1
gap> for i in [1..48] do
> perm[i] := i+1;
> od;
gap> g := Sortex(perm);
(1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,25,26,27,28,
29,30,31,32,33,34,35,36,37,38,39,40,41,42,43,44,45,46,47,48,49)
gap> h := (1,2);
(1,2)
gap> G := Group([g,h]);
<permutation group with 2 generators>
gap> Read("./Courses/compalg2006/tex/gap/gap5.2.1");
gap> s := 2;
2
gap> isaltsym(G,s);
true
gap> h := (47,48,49);
(47,48,49)
```

```
gap> H := Group([g,h]);
<permutation group with 2 generators>
gap> isaltsym(H,s);
true
gap> g := (4,8)(5,6)(9,10);
(4,8)(5,6)(9,10)
gap> h := (2,5)(3,4)(7,10)(8,9);
(2,5)(3,4)(7,10)(8,9)
gap> i := (1,3,5,2,4)(6,8,10,7,9);
(1,3,5,2,4)(6,8,10,7,9)
gap> I := Group([g,h,i]);
Group([ (4,8)(5,6)(9,10), (2,5)(3,4)(7,10)(8,9), (1,3,5,2,4)(6,8,10,7,9) ])
gap> isaltsym(I,s);
false
```

5.3 Bloksystemen

5.3.1 Inleiding

Beschouw een groep $G \leqslant \mathrm{Sym}(n)$ voortgebracht door een verzameling generatoren $\{x_1, \ldots, x_r\}$. Met de algoritmen uit sectie 5.1 kunnen we eenvoudig nagaan of G transitief werkt op Ω. Indien G inderdaad transitief werkt, dan is de natuurlijke vraag of G primitief werkt op Ω. In deze paragraaf veronderstellen we dat G transitief werkt op Ω. We herhalen dat G primitief werkt op Ω als en slechts als G geen niet triviale partitie van Ω invariant laat. Indien we dus een bloksysteem kunnen vinden, dan kunnen we meteen besluiten dat G imprimitief werkt op Ω. We bespreken in deze paragraaf een algoritme om een bloksysteem te bepalen, indien er een bestaat. Beslissen of een permutatiegroep imprimitief werkt, m.a.w het vinden van een bloksysteem, is een van de (weinige) computationele problemen over permutatiegroepen die efficiënt kunnen opgelost worden zonder de kennis van een zogenaamde verzameling van sterk voortbrengende generatoren.

Voor we dit algoritme beschouwen, voeren we een alternatieve manier in om bloksystemen te beschrijven.

Definitie 5.3.1. *Stel dat een permutatiegroep G op Ω werkt. Een equivalentierelatie $\sim$ op Ω wordt een G-*congruentie *(***E:** *G-*congruence*) genoemd als $\alpha \sim \beta$ impliceert dat $\alpha^g \sim \beta^g$ voor alle $\alpha, \beta \in \Omega$ en alle $g \in G$.*

Veronderstel dat G transitief op Ω werkt. Stel dat er een bloksysteem $\{\Delta^g | g \in G\}$ is voor deze werking. Dan definiëren we als volgt een equivalentierelatie op Ω: $\alpha \sim \beta$ als en slechts als α en β behoren tot het zelfde blok. Het is duidelijk dat deze relatie een G-congruentie is. Het omgekeerde is echter ook waar: als er een G-congruentie bestaat, dan definiëren de equivalentieklassen een bloksysteem. M.a.w. bloksystemen en G-congruenties zijn equivalent.

Definitie 5.3.2. *Als $\mathcal{S}$ een relatie is over Ω, dan is de G-congruentie voortgebracht door $\mathcal{S}$ de doorsnede van alle G-congruenties die $\mathcal{S}$ bevatten.*

In het kader van deze definitie is het vinden van een bloksysteem dus equivalent met het vinden van de G-congruentie voortgebracht door $\{(\alpha_1, \alpha_i)|2 \leq i \leq k\}$.

Het basis algoritme dat we beschrijven is in staat het kleinste blok van G te vinden dat $k > 1$ elementen $\alpha_1, \ldots, \alpha_k \in \Omega = \{1 \ldots n\}$ bevat, samen met de andere blokken in het geassocieerde bloksysteem. Uiteraard is het mogelijk dat er geen dergelijk bloksysteem bestaat. Door de initiële verzameling $\{\alpha_1, \ldots, \alpha_k\}$ te variëren, kunnen we effectief een bloksysteem vinden of aantonen dat er geen bestaat, door eenvoudig het algoritme uit te voeren op de $n - 1$ initiële verzamelingen $\{1, \alpha\}$, $2 \leq \alpha \leq n$.

Deze procedure kan zelfs efficiënter. Het is duidelijk dat als α en β in dezelfde baan liggen van de stabilizator G_1 van het punt 1, dat een blok de elementen $1, \alpha$ bevat als en slechts als het de elementen $1, \beta$ bevat. We moeten daarom niet de $n - 1$ koppels $\{1, \alpha\}$, $2 \leq \alpha \leq n$, gebruiken, maar enkel de koppels $\{1, \beta\}$ met β die varieert over de representanten van de banen van G_1. We vermelden wel dat het bepalen van de groep G_1 op de meest efficiënte wijze gebeurt indien er een zogenaamde basis en verzameling sterk voortbrengende generatoren gekend is. Indien deze niet beschikbaar is, kunnen we echter wel een random stabilizatorgroep van 1 berekenen met behulp van de functie RandomStab. Een deelgroep van G_1 kunnen we snel bepalen door deze functie enkele keren op te roepen.

Indien elke G-congruentie voortgebracht door $\{1, \alpha\}$, $2 \leq \alpha \leq n$ een triviale relatie is, wat zich in dit geval manifesteert doordat deze relatie maar een equivalentieklasse heeft, namelijk Ω, dan besluiten we dat de groep G primitief werkt op Ω.

5.3.2 Het atkinson algoritme

Het voorgestelde algoritme MinimalBlock manipuleert een geïnitialiseerde equivalentierelatie $\sim$ op $\{1 \ldots n\}$. De k gegeven punten worden in één klasse geplaatst, alle overige punten van Ω vormen een verschillende klasse. Tijdens de uitvoering zullen sommige van deze klassen samengevoegd worden. Na de uitvoering is de aangepaste $\sim$ een G-congruentie voortgebracht door verzameling $\{(\alpha_1, \alpha_i)|2 \leq i \leq k\}$. Op elk ogenblik zullen we van elke klasse de representant opslaan. We vermelden nog dat representanten van samengevoegde klassen die overbodig worden door de samenvoeging, opgeslagen worden in een lijst q van lengte l. We initialiseren deze lijst als $\{\alpha_i|2 \leq i \leq k\}$. De While-lus doorloopt de punten in de lijst q. De equivalentieklasse waartoe een punt α behoort is genoteerd als Class(α), en de representant als Rep(α). Zoals gebruikelijk veronderstellen we dat de groep G voortgebracht wordt door een lijst X van generatoren.

Algoritme 5.7 een minimaal bloksysteem

```
MinimalBlock(G,{α_1, ..., α_k})
    Invoer: een groep G = ⟨X⟩, die transitief werkt.
    Uitvoer: G-congurentie ~ voortgebracht door {(α_1, α_i)|2 ≤ i ≤ k}
 1  Initialiseer ~ als {{α_i|1 ≤ i ≤ k} ∪ {γ}|γ ≠ α_i(1 ≤ i ≤ k)}}.
 2  for i ∈ [1 ... k − 1]
 3      do q[i] := α_{i+1};
 4  l := k − 1; i := 1;
 5  while i ≤ l
 6      do γ := q[i]; i := i + 1;
 7         for x ∈ X
 8             do δ :=Rep(γ);
 9                κ :=Rep(γ^x);
10                λ :=Rep(δ^x);
11                if κ ≠ λ
12                  then Voeg klasse Class(κ) en Class(λ) samen;
13                       Kies κ representant van de nieuwe klasse;
14                       l := l + 1; q[l] :=Rep(λ).
15  return ~;
```

We voeren dit algoritme uit op de groep $G := \langle x_1, x_2 \rangle$, $x_1 = (1, 2, 3, 4, 5, 6)$, $x_2 = (2, 6)(3, 5)$. Stel $\alpha_1 = 1$, $\alpha_2 = 3$. De representant van de klasse $\{1, 3\}$ wordt 1, 3 wordt op de lijst q geplaatst. De lus wordt gestart met $\gamma = 3$. Met $x = x_1$ krijgen we $\delta = 1$, $\kappa = 4$, $\lambda = 2$, de klassen $\{2\}$ en $\{4\}$ worden dus samengevoegd, we kiezen 4 als representant en plaatsen 2 op de lijst q. Met $x = x_2$ krijgen we $\delta = 1$, $\kappa = 5$, $\lambda = 1$, waarmee we dus de klassen $\{1, 3\}$ en $\{5\}$ samenvoegen. We kiezen 5 als representant, en plaatsen 1 op de lijst q. Het volgende element in de lijst q, $\gamma = 2$, levert, met $x = x_1$, $\delta = 4$ en $\kappa = \lambda = 5$ maar met $x = x_2$, $\delta = 5$ en $\kappa = 6$, $\lambda = 4$, dus worden de klassen $\{2, 4\}$ en $\{6\}$ samengevoegd, we kiezen 6 als representant en plaatsen 4 in de lijst q. Met $\gamma = 1$, $x = x_1$ vinden we $\delta = 5$, $\kappa = \lambda = 6$, $x = x_2$ geeft $\delta = 5$, $\kappa = \lambda = 5$. Tenslotte geeft $\gamma = 4$, $x = x_1$, $\delta = 6$, $\kappa = \lambda = 5$ en $x = x_2$, $\delta = 6$, $\kappa = \lambda = 6$. De equivalentieklassen zijn dus $\{1, 3, 5\}$ en $\{2, 4, 6\}$, en deze bepalen een bloksysteem.

We tonen nu aan dat het algoritme wel degelijk aan de beschrijving voldoet.

Stelling 5.3.3. *De equivalentierelatie bepaald door het algoritme* MinimalBlock *is de G-congruentie voortgebracht door $\{(\alpha_1, \alpha_i)|2 \le i \le k\}$.*

Proof. We noteren de relatie die teruggegeven wordt door MinimalBlock als $\sim_F$. De G-congruentie die voortgebracht wordt door $\{(\alpha_1, \alpha_i)|2 \le i \le k\}$ noteren we als $\equiv$.

We bewijzen dat $\sim_F = \equiv$.

We tonen eerst aan dat uit $\mu \sim \nu$, op elk moment in de uitvoering en voor alle $\mu, \nu \in \Omega$, $\mu \equiv \nu$ volgt. Na de initialisatie is dit in elk geval waar, omdat dan $\{\alpha_i | 1 \leq i \leq k\}$ de enige niet triviale $\sim$-klasse is. Wanneer twee klassen samengevoegd worden tijdens de uitvoering, dan bevatten ze de beelden γ^x en δ^x ($\gamma, \delta \in \Omega$, $x \in G$), waarbij $\gamma \sim \delta$ voor het samenvoegen. We veronderstellen dat $\gamma \equiv \delta$ en gaan verder door volledige inductie. Omdat $\equiv$ een G-congruentie is, geldt $\gamma^x \equiv \delta^x$, en de eigenschap $\mu \sim \nu$ impliceert $\mu \equiv \nu$ blijft waar na samenvoegen. Per inductie hebben we aangetoond dat deze eigenschap altijd geldt, en dus $\sim_F \subseteq \equiv$.

Omgekeerd, om aan te tonen dat $\equiv \subseteq \sim_F$ is het voldoende om aan te tonen dat $\sim_F$ een G-congruentie is, omdat ze zeker $\{(\alpha_1, \alpha_i) | 2 \leq i \leq k\}$ bevat. We moeten dus aantonen dat $\mu \sim_F \nu$ $\mu^g \sim_F \nu^g$ impliceert voor alle $g \in G$. Het is duidelijk dat het volstaat om dit aan te tonen voor de generatoren van G. Merk ook op dat telkens wanneer een punt λ toegevoegd wordt aan de lijst q, er geldt dat $\text{Rep}(\lambda) = \lambda$ voor λ toegevoegd werd en $\text{Rep}(\lambda) \neq \lambda$ nadat λ toegevoegd werd. Dit toont trouwens ook aan dat elk element hoogstens eenmaal aan de lijst q wordt toegevoegd, en dat het algoritme dus noodzakelijk eindigt na een eindig aantal stappen.

Stel dat l_F de lengte van de lijst q is op het einde van de uitvoering. Voor $\lambda \in [1 \ldots n]$ definiëren we $w(\lambda) = k$ als $q[k] = \lambda$ en $w(\lambda) = l_F + 1$ als λ nog niet voorkomt in de lijst q op het einde van de uitvoering. We gebruiken inductie op $z := 2l_F + 2 - w(\mu) - w(\nu)$. Als $z = 0$, dan behoren noch μ noch ν tot de lijst, dus, gezien de opmerking hierboven, $\text{Rep}(\mu) = \mu$ en $\text{Rep}(\nu) = \nu$. Maar $\mu \sim_F \nu$ impliceert dat $\mu = \nu$, dus valt er in dit geval niets te bewijzen. Als $z > 0$, dan behoort ten minste een van de elementen μ, ν tot de lijst, stel bijvoorbeeld $\mu = q[k]$. Wanneer we $i = k$ bereiken tijdens de uitvoering en we de actie van de generator x van G beschouwen, hebben we $\gamma = \mu$ en $\delta = \text{Rep}(\gamma) \neq \gamma$. Op dit punt is δ een $\sim_F$-klasse representant en behoort het dus niet tot de lijst q. Daaruit volgt $w(\delta) > w(\gamma)$. De klassen van γ^x en δ^x worden nu samengevoegd, dus $\gamma^x \sim_F \delta^x$. Omdat $w(\delta) > w(\gamma)$ hebben we dat $\delta^x \sim_F \nu^x$ door de inductiehypothese, waaruit het resultaat volgt. ■

5.3.3 Het samenvoegen van klassen

Voor de volledigheid beschrijven we het samenvoegen van klassen. De informatie over de $\sim$-klassen is opgeslagen in een array p met n posities, die voldoet aan de voorwaarden $p[\alpha] \sim \alpha$ en $p[\alpha] = \alpha \iff \text{Rep}(\alpha)$. Het opvragen van een representant gebeurt door de volgende implementatie van Rep.

Algoritme 5.8 representant van een klasse

REP(κ, p)

Invoer: $\kappa \in [1 \ldots n]$, array p

Uitvoer: Representant van de klasse die κ bevat.

1 $\lambda := \kappa$; $\rho := p[\lambda]$;
2 **while** $\rho \neq \lambda$
3 **do** $\lambda := \rho$; $\rho := p[\lambda]$;
4 **return** λ;

De volgende procedure voegt twee klassen samen. We maken gebruik van een tweede array c. Deze array bevat op positie α de kardinaliteit van de klasse die α bevat. Men kan aantonen dat het iets efficiënter is om de nieuwe representant te kiezen uit de klasse met de grootste kardinaliteit. De procedure voegt twee klassen samen, past de arrays c en p aan, en slaat de verwijderde representant op in de lijst q. Voor de arrays c en p en de lijst q wordt er daarom een pointer meegegeven.

Algoritme 5.9 samenvoegen van klassen

MERGE(κ,λ, $*c$, $*p$, $*q$, $*l$)

Invoer: $\kappa, \lambda \in \Omega$, c,p,q,l.

1 $\phi :=$REP(κ, p); $\psi :=$REP$(\lambda, *p)$;
2 **if** $\phi \neq \psi$
3 **then if** $c[\phi] \geq c[\psi]$
4 **then** $\mu := \phi$; $\nu := \psi$;
5 **else** $\mu := \psi$; $\nu := \phi$;
6 $p[\nu] := \mu$; $c[\mu] := c[\mu] + c[\nu]$;
7 $l := l + 1$; $q[l] := \nu$;

Tenslotte geven we een versie van MINIMALBLOCK die REP en MERGE gebruikt.

Algoritme 5.10 een minimaal bloksysteem (2)

MinimalBlock(G,$\{\alpha_1, \dots, \alpha_k\}$)

Invoer: een groep $G = \langle X \rangle$, die transitief werkt.
Uitvoer: G-congurentie $\sim$ voortgebracht door $\{(\alpha_1, \alpha_k) | 2 \leq i \leq k\}$

