Auteursrechterlijke overeenkomst

Opdat de Universiteit Hasselt uw eindverhandeling wereldwijd kan reproduceren, vertalen en distribueren is uw akkoord voor deze overeenkomst noodzakelijk. Gelieve de tijd te nemen om deze overeenkomst door te nemen, de gevraagde informatie in te vullen (en de overeenkomst te ondertekenen en af te geven).

Ik/wij verlenen het wereldwijde auteursrecht voor de ingediende eindverhandeling met

Titel: Graph Mining: het efficient en exhaustief genereren van grafen
Richting: 2de masterjaar in de informatica - databases Jaar: 2009

in alle mogelijke mediaformaten, - bestaande en in de toekomst te ontwikkelen - , aan de Universiteit Hasselt.

Niet tegenstaand deze toekenning van het auteursrecht aan de Universiteit Hasselt behoud ik als auteur het recht om de eindverhandeling, - in zijn geheel of gedeeltelijk -, vrij te reproduceren, (her)publiceren of distribueren zonder de toelating te moeten verkrijgen van de Universiteit Hasselt.

Ik bevestig dat de eindverhandeling mijn origineel werk is, en dat ik het recht heb om de rechten te verlenen die in deze overeenkomst worden beschreven. Ik verklaar tevens dat de eindverhandeling, naar mijn weten, het auteursrecht van anderen niet overtreedt.

Ik verklaar tevens dat ik voor het materiaal in de eindverhandeling dat beschermd wordt door het auteursrecht, de nodige toelatingen heb verkregen zodat ik deze ook aan de Universiteit Hasselt kan overdragen en dat dit duidelijk in de tekst en inhoud van de eindverhandeling werd genotificeerd.

Universiteit Hasselt zal mij als auteur(s) van de eindverhandeling identificeren en zal geen wijzigingen aanbrengen aan de eindverhandeling, uitgezonderd deze toegelaten door deze overeenkomst.

Ik ga akkoord,

DAENEN, Jonny

Datum: 14.12.2009

Graph Mining

Het efficient en exhaustief genereren van grafen

Jonny Daenen

promotor :
Prof. dr. Jan VAN DEN BUSSCHE

Eindverhandeling voorgedragen tot het bekomen van de
graad master in de informatica databases

universiteit hasselt

Dankwoord

Zonder de hulp en steun van enkele mensen was het niet mogelijk dat deze masterthesis tot stand kwam. Graag wil ik dan ook in de eerste plaats mijn promotor prof. dr. Jan Van den Bussche en mijn begeleider dr. Dries Van Dyck bedanken omdat ze mij de mogelijkheid hebben gegeven dit onderwerp te bestuderen en omdat ze mij de nodige hulp verschaft hebben.

Hiernaast wil ik ook mijn vriendin Helena bedanken voor haar steun, begrip, vertrouwen en de momenten van ontspanning. Verder wil ik ook mijn medestudenten en vrienden bedanken voor de discussies, hulp, het delen van kennis en de plezante belevenissen. Ten slotte mag ik mijn ouders en grootouders niet vergeten die mij hun steun, vertrouwen en de mogelijkheid tot studeren gegeven hebben.

Abstract

Vele zaken uit onze dagelijkse wereld kunnen voorgesteld worden met behulp van grafen. Denk hierbij bijvoorbeeld aan moleculen, sociale netwerken, 3D-modellen, processen, ... Soms kan het interessant zijn om in een verzameling van zulke structuren op zoek te gaan naar terugkerende patronen, zodat we bijvoorbeeld objecten makkelijker kunnen classificeren, of gemeenschappelijke eigenschappen kunnen koppelen aan een substructuur, ... Een manier om dit aan te pakken is om de objecten in een databank op te slaan en op zoek te gaan naar patronen die frequent voorkomen. Deze procedure noemen we *graph mining*.

Dit probleem is goed oplosbaar als we het over gewone verzamelingen van objecten hebben, we genereren de mogelijke verzamelingen en testen hun voorkomen. Als we hetzelfde bij grafen willen doen, dan zien we dat het probleem zich hier op een ander niveau bevindt. Het aantal grafen waarvan we het voorkomen kunnen testen groeit namelijk zeer snel: er zijn in tegenstelling tot aantal mogelijke deelverzamelingen veel meer mogelijke subgrafen die we kunnen testen. Verder is het een zeer ingewikkeld probleem om te beslissen of twee grafen "gelijk" zijn. Grafen dienen altijd op een geordende manier aan een algoritme gegeven te worden, bijvoorbeeld door een orde op de knopen te leggen. We kunnen nu zeggen dat twee grafen gelijk zijn indien ze enkel verschillen in deze orde, we noemen ze in dit geval *isomorf*. Om het graph mining probleem op te lossen moeten we dus voor elke mogelijke graaf testen hoeveel keer hij isomorf is met een subgraaf in de database.

Het ingewikkelde isomorfismeprobleem komt niet enkel voor bij de frequentietest, maar ook bij het genereren van grafen. Indien we op zoek zijn naar frequente patronen, dan moeten we natuurlijk ook kandidaatpatronen genereren. Uit dit generatieproces komen mogelijk isomorfe "kopieën", die we natuurlijk niet opnieuw op frequentie willen testen omdat dit geen bijdrage meer levert. Een naïve manier om dit te vermijden is om alle reeds gegenereerde grafen te testen op isomorfisme met de nieuwe graaf, dit is duidelijk niet performant wegens de dure isomorfietest. Bij deze aanpak dienen we voor elke nieuwe graaf deze ingewikkelde test één maal uit te voeren voor al zijn voorgangers, hetgeen duidelijk veel berekeningen zal vergen. Hiernaast is het vaak ook handig om niet alle grafen te testen of frequent voorkomen,

maar om een bepaalde klasse van grafen te beschouwen die voldoen aan een bepaalde eigenschap. Dit kan het geval zijn indien we bijvoorbeeld enkel geïnteresseerd zijn in complete subgrafen.

Het doel van deze thesis bestaat eruit om een duidelijk beeld te vormen van hoe graafisomorfisme in elkaar zit, en om op zoek te gaan naar performante enumeratieprocessen die isomorfe kopieën vermijden. Verder onderzoeken we hoe bepaalde klassen van grafen die aan een bepaalde eigenschap voldoen beschreven en gegenereerd kunnen worden, eventueel gebruik makend van de besproken algoritmen.

In het eerste deel van deze thesis onderzoeken we enkele algoritmen die trachten de generatie van "isomorfe kopieën" te vermijden. We diepen eerst de noties van grafen, isomorfismen en andere verwante onderwerpen uit, alvorens over te gaan tot de concrete algoritmen.

In het tweede deel gaan we dieper in op het genereren van klassen van grafen die aan een bepaalde eigenschap voldoen. We bestuderen hiervoor de notie van graafgrammatica's, graafexpressies en de hiermee verwante k-*trees*. Ook wordt er een aanzet gegeven naar mogelijke algoritmen om dit soort klassen van grafen efficiënt te enumereren.

Inhoudsopgave

Lijst van figuren

Lijst van algoritmen

Hoofdstuk 1

Inleiding

Wanneer we een database met gegevens beschouwen, kunnen we op zoek gaan naar patronen die frequent in de dataset voorkomen. Er zijn reeds een groot aantal bekende algoritmes die dit probleem oplossen, zoals APRIORI, waarvan een beschrijving wordt gegeven door Tan et al. [33, p. 333 ev.]. De data bestaat uit transacties, zo een transactie kan bijvoorbeeld een verzameling gekochte goederen zijn: {melk, water, brood, boter}. We kunnen vervolgens de verzameling van alle mogelijke goederen beschouwen en hier deelverzamelingen van nemen. Voor elk van deze deelverzamelingen kunnen we testen of ze wel frequent in de database voorkomen, met andere woorden, ze zijn aanwezig in "voldoende" transacties. Dit wordt gedefinieerd door middel van een constante: de *minimale support*. Deze geeft aan hoeveel keer een lijst van goederen minimaal moet voorkomen in de database om als frequent beschouwd te worden. Het algoritme zal als uitvoer een lijst van verzamelingen geven die allemaal frequent zijn.

Hieruit kunnen we vervolgens nuttige regels construeren die in praktijk uitgebuit kunnen worden in bijvoorbeeld reclamecampagnes. Zo kan er bijvoorbeeld de regel "Wie brood koopt, koopt in 90% van de gevallen ook boter" afgeleid worden. Supermarkten kunnen hun conclusies trekken en boter en brood dicht bij elkaar plaatsen, zowel in de winkel als in reclamefolders.

Deze techniek is vanzelfsprekend niet enkel van toepassing op supermarkten of andere commerciële instellingen, maar wordt ook gebruikt in de wetenschap, geneeskunde,... Meer voorbeelden van toepassingen worden gegeven door Tan et al. [33, Hoofdstuk 1].

Er is echter ook data die we kunnen voorstellen met behulp van een *graaf*. Een graaf is een structuur die in het algemeen gebruikt kan worden om objecten te modelleren. We kunnen nu een database van grafen[1] beschouwen en hetzelfde idee toepassen. We kunnen nu op zoek gaan naar grafen die

[1]Soms bestaat de database slechts uit één grote graaf waarin we interne patronen willen ontdekken, maar wij beschouwen in essentie een verzameling van grafen.

frequent voorkomen in de database, waarmee we in dit geval bedoelen dat ze voldoende moeten voorkomen als een subgraaf van graaf uit de database.

Deze mining-methode heeft toepassingen in verschillende gebieden. Zo is het minen bijvoorbeeld belangrijk bij het zoeken naar nieuwe medicijnen (zie Agrafiotis et al. [3] en Borgelt et al. [9]). Een ander voorbeeld is het ontdekken van sociale netwerken aan de hand van e-maillogs (zie Tyler et al. [34]). Naast de traditionele aanpak van het eerder beschreven "supermarktprobleem" met behulp van verzamelingen van items, kunnen we dit ook aanpakken door graph mining technieken te gebruiken. Katsutoshi et al. [36] geven aan hoe dit mogelijk is en Kuroda et al. [22] geven een voorbeeld van ontdekkingen op de markt van alcoholische dranken, aan het licht gebracht door deze techniek te gebruiken. Het is duidelijk dat er veel nuttige toepassingen bestaan die objecten met een bepaalde structuur minen.

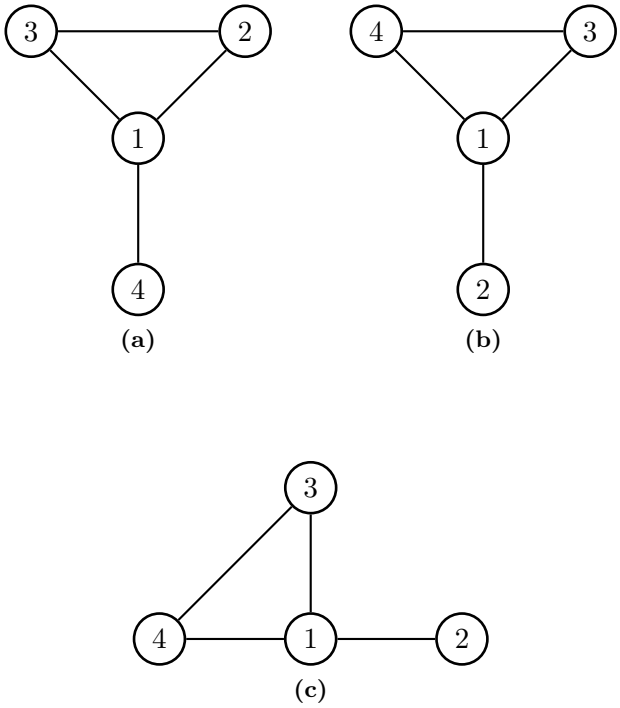
Maar de algoritmen moeten ook hier kandidaatgrafen genereren die getest moeten worden op frequentie, we doen dit door de grafen één voor één te genereren. Dit is echter, in tegenstelling tot het construeren van *itemsets* geen eenvoudige opdracht, er bestaan massa's grafen, zelfs voor een beperkt aantal knopen. Verder is het ook niet eenvoudig om te controleren of een graaf voorkomt als subgraaf van een andere: we moeten kijken of de gegeven graaf *isomorf* of "structureel identiek" is met minstens één subgraaf van de andere graaf.

Wanneer we een algoritme een graaf als invoer geven, moet er noodzakelijkerwijs een orde op liggen. Hiermee bedoelen we dat bijvoorbeeld de knopen in een bepaalde volgorde aan het algoritme gegeven moeten worden. We kunnen nu zeggen dat twee grafen *isomorf* zijn met elkaar als ze enkel in deze orde verschillen.

Voorbeeld 1.1. *De grafen in Figuur 1.1 zijn isomorf met elkaar, merk op dat de manier waarop we de graaf tekenen helemaal niet uitmaakt. De knopen van de grafen in Figuur 1.1a en 1.1b zijn op een andere manier geordend, en de graaf in Figuur 1.1c is exact dezelfde graaf als deze in Figuur 1.1b, enkel op een andere manier afgebeeld.*

Wanneer we grafen genereren is het mogelijk dat er zogenaamde "isomorfe kopieën" gegenereerd worden. Deze grafen stellen dus in essentie dezelfde graaf voor en het is bijgevolg nutteloos dat deze allemaal geconstrueerd worden. Dit aantal "nutteloze grafen" kan oplopen tot $n!$, waarbij n het aantal knopen van de graaf is[2]. Hierdoor testen we eenzelfde graaf enorm veel gaan op zijn voorkomen in de database. Dit is echter complete verspilling van tijd, en we willen dus een manier ontwikkelen om deze isomorfe grafen niet te genereren.

[2] $n!$ is het aantal mogelijke permutaties van n knopen, zoals we later zullen zien. Hiernaast is het echter ook nog mogelijk dat eenzelfde graaf meerdere malen in de uitvoer voorkomt, dit is natuurlijk ook niet gewenst.



Figuur 1.1: Isomorfe grafen. Ook al verschilt de orde op de knopen, de grafen zijn eigenlijk hetzelfde. De manier waarop ze getekend worden maakt ook niets uit.

Een manier om dit te bekomen is door voor elke gegenereerde graaf te gaan kijken of er al een isomorfe variant gegenereerd is. Dit houdt in dat we alle grafen die ervóór gegenereerd zijn moeten testen op isomorfie met de nieuwe. We kunnen weliswaar wat trucjes gebruiken, zoals kijken naar het aantal knopen, maar er zullen in het algemeen tests zijn die we niet kunnen vereenvoudigen. Dit is echter een zeer naïeve methode, we zouden dan alle eerder gegenereerde grafen in het geheugen moeten houden!

Een betere methode is echter om de grafen in te delen in groepjes die elk bestaan uit alle grafen die isomorf zijn met elkaar. Vervolgens voorzien we een manier om uit deze groep precies één graaf te kiezen, deze noemen we de *canonische graaf* van deze groep. Telkens wanneer we een graaf genereren, kijken we of deze het canonische object van zijn groep is. Is dit niet zo, dan weigeren we hem. De berekening van een dergelijk canonisch object is echter niet eenvoudig, het kan vergeleken worden met een isomorfismetest.

1.1 Aanpak en indeling

In deze thesis concentreren we ons eerst op de notie van isomorfie en gaan vervolgens op zoek naar algoritmen die toelaten om grafen te genereren op een isomorfvrije manier. Hiernaast bestuderen we ook de noties van graafgrammatica's en graafexpressies omdat deze ons toestaan klassen van grafen te beschrijven.

De aanpak die we gehanteerd hebben is de volgende: de nodige informatiebronnen werden gezocht en vervolgens verwerkt. We hebben in deze thesis gekozen om de uitleg zo duidelijk mogelijk weer te geven, zonder dat de lezer al te vaak op zoek moet gaan naar extern referentiemateriaal. Hierom hebben we gekozen om de bewijzen — die vaak nogal beknopt worden weergegeven — zo goed mogelijk uit te werken. Het uitzoeken van hoe deze bewijzen in elkaar staken was een deel van de weg die afgelegd diende te worden om de materie onder de knie te krijgen. Verder hebben we ook getracht om waar mogelijk conclusies te trekken en verschillende aanpakken te vergelijken om zo nieuwe inzichten te verwerven.

In hoofdstuk 2 bespreken we enkele begrippen die nodig zijn om de algoritmen en constructies te begrijpen, en het nut ervan in te zien. De uitleg over grafen en groepen is ruwweg gebaseerd op de uitleg van Diestel [14], Bollobás [8] en Scott [30]. Verder bespreken we ook nog wat een *full-invariant* algoritme en een graafcanonisatie algoritme is, zoals beschreven door Gurevich [17].

In hoofdstuk 3 worden twee aanpakken beschreven, namelijk het orderly algoritme en het de techniek die gebruikt wordt door McKay [24]. Het orderly algoritme wordt beschreven door Read [26] en een specifieke invulling hiervan is het gSpan-algoritme, gegeven door Yan [37]. We zullen elk van deze algoritmen bekijken en trachten om een vergelijking ertussen op te

bouwen. Elk van deze algoritmen maakt gebruik van een aparte voorstelling voor objecten/grafen, van een groei-operatie (om van een "kleinere" graaf een grotere te construeren) en van een soort canonisatie. We proberen ook te kijken naar hoe de algoritmen in elkaar passen en welke delen we kunnen hergebruiken.

In hoofdstukken 4 en 5 geven we een overzicht van het geïmplementeerde framework en het generatieprogramma, waarna we een vergelijking tussen de verschillende technieken trachten te maken. Bij het beschrijven van het framework geven we eveneens een verantwoording voor bepaalde keuzes en waar nodig worden verschillende mogelijke algoritmen gegeven.

Hoofdstuk 6 beschijft een manier die ons toelaat om klassen van grafen te beschrijven: de graafgrammatica's. Hierbij komt ook nog de notie van graafexpressies naar boven, die in hoofdstuk 7 en 8 gelinkt wordt aan het concept partiële *k-trees*. Hiertoe hebben Van Dyck [35], Bronner et al. [11] en Heggernes [19] onder andere bijgedragen. De algoritmen die bij de partiële *k-trees* hoorden hebben we te danken aan onder andere Arnborg [4] en Bodlaender [6]. Hoofdstuk 8 gaat dieper in op deze algoritmes, die eigenlijk voor partiële *k-trees* ontworpen zijn, maar nu ook blijken te werken voor graafexpressies.

In hoofdstuk 9 tenslotte, geven we een overzicht van verschillende algoritmen die alle grafen van een klasse isomorfvrij kunnen genereren, waarbij die klasse beschreven wordt door een graafgrammatica. We geven hier ook een aanzet naar het toekomstige werk dat in dit gebied verricht kan worden.

Hoofdstuk 2

Achtergrond in graaftheorie

In dit hoofdstuk definiëren we enkele fundamentele begrippen die in hetgeen volgt belangrijk zijn. We zullen het hebben over grafen, groepen, isomorfismen, ...

2.1 Grafen

Definitie 2.1 (Ongerichte graaf)**.** Een *ongerichte graaf* is een paar $G = (V, E)$, waarbij V een verzameling knopen is, en E een verzameling bogen. Hierbij is $E \subseteq \{\{v, w\} \mid v, w \in V\}$. Om de verzameling van knopen (resp. bogen) van een graaf G aan te duiden, schrijven we ook $V(G)$ (resp. $E(G)$).

Definitie 2.2 (Gerichte graaf)**.** Een *gerichte graaf* is een paar $G = (V, E)$, waarbij V een verzameling knopen is, en E een verzameling bogen. Hierbij is $E \subseteq V \times V$. Bogen zijn met andere woorden van de vorm (v_1, v_2). Om de verzameling van knopen (resp. bogen) van een graaf G aan te duiden, schrijven we ook $V(G)$ (resp. $E(G)$).

We definiëren de *orde* van een graaf G als het aantal knopen waaruit G bestaat en noteren de orde met $o(G) = |V(G)|$. De *grootte* van een graaf G is gelijk aan het aantal bogen in de graaf: $size(G) = |E(G)|$.

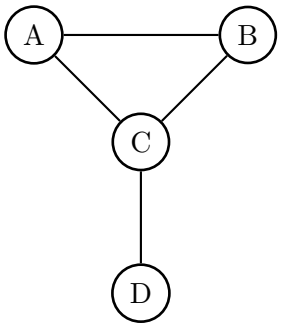
Voor een knoop $v \in V(G)$ definiëren we de buren als: $N(v) = \{w \mid \{v, w\} \in E\}$. De *graad* van deze knoop wordt gegeven door de grootte van deze verzameling, namelijk $|N(v)|$, en wordt aangeduid door $d(v)$. Als $d(v) = 0$, dan noemen we v een *geïsoleerde knoop*.

Voorbeeld 2.3. *In Figuur 2.1 zien we een graaf G waarbij de verzameling knopen gelijk is aan*

$$V(G) = \{A, B, C, D\},$$

en de verzameling bogen

$$E(G) = \{\{A, B\}, \{A, C\}, \{B, C\}, \{C, D\}\}.$$



Figuur 2.1: Een graaf met 4 knopen en 4 bogen.

Verder geldt

$$d(A) = 2 \qquad d(B) = 2 \qquad d(C) = 3 \qquad d(D) = 1. \tag{2.1}$$

De orde van deze graaf is 4, net zoals zijn graad. De buren van knoop A zijn B en C: $N(A) = \{B, C\}$.

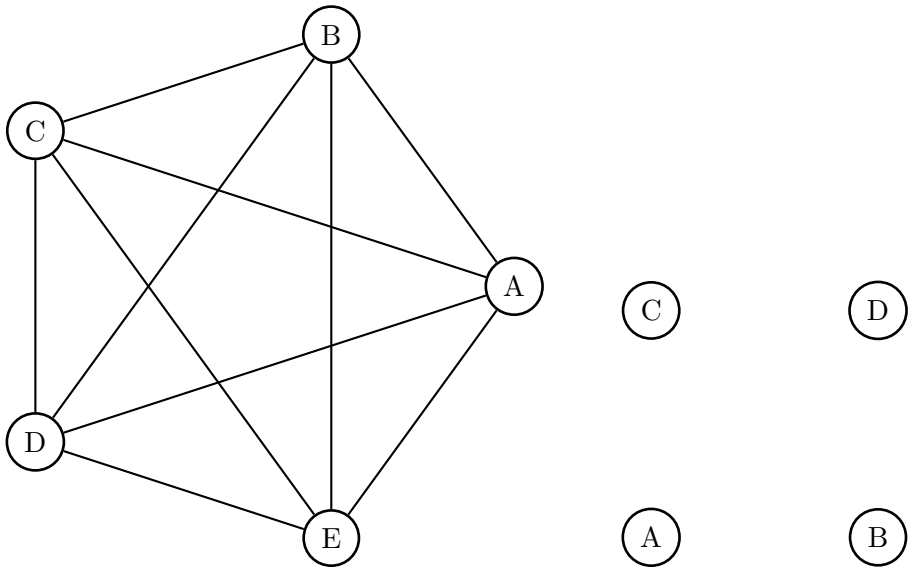
Een speciaal soort graaf is de *complete graaf*. Dit is een graaf die een aantal knopen bevat, die allemaal met elkaar verbonden zijn. We kunnen dus zeggen dat voor een complete graaf $G = (V, E)$ geldt dat

$$E = \{\{v, w\} \mid v, w \in V, v \neq w\}.$$

De verzameling bogen is dus maximaal. We noteren de complete graaf op n knopen als K_n. De tegenhanger van deze graaf is de *lege graaf*. Deze graaf bevat een lege bogenverzameling ($E = \emptyset$). We noteren de lege graaf op n knopen als E_n.

Voorbeeld 2.4. *In Figuur 2.2a zien we de complete graaf op vijf knopen:* K_5. *Elke knoop in de graaf is rechtstreeks verbonden met elke andere knoop in de graaf. In Figuur 2.2b zien de lege graaf op vier knopen:* E_4. *In deze laatste graaf zijn geen bogen terug te vinden, de knopen staan allen los van elkaar.*

Definitie 2.5 (Subgraaf)**.** Een graaf $H = (V_H, E_H)$ is een *subgraaf* van een graaf $G = (V_G, E_G)$ als en slechts als $V_H \subseteq V_G$ en $E_H \subseteq E_G$, en noteren dit als $H \subseteq G$. We zeggen dan ook wel dat G een *supergraaf* is van H. In het bijzonder geldt dat $G \subseteq G$. We noemen H een *geïnduceerde subgraaf* van G als en slechts als geldt dat $H \subseteq G$ en $E_H = \{\{v, w\} \mid v, w \in V(H) \wedge \{v, w\} \in E(G)\}$. We zeggen dan dat H *geïnduceerd* wordt door V_H in G en schrijven $H = G[V_H]$. Een geïnduceerde subgraaf bevat dus een



(a) De complete graaf K_5 **(b)** De lege graaf E_4

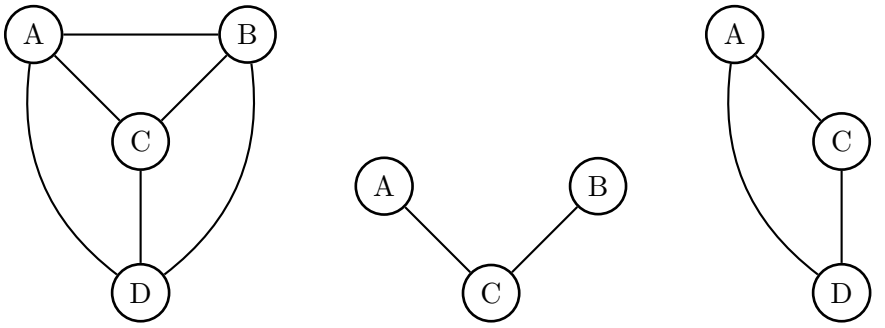
Figuur 2.2: Een complete en een lege graaf.

deelverzameling van de knopen uit G, en alle bogen uit G die twee knopen uit deze deelverzameling verbinden.

Tot slot noemen we H een *opspannende deelgraaf* als geldt dat $H \subseteq G$ en $V_H = V_G$.

Merk op dat de bovenstaande definitie niet toelaat dat een graaf $H \subseteq G$ een boog uit G bevat, maar niet alle knopen die door deze boog verbonden worden: in dat geval zou H immers geen geldige graaf zijn.

Voorbeeld 2.6. *In Figuur 2.3a zien we een graaf G, bestaande uit vier knopen en zes bogen. In Figuur 2.3b zien we een mogelijke subgraaf van G, waarbij enkel de knopen A, B en C behouden werden, en slechts twee van de zes bogen. Merk op dat deze subgraaf geen geïnduceerde subgraaf van G is, omdat de boog $\{A, B\}$ verwijderd is. Tot slot zien we in Figuur 2.3c een subgraaf van G waarbij enkel de knopen A, C en D zijn overgehouden, samen met alle bogen die tussen deze knopen bestonden in G. Hierdoor is het duidelijk dat deze graaf wél een geïnduceerde graaf van G is, hij is namelijk gelijk aan $G[\{A, C, D\}]$.*



(a) de graaf G **(b)** een subgraaf van G **(c)** een geïnduceerde subgraaf van G

Figuur 2.3: Een graaf met enkele subgrafen.

2.2 Isomorfismen

In dit deel zullen we trachten om een algemene definitie te geven van een isomorfisme toegepast op een algemene *structuur*. We zullen eerst aangeven waaruit zo een structuur bestaat, en vervolgens vermelden hoe we een graaf kunnen bekijken in de context van deze definitie. Dit zal aanleiding geven tot een definitie voor graafisomorfisme.

Definitie 2.7 (Vocabulary)**.** Een *vocabulary* τ is een verzameling van

- constante symbolen,
- relatiesymbolen met een zekere ariteit $k \in \mathbb{N}$, en
- functiesymbolen met een zekere ariteit $l \in \mathbb{N}$.

Met ariteit wordt geduid op het aantal argumenten van een functie of een relatie. De functie $f(x, y) = z$ heeft ariteit 2, net als de relatie $R(u, v)$.

Definitie 2.8 (Structuur)**.** Een *structuur* van type τ is een koppel $\mathcal{A} = \{\theta_A, A\}$, waarbij A een verzameling is (het *universum*) en θ_A een functie die elk symbool uit τ een betekenis geeft:

- $\theta_A(c) \in A$, waarbij c een constante is,
- $\theta_A(R) \subseteq A^k$, waarbij R een relatiesymbool met ariteit k is, en
- $\theta_A(f)$ een functie is van $A^l \to A$, waarbij f een functie is ariteit l.

Definitie 2.9 (Isomorfisme)**.** Neem een *vocabulary* τ en twee structuren $\mathcal{A} = (\theta_A, A)$ en $\mathcal{B} = (\theta_B, B)$, beide van type τ. Er geldt dat $\mathcal{A}$ *isomorf* is met $\mathcal{B}$, genoteerd $\mathcal{A} \cong \mathcal{B}$ als er een bijectie $\pi : A \to B$ bestaat die alles bewaart:

- $\pi(\theta_A(c)) = \theta_B(c)$, waarbij c een constante is uit τ,
- als R een relatiesymbool is met ariteit k, dan geldt

$$\forall a_1, \ldots, a_k \in A^k : (a_1, \ldots, a_k) \in \theta_A(R) \Leftrightarrow (\pi(a_1), \ldots, \pi(a_k)) \in \theta_B(R), \tag{2.2}$$

- g is een functiesymbool met ariteit l, dan geldt

$$\forall a_1, \ldots, a_{l+1} \in A^{l+1} : \\ g^{\theta_A}(a_1, \ldots, a_l) = a_{l+1} \Leftrightarrow g^{\theta_B}(\pi(a_1), \ldots, \pi(a_l)) = \pi(a_{l+1}). \tag{2.3}$$

  Hierbij is g^{θ_A} een andere notatie voor $\theta_A(g)$.

Als dit geldt schrijven we $\pi(\mathcal{A}) = \mathcal{B}$, of equivalent: $\mathcal{A}^\pi = \mathcal{B}$.

Voorbeeld 2.10. *Beschouw een een vocabulary τ_g voor gerichte grafen. We nemen één relatie E, met ariteit 2, andere relatiesymbolen, functiesymbolen of constanten zijn niet aanwezig. Een structuur $G = (\theta_G, V)$ van type τ_g bevat een verzameling knopen V en een functie θ_G. θ_G levert ons een deelverzameling van de knopen op, die de relatie E voorstelt. Deze relatie E bevat elementen de vorm (v, w), waarbij $(v, w) \in E$ als en slechts als er in de graaf G een boog bestaat van knoop v naar w. Wanneer we op deze structuur de notie van isomorfie beschouwen, zoals gedefinieerd in definitie 2.9, bekomen we het volgende: Twee gerichte grafen G en H zijn isomorf, genoteerd $G \cong G$, als er een bijectie $\pi : V_G \to V_H$ bestaat zodat*

$$\forall a_1, a_2 \in V_G^2 : (a_1, a_2) \in \theta_G(E) \Leftrightarrow (\pi(a_1), \pi(a_2)) \in \theta_H(E).$$

Hierbij is $V_G^2 = V_G \times V_G$. Met andere woorden, er dient een bijectie te bestaan van de ene knopenverzameling naar de andere, zodat de bogen tussen de overeenkomstige knopen behouden blijven. Omdat π een bijectie moet zijn, is het duidelijk dat het aantal knopen hetzelfde dient te zijn in de twee grafen.

Voorbeeld 2.11. *Wanneer we de gerichte grafen uit voorbeeld 2.10 veranderen in ongerichte grafen moeten we ook de boogrelatie E aanpassen. Een boog $(v, w) \in E$ wil in dit geval zeggen dat er een* ongerichte *boog van v naar w in de graaf aanwezig is. We zorgen echter ook dat indien een boog $(v, w) \in E$, dat dan ook $(w, v) \in E$. Er geldt dus:*

$$(v, w) \in E \wedge (w, v) \in E \Leftrightarrow \text{ de graaf bevat een boog tussen } v \text{ en } w.$$

Op deze manier kunnen we een ongerichte graaf voorstellen, waarbij er wel wat redundante informatie in de relatie E zit (elke boog wordt namelijk twee keer bijgehouden). Vergelijking 2.10 blijft exact hetzelfde. Indien we slechts één koppel per boog in de relatie E zouden bijhouden, dan dienden we deze vergelijking aan te passen, zodat in $\theta_H(E)$ ofwel de boog $(\pi(a_1), \pi(a_2))$ ofwel $(\pi(a_2), \pi(a_1))$ moet zitten.

Voorbeeld 2.12. *We beschouwen een vocabulary voor grafen waarin twee knopen gemarkeerd zijn: een startknoop en een eindknoop. Dit soort grafen kan bijvoorbeeld gebruikt worden als invoer voor een algoritme dat het kortste pad tussen de twee knopen gaat berekenen. We beschouwen gerichte grafen. We kunnen de vocabulary beschreven in voorbeeld 2.10 uitbreiden met twee constanten s en e, die respectievelijk de start- en de eindknoop voorstellen. De functie θ_G dient deze twee constanten af te beelden op een knoop uit de verzameling V. De notie van isomorfisme dient hierbij aangepast te worden, zodat de gemarkeerde knopen ook op elkaar afgebeeld worden door de bijectie π.*

Opmerking 2.13. In de plaats van de structuur in definitie 2.8 kunnen we een *relationele structuur* beschouwen, waarbij we enkel gebruik maken van constanten en relaties. De functies met ariteit l kunnen immers "gecodeerd" worden als relaties met ariteit $l+1$. Ook kunnen we een *functionele structuur* beschouwen, waarbij enkel gebruik gemaakt wordt van functies.

Uit bovenstaande voorbeelden besluiten we:

Besluit 2.14. *(Graafisomorfisme) We noemen twee grafen G_1 en G_2 isomorf als en slechts als er een bijectie $\pi : V(G_1) \to V(G_2)$ bestaat zodat*

- $\forall v \in V(G_1) : \pi(v) \in V(G_2)$
- $\{v, w\} \in E(G_1) \Leftrightarrow \{\pi(v), \pi(w)\} \in E(G_2)$

Definitie 2.15. (Permutatie) Een *permutatie* van een verzameling A is een bijectie van de vorm $A \to A$.

Wanneer we tussen de knopen van twee verschillende grafen, met een gelijke knopenverzameling, een isomorfisme beschouwen, dan zal dit een permutatie zijn. Iedere knoop wordt immers afgebeeld op een knoop uit dezelfde verzameling. Wanneer we spreken over "een permutatie van een graaf" bedoelen we eigenlijk een permutatie van zijn knopen. Er bestaan $n!$ mogelijke permutaties van een verzameling van grootte n. Indien een graaf n knopen bevat, dan zijn er dus $n!$ isomorfe grafen.

Voorbeeld 2.16. *Wanneer we op de knopen van de graaf in Figuur 2.4a de permutatie uit vergelijking 2.4 toepassen, bekomen we de graaf in Figuur 2.4b. Het is duidelijk dat deze nieuwe graaf isomorf is met de eerste, omdat*

we een bijectie van de knopen hebben (namelijk π). In ons geval was deze bijectie een permutatie van de knopenverzameling.

$$\pi = \begin{cases} 1 \to 1 \\ 2 \to 3 \\ 3 \to 4 \\ 4 \to 2 \end{cases} \tag{2.4}$$

De notie van permutatie is belangrijk, omdat we gebruik zullen maken van grafen waarbij de knopenverzameling gelijk is.

Definitie 2.17. Gegeven een structuur $\mathcal{A} = (\theta_A, A)$, dan noemen een permutatie $\pi : A \to A$ een *automorfisme* als π een isomorfisme is van $\mathcal{A}$ naar $\mathcal{A}$.

Het is duidelijk dat de identieke functie een automorfisme is voor eender welk object.

Voorbeeld 2.18. *Beschouw opnieuw de graaf uit Figuur 2.4a. Indien we de identieke permutatie toepassen op de graaf, dan levert ons dit duidelijk dezelfde graaf op, maar indien we de permutatie uit vergelijking 2.5 toepassen, dan bekomen we eveneens dezelfde graaf.*

$$\pi_{aut} = \begin{cases} 1 \to 2 \\ 2 \to 1 \\ 3 \to 3 \\ 4 \to 4 \end{cases} \tag{2.5}$$

Dit kunnen we eenvoudig inzien door te kijken wat de permutatie met de graaf doet. De graaf bestaat uit de bogenverzameling

$$E = \{\{1,2\},\{1,3\},\{2,3\},\{3,4\}\}.$$

Het toepassen van π_{aut} hierop geeft ons

$$E' = \{\{2,1\},\{2,3\},\{1,3\},\{3,4\}\}.$$

Merk op dat dit exact dezelfde verzameling[1] *is! Hieruit volgt rechtstreeks dat π_{aut} een automorfisme is voor de beschouwde graaf.*

2.2.1 De cyclusnotatie

Tot slot van deze sectie beschrijven we een compacte notatie van permutaties: de cyclusnotatie of cykelnotatie. Een beknopte, maar goede beschrijving van deze notatie wordt gegeven door Davis [13] en door Scott [30]. Wij

[1] De orde in een verzameling is niet van belang.

geven hier een beschrijving aan de hand van de grafen uit voorbeeld 2.16. We gaan ervan uit dat de knopenverzameling van alle grafen gelijk is aan $\{1, \ldots, n\}$, voor een zekere $n \in \mathbb{N}$. Een permutatie van deze verzameling beeldt de knopen onderling op elkaar af.

We beginnen met een knoop in de oorspronkelijke graaf en noteren zijn nummer, bijvoorbeeld 2. Hierna schrijven we de knoop waarop deze wordt afgebeeld, 3 in dit geval, nu doen we hetzelfde voor knoop 3 en we zien dat deze op 4 wordt afgebeeld. Tot nu toe hebben we dus

$$2\ 3\ 4. \tag{2.6}$$

Vervolgens gaan we verder met 4, maar dit wordt op 2 afgebeeld, maar dit hebben we reeds genoteerd, de cykel is dus rond en we omgeven de verkregen sequentie met haken:

$$(2\ 3\ 4). \tag{2.7}$$

We kiezen vervolgens een nieuwe knoop uit die nog niet gebruikt is. In dit geval blijft er slechts eentje over, namelijk knoop 1. Hierop passen we dezelfde procedure toe en we verkrijgen de cyclus (1). Tot slot "plakken" we de verkregen cycli op willekeurige wijze achter mekaar en verkrijgen een compacte notatie voor de permutatie:

$$(2\ 3\ 4)(1). \tag{2.8}$$

We merken ten slotte op dat de cycli die slechts één element bevatten vaak worden weggelaten. We bekomen dus

$$(2\ 3\ 4) \tag{2.9}$$

als finale voorstelling van de permutatie. Deze notatie is veel korter dan bijvoorbeeld degene die gebruikt wordt in vergelijking 2.4.

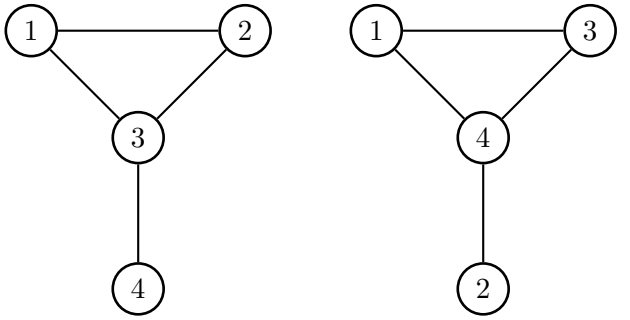
Een alternatief om permutaties te noteren is de "gewone" notatie: 1 3 4 2 komt overeen met de permutatie (2 3 4). De posities van de cijfers worden afgebeeld op het cijfer dat op deze positie staat.

2.3 Groepen

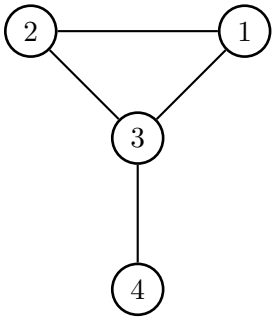
In deze sectie zullen we een wiskundige structuur beschrijven, die handig zal blijken wanneer we het over isomorfismen en generatie van grafen hebben, namelijk groepen. Meer uitleg over groepen wordt gegeven door Scott [30].

Definitie 2.19. Een *groep* is een geordend paar $(\mathcal{G}, \circ)$, waarbij $\mathcal{G}$ een verzameling is, en $\circ$ een binaire operatie op G. Het geordend paar moet aan de volgende voorwaarden voldoen:

- $\mathcal{G}$ is gesloten onder $\circ$,



(a) een graaf G **(b)** (2 3 4) toegepast op G



(c) (1 2) toegepast op G

Figuur 2.4: De cykelnotatie.

- $\exists e \in \mathcal{G}$, zodat als $a \in \mathcal{G}$, dan $a \circ e = a$.
  e wordt ook wel het *neutraal element* , of de identiteit van G genoemd,
- $\forall a \in \mathcal{G}, \exists a^{-1} \in \mathcal{G} : a \circ a^{-1} = e$.
  a^{-1} wordt ook wel het invers van a in $\mathcal{G}$ genoemd,
- de operatie $\circ$ is associatief:

$$a \circ (b \circ c) = (a \circ b) \circ c. \tag{2.10}$$

Vaak wordt de groep $(\mathcal{G}, \circ)$ eenvoudigweg genoteerd als $\mathcal{G}$.

Merk op dat bovenstaande definitie geen grens geeft op de grootte van een groep, het is dus perfect mogelijk dat een groep uit een oneindig aantal elementen bestaat. Verder zien we ook dat de commutativiteit geen voorwaarde is om een groep te vormen, indien deze wel aanwezig is, dan wordt er van een zogenaamde *Abelse groep*[2] gesproken. Hier gaan we niet verder op in.

Definitie 2.20. De verzameling $Sym(X)$ is de verzameling van alle permutaties op de verzameling X.

Voorbeeld 2.21. *Neem de verzameling A als de verzameling van alle grafen op n knopen. Voor elke $G \in A$ geldt dat $V(G) = \{1, \ldots, n\}$. De verzameling $\{1, \ldots, n\}$ noemen we een* initieel segment *van de verzameling van de natuurlijke getallen $\mathbb{N}$. We nemen nu alle mogelijke bijecties van de vorm $\{1, \ldots, n\} \rightarrow \{1, \ldots, n\}$ en noem deze verzameling $\mathcal{G}$. Merk op dat volgens definitie 2.15 elk van deze bijecties een permutatie is, omdat ze een verzameling op zichzelf afbeeld. We noteren ook $\mathcal{G} = S_n = Sym(1, \ldots, n)$.*

Merk op dat een permutatie $\pi \in \mathcal{G}$ geen betekenis heeft op een graaf $G \in A$, enkel op $V(G)$. We definiëren nu met behulp van de permutaties op de knopen de permutaties op de grafen als volgt: Neem een $\pi \in \mathcal{G}$, beschouw de functie $\overline{\pi} : A \rightarrow A$ die als volgt gedefinieerd is:

$$\overline{\pi}(G) = G' = (V(G) = \{1, \ldots, n\}, \pi(E)),$$

waarbij

$$\pi(E) = \{\{\pi(v), \pi(w)\} \mid \{v, w\} \in E\}.$$

De functie $\overline{\pi}$ beeldt dus grafen af op een isomorfe graaf, de knopen worden onderling "verwisseld". Het is duidelijk dat deze functie op elk element van A toegepast kan worden en dat het resultaat opnieuw een element van A is: de functie geeft namelijk een isomorfe graaf terug die evenveel dezelfde knopenverzameling bevat, en daar we alle grafen met deze knopenverzameling beschouwen, mogen we besluiten dat deze graaf opnieuw in A zit. We kunnen

[2] voor meer informatie over groepen verwijzern we naar Scott [30].

aantonen dat $\overline{\pi}$ een bijectie is, waardoor we (samen met het feit dat we opnieuw een element van A verkrijgen) mogen besluiten dat $\overline{\pi}$ een permutatie van A is.

Voor iedere permutatie π op de knopen kunnen we vervolgens een permutatie op de grafen $\overline{\pi}$ construeren. We zeggen dat een permutatie van de knopen een permutatie op de grafen induceert.

We nemen nu voor $\mathcal{G}$ alle permutaties op de grafen die geïnduceerd worden door de permutaties op de knopen:

$$\mathcal{G} = \{\overline{\pi} \mid \pi \in S_n\}.$$

We weten dat dit de permutaties op A (!) zijn waaronder graafisomorfisme bewaard wordt. Andere mogelijke permutaties kunnen bijvoorbeeld een graaf met drie bogen afbeelden op eentje met twee bogen. Dit soort permutaties willen we echter niet zien als isomorfismen, omdat ze de structuur van de graaf niet bewaren[3]*.*

We kunnen nu aantonen dat $(\mathcal{G}, \circ)$ een groep is, waarbij $\circ$ de compositie van de permutaties voorstelt die als volgt bekomen wordt:

$$(\overline{\pi} \circ \overline{\rho})(G) = \overline{\pi}(\overline{\rho}(G)).$$

- *Het is duidelijk dat $\mathcal{G}$ gesloten is onder $\circ$, omdat de compositie ons een compositie van knooppermutaties oplevert. Dit laatste levert ons opnieuw een geldige knooppermutatie op die, aangenomen dat knooppermutaties gesloten zijn onder $\circ$, een geldige graafpermutatie induceert.*

- *Het* neutraal element *is de identieke permutatie die geïnduceerd wordt door de identieke knooppermutatie. Elke graaf wordt dus op zichzelf afgebeeld.*

- *De inverse van een graafpermutatie is de graafpermutatie die geïnduceerd wordt door de inverse van de overeenkomstige knooppermutatie.*

- *De associativiteit is triviaal.*

Tot slot merken we op dat in dit geval $\mathcal{G} \subset Sym(A)$.

Definitie 2.22 (Permutatiegroep)**.** We noemen een groep $\mathcal{G}$ een *permutatiegroep* werkend op een verzameling A als geldt dat $\mathcal{G} \subseteq Sym(A)$ en $\mathcal{G}$ een groep is.

Neem nu een permutatiegroep die inwerkt op A. We definiëren nu een *orbit* van een element $a \in A$ ten opzichte van de groep $\mathcal{G}$ als volgt:

$$\mathcal{G}(a) = \{g(a) \mid g \in \mathcal{G}\} \subseteq A. \tag{2.11}$$

[3]Merk op dat het in sommige situaties misschien wél een goede notie van isomorfisme kan zijn.

Dit is de verzameling van alle elementen uit A waarop a kan afgebeeld worden door middel van een functie uit $\mathcal{G}$. In dit geval bevat onze groep alle mogelijke afbeeldingen, maar dit is niet altijd zo. Merk op dat a altijd in zijn eigen orbit zit, om wille van het neutrale element (in dit geval de identieke functie).

Voordat we verdergaan met een concrete toepassing van groepen, behandelen we nog het concept van een *stabilizer subgroep*. Een *stabilizer subgroep* voor een element $a \in A$ is een subgroep van $\mathcal{G}$, aangeduid met $\mathcal{G}_a$:

$$\mathcal{G}_a = \{g \in \mathcal{G} \mid g(a) = a\}. \tag{2.12}$$

De stabilizer geeft dus de functies (of permutaties) die een element op zichzelf afbeelden, of met andere woorden, de automorfismen van dat element. Merk op dat de identieke functie altijd in deze subgroep zit, maar dat dit niet noodzakelijk het enige element is. We noemen de stabilizer subgroep van een graaf ten opzichte van een bepaalde groep de *automorfismegroep* van de betreffende graaf.

Voorbeeld 2.23. *Als we deze definities toepassen op voorbeeld 2.21, dan bekomen we als orbit van een graaf G al de grafen die ermee isomorf zijn. De stabilizer geeft ons alle permutaties die een object op zichzelf afbeelden: de automorfismes.*

In voorbeeld 2.21 hebben we alle grafen met een vast aantal knopen beschouwd, maar in het algemeen willen we deze restrictie vermijden. Hoe kunnen we nu een permutatiegroep construeren die werkt op de verzameling met *alle* grafen $\mathbb{GS}$?

We definiëren de toepassing van een permutatie $\pi : \mathbb{GS} \to \mathbb{GS}$ op een graaf G als een element uit $S_1 \times S_2 \times \cdots$ (zie McKay [24]). Herinner dat S_i de verzameling van alle permutaties op de verzameling $\{1, 2, \ldots, i\}$ is. We beschouwen hier de permutaties op grafen zoals beschouwd in voorbeeld 2.21. Om het verschil in graad te overbruggen hebben we gebruik gemaakt van een permutatie die bestaat uit een oneindig aantal delen: een permutatie voor elke graad $n \in \mathbb{N}$:

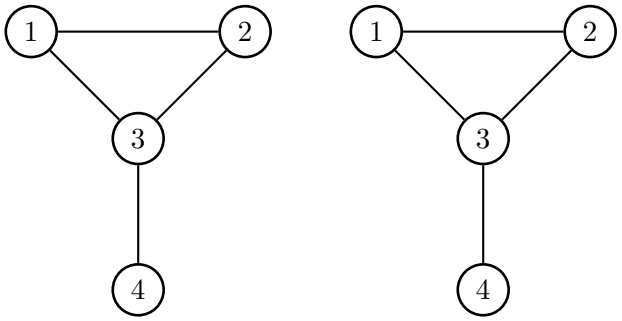
$$\pi = (\pi_1, \pi_2, \ldots),$$

waarbij $\pi_i \in S_i$. De inwerking van een dergelijke π op een graaf G van graad n is als volgt gedefinieerd:

$$\pi(G) = \pi_n(G), \text{ waarbij } o(G) = n.$$

We passen met andere woorden de "juiste" permutatie uit π toe op de graaf met graad n.

Op een analoge manier als in voorbeeld 2.21 kunnen we nu een groep $\mathcal{G}$ construeren die bestaat uit alle permutaties die isomorfe grafen op elkaar afbeelden. Ook hierbij kunnen we nu de orbit beschouwen, voor deze groep kunnen we dus iets meer algemeen zeggen dat:



(a) Een graaf G. **(b)** (1 2) toegepast op G

Figuur 2.5: Automorfismen.

Stelling 2.24. *Beschouw een permutatiegroep $\mathcal{G}$, werkend op $\mathbb{GS}$, die bestaat uit alle permutaties die isomorfe grafen om elkaar afbeelden, dan geldt: twee grafen zijn isomorf als en slechts als ze in elkaars orbit zitten.*

Als we de *stabilizer* berekenen van een $G \in \mathbb{GS}$, genoteerd $\mathcal{G}_G$, dan verkrijgen we alle permutaties die de graaf op zichzelf afbeelden. Met dit laatste bedoelen we dat de knopenverzameling behouden blijft, en dat de bogenverzameling behouden blijft, anders zijn de verkregen grafen immers niet gelijk. We verkrijgen dus, door de permutatie toe te passen, een graaf, waarbij de knopen nog steeds op dezelfde manier verbonden zijn: als knoop 1 met knoop 2 verbonden is, dan is dit na het toepassen van de permutatie ook nog steeds het geval. We noemen dit soort permutatie een *automorfisme van G*. Het zal in vele gevallen blijken dat niet enkel de identieke permutatie voor dit resultaat zorgt.

Voorbeeld 2.25. *In Figuur 2.5 zien we twee grafen. De ene graaf kan uit de andere bekomen worden door de permutatie* (1 2) *toe te passen. De graaf in Figuur 2.5a heeft als knopenverzameling $V = \{1, 2, 3, 4\}$ en als bogenverzameling $E = \{\{1, 2\}, \{1, 3\}, \{2, 3\}, \{3, 4\}\}$. De graaf in Figuur 2.5b heeft dezelfde knopenverzameling, en dezelfde bogenverzameling. Het is duidelijk dat de permutatie* (1 2) *de eerste, de tweede en de derde boog permuteert. Dit heeft echter als resultaat dat dezelfde bogenverzameling terug bekomen wordt (eenvoudig na te gaan). Hieruit mogen we besluiten dat we terug dezelfde graaf bekomen, en dat de permutatie* (1 2) *een automorfisme voor deze graaf is.*

2.4 Invarianten en Canonisatie

Het is niet eenvoudig om te testen of twee grafen isomorf zijn. In deze sectie gaan we dieper in op algoritmen die ons helpen bij het controleren van deze eigenschap. We beginnen met grafen waarop isomorfisme-tests bijna triviaal zijn: de geordende grafen. Hierna bekijken we twee soorten algoritmen die beslissen of twee grafen geordend zijn, hun orde buiten beschouwing gelaten. Deze algoritmen werden voorgesteld door Gurevich [17].

We merken op dat het in praktijk niet mogelijk is om aan een algoritme een ongeordende graaf mee te geven. Indien we willen controleren of twee ongeordende grafen isomorf zijn, moeten we ze ingeven met een zekere orde op de knopen. Het algoritme dient deze orde dus niet in rekening te brengen. We komen hier in de loop van de sectie nog op terug.

Definitie 2.26. Een geordende graaf is een geordend paar $(G, <)$, waarbij G een graaf is, bestaande uit knopen en bogen, en $<$ een orde op $V(G)$. We duiden de knopen (resp. bogen) van een geordende graaf $(G, <)$ aan met $V(G, <)$ (resp. $E(G, <)$), of gewoon met $V(G)$ en $E(G)$, wanneer het uit de context duidelijk is dat G geordend is.

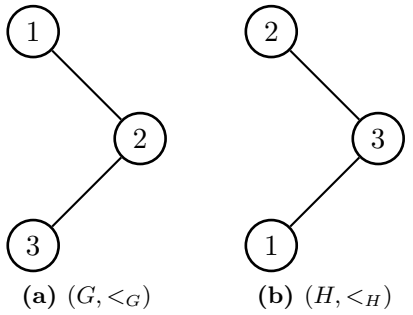
Op geordende grafen kunnen we vervolgens ook de notie van isomorfisme toepassen, zoals we eerder gedaan hebben bij "gewone" grafen. Hier komt er nog een nieuwe relatie bij: een orde op de knopen, die ook behouden moet blijven onder het isomorfisme. Na het toepassen van de definitie komen we tot volgend besluit:

Besluit 2.27 (Isomorfie van geordende grafen)**.** *Beschouw twee geordende grafen $(G, <_G)$ en $(H, <_H)$, we zeggen dat deze grafen (geordend) isomorf zijn met mekaar als en slechts als er een bijectie $\pi : V(G, <_G) \to V(H, <_H)$ bestaat zodat*

- $\forall x \in V(G, <_G) : \pi(x) \in V(H, <_H)$,
- $\forall \{x, y\} \in E(G, <_G) : \{\pi(x), \pi(y)\} \in E(H, <_H)$,
- $\forall x, y \in V(G, <_G) : x <_G y \Leftrightarrow \pi(x) <_H \pi(y)$.

Deze definitie van isomorfisme zegt ons dat twee geordende grafen isomorf zijn met mekaar wanneer de grafen isomorf zijn met mekaar, èn als de orde op de knopen bewaard blijft onder het isomorfisme.

Voorbeeld 2.28. *In Figuur 2.6 zijn twee verschillende ordes op de knopen van een graaf te zien. De graaf uit Figuur 2.6b kan bekomen worden door deze in Figuur 2.6a door de permutatie* (1 2 3) *toe te passen. Het is duidelijk dat de knooporde* $1, 2, 3$ *getransformeerd wordt naar* $2, 3, 1$*, hetgeen niet overeenkomt met de knooporde van de tweede graaf. Daar er geen andere permutaties mogelijk zijn tussen de twee grafen, mogen we besluiten dat ze*



(a) $(G, <_G)$ **(b)** $(H, <_H)$

Figuur 2.6: Geordend isomorfisme.

niet geordend *isomorf zijn met elkaar. Het is duidelijk dat ze wel* ongeordend *isomorf zijn, we hebben immers een permutatie gevonden die de ene graaf op de andere afbeeldt.*

Lemma 2.29. *Gegeven twee geordende grafen* $(G_1, <_1)$ *en* $(G_2, <_2)$*, dan geldt:*

$$(G_1, <_1) \cong (G_2, <_2) \Rightarrow G_1 \cong G_2.$$

Bovenstaand lemma zegt ons dat geordend isomorfisme ongeordend isomorfisme impliceert. Indien we dus weten dat twee geordende grafen isomorf zijn, dan zijn de grafen zonder de orde bijgevolg ook isomorf, hetgeen voor de hand ligt. We kunnen namelijk de permutatie tussen de geordende grafen gebruiken als permutatie tussen de ongeordende varianten.

Het begrip "ongeordende graaf" is nogal moeilijk te beschouwen, zoals we al eerder vermeld hebben. Formeel kunnen we zo een graaf zien als eentje waarbij geen orde op de knopen ligt, zoals bij de definitie van een "gewone" graaf. Verder is het mogelijk een dergelijke graaf te bekijken als als een verzameling van geordende grafen, die *ongeordend isomorf* zijn (de orbits), zoals McKay [24] doet. Met "ongeordend isomorf" bedoelen we dat er een isomorfisme tussen de geordende versies bestaat.

Verder zeggen we dat twee ongeordende grafen G en H (ongeordend) isomorf zijn indien er voor elke twee geordende grafen $(G, <_G)$ en $(H, <_H)$ een permutatie bestaat die de knopenverzameling en de bogenverameling op elkaar afbeeldt. Dit komt erop neer dat de derde voorwaarde uit besluit 2.27 wegvalt. Om te noteren dat grafen geordend isomorf zijn schrijven we $(G, <_G) \cong (H, <_H)$, en om aan te duiden dat ze ongeordend isomorf zijn $G \cong H$. Met andere woorden: twee grafen zijn (ongeordend) isomorf als we voor eender welke orde op hun knopen een bijectie tussen de grafen kunnen vinden, die de knooporde niet noodzakelijk bewaart.

Opmerking 2.30. Het is echter op algoritmisch niveau niet praktisch mogelijk om een graaf ongeordend voor te stellen, we moeten een algoritme altijd input geven die geordend is. We zullen voor elke ongeordende graaf bijgevolg een representant moeten kiezen. Het is duidelijk wanneer de representante grafen van twee ongeordende grafen ongeordend isomorf zijn, dat dan ook de ongeordende grafen isomorf zijn.

Voorbeeld 2.31. *De permutatie* (1 2 3) *beeldt de graaf uit Figuur 2.6a af op de graaf in Figuur 2.6b, zoals vermeld in voorbeeld 2.28. De orde wordt echter niet juist afgebeeld, waardoor de grafen niet geordend isomorf zijn. De grafen zijn wel ongeordend isomorf omdat de eis niet geldt voor ongeordend isomorfisme tussen twee grafen. Merk op dat de orde in beide grafen overeenkomt met* $\{(1,2),(1,3),(2,3)\}$*, hetgeen betekent dat* 1 *vóór* 2 *en* 3 *komt en* 2 *vóór* 3*. De permutatie maakt hiervan:* $\{(2,3),(2,1),(3,1)\}$.

Kunnen we nu een algoritme schrijven dat, gegeven twee geordende grafen, beslist of ze isomorf zijn met mekaar of niet? Om dit te kunnen construeren, gaan we gebruik maken van een encodering van onze graaf. Om tot een dergelijke encodering te komen, beschouwen we de *adjacentiematrix* van een graaf.

Definitie 2.32 (Adjacentiematrix)**.** De *adjacentiematrix* van een geordende graaf $(G,<)$ is een matrix M die bestaat uit $o(G)$ rijen en evenveel kolommen. Een positie (i,j) in de matrix M, waarbij $i,j \in [1,o(G)]$, heeft de waarde '1' als en slechts als er een boog $(v_i,v_j) \in E(G)$, waarbij v_k de k^{de} knoop is volgens de orde $<$ (we beginnen bij index 1). In alle andere gevallen staat er een '0' op die plaats.

Merk op dat we de bogenverzameling beschouwen als bestaande uit koppels van knopen, zoals bij een gerichte graaf. We kunnen dit als volgt op een ongerichte graaf toepassen: een positie (i,j) in de matrix M, waarbij $i,j \in [1,o(G)]$, heeft de waarde '1' als en slechts als er een boog $\{v_i,v_j\} \in E(G)$. Op deze manier verkrijgen we een matrix die symmetrisch is.

Voorbeeld 2.33. *De graaf in 2.1 kan voorgesteld worden met behulp van een adjacentiematrix die 4 rijen en 4 kolommen bevat (evenveel als de graad van G). De eerste rij en kolom komen overeen met knoop A, de tweede met B, de derde met C en de vierde met D:*

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

Indien we een adjacentiematrix voor een bepaalde geordende graaf hebben, kunnen we een encodering voor deze graaf beschouwen als zijnde de aaneenschakeling van de rijen van de matrix. We plakken met andere woorden de rijen van de matrix achter mekaar en bekomen een binaire string.

Let wel, we behouden de volgorde van de rijen. De functie die dit berekent, en als input een geordende graaf neemt, noemen we `Code`.

Stelling 2.34. *De `Code` van twee geordende grafen is identiek als en slechts als de twee grafen geordend isomorf zijn met elkaar.*

$$\texttt{Code}(G_1, <_1) = \texttt{Code}(G_2, <_2) \Leftrightarrow (G_1, <_1) \cong (G_2, <_2) \tag{2.13}$$

Bewijs. $\Rightarrow$ Als de `Code` van twee geordende grafen identiek is, dan wil dit zeggen dat hun matrices ook identiek zijn. Het proces dat de codes genereert kan namelijk ondubbelzinnig omgekeerd worden. We weten dus dat:

$$M_{G_1} = M_{G_2},$$

waarbij M_{G_i} de matrixvoorstelling is van de graaf G_i. Omdat de matrices identiek zijn, stellen ze exact dezelfde graaf voor (iedere matrixvoorstelling stelt slechts één graaf voor), dus

$$(G_1, <_1) = (G_2, <_2).$$

In het bijzonder geldt dus

$$(G_1, <_1) \cong (G_2, <_2).$$

$\Leftarrow$ Indien beide grafen slechts uit één of twee knopen bestaan, is het triviaal. We nemen aan dat de eigenschap geldt op grafen met n knopen of minder. Neem nu een graaf G met $n+1$ knopen. Omdat we hier geordend isomorfisme beschouwen, komen de overeenkomstige knopen op dezelfde plaats in de twee grafen. Als we dus de laatste knoop uit iedere graaf verwijderen, dan zijn de resulterende grafen nog steeds isomorf. We weten wegens de inductiehypothese dat de codes van deze grafen identiek zijn. Wegens besluit 2.27 weten we dat de knoop die we verwijderd hebben met exact dezelfde knopen verbonden is in beide grafen, wanneer we de orde beschouwen op de knopen. Omdat de resulterende matrix waaruit we de `Code` bouwen geordend is volgens de knopen, worden dus net dezelfde rijen en kolommen toegevoegd aan beide matrices. We verkrijgen dus twee identieke matrices voor de grafen, waardoor de `Code` ook identiek is. □

Voorbeeld 2.35. *Beschouw de matrix die we in voorbeeld 2.33 geconstrueerd hebben voor de graaf uit Figuur 2.1. Als we de rijen achter elkaar plakken bekomen we:*

$$0110101011010010,$$

hetgeen een binaire stringrepresentatie van de graaf is.

Opmerking 2.36. Het onderste triangulaire deel van de matrix is hetzelfde als het bovenste, en de diagonaal bestaat uit 0'en. Dit is omdat we een ongerichte graaf beschouwen zonder bogen van een knoop naar zichzelf. In principe hebben we dus enkel het bovenste triangulaire deel nodig om een code te bekomen die de graaf uniek voorstelt. Merk ook op dat we meerdere bogen tussen dezelfde knopen niet toelaten.

Voorbeeld 2.37. *De code die overeenstemt met de graaf G in Figuur 2.6a wordt als volgt berekend:*

$$\begin{pmatrix} 0 & 1 & 0 \\ 1 & 0 & 1 \\ 0 & 1 & 0 \end{pmatrix} \Rightarrow 010101010.$$

De code voor de graaf in Figuur 2.6a bekomen we door:

$$\begin{pmatrix} 0 & 0 & 1 \\ 0 & 0 & 1 \\ 0 & 1 & 1 \end{pmatrix} \Rightarrow 001001110.$$

De twee codes zijn verschillend, hetgeen erop wijst dat de grafen niet geordend isomorf zijn.

We hebben net bewezen dat door de `Code` van een geordende graaf te berekenen, we kunnen beslissen om hij geordend isomorf is met een andere graaf. Maar hoe zit het nu met gewone grafen, waar deze orde niet voorhanden is? Hoe kunnen we zeggen of twee *ongeordende grafen* isomorf zijn met elkaar? (Of equivalent: hoe beslissen we of twee geordende grafen *ongeordend* isomorf zijn?) Dit brengt ons bij de volgende definitie:

Definitie 2.38. Een *full-invariant algoritme* voor grafen is een algoritme I dat een binaire string toekent aan een geordende graaf zodat

$$I(G_1, <_1) = I(G_2, <_2) \Leftrightarrow G_1 \cong G_2 \tag{2.14}$$

Deze definitie zegt ons dat I een algoritme is dat op *geordende grafen* werkt, en dat we door de output te vergelijken kunnen beslissen of ze ongeordend isomorf zijn met elkaar. Een voorbeeld van zo een algoritme is eentje dat gegeven een graaf G met een bepaalde ordening, de kleinste $\texttt{Code}(G, <)$ berekent (door alle mogelijke alternatieve ordeningen af te lopen). Met "kleinste" code bedoelen we de minimale code ten opzichte van een zekere orde op de codes (bijvoorbeeld de lexicografishe orde). Het is intuïtief al duidelijk dat dit probleem een stuk ingewikkelder is dan beslissen of twee geordende grafen isomorf zijn: bij dit algoritme moeten namelijk zeer veel mogelijke knoopordes beschouwd worden (allemaal indien we dit niet intelligenter aanpakken).

Definitie 2.39. Een *graafcanonisatie algoritme* is een algoritme C dat een geordende graaf $(G, <_G)$ omzet in een andere, ongeordend isomorfe geordende graaf $(H, <_H)$. Het algoritme voldoet aan de volgende voorwaarden:

1. $C(G, <) = (G', <') \Rightarrow G' \cong G$,
2. $G_1 \cong G_2 \Rightarrow C(G_1, <_1) \cong C(G_2, <_2)$, voor willekeurige ordes $<_1, <_2$.

Een canonisatie-algoritme geeft volgens deze definitie een graaf terug die ongeordend isomorf is met de invoergraaf, hetgeen betekent dat er een nieuwe orde op de knopen kan gelegd zijn. Wanneer we dit algoritme toepassen op twee grafen die ongeordend isomorf zijn, verkrijgen we twee nieuwe grafen die geordend isomorf zijn.

Uit de twee eigenschappen van een canonisatie-algoritme, gegeven in de definitie, kunnen we afleiden dat ook hetvolgende geldt:

$$C(G_1, <_1) \cong C(G_2, <_2) \Rightarrow G_1 \cong G_2.$$

Hiermee bedoelen we dat wanneer de output van het algoritme voor twee grafen geordend isomorf is, dan zijn deze twee grafen ongeordend isomorf. We kunnen dit als volgt aantonen:
We weten dat

$$C(G_1, <_1) = (G'_1, <'_1) \cong (G'_2, <'_2) = C(G_2, <_2),$$

en omdat geordend isomorfisme ongeordend isomorfisme impliceert (lemma 2.29), mogen we concluderen dat

$$G'_1 \cong G'_2.$$

Wegens de eerste eigenschap van het canonisatie-algoritme weten we echter dat $G_1 \cong G'_1$ en $G_2 \cong G'_2$, waaruit volgt dat

$$G_1 \cong G'_1 \cong G'_2 \cong G_2.$$

In het bijzonder geldt dus

$$G_1 \cong G_2,$$

wegens de transitiviteit van de isomorfisme-relatie. Merk op dat hieruit volgt dat we ongeordend isomorfisme kunnen beslissen door te controleren of de uitvoer van het algoritme geordend isomorf is met een andere uitvoer. Geordend isomorfisme heel eenvoudig te controleren is, zoals we hierboven hebben besproken.

Het is duidelijk dat we uit een graafcanonisatie algoritme gemakkelijk een *full-invariant* algoritme kunnen construeren, we berekenen simpelweg de `Code` van de uitvoer. We bewijzen zo meteen dat dit een geldig *full-invariant* algoritme oplevert. We tonen ook aan dat de uitvoeringstijd van dit algoritme slechts polynomiaal veel hoger ligt dan deze van het graafcanonisatie

algoritme. Indien we dus een polynomiaal algoritme voor graafcanonisatie gevonden hebben, hebben we ook een polynomial *full-invariant* algoritme! De omgekeerde richting is niet op het eerste zicht duidelijk, maar het zal blijken dat ook dit geldt.

Definitie 2.40 (Full-invariant hypothese)**.** Er bestaat een *full-invariant* algoritme op geordende grafen dat een polynomiale uitvoeringstijd heeft.

Definitie 2.41 (Graafcanonisatie hypothese)**.** Er bestaat een graafcanonisatie algoritme op geordende grafen dat een polynomiale uitvoeringstijd heeft.

We tonen nu aan dat, indien één van deze hypothesen geldt, dat dan de andere ook geldt. Het is echter belangrijk om te weten dat het niet geweten is of één van deze hypothesen geldt.

Stelling 2.42. *De graafcanonisatie hypothese impliceert de full-invariant hypothese.*

Bewijs. We moeten aantonen dat indien een graaf canonisatie algoritme C bestaat, dat in polynomiale tijd loopt, dat er dan een full-invariant algoritme I bestaat dat ook in polynomiale tijd loopt.

We nemen aan dat we een canonisatie-algoritme C hebben. We construeren nu een algoritme I:

Algoritme 1: Polynomiaal invariant algoritme

Input: $(G, <)$
Output: $I(G, <)$
1 bereken $C(G, <) = (G', <')$;
2 **return** `Code`$(G', <')$;

Om te bewijzen dat dit algoritme werkt, tonen we eerst aan $\Rightarrow$ van vergelijking 2.14 aan. Gegeven twee geordende grafen $(G_1, <_1)$ en $(G_2, <_2)$, we gebruiken vervolgens het canonisatie algoritme C om $(G'_1, <'_1)$ en $(G'_2, <'_2)$ te berekenen. We weten nu dat

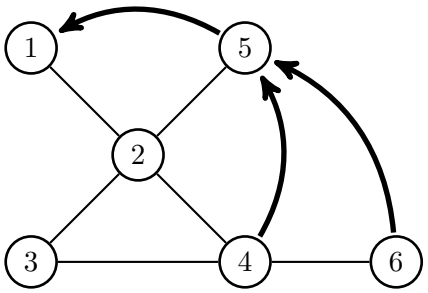
$$G_1 \cong G'_1 \text{ en } G_2 \cong G'_2, \tag{2.15}$$

wegens de definitie van een invariant algoritme. Vervolgens nemen we de `Code` van deze twee geordende grafen en bekomen twee binaire strings: `Code`$(G'_1, <'_1) = s_{G_1}$ en `Code`$(G'_2, <'_2) = s_{G_2}$. Indien $s_{G_1} = s_{G_2}$, weten we wegens vergelijking 2.13 dat

$$(G'_1, <'_1) \cong (G'_2, <'_2). \tag{2.16}$$

Dus in het bijzonder wegens 2.29 zijn de twee grafen G'_1 en G'_2 ook isomorf zonder de ordening, dus

$$G'_1 \cong G'_2. \tag{2.17}$$



Figuur 2.7: Een geëxpandeerde graaf.

