Faculteit Ingenieurswetenschappen
Faculteit Wetenschappen
Vakgroep Toegepaste Wiskunde en Informatica
Voorzitter: Prof. Dr. G. VANDEN BERGHE

Grenzen van Patches

door

Bart COPPENS

Promotor: Prof. Dr. G. BRINKMANN
Co-Promotor: Prof. Dr. V. FACK

Scriptie ingediend tot het behalen van de academische graad van
licentiaat in de informatica

Academiejaar 2006–2007

Voorwoord

Eerst en vooral wil ik mijn promotor prof. dr. Gunnar Brinkmann bedanken voor zijn steun en zijn altijd interessante ideeën en opmerkingen. Het was een plezier om met hem te kunnen samenwerken. Verder wil ik natuurlijk ook mijn ouders bedanken.

Bart Coppens, 31 Mei 2007

Toelating tot bruikleen

Bart Coppens, 31 Mei 2007

Grenzen van Patches

door

Bart COPPENS

Scriptie ingediend tot het behalen van de academische graad van
licentiaat in de informatica

Academiejaar 2006–2007

Promotor: Prof. Dr. G. BRINKMANN
Co-Promotor: Prof. Dr. V. FACK
Faculteit Ingenieurswetenschappen
Faculteit Wetenschappen

Vakgroep Toegepaste Wiskunde en Informatica
Voorzitter: Prof. Dr. G. VANDEN BERGHE

Samenvatting

De groei van fullerenen is een interessant probleem. Dit kan theoretisch worden gemodelleerd met transformaties van de als grafen voorgestelde abstracte chemische structuren van fullerenen. Deze transformaties kunnen worden voorgesteld aan de hand van een specifieke soort van grafen, *patches* genaamd. In het transformatieproces is het van belang dat twee verschillende patches eenzelfde *rand* hebben.

In deze thesis zullen we bespreken hoe we deze transformaties efficiënt kunnen genereren, en wat de aandachtspunten zijn in dit generatieproces.

In het bijzonder zal bekeken worden hoe we snel deze randen kunnen genereren, en hoe we zo een rand op alle mogelijke manieren kunnen invullen met vijf- en zeshoeken. Daarna zal dit worden toegepast op twee specifieke problemen, namelijk het genereren van een lijst van *groeiparen* en het genereren van een lijst van *isomerisatieparen* en *isomerisatiepatches*.

Trefwoorden

theoretische chemie, fullerenen, patches, planaire grafen, snelle generatie van objecten, invullingen genereren

Grenzen van Patches
Boundaries of Patches

Bart Coppens

Supervisor(s): Gunnar Brinkmann

Abstract—**In chemistry, the question of how fullerenes grow is an interesting one. We look at a graph-theoretical model of this problem that uses patch transformations as a way to simulate how a fullerene can change its structure. This is done by looking at patches that have the same boundary and can thus be replaced in the fullerene by just changing the interior of the boundary. We can make a list of 'interesting' patch transformations through exhaustive generation of all possible boundary codes of applicable patches and the possible interiors of these boundaries.**

Keywords—**theoretical chemistry, fullerenes, patches, planar graphs, fast generation of objects, generating interiors**

I. Introduction

FULLERENES are carbon-based molecules that form a closed, ball-shaped structure. The carbon molecules on this closed structure all form pentagons and hexagons in such a way that every carbon atom has exactly three neighbouring carbon atoms with which it is bonded. This can be easily modeled with planar graphs by representing each carbon atom as a node and every (single or double) bond between carbon atoms as an edge between two nodes.

We will focus on parts of this graph structure called *patches*. A (h, p)-*patch* (or simply a patch) is a 2-connected planar graph with every bounded face either a hexagon or a pentagon. All its nodes that are not adjacent to the unbounded face have precisely three neighbours. The nodes that are adjacent to the unbounded face can have either two or three neighbours. We will be interested in the boundary of these patches, which can be represented as a cyclic list of the node valencies and how we can fill this boundary with all possible interiors of pentagons and hexagons.

A very interesting question is how these fullerenes actually form. One of the current theories is the '*fullerene road model*' ([1]). This theory says that larger fullerenes grow from smaller fullerenes by a series of transformations that change the internal structure of the fullerene, either by injecting more carbon atoms, or by changing the faces that are formed by the atoms. These transformations are called growth transformations and isomerisation transformations, respectively.

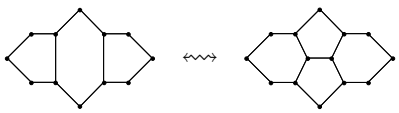


Fig. 1. The Endo-Kroto C_2 insertion

A good theoretical model for these transformations is interpreting these transformations as local transformations of the graph structure. We model these by 'cutting' a patch out of the fullerene and replacing this patch by a different patch that has the same boundary. A pair of two patches with the same boundary and where one of the two patches has strictly more nodes than the other, is called a *growth pair*. A well-known example of this is the Endo-Kroto C_2 insertion, as seen in figure 1. A fullerene can acquire two carbon atoms by doing a local transformation that replaces the left patch with the right one.

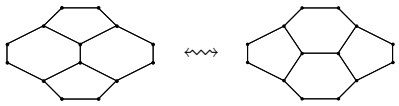


Fig. 2. The Stone-Wales isomerisation transformation

Another important transformation is the Stone-Wales isomerisation transformation, as seen in figure 2. This is an example of an *isomerisation patch*. This is a patch with a boundary that has more symmetry than the patch itself. We can then rotate the patch inside the fullerene, such that the rotation is a symmetry of the boundary but not of the patch. Similarly we have *isomerisation pairs* that are non-isomorphic patches with the same boundary and the same node count.

The core idea here is that if such a transformation is applied to a fullerene, it changes the adjacency of hexagons to pentagons in the fullerene. It thus changes the local structure of the fullerene. This is especially important when one considers the *Isolated Pentagon Rule* (IPR). The IPR is a chemical theory that claims that it is energetically unfavourable for a fullerene to have two adjacent pentagons. With an isomerisation transformation, a fullerene can change from an (intermediate) unstable configuration to a more stable configuration by changing the positions of the pentagons.

Our goal is to enumerate all 'relevant' growth pairs, isomerisation pairs and isomerisation patches. We do this in three steps. First we generate all possible non-isomporphic boundaries for a specified length and number of pentagons. Secondly, we try to fill each such boundary in all possible ways with pentagons and hexagons and then filter isomorphic patches. Finally, we can apply filters to these lists of patches: we can look for growth pairs, isomerisation pairs and isomerisation patches. In this step we also can filter away pairs that we can consider to be 'irrelevant' (for example, because they cannot be embedded in a fullerene). This way, we can check our results against those from [2] and [3] in order to check these results independently.

II. Generation of non-isomorphic boundaries

We want to generate all possible boundary encodings that satisfy some parameters and can possibly be filled with at least one interior of hexagons and pentagons. In particular, these parameters should be chosen in such a way that it is very easy to generate all these boundaries.

A boundary can be represented as a cyclic list of boundary

node valencies. We will see this as a string of the characters '2' and '3', where the '2' represents a node in the boundary that will have 2 neighbours (both its cyclic neighbours in the boundary representation) and the '3' represents a node in the boundary that has 3 neighbours (two of which are in the boundary and one which is on the 'inside'). For example, the Endo-Kroto patches in figure 1 have a boundary code of '232223232223'. Hence, to generate all possible boundaries, it suffices to enumerate all possible permutations of '2' and '3', for a specific amount of each. We call the number of '2's in the boundary code v_2 and the number of '3's v_3.

We can determine these amounts easily if we have the length of the boundary l and the number of pentagons p that any interior of the boundary must have. There is a well-known relation between the number of pentagons in a patch and the boundary code, that says that $v_2 - v_3 = 6 - p$. Since $l = v_2 + v_3$, we can easily get the v_2 and v_3 from a given p and l:

$$v_2 = (l - p)/2 + 3$$
$$v_3 = (l + p)/2 - 3$$

With this information, we can generate a list of all permutations of a v_2 and 3 for a given l and p. However, since this list that encodes the boundary is cyclic, it is important to be careful not to generate all cyclic permutations or mirror images of a boundary. These are essentially the 'same' boundary, but if we would generate all these permutations, we would have done work that could have been avoided.

We solve this problem by introducing a canonical form for boundaries. This is a unique encoding of a boundary, chosen in such a way that from an arbitrary representation of a boundary, we can always find the canonical encoding of this boundary. Now instead of generating all boundary encodings, we will generate *only* canonical encodings. We do this by cutting away branches of the generation process that can't lead to a canonical boundary encoding. Furthermore, during the generation process, we will collect information that will speed up a final check to see if a boundary encoding is indeed canonical.

III. Interior of a boundary

We developed an efficient algorithm to determine all possible interiors of a given border. This is done by recursively as follows. A pentagon or hexagon is 'glued' to an existing border inside of the structure we are generating. Each time this happens, one or two new interior borders are created, which we fill recursively. Of course, care has to be taken that this is done in such a way that we generate all possible interiors. During this generation process, bounding criteria are applied to the structures that are being generated. With good bounding criteria, it is possible to cut a branch of the generating process to speed up the process.

The most efficient bounding criterium we use is implemented using a form of *dynamic programming*. During the interior of a boundary that is part of a structure that we are generating, we store information on whether certain boundaries we encounter have a interior or not. Once we find out that a certain boundary does not have any possible interior, we re-use this information when we encounter the same boundary later in the process.

We take this dynamic programming one step further and make use of *inter-process* 'dynamic programming'. Instead of just remembering the boundaries we saw in the current generation process, we store the results of the dynamic programming step in a file. The next time the program is executed, it will read this file and use this as the basic dynamic programming information. That way, we can take advantage of the information amassed over the course of a lot of executions of the algorithm, rather than just the current execution.

Of course, we also have to prove that our algorithm is indeed correct and terminates. Both these theorems are important to ensure that the program does is expected. Unfortunately there exist boundaries that have an *infinite* number of different interiors. This is because it is possible that while interior a boundary with hexagons, we encounter exactly the same boundary on the inside. That way, we can keep adding the hexagons in the same manner to this inside and continue getting the same boundary over and over, while the amount of added hexagons keeps increasing.

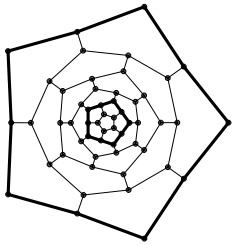


Fig. 3. Example of a patch that has a boundary that is equal to an 'internal boundary'

To this end, we prove a theorem that guarantees us that this will never happen in a specific case. Specifically, when $p < 6$, we prove that our algorithm will always terminate. In case $p = 6$, such a result is not possible. For example, in figure 3, we see a patch that has a boundary that occurs twice in such a way, that we can create an infinite series of ever-growing patches that have the same boundary.

A chemical interpretation of this is simple: just look at nanotubes. These are fullerenes that look like long wires with two half-spherical caps attached to the sides. If you cut such a tube in half, you get two patches, each with $p = 6$ (because the total number of pentagons is always 12 in a fullerene). But you can just make this tube longer and cut it in such a way that you again have the same boundaries. That way, the longer the nanotube gets, the more nodes will be in the patch.

References

[1] D.E. Manopoulos and P.W. Fowler, "Downhill on the fullerene road: a mechanism for the formation of C_{60}," *The Chemical physics of the fullerenes 10 (and 5) years later*, 1996.

[2] G. Brinkmann and P.W. Fowler, "A catalogue of growth transformations of fullerene polyhedra," *J. Chem. Inf. Comput. Sci*, , no. 43, pp. 1837–1843, 2003.

[3] G. Brinkmann, P.W. Fowler, and C. Justus, "A catalogue of isomerization transformations of fullerene polyhedra," *J. Chem. Inf. Comput. Sci*, , no. 43, pp. 917–927, 2003.

Inhoudsopgave

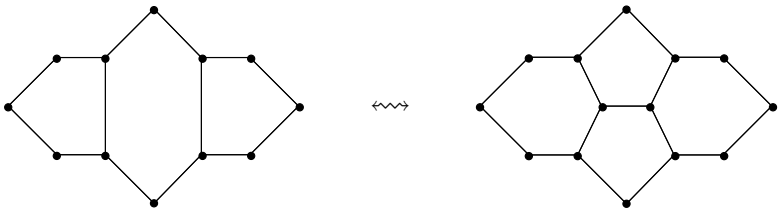
Hoofdstuk 1

Inleiding

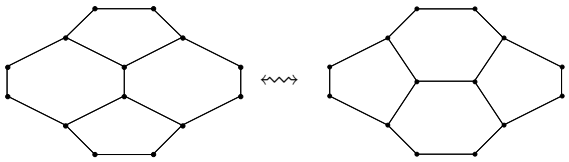
Sinds de ontdekking van het Buckminsterfullereen in 1985 heeft de familie van fullerenen grote aandacht getrokken. Niet alleen is de zogenaamde 'Buckyball' een algemeen bekend begrip geworden, ook de verwante nanotubes duiken vaak op in de wetenschapsberichtgeving. Beide zijn lid van de chemische klasse van fullerenen. Dit is een specifieke soort van moleculen die helemaal bestaan uit koolstofatomen. Het is interessant om deze moleculen op een theoretische wijze te modeleren, omdat op deze manier op een eenvoudige manier voorspellingen kunnen worden gemaakt over het ontstaan en het gedrag van deze moleculen.

De koolstofatomen van fullerenen vormen het oppervlak van een bolvormig lichaam. De vlakken tussen deze atomen kunnen worden voorgesteld als vijf- en zeshoeken, waarbij elk koolstofatoom precies drie buren heeft. Op deze manier kunnen we zo een fullereen zeer eenvoudig abstract voorstellen door een graaf. Dan ziet de voorstelling van een fullereen er eenvoudigweg uit als een aantal 'toppen', die elk 3 'buren' hebben op zó een wijze dat er enkel vijf-en zeshoeken zijn.

Een interessant probleem is hoe deze fullerenen kunnen ontstaan en veranderen. Één theorie stelt dat grotere fullerenen 'groeien' uit kleinere fullerenen. Hiermee wordt bedoeld dat het aantal koolstofatomen verhoogt tijdens een lokaal groeiproces. Een andere actie die kan gebeuren op deze fullerenen is dat ze veranderen in structuren met evenveel koolstofatomen, maar met een andere 'vorm'. Dit wordt dan analoog het isomerisatieproces genoemd. Bij dit laatste kan bijvoorbeeld het verschil tussen twee fullerenen zijn dat in het ene fullereen geen enkele vijfhoek de andere raakt, terwijl bij het andere fullereen twee vijfhoeken aan mekaar grenzen. Dit is interessant bij de *Isolated Pentagon Rule*. Dit is een chemische theorie die stelt dat een situatie waar in een fullereen 2 vijfhoeken aan mekaar grenzen geen stabiele configuratie zal



Figuur 1.1: De Endo-Kroto C_2 invoeging



Figuur 1.2: De Stone-Wales isomerisatietransformatie

zijn. Hierdoor kan het helpen om te weten op welke manieren zo een 'instabiele' fullereen kan veranderen in een 'stabielere' fullereen.

Een bekend voorbeeld van een groeitransformatie is de Endo-Kroto C_2 invoeging. Dit is een voorgesteld mechanisme waarbij fullerenen groeien door de invoeging van twee koolstofatomen in een fullereen. Deze specifieke transformatie, die voorgesteld is in afbeelding 1.1, zal een zeshoek waaraan de twee tegenoverliggende vijfhoeken grenzen, omzetten in een specifieke aaneenschakeling van twee zeshoeken en twee vijfhoeken.

Ook van de isomerisatietransformaties is er een bekend voorbeeld. De Stone-Wales pyracyleen herschikking zal een bepaalde configuratie van twee zeshoeken en twee vijfhoeken roteren om zijn centrum in het fullereen, zodat een ander fullereen ontstaat. Deze transformatie is afgebeeld in figuur 1.2.

Beide soorten transformatie kunnen dan gebruikt worden om de 'fullerenenweg' (zie [MF96]) te beschrijven. Dit is een manier waarop fullerenen, zoals bijvoorbeeld een C_{60} Buckminsterfullereen, via kleine stapjes van isomerisatie en groeien wordt opgebouwd uit kleinere fullerenen, op een manier die zo weinig mogelijk energie vraagt. Dit wil zeggen dat om al deze transformaties uit te voeren, zo weinig mogelijk (thermische) energie benodigd is om tot het eindresultaat te komen.

Om deze processen te modeleren op het abstracte niveau van grafen worden patches gebruikt. Een patch is een graaf waarbij, behalve de 'randtoppen', alle toppen 3 buren hebben, en waar in dit geval alle interne vlakken vijf- en zeshoeken zijn. We kunnen deze patches bekomen indien we een aaneengesloten stuk van vijf- en zeshoeken weghalen uit een fullereen. Er zijn dan een stuk van een fullereen en een patch die in mekaar passen op deze rand. Om het groei- of

isomerisatieproces dan te modelleren, kunnen we dit beschouwen als een operatie waar we een patch vervangen door een andere patch die eenzelfde rand heeft.

Het groeien van een fullereen kan dan worden beschouwd als het vervangen van een patch door een andere patch met eenzelfde rand, maar met meer toppen. Om deze transformaties goed te kunnen begrijpen, is het interessant dat er lijsten worden gemaakt van dit soort patches. Deze abstracte beschrijvingen van mogelijke groeiprocessen van fullerenen kunnen chemici dan helpen bij het achterhalen hoe dit proces in de natuur voorkomt.

Het probleem is dan om lijsten van deze patches met eenzelfde rand te kunnen genereren. In deze thesis zal zowel een algoritme worden gegeven om efficiënt randen te genereren, als een algoritme dat, gegeven een bepaalde rand, alle mogelijke patches met deze rand zal kunnen teruggeven op een zo snel mogelijke manier. Hiertoe zullen we een aantal manieren bekijken om dit inderdaad zo snel mogelijk te doen. Hierna zal dit worden toegepast op het probleem van groeitransformaties.

Hoofdstuk 2

Definities

2.1 Grafentheorie

Een *graaf* is een eindige verzameling *toppen*, en een eindige verzameling van verzamelingen van twee verschillende toppen, de *bogen* genoemd. Een boog kan dan worden gezien als een verbinding tussen 2 toppen van de graaf. We zullen een graaf G dus bekijken als het koppel (V, E), waar V de toppenverzameling is, en E de bogenverzameling. Bovendien hebben we ook functies die gegeven een graaf respectievelijk de toppen- en bogenverzameling zullen teruggeven: $E(G) = E$ en $V(G) = V$.

Er zijn ook andere definities mogelijk voor een graaf. Deze definities kunnen speciale gevallen toelaten, die in onze definitie niet voor kunnen komen. Zo zijn bijvoorbeeld *Lussen* 'bogen' met eenzelfde top als begin- en eindpunt, en een graaf heeft *parallelle bogen* indien er het mogelijk is dat er tussen een paar van begin- en eindtoppen meerdere verschillende bogen zijn. Indien parallelle bogen zijn toegelaten in een graaf, spreekt men over een *multigraaf*. In een *gerichte graaf* zal een boog van top a naar top b niet eenzelfde boog zijn als een boog van b naar a. Indien dit niet mogelijk is, wordt de graaf *niet-gericht* genoemd, dit zijn dus grafen waarvoor de bogen niet gericht zijn.

Als er over *een ingebedde graaf* wordt gesproken, wordt bedoeld dat de graaf niet alleen een toppen- en bogenverzameling heeft, maar ook een *inbedding* in een object. Een inbedding is een afbeelding die de toppen en bogen specifieke plaatsen zal geven op dit object. Een veelgebruikt object voor inbeddingen is het oppervlak van een bol. Dit is het object dat wij zullen gebruiken. Een ander vaak gebruikt object is het 2-dimensionale vlak. Indien een graaf een inbedding in het vlak bezit waarbij geen enkele boog een andere boog kruist, spreken we van een *planaire*

graaf, met een *planaire inbedding*. Een planaire inbedding wordt ook een *tekening* van de graaf genoemd.

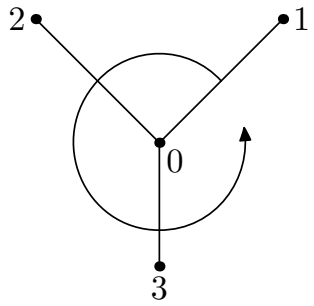
Indien we een inbedding in het boloppervlak hebben, kunnen we het verschil nemen tussen dit oppervlak en de graaf. Hierin kunnen we dan verschillende aaneengesloten deelverzamelingen onderscheiden. Zo een deelverzameling wordt een *vlak* genoemd. We zullen één vlak een bijzondere naam geven: het *buitenvlak*. De andere vlakken noemen we *begrensd*. Toppen die aan het buitenvlak grenzen, worden *randtoppen* genoemd, de andere toppen zijn *intern*. De verzameling van bogen en toppen die aan het buitenvlak liggen, wordt de *rand* genoemd.

De *duale* van een vlakke graaf G is een 'speciale' graaf, en heeft voor elk vlak in G een top, en zal een boog hebben tussen twee toppen voor elke boog in G die aan twee niet noodzakelijk verschillende vlakken ligt. Doordat vlakken in G door meerdere bogen aan mekaar kunnen grenzen, is het mogelijk dat er in de duale van G tussen twee toppen *verschillende* bogen zijn. Deze duale zal dus geen graaf meer zijn volgens onze definitie. Dit is echter geen wezenlijk probleem.

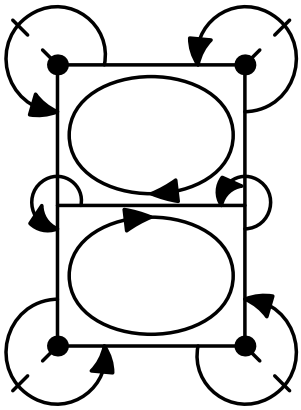
We willen ons echter niet bezig houden met specifieke inbeddingen die toppen op coördinaten op een boloppervlak afbeelden. We zullen hiertoe gebruik maken van een *rotatiesysteem*. Voor een rotatiesysteem hebben we in elke top een cyclisch geordende lijst van buren. Deze zijn geordend volgens een voor elke top vast gekozen *draairichting*. Indien we in de gekozen richting kijken welke bogen we rond een bepaalde top tegenkomen, zullen deze op de volgorde van deze lijst te vinden zijn, op een cyclische verschuiving na. Voor de eenvoud zullen we verder stellen dat de draairichting rond elke top dezelfde is.

Een voorbeeld kan dit duidelijker maken. In figuur 2.1 is te zien hoe we een vaste draairichting kiezen in de top. Indien we '1' als eerste buur in de lijst nemen, draaien we in de gekozen richting, en vinden we als burenlijst $(1, 2, 3)$.

Dankzij zo'n rotatiesysteem kunnen we in een (planaire) inbedding eenvoudig de vlakken afbakenen. Dit kan gedaan worden met het *Face Tracing Algorithm* ([GT87]). Indien we gewoon een boog die op de rand van een vlak ligt zouden nemen als herkenpunt van een vlak, zouden we het onderscheid niet kunnen maken tussen twee vlakken die deze boog als gemeenschappelijke rand hebben. Dit onderscheid kan wel gemaakt worden, indien we specifiek een boog kiezen, en dan specifiek een starttop en een eindtop kiezen. Indien we zo een starttop en eindtop vastleggen, noemen we dit een 'gerichte boog' naar analogie met een gerichte boog in een gerichte graaf. Dit doet natuurlijk niets af aan het niet-gerichte karakter van onze grafen. We hebben nu reeds



Figuur 2.1: Draaien in een gekozen richting rond een top: de buren van top 0 zijn $(1, 2, 3)$ in die vaste volgorde



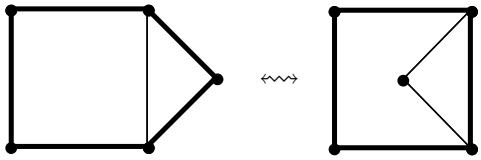
Figuur 2.2: Twee vlakken die kunnen worden overlopen door middel van een gekozen draairichting

voor alle toppen in de graaf een vaste draairichting gekozen. In de cyclische burenlijst van de eindtop kijken we naar de cyclische opvolger van de begintop. We nemen deze opvolger als nieuwe eindtop, en de oude eindtop als nieuwe begintop. Op deze manier kunnen we dan de randen van een vlak overlopen. Dit kan in het algemeen echter alleen in het geval dat de graaf geen toppen heeft die precies twee hebben. Gelukkig zal deze technische beperking in de gevallen waar wij deze zullen tegenkomen eenvoudig omzeild kunnen worden.

In figuur 2.2 zien we een voorbeeld van hoe we met zo een rotatiesysteem eenvoudig een vlak kunnen afbakenen, zonder de specifieke coördinaten van de toppen te moeten kennen.