```
1  for i ∈ [1...n] do p[i] := i; c[i] := 1;
2  for i ∈ [1...k − 1] do p[α_{i+1}] := α_1;
3  for i ∈ [1...k − 1] do q[i] := α_{i+1};
4  c[α_1] := k; i := 1; l := k − 1;
5  while i ≤ l
6        do γ := q[i]; i := i + 1;
7           for x ∈ X
8                do δ :=Rep(γ, *p);
9                   Merge(γ^x, δ^x, *c, *p, *q, *l);
10 for i ∈ [1...n] do Rep(i, *p);
11 return p;
```

We geven een kort praktisch voorbeeld in GAP (file: `gap5.3.1.i`).

```
   Loading the library. Please be patient, this may take a while.
GAP4, Version: 4.4.6 of 02-Sep-2005, powerpc-apple-darwin8.3.0-gcc
Components:  small 2.1, small2 2.0, small3 2.0, small4 1.0, small5 1.0,
             small6 1.0, small7 1.0, small8 1.0, small9 1.0, small10 0.2,
             id2 3.0, id3 2.1, id4 1.0, id5 1.0, id6 1.0, id9 1.0, id10 0.1,
             trans 1.0, prim 2.1  loaded.
Packages:    TomLib 1.1.2  loaded.
gap> G := Group([(1,2)(3,4)(5,9)(6,7),(2,5,7),(2,4,5,8,7,3)(6,9),(3,4,8)]);
Group([ (1,2)(3,4)(5,9)(6,7), (2,5,7), (2,4,5,8,7,3)(6,9), (3,4,8) ])
gap> Order(G);
162
gap> orbits := Orbits(G,[1..9]);
[ [ 1, 2, 5, 4, 9, 7, 8, 3, 6 ] ]
gap> blocks := Blocks(G,[1..9]);
[ [ 1, 6, 9 ], [ 2, 5, 7 ], [ 3, 4, 8 ] ]
```

5.4 Basis en sterk voortbrengende generatoren

5.4.1 Inleiding

In 1970 voerde C. Sims[1] een ketting van deelgroepen van een eindige permutatiegroep in, die een belangrijke rol zou spelen bij de ontwikkeling van efficiënte algoritmen voor eindige permutatiegroepen.

[1] Charles C. Sims. *Computational methods in the study of permutation groups.* in *Computational problems in abstract algebra.* J. Leech, ed. Oxford 1970. (Oxford 1967), Pergamon Press

We beginnen deze paragraaf met de invoering van alle notaties en definities in verband met het nieuwe concept. We veronderstellen opnieuw dat G een eindige permutatiegroep is die werkt op $\Omega = \{1, \ldots, n\}$ en dat G voortgebracht wordt door een geordende verzameling S van elementen van $\mathrm{Sym}(n)$. Stel dat $B = [\beta_1, \ldots, \beta_k]$ een geordende rij van verschillende elementen is van Ω. Definieer $G^{(i)} := G_{\beta_1,\ldots,\beta_{i-1}}$ voor $1 \leq i \leq k+1$, de **fixator** van de rij $[\beta_1, \ldots, \beta_{i-1}]$ in G. (dus $G^{(1)} = G$).

Voor $1 \leq i \leq k+1$ definiëren we $S^{(i)} := S \cap G^{(i)}$, $H^{(i)} := \langle S^{(i)} \rangle$ en $\Delta^{(i)} := \beta_i^{H^{(i)}}$. We noteren de rechtse transversaal van $H^{(i)}_{\beta_i}$ in $H^{(i)}$ als $U^{(i)}$. Door de orbit-stabilizer stelling weten we dat $U^{(i)} = \{u^{(i)}_\beta | \beta \in \Delta^{(i)}\}$, waarbij β_i door $u^{(i)}_\beta$ afgebeeld wordt op β.

Wanneer B en S gekend zijn, dan kunnen de verzamelingen $S^{(i)}$ eenvoudig berekend worden, waaruit de baan $\Delta^{(i)}$ en een bijhorende Schreiervector v_i met behulp van OrbitSv kan berekend worden. Met behulp van v_i en de functie Trace kunnen we de verzamelingen $U^{(i)}$ berekenen. Met de notatie Δ^* bedoelen we een datastructuur die de banen $\Delta^{(i)}$ en de transversalen $U^{(i)}$ bevat.

Een geordende rij B wordt een *basis* (**E:** *base*) genoemd als en slechts als het enige element van G dat alle elementen $\beta_1, \ldots, \beta_k$ fixeert de identieke is. Met andere woorden, $G^{(k+1)} = \{\mathbf{1}\}$ en

$$\{\mathbf{1}\} = G^{(k+1)} \leqslant G^{(k)} \leqslant \ldots \leqslant G^{(2)} \leqslant G^{(1)} = G$$

De geordende rij S wordt een *verzameling van sterk voortbrengende generatoren* (**E:** *strong generating set*) ten opzichte van een basis B genoemd, als en slechts als $H^{(i)} = \langle S^{(i)} \rangle = G^{(i)}$ voor alle $i = 1 \ldots k+1$. Dit is per definitie waar voor $i = 1$. We zullen een korte notatie invoeren: een koppel (B, S) is een *basis en sterk voortbrengende verzameling* (**E:** *basis en strong generating set*), genoteerd BSGS, als en slechts als B een basis is voor de groep G en S een verzameling sterk voortbrengende generatoren voor de groep G ten opzichte van de basis B. We noemen $G^{(i)}$ de *i-de basis stabilizator* (**E:** *i-th basis stabilizer*) en $\Delta^{(i)} = \beta_i^{G^{(i)}}$ de *i-de basisbaan* (**E:** *i-th basic orbit*).

Als (B, S) een BSGS is, dan beschikken we via de transversalen $U^{(i)}$, $i = 1, \ldots, k$ over een handige manier om elk element te schrijven als een uniek produkt $g = u_k u_{k-1} \ldots u_1$, $u_i \in U^{(i)}$. Merk tenslotte ook op dat de orde van groep G gelijk is aan

$$|G| = |U^{(k)}| \cdot |U^{(k-1)}| \cdots |U^{(2)}| \cdot |U^{(1)}|$$

Als B een basis is en $g \in G$, dan noemen we de geordende rij $[\beta_1^g, \ldots, \beta_k^g]$ een *basisbeeld* (**E:** *base image*) van B onder g.

Lemma 5.4.1. *Het basisbeeld van B onder g bepaalt $g \in G$ uniek*

Bewijs. Als $B^g = B^h$ voor twee elementen $g, h \in G$, dan geldt uiteraard $B^{gh^{-1}} = B$, maar enkel $\mathbf{1} \in G$ fixeert de basis B volledig, waaruit $g = h$ volgt. ∎

Eén van de voordelen die de representatie van elementen van een groep als basisbeelden heeft, is het feit dat een basis van een groep dikwijls (veel) minder elementen bevat

dan de graad van de groep. Het is ook duidelijk dat een groep G, van graad n, met een basis van lengte k, een orde heeft die hoogstens n^k is. De symmetrische groep S_n daarentegen heeft geen basissen die minder dan $n-1$ elementen bevatten. Kiezen we als basis $B = [n, n-1, \ldots, 3, 2]$, dan is $G^{(i)} \cong S_{n-i+1}$. Momenteel is het zo dat het concept "basis en sterk voortbrengende generatoren" aan drie belangrijke eigenschappen voldoet die we als volgt kunnen omschrijven:

Universaliteit: Voor de meerderheid van de fundamentele algoritmen voor permutatiegroepen blijkt ee BGSG een zeer efficiënte voorstelling van de groep te zijn.

Erfelijkheid: Bijna alle algoritmen die gebruikt worden om deelgroepen te construeren en homomorfe beelden van permutatiegroepen hebben de eigenschap dat de deelgroep of het beeld de BSGS erft van de oorspronkelijke groep

Beschikbaarheid: Er zijn efficiënte algoritmen beschikbaar om een BSGS te construeren.

We geven een voorbeeld van een BSGS voor de groep G voortgebracht door $a = (1, 2, 4, 5, 7, 3, 6)$ en $b = (2, 4)(3, 5)$. Deze groep heeft orde 168 en stelt de actie van $\mathrm{P\Gamma L}(3, 2)$ op de punten van $\mathrm{PG}(2, 2)$ voor. Een basis is bijvoorbeeld $B = [1, 2, 4]$ en een verzameling sterk voortbrengende generatoren $S = [s_1 = a, s_2 = b, s_3 = (4, 5)(6, 7), s_4 = (4, 6)(5, 7)]$. De stabilisatoren zijn $G = G^{(1)} = \langle a, b\rangle$, $G_1 = G^{(2)} = \langle b, s_3, s_4\rangle$, $G_{1,2} = G^{(3)} = \langle s_3, s_4\rangle$. De transversalen zijn $U^{(1)} = \{\mathbf{1}, a, a^2, a^3, a^4, a^5, a^6\}$, $U^{(2)} = \{\mathbf{1}, b, bs_3, bs_4$ $, bs_3b, bs_3s_4\}$ en $U^{(3)} = \{\mathbf{1}, s_3, s_4, s_3s_4\}$. De Schreiervectoren zijn $v^{(1)} = [0, 1, 1, 1, 1, 1, 1, 1]$, $v^{(2)} = [0, 0, 2, 2, 3, 4, 4]$, $v^{(3)} = [0, 0, 0, 0, 3, 4, 4]$.

In MAGMA kunnen we dergelijke gegevens eenvoudig opvragen (file `magma5.4.1.i`)

