Genetisch Programmeren met de Relationele Algebra

Eindwerk voorgedragen tot het behalen van de graad van bachelor in de informatica

Joris Gillis

Promotor: Prof. dr. Jan Van den Bussche

Academiejaar 2006-2007

Abstract

Multirelationele data mining betreft het zoeken van patronen over verzamelingen tupels, in tegenstelling tot het “klassieke” data mining dat patronen zoekt op één of enkele tupels, zoals voldoet een tupel aan een klassificatie of ligt een tupel dicht bij een ander tupel. We gebruiken Genetisch Programmeren als benadering om structurele verbanden te ontdekken tussen meerdere relaties en in relaties. Dit is één van de problemen in multirelationele data mining.

Genetisch Programmeren wordt aangewend om relationele algebra queries te induceren uit een verzameling voorbeelden. Voorbeelden bestaan uit een database, een verzameling relaties, en een oplossingsrelatie. Het inductieproces wordt beperkt tot structurele queries die structurele verbanden ontdekken.

We tonen experimenteel aan dat Genetisch Programmeren in staat is om een query te induceren.

Voorwoord

> "The question of whether a computer can think is no more interesting than the question of whether a submarine can swim."
> — *Edsger Dijkstra*

Ongeveer anderhalf jaar geleden was ik diep onder de indruk van John Koza's "invention machine". Hij slaagt erin om een cluster van computers "uitvindingen" te laten doen. Hij gebruikt daar twee ingrediënten voor: Genetisch Programmeren, en een computer cluster. Ik had mijn enthousiasme misschien moeten temperen. Machine Learning is nu eenmaal niet zo een deterministisch gegeven. Er is nogal wat trial-and-error en tweaking vereist. Bovendien beschikte ik niet over de uitgebreide rekenkracht die Koza wel heeft.

Gelukkig heeft mijn promotor, prof. dr. Jan Van den Bussche, mij op het juiste pad gezet. Ik ben hem heel dankbaar voor de vele besprekingen, en uren leeswerk die hij heeft geïnvesteerd in deze scriptie. Het schrijven van deze scriptie was een verhelderende ervaring die mij veel heeft bijgeleerd over Machine Learning, databases en verzamelingenleer.

Ik wil ook Geert Jan Bex bedanken voor zijn advies over Genetische Algoritmes en Genetisch Programmeren, en zijn correcties in hoofdstuk 3.

Ik bedank mijn ouders en vriendin om allerlei redenen. Bedankt voor de steun, de aanmoedigingen en deze tekst te lezen en te verbeteren.

> "Progress is possible only if we train ourselves to think about programs without thinking of them as pieces of executable code."
> — *Edsger Dijkstra*

Inhoudsopgave

Abstract **iii**

Voorwoord **v**

1 Inleiding **1**
1.1 Probleemstelling . . . 1
1.1.1 Genetisch Programmeren . . . 1
1.1.2 Multirelationele Data Mining . . . 1
1.1.3 Relationele Algebra . . . 2
1.1.4 Doelstelling . . . 2
1.2 Toepassingen . . . 2
1.3 Voorgaand werk . . . 3
1.4 Opbouw van de Tekst . . . 3

2 Machine Learning **5**
2.1 Machine Learning . . . 5
2.2 Hypothese . . . 6
2.3 Hypotheseruimte . . . 8
2.4 Machine Learning vs. Artificial Intelligence . . . 9

3 Genetische Algoritmes en Genetisch Programmeren **11**
3.1 Genetische Algoritmes . . . 11
3.1.1 Geschiedenis en Gebruik . . . 11
3.1.2 Populatie en Iteratie . . . 12
3.1.3 Generische vorm . . . 14
3.1.4 Voorbeeld: Knapzak Probleem . . . 20
3.1.5 Zoekruimte en Lokale Maxima . . . 22
3.1.6 GA en Alternatieven . . . 22
3.1.7 Schemastellingen . . . 24
3.1.8 De Schemastelling van Holland . . . 28
3.1.9 Nog meer Schemastellingen . . . 29
3.2 Genetisch Programmeren . . . 30
3.2.1 Geschiedenis . . . 30
3.2.2 Representatie probleem . . . 30
3.2.3 Genotype en Fenotype . . . 31

3.2.4 Terminals en Functies 31
3.2.5 Initiële populatie 32
3.2.6 Genetische operaties 33
3.2.7 Fitheidsfunctie en Testcases 37
3.2.8 Schemastelling 39

4 Databases **41**
4.1 Databases en Relaties 41
4.2 Relationele Algebra 43
4.2.1 Syntaxis 43
4.2.2 Semantiek 43
4.3 Het Probleem 44
4.3.1 Definitie 44
4.3.2 Tekortkomingen van de Relationele Algebra 45
4.3.3 Oplossing Tekortkomingen 45
4.4 Cylindrische Algebra 46
4.4.1 Syntaxis 46
4.4.2 Semantiek 46
4.4.3 Uniformeren en Uitbreiden 47
4.5 Equivalentie Relationele Algebra en Cylindrische Algebra 48
4.6 Query Learning 54
4.7 Multirelationele Data Mining 55
4.8 Genetisch Programmeren met de Relationele Algebra 55
4.8.1 Leerstructuur: Populatie en Expressieboom 55
4.8.2 Fitheidsfunctie 56
4.8.3 Genetische Operatoren 56
4.8.4 Selectieve Evaluatie 57

5 Software en Experimenten **59**
5.1 Software 59
5.1.1 Opbouw 59
5.1.2 Configuratie 60
5.1.3 Databases en Oplossingen 60
5.2 Experimenten 61
5.2.1 Begrippen 61
5.2.2 Experiment 1: $R \cap S$ 62
5.2.3 Experiment 2: $(R \cup S) \cap T$ 63
5.2.4 Experiment 3: Structurele query 65

6 Conclusie en Verder Onderzoek **69**
6.1 Conclusie 69
6.2 Verder Onderzoek 70
6.2.1 Vragen 70
6.2.2 Nieuwe Technieken 71

Bibliografie **75**

Hoofdstuk 1

Inleiding

Deze scriptie handelt over de automatische inductie van queries uit voorbeelden. In dit hoofdstuk schetsen we het probleem en vervolgens, hoe we dit probleem trachten op te lossen en wat er uit de oplossing volgt.

1.1 Probleemstelling

In deze tekst passen we *Genetisch Programmeren* toe op de *relationele algebra* om *relational data mining* te doen. In deze sectie introduceren we een aantal relevante begrippen.

1.1.1 Genetisch Programmeren

Genetisch Programmeren (ook wel GP) is een Machine Learning techniek (zie hoofdstuk 2) die gebruik maakt van het biologische begrip “survival of the fittest”. Kort samengevat, we halen inspiratie uit de genetica en gebruiken deze kennis om de computer automatisch te laten programmeren. Dit onderwerp zullen we grondiger behandelen in hoofdstuk 3.

1.1.2 Multirelationele Data Mining

Multirelationele data mining, ook wel relationele data mining genoemd, is een nieuwe vorm van data mining. Data mining is het automatisch doorzoeken van grote hoeveelheden data om er kennis uit te puren. Traditioneel wordt data mining (of Knowledge Discovery in Databases of KDD) toegepast op één enkele relatie. In deze tekst gaan we echter gebruik maken van meerdere relaties. In hoofdstuk 4 definiëren we een database als zijnde een verzameling van tabellen. In de klassieke KDD worden er a.h.v. de data classificatiemethoden geconstrueerd, of worden de data gegroepeerd in clusters van gelijkaardige objecten.

We zijn echter niet geïnteresseerd in zulke technieken in deze tekst. Wij willen verbanden in de database bloot leggen. Dit is relational data mining.

1.1.3 Relationele Algebra

We willen de computer laten programmeren, en we willen de computer laten werken op databases. Er ontbreekt dus nog een essentiëel stuk, nl. een programmeertaal voor databases. Deze is gemakkelijk gevonden. Codd, de vader van de moderne relationele database, heeft eveneens de relationele algebra ontwikkeld. Een *query taal* om databases te ondervragen. Deze taal kunnen we gebruiken als programmeertaal.

1.1.4 Doelstelling

We ontwikkelen een algoritme dat gegeven een verzameling voorbeelden een query induceert die het gevraagde verband abstract beschrijft in termen van relationele algebra. Een verzameling voorbeelden bestaat in dit geval uit koppels. Een koppel bestaat uit een database (een verzameling tabellen) en het resultaat van de query uitgewerkt voor de database. M.a.w. we geven het algoritme een beschrijving van wat de query moet doen a.h.v. voorbeelden en verwachten dat het algoritme de query induceert.

Merk op dat dit het omgekeerde is van query evaluatie. *Query evaluatie* is een onderwerp waarop reeds zeer veel onderzoek is uitgevoerd. Kijk bijvoorbeeld naar de databases, deze bevatten allemaal ingenieuze algoritmes om zo snel mogelijk van een query naar het resultaat te komen. Over het omgekeerde, wat we *query afleiding* noemen, is veel minder bekend. Toch is dit een interessant probleem omdat er veel toepassingen zijn die steunen op dit probleem, zie 1.2.

Wat verwachten we van ons algoritme? We verwachten dat het algoritme in staat is om de query aan te induceren, als het beschikt over "genoeg" en "goede" voorbeelden. Hoofdstuk 5 geeft meer uitleg over wat genoeg en goede voorbeelden zijn.

1.2 Toepassingen

De meeste toepassingen liggen in KDD systemen, aangezien we een data mining problemen trachten op te lossen. Het zou ook in "huis-tuin-keuken" database systemen gebruikt kunnen worden als een extreme vorm van "Query-By-Example".

Om het algoritme toe te passen hebben we veel data nodig, en een verband waarvan we weten dat het zich schuilt houdt in de data. Een voorbeeld van zulk een situatie is het ministerie van Financiën. Zij beschikken over de belastingsaangiften (= veel data), en we mogen er zeker van zijn dat er redelijk wat fraudeurs verstopt zitten in deze data (= verband). De minister van Financiën zou nu kunnen beslissen om zijn mensen in een fragment van de database alle fraudeurs te laten zoeken. Daarna kan het algoritme een query afleiden die mogelijke fraudeurs uit de belastingsaangiften filtert.

1.3 Voorgaand werk

Ryu en Eick hebben een knowledge mining systeem, MASSON, voorgesteld [25]. Hun systeem werkt op een object-geörienteerde database. Zij concentreren zich echter meer op classificatie aangezien ze niet werken met structurele queries, en vergelijkingen met constanten toelaten.

Acar en Motro stellen ook een systeem voor dat Genetisch Programmeren gebruikt [1]. Zij laten echter alleen unie, intersectie, verschil en selectie toe. Hierdoor beperken ze in grote mate de uitdrukkingskracht van de queries die hun systeem kan genereren.

1.4 Opbouw van de Tekst

In hoofdstuk 2 geven we een introductie tot het onderzoeksdomein Machine Learning. Dit is een zeer korte, top-level beschrijving van wat Machine Learning inhoudt, wat de basisbegrippen zijn, en waarvoor het gebruikt kan worden. We illustreren dit met een aantal voorbeelden.

Hoofdstuk 3 handelt over het Genetisch Algoritme en Genetisch Programmeren. Dit geeft een langere introductie tot het onderwerp, ook a.h.v. een aantal voorbeelden. Enkele belangrijke problemen en hun mogelijke oplossingen worden aangehaald. Het doel van dit hoofdstuk is de lezer voldoende vertrouwd te maken met Genetisch Programmeren, zodat de software en de experimenten begrijpbaar worden.

Tot dan toe hebben we het voornamelijk gehad over de methode om query afleiding te implementeren. In hoofdstuk 4 definiëren we de relationele algebra, het type database en de soort queries die we gaan gebruiken.

Wat ons brengt tot de experimenten in hoofdstuk 5. We stellen een aantal experimenten op en bestuderen de resultaten.

In het laatste hoofdstuk worden de resultaten van deze tekst samengevat en komen we tot een conclusie. We geven tevens een aantal voorstellen voor toekomstig onderzoek, en een aantal verbeteringen die kunnen aangebracht worden aan de software, zodat deze sneller werkt.

Hoofdstuk 2

Machine Learning

Genetisch Programmeren (GP) vindt zijn oorsprong in het Genetische algoritme (GA). GP werd ontwikkeld en beschreven door John R. Koza [15]. Koza is nog steeds actief betrokken bij de ontwikkeling van GP. Hij heeft in de loop der jaren vier boeken gepubliceerd over zijn vooruitgang inzake Genetisch Programmeren.

GA's en GP zijn beide Machine Learning (ML) technieken. Om de betekenis van GP beter te begrijpen is het essentieel dat we vertrouwd zijn met de basisconcepten van ML.

Dit hoofdstuk introduceert Machine Learning en Genetische algoritmen, en is gebaseerd op het boek "Machine Learning" van Mitchell [19].

2.1 Machine Learning

Machine Learning (ML) is een multidisciplinair gebied. Het is gebaseerd op kennis uit de volgende theorieën (niet exhaustief): kanstheorie en statistiek, informatietheorie, neurobiologie, ... Men hoeft echter geen expert te zijn in al deze vakken, ML bouwt verder op bepaalde resultaten uit deze vakken.

Leerprobleem Het is belangrijk dat we een goede definitie opstellen van wat leren is. Dit is noodzakelijk om aan te tonen dat een algoritme leert. De definitie die wij zullen gebruiken is de volgende:

Definitie 2.1. *Een algoritme leert van een ervaring E ten aanzien van een klasse taken T en de performatiemaat P als zijn performantie in taken T, zoals gemeten door P, verhoogt met de ervaring E.*

Deze definitie is vaag over wat P en E inhouden. Dit komt omdat een ervaring een andere betekenis heeft voor elk probleem, en de performantie van elk probleem op een andere manier opgemeten wordt.

Een tweede mogelijke verklaring voor de vaagheid van de definitie is het ontbreken van een theorie over leren.

Laten we een voorbeeld beschouwen van de invulling van deze definitie. Dit probleem wordt later verder uitgewerkt.

***Voorbeeld 1: Leerprobleem*: "Slaagkans van een student"**

Taak Voorspel of een student slaagt of niet

Performantiemaat Het percentage correct voorspelde studenten, uit een testgroep

Ervaring Tabel met studenten en of ze al dan niet geslaagd zijn

□

Het doel van Machine Learning is om algoritmen te vinden die leerproblemen oplossen.

2.2 Hypothese

Leren bestaat uit het vormen van een model over het te leren onderwerp. Als we een concept willen leren, zoals b.v. "Slaagkans van een student"(zie 2.1), stellen wij als mens een mentaal model op van welke factoren dit concept beïnvloeden en hoe ze interageren. ML gaat op gelijkaardige manier te werk. Afhankelijk van het gebruikte algoritme gebruiken we een *leerstructuur* die overeenkomt met het mentale model dat een mens opstelt tijdens het leren. Een algoritme leert, als het een juiste invulling van de leerstructuur kan vinden voor een leerprobleem. Enkele voorbeelden van zulke leerstructuren zijn: wiskundige vergelijkingen met variabelen, beslissingsbomen, neurale netwerken, ...
Een invulling van een leerstructuur wordt een *hypothese* genoemd. We bekijken eerst twee voorbeelden van leerstructuren en hoe hypothesen eruit kunnen zien voor die leerstructuren.

Voorbeeld 2: Stel dat we het leerprobleem "slaagkans van een student" (zie 2.1) willen leren, dan kunnen we een aantal variabelen definiëren die de slaagkans van een student kunnen bepalen. We kunnen vervolgens een hypothese formuleren als een eerste graadsvergelijking met een aantal variabelen. Stel dat we volgende variabelen selecteren:

- x1 = aantal studie uren
- x2 = aantal lessen bijgewoond
- x3 = gemiddeld aantal pinten per week
- x4 = favoriete getal

We kunnen nu een vergelijking opstellen:

$$x = a_0 + a_1.x_1 + a_2.x_2 + a_3.x_3 + a_4.x_4 \tag{2.1}$$

Waarbij we geïnteresseerd zijn in de waarde van de gewichten a_0 t.e.m. a_4. Een mogelijke hypothese is weergegeven in vergelijking 2.2. De betekenis van deze

hypothese is dat een student die veel studeert, veel lessen bijwoont en weinig pinten drinkt een grote kans op slagen heeft. De specifieke waarden van a_0 t.e.m. a_4 geven de juiste verhouding tussen de variabelen. Merk op dat $a_4 = 0$. Blijkbaar speelt het favoriete getal van de student geen rol in zijn slaagkans volgens deze hypothese.

$$x = 18.43 + 3x_1 + 5x_2 - 2.73x_3 + 0.x_4 \tag{2.2}$$

Een eerste graadsvergelijking geeft een continue waarde als resultaat. In dit geval zouden we kunnen stellen dat als deze waarde groter is dan 50 dat de student slaagt. □

Voorbeeld 3: Stel dat we het leerprobleem "Slaagkans van een student" (zie 2.1) willen leren met een beslissingsboom. We classificeren studenten in 2 categorieën, namelijk "Geslaagd" en "Niet Geslaagd". Hoe ziet een hypothese eruit?

We veronderstellen dat de studenten worden voorgesteld aan de hand van tupels met attributen, en dat die attributen dezelfde zijn als de variabelen uit het vorige voorbeeld.

Een hypothese voor een beslissingsboom is een ingevulde boom B. De knopen van B zijn tests. Een voorbeeld van een test is: gemiddeld aantal pinten per week < 10. De bladeren van B zijn classificaties. In dit voorbeeld zijn de bladeren van boom B ofwel "Geslaagd" ofwel "Niet Geslaagd".

Een instantie, in dit geval een student, wordt geclassificeerd door te starten in de root van de boom en door telkens de tests uit te voeren in de interne knopen, waarbij de attribuutsnamen worden vervangen door de waarde van de student in kwestie. We volgen na elke test de tak die het resultaat aangeeft. We herhalen deze procedure tot we in een blad komen. De classificatie van dat blad is de classificatie van de student. □

Een hypothese wordt opgesteld door gebruik te maken van *trainingsdata* (ook wel "ervaring" genoemd). Voor elke leerstructuur bestaan er algoritmen die zoeken naar een zo accuraat mogelijke hypothese.

Voorstelling hypothese Merk op dat we een leerprobleem op twee verschillende manieren hebben geleerd. Eénmaal door gebruik te maken van een eerste graadsvergelijking f (zie 2.1), en éénmaal door middel van een beslissingsboom B. f kan continue waarden voorspellen, maar kan slechts een lineair verband tussen de variabelen bevatten. B voorspelt discrete classificaties, maar is in staat om b.v. een trapfunctie te leren, wat moeilijker is voor f.

Elke leerstructuur heeft zo zijn voor- en nadelen. Dit is ook de reden waarom er meer dan één leerstructuur bestaat. Door de jaren heen zijn er steeds meer geavanceerde leerstructuren ontwikkeld die steeds meer kunnen leren. Het zou echter gemakkelijker zijn als er slechts één leerstructuur bestaat die alle leerproblemen aankan, en deze eveneens efficiënt kan leren. Genetisch Programmeren probeert hier een oplossing voor te bieden.

2.3 Hypotheseruimte

Als we terug kijken naar ons voorbeeld "Slaagkans van een student" uit 2.1, dan zien we dat a_0 t.e.m. a_4 één voor één oneindig veel waarden kunnen aannemen. Elke mogelijk toekenning van de variabelen resulteert in een hypothese. We kunnen dit bekijken als een vijfdimensionale ruimte, waarbij elke dimensie overeenstemt met een gewicht a_i.

Om te leren willen we komen tot een hypothese die de trainingsdata zo accuraat mogelijk benadert. In de praktijk wordt er vaak gewerkt met *trainingsdata en testdata.* Trainingsdata worden gebruikt om een hypothese te formuleren. De kwaliteit van de hypothese wordt daarna getest d.m.v. testdata. Dit gaat *overfitting* van de hypothese op de trainingsdata tegen. Overfitting is het fenomeen dat het leeralgoritme dingen aanleert die specifiek gelden voor de trainingsdata maar die geen geldige veralgemening zijn over alle data. Dit ligt echter buiten het bereik van deze inleiding.

Grosso modo bepaalt elke mogelijke leerstructuur voor elk leerprobleem een hypotheseruimte, zie het voorbeeld van de eerste graadsvergelijking. Deze ruimte bevat alle mogelijke hypothesen die in de gekozen leerstructuur voorstelbaar zijn, en bijgevolg leerbaar zijn door een leeralgoritme. Merk dus op dat het mogelijk is dat hetgeen we willen leren niet in de hypotheseruimte voorkomt. Een andere leerstructuur is aan de orde om het leerprobleem op te lossen. Leren is bijgevolg *gereduceerd tot* het *zoeken* naar de meest accurate hypothese in de hypotheseruimte.

Voorbeeld 4: Zo is het bijvoorbeeld niet mogelijk om het aantal uren dat een student sport, noem deze variabele x_5, te betrekken in de eerste graadsvergelijking in vergelijking 2.1. Als x_5 een invloed zou hebben op de slaagkans dan zou dit niet geleerd kunnen worden. Simpelweg omdat we geen manier hebben om deze variabele voor te stellen.

Hoe zoeken we in de hypotheseruimte van een eerste graadsvergelijking? We zoeken in de hypotheseruimte overeenkomstig met een eerste graadsvergelijking d.m.v. het least squares algoritme, afkomstig uit de statistiek. Dit algoritme minimaliseert de uitkomst van de waarde van de formule in vergelijking 2.3.

$$\sum_{i=1}^{n}(\hat{y}_i - y_i)^2 \tag{2.3}$$

Met $\hat{y}_i$ de voorspelde waarde voor y_i.

Beschouw een assenkruis met op de x-as de trainingsdata en op de y-as de waarden y_i die we willen leren voorspellen. Het least squares algoritme *fit een rechte* tussen de waarden y_i zo dat de afstand van elke waarde y_i tot de rechte minimaal is. Dusdoende moeten we niet elke mogelijke toekenning van x_0 t.e.m. x_4 beschouwen om een accurate hypothese te vinden in de hypotheseruimte. □

2.4 Machine Learning vs. Artificial Intelligence

Tot slot van deze sectie over ML wil ik nog wijzen op het verschil tussen het zojuist besproken ML en Artificial Intelligence (AI).

In ML wordt onderzocht hoe een computer bepaalde leerproblemen kan leren. Zoals we geïllustreerd hebben in deze sectie zijn er een aantal leerstructuren en bijbehorende algoritmen ontwikkeld. De resultaten verschillen van leerstructuur tot leerstructuur en van algoritme tot algoritme. Er zijn bepaalde leerproblemen waar al veel onderzoek naar verricht is en waarvoor zeer goede ML oplossingen bestaan, b.v.: spamfilters. Werken met ML is over het algemeen een trial-and-error proces. ML is een onderdeel van AI.

In AI onderzoekt men intelligent gedrag en leren in machines. Het ultieme doel van AI is om synthetische intelligentie te produceren. Een machine/software die de vergelijking met een mens zou doorstaan. Een algoritme dat we niet moeten vertellen hoe het moet leren zoals het geval is bij de ML algoritmen.

Hoofdstuk 3

Genetische Algoritmes en Genetisch Programmeren

Dit hoofdstuk is een inleiding tot Genetische Algoritmes en Genetisch Programmeren. De algemene structuur van deze algoritmes wordt uitgelegd, als ook de aanverwante begrippen en concepten.

De eerste sectie handelt over Genetische Algoritmes waaruit later Genetisch Programmeren is ontwikkeld. We bestuderen de generische vorm van het algoritme, en bespreken een deel van de theorie die opgebouwd is voor het algoritme.

Vervolgens richten we ons op Genetisch Programmeren. In deze tweede sectie bekijken we de voordelen van GP t.o.v. GA. We bestuderen ook hoe het algoritme werkt, en breiden de theorie van de GA uit zodat deze ook toepasbaar is op GP.