Wegens de transitiviteit geldt

$$G_1 \cong G_1' \cong G_2' \cong G_2 \Rightarrow G_1 \cong G_2. \tag{2.18}$$

Vervolgens tonen we $\Leftarrow$ aan: Gegeven twee grafen G_1 en G_2 , dan moet gelden dat $G_1 \cong G_2 \Rightarrow I(G_1, <_1) = I(G_2, <_2)$. Als de grafen isomorf zijn, weten we dat $C(G_1, <_1) \cong C(G_2, <_2)$ wegens de eigenschappen van C. Wegens stelling 2.34 weten we ook dat dan noodzakelijk moet gelden dat $\texttt{Code}(C(G_1, <_1)) = \texttt{Code}(C(G_2, <_2))$.

Het is duidelijk dat het algoritme correct is, en dat het in polynomiale tijd loopt. □

We weten nu dat, indien we een (polynomiaal) canonisatie algoritme hebben, dat we dan ook een (polynomiaal) full-invariant algoritme hebben. We tonen vervolgens aan dat de omgekeerde implicatie ook geldt, waarvoor we gebruik maken van een *full-invariant* algoritme dat werkt om een speciaal soort graaf: een *geëxpandeerde graaf.*

Definitie 2.43. Een *geëxpandeerde graaf* is een graaf G, waaraan een extra binaire relatie op de knopen wordt toegevoegd.

Voorbeeld 2.44. *De graaf in Figuur 2.7 is "verhoogd" met een extra relatie op de knopen. De relatie is in dit geval gelijk aan*

$$\{(5,1),(4,5),(6,5)\},$$

en wordt aangeduid met de vette pijlen. Merk op dat deze verzameling bestaat uit koppels en niet uit verzameling, hetgeen een volgorde op de elementen legt.

Lemma 2.45. *Als er een full-invariant algoritme I_0 bestaat met als invoer een 'gewone' graaf, dat in polynomiale tijd loopt, dan bestaat er ook een full-invariant algoritme I. Dit algoritme I aanvaardt als invoer geëxpandeerde grafen en loopt eveneens in polynomiale tijd.*

Bewijs. Om het bestaan van dit algoritme aan te tonen, definiëren we een transformatie ζ die een geëxpandeerde graaf omzet in een gewone graaf. Met andere woorden, deze transformatie encodeert de informatie, die aanwezig is in de additionele relatie, in een nieuwe graaf. Bij deze encodering gaat de oorspronkelijke informatie niet verloren. Een eigenschap die ζ moet bezitten is de volgende:

$$(G_1, R_1) \cong (G_2, R_2) \Leftrightarrow \zeta(G_1, R_1) \cong \zeta(G_2, R_2), \tag{2.19}$$

waarbij R_1 (resp. R_2) de extra relatie op $V(G_1)$ (resp. $V(G_2)$) is. De gewenste I is dan $I(G, R) = I_0(\zeta(G, R))$, waarbij I_0 een full-invariant algoritme is dat op een 'gewone' graaf werkt.

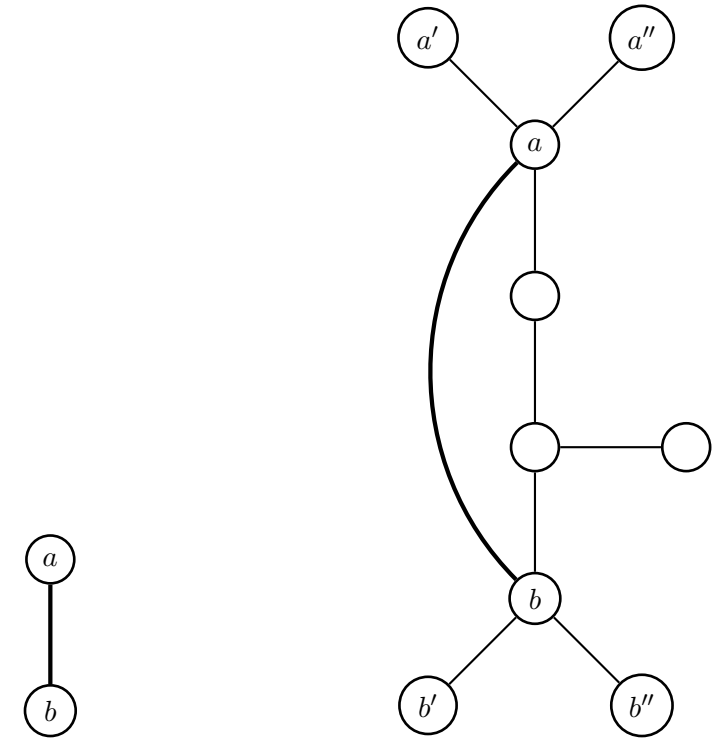
De constructie van ζ gebeurt als volgt: neem $H = \zeta(G, R)$. Voeg voor elk element $x \in V(G)$ twee nieuwe knopen x' en x'' in, en verbind deze met x. De nieuwe knopen hebben dus allemaal graad 1: $d(x') = d(x'') = 1$. Hierna voegen we nog een extra structuur tussen de twee knopen toe de een encodering is van de extra relatie ertussen (als deze knopen gerelateerd zijn). Voor elk $(a, b) \in R$ waartussen een boog in de graaf bestaat, zoals in Figuur 2.8a, bouwen we een nieuwe structuur zoals weergegeven in Figuur 2.8b. Merk op dat de boog in het vet de oorspronkelijke boog tussen de knopen is, indien deze niet aanwezig was in de graaf, maar (a, b) zat wel in de relatie, dan zou deze boog wegvallen, maar de rest behouden blijven.

Op deze manier hebben we een graaf H verkregen met $3n + 3r$ knopen, waarin de relatie R verwerkt is. We kunnen nog altijd de oorspronkelijke knopen terugvinden, omdat zij de enige zijn die twee buren van graad 1 hebben. Via deze constructie is er dus maar één representatie mogelijk voor (G, R). □

Voorbeeld 2.46. *In Figuur 2.9 is de graaf uit Figuur 2.7 te zien, na de toepassing van de functie ζ. De relatiepijlen uit de oude graaf zijn vervangen door de donkere knopen. Deze knopen hebben echter dezelfde betekenis als de andere knopen in de graaf, ze hebben slechts een ander uitzicht om aan te geven dat zij degenen zijn die toegevoegd zijn. Op deze manier gaat het overzicht namelijk niet verloren. Merk op dat we gemakkelijk kunnen zien welke knopen in de oorspronkelijke graaf zaten: degenen die twee buren van graaf één hebben. Het is duidelijk dat dit voor geen van de donkere knopen het geval is.*

Stelling 2.47. *De full-invariant hypothese impliceert de graaf canonisatie hypothese.*

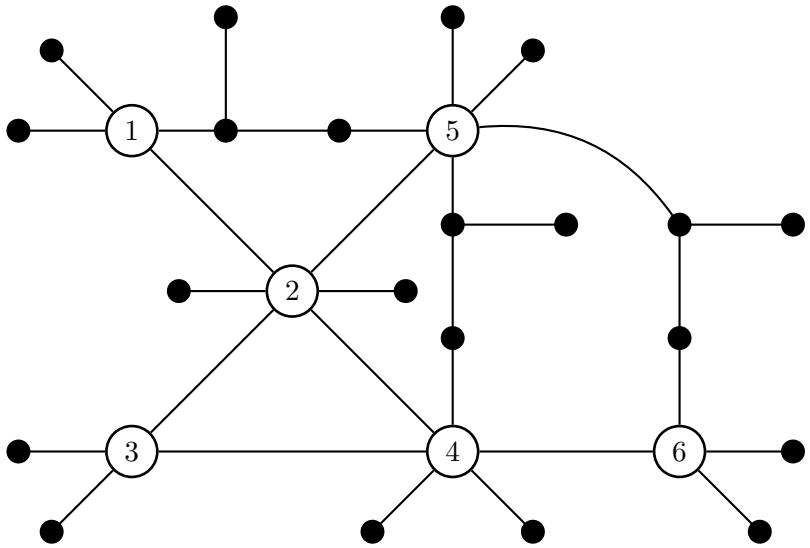
Bewijs. We beginnen met de veronderstelling dat er een full-invariant algorite I_0 bestaat dat uitvoert in polynomiale tijd en dat I een full-invariant algoritme is voor gexpandeerde grafen.



(a) een boog $\{a, b\}$ waarbij (a, b) in de relatie zit.

(b) de boog $\{a, b\}$ na encodering met ζ.

Figuur 2.8: Encoderen van een geëxpandeerde graaf.



Figuur 2.9: De geëxpandeerde graaf uit Figuur 2.7, omgezet met de functie ζ.

Notatie: Gegeven een knoop $v \in V(G)$, dan is $[v]$ de binaire relatie $\{(v, v)\}$. Gegeven een reeks van knopen $v_1, \ldots, v_k \in V(G)$ met $k > 1$, dan is $[v_1, \ldots, v_k]$ de partiële orde $\{(v_i, v_j) : 1 \leq i < j \leq k\}$ op de knopen van G. Merk op dat de orde slechts partieel is omdat niet noodzakelijk alle knopen van G in de orde worden opgenomen.

We construeren vervolgens een algoritme C: Gegeven een geordende graaf $(G, <_0)$ die n knopen bevat:

- Stap 1: bereken $I(G, [v])$ voor elke $v \in V(G)$ en ken de kleinst bekomen string (volgens de lexicografische orde) toe aan s_1. Neem vervolgens de kleinste (ten opzichte van de orde $<_0$) $v \in V(G)$ waarvoor geldt dat $I(G, [v]) = s_1$. We nemen dus de kleinste knoop die ons de kleinste string oplevert. Merk op dat er in het algemeen meerdere knopen kunnen zijn die aanleiding geven tot de kleinste string, omdat I dezelfde output geeft (moet geven) op isomorfe grafen. Omwille van dit probleem kiezen we de kleinste knoop.

- Stap k, met $1 < k < n$: We nemen aan dat er al $k-1$ knopen ontdekt zijn in de vorige stappen, namelijk $v_1, \ldots, v_{k-1}$. We berekenen nu $I(G, [v_1, \ldots, v_{k-1}, v])$ voor elke $v \in V(G)$ en $v \notin v_1, \ldots, v_{k-1}$. We kennen de kleinste string toe aan s_k en berekenen de kleinste $v \in V(G)$ en $v \notin v_1, \ldots, v_{k-1}$ ten opzichte van de orde $<_0$ waarvoor geldt dat $I(G, [v_1, \ldots, v_{k-1}, v]) = s_k$.

- Stap n: er zijn reeds $n-1$ knopen berekend, er blijft dus nog maar 1

knoop over: v_n. We kunnen nu het resultaat samenstellen:

$$C(G, <_0) = (G, [v_1, \ldots, v_n]).$$

Merk op dat $[v_1, \ldots, v_n]$ een orde op alle knopen legt, waardoor deze niet langer partieel is.

Het is duidelijk dat dit algoritme in polynomiale tijd loopt. We moeten nu enkel nog aantonen dat de twee eigenschappen uit definitie 2.39 gelden. Het is duidelijk dat $C(G, <) = (H, <') \Rightarrow G \cong H$, omdat we aan de graaf G niets veranderen. Er geldt met andere woorden dat $C(G, <) = (G, <')$, hetgeen aangeeft dat het algoritme enkel een nieuwe orde op de knopen legt.

De tweede noodzakelijke eigenschap tonen we aan met behulp van inductie op het aantal knopen. Neem

$$C(G, <_1) \;=\; (G, [v_1, \ldots, v_m])$$

$$C(H, <_2) \;=\; (H, [w_1, \ldots, w_n])$$

en veronderstel dat $G \cong H$, we laten vervolgens zien dat $C(G, <_1) \cong C(H, <_2)$ hieruit volgt.

We weten uit besluit 2.14 dat het aantal knopen van de grafen noodzakelijk gelijk moet zijn, omdat ze isomorf zijn, dus $m = n$. We zullen vanaf hier k gebruiken om naar dit aantal te verwijzen. We tonen nu aan, door middel van inductie op k, dat $(G, [v_1, \ldots, v_k]) \cong (H, [w_1, \ldots, w_k])$.

- basisgeval $k = 1$: We weten dat G en H isomorf zijn met mekaar, en dat er dus noodzakelijk een isomorfisme bestaat tussen de twee grafen. Neem zo een isomorfisme van G naar H en noem het f. We weten dat

  $$I(G, [v]) = I(H, [f(v)]), \tag{2.20}$$

  omdat f een knoop in H selecteert die dezelfde 'betekenis' heeft als v in G, waardoor de grafen geordend isomorf zijn. Vervolgens bekomen we:

  $$\min_{v \in V(G)} I(G, [v]) = \min_{v \in V(G)} I(G, \{(v, v)\}). \tag{2.21}$$

  Deze gelijkheid geldt omdat de minimale string die G voorstelt geproduceerd wordt door een bepaalde knoop. Deze gekozen knoop heeft een equivalente knoop in H die gegeven wordt door isomorfisme f, die wegens 2.20 dezelfde string moet opleveren. Dit gaat op voor elke knoop, waardoor knopen die niet-minimale strings voor G geven, een equivalente knoop onder f in H hebben die dus ook een niet-minimale string geeft. Omgekeerd geldt dit ook van H naar G, waaruit we kunnen concluderen dat de minima gelijk zijn.

We noemen de knoop die voor de graaf G (resp. H) in stap 1 wordt gekozen v_1 (resp. w_1). Samen met 2.21 weten we nu dat deze knopen hetzelfde minimum moeten geven. Indien dit niet zo was, dan zou er een equivalente knoop in de andere graaf bestaan, die een 'lagere' string zou opleveren. We besluiten dus $(G, [v_1]) \cong (H, [w_1])$ omdat I hetzelfde resultaat oplevert.

- Inductie: Veronderstel dat we voor een bepaalde $k < n$ al bewezen hebben dat er een isomorfisme

$$f : (G, [v_1, \ldots, v_k - 1]) \mapsto (H, [w_1, \ldots, w_k - 1])$$

bestaat. Dan weten we dat

$$I(G, [v_1, \ldots, v_k - 1, v]) = I(H, [w_1, \ldots, w_k - 1, f(v)])$$

voor alle $v \in V(G) - v_1, \ldots, v_k - 1$ en dus

$$\min\{s \mid \exists v \in V(G) - \{v_1, \ldots, v_k - 1\}, s = I(G, [v_1, \ldots, v_k - 1, v])\} =$$
$$\min\{s \mid \exists w \in V(H) - \{w_1, \ldots, w_k - 1\}, s = I(H, [w_1, \ldots, w_k - 1, w])\}.$$

De redenering hierachter is analoog aan het geval $k = 1$. We kunnen nu dus besluiten dat $(G, [v_1, \ldots, v_k]) \cong (H, [w_1, \ldots, w_k])$.

De inductie geeft aan dat we hiermee door kunnen gaan tot we alle knopen behandeld hebben, en dus een volledige orde krijgen op de n knopen. We bekomen dan $(G, [v_1, \ldots, v_m]) \cong (H, [w_1, \ldots, w_n])$, hetgeen we dienden te bewijzen.

□

Besluit 2.48. *Gegeven een polynomiaal full-invariant algoritme, dan weten we dat er een algoritme bestaat om de canonische versie van een graaf te berekenen, dat evenens loopt in polynomiale tijd. De omgekeerde implicatie is ook waar. We mogen met andere concluderen dat beide algoritmes "even complex" zijn. We kunnen namelijk altijd een algoritme construeren dat polynomiaal meer uitvoeringstijd vergt. Indien het originele algoritme een polynomiale of ergere uitvoeringstijd heeft, verergert deze kost niet in orde van grootte.*

2.4.1 Samenvatting

De twee soorten algoritmes doen dus in essentie hetzelfde: ze helpen ons met het beslissen of twee grafen isomorf zijn. Het full-invariant algoritme zorgt hiervoor door een bepaalde string te berekenen, die we met die van een andere graaf kunnen vergelijken om de isomorfie-test uit te voeren. Een graafcanonisatie algoritme geeft ons voor een graaf met eender welke orde

erop, een herordening van de knopen van de graaf. Deze herordening zal voor alle isomorfe grafen dezelfde zijn. De isomorfie-test op grafen reduceert bij het laatstgenoemde algoritme tot de zéér eenvoudige isomorfie-test voor geordende grafen. We hebben aangetoond dat deze algoritmen "even complex" zijn: indien er voor het één een polynomiaal algoritme bestaat, dan is dat ook het geval voor het andere.

Tot slot herhalen we op dat ieder algoritme, uitgevoerd op een computer, de knopen van de graaf in een bepaalde volgorde dient te verwerken. We kunnen niet simpelweg een ongeordende graaf meegeven als invoer voor een algoritme. Dit geeft aan dat er altijd een orde op de knopen ligt wanneer we algoritmen op grafen beschouwen. We kunnen dan we tussen dit soort grafen een *ongeordend isomorfisme* beschouwen, waaronder de orde niet bewaard dient te blijven.

In hetgeen volgt zullen we de term *canonische graaf* of *canonieke vorm* gebruiken om het resultaat van een algoritme C op een bepaalde graaf aan te duiden. De canonieke vorm geeft dus in essentie een nieuwe volgorde op de knopen van de graaf.

Hoofdstuk 3

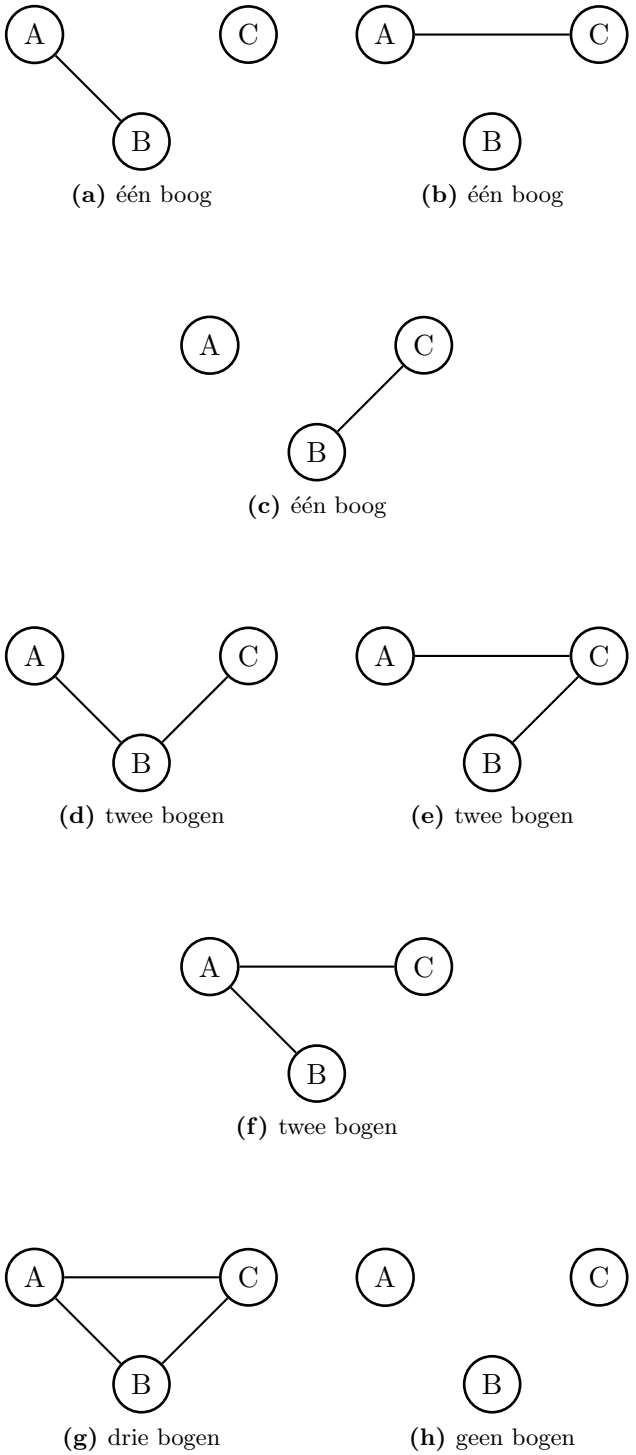
Generatie van grafen

3.1 Inleiding

In vele toepassingen hebben we een bepaalde klasse van grafen nodig die we allemaal in beschouwing moeten nemen in een algoritme. Een nadeel is echter dat de opslag van deze objecten, waarvan het aantal in de miljoenen of miljarden kan oplopen, niet meer efficiënt is. Het aantal grafen bijvoorbeeld, groeit zeer snel in functie van het aantal mogelijke knopen, zoals te zien is in Tabel 3.1[1]. We zien dat het aantal samenhangende ongerichte grafen (waarbij er een pad tussen elk paar knopen bestaat) iets minder snel groeit dan de "gewone" ongerichte grafen, maar nog steeds enorm snel. Merk op dat deze tabel de isomorfe kopieën van grafen niet in rekening brengt: bij de grafen op drie knopen bijvoorbeeld hebben we acht verschillende geordende grafen, en slechts vier ongeordende, zoals te zien is in Figuur 3.1. Het is duidelijk dat er veel meer grafen zijn wanneer we de isomorfe versies ook nog in beschouwing brengen. In het algemeen willen we deze isomorfe "kopieën" vermijden, omdat deze toch geen extra bijdrage leveren.

Een andere aanpak is om de grafen *at runtime* te genereren en ze vervolgens één voor één te beschouwen. Deze "klassieke" methode construeert uit een lijst met reeds gegenereerde grafen een nieuwe lijst. We kunnen bijvoorbeeld uit een lijst van alle grafen met q bogen een lijst bouwen die alle mogelijke grafen bevat met $q+1$ bogen. We duiden met $\mathcal{U}_q$ de verzameling grafen aan die q bogen bevatten. Indien we aan elk element van $\mathcal{U}_q$ op alle mogelijke manieren een boog toevoegen, dan bekomen we uiteindelijk alle grafen met $q+1$ bogen: de verzameling $\mathcal{U}_{q+1}$. Een nadeel van het toepassen van deze techniek is dat er ontzettend veel isomorfe grafen gegenereerd worden, waarin we eigenlijk geen interesse hebben. Als we bijvoorbeeld de graaf in Figuur 3.1h uitbreiden met één boog, op alle mogelijke manieren, bekomen we de drie isomorfe grafen uit figuren 3.1a, 3.1b en 3.1c. Om dit probleem aan te pakken is het noodzakelijk om alle reeds gegenereerde gra-

[1] bron: Zabrocki [38] en reeks A000088 van Sloane [32]



(a) één boog **(b)** één boog

(c) één boog

(d) twee bogen **(e)** twee bogen

(f) twee bogen

(g) drie bogen **(h)** geen bogen

Figuur 3.1: Alle grafen met drie knopen. Merk op dat er (in dit geval) eigenlijk maar één graaf is voor elk aantal bogen, de anderen zijn isomorf.

# knopen	# geconnecteerde grafen	# grafen
1	1	1
2	1	2
3	2	4
4	6	11
5	21	34
6	112	156
7	853	1044
8	11 117	12 346
9	261 080	274 668
10	11 716 571	12 005 168
11	1 006 700 565	1 018 997 864
12	164 059 830 476	165 091 172 592
13	50 335 907 869 219	50 502 031 367 952
14	29 003 487 462 848 061	29 054 155 657 235 488
15	31 397 381 142 761 241 960	31 426 485 969 804 308 768
16	63 969 560 113 225 176 176 277	64 001 015 704 527 557 894 928

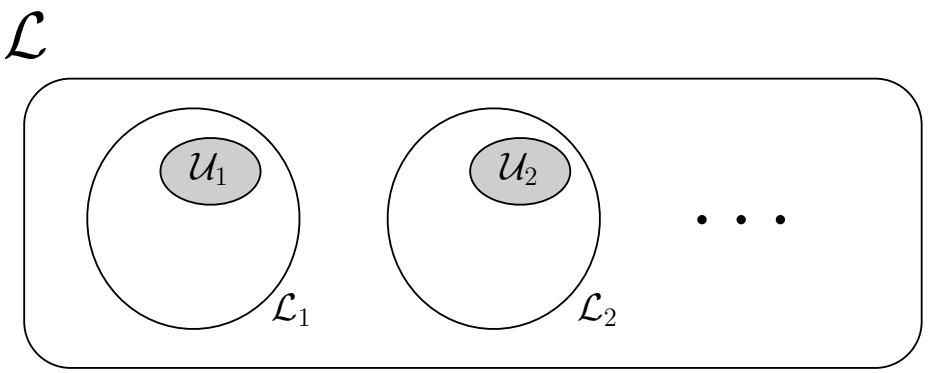
Tabel 3.1: Aantal grafen op n knopen.

fen van dezelfde grootte (herinner: het aantal bogen) te overlopen en te kijken of daarin al minstens één isomorfe kopie aanwezig is. Het spreekt voor zich dat dit ontzettend veel werk kost wanneer we grote verzamelingen grafen beschouwen. Het probleem verergert zelfs naarmate we meer geldige grafen ontdekken: we moeten steeds meer grafen op isomorfie gaan controleren! We kunnen natuurlijk wel optimalisaties toepassen in de isomorfietest, zoals naar het aantal knopen kijken, maar uiteindelijk zullen er toch nog een behoorlijk aantal dure controles uitgevoerd moeten worden. Een ander probleem is het feit dat we de hele lijst met grafen die we moeten doorzoeken niet meer in het geheugen kunnen houden als het algoritme op een computer uitgevoerd wordt. Hierdoor moeten we op een zeker ogenblik terugvallen op secundaire opslag, hetgeen bijzonder traag is in vergelijking met het geheugen.

We beschrijven nu enkele alternatieve generatieprocessen voor de "klassieke" methode:

- orderly generatie, door Read [26],
- gSpan, door Yan et al. [37], en
- Mckay's framework, door McKay [24].

Iedere aanpak heeft als doel de behoefte om de hele lijst met reeds gegenereerde objecten bij te houden overbodig te maken, door ervoor te zorgen dat we enkel naar de actieve graaf moeten kijken.



Figuur 3.2: Overzicht van orderly constructie.

3.2 Orderly generatie

Beschouw een verzameling van combinatorische objecten $\mathcal{L}$, waarvan de elementen ingedeeld kunnen worden aan de hand van een bepaalde eigenschap $\mathcal{P}$ (bijvoorbeeld het aantal knopen, het aantal bogen, ...), waarvan de waarde wordt aangegeven door de variabele q. We kunnen vervolgens subsets $\mathcal{L}_1, \mathcal{L}_2, \ldots$ construeren, waarbij $\mathcal{L}_q \subseteq \mathcal{L}$ en q de waarde van eigenschap $\mathcal{P}$ aangeeft. Ook hier kunnen we de notie van isomorfisme toepassen: isomorfe objecten zitten zeker in dezelfde $\mathcal{L}_i$. Merk op dat de eigenschap, op basis waarvan de objecten ingedeeld worden noodzakelijkerwijs invariant moet zijn onder isomorfisme, als we willen dat dit geldt. Indien dit niet het geval zou zijn, dan zou dit betekenen dat er isomorfe versies van eenzelfde object andere waarden voor eigenschap $\mathcal{P}$ kunnen hebben, waardoor ze in een andere verzameling terecht komen.

We willen nu, voor elke waarde i voor eigenschap $\mathcal{P}$ een lijst van objecten $\mathcal{U}_i$ construeren. Hierbij bevat elke $\mathcal{U}_i$ voor iedere isomorfismeklasse in $\mathcal{L}_i$ precies één object: het *canonische object* voor deze klasse. Een *isomorfismeklasse* is een verzameling waarin alle objecten zitten die onderling isomorf zijn. Het is duidelijk dat we hiervoor een canonisatie-algoritme kunnen gebruiken, dat ons voor elk object uit zo een klasse hetzelfde object teruggeeft. Het is dit canonisch object dat we in $\mathcal{U}_i$ zullen opnemen. De constructie van de verzamelingen is te zien in Figuur 3.2.

Vervolgens definiëren we een orde op de objecten: de orde $\prec$. We noteren $A \prec B$ als A in de orde vóór B komt en $A \preccurlyeq B$ als $A \prec B \vee A = B$.

Tot slot hebben we nog nood aan een operatie $\varphi : \mathcal{L}_q \to 2^{\mathcal{L}_{q+1}}$, die ons toelaat om uit een object, dat de waarde q voor eigenschap $\mathcal{P}$ bezit, objecten met waarde $q{+}1$ voor deze eigenschap te construeren. Deze operatie geeft ons de mogelijkheid om uit de verzameling $\mathcal{L}_q$ de verzameling $\mathcal{L}_{q+1}$ te bouwen. Merk op dat het hier nog steeds gaat over *alle* objecten, niet enkel

de canonische.

Met behulp van de voorgaande constructies zijn we in staat om een algoritme te bouwen dat, gegeven een verzameling van canonische objecten $\mathcal{U}_q$ de verzameling $\mathcal{U}_{q+1}$ bouwt:

Algoritme 3: Orderly algoritme

```
   Input: U_q
   Output: U_{q+1}
 1 U_{q+1} = ∅;
 2 sorteer U_q volgens ≺;
 3 foreach X ∈ U_q do
 4 |   A = φ(X);
 5 |   foreach Y ∈ A do
 6 |   |   if last(U_{q+1}) ≺ y ∧ canon(Y) == Y then
 7 |   |   |   voeg Y achteraan toe aan U_{q+1};
 8 |   |   end
 9 |   end
10 end
11 output U_{q+1};
```

De nieuwe lijst $\mathcal{U}_{q+1}$ wordt opgebouwd door objecten uit $\mathcal{U}_q$ in volgorde te doorlopen, ze één voor één uit te breiden en vervolgens te controleren of ze wel toegevoegd mogen worden. In deze laatste test wordt naast een ordecontrole ook nagegaan of het object wel canonisch is. Merk op dat de volgorde van deze test veel uitmaakt: het testen van de orde is in het algemeen veel goedkoper dan de canoniciteitscontrole: Indien het principe van *lazy evaluation*[2] wordt toegepast, dan zal indien de orde-test faalt de canoniciteitstest zelfs niet meer berekend moeten worden!

Afhankelijk van de keuzes van canonische vorm, orde en de groeioperatie φ kan het algoritme efficiënter zijn. We zullen nu de drie voorwaarden bespreken die moeten gelden opdat het orderly algoritme correct werkt.

Voorwaarde 3.1 (Canonische generatie)**.** *Ieder canonisch object uit $\mathcal{L}_{q+1}$ moet geproduceerd kunnen worden door φ toe te passen op ten minste één canonisch object uit $\mathcal{L}_q$.*

Bewijs. Indien het niet het geval is dat er een canonisch object in $\mathcal{L}_q$ bestaat dat de voorwaarde vervult, dan is er ten minste één canonisch object in $\mathcal{L}_{q+1}$ dat niet gegenereerd kan worden uit een object uit $\mathcal{U}_q$. Het object zal niet door het algoritme gegenereerd worden en komt dus niet terecht in $\mathcal{U}_{q+1}$. We kunnen er nu op geen enkele manier meer voor zorgen dat het algoritme de volledige verzameling $\mathcal{U}_{q+1}$ genereert. □

Om de volgende voorwaarde te kunnen formuleren hebben we een functie $f : \mathcal{U}_{q+1} \to \mathcal{U}_q$ nodig. Deze functie beeldt een object uit $\mathcal{U}_{q+1}$ af op het eerste

[2] indien de eerste component van de AND-test false geeft, wordt de tweede niet meer berekend, het resultaat ligt namelijk al vast (false).

(volgens de orde $\prec$) canonisch object uit $\mathcal{U}_q$ dat dit genereert.

Voorwaarde 3.2 (Monotoniciteit)**.** *f is zwak monotoon. Hiermee bedoelen we dat voor elke twee objecten $X, Y \in \mathcal{U}_{q+1}$ geldt:*

$$X \prec Y \Rightarrow f(X) \preccurlyeq f(Y).$$

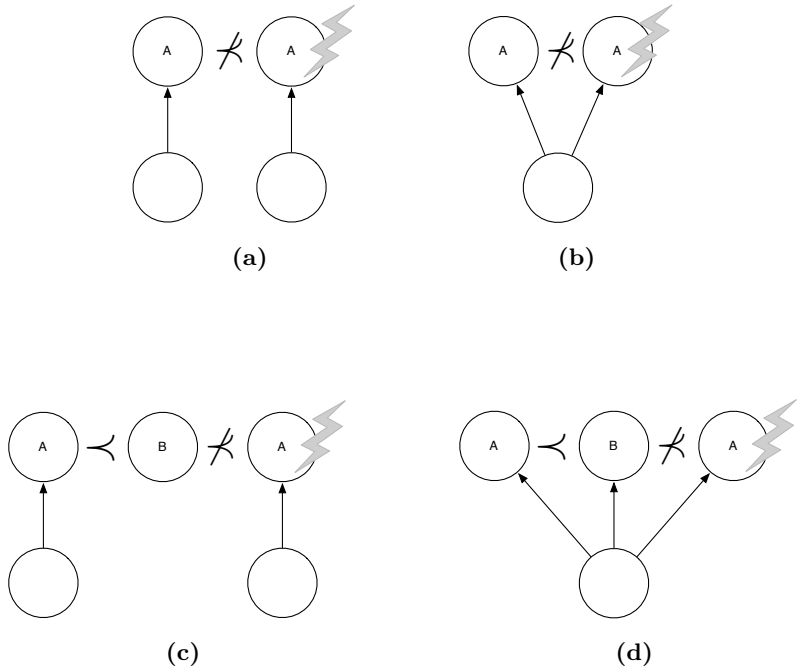
Bewijs. Als f niet zwak monotoon is, dan bestaan er een twee elementen $X, Y \in \mathcal{U}_{q+1}$, zodat $X \prec Y$ en $f(X) \succ f(Y)$. Omdat we $\mathcal{U}_q$ in volgorde doorlopen, wordt $f(X)$ dus later dan $f(Y)$ uitgebreid, dus X wordt later "ontdekt" dan Y. Omdat bij de generatie van X, Y reeds beschouwd is, weten we dat het laatste element in $\mathcal{U}_{q+1} \succcurlyeq Y$. Verder weten we ook dat $X \prec Y$ en dat X bijgevolg niet meer aan $\mathcal{U}_{q+1}$ kan worden toegevoegd, omdat $\mathcal{U}_{q+1}$ enkel grotere elementen aanvaardt dan deze reeds bevat. De toevoeging van Y vóór X zorgt er met andere woorden voor dat de toevoeging van X verhinderd wordt. Dit geeft de noodzaak van voorwaarde 3.2 weer. □

Voorwaarde 3.2 zorgt ervoor dat de canonische objecten in de juiste volgorde gegenereerd moeten worden, zodat ze allemaal de kans krijgen om aan de lijst toegevoegd te worden. Merk op dat het mogelijk is dat eenzelfde element verschillende objecten genereert, en dat de volgorde hiervan niet vastligt. Het is dus perfect mogelijk dat we eenzelfde scenario krijgen bij het toepassen van φ op slechts één object. Ook hier kan de toevoeging van bepaalde objecten geblokkeerd worden door anderen. We moeten er dus voor zorgen dat de kleinste (ten opzichte van $\prec$) objecten eerst gegenereerd worden. We realiseren dit door een extra beperking op φ te leggen die deze volgorde garandeert:

Voorwaarde 3.3. *Het resultaat van de operatie φ, toegepast op een element uit $\mathcal{U}_q$, geeft als resultaat een lijst met objecten uit $\mathcal{L}_{q+1}$, geordend volgens $\prec$.*

Deze laatste voorwaarde zorgt ervoor dat de zojuist vermelde situatie niet kan voorkomen. Merk op dat objecten die reeds in de lijst zaten niet volgens deze volgorde gegenereerd moeten worden, omdat ze toch verworpen zullen worden. Dit geeft aan dat voorwaarde 3.3 strenger is dan noodzakelijk. Het zal echter in het algemeen meer werk vragen om deze objecten afzonderlijk te gaan behandelen dan om ze allemaal te sorteren. Het algoritme geeft ook aan dat de objecten van het volgende niveau in volgorde gegenereerd worden, hetgeen zorgt dat deze objecten niet meer gesorteerd moeten worden.

Tot slot vermelden we nog dat wanneer een canonisch object gegenereerd wordt door meerdere objecten uit $\mathcal{U}_q$, dat alles behalve het eerste dan automatisch verworpen wordt door het algoritme. Indien er geen objecten tussen deze twee identieke nieuwe objecten (X en X') gegenereerd worden



Figuur 3.3: Illustratie van orderly: het meermaals genereren van eenzelfde canonische object.

(Figuur 3.3a), dan weten we dat $X \not\prec X'$, waardoor, indien X al is toegevoegd, X' geweigerd wordt. Het is duidelijk dat dit zowel geldt wanneer $f(X) \prec f(X')$ als wanneer $f(X) = f(X')$ (zelfde ouder, zie Figuur 3.3b), wegens de zojuist besproken voorwaarden. Wanneer er wel objecten tussen gegenereerd worden, dan kan de waarde (met betrekking tot de orde) van het laatste element uit de $\mathcal{U}_{q+1}$ enkel maar groeien, zodat het identieke object opnieuw geweigerd wordt. Dit is te zien in Figuur 3.3c en 3.3d.

Voorbeeld 3.4. *Als voorbeeld bouwen we een orderly algoritme waarmee we alle gerichte (!) grafen met vijf knopen genereren. We kiezen voor de voorstelling van een graaf de matrixrepresentatie. We nemen als $\mathcal{L}$ de verzameling van alle gerichte grafen met vijf knopen en als $\mathcal{L}_q \subset \mathcal{L}$ de deelverzameling met q bogen. De eigenschap $\mathcal{P}$ is hier: "Het aantal bogen van de graaf". We definiëren de* `Code` *van een graaf G als de aaneenschakeling van de rijen van de matrix die overeenkomt met G, zoals in hoofdstuk 2. De canonische graaf definiëren we als de isomorfe graaf met de grootste* `Code`*. De orde die op de codes ligt is in dit geval de lexicografische orde. De uitbreidingsfunctie φ verandert in dit geval één '0' in de matrix in een '1'. We*

doen deze vervanging van achter naar voor, omdat we eerst de grafen met een kleinere code willen genereren. Merk op dat er op de diagonaal van de matrix geen veranderingen dienen te gebeuren, omdat we self-loops *willen vermijden.*

De code van de graaf in Figuur 3.4a wordt als volgt berekend:

$$\begin{pmatrix} 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix} \Rightarrow 01000\ 00100\ 00011\ 00000\ 00000.$$

Wanneer we nu de uitbreidingsfunctie φ toepassen, bekomen we als eerste

$$01000\ 00100\ 00011\ 00000\ 00001,$$

hetgeen een getal op de diagonaal van de matrix verandert en dus niet geldig is. Vervolgens bekomen we

$$01000\ 00100\ 00011\ 00000\ 00010,$$

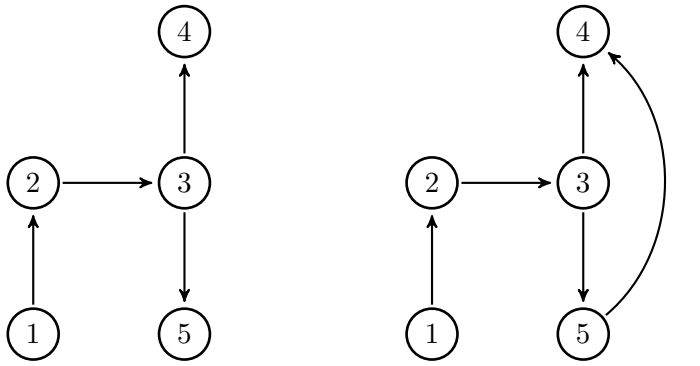
hetgeen overeenkomt met de matrix

$$\begin{pmatrix} 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \end{pmatrix},$$

waarvan de overeenkomstige graaf te zien is in Figuur 3.4b.

Het is duidelijk dat een canoniciteitscontrole veel intensieve berekeningen vergt, maar dat deze in dit geval performanter is dan in het klassieke algoritme. In de klassieke setting dient elke nieuwe graaf namelijk vergeleken te worden (op isomorfie getest) met *alle* oude objecten, en volgens Gurevich [17] is het berekenen van een canonische versie van een graaf "even moeilijk" als een isomomorfietest, zoals we hebben aangetoond in sectie 2.4. Er worden in de klassieke setting bijgevolg veel meer "zware controles" uitgevoerd dan bij het orderly algoritme.

Zoals eerder vermeld kunnen we de canoniciteitscontrole in het algoritme uitstellen tot na de ordetest, hetgeen in het geval van weigering een dure canonicteitscontrole uitspaart. Verder dienen we enkel het laatste object uit $\mathcal{U}_{q+1}$ bij te houden, omdat we enkel het laatst toegevoegde element nodig hebben om de ordetest uit te voeren. Deze laatste eigenschap haalt de geheugenvereisten enorm omlaag (we moeten geen lijsten meer bijhouden). Merk wel op dat, indien we $\mathcal{U}_{q+2}$ willen berekenen, we de volledige lijst $\mathcal{U}_{q+1}$ opnieuw in het geheugen nodig hebben. Hierop komen we later nog terug.



(a) Een gerichte graaf met 5 knopen.

(b) Een nieuwe gerichte graaf na het toevoegen van een boog.

Figuur 3.4: Het groeiproces van gerichte grafen op 5 knopen.

3.2.1 Een generisch algoritme

We construeren nu een algemeen orderly algoritme, dat toelaat om allerlei soorten combinatorische objecten te genereren: een *generisch orderly algoritme*. We vullen één voor één de drie nodige voorwaarden in.

Neem voor $\mathcal{L}$ de verzameling van alle vectoren van dimensie n, waarbij de componenten van de vectoren gekozen worden uit de verzameling $\{0, 1, 2, \ldots, k-1\}$, voor een zekere k. Voor $\mathcal{L}_q$ nemen we de deelverzameling van $\mathcal{L}$ die enkel vectoren bevat waarvoor geldt dat de som van de componenten gelijk is aan q:

$$v = (v_1, \ldots, v_n) \in \mathcal{L}_q \Leftrightarrow q = \sum_{i=1}^{n} v_i, \text{ waarbij } v_1, \ldots, v_n \in \{0, \ldots, k-1\}.$$

Vervolgens nemen we een permutatiegroep $\mathcal{G}$ voor $\mathcal{L}$, hierbij werken de permutaties op de indexen van de componenten van de vector. We kunnen een permutatie van een vector dus zien als een permutatie van de verzameling $\{1, \ldots, n\}$. Merk op dat $\mathcal{G}$ niet noodzakelijk alle permutaties bevat tussen de vectoren, maar evengoed een deel hiervan kan zijn, dat nog steeds aan de voorwaarden van een groep voldoet. Dit zal later verduidelijkt worden. De `Code` van een vector v is de getalvoorstelling in het k-delig stelsel, bekomen door de componenten allen achter elkaar te plaatsen.

Voorbeeld 3.5. *Neem* $k = 3$ *en* $n = 5$

$$v = (1, 0, 2, 0, 2),$$

dan is

$$\texttt{Code}(v) = 10202_3.$$

De *canonische* vector van een vector v definiëren we als de isomorfe vector (volgens $\mathcal{G}$) met de grootste `Code`. De *orde* op de codes wordt hierbij gegeven door de omgekeerde standaard orde op de natuurlijke getallen: $a \prec b \Leftrightarrow a > b$, waarbij a en b codes zijn. Voor de *uitbreidingsoperatie* φ nemen we de functie die de volgende lijst van objecten teruggeeft wanneer deze toegepast wordt op een vector v:

- De vector bekomen door de laatste niet-nul component van v te verhogen met '1', indien deze kleiner is dan $k - 1$.
- De vectoren bekomen door alle nullen na de laatste niet-nul component om de beurt met '1' te verhogen. (van voor naar achter)

Indien de vector enkel uit 0'en bestaat, dan slaan we de eerste stap over en beginnen we van voor naar achter met het vervangen van een '0' door een '1'. Merk op dat φ ons een lijst teruggeeft van vectoren die geordend is volgens $\prec$, zoals geïllustreerd wordt in hetvolgende voorbeeld.

Voorbeeld 3.6. *Neem voor $k = 4$ en voor $n = 7$, en beschouw de vector $v = (3, 1, 0, 2, 0, 0)$, dan verkrijgen we door toepassing van de zojuist gedefiniëerde φ:*

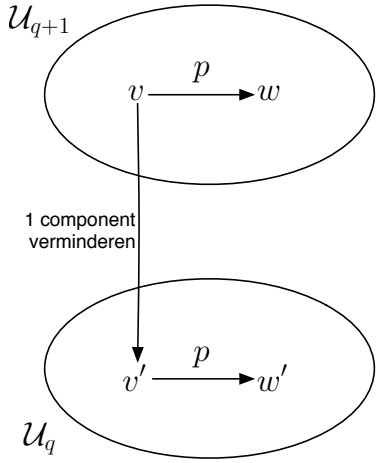
$$(3, 1, 0, 3, 0, 0),$$
$$(3, 1, 0, 2, 1, 0),$$
$$(3, 1, 0, 2, 0, 1).$$

De uitbreidingsfunctie φ maakt een nieuwe laatste niet-0 component. Het omgekeerde van de uitbreidingsfunctie is de laatste niet-0 component aanpassen. Hierbij is duidelijk dat $310300_4 > 310210_4 > 310201_4$, waarbij $<$ de orde op de natuurlijke getallen is. Hieruit volgt dat $(3, 1, 0, 3, 0, 0) \prec (3, 1, 0, 2, 1, 0) \prec (3, 1, 0, 2, 0, 1)$ (herinner: de orde is van klein naar groot, de grotere getallen komen dus eerst). De vectoren zijn in de juiste volgorde gegenereerd.

Het is duidelijk dat aan voorwaarde 3.3 voldaan is, door de manier waarop we φ geconstrueerd hebben. We bewijzen vervolgens de twee andere voorwaarden, om aan te tonen dat het orderly algoritme correct werkt met deze constructie.

Stelling 3.7. *Neem v een canonische vector uit $\mathcal{L}_{q+1}$, verlaag het laatste niet-nul element met 1 om een vector v' te bekomen, dan is $v' \in \mathcal{U}_q$. Met andere woorden er bestaat een canoniche vector v' die v genereert door er φ op toe te passen.*

Bewijs. We kunnen eenvoudig inzien dat er canonische vectoren in $\mathcal{U}_q$ zitten die in slechts één component verschillen van een vector w, die isomorf is met v. Als we namelijk in de vector v één component verminderen met '1' om een vector v' te bekomen, en we nemen hier de canonische versie van, dan beschouwen we de permutatie p om uit v' deze canonische versie te verkrijgen. Pas nu eerst p toe op v om te komen tot een vector w. We kunnen nu gemakkelijk inzien dat we in w slechts één component dienen te verminderen met '1' om de canonische versie van vector v' te bekomen: indien we in v component i hebben verminderd, dan verminderen we de j^{de} component van w, waarbij j het beeld van i onder p is. Merk op dat w niet noodzakelijk canonisch is! Een overzicht van de redenering is gegeven in Figuur 3.5.



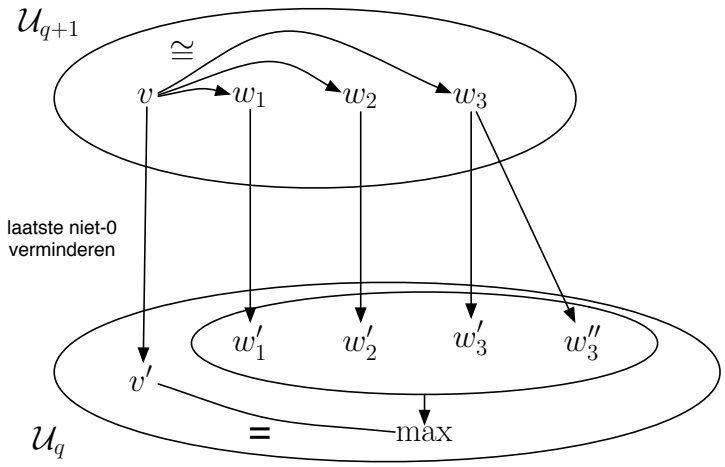
Figuur 3.5: Illustratie van de vector-constructie.

Neem nu uit al deze vectoren uit $\mathcal{U}_q$ (die in slechts één component verschillen van mogelijke w's) degene met de grootste `Code`, met andere woorden degene die het eerst in de lijst $\mathcal{U}_q$ voorkomt, en noem deze vector w'. We zullen vervolgens aantonen dat deze $w' = v'$, en v' met andere woorden canonisch is, waarbij v' bekomen wordt door de laatste (!) niet-nul component van v met '1' te verminderen. Een overzicht wordt gegeven in Figuur 3.6. Merk op dat in de figuur aangegeven wordt dat een vector meerdere vectoren geassocieerd kunnen worden met een vector w_i.

Wegens de constructie van φ en het feit dat w gegenereerd wordt door w', weten we dat

$$\texttt{Code}(w') < \texttt{Code}(w) \leq \texttt{Code}(v).$$

Merk op dat we tussen de codes $<$ gebruiken, omdat de orde op de natuurlijke getallen beschouwd wordt.



Figuur 3.6: Illustratie van de canonisatie.

We vergelijken vervolgens w' met v:

$$\begin{aligned} w' &= (w'_1, \ldots, w'_i, \ldots, w'_j, \ldots, w'_n), \\ v &= (v_1, \ldots, v_i, \ldots, v_j, \ldots, v_n), \end{aligned} \tag{3.1}$$

hierbij is i de kleinste positie l waar $v_l > w'_l$ en j de grootste. Merk op dat het niet mogelijk dat er een positie h bestaat zodat $h \leq i$ en $v_h < w'_l$, omdat dan de code van v kleiner zou zijn dan die van w' en daardoor vergelijking 3.2.1 overtreden wordt. Voor de posities $h > j$ geldt dat $v_h \leq w'_h$.

Als $v_i > w'_i + 1$, neem dan voor x de vector bekomen uit v door van v_i '1' af te trekken en noem x_c zijn canonische vector. Dan geldt er dat

$$\texttt{Code}(w') < \texttt{Code}(x) \leq \texttt{Code}(x_c), \tag{3.2}$$

omdat $x_i > w'_i$ en alles daarvoor identiek is. We weten vervolgens dat x slechts op één plaats verschilt van v, dus logischer wijze verschilt x_c slechts op één plaats met een isomorf van v. We hebben in het begin echter w' zo gekozen dat deze de grootste is, en nu blijkt dat x_c groter is, hetgeen een tegenstrijdigheid oplevert. We besluiten dus dat $v_i = w_i + 1$.

Veronderstel dat $j > i$, we construeren een vector x uit v, waarbij $x_j = v_j - 1$. Het is duidelijk dat $\texttt{Code}(x)$ opnieuw groter is dan $\texttt{Code}(w')$, waardoor vergelijking 3.2 opnieuw geldt, en we op dezelfde wijze een contradictie bekomen. Het is dus toegestaan om te besluiten dat $i = j$, hetgeen ons zegt dat v en w slechts verschillen op positie i. Indien na positie i nog verschillen zouden opduiken, zouden dat plaatsen zijn waar de waarden van v kleiner zijn dan die van w' op de overeenkomstige posities (zie eerder). Dit

is onmogelijk, omdat we dan nooit meer aan de som $q+1$ kunnen komen zonder componenten te verhogen. Deze verhoging is verboden, omdat j de laatste positie was waar v de waarde van w overtrof.

Tot nu toe hebben we dus voor de vectoren v en w een aantal beperkingen ontdekt, die ons aangeven dat ze van de volgende vorm zijn:

$$\begin{aligned} w' &= (w'_1, \ldots, w'_i, \ldots, b, 00\cdots0), \\ v &= (v_1, \ldots, w'i+1, \ldots, b, 00\cdots0). \end{aligned} \tag{3.3}$$

In deze vegelijking merken we op dat de waarde van v op positie i één hoger is dan die van w en dat alles wat hierna en hiervoor komt identiek is. Veronderstel nu dat het laatste niet-nul component, aangeduid met b, na component i komt. Construeer x op de volgende manier:

$$x = (w'_1, \ldots, w'_i+1, \ldots, b-1, 00\cdots0). \tag{3.4}$$

Opnieuw geldt vergelijking 3.2, hetgeen opnieuw een contradictie oplevert. We besluiten nu dat ze enkel verschillen met '1' op de laatste niet-nul positie i en mogen concluderen dat w' de canonische ouder van v is. □

Stelling 3.8. *De functie f, die voor een vector v de kleinste vector geeft die v genereert, is zwak monotoon.*

Bewijs. Beschouw twee vectoren $v, w \in \mathcal{U}_{q+1}$, waarbij $v \prec w$, en dus ook $\texttt{Code}(v) > \texttt{Code}(w)$. Als we de componenten van v en w van voor naar achter vergelijken, dan nemen we i als de kleinste positie waarin de twee vectoren verschillen. Er geldt (wegens de aanname $v \prec w$ en $\texttt{Code}(v) > \texttt{Code}(w)$) dat $v_i > w_i$. We kunnen onmiddellijk afleiden dat er in w na positie i noodzakelijkerwijs nog een niet-nul component moet volgen, omdat het anders niet mogelijk is om nog aan de som $q+1$ te komen; de kleinere i-component moet immers gecompenseerd worden ($v, w \in \mathcal{U}_{q+1}$). We kunnen twee gevallen onderscheiden;

Geval 1 v_i is niet de laatste niet-nul component van v

Als we zowel in v als w de laatste component die niet gelijk is aan een '0' verminderen met '1', dan bekomen we hun ouders. Omdat component i ongewijzigd is gebleven, en alles ervoor ook, is het duidelijk dat voor de resulterende vectoren $v' = f(v)$ en $w' = f(w)$ geldt dat $v' \prec w'$ (of $f(v) \prec f(w)$).

Geval 2 v_i is de laatste niet-nul component van v.

Als $v_i > w_i+1$, dan geldt $v_i-1 \geq w_i+1$, omdat w_i niet verandert. We kunnen nu een analoge redenering maken. Als daarentegen $v_i = w_i+1$

dan weten we dat v' in de eerste i componenten overeenstemt met w. Omdat w enkel de '1' dient te compenseren is het dus noodzakelijk dat er slechts één niet-nul component volgt dat gelijk is aan '1' (negatieve componenten zijn niet toegestaan). Als we deze component verminderen met '1', bekomen we w', een vector die identiek is aan v. Er geldt dus $f(v) = f(w)$, hetgeen geldig is omdat f een *zwakke* monotone functie dient te zijn, hetgeen hiermee bewezen is.

□

Dit algemene probleem kunnen we toepassen op enkele specifieke problemen. Zo is er de generatie van gerichte grafen op vijf knopen (zonder self-loops en multi-edges), die eerder al werd besproken. We kunnen de `Code` van zo een gerichte graaf als een 20-bit string nemen (25 matrix entries waarvan de 5 op de diagonaal altijd 0 zijn). Toegepast in de zojuist beschreven constructie geeft dit ons $k = 2$ en $n = 20$. De uitbreidingsfunctie blijft dezelfde en zal, in tegenstelling tot onze eerste poging, veel minder 0'en in 1'en moeten veranderen (zie voorbeeld 3.4, daar moesten *alle* 0'en om de beurt vervangen worden). Een analoge constructie is ook toepasbaar voor gewone grafen, waar we enkel het bovenste triangulaire deel van de matrix nodig hebben.

Om alle (ongerichte, ongelabelde) grafen te generen met m knopen (m is constant), bouwen we een vector-code op, zodat we het hierboven beschreven algoritme kunnen toepassen. We nemen voor $\mathcal{L}$ de verzameling van alle ongrichte grafen op m knopen en voor $\mathcal{L}_q$ de verzameling van alle ongerichte grafen met m knopen en q bogen. We dienen nu nog een grootte van de vector de definiëren (n), een waarde voor k, een code voor de grafen, en een permutatiegroep $\mathcal{G}$.

Herinner dat een graaf voorgesteld kan worden met behulp van een $m \times m$-matrix en dat we enkel het bovenste triangulaire deel ervan nodig hebben om een representatieve string voor de graaf te construeren. We verkrijgen dus een vector van de vorm

$$((1,2),\ldots(1,m),(2,3),\ldots,(2,m),\ldots(m-1,m))\,.$$

In deze vector is (i,j) de waarde in de matrix op rij i, kolom j. Het aantal elementen in de vector is gelijk aan $\binom{m}{2}$, met andere woorden $n = \binom{m}{2}$ de constructie van de vectoren. Omdat we ongelabelde grafen beschouwen, bevat de matrix enkel waarden uit de verzameling $\{0,1\}$ en is bijgevolg is $k = 2$.

Vervolgens kiezen we nog een groep van permutaties die we gebruiken om isomorfisme tussen de vectoren voor te stellen. We dienen deze op zo een manier te kiezen dat ze overeenkomen met de notie van graafisomorfisme: $\mathcal{G}$ = "alle permutaties van de bogen die geïnduceerd worden door de permutaties van de knopen".

Voorbeeld 3.9. *Beschouw de graaf in Figuur 3.7a, deze komt overeen met de code* $(1,1,0,0,0,0)$*. Wanneer we nu positie 2 met positie 6 verwisselen bekomen we de code* $(1,0,0,0,0,1)$*, hetgeen overeenkomt met de graaf in Figuur 3.7b. Deze graaf is duidelijk niet isomorf met de graaf uit Figuur 3.7a, volgens onze notie van isomorfisme. We dienen de toegelaten permutaties van componenten van de vector dus te beperken. Merk op dat een permutatie van de knopen* $(1\ 2)$ *de volgende mapping van de bogen induceert:*

$$\begin{cases} \{1,2\} \to \{1,2\} \\ \{1,3\} \to \{2,3\} \\ \{1,4\} \to \{2,4\} \\ \{2,3\} \to \{1,3\} \\ \{2,4\} \to \{1,4\} \\ \{3,4\} \to \{3,4\}. \end{cases} \tag{3.5}$$

De overeenkomstige permutatie van de posities komt dus overeen met

$$\begin{cases} 1 \to 1 \\ 2 \to 4 \\ 3 \to 5 \\ 4 \to 2 \\ 5 \to 3 \\ 6 \to 6. \end{cases} \tag{3.6}$$

De vector die we verkrijgen na het toepassen van deze permutatie op $(1,1,0,0,0,0)$*, bekomen we* $(1,0,0,1,0,0)$*, waarvan de overeenkomstige graaf te zien is in Figuur 3.7c. Het is duidelijk dat deze graaf wel isomorf is.*

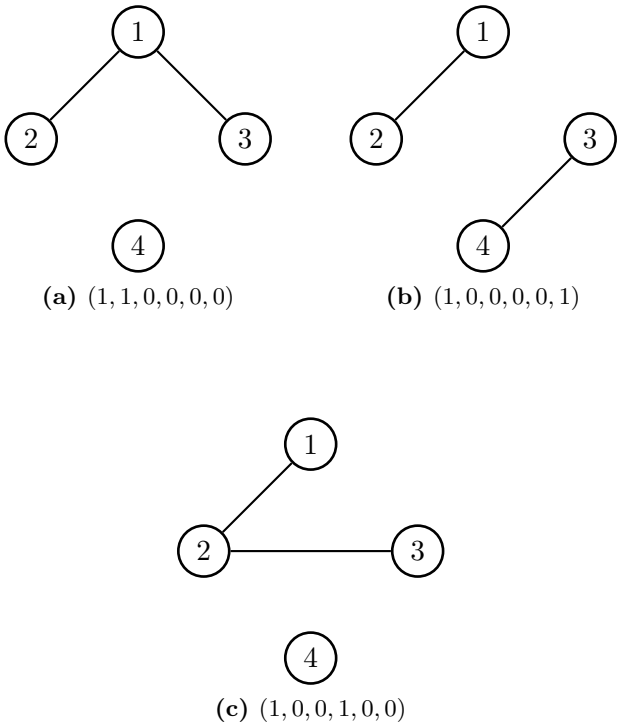
In Bijlage A kan een uitwerking teruggevonden worden van het algoritme, toegepast op grafen met drie knopen en twee mogelijke booglabels.

3.2.2 Bomen

Een interessant voorbeeld van een orderly algoritme is dat van gewortelde bomen. *Gewortelde bomen* zijn bomen waarbij één knoop is aangeduid als zijnde de wortel van de boom. We kunnen een eenvoudige representatie bouwen voor dit soort bomen: de "haakjesnotatie". Deze wordt recursief gedefinieerd:

1. De notatie voor een blad is de lege string.
2. De notatie voor een gewortelde boom is de concatenatie van de volgende string voor elke principiële subtree:

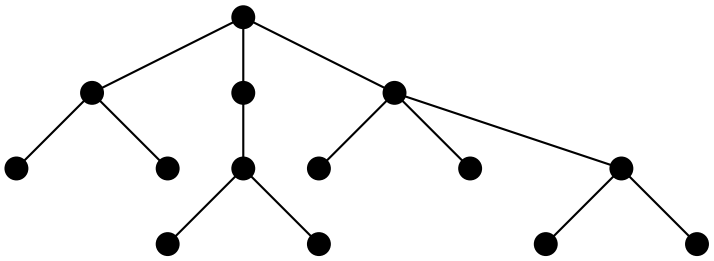
   '(' *representatie van subtree* ')'.



(a) $(1, 1, 0, 0, 0, 0)$ **(b)** $(1, 0, 0, 0, 0, 1)$

(c) $(1, 0, 0, 1, 0, 0)$

Figuur 3.7: Het zoeken van de correcte permutatiegroep.



Figuur 3.8: Een boom.

Bovenstaande notatie geeft ons voor elke boom een string die bestaat uit een reeks geneste haakjes. Een *principiële subtree* is hierbij een kind van de wortel, samen met al zijn nakomelingen. Wanneer we '(' vervangen door '0' en ')' door '1' bekomen we een equivalente representatie, maar dan enkel met de symbolen '0' en '1'.

Voorbeeld 3.10. *De code voor de boom in Figuur 3.8 is*

$$(()())((()()))(()()(()())),$$

of equivalent

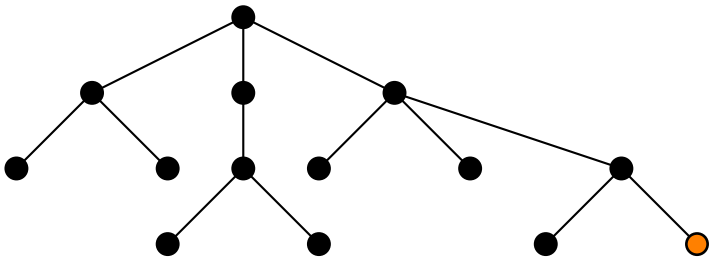
$$00101100010111001010010111.$$

In hetgeen volgt zullen we een orde op de kinderen van een knoop in de boom leggen, dit soort boom wordt aangegeven met de term *geplante boom* of *planted tree*.

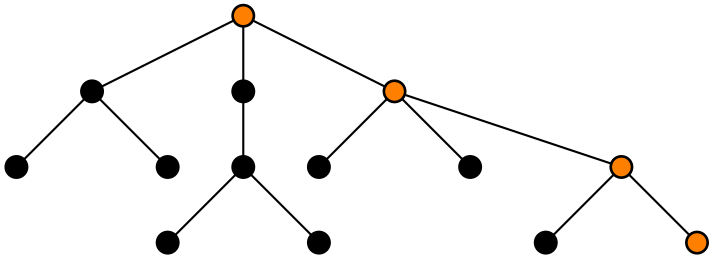
We nemen voor $\mathcal{L}$ de verzameling van alle bomen, en voor $\mathcal{L}_q$ de deelverzameling hiervan die alle bomen met q bogen bevat. We beschouwen de bomen waarbij er een andere volgorde ligt op de subtrees als isomorf en definiëren de canonische vorm van een boom op de volgende recursieve manier: een boom is canonisch als en slechts als

1. de principiële subtrees canonisch zijn,
2. de volgorde van de principiële subtrees als volgt is:
   (a) degene met de meeste bogen eerst,
   (b) bij een gelijk aantal bogen komt degene met de grootste `Code` eerst.

We kunnen nu de uitbreidingsfunctie φ als volgt beschouwen: voeg op elke mogelijke wijze een nieuw kind toe aan de bestaande boom. Het is duidelijk dat dit heel erg veel kinderen gaat opleveren. De methode die we nu beschrijven genereert veel minder kinderen, en zorgt er toch voor



(a) De meest-rechtse knoop



(b) De knopen op het meest-rechtse pad

Figuur 3.9: Meest-rechtse knoop en meest-rechtse pad.

dat alle bomen gegenereerd worden. Noem de laatste knoop van de boom (laatst vermeld in de `Code`, zie Figuur 3.9a) de *meest-rechtse knoop* en het pad van de wortel naar deze knoop het *meest-rechtse pad* (zie Figuur 3.9b). Vervolgens construeren we de uitbreidingsfunctie φ als de functie die aan elke knoop op het meest-rechtse pad een nieuw kind toevoegd. Let wel, dit nieuwe kind moet het laatste kind van de betreffende knoop zijn. Het invoegen van een nieuwe knoop komt overeen met het invoegen van een "01" in de code, op een geldige plaats. De volgorde waarin dit gebeurt is eerst zo ver mogelijk vanvoor in de `Code` en hierna iedere keer meer naar achteren, zodat de kinderen in de juiste volgorde gegenereerd worden (kleinste binaire codes eerst) en voorwaarde 3.3 automatisch voldaan is. Indien we aantonen dat de overige voorwaarden gelden voor deze constructie, dan hebben we een correct orderly algoritme voor het genereren van bomen!

Voorbeeld 3.11. *De boom in Figuur 3.10a kan met behulp van de haakjesnotatie genoteerd worden als*

$$()(()()).$$

We zien in de figuur dat knopen A, C en E uitgebreid kunnen worden met een nieuwe knoop, omdat ze zich op het meest-rechtse pad bevinden. De

plaatsen in de code waar dit toegelaten is duiden we aan met een '$*$':

$$()(()(*)*)*.$$

Wanneer we de code noteren in de '01'-notatie, bekomen we

$$01001011.$$

De groeioperatie geeft ons volgende drie nieuwe codes:

$$0100100111, \tag{3.7}$$
$$0100101011, \tag{3.8}$$
$$0100101101. \tag{3.9}$$

De overeenkomstige grafen zijn te zien in respectievelijk Figuur 3.10b, 3.10c en 3.10d. Het is duidelijk dat de codes gegenereerd worden in de stijgende volgorde van de binaire getallen die ze voorstellen.

Stelling 3.12. *Voor iedere canonische boom T met $q+1$ bogen bestaat er een canonische boom T' die q bogen heeft en waaruit T gegenereerd kan worden door de toepassing van φ.*

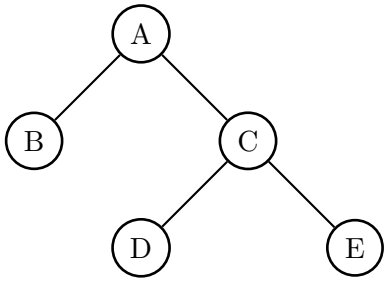
Bewijs. Neem de meest-rechtse knoop p uit T en verwijder deze om een boom T' te bekomen. We tonen vervolgens aan dat T' canonisch is, door twee gevallen te beschouwen:

Geval 1 p is rechtsreeks verbonden met de wortel

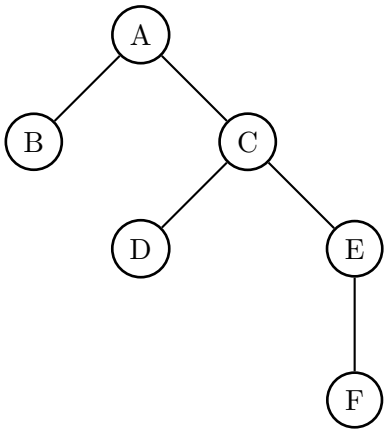
Als p rechtstreeks aan de wortel hangt, dan is p op zich een principiële subtree van T. We weten ook dat dit de laatste subtree was. De overige subtrees zijn dus nog steeds canonisch en staan in de juiste volgorde, waardoor T' alle eigenschappen bezit die nodig zijn om canonisch genoemd te worden. In Figuur 3.11a wordt deze situatie weergegeven.

Geval 2 p is niet rechtstreeks verbonden met de wortel

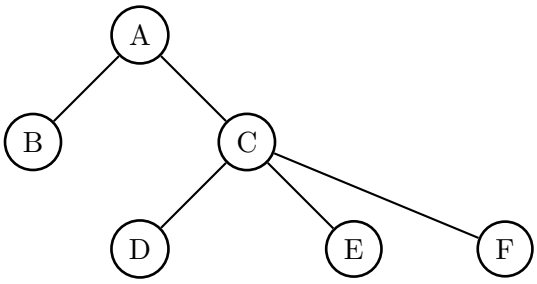
p is in dit geval een onderdeel van de laatste principiële subtree van T, maar we kunnen in deze subtree telkens dieper gaan kijken, totdat we geval 1 kunnen toepassen. We mogen dus aannemen dat de subtree canonisch blijft. Verder is het zo dat deze subtree noodzakelijk minder bogen bevat dan hij oorspronkelijk had, waardoor hij nog steeds op de juiste plaats staat. We kunnen twee gevallen onderscheiden: ofwel had hij al het minste bogen, ofwel had hij de kleinste code van subtrees met eenzelfde aantal bogen. In beide gevallen in het zo dat deze graaf de enige wordt met het kleinste aantal bogen, waardoor hij dus vanachter dient te staan. De graaf is bijgevolg canonisch. Figuur 3.11b geeft dit geval schematisch weer.



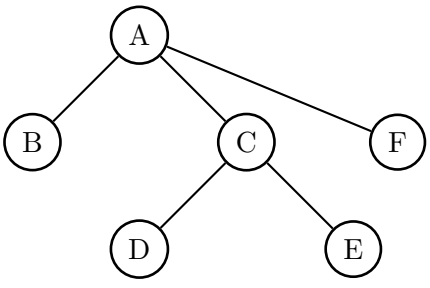
(a) Een boom met 5 knopen



(b) Uitbreiding 1



(c) Uitbreiding 2



(d) Uitbreiding 3

Figuur 3.10: Expansie van bomen.