```
Magma V2.8-10     Mon Feb 13 2006 14:59:15 on colossus [Seed = 2302302753]
Type ? for help.  Type <Ctrl>-D to quit.
> G<a,b,c> := PermutationGroup<11|(1,4,5,11,6,10,3,2)(7,8),
(1,4,5,11,6,10,3,2)(8,9),(1,4,2,3)(5,11,10,6)>;
> BSGS(G);
> Base(G);
[ 7, 1, 2, 3 ]
> print BasicStabiliserChain(G);
[
    Permutation group G acting on a set of cardinality 11
    Order = 1008 = 2^4 * 3^2 * 7
        (1, 4, 5, 11, 6, 10, 3, 2)(7, 8)
        (1, 4, 5, 11, 6, 10, 3, 2)(8, 9)
        (1, 4, 2, 3)(5, 11, 10, 6),
    Permutation group acting on a set of cardinality 11
    Order = 336 = 2^4 * 3 * 7
        (1, 4, 5, 11, 6, 10, 3, 2)(8, 9)
        (1, 4, 2, 3)(5, 11, 10, 6),
    Permutation group acting on a set of cardinality 11
    Order = 42 = 2 * 3 * 7
        (2, 10, 11, 6, 4, 3)(8, 9)
        (3, 11, 10)(4, 5, 6),
    Permutation group acting on a set of cardinality 11
    Order = 6 = 2 * 3
        (3, 11, 10)(4, 5, 6)
        (3, 4)(5, 11)(6, 10)(8, 9),
    Permutation group acting on a set of cardinality 11
    Order = 1
]
> print StrongGenerators(G);
{@
    (1, 4, 5, 11, 6, 10, 3, 2)(7, 8),
    (1, 4, 5, 11, 6, 10, 3, 2)(8, 9),
    (1, 4, 2, 3)(5, 11, 10, 6),
    (2, 10, 11, 6, 4, 3)(8, 9),
    (3, 11, 10)(4, 5, 6),
    (3, 4)(5, 11)(6, 10)(8, 9)
```

```
@}

> print BasicOrbits(G);
[
    {@ 7, 8, 9 @},
    {@ 1, 4, 5, 2, 11, 3, 6, 10 @},
    {@ 2, 10, 11, 6, 4, 3, 5 @},
    {@ 3, 11, 10, 4, 5, 6 @}
]
> print SchreierVectors(G);
[
    [ 0, 1, 2 ],
    [ 0, 2, 2, 3, 2, 3, 2, 3 ],
    [ 0, 4, 4, 4, 4, 4, 5 ],
    [ 0, 5, 5, 6, 6, 6 ]
]
>
```

Voor we een algoritme bespreken dat, gegeven een permutatiegroep G, een BSGS construeert, beschrijven we eerst een aantal eenvoudige algoritmen die fundamenteel zijn en die het nut van het concept BSGS aantonen.

Het algoritme STRIP voert de zogenaamde membership test uit. Gegeven een groep G en een element g, dan is het nuttig om te testen of $g \in G$. De eerste versie van het voorgestelde algoritme maakt gebruik van de datastructuur Δ^*. De tweede versie maakt gebruik van oproepen van de functies TRACE en ORBITSV.

Algoritme 5.11 membership test

STRIP(g,B,S,Δ^*)

Invoer: $g \in \mathrm{Sym}(\Omega)$, B,S,Δ^*
Uitvoer: $h \in \mathrm{Sym}(\Omega)$ en $i \in [1 \ldots k+1]$

1 $h := g$;
2 **for** $i \in [1 \ldots k]$
3 **do** $\beta := \beta_i^h$;
4 **if** $\beta \notin \Delta^{(i)}$ **then return** h, i;
5 Bereken $u_i \in U^{(i)}$ zodat $\beta_i^{u_i} = \beta$;
6 $h := hu_i^{-1}$;
7 **return** $h, k+1$;

Algoritme 5.12 membership test (2)

STRIP(g,X,B,S)

```
    Invoer: g ∈ Sym(Ω), B,S, X brengt G voort
    Uitvoer: h ∈ Sym(Ω) en i ∈ [1 ... k + 1]
1   for i ∈ [1 ... k]
2       do Δ^(i), v_i := OrbitSv(β_i, X)
3   h := g;
4   for i ∈ [1 ... k]
5       do β := β_i^h;
6           if β ∉ Δ^(i) then return h, i;
7           u_i := Trace(β, v_i, X);
8           h := h u_i^{-1};
9   return h, k + 1;
```

Als het teruggegeven element h de identieke is, dan kunnen we besluiten dat $g = u_k u_{k-1} \dots u_1$ en dus $g \in G$. Ook het omgekeerde geldt. Indien niet de identieke teruggegeven werd, dan werd ook een getal $i \leq k$ teruggegeven, en dan weten we dat de test $\beta_i \in \Delta^{(i)}$ faalde. We kunnen het algoritme lichtjes aanpassen om voor een gegeven $g \in G$, de unieke factorisatie $u_k u_{k-1} \dots u_1$ te berekenen.

Algoritme 5.13 membership test (3)

FACTOR(g,X,B,S)

```
     Invoer: g ∈ G, B,S, X brengt G voort
     Uitvoer: h ∈ Sym(Ω) en i ∈ [1 ... k + 1]
 1   for i ∈ [1 ... k]
 2       do Δ^(i), v_i := OrbitSv(β_i, X)
 3   h := g;
 4   word := ϵ (het "lege" woord, of het eenheidselement van de groep)
 5   for i ∈ [1 ... k]
 6       do β := β_i^h;
 7           u_i := Trace(β, v_i, X);
 8           word := u_i × word;
 9           h := h u_i^{-1};
10   return word;
```

Met een opnieuw kleine aanpassing, kunnen we hetzelfde algoritme gebruiken om aan de hand van een gegeven B, en een basisbeeld B^g, voor een onbekende $g \in G$, het element g te bepalen.

Algoritme 5.14 beeld van een basis

ELEMENTBASEIMAGE(g,X,B,B^g)

Invoer: B,S, X brengt G voort, basisbeeld B^g, voor een $g \in G$.
Uitvoer: $g \in G$

1 **for** $i \in [1 \ldots k]$
2   **do** $\Delta^{(i)}, v_i :=$ORBITSV(β_i, X)
3 $g := \mathbf{1}$;
4 $[\gamma_1, \ldots, \gamma_k] := B^g$;
5 **for** $i \in [1 \ldots k]$
6   **do** $u_i :=$TRACE(β, v_i, X);
7         $g := u_i g$;
8         $[\gamma_1, \ldots, \gamma_k] := [\gamma_1, \ldots, \gamma_k]^{u_i^{-1}}$;
9 **return** g;

Deze functie laat zich in gap eenvoudig implementeren. We gaan wel een paar specifieke GAP-functies gebruiken. Stel dat G een permutatie groep is. De GAP-functie `StabChain(G)` berekent een BSGS voor de groep G en bijhorende basisbanen $\Delta^{(i)}$ en Schreiervectoren $v^{(i)}$. Al deze gegevens worden opgeslagen in een record, dat een aantal velden heeft, en waarbij het laatste veld kan verwijzen naar een record van hetzelfde type. Het teruggegeven resultaat is dus een recursieve structuur. Deze bundel gegevens speelt een rol zoals de dataverzameling Δ^* die we in onze pseudocode gebruikten. De GAP-functie `InverseRepresentative` implementeert in feite onze TRACE functie. Deze GAP-functie neemt als eerse argument een record zoals teruggegeven door `StabChain`, en een permutatie g. (files: `gap5.4.1` en `gap5.4.1.i`).

```
permutationbybaseimage :=
    function(S,baseim)
    local dummy,i,g;
    i := 1;
    dummy := ();
    repeat
      g := baseim[i]^dummy;
      if not g in S.orbit then
        return fail;
      else
        dummy := dummy*InverseRepresentative(S,g);
      fi;
      if IsBound(S.stabilizer) then
        S := S.stabilizer;
        i := i + 1;
      fi;
    until not IsBound(S.stabilizer);
    return dummy^-1;
end;
```

```
   Loading the library. Please be patient, this may take a while.
GAP4, Version: 4.4.6 of 02-Sep-2005, powerpc-apple-darwin8.2.0-gcc
Components:  small 2.1, small2 2.0, small3 2.0, small4 1.0, small5 1.0,
             small6 1.0, small7 1.0, small8 1.0, small9 1.0, small10 0.2,
             id2 3.0, id3 2.1, id4 1.0, id5 1.0, id6 1.0, id9 1.0, id10 0.1,
             trans 1.0, prim 2.1  loaded.
Packages:    TomLib 1.1.2  loaded.
gap> a := (1,8,9)(2,11,15)(3,10,12)(4,14,19)(5,16,17)(6,21,20)(7,13,18);
(1,8,9)(2,11,15)(3,10,12)(4,14,19)(5,16,17)(6,21,20)(7,13,18)
gap> b := (9,18,20)(12,19,17);
(9,18,20)(12,19,17)
gap> c := (10,21,11)(13,16,14);
(10,21,11)(13,16,14)
gap> G := Group([a,b,c]);
Group([ (1,8,9)(2,11,15)(3,10,12)(4,14,19)(5,16,17)(6,21,20)(7,13,18),
  (9,18,20)(12,19,17), (10,21,11)(13,16,14) ])
gap> S := StabChain(G);
<stabilizer chain record, Base [ 9, 1, 8, 2, 10, 12 ], Orbit length 21, Size:
27783>
gap> baseim := [3,21,18,13,19,6];
[ 3, 21, 18, 13, 19, 6 ]
gap> permutationbybaseimage(S,baseim);
(1,21,9,3,11,15,4,14,17)(2,13,20,7,10,19,5,8,18)(6,16,12)
```

We kunnen het concept BSGS gebruiken om een algoritme te ontwerpen dat alle elementen van een groep bepaalt. Dit algoritme maakt gebruik van het feit dat we voor elk element $g \in G$ een unieke factorisatie in elementen $u_i \in U^{(i)}$ hebben. Het is echter niet nuttig dit uit te voeren voor grote groepen. Dit recursief algoritme wordt opgeroepen met $l = 1$ en $h = \mathbf{1}$.

Algoritme 5.15 opsomming van alle elementen

ENUMERATE(B,S, Δ^*,l,h,$*list$)

Invoer: B,S, Δ^*, l: niveau van de recursie
$*list$ pointer naar lijst waarin de elementen opgeslagen worden.
h: element uit G uit de vorige laag van de recursie.
Uitvoer: alle elementen van G in lijst $list$; $k := |B|$;

1 **for** u in $U^{(l)}$
2   **do** $g := u \cdot h$; **if** $l = k$
3   **then** voeg g toe aan $list$;
4   **else** ENUMERATE(B,S,Δ^*,$l+1$,g,$*list$);
5 **return** $list$;

Tenslotte kunnen we ook een algoritme ontwerpen om alle basisbeelden te bepalen. We ontwerpen dit weer als een recursief algoritme.

Algoritme 5.16 opsomming van alle basisbeelden

EnumerateBaseImage(B, Δ^*,l,g,$*list$)

Invoer: Δ^*, l: niveau van de recursie
$B = [\gamma_1, \gamma_2, \ldots, \gamma_{l-1}]$ beginsequentie voor basisbeelden.
$g = u_{l-1} \cdot u_{l-2} \cdot \ldots \cdot u_1$: een element $g \in G$
$*list$ pointer naar lijst waarin de basisbeelden opgeslagen worden
Uitvoer: alle elementen van G in lijst $list$; $k := |B|$;