3.1 Genetische Algoritmes

In het vorige hoofdstuk hebben we in het kort besproken wat Machine Learning inhoudt. We hebben de hypotheseruimte geïntroduceerd en een aantal voorbeelden bekeken van algoritmes die deze hypotheseruimte efficiënt doorzoeken, m.a.w. niet exhaustief.

Deze sectie behandelt de Genetische Algoritmes, ook wel GA's. Dit is een klasse van algoritmes die losjes gebaseerd zijn op biologische concepten, en meer bepaald op de genetica. We noemen het een klasse van algoritmes omdat er een algemene vorm bestaat (een generische vorm) die naargelang de applicatie wordt ingevuld.

3.1.1 Geschiedenis en Gebruik

GA's werden geïntroduceerd in 1954. Nils Aall Barricelli simuleerde de evolutie van automaten die een simpel kaartspel spelen [2]. Hoewel dit het begin betekende voor GA's, werden ze pas echt populair in de jaren '70 van de vorige eeuw. John Holland, met een achtergrond in cellulaire automaten, legde de basis voor de studie van de Genetische Algoritmes [11; 12].

Genetische Algoritmes zijn *zoekalgoritmes*, om optimalisatie- en zoekproblemen op te lossen. Optimalisatieproblemen worden over het algemeen voorgesteld door een bitstring. Elke bit stelt een bepaalde parameter of een deel van een parameter voor. Meer hierover in de sectie Generische vorm (zie 3.1.3). Het doel is een toekenning van de parameters te bekomen, zodat de output van het probleem optimaal (maximaal of minimaal) is. Een mooi voorbeeld van een zoekprobleem is zoeken in een complex netwerk, zoals het World Wide Web. Menczer heeft hiervoor een techniek ontwikkeld die een combinatie van agenten en genetische algoritmes gebruikt [17; 18].

GA's worden gebruikt voor een wijde waaier aan toepassingen. Enkele van deze toepassingen zijn: opstellen van tijdroosters, neurale netwerken, protein folding, artificiële creativiteit, ... Een uitgebreidere lijst is te vinden op o.a. Wikipedia [30].

3.1.2 Populatie en Iteratie

We hebben reeds vermeld dat de Genetische Algoritmes losjes gebaseerd zijn op biologische principes. In deze deelsectie beschrijven we hoe het principe van *natuurlijke selectie* aanwezig is in het algoritme. We leggen kort de werking van het algoritme uit.

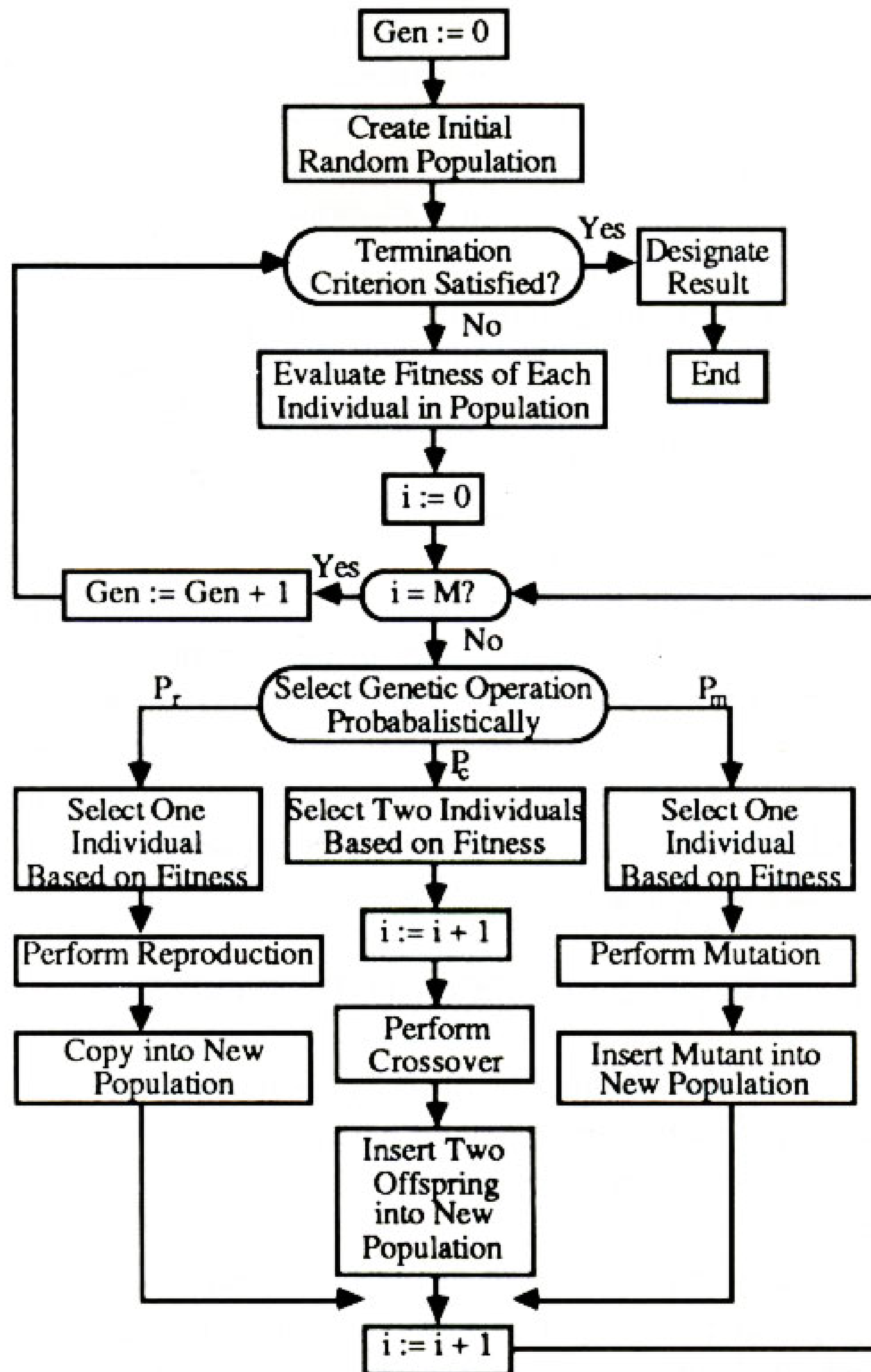
Het Genetisch Algoritme werkt met individuen. Een individu is een hypothese voorstelling, zie 3.1.3. Individuen zijn gegroepeerd in een populatie. We kunnen een populatie vergelijken met een populatie vinken, zoals in Darwins experimenten op de Galápagoseilanden [4]. De ene vink is "sterker" of meer "aangepast" om te overleven dan een andere vink. De sterkste vinken blijven langer leven en hebben meer kans om zich voort te planten. In het GA trekken we deze vergelijking door. We bepalen welke de "sterke" hypothese voorstellingen zijn, en geven deze meer kansen om te overleven. Tot zover de biologie in het GA.

Het Genetisch Algoritme is iteratief en random. Iteratie i begint met een populatie P_{i-1} en stopt met als resultaat populatie P_i. In elke iteratie wordt *één individu* als *oplossing* voor het doelprobleem gekozen.

De populatie die als input dient voor iteratie 1, wordt at random gecreëerd.

In iteratie i, wordt er gebruik gemaakt van de fitheidsfunctie en de genetische operaties (zie 3.1.3), om populatie P_{i-1} om te zetten in populatie P_i, voor $i = 1, 2, 3, \ldots$ P_i bevat *evenveel individuen* als P_{i-1}. De individuen in P_i worden gecreëerd met de verwachting dat ze betere oplossingen zijn van het doelprobleem. We kunnen echter geen uitspraken doen over hoe veel beter individuen in P_i het doelprobleem oplossen noch of ze beter zijn, aangezien de genetische operaties random werken.

Figuur 3.1 geeft een beschrijving van het GA in een flowchart.



Figuur 3.1: Flowchart van het Genetisch Algoritme. Overgenomen uit het boek van Koza [15]

3.1.3 Generische vorm

We gebruiken de naam Genetisch Algoritme (GA), om de generische vorm[1] van alle Genetische Algoritmes aan te duiden.

Het ontwerpen van een GA omvat 6 grote beslissingen:

1. Representatie van het probleem: Hypothesevoorstelling
2. Stopcriteria en toekenning van het resultaat
3. Fitheidsfunctie
4. Selectiemethode
5. Genetische operaties
6. Parameters

We bespreken deze beslissingen één voor één. Tussendoor wordt de werking van het GA uitgelegd, in het onderdeel Populatie en Iteratie. Het *doelprobleem* is het optimalisatie- of zoekprobleem dat zich stelt, en dat we oplossen door een GA te construeren.

a. Hypothesevoorstelling

Het eerste probleem dat we moeten oplossen is de voorstelling van het probleem en zijn oplossing.

Over het algemeen bestaan optimalisatieproblemen uit een reeks parameters. Een bepaalde *combinatie van waarden voor de parameters* resulteert in een optimale oplossing. De optimaliteit van een bepaalde toekenning van waarden aan de parameters wordt bepaald door de fitheidsfunctie, zie verder. De parameters kunnen in veel gevallen worden voorgesteld door een bitstring. Als we werken met booleaanse parameters, dan stelt 0 vals of afwezig voor en 1 waar of aanwezig, of vice versa. Het Genetisch Algoritme is grotendeels onafhankelijk van de gebruikte notaties, maar des te afhankelijker van de fitheidsfunctie, zoals we in het stuk Fitheidsfunctie zullen zien. Als we werken met ordinale of nominale variabelen, dan kunnen we per variabele meerdere bits toekennen.

Bij het oplossen van een zoekprobleem kunnen we als representatie een string gebruiken waarin elk karakter een beslissing voorstelt. Denk hier bijvoorbeeld aan het vinden van een pad door een doolhof. Elk karakter stelt een stap voor, en het karakter geeft aan wat voor stap moet gezet worden. De string kan een vaste lengte hebben of een variabele lengte.

Strings zijn niet de enige manier om een probleem voor te stellen. Problemen kunnen ook geformuleerd worden als bomen (zie 3.2) of andere datastructuren. Het belangrijkste is dat we de genetische operaties zo kunnen definiëren op de gekozen datastructuur zodat ze een nuttige operatie uitvoeren.

[1]Deze sectie is gebaseerd op het boek van Koza [15]

Een goede representatie is cruciaal, en vergt redelijk wat *creativiteit*. Als een bepaalde representatie een oplossing niet kan voorstellen, kan het GA deze oplossing ook niet vinden, en missen we bijna zeker een optimale oplossing. Deze stap, in combinatie met het bepalen van de fitheidsfunctie, is de moeilijkste stap in het bouwen van GA's. Niet toevallig zijn dat ook de twee stappen waarbij menselijke hulp nodig is.

b. Stopcriteria en Resultaat

Het is belangrijk dat we een grens op i kunnen plaatsen, bijgevolg op de uitvoering van het algoritme, anders zou het GA oneindig blijven doorwerken. In de natuur blijft evolutie echter altijd doorwerken, waarom laten we het GA dan niet oneindig verder werken? Omdat in de natuur het doelprobleem steeds verandert. In het GA stellen we een objectief op en proberen dit te bereiken. Eens een optimale oplossing is gevonden heeft het geen zin meer om verder te evolueren, want er zal zich toch geen optimalere oplossing voordoen. Een aantal mogelijke *stopcriteria* zijn:

- Een vastgelegd aantal iteraties.
- De fitheid van het beste individu gaat over/onder een vooropgestelde grens.
- Het verschil tussen de fitheid van het beste individu in P_{i-1} en P_i overschrijdt een vooropgestelde grens.
- De standaarddeviatie van de fitheden in P_i overschrijdt een vooropgestelde grens (op dat moment bestaat de populatie voor een heel groot deel uit identieke individuen).

Elk van deze stopcriteria heeft zijn voor- en nadelen.

In elke iteratie wordt in de populatie een *beste individu* geselecteerd, zoals gemeten door de fitheidsfunctie. Dit individu is de oplossing van het algoritme in de huidige iteratie.

c. Fitheidsfunctie

Hoe komen we nu juist van P_{i-1} tot P_i? Eerst moeten we een *onderscheid* maken tussen de individuen in een populatie. We definiëren een functie, *fitness*(x), de fitheidsfunctie. De fitheidsfunctie kent aan elk individu x een fitheid toe. De fitheid van een individu x is een indicator voor hoe goed x het doelprobleem oplost. Wat goed oplossen betekent is *probleemspecifiek*.

Hoe wordt fitheid berekend? Er zijn een aantal verschillende methoden om de fitheid te meten van een individu.

Ruw Fitheid is gemakkelijk meetbaar met testcases. Op voorhand worden een aantal testcases uitgezet, waarvan de optimale uitkomst bekend is. Elk individu wordt onderworpen aan deze testcases, en de uitkomst wordt vergeleken met de optimale uitkomst. Ruwe fitheid kan dus gemeten worden als het verschil tussen de uitkomst van de testcases en de optimale uitkomst. Ruwe fitheid kan ook gemeten worden als de uitkomst van de testcases. Afhankelijk van de keuze moet $fitness(x)$ *geminimaliseerd, dan wel gemaximaliseerd* worden. Ruwe fitheid wordt genoteerd als $r(x)$.

Gestandaardiseerd Gestandaardiseerde fitheid wordt genoteerd als $s(x)$. Als de ruwe fitheid wordt geminimaliseerd:

$$s(x) = r(x) \tag{3.1}$$

Als de ruwe fitheid wordt gemaximaliseerd:

$$s(x) = \max_{x \in P_i}[r(x)] - r(x) \tag{3.2}$$

met P_i de huidige generatie. Deze fitheidsfunctie moet *altijd geminimaliseerd* worden.

Aangepast Het grote voordeel van de aangepaste fitheid is dat deze een duidelijk verschil maakt tussen goede en zeer goede individuen. Dit is voornamelijk *op het einde van het algoritme* van belang, omdat het verschil tussen de individuen dan zeer klein wordt.
Aangepaste fitheid wordt genoteerd als $a(x)$, en voldoet aan volgende formule:

$$a(x) = \frac{1}{1 + s(x)} \tag{3.3}$$

$a(x)$ moet altijd *gemaximaliseerd* worden.

Genormaliseerd Deze fitheidsfunctie wordt vaak samen met *proportionele selectie* gebruikt.
Genormaliseerde fitheid wordt genoteerd als $n(x)$, en wordt gekenmerkt door:

- $0 \leq n(x) \leq 1$
- Hoe hoger $n(x)$, des te beter
- $\sum_{x \in P_i} n(x) = 1$

$$n(x) = \frac{a(x)}{\sum_{x \in P_i} a(x)} \tag{3.4}$$

Welke fitheidsfunctie gekozen wordt is afhankelijk van het probleem. In de rest van dit document nemen we aan dat een gepaste fitheidsfunctie gekozen is, en benoemen we deze met *fitness*(x). We spreken alleen nog over optimaliseren. *Optimaliseren* impliceert minimaliseren of maximaliseren, afhankelijk van de gebruikte fitheidsfunctie.

d. Selectie

De genetische operaties uit het volgende punt vergen selecties uit de populatie. Op dit punt steekt “survival of the fittest” terug de kop op. Elk individu in een populatie heeft een fitheid. Individuen met een hoge fitheid (hoog moet hier geïnterpreteerd worden als laag als de fitheid wordt geminimaliseerd) hebben meer kans om geselecteerd te worden. Individuen die meer worden geselecteerd worden meer betrokken in de genetische operaties. Aangezien genetische operaties de volgende populatie opbouwen, hebben fitte individuen dus meer zeggenschap in de opbouw van de volgende populatie. Ze hebben m.a.w. meer kans om te overleven en zich voort te planten.

Hoe verloopt de selectie van individuen? We introduceren 3 selectiemethoden:

Proportioneel Elk individu heeft een kans gelijk aan zijn genormaliseerde fitheid om geselecteerd te worden. Deze methode staat ook bekend als *roulette-wheel* selectie.

Als één bepaald individu alle andere domineert qua fitheid, gaat er veel *diversiteit verloren* aangezien de andere individuen in de populatie bijna geen kans hebben om zich voort te planten of te overleven. In de initiële fase van het GA is dit niet zo’n beste methode, omdat we hoogst waarschijnlijk een lokaal optimum hebben gevonden. Door meteen dit individu alle kansen te geven en aan de andere individuen voorbij te gaan, zal het algoritme niet convergeren naar een meer globaal optimale oplossing. Naar het einde van het algoritme toe, is het echter wel nuttig dat er diversiteit verloren gaat en het algoritme convergeert naar één of meerdere optimale oplossingen. Mutatie, zie paragraaf e, zorgt dat er opnieuw meer diversiteit verschijnt in een populatie.

Rang Alle individuen worden gerangschikt in een rij volgens fitheid. Het individu met de laagste fitheid krijgt 1 als fitheid toegewezen. Het tweede minst fitte individu krijgt 2. Tot we het fitste individu bereikt hebben. Daarna kan er weer proportioneel geselecteerd worden.

Deze methode verliest veel informatie over de fitheid. Als er een groot verschil is tussen twee opeenvolgende individuen in de rangorde, wordt dit niet uitgedrukt in de selectie. Als het verschil tussen individuen echter klein is naar het einde toe, zorgt dit algoritme voor een grotere kans voor de allerbesten. Ook ontwijkt deze methode het verlies aan diversiteit in de initiële fase van het algoritme. Dit is het gevolg van het deels negeren van de waarde van de fitheid.

Toernooi Twee individuen worden willekeurig gekozen uit de populatie. Het individu met de hoogste fitheid wordt geselecteerd.

Het nadeel is dat fitte individuen niet noodzakelijk gekozen worden, en we dus mogelijk goede oplossingen uit de populatie verdrijven. Door een kleine aanpassing door te voeren kunnen we dit effect echter omzeilen. Bij het creëeren van de volgende populatie laten we het beste individu zichzelf "for free" een aantal keren reproduceren in de volgende populatie. Op deze manier verliezen we het beste individu uit populatie P_i niet bij de creatie van populatie P_{i+1}. Deze strategie kan uitgebreid worden naar de top 2 of 3 individuen in P_i.

Elke methode heeft zo zijn voor- en nadelen. Al deze methoden zijn in praktijk uitgetest (o.m. Koza). Het beste is om met een aantal methodes te experimenteren in een toepassing, en diegene te kiezen die het snelste het beste resultaat oplevert.

e. Genetische Operaties

Genetische operaties zorgen voor de creativiteit in het genetisch algoritme. De operaties zijn geïnspireerd hebben hun mosterd in de biologie gehaald.

We kunnen de genetische operaties indelen in twee groepen. De seksuele en de aseksuele. Seksuele operaties vergen twee individuen, aseksuele operaties één individu.

Een tweede indeling is volgens belang van de operatie. We onderscheiden twee groepen. De primaire operaties en de secundaire operaties. De primaire operaties zijn de belangrijkste operaties en zijn het meest effectief. De secundaire operaties zijn bruikbaar, maar zijn over het algemeen minder effectief in het zoeken naar een optimale oplossing.

Voor selectie gebruiken we steeds één van de bovenstaande methoden. We bespreken nu enkele belangrijke genetische operaties.

Primaire operaties

Reproductie is een aseksuele operatie. Reproductie selecteert een individu x uit de P_{i-1} en plaatst x in P_i.

Crossover is de enige seksuele operatie die we bespreken. Deze operatie wordt ook wel seksuele recombinatie of gewoon recombinatie genoemd. We gebruiken crossover en recombinatie door mekaar in de rest van dit document. Twee ouders, zeg z_1 en z_2, worden geselecteerd uit P_{i-1}. Er wordt willekeurig een crossoverpunt p gekozen. Een eerste kind wordt gemaakt, door het stuk voor p van z_1 en het stuk na p van z_2 samen te nemen. Het tweede kind wordt gemaakt door z_1 en z_2 van plaats te veranderen. De kinderen worden in P_i geplaatst.

We hebben nu de one-point crossover geïntroduceerd. In de literatuur zijn er nog andere crossover operaties voorgesteld zoals two-point crossover. Op verschillende datastructuren zijn ook andere crossover operaties mogelijk. Dit valt echter buiten het bestek van dit document.

Voorbeeld 1: Laat ons een voorbeeld bekijken van een one-point crossover op twee bitstrings. De dollartekens in de volgende twee bitstrings geven de locatie van de crossover aan.

```
11000111$0010010010
00101010$1010010010
```

Het uitvoeren van de crossover resulteert in de volgende twee bitstrings.

```
110001111010010010
001010100010010010
```

□

Secundaire operaties In deze tekst bespreken we twee secundaire operaties, nl. mutatie en encapsulatie. We beperken ons hier tot mutatie.

Mutatie is het willekeurig veranderen van karakters in de individuen. Stel we selecteren een individu x uit P_{i-1}. We kiezen bijvoorbeeld een locatie j in een bitstring at random. We vervangen x(j), het j^e karakter in x, door het omgekeerde van x(j), en bekomen zo y. We voegen y toe aan P_i.

De primair-secundair indeling is voorgesteld door Koza [15]. Andere onderzoekers claimen echter dat mutatie noodzakelijk is in Genetische Algoritmes. Mutatie kan dus ook als een primaire genetische operatie beschouwd worden.

Voorbeeld 2: Stel dat we volgende string hebben geselecteerd uit de populatie om mutatie op uit te voeren:

```
110001110010010010
```

Stel dat we positie 9 kiezen om mutatie uit te voeren. $x(9) = 0$, door de mutatie toe te passen wordt $x(9) = 1$. Dit geeft de volgende string:

```
110001111010010010
```

□

f. Parameters

Een goed GA construeren is een *trial-and-error proces* op basis van een aantal parameters. De optimale waarden voor de parameters dienen dus bepaald te worden d.m.v. experimenten. De eerste twee worden de primaire parameters genoemd, de andere noemen we de secundaire parameters.

- *Populatiegrootte*: Hoe groot is de populatie? Een te kleine populatie mist mogelijk oplossingen, doordat we te weinig plaats hebben om te onderzoeken of oplossingen op te slaan. Een te grote populatie leidt tot lange uitvoertijden.

- *Aantal iteraties*: Als er geen ander stopcriteria is gedefiniëerd, geeft deze parameter aan hoeveel iteraties het algoritme moet uitvoeren, voordat het een resultaat geeft. Ook hier leidt een te kleine waarde tot het missen van oplossingen, en een te grote waarde tot lange uitvoertijden. Maar hoe meer iteraties hoe groter de kans dat we een globaal optimum vinden. Het algoritme langer laten uitvoeren geeft echter geen garantie op een betere oplossing, aangezien het algoritme random is.
- *Reproductiegraad*: Betreft het percentage individuen dat kan reproduceren naar een volgende iteratie.
- *Crossovergraad*: Betreft het percentage individuen in de volgende populatie, dat gecreëerd is d.m.v. recombinatie of crossover.
- *Mutatiegraad*: Betreft het percentage individuen uit de huidige populatie die gemuteerd voorkomen in de volgende populatie.

Van de secundaire parameters moeten er slechts 2 ingesteld worden. De 3e kan bekomen worden door de formule: reproductiegraad + crossovergraad + mutatiegraad = 100%.

Merk op dat er met elke genetische operatie een parameter geassocieerd is. Als we genetische operaties toevoegen, voegen we ook parameters toe. Vanzelfsprekend verandert de formule uit de voorgaande paragraaf mee.

Samenvatting

Een GA voldoet aan de volgende *condities*. Elk van deze condities hebben we besproken in de voorgaande onderdelen:

1. Een entiteit die zich kan reproduceren. De hypothese-voorstelling en genetische operaties.
2. Een populatie van entiteiten.
3. Er is variatie mogelijk tussen entiteiten in een populatie. Er zijn verschillende bitstrings.
4. Er is verschil in overlevingskansen tussen entiteiten. Dit verschil wordt bepaald door de fitheid.