De toepassing van φ voegt knopen toe aan het meest-rechtse pad, en maakt bijgevolg een nieuwe meest-rechtse knoop. Het is duidelijk dat dit net de knoop is die we verwijderd hebben. Bijgevolg is er altijd een canonische ouder voor elke canonische boom. □

Stelling 3.13. *De functie f is zwak-monotoon.*

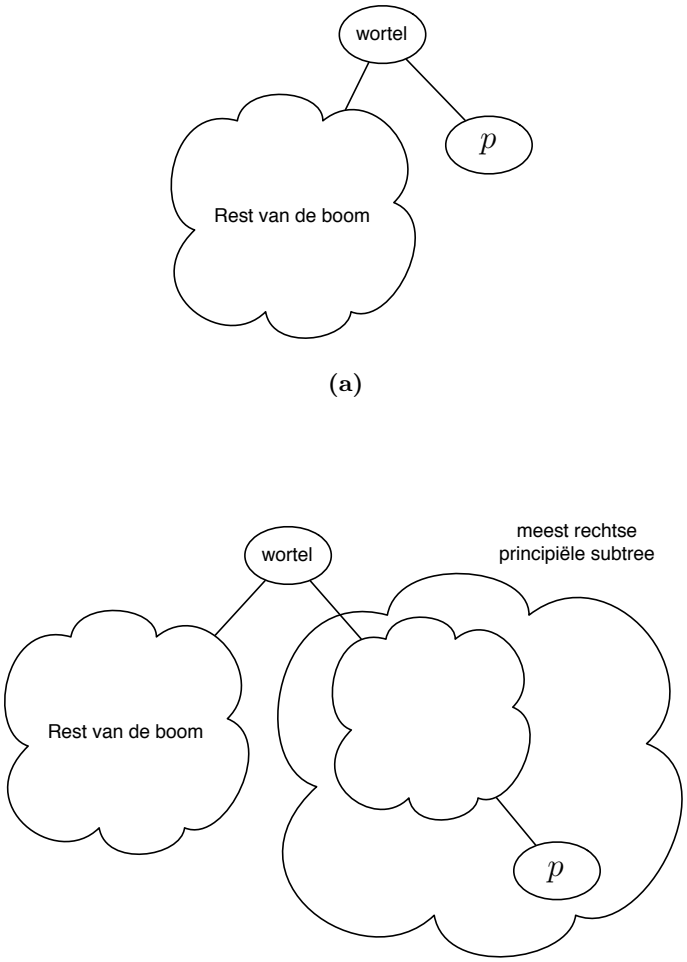
Bewijs. Stel dat de stelling geldt voor bomen met q bogen en minder. We nemen twee canonische bomen T_1 en T_2, die elk $q+1$ bogen bevatten. Door de meest rechtse knoop in beide grafen te verwijderen bekomen we twee nieuwe grafen: $T_1' = f(T_1)$ en $T_2' = f(T_2)$, die elk q bogen bevatten. Als $T_1 \prec T_2$, dan geldt ook $\texttt{Code}(T_1) < \texttt{Code}(T_2)$ (en omgekeerd). We moeten nu aantonen dat $T_1' \preccurlyeq T_2'$ of $\texttt{Code}(T_1') \leq \texttt{Code}(T_2')$.

We vergelijken één voor één de principiële subtrees van T_1 en T_2 en nemen de eerste waarin ze verschillen. Als dit niet de laatste subtree is, dan blijven de resulterende bomen verschillend op deze, omdat er niets aan het eerste verschilpunt veranderd is. T_1' (resp. T_2') kan namelijk enkel bekomen worden uit T_1 (resp. T_2) door het meest-rechtse pad (laatste subtree) aan te passen. In dit geval geldt $T_1' \prec T_2'$ en dus zeker $T_1' \preccurlyeq T_2'$. Indien het om de laatste subtree gaat, dan moet de code van deze subtree in T_1 kleiner zijn dan die in T_2. Deze boom heeft maximaal q bogen, en als we hierop de inductiehypothese toepassen, kunnen we concluderen dat de resulterende subtree in T_1' nog steeds kleiner is dan die in T_2', waardoor het gestelde geldt. □

3.2.3 Breadth-first vs. Depth-first

De manier waarop het beschreven algoritme werkt noemen we *breadth-first*: de zoekruimte - in dit geval een boom - wordt laag per laag ontdekt. Eerst worden alle objecten met $q = 1$ allemaal geconstrueerd, daarna al degene met $q = 2$, enz... Het is aan de andere kant ook mogelijk om te werken op de *depth-first* manier: dit houdt in dat we elk gevonden object op een bepaalde laag eerst helemaal uitdiepen: we gaan op elk nieuw object recursief verder. Als de recursie op een bepaalde tak uitgeput is, keren we via backtracking terug en genereren we het volgende object dat we opnieuw recursief uitbreiden. Op deze manier is het niet nodig om voor een volgende laag de hele vorige laag in te laden en is er (behalve in extreem grote zoekruimten) geen nood aan externe opslag. De *depth-first* manier van werken geeft ons dus een manier om het algoritme uit te voeren, zonder veel geheugenvereisten!

Merk wel op dat dit enkel kan werken indien elk niveau in volgorde wordt geproduceerd. Zoals reeds eerder aangehaald, stond voorwaarde 3.3 in voor deze eis. We weten dat in het oorspronkelijke algoritme alles in volgorde gegenereerd wordt, omdat enkel grotere objecten toegevoegd worden. We houden voor elk niveau het laatste object bij en kunnen op deze manier een recursief algoritme bouwen.



Figuur 3.11: Verschillende gevallen van canonische ouders voor bomen.

Opmerking 3.14. Bij *breadth-first* kan de geheugeneis ook laag gehouden worden door de objecten slechts één per één, of in kleine stukjes terug in te lezen. Het is in dit geval duidelijk dat we naast het geheugen nog over secundaire opslag dienen te beschikken, hetgeen voor grote vertragingen zorgt.

3.2.4 Samenvatting

Een orderly algoritme stelt als voorwaarden dat grotere objecten kunnen "groeien" uit kleinere. Ook moet de relatie kind-ouder zorgen dat de kinderen in de juiste volgorde worden gegenereerd. Wanneer we een nieuw object construeren dat kleiner is dan een eerder geconstrueerd object, dan laten we dit weg. Verder stellen eisen we dat de objecten die gegenereerd worden canonisch zijn, indien dit niet het geval is, worden ze geweigerd. We genereren met andere woorden enkel canonische objecten in een zekere volgorde.

Opdat dit werkt is het noodzakelijk dat elk canonisch object gegenereerd kan worden door een kleiner canonisch object. Anders zouden er objecten zijn die niet in de ouput verschijnen. Het is duidelijk dat, als alle canonische objecten met een kleinere orde gegenereerd worden, dit ook het geval is voor de canonische objecten met een grotere orde.

Tot slot vermelden we nog dat deze techniek in essentie hetzelfde is als deze voorgesteld door Faradzev [15]. Faradzev concentreert zich echter enkel op de objecten van één bepaalde orde, en beschouwt de objecten nodig om daar te geraken eerder als minderwaardig (hulpmiddel). Read doet dit niet, hetgeen ons alle objecten oplevert tot en met een bepaalde orde.

3.3 gSpan

3.3.1 Inleiding

In deze sectie beschrijven we een algoritme dat gebruikt wordt om in een database met grafen frequente patronen te ontdekken. Het algoritme wordt beschreven door Yan et al.[37] en werkt met gelabelde grafen. Ook dit algoritme dient alle mogelijke grafen te genereren om te tesen of hun patronen wel frequent zijn en het is dan ook belangrijk dat geen enkele graaf twee keer gegenereerd wordt (geen isomorfismen).

De manier waarop dit gebeurd is door eerst een voorstelling voor de grafen te bouwen, en hierop vervolgens een orde te leggen zodat we de "minimale voorstelling" van een graaf kunnen berekenen. Hierna construeren we een groei-operatie die een boog aan een bestaande graaf toevoegt om een nieuwe te bekomen (een "kind"). Het is duidelijk dat deze constructie bijzonder veel lijkt op de manier waarop de algoritmen beschreven in de vorige sectie geconstrueerd werden en het is dan ook niet verassend dat dit algoritme in essentie een orderly algoritme is, zoals gedefinieerd door Read [26].

Dit algoritme is echter meer toegespitst op het werkelijk *minen* van een database, dan enkel maar het genereren van grafen. Er wordt dus ook gekeken naar het uitsluiten van paden in de zoekboom die geen bijdrage meer kunnen leveren. De voordelen van het algoritme zijn dat het goed schaalbaar is, en dat er een parallellisatie mogelijk is. Verder worden er geen kandidaten gegenereerd zoals bij andere *data mining* algoritmen, en wordt de isomorfisme-test gecombineerd met de groei-operatie. gSpan doorloopt de zoekruimte — in de vorm van een boom — *depth-first*, waarvan we in de vorige sectie al een beschrijving gegeven hebben.

Allereerst geven we de definitie van een gelabelde graaf, die niet zo erg verschillend is van de ongelabelde variant:

Definitie 3.15 (gelabelde graaf)**.** Een (ongeordende) *gelabelde graaf* is een 4-tupel $G = (V, E, L, l)$, waarbij

- V een verzameling knopen is,
- E een verzameling bogen is, met $E \subseteq \{\{v, w\} \mid v, w \in V\}$,
- L een verzameling labels is,
- $l : V \cup E \rightarrow L$ een functie is die aan elke knoop en boog *precies één* label toekent.

Merk op dat, indien $\epsilon \in L$, waarbij ϵ het lege label voorstelt, we een *partieel gelabelde graaf* kunnen construeren. Met dit laatste bedoelen we een graaf, waarbij niet alle knopen en bogen voorzien zijn van een label. De verzameling knopen (resp. bogen) van G noteren we nog steeds als $V(G)$ (resp.

$E(G)$). De labeling functie l kunnen we voor een graaf G noteren met de subscriptnotatie: l_G. In het algemeen zal de verzameling L gedeeld worden door de grafen die we beschouwen, zodat deze beschouwd kan worden als een constante.

In hetgeen volgt zullen we een ongerichte boog echter noteren als een geordend paar (v_i, v_j). In plaats van v_i en v_j te gebruiken is het eveneens mogelijk om simpelweg i en j te gebruiken, omdat we de knopen kunnen voorstellen als getallen. De gebruikte notatie geeft echter een meer algemene definitie van een knoop, waarbij i en i aangeven welke plaats de knoop in de orde op de knopen heeft.

We kunnen nu onze notie van isomorfie (en hiermee dus ook automorfie) uitbreiden op gelabelde grafen. Naast de relatie die de bogen voorstelt beschouwen we enkele additionele relaties, voor elk label l:

- l_1, een unaire relatie die alle knopen bevat die gelabeld zijn met l,
- l_2, een binaire relatie met alle bogen die voorzien zijn van het label l.

Merk op dat de relatie met bogen in essentie nutteloos geworden is. Het is echter praktisch om deze erbij te houden. Wanneer we de notie van isomorfie uitwerken op structuren van deze vorm, kunnen we hetvolgende besluiten:

Besluit 3.16. *Een isomorfisme f tussen twee gelabelde grafen G en H is een bijectieve functie $f : V(G) \rightarrow V(H)$, zodat*

1. $\forall u \in V(G) : l_G(u) = l_H(f(u))$,
2. $\forall (u, v) \in E(G) : l_G(u, v) = l_H(f(u), f(v))) \wedge (f(u), f(v)) \in E(H)$.

We moeten natuurlijk ook weten wat de frequentie van een graaf in een database is, deze wordt als volgt gedefinieerd:

Definitie 3.17 (*frequent subgraph mining*)**.** Beschouw een database van grafen $\mathbb{GS} = \{G_i \mid i = 0, \ldots, n\}$ en een getal $MinSup$. Neem nu

$$\kappa(G, H) = \begin{cases} 1: & \exists H' \subseteq H : G \cong H', \\ 0: & anders. \end{cases} \tag{3.10}$$

Definieer de *support* σ van een graaf G in een database $\mathbb{GS}$ als

$$\sigma(G, \mathbb{GS}) = \Sigma_{G_i \in \mathbb{GS}} \kappa(G, G_i).$$

We willen nu alle grafen G waarvoor $\sigma(G, \mathbb{GS}) \geq MinSup$. Dit probleem noemen we *frequent subgraph mining*.

De grafen worden door het algoritme van klein naar groot gegenereerd, volgens een zekere orde. Na het definiëren van een uitbreidingsoperatie kunnen we alle grafen in een boom voorstellen, waarin de kinderen verbonden zijn met hun ouder(s). We construeren deze boom op zo een manier dat, indien deze in pré-orde doorlopen wordt, we alle grafen van klein naar groot tegenkomen. Om het algoritme efficiënter te maken, laten we toe dat we een bepaalde tak in een boom mogen *prunen*, om twee redenen:

1. de huidige graaf is niet frequent of
2. de huidige graaf is niet canonisch.

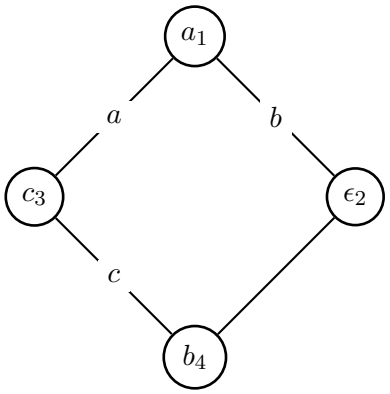
We merken tot slot nog op dat gSpan enkel *ongerichte geconnecteerde grafen* beschouwt, in dit soort grafen bestaat tussen elk paar knopen een pad. *Multi-edges* en *loops* worden in de standaardversie van het algoritme niet ondersteund, maar kunnen als uitbreiding toegevoegd worden. Met *Multi-edges* bedoelen we meerdere bogen tussen dezelfde knopen, met *loops* bedoelen we bogen van een knoop naar zichzelf.

3.3.2 Een orderly algoritme

We beginnen met het gegeven probleem te modelleren aan de hand van de orderly generatie-aanpak voor combinatorische objecten, beschreven in sectie 3.2.1. We laten in deze sectie het deel over *graph mining* buiten beschouwing en concentreren ons enkel op het genereren van grafen. We nemen $\mathcal{L}$ als de verzameling van alle grafen op m knopen, waarbij m een constante is. Vervolgens nemen we de voorstelling van de graaf zoals het uigewerkte voorbeeld uit sectie 3.2.1 (bovenste triangulaire deel van de matrix). Maar we zien dadelijk dat we hieraan niet genoeg hebben: we moeten namelijk de labels van de graaf ook in de code encoderen, willen we een correcte voorstelling bekomen. Om dit op te lossen beschouwen we de verzameling van alle labels L. We nemen nu voor $k = |L| + 1$, elk label krijgt een getal dat ermee overeenkomt uit $\{1, \ldots, k-1\}$: we construeren een bijectie tussen de twee verzamelingen. Het getal '0' blijft zoals hiervoor voorzien om aan te duiden dat er geen boog tussen de knopen bestaat. Merk op dat het lege label ϵ enkel toegestaan is indien dit een element is uit L.

Deze constructie geeft de mogelijkheid om aan elke boog in de graaf een label te koppelen. Maar hoe zit het met de knopen? Deze kunnen namelijk ook gelabeld zijn en ook dit dient in de code verwerkt te worden. Om dit op te lossen voeren we aan het begin de code nog m componenten toe die de labels van de knopen voorstellen. Merk wel op dat de waarde '0' geen betekenis heeft voor de knopen en niet gebruikt moet worden. We verkrijgen een graafrepresentatie van de volgende vorm:

$$(l_1, \ldots l_m, (1,2), \ldots (1,m), (2,3), \ldots, (2,m), \ldots (m-1,m)),$$



Figuur 3.12: Een graaf met 4 knopen.

waarbij l_i het overeenkomstige labelnummer is voor knoop i. Merk op dat de grootte van de vectoren gegeven is door $n = m + \binom{m}{2}$.

Voorbeeld 3.18. *Als* $L = \{\epsilon, a, b, c, d\}$ *en* $m = 4$*, verkrijgen we* $k = 6$*, hetgeen ons de verzameling* $\{0, 1, 2, 3, 4, 5\}$ *oplevert. De vector voor de graaf uit Figuur 3.12 ziet er als volgt uit:*

$$(2, 1, 4, 3, 3, 0, 2, 0, 1, 4).$$

De labels staan in de knopen van de graaf, voozien van een subscript dat het nummer van de knoop aanduidt. De bijectie tusen de labels en de getallen is triviaal ($\epsilon \leftrightarrow 1$*,* $a \leftrightarrow 2$*, ...).*

Op deze manier hebben we succesvol een code geconstrueerd die ons toelaat om de grafen voor te stellen, we hebben nu enkel nog een permutatiegroep $\mathcal{G}$ nodig. In dit geval nemen we hiervoor opnieuw de permutaties op de codes die geïnduceerd worden door de permutaties op de knopen. Dit is de verzameling S_n van alle permutaties op $\{1, \ldots, m\}$[3], die we ook bij de vorige constructie ook gebruikt hebben, al hebben we daar niet expliciet vermeld dat deze overeenkomt met S_n.

We verkrijgen nu een algoritme dat ons toelaat om alle grafen te genereren op m knopen. Merk op dat we in het algemeen niet enkel geïnteresseerd zijn in grafen met een vast aantal knopen, maar in grafen met knopen tot een bepaalde constante. Het generisch algoritme beperkt ons echter tot één soort van grafen: deze met een constant aantal knopen. Om dit te omzeilen, moeten we een invulling van de voorwaarden van het orderly algoritme construeren die ons toelaat om uit grafen met minder knopen grafen met meer knopen te bouwen. Dit komt overeen met de aanpak die gebruikt werd om

[3]Zie hoofdstuk 2.

bomen te genereren, zoals in sectie 3.2.2. Het gSpan-algoritme zal op analoge wijze tewerk gaan, maar maakt gebruik van een iets andere code. Merk op dat bij de constructie van bomen er geen rechtstreekse vergelijking mogelijk was tussen de codes van bomen met q knopen en deze met $q-1$ knopen. We kunnen dus niets zeggen over volgorde van objecten wanneer deze met een *depth-first search* gegenereerd worden. Het gSpan-algoritme zal deze beperking niet hebben, waardoor we kunnen zeggen dat *alle* objecten van klein naar groot gegenereerd worden.

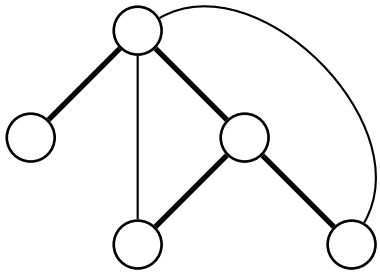
Opmerking 3.19. Het is natuurlijk mogelijk om met dit algoritme alle grafen te genereren die we willen: we laten m lopen van $[1 \dots x]$ en voeren voor elk van deze waarden het algoritme uit. Op deze manier kunnen we nog steeds niveau per niveau genereren. We hebben echter wel nood aan verschillende instanties van het algoritme, waardoor we ook in parallel kunnen werken. Het is niet verassend dat de complexiteit per laag in het algoritme zeer snel zal stijgen, omdat het aantal grafen exponentieel stijgt met het aantal bogen, zoals we reeds gezien hebben.

3.3.3 DFS-tree

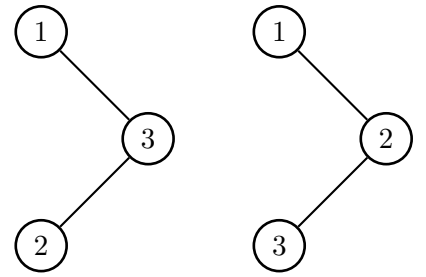
Definitie 3.20 (DFS tree)**.** Een *depth-first search tree* of *DFS Tree* T van een graaf G is de boom die we verkrijgen door, vertrekkende van een bepaalde knoop, een *depth-first search* uit te voeren. Er geldt dat $T \subseteq G$ en dat $V(T) = V(G)$.

Uit de definitie blijkt dat dit een boom is die bestaat uit alle knopen van de graaf, en een deelverzameling van zijn bogen. De vette lijnen in Figuur 3.13 geven een boom die alle knopen van de graaf bevat, samen met een deelverzameling van zijn bogen. De manier waarop we een DFS-tree bekomen is door te vertrekken bij een willekeurige knoop, en vervolgens een buur van deze knoop uit te kiezen die we nog niet beschouwd hebben, hierop werken we recursief verder totdat we bij een knoop zitten die geen onbezochte buren heeft. Indien we een dergelijke knoop tegenkomen, keren we terug naar de ouder en proberen we daar een volgende buur te bezoeken. Dit blijven we doen totdat we alle knopen in de graaf bezocht hebben. Het is duidelijk dat de paden die we afgelegd hebben een boom vormen.

Er zijn natuurlijk veel mogelijke *DFS-trees* voor één graaf, en elk van hen geeft aanleiding tot een zekere orde op de knopen: de volgorde waarin de knopen "ontdekt" zijn. Het is niet moeilijk om in te zien dat een DFS-tree eigenlijk gewoon een permutatie van de knopen is, maar Figuur 3.14a laat ons zien dat niet iedere permutatie van de knopen een geldige DFS-tree oplevert: er is namelijk geen boog van knoop 1 naar 2, waardoor, indien we met 1 beginnen, knoop 2 noodzakelijk pas na knoop 3 ontdekt *kan* worden. De orde $1, 2, 3$ kan bijgevolg niet gelinkt worden aan een DFS-tree. De knooporde die in Figuur 3.14b gebruikt wordt stelt echter wel een geldige



Figuur 3.13: Een graaf met een DFS-tree (vet).



(a) een ongeldige DFS-tree

(b) een geldige DFS-tree

Figuur 3.14: Knoopordes en *DFS-trees*.

DFS-tree voor.

Als voorstelling van de graaf zullen we een opeenvolging van zijn bogen nemen, in een bepaalde volgorde. We beschouwen bepaalde ordes als geldig en anderen als ongeldig. We duiden de knopen zoals hierboven aan met v_i, waarbij i het volgnummer is van de knoop in de orde opgelegd door de DFS-tree. Herinner dat de bogen van de vorm (v_i, v_j) zijn.

3.3.4 Canonische graaf

Om een canonische versie van een graaf te bepalen, is het nodig dat we een orde op de knopen leggen. Als we een algoritme selecteren dat voor iedere isomorfe graaf diezelfde orde berekent, dan hebben we succesvol een graaf-canonisatie algoritme geconstrueerd (zie sectie 2.4). Deze volgorde van de knopen *kan* overeenkomen met één bepaalde DFS-tree. *DFS-trees* bepalen eveneens een ordening op de knopen, we kunnen dus een canonische ver-

sie van een graaf bekomen als we één bepaalde DFS-tree kiezen voor een verzameling isomorfe grafen.

Om dit te verwezenlijken bouwen we een `Code` op als een opeenvolging van bogen in een bepaalde volgorde. Gegeven een graaf G en een DFS-tree T splitsen we de verzameling bogen $E(G)$ op in twee delen:

- De bogen die zowel in T als in G aanwezig zijn, deze verzameling noemen we de *forward edge set*: $E_{f,T}$.
- de bogen die in G aanwezig zijn, maar niet in T, deze verzameling noemen we de *back edge set*: $E_{b,T}$.

Formeel definiëren we deze verzamelingen als volgt:

$$E_{f,T} = \{e \mid e = (v_i, v_j) \in E(G) \wedge i < j\} \tag{3.11}$$

als de *forward edge set* in G_T is, en

$$E_{b,T} = \{e \mid e = (v_i, v_j) \in E(G) \wedge i > j\} \tag{3.12}$$

als de *back edge set* in G_T. Dit is correct, omdat *forward edges* altijd van een reeds ontdekte knoop naar een nieuwe gaan, dus van een knoop met een lager nummer naar een knoop met een hoger nummer. De *back edges* gaan altijd naar knopen met een lager nummer. Wanneer het niet duidelijk is over welke graaf het gaat, duiden we deze verzamelingen aan met $E_{f,T}(G)$ en $E_{b,T}(G)$. We leggen vervolgens een onderlinge partiële orde op de bogen in deze verzamelingen, neem $e_1 = (v_{i_1}, v_{j_1})$ en $e_2 = (v_{i_2}, v_{j_2})$:

$$\forall\ e_1, e_2 \in\ E_{f,T} : e_1 \prec_{f,T}\ e_2 \Leftrightarrow j_1 < j_2, \tag{3.13}$$

en

$$\forall\ e_1, e_2 \in\ E_{b,T} : e_1 \prec_{b,T}\ e_2 \Leftrightarrow i_1 < i_2 \vee (i_1 = i_2 \wedge j_1 < j_2). \tag{3.14}$$

Merk op dat deze ordes enkel *in* de verzamelingen zelf een orde definiëren. Om *back edges* en *forward edges* onderling te vergelijken stellen we de volgende orde voor:

$$\begin{aligned}\forall\ e_1, e_2 \in\ E : e_1 \prec_{bf,T}\ e_2 \Leftrightarrow\ & e_1 \in E_{b,T}, e_2 \in E_{f,T}, i_1 < j_2 \\ & \vee\ e_1 \in E_{f,T}, e_2 \in E_{b,T}, j_1 \leq\ i_2.\end{aligned} \tag{3.15}$$

Door het combineren van de drie ordes, kunnen we een lineaire orde bouwen op de hele verzameling van bogen, we noemen deze orde $\prec_{E,T}$. Samengevat kunnen we zeggen dat voor $e_1 = (v_{i_1}, v_{j_1})$ en $e_2 = (v_{i_2}, v_{j_2})$ geldt dat $e_1 \prec E, Te_2$ als:

- $i_1 = i_2 \wedge j_1 < j_2$,

- $i_1 < j_2 \wedge j_1 = i_2$,
- $\exists e_3 : e_1 \prec_{E,T} e_3 \wedge e_3 \prec_{E,T} e_2$.

De eerste mogelijkheid zegt ons dat als er twee bogen zijn die vanuit eenzelfde knoop vertrekken, dat degene naar de "kleinste" knoop (ten opzichte van de orde op de knopen, bepaald door T) eerst komt. Merk op dat dit zowel geldt voor *forward edges* als voor *back edges*. De tweede mogelijkheid is dat de eerste boog een *forward edge* is en de tweede een *back edge* die vertrekt uit de "nieuwe" knoop die ontdekt is door de *forward edge*. Dit laatste zegt ons eigenlijk dat de *back edges*, vertrekkende uit een bepaalde knoop, onmiddellijk na het ontdekken van deze knoop door een *forward edge* komen. De laatste mogelijkheid is de transitiviteit. Om ons een beter beeld van deze eigenschappen te geven kunnen we dit voorstellen als volgt: de *back edges* die vanuit een bepaalde knoop vertrekken, volgen dadelijk na de *forward edge* naar deze knoop, de *forward edges* worden opgesomd in de volgorde aangegeven door de DFS-tree T.

We hebben nu een voorstelling voor grafen ontworpen die enkel bestaat uit de bogen. Hierbij is iedere boog een geordend paar knopen. Omdat we ook de mogelijkheid willen om met gelabelde grafen te werken, dienen we de labels van zowel de knopen als de bogen te incorporeren in deze graafvoorstelling. We doen dit door de representatie van een boog uit te breiden met drie nieuwe velden:

- het label voor de eerste knoop van de boog,
- het label van de boog zelf, en
- het label van de tweede knoop van de boog.

In plaats van een boogvoorstelling van de vorm (v_i, v_j) verkrijgen we dus eentje van de vorm

$$(v_i, v_j, l_i, l_{\{i,j\}}, l_j).$$

Hierbij zijn v_i en v_j de knopen waaruit de boog bestaat, en $l_i, l_{\{i,j\}}$ en l_j de labels van respectievelijk de eerste knoop, de boog zelf en de tweede knoop.

Gegeven de zonet geconstrueerde orde, is het mogelijk om te zeggen of een bepaalde opeenvolging van bogen in de juiste volgorde staat of niet.

Definitie 3.21 (Geldige DFS-code)**.** Een opeenvolging van bogen $(e_1, \ldots, e_n)$ wordt als een *geldige DFS-code* beschouwd als voor al de bogen e_i, e_j in deze lijst geldt dat

$$i < j \Rightarrow e_1 \prec_{E,T} e_2.$$

Uit het voorgaande kunnen we zeggen waaraan twee opeenvolgende bogen in een DFS-code moeten voldoen opdat de code als geldig beschouwd kan worden.

Eigenschap 3.22. *Gegeven een graaf G, een DFS-tree T en een DFS-code $\alpha = code(G,T)$, en we nemen aan dat α bestaat uit meer dan twee bogen: $\alpha = (a_0, a_1, \ldots, a_m)$ met $m \geq 2$. We beschouwen vervolgens twee opeenvolgende bogen in de code*

- $a_k = (i_k, j_k, l_{i_k}, l_{(i_k,j_k)}, l_{j_k})$ *en*
- $a_{k+1} = (i_{k+1}, j_{k+1}, l_{i_{k+1}}, l_{(i_{k+1},j_{k+1})}, l_{j_{k+1}})$,

waarbij $0 \leq k < m$, dan gelden de volgende regels:

Regel 1 als a_k een back edge *is, dan geldt:*

1. *als a_{k+1} een* forward edge *is: $i_{k+1} \leq i_k$ en $j_{k+1} = i_k + 1$,*
2. *als a_{k+1} een* back edge *is: $i_{k+1} = i_k$ en $j_k < j_{k+1}$.*

Regel 2 als a_k een forward edge *is, dan geldt:*

1. *als a_{k+1} een* forward edge *is: $i_{k+1} \leq j_k$ en $j_{k+1} = j_k + 1$,*
2. *als a_{k+1} een* back edge *is: $i_{k+1} = j_k$ en $j_{k+1} < i_k$.*

De regels komen op neer op het volgende:

Regel 1 Als na een *back edge* een *forward edge* volgt, dat moet deze vertrekken uit de eerste knoop van de *back edge*,of een kleinere. De boog moet naar de volgend knoop gaan. Indien er een *back edge* volgt, dan moet deze vertrekken vanuit dezelfde knoop, maar naar een knoop gaan die groter is dan de knoop waarheen de vorige *back edge* ging.

Regel 2 Als na een *forward edge* een *forward edge* volgt, dan dient deze te vertrekken vanuit de knoop waarin de vorige *forward edge* aankwam, of een kleinere knoop. Hij dient aan te komen in de eerstvolgende niet ontdekte knoop. Wanneer er een *back edge* volgt dan vertrekt deze van de knoop die in de *forward edge* ontdekt is, en gaat naar een kleinere knoop.

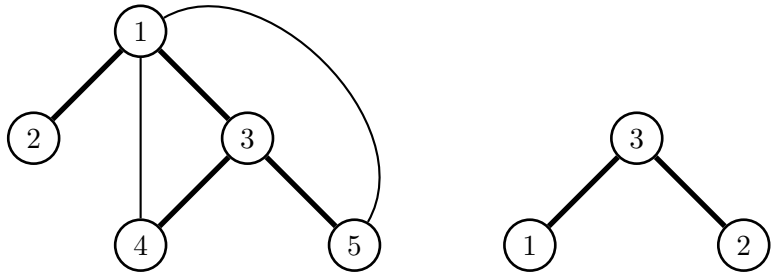
Voorbeeld 3.23. *In Figuur 3.15a zien we een graaf met een bepaalde orde op zijn knopen. De vette bogen stellen de bogen uit de DFS-tree voor. De code voor deze boom is*

$$(1,2)(1,3)(3,4)(4,1)(3,5)(5,1).$$

Deze code voldoet aan eigenschap 3.22 en is met andere woorden een geldige code.

Een voorbeeld van een graaf-boom-combinatie die geen geldige DFS-code kan opleveren is te zien in Figuur 3.15b. We kunnen met behulp van de orde op de knopen deze code construeren:

$$(1,3)(3,2),$$



(a) Deze graaf heeft meerdere *DFS-trees* voor de gegeven orde

(b) De graaf met deze orde heeft geen DFS-tree

Figuur 3.15: Grafen met een DFS-tree (vet).

hetgeen duidelijk geen geldige DFS-code is, omdat knoop 3 *voor knoop* 2 *"ontdekt" wordt. De orde op de knopen kan in dit geval onmogelijk aanleiding geven tot de constructie van een boom, waardoor er geen geldige DFS-code voor deze graaf-orde-combinatie bestaat.*

De stappen die we moeten ondernemen om een DFS-code te bekomen zijn de volgende:

1. Kies een willekeurige knoop x en markeer deze als bezocht.
2. Ga naar een willekeurige buur van de actieve knoop.
3. Markeer deze buur en ga naar stap 2.
4. Indien er geen buren beschikbaar gaan we terug naar de knoop via dewelke we hier zijn beland en gaan we naar stap 2.

Dit algoritme werd eerder al informeel besproken en geeft aanleiding tot de constructie van een boom. Het is namelijk zo dat alle paden die we afleggen samen een boom vormen.

3.3.5 Een orde op de DFS-codes

Tot nu toe hebben we een voorstelling voor grafen geconstrueerd. Hetgeen we nog ontbreken is een methode die ons toelaat om twee DFS-codes te vergelijken met mekaar. We kunnen dan een canonische vorm construeren aan de hand daarvan. We willen dus een orde op de DFS-codes bekomen, om het mogelijk te maken te bepalen welke de kleinste DFS-code van een zekere graaf is. Dit leidt tot de volgende definitie:

Definitie 3.24. beschouw de verzameling van alle DFS-codes van geconnecteerde gelabelde grafen: $Z(G) = \{code(G,T) \mid T \text{ is een DFS-tree voor } G\}$. Veronderstel vervolgens dat we beschikken over een lineaire orde $\prec_L$ op de verzameling van labels L, dan kunnen we deze orde combineren met de orde op de bogen ($\prec_{E,T}$) om te komen tot een samengestelde lineaire orde $\prec_e$ op de verzameling $E \times\ L \times\ L \times\ L$. We definiëren nu de lineaire *DFS lexicografische orde* als volgt: Neem $\alpha, \beta \in\ Z$, waarbij $\alpha = code(G_\alpha, T_\alpha) = (a_0, a_1, \ldots, a_m)$ en $\beta = code(G_\beta, T_\beta) = (b_0, b_1, \ldots, b_n)$, dan is

$$\alpha \leq \beta \Leftrightarrow \exists t, 0 \leq t \leq min(m,n), \forall k < t :$$
$$(a_k = b_k \wedge a_t \prec_e b_t) \vee (a_k = b_k \text{ voor } 0 \leq k \leq m, \wedge m \geq m).$$

De orde die we gebruiken is als volgt gedefinieerd: Neem

$$\alpha = (a_0, a_1, \ldots, a_m)$$
$$\beta = (b_0, b_1, \ldots, b_n),$$

waarbij $\alpha, \beta \in Z(G)$, dan geldt dat $\alpha \leq \beta$ als één van de volgende regels geldt: Beschouw de *forward edge set* en *back edge set* van beide grafen: $E_{\alpha,f}$,$E_{\alpha,b}$,$E_{\beta,f}$,$E_{\beta,b}$. Neem vervolgens $a_t = (i_a, j_a, l_{i_a}, l_{(i_a,j_a)}, l_{j_a})$ en $b_t = (i_b, j_b, l_{i_b}, l_{(i_b,j_b)}, l_{j_b})$.

1. Er bestaat een t, $0 \leq t \leq \min(m,n)$, zodat $a_k = b_k$ voor $k < t$, en $a_t < b_t$, deze laatste ongelijkheid geldt als

$$\begin{cases} a_t \in E_{\alpha,b} \wedge b_t \in E_{\beta,f} \\ a_t \in E_{\alpha,b} \wedge b_t \in E_{\beta,b} \wedge j_a < j_b \\ a_t \in E_{\alpha,b} \wedge b_t \in E_{\beta,b} \wedge j_a = j_b \wedge l_{(i_a,j_a)} < l_{(i_b,j_b)} \\ a_t \in E_{\alpha,f} \wedge b_t \in E_{\beta,f} \wedge i_b < i_a \\ a_t \in E_{\alpha,f} \wedge b_t \in E_{\beta,b} \wedge i_a = i_b \wedge l_{i_a} < l_{i_b} \\ a_t \in E_{\alpha,f} \wedge b_t \in E_{\beta,f} \wedge i_a = i_b \wedge l_{i_a} = l_{i_b} \wedge l_{(i_a,j_a)} < l_{(i_b,j_b)} \\ a_t \in E_{\alpha,f} \wedge b_t \in E_{\beta,f} \wedge i_a = i_b \wedge l_{i_a} = l_{i_b} \wedge l_{(i_a,j_a)} = l_{(i_b,j_b)} \wedge l_{j_a} < l_{j_b}, \end{cases}$$

2. $a_k = b_k, k = 0 \ldots m,$ en $n \geq m$.

Met behulp van deze orde hebben we de mogelijkheid om een orde op de DFS-codes zelf te leggen. We beschikken nu over de mogelijkheid om te beslissen welke van de verschillende DFS-codes van een zekere graaf de kleinste is. Deze DFS-code stemt overeen met een bepaalde DFS-tree $T \subseteq G$ en legt een orde op de knopen van G. Het blijkt dus dat dit een methode is om een canonische orde op de knopen te bepalen.

Definitie 3.25. Beschouw een graaf G, dan is $Z(G) = \{code(G,T) \mid \forall T :$ T is een DFS-tree voor $G\}$. Neem nu uit $Z(G)$ het kleinste element, gebruik makende van de *DFS lexicografische orde*: $\min(Z(G))$. Dit element noemen we de *minimale DFS-code van de graaf* G en is eveneens de canonische vorm van G. We noteren ook $\min(G)$ in de plaats van $\min(Z(G))$.

Merk op dat we aan min een graaf meegeven waarop reeds een bepaalde orde op de knopen ligt, we schrijven deze echter niet expliciet. Het is echter vanzelfsprekend dat we aan een algoritme een geordende input meegeven, zoals we besproken hebben in sectie 2.4. Uit het voorgaande kunnen we volgende eigenschap afleiden:

Eigenschap 3.26. *Zij G en G' twee grafen, dan geldt:*

$$\min(G, <) = \min(G', <') \Leftrightarrow G \cong G'.$$

Merk op dat min een *full-invariant* algoritme is, volgens 2.38. In hetgeen volgt zullen we deze orde op de knopen echter vaak niet vermelden bij de input.

Tot hiertoe hebben we een voorstelling van een graaf geconstrueerd, die enkel bestaat uit bogen: de DFS-code. Om deze code te bekomen hebben we een orde gelegd op de bogen van de graaf die door een bepaalde orde op de knopen geïnduceerd werd. Hierna hebben we een orde op de verschillende codes gelegd om te kunnen bepalen welke de canonieke vorm is. De volgende stap is om te bepalen welke grafen kunnen "groeien" uit anderen, of equivalent: welke DFS-codes worden uit anderen gegenereerd.

3.3.6 Groei-operatie

Definitie 3.27. Beschouw een DFS-code $\alpha = (a_0, a_1, \ldots, a_m)$, dan noemen we elke geldige DFS-code $\beta = (a_0, a_1, \ldots, a_m, b)$ een *kind* van α en α de *ouder* van β. Verder definiëren we kinderen$(\alpha)\{\beta \mid \alpha$ is de ouder van $\beta\}$.

Deze definitie geeft ons de goeioperatie φ. Deze functie bestaat eruit om de code achteraan uit te breiden met een nieuwe boog. Er zijn twee mogelijke versies van bogen die toegevoegd kunnen worden: een *forward edge* of een *back edge*. We onderscheiden vier gevallen volgens eigenschap dan moet eigenschap 3.22:

Geval 1 We voegen een *forward edge* toe, achter een *back edge*. De nieuwe boog dient te vertrekken uit dezelfde knoop als deze *back edge*, of een kleinere. De aankomst dient te zijn in de eerstvolgende niet-ontdekte knoop.

Geval 2 We voegen een *forward edge* achter een *forward edge* toe. In dit geval moeten we vertrekken van een eerder ontdekte knoop en moeten we aankomen in de eerstvolgende niet-ontdekte.

Geval 3 We voegen een *back edge* toe, achter een *back edge*. We moeten nu van dezelfde knoop vertrekken, en aankomen in een reeds-ontdekte knoop die later is dan de aankomstknoop van de vorige boog.

Geval 4 We voegen een *back edge* toe, achter een *forward edge*. Deze dient te vertrekken van de knoop die in de laatste *forward edge* bereikt werd. Hij dient aan te komen in een knoop die kleiner is dan de *forward edge* naar de knoop vanwaar we vertrekken.

Opdat we een geldige DFS-code verkrijgen, kunnen we een DFS-code uitbreiden door vanachter een *back edge* toe te voegen. Het is hierbij noodzakelijk dat deze vertrekt vanuit de laatste (dus de *meest-rechtse knoop*) naar een andere knoop. Een andere mogelijkheid is om achteraan een *forward edge* toevoegen, dan eisen we dat deze vertrekt van een knoop op het meest-rechtse pad, omdat anders de DFS-code niet in overeenstemming is met de DFS-tree. We bekijken de relatie tussen de DFS-tree en de DFS-code als volgt: de DFS-code geeft ons de *depth-first search* van de boom in de vorm van *forward edges*, ertussen worden ook de *back edges* gecodeerd zodat de hele graaf hiermee voorgesteld wordt.

Voorbeeld 3.28. *De DFS-code die overeenkomt met de graaf in Figuur 3.16 is*

$$(1,2)(1,3)(3,4)(4,1)(1,5)(5,6)(6,1).$$

We bespreken nu één voor één de mogelijke toevoegingen van de verschillende bogen en waarom ze al dan niet geldig zijn. We voegen een forward edge *toe na de* back edge $(6,1)$*, omdat we altijd achteraan toevoegen:*

$(1,7)$ *geldig,*

$(2,7)$ *ongeldig. Als we de overeenkomstige DFS-tree bekijken, dan bezoeken we eerst alle knopen, en daarna gaan we pas dieper in knoop* 2 *kijken. Merk op dat we sinds knoop 2 opnieuw omhoog zijn gegaan in de boom, hetgeen betekent dat knoop 2 uitgeput was. We mogen dus niet opnieuw knoop 2 uitbreiden om een nieuwe knoop te ontdekken.*

$(3,7)$ *ongeldig, wegens dezelfde reden,*

$(4,7)$ *ongeldig, wegens dezelfde reden,*

$(5,7)$ *geldig,*

$(6,7)$ *geldig.*

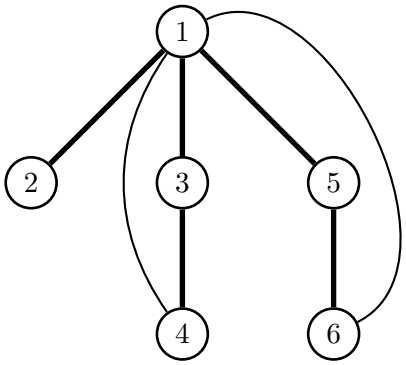
We voegen een back edge *toe na de* back edge $(6,3)$*:*

$(3,2)$ *ongeldig,* $3 \neq 6$*,*

$(4,1)$ *ongeldig,* $4 \neq 6$*,*

$(5,2)$ *ongeldig,* $5 \neq 6$*,*

$(5,3)$ *ongeldig,* $5 \neq 6$*,*



Figuur 3.16: Een graaf met een DFS-tree.

$(5, 4)$ *ongeldig,* $5 \neq 6$*,*

$(6, 2)$ *geldig,* $6 = 6 \wedge 1 < 2$*,*

$(6, 3)$ *geldig,* $6 = 6 \wedge 1 < 3$*,*

$(6, 4)$ *geldig,* $6 = 6 \wedge 1 < 4$*.*

3.3.7 Zoekruimte

Nu we de groei-operatie gedefinieerd hebben, kunnen we de zoekruimte zien als een boom die bestaat uit alle mogelijke DFS-codes, waarbij de bogen staan voor de ouder-kind relatie. Deze boom bevat elke geldige DFS-code, en bevat voor elk van deze een boog naar exact één ouder. Zoals eerder vermeld, zitten niet alle mogelijke grafen in deze boom, omdat bepaalde grafen geen overeenkomstige geldige DFS-code hebben.

Definitie 3.29. De DFS code tree is een boom $\mathbb{T}$ waarin elke knoop precies één DFS-code voorstelt, de ouder-kindrelatie (zie definitie 3.27) tussen DFS-codes wordt aangegeven door de bogen. Verder nemen we aan dat op de kinderen van een knoop in deze boom ook een orde ligt, en dat deze orde overeenkomt met de DFS lexicografische orde. Als we de boom in pré-orde zouden doorlopen, dan verkrijgen we een lijst van DFS-codes van klein naar groot (volgens $\prec_e$).

Omdat deze boom alle geldige DFS-codes bevat kunnen we de volgende eigenschap afleiden:

Eigenschap 3.30. *De DFS code tree* $\mathbb{T}$ *bevat de minimale DFS-code* $\min(G)$ *voor elke graaf* G*.*

Merk op dat niet-geldige codes niet in $\mathbb{T}$ voorkomen, verder weten we dat elke code overeenkomt met precies één geordende graaf. De geordende graaf komt bijgevolg niet in $\mathbb{T}$ voor.

Definitie 3.31. Gegeven twee DFS-codes $\alpha, \beta \in \mathbb{T}$, dan noemen we α een *voorouder* van β als er in $\mathbb{T}$ een pad bestaat van α naar β. In dit geval kunnen we ook zeggen dat β een *afstammeling* is van α. We definiëren twee functies: voorouders(α) = {alle voorouders van α} en afstammelingen(α) = {alle afstammelingen van α}.

Omdat het algoritme het doel heeft om een databank met grafen te *minen*, kunnen we al een optimalisatie invoeren. De bedoeling is om alle grafen in $\mathbb{T}$ te genereren, en hun voorkomen te testen in de databank. We weten dat de ouder van een graaf G ook isomorf is met een subgraaf van G, omdat deze enkele knopen en/of bogen minder bevat. Als een grote graaf echter veel voorkomt, dan weten we dat al zijn subgraafjes ook een groot voorkomen hebben: minstens evenveel. Omgekeerd kunnen we zeggen dat als minstens één subgraaf niet voldoende voorkomt in de databank, dat dan ook de grotere graaf niet genoeg zal voorkomen. Met "niet voldoende" duiden we op de voorgedefinieerde constante waarde *MinSup*, die de minimale *support* bepaalt. We kunnen op deze manier een hoop testen vermijden:

Eigenschap 3.32. *Als een graaf G frequent voorkomt in een database $\mathbb{GS}$, dan is elke subgraaf van G ook frequent. Als G aan de andere kant infrequent is, dan is geen enkele graaf, waarvan G een subgraaf is, frequent. We kunnen ook zeggen: Als een DFS-code α frequent is, dan geldt $\forall \beta \in$ voorouders$(\alpha) : \beta$ is frequent. Als een DFS-code α infrequent is, dan geldt $\forall \beta \in$ afstammelingen$(\alpha) : \beta$ is infrequent.*

Omdat in $\mathbb{T}$ de DFS-codes van groot naar klein voorkomen als we de boom in pré-orde doorlopen, weten we dat we eerst de minimale DFS-code van een graaf zullen tegenkomen, en hierna pas de "isomorfe kopiëen". We tonen nu aan dat, als we een niet-minimale DFS-code uit $\mathbb{T}$ verwijderen, samen met al zijn afstammelingen, dat $\mathbb{T}$ dan nog steeds alle minimale DFS-codes bevat.

Stelling 3.33. *Gegeven een niet-minimale DFS-code β, en een minimale DFS-code $\alpha = \min(\beta)$, dan geldt:*

$$\forall \delta \in \textit{kinderen}(\beta) : \min(\delta) \in \mathbb{D}_\alpha \cup \textit{kinderen}(\alpha), \ \textit{waarbij } \mathbb{D}_\gamma = \{\eta \mid \eta \prec_e \gamma\}.$$

Deze stelling zegt ons dat alle kinderen van een niet-minimale code reeds ontdekt moeten zijn als we $\mathbb{T}$ in pré-orde doorlopen. Een andere manier om dit op te vatten is dat de kinderen van een niet-minimale code ook niet-minimaal zijn. Merk op dat dit hetzelfde is als zeggen dat de canonische (=minimale) objecten allemaal gegenereerd kunnen worden uit andere canonische objecten. Herinner dat dit een orderly-voorwaarde was!

Bewijs. Neem voor $\beta = (b_0, \ldots, b_n)$ en $\alpha = (a_0, \ldots, a_n)$, waarbij $\alpha \prec_e \beta$ en dus $\alpha \in \mathbb{D}_\beta$. Wanneer we aan de code β een nieuwe boog e toevoegen, om een kind te bekomen, verkrijgen we een DFS-code $\beta' = (b_1, \ldots, b_n, e)$. We noemen de (ongeordende) graaf die hiermee wordt voorgesteld $G_{\beta'}$. We construeren nu via volgend algoritme een DFS-code uit α:

Algoritme 4: Construct $G_{\beta'}$

```
1  i = 1;
2  if a_0 > e then
3  |  vervolledig de graaf met bogen a'_1, ..., a'_{n+1};
4  |  return (e, a'_1, ..., a'_{n+1}) ;
5  end
6  else
7  |  while i ≤ n do
8  |  |  if (a_0, ..., a_{i-1}, e) is niet geldig of (a_0, ..., a_i) ≤ (a_0, ..., a_{i-1}, e)
   |  |  then
9  |  |  |  i++;
10 |  |  end
11 |  |  else
12 |  |  |  vervolledig de graaf met bogen a'_{i+1}, ..., a'_{n+1};
13 |  |  |  return (a_0, ..., a_{i-1}, e, a'_{i+1}, ..., a'_{n+1}) ;
14 |  |  end
15 |  end
16 end
```

Dit algoritme bouwt stap per stap een code voor de graaf $G_{\beta'}$ op, gebaseerd op α. In elke stap wordt een nieuwe boog uit α bijgevoegd, en wordt deze vergeleken met de oude code waarin e op het einde werd toegevoegd. Merk op dat e niet letterlijk wordt overgenomen, de knoopnummers hangen af van de plaats in de boom, de labels blijven echter wel hetzelfde. Op deze manier kijken we waar e in α kan worden toegevoegd, zodat we een kleinere code dan α verkrijgen. Indien we zo een positie vinden, voegen we nieuwe bogen toe op zo een manier dat de rest van de graaf "hersteld" wordt. We kunnen dit bekijken als het achtereenvolgens kiezen van bogen uit de graaf, zodat deze een boom vormen met een kleinere code (of een kind).

We verkrijgen dus twee mogelijke vormen van codes:

- $\alpha_e = (a_0, \ldots, a_{i-1}, e, a'_{i+1}, \ldots, a'_{n+1})$, of
- $\alpha_e = (a_0, \ldots, a_n, e)$.

Het is duidelijk dat de tweede vorm een kind is van α. Wegens de constructie van de eerste vorm weten we dat de code kleiner is dan die van α, waaruit we mogen concluderen dat $\alpha_e \in \mathbb{D}_\alpha$. We bekomen dus $\alpha_e \in \mathbb{D}_\alpha \cup \text{kinderen}(\alpha)$. Omdat $\alpha < \beta$ geldt ook

$$\alpha_e \in \mathbb{D}_\alpha \cup \text{kinderen}(\alpha) \subseteq \mathbb{D}_\beta.$$

□

We weten dat alle kinderen van een niet-minimale DFS-code ook niet minimaal zijn. We tonen nu aan dat hetzelfde geldt voor alle afstammelingen ervan, met als doel aan te tonen dat $\mathbb{T}$ na het *prunen* nog steeds alle minimale codes bevat.

Stelling 3.34. *Beschouw een graaf G en zijn DFS-codes in $\mathbb{T}$: $\alpha_0, \alpha_1, \ldots, \alpha_n$, waarbij $\forall i, j \leq n, \alpha_i \leq \alpha_j$ (volgens DFS lexicografische orde). Hierbij is $\alpha_0 = min(G)$. Als we nu $\alpha_i, 1 \leq i \leq n$* prunen *samen met al hun afstammelingen, dan voldoet de resulterende boom nog steeds aan eigenschap 3.30.*

Bewijs. Neem een code β die niet minimaal is. We bewijzen de volgende eigenschap.

$$\forall \delta \in \text{afstammelingen}(\beta) : \\ \min(\delta) \in \mathbb{D}_\beta \Rightarrow \forall \gamma \in \text{kinderen}(\delta) : \min(\gamma) \in \mathbb{D}_\beta. \quad (3.16)$$

Als deze eigenschap geldt, weten we namelijk dat, wanneer de nakomelingen van β reeds ervoor gegenereerd zijn, dat dit dan ook het geval is voor hun kinderen. Dit laat ons toe om alle nakomelingen van β te schrappen door simpelweg β niet uit te breiden.

Om de eigenschap aan te tonen maken we gebruik van inductie. Een nakomeling δ van β waarvan $\min(\delta) \prec_e \beta$ heeft volgens stelling 3.33 enkel kinderen waarvan de minimale code voor de minimale van β komt. Omdat de minimale DFS-code van β voor β zelf komt, weten we dat deze minima dus ook in $\mathbb{D}_\beta$ zitten. Merk op dat we eenzelfde redenering kunnen maken voor de kinderen hiervan, hetgeen bewijst dat de eigenschap geldt. □

Alvorens over te gaan tot het gSpan-algoritme, tonen we nog aan dat het aantal DFS-codes van een graaf, die een kind zijn van een minimale code, begrensd kan worden tot het product van zijn knopen en bogen.

Stelling 3.35. *beschouw een graaf G, waarbij $|E(G)| \geq 2$ en definieer $\mathcal{Z}(G) = \{\beta \mid \beta \in Z(G)$ en $\exists \alpha : min(\alpha) = \alpha$ en $\beta \in children(\alpha)\}$, dan geldt*

$$|\mathcal{Z}(G)| < |E(G)| \times |V(G)|. \quad (3.17)$$

Bewijs. Veronderstel dat $|E(G)| = m \geq 2$, dan weten we dat er maximaal m niet-isomorfe subgrafen van G bestaan die elk $m-1$ bogen bevatten. We bekomen deze grafen door uit Gelke mogelijke boog weg te nemen. Elk van deze subgrafen kan eventueel een minimale code zijn, dus er zijn maximaal m minimale codes die een parent van G zijn. Als de verwijderde boog een *forward edge* is ten opzichte van de ouder, dan zijn er maximaal $|V(G)| - 1$ (het aantal knopen van de ouder) mogelijkheden waarop deze boog terug toegevoegd kan worden. Er kan namelijk vanuit elke knoop geldige boog bestaan, bijvoorbeeld als de parent een complete graaf is.

Wanneer de verwijderde boog een *back edge* is, dan moet de boog "groeien" vanuit de meest-rechtse knoop naar één van de anderen. Omdat we geen boog van een knoop naar zichzelf toestaan, alsook geen twee bogen tussen één paar knopen, kunnen we maar op $|V(G)| - 2$ mogelijke manieren een boog toevoegen. We bekomen dus

$$m \times (|V(G)| - 1) + m \times (|V(G)| - 2).$$

Merk op dat dit niet geheel correct is, omdat de verwijderde boog ofwel een *forward edge* is, ofwel een *back edge*, maar nooit beide. De volgende formule geeft dus een betere indicatie:

$$c \times m \times (|V(G)| - 1) + (1 - c) \times m \times (|V(G)| - 2),$$

waarbij $0 \leq c \leq 1$ en c de frequentie van de minimale subgrafen aangeeft waarbij de verwijderde boog een *forward edge* was. We bekomen dus volgende grens:

$$\mathcal{Z}(G) \leq c \times |E(G)| \times (|V(G)| - 1) + (1 - c) \times |E(G)| \times (|V(G)| - 2).$$

We kunnen vervolgens een afschatting bekomen die een iets ruwere bovengrens geeft:

$$\begin{aligned} \mathcal{Z}(G) &< c \times |E(G)| \times |V(G)| + (1 - c) \times |E(G)| \times |V(G)| && (3.18) \\ &= |E(G)| \times |V(G)|. && (3.19) \end{aligned}$$

Dit geeft aan dat

$$\mathcal{Z}(G) < |E(G)| \times |V(G)|. \quad (3.20)$$

□

3.3.8 Het algoritme

In deze sectie doen we het gSpan-algoritme uit de doeken, we lichten de gebruikte procedures één voor één toe.

GraphSet_Projection

Het algoritme wordt aangeroepen met een verzameling van grafen $\mathbb{GS}$ en wordt weergegeven in Algoritme 5.

Stap 1 We nemen alle labels van knopen en bogen die voorkomen in $\mathbb{GS}$. Deze sorteren we volgens frequentie (voorkomen in $\mathbb{GS}$) en we verwijderen meteen de infrequente omdat deze toch geen bijdrage meer kunnen leveren. Als we een graaf zouden genereren waarin zo een infrequent label voorkomt, kan deze toch nooit frequent zijn wegens

Algoritme 5: GraphSet_Projection

```
Input: 𝔾𝕊, MinSup
Output: S
Sorteer de labels van de knopen en de bogen in 𝔾𝕊volgens frequentie;
Verwijder infrequente knopen en bogen;
Neem voor de orde op de labels de dalende frequentie;
S_1 = alle frequente grafen met één boog;
S = S ∪ S_1;
foreach boog e ∈ S_1 do
    s = e;
    s.GS = {G ∈ 𝔾𝕊 | e ∈ E(G)};
    Subgraph_Mining(𝔾𝕊,s);
    𝔾𝕊 = 𝔾𝕊 − e;
    if |𝔾𝕊| < MinSup then
        break;
    end
end
```

eigenschap 3.32. Vervolgens herdefiniëren we de knopen volgens afnemende frequentie en stoppen alle frequente *1-boog grafen* in S_1. Hierna sorteren we S_1 volgens DFS lexicofrafische orde en stoppen alle grafen in S. De verzameling S_1 zullen we in de volgende stappen gebruiken om reeds bestaande grafen uit te breiden.

Stap 2 Voor elke boog in S_1 zoeken we uit welke grafen in $\mathbb{GS}$ deze boog bevatten en onthouden deze verzameling. Hierna gaan we alle grafen die afstammen van deze boog genereren met de procedure *Subgraph_Mining*.

Stap 3 Nadat alle afstammelingen gegenereerd zijn, verwijderen we deze boog uit *alle* grafen in $\mathbb{GS}$. Dit is toegelaten omdat we alle frequente grafen met de gekozen boog (noem deze e) gegenereerd hebben. Het is dus niet meer mogelijk dat frequente grafen die nog gegenereerd worden door te vertrekken van een andere boog, de boog e bevatten.

Stap 4 Als het aantal grafen in $\mathbb{GS}$ kleiner is geworden dan de minimale frequentie, dan heeft het geen nut meer om verder te doen, aangezien geen enkele subgraaf een frequentie kan halen die groot genoeg is.

Subgraph_mining

In Algoritme 6 wordt de procedure `Subgraph_mining` weergegeven.

Stap 1 De procedure *Subgraph_Mining* kijkt eerst of we te maken hebben met een minimale code. Indien dit niet het geval is, dan weten we wegens stelling 3.33 en 3.34 dat we deze knoop in de zoekruimte niet verder moeten uitwerken en hem dus mogen *prunen*. Indien het wel een minimale code is, dan weten we dat we te maken hebben met een frequente

Algoritme 6: Subgraph_mining

```
1  if s ≠ min(s) then
2  |  return;
3  end
4  Voeg s toe aan S;
5  Genereer alle geldige kinderen van s door achteraan in de code een boog bij
   te voegen;
6  Enumerate(s);
7  foreach c ∈ kinderen(s) do
8  |  if support(c) ≥ MinSup then
9  |  |  Subgraph_Mining(𝔾𝕊,s);
10 |  end
11 end
```

graaf (we eisen dat deze procedure *nooit* aangeroepen wordt voor een infrequente subgraaf) en voegen we hem dus toe aan S (de verzameling frequente subgrafen).

Stap 2 De volgende stap bestaat eruit om alle geldige kinderen van s (de meegegeven subgraaf) te genereren en met behulp van de procedure *Enumerate* te bepalen in welke grafen deze kinderen voorkomen.

Stap 3 Na het construeren van de kinderen, kunnen we eenvoudig testen welke voldoen aan de minimale frequentie: we tellen het aantal grafen uit $\mathbb{GS}$ waarmee deze code geassocieerd is. We roepen de procedure recursief aan voor de juiste kinderen.

Enumerate

In Algoritme 7 wordt de procedure `Enumerate` weergegeven.

Algoritme 7: Enumerate

```
1  foreach G ∈ s.GS do
2  |  𝔾_s = de voorkomens van s in G;
3  |  foreach c ∈ kinderen(s) : c ⊆ G do
4  |  |  foreach H ∈ G_s do
5  |  |  |  if c ⊆ H then
6  |  |  |  |  c.GS = c.GS ∪ {G};
7  |  |  |  end
8  |  |  end
9  |  end
10 end
```

Stap 1 De procedure *Enumerate* loopt voor elke graaf G uit $\mathbb{GS}$ waarin s voorkomt alle voorkomens van s in G af en kijkt hoe we s geldig kunnen

uitbreiden zodat de resulterende graaf ook een subgraaf van G is. Voor deze uitbreiding (die een kind van s is) slaan we op in welke graaf hij voorkomt. Als een voorkomen van s in G alle mogelijke geldige kinderen heeft gegenereerd, dan moeten we de andere voorkomens van s in G niet meer beschouwen, aangezien ze geen bijdrage meer kunnen leveren aan de geldige kinderen van s.

3.3.9 Pruning van niet-minimale codes

In het algoritme worden niet-minimales codes niet verder uitgewerkt, maar *gepruned*. Op deze manier vermijden we dubbele berekeningen (van reeds beschouwde subgrafen) door onze zoekruimte te beperken tot het noodzakelijke. In het algemeen kunnen we dit op twee manieren oplossen: *Pre-pruning* houdt in dat we zodra een kind gegenereerd is testen of het een canonisch kind is. Deze manier is zeer kostelijk, omdat de veel van de deze codes vaak niet frequent zijn. Daar de *min*-opratie zeer complex is, kunnen we in vele gevallen beter de frequentie-test doen en daarna pas de *min*-test, dit noemen we *post-pruning*. Maar nu blijven we echter wel achter met het tellen van de voorkomens voor dubbele codes, hetgeen ons een groot aantal nutteloze berekeningen oplevert. Om deze reden komen we tot een hybride aanpak, die in vier stappen gebeurt:

Stap 1 We beschouwen een code s, waarbij we de eerste boog in de code aanduiden met e_0. We weten nu dat een mogelijk kind van s geen boog bevat die kleiner is dan e_0, indien dit wel zo was, dan zou een uitbreiding van deze kleinere boog reeds vroeger gebeurd zijn en zou deze graaf al ontdekt geweest zijn. (Herinner dat e_0 een boog uit S_1 moet zijn, omdat we enkel deze bogen uitbreiden, en omdat uitbreidingen *achteraan* de code gebeuren).

Stap 2 Voor elke *back edge* (v_i, v_j), met $i < j$, die we toevoegen aan s, mag er geen grotere boog zijn (volgens de DFS lexicografishe orde), die verbonden is met v_j. Indien dit wel het geval is, dan wordt de *minimale* DFS-code van het nieuwe kind kleiner dan die van zijn ouder en is deze bijgevolg al eens gegenereerd.

Stap 3 Bogen groeien enkel vanuit het *meest-rechtse pad*, dit levert ons minder kinderen op dan wanneer we alle booguitbreidingen zouden beschouwen.

Stap 4 Op alle overige codes wordt *post-pruning* toegepast.

3.3.10 $min(s)$ berekenen

Een logische manier waarop we $min(s)$ zouden berekenen, is om alle mogelijke DFS-codes voor s te beschouwen en hieruit de kleinste te nemen. Een

betere manier is echter om te beginnen met het genereren van mogelijk minimale codes, en zodra we al een deel van de code berekend hebben, deze te vergelijken met s. Indien de gedeeltelijke code groter is dan s, dan mogen we een volgende voorstelling van de graaf beschouwen, omdat deze groter zal zijn dan s, en dus ook niet minimaal is. Indien de gedeeltelijke code kleiner is dan s, dan weten we dat s niet minimaal is en moeten we niet verderzoeken. Op deze manier vermijden we de generatie van alle mogelijke codes voor een bepaalde graaf, en is het verkrijgen van de echte minimale code geen noodzakelijkheid. We gaan in essentie op zoek naar een voorstelling van de graaf die "bewijst" dat s niet minimaal is.

3.3.11 Genereren van kinderen

In de procedure *Subgraph_Mining* moeten we alle geldige kinderen van s berekenen. In deze sectie introduceren we drie mogelijke methoden die dit kunnen doen:

1. We genereren alle geldige kinderen uit s en gaan vervolgens op zoek naar de voorkomens in $\mathbb{GS}$. In dit geval is het mogelijk dat sommige kinderen niet voorkomen, en de "tellers" ervan op 0 blijven staan.

2. Deze aanpak vermijdt het vooraf genereren van kinderen, in plaats daarvan gaan we in de grafen uit $\mathbb{GS}$ waarin s voorkomt, kijken naar de mogelijke geldige kinderen van s. Indien we een kind vinden dat overeenkomt met een geldige code, dan voegen we dit toe aan de lijst en zetten zijn teller op 1. Het is duidelijk dat er nu geen kinderen gegenereerd worden die nergens in $\mathbb{GS}$ voorkomen.

3. De laatste methode maakt gebruik van eigenschap 3.32: we gaan kijken naar samenstellingen met buren om te concluderen of kinderen infrequent gaan zijn. We weten bijvoorbeeld dat als $(e_0, e_1, \ldots, e_x, x_y)$ frequent is, dat dan ook $(e_0, e_1, \ldots, e_x)$ en $(e_0, e_1, \ldots, x_y)$ frequent zijn (indien ze geldig zijn). Omgekeerd weten we dan ook dan indien $(e_0, e_1, \ldots, e_x)$ en/of $(e_0, e_1, \ldots, x_y)$ niet frequent zijn (op voorwaarde dat ze wel geldig zijn), dat dan $(e_0, e_1, \ldots, e_x, x_y)$ ook niet frequent is, en we deze mogen schrappen als kind, ook al is het een minimale code!

3.3.12 Orderly generatie en gSpan

We hebben eerder al de voorwaarden voor het generisch orderly algoritme ingevuld om grafen op m knopen te genereren, waarbij m een constante is. gSpan geeft ons een algoritme dat deze beperking niet heeft, en we tonen nu aan dat dit algoritme in essentie een orderly algoritme is, waaraan wat extra's zijn toegevoegd om *graph mining* mogelijk te maken. We tonen voor elk van de drie "orderly" voorwaarden aan dat ze gelden:

Stelling 3.36. *Elke minimale DFS-code kan gegenereerd worden uit een kleinere minimale DFS-code.*

Bewijs. We hebben reeds aangetoond in stelling 3.33 en 3.34 dat minimale codes geen minimale nakomelingen hebben. Dit is hetzelfde als zeggen dat minimale codes geen kind kunnen zijn van een niet-minimale code. De enige optie die dan overblijft is dat minimale codes kinderen zijn van andere minimale codes. □

Stelling 3.37. *De functie $f : \mathcal{L}_{q+1} \to \mathcal{L}_q$ is zwak monotoon.*

Bewijs. Neem twee codes $\alpha = (a_1, \ldots, a_n)$, $\beta = (b_1, \ldots, b_m)$ zodat $\alpha \prec_e \beta$. Omdat α kleiner is, is deze ofwel een kortere versie van β, ofwel geldt:

$$\exists i \in [1, n] : (a_1, \ldots, a_{i-1}) = (b_1, \ldots, b_{i-1}) \wedge a_i \prec_e b_i.$$

We beschouwen deze twee gevallen apart:

Geval 1 Als we de ouder van de twee codes berekenen, komt dit overeen met het weghalen van de laatste boog in de code. De overige bogen worden niet aangetast en α zal dus nog steeds een kortere versie van β zijn. Hierdoor geldt dat $f(\alpha) \prec_e f(\beta)$.

Geval 2 Als $i = n = m$, dan verkrijgen we twee identieke codes, waardoor $f(\alpha) = f(\beta)$. Als er een verschil is voor de laatste boog, dan zal dit verschil blijven bestaan na het verwijderen van deze laatste. We bekomen dus opnieuw $f(\alpha) \prec_e f(\beta)$.

□

Stelling 3.38. *De groei-operatie φ genereert de kinderen van een graaf volgens de orde $\prec_e$.*

Bewijs. Dit geldt wegens de constructie van de groei-operatie. □

Merk op dat in het gSpan-algoritme geen controle wordt gedaan op de orde, zoals bij orderly generatie. Deze controle is hier niet nodig, omdat het onmogelijk is om dubbels te verkrijgen, elke code is immers uniek. Het groeiproces garandeert overigens ook dat er geen kleinere codes meer zullen volgen.

Het is duidelijk te zien dat de manier waarop de encodering van de grafen analoog is aan die van de gewortelde bomen. Vooral de uitbreidingsoperatie werkt op dezelfde manier: het rechtse pad wordt gebruikt om er zeker van de zijn dat we alle objecten kunnen genereren.

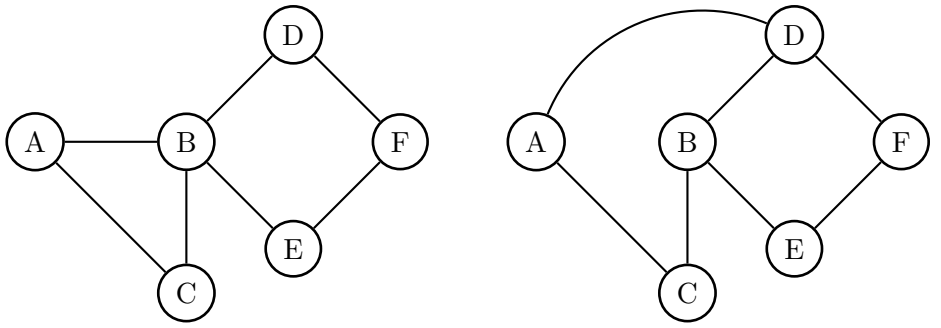
Een verschil dat we hier echter bij opmerken is dat we *alle* objecten in volgorde genereren, niet enkel de objecten van een bepaald niveau. Dit komt door de uitbreidingsoperatie die gSpan gebruikt.

3.3.13 Samenvatting

Het is duidelijk dat gSpan een orderly algoritme is, het voldoet aan alle gestelde voorwaarden. gSpan concentreert zich echter ook op het zoeken van patronen in een database, niet enkel op het genereren van grafen.

De voorstelling van een graaf bestaat uit een opeenvolging van bogen, die in essentie overeenkomt met het doorlopen van de graaf "als een boom". De gebruikte DFS-codes laten toe om een uitbreiding van de graaf te definiëren die veel beperkter is dan wanneer we alle mogelijke kinderen beschouwen: er worden enkel uitbreidingen van het *meest-rechtse pad* beschouwd.

Omdat het algoritme een oplossing is voor het *graph mining* probleem, kunnen er optimalisaties ingevoerd worden die hiermee samenhangen. Zo kunnen we onder andere zeggen dat als een graaf niet frequent is, dat geen enkele van zijn nakomelingen dit zal zijn. Dit soort van *pruning* snijdt een heel stuk van de zoekruimte weg, hetgeen natuurlijk leidt tot snelheidswinst. Verder kunnen we de canoniciteitstest en de frequentietest op een plaats zetten die ons in het algemeen tijd bespaart.



(a) een graaf *mét* een triangle, dus geen TFG

(b) een graaf *zonder* een triangle, een TFG

Figuur 3.17: TFG's.

3.4 McKay's Framework

3.4.1 Inleiding

In deze sectie gaan we dieper in op een alternatieve manier om grafen te genereren, op zo een manier dat er nooit "isomorfe dubbels" gegenereerd worden. Dit algoritme wordt beschreven door McKay [24], hij geeft een abstract framework voor het genereren van combinatorische objecten.

We volgen, net zoals in McKay's paper, een algemeen en een specifiek "pad" in de uitleg. Het algemene pad zal handelen over algemene objecten, terwijl het specifieke deel toegespitst is op grafen. Dit geeft al dadelijk aan dat dit algoritme niet enkel van toepassing is op grafen, maar ook zijn nut heeft bij het genereren van andere objecten. Het deel over grafen zal gaan over het genereren van *triangle-free graphs* of TFG's, dit zijn grafen die geen 'driehoeken' bevatten. Onder een driehoek verstaan we een cykel in de graaf die uit drie knopen bestaat: drie knopen die onderling verbonden zijn. In Figuur 3.17a zien we een graaf waarbij de knopen A, B en C een cykel vormen, deze graaf is dus niet driehoeksvrij. De graaf in Figuur 3.17b daarentegen bevat geen substructuur van die aard, en is hierom dus een *triangle-free graph* of een TFG.

