

Smart Schema Validator

Peter Boyen

promotor :
Prof. dr. Frank NEVEN

Samenvatting

XML (eXtensible Markup Language) is de meest gebruikte taal voor het uitwisselen van data op het internet. Het is de W3C standaard voor het uitwisselen van informatie over het internet. Een XML-document voldoet gewoonlijk aan een bepaalde structuur, opgelegd door een XML-schema. De meest populaire schema's zijn DTD en XML Schema.

DTD's en XML Schema's zijn om te zetten in een grammatica met reguliere expressies. Deze vereisen echter dat deze expressies *one-unambiguous* zijn. Bij XML Schema noemt men deze vereiste Unique Particle Attribution (UPA).

In deze thesis onderzoeken we een aantal methodes. We bestuderen een methode om na te gaan of een reguliere expressie *one-unambiguous* is, een methode om na te gaan of de taal van een expressie *one-unambiguous* is en een methode om reguliere expressies om te zetten in een equivalente *one-unambiguous* reguliere expressie als er een dergelijke expressie bestaat. We zouden deze methodes willen gebruiken in een smart schema validator, zodat die een alternatief kan aanbieden als er een fout tegen UPA gemaakt wordt.

Praktische tests op een uitgebreide verzameling van willekeurig gegenereerde reguliere expressies tonen aan dat het algoritme in de praktijk bruikbaar is qua uitvoeringstijd. Een groot deel van de onderzochte expressies had echter geen *one-unambiguous* equivalent.

Om dat de expressies automatisch gegenereerd worden is het resultaat van het algoritme vaak onnodig lang. Dit probleem kan zich in het algemeen voordoen bij automatisch gegenereerde expressies. Daarom hebben we besloten ook methodes te onderzoeken die een expressie verkorten.

We hebben enkele variaties van twee algemene methodes bestudeerd: een algoritme verkleinen, met behulp van herschrijfgeregels, of een kleinere expressie genereren uit de minimale deterministische automaat.

In onze tests bleek de methode met behulp van herschrijfgeregels beter te werken, met als extra voordeel dat deze methode de *one-unambiguity* van de originele expressie bewaard.

Woord vooraf

Bij het indienen van deze thesis gaat mijn dankbaarheid uit naar de mensen die mij op één of andere manier geholpen hebben.

Eerst wil ik mijn promotor Prof. Dr. Frank Neven en mijn begeleider Wouter Gelade danken. Zij hebben mij geholpen bij het begrijpen van de stof, het vormgeven van de tekst en de implementatie van de algoritmes. Ook dank ik Dr. Geert Jan Bex voor de inzage in zijn code en het gebruik van het iDTD algoritme.

Ik ben ook mijn ouders en mijn vriendin Sara dankbaar voor het nalezen van mijn thesis en voor hun steun en vertrouwen.

Inhoudsopgave

1	Inleiding	10
1.1	Verdere indeling van deze thesis	14
1.1.1	Indeling van Deel I	14
1.1.2	Indeling van Deel II	14
2	Definities	15
2.1	Reguliere expressies	15
2.2	One-unambiguous reguliere expressies	16
I	One-unambiguous reguliere expressies	18
3	Omzetting naar one-unambiguous reguliere expressies	19
3.1	Inleiding	19
3.2	Definities	19
3.3	Nagaan of een reguliere expressie one-unambiguous is	21
3.3.1	De invoer omzetten in een syntaxboom	21
3.3.2	De syntaxboom transformeren naar zijn Glushkov automaat	22
3.4	Bepalen of een taal one-unambiguous is	31
3.4.1	De Glushkov automaat determiniseren en minimaliseren	32
3.4.2	Algoritme om te bepalen of een taal one-unambiguous is	33
3.5	Een equivalente one-unambiguous reguliere expressie maken	40
3.6	Voorbeelduitwerking van het hele algoritme	42
3.7	Mogelijke verbeteringen op het algoritme	45

II	Reguliere expressies inkorten	46
4	Kortere reguliere expressies vinden	47
4.1	Inleiding	47
4.2	Naïeve implementatie	47
4.3	Lengtematen	48
5	Herschrijfregels voor reguliere expressies	49
5.1	Inleiding	49
5.2	Axiomasysteem F1	50
5.3	Implementatie	51
5.3.1	Gelijkheid controleren	52
5.3.2	Oplossen van vergelijkingen	52
5.3.3	Toegevoegde regels	53
5.3.4	Unie-operator	54
5.3.5	One-unambiguity bewaren	54
5.3.6	Volgorde van het toepassen van regels	55
6	Kortere expressies bekomen uit de minimale deterministische automaat	56
6.1	Lineaire vergelijkingen	56
6.1.1	Omzetting naar lineaire vergelijkingen	57
6.1.2	Oplossen van lineaire vergelijkingen	58
6.1.3	Volgorde van het oplossen van lineaire vergelijkingen	58
6.2	State elimination	59
6.2.1	Volgorde van het verwijderen van toestanden	60
6.3	Chopping	62
6.3.1	Vertical chopping	63
6.3.2	Horizontal chopping	68
7	Praktische resultaten van methodes	71
7.1	Inleiding	71
7.2	One-unambiguous reguliere expressies	72
7.3	Reguliere expressies inkorten	76
7.4	Het korter maken van one-unambiguous reguliere expressies	84
7.5	Het one-unambiguous zijn van korter gemaakte reguliere expressies	90
8	Gebruik van de implementatie	92

9	Conclusie	94
III	Appendices	98
A	Bewijzen	99
A.1	Stelling 1	99
A.2	Stelling 5	102

Lijst van figuren

3.1	Deterministische automaat of DFA.	20
3.2	Niet-deterministische automaat of NFA.	21
3.3	Syntaxbomen voor: 1) $(a + a^*)b$ en 2) $a(b + c)c$	22
3.4	Glushkov automaat voor de expressie $a^*a + bc$	26
3.5	Minimale deterministische automaat voor $L(a^*a + bc)$	32
3.6	Automaat: 1) met orbitProperty en 2) zonder orbitProperty.	34
3.7	Automaat: 1) met a consistent en 2) de $\{a\}$ -cut van 1).	35
3.8	Orbit automaat voor a_1 uit Figuur 3.6 1).	36
3.9	Bereikbare automaat voor a_1 uit Figuur 3.6 1).	40
3.10	Syntaxboom voor de markering $a_1^*a_2$	43
3.11	Glushkov automaat voor de reguliere expressie a^*a	43
3.12	Gedetermineerde en geminimaliseerde Glushkov automaat voor de reguliere expressie a^*a	44
3.13	De a -cut van de gedetermineerde en geminimaliseerde Glushkov automaat voor de reguliere expressie a^*a	44
3.14	Bereikbare automaat voor a_1 van de a -cut van de gedetermineerde en gemi- nimaliseerde Glushkov automaat voor de reguliere expressie a^*a	45
5.1	Syntaxboom voor $(d + a) + (c + b)$, vervolgens omgevormd naar rechtsdiepe boom, en daarna gesorteerd.	53
6.1	Deterministische automaat.	57
6.2	Deterministische automaat voorbereid voor state elimination.	60
6.3	De automaten waarin eerst toestand X_2 verwijderd is, en vervolgens toestand X_1	60
6.4	De automaten waarin eerst toestand X_1 verwijderd is, en vervolgens toestand X_2	61
6.5	De minimale, deterministische automaat voor a^*b^* met uitbreidingen voor state elimination.	62
6.6	Automaat met bridge state c_1	63

6.7	Twee resultaatautomaten na vertical chopping.	64
6.8	Automaat met min-, anc- en max-waarden.	65
6.9	Automaat waarbij het berekenen van de min-, anc- en max-waarden kan mis- lopen.	66
6.10	Automaat met twee aparte paden.	68
6.11	Twee resultaatautomaten na horizontal chopping.	69
7.1	De lengte van de nieuwe one-unambiguous expressies voor een alfabet met grootte vijf (outliers voor lengte honderd zijn niet allemaal weergegeven, deze gaan tot 6882).	74
7.2	De lengte van de nieuwe one-unambiguous expressies voor een alfabet met grootte vijftientig (outliers voor lengte honderd zijn niet allemaal weergege- ven, deze gaan tot 70694).	74
7.3	De $\log_2$ van de gemiddelde uitvoeringstijd in milliseconden om één expressie om te zetten naar een equivalente one-unambiguous expressie voor een alfabet met grootte vijf.	75
7.4	De $\log_2$ van de gemiddelde uitvoeringstijd in milliseconden om één expressie om te zetten naar een equivalente one-unambiguous expressie voor een alfabet met grootte vijftientig	75
7.5	De lengte na het toepassen van methodes om een expressie in te korten op een willekeurige expressie van lengte tien met een alfabet van grootte vijf (outliers voor state elimination zijn niet allemaal weergegeven, deze gaan tot 56). . . .	78
7.6	De lengte na het toepassen van methodes om een expressie in te korten op een willekeurige expressie van lengte twintig met een alfabet van grootte vijf (outliers voor state elimination zijn niet allemaal weergegeven, deze gaan tot 1418).	79
7.7	De lengte na het toepassen van methodes om een expressie in te korten op een willekeurige expressie van lengte vijftig met een alfabet van grootte vijf (boxplot voor state elimination is niet volledig weergegeven, de outliers gaan tot 744304).	79
7.8	De lengte na het toepassen van methodes om een expressie in te korten op een willekeurige expressie van lengte honderd met een alfabet van grootte vijf (boxplot voor state elimination is niet volledig weergegeven, de outliers gaan tot 1741864).	80
7.9	De lengte na het toepassen van methodes om een expressie in te korten op een willekeurige expressie van lengte vijfhonderd met een alfabet van grootte vijf (outliers voor state elimination zijn niet allemaal weergegeven, deze gaan tot 977470).	80
7.10	De lengte na het toepassen van methodes om een expressie in te korten op een willekeurige expressie van lengte vijftig met een alfabet van grootte vijfen- twintig (boxplot voor state elimination is niet volledig weergegeven, de outliers gaan tot 175752).	81

7.11	De lengte na het toepassen van methodes om een expressie in te korten op een willekeurige expressie van lengte honderd met een alfabet van grootte vijftwintig (boxplot voor state elimination is niet volledig weergegeven, de outliers gaan tot 1830812).	81
7.12	De lengte na het toepassen van methodes om een expressie in te korten op een willekeurige expressie van lengte tweehonderdvijftig met een alfabet van grootte vijftwintig (boxplot voor state elimination is niet weergegeven, waardes lopen tussen 1514 en 1808170).	82
7.13	De lengte na het toepassen van methodes om een expressie in te korten op een willekeurige expressie van lengte vijfhonderd met een alfabet van grootte vijftwintig (boxplot voor state elimination is niet volledig weergegeven, de outliers gaan tot 2020629).	82
7.14	De $\log_2$ van de gemiddelde uitvoeringstijd in milliseconden nodig voor het toepassen van een methode om één willekeurige expressie in te korten met alfabet van grootte vijf.	83
7.15	De $\log_2$ van de gemiddelde uitvoeringstijd in milliseconden nodig voor het toepassen van een methode om één willekeurige expressie in te korten met alfabet van grootte vijftwintig.	83
7.16	De lengte na het toepassen van methodes om een expressie in te korten op een one-unambiguous expressie van lengte tien met een alfabet van grootte vijf.	85
7.17	De lengte na het toepassen van methodes om een expressie in te korten op een one-unambiguous expressie van lengte twintig met een alfabet van grootte vijf.	86
7.18	De lengte na het toepassen van methodes om een expressie in te korten op een one-unambiguous expressie van lengte vijftig met een alfabet van grootte vijf.	86
7.19	De lengte na het toepassen van methodes om een expressie in te korten op een one-unambiguous expressie van lengte honderd met een alfabet van grootte vijf. (Er ontbreekt een outlier voor rewrite, namelijk 1078)	87
7.20	De lengte na het toepassen van methodes om een expressie in te korten op een one-unambiguous expressie van lengte vijfhonderd met een alfabet van grootte vijf.	87
7.21	De lengte na het toepassen van methodes om een expressie in te korten op een one-unambiguous expressie van lengte vijftig met een alfabet van grootte vijftwintig.	88
7.22	De lengte na het toepassen van methodes om een expressie in te korten op een one-unambiguous expressie van lengte honderd met een alfabet van grootte vijftwintig.	88
7.23	De $\log_2$ van de gemiddelde uitvoeringstijd in milliseconden nodig voor het toepassen van een methode om één one-unambiguous expressie in te korten met alfabet van grootte vijf.	89
7.24	De $\log_2$ van de gemiddelde uitvoeringstijd in milliseconden nodig voor het toepassen van een methode om één one-unambiguous expressie in te korten met alfabet van grootte vijftwintig.	89

8.1 Grafische user interface voor de implementatie.	93
---	----

Lijst van tabellen

7.1	Het aantal niet one-unambiguous expressies in willekeurig gegenereerde expressies met een alfabet van grootte vijf.	72
7.2	Het aantal niet one-unambiguous expressies in willekeurig gegenereerde expressies met een alfabet van grootte vijfentwintig.	72
7.3	Het aantal expressies waarvoor geen one-unambiguous tegenhanger bestaat bij een alfabet van grootte vijf.	72
7.4	Het aantal expressies waarvoor geen one-unambiguous tegenhanger bestaat bij een alfabet van grootte vijfentwintig.	73
7.5	Het aantal expressies waarvoor een resultaat kon geleverd worden binnen de vijf minuten door een methode om één willekeurige expressie in te korten met alfabet van grootte vijf.	84
7.6	Het aantal expressies waarvoor een resultaat kon geleverd worden binnen de vijf minuten door een methode om één willekeurige expressie in te korten met alfabet van grootte vijfentwintig.	84
7.7	Het aantal expressies waarvoor een resultaat kon geleverd worden binnen de vijf minuten door een methode om een one-unambiguous expressie in te korten met een alfabet van grootte vijf.	90
7.8	Het aantal expressies waarvoor een resultaat kon geleverd worden binnen de vijf minuten door een methode om een one-unambiguous expressie in te korten met een alfabet van grootte vijfentwintig.	90
7.9	Het aantal niet one-unambiguous expressies na het toepassen van methodes om de expressie in te korten met een alfabet van grootte vijf.	90
7.10	Het aantal niet one-unambiguous expressies na het toepassen van methodes om de expressie in te korten met een alfabet van grootte vijfentwintig.	91

Hoofdstuk 1

Inleiding

XML ([W3C, 2007a]) is de meest gebruikte taal voor het uitwisselen van data op het internet. Het is de W3C standaard voor het uitwisselen van informatie over het internet. Het is de opvolger van SGML en is ontworpen om leesbaar te zijn voor zowel mensen als computers. XML laat de gebruiker toe tags te definiëren die de data opdelen.

Hieronder zien we een voorbeeld XML-bestand dat een deel van de inventaris van een boekhandel voorstelt. Voor elk boek wordt er informatie zoals auteur, genre en titel gegeven. Als een boek aanwezig is in de winkel, wordt het aantal aanwezige boeken weergegeven. Van een boek waarvoor een bestelling geplaatst is, wordt aangegeven hoeveel er besteld zijn.

```
<?xml version="1.0"?>
<inventaris>
  <item>
    <boek>
      <auteur>Terry Pratchett</auteur>
      <genre>Fantasy</genre>
      <genre>Humor</genre>
      <titel>Pyramids</titel>
    </boek>
    <besteld>3</besteld>
  </item>
  <item>
    <boek>
      <auteur>Terry Pratchett</auteur>
      <genre>Fantasy</genre>
      <genre>Humor</genre>
      <titel>The light fantastic</titel>
    </boek>
    <aantal>3</aantal>
  </item>
</inventaris>
```

Om applicaties van dezelfde XML-data gebruik te laten maken, voldoet een XML-bestand gewoonlijk aan een bepaalde structuur, opgelegd door een XML-schema. De schema's leggen een vaste verzameling tags en een vaste structuur op voor XML-documenten die eraan voldoen.

Een dergelijk XML-schema biedt de volgende voordelen ([Bex et al., 2006]):

- het laat automatische verificatie van de documentstructuur toe,
- het geeft een betekenis aan de data,
- het laat optimalisatie van het verwerken van XML-data toe,
- het wordt door verschillende applicaties gebruikt,
- we kunnen de structuur van verschillende documenten begrijpen door één schema te bestuderen.

Er zijn verschillende soorten XML-schema's, maar de meest gebruikte hiervan zijn Document Type Definitions (DTD's) ([W3C, 2007a]), XML Schema ([W3C, 2007b]) en Relax NG ([James Clark and Murata Makoto, 2007]).

We bekijken hier XML Schema als voorbeeld. Een schema gemaakt in XML Schema noemen we een XML Schema definitie en kan zelf weergegeven worden in XML.

Hieronder geven we een deel van een XML Schema definitie voor het hier gebruikte voorbeeld-XML-bestand. Deze XML Schema definitie geeft aan dat een item-tag bestaat uit ofwel een boek-tag en een aantal-tag ofwel een boek-tag en een besteld-tag.

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
            elementFormDefault="qualified">

  <xs:element name="item">
    <xs:complexType>
      <xs:choice>
        <xs:sequence>
          <xs:element ref="boek"/>                (*)
          <xs:element ref="aantal"/>
        </xs:sequence>
        <xs:sequence>
          <xs:element ref="boek"/>                (**)
          <xs:element ref="besteld"/>
        </xs:sequence>
      </xs:choice>
    </xs:complexType>
  </xs:element>

</xs:schema>
```

Deze definitie beschrijft de structuur van het hier gebruikte voorbeeld XML-bestand, maar ze voldoet niet aan de XML Schema standaard, want ze breekt de Unique Particle Attribution (UPA) regel ([Martens et al., 2006]). De UPA-regel zegt dat, telkens we een tag tegenkomen, we onmiddellijk moeten kunnen aangeven bij welk element die tag hoort (zonder naar zijn kinderen of naar later voorkomende tags te kijken). In dit voorbeeld betekent dit het volgende: als we in het XML-bestand een boek-tag tegenkomen, kunnen we niet weten of het over een boek-tag gaat aangegeven met (*) of met (**).

We kunnen deze XML Schema definitie echter wel verbeteren, want we kunnen bij item aangeven dat het moet beginnen met een boek-tag, en dat er dan ofwel een aantal-tag of een besteld-tag moet volgen. In dit geval komt het boek-element maar één keer voor en kan er dus geen verwarring optreden.

Een verbeterd voorbeeld ziet eruit als volgt:

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
            elementFormDefault="qualified">

  <xs:element name="item">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="boek"/>
        <xs:choice>
          <xs:element ref="aantal"/>
          <xs:element ref="besteld"/>
        </xs:choice>
      </xs:sequence>
    </xs:complexType>
  </xs:element>

</xs:schema>
```

Het is mogelijk ([Martens et al., 2006]) deze XML-schema's om te zetten in grammatica's met reguliere expressies (zie paragraaf 2.1), waarbij de UPA-regel dan zou overeenkomen met de eis dat deze reguliere expressies one-unambiguous zijn (zie paragraaf 2.2). Informeel worden one-unambiguous reguliere expressies vaak deterministische reguliere expressies genoemd.

Er zijn twee voorname redenen voor deze beperking ([Martens et al., 2006]):

- one-unambiguous expressies laten efficiënte validatie toe. Deze beperking is hiervoor echter niet noodzakelijk. Ook voor reguliere expressies zonder deze beperking zijn er efficiënte validatie-algoritmes.
- de belangrijkste reden om deze beperking nog te behouden is om compatibiliteit met SGML¹ te bewaren.

¹De voorganger van XML.

Als we onze eerste poging tot XML Schema definitie door een XML-validator laten controleren, zal deze aangeven dat het de UPA-regel schendt. In het hier gebruikte voorbeeld is het vrij eenvoudig om een verbeterde versie te vinden die nog altijd de gewenste mogelijkheden uitdrukt. Als we echter een ingewikkelde XML Schema definitie met onder andere overerving van types hebben, wordt het nagenoeg onmogelijk om precies te bepalen waar de definitie de regel schendt en hoe we dit kunnen oplossen.

Er is een methode om veel, maar niet alle, reguliere expressies om te zetten in one-unambiguous reguliere expressies ([Brüggemann-Klein and Wood, 1998], [Brüggemann-Klein, 1993]). Wat we zouden willen is dat een validator die een overtreding op de UPA-regel detecteert, onmiddellijk één of meer alternatieven aanbiedt die dezelfde structuur beschrijven, maar die wel aan UPA voldoen (indien mogelijk natuurlijk).

In Deel I van deze thesis doorgronden we drie algoritmes: het algoritme om na te gaan of een reguliere expressie al dan niet one-unambiguous is, het algoritme om na te gaan of een taal one-unambiguous is en het algoritme om equivalente one-unambiguous reguliere expressies te maken, zoals die beschreven staan in [Brüggemann-Klein and Wood, 1998] en [Brüggemann-Klein, 1993]. We bespreken ook de implementatie daarvan.

De reguliere expressie $((a \cdot b) + (a \cdot c)) \cdot d$ is niet one-unambiguous: als we in een woord een symbool a tegenkomen, kunnen we niet onmiddellijk zeggen met welke a uit de expressie deze overeenkomt.

Als we het algoritme uit Deel I toepassen, krijgen we een nieuwe expressie die wel one-unambiguous is, namelijk $(\epsilon \cdot (a \cdot (\epsilon \cdot ((c \cdot (\epsilon \cdot (d \cdot \epsilon))) + (b \cdot (\epsilon \cdot (d \cdot \epsilon)))))))$. Het is eenvoudig om hieruit de nutteloze concatenaties met ϵ te verwijderen, en dan krijgen we de expressie $(a \cdot ((c \cdot d) + (b \cdot d)))$.

De expressie $(a \cdot ((c \cdot d) + (b \cdot d)))$ is wel one-unambiguous, maar het symbool d wordt onnodig herhaald. De expressie $(a \cdot (c+b) \cdot d)$ is equivalent met de vorige en is evenzeer one-unambiguous. Deze expressie is ook duidelijk korter. In dit voorbeeld sparen we een kopie van een enkel symbool uit, maar in de plaats van het symbool d kan een subexpressie komen van gelijk welke lengte. In deze subexpressie zouden dan ook kopieën van subexpressies kunnen voorkomen, en daarin ook weer, zodat bepaalde subexpressies vier, acht,... keer gekopieerd worden. Dit levert ons mogelijk een exponentieel grote expressie op, waar dit niet nodig is.

We willen methodes onderzoeken om een expressie korter te maken, voornamelijk om de leesbaarheid voor mensen te verbeteren. Deze methodes zouden we kunnen toepassen op resultaten van het algoritme uit Deel I als ze de one-unambiguïty van de expressie bewaren. Dergelijke methodes kunnen we echter ook gebruiken op andere computergegenereerde expressies.

In Deel II van deze thesis onderzoeken we een aantal methodes om expressies korter te maken. Sommige hiervan bewaren de one-unambiguïty van de expressie, andere niet. We hebben ook verschillende methodes geïmplementeerd, hiervan bespreken we de implementatie.

1.1 Verdere indeling van deze thesis

Hoofdstuk 2 behandelt de definities van reguliere expressies en one-unambiguous reguliere expressies.

1.1.1 Indeling van Deel I

In Hoofdstuk 3 bespreken we het algoritme dat we gebruiken om na te gaan of een reguliere expressie one-unambiguous is. Indien er een equivalente one-unambiguous expressie bestaat, kunnen we deze ook construeren.

1.1.2 Indeling van Deel II

We merken op dat het algoritme uit het vorige hoofdstuk vaak langere expressies teruggeeft dan nodig. We gaan in Hoofdstuk 4 dan ook na wat precies de lengte van een reguliere expressie is. In de volgende hoofdstukken bestuderen we een aantal specifieke methodes om expressies korter te maken.

We onderzoeken een verzameling herschrijfgeregels die een reguliere expressie korter maken in Hoofdstuk 5. Deze regels zijn gebaseerd op een axiomasysteem van Salomaa ([Salomaa, 1966]).

In Hoofdstuk 6 bestuderen we een aantal methodes om een zo klein mogelijke expressie te verkrijgen uit een automaat voor een taal.

We vergelijken de besproken methodes om een korte expressie te verkrijgen en bestuderen de resultaten van onze tests in Hoofdstuk 7.

Hoofdstuk 8 geeft een korte uitleg bij het gebruik van de implementatie bij deze thesis.

Ten slotte bevat Hoofdstuk 9 nog een korte conclusie van deze thesis.

Hoofdstuk 2

Definities

2.1 Reguliere expressies

Neem $\Sigma = \{a, b, \dots\}$ een eindige verzameling symbolen, dit noemen we het alfabet. Een woord of string w over dit alfabet is een eindige opeenvolging van symbolen uit dit alfabet. De lege string stellen we voor met ϵ . Het symbool $\emptyset$ stelt de lege verzameling voor. De verzameling woorden of strings over dit alfabet stellen we voor met Σ^* . Een taal is een deelverzameling van Σ^* .

Met $w_1 \cdot w_2$ geven we de concatenatie van twee woorden w_1 en w_2 aan. We gebruiken w^i om aan te geven dat we i keer w met zichzelf concateneren.

Definitie 1. Een reguliere expressie over een alfabet $\Sigma = \{a, b, \dots\}$ wordt recursief gedefinieerd als volgt [Gelade et al., 2007]:

1. een enkel symbool uit het alfabet Σ , een ϵ of een $\emptyset$ is een reguliere expressie,
2. als E en F reguliere expressies zijn, dan zijn

- $(E + F)$,
- $(E \cdot F)$ en
- E^*

reguliere expressies,

De $\cdot$ wordt in de praktijk vaak weggelaten. Wanneer het duidelijk is, worden ook de haakjes rond $+$ en $\cdot$ vaak weggelaten, de volgorde van de operatoren is dan eerst $*$, vervolgens $\cdot$ en ten slotte $+$.

Voorbeeld 1. $ab + cd^*$ is equivalent met $((a \cdot b) + (c \cdot (d^*)))$

Een reguliere expressie E beschrijft nu een verzameling woorden, de taal $L(E)$ van de expressie, als volgt:

1. $L(a) = \{a\}, L(\epsilon) = \{\epsilon\}$ en $L(\emptyset) = \emptyset$,
2. neem twee reguliere expressies E en F . Dan is:
 - $L(E + F) = L(E) \cup L(F)$,
 - $L(E \cdot F) = \{w_1 \cdot w_2 \mid w_1 \in L(E), w_2 \in L(F)\}$
 - $L(E^*) = \{\epsilon\} \cup \bigcup_{i=1}^{\infty} L(E)^i$.

Als een woord w zich in $L(E)$ bevindt, zeggen we dat w gedefinieerd wordt door E .

2.2 One-unambiguous reguliere expressies

Definitie 2. Een subexpressie van een reguliere expressie is een substring van deze expressie die zelf een geldige reguliere expressie is.

Voorbeeld 2. De expressie $(a \cdot b)$ is een subexpressie van de reguliere expressie $((a \cdot b) + c)^*$. Let hierbij wel op: de expressie $(b + c)$ is geen subexpressie van de reguliere expressie $(a + b) + (c + d)$, maar wel van de equivalente expressie $(a + (b + c)) + d$, ook al schrijven we beide expressies vaak als $a + b + c + d$.

We definiëren een nieuw alfabet $\Pi = \{a_1, b_1, a_2, b_2, \dots\}$, dat symbolen met een subscript bevat.

Om een onderscheid te kunnen maken tussen twee voorkomens van hetzelfde symbool in een reguliere expressie, voeren we een markering in. Een markering voor een reguliere expressie E over het alfabet Σ (zie paragraaf 2.1) is een reguliere expressie E' over Π waarin elk symbool uit de originele expressie vervangen wordt door een uniek symbool¹. Als we van een gemarkeerd symbool het origineel symbool willen, duiden we dit aan met $^{\natural}$. Als we een markering nemen van een subexpressie, gaan we er van uit dat de symbolen met hetzelfde unieke symbool gemarkeerd worden.

Voorbeeld 3. De reguliere expressie aab^* heeft $a_1a_2b_1^*$ of $a_3a_4b_5^*$ als mogelijke markeringen. Hierbij is dan $(a_1)^{\natural} = a$. Als we $a_1a_2b_1^*$ als markering nemen, dan heeft de subexpressie b^* de markering b_1^* .

Een reguliere expressie is unambiguous als we voor elk symbool in een string gedefinieerd door die expressie kunnen nagaan met welk symbool in de reguliere expressie het overeenkomt. De expressie is one-unambiguous als we dit kunnen bepalen zodra we het symbool in de string tegenkomen zonder vooruit te kijken.

Definitie 3. Een reguliere expressie E is one-unambiguous als en slechts als voor alle woorden u, v, w over Π en alle symbolen x, y in Π geldt dat $uxv, yvw \in L(E')$ en $x \neq y$ impliceert dat $x^{\natural} \neq y^{\natural}$.

¹Dit wordt aangegeven door hetzelfde symbool te gebruiken met een subscript.

Voorbeeld 4. De expressie $E = a^*a$ is niet one-unambiguous. We nemen de markering $E' = a_1^*a_2$. De woorden $a_1a_1a_2$ en a_1a_2 behoren allebei tot $L(E')$. Neem nu $u = a_1$, $x = a_1$, $y = a_2$, $v = a_2$ en $w = \epsilon$. Dan hebben we twee woorden $uxv = a_1a_1a_2$ en $uyw = a_1a_2$ in $L(E')$ en $a_1 = x \neq y = a_2$ en toch $a = x^\sharp = y^\sharp = a$. We konden voor een symbool a in de string niet nagaan of het bij a_1 of a_2 hoorde. De equivalente expressie aa^* is echter wel one-unambiguous.

We zeggen dat een taal one-unambiguous is als er een one-unambiguous reguliere expressie is die de taal beschrijft.

Deel I

**One-unambiguous reguliere
expressies**

Hoofdstuk 3

Omzetting naar one-unambiguous reguliere expressies

3.1 Inleiding

In dit hoofdstuk wordt een algoritme beschreven dat van een reguliere expressie:

- kan nagaan of ze one-unambiguous is,
- kan nagaan of er een equivalente one-unambiguous reguliere expressie bestaat,
- als ze bestaat, een equivalente one-unambiguous reguliere expressie kan produceren.

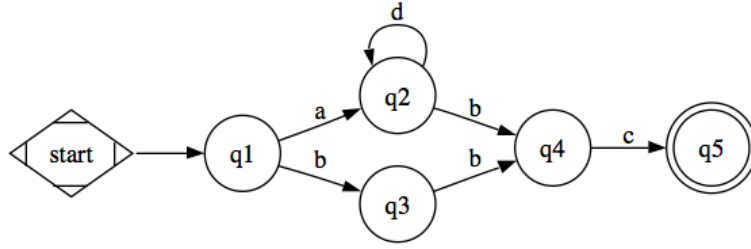
Dit algoritme werkt met optimale worst-case tijdscomplexiteit. Het algoritme is een uitwerking van de ideeën in [Brüggemann-Klein and Wood, 1998] en [Brüggemann-Klein, 1993].