Stelling (Theorema 3.2.2 in [GT87])**.** *Elk rotatiesysteem van een graaf definiëert een unieke lokaal georiënteerde inbedding voor deze graaf (op equivalentie van inbedding na). Omgekeerd definiëert ook elke lokaal georiënteerde inbedding een rotatiesysteem voor de graaf.*

We noemen zo een rotatiesysteem dan ook een *combinatorische inbedding.*



Figuur 2.3: Twee ingebedde planaire grafen die isomorf maar niet combinatorisch isomorf zijn

Een *pad* tussen 2 toppen is een geordende lijst van toppen, waarbij elke top een buur is van zijn opvolger in de lijst (behalve de laatste top). Een graaf wordt *samenhangend* genoemd indien er tussen elke 2 toppen van de graaf een pad bestaat. Indien de verwijdering van een top ervoor zorgt dat een graaf niet meer verbonden is, wordt dit een *cut vertex* genoemd. Indien in een graaf met minder dan n toppen na het verwijderen van elke mogelijke combinatie van $n-1$ toppen de graaf nog steeds verbonden is, wordt de graaf *n-samenhangend* genoemd. In het bijzonder is een 2-samenhangende graaf een graaf *zonder* cut vertex.

Een *cykel* is een pad waarbij de begin- en eindtop dezelfde zijn, en alle andere toppen op het pad verschillend zijn.

Twee niet noodzakelijk verschillende grafen $G = (V, E)$ en $G' = (V', E')$ worden *isomorf* genoemd indien er een bijectie $f : V(G) \to V(G')$ bestaat zodanig dat $(a, b) \in E(G) \Leftrightarrow (f(a), f(b)) \in E(G')$. Indien de twee grafen gelijk zijn, wordt gesproken van een *automorfisme*. Dit wordt vaak ook een *symmetrie* van de graaf genoemd.

Voor tekeningen van planaire grafen komt de bovenstaande definitie soms niet overeen met wat we intuïtief zouden verstaan onder isomorfe grafen. We kunnen twee inbeddingen geven van een planaire graaf, waarbij bijvoorbeeld een vierhoek wordt afgebeeld op een vijfhoek. Dit is te zien in de figuur 2.3: we kunnen eenvoudig een bijectieve afbeelding zien die alle toppen van de linkergraaf op de rechtergraaf afbeeldt, en de buur-relatie bewaart. Echter, de linkergraaf heeft als buitenrand een vijfhoek, terwijl de rechtergraaf een vierkant als buitenrand heeft.

Om dit op te lossen, gebruiken we de term *combinatorisch isomorfisme*. Twee ingebedde planaire grafen worden combinatorisch isomorf genoemd indien een bijectie tussen de toppen en bogen van beide grafen bestaat die zó kan worden uitgebreid dat dat vlakken ook op vlakken worden afgebeeld, waarbij de adjacentie van vlakken met bogen en toppen bewaard blijft. Veelhoeken worden dus op veelhoeken van dezelfde grootte afgebeeldt. Dit zal dus ook een 'isomorfisme' geven tussen de rotatiesystemen van beide ingebedde grafen.

Het is belangrijk op te merken dat het niet veel uitmaakt of we een inbedding kunnen maken op een boloppervlak of op het 2-dimensionale vlak. Indien we een inbedding op de bol hebben,

dan is er een combinatorisch isomorfisme tussen deze inbedding op het boloppervlak en een inbedding op het 2-dimensionale vlak. We kunnen dus beide soorten inbeddingen door mekaar gebruiken. Merk voorts op dat we dit isomorfisme zo kunnen kiezen dat het begrensde vlak van de inbedding op het boloppervlak zal worden afgebeeldt op een 'oneindig' vlak op het 2-dimensionale vlak. Indien we dus een inbedding hebben op de bol waarbij de bogen elkaar niet snijden, hebben we meteen ook een planaire inbedding in het vlak.

2.2 Patches

Een (h, p)-*patch*, of kortweg *patch*, is een ingebedde, 2-samenhangende graaf, waarbij elk begrensd vlak een vijf- of zeshoek is, en waarbij elke interne top exact drie buren heeft. Toppen op de rand kunnen zowel twee als drie buren hebben. Met de *rand* worden dus de toppen bedoeld die grenzen aan het buitenvlak. Zo'n rand wordt vaak voorgesteld door een cyclische *randcode*. Dit is een lijst van het aantal buren dat de toppen aan de rand hebben, en zal dus bestaan uit een opeenvolging van 2 en 3. Voor de eenvoud en duidelijkheid kan een exponentnotatie gebruikt worden, waarbij een stuk code een exponent krijgt, die aanduidt hoeveel maal dit stuk rand wordt herhaald. Zo zal bijvoorbeeld '$(332)^n 2^4$' de rand voorstellen die begint met n keer '332' achter elkaar, gevolgd door '2222'. Een *invulling* van een rand is een concrete patch die de gegeven rand heeft.

We noemen twee niet noodzakelijk verschillende patches nu isomorf, indien er een een combinatorisch isomorfisme tussen de twee patches bestaat. Er moet dus een bijectie bestaan tussen de toppen, de bogen en de vlakken van de patches, zodanig dat die bijectie de adjacentie van toppen, bogen en vlakken bewaart. Vijfhoeken worden op vijfhoeken afgebeeld, zeshoeken op zeshoeken, en buitenvlak op het buitenvlak. Analoog is een automorfisme van een patch dus een combinatorisch isomorfisme van de patch met zichzelf.

Het *aantal voorkomens* van een '2' en een '3' in een rand B van patch P wordt voorgesteld als respectievelijk $v_{2,B}(P)$ en $v_{3,B}(P)$. De *lengte* van de rand van een patch wordt voorgesteld door $b(P)$. Indien er geen verwarring is over welke patch het gaat, zal de vermelding van de patch P en de rand B weggelaten worden.

Een heel bekend basisresultaat over patches is volgende een relatie tussen het aantal vijfhoeken dat in een patch zit, en v_2 en v_3.

Stelling (Lemma 1 in [JBG03])**.** *Voor een (h, p)-patch P met rand B hebben we dat $v_{2,B}(P) - v_{3,B}(P) = 6 - p$ en $b(P) = 2v_{3,B}(P) + 6 - p$*

In deze thesis zal steeds worden gesproken over patches met $p < 6$, met soms een kleine uitbreiding naar $p \leq 6$. Dit komt doordat we voornamelijk geïnteresseerd zijn in de toepassingen in de chemie, waar blijkt dat indien $p > 6$, we reeds het 'grootste' deel van de fullereen zouden bekijken. Fullerenen bevatten namelijk precies 12 vijfhoeken, dus indien we meer dan zes vijfhoeken in een patch hebben, zal in de andere helft van de fullereen minder dan zes vijfhoeken bevatten.

Een *deelpatch* P' van een patch P is een patch waarvoor geldt dat $V(P') \subseteq V(P)$ en $E(P') = E(P) \cap \{\{a, b\} | a \neq b, a \in V(P'), b \in V(P')\}$. Voor een een gegeven patch en deelpatch is de *interne rand* van de deelpatch de 'rand' van een deelpatch waar deze grenst aan de patch. Het *verschil* $P - P'$ tussen een patch en een deelpatch zijn de veelhoeken die bevat zitten in P maar niet in P'.

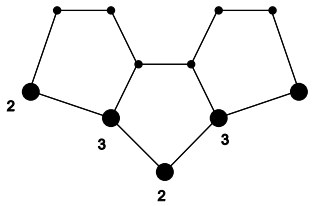
Een *pre-patch* is net zoals een gewone patch een ingebedde 2-samenhangende graaf. Alle vlakken zijn vijf-of zeshoeken, met uitzondering van een of twee vlakken: het onbegrensde vlak en eventueel een enkel intern vlak. Ook hier zijn alle 'interne' toppen van graad drie, en hebben randtoppen (de toppen die grenzen aan het onbegrensde vlak of het intern vlak) een graad van twee of drie.

Hoofdstuk 3

Symmetrieën

3.1 Symmetrie van de rand

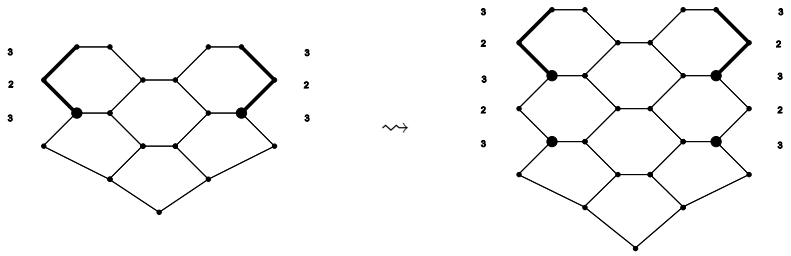
Een interessante vraag waarvan het antwoord later zeer nuttig zal blijken, is of er voor arbitrair grote n ($n \geq 5$) een patch bestaat die een rand heeft van lengte n, en waarvan die rand een symmetriegroep heeft met ten minste n elementen. Een eenvoudige manier om dit aan te tonen, is om een rand van de vorm '$(23)^n$' te beschouwen. Deze rand bevat ten minste n symmetrieën, namelijk n cyclische verschuivingen van de rand. Dit gecombineerd met een spiegeling, levert een totaal van $2n$ symmetrieën. Dus rest ons nog aan te tonen dat voor elke $n \geq 5$, er een invulling bestaat voor zo'n rand. We zullen dit probleem opsplitsen in 2 deelproblemen, namelijk randen van de vorm '$(23)^{2n}$' en '$(23)^{2n+1}$'. Eerst tonen we een lemma hiervoor aan.



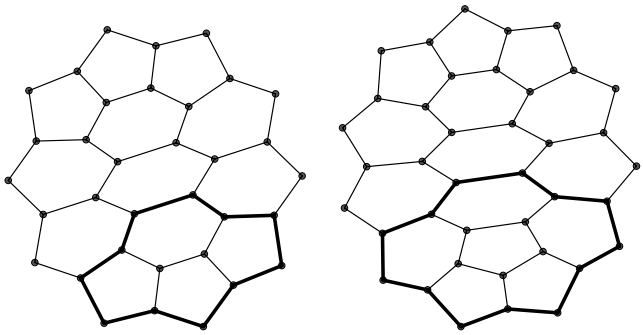
Figuur 3.1: Startpatch van de constructie

Lemma 3.1.1. *Voor elke $n \geq 1$ bestaat er een patch met rand '$(23)^{2n}2223322$'.*

Bewijs. Beschouw eerst de patch in figuur 3.1. Deze heeft een rand van de vorm '$(23)^{2\cdot1}2^33^22^2$'. Deze kunnen we uitbreiden naar een patch met rand '$(23)^{2\cdot2}2^33^22^2$', waarbij in figuur 3.2 wordt getoond hoe dit kan. Hierna kunnen we inductief deze constructie blijven gebruiken. Het is onmiddellijk duidelijk dat we inductief elk zo'n patch waar de rand '$(23)^{2\cdot n}2^33^22^2$' is, kunnen



Figuur 3.2: Manier om de patch uit te breiden met drie zeshoeken



Figuur 3.3: (a) Een patch met rand '$(23)^{10}$' (b) Een patch met rand '$(23)^{11}$'.

uitbreiden naar een patch met als rand '$(23)^{2\cdot(n+1)}2^3 3^2 2^2$' door 3 zeshoeken aan de patch toe te voegen. □

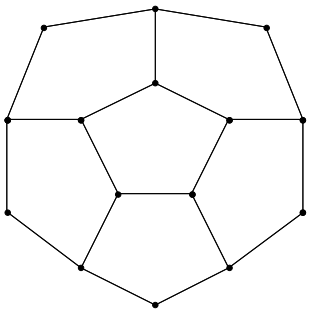
Dit lemma kunnen we dan gebruiken om de stelling aan te tonen in de 2 vooropgestelde delen:

Stelling 3.1.2. *Voor elke $n \geq 5$ bestaat er een patch met rand '$(23)^n$'.*

Bewijs. We starten met patches met rand '$(23)^{2n}$'. Hiervoor nemen we eenvoudig een patch met rand '$(23)^{2n-4}2^3 3^2 2^2$'. Dit kan gedaan worden zoals beschreven in Lemma 3.1.1 voor $n \geq 3$. We kunnen nu de discrepantie tot '$(23)^{2n}$' weghalen, door op deze patch een kleine hulppatch te plaatsen met rand '$2223322(23)^2$', die bestaat uit 3 vijfhoeken en een zeshoek, zoals voorgesteld in figuur 3.3a. We bekomen zo een patch met als rand '$(23)^{10}$' in het geval van figuur 3.3a. Dit kan dus ook voor andere n.

Dit kunnen we dan analoog doen voor het geval met rand '$(23)^{2n+1}$', waar we met dezelfde patch starten, maar er een patch tegenplakken bestaande uit 3 vijfhoeken en 3 zeshoeken. Dit is te zien in figuur 3.3b. Zo bekomen we bijvoorbeeld de patch uit figuur 3.3b met rand '$(23)^{11}$'.

Dan rest ons nog het geval '$(23)^5$'. We kunnen hier gewoon een voorbeeld van geven, namelijk figuur 3.4. Dit is een patch met als enige interne vlakken precies de 6 vijfhoeken, en een rand van '$(23)^5$'. □



Figuur 3.4: Een patch met rand '$(23)^5$'

Nu we weten dat er inderdaad patches bestaan waarvan de rand een arbitrair grote symmetriegroep heeft, is het ook interessant om te weten of er een eenvoudige voorwaarde bestaat die kan zeggen of een gegeven rand zeker *niet* zo'n grote symmetrie bevat.

Elke symmetriegroep van de rand zal bestaan uit ten minste een cyclische verschuiving, die ten hoogste b symmetrieën zal voortbrengen, met b de randlengte. Dit kan dan eventueel nog gecombineerd worden met een spiegeling. Deze verschuiving zal steeds een stuk rand opschuiven. Met andere woorden, de rand zal een cyclische opeenvolging van een kleiner stukje rand moeten zijn. Met deze informatie kunnen we nu een eenvoudige eigenschap bewijzen.

Lemma 3.1.3. *Indien de rotatiedeelgroep van de symmetriegroep van de rand van een patch met $p \leq 6$ meer dan 6 elementen bevat, dan is het aantal vijfhoeken in die patch met die rand $p = 6$.*

Bewijs. Neem een voortbrengend element c van de deelgroep van de rotaties van de symmetriegroep, dat zo gekozen is dat er zo weinig mogelijk toppen zitten op het pad van elke top naar zijn afbeelding onder c. Bbeschouw een willekeurige top v van de rand, en laat v' zijn beeld zijn onder de rotatie c. Dan noemen we A het stuk van de rand van v tot v' dat minimale lengte heeft. Nu kan een rand met n rotaties (inclusief identiteitsrotatie) in de symmetriegroep er uitzien als A^n, met $n > 0$. We kunnen hiervoor het aantal vijfhoeken berekenen met de $v_2 - v_3 = 6 - p$ formule als volgt: $p = 6 - (v_2 - v_3) = 6 - n \cdot (v_{2,A} - v_{3,A})$. We zullen deze gelijkheid van p met een functie die afhangt van n nu gebruiken samen met grenzen op p. Op die manier gaan we grenzen op n bekomen.

We hebben 2 grenzen op p. De eerste is de gegeven grens van $p \leq 6$. Die levert nu op dat

$$6 \geq 6 - n \cdot (v_{2,A} - v_{3,A}) \Leftrightarrow n \cdot (v_{2,A} - v_{3,A}) \geq 0$$

Aan de andere kant levert de natuurlijke grens $p \geq 0$ samen met de eerder bekomen $p =$

$6 - n \cdot (v_{2,A} - v_{3,A})$ op dat

$$6 - n \cdot (v_{2,A} - v_{3,A}) \geq 0 \Leftrightarrow 6 \geq n \cdot (v_{2,A} - v_{3,A})$$

We onderscheiden nu 3 gevallen

1. $(v_{2,A} - v_{3,A}) < 0$: Gezien $n > 0$ is, zal nooit aan de ongelijkheid $n \cdot (v_{2,A} - v_{3,A}) \geq 0$ voldaan zijn. Dit geval zal zich dus nooit voor doen met de gestelde grenzen.

2. $(v_{2,A} - v_{3,A}) > 0$: Ook hier gebruiken we $n > 0$: indien $(v_{2,A} - v_{3,A}) = 1$ dan kan n ten hoogste 6 zijn, anders is niet meer voldaan aan $6 \geq n \cdot (v_{2,A} - v_{3,A})$; indien het verschil groter is zal n zelfs nog kleiner zijn. n zal dus niet onbeperkt kunnen zijn: zodra $n > 6$ geldt dit geval niet meer.

3. $(v_{2,A} - v_{3,A}) = 0$: Nu krijgen we dat $6 \geq 0 \geq 0$, wat altijd waar is. In dit geval is dus $p = 6$.

□

Opmerking 3.1.4. *In een patch wordt elk automorfisme vastgelegd door de actie van dit automorfisme op één enkele 'gerichte boog', en een draairichting.*

Bewijs. In een patch wordt door een automorfisme elke vijfhoek op een vijfhoek, en elke zeshoek op een zeshoek afgebeeld. Als we dus een starttop hebben en een adjacente doeltop, kunnen we hun beeld bepalen. Deze zal in exact twee vlakken liggen (waarvan ten hoogste één het buitenvlak kan zijn). We bekijken het rotatiesysteem. Neem het vlak horende bij de gekozen toppen. De afbeelding van dit vlak onder het automorfisme kan ten hoogste één van 2 verschillende vlakken zijn. De initiëel opgegeven draairichting zal dan bepalen welk van de twee vlakken zal worden gekozen. Nu is dus de afbeelding van een heel vlak bepaald onder dit automorfisme. Aangezien een patch 2-samenhangend is en het intern aantal buren steeds 3 is, zal de afbeelding van elk vlak kunnen worden bepaald door iteratief de afbeelding van vlakken te bepalen die grenzen aan reeds bepaalde vlakken. □

Gevolg 3.1.5. *Het aantal symmetrieën van de hele patch is altijd kleiner of gelijk aan het aantal symmetrieën van de rand.*

Bewijs. Een symmetrie van de patch zal sowieso de rand op de rand afbeelden. Stel nu dat er meer symmetrieën in de patch zitten dan in de rand. Dan zijn er minstens twee *verschillende*

symmetrieën van de patch, die echter eenzelfde symmetrie van de rand zullen bevatten. Gezien opmerking 3.1.4 weten we echter dat dan de symmetrie van de hele patch vast moet liggen. Dit is een contradictie. □

Opmerking 3.1.6. *Als bij een patch $p = 6$ is, dan blijft het aantal symmetrieën van de patch zelf nog steeds beperkt.*

Bewijs. Dit komt doordat het aantal symmetrieën zal afhangen van het aantal vijfhoeken, indien deze aanwezig zijn. Alle vijfhoeken moeten namelijk op een vijfhoek worden afgebeeld. Indien $p > 0$, zal een bovengrens voor het aantal symmetrieën dat wordt bepaald door het aantal vijfhoeken, kunnen worden berekend door $2 \cdot 2 \cdot 5 \cdot p = 20p$. We gebruiken hiertoe opmerking 3.1.4. Merk op dat we later een betere grens zullen zien.

Hier wordt dus het aantal mogelijkheden bepaald door de manieren waarop 1 boog van een specifieke vijfhoek kan worden afgebeeld op een willekeurige boog van één van de vijfhoeken, en waarbij de eerste factor 2 de mogelijke oriëntaties van de boog meerekent, en de tweede factor 2 de reflecties in rekening brengt. Aangezien de maximale symmetrie $2b$ is, met b de randlengte, is het voordeliger om de symmetrieën van de vijfhoeken te beschouwen in plaats van de rand indien $10p \leq b$. Gezien in bovenstaand geval $p = 6$, wordt dit dus $b \geq 60$. □

3.2 Symmetrie van een patch

3.2.1 De Brouwer fixpunt stelling

Een *fixpunt* (ook wel *dekpunt* genoemd) van een functie is een waarde van het domein van de functie, waarvoor het functieresultaat dezelfde waarde oplevert. Anders gezegd, x is een fixpunt van f als en slechts als $x = f(x)$.

Brouwer fixpunt stelling. *Elke continue functie die de gesloten eenheidsbol in $\mathbb{R}^n$ op zichzelf afbeeldt, bevat ten minste één fixpunt.*

3.2.2 Toepassing op een patch

We kunnen nu deze fixpuntstelling toepassen op een automorfisme van de rand van een patch P. De rotaties van de rand zijn een cyclische deelgroep, die wordt voortgebracht door een element c van de automorfismegroep. Beschouw nu $f(v) = c(v)$, voor $v \in V(P)$. We kunnen deze functie nu als volgt 'uitbreiden' naar een functie f' die elk punt in de inbedding van de patch afbeeldt op een (ander) punt in de patch.

Zoals reeds gezegd, noemen we een isomorfisme tussen twee vlakke grafen *combinatorisch* indien er een bijectie tussen de toppenverzamelingen van de grafen bestaat die buren op buren afbeeldt, en die kan worden uitgebreid zodat het niet alleen het aangrenzend zijn van toppen bewaart, maar ook het aangrenzend zijn van vlakken met toppen en bogen.

Verder noemen we een combinatorisch isomorfisme tussen twee vlakke grafen *topologisch* indien er een homeomorfe uitbreiding bestaat van dit isomorfisme. Dit wil zeggen dat er een continue afbeelding van *héél* het vlak (of het boloppervlak) bestaat, die de toppen, bogen en vlakken van beide grafen op mekaar afbeeldt, zó dat de adjacentie ervan bewaard blijft.

Stelling (Theorema 4.3.1 (ii) in [Die00])**.** *Elk combinatorisch isomorfisme tussen twee 2-samenhangende vlakke grafen is topologisch.*

Aangezien patches 2-samenhangend zijn, en een automorfisme van een patch per defintie combinatorisch is, weten we dat er voor elk automorfisme van de patch een homeomorfisme bestaat dat heel het vlak op zichzelf zal afbeelden, zodanig dat de toppen, bogen en vlakken van de patch worden afgebeeld op hun beelden onder het automorfisme. Nu kunnen we de stelling van Brouwer toepassen op deze functie (aangezien de patch een samenhangend geheel is van vijf- en zeshoeken, kunnen we deze eerst *omvormen* tot een eenheidsbol).

Voor f' moet er dus een fixpunt bestaan zodat $f'(x) = x$. Aangezien nu de rotaties worden voortgebracht door dit enkele element c, hebben we dus dat elke rotatie het eindresultaat zal zijn van een aantal maal c toe te passen. We hebben dus ook voor de uitgebreide functie dat $f'(\dots f'(f'(x))) = f'(\dots f'(x)) = \dots = x$. Er is dus een fixpunt dat gelijk is voor elke rotatie.

Stelling 3.2.1. *Een patch heeft ten hoogste 12 automorfismen.*

Bewijs. Nemen we f een afbeelding op $V(P)$ die een rotatie van de rand is, met $f(v) = c(v)$, waar c een voortbrengend element is voor de rotaties van de rand. Laat dan f' de afbeelding zijn die wordt verkregen door f uit te breiden naar de hele inbedding van P. Nu kan een fixpunt x van een patch P onder f' in drie algemene gevallen worden onderverdeeld.

1. x is een top die in de toppenverzameling $V(P)$ zit. Nu weten we dat in een patch een top ten minste 2 en ten hoogste 3 buren heeft. We kunnen in het rotatiesysteem dus alle bogen bepalen die 'vertrekken' in x. Aangezien een automorfisme van een patch wordt bepaald door de afbeelding van een 'gerichte' boog en een richting, kunnen we de bogen bekijken die x als een van de twee eindpunten hebben. Aangezien x vastligt onder f' kunnen we alleen nog de draairichting en de buur van x kiezen om f vast te leggen. Aangezien bovendien

f hier een automorfisme is dat cyclisch wordt voortgebracht, ligt de draairichting van het rotatiesysteem reeds vast. Dus is de bovengrens voor het aantal automorfismen dat f zal voortbrengen beperkt tot het aantal buren dat we kunnen kiezen. In het slechtste geval is dit dus 3.

2. x bevindt zich op een boog van $E(P)$. Net zoals in het vorige geval kunnen we nu een bovengrens bepalen voor het aantal automorfismen dat f voortbrengt. Aangezien we reeds een boog hebben, kunnen we enkel de oriëntatie van de boog en de richting kiezen. We bekomen dan dat f' ten hoogste 4 verschillende automorfismen zal voortbrengen. Maar net als hierboven ligt de de draairichting reeds vast, en zal het aantal automorfismen dat f voortbrengt maximaal 2 is.

3. In alle andere gevallen zal x zich in een vlak van P bevinden. Dus onder het automorfisme zal het vlak waar x in ligt steeds worden afgebeeld op zichzelf. Indien dit vlak een n-hoek is, zal f' dus ten hoogste n verschillende automorfismen genereren. Aangezien x wordt verondersteld binnen P te liggen, zal dit dus een vijfhoek of een zeshoek zijn. We hebben dus een bovengrens van 6 automorfismen.

