Universiteit Hasselt

Bachelorproef voorgedragen tot het behalen van de graad van bachelor in de informatica/ICT/kennistechnologie

DNAQL Simulator

Aangeboden door:
Tom Desair

Promotor:
Prof. Dr. Jan Van den Bussche
Begeleider:
Joris Gillis

Academiejaar 2010 - 2011

Abstract

DNA computing is een onderzoeksgebied binnen de informatica en de biologie waarbij men gebruik maakt van biologische moleculen voor het uitvoeren van algoritmen. Hiervoor wordt er meestal gebruik gemaakt van DNA, een molecule die al onze overerfbare informatie codeert en mee de gehele werking van ons organisme bepaalt. Door zijn robuustheid en kleine afmetingen is DNA uiterst geschikt voor het bewaren van informatie, wat in principe ook zijn natuurlijke functie is.

Aan de Universiteit Hasselt ontwikkelde men een formeel datamodel voor databases, die gecodeerd zijn met DNA, en een bijhorende query-taal DNAQL. In deze bachelorproef wordt dieper ingegaan op dit datamodel alsook op de benodigde voorkennis om dit model te begrijpen. We gaan de volledige werking van het model na en kijken of er ergens problemen optreden. Er werd bovendien ook een interpreter/simulator voor de DNAQL query-taal geïmplementeerd die hier ook besproken wordt. Deze interpreter stelt anderen in staat om gemakkelijk de DNAQL taal te leren kennen en leidde voor mijzelf tot een beter begrip van het gehele model. Bij het maken van de interpreter werd het datamodel ook lichtjes gewijzigd.

Tijdens het implementeren en schrijven van deze bachelorproef kwamen er enkele problemen en moeilijkheden naar boven. Deze problemen hadden vooral te maken met de operaties binnen DNAQL en dan in het bijzonder bij de hybridisatie van DNA, met name het bepalen of deze al dan niet eindig is en het zoeken naar een efficiënte manier om deze te berekenen. Voor sommige van deze problemen werd er een oplossing gevonden, andere zijn een onderwerp voor verder onderzoek.

Voorwoord

De tekst die hier voor u ligt, is het resultaat van mijn bachelorproef die ik in het academiejaar 2010 – 2011 heb voorgelegd voor het behalen van mijn bachelordiploma Informatica aan de Universiteit Hasselt. Het datamodel waar deze tekst over handelt werd ook aan de Universiteit Hasselt ontwikkeld in het kader van het doctoraat van de heer Joris Gillis. Dit model is nog maar zeer recent ontworpen en zal in de toekomst hoogstwaarschijnlijk nog sterk verder ontwikkeld worden en veranderen. Zelfs tijdens het maken van deze proef onderging het model reeds wijzigingen. Het feit dat deze bachelorproef samenhangt met dergelijk recent onderzoek, maakte het voor mij zeer speciaal en aangenaam om hieraan te werken.

Ik wil dan ook in eerste instantie mijn begeleider Joris Gillis uitvoerig bedanken voor het veelvuldig nalezen van dit proefschrift en voor de uitstekende begeleiding die hij mij geboden heeft bij het uitvoeren van deze bachelorproef. Een tweede woord van dank gaat uiteraard ook uit naar mijn promotor professor Jan Van den Bussche voor zijn begeleiding, aanwijzigen en motiverende gesprekken. Ik zou ook graag Balazs Nemeth, tweede bachelor student Informatica aan de Universiteit Hasselt, willen bedanken voor zijn hulp bij het teken-algoritme. Tenslotte wil ik ook mijn ouders en mijn vriendin danken voor het eveneens veelvuldig nalezen van mijn tekst, ondanks dat ze de inhoud niet altijd begrepen, en voor hun steun en aanmoediging.

Inhoudsopgave

Lijst van figuren

Lijst van tabellen

Lijst van listings

Inleiding

Deze bachelorproef handelt over DNA computing en meer specifiek over een databasemodel binnen DNA computing. DNA computing is een relatief nieuwe techniek waarbij men biologische molecules gebruikt voor het uitvoeren van berekeningen en algoritmen. Vanuit een database-perspectief is DNA zeer interessant omdat het enerzijds zeer klein is en anderzijds enorm robuust. Aan de Universiteit Hasselt ontwikkelt men momenteel een formeel databasemodel dat past binnen DNA computing. Dit model heet het *"sticker complex datamodel"*. Er werd hiervoor ook een querytaal ontwikkeld zodat de data ondervraagd en gemanipuleerd kan worden.

Het doel van deze bachelorproef is tweeledig. Enerzijds is er het theoretische gedeelte dat bestaat uit het in begrijpbaar Nederlands uitleggen van DNA computing en het sticker complex datamodel met al zijn operaties. Anderzijds is er het implementatie gedeelte waarin er een simulator/interpreter geschreven werd voor de volledige DNAQL programmeertaal. Om deze simulator te kunnen realiseren, werd er ook een bestandsformaat en een visualisator ontwikkeld.

U vraagt zich misschien af waarom de titel van mijn bachelorproef *DNAQL Simulator* is en niet *DNAQL Interpreter*. Dit zou inderdaad niet onlogisch zijn. Mijn implementatie gaat, net zoals andere interpreters, een DNAQL programma omzetten naar een abstracte syntaxboom om deze vervolgens te 'interpreteren'. Het verschil zit hem echter in het feit dat mijn implementatie de operaties van het sticker complex datamodel slechts simuleert en niet werkelijk uitvoert op echte DNA-strengen. De operaties gebeuren dus op computermodellen van DNA-strengen die dan op het scherm of in een bestand kunnen worden weergegeven.

De opbouw van dit proefschrift is als volgt: Vooraleerst geef ik een korte bespreking van het DNA molecule en de processen die hieraan gerelateerd zijn. Daarna ga ik even in op wat DNA computing juist is en hoe het ontstaan is. Vervolgens begin ik met de eigenlijke uiteenzetting van het sticker complex datamodel en de DNAQL programmeertaal. Tenslotte overloop ik het gebruik van de simulator en het ontworpen bestandsformaat, alsook de verschillende geïmplementeerde datastructuren en algoritmen. Op het einde vindt u mijn conclusies over het sticker complex datamodel en mijn bijhorende implementatie.

Hoofdstuk 1

Structuur en processen van DNA

Omdat DNA computing bijna volledig steunt op DNA en bijhorende technieken, is het belangrijk dat u een goede notie hebt van dit molecule. Deze sectie geeft een beknopte uiteenzetting over DNA en bijhorende processen. Ik behandel enkel de grote lijnen die nodig zijn voor een beter begrip van dit proefschrift. Indien u reeds vertrouwd bent met de structuur en processen van DNA, kan u dit hoofdstuk gerust overslaan en verder gaan naar hoofdstuk 2.

1.1 Structuur en functie van DNA

DNA is de afkorting voor *'deoxyribonucleic acid'* (of desoxyribonucleïnezuur in het Nederlands) en speelt een belangrijke rol in de genetica [Rus10]. De basisbouwsteen van een DNA molecule is de nucleotide. Een nucleotide bestaat uit een suiker en een fosfaat-groep waaraan een base gekoppeld is. Binnen een DNA molecule zijn er slechts vier mogelijke basen: adenine (A), cytosine (C), guanine (G) en thymine (T). De structuur van deze basen en hun nucleotiden is weergegeven in figuur 1.1. De suikers en fosfaatgroepen van de verschillende nucleotiden vormen binnen een DNA molecule een aaneengesloten keten die ook wel een suiker-fosfaat ruggengraat of een fosfaat-deoxyribose ruggengraat genoemd wordt [Rus10, CRU$^+$08]. De term 'ruggengraat' hebben ze te danken aan het feit dat tussen de verschillende nucleotiden enkel de gekoppelde basen eventueel verschillen. De oriëntatie[1] van de nucleotiden bepaalt ook de oriëntatie van de afzonderlijke ketens die we ook wel DNA-strengen noemen. Het uiteinde van de streng waar zich een fosfaat bevindt, noemen we het 5'-uiteinde. Het uiteinde waar de suiker zit, noemen we het 3'-uiteinde [CRU$^+$08]. Op deze manier kan men aangeven of we de DNA-streng vanaf het 3'-uiteinde naar 5'-uiteinde lezen (wat we een 3'-5' oriëntatie noemen) of vanaf het 5'-uiteinde naar het 3'-uiteinde (een 5'-3' oriëntatie). Een n-letter sequentie van opeenvolgende basen wordt een

[1] Als we van links naar rechts kijken, komen we dan eerst de suiker tegen of eerst de fosfaatgroep?

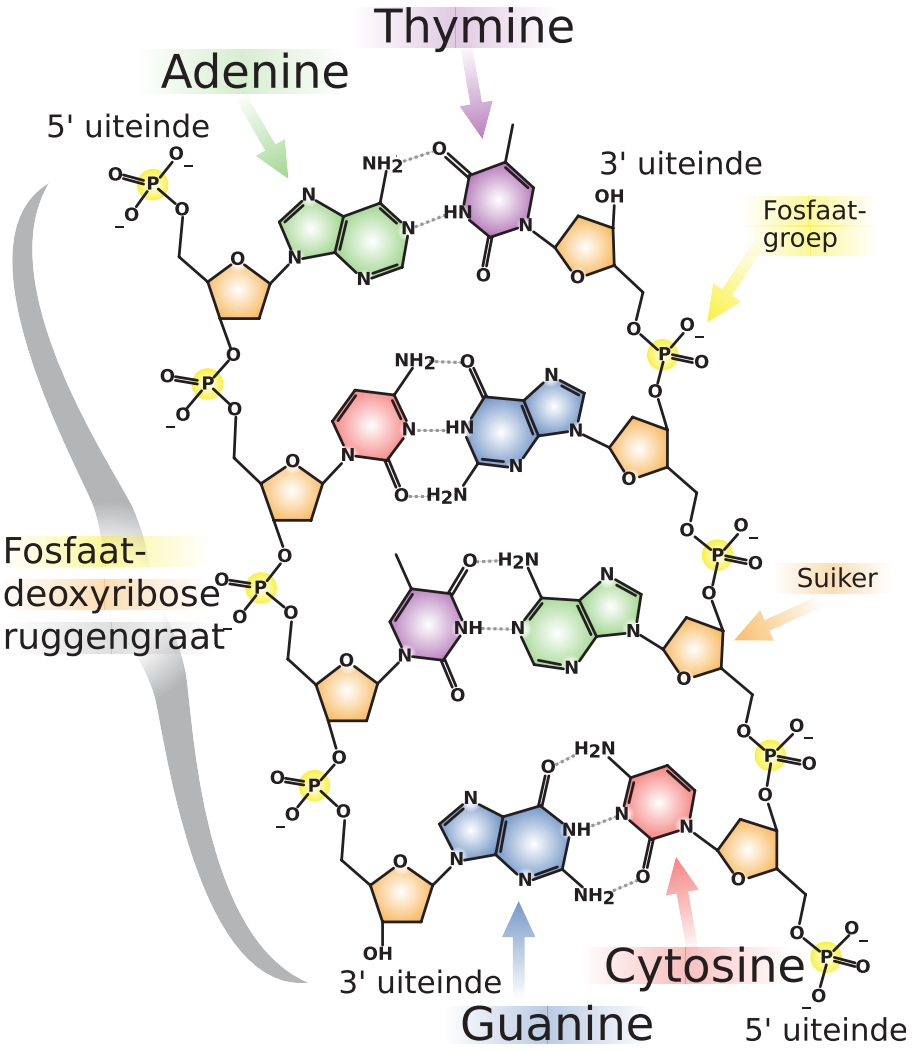
n-mer of een *oligonucleotide* van lengte n genoemd. Oligonucleotide korten we meestal af met *oligo*. De lengte van een DNA-streng wordt gemeten in aantal basen.