Het algoritme zal de invoerstring eerst in een syntaxboom omzetten. Op deze boom worden een aantal transformaties toegepast die het toelaten deze efficiënt om te zetten naar een automaat. Deze automaat laat ons toe te beslissen of de expressie one-unambiguous was.

Als we deze automaat determiniseren en minimaliseren, kunnen we nagaan of de expressie een equivalente one-unambiguous expressie heeft. Als dit zo is, kunnen we van deze geminimaliseerde automaat gebruik maken om een dergelijke expressie te vormen.

3.2 Definities

In dit hoofdstuk gebruiken we notatie zoals die is aangegeven in Hoofdstuk 2. We gebruiken E voor een reguliere expressie, E' voor de gemarkeerde expressie en $L(E)$ is de taal beschreven door de expressie E .



Figuur 3.1: Deterministische automaat of DFA.

Definitie 4. Een deterministische automaat of DFA is een tuple $M = (Q, \Sigma, \delta, q_s, F)$ en bestaat uit:

- Q een eindige verzameling toestanden,
- Σ een eindig alfabet,
- $\delta : Q \times \Sigma \rightarrow Q$ een overgangsfunctie,
- q_s een starttoestand en
- F een verzameling eindtoestanden.

Als voor toestanden q en p en symbool a , $\delta(q, a) = p$, zeggen we dat q een transitie heeft naar p met symbool a . We geven deze transitie weer als (q, a, p) .

Definitie 5. Een run van een deterministische automaat op een woord $w = a_1 \dots a_n$ is een functie $\lambda : \{1, \dots, n+1\} \rightarrow Q$ zodat:

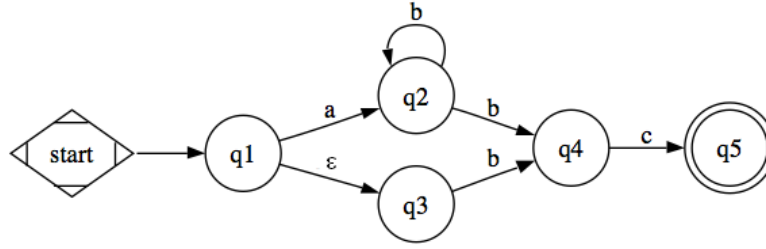
- $\lambda(1) = q_s$ en,
- voor alle $i \in \{1, \dots, n\}$, $\lambda(i+1) = \delta(\lambda(i), a_i)$.

Een run is een accepterende run als $\lambda(n+1) \in F$. Een woord wordt geaccepteerd als er een accepterende run voor is. De verzameling van alle woorden die geaccepteerd wordt door een automaat M , noemen we de taal van M , en noteren we met $L(M)$.

Voorbeeld 5. In Figuur 3.1 is een deterministische automaat weergegeven. Deze accepteert het woord *addbc* omdat er een accepterende run voor is, namelijk: $\lambda(1) = q_1, \lambda(2) = q_2, \lambda(3) = q_2, \lambda(4) = q_2, \lambda(5) = q_4, \lambda(6) = q_5$. Dit geeft een pad doorheen de automaat waarvan de labels het woord vormen. Het pad eindigt in een eindtoestand, dus het woord wordt geaccepteerd.

Definitie 6. Een niet-deterministische automaat of NFA is een deterministische automaat, maar de transitiefunctie is van de vorm $\delta : Q \times (\Sigma \cup \{\epsilon\}) \rightarrow P(Q)$ met $P(Q)$ de powerset¹ van Q .

¹De verzameling van alle mogelijke deelverzamelingen.



Figuur 3.2: Niet-deterministische automaat of NFA.

Definitie 7. Een run van een niet-deterministische automaat op een woord $w = a_1 \dots a_n$ met $a_i \in (\Sigma \cup \{\epsilon\})$ is een functie $\lambda : \{1, \dots, n+1\} \rightarrow Q$ zodat:

- $\lambda(1) = q_s$ en,
- voor alle $i \in \{1, \dots, n\}$, $\lambda(i+1) \in \delta(\lambda(i), a_i)$.

Een run is een accepterende run als $\lambda(n+1) \in F$. Een woord wordt geaccepteerd als het woord in een vorm $w = a_1 \dots a_n$ met $a_i \in (\Sigma \cup \{\epsilon\})$ kan worden geschreven, door enkel epsilon toe te voegen, zodat er een accepterende run voor is. De verzameling van alle woorden die geaccepteerd wordt door M , noemen we de taal van M , en noteren we met $L(M)$.

Voorbeeld 6. In Figuur 3.2 is een niet-deterministische automaat weergegeven. Deze accepteert het woord bc omdat er een woord is ϵbc waarvoor er een accepterende run is, namelijk: $\lambda(1) = q_1, \lambda(2) = q_3, \lambda(3) = q_4, \lambda(4) = q_5$. Dit speelt een pad doorheen de automaat waarvan de labels het woord vormen. Het pad eindigt in een eindtoestand, dus het woord wordt geaccepteerd.

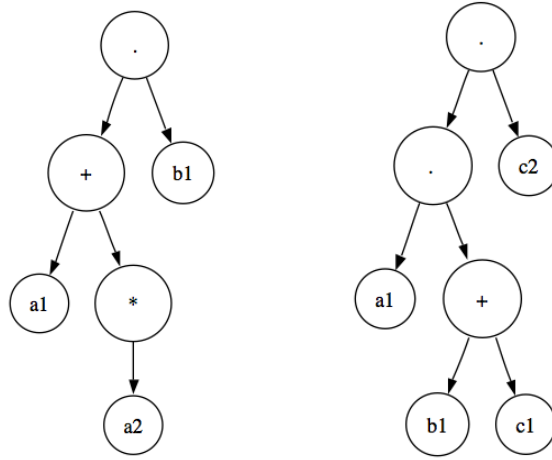
3.3 Nagaan of een reguliere expressie one-unambiguous is

We willen hier nagaan of een reguliere expressie one-unambiguous is. Om dit te controleren zetten we eerst de stringrepresentatie van de expressie om in een syntaxboom. Met behulp van deze boom construeren we een eindige automaat waarvan de eigenschappen aangeven of de reguliere expressie one-unambiguous is.

3.3.1 De invoer omzetten in een syntaxboom

De invoer voor het algoritme is een reguliere expressie die gecontroleerd en eventueel omgezet moet worden. We nemen aan dat we een stringrepresentatie van een reguliere expressie krijgen als invoer. Deze zetten we dan om in een syntaxboom waarop de rest van het algoritme werkt. Dit kan gebeuren in lineaire tijd ([Aho et al., 1986]).

Definitie 8. Een syntaxboom is een binaire boom waarvan een interne knoop één van de operatoren en een bladknoop een symbool als label heeft.



Figuur 3.3: Syntaxbomen voor: 1) $(a + a^*)b$ en 2) $a(b + c)c$.

Wanneer we de string parsen, vervangen we onmiddellijk de symbolen door hun gemarkeerde tegenhanger (zie paragraaf 2.1). Dit is eenvoudig uit te voeren door elk symbool met een getal te concateneren. Hiervan zullen we gebruik maken in paragraaf 3.4.2. We moeten wel van elk gemarkeerd symbool kunnen nagaan met welk niet gemarkeerd symbool het overeenkomt, maar dit is dan uit te voeren door de string op te splitsen.

Voorbeeld 7. De reguliere expressie $(a + a^*)b$ zetten we om in de syntaxboom in Figuur 3.3 1). De expressie $a(b + c)c$ wordt de boom in Figuur 3.3 2).

3.3.2 De syntaxboom transformeren naar zijn Glushkov automaat

We gaan de reguliere expressie omzetten in een eindige automaat. Deze automaat noemen we de Glushkov automaat van de expressie. We zullen zien dat een reguliere expressie one-unambiguous is als en slechts als zijn Glushkov automaat een deterministische eindige automaat (of DFA) is.

Inleidende definities

Definitie 9. Voor een taal L en elk symbool a , definiëren we de volgende verzamelingen van symbolen b ([Brüggemann-Klein and Wood, 1998]):

- $first(L) = \{b \mid \text{Er is een woord } w \text{ zodat } bw \in L\},$
- $last(L) = \{b \mid \text{Er is een woord } w \text{ zodat } wb \in L\},$
- $follow(L, a) = \{b \mid \text{Er zijn woorden } v \text{ en } w \text{ zodat } vabw \in L\},$
- $followlast(L) = \{b \mid \text{Er zijn woorden } v \text{ en } w \text{ zodat } v \in L, v \neq \epsilon \text{ en } vbw \in L\}.$

We breiden deze definities uit voor een reguliere expressie E , zodat $first(E) = first(L(E)), \dots$

Voorbeeld 8. De reguliere expressie $E = a^*a + bc$ heeft de volgende first-, last- en follow-verzamelingen:

- $first(E) = \{a, b\}$,
- $last(E) = \{a, c\}$,
- – $follow(E, a) = \{a\}$,
- $follow(E, b) = \{c\}$,
- $follow(E, c) = \emptyset$.

De markering voor E , namelijk $E' = a_1^*a_2 + b_1c_1$, heeft de volgende first-, last- en follow-verzamelingen:

- $first(E') = \{a_1, a_2, b_1\}$,
- $last(E') = \{a_2, c_1\}$,
- – $follow(E', a_1) = \{a_1, a_2\}$,
- $follow(E', a_2) = \emptyset$,
- $follow(E', b_1) = \{c_1\}$,
- $follow(E', c_1) = \emptyset$.

Stelling 1. Voor elke markering E' , behoort een woord $x_1 \dots x_n$ over Π met $n \geq 1$ tot $L(E')$ als en slechts als:

- $x_1 \in first(E')$,
- $x_n \in last(E')$,
- $x_{i+1} \in follow(E', x_i)$ voor $1 \leq i < n$.

Bewijs. Als het woord $x_1 \dots x_n$ tot de taal behoort, volgt uit de definities van de verzamelingen, dat de verzamelingen de gewenste symbolen bevatten.

Er rest ons dus enkel de andere richting te bewijzen. Dit bewijzen we door inductie op de structuur van de markering:

Basis:

We onderscheiden twee basisgevallen:

- de markering is ϵ of $\emptyset$,
- de markering is van de vorm x_i met x_i een symbool uit het alfabet.

Als de markering van de vorm ϵ of $\emptyset$ is, dan behoort er geen enkel woord van de vorm $x_1 \dots x_n$ met $n \geq 1$ tot de taal. De first-, last- en follow-verzamelingen zijn dan ook altijd leeg. Een dergelijk woord $x_1 \dots x_n$ komt dus enkel voor als de verzamelingen de juiste informatie bevatten, zijnde nooit.

Als de markering de vorm x_i heeft, dan is het enige woord dat gedefinieerd wordt x_i . De enige informatie die dan in de first-, last- en follow-verzamelingen komt, is $first(E') = \{x_i\}$ en $last(E') = \{x_i\}$.

Inductiehypothese: De uitspraak is geldig voor markeringen E'_1 en E'_2 .

Te bewijzen: De uitspraak is geldig voor $E'_1 \cdot E'_2$, $E'_1 + E'_2$ en E'_1^* .

- $E'_1 \cdot E'_2$: Aangezien het gaat om een markering, weten we dat de symbolen in E'_1 en E'_2 disjunct zijn. Dus zijn ook hun first-, last- en followverzamelingen disjunct. We hebben nu de first-, last- en followverzamelingen voor E , hieruit kunnen we deze verzamelingen construeren voor E'_1 en E'_2 :

- $first(E'_1)$ is de deelverzameling van $first(E')$ met enkel symbolen uit E'_1 ,
- $last(E'_1)$ zijn de symbolen uit E'_1 die in $follow(E')$ gevolgd kunnen worden door een symbool uit E'_2 , samen met de deelverzameling van $last(E')$ met enkel symbolen uit E'_1 ,
- $follow(E'_1)$ is de deelverzameling van $follow(E')$ met enkel symbolen uit E'_1 ,
- $first(E'_2)$ zijn de symbolen uit E'_2 die in $follow(E')$ voorafgegaan kunnen worden door een symbool uit E'_1 , samen met de deelverzameling van $first(E')$ met enkel symbolen uit E'_2 ,
- $last(E'_2)$ is de deelverzameling van $last(E')$ met enkel symbolen uit E'_2 ,
- $follow(E'_2)$ is de deelverzameling van $follow(E')$ met enkel symbolen uit E'_2 .

Omdat E' van de vorm $E'_1 \cdot E'_2$ is, kan men nu elk woord $x_1 \dots x_n$ in $L(E')$ opsplitsen in twee woorden $x_1 \dots x_j$ en $x_{j+1} \dots x_n$ door na te gaan waar de overgang van symbolen tussen E'_1 en E'_2 plaatsvindt. Stel dat $1 \leq j < n$, dus geen van de twee woorden is de lege string, dan behoort x_1 tot $first(E'_1)$, want het behoort tot $first(E')$ en is een symbool uit E'_1 . Het symbool x_j behoort tot $last(E'_1)$ want $x_{j+1} \in follow(E', x_j)$ (gegeven) en x_j is een symbool uit E'_1 en x_{j+1} is een symbool uit E'_2 . Alle $x_{i+1} \in follow(E'_1, x_i)$ voor $1 \leq i < j$ want het zijn symbolen uit E'_1 . De redenering voor E'_2 is volledig analoog.

De twee woorden $x_1 \dots x_j$ en $x_{j+1} \dots x_n$ worden nu gedefinieerd door E'_1 en E'_2 vanwege de inductiehypothese. Dus wordt $x_1 \dots x_n$ gedefinieerd door E' .

Als $j = 0$, dan krijgen we de twee woorden ϵ en $x_1 \dots x_n$. Het symbool x_1 zit in $first(E'_2)$, want het zit in $first(E')$ en is een symbool uit E'_2 . Het symbool x_n zit in $last(E'_2)$ want het zit in $last(E')$ en is een symbool uit E'_2 . De nodige follow-informatie zit ook in $follow(E'_2)$ zodat $x_1 \dots x_n$ gedefinieerd wordt door E'_2 . Aangezien er een symbool uit E'_2 in $first(E')$ zit, wil dit zeggen dat er een woord $\epsilon \cdot w \in L(E')$ is, met $w \in L(E'_2)$. Hieruit volgt dat $\epsilon \in L(E'_1)$. Dus wegens de inductiehypothesen wordt $x_1 \dots x_n$ gedefinieerd door E'_2 , en omdat $\epsilon \in E'_1$, wordt $x_1 \dots x_n$ ook gedefinieerd door E' .

De redenering voor $j = n$ is analoog.

De redenering voor de overige twee gevallen is analoog. Voor een volledige uitwerking van dit bewijs, zie Appendix A. $\square$

Zoals reeds gedefinieerd in paragraaf 2.2 is $x^\natural$ de ongemarkeerde versie van symbool x , bijvoorbeeld $a_1^\natural = a$.

Noem $\text{sym}(E)$ de verzameling symbolen in een reguliere expressie E .

Definitie 10. Voor een reguliere expressie E is de Glushkov automaat $M_E = (Q_E, \Sigma, \delta_E, q_I, F_E)$ met [Brüggemann-Klein and Wood, 1998]:

- $Q_E = \text{sym}(E') \cup \{q_I\}$, de gemarkeerde symbolen samen met een nieuwe starttoestand q_I ,
- $\Sigma = \text{sym}(E)$,
- voor $a \in \Sigma$, $\delta_E(q_I, a) = \{x \mid x \in \text{first}(E'), x^\natural = a\}$,
voor $x \in \text{sym}(E')$, $a \in \Sigma$, $\delta_E(x, a) = \{y \mid y \in \text{follow}(E', x), y^\natural = a\}$,
- q_I is een nieuwe starttoestand,
- als $\epsilon \in L(E)$: $F_E = \text{last}(E') \cup \{q_I\}$,
als $\epsilon \notin L(E)$: $F_E = \text{last}(E')$.

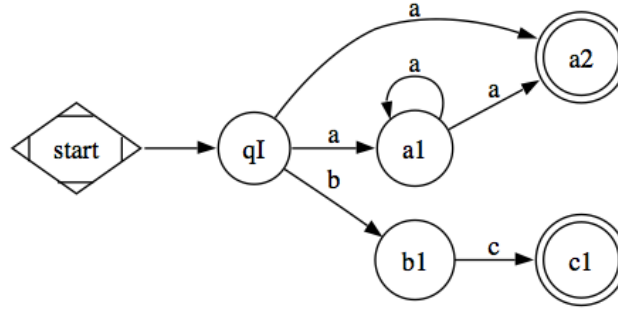
Deze automaat wordt dus volledig gedefinieerd door de first-, last- en follow-verzamelingen van de gemarkeerde reguliere expressie. Bij het parsen van de stringrepresentatie van de expressie naar zijn syntaxboom (zie paragraaf 3.3.1) hebben we de expressie al in een markering omgezet. Hoe we nu uit deze boom de informatie voor de verzamelingen verkrijgen, en zo dus ook de Glushkov automaat construeren, wordt verder in dit onderdeel beschreven.

Voorbeeld 9. De markering $E' = a_1^*a_2 + b_1c_1$ gaf ons de volgende verzamelingen:

- $\text{first}(E') = \{a_1, a_2, b_1\}$,
- $\text{last}(E') = \{a_2, c_1\}$,
- - $\text{follow}(E', a_1) = \{a_1, a_2\}$,
 - $\text{follow}(E', a_2) = \emptyset$,
 - $\text{follow}(E', b_1) = \{c_1\}$,
 - $\text{follow}(E', c_1) = \emptyset$.

Dit levert ons dan de Glushkov automaat op in Figuur 3.4 voor de reguliere expressie $a^*a + bc$.

Stelling 2. Een reguliere expressie E is one-unambiguous als en slechts als zijn Glushkov automaat M_E deterministisch is.



Figuur 3.4: Glushkov automaat voor de expressie $a^*a + bc$.

Bewijs. Uit Stelling 1 volgt dat voor elke markering E' , voor woorden u_1, u_2, v_1, v_2 over Π en symbool z uit Π :

Als $u_1zv_1, u_2zv_2 \in L(E')$, dan $u_1zv_2 \in L(E')$. Want het eerste symbool van u_1 zit in $first(E')$, het laatste symbool van u_2 zit in $last(E')$ en ook de nodige follow-informatie is aanwezig om u_1zv_2 in de taal te plaatsen.

Als we terugkijken naar Definitie 3 onderscheiden we twee gevallen. Het kan zo zijn dat u de lege string is, of u is een woord dat eindigt op een symbool in Π , noem dit z .

Met behulp hiervan bekomen we de volgende alternatieve definitie voor een one-unambiguous reguliere expressie:

Een reguliere expressie E is one-unambiguous als en slechts als:

- voor alle $x, y \in first(E')$, $x \neq y$ impliceert dat $x^\natural \neq y^\natural$ en,
- voor alle $z \in sym(E')$ en $x, y \in follow(E', z)$, $x \neq y$ impliceert dat $x^\natural \neq y^\natural$.

Als we nu de definitie van de Glushkov automaat bekijken, merken we op dat hieruit volgt dat alle transities van de automaat deterministisch zijn. $\square$

Een reguliere expressie omzetten in ster-normaalvorm

Als we een naïef algoritme zouden toepassen om de expressie om te zetten in zijn Glushkov automaat, dan heeft dat algoritme kubische tijd nodig. We kunnen dit echter verminderen als de expressie in een andere vorm staat. Deze vorm heet de ster-normaalvorm. Als de expressie in deze vorm staat hebben we slechts kwadratische tijd nodig.

Definitie 11. Een reguliere expressie is in ster-normaalvorm als voor elke subexpressie H^* van E , $followlast(H') \cap first(H') = \emptyset$ en $\epsilon \notin L(H)$.

Dit betekent dat de Glushkov automaat die een subexpressie onder een ster beschrijft, geen transities van een eindtoestand naar een starttoestand mag hebben. De automaat mag ook niet de lege string accepteren. Op deze manier kan de ster-operator geen transities aan de automaat toevoegen die er al waren.

Stelling 3. *Voor elke expressie E is er een equivalente expressie $E^\bullet$ in ster-normaalvorm met dezelfde Glushkov automaat. Deze expressie kan uit de originele expressie berekend worden in lineaire tijd.*

Bewijs. De lineariteit van de omzetting zal duidelijk blijken uit de constructie die volgt. Voor het bewijs van de equivalentie van de expressies en de gelijkheid van de automaten verwijzen we naar [Brüggemann-Klein, 1993]. $\square$

Een eerste stap bij het omzetten naar ster-normaalvorm, is voor elke subexpressie te bepalen of de taal beschreven door die subexpressie de lege string bevat. Om dit te berekenen roepen we op de rootknoop van de syntaxboom de functie `nullable()` aan. Die berekent de waarde recursief als volgt:

case

- v is een knoop met label $\emptyset$:
 `nullable() = false;`
- v is een knoop met label ϵ :
 `nullable() = true;`
- v is een knoop met label x :
 `nullable() = false;`
- v is een knoop met label $+$:
 `nullable() = leftchild.nullable() OR rightchild.nullable();`
- v is een knoop met label $.$:
 `nullable() = leftchild.nullable() AND rightchild.nullable();`
- v is een knoop met label $*$:
 `nullable() = true;`

end case

Om de syntaxboom om te zetten naar ster-normaalvorm definiëren we in elke knoop twee functies: `makeStarNormal()` en `makeStarNormalCut()`. De intuïtie achter deze twee functies is dat `makeStarNormal()` de knoop zal omzetten in ster-normaalvorm, en `makeStarNormalCut()` de knoop zal omzetten in ster-normaalvorm waarbij de terugkerende transities in de corresponderende automaat verwijderd zijn. Deze laatste zal enkel aangeroepen worden op knopen die zich onder een ster-operator bevinden en deze operator zal ervoor zorgen dat deze transities in de automaat terug opgenomen worden.

Om de omzetting uit te voeren, roepen we op de root van de syntaxboom de recursieve functie `makeStarNormal()` op. Elke knoop zal dezelfde functie aanroepen op zijn kinderen (en gewoon stoppen als hij geen kinderen heeft), behalve in het geval van de ster-operator. De ster-operator roept de functie `makeStarNormalCut()` aan op zijn enige kind.

De functie `makeStarNormalCut()` vervangt de subexpressie E door een nieuwe expressie $E^{\bullet\circ}$.

De expressie $E^{\bullet\circ}$ bereken je als volgt:

```

if(E ==  $\emptyset$  or E ==  $\epsilon$ )
     $E^{\bullet\circ} = \emptyset$ 
if(E == a)
     $E^{\bullet\circ} = E$ 
if(E == F + G)
     $E^{\bullet\circ} = F^{\bullet\circ} + G^{\bullet\circ}$ 
if(E == FG)
    if( $\epsilon \notin L(F)$  and  $\epsilon \notin L(G)$ )
         $E^{\bullet\circ} = F^{\bullet}G^{\bullet}$ 
    if( $\epsilon \notin L(F)$  and  $\epsilon \in L(G)$ )
         $E^{\bullet\circ} = F^{\bullet\circ}G^{\bullet}$ 
    if( $\epsilon \in L(F)$  and  $\epsilon \notin L(G)$ )
         $E^{\bullet\circ} = F^{\bullet}G^{\bullet\circ}$ 
    if( $\epsilon \in L(F)$  and  $\epsilon \in L(G)$ )
         $E^{\bullet\circ} = F^{\bullet\circ} + G^{\bullet\circ}$ 
if(E ==  $F^*$ )
     $E^{\bullet\circ} = F^{\bullet\circ}$ 

```

Bij de kindknopen wordt enkel in het geval van ϵ het kind veranderd naar $\emptyset$, verder gebeurt er niets.

De operatoren roepen enkel de juiste functies aan op hun kinderen als volgt:

- elk voorkomen van $F^{\bullet}$ of $G^{\bullet}$ in deze definitie is een aanroep van `makeStarNormal()` en,
- voorkomens van $F^{\bullet\circ}$ of $G^{\bullet\circ}$ zijn een aanroep van `makeStarNormalCut()`.

Hierbij zijn er twee uitzonderingen waarop gelet moet worden:

- als beide kinderen van een concatenatie-knoop de lege string definiëren, wordt deze een union-knoop. De logica hierachter is dat, aangezien allebei de subexpressies de lege string definiëren, we in elke iteratie van de ster-operator één van de twee subexpressies kunnen kiezen. Als een woord door een opeenvolging van deze expressies gedefinieerd wordt, wordt dit behandeld door de ster-operator.
- een sterknoop vervangt zichzelf volledig door de `makeStarNormalCut()` van zijn kind. Twee opeenvolgende ster-operatoren in de syntaxboom hebben geen nut.

Uiteindelijk hebben we dan de syntaxboom van $E^{\bullet}$ en die is in ster-normaalvorm.

Aangezien we in elke knoop constante tijd gebruiken en elke knoop maar één keer bezoeken, is het algoritme lineair in het aantal knopen.

Voorbeeld 10. Als we de reguliere expressie $(a^*b^*)^*$ omzetten in ster-normaalvorm krijgen we de expressie $(a + b)^*$.

Een reguliere expressie trim maken

We passen de volgende vervangingsregels toe in een pass door de boom om de reguliere expressie trim te maken ([Brüggemann-Klein and Wood, 1998]):

- $E + \emptyset, \emptyset + E \rightarrow E$
- $E\emptyset, \emptyset E \rightarrow \emptyset$
- $\emptyset^* \rightarrow \epsilon$

Dit zorgt ervoor dat ofwel de hele expressie $\emptyset$ wordt, ofwel dat alle voorkomens van $\emptyset$ uit de expressie verwijderd worden. Het is duidelijk dat:

- de resulterende expressie equivalent is met de originele,
- deze transformatie in lineaire tijd kan gebeuren.

Deze transformatie is niet nodig om te bepalen of de expressie one-unambiguous is, maar ze is essentieel om een equivalente one-unambiguous reguliere expressie te vormen (zie paragraaf 3.4.2).

Het is ook belangrijk om aan te tonen dat een reguliere expressie one-unambiguous is als en slechts als zijn trim reguliere expressie one-unambiguous is. Aangezien E^{tt} een markering is voor E^t en $L(E') = L(E^{tt})$ is E one-unambiguous als en slechts als E^t one-unambiguous is wegens de definitie (zie paragraaf 2.2).

De Glushkov automaat construeren voor een expressie in ster-normaalvorm

Om de Glushkov automaat te construeren moeten we de first-, last- en follow-verzamelingen van de expressie aanmaken. Dit kunnen we doen door de syntaxboom van $E^\bullet$ in postorde te doorlopen.

In elke knoop v definiëren we ook de functies nullable(), first() en last(). Voor elk gemarkeerd symbool x (dat we ingevoerd hebben in paragraaf 3.3.1) definiëren we ook een globale verzameling follow(x).

De functie nullable() geeft aan of de taal van de subexpressie de lege string bevat en werkt op dezelfde manier als in paragraaf 3.3.2. De functies first() en last() stellen de corresponderende verzamelingen voor en bevatten dus de eerste en laatste gemarkeerde symbolen van de subexpressie. De functie follow(x) houdt dus ook de gemarkeerde symbolen bij die op x kunnen volgen. Bij deze berekeningen maken we gebruik van de inductieve definitie van de first-, last- en follow-verzamelingen uit [Brüggemann-Klein, 1993] die equivalent zijn met onze gegeven definitie.

In elke knoop v in de boom wordt de volgende pseudo-code uitgevoerd:

case

v is een knoop met label $\emptyset$:

```
    nullable() = false;
    first() =  $\emptyset$ ;
    last() =  $\emptyset$ ;
```

v is een knoop met label ϵ :

```
    nullable() = true;
    first() =  $\emptyset$ ;
    last() =  $\emptyset$ ;
```

v is een knoop met label x :

```
    nullable() = false;
    follow(x) =  $\emptyset$ ;
    first() =  $x$ ;
    last() =  $x$ ;
```

v is een knoop met label $+$:

```
    nullable() = leftchild.nullable() OR rightchild.nullable();
    first() = leftchild.first()  $\cup$  rightchild.first(); (*)
    last() = leftchild.last()  $\cup$  rightchild.last(); (*)
```

v is een knoop met label $.$:

```
    nullable() = leftchild.nullable() AND rightchild.nullable();
    for alle x in leftchild.last() do
        follow(x) = follow(x)  $\cup$  rightchild.first(); (**)
    if leftchild.nullable()
        first() = leftchild.first()  $\cup$  rightchild.first(); (*)
    else
        first() = leftchild.first();
    if rightchild.nullable()
        last() = leftchild.last()  $\cup$  rightchild.last(); (*)
    else
        last() = rightchild.last();
```

v is een knoop met label $*$:

```
    nullable() = true;
    for alle x in child.last() do
        follow(x) = follow(x)  $\cup$  child.first(); (***)
    first() = child.first();
    last() = child.last();
```

end case

Als we verzamelingen voorstellen als gesorteerde lijsten, kunnen we de unie implementeren in lineaire tijd ten opzichte van de grootte van beide verzamelingen. Als we dit algoritme dus voor een willekeurige expressie uitvoeren, hebben we kubische tijd nodig.

De unies aangegeven met (*) en (**) zijn echter altijd disjunct, want het gaat hier over symbolen uit verschillende subexpressies van dezelfde markering. Alleen bij de unie aangegeven met (***) kunnen de verzamelingen elementen gemeen hebben. Maar uit de definitie van een expressie in ster-normaalvorm (zie paragraaf 3.3.2) volgt dat voor zulke expressies deze unie ook disjunct is.

Dit laat ons toe om verzamelingen voor te stellen als niet gesorteerde lijsten. De unies aangegeven met (*) kunnen we nu met een lijstconcatenatie zonder kopiëren uitvoeren in constante tijd, omdat we de bronlijsten verder in het algoritme niet nodig hebben. Bij de unies aangegeven met (**) en (***) hebben we de tweede set nog nodig in een lus. Deze kunnen dus geïmplementeerd worden door de elementen van de tweede verzameling één voor één te kopiëren naar het einde van de eerste verzameling. Deze operatie is dan lineair in de grootte van enkel de tweede verzameling.

De unies aangegeven met (**) en (***) zijn van de vorm $X = Y \cup Z$, waarbij Y en Z disjunct zijn. We hebben Z nog wel verder nodig in de for-lus, dus we implementeren dit door alle elementen van Z te kopiëren naar het einde van Y . De tijdsduur van deze operatie is dus lineair in de grootte van Z .