3.1.4 Voorbeeld: Knapzak Probleem

Hoe werkt een GA in de praktijk? We gaan in deze sectie een klein voorbeeldje uitwerken. Als voorbeeld nemen we het *knapzak probleem*. We beschikken over een knapzak. De knapzak heeft een maximum capaciteit aan gewicht, noem dit c. Rond ons liggen objecten. Elk object heeft een gewicht en een waarde. Welke objecten nemen we mee in de knapzak, zodat het gewicht van de objecten $\leq c$, en de waarde van de objecten in de knapzak maximaal is?

	Objecten					
Objecten	a	b	c	d	e	f
Gewicht	2	4	1	7	9	3
Waarde	25	10	45	100	250	30

Tabel 3.1: Objecten die in de knapzak gestoken kunnen worden.

Formeel, stel dat O de verzameling van alle objecten is. We kiezen $I \subseteq O$ zo dat vergelijkingen 3.5 en 3.6 gelden. Waarbij we $g(x)$ definiëren als het gewicht van object x, en $w(x)$ als de waarde van object x.

$$\sum_{x \in I} g(x) \leq c \tag{3.5}$$

$$\sum_{x \in I} w(x) = \max_{J \subseteq O} \sum_{x \in J} w(x) \tag{3.6}$$

Representatie

Dit probleem wordt ook wel het *0-1* knapzak probleem genoemd. 1 als een object in de knapzak zit, 0 als het er niet inzit. Een representatie light in dit geval dus voor de hand: een bitstring met voor elk object één bit.

In tabel 3.1, zijn zes objecten opgenomen, met hun gewicht en waarde. Stel dat de capaciteit van onze knapzak acht is, m.a.w. $c = 8$. Bekijk de individuen in tabel 3.2. Dit zijn individuen die niet meer wegen dan toegelaten. Deze individuen zijn correct. Bekijk vervolgens de individuen in tabel 3.3. Deze individuen wegen te veel. Herinner dat we slechts $c = 8$ capaciteit hebben in de knapzak. De objecten in individuen in 3.3 hebben echter wel een hogere waarde.

Fitheidsfunctie

Als fitheidsfunctie opteren we voor de som van de waarden van de objecten in de knapzak, voorgesteld door een individu. Dit is een standaard fitheidsfunctie (zie pagina 15).

Er doet zich echter een probleem voor als we deze fitheidsfunctie kiezen. De individuen in tabel 3.3 hebben namelijk een zeer hoge fitheid, maar zijn geen correcte oplossingen, want ze wegen te veel. Dit is een fenomeen dat wel vaker voorkomt bij GA's. Dit is oplosbaar door 0 als fitheid toe te kennen aan individuen met $g(x) > c$. Op deze manier hebben foutieve individuen een zeer kleine kans op overleven en voortplanten, en verdwijnen ze automatisch uit de populatie. Dit fenomeen vertraagt het GA, en moet, indien mogelijk, zoveel mogelijk worden vermeden.

a	b	c	d	e	f	Gewicht	Waarde
0	0	1	0	0	0	1	45
1	1	1	0	0	0	7	80
0	1	1	0	0	1	8	85
0	0	0	1	0	0	7	100

Tabel 3.2: Correcte individuen, met gewicht en waarde van het individu

a	b	c	d	e	f	Gewicht	Waarde
0	0	1	0	1	0	10	295
1	1	1	1	0	1	17	210

Tabel 3.3: Foutieve individuen, met gewicht en waarde van het individu

3.1.5 Zoekruimte en Lokale Maxima

In het GA werken we met een representatie van het doelprobleem. Zoals in 2.3 uitgelegd kunnen we aan een representatie een ruimte koppelen. Hoe doorzoekt het GA die ruimte? Er zijn een aantal mogelijkheden. We kunnen werken met *hillclimbing*. Hillclimbing is een techniek waarbij een hypothese wordt omgevormd tot of vervangen wordt door een andere hypothese volgens een heuristiek. De heuristiek is meestal een error-functie, die geminimaliseerd dient te worden. Een veel voorkomend probleem bij algoritmes die *hillclimbing* gebruiken is dat ze uiteindelijk vast komen te zitten op een eilandje in de zoekruimte. De hypotheses op dat eilandje zijn vaak lokaal optimaal, en niet noodzakelijk globaal optimaal. Het probleem van vast geraken in een lokaal optimum wordt grotendeels veroorzaakt door dat we werken met één punt.

Het GA werkt met meerdere punten. We voeren hillclimbing uit door punten te plaatsen in de ruimte waar optima voorkomen. We zoeken de top van de heuvel door punten te combineren en te muteren. Merk op dat het GA op meerdere heuvels tegelijkertijd werkt. Dit volgt uit de schemastelling, zie later. Het is ook duidelijk dat seksuele recombinatie en mutatie punten in de ruimte kunnen wegslingeren van een heuvel en zo een nieuwe heuvel kunnen ontdekken.

Het gevaar dat schuilt in hillclimbing wordt op deze manier grotendeels ontweken. We kunnen echter niet garanderen dat het volledig wordt ontweken. Maar een voldoende grote populatie en een voldoende aantal iteraties zullen in de meeste gevallen tot bevredigende resultaten leiden.

3.1.6 GA en Alternatieven

Waarom gebruiken we het Genetisch Algoritme? We bespreken hoe het GA de zoekruimte onderzoekt, en bekijken wat de alternatieven zijn voor het GA, nl. blind random search en greedy exploitation.

Merk op dat er nog andere Machine Learning technieken bestaan die een heuristiek bieden om in een heel grote zoekruimte een oplossing te vinden, bijvoorbeeld simulated annealing. Deze technieken zijn net als het GA ontwikkeld omdat de twee strategieën die we gaan voorstellen niet praktisch zijn.

We gebruiken meermaals het begrip kost in de volgende discussie. Kost kan betekenen de tijd die we nodig hebben om verder te zoeken in de ruimte. Of het verlies dat we maken door een suboptimale oplossing te gebruiken in de situatie waarvoor het GA een oplossing probeert te vinden.

Laten we eerst definiëren wat het Genetisch Algoritme nu juist is. Bekijk hiervoor definitie 3.1.

Definitie 3.1. *Het Genetisch Algoritme gebruikt strings van lengte L, over een eindig alfabet Σ. Σ bestaat uit K tekens.*

Het GA begint met een initiële random populatie P_0. De kans dat in P_0 een globaal optimum zit is minimaal, want de populatie bevat een zeer klein deel van de volledige zoekruimte. Om P_1 te construeren kunnen we opnieuw een random populatie kiezen. Opnieuw hebben we een zeer kleine kans dat er een globaal optimum in P_1 zit. Voor P_2 kunnen we opnieuw een random populatie kiezen. Deze strategie wordt *blind random search* genoemd. Blind random search gebruikt geen informatie uit de vorige iteratie(s). Stel dat $L = 60$ en $K = 2$. We veronderstellen dus dat het GA binaire strings gebruikt van lengte 60. Dit zou b.v. een oplossing zijn voor het 0-1 knapzak probleem met 60 objecten. Dit betekent dat de zoekruimte van dit probleem 2^{60} mogelijke strings bevat. $2^{60} \approx 10^{18}$. Als we één miljard (10^9) strings kunnen genereren en evalueren per seconde, dan duurt het ongeveer 32 jaar voor we alle mogelijke strings hebben geëvalueerd. Het is duidelijk dat problemen met langere strings en een minstens even groot alfabet, of problemen met even lange strings maar een groter alfabet nog veel meer tijd zullen vragen. Blind random search uitvoeren duurt bij gevolg bijzonder lang. Deze strategie zal uiteindelijk wel de hele zoekruimte *geëxploreerd* hebben.

Een tweede strategie die we kunnen toepassen is adaptatie. Een *adaptief algoritme* gaat informatie uit de vorige populatie(s) gebruiken om op een "intelligente" manier de volgende populatie te creëeren. Een voorbeeld van een adaptief algoritme is het *greedy exploitation* algoritme. We kiezen het beste individu uit P_0 en gebruiken dit als oplossing. Wat is hier het voordeel van? Nieuwe individuen onderzoeken brengt inherent een kost met zich mee. Want we creeëren mogelijk individuen die minder fit zijn dan het fitste individu in P_0. Kiezen voor de beste oplossing die voorhanden is, brengt geen extra kost met zich mee. Vergelijk dit met de enorme kost die blind random search met zich meebrengt. Wat is het nadeel van greedy exploitation? Zoals eerder aangehaald is de kans dat we een globaal optimum aantreffen in P_0 bijzonder klein. Door de oplossing toch uit deze verzameling te kiezen, mislopen we bijna gegarandeerd een veel betere oplossing. We verliezen dus heel veel door een inefficiënte oplossing te gebruiken. Stel dat f_{max} de maximale fitheid is, en f_0 de fitheid van het beste individu in P_0. Als we niet verder exploreren kost ons dat bij elke toepassing van de oplossing: $k = f_{max} - f_0$. k kan bijzonder groot zijn in praktische situaties.

Merk op dat blind random search ook een bijzonder slecht adaptief algoritme is. We onthouden uit vorige populaties welke individuen we reeds gegenereerd hebben. Deze genereren we niet opnieuw in toekomstige populaties.

Merk op dat we twee extreme gevallen van *exploratie* en *exploitatie* hebben beschreven. Wat we nodig hebben is een algoritme dat een tradeoff vindt tussen exploratie en exploitatie. Het GA is een voorbeeld van zo een algoritme.

3.1.7 Schemastellingen

In deze deelsectie bekijken we de bedoeling van en de intuïtie achter schemastellingen. Eerst beschouwen we een aantal definities voor we verdergaan met de bespreking van schemastellingen.

Definitie 3.2. *Een schema is een string over het uitgebreide alfabet* $\Theta = \Sigma \cup \{*\}$, *met* $* \notin \Sigma$. $*$ *is het 'eender wat' symbool.*

Definitie 3.3. *Een individu, string, x voldoet aan een schema* σ *als* $x_i = \sigma_i$ *of* $\sigma_i = *$. *Waarbij* x_i *staat voor het* i^e *karakter in x. Idem voor* σ_i.

Wat is een schemastelling? Een schemastelling wordt gebruikt om de werking van het Genetische Algoritme of Genetisch Programmeren beter te begrijpen. Het baseert zich op het *onderverdelen van de zoekruimte* in kleinere deelzoekruimten. De deelzoekruimten noemen we schema's.

Hoe proberen we het GA beter te begrijpen? Een schemastelling is een *mathematische modellering* van $m(H,t)$. H is een schema, t is een tijdstip en komt overeen met een populatie P_i. $m(H,t)$ is het aantal individuen dat voldoet aan het schema H. M.a.w. een schemastelling modelleert hoe, en waarom, het aantal individuen van generatie tot generatie varieert voor een schema.

In deze deelsectie nemen we aan dat we werken met een string, niet noodzakelijk een binaire string. We gaan er steeds vanuit dat de zoekruimte groot is, zodat een random sample eruit een zeer kleine kans heeft op het bevatten van een globaal optimum.

Definitie 3.4. *Specificiteit is het aantal niet* $*$ *tekens in een schema. Wordt genoteerd als* $O(\sigma)$ *voor* σ *een schema.*

a. Schema's en Hypotheses

Deelsectie 3.1.6 toont aan dat we zijn aanbeland in een typische situatie in de informatica. Er zijn twee parameters, *snelheid* (exploitatie) en *optimaliteit* (exploratie) van de oplossing. Een algoritme dat de ene parameter bevoordeelt, benadeelt de andere, en visa versa. Met het Genetisch Algoritme kunnen we een trade-off zoeken.

In de rest van dit onderdeel gaan we aantonen dat het GA adaptief werkt en tot een meer optimale oplossing kan komen dan greedy exploitation, gegeven genoeg tijd. Merk echter op, zoals eerder aangehaald is het GA random en kunnen we dus geen concrete uitspraken doen over wat genoeg tijd is, noch over de optimaliteit van de oplossing die gevonden wordt. In de praktijk is aangetoond dat GA in staat is om tot suboptimale of zelfs de optimale oplossing te komen voor een probleem. In de theorie is echter nog niet bewezen dat het GA noodzakelijk de optimale oplossing vindt.

Cruciaal voor het GA is dat een populatie meer informatie bevat dan alleen maar over de aanwezige individuen. Een schema met één $*$ heeft K individuen die eraan voldoen. Een schema met twee $*$ heeft K^2 individuen die eraan voldoen. In het algemeen, als een schema σ specificiteit $O(\sigma)$ heeft, en we werken met strings van lengte L, dan zijn er $K^{L-O(\sigma)}$ individuen die voldoen aan σ. Er zijn $\binom{L}{O(\sigma)}K^{O(\sigma)}$ schema's met specificiteit $O(\sigma)$. Elk individu voldoet aan 2^L schema's. Dit laatste is makkelijk in te zien als we binair optellen van 0 tot $2^L - 1$. Dit kan door L bits te gebruiken. Voor elke getal tussen 0 en $2^L - 1$ vervangen we 0 door $*$ en 1 door het karakter dat op de overeenkomstige plaats in het individu staat.

Schema's zijn hypotheses over het probleem. Waarom is een schema een hypothese? Een schema specificeert een aantal tekens, en stelt hierdoor impliciet dat de fitheid alleen afhankelijk is van de gespecificeerde tekens. De niet-gespecificeerde tekens zijn immers "don't care" symbolen en kunnen om het even wat zijn. Een schema met veel fitte individuen is waarschijnlijk een goede hypothese. Verder in de tekst geven we een meer formele betekenis aan schema' met veel fitte individuen.

Laten we eerst eens kijken naar een voorbeeld van een GA, met bitstrings van lengte 3. Dit GA lost het knapzak probleem op, zie 3.1.4.

Voorbeeld 3: We beschouwen 3 objecten: A, B, en C. Het gewicht en de waarde van elk van deze objecten is niet relevant in dit voorbeeld.

In een bepaalde iteratie i zit in de populatie een string van de vorm 101. Deze string heeft een fitheid van $\mathit{fitness}(101)$. In tabel 3.4 zijn alle schema's opgesomd waarin 101 voldoet. Elk schema stelt een hypothese voor. De hypothese die hoort bij een schema wordt ook in tabel 3.4 weergegeven.

In iteratie i ziet de populatie er uit zoals weergegeven in tabel 3.5. Schema $11*$ wordt tweemaal gerepresenteerd, nl. door 110 en 111. De gemiddelde fitheid, zie verder, voor schema $11*$ is dus $\frac{4+6}{2} = 5$. Het schema $*01$ wordt alleen door 001 gerepresenteerd. De gemiddelde fitheid van $*01$ is 3. Uit deze populatie kunnen we dus afleiden dat de hypothese "A en B zijn het belangrijkste" waarschijnlijker is dan de hypothese "C is het belangrijkste en B hoort niet thuis in de knapzak". Merk op dat een groot aantal schema's niet gerepresenteerd is.

Het schema $***$ geeft de gemiddelde schemafitheid weer in de huidige populatie. In dit geval is deze: $\mathit{fitness}(***) = \frac{4+5+1+3+6}{5} = 3.8$

□

Schema	**Hypothese: Waarom is de oplossing goed? Omdat ...**
$***$	Het maakt niet uit wat we kiezen, alles is een goede oplossing
$1**$	A in de knapzak zit
$*0*$	B niet in de knapzak zit
$**1$	C in de knapzak zit
$10*$	A er in en B er niet in zit
$1*1$	A en C in de knapzak zitten
$*01$	B er niet in en C er wel in zit
101	A en C er in en B er niet in zit

Tabel 3.4: De schema's waaraan 101 voldoet, inclusief de hypothese

Individu	**Fitheid**
110	4
100	5
010	1
001	3
111	6

Tabel 3.5: Populatie in een iteratie, met fitheid

b. Wat zijn goede schema's?

Hoe komen we nu te weten welke hypothese correct aangeeft waarom een individu beter presteert dan gemiddeld? We zouden kunnen proberen om alle schema's, waartoe een individu met boven gemiddelde prestatie behoort, op te sommen. Deze lijst nemen we aan als de mogelijke verklaringen voor de boven gemiddelde prestatie. Maar hoe lager de specificiteit van een schema, hoe meer individuen er voldoen aan dat schema. Dit leidt tot schema's waaraan zowel fitte als zwakke individuen voldoen. Zulke schema's zijn dus geen goede hypothese voor boven gemiddeld gedrag. Kijk bijvoorbeeld naar de individuen 100 en 001, in tabel 3.5. Beide voldoen aan het schema $*0*$. Maar 100 presteert boven het gemiddelde, terwijl 001 onder het gemiddelde presteert. Het afwezig zijn van B is dus geen goede hypothese om te bepalen of een oplossing goed, dan wel slecht presteert.

Definitie 3.5. *De gemiddelde schemafitheid fitness(σ) voor een schema σ, is gedefiniëerd als zijnde de gemiddelde fitheid van alle individuen die voldoen aan σ, in de populatie van de huidige iteratie.*

De gemiddelde schemafitheid is een statistische variabele, een schatting met een afwijking. Naarmate er meer individuen aan een schema voldoen daalt de afwijking. Merk op dat er expliciet geen enkele berekening wordt gedaan door het GA op de schema's. Alle informatie over de schema's is vervat in de populatie, in de vorm van de individuen en de fitheid van de individuen. Intuïtief is het duidelijk

dat we nieuwe individuen van veelbelovende schema's, met een hoge gemiddelde schemafitheid, willen uittesten.

Het Genetisch Algoritme voorziet ons van een manier om nieuwe, mogelijk meer optimale punten te onderzoeken. Het GA drijft de zoektocht naar een oplossing namelijk in de richting van veelbelovende oplossingen. Maar terwijl we een nieuwe populatie samenstellen met nieuwe, nog niet onderzochte punten, moeten we ook onthouden waarom we deze richting zijn in gegaan. Het is mogelijk dat de nieuwe punten toch niet zo een goede gok bleken te zijn. Wat doen we dan als we de vorige populatie kwijt zijn? De nieuwe generatie moet dus ook bewijs bevatten dat de richting die we insloegen de juiste lijkt te zijn, gegeven de data in de vorige generatie. Langs de andere kant moeten we ook rekening houden met het feit dat we slechts beschikken over onvolledige informatie. Volgende generaties kunnen mogelijk aantonen dat we de foute richting hebben gekozen, en dat we beter een andere richting zouden uitgaan, of goed zaten.

Samengevat, we willen nieuwe individuen creëren om de zoekruimte verder te exploreren in de richting die lijkt te leiden naar betere oplossingen. Tegelijkertijd willen we de reeds gevonden oplossingen niet uit het oog verliezen, want het is mogelijk dat we in de foute richting exploreren. We kiezen dus voor een gedeeltelijke exploitatie van de gekende oplossingen.

c. Casino

Holland beschrijft in zijn boek “Adaptation In Natural and Artificial Systems” hoe we het GA kunnen vergelijken met het *multi-armed-bandit* probleem [11]. Genoemd naar de beroemde gokmachines in casino's met één arm, ook wel slotmachines. Het multi-armed-bandit probleem is een veralgemening van het *two-armed-bandit* probleem. Dit probleem is gerelateerd aan de keuze tussen *exploitatie* van bekende oplossingen en verdere *exploratie* van de zoekruimte, en biedt een intuïtieve verklaring voor de werking van het GA a.h.v. gemiddelde schemafitheid. We kunnen de armen van de bandit vergelijken met de schema's in GA. Voor een exactere uitleg verwijzen we naar het boek van Holland [11].

Beschouw een slotmachine met 2 armen. Als we weten welke arm van de twee, de beste payoff geeft, dan is de beste strategie voor de hand liggend. We trekken altijd aan die arm. Maar als we niets weten over de payoff van de respectievelijke armen, moeten we testen. We trekken x keer aan arm 1 en een x aantal keer aan arm 2. Dit levert ons volgende statistische variabelen op: p_1, p_2, σ_1^2 en σ_2^2. Namelijk de schattingen van de respectievelijke gemiddeldes van arm 1 en arm 2, en de schattingen van de respectievelijke varianties op de armen.

Kiezen we nu exploitatie van de betere arm, noem deze arm 1, of verdere exploratie van de zoekruimte? Exploratie komt overeen met toch nog aan arm 2 te trekken om te checken of arm 1 effectief de betere arm is. Deze keuze zijn we reeds eerder tegengekomen bij de bespreking van blind random search en greedy exploitation, zie 3.1.6. Wat is de *kost van exploitatie*? Het verschil tussen de maximale payoff p_{max} en p_1. Wat is de *kost van exploratie*? Het verschil tussen p_1, het gemiddelde van de betere arm, en p_2, het gemiddelde van de mindere arm.

p_{max} is mogelijk veel groter dan p_1. Alleen exploiteren zou ons dus veel kosten. Merk op dat door dom toeval p_1 wel dicht in de buurt van p_{max} kan liggen, in dat geval kost het ons veel om te exploreren, maar we zijn ons hiervan niet bewust. We hebben dus een strategie nodig die de betere arm exploiteert, en tegelijk verder exploreert. Zodoende wordt de kost geminimaliseerd.

Holland toont aan dat als we het aantal testen vasthouden, net als de populatie in GA, en we een exponentieel aantal testen reserveren voor de betere arm dit leidt tot een optimale strategie om aan de armen te trekken. Dit resultaat kan veralgemeend worden naar meerdere armen.

Dit resultaat is puur intuïtief en verklaart theoretisch gezien nog niet waarom het GA werkt.

3.1.8 De Schemastelling van Holland

In de vorige deelsectie hebben we bekeken wat een schemastelling is. Een schemastelling is een mathematische modellering van het aantal individuen volgens schema's. Schema's zijn deelruimten van de zoekruimte. Holland stelt een schemastelling voor die een ondergrens stelt aan het verwachte aantal individuen dat aan een schema σ voldoen in populatie $i+1$, gegeven het aantal individuen in i.

Stelling 3.1 (Schemastelling)**.** *Zij $fitness(\sigma, i)$ de gemiddelde schemafitheid voor een schema σ in iteratie i, gedefiniëerd als 3.7. Met $m(\sigma, i)$ het aantal individuen dat voldoet aan de schema σ, en $fitness(x_j, i)$ de fitheid van individu x_j in P_i. Het Genetisch Algoritme zal in P_{i+1} een verwacht aantal individuen $m(\sigma, i+1)$ crëeren met $m(\sigma, i+1) \geq \frac{fitness(\sigma,i)}{\overline{fitness}(i)} m(\sigma, i)(1-\epsilon_c)(1-\epsilon_m)$*
Met ϵ_c de kans op crossover, en ϵ_m de kans op mutatie. $\overline{fitness}(i)$ is de gemiddelde fitheid van de individuen in P_i.

$$fitness(\sigma, i) = \frac{\sum_{x_j \in \sigma} fitness(x_j, i)}{m(\sigma, i)} \tag{3.7}$$

Uit de formule van deze schemastelling volgt dat er wordt verwacht dat schema's met een bovengemiddelde fitheid, m.a.w. veel fitte individuen, door nog meer individuen zullen worden gerepresenteerd in de volgende populatie. Merk op dat schema's met ondergemiddelde fitheid niet meteen uitsterven, maar wel verdwijnen uit de populatie, die komt overeen met exploratie.

We hebben de parameters ϵ_c en ϵ_m ingevoerd. Deze geven respectievelijk de kans op crossover en mutatie. Waarom zijn deze parameters opgenomen in de formule? Als we crossover of mutatie toepassen, selecteren we een willekeurig individu x uit de populatie. Dit individu x voldoet aan meerdere schema's. De kans bestaat dat x nadat er crossover of mutatie op is uitgevoerd niet meer aan dezelfde schema's voldoet. Deze kans is groter voor schema's met een hoge specificiteit. Met andere woorden crossover en mutatie verstoren schema's. We nemen ook

aan dat als een schema verstoord kan worden, door een individu dat niet langer aan het schema voldoet door het uitvoeren van crossover of mutatie, dat dit ook effectief gebeurt. Dit noemt men een pessimistische schemastelling (zie sectie GP schema's). De gegeven formule bepaalt dus ook slechts een ondergrens voor het aantal individuen. De formule houdt bijvoorbeeld geen rekening met het omgekeerde effect, individuen die door het uitvoeren van crossover of mutatie ineens wel aan het schema voldoen.