Twee DNA-strengen kunnen in de breedte aan elkaar gekoppeld worden met behulp van waterstofbruggen tussen de tegenover elkaar liggende basen. In figuur 1.1 worden de waterstofbruggen aangegeven met een stippellijn. Uit de moleculaire opbouw van de basen kunnen we afleiden dat er enkel waterstofbruggen mogelijk zijn tussen A en T (met telkens twee waterstofbruggen) en tussen G en C (met telkens drie waterstofbruggen) [CRU$^+$08]. In een DNA molecule komen tussen de twee DNA-strengen dus enkel de basenparen A-T, T-A, G-C en C-G voor. Dit noemt men de *Watson-Crick base pairing* naar de onderzoekers James D. Watson en Francis Crick die als eersten de structuur van een DNA molecule vastlegden [Wri99]. We noemen de sequentie van basen van de ene DNA-streng *complementair* aan de sequentie van basen van de andere DNA-streng. Waterstofbruggen zijn bindingen waarbij een waterstofatoom "gesandwiched" wordt tussen twee elektronegatieve atomen (bij DNA is dit stikstof of zuurstof) [Rus10]. De sterkte van een waterstofbrug is slechts 1/20 van een gewone atoombinding waardoor deze makkelijker (en sneller) verbroken kan worden [CRU$^+$08]. Dit is een belangrijke eigenschap die het mogelijk maakt om twee DNA-strengen van elkaar te scheiden zonder de suiker-fosfaat ruggengraat te breken. Merk op dat twee aan elkaar gekoppelde DNA-strengen een tegengestelde oriëntatie hebben. Indien we de molecule van links naar rechts zouden lezen, dan heeft de ene streng een 3'-5' oriëntatie en de andere een 5'-3' oriëntatie. Het is echter niet vereist dat de afzonderlijke DNA-strengen over de volledige breedte aan elkaar gekoppeld zijn. Twee strengen die (gedeeltelijk) gekoppeld zijn noemen we "double-stranded DNA". Indien een DNA-streng met geen enkele andere DNA-streng gekoppeld is, noemen we hem "single-stranded DNA".

De twee strengen in double-stranded DNA zijn rond elkaar verstrengeld tot een dubbele helix en vormen het eigenlijke DNA molecule. De chromosomen van een organisme bestaan uit verschillende van deze helices. Hierin zijn al de genetische eigenschappen van dat organisme opgeslagen. Met genetische eigenschappen bedoel ik al de eigenschappen van een organisme, zowel innerlijke als uiterlijke, die van ouder op kind kunnen worden doorgegeven. Deze 'informatie' is gecodeerd met behulp van de basen in de verschillende DNA molecules. De chromosomen bepalen dus voor een groot deel het uiterlijk en de werking van elk organisme!

Maar hoe wordt de genetische informatie in de basen omgezet naar specifieke kenmerken van een cel? In de kern van elke cel, dat meestal omgeven is door een membraan, wordt (een deel van) deze genetische informatie gekopieerd met behulp van mRNA moleculen. RNA staat voor *'ribonucleic acid'* (of ribonucleïnezuur in het Nederlands) en de m staat voor *'messenger'*. mRNA is immers de 'boodschapper' die de informatie van de kern naar de rest van de cel zal brengen. RNA is zeer gelijkaardig aan DNA. De verschillen bestaan eruit dat RNA een andere suiker gebruikt (ribose in plaats van desoxyribose) en meestal in enkelvoudige (single-stranded) strengen voorkomt. Bij RNA zijn er ook maar vier mogelijke basen: adenine (A), cytosine (C), guanine (G) en uracil (U). T werd dus vervangen door U. Het enzym[2] RNA polymerase zal in de kern van de

[2]Een enzym is een soort proteïne dat een bepaalde reactie stimuleert, tegengaat of start.



Figuur 1.1: Structuur van een DNA molecule (bron: Wikipedia)

cel de informatie (gecodeerd door de basen) in de DNA moleculen kopiëren door een mRNA molecule te construeren waarbij de gekoppelde basen bepaald worden door de overeenkomstige basen in de DNA molecule [Rus10]. Op plaatsen waar de A base in de DNA molecule voorkomt, zal RNA polymerase een U base aan de mRNA molecule koppelen. Analoog gebeurt dit ook voor T met A, C met G en G met C. RNA polymerase zal op die manier een complementaire mRNA molecule produceren die al de (nodige) informatie codeert.

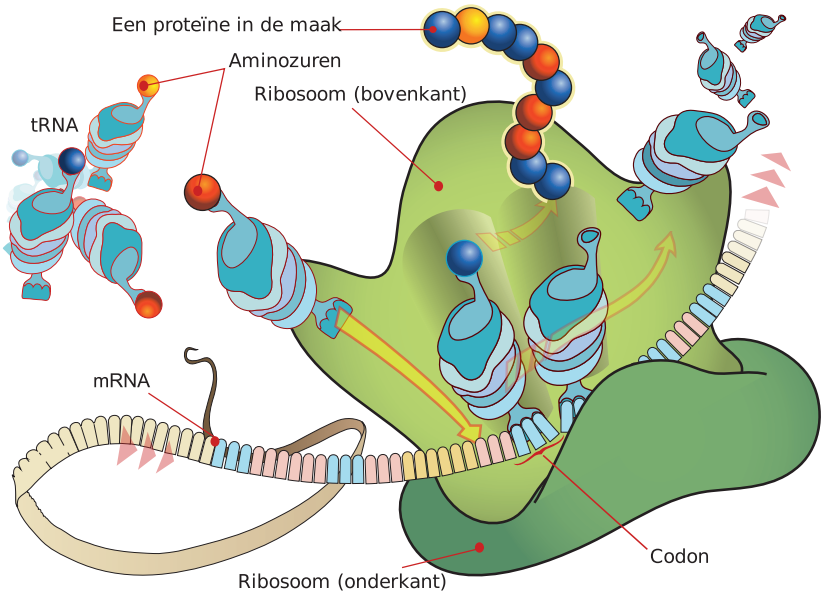
Het geconstrueerde mRNA molecule zal zich buiten de kern verplaatsen waar er ribosoom enzymen op binden. De ribosomen 'lezen' dan dit RNA molecule en bouwen op basis van de gelezen informatie een proteïne (ook wel eiwit genoemd). De basisbouwblokken voor een proteïne zijn aminozuren. Dit zijn kleine moleculen die bestaan uit onder andere koolstof-, waterstof-, zuurstof- en stikstofatomen. In de bouw van proteïnen worden slechts twintig aminozuren gebruikt [Rus10]. Elk aminozuur wordt in RNA gecodeerd door een combinatie van drie opeenvolgende basen. Zo een groepje van drie basen in RNA noemen we een *codon* en dient als universele code (over al de organismen heen) voor een bepaald aminozuur [CRU$^+$08]. De codons UAA, UAG en UGA vormen de stop-codons en geven het einde van de codering van elk proteïne aan. Met drie basen kunnen we ($4^3 =$) 64 woorden vormen. Zoals u kan zien hebben we veel meer mogelijke combinaties als mogelijke aminozuren. Toch worden alle 64 combinaties gebruikt waarbij verschillende combinaties eenzelfde aminozuur vertegenwoordigen. Dit is een soort van ingebouwde veiligheid voor als er iets mis gaat tijdens de transcriptie van het DNA.

Opdat een ribosoom het passende aminozuur bij een codon vindt, moet het gebruik maken van tRNA (waarbij de t staat voor *'transfer'*). Deze molecule bestaat uit een korte RNA streng waarbij er zich aan het ene uiteinde drie basen bevinden en aan het andere uiteinde een aminozuur gekoppeld is. Een ribosoom zal een RNA molecule lezen in groepjes van drie basen en aan elk codon een tRNA molecule (met aminozuur) koppelen. De basen van dit tRNA molecule moeten complementair zijn aan het huidige codon. Vervolgens zal het ribosoom het aminozuur aan de reeds gelezen aminozuren koppelen en een codon opschuiven waarbij de tRNA molecule terug wordt afgestoten. Als het ribosoom uiteindelijk een stop-codon leest, is het proteïne voltooid en zal het worden losgekoppeld van zijn ribosoom. Een schematische weergave van de werking van een ribosoom kan u zien in figuur 1.2.

We kunnen stellen dat er achter elke activiteit van levende cellen één of meerdere proteïnen zitten of, anders gezegd, dat het leven in feite een manifestatie van proteïnen is [dD87]. Omdat het DNA bepaalt welke proteïnen er worden aangemaakt, is het verantwoordelijk voor de gehele werking en functie van elke cel en van elk organisme.