Omdat de unies altijd disjunct zijn, wil dit zeggen dat elk symbool altijd maar hoogstens één keer aan elke follow-verzameling kan worden toegevoegd. De totale tijdsduur die we hooguit nodig hebben om al deze unies aangegeven met (**) en (***) te berekenen, is dus evenredig met het kwadraat van het aantal symbolen. Het hele algoritme kan dan ook uitgevoerd worden in kwadratische tijd.

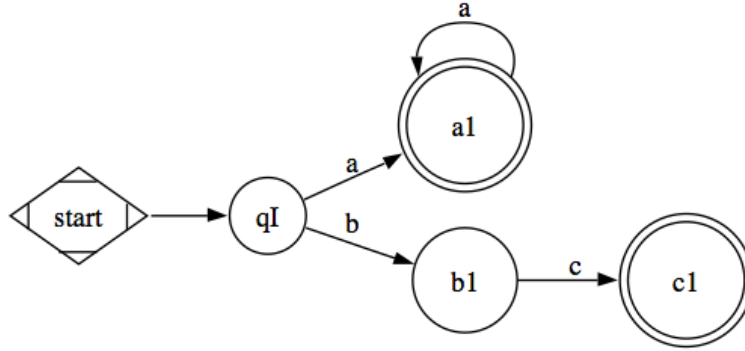
Hiermee hebben we dus de Glushkov automaat van de expressie bepaald zoals gedefinieerd in paragraaf 3.3.2.

We controleren of deze automaat deterministisch is. Wanneer dit het geval is, is de reguliere expressie one-unambiguous (wegens Stelling 2). Als de reguliere expressie one-unambiguous was, stopt het algoritme, anders gaat het door en zal het een equivalente one-unambiguous reguliere expressie construeren als deze bestaat.

3.4 Bepalen of een taal one-unambiguous is

Omdat voor elke taal L er een unieke minimale deterministische automaat is (op het hernoemen van toestanden na), willen we deze automaat gebruiken om na te gaan of de taal one-unambiguous is (zie paragraaf 2.2). We noemen deze automaat $MS(L)$.

Eerst nemen we een reguliere expressie E die de taal L beschrijft. We kunnen dan zijn Glushkov automaat gebruiken om de automaat $MS(L)$ te construeren.



Figuur 3.5: Minimale deterministische automaat voor $L(a^*a + bc)$.

3.4.1 De Glushkov automaat determiniseren en minimaliseren

In paragraaf 3.3 verkregen we de Glushkov automaat van de reguliere expressie. Hiervan willen we nu de automaat $MS(L(E))$ bestuderen. Om deze DFA te verkrijgen, determiniseren we eerst de verkregen NFA. Dit doen we met de subset construction ([Hopcroft et al., 2000]).

Deze stap kan wel een exponentiële vergroting van de input opleveren, maar dit is worst-case optimaal.

Voorbeeld 11. Als we de taal $\{ww' \mid |w| = |w'| = n \text{ en } w \neq w'\}$ nemen over het alfabet $\Sigma = \{0, 1\}$. Deze taal bestaat uit alle woorden van even lengte waarvan de twee helften niet gelijk zijn.

Een reguliere expressie voor deze taal is $\bigvee_{i < n} (\Sigma^i 0 \Sigma^{n-i-1} \Sigma^i 1 \Sigma^{n-i-1} + \Sigma^i 1 \Sigma^{n-i-1} \Sigma^i 0 \Sigma^{n-i-1})$. De expressie geeft gewoon een unie van alle mogelijke plaatsen waarop de twee helften kunnen verschillen. Deze expressie is kwadratisch ten opzichte van n .

Een DFA moet echter alle informatie van het eerste woord bijhouden voor het die kan vergelijken met de informatie van het tweede woord. Dit levert ons een automaat op die exponentieel groot is ten opzichte van n .

Vervolgens minimaliseren we de geconstrueerde DFA. Dit doen we met equivalence-class construction ([Hopcroft et al., 2000]).

Voor deze beide algoritmes bestaan on-the-fly versies die in average-case beter presteren, maar deze algoritmes werken met optimale worst-case tijdscomplexiteit.

De automaat in Figuur 3.4 wordt na determinisatie en minimalisatie de automaat in Figuur 3.5.

3.4.2 Algoritme om te bepalen of een taal one-unambiguous is

Als de Glushkov automaat van een reguliere expressie een DFA is, is de expressie one-unambiguous. We willen nu een eigenschap vinden van deze DFA die bewaard blijft onder minimalisatie.

Als we willen nagaan of een taal one-unambiguous is, maken we een reguliere expressie E voor die taal. Deze hoeft niet one-unambiguous te zijn. Voor deze expressie construeren we dan de Glushkov-automaat. We determiniseren en minimaliseren die en kijken of die de eigenschap bezit.

Als de taal one-unambiguous is, moet deze automaat de eigenschap bezitten, want er is een expressie F , die one-unambiguous is en de taal beschrijft. De minimale deterministische automaat $MS(L(F))$ voor deze expressie heeft de eigenschap. Maar elke minimale deterministische automaat is uniek voor de taal en dus $MS(L(F)) = MS(L(E))$. De minimale deterministische automaat $MS(L(E))$ zal dus ook de eigenschap hebben.

Een dergelijke eigenschap die bewaard blijft onder minimalisatie, blijkt de structuur van orbits en gates te zijn. Deze zullen we dan verder ook gebruiken om te bepalen of een taal one-unambiguous is.

Definitie 12. Een orbit van een toestand q van een automaat M , aangeduid met $O(q)$, is de strongly connected component van q . Dit is de verzameling van toestanden die bereikbaar zijn vanuit q , en van waaruit q ook bereikt kan worden. Een orbit is triviaal als hij maar uit één toestand bestaat en er geen transities van q naar zichzelf zijn.

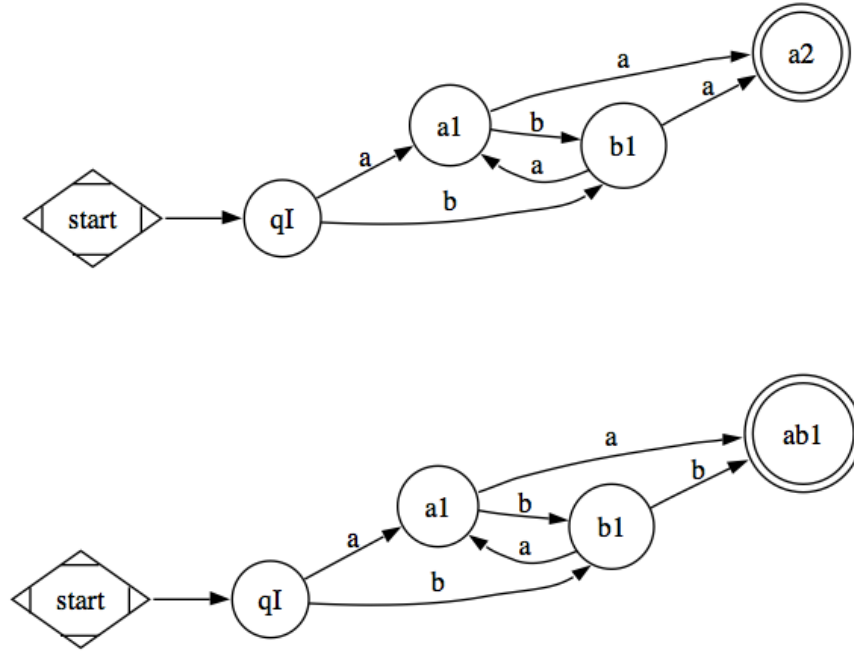
Voorbeeld 12. In Figuur 3.4 en Figuur 3.5 bestaan alle orbits van de automaten maar uit één toestand en zijn alle orbits behalve die van a_1 triviaal. In Figuur 3.6 hebben de automaten de volgende orbits: $\{q_I\}$, $\{a_1, b_1\}$, $\{a_2\}$. Hierbij zijn $\{q_I\}$ en $\{a_2\}$ telkens triviaal. Het symbool b_1 zit in $O(a_1)$, want je kan b_1 bereiken vanuit a_1 (met een b-transitie) en je kan dan vanuit b_1 weer a_1 bereiken (met een a-transitie).

Definitie 13. Een toestand q is een gate van zijn orbit $O(q)$ als het een eindtoestand is of als hij een transitie heeft naar een toestand buiten $O(q)$.

Voorbeeld 13. In Figuur 3.6 zijn alle toestanden gates van hun respectievelijke orbits. De toestanden q_I , a_1 en b_1 omdat ze transities hebben naar toestanden buiten hun orbit. De toestanden a_2 en ab_1 omdat ze eindtoestanden zijn. Moesten we de transitie vanuit b_1 naar de eindtoestand verwijderen, zou dit geen gate meer zijn, want dan had het enkel nog een transitie naar a_1 en dat is een element van $O(b_1)$.

Definitie 14. Een automaat M heeft de orbitProperty als alle gates van elke orbit op dezelfde manier met de andere orbits verbonden zijn. Meer specifiek moeten voor elke twee gates q_1 en q_2 van dezelfde orbit de volgende voorwaarden gelden:

1. q_1 en q_2 zijn ofwel allebei eindtoestanden, ofwel geen van beide eindtoestanden en,
2. voor elke toestand q buiten $O(q_1) = O(q_2)$ geldt dat er enkel een transitie (q_1, a, q) is als en slechts als er ook de transitie (q_2, a, q) is voor elk willekeurig symbool a .



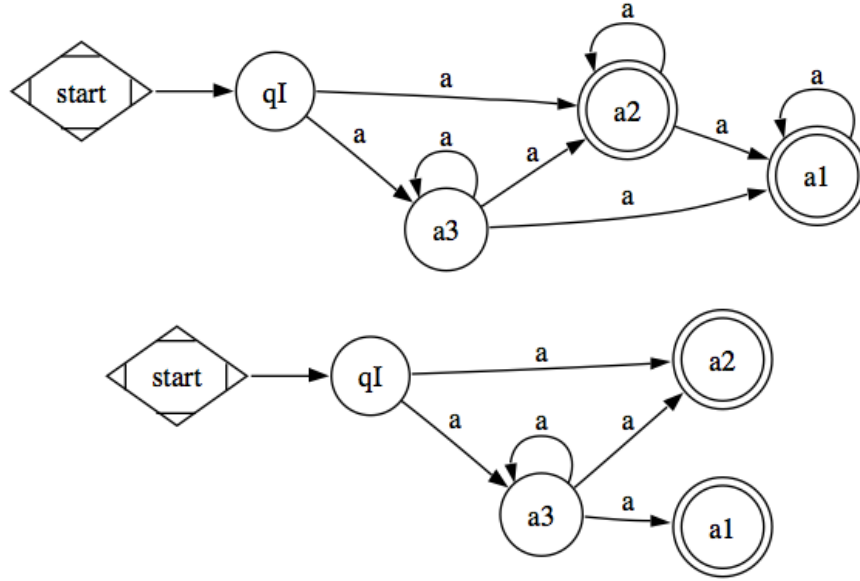
Figuur 3.6: Automaat: 1) met orbitProperty en 2) zonder orbitProperty.

Voorbeeld 14. In Figuur 3.4 en Figuur 3.5 hebben de automaten triviaal de orbitProperty omdat alle orbits maar één enkele gate hebben. In Figuur 3.6 1) heeft de automaat de orbitProperty. Hiervoor bestuderen we de enige orbit met meer dan één toestand, namelijk $\{a_1, b_1\}$. Toestanden a_1 en b_1 voldoen aan voorwaarde 1 omdat het allebei niet-eindtoestanden zijn. Ze voldoen ook aan voorwaarde 2, want zowel transitie (a_1, a, a_2) als transitie (b_1, a, a_2) zijn aanwezig in de automaat en er zijn geen andere transities naar toestanden buiten de orbit. De automaat in Figuur 3.6 2) heeft de orbitProperty niet. Hij voldoet wel aan voorwaarde 1, maar niet aan voorwaarde 2, want er is wel een transitie (a_1, a, ab_1) , maar geen transitie (b_1, a, ab_1) .

Om na te gaan of een automaat een one-unambiguous taal beschrijft, willen we een verdeel-en-heerstactiek toepassen. We willen automaten onderverdelen in automaten voor hun orbits en dan de eigenschappen van die automaten nagaan. Transities die vertrekken uit de eindtoestanden zouden verschillende subautomaten (voor expressies onder een ster-operator) kunnen verenigen tot één orbit, waar we die wel zouden willen opsplitsen. We willen dus deze transities kunnen verwijderen.

Definitie 15. Een symbol a in Σ is M -consistent als er een toestand $f(a)$ is zodat alle eindtoestanden van M een a -transitie hebben naar $f(a)$.

Definitie 16. Als M een NFA en S een verzameling symbolen is, dan is de S -cut M_S van M , de NFA gemaakt door van M , voor elke a in S alle a -transities vertrekkende uit een eindtoestand van M te verwijderen.



Figuur 3.7: Automaat: 1) met a consistent en 2) de $\{a\}$ -cut van 1).

Voorbeeld 15. In Figuur 3.7 1) is het symbool a M -consistent, want vanuit elke eindtoestand is er een a -transitie naar $f(a) = a_1$. Figuur 3.7 2) toont de $\{a\}$ -cut van M , waarin we alle a -transities vertrekkende uit a_1 of a_2 verwijderd hebben.

Definitie 17. Voor een toestand q van M , definiëren we de orbit automaat M_q van q als de automaat met:

- als toestanden $O(q)$,
- als starttoestand q en,
- als eindtoestanden de gates van $O(q)$.

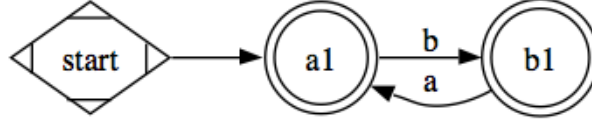
De taal van deze automaat noemen we de orbit-taal.

Voorbeeld 16. De orbit automaat voor de toestand a_1 bij de automaat uit Figuur 3.6 1) is weergegeven in Figuur 3.8.

Stelling 4. Als M een minimale DFA is, dan is $L(M)$ one-unambiguous als en slechts als M de orbitProperty heeft en alle orbit-talen van M one-unambiguous zijn.

Voor het bewijs verwijzen we naar [Brüggemann-Klein and Wood, 1998].

In paragraaf 3.3.2 maakten we een reguliere expressie trim. Dit feit laat ons toe om een natuurlijke overeenkomst te bekijken tussen een trim reguliere expressie en zijn Glushkov automaat.



Figuur 3.8: Orbit automaat voor a_1 uit Figuur 3.6 1).

De toestanden van de Glushkov automaat M_E zijn gelijk aan $\text{sym}(E')$ (zie paragraaf 3.3.2) samen met de nieuwe starttoestand. Als H nu een subexpressie is van E , dan is $\text{sym}(H') \subseteq \text{sym}(E')$. Als er een $x, y \in \text{sym}(H')$ waarvoor $y \in \text{follow}(H', x)$, dan volgt hieruit dat $y \in \text{follow}(E', x)$, als E trim is. Hieruit volgt dat de automaat van H een subautomaat is van die van E . Het is echter belangrijk dat E trim is en dus geen nutteloze toestanden bevat. Anders kan een transitie in M_H niet voorkomen in M_E omdat de transitie naar een nutteloze toestand zou leiden.

Voorbeeld 17. Neem $E = aa\emptyset$ en $H = aa$. M_H heeft twee transities, want, $\text{first}(H') = \{a_1\}$ en $a_2 \in \text{follow}(H', a_1)$. Maar M_E heeft geen transities, want aangezien de taal geen woorden bevat, is de follow-verzameling leeg.

Dus door het trim zijn van de expressie kunnen we orbits en gates nu karakteriseren als volgt:

Stelling 5. *Stel, E is een trim reguliere expressie en $x \in \text{sym}(E')$. Dan karakteriseren we de orbits en gates in M_E als volgt:*

1.
 - Als er geen subexpressie H^* is waarvoor $x \in \text{sym}(H')$, dan is $O(x) = \{x\}$ en dan is dit een triviale orbit².
 - Als H de maximale subexpressie onder een $*$ is waarvoor $x \in \text{sym}(H')$, dan $O(x) = \text{sym}(H')$ en is deze orbit nooit triviaal³.
 - De orbit van de starttoestand is triviaal.
2. Als H een maximale subexpressie onder een $*$ is, met $\text{sym}(H') \neq \emptyset$, dan zijn de elementen van $\text{last}(H')$ de gates van de orbit van $\text{sym}(H')$. Elke transitie in de volledige automaat M_E tussen toestanden in $\text{sym}(H')$ is al aanwezig in M_H .

Deze stelling geeft de intuïtie weer dat we de orbits van een Glushkov automaat van E kunnen interpreteren als subautomaten voor subexpressies van E . In het geval van een ster-operator zijn er dan ook terugkerende transities. We hebben enkel maximale subexpressies onder een ster-operator bekeken, maar we kunnen indien gewenst de Glushkov automaat van deze subexpressies maken waarvoor deze stelling weer opgaat.

²Er is dus geen transitie van x naar zichzelf.

³Als x het enige element is, is er een transitie van x naar zichzelf.

Bewijs. Dit bewijzen we door inductie op de structuur van de expressie:

Basis:

We onderscheiden twee basisgevallen:

- de expressie is ϵ of $\emptyset$,
- de expressie is van de vorm x_i met x_i een symbool uit het alfabet.

Als de expressie van de vorm ϵ of $\emptyset$ is, dan heeft de Gushkov automaat de starttoestand als enige toestand⁴ en heeft deze geen transities. Hiervoor zijn de beweringen triviaal waar.

Als de expressie van de vorm x_i is, dan heeft de Glushkov automaat een enkele transitie, namelijk van de starttoestand naar x_i en is x_i een eindtoestand. Er zijn dus twee aparte orbits en deze zijn beide triviale orbits. Ook hier zijn de uitspraken dus triviaal waar.

Inductiehypothese: De uitspraak is geldig voor expressies E_1 en E_2 .

Te bewijzen: De uitspraak is geldig voor $E_1 \cdot E_2$, $E_1 + E_2$ en E_1^* .

We tonen hier enkel het interessantste geval aan, namelijk dat de uitspraak geldig is voor E_1^* .

- E_1^* : De automaat heeft de transities voor E_1 , met transities toegevoegd gaande van $last(E'_1)$ naar $first(E'_1)$.
 1. – We kunnen steeds zo'n maximale subexpressie H^* vinden, namelijk E_1^* zelf.
 - Aangezien E trim is, zijn er geen nutteloze toestanden in M_{E_1} . Dit wil zeggen dat alle toestanden op een pad liggen van de starttoestand naar een eindtoestand. Dit pad wordt bepaald door de follow-informatie voor E'_1 . Dit wil ook zeggen dat er een pad is van x naar een element van $last(E'_1)$. Vanuit deze toestand is er een transitie naar elke toestand in $first(E'_1)$. En van hieruit is er dus een pad naar elke toestand. Dezelfde redenering gaat op voor elke toestand zodat er voor elke toestand een pad is naar x . Hieruit volgt dat $O(x) = sym(H')$.
Deze orbit zal ook nooit triviaal zijn, want als x de enige toestand is buiten de starttoestand, zal de constructie van de automaat voor E_1^* ervoor zorgen dat er een transitie van x naar zichzelf wordt toegevoegd.
 - De orbit van de starttoestand is altijd triviaal, want er zijn door de definitie van de Glushkov automaat nooit transities die teruggaan naar de starttoestand.
 2. E_1 is een maximale subexpressie onder een $*$ met $sym(H') \neq \emptyset$, want ze bevat x . Aangezien er geen toestanden buiten de orbit zijn, zijn de gates volgens hun definitie de eindtoestanden. En deze zijn volgens de definitie van de Glushkov automaat gelijk aan $last(H')$.
Aangezien $M_E = M_{H^*}$ zitten alle transities tussen toestanden in $sym(H')$ ook al in M_E .

Voor het volledige bewijs, zie Appendix A. □

⁴Deze transitie is respectievelijk wel en niet een eindtoestand.

Met behulp van onder andere Stelling 4 komen we tot de volgende functie die we kunnen toepassen op de minimale DFA uit het vorige deel. Vervolgens worden de werking en implementatiedetails van de aparte onderdelen besproken. Voor de correctheid van dit algoritme verwijzen we naar [Brüggemann-Klein and Wood, 1998]:

```
function one-unambiguous(M: minimal DFA): Boolean;
  Bereken  $S = \{a \in \Sigma \mid a \text{ is } M \text{ consistent}\}$ ;
  if ( M heeft een enkele, triviale orbit ) then return true;
  if ( M heeft een enkele, niet-triviale orbit en  $S = \emptyset$  ) then return false;
  Bereken de orbits van  $M_S$ ;
  if ( !OrbitProperty( $M_S$ ) ) then return false;
  for ( elke orbit K van  $M_S$  ) do
    kies een x uit K;
    if ( !one-unambiguous( $(M_S)_x$ ) ) then return false;
  end
  return true;
```

De meeste onderdelen van deze functie zijn redelijk triviaal te implementeren, zoals het berekenen van de verzameling consistente symbolen of het nagaan of M uit een enkele triviale orbit bestaat. Er is echter een onderdeel van deze functie dat niet zo eenvoudig te implementeren is, namelijk het bepalen van de orbits.

Orbits of strongly connected components

Als we de orbits in een automaat interpreteren als de strongly connected components in een gerichte graaf, dan kunnen we reeds bestaande algoritmes gebruiken om deze te berekenen. We noemen een gerichte graaf $G = (V, E)$, met zijn verzameling knopen V , en een verzameling edges E .

We gebruiken een algoritme uit [Nuutila and Soisalon-Soininen, 1994] voor het bepalen van de orbits. We geven hier algoritme 1 uit deze paper, dit kan nog verbeterd worden (zoals ook aangegeven in de paper), maar dit heeft optimale worst-case tijdscomplexiteit:

```
begin
  stack =  $\emptyset$ ;
  for elke knoop v in V do
    if v is nog niet bezocht VISIT(v);
  end
```

We bezoeken dus alle knopen met de $VISIT()$ -functie. Deze functie maakt gebruik van de $MIN(x, y)$ -operatie. $MIN(x, y)$ heeft twee knopen x en y als input en geeft terug welk van de twee knopen als eerste werd bezocht door de $VISIT()$ -functie. Dit kan geïmplementeerd worden met behulp van een array en door een depth-first getal toe te kennen aan elke knoop die bezocht wordt.

De functie *VISIT()* ziet er uit als volgt:

```
1 procedure VISIT(v):
2 begin
3   root[v] = v;
4   InComponent[v] = false;
5   PUSH(v, stack);
6   for elke knoop w zodat (v, w) in E zit do begin
7     if w is nog niet bezocht then VISIT(w);
8     if !InComponent[w] then root[v] = MIN(root[v], root[w]);
9   end
10  if root[v] == v then
11    repeat
12      w = POP(stack);
13      InComponent[w] = true;
14    until w == v
15 end
```

De intuïtieve redenering achter dit algoritme om de strongly connected components te bepalen is als volgt:

Voor elke knoop houden we bij wat de (verwachte) root is van zijn strongly connected component. Dus als het algoritme beëindigd is, dan behoren de knopen met dezelfde root tot dezelfde component.

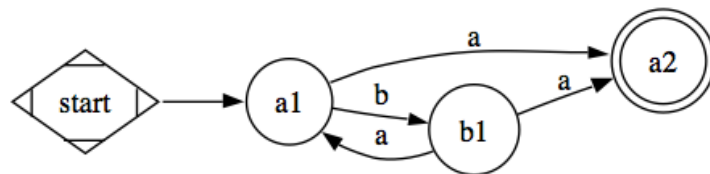
Het hoofdalgoritme zorgt ervoor dat alle knopen zeker bezocht worden door de *VISIT()*-functie. Het initialiseert ook de stack die in *VISIT()* gebruikt gaat worden.

Regels 3-5 van de *VISIT()*-functie zorgen voor een aantal initialisaties als de knoop voor het eerst bezocht wordt:

1. *root[v]* bevat de op dat moment geschatte root van de component van de knoop *v*. Als we de knoop *v* voor het eerst bezoeken, schatten we dat hij zelf de root van zijn component wordt.
2. *InComponent[v]* bevat een boolean die aangeeft of de hele component van deze knoop al gevonden is. Als we de knoop *v* voor het eerst bezoeken, zetten we deze boolean op false.
3. We pushen de knoop *v* op de stack.

In deze initialisatiefase moeten we er ook voor zorgen dat de *MIN()*-functie naar behoren kan werken door een depth-first getal bij te houden. Ook moeten we bij elke knoop bijhouden of hij al bezocht is, want dit wordt gebruikt in het hoofdalgoritme.

Regels 6-7 zorgen ervoor dat alle knopen bereikbaar vanuit elke knoop, bezocht worden in een depth-first volgorde. We bezoeken alle knopen rechtstreeks bereikbaar vanuit *v*, als deze nog niet bezocht zijn.



Figuur 3.9: Bereikbare automaat voor a_1 uit Figuur 3.6 1).

In regels 8-9 gaat de *VISIT()*-functie na of de bereikbare knopen voor deze knoop een lagere root-kandidaat hebben. Als dit zo is, maakt ze daarvan de nieuwe root-kandidaat voor de huidige knoop. Als een kind een lagere root-kandidaat heeft (en zijn volledige component nog niet ontdekt is), wil dit zeggen dat dit kind al bezocht is en je datzelfde kind nu opnieuw tegenkomt, dus per definitie dat je in dezelfde component zit (en deze component kan uitbreiden als die knoop dan weer een nog vroeger bezochte knoop tegenkomt enz.).

Als alle kinderen bezocht zijn door de *VISIT()*-functie, staat de root-kandidaat van een knoop enkel op zichzelf als hij de root van de component is, anders had hij een knoop met een lagere root-kandidaat moeten tegenkomen.

Regels 10-14 gebruiken dit feit. Namelijk, als een knoop die zijn eigen root is, bereikt wordt, dan is zijn component volledig gedetecteerd en dan staan de knopen daarvan boven op de stack. Eventuele andere componenten die onderweg zijn tegengekomen, zijn allemaal bovenop de stack gekomen en daar ook al weggehaald in deze stap.

De knopen van deze component worden dan van de stack gehaald en met behulp van hun *inComponent*-boolean wordt aangegeven dat hun component volledig gedetecteerd is.

Dit algoritme is lineair in de grootte van de graaf (het aantal knopen samen met het aantal edges, of voor de automaat het aantal toestanden samen met het aantal transities van de automaat).

3.5 Een equivalente one-unambiguous reguliere expressie maken

We hebben nu een minimale DFA voor een taal waarvan we weten dat ze one-unambiguous is. Nu rest er ons nog een one-unambiguous reguliere expressie te construeren die deze taal beschrijft.

Definitie 18. De automaat M^q is de automaat verkregen uit M met als toestanden enkel de toestanden die bereikbaar zijn uit q en met q als starttoestand. We noemen deze automaat de bereikbare automaat.

Voorbeeld 18. De bereikbare automaat voor de toestand a_1 bij de automaat uit Figuur 3.6 1) is weergegeven in Figuur 3.9.

Stelling 6. *Als $L(M)$ one-unambiguous is, kunnen we een one-unambiguous reguliere expressie die $L(M)$ beschrijft vormen uit one-unambiguous reguliere expressies voor de orbittalen.*

Voor het bewijs van deze stelling verwijzen we naar [Brüggemann-Klein and Wood, 1998]. We kunnen de reguliere expressie bepalen uit de orbittalen op de volgende recursieve manier:

- Als je een enkele triviale orbit hebt, is de reguliere expressie voor deze DFA ϵ .
- Als er minstens één symbool zit in S , de verzameling van M -consistente symbolen, dan is de reguliere expressie voor M van de vorm:

$$E_S \cdot (\oplus_{a \in S} a E_a)^*$$

Hierbij is E_S de reguliere expressie voor M_S en E_a zijn de REs voor $M_S^{f(a)}$ voor elke a in S .

De automaat M_S kunnen we construeren uit de DFA door voor elk symbool in S , elke transitie uit een eindtoestand met dat symbool te verwijderen.

De automaat $M_S^{f(a)}$ kunnen we construeren door uit M_S de toestanden (en transities) te verwijderen die niet vanuit $f(a)$ bereikbaar zijn. Hiervoor kunnen we gebruik maken van de informatie die de analyse van de orbits heeft opgeleverd.

Op deze automaten roepen we recursief ons algoritme aan om de nodige reguliere expressies te verkrijgen.

- Als er geen M -consistente symbolen zijn, heeft M meerdere orbits. Dit volgt uit het algoritme om te bepalen of de taal one-unambiguous is: als we een enkele orbit hadden en een lege verzameling M -consistente symbolen, had het algoritme aangegeven dat de taal niet one-unambiguous was.

De reguliere expressie voor M is dan van de vorm:

$$E_{q_I} \cdot (\oplus_{1 \leq i \leq n} a_i E_i)$$

Hierbij is E_{q_I} de reguliere expressie voor M_{q_I} , E_i zijn de expressies voor M^{q_i} , a_i zijn de symbolen van de transities die uit de orbit van q_I vertrekken en q_i zijn de toestanden waar deze transities naartoe gaan.

Als de automaat M_{q_I} eindtoestanden bevat, moeten we aan deze unie ook nog ϵ toevoegen. Dit wordt niet aangegeven in de formule uit [Brüggemann-Klein and Wood, 1998].

De automaat M_{q_I} bekomen we door als toestanden de orbit van q_I te nemen, met q_I als starttoestand en de gates van de orbit als eindtoestanden. Deze informatie hebben we al berekend bij het analyseren van de orbits.

De symbolen a_i kunnen we ook uit vorige informatie halen en daarmee bepalen we de q_i . De automaat M^{q_i} kunnen we construeren door uit M de toestanden (en transities) te verwijderen die niet vanuit q_i bereikbaar zijn. Hiervoor kunnen we gebruik maken van de informatie die de analyse van de orbits heeft opgeleverd.

Op deze automaten roepen we recursief ons algoritme aan om de nodige reguliere expressies te verkrijgen.

Het idee achter dit algoritme is als volgt:

- een lege automaat geeft de lege string,
- als er minstens één M -consistent symbool is, kregen we $E_S \cdot (\oplus_{a \in S} a E_a)^*$. Dit wil zeggen dat we eerst de expressie geven voor één iteratie van de automaat en vervolgens voor elk symbool aangeven wat er nog kan volgen.
- als er geen M -consistente symbolen zijn, kregen we $E_{q_I} \cdot (\oplus_{1 \leq i \leq n} a_i E_i)$. Dit betekent dat we de subexpressie voor de beginorbit maken en dan voor elk symbool de expressies toevoegen voor de orbit waar naartoe gegaan kan worden. Als de orbit automaat van de starttoestand eindtoestanden bevat, dan kan er na de eerste subexpressie ook de lege string volgen, dus moeten we dan ook epsilon aan de unie toevoegen.