3.4.2 Constructies en algoritme

We gaan van start met een (mogelijk oneindige) verzameling *geordende objecten* en noemen deze $\mathcal{L}$. Verder beschouwen we nog een permutatie groep $\mathcal{G}$ die werkt op $\mathcal{L}$. De orbits van $\mathcal{G}$ in $\mathcal{L}$ noemen we *ongeordende objecten* en de verzameling van al deze ongeordende objecten noemen we $\mathcal{U}$. Een object

$X \in \mathcal{L}$ heeft een orde $o(X) \in N$, en een orbit $S \in \mathcal{U}$ bevat enkel objecten van dezelfde orde. Dit laatste laat ons toe het begrip *orde* te definiëren op objecten uit $\mathcal{U}$ als de gemeenschappelijke grootte van de objecten in de orbit: $o(S) = o(X)$, waarbij $X \in S$.

Voorbeeld 3.39. *Als we dit bekijken in de context van TFG's dan komt een geordend object overeen met een graaf. Hier is $\mathcal{L}$ de verzameling van alle grafen, waarbij $o(X) \in \mathbb{N}$, met $X \in \mathcal{L}$. De verzameling $\mathcal{U}$ bevat in dit geval orbits van grafen.*

Als we voor $\mathcal{G}$ de verzameling met alle *mogelijke permutaties nemen, dan bevat de orbit van een graaf alle grafen die ermee isomorf zijn. Meer formeel kunnen we zeggen dat $\mathcal{G} = S_1 \times S_2 \times \cdots$, met S_i de symmetrische groep van graad i. Dit is analoog aan de manier waarop we dit in sectie 2.3 gedaan hebben.*

Merk op dat zo een orbit overeenkomt met een "ongeordende graaf" of een "abstracte graaf": een graaf waarbij geen orde op de knopen gedefinieerd is. Een ongeordende graaf kan gezien worden zoals in Figuur 3.18a, waarbij er geen orde op de knopen ligt. De grafen in de figuren 3.18b en 3.18c bezitten daarentegen wel een orde op hun knopen. We kunnen een ongeordend object ook zien als een verzameling van grafen die isomorf zijn, met andere woorden een orbit. Merk op dat we aan een algoritme geen ongeordende objecten kunnen meegeven, zoals reeds vermeld in sectie 2.4.

Nu we gedefinieerd hebben met wat voor objecten we werken, is het tijd om ons probleem te beschrijven. Hetgeen we willen bereiken is het volgende: voor elk ongelabeld object precies één gelabeld object produceren (een concrete voorstelling ervan), en dit voor alle objecten tot een bepaalde orde n.

Om hiertoe te komen zullen we een aantal extra verzamelingen associëren met een gelabeld object $X \in \mathcal{L}$

- $L(X)$: de *lower objects* van X,
- $U(X)$: de *upper objects* van X.

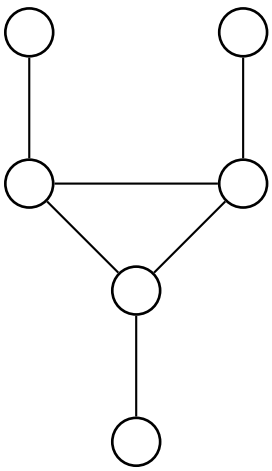
Volgende voorwaarden zijn van toepassing:

Eigenschap 3.40.

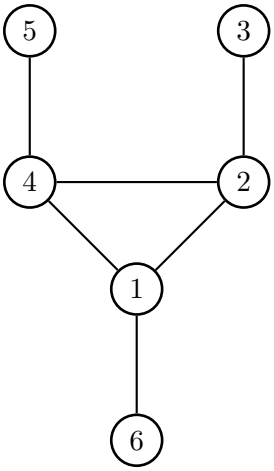
$$\forall X_1, X_2 \in \mathcal{L} : X_1 \neq X_2 \Rightarrow$$
$$\{X_1\}, L(X_1), U(X_1), \{X_2\}, L(X_2), U(X_2) \text{ zijn onderling disjunct.} \quad (3.21)$$

Eigenschap 3.41. *Er geldt dat*

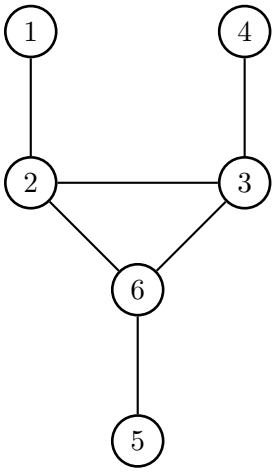
- $\forall Y \in L(X) : o(Y) = o(X)$,
- $\forall Y \in U(X) : o(Y) = o(X)$.



(a) een ongeordende graaf



(b) een geordende graaf



(c) een andere geordende graaf

Figuur 3.18: Geordende en ongeordende grafen.

De laatste voorwaarde zegt ons dat de orde van objecten uit $L(X)$ en $U(X)$ gelijk is aan die van X. De betekenis van deze verzamelingen zal in hetgeen volgt verduidelijkt worden. We definiëren vervolgens nog twee grotere verzamelingen:

$$\check{\mathcal{L}} = \bigcup_{X \in \mathcal{L}} L(X), \tag{3.22}$$

alle *lower objects* en

$$\hat{\mathcal{L}} = \bigcup_{X \in \mathcal{L}} U(X), \tag{3.23}$$

alle *upper objects*.

We veronderstellen dat er een relatie $R_f \subseteq \check{\mathcal{L}} \times \hat{\mathcal{L}}$ bestaat, samen met een permutatie groep $\mathcal{G}$, die werkt op $\mathcal{L} \cup \check{\mathcal{L}} \cup \hat{\mathcal{L}}$. Om naar de relatie R_f te kunnen verwijzen op een eenvoudige manier, decomponeren we deze in twee functies:

$$f(\check{Y}) = \{\hat{X} \in \hat{\mathcal{L}} \mid (\check{Y}, \hat{X}) \in R_f\}, \tag{3.24}$$

$$f'(\hat{X}) = \{\check{Y} \in \check{\mathcal{L}} \mid (\check{Y}, \hat{X}) \in R_f\}. \tag{3.25}$$

Merk op dat de functie f alle *upper objects* teruggeeft die in relatie staan met een bepaald *lower object*. De functie f' doet precies het omgekeerde. Verder gebruiken we symbolen waaraan tekens zijn toegevoegd, zoals $\check{X}$ en $\hat{X}$, om duidelijk te maken wanneer we over *lower objects* of *upper objects* spreken. De relatie R_f en de groep $\mathcal{G}$ moeten voldoen aan volgende axioma's:

Eigenschap 3.42. *C1* $\mathcal{G}(\mathcal{L}) = \mathcal{L}$, $\mathcal{G}(\check{\mathcal{L}}) = \check{\mathcal{L}}$ *en* $\mathcal{G}(\hat{\mathcal{L}}) = \hat{\mathcal{L}}$.

C2 $\forall X \in \mathcal{L} : \forall g \in \mathcal{G} : L(X^g) = L(X)^g \wedge U(X^g) = U(X)^g$.

C3 $\forall \check{Y} \in \check{\mathcal{L}} : f(\check{Y}) \neq \emptyset$.

C4 $\forall \check{Y} \in \check{\mathcal{L}}, \forall g \in \mathcal{G}, \forall \hat{X}_1 \ \in f(\check{Y}), \forall \hat{X}_2 \in f(\check{Y}^g), \exists h \in \mathcal{G} : \hat{X}_1^h = \hat{X}_2$.

C5 $\forall \hat{X} \in \hat{\mathcal{L}}, \forall g \in \mathcal{G}, \forall \check{Y}_1 \in f'(\hat{X}), \forall \check{Y}_2 \in f'(\hat{X}^g), \exists h \in \mathcal{G} : \check{Y}_1^h = \check{Y}_2$.

C6 $\forall X \in \mathcal{L}, \forall g \in \mathcal{G} : o(X^g) = o(X)$.

C7 $\forall \check{Y} \in \check{\mathcal{L}}$ *en*$\hat{X} \ \in f(\check{Y}) : o(\hat{X}) < o(\check{Y})$.

Om de betekenis hiervan iets duidelijker te maken, geven we voor elk axioma een korte toelichting:

C1 Het toepassen van $\mathcal{G}$ op een verzameling kunnen we zien als het toepassen van een element uit $\mathcal{G}$ op elk element van deze verzameling. Dit axioma houdt in dat, indien we eenzelfde $g \in \mathcal{G}$ toepassen op *alle* elementen van de verzameling $\mathcal{L}$, dat we dan terug al de elementen bekomen uit $\mathcal{L}$. Hetzelfde geldt voor de twee andere verzamelingen. We zeggen dan dat $\mathcal{L}$, $\check{\mathcal{L}}$ en $\hat{\mathcal{L}}$ bewaard blijven onder $\mathcal{G}$.

C2 De betekenis van X^g is de permutatie g toegepast op het object X[4]. Dit axioma houdt in dat wanneer we een permutatie uit $\mathcal{G}$ toepassen op een gelabeld object en hiervan de *lower objects* (resp. *upper objects*) berekenen, dat we dan dezelfde verzameling bekomen dan wanneer we eerst de *lower objects* (resp. *upper objects*) berekenen en vervolgens permuteren.

C3 Elk *lower object* staat in relatie met minstens één *upper object*.

C4 Elementen van $f(\check{Y})$ zijn equivalent met mekaar onder $\mathcal{G}$, of anders gezegd, ze zijn isomorf. Hetgeen dit betekent kunnen we inzien door voor g de identieke permutatie te kiezen. In dat geval moet er een h bestaan, zodat $\hat{X}_1^h = \hat{X}_2$. Elk element uit $f(\check{Y})$ moet dus op een element uit deze verzameling afgebeeld kunnen worden. Hetzelfde kunnen we doen voor de f-verzamelingen van isomorfe grafen: de elementen hieruit moeten onderling ook allemaal op elkaar afgebeeld kunnen worden. Merk op dat het geen vereiste is dan $f(\check{Y})$ een orbit vormt.

C5 Analoog aan het vorige axioma.

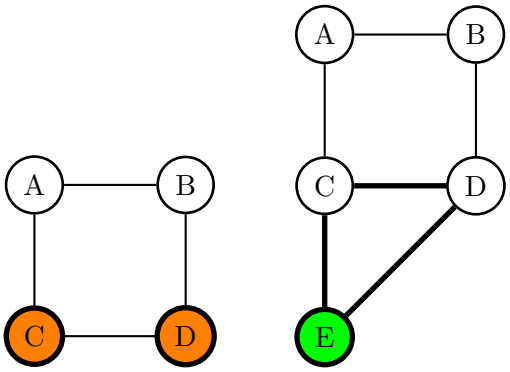
C6 De orde van een gelabeld object blijft behouden onder een permutatie uit G.

C7 Elk *lower object* staat enkel in relatie met *upper objects* die van een lagere orde zijn

Voorbeeld 3.43. *Bij TFG's kunnen we ook* lower objects *en* upper objects *associëren met de grafen. De* lower objects *zijn van de vorm* $\langle X, v\rangle$*, waarbij* X *een (geordende) graaf is, en* $v \in V(X)$*. Het* lower object *geeft aan hoe we van een graaf een kleinere graaf kunnen maken, door een knoop* v *te verwijderen.* $L(X)$ *bevat al deze (graaf,knoop)-paren.*

De upper objects *zijn van de vorm* $\langle X, W\rangle$*, waarbij* X *opnieuw de oorspronkelijke graaf is, en* $W \subseteq V(X)$*. De verzameling van knopen* W *geeft aan hoe we de graaf* X *kunnen uitbreiden: we voegen een nieuwe knoop toe en verbinden deze met alle knopen in* W*. De verzameling* $U(X)$ *bevat al de mogelijke (graaf,knoop-deelverzameling)-paren. In het geval van TFG's zijn deze* W*'s beperkt tot deelverzamelingen van* $V(X)$ *die geen twee aanliggende knopen bevatten. Indien dit wel toegelaten zou zijn, dan zou de nieuwe knoop verbonden kunnen worden met twee knopen die al verbonden waren, hetgeen een driehoek zou introduceren in de graaf. In Figuur 3.19 zien we hoe dit zou kunnen gebeuren: we selecteren twee knopen in Figuur 3.19a die al met elkaar verbonden zijn. Dit* upper object *zal aanleiding geven tot de creatie van het* lower object *dat te zien is in Figuur 3.19b, waarin duidelijk een driehoek geïntroduceerd wordt.*

[4]Een andere gangbare notatie is $g(X)$



(a) *upper object* **(b)** *lower object*

Figuur 3.19: Introductie van een driehoek in een graaf door middel van uitbreiding van een *upper object* naar een *lower object*.

We merken op dat in de axioma's een permutatie g toegepast kan worden op een lower object *of een* upper object*. Om dit mogelijk te maken moeten we echter wel de betekenis van de toepassing van de permutatie op deze objecten definiëren, we doen dit als volgt:*

$$\langle X, v\rangle^g = \langle X^g, v^g\rangle, \tag{3.26}$$

$$\langle X, W\rangle^g = \langle X^g, W^g\rangle, \tag{3.27}$$

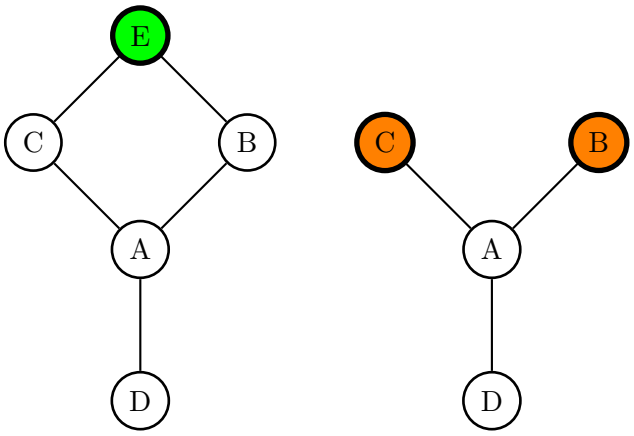
Nu we bepaald hebben hoe de lower objects *en* upper objects *opgebouwd zijn, kunnen we er een relatie tussen definiëren. Het is eenvoudig in te zien dat $\langle X, v\rangle \in L(X)$ in relatie staat met $\langle X - v, W\rangle$, waarbij W gelijk is aan de buren van v in X. Formeel bekomen we dus*

$$f : \langle X, v\rangle \mapsto \{\langle X - v, W\rangle^g | g \in \mathcal{G} \wedge W = N(v)\}. \tag{3.28}$$

In Figuur 3.19 is te zien hoe een zeker lower object *met een* upper object *gerelateerd is (en dus ook omgekeerd). De functie f' tussen de* upper objects *en de* lower objects *stelt een groeioperatie voor, de functie f een krimpoperatie. Het algoritme zal beginnen met een klein object, dat uitgebreid wordt tot grotere objecten, die op hun beurt ook weer uitgebreid worden. We zullen dus de functie f' nodig hebben.*

Tot slot vermelden we nog dat er bij grafen één object $X \in \mathcal{L}$ is waarbij $L(X) = \emptyset$, namelijk de triviale graaf K_1.

We zeggen (zoals eerder) dat twee geordende objecten $X_1, X_2 \in \mathcal{L}$ isomorf zijn als $\exists g \in \mathcal{G} : X_1^g = X_2$. Vervolgens definiëren we de *automorfisme groep* $Aut(X)$ voor een bepaalde $X \in \mathcal{L}$ als $Aut(X) = \{g \in \mathcal{G} \mid X^g = X\}$.



(a) *lower object* **(b)** *upper object*

Figuur 3.20: Relatie tussen *lower object* en *upper object*.

De automorfisme groep van X (ten opzichte van $\mathcal{G}$) bevat alle permutaties uit $\mathcal{G}$ die automorfismen van X zijn. Uit voorwaarde C2 kunnen we zien dat $\forall a \in Aut(X) : L(X)^a = L(X)$, dit omdat $\forall g \in \mathcal{G} : L(X^g) = L(X)^g$, $a \in Aut(X) \subseteq \mathcal{G}$ en wegens de definitie van een automorfisme geldt dat $L(X)^a = L(X^a) = L(X)$.

De volgende stap is om de *ongeordende objecten* onder te verdelen. We definiëren een *ongeordend object* $S \in \mathcal{U}$ als een *onreduceerbaar object* als $\forall X \in S : L(X) = \emptyset$ en als *reduceerbaar object* in het andere geval. We splitsen met andere woorden de *ongeordende objecten* op in objecten mét en zonder *lower objects*. We definiëren $\mathcal{U}_0$ als de verzameling van alle *onreduceerbare objecten* en $\mathcal{U}_1$ als de verzameling van alle *reduceerbare objecten*. Het is duidelijk dat

$$\mathcal{U} = \mathcal{U}_0 \cup \mathcal{U}_1. \tag{3.29}$$

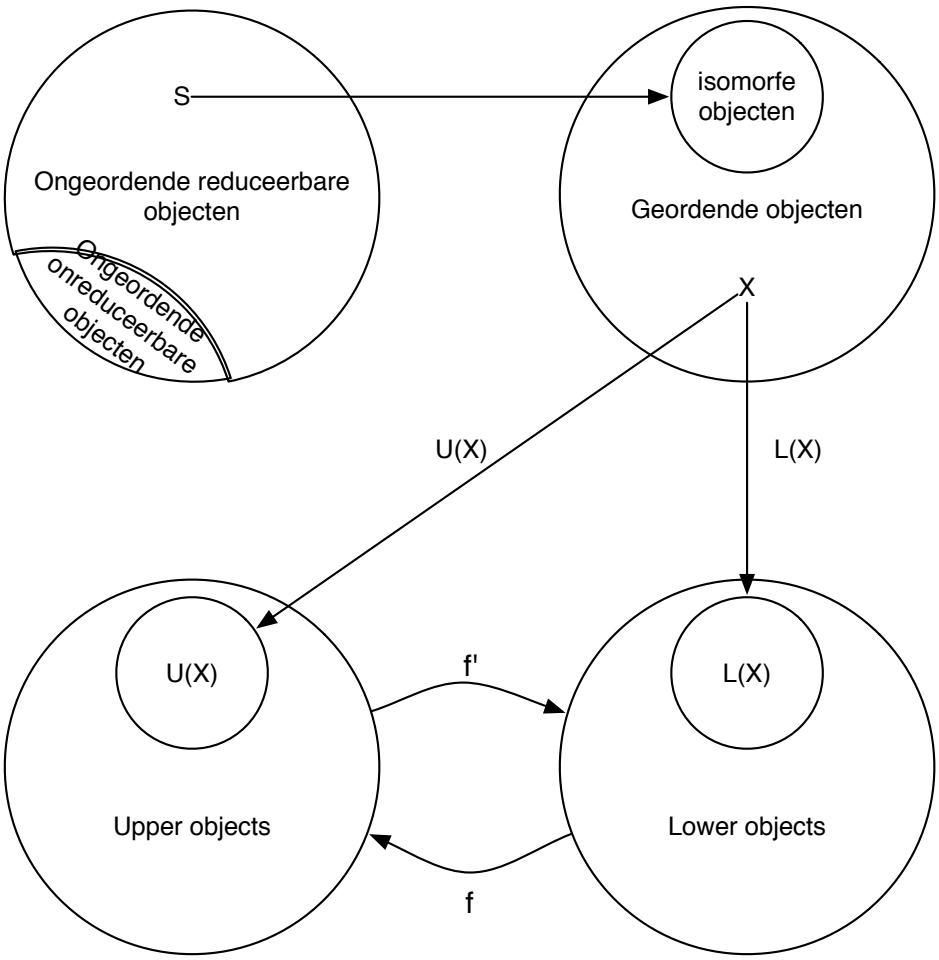
De constructies die we tot nu toe gedefinieerd hebben zijn te zien in Figuur 3.21.

Voorbeeld 3.44. *Bij TFG's hebben we slechts één* onreduceerbaar object*, namelijk de ongelabelde(!) graaf* K_1.

Tot slot hebben we een functie $m : \mathcal{L} \to 2^{\check{\mathcal{L}}}$ nodig die aan de voorwaarden vermeld in eigenschap 3.45 voldoet. De betekenis van $2^{\check{\mathcal{L}}}$ is de verzameling van alle deelverzamelingen van $\check{\mathcal{L}}$. De functie m beeldt met andere woorden een *geordend object* af op een verzameling *lower objects*.

Eigenschap 3.45. *De functie* m *voldoet aan volgende voorwaarden:*

M1 $\forall X \in \mathcal{L} : L(X) = \emptyset \Rightarrow m(X) = \emptyset$.



Figuur 3.21: Constructies volgens McKay [24].

M2 $\forall X \in \mathcal{L} : L(X) \neq \emptyset \Rightarrow m(X)$ *is een orbit van de toepassing van de automorfisme groep van* X *(dit is* $Aut(X)$*) op de* lower objects *van* X *(*$L(X)$*). Dit wil zeggen dat we een orbit kunnen maken door een een* lower object *te kiezen uit* $L(X)$ *en hier de automorfismen van* X *op toe te passen.*

M3 $\forall X \in \mathcal{L}, \forall g \in \mathcal{G} : m(X^g) = m(X)^g$*. Met andere woorden, de "m" van isomorfe objecten is dezelfde, maar dan met het isomorfisme op deze verzameling toegepast.*

Deze eigenschappen laten ons toe om volgend lemma aan te tonen:

Lemma 3.46. *Er bestaat een mapping* $p : \mathcal{U}_1 \to \mathcal{U}$ *met als eigenschap:*

$$\forall S \in \mathcal{U}_1, \forall X \in S, \forall \check{X} \in m(X), \forall \hat{Y} \in f(\check{X}) : \hat{Y} \in U(Y), \text{ waarbij } Y \in p(S). \tag{3.30}$$

De functie p *geeft dus voor een bepaalde orbit een andere orbit met een kleinere orde.*

Bewijs. We nemen een S uit $\mathcal{U}_1$ en kiezen een X_1, X_2 uit deze S. We kiezen vervolgens een willekeurige $\check{X}_i \in m(X_i)$, voor $i = 1, 2$. Hierna kiezen we (opnieuw willekeurig) een $\hat{Y}_i \in f(\check{X}_i)$, hetgeen inhoudt dat we *upper objects* kiezen die in relatie staan met de eerder gekozen *lower objects*.

Voor de gekozen *upper objects* $\hat{Y}_i$ geldt dat $\hat{Y}_i \in U(Y_i)$, voor $i = 1, 2$. We weten uit definitie 3.40 dat deze Y_i uniek is voor elk *upper object*, en kunnen dus de Y_i bepalen voor elke $\hat{Y}_i$.

Als we nu aantonen dat $Y_1 \cong Y_2$, dan weten we dat ze uit dezelfde $S' \in \mathcal{U}$ komen, omdat alle isomorfe grafen in dezelfde orbit zitten. Het lemma zegt ons dat voor alle $S \in \mathcal{U}$ geldt dat voor elke X in deze orbit een geassocieerd *upper object* een element is van $U(Y)$ voor een zekere $Y \in p(S)$. Voor verschillende X'en en $\check{X}$'en mag deze Y dus verschillen.

We zijn begonnen met het kiezen van een $X_1, X_2 \in S$, hetgeen wil zeggen dat $\exists g \in G$ zodat

$$X_1^g = X_2, \tag{3.31}$$

en in het bijzonder

$$m(X_1^g) = m(X_2). \tag{3.32}$$

Wegens M3 mogen we de permutatie naar buiten brengen om te komen tot

$$m(X_1)^g = m(X_2). \tag{3.33}$$

M2 zegt ons dat m *een* orbit teruggeeft van *lower objects* we weten dus wegens 3.33 dat $\exists l \in Aut(X_1)$, zodat voor de gekozen elementen $\check{X}_1, \check{X}_2$ geldt

$$\check{X}_2 = \check{X}_1^{l \circ g} = \check{X}_1^h, \text{ waarbij } h = l \circ g \wedge h \in G, \tag{3.34}$$

er geldt namelijk dat $(\mathcal{G}, \circ)$ een groep is en dus gesloten is onder samenstelling van functies $\circ$. We passen vervolgens f toe op $\check{X}_1$ en $\check{X}_2$ en kiezen

$$\hat{Y}_1 \in f(\check{X}_1), \hat{Y}_2 \in f(\check{X}_2). \tag{3.35}$$

Omdat $\check{X}_2 = \check{X}_1^h$ geldt $f(\check{X}_2) = f(\check{X}_1^h)$, en wegens C4 dus

$$\exists\ k \in\ G : \hat{Y}_1^k = \hat{Y}_2 \tag{3.36}$$

Stel nu dat $\hat{Y}_1 \in U(Y_1)$, dan geldt ook dat $\hat{Y}_1^k \in U(Y_1)^k$. Wegens eigenschap C2 geldt nu

$$\hat{Y}_1^k \in U(Y_1)^k = U(Y_1^k). \tag{3.37}$$

Verder weten we ook dat

$$\hat{Y}_1^k = \hat{Y}_2 \in U(Y_2), \tag{3.38}$$

hetgeen betekent dat zowel $U(Y_2)$ als $U(Y_1^k)$ $\hat{Y}_1^k$ bevatten, en wegens eigenschap 3.40 (disjunctie van de *upper objects* voor verschillende objecten) moet dus gelden dat

$$Y_2 = Y_1^k. \tag{3.39}$$

Dit laatste geeft aan dat $Y_2 \cong Y_1$. □

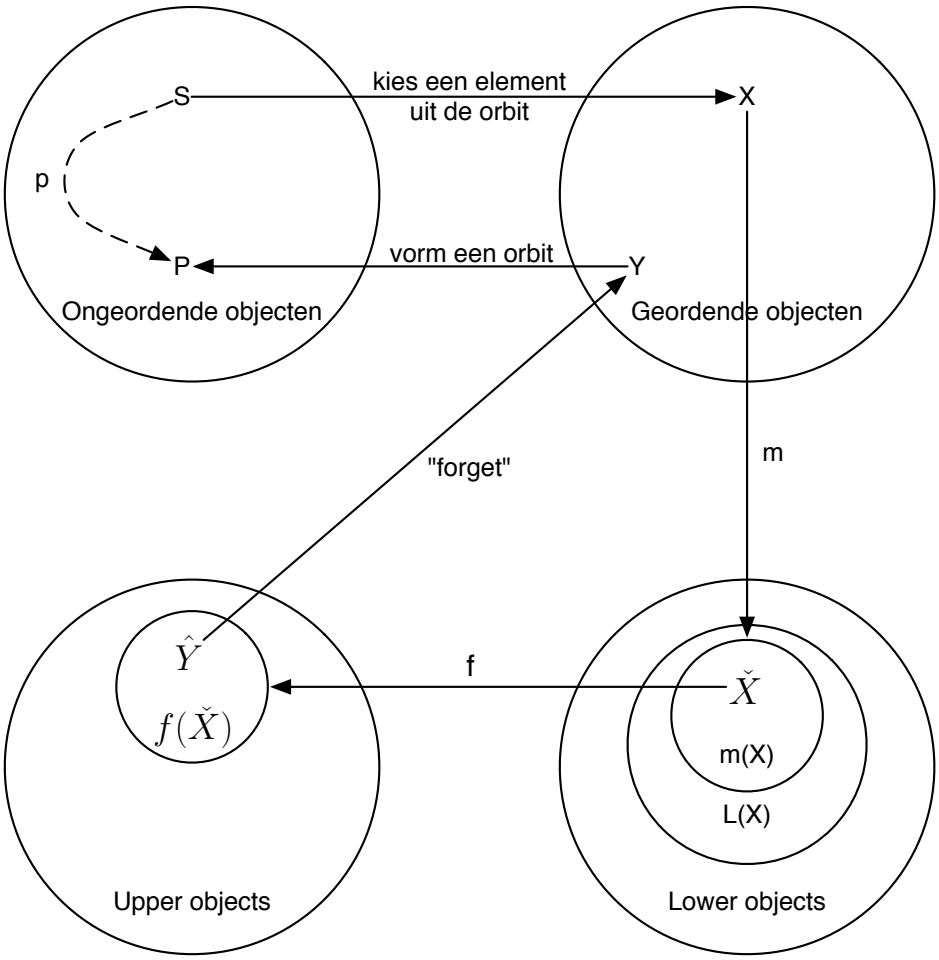
Voorbeeld 3.47. *Het construeren van zo een functie m die aan bovenstaande voorwaarden voldoet is zeer moeilijk. Een mogelijke $m(X)$ is de volgende: beschouw alle mogelijke knoopordes van een graaf X, en kies degene die het grootste is volgens een bepaalde orde op gelabelde grafen. Neem vervolgens de knoop $v*$ uit deze (canonische) graaf als degene die het eerst voorkomt in de knooporde. Definieer nu $m(X)$ als de verzameling van* lower objects *$\langle X, v\rangle$, zodat v automorf is met de knoop die in X overeenkomt met $v*$ (terugmapping van de canonische graaf). We komen later nog terug op een meer algemene constructie van de functie m voor grafen.*

We hebben aangetoond dat er zo een functie p bestaat, maar wat is nu het nut hiervan? Wel, het lemma zegt ons dat we eender welk *ongeordend object* $X \in S$ mogen kiezen, vervolgens een willekeurig *lower object* uit zijn $\check{X} \in m(X)$ kiezen, en hiervoor een willekeurig *upper object* $\hat{X} \in f(\check{X})$ mogen selecteren. Als we hier vervolgens orbit S' van nemen, dan komen we volgens het lemma altijd in dezelfde S' uit, onafhankelijk van de drie gemaakte keuzes. In Figuur 3.22 zien we een schematisch overzicht van de functies m en p.

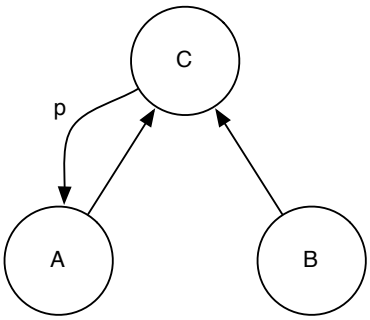
We noemen $S' = p(S)$ de parent van S en de verzameling

$$\{S, p(S), p(p(S)), \ldots\}$$

de *voorouders* of *ancestors* van S. Merk op dat S een voorouder van zichzelf is en dat er slechts een eindig aantal voorouders bestaan, p is dus een



Figuur 3.22: Constructie van m en p volgens McKay [24].



Figuur 3.23: De functie p.

nilpotente functie. Dit laatste kunnen we aantonen omdat we uit C7 weten dat $\forall \check{X} \in \check{\mathcal{L}} : f(\check{X}) = \hat{Y}$, waarbij $o(\hat{Y}) < o(\check{X})$, met andere woorden dat de orde alsmaar kleiner toepast als we f blijven toepassen.

We associëren met elk *reduceerbaar object* één ouder, en stellen op deze manier een verzameling bomen op: $\{(S, p(S)) \mid S \in \mathcal{U}_1\}$. Het zal blijken dat deze bomen altijd gericht zijn naar de *onreduceerbare objecten*. Hiermee wordt bedoeld dat als we op recursieve wijze de parent van een knoop in deze boom nemen, dat we dan altijd bij één van de *onreduceerbare objecten* uitkomen. De boom bestaat dus blijkbaar uit *ongeordende objecten*, voor elke orbit van *geordende objecten* hebben we precies één knoop in de boom. Het is mogelijk dat deze knopen meerdere kinderen hebben, maar de functie p laat toe om aan elke knoop slechts één ouder toe te kennen. Dit laatste maakt het mogelijk om een boom te construeren in plaats van een graaf. In Figuur 3.23 is te zien hoe de objecten A en B beide aanleiding kunnen geven (via *upper objects*en *lower objects*) tot eenzelfde C-object. Het C-object kan door middel van de functie p nu één "ouder" aanduiden. Hetgeen we willen bereiken is dat we voor elke knoop in de boom van *ongeordende objecten* precies één *lower object* gaan "printen". Het volgende algoritme vervult

deze taak:

Algoritme 8: `scan`

Input: X,n
Output: één representant voor elke $S \in \mathcal{U}[X]$, met $o(S) \leq n$

```
1  output X;
2  foreach orbit A verkregen door Aut(X) toe te passen op U(X) do
3      kies willekeurig een X̂ ∈ A;
4      if f'(X̂) ≠ ∅ then
5          kies willekeurig een Y̌ ∈ f'(X̂);
6          neem Y, zodat Y̌ ∈ L(Y);
7          if o(Y) ≤ n ∧ Y̌ ∈ m(Y) then
8              scan(Y, n)
9          end
10     end
11 end
```

We beschrijven de werking van het algoritme aan de hand van graafgeneratie. Het algoritme gaat van start met één *geordend object* en print dit onmiddellijk uit. Vervolgens worden de *upper objects* berekend, die onderverdeeld worden in automorfisme-orbits. Op deze manier verkrijgen we groepjes van *upper objects*, waarbij degenen met "dezelfde" geselecteerde knopen bij mekaar komen te zitten. Uit elk van deze groepjes wordt vervolgens één object gekozen. Omdat ze allemaal tot dezelfde graaf zouden uitbreiden is het niet nodig om er meerdere te kiezen. Van dit ene object nemen we een gerelateerd *lower object*, hetgeen ons de graaf na het groeien geeft, waarbij de nieuwe knoop gemarkeerd is. Vervolgens "vergeten" we deze markering en bekomen de geordende graaf Y. Hierna berekenen we $m(Y)$, hetgeen ons een automorfisme-orbit geeft van *lower objects*. De functie m geeft ons op deze manier één soort *lower object* dat verantwoordelijk is voor het genereren van Y. Merk op dat er meerdere automorfe objecten kunnen worden teruggegen, om ervoor te zorgen dat onze eerdere keuzes niets uitmaken. Als we inderdaad met zo een "toegelaten" object te maken hebben, dan slaagt de test en gaan we recursief verder met het nieuwe object. Merk opdat de uitvoer van het algoritme het over $\mathcal{U}[X]$ heeft, hiermee bedoelen we de abstracte grafen in de zoekruimte die bereikt kan worden door verder te werken met X. We kunnen namelijk enkel de hele verzameling $\mathcal{U}$ genereren indien we alle *onreduceerbare objecten* recursief uitwerken.

We moeten nu echter nog bewijzen dat elk object *precies* één keer gegenereerd wordt door het algoritme. We doen dit door te controleren of elk object minstens één keer, en hoogstens één keer geconstrueerd wordt.

Stelling 3.48. *Veronderstel dat $X_0 \in S_0 \in \mathcal{U}$, waarbij $o(X_0) \leq n$. Dan zal de uitvoer van het programma* `scan`*, met de parameters X_0 en n, exact één* lower object *bevatten voor elk* upper object *van ten hoogste orde n, dat afstamt van het* upper object *S_0.*

Bewijs. We delen eerst en vooral de afstammelingen van S_0 op in *generaties*: we zeggen dat een bepaalde afstammeling S van S_0 tot generatie i behoort als S $i+1$ voorouders heeft (gemeten vanaf S_0). Herinner dat een *upper object* een voorouder is van zichzelf, dus generatie $0 = \{S_0\}$.

We gaan nu door middel van inductie aantonen dat het algoritme wel degelijk doet wat ervan verwacht wordt, en doen dat door het bewijs op te splitsen in twee delen: we tonen achtereenvolgens aan dat voor elk *ongeordend object* één *geordend object* uitgeschreven wordt, en dat er maximum één object voor elk *ongeordend object* wordt uitgeschreven. Als inductiehypothese nemen we aan dat de stelling geldt voor alle objecten van orde ten hoogste n en met generatie ten hoogste i. Het is duidelijk dat de stelling klopt voor $i = 0$, dan wordt namelijk enkel het enige element van generatie 0 geprint, S_0.

DEEL 1: We tonen aan dat voor elke S eentje gemaakt wordt.

Veronderstel het bestaan van een S, die afstamt van S_0 en behoort tot generatie $i+1$. We noteren vanaf nu het behoren van S tot generatie i als $gen(S) = i$. Neem een $Y' \in S$, kies vervolgens een $\check{Y}' \in m(Y')$ en een $\hat{X}' \in f(\check{Y}')$. Merk op dat $\hat{X}'$ van een lagere orde is dan zowel Y' als $\check{Y}'$.

We weten dat er een T bestaat die afstamt van S_0, en een voorouder is van S, zodat

$$gen(T) < gen(S). \tag{3.40}$$

Wegens de inductiehypothese kunnen we besluiten dat T geprint wordt, en dat er dus een *lower object* $X \in T$ bestaat zodat $scan(X, n)$ wordt aangeroepen. Omdat S een afstammeling is van T, zal op een bepaald moment in de *for-loop* een $\hat{X} \in A$ geselecteerd worden zodat

$$\exists\, g \in G : \hat{X}^g = \hat{X}'. \tag{3.41}$$

Wegens het voorgaande weten we nu dat

$$\check{Y}' \in f'(\hat{X}') \Leftrightarrow \check{Y}' \in f'(\hat{X}^g), \tag{3.42}$$

en axioma C5 geeft vervolgens aan dat

$$\check{Y}' \in f'(\hat{X}^g) \wedge \check{Y} \in f'(\hat{X}) \Rightarrow \exists h \in G : \check{Y}' = \check{Y}^h. \tag{3.43}$$

We weten ook dat $\check{Y}$ deel uitmaakt van een bepaalde verzameling van *upper objects*

$$\begin{aligned} \check{Y} \in U(Y) &\Leftrightarrow \check{Y}^k \in U(Y)^k && (3.44)\\ &\Leftrightarrow \check{Y}^k \in U(Y^k) && (3.45)\\ &\Leftrightarrow \check{Y}^k \in U(Y'). && (3.46) \end{aligned}$$

We weten ook dat $\check{Y}' \in U(Y')$, en wegens eigenschap 3.40 geldt dat

$$Y^k = Y'. \tag{3.47}$$

We kunnen nu besluiten dat $Y', Y^k, Y \in S$. Onze aanname bestond eruit dat $\check{Y}' \in m(Y')$, dit wil zeggen dat

$$\check{Y}^k \in m(Y^k) \text{ (wegens 3.47)} \tag{3.48}$$
$$\Leftrightarrow \check{Y}^k \in m(Y)^k \text{ (wegens M3)} \tag{3.49}$$
$$\Leftrightarrow \check{Y} \in m(Y). \tag{3.50}$$

We hebben zojuist aangetoond dat er zeker een object gegenereerd wordt door een T, dat in $m(Y)$ zit. Hierdoor kunnen we besluiten dat $Y \in S$ geprint wordt, hetgeen wil zeggen dat er minstens één representant van S afgeprint wordt.

DEEL 2: We tonen nu aan dat voor elke S maximum één *lower object* geprint wordt.

Veronderstel dat er twee *lower object* voor S geprint worden, $Y_1, Y_2 \in S$, met $Y_1 \neq Y_2$. We weten, omdat beide *lower objects* in dezelfde orbit S zitten, dat

$$\exists g \in G : Y_2 = Y_1^g. \tag{3.51}$$

Opdat beide objecten uitgeschreven worden, moet de *if-test* uit het algoritme slagen, neem

$$\check{Y}_1 \in m(Y_1)$$
$$\check{Y}_2 \in m(Y_2),$$

als de *lower objects* die hiervoor zorgen. Wegens 3.51 en M3 weten we dat

$$\check{Y}_2 \in m(Y_2) = m(Y_1^g) = m(Y_1)^g, \tag{3.52}$$

ook geldt dat

$$\check{Y}_1^g \in m(Y_1)^g. \tag{3.53}$$

Omdat we weten dat $m(Y_1)$ een orbit is van automorfismen (zie M2) concluderen we dat

$$\exists g \in G : \check{Y}_2 = \check{Y}_1^h. \tag{3.54}$$

Neem nu

$$\hat{X}_1 \in f(\check{Y}_1)$$
$$\hat{X}_2 \in f(\check{Y}_2) = f(\check{Y}_1^h), \tag{3.55}$$

wegens 3.55 en C4 zien we dat

$$\exists k \in G : \hat{X}_2 = \hat{X}_1^k. \tag{3.56}$$

Neem nu X_1, X_2, zodat

$$\hat{X}_1 \in U(X_1) \tag{3.57}$$
$$\hat{X}_2 \in U(X_2). \tag{3.58}$$

We werken nu 3.57 uit:

$$\hat{X}_1 \in U(X_1) \Leftrightarrow \hat{X}_1^k \in U(X_1)^k \tag{3.59}$$
$$\Leftrightarrow \hat{X}_1^k \in U(X_1^k)(\text{wegens C2}) \tag{3.60}$$
$$\Leftrightarrow \hat{X}_1^k \in U(X_2)(\text{wegens 3.56}) \tag{3.61}$$

en samen met 3.58 kunnen we nu concluderen, wegens eigenschap 3.40, dat

$$X_2 = X_1^k. \tag{3.62}$$

We hebben X_1, X_2 die beide geprint moeten zijn, als Y_1 en Y_2 geprint zijn. Maar we weten dan $X_1^k = X_2$, dus $X_1 \cong X_2$, waarbij ze beide van generatie i zijn. Maar de inductiehypothese zegt ons dat geen graaf tweemaal wordt uitgeprint, tenzij de representanten exact hetzelfde zijn. We mogen dus concluderen dat

$$X_1 = X_2, \tag{3.63}$$

omdat ze op deze manier beide uitgeprint worden, aangezien de uitgeprinte graaf gelijk is aan elk van de twee objecten. We mogen nu concluderen dat de hoger vermelde $k \in Aut(X_1)$, hetgeen leidt tot het feit dat $\hat{X}_1$ en $\hat{X}_2$ in dezelfde orbit A worden geplaatst in de *for-loop*. De regel onder de for-loop geeft aan dat er precies n $\hat{X}$ uit A gekozen wordt. Opnieuw kunnen we redeneren zoals hiervoor, en concluderen dat $\hat{X}_1 = \hat{X}_2$

Voor beide *upper objects* wordt dus dezelfde Y gekozen (omdat ze hetzelfde object zijn) en geldt dus $Y_1 = Y_2$, hetgeen in strijd is met onze aanname ($Y_1 \neq Y_2$). □

Samengevat:

Deel 1: We weten dat de ouder van S (T) geprint moet zijn onder de vorm X, omdat die van generatie i is. Vervolgens weten we dat in de *for-loop* alle orbits uit $U(X)$ gegenereerd worden en dat hier noodzakelijkerwijs een *upper object* in moet zitten dat in relatie staat met een *lower object* van een $Y' \in S$. Vervolgens kijken we of de voorwaarden vervuld zijn om $scan(Y, n)$ op te roepen, en indien dit zo is, dan moet Y geprint worden. We tonen ook aan dat Y isomorf is met Y', waardoor we aangetoond hebben dat er minstens één *lower object* voor S wordt geprint.

Deel 2: Er van uitgaande dat we toch 2 *lower objects* uitprinten voor een bepaald *upper object*, werken we het algoritme van achter vaar voor door, en tonen aan dat dit leidt tot een contradictie.

Nu we bewezen hebben dat het algoritme werkt, hadden we graag nog wat geweten over de efficiëntie ervan. De volgende stelling geeft een bovengrens voor het aantal keer dat $\hat{Y} \in m(Y)$ berekend moet worden.

Stelling 3.49. *We beschouwen de berekeningen die het algoritme $scan(X_0, n)$ uitvoert. Neem N_1 als het aantal keer dat de test $\check{Y} \in m(Y)$ uitgevoerd wordt, en N_2 het aantal keer dat deze test slaagt. We mogen dan zeggen*

dan dat $N_1 \leq cN_2$*, waarbij* c *gelijk is aan het gemiddeld aantal orbits van* $Aut(X)$ *op* $L(X)$*.*

Bewijs. Neem $S \in \mathcal{U}_1$ en $o(S) \leq n$. Kies vervolgens $Y_1, Y_2 \in S$, een $\check{Y}_1 \in L(Y_1)$ en een $\check{Y}_2 \in L(Y_2)$. Neem ook nog een $\hat{X}_1 \in f(\check{Y}_1)$ en een $\hat{X}_2 \in f(\check{Y}_2)$.

Als $\exists g \in \mathcal{G} : \check{Y}_1^g = \check{Y}_2$, dan weten we dat $\exists h \in \mathcal{G} : \hat{X}_1^h = \hat{X}_2$ (C4). Zoals in het bewijs van stelling 3.48, kunnen we dan besluiten dat $\hat{X}_1 = \hat{X}_2$. We mogen dus concluderen dat er slechts één van de twee *lower objects* $\check{Y}_1, \check{Y}_2$ uit $f'(\hat{X}_1)$ gekozen wordt. Dit wil zeggen dat elk object dat het object S zal genereren hoogstens één test zal uitvoeren van de vorm "$\check{Y}_i \in m(Y)$" voor S. Elk object waaruit S gegenereerd kan worden voert met andere woorden maximaal één test uit.

In het algoritme kunnen we dit als volgt zien: Op regel 2 worden de orbits gevormd, en uit iedere orbit wordt op regel 3 een object gekozen. Het is duidelijk dat elke keuze van de orbit slechts één oproep van m kan veroorzaken, omdat er in regels 4-10 geen lus-opdracht voorkomt. De enige manier dat we dus twee tests voor eenzelfde object kunnen doen is als objecten uit twee orbits leiden tot dit object. Dit is echter niet mogelijk. Voor elk mogelijk gegenereerd object wordt de test op regel 7 dus maximaal één keer uitgevoerd.

Als we nu alle test willen tellen, nemen we het aantal objecten dat we willen genereren als N_2. Dit aantal is vanzelfsprekend ook het aantal tests dat zal slagen (als we de *onreduceerbare objecten* niet meerekenen). Voor elk van deze objecten worden maximaal "het aantal orbits van $Aut(X)$ op zijn *lower objects*" tests uitgevoerd. Dit laatste geldt omdat er wegens bovenstaande redenering slechts één automorf object gekozen wordt uit de orbit. Wanneer we dus het gemiddelde aantal automorfisme orbits van een object nemen en dit c noemen, worden er maximum $c \times N_2$ tests uitgevoerd. □

3.4.3 Constructie van m voor grafen

Definitie 3.50 (Partitie)**.** Een *partitie* van een verzameling knopen V, behorend tot een graaf $G = (V, E)$, is een reeks $\pi = (V_1, V_2, \ldots, V_k)$. Hierbij geldt:

1. $V_i \subseteq V, i = 1, 2, \ldots, k$
2. $V_i \neq \emptyset, i = 1, 2, \ldots, k$
3. $\bigcup_{i=1\ldots\ k} V_i = V$.
4. $V_i \cap V_j = \emptyset, i \neq j$.

Elke verzameling V_i in π is dus niet ledig en is een deelverzameling van $V = \{1, \ldots, n\}$. Verder zijn ze onderling disjunct. Een dergelijke V_i noemen we ook wel een *cel* van V.

We kunnen nu ook een permutatie definiëren op deze structuur. Een permutatie g, die op V inwerkt, kunnen we *celgewijs* op π toepassen: $\pi^g = (V_1^g, V_2^g, \ldots, V_k^g)$. Herinner dat π^g de notatie is om aan te geven dat de permutatie g toegepast wordt op π. Vervolgens definiëren we de *automorfisme groep* van (X, π) als

$$Aut_\pi(X) = \{g \in S_n \mid X = X^g \wedge \pi^g = \pi\}. \tag{3.65}$$

Merk op dat er een extra voorwaarde ligt op de permutatie g: ze moet namelijk ook zorgen dat π op zichzelf wordt afgebeeld. Dit laatste houdt in dat elke *cel* uit de partitie op zichzelf afgebeeldt dient te worden door de permutatie π, namelijk: $V_i^g = V_i, i = 1..k$. Hierbij dient g geïnterpreteerd te worden als permutatie die op elk element van de verzameling wordt toegepast.

Vervolgens beschouwen we een functie c, die partities transformeert, neem $\pi = (V_1, V_2, \ldots, V_k)$, dan is

$$c(\pi) = (\{1, 2, \ldots, |V_1|\}, \{|V_1| + 1, \ldots, |V_1| + |V_2|\}, \ldots, \{n - |V_k| + 1, \ldots, n\}). \tag{3.66}$$

Hierbij is $|V_i|$ het aantal knopen in cel V_i. De functie c zet dus een partitie om in een andere partitie, waarbij de knopen oplopend in de opeenvolgende cellen gestopt worden. Merk op dat c onafhankelijk werkt van de inhoud van de verschillende cellen, enkel de grootte wordt in rekening genomen.

Definitie 3.51. Een functie C is een *canonische labelling map* indien voor elke graaf $X = (V, E)$ en elke mogelijke partitie π van V geldt dat:

N1 $\exists g \in S_n : C(X, \pi) = X^g \wedge \pi^g = c(\pi)$, en

N2 $\forall\, h \in S_n : C(X^h, \pi^h) = C(X, \pi)$.

De eerste eigenschap zegt ons dat de functie een graaf, samen met een partitie van zijn knopen, afbeeldt op een permutatie van de invoergraaf. Deze permutatie moet zorgen dat, indien ze toegepast wordt op de partitie π, ze de knopen over de verschillende cellen verdeelt, zodat we komen tot $c(\pi)$.

Vaak is het zo dat we grafen willen genereren die voldoen aan een bepaalde eigenschap. Denk bijvoorbeeld aan de TFG's die we eerder wilden genereren. We hebben in dat geval de *upper objects* beperkt, zodat we enkel grafen van de juiste vorm kregen. Dit principe kunnen we veralgemenen naar een algemene eigenschap $\mathcal{P}$ die *knoop-overerfbaar* is. In deze sectie zullen we dieper ingaan op dit soort overerfbaarheid.

Definitie 3.52 (Knoop-overerfbaar)**.** Een graaf-eigenschap $\mathcal{P}$ wordt *knoop-overerfbaar* genoemd, indien voor elke graaf X, die deze eigenschap bezit, elke geïnduceerde subgraaf van X deze eigenschap ook bezit.

Volgens McKay kan deze definitie echter nog wat verfijnd worden:

Definitie 3.53 (Zwak knoop-overerfbaar)**.** Een graaf-eigenschap $\mathcal{P}$ wordt *zwak knoop-overerfbaar* genoemd, indien voor elke graaf X, die deze eigenschap bezit, er minstens één subgraaf van X deze eigenschap ook bezit.

We veronderstellen een eigenschap $\mathcal{P}$, die invariant is onder isomorfismen. Met dit laatste wordt bedoeld dat als $\mathcal{P}$ geldt voor een graaf G, dat dan alle grafen die isomorf zijn met G deze eigenschap ook bezitten. Verder nemen we aan dat voor een graaf X geldt dat $V(X) = \{1, \ldots, o(X)\}$, herinner dat $o(X)$ de orde van de graaf is. Zoals voorheen zullen we de *lower objects* $L(X)$ nemen als $\langle X, v\rangle, v \in V(X)$ en $L(K_1) = \emptyset$. De *upper objects* $U(X)$ nemen we als paren van de vorm $\langle X, W\rangle$, zodat $W \subseteq V(X)$ en de graaf X_W (gevormd door een knoop toe te voegen, en deze verbinden met alle knopen in W) de eigenschap $\mathcal{P}$ bezit.

De functie f, die de relatie tussen de *lower objects* en de *upper objects* weergeeft, definiëren we als

$$f(\langle Y, v\rangle) = \{\langle X, W\rangle^g | g \in \mathcal{G}\}, \tag{3.68}$$

waarbij W zo gekozen is, zodat $X_W = Y$ en de burenverzameling van v precies overeenkomt met W. Merk op dat uit de constructie duidelijk is dat de *lower objects* enkel in relatie staan met *upper objects* van een lagere orde. De reden waarom $\mathcal{P}$ zwak knoop-overerfbaar moet zijn, is omdat het anders mogelijk is dat er geen *lower objects* bestaan voor een object, met andere woorden $f(\check{Y}) = \emptyset$ voor een zekere $\check{Y}$. Dit is echter verboden volgens axioma C3.

We construeren vervolgens een generische versie van m voor grafen. Om hiertoe te komen, maken we gebruik van een hulp-functie λ, die werkt op gelabelde grafen die de eigenschap $\mathcal{P}$ bezitten, en de volgende eigenschappen heeft:

L1 $\lambda(X) \neq \emptyset \wedge \lambda(X) \subseteq V(X)$,

L2 $\forall g \in S_n : \lambda(X^g) = \lambda(X)^g$.

Uit deze twee voorwaarden kunnen we afleiden dat $\lambda(X)$ een niet-lege verzameling knopen teruggeeft, die bovendien een unie van orbits van $Aut(X)$ is. Als X bestaat uit slechts één knoop, dan geeft $m(X)$ gewoon de lege verzameling terug: $m(X) = \emptyset$. In het andere geval, neem $\pi = (\lambda(X), V(X) - \lambda(X))$, waarbij de tweede cel weggelaten wordt indien deze leeg is. Kies nu een $g \in S_n$, zodat geldt dat $X^g = C(X, \pi)$ en $\pi^g = c(\pi)$. Neem voor W de orbit van $Aut(X)$ die de knoop $1^{g^{-1}}$ bevat, waarbij 1 de eerste knoop in X^g is. Met andere woorden $W = \{v \mid v \in V(X) \wedge \exists g \in Aut(X) : v = v_1^g\}$, waarbij $v_1 = 1^{g^{-1}}$. We nemen nu

$$m(X) = \{\langle X, v\rangle \mid v \in W\}.$$

We hebben een partitie van onze knopen gemaakt, en vervolgens een permutatie g gekozen, die aan enkele voorwaarden voldoet. Vervolgens nemen we de knoop die door de permutatie op '1' wordt afgebeeld. Van deze knoop nemen we de orbit onder de automorfismen van X, om zo de gelijkwaardige knopen te bekomen. We tonen vervolgens aan dat we hiermee een functie hebben gecreëerd die voldoet aan alle nodige voorwaarden.

Lemma 3.54. *De voorgaande constructie van $m(X)$ is* well-defined *en voldoet aan eigenschappen M1, M2 en M3.*

Bewijs. **m(X) voldoet aan M1**: Als $L(X) = \emptyset$, dan hebben we een graaf met slechts één knoop (K_1). Onze constructie geeft aan dat we voor K_1 $\emptyset$ teruggeven.
m(X) voldoet aan M2: Als $L(X) \neq \emptyset$, dan moet het een orbit van een automorfisme-groep van X zijn, toegepast op $L(X)$. Dit is makkelijk in te zien, we kiezen namelijk één knoop uit X (nl. degene die op '1' wordt afgebeeld door g), en passen hier vervolgens alle automorfismen van X op toe om zo de verzameling W te verkrijgen. Vervolgens nemen we de *lower objects* die gevormd worden door knopen, die in W zitten, te verwijderen uit de graaf X. We weten dat deze elementen een orbit vormen onder $Aut(X)$ omdat:

$$\exists h \in \ Aut(X) : X^h = X, \tag{3.69}$$

$$\forall h \in \ Aut(X), \forall v \in \ W, \exists w \in \ W : v^h = w. \tag{3.70}$$

Uit het voorgaande kunnen we nu besluiten dat

$$\forall \ h \in \ Aut(X) : \langle X, v \rangle \ \in \ m(X) \Rightarrow \langle X^h, v^h \rangle \ \in \ m(X). \tag{3.71}$$

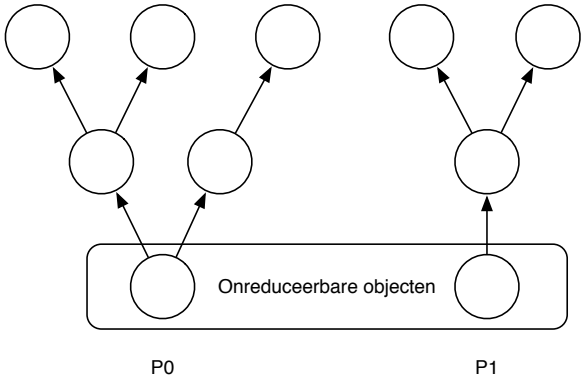
m(X) voldoet aan M3: Dit volgt uit N2 en L2.
m(X) is *well-defined*: Wanneer we X en een tweede permutatie van X X^h beschouwen, zodat $X^h = C(X, \pi)$ en $\pi^h = c(\pi)$, dan is $g^{-1} \circ h$ een automorfisme van X en bevinden 1^{-g} en 1^{-h} zich in dezelfde orbit van X. □

Het is bijgevolg mogelijk om aan de hand van een canonisatiefunctie een functie m te construeren. Neem voor de partitie van de knopen $\pi = \{V(X)\}$, dus alle knopen in één cel. Dan geldt

$$\lambda(X) = \{V(X)\}.$$

Elke canonische functie C kan bijgevolg gebruikt worden om m te construeren: Op invoer X:

1. X_c = canoniseer X met C.
2. Map de eerste knoop van X_c naar de overeenkomstige knoop in X.



Figuur 3.24: Parallellisatiemethode 1: elke boom wordt gegenereerd door een ander proces.

3. Maak *lower object* L door de betreffende knoop te markeren.
4. Pas $Aut(X)$ toe op L.
5. Geef de bekomen orbit terug.

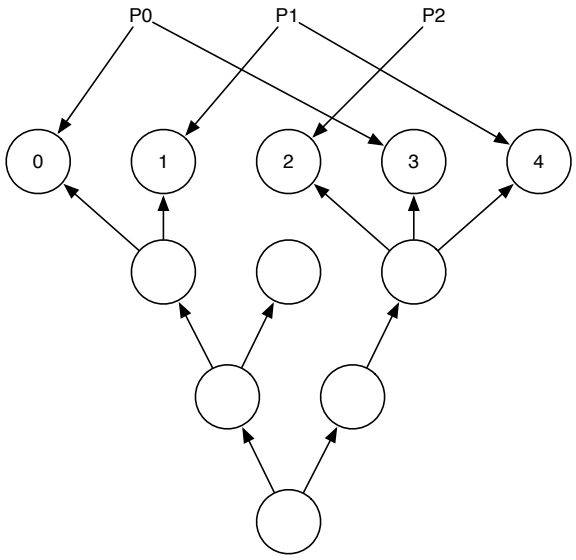
3.4.4 Optimalisatie

Omdat elk object in de zoekruimte teruggebracht kan worden tot eentje van de *onreduceerbare objecten*, kunnen we de zoekruimte beschouwen als een verzameling disjuncte bomen. We noemen zo een verzameling bomen een *bos*.

Als we alle objecten in de zoekruimte willen genereren, is het noodzakelijk om een element uit *onreduceerbare objecten* mee te geven als start-object. Merk op dat we het algoritme meerdere keren dienen uit te voeren: één keer per *onreduceerbaar object*. Het is dan ook gemakkelijk in te zien dat we verschillende instanties van het algoritme kunnen laten lopen, die elk starten van een ander *onreduceerbaar object*. Op deze manier kunnen we de verschillende bomen uit de zoekruimte in parallel genereren, zoals te zien in Figuur 3.24.

Maar wat als we maar over één *onreduceerbaar object* beschikken, zoals bij grafen? Dan is er slechts sprake van één boom in de zoekruimte. Er is gelukkig nog een alternatieve techniek die we kunnen gebruiken om het algoritme te verspreiden over verschillende processen, en zo een snelheidswinst kunnen behalen door middel van parallellisatie.

We doen dit door een bepaalde generatie in de zoekruimte af te bakenen. Elk algoritme genereert vanuit eenzelfde start-object een aantal andere objecten, maar wanneer we objecten verkrijgen uit een bepaalde generatie i, dan passen we een verdeeltechniek toe. We zorgen dat elk proces het aantal



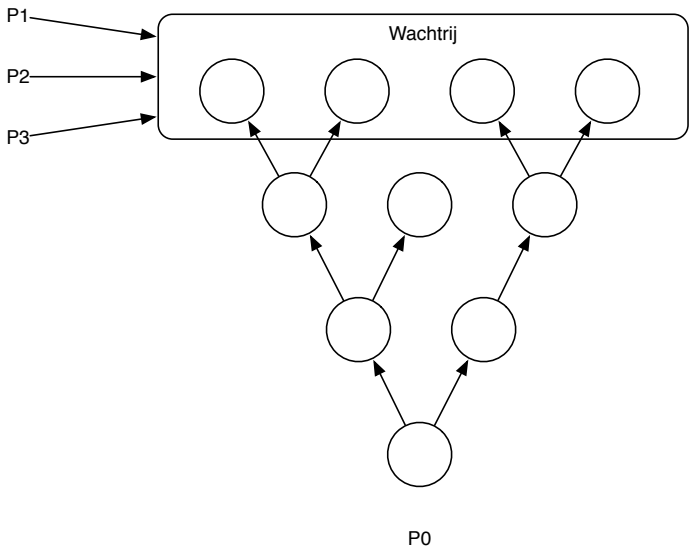
Figuur 3.25: Parallellisatiemethode 2: een grensgeneratie wordt verdeeld over de processen.

reeds gegenereerde objecten van generatie i bijhoudt. Een proces is enkel toegelaten om dit object enkel verder uit te breiden wanneer

$$counter \,\%\, n == p.$$

Hierbij is n het aantal processen en p het nummer van het actieve proces. Op deze manier kiest elk proces andere objecten uit generatie i om uit te breiden. Wanneer we alle processen beschouwen is het eenvoudig in te zien dat alle objecten uit generatie i uitgebreid worden en hierdoor dus de hele zoekboom beschouwd wordt. Een moeilijkheid bij deze techniek is dat we moeten zorgen dat de objecten uit generatie i relatief snel berekend kunnen worden, omdat deze, samen met de onderliggende generaties, door elk proces gegenereerd moeten worden. We willen met andere woorden zoveel mogelijke dubbele berekeningen vermijden door de grens-generatie i zo goed mogelijk te kiezen. In Figuur 3.25 is een schematisch overzicht van deze aanpak gegeven.

Een andere aanpak bestaat eruit om één proces alle grafen tot en met generatie i te laten genereren, en deze niet uit te diepen, maar in een globale wachtrij te plaatsen. Hiernaast kunnen we dan enkele processen laten lopen die uit deze wachtrij het volgende object nemen en dit helemaal uitdiepen tot op de gewenste generatie. Het voordeel is hierbij dat elk proces zo lang mogelijk bezig is. Bij de andere aanpak is het mogelijk dat één proces altijd de objecten krijgt waarbij niet veel geldige uitbreidingen zijn. Dit laatste proces zal vlugger klaar zijn dan de anderen. Wanneer deze methode



Figuur 3.26: Parallellisatiemethode 3: een globale wachtrij staat ter beschikking van de andere processen.

gebruikt wordt, zal dit proces gewoonweg een nieuw object krijgen om uit te breiden, totdat de wachtrij uitgeput is. Deze aanpak laat ons toe om de objecten tot en met generatie i eventueel op voorhand te berekenen, of in parallel met de andere processen. Het is in beide gevallen goed mogelijk om een hogere grens-generatie te nemen, omdat er geen sprake is van dubbele generatie. Indien het genererende proces in parallel werkt, moet er wel op toegezien worden dat deze de objecten voldoende snel kan genereren, zodat de anderen niet hoeven te wachten tot er iets in de wachtrij verschijnt.

Een voordeel is dat de processen niet hoeven te weten hoeveel anderen er meewerken aan het generatieproces, en dat er *at runtime* nieuwe processen kunnen worden toegevoegd die helpen met het genereren van de grafen. Een nadeel is dat er wat rekenkracht naar de netwerkstructuur dient te gaan (*client-server* communicatie). Als het rekenwerk nodig om één graaf uit te grensgeneratie volledig uit te diepen echter lang genoeg duurt, dan is deze communicatie verwaarloosbaar. In Figuur 3.26 wordt de werking van deze methode grafisch weergegeven.

3.4.5 Samenvatting

Tot slot van deze sectie geven we nog een samenvatting van het algoritme, specifiek toegepast op grafen.

Het algoritme begint met een bepaald object uit de zoekruimte uit te breiden via de groeioperatie. Deze groei-operatie bestaat eruit om op alle mogelijke manieren knopen te markeren. We verkrijgen op deze manier

een verzameling *upper objects*, die allen bestaan uit dezelfde graaf, gecombineerd met een andere deelverzameling van zijn knopen. De *upper objects* moeten we bekijken als dezelfde graaf, waaraan een nieuwe knoop met de geselecteerde knopen verbonden zal worden.

Vervolgens delen we deze *upper objects* op in groepjes. Elk groepje bevat *upper objects* die automorf zijn met mekaar. Dit komt neer dat *upper objects* die tot eenzelfde (ongerichte) graaf zullen leiden gegroepeerd worden.

Vervolgens kiezen we uit elk van deze groepjes precies één graaf. Deze graaf vormen we om tot een *lower object*, hetgeen inhoudt dat we de knoop effectief bijvoegen en markeren, en de overige markeringen verwijderen. Vervolgens verwijderen we deze nieuwe markering om te weten welke graaf we gaan genereren.

Opmerking 3.55. Het markeren van deze nieuwe knoop heeft helemaal geen zin, het wordt onmiddellijk erna toch weer verwijderd. In het algoritme wordt de functie f' toegepast om een verzameling *lower objects* te berekenen, waaruit we eentje kiezen. Het is perfect mogelijk om deze stap over te slaan en het *lower object* rechtstreeks te berekenen uit het *upper object*, omdat dit sowieso ook in f' zit en de keuze willekeurig is. Deze twee zaken zullen de snelheid van het algoritme ten goede komen omdat ze onnodige berekeningen vermijden.

Nu we een graaf uit een kleinere gebouwd hebben, kijken we of deze wel op de toegelaten manier geconstrueerd is. We berekenen hiertoe de functie m, die ons in principe een lijst automorfe knopen van de graaf teruggeeft. Deze knopen zijn degenen die bijgevoegd mogen worden. Met andere woorden, indien we onze kleinere graaf hebben uitgebreid door één van deze knopen toe te voegen, dan is onze uitbreiding geldig, anders niet. We dienen natuurlijk ook te controleren of onze graaf van de juiste orde is. Indien dit niet zo is, zijn we te ver in onze zoekboom afgedaald en staken we onmiddellijk de uitbreiding van deze graaf. Merk op dat het efficiënter zou zijn als we al vroeger hierop zouden testen.

Op deze manier wordt aan elk object in de zoekboom precies één subgraaf gekoppeld die deze graaf mag genereren. Het maakt niet uit welke geordende graaf we genereren, de functie m zal *lower objects* genereren die hiermee verwant zijn. De automorfisme orbit van knopen die m teruggeeft is nodig omdat we in een geordende graaf een knoop met dezelfde betekenis geselecteerd kunnen hebben en die ook willen toelaten, omdat het in essentie over dezelfde graaf gaat.

Bijlage B geeft een voorbeeld van de uitvoer van het algoritme een geeft de achterliggende gedachte weer.

3.5 Vergelijking

Het is duidelijk dat de methode beschreven in sectie 3.2 een meer algemeen framework schetst en dat gSpan hier een invulling voor geeft. We zullen een vergelijking trachten te maken tussen het algoritme dat door McKay voorgesteld wordt, en gSpan.

Ten eerste kunnen we bij McKay zien dat er isomorfietests vermeden worden omdat grafen in orbits geplaatst worden volgens automorfismegroepen. De objecten die op dezelfde manier groeien worden dus uitgeschakeld. Het is echter wel nog steeds mogelijk om via een ander object ook te groeien. Het nut van de functie m wordt hierdoor duidelijk: er mag slechts één subgraaf in staat zijn om een graaf te genereren. Op deze manier kunnen we voor elk object het aantal aanroepen van de complexe functie m begrenzen tot het aantal automorfisme-orbits van zijn *lower objects*. Dit aantal is hoogstens het aantal van zijn knopen.

Bij gSpan kunnen we een gelijkaardig onderzoek doen, daar weten we dat elke graaf maximaal $V \times E$ isomorfen heeft die een kind kunnen zijn van een minimale DFS-code. Voor elk van deze kinderen van minimale codes dient er dus een canonische graaf berekend te worden. Deze bovengrens ligt beduidend hoger dan die bij het algoritme van McKay. Let wel, dit is slechts een bovengrens, in praktijk kan dit aantal een stuk lager liggen. We kunnen hieruit met andere woorden nog geen conclusies trekken over de relatieve uitvoeringstijd van beide algoritmen.

In hetgeen volgt zullen we de ideeën van deze twee algoritmen trachten te combineren. We gaan proberen om in het framework van McKay de groeioperatie van gSpan te verwerken.

3.5.1 gSpan modelleren in het McKay-framework

Daar het algoritme geïntroduceerd door McKay een generisch algoritme is, kunnen we proberen om gSpan in dit framework te laten passen. We zorgen voor de gepaste graafvoorstelling, samen met een geschikte groeifunctie. We brengen de labels van de grafen niet in rekening.

We duiden de *lower objects* aan als een graaf X, samen met een boog e: $\langle X, e\rangle$. De *upper objects* stellen we voor als $\langle X, W\rangle$, waarbij W een verzameling van knopen is die we als volgt interpreteren:

- Als W uit slechts één knoop bestaat, dan wil dit zeggen dat we deze knoop kunnen uitbreiden met een boog naar een nieuwe knoop. T wordt gedefinieerd door de orde op de knopen van X.
- Als W precies twee knopen bevat, dan kunnen we de graaf uitbreiden met een boog tussen deze knopen.

We bekomen dus:

- $L(X) = \{\langle X, e\rangle \mid e \in E(X)\}$,
- $U(X) = \{\langle X, W\rangle \mid W \in V(X)\} \cup \{\langle X, W\rangle \mid W \in E(X)\}$

Merk op dat we enkel grafen toelaten waarbij de orde een boom T voorstelt. We kunnen nu de operatie f als volgt definiëren:

$$f(\langle X, e\rangle) \mapsto \{\langle X - e, W\rangle^g \mid g \in \mathcal{G}\},$$

waarbij W één knoop bevat als e een *forward edge* is, en twee knopen indien e een *back edge* is, namelijk de twee knopen die verbonden worden door e. We tonen aan dat f, samen met een permutatiegroep op de knopen $\mathcal{G}$ voldoet aan de axioma's. We nemen de inwerking van een permutatie $g \in \mathcal{G}$ op de *lower objects* en *upper objects* als de inwerking op beide componenten.

C1-C3 triviaal

C4

$$\begin{aligned} f(\langle X, e\rangle) &= \{\langle X - e, W\rangle^g \mid g \in \mathcal{G}\} && (3.72)\\ &= \{\langle X^g - e^g, W^g\rangle \mid g \in \mathcal{G}\} && (3.73)\end{aligned}$$

waarbij W zoals hiervoor.

$$\begin{aligned} f(\langle X, e\rangle^h) &= f(\langle X^h, e^h\rangle) && (3.74)\\ &= \{\langle X^h - e^h, W'\rangle^g \mid g \in \mathcal{G}\}. && (3.75)\end{aligned}$$

Het is duidelijk te zien dat $W' = W^h$, omdat de knopen in W afhangen van e. We bekomen dus

$$\begin{aligned} f(\langle X, e\rangle^h) &= \{\langle X^h - e^h, W^h\rangle^g \mid g \in \mathcal{G}\} && (3.76)\\ &= \{\langle (X^h)^g - (e^h)^g, (W^h)^g\rangle \mid g \in \mathcal{G}\} && (3.77)\\ &= \{\langle X^g - e^g, W^g\rangle \mid g \in \mathcal{G}\}, && (3.78)\end{aligned}$$

dit laatste mogen we schrijven omdat de compositie van elementen uit $\mathcal{G}$ terug een element uit $\mathcal{G}$ oplevert ($\mathcal{G}$ is een groep). Omdat we alle elementen van $\mathcal{G}$ toepassen, bekomen we opnieuw alle elementen uit $\mathcal{G}$.