```
1 if l = k + 1 (* k is lengte van basis, eventueel te vinden in Δ**)
2    then Voeg B toe aan list.
3    else
4         for alle γ_i ∈ (Δ^(i))^g
5              do u_i :=Trace(γ_i^(g^-1), v^(i)) met licht misbruik van de argumenten van Trace
6                 EnumerateBaseImage(B, Δ*, l + 1, u_l · g, *list)
```

Een oproep EnumerateBaseImage($[], \Delta^*, 1, \mathbf{1}, *list$) zorgt voor de berekening van alle basisbeelden en slaat die op in een lijst $list$.

5.4.2 Het Schreier-Sims algoritme

We beschrijven nu een basis versie van het zogenaamde Schreier-Sims algoritme om voor een gegeven groep G een BSGS te bepalen. Het algoritme is gebaseerd op het volgende resultaat.

Lemma 5.4.2. *Stel dat B,S, $S^{(i)}$ en $H^{(i)}$ gedefinieerd zijn zoals boven. Dan is (B, S) een BSGS voor G als en slechts als $H^{(k+1)} = \{\mathbf{1}\}$ en $H^{(i)}_{\beta_i} = H^{(i+1)}$ voor alle $1 \leq i \leq k$.*

Bewijs. Per definitie is (B, S) een BSGS als en slechts als $G^{(k+1)} = \{\mathbf{1}\}$ en $H^{(i)} = G^{(i)}$ voor alle $1 \leq i \leq k$. Omdat $H^{(1)} = G^{(1)}$ en $G^{(i)}_{\beta_i} = G^{(i+1)}$ voor alle $1 \leq i \leq k$ is de voorwaarde uit het lemma zeker noodzakelijk opdat (B, S) een BSGS zou zijn. Omgekeerd, als de voorwaarde geldig is, dan zien we door inductie op i dat $H^{(i)} = G^{(i)}$ voor alle $1 \leq i \leq k+1$ en de veronderstelling $H^{(k+1)} = \{\mathbf{1}\}$ impliceert dat $G^{(k+1)} = \{\mathbf{1}\}$. ∎

Het besproken algoritme, SchreierSims genaamd, neemt twee argumenten als invoer: een (mogelijks) lege lijst B en een verzameling generatoren S voor de groep G. Het algoritme zal de beide verzamelingen uitbreiden totdat (B, S) een BSGS voor G is. Het algoritme begint met de uitbreiding van de verzameling B (indien nodig) door te controleren dat elke generator van S niet alle elementen van B vasthoudt. Indien dit wel zo is,

dan worden elementen van Ω toegevoegd. Na deze initialisatie worden de basisbanen en bijhorende transversalen bepaald. Op dat ogenblik geldt zeker $H^{(k+1)} = \{\mathbf{1}\}$, en er wordt gestart met de controle van de voorwaarden $H^{(i)}_{\beta_i} = H^{(i+1)}$ voor alle $i = k, k-1, \ldots 1$. Dit wordt in de hoofdlus van het algoritme gedaan, die begint op lijn 7. Omdat $H^{(i+1)} \leqslant H^{(i)}_{\beta_i}$ moeten we enkel controleren of $H^{(i)}_{\beta_i} \leqslant H^{(i+1)}$, wat we kunnen doen door te controleren of alle generatoren van $H^{(i)}_{\beta_i}$ behoren tot $H^{(i+1)}$.

Deze generatoren kunnen we berekenen met behulp van de stelling van Schreier, zoals we die gebruikt hebben in de functie OrbitStabilizer (Stelling 5.1.1). In deze implementatie zullen we ook controleren dat er geen Schreiergeneratoren toegevoegd worden die per definitie triviaal zijn. Het is gebruikelijk om de controles te doen startend bij k en zo afdalend naar 1, omdat we dan zeker zijn dat als de voorwaarde voldaan is voor een i, deze dan ook voldaan is voor grotere i, waardoor we de functie Strip kunnen gebruiken om te controleren of de gevonden generatoren tot $H^{(i+1)}$ behoren.

Als deze controle faalt voor een Schreiergenerator g, dan stellen we de Boolean variabele gelijk aan `false`, en voegen we het element h, dat teruggegeven wordt door Strip, toe aan de verzameling $S^{(i+1)}$. Dit zorgt voor een vervanging van $H^{(i+1)}$ door een grotere groep (die uiteraard deelgroep is van $G^{(i+1)}$). Als h alle elementen van B fixeert, dan voegen we een nieuw element uit Ω dat niet gefixeerd wordt door h, toe aan B, waarbij de voorwaarde $H^{(k+1)} = \{\mathbf{1}\}$ behouden blijft.

Voor een bepaalde i in de functie SchreierSims is het zo dat elk element dat toegevoegd wordt aan $S^{(i)}$ de groep $H^{(i)}$ groter maakt, zodat we maar een eindig aantal generatoren zullen kunnen toevoegen. Het beschreven algoritme zal dus eindigen. Als het eindigt, dan zullen alle voorwaarden uit Lemma 5.4.2 voldaan zijn, zodat (B, S) een BSGS is voor G.

Algoritme 5.17 het Schreier-Sims algoritme

SchreierSims($*B$,$*S$)

Invoer: B en S, zoals boven beschreven.

1 **for** $x \in S$ **do if** $x \in G_{\beta_1,\ldots,\beta_k}$
2   **then** Zoek $\gamma \in \Omega$ met $\gamma^x \neq \gamma$;
3   Voeg γ toe aan B; $k := k+1$;
4 **for** $i \in [1 \ldots k]$
5   **do** $S^{(i)} := S \cap G_{\beta_1,\ldots,\beta i-1}$; $H^{(i)} := \langle S^{(i)} \rangle$; $\Delta^{(i)} := \beta_i^{H^{(i)}}$;
    Nu is $H^{(k+1)} = \{\mathbf{1}\}$.
6   $i := k$;
7 **while** $i \geq 1$
8   **do** (test voorwaarde $H^{(i)}_{\beta_i} = H^{(i+1)}$)
9   **for** $\beta \in \Delta^{(i)}$
10   **do** Bereken $u_\beta \in H^{(i)}$ zodat $\beta_i^{u_\beta} = \beta$;
11   **for** $x \in S^{(i)}$ **do if** $u_\beta x \not\equiv u_{\beta^x}$
12   **then** (Controleer of Schreier generator $u_\beta x u_{\beta^x}^{-1} \in H^{(i+1)}$)
13   y :=`true`;
14   h, j :=Strip($u_\beta x u_{\beta^x}^{-1}$,$B$,$S$,$\Delta^*$);
15   **if** $j \leq k$
16   **then** (Nieuwe sterke generator h op niveau j)
17   y :=`false`;
18   **elseif** $h \neq 1$
19   **then** (h fixeert alle basispunten)
20   y :=`false`;
21   Zoek $\gamma \in \Omega$ met $\gamma^h \neq \gamma$;
22   Voeg γ toe aan B;
23             $k := k+1$; $S^{(k)} := []$;
24   **if not** y
25   **then for** $l \in [i+1 \ldots j]$
26   **do** Voeg h toe aan $S^{(l)}$;
27                 $H^{(l)} := \langle S^{(l)} \rangle$;
28                 $\Delta^{(l)} := \beta_l^{H^{(l)}}$;
29             $i := j$;
30   **continue** i;
          ($u_\beta x u_{\beta^x}^{-1} \in H^{(i+1)}$ gecontroleerd)
    (voorwaarde $H_{\beta_i^{(i)}} = H^{(i+1)}$ gecontroleerd)
31   $i := i-1$;
32 $S := \cup_{i=1}^{k} S^{(i)}$;

We doorlopen het volledige algoritme met het volgende voorbeeld. Stel $a = (1, 2, 4, 3)$ en $b = (1, 2, 5, 4)$, $\Omega = \{1, 2, 3, 4, 5\}$. We bepalen een BSGS voor de groep voortgebracht door a en b. We kiezen $B = \emptyset$ als beginwaarde. Het algoritme zal onmiddellijk $1 \in \Omega$ toevoegen. Na deze eerste initialisatie hebben we $B = [1]$, $k = 1$ en geen element van $S = [a, b]$ fixeert B. We vinden $\Delta^{(1)} = [1, 2, 4, 5, 3]$, met

$$U^{(1)} = [1, a, a^2 = (1, 4)(2, 3), ab = (1, 5, 4, 3, 2), a^3 = (1, 3, 4, 2)]$$

We starten nu met de hoofdlus op lijn 7, met $i = 1$. We doorlopen $\Delta^{(1)}$, $\beta = 1$, met $u_\beta = \mathbf{1}$ en $\beta^a = 2$, dan vinden we $u_\beta a \equiv u_{\beta^a}$, maar $\beta^b = 2$ en de Schreiergenerator is $u_\beta b u_{\beta^a}^{-1} = ba^{-1} = (2, 5)(3, 4)$. Omdat $k = 1$, zal Strip hierop $(2, 5)(3, 4)$ en 2 teruggeven, dus stellen we $c = (2, 5)(3, 4)$, voegen we 2 toe aan B (2 wordt niet gefixeerd door c), vergroten we k tot 2 en stellen we $S^{(2)} = [c]$. We berekenen $\Delta^{(2)} = [2, 5]$, met $U^{(2)} = [\mathbf{1}, c]$. (Merk op dat we c niet toevoegen aan $S^{(1)}$, omdat $H^{(1)}$ onveranderd blijft.)

Nu keren we terug naar lijn 7 met $i = 2$. We zien dat de twee Schreiergeneratoren cc^{-1} en c^2 triviaal zijn, de eerste per definitie, dus keren we terug naar lijn 6 met $i = 1$. De Schreiergenerator die voorheen een probleem veroorzaakte, levert als invoer voor Strip nu de indentieke, dus kunnen we overgaan naar $\beta = 2$, $u_\beta = a$. Hier zijn beide Schreiergeneratoren per definitie triviaal, dus kunnen we overgaan naar $\beta = 4$, $u_\beta = a^2$. We hebben $u_\beta a \equiv u_{\beta^a}$, maar $\beta^b = 1$, dus de geassocieerde Schreiergenerator is $u_\beta b = a^2 b = (2, 3, 5, 4)$. Omdat $3 = 2^{a^2 b} \notin \Delta^{(2)}$ zal Strip toegepast op $a^2 b$ juist $a^2 b$ en 2 teruggeven. Dus voegen we $d := a^2 b$ toe aan $S^{(2)}$. We hebben nu $S^{(2)} = [c, d]$ en we berekenen $\Delta^{(2)} = [2, 5, 3, 4]$ met $U^{(2)} = [1, c, d, cd]$.

Opnieuw gaan we naar lijn 7 met $i = 2$. De Schreiergeneratoren niet triviaal per definitie zijn c^2 (voor $\beta = 5$), $dc(cd)^{-1}$ en $(d^2 c^{-1})$ (voor $\beta = 3$), en $cdcd^{-1}$ en cd^2 (voor $\beta = 4$), en deze zijn allemaal triviaal, dus gaan we terug naar lijn 7 met $i = 1$ (voor de derde maal). De vorige maal kwamen we tot bij $\beta = 4$, en nu zal Strip toegepast op de Schreiergenerator $a^2 b$ de identieke teruggeven ($a^2 b d^{-1}$). We gaan verder met $\beta = 5$, $u_\beta = ab$ en we vinden dat de geassocieerde Schreiergenerators $aba(ab)^{-1} = (2, 3, 5, 4)$ en $ab^2 a^{-2} = (2, 3, 4, 5)$ (die als input voor Strip respectievelijk $aba(ab)^{-1} d^{-1} = \mathbf{1}$ en $\mathbf{1}$ teruggeven). Tenslotte vinden we voor $\beta = 4$, $u_\beta = a^3$ en als geassocieerde Schreiergenerators $a^4 = \mathbf{1}$ en $a^3 ba^{-3}$, die als argument voor Strip $a^3 ba^{-3}(cd)^{-1} = \mathbf{1}$. Dus het algoritme eindigt, en daarenboven hebben we aangetoond dat de groep orde $|\Delta^{(1)}||\Delta^{(2)}| = 20$ heeft.

5.4.3 Basisverandering

Veel algoritmen voor eindige permutatiegroepen vereisen een BSGS. Maar voor sommige algoritmen is het nuttig een vooraf gekozen basis mee te geven, of soms ligt een basis voor sommige groepen voor de hand door hun specifieke structuur. Bijvoorbeeld, stel dat we in een groep $G \leqslant \mathrm{Sym}(n)$ de stabilisatorgroep van $\alpha_1, \ldots, \alpha_r$ willen berekenen.

De beste manier om dit te doen is om een BSGS te bepalen met een basis B waarvoor $\{\alpha_1, \ldots, \alpha_r\} \subseteq B$, want de gezochte groep is dan eenvoudig $G^{(r)}$.

Het loont dus de moeite om te onderzoeken hoe we een basis, gevonden door SCHREIERSIMS, kunnen veranderen. Een dergelijke basisverandering zul aanleiding geven tot een nieuwe verzameling sterk voorbrengende generatoren. De fundamentele stap om een basis te wijzigen is het omwisselen van twee elementen β_i en β_{i+1}, en werd ontworpen door C. Sims[2].

Algoritme 5.18 basisverandering

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

Invoer: BSGS (B, S) voor een groep $G \leqslant \mathrm{Sym}(n)$, $i \in [1 \ldots |B| - 1]$.

1 $s := |\Delta^{(i)}||\Delta^{(i+1)}|/|\beta_{i+1}^{G^{(i)}}|$;
(* s is the grootte van de nieuwe $\Delta^{(i+1)}$ *)
2 $T := S^{(i+2)}$; $\Gamma := \Delta^{(i)} \setminus \{\beta_i, \beta_{i+1}\}$;
3 **while** $|\beta_i^{\langle T \rangle}| \neq s$
4   **do** Bepaal $\gamma \in \Gamma$, $x \in U^{(i)}$ zodat $\gamma = \beta_i^x$;
5   **if** $\beta_{i+1}^{x^{-1}} \notin \Delta^{(i+1)}$
6   **then** $\Gamma := \Gamma \setminus \gamma^{\langle T \rangle}$;
7   **else** Bepaal $y \in G^{(i+1)}$ zodat $\beta_{i+1}^y = \beta_{i+1}^{x^{-1}}$;
8   **if** $\beta^{yx} \notin \beta_i^{\langle T \rangle}$
9   **then** $T := T \cup \{yx\}$; $\Gamma := \Gamma \setminus \beta_i^{\langle T \rangle}$;
10 $S := S \cup T$;
11 verwissel β_i en β_{i+1};
12 verwijder redundante generatoren uit S;
13 herbereken $\Delta^{(i)}$, $\Delta^{(i+1)}$ en Schreiervectoren

Om de correctheid van het algoritme te bewijzen, moeten we aantonen dat de nieuwe verzameling S die op het einde van de procedure onstaat, een verzameling sterk voortbrengende generatoren is ten opzichte van de nieuwe basis. Door β_i en β_{i+1} om te wisselen, blijven alle $G^{(k)}$, $k = 1, \ldots, i, i+2, r$, met r de lengte van B, gelijk. Enkel $G^{(i+1)}$ moet veranderd worden in $H := G^{(i)}_{\beta_{i+1}}$[3]. Het is dus voldoende om aan te tonen dat de verzameling T, die toegevoegd wordt aan S in lijn 10, de groep H voortbrengt.

[2]Charles C. Sims. *Computation with permutation groups.* in *Proc. Second Symp. on Symbolic and Algebraic Manipulation.* ACM Press, 1971. en Charles C. Sims. *Determining the conjugacy classes of permutation groups.* in *Computers in Algebra and Number Theory*, Garret Birkhoff and Marshall Hall Jr., ed., volume 4 of *Proc. Amer. Math. Soc.*, pages 191 – 195, Providence, RI, 1971. (New York, 1970), Amer. Math. Soc.

[3]Met $G^{(i)}$ bedoelen we hier wel degelijk de $G^{(i)}$ uit de originele ketting.

Initieel is $T = S^{(i+2)}$ en de enige elementen die eraan toegevoegd worden zijn van de vorm yx, met $y, x \in G^{(i)}$ en yx fixeert β_{i+1}, dus het is duidelijk dat $T \subseteq H$.

Gebruikmakend van de orbit-stabilizer stelling 2.1.6 vinden we

$$|G^{(i)}| = |\Delta^{(i)}||G^{(i+1)}| = |\Delta^{(i)}||\Delta^{(i+1)}||G^{(i+2)}| = |\beta_{i+1}^{G^{(i)}}||H|$$

Daaruit volgt $|H| = s|G^{(i+2)}|$, met s gedefinieerd in lijn 1 van de procedure en $s = |\beta_i^H|$.

Omdat $\langle T \rangle \leq H$, en omdat we de hoofd while-lus (lijn 3) niet verlaten totdat $|\beta_{i+1}^{\langle T \rangle}| = s$, zou moeten gelden dat $\langle T \rangle = H$ bij het verlaten van de while-lus, dus de vraag is of de elementen die tijdens de while-lus aan T toegevoegd worden, voldoende zijn om H voort te brengen.

Een willekeurig element van $G^{(i)}$ kan geschreven worden als yx voor een $y \in G^{(i+1)}$ (we bedoelen hierbij de originele $G^{(i+1)} = G_{\beta_i}^{(i)}$ uit de ketting), en $x \in U^{(i)}$. Voor een gegeven $x \in U^{(i)}$ bestaat er een dergelijk element $yx \in H$ als en slechts als er een element $y \in G^{(i+1)}$ bestaat waarvoor $\beta_{i+1}^{yx} = \beta_{i+1}$, hetgeen het geval is als en slechts als $\beta_{i+1}^{x^{-1}} \in \beta_{i+1}^{G^{(i+1)}} = \Delta^{(i+1)}$. Dit rechtvaardigt de if-lus op lijn 5.

Stel dat y_1x en y_2x twee elementen van H zijn van deze vorm. Dan geldt $(y_1x)(y_2x)^{-1} = y_1y_2 \in G^{(i+2)} \leq \langle T \rangle$, dus, voor een vaste x, moet er maar één dergelijk element aan T toegevoegd worden. Dit is juist wat er gebeurt in de procedure. Daarenboven moet er geen element yx toegevoegd worden waarvoor $\beta_i yx = \beta_i^t$ voor een bepaalde t die behoort tot $\langle T \rangle$; dit rechtvaardigt het verwijderen van de elementen $\beta_i^{\langle T \rangle}$ uit Γ op lijn 9. Anderzijds, als er geen element $yx \in H$ voor de gegeven x bestaat, dan kan er ook geen element van de vorm yxt zijn voor elke $t \in \langle T \rangle$. Dit rechtvaardigt het verwijderen van $\gamma^{\langle T \rangle}$ uit Γ in lijn 6.

In een variant op dit algoritme, beschreven door Á. Seress[4], worden random elementen van H berekend door een Schreiervector v voor $\beta_{i+1}^{G^{(i)}}$ te berekenen en dan RandomStab($\beta_{i+1}, v, S^{(i)}$) herhaaldelijk toe te passen. We kunnen dezelfde stop conditie $|\beta_{i+1}^{\langle T \rangle}| = s$ gebruiken.

Tenslotte beschrijven we kort enkele mogelijkheden om andere basisveranderingen door te voeren. Een eenvoudige methode, opgemerkt door C. Sims, bestaat erin om toe te voegen met een zekere $g \in G$, waardoor de basis $[\beta_1, \ldots, \beta_k]$ vervangen wordt door $[\beta_1^g, \ldots, \beta_k^g]$. Om willekeurige basisveranderingen uit te voeren kunnen we als volgt te werk gaan. Veronderstel dat we β_1 willen vervangen door α. Als $\alpha \in \Delta^{(1)}$, dan geldt $\beta_1^g = \alpha$, voor een $g \in G$, dus kunnen we toevoegen met g^{-1}. Anders veronderstellen we, als inductiehypothese, dat we β_2 kunnen vervangen door α (of als $k = 1$, voegen we gewoon α toe), we introduceren α als nieuw redundant basispunt β_2, waarna we BaseSwap($*B, *S, 1$) uitvoeren. Nadien beschikken we over een basis waarin sommige basispunten overbodig zijn. Deze kunnen we met behulp van BaseSwap verplaatsen naar het einde van de basis, waar ze tenslotte pijnloos verwijderd kunnen worden.

[4]Ákos Seress. *Permutation Group Algorithms*. Cambridge Tracts in Mathematics 152. Cambridge University Press, 2003.

Figuur 5.1: Rubik's $3 \times 3 \times 3$ kubus, in sleutelhangerformaat

Tenslotte beschikken we over een BSGS met de gewenste basis. Het is echter niet ondenkbaar dat de verzameling S redundante generatoren bevat. Het volgende algoritme kan deze verwijderen.

Algoritme 5.19 verwijderen van redundante generatoren

RemoveGens(B,$*S$,Δ^*)

Invoer: BSGS (B, S) voor een groep.

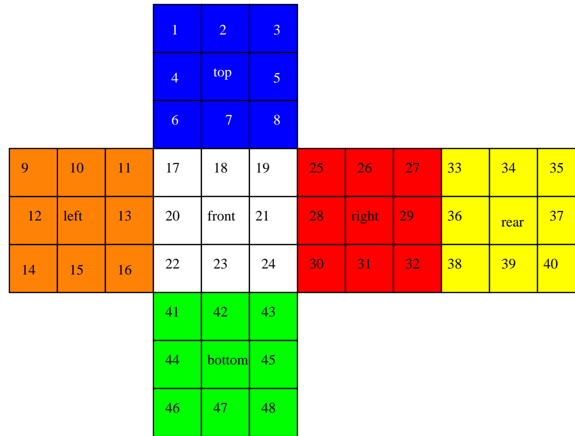
1 **for** i **from** k **downto** 1
2   **do for** $g \in S^{(i)} \backslash S^{(i+1)}$
3   **do if** $\beta_i^{\langle S^{(i)} \backslash \{g\} \rangle} = \Delta^{(i)}$
4   **then** verwijder g uit S

5.5 Rubik's kubus en GAP

De $3 \times 3 \times 3$ kubus (Figuur 5.1) telt 54 vlakken. Elke rotatie van een vlak beschouwen we als een generator. Een dergelijke rotatie houdt de middelste vlakjes van een groot vlak op zijn plaats. We nummeren de resterende vlakjes als volgt (Figuur 5.2) en definiëren de groep van de kubus. (`gap5.5.1.i` en `gap5.5.1`).