$(1-\epsilon_c)(1-\epsilon_m)$ is de kans dat we noch kiezen om crossover uit te voeren, noch kiezen om mutatie uit te voeren. Dit leidt tot het gedeelte individuen dat niet overgeleverd is aan crossover en mutatie. Dit zijn dus de individuen waarvan we zeker zijn dat ze aan dezelfde schema's blijven voldoen.

3.1.9 Nog meer Schemastellingen

Dit is een korte samenvatting van de hoofdstukken 3 t.e.m. 5 uit het boek "Foundations of Genetic Programming" van Langdon en Poli [16].

De schemastelling van Holland werd eerst beschouwd als een tautologie. Recenter gaat men er echter vanuit dat de overinterpretatie van de stelling aan de basis ligt van die tautologie [23]. Er zijn echter nog vele andere schemastellingen. Zowel pessimistische, die een ondergrens geven, als exacte, die het exacte verwachte aantal individuen in de volgende populatie berekenen.

We werken altijd met verwachte aantallen, omdat het GA random blijft, en we dus nooit exact kunnen voorspellen hoeveel individuen er van een bepaald schema in een bepaalde populatie gaan voorkomen.

Voor de schemastelling van Holland, was er de stelling van Price [22]. George R. Price, een populatiegeneticus, heeft een stelling opgesteld over selectie en covariantie. Deze stelling handelt over de frequentie van een gen van generatie i naar generatie $i+1$. Simpel gesteld verhoogt de frequentie van een gen als een gen een hoge covariantie heeft met individuen met een hoge fitheid. M.a.w. de frequentie van een gen dat in veel fitte individuen voorkomt verhoogt het meeste. Er is aangetoond dat de schemastelling van Holland volgt uit de stelling van Price.

Er bestaan echter nog andere schemastelling, bijvoorbeeld de schemastelling van Stephens en Waelbroeck [27; 28]. Zij beschreven een schemastelling die de exacte verwachte waarde voor het aantal individuen berekent. Hier gaan we echter niet dieper op in.

We merken op dat de schemastellingen een formule zijn die kan uitgewerkt worden, maar geen voorspellingen doet. Het uitrekenen van zo'n formule is identiek hetzelfde als het uitvoeren van een Genetisch Algoritme.

3.2 Genetisch Programmeren

> A computer program is merely a mathematical transformation (i.e., function) that maps certain inputs into certain outputs.
>
> How can computers learn to solve problems without being explicitly programmed? In other words, how can computers be made to do what is needed to be done, without being told how to do it?
> — *Arthur Samuel*

GP is gebaseerd op GA. In deze sectie beschrijven we hoe het Genetisch Programmeren (GP) verschilt van het Genetisch Algoritme, 3.2.2. Indien anders vermeld, geldt wat gezegd is over GA ook voor GP.

We bekijken waar GP vandaan komt in 3.2.1, en welke genetische operaties we gaan gebruiken in dit experiment, in 3.2.6. Toepassingen van GP werken met bomen, zoals uitgelegd in 3.2.2. We specifiëren de terminals en functies die gebruikt worden in dit experiment in 3.2.4. In 3.2.7 bekijken we nog hoe we de fitheid van programma's bepalen. We sluiten deze sectie af met de schemastellingen voor Genetisch Programmeren.

3.2.1 Geschiedenis

Genetisch Programmeren is ontstaan door de tekortkomingen van het Genetisch Algoritme. De eerste experimenten zijn echter uitgevoerd in de jaren 80 van vorige eeuw door Stephen F. Smith (1980) en Nichael L. Cramer (1985) [26; 3]. Koza wordt echter beschouwd als de *vader van Genetisch Programmeren* [15].

Er is ondertussen een serie boeken verschenen, waarin steeds meer geavanceerde technieken worden besproken. Koza beschikt over een cluster, die ook wel "invention machine" wordt genoemd, waarop hij zijn experimenten runt. Uit deze experimenten zijn 36 uitvindingen gekomen die competitief zijn met menselijke uitvindingen. Twee van deze uitvindingen zijn gepatenteerd.

3.2.2 Representatie probleem

GA's gebruiken strings van een vaste lengte om een probleem voor te stellen. Dit legt echter een zware beperking op onze representatie, en bijgevolg op de oplossingen die we kunnen vinden. Problemen die een dynamische representatie nodig hebben kunnen we niet oplossen m.b.v. een GA. Er zijn echter een heel aantal interessante problemen die een dynamische representatie vergen.

Genetisch Programmeren is een uitbreiding van GA's in de zin dat we programma's evolueren in plaats van strings. Programma's hebben als voordeel dat ze de oplossing voor veel problemen kunnen beschrijven. In tekstvorm zijn programma's echter moeilijk bruikbaar, daarom gebruiken we in GP de meer gebruikelijke vorm van programma's in computers, namelijk de boomvoorstelling. Bomen zijn flexibeler wat betreft genetische operaties dan tekst. We kunnen altijd een tak of een

deelboom ergens in de boom toevoegen. Maar zoals we verder zullen opmerken kunnen de genetische operaties ook de semantiek en/of syntax van een programma aantasten. Dit zou leiden tot individuen met een extreem lage fitheid, die de populatie laten dichtslibben. Het komt er dus op aan om genetische operaties zo te definiëren dat ze zowel de syntax en semantiek behouden, alsook effectief genoeg zijn om de zoekruimte te exploreren.

3.2.3 Genotype en Fenotype

Biologisch gesproken is er een verschil tussen het genotype en het fenotype van een individu. "Het genotype van een individu is zijn genoom, en draagt bij tot de vorming van karakteristieken"[2]. "Het fenotype van een individu is ofwel zijn fysieke voorkomen en constitutie of een specifieke manifestatie van een kenmerk, zoals grootte, oogkleur, ... De interactie tussen genotype en fenotype kan als volgt beschouwd worden: genotype + omgevingsfactoren → fenotype"[3].

In dit document maken we geen onderscheid tussen genotype en fenotype. Een programmaboom (of expressie in ons geval) is het "DNA" van een individu, en is tegelijkertijd het volledige fenotype, want het is onmogelijk dat delen van een expressie nu eens wel en dan eens niet worden gebruikt. De hele expressieboom wordt gebruikt om de expressie te evalueren. Deze simplificatie laat ons toe om makkelijker en sneller te "programmeren".

Als we onze bomen echter hadden geëncodeerd in XML, dan zou het wel mogelijk zijn om meer informatie in het XML bestand te steken, en deze afhankelijk van de omstandigheden wel of niet op te nemen in de datastructuur (de boom) die ons individu voorstelt in het hoofdgeheugen van de computer.

Merk op dat we in het begin van dit hoofdstuk al stelden dat GA en GP ruwweg zijn gebaseerd op biologische principes. We kunnen deze principes beter proberen te simuleren met minder simplificaties. In de bibliografie zijn enkele werken opgenomen die gebruik maken van genotype-fenotype [14; 6]. Merk op dat deze werken dichter bij de biologie staan dan ons gebruik van GP. Ze zijn echter nog mijlenver verwijderd van de reële biologische wereld.

3.2.4 Terminals en Functies

In GP moeten we ook een verzameling van terminals definiëren. Dit is de verzameling van waarden die voor kan komen in de bladeren van de boom. We noemen deze verzameling T.

Tevens is het noodzakelijk om een verzameling van functies of operators te definiëren. Dit is de verzameling van waarden die in een inwendige knoop kunnen voorkomen. We noemen deze verzameling F.

[2]Zie: http://en.wikipedia.org/wiki/Genotype
[3]Zie: http://en.wikipedia.org/wiki/Phenotype

Sluiting

Definitie 3.6. *Zij DT de verzameling van datatypes. Formeel:* $DT = \{x | x \in T \vee \exists f \in F : x \text{ is het return type van } f\}$.

Elke functie in F moet in staat zijn om eender welk datatype uit DT aan te nemen als zijn parameters. Dit is nodig omdat we door recombinatie en mutatie willekeurige deelbomen en knopen kunnen plaatsen onder een functie. Als een functie zou falen door een bepaald datatype, komen we weer in de situatie waar we zitten met individuen die "niet werken". Deze ongeldige uitdrukkingen doen alleen maar de populatie dichtslibben.

Adequatie

Zoals eerder aangehaald is het belangrijk dat de representatie genoeg kracht bezit om de oplossing uit te drukken. Dit betekent dat we F uitgebreid genoeg moeten maken zodat de programma's die worden gegenereerd voldoende expressiekracht bezitten om de nodige bewerkingen uit te drukken in functie van F. Verzameling T moet genoeg variabelen bevatten zodat we het probleem kunnen oplossen.

We weten alleen na het runnen van GP of F en T adequaat waren om het probleem op te lossen. Als de oplossing van GP slechtst slaagt in het oplossen van een klein percentage testcases, is dit een symptoom van de inadequaatheid van F en T. Merk op dat door het random karakter van GP F en T niet noodzakelijk inadequaat zijn als er geen goede oplossing wordt gevonden.

Relationele Algebra

In de relationele algebra hebben we maar 1 datatype dat als input voor een functie kan gebruikt worden, namelijk de relatie. In dit document gaan we er tevens vanuit dat de relatie een tabel van integer waarden is. De sluiting lijkt bijgevolg voldaan, doch later zullen we aantonen dat de relationele algebra niet geschikt is voor GP in zijn standaard vorm. Merk op dat het ook mogelijk is om de genetische operaties zo aan te passen dat we fouten in de sluiting vermijden. Dit vergt echter meer rekenkracht.

3.2.5 Initiële populatie

Voor we kunnen beginnen met genetisch programmeren hebben we een initiële populatie nodig. We beschikken over een verzameling terminals T en een verzameling functies F. Noem nu $C = T \cup F$. In de initiële populatie zitten bomen waarvan de knopen bestaan uit elementen van F en bladeren bestaan uit elementen van T.

Hoe worden de bomen geconstrueerd voor de initiële populatie? We bespreken vier technieken om een willekeurige boom te construeren. Elke techniek heeft zo zijn voor- en nadelen. We definiëren hier echter geen kansfunctie over bomen, en willekeurig is bijgevolg niet helemaal correct in deze context.

Free Met een zekere kans kiezen we T of F. Meestal is de kans kleiner dat we T kiezen. Kies willekeurig een element x uit de gekozen verzameling. Als $x \in T$ dan zitten we reeds in een blad en moeten niets verder construeren. Als $x \in F$, dan heeft x een ariteit. Werk recursief verder.

Deze techniek kan echter leiden tot zeer grote of zeer kleine 1-knoops bomen. Vooral deze laatste zijn zeer simplistische oplossingen en bijgevolg meestal foutief. Deze techniek geeft echter ook bomen die meer natuurlijk ogen.

Full Elke tak van een geconstrueerde boom heeft een bepaalde diepte. Stel dat alle takken n diep moeten zijn. De eerste $n - 1$ willekeurige keuzes die we maken, kiezen we uit F. De laatste, n^{de} keuze maken we uit T.

Dit leidt tot bomen met takken die allemaal exact even lang zijn. Veel praktische queries hebben echter takken met sterk verschillende diepte. Deze bomen ogen meer geknutseld.

Grow In deze techniek stellen we een maximum diepte n op. De lengte van alle takken is gelijk aan of minder dan n. Op deze manier hebben we een zeker mate van variatie in P_0.

De vraag die rijst is, wat is een goede waarde voor n? Een te kleine waarde voor n zal leiden tot te simplistische oplossingen, een te hoge waarde tot te complexe oplossingen. Hier komt recombinatie op naar voren. Recombinatie kan bomen snoeien en doen groeien. Een te kleine of te grote waarde leidt dus tot langere uitvoertijden, maar in het algemeen zal GP dit obstakel kunnen overwinnen, gegeven genoeg tijd.

Ramped half-and-half Deze techniek combineert de vorige twee technieken. Er wordt een n gekozen. 50% van de bomen wordt gecreëerd d.m.v. de Full techniek. Er wordt een kleine wijziging doorgevoerd. Er wordt een gelijk aantal bomen geconstrueerd met diepte 2 tot n. Dit betekent dat er na de Full methode evenveel bomen met diepte 2, 3, 4, ... $n - 1$, n zijn.

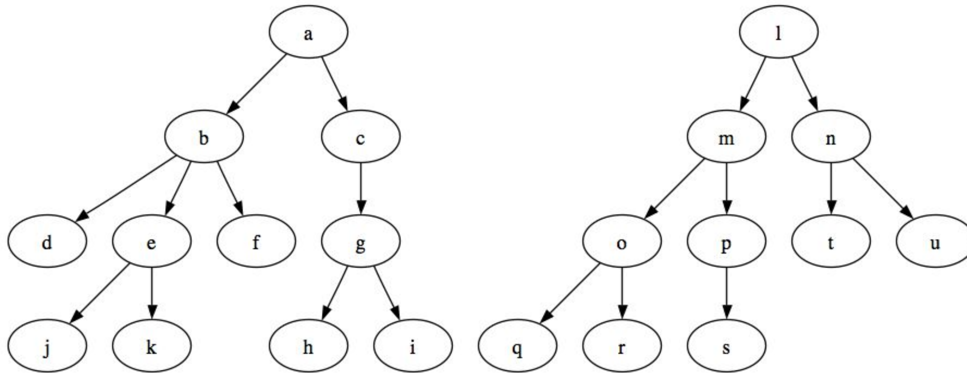
De overie 50% van P_0 wordt geconstrueerd d.m.v. de Grow techniek.

Dit levert een gevarieerde verzameling bomen op waarin elke diepte aanwezig is (als de populatiegrootte minstens $2 \times n$ is). Ook hier rijst de vraag wat een goede waarde voor n is. We kunnen proberen te schatten wat de complexiteit is van de te zoeken query, en n dan een stuk groter kiezen. Zo heeft GP de nodige speelruimte.

Deze bomen ogen echter nog steeds zeer geknutseld.

3.2.6 Genetische operaties

In deze deelsectie bespreken we welke operaties we gebruiken, en hoe deze toepasbaar zijn op bomen. In de bespreking wordt ook aangegeven wat de knelpunten en voordelen zijn als we werken met programma's.



Figuur 3.2: De ouderbomen: links ouder 1, rechts ouder 2

We veronderstellen dat populatie P_{i-1} gegeven is en we populatie P_i construeren.

Reproductie

Dit is veruit de gemakkelijkste operatie. Een individu wordt geselecteerd uit P_{i-1} en in P_i geplaatst. De parameter *reproductiegraad* bepaalt hoeveel maal deze operatie wordt toegepast.

Deze operatie zorgt dat een aantal, voornamelijk fitte, individuen overleven. Reproductie zorgt voor een soort geheugen over de vorige populaties voor de volgende populatie.

Recombinatie of Crossover

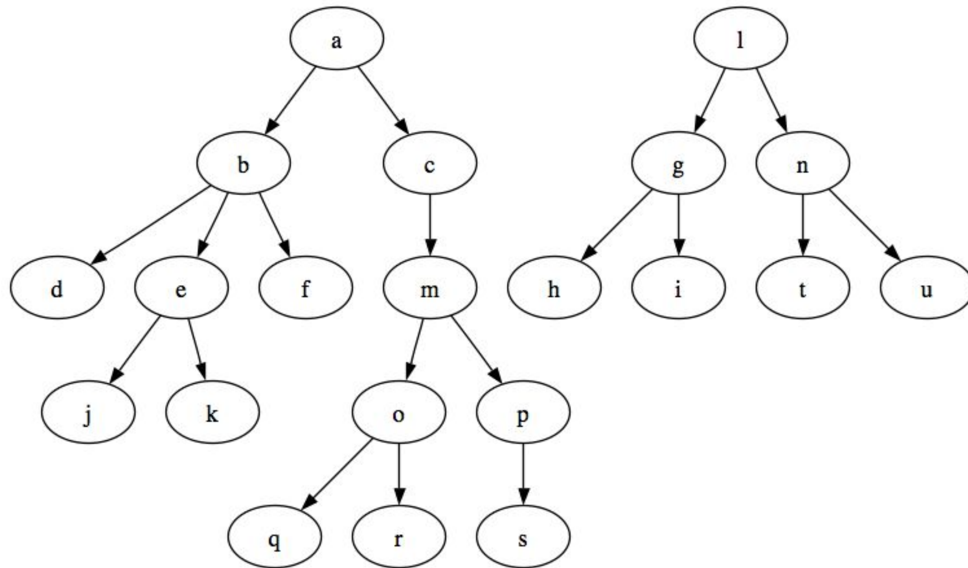
Twee "ouders" worden geselecteerd uit P_{i-1}. In elke ouder wordt een willekeurig crossoverpunt geselecteerd. Noem deze punten p_1 en p_2. Onder elk van deze punten hangt een deelboom (mogelijk bestaande uit slechts één knoop). We maken nu twee "kinderen", door de ouders te nemen en de deelbomen onder p_1 en p_2 van boom te veranderen. De deelboom onder p_1 hangt nu in de tweede ouder, en de deelboom onder p_2 hangt nu in de eerste ouder. De kinderen worden aan P_i toegevoegd.

Voorbeeld 4: Beschouw de ouderbomen ouder 1 en ouder 2 in figuur 3.2. Stel dat in ouder 1 "g" wordt gekozen als crossoverpunt, en in ouder 2 "m" wordt gekozen als crossoverpunt. De kinderen die hieruit voortkomen zijn weergegeven in figuur 3.3.

□

Hoe frequent we recombinatie toepassen wordt bepaald door de *recombinatiegraad* of *crossovergraad*.

Recombinatie of crossover brengt creativiteit in de populatie. Een bepaalde deelboom is misschien bijzonder goed in het oplossen van een deelprobleem, maar



Figuur 3.3: De kinderen verkregen door recombinatie

de rest van de boom verkwanselt dit door niet goed om te gaan met dit deelresultaat. Recombinatie kan deze deelboom dan verplaatsen naar een betere boom. In deze nieuwe boom komt de deelboom mogelijk meer tot zijn recht. Op deze manier hebben we een beter individu gecreëerd.

Hou echter in het achterhoofd hoe vaak we misschien en mogelijk hebben gebruikt in de voorgaande paragraaf. Er kan veel gebeuren door recombinatie. GP is echter random, dit betekent dat niet alles wat kan gebeuren ook noodzakelijk gebeurt. En wat recombinatie uitvoert is niet noodzakelijk goed, juist omdat het random werkt.

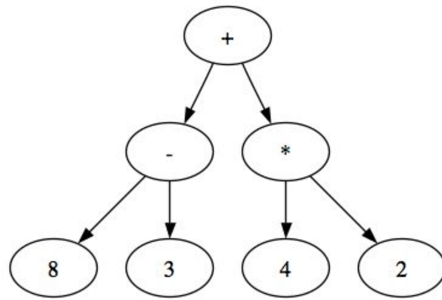
Mutatie

We selecteren een boom uit P_{i-1}. We kiezen willekeurig een knoop uit de boom. We veranderen de knoop in een andere willekeurig gekozen knoop. Het is echter niet altijd mogelijk om een inwendige knoop te veranderen. Een inwendige knoop komt overeen met een functie, niet elke functie heeft noodzakelijk evenveel parameters (dezelfde ariteit). Dit kunnen we oplossen door willekeurig een functie te kiezen met dezelfde ariteit.

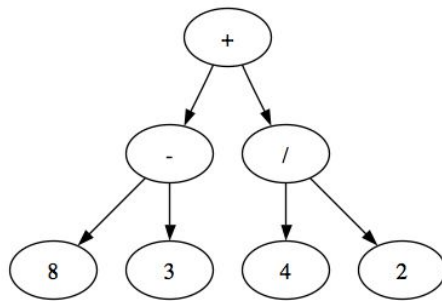
Mutatie is het makkelijkst toepasbaar op bladeren, aangezien deze terminals bevatten. Deze kunnen ook niet zomaar veranderd worden in eender wat, zonder dat de semantiek van de boom verloren gaat, maar hebben wel meer mogelijkheden. In een Java programma kunnen ook we niet zomaar een string in de plaats van een double meegeven aan een functie. Hierover meer in 3.2.4.

Het resultaat wordt toegevoegd aan P_i. Hoeveel we muteren hangt af van de *mutatiegraad*.

Mutatie zorgt voor meer diversiteit in de populatie. Het effect is echter kleiner



Figuur 3.4: De ouderboom



Figuur 3.5: De gemuteerde ouderboom

dan dat van recombinatie.

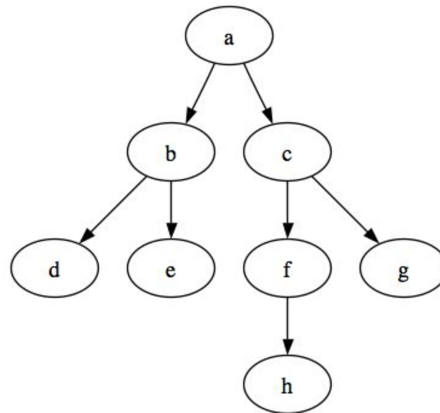
Voorbeeld 5: In figuur 3.4 zien we de ouderboom. In de ouderboom wordt at random een knoop geselecteerd, bijvoorbeeld de vermenigvuldiging $*$. We kiezen at random een interne knoop met dezelfde ariteit, bijvoorbeeld de deling /. We vervangen de $*$-knoop door de /-knoop en de mutatie is gebeurd, zie figuur 3.5.

□

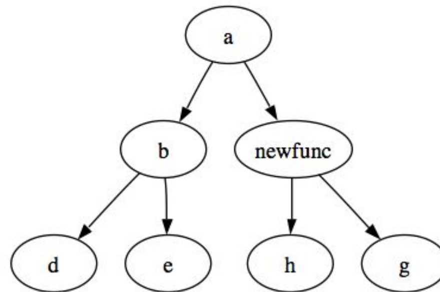
Encapsulatie

We selecteren een boom uit populatie P_{i-1}. In deze boom selecteren we willekeurig een inwendige knoop x. Onder x hangt een deelboom D. We verwijderen alle inwendige knopen van D, en vervangen deze door één nieuwe knoop met als label een *nieuwe functienaam*. Vervolgens hangen we de bladeren onder de nieuwe knoop. De functienaam kan nu gebruikt worden in andere bomen. De nieuwe functie wordt toegevoegd aan F (de functie verzameling), zodat deze kan worden geëvalueerd in de volgende iteraties. Hoeveel bomen we encapsuleren hangt af van de *encapsulatiegraad*.

Op deze manier isoleren we een nuttig stuk "code" uit de boom. Dat stuk code kan vanaf nu niet meer gewijzigd worden. Dit heeft als voordeel dat het sneller kan overgedragen worden naar andere bomen, en de functie niet kan kapot gemaakt worden door mutatie of recombinatie.



Figuur 3.6: De boom voor dat encapsulatie is toegepast



Figuur 3.7: De boom bekomen door encapsulatie toe te passen

Voorbeeld 6: In figuur 3.6 zien we een boom. Stel dat "c" at random wordt gekozen. We verwijderen de inwendige knopen onder "c", dat is alleen "f". In de boom plaatsen we een nieuwe knoop, "newfunc", in de plaats van "c". De bladeren "h" en "g" worden onder de nieuwe knoop gehangen. Het resultaat is te zien in figuur 3.7.