Aangezien we een one-unambiguous reguliere expressie maken voor $L(M)$ uit ten hoogste $(k+1)$ expressies voor kleinere automaten met k het aantal orbits, is het algoritme exponentieel in de grootte van M .

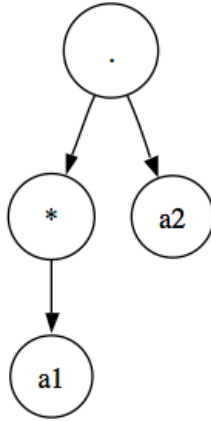
Voorbeeld 19. We nemen de taal $\Sigma^* w$ met $|w| = w_1 \dots w_n$. Deze taal aanvaardt alle woorden die eindigen met w .

Voor deze taal kunnen we een DFA voor maken met lineaire grootte ten opzichte van n . De eerste toestand heeft voor elk symbool behalve w_1 een transitie naar zichzelf. Deze toestand hoort bij de steroperator. Voor elk symbool w_i voor $1 \leq i < n$ gaat er een transitie van zijn toestand naar een volgende toestand voor het volgende symbool en voor alle andere symbolen terug naar de ster-toestand. De toestand horende bij w_n is de eindtoestand en heeft voor elk symbool een transitie terug naar de ster-toestand. Voor elk symbool houdt de automaat dus bij dat hij ofwel in de steroperator ofwel in w zit. Welk van de twee het is, wordt besloten met behulp van het volgende symbool (of het ontbreken daarvan).

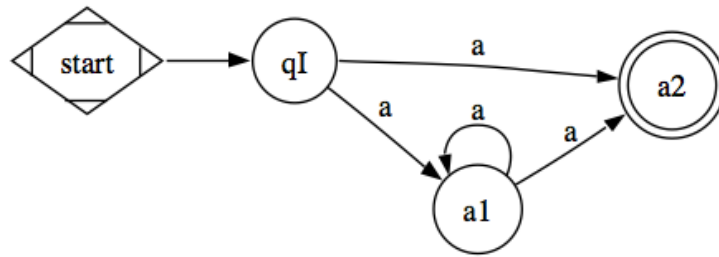
Een one-unambiguous expressie voor deze taal mag echter per definitie niet naar het volgende symbool in het woord kijken om te beslissen bij welk symbool uit de expressie het hoort. Een expressie voor deze taal moet rekening houden met alle mogelijke routes die we naar de eindtoestand kunnen volgen. Dit levert ons een exponentieel grote expressie op.

3.6 Voorbeelduitwerking van het hele algoritme

Stel, we krijgen de reguliere expressie a^*a , als invoer. Deze zetten we dan eerst om in een markering $a_1^*a_2$, waarna we deze omzetten in zijn syntaxboom (zie paragraaf 3.3.1). Deze syntaxboom is weergegeven in Figuur 3.10.



Figuur 3.10: Syntaxboom voor de markering $a_1^*a_2$.



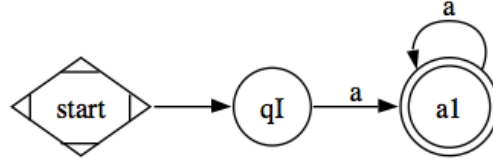
Figuur 3.11: Glushkov automaat voor de reguliere expressie a^*a .

De expressie staat al in ster-normaalvorm en is al trim, dus we berekenen de first-, last- en followverzamelingen (zie paragraaf 3.3.2):

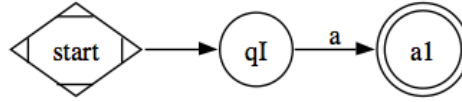
- $first(E') = \{a_1, a_2\}$,
- $last(E') = \{a_2\}$,
- – $follow(E', a_1) = \{a_1, a_2\}$,
- $follow(E', a_2) = \emptyset$.

Dit geeft ons de Glushkov automaat uit Figuur 3.11.

Deze automaat is niet deterministisch, er is bijvoorbeeld een a -transitie vanuit de starttoestand naar a_1 en a_2 . De originele expressie was dus niet one-unambiguous. We gaan dus verder met het algoritme. De gedeterminiseerde automaat is te zien in Figuur 3.12. Deze resulterende automaat is ook minimaal (zie paragraaf 3.4.1).



Figuur 3.12: Gedetermineerde en geminimaliseerde Glushkov automaat voor de reguliere expressie a^*a .



Figuur 3.13: De a -cut van de gedetermineerde en geminimaliseerde Glushkov automaat voor de reguliere expressie a^*a .

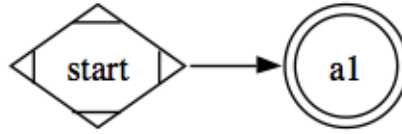
We gaan nu na of de taal van de expressie one-unambiguous is (zie paragraaf 3.4). $S = \{a\}$. We berekenen de orbits van de a -cut van de automaat (te zien in Figuur 3.13), deze zijn duidelijk $\{qI\}$ en $\{a_1\}$. Aangezien de orbits maar één toestand bevatten, voldoet de automaat triviaal aan de orbitProperty. De taal is dus one-unambiguous als de talen van de orbit automaten one-unambiguous zijn. Deze orbit automaten bestaan uit een enkele triviale orbit en hun talen zijn dus one-unambiguous.

Nu rest er ons enkel nog een one-unambiguous reguliere expressie te construeren die de taal $L(a^*a)$ beschrijft (zie paragraaf 3.5).

Aangezien $S = \{a\}$, is de expressie van de vorm $E_S \cdot (a \cdot E_a)^*$.

- E_S is de expressie voor de automaat uit Figuur 3.13. Deze automaat heeft geen M -consistente symbolen. E_S is dus van de vorm $E_{qI} \cdot (aE_{a_1})$.
 - E_{qI} is de expressie voor de automaat met enkel qI . Deze is triviaal en heeft dus ϵ als reguliere expressie.
 - E_{a_1} is de expressie voor de automaat weergegeven in Figuur 3.14. Deze automaat bestaat uit een enkele triviale orbit en heeft dus ϵ als reguliere expressie.
- E_a is de expressie voor de automaat weergegeven in Figuur 3.14. Deze automaat bestaat uit een enkele triviale orbit en heeft dus ϵ als reguliere expressie.

Als we dit nu samenvoegen, krijgen we als resulterende one-unambiguous reguliere expressie $E_S \cdot (a \cdot E_a)^* = E_{qI} \cdot (aE_{a_1}) \cdot (a \cdot \epsilon)^* = \epsilon \cdot a \cdot \epsilon \cdot (a \cdot \epsilon)^*$. Of na het verwijderen van overbodige concatenaties met ϵ krijgen we aa^* .



Figuur 3.14: Bereikbare automaat voor a_1 van de a -cut van de gedeterminiseerde en geminimaliseerde Glushkov automaat voor de reguliere expressie a^*a .

3.7 Mogelijke verbeteringen op het algoritme

Er zijn enkele elementen die we nog zouden willen verbeteren aan het algoritme.

Zoals bleek in paragraaf 3.6 is het resultaat van het algoritme vaak groter dan nodig. Daar bleek dat het algoritme vaak concatenaties met ϵ toevoegt, ook al zijn deze nutteloos. Ook levert het algoritme vaak constructies op van de vorm $aX + bX$ met X een willekeurige expressie, die korter weergegeven kan worden als $(a + b)X$. Hoe we een reguliere expressie kunnen omzetten in een kleinere equivalente reguliere expressie wordt verder besproken.

In [Han and Wood, 2007] wordt het idee aangebracht om automaten op te splitsen bij bridge toestanden. Daar wordt dit gebruikt om een zo klein mogelijke expressie te verkrijgen met state elimination. We zouden hier iets gelijkaardigs kunnen doen bij ons algoritme. We zouden op bridge toestanden kunnen nagaan of het opsplitsen van de automaat de one-unambiguity niet in gevaar brengt. Als dit niet zo is, zouden we met het originele algoritme een expressie verkregen hebben van de vorm $aX + bX$, maar door de automaat op te splitsen kunnen we de kortere expressie $(a + b)X$ bekomen.

Dit soort van mogelijke opsplitsing is ook al te detecteren in de syntaxboom. We zouden een aantal regels kunnen opstellen voor elke operator, zodat, als een knoop aan die regels voldoet, het algoritme enkel op de subexpressies aangeroepen dient te worden. Aangezien de tijdsduur van het algoritme sterk afhangt van het aantal knopen in de syntaxboom, zou dit een serieuze tijdsduurwinst kunnen opleveren.

Er zijn ook talen waarvoor geen one-unambiguous reguliere expressie is, hiervoor zouden we misschien een one-unambiguous expressie willen bekomen die een taal beschrijft die de originele taal bevat en zo weinig mogelijk extra.

Deel II

Reguliere expressies inkorten

Hoofdstuk 4

Kortere reguliere expressies vinden

4.1 Inleiding

In paragraaf 3.7 merkten we op dat het algoritme zoals beschreven in Hoofdstuk 3 vaak reguliere expressies teruggaf die langer waren dan nodig. Dit is iets wat met automatisch gegenereerde expressies een probleem kan vormen. In principe zijn dit geen foute oplossingen, maar kortere reguliere expressies hebben een aantal voordelen:

- het belangrijkste voordeel is dat kleinere expressies leesbaarder zijn voor mensen.
- het is ook mogelijk dat een kortere reguliere expressie one-unambiguous is, waar de langere het niet is. Het korter maken van een expressie vermindert in het algemeen het aantal voorkomens van een bepaald symbool, dus is er minder kans dat deze symbolen conflicteren. In het extreme geval, waar we een expressie vinden waarin elk symbool hoogstens één keer voorkomt, is deze triviaal one-unambiguous.

Voorbeeld 20. De expressie $ab + ac$ is niet one-unambiguous, maar de kortere equivalente expressie $a(b + c)$ is wel one-unambiguous.

Omwille van de voorgenoemde redenen zouden we een algoritme willen ontwerpen dat een reguliere expressie omzet in een kortere reguliere expressie. Dit algoritme zou dan als pre- en als postprocessingstap voor het algoritme in Hoofdstuk 3 gebruikt kunnen worden. De duur van dat algoritme hangt sterk af van de lengte van de input-expressie. Daarom is het zinvol vooraf aandacht te besteden aan het korter maken van de expressie. Ook is het mogelijk dat we de expressie door het korter maken al one-unambiguous maken.

4.2 Naïeve implementatie

In principe is het mogelijk om voor een expressie alle mogelijke expressies die korter zijn op te sommen en vervolgens na te gaan of ze equivalent is met de originele expressie. Dit zou ons toelaten om de kleinst mogelijke expressie te vinden die equivalent is met de originele expressie. Dit algoritme is echter van geen praktisch nut, omdat er immens veel expressies bijkomen bij elk verhogen van de lengte, waardoor deze strategie snel onhaalbaar wordt.

4.3 Lengtematen

Om het te kunnen hebben over een kortere reguliere expressie, moeten we eerst weten wat de lengte van een reguliere expressie is ([Gelade et al., 2007], [Lee and Shallit, 2004]).

Definitie 19. *De lengte van een reguliere expressie E over Σ is gelijk aan het aantal knopen in zijn syntaxboom, zijnde het aantal symbolen plus het aantal operatoren dat voorkomt in de expressie.*

Voorbeeld 21. De expressie $a.a.(b + c)^*$ heeft lengte 8. Let hierbij op dat, als we deze expressie anders schrijven als $(a)a((b + c))^*$, de lengte van de expressie onveranderd blijft. We bekijken hier enkel de structuur van de expressie en niet de manier waarop we deze schrijven. Het is natuurlijk wel vaak aan te raden voor de leesbaarheid om overbodige haakjes en concatenatie-operatoren te vermijden.

Dit is de lengtemaat die we verder zullen hanteren. Er is geen algemene overeenkomst van wat de lengte van een reguliere expressie precies is. Er zijn nog andere maten die we kunnen gebruiken om aan te geven hoe complex een reguliere expressie is:

- het totaal aantal symbolen in de expressie (inclusief haakjes) ([Lee and Shallit, 2004])
- het aantal symbolen uit Σ in de expressie,
- de diepste nesting van de ster-operator,
- de lengte van het langste niet-herhalende pad door de expressie,
- het grootste aantal symbolen in een unie-operator.

Voor de precieze inductieve definities van de vier laatste complexiteitsmaten verwijzen we naar [Ehrenfeucht and Zeiger, 1976].

Een reguliere expressie heeft zelden een unieke kleinste reguliere expressie. Het omwisselen van de volgorde van een unie-operator verandert bijvoorbeeld niets aan de lengte, maar levert wel een andere expressie op.

In de verdere hoofdstukken zullen we onderzoeken hoe we een reguliere expressie kunnen omvormen in een zo klein mogelijke equivalenten reguliere expressie.

Hoofdstuk 5

Herschrijfregels voor reguliere expressies

5.1 Inleiding

Het doel van dit hoofdstuk is een aantal heuristieken vinden die expressies in kleinere expressies omzetten. We zouden graag een verzameling van regels bekomen die ons toestaat om elke reguliere expressie om te zetten in een zo klein mogelijke equivalente reguliere expressie.

We kunnen een aantal gelijkheidsregels opstellen, deze noemen we axioma's. In deze gelijkheid kunnen we dan elk voorkomen van eenzelfde symbool vervangen door eenzelfde subexpressie, of door subexpressies die gelijk zijn. Deze gelijkheid is dan ook geldig. Dit noemen we de substitutie-regel. Deze regel zullen we vaak toepassen zonder deze expliciet te vermelden.

Voorbeeld 22. Als we de gelijkheden $E = E + E$ en $F + G = G + F$ hebben, kunnen we hieruit, door E gelijk te stellen aan ab , de gelijkheid $ab = ab + ab$ afleiden. We kunnen ook de gelijkheid $a + b = (a + b) + (b + a)$ afleiden.

Het is echter niet mogelijk om met behulp van enkel een eindig aantal gelijkheidsregels en de substitutie-regel elke reguliere expressie om te zetten in elke equivalente reguliere expressie. Hieruit volgt dan ook dat we niet elke expressie kunnen omzetten in een zo klein mogelijke equivalente expressie. Als we dit wel konden, hadden we regels die, als we ze in de andere richting uitvoeren, ons elke equivalentie zouden laten aantonen.

We hebben dus nog iets extra nodig. Salomaa stelt in [Salomaa, 1966] twee mogelijke systemen voor die toelaten alle equivalenties tussen reguliere expressies te bewijzen. We onderzoeken nu hoe we het eerste hiervan praktisch kunnen benutten om een expressie om te vormen in een zo klein mogelijke equivalente expressie.

5.2 Axiomasysteem F1

We bespreken hier het axioma-systeem F1 uit [Salomaa, 1966]. Dit systeem maakt gebruik van twee regels:

1. de *substitutieregel* zoals besproken in paragraaf 5.1,
2. het *oplossen van vergelijkingen*, die zegt dat als we $E = EF + G$ kunnen aantonen waarbij $L(F)$ niet ϵ bevat, dan kunnen we hieruit $E = GF^*$ afleiden.

Buiten deze regels maakt het gebruik van elf axioma's¹:

$$E + (F + G) = (E + F) + G \quad (5.1)$$

$$E(FG) = (EF)G \quad (5.2)$$

$$E + F = F + E \quad (5.3)$$

$$EF + EG = E(F + G) \quad (5.4)$$

$$EG + FG = (E + F)G \quad (5.5)$$

$$E + E = E \quad (5.6)$$

$$\epsilon E = E \quad (5.7)$$

$$\emptyset E = \emptyset \quad (5.8)$$

$$E + \emptyset = E \quad (5.9)$$

$$\epsilon + E^*E = E^* \quad (5.10)$$

$$(\epsilon + E)^* = E^* \quad (5.11)$$

Definitie 20. Een bewijs in F1 is een lijst van gelijkheden waarbij elke gelijkheid ofwel een axioma is, ofwel volgt uit ervoor voorkomende gelijkheden, gebruik makend van één van de twee regels.

Definitie 21. Een gelijkheid is afleidbaar in F1 als er een bewijs is dat eindigt met deze gelijkheid.

Voorbeeld 23. Hieronder vinden we een bewijs voor de gelijkheid $EE^* = E^*E$. De axioma's zelf zijn hier voor de overzichtelijkheid weggelaten. Elke regel volgt uit de voorgaande door gebruik te maken van een axioma of een regel die erachter weergegeven staat:

$$\begin{aligned} \vdash E^*E &= E^*E \\ \vdash E^*E &= (\epsilon + E^*E)E \quad (\text{axioma 5.10}) \\ \vdash E^*E &= \epsilon E + E^*EE \quad (\text{axioma 5.5}) \\ \vdash E^*E &= E + E^*EE \quad (\text{axioma 5.7}) \\ \vdash E^*E &= E^*EE + E \quad (\text{axioma 5.3}) \\ \vdash E^*E &= EE^* \quad (\text{regel 2}) \end{aligned}$$

¹In de onderstaande axioma's hebben we de originele $\emptyset^*$ zoals ze voorkwamen in [Salomaa, 1966] vervangen door de equivalente ϵ .

Definitie 22. We zeggen dat een gelijkheid $E_1 = E_2$ geldig is, als $L(E_1) = L(E_2)$.

Voorbeeld 24. Als a en b symbolen zijn uit Σ , dan is de gelijkheid $a + b = b + a$ geldig, de gelijkheid $a = b$ echter niet.

Definitie 23. We zeggen dat een axiomasysteem consistent is, als alle gelijkheden die we eruit kunnen afleiden geldig zijn.

Definitie 24. We zeggen dat een axiomasysteem volledig is als alle geldige vergelijkingen tussen expressies eruit afgeleid kunnen worden.

Voor de bewijzen van de consistentie en volledigheid van het axiomasysteem F1 verwijzen we naar [Salomaa, 1966].

5.3 Implementatie

We hebben nu een verzameling vergelijkingen en twee regels waarmee we in principe elke vergelijking in elke equivalente vergelijking kunnen omzetten. Dit wil dus zeggen dat we ook voor elke vergelijking een zo klein mogelijke equivalente vergelijking kunnen vinden.

Voor elke expressie is er een eindig aantal axioma's dat we met behulp van de substitutieregel kunnen toepassen op elke subexpressie. We kunnen deze één voor één afgaan en deze toepassen indien mogelijk. Vervolgens kunnen we dit blijven herhalen op de nieuwe expressie in de hoop een zo klein mogelijke expressie te bekomen. Het probleem met deze methode is dat we regels zoals regel 5.6 op elke mogelijke subexpressie kunnen toepassen en dat deze de expressie dus bij elke stap zou vergroten. We zouden dus een immens aantal mogelijkheden hebben dat bij elke stap alleen maar zou vergroten. Aangezien we niet weten wat de minimale lengte is voor onze expressie, zouden we ook geen idee hebben wanneer te stoppen, behalve na een bepaald aantal stappen.

Deze implementatie zou dus geen praktisch nut hebben. We merken wel op dat de gelijkheden 5.4-5.11 elk een duidelijk langere en kortere kant hebben (respectievelijk links en rechts). We kunnen deze regels dus wel gebruiken om op een greedy manier de syntaxboom van onze expressie te verkleinen. Deze methode heeft dezelfde nadelen als de meeste greedy methodes: soms moeten we een expressie eerst langer maken om ze korter kunnen maken dan ze was.

Dit wil dan ook zeggen dat we niet altijd de juiste oplossing krijgen bij ons algoritme, maar dat we gewoon een verbetering uitvoeren. Een groot voordeel van deze methode is dat, aangezien de expressie in elke stap enkel kleiner zal worden, het algoritme gegarandeerd stopt, en het resultaat zal nooit een langere expressie zijn.

Voorbeeld 25. Als we de expressie $ab + ac$ hebben, dan passen we gelijkheid 5.4 toe om $a(b + c)$ te krijgen.

We gebruiken de gelijkheden dus als transformatieregels voor onze syntaxboom. Hierbij maken we impliciet gebruik van de substitutieregel.

5.3.1 Gelijkheid controleren

Als we nu echter $ab + ab$ tegenkomen, willen we hierop gelijkheid 5.6 toepassen, maar ab is geen enkel symbool, maar een concatenatie. We zouden dus voor elke subexpressie van een unie willen nagaan of ze gelijk zijn.

Definitie 25. *Twee expressies zijn identiek, weergegeven als $E \equiv F$, als ze bestaan uit dezelfde symbolen in dezelfde volgorde ([Salomaa, 1966]).*

Het is duidelijk dat het eenvoudig is om na te gaan of twee reguliere expressies identiek zijn. We controleren gewoon of ze dezelfde symbolen in dezelfde volgorde hebben. Dit kan in lineaire tijd gebeuren.

We zouden echter ook graag hebben dat we $(a + b)(b + a)^* + \epsilon$ konden omzetten in $(a + b)^*$, gebruik makend van gelijkheid 5.2. We willen dus nagaan of twee expressies gelijk zijn na gelijk welke toepassingen van de gelijkheden 5.1-5.3. Aangezien er $n!$ permutaties zijn in een unie met n elementen, is deze vorm van gelijkheid niet zo eenvoudig te controleren.

We hebben dit opgelost door elke subexpressie bestaande uit geneste unies of concatenaties eerst voor te stellen door een rechts-diepe boom. We vormen bijvoorbeeld de expressie $((a \cdot b) \cdot (c \cdot d))$ om naar $(a \cdot (b \cdot (c \cdot d)))$. Op deze manier is het linkerkind van een unie- of concatenatieknoop nooit een knoop met dezelfde operator.

We hebben nu de mogelijke vormen van de boom omgezet naar één soort boom. Nu moeten we nog rekening houden met de mogelijke permutaties binnen geneste unie-operatoren. Om deze om te vormen naar een unieke vorm, sorteren we de subexpressies in lexicografische volgorde.

Voorbeeld 26. In Figuur 5.1 zien we de syntaxboom voor de reguliere expressie $(d + a) + (c + b)$. Deze wordt vervolgens omgevormd naar een rechtsdiepe boom waarna de subexpressies gesorteerd worden in lexicografische volgorde.

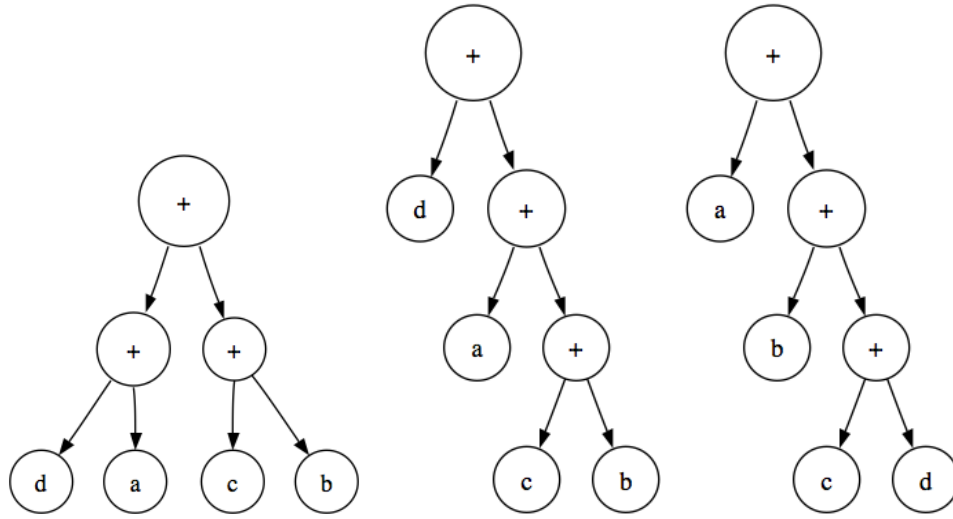
Het spreekt voor zich dat de kinderen in deze bomen ook subexpressies kunnen zijn, dus als we deze transformatie willen toepassen, moeten we eerst de kinderen omvormen, anders komt de lexicografische volgorde mogelijk anders uit.

Deze operaties veranderen niets aan de lengte van de expressie, enkel aan de vorm van de boom.

Als we deze operaties uitgevoerd hebben, kunnen we de gelijkheid van expressies gewoon nagaan door te controleren of de expressies identiek zijn.

5.3.2 Oplossen van vergelijkingen

Tot nu toe hebben we enkel de axioma's en de substitutieregel toegepast. We zouden ook graag gebruik maken van regel 2. Aangezien we in paragraaf 5.3 beslist hebben om elke subexpressie enkel kleiner te maken, zullen we nooit een expressie E hebben die we later omvormen naar $EF + G$. Wat wel mogelijk is, is een expressie van de vorm $EF + G$ waar we later E van maken (waarbij F niet de lege string definieert). Als dit gebeurt kunnen we de expressie vervangen door GF^* , ervan uitgaand dat deze korter is dan E .



Figuur 5.1: Syntaxboom voor $(d+a) + (c+b)$, vervolgens omgevormd naar rechtsdiepe boom, en daarna gesorteerd.

Hiervoor zouden we voor elke subexpressie van de juiste vorm heel veel informatie moeten opslaan en blijven bewaren terwijl de expressie omgevormd wordt. Dit samen met het feit dat de regel aan effectiviteit inboet door onze greedy toepassing van de regels, heeft ervoor gezorgd dat deze regel niet in de praktijk is toegepast.

5.3.3 Toegevoegde regels

We hebben buiten de axioma's in [Salomaa, 1966] nog een aantal andere regels toegevoegd:

$$\emptyset^* = \epsilon \quad (5.12)$$

$$\epsilon^* = \epsilon \quad (5.13)$$

$$E^* + \epsilon = E^* \quad (5.14)$$

$$E^{**} = E^* \quad (5.15)$$

$$E^* + E = E^* \quad (5.16)$$

$$F^* + E^* E = F^* + E^* \quad (5.17)$$

$$E\epsilon = E \quad (5.18)$$

$$E\emptyset = \emptyset \quad (5.19)$$

$$\epsilon + EE^* = E^* \quad (5.20)$$

We geven een bewijs voor regel 5.17 (we gebruiken regels 5.1 en 5.3 impliciet):

$$\begin{aligned}
\vdash F^* + E^* E &= F^* + E^* E \\
\vdash F^* + E^* E &= F^* F + \epsilon + E^* E && (\text{axioma 5.10}) \\
\vdash F^* + E^* E &= F^* F + \epsilon + E^* E + \epsilon && (\text{axioma 5.6}) \\
\vdash F^* + E^* E &= F^* + E^* E + \epsilon && (\text{axioma 5.10}) \\
\vdash F^* + E^* E &= F^* + E^* && (\text{axioma 5.10})
\end{aligned}$$

Regel 5.20 volgt uit het bewijs in Voorbeeld 23 en regel 5.10. De geldigheid van de rest van de gelijkheden is triviaal.

Er zijn nog veel gelijkheden die we zouden kunnen toevoegen, maar zoals reeds vermeld kunnen we niet voor alle expressies het perfecte resultaat bekomen met een eindige verzameling gelijkheden.

Een belangrijk probleem dat bleek uit de testresultaten, is dat deze regels moeilijk expressies verkleinen met verschillende geneste ster-operatoren. Om dit tegen te gaan, hebben we nog een aantal regels ingevoerd. We zetten namelijk eerst de expressie om naar ster-normaalvorm, zoals uitgelegd in paragraaf 3.3.2.

Het trim maken van de expressie is niet nodig, aangezien de regels hiervoor reeds bevat zijn in de originele elf axioma's.

5.3.4 Unie-operator

Wanneer we de expressie $E^* E + F + \epsilon$ hebben, willen we hierop gelijkheid 5.10 toepassen. Maar door het sorteren is het mogelijk dat $E^* E$ en ϵ niet bij elkaar in de buurt staan. We moeten dus telkens als we een expressie van de vorm $E^* E$ tegenkomen in een unie-operator, voor heel de unie-subboom nagaan of er een ϵ in voorkomt.

Een gelijkaardig probleem hebben we met verschillende van de andere regels. De enige mogelijkheid is dan telkens in een unie-subboom alle kinderen na te kijken. De andere regels kunnen in één knoop telkens in constante tijd geëvalueerd worden. Voor deze regels hebben we lineaire tijd in het aantal subexpressies binnen de unie nodig.

5.3.5 One-unambiguity bewaren

Als we deze regels willen gebruiken als postprocessing voor het algoritme in Hoofdstuk 3, is het natuurlijk noodzakelijk dat deze regels de one-unambiguity van de expressie bewaren.

Stelling 7. *De gebruikte regels om een expressie te verkleinen bewaren de one-unambiguity van de expressie.*

Bewijs. We gebruiken de alternatieve definitie gegeven in het bewijs van Stelling 2 om aan te geven dat een expressie one-unambiguous is.

Voor elke regel geldt dat, als de originele expressie one-unambiguous was, de nieuwe expressie dit ook is. We tonen hier het interessantste geval.

- $EG + FG \rightarrow (E + F)G$: we willen aantonen dat, als een expressie van de vorm $EG + FG$ one-nambiguous is, de equivalente expressie $(E + F)G$ dat ook is.

We stellen dat $(E + F)G$ niet one-unambiguous is, en gaan op zoek naar een contradictie. Dat $(E + F)G$ niet one-unambiguous is, wil zeggen dat het één van de twee voorwaarden van de definitie schendt:

- Stel dat de expressie niet voldoet aan de eerste voorwaarde van de definitie. Dan zijn er gemarkeerde symbolen x en y uit $first(((E + F)G)')$ waarvoor $x \neq y$ en $x^\sharp = y^\sharp$. Als x en y beide uit E' of F' komen, dan was de originele expressie niet one-unambiguous. Minstens één van de twee symbolen komt dus uit G' , neem x . Aangezien er een symbool van G' in $first(((E + F)G)')$ zit, wil dit zeggen dat E of F de lege string definieert, neem E .

We nemen nu dat $EG + FG$ gemarkeerd wordt als $E'G'_1 + F'G'_2$ en dat $(E + F)G$ gemarkeerd wordt als $(E' + F')G'_1$. In beide gevallen geldt nu $x \neq y$ en $x^\sharp = y^\sharp$, maar dit is onmogelijk aangezien $EG + FG$ one-unambiguous was.

Dus onze aanname was incorrect, er zijn geen dergelijke symbolen x en y .

- Stel dan dat de expressie niet voldoet aan de tweede voorwaarde van de definitie. Dan zijn er symbolen x, y en z zodat $x, y \in follow(((E + F)G)', z)$, met $x \neq y$ en $x^\sharp = y^\sharp$. De symbolen x, y en z kunnen zich niet binnen E' , F' of G' bevinden, anders was de originele expressie niet one-unambiguous. Het moet dus zo zijn dat $z \in last(E')$ of $z \in last(F')$, neem dat $z \in last(E')$. Dan zitten x en y in G' .

We nemen nu dat $EG + FG$ gemarkeerd wordt als $E'G'_1 + F'G'_2$ en dat $(E + F)G$ gemarkeerd wordt als $(E' + F')G'_1$. In beide gevallen geldt nu $x \neq y$ en $x^\sharp = y^\sharp$, maar dit is onmogelijk aangezien $EG + FG$ one-unambiguous was.