```
   Loading the library. Please be patient, this may take a while.
GAP4, Version: 4.4.6 of 02-Sep-2005, powerpc-apple-darwin8.2.0-gcc
Components:  small 2.1, small2 2.0, small3 2.0, small4 1.0, small5 1.0,
             small6 1.0, small7 1.0, small8 1.0, small9 1.0, small10 0.2,
             id2 3.0, id3 2.1, id4 1.0, id5 1.0, id6 1.0, id9 1.0, id10 0.1,
             trans 1.0, prim 2.1  loaded.
Packages:    TomLib 1.1.2  loaded.
gap> t:=(1,3,8,6)(2,5,7,4)(9,33,25,17)(10,34,26,18)(11,35,27,19);
(1,3,8,6)(2,5,7,4)(9,33,25,17)(10,34,26,18)(11,35,27,19)
```

Figuur 5.2: Rubik's $3 \times 3 \times 3$ kubus, vlakjes genummerd




```
gap> l := ( 9,11,16,14)(10,13,15,12)( 1,17,41,40)( 4,20,44,37)( 6,22,46,35);
(1,17,41,40)(4,20,44,37)(6,22,46,35)(9,11,16,14)(10,13,15,12)
gap> f := (17,19,24,22)(18,21,23,20)( 6,25,43,16)( 7,28,42,13)( 8,30,41,11);
(6,25,43,16)(7,28,42,13)(8,30,41,11)(17,19,24,22)(18,21,23,20)
gap> r := (25,27,32,30)(26,29,31,28)( 3,38,43,19)( 5,36,45,21)( 8,33,48,24);
(3,38,43,19)(5,36,45,21)(8,33,48,24)(25,27,32,30)(26,29,31,28)
gap> re := (33,35,40,38)(34,37,39,36)( 3, 9,46,32)( 2,12,47,29)( 1,14,48,27);
(1,14,48,27)(2,12,47,29)(3,9,46,32)(33,35,40,38)(34,37,39,36)
gap> b := (41,43,48,46)(42,45,47,44)(14,22,30,38)(15,23,31,39)(16,24,32,40);
(14,22,30,38)(15,23,31,39)(16,24,32,40)(41,43,48,46)(42,45,47,44)
gap> cube := Group([t,l,f,r,re,b]);
<permutation group with 6 generators>
gap> Size(cube);
43252003274489856000
```

Om te beginnen hebben we de orde van de groep bepaald. We onderzoeken de werking van de groep op de 48 vlakjes. Het is duidelijk de een vlakje dat behoort tot een van de hoekblokjes (een blokje dat drie vlakken bevat), niet kan verplaatst worden naar een vlakje dat behoort tot de zijblokjes (een blokje dat twee vlakjes bevat). Dit betekent dat er minstens twee banen moeten zijn.