C5 analoog aan C4

C6 triviaal

C7 Neem voor de orde het aantal bogen.

De volgende stap die we zullen nemen is de constructie van een *canonische labelling map* C. We gebruiken volgend algoritme om de functie m te berekenen:

Algoritme 9: m_{gSpan}

Input: X
1 bepaal $g \in \mathcal{G}$, zodat $X^g = X' = \min(X)$;
2 $v = 1^{-g}, 1 \in V(X')$;
3 **return** $\{\langle X, w\rangle \mid \exists a \in Aut(X) : w = v^a\}$;

Beschouw een partitie π die ons hetvolgende oplevert:

$$\pi = (V_1, V_2, \ldots, V_n)$$

We weten dat min een functie is die een canonische graaf berekent. Neem nu een geordende graaf X en zijn minimale DFS-code $X' = \min(X)$. We weten dat deze minimale graaf een permutatie is van X: $\exists g \in \mathcal{G} : X^g = X'$.

Wanneer we deze mapping g toepassen op π bekomen we

$$(V_1^g, V_2^g, \ldots, V_n^g).$$

We construeren vervolgens een functie h als volgt:

Algoritme 10: h

1 $counter = 0$;
2 $h = \emptyset$;
3 **for** $i = 1; i \leq n; i++$ **do**
4 | **foreach** $v \in V_i$, *van klein naar groot* **do**
5 | | $h = h \cup \{v \mapsto counter\}$;
6 | | $counter++$;
7 | **end**
8 **end**

Op deze manier verkrijgen we door het toepassen van h na g een partitie van de vorm $c(\pi)$. We noemen deze gecomponeerde functie $l = h \circ g$.

Stelling 3.56. *Het zojuist gegeven algoritme is een* canonische labelling map.

Bewijs. We definiëren $C(X, \pi) = X^l$. N1 geldt omdat $l \in \mathcal{G}$ en we weten dat

$$\pi^l = \pi^{h \circ g} = (\pi^g)^h = c(\pi),$$

wegens de voorgaande constructie.

We tonen vervolgens aan dat N2 geldt. We weten dat er een permutatie $p, q \in \mathcal{G}$ bestaat zodat $X^p = \min(X)$ en $(X^h)^q = \min(X^h)$. We weten ook dat $\min(X) = \min(X^h)$, en dus geldt dat

$$X^p = \min(X) = \min(X^h) = (X^h)^q.$$

Hieruit volgt dat

$$\pi^p = (\pi^h)^q.$$

We weten dat er een permutatie k bestaat die ervoor zorgt dat $(\pi^p)^k = c(\pi)$. Deze k heeft een analoog effect op $(\pi^h)^q$, waaruit meteen volgt dat er een $l = k \circ q$ bestaat zodat $(\pi^h)^l = c(\pi)$. Er geldt nu $C(X^h, \pi^h) = X^{l \circ h} = X^{k \circ p} = C(X, \pi)$.

□

We zijn gekomen tot een canonische labelling map die ons toelaat om de functie m in te vullen. Op deze manier hebben we aan alle voorwaarden voldaan om het algoritme aan te passen aan het framework van McKay.

We hebben echter geen gebruik gemaakt van het *meest-rechtse pad* en de *meest-rechtse knoop*, om op deze manier minder kinderen te genereren. De toepassing hiervan is minder triviaal. De eigenschap $\mathcal{P}$ die hier van toepassing is, is de volgende: "X komt overeen met een geldige DFS-code.". We kunnen vervolgens proberen om het aantal uitbreidingen van een graaf te beperken. Merk op dat we het hier hebben over geordende grafen, en het concept van "overeenkomst met een DFS-code" triviaal is. McKay [24, p. 9] vermeld in zijn paper dat het voldoende is dat er een subgraaf van X is die ook aan $\mathcal{P}$ voldoet (zie definitie 3.53). We weten dat voor elke ongeordende graaf minstens één geldige DFS-code dient te bestaan, aan deze regel is dus voldaan.

Op het eerste zicht lijkt het niet mogelijk, omdat McKay ongeordende en geordende objecten beschouwd. Bij orderly algoritmen worden in essentie enkel de geordende in rekening gebracht.

Het dieper uitwerken van dit algoritme zal gebeuren in deel II van deze thesis. We zullen proberen om het concept van de DFS-code te verwerken in Mckay's framework om het algoritme meer performant te maken. Of dit mogelijk is hangt af van het feit of we een functie f kunnen construeren die de uitbreiding enkel toelaat via degene die door gSpan toegelaten zijn en nog steeds voldoet aan de voorwaarden. Verder zullen we proberen om een algemeen orderly algoritme te modelleren in dit framework.

Hoofdstuk 4

Algoritmen en implementatie

Wat de implementatie betreft is de aandacht gegaan naar het uitwerken van een framework voor McKay- en Orderly-gebaseerde algoritmes. In dit hoofdstuk geven we een beschrijving over hoe het framework opgebouwd is en argumenteren we waarom bepaalde ontwerpkeuzes gemaakt zijn. We geven ook verschillende alternatieven voor snellere implementaties en beschrijven hoe de componenten van het framework vervangen kunnen worden om snellere werking te bekomen.

We beschrijven eerst een algemeen framework voor combinatorische objecten en permutaties die erop inwerken. Daarna behandelen we enkele standaardimplementaties voor deze abstracte klassen. Vervolgens beschrijven we achtereenvolgens hoe het algemene McKay-algoritme en orderly-algoritme werken en hoe ze gebruikt kunnen worden voor andere soorten of klassen van objecten. Hierna beschrijven we hoe we de rapporteringssystemen geïmplementeerd hebben.

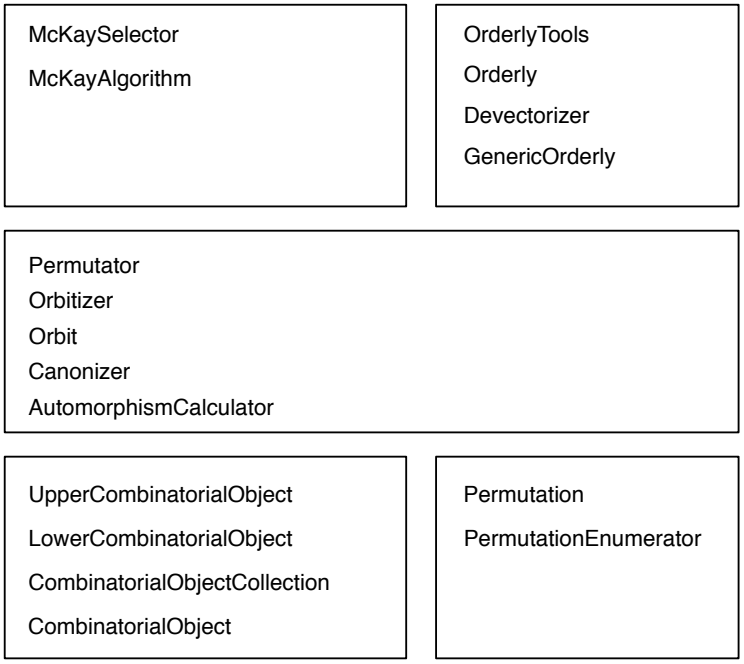
Bij deze beschrijvingen houden we in het achterhoofd dat het framework in C++ geschreven is. Dit laatste geeft soms de aanzet om dieper in te gaan op de allocatie van objecten en het werken met pointers. Waar nodig lichten we dit toe.

Het resulterende programma en de mogelijkheden ervan komen in het volgende hoofdstuk aan bod.

4.1 Overzicht

In Figuur 4.1 is een overzicht van de opbouw van het framework gegeven De twee onderste lagen omvatten het werkelijke framework, de bovenste laag bestaat uit een implementatie van de algoritmes.

We hebben gekozen om te werken met *interfaces* en *abstracte klassen* om zo algemeen en zo modulair mogelijk te kunnen werken. Dit heeft als gevolg dat we verschillende implementaties relatief eenvoudig kunnen uittesten. In deze sectie bespreken we de gebruikte klassen in het algemeen. In de vol-



Figuur 4.1: Een overzicht van het abstract framework.

gende sectie bespreken we enkele standaardimplementaties van de specifieke klasse.

De onderste laag bestaat uit de basisobjecten die gebruikt zijn. Ons doel was om zo algemeen mogelijk te werken, en hiertoe is besloten om geen beperking te zetten op het soort objecten. Het framework is bijgevolg in staat om ingevuld te worden door gewone combinatorische objecten. De klassen waar het om gaat zijn:

- `CombinatorialObject`
  Een klasse die een combinatorisch object voorstelt, deze bevat methoden die toelaten het object te klonen, te vergelijken, ... Een invulling van deze klasse is bijvoorbeeld de klasse `Graph`, die een graaf voorstelt en alle nodige methodes implementeert.

- `LowerCombinatorialObject` en `UpperCombinatorialObject`
  Met het algoritme van McKay reeds in het achterhoofd hebben we basisklassen gemaakt die de *lower objects* en *upper objects* voorstellen.

- `CombinatorialObjectCollection`
  Deze klasse stelt een verzameling combinatorische objecten voor.

Verder hebben we natuurlijk ook nood aan permutaties:

- `Permutation`
  Een object dat een permutatie voorstelt, maar geen implementatie biedt. Een standaardimplementatie wordt gegeven door `StandardPermutation`, die een array van getallen gebruikt om de permutatie intern voor te stellen.

- `PermutationEnumerator`
  Veel naïve algoritmes zoeken alle mogelijke permutaties af om iets te bekomen (automorfismes, canonische vorm,...). Om deze te kunnen testen hadden we dan ook nood aan een interface die toelaat om permutaties te enumereren.

Nu we permutaties en objecten hebben, kunnen we naar isomorfismen gaan kijken:

- `Permutator`
  We moeten natuurlijk in staat zijn om een permutatie toe te passen op een object. We hebben gekozen om dit los te koppelen van zowel het object zelf als de permutatie. Indien het object verantwoordelijk was voor het toepassen van de permutatie, dan konden we speciale eigenschappen van een permutatie-object niet uitbuiten, omgekeerd kunnen we dezelfde redenering volgen. Op deze manier is het mogelijk om zowel algemeen te werken als bijvoorbeeld een implementatie te maken die specifieke eigenschappen van speciale permutaties of combinatorische objecten uitbuit!

- `AutomorphismCalculator`
  Deze klasse staat in voor het genereren van automorfismen van een bepaald combinatorisch object. Het is ook een `PermutationEnumerator`, omdat het object in staat is om verschillende permutaties op te sommen (die in dit geval automorfismen zijn).

- `Orbit`
  Deze klasse stelt een orbit voor.

- `Orbitizer`
  Deze klasse is in staat om alle orbits van een object te berekenen van *lower objects* en *upper objects*.

- `Canonizer`
  Een klasse die de canonische vorm van een object berekent. Dit is zowel voor het orderly-algoritme als voor het algoritme van McKay van belang.

- `CombinatorialObjectWriter`
  Dit is een klasse die in staat is objecten te verwerken op een willekeurige manier. Ze zal in het algemeen gebruikt worden om objecten na

generatie werkelijk te gebruiken. In ons geval was het meestal gewoon het schrijven van het object naar een bestand of de standaard uitvoer, of het bijhouden van het aantal objecten dat reeds weggeschreven is. In het algemeen kan dit gebruikt worden voor alle taken die een gebruiker wil uitvoeren op of met de gegenereerde grafen.

Op de bovenste laag bevinden zich de algoritmen en hun specifieke "tools":

- `McKaySelector`
  Dit is de functie "m" uit het algoritme, die een verzameling van *lower objects* genereert.
- `Mckay`
  Een implementatie van het algoritme van McKay.
- `OrderlyTools`
  Een verzameling van een groei-operatie, een canonisatiefunctie en een vergelijkingsoperator, die dienen om het orderly algoritme correct te laten werken op gekozen objecten.
- `Orderly`
  Een implementatie van het orderly algoritme.
- `Devectorizer`
  Zet een vector om naar een object, hier komen we nog op terug.
- `GenericOrderly`
  Een implementatie van het generische orderly algoritme, dat werkt op basis van vectoren.

4.2 Standaard Objecten

Er zijn enkele klassen die we nodig gaan hebben in de rest van het programma, maar die niet verschillen voor het type object. Dit zijn

- `StandardCombinatorialObjectWriter`
  Deze klasse schrijft een combinatorisch object uit naar de standaard uitvoer.
- `StandardCombinatorialObjectCollection`
  Een verzameling combinatorische objecten, met de gebruikelijke operaties erop gedefinieerd (toevoegen, verwijderen, contains, ...).
- `StandardOrbit`
  Dit is een andere naam voor de `StandardCombinatorialObjectCollection`.

- `StandardOrbitCollection`
  Een verzameling orbits.

- `StandardPermutation`
  De voorstelling van een permutatie.

Verder kunnen we voor enkele van de abstracte klassen een algemene implementatie geven, die onafhankelijk is van het type combinatorisch object. In de rest van deze sectie bespreken we enkele van deze standaardobjecten, samen met de onderzochte verbeteringen.

4.2.1 De StandardLowerOrbitizer

Dit object berekent alle lower objects en stopt ze in een orbit. In dit geval gebeurt het aan de hand van een `AbstractLowerCalculator`, een `AbstractAutomorphismCalculator` en een `AbstractPermutator`.

De orbitizer laat de `AbstractLowerCalculator` alle *lower objects* berekenen en gaat ze vervolgens allemaal af. De oorspronkelijke was degene weergegeven in Algoritme 11: we nemen gaan alle *lower objects* af en kijken voor elk automorfisme op de toepassing hiervan ergens in een bucket zit. Indien dit zo is, kunnen we het *lower object* aan de betreffende bucket toevoegen. Indien het object in geen enkele bucket thuishoort dan maken we een nieuwe bucket. De complexiteit van dit algoritme is $O(n^5)$. Dit omdat we $O(n)$ *lower objects* hebben, in het ergste geval nog eens $O(n)$ buckets, en $O(n)$ automorfismen. Het toepassen van een automorfisme beschouwen we als $O(n^2)$ bij grafen, een lid-test kan in $O(n)$[1]. Merk wel op dat het aantal automofismen en het aantal buckets omgekeerd evenredig zijn. Indien we slechts één automorfisme hebben, dan zullen alle objecten in een verschillende bucket terechtkomen. Als we alle mogelijke permutaties als automorfisme hebben, dan vallen ze in één bucket. De gegeven complexiteit kan dus nog verfijnd worden.

Een andere aanpak is om telkens we een nieuwe bucket moeten maken, alle automorfismes toegepast worden, zodat de bucket dadelijk vol zit. Het algoritme wordt gegeven in Algoritme 12. Wanneer we nu objecten aflopen, dienen we enkel te kijken of ze ergens in een bucket zitten en niet meer alle automorfismen af te lopen, hetgeen ons veel tijd bespaart. De complexiteit is $O(n^4)$. Dit omdat we $O(n)$ *lower objects* hebben, $O(n)$ buckets en een lid-test $O(n)$ vraagt. Indien we een nieuwe bucket maken vraagt dit $O(n)$ om automorfismes om af te lopen en $O(n^2)$ om het automorfisme toe te passen. De ongelijkheidstest vraagt $O(n)$.

Bij *lower objects* van grafen zoals ze in deze tekst beschouwd zijn, is het toepassen van een automorfisme mogelijk in constante tijd: er moet

[1] indien het efficient geïmplementeerd is kunnen we met binair zoeken $O(\log(n))$ bekomen

slechts één knoop afgebeeld worden. De verkregen complexiteiten van de verschillende algoritmen zijn respectievelijk $O(n^4)$ en $O(n^3)$.

Het tweede algoritme berekent voor elke bucket maximaal één keer de toepassing van alle automorfismen, terwijl het eerste dat mogelijke meerdere keren doet voor elke bucket.

In de implementatie hebben we gekozen voor het tweede algoritme.

Opmerking 4.1. Naast deze standaardimplementatie hebben we er ook eentje ontworpen die specifiek is voor grafen. Bij grafen is het eenvoudiger om orbits te bouwen: we markeren een knoop en houden bij dat deze gemarkeerd is (in bijvoorbeeld een lijst). Ook de knopen die nog niet gemarkeerd zijn worden bijgehouden. Vervolgens maken we alle *lower objects* aan waarbij een automorfe knoop gemarkeerd is. Al deze knopen worden aangeduid als gebruikt. Al deze "automorfe" *lower objects* horen vanzelfsprekend in dezelfde orbit thuis. Vervolgens nemen we een volgende knoop die nog nooit gemarkeerd is en herhalen de procedure. Op deze manier maken we telkens een nieuwe bucket aan, en moeten de *lower objects* niet op voorhand berekend zijn: alles gebeurt in één keer!

Algoritme 11: Naive LowerOrbitizer

```
   Input: verzameling lower objectsLO, verzamelingen automorfismen Auts
   Output: orbits van lower objects
 1 BucketCollection BC = empty;
 2 foreach lo ∈ LO do
 3 |   found = false;
 4 |   foreach b ∈ BC do
 5 |   |   foreach a ∈ Auts do
 6 |   |   |   if b bevat a(lo) then
 7 |   |   |   |   voeg lo toe aan b;
 8 |   |   |   |   found = true; break;
 9 |   |   |   end
10 |   |   end
11 |   |   if found then
12 |   |   |   break;
13 |   |   end
14 |   end
15 |   if not found then
16 |   |   voeg nieuwe bucket toe aan BC;
17 |   |   stop lo in deze bucket;
18 |   end
19 end
20 return BC;
```

Algoritme 12: Betere LowerOrbitizer

Input: verzameling *lower objectsLO*, verzamelingen automorfismen *Auts*
Output: orbits van *lower objects*

```
1  BucketCollection BC = empty;
2  foreach lo ∈ LO do
3  |  found = false;
4  |  foreach b ∈ BC do
5  |  |  if b bevat lo then
6  |  |  |  found = true; break;
7  |  |  end
8  |  end
9  |  if not found then
10 |  |  voeg nieuwe bucket b toe aan BC;
11 |  |  stop lo in b;
12 |  |  foreach a ∈ Auts do
13 |  |  |  if a(lo) ≠ a then
14 |  |  |  |  stop a(lo) in b;
15 |  |  |  end
16 |  |  end
17 |  end
18 end
19 return BC;
```

4.2.2 De StandardUpperOrbitizer

Dit object werkt ongeveer hetzelfde als de `StandardLowerOrbitizer`, maar dan natuurlijk met *upper objects*. De algoritmen die we hier gebruiken zijn dezelfde, maar de complexiteit is anders in het geval van de *upper objects* die we bij grafen gebruiken: het toepassen van een automorfisme op een *upper object* van een graaf houdt nu in dat er meerdere knopen gepermuteerd moeten worden. We hebben hiervoort $O(n)$ nodig.

4.2.3 De StandardPermutationEnumerator

Het is in ons programma vaak nodig om alle mogelijke permutaties op te sommen. Denk bijvoorbeeld aan het bepalen van automorfismen of het berekenen van een canonische vorm. De eenvoudigste aanpak is vaak het opsommen van alle permutaties en kijken welke geldig zijn.

Hiervoor hebben we een klasse gemaakt die ons alle mogelijke permutaties teruggeeft, één na één. Een simpel recursief algoritme wordt gegeven in Algoritme 13. Hier dienen twee globale arrays te zijn van grootte n (aantal elementen dat we willen permuteren). Telkens als `printCurrent` aangeroepen wordt, dan is `Perm` gevuld met een permutatie: element 0 wordt afgebeeld op perm[0] enz... De array `Done` wordt gebruikt om bij te houden welke elementen al gebruikt zijn.

Merk op dat we niet graag een recursief algoritme hebben, omdat we soms geïnteresseerd zijn in het één na één genereren van permutaties. Dit laatste is een belangrijk iets, omdat vanaf een bepaald moment de permutaties gewoon niet meer in het geheugen zullen gaan. In Tabel 4.1 zien we hoe snel het aantal permutaties groeit. Om geheugentekort te vermijden willen we de mogelijkheid om de permutaties achtereenvolgens te genereren.

Hierom werd de zoektocht naar een iteratieve procedure gestart en na een tijdje brainstormen en uitproberen hebben we het stap-algoritme ontworpen dat te zien is in Algoritme 14.

Voorbeeld 4.2. *Wat het algoritme in essentie doet is hetvolgende: Begin achteraan in de vector die een permutatie voorstelt, en probeer een groot getal één plaats naar voren te schuiven. Dit mag enkel indien het getal kleiner is. Indien we starten met een vector*

1234

die een identieke permutatie voorstelt, dan schrappen we het laatste cijfer, en blijven dit doen tot we een cijfer schrappen dat kleiner is dan eentje dat reeds geschrapt is. Bij 3 doet dit geval zich al voor. Dan vullen we op die positie het eerstvolgende geschrapte cijfer in dat hoger is. In dit geval is enkel 4 beschikbaar. Overige geschrapte cijfers komen in volgorde achteraan te staan. We bekomen:

1243

Nu schrappen we 3 en 4, wanneer 2 geschrapt wordt, zien we dat er hogere getallen beschikbaar zijn. We vullen het eerstvolgende in (3) en vervolledigen de reeks met de geschrapte getallen in oplopende volgorde:

1324

Indien we deze relatief eenvoudige procedure blijven toepassen bekomen we

achtereenvolgens

1342
1423
1432
2134
2143
2314
2341
2413
2431
3124
3142
3214
3241
3412
3421
4123
4132
4213
4231
4312
4321.

Het is duidelijk dat al de $24 = 4!$ *permutaties gegenereerd zijn!*

Verder hebben we aan onze `StandardPermutationEnumerator` nog enkele optimalisaties toegevoegd:

Caching Onze klasse kan ingesteld worden op caching, indien dit het geval is, worden de gegenereerde permutaties allemaal bijgehouden en indien we ze erna opnieuw nodig hebben, kunnen we *rewinden* en moeten ze enkel uit het geheugen opgehaald worden. Stel dat we automorfismen en dadelijk erna een canonisatie berekenen, waarbij beide procedures alle permutaties moeten afgaan, dan kunnen we ze gewoon laten cachen. We hebben ook een *AutoCaching* ingebouwd, waarbij een bovengrens gespecifieerd kan worden tot welke n er gecached moet worden. Dit laatste heeft als nut dat we eenvoudig een grens kunnen zetten op de caching, zodat er automatisch gezorgd wordt dat het geheugen niet volloopt.

n	# permutaties = $n!$
1	1
2	2
3	6
4	24
5	120
6	720
7	5 040
8	40 320
9	362 880
10	3 628 800

Tabel 4.1: Aantal permutaties op n knopen

Initialisatie Vaak is het zo dat verschillende objecten die na mekaar verwerkt worden dezelfde grootte hebben. De mogelijke permutaties zijn in dit geval dezelfde. Wanneer we onze enumerator dan initialiseren aan de hand van zulk object, dan is het niet nodig dat al het geheugen opnieuw wordt vrijgegeven en aangemaakt. In plaats daarvan zullen we gewoon *rewinden*. De *rewind*-procedure bewaart de caching, dus naast het nutteloos vrijgeven van intern geheugen, kunnen we de inhoud van de cache hergebruiken, hetgeen eveneens leidt tot performantiewinst!

Minder kopiëren Wanneer we permutaties laten berekenen, dan moet deze teruggegeven worden. Hierbij komt veel kopieer- en allocatiewerk aan te pas, zelfs als we met pointers werken. Indien we een pointer vanuit ons intern object willen teruggeven, dan willen we dat deze van buitenaf beschikbaar is. Intern werken we enkel met ofwel het caching systeem, ofwel een array. Er moet dus een object "gekloond" worden, en hiervoor moet weer geheugen gereserveerd worden. Om dit laatste te vermijden hebben we een procedure geschreven waaraan een pointer naar een permutatie kan worden meegegeven. We werken hier met een `StandardPermutation`, die intelligent het gegeheugen beheert (hierover later meer). Onze `getNext`-functie kent de nieuwe permutatie toe aan het geheugen dat we meekrijgen, op deze manier is er geen nood aan nieuwe allocatie, enkel kopieerwerk.

4.2.4 De StandardAutomorphismCalculator

Het bereken van automorfismen voor combinatorische objecten kunnen we doen zoals aangegeven in Algoritme 15. We gebruiken hier een `PermutationEnumerator`, die ons achtereenvolgens de verschillende permutaties geeft. Voor elke permutatie wordt getest of het een automorfisme is voor

Algoritme 13: Recursief permutaties genereren

```
   Input: pos
   // als we de hele vector gevuld hebben, dan mogen we hem
      printen
   // maw de positie moet eentje te ver staan
 1 if pos == n then
 2 |  printCurrent();
 3 end
 4 else
   |  // we proberen alle getallen van klein naar groot
 5 |  for int i = 0; i < n; i++ do
 6 |  |  if not Done[i] then
   |  |  |  // we voegen getal i achteraan toe
 7 |  |  |  Perm[pos] = i;
   |  |  |  // volgende recusrieve stappen mogen dit getal niet
   |  |  |     meer gebruiken
 8 |  |  |  Done[i] = true;
 9 |  |  |  generaterec(pos+1);
   |  |  |  // vorige recursieve stappen mogen terug gebruik
   |  |  |     maken van dit getal
10 |  |  |  Done[i] = false;
11 |  |  end
12 |  end
13 end
```

Algoritme 14: Iteratieve stap op permutaties te genereren

```
1  index = n-1;
2  stop = false;
3  while not stop AND index >= 0 do
4      if perm[index] != n-1 then
5          done[perm[index]] = false;
           // kijk of er een groter getal vrij is
6          old = perm[index];
7          isThereBigger = false;
8          for i = old+1; i < n AND !isThereBigger; i++ do
9              if not done[i] then
10                 isThereBigger = true;
11             end
12         end
           // is er een groter getal vrij?
13         if isThereBigger then
               // eerstvolgende getal kiezen dat groter is
14             for j = old+1; j < n; j++ do
15                 if not done[j] then
16                     perm[index] = j;
17                     done[j] = true;
18                     break;
19                 end
20             end
21             pos = index+1;
               // ga alle mogelijke getallen OPLOPEND af
22             for i = 0; i < n AND pos < n; i++ do
                   // als ze niet ingevuld zijn
23                 if not done[i] then
                       // gebruiken we ze
24                     perm[pos] = i;
25                     done[i] = true;
                       // de cursor naar de volgende positie
                          verplaatsen
26                     pos++;
27                 end
28             end
               // we hebben een nieuwe permutatie
29             stop = true;
30         end
31         else
               // anders gaan we 1 positie meer naar voor kijken
32             index–;
33         end
34     end
35     else
36         done[perm[index]] = false;
           // anders gaan we 1 positie meer naar voor kijken
37         index–;
38     end
39 end
```

het object G. Indien dit zo is wordt de permutatie teruggegeven. Bij de volgende aanroep dient de enumerator verder te gaan waar hij gestopt is.

Merk op dat we hier gelijkaardige optimalisaties kunnen doorvoeren als bij de `StandardPermutationEnumerator`. Deze klasse is dan ook voorzien van caching-mogelijkheden en een allocatie-vermijdende `getNext`-functie. Verder is er ook de mogelijkheid om alle automorfismen in één keer te berekenen (precompute).

Algoritme 15: Iteratieve stap op automorfismen te genereren

```
   Input: AbstractPermutationEnumerator pe, CombinatorialObject G
   Output: automorfisme p
1  Permutation p;
2  while pe.getNext(p) do
3  |  if p(G) = G then
4  |  |  return p;
5  |  end
6  end
7  return NULL;
```

4.2.5 De StandardAutomorphismCalculatorBoosted

Deze klasse is identiek aan de vorige, met als uitzondering dat er iets intelligenter wordt omgesprongen met geheugengebruik: in plaats van altijd een kopie te verkrijgen van een gepermuteerd object, permuteren we nu objecten zelf, zodat er geen nood is aan continue geheugenallocatie en vrijgave.

4.3 Intelligente allocatie

Zoals reeds in de vorige sectie besproken kunnen we objecten intelligenter laten omspringen met hun allocatie. We bespreken hier de techniek die werd toegepast om onnodige geheugenallocatie te vermijden.

In het algemeen heeft een object een copy constructor en een assignment operator. Wanneer de copy constructor aangeroepen wordt, wordt er een nieuw object gemaakt en dient er dus noodzakelijkerwijs nieuw geheugen gebruikt te worden. Maar als de assignment operator aangeroepen wordt, dan kunnen we vaak deze allocatie vermijden. Denk bijvoorbeeld aan het object voor een graaf met n knopen, waarbij de bogen in een $n \times n$ matrix worden opgeslagen. Wanneer we een graaf met evenveel knopen hieraan willen toekennen, is het onnodig om de matrix vrij te geven en opnieuw te alloceren!

Deze techniek werd onder andere toegepast op grafen, vectoren en permutaties en is belangrijk om het programma sneller te laten lopen.

4.4 McKay

Wat betreft het algoritme van McKay hebben we volgende zaken nodig:

- `AbstractUpperOrbitizer`
  Dit object hebben we hiervoor reeds besproken en geeft alle *upper objects* terug, mooi in orbits verdeeld volgens de automorfismen van het combinatorisch object.

- `AbstractMcKaySelector`
  Dit is de functie m uit het algoritme. Ze zet een combinatorisch object om in één orbit van *lower objects*.

- `AbstractCombinatorialObjectWriter`
  Dit object bevat de opdrachten die uitgevoerd moeten worden met betrekking tot een gegenereerd object. Telkens een object in het algoritme "uitgeprint" mag worden, wordt de write-functie van een object van dit type aangeroepen. Een voor de hand liggende implementatie schrijft het object naar de standaard uitvoer (`StandardCombinatorialObjectWriter`) of houdt de tel bij met het aantal gegenereerde objecten (`StandardCombinatorialObjectCounter`).

Het is duidelijk dat deze functies voldoende zijn om Algoritme 8 te implementeren. In hetgeen volgt lichten we het gebruik van de `AbstractMcKaySelector` nog toe.

Voor grafen hebben we een concrete implementatie gemaakt onder da naam `NaiveMcKaySelector`. Deze klasse heeft een `AbstractAutomorphismCalculator` en een `AbstractCanonizer` nodig om $m(G)$ te berekenen. We hebben de theorie gebruikt in sectie 3.4.3 beschreven werd om door middel van een canonisatiemethode een functie m te kunnen construeren.

De canonizer berekent eerst de canonische vorm van het object en levert ons een permutatie van de knopen op, om deze canonische vorm te bekomen. Vervolgens kijken we welke knoop op 0 werd afgebeeld en gebruiken we de automorfismen op een orbit van het op deze manier geconstrueerde *lower object* te berekenen. Het is duidelijk dat deze canonisatieprocedure dieper bekeken moet worden.

We hebben een standaardcanonisatieprocedure ontworpen die ons toelaat om alle objecten te canoniseren, indien er een orde op gedefinieerd is: de `NaiveCanonizer`. Objecten van dit type lopen voor een gegeven combinatorisch object alle permutaties af en passen deze één voor één toe op het object. Uiteindelijk wordt het minimale object volgens de orde gekozen dat op deze manier bekomen is.

Hiernaast hebben we nog een ander soort canonizer gebouwd, specifiek voor grafen en gebaseerd op nauty: de `NautyCanonizer`. Meer informatie over nauty kan gevonden worden in de online documentatie [23].

4.5 Orderly

Een eerste implementatie is het algoritme zoals gegeven in Algoritme 2. Aan de hand van een gegeven verzameling objecten (geordend!) wordt een volgende verzameling opgebouwd door ze één voor één uit te breiden. Deze nieuwe verzameling wordt opnieuw gebruikt als basis voor het volgende niveau en is reeds gesorteerd.

Hiernaast hebben we de diepte-eerst techniek (depth-first), beschreven in sectie 3.2.3, toegepast (de klasse `OrderlyDepthFirst`). Om dit te realiseren houden we voor elk niveau in de zoekboom één object bij: het laatst gegenereerde van dat niveau. Vervolgens kunnen we op een recursieve manier alle kinderen beginnen genereren tot op een bepaald niveau.

In beide gevallen heeft het orderly algoritme nood aan de volgende informatie:

- een functie φ die de kinderen van een object berekent,
- een functie die test of een object canonisch is, en
- een functie die bepaalt of een object kleiner is dan het andere.

Deze drie eisen hebben we gegroepeerd in één object: de `OrderlyTools`. Indien we het orderly algoritme willen gebruiken voor het genereren van een bepaald soort van objecten geven we dus een lijst met initiële objecten mee tesamen met een invulling van de orderly-tools. Voor de depth-fitst implementatie hebben we wel nog een aparte φ nodig: eentje die aan de hand van een ouder en kind het volgende kind van deze ouder teruggeeft. Op deze manier kunnen we de kinderen opvragen wanneer ze nodig zijn en hebben we de mogelijkheid om geheugen uit te sparen omdat niet het hele niveau ingeladen is.

4.5.1 Generisch Algoritme

Verder hebben we het generische algoritme voor het genereren van algemene combinatorische objecten aan de hand van vectoren geïmplementeerd. Dit algoritme werd beschreven in sectie 3.2.1. Het initiële object is in dit geval de 0-vector en de orderly-tools zijn geïmplementeerd zoals uitgewerkt in de betreffende sectie.

Om dit algoritme te laten lopen zijn er twee zaken nodig:

- een groep permutaties, die afhankelijk zijn van wat we willen genereren, en
- een `Devectorizer`.

Deze laatste is een object dat een vector omzet naar het overeenkomstige combinatorisch object. In ons geval hebben we een `SimpleGraphDevectorizer` geschreven die een graaf opbouwt uit een gegenereerde vector. Voor

de permutatiegroep hebben we een `GraphPermutationConverter`, deze genereert alle permutaties op de vectoren die geïnduceerd worden door deze op de graaf. Herinner dat de vectoren een andere lengte hebben dan het aantal knopen in de graaf en dat de knooppermutaties (grootte n) omgezet dienen te worden tot permutaties op de bogen (grootte $n * (n-1)$).

4.6 Rapportering

Omdat het wenselijk is gegevens te verkrijgen die ons toelaten de algoritmen te vergelijken, hebben we elk van de algoritmen uitgebreid met een rapporteringsysteem voor grafen.

Telkens er een graaf uit een andere gegenereerd wordt voegen we een nieuw rapport als kind toe. Indien een graaf behandeld is en we gaan omhoog in de zoekboom (backtracken), dan worden al zijn kindrapporten naar de schijf geschreven, verwijderd, en geaccumuleerd in zijn rapport. Op deze manier blijft het geheugengebruik relatief beperkt.

In het rapport worden volgende gegevens verzameld:

- aantal kandidaat kinderen
- aantal geldige kinderen
- aantal geweigerde kinderen
- faalratio van kinderen
- aantal kandidaat afstammelingen
- aantal geldige afstammelingen
- aantal geweigerde afstammelingen
- faalratio van afstammelingen
- tijd om de betreffende graaf te genereren
- tijd om alle kinderen te genereren
- tijd om alle afstammelingen te genereren
- tijd om zijn subtree in de search space te genereren (afstammelingen inclusief hijzelf)
- een afbeelding van zichzelf
- een afbeelding van de ouder
- afbeeldingen van zijn kinderen
- een figuur die zijn subtree in de searchspace voorstelt

4.7 Unit Testing

Om ervoor te zorgen dat het framework naar behoren werkt, hebben we gebruik gamaakt van *Unit Tests*. Hierdoor is de werking van het framework betrouwbaar en werden eventuele bugs veel sneller een gemakkelijker ontdekt. We hebben gebruik gemaakt van het ingebouwde C++ Unit Test framework van de XCode programmeeromgeving onder Mac OS X.

4.8 GraphGen

Gebruik makende van ons eigen framework, hebben we een programma geimplementeerd om tests uit te voeren en rapporten te genereren van de gebruikte algoritmen. Het is een console-applicatie en de belangrijkste commando's zijn te vinden in Bijlage G. In het volgende hoofdstuk gebruiken we de resultaten van dit programma om conclusies betreffende de besproken algoritmen te trekken.

Hoofdstuk 5

Vergelijking met behulp van GraphGen

Het framework uit het vorige hoofdstuk werd gebruikt om een programma te implementeren dat het mogelijk maakt om de nodige tests op de algoritmen uit te voeren: *GraphGen*.

GraphGen gebruikt het framework en koppelt alles aan elkaar. Het is een console-applicatie die toelaat op grafen te bouwen en te manipuleren, en hier de algoritmes op te runnen. De applicatie laat toe op een reeks van grafen en permutaties bij te houden in het geheugen, en deze op het gepaste moment in te laden. Verder kunnen we enkele tests runnen die ons een idee geven van de performatie van de verschillende algoritmen. Tot slot is het ook mogelijk om rapporten over de algoritmen te genereren betreffende onder andere de doorzochte search space. Deze output is in de vorm een een doorzoekbare website en is voorzien van afbeeldingen. Bijlage G geeft een overzicht van de belangrijkste commando's van het programma.

In dit hoofdstuk bespreken we de resultaten die het programma ons opgeleverd heeft en proberen we ze te kaderen. Het doel is om een vergelijking van de verschillende algoritmen op te stellen en zo beter te begrijpen hoe ze zich tegenover elkaar verhouden. Deze vergelijking zal gebeuren op verschillende niveaus.

5.1 Constructie

Allereerst viel het ons bij het implementeren op dat het orderly-algoritme aanzienlijk eenvoudig leek om in te vullen. Hiermee bedoelen we dat de voorwaarden die nodig waren om het algoritme "aan de praat" te krijgen veel duidelijker waren: we moeten aangeven met wat voor objecten we werken, en vervolgens een uitbreidingsfunctie, een canonisatieprocedure en een orde definiëren. Eens we een invulling hiervoor hebben zijn we klaar en doet het algoritme al het werk, mits de invulling aan de nodige voorwaarden

voldoet. De moeilijkheid hierbij zit hem echter in het op elkaar afstemmen van de canonisatie en de groeifunctie, zodat de voorwaarden niet geschonden worden.

Bij het generisch algoritme is het nog eenvoudiger: we geven aan welke permutatiegroep er gebruikt kan worden en op welke manier een vector "vertaald" dient te worden naar een object. Zodra we deze zaken hebben ingevuld kan het algoritme van start gaan en alle objecten isomorfvrij genereren. Er is wel de nodige zorg vereist bij het opstellen van de permutatiegroep.

Het algoritme dat McKay presenteert is iets minder doorzichtig. Het lijkt een abstracter algoritme waarbij het niet altijd duidelijk is hoe we iets moeten invullen. De voorwaarden die moeten gelden zijn minder duidelijk, hetgeen het moeilijk maakt om voor willekeurige combinatorische objecten een invulling van het algoritme te construeren. Een voordeel bij de aanpak van McKay is echter dat de canonisatie volledig gescheiden kan worden van de groeiprocedure, zoals duidelijk was bij het genereren van grafen. De groeifunctie wordt hier gegeven door de *upper objects*, en de link met *lower objects*, terwijl de canonisatieprocedure verborgen zit in de functie m. Het voordeel van deze aanpak is dat we zeer snel kunnen wisselen tussen verschillende canonisatieprocedures of zelfs groeifuncties, zonder dat we rekening moeten houden met de andere.

5.2 Tests

De algoritmen waarmee we de tests hebben uitgevoerd zijn het generisch orderly algoritme, dat alle grafen op n knopen genereert, en het algoritme van McKay dat alle grafen tot op n knopen genereert. Omdat dit niet echt een eerlijke vergelijking is, hebben we ervoor geopteerd om het generisch algoritme van voor $n = 1, \ldots, k$ te laten lopen, zodat we alle grafen tot op k knopen hebben. Voor McKay laten we dan eveneens alle grafen tot op k knopen genereren. Op deze manier genereren beide algoritmen dezelfde grafen en kunnen we een beter beeld vormen.

Het is echter belangrijk om in te zien dat de algortimen met een andere canonisatiefunctie en/of groeifunctie zich misschien heel anders gaan gedragen dan in deze setting, en dat de resultaten bijgevolg sterk kunnen variëren.

De tests zijn uigevoerd op een MacBook met een processor van 2.4 GHz en 4 GB RAM-geheugen (677 MHz) onder Mac OS X Leopard (10.5.6).

5.2.1 Orderly

We hebben het orderly algoritme achtereenvolgens alle grafen op $2, 3, 4, 5$ en 6 knopen laten genereren. Het resultaat is zichtbaar in Tabel 5.1. De grafen waarvan we vertrokken zijn werden in de telling niet in rekening gebracht. Dit is in elke run de lege graaf op n knopen. Om het algoritme beter te

n	geldige grafen	ongeldige grafen	totaal	faalpercentage	tijd
2	1	0	1	0%	0,00005s
3	3	3	6	50%	0,00029s
4	10	14	24	58%	0,00111s
5	33	58	91	63%	0,01213s
6	155	238	393	60%	0,38287s

Tabel 5.1: Runs van het generisch orderly algoritme.

n	geldige grafen	ongeldige grafen	totaal	faalpercentage	tijd
3	4	4	7	57%	0,000340s
4	14	18	31	58%	0,001454s
5	47	76	122	62%	0,013588s
6	202	314	515	61%	0,396461s

Tabel 5.2: Geaccumuleerde runs van het generisch orderly algoritme.

kunnen vergelijken met dat van McKay, beschouwen we de geaccumuleerde data, weergegeven in Tabel 5.2. Hierbij zijn ook de initiële objecten niet in beschouwing genomen in de aantallen.

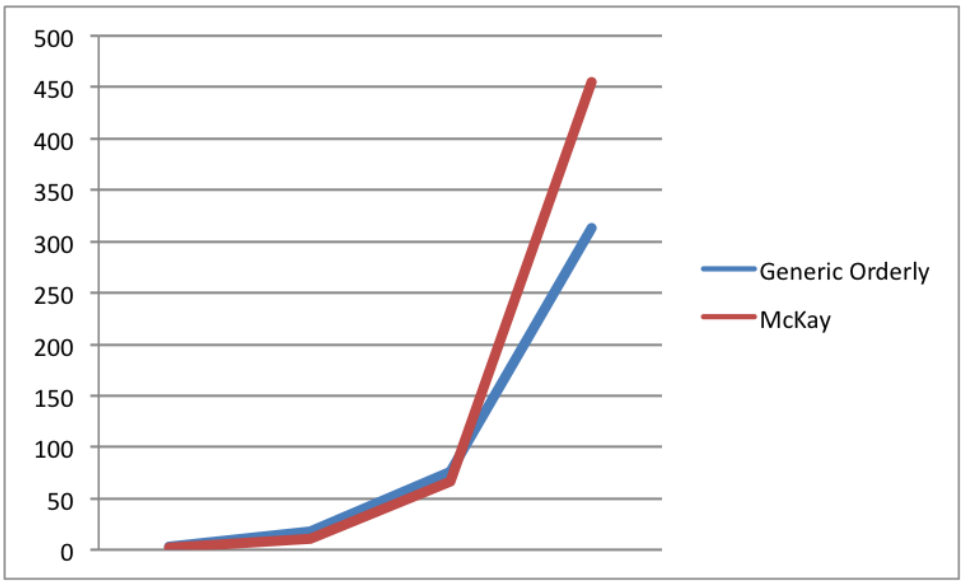
We hebben bij de uitvoering van het algoritme ook gecontroleerd of de orde-test die vóór de canoniciteitstest gebeurt effect had. Dit was niet het geval in de setting van dit generisch algoritme: er werd geen enkele keer een object gegenereerd dat niet groter was dan een reeds gegenereerd object op hetzelfde niveau. Dit sluit echter niet uit dat deze test bij een andere invulling van het orderly framework wel nuttig blijkt. Het geeft echter aan dat de uitbreidingsfunctie goed gedefinieerd is en goed zijn werk doet: hij zorgt niet voor herhalingen van objecten.

5.2.2 McKay

Dit algoritme hebben we laten runnen voor waarden voor $n = 3, 4, 5, 6$, zodat we het kunnen vergelijken met de geaccumuleerde data van het orderly algoritme. De verkregen data is te zien in Tabel 5.3. Het McKay algoritme werd hier in combinatie gebruikt met het naief canonisatiealgoritme.

n	geldige grafen	ongeldige grafen	totaal	faalpercentage	tijd
3	6	2	8	25%	0,001417s
4	17	11	28	39%	0,010311s
5	51	67	118	56%	0,261635s
6	207	455	662	68%	14,4004s

Tabel 5.3: Runs van het algoritme van McKay.



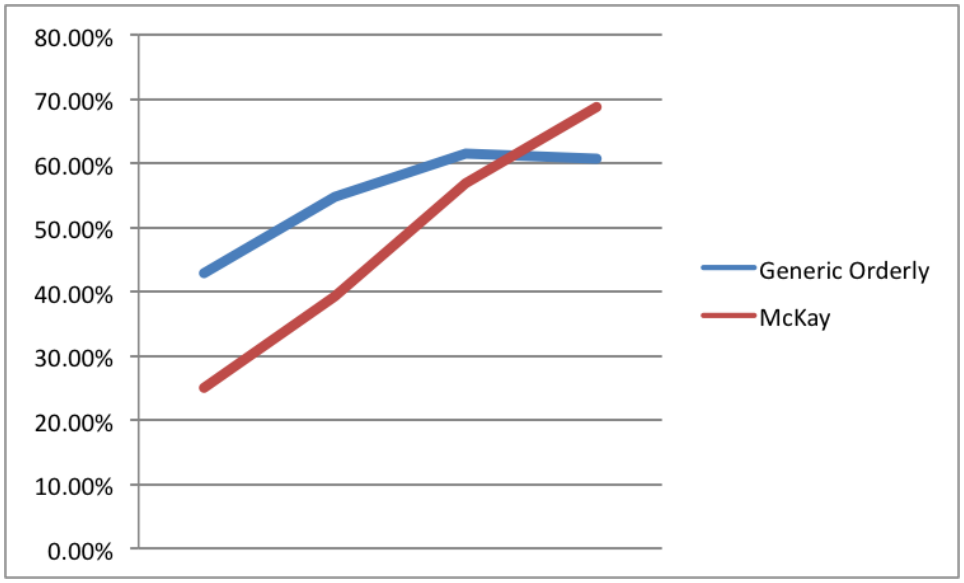
Figuur 5.1: Aantal niet-canonische objecten.

5.2.3 Conclusie

Bij het orderly algoritme zien we dat het percentage aan gewijgerde objecten reeds hoog is van het begin af, terwijl dat bij het algoritme van McKay geleidelijk aan toeneemt. Het geeft aan dat er steeds meer en meer onnodige objecten gegenereerd worden, in verhouding tot de geldige. In Figuur 5.1 is te zien hoe het aantal objecten dat faalt bij het algoritme van McKay sterker exponentieel stijgt dan bij het orderly algoritme. Dit impliceert dat deze aanpak ons voor grotere n veel meer niet-canonische objecten zal opleveren om te controleren. Bij het orderly-algoritme stijgt dit eveneens exponentieel, maar net iets minder snel. We merken ook op dat het McKay-algortime in combinatie met een andere canonisatie nog steeds dit aantal nutteloze grafen genereert, omdat dit aantal afhangt van het aantal automorfismeorbits van *upper objects*.

Het exponentieel stijgen van beide curves wordt verantwoord door het aantal grafen op n knopen, dit stijgt namelijk ook zeer snel. Het percentage foute objecten geeft ons meer informatie. De curves in Figuur 5.2 geven aan dat het orderly algoritme afvlakt rond $n = 6$, en het McKay algoritme gaat hier in stijgende lijn verder. Dit geeft aan dat het McKay algoritme in verhouding tot de geldige objecten steeds meer en meer ongeldige objecten gaat genereren naarmate n groeit.

Tot slot bekijken we nog een plot dit de tijd van uitvoering aangeeft (Figuur 5.3). Het is duidelijk dat deze exponentieel zijn in het geval van alle algoritmen. We hebben hier ook een run van het McKay algoritme met nauty als canonisatietechniek bijgevoegd. Wat betreft dit laatste zien we



Figuur 5.2: Percentage niet-canonische objecten.

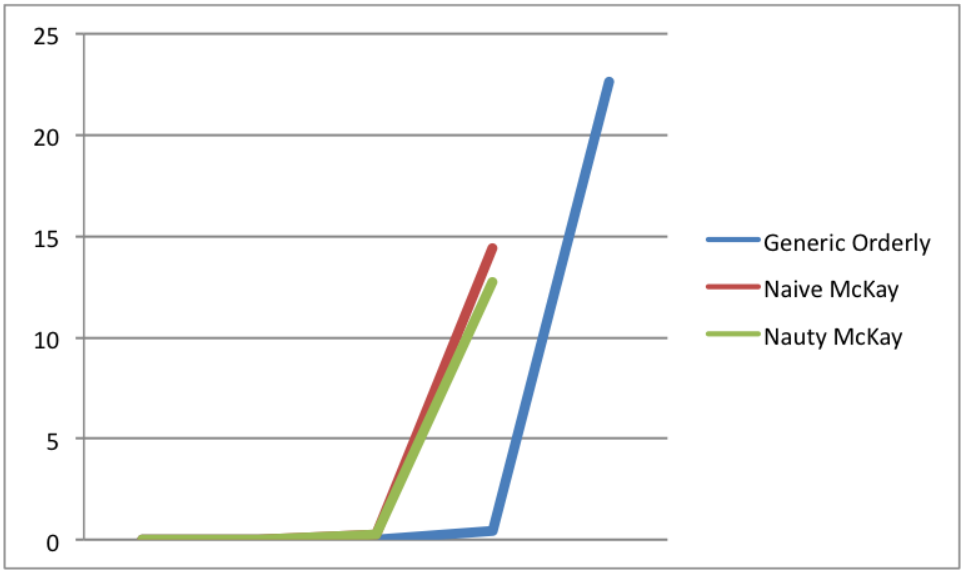
dat deze toch wel een snelheidswinst geeft, maar duidelijk niet zorgt dat het exponentiële karakter verdwijnt.

5.3 Aantal kinderen

Een ander aspect van de algoritmen waarnaar we gekeken hebben is het aantal kinderen dat gegenereerd werd. In Figuur 5.4 zien we boxplots die aangeven hoe de distributie van het aantal kinderen is. Ze geven weer hoe dit zit voor het algoritme van McKay voor $n = 3, 4, 5, 6$. Het is duidelijk dat hierbij een stijging te merken is van het aantal kinderen. We mogen dus concluderen dat, hoe dieper we in de boom gaan, hoe meer kinderen er voor elke graaf gegenereerd worden. Dit is ook logisch, omdat er volgens het gebruikte uitbreidingsprincipe meer mogelijkheden zijn om een grote graaf te laten "groeien". Voor $n = 6$ bevindt het grootste deel zich tussen 15 en 40.

Wanneer we kijken naar de geaccumuleerde boxplots van orderly in Figuur 5.6 (Figuur 5.5 geeft de niet-geaccumuleerde data weer), dan zien we dat de verdeling meer stabiel blijft, er worden meestal slechts een paar kinderen gegenereerd, en het maximaal is hier 15.

Deze discripantie tussen de twee algoritmen is een mogelijke verklaring voor het aantal niet-canonische kinderen: de zoekboom van het orderly algoritme zal waarschijnlijk smaller zijn dan die van het algoritme van McKay, hetgeen de pruning meer efficiënt kan maken. Deze vermoedens worden bevestigd door de zoekruimtes die gegenereerd worden door ons programma.



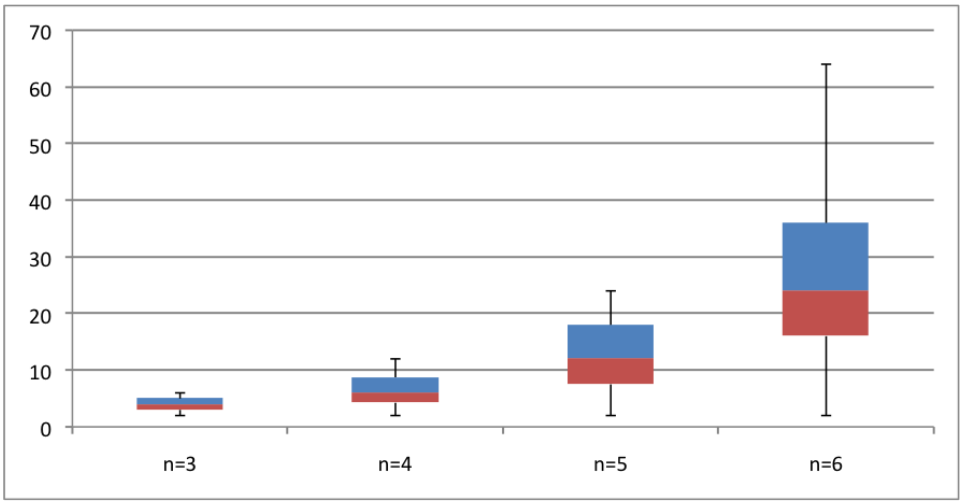
Figuur 5.3: De uitvoeringstijd van de algoritmen.

Enkele van deze zijn te vinden in Bijlage C. Merk hierbij wel op dat de zoekbomen voor orderly enkel over grafen met n knopen gaan en die van McKay over grafen kleiner of gelijk aan n knopen.

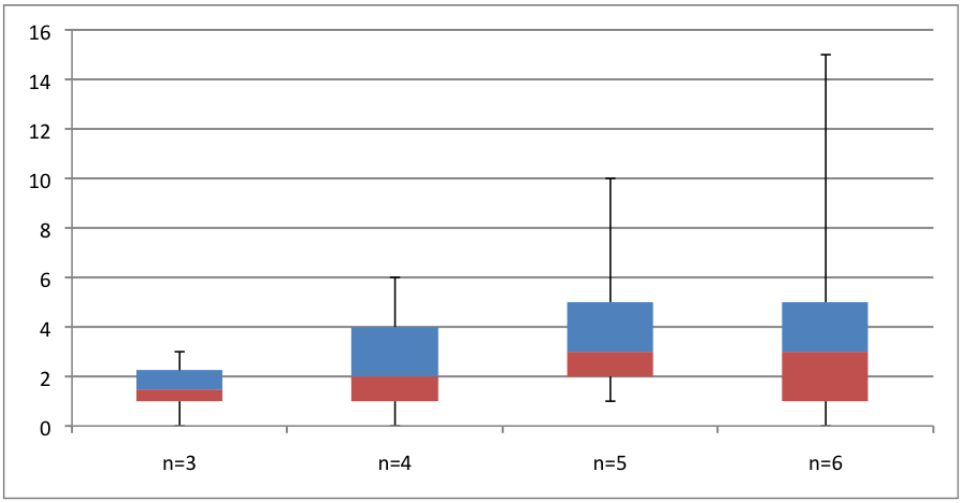
5.4 Conclusie

We hebben gezien dat het orderly algoritme eenvoudiger in te vullen is dan het algoritme van McKay. Verder geeft de generische aanpak ons goede resultaten wat betreft snelheid en aantal niet-canonische afstammelingen. Wat betreft de zoekbomen hebben we ondervonden dat deze van het generisch orderly algoritme smaller zijn dat die van het algortime van McKay. We hebben ook opgemerkt dat het aantal nutteloze objecten bij McKay te maken heeft met de orbits van *upper objects* die gevormd worden. Om er minder te genereren zullen we de uitbreidingsfunctie dus moeten beperken.

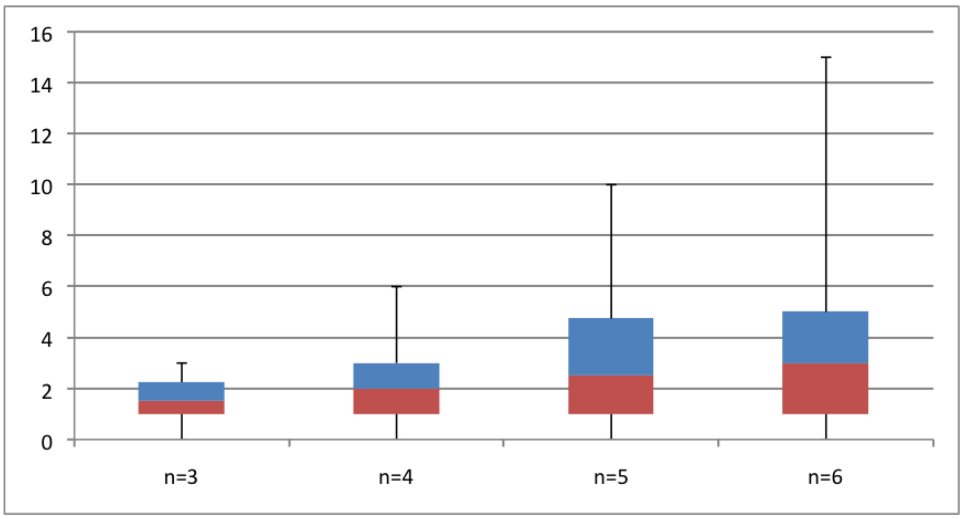
We willen opmerken dat we bij de implementatie vooral de nadruk hebben gelegd op het begrijpen van de algoritmen. Verder wijzen we erop dat de verschillen in snelheid mogelijk nog opgelost kunnen worden door gebruik te maken van een meer efficiënte implementatie die meer toegespitst is op grafen, of door andere canonisatie- en/of groeifuncties te gebruiken.



Figuur 5.4: Mckay.



Figuur 5.5: Orderly.



Figuur 5.6: Orderly geaccumuleerd.

Hoofdstuk 6

Graafgrammatica's

In de vorige hoofdstukken werden technieken gepresenteerd die ons toelaten om objecten, meer bepaald grafen, op een isomorfvrije manier te genereren. Het is vaak wenselijk dat we een bepaalde klasse van grafen genereren waarbij alle leden aan een bepaalde eigenschap voldoen. Denk hierbij bijvoorbeeld aan alle grafen met een even aantal knopen, of alle mogelijke cykels. Dergelijke klassen kunnen interessant zijn in de context van *graph mining*: Indien we enkel geïnteresseerd zijn in het zoeken naar voorkomens van grafen met een bepaalde eigenschap, dan willen we ook dit efficiënt kunnen oplossen. We hebben bijgevolg een manier nodig om alle grafen die de betreffende eigenschap bezitten op te sommen.

Een voor de hand liggende manier om dit te doen is door, met behulp van de besproken algoritmen, alle grafen isomorfvrij te genereren en één voor één te controleren op de betreffende eigenschap. Op deze manier bekomen we een methode om al de grafen uit de betreffene klasse isomorfvrij te enumereren, maar het is duidelijk dat er veel overbodige grafen gegenereerd worden. Herinner uit de vorige hoofdstukken dat de algoritmen gebruik maken van een zeer complexe canonisatieprocedure, die voor elke kandidaat-graaf aangeroepen wordt. Wanneer nu onze klasse grafen een zeer kleine subklasse is van de verzameling van *alle* grafen, is het eenvoudig in te zien dat er veel overbodig werk gebeurt.

Hoe kan dit efficiënter aangepakt worden? Een oplossing bestaat eruit ervoor te zorgen dat de generatieprocedure zelf enkel deze klasse van grafen tracht te genereren. Mckay [24] geeft een voorstel van een overerfbare eigenschap (zie ook sectie 3.4), die bruikbaar is in combinatie met het algoritme. Op deze manier zorgen we dat enkel de juiste grafen gegenereerd worden door *upper objects* te elimineren.

Voor orderly algoritmen kunnen we een gelijkaardige notie bedenken, beschouw het begrip *augmentation hereditary*, met de terminologie uit sectie 3.2 in het achterhoofd:

Definitie 6.1 (Augmentation hereditary eigenschap)**.** Een eigenschap ξ van

een canonisch object is *augmentation hereditary* indien volgende eis geldt:

$$x \in \mathcal{L}_{q+1} \wedge x \text{ voldoet aan } \xi \Rightarrow f(x) \text{ voldoet aan } \xi.$$

Merk op dat de functie $f(x)$, die de eerste (volgens een zekere orde op de objecten) canonische (!) ouder van een object aangeeft, gebruikt wordt in deze definitie. Dit laatste geeft aan dat het *augmentation hereditary* zijn van een eigenschap afhankelijk is van het gebruikte constructieproces van de objecten (de uitbreidingsfunctie φ). Gegeven een orderly algoritme, dat een bepaalde notie van canoniciteit ($canon(X)$) gebruikt tesamen met een groeifunctie φ, dan construeren we een nieuwe notie van canoniciteit die zorgt dat alle gegenereerde object eigenschap ξ bezitten: Een object X is canonisch indien $canon(X) = true$ en X voldoet aan de eigenschap ξ. De nieuw bekomen canonische vorm kan dadelijk gebruikt worden in het algoritme, zonder de groeifunctie φ aan te passen. Door alle eigenschappen van een orderly algoritme te controleren kunnen we ons ervan verzekeren dat het algoritme correct werkt:

1. Elk canonisch object moet door een kleiner canonisch object gegenereerd kunnen worden: dit is nog steeds zo, omdat we weten dat volgens de oude definitie er een canonische ouder bestaat en wegens het gelden van de *augmentation hereditary*-eigenschap weten we ook dat de kleinste canonische ouder voldoet aan de betreffende eigenschap ξ. Er is dus zeker een canonisch object dat we kunnen uitbreiden om het beschouwde te bekomen.

2. Een de monotoniciteitsvoorwaarde verandert niets, de ouders van de grafen blijven nog steeds dezelfde.

3. Ook de uitbreidingsfunctie φ verandert niet.

Voorbeeld 6.2. *Herinner het generisch algoritme dat gebruikt kan worden om grafen op n knopen te construeren door gebruik te maken van vectoren, gepresenteerd in sectie 3.2.1. De uitbreidingsfunctie voegt telkens één boog toe aan een reeds ontdekte graaf. Een* augmentation hereditary *eigenschap is bijvoorbeeld: "de graaf bevat geen cykel van 3". Het is duidelijk dat wanneer een graaf deze eigenschap bezit, elke ouder deze eigenschap noodzakelijkerwijs ook bezit, omdat deze minder bogen heeft en een subgraaf is van de beschouwde graaf.*

Een eigenschap die in deze context niet augmentation hereditary *is, is bijvoorbeeld: "de graaf bevat een even aantal bogen". Omdat elke mogelijke ouder precies één boog minder heeft, is de eigenschap niet waar voor $f(X)$, waarbij X voldoet aan de eigenschap.*

De eigenschappen die we beschouwen moeten dus "generatief" zijn. Hiermee bedoelen we dat het mogelijk is op een natuurlijke manier te zeggen

hoe de grafen die aan de eigenschap voldoen kunnen voortkomen uit andere grafen die ook aan de eigenschap voldoen. Hiertoe beschouwen we graafgrammatica's, een soort contextvrije grammatica's die toelaten om een klasse van grafen te beschrijven. Courcelle [12] introduceert het begrip van een HR-*equational* graafeigenschap als volgt: een klasse van grafen is HR-*equational* indien ze gegenereerd kan worden door een graafgrammatica. De uitleg die nu volgt tracht op een informele manier een idee te geven van deze begrippen.

Beschouw een relationele structuur als input (bijvoorbeeld een tabel uit een database), dan is een *transductie* een omzetting van deze structuur naar een graaf. De omzettingsfunctie die dit doet noemen we ϕ. Wanneer we de relationele structuur als een tabel beschouwen, dan kan een transductie een query zijn die ons een tabel teruggeeft die de graafvoorstelling bevat. Courcelle beschouwt als "querytaal" de *Monadic Second-order Logic* (MSO) en beschouwt als inputs alle mogelijke binaire bomen. Zij ϕ nu een MSO-transductie, beschouw dan

$$GL(\phi) = \{\phi(T) \mid \text{T is een binaire boom}\}.$$

$GL(\phi)$ bestaat met andere woorden uit de grafen, die bekomen worden door alle binaire bomen door middel van ϕ af te beelden op een graaf. De verzameling kan gezien worden als het "beeld" van ϕ op alle mogelijke binaire bomen. Courcelle zijn *equationality theorem* zegt ons dat een verzameling grafen $\mathcal{V}$ HR-*equational* is als en slechts als er een ϕ bestaat zodat $\mathcal{V} = GL(\phi)$.

We introduceren in dit hoofdstuk op een intuïtieve manier de notie van graafgrammatica's, omdat de eigenschap of een graaf behoort tot zulke grammatica een generatief karakter lijkt te hebben. Graafgrammatica's voldoen aan de HR-*equational*-eigenschap. De voorstelling van deze grammatica's gebeurt aan de hand van contextvrije grammatica's en reguliere boomgrammatica's, die toelaten om op een eenvoudige manier te komen tot een grammatica die grafen kan genereren. In de volgende hoofdstukken trachten we manieren te vinden om alle grafen die door zo een grammatica gegenereerd kunnen worden te produceren op een efficiënte manier, gebruik makend van de tot hiertoe geziene methodes, tesamen met enkele nieuwe aanpakken.

6.1 Contextvrije grammatica's

Sipser [31] geeft een zeer goede beschrijving van de werking van contextvrije grammatica's en Courcelle [12] geeft in zijn inleiding een beknopte uitleg, die voldoende is om het concept te begrijpen. De inhoud van deze sectie is voor een groot deel gebaseerd op deze twee werken.

Een contextvrije grammatica is een eindige verzameling van herschrijfregels, die gedefinieerd worden aan de hand van twee alfabetten: een *terminal*

alfabet $\mathcal{A}$ en een *non-terminal alfabet* $\mathcal{N}$. Een herschrijfregel of *productie* is van de vorm $S \to T$ waarbij $S \in \mathcal{N}$ en $T \in (\mathcal{N} \cup \mathcal{A})^*$[1]. Het tweede deel bestaat dus uit een geordende reeks van terminals en/of nonterminals. Een contextvrije grammatica heeft ook altijd een start-nonterminal, die in het algemeen vooraan staat in de eerste productie. Het voorgaande leidt tot de volgende definities:

Definitie 6.3 (Contextvrije grammatica)**.** Een contextvrije grammatica is een 4-tupel $G_r = (V, \Sigma, R, S)$, waarbij

1. V een eindige verzameling is, de nonterminals,
2. Σ een eindige verzameling is, met $V \cap \Sigma = \emptyset$, de terminals,
3. R een verzameling producties is die bestaat uit een nonterminal en een aaneenschakeling (string) van nonterminals en terminals, en
4. $S \in V$, de start-nonterminal, of het startsymbool.

Voorbeeld 6.4. *Beschouw volgende contextvrije grammatica* G_1*:*

$$A \to 0A1$$
$$A \to B$$
$$B \to 2$$

Hierbij is $\mathcal{N} = \{A, B\}$ *en* $\mathcal{A} = \{0, 1, 2\}$*. De start-nonterminal is in dit geval* A*.*

Beschouw string u, v, w die uit terminals en nonterminals bestaan, en een productie $A \to w$. We zeggen dan dat *uwv bekomen wordt* uit uAv en noteren dit als $uAv \Rightarrow uwv$. Van twee strings u, v van deze vorm zeggen we dat v *gegenereerd* kan worden uit u indien er een reeks $u_1, \ldots, u_k$ bestaat zodat

$$u \Rightarrow u_1 \Rightarrow \ldots \Rightarrow u_k \Rightarrow v.$$

We noteren dit met $u \Rightarrow^* v$.

Voorbeeld 6.5. *Beschouw opnieuw de grammatica uit voorbeeld 6.4. Een mogelijke toepassing van de regels om de string* `00211` *te bekomen is:*

$$A \Rightarrow 0A1 \Rightarrow 00A11 \Rightarrow 00B11 \Rightarrow 00211.$$

Er geldt bijgevolg dat $A \Rightarrow^*$ `00211`*.*

[1] A^* is de verzameling van alle mogelijke concatenaties van elementen uit A. Meer informatie over reguliere expressies kan gevonden worden in Sipser [31].

Definitie 6.6. De *taal van een contextvrije grammatica* G_r, genoteerd $L(G_r)$, wordt gegeven door:

$$L(G_r) = \{w \in \Sigma^* \mid S \Rightarrow^* w\}.$$

Met andere woorden, alle strings die *gegenereerd* kunnen worden uit het startsymbool.

Wanneer we een afleiding van het startsymbool tot een string over terminals beschouwen, dan is het mogelijk deze afleiding te bekijken als een boom. De wortel van de boom is het startsymbool zelf en de kinderen zijn alle terminals en nonterminals die het startsymbool vervangen (de linkerkant van de gekozen productie). De verkregen nonterminals krijgen op hun beurt op dezelfde manier kinderen toegewezen. De bladeren van de boom zijn de terminals en indien we ze in volgorde samennemen verkrijgen we de string die bekomen werd door af te leiden volgens de gekozen productieregels. Zo een boom noemen we een *afleidingsboom* of *parse tree*. We merken reeds op dat een string die door een grammatica gegenereerd wordt niet noodzakelijk een unieke afleidingsboom heeft.

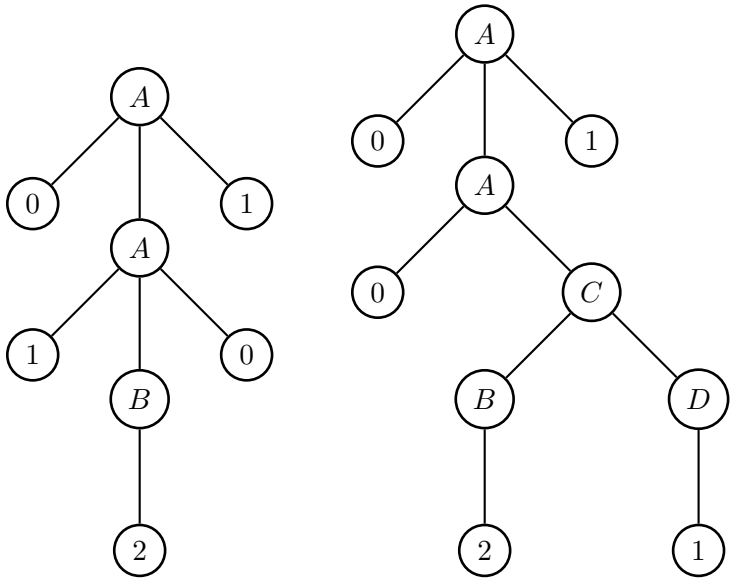
Voorbeeld 6.7. *Voortbouwend op voorbeeld 6.5 construeren we aan de hand van de afleidingsstappen de boom die te zien is in Figuur 6.1a. Indien we volgende regels aan de grammatica toevoegen:*

$$A \to 0CD$$
$$C \to BD$$
$$D \to 1$$

dan kunnen we de gegeven string ook op een andere manier bekomen, zoals de afleidingsboom in Figuur 6.1b aangeeft.