□

3.2.7 Fitheidsfunctie en Testcases

Het laatste onderdeel dat we nog moeten bespreken is de fitheidsfunctie. Zoals aangehaald in het begin van de sectie over Genetisch Programmeren is een programma niets anders dan een functie die een input omzet in een output. We bespreken nu hoe dit inzicht van pas komt in het maken van een fitheidsfunctie.

In de Theoretische Informatica is aangetoond dat er geen algoritme bestaat dat gegeven een formele beschrijving van de oplossing voor een probleem en een programma, beslist of het programma effectief het probleem oplost. Het correct zijn van programma's is echter een belangrijk aspect in de praktijk. Er zijn ver-

schillende technieken ontwikkeld om programma's op hun correctheid te testen en na te gaan of ze stoppen. We denken hier bijvoorbeeld aan technieken zoals formele correctheidsbewijzen met invariante relaties en unit testing, hoewel deze laatste techniek correctheid niet garandeert. Dat een programma stopt is in dit experiment niet van belang aangezien we niet beschikken over recursie noch over iteratie in onze programmeertaal, zie hoofdstuk 4.

In alle GP toepassingen worden testcases gebruikt als basis voor de fitheidsfunctie. Testcases bestaan uit *(input, output)* paren, waarbij *input* de input voor het probleem is, en *output* de oplossing van het probleem voor de gegeven input. De testcases moeten *algemeen* genoeg zijn, om overfitting tegen te gaan. De testcases moeten *alle aspecten* van een probleem belichten, als dit niet het geval is worden individuen niet beloond om het volledige probleem op te lossen.

Laten we een voorbeeld bekijken.

Voorbeeld 7: In dit voorbeeld bekijken we hoe we testcases kunnen opstellen om een wiskundige expressie te induceren. Stel dat we de functie $f(x) = x^3$ willen induceren. Laten we er echter vanuitgaan dat we niet weten dat deze functie bestaat en we GP deze functie willen laten construeren.

Eerst bepalen we T en F. $T = \mathbb{R}$, $F = \{+, -, *, /\}$.

In tabel 3.6 worden een aantal testcases opgesomd. Merk op dat we niet alle mogelijkheden kunnen opsommen. We gebruiken inputwaarden die niet allemaal in mekaars directe omgeving liggen om een beter resultaat te bereiken. Toch zijn er voor deze testcases heel veel mogelijke oplossingen. GP is niet verplicht om een derdegraadfunctie te vinden. GP kan even goed een vierdegraadfunctie vinden die even zeer door deze punten gaat, of een vijfdegraadfunctie, ... Merk wel op dat we, volgens het principe van *Occam's razor* de eenvoudigste oplossingen prefereren, en dus doelen op het vinden van de derdegraadfunctie.

We moeten nog een fitheidsfunctie definiëren. We kiezen voor een error functie als fitheid, wat een redelijk normale keuze is als we werken met functies die getallen produceren. Er bestaan echter veel verschillende soorten error functies. We kunnen de absolute afwijking berekenen, of de kwadratische afwijking, om slechtst enkele te noemen. Merk op dat als een individu 100 hoger of lager uitkomt dan $f(2)$, de oplossing relatief meer fout is dan een verschil van 100 tussen de uitkomst en $f(50)$. Dit is echter makkelijk te omzeilen door te volgende formulering: $\frac{e(x)}{f(x)}$ met e een wiskundige expressie. Wat een relatieve fout geeft.

□

In deze tekst maken we gebruik van inputtabellen I_j en een resultaatstabel X. De fitheid van een expressie wordt gemeten a.h.v. het *symmetrisch verschil*. Het symmetrisch verschil van twee tabellen R en S is $(R - S) \cup (S - R)$. Oftewel de verzameling van tupels die in R voorkomen en niet in S en de tupels die in S voorkomen maar niet in R. We tellen het aantal tupels in het symmetrisch verschil, en normaliseren dit over de lengte van de resultaatstabel X.

De fitheid van een individu e berekenen we als: $\mathit{fitheid}(e) = \sum_{x \in testcase} \frac{e(x)}{f(x)}$.

Input	Output
1	1
2	8
3	27
4	64
10	1000
14	2744
50	125000

Tabel 3.6: Testcases voor x^3 te induceren

3.2.8 Schemastelling

In de vorige sectie hebben we gekeken naar de schemastellingen voor het GA. In deze deelsectie bekijken we schemastellingen voor GP. Er zijn een aantal verschillende mogelijkheden om een schema te definiëren.

Koza, Altenberg, O'Reilly en Whigham definiëren een schema als een component van een boom. Een schema moet niet noodzakelijk in de wortel van de boom beginnen. Dit geeft als complicatie dat één boom op meerdere plaatsen kan voldoen aan hetzelfde schema. De schemastelling van de voorgenoemde worden dan ook *Programma Component Schema* stellingen genoemd. Het nadeel van deze aanpak is dat een schema geen deelruimte voorstelt van de zoekruimte. We besteden verder geen aandacht aan deze stellingen.

Een volgende stap in de ontwikkeling van de theorie achter GP waren de *pessimistische schemastellingen*. Rosca definiëert schema's als bomen met het "don't care" symbool $*$ [24]. $*$ stelt een arbitraire boom voor. Er bestaan oneindig veel van deze schema's, en elk schema heeft oneindig veel individuen die eraan voldoen.

Een volgende stap werd gezet door Langdon en Poli [16]. Zij definiëren "fixed-size-and-shape" schema's. Hierbij wordt het # symbool gebruikt om een willekeurige knoop van een bepaalde ariteit aan te duiden. De ariteit wordt afgeleid uit het schema. Wederom zijn er een oneindig aantal schema's, maar er zijn slechts een eindig aantal individuen die voldoen aan een "fixed-size-and-shape" schema.

De schemastellingen van Rosca en Langdon en Poli zijn pessimistisch. Ze bepalen een ondergrens op het verwachte aantal individuen. Geen van beide houdt rekening met het zogenaamde "*schema creation*". Deze twee stellingen nemen altijd het slechts mogelijke gedrag aan. Als er iets fout zou kunnen gaan, zoals een crossover die een individu dat in schema σ zat, verplaatst naar een schema τ, nemen we aan dat dit ook fout gaat. Dit laatste is nu schema creation.

Een derde categorie van schemastellingen, zijn de *exacte schemastellingen*. Stephens en Waelbroeck hebben exacte schemastellingen uitgewerkt voor het GA [27; 28]. In "Foundations of Genetic Programming" stellen Langdon en Poli een exacte schemastelling voor GP voor [16]. Ze noemen deze de *hyperschemastel-*

ling. In hyperschema's zijn er twee verschillende soorten "don't care" symbolen. Namelijk $*$ voor arbitraire bomen en $\#$ voor arbitraire knopen met een gespecifieerde ariteit. Hyperschema's zijn dus een combinatie van Rosca schema's en Langdon-Poli schema's.

Origineel is deze exacte schemastelling gedefiniëerd voor *one-point crossover* [21]. One-point crossover is een veralgemening van de one-point crossover uit het GA. Bomen worden gealigneerd, waarna uit het gealigneerde deel een punt wordt gekozen waar de deelbomen worden omgewisseld. Op bomen die dezelfde structuur hebben is dit eenvoudig te implementeren. Bomen die echter een verschillende structuur hebben kunnen ook in bepaalde mate gealigneerd worden. Dit komt neer op het zoeken naar een gemeenschappelijke structuur vertrekkende vanuit wortel. Meer informatie over one-point crossover is te vinden in [21]. Later is de stelling ook uitgebreid tot de standaard crossover, zoals beschreven in deze sectie.

Al deze schemastellingen plakken een getal op het verwachte aantal individuen dat aan een schema voldoen in populatie P_{i+1} gegeven populatie P_i. Het GA en GP zijn beide random, en alle schemastellingen zijn sterk verstrengeld met de populatie. Het uitrekenen van de formules voorzien door de schemastellingen komt in principe neer op het uitvoeren van het algoritme. Er is geen enkele stelling die stelt dat het GA of GP effectief altijd zal convergeren naar een oplossing. Er is ook geen stelling die een uitspraak doet over hoe goed een oplossing, berekent door GA of GP, is. Dit is nog een groot gebrek in de theorie achter het GA en GP.

Hoofdstuk 4

Databases

Tot dusver hebben we ons geconcentreerd op de methode om query afleiding te implementeren. Laten we nu eens kijken naar het aspect database en relationele algebra. We beginnen met te specifiëren wat een database, een relatie, een query en een globaal schema zijn. In de volgende sectie wordt de relationele algebra voorgesteld. Waarna we overgaan naar de definitie van het probleem dat we proberen op te lossen m.b.v. Genetisch Programmeren. We zullen echter vaststellen dat er een aantal problemen ontstaan bij het gebruik van de relationele algebra (RA) in GP. Daarom wordt de cylindrische algebra geïntroduceerd, omdat deze algebra beter voldoet aan onze eisen. Op dit punt is het noodzakelijk dat we de equivalentie van de cylindrische algebra (CA) en de relationele algebra aantonen en ons probleem herdefiniëren. Het hoofdstuk wordt afgesloten met een sectie over de implementatie van cylindrische algebra in het Genetisch Programmeren.

4.1 Databases en Relaties

In deze sectie definiëren we wat een database, een relatie en een query zijn. Hiervoor introduceren we de concepten databaseschema en relatieschema.

We veronderstellen volgende verzamelingen: $\mathbb{U}$, $\mathbb{R}$ en $\mathbb{V}$. $\mathbb{U}$ is het universum van alle attribuutnamen. In deze tekst stellen we elementen van $\mathbb{U}$ voor door de eerste letters van het alfabet: A, B, C, ... $\mathbb{R}$ is het universum van alle relatienamen. Voor relatienamen gebruiken we de letters R, S, T, ... $\mathbb{V}$ is het universum van alle waarden.

Databases, Relaties en Schema's Een *databaseschema* $\mathbb{S}$ is een eindige verzameling relatienamen, plus een toekenning van een relatieschema aan elke relatienaam. Formeel is dus $\mathbb{S}$ een functie $\mathbb{S} : \mathcal{S} \rightarrow \mathcal{P}^{fin}(\mathbb{U})$, waar $\mathcal{S}$ een eindig deel van $\mathbb{R}$ is en waarbij $\mathcal{P}^{fin}(\mathbb{U})$ de verzameling van alle eindige deelverzamelingen van $\mathbb{U}$ is. We zien dus dat een *relatieschema* niets anders is dan een eindige verzameling attribuutnamen.

Laten we ook nog enkele functies definiëren over deze concepten. De functie

relaties($\mathbb{S}$) geeft, gegeven een databaseschema $\mathbb{S}$, de relatienamen in dat databaseschema. Dus in de hierboven gebruikte notatie is *relaties*($\mathbb{S}$) als een $\mathcal{S}$.

We gebruiken de volgende conventie i.v.m. relaties, relatieschema's en relatie instanties. Als we spreken over een relatie (relatienaam) dan gebruiken we T. Voor relatieschema's gebruiken we R. Relatie instanties duiden we aan met r. We volgen deze conventie, tenzij anders aangegeven, als er bijvoorbeeld meerdere relaties aangehaald worden.

Tupel Laten we formeel definiëren wat we verstaan onder tupel. Zij $R \subseteq \mathbb{U}$ een relatieschema. Een tupel over R (van type R) is dan een functie $R \to \mathbb{V}$. Een relatie instantie van een relatieschema is een verzameling van tupels over dat relatieschema. De verzameling van alle relatie instanties van een relatieschema R noteren we als $inst(R)$.

Database Instantie Voor de eenvoud werken we in dit document alleen met *relaties met getallen*. Formeel werken we dan vanaf nu met $\mathbb{V} = \mathbb{N}$.

Zij $\mathbb{S}$ een databaseschema. We definiëren nu wat een instantie is van $\mathbb{S}$. Formeel is een database instantie d over $\mathbb{S}$ (van $\mathbb{S}$) een koppel (n, D) met

- $n \in \mathbb{N}$
- D kent aan elke $T \in relaties(\mathbb{S})$ een relatie $d(T)$ toe, waar $d(T)$ van type $\mathbb{S}(T)$ is; en waar alle waarden komen uit $\{1, ..., n\}$

We noteren het domein van d als volgt, $dom(d) = \{1, ..., n\}$. Merk op dat n de maximale waarde is die in d kan voorkomen.

Voor een relatieschema R en een relatie instantie $r \in inst(R)$, definiëren we $adom(r) := \bigcup_{A \in R}\{t(A) \mid t \in r\}$. Als r een relatie instantie is in een database instantie, dan geldt: $adom(r) \subseteq dom(d)$.

We noteren met $inst(\mathbb{S})$ de verzameling van alle database instanties over databaseschema $\mathbb{S}$.

Query Zij $\mathbb{S}$ een databaseschema, zij R een relatieschema. Een query van type $\mathbb{S} \to R$ is een functie $\mathbb{Q} : inst(\mathbb{S}) \to inst(R)$ zodat voor alle database instanties d $adom(\mathbb{Q}(d)) \subseteq dom(d)$.

We beschouwen niet zomaar elke functie als een echte database query. Van zulke queries verwachten we in de eerste plaats dat ze in bulk werken met de data en voornamelijk kijken naar de relaties die bestaan tussen de elementen in de database. Daarom laten we alleen puur structurele queries toe. Een query $\mathbb{Q}$ is puur structureel als voor elke database instantie $d = (n, D)$ en permutatie $f : \{1, ..., n\} \to \{1, ..., n\}$ geldt: $\mathbb{Q}(f(d)) = f(\mathbb{Q}(d))$. Intuïtief betekent dit dat het resultaat van een query niets te maken heeft met de getallen, maar met de plaatsing van de getallen.

Voor een verdere discussie van onze beslissing om alleen puur structurele queries toe te laten verwijzen we naar sectie 4.7.

Globaal relatieschema Een query is gedefiniëerd over een databaseschema $\mathbb{S}$ en een output relatieschema R. We definiëren nu het *globale relatieschema* van een query $\mathbb{Q}$ als de verzameling van alle attribuutnamen in de relatieschema's van $\mathbb{S}$ en de attribuutnamen in het output relatieschema R. Formeel is dit $\bigcup_{S \in \mathbb{S}} \mathbb{S}(S) \cup R$.

4.2 Relationele Algebra

Na de definitie van databases, relaties en queries kunnen we overgaan tot de definitie van de relationele algebra, een formalisme om queries uit te drukken. We splitsen deze definitie op in twee delen. Het eerste deel definiëert de syntaxis van de relationele algebra, het tweede deel definiëert de semantiek van de taal.

Veronderstel dat we een databaseschema $\mathbb{S}$ hebben.

4.2.1 Syntaxis

We definiëren de syntaxis, zie tabel 4.1 van de relationele algebra aan de hand van een attribuut grammatica. Deze legt voorwaarden op aan de onderdelen van een regel.

Tevens definiëren we de functie $\overline{att}(e)$ recursief op de structuur van de relationele algebra. Deze functie geeft al resultaat alle attribuutnamen die gebruikt zijn in de expressie.

We voeren de functie $att(e)$ in een RA expressie. Stel dat r het resultaat is van de evaluatie van de expressie e. r is een relatie, en is bijgevolg gedefiniëerd over een relatieschema R. De functie $att(e) = R$.

4.2.2 Semantiek

De semantiek van de relationele algebra bestaat erin dat we aan elke expressie e een query $[\![e]\!] : \mathbb{S} \to att(e)$ toekennen. Voor een willekeurige database instantie $d = (n, D)$ van $\mathbb{S}$, definiëren we $[\![e]\!](d)$ per inductie als volgt:

- $[\![T]\!](d) := d(T)$.
- $[\![e_1 \cup e_2]\!](d) := [\![e_1]\!](d) \cup [\![e_2]\!](d)$
- $[\![e_1 \cap e_2]\!](d) := [\![e_1]\!](d) \cap [\![e_2]\!](d)$
- $[\![e_1 - e_2]\!](d) := [\![e_1]\!](d) - [\![e_2]\!](d)$
- $[\![\sigma_{A=B}(e)]\!](d) := \{t \in [\![e]\!](d) \mid t(A) = t(B)\}$
- $[\![\pi_{A_1,\ldots,A_i}(e)]\!](d) := \{t|_{A_1,\ldots,A_i} \mid t \in [\![e]\!](d)\}$
- $[\![\rho_{A \to B}(e)]\!](d) := \{t|_{att(e)-\{A\}} \cup \{(B, t(A))\} \mid t \in [\![e]\!](d)\}$
- $[\![e_1 \bowtie e_2]\!](d) := \{t : att(e_1) \cup att(e_2) \to \{1, \ldots, n\} \mid t|_{att(e_1)} \in [\![e_1]\!](d) \wedge t|_{att(e_2)} \in [\![e_2]\!](d)\}$

e	$\rightarrow$	T	voorwaarde $T \in relaties(\mathbb{S})$ $\overline{att}(e) := \mathbb{S}(T)$
	$\mid$	$e_1 \cup e_2$	voorwaarde $att(e_1) = att(e_2)$ $\overline{att}(e) := \overline{att}(e_1) \cup \overline{att}(e_2)$ $att(e) := att(e_1)$
	$\mid$	$e_1 \cap e_2$	voorwaarde $att(e_1) = att(e_2)$ $\overline{att}(e) := \overline{att}(e_1) \cup \overline{att}(e_2)$ $att(e) := att(e_1)$
	$\mid$	$e_1 - e_2$	voorwaarde $att(e_1) = att(e_2)$ $\overline{att}(e) := \overline{att}(e_1) \cup \overline{att}(e_2)$ $att(e) := att(e_1)$
	$\mid$	$\sigma_{A=B}(e_1)$	voorwaarde $A, B \in att(e_1)$ $\overline{att}(e) := \overline{att}(e_1)$ $att(e) := att(e_1)$
	$\mid$	$\pi_{A_1,...A_i}(e_1)$	voorwaarde $A_1, ..., A_i \in att(e_1)$ $\overline{att}(e) := \overline{att}(e_1)$ $att(e) := \{A_1, ..., A_n\}$
	$\mid$	$\rho_{A \rightarrow B}(e_1)$	voorwaarde $A \in att(e_1)$ en $B \notin att(e_1)$ $\overline{att}(e) := \overline{att}(e_1) \cup \{B\}$ $att(e) := (att(e_1) - \{A\}) \cup \{B\}$
	$\mid$	$e_1 \bowtie e_2$	$\overline{att}(e) := \overline{att}(e_1) \cup \overline{att}(e_2)$ $att(e) := att(e_1) \cup att(e_2)$

Tabel 4.1: Grammatica die de Relationele Algebra definiëert

4.3 Het Probleem

4.3.1 Definitie

In dit document zoeken we een oplossing voor een probleem. We gaan nu over tot de formele definitie van het probleem.

Eerst definiëren we nog een functie, nl. $att(\mathbb{S})$ met $\mathbb{S}$ een databaseschema. $att(\mathbb{S}) = \bigcup_{T \in relaties(\mathbb{S})} \mathbb{S}(T)$.

Het probleem dat we gaan definiëren is een Machine Learning probleem, m.a.w. een leerprobleem. We leren een eerste-orde query[1] a.h.v. voorbeelden. Dit is een *concept learning* probleem waarbij het concept dat we leren een query is. Zulk een concept is gedefiniëerd over $inst(\mathbb{S})$, met $\mathbb{S}$ een databaseschema. Het is nu de bedoeling om het concept, de query, te benaderen d.m.v. een verzameling van voorbeelden waaruit het leeralgoritme moet veralgemenen.

Gegeven Zij $\mathbb{S}$ een databaseschema. Zij O een output relatieschema. Een ein-

[1] Een query is eerste-orde als er een relationele algebra expressie bestaat die deze query uitdrukt.

dige verzameling voorbeelden, bestaande uit koppels $(d = (n, D), r)$, met database instantie d van $\mathbb{S}$ en een relatie instantie r over het relatieschema O. Zij Y een verzameling "extra" attribuutnamen.

Gevraagd Een RA expressie e van type $\mathbb{S} \rightarrow O$, zodat er voor elke subexpressie e' van e geldt: $att(e') \subseteq att(\mathbb{S}) \cup O \cup Y$. De expressie genereert, voor elk voorbeeld (d, r) een output die zoveel mogelijk gelijkt op r, vertrekkende van d. De expressie moet zo beknopt mogelijk zijn ("Occam's razor").

We passen "Occam's razor" toe, omdat het theoretisch mogelijk is om een query te schrijven die herkent of een gegeven database instantie d gelijk is aan een van de voorbeelden, indien dit klopt geeft de query exact de output relatie van het voorbeeld. Indien d niet gelijk is aan een voorbeeld wordt de lege verzameling terug gegeven. Het is echter de bedoeling dat het algoritme dat dit leerprobleem aanpakt veralgemeent. Het algoritme moet in staat zijn om een zinnige output te geven voor nieuwe koppels (d', r').

Wij passen Genetisch Programmeren toe, omdat we de relationele algebra kunnen beschouwen als een programmeertaal. Een programmeertaal is nu juist de geschikte hypothese voorstelling in het geval van GP. Meer over de implementatie in sectie 4.8.

4.3.2 Tekortkomingen van de Relationele Algebra

De relationele algebra kampt echter met een zwaar tekort als het gaat om GP. De unie, doorsnede en verschil verwachten dat de twee inputrelaties hetzelfde relatieschema hebben. Als GP echter mutatie of recombinatie toepast, is het mogelijk dat de twee inputrelaties een verschillend schema hebben. Laten we even een eenvoudig voorbeeld van dit probleem bekijken. Stel dat er op een bepaald moment in de populatie een individu x van de vorm $R - S$ voorkomt, met R en S relaties. Als x wordt geselecteerd voor mutatie, en S wordt vervangen door T, een relatie, waarbij $att(S) \neq att(T)$, dan is het verschil niet langer gedefiniëerd. Individu x is op dat moment waardeloos geworden en neemt alleen maar plaats in in de populatie.

Selectie (σ) en projectie(π) kampen met een gelijkaardig probleem. Beide hebben als input een relatie en attribuutnamen als parameters. Als deze attribuutnamen niet voorkomen in de input relatie zijn beide operaties niet gedefiniëerd.

4.3.3 Oplossing Tekortkomingen

We zouden aan deze tekortkomingen kunnen voorbijgaan. Stel dat we gewoon verder werken met de RA, en als er ongedefiniëerd expressies voorkomen, negeren we deze. Dit zal echter leiden tot een populatie die capaciteit verliest aan individuen die genegeerd worden.

Een betere oplossing, die de volledige capaciteit gebruikt, is het aanwenden van een andere algebra die niet onderhevig is aan deze tekortkomingen. De cylindrische

$$\begin{array}{lcll} e & \rightarrow & T & (T \in \mathit{relaties}(\mathbb{S})) \\ & | & \mathit{cyl}_A(e) & (A \in U) \\ & | & e \cup e & \\ & | & e^c & \\ & | & \sigma_{A=B}(e) & A, B \in U \end{array}$$

Tabel 4.2: Syntaxis van de Cylindrische Algebra

algebra is zo een algebra. Als we ook kunnen aantonen dat deze twee algebra's equivalentie zijn qua uitdrukkingskracht, kunnen we de RA vervangen door de CA.

4.4 Cylindrische Algebra

Strikt bekeken moeten we eigenlijk spreken van de Cylindrische verzameling Algebra. Dit is een speciaal soort cylindrische algebra, die uitgebreid werd bestudeerd door Henkin en Tarski sinds 1961 [10; 7; 8; 9]. Maar voor het gemak gebruiken we cylindrische algebra om de Cylindrische verzameling Algebra aan te duiden. Imielinski en Lipski waren de eersten die een verband zagen tussen de cylindrische algebra en de relationele algebra van Codd [13].

Net zoals bij de definitie van de relationele algebra zullen we de definitie van de cylindrische algebra (CA) opdelen in twee delen. Eerst definiëren we de syntaxis van deze algebra, daarna definiëren we de semantiek.