```
gap> orbits := Orbits(cube,[1..48]);
[ [ 1, 3, 17, 14, 8, 38, 9, 41, 19, 48, 22, 6, 30, 33, 43, 11, 46, 40, 24,
      27, 25, 35, 16, 32 ],
  [ 2, 5, 12, 7, 36, 10, 47, 4, 28, 45, 34, 13, 29, 44, 20, 42, 26, 21, 37,
      15, 31, 18, 23, 39 ] ]
gap> cube1 := Action(cube,orbits[1],OnPoints);
<permutation group with 6 generators>
gap> cube2 := Action(cube,orbits[2],OnPoints);
<permutation group with 6 generators>
gap> NrMovedPoints(cube1);
24
gap> NrMovedPoints(cube2);
24
```

We vinden juist twee banen. We onderzoeken de groep verder door de actie van de groep op elke baan te onderzoeken. De actie op elke baan zal zeker transitief zijn. We weten dat ze ook getrouw zal zijn, en we onderzoeken of ze primitief is.

```
gap> blocks1 := Blocks(cube1,MovedPoints(cube1));
[ [ 1, 7, 22 ], [ 2, 14, 20 ], [ 3, 12, 16 ], [ 4, 17, 18 ], [ 5, 9, 21 ],
  [ 6, 10, 24 ], [ 8, 11, 23 ], [ 13, 15, 19 ] ]
gap> blocks2 := Blocks(cube2,MovedPoints(cube2));
[ [ 1, 11 ], [ 2, 17 ], [ 3, 19 ], [ 4, 22 ], [ 5, 13 ], [ 6, 8 ], [ 7, 24 ],
  [ 9, 18 ], [ 10, 21 ], [ 12, 15 ], [ 14, 20 ], [ 16, 23 ] ]
```

Beide groepen werken imprimitief. Nu bepalen we we actie van de beide groepen op hun respectievelijke blocksystemen.

```
gap> hom1 := ActionHomomorphism(cube1,blocks1,OnSets);
<action homomorphism>
gap> cube1b := Image(hom1);
Group([ (1,2,4,3), (1,3,6,5), (1,5,8,2), (3,4,7,6), (5,6,7,8), (2,8,7,4) ])
gap> Size(cube1b);
40320
gap> hom2 := ActionHomomorphism(cube2,blocks2,OnSets);
<action homomorphism>
gap> cube2b := Image(hom2);
Group([ (1,3,4,2), (1,5,11,10), (2,6,7,5), (3,10,12,8), (4,8,9,6),
  (7,9,12,11) ])
gap> Size(cube2b);
479001600
```

Voor de groep `cube1` vinden we als orde 8!. Een permutatiegroep die werkt op 8 elementen en die orde 8! heeft, kan niets anders zijn dan S_8. Voor `cube2` vinden we orde 12!, en deze groep werkt op 12 elementen, dus we besluiten ook hier onmiddellijk dat deze groep S_{12} is. Nu onderzoeken we de kern van beide acties. Elk element uit de kern van actie `hom1`, respectievelijk `hom2`, houdt alle hoekblokjes, respectievelijk zijblokjes, op hun plaats, en heeft dus orde 3, respectievelijk orde 2.

```
gap> cube1c := Kernel(hom1);
<permutation group with 7 generators>
gap> cube2c := Kernel(hom2);
<permutation group with 11 generators>
gap> Size(cube1c);
2187
gap> Factors(Size(cube1c));
[ 3, 3, 3, 3, 3, 3, 3 ]
gap> IsElementaryAbelian(cube1c);
true
gap> Size(cube2c);
2048
gap> IsElementaryAbelian(cube2c);
true
```

De orde van `cube1c`, de kern van de eerste actie blijkt 3^7 te zijn. Aangezien elk element orde 3 heeft, is het niet ondenkbaar dat deze groep gelijk is aan C_3^7. Aangezien blijkt dat de groep elementair abels is, en dus gelijk aan het direct produkt van cyclische groepen, besluiten we onmiddellijk dat de groep gelijk is aan C_3^7. Analoog vinden we dat de groep `cube2c` gelijk is aan C_2^{11}. Merk op dat er 8 hoekblokjes zijn en 12 zijblokjes. Indien de groep van de kubus toeliet dat de vlakjes van een willekeurig hoekblokje, respectievelijk zijblokje, gepermuteerd werden, en dat daarbij alle andere hoekblokjes, respectievelijk zijblokjes, invariant bleven (en daarmee bedoelen we dat alle vlakjes op hun plaats blijven), dan zou de groep `cube1c`, respectievelijk `cube2c`, gelijk zijn aan C_3^8, respectievelijk C_2^{12}. Stel dat we drie vlakjes van een hoekblokje permuteren, dan moeten nog drie vlakjes van een ander hoekblokje meepermuteren, en dat zelfs in de juiste richting. Voor de vlakjes op de zijblokjes geldt min of meer hetzelfde principe.

```
gap> cube1c := Kernel(hom1);
<permutation group with 7 generators>
gap> cube2c := Kernel(hom2);
<permutation group with 11 generators>
gap> Size(cube1c);
2187
gap> Factors(Size(cube1c));
[ 3, 3, 3, 3, 3, 3, 3 ]
gap> IsElementaryAbelian(cube1c);
true
gap> Size(cube2c);
2048
gap> IsElementaryAbelian(cube2c);
true
gap> (1,7,22) in cube1c;
false
gap> (1,7,22)(2,14,20) in cube1c;
false
gap> (1,7,22)(2,20,14) in cube1c;
true
gap> (1,7,22)(3,12,16) in cube1c;
true
gap> (1,7,22)(3,12,16)(2,14,20) in cube1c;
false
gap> (1,7,22)(3,16,12)(2,14,20) in cube1c;
true
gap> (1,11) in cube2c;
false
gap> (1,11)(2,17) in cube2c;
true
gap> (1,11)(2,17)(3,19) in cube2c;
false
gap> (1,11)(2,17)(3,19)(4,22) in cube2c;
true
```

Nu onderzoeken we het verband tussen de groep `cube1` en de kern en het beeld van `hom1`, en tussen `cube2` en de kern en het beeld van `hom2`. Stel dat G en M twee groepen zijn, en dat er een homomorfisme bestaat $\varphi : G \to \mathrm{Aut}(M)$. De bewerking in het semidirect produkt $G \ltimes_\varphi M$ werd gedefinieerd als $(g,m)(h,n) := (gh, m^{\varphi(h)}n)$. We zullen dit concept in een ruimere context plaatsen.

Stel opnieuw dat G een groep is, en $N \trianglelefteq G$. We zeggen dat G een *groep extensie* (**E:** *group extension*) is van N door G/N. G wordt een *gespleten extensie* (**E:** *split extension*) van N door G/N genoemd als en slechts als er een groep $C \leqslant G$ bestaat waarvoor $NC = G$ (dus $G = \langle N, C\rangle$) en $N \cap C = \{\mathbf{1}\}$; we noemen C een *complement* (**E:** *complement*) van N in G.

Beschouw nu een semidirect produkt $G \ltimes_\varphi M$. We definiëren nu een homomorfisme (en het zal een endomorfisme zijn) $\theta_1 : M \to G \ltimes_\varphi M$, $m \mapsto (\mathbf{1}_G, m)$. Het is duidelijk dat θ_1 injectief is. We definiëren een homomorfisme (en het zal een epimorfisme zijn) $\theta_2 : G \ltimes_\varphi M \to G$, $(g,m) \mapsto g$. We beschouwen nu een element $\theta(m)$ en een willekeurig element $(g,n) \in G \ltimes_\varphi M$. We berekenen eerst

$$(g,n) \cdot (g^{-1}, (n^{-1})^{\varphi(g^{-1})}) = (\mathbf{1}, n^{\varphi(g^{-1})}(n^{-1})^{\varphi(g^{-1})}) = (\mathbf{1}, \mathbf{1}).$$

Dus de inverse van (g,n) is $(g^{-1}, (n^{-1})^{\varphi(g^{-1})})$. Dan berekenen we

$$(\mathbf{1}, m)^{(g,n)} = (g^{-1}, (n^{-1})^{\varphi(g^{-1})})(\mathbf{1}, m)(g,n) = (g^{-1}, ((n^{-1})^{\varphi(g^{-1})})^{\varphi(\mathbf{1})}m)(g,n)$$

$$= (gg^{-1}, n^{-1}m^{\varphi(g)}n) = (\mathbf{1}, n^{-1}m^{\varphi(g)}n) \in \mathrm{im}(\theta_1).$$

Dus $\text{im}(\theta_1) \trianglelefteq G \ltimes_\varphi M$. Dus deze groep is een extensie van $\text{im}(\theta_1)$ door $G \ltimes_\varphi M/\,\text{im}(\theta_1)$. Het is duidelijk dat $\ker(\theta_2) \cong M$ en $G \ltimes_\varphi M/\ker(\theta_2) \cong G$. De homomorfisme θ_1 en θ_2 stilzwijgend in beschouwing nemend, kunnen we dus zeggen dat $G \ltimes_\varphi M$ een extensie is van M door G.

Stel nu omgekeerd dat $M \trianglelefteq E$ en dat er een complement G van M in E bestaat. Elk element $e \in E$ kan dan op een unieke wijze geschreven worden als $e = gm$, $g \in G$ en $m \in M$. Daarenboven is $gmhn = ghm^h n$. Dus φ beeld elk element $h \in G$ af op de toevoeging door h. We besluiten dat de gespleten extensie E van M door G, isomorf is met het semidirect produkt $G \ltimes_\varphi M$. Dit stelt ons nu in staat om de groep van de kubus verder te onderzoeken. Noteren we `cube1` als E, en de kern van het homomorfisme `hom1` als M. Indien we een complement G van M in E vinden dan besluiten we onmiddellijk dat $E = G \ltimes M$. In de onderstaande sessie zoeken we een complement, we verifiëren de complement eigenschap, en we gaan na dat het complement wel degelijk isomorf is met het beeld van `hom1`.

```
gap> cmpl1 := Complementclasses(cube1,Kernel(hom1));
[ Group([ (1,3,4,2)(7,16,17,14)(12,18,20,22),
      (1,2,3,4,5,6,13)(7,14,16,17,21,10,15)(9,24,19,22,20,12,18),
      (1,2,3,4,5,8,13)(7,14,16,17,21,23,15)(9,11,19,22,20,12,18) ]) ]
gap> cmpl := last[1];
Group([ (1,3,4,2)(7,16,17,14)(12,18,20,22), (1,2,3,4,5,6,13)(7,14,16,17,21,10,
    15)(9,24,19,22,20,12,18), (1,2,3,4,5,8,13)(7,14,16,17,21,23,15)(9,11,19,
    22,20,12,18) ])
gap> Size(Intersection(cmpl,Kernel(hom1)));
1
gap> ClosureGroup(cmpl,Kernel(hom1))=cube1;
true
gap> IsBijective(RestrictedMapping(hom1,cmpl));
true
```

We besluiten dat `cube1` isomorf is met het semidirect produkt $E_1 := S_8 \ltimes C_3^7$. Het is meteen duidelijk dat E_1 een groep van index 3 is in het wreath produkt $C_3 \wr S_8$. Het feit dat de index 3 is, komt overeen met het feit dat we geen drie vlakjes van een hoekblokje kunnen permuteren, zonder alle vlakjes van alle andere hoekblokjes invariant te laten.

We onderzoeken `cube2` op een volkomen analoge manier. Uit de onderstaande sessie concluderen we dat `cube2` isomorf is met $E_2 := S_{12} \ltimes C_2^{11}$. Deze groep is een groep van index 2 in het wreath produkt $C_2 \wr S_{12}$.