6.2 Reguliere boomgrammatica's

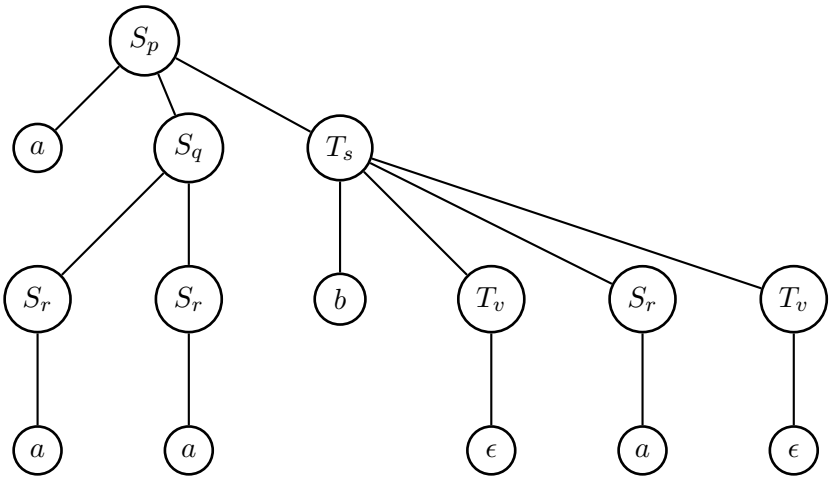
In de vorige sectie hebben we contextvrije grammatica's beschreven en aangegeven dat ze gebruikt kunnen worden om een bepaalde klasse van strings te genereren. Beschouw hetvolgende voorbeeld van een grammatica G_1, waarbij de letter voor elke productie enkel een naam voor de betreffende



(a) De afleidingsboom van `00211`

(b) Een andere afleidingsboom van `00211` volgens de uitgebreide grammatica

Figuur 6.1: Twee afleidingsbomen voor `00211`.



Figuur 6.2: Een afleidingsboom voor `abaaa`.

regel is:

$$
\begin{aligned}
&p : S \to aST \\
&q : S \to SS \\
&r : S \to a \\
&s : T \to bTST \\
&t : T \to a \\
&u : T \to c \\
&v : T \to \epsilon^2
\end{aligned}
$$

Een voorbeeld van een string die gegenereerd wordt door deze grammatica is `aaaba`. In Figuur 6.2 zien we de afleidingsboom voor deze string. De toegepaste afleidingsregels zijn aangeduid met de subscripts.

Het is mogelijk om de grammatica en dit generatieproces op een alternatieve manier te bekijken, door enkel de nonterminals te beschouwen. We lichten dit toe aan de hand van de zojuist gegeven grammatica.

Elke productie in de grammatica kunnen we bekijken als een functie, waarbij de ariteit (aantal argumenten) gelijk is aan het aantal nonterminals die voorkomen aan de rechterkant van de regel. In ons geval hebben we zeven zulke functies: p, q, r, s, t, u, v, waarvan de ariteiten respectievelijk overeenkomen met $2, 2, 0, 3, 0, 0, 0$. De betekenis die we aan deze functies

[2] ϵ staat voor de lege string.

geven is de volgende:

$$\begin{aligned} p(x,y) &= axy \\ q(x,y) &= xy \\ r() &= a \\ s(x,y,z) &= bxyz \\ t() &= a \\ u() &= c \\ v() &= \epsilon \end{aligned}$$

De functie p bijvoorbeeld "plakt" zijn argumenten achter een `a`, de functie u geeft de (constante) string `c` terug. Deze "constante" functies met ariteit 0 noteren we vaak zonder haakjes, om een beter overicht te verkrijgen. Merk op dat het mogelijk is om alle strings die door de grammatica gegenereerd worden voor te stellen door een *term* die uit de zojuist gegeven functies bestaat:

$$\|p(q(r,r),s(v,r,v))\| = \texttt{aaaba}.$$

Hierbij is $p(q(r,r),s(v,r,v))$ een term en $\|t\|$ de evaluatie (betekenis) van een term t. We gebruiken deze notatie om duidelijk het verschil aan te geven tussen de "string" $p(q(r,r),s(v,r,v))$ en de betekenis ervan. Om zo een term te bekomen, kijken we in een afleidingsboom welke regels we hebben toegepast en schrijven deze neer. Een *term* is in dit geval een geneste toepassing van functie-symbolen. Met een *geldige term* bedoelen we eentje waarbij de nesting (plaatsing van haakjes) van de functiesymbolen correct is en waarbij de ariteiten van de functies juist gebruikt worden. Indien we de verzameling functies P noemen, dan is $T(P)$ de verzameling van *geldige* termen over P.

Voorbeeld 6.8. *In ons geval is $P = \{p,q,r,s,t,u,v\}$ met de daarbij horende ariteiten. Nu is $s(t,p(r,v),u) \in T(P)$, dus een geldige term, en $s(t,p(r,v),u,t) \notin T(P)$ omdat de ariteit van s niet gelijk is aan 4, bijgevolg geen geldige term. Ook zijn de strings "$s()t$" en "$s)$" geen geldige termen omdat ze geen geldige vorm hebben.*

Omdat we een alternatieve voorstelling voor onze strings in de grammatica gevonden hebben, is het in principe mogelijk om enkel de termen genereren en deze vervolgens evalueren om de gepaste string te bekomen.

We stellen een grammatica G_1' op die dit soort termen genereert:

$$
\begin{aligned}
p &: S \to p(S,T) \\
q &: S \to q(S,S) \\
r &: S \to r() \\
s &: T \to s(T,S,T) \\
t &: T \to t() \\
u &: T \to u() \\
v &: T \to v()
\end{aligned}
$$

Deze grammatica werd bekomen door alle terminals te schrappen uit de regels en elke nonterminal als parameter van de functie voor die regel te gebruiken (herinner dat de ariteit van de functies gekozen werd aan de hand van de nonterminals). G_1' genereert nog steeds strings, maar deze hebben de vorm van een geldige term. We kunnen bijgevolg ook zeggen dat G_1' termen genereert.

Merk op dat ook de eerste term uit voorbeeld 6.8 niet in $L(G_1')$ zit, omdat het startsymbool nooit uitbreid kan worden naar een string die s vanvoor heeft staan. Wanneer we $s(t, p(r, v), u)$ evalueren bekomen we de string *baaac*, die niet in $L(G_1)$ zit. Zonder te bewijzen nemen we aan dat de bovenstaande constructie van een nieuwe grammatica, na evaluatie van de termen, dezelfde strings genereert.

Opmerking 6.9. Merk op dat een lidmaatschapstest van de strings die we bekomen door de termen te evalueren hier niet op dezelfde manier kan gebeuren als bij "gewone" contextvrije grammatica's. Het is namelijk niet duidelijk hoe we een string kunnen omzetten naar een term. En indien het wel zo is, hebben we dan wel één van de termen gekozen die gegenereerd wordt door de grammatica? Zijn er nog anderen die mogelijk wel gegenereerd worden? Dit lijkt geen eenvoudige opgave. We komen hier later nog op terug.

Wanneer we in staat zijn om de strings van G_1' te enumereren, dan is het duidelijk dat we, door evaluatie van de termen, ook alle strings van G_1 kunnen opsommen. Het is belangrijk om in te zien dat we een term-genererende grammatica niet enkel kunnen gebruiken om strings te genereren. Stel dat onze evaluatie-functie in plaats van strings samen te voegen nu eens met grafen zou werken!? Bovenstaande kijk op contextvrije grammatica's geeft ons bijgevolg een manier om grafen te produceren die we in de volgende sectie zullen uitwerken. Hiertoe zijn echter nog enkele formele definities nodig.

Definitie 6.10 (Ranked alfabet)**.** Een *ranked alfabet* Σ is een eindige verzameling symbolen, waarbij elk van deze symbolen een bepaalde ariteit heeft. Σ kan onderverdeeld worden in subsets van eenzelfde ariteit Σ_a:

$$\Sigma = \Sigma_1, \Sigma_2, \ldots.$$

Gegeven een ranked alfabet Σ, dan wordt met $\mathcal{T}(\Sigma)$ de verzameling mogelijke bomen over Σ aangeduid. Voor elke interne knoop in de boom $v \in \Sigma_{a_v}$ moet gelden dat het aantal kinderen gelijk is aan a_v (de ariteit van v). De bomen komen dus in essentie overeen met alle geldige termen (een term die bestaat uit functiesymbolen kan, zoals reeds gesuggereerd werd, inderdaad gezien worden als een boom). Hiernaast beschouwen we nog een verzameling $\mathcal{N}$, met $\mathcal{N} \cap \Sigma = \emptyset$. De symbolen in $\mathcal{N}$ hebben allen ariteit 0. $\mathcal{T}(\Sigma, \mathcal{N})$ is gedefinieerd als alle bomen over $\Sigma \cup \mathcal{N}$.

Definitie 6.11 (Reguliere boomgrammatica)**.** Een *reguliere boomgrammatica* is een 4-tupel $(\mathcal{N}, \Sigma, R, S)$, waarbij

1. $\mathcal{N}$ een eindige verzameling is, de nonterminals,
2. Σ een eindig ranked alfabet is, met $\Sigma \cap \mathcal{N} = \emptyset$, de terminals,
3. R een verzameling producties is die bestaat uit een nonterminal en een element uit $\mathcal{T}(\Sigma, \mathcal{N})$, en
4. $S \in V$, de start-nonterminal, of het startsymbool.

Reguliere boomgrammatica's zijn niets anders dan gewone contextvrije grammatica's, buiten het feit dat ze geen strings maar termen produceren. Aan deze termen kunnen we een betekenis koppelen door ze op een bepaalde manier te evalueren. Herinner dat deze termen geschreven kunnen worden in boom-vorm, en dat de grammatica's in essentie bomen produceren, vandaar de naam reguliere boomgrammatica's.

Om te controleren of een term aanvaard wordt beschouwen we de boom die de term voorstelt. De boom bestaat uit symbolen van het beschouwde rankes alfabet. Alle bladeren van de boom kunnen gematcht worden met symbolen met ariteit 0. Aan elk blad kennen we de verzameling alle mogelijke non-terminals toe die de betreffende constante kunnen afleiden. Vervolgens gaan we elke knoop behandelen voor wie de kinderen allemaal een dergelijke verzameling toegekend gekregen hebben. Voor deze knopen zoeken we de nonterminals die de kind-nonterminals en/of terminals kunnen produceren. Op deze manier worden de knopen van de boom één voor één gekoppeld aan de verzameling nonterminals waarmee ze kunnen "matchen". Indien het startsymbool in de verzameling zit die gekoppeld is aan de wortel van de boom, dan mogen we besluiten dat de hele boom/term uit de betreffende grammatica afgeleid kan worden. Het is duidelijk dat de boomstructuur ons enorm helpt bij het herkennen van de termen.

Stelling 6.12. *Het is mogelijk om te controleren of een gegeven term gegenereerd wordt door een reguliere boomgrammatica.*

6.3 Graafgrammatica's

Zoals reeds aangegeven is het mogelijk om de termen van een reguliere boomgrammatica een betekenis te geven die geen string is. In deze sectie construeren we ranked alfabet voor grafen, waarbij we aan de gebruikte functiesymbolen een betekenis koppelen. Op deze manier construeren we een soort framework dat ons toelaat om grammatica's te bouwen die grafen produceren.

De aanpak die we gevolgd hebben is deze gepresenteerd door Blockeel et al. [5]. Beschouw een eindige verzameling $\mathcal{E}$ met booglabels en een gemeenschappelijke verzameling knopen $\mathcal{K}$. Een graaf is nu een triple (V, E, S) waarbij $V \subseteq \mathcal{K}$, $E \subseteq V \times \mathcal{E} \times V$ en $S \subseteq V$. De verzameling V is gelijkaardig aan deze die we gebruikten bij onze eerdere voorstelling van een graaf. De bogen worden uitgebreid met een label en zijn geordende tupels, wat aangeeft dat we met gerichte grafen werken (anders zouden een set bestaande uit twee knopen tesamen met een label een betere keuze zijn in plaats van deze triples). Ook is er een nieuwe verzameling bijgekomen die iets aangeeft over de knopen, namelijk de *sources* S, die belangrijk zijn voor de evaluatie van termen. Indien $S = \emptyset$, dan noemen we de betreffende graaf een *gewone* graaf. We duiden deze verzamelingen voor een graaf G respectievelijk aan met $V(G)$, $E(G)$, $S(G)$.

Het is de bedoeling dat we termen construeren die een bepaalde vorm hebben. Deze termen noemen we *graafexpressies*. De correcte vorm hiervan wordt gegeven door de volgende contextvrije grammatica[3]:

$$e \to a(x, y) \tag{6.1}$$

$$e \to e \mid e \tag{6.2}$$

$$e \to (\mathrm{lc}\, x)e \tag{6.3}$$

Hierbij zijn $a \in \mathcal{E}$ en $x, y \in \mathcal{K}$, de grammatica is met andere woorden afhankelijk van de gekozen verzamelingen.

Voorbeeld 6.13. *Enkele voorbeelden van graafexpressies voor* $\mathcal{K} = \{1, 2, 3\}$ *en* $\mathcal{E} = \{a, b\}$ *zijn:*

$$(\mathrm{lc}\, 1)a(1, 2) \mid b(2, 1),$$

$$b(1, 2) \mid (\mathrm{lc}\, 1)a(1, 2) \mid b(2, 3).$$

Opmerking 6.14. De boom die overeenkomt met de term bepaalt de "scope" van de functies. Hiermee bedoelen we, dat als we de term als een boom bekijken, het duidelijk is waar de extra haakjes in de expressie moeten staan. Waar nodig zullen we de graafexpressies aanvullen met extra haakjes, omdat de "platte" stringvoorstelling soms ambigu is. Zo zijn de graafexpressies

$$b(4, 2) \mid ((\mathrm{lc}\, 2)a(1, 2)) \mid b(2, 3)$$

[3]De grammatica wordt hier slechts gebruikt om de opbouw van een extressie te geven en heeft verder niets te maken met de graafgrammatica's die we gaan introduceren.

en

$$b(4,2) \mid (\mathrm{lc}\, 2)(a(1,2) \mid b(2,3))$$

verschillend van elkaar.

We hebben nu een vorm van expressies gedefinieerd, maar we moeten er nog een betekenis aan geven. Dit gebeurt door twee operatoren te gebruiken:

Merge De |-operator dient om twee grafen aan elkaar te "plakken" door naar de sources te kijken. Gegeven twee grafen G_1 en G_2, dan is deze operatie enkel toegelaten indien $V(G_1) \cap V(G_2) \subseteq S(G_1) \cap S(G_2)$. De gemeenschappelijke knopen mogen dus enkel sources zijn. Het resultaat van het toepassen van deze operatie is het volgende:

$$G_1 \mid G_2 = (V(G_1) \cup V(G_2), E(G_1) \cup E(G_2), S(G_1) \cup S(G_2)).$$

Projectie Beschouw een graaf G en een knoop $x \in S(G)$, dan is

$$(\mathrm{lc}\, x)G = (V(G), E(G), S(G) \setminus \{x\})$$

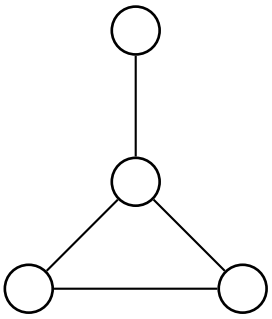
Merk op dat de definitie van de operatoren niets zegt over de evaluatie van de graafexpressies. Dit is het geval omdat de expressies "slechts" bomen zijn en we nog aan moeten geven hoe de evaluatie gebeurt. Dit zullen we doen door de verschillende gebruikte symbolen een betekenis te geven, die afhankelijk is van de argumenten. Vanzelfsprekend zal de merge-functie de merge-operator gebruiken en de local-functie de local-operator. Vooraleer we verder hierop in gaan, hebben we nog nood aan een definitie van isomorfisme tussen de grafen.

Definitie 6.15. een isomorfisme tussen twee grafen G en H is een bijectie $\pi : V(G) \to V(H)$, waarbij

- $\forall x \in S(G) : \pi(x) = x$, π is met andere woorden de identiteit op $S(G)$,
- $\{(\pi(u), a, \pi(v)) \mid (u, a, v) \in E(G)\} = E(H)$, oftewel: alle bogen van G worden op alle bogen van H afgebeeld door π.

Lemma 6.16. *Voor elke twee grafen G en H bestaan er twee grafen G' en H', zodat $G' \cong G$ en $H' \cong H$ en de mergeoperatie $\mid$ toegelaten is op G' en H'.*

Voorgaand lemma kan als volgt ingezien worden: we kunnen, met behulp van knopen uit $\mathcal{K}$ de niet-sources van G en H allemaal anders noemen en op deze manier isomorfe grafen verkrijgen. Het is duidelijk dat ze dan enkel sources gemeenschappelijk kunnen hebben en de merge-operatie bijgevolg toegelaten is. Let wel: we veronderstellen dat $\mathcal{K}$ voldoende groot is.



Figuur 6.3: Een abstracte graaf.

Vervolgens is het mogelijk om een betekenis te koppelen aan een graafexpressie e:

$$\begin{aligned} \|a(x,y)\| &= (\{x,y\},\{(x,a,y)\},\{x,y\}) \\ \|e_1 \mid e_2\| &= \|e_1\| \mid \|e_2\| \\ \|(\mathrm{lc}\, x)e\| &= (\mathrm{lc}\, x)\|e\| \end{aligned}$$

Merk op dat de linkerkant hierbij bestaat uit een graafexpressie (beschouwd als string of boom), de rechterkant gebruikt de operatoren zoals we ze net gedefinieerd hebben. Lemma 6.16 is hier niet onbelangrijk: het geeft aan dat we elke twee grafen kunnen mergen en dat onze "evaluate"-functie bijgevolg op geen enkele input "vastloopt".

Lemma 6.17. *Elke graaf kan geschreven worden als een graafexpressie.*

Bewijs. Neem $G = (V, E, S)$, een expressie die de graaf G voorstelt kan bekomen worden door eerst de merge te nemen van alle bogen $a(x, y)$, zodat $(x, a, y) \in E$, gevolgd door projecties van de vorm $(\mathrm{lc}\, x)$, waarbij $x \notin S$. □

Voorbeeld 6.18. *De (abstracte) graaf uit Figuur 6.3 kan als volgt uitgedrukt worden door een graafexpressie:*

$$(\mathrm{lc}\, 1)(\mathrm{lc}\, 2)(\mathrm{lc}\, 3)(\mathrm{lc}\, 4)(a(1,2) \mid a(2,3) \mid a(2,4) \mid a(3,4)).$$

Merk op dat in de beschouwde graaf de verzameling sources leeg is, vandaar alle locals aan het begin van de expressie.

De constructie die gepresenteerd werd in het bewijs is niet noodzakelijk de meest "zuinige" in de zin van zo weinig mogelijk sources te "verspillen". Hiermee bedoelen we dat we sources kunnen hergebruiken in de voorstelling van een graaf door een graafexpressie. De functies van de verschillende symbolen en de sources zijn nu duidelijk: we "plakken" grafen aan elkaar door de gemeenschappelijke sources als dezelfde knopen te nemen en al de rest van de twee grafen samen te voegen. Vervolgens kunnen we sources

"vergeten" of "anoniem" maken door de local-operatie te gebruiken. Na toepassing van deze operatie op een source is het niet meer mogelijk om deze knoop te gebruiken in een merge-operatie: hij kan niet meer met andere knopen verbonden worden. In een graafexpressie is het dus mogelijk om een vast aantal sources te gebruiken en er een onbeperkt aantal knopen mee te beschrijven. Dit maakt het eenvoudig om in te zien dat een graafexpressie eigenlijk een abstracte graaf beschrijft: de knopen die na evaluatie van de expressie geen sources zijn, zijn niet bekend. Hun identiteit is willekeurig. Er ligt dus met andere woorden geen orde op de knopen, we kunnen ze niet aanduiden. Dit komt omdat de merge-operator verondersteld wordt isomorfe grafen te vinden (herinner: isomorfisme behoudt de sources!) zodat de niet-sources niet gemeenschappelijk zijn.

Voorbeeld 6.19. *Beschouw de drie grafen G, H en I in Figuur 6.4. De sources zijn de knopen die gelabeld zijn met een nummer, hetzelfde nummer betekent dezelfde knoop uit de knopenverzameling $\mathcal{K}$. Wanneer we G en H mergen, dan zijn de gemeenschappelijke sources* 2 *en* 4*. Het resultaat van deze merge-operatie is te zien in Figuur 6.5a. Wanneer we op de resulterende graaf J* $(\mathrm{lc}\,2)$ *toepassen, verkrijgen we de graaf in Figuur 6.5b. Indien we deze mergen met I is enkel de source* 4 *gemeenschappelijk, omdat we source* 2 *"anoniem" gemaakt hebben. De toepassing van de merge-operatie is te zien in Figuur 6.5c. In de verkregen graaf zien we dat knoop* 2 *opnieuw aanwezig is, maar deze keer met een andere functie. Indien we de local-operatie niet hadden toegepast, dan had dit ons de graaf in Figuur 6.5d opgeleverd. De boog tussen* 2 *en* 4 *bestond reeds, indien dit niet het geval was zou deze toegevoegd zijn.*

Om een maat te hebben voor het aantal sources dat gebruikt wordt in een expressie introduceren we het begrip expressionwidth:

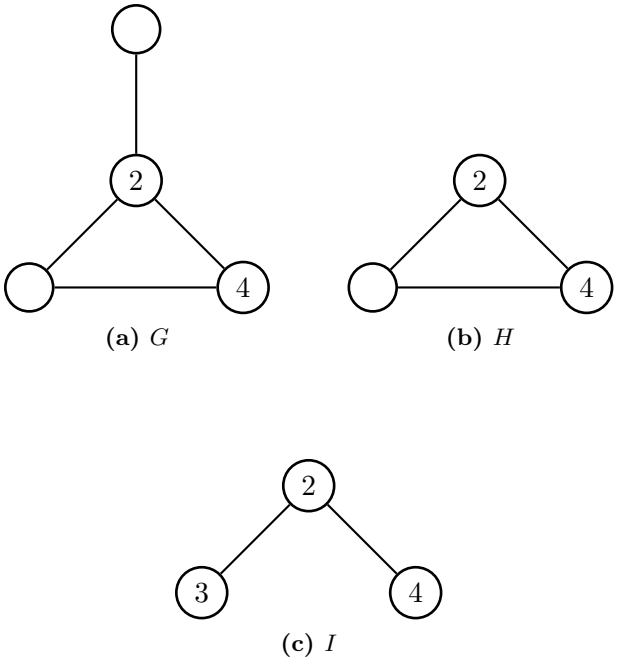
Definitie 6.20 (Expressionwidth)**.** De *breedte* van een graafexpressie e is het aantal *verschillende* sources dat gebruikt wordt in e, min 1. De *expressionwidth* van een graaf G is de minimale breedte van een expressie die G voorstelt.

Voorbeeld 6.21. *Beschouw opnieuw de graaf in Figuur 6.3, een andere expressie dan deze gegeven in voorbeeld 6.18 is*

$$(\mathrm{lc}\,1)(\mathrm{lc}\,2)(\mathrm{lc}\,3)(a(2,3) \mid a(2,4) \mid a(3,1) \mid (\mathrm{lc}\,1)a(1,2)).$$

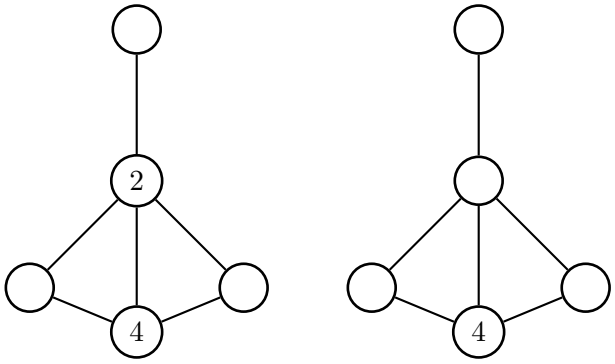
Deze maakt slechts gebruik van drie sources om een graaf die bestaat uit vier knopen te bekomen.

Nu bepaald is wat graafexpressies zijn geven we aan hoe we een reguliere boomgrammatica kunnen construeren die enkel termen genereert



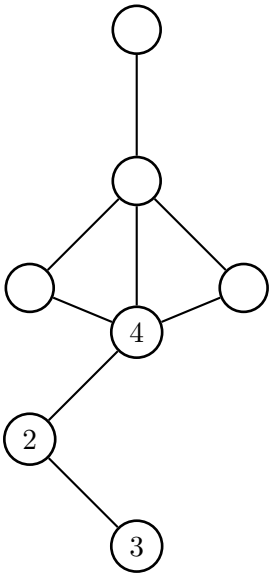
(a) G **(b)** H **(c)** I

Figuur 6.4: Enkele grafen.

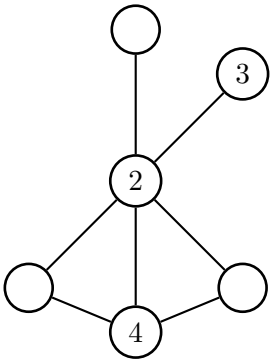


(a) $J = G \mid H$

(b) $K = (\mathrm{lc}\, 2)J$



(c) $L = I \mid K$



(d) $M = I \mid J$

Figuur 6.5: Resultaat van de toepassing van de operatoren op verschillende grafen.

die overeenkomen met graafexpressies. Beschouw het ranked alfabet $\Sigma = \Sigma_0 \cup \Sigma_1 \cup \Sigma_2$, waarbij:

$$\Sigma_0 = \{\text{“}a(x,y)\text{”} \mid a \in \mathcal{E}, x, y \in \mathcal{K}_0\} \tag{6.4}$$
$$\Sigma_1 = \{\text{“}(\mathrm{lc}\, x)\text{”} \mid x \in \mathcal{K}_0\} \tag{6.5}$$
$$\Sigma_2 = \{\text{“}|\text{”}\} \tag{6.6}$$

Samen met een verzameling nonterminals geeft dit ons de mogelijkheid om reguliere boomgrammatica te construeren die graafexpressies produceert. Dit soort grammatica's noemen we *graafgrammatica's*. Merk op dat het aantal sources dat gebruikt kan worden in een graafgrammatica beperkt is. De verzamelingen hierboven beschreven zijn immers allemaal eindig. Dit heeft als direct gevolg dat de graafexpressies die we uit een graafgrammatica kunnen genereren ook een beperkt aantal sources gebruiken. De *graaftaal* die door een graafgrammatica G_r gegenereerd wordt zijn de evaluaties van alle graafexpressies die we uit het startsymbool van de grammatica kunnen afleiden. We noteren deze verzameling als $GL(G_r)$.

Deze methode geeft ons een manier om een bepaalde klasse grafen te beschrijven aan de hand van een contextvrije grammatica.

Voorbeeld 6.22. *Een voorbeeld van een grammatica die alle simpele paden genereert:*

$$C \to (\mathrm{lc}\, x, y)(P)$$
$$P \to (\mathrm{lc}\, z)(Z \mid a(z,y))$$
$$Z \to (\mathrm{lc}\, y)(P \mid a(y,z))$$
$$P \to a(x,y)$$
$$Z \to a(x,z)$$

De notatie $(\mathrm{lc}\, x, y)(P)$ *is een verkorte versie van* $(\mathrm{lc}\, x)((\mathrm{lc}\, y)(P))$. *En voor de mergeoperator hebben we gebruik gemaakt van de infix-notatie. Merk op dat* $(\mathrm{lc}\, x)$ *een functiesymbool op zich is.*

Enkele expressies die gegenereerd worden door deze grammatica zijn:

$$(\mathrm{lc}\, x, y)(a(x,y)), \textit{en}$$

$$(\mathrm{lc}\, x, y)((\mathrm{lc}\, z)(a(x,z) \mid a(z,y))).$$

6.4 Samenvatting

Vaak willen we enkel grafen genereren die aan een zekere eigenschap voldoen. Indien zo een eigenschap *augmentation hereditary* is, dan kunnen we het orderly algoritme gebruiken om te vermijden dat we veel nutteloze objecten genereren.

We introduceerden vervolgens een soort grammatica's die ons interessant lijken om klassen van grafen te beschrijven. De eigenschap van de grafen in de betreffene klassen is dat ze gegenereerd worden door de bijhoordende grammatica. We hebben gezien hoe we een contextvrije grammatica kunnen bekijken als eentje die termen (bomen) genereert. Vervolgens hebben we aangegeven hoe het mogelijk is om deze gegenereerde termen te bekijken als grafen, door ze op een bepaalde wijze te evalueren.

We merken ten slotte op dat de in dit hoofdstuk gebruikte formuleringen gebaseerd zijn op de graafgrammatica van Courcelle [12]. In de "klassieke" graafgrammatica literatuur wordt echter vaak gewerkt met grammatica's die rechtstreeks met grafen werken in plaats van met termen (zie bijvoorbeeld het "Handbook of Graph Grammars and Computing by Graph Transformations" [29]).

Hoofdstuk 7

Graafexpressies en partiële *k-trees*

In het vorige hoofdstuk werd een manier beschreven om de klassen van grafen te beschrijven aan de hand van zogenaamde graafgrammatica's. De grafen die we willen genereren zijn degenen die voldoen aan de eigenschap "wordt gegenereerd door G_r", voor een zekere graafgrammatica G_r. De bedoeling is dat we trachten een algoritme te construeren dat dit probleem op een efficiënte manier oplost.

Een eerste idee is, zoals reeds aangehaald in het vorige hoofdstuk, om alle grafen te construeren met behulp van een (isomorf-vrij) enumeratie-algoritme en vervolgens voor elk van deze te testen of de graaf gegenereerd wordt door de grammatica. Een algoritme hiervoor is gegeven in Algoritme 16. Zoals reeds in opmerking 6.9 op p. 142 aangehaald, is het echter niet eenvoudig om te controleren of een object aanvaard wordt door een grammatica. In dit geval ligt het niet voor de hand hoe we een graaf terug omzetten naar een graafexpressie, waarvan het lidmaatschap wel relatief eenvoudig te bepalen is.

Dit probleem vertoont overeenkomsten met het "parsing probleem" voor contextvrije grammatica's: Gegeven een string s, kan deze string gegenereerd worden door een zekere grammatica G_c? In het geval van graafgrammatica's: Gegeven een graaf G, kan deze graaf gegenereerd worden door een graafgrammatica G_r? Schürr [27] geeft een lijst van verwante literatuur in verband met het parsing probleem voor graafgrammatica's.

Merk op dat, zoals reeds vermeld, er in het algemeen veel overbodige grafen gegenereerd worden. We kunnen dus best op zoek gaan naar een meer efficiënte aanpak.

Een andere aanpak die we kunnen volgen is proberen een orderly algoritme te construeren, gebruik makend van een *augmentation hereditary* eigenschap, zoals beschreven in het begin van vorig hoofdstuk. Hierbij moeten we in essentie zorgen dat we alle graafexpressies kunnen genereren op

Algoritme 16: Naïve isomorfvrije enumeratie

```
   Input: graafgrammatica G_r, enumeratie-algoritme A
   Output: Alle grafen die gegenereerd worden door G_r
1  while A heeft nog niet-geproduceerde grafen do
2  |  G = volgende graaf geproduceerd door A;
3  |  if G ∈ GL(G_r) then
4  |  |  output G;
5  |  end
6  end
```

een isomorf-vrije manier.

Een eerste eis die we hebben is dat elke expressie slechts één keer gegenereerd wordt. Dit kan opgelost worden door gebruik te maken van een deterministisch gemaakte boomautomaat. De volgende stap is om ervoor te zorgen dat, wanneer er twee verschillende expressies dezelfde graaf voorstellen, er slechts eentje geproduceerd wordt. Dit bekomen we door de notie van isomorfie te definiëren op expressies: gegeven twee graafexpressies e_1 en e_2, dan geldt

$$e_1 \cong e_2 \Leftrightarrow \|e_1\| \cong \|e_2\|.$$

Indien we nu van elke isomorfieklasse van expressies slechts één exemplaar genereren, dan genereren we ook van elke graafklasse slechts één exemplaar. De expressie die uit een klasse gekozen wordt noemen we *canonische expressie* van die klasse.

Opmerking 7.1. De expressies stellen eigenlijk een hele klasse isomorfe grafen voor[1]. We kunnen dus eigenlijk beter schrijven dat

$$e_1 \cong e_2 \Leftrightarrow \|e_1\| = \|e_2\|.$$

Herinner dat de gepresenteerde enumeratie-algoritmen gebruik maakten van een canoniciteits-functie, en dat de notie van isomorfisme ook erg belangrijk is.

In het volgende hoofdstuk tonen we aan dat het controleren of twee graafexpressies isomorf zijn in polynomiale tijd gedaan kan worden. Om dit te kunnen doen dienen we een gerelateerd begrip te beschrijven, genaamd partiële *k-trees*. Het overige deel van dit hoofdstuk beschrijft de opbouw en eigenschappen van partiële *k-trees*.

7.1 Treewidth

In deze sectie introduceren we het begrip treewidth aan de hand van een treedecompositie van een graaf. We beschouwen onze oorspronkelijke definitie van grafen (geen sources, ongericht).

[1]Herinner dat een expressie eigenlijk een abstracte graaf voorstelt.

Definitie 7.2 (tree-decompositie)**.** Een tree-decompositie van een graaf G is een paar $TD = (T, (X_t)_{t \in T})$, waarvoor geldt dat:

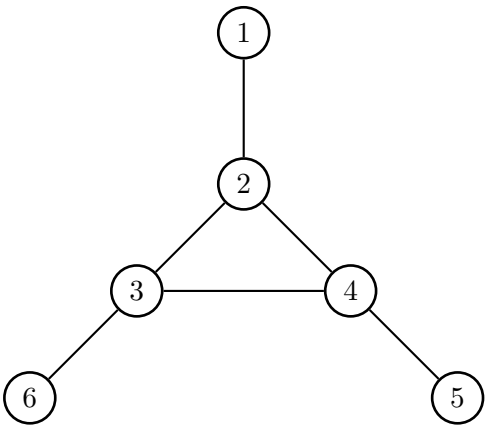
1. T is een boom,
2. $\forall t \in V(T) : X_t \subseteq V(G)$,
3. $\cup_{t \in V(T) X_t = V(G)}$,
4. $\forall e = \{v, w\} \in E(G) : \exists t \in V(T) : \{v, w\} \subseteq X_t$,
5. $\forall v \in V(G) : \{t \mid v \in X_t\}$ vormt een subboom van T, of equivalent:
   $\forall t_1, t_2, t_3 \in V(T)$: indien t_2 op het (unieke) pad van t_1 naar $t3 \Rightarrow X_{t_1} \cup X_{t_3} \subseteq X_{t_2}$. Verder noemen we de nodes van T de *bags* van de tree-decompositie.

De bovenstaande definitie zegt in essentie het volgende: een tree-decompositie van een graaf is een boom, waarbij de nodes van deze boom gelinkt worden aan bepaalde knopen uit de graaf. Verder geldt er dat:

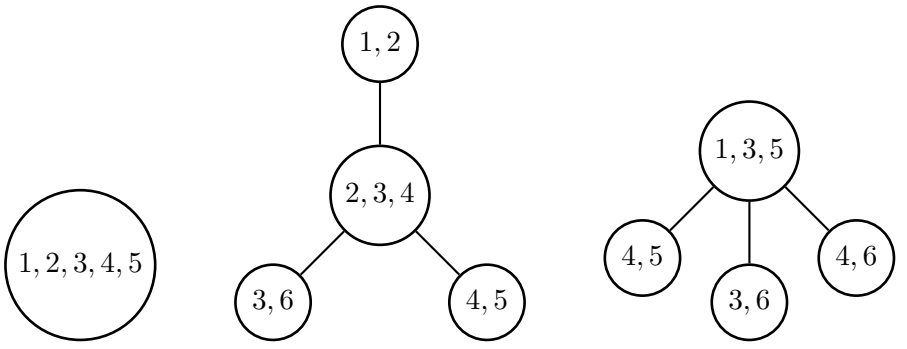
- Elke knoop uit de graaf moet minstens in één *bag* (node van de boom) voorkomen (deze eis wordt gegeven door de derde voorwaarde in de definitie).
- Elk paar knopen dat verbonden is in de graaf komt minstens in één zak *samen* voor (voorwaarde 4).
- Indien we twee nodes uit de boom beschouwen, dan bevat elke node op het *unieke* pad tussen deze twee minstens de gemeenschappelijke knopen (voorwaarde 5). We kunnen dit ook zien als de eis dat alle nodes die eenzelfde knoop bevatten geconnecteerd moeten zijn (er bestaat een pad tussen elk paar knopen).

In hetgeen volgt duiden we de koppeling tussen de nodes van de boom en de knopen van de graaf aan door te zeggen dat de nodes knopen "bevatten". Merk op dat de boom in essentie ook een graaf is (en we dus onze graafoperatoren erop kunnen gebruiken, zoals $V(T)$), maar dat we ter verduidelijking de naam *nodes* of *bags* gebruiken in plaats van "knopen". Deze naamgevingen gebruiken we in de rest van dit hoofdstuk, tenzij anders vermeld.

Voorbeeld 7.3. *In Figuur 7.1 zien we een graaf die bestaat uit zes knopen en zes bogen. Twee mogelijke tree-decomposities zijn te zien in Figuur 7.2a en 7.2b. De eerste tree-decompositie noemen we de* triviale tree-decompositie*, deze bestaat uit slechts één* bag *waarin alle knopen vervat zijn. Het is eenvoudig om te controleren dat dit volgens definitie 7.2 een geldige tree-decompositie is. De tweede tree-decompositie is gegeven in Figuur 7.2b, deze heeft duidelijk een boomstructuur en bevat alle knopen die in de graaf voorkomen. Wanneer we de bogenverzameling beschouwen, is het duidelijk dat*



Figuur 7.1: Een graaf op 6 knopen.



(a) de triviale **(b)** een niet-triviale **(c)** een ongeldige tree-decompositie

Figuur 7.2: Enkele tree-decomposities voor de graaf in Figuur 7.1.

elk paartje uit deze verzameling ergens samen in een node van de boom voorkomt. Hiernaast zien we ook dadelijk dat de geconnecteerdheidseis voor nodes met eenzelfde knoop geldt. Tot slot zitten enkel geldige knopen uit de graaf in de nodes van de boom. We mogen dus besluiten dat ook dit een tree-decompositie voor de graaf uit Figuur 7.1 is.

De boom die in Figuur 7.2c wordt gegeven is geen tree-decompositie voor de beschouwde graaf, en wel hierom:

- *knoop 1 komt in geen enkele node voor. (voorwaarde 3)*
- *de boog* $\{3, 4\}$ *bestaat, maar er is geen node waarin deze knopen samen voorkomen. (voorwaarde 4)*
- *de nodes die knoop 4 bevatten zijn niet onderling geconnecteerd. (voorwaarde 5)*

Merk op dat de aanwezigheid van de node die $\{4, 6\}$ *bevat in deze laatste boom overbodig is, maar de definitie verbiedt de aanwezigheid van dit soort nodes niet. We mogen dus* bags *maken met redundante informatie in.*

Bovenstaand voorbeeld geeft al aan dat er verschillende tree-decomposities voor een graaf kunnen bestaan. Het geeft ook aan dat er altijd minstens één tree-decompositie bestaat, namelijk de *triviale tree-decompositie*. Deze bestaat uit slechts één *bag* waarin alle knopen van de graaf zitten.

Nu we weten wat het begrip tree-decompositie inhoudt, zijn we klaar om aan de hand hiervan het concept *treewidth* in te voeren.

Definitie 7.4 (Treewidth)**.** De *breedte* of *width* van een tree-decompositie $TD = (T, (X_t)_{t \in T})$ wordt gegeven door:

$$w(TD) = \max_{t \in V(T)} \|X_t\| - 1.$$

De breedte wordt met andere woorden gegeven door het aantal knopen in de grootste *bag* min 1. De *treewidth* van een graaf G wordt als volgt gedefinieerd:

$$tw(G) = \min\{w(TD) \mid TD \text{ is een tree-decompositievan } G\}.$$

De *treewidth* van een graaf is dus het minimum van de breedtes van zijn tree-decomposities.

Stelling 7.5. *De* treewidth *van een graaf is 1 als en slechts als de graaf een bos*[2] *is:*

$$tw(G) = 1 \Leftrightarrow G \text{ is een boom.}$$

[2]Een bos is een boom die uit verschillende niet-geconnecteerde componenten mag bestaan.

Bewijs. We geven een idee van het bewijs voor: G is een boom (!) $\Rightarrow$ $tw(G) = 1$ met behulp van inductie.

basis $V(G) = 1$ of $V(G) = 2$
Indien G één of twee knopen bevat, kunnen we een tree-decompositie maken met één node. In deze node zitten in dat geval de twee knopen en bijgevolg is de *treewidth* 1.

inductie Neem aan dat de stelling geldt voor bomen met n knopen. Beschouw nu een boom G die bestaat uit $n + 1$ knopen. Indien we een blad uit de boom verwijderen, dan bekomen we opnieuw een boom met n knopen, noem deze boom G'. We weten dat $tw(G') = 1$ wegens de inductiehypothese, en bijgevolg weten we ook dat er een tree-decompositie bestaat waarbij er maximaal twee knopen in elke node zitten, noem deze TD'.

Noem v de knoop die we uit G hebben moeten verwijderen om G' te bekomen en w de ouder van v in G. Zoek in TD' een node waarin w voorkomt en voeg hieraan een nieuwe kindnode toe met daarin $\{v, w\}$ en noem deze tree-decompositie TD. De boog tussen v en w zit nu verwerkt in TD en we hebben op deze manier een tree-decompositie geconstrueerd voor de graaf G. Het is duidelijk dat de breedte van deze tree-decompositie gelijk is aan 1. Het is eveneens duidelijk dat er geen tree-decompositie bestaat met breedte 0, omdat de boog tussen v en w bestaat.

Om aan te tonen dat $tw(G) = 1 \Rightarrow G$ is een bos kan er gebruik gemaakt worden van zogenaamde *verboden minoren*. Een volledige uitleg, en een redenering waarom het voldoende is om de stelling voor bomen aan te tonen wordt gegeven door Bodlaender [7]. □

Stelling 7.6. *De treewidth van een subgraaf $G' \subseteq G$ is maximaal $tw(G)$.*

Bewijs. Beschouw een tree-decompositie TD voor G met $w(TD) = tw(G)$, dan is het mogelijk om hieruit eem tree-decompositie TD' te bouwen voor elke graaf G' met $G' \subseteq G$, waarbij $w(TD') \leq w(TD)$. De manier waarop dit gebeurt is door alle knopen die aanwezig zijn in G, maar niet in G' te verwijderen uit de nodes van TD. Het is mogelijk dat door deze verwijdering lege nodes ontstaan, maar dit wordt niet verboden door de definitie van een tree-decompositie. Merk op dat een lege node in de TD aangeeft dat de onderdelen van de tree-decompositie die door deze verbonden worden niets met elkaar te maken hebben in de graaf (ze zijn niet verbonden!). Het is eenvoudig na te gaan dat TD' een geldige tree-decompositie voor G' is.

In het ergste geval hebben we niets uit TD verwijderd en is $w(TD') = w(TD)$, en merk ook op dat de bekomen tree-decompositie TD' voor G' niet noodzakelijk een minimale[3] is. Er geldt bijgevolg dat $tw(G') \leq tw(G)$. □

[3]Eentje met minimale breedte.

Eigenschap 7.7. *De* treewidth *van een graaf G is kleiner dan het aantal knopen van G:*

$$tw(G) < |V(G)| = o(G).$$

Bewijs. We nemen de triviale tree-decompositie voor G, waarbij we één bag bekomen met $n = |V(G)|$ knopen erin. De breedte van deze tree-decompositie is bijgevolg $n - 1$. □

7.2 Partiële k-trees

We introduceren nu een bepaalde klasse van grafen die we in verband brengen met de in de vorige sectie geïntroduceerde notie van *treewidth*.

Definitie 7.8. Een *k-tree* is een *k-clique* (ook wel een *triviale* k-tree genoemd). Een *k-tree* kan anderzijds ook bekomen worden door een bestaande *k-tree* uit te breiden: neem een *k-clique* uit de *k-tree* en verbind een nieuwe knoop met al de knopen uit deze *clique*.

Bovenstaande definitie geeft aan dat *k-trees* bestaan uit *$k+1$-cliques* (of een *k-clique* zijn) en dat ze nooit een *$k+2$-clique* of groter bevatten.

Voorbeeld 7.9. *In Figuur 7.3 zien we vijf* 3*-trees. De eerste graaf is een eenvoudige* 3-clique*, en indien we een nieuwe knoop verbinden met elke knoop uit deze* clique*, dan bekomen we de graaf uit Figuur 7.3b. Indien we vervolgens een* k-clique *nemen uit de nieuw bekomen graaf, zijn we in staat om weer een nieuwe graaf te construeren. Merk op dat we niet altijd isomorfe grafen bekomen door een graaf op de verschillende mogelijke manieren uit te breiden (indien we hetzelfde aantal knopen beschouwen): Figuur 7.3e is duidelijk niet isomorf met 7.3d, dit is duidelijk te zien aan het feit dat er niet evenveel knopen van elke graad zijn.*

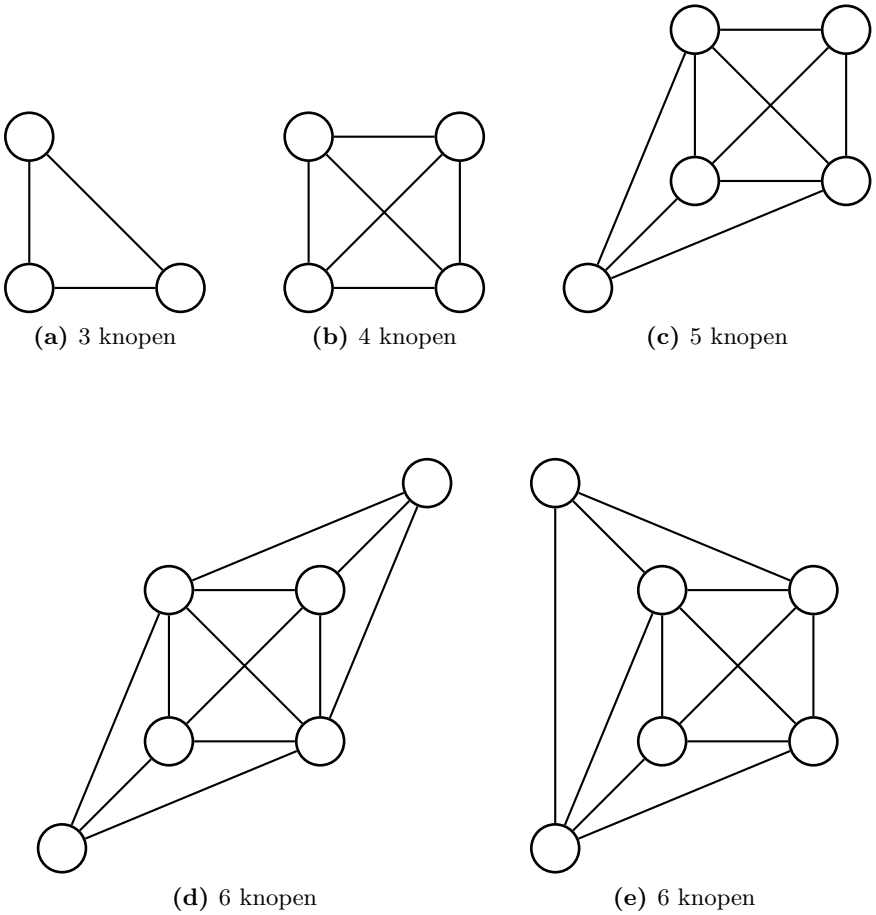
Definitie 7.10. Een partiële *k-tree* is een subgraaf van een *k-tree*.

In hetgeen volgt zullen we een verband leggen tussen deze notie van partiële *k-trees* en het begrip *treewidth*. Om dit mogelijk te maken tonen we eerst enkele lemma's aan.

Lemma 7.11. *Elke tree-decompositie van de complete graaf op k knopen K_k bevat een* bag *waarin alle knopen van de graaf voorkomen.*

Bewijs. Bewijs door middel van contrapositie. Beschouw een tree-decompositie $TD = (T, (X_t)_{t \in T})$ voor K_k. Veronderstel dat er geen t bestaat zodat $t \in T$ en $V(K_k) = X_t$. Met andere woorden, we veronderstellen dat het lemma niet geldt en er dus geen dergelijke *bag* bestaat.

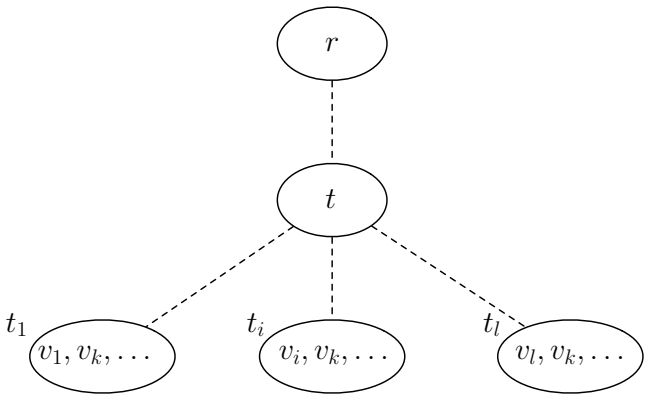
Neem als wortel van T een node r, zodat $|X_r|$ maximale grootte heeft (een maximale *bag*) en neem $X_r = v_1, \ldots, v_l$. We weten wegens de veronderstelling dat $l < k$.



(a) 3 knopen **(b)** 4 knopen **(c)** 5 knopen

(d) 6 knopen **(e)** 6 knopen

Figuur 7.3: Enkele partiële 3-*trees*.



Figuur 7.4: Treedecompositie van een *k-clique*.

Definieer vervolgens voor elke $v_i \in X_r$:

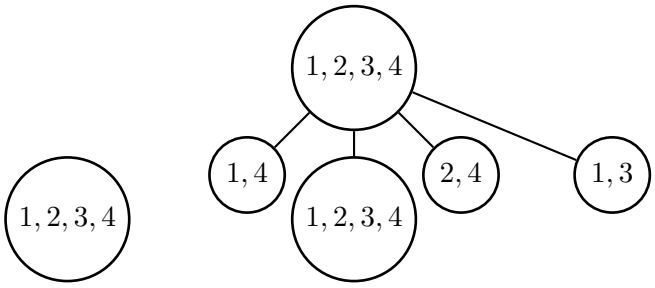
$$t_i \in V(T) : \{v_i, v_k\} \subseteq X_{t_i}, \text{ waarbij } v_k \in V(K_k) \setminus X_r (1 \leq i \leq l).$$

Met andere woorden, we associëren met elke knoop uit de wortel een node waarin de betreffende knoop voorkomt en hiernaast ook nog minstens één knoop die niet in de wortel voorkomt. Merk op dat deze knoop v_k voor elke waarde van i hetzelfde is en dat sommige t_i's gelijk kunnen zijn. Beschouw nu $t_1, \ldots, t_l$ uit de tree-decompositie en neem de laagste gemeenschappelijke voorouder van deze nodes (degene die het meest vanonder zit in de boom). We noemen deze voorouder t. Een schematisch overzicht wordt gegeven in Figuur 7.4.

We weten nu dat X_t de knopen $v_1, \ldots, v_l$ bevat, omdat t op het pad van elke t_i naar r ligt. Omdat elke t_i de knoop v_k bevat, weten we dat ook t deze knoop bevat (laagste gemeenschappelijke voorouder). Dit geldt allemaal wegens de connectiviteitseis van een tree-decompositie. We hebben nu $|X_t| > |X_r|$, omdat X_t alle knopen uit X_r bevat, evenals de knoop v_k. Dit spreekt de maximaliteit van X_r tegen. Onze aanname dat er geen node bestaat die alle knopen bevat is bijgevolg incorrect, wat maakt dat het lemma bewezen is. □

Voorbeeld 7.12. *Alle mogelijke treedecomposities van de graaf in Figuur 7.3b (de* 4-clique*) bevatten ten minste één node die alle vier de knopen bevat. Hiernaast kunnen natuurlijk nog andere nodes aanwezig zijn, maar deze zijn niet noodzakelijk. In Figuur 7.5 zijn twee tree-decompositions van de betreffende graaf gegeven.*

Lemma 7.13. *Beschouw een graaf G met $tw(G) \leq k$, waarbij $k < |V(G)|$, dan bestaat er een tree-decompositie waarin alle* bags *grootte $k+1$ hebben.*



(a) De triviale tree-decompositie.

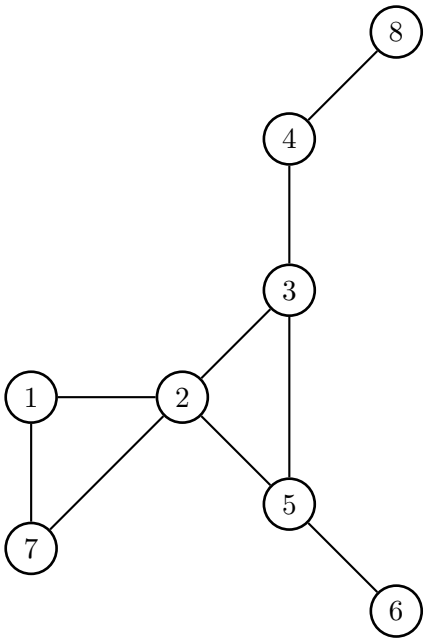
(b) Een niet-triviale tree-decompositie

Figuur 7.5: Enkele tree-decomposities voor de graaf in Figuur 7.3b.

Bewijs. Neem een tree-decompositie TD van G met een breedte $\leq k$. Dit is mogelijk omdat de *treewidth* $\leq k$ is. Neem vervolgens een maximale *bag* en zoek een knoop v die niet in deze *bag* voorkomt. Voeg nu deze knoop aan alle *bags* toe. Deze techniek herhalen we tot onze tree-decompositie een breedte heeft van k en er dus ergens een *bag* is die $k+1$ knopen bevat. We nemen vervolgens voor de root van de tree-decompositie één van de maximale bags. We gaan nu recursief op de kinderen van de root volgende procedure toepassen: voor elk kind dat minder als $k+1$ knopen bevat, vullen we de knopen aan met nieuwe die in zijn ouder voorkomen tot er $k+1$ knopen in het kind zitten. Het is duidelijk dat we op deze manier een tree-decompositie verkrijgen waarvan alle *bags* $k+1$ knopen bevatten. □

Opmerking 7.14. Indien $k \geq |V(G)|$, dan is het niet mogelijk om *bags* van grootte $k+1$ te construeren, omdat we niet beschikken over voldoende knopen.

Voorbeeld 7.15. *De graaf in Figuur 7.6 heeft acht knopen en we nemen aan dat de* treewidth *kleiner of gelijk is aan* 2*. We kunnen dus zeker een tree-decompositie opstellen waarbij alle* bags *vijf knopen bevatten, dus een tree-decompositie met breedte 4. We weten dat dit mogelijk is, omdat k in dit geval 4 is, hetgeen minder is dan het aantal knopen, en ook de* treewidth *is kleiner dan 4. Een tree-decompositie is gegeven in Figuur 7.7a. We gaan deze nu omzetten naar eentje waarbij elke* bag *vijf knopen bevat. Dit doen we door eerst de* bag *te nemen waarin* $\{2, 3, 5\}$ *zit en de hele tree-decompositie uit te breiden met knopen die hierin niet voorkomen. We doen dit achtereenvolgens door de knopen* 1 *en* 4 *te beschouwen en bekomen zo de graaf in Figuur 7.7b.*



Figuur 7.6: Een graaf op 8 knopen.

Hierna voeren we de recursieve procedure uit en bekomen we een tree-decompositie van de gewenste vorm (Figuur 7.7d). Merk op dat twee bags *in deze tree-decompositie identiek zijn, we kunnen ze* mergen *(samenvoegen) om dit te vermijden.*

Het bovenstaande lemma laat duidelijk toe om een tree-decompositie te construeren waarbij alle *bags* dezelfde grootte hebben.

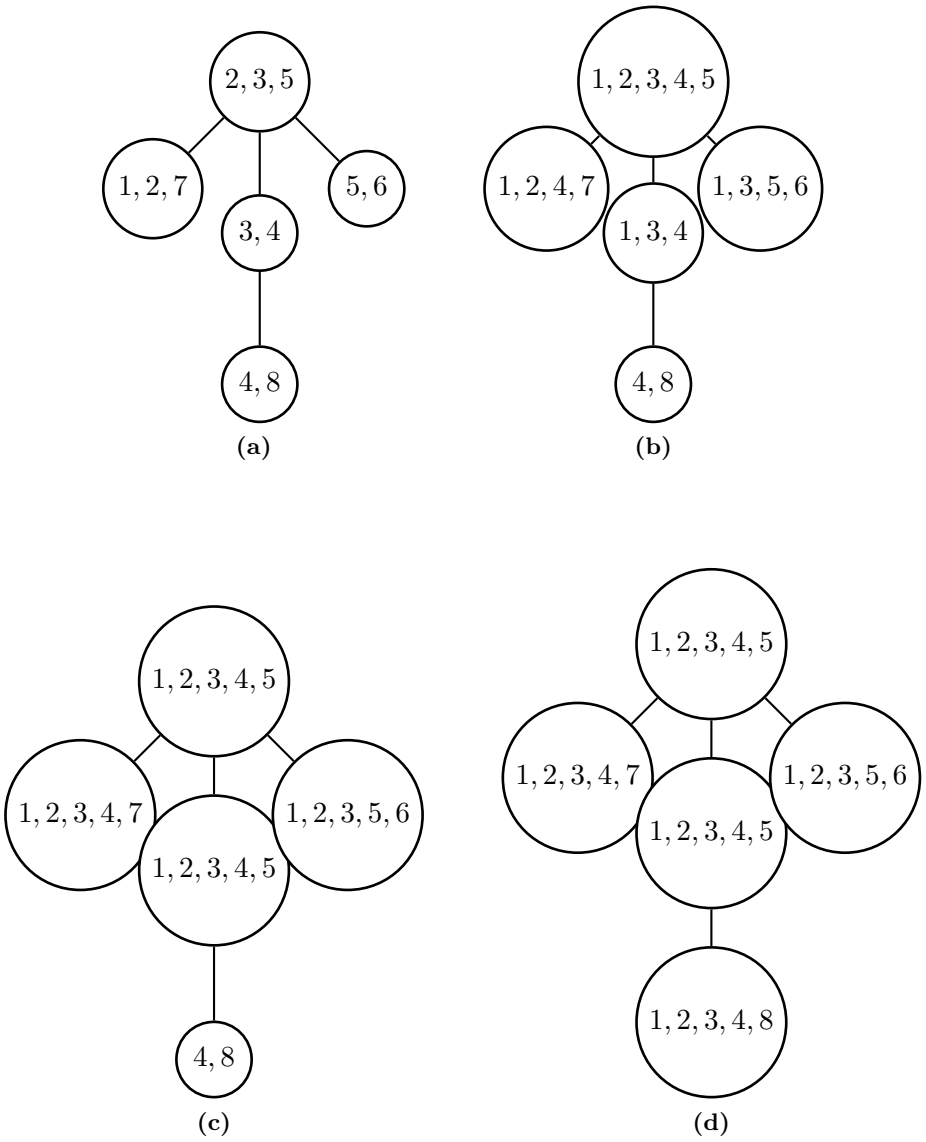
Lemma 7.16. *Beschouw een graaf G met $tw(G) = k$, dan bestaat er een tree-decompositie $TD = (T, (X_t)_{t \in T})$ van breedte k zodat voor elk paar nodes $t, t' \in V(T)$, waarbij t de buur is van t', geldt dat*

$$|X_t \cap X_{t'}| = |X_t| - 1 = |X_{t'}| - 1.$$

Met andere woorden, ze verschillen in precies één knoop en zijn even groot.

Bewijs. Wegens lemma 7.13 weten we dat er een tree-decompositie TD bestaat waarvan elke *bag* $k+1$ knopen bevat. We zorgen dat elke *bag* van deze tree-decompositie uniek is door gelijkaardige nodes te mergen met elkaar of weg te laten. We nemen nu een willekeurige node als root van TD en passen vervolgens de nodes van de boom aan zodat voor elke node t met kinderen t_i $(1 \leq i \leq l)$ geldt dat

$$|X_t \setminus X_{t_i}| = 1.$$



Figuur 7.7: Een run van het algoritme om overal 5 knopen in de bags te bekomen.

Dit doen we als volgt: Voor elke t_i: vervang t_i door een pad $t_{i,1}, \ldots, t_{i,c_i}$, waarbij $c_i = |X_t \setminus X_{t_i}|$. c_i geeft dus aan hoeveel knopen uit X_{t_i} vervangen moeten worden (herinner: $|X_t| = |X_{t_i}|$). Het pad wordt als volgt opgesteld:

- $t_{i,1}$ wordt een kind van t,
- $t_{i,j}$ wordt een kind van $t_{i,j-1}$ $(1 < j \leq c_i)$,
- t_{i,c_i} wordt de ouder van de kinderen van t_i.

We bepalen nu de knopen die in elke $X_{t_{i,j}}$ terecht komen: neem $\{v_1, \ldots, v_{c_i}\} = X_t \setminus X_{t_i}$ en $\{w_1, \ldots, w_{c_i}\} = X_{t_i} \setminus X_t$ (herinner dat beide *bags* evenveel knopen bevatten en bijgevolg ook evenveel niet-gemeenschappelijke knopen). Definieer vervolgens:

- $X_{t_{i,1}} = (X_t \setminus \{v_1\}) \cup w_1$,
- $X_{t_{i,j}} = (X_t \setminus \{v_1, \ldots, v_j\}) \cup w_1, \ldots, w_j = X_{t_{i,j-1}} \setminus \{v_j\} \cup \{w_j\}$, en
- $X_{t_{i,c_i}} = (X_t \setminus \{v_1, \ldots, v_{c_i}\}) \cup w_1, \ldots, w_{c_i} = X_{t_{i,c_i-1}} \setminus \{v_{c_i}\} \cup \{w_{c_i}\}$.

Merk op dat de volgorde van de v_i's en w_i's niet uitmaakt.

Alle *bags* zijn nu duidelijk even groot, wegens de constructie is het duidelijk dat de ouders en kinderen overal in slechts één knoop verschillen, hetgeen het lemma bewijst. □

Definitie 7.17 (genormaliseerde tree-decompositie)**.** Een tree-decompositie van een graaf G met *treewidth* k die in elke *bag* $k+1$ knopen heeft en die voldoet aan de voorwaarde van lemma 7.16 noemen we een *genormaliseerde tree-decompositie* van G. Het proces om een niet-genormaliseerde tree-decompositie om te zetten in een genormaliseerde, noemen we *normalisatie* van de tree-decompositie.

Nu we de nodige lemma's bewezen hebben, gaan we over tot het aantonen van een belangrijke stelling:

Stelling 7.18. *G is een partiële k-*tree *als en slechts als* treewidth$(G) \leq k$.

Bewijs. We weten uit stelling 7.6 dat een tree-decompositie voor G omgevormd kan worden tot een tree-decompositie voor $G' \subseteq G$, met maximaal dezelfde breedte. Om deze reden volstaat het om de eigenschap te bewijzen voor "gewone" *k-trees*. We doen dit door middel van inductie.

⇒

basis $n = k$ of $n = k+1$ Indien $n = k$ is G een *k-clique* en wegens lemma 7.11 bevat elke tree-decompositie van deze graaf een *bag* waarin alle knopen vervat zijn. Omdat er voor elke graaf minstens één tree-decompositie bestaat (de triviale), mogen we besluiten dat de *treewidth* van de graaf gelijk is aan $k - 1 \leq k$.

Het geval $n = k + 1$ is analoog, we bekomen hier een *treewidth* van $k \leq k$.

inductie Veronderstel dat de stelling geldt voor n knopen. Beschouw nu een *k-tree* G' op n knopen, die uitgebreid kan worden tot G door een nieuwe knoop v met een *k-clique* te verbinden. We weten dat G' n knopen bevat en de inductiehypothese zegt ons dat $tw(G') \leq k$, er bestaat dus een tree-decompositie TD' met breedte k of minder. Lemma 7.11 zegt ons dat er een node bestaat voor elke *k-clique* in G, die minstens alle knopen van deze *clique* bevat. Voor de *clique* waarmee we v gaan verbinden geldt dit dus ook. We breiden vervolgens TD' uit tot TD door een nieuwe node toe te voegen, waarin we alle buren van v stoppen, samen met v zelf. Deze node hangen we aan een willekeure node die de buren van v bevat. Merk op dat de grootte van de *bag* $k+1$ is en dat de bekomen *treewidth* bijgevolg nog steeds gelijk is aan k. Het is duidelijk dat TD een tree-decompositie is van G, alle bogen waarmee v verbonden is worden "bedekt" door de toegevoegde node.

Merk op dat het mogelijk is (in geval van een partiële *k-tree*) dat het aantal knopen lager is dan $k+1$, in dat geval is de *treewidth* maximaal $n-1$.

$\Leftarrow$ Neem aan dat G minstens $k+1$ knopen heeft. Beschouw een graaf G met een tree-decompositie $TD = (T, (X_t)_{t \in T})$ van breedte $\leq k$. Wegens lemma 7.11 en 7.13 mogen we aannemen dat:

- elke *bag* van TD $k+1$ knopen bevat, en
- alle ouders in precies één knoop van hun kind verschillen.

De TD is met andere woorden genormaliseerd. Neem voor de root van T een willekeurige node r.

We kunnen nu uit TD een *k-tree* G' construeren waarvoor geldt dat $G \subseteq G'$. Dit gebeurt op de volgende manier:

1. Start met de knopen uit r en verbind deze allen onderling (we vormen een $k+1$-*clique*).
2. Ga nu recursief verder op de kinderen t_i en pas de volgende procedure toe: We weten dan $|X_t \setminus X_{t_i}| = 1$ en $|X_t| = |X_{t_i}|$, dus dat de ouder-*bag* verschilt in precies één knoop van de kind-*bag*. Beschouw nu deze knoop voor elk kind en noem hem v_i. Verbind nu v_i met de andere knopen uit X_{t_i}, dit vormt opnieuw een $k+1$-*clique* en we bekomen opnieuw een *k-tree* (onze constructie voldoet aan de definitie). Indien we deze procedure recursief toepassen op de kinderen bekomen we een *k-tree* G'. Indien we nu kunnen aantonen dat $G \subseteq G'$ weten we dat G een partiële *k-tree* is. Dit is niet zo moeilijk in te zien: voor elke boog in G is er een *bag* die de knopen bevat die door de betreffende boog verbonden worden. We hebben bij onze constructie voor elke *bag* alle knopen verbonden, dus we hebben minstens alle bogen uit

G gebruikt. Ook alle knopen zijn gebruikt omdat TD deze allemaal dient te bevatten. Het is dus duidelijk dat $G \subseteq G'$ en omdat G' een *k-tree* is, is G een partiële *k-tree*.

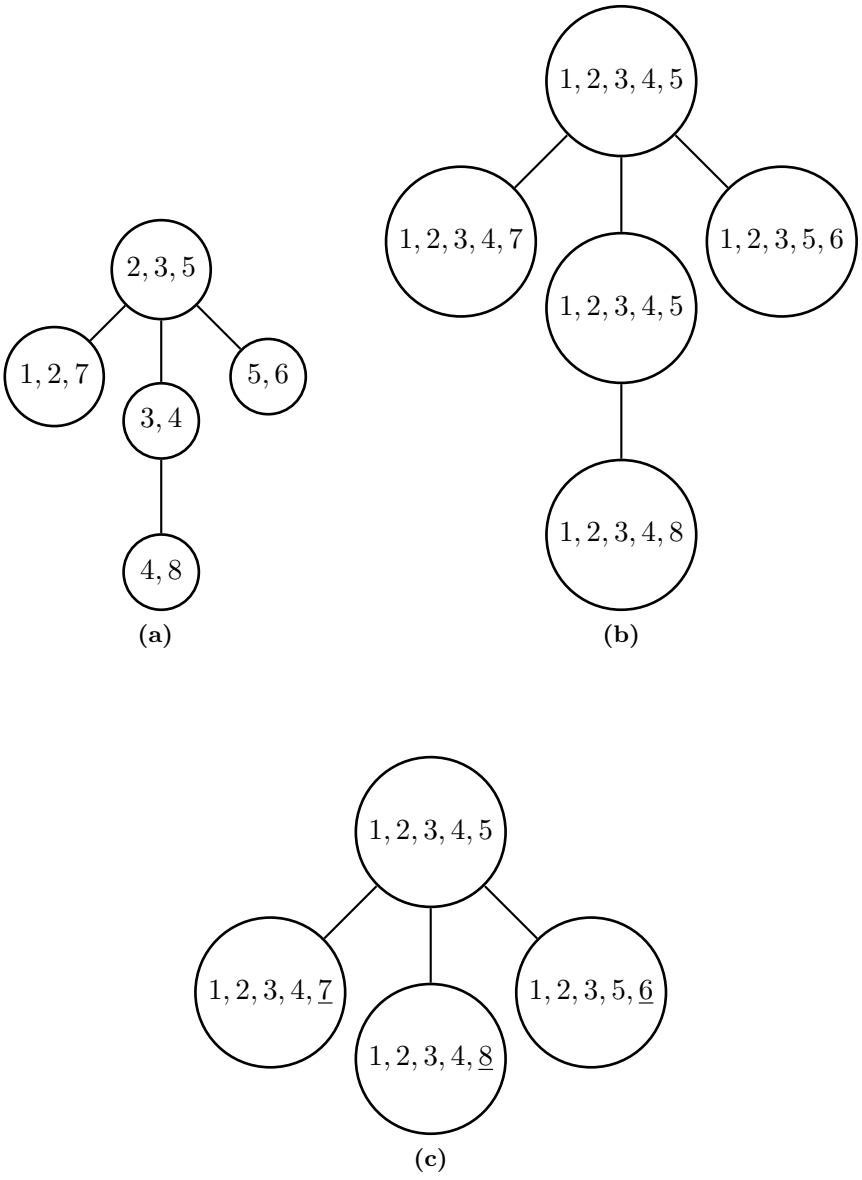
□

Opmerking 7.19. Bovenstaand bewijs geeft ons in essentie ook een procedure om van een tree-decompositie naar een *k-tree* te gaan. Het is eveneens duidelijk dat de constructie van een *k-tree* ons aangeeft hoe we een tree-decompositie kunnen opstellen voor de betreffende graaf: een knoop die enkel aan k knopen vasthangt kan, samen met zijn buren, in één *bag* gestopt worden, waarna hij verwijderd wordt uit de graaf en we dezelfde procedure herhalen. Op deze manier kunnen we bottom-up onze tree-decompositie opstellen. Als we een knoop verwijderen die ergens al in bepaalde *bags* voorkomt, dan wordt de nieuwe *bag* de ouder.