Zij U een relatieschema. De cylindrische algebra is gedefiniëerd over een beperkt aantal schema's, nl. de U-databaseschema's. Elke relatie in een U-databaseschema heeft U als relatieschema.

Het resultaat van een CA expressie e, geëvalueerd over een database instantie $d = (n, D)$, hangt af van n. Dit in tegenstelling tot expressies in de RA. RA expressies zijn alleen afhankelijk van het actief domein van een database instantie.

4.4.1 Syntaxis

De syntaxis van de cylindrische algebra wordt samengevat in 4.2 als een contextvrije taal. We veronderstellen een U-databaseschema $\mathbb{S}$ met een relatieschema U. De syntaxis hangt alleen af van het databaseschema $\mathbb{S}$.

We definiëren $att(e)$ ook voor de cylindrische algebra. Dit is zeer eenvoudig: $att(e)$ is gelijk aan U.

4.4.2 Semantiek

Zij $\mathbb{S}$ een databaseschema. De semantiek van de cylindrische algebra bestaat erin dat we een aan elke expressie e een query $[\![e]\!] : \mathbb{S} \rightarrow att(e)$ toekennen. Voor een willekeurige database instantie $d = (n, D)$ over $\mathbb{S}$ definiëren we per inductie:

- $[\![T]\!](d) := d(T)$
- $[\![cyl_A(e)]\!](d) := \{t' : U \to \{1, ..., n\} \mid \exists t \in [\![e]\!](d) : t'_{|U-\{A\}} = t_{|U-\{A\}}\}$
- $[\![e_1 \cup e_2]\!](d) := [\![e_1]\!](d) \cup [\![e_2]\!](d)$
- $[\![e^c]\!](d) := \{t : U \to \{1, ..., n\} \mid t \notin [\![e]\!](d)\}$
- $[\![\sigma_{A=B}(e)]\!](d) := \{t \in [\![e]\!](d) \mid t(A) = t(B)\}$

4.4.3 Uniformeren en Uitbreiden

Tenslotte voeren we nog twee begrippen in. Beide begrippen zijn noodzakelijk om in de volgende sectie de equivalentie tussen de relationele en de cylindrische algebra te kunnen aantonen. De twee begrippen zijn *uniform databaseschema* en de *uitbreiding van een databaseschema*.

Uniformeren

Databaseschema Zij $\mathbb{S}$ een databaseschema. Zij $\mathcal{S} = relaties(\mathbb{S})$. We uniformeren $\mathbb{S}$, genoteerd $\overline{\mathbb{S}}$ door voor elke $T \in \mathcal{S}$, $T \mapsto att(\mathbb{S})$. $relaties(\mathbb{S}) = relaties(\overline{\mathbb{S}})$.

Merk op dat het uniformeren van een U-databaseschema uit de cylindrische algebra geen effect heeft, aangezien $att(\mathbb{S}) = U$, en alle relaties in een U-databaseschema U als relatieschema hebben.

Het uniformeren van een databaseschema resulteert bijgevolg in een U-databaseschema (met $U = att(\mathbb{S})$).

Database instantie Zij $d = (n, D)$ een database instantie over $\mathbb{S}$. Als we $\mathbb{S}$ uniformeren, moeten we dit ook doen voor elke database instantie van $\mathbb{S}$. Voor elke $T \in relaties(\overline{\mathbb{S}})$, $\overline{d}(T) := \{t : \overline{\mathbb{S}}(T) \to \{1, ..., n\} \mid t|_{\mathbb{S}(T)} \in d(T)\}$.

Uitbreiding

Databaseschema Zij $\overline{\mathbb{S}}$ een uniform databaseschema. Zij $\mathcal{S} = relaties(\mathbb{S})$. We breiden $\overline{\mathbb{S}}$ uit met een verzameling attribuutnamen Y, genoteerd $uitbr_Y(\overline{\mathbb{S}})$. De uitbreiding wordt berekend door voor elke $T \in \mathcal{S}$, $T \mapsto att(\overline{\mathbb{S}}) \cup Y$.

Database instantie Zij $\overline{d} = (n, D)$ een database instantie over uniform databaseschema $\overline{\mathbb{S}}$. Als we $\overline{\mathbb{S}}$ uitbreiden met Y, moeten we eveneens $\overline{d}$ aanpassen. Voor elke $T \in relaties(\overline{\mathbb{S}})$, $uitbr_Y(\overline{d})(T) := \{t : \overline{\mathbb{S}}(T) \cup Y \to \{1, ..., n\} \mid t|_{\overline{\mathbb{S}}(T)} \in \overline{d}(T)\}$.

Het uitbreiden van een database instantie komt overeen met het uitbreiden van elke relatie instantie in de database instantie.

Relatie instantie Zij r een relatie instantie over een relatieschema R. De uitbreiding van r met een verzameling attribuutnamen Y, wordt genoteerd als $uitbr^n_Y(r)$. Waarbij n dezelfde n is als in een database instantie. Aangezien deze database instantie nu niet bekend is, wordt n meegegeven aan de functie. We definiëren $uitbr^n_Y(r) = \{t : R \cup Y \rightarrow \{1, ..., n\} \mid t|_R \in r\}$.

Merk op: uniformeren is uitbreiden met $att(\mathbb{S})$.

4.5 Equivalentie Relationele Algebra en Cylindrische Algebra

In deze sectie bewijzen we dat de cylindrische algebra *dezelfde uitdrukkingskracht* heeft als de relationele algebra op structurele queries met relaties die verzamelingen zijn. We bewijzen hiervoor stelling 4.1.

Deze stelling komt uit een survey artikel van Van den Bussche [29]. In dat artikel wordt de stelling echter geformuleerd in termen van de relationele calculus. We formuleren deze stelling nu in de relationele algebra. Merk op dat O puur mathematisch in de stelling overbodig is, aangezien $O \subseteq Y$. Later hebben we O echter wel nodig.

Stelling 4.1. *Zij $\mathbb{S}$ een databaseschema. Zij e een relationele algebra expressie over $\mathbb{S}$. Zij $O := att(e)$. Zij $Y := \overline{att}(e)$. Dan bestaat een cylindrische algebra expressie e' over $uitbr_{O \cup Y}(\overline{\mathbb{S}})$ zodat voor elke database instantie $d = (n, D)$ over $\mathbb{S}$ geldt:*
$[\![e']\!](uitbr_{O \cup Y}(\overline{d})) = uitbr^n_{att(\mathbb{S}) \cup O \cup Y}([\![e]\!](d))$.

Bewijs. We bewijzen deze stelling d.m.v. inductie op de syntaxis van de RA. Voor elke regel van de RA claimen we eerst wat e' moet zijn. Vervolgens bewijzen we deze claim. Omdat we via inductie werken, mogen we aannemen dat voor elke deelexpressie van e reeds een equivalentie CA expressie bestaat.

Geval $e = T$: In dit geval werkt $e' = T$:

$$\begin{aligned}
[\![T]\!](uitbr_{O \cup Y}(\overline{d})) &= (uitbr_{O \cup Y}(\overline{d}))(T) && (4.1)\\
&= uitbr_{O \cup Y}(uitbr_{att(\mathbb{S})}(d))(T) && (4.2)\\
&= (uitbr_{att(\mathbb{S}) \cup O \cup Y}(d))(T) && (4.3)\\
&= uitbr^n_{att(\mathbb{S}) \cup O \cup Y}(T) && (4.4)
\end{aligned}$$

We starten in de linkerkant van het de te bewijzen formule. In 4.1 gebruiken we de definitie van de semantiek. Het uniformeren van een database instantie, is het uitbreiden van die database instantie in 4.2. Stel dat we een database instantie uitbreiden met een verzameling attribuutnamen Y en daarna diezelfde instantie nogmaals uitbreiden met een verzameling attribuutnamen Z. Dit is

identiek hetzelfde als het uitbreiden van deze instantie met $Y \cup Z$. In de laatste stap, 4.4, gebruiken we de observatie dat de uitbreiding van een database instantie hetzelfde is als het uitbreiden van alle relatie instanties. In het bijzonder dus van relatie instantie van relatie T.

Geval $e = e_1 \cup e_2$**:** In dit geval werkt $e' = e'_1 \cup e'_2$:

$$\begin{aligned}
[\![e'_1 \cup e'_2]\!](uitbr_{O\cup Y}(\overline{d})) &= [\![e'_1]\!](uitbr_{O\cup Y}(\overline{d})) \cup [\![e'_2]\!](uitbr_{O\cup Y}(\overline{d})) \quad (4.5)\\
&= uitbr^n_{att(\mathbb{S})\cup O\cup Y}([\![e_1]\!](d)) \cup \\
&\quad\; uitbr^n_{att(\mathbb{S})\cup O\cup Y}([\![e_2]\!](d)) \quad (4.6)\\
&= uitbr^n_{att(\mathbb{S})\cup O\cup Y}([\![e_1]\!](d) \cup [\![e_2]\!](d)) \quad (4.7)\\
&= uitbr^n_{att(\mathbb{S})\cup O\cup Y}([\![e_1 \cup e_2]\!](d))
\end{aligned}$$

De linkerkant van de te bewijzen formule kunnen we herschrijven door de semantiek van de CA te gebruiken. Uit de inductie en de uniformering volgt formule 4.6. Volgens lemma 4.1 mogen we de unie binnen de uitbreiding brengen. Tot slot herschrijven we de formule door gebruik te maken van de semantiek van de RA.

Geval $e = e_1 \cap e_2$**:** In dit geval werkt $e' = (e'^c_1 \cup e'^c_2)^c$:

Hieronder nemen we regelmatig het complement r^c van een U-relatie instantie r, met $U = att(\mathbb{S}) \cup O \cup Y$, waarmee we bedoelen dat $r^c = \{t : U \to \{1, ..., n\} \mid t \notin r\}$.

$$\begin{aligned}
[\![(e'^c_1 \cup e'^c_2)^c]\!](uitbr_{O\cup Y}(\overline{d})) &= \left([\![e'^c_1 \cup e'^c_2]\!]\left(uitbr_{O\cup Y}(\overline{d})\right)\right)^c\\
&= \left([\![e'^c_1]\!]\left(uitbr_{O\cup Y}(\overline{d})\right) \cup [\![e'^c_2]\!]\left(uitbr_{O\cup Y}(\overline{d})\right)\right)^c\\
&= \left([\![e'_1]\!]\left(uitbr_{O\cup Y}(\overline{d})\right)^c \cup [\![e'_2]\!]\left(uitbr_{O\cup Y}(\overline{d})\right)^c\right)^c\\
&= [\![e'_1]\!]\left(uitbr_{O\cup Y}(\overline{d})\right) \cap [\![e'_2]\!]\left(uitbr_{O\cup Y}(\overline{d})\right) \quad (4.8)\\
&= uitbr^n_{att(\mathbb{S})\cup O\cup Y}\left([\![e_1]\!](d)\right) \cap\\
&\quad\; uitbr^n_{att(\mathbb{S})\cup O\cup Y}\left([\![e_2]\!](d)\right)\\
&= uitbr^n_{att(\mathbb{S})\cup O\cup Y}\left([\![e_1]\!](d) \cap [\![e_2]\!](d)\right)\\
&= uitbr^n_{att(\mathbb{S})\cup O\cup Y}\left([\![e_1 \cap e_2]\!](d)\right)
\end{aligned}$$

Eerst ontmantelen we de formule, aan de hand van de semantiek van de CA. We kunnen de wet van De Morgan toepassen op deze ontmanteling, wat leidt tot formule 4.8. De volgende stap is het toepassen van de inductie. Vervolgens passen we lemma 4.2 toe. De semantiek van de RA leidt tot wat we wilden bewijzen.

In de syntaxis van de CA komt de doorsnede niet voor, omdat de doorsnede kan uitgedrukt worden in functie van de unie en het complement, $r \cap s = (r^c \cup s^c)^c$. De doorsnede is bijgevolg geen primitieve operator, maar een afkorting. In de RA is

de doorsnede ook geen primitieve operator. De doorsnede kan uitgedrukt worden in functie van de unie en het verschil als volgt: $r \cap s = r - (r - s)$.

Geval $e = e_1 - e_2$: In dit geval werkt $e' = (e_1'^c \cup e_2')^c$:

$$\begin{aligned}
[\![(e_1'^c \cup e_2')^c]\!] \left(uitbr_{O \cup Y}\left(\overline{d}\right)\right) &= [\![e_1'^c \cup e_2']\!] \left(uitbr_{O \cup Y}\left(\overline{d}\right)\right)^c \\
&= \left([\![e_1']\!] \left(uitbr_{O \cup Y}\left(\overline{d}\right)\right)^c \cup [\![e_2']\!] \left(uitbr_{O \cup Y}\left(\overline{d}\right)\right)\right)^c \\
&= [\![e_1']\!] \left(uitbr_{O \cup Y}\left(\overline{d}\right)\right) \cap [\![e_2']\!] \left(uitbr_{O \cup Y}\left(\overline{d}\right)\right)^c \\
&= uitbr^n_{att(\mathbb{S}) \cup O \cup Y}\left([\![e_1]\!](d)\right) \cap \\
&\quad\; uitbr^n_{att(\mathbb{S}) \cup O \cup Y}\left([\![e_2]\!](d)\right)^c \\
&= uitbr^n_{att(\mathbb{S}) \cup O \cup Y}\left([\![e_1]\!](d) \cap \left([\![e_2]\!](d)\right)^c\right) \qquad (4.9) \\
&= uitbr^n_{att(\mathbb{S}) \cup O \cup Y}\left([\![e_1]\!](d) - [\![e_2]\!](d)\right) \\
&= uitbr^n_{att(\mathbb{S}) \cup O \cup Y}\left([\![e_1 - e_2]\!](d)\right)
\end{aligned}$$

We beginnen met de CA expressie te ontmantelen en lemma's 4.3 en 4.2 toe te passen. De inductie komt meespelen en brengt ons tot 4.9. Vervolgens passen we lemma 4.4 toe om het verschil te introduceren, en tegelijkertijd van de doorsnede en complement af te geraken. Ten slotte gebruiken we de semantiek van de RA om bewijs af te ronden.

Merk op dat we ook volgende afkorting hadden kunnen gebruiken: $e' = (e_1' \cap e_2^c)$.

Geval $e = \sigma_{A=B}(e_1)$: In dit geval werkt $e' = \sigma_{A=B}(e_1')$:

$$\begin{aligned}
[\![\sigma_{A=B}(e_1')]\!](uitbr_{O \cup Y}(\overline{d})) &= \{t \in [\![e_1']\!](uitbr_{O \cup Y}(\overline{d}) \mid t(A) = t(B)\} \\
&= \{t \in uitbr^n_{att(\mathbb{S}) \cup O \cup Y}([\![e_1]\!](d)) \mid t(A) = t(B)\} \\
&= [\![\sigma_{A=B}(uitbr^n_{att(\mathbb{S}) \cup O \cup Y}([\![e_1]\!](d)))]\!](d) \\
&= uitbr^n_{att(\mathbb{S}) \cup O \cup Y}([\![\sigma_{A=B}(e_1)]\!](d))
\end{aligned}$$

We beginnen met de semantiek van de CA toe te passen. In de volgende stap passen we de inductie toe. Dit komt overeen met de selectie op de uitbreiding van de semantiek van e_1. Tot slot passen we lemma 4.5 toe. We mogen dat lemma toepassen aangezien $A, B \in att(e)$.

Geval $e = \pi_{A_1,...,A_m}(e_1)$: We voeren eerst nog wat extra notatie in. Veronderstel dat $Z = \{B_1, ..., B_k\}$ een verzameling van attribuutnamen is, zodat $Z \subseteq att(\mathbb{S}) \cup O \cup Y$. Zij r een relatie instantie, formeel noteren we:
$cyl_Z(r) \equiv cyl_{B_1}\left(cyl_{B_2}\left(...\left(cyl_{B_k}(r)\right)\right)\right)$.
Eerst bewijzen we dat $cyl_Z(r) = cyl_{B_1}(cyl_{B_2}(...(cyl_{B_k}(r))))$. We bewijzen dit in lemma 4.6

In dit geval werkt $e' = cyl_{O-\{A_1,...,A_m\}}(e_1')$:

$$\begin{aligned}
& [\![cyl_{O-\{A_1,...,A_m\}}(e_1')]\!](uitbr_{O\cup Y}(\overline{d})) \\
= \ & \{t' : att(\mathbb{S}) \cup O \cup Y \rightarrow \{1,...,n\} \mid t \in [\![e_1']\!](uitbr_{O\cup Y}(\overline{d})) : \\
& t|_{O-\{A_1,...,A_m\}} = t'|_{O-\{A_1,...,A_m\}}\} \\
= \ & \{t' : att(\mathbb{S}) \cup O \cup Y \rightarrow \{1,...,n\} \mid t \in uitbr^n_{att(\mathbb{S})\cup O\cup Y}([\![e_1]\!](d)) : \\
& t|_{O-\{A_1,...,A_m\}} = t'|_{O-\{A_1,...,A_m\}}\} \\
= \ & uitbr^n_{att(\mathbb{S})\cup O\cup Y}(\{t|_{O-\{A_1,...,A_m\}} \mid t \in [\![e_1]\!](d)\}) \\
= \ & uitbr^n_{att(\mathbb{S})\cup O\cup Y}([\![\pi_{A_1,...,A_m}(e_1)]\!](d))
\end{aligned}$$

Zoals gewoonlijk passen we eerst de semantiek van de CA toen, waarna we de inductie toepassen. We weten dat $\{A_1, ..., A_m\} \subset att(e_1)$. Bijgevolg kunnen we de beperking binnen de uitbreiding brengen. Tot slot gebruiken we de semantiek van de RA om te herschrijven.

Geval $e = \rho_{A\rightarrow B}(e_1)$ In dit geval werkt $e' = cyl_A(\sigma_{A=B}(cyl_B(e_1')))$: We gebruiken $U = att(\mathbb{S}) \cup O \cup Y$ als afkorting.

$$\begin{aligned}
& [\![cyl_A(\sigma_{A=B}(cyl_B(e_1')))]\!](uitbr_{O\cup Y}(\overline{d})) \\
= \ & \{t' : U \rightarrow \{1,...,n\} \mid \exists t \in [\![\sigma_{A=B}(cyl_B(e_1'))]\!](uitbr_{O\cup Y}(\overline{d})) : \\
& t'|_{U-\{A\}} = t|_{U-\{A\}}\} \\
= \ & \{t' : U \rightarrow \{1,...,n\} \mid \exists t'' \in \{t \in [\![cyl_B(e_1')]\!](uitbr_{O\cup Y}(\overline{d})) \mid t(A) = t(B)\} : \\
& t'|_{U-\{A\}} = t''|_{U-\{A\}}\} \\
= \ & \{t' : U \rightarrow \{1,...,n\} \mid \exists t'' \in \{t : U \rightarrow \{1,...,n\} \mid \\
& \exists t''' \in [\![e_1']\!](uitbr_{O\cup Y}(\overline{d})) : t'''|_{U-\{B\}} = t|_{U-\{B\}} \wedge t(A) = t(B)\} : \\
& t'|_{U-\{A\}} = t''|_{U-\{A\}}\} \\
= \ & \{t' : U \rightarrow \{1,...,n\} \mid \exists t'' \in uitbr^n_U(\{t \cup \{(B, t(A))\} \mid t \in [\![e_1]\!](d)\}) : \\
& t'|_{U-\{A\}} = t''|_{U-\{A\}}\} \\
= \ & uitbr^n_U(\{t|_{att(e_1)-\{A\}} \cup \{(B, t(A))\} \mid t \in [\![e_1]\!](d)\}) \\
= \ & uitbr^n_U([\![\rho_{A\rightarrow B}(e_1)]\!](d))
\end{aligned}$$

We starten met de semantiek van de cylindrificatie toe te passen. Vervolgens werken we in een "interne" verzameling de semantiek van de selectie uit. We passen nogmaals de semantiek van de cylindrificatie toe, en merken dan op dat de "interne" verzameling herschrijfbaar is naar een gemakkelijkere vorm. Er staat namelijk dat B onbeperkt is door t''' maar dat B wel beperkt wordt door $t(A)$. We weten dat $t(A) = t'''(A)$. Dit komt erop neer dat we een attribuut B toevoegen met in elk tupel dezelfde waarde als A. Tegelijkertijd passen we ook de inductie toe, en brengen alles binnen de uitbreiding. In de voorlaatste stap merken we op dat we geen beperking zetten op de waarde van A. Dit kan echter eenvoudiger

geschreven worden door attribuut A weg te laten en dan uit te breiden. Tot slot gebruiken we de semantiek van de RA om het resultaat te herschrijven.

Geval $e = e_1 \bowtie e_2$ In dit geval werkt $e' = e'_1 \cap e'_2$:

$$
\begin{aligned}
& [\![e'_1 \cap e'_2]\!](uitbr_{O \cup Y}(\overline{d})) \\
= \;& \{t \mid t \in [\![e'_1]\!](uitbr_{O \cup Y}(\overline{d})) \wedge t \in [\![e'_2]\!](uitbr_{O \cup Y}(\overline{d}))\} \\
= \;& \{t \mid t \in uitbr_{att(\mathbb{S}) \cup O \cup Y}([\![e_1]\!](d)) \wedge t \in uitbr_{att(\mathbb{S}) \cup O \cup Y}([\![e_2]\!](d))\} \\
= \;& \{t : att(\mathbb{S}) \cup O \cup Y \to \{1, ..., n\} \mid t|_{att(e_1)} \in [\![e_1]\!](d) \wedge t|_{att(e_2)} \in [\![e_2]\!](d)\} \\
= \;& uitbr_{att(\mathbb{S}) \cup O \cup Y}([\![e_1 \bowtie e_2]\!](d))
\end{aligned}
$$

Na het gebruikelijk toepassen van de semantiek van de CA en de inductie, passen we de definitie van de uitbreiding toe. Wat ons exact tot het te bewijzen brengt.