Aangezien in alle beschreven gevallen het aantal automorfismen dat f voortbrengt kleiner dan of gelijk aan 6 is, besluiten we dat er ten hoogste 6 worden voortgebracht door f. Omdat we reeds een cyclische automorfisme hebben, kan dit herhaald worden voor het automorfisme na een mogelijke spiegeling (indien deze bestaat). Als we nu eerst P spiegelen, en dan er weer f op toepassen, zullen dus ten hoogste 6 automorfismen meer worden gegenereerd. In totaal zijn er dus ten hoogste 12 automorfismen van een patch. □

Hoofdstuk 4

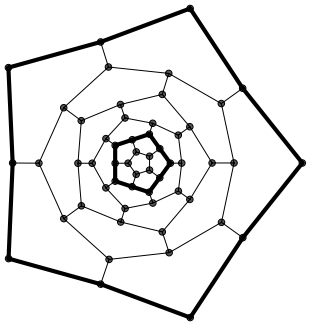
Oneindige lussen

Het is mogelijk dat er een patch bestaat met een interne rand die gelijk is aan de rand van de patch zelf. Of analoog, er bestaan patches waarbij we twee gelijke interne randen hebben, en waar de ene rand 'binnenin' de andere zit. We nemen natuurlijk aan dat het verschil in de patch tussen deze binnen- en buitenrand minstens één vijf- of zeshoek bevat. Dit is een situatie waar we later zullen voor moeten opletten bij het ontwerpen van een algoritme. Anders lopen we het risico dat tijdens het vullen van een rand, we eenzelfde rand tegenkomen als degene die we al probeerden in te vullen. We zouden zo een oneindige lus bekomen, iets wat we natuurlijk moeten vermijden. Dit is geen probleem van het algoritme, maar wordt veroorzaakt door de aard van het vraagstuk: het is mogelijk dat er gewoonweg geen eindige verzameling invullingen is. De vraag zal zijn óf, en op welke manier we hier dan rekening gaan houden met deze situatie. We geven hier een aantal opmerkingen die ons later zullen helpen hier mee om te gaan in het algoritme.

We beginnen met een eenvoudige opmerking over deze situatie.

Opmerking 4.0.2. *Een patch kan slechts een interne rand hebben die gelijk is aan de rand van de patch, indien het verschil tussen de interne rand en de buitenste rand van de patch enkel zeshoeken bevat, en dus geen enkele vijfhoek.*

Bewijs. Dit komt doordat de formule die het aantal vijfhoeken bepaalt in een patch enkel afhankelijk is van de rand zelf: $v_2 - v_3 = 6 - p$. Stel namelijk dat b de buitenste rand is, en dat b' de interne rand die gelijk is aan b, dan hebben we: $v_{2,b} - v_{3,b} = 6 - p = v_{2,b'} - v_{3,b'}$. Aangezien er in de patch zelf even veel vijfhoeken moeten zitten als in de deelpatch met rand b', kan er dus geen vijfhoek zitten tussen b en b'. □



Figuur 4.1: Een rand die kan worden aangevuld met zeshoeken zodat er een binnenrand ontstaat die dezelfde is als de buitenrand.

We kunnen reeds een eis opleggen waaraan een rand moet voldoen om in deze situatie te kunnen komen. Het is namelijk zo dat dit slechts zal kunnen voorkomen indien $p \geq 6$. Indien $p = 6$ kunnen we zeer eenvoudig een voorbeeld vinden waar dit het geval is. Neem bijvoorbeeld de patch afgebeeld in figuur 4.1. We zien hier dat de buitenste rand dezelfde vorm heeft als de vet getekende binnenrand. Nu is het binnenste van deze rand ook een patch. We kunnen dus het stuk tussen de twee vetgedrukte randen zovaak als we maar willen hier tussen plakken.

Een manier om dit soort patches te verkrijgen is de volgende. We nemen een eenvoudige nanotube: een fullereen in de vorm van een dunne draad, die aan beide 'uiteinden' begrensd wordt door 2 halve sferen met elk een aantal vijfhoeken in. We kunnen deze draad nu halverwege doorsnijden, en er een ring van zeshoeken tussenplakken. Dit kan zo gedaan worden dat de rand gelijk blijft. Indien we dit nu bekijken vanuit het standpunt van patches, is elke verlenging die we in de draad maken met een extra ring zeshoeken toe te voegen een patch waarbij we een binnenrand hebben die gelijk is aan de buitenrand. In principe is dit zelfs de manier waarop figuur 4.1 gegenereerd werd: meerbepaald door een programma dat zogenaamde 'tubetype fullerenen' genereert.

In het geval van $p > 6$ zal het dan bijvoorbeeld kunnen gebeuren dat we een vijfhoek plaatsen (bijvoorbeeld aan de rand), zodat we een binnenrand krijgen die de vorm heeft van figuur 4.1, waardoor we ook in eenzelfde situatie komen. Dit zal dus ook een probleem kunnen geven. Nu rest ons nog te bewijzen dat indien we hebben dat $p < 6$, dit niet voor kan komen.

Lemma 4.0.3. *Indien we een rand met invulling hebben waarvoor $p < 6$ is, dan zal er geen binnenrand zijn die gelijk is aan de buitenrand.*

Bewijs. Indien een rand een invulling bevat, geeft [JBG03] formules die een grens op het aantal

zeshoeken geeft voor een bepaalde randlengte. Voor $p < 6$ worden expliciete bovengrenzen gegeven. Stel nu dat er een rand met invulling bestaat waarvoor $p < 6$ geldt, en waar een binnenrand gelijk is aan de buitenrand. Dan kunnen we als volgt te werk gaan. We bepalen het maximaal aantal zeshoeken voor de rand. Indien het verschil tussen de rand en de binnenrand reeds meer zeshoeken bevat, bekomen we dat het aantal zeshoeken van de totale patch reeds groter is dan het maximum, een contradictie.

Indien echter het maximum nog niet is overschreden door de patch, kunnen we aan de binnenrand opnieuw hetzelfde aantal zeshoeken toevoegen, zodat we weer dezelfde binnenrand bekomen. Deze operatie herhalen we, totdat het maximum overschreden is. Ook dit levert dan uiteindelijk een contradictie op, zodat de gestelde invulling niet kan bestaan. □

Indien er echter geen invulling van de rand bestaat, zal een ander bewijs gebruikt moeten worden. Er moet worden aangetoond dat indien $p < 6$, er geen pre-patch met tenminste een zeshoek in bestaat zodanig dat de binnenrand gelijk is aan de buitenrand. Of deze binnenrand een echte invulling bevat is niet van belang. Dit kan gedaan worden met een bewijs uit het ongerijmde. We stellen dat zo'n rand zou bestaan, waaruit we dan weer afleiden dat de gestelde situatie niet kan bestaan. We beginnen met een paar lemma's.

Lemma 4.0.4. *Zij P een prepatch met $p < 6$, binnenrand r en buitenrand R gelijk aan de binnenrand, met minstens een zeshoek tussen r en R. Dan kan P uitgebreid worden met zeshoeken zodanig dat de binnenrand niet aan de buitenrand grenst en dat de nieuwe binnenrand nog altijd gelijk is aan R.*

Bewijs. Indien r en R sowieso al niet aan mekaar grenzen, zijn we reeds klaar. We beschouwen het geval waar dit dus niet zo is.

We beschouwen nu alle bogen van R die ook op r liggen. Indien we al deze bogen weglaten uit de prepatch, ontstaat een verzameling van patches (niet noodzakelijk een enkele patch). Dit komt doordat een prepatch ten hoogste een enkel intern vlak heeft, dat in dit geval begrensd werd door vijfhoeken, zeshoeken en bogen die op de binnen- én buitenrand liggen. Merk verder op dat de randlengte van elk van deze patches kleiner zal zijn dan twee keer de randlengte van de prepatch. Verder is ook het aantal patches dat door deze actie ontstaat ook begrensd in functie van de randlengte van de prepatch.

We kunnen P nu met het verschil tussen R en r (dat minstens een enkele zeshoek bedraagt) aanvullen tot een prepatch P' waarbij nog steeds de binnenrand gelijk is aan de buitenrand, en die meer zeshoeken bevat dan P. Dit proces kunnen we zovaak herhalen als we willen. Merk

dus op dat elke iteratie van dit proces het aantal zeshoeken in de prepatch met ten minste één zal verhogen.

Dit proces passen we dan herhaaldelijk toe, zodat we een reeks prepatches $P, P^{(1)}, P^{(2)}, ...$ bekomen. Dit zal er uiteindelijk voor zorgen dat we een prepatch $P^{(n)}$ hebben kunnen construeren uit de oorspronkelijke prepatch P, met buitenrand R gelijk aan de binnenrand $r^{(n)}$, maar waar deze randen in geen enkele top aan mekaar grenzen. Stel namelijk dat dit *niet* zo zou zijn. Na elke uitbreiding van de vorige prepatch met minstens een zeshoek (zodat dus de binnenrand gelijk is aan de buitenrand), zal deze zoals beschreven kunnen uiteenvallen in een op voorhand bepaald aantal patches (door de bogen waar de binnenrand gelijk is aan de buitenrand te verwijderen). Aangezien dit aantal patches nu onafhankelijk van het aantal iteraties van dit proces begrensd is, net zoals de randlengte van de individuele patches begrensd is, kunnen we ook hier grenzen bepalen voor het aantal zeshoeken in elk van deze patches dankzij [JBG03]. Aangezien we dit proces zovaak kunnen herhalen als we willen, zal ooit een patch ontstaan die niet aan deze conditie voldoet. Dit levert de gevraagde contradictie. □

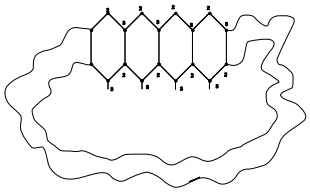
We weten nu dat indien we een prepatch P met de binnenrand r gelijk aan de buitenrand R hebben, we deze kunnen uitbreiden tot een prepatch waarvan de binnenrand gelijk is aan de buitenrand, maar waar ze mekaar echter niet aanraken. Voor we nu het hoofdbewijs geven, voeren we eerst nog een extra definitie in.

Een *ring* van zeshoeken is een 2-samenhangende graaf in een patch of pre-patch met minstens twee begrensde vlakken, en met alle begrensde vlakken op één na een zeshoek, zodat de duale graaf geïnduceerd door de zeshoeken precies één cykel is. Hieruit volgt dat elke zeshoek ten hoogste aan twee andere zeshoeken grenst, en dat elke top ten hoogste graad 3 heeft. We duiden de buitenrand van een ring r aan met $r_\uparrow$, terwijl we de binnenrand aanduiden met $r_\downarrow$. Analoog duiden we de buitenrandlengte aan met $l_\uparrow$ en de binnenrandlengte met $l_\downarrow$

Lemma 4.0.5. *In een ring van zeshoeken r geldt dat $l_\downarrow = l_\uparrow - 2(6-p)$, met $p = 6 - v_{2,\uparrow} + v_{3,\uparrow}$.*

Bewijs. We noemen het aantal zeshoeken in de ring $|H|$. Indien we een draairichting rond de zeshoekenring beschouwen, zien we dat elke zeshoek op één boog met zijn 'rechterbuur' grenst (omdat we in en pre-patch werken). Op elk van deze plaatsen zit dus in de buitenrand een top met graad 3, terwijl de andere eindtop van de verbinding tussen binnen- en buitenrand, aan de binnenrand van de ring, graad 2 zal hebben. We hebben dus dat $v_{2,\downarrow} = |H|$ en $v_{3,\uparrow} = |H|$.

We willen nu de lengte van de binnenring berekenen uit het aantal zeshoeken in de ring. We beginnen met van elke zeshoek elke top te tellen, wat ons $6|H|$ toppen oplevert. Hierdoor



Figuur 4.2: Een ring van zeshoeken waarbij de binnenrand gelijk is aan de buitenrand. Merk op dat $v_{2,\downarrow} = H = v_{3,\uparrow}$.

tellen we duidelijk teveel: de buitenrand wordt bijvoorbeeld meegerekend, en die willen we niet hebben. We moeten hier dus een totaal van $l_\uparrow = v_{2,\uparrow} + v_{3,\uparrow}$ van aftrekken. Dan hebben we echter nog steeds de buitenste toppen met graad 3, en de binnenste met graad 2 dubbel geteld. We bekomen dus dat $l_\downarrow = 6|H| - (v_{2,\uparrow} + v_{3,\uparrow}) - v_{3,\uparrow} - v_{2,\downarrow}$.

We werken de oorspronkelijke formule nu als volgt uit:

$$\begin{aligned} 6v_{3,\uparrow} - (v_{2,\uparrow} + v_{3,\uparrow}) - 2v_{3,\uparrow} &= 3v_{3,\uparrow} - v_{2,\uparrow} \\ &= v_{3,\uparrow} + 2v_{3,\uparrow} - v_{2,\uparrow} \end{aligned}$$

Die term $2v_{3,\uparrow}$ kunnen we nu vervangen door gebruik te maken van de gelijkheid $v_{3\uparrow} = v_{2,\uparrow} - (6-p)$:

$$\begin{aligned} v_{3,\uparrow} + 2v_{3,\uparrow} - v_{2,\uparrow} &= v_{3,\uparrow} + v_{2,\uparrow} - 2(6-p) \\ &= l_\uparrow - 2(6-p) \end{aligned}$$

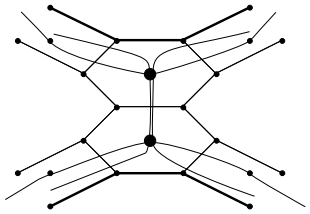
Wat het gestelde resultaat is. □

Gevolg 4.0.6. *Indien in een ring geldt dat $p < 6$, dan is $l_\downarrow < l_\uparrow$.*

Bewijs. Dit is een direct gevolg van het vorige lemma, doordat in dit geval $6 - p > 0$ is. □

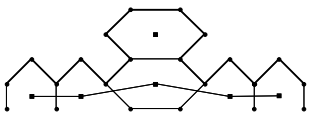
Lemma 4.0.7. *Uit een prepatch P met buitenrand $r_\uparrow$ gelijk aan de binnenrand $r_\downarrow$ met randlengte l en waarbij $r_\uparrow$ en $r_\downarrow$ elkaar niet raken, kunnen we een prepatch P' construeren waarbij de buitenrand $r'_\uparrow$ gelijk aan de binnenrand $r'_\downarrow$ met randlengte $l' \leq l$ en zó dat de zeshoeken die grenzen aan $r'_\uparrow$ een ring vormen.*

Bewijs. Indien de zeshoeken die aan de rand grenzen reeds een ring vormen, is het resultaat triviaal. Indien deze zeshoeken geen ring vormen, wil dit zeggen dat er niet precies één cykel zit in de door de randzeshoeken geïnduceerde duale graaf. We kunnen nu de patch zo aanpassen, dat dit wel het geval zal zijn.



Figuur 4.3: Stuk van een patch met 3 cykels in de door de randzeshoeken geïnduceerde duale graaf

We kijken eerst naar het geval waar er minstens 2 cykels bevat zitten. Dit is een situatie zoals voorgesteld in figuur 4.3. De binnenrand $r_{\downarrow}$ zal volledig bevat zitten in een van deze cykels (dit is een gevolg van de Jordan curvestelling). We kunnen dan eenvoudigweg het stuk van de buitenrand 'wegsnijden' dat buiten deze binnenrand-bevattende cykel valt. We passen de binnenrand dan natuurlijk op gelijkaardige manier aan, zodat de nieuw bekomen randen gelijk blijven. Deze actie zal de nieuw verkregen rand nooit langer laten worden dan de oude rand.



Figuur 4.4: Stuk van een patch waar de cykel in de door de randzeshoeken geïnduceerde duale graaf niet alle zeshoeken bevat

Het andere geval is waar niet alle randzeshoeken in de cykel zullen liggen van de door hen geïnduceerde duale graaf. Dit is een situatie zoals voorgesteld in figuur 4.4. Ook hier kan eenvoudig de rand worden aangepast zodat deze zeshoek wordt geëlimineerd, op een manier die er voor zal zorgen dat de nieuwe binnenrand gelijk zal blijven aan de nieuwe buitenrand, en de randlengte niet groter zal zijn dan de oorspronkelijke randlengte. □

Stelling 4.0.8. *Er bestaan geen prepatches met $p < 6$ met minstens één zeshoek tussen de binnen- en buitenrand, waarbij de binnenrand gelijk is aan de buitenrand.*

Bewijs. Stel dat er inderdaad randen bestaan met $p < 6$ waarbij er een prepatch van kan worden gemaakt met minstens één zeshoek en waarbij de binnenrand gelijk is aan de buitenrand. Dan kijken naar wij naar de verzameling van al deze randen, en nemen een rand met de kleinst mogelijke randlengte uit deze verzameling. Noem deze R_{min}, met lengte l_{min}.

Dankzij lemma 4.0.4 weten we nu dat deze rand kan worden ingevuld tot een prepatch P waarbij de binnenrand volledig van de buitenrand wordt gescheiden door een aantal zeshoeken.

Nu kunnen we zonder de algemeenheid te schaden uit deze prepatch P een ring van zeshoeken halen die we r noemen, en die de buitenrand gemeen heeft met de buitenrand van P. Indien we dat namelijk niet kunnen doen, kunnen we een nieuwe P construeren die nog steeds voldoet aan de voorwaarden dankzij lemma 4.0.7. De randlengte van de patch die uit die constructie komt, zal ofwel gelijk zijn aan l_{min}, waarna we gewoon door kunnen gaan, ofwel strikt kleiner zijn. Dit zou echter de minimaliteit van R_{min} tegenspreken. Deze kopie zal dan ook elk van deze kleinere randen bevatten. Er is dan echter een rand met lengte kleiner dan l_{min} waar we een prepatch kunnen invullen zodat deze een binnenrand heeft gelijk aan de buitenrand. Dit is in contradictie met de minimaliteit van R_{min}.

We weten nu dat de lengte van de binnenrand $r_\downarrow$ van deze ring strikt kleiner zal zijn dan de lengte van de buitenrand. Indien de lengte niet strikt kleiner zou zijn, is dit in contradictie met het gestelde dat $p < 6$.

Noem de lengte van deze binnenzijde $l_\downarrow$, terwijl de lengte van de buitenzijde $r_\uparrow$ de lengte l_{min} van de rand R_{min} is. Dus is $l_\downarrow < l_{min}$. Nu kunnen we P verder invullen door tegen de binnenrand van P, een kopie van P te hechten die we P' noemen. Nu kunnen we in P' ook de kopie van de ring r identificeren. Noem deze r'. Ook deze zal een binnenrand hebben met lengte $l_\downarrow$. Merk echter op dat we nu een situatie hebben dat er tussen $r_\downarrow$ en $r'_\downarrow$ een prepatch zit waarbij de binnenrand gelijk is aan de buitenrand met minstens één zeshoek er in. De lengte van deze rand is echter strikt kleiner dan de randlengte van R_{min}, wat de minimaliteit van deze rand tegenspreekt. Dit geval zal zich dus ook niet voordoen. We besluiten dus dat geen enkele mogelijkheid bestaat waarvoor een prepatch met binnenrand gelijk aan de buitenrand en $p < 6$ zou zijn, wat het gestelde aantoont. □

Hoofdstuk 5

De canonische vorm

Een probleem in de implementatie zal zijn dat we vaak isomorfe grafen zullen tegenkomen. Dit kan zowel in de uitvoer van het algoritme zijn, als in intermediair nodige lijsten met grafen. Dit laatste is bijvoorbeeld het geval in de implementatie van de catalogus met groeiparen voor fullerenen, zoals beschreven in subsectie 9.2. Dit zal dus belangrijk zijn voor het algoritme, omdat isomorfe resultaten in de lijst geen enkele nieuwe informatie zullen geven.

We willen natuurlijk geen dubbel werk doen, waardoor we graag isomorfe grafen willen wegfilteren. We kunnen dit doen door een *canonische vorm* van een graaf te introduceren. Een canonische vorm is een 'object' dat zó gekozen is, dat we deze voor elke graaf kunnen berekenen en zó dat de canonische vorm van twee grafen gelijk is als en slechts als de grafen ook isomorf zijn. Indien we dus 2 grafen hebben, kunnen we van elk de canonische vorm berekenen, deze dan vergelijken en beslissen of ze al-dan-niet isomorf zijn. Met een 'object' wordt een wiskundige voorstelling bedoeld die we liefst zo eenvoudig mogelijk op de computer kunnen voorstellen. In veel gevallen zal het dus een getal zijn, of een lijst van symbolen.

In het algemene geval zal zo'n canonische vorm berekenen moeilijk zijn. Graaf-isomorfisme controleren is namelijk zeer waarschijnlijk een moeilijk probleem. Het is een zeer bekend probleem in de computationele complexiteit. De klasse van problemen die reduceerbaar zijn tot graaf-isomorfisme wordt in de computationele complexiteit ook wel **GI** genoemd. Het behoort tot de klasse **NP** van problemen, maar het is niet bekend of dit probleem **NP**-compleet is, in de klasse **P** zit, of daar tussenin zit. Dit zou natuurlijk betekenen dat $\mathbf{P} \neq \mathbf{NP}$, en dat **GI** niet **NP**-compleet is en geen polynomiale-tijd algoritme heeft.

Er zijn een aantal interessante aanwijzingen die doen vermoeden dat het op zijn minst niet **NP**-compleet is. Bijvoorbeeld, indien graaf-isomorfisme **NP**-compleet zou zijn, dan zou de

polynomiale-tijd hiërarchie in mekaar vallen ([Sch88], [BHZ87]).

Het is wel bekend dat bepaalde specifieke instanties van dit probleem wel degelijk in **P** zitten. Er zijn bijvoorbeeld polynomiale-tijd algoritmes voor isomorfisme-detectie van planaire grafen. In het bijzonder is er zelfs een algoritme van Hopcroft en Wong dat in lineaire tijd isomorfie van twee planaire grafen controleert ([HW74]). Het probleem van dit algoritme is echter dat nog niemand dit algoritme ook daadwerkelijk in lineaire tijd heeft kunnen implementeren. Dit wil zeggen dat het van bepaalde stappen in dit algoritme niet geweten is hoe deze daadwerkelijk moeten worden geïmplementeerd zodat een linair algoritme bekomen wordt. Een ander bekend polynomiaal algoritme is een algoritme dat isomorfe grafen zal herkennen indien er op voorhand een grens gezet is op de genus van het lichaam waarop de grafen inbedbaar zijn ([Mil80]).

5.1 Canonische vorm voor randen

In het specifieke geval van deze randen en patches is het echter relatief eenvoudig om een canonische vorm te berekenen. Aangezien we dankzij de opbouw van de code op voorhand altijd zullen weten welke toppen in de patch de rand voorstellen, zullen we in beide gevallen beginnen met naar de rand te kijken. We hebben reeds gezien in 3.1 dat een isomorfisme bepaald wordt door een 'gerichte boog' en een draairichting. Het probleem is nu om een object te bepalen aan de hand van deze informatie, zodat we deze patch of rand op een unieke manier kunnen voorstellen.

We beginnen met het maken van een canonische vorm voor randen. Deze is iets simpeler dan degene die we zullen gebruiken voor hele patches. In het bijzonder kunnen we zelfs de informatie van de canonische vorm van een rand gebruiken in die van de patch.

We gaan nu zelf alle verschuivingen en spiegelingen van de rand bekijken. Aan het resultaat van elk van deze acties gaan we een object koppelen dat het resultaat van deze actie zal representeren. Als we deze objecten dan ordenen, en we kiezen er het kleinste object uit, dan hebben we een unieke voorstelling voor deze rand. Immers, we hebben hiermee alle mogelijke isomorfismen van de rand overlopen, en aan elk hebben we een object gekoppeld dat *onafhankelijk* is van de oorspronkelijke toppenlabeling van de patch. Aangezien we daar dan een uniek bepaalbare representant voor kiezen, bekomen we een goede canonische vorm voor patches.

Het is dus de bedoeling om aan de hand van de informatie die we krijgen uit de rand, een unieke representatie te vinden voor deze rand. Hiertoe bekijken we de hele rand als een string. Indien we nu het effect van alle mogelijke cyclische verschuivingen en spiegelingen van deze rand

bekijken op de string, kunnen we de kleinst mogelijke string bepalen. Dit doen we door een karakter-per-karakter vergelijking te doen op deze strings.

We doen dit als volgt. Gegeven de oorspronkelijke rand, kunnen we elke positie in deze rand als begin van de string nemen, en dan cyclisch rond de rand lopen om de string te bepalen. We kunnen dit dan in de 2 mogelijke draairichtingen doen. De kleinste van deze strings wordt dan als canonische vorm van de rand genomen. Deze is duidelijk niet afhankelijk van de oorspronkelijke labeling van de rand: alle mogelijke startposities en richtingen worden immers bekeken.