Dus onze aanname was incorrect, er zijn geen dergelijke symbolen x en y .

Aangezien er geen symbolen bestaan waarvoor de definitie niet geldt, wil dit zeggen dat de nieuwe expressie one-unambiguous is.

De andere gevallen zijn ofwel triviaal ofwel analoog. □

5.3.6 Volgorde van het toepassen van regels

Het is mogelijk dat het toepassen van een bepaalde regel, het onmogelijk maakt een andere regel toe te passen. Dit kan ertoe leiden dat bepaalde volgordes van het toepassen van regels een beter resultaat opleveren dan andere.

We konden backtracking gebruiken om deze volgordes na te gaan, maar de hoeveelheid volgordes is hiervoor veel te groot. Daarom hebben we telkens een paar van de beste mogelijkheden uitgekozen en nagegaan welke van die mogelijkheden ons het beste resultaat opleverde. Dit is een beperktere vorm van backtracken.

Hoofdstuk 6

Kortere expressies bekomen uit de minimale deterministische automaat

Een reguliere taal wordt uniek bepaald door één minimale, deterministische automaat. Het is dan ook niet onlogisch om vanuit deze automaat van de expressie een zo klein mogelijke equivalente expressie proberen te vormen. Er zijn twee algemeen bekende manieren om een automaat om te zetten in een reguliere expressie die de taal van de automaat beschrijft, namelijk een stelsel van lineaire vergelijkingen oplossen ([Eilenberg, 1974]) en state elimination.

In [Neumann, 2005] vinden we nog een derde methode, namelijk de transitieve sluitingsmethode. Deze zullen we hier niet bespreken omdat deze methode zeer lange reguliere expressies genereert.

Het genereren van een reguliere expressie voor een DFA is worst-case exponentieel.

Voorbeeld 27. We nemen de taal $\{w \mid w = w_{i,j}w_{j,k} \dots w_{l,m}\}$. Elk symbool heeft twee indices en een symbool mag enkel gevolgd worden door een ander symbool als de tweede index en de eerste index van deze symbolen overeenkomen.

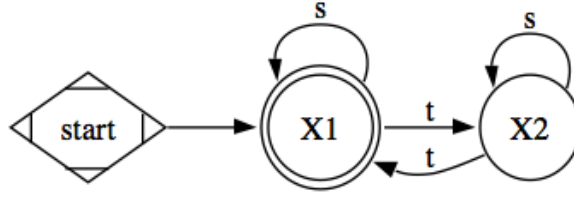
Een DFA voor deze taal moet voor elk symbool maar een state bijhouden en transities naar de states voor de symbolen waar het naar toe kan toevoegen. Dit kan in polynomiale grootte ten opzichte van het aantal indices.

Een reguliere expressie moet echter alle mogelijke volgordes waarin deze symbolen kunnen voorkomen aangeven. Dit levert ons een exponentieel grote reguliere expressie op.

We zouden ook een reguliere expressie kunnen maken uit de automaat die heel kort is, maar die niet noodzakelijk precies de originele taal aangeeft. In [Bex et al., 2006] wordt een algoritme besproken dat een automaat omzet naar een expressie die de taal van de automaat bevat.

6.1 Lineaire vergelijkingen

We kunnen een automaat omzetten in een stelsel van lineaire vergelijkingen en dit dan oplossen om een reguliere expressie te bekomen. Deze methode is equivalent met de meer intuïtieve methode van state elimination, zoals zal blijken in paragraaf 6.2.1. We gaan hier dus niet al te diep op in.



Figuur 6.1: Deterministische automaat.

6.1.1 Omzetting naar lineaire vergelijkingen

We zetten een automaat om in zijn stelsel van lineaire vergelijkingen als volgt:

- we maken een vergelijking voor elke toestand, gebruik makend van de overgangsfunctie δ , deze vergelijking is een unie van de volgende elementen:
 - voor elke toestand waar we naartoe kunnen, het symbool nodig om ernaartoe te gaan gevolgd door de toestand,
 - een ϵ als de toestand een eindtoestand is.

Voorbeeld 28. Voor de automaat weergegeven in Figuur 6.1 hebben we de volgende overgangsfunctie:

	X_1	X_2
X_1	s	t
X_2	t	s

En, aangezien X_1 een eindtoestand is, levert dit ons het volgende stelsel vergelijkingen op:

$$\begin{aligned} X_1 &= sX_1 + tX_2 + \epsilon \\ X_2 &= tX_1 + sX_2 \end{aligned}$$

Iedere vergelijking drukt uit wat onze mogelijkheden zijn in elke toestand om een accepterende run (zie paragraaf 3.2) te maken voor een woord. In het voorbeeld kunnen we, wanneer we tijdens onze run in toestand X_1 aankomen, alle woorden aanvaarden die:

- beginnen met een s , en die dan vanuit X_1 aanvaard worden of,
- beginnen met een t , en die dan vanuit X_2 aanvaard worden of,
- de lege string zijn.

De oplossing van de eerste vergelijking geeft ons dan alle woorden waarvoor we accepterende runs kunnen maken.

6.1.2 Oplossen van lineaire vergelijkingen

We willen nu de oplossing verkrijgen van de vergelijking van de starttoestand, deze zal een expressie geven die de taal van de automaat uitdrukt.

Stelling 8. *Voor een lineaire vergelijking in de vorm $X = EX + T$ is $X = E^*T$ een oplossing.*

Voor een bewijs van deze stelling verwijzen we naar [Eilenberg, 1974].

Buiten deze formule gebruiken we ook nog de commutativiteits-, transitiviteits- en distributiviteitsregels en ook dat een concatenatie met epsilon mag worden weggelaten (zie paragraaf 5.2).

Als we de tweede vergelijking in het voorbeeld in paragraaf 6.1.1 oplossen, krijgen we:

$$X_2 = s^*tX_1$$

Als we dit invullen in de eerste vergelijking en deze daarna oplossen, krijgen we:

$$\begin{aligned} X_1 &= sX_1 + t(s^*tX_1) + \epsilon \\ \implies X_1 &= (s + ts^*t)X_1 + \epsilon \\ \implies X_1 &= (s + ts^*t)^* \epsilon \\ \implies X_1 &= (s + ts^*t)^* \end{aligned}$$

En de laatste regel, $(s + ts^*t)^*$, is een reguliere expressie die de taal van de automaat beschrijft.

6.1.3 Volgorde van het oplossen van lineaire vergelijkingen

In paragraaf 6.1.2 zijn we tot een reguliere expressie gekomen voor de taal van de automaat weergegeven in Figuur 6.1. Dit is echter niet de enige mogelijke expressie die we hadden kunnen bekomen. Als we in het stelsel hadden gekozen om eerst vergelijking X_1 uit te werken, zouden we het volgende bekomen hebben:

$$\begin{aligned} X_1 &= s^*(tX_2 + \epsilon) \\ \implies X_2 &= t(s^*(tX_2 + \epsilon)) + sX_2 \\ \implies X_2 &= ts^*tX_2 + ts^*\epsilon + sX_2 \\ \implies X_2 &= (s + ts^*t)X_2 + ts^* \\ \implies X_2 &= (s + ts^*t)^*ts^* \\ \implies X_1 &= s^*(t((s + ts^*t)^*ts^*) + \epsilon) \\ \implies X_1 &= s^*t(s + ts^*t)^*ts^* + s^* \end{aligned}$$

Deze expressie beschrijft ook de taal van de automaat, maar is veel langer dan degene bekomen in paragraaf 6.1.2.

In het volgende deel zullen we merken dat state elimination een equivalente methode is (zie paragraaf 6.2.1), maar dat deze meer intuïtief is.

6.2 State elimination

Een andere manier om een expressie te vinden die de taal van de automaat beschrijft, is state elimination. Het idee hierachter is eenvoudig: We gaan van een automaat één voor één de toestanden verwijderen waarbij we toch de overgangsinformatie behouden. Dit doen we tot we enkel de start- en eindtoestand overhouden. Om dit te doen maken we gebruik van automaten die buiten symbolen ook reguliere expressies op hun transities kunnen hebben.

Definitie 26. *Een expressie-automaat is een eindige automaat die reguliere expressies toelaat als labels voor transities ([Han and Wood, 2007]). Als een dergelijke automaat enkel symbolen toelaat is het een gewone eindige automaat.*

Om state elimination te definiëren willen we dat de automaat een begintoestand heeft waarin geen transities aankomen en dat de automaat een enkele eindtoestand heeft waaruit geen transities vertrekken.

Om dit te bekomen gaan we eerst na:

- of er transities zijn die aankomen in de starttoestand. Als dit zo is, voegen we een nieuwe starttoestand toe en verbinden deze met de oude met behulp van een ϵ -transitie.
- of er meerdere eindtoestanden zijn. Als dit zo is, voegen we een nieuwe eindtoestand toe en verbinden alle oude eindtoestanden met deze nieuwe met behulp van ϵ -transities.
- of er transities vertrekken vanuit de enkele eindtoestand. Als dit zo is, voegen we een nieuwe eindtoestand toe en verbinden de oude eindtoestand hiermee met behulp van een ϵ -transitie.

We geven een transitie van toestand p naar toestand q voor symbool a verder aan met (p, a, q) . Een transitie van toestand p naar toestand q voor een expressie E geven we gelijkaardig aan met (p, E, q) .

Als de automaat aangepast is verwijderen we één voor één de toestanden van de automaat. Alle toestanden behalve de start- en eindtoestand worden verwijderd. Hierbij behouden we de beschreven taal door de labels van de transities aan te passen.

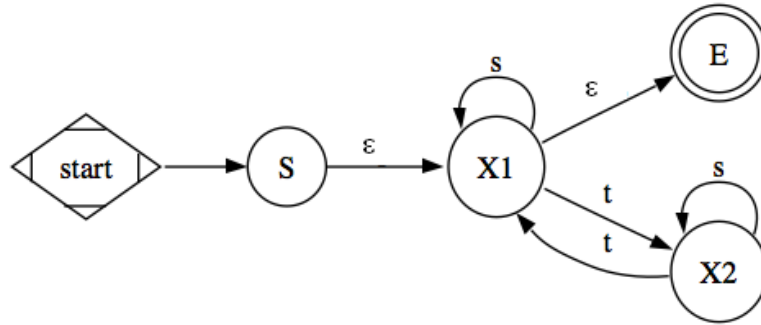
Als we een toestand q willen verwijderen, bekijken we drie soorten transities:

- alle transities (p_1, E, q) die bij die toestand binnenkomen,
- alle transities (q, F, q) die een terugkerende transitie naar de toestand zelf zijn,
- alle transities (q, G, p_2) die vanuit de toestand vertrekken.

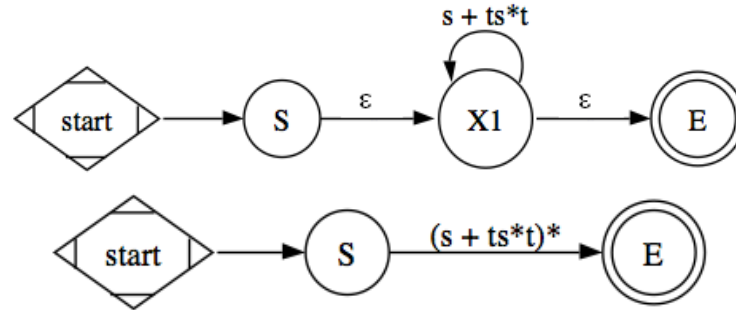
Voor elke p_1, p_2 en elke F die we zo tegenkomen, voegen we nu de transitie $(p_1, E \cdot F^* \cdot C, p_2)$ toe.

Telkens als we tussen twee toestanden p en q twee transities (p, E, q) en (p, F, q) vinden, vervangen we deze door één enkele transitie $(p, E + F, q)$.

Uiteindelijk blijven alleen de start- en eindtoestand over. De reguliere expressie voor de automaat is dan te vinden op de transitie tussen deze twee toestanden.



Figuur 6.2: Deterministische automaat voorbereid voor state elimination.



Figuur 6.3: De automaten waarin eerst toestand X_2 verwijderd is, en vervolgens toestand X_1 .

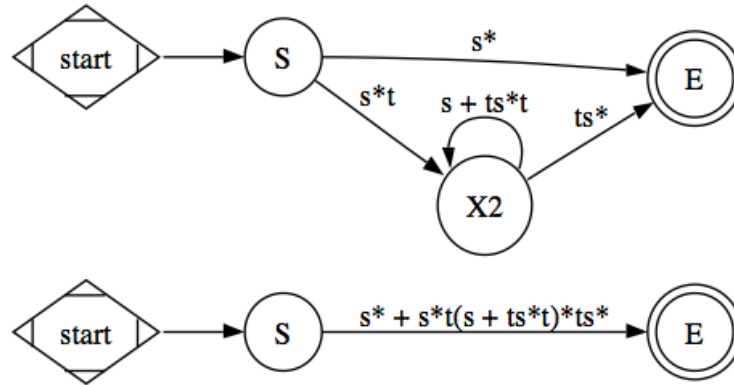
De operatie die een toestand verwijdert, is zeer intuïtief. Om van een toestand p naar een toestand q te geraken, moeten we minstens twee transities volgen, een E -transitie en een G -transitie. Hiertussen kunnen we zo veel F -transities volgen als gewenst, want deze brengen ons terug naar dezelfde toestand. Als we de toestand nu verwijderen, kunnen we dus van toestand p naar toestand q via een E , zoveel keer F als gewenst (dit geven we weer met een ster-operator), en vervolgens een G . Dit levert ons dan het nieuwe label $E \cdot F^* \cdot G$ op voor de transitie tussen de toestanden p en q .

Voorbeeld 29. Als we de automaat in Figuur 6.1 voorzien van een nieuwe start- en eindtoestand, bekomen we de automaat in Figuur 6.2.

Als we uit deze automaat achtereenvolgens de toestanden X_2 en X_1 verwijderen, bekomen we de automaten in Figuur 6.3. De expressie die zich op de resterende transitie bevindt, is een expressie die de taal van de originele automaat beschrijft.

6.2.1 Volgorde van het verwijderen van toestanden

Ook hier, net als in paragraaf 6.1.3, merken we op dat dit slechts één mogelijke uitkomst is; als we namelijk eerst toestand X_1 verwijderen en dan pas X_2 , bekomen we de automaten en dus ook de expressie uit Figuur 6.4.



Figuur 6.4: De automaten waarin eerst toestand X_1 verwijderd is, en vervolgens toestand X_2 .

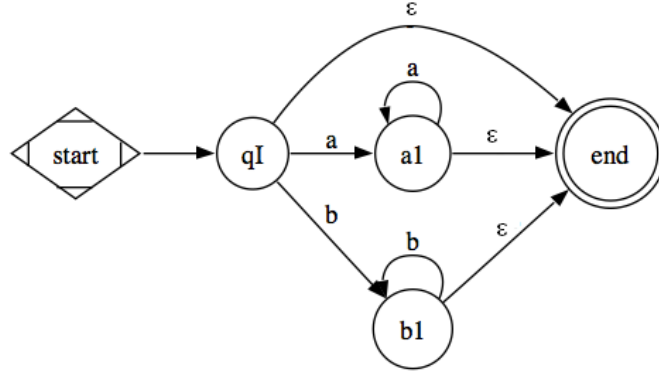
Deze voorbeelden duiden de equivalentie aan tussen state elimination en het oplossen van lineaire vergelijkingen. Als we eerst toestand X_2 verwijderen, bekomen we de eerste automaat in Figuur 6.3. We merken op dat, als we deze automaat zouden omzetten in lineaire vergelijkingen, we dan dezelfde vergelijking krijgen als in het voorbeeld uit paragraaf 6.1.2, waarbij we eerst de vergelijking van X_2 oplossen. Als we vervolgens toestand X_1 verwijderen, merken we dat we dezelfde oplossing krijgen als in dat voorbeeld.

We krijgen ook dezelfde overeenkomst bij de tussenstappen en de uitkomst, als we eerst toestand X_1 verwijderen en de vergelijking voor X_1 uitwerken.

De overeenkomst tussen deze twee methodes is eenvoudig te begrijpen als we kijken naar het voorbeeld in 6.1.2. Het symbool waarmee we van toestand X_1 naar toestand X_2 kunnen gaan, bevindt zich in de vergelijking $X_1 = sX_1 + tX_2 + \epsilon$. Vervolgens merken we dat de oplossing van X_2 ons $X_2 = s^*tX_1$ geeft, wat het symbool op de transitie is waarmee het naar zichzelf gaat met een ster, gevolgd door het symbool op de transitie waarmee het naar andere toestanden kan. Dit is hetzelfde als we in state elimination doen.

Nu is het mogelijk alle mogelijke volgordes waarin we toestanden kunnen verwijderen uit te proberen en dan de kortste resulterende expressie te nemen. Op deze manier kunnen we de best mogelijke expressie bekomen die we kunnen verkrijgen door state elimination. Het mogelijk aantal volgordes is echter zeer groot: als er n toestanden verwijderd moeten worden, zijn er $n!$ volgordes waarin we ze kunnen verwijderen.

In [Morais et al., 2005] wordt er een klasse van automaten besproken waarvoor de volgorde van de te elimineren toestanden eenvoudig te beslissen is. De klasse van automaten in kwestie is echter van weinig nut bij het omvormen van willekeurige expressies, aangezien ze onder andere acyclisch moeten zijn, wat de ster-operator al uitsluit.



Figuur 6.5: De minimale, deterministische automaat voor a^*b^* met uitbreidingen voor state elimination.

De optimale volgorde van state elimination geeft voor een deterministische automaat spijtig genoeg niet altijd een zo klein mogelijke expressie.

Voorbeeld 30. De minimale deterministische automaat voor a^*b^* is weergegeven in Figuur 6.5. Aan de vorm van de automaat is duidelijk te zien dat we nooit de minimale expressie a^*b^* zullen bekomen, want als we bijvoorbeeld de toestand a_1 willen verwijderen, zullen we altijd aa^* in het resultaat bekomen.

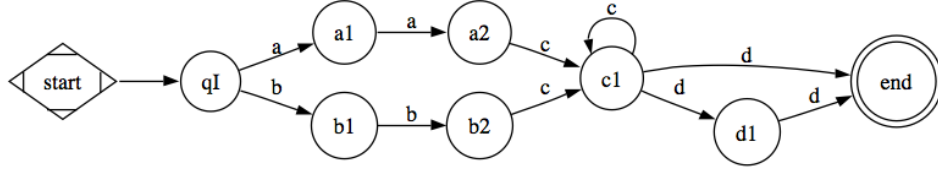
6.3 Chopping

We willen bij state elimination niet voor een automaat alle mogelijke volgordes van het verwijderen van toestanden moeten nagaan. In [Han and Wood, 2007] wordt er opgemerkt dat het niet noodzakelijk is om alle volgordes van toestanden na te gaan, maar dat er een aantal equivalent zijn. De auteurs gebruiken twee technieken die zij 'vertical chopping' en 'horizontal chopping' noemen om een automaat op te splitsen in meerdere automaten. De volgordes binnen deze automaten zijn onafhankelijk en zo zijn er veel equivalente volgordes van de originele automaat die niet bekeken moeten worden. Ze bekomen de kleinst mogelijke expressie door de expressies resulterend uit die automaten te combineren. Deze twee technieken zullen we nu verder bespreken.

Het idee van deze twee opsplitsingsmethodes is gebaseerd op de volgende stelling:

Stelling 9. *Voor een trim expressie-automaat geldt: alle expressies die voorkomen op labels van transities, komen ook voor in het resultaat dat we berekenen met state elimination. In het resultaat komen ook alleen expressies voor die voorkomen op de labels.*

Bewijs. Aangezien state elimination enkel gebruik maakt van concatenaties en ster-operatoren toegepast op de labels van transities, en elke toestand een pad naar een eindtoestand heeft, is de stelling correct. $\square$



Figuur 6.6: Automaat met bridge state c_1 .

Elke resulterende expressie maakt gebruik van de expressies die zich op de labels van de transitie bevinden, en enkel die expressies. De enige manier waarop het ene resultaat langer kan zijn dan het andere, is doordat het bepaalde expressies herhaalt. Als we nu de automaten kunnen opsplitsen in kleinere automaten zonder dat deze automaten transitie delen, kunnen we een zo klein mogelijke expressie opstellen door de expressies voor deze automaten samen te voegen.

6.3.1 Vertical chopping

Door bepaalde toestanden later te verwijderen, krijgen we kortere resultaatexpressies. Een bepaald soort toestand waarvoor dit geldt, noemen we een bridge state ([Han and Wood, 2007]).

Definitie 27. Een bridge state is een toestand q_b van een automaat die aan de volgende voorwaarden voldoet:

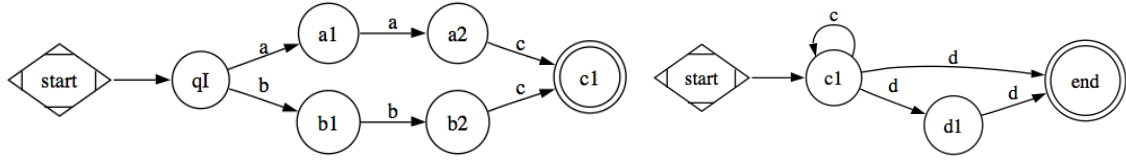
- de toestand is geen start- of eindtoestand,
- elk pad in de automaat van de start- naar een eindtoestand moet op zijn minst één keer door q_b komen,
- Als een pad voor de eerste keer langs q_b komt, mag het niet meer langs toestanden komen die het voor q_b is tegengekomen.

Voorbeeld 31. De automaat in Figuur 6.6 heeft één bridge state, namelijk de toestand c_1 .

Met behulp van een bridge state kunnen we de automaat M opdelen in twee automaten M_1 en M_2 die maar één toestand gemeen hebben, namelijk de bridge state. Als M de deterministische automaat $(Q, \Sigma, \delta, q_s, \{f\})$ is¹, en we delen deze op bij bridge state q_b , dan krijgen we de twee automaten $M_1 = (Q_1, \Sigma, \delta_1, q_s, \{q_b\})$ en $M_2 = (Q_2, \Sigma, \delta, q_b, \{f\})$. Hierbij is Q_2 de verzameling toestanden die zich op een pad van q_b naar f bevinden, en δ_2 zijn de transitie tussen deze toestanden. De verzameling $Q_1 = Q \setminus Q_2$ en δ_1 zijn de transitie tussen deze toestanden, behalve eventuele (b_q, a, b_q) -transities, met $a \in \Sigma$.

Voorbeeld 32. Als we de automaat uit Figuur 6.6 opdelen bij zijn bridge state, bekomen we de twee automaten weergegeven in Figuur 6.7.

¹Indien nodig, gebruiken we de constructie uit paragraaf 6.2 om ervoor te zorgen dat M maar één eindtoestand heeft.



Figuur 6.7: Twee resultaatautomaten na vertical chopping.

In [Han and Wood, 2007] noemen ze deze techniek vertical chopping omdat we de automaat verticaal opdelen.

Stelling 10. *As we een automaat M opdelen bij een bridge state q_b en de resulterende automaten zijn M_1 en M_2 , dan is de taal van de originele automaat $L(M) = L(M_1) \cdot L(M_2)$.*

Bewijs. Als we een woord w_1w_2 hebben uit $L(M_1) \cdot L(M_2)$, kunnen we een run van M construeren die dit woord aanvaardt, omdat alle transities van beide subautomaten ook in de originele automaat voorkomen.

Als we een woord w hebben uit $L(M)$ dan kunnen we de accepterende run voor M opdelen in accepterende runs voor M_1 en M_2 , omdat wegens de definitie van bridge state, elke run langs de bridge state b_q moet komen en als het langs de bridge state geweest is, niet meer in een vroegere state mag terugkomen. $\square$

Als we verschillende bridge states vinden, kunnen we opsplitsen op al deze bridge states, want als een toestand p een bridge state is in de originele automaat, en we splitsen op een andere bridge state q , dan is p nog altijd een bridge state in zijn automaat. Ook komen er door het opsplitsen van de automaat geen nieuwe bridge states bij. We kunnen dus alle bridge states van een automaat eenmalig berekenen en dan splitsen op al deze toestanden.

Stelling 11. *Een optimale volgorde voor het verwijderen van toestanden uit een expressie-automaat M zonder drie opeenvolgende bridge states, moet eerst alle niet-bridge states verwijderen voor het bridge states verwijdert.*

Voor het bewijs van deze stelling verwijzen we naar [Han and Wood, 2007].

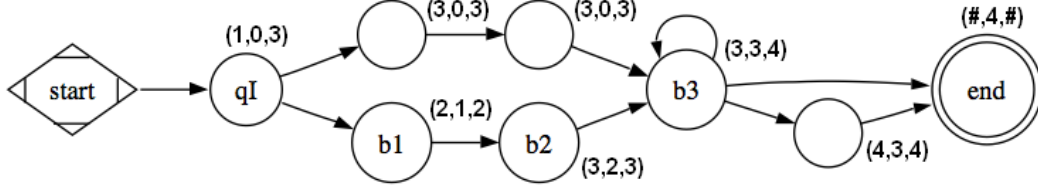
Hieruit volgt dat we een automaat kunnen opdelen bij zijn bridge states en een optimale volgorde berekenen voor state elimination van de resulterende automaten. De optimale expressie voor de originele automaat is dan de concatenatie van de state elimination van de opgesplitste automaten.

Bridge states opsporen

We willen de bridge states van een automaat opsporen. We tonen hier aan dat we dit kunnen in lineaire tijd ten opzichte van de grootte van de automaat.

We noemen een pad enkelvoudig als er geen toestand is die er twee keer in voorkomt.

Stelling 12. *Als we een enkelvoudig pad P hebben van q_s naar f in een automaat $M = (Q, \Sigma, \delta, q_s, \{f\})$, dan kunnen enkel de toestanden in P bridge states van M zijn.*



Figuur 6.8: Automaat met min-, anc- en max-waarden.

Bewijs. Als er een toestand q is, die een bridge state is en die niet in P zit, spreekt dit de tweede voorwaarde uit de definitie van bridge states tegen. $\square$

We maken nu gebruik van depth-first search ([Cormen et al., 2001]) om een enkelvoudig pad van q_s naar f te vinden. We noemen de verzameling toestanden op dit pad $C_B = \{q_s, b_1, b_2, \dots, b_k, f\}$. We gaan nu uit deze verzameling toestanden de toestanden verwijderen die een voorwaarde uit Definitie 27 schenden.

We gaan nu nogmaals met depth-first search door onze automaat. Deze keer houden we bij elke toestand q drie waardes bij:

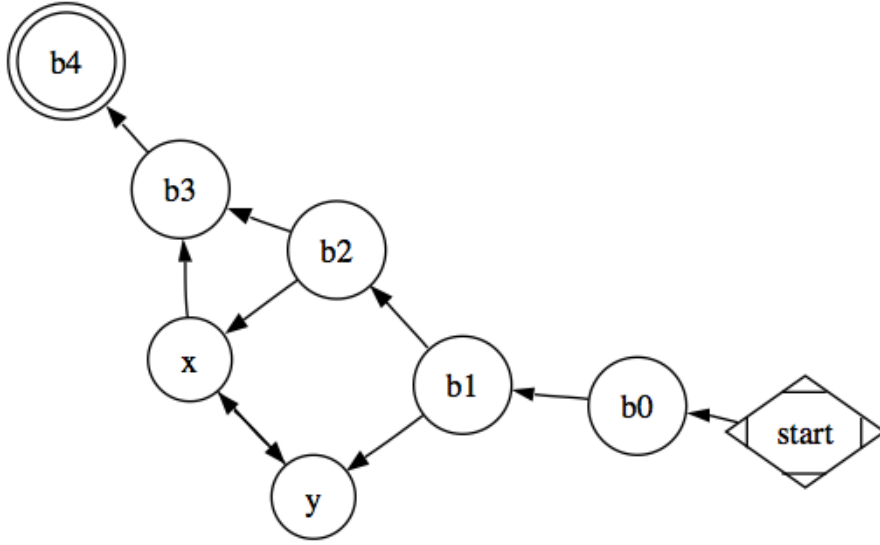
- min: de index i van een toestand $b_i \in C_B$ waarbij er een pad is van q naar b_i , maar er is geen pad naar een andere toestand in C_B met een lagere index tenzij die langs een andere toestand in C_B komt.
- anc: Voor elke toestand $b_i \in C_B$ is anc gelijk aan i . Voor de andere toestanden is anc de grootste index van een toestand $b_i \in C_B$ zodat er een pad is van b_i naar q .
- max: de index i van een toestand $b_i \in C_B$ waarbij er een pad is van q naar b_i , maar er is geen pad naar een andere toestand in C_B met een hogere index, tenzij die langs een andere toestand in C_B komt.

Voorbeeld 33. In Figuur 6.8 zien we de automaat uit Figuur 6.6 met bij elke toestand een tuple (min, anc, max) . We hebben de eindtoestand een index van 4 en de starttoestand een index van 0 toegekend. De min- en max-waardes van de eindtoestand zijn ongedefinieerd, dit is weergegeven met een #.

We bekijken nu specifiek het tuple bij de toestand b_3 . De min-waarde is de kleinste index waarvan de toestand bereikbaar is vanuit b_3 , zonder langs een andere toestand te gaan die ook tot het enkelvoudige pad behoort. Dit is hier duidelijk de toestand b_3 zelf, dus de min-waarde is gelijk aan 3. De anc-waarde van een toestand op het enkelvoudige pad is zijn index in dat pad, dus dat is hier ook 3. De max-waarde is de grootste index waarvan de toestand bereikbaar is vanuit b_3 , zonder langs een andere toestand te gaan die ook tot het enkelvoudige pad behoort. Dit is hier de eindtoestand, dus wordt de max-waarde 4.

We merken op dat de max-waarde bij de toestand b_2 gelijk is aan 3. Ook al is de toestand met index 4 bereikbaar vanuit deze toestand, daarvoor moeten ze langs de toestand met index 3.

De methode die in [Han and Wood, 2007] beschreven wordt om deze waardes te berekenen, kan in bepaalde gevallen fouten opleveren. We leggen eerst de methode uit zoals in de paper voorgesteld en geven daarna de fout en een mogelijke verbetering hiervoor aan.



Figuur 6.9: Automaat waarbij het berekenen van de min-, anc- en max-waarden kan mislopen.

Om deze waardes te berekenen gebruiken we depth-first search. We zorgen ervoor dat eerst de toestanden in het enkelvoudig pad bezocht worden. Een toestand neemt de anc-waarde van zijn voorganger over. Dit is de grootste index waarbij er een pad is naar de toestand.