1.2 Denaturatie en hybridisatie

Het proces waarbij we twee gekoppelde DNA-strengen (double-stranded DNA) van elkaar scheiden door de temperatuur in de proefbuis te verhogen, noemen we *denaturatie*. Door de hogere temperatuur worden de waterstofbruggen ver-



Figuur 1.2: Werking van een ribosoom (bron: Wikipedia)

broken en bekomen we opnieuw twee single-stranded DNA-strengen. Dit proces wordt ook wel smelten genoemd. De temperatuur waarbij denaturatie optreedt, noemt men het *smeltpunt*. Het smeltpunt kan bepaald worden uit de verhouding van het aantal GC-verbindingen tot het aantal AT-verbindingen. Een GC-verbinding is door de drie waterstofbruggen immers sterker dan een AT-verbinding met maar twee waterstofbruggen. Voor het DNA in de menselijke chromosomen ligt het smeltpunt ongeveer bij 85°C [dD87].

Het omgekeerde proces waarbij twee complementaire DNA-strengen terug met elkaar verbonden worden op basis van de *Watson-Crick base pairing*, heet *hybridisatie* of *renaturatie*. Dit proces initialiseert men door twee (deels-)complementaire DNA-strengen in eenzelfde oplossing te brengen (wat na denaturatie reeds het geval is) en de temperatuur te verlagen tot onder het smeltpunt [CRU+08, Rus10]. De twee DNA-strengen zullen zich dan 'automatisch' via basenparing opnieuw aan elkaar koppelen. Na hybridisatie zijn niet noodzakelijk alle basenparen met elkaar verbonden door waterstofbruggen. Indien de twee strengen niet volledig complementair zijn, zullen enkel de complementaire delen aan elkaar hybridiseren en zullen de overige delen 'open' blijven. Een dergelijke gedeeltelijke hybridisatie is enkel mogelijk als deze complementaire delen groot genoeg zijn.

1.3 Polymerase

Omdat de twee DNA-strengen van een DNA molecule complementair zijn, hebben we aan één streng genoeg om alle informatie in een DNA molecule te kennen. We kunnen voor een kleine DNA molecule op basis van één 'template' streng de volledige molecule (met twee strengen) herstellen. Het opnieuw opbouwen van een DNA molecule op basis van één DNA-streng gebeurt door een enzym dat we *DNA polymerase* noemen [CRU$^+$08, Rus10]. Dit enzym zal een bestaande suiker-fosfaat ruggengraat verder uitbouwen waarbij er telkens op basis van de *Watson-Crick base pairing* een juiste base gekoppeld wordt aan de reeds bestaande, tegenoverliggende DNA-streng. DNA polymerase speelt een belangrijke rol in de DNA replicatie bij celdeling.

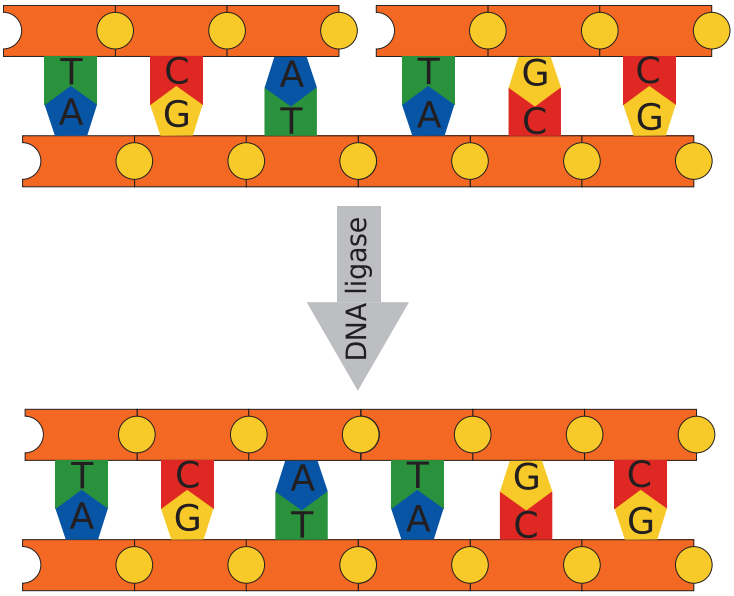
Om DNA polymerase te kunnen toepassen bij kunstmatige DNA synthese moeten we gebruik maken van een *primer* [CRU$^+$08]. De reden hiervoor is dat DNA polymerase enkel kan starten vanaf het 3'-uiteinde van een reeds bestaande DNA-streng. Maar als we te maken hebben met één enkel (naakte) DNA-streng, heeft DNA polymerase nergens een start-aanhechtingspunt. In een levende cel wordt hiervoor een ander enzym gebruikt, genaamd *primase*. Dit enzym zal een korte RNA streng produceren die complementair is aan het begin (3'-uiteinde) van de DNA-streng en deze hieraan vastkoppelen. Op deze manier ontstaat er een startpunt (een 3'-uiteinde) voor de DNA polymerase om de gehele DNA-streng aan te vullen. Deze korte RNA streng noemen we een *primer*. In een laboratorium werkt dit echter niet met primase en moeten we manueel de primer specificeren en synthetiseren. Hiervoor moeten we een begin-sequentie van de te behandelen DNA-streng kennen. De gesynthetiseerde primer kan dan aan het begin van de DNA-streng hybridiseren (zie sectie 1.2) zodat DNA polymerase zich kan vasthechten.

1.3.1 Polymerase Chain Reaction (PCR)

Een techniek die gebruik maakt van polymerase is PCR, wat staat voor *Polymerase Chain Reaction*. Dit is een eenvoudige en goedkope techniek voor het veelvuldig klonen van (een deel van) een DNA molecule waarbij een combinatie van technieken zoals denaturatie, hybridisatie, primers en polymerase aangewend wordt. Hierbij gebruikt men een linker en een rechter primer die het begin en einde aangeven van (het deel van) de te klonen molecule. De linker primer is hierbij complementair aan het 'startpunt' op de 3'-5' streng en de rechter primer complementair aan het 'eindpunt' op de 5'-3' streng. In elke iteratie van de 'kettingreactie' neemt het aantal klonen exponentieel toe. Voor de details van PCR verwijs ik naar [Mul90, Rus10]. Vroeger moest men DNA klonen met behulp van levende bacteriën maar dankzij PCR is dit nu veel eenvoudiger.

1.4 Ligase

Met behulp van het enzym DNA ligase kunnen we twee enkele DNA-strengen, die naast elkaar liggen op een derde enkele DNA-streng, aan elkaar koppelen



Figuur 1.3: Werking van DNA ligase

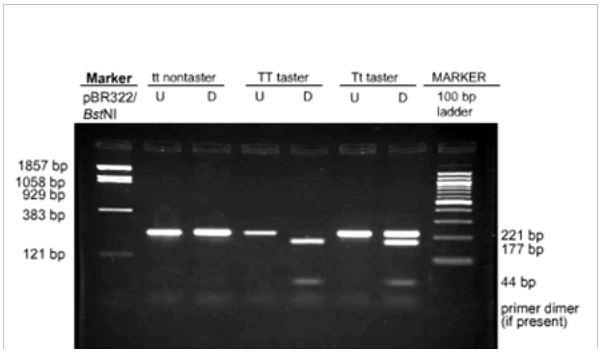
door hun suiker-fosfaat ruggengraten met elkaar te verbinden [Rus10]. Deze situatie is geschetst in figuur 1.3. DNA ligase sluit de gaten in de suiker-fosfaat ruggengraat die ontstaan nadat verschillende DNA-strengen gehybridiseerd zijn op eenzelfde ander DNA-streng. Dit gebeurt door het vrije 5' fosfaat uiteinde van de ene streng te verbinden met het vrije 3' suiker uiteinde van de andere streng. Na het toepassen van DNA ligase is de DNA molecule volledig gevormd.

1.5 Lengte van een DNA molecule bepalen

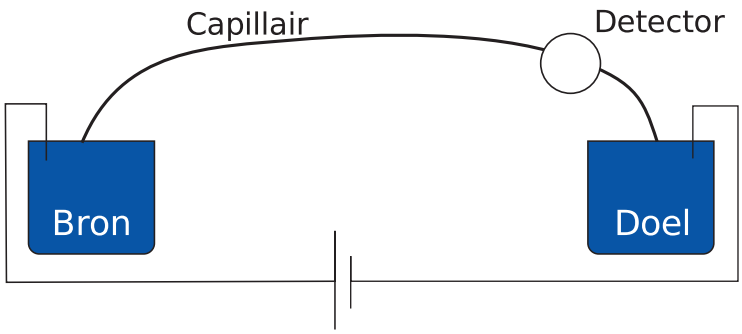
Een veel gebruikte techniek voor het sorteren van DNA moleculen op lengte is het gebruik van *agarose gelelektroforese* [Rus10]. Hierbij wordt er gebruik gemaakt van een elektrisch veld om de negatief geladen[3] DNA moleculen te bewegen van een negatieve pool naar een positieve pool door een gel-matrix met agarose[4]. Deze gel is verdeeld in verschillende lanen waar we verschillende stalen in kunnen aanbrengen. Bovendien voegt men ook DNA moleculen van gekende lengte toe in een aparte laan die moeten dienen als "lengtemarkeringen". De concentratie van agarose bepaalt de grootte van de poriën in de gel die fungeren als een 'zeef'. Als het elektrische veld dan wordt geactiveerd, zullen de kleine DNA moleculen sneller en verder doorheen deze 'zeef' bewegen dan de groteren. Men kan dit vergelijken met een klein kind en een groter kind in een binnenspeeltuin. Het kleine kind zal zich veel gemakkelijker en sneller doorheen de speeltuin bewegen als het grotere kind. De DNA moleculen sorteren zichzelf

[3]De DNA moleculen zijn negatief geladen omdat de gel waarin ze zich bevinden een pH-waarde groter als acht heeft.

[4]Agarose is een molecule die bestaat uit een keten van suikers en wordt gebruikt in de productie van agar, een gelatineachtige stof die onder andere in voedingsmiddelen zit.



Figuur 1.4: Agarose gelelektroforese



Figuur 1.5: Opstelling capillaire gelelektroforese

op deze manier volgens grootte (zie figuur 1.4). Hierna kunnen we eenvoudig de gewenste DNA moleculen uit de gel halen.