Het probleem is hier echter dat we steeds ten hoogste evenveel karakters moeten vergelijken als de rand lang is, alvorens we kunnen beslissen of een bepaalde startpositie en richting een kleiner resultaat oplevert dan we totnogtoe hadden. Dit kan dus vrij traag zijn. Voor randen die kort genoeg zijn, kunnen we echter iets efficiënter werken in de implementatie. Het is namelijk zo dat we een rand van de vorm '23223...' ook kunnen zien als een binair getal '01001...' waarbij we een '2' door een '0' bit voorstellen, en een '3' door een '1' bit. Indien we nu de randlengte beperken tot de gangbare lengte in bits voor een `integer` of `long` datatype van de computer, kunnen we veel sneller een cyclische verschuiving van de rand vergelijken met de vorige. In de praktijk zullen we op een 64-bit processor dus gewoon in één enkele stap een randverschuiving uitvoeren, wat natuurlijk extreem efficiënt zal zijn.

We overlopen nu eerst de rand vanaf een willekeurige startpositie en leesrichting, en stoppen deze rand in een integer. Dan gaan we elke verschuiving van de randcode in slechts een paar instructies kunnen uitvoeren. Dit kan eenvoudig met een bitshift operator en een paar extra binaire bewerkingen. Dan kunnen we de resulterende code ook vergelijken met de voorgaande. Dit kunnen we dan eenvoudig herhalen voor de andere leesrichting. Dit zal potentieel veel sneller uitvoeren, omdat alle tijdelijke informatie in de CPU blijft. De echte complexiteit zal echter onveranderd blijven, aangezien we het verschuiven en vergelijken nog steeds element per element moeten doen in de CPU. Voor onze toepassingen zal dit echter zeer goed zijn, aangezien we in deze toepassingen geen randen tegenkomen die niet in 64 bit passen.

Nu we een canonische vorm bepaald hebben voor de rand, kunnen we hier een aantal interessante dingen mee doen. Niet alleen kunnen we nu twee randen op gelijkheid controleren aan de hand van deze canonische vorm, we kunnen ook een totaal andere toepassing bedenken.

Opmerking 5.1.1. *Tijdens het bepalen van de canonische vorm, wordt de canonische vorm even vaak gegenereerd als er automorfismen zijn van de rand. Elke configuratie die de canonische vorm genereert, komt overeen met een automorfisme van de rand.*

Bewijs. Elke keer een configuratie aanleiding geeft tot een canonische vorm, wil dit zeggen dat we deze specifieke configuratie niet kunnen onderscheiden van een canonische vorm. Dit kan natuurlijk alleen als dit dezelfde rand is, op isomorfie na. In het bijzonder kunnen we een van deze configuraties uitkiezen als *referentierand*. Elke 'gerichte boog' en richting die aanleiding zullen geven tot een canonische configuratie, kunnen dan worden gezien als de afbeelding van de 'gerichte boog' en richting van de referentierand. □

5.2 Canonische vorm voor patches

Nu we een canonische vorm beschreven hebben voor randen, wordt het tijd dat we er ook een voor patches definiëren. Eerst wordt het algemeen idee beschreven, waarna een optimalizatie zal worden toegevoegd die gebruik zal maken van de canonische vorm van de rand. Voor de canonische vorm van een patch gebruiken we natuurlijk het idee van isomorfe patches die moeten worden vergeleken.

Om de canonische vorm van een patch te vinden, hebben we eerst een afbeelding nodig tussen een patch en een object. Van die objecten kan dan de canonische vorm worden bepaald als het kleinste van de objecten. We bepalen de objecten nu als volgt. Een object zal het resultaat zijn van een zeer specifiek herbenoemen van de toppen van de graaf.

1. We overlopen elke mogelijke 'geordende boog' van de rand, en kiezen voor elk zo'n boog elke mogelijke draairichting (dat zijn er dus twee). We gaan nu proberen een labeling aan de toppen te geven die 'zo klein mogelijk' is, en die tegelijkertijd eenduidig bepaald is. Die 'zo klein mogelijk' slaat natuurlijk op het resultaat dat we zullen bekomen als object. We geven de starttop van deze boog het label 0, en de andere top van de boog het label 1. Nu draaien we rond de starttop. We draaien vanaf label 1 in de richting die we gekozen hebben. Elke top die we tegenkomen, geven we het volgende beschikbare label. In dit geval zal dit ten hoogste nog een enkele top zijn.

2. De representatie van de graaf onder deze actie wordt nu als volgt opgebouwd. Het is een lijst waarbij voor elke top plaats voorzien is voor de labels van zijn drie mogelijke buren. De eerste buur die op die manier wordt toegevoegd, zal steeds de buur met het kleinste label zijn. Dan draaien we rond de bogen van de starttop in de gekozen richting, en vullen in die volgorde de andere buren in. Randtoppen die slechts 2 buren hebben, zullen een apart gekozen waarde krijgen op de plaats waar de 'aansluiting' naar buiten is.

3. Elke keer we een de buren van een top in de lijst hebben geschreven, gaan we verder met de top met kleinst mogelijke labeling waarvan de buren nog niet aan deze lijst zijn toegevoegd. We kijken dan weer naar de buur van deze top die het kleinst mogelijke label heeft, en draaien dan weer in de gekozen richting. Buurtoppen die reeds een label hebben gekregen behouden hun label, de anderen krijgen het kleinst mogelijke label dat nog niet is uitgedeeld. Ook hier stoppen we de labelings van de buren in de representatie.

4. Als alle buren van alle toppen in de tabel zitten, kunnen we deze representatie vergelijken met reeds eerder gegenereerde representaties. We beschouwen hiertoe de lijst met buren-labelings als een lijst getallen, en gebruiken een lexicografische ordening om deze lijsten te vergelijken.

Het bepalen van een canonische vorm voor een patch kan echter gebruik maken van de informatie die we kunnen halen uit het bepalen van de canonische vorm van de rand. We weten dat door het bepalen van deze canonische vorm, we ook informatie krijgen over alle isomorfismen van de rand. In plaats van alle mogelijke startposities en richtingen te proberen, kunnen we alleen starten met de startposities en richtingen die we bekomen uit de informatie van het bepalen van de canonische vorm van de rand.

Aangezien ook deze informatie onafhankelijk is van enige labelling die we geven aan de toppen, weten we ook hier dat de canonische vorm nog steeds onafhankelijk zal zijn van de labelling. Bovendien, indien we reeds de canonische vorm en isomorfismen van de rand op voorhand hebben bepaald, zal dit mogelijk veel sneller zijn dan de vorige canonische vorm. Veel randen zullen namelijk slechts een beperkt aantal automorfismes hebben, zodat ook het aantal startposities sterk zal beperkt worden.

We kunnen nu ook kijken naar de complexiteit van dit algoritme. Dit is een zeer belangrijke vraag, aangezien we dit algoritme in elke toepassing zullen moeten gebruiken om isomorfe patches te kunnen wegfilteren. Indien dit algoritme te traag zou zijn voor grotere patches, zouden we bijvoorbeeld het probleem kunnen hebben dat het te filteren te traag zou worden. Gelukkig is dit algoritme snel genoeg, en dit kunnen we bewijzen.

De complexiteit van dit algoritme kunnen we bekijken vanuit het oogpunt van het totale aantal toppen dat in de patch zit. Nu kunnen we aantonen dat het algoritme in $O(n)$ tijd zal lopen, waarbij n het aantal toppen in de patch is.

Stelling 5.2.1. *Indien het aantal vijfhoeken van een patch $p < 6$ is, en we reeds de symmetrieën van de rand van deze patch bepaald hebben, kunnen we de canonische vorm van deze patch in $O(n)$ tijd berekenen, waarbij n het aantal toppen van de patch is.*

Bewijs. Voor elke mogelijke startpositie op de rand, gaan we 1 keer een lus overlopen. In deze lus gaan we elke top van de graaf een label geven. We kunnen deze lus uitvoeren in $O(n)$ tijd. Dit komt doordat elke van de n toppen ten hoogste graad 3 heeft, zodat we $O(1)$ tijd nodig hebben per top. Indien we de vergelijking met de kleinste totnogtoe gegenereerde representant moeten maken (indien deze reeds bestaat), dan zal dit ook van de orde $O(n)$ zijn. Immers, de representatie bestaat uit evenveel lijstjes van buren als er toppen zijn, en elk van deze lijstjes is van constante lengte 3.

We maken nu gebruik van het feit dat we in 3.1.3 reeds bewezen hebben dat indien $p < 6$, het aantal symmetrieën van de rand een vaste bovengrens heeft. We hoeven dus (gegeven de randautomorfismen) slechts $O(1)$ representanten bekijken. We bekomen dus een totale tijdscomplexiteit van $O(n)$. □

Ook indien we *niet* de randautomorfismen hebben, zullen we meestal nog steeds in lineaire tijd de canonische vorm van de patch kunnen genereren. We moeten dan wel de canonische vorm iets aanpassen: in plaats van te beginnen op de rand, en dan te kijken naar de randautomorfismen om het aantal startposities te reduceren, gaan we kijken naar de binnenkant van de patch.

We kijken naar het aantal vijfhoeken in de patch. We gaan er namelijk steeds van uit dat het aantal vijfhoeken in een patch een strikte bovengrens $p < 6$ heeft. Indien we de canonische vorm aanpassen zodat we niet meer kijken naar de rand, maar naar de vijfhoeken in de patch, kunnen we het probleem oplossen. We hebben namelijk maar $O(1)$ vijfhoeken, en dus ook maar een constant aantal startposities. Dus indien we de vijfhoeken kennen, kunnen we een canonische vorm genereren in $O(n)$ tijd. Dit kan natuurlijk alleen indien er minstens 1 vijfhoek is.

Dan hoeft alleen nog te worden opgemerkt hoe we de vijfhoeken kunnen herkennen in $O(n)$ tijd. Indien we een vaste draairichting van buren fixeren rond de toppen, kunnen we bij elke top in $O(1)$ tijd kijken of hij in een vijfhoek ligt. Dit kunnen we dan voor elke top in de graaf doen, zodat dit inderdaad in lineaire tijd bepaalbaar is.

Indien er geen vijfhoek is, zal deze aanpak natuurlijk niet werken, hoewel waarschijnlijk gelijkaardige technieken zouden kunnen worden bedacht. In principe is dat natuurlijk helemaal niet nodig omdat we reeds de informatie van de rand hebben. Indien we dit natuurlijk toch zouden willen, kan altijd geprobeerd worden om het artikel van Hopcroft en Wong te implementeren

([HW74]), om zodoende ook een canonische vorm in lineaire tijd te berekenen.

Een belangrijke opmerking is dat deze methode wél zal werken indien $p = 6$: we hebben reeds gezien dat in dat geval er arbitrair grote symmetrie kan zijn. Dus ook een lineair aantal startposities die we moeten bekijken indien we deze informatie zouden gebruiken. Bij $p = 6$ weten we echter wel dat er een constant aantal vijfhoeken is waar we naar kunnen kijken.

Hoofdstuk 6

Dynamisch programmeren

6.1 Dynamisch programmeren

Dynamisch programmeren is een speciale vorm van het ontwerpen van algoritmes. Bij het gebruik van dynamisch programmeren in het ontwerp van een algoritme, wordt gebruik gemaakt van de structuur die deelproblemen vertonen om het totale probleem op te lossen. Vaak worden de oplossingen van de deelproblemen dan opgeslagen in een lijst of tabel, zodat ze eenvoudig kunnen worden opgevraagd om het grote probleem op te lossen. Zo kan dus het resultaat van een reeds opgelost deelprobleem achteraf hergebruikt worden in het algoritme. Doordat deze deelproblemen dan slechts een enkele keer volledig worden uitgewerkt, kan een aanzienlijke snelheidswinst worden bekomen indien deze deelproblemen vaak terugkeren in de loop van het algoritme.

In het algoritme om patches in te vullen, kan dit ook worden uitgebuit. Indien we op de naïeve manier het algoritme zouden implementeren, zonder van dynamisch programmeren gebruik te maken, kan de volgende situatie zich voordoen. Stel dat we tijdens het invullen van een rand een bepaalde interne rand tegenkomen die geen invullingen bevat. Aangezien we dit niet op voorhand weten, kan het zijn dat het programma veel werk moet doen om dit te achterhalen. Het is nu echter mogelijk dat we deze interne rand meerdere malen tegenkomen in de loop van de uitvoering van het programma. Dit kan gebeuren op vele manieren. Bijvoorbeeld indien een interne rand een bepaalde symmetrie vertoont, kan het zijn dat eens we deze rand hebben opgesplitst in 2 kleinere randen, beide nieuw gevormde interne randen gelijk zijn. Indien dan een bepaalde interne rand die voortkomt uit deze situatie, geen invulling heeft, zullen we dit eerst achterhalen in een van de twee deelproblemen. Achteraf zullen we echter in de tweede

rand exact dezelfde situatie tegenkomen. Indien we hier geen aandacht aan besteden, gaat veel rekentijd nutteloos verloren. Deze situatie kan zich natuurlijk voordoen op veel minder duidelijke voorbeelden, waardoor dit zeer moeilijk te anticiperen is zonder gebruik te maken van dynamisch programmeren.

Nu is dynamisch programmeren gewoon een benaming van hoe je een probleem kan oplossen: het is een soort *meta-algoritme*. Om dit meta-algoritme van dynamisch programmeren toe te passen op het invullen van randen, moeten specifieke beslissingen worden genomen. Zo beslissen we om niet alle mogelijke invullingen bij te houden van alle randen die we tegen komen. Dit zou namelijk veel te veel plaats innemen, en niet efficiënt genoeg zijn. In plaats daarvan zullen we het toepassen op een deelprobleem, namelijk beslissen of een rand een invulling kán hebben.

Hiertoe houden we in een structuur bij welke randen we reeds zijn tegengekomen, en welke daarvan géén invulling hadden. Op deze manier kunnen we tijdens het invullen van een rand gebruik maken van informatie die we reeds berekend hebben. Indien we namelijk een bepaalde rand reeds tegengekomen zijn, en toen beslist hebben dat deze geen invullingen heeft, kunnen we onmiddellijk stoppen met het invullen van deze rand, en als resultaat teruggeven dat deze rand geen invullingen heeft.

We kunnen deze informatie bijhouden aan de hand van de randcode van de rand die we zullen proberen in te vullen. Deze heeft wel als nadeel, dat ze niet uniek is voor de rand: spiegelingen en cyclische verschuivingen worden niet in rekening gebracht. We kunnen een aantal mogelijke manieren beschouwen om hiermee rekening te houden.

1. Er wordt helemaal geen rekening mee gehouden. Aangezien deze informatie slechts bedoeld is om het programma versnellen, is het niet inherent fout om de verschillende representaties van de rand hier te negeren. Het enige resultaat zal zijn dat er meer werk opnieuw zal worden gedaan, dan indien we hiermee wél rekening zouden houden. Wel zal deze methode de snelste zijn qua opstellen van de tabel: indien we een rand slechts één enkele maal tegenkomen, wordt geen enkele extra inspanning gedaan om de verschuivingen te bepalen. Indien we een rand echter vele malen tegenkomen, maar op verschoven of gespiegelde manieren, kan dit zeer nadelig zijn. We hebben deze rand dan in principe reeds gezien, maar zullen toch opnieuw proberen hem in te vullen.

2. We kunnen proberen alle verschuivingen en spiegelingen in de lijst te stoppen. Dit heeft als nadeel dat indien we de lijst opstellen, extra veel werk moeten doen. Bovendien zal de lijst met elementen zeer snel groeien, wat het geheugengebruik niet ten goede zal ko-

men, zeker indien deze specifieke rand slechts één enkele keer wordt bekomen in het hele generatieproces. Als groot voordeel heeft deze techniek natuurlijk wel dat elke keer we dezelfde rand tegenkomen, we hem zeker in de lijst terug zullen vinden, zonder enige extra inspanning te moeten doen.

3. Een tussenweg tussen bovenstaande manieren is een canonische vorm van de rand opstellen. We zagen in hoofdstuk 5 dat het voordelig kan zijn om een canonische vorm te bepalen voor de patch of een rand, om dubbel werk tegen te gaan. Indien we de canonische vorm bepalen van de rand, kunnen we aan deze canonische vorm koppelen of de rand een invulling heeft. Als we dan een rand tegenkomen, bepalen we de canonische vorm ervan, en kijken in de lijst naar de informatie die misschien reeds is opgeslagen bij deze canonische vorm. Deze canonische vorm moet dan natuurlijk liefst zo efficiënt mogelijk te bepalen zijn, omdat deze vaak zal moeten worden bepaald. Deze manier heeft als groot voordeel dat we maar één enkele keer een rand zullen opslaan, ook al bezit deze veel symmetrieën. Indien de rand maar weinig symmetrie heeft, is dit zeer voordelig. Bovendien kan een groot deel van het bepalen van de canonische vorm gebeuren tijdens het bepalen van andere informatie die we sowieso zullen nodig hebben tijdens het invullen, zodat zeer weinig extra werk zal moeten worden gedaan om dit uit te voeren.

Omwille van het feit dat het gebruik van de canonische vorm slechts zeer weinig overhead heeft, zullen we deze methode gebruiken voor het dynamisch programmeren.

6.2 Cross-process 'dynamisch' programmeren

We kunnen echter nog iets beter doen dan alleen dit eenvoudige dynamisch programmeren toe te passen. We zijn namelijk niet alleen geïnteresseerd in de tijdscomplexiteit van het zoeken van invullingen, maar vooral in het in de praktijk sneller oplossen van het probleem om randen in te vullen. We kunnen dus verbeteringen aanbrengen die de beschrijving van de complexiteit zullen bemoeilijken, maar in de praktijk het algoritme sneller zullen laten uitvoeren. Dit kan worden toegepast op de informatie die we bekomen uit het dynamisch programmeren. De informatie van de invulbaarheid kan ook apart worden opgeslagen.

We kunnen een lijst met informatie over de invulbaarheid van bepaalde randen bijvoorbeeld een enkele keer op voorhand genereren, en dan deze lijst opslaan op schijf. Elke keer het programma dan start, kunnen we deze informatie hergebruiken. Indien we dit doen, hoeft zelfs

de eerste keer dat we een rand tegenkomen in een invulprobleem, deze (meestal) niet worden gecontroleerd op invulbaarheid. Er kan direct in de databank worden gekeken naar deze informatie. Het spreekt natuurlijk voor zich dat deze preprocessing stap natuurlijk wel veel tijd kan innemen indien we op zoveel mogelijk randen voorbereid willen zijn, wat niet wenselijk is.

Een nog betere manier is echter om deze externe tabel met gegevens *tijdens* het uitvoeren van het gewone programma aan te maken. Aangezien we tijdens de normale loop van het programma sowieso zullen bepalen of een rand invulbaar is, kunnen we deze informatie gewoon na het bepalen naar dit bestand schrijven. Op deze manier wordt geen extra tijd verspild op voorhand, en kan het programma dus direct gebruikt worden. Indien we dus het programma twee keer na mekaar uitvoeren, zal tijdens de tweede uitvoering de informatie over invulbaarheid van randen die berekend werd in de eerste uitvoering, direct kunnen worden gebruikt in de tweede uitvoering. Dit delen van informatie tussen twee opeenvolgende uitvoeringen van het programma zelf, noemen we *cross-process* dynamisch programmeren.

In dit geval zullen we de volgende *codering voor in het bestand gebruiken.* Een '1'-bit zal staan voor 'niet invulbaar', terwijl een '0'-bit zal staan voor 'misschien invulbaar *of* wel invulbaar'. Met 'misschien invulbaar' wordt bedoeld dat het nog niet geweten is of deze rand een invulling bevat. We zullen dus de rand moeten invullen om te kijken of hij invulbaar is. Indien hij niet invulbaar is, kan dan deze bit worden veranderd in een '1' bit. De reden dat we geen onderscheid maken tussen 'misschien' en 'zeker' invulbaar, is omdat dit onderscheid alleen extra databits zou nodig hebben in een opslagmethode. Dit is echter helemaal niet nodig: de actie die in beide gevallen moet worden ondernomen is namelijk dezelfde. In beide gevallen zullen we de rand moeten invullen.

Men kan zich hier de vraag stellen hoe we deze informatie zo efficiënt mogelijk op schijf kunnen opslaan. Aangezien veel schijfruimte minder kost dan geheugenruimte, is het minder erg dat we niet te nauw letten op hoe plaats-efficiënt de opslag is, zolang we de data maar snel kunnen opvragen.

6.2.1 Opslagmethode

Eerst en vooral moet worden beslist hoe we de informatie achteraf zullen opvragen. Indien we de canonische vorm voor randen met lengte kleiner dan 33 of 65 gebruiken, zoals beschreven in 5.1, kunnen we dit op twee manieren aanpakken. Indien we deze canonische vorm bekijken als een binair getal van lengte 32 of 64, kan dit bijvoorbeeld een index in een bestand zijn. De

inhoud van het bestand op de positie bepaald door dit getal, kan dan bevatten of de rand met deze canonische vorm een invulling heeft, of net niet. Het probleem met deze aanpak is dat de canonische vorm beschouwd als integer, het onderscheid niet meer kan maken tussen randen met canonische vorm '01' of '0001', aangezien beide de positie '1' in het bestand zouden aanduiden. Een oplossing zou er in kunnen bestaan om per lengte een apart bestand aan te maken, en deze dan op deze manier te indexeren.

Een overzichtelijkere manier zou echter alle informatie in een enkel bestand proberen op te slaan. We kunnen dit doen door de informatie te bewaren op de manier waarop *binaire heaps* meestal worden bewaard. Dit wordt gedaan als volgt. We beschouwen het bestand nog steeds als een grote array waarvan we eenvoudig de waarde op een indexpositie kunnen bepalen. Op posities 0 en 1 staat de informatie van de canonische vormen met waarde '0' en '1' respectievelijk. Hierna volgt op geordende wijze alle informatie van de canonische vormen met lengte 2. Dan komt de informatie over canonische vormen met lengte 3, en zo gaan we maar door.

Voor randlengte l zijn in totaal 2^l mogelijke posities waar we moeten kunnen opslaan of deze canonische vorm, indien deze positie inderdaad een canonische vorm voorstelt, al dan niet een invulling heeft. Om alle randlengtes tot en met l zo op te slaan, hebben we dus een opslagcapaciteit nodig van

$$\sum_{i=1}^{l} 2^i = 2^{l+1} - 2$$

We starten daarbij de sommatie vanaf 1 omdat we de rand met lengte 0 niet meetellen.

Als we de index als bitindex beschouwen, en 8 bits per byte nemen, kunnen we bepalen hoeveel diskruimte hiervoor benodigd is. Voor randlengtes tot en met 32 hebben we dan ongeveer $(2^{33} - 2)/(8 \cdot 1024 \cdot 1024) \approx 1024$ MiB ruimte benodigd. Helaas zal dit natuurlijk exponentieel snel stijgen in de randlengte. Indien we de volledige 64 bit willen kunnen addresseren, zouden we $(2^{65} - 2)/(8 \cdot 2^{40}) \approx 2^{65}/2^{43} = 2^{22}$ TiB nodig hebben. Dit is natuurlijk niet echt doenbaar. We kunnen gelukkig een kleine verbetering aanbrengen.

Opmerking 6.2.1. *Indien het aantal vijfhoeken $p < 6$, zal de rand van de patch steeds een subsequentie '22' bevatten.*

Bewijs. We weten reeds dat $v_2 - v_3 = 6 - p > 0$. Daaruit halen we dat $v_2 > v_3$ zal moeten zijn, waardoor er zeker ten minste een stuk van de rand '22' zal zijn. □

Gevolg 6.2.2. *De canonische vorm van de rand zal steeds beginnen met de bits '00' indien $p < 6$.*

Bewijs. In de bitstring stellen we het stukje rand '22' als '00' voor. Bij het bepalen van de canonische rand zullen we zeker proberen te beginnen met deze substring. Aangezien dit kleiner zal zijn dan enige positie die *niet* met '00' begint, zal de canonische vorm zeker met '00' beginnen. □

Met andere woorden, als aan de voorwaarden $p < 6$ voldaan is, kunnen we de eerste twee bits van de canonische vorm weglaten, wat een besparing in de benodigde opslagruimte met een factor 4 zal betekenen. Indien de rand $p = 6$ is, stoppen we sowieso, zodat we daar geen rekening mee moeten houden. (Zometeen zullen we hier echter een omweg voor bespreken.) We kunnen eveneens een opmerking maken betreffende het *einde* van een canonische vorm.