```
gap> cmpl2 := Complementclasses(cube2,Kernel(hom2));
[ Group([ (1,2,4,3)(11,17,22,19), (1,2,3,4,5,6,7,9,10,12,16)(8,24,18,21,15,23,
        11,17,19,22,13), (1,2,3,4,5,6,7,9,10,14,16)(8,24,18,21,20,23,11,17,19,
        22,13) ]),
  Group([ (1,17,4,19)(2,22,3,11)(5,13)(6,8)(7,24)(9,18)(10,21)(12,15)(14,
        20)(16,23), (1,2,3,4,5,6,7,9,10,12,16)(8,24,18,21,15,23,11,17,19,22,
        13), (1,2,3,4,5,6,7,9,10,14,16)(8,24,18,21,20,23,11,17,19,22,13) ]),
  Group([ (1,2,4,3)(11,17,22,19), (1,2,3,4,5,6,7,9,10,12,16)(8,24,18,21,15,23,
        11,17,19,22,13), (1,2,3,4,5,6,7,9,10,20,16)(8,24,18,21,14,23,11,17,19,
        22,13) ]),
  Group([ (1,17,4,19)(2,22,3,11)(5,13)(6,8)(7,24)(9,18)(10,21)(12,15)(14,
        20)(16,23), (1,2,3,4,5,6,7,9,10,12,16)(8,24,18,21,15,23,11,17,19,22,
        13), (1,2,3,4,5,6,7,9,10,20,16)(8,24,18,21,14,23,11,17,19,22,13) ]) ]
gap> cmpl := cmpl2[1];
Group([ (1,2,4,3)(11,17,22,19), (1,2,3,4,5,6,7,9,10,12,16)(8,24,18,21,15,23,
    11,17,19,22,13), (1,2,3,4,5,6,7,9,10,14,16)(8,24,18,21,20,23,11,17,19,22,
    13) ])
gap> Size(Intersection(cmpl,Kernel(hom2)));
1
gap> ClosureGroup(cmpl,Kernel(hom2))=cube2;
true
```

```
gap> IsBijective(RestrictedMapping(hom2,cmpl));
true
```

Tenslotte onderzoeken we het verband tussen E_1, E_2 en de groep `cube` zelf. E_1 en E_2 zijn in de respectievelijke acties van `cube` op de twee banen. Daaruit volgt onmiddellijk dat `cube` een deelgroep moet zijn van het direct produkt van E_1 en E_2. Een eenvoudige berekening toont aan dat `cube` een deelgroep van index 2 is in dit direct produkt.

```
gap> (Size(cube1)*Size(cube2))/Size(cube);
2
```

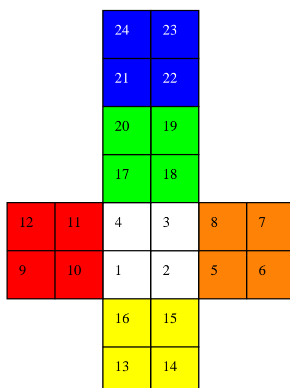
Tot nu toe heeft dit voorbeeld het gebruik van homomorfismen goed geïllustreerd. We berekenen nu een BSGS voor de groep `cube` (`gap5.5.2.i`).

```
   Loading the library. Please be patient, this may take a while.
GAP4, Version: 4.4.6 of 02-Sep-2005, powerpc-apple-darwin8.2.0-gcc
Components:  small 2.1, small2 2.0, small3 2.0, small4 1.0, small5 1.0,
             small6 1.0, small7 1.0, small8 1.0, small9 1.0, small10 0.2,
             id2 3.0, id3 2.1, id4 1.0, id5 1.0, id6 1.0, id9 1.0, id10 0.1,
             trans 1.0, prim 2.1  loaded.
Packages:    TomLib 1.1.2  loaded.
gap> t:=(1,3,8,6)(2,5,7,4)(9,33,25,17)(10,34,26,18)(11,35,27,19);
(1,3,8,6)(2,5,7,4)(9,33,25,17)(10,34,26,18)(11,35,27,19)
gap> l := ( 9,11,16,14)(10,13,15,12)( 1,17,41,40)( 4,20,44,37)( 6,22,46,35);
(1,17,41,40)(4,20,44,37)(6,22,46,35)(9,11,16,14)(10,13,15,12)
gap> f := (17,19,24,22)(18,21,23,20)( 6,25,43,16)( 7,28,42,13)( 8,30,41,11);
(6,25,43,16)(7,28,42,13)(8,30,41,11)(17,19,24,22)(18,21,23,20)
gap> r := (25,27,32,30)(26,29,31,28)( 3,38,43,19)( 5,36,45,21)( 8,33,48,24);
(3,38,43,19)(5,36,45,21)(8,33,48,24)(25,27,32,30)(26,29,31,28)
gap> re := (33,35,40,38)(34,37,39,36)( 3, 9,46,32)( 2,12,47,29)( 1,14,48,27);
(1,14,48,27)(2,12,47,29)(3,9,46,32)(33,35,40,38)(34,37,39,36)
gap> b := (41,43,48,46)(42,45,47,44)(14,22,30,38)(15,23,31,39)(16,24,32,40);
(14,22,30,38)(15,23,31,39)(16,24,32,40)(41,43,48,46)(42,45,47,44)
gap> cube := Group([t,l,f,r,re,b]);
<permutation group with 6 generators>
gap> S := StabChain(cube);
<stabilizer chain record, Base [ 1, 2, 3, 4, 5, 6, 7, 8, 12, 13, 14, 15, 16,
  21, 23, 24, 29, 31 ], Orbit length 24, Size: 43252003274489856000>
gap> B := BaseStabChain(S);
[ 1, 2, 3, 4, 5, 6, 7, 8, 12, 13, 14, 15, 16, 21, 23, 24, 29, 31 ]
gap> Length(B);
18
gap> sg := StrongGeneratorsStabChain(S);
[ (31,45)(39,47), (29,36)(31,45), (29,47,31)(36,39,45), (24,30,43)(32,48,38),
  (24,38)(30,48)(31,39)(32,43)(45,47), (23,31,29)(36,42,45),
  (23,45,36)(29,42,31), (21,23,36)(28,42,29), (16,41,22)(24,30,43),
  (16,48,22,32,41,38)(23,45,29,47)(24,43,30)(31,36,39,42),
  (15,45,36)(29,44,31), (14,22,30,38)(15,23,31,39)(16,24,32,40)(41,43,48,
    46)(42,45,47,44), (13,28,29)(20,21,36)(24,43,30)(32,38,48),
  (13,42)(20,23)(21,36)(24,30,43)(28,29)(32,48,38), (12,31,29)(36,37,45),
  (8,30,19,24,25,43)(16,38,22,48,41,32), (8,41,24,32)(13,28,42,29)(16,30,38,
    25)(19,22,43,48)(20,21,23,36), (7,31,42,28,13)(8,30,19,24,25,43)(16,38,22,
    48,41,32)(18,45,23,21,20), (7,42,28,29,13)(8,30,19,24,25,43)(16,38,22,48,
    41,32)(18,23,21,36,20), (6,25,43,16)(7,28,42,13)(8,30,41,11)(17,19,24,
    22)(18,21,23,20), (5,21,36,23)(8,48,43,22)(16,19,38,30)(24,41,25,32)(26,
    28,29,42), (5,45,20,23)(8,48,43,22)(13,42,26,31)(16,19,38,30)(24,41,25,32)
    , (4,45,29,42,13,10,31,36,23,20)(14,30,40,24,46,43)(15,39,44,47)(16,22,41)
    , (3,38,43,19)(5,36,45,21)(8,33,48,24)(25,27,32,30)(26,29,31,28),
  (2,29)(31,39)(34,36)(45,47), (1,3,8,6)(2,5,7,4)(9,33,25,17)(10,34,26,18)(11,
    35,27,19), (1,14,48,27)(2,12,47,29)(3,9,46,32)(33,35,40,38)(34,37,39,36),
  (1,17,41,40)(4,20,44,37)(6,22,46,35)(9,11,16,14)(10,13,15,12) ]
gap> Length(sg);
28
```

We beschouwen nu de $2\times2\times2$ kubus om opnieuw mogelijkheden van homomorfismen te illustreren. Deze voorbeelden zijn ook toepasbaar op de $3 \times 3 \times 3$ kubus.

De $2\times2\times2$ kubus kan men zien als de $3\times3\times3$ kubus waarbij de zijvlakjes verwijderd zijn. Men kan elk vlak draaien, en deze rotaties vormen weer de generatoren van de

Figuur 5.3: Rubik's $2 \times 2 \times 2$ kubus, vlakjes genummerd



groep. Uit de analyse van de $3 \times 3 \times 3$ kubus volgt dan onmiddellijk dat deze groep $S_8 \ltimes C_3^7$ is. Geen enkel vlakje wordt vastgehouden door alle generatoren, dit in tegenstelling tot de $3 \times 3 \times 3$ kubus. Voor dit voorbeeld zijn we niet alleen geïnteresseerd in de analyse van de groep, maar ook in de berekening van een oplossing van de kubus. We nummeren de vlakjes van de kubus (Figuur 5.3). De generatoren die vlakje 2 invariant laten (en bijgevolg ook vlakjes 5 en 16) noteren we met b (bovenvlak, tegenwijzerzin), l (linkervlak, tegenwijzerzin) en a (achtervlak, tegenwijzerzin). Wanneer we een vlak in tegenwijzerzin draaien, dan kijken we recht op dat vlak. Indien de kubus zich in een willekeurige stand bevindt, dan kunnen we steeds de drie vlakjes 2,5 en 16 zo richten dat zij zich rechts onderaan bevinden. Zo zijn bovenvlak, linkervlak en achtervlak eenduidig bepaald. De onderstaande sessie heeft als besluit dat de groep `cube` isomorf is met $C_3^6 \ltimes S_7$. (`gap5.5.3.i` en `gap5.5.3`). We bepalen eveneens een BSGS.

```
   Loading the library. Please be patient, this may take a while.
GAP4, Version: 4.4.6 of 02-Sep-2005, powerpc-apple-darwin8.2.0-gcc
Components:  small 2.1, small2 2.0, small3 2.0, small4 1.0, small5 1.0,
             small6 1.0, small7 1.0, small8 1.0, small9 1.0, small10 0.2,
             id2 3.0, id3 2.1, id4 1.0, id5 1.0, id6 1.0, id9 1.0, id10 0.1,
             trans 1.0, prim 2.1  loaded.
Packages:    TomLib 1.1.2  loaded.
gap> b := (17,18,19,20)(4,8,22,11)(3,7,21,12);
(3,7,21,12)(4,8,22,11)(17,18,19,20)
gap> l := (9,10,12,11)(13,1,17,21)(15,4,20,24);
(1,17,21,13)(4,20,24,15)(9,10,12,11)
gap> a := (21,22,23,24)(7,14,9,20)(6,13,11,19);
(6,13,11,19)(7,14,9,20)(21,22,23,24)
gap> cube := Group([b,l,a]);
Group([ (3,7,21,12)(4,8,22,11)(17,18,19,20), (1,17,21,13)(4,20,24,15)(9,10,12,
    11), (6,13,11,19)(7,14,9,20)(21,22,23,24) ])
gap> Size(cube);
3674160
gap> orbits := Orbits(cube,[1..24]);
[ [ 1, 17, 18, 21, 19, 12, 13, 22, 20, 6, 3, 11, 23, 24, 7, 4, 9, 15, 14, 8,
      10 ], [ 2 ], [ 5 ], [ 16 ] ]
gap> cube1 := Action(cube,orbits[1]);
Group([ (1,11,13,14)(3,19,17,18)(7,21,20,10),
  (4,14,10,8)(5,12,7,19)(9,17,13,15), (2,5,13,20)(3,6,15,10)(8,19,11,16) ])
gap> blocks := Blocks(cube1,[1..21]);
[ [ 1, 18, 21 ], [ 2, 6, 16 ], [ 3, 11, 20 ], [ 4, 9, 12 ], [ 5, 8, 15 ],
  [ 7, 14, 17 ], [ 10, 13, 19 ] ]
gap> hom := ActionHomomorphism(cube1,blocks,OnSets);
```

```
<action homomorphism>
gap> cube1a := Image(hom);
Group([ (1,3,7,6), (4,6,7,5), (2,5,7,3) ])
gap> Size(cube1a);
5040
gap> Factorial(7);
5040
gap> cube1b := Kernel(hom);
<permutation group with 6 generators>
gap> Size(cube1b);
729
gap> 3^6;
729
gap> IsElementaryAbelian(cube1b);
true
gap> cmpl1 := Complementclasses(cube1,Kernel(hom));
[ Group([ (1,7,10,3)(11,21,17,19)(13,20,18,14),
      (1,2,3,4,5,7,10)(6,11,12,15,17,19,21)(8,14,13,18,16,20,9),
      (1,3,2,4,5,7,10)(6,12,15,17,19,21,11)(8,14,13,18,20,16,9) ]) ]
gap> cmpl := cmpl1[1];
Group([ (1,7,10,3)(11,21,17,19)(13,20,18,14),
  (1,2,3,4,5,7,10)(6,11,12,15,17,19,21)(8,14,13,18,16,20,9),
  (1,3,2,4,5,7,10)(6,12,15,17,19,21,11)(8,14,13,18,20,16,9) ])
gap> ClosureGroup(cmpl,Kernel(hom))=cube1;
true
gap> IsBijective(RestrictedMapping(hom,cmpl));
true
gap> S := StabChain(cube);
<stabilizer chain record, Base [ 1, 3, 4, 6, 7, 9 ], Orbit length 21, Size:
3674160>
```

Een basisbeeld bepaalt een uniek element g. Indien we de inverse van g uitvoeren, dan lossen we de puzzle op. Het element g kunnen we bepalen met behulp van ElementBaseImage (Algoritme 5.14, implementatie in GAP: `gap5.4.1`). Het tweede probleem is het vinden van een woord $w_g(b, l, a)$, waarmee we de stappen handmatig kunnen uitvoeren. Met behulp van een homomorfisme tussen `cube` en een vrije groep in drie generatoren kunnen we dergelijke woorden berekenen.