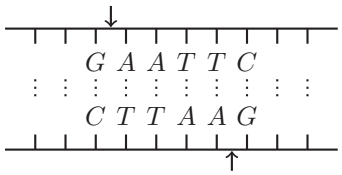
Een nadeel van gelelektroforese is dat om de DNA moleculen te kunnen zien, we ze moeten merken met een radioactieve stof of een kleuring van de gel moeten toepassen. Bovendien is het aantal opeenvolgende nucleotiden bij gelelektroforese beperkt tot 600 per uitvoering [SG90]. Een andere techniek die dit laatste nadeel niet heeft is *capillaire gelelektroforese*. Deze techniek werkt op basis van een capillair (een zeer dun buisje met een inwendige diameter die kleiner is als 0.5 mm) gevuld met een gel en een elektrisch veld (zoals in bovenstaande techniek). De werking van capillaire elektroforese is gelijkaardig aan die van een gelelektroforese met als verschil dat de te scheiden moleculen door een capillair geleid worden en dat we gebruik maken van een detector. Een schets van de opstelling bij capillaire elektroforese ziet u in figuur 1.5. De moleculen bevinden zich bij de start in het 'Bron'-reservoir en worden via het capillair naar het 'Doel'-reservoir geleid. Als de radioactief gemerkte moleculen tijdens deze overgang langs de detector voorbijgaan, registreert deze de moleculen voor automatisch verwerking. Capillaire gelelektroforese werkt sneller, kan een kleiner verschil in basen detecteren en is het toepasbaar op grotere hoeveelheden moleculen dan de klassieke gelelektroforese [SG90].

1.6 Immobilisatie

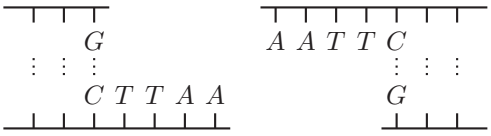
Immobilisatie van DNA is eveneens een veel gebruikte techniek in DNA computing. Hierbij bindt men een DNA molecule (na modificatie van bepaalde atomen) aan een oppervlak [QL00], of aan het *biotin-avidin complex* dat gebonden is aan een magnetische korrel [LFDR87, Amo05]. Als een DNA molecule geïmmobiliseerd is, wil dit zeggen dat deze niet vrij rond kan zweven in de oplossing waarin de molecule zich bevindt. Bovendien kunnen we geïmmobiliseerde DNA moleculen gemakkelijk uit een oplossing filteren door gebruik te maken van de magnetische korrels of van het bindingsoppervlak.

1.7 Restrictie enzymen

Restrictie enzymen zijn enzymen die een DNA molecule kunnen splitsen door de suiker-fosfaat ruggengraten op bepaalde locaties te breken. Restrictie enzymen danken hun naam aan het feit dat ze één van de natuurlijke afweermechanismen tegen vreemd (en soms kwaadaardig) DNA zijn door het vreemde DNA te verknippen. De introductie van vreemd DNA in een cel gebeurt bijvoorbeeld door virussen. Restrictie enzymen "beperken" met andere woorden de compatibiliteit van gastheer en indringer [dD87]. Het restrictie enzym herkent een bepaalde sequentie van basen-paren in het DNA, wat we de *restrictie-enzymherkenningsplaats* noemen, en splitst vervolgens de DNA molecule binnen of in de buurt van deze sequentie [Rus10]. De meeste restrictie-enzymherkenningsplaatsen hebben in hun twee strengen meestal dezelfde basensequentie die door de tegengestelde oriëntatie op elkaar hybridiseren. Ze vormen als het ware een 'palindroom'. Indien de splitsing asymmetrisch gebeurt (de twee suiker-fosfaat ketens worden op verschillende hoogte verbroken), dan hebben de bekomen DNA fragmenten *sticky* uiteinden. Indien de splitsing symmetrisch is (de twee suiker-fosfaat ketens worden op dezelfde hoogte verbroken), dan spreken we van *blunt* uiteinden. U ziet hiervan voorbeelden in figuur 1.6 en in figuur 1.7.

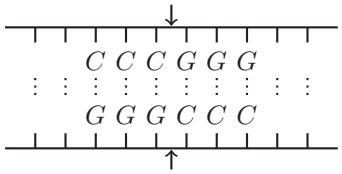


(a) DNA molecule met aangeduide splitspunten



(b) Het DNA molecule na de splitsing

Figuur 1.6: DNA splitsing met het restrictie enzym EcoRI (*sticky* uiteinden)



(a) DNA molecule met aangeduide splitspunten

(b) Het DNA molecule na de splitsing

Figuur 1.7: DNA splitsing met het restrictie enzym SmaI (*blunt* uiteinden)

Momenteel zijn er meer dan 400 restrictie enzymen volledig gekend en zijn er minstens 2000 gedeeltelijk gekend [Rus10]. Ze spelen bovendien niet alleen een belangrijke rol in de natuur, maar zijn ook van groot belang bij het 'snijden' en klonen van DNA, het 'mengen' van DNA en bij erfelijkheidsexperimenten.

1.8 Uitlezen en wegschrijven van DNA

De twee laatste concepten die van groot belang zijn bij DNA computing is het produceren en lezen van DNA. Dit is immers nodig om gegevens te kunnen bewaren en berekende resultaten te kunnen interpreteren. Om van een gegeven DNA molecule de sequentie van de basen te bepalen, bestaan tegenwoordig vele goedkope en snelle technieken [CRU+08]. Dit proces heet *DNA sequencing* en wordt gerealiseerd met behulp van een DNA sequencer. Eén van de laatste nieuwe DNA sequencing technieken werkt op basis van een wegwerp-siliciumchip die zeer goedkoop geproduceerd kan worden. Uiteindelijk zouden de kosten van het sequencen hierdoor enorm kunnen dalen [Dij11].

De techniek voor het maken van DNA moleculen met een vooraf bepaalde volgorde van basen heet *DNA synthese*. Ook hiervoor bestaan er verschillende technieken. Het apparaat dat hierbij gebruikt wordt, is een DNA synthesizer. De langste DNA molecule die men tot nu toe kunstmatig heeft kunnen aanmaken is 582.970 basen lang [Bal08]. Voor de meeste toepassingen is dit aantal uiteraard voldoende maar voor grote data hoeveelheden zal dit waarschijnlijk ontoereikend zijn. Een mogelijke oplossing zou eruit bestaan om de hoeveelheid data op te splitsen over meerdere moleculen. Maar waarschijnlijk zal de technologie nog verbeteren zodat het aantal mogelijke basen nog zal stijgen. Indien men het synthetiseren van DNA uitbesteed, kost dit ongeveer €0.25 per basenpaar.

Hoofdstuk 2

Geschiedenis van DNA computing

Het is ook belangrijk om te begrijpen waar DNA computing vandaan komt en waarom men onderzoek voert naar DNA computing, twee aspecten waar ik in deze sectie dieper op inga. Een goede manier om dit te doen, lijkt mij via een beschrijving van het allereerste DNA computing experiment dat aan de basis lag van de start van DNA computing onderzoek. Op het einde van deze sectie besluit ik met nog enkele voorbeelden van DNA computing experimenten.

2.1 Het wat en waarom van DNA computing

Martyn Amos geeft aan het concept DNA computing volgende definitie [Amo09]:

> "DNA computing (of meer in het algemeen, biomolecular computing) is een relatief nieuw onderzoeksveld dat handelt over het gebruik van biologische molecules als fundamentele componenten voor computersystemen."

Zoals u kan zien is dit een zeer brede definitie. Het is dan ook zeer moeilijk om het begrip DNA computing exact te definiëren. Dit komt omdat er binnen het DNA computing onderzoeksveld verschillende stromingen zijn die elk een eigen werkwijze toepassen bij het gebruiken van biologische molecules voor het uitvoeren van berekeningen. Zo gebruiken sommige onderzoekers specifieke DNA computing technieken voor een specifiek probleem terwijl anderen algemenere modellen proberen te ontwikkelen die toepasbaar zijn op verschillende problemen (zoals bijvoorbeeld het simuleren van een computerchip [OR96]). DNA computing is dus meer een verzamelnaam voor alle technieken waarbij we biologische moleculen, en in het bijzonder DNA moleculen, gebruiken voor het uitvoeren van berekeningen. Hoewel de eerste ideeën hierrond reeds geopperd werden in de jaren '50 [Fey61], werd het eerste succesvolle experiment pas gerealiseerd in 1994 door Leonard Adleman bij het oplossen van het 'Hamilton

pad'-probleem [Adl94]. Dat was de officiële start van een nieuw boeiend onderzoeksgebied.

Waarom zouden we biologische moleculen gebruiken voor het uitvoeren van berekeningen? We kunnen deze vraag beantwoorden door het identificeren van twee knelpunten in de huidige computertechniek. Het eerste knelpunt bevindt zich in het gebruik van silicium chips. De miniaturisatie van de silicium chips van de afgelopen jaren heeft geleid tot een enorme vooruitgang in (processor-) snelheid. Er staat echter een limiet op de kleinst mogelijke schaal dat silicium chips kunnen worden geproduceerd. Dit komt door het *Heisenberg Uncertainty Principle* (HUP). Het HUP stelt dat indien computerchips uit slechts enkele atomen bestaan, kwantumeffecten zo een invloed kunnen hebben dat bij het observeren van de toestand van de chip, deze toestand mogelijk verandert [Adl94]. Bijgevolg kunnen we nooit de toestand aan de uitgang van een chip (waarde 0 of 1) met zekerheid kennen en zijn deze chips niet bruikbaar voor het bouwen van een computersysteem.

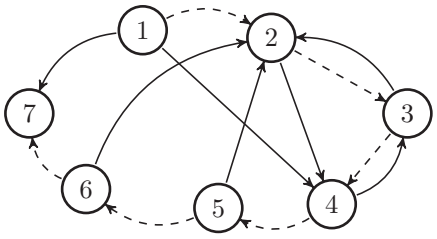
Een tweede knelpunt ligt in de *Von Neumann*-architectuur dat in (bijna) al de huidige computersystemen gebruikt wordt. Het wordt dan ook de *Von Neumann bottleneck* genoemd [Amo05]. Deze architectuur schrijft voor dat zowel het uit te voeren programma als de data voor dat programma zich in hetzelfde geheugen moeten bevinden. Bijgevolg moet de CPU telkens een instructie uit het geheugen halen, daarna eventueel data uit datzelfde geheugen halen en tenslotte het resultaat van de instructie terug in het geheugen plaatsen. De snelheid waarmee de CPU met het geheugen kan interageren is dus ook een beperkende snelheidsfactor. Computergeheugen bestaat immers ook uit silicium chips die gelimiteerd zijn in snelheid wegens het HUP.