We kunnen de kinderen van een toestand opdelen in twee verzamelingen: de verzameling van kindtoestanden die in C_B zitten noemen we T_1 , de overige kindtoestanden noemen we T_2 .

Als we alle kinderen van een toestand bezocht hebben, berekenen we min en max als volgt:

- min-waarde = $MIN(MIN_{q \in T_1}(q.anc), MIN_{q \in T_2}(q.min))$,
- max-waarde = $MAX(MAX_{q \in T_1}(q.anc), MAX_{q \in T_2}(q.max))$.

De min-waarde kijkt wat de kleinste index is van de toestanden die hij rechtstreeks kan bereiken en wat de kleinste index is die hij via zijn kinderen kan bekijken. Dit geeft ons inderdaad de gewenste waarde. Een gelijkaardige redenering gaat op voor de max-waarde.

Het probleem met deze methode is dat we soms een minimum- of maximumwaarde willen bekijken die nog niet gedefinieerd is. We bekijken de automaat in Figuur 6.9. Als we de min- en max-waarde van toestand b_2 willen berekenen, gaan we recursief naar toestand x . Stel dat we van daaruit doorgaan naar toestand y . Voor deze toestand zijn al zijn kinderen bezocht door de depth-first search. Als we van deze toestand de min-waarde willen berekenen, moeten we de min-waarde van toestand x nemen, maar deze is nog niet gedefinieerd. Dit is ook zo voor de max-waarde.

Later worden de min- en max-waarde van toestand x wel juist ingesteld op 3, maar dit verandert niets meer aan de waardes van toestand y . Als we nu de min- en max-waardes van toestand b_1 berekenen, is onze enige optie de anc-waarde van toestand b_2 over te nemen. Dit zorgt ervoor dat in elke toestand op het pad de min- en max-waarde op de index van de

volgende toestand in het pad worden ingesteld. Dit zou ertoe leiden dat de bridge states fout ingesteld kunnen worden.

De oorzaak van dit probleem bevindt zich in het feit dat er een cyclus is tussen de toestanden x en y . Omdat de twee toestanden kinderen van elkaar zijn, is het mogelijk dat de kinderen van een toestand wel bezocht zijn, maar dat er geen min- en max-waardes gedefinieerd zijn. We zouden dit probleem kunnen verhelpen door voor alle toestanden die niet op het pad liggen, de strongly connected components te bepalen (zie paragraaf 3.4.2). Het is duidelijk dat voor elke toestand van deze componenten de min- en max-waardes gelijk zijn, en dat er geen cyclussen tussen verschillende componenten kunnen bestaan. Als we dus elke component voorstellen met een enkele toestand, is het algoritme zoals gegeven wel correct.

Er is ook een andere mogelijkheid. We kunnen om de min-waardes te berekenen een aangepaste vorm van depth-first search gebruiken. Net zoals bij depth-first search bezoeken we een toestand enkel als die nog niet bezocht is. We vertrekken bij de eerste toestand van het enkelvoudige pad. Van daaruit bezoeken we elke transitie die naar die toestand gaat (dit is het omgekeerde van depth-first search), deze heeft duidelijk als min-waarde de index van de eerste toestand. Vanuit deze toestand gaan we zo verder en kennen elke toestand met een transitie naar die toestand dezelfde min-waarde toe. Als we een toestand uit het pad tegenkomen, krijgt deze de min-waarde, maar gaan we niet verder.

Als zo al de bereikbare toestanden bezocht zijn, nemen we de volgende toestand in het pad en herhalen we deze procedure. Aangezien we hier vertrekken vanuit de te bereiken toestand is er geen probleem met cyclussen in de automaat. Als je in een toestand aankomt, weet je altijd welke waarde er toegekend moet worden. Het is ook duidelijk dat de juiste waarde toegekend wordt, omdat we altijd eerst de kleinste mogelijke waarde toekennen aan knopen.

Dit algoritme is duidelijk lineair in de grootte van de automaat.

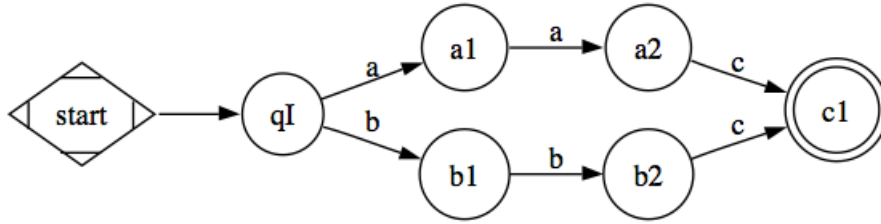
We kunnen een gelijkaardig algoritme gebruiken om de max-waarde te berekenen, door te beginnen bij de laatste toestand van het pad en telkens de vorige toestand te nemen.

De min-waarde van een toestand q impliceert dat er een pad is van q naar b_{min} . Als een toestand b_i een min-waarde heeft die kleiner is dan i , betekent dit dat b_i in een cyclus zit. Als b_i een max-waarde heeft die groter is dan $i + 1$, wil dit zeggen dat er een ander pad is van b_i naar b_{max} dat niet door $b_{i+1}, \dots, b_{max-1}$ gaat.

Nu kunnen we nagaan welke toestanden uit C_B een voorwaarde uit Definitie 27 schenden. We controleren alle $b_i \in C_B$, stel dat deze de waardes (h, i, j) hebben.

- We verwijderen q_s en f uit C_B , want deze schenden de eerste voorwaarde in Definitie 27.
- Als $j > i + 1$ wil dit zeggen dat er een pad is van b_i naar b_j dat niet langs $b_{i+1}, \dots, b_{j-1}$ gaat. We verwijderen dus de toestanden $b_{i+1}, \dots, b_{j-1}$, want deze schenden de tweede voorwaarde in Definitie 27.
- Als $h < i$, dan wil dit zeggen dat er een pad is voor b_i dat toestanden bezoekt die voor b_i kwamen in een enkelvoudig pad. Dit schendt de derde voorwaarde in Definitie 27. Dit is zo voor $b_{h+1}, \dots, b_i$. Deze toestanden zijn dus geen bridge states en worden verwijderd uit C_B .

De overgebleven toestanden zijn dan de bridge states van de automaat.



Figuur 6.10: Automaat met twee aparte paden.

Voorbeeld 34. We bekijken de waarden bij de toestanden van de automaat in Figuur 6.8. De toestand q_I heeft een max-waarde van 3, dit is groter dan $0 + 1$, dus de toestanden b_1 en b_2 zijn geen bridge states. De start- en eindtoestand zelf zijn ook geen bridge states. Uiteindelijk blijkt b_3 de enige bridge state in deze automaat te zijn.

Dit algoritme werkt in lineaire tijd ten opzichte van de grootte van de automaat, omdat onze eigen algoritmes lineair zijn en we enkel nog gebruik maken van depth-first search en in elke toestand maar constante tijd gebruiken.

Nu moeten we alleen nog van elke toestand weten in welke automaat hij thuishoort, maar dit is eenvoudig na te gaan met behulp van de anc-waarde. De bridge toestand zelf hoort natuurlijk thuis in beide automaten. Verder behoren de toestanden met een anc-waarde kleiner dan de index van de bridge state tot de linkerautomat. De andere toestanden behoren tot de rechteraautomat. Dit is eenvoudig in te zien met behulp van de definitie van de anc-waarde.

6.3.2 Horizontal chopping

We willen een zo klein mogelijke reguliere expressie vinden die de taal van een automaat beschrijft. Hier gaan we er van uit dat we een nieuwe automaat hebben die het resultaat is van het opsplitsen van onze originele automaat met behulp van vertical chopping (zie paragraaf 6.3.1). Daardoor hebben we nu een automaat zonder bridge states (Definitie 27) en met maar één start- en eindstate.

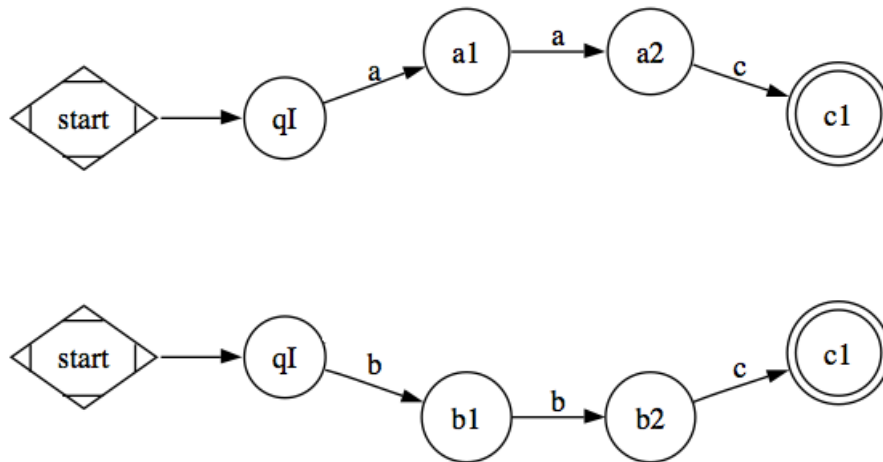
We kunnen deze automaat soms nog opsplitsen, omdat na de eerste toestand er verschillende paden door de automaat mogelijk zijn die elkaar niet meer kunnen kruisen.

Voorbeeld 35. We kunnen de automaat in Figuur 6.10 opsplitsen in de twee automaten weergegeven in Figuur 6.11.

In [Han and Wood, 2007] noemen ze deze techniek horizontal chopping omdat we de automaat horizontaal opdelen.

We kunnen nu deze automaat opdelen, en als reguliere expressie voor de originele automaat geven we de unie van de resultaatautomaten terug.

Om horizontal chopping te kunnen toepassen, moeten we subautomaten vinden die enkel de start- en eindtoestand delen. Anders bestaat het gevaar dat we labels van bepaalde transities dupliceren in verschillende automaten, en dan krijgen we niet meer gegarandeerd de kleinste mogelijke expressie.



Figuur 6.11: Twee resultaatautomaten na horizontal chopping.

We gebruiken weer depth-first search ([Cormen et al., 2001]) om deze subautomaten op te sporen. We beginnen met aan elk kind van de starttoestand een unieke groepsindex toe te kennen. Tijdens het doorlopen van de automaat, als een toestand geen groepsindex heeft, krijgt hij de groepsindex van de voorganger. Als een toestand al wel een groepsindex heeft, bekijken we deze twee indices nu als dezelfde groep.

We kunnen deze groepen dus berekenen in een tijd lineair in de grootte van de automaat. We splitsen de automaat nu op in een automaat voor elk van deze groepen.

Als er een transitie is van de starttoestand naar de eindtoestand, dan wordt hiervoor een aparte automaat gemaakt. De resulterende expressie voor deze automaat is natuurlijk altijd de expressie op het label van de transitie.

Als er een transitie in de automaat is van een toestand naar de starttoestand, is er bij het opsplitsen in subautomaten op deze manier geen garantie dat het resultaat zo klein mogelijk zal zijn. Dit wil namelijk zeggen dat er een lus is die de starttoestand bevat, en deze lus zullen we dan moeten kopiëren voor alle deelautomaten, anders is de taal van de originele automaat niet meer de unie van die van de subautomaten. Als we echter transities kopiëren over verschillende automaten, is het mogelijk dat deze meer dan nodig voorkomen in het eindresultaat; hierdoor is de lengte van het resultaat niet noodzakelijk meer optimaal (zie paragraaf 6.3).

Na horizontal chopping is het mogelijk dat er zich weer bridge states (Definitie 27) in de resultaatautomaten bevinden. Als dit zo is, kunnen we weer vertical chopping (zie paragraaf 6.3.1) toepassen. Dit herhalen we tot we de automaat niet meer kunnen opdelen.

Als we een niet meer op te delen automaat hebben, rest er ons niets meer dan alle mogelijke volgordes voor het verwijderen van toestanden uit te proberen om de kleinst mogelijke expressie te bekomen. Dit kunnen we doen door te backtracken. Backtracken is eigenlijk een depth-first search van de mogelijke volgordes.

We kunnen onze backtracking nog wel beperken. Als de som van de lengtes van alle labels van transities al groter is dan de lengte van het huidige beste resultaat, dan is het niet meer mogelijk nog een korter resultaat te bekomen. Als dit gebeurt, kunnen we dus een tak van de zoekboom vroegtijdig afsluiten. Dit betekent dat we een aantal mogelijkheden uitsluiten, waardoor het algoritme versneld wordt.

Onze backtracking kan nog meer beperkt worden door ze te laten leiden door een heuristiek. Neem een transitie (p, E, q) , het gewicht van deze transitie is het aantal karaktersvoorkomens in E . Het gewicht van een toestand is de som van de gewichten van de transities naar die toestand, de transities vertrekkende uit die toestand en de transities van de toestand naar zichzelf. In het algemeen krijgen we een korter resultaat door eerst de toestanden met een laag gewicht te elimineren. Als we gebruik maken van deze heuristiek, kunnen we het huidige beste resultaat zo snel mogelijk zo kort mogelijk krijgen, waardoor we hopelijk veel volgordes niet hoeven na te gaan.

Hoofdstuk 7

Praktische resultaten van methodes

7.1 Inleiding

Om de effectiviteit te testen van de algoritmes in deze thesis zijn er een aantal tests uitgevoerd. De resultaten van deze tests worden in dit hoofdstuk besproken. De tests zijn uitgevoerd op een MacBook 1.83 GHz Intel Core Duo met 1GB 667 MHz DDR2 SDRAM.

Voor de tests zijn een aantal willekeurig gegenereerde reguliere expressies gebruikt. Er zijn in totaal negenhonderd expressies gegenereerd; honderd expressies elk van lengtes tien, twintig, vijftig, honderd en vijfhonderd, met een alfabet van grootte vijf, namelijk $\{a, b, c, d, e\}$; nog eens honderd expressies van lengtes vijftig, honderd, tweehonderdvijftig en vijfhonderd, met een alfabet van grootte vijftientig, namelijk $\{a, b, \dots, x, y\}$. Deze expressies maken gebruik van hun alfabet, de expressies ϵ en $\emptyset$ en de drie operatoren $\cdot$, $+$ en $*$.

Het alfabet met grootte vijf is bewust vrij klein gekozen, zodat de kans op het niet one-unambiguous zijn en de mogelijkheid op het korter maken van de expressies groot genoeg zou zijn. Het andere alfabet is gekozen om het effect van de grootte van het alfabet op de algoritmes te testen.

Deze willekeurige expressies zijn gegenereerd door willekeurige syntaxbomen te genereren: eerst kiezen we een operator willekeurig uit de drie mogelijke operatoren (concatenatie, unie en ster). Hiervan maken we de wortel van de syntaxboom. Vervolgens kiezen we telkens een willekeurig niet-ingevuld kind van een operator en hiervoor kiezen we weer een willekeurige operator. Dit doen we tot de som van het aantal knopen dat voorradig is en het aantal knopen dat we nog moeten invullen, gelijk is aan de gewenste lengte. Voor elke nog in te vullen knoop kiezen we willekeurig een waarde uit het alfabet, ϵ of $\emptyset$.

lengte		10	20	50	100	500
ambiguous		32	63	92	98	98

Tabel 7.1: Het aantal niet one-unambiguous expressies in willekeurig gegenereerde expressies met een alfabet van grootte vijf.

lengte		50	100	250	500
ambiguous		88	97	100	98

Tabel 7.2: Het aantal niet one-unambiguous expressies in willekeurig gegenereerde expressies met een alfabet van grootte vijftientig.

7.2 One-unambiguous reguliere expressies

Eerst en vooral hebben we getest hoeveel expressies van elke lengte niet one-unambiguous waren (zie paragraaf 3.3). De resultaten hiervan zijn weergegeven in Tabel 7.1 voor alfabetgrootte vijf en in Tabel 7.2 voor alfabetgrootte vijftientig. Zoals te verwachten was, merken we op dat naarmate de lengte van de expressies groter wordt, er meer niet one-unambiguous expressies zijn. Er is namelijk meer kans op gelijke symbolen die Definitie 3 kunnen schenden. De grootte van het alfabet blijkt minder belang te hebben dan verwacht.

Vervolgens gaan we na voor welke expressies die niet one-unambiguous zijn, er ook daadwerkelijk geen equivalente one-unambiguous reguliere expressie is (zie paragraaf 3.4). Deze resultaten zijn weergegeven in Tabel 7.3 en Tabel 7.4. Ook dit aantal lijkt te stijgen met de lengte van de expressies, behalve bij het alfabet van lengte vijf als we aan lengte vijfhonderd komen. Bij vijfhonderd daalt het aantal dan opmerkelijk. De verklaring hiervoor is te vinden in het willekeurig genereren van de expressies. Bij lengte vijfhonderd bleken er enorm veel expressies te zijn die equivalent zijn met de expressie $(a + b + c + d + e)^*$ en deze expressie is natuurlijk one-unambiguous.

Nu willen we nagaan of het algoritme om een equivalente reguliere expressie te genereren (zie paragraaf 3.5) in de praktijk bruikbaar is. We hebben dit algoritme toegepast op de willekeurig gegenereerde expressies die niet one-unambiguous waren, maar waarvoor wel een equivalente one-unambiguous expressie bestaat. De lengte van het resultaat is weergegeven in de boxplots in Figuur 7.1 en Figuur 7.2. De gemiddelde uitvoeringstijd in milliseconden die het algoritme nodig had om een expressie van een bepaalde lengte om te zetten, is weergegeven in de grafieken in Figuur 7.3 en Figuur 7.4.

lengte		10	20	50	100	500
ambiguous		0/32	4/63	31/92	61/98	26/98

Tabel 7.3: Het aantal expressies waarvoor geen one-unambiguous tegenhanger bestaat bij een alfabet van grootte vijf.

lengte	50	100	250	500
ambiguous	25/88	67/97	96/100	97/98

Tabel 7.4: Het aantal expressies waarvoor geen one-unambiguous tegenhanger bestaat bij een alfabet van grootte vijfentwintig.

Voorbeeld 36. De expressie $c^* + a^* \cdot c + e^*$ van lengte tien, heeft de expressie $(\epsilon \cdot (\epsilon + (e \cdot (\epsilon \cdot (e \cdot \epsilon)^*) + (c \cdot (\epsilon \cdot (c \cdot \epsilon)^*) + (a \cdot (\epsilon \cdot (a \cdot \epsilon)^*) \cdot (c \cdot \epsilon))))))$ van lengte vierendertig als resultaat. Er is een willekeurig gegenereerde expressie van lengte honderd met alfabetgrootte vijfentwintig die na toepassing van het algoritme een lengte heeft van 70694.

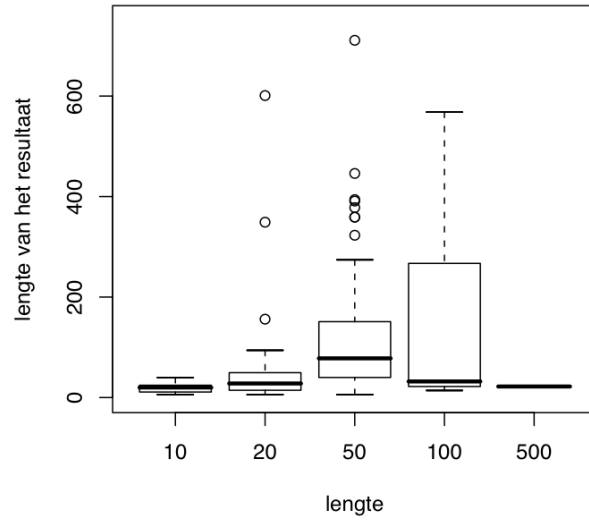
De resultaatlengtes stijgen met de lengte van de originele expressie en zijn, zoals verwacht, gemiddeld een stuk groter dan de originele expressies. We merken op dat, bij lengte vijfhonderd en alfabetgrootte vijf, het resultaat altijd lengte van 22 heeft. Dit komt voort uit het feit dat elke originele expressie van lengte vijfhonderd waarvoor er een one-unambiguous equivalent was, als resultaat de expressie $\epsilon \cdot (a \cdot \epsilon + b \cdot \epsilon + c \cdot \epsilon + d \cdot \epsilon + e \cdot \epsilon)^*$ had. Deze expressie heeft lengte 22.

De groei van de uitvoeringstijd levert geen grote verrassingen op. De tijd groeit mee met de lengte en wordt veel groter naarmate de lengte van de expressie toeneemt. Dit is geen verrassing aangezien het algoritme twee stappen uitvoert waarbij de grootte exponentieel kan toenemen (zie paragraaf 3.4.1 en 3.5). Wat misschien wel verrassend is, is hoe lang de nodige tijd nog vrij beperkt blijft. Zelfs expressies van lengte honderd kunnen nog aan zeer snel tempo verwerkt worden.

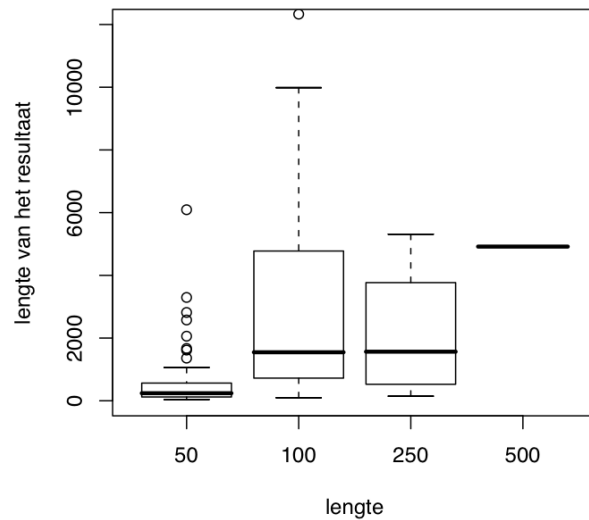
Het feit dat de minimale deterministische automaat van de expressie $(a + b + c + d + e)^*$ maar uit een enkele state bestaat, zal het algoritme om deze automaat naar een one-unambiguous reguliere expressie om te vormen zeer snel laten gaan. Aangezien elk resultaat voor een originele expressie met lengte vijfhonderd en alfabetgrootte vijf equivalent was met deze expressie, wil dit zeggen, dat de tijd die we hiervoor in de grafiek van Figuur 7.3 zien, enkel de nodige tijd is voor het construeren (zie paragraaf 3.3.2), determiniseren en minimaliseren (zie paragraaf 3.4.1) van de automaat. De tijd die het algoritme zou nodig hebben in een automaat die exponentieel groter is dan de invoer, zou dus nog exponentieel kunnen zijn ten opzichte van deze.

In Hoofdstuk 1 zeiden we dat er een methode was om veel, maar niet alle, reguliere expressies om te zetten in one-unambiguous reguliere expressies. Deze methode hebben we hier nu getest. De praktische bruikbaarheid van dit algoritme hangt natuurlijk ook af van hoeveel en welke expressies het kan omzetten. Als we kijken naar Tabel 7.4 merken we dat er toch veel expressies zijn die niet omgezet kunnen worden.

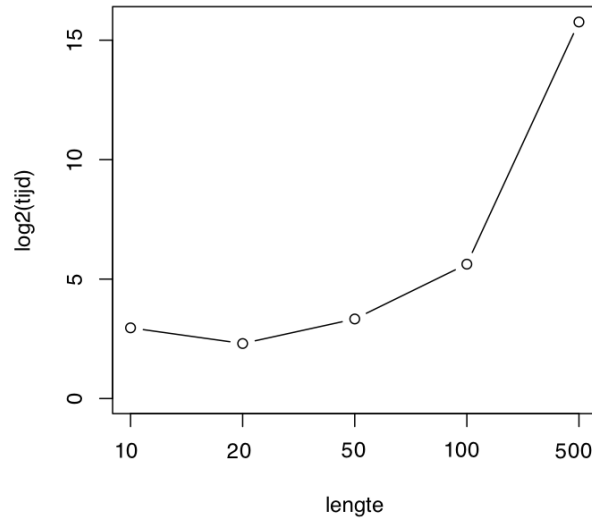
De getallen in Tabel 7.3 voor een kleinere alfabetgrootte lijken aan te geven dat er meer expressies zijn met een one-unambiguous equivalent, maar als we kijken naar de resultaatexpressies van lengte vijfhonderd, merken we dat de enige talen die we hebben kunnen omzetten naar one-unambiguous expressies diegene waren die equivalent zijn met de expressie $(a + b + c + d + e)^*$. De zesentwintig andere expressies hadden geen equivalente one-unambiguous expressies. Bij lengte honderd waren er eenenzestig expressies die geen one-unambiguous tegenhanger hebben.



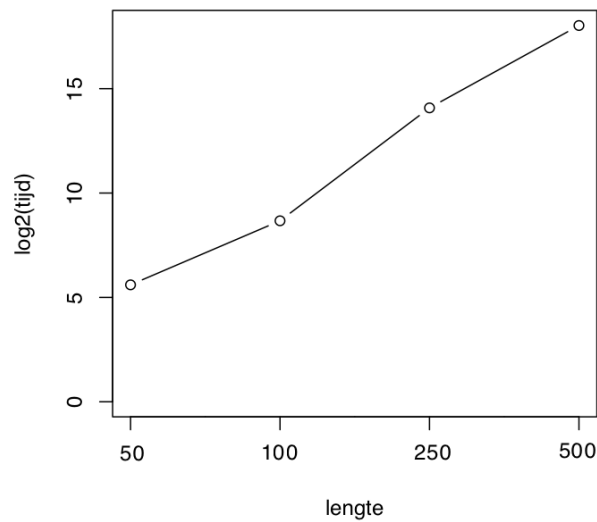
Figuur 7.1: De lengte van de nieuwe one-unambiguous expressies voor een alfabet met grootte vijf (outliers voor lengte honderd zijn niet allemaal weergegeven, deze gaan tot 6882).



Figuur 7.2: De lengte van de nieuwe one-unambiguous expressies voor een alfabet met grootte vijfentwintig (outliers voor lengte honderd zijn niet allemaal weergegeven, deze gaan tot 70694).



Figuur 7.3: De $\log_2$ van de gemiddelde uitvoeringstijd in milliseconden om één expressie om te zetten naar een equivalente one-unambiguous expressie voor een alfabet met grootte vijf.



Figuur 7.4: De $\log_2$ van de gemiddelde uitvoeringstijd in milliseconden om één expressie om te zetten naar een equivalente one-unambiguous expressie voor een alfabet met grootte vijfentwintig

Hieruit blijkt dat er toch een vrij grote hoeveelheid talen zijn die niet one-unambiguous zijn. Dit zijn natuurlijk willekeurig gegenereerde voorbeelden en daaruit kunnen we weinig opmaken voor de praktijk. Als dit echter in de praktijk problematisch blijkt, is het aan te raden om een manier te zoeken die ons een zo klein mogelijke uitbreiding van een taal oplevert waardoor deze toch one-unambiguous wordt.

7.3 Reguliere expressies inkorten

We hebben vijf methodes getest om een reguliere expressie korter te maken:

- het toepassen van de herschrijfgeregels zoals besproken in Hoofdstuk 5 (in de tabellen en figuren aangegeven met 'rewrite' of 'rewr'),
- het toepassen van dezelfde herschrijfgeregels, waarbij telkens de beste mogelijkheden gebacktrackt worden (in de tabellen en figuren aangegeven met 'backtrack' of 'back'),
- state elimination zoals besproken in paragraaf 6.2 (in de tabellen en figuren aangegeven met 'eliminate' of 'elim'),
- state elimination, met gebruik van vertical en horizontal chopping zoals besproken in paragraaf 6.3 en met backtracking op de resultaatautomaten om het best mogelijke resultaat met state elimination te verkrijgen (in de tabellen en figuren aangegeven met 'chopping' of 'chop'),
- iDTD, een algoritme dat een expressie omzet naar een expressie waarin elk symbool uit het alfabet hoogstens één keer voorkomt zoals besproken in [Bex et al., 2006] (in de tabellen en figuren aangegeven met 'iDTD').

We hebben deze algoritmes toegepast op de negenhonderd willekeurig gegenereerde expressies vermeld in paragraaf 7.1. In tegenstelling tot de algoritmes die getest werden in paragraaf 7.2, die allemaal na acceptabele tijd eindigden, bleken deze algoritmes vaak veel te lang te duren om realistisch te testen. We hebben dit opgelost door een tijdslimiet op de uitvoer te zetten. Per expressie heeft elk algoritme hooguit vijf minuten tijd gekregen om een resultaat terug te geven. Als het algoritme deze tijd overschreed werd de lengte van het resultaat niet gebruikt bij de vergelijking van de lengtes; de tijdsduur werd meegeteld voor vijf minuten in de gemiddelde uitvoeringstijd.

We bestuderen hier drie elementen van de methodes. Ten eerste de lengte na uitvoer van het algoritme, dit is weergegeven in de boxplots in Figuur 7.5 tot en met Figuur 7.13. In deze figuren is voor elke geteste lengte en voor elke alfabetgrootte een boxplot gemaakt voor iedere methode waarop we de lengte van het resultaat kunnen vergelijken. Vervolgens is de gemiddelde uitvoeringstijd nodig om één expressie te verwerken weergegeven in de grafieken in Figuur 7.14 en Figuur 7.15. Ten slotte is het aantal expressies waarvoor binnen vijf minuten een resultaat geleverd kon worden, te vinden in Tabel 7.5 en Tabel 7.6.

Voorbeeld 37. De reguliere expressie $(\epsilon^* + c^* \cdot e \cdot a)^*$ heeft een lengte van tien. Na toepassing van de algoritmes krijgen we de volgende resultaten:

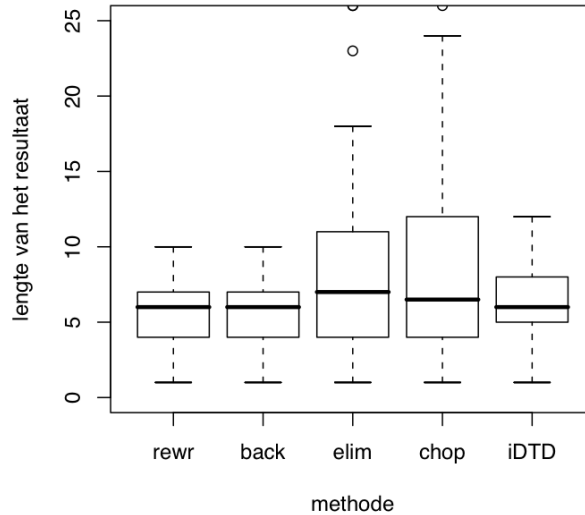
- het toepassen van de herschrijfgeregels, met of zonder backtracken geeft ons de expressie $(c^* \cdot e \cdot a)^*$ van lengte zeven,
- het gewone state elimination algoritme levert ons de expressie $(\epsilon + (((e + ((c \cdot c^*) \cdot e)) \cdot ((a \cdot e) + (((a \cdot c) \cdot c^*) \cdot e))))^* \cdot a))$ van lengte zesentwintig op,
- het state elimination algoritme met chopping geeft de expressie $((e + ((c \cdot c^*) \cdot e)) \cdot a)^*$ van lengte elf,
- iDTD geeft als resultaat $(?(+(?(+(c)))e)(a)))$. Hierbij geeft $?$ het nul of één keer en $+$ het één of meer keer voorkomen van de subexpressie aan. Als deze twee symbolen tesamen voorkomen kunnen we deze vervangen door een $*$. Deze expressie is dus eigenlijk de expressie $(c^* \cdot e \cdot a)^*$ van lengte zeven.