Opmerking 6.2.3. *De canonische vorm van een rand met $p < 6$ en $v_3 > 0$ zal altijd eindigen op een '1' bit.*

Bewijs. We weten reeds dat deze zal beginnen met '00'. Deze vorm zal dus beginnen met '0^n1', omdat er minstens één '3' in de rand zit. Indien de laatste positie géén '1' bevatte in de canonische vorm, dan moet deze positie ingenomen worden door een '0'. We kunnen dan echter op een cyclische wijze een canonische vorm maken die vertrekt van deze laatste positie, die dan met de vorm '$0^{n+1}1$' zal beginnen. Dit is echter strikt kleiner dan de oorspronkelijke canonische vorm, wat de minimaliteit ervan tegenspreekt. □

Het geval $v_3 = 0$ met $p < 6$ zal zich enkel voordoen indien de rand precies een vijf- of zeshoek van '2's zal voorstellen. Dit is duidelijk, gezien $v_2 - 0 = 6 - p > 0$. Indien we dus apart bijhouden dat een vijfhoek en een zeshoek van '2's invulbaar zijn, en dat de rest van de mogelijke strings '2^n' niet invulbaar is, kunnen we ook deze laatste bit weghalen van de canonische voorstelling bij het opslaan.

Indien we het geval $p = 6$ nog in beschouwing zouden nemen, kunnen we dit enigszinds als speciaal beschouwen. Aangezien $v_2 = v_3$ zal ofwel voorkomen dat er minstens eenmaal een '00' als subsequentie voorkomt, of dat de rand van de vorm '$(23)^n$' is. In hoofdstuk 3.1 hebben we echter reeds gezien dat er voor elke rand van de vorm '$(23)^n$', met $n \geq 5$ een invulling bestaat. Indien we dit geval tegenkomen, kunnen we sowieso antwoorden dat er zeker een invulling bestaat.

6.2.2 Gebruik in het programma

Nu we een idee hebben hoe deze informatie op schijf staat, moeten we weten hoe we deze in het geheugen bewaren, liefst zo ruimte- en tijdsefficiënt als mogelijk. Een mogelijkheid zou zijn de tabel in te lezen, en in het geheugen voor te stellen op exact dezelfde manier als op schijf. Helaas verspilt dit nogal wat geheugenruimte, vooral bij grotere randlengtes. Alle cyclische verschuivingen en alle spiegelingen van de canonische rand (die met '00' beginnen) zullen namelijk ook in deze voorstelling zitten, terwijl deze geen nieuwe informatie toevoegen.

Voorts is het onwaarschijnlijk dat álle mogelijke randen zullen voorkomen tijdens het zoeken naar invullingen van een bepaalde rand. Er zouden dus waarschijnlijk veel te veel gegevens worden in het geheugen geladen, indien we alle mogelijke canonische vormen uit het bestand naar het geheugen zouden lezen. Een andere mogelijkheid is natuurlijk om helemaal niets te bewaren in het geheugen, maar om steeds de gegevens uit het bestand uit te lezen. Een gulden middenweg dringt zich op.

Een mooi compromis zou zijn om in het geheugen met een binaire zoekboom te werken, met als sleutelwaarde de canonische rand. Indien we een bepaalde canonische rand hebben, zoeken we deze op in de boom. Het kan zijn dat de boom deze de rand reeds bevat: dan nemen we gewoon de waarde die in de boom zit als antwoord. Het kan echter ook zijn dat de canonische rand nog niet in de boom zit: dan zoeken we het antwoord op in het bestand, voegen deze informatie aan de boom toe, en geven dit antwoord terug. Op deze manier zal hopelijk niet het hele databestand moeten worden ingelezen.

Voor de kortere randlengtes kunnen we echter een uitzondering maken. Deze kunnen we rechtstreeks in het geheugen voorstellen op de manier waarop ze op schijf staan. De kortste randlengtes zullen namelijk in absolute waarde minder 'overbodige' randsequenties voorstellen, zodat opslag efficiënter zal zijn.

Hoofdstuk 7

Genereren van mogelijk canonische randen

We willen alle patches met een bepaalde *randlengte* en een *vast aantal vijfhoeken* vinden. Dit probleem valt op te splitsen in twee deelproblemen: genereer alle mogelijke randen die aan deze grensvoorwaarden voldoen, en genereer van elke van deze randen de invulling. Hiermee zijn we echter nog niet helemaal tevreden, we willen geen isomorfe randen bekijken: dit zou ons enkel teveel werk laten doen. We zullen deze niet-isomorfe randen efficiënt proberen te genereren door tijdens het genereren van de randen reeds overbodige randen weg te snoeien. Uiteindelijk zullen we dan kijken of de gegenereerde rand gelijk is aan zijn canonische vorm.

Om een rand voor te stellen, gaan we hier voor het gemak uit van een stringrepresentatie voor een rand. Om dan alle randen van een bepaalde lengte en met een bepaald aantal vijfhoeken te genereren, moeten we weten hoe deze strings er uit gaan zien. Uit de randlengte en het aantal vijfhoeken kunnen we echter direct de waarde van v_2 en v_3 bepalen. We kunnen zelfs een kleine beperking opleggen aan combinaties van randlengte en aantal vijfhoeken, zodat we bij bepaalde combinaties van randlengte en aantal vijfhoeken direct kunnen zeggen dat er zo geen randen zijn.

Opmerking 7.0.4. *Indien de randlengte l is en het aantal vijhoeken p, dan zal $l + p$ deelbaar moeten zijn door 2. Bovendien zal dan $v_2 = (l - p)/2 + 3$ en $v_3 = (l + p)/2 - 3$*

Bewijs. We beginnen met de definitie van l in termen van v_2 en v_3 en met de formule voor het

aantal vijfhoeken:

$$\begin{cases} l = v_2 + v_3 \\ 6 - p = v_2 - v_3 \end{cases}$$

Nu kunnen we eerst de eerste vergelijking lid aan lid optellen bij de tweede, zodat we bekomen dat $6 - p + l = 2 \cdot v_2 \Leftrightarrow v_2 = (l - p)/2 + 3$. Anderzijds kunnen we de tweede vergelijking van de eerste aftrekken, met $l - 6 + p = 2 \cdot v_3 \Leftrightarrow v_3 = (l + p)/2 - 3$ als resultaat. Bovendien zal $l - p$ deelbaar moeten zijn door 2, v_2 is immers een natuurlijk getal. Hetzelfde geldt dan ook voor $l + p$. □

We maken dan een sequentie met de aldus bepaalde v_2 2's en v_3 3's, en overlopen dan alle mogelijke permutaties van deze string. Zo bekomen we alle mogelijke randvoorstellingen met de gevraagde parameters.

7.1 Het aantal randen

Het grote nadeel aan deze aanpak is echter dat hij al voor een relatief korte randlengte zeer traag zal worden. Het probleem zit hem zelfs niet noodzakelijk in het algoritme. Een groot probleem is dat er gewoon zoveel (strikt verschillende) permutaties zijn die we moeten beschouwen. Neem bijvoorbeeld alle randen van lengte $l = 50$ en met $p = 2$ vijfhoeken. We weten met voorgaande formules nu dat dan $v_2 = 48/2 + 3 = 27$ is en $v_3 = 52/2 - 3 = 23$ is.

We willen eerst en vooral weten hoeveel mogelijke randen we zullen genereren met deze v_2 en v_3. Een eenvoudige manier om dit te bekijken is als volgt. Als we de posities van de '2' vastleggen, liggen deze van '3' ook automatisch vast. Er zijn in totaal $l = 50$ mogelijke posities. We kunnen dus alle mogelijke manieren bekijken waarop we v_2 nummers kunnen trekken uit de rij getallen 1..50, zonder rekening te houden met volgorde. Elk van deze manier zal dan overeenkomen met een enkele rand: op elke positie die we trekken zetten we een '2', de rest wordt ingevuld met een '3'. Dit zijn dus in totaal $\binom{l}{v_2} = \frac{l!}{(l-v_2)!v_2!} = \frac{50!}{(50-27)!27!} = 108043253365600$ mogelijke randen die moeten worden gegenereerd. Dit is reeds een immens groot aantal. Dan hebben we ze nog niet eens de canonische vorm gegenereerd van de rand, om te kijken of deze rand wel canonisch is.

7.2 Canonische randen

Er kan natuurlijk worden opgemerkt dat veel van deze randen geen invulling zullen bevatten. Aangezien namelijk $v_2 = 27$ is, zal zelfs begonnen worden met een rand die de vorm '$2^{27}3^{23}$'.

Deze heeft geen invulling omwille van de lange opeenschakeling van '2's. Bovendien zullen we alle cyclische verschuivingen van randen ook gegenereerd hebben, wat natuurlijk veel te veel werk omvat. We kunnen beide problemen proberen te vermijden tijdens de generatie zelf. Door te snoeien van ongewenste oplossingen *tijdens* het genereren van de permutaties, zal dit een sneller resultaat opleveren. Dit is analoog naar de techniek beschreven in [Rea78], [Far78], [CR79b] en [CR79a]. Dit zal weliswaar het exponentiele karakter niet kunnen veranderen, maar zal ons toelaten toch net iets grotere randlengtes te bekijken dan zonder dit soort snoeien.

Eerst en vooral bekijken we hoe we permutaties kunnen genereren zonder het resultaat te moeten filteren achteraf. Dit kan zeer eenvoudig recursief worden gedaan. We hebben een string s van lengte l, die we willen opvullen met v_2 '2's en v_3 '3's. Het eerste symbool van een string kan ingevuld worden met een '2' indien $v_2 > 0$, en het kan worden ingevuld met een '3' indien $v_3 > 0$. Op beide manieren bekomen we een deelresultaat, dat we nog moeten aanvullen met een string van lengte $l - 1$ die dan $v_2 - 1$ '2's en v_3 '3's zal hebben, of omgekeerd. Indien we op deze manier l symbolen hebben geplaatst, of equivalent, als $v_2 = 0$ en $v_3 = 0$, dan hebben we een voorstelling van een rand gegenereerd, en kunnen we hier de gewenste dingen mee doen, zoals bijvoorbeeld deze proberen in te vullen, of controleren of we deze rand reeds hebben ingevuld, etc. Indien we nu efficiënt hele reeksen van strings willen wegsnoeien, kunnen we dit doen aan het begin van deze functie. Dit simpele algoritme is in pseudocode samengevat in Algoritme 1. Het enige waar hier moet worden op gelet, is dat dit algoritme geen rekening houdt met het geval waar de snoeifunctie nog meer informatie nodig heeft om (efficiënt) te kunnen werken. Dit is echter een triviale aanpassing in een implementatie, die niets bijbrengt aan het inzicht van hoe het algoritme zelf functioneert.

We kunnen reeds beginnen te snoeien op alle strings die een subsequentie '2^5' hebben. Alleen indien de volledige string '2^5' of '2^6' is, mag deze sequentie voorkomen in de string. Anders is het niet mogelijk om een zeshoek in te vullen in deze rand. Dit kan eenvoudig weggesnoeid worden door bij te houden hoeveel '2's we opeenvolgend hebben geplaatst. Indien we dan zien dat we 5 '2's hebben geplaatst, kunnen we deze tak van het generatieproces direct snoeien. We zullen niet eens *proberen* om alle randrepresentaties met '2^5' als deelstring te genereren, laat staan proberen in te vullen. Dit zal wel geen rekening houden met de gevallen waar we een rand hebben van de vorm '$2^4 3^n 2$', waar we pas na een cyclische verschuiving toe te passen wel een subsequentie '2^5' vinden. We zullen zometeen echter een manier zien om dit probleem ook op te lossen.

Algorithm 1: Algoritme dat gesnoeide permutaties van randen genereert en verwerkt

functie: `GenereerPermutaties`

input: Een doelstring s, samen met de huidige positie p in de string

input: Het aantal '2's en het aantal '3's dat in s moet worden geplaatst, respectievelijk v_2 en v_3

```
if RandMagWordenGesnoeid(s, p, v2, v3) then
    Return()
if v2 = 0 en v3 = 0 then
    VerwerkRand(s)
    Return()
if v2 > 0 then
    s[p] ← 2
    GenereerPermutaties(s, p + 1, v2 − 1, v3)
if v3 > 0 then
    s[p] ← 3
    GenereerPermutaties(s, p + 1, v2, v3 − 1)
```

We kunnen dus nog beter proberen te doen. Bij het verwerken van een rand zullen we zijn canonische vorm berekenen, en elke canonische vorm slechts eenmalig toelaten. We kunnen nu proberen om een aantal voorstellingen van randen die zeker *geen* canonische randvoorstelling zijn, ook reeds op voorhand weg te snoeien. We kunnen de in hoofdstuk 5.1 gedefiniëerde canonische vorm op `integer`s op een eenduidige manier terug in relatie brengen met een randvoorstelling. We zeggen dat een randvoorstelling canonisch is, als de waarden in de string precies overeenkomen met de relevante bitwaarden in de bitstring die de integer voorstelt. De truc hier zal dan zijn om zeker alle canonische randvoorstellingen over te houden, en een bepaald aantal randvoorstellingen die zeker niet canonisch zullen zijn, weg te snoeien.

We hebben in Opmerking 6.2.1 reeds gezien dat elke canonische vorm voor randen met $p < 6$ zal beginnen met de bits '00', zodoende zal onze canonische randvoorstelling ook met '22' moeten beginnen. Indien we dus randvoorstellingen wegsnoeien die niet met '22' beginnen, zal elke rand (op spiegelingen en verschuivingen na) nog steeds minstens één keer worden gegenereerd, namelijk de canonische voorstelling ervan die begint met '22'. We zullen echter een hele boel overbodige randvoorstellingen weggesnoeid hebben, omdat ze duidelijk niet canonisch zijn.

We kunnen een analoge opmerking maken betreffende het einde van een canonische randrepresentatie. In Opmerking 6.2.3 merkten we ook op dat een canonische vorm op een '1' bit zal

eindigen indien $p < 6$ en $v_3 > 0$. Dit valt ook hier handig te gebruiken. Indien $v_3 = 0$ en $v_2 = 5$ of $v_2 = 6$ kunnen we onmiddellijk de enige rand teruggeven die mogelijk zal zijn: '2^5' of '2^6', waarna we meteen stoppen. Indien $v_3 > 0$ kunnen we dit tijdens het plaatsen gaan gebruiken. Indien het aantal te plaatsen '3's nul zal zijn, kijken we naar het aantal te plaatsen '2's. Indien het aantal te plaatsen '2's ook nul is, hebben we een volledige string gegenereerd, en kunnen we deze string eventueel doorgeven aan de invulfunctie. Echter, indien we nog minstens één '2' moeten plaatsen, weten we gegarandeerd dat alle strings in deze deelboom *niet* op een '3' zullen eindigen. Indien we de enige gevallen waar dit wél mogelijk is reeds hebben uitgesloten (dus '2^5' en '2^6'), kunnen we direct hier snoeien.

Dit lost meteen ook het probleem op dat we zopas hadden om de substrings van de vorm '2^5' te detecteren. Aangezien we enkel strings zullen toelaten met een '3' op de laatste positie (behalve de reeds vermeldde gevallen), hoeven we ons ook geen zorgen meer te maken over deze randproblematiek.

We kunnen echter nog veel meer randvoorstellingen wegsnoeien die duidelijk niet canonisch zijn. Hiertoe bewijzen we eerst een eigenschap van canonische randvoorstellingen. We nemen hiervoor $\cdot < \cdot$ als de lexicale ordening op strings van *gelijke lengte* over het alfabet $\{2, 3\}$, waarbij elk karakter van de ene string vergeleken wordt met het karakter van de andere string op dezelfde plaats. Met andere woorden, we hebben een ordening $<$: $(\{2,3\}^n) \times (\{2,3\}^n) \to \mathbb{B}$, voor elke $n \in \mathbb{N}$. Met $s[n..m[$ duiden we de substring van s aan die van positie n tot positie m gaat, en waarbij strings vanaf 0 worden geïndexeerd met notatie $s[i]$. Verder duiden we met $s{+\!+}t$ de concatenatie aan van 2 strings s en t. Hiermee kunnen we nu een aantal eigenschappen en opmerkingen bewijzen.

Eigenschap 7.2.1. *Neem een canonische randvoorstelling s met lengte $l > 1$. Dan geldt voor alle natuurlijke getallen i en j die voldoen aan $0 \leq i \leq l$ en $1 \leq j \leq l - i$, dat $s[0..j[\leq s[i..(i+j)[$.*

Bewijs. We bewijzen uit het ongerijmde. Stel dus dat er minstens één i en j bestaan waarvoor $s[i..(i+j)[< s[0..j[$ zou zijn. Dan kunnen we s cyclisch bekijken vanaf het startpunt i, waarvan we het resultaat s' noemen. We zullen dan, wegens de definitie van $<$, hebben dat $s' < s$. Met s' komt dus een canonische voorstelling overeen die strikt kleiner is dan die van s. Echter, s is ondersteld een canonische randvoorstelling te zijn, en die is per definitie minimaal. We hebben dus een contradictie. □

We kunnen dit nu gebruiken in de snoeicriteria van het genereren van permutaties. We

weten dat alle echte canonische randvoorstellingen aan deze eigenschap zullen voldoen. Indien we nu proberen alle randvoorstellingen die niet voldoen aan deze eigenschap op voorhand weg te snoeien, zullen we nog steeds alle canonische randen genereren die we willen invullen, maar we zullen op deze manier een hele hoop dezelfde randen wegsnoeien.

Dit kan zeer eenvoudig als volgt gedaan worden. We gaan tijdens het genereren van de randvoorstellingen proberen een i en j te vinden waarvoor deze eigenschap *niet* geldt; we kunnen dan de hele tak die geassociëerd is met dit tegenvoorbeeld direct wegsnoeien. Tijdens het generatieproces houden we een lijst bij van posities waarvan we denken dat er een tegenvoorbeeld mee kan worden geconstrueerd.

We noemen de randvoorstelling die we op dit moment aan het genereren zijn s, en de lengte die hij op dit moment heeft l. In de meeste gevallen zal deze s nog niet opgevuld zijn tot de totale lengte. Dit is een groot voordeel. Indien $p < 6$, weten we dat een canonische randvoorstelling moet beginnen met een '2'. Alle posities j waarvoor we weten dat $s[j] = 3$, kunnen we meteen al schrappen als mogelijk tegenvoorbeeld. We zoeken immers een substring die *strikt kleiner* is dan een substring van gelijke lengte die aan het begin van de string start. Een substring beginnende met een '3' zal echter steeds strikt groter zijn. We kunnen het aantal controles zelfs nog verder beperken. Hiervoor gebruiken we een opmerking die we kunnen maken over strings en hun ordening.

Opmerking 7.2.2. *Neem s een string van lengte minstens l, en t een string van lengte minstens $l+1$. Indien $s[0..l[< t[0..l[$, dan is $2{+}{+}s[0..l[< t[0..(l+1)[$.*

Bewijs. Beschouw s en t terug als de binaire representatie van natuurlijke getallen, door middel van een functie f_n die strings van lengte n afbeeldt op hun getalwaarde: voor 2 strings x en y van lengte n geldt dan dat $x < y \Leftrightarrow f_n(x) < f_n(y)$. Dan hebben we eerst en vooral al dat $f_{l+1}(2{+}{+}s[0..l[) = f_l(s[0..l[) < f_l(t[0..l[)$.

Van de getalwaarde van bitstrings weten we dat de string naar links opschuiven equivalent is met de getalwaarde vermenigvuldigen met 2, en dat indien de laatste bit '0' is, deze veranderen naar '1' hetzelfde zal zijn als 1 bij de getalwaarde optellen. Bovendien merken we op dat ofwel $t = t[0..l[{+}{+}2$ ofwel $t = t[0..l[{+}{+}3$.

Indien nu $t[l] = 2$, bekomen we dus verder dat $f_l(t[0..l[) \leq 2f_l(t[0..l[) = f_{l+1}(t[0..l[{+}{+}2)$. Indien anderzijds $t[l] = 3$, dan bekomen we dus verder dat $f_l(t[0..l[) < 2f_{l+1}(t[0..l[) + 1 = f_{l+1}(t[0..l[{+}{+}3)$. Samen leidt dit dus tot $f_l(t[0..l[) \leq f_{l+1}(t[0..(l+1)[)$.

Dit alles kunnen we dan samenvoegen tot de gevraagde ongelijkheid $f_{l+1}(2{+}{+}s[0..l[) <$

$f_{l+1}(t[0..(l+1)[) \Leftrightarrow 2{+}{+}s[0..l[< t[0..(l+1)[.$ □

Gevolg 7.2.3. *Neem s een string van lengte l, en t een string van lengte $l+n$. Indien $s[0..l[< t[0..l[$, dan is $2^n{+}{+}s[0..l[< t[0..(l+n)[$.*

Bewijs. Het resultaat volgt door n keer opmerking 7.2.2 toe te passen. □

Dit kunnen we nu gebruiken om het aantal te controleren posities sterk te beperken. We merken hiertoe het volgende op.

Opmerking 7.2.4. *Indien we een (tijdelijke) randrepresentatie s met lengte l en die begint met een '2' gecontroleerd hebben vanaf positie i met $s[i] = 2$, dan zal het nooit zijn dat voor een j zo gekozen dat $i < j < l$ en waarvoor geldt dat $(\forall k : i < k \leq j)(s[k] = 2)$, geldt dat $s[k..l[< s[0..(l-k)[$.*

Bewijs. Neem een j met $l > j > i$ zodat $(\forall k : i < k \leq j)(s[k] = 2)$. We weten dat we reeds gecontroleerd hebben vanaf i, zodat we weten dat $s[0..(l-i)[\leq s[i..l[$. We nemen nu zo een k als startpunt om te controleren, zodat we een deelstring $s[k..l[$ krijgen. We nemen nu de negatie van wat we willen bewijzen aan. Dat wil zeggen dat het mogelijk moet zijn om te krijgen dat $s[k..l[< s[0..(l-k)[$. Echter, dan zou $s[i..l[< s[0..(l-i)[$, want $s[i..l[= 2^{k-i}{+}{+}s[k..l[$. We hebben echter ondersteld dat dit niet waar is, een contradictie dus. □

Hiermee kunnen we dus zeer efficiënt startposities kiezen. Enkel indien we een '2' na een '3' plaatsen gaan we een markering maken, dat deze positie een mogelijke startpositie vormt om de niet-canoniciteit aan te tonen van een bepaalde deelboom. Dit kan zeer efficiënt worden geïmplementeerd door het volgende op te merken.

Opmerking 7.2.5. *Indien de randen gegenereerd worden in lexicografische volgorde, dan zal na het verwerpen van een startpositie, deze niet meer beschouwd moeten worden als mogelijke startpositie tot het proces backtrackt tot vóór deze positie: voor elke mogelijke vanaf dit punt gegenereerde string s' die niet backtrackte tot voor positie i zal voor elke $j > i$ gelden dat $s'[i..j[> s'[0..(j-i)[$.*

Bewijs. Stel dat we een mogelijke startpositie i hebben, en we zitten op dit moment op positie $j > i$ in een string s die we aan het genereren zijn. De enige reden dat i nog steeds een mogelijke startpositie is op dit punt, is dat $s[i..(j-1)[= s[0..(j-i-1)[$. Als we op dit punt in het generatieproces i als startpositie verwerpen, is dit omdat $s[i..j[> s[0..(j-i)[$.

We kunnen nu de lexicografische generatievolgorde gebruiken. We bekijken alle strings die verder gegenereerd worden in lexicografische volgorde vanaf i (dus zonder voor i terug te keren). Deze strings zullen altijd strikt groter zijn dan degene die we nu bekijken. Deze strings zijn dan deelstring vanaf positie i van een string s'. Voor een $j > i$ zal dan $s'[i..j[> s[i, j[> s[0..(j-i)[$. We willen echter weten of $s'[i..j[> s'[0..(j-i)[$ ook geldt.

Indien $i \geq j - i$ is er geen overlap tussen de twee substrings die we vergelijken. We hebben dus direct dat $s'[0..(j-i)[= s[0..(j-i)[$. Omdat de randen nu in een lexicografische ordening worden gegenereerd, zal $s'[i..j[> s[i..j[> s[0..(j-i)[= s'[0..(j-i)[$.

Indien $i < j - i$ is er een bepaalde overlap tussen de twee substrings die we vergelijken. De stukken string die overlappen zijn $s'[i..(j-i)[$ en $s[i..(j-i)[$. Dit stuk kan dus veranderd worden tijdens het backtrack proces. Er zal echter altijd een stuk niet overlappen: $s[0..(j-2i)[= s'[0..(j-2i)[$.

We hebben dus $s'[i..(j-i)[\geq s[i..(j-i)[$. Indien $s'[i..(j-i)[= s[i..(j-i)[$ hebben we direct ook dat $s'[0..(j-i)[= s[0..(j-i)[$, omdat we niet backtracken voor i. Dit is dan volledig analoog als het voorgaande geval.