Omdat we met de huidige techniek stilaan tegen deze limieten aanlopen, zijn bovenstaande knelpunten de grootste drijfveren voor het onderzoek naar DNA computing. DNA werkt immers op een veel kleinere schaal dan de huidige chips, zonder onderhevig te zijn aan het HUP. DNA werkt bovendien op een enorm geparallelliseerde manier. Al de operaties (zie hoofdstuk 1) kunnen immers op verschillende (duizenden) DNA-strengen tegelijkertijd gebeuren. Maar ook andere computing paradigma worden onderzocht als mogelijke vervanging voor de silicium chips. Zo voert men onder meer onderzoek naar quantum computing, optical computing, nanocomputers ... [Amo09]. Naast deze algemene motivatie kunnen we soms ook een modelspecifieke motivatie afleiden maar hier gaan we later dieper op in.

2.2 Het eerste DNA computing experiment

De eerste geautomatiseerde berekening waarbij gebruik gemaakt werd van DNA werd uitgevoerd door Leonard Adleman in 1994 [Adl94]. Hij berekende met behulp van DNA een oplossing van het 'Hamilton Pad'-probleem voor een gerichte graaf met zeven knopen. Het 'Hamilton Pad'-probleem bestaat uit het vinden van een pad in een gegeven graaf zodat dit pad precies eenmaal langs elke knoop van die graaf loopt. Bovendien is het 'Hamilton Pad'-probleem een NP-compleet probleem [Sip06].

De graaf waarvoor Adleman het 'Hamilton Pad'-probleem oploste, is gegeven in figuur 2.1 waarbij het unieke Hamilton pad aangegeven wordt door de gestippelde bogen. Aldeman gebruikte volgende stappen om dit probleem op te lossen met DNA [Adl94, Amo09]:



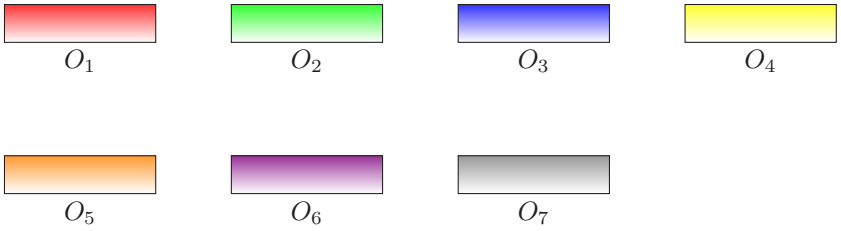
Figuur 2.1: De graaf van het eerste DNA computing experiment

Stap 1: Aan elke knoop i in de graaf werd een willekeurige oligo (DNA-streng) met lengte 20 toegewezen, die we hier O_i noemen (zie figuur 2.2). Aan elke boog, tussen een knoop i en een knoop j, werd ook een oligo $O_{i\to j}$ van lengte 20 toegewezen waarvan de eerste tien basen complementair waren aan de laatste tien basen van O_i en de laatste tien basen complementair waren aan de eerste tien basen van O_j. Merk op dat op deze manier ook de gerichtheid van de bogen bewaard wordt want $O_{3\to 4}$ is niet gelijk aan $O_{4\to 3}$. Deze situatie wordt geschetst in figuur 2.3 waarbij de complementariteit wordt aangegeven door de omgekeerde gradiëntie (wit vanboven of wit vanonder). Ik zal dit voorbeeld verder uitwerken voor de eerste twee knopen. Stel dat O_1 gelijk is aan de oligo *AGTCAGTCAGTCAGTCAGTC* en O_2 gelijk is aan **CATGCATGCATGCATGCATG**. Wegens constructie is oligo $O_{1\to 2}$ (de boog tussen knoop 1 en 2) gelijk aan: *AGTCAGTCAG***GTACGTACGT**. Het pad van knoop 1 naar knoop 2 vormt zich dan op volgende manier:

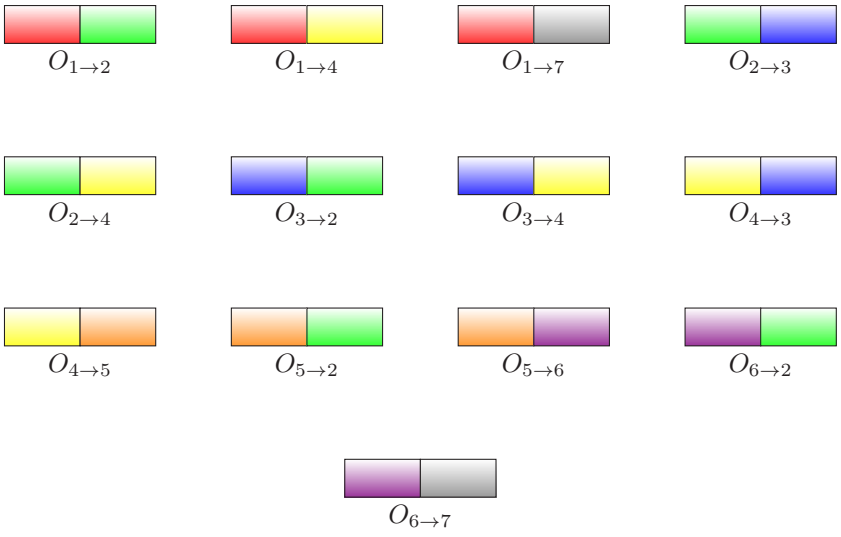
*AGTCAGTCAG***GTACGTACGT**
*AGTCAGTCAGTCAGTCAGTC***CATGCATGCATGCATGCATG**

Vaste hoeveelheden van elk van deze oligo's, zowel voor knopen als voor bogen, werden in een oplossing gebracht zodat hybridisatie (zie sectie 1.2) kon plaatsvinden. Daarna werd er ligase (zie sectie 1.4) toegevoegd zodat er complete DNA moleculen ontstonden. Op deze manier worden alle mogelijke paden in de graaf gegenereerd in DNA moleculen. Door de grote hoeveelheid oligo's is bovendien de kans dat het Hamilton pad zich tussen deze paden bevindt, zeer groot. Deze aanpak lost op deze manier het probleem op van het genereren van een exponentieel aantal verschillende paden door gebruik te maken van een slechts polynomiaal aantal initiële oligo's [Amo09].

Stap 2: Het eerste deel van de tweede stap bestaat uit het toepassen van PCR (zie sectie 1.3) om het aantal DNA moleculen die een pad representeren van knoop 1 naar knoop 7 sterk te doen toenemen zodat ze de andere paden overheersen in aantal. Hiervoor gebruikte men het complement van O_1 als linker primer en O_7 als rechter primer in de PCR reactie om zo de juiste DNA moleculen voor reproductie te selecteren. Het resultaat van deze PCR werd daarna met



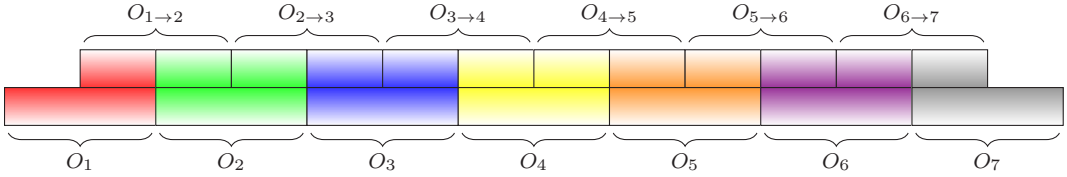
Figuur 2.2: DNA codering van de graafknopen waarbij elke kleur een bepaald DNA streng voorstelt



Figuur 2.3: DNA codering van de bogen

behulp van agarose gelelektroforese (zie sectie 1.5) gesorteerd op lengte zodat men al de DNA moleculen van lengte 140 (paden die zeven knopen bevatten) kon uitsorteren. Deze uitgesorteerde DNA moleculen werden dan uitgezuiverd via affiniteitszuivering met een biotin-avidin complex gekoppeld aan magnetische korrels (zie sectie 1.6) [Adl94]. Hierbij werd het double-stranded DNA via denaturatie gesplitst in single-stranded DNA stengen. Vervolgens voegde men een oligo toe die complementair is aan O_2 en gekoppeld is aan een magnetische korrel (met behulp van het biotin-avidin complex). Al de DNA strengen die knoop 1 bevatten, hybridiseerden vervolgens aan deze oligo en werden met behulp van de magnetische korrel uitgefilterd. Dit proces werd herhaald voor O_3, O_4, O_5 en O_6. Op deze manier elimineerde men alle paden die niet alle knopen bevatten.

Stap 3: In de laatste stap werd PCR gebruikt om de DNA molecule die het unieke Hamilton pad voor de gegeven graaf codeerde, te identificeren. Voor een graaf met n knopen, moet men $n-1$ PCR reacties uitvoeren waarbij het complement van O_1 telkens als linker primer gebruikt wordt en O_i als rechter primer in de i^{de} PCR reactie. In het huidige voorbeeld moet de DNA molecule die het unieke Hamilton pad representeert, in deze $n-1$ (hier dus zes) PCR reacties DNA moleculen genereren van lengte 40, 60, 80, 100, 120 en 140. Enkel de DNA moleculen die exact deze resultaten geven over de $n-1$ PCR reacties heen, geven de juiste oplossing omdat zij dan exact elke knoop juist eenmaal bezoeken. Als we de gebruikte rechtse primers dan sorteren volgens de lengte van de DNA moleculen die ze genereren, dan kunnen we de volgorde van de knopen in het Hamilton pad bepalen. De DNA molecule die het unieke Hamilton pad in dit voorbeeld representeert, is schematisch weergegeven in figuur 2.4.



Figuur 2.4: DNA codering die de uiteindelijke oplossing voorstelt

Op deze manier loste Leonard Aldeman het 'Hamilton pad'-probleem voor deze graaf op en bood hij ook een algemene werkwijze om dit probleem op te lossen voor andere grafen. Het voordeel van deze werkwijze is het sterke parallelle karakter waarmee we op een snelle en parallelle manier alle mogelijke paden in de graaf kunnen genereren alsook snel al deze paden tegelijkertijd filteren. Dit is duidelijk één van de voordelen van DNA computing bij het oplossen van NP-complete problemen.