□

Lemma 4.1. *Zij r en s relatie instanties gedefiniëerd over relatieschema R. Voor alle $n \in \mathbb{N}$ en alle $Y \in \mathcal{P}^{fin}(\mathbb{U})$ geldt: $uitbr^n_Y(r \cup s) = uitbr^n_Y(r) \cup uitbr^n_Y(s)$*

Bewijs.

$$
\begin{aligned}
uitbr^n_Y(r \cup s) &= \{t : R \cup Y \to \{1, ..., n\} | t_{|S} \in r \cup s\} \\
&= \{t : R \cup Y \to \{1, ..., n\} | t_{|S} \in r\} \cup \\
&\quad\; \{t : R \cup Y \to \{1, ..., n\} | t_{|S} \in s\} \\
&= uitbr^n_Y(r) \cup uitbr^n_Y(s)
\end{aligned}
$$

□

Lemma 4.2. *Zij r en s relatie instanties gedefiniëerd over relatieschema R. Voor alle $n \in \mathbb{N}$ en alle $Y \in \mathcal{P}^{fin}(\mathbb{U})$ geldt: $uitbr^n_Y(r \cap s) = uitbr^n_Y(r) \cap uitbr^n_Y(s)$*

Bewijs.

$$
\begin{aligned}
uitbr^n_Y(r \cap s) &= \{t : R \cup Y \to \{1, ..., n\} | t_{|S} \in r \cap s\} \\
&= \{t : R \cup Y \to \{1, ..., n\} | t_{|S} \in r\} \cap \\
&\quad\; \{t : R \cup Y \to \{1, ..., n\} | t_{|S} \in s\} \\
&= uitbr^n_Y(r) \cap uitbr^n_Y(s)
\end{aligned}
$$

□

Lemma 4.3. *Zij r een relatie instantie gedefiniëerd over relatieschema R. Voor alle $n \in \mathbb{N}$ en alle $Y \in \mathcal{P}^{fin}(\mathbb{U})$ geldt: $uitbr^n_Y(r)^c = uitbr^n_Y(r^c))$*

Bewijs.

$$
\begin{aligned}
uitbr^n_Y(r)^c &= \{t : R \cup Y \to \{1, ..., n\} | t_{|S} \in r\}^c \\
&= \{t : R \cup Y \to \{1, ..., n\} | t_{|S} \notin r\} \\
&= \{t : R \cup Y \to \{1, ..., n\} | t_{|S} \in r^c\} \\
&= uitbr^n_Y(r^c)
\end{aligned}
$$

□

Lemma 4.4. *Zij r en s relatie instanties gedefiniëerd over relatieschema R. $r \cap s^c = r - s$.*

Bewijs.

$$
\begin{aligned}
r \cap s^c &= \{t | t \in r \wedge t \in s^c\} \\
&= \{t | t \in r \wedge t \notin s\} \\
&= r - s
\end{aligned}
$$

□

Lemma 4.5. *Zij r een relatie instantie over een relatieschema R, zij Y een verzameling attribuutnamen en zij $n \in \mathbb{N}$. Zij $A, B \in R$. Er geldt dan dat: $\sigma_{A=B}(uitbr^n_Y(r)) = uitbr^n_Y(\sigma_{A=B}(r))$.*

Bewijs.

$$
\begin{aligned}
\sigma_{A=B}(uitbr^n_Y(r)) &= \{t \in uitbr^n_Y(r) \mid t(A) = t(B)\} \\
&= \{t : R \cup Y \to \{1, ..., n\} \mid t|_R \in r \wedge t(A) = t(B)\} \\
&= \{t : R \cup Y \to \{1, ..., n\} \mid t|_R \in r \wedge t|_R(A) = t|_R(B)\} \\
&= \{t : R \cup Y \to \{1, ..., n\} \mid t|_R \in \sigma_{A=B}(r)\} \\
&= uitbr^n_Y(\sigma_{A=B}(r))
\end{aligned}
$$

□

Lemma 4.6. *Zij r een relatie instantie over een relatieschema R, zij $Z = \{A_1, A_2, ..., A_k\}$ een verzameling attribuutnamen waarvoor geldt $Z \subseteq R$ en $k, n \in \mathbb{N}$. We bewijzen dat $cyl_{A_1}(cyl_{A_2}(...(cyl_{A_k}(r)))) = \{t' : R \to \{1, ..., n\} \mid \exists t \in r : t|_{R-Z} = t'|_{R-Z}\} \equiv cyl_Z(r)$.*

Bewijs. We bewijzen dit lemma per inductie op k.

Basisgeval $k = 0$: Voor $k = 0$ geldt de stelling trivaal. $r = \{t' : R \to \{1, ..., n\} \mid \exists t' \in r : t = t'\}$.

Inductiestap $k > 0$: We stellen Z' gelijk aan $\{A_2, ..., A_k\}$.

$$\begin{aligned}
cyl_{A_1}(cyl_{A_2}(...(cyl_{A_k}(r)))) &= \{t' : R \rightarrow \{1, ..., n\} \mid \exists t \in cyl_{A_2}(...(cyl_{A_k}(r))) : \\
&\quad t|_{R-\{A_1\}} = t'|_{R-\{A_1\}}\} \\
&= \{t' : R \rightarrow \{1, ..., n\} \mid \exists t : R \rightarrow \{1, ..., n\} : \\
&\quad \exists t'' \in r : t|_{R-\{A_1\}} = t'|_{R-\{A_1\}} \wedge t|_{R-Z'} = t''|_{R-Z'}\} \\
&= \{t' : R \rightarrow \{1, ..., n\} \mid \exists t'' \in r : t''|_{R-Z} = t'|_{R-Z}\}
\end{aligned}$$

Het eerste gelijkheidsteken is het uitschrijven van de definitie van cylindrificatie. Het tweede gelijkheidsteken is de toepassing van de inductie. De derde stap kunnen we in twee delen beschouwen. Beschouw eerst de inclusie van links naar rechts. Zij t' in element van de linkerkant. We weten dat $t'|_{R-\{A_1\}} = t|_{R-\{A_1\}}$. Door het feit dat $A_1 \in Z$ volgt hieruit dat $t'|_{R-Z} = t|_{R-Z}$. We weten dat $Z' \subset Z$ en $t|_{R-Z'} = t''|_{R-Z}$. Hieruit volgt dat $t'|_{R-Z} = t''|_{R-Z}$. De inclusie van rechts naar links is triviaal.

□

4.6 Query Learning

Nu we de equivalentie tussen de cylindrische algebra en de relationele algebra hebben aangetoond kunnen we de probleemstelling aanpassen naar de CA i.p.v. de moeilijker te gebruiken RA. We dopen het probleem nu met de naam *query learning*.

In het oorspronkelijke probleem probeerden we uit een eindige verzameling voorbeelden een RA expressie af te leiden. In stelling 4.1 hebben we aangetoond dat voor elke RA expressie e over een databaseschema $\mathbb{S}$ met output relatieschema O die uitgebreid wordt, een equivalente CA expressie e' bestaat over $uitbr_{O \cup Y}(\overline{\mathbb{S}})$. Waarbij we Y definiëerden als $\overline{att}(e)$. Aangezien we e niet kennen en bijgevolg zeker $\overline{att}(e)$ niet kennen, moeten we de ontbrekende attribuutnamen als een parameter toevoegen aan het probleem. We noemen deze parameter $\hat{Y} = Y - (att(\mathbb{S}) \cup O)$. Dit is ook de reden waarom we O niet konden weglaten uit stelling 4.1, want O en $att(\mathbb{S})$ zijn bekend maar Y niet.

Merk op dat als we $\hat{Y}$ te klein kiezen, de stelling niet geldt. Dit betekent dat GP de juiste query niet zal kunnen vinden.

Gegeven Een databaseschema $\mathbb{S}$. Een output relatieschema O. Een eindige verzameling voorbeelden, bestaande uit koppels $(d = (n, D), r)$. Met $d \in inst(\mathbb{S})$ en r een relatie instantie over relatieschema O. Een verzameling extra attributen $\hat{Y}$.

Gevraagd Een cylindrische algebra expressie e' over databaseschema $uitbr_{O \cup \hat{Y}}(\mathbb{S})$. De expressie e' genereert voor elk voorbeeld, een output die zoveel mogelijk gelijkt op r, vertrekkende van d. De expressie moet zo beknopt mogelijk zijn ("Occam's Razor").

Onze doelstelling is om structurele queries te zoeken m.b.v. Genetisch Programmeren. Deze queries geven een abstracte beschrijving van het structurele verband waarvan we voorbeelden geven. Dit is één van de problemen waarmee we te maken hebben in mutli-relational data mining, zie 4.7.

4.7 Multirelationele Data Mining

Multirelationele data mining of ook wel relational data mining houdt zich bezig met het zoeken naar *patronen over een verzameling tupels* in een database. Voorbeelden van patronen zijn: "driehoeken in een graaf", "Fraudeurs in de belastingsdatabase", ... Dit verschilt dus van de "klassieke" data mining dat patronen zoekt over één of enkele tupels. Voorbeelden zijn: frequent itemsets, "is een persoon een big spender?", ... In de paper van Džeroski krijgen we een introductie tot het veld [5]. Multirelationele data mining leent een aantal dingen van *Inductive logic programming* (ILP) en *Deductive Databases*. Meer informatie over ILP is te vinden in de paper van Muggleton en De Raedt [20].

Omdat we geïnteresseerd zijn in structurele verbanden tussen relaties laten we alleen structurele queries toe. Als we constanten zouden toelaten in de selectie zouden we eerder aan classificatie doen. Classificatie, beslissingsbomen en dergelijke zijn algoritmes die ook voorkomen in multirelationele data mining, maar we laten die in deze tekst buiten beschouwing.

4.8 Genetisch Programmeren met de Relationele Algebra

In de vorige secties hebben we de relationele algebra en de cylindrische algebra besproken alsook hun equivalentie, onder bepaalde voorwaarden, aangetoond. In deze sectie zullen we ons verdiepen in hoe we nu gaan Genetisch Programmeren met databases. Hiervoor bespreken we eerst de leerstructuur, die we invullen om een probleem op te lossen. Verder bespreken we hoe we de fitheid van een expressie bepalen. We besluiten met enkele belangrijke punten i.v.m. de genetische operaties.

4.8.1 Leerstructuur: Populatie en Expressieboom

Ons doel is het vinden van een expressie die van een database naar een oplossing gaat. Om ons doel te bereiken gaan we op een gerichte manier zoeken tussen alle mogelijk expressies naar die expressie die vertrekkende van de database het dichtste in de buurt van de oplossing komt (en mogelijk exact de oplossing uitkomt). We gebruiken als leerstructuur een expressieboom. Een expressieboom stelt een expressie op een zeer natuurlijke wijze voor om te evalueren. We leren door de een populatie van leerstructuren op te starten en voor elke expressie zijn fitheid

bij te houden. De fitheid van elke leerstructuur gebruiken we in de selectie van de volgende populatie.

We gebruiken de term expressie en expressieboom door mekaar om hetzelfde aan te duiden.

4.8.2 Fitheidsfunctie

Hoe bepalen we nu de fitheid van een expressie? We weten de uitkomst van de expressie over de databases in de voorbeelden. We kunnen elk individu evalueren over de databases uit de voorbeelden. Het probleem is dus herleid tot de het bepalen van de gelijkenis van twee relaties. Namelijk de relatie die de oplossing vormt, en de relatie die wordt bekomen door een expressie uit de populatie te evalueren.

Om de gelijkenis tussen twee relaties te bepalen gebruiken we het *symmetrisch verschil*, en normaliseren dit. Zij R_1 en R_2 relaties. Het symmetrisch verschil tussen R_1 en R_2 is de som van het aantal tupels dat in in R_1 voorkomt en niet in R_2, en het aantal tupels dat in R_2 voorkomt en niet in R_1. M.a.w. $\#(R_1 - R_2) + \#(R_2 - R_1)$, waarbij # de functie is die het aantal tupels in een verzameling teruggeeft. We normaliseren het symmetrisch verschil door te delen door het aantal tupels in de oplossing.

Om GP af te wenden van al te simplistische expressies, en onnodig complexe expressie te vermijden, laten we de lengte van de oplossing ook een rol spelen in de fitheid. Expressies die bestaan uit een relatie worden bestraft door hun fitheid te verdubbelen en er nog een constante bij op te tellen. Een hele relatie is namelijk geen interessant structureel verband, noch binnen een relatie, noch tussen relaties. Als een expressie minstens 2 niveaus telt, wordt de diepte gedeeld door tien en bij de fitheid opgeteld.

4.8.3 Genetische Operatoren

Tot slot beschrijven we nog welke Genetische operatoren we hebben geïmplementeerd.

Operatoren

Het grootste probleem bij de implementatie vormt de mutatie. Om te muteren moeten we voor de selectie en de cylindrificatie een willekeurig attribuut kiezen.

De genetische operator encapsulatie is wel beschreven in hoofdstuk 3 maar is uiteindelijk niet geïmplementeerd. We komen hierop terug in de conclusie, zie hoofdstuk 6.

Willekeurig attribuut

In het Genetisch Programmeren komt het soms voor dat we een willekeurige attribuut moeten kiezen, bijvoorbeeld als we mutatie uitvoeren, of de initiële populatie

construeren. Dit komt voor waneer we een willekeurige operator kiezen uit de verzameling van operatoren. Als we bijvoorbeeld selectie kiezen, dan moeten we nog bepalen tussen welke twee attribuutnamen we een gelijkheid willen laten gelden. Aangezien we op voorhand de ariteit van de database weten, selecteren willekeurig twee verschillende attribuutnamen. Als we cylindrificatie selecteren, kiezen we slechts één willekeurige attribuutnaam.

4.8.4 Selectieve Evaluatie

Het algoritme is traag. We hebben deze techniek geïmplementeerd om het enigszins te versnellen. In hoofdstuk 6 beschrijven nog een paar andere technieken om het algoritme te versnellen, die niet geïmplementeerd zijn.

Merk op dat na enkele generaties er groepjes ontstaan van identieke individuen. Dit fenomeen is te verklaren doordat een fit individu een grote kans heeft om meermaals geselecteerd te worden tijdens de reproductie. Selectieve evaluatie is gebaseerd op deze observatie en het feit dat we voor elk zulk groepje maar één individu effectief moeten evalueren. De andere individuen in zo'n groepje hebben exact dezelfde fitheid.

Hoofdstuk 5

Software en Experimenten

In dit hoofdstuk rapporteren we de resultaten van enkele experimenten die tot doel hebben na te gaan of Genetisch Programmeren in staat is om onze doelstelling te bereiken.

Eerst en vooral hebben we software nodig die GP implementeert op de cylindrifische algebra. Hoe deze software is opgebouwd wordt beschreven in sectie 5.1. Daarna (sectie 5.2) beschrijven we elk experiment bondig en de resultaten worden besproken.

5.1 Software

De bedoeling van deze sectie is om in het kort enkele dingen over de software die we geschreven hebben uit de doeken te doen. We beginnen met te beschrijven hoe de software in grote lijnen geconstrueerd werd, en welke technieken we hebben gebruikt. Vervolgens bekijken we hoe de software kan geconfigureerd worden. Ten slotte leggen we uit hoe de voorbeelden worden opgeslagen en gebruikt in de software.

5.1.1 Opbouw

De software die gecreëerd is voor dit eindwerk draagt de naam *Genetic Relation Programmer*, of kortweg GRP. GRP is geschreven in Java en maakt gebruik van de Xerces SAX parser[1], om de input te verwerken. Buiten Genetisch Programmeren, bevat GRP ook twee tools: *TestRunner* en *Generator*. TestRunner leest een configuratiebestand in, waarin een reeks testen worden beschreven, en runt deze reeks automatisch. Dit is dus een handige tool om grotere experimenten op te zetten. Het is ook belangrijk dat de oplossing die we geven voor elke database in de voorbeelden ook juist is. Hiervoor is de tool Generator geschreven die gegeven een database en een expressie een xml bestand maakt van de oplossing.

GRP bestaat uit 3 packages. Namelijk het package Evaluator, het package Programmer en het package GRP.

[1] http://xerces.apache.org

Evaluator bevat alle klassen voor het opstellen en evalueren van een expressieboom. Belangrijk in dit package zijn de *Evaluators* en *FitnessMeasures*. De eerste zijn klassen die een score plakken op de gelijkheid, het verschil of een combinatie van beide, van twee relaties. De tweede zijn klassen die de scores van de Evaluators manipuleren. De verschillende FitnessMeasures zijn beschreven in hoofdstuk 3 onder Generische vorm, in het puntje over fitheid functies. De klassen die een expressie evalueren zijn ad hoc geschreven. Er worden een aantal verbeteringen voorgesteld in hoofdstuk 6.

Programmer bevat de klassen die zich bezig houden met Genetisch Programmeren. Dit package is onderverdeeld in 3 packages: Creation, Operation en Selection. Het package Creation bevat de klassen die "willekeurige" bomen genereren. Het package Selection bevat klassen om uit een populatie (een ExpressionTree array en een double array) een individu te selecteren. De rang selectie is niet geïmplementeerd, omdat we al goed resultaten hadden geboekt met de toernooi en proportionele selectie. Het derde package, Operation, bevat de verschillende genetisch operatoren. Zoals eerder vermeld is encapsulatie niet geïmplementeerd. Het Programmer package bevat ook nog de klasse PopulationImpl. Deze klasse implementeert het effectieve algoritme, en houdt de populatie bij. Ten slotte zit er nog een klasse in Programmer die statistieken over de populatie bijhoudt, zoals de gemiddelde fitheid, het beste individu, ...

Het package GRP bevat de TestRunner tool en een klasse die ervoor zorgt dat GP zijn werk kan doen.

5.1.2 Configuratie

De software is volledig configureerbaar. Alle parameters van het GP en welke algoritmes we willen gebruiken voor bepaalde deeltaken van het GP worden ingesteld in het configuratiebestand. Configuratie bestanden bevatten regels van de vorm: key=value. Zie bijvoorbeeld figuur 5.1.

GRP

Merk op in figuur 5.1 dat Programmer.Creation.RampedCreator een Java klasse aanduidt. We beschikken over verschillende selectiemethoden, verschillende fitheidfuncties en verschillende methodes om een random populatie aan te maken. Elk van deze opties is geïmplementeerd in een aparte klasse. Om GP uit te voeren moeten we een keuze maken uit deze opties en dit doorgeven aan het algoritme. Het configuratiebestand beschrijft onze keuze a.d.h.v. klassenamen. We maken in de software gebruik van reflectie om a.d.h.v. de naam de klasse, de klasse zelf te instantiëren.

5.1.3 Databases en Oplossingen

Eens het algoritme geconfigureerd is, moeten we nog beschikken over voorbeelden. De voorbeelden bestaan uit databases en voor elke database een oplossing. Data-

```
###
# parameters voor GP
###
reproduction=0.40
crossover=0.55
mutatie=0.04
encapsulatie=0.01

populationsize=2000
generations=100


###
# bouwstenen GP: welke evaluator, selector,
# fitnessmeasure en creator?
##
# hoe maken we de random populatie?
creator=Programmer.Creation.RampedCreator
creator.depth=5
```

Figuur 5.1: Een stuk configuratie bestand

bases en oplossingen worden opgeslagen in een xml bestand. Deze xml bestanden worden ingelezen m.b.v. de Xerces SAX parser.

5.2 Experimenten

Deze sectie beschrijven we alle experimenten. Elk experiment is opgenomen in een afzonderlijke deelsectie. Voor elk experiment beschrijven we de testcases, die we gebruiken om de de fitheid van een individu te berekenen, hoe de evolutie verliep en wat de resultaten zijn van het experiment. We beginnen met een aantal begrippen die in de bespreking van de experimenten worden gebruikt.

De experimenten beginnen zeer eenvoudig, met zeer triviale queries. Daarna proberen we een echt structureel verband bloot te leggen.

5.2.1 Begrippen

Diepte wordt hier geïnterpreteerd als het aantal knopen op het langste pad van wortel tot blad. De conventionele betekenis van diepte, het aantal bogen op het langste pad, wordt verkregen door één af te trekken van de hier gegeven diepte.

Aantal generaties tot oplossing is het aantal generaties dat GP nodig heeft om een oplossing (niet noodzakelijk de optimale) te vinden, en deze vast te houden

tot het einde van de run.

5.2.2 Experiment 1: $R \cap S$

Vraagstelling

In dit experiment proberen we na te gaan of GRP überhaupt wel in staat is om een eenvoudige expressie te ontdekken, die evenzeer door een blind random search kan gevonden worden. Voor dit doel gebruiken we een expressie met diepte twee, $R \cap S$.

Testopstelling

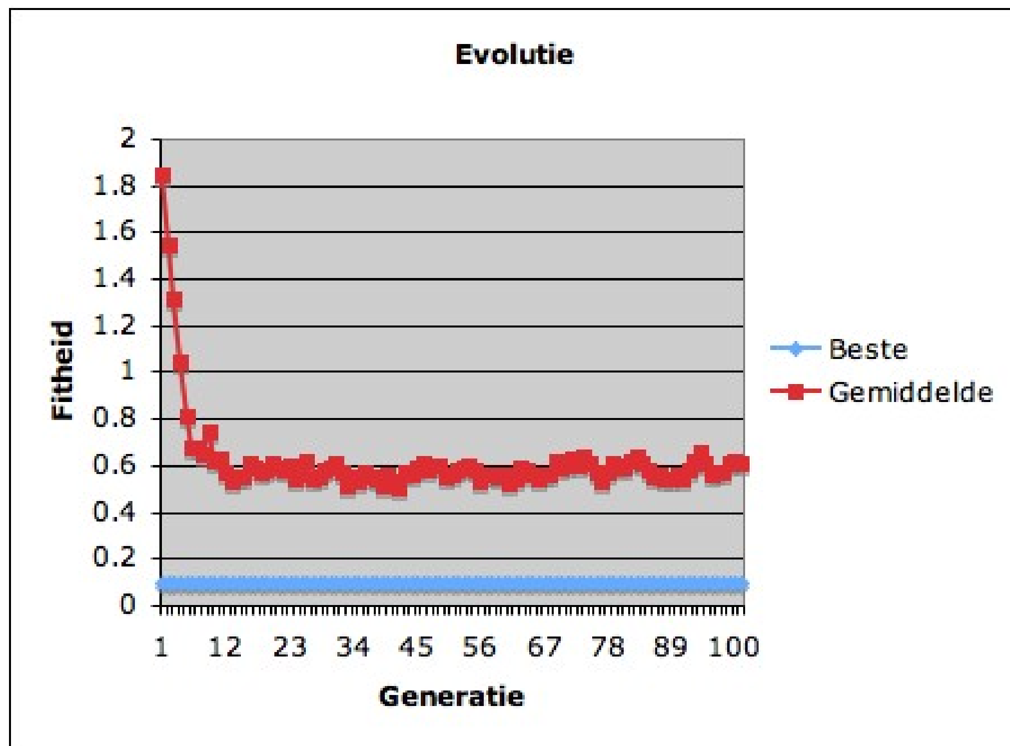
Voor dit experimenten gebruikten we de twee databases weergegeven in tabel 5.1, samen met de oplossing. De parameters die we gebruikten zijn opgenomen in 5.2. GRP werd 20 maal uitgevoerd op deze opstelling.

Resultaat

GRP slaagde in 100% van de gevallen in het ontdekken van de query. In figuur 5.2 zien we hoe de evolutie verloopt a.d.h.v. de gemiddelde fitheid van de populatie (bovenste lijn), en de fitheid van het beste individu (onderste lijn). Het valt onmiddellijk op dat de juiste oplossing al in de eerste generatie wordt gevonden. Dit probleem is dus zo eenvoudig dat we zelfs geen genetisch programmeren nodig hebben om een oplossing te vinden. Uit de gemiddelde populatiefitheid kunnen we afleiden dat er in de eerste 5 generaties nog redelijk wat inefficiënte oplossingen aanwezig zijn in de populatie. Na 10 generaties schommelt de gemiddelde populatie fitheid nog wat, maar er wordt geen substantiële vooruitgang meer geboekt. De schommeling wijst er op dat GP nog steeds bezig is met nieuwe oplossingen te zoeken, en daarbij soms meer en soms minder succes heeft.

Conclusie

We gebruiken een RampedCreator, wat betekent dat als de populatie groot genoeg is, we alle expressies van diepte twee al in de random populatie hebben. Met twee relaties, en 5 operators, zijn er 14 mogelijke expressie te vormen met diepte twee (zes verschillende met de unaire operatoren en 8 verschillende met de binaire operatoren). Aangezien we beschikken over een populatie met 300 individuen, worden er 150 individuen aangemaakt met de FullCreator. Als maximale diepte gaven we vier. Dit betekent dat er 50 individuen worden gecreëerd met diepte twee. $R \cap S$ wordt voorgesteld door twee bomen (de relaties omdraaien verandert niets aan het resultaat). Dit leidt tot een kans van $\frac{1}{7}$ dat we de juiste boom creëren. De kans dat we de juiste boom niet aanmaken is een binomiaal kans: $\binom{50}{50} \times (\frac{1}{7})^0 \times (\frac{6}{7})^{50} = (\frac{6}{7})^{50} = 4.494 \times 10^{-4}$. We hebben m.a.w. een bijzonder kleine kans dat de juiste oplossing niet van de eerste generatie in de populatie aanwezig is. Tevens bestaat de kans ook nog dat de GrowCreator de juiste oplossing maakt.



Figuur 5.2: Typische evolutie van experiment 1

Het feit dat de oplossing gevonden wordt voor dat we Genetisch Programmeren is ook te merken aan het gemiddeld aantal generaties dat nodig is om de oplossing te vinden, nl. 0.05 met een standaarddeviatie van 0.224. In het experiment was er 1 run van GP waarin de oplossing niet in de random populatie aanwezig was.