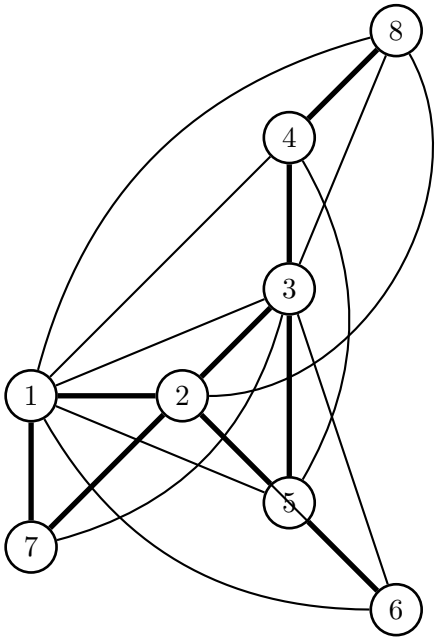
Voorbeeld 7.20. *We gaan de voor de graaf G in Figuur 7.6 een* k-tree *construeren uit een mogelijke tree-decompositie. Een tree-decompositie wordt gegeven in Figuur 7.8a. Merk op dat hier maximaal drie knopen in elke bag zitten, de graaf heeft dus maximaal* treewidth 2. *We hebben bijgevolg te maken met een partiële* 2-tree, *maar ook met een partiële* 3-tree, *partiële* 4-tree, *partiële* 5-tree, ... *Stel dat we de graaf bekijken als partiële* 4-tree *en we een tree-decompositie willen bekomen die genormaliseerd is: dan moeten we zorgen dat elke zak vijf knopen heeft en dat ouders in precies één knoop verschillen van hun kind. Merk op dat we in dit geval geen* bags *moeten verspreiden over meerdere zoals in lemma 7.16 en het enkel noodzakelijk is om dubbele* bags *te verwijderen. We bekomen de tree-decompositie gegeven in Figuur 7.8c, de knopen die verschillen van de ouderbags zijn onderlijnd weergegeven.*

Vervolgens gaan we voor elke bag *al zijn knopen onderling verbinden, het resultaat is weergegeven in Figuur 7.9. De vette bogen in deze figuur geven de aanwezigheid van de oorspronkelijke graaf aan om weer te geven dat deze wel degelijk een subgraaf is.*

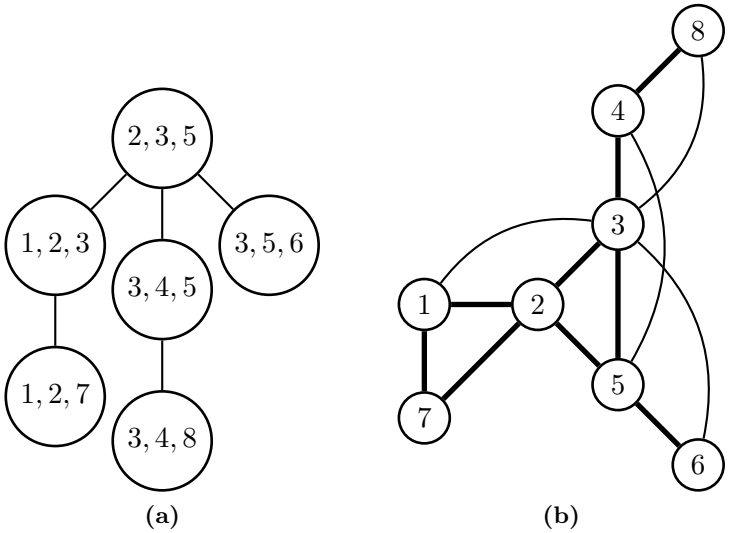
Een andere mogelijkheid is om een 2-tree *op te stellen, zo deze die wordt weergegeven in Figuur 7.10b: er zijn inderdaad geen* 4-cliques *aanwezig! De betreffende* k-tree *is gebaseerd op de tree-decompositie die gegeven wordt in Figuur 7.10a. Merk ten slotte op dat het niet mogelijk is om een* 1-tree *op te stellen, omdat er een* 3-clique *aanwezig is in de graaf (twee zelfs) en deze knopen allen samen in één* bag *zullen terechtkomen, wat een minimale* treewidth *van 2 oplevert.*



Figuur 7.8: Een run van de normalisatieprocedure.



Figuur 7.9: De *k-tree* die overeenkomt met de tree-decompositie in Figuur 7.8c.



Figuur 7.10: Een tree-decompositie van de graaf uit Figuur 7.6 met breedte 2 en bijhorende *k-tree*.

7.3 De *treewidth* berekenen

Het begrip *treewidth* is niet onbelangrijk bij zoeken naar efficiënte algoritmen voor graafproblemen. Volgens Arnborg [4] is het zo dat algoritmen eigenschappen van *k-trees* en partiële *k-trees* kunnen uitbuiten om performanter te worden. Hiervoor is echter vaak een *k-tree*-embedding nodig. Met embedding bedoelen we dat de betreffende *k-tree* een supergraaf moet zijn van een gegeven graaf (dan is deze graaf een partiële *k-tree*).

Dit stelt ons voor twee problemen:

1. Het vinden van zo een *k-tree*-embedding, en
2. het vinden van de kleinst mogelijke k, zodat we de gegeven graaf kunnen embedden in een *k-tree*.

Het laatste probleem is van groot belang, omdat de complexiteit van de betreffende algoritmen vaak exponentieel (of erger) is in functie van k. Arnborg bewijst in zijn artikel dat dit laatste probleem NP-compleet is, door aan te tonen dat we het "MCLA" probleem kunnen uitdrukken in termen van ons probleem (het berekenen van de minimale *treewidth*). "MCLA" staat voor *Minimum Cut Lineair Arrangement* en werkt als volgt: definieer de functie c_π als volgt:

$$c_\pi(G) = \max_{1 \leq i < |V(G)|} |\{(u,v) \in E(G) : \pi(u) \leq i < \pi(v)\}|.$$

De functie bepaalt voor een bepaalde permutatie π hoe de meeste bogen gescheiden kunnen worden. Garey en Johnson [16] hebben reeds aangetoond dat dit probleem NP-compleet is. Voor het bewijs dat we dit probleem kunnen uitdrukken in termen van de minimale *treewidth*, verwijzen we naar het betreffende artikel van Arnborg.

Herinner uit stelling 7.18 dat de noties van partiële *k-tree* en *treewidth* samenvallen, hieruit volgt rechtstreeks:

Stelling 7.21. *Het berekenen van de* treewidth *van een graaf is NP-compleet.*

In datzelfde artikel toont Arnborg ook aan dat, indien we de k als constante bekijken[4] dat we in polynomiale tijd kunnen verifiëren of een graaf een partiële *k-tree* is. We beschrijven in deze sectie het betreffende algoritme en tonen aan dat het werkt, maar we gebruiken de terminologie van Bodlaender [6]. De reden voor het gebruik van de andere terminologie is dat deze van Bodlaender iets duidelijker gedefinieerd is.

Definitie 7.22. Een *separator* S van een graaf $G = (V, E)$ is een deelverzameling van de knopen die ervoor zorgt dat $G[V \setminus S]$ bestaat uit verschillende (niet-lege) *componenten*. Een k-vertex separator is een separator die uit k knopen bestaat.

[4] Dus voor een vaste k.

Eigenschap 7.23. *Alle minimale separators van een* k-tree *hebben grootte* k*.*

We duiden de verzameling van verschillende k-vertex separators van een graaf G aan met $C_1, C_2, \ldots$, en voor elke C_i hebben we een verzameling van l componenten (l is afhankelijk van i) die bekomen worden door de knopen van C_i uit de graaf te verwijderen. Deze laatste lijst duiden we aan met $C_i^1, C_i^2, \ldots, C_i^l$. Verder beschouwen we nog een speciaal soort graaf $G(C_i, C_i^j)$, ook wel genoteerd als $G(i, j)$. Deze graaf wordt als volgt geconstrueerd: beschouw de unie van knopen uit C_i^j en C_i noem deze L. Neem nu de geïnduceerde graaf $G[L]$ (zie hoofdstuk 2), en maak de subset van knopen die gelijk is aan C_i compleet (verbind ze allen onderling). In essentie wordt de betreffende component genomen, geaugmenteerd met de compleet gemaakte separator. We noemen dit ook wel een *geaugmenteerde component.*

Eigenschap 7.24 (k-decomposable)**.** *Een graaf* G *is* k-decomposable *indien* $V(G) < k+1$*, of indien we een seperator* S *voor* G *kunnen vinden, zodat voor de overblijvende componenten* C *geldt dat: de graaf* $G(S, C)$ k-decomposable *is.*

Eigenschap 7.25. *Een graaf die bestaat uit* $k+1$ *knopen of minder is een partiële* k-tree*.*

Bewijs. Dit is eenvoudig in te zien: doordat we weten dat de $k+1$-*clique* een k-*tree* is (en dus in het bijzonder een partiële k-*tree*), weten we dat een graaf met $k+1$ knopen een subgraaf is van deze complete graaf en bijgevolg dus ook een partiële k-*tree*. Indien de graaf uit minder dan $k+1$ knopen bestaat is het duidelijk een subgraaf van eentje met $k+1$ knopen en dus ook van een k-*tree*. □

Het doel van het algoritme is om een *embedding* van de graaf te vinden in een k-*tree*, voor een gegeven k. Herinner: met "embedding" wordt bedoeld dat we naar een supergraaf zoeken. Het algoritme maakt gebruik van *dynamisch programmeren*, een techniek die ons toelaat om kandidaatoplossingen van klein naar groot te beschouwen zonder dezelfde berekeningen meerdere malen uit te voeren. Omdat alle k-*trees* minimale separators van grootte k hebben, gaan we alle separators van klein naar groot beschouwen, en vervolgens kijken of er voldaan is aan de k-*decomposability* eigenschap van partiële k-*trees*(zie Bodlaender [7, Thm. 25]).

We lichten nu de verschillende stappen van het algoritme dat weergegeven wordt in Algoritme 17 één voor één toe, samen met de gemaakte veronderstellingen. Hierna tonen we aandat het algoritme in polynomiale tijd loopt en dat het correct is.

Algoritme 17: Herkennen van partiële *k-trees*

Input: G,n
Output: `YES` of `NO`
`// Initialisatie`
foreach $S \subseteq V(G)$, *waarbij* $|S| = k$ **do**
  if S *is een separator van* G **then**
    Voeg S toe aan de datastructuur als een C_i;
    Voeg alle geaugmenteerde componenten $G(i,j)$ toe;
    Zet voor elke $G(i,j)$ pkt = `UNDEFINED`;
  end
end
Sorteer alle C_i^j's volgens toenemende grootte;
`// Triviale fase`
foreach $G(i,j)$ *met precies* $k+1$ *knopen* **do**
  $pkt(i,j)$ =`YES`;
end
`// Dynamische fase`
foreach $G(i,j)$ *van grootte* h **do**
  foreach $v \in V(G(i,j))$ **do**
    Beschouw alle k-*vertex separators* $C_m \subseteq C_i \cup \{v\}$, waarbij $C_m \neq C_i$;
    Bepaal nu voor al deze C_m'en alle $G(m,l) \subseteq C_i^j \cup C_m$;
    U = de unie hiervan (over alle m'en en l'en) ;
    if $C_i^j \subseteq U$ **then**
      $pkt(i,j)$ = `YES`;
      Spring uit de binnenste loop ;
    end
  end
  if $pkt(i,j)$ = *`UNDEFINED`* **then**
    $pkt(i,j)$ = `NO` ;
  end
  `// Conclusiefase`
  if *er bestaat een* C_m *voor* G *zodat* $pkt(m,l)$ =*`YES`* *voor alle* l **then**
    return *`YES`*;
  end
  if $\forall C_p$ *voor* $G : \exists q : pkt(p,q)$ =*`NO`* **then**
    return *`NO`*;
  end
end

7.3.1 Initialisatiefase

Het eerste wat er in het algoritme gebeurt is het bepalen van de *k-vertex separators*. Dit doen we door alle mogelijke knopenverzamelingen van grootte k te beschouwen, die een deel zijn van de knopenverzameling van de graaf G. Voor elk van deze deelverzamelingen gaan we na of het wel degelijk een separator is van G. De separators worden aangegeven met C_i en voor ieder van hen bekomen we l componenten C_i^j ($1 \leq j \leq l$, waarbij l afhankelijk is van i). Met elke C_i^j wordt er verder nog de geaugmenteerde component $G(i,j)$ geassocieerd, zoals hierboven gedefinieerd. Op deze manier bouwen we een datastructuur op die voor elke separator C_i de lijst van $G(i,j)$'s bijhoudt, samen met een indicatie of deze laatste graaf een partiële *k-tree* is (`YES`) of niet (`NO`), of dat er nog geen informatie over bekend is (`UNDEFINED`). We definiëren de functie $pkt(i,j)$ als degene die deze waarde voor een gegeven $G(i,j)$ teruggeeft. De datastructuur wordt volledig geïnitialiseerd op `UNDEFINED`.

Eens we deze data vergaard hebben, dan sorteren we de $G(i,j)$'s van klein naar groot (volgens het aantal knopen).

7.3.2 Triviale Fase

Zoals eigenschap 7.25 ons zegt, mogen we alle $G(i,j)$'s die $k+1$ knopen bevatten classificeren als partiële *k-tree*. Voor al deze zetten we dus:

$$pkt(i,j) = \texttt{YES}.$$

Merk op dat er geen $G(i,j)$'s bestaan met minder dan $k+1$ knopen, omdat $G(i,j)$ bestaat uit de volgende delen:

- componenten C_i^j, waarvoor geldt dat $o(C_i^j) \geq 1$ (anders is het geen component), en
- de complete *k-vertex separator* C_i met $o(C_i) = k$.

Hieruit volgt dat $o(G(i,j)) \geq k+1$.

7.3.3 Dynamische Fase

We beschouwen nu alle $G(i,j)$'s in de volgorde zoals bepaald in de initialisatiefase, buiten deze die $k+1$ knopen bevatten (daarvoor hebben we al besloten of ze een partiële *k-tree* zijn). We gaan nu voor al deze grafen beslissen of ze een partiële *k-tree* zijn en doen dit als volgt:

Voor elke knoop v in $G(i,j)$:

- bepaal alle *k-vertex separators* C_m uit de lijst waarvoor geldt dat $C_m \subseteq C_i \cup \{v\}$. Merk op dat v een deel kan zijn van C_i, waardoor C_i de enige mogelijke C_m is. In de volgende stap geven we aan dat het geen zin heeft dat C_i gelijk is aan C_m, daarom eisen we $C_m \neq C_i$.

- Beschouw alle $G(m,l)$ die een subgraaf zijn van $(G(i,j) - C_i) \cup C_m = C_i^j \cup C_m$ en die een partiële *k-tree* zijn. Indien nu geldt dat $o(G(m,l)) < o(G(i,j))$, dan bestaat de controle of $G(m,l)$ een partiële *k-tree* is uit een opzoeking in onze datastructuur. We berekenen de grafen namelijk van klein naar groot en weten reeds dat $G(m,l)$ een component is van de volledige graaf, na verwijdering van C_i. We tonen aan dat $G(m,l) \subset G(i,j)$: we weten reeds dat $C_m \subset C_i \cup \{v\}$, en dat v zeker in C_m zit. Het is eenvoudig in te zien dat er precies één knoop meer in $C_i \cup \{v\}$ zit dan in $C_m \cup \{v\}$. Bijgevolg is de orde van $G(m,l)$ inderdaad kleiner dan die van $G(i,j)$en kunnen we de opzoektechniek gebruiken[5]

  Merk op dat, indien $C_m = C_i$, we nog steeds niets weten over $G(i,j)$ en we dit geval dus buiten beschouwing laten.

- Neem de unie van alle C_m^l'en uit de vorige stap, over de verschillende m'en en l'en. Indien deze unie de graaf $G(i,j) - C_i$ bevat, zet dan

  $$pkt(i,j) = \texttt{YES},$$

  en stop met de knopen verder te beschouwen (met andere woorden, spring uit de binnenste lus).

Indien nu nog steeds $pkt(i,j) = \texttt{UNDEFINED}$, dan zetten we $pkt(i,j) = \texttt{NO}$. Dit omdat we niet in staat zijn geweest om een embedding te vinden voor een subgraaf. Dit wil echter niet zeggen dat de gehele graaf geen partiële *k-tree* is, omdat de betreffende component niet noodzakelijk door alle separators "geproduceerd" kan worden.

7.3.4 Conclusiefase

Op het einde van elke $G(i,j)$ kunnen we mogelijk al conclusies trekken over de gehele graaf:

- Als er een separator C_m is, zodat voor alle l geldt dat $pkt(m,l) = \texttt{YES}$, dan mogen we concluderen dat G een partiële *k-tree* is. Dit mag omdat we dan C_m als basis kunnen gebruiken, en voor elk van de verschillende $G(m,l)$een *k-tree* kunnen opstellen waarin deze embedbaar is. Wanneer we al deze *k-trees* samenvoegen tot één graaf, door de knopen uit C_m als gemeenschappelijk te beschouwen, bekomen we één grote *k-tree*, waarin G embedbaar is.

- Als elke mogelijke separator C_m van G minstens één $G(m,l)$ heeft waarvoor $pkt(m,l) = \texttt{NO}$, dan mogen we besluiten dat G *geen* partiële *k-tree* is. Dit wegens het feit dat er in dat geval voor elke separator minstens één component niet embedbaar is in een partiële *k-tree*.

[5] Dit is dynamisch programmeren!

7.3.5 Complexiteit en correctheid

Stelling 7.26. *De complexiteit van het algoritme om* k-trees *te herkennen bedraagt* $O(n^{k+2})$.

Bewijs. We analyseren de verschillende fasen van het algoritme:

- Om alle *k-vertex separators* te vinden moeten we alle mogelijke deelverzamelingen van grootte k beschouwen uit de knopen van G. Dit zijn er $n \times (n-1) \times \ldots \times (n-k+1) = O(n^k)$. Om vervolgens te controleren of de verzameling wel degelijk een separator is van G hebben we $O(n^2)$ tijd nodig. Deze controle gebeurt als volgt: verwijder de betreffende knopen uit de graaf ($O(n)$) en ga na of de graaf geconnecteerd is. Indien niet is de verzameling een separator. Nagaan of de resulterende graaf geconnecteerd is kan in $O(n^2)$ op de volgende manier: beschouw de verzameling met een willekeurige knoop in. Breid telkens de verzameling uit met alle buren van de (nieuwe) knopen, dit laatste kunnen we maximaal n keer doen (we kunnen maximaal n knopen toevoegen en in het slechtste geval is dit ééntje per keer). Het opzoeken van de buren van een knoop vraagt $O(n)$ tijd. Bijgevolg heeft een controle of de graaf geconnecteerd is $O(n^2)$ tijd nodig en een test of een verzameling knopen een separator is $O(n + n^2) = O(n^2)$. De totale initialisatiefase vergt dus $O(n^{k+2})$ tijd.

- Vervolgens moeten de C_i^j's sorteren, hetgeen met bucketsort $O(n^{k+1})$ tijd vraagt.

- Indien tijdens de uitvoering van het algoritme voor elke separator incrementeel bijgehouden wordt hoeveel van de door hem veroorzaakte componenten er geldige en ongeldige partiële *k-trees* zijn, kunnen we de eindcondities in constante tijd controleren.

- Elke separator veroorzaakt hoogstens $n-1$ verschillende componenten, dus de for-lus zal $O(n^k \times (n-1))$ keer uitgevoerd worden. Elke $G(i,j)$ bevat hoogstens $n-1$ knopen. Merk op dat de buitstenste for-lus over $O(n^k)$ separators loopt, waarbij er telkens $O(n)$ conponenten zijn, waarvoor alle interne knopen worden afgelopen. In principe heeft dit complexiteit $O(n^{k+2})$, maar omdat we weten dat voor elke separator, alle knopen van de componenten in het totaal slechts één keer beschouwd worden mogen we besluiten dat het slechts $O(n^{k+1})$ tijd vergt. Merk op dat we in essentie de geaugmenteerde componenten beschouwen, en dat voor elke zulke component de knopen van de serarators ook nog eens beschouwd kunnen worden, maar deze zijn constant, waardoor de complexiteit hetzelfde blijft:

$$O(n^k) \times (O(n) + k \times O(n))) = O(n^{k+1}).$$

Het berekenen van de C_m'en die in $C_i \cup \{v\}$ vervat zijn kan in constante tijd gebeuren: $|C_i \cup \{v\}| = k + 1$, herinner dat k een constante is en dat dit dus ook het geval is voor aantal mogelijke deelverzamelingen van grootte k. We testen een constant aantal verzamelingen met een subset-test. De subset-test zelf wordt ook in constante tijd uitgevoerd, omdat we "slechts" een vast aantal van k elementen testen.

- De dominerende complexiteit van de gehele for-loop komt neer op $O(n^{k+1} \times n) = O(n^{k+2})$.

□

Om de correctheid van het algoritme aan te tonen hebben we enkele lemma's nodig:

Lemma 7.27. *Beschouw een partiële* k-tree G*, dan kunnen we een* k-tree *met dezelfde knopen construeren die een supergraaf is, zodat er een knoop uit* G *bestaat waarvoor geldt dat alle buren onderling verbonden zijn in de* k-tree*. Verder is het aantal buren van deze knoop gelijk aan* k*.*

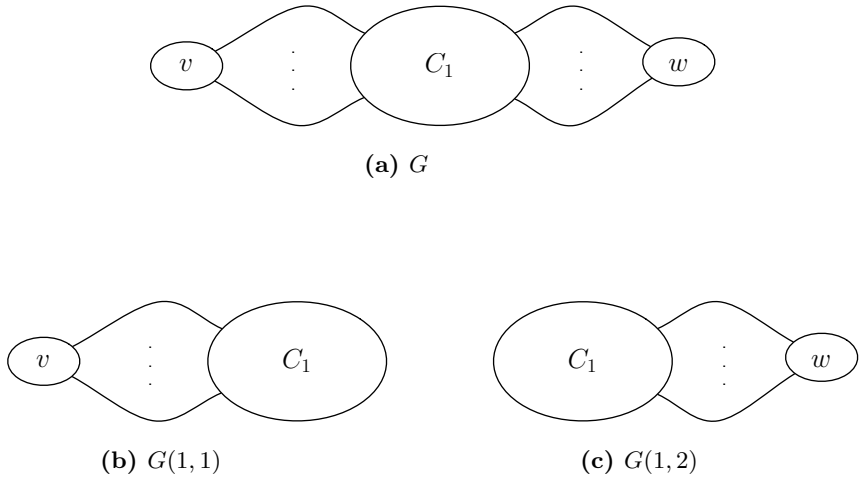
Bewijs. We construeren een tree-decompositie die genormaliseerd is en waarin dus in elke *bag* $k+1$ knopen zitten. Beschouw nu een blad van deze tree-decompositie, we weten nu dat er in dit blad één knoop zit die niet in de ouder zit. Indien we de overeenkomstige *k-tree* van deze tree-decompositie beschouwen, zullen alle knopen in de beschouwde *bag* met elkaar verbonden worden. De k buren van v zijn bijgevolg allen onderling verbonden. Merk op dat we v zo gekozen hebben dat er geen andere buren meer zijn. De buren van v vormen bijgevolg een separator van de *k-tree* en dus zéker van de partiële *k-tree*. □

Lemma 7.28. *Een graaf* G *met* $o(G) \geq k + 2$ *is een partiële* k-tree *als en slechts als er een* k-vertex separator C_i *bestaat zodat alle grafen* $G(i, j)$ *partiële* k-trees *zijn.*

Bewijs. $\Rightarrow$

basis Beschouw een partiële *k-tree* G die uit $k + 2$ knopen bestaat en beschouw een *k-vertex separator* C_1. Omdat we een graaf met precies $k + 2$ knopen beschouwen zorgt C_1 ervoor dat de graaf uiteenvalt in precies twee knopen die niet met elkaar verbonden zijn, noem deze v en w.

We nemen nu de twee grafen $G(1, 1)$ en $G(1, 2)$ die de complete versie van C_1 en respectievelijk de knoop v en w bevatten. De constructies zijn weergegeven in Figuur 7.11. Wegens eigenschap 7.25 weten we dat deze grafen partiële *k-trees* zijn, omdat ze minder dan $k + 2$ knopen bevatten.



(a) G

(b) $G(1,1)$

(c) $G(1,2)$

Figuur 7.11: partiële $k+2$-*tree* en geaugmenteerde componenten

Merk op: Het vinden van een k-*vertex separator* is altijd mogelijk, omdat we een graaf beschouwen die uit $k+2$ knopen bestaat en een partiële k-*tree* is. Hieruit weten we dat het geen $k+2$-*clique* is: de enige graaf van $k+2$ knopen waarin het onmogelijk is om een k-*vertex separator* te vinden.

inductie Beschouw nu een graaf $G = (V, E)$, met $o(G) \geq k+3 (= n)$ en neem aan dat de eigenschap geldt voor $n \leq k+2$. Indien G een partiële k-*tree* is, dan is er wegens lemma 7.27 een knoop v waarvoor geldt dat we een een k-*tree* kunnen construeren die een supergraaf is van G (door enkel bogen toe te voegen) en waarin de buren van v een *clique* vormen. Neem S als de burenverzameling van v. Merk op dat $|S| = k$, omdat v enkel in een blad zit in de tree-decompositie.

Definieer nu de graaf

$$G_1 = (V - \{v\}, (E \cup \{\{u,w\} \mid u,w \in S\}) \setminus \{\{u,w\} \mid u,w \in E \wedge (u = v \vee w = v)\}). \quad (7.1)$$

G_1 is dus met andere woorden de graaf die bekomen wordt door v samen met al zijn bogen uit G te verwijderen en van de buren van v een complete subgraaf te maken. Merk op dat S een separator voor G is, omdat deze v scheidt van de rest van de graaf. Ook dit is zichtbaar in de tree-decompositie: v zit noodzakelijkerwijs in een blad. G_1 is een partiële k-*tree* die minstens $k+2$ knopen bevat en wel om de volgende reden: We weten dat er een k-*tree* bestaat waarvan

G een subgraaf is en waarin S volledig geconnecteerd is. We weten ook dat v slechts met k knopen verbonden is. Indien we v verwijderen bekomen we een *k-tree* die duidelijk een supergraaf is van G_1 (enkel bogen van S zijn toegevoegd, en die zitten allemaal in de *k-tree*). Bijgevolg is G_1 een partiële *k-tree* met minstens $k+2$ knopen en wegens de inductiehypothese geldt dat er een *k-vertex separator* C_i bestaat zodat alle $G(i,j)$'s partiële *k-trees* zijn.

Nu hebben we drie gevallen:

geval 1 $C_i = S$
: We weten dat het lemma geldt en dat elke component van G_1 samen met de complete C_i $(= S)$ een partiële *k-tree* is. De enige component die overblijft om te controleren is $S \cup \{v\}$ en deze is ook duidelijk een partiële *k-tree* (maak S compleet en voeg v toe $\Rightarrow k+1$-*clique*). Bijgevolg zijn alle componenten partiële *k-trees*.

geval 2 $C_i \cap S = \emptyset$
: Beschouw de component C_i^j die S bevat, dit is een partiële *k-tree*. Verbind vervolgens v met S en we bekomen opnieuw een partiële *k-tree* (constructieve definitie van een *k-tree*). C_i is in dit geval dus ook een separator voor G (bevat maximum evenveel bogen in $G[S]$ dan G_1 in $G_1[S]$) en levert allemaal $G(i,j)$'s die partiële *k-trees* zijn.

  Maar wat indien S verspreid is over verschillende componenten? Dit is per definitie onmogelijk, omdat S in G_1 een *clique* vormt. Dit laat meteen het nut zien van de constructie van G_1 met een complete S.

geval 3 $C_i \cap S \neq \emptyset$ en $C_i \neq S$
: Dit geval komt voor wanneer S deels samenvalt met de separator, maar minstens één knoop in een component ligt. Opnieuw kunnen we hier aannemen dat S niet verspreid ligt over meerdere componenten (zie geval 2). We verbinden ook hier v met alle knopen van S en dit levert geen probleem op: de gewijzigde component blijft een partiële *k-tree*. v wordt namelijk slechts met één component verbonden op een correcte manier.

Deze drie gevallen beschrijven alle mogelijkheden en zorgen ervoor dat het bewijs in deze richting compleet is.

$\Leftarrow$ Om een *k-tree* te construeren waarvan G een subgraaf is gaan we als volgt te werk: neem de *k-tree*-embeddings van de $G(i,j)$'s en voeg deze samen door de knopen uit C_i als gemeenschappelijk te beschouwen. Merk op dat er geen *clique* van grootte $k+2$ of meer gevormd kan worden door deze constructie toe te passen en we dus noodzakelijkerwijs een *k-tree* bekomen.

Om aan te tonen dat het wel degelijk een supergraaf is van G moeten we controleren of alle knopen en alle bogen wel degelijk aanwezig zijn in de *k-tree*. De aanwezigheid van de knopen is duidelijk, omdat de verschillende geaugmenteerde componenten allemaal samen alle knopen bevatten. De bogen van de separator zelf zitten sowieso allemaal erin, dus enkel de bogen in de verschillende componenten moeten nog gecontroleerd worden. Het is duidelijk dat de knopen uit verschillende componenten geen contact met elkaar hebben en dat de *k-trees* alle nodige bogen bevatten. De gehele *k-tree* is bijgevolg een supergraaf voor G.

□

Lemma 7.29. *Een geaugmenteerd component $G(i,j)$ is een partiële* k-tree *als en slechts alser een knoop v bestaat in $G(i,j)$ en een verzameling F met* k-vertex separators $C_m \neq C_i$ *die elk een subset zijn van $C_i \cup \{v\}$, zodat de verzameling $\{G(m,l) - C_m\}$ een partitie*[6] *is van $\{G(i,j) - C_i - \{v\}\}$*

Bewijs. $\Rightarrow$
We weten dat een *k-tree* geconstrueerd kan worden uit elke k-complete subgraaf (zie Proskurowski [25]). Als $G(i,j)$ een partiële *k-tree* is, dan kunnen we elke *k-tree*-embedding T construeren vanuit C_i (herinner: C_i is compleet en bestaat uit k knopen in $G(i,j)$): Neem C_i en voeg er een knoop v aan toe die je met alle knopen verbindt. Het overige deel van de *k-tree* kunnen we vervolgens construeren met behulp van een reeks *k-trees* die als basis een deelverzameling van $C_i \cup \{v\}$ hebben (van grootte k). Deze *k-trees* overlappen enkel op $C_i \cup \{v\}$. We moeten nu enkel nog aantonen dat ze een partitie vormen:

Indien ze geen partitie vormen, dan zijn er twee componenten die overlappen. Stel er zijn twee verschillende separators C_m en $C_{m'}$, zodat de twee geaugmenteeerde componenten $G(m,l)$ en $G(m',l')$ één of meer knopen gemeenschappelijk hebben. Neem een knoop x uit deze doorsnede, een knoop y uit C_m^l die niet in de doorsnede zit en een knoop y' uit C_m^l die niet in de doorsnede zit. We weten dat y (onrechtstreeks) verbonden moet zijn met x, buiten C_m, anders zouden ze namelijk in verschillende componenten terecht zijn gekomen. Nu moet y noodzakelijk ook in $C_{m'}^{l'}$ zitten, omdat:

- Als y verbonden is met x in C_m^l dan kunnen ze niet uit elkaar gehaald worden door $C_{m'}$, en
- indien y verbonden is met x via w', dan zou w' in C_m^l zitten, en dus ook in $G(m,l)$. Wegens $G(m,l) \subseteq C_i^j \cup C_m$, mag w' er dus niet inzitten.

Is het mogelijk dat alle C_m'en deels buiten C_i^j vallen? Dat zie je aan constructie: je neemt de C_m, die een knoop uit C_i laat vallen en daarmee bouw je een deel van de C_i^j, het is duidelijk dat dat volledig binnen C_i^j

[6] Zie definitie 3.50.

valt. Voor elk van de stukjes gaat dat op. Het idee is dat we de stukjes met verschillende separatoren omzetten naar partiële *k-trees*, zoals ook een tree-decompositie wordt opgebouwd. Het is enkel verwarrend omdat we de hele graaf beschouwen, dus ook overbodige componenten. Hierom achten we het lemma van Bodlaender, dat we later zullen geven, iets duidelijker.

$\Leftarrow$ Veronderstel dat er een dergelijke verzameling F bestaat, dan kan T op de volgende manier geconstrueerd worden: voeg v toe aan C_i bepaal een embedding voor $G(m, l)$'en, die hangen we vervolgens aan het gemeenschappelijke deel hangen. Merk op dat het op deze manier een *k-tree* blijft, omdat we k knopen uitbreiden, de subgrafen (componenten) hebben niets met elkaar te maken.

□

Stelling 7.30. *Het gegeven algoritme klassificeert gegeven grafen correct als partiële* k-trees.

Bewijs. Wegens het voorgaande betoog weten we dat het algoritme alle C_i^j's correct klassificeert en dat het de juiste conclusie trekt indien het algoritme stopt. Het is ook duidelijk dat het algoritme altijd stopt, omdat we alle separators aflopen. □

Een schets van hoe het algoritme loopt op een bepaalde invoer is gegeven in Bijlage D.

Opmerking 7.31. Tot slot merken we op dat er een algoritme bestaat, beschreven door Robertson en Seymour [28] dat in $O(n^2)$ beslist of een gegeven graaf een partiële *k-tree* is, voor een vaste k. Hiervoor heeft men wel kennis nodig van alle verboden minoren[7] voor de betreffende k. Voor kleine waarden voor k zijn deze verzamelingen van minors bekend, voor grotere echter niet.

Ook al is een algoritme uitvoerbaar in $O(n^2)$, dan wil dit nog niet zeggen dat het efficiënt is. Het is perfect mogelijk dat de parameter k verschoven is naar voren, bijvoorbeeld $2^{2^k} n^2$. Het is duidelijk dat de voorfactor in een soortgelijke constructie heel snel groeit en deze het algoritme alsnog praktisch onbruikbaar kan maken voor kleine waarden van k.

[7] Dit is een graaf die bekomen wordt door bogen en knopen op een speciale manier samen te nemen, o.a. Bodlaender [7] geeft hiervan een beschrijving.

7.4 Graafexpressies en partiële k-*trees*

Wanneer we de definitie van *expressionwidth* (definitie 6.20) terug beschouwen, dan kunnen we volgende belangrijke stelling aantonen:

Stelling 7.32. *De* expressionwidth *van een gewone graaf is gelijk aan zijn* treewidth.

Bewijs. We geven een indicatie van het bewijsidee:
$\Rightarrow$ Stel dat we een minimale graafexpressie e hebben, die gebruik maakt van $k+1$ sources. We geven een nieuw soort evaluatiefunctie voor de expressies die als resultaat een tree-decompositie van de graaf, die voorgesteld wordt door de expressie, oplevert.

$a(x, y)$ Maak een *bag* met daarin x en y.

$e_1 \mid e_2$ Maak een tree-decompositie voor e_1 (resp. e_2) en noem deze T_1 (resp. T_2). Construeer nu een nieuwe *bag* waarin de gemeenschappelijke sources van $\|e_1\|$ en $\|e_2\|$ zitten. Voeg aan deze *bag* de kinderen T_1 en T_2 toe.

$(\mathrm{lc}\, x)e$ Maak een tree-decompositie voor e en noem deze T. Construeer nu een *bag* met daarin alle sources van de root van T, uitgezonderd x. Voeg T toe als enig kind van deze nieuwe *bag*.

Deze constructie levert ons duidelijk een tree-decompositie op. Verder gebruiken we overal enkel de sources, hetgeen ons bags van maximaal $k+1$ knopen oplevert.

$\Leftarrow$ Stel dat we een graaf hebben met *treewidth* k, dan wil dit zeggen dat er een genormaliseerde tree-decompositie bestaat voor deze graaf, waarbij alle *bags* een grootte hebben van $k+1$. Begin bij de root en stel voor de $k+1$ knopen een expressie e op. Dit is mogelijk door gebruik te maken van $k+1$ sources. We werken vervolgens recursief verder en passen de volgende procedure toe: bepaal een graafexpressie e_i voor elk kind i, waarbij dezelfde sources gebruikt worden voor de gemeenschappelijke knopen (k). De ene knoop w_i die niet gemeenschappelijk is (herinner: de tree-decompositie is genormaliseerd), die local'en we. We verkrijgen dus:

$$(\mathrm{lc}\, w_1)e_1 \mid e.$$

De extra knoop in de ouder kan dus gebruikt worden door de andere kinderen, uiteindelijk verkrijgen we een graafexpressie van de vorm

$$(\mathrm{lc}\, w_1)e_1 \mid (\mathrm{lc}\, w_2)e_2 \mid \ldots \mid (\mathrm{lc}\, w_i)e_i \mid e,$$

waarbij i het aantal kinderen van de knoop is.

□

Het voorgaande en stelling 7.21 laten ons toe om hetvolgende te besluiten:

Besluit 7.33. *Het bepalen van de* expressionwidth *van een graaf is NP-compleet.*

Herinner dat we in hoofdstuk 6 reeds opgemerkt hebben dat het aantal sources dat gebruikt kan worden in een graafgrammatica beperkt is, waardoor alle gegenereerde expressies ook een beperkt aantal sources gebruiken. Dit leidt tot hetvolgende:

Lemma 7.34. *De graafexpressies die gegenereerd worden door een graafgrammatica gebruiken een beperkt aantal sources*

Besluit 7.35. *Een graafgrammatica genereert enkel grafen met een begrensde* treewidth.

Bewijs. Lemma 7.34 geeft aan dat elke gegenereerde graafexpressie een begrensd aantal sources gebruikt. We weten reeds uit stelling 7.32 dat *treewidth* en *expressionwidth* in essentie hetzelfde zijn. □

Dit heeft als gevolg dat we bepaalde eigenschappen van partiële *k-trees* kunnen gebruiken om efficiënte algoritmen te bouwen voor het genereren van de graaftaal van een grammatica. In het volgende hoofdstuk gaan we dieper in op enkele van deze eigenschappen.

Hoofdstuk 8

Isomorfie en canonisatie van partiële k-*trees* in polynomiale tijd

In het algemeen is het niet bekend of het isomorfieprobleem van grafen in polynomiale tijd oplosbaar is, of dat het NP-compleet is. Wat wel aangetoond is door Johnson [21] is dat het probleem in polynomiale tijd oplosbaar is wanneer het beperkt wordt tot partiële k-*trees* met een begrensde k. Zoals in het vorige hoofdstuk reeds werd aangegeven, is het mogelijk om van twee partiële k-*trees*, met een bovengrens op hun k, in polynomiale tijd te beslissen of ze al dan niet isomorf zijn. Bodlaender [6] laat zien dat er een dergelijk algoritme bestaat. In dit hoofdstuk gaan we dieper in op dit algoritme en geven we extra informatie waar nodig.

Hiernaast bekijken we ook nog een voorstel tot canonisatie van partiële k-*trees*. Ook dit zal mogelijk blijken te zijn in polynomiale tijd voor grafen met een begrensde *treewidth*[1].

In het volgende hoofdstuk zullen we aangeven hoe deze algoritmen kunnen gebruikt worden bij de constructie van algoritmen voor het genereren van graafexpressies.

8.1 Isomorfie van partiële k-*trees*

Het algoritme bouwt verder op de partiële k-*tree*-test die we in het vorige hoofdstuk beschreven hebben en gebruikt vergelijkbare technieken (dynamisch programmeren, de k-*decomposable*-eigenschap).

We gebruiken eenzelfde notatie en aanpak wat betreft de separators. Lemma 7.28 en 7.29 zijn hier opnieuw van toepassing. We kunnen het tweede lemma echter op een meer duidelijke, maar equivalente manier formuleren

[1] dit is hetzelfde als zeggen dat we partiële k-*trees* beschouwen met een begrensde k.

als volgt:

Lemma 8.1. *Een graaf $G(i,j)$ die minstens $k+2$ knopen bevat is een partiële* k-tree *als en slechts als er een knoop v bestaat in C_i^j, zodat voor elke geconnecteerde component A van de graaf $C_i^j[V(C_i^j)-\{v\}]$ (de graaf verkregen door v te verwijderen uit C_i^j) er een* k-vertex separator $C_m \subset C_i \cup \{v\}$ *bestaat, zodat:*

- *Geen enkele knoop in A verbonden is met de unieke knoop in $C_i \cup \{v\} \setminus C_m$, en*
- *$G(C_m, A)$ is een partiële* k-tree.

Het verschil is dat het hier duidelijker is dat we overgaan naar de kleinere partiële *k-tree* en we de grote graaf niet echt meer in beschouwing nemen. Merk op dat $C_m \neq C_i$. Het is duidelijker dat dit lemma geldt: wanneer we weten dat het om een partiële *k-tree* gaat, dan kunnen we een tree-decompositie bouwen met als root C_i, hier een knoop van losmaken, en v erbij nemen om de verschillende componenten te "bedekken".

Vervolgens definiëren we een notie van isomorfisme tussen (separator,component)-paren van verschillende grafen G en H. De separators en componenten van de graaf H duiden we aan met de letter D, voorzien van het gebruikelijke bijschrift. Die van G nog steeds met C.

Definitie 8.2. Neem twee grafen G en H, waarvan de separators genoteerd worden met C_i resp. $D_{i'}$. Beschouw een bijectie $f : C_i \to D_{i'}$. Een paar (C_i, C_i^j) is *f-isomorf* met een component $(D_{i'}, D_{i'}^{j'})$, genoteerd

$$(C_i, C_i^j) \cong^f (D_{i'}, D_{i'}^{j'}),$$

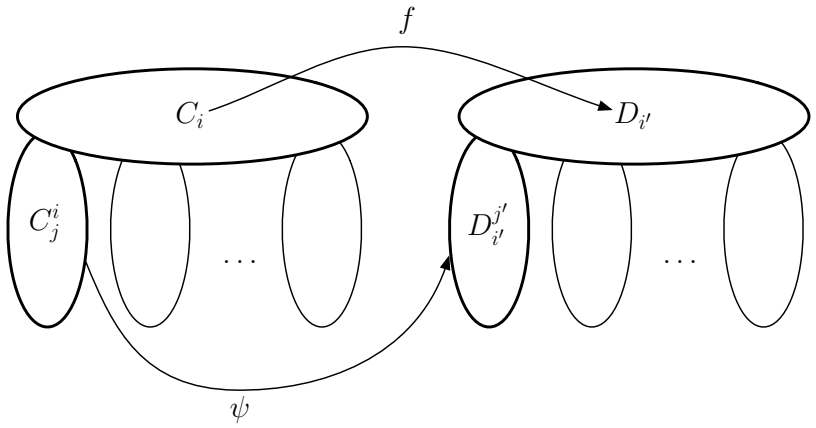
als en slechts alser een functie $\psi : C_i \cup C_i^j \to D_{i'} \cup D_{i'}^{j'}$ bestaat, zodat:

1. $\forall v \in C_i : \psi(v) = f(v)$, en
2. $\forall v, w \in C_i \cup C_i^j : \{v,w\} \in E(G) \Leftrightarrow \{\psi(v), \psi(w)\} \in E(H)$

Bovenstaande definitie geeft aan dat twee zulke paren f-isomorf zijn indien we de functie kunnen uitbreiden naar de betreffende componenten en we een isomorfisme verkrijgen. Merk op dat f een isomorfisme moet zijn tussen de separators. Figuur 8.1 geeft dit schematisch weer.

We zijn nu klaar om het algoritme te beschrijven. Eerst runnen we het algoritme van Arnborg (zie hoofdstuk 7), maar we zorgen dat we buiten het antwoord `YES` nog wat extra informatie bijhouden:

- een *k-vertex separator* C_r, zodat alle $G(r,l)$ partiële *k-trees* zijn (lemma 7.28), en



Figuur 8.1: f-isomorfisme

- voor elke in het vorige punt beschouwde $G(i,j)$, houden we een knoop $v \in C_i^j$ bij, zodat voor elke component A, zoals beschreven in lemma 8.1 geldt:
  - geen knoop uit A is verbonden met de unieke knoop (zie lemma), en
  - $G(C_m, A)$ is een partiële *k-tree*, met ofwel minder dan $k{+}1$ knopen ofwel hebben we gelijkaardige informatie voor $G(C_m, A)$.

Deze data vormt de "representatie van G als partiële *k-tree*" en we beschouwen enkel deze data. Merk op dat er $O(n)$ $G(i,j)$'s en C_i^j's zijn. Dit is eenvoudig in te zien: we weten dat het algoritme van Arnborg ons een separator kan afleveren mits enkele aanpassingen. Een separator verdeelt de graaf vervolgens in $O(n)$ componenten. Elk van die componeten kunnen we nog augmenteren, ook dit zijn er $O(n)$. Herinner dat het algoritme van Arnborg $O(n^{k+2})$ tijd vergt.

De volgende stap is om de separators $D_{i'}$ en de $D_{i'}^{j'}$'s te berekenen voor de graaf H, dit kan in $O(n^{k+1})$. Volgende lemma's zijn nodig om te beschrijven hoe het algoritme te werk gaat.

Lemma 8.3. *Beschouw een graaf G, met een k-*vertex separator *C_r, zodat alle $G(i,j)$'s partiële k-*trees *zijn. Veronderstel dat het aantal componenten C_i^j gelijk is aan m.*

*Voor een graaf H geldt nu dat $G \cong H$ als en slechts als H een k-*vertex separator *D_s heeft, zodat er een bijectie $f : C_r \to D_s$ bestaat, zodat:*

- *$\forall v, w \in C_r : (v,w) \in E(G) \Leftrightarrow (f(v), f(w)) \in E(H)$, met andere woorden: de bogen tussen de knopen in de separator worden afgebeeld op bestaande bogen in H, ook tussen knopen uit D_s.*

- *Beschouw de geconnecteerde componenten van* $H[V(H) - D_s]$ *en noem deze* $D_s^1, \ldots, D_s^{m'}$. *Er geldt dat* $m = m'$ *en dat er een bijectie* $\Phi : \{1, \ldots, m\} \to \{1, \ldots, m\}$ *bestaat, zodat voor alle* l *met* $1 \leq l \leq m$ *geldt dat*

$$(C_r, C_r^l) \cong^f (D_s, D_s^{\Phi(l)})$$

Bewijs. $\Rightarrow$ Veronderstel dat $G \cong H$, dan bestaat er een isomorfisme ψ van G naar H. Neem vervolgens $D_s = \psi(C_r)$. D_s bestaat uit de "overeenkomstige" knopen van G in H en is dus zeker een separator. Hiernaast geldt dus ook dat alle $H(s, j')$'s partiële k-*trees* zijn. We tonen vervolgens aan dat deze separator een goede keuze is om de voorwaarden uit het lemma waar te maken.

Kies voor $f = \psi_{|C_r}$, dus hetzelfde als ψ, maar dan beperkt tot het domein van de separator. Het spreekt voor zich dat f een bijectie is van C_r naar D_s, wegens de keuze van D_s.

Elke C_r^j wordt door ψ gemapt op zekere $D_s^{j'}$, waarbij we de Φ voor elke j als volgt invullen $\Phi(j) = j'$. Het is duidelijk dat er m componenten zijn en dat we dus een bijectie bekomen van de volgende vorm:

$$\Phi : \{1, \ldots, m\} \to \{1, \ldots, m\}.$$

We tonen tot slot aan dat voor alle l, $1 \leq l \leq m$ geldt dat

$$(C_r, C_r^l) \cong^f (D_s, D_s^{\Phi(l)}).$$

voor de ψ uit definitie 8.2 kiezen we de door ons geconstrueerde ψ, beperkt tot het domein van separator en component. Voorwaarde 1 uit de definitie is voldaan omdat $f = \psi_{|C_r}$. Voorwaarde 2 is eveneens voldaan omdat de componenten en separators uit de grafen op elkaar worden afgebeeld door het isomorfisme, en de bogen dus behouden blijven.

$\Leftarrow$ Construeer $\psi : V(G) \to V(H)$ als volgt:

- als $v \in C_r : \psi(v) = f(v)$
- als $v \in C_r^l (1 \leq l \leq m)$, dan is er, wegens het feit dat $(C_r, C_r^l) \cong^f (D_s, D_s^{\Phi(l)})$ en definitie 8.2, een functie $\psi^l : C_r \cup C_r^l \to D_s \cup D_s^{\Phi(l)}$, zodat

  1. $\forall v \in C_r : f(v) = \psi^l(v)$
  2. $\forall v, w \in C_r \cup C_r^l : (v, w) \in E(G) \equiv (\psi^l(v), \psi^l(w)) \in E(H)$.

  We kiezen in dit geval $\psi = \psi^l$.

We tonen nu aan dat onze constructie van ψ een geldig isomorfisme van G naar H oplevert.

- $v, w \in C_r$

$$\begin{aligned}&\{v, w\} \in E(G)\\ &\Leftrightarrow \{f(v), f(w)\} \in E(H)\\ &\Leftrightarrow \{\psi(v), \psi(w)\} \in E(H).\end{aligned}$$

- $v, w \in C_r^l$

$$\begin{aligned}&\{v, w\} \in E(G)\\ &\Leftrightarrow \{\psi^l(v), \psi^l(w)\} \in E(H)\\ &\Leftrightarrow \{\psi(v), \psi(w)\} \in E(H).\end{aligned}$$

- $v \in C_r^{l_1}, w \in C_r^{l_2}$
  Met andere woorden, v en w zitten in verschillende componenten. Er kan dus onmogelijk een rechtstreekse boog tussen v en w zijn, dus

$$\{v, w\} \notin E(G).$$

  Verder geldt nu dat

$$\begin{aligned}\psi(v) &\in D_s^{\Phi(l_1)},\\ \psi(w) &\in D_s^{\Phi(l_2)}.\end{aligned}$$

  Omdat Φ een bijectie is, en $l_1 \neq l_2$ is ook $\Phi(l_1) \neq \Phi(l_2)$. $\psi(v)$ en $\psi(w)$ zitten bijgevolg in verschillende componenten in H, waardoor er ook geen boog tussen hen bestaat:

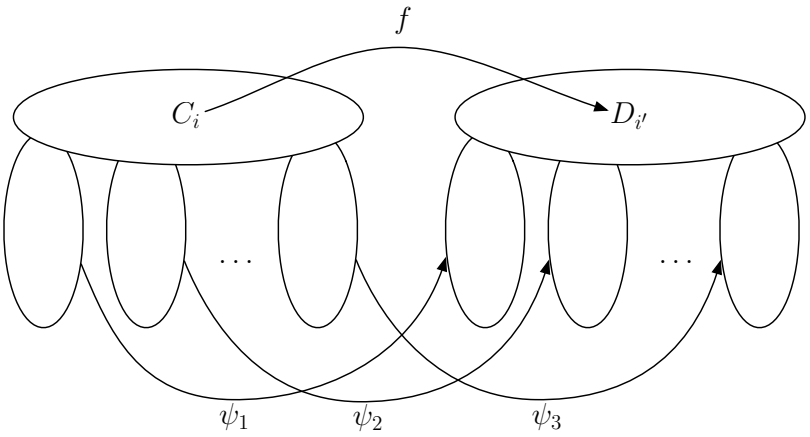
$$\{\psi(v), \psi(w)\} \notin E(H).$$

- $v \in C_r, w \in C_r^l$

$$\begin{aligned}&\{v, w\} \in E(G)\\ &\Leftrightarrow \{\psi^l(v), \psi^l(w)\} \in E(H)\\ &\Leftrightarrow \{f(v), \psi^l(w)\} \in E(H)\\ &\Leftrightarrow \{\psi(v), \psi(w)\} \in E(H).\end{aligned}$$

- $v \in C_r^l, w \in C_r$
  Dit is volledig analoog aan het vorige geval.

□



Figuur 8.2: Isomorfisme van een partiële *k-tree*.

Dit lemma zegt ons dat, wanneer we twee isomorfe grafen beschouwen, we op de volgende manier een isomorfisme kunnen construeren: beschouw twee separators en construeer een isomorfisme. Bouw een isomorfisme tussen elke $G(r, j)$ en $H(s, j')$, dat het isormfisme tussen de separators respecteert. Omdat de verschillende geaugmenteerde componenten enkel een overlap hebben op de separator en de verschillende isomorfismes het separator-isomorfisme bewaren, kunnen we één groot isomorfisme bouwen voor de hele graaf. Hoe dit werkt is te zien in Figuur 8.2. De verschillende ψ_i's zijn allemaal uitbreidingen van de functie f en bewaren bijgevolg de mapping van de separator. Hieruit volgt dat we, wegens het feit dat de componenten allen isomorf zijn en geen invloed op elkaar uitoefenen, de isomorfismes kunnen combineren tot een groot isomorfisme voor de hele graaf, gebruik makende van f en de ψ_i's.

Het is nu enkel nog de opdracht om de juiste bijecties te vinden, die we dan kunnen samenvoegen. Herinner dat we voor G een separator verkregen hebben door het algoritme van Arnborg te runnen. We gaan nu voor elke mogelijke separator van H moeten kijken of we een isomorfisme kunnen construeren, vervolgens moeten we ook in staat zijn om een isomorfisme tussen de "blaadjes" te bouwen, hetgeen ook weer veel mogelijkheden met zich meebrengt.

Opnieuw wordt hier gebruik gemaakt van dynamisch programmeren, we trachten eerst isomorfismes te vinden voor subgrafen en deze vervolgens te gebruiken bij de zoektocht naar grotere isomorfismes. Alvorens we dieper ingaan op het algoritme, hebben we nog één lemma nodig, dat we niet aantonen (het bewijs is analoog aan dat van lemma 8.3).

Lemma 8.4. *Beschouw $G(i, j)$, en neem aan dat dit een partiële k-*tree *is. Kies v in C_i^j, zodat voor elke geconnecteerde component A_p van de graaf*

$C_i^j[V(C_i^j) - \{v\}]$ *er een* k-vertex separator $C_{m_p} \subset C_i \cup \{v\}$ *bestaat, zodat:*

- *Geen enkele knoop in* A_p *verbonden is met de unieke knoop in* $C_i \cup \{v\} \setminus C_{m_p}$, *en*
- $G(C_{m_p}, A_p)$ *is een partiële* k-tree.

Noem m *het aantal van deze componenten* A_p.

$D_{i'}$ *is een* k-vertex separator *voor een graaf* H, *en* $D_{i'}^{j'}$ *is een geconnecteerde component van* $H[V(H) - D_{i'}]$, *zoals hiervoor.*

Neem nu een bijectie $f : C_i \to D_{i'}$, *dan is* $(C_i, C_i^j) \cong^f (D_{i'}, D_{i'}^{j'})$ *als en slechts alsaan de volgende voorwaarden voldaan is:*

1. $\forall v, w \in C_i : (v, w) \in E(G) \Leftrightarrow (f(v), f(w)) \in E(H)$.
2. $\exists w \in D_{i'}^{j'}$, *zodat het aantal geconnecteerde componenten van de graaf die we bekomen door* w *te verwijderen uit* $D_{i'}^{j'}$ *gelijk is aan* m, *en voor elk van deze componenten* B_q *bestaat er een* k-vertex separator $D_{m_q} \subset D_{i'} \cup \{w\}$, *zodat*
   - *geen enkele knoop in* B_q *verbonden is met de unieke knoop in* $D_{i'} \cup \{w\} \setminus D_{m_q}$
   - *er bestaat een bijectie* $\Phi : \{1, \ldots, m\} \to \{1, \ldots, m\}$, *zodat voor alle* l, *met* $1 \leq l \leq m$, *geldt dat:*

     $$(C_{m_l}, A_l) \cong^{f'} (D_{m_{\Phi(l)}}, B_{\Phi(l)}),$$

     waarbij $f'(x) = f(x)$ *als* $x \in C_i \cap C_{m_l}$ *en* $f'(v) = w$.
   - *voor elke* B_q *is er een* l, *zodat* $B_q = D_{m_q}^l$ *en* $|B_q| < |D_{i'}^{j'}|$

Het algoritme moet voor elke (C_i, C_i^j) die in de voorstelling van G als partiële *k-tree* voorkomt gaan zoeken met welke mogelijke $(D_{i'}, D_{i'}^{j'})$ deze matcht. Dit doen we door alle mogelijke (separator-component)-paren van H $(D_{i'}, D_{i'}^{j'})$ af te gaan en dan ook nog eens alle mogelijke mappings $f : C_i \to D_{i'}$ te beschouwen. We proberen dit echter zo efficiënt mogelijk te doen.

We sorteren eerst alle $(D_{i'}, D_{i'}^{j'})$ volgens grootte en behandelen ze van klein naar groot, één voor één. We onderscheiden twee gevallen, waarbij het eerste zich voordoet als $|D_{i'}^{j'}| \leq 1$ (merk op dat een component nooit leeg kan zijn en de grootte exact 1 zal zijn). In dit geval gaan we voor elke mogelijke (C_i, C_i^j) trachten een mapping te vinden. Herinner dat er $O(n)$ zulke paren zijn. Voor elk van deze moeten we alle mogelijke functies f beschouwen van de hiervoor beschreven vorm. Omdat de functie verzamelingen van grootte k op elkaar afbeeldt, zijn er $k!$ mogelijkheden, maar herinner: k is

een constante. Alle functies beschouwen vergt dus $O(1)$ tijd. Nu moeten we controleren of wel degelijk

$$(C_i, C_i^j) \cong^f (D_{i'}, D_{i'}^{j'}).$$

Hierbij is $\psi(x)$ uit definitie 8.2 gelijk aan

$$\psi(x) = \begin{cases} f(x) & \text{indien } x \in C_i \\ w & \text{indien } x = v \end{cases}$$

Merk op dat we in constante tijd kunnen checken of de bogen in beide grafen op elkaar gemapt worden door f. Er zijn namelijk $O((k+1)^2) = O(1)$ bogen. De totale tijd nodig voor alle paren waarbij de componenten grootte 1 hebben bedraagt dus

$$\begin{aligned} O(n^{k+1} \times n \times k! \times (k+1)^2) &= O(n^{k+1} \times n \times 1 \times 1) \\ &= O(n^{k+2}) \end{aligned}$$

(Herinner: de n^{k+1} is afkomstig van de $O(n^k)$ separators en $O(n)$ mogelijke componenten).

In het andere geval, wanneer de grootte van de component groter of gelijk is aan 2, dan gaan we ook alle geldige (C_i, C_i^j)'s af. Herinner dat het algoritme van Arnborg ons een knoop v oplevert die aan bepaalde voorwaarden voldoet (lemma's 7.28 en 8.1). Ook hier lopen we alle mogelijke functies f af.

Eerst testen we of elke boog uit de separator C_i gemapt wordt op eentje uit de separator van H $(D_{i'})$:

$$\forall v, w \in C_i : (v, w) \in E(G) \Leftrightarrow (f(v), f(w)) \in E(H).$$

Indien dit zo is, dan gaan we voor elke mogelijke w in $D_{i'}^{j'}$ hetvolgende doen:

- bereken alle geconnecteerde componenten van $D_{i'}^{j'}[V(D_{i'}^{j'}) - \{w\}]$
- er zijn maximum $k+1$ verschillende separators die een deelverzameling zijn van $D_{i'} \cup \{w\}$. Voor elk van deze D' testen we of er knopen uit B_q verbonden zijn met de unieke knoop in $(D_{i'} \cup \{w\}) \setminus D'$.
- voor elke D' waarvoor er zo geen knoop bestaat bepalen we voor welke (C_{m_p}, A_p) er geldt dat

  $$(C_{m_p}, A_p) \cong^{f'} (D', B_q),$$

  met f' zoals in lemma 8.4. Omdat $|B_q| < |D_{i'}^{j'}|$ hebben we deze informatie al berekend en kunnen we "deze opzoeken" (dynamische programmeerstijl).

Er rest ons nu nog om de bijectie Φ te construeren voor alle "kinderen". Dit doen we door het probleem te vertalen naar het vinden van een perfecte matching in een bipartite graaf[2]. De ene verzameling knopen stellen alle A_p's voor, de andere alle B_q's en er zijn bogen tussen hen indien er een D' bestaat zodat

$$(C_{m_p}, A_p) \cong^{f'} (D', B_q).$$

Met andere woorden, enkel de componenten die we op mekaar kunnen mappen zijn verbonden. Als we nu zo een perfecte matching kunnen vinden, dus een bijectie van alle A_p's naar alle B_q's, dan mogen we besluiten dat

$$(C_i, C_i^j) \cong^f (D_{i'}, D_{i'}^{j'}).$$

Het vinden van zo een perfecte matching kan in tijd $O(n^{2.5})$ (zie bijvoorbeeld Hopcroft en Karp [20]). In ons geval is het aantal knopen van de bipartite graaf gelijk aan $2m$ (m voor G en m voor H), en $m = O(|D_{i'}^{j'}|)$ (er kunnen maximaal zoveel componenten zijn die we moeten matchen als er knopen in deze deelgraaf zijn).

In het algemeen wordt de totale tijd nodig om één triple (C_i, C_i^j), $(D_{i'}, D_{i'}^{j'})$ en f te verwerken begrensd door $O(|D_{i'}^{j'}|^{3.5}) = O(n^{3.5})$ De totale tijd nodig wordt bijgevolg begrensd door $O(n^k \times n \times n^{3.5}) = O(n^{k+4.5})$

Na het opstellen van bovenstaande informatie, moeten we de uiteindelijke test nog doen om te controleren of de twee grafen isomorf zijn. Dit doen we door lemma 8.4 te controleren. We stellen voor elke mogelijke separator $D_{i'}$ opnieuw een bipartite graaf op met in de ene verzameling alle componenten die bekomen worden door gebruik te maken van C_r en in de andere alle $D_{i'}^{j'}$'s. We zoeken in onze gemaakte datastructuur op welke componenten met welke kunnen matchen en laten vervolgens een perfect matching algoritme uitzoeken of er een isomorfisme mogelijk is. Zodra we voor een zekere $C_{i'}$ zo een isomorfisme vinden, mogen we besluiten dat de grafen isomorf zijn. Indien geen één van de separators van H een dergelijk isomorfisme kan opleveren, dan concluderen we dat ze niet isomorf zijn.

Stelling 8.5. *Er bestaat een algoritme dat beslist of twee partiële k-trees isomorf zijn in tijd $O(n^{k+4.5})$*

Het algoritme wordt in pseudocode gegeven in Algoritme 18. Een voorbeeldrun van het algoritme wordt gegeven in Bijlage E.

[2]Een biparite graaf is een graaf waarbij de knopenverzameling in twee delen geplitst is. Eventuele bogen gaan van knopen uit de ene verzameling naar de andere, niet intern in de verzamelingen

Algoritme 18: Isomorfisme van partiële *k-trees*

Input: G,H
Output: `true` of `false`

run het algoritme van Arnborg;
bepaal alle $D_{i'}$ en $D_{i'}^{j'}$;
sorteer alle $(D_{i'}, D_{i'}^{j'})$ volgens grootte;
foreach $(D_{i'}, D_{i'}^{j'})$ *van klein naar groot* **do**
  if $|D_{i'}^{j'}| \leq 1$ **then**
    foreach (C_i, C_i^j) **do**
      foreach f **do**
        if $(C_i, C_i^j) \cong^f (D_{i'}, D_{i'}^{j'})$ **then**
          $iso(i, j, i', j', f) =$ `true`;
        end
        else
          $iso(i, j, i', j', f) =$ `false`;
        end
      end
    end
  end
  // $|D_{i'}^{j'}| > 1$
  else
    foreach (C_i, C_i^j) **do**
      v wordt gegeven door het algoritme van Arnborg;
      foreach f **do**
        foreach $w \in D_{i'}^{j'}$ **do**
          foreach *component* B_q *van* $D_{i'}^{j'}[V(D_{i'}^{j'}) - \{w\}]$ **do**
            foreach $D' \subset D_{i'} \cup \{w\}$ **do**
              if *geen knoop uit* B_q *is verbonden met de unieke knoop* **then**
                foreach C_{m_p} *en daarvoor veroorzaakte component* A_p **do**
                  // `opzoeken, is reeds berekend:` $|B_q| < |D_{i'}^{j'}|$
                  if $(C_{m_p}, A_p) \cong^{f'} (D', B_q)$ **then**
                    verbind A_p met B_p in Z;
                  end
                end
              end
            end
          end
          Voer een perfect matching op de bipartite graaf Z uit;
          if *bijectie gevonden* **then**
            $iso(i, j, i', j', f) =$ `true`;
          end
        end
      end
    end
  end
end
// `test of de grafen isomorf zijn met behulp van de hiervoor berekende data`
return *finalIsoTest()*;

Algoritme 19: finalIsoTest

```
1  foreach k-vertex separator D_{i'} do
2      foreach f do
3          if f(C_r) = D_{i'} then
4              Z is een bipartite graaf met in de ene verzameling alle C_r^l's en in
               de andere alle D_{i'}^{j'}'s;
5              foreach D_{i'}^{j'} do
                   // het volgende kan opgezocht worden:
                       (C_r, C_r^l) ≅^f (D_{i'}, D_{i'}^{j'})
6                  verbind D_{i'}^{j'} in Z met alle C_r^l's waarvoor geldt dat
                   iso(r, l, i', j', f) = true;
7              end
8              Bereken perfect matching van Z;
9              if matching gevonden then
10                 return true;
11             end
12         end
13     end
14 end
15 return false;
```

8.2 Canonische vorm voor partiële *k-trees*

In deze sectie geven we een aanzet tot een canonische vorm voor partiële *k-trees* die efficiënt berekend kan worden voor een constante k. We schetsen op een hoog-niveau manier de gedachte achter een polynomiaal algoritme. Deze constructie maakt gebruik van het feit dat de verschillende componenten van een *k-tree* onderling onafhankelijk zijn en we bijgevolg voor elke component apart een canonische vorm kunnen berekenen. Deze worden vervolgens samengevoegd tot één geheel door de separator als gemeenschappelijk te beschouwen.

We beschouwen een partiële *k-tree* $G = (V, E)$ en zij

$$S(G) = \{S \mid S \subseteq V, |S| = k+1, \forall C \in \mathit{components}(G[V(G) \setminus S]) : S_C \neq S\}.$$

Hierbij is

$$S_C = \{w \in S \mid \exists u \in V(C) : \{u, w\} \in E(G)\}$$

Merk op dat $S(G)$ bijgevolg bestaat uit $k+1$-sets van knopen die de graaf in componenten verdelen. Het is in dit geval echter wel toegestaan dat de graaf in slechts één component verdeeld wordt. Hierbovenop geldt ook nog eens dat er voor elke component minstens één knoop uit deze verzameling moet zijn die los van de betreffende component staat. Dit geeft aan dat er een k-subset is die de graaf in minstens twee componenten verdeelt, dus er zitten *k-vertex separators* in de $k+1$ separators verborgen.

Het idee is als volgt: We construeren voor elke mogelijke $S \in S(G)$ een tree-decompositie met S als wortel-*bag* en nemen de minimale. De uiteindelijke canonische vorm wordt gepresenteerd als een soort graafexpressie, maar dan ongelabeld en ongericht. Om te kunnen bepalen welke de minimale is, hebben we nood aan de volgende verzameling:

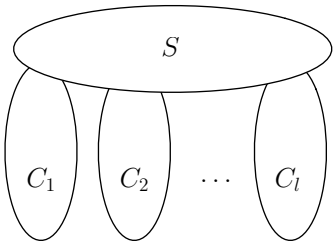
$$D(G) = \{(S, \phi) \mid S \in S(G) \text{ en } \phi : S \to \{1, \ldots, k+1\}\}$$

$D(G)$ geeft ons met andere woorden alle mogelijke koppels die bestaan uit een "separator" en een ordening op zijn knopen. Deze ordening op de knopen kan ook gezien worden als het associëren van de betreffende knoop met een zekere source uit de graafexpressie. We definiëren vervolgens de canonische vorm van G op de volgende manier:

$$\mathit{canon}(G) = \min_{(S,\phi) \in D(G)} \mathit{canon}(G, S, \phi).$$

Hierbij definiëren we de functie $\mathit{canon}(G, S, \phi)$ op de volgende recursieve wijze:

geval 1 $|V(G)| \leq k+1$. Dit wil zggen dat S alle knopen van de graaf G bevat. Bijgevolg geeft ϕ een orde op alle knopen van de graaf en kunnen we



Figuur 8.3: Canonische vorm berekenen.

een expressie op de volgende manier opstellen:

$$canon(G, S, \phi) = \underbrace{|_{\forall\{v,w\}\in E(G)}\ \{\phi(v), \phi(w)\}}_{\text{in lexicografische volgorde}}.$$

De verschillende bogen worden nu geordend volgens de lexicografische orde. De toewijzing van getallen aan de knopen door ϕ zorgt dat elke boog een vaste plaats krijgt in deze orde. We mergen met andere woorden de verschillende bogen en gebruiken de geassocieerde sources, die gegeven worden door ϕ, om de bogen tussen de knopen te leggen.

geval 2 $|V(G)| > k+1$. De sources die aan de knopen van S gekoppeld worden door ϕ laten toe om al een expressie te construeren die S voorstelt:

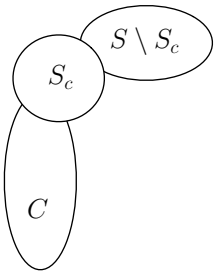
$$\underbrace{|_{\forall\{v,w\}\in E(G[S])}\ \{\phi(v), \phi(w)\}}_{\text{in lexicografische volgorde}}$$

Nu rest ons nog om voor de rest van de graaf een canonische voorstelling te berekenen. We weten dat $S\,G$ opsplitst in allemaal verschillende componenten, deze situatie wordt geschetst in Figuur 8.3. Voor iedere component C hebben we nu minstens één knoop in S die volledig losstaat van C (niet rechtstreeks geconnecteerd), zoals we zien in Figuur 8.4. Hierbij is $S \setminus S_c$ de verzameling knopen die niets met de component te maken hebben.

We gaan nu voor elk van deze componenten afzonderlijk een canonische vorm berekenen. Dit gebeurt door de knoop die niets met de component te maken heeft weg te laten, en de source die we hiervoor gebruikt hebben te gebruiken voor een nieuwe knoop uit de component.

Noem nu $G(i) = S_c \cup C$, zoals weergegeven in Figuur 8.4. Deze graaf is duidelijk kleiner dan G. We gaan voor de bekomen graaf weer hetzelfde doen: voor alle mogelijke $k+1$ verzamelingen die de graaf in componenten delen die minstens losstaan van één knoop gaan we de canonische vorm volgens deze verzameling berekenen:

$$\min_{\forall(S_i,\psi_i)\in D(G(i))} (\mathrm{lc}\{C \setminus S_C\}) canon(G(i), S_i, \psi_i).$$



Figuur 8.4: Een component nader bekeken.

Maar we willen natuurlijk dat we de mapping die we aan de knopen van S_c gegeven hebben behouden, enkel de mapping van de andere knopen mag hergebruikt worden. Anders is het namelijk niet altijd mogelijk om de component te mergen met de separator om de gewenste graaf te bekomen. Er moet dus gelden dat :

1. $\forall v \in S_C : \psi(v) = \phi(v)$, en
2. $S_C \subseteq S$,

waarbij ψ de nieuwe mapping is. Merk op dat in de formule de knopen die we uit C kiezen gelocald worden, zodat we de sources kunnen hergebruiken voor het eerste deel van onze constructie: S. Indien we alles samenvoegen bekomen we hetvolgende:

$$canon(G, S, \phi) = \underbrace{|_{\forall \{v,w\} \in E(G[S])} \{\phi(v), \phi(w)\}}_{\text{in lexicografische volgorde}} \underbrace{|_{\forall C_i} \min_{\forall (S_i, \psi_i) \in D(G(i))} (\mathrm{lc}\, C_i \setminus S_{C_i}) canon(G(i), S_i, \psi_i)}_{\text{in lexicografische volgorde}} .$$

Wat er gebeurt is dat we eerst en vooral een voorstelling berekenen voor de verzameling S en vervolgens een minimale voorstelling berekenen voor de rest van de graaf, door één knoop los te scheuren. We weten namelijk dat voor elke component in de graaf G waarmee we `canon` aanroepen, er een knoop is die volledig los staat van de betreffende component. We mogen dus deze knoop weglaten, een volgende uit de component zelf in de plaats nemen en hier de canonische representatie van berekenen. De mapping zal deze nieuwe knoop noodzakelijkerwijs afbeelden op het getal waar de verwijderde knoop op werd afgebeeld. Als we vervolgens deze knoop "local" houden, dan kunnen we een voorstelling voor de graaf bekomen, zonder dat ze iets met elkaar te maken hebben. Op deze manier komen we altijd toe met onze $k+1$ sources. Dit is rechtstreeks te zien wanneer we *k-trees* beschouwen: er wordt altijd een knoop verbonden met een reeds aanwezige *k-clique*, de andere

knopen hebben niets met deze nieuwe te maken. Ook in de genormaliseerde tree-decomposities is dit duidelijk: we voegen telkens een nieuwe knoop toe en er gaat eentje weg.