Er moet hier echter wel een kanttekening bij gemaakt worden. Juris Hartmanis berekende immers dat als we ditzelfde probleem zouden oplossen voor een graaf van 200 knopen, de initiële verzameling DNA-strengen nodig voor het oplossen van dit probleem meer zou wegen dan de aarde [Amo09]. Het exponentiële karakter van NP-complete problemen zit hem bij DNA computing dus niet in

de tijdscomplexiteit[1] maar wel in de plaatscomplexiteit[2]. Bovendien duurde het gehele Adleman-experiment zeven dagen [Adl94], wat vooral te wijten is aan dat elke stap manueel werd uitgevoerd in een laboratorium. Snelheid was echter op dat moment geen prioriteit in het experiment.

2.3 Andere DNA computing experimenten

Er zijn tegenwoordig nog vele andere voorbeelden van (theoretische) DNA computing experimenten voor verschillende andere problemen zoals het *satisfiability* probleem (SAT), het *Maximal Clique* probleem, binair optellen... Door de jaren heen zijn er ook al verschillende pogingen ondernomen om een formeel theoretisch model voor DNA computing op te stellen. Een goede en toegankelijke beschrijving van zulke experimenten en modellen vindt u in [Amo09] of [Amo05]. Er zijn verschillende auteurs die modellen voor DNA computing ontwikkeld hebben die Turing-compleet zijn [Amo05, OR96]. Hiermee hebben ze bewezen dat elk probleem waarvoor een algoritme bestaat, ook kan worden opgelost met behulp van DNA computing. Deze modellen simuleren ofwel rechtstreeks een Turing machine ofwel simuleren ze Booleaanse circuits die equivalent zijn met Turing machines.

Een groot deel van het recentelijk onderzoek is gefocust op het uitvoeren van berekeningen gebaseerd op enkel spontane DNA reacties zonder (of met minimale) interventie van buitenaf, wat we 'DNA self-assembly' noemen. Voorbeelden hiervan vindt u in [BPEA+01, SKK+99, SSZW06]. Hoewel deze self-assembly modellen zeer aantrekkelijk zijn omdat ze volledig autonoom werken, zijn ze in de praktijk moeilijker te realiseren dan het Adleman model.

Zeer recent (juni 2011) werd er een artikel gepubliceerd in het tijdschrift *Science* waarin een DNA computing experiment beschreven is dat gebruikt maakt van een "compiler" [Rei11, QW11]. Hiervoor construeerden ze logische *AND* en *OR* poorten met DNA moleculen. Door de eenvoud van zowel deze logische operaties en als de regels van DNA basenparing, waren de wetenschappers in staat om een "compiler" te ontwikkelen die hen voor een gegeven "algoritme" aangaf welke moleculen ze moesten synthetiseren en in welke concentraties en volgorde zij ze moesten combineren. Op deze manier kunnen zij op grote schaal DNA circuits bouwen. Om aan te tonen dat hun project werkt, construeerden ze een DNA computing systeem dat de vierkantswortel van een vier-bit getal kon berekenen. Tijdens een dergelijke berekening werd er gebruik gemaakt van 130 verschillende DNA moleculen. Dit artikel toont aan dat DNA computing een zeer actueel topic is en in de toekomst nog een grote rol kan spelen.

[1] Je kan dit beschouwen als de hoeveelheid tijd die nodig is voor een berekening.
[2] Dit is de hoeveelheid ruimte/geheugen die we nodig hebben.

Hoofdstuk 3

Een formeel model voor databases in DNA

3.1 Inleiding

In dit deel van mijn proefschrift leg ik een formeel model voor databases in DNA uit. Dit formeel model werd ontwikkeld aan de Universiteit Hasselt en de transnationale Universiteit Limburg door Joris J.M. Gillis en dr. Jan Van den Bussche. De gehele hierna volgende uiteenzetting in dit deel is dan ook gebaseerd op hun paper [GV10]. Dit formele model bouwt verder op het Adleman model (zie sectie 2.2) maar dan meer gericht op databases. In dit formele model zullen we ook gebruik maken van een vast aantal 'operaties' die in een constant aantal stappen in een laboratorium kunnen worden uitgevoerd, net zoals dit het geval is in het Adleman model. Het is dus een model waarbij er wel interventies van buitenaf zullen plaatsvinden.

3.2 Motivatie

Vanuit een database-perspectief is DNA computing zeer aantrekkelijk. De kleine schaal van DNA moleculen is interessant om grote hoeveelheden data te kunnen opslaan op een kleine hoeveelheid ruimte. DNA is ook veel robuuster dan de huidige (magnetische) harde schijven. DNA moleculen zijn immers veel schokbestendiger en lijden ook minder aan dataverlies (door verlies aan magnetische lading). Bovendien kunnen we DNA moleculen ook invriezen zodat ze nog veel langer bewaren.

Bovendien zijn de sterk parallelle operaties op DNA ook zeer bruikbaar bij het uitvoeren van query's. We kunnen immers zeer veel DNA moleculen (die bepaalde data representeren) tegelijkertijd bewerken en uitfilteren. Op deze manier kunnen we zeer veel data tegelijkertijd verwerken bij het uitvoeren van een query wat een groot voordeel is voor database systemen.

Met dit formeel model onderzoekt men een mogelijke oplossing voor volgende vergelijking:

$$\frac{\text{relationele databases en relationele algebra}}{\text{algemene conventionele computing}} = \frac{?}{\text{DNA computing (Adleman model)}}$$

3.3 Een eerste, intuïtieve benadering

Voordat we aan de formele definitie van het model beginnen, wil ik het probleem eerst op een intuïtieve manier benaderen. Ik wil hierbij even stilstaan bij de belangrijkste concepten in een database-systeem en hoe we deze in DNA zouden kunnen implementeren.

Een eerste deelprobleem dat we moeten oplossen is het voorstellen van waarden die we in onze database kunnen toevoegen. Een tabel (relatie) in een database is immers een verzameling van tupels die aan elke attribuutnaam (uit het relatie-schema) een waarde toekennen. Een mogelijke aanpak zou zijn om al de letters, cijfers en symbolen van ons alfabet te coderen in DNA door er een bepaalde basen-sequentie aan te koppelen met een vaste lengte. Als we het symbool j dan gelijkstellen aan de sequentie $AGCT$ en het symbool a aan $TTCA$, dan kunnen we het woord ja weergeven met behulp van de oligo $AGCTTTCA$. Deze aanpak heeft echter twee nadelen. Een eerste nadeel wordt gevormd door het grootte aantal symbolen. We zullen hiervoor immers een groot aantal oligo's moeten bepalen die genoeg van elkaars complementen verschillen zodat ze niet hybridiseren. Een ander nadeel is het probleem bij niet-tekstuele data. Wat moeten we doen als we foto's of muziek in onze database willen bewaren? Een andere aanpak zou er uit kunnen bestaan om de bit-waarden 1 en 0 te koppelen aan een basen-sequentie en op die manier elke waarde 'binair' voor te stellen zoals dat nu in computers gebeurd. Een nadeel hierbij is dat DNA in staat is om met veel meer mogelijke 'bit-waarden' te werken en dat die mogelijkheid onbenut blijft. In ons model zullen we dus werken met dit 'bit'-model maar waarbij het aantal mogelijke bit-waarden meer dan twee is. Hierdoor kunnen we zowel al de mogelijke data representeren en blijft de representatie hiervan beperkter in omvang. Omdat de maximale lengte van een DNA molecule beperkt is, zullen we elke tupel in een aparte molecule representeren in plaats van alle tupels in één molecule achterelkaar te plaatsen. Dit maakt de simulatie van operaties zoals het cartesisch product, verschil en selectie ook veel gemakkelijker.

Een tweede probleem bevindt zich een niveau hoger, het tupel-niveau. Een relatie bestaat uit een relatie-schema en een verzameling van tupels die aan elke attribuutnaam een waarde toekennen. Om deze toekenning van waarden in DNA te kunnen weergeven, moeten we een manier vinden om per tupel de attributen en hun toegekende waarde te bewaren. We zouden in het relatie-schema een vaste volgorde van de attributen kunnen opleggen zodat ook de waarden per tupel in deze volgorde moeten voorkomen. Tijdens het uitvoeren van database query's kan de volgorde van de attributen echter wijzigen wat een probleem geeft bij deze aanpak. Om dit op te lossen kunnen we vooraf aan elke

waarde de attribuutnaam vermelden. Op die manier kunnen we altijd, ook al zijn de attributen van volgorde veranderd, elke waarde aan het juiste attribuut koppelen. Er ontstaat nu echter een nieuw probleem. Hoe kunnen we nu bepalen waar de attribuutnaam stopt en het waardeveld begint? We zullen een soort van 'scheidingstekens' moeten introduceren (die we *tags* zullen noemen) om aan te geven waar de attribuutnaam begint, waar het waardeveld begint en waar dit waardeveld eindigt binnen het DNA molecule. Later zullen deze *tags* ook handig blijken bij het manipuleren van de DNA moleculen.

We gaan nu opnieuw een niveau hoger, het relatie-niveau. Zoals ik reeds vermeld heb, bestaat een relatie uit een verzameling van tupels. We kunnen ons nu afvragen of we ofwel voor elke relatie een aparte proefbuis[1] moeten voorzien, ofwel alle relaties in één proefbuis samenvoegen. Indien we ze allemaal zouden samenvoegen, moeten we elk tupel ook voorzien van zijn relatienaam waardoor we weer extra tags zouden moeten introduceren. Bovendien zouden we dan veel extra redundante informatie bijhouden. We zullen er dus voor kiezen om voor elke relatie een aparte proefbuis te voorzien.

Een laatste (groot) probleem is het schrijven en uitvoeren van query's. Als eerste poging zouden we kunnen stellen dat we standaard SQL gebruiken zoals in de huidige databases en dit proberen te simuleren in DNA. Hoewel dit zeker mogelijk is (SQL en DNA computing zijn immers beiden Turing-compleet), is het toch niet zo'n geschikte query-taal voor DNA. De operaties in SQL leunen minder aan bij de 'natuurlijke' operaties op DNA en de implementatie ervan zou het model zeer complex maken. SQL zou ook minder herbruikbaar zijn als losstaand DNA computing model. We zullen dus een query-taal moeten ontwikkelen die dichter aanleunt bij de operaties op DNA zodat deze ook bruikbaar blijft voor andere DNA computing modellen. Zo een query-taal zou ook een oplossing bieden voor de vergelijking uit sectie 3.2 wat één van onze hoofddoelstellingen is [GV10]. We moeten slechts een beperkt aantal operaties ontwikkelen omdat query-talen typisch met een beperkte set operaties werken die beperkt zijn in computationele complexiteit. Door dit vast aantal operaties en omdat we de complexiteit willen beperken, zullen we geen gebruik maken van 'DNA self-assembly' operaties maar van operaties zoals in het Adleman model. Dit soort operaties bieden ons ook veel meer controle en optimalisatie mogelijkheden.