5.2.3 Experiment 2: $(R \cup S) \cap T$

Vraagstelling

In dit experiment proberen we een iets complexere, doch nog steeds eenvoudige, query te evolueren. De kans dat we in de random populatie meteen de juiste oplossing hebben wordt nu veel kleiner. Er zijn namelijk 1863 bomen die we kunnen construeren met de operatoren en drie relaties. Als we een populatie van 1863 bomen hebben, en de FullCreator gebruiken, hebben we nog slechts een kans van $\frac{4}{621}$, dat de expressie, of de equivalente expressie $(R \cap T) \cup (S \cap T)$ aanwezig is in de populatie.

Het is onrealistisch om te geloven in deze kans, want nu beschikken we over extra kennis. In het algemeen weten we niet hoe diep een expressie moet zijn, dus moeten we een veel hogere diepte instellen om zeker te zijn dat we bomen creëren die groot genoeg zijn. We kunnen dit ook overlaten aan crossover, alhoewel het dan veel langer zal duren om bomen van de juiste diepte te verkrijgen.

R

A	B
1	1
2	1
3	2
3	3

S

B	C
2	2
2	5
3	3
4	1
5	4

Oplossing

A	B	C
3	2	2
3	2	5
3	3	3

R

A	B
2	1
3	1
4	2
4	3

S

B	C
2	1
2	3
3	3
4	4
5	4

Oplossing

A	B	C
4	2	1
4	2	3
4	3	3

Tabel 5.1: Databases voor experiment 1

Parameter	**Waarde**
Reproductie	0.40
Crossover	0.55
Mutatie	0.05
Populatiegrootte	300
Generaties	100
Creator	RampedCreator
Diepte	4
Selector	Proportional
FitnessMeasure	Raw

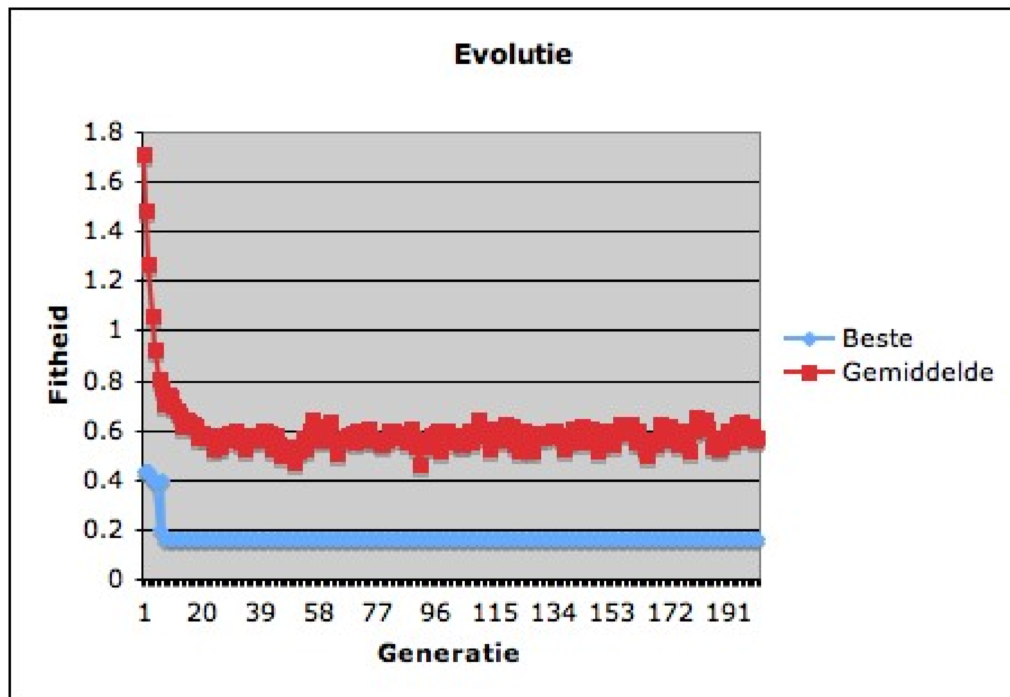
Tabel 5.2: Parameters voor experiment 1

Testopsteling

De gebruikte parameters zijn opgesomd in tabel 5.4. De gebruikte databases zijn opgenomen in tabel 5.3. Dit GRP werd 20 maal uitgevoerd.

Resultaat

In 80% van de gevallen werd de juiste oplossing gevonden. De evolutie, zie grafiek 5.3, verloopt gedeeltelijk zoals de evolutie in experiment 1. We merken echter op dat de juist oplossing “pas” na 7 generatie wordt bereikt. Eens deze is gevonden, blijft GP vruchteloos verder zoeken naar een beter oplossing.



Figuur 5.3: Typische evolutie van experiment 2

Conclusie

We kunnen besluiten dat een al iets complexere query met twee operatoren zeer snel gevonden wordt. In ons experiment gebruikten we 200 generaties, maar 100 generaties had waarschijnlijk volstaan. Gemiddeld werd de oplossing gevonden na 25.65 generaties, met een standaard deviatie van 52.592.

5.2.4 Experiment 3: Structurele query

Vraagstelling

Zoals uitgelegd in het hoofdstuk over Databases, hoofdstuk 4, zijn we geïnteresseerd in structurele verbanden tussen meerdere relaties of in relaties. We werken met database schema $\mathbb{S}(T) = \{A, B\}$, en $relaties(\mathbb{S}) = \{T\}$. In dit experiment willen we GP de query laten zoeken die alle koppels selecteren waarvan de B-waarde eveneens in de A-kolom voorkomt. We bedoelen niet $\sigma_{A=B}(t)$.

Een voorbeelddatabase is opgenomen in Testopstelling.

Laten we eerst eens kijken naar hoe deze query er kan uitzien in relationele algebra en in de cylindrische algebra. De RA expressie $R \bowtie \rho_{A\to B}(\pi_A(R))$ lost het probleem op. De CA expressie $R \cap cyl_A(\sigma_{A=B}(cyl_B(R)))$ is equivalent aan de RA expressie en selecteert bijgevolg ook alle koppels met B-waarde die in de A-kolom voorkomen. Merk op dat we volgens de constructie in het equivalentiebewijs in hoofdstuk 4 een cyl_B zijn vergeten. Zowel de projectie als de renaming

R

A	B
1	1
2	1
3	2

S

B	C
2	2
2	5
3	3

T

A	B	C
1	1	1
6	7	8
2	4	5
3	3	3
8	3	9

Oplossing

A	B	C
1	1	1
3	3	3

R

A	B
1	1
5	9
8	3

S

B	C
6	2
2	5
3	9

T

A	B	C
1	1	1
2	4	5
3	3	3
8	3	9
8	6	1

Oplossing

A	B	C
1	1	1
8	3	9

Tabel 5.3: Databases voor experiment 2

Parameter	**Waarde**
Reproductie	0.40
Crossover	0.55
Mutatie	0.05
Populatiegrootte	300
Generaties	200
Creator	RampedCreator
Diepte	3
Selector	Tournament
FitnessMeasure	Raw

Tabel 5.4: Parameters voor experiment 2

vragen een cylindrificatie van B. Maar tweemaal na mekaar hetzelfde attribuut cylindrificeren is gelijk aan eenmaal dat attribuut cylindrificeren.

Dit verband heeft toepassingen in grafen. We kunnen een graaf voorstellen in een relatie door elke knoop af te beelden op een nummer en elke boog tussen twee knopen voor te stellen door een koppel (A, B), waarbij A de knoop is waaruit een boog vertrekt naar B. Het verband of concept waarvoor we in dit experiment een query zoeken is: “de knopen met minstens één inkomende en minstens één uitgaande boog”.

A	B
1	4
2	5
3	6
4	7
5	8
6	9
1	7
2	8
3	9
4	10
5	11
6	12

Tabel 5.5: Voorbeeld database voor experiment 3

Parameter	**Waarde**
Reproductie	0.40
Crossover	0.55
Mutatie	0.05
Populatiegrootte	400
Generaties	75
Creator	RampedCreator
Diepte	5
Selector	Proportioneel
FitnessMeasure	Raw

Tabel 5.6: Parameters voor experiment 3

Testopstelling

In tabel 5.5 is een voorbeeld database opgenomen. In dit experiment zijn tien gelijkaardige databases gebruikt, van variërende grootte. Tabel 5.6 somt de gebruikte parameter instellingen op.

Resultaat

Na 75 generaties was dit het beste individu:

```
Intersection:
-Union:
--Cylindrification: A
---Leaf: R
--Leaf: R
```

```
-Intersection:
--Leaf: R
--Cylindrification: A
---Selection: A = B
----Cylindrification: B
-----Leaf: R
```

Merk op dat de eerste intersectie en alles wat onder de unie valt weg kan gelaten worden zonder dat de semantiek van de expressie verandert.

Conclusie

Onze software is dus wel degelijk in staat om structurele queries op te stellen. Het grootste probleem is het geheugen verbruik en de uitvoertijd. Deze zijn buiten proporties. Er was bijna 2 GB RAM nodig, en het algoritme heeft ±18 uur uitgevoerd. In hoofdstuk 6 stellen we een aantal verbeteringen voor om zowel het geheugengebruik als de uitvoertijd terug te dringen.

Hoofdstuk 6

Conclusie en Verder Onderzoek

In de eerste hoofdstukken hebben we Machine Learning, Genetische Algoritmen en Genetisch Programmeren geïntroduceerd. In hoofdstuk 4 hebben we databases geformaliseerd en hebben we aangetoond dat we even goed met de cylindrische algebra kunnen werken, omdat deze equivalent is met de relationele algebra maar minder nadelen heeft als het aankomt op genetisch programmeren. Vervolgens hebben we deze ideeën samengevoegd in software, experimenten bedacht en bekeken of de voorgestelde ideeën werkten.

Dit hoofdstuk trekt conclusies uit de vorige hoofdstukken. Voorts behandelen we nog vragen die onbeantwoord zijn gebleven in deze tekst, en we stellen enkele technieken voor om de software te versnellen.

6.1 Conclusie

Hoe goed het algoritme werkt is in grote mate afhankelijk van de voorbeelden die het als input krijgt. Dit hebben we in de praktijk gemerkt. We gebruikten in de experimenten veel en zeer expliciete voorbeelden. Als we dit niet deden, kregen we steevast een meer simplistische oplossing. De voorbeelden, van bijvoorbeeld experiment 3, bevatten maar 1 verband tussen A en B, nl. het verband dat we zochten.

Als we beschikken over te weinig voorbeelden vallen Machine Learning technieken gemakkelijk ten prooi aan *overfitting*. Een leeralgoritme heeft te lijden onder overfitting als het slechts een gedeeltelijke of foutieve veralgemening maakt uit de gegeven voorbeelden.

Er bestaan ook verbanden waarvoor het praktisch onmogelijk is om goede voorbeelden te verzinnen. Stel bv. dat we het verband willen vatten dat twee relaties disjunct zijn. We zouden database instanties geven met die twee relaties in, en als oplossing steeds de lege verzameling geven. Er zijn echter meer expressies die altijd de lege relatie als resultaat hebben, bv. $R \cap R^c$.

Eens we op het punt zijn gekomen dat we beschikken over voldoende goede voorbeelden, lijkt het algoritme te werken. M.a.w. het doel wordt bereikt, de gezochte query wordt gevonden. Het algoritme vindt niet noodzakelijk de query die

wij in gedachte hadden, maar vond in de experimenten wel een equivalente query.

We mogen besluiten dat het mogelijk is om m.b.v. Genetisch Programmeren Multirelationele Data Mining uit te voeren. Bijgevolg zijn we geslaagd in ons opzet.

6.2 Verder Onderzoek

We zijn dan wel geslaagd in ons opzet, een oplossing brengt ook nieuwe vragen met zich mee. In deze sectie beschouwen we enkele van de vragen die we niet hebben beantwoord in deze tekst. Tijdens het experimenteren met de software en schrijven van deze tekst, zijn er ook nog vragen gerezen. Deze vragen en problemen bundelen we nog in deze sectie. Voor sommige problemen stellen we ook een oplossing voor.

6.2.1 Vragen

We kunnen twee grote categorieën van vragen onderscheiden. Ten eerste vragen we ons af of dit algoritme kan uitgebreid worden naar andere datastructuren, en welke problemen deze uitbreidingen met zich zouden meebrengen. Ten tweede hebben we opgemerkt, tijdens de experimenten, dat er allerlei nutteloze expressies opduiken met redundanties. We vragen ons dus af in hoeverre we de populatie moeten controleren en manipuleren, dan wel het algoritme de volledige vrijheid geven.

a. Uitbreiding

In deze tekst hebben we alleen relaties met getallen toegelaten. We kunnen ons afvragen of het mogelijk is om te werken met andere datastructuren. Aangezien we alleen met structurele queries werken, is onze grootste bekommernis om, op een zinnige manier, gelijkheid te definiëren op de nieuwe datastructuren. Als we met een mengeling van datastructuren werken (zoals dit gebeurt in de meeste hedendaagse databases) moeten we ofwel tussen datastructuren een gelijkheid kunnen definiëren, ofwel het algoritme uitbreiden om selecties tussen verschillende datatypes te vermijden.

b. Populatie controle

Encapsulatie De Genetische Operatie encapsulatie wordt beschreven in hoofdstuk 3, maar is niet geïmplementeerd in de software. We vragen ons echter af of deze operatie ons sneller tot een goed resultaat zou brengen. We moeten ons dan ook afvragen hoe we dit best aanpakken. Willekeurig deelbomen selecteren levert waarschijnlijk weinig nuttige resultaten op. Alle mogelijke beschikbare deelbomen beschouwen en diegene die het meeste voorkomt selecteren vergt waarschijnlijk te

veel rekenwerk. Er is dus nog veel ruimte om heuristieken te onderzoeken. Mogelijk is *frequent subtree mining* hier aan de orde.

c. Intelligentere Pruning In hoofdstuk 4 beschreven we een simpele vorm van pruning op de individuen. Dit zou echter nog uitgebreid kunnen worden. Laten we naar een fragment kijken uit een boom die gegenereerd werd voor experiment 3.

```
---------Cylindrification: A
----------Selection: A = B
-----------Cylindrification: A
------------Selection: A = B
-------------Cylindrification: B
--------------Cylindrification: A
---------------Selection: A = B
----------------Cylindrification: B
-----------------Leaf: R
```

Als we onderaan beginnen, zien we dat er eerst een cylindrificatie wordt uitgevoerd op B, gevolgd door een selectie, gevolgd door een cylindrificatie van A en nogmaals een cylindrificatie van B. Dit is equivalent aan alleen het uitvoeren van de twee laatste cylindrificaties. Tevens zien we ook dat de eerste vier operaties tweemaal hetzelfde zijn. Deze vier operaties kunnen dus vervangen worden door de eerste twee.

Pruning kunnen we beschouwen als een nieuwe genetische operatie. Toch kunnen we ons afvragen hoever we kunnen gaan met operaties zoals het prunen van bomen, of het vervangen van bomen (stukken van bomen) door meer eenvoudige equivalente expressies. Mogelijk bevooroordelen we het algoritme dan al te fel naar een eenvoudige expressie, en is het niet meer in staat om degelijke veralgemeningen uit te voeren op complexere verbanden.

Langs de andere kant vullen zulke redundanties ook alleen maar geheugen.

Of volstaat het om bomen met zo'n een verschijnsel te bestraffen met een lagere fitheid? Stel dat het volstaat om zulke bomen te bestraffen, dan is het uit puur efficiëntie opzicht waarschijnlijk nog altijd beter als we prunen.

6.2.2 Nieuwe Technieken

Het leeuwendeel van de tijd dat het algoritme uitvoert wordt gespendeerd aan het evalueren van de expressies in de populatie. We beschrijven drie technieken die geïmplementeerd kunnen worden om de evaluatie te versnellen en bijgevolg het algoritme te versnellen.

Lazy Cylindrification en Cylindrifcation-On-The-Fly

De grote boosdoener tijdens het evalueren is het cylindrificeren. Cylindrificeren vraagt exponentieel veel tijd en ruimte in het aantal attributen dat we cylindrificeren.

Een eerste techniek om het effect wat in te perken is *Cylindrification-On-The-Fly*. Cylindrificeer pas als het echt nodig is. Bijvoorbeeld als we de cylindrificatie operator tegenkomen, of als twee relaties niet hetzelfde schema hebben als we de unie nemen, of als er een attribuut ontbreekt in een selectie.

Maar we kunnen nog luier worden. *Lazy Cylindrification* gaat uit van het feit dat een cylindrificatie pas uitgerekend dient te worden op het ogenblik dat er in een gecylindrificeerde kolom iets wordt bijgevoegd dan wel verwijderd. Dit betekent, dat de cylindrificatie operator slechts aangeeft in een relatie dat een attribuut gecylindrificeerd is, zonder daadwerkelijk de cylindrificatie uit te rekenen. De andere operatoren kunnen nu op een intelligente manier omspringen met deze kennis en in bepaalde gevallen de cylindrificatie volledig vermijden. Laten we een aantal voorbeelden bekijken van hoe we effeciënter kunnen evalueren als we Lazy Cylindrification toepassen. In de voorbeelden wordt een intuïtieve uitleg gegeven, het moet echter ook mogelijk zijn om dit te bewijzen als deze techniek geïmplementeerd wordt.

Voorbeeld 1: Stel dat we een selectie ($A = B$) uitvoeren. Er kunnen zich een aantal gevallen voordoen i.v.m. de attributen A en B:

- Geen van beide is gecylindrificeerd: Per tupel gaan we na of $A = B$.
- Eén van beide is gecylindrificeerd: Dan kopiëren we de inhoud van de niet-gecylindrificeerde kolom naar de gecylindrificeerde kolom.
- Beide zijn gecylindrificeerd: Een van beide attributen dient gecylindrificeerd te worden, zodat we terug in het vorig geval belanden.

□

Voorbeeld 2: Stel dat we de unie van twee relaties R en S willen berekenen. Stel dat R attributen A en C bevat, en S attributen B en C bevat. Stel dat attribuut C in beide relaties gecylindrificeerd is.

In R zullen we de cylindrificatie van B moeten uitrekenen en in S zullen we de cylindrificatie van A moeten uitrekenen. De cylindrificatie van C kunnen we in beide relaties achterwege laten, omdat deze voor elk tupel in R en S alle waarden heeft. □

Een nadeel van deze technieken is, dat als veel cylindrificaties moeten uitgevoerd worden deze technieken waarschijnlijk trager gaan werken dan wanneer we alle database instanties op voorhand uniformeren en uitbreiden.

Multi-Threading

Theoretisch gezien verandert dit niets aan de complexiteit van het algoritme. Maar in multi-processor en multi-core systemen kan het concurrent evalueren van individuen een praktische tijdswinst opleveren.

Dit heeft als nadeel dat de software complexer zal worden omdat deze nu rekening moet houden met concurrente threads.

Bibliografie

[1] A.C. Acar and A. Motro. Intensional encapsulations of database subsets via genetic programming. *Lecture Notes in Computer Science*, 3588:365–374, 2005.

[2] N. A. Barricelli. Esempi numerici di processi di evoluzione. *Methodos*, pages 45–68, 1954.

[3] N.L. Cramer. A representation for the adaptive generation of simple sequential programs. In *Proceedings of the 1st International Conference on Genetic Algorithms*, pages 183–187. Lawrence Erlbaum Associates, 1985.

[4] C.R. Darwin. *The Origin of Species by Means of Natural Selection.* H. M. Caldwell, 1878.

[5] S Džeroski. Multi-relational data mining: An introduction. *SIGKDD Explorer Newsletter*, 5(1):1–16, 2003.

[6] C. Ferreira. Gene expression programming: a new adaptive algorithm for solving problems. *Complex Systems*, 13:87, 2001.

[7] L. Henkin, J.D. Monk, and A. Tarski. *Cylindric Algebras. Part I.* North-Holland, 1971.

[8] L. Henkin, J.D. Monk, and A. Tarski. *Cylindric Algebras. Part II.* North-Holland, 1985.

[9] L. Henkin, J.D. Monk, A. Tarski, H. Andréka, and I. Németi. *Cylindric Set Algebras*, volume 883 of *Lecture Notes in Mathematics*. Springer-Verlag, 1981.

[10] L. Henkin and A. Tarski. Cylindric algebras. In R.P. Dilworth, editor, *Lattice Theory*, volume 2 of *Proceedings of Symposia in Pure Mathematics*, pages 83–113. American Mathematical Society, 1961.

[11] J. Holland. *Adaptation In Natural and Artificial Systems.* The University of Michigan Press, 1975.

[12] J. Holland. *Adaptation in Natural and Artificial Systems.* The University of Michigan Press, 1992.

[13] T. Imielinski and W. Lipski. The relational model of data and cylindric algebras. *JCSS*, 28:80–102, 1984.

[14] D.B. Kell. Genotype-phenotype mapping: genes as computer programs. *Trends in Genetics*, 18(11):555–559, 2002.

[15] J.R. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection.* The MIT Press, Cambridge, Massachusetts, 1992.

[16] W.B. Langdon and R. Poli. *Foundations of Genetic Programming.* Springer-Verlag, 2002.

[17] F. Menczer and R. K. Belew. Adaptive retrieval agents: Internalizing local context and scaling up to the web. *Machine Learning*, 39(2-3):203–224, 2000.

[18] F. Menczer, G. Pant, and P. Srinivasan. Topical web crawlers: Evaluating adaptive algorithms. *ACM Transactions on Internet Technology*, 4(4):378–419, 2004.

[19] T.M. Mitchell. *Machine Learning.* McGraw-Hill Companies, Inc., Singapore, 1997.

[20] S. Muggleton and L. De Raedt. Inductive logic programming: Theory and methods. *Journal of Logic Programming*, 19/20:629–679, 1994.

[21] R. Poli and W.B. Langdon. Genetic programming with one-point crossover. In P.K. Chawdhry, R. Roy, and R.K. Pant, editors, *Soft Computing in Engineering Design and Manufacturing*, pages 180–189. Springer-Verlag London, 1997.

[22] G.R. Price. Selection and covariance. *Nature*, 227:520–521, 1970.

[23] N.J. Radcliffe. Schema processing. In Thomas Bck, David B. Fogel, and Zbigniew Michalewicz, editors, *Handbook of Evolutionary Computing*, pages B2.5–1–10. Oxford University Press, 1997.

[24] J.P. Rosca. Analysis of complexity drift in genetic programming. In J.R. Koza, K. Deb, M. Dorigo, D.B. Fogel, M.H. Garzon, H. Iba, and R.L. Riolo, editors, *Genetic Programming 1997: Proceedings of the Second Annual Conference*, pages 286–294, Stanford University, 1997. Morgan Kaufmann.

[25] T. Ryu and C. Eick. Deriving queries from examples using genetic programming. In *Proc. Second Int. Conference on Knowledge Discovery and Data Mining*, Portland, Oregon, 1996.

[26] S.F. Smith. *A learning system based on genetic adaptive algorithms.* PhD thesis, University of Pittsburgh, 1980.

[27] C.R. Stephens and H. Waelbroeck. Effective degress of freedom in genetic algorithms and the block hypothesis. In Thomas Bck, editor, *Proceedings of the Seventh International Conference on Genetic Algorithms (ICGA97)*, pages 34–40. Morgan Kaufmann, 1997.

[28] C.R. Stephens and H. Waelbroeck. Schemata evolution and building blocks. *Evolutionary Computation*, 109, 1999.

[29] J. Van den Bussche. Applications of Alfred Tarski's ideas in database theory. In L. Fribourg, editor, *Computer Science Logic*, volume 2142 of *Lecture Notes in Computer Science*. Springer, 2001.

[30] Wikipedia. Genetic algorithm.