Dan blijft nog over dat $i < j - i$ en $s'[i..(j-i)[> s[i..(j-i)[$. Echter,

$$\begin{aligned} s'[0..(j-2i)[&= s[0..(j-2i)[\\ &= s[i..(j-i)[\\ &< s'[i..(j-i)[\end{aligned}$$

Dus het gestelde volgt. □

Indien we dus in een bepaalde deelboom van het generatieproces vinden dat de substring startende van een mogelijke startpositie zeker groter zal zijn dan de volledige string, moeten we deze startpositie niet verder bekijken zolang het generatieproces niet backtrackt tot *voor* deze mogelijke startpositie. We kunnen nog een andere belangrijke opmerking maken. Elke keer we de lijst mogelijke startposities bekijken om te onderzoeken of we een mogelijke startpositie kunnen schrappen van de lijst, hoeven we per mogelijke startpositie slechts *één* karakter te vergelijken met de reeds gegenereerde string.

Opmerking 7.2.6. *Indien we een canonische randvoorstelling s aan het genereren zijn, en we hebben een mogelijke startpositie i, dan hoeven we bij het toevoegen van een enkel karakter c*

aan s op plaats j slechts 1 karakter vergelijken om te beslissen of i nog steeds een mogelijke startpositie is.

Bewijs. Voor het plaatsen van c weten we doordat i een mogelijke startpositie is, dat $s[\,i..j[\,= s[0..(j-i)[$ is (als $s[i..j[< s[0..(j-i)[$ was, dan hadden we reeds eerder beslist dat s zeker niet canonisch is). Indien we nu de toewijzing $s[j] := c$ doen, dan zijn er slechts twee keuzes voor de mogelijke startpositie i. Ofwel is $c \leq s[j-i]$, ofwel is $c > s[j-i]$. In beide gevallen zal deze ene vergelijking bepalen of $s[i..(j+1)[\leq s[0..(j-i+1)[$ zal gelden aangezien de eerste $j-i$ karakters reeds gelijk zijn. □

Er kan een belangrijke opmerking gemaakt worden betreffende de tijdscomplexiteit van deze test.

Opmerking 7.2.7. *Voor een enkele rand bekeken is de complexiteit van dit vergelijken van karakters even snel als het in één keer achteraf te bekijken. Voor alle randen tegelijk worden echter vele verschillende randen door eenzelfde test reeds bekeken.*

Bewijs. Indien we deze test voor een enkele rand met lengte n bekijken, is dit eenvoudig in te zien. In plaats van op het einde een vergelijking met complexiteit $O(n)$, doen we nu n vergelijkingen met elk $O(1)$ complexiteit.

Door deze test echter uit te smeren over het hele generatieproces, zal dit in totaal echter veel efficiënter zijn dan voor elke string de hele vergelijking uit te voeren. Vooral de tests in het begin van de rand, zullen hier veer voordeel uit halen: alle randen die het beginstuk gemeen hebben, zullen reeds een deel van de tests *gedeeld* hebben. We kunnen dit bekijken alsof de tests in het begin van de string '*parallel*' lopen voor een hele hoop strings. □

Dit is ook de aanpak die in het 'minibaum' programma wordt gebruikt ([Bri92]).

We kunnen ook in de *tegenovergestelde* richting een lijst met mogelijke startposities aanduiden. Bij het plaatsen van elk karakter, kunnen we kijken of de string vanaf het net toegevoegde karakter al-dan-niet groter is dan de string in de normale richting. Ook hier kunnen we dan een lijst van *mogelijke omgekeerde startposities* van bijhouden.

Jammer genoeg moeten we ook opmerken dat zolang we de hele string niet hebben gegenereerd, we niet kunnen bepalen of het ook *echt* een canonische vorm zal zijn. Dit komt doordat we zolang we de eindtekens van de rand niet hebben gegenereerd, we de rand niet cyclisch kunnen bekijken op canoniciteit. We kunnen dit in principe wél doen in de laatste stap. Indien we tijdens

het genereren de mogelijke startposities bijgehouden hebben (in beide richtingen), kunnen we deze informatie gebruiken om te controleren of deze randvoorstelling inderdaad canonisch is. Dit is ook een vorm van dynamisch programmeren: tijdens de uitvoer van het algoritme verzamelen we informatie informatie die later nuttig zal blijken te zijn om het algoritme te versnellen.

We kunnen nu het hele snoeiproces samenvatten in Algoritme 2.

Algorithm 2: Snoeicriteria bij het genereren van permutaties

functie: `wordtRandGesnoeid`

input: De string die we aan het genereren zijn: s van lengte l

input: Een lijst L met mogelijke startposities waar we elementen uit kunnen verwijderen

if *meer dan 4 opeenvolgende '2's geplaatst* of *meer dan 4 opeenvolgende '3's geplaatst* **then**
  `Return(`*SnoeiRand*`)`

for *elke mogelijke startpositie* $p \in L$ **do**
  if $s[l-1] < s[l-1-p]$ **then**
    `Return(`*SnoeiRand*`)`
  if $s[l-1] > s[l-1-p]$ **then**
    Verwijder p uit de lijst L met mogelijke startposities

if *Aantal '3' te plaatsen = 0* **then**
  if *Aantal '2' te plaatsen = 0* **then**
    `AccepteerRand()`
  `Return(`*SnoeiRand*`)`
`Return(`*SnoeiRandNiet*`)`

7.3 Snelheid

Het is natuurlijk ook zeer belangrijk om te weten in hoeverre de bepaalde criteria ook daadwerkelijk een versnelling toebrengen aan het programma. We moeten hier wel goed opletten wat we gaan meten. Het is namelijk zo dat een deel van de snoeicriteria een zekere kost hebben om te berekenen, die het genereren trager zal kunnen maken. Het eindresultaat zou echter kunnen zijn dat er veel minder randen *in totaal* worden gegenereerd, waardoor we bij het verwerken van al de randen veel minder werk moeten doen.

We zullen de volgende meetopstelling gebruiken. Eerst worden *alle* randen gegenereerd met het correcte aantal '2's en '3's. Van elk van deze randen, gaan we dan de canonische vorm bepalen, en kijken of hij wel voldoet aan de voorwaarde dat hij geen substring '2^5' of '3^5' mag

bevatten. Als de randvoorstelling dan gelijk is aan de canonische vorm, en voldaan is aan de randvoorwaarden, registreren we dat we een nieuwe canonische rand hebben gegenereerd. We meten ook hoeveel randen er *in totaal* gegenereerd werden door deze methode (zonder te filteren op canoniciteit).

Dan kunnen we snoeicriteria toevoegen. We meten het effect van deze, door zoals bij de naïeve generatiemanier ook alle gegenereerde randen achteraf door een canoniciteitsfilter te sturen. Hier kunnen we dan gebruik maken van de informatie die we reeds bijhielden tijdens het genereren: we hoeven alleen de mogelijke startposities en de mogelijke omgekeerde startposies te bekijken bij de canoniciteitsfilter. Niet alleen zullen we zien hoe snel dit is, maar we gaan ook kijken of deze filter veel randen weg kon snoeien. Hiertoe vergelijken we het aantal randen dat we door de canoniciteitsfilter moesten sturen. Dit is dan weer te geven door een eenvoudig percentage ten opzichte van het 'trage' algoritme.

De snelheidsmetingen zelf werden gedaan op een AMD64 3500+ processor met 2G ram geheugen, en onder een Linux 2.6 kernel. Het generatieprogramma werd gecompileerd met `g++ (GCC) 4.0.3 20060104 (prerelease) (Debian 4.0.2-6)` en de `-O3` optimalizatie.

We zullen dus om een goed overzicht te krijgen van de snoeicriteria, op twee factoren moeten letten. De factoren die we gaan bekijken zijn snelheid van generatie en aantal gegenereerde randen. Met de snelheid van generatie wordt het aantal seconden bedoeld dat benodigd is om alle randen met gegeven parameters te genereren. Voor snelheid van generatie is het ook nuttig om de afgeleide waarde te zien die aangeeft hoeveel randen we per seconde genereerden.

In deze sectie zullen we een samenvatting en bespreking geven van de metingen die we hebben uitgevoerd. Deze metingen zijn samengevat te vinden in tabel 7.1. We dienen uiteraard weer op te merken dat, aangezien $l + p$ even moet zijn, niet alle combinaties van l en p enige rand zullen kunnen teruggeven. Deze zijn dan ook uit de tabellen weggelaten.

We beginnen met te kijken naar de implementatie van de gewone generator. In de meest linkse kolommen met metingen in de tabel zijn de metingen samengevat van dit proces. Om een representatief overzicht te geven van de looptijd van het algoritme, variëren de parameters in redelijke intervallen: $p \in \{0, .., 5\}$ en $l \in \{20, .., 28\}$. We merken direct op dat zoals reeds eerder berekend, de randlengten verhogen met 2 er voor zorgt dat de generatietijd ongeveer verviervoudigt.

In de tweede reeks kolommen, is het eenvoudige algoritme toegevoegd met een snoeimethode waar we alle randen van met substring '2^5' wegsnoeien, en waar de rand moet beginnen met '22'.

We kunnen dan direct vergelijken hoe dit algoritme presteert ten opzichte van het voorgaande. We hebben ongeveer slechts de helft van de uitvoeringstijd nodig van het vorige algoritme. Tegelijkertijd snoeien we reeds zo'n 65% van het aantal gegenereerde randen weg voor deze parameters! Dit is duidelijk reeds een zeer goed snoeicriterium.

We kijken nu naar de derde reeks metingen in de tabel. Hier snoeien we niet alleen op de randen met '2^5' als deelstring, maar we gaan ook eisen dat de rand eindigt op een '3'. We snoeien direct ook dat '2^5' geen deelstring meer mag zijn. We merken meteen op dat het aantal randen dat gesnoeid wordt, ten opzichte van het standaard algoritme, reeds oploopt tot ongeveer 92%. Tegelijkertijd is de uitvoertijd van het algoritme reeds gereduceerd van 13s naar 2s in het geval van $p = 5$ en $l = 27$. Dit is duidelijk zeer positief.

Het zwaarste snoeicriterium werd toegevoegd in de metingen van de laatste kolommen. Wat we meteen kunnen opmerken, is het grote aantal gesnoeide randen. Zo'n 96 á 98% van de oorspronkelijke randen werd weggesnoeid door niet alleen '2^5' weg te snoeien en te kijken of de rand eindigt op een '3', maar ook door steeds te kijken of een rand nog een canonische rand kan zijn. Dit is echter een redelijk zwaar criterium: indien we zouden kijken naar de tijd die het algoritme spendeert aan het genereren van de randen *zonder* het snoeien op canoniciteit, dan zien we dat dit slechts ongeveer de helft sneller zou zijn. Echter, aangezien we een heel groot aantal randen direct wegsnoeien, is het eindresultaat overweldigend positief. Bij het eerdervermelde voorbeeld van $p = 5$ en $l = 27$ wordt de uitvoersnelheid zelfs gereduceerd van 13s naar 0.2s. Dit is duidelijk een zeer goede keuze om te gebruiken als generator van canonische randen.

		Eenvoudig algoritme			Snoei op '2's			Snoei op '2's en '3's			Snoei op alles		
p	l	**#randen**	**tijd (s)**	**randen/s**	**tijd (s)**	**randen/s**	**% gesnoeid**	**tijd (s)**	**randen/s**	**% gesnoeid**	**tijd (s)**	**randen/s**	**% gesnoeid**
0	20	581	0.05	11620	0.04	14525	58.9474	0.01	58100	94.275	0	-	98.4043
0	22	2199	0.23	9560.87	0.15	14660	60.6061	0.03	73300	94.3866	0.01	219900	98.5759
0	24	7945	0.95	8363.16	0.61	13024.6	61.9565	0.13	61115.4	94.559	0.01	794500	98.7383
0	26	29229	3.96	7381.06	2.53	11553	63.0769	0.5	58458	94.7693	0.05	584580	98.8797
0	28	105846	16.36	6469.8	10.29	10286.3	64.0212	2	52923	95.0015	0.18	588033	99.0059
1	21	1875	0.14	13392.9	0.09	20833.3	62.8571	0.02	93750	92.9682	0	-	98.0952
1	23	6607	0.57	11591.2	0.36	18352.8	64.0316	0.09	73411.1	93.3106	0.01	660700	98.3356
1	25	23298	2.33	9999.14	1.44	16179.2	65	0.37	62967.6	93.6658	0.03	776600	98.551
1	27	82274	9.56	8606.07	5.83	14112.2	65.812	1.41	58350.4	94.0224	0.13	632877	98.7313
2	20	1545	0.08	19312.5	0.05	30900	65.2632	0.01	154500	91.5662	0.01	154500	97.5073
2	22	5139	0.33	15572.7	0.19	27047.4	66.2338	0.07	73414.3	92.092	0.01	513900	97.8783
2	24	17728	1.34	13229.9	0.8	22160	67.029	0.25	70912	92.5994	0.03	590933	98.1746
2	26	60589	5.46	11096.9	3.2	18934.1	67.6923	0.95	63777.9	93.0837	0.09	673211	98.4246
2	28	209762	22.25	9427.51	12.88	16285.9	68.254	3.6	58267.2	93.5429	0.31	676652	98.633
3	21	3800	0.19	20000	0.11	34545.5	68.5714	0.04	95000	91.0271	0.01	380000	97.3572
3	23	12623	0.76	16609.2	0.42	30054.8	69.17	0.16	78893.8	91.6719	0.02	631150	97.7677
3	25	42391	3.03	13990.4	1.71	24790.1	69.6667	0.59	71849.2	92.2699	0.06	706517	98.0918
3	27	143422	12.26	11698.4	6.84	20968.1	70.0855	2.23	64314.8	92.8246	0.21	682962	98.3624
4	20	2598	0.1	25980	0.05	51960	71.0526	0.03	86600	90.256	0	-	96.8117
4	22	8542	0.41	20834.1	0.22	38827.3	71.4286	0.09	94911.1	90.9966	0.02	427100	97.3291
4	24	27879	1.63	17103.7	0.88	31680.7	71.7391	0.35	79654.3	91.6738	0.04	696975	97.748
4	26	93113	6.57	14172.5	3.49	26679.9	72	1.31	71078.6	92.2953	0.12	775942	98.083
4	28	311590	26.43	11789.3	13.98	22288.3	72.2222	4.87	63981.5	92.8667	0.42	741881	98.3586
5	21	5339	0.22	24268.2	0.11	48536.4	73.8095	0.05	106780	90.6851	0.01	533900	96.9026
5	23	17292	0.86	20107	0.43	40214	73.913	0.19	91010.5	91.3865	0.02	864600	97.4035
5	25	56748	3.41	16641.6	1.73	32802.3	74	0.71	79926.8	92.0297	0.08	709350	97.8114
5	27	188232	13.7	13739.6	6.89	27319.6	74.0741	2.65	71030.9	92.6209	0.24	784300	98.1378

Figuur 7.1: Metingen zonder snoeicriteria en met $p \in \{0, .., 5\}$ en $l \in \{20, .., 28\}$. De *tijd (s)* kolommen geven aan hoe lang het gegeven algoritme er over deed om de canonische randen te genereren met de gegeven parameters. De **# randen** kolom geeft het aantal canonische randen met de gegeven parameters. De *randen/s* kolom tonen hoeveel canonische randen het algoritme kon genereren per seconde. De *% gesnoeid* kolom bij elk niet-'traag' algoritme geeft aan hoeveel randen werden weggesnoeid alvorens op canoniciteit te filteren ten opzichte van het 'trage' algoritme.

Hoofdstuk 8

Het algoritme om invullingen van een rand te genereren

8.1 Datastructuren

Om het algoritme te implementeren hebben we een aantal dingen nodig. Eerst en vooral moeten we weten hoe we intern de data gaan voorstellen. Aangezien we tijdens het invullen zelf nog niet van een patch kunnen spreken, noemen we dit een *tijdelijke patch*. Intern zullen we deze tijdelijke patch voorstellen met een vector van buren. Een vector is een aaneensluitend stuk geheugen waar we snelle random-acces hebben. Op de i-de positie bewaren we dan de 3 mogelijke buren van de top met label i. De buren zijn dan simpelweg de labels van de buren. Merk op dat een tijdelijke patch verschillend is van een prepatch; een prepatch heeft ten hoogste een enkel niet ingevuld intern vlak, terwijl een tijdelijke patch er meerdere kan hebben.

Aangezien we met een tijdelijke patch werken, moeten we hier twee kleine uitzonderingen maken. Een top kan namelijk 2 soorten 'buren' hebben, die geen label als zodanig hebben. Dit zijn eerst en vooral de randtoppen die graad 2 hebben. Hier kunnen we zeggen dat de derde buur een buur aan de buitenkant van de patch is. Het andere geval zijn de toppen die 3 buren hebben, maar waar we in de loop van het algoritme nog geen 3 buren aan gekoppeld hebben. We zullen deze 2 gevallen elk een apart label geven. Dit label kunnen we dan als speciaal geval opnemen in de code. Een top die uiteindelijk 3 buren moet hebben, maar die bijvoorbeeld op een bepaald moment in de uitvoering van het algoritme nog maar 1 buur heeft, zal in de positie van de 2 ontbrekende buren het label *InBoog* hebben. De 'in' verwijst ernaar dat deze bogen naar de binnenkant van de patch wijzen. Alle randbogen met graad 2 zullen op gelijkaardige

manier een boog gelabeld *UitBoog* hebben, waar 'uit' verwijst naar het feit dat we dit kunnen zien als een boog die naar de buitenkant van de patch wijst.

Bij het starten van het algoritme, zal de tijdelijke patch dus uit een lijst toppen bestaan, die elk als buren in deze lijst de 2 labels heeft van toppen die in de rand zitten. Het andere label zal dan *InBoog* zijn voor toppen die in de randcode als '3' voorgesteld zijn, en zal *UitBoog* zijn voor de toppen die in de randcode als '2' zijn voorgesteld.

Om de vlakken af te bakenen gaan we het face tracing algoritme gebruiken. In het geval van een patch kan het geen kwaad dat de rand enkele toppen bevat met graad 2, hoewel dit in het algemene algoritme niet wordt toegestaan. We moeten hier echter speciale voorzorgen nemen, omdat tijdens de constructie van een tijdelijke patch, de graaf niet 2-samenhangend is. Zolang ook hier rekening mee wordt gehouden, is er gelukkig geen probleem.

Om de draairichting in een vlak eenvoudig te kunnen volgen, leggen we deze dus vast in de manier waarop we de buur-lijst van een top opslaan. Indien we een top bekijken, fixeren we willekeurig een van zijn buren. Dit is de top die eerst in de lijst gaat. Dan draaien we in een op voorhand gekozen richting rond de buren. In deze volgorde worden de andere buren dan in de lijst gezet.

Dan kunnen we eenvoudig het face tracing algoritme toepassen. Indien we nu een boog kiezen van de top v_1 naar v_2, dan kunnen we in v_2 bepalen op welke positie van de burenlijst v_1 zit. Dankzij de constructie van de burenlijst kunnen we nu de top bepalen die volgt op v_1 indien we in een vast gekozen richting de buren rond v_2 zouden bekijken. Indien we deze top v_3 noemen, hebben we nu een geordende boog van v_2 naar v_3 waar we weer dezelfde stappen kunnen toepassen.

Het algoritme zal een *lijst* van tijdelijke patches teruggeven. Het is belangrijk om op te merken, dat in deze implementatie dit volledige kopieën van de tijdelijke patches zullen zijn.

8.2 Functies benodigd voor het algoritme

Om het algoritme te beschrijven, hebben we een aantal functies nodig die we als bouwsteen kunnen gebruiken. We geven eerst een opsomming van deze functies, alvorens het algoritme zelf te beschrijven.

- `ZoekStartPos((starttop, startrichting))`: Om zo snel mogelijk de zoekruimte te beperken, gaan we proberen een heuristiek te kiezen die een keuze voor startpunt neemt die

goed lijkt met dit doel voor ogen. We nemen de startpositie zo dat deze start aan de langst mogelijke sequentie van opeenvolgende '2's in de rand. Dit is omdat we op deze manier hopelijk snel een vijfhoek of een zeshoek kunnen plaatsen, zonder al te veel extra toppen toe te moeten voegen.

Deze functie zal als startpositie het begin kiezen van de opeenvolging van '2' waarmee de canonische vorm van de rand zal beginnen. We starten op deze sequentie, maar op de positie die zo ligt dat we in de vast gekozen draairichting kunnen meedraaien.

- `ControleerInvulbaarheid(prepatch, (starttop, startrichting))`: Deze functie zal in de eerste plaats controleren of het aantal vijfhoeken $p \geq 0$. Dit doet het aan de hand van de $v_2 - v_3 = 6 - p$ formule. Het is ook mogelijk om deze functie uit te breiden met de dynamisch programmeren functionaliteit zoals beschreven in hoofdstuk 6.2.

- `VoegBoogToe(prepatch, vantop, naartop)`: Deze functie zal de gegeven tijdelijke patch aanpassen zodat er een boog zal gaan tussen de twee toppen die worden meegegeven als argument.

- `VoegTopToe(prepatch, (top, richting)`: Hier wordt een nieuwe top aan de gegeven tijdelijke patch toegevoegd, die adjacent is aan een opgegeven top. De functie zal als resultaat het label van de nieuwe top teruggeven, zodat deze later kan worden verwijderd.

- `VerwijderTop(prepatch, top)`: Zal de gegeven top verwijderen van de gegeven tijdelijke patch.

- `KopiëerPatch(prepatch)`: Maakt een complete kopie van de hele tijdelijke patch.

- `VeelhoekAanRand(prepatch, (top, richting))`: Voegt een vijf-of zeshoek toe aan de rand, vertrekkende van de gegeven positie op de prepatch

8.3 Het algoritme zelf

We beschrijven nu hoe het algoritme zijn werk zal doen. Een algemeen overzicht in pseudocode van het algoritme is te zien in algoritme 3.

We willen gegeven een tijdelijke patch, de invullingen geven van een bepaalde (interne) rand van deze tijdelijke patch. Er worden geen aannames gemaakt over hoe deze tijdelijke patch er uitziet. Het enige wat we krijgen is deze patch, en een boog op de (interne) rand in deze patch

die moet worden ingevuld. Het is dus goed mogelijk dat er meerdere plaatsen zijn waar er iets moet worden ingevuld, daar wordt geen rekening mee gehouden. Het resultaat wat we willen bekomen met het algoritme, is dat we een lijst terugkrijgen met alle mogelijke invullingen van deze rand in de tijdelijke patch.

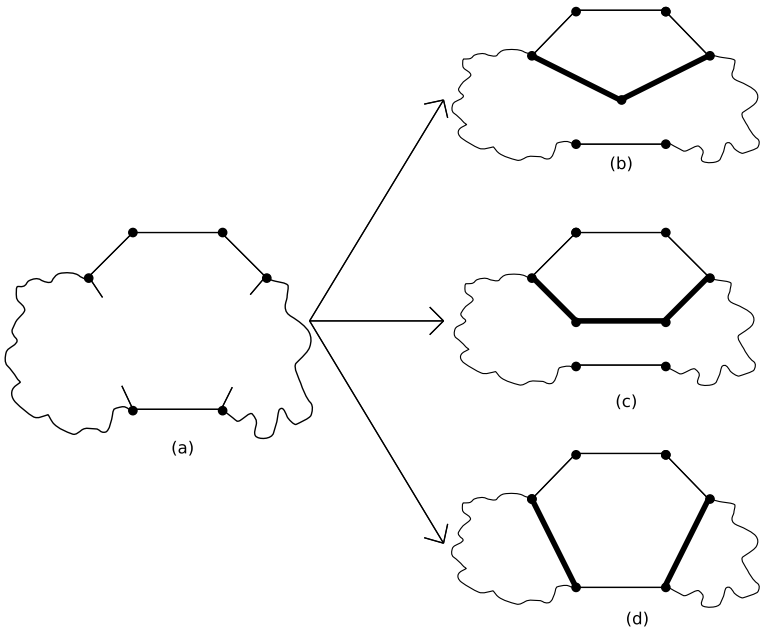
We weten dat aan elke randboog van de patch een vijf- of zeshoek zal grenzen. We willen nu op alle mogelijke manieren proberen de tijdelijke patch aan te passen, zó dat deze boog aan een vijf- of zeshoek in de binnenkant van de rand zal grenzen. Eens we deze veelhoek hebben toegevoegd, zullen mogelijks meerdere nieuwe interne randen ontstaan, in de plaats van de oorspronkelijke rand. We kunnen dan het algoritme recursief op elk van deze randen toepassen.

Van een gegeven startboog op een (interne) rand van een tijdelijke patch, kunnen we zoals reeds gezegd in een vast gekozen manier alle bogen en toppen overlopen. Om te beginnen, kijken we naar de informatie die deze rand ons direct kan vertellen over de invulbaarheid ervan. Dit wordt gedaan met de `ControleerInvulbaarheid` functie. Door bijvoorbeeld de v_2 en v_3 te bepalen van deze rand, kunnen we reeds op voorhand kijken of $p < 0$, waardoor we onmiddellijk kunnen stoppen met een invulling van deze specifieke rand te zoeken. Een andere mogelijkheid hier is het zoeken naar een opeenvolgende sequentie van '2^n' met $n \geq 7$. In dat geval kan immers nooit een volledige invulling bestaan met vijf- en zeshoeken. Er past namelijk geen zeshoek in dat stuk rand. Een andere mogelijkheid is natuurlijk het reeds vermelde dynamische programmeren.