Het lijkt dus zeer goed mogelijk om een database-model voor DNA computing te ontwikkelen. Er zijn echter nog veel zaken die we formeel moeten uitklaren en dit gebeurt dan ook in de rest van dit hoofdstuk.

3.4 Het Sticker Complex datamodel

Het sticker complex datamodel is een familie van datastructuren en bijhorende operaties waarop bepaalde beperkingen gelden. Sticker complexen zijn datastructuren die een abstractie zijn van complexen van DNA-strengen. De beperkingen die in dit datamodel worden gehanteerd zijn nodig om onrealistische of onhandelbare DNA structuren te voorkomen. We willen ervoor zorgen dat we

[1] Met "proefbuis" bedoel ik de container waarin de oplossing met DNA moleculen zich bevindt. Dit hoeft niet persé een échte proefbuis te zijn.

enkel DNA-strengen moeten beschouwen die niet op zichzelf kunnen hybridiseren (door bijvoorbeeld dubbel te plooien). Dergelijke fenomenen zouden het model te complex maken. Ondanks deze beperkingen is het sticker complex datamodel samen met zijn operaties toch nog krachtig genoeg om de relationele algebra te kunnen simuleren [GV10]. Het sticker complex datamodel kan dus binnen DNA computing als tegenhanger voor de relationele algebra dienen. Het is echter goed mogelijk dat er binnen DNA computing nog andere toepassingen voor het sticker complex datamodel zijn, iets wat verder onderzocht zal moeten worden. In deze sectie leg ik de formele definitie van het sticker complex datamodel uit.

3.4.1 Het gebruikte alfabet

Voor de formele definitie beschouwen we de volgende disjuncte en eindige alfabetten:

- Λ Het alfabet van atomische waarden symbolen (bv. cijfers en letters)
- Ω Het alfabet van attribuutnamen
- Θ Het alfabet $\{\#_1, \#_2, \#_3, \#_4, \#_5, \#_6, \#_7, \#_8, \#_9\}$ van *tags*

We beschouwen bovendien ook het alfabet $\Sigma = \Lambda \cup \Omega \cup \Theta$, dat we het *positieve alfabet* noemen. Aan elk symbool in elk alfabet is een uniek (enkelvoudig) DNA-streng gekoppeld die dat symbool 'codeert'. Al deze DNA-strengen zijn van gelijke lengte. Je zou dit kunnen vergelijken met de *codons* die aminozuren representeren (zie sectie 1.1). Deze DNA-streng representaties zullen per database systeem verschillen en zijn hier dus niet definitief bepaald. U moet zich er gewoon van bewust zijn dat elk symbool eigenlijk staat voor een bepaalde sequentie van basen aan een DNA-streng. Het is echter belangrijk de sequenties zo te kiezen dat ze genoeg van elkaar verschillen om gedeeltelijke hybridisaties te voorkomen. We willen immers niet dat twee symbolen slechts met enkele (of de helft) van hun basen aan elkaar hybridiseren. Voor elk symbool a bestaat er, wegens de *Watson-Crick base pairing*, ook een complementair symbool $\overline{a}$ waaraan een equivalente complementaire DNA-streng gekoppeld is. Het alfabet van alle complementaire symbolen noteren we met $\overline{\Sigma}$ en wordt dus gegeven door $\overline{\Sigma} = \{\overline{a} \mid a \in \Sigma\}$. We noemen dit alfabet het *negatieve alfabet* en het is disjunct aan Σ.

3.4.2 Pre-complexen

We definiëren eerst *pre-complexen* die de algemene structuur van sticker complexen hebben. Hierna kunnen we sticker complexen definiëren in termen van pre-complexen waarop bepaalde beperkingen gelden. Een pre-complex is een eindige, gelabelde, gerichte graaf waarbij de bogen basen-sequenties voorstellen in een DNA-streng en de knopen de eindpunten van deze (deel-)strengen representeren. De labels van de bogen komen uit het alfabet $\Sigma \cup \overline{\Sigma}$. Een pre-complex beschikt ook nog over een extra *matching*-functie μ die aangeeft welke bogen aan elkaar gehybridiseerd zijn (zie sectie 1.2) op basis van hun labels (en de DNA sequenties die de labels representeren). Tenslotte heeft een pre-complex ook nog twee extra predikaten: *immob* en *blocked*. Het predikaat *immob* geeft

aan welke bogen (basen-sequenties) geïmmobiliseerd zijn en dus niet vrij kunnen rondzweven. Dit wordt gerealiseerd zoals beschreven is in sectie 1.6. Bovendien kunnen de bogen die (onrechtstreeks) gekoppeld zijn aan een geïmmobiliseerde boog ook niet vrij rondzweven. Het predikaat *blocked* geeft aan welke bogen "geblokkeerd" zijn en dus niet kunnen deelnemen aan hybridisatie. Het blokkeren van een bepaalde DNA (deel-)streng wordt gerealiseerd door het toevoegen van een oligo die complementair is aan de te blokkeren DNA-streng of via polymerase. Met behulp van dit predikaat kunnen we het aantal locaties waar hybridisatie kan plaatsvinden beperken. Formeel is een pre-complex dus een 6-tupel $\big(V,\ E,\ \lambda,\ \mu,\ \mathit{immob},\ \mathit{blocked}\big)$ waarbij:

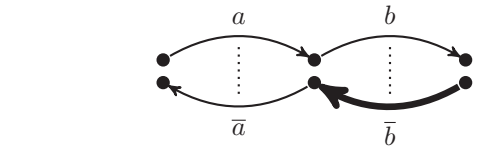
1. V een eindige verzameling knopen is;
2. $E = \{(v_1, v_2) \mid ((v_1, v_2) \in V \times V) \wedge (v_1 \neq v_2)\}$, een eindige verzameling van gerichte bogen;
3. $\lambda : E \longrightarrow \Sigma \cup \overline{\Sigma}$ een totale functie is die een label toekent aan elke boog;
4. $\mu = \{\{e_1, e_2\} \mid (\{e_1, e_2\} \in E \times E) \wedge (e_1 \neq e_2)\}$, een partiële matching van de bogen is waarbij elke boog e_i hoogstens één maal voorkomt in een paar in μ;
5. $\mathit{immob} \subseteq E$;
6. $\mathit{blocked} \subseteq E$.

We kunnen een pre-complex ook grafisch weergeven. Ik hanteer hierbij volgende conventies: V en E worden zoals klassieke grafen weergegeven waarbij er labels bij de corresponderende bogen staan zoals aangegeven wordt door λ; tussen elk koppel $\{e_1, e_2\}$ van μ is een stippellijn getekend om de partiële matching aan te duiden; bogen die in *immob* zitten worden met een dikke pijl getekend en bogen in *blocked* met een gestippelde pijl. Op deze manier kan men al de informatie van een pre-complex grafisch weergeven. Een voorbeeld hiervan ziet u in figuur 3.1. Als we in het pre-complex van figuur 3.1(a) veronderstellen dat label a overeenkomt met de oligo **ACGTGCATCA** en label b met de oligo $GTTCACCGAA$, dan representeert dit pre-complex de DNA molecule:

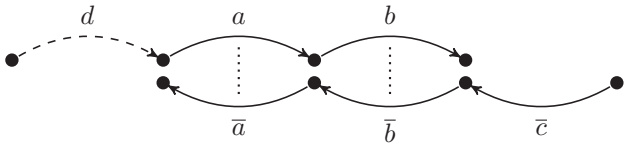
ACGTGCATCA*GTTCACCGAA*
TGCACGTAGT*CAAGTGGCTT*

waarbij $CAAGTGGCTT$ gebonden is aan een bindingsoppervlak of een magnetische korrel. Merk ook op hoe de richting van de bogen gebruikt wordt om de oriëntatie van de oligo aan te geven. De pijl gaat altijd van het 5' uiteinde naar het 3' uiteinde.

In dit model spreken we ook over een "streng" en een "component" van een pre-complex waarbij we deze begrippen als volgt definiëren: Zij P een pre-complex gedefinieerd zoals hierboven met $\big(V,\ E,\ \lambda,\ \mu,\ \mathit{immob},\ \mathit{blocked}\big)$. Een *streng* s van P is eenvoudigweg een geconnecteerde deelgraaf van de gerichte graaf (V, E). Merk op dat we μ dus buiten beschouwing laten. Bovendien is een streng van een pre-complex zelf ook een pre-complex. We kunnen s dan ook definiëren als $\big(V',\ E',\ \lambda|_{E'},\ \emptyset,\ \mathit{immob} \cap E',\ \mathit{blocked} \cap E'\big)$ met $\big(V',\ E'\big)$ de geconnecteerde deelgraaf. Intuïtief komt een *streng* overeen met een (enkelvoudig) DNA-streng dat gevormd wordt door opeenvolgende bogen.



(a) Een pre-complex met geïmmobiliseerde boog



(b) Een pre-complex met een geblokkeerde boog

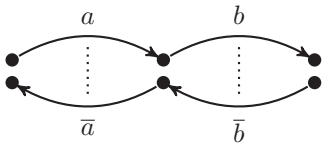
Figuur 3.1: Twee voorbeeld pre-complexen

We zeggen dat twee strengen s_1 en s_2 van pre-complex P *gekoppeld* zijn als er een boog e_1 in s_1 en een boog e_2 in s_2 bestaat zodat $\{e_1, e_2\} \in \mu$. De twee strengen s_1 en s_2 behoren bovendien tot dezelfde *component* als ze ofwel aan elkaar gekoppeld zijn, ofwel als s_1 gekoppeld is aan een derde streng s_3 die tot de component van s_2 behoort. Alle strengen van een component zijn dus rechtstreeks of onrechtstreeks aan elkaar gekoppeld. Een *component* van een pre-complex is dus een deel-pre-complex gevormd door een zo groot mogelijke verzameling van strengen die (onrechtstreeks) gekoppeld zijn.