Het eerste wat opvalt in de boxplots die de lengte weergeven, is dat het gewone state elimination algoritme niet geschikt is voor ons doel. De expressies die het genereert zijn veel te lang. Ook de tijd die het algoritme nodig heeft is te lang, zeker voor grotere expressies.

Het iDTD-algoritme neemt een redelijke tijd in beslag, maar deze tijd stijgt niet zo snel als die van de andere algoritmes (zie Figuur 7.14 en Figuur 7.15). De expressies die het genereert zijn extreem kort in vergelijking met de andere (zie bijvoorbeeld Figuur 7.8), maar dat komt door het feit dat deze expressie niet dezelfde taal voorstelt als de originele expressie. De gegenereerde expressies beschrijven een taal die de originele taal omvat en die voorgesteld kan worden door een expressie waarin elk symbool van het alfabet hoogstens eenmaal voorkomt ([Bex et al., 2006]). De weinige expressies die het algoritme niet kon omzetten (aangegeven in Tabel 7.5 en Tabel 7.6), hebben de tijdslimiet van vijf minuten niet overschreden, maar genereerden een foutmelding van het iDTD-programma omdat het programma niet voorzien is op concatenaties met de lege verzameling. Deze zullen in de praktijk ook zelden of nooit voorkomen.

De lengtes van het resultaat van het backtracken van de herschrijfgeregels en van het gewone herschrijfalgoritme zijn in geen van de gevallen statistisch significant (nagegaan met een Student's t-test met een confidence level van 0,95). Als we de twee echter vergelijken in de grafieken in Figuur 7.14 en Figuur 7.15 merken we dat het backtrack-algoritme veel meer tijd vraagt. Deze tijds-kost gaat zo ver dat we in Tabel 7.5 en Tabel 7.6 zien dat het gewone herschrijfalgoritme nog alle expressies aankon, maar dat het backtrack-algoritme faalde bij grotere lengtes.

Het algoritme dat gebruik maakt van vertical en horizontal chopping, maakt vrij korte reguliere expressies, zeker in vergelijking met gewone state elimination (zie bijvoorbeeld Figuur 7.10). Deze verschillen waren statistisch significant bij de alfabetgrootte van vijftientig (nagegaan met een Student's t-test met een confidence level van 0,95). We merken op dat de zeer korte resultaten voor de expressies van lengte vijfhonderd en alfabetgrootte vijf een gevolg zijn van het in paragraaf 7.2 vermelde feit dat een groot deel van die expressies equivalent zijn met de expressie $(a + b + c + d + e)^*$. Omdat de chopping-methode gebruik maakt van automaat minimalisatie, kan ze deze expressies eenvoudig ontdekken.



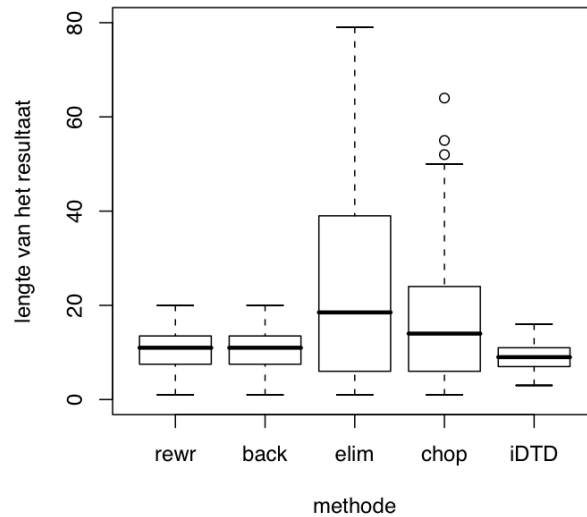
Figuur 7.5: De lengte na het toepassen van methodes om een expressie in te korten op een willekeurige expressie van lengte tien met een alfabet van grootte vijf (outliers voor state elimination zijn niet allemaal weergegeven, deze gaan tot 56).

Uiteindelijk lijkt het algoritme dat de herschrijfgeregels toepast de beste verhouding tussen tijd en lengtewinst te hebben. Het is ook het enige algoritme dat er in slaagde om voor alle expressies een resultaat te leveren binnen de vijf minuten (zie Tabel 7.5 en Tabel 7.6). Het state elimination algoritme met chopping overtreft het in lengte in Figuur 7.9, maar dat ligt aan de eerder vermelde willekeurig gegenereerde expressies.

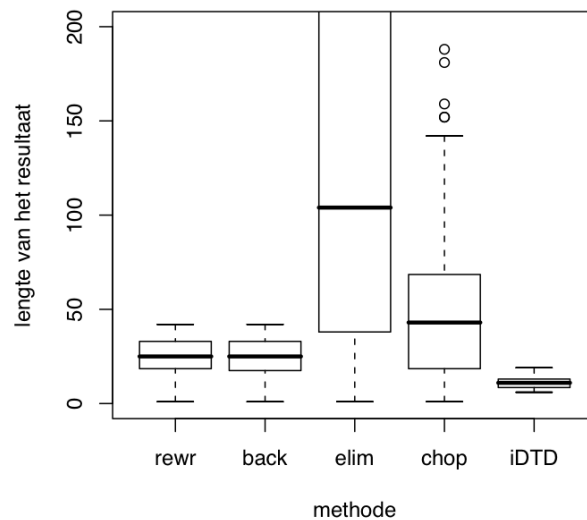
De praktijk verschilt echter op enkele punten van onze tests. We kunnen er van uitgaan dat in de praktijk de meeste expressies meer betekenis zullen hebben dan $(a+b+c+d+e)^*$, hierdoor zal het herschrijfalgoritme een betere keuze zijn. Vaak moet het zelfs nog mogelijk zijn om voor een specifieke toepassing een aantal nieuwe regels toe te voegen die de effectiviteit nog kunnen vergroten. Vertical chopping (zie paragraaf 6.3) zullen we in de praktijk meer kunnen toepassen dan bij de willekeurig gegenereerde reguliere expressies. Dit zal bijvoorbeeld bij CHARE's¹ kunnen gebeuren en deze komen in de praktijk veel voor ([Bex et al., 2006]).

Om uitsluitsel te geven welke methode het beste is, zou er een test gedaan moeten worden met expressies die in de praktijk voorkomen.

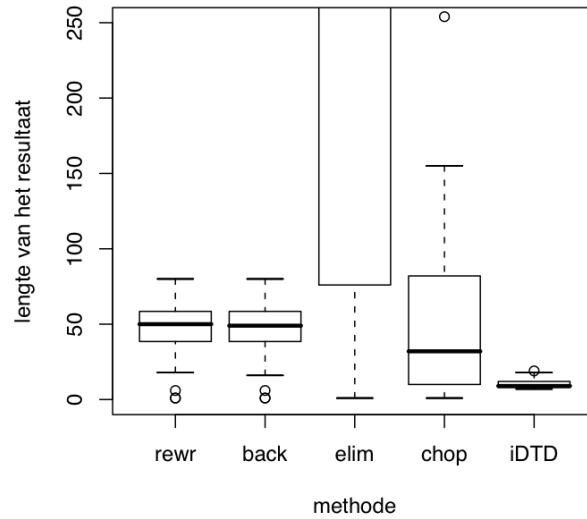
¹Een speciale reguliere expressie; elk alfabetstroom mag hoogstens één keer voorkomen en de expressie bestaat uit een aantal concatenaties van factoren met een speciale vorm.



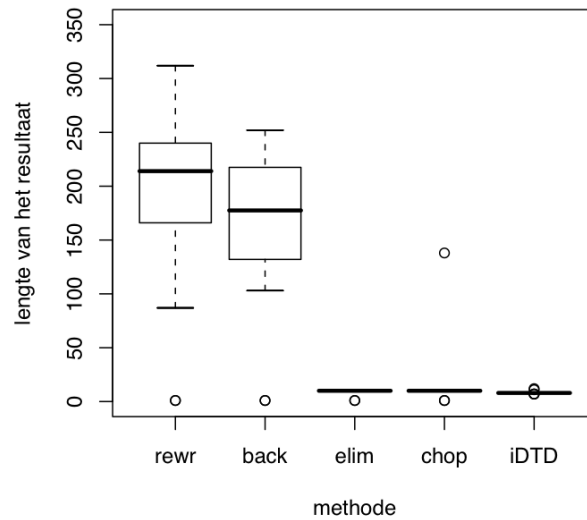
Figuur 7.6: De lengte na het toepassen van methodes om een expressie in te korten op een willekeurige expressie van lengte twintig met een alfabet van grootte vijf (outliers voor state elimination zijn niet allemaal weergegeven, deze gaan tot 1418).



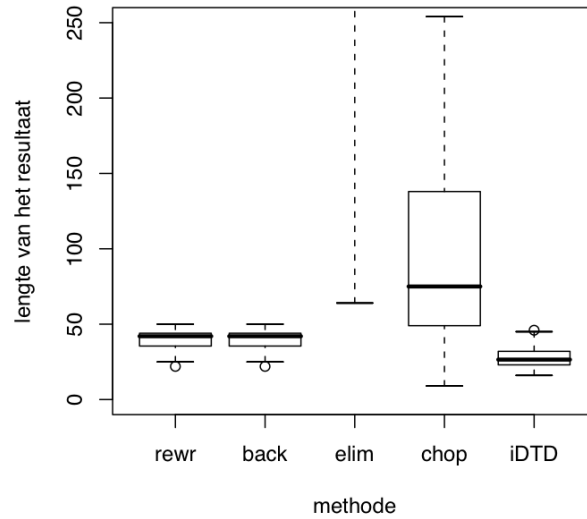
Figuur 7.7: De lengte na het toepassen van methodes om een expressie in te korten op een willekeurige expressie van lengte vijftig met een alfabet van grootte vijf (boxplot voor state elimination is niet volledig weergegeven, de outliers gaan tot 744304).



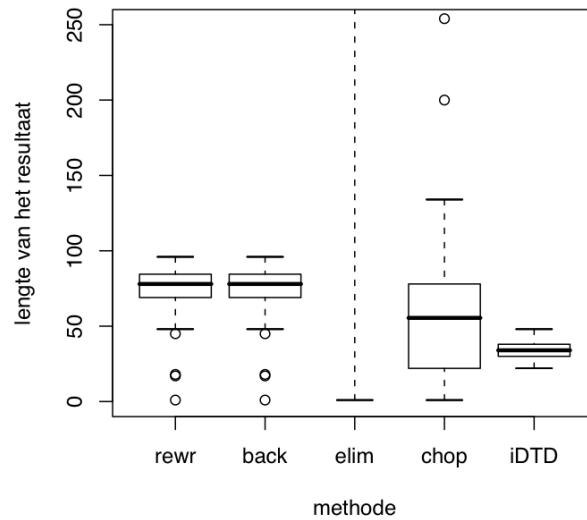
Figuur 7.8: De lengte na het toepassen van methodes om een expressie in te korten op een willekeurige expressie van lengte honderd met een alfabet van grootte vijf (boxplot voor state elimination is niet volledig weergegeven, de outliers gaan tot 1741864).



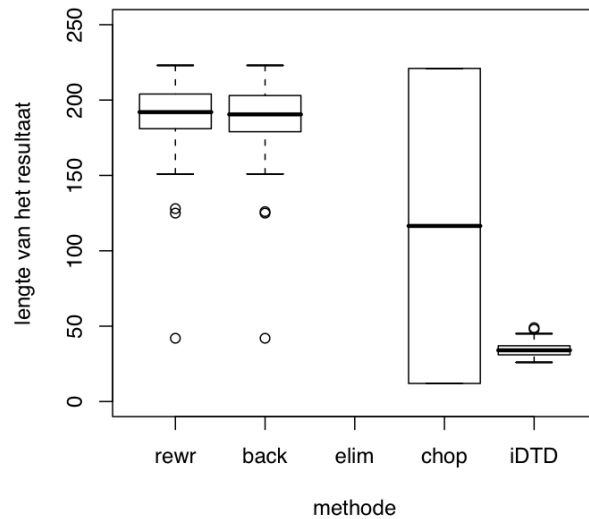
Figuur 7.9: De lengte na het toepassen van methodes om een expressie in te korten op een willekeurige expressie van lengte vijfhonderd met een alfabet van grootte vijf (outliers voor state elimination zijn niet allemaal weergegeven, deze gaan tot 977470).



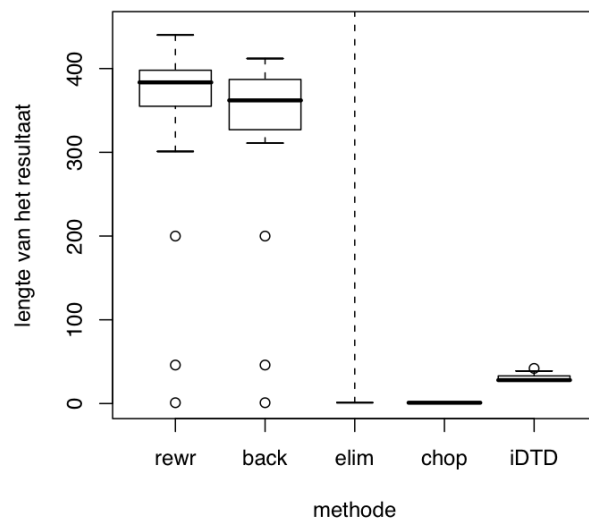
Figuur 7.10: De lengte na het toepassen van methodes om een expressie in te korten op een willekeurige expressie van lengte vijftig met een alfabet van grootte vijftientig (boxplot voor state elimination is niet volledig weergegeven, de outliers gaan tot 175752).



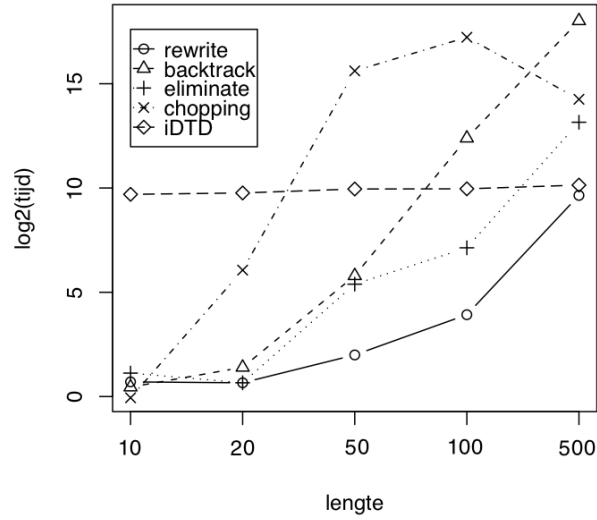
Figuur 7.11: De lengte na het toepassen van methodes om een expressie in te korten op een willekeurige expressie van lengte honderd met een alfabet van grootte vijftientig (boxplot voor state elimination is niet volledig weergegeven, de outliers gaan tot 1830812).



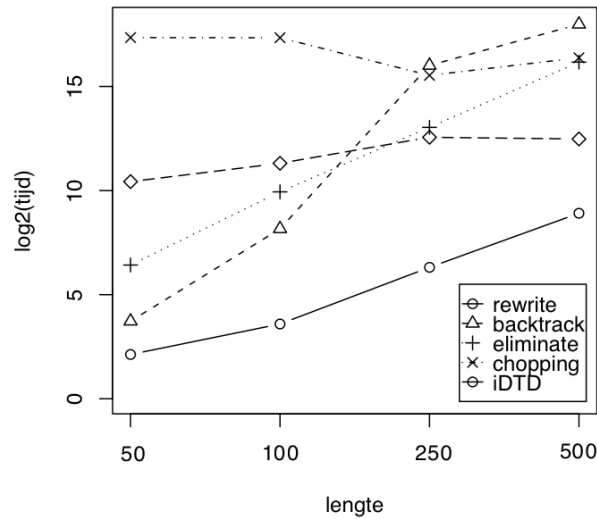
Figuur 7.12: De lengte na het toepassen van methodes om een expressie in te korten op een willekeurige expressie van lengte tweehonderdvijftig met een alfabet van grootte vijftientig (boxplot voor state elimination is niet weergegeven, waarden lopen tussen 1514 en 1808170).



Figuur 7.13: De lengte na het toepassen van methodes om een expressie in te korten op een willekeurige expressie van lengte vijfhonderd met een alfabet van grootte vijftientig (boxplot voor state elimination is niet volledig weergegeven, de outliers gaan tot 2020629).



Figuur 7.14: De $\log_2$ van de gemiddelde uitvoeringstijd in milliseconden nodig voor het toepassen van een methode om één willekeurige expressie in te korten met alfabet van grootte vijf.



Figuur 7.15: De $\log_2$ van de gemiddelde uitvoeringstijd in milliseconden nodig voor het toepassen van een methode om één willekeurige expressie in te korten met alfabet van grootte vijfentwintig.

lengte	10	20	50	100	500
rewrite	100	100	100	100	100
backtrack	100	100	100	100	16
eliminate	100	100	100	97	81
chopping	100	100	87	48	75
iDTD	91	90	95	100	100

Tabel 7.5: Het aantal expressies waarvoor een resultaat kon geleverd worden binnen de vijf minuten door een methode om één willekeurige expressie in te korten met alfabet van grootte vijf.

lengte	50	100	250	500
rewrite	100	100	100	100
backtrack	100	100	90	18
eliminate	97	84	17	4
chopping	53	18	2	1
iDTD	90	91	96	98

Tabel 7.6: Het aantal expressies waarvoor een resultaat kon geleverd worden binnen de vijf minuten door een methode om één willekeurige expressie in te korten met alfabet van grootte vijfentwintig.

7.4 Het korter maken van one-unambiguous reguliere expressies

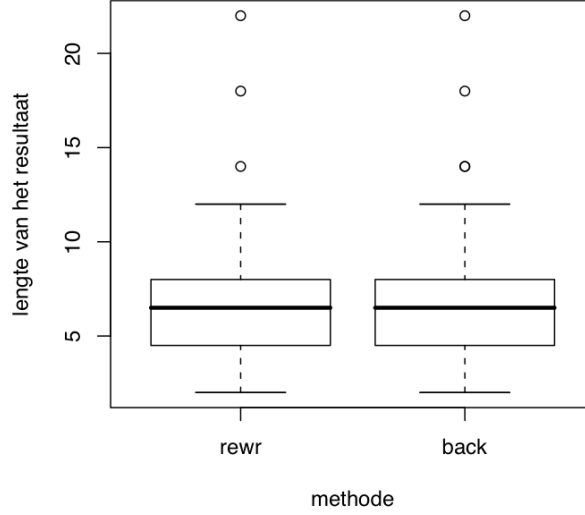
We begonnen met het korter maken van reguliere expressies omdat het resultaat van het algoritme dat expressies omzette in equivalente one-unambiguous reguliere expressies, vaak onnodig lang was. We gaan nu dus na hoe goed de methodes werken op het resultaat van dat algoritme.

We kunnen geen methodes gebruiken die gebaseerd zijn op state elimination aangezien deze de one-unambiguity van de expressie in gevaar brengen. Dit is duidelijk, aangezien deze methodes eerst terug naar de automaat gaan waaruit we in paragraaf 3.5 met speciale zorg de one-unambiguous expressie hebben gemaakt.

De methodes die gebruik maken van herschrijfgeregels, kunnen we wel gebruiken, aangezien deze de one-unambiguity bewaren (zie paragraaf 5.3.5).

Hier maken we weer gebruik van een tijdslimiet voor elke methode van vijf minuten per expressie.

De lengte van de resultaten van deze tests zijn te vinden in de boxplots in Figuur 7.16 tot en met 7.22. De $\log_2$ van de gemiddelde tijd is weergegeven in de grafieken in Figuur 7.23 en 7.24. Hoeveel van de expressies resultaat opleverden binnen de vijf minuten, is te zien in Tabel 7.7 en Tabel 7.8.



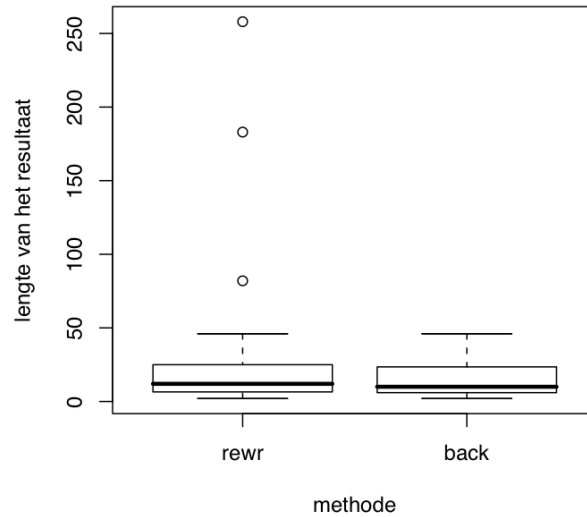
Figuur 7.16: De lengte na het toepassen van methodes om een expressie in te korten op een one-unambiguous expressie van lengte tien met een alfabet van grootte vijf.

Voorbeeld 38. De expressie $c^* + (a * \cdot c) + e^*$ is niet one-unambiguous. Het resultaat van het algoritme om deze expressie one-unambiguous te maken is de expressie $(\epsilon \cdot (\epsilon + ((e \cdot (\epsilon \cdot (e \cdot \epsilon)^*)) + ((c \cdot (\epsilon \cdot (c \cdot \epsilon)^*)) + (a \cdot ((\epsilon \cdot (a \cdot \epsilon)^*) \cdot (c \cdot \epsilon)))))))$. Deze expressie is one-unambiguous, maar is overbodig lang. Als we deze expressie verkorten, krijgen we de equivalente expressie $(c^* + (e^* + (a \cdot (a^* \cdot c))))$ en deze is duidelijk ook one-unambiguous.

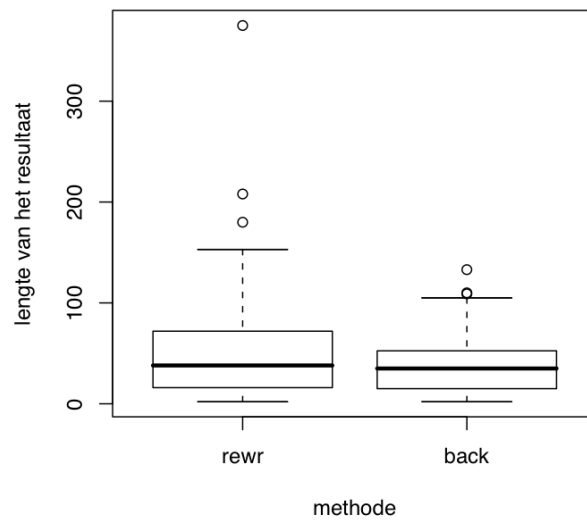
In Tabel 7.8 zien we dat er bij alfabetgrootte vijftientig voor lengtes tweehonderdvijftig en vijfhonderd nauwelijks resultaten bekomen zijn binnen de vijf minuten. Er zijn bij deze alfabetgrootte en deze lengtes dan ook geen figuren gemaakt. De drie expressies bij rewrite hadden lengte 553, 67 en 268. De expressie bij backtrack had lengte 67.

We merken hier op dat de uitvoeringstijd voor beide methodes hoog ligt (zie Figuur 7.23 en Figuur 7.24), maar dat het resultaat van het backtracken op langere expressies ook echt vrucht blijkt af te werpen. Het resultaat is statistisch significant korter (nagegaan met een Student's t-test met een confidence level van 0,95) bij alfabetgrootte vijftientig en expressies van lengte vijftig (zie Figuur 7.21) en honderd (zie Figuur 7.22).

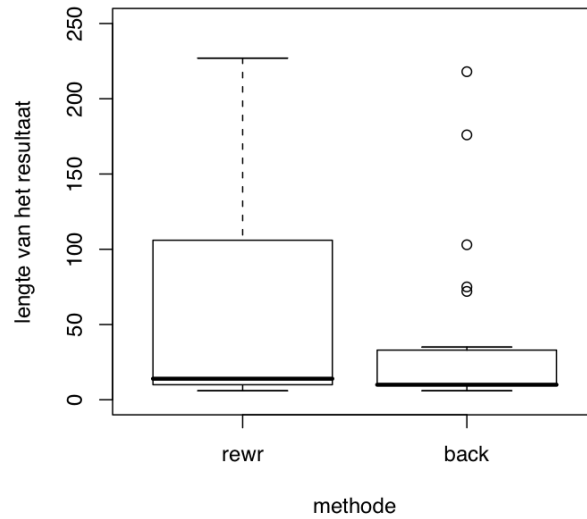
De tijd bij alfabetgrootte vijf en lengte van expressie vijfhonderd (te zien in Figuur 7.23) daalt sterk ten opzichte van de andere lengtes. Dit is weer het gevolg van het willekeurig genereren van expressies. Het resultaat van het one-unambiguous maken van de expressies is daar overal $\epsilon \cdot (a \cdot \epsilon + b \cdot \epsilon + c \cdot \epsilon + d \cdot \epsilon + e \cdot \epsilon)^*$. De tijd nodig om deze expressie om te zetten naar $(a + b + c + d + e)^*$ is zeer klein.



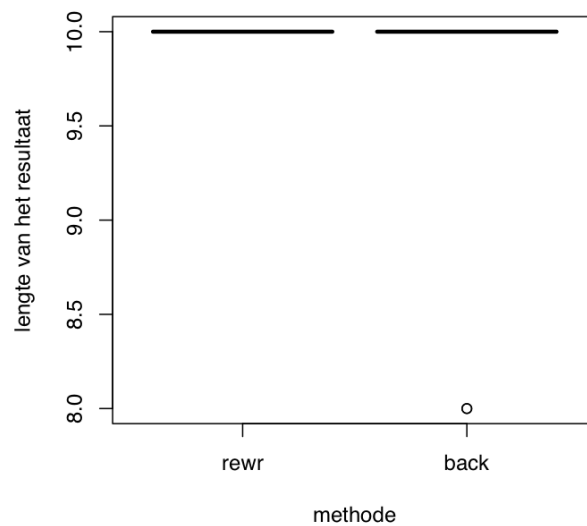
Figuur 7.17: De lengte na het toepassen van methodes om een expressie in te korten op een one-unambiguous expressie van lengte twintig met een alfabet van grootte vijf.



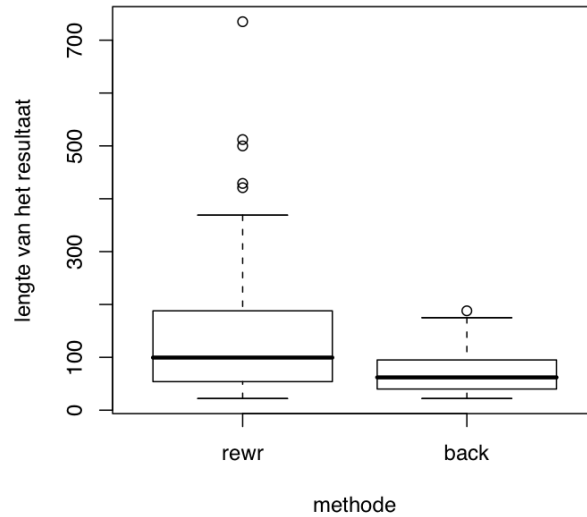
Figuur 7.18: De lengte na het toepassen van methodes om een expressie in te korten op een one-unambiguous expressie van lengte vijftig met een alfabet van grootte vijf.



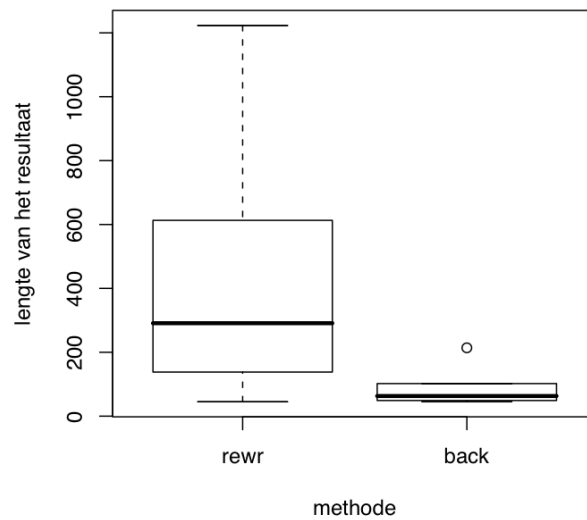
Figuur 7.19: De lengte na het toepassen van methodes om een expressie in te korten op een one-unambiguous expressie van lengte honderd met een alfabet van grootte vijf. (Er ontbreekt een outlier voor rewrite, namelijk 1078)



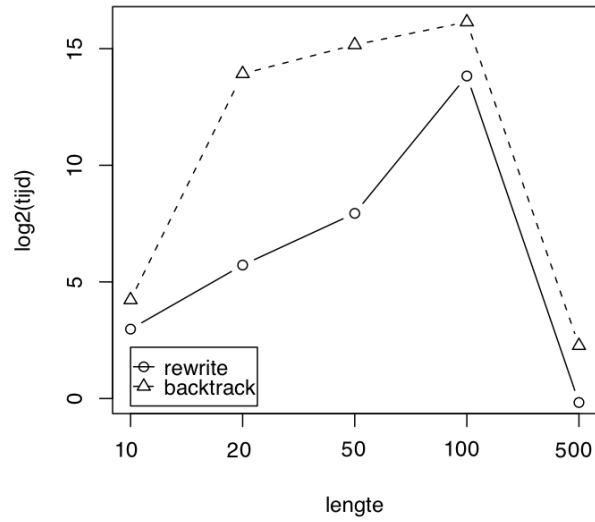
Figuur 7.20: De lengte na het toepassen van methodes om een expressie in te korten op een one-unambiguous expressie van lengte vijfhonderd met een alfabet van grootte vijf.



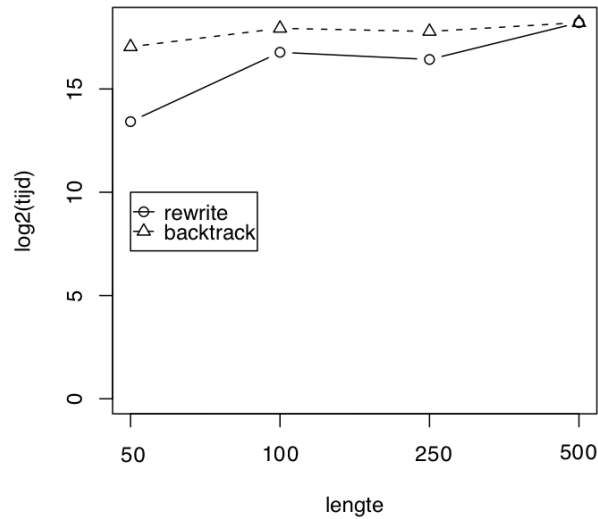
Figuur 7.21: De lengte na het toepassen van methodes om een expressie in te korten op een one-unambiguous expressie van lengte vijftig met een alfabet van grootte vijftientig.