```
   Loading the library. Please be patient, this may take a while.
GAP4, Version: 4.4.6 of 02-Sep-2005, powerpc-apple-darwin8.2.0-gcc
Components:  small 2.1, small2 2.0, small3 2.0, small4 1.0, small5 1.0,
             small6 1.0, small7 1.0, small8 1.0, small9 1.0, small10 0.2,
             id2 3.0, id3 2.1, id4 1.0, id5 1.0, id6 1.0, id9 1.0, id10 0.1,
             trans 1.0, prim 2.1  loaded.
Packages:    TomLib 1.1.2  loaded.
gap> b := (17,18,19,20)(4,8,22,11)(3,7,21,12);
(3,7,21,12)(4,8,22,11)(17,18,19,20)
gap> l := (9,10,12,11)(13,1,17,21)(15,4,20,24);
(1,17,21,13)(4,20,24,15)(9,10,12,11)
gap> a := (21,22,23,24)(7,14,9,20)(6,13,11,19);
(6,13,11,19)(7,14,9,20)(21,22,23,24)
gap> cube := Group([b,l,a]);
Group([ (3,7,21,12)(4,8,22,11)(17,18,19,20), (1,17,21,13)(4,20,24,15)(9,10,12,
    11), (6,13,11,19)(7,14,9,20)(21,22,23,24) ])
gap> Read("gap5.4.1");
gap> S := StabChain(cube);
<stabilizer chain record, Base [ 1, 3, 4, 6, 7, 9 ], Orbit length 21, Size:
3674160>
gap> f := FreeGroup("bo","li","ac");
<free group on the generators [ bo, li, ac ]>
gap> hom := GroupHomomorphismByImages(f,cube,GeneratorsOfGroup(f),
> GeneratorsOfGroup(cube));
[ bo, li, ac ] -> [ (3,7,21,12)(4,8,22,11)(17,18,19,20),
  (1,17,21,13)(4,20,24,15)(9,10,12,11), (6,13,11,19)(7,14,9,20)(21,22,23,24) ]
gap> baseim := [17,11,10,22,23,24];
[ 17, 11, 10, 22, 23, 24 ]
gap> g := permutationbybaseimage(S,baseim);
(1,17,15,4,10,12)(3,11)(6,22)(7,23)(8,20)(9,24,13)(14,19)(18,21)
gap> PreImagesRepresentative(hom,g^-1);
bo*ac^2*bo^-1*li^-1*ac*bo*li*ac^-1*li^-1*bo^-1*ac^-1*bo
```

Deze methode levert de stappen die we moeten uitvoeren om de puzzle op te lossen. De gevonden oplossing kan echter een lang woord in de generatoren zijn. Het vinden van

een kortste woord is een fundamenteel probleem. In termen van het Cayleygraaf kunnen we het als volgt omschrijven. Stel dat Γ een samenhangend graaf is. De *diameter* (**E:** *diameter*) van een graaf wordt gedefinieerd als de maximale afstand tussen twee toppen van Γ. Indien twee toppen adjacent zijn, dan is de afstand tussen de twee toppen als 1 gedefinieerd. De afstand tussen twee niet adjacente toppen wordt gedefinieerd als de lengte van het kortste pad dat de twee toppen verbind. Als $G = \langle X \rangle$ een groep is, dan is de diameter van het Cayleygraaf $\Gamma_X(G)$ gelijk aan $\max\{l_X(g) | g \in G\}$ waarbij $l_X(g)$ de lengte van het kortste woord is in $(X \cup X^{-1})$ dat g voorstelt.

Het bepalen van de diameter van een Cayleygraaf, en daarbij het bepalen van $l_X(g)$ voor een element $g \in G$, is een fundamenteel probleem in de computationele groepentheorie. Hoewel we de algoritmen voor eindige permutatiegroepen ons in staat stellen om een element $g \in G$ zeer efficiënt uit te drukken in een woord in de sterk voortbrengende generatoren (en via SLPs in de generatoren zelf), laat de ontwikkeling van deze algoritmen ons schijnbaar niet toe $g \in G$ te bepalen in relatief korte woorden in de oorspronkelijke generatoren.

Ondertussen is het geweten dat de diameter van het Cayleygraaf van de groep van de $3 \times 3 \times 3$ kubus tussen de 24 en 42 ligt, waarbij het waarschijnlijk is dat de diameter dicther bij 24 ligt dan bij 42.

Het vinden van een woord in de oorspronkelijke generatoren voor een element $g \in G$, wordt ook het *factorisatie probleem* genoemd. Er bestaan reeds heuristische algoritmen om de sterke generatoren van een permutatiegroep uit te drukken in relatief korte woorden in $X \cup X^{-1}$, en daarmee kunnen de elementen van een groep dan ook uitgedrukt worden in relatief korte woorden in $X \cup X^{-1}$. Op dit ogenblik is men echter nog maar in staat om een willekeurige elementen uit de groep van de $3 \times 3 \times 3$ kubus uit de drukken in woorden van lengte ten hoogste 100 in de 6 generatoren.

We hernemen nog eens het voorbeeld van de $2 \times 2 \times 2$-kubus en we berekenen met behulp van de GAP-package `grape` de diameter van het Cayleygraaf van de groep (`gap5.5.5.i`).

```
   Loading the library. Please be patient, this may take a while.
GAP4, Version: 4.4.6 of 02-Sep-2005, i686-pc-linux-gnu-gcc
Components:  small 2.1, small2 2.0, small3 2.0, small4 1.0, small5 1.0,
             small6 1.0, small7 1.0, small8 1.0, small9 1.0, small10 0.2,
             id2 3.0, id3 2.1, id4 1.0, id5 1.0, id6 1.0, id9 1.0, id10 0.1,
             trans 1.0, prim 2.1  loaded.
Packages:    TomLib 1.1.2  loaded.
gap> b := (17,18,19,20)(4,8,22,11)(3,7,21,12);
(3,7,21,12)(4,8,22,11)(17,18,19,20)
gap> l := (9,10,12,11)(13,1,17,21)(15,4,20,24);
(1,17,21,13)(4,20,24,15)(9,10,12,11)
gap> a := (21,22,23,24)(7,14,9,20)(6,13,11,19);
(6,13,11,19)(7,14,9,20)(21,22,23,24)
gap> cube := Group([b,l,a]);
Group([ (3,7,21,12)(4,8,22,11)(17,18,19,20), (1,17,21,13)(4,20,24,15)(9,10,12,
    11), (6,13,11,19)(7,14,9,20)(21,22,23,24) ])
gap> Size(cube);
3674160
gap> LoadPackage("grape");

Loading  GRAPE 4.2  (GRaph Algorithms using PErmutation groups),
by L.H.Soicher@qmul.ac.uk.

true
gap> Gamma := CayleyGraph(cube);;
```

```
gap> time;
136820
gap> Diameter(Gamma);
14
gap> time;
986720
```

5.6 Oefeningen

1. Implementeer de functie RandomStab in GAP en/of Magma
2. Gegeven $G = \langle a, b, c, d \rangle$, met $a = (1,2)(3,4)(5,9)(6,7)$, $c = (2,4,5,8,7,3)(6,9)$, $b = (2,5,7)$ en $d = (3,4,8)$. Bewijs dat $G \cong C_3 \wr S_3$. Gebruik de methodes uit Paragraaf 5.5

Bibliografie

[1] G. Butler. *Fundamental algorithms for permutation groups*, volume 559 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, 1991.

[2] The GAP Group. *GAP – Groups, Algorithms, and Programming, Version 4.4*, 2005. (`http://www.gap-system.org`).

[3] Derek F. Holt, Bettina Eick, and Eamonn A. O'Brien. *Handbook of computational group theory*. Discrete Mathematics and its Applications (Boca Raton). Chapman & Hall/CRC, Boca Raton, FL, 2005.

[4] A. Hoogewijs. *Computeralgebra*. cursus licenties wiskunde en informatica. Universiteit Gent, 2006.

[5] MAGMA. http://magma.maths.usyd.edu.au/magma/, 2006.

Index

G-congrentie, 75
i-de basis stabilizator, 81
i-de basisbaan, 81
n-voudig transitief, 25

abels, 1
actie, 22
afgeleide deelgroep, 39
alternerende groep, 3
automorfisme, 21

baan, 24
basis, 81
basis en sterk voortbrengende verzameling, 81
basisbeeld, 81
black-box groep, 54
blok, 26
bloksysteem, 26

Cayleygraaf, 23
centralisator, 25
centrum, 22
commutatief, 1
commutator, 39
commutatordeelgroep, 39
complement, 96
cyclische groep, 2
  generator, 3
cykel, 3

deelgroep, 2
defiërende relators, 28
deterministisch algoritme, 55
diëdergroep, 3
diameter, 101
direct product, 2

eenheidselement, 1
eindige groep, 1
endomorfisme, 21
enkelvoudig, 7
epimorfisme, 21
evaluatie, 53
even, 3

fixator, 24

generator, 3
gespleten extensie, 96
getrouw, 23
graad van een permutatievoorstelling, 23
groep, 1
  abels, 1
  commutatief, 1
  cyclisch, 2
  eenheidselement, 1
  eindige groep, 1
  orde, 1
  triviale groep, 1
groep extensie, 96
groep voorstelling, 28

homomorfisme, 21

imprimitief, 26
index, 2
intransitief, 25
inwendig automorfisme, 21
isomorf, 22

isomorfisme, 21

kern, 24
kern van een actie, 23
kernel, 22

Las Vegas algoritme, 55

monomorfisme, 21
Monte Carlo algoritme, 55

nevenklasse actie, 23
nilpotent, 39
normaaldeler, 7
normale sluiting, 7
normalisator, 25

oneven, 3
oplosbaar, 39
orbit-stabilizer, 24
orde, 1, 2

perfect, 40
permutatie, 3
permutatiegroep, 3
permutatievoorstelling, 22
primitief, 26

quotiëntgroep, 7

randomized algorithm, 55
rang, 27
rechts reguliere actie, 23
rechtse nevenklasse, 2
rechtse transversaal, 2
regulier, 26
relatie, 28
relator, 28
representant, 2

Schreiergeneratoren, 69
Schreiervector, 70
semidirect produkt, 22
SLP elementen, 53
SLP group, 53
stabilisator, 24
straight-line program, 53
symmetrische groep, 3

toegevoegd, 25
toegevoegde deelgroepen, 25
toevoeging, 25
toevoeginsklasse, 25
transitief, 25
transpositie, 3
tweede centrale afgeleide, 39
tweede normale afgeleide, 39

uiwendig automorfisme, 21

verzameling van sterk voortbrengende generatoren, 81
vrije groep, 27

wreath produkt, 26