Voorgaand betoog kan verwerkt worden in een polynomiaal algoritme zoals we gedaan hebben met het graafisomorfisme probleem. Een voorbeeld van hoe een canonische graafexpressie te berekenen volgens deze procedure is gegeven in Bijlage F. We merken verder op dat de gepresenteerde denkwijze ons eigenlijk een string oplevert en bijgevolg een full-invariant algoritme is. We hebben echter gezien in Hoofdstuk 2 dat we hieruit een canonisatie-algoritme kunnen bekomen. Deze constructie kon een polynomiaal invariant algoritme omzetten in een polynomiaal canonisatie algoritme, net hetgeen waarnaar we op zoek zijn.

8.3 Conclusie voor graafexpressies

We hebben aangetoond dat we isomorfie van partiële *k-trees* kunnen beslissen in polynimiale tijd, wanneer we weten dat de *treewidth* van beide grafen begrensd is. We weten ook dat grafen die door eenzelfde graafgrammatica gegenereerd worden een begrensde *treewidth* hebben, en kunnen bijgevolg de grafen berekenen die voorgesteld wordt door de expressies (door ze te evalueren) en vervolgens het algoritme van Bodlaender laten lopen hierop. Indien de grafen isomorf zijn, weten we ook dat onze expressies isomorf zijn, dit hebben we namelijk zo gedefinieerd.

Hetzelfde verhaal gaat op voor de canonisatieprocedure: bereken een graaf en stel de canonische graafexpressie op die de graaf voorstelt. Op deze manier hebben we een canonische vorm van expressies geconstrueerd. Een minpunt is echter dat deze canonische vorm niet noodzakelijk gegenereerd wordt door een beschouwde graafgrammatica en we dus niet enkel de canonische expressies moeten produceren.

Hoofdstuk 9

Enumeratie van graafgrammatica's

Tot hiertoe hebben we enkele indicaties gegeven van mogelijke paden die we kunnen volgen om algoritmen te ontwerpen die alle grafen van een graafgrammatica genereren. De problemen en bijhorende oplossingen die we zijn tegengekomen zijn de volgende:

- Om te kijken of een gegeven graafexpressie aanvaard wordt door een reguliere boomgrammatica, is het mogelijk om een *boomautomaat* te gebruiken.
- Het *Parsing Probleem* dat werd aangehaald in hoofdstuk 7 houdt in dat we niet dadelijk inzien hoe een graaf omgezet kan worden naar een graafexpressie die door een grammatica gegenereerd kan worden. Het is immers mogelijk dat een deel van de expressies die de graaf voorstellen gegenereerd wordt en een deel niet. Een overzicht van parsing aanpakken wordt gegeven door Schürr [27].
- Bij het genereren van graafexpressies willen we natuurlijk zeker zijn dat we elke expressie slechts één maal genereren. Dit kan door gebruik te maken van een boomautomaat die *deterministisch* gemaakt is.
- De notie van isomorfie tussen graafexpressies werd behandeld door een parallellisme met partiële *k-trees* aan te tonen. Hierdoor hebben we ingezien dat er een efficiënt (polynomiaal) algoritme bestaat wanneer we grafen van een graafgrammatica beschouwen. Dit komt omdat de *treewidth* van deze grafen begrensd is en we k als constante kunnen zien. Het gegeven algoritme buit de eigenschap uit door het probleem op te splitsen en maakt gebruik van dynamisch programmeren.
- Een ander efficiënt algoritme dat we gevonden hebben is een canonisatiealgoritme voor partiële *k-trees*. Ook dit kan in polynomiale tijd.

Een probleem is dat we echter niet weten of een canonische expressie gegenereerd wordt door een gegeven grammatica.

In dit hoofdstuk presenteren we de mogelijke aanpakken om algoritmen te construeren met behulp van de tot hiertoe opgedane kennis. Naast de volgende technieken kunnen we ook de aanpak beschouwen die in het begin van Hoofdstuk 7 gepresenteerd werd en in Algoritme 16 weergegeven werd.

9.1 De klassieke methode

Als eerste is er natuurlijk de "klassieke" methode, die inhoudt dat we alle grafen bijhouden en voor elke nieuw gegenereerde graaf testen of hij isomorf is met eentje uit de lijst. Algoritme 20 beschrijft hoe dit gebeurt. Merk op dat we een methode nodig hebben die alle mogelijke expressies één voor één genereert, maar deze is voorhanden, zoals hierboven aangegeven.

Een zeer groot nadeel is dat we alle grafen dienen bij te houden, en de geheugenvereisten dus zeer groot zijn. Een goede zaak is dat we voor de isomorfietest gebruik kunnen maken van de isomorfie-test op partiële *k-trees*: deze kan in polynomiale tijd uitgevoerd worden. We zijn immers reeds zeker van een begrendsde *treewidth* van de grafen. Maar de isomorfie-check dient met *alle* voorgaande grafen te gebeuren en deze groep wordt enkel groter.

Het is wenselijk dat de zoekruimte van de grafen *levelwise* ontdekt wordt. Hiermee bedoelen we in dit geval dat de expressies van klein naar groot gegenereerd worden. Stel dat we bijvoorbeeld een groeimethode hebben en we telkens een gegeven expressie blijven uitbreiden tot in het oneindige, dan kunnen bepaalde expressies nooit gegenereerd worden.

Algoritme 20: De klassieke methode

```
1 foreach e ∈ L(G_r) do
2 |   foreach reeds gegenereerde graaf G do
3 |   |   if not ‖e‖ ≅ G then
4 |   |   |   output ‖e‖;
5 |   |   end
6 |   end
7 end
```

9.2 Alle partiële k-*trees* gebruiken

9.2.1 Het generisch algoritme

Laten we eerst eens kijken hoe we de klasse van alle partiële *k-trees* op n knopen kunnen genereren met behulp van het orderly algoritme. We kunnen dit inefficiënt aanpakken en alle grafen genereren, die vervolgens

getest worden met het algoritme van Arnborg of ze wel een partiële *k-tree* zijn. Zoals reeds vermeld in hoofdstuk 6 worden er veel onnodige grafen gegenereerd. Is het zijn van een partiële *k-tree* misschien *augmentation hereditary*-eigenschap voor een bepaalde groeioperatie? Dit blijkt inderdaad zo te zijn, als we de generische vector-methode gebruiken, dan zijn alle parents (met één boog minder) inderdaad partiële *k-trees* en bijgevolg zeker de kleinste parent! Op deze manier hebben we een methode geconstrueerd die alle partiële *k-trees* genereert op n knopen. We kunnen het algoritme aanroepen met verschillende waarden van n om zo alle partiële *k-trees* te enumereren.

Het is eenvoudig in te zien dat geen enkele graaf dubbel gegenereerd zal worden, omdat het orderly algoritme hiervoor zorgt. Het is echter niet meteen duidelijk hoe we een dergelijke *recognizer* voor onze grammatica kunnen schrijven, daar de omzetting van een graaf naar graafexpressie niet gedefinieerd is. Met andere woorden we blijven nog steeds achter met het parsing probleem.

Algoritme 21 werkt duidelijk op dezelfde manier als Algoritme 16.

Algoritme 21: Enumereren van grafen beschreven door een graafgrammatica

```
1 foreach n ∈ {1, 2, ..., k} do
2 |   foreach k-tree K met o(K) = n do Orderly algoritme gebruiken
3 |   |   if G_r genereert K then
4 |   |   |   output K;
5 |   |   end
6 |   end
7 end
```

9.2.2 Een specifiek orderly algoritme

Merk op dat we in hoofstuk 8 een methode ontwikkeld hebben om een canonische vorm te berekenen van partiële *k-trees* in polynomiale tijd. Indien we een uitbreidingsoperatie kunnen vinden die hiermee samenwerkt en aan de eisen van het orderly algoritme voldoet, dan is het mogelijk om een zeer efficiënt algoritme te construeren. Wanneer we het generisch algoritme gebruiken, dan wordt er namelijk ook een generisch canonisatie-algoritme gebruikt, terwijl we met een specifiek algoritme de eigenschappen van partiële *k-trees* kunnen uitbuiten.

9.3 Enkel de gewenste grafen

Indien we nog minder ongewenste grafen willen genereren, dan is het nodig dat we een graafgrammatica kunnen omvormen naar een soort groeioperatie,

die enkel de toegelaten grafen van de grammatica produceert. Indien de groeioperatie aan de nodige voorwaarden voldoet, dan is het mogelijk om ze samen met een orderly algoritme of het algoritme van McKay te gebruiken. Dit is de gewenste aanpak: alle objecten genereren, door zo weinig mogelijk onnodige grafen te beschouwen. De algoritmen zorgen er vervolgens zelf voor dat we geen last hebben van isomorfe kopiëen.

Hoofdstuk 10

Samenvatting en toekomstig werk

In deze thesis hebben we ons toegespitst op het enumereren van grafen op een isomorfvrije manier. We hebben eerst gezorgd dat we over de nodige kennis beschikten van het begrip isomorfie, door het op een algemene manier te bekijken. Hierna hebben we enkele algoritmen besproken:

- Het orderly algoritme, met in het bijzonder een generisch algoritme,
- gSpan, en
- het algoritme van McKay.

Het bleek dat gSpan een instantie was van het orderly algoritme. Verder hebben we, na het implementeren van een framework en een testprogramma, besproken wat de verschillen zijn in de aanpakken, en wat de goede en slechte kanten zijn van de algoritmen.

Het volgende deel bestond eruit om te kijken naar klassen van grafen met een bepaalde eigenschap. We hebben gekozen om klassen te beschrijven aan de hand van graafgrammatica's en hebben de intuïtie hierachter onderzocht. Graafexpressies uit een graafgrammatica bevatten altijd een begrensd aantal sources. Dit laatste hebben we in verband gebracht met de notie van *treewidth*, waar reeds een hele theorie rond bestaat. Op deze manier waren we in staat om efficiënte algoritmen te beschouwen die ook nuttig zijn voor graafexpressies, zoals het testen of twee expressies isomorf zijn.

Tot slot hebben we aan de hand van de beschreven theorie nog een overzicht gegeven van mogelijke algoritmen die ons toelaten om de graafexpressies die gegenereerd worden door een graafgrammatica op een isomorfvrije manier op te sommen. Ook hebben we mogelijke problemen angehaald die hierbij komen kijken.

Het is duidelijk dat deze technieken nog verder onderzocht en uitgediept moeten worden en dat dit onderwerp verre van uitgeput is. De bedoeling is

dan ook om verder te borduren op het werk dat in deze thesis gepresenteerd werd, om zo performante algoritmen te bekomen die bruikbaar zijn in de praktijk.

Bijlage A

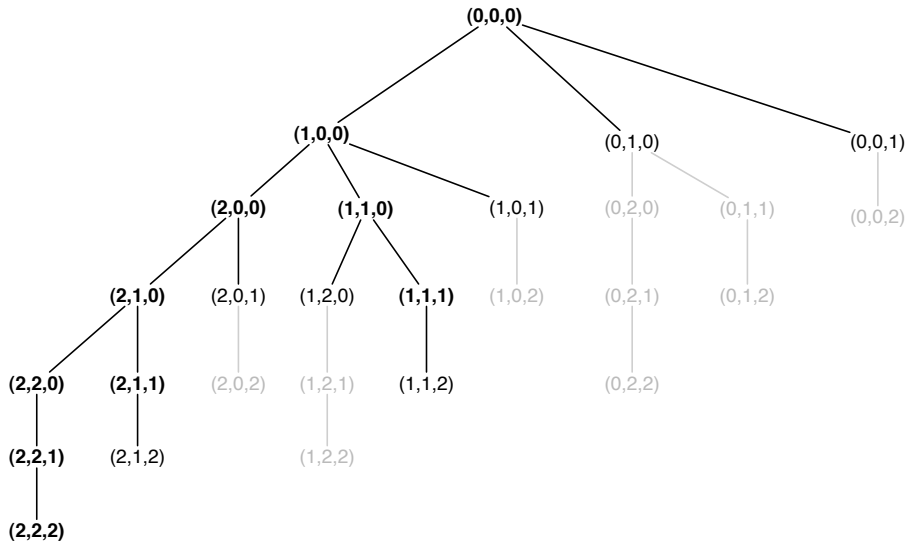
Uitwerking van Orderly

We zullen nu alle grafen genereren die uit drie knopen bestaan, en twee mogelijke labels kunnen hebben. We volgen de aanpak die in sectie 3.2.1 gevolgd werd voor het genereren van grafen. In ons geval is $k = 2$ en $n = 3$, we hebben met andere woorden een vector met drie componenten die elk waarden uit $\{0, 1, 2\}$ kunnen aannemen. Hierbij stelt de waarde 0 "geen boog" voor. De uitwerking van het algoritme voor algemene combinatorische objecten, toegepast op dit soort grafen, is gegeven in Figuur A.1. De vette knopen in de boom stellen de canonische vectoren voor, de lichtgrijze worden nooit gegenereerd. Merk op dat op elke "laag" van de boom de som van de elementen in de vector voor elk object hetzelfde is.

Om te zien hoe we de canonische vorm van een vector moeten berekenen moeten we, zoals in sectie 3.2.1 vermeld, de permutaties op de componenten van de vector beschouwen, die geïnduceerd worden door de permutaties op de knopen. In dit geval werken we met drie knopen, hetgeen ons zes permutaties oplevert. In Tabel A.1 zien we in de eerste kolom de permutaties die mogelijk zijn op de knopen. De volgende drie kolommen laten ons zien welk effect dit heeft op de elementen van de bogenverzameling, met andere woorden: welke boog wordt afgebeeld op welke. De laatste kolom zegt ons wat de overeenkomstige permutatie van de elementen van de vector is.

Omdat de vectorvoorstelling van een graaf de drie elementen van het bovenste triangulaire deel van de overeenkomstige matrix bevat, weten we dat deze elementen overeenkomen met de drie mogelijke bogen. We weten nu ook welke bogen op welke afgebeeld worden en kunnen op deze manier een permutatie van de vector construeren, die geïnduceerd wordt door eentje op de knopen van de graaf.

Om tot een maximale voorstelling van een vector te komen volstaat in dit geval het om alle toegelaten permutaties van de componenten af te gaan, in dit geval zijn het alle mogelijke. Uit deze verzameling permutaties kiezen we de vector die de hoogste getalvoorstelling heeft: we plakken alle cijfers achter elkaar en nemen dat getal. Op deze manier kunnen we bepalen welke vector



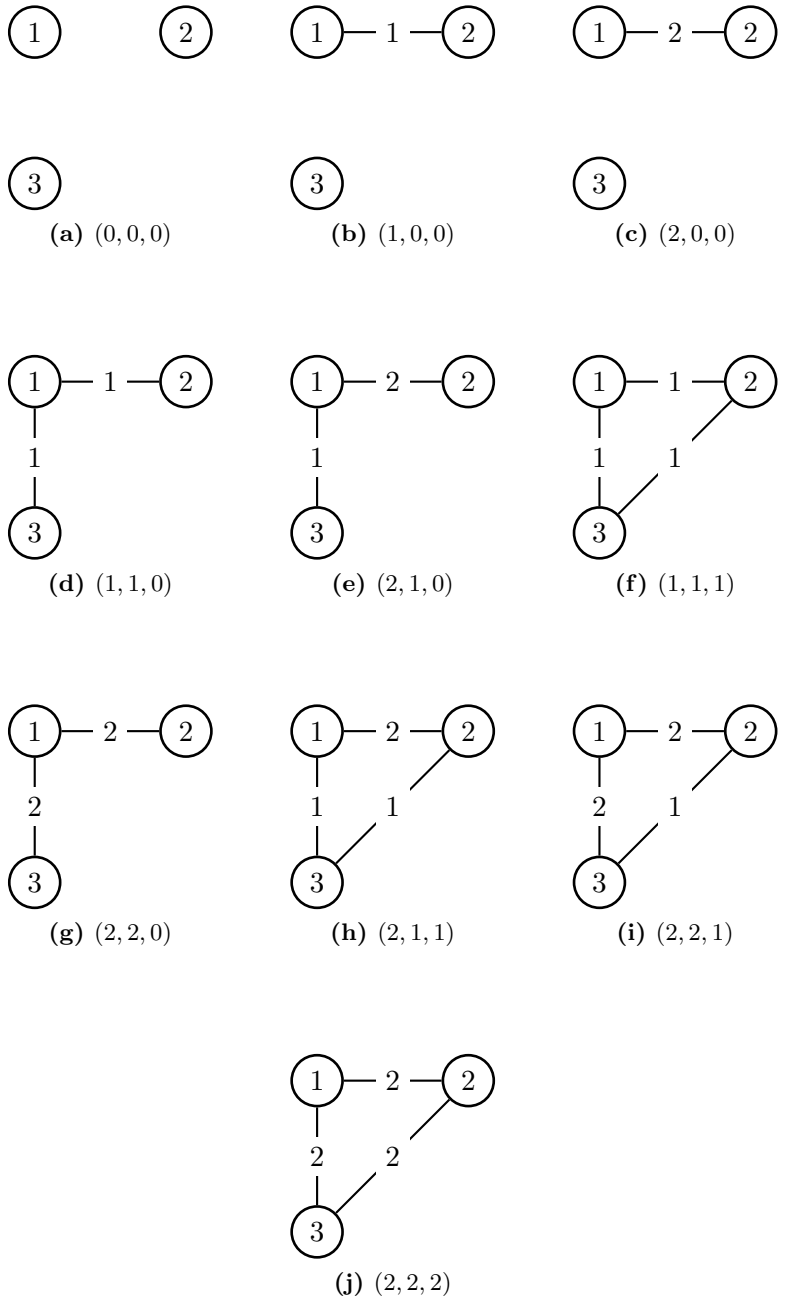
Figuur A.1: De zoekruimte van het algoritme, voor $n = 3$ en $k = 3$. De vette knopen stellen de canonische vectoren voor.

	$v_1 = \{1,2\}$	$v_2 = \{1,3\}$	$v_3 = \{2,3\}$	
()	$\{1,2\}$	$\{1,3\}$	$\{2,3\}$	()
(1 2)	$\{1,2\}$	$\{2,3\}$	$\{1,3\}$	$(v_2\ v_3)$
(1 3)	$\{2,3\}$	$\{1,3\}$	$\{1,2\}$	$(v_1\ v_3)$
(2 3)	$\{1,3\}$	$\{1,2\}$	$\{2,3\}$	$(v_2\ v_3)$
(1 2 3)	$\{2,3\}$	$\{1,2\}$	$\{1,3\}$	$(v_1\ v_3\ v_2)$
(1 3 2)	$\{1,3\}$	$\{2,3\}$	$\{1,2\}$	$(v_1\ v_2\ v_3)$

Tabel A.1: De permutaties van de posities van de vector, geïnduceerd door de permutaties van de kopen.

canonisch is, en welke niet, en kunnen we in de boom prunen. Herinner dat we de vectoren genereren van groot naar klein: als we terug een grotere vector tegenkomen dan degene die laatst toegevoegd werd op dat niveau, dan mag deze niet toegevoegd worden. Het is eenvoudig in te zien dat het de getallen in de vector altijd van groot naar klein moeten staan, omdat er anders wel een permutatie bestaat (we mogen ze allemaal beschouwen!) die dit kan opleveren. We merken op dat in het algemeen niet alle permutaties op de bogen geldig zullen zijn, omdat dit er normaal gezien meer zijn dat het aantal mogelijke permutaties van de knopen.

Tot slot zijn alle grafen die gegenereerd worden door het algoritme weergegeven in Figuur A.2. Merk op dat de getallen die in de knopen staan geen labels zijn, maar de knooporde weergeven.



Figuur A.2: De grafen gegenereerd door het orderly algoritme.

Bijlage B

Uitwerking van McKay

We werken het algoritme van McKay uit voor grafen, zoals gedefinieerd is sectie 3.4. We volgen het algoritme niet helemaal, maar proberen de gedachte erachter te illustreren. We beginnen met het onreduceerbaar object K_1: Het is duidelijk dat er slechts één orde op de knopen bestaat voor

Figuur B.1: Generatie 0.

deze graaf en dat hij dus het canonische object is. Als we de uitbreidingen van deze graaf berekenen bekomen we de twee grafen die te zien zijn in Figuur B.2 De gekleurde knopen stellen degenen voor die in de orbit van de

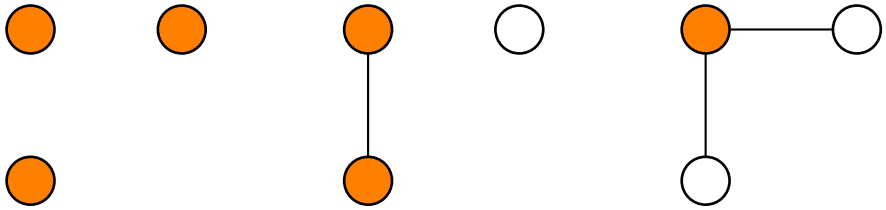
(a) aanvaard **(b)** aanvaard

Figuur B.2: Grafen uit generatie 1.

knoop liggen die eerst komt in de canonische orde. We werken met maximale orde, waarmee we bedoelen dat we de maximale string willen bekomen door het achter elkaar kleven van de rijen van de matrix. Om te weten welke graaf deze graaf *mag* genereren, moeten we dus één van deze gemarkeerde knopen verwijderen.

De graaf zonder boog levert ons enkel 0'en op in de matrix en is dus duidelijk de minimale voorstelling die uitgeprint mag worden. De graaf met twee knopen en één boog heeft ook maar één mogelijke matrix en is bijgevolg

ook canonisch. We gaan verder met de graaf uit Figuur B.2a. De mogelijke kinderen worden weergegeven in Figuur B.3. De eerste graaf heeft elk van zijn knopen gemarkeerd. De graaf die deze moet genereren is dus de graaf die uit twee losse knopen bestaat, hetgeen in dit geval waar is, waardoor de graaf aanvaard wordt. De tweede graaf wordt ook aanvaardt, de verbonden knopen zijn degenen die eerst komen in de maximale knooporde. Wanneer we één van deze wegdoen bekomen we de graaf die uit twee knopen bestaat, zonder bogen. Het is inderdaad zo dat deze graaf de nieuwe gegenereerd heeft, met als gevolg dat deze laatste aanvaard wordt. De derde graaf heeft slechts één gemarkeerde knoop, deze heeft de hoogste graad en dient eerst te komen in de orde, willen we de orde maximaal houden. De genererende graaf dient ook hier te bestaan uit twee losse knopen.

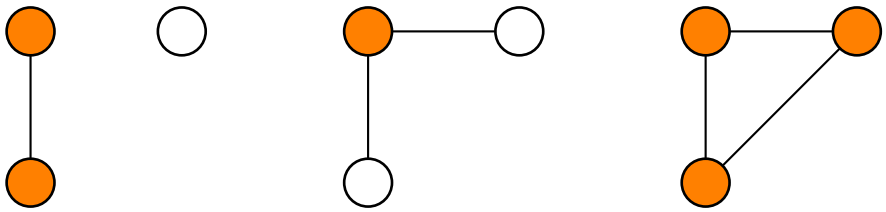


(a) aanvaard **(b)** aanvaard **(c)** aanvaard

Figuur B.3: De eerste grafen uit generatie 2.

We gaan nu verder met de grafen die gegenereerd worden door de graaf in Figuur B.2b. De graaf in Figuur B.4a dient gegenereerd te worden door de graaf met twee losse knopen, dit is echter niet het geval waardoor deze geweigerd wordt. Eenzelfde redenering gaat op voor de graaf uit Figuur B.4b. Deze weigeringen zijn duidelijk terecht, de grafen zijn immers al gegenereerd! Tot slot beschouwen we nog de graaf uit Figuur B.4c, deze heeft slechts één orbit van knopen. De graaf die deze moet genereren is degene die bestaat uit twee knopen die verbonden zijn met één boog, aan deze voorwaarde is duidelijk voldaan.

We hebben nu alle grafen gegenereerd tot en met orde 3, zonder dubbels. We kunnen elk van de verkregen objecten nog eens uitbreiden om alle grafen van orde 4 te bekomen. Merk op dat we in essentie de ongeordende grafen beschouwd hebben, het algoritme vult dit in door voor een geordende graaf de gepaste geordende ouder te berekenen.



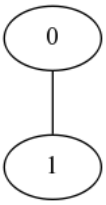
(a) geweigerd **(b)** geweigerd **(c)** aanvaard

Figuur B.4: Generatie 2: vervolg.

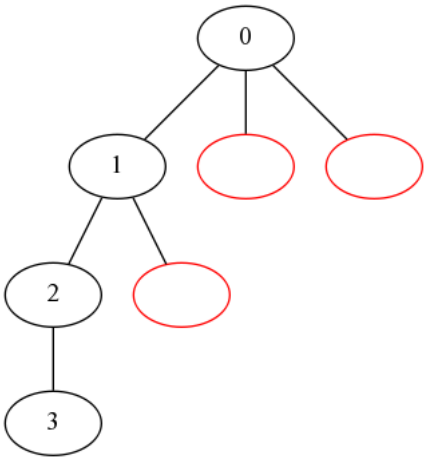
Bijlage C

Zoekruimtes

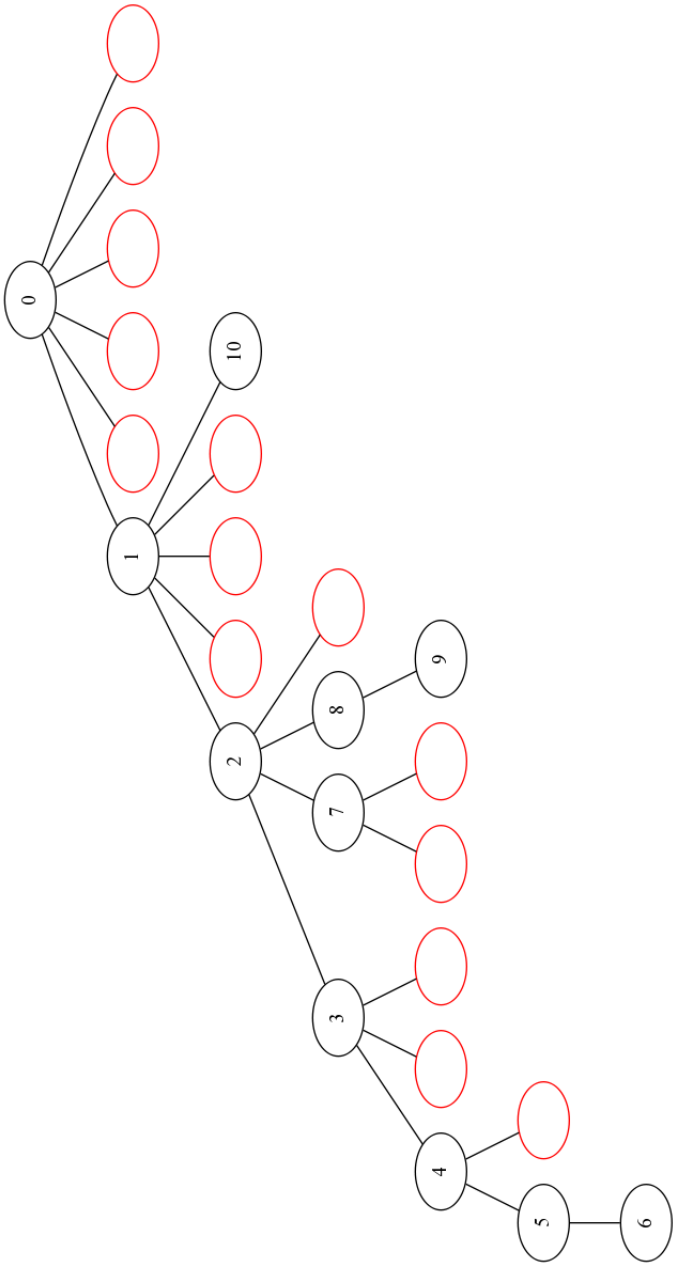
We presenteren enkele afbeeldingen van zoekruimtes die gegenereerd werden door GraphGen in combinatie met GraphViz [1]. Elke knoop komt overeen met een graaf en zijn ouder is de graaf die hem "gegenereerd" heeft. De knopen zonder nummer zijn de niet-canonische grafen, hier stopt de zoektocht. Merk hierbij wel op dat de zoekbomen voor orderly enkel over grafen met n knopen gaan, en die van McKay over grafen kleiner of gelijk aan n knopen.



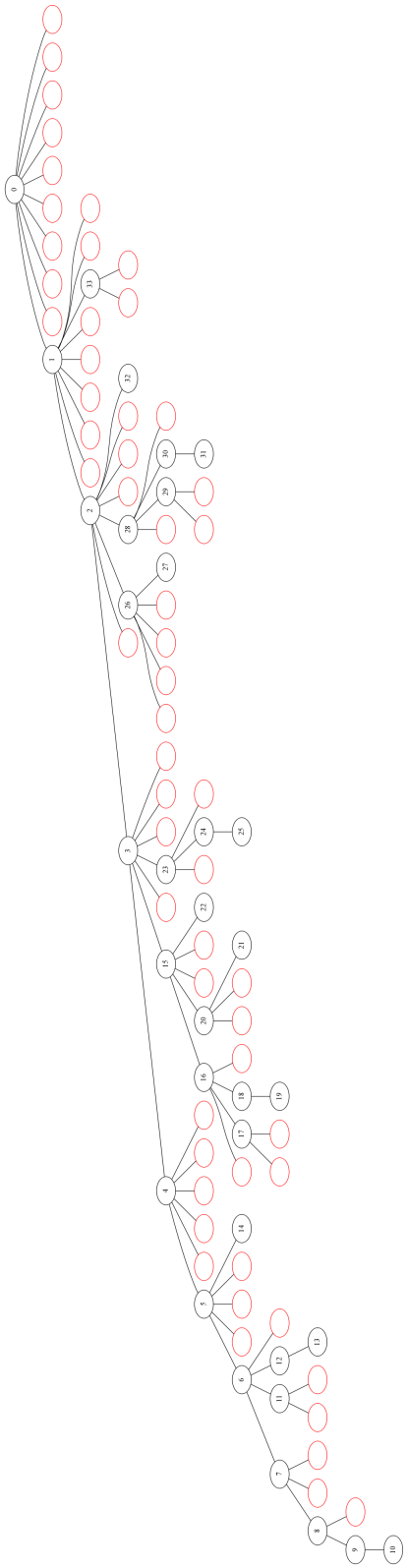
Figuur C.1: Generic orderly zoekruimte voor $n = 2$



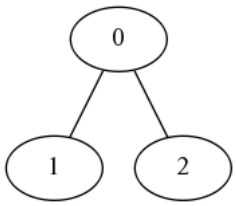
Figuur C.2: Generic orderly zoekruimte voor $n = 3$



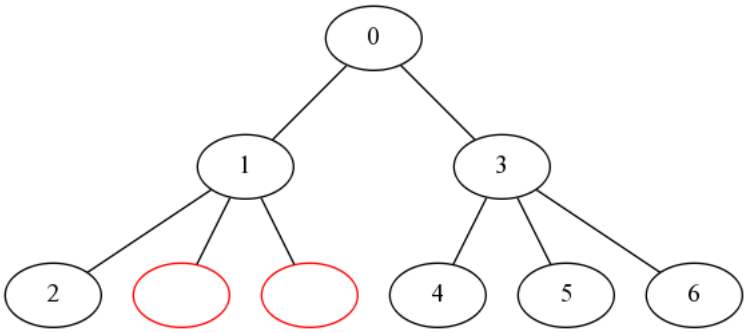
Figuur C.3: Generic orderly zoekruimte voor $n = 4$



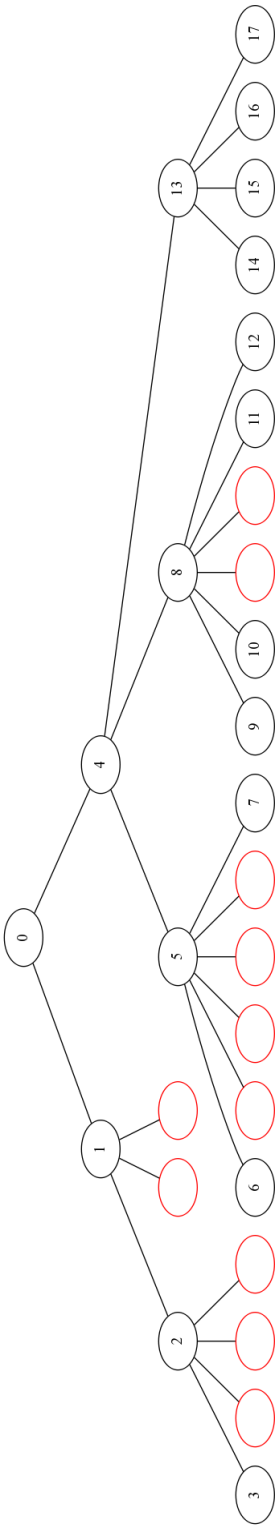
Figuur C.4: Generic orderly zoekruimte voor $n = 5$



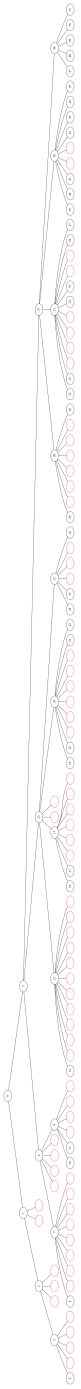
Figuur C.5: McKay zoekruimte voor $n = 2$



Figuur C.6: McKay zoekruimte voor $n = 3$



Figuur C.7: McKay zoekruimte voor $n = 4$



Figuur C.8: McKay zoekruimte voor $n = 5$

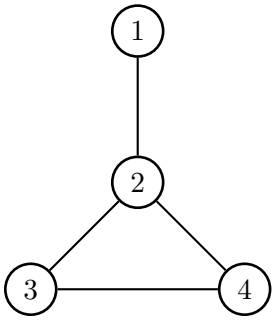
Bijlage D

Uitwerking van partiële k-*tree*-test

We geven de uitwerking van het partiële k-*tree*-algortitme van Arnborg op enkele grafen.

D.1 Run 1

Beschouw de volgende graaf:

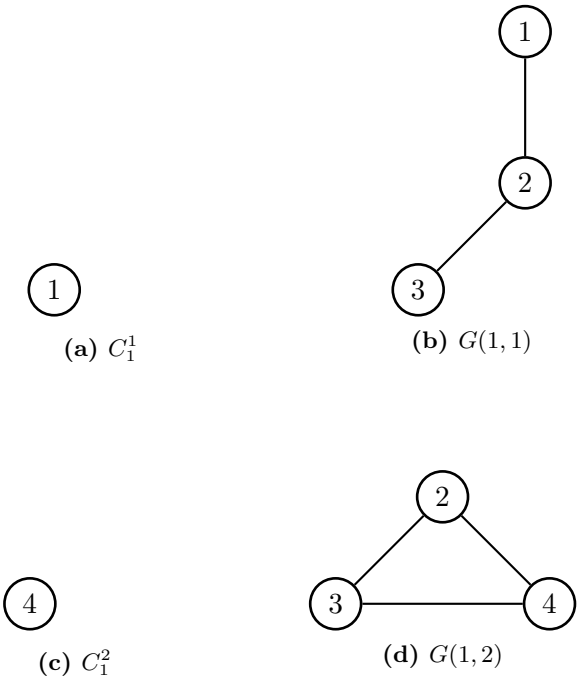


Figuur D.1: Een graaf op 4 knopen.

We willen testen of deze graaf een partiële 2-*tree* is. Onze eerste taak is bijgevolg om alle 2-separators te identificeren. Na alle deelverzamelingen van twee knopen afgelopen te hebben vinden we:

$$C_1 = \{2, 3\}$$
$$C_2 = \{2, 4\}$$

De bijhorende componenten en geaugmenteerde componenten worden gegeven in figuren D.2 en D.3. Al de geaugmenteerde componenten hebben een grootte van $k + 1 = 3$, waardoor ze triviaal partiële k-*trees* zijn. Bijgevolg



(a) C_1^1 **(b)** $G(1,1)$

(c) C_1^2 **(d)** $G(1,2)$

Figuur D.2: Componenten van C_1.

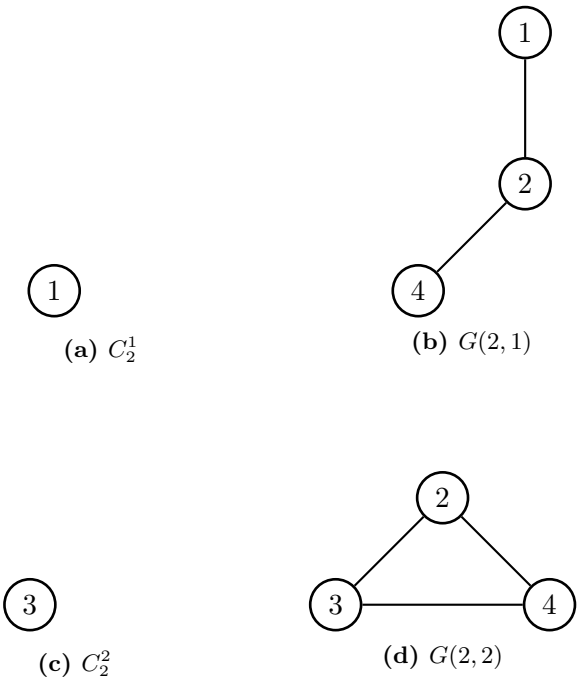
geldt voor elke separator dat de geassocieerde geaugmenteerde componenten allen partiële *k-trees* zijn en is de gehele graaf dus ook een partiële *k-tree*.

D.2 Run 2

Beschouw de graaf in Figuur D.4. We proberen ook hier aan te tonen dat we met een partiële 2-*tree* te maken hebben en beginnen eveneens met het zoeken van alle separators:

$$C_1 = \{2, 3\}$$
$$C_2 = \{2, 4\}$$
$$C_3 = \{2, 5\}$$
$$C_4 = \{3, 4\}$$

De componenten C_1^1, C_2^1, C_2^2, C_2^3, C_3^1 en C_4^1 bestaan uit slechts één knoop, hetgeen maakt dat de bijhorende geaugmenteerde componenten bestaan uit drie knopen, hetgeen hun triviale partiële *k-trees* maakt. De componenten C_1^2, C_3^2 en C_4^2 bestaan uit twee knopen. Merk op dat *alle* geaugmenteerde componenten die door C_2 veroorzaakt worden triviale partiële *k-trees* zijn,

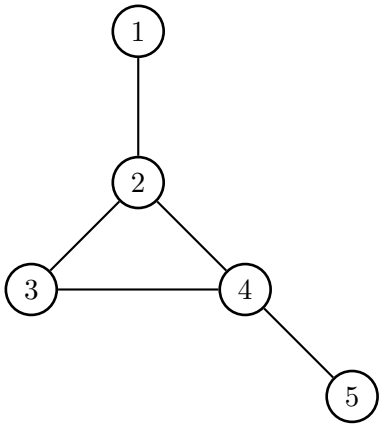


(a) C_2^1 **(b)** $G(2,1)$

(c) C_2^2 **(d)** $G(2,2)$

Figuur D.3: Componenten van C_2.

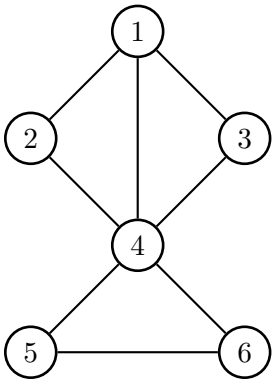
hetgeen er opnieuw voor zorgt dat de graaf een partiële *k-tree* is en we niet verder moeten zoeken.



Figuur D.4: Een graaf op 5 knopen.

D.3 Run 3

Als laatste voorbeeld nemen we de graaf uit Figuur D.5.



Figuur D.5: Een graaf op 6 knopen.

De separators worden gegeven door:

$$C_1 = \{1, 4\}$$
$$C_2 = \{2, 4\}$$
$$C_3 = \{3, 4\}$$
$$C_4 = \{4, 5\}$$
$$C_5 = \{4, 6\}$$

In Figuur D.6 worden de verschillende componenten weergegeven. Het resultaat na de triviale fase is

$$pkt(1,1) = \texttt{true}$$
$$pkt(1,2) = \texttt{true}$$
$$pkt(4,2) = \texttt{true}$$
$$pkt(5,2) = \texttt{true}$$

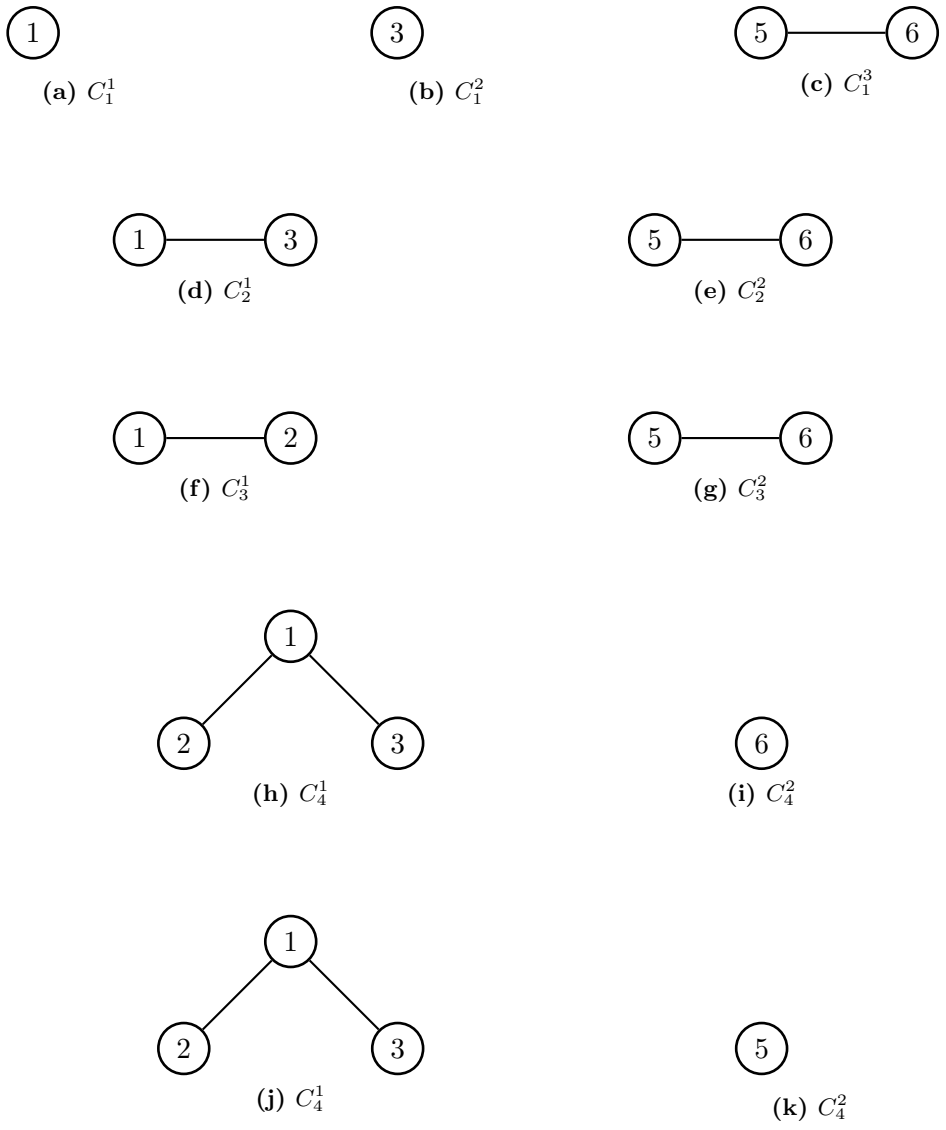
Vervolgens komen we in de niet-triviale fase waarin we bijvoorbeeld beginnen met $G(1,3)$[1], deze graaf is te zien in Figuur D.7. We moeten nu al zijn knopen afgaan. Herinner dat de knopen 1 en 4 geen zin zullen hebben, omdat ze deel uitmaken van de separator. We gaan dus verder met knoop 5. We dienen nu een separator C_m te kiezen zodat $C_m \subseteq C_1 \cup \{5\} = \{1,4,5\}$. enkel separator C_4 voldoet aan deze voorwaarde.

We zoeken nu naar alle $G(4,l)$'s die een subgraaf zijn van diegene gevormd door 5, 6 en 4 (de driehoek). De partiële *k-tree* (opzoeking, dynamisch programmeren!) $G(4,2)$ is net dezelfde graaf en "past" dus. Dit blijkt ook de enige te zijn. Nu moet er gelden dat C_1^3 een subgraaf is van de unie. Dit is het geval, de graaf die bestaat uit knopen 5 en 6 past inderdaad in de driehoek. We mogen dus hetvolgende doen:

$$pkt(1,3) = \texttt{true}.$$

We komen nu bij de tussentijdse controle en ontdekken dat alle componenten van de separator C_3 partiële *k-trees* zijn. Bijgevolg is de graaf een partiële *k-tree* voor $k = 2$ en geeft het algoritme `true` terug.

[1] We moeten de componenten wel in volgorde aflopen, maar deze is een minimale (buiten de triviale componenten).



(a) C_1^1 **(b)** C_1^2 **(c)** C_1^3

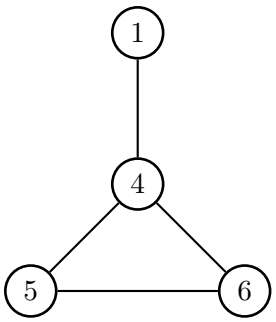
(d) C_2^1 **(e)** C_2^2

(f) C_3^1 **(g)** C_3^2

(h) C_4^1 **(i)** C_4^2

(j) C_4^1 **(k)** C_4^2

Figuur D.6: De verschillende componenten.

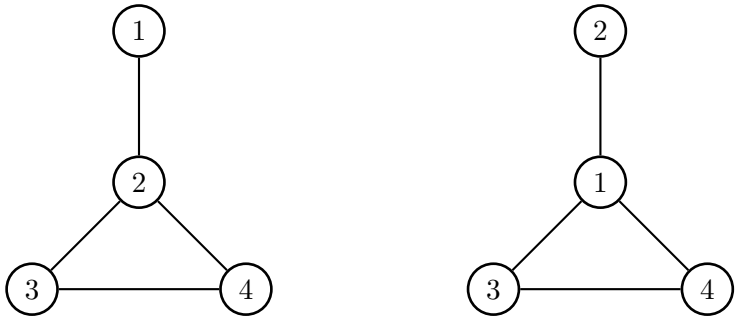


Figuur D.7: Een graaf op 4 knopen.

Bijlage E

Uitwerking van partiële k-*tree*-isomorfisme

We geven een kort voorbeeld van het testen van isomorfisme tussen twee grafen. Beschouw een graaf G en H:



(a) G **(b)** H

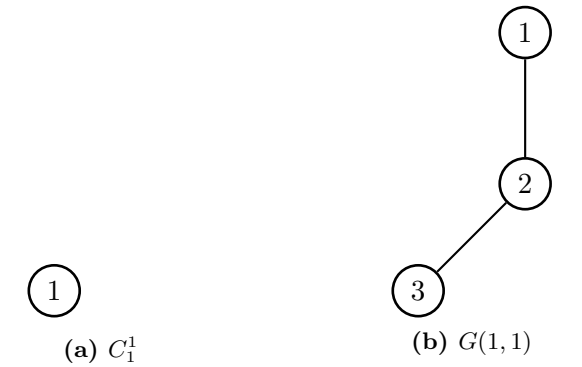
Figuur E.1: Twee isomorfe grafen op 4 knopen.

Stel dat we door een run van het algoritme van Arnborg op G de data in Figuur E.2 bekomen, tesamen met de separator $C_1 = \{2, 3\}$.

We gaan vervolgens voor H al de separators en bijhorende componenten berekenen.

$$D_1 = \{1, 4\}$$
$$D_2 = \{1, 3\}$$

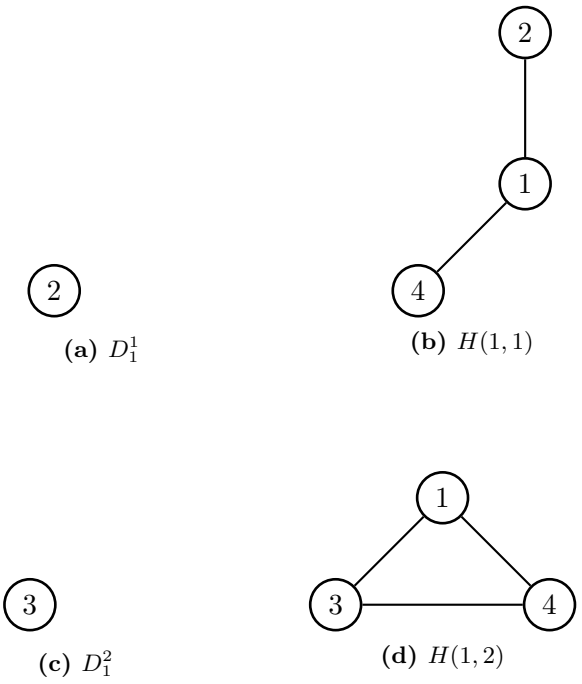
De volgende stap is om voor elk separator-component paar van H naar elk zulk paar van G te gaan kijken en voor elke mogelijke mapping van de



(a) C_1^1 **(b)** $G(1,1)$

(c) C_1^2 **(d)** $G(1,2)$

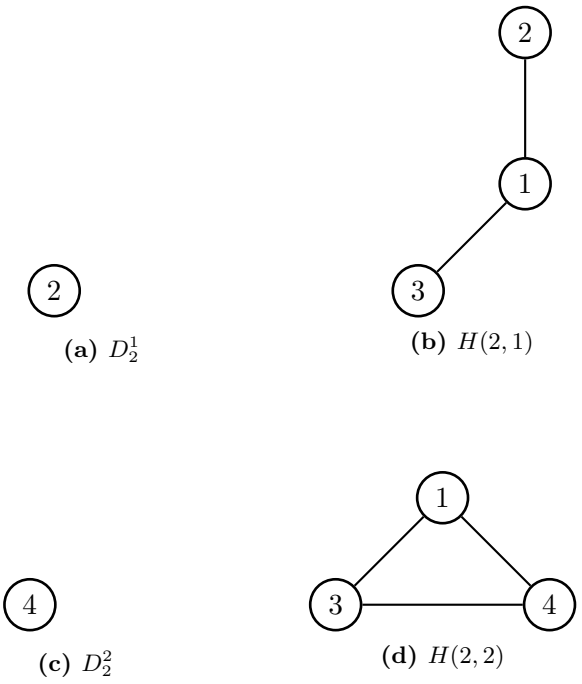
Figuur E.2: Componenten van C_1.



(a) D_1^1 **(b)** $H(1,1)$

(c) D_1^2 **(d)** $H(1,2)$

Figuur E.3: Componenten van D_1.



(a) D_2^1 **(b)** $H(2,1)$

(c) D_2^2 **(d)** $H(2,2)$

Figuur E.4: Componenten van D_2.

separator van G op die van H te controleren of ze f-isomorf zijn. We werken dit voor één geval uit, de andere gevallen zijn analoog.

Neem (D_1, D_1^1), D_1^1 bevat slechts één knoop, dus we mogen het speciale geval beschouwen. We kijken vervolgens naar (C_1, C_1^1) en beschouwen alle mappings $f : C_1 \to D_1$. Een voorbeeld van zo een mapping is

$$f_1 = \begin{cases} 2 \to 1 \\ 3 \to 4 \end{cases}$$

Er geldt nu dat $(D_1, D_1^1) \cong^f (C_1, C_1^1)$ omdat we een ψ kunnen constueren die de grafen op elkaar mapt:

$$\psi = \begin{cases} 2 \to 1 \\ 3 \to 4 \\ 1 \to 2 \end{cases}$$

Hieruit volgt dat:

$$iso(1, 1, 1, 1, f_1) = \texttt{true}$$

Een mapping f_2 die niet zou werken is

$$f_2 = \begin{cases} 2 \to 4 \\ 3 \to 1 \end{cases}$$

We hebben namelijk als enige ψ:

$$\psi = \begin{cases} 2 \to 4 \\ 3 \to 1 \\ 1 \to 2 \end{cases}$$

En de boog $\{1, 2\}$ zou op $\{2, 4\}$ gemapt worden, hetgeen geen boog is in H! Bijgevolg

$$iso(1, 1, 1, 1, f_2) = \texttt{false}$$

Op analoge wijze kunnen we onder andere de volgende entries bekomen:

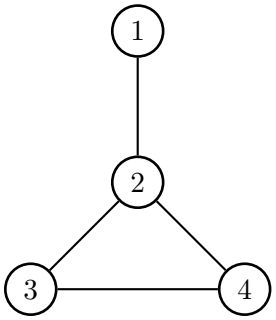
$$iso(1, 1, 1, 1, f_1) = \texttt{true}$$
$$iso(1, 2, 1, 2, f_1) = \texttt{true}$$
$$iso(1, 1, 2, 1, f_3) = \texttt{true}$$
$$iso(1, 2, 2, 2, f_3) = \texttt{true}$$

Op het einde wordt er een bipartite graaf opgebouwd per functie. Dit houdt in essentie in dat we gaan kijken of de betreffende functie uigebreid kan worden naar de verschillende componenten tot een isomorfisme. Indien dit bij elk van de componenten lukt, dan kunnen we een isomorfisme van de graaf G naar H construeren!

Bijlage F

Uitwerking van partiële k-*tree*-canonisatie

We geven de aanzet tot de berekening van de canonische vorm van een partiële 2-*tree*. Beschouw de graaf in Figuur F.1. $S(G)$ is hier gelijk aan alle



Figuur F.1: Een graaf G op 4 knopen.

verzamelingen van grootte drie, zodat de componenten die overblijven niet verbonden zijn met minstens één knoop uit de verzameling. Deze verzamelingen zijn:

$$S_1 : \{1, 2, 3\}$$
$$S_2 : \{1, 2, 4\}$$
$$S_2 : \{2, 3, 4\}$$

We gaan nu voor elk van deze verzamelingen, samen met elke mogelijke mapping van hun knopen op de verzameling $\{1, 2, 3\}$ de canonische vorm moeten bepalen. Hierna dienen we de minimale van de drie te nemen als canonische vorm van de graaf G.

F.1 S_1

We gaan van start met S_1 en de mapping

$$\phi_1 = \begin{cases} 1 \to 1 \\ 2 \to 2 \\ 3 \to 3 \end{cases}$$

Eerst bepalen we de string voor alle bogen uit S in lexicografische volgorde:

$$\{1,2\} \mid \{2,3\}.$$

Voor het volgende deel moeten we knoop 1 locallen, omdat deze niet verbonden is met knoop 4 (de knoop die we erbij gaan nemen, we hebben geen andere keuze). Onze enige optie is bijgevolg de graaf $G[\{2,3,4\}]$, met als enige mapping

$$\psi_1 = \begin{cases} 1 \to 4 \\ 2 \to 2 \\ 3 \to 3 \end{cases}$$

De expressie die we verkrijgen is $\{1,2\} \mid \{1,3\} \mid \{2,3\}$, hetgeen ons de totale expressie

$$\{1,2\} \mid \{2,3\} \mid (\mathrm{lc}\, 1)(\{1,2\} \mid \{1,3\} \mid \{2,3\})$$

oplevert.

We gaan vervolgens voor de volgende ϕ de expressie berekenen:

$$\phi_2 = \begin{cases} 1 \to 1 \\ 2 \to 3 \\ 3 \to 2 \end{cases}$$

Dit levert ons

$$\{1,3\} \mid \{2,3\}.$$

Het overige deel wordt $\{1,2\} \mid \{1,3\} \mid \{2,3\}$ en het totaal

$$\{1,3\} \mid \{2,3\} \mid (\mathrm{lc}\, 1)(\{1,2\} \mid \{1,3\} \mid \{2,3\})$$

Merk op dat het tweede deel altijd hetzelfde zal zijn.

$$\phi_3 = \begin{cases} 1 \to 2 \\ 2 \to 1 \\ 3 \to 3 \end{cases}$$

geeft ons

$$\{1,2\} \mid \{1,3\} \mid (\mathrm{lc}\, 2)(\{1,2\} \mid \{1,3\} \mid \{2,3\})$$

$$\phi_4 = \begin{cases} 1 \to 2 \\ 2 \to 3 \\ 3 \to 1 \end{cases}$$

geeft ons

$$\{1,3\} \mid \{2,3\} \mid (\mathrm{lc}\, 2)(\{1,2\} \mid \{1,3\} \mid \{2,3\})$$

$$\phi_5 = \begin{cases} 1 \to 3 \\ 2 \to 1 \\ 3 \to 2 \end{cases}$$

geeft ons

$$\{1,2\} \mid \{1,3\} \mid (\mathrm{lc}\, 3)(\{1,2\} \mid \{1,3\} \mid \{2,3\})$$

$$\phi_6 = \begin{cases} 1 \to 3 \\ 2 \to 2 \\ 3 \to 1 \end{cases}$$

geeft ons

$$\{1,2\} \mid \{2,3\} \mid (\mathrm{lc}\, 3)(\{1,2\} \mid \{1,3\} \mid \{2,3\})$$

De kleinste string tot hiertoe is

$$\{1,2\} \mid \{1,3\} \mid (\mathrm{lc}\, 2)(\{1,2\} \mid \{1,3\} \mid \{2,3\}).$$

F.2 S_2

$$\phi_1 = \begin{cases} 1 \to 1 \\ 2 \to 2 \\ 4 \to 3 \end{cases}$$

geeft ons

$$\{1,2\} \mid \{2,3\}$$

en

$$\{1,2\} \mid \{1,3\} \mid \{2,3\}$$

$$\{1,2\} \mid \{2,3\} \mid (\mathrm{lc}\, 1)(\{1,2\} \mid \{1,3\} \mid \{2,3\}).$$

$$\phi_2 = \begin{cases} 1 \to 1 \\ 2 \to 3 \\ 4 \to 2 \end{cases}$$

geeft ons

$$\{1,3\} \mid \{2,3\} \mid (\mathrm{lc}\,1)(\{1,2\} \mid \{1,3\} \mid \{2,3\}).$$

$$\phi_3 = \begin{cases} 1 \to 2 \\ 2 \to 1 \\ 4 \to 3 \end{cases}$$

geeft ons

$$\{1,2\} \mid \{1,3\} \mid (\mathrm{lc}\,2)(\{1,2\} \mid \{1,3\} \mid \{2,3\}).$$

$$\phi_4 = \begin{cases} 1 \to 2 \\ 2 \to 3 \\ 4 \to 1 \end{cases}$$

geeft ons

$$\{1,3\} \mid \{2,3\} \mid (\mathrm{lc}\,2)(\{1,2\} \mid \{1,3\} \mid \{2,3\}).$$

$$\phi_5 = \begin{cases} 1 \to 3 \\ 2 \to 1 \\ 4 \to 2 \end{cases}$$

geeft ons

$$\{1,2\} \mid \{1,3\} \mid (\mathrm{lc}\,3)(\{1,2\} \mid \{1,3\} \mid \{2,3\}).$$

$$\phi_6 = \begin{cases} 1 \to 3 \\ 2 \to 2 \\ 4 \to 1 \end{cases}$$

geeft ons

$$\{1,2\} \mid \{2,3\} \mid (\mathrm{lc}\,3)(\{1,2\} \mid \{1,3\} \mid \{2,3\}).$$

Voor S_2 is de kleinste string bijgevolg

$$\{1,2\} \mid \{1,3\} \mid (\mathrm{lc}\,2)(\{1,2\} \mid \{1,3\} \mid \{2,3\}).$$

Deze is echter gelijk aan degene die tot hiertoe de kleinste was.

F.3 S_3

$$\phi_1 = \begin{cases} 2 \to 1 \\ 3 \to 2 \\ 4 \to 3 \end{cases}$$

geeft ons

$$\{1,2\} \mid \{1,3\} \mid \{2,3\}$$

Dit zal voor elke mapping hetzelfde zijn, omdat het we de 3-*clique* beschouwen. Voor ψ hebben we nu echter twee mogelijkheden (we beschouwen telkens één knoop die afgeschermd wordt):

$$\psi_1 = \begin{cases} 2 \to 1 \\ 1 \to 2 \\ 4 \to 3 \end{cases}$$

of

$$\psi_2 = \begin{cases} 2 \to 1 \\ 3 \to 2 \\ 1 \to 3 \end{cases}$$

De eerste optie levert ons een expressie op van de vorm

$$\{1,2\}$$

Merk op dat we in principe ook de boog $\{2,4\}$ moeten bijvoegen, maar dat hangt van de implementatie af. Wij opteren er hier voor om dat niet te doen. De tweede optie levert ons

$$\{1,3\}$$

We verkrijgen de twee expressies

$$\{1,2\} \mid \{1,3\} \mid \{2,3\} \mid (\mathrm{lc}\, 2)(\{1,2\})$$

$$\{1,2\} \mid \{1,3\} \mid \{2,3\} \mid (\mathrm{lc}\, 3)(\{1,3\})$$

De rest van de mogelijkheden geeft analoge resultaten. Als we ervan uitgaan dat "{" na "(" komt, dan zal geen enkele expressie echter voor

$$\{1,2\} \mid \{1,3\} \mid (\mathrm{lc}\, 2)(\{1,2\} \mid \{1,3\} \mid \{2,3\})$$

komen, de tot nu toe minimale. We mogen bijgevolg concluderen dat dit de minimale of canonische expressie is voor de graaf G.

We merken tot slot nog op dat we bij het construeren vaak al vroeger kunnen zien of we op de goede weg zijn: Indien onze expressie tot hiertoe al groter is dan de tot nu toe kleinste, dan mogen we al stoppen.

Bijlage G

GraphGen commando's

Tot slot van dit hoofdstuk geven we nog aan hoe GraphGen gebruikt kan worden. We beschrijven alle commando's, meer help kan verkregen worden in het programma zelf. De commando's zijn gegroepeerd per functionaliteit. Sommige commando's bieden nog additionele opties, deze worden beschreven in de interne help-functie.

`help` Dit commando aanvaard één argument: een ander commando. Het geeft hierover meer informatie.

`quit` Sluit het programma af

`addedge` Voeg een boog toe aan de actieve graaf. bvb. `addedge 0 3` voegt een boog toe tussen knoop 0 en knoop 3.

`deledge` Verwijder een boog tussen twee knopen.

`addnode` Voeg een nieuwe knoop toe.

`delnode` Verwijder de laatste knoop.

`calclow` Bereken alle *lower objects* volgens de standaard implementatie.

`calcup` Bereken alle *upper objects* volgens de standaard implementatie

`canon` Canoniseer de actieve graaf volgens een standaardcanonisatiealgoritme.

`permute` Permuteer de graaf. bvb. `permute (1 2 3)(4 5)`. Dit kan ook gebruikt worden met de naam van een opgeslagen permutatie: `permute perm1`.

`autg` Bereken de automorfismen van de actieve graaf.

`loadg` Laad een actieve graaf uit het geheugen in. De parameter is de naam van de graaf. bvb. `loadg graaf1`. Merk op dat de graaf gekopieerd wordt in de actieve graaf.

`storeg` Sla een graaf op in het geheugen onder een bepaalde naam.

`delg` Verwijder een graaf uit het geheugen. Ook hier is het enige argument de naam van de te verwijderen graaf.

`printg` Toon de actieve graaf op het scherm.

`displg` Geef alle grafen in het geheugen weer.

`writedot` Schrijf een dot-file die toelaat de graaf te visualiseren met GraphViz [1].

`exportpng` Exporteer de actieve graaf naar een png-bestand, hiervoor is GraphViz vereist.

`loadps` Laad een permutatie in de actieve met behulp van de standaardnotatie.

`loadpc` Laad een permutatie in de actieve met behulp van de cykelnotatie.

`loadp` Laad een permutatie uit het geheugen.

`storep` Sla een permutatie tijdelijk in het geheugen op.

`delp` Verwijder een permutatie uit het geheugen.

`printp` Toon de actieve permutatie op het scherm

`displp` Geef alle permutaties in het geheugen weer.

`mckay` Run het algoritme van McKay, vertrekkende vanuit de actieve graaf.

`mckayreport` Zelfde als `mckay`, hierbij wordt echter ook een website gegenereerd.

`orderly` Run het generisch orderly algoritme, toegepast op grafen voor een bepaalde n.

`orderlyreport` Zelfde als `orderly`, hierbij wordt echter ook een website gegenereerd.

Het programma is in staat om HTML-output te genereren zodat we door de gegenereerde grafen kunnen browsen en bij elke graaf informatie krijgen over hoeveel nakomelingen hij genereerde, hoe lang het duurde,... zoals beschreven op het einde van hoofdstuk 4.

Bibliografie

[1] Graphviz. `http://www.graphviz.org` last checked: 1 January 2009.

[2] Tree decomposition - wikipedia, the free encyclopedia. `http://en.wikipedia.org/wiki/Treewidth` last checked: 1 January 2009.

[3] Dimitris K. Agrafiotis, Deepak Bandyopadhyay, Jörg K. Wegner, and Herman van Vlijmen. Recent advances in chemoinformatics. *J. Chem. Inf. Model.*, 47(4):1279–1293, 2007.

[4] Stefan Arnborg, Derek G. Corneil, and Andrzej Proskurowski. Complexity of finding embeddings in a k-tree. *SIAM J. Alg. Disc. Meth.*, 8(2):277–284, april 1987.

[5] Hendrik Blockeel, Christophe Costa Florencio, Jan Ramon, Jan Van den Bussche, and Dries Van Dyck. Context-free graph grammers as a language-bias mechanism for graph pattern mining.

[6] H. L. Bodlaender. Polynomial algorithms for graph isomorphism and chromatic index on partial k-trees. *J. Algorithms*, 11:631–643, 1990.

[7] Hans L. Bodlaender. A partial k-arboretum of graphs with bounded treewidth. *Theor. Comput. Sci.*, 209(1-2):1–45, 1998.

[8] Béla Bollobás. *Modern Graph Theory*. Springer, 1998.

[9] Christian Borgelt, Michael R. Berthold, and David E. Patterson. Molecular fragment mining for drug discovery. In *ECSQARU*, pages 1002–1013, 2005.

[10] Gunnar Brinkmann. Isomorfic rejection in structure generation programs.

[11] D. Bronner and B. Ries. An introduction to treewidth. `http://infoscience.epfl.ch/record/89584/files/reportfinal.ps`, march 2006.

[12] Bruno Coucelle. Graph algebras and monadic second-order logic. to be published, in preparation. `http://www.labri.fr/perso/courcell/ActSci.html`.

[13] Tom Davis. Cycle notation. Last checked: 25 October 2008 `http://mathcircle.berkeley.edu/BMC3/perm/node3.html`.

[14] Reinhard Diestel. *Graph Theory*, volume 3. Springer-Verlag Heidelberg, 2005.

[15] I. A. Faradzev. Constructive enumeration of combinatorial objects. *Problèmes Combinatoires et Théorie des Graphes: Colloques Internationaux*, 206:131–135, 1978.

[16] M. R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman, 1979.

[17] Yuri Gurevich. From invariants to canonization. *The Bull. of Euro. Assoc. for Theor. Computer Sci*, (63), 1997.

[18] Jiawei Han and Micheline Kamber. *Data Mining: Concepts and Techniques*. Elsevier Science and Technology Books, 2 edition, 2006.

[19] Pinar Heggernes. Treewidth, partial k-trees, and chordal graphs. Unpublished, `http://www.ii.uib.no/~pinar/chordal.pdf`, september 2006.

[20] John E. Hopcroft and Richard M. Karp. An $n^{5/2}$ algorithm for maximum matchings in bipartite graphs. *SIAM J. Comput.*, 2(4):225–231, 1973.

[21] David S. Johnson. The np-completeness column: An ongoing guide. *J. Algorithms*, 6(3):434–451, 1985.

[22] Mami Kuroda, Katsutoshi Yada, Hiroshi Motoda, and Takashi Washio. Knowledge discovery from consumer behavior in an alcohol market by using graph mining technique : An example of using an active mining process for a typical business application. *Joint Workshop of Vietnamese Society of AI, SIGKBS-JSAI, ICS-IPSJ, and IEICE-SIGAI on Active Mining*, 104(486):21–26, 2004.

[23] Brendan D. McKay. *Nauty User Guide*, 2.2 edition. `http://cs.anu.edu.au/~bdm/nauty/nug.pdf`.

[24] Brendan D. McKay. Isomorph-free exhausive generation. *J. Algorithms*, (26):306–324, 1998.

[25] Andrzej Proskurowski. Separating subgraphs in k-trees: Cables and caterpillars. *Discrete Mathematics*, 49(3):275–285, 1984.

[26] Ronald C. Read. Every one a winner. *Annals of Discrete Mathematics 2*, pages 107–120, 1978.

[27] J. Rekers and Andy Schürr. A graph grammar approach to graphical parsing. In *VL*, pages 195–202, 1995.

[28] Neil Robertson and Paul D. Seymour. Graph minors .xiii. the disjoint paths problem. *J. Comb. Theory, Ser. B*, 63(1):65–110, 1995.

[29] Grzegorz Rozenberg, editor. *Handbook of Graph Grammars and Computing by Graph Transformations*. World Scientific, 1997.

[30] W.R. Scott. *Group Theory*. Dover Publications, Inc., 1987.

[31] Michael Sipser. *Introduction to the Theory of Computation*. Thomson Course Technology, 2006.

[32] N. J. A. Sloane. Reeks a000088: grafen op n ongelabelde knopen. `http://www.research.att.com/~njas/sequences/A000088` last checked: 19 January 2009.

[33] Pang-Ning Tan, Micharl Steinbach, and Vipin Kumar. *Introduction to Data Mining (International Edition)*. Addison Wesley, 2006.

[34] Joshua R. Tyler, Dennis M. Wilkinson, and Bernardo A. Huberman. Email as spectroscopy: Automated discovery of community structure within organizations. Technical report, HP Labs, 1501 Page Mill Road, Palo Alto CA, 94304.

[35] Dries Van Dyck. Partial k-trees and bounded treewidth. september 2006.

[36] Katsutoshi Yada, Hiroshi Motoda, Takashi Washio, and Asuka Miyawaki. Consumer behavior analysis by graph mining technique. In *KES*, pages 800–806, 2004.

[37] Xifeng Yan and Jiawei Han. gspan: Graph-based substructure pattern mining. *Data Mining, IEEE International Conference on*, page 721, 2002.

[38] Mike Zabrocki. Number of graphs on n unlabelled vertices. `http://garsia.math.yorku.ca/~zabrocki/math3260w03/nall.html` last checked: 1 January 2009.