Figuur 7.22: De lengte na het toepassen van methodes om een expressie in te korten op een one-unambiguous expressie van lengte honderd met een alfabet van grootte vijftientig.



Figuur 7.23: De $\log_2$ van de gemiddelde uitvoeringstijd in milliseconden nodig voor het toepassen van een methode om één one-unambiguous expressie in te korten met alfabet van grootte vijf.



Figuur 7.24: De $\log_2$ van de gemiddelde uitvoeringstijd in milliseconden nodig voor het toepassen van een methode om één one-unambiguous expressie in te korten met alfabet van grootte vijfentwintig.

lengte	10	20	50	100	500
rewrite	32/32	59/59	61/61	36/37	72/72
backtrack	32/32	56/59	55/61	29/37	72/72

Tabel 7.7: Het aantal expressies waarvoor een resultaat kon geleverd worden binnen de vijf minuten door een methode om een one-unambiguous expressie in te korten met een alfabet van grootte vijf.

lengte	50	100	250	500
rewrite	62/63	21/30	3/4	0/1
backtrack	38/63	6/30	1/4	0/1

Tabel 7.8: Het aantal expressies waarvoor een resultaat kon geleverd worden binnen de vijf minuten door een methode om een one-unambiguous expressie in te korten met een alfabet van grootte vijfentwintig.

7.5 Het one-unambiguous zijn van korter gemaakte reguliere expressies

In paragraaf 4.1 vermeldde we dat een kortere reguliere expressie one-unambiguous kon zijn, waar een langere expressie dat niet was. Om dit te staven hebben we in Tabel 7.9 en Tabel 7.10 het aantal one-unambiguous reguliere expressies weergegeven na het korter maken ervan.

State elimination hebben we niet onderzocht, omdat de expressies te lang waren om interessant te zijn.

Het iDTD-algoritme hebben we ook niet onderzocht, omdat de resultaten daar altijd triviaal one-unambiguous zijn.

Het aantal niet one-unambiguous expressies daalt inderdaad met het inkorten van de expressie, maar het is duidelijk dat het korter maken van de expressie zeker geen garantie is voor one-unambiguity.

lengte	10	20	50	100	500
onaangepast	32/100	63/100	92/100	98/100	98/100
rewrite	15/100	52/100	89/100	97/100	98/100
backtrack	15/100	52/100	89/100	97/100	14/16
chopping	8/100	26/100	55/87	25/48	1/75

Tabel 7.9: Het aantal niet one-unambiguous expressies na het toepassen van methodes om de expressie in te korten met een alfabet van grootte vijf.

lengte	50	100	250	500
onaangepast	88/100	97/100	100/100	98/100
rewrite	82/100	97/100	100/100	98/100
backtrack	82/100	97/100	90/90	16/18
chopping	34/53	5/18	1/2	0/1

Tabel 7.10: Het aantal niet one-unambiguous expressies na het toepassen van methodes om de expressie in te korten met een alfabet van grootte vijftwintig.

Hoofdstuk 8

Gebruik van de implementatie

De implementatie bij deze thesis is een uitwerking van de verschillende algoritmes die besproken zijn. Eerst en vooral zijn de algoritmes uit Deel I in verband met one-unambiguous reguliere expressies geïmplementeerd.

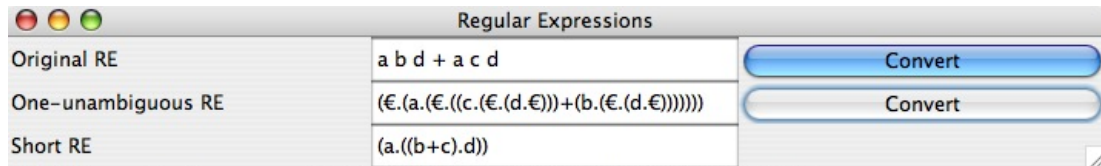
Ook zijn er vier methodes geïmplementeerd om een expressie korter te maken:

- het toepassen van de herschrijfgeregels zoals besproken in Hoofdstuk 5,
- het toepassen van dezelfde herschrijfgeregels, waarbij telkens de beste mogelijkheden gebacktrackt worden,
- state elimination zoals besproken in paragraaf 6.2,
- state elimination, met gebruik van vertical en horizontal chopping zoals besproken in paragraaf 6.3 en met backtracking op de resultaatautomaten om het best mogelijke resultaat met state elimination te verkrijgen.

We hebben een programma geschreven waarvan de GUI te zien is in Figuur 8.1. Dit programma laat toe een expressie in het eerste vak in te vullen. Als er op de knop bij dat vak gedrukt wordt, zal de expressie omgezet worden in een equivalente one-unambiguous expressie. Als de expressie reeds one-unambiguous was, zal deze niet veranderen. Als de taal niet one-unambiguous is, zal er 'null' verschijnen.

Het resultaat van het one-unambiguous maken zal in het tweede tekstvak verschijnen. We kunnen ook een expressie rechtstreeks in het tweede vak invullen. Als we op de knop bij dit vak drukken, zal het programma de herschrijfgeregels op deze expressie toepassen (zonder backtracken) om de expressie korter te maken.

Voorbeeld 39. In Figuur 8.1 zien we de resultaten voor de expressie $a \cdot b \cdot d + a \cdot c \cdot d$.



Figuur 8.1: Grafische user interface voor de implementatie.

We hebben ook een command line-versie van ons programma gemaakt dat meer mogelijkheden biedt, deze versie heet CoverProgCL. Het programma kan al de genoemde algoritmes uitvoeren. Het heeft verschillende mogelijke instellingen die op de command line kunnen worden meegegeven:

- '-e "<expressie>': geeft de reguliere expressie waarop de algoritmes toegepast moeten worden. Deze optie of de '-input' optie zijn verplicht.
- '-input <inputfile>': geeft de bestandsnaam waaruit de expressies moeten worden ingelezen waarop de algoritmes toegepast moeten worden (elke expressie moet op een nieuwe regel). Deze optie of de '-e' optie zijn verplicht.
- '-output <outputfile>': geeft de bestandsnaam naar waar het resultaat uitgeschreven moet worden. Als dit niet meegegeven wordt, wordt de output naar de command line output geschreven.
- '-u boolean': geeft aan of de expressie moet omgezet worden naar een one-unambiguous expressie. Deze optie is standaard 'true'.
- '-s <methode>': geeft de methode aan waarmee de expressie verkort moet worden. Als de expressie one-unambiguous gemaakt moet worden, gebeurt dit als laatste. Als dit niet meegegeven wordt, wordt de expressie niet verkort. De mogelijke methodes zijn 'rewrite', 'backtrack', 'eliminate' of 'chopping'. Bij backtrack moet er nog een extra parameter meegegeven worden, namelijk het aantal mogelijkheden dat op elk niveau onderzocht moet worden.
- '-i boolean': geeft aan of de infix-notatie gebruikt wordt. Deze optie is standaard 'false'. De infix-notatie is een andere manier om reguliere expressies op te schrijven. De expressie $a \cdot b + c^*$ wordt in infix-notatie $(+(.(a)(b))(*c))$.

Voorbeeld 40. Als we de expressie $a^* \cdot a \cdot b + c$ willen omzetten naar een equivalente one-unambiguous expressie, en daarna het resultaat verkorten met de herschrijfregels, dan gebruiken we het commando 'java CoverProg/CoverProgCL -u true -s rewrite -e "a* a b + c"'. Dit levert ons als resultaat ' $((a.(a^*.b))+c)$ ' op.

Hoofdstuk 9

Conclusie

In deze thesis hebben we een algoritme geïmplementeerd dat voor een reguliere expressie een equivalente reguliere expressie kan construeren die one-unambiguous is. We hebben hiervan het praktisch nut onderzocht met behulp van uitgebreide tests met willekeurig gegenereerde reguliere expressies.

Van een algoritme dat twee stappen bevat die worst-case exponentieel zijn, zouden we verwachten dat de uitvoeringstijd onacceptabel zou zijn. In onze tests bleek het algoritme toch vlot te lopen. Voor reguliere expressies van lengte honderd ligt de gemiddelde uitvoeringstijd nog ver onder een seconde per expressie.

Het algoritme zou dus in de praktijk gebruikt kunnen worden, maar een groot aantal van de talen in de willekeurig gegenereerde expressies uit onze tests bleek niet one-unambiguous te zijn. Als dit in de praktijk ook zo zou blijken te zijn, zou dit het nut van deze algoritmes zeer beperken. Dan is het misschien nuttig om op zoek te gaan naar een zo klein mogelijke taal die de originele taal bevat en toch one-unambiguous is.

Het algoritme genereert wel vaak reguliere expressies die langer zijn dan nodig om one-unambiguous te zijn. Om de lengte van het resultaat te verkorten, zijn er enkele methodes onderzocht die een reguliere expressie omzetten in een equivalente reguliere expressie die korter is.

We hebben twee algemene ideeën onderzocht: de expressie met behulp van heuristieken omzetten naar een kleinere expressie of een kleinere expressie proberen te genereren met behulp van de minimale deterministische automaat voor de taal.

Bij het verkleinen van de expressie met behulp van herschrijfgeregels hebben we voor een greedy aanpak gekozen: bij elke stap wordt de expressie kleiner. Hierdoor krijgen we niet altijd de kleinst mogelijke expressie, maar het algoritme heeft wel een duidelijk stoppunt. We hebben ook de mogelijkheid onderzocht om bij elke stap een paar van de beste mogelijkheden uit te proberen. Deze methode levert soms een kortere expressie, maar duurt een stuk langer. Een voordeel van deze methodes is dat we onze herschrijfgeregels eenvoudig zo kunnen kiezen dat ze de one-unambiguity van de expressie bewaren.

Het proberen te verkleinen van expressies door de minimale deterministische automaat om te zetten naar een zo klein mogelijke reguliere expressie, blijkt niet zo efficiënt te zijn. Gewone state elimination is in de praktijk onbruikbaar omdat het zeer grote expressies oplevert. Een

verbetering hierop die de automaat opdeelt en het resultaat construeert uit de best mogelijke resultaten van die deelautomaten blijkt beter te werken. Deze methodes zijn minder efficiënt dan de methodes met herschrijfgeregels, maar zijn wel goed in het opsporen van bepaalde structuren in de expressie. Deze methodes ontdekten in onze tests altijd de expressies die equivalent waren met Σ^* , waar de methodes met herschrijfgeregels dit niet deden.

Bibliografie

- [Aho et al., 1986] Aho, A., Sethi, R., and Ullman, J. (1986). Compiler design: Principles, tools, and techniques. [3.3.1](#)
- [Bex et al., 2006] Bex, G. J., Neven, F., Schwentick, T., and Tuyls, K. (2006). Inference of concise DTDs from XML data. In *VLDB'2006: Proceedings of the 32nd international conference on Very large data bases*, pages 115–126. VLDB Endowment. [1](#), [6](#), [7.3](#), [7.3](#)
- [Brüggemann-Klein, 1993] Brüggemann-Klein, A. (1993). Regular expressions into finite automata. *Theor. Comput. Sci.*, 120(2):197–213. [1](#), [3.1](#), [3.3.2](#), [3.3.2](#)
- [Brüggemann-Klein and Wood, 1998] Brüggemann-Klein, A. and Wood, D. (1998). One-unambiguous regular languages. *Inf. Comput.*, 140(2):229–253. [1](#), [3.1](#), [9](#), [10](#), [3.3.2](#), [3.4.2](#), [3.4.2](#), [3.5](#)
- [Cormen et al., 2001] Cormen, T. H., Leiserson, C. E., Rivest, R. L., and Stein, C. (2001). *Introduction to Algorithms, Second Edition*. The MIT Press. [6.3.1](#), [6.3.2](#)
- [Ehrenfeucht and Zeiger, 1976] Ehrenfeucht, A. and Zeiger, H. P. (1976). Complexity Measures for Regular Expressions. *J. Comput. Syst. Sci.*, 12(2):134–146. [4.3](#)
- [Eilenberg, 1974] Eilenberg, S. (1974). *Automata, Languages, and Machines*, volume A. Academic Press, New York, 1 edition. [6](#), [6.1.2](#)
- [Gelade et al., 2007] Gelade, W., Martens, W., and Neven, F. (2007). Optimizing Schema Languages for XML: Numerical Constraints and Interleaving. In *ICDT*, pages 269–283. [1](#), [4.3](#)
- [Han and Wood, 2007] Han, Y.-S. and Wood, D. (2007). Obtaining shorter regular expressions from finite-state automata. *Theor. Comput. Sci.*, 370(1-3):110–120. [3.7](#), [26](#), [6.3](#), [6.3.1](#), [6.3.1](#), [6.3.1](#), [6.3.2](#)
- [Hopcroft et al., 2000] Hopcroft, J. E., Motwani, R., and Ullman, J. D. (2000). Introduction to Automata Theory, Languages, and Computation (2nd Edition). Hardcover. [3.4.1](#), [3.4.1](#)
- [James Clark and Murata Makoto, 2007] James Clark and Murata Makoto (2007). RELAX NG Specification. <http://www.relaxng.org/spec-20011203.html>. [1](#)
- [Lee and Shallit, 2004] Lee, J. and Shallit, J. (2004). Enumerating regular expressions and their languages. In *CIAA*, pages 2–22. [4.3](#), [4.3](#)

- [Martens et al., 2006] Martens, W., Neven, F., Schwentick, T., and Bex, G. J. (2006). Expressiveness and complexity of XML Schema. *ACM Trans. Database Syst.*, 31(3):770–813. [1](#)
- [Morais et al., 2005] Morais, J. J., Moreira, N., and Reis, R. (2005). Acyclic Automata with Easy-to-Find Short Regular Expressions. In *CIAA*, pages 349–350. [6.2.1](#)
- [Neumann, 2005] Neumann, C. (2005). Converting Deterministic Finite Automata to Regular Expressions. [6](#)
- [Nuutila and Soisalon-Soininen, 1994] Nuutila, E. and Soisalon-Soininen, E. (1994). On finding the strongly connected components in a directed graph. *Inf. Process. Lett.*, 49(1):9–14. [3.4.2](#)
- [Salomaa, 1966] Salomaa, A. (1966). Two Complete Axiom Systems for the Algebra of Regular Events. *J. ACM*, 13(1):158–169. [1.1.2](#), [5.1](#), [5.2](#), [1](#), [5.2](#), [25](#), [5.3.3](#)
- [W3C, 2007a] W3C (2007a). Extensible Markup Language (XML) 1.0 (Fourth Edition). <http://www.w3.org/TR/REC-xml/>. [1](#)
- [W3C, 2007b] W3C (2007b). W3C XML Schema. <http://www.w3.org/XML/Schema>. [1](#)

Deel III

Appendices

Bijlage A

Bewijzen

In deze appendix is de volledige uitwerking van bepaalde bewijzen uit deze thesis te vinden.

A.1 Stelling 1

Stelling (1 op p.23). *Voor elke markering E' , behoort een woord $x_1 \dots x_n$ over Π met $n \geq 1$ tot $L(E')$ als en slechts als:*

- $x_1 \in \text{first}(E')$,
- $x_n \in \text{last}(E')$,
- $x_{i+1} \in \text{follow}(E', x_i)$ voor $1 \leq i < n$.

Bewijs. Als het woord $x_1 \dots x_n$ tot de taal behoort, volgt uit de definities van de verzamelingen, dat de verzamelingen de gewenste symbolen bevatten.

Er rest ons dus enkel de andere richting te bewijzen. Dit bewijzen we door inductie op de structuur van de markering:

Basis:

We onderscheiden twee basisgevallen:

- de markering is ϵ of $\emptyset$,
- de markering is van de vorm x_i met x_i een symbool uit het alfabet.

Als de markering van de vorm ϵ of $\emptyset$ is, dan behoort er geen enkel woord van de vorm $x_1 \dots x_n$ met $n \geq 1$ tot de taal. De first-, last- en follow-verzamelingen zijn dan ook altijd leeg. Een dergelijk woord $x_1 \dots x_n$ komt dus enkel voor als de verzamelingen de juiste informatie bevatten, zijnde nooit.

Als de markering de vorm x_i heeft, dan is het enige woord dat gedefinieerd wordt x_i . De enige informatie die dan in de first-, last- en follow-verzamelingen komt, is $\text{first}(E') = \{x_i\}$ en $\text{last}(E') = \{x_i\}$.

Inductiehypothese: De uitspraak is geldig voor markeringen E'_1 en E'_2 .

Te bewijzen: De uitspraak is geldig voor $E'_1 \cdot E'_2$, $E'_1 + E'_2$ en E'^*_1 .

- $E'_1 \cdot E'_2$: Aangezien het gaat om een markering, weten we dat de symbolen in E'_1 en E'_2 disjunct zijn. Dus zijn ook hun first-, last- en followverzamelingen disjunct. We hebben nu de first-, last- en followverzamelingen voor E , hieruit kunnen we deze verzamelingen construeren voor E'_1 en E'_2 :
 - $first(E'_1)$ is de deelverzameling van $first(E')$ met enkel symbolen uit E'_1 ,
 - $last(E'_1)$ zijn de symbolen uit E'_1 die in $follow(E')$ gevolgd kunnen worden door een symbool uit E'_2 , samen met de deelverzameling van $last(E')$ met enkel symbolen uit E'_1 ,
 - $follow(E'_1)$ is de deelverzameling van $follow(E')$ met enkel symbolen uit E'_1 ,
 - $first(E'_2)$ zijn de symbolen uit E'_2 die in $follow(E')$ voorafgegaan kunnen worden door een symbool uit E'_1 , samen met de deelverzameling van $first(E')$ met enkel symbolen uit E'_2 ,
 - $last(E'_2)$ is de deelverzameling van $last(E')$ met enkel symbolen uit E'_2 ,
 - $follow(E'_2)$ is de deelverzameling van $follow(E')$ met enkel symbolen uit E'_2 .

Omdat E' van de vorm $E'_1 \cdot E'_2$ is, kan men nu elk woord $x_1 \dots x_n$ in $L(E')$ opsplitsen in twee woorden $x_1 \dots x_j$ en $x_{j+1} \dots x_n$ door na te gaan waar de overgang van symbolen tussen E'_1 en E'_2 plaatsvindt. Stel dat $1 \leq j < n$, dus geen van de twee woorden is de lege string, dan behoort x_1 tot $first(E'_1)$, want het behoort tot $first(E')$ en is een symbool uit E'_1 . Het symbool x_j behoort tot $last(E'_1)$ want $x_{j+1} \in follow(E', x_j)$ (gegeven) en x_j is een symbool uit E'_1 en x_{j+1} is een symbool uit E'_2 . Alle $x_{i+1} \in follow(E'_1, x_i)$ voor $1 \leq i < j$ want het zijn symbolen uit E'_1 . De redenering voor E'_2 is volledig analoog.

De twee woorden $x_1 \dots x_j$ en $x_{j+1} \dots x_n$ worden nu gedefinieerd door E'_1 en E'_2 vanwege de inductiehypothese. Dus wordt $x_1 \dots x_n$ gedefinieerd door E' .

Als $j = 0$, dan krijgen we de twee woorden ϵ en $x_1 \dots x_n$. Het symbool x_1 zit in $first(E'_2)$, want het zit in $first(E')$ en is een symbool uit E'_2 . Het symbool x_n zit in $last(E'_2)$ want het zit in $last(E')$ en is een symbool uit E'_2 . De nodige follow-informatie zit ook in $follow(E'_2)$ zodat $x_1 \dots x_n$ gedefinieerd wordt door E'_2 . Aangezien er een symbool uit E'_2 in $first(E')$ zit, wil dit zeggen dat er een woord $\epsilon \cdot w \in L(E')$ is, met $w \in L(E'_2)$. Hieruit volgt dat $\epsilon \in L(E'_1)$. Dus wegens de inductiehypothesen wordt $x_1 \dots x_n$ gedefinieerd door E'_2 , en omdat $\epsilon \in E'_1$, wordt $x_1 \dots x_n$ ook gedefinieerd door E' .

De redenering voor $j = n$ is analoog.

- $E'_1 + E'_2$: Aangezien het gaat om een markering, weten we dat de symbolen in E'_1 en E'_2 disjunct zijn. Dus zijn ook hun first-, last- en followverzamelingen disjunct. We hebben nu de first-, last- en followverzamelingen voor E' , hieruit kunnen we deze verzamelingen construeren voor E'_1 en E'_2 :

- $first(E'_1)$ is de deelverzameling van $first(E')$ met enkel symbolen uit E'_1 ,
- $last(E'_1)$ is de deelverzameling van $last(E')$ met enkel symbolen uit E'_1 ,
- $follow(E'_1)$ is de deelverzameling van $follow(E')$ met enkel symbolen uit E'_1 ,
- $first(E'_2)$ is de deelverzameling van $first(E')$ met enkel symbolen uit E'_2 ,
- $last(E'_2)$ is de deelverzameling van $last(E')$ met enkel symbolen uit E'_2 ,
- $follow(E'_2)$ is de deelverzameling van $follow(E')$ met enkel symbolen uit E'_2 .

Omdat E' van de vorm $E'_1 + E'_2$ is, volgt dat elk woord $x_1 \dots x_n$ in $L(E')$ ofwel volledig uit symbolen uit E'_1 of symbolen uit E'_2 bestaat.

Stel dat $x_1 \dots x_n$ enkel uit symbolen uit E'_1 bestaat: het symbool x_1 behoort dan tot $first(E'_1)$ want het behoort tot $first(E')$ en is een symbool uit E'_1 . Het symbool x_n behoort tot $last(E'_1)$ want het behoort tot $last(E')$ en is een symbool uit E'_1 . Alle $x_{i+1} \in follow(E'_1, x_i)$ voor $1 \leq i < n$ want het zijn symbolen uit E'_1 .

De redenering voor E'_2 is volledig analoog.

Het woord $x_1 \dots x_n$ wordt nu gedefinieerd door E'_1 of E'_2 wegens de inductiehypothese. Dus wordt $x_1 \dots x_n$ gedefinieerd door E' .

- E'_1^* : We hebben de first-, last- en followverzamelingen voor E' , hieruit kunnen we deze verzamelingen construeren voor E'_1 :

- $first(E'_1)$ is $first(E')$,
- $last(E'_1)$ is $last(E')$,
- $follow(E'_1, x)$ is een deelverzameling van $follow(E', x)$:
 - * Als $x \notin last(E')$, dan is $follow(E'_1, x) = follow(E', x)$,
 - * Als $x \in last(E')$, dan bevat $follow(E'_1, x)$ ten minste alle elementen van $follow(E', x)$ die niet in $first(E')$ zitten.

Omdat E' van de vorm E'_1^* is, kunnen we elk woord $x_1 \dots x_n$ opdelen in niet-lege delen die tot $L(E'_1)$ behoren. Deze opsplitsing doen we door de expressie op te delen op alle plaatsen waar een symbool uit $last(E')$ gevolgd wordt door een symbool uit $first(E')$.

We splitsen E' nu op in dergelijke zo klein mogelijke niet-lege delen. Noem zo een woord $x_j \dots x_k$: het symbool x_j behoort dan tot $first(E'_1)$, want het behoort tot $first(E')$. Het symbool x_k behoort tot $last(E'_1)$, want het behoort tot $last(E')$. Alle $x_{i+1} \in follow(E'_1, x_i)$ voor $j \leq i < k$ want deze behoorden tot $follow(E')$.

Er waren elementen van $follow(E')$ die niet tot $follow(E'_1)$ behoren. Dit zijn echter elementen van $first(E')$ die in $follow(E', x)$ met $x \in last(E')$ zaten. Deze kunnen hier niet voorkomen, want dan hadden we het woord hier gesplitst.

De woorden $x_j \dots x_k$ worden nu gedefinieerd door E'_1 vanwege de inductiehypothese. Dus wordt $x_1 \dots x_n$ gedefinieerd door E' .

Uit deze constructies volgt nu dat de stelling geldt voor alle markeringen. □

A.2 Stelling 5

Stelling (5 op p.36). *Stel, E is een trim reguliere expressie en $x \in \text{sym}(E')$. Dan karakteriseren we de orbits en gates in M_E als volgt:*

1.
 - Als er geen subexpressie H^* is waarvoor $x \in \text{sym}(H')$, dan is $O(x) = \{x\}$ en dan is dit een triviale orbit¹.
 - Als H^* de maximale subexpressie onder een $*$ is, dan $O(x) = \text{sym}(H')$ en is deze orbit nooit triviaal².
 - De orbit van de starttoestand is triviaal.
2. Als H^* een maximale subexpressie onder een $*$ is met $\text{sym}(H') \neq \emptyset$ dan zijn $\text{last}(H')$ de gates van de orbit van $\text{sym}(H')$. Elke transitie in de volledige automaat M_E tussen toestanden in $\text{sym}(H')$ is al aanwezig in M_{H^*} .

Bewijs. Dit bewijzen we door inductie op de structuur van de expressie:

Basis:

We onderscheiden twee basisgevallen:

- de expressie is ϵ of $\emptyset$,
- de expressie is van de vorm x_i met x_i een symbool uit het alfabet.

Als de expressie van de vorm ϵ of $\emptyset$ is, dan heeft de Gushkov automaat de starttoestand als enige toestand³ en heeft deze geen transities. Hiervoor zijn de beweringen triviaal waar.

Als de expressie van de vorm x_i is, dan heeft de Glushkov automaat een enkele transitie, namelijk van de starttoestand naar x_i en is x_i een eindtoestand. Er zijn dus twee aparte orbits en deze zijn beide triviale orbits. Ook hier zijn de uitspraken dus triviaal waar.

Inductiehypothese: De uitspraak is geldig voor expressies E_1 en E_2 .

Te bewijzen: De uitspraak is geldig voor $E_1 \cdot E_2$, $E_1 + E_2$ en E_1^* .

- $E_1 \cdot E_2$: De twee automaten voor E_1 en E_2 hebben unieke toestanden en zijn met elkaar verbonden door transities die vanuit $\text{last}(E_1')$ naar $\text{first}(E_2')$ gaan.
 1. – Als er geen dergelijke subexpressie H^* is, dan volgt uit de inductiehypothese dat één van de twee automaten voor de subexpressies een triviale orbit $\{x\}$ bevat. Deze orbit zal door het samenvoegen van de automaten triviaal blijven, want er worden enkel transities van M_{E_1} naar M_{E_2} toegevoegd.
 - Als er een maximale subexpressie H^* is met $x \in \text{sym}(H')$, dan bevindt deze zich in één van de twee subexpressies en hiervoor geldt de inductiehypothese.
 - De orbit van de starttoestand is altijd triviaal, want er zijn door de definitie van de Glushkov automaat nooit transities die teruggaan naar de starttoestand.

¹Er is dus geen transitie van x naar zichzelf.

²Als x het enige element is, is er een transitie van x naar zichzelf.

³Deze transitie is respectievelijk wel en niet een eindtoestand.

2. Als er een maximale subexpressie H is met $x \in \text{sym}(H')$, dan bevindt deze zich in een van de twee subexpressies en hiervoor geldt de inductiehypothese. De concatenatie voegt geen transities binnen dezelfde subautomaten toe.
- $E_1 + E_2$: De twee automaten voor E_1 en E_2 hebben unieke toestanden. De starttoestand heeft transities naar $\text{first}(E'_1)$ en $\text{first}(E'_2)$. Verder zijn de toestanden van de twee automaten nooit met elkaar verbonden.
 1. – Als er geen dergelijke subexpressie H^* is, volgt uit de inductiehypothese dat één van de twee automaten voor de subexpressies een triviale orbit $\{x\}$ bevat. Deze orbit zal door het samenvoegen van de automaten triviaal blijven, want er worden geen transities toegevoegd naar toestanden van dezelfde automaat.
 - Als er een maximale subexpressie H^* is met $x \in \text{sym}(H')$, dan bevindt deze zich in een van de twee subexpressies en hiervoor geldt de inductiehypothese.
 - De orbit van de starttoestand is altijd triviaal, want er zijn door de definitie van de Glushkov automaat nooit transities die teruggaan naar de starttoestand.
 - 2. Als er een maximale subexpressie H is met $x \in \text{sym}(H')$, dan bevindt deze zich in een van de twee subexpressies en hiervoor geldt de inductiehypothese. De unie voegt geen transities binnen dezelfde subautomaten toe.
- E_1^* : De automaat heeft de transities voor E_1 , met transities toegevoegd gaande van $\text{last}(E'_1)$ naar $\text{first}(E'_1)$.
 1. – We kunnen steeds zo'n maximale subexpressie H^* vinden, namelijk E_1^* zelf.
 - Aangezien E trim is, zijn er geen nutteloze toestanden in M_{E_1} . Dit wil zeggen dat alle toestanden op een pad liggen van de starttoestand naar een eindtoestand. Dit pad wordt bepaald door de follow-informatie voor E'_1 . Dit wil ook zeggen dat er een pad is van x naar een element van $\text{last}(E'_1)$. Vanuit deze toestand is er een transitie naar elke toestand in $\text{first}(E'_1)$. En van hieruit is er dus een pad naar elke toestand. Dezelfde redenering gaat op voor elke toestand zodat er voor elke toestand een pad is naar x . Hieruit volgt dat $O(x) = \text{sym}(H')$.
Deze orbit zal ook nooit triviaal zijn, want als x de enige toestand is buiten de starttoestand, zal de constructie van de automaat voor E_1^* ervoor zorgen dat er een transitie van x naar zichzelf wordt toegevoegd.
 - De orbit van de starttoestand is altijd triviaal, want er zijn door de definitie van de Glushkov automaat nooit transities die teruggaan naar de starttoestand.
 - 2. E_1 is een maximale subexpressie onder een $*$ met $\text{sym}(H') \neq \emptyset$, want ze bevat x . Aangezien er geen toestanden buiten de orbit zijn, zijn de gates volgens hun definitie de eindtoestanden. En deze zijn volgens de definitie van de Glushkov automaat gelijk aan $\text{last}(H')$.
Aangezien $M_E = M_{H^*}$ zitten alle transities tussen toestanden in $\text{sym}(H')$ ook al in M_E .

□

Auteursrechterlijke overeenkomst

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

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

Smart Schema Validator

Richting: **Master in de informatica**

Jaar: **2007**

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

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

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

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

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

Ik ga akkoord,

Peter Boyen

Datum: **23.05.2007**