Indien de rand reeds een echte vijf- of zeshoek voorstelt, kunnen we hier in principe reeds stoppen. Dit wil zeggen, de rand is van de vorm '2^5' of '2^6'. Er zullen geen andere invullingen zijn van deze rand, aangezien er geen enkele '3' top is waar we iets aan kunnen veranderen. We geven dan ook als lijst met mogelijke invullingen deze rand in de tijdelijke patch terug.

Indien we niet meteen kunnen beslissen dat deze rand niet noodzakelijk geen invullingen bevat, kan verdergegaan worden. We overlopen deze rand met `ZoekStartPos`. Hierdoor bepalen we een startpositie waarachter (in de vast gekozen richting) zoveel mogelijk opeenvolgende randbogen zullen zijn die naar de buitenkant van deze rand wijzen. Vanaf deze startpositie kunnen we dan beginnen met het aanpassen van de patch zodat er een vijf- of zeshoek aan grenst aan de binnenzijde van de rand. We vertrekken vanop de positie *voor* deze top, zodat onze eerste eindtop deze 'beginpositie' zal zijn.

Zoals reeds opgemerkt, kunnen we deze startpositie zo kiezen dat ze samen kan worden bepaald met de canonische vorm van de rand. Aangezien we deze canonische vorm reeds berekenen



Figuur 8.1: De mogelijke keuzes die we in het algoritme kunnen maken

indien we dynamisch willen programmeren, zullen we deze startpositie reeds bepaald hebben. Het is echter conceptueel duidelijker indien dit wordt opgesplitst in twee aparte functies in de beschrijving.

De rand wordt nu overlopen in de vast gekozen richting. Zolang de volgende ('gerichte') boog een gewone boog is, gaan we verder. Een gewone boog wil hier gewoon zeggen dat het eindpunt ervan reeds bekend is en in de patch ligt. Eens de volgende boog een 'inboog' is, moeten we de patch veranderen. Indien we een rand willen ingevuld hebben mogen we immers geen enkele inboog meer in de patch hebben. Een inboog bij een top van de patch zou betekenen dat de interne toppen niet allemaal graad 3 hebben, of dat er randtoppen zijn die een '3' top zijn maar geen 3 buren hebben in de patch.

Hebben we reeds 5 bogen overlopen, dan kunnen we nog slechts een zeshoek maken indien we willen dat deze patch een invulling heeft. Hiertoe voegen we een boog toe in de patch tussen deze top met een 'InBoog', en de starttop. Nu bekomen we een nieuwe interne rand, namelijk de rand die grenst aan de net vervolmaakte boog van de zeshoek. Indien we nu alle invullingen van de oorspronkelijke patch willen, gaan we nu recursief het algoritme oproepen op deze interne rand en de huidige patch. Het resultaat dat de recursieve oproep zal geven, zal dan het resultaat zijn van de huidige oproep.

Indien er op het moment dat er minder dan 5 bogen overlopen zijn een 'InBoog' komt, kunnen

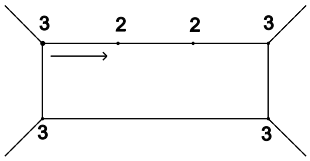
we kiezen. Stel even dat we reeds 4 bogen zijn tegengekomen. De eerste keuze die we kunnen maken, is het vervolledigen tot een vijfhoek. Dan kunnen we een recursieve oproep doen net zoals in de vorige paragraaf. Echter, in plaats van de lijst die we als resultaat van deze oproep terug krijgen, terug te geven, houden we deze bij. Want we kunnen ook een top toevoegen aan de patch. Hiertoe verwijderen we eerst de reeds toegevoegde rand. We voegen de top toe aan de patch, en verbinden deze met de huidige top die we bekijken. Het aantal InBogen van deze top zal dus met één verminderen, terwijl de nieuw toegevoegde top 2 inbogen heeft. Op deze manier kunnen we een zeshoek maken door deze nieuwe top dan opnieuw te verbinden met de starttop. Het nog niet ingevulde deel van de tijdelijke patch proberen we dan opnieuw verder in te vullen door het algoritme recursief op te roepen, en de resultaten van die oproep aan de huidige lijst resultaten toe te voegen. Op deze manier zal de recursieve oproep dit stuk rand kunnen vervolledigen. Hoewel dit strikt genomen geen tijdelijke patch is, werkt deze aanpak wel.

Er moet worden opgemerkt dat dit alles zonder problemen kan, indien de resultaten in de resultatenlijst steeds complete kopieën zijn, en niet simpelweg referenties. Indien we namelijk slechts referenties naar ingevulde tijdelijke patches zouden opslaan, zouden we deze aanpassen door de boog terug te verwijderen.

Hetzelfde kan gedaan worden in het geval dat we slechts 3 bogen zijn tegen gekomen. Er kunnen dan eerst één en dan twee toppen worden toegevoegd in totaal, om respectievelijk een vijfhoek en een zeshoek te vervolledigen op de wijze zoals voorheen.

Een laatste mogelijkheid is het toevoegen van een boog tussen de huidige top, en een andere top die reeds in de rand zit. We letten hierbij op dat we niet simpelweg een vijfhoek maken zoals we reeds maakten in de eerste stap. Zo worden twee nieuwe interne randen aangemaakt. We kunnen dan het algoritme recursief toepassen op deze twee deelproblemen. Het resultaat van beide problemen dient dan gecombineerd te worden tot alle mogelijke combinaties van de invullingen van beide interne randen.

Dit is voorgesteld in figuur 8.2. Stel dat het algoritme vertrekt in de richting van de pijl. Eens het bij de eerste 3' komt, moet het algoritme kiezen. Het kan een nieuwe top toevoegen, en dan daar proberen mee verder te gaan. Het kan in ieder geval nog niet naar de startpositie een boog trekken: dit zou slechts een vierhoek opleveren. We kunnen nu echter kiezen om een 'sprong' te maken naar een ander stuk rand. De volgende boog is ook weer een plaats om een sprong te maken, hier worden we duidelijk verplicht om terug naar de startpositie terug te keren.



Figuur 8.2: Het algoritme kan een rand opsplitsen in 2 aparte binnenranden

Zo bekomen we een zeshoek. De rand is nu duidelijk opgesplitst in 2 delen: een linker- en een rechterdeel.

Hier kunnen we opmerken dat we overbodig werk kunnen doen, indien we een slechte keuze maken. Stel dat we de rand hierdoor opsplitsen in 2 interne deelranden genaamd r_1 en r_2. We bepalen hier eerst alle invullingen van r_1, en dan combineren we deze met die van r_2. Indien echter r_1 zeer veel invullingen heeft, maar r_2 geen enkele, doen we veel te veel werk. We kunnen dit proberen te voorkomen door r_2 op invulbaarheid te controleren alvorens we de invullingen van r_1 proberen te bepalen. Doordat we, zoals bij het starten van het algoritme, gebruik maken van snelle methoden zoals randcondities en dynamisch programmeren, kunnen we zeer snel een goede gok maken betreffende de invulbaarheid van r_2. Indien deze methode als resultaat heeft dat r_2 geen invullingen heeft, zullen we niet eens proberen r_1 in te vullen.

Indien we al deze mogelijke invullingen hebben bepaald, kunnen we deze teruggeven als resultaat. Indien we geen invullingen gevonden hebben, zullen we een lege lijst moeten teruggeven, omdat alle mogelijke invullingen deze inboog moesten weghalen.

Nu dient te worden opgemerkt dat indien er toppen worden toegevoegd om een stuk rand te vervolledigen tot een vijfhoek of tot een zeshoek, we niet proberen deze toegevoegde toppen ook eens te verbinden met nog 'vrije' toppen in het huidige stuk van de binnenrand. Dit komt door de structuur van het probleem. Aangezien we immers onderstellen dat $p < 6$ is, weten we dus dat we zullen beginnen met een stuk rand van de vorm '22'. Willen we nu zo veel mogelijk keuzevrijheid in het toevoegen van nieuwe bogen en toppen, dan gaan we er van uit dat we beginnen met '223'. In dit kleinste geval zijn we dus reeds 3 bogen tegengekomen: van de startpositie naar de eerste '2', van die naar de volgende, en van daar naar de eerstkomende '3'.

Willen we nu bijvoorbeeld het minimum aantal toppen (dat is 1) toevoegen, maar wel om zo een zeshoek te vervolledigen, dan zou de gedachte kunnen zijn om langs die toegevoegde top een 'omweg' te maken langs een anders deel van de rand dat de vorm '33' heeft. Dit zal echter nooit het geval kunnen zijn. We hebben namelijk al 3 bogen. Dan voegen we nog de boog toe

tot de net toegevoegde top, zodat we reeds 4 bogen hebben. Natuurlijk moeten we van deze top naar dit ander stukje rand, en van dat stukje rand terug naar de startpositie. Dan zijn we reeds aan 6 bogen, terwijl we er nog de boog die de twee '3' toppen verbindt moeten bijtellen, zodat we aan 7 toppen komen. Dit kan natuurlijk niet in een patch. Andere gevallen kunnen om gelijkaardige redenen niet.

Deze informatie kunnen we dan ook in het algemeen gebruiken bij de stap waar we stukken van de rand gaan combineren. We kunnen beslissen om dit niet eens te proberen, indien we reeds 4 bogen hebben bekeken, daar dan zoals reeds gezegd dan niet eens meer een zeshoek zullen kunnen vormen daar we twee bogen nodig hebben om er naar toe en weer weg van te raken, en één om dit stuk rand te passeren.

In het algemeen zeggen we dat we ofwel een vijfhoek, ofwel een zeshoek proberen toe te voegen. Dit kan op de beschreven manier worden geïmplementeerd. We kunnen echter ook de implementatie van het algoritme opsplitsen in twee delen. Het eerste deel bepaalt dan de canonische startpositie en kijkt naar de invulbaarheid in het algemeen (aantal vijfhoeken, dynamisch programmeren), terwijl het tweede deel de daadwerkelijke invullingen probeert.

Dit tweede deel kan dan echter bijvoorbeeld geen aparte 'voeg zeshoek toe' operatie hebben. Wel de mogelijkheid om een enkele boog toe te voegen, om zo een vijf- of zeshoek te vervolledigen, of een enkele top en boog toe te voegen. In dat laatste geval wordt dan het tweede deel recursief opgeroepen, in plaats van een vijf of zeshoek in te vullen.

8.4 Correctheid en eindigheid van het algoritme

Het is belangrijk te weten dat het beschreven algoritme, indien het foutloos geïmplementeerd zou worden, ook het gewenste resultaat zal teruggeven. Uit de constructie van het algoritme volgt duidelijk dat indien het algoritme een lijst teruggeeft, deze inderdaad correct zal zijn. Het is namelijk zo dat in elke invulling van een rand, elke boog zal grenzen aan een vijf-of zeshoek. Aangezien zo een vlak op slechts een paar manieren kan ingepast worden in een patch, en het algoritme deze allen bekijkt, zal dit inderdaad een correct resultaat opleveren.

Het is echter niet direct duidelijk dat het algoritme zal termineren. Inderdaad, indien het aantal vijfhoeken $p \geq 6$ is, zal het algoritme in deze vorm mogelijk niet termineren. Dit komt, zoals reeds vermeld in hoofdstuk 4, doordat dit soort randen oneindig veel invullingen kan hebben. In dat geval zal het algoritme *nooit* alle invullingen kunnen vinden, hoe lang het ook

Algorithm 3: Het algoritme om een rand in te vullen met alle mogelijke invullingen

functie: `VulPrepatchIn(prepatch, (starttop, richting))`

input : Een rand voorgesteld als tijdelijke patch G met een 'gerichte boog'

$(starttop, startrichting) \in E(G)$

output: Lijst met tijdelijke patches van alle mogelijke invullingen van die rand

Lijst resultaat ← lege lijst

'Gerichte boog' (starttop,startrichting) ← `ZoekStartPos`((starttop,startrichting))

if *niet* `ControleerInvulbaarheid`(G, (starttop,startrichting)) **then**
  return lege lijst

Zoek de eerste 'InBoog' in de rand in G, gegeven door (starttop,startrichting)

if *Geen 'InBoog' gevonden* **then**
  if *het vlak was een vijf- of zeshoek* **then**
    return { `KopiëerPatch`(G) }
  else
    return lege lijst

(huidigetop,richting) ← de eerste 'InBoog' die we vonden

forall $0 \leq i \leq 2$ **do**
  Voeg vanaf (huidigetop,richting) i nieuwe toppen toe
  Probeer een vijf- of zeshoek te maken door de laatste top met de begintop te verbinden:
  resultaat ← resultaat ∪ `VulPrepatchIn`(`VeelhoekAanRand`(G, (starttop,startrichting)), (richting, huidigetop))

if *3 bogen gezien* **then**
  Voeg een zeshoek toe die langs een ander stuk van de rand gaat:
  forall *toppen 't' die als buur een 'InBoog' in het huidige vlak hebben* **do**
    if *(t ==* starttop*) of (t is reeds buur van* starttop*)* **then** continue
    G' ← `VoegBoogToe`(`KopiëerPatch`(G), huidigetop, t)
    if *niet* `ControleerInvulbaarheid`(G', *(*richting, *(*richting, huidigetop*))*) **then**
    continue
    forall *prepatch prepatch uit de resultatenlijst van* `VulPrepatchIn`(G', (huidigetop,richting)) **do**
      resultaat ← resultaat ∪ (*prepatch*, (richting, huidigetop))

return resultaat

zou blijven draaien.

Indien we het aantal vijfhoeken tot $p < 6$ beperken, zal het algoritme echter wel degelijk termineren. Eerst en vooral hebben we in hoofdstuk 4 bewezen dat indien $p < 6$, er geen prepatch is waar de binnenrand gelijk is aan de buitenrand door toevoeging van zeshoeken. Nu is het eenvoudig aan te tonen dat het algoritme inderdaad zal eindigen door te bewijzen dat het algoritme de rand zo zal manipuleren dat de rand nooit zal groeien.

Lemma 8.4.1. *De randen in de recursieve oproepen van algoritme 3 zullen nooit groeien indien* $p < 6$.

Bewijs. Aangezien $p < 6$ is, zal de rand steeds een deelsequentie met '22' bevatten. Onze canonische startpositie zal dus ook steeds zo gekozen zijn dat het algoritme start met het overlopen van twee bogen die eindigen op toppen die geen 'inbogen' hebben. De enige manier om dit stuk rand aan te vullen zodat de rand in de recursieve oproepen niet groeit, is indien we beginnen met '223', waardoor we hier een zeshoek kunnen plaatsen. Alle andere mogelijkheden om op deze plaats een vijf- of zeshoek te plaatsen zullen de nieuw gecreëerde randen kleiner laten worden ten opzichte van de oorspronkelijke (interne) rand. □

Stelling 8.4.2. *Het algoritme zal altijd eindigen indien* $p < 6$ *is.*

Bewijs. Voor een bepaalde randlengte bestaan slechts een eindig aantal mogelijke randen. De rand zal nooit zal groeien (dankzij lemma 8.4.1). Bovendien weten we dankzij stelling 4.0.8 dat er geen prepatch zal zijn waavoor we een binnenrand hebben die gelijk is aan de buitenrand. Merk op dat de rand alleen van gelijke lengte kan blijven als we een prepatch vormen: indien we een prepatch aanvullen zodanig dat het enkel nog een tijdelijke patch is, zal de randlengte kleiner geworden zijn. Hierdoor kunnen we slechts een eindig aantal randen van eenzelfde lengte op deze manier tegenkomen en zal het algoritme steeds eindigen. □

8.5 Het geval $p = 6$

De voorgaande bespreking ging uit van het geval waar $p < 6$ is. Indien $p = 6$, kan het zoals reeds in hoofdstuk 4 besproken, gebeuren dat er oneindig veel invullingen zijn.

Om het oneindige doorlopen tegen te gaan, kan worden geëist dat in het geval $p = 6$ er steeds een vijfhoek moet worden geplaatst in de buitenste rand. We plaatsen dan de vijfhoek op *alle* mogelijke plaatsen op de rand. Dan roepen we recursief het algoritme op. Vanaf $p < 6$

is, zal dan het normale algoritme overnemen. In het geval dat we de buitenrand bedekken met een vijfhoek, kan wel nog rekening worden gehouden met de symmetrie van de rand, om niet te veel werk te doen. Als de rand namelijk een cyclische verschuiving als symmetrie bevat, kan de vijfhoek worden geplaatst op een bepaalde positie. We kunnen er dan echter op letten dat we niet weer een vijfhoek plaatsen op een plaats die 'hetzelfde' is onder deze symmetrie: we plaatsen geen vijfhoek meer op de plaats van de eerste positie die getransformeerd wordt door deze verschuiving.

Natuurlijk kunnen dan niet meer alle mogelijke invullingen van deze rand worden bereikt. Aangezien het echter a priori onmogelijk is om in alle gevallen alle invullingen te geven van dit soort randen, is dit sowieso een verbetering ten opzichte van het gewone algoritme.

8.6 Snelheid

We kunnen nu onderzoeken hoe de besproken technieken om het programma zo snel mogelijk te laten draaien, in de praktijk werken. Hiertoe zullen enkele metingen worden gedaan op het algoritme, waar bepaalde opties al dan niet zijn ingeschakeld. De tijd is gemeten bij het invullen van alle canonische randen van bepaalde lengtes. Deze werden door het algoritme uit hoofstruk 7 gegenereerd. Enkel de tijd die benodigd was voor het invullen van de randen werd gemeten. Het is vooral interessant hoeveel sneller de resultaten zijn *relatief* ten opzichte van mekaar. De metingen zijn samengevat in tabel 8.3.

Eerst en vooral kijken we hoe het gewone dynamische programmeren zal presteren. We hebben hiervoor de C++ templateklasse `std::map` gebruikt, dat intern meestal wordt geïmplementeerd door een rood-zwart boom. Om te kijken in hoeverre het aantal randen dat in deze boom zit het programma zal versnellen, splitsen we deze metingen op in 4 gevallen. In eerste geval wordt helemaal geen dynamisch programmeren toegepast. In het tweede en derde geval, gaan we gradueel het aantal canonische randvormen dat bijgehouden wordt verhogen. Tot ten slotte in het vierde geval alle canonische vormen worden bijgehouden. We zien dat het aanzetten van dynamisch programmeren voor 'kleine' randlengtes relatief weinig verschil zal uitmaken. Eens we echter ook langere randen gaan bijhouden, wordt het verschil zeer zichtbaar: van 52s in het niet-dynamisch programmeren geval gaan we naar 25s waar alle canonische vormen worden bijgehouden.

Vervolgens gaan we het crossprocess dynamisch programmeren introduceren. We implementeren dit als volgt: we hebben een cross-process bestand waarvan we de inhoud met de `mmap` functie op het geheugen gaan afbeelden. Indien we de informatie willen opvragen van een ca-

nonische rand, zullen we de juiste positie in het bestand bepalen, en deze dan uitlezen in deze afbeelding. Omgekeerd, indien we nieuwe informatie over een rand verkrijgen, zullen we meteen deze informatie op de juiste positie in deze afbeelding wegschrijven.

We beginnen met de metingen op een *leeg* crossprocess bestand. Er wordt dus niet begonnen met crossprocess informatie: deze wordt met dit process pas geïnitialiseerd. We merken op dat dit reeds sneller zal zijn dan gewoon een gebalanceerde boom te gebruiken om de randen te bewaren. Dit komt doordat de boom steeds een heel pad moet doorlopen alvorens hij de informatie heeft. Bij het crossprocess bestand kan echter direct een index worden bepaald, waarna het slechts 1 leesactie kost om de gegevens op te halen.

Dan kunnen we kijken naar wat er zal gebeuren indien het crossprocess bestand reeds informatie bevat. Om dit te testen gaan we gewoon dit bestand gevuld hebben in het vorige proces met dezelfde parameters, om de metingen enige waarde te geven. Dit is natuurlijk slechts een simulatie van de praktijk, waar het bestand reeds gevuld zal zijn met veel meer randen van voorgaande uitvoeringen.

We zien meteen dat het gebruik van deze crossprocess informatie slechts een kleine versnelling zal geven. Een laatste techniek die we kunnen proberen is het *vooraf laden* (of preloaden) van een deel van het bestand. Dit wil zeggen dat we niet gewoon het bestand afbeelden op het geheugen, maar ook nog eens een heel stuk van het bestand direct in het geheugen laden. We zien echter dat dit slechts een klein verschil maakt.

Een laatste opmerking die dient gemaakt te worden, betreft de grootte van het bestand. Na het uitvoeren van de $l = 12..30$ en $p = 2$ test, vinden we dat het 129M groot is. Indien we dus het bestand volledig in het geheugen willen laden, zal dit 129M geheugenruimte vragen. We merken echter op dat het bestand *op schijf* een stuk kleiner is, namelijk slechts 4.9M. Dit komt doordat we slechts naar plaatsen in het bestand schrijven waar we informatie van hebben, en aangezien we slechts een relatief klein aantal van alle mogelijke canonische randen zijn tegengekomen, zien zullen we niet zoveel informatie hebben weggeschreven. Hierop kan het bestandssysteem dan inspelen: het kan intern bijhouden welke *blokken* nog niet beschreven zijn, en deze gewoon als '*leeg*' markeren, zonder er schijfruimte aan te spenderen.

Beschrijving opties	**Tijd (s)**
$l = 12..30$, $p = 2$	
Geen boom, geen crossprocess	52.21
Tot $l = 14$ gelimiteerde boom, geen crossprocess	50.88
Tot $l = 20$ gelimiteerde boom, geen crossprocess	38.21
Ongelimiteerde boom, geen crossprocess	25.06
Geen boom, lege crossprocess	22.92
Tot $l = 14$ gelimiteerde boom, lege crossprocess	25.43
Tot $l = 20$ gelimiteerde boom, lege crossprocess	25.15
Ongelimiteerde boom, lege crossprocess	25.11
Geen boom, gevulde crossprocess	20
Tot $l = 14$ gelimiteerde boom, gevulde crossprocess	20.99
Tot $l = 20$ gelimiteerde boom, gevulde crossprocess	20.42
Ongelimiteerde boom, gevulde crossprocess	20.37
Geen boom, gevulde crossprocess, preload 16M	20.23
Geen boom, gevulde crossprocess, preload 32M	19.89
Geen boom, gevulde crossprocess, preload 64M	19.84
Ongelimiteerde boom, gevulde crossprocess, preload 64M	20.16
$l = 17..21$, $p = 5$	
Geen boom, geen crossprocess	10.32
Ongelimiteerde boom, geen crossprocess	6.99
Geen boom, gevulde crossprocess	6.93
$l = 25$, $p = 5$	
Geen boom, geen crossprocess	457.68s
Ongelimiteerde boom, geen crossprocess	400.96
Geen boom, gevulde crossprocess	391.51s

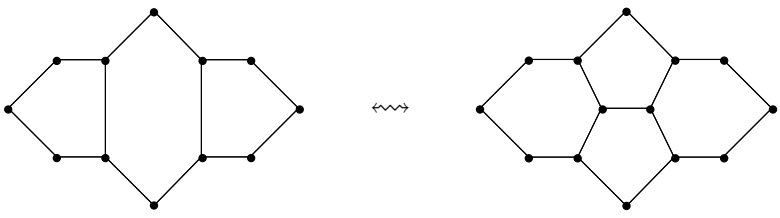
Figuur 8.3: Metingen van het invulalgoritme voor randen. Alle canonische randen met de gegeven parameters werden ingevoerd in één proces. De duurtijd van de generatie van de invullingen zelf is in seconden. Met 'lege crossprocess' wordt bedoeld dat de crossprocess file initiëel leeg is, met 'gevulde crossprocess' wordt bedoeld dat de crossprocess file al gevuld werd door het proces al eens uit te voeren en daarmee de crossprocess file al te vullen.

Hoofdstuk 9

Toepassing: groeitransformaties in fullerenen

Een toepassing van het algoritme is het zoeken van groeitransformaties in fullerenen. Een fullereen is een gesloten chemische structuur bestaande uit koolstofatomen. Elk koolstofatoom heeft exact 3 buren, zó dat de vlakken afgebakend door de koolstofatomen een rooster van vijf- en zeshoeken vormt. Een bekend voorbeeld is de C_{60} Buckminsterfullereen die de vorm heeft van een voetbal. Aangezien de aaneenschakeling van de koolstofatomen in een fullereen een rooster van vijf- en zeshoeken vormt, kunnen we delen van dit rooster beschouwen als (h, p)-patches. De koolstofatomen vormen dan de toppen, en de (enkelvoudige en dubbele) bindingen tussen de atomen zijn de bogen.

Een interessant probleem is hoe deze fullerenen kunnen groeien. Om dit uit een theoretisch oogpunt te kunnen bestuderen, is het mogelijk om het groeien van een fullereen te bekijken als een operatie waar er eerst een stuk patch uit het rooster wordt geknipt, dat daarna wordt vervangen door een patch met dezelfde rand, maar met een ander aantal koolstofatomen. Een koppel van 2 patches met dezelfde rand, maar met een verschillend aantal toppen, noemen we dan een groeikoppel. De reeds besproken Endo-Kroto C_2 invoeging zoals te zien in figuur 9.1,



Figuur 9.1: De Endo-Kroto C_2 invoeging

is hier een mooi voorbeeld van.

In [BF03] wordt deze beschrijving gebruikt om een catalogus op te stellen van een aantal (kleine) groeikoppels. We kunnen nu ons algoritme gebruiken om deze catalogus zelf op te stellen, zodat we deze onafhankelijk kunnen testen. Het spreekt voor zich dat we enkel naar niet-isomorfe invullingen van een rand zullen kijken.

In principe zouden we ook kunnen proberen te kijken naar grotere gevallen, doordat we de resultaten van de twee verschillende implementaties van de catalogus kunnen vergelijken. De klemtoon werd in dit artikel echter gelegd op relatief *kleine* randlengtes (en zodoende ook kleine patches). Dit is omdat een groeitransformatie in een fullereen een lokaal chemisch proces is. Het wordt door chemici als onwaarschijnlijk aangenomen dat er in één stap een hele grote transformatie zou optreden: dit zou immers te veel energie vragen.

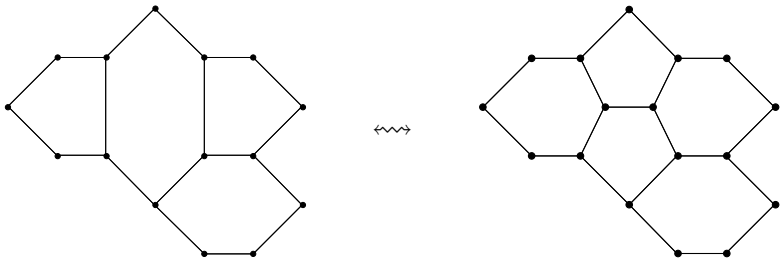
9.1 Technische details voor de catalogus

Om deze catalogus zo compact mogelijk te houden, zijn een tweetal beperkingen ingevoerd, die het aantal patches erin verminderen zonder af te doen aan het nut ervan. Het is namelijk zo dat indien we gewoonweg alle randen van een bepaalde lengte zouden bekijken, en dan gewoon alle groeiparen in de catalogus zouden stoppen, deze veel te groot zou zijn. Veel van die informatie is echter redundant, of zelfs nutteloos indien ze moet worden gebruikt voor groeiparen van fullerenen.

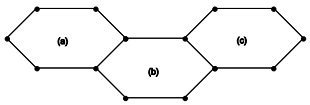
9.1.1 Irreducibele groeiparen

Een eerste beperking is de introductie van *irreducibele groeiparen*. Het probleem is namelijk dat indien we gewoon alle groeipatches zouden bekijken, er veel redundante informatie in zal zitten. Deze zit in *toeschouwervlakken* van een paar: dit zijn vlakken die een kleiner groeipaar groter maken, op zo'n manier dat we dit vlak even goed niet konden beschouwen. Dit is goed te zien in figuur 9.2. Er zit een zeshoek in de rand die duidelijk geen nieuwe informatie brengt ten opzichte van de informatie die we reeds verkregen uit figuur 9.1 Door irreducibiliteit van deze paren te introduceren, wordt dit probleem verholpen.

Een groeipaar wordt irreducibel genoemd indien er een isomorfisme van de 2 randen bestaat, die geen enkel randgedeelte van een *verwijderbaar* vlak afbeeldt op het randgedeelte van een verwijderbaar vlak van dezelfde grootte in de andere patch. Een verwijderbaar vlak is hier een vlak dat kan worden verwijderd uit de patch, zonder dat de patch hierdoor ongeconnecteerd



Figuur 9.2: De Endo-Kroto C_2 invoeging met een toeschouwervlak



Figuur 9.3: Voorbeeld van een verwijderbaar vlak. (a) en (c) zijn verwijderbare vlakken, (b) is een niet verwijderbaar vlak

raakt. De vlakken (a) en (b) in figuur 9.3 zijn voorbeelden van een verwijderbare vlakken.

De bedoeling hier van is als volgt. Neem bijvoorbeeld dat er zulk een isomorfisme bestaat tussen de randen van twee patches, en er bijvoorbeeld een zeshoek op een zeshoek wordt afgebeeld. Als deze zeshoek nu verwijderbaar is, kunnen we deze – zoals de naam reeds aangeeft – verwijderen. De verwijdering van de de zeshoek in beide patches zal er voor zorgen dat er weer twee nieuwe patches ontstaan, met nieuwe randen die aan elkaar gelijk zullen zijn. Dit is dan ook de reden dat geëist wordt dat de patch niet gedisconnecteerd raakt. Beide patches met de zeshoek verwijderd hebben echter nog steeds een isomorfisme tussen de randen. Deze twee patches vormen dus nog steeds een groeipaar, maar met precies één enkele zeshoek minder. Daardoor kunnen we zeggen dat dit groeipaar 'eenvoudiger' zal zijn dan het paar waarvan deze afkomstig zijn. Deze patches kunnen dus nog worden vereenvoudigd, waardoor we ze weg zullen laten uit de catalogus.

Merk op dat door het verwijderen van zo een verwijderbaar vlak uit een groeipaar, de resulterende randlengte zowel kan groeien, gelijkblijven of kleiner worden ten opzichte van de oorspronkelijke randlengte. Dit is geheel afhankelijk van hoe deze vlakken in de patches geplaatst zullen zitten.

Er is een specifieke reden dat de eis voor irreducibiliteit zo is dat er minstens één isomorfisme moet bestaan dat een niet-verwijderbaar vlak afbeeldt op een niet-verwijderbaar vlak van een andere grootte, opdat het paar irreducibel genoemd zou worden. Men zou kunnen denken dat het beter zou zijn dat alle isomorfismes hier zouden aan voldoen. Op het eerste zicht lijkt het zo dat indien er een isomorfisme bestaat dat twee verwijderbare vlakken van dezelfde grootte

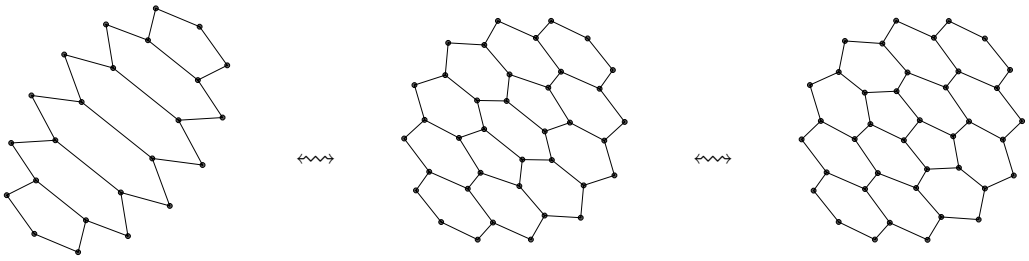
Figuur 9.4: Twee Endo-Kroto C_2 invoegingen aan mekaar

op mekaar afbeeldt, het niet veel zin meer heeft om de patches in de catalogus te stoppen. We moeten dit echter plaatsen in de context van de catalogus.

Neem bijvoorbeeld een groeipaar, waarbij er 2 isomorfismen zijn tussen de randen. Een van de isomorfismen beeldt een verwijderbaar vlak af op een verwijderbaar vlak van dezelfde grootte; het andere isomorfisme beeldt geen enkel verwijderbaar vlak af op een verwijderbaar vlak van dezelfde grootte. Het probleem is hier dat we deze patches willen inbedden in een fullereen. Ondanks dat er een isomorfisme tussen de randen is dat een verwijderbare zeshoek afbeeldt op een verwijderbare zeshoek, zal deze toch nog informatie aan de catalogus bijdragen. Het isomorfisme dat namelijk de zeshoek op een vijfhoek afbeeldt, kan in een fullereen zorgen voor een strikt verschillende fullereen. Indien we echter de rand zouden kleiner maken met het eerste isomorfisme, gooien we deze informatie weg.

Er moet verder nog opgemerkt worden dat de 'unie' van twee irreducibele groeiparen wel terug een irreducibel groeipaar kan vormen. Dit is te zien in figuur 9.4, die duidelijk opgebouwd is uit 2 Endo-Croto C_2 invoegingen, maar wel als irreducibel wordt gerekend.

In de catalogus dient voorts nog te worden opgemerkt dat irreducibiliteit van patches niet transitief is. Het kan bijvoorbeeld zo zijn dat er een irreducibel groeipaar (A, B) is, en een irreducibel groeipaar (A, C), maar dit impliceert niet dat (B, C) een irreducibel paar is. Dit is bijvoorbeeld mooi te zien op figuur 9.5. Het is echter ook goed in te zien indien we even een irreducibel paar (A, B) beschouwen (dus niet noodzakelijk een groeipaar). Kies A zo dat het minstens een verwijderbare zeshoek heeft, en dat de rand maar 1 automorfisme bezit. Dan hebben we precies één randisomorfisme ϕ van A naar B, zodat geen enkel verwijderbaar vlak van A op een verwijderbaar vlak van dezelfde grootte van B wordt afgebeeld. Nu bestaat er dus ook precies één isomorfisme ϕ^{-1} van B naar A, zodat (B, A) ook een irreduciebel paar is. Echter, (A, A) is geen irreducibel paar, aangezien de identiteitsafbeelding op de rand van A minstens één verwijderbare zeshoek op een zeshoek zal afbeelden.



Figuur 9.5: Irreducibiliteit is niet transitief. Zowel ((a), (b)) als ((a), (c)) zijn irreducibele groeiparen, maar ((c), (b)) is een reducibel groeipaar

9.1.2 Inbedbaarheid in fullerenen

Een tweede beperking is dat de patch inbedbaar moet zijn in een echte fullereen. In het bijzonder, patches met een rand die zo gevormd is dat een mogelijke inbedding minstens een zevenhoek moet bevatten, zullen nooit in een fullereen kunnen voorkomen. Dit soort randen kan onmogelijk in een fullereen gerealizeerd worden, wegens de definitie van een fullereen: deze bevat slechts vijf- en zeshoeken als vlak. Het heeft dan ook geen enkele zin dit soort patches op te nemen in een catalogus die specifiek bedoeld is om inbedbare patches te bevatten.

Een goed voorbeeld van dit soort restrictie, zijn randen met een sequentie '23^52' er in. Het probleem is dat deze randen moeten worden ingebed in een echte fullereen. Om deze rand vanuit het perspectief van de fullereen te bekijken, bekijken we de rand 'van buiten naar binnen': overal waar een '3' staat, zien we een '2', en omgekeerd. Dat wil zeggen, alle toppen van de rand die een buur hebben die naar binnen wijst hebben nu een buur die naar buiten wijst, en toppen met graad 3, toppen met graad 2 zullen worden. Hetzelfde geldt dan voor toppen met graad 2. We bekomen dus een rand met sequentie '32^53', die we niet kunnen invullen met een vijf- of zeshoek. We zullen minstens een zevenhoek nodig hebben om deze rand in te vullen aan de kant van de fullereen, iets wat tegenstrijdig is met de definitie van de fullereen.

Het is echter mogelijk dat, ondanks het wegfilteren van deze restrictie, er een 'verborgen', niet-lokaal probleem is, waardoor deze rand toch niet voor zou kunnen komen in fullerenen. Om te voorkomen dat deze randen toch worden opgenomen in de catalogus, kan worden gezocht in een aantal fullerenen, of deze randen al dan niet voorkomen in een beperkt aantal gevallen. Indien de rand inderdaad daar in zit, is er geen enkel probleem. Het probleem stelt zich pas als de rand *niet* zou gevonden worden. Dit deel is niet opnieuw geïmplementeerd: dit werd reeds onafhankelijk gecontroleerd in [Fra05].

9.2 Het gebruik van het algoritme in deze context

De catalogus is als volgt gerangschikt. Er wordt eerst en vooral onderscheid gemaakt tussen patches met een bepaald aantal vijfhoeken. Deze worden dan onderverdeeld volgens lengte: er wordt begonnen met de patches met een bepaald aantal vijfhoeken en een bepaalde randlengte. De randlengte wordt dan opgevoerd tot een bepaalde bovengrens, waarna overgegaan wordt naar de volgende aantal vijfhoeken. Per randlengte worden dan alle irreducibele koppels gegeven, waarbij geordend wordt op het aantal toppen dat de kleinste patch van een groeikoppel heeft.

Het is vrij duidelijk hoe het algoritme hier toepasbaar is. We itereren in de eerste plaats over het aantal vijfhoeken. Dan itereren we over de lengtes die we willen hebben voor dit aantal vijfhoeken. We kunnen dan het algoritme en de formules van hoofdstuk 7 toepassen. Dit algoritme zal ons alle canonische randvoorstellingen teruggeven met de gevraagde lengte en aantal vijfhoeken. Elk van deze randen vullen we dan in met het algoritme uit 8.3. Dan bepalen we met extra functies welke unieke irreducibele groeiparen dit oplevert.

Dan rest ons nog te controleren of beide resultaten correct zijn. We kunnen dit eenvoudig doen door de uitvoer van het nieuwe programma en die van het oude programma zó om te zetten dat we deze eenvoudig kunnen vergelijken. De uitvoer van beide programma's is een lijst van groeiparen. Deze kunnen nu door een apart programma worden ingelezen. Dan kan de canonische vorm worden bepaald van elke patch van zo'n paar. Dan schrijven we per groeipaar een lijn naar een bestand: we zullen de canonische vormen wegschrijven, zó dat de kleinste canonische vorm als eerste komt.

Dan kunnen we deze bestanden sorteren, en vergelijken met mekaar. Als de bestanden dezelfde zijn, komt de uitvoer van beide programma's inderdaad overeen. We hebben deze test gedaan, en constateren dat onze resultaten overeenstemmen met die in [BF03].

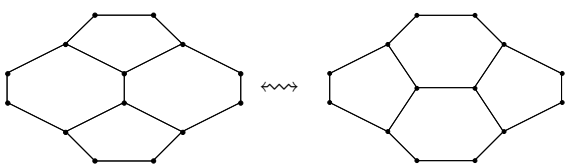
Hoofdstuk 10

Toepassing: isomerisatietransformaties in fullerenen

We kunnen hetzelfde nu doen voor isomerisatietransformaties van fullerenen. Dit wordt beschreven in [BFJ03]. Hier wordt een patch vervangen door een patch met eenzelfde rand en evenveel toppen (dus koolstofatomen). Door deze patches te verwisselen in een fullereen, bekomen we echter een verschillende fullereen. Een voorbeeld hiervan is de Stone-Wales isomerisatietransformatie, zoals te zien is in figuur 10.1

Ook hiervoor is reeds een catalogus opgesteld, die we nu op eenvoudige wijze onafhankelijk kunnen nakijken, net zoals bij de groeitransformaties. Het probleem van isomerisatie is echter iets uitgebreider dan bij de groeitransformaties. Er kunnen namelijk op twee verschillende manieren isomerisatietransformaties worden verkregen, in tegenstelling tot alleen een groeipaar van patches.

Eerst en vooral moeten we verduidelijken wat we precies bedoelen met een isomerisatietransformatie. We willen een bepaalde fullereen omzetten naar een andere, strikt verschillende fullereen, die exact evenveel koolstofatomen heeft als de fullereen waarmee we gestart zijn. Dit



Figuur 10.1: De Stone-Wales isomerisatietransformatie

kan dus gemoddeleerd worden door uit een fullereen een patch te halen, en deze patch te vervangen door een patch met dezelfde rand en evenveel toppen. Het is echter belangrijk om op te merken dat deze patch *niet verschillend* hoeft te zijn. Immers, het kan zijn dat door gewoon de patch te spiegelen, de rand nog steeds gelijk blijft, maar dat de fullereen wel degelijk een andere structuur zal krijgen. Denk maar aan de Stone-Wales pyracyleen herschikking van figuur 10.1. Deze kan in een fullereen een groot verschil maken, doordat de interne structuur verandert.

Deze isomerisatietransformaties kunnen belangrijk zijn voor de *Isolated Pentagon Rule* (IPR). Dit is een chemische theorie die zegt dat een situatie waar in een fullereen 2 vijfhoeken aan mekaar grenzen geen stabiele configuratie is. Dit komt doordat de kromming die in een fullereen zit, voornamalijk uit de vijfhoeken afkomstig is. Door twee vijfhoeken naast mekaar te hebben, zou er lokaal te veel energie nodig zijn om deze situatie vol te houden. Met deze isomerisatie kan dan worden aangegeven dat de fullereen van een energetisch 'slechte' configuratie wordt getransformeerd in een fullereen die 'stabieler' zal zijn.

We zullen dus onderscheid maken tussen een *isomerisatiepatch* enerzijds, en een *isomerisatiepaar* anderzijds.

Om een isomerisatiepatch te hebben, moet er dus eerst en vooral een zekere symmetrie zitten in de rand zelf. Zonder deze symmetrie zouden we de rand al niet op een andere manier in de fullereen kunnen kleven. Bovendien moet de invulling van deze rand minder symmetrie bevatten dan de rand zelf, zodat we de rand kunnen veranderen, op zó een manier dat ook een strikt andere invulling in de fullereen kan plaats vinden. We willen dus voor een isomerisatiepatch dat de grootte van de symmetriegroep van de rand strikt groter is dan de grootte van de symmetriegroep van de patch die deze rand heeft.

Dit kan zeer snel gebeuren, aangezien we het bepalen van de automorfismen van de rand en de patch kunnen bepalen tijdens het berekenen van de canonische vormen van beide. Aangezien we de canonische vorm van de rand al bepalen om isomorfe randen weg te filteren, en we de canonische vorm van de patches al zullen bepalen om de isomorfe patches weg te filteren, is deze informatie bijna 'gratis'.

Bovendien willen we net als voorheen dat we geen patches in de catalogus gaan opnemen, die in principe overbodig zijn. Ook hier zullen we dus enkel de irreduciebele patches in de catalogus steken. We gebruiken precies dezelfde definitie voor irreduciebel, behalve dat we hier praten over een automorfisme van de rand, in plaats van een isomorfisme tussen twee randen.

Verder hebben we dus ook de isomerisatieparen. Deze zijn gelijkaardig met de groeiparen:

het zijn twee niet-isomorfe patches met een zelfde rand, maar met een niet-isomorf binnenste. Ook hier worden op dezelfde manier als bij groeiparen de irreducibele paren bepaald.

Ook hier kunnen we op dezelfde manier controleren of de isomerisatieparen dezelfde zijn als die bekomen in het artikel [BFJ03]. Enkel voor de isomerisatiepatches moeten we een kleine aanpassing maken aan de code, aangezien er nu slechts één patch per regel uitvoer zal mogen staan.

Ons besluit is hier ook dat onze resultaten overeenstemmen met die in het artikel.

Hoofdstuk 11

Conclusie en verder onderzoek

11.1 Conclusie

We kunnen besluiten dat we snelle algoritmes voor het genereren van canonische randen en voor het invullen van deze randen kunnen implementeren. Wat belangrijk is, is dat de algoritmes inderdaad correct zijn. Wanneer $p < 6$ hebben we verder bewezen dat het algoritme om randen in te vullen eindigt, terwijl we zien dat indien $p = 6$, de structuur van het probleem voor een oneindig aantal invullingen kan zorgen. Met deze algoritmes kunnen we dan interessante resultaten onafhankelijk controleren.

11.2 Verder onderzoek

Het gegeven algoritme om randen in te vullen, is essentieel een exponentieel algoritme. Een heel belangrijke vraag die kan worden gesteld is of dit inherent is aan het probleem. De vraag of een patch met $p < 6$ een invulling heeft, zou bijvoorbeeld **NP**-compleet kunnen zijn, maar het kan bijvoorbeeld even goed een **P** probleem zijn. Evenzo is het interessant zich af te vragen wat de complexiteit is van de vraag 'Hoeveel verschillende invullingen heeft deze rand in totaal'?

Het probleem leunt vrij dicht aan bij het **TILING** probleem. Hier moet een groot vierkant worden opgevuld met kleinere vierkantjes van gelijke grootte, zodanig dat kleuren die zijn aangebracht op de randen van de vierkantjes overeenkomen met die van hun buren, en met een gegeven onderverdeling in kleuren van de rand van het grote vierkant. Dit probleem is bekend **NP**-compleet te zijn (zie bijvoorbeeld [van97]). Dit verschilt echter als zodanig van ons probleem, dat er een onbeperkt aantal van verschillende soorten vierkantjes is, tegenover slechts een vijfhoek en een zeshoek als mogelijke tegels.

Een ander gerelateerd probleem dat net als het onze slechts 2 verschillende soorten tegels heeft, is de vraag of een stuk van het vlak kan betegeld worden met een paar van polyomino's. Het beslissingsprobleem of een gegeven eindige deelverzameling van het vierkante rooster een invulling heeft met rechte trimino's en vierkante tetromino's is **NP**-compleet (zie [MR01]). Het gelateerde telprobleem is ook **#P**-compleet. Het grote verschil met ons probleem is hier echter dat er ook 'gaten' mogen zitten in het gebied dat moet worden opgevuld. Een patch, daarentegen, bevat geen enkel begrensd vlak dat geen vijf- of zeshoek is.

Indien het probleem **NP**-compleet zou blijken te zijn (en $\mathbf{P} \neq \mathbf{NP}$), dan zou dit betekenen dat er onmogelijk een polynomiaal algoritme zou kunnen worden gevonden. Dit is natuurlijk een zeer belangrijke vraag.

Bibliografie

[BF03] G. Brinkmann and P.W. Fowler. A catalogue of growth transformations of fullerene polyhedra. *J. Chem. Inf. Comput. Sci*, (43):1837–1843, 2003.

[BFJ03] G. Brinkmann, P.W. Fowler, and C. Justus. A catalogue of isomerization transformations of fullerene polyhedra. *J. Chem. Inf. Comput. Sci*, (43):917–927, 2003.

[BHZ87] Ravi B. Boppana, Johan Håstad, and Stathis Zachos. Does co-NP have short interactive proofs? *Inf. Process. Lett.*, 25(2):127–132, 1987.

[Bri92] Gunnar Brinkmann. Generating cubic graphs faster than isomorphism checking. *Preprint*, 1992.

[CR79a] Charles J. Colbourn and Ronald C. Reed. Orderly algorithms for generating restricted classes of graphs. *Journal of Graph Theory*, 3:187–195, 1979.

[CR79b] Charles J. Colbourn and Ronald C. Reed. Orderly algorithms for graph generation. *Intern. J. Computer Math.*, 7:167–172, 1979.

[Die00] Reinhard Diestel. *Graph Theory.* Springer-Verlag, derde edition, 2000. electronic version, http://www.math.uni-hamburg.de/home/diestel/books/graph.theory/.

[Far78] I. A. Faradžev. Constructive enumeration of combinatorial objects. In *Problèmes combinatoires et théorie des graphes (Orsay, 9-13 Juillet 1976). Colloq. Internat. du C.N.R.S., No. 260*, pages 131–135, 1978.

[Fra05] Dieter Franceus. Groei van fullerenen. Master's thesis, Universiteit Gent, 2005.

[GT87] Jonathan L. Gross and Thomas W. Tucker. *Topological Graph Theory.* John Wiley & Sons, 1987.

[HW74] John E. Hopcroft and J. K. Wong. Linear time algorithm for isomorphism of planar graphs (preliminary report). In *STOC*, pages 172–184, 1974.

[JBG03] Gunnar Brinkmann Jörn Bornhöft and Juliane Greinus. Pentagon-hexagon-patches with short boundaries. *Eur. J. Comb.*, 24(5):517–529, 2003.

[MF96] D.E. Manopoulos and P.W. Fowler. Downhill on the fullerene road: a mechanism for the formation of C_{60}. *The Chemical physics of the fullerenes 10 (and 5) years later*, 1996.

[Mil80] Gary L. Miller. Isomorphism testing for graphs of bounded genus. In *STOC*, pages 225–235, 1980.

[MR01] Cristopher Moore and J. M. Robson. Hard tiling problems with simple tiles. *Discrete & Computational Geometry*, 26(4):573–590, 2001.

[Rea78] Ronald C. Read. Every one a winner or how to avoid isomorphism search when cataloguing combinatorial configurations. *Annals of Discrete Mathematics 2*, pages 107–120, 1978.

[Sch88] Uwe Schöning. Graph isomorphism is in the low hierarchy. *J. Comput. Syst. Sci.*, 37(3):312–323, 1988.

[van97] Boas van Emde. The convenience of tilings. In *Complexity, Logic, and Recursion Theory, Marcel Dekker, Inc.* 1997.