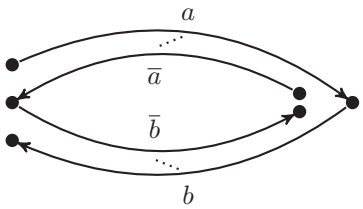
3.4.3 Sticker complexen

Een *sticker complex* is een pre-complex dat voldoet aan volgende acht beperkingen:

1. Er zijn geen losstaande knopen. Elke knoop komt dus in minstens één boog voor.
2. Elke knoop heeft hoogstens één inkomende en hoogstens één uitgaande boog. Elke streng heeft dus de vorm van een keten of een cyclus.
3. De labels op een keten zijn "homogeen". Hiermee wordt bedoeld dat alle bogen van die keten ofwel labels uit Σ hebben, ofwel labels uit $\overline{\Sigma}$. De corresponderende strengen worden respectievelijk positieve strengen of negatieve strengen genoemd.
4. Negatieve strengen zijn sterk beperkt: elke negatieve streng mag bestaan uit slechts één of twee bogen. Deze korte negatieve strengen worden ook wel *stickers* genoemd en worden gebruikt voor de herkenning en splitsing van strengen die de eigenlijke data bevatten.
5. De matching-functie μ mag enkel nog worden toegepast op bogen met complementaire labels. Deze functie geeft dus enkel nog aan op welke plaatsen de stickers gehybridiseerd zijn aan de positieve strengen.



(a) Een sticker complex met een sticker $\overline{b}\overline{a}$



(b) De sticker werd vervangen door $\overline{a}\overline{b}$

Figuur 3.2: Oriëntatie invariantie bij stickers

6. Een boog kan geïmmobiliseerd worden enkel als het de enige boog is van een negatieve streng.
7. De bogen die voorkomen in *blocked* mogen niet voorkomen in μ.
8. Elke component mag hoogstens één geïmmobiliseerde boog bevatten.

Deze beperkingen hebben als doel om enerzijds het klassiek DNA gedrag af te dwingen (zie regels 1, 2, 5 en 7) en anderzijds de werking van dit model te vereenvoudigen en mogelijke neveneffecten te voorkomen (zie regels 3, 4, 6 en 8). Later zullen we merken dat ondanks het feit dat deze regels het model sterk beperken en vereenvoudigen, het toch nog krachtig genoeg is om aan database-manipulatie te doen. Merk op dat de pre-complexen in figuur 3.1 beiden geen sticker complexen zijn. Het pre-complex in figuur 3.1(a) heeft immers een geïmmboliseerde boog die deel is van een sticker van lengte twee wat niet mag wegens regel 6. Het pre-complex in figuur 3.1(b) heeft een negatieve streng van lengte drie wat niet mag wegens regel 4.

De aandachtige lezer heeft misschien opgemerkt dat we bij de matching-functie μ geen vereiste leggen op de tegengestelde oriëntatie van de bogen, wat vereist wordt door de werking van DNA (de ene streng moet een 5'-3' oriëntatie hebben en de andere een 3'-5' oriëntatie). Dit komt omdat we stickers van lengte één triviaal in een andere richting kunnen plaatsen en stickers van lengte twee altijd kunnen plooien zodat we de gewenste oriëntatie bekomen. Een voorbeeld hiervan vindt u in figuur 3.2. Deze redenering klopt echter niet altijd en in sectie 5.1 leg ik uit waarom.

3.4.4 Redundantie in complexen

In de praktijk zal een proefbuis verschillende kopieën van eenzelfde DNA-streng bevatten, net zoals dit in andere DNA computing experimenten het geval is (zie sectie 2). In het sticker complex datamodel zal elke component van een sticker

complex dus ook voor mogelijk meerdere voorkomens van het gerepresenteerde DNA complex staan. Dit fenomeen wordt geformaliseerd met de noties *isomorfisme*, *onderschikking*, *equivalentie*, *redundante extensie* en *minimaliteit*.

Isomorfisme: Een sticker complex $C_1 = \big(V_1,\ E_1,\ \lambda_1,\ \mu_1,\ immob_1,\ blocked_1\big)$ is *isomorf* aan een sticker complex $C_2 = \big(V_2,\ E_2,\ \lambda_2,\ \mu_2,\ immob_2,\ blocked_2\big)$ als en slechts als er een bijectie $f : V_1 \longrightarrow V_2$ bestaat zodat:

$$\begin{gathered}
(v_1, v_2) \in E_1 \Leftrightarrow (f(v_1), f(v_2)) \in E_2 \\
\lambda_1((v_1, v_2)) = \sigma \Leftrightarrow \lambda_2((f(v_1), f(v_2))) = \sigma \\
\{(v_1, v_2), (u_1, u_2)\} \in \mu_1 \Leftrightarrow \{(f(v_1), f(v_2)), (f(u_1), f(u_2))\} \in \mu_2 \\
(v_1, v_2) \in immob_1 \Leftrightarrow (f(v_1), f(v_2)) \in immob_2 \\
(v_1, v_2) \in blocked_1 \Leftrightarrow (f(v_1), f(v_2)) \in blocked_2
\end{gathered}$$

Intuïtief komt isomorfisme neer op twee sticker complexen die gelijk zijn van vorm (waarbij ook de labels, μ, *blocked* en *immob* in acht worden genomen) maar die beiden een andere verzameling knopen gebruiken.

Onderschikking: Een sticker complex C' noemen we *ondergeschikt* aan een sticker complex C als er voor elke component D in C een isomorfe component D' in C' bestaat (Herinner je dat een component van een sticker complex zelf ook een sticker complex is). We kunnen hier echter niet spreken van een bijectie omdat het aantal componenten in C' kan verschillen van het aantal componenten in C.

Equivalentie: Twee complexen C_1 en C_2 zijn *equivalent* als en slechts als ze ondergeschikt zijn aan elkaar.

Redundante extensie: Wanneer een sticker complex C' equivalent is aan een ander sticker complex C en aan een extensie van C, dan noemen we C' een *redundante extensie* van C. C' is dus een *redundante extensie* van C als C' equivalent is aan C en het aantal componenten in C' groter is dan het aantal componenten in C.

Minimaliteit: Een component D van een sticker complex C is redundant als er een andere component in C bestaat die isomorf is aan D. Als we een redundante component van C verwijderen, bekomen we een sticker complex dat nog steeds equivalent is aan C. Een sticker complex dat geen redundante componenten bevat wordt *minimaal* genoemd. Elk sticker complex C heeft bovendien een uniek (op isomorfisme na) *minimaal sticker complex* C' dat equivalent is aan C. We noemen C' *de minimalisatie* van C.

3.5 Data representatie

De volgende vraag die we ons moeten stellen is hoe we sticker complexen kunnen gebruiken voor het representeren van gestructureerde data. De symbolen van de alfabetten Λ, Ω en Θ zullen hiervoor elk op een andere manier gebruikt worden. Attributen (Ω) zullen we gebruiken om de structuur van de data aan te geven en tags (Θ) zullen dienen als scheidingstekens en hulp-markeringen bij datamanipulatie. Atomische waardesymbolen (Λ) dienen voor het representeren van de eigenlijke data. Maar omdat Λ een eindig (en meestal beperkt) alfabet is, zullen we gebruik moeten maken van *strings* van atomische symbolen voor het representeren van ingevoerde gegevens. Men kan dit vergelijken met de manier waarop bits bij de huidige computers gebruikt worden voor het vormen van integers en karakters. Analoog aan de woord-lengte van een conventionele CPU, die meestal 32 of 64 bits bedraagt, veronderstellen we in het sticker complex datamodel een bepaalde *dimensie* ℓ, een natuurlijk getal, zodat elke data-blok gecodeerd wordt door een ℓ-vector van atomische data symbolen. Om de vergelijking met een conventionele CPU te vervolledigen, kunnen we stellen dat bij een 32-bit CPU het alfabet Λ gelijk is aan $\{1,\ 0\}$ en deze een dimensie van 32 heeft.

Formeel stellen we dat een sticker complex C *dimensie* ℓ *heeft* als en slechts als elke boog e die gelabeld is met een (positief) atomisch symbool, voorkomt in een bepaalde sequentie $\big(e_0,\ e_1,\ \ldots,\ e_\ell,\ e_{\ell+1}\big)$ van $\ell+2$ opeenvolgende bogen. Hierbij is e_0 gelabeld met $\#_3$, $e_{\ell+1}$ gelabeld met $\#_4$ en elke e_i met $i = 1, ..., \ell$ is gelabeld met een positief atomische symbool. Dus, $e \in \{e_1, e_2, ..., e_\ell\}$. Een sticker complex met dimensie ℓ wordt ook wel een ℓ-complex genoemd. Zo een sequentie van ℓ opeenvolgende bogen $\big(e_0,\ e_1,\ ...,\ e_{\ell+1},\ e_\ell\big)$ noemen we ook wel een ℓ-vector. Het begin van elke 'data-woord' zal dus worden aangekondigd met een $\#_3$ tag en eindigen met een $\#_4$ tag.

Omdat het doel van het sticker complex datamodel het implementeren van een database in DNA is, willen we relaties (zoals we die kennen vanuit de relationele algebra) representeren met behulp van complexen. Data elementen slaan we op als ℓ-vector strengen van atomische waarde symbolen. Een tupel t van verschillende data elementen heeft dimensie ℓ als alle data elementen die voorkomen in t in een ℓ-vector streng zitten. Zij t nu een tupel van dimensie ℓ over een relatie schema R. We veronderstellen een vaste volgorde op de attributen $A_1, A_2, ..., A_n$ (met $n \geq 1$) van R. We representeren t dan door het volgende ℓ-complex:

$$complex(t) = \#_2 A_1 \#_3 t(A_1) \#_4 \#_2 A_2 \#_3 t(A_2) \#_4 \ldots \#_2 A_n \#_3 t(A_n) \#_4$$

Elke attribuutnaam zal dus voorafgegaan worden door een $\#_2$ tag en elk waardeveld zal, zoals hoger reeds aangegeven, omgeven worden door $\#_3$ en $\#_4$. Een relatie r (een verzameling van tupels) van dimensie ℓ wordt dan gerepresenteerd door het ℓ-complex:

$$complex(r) = \bigcup \{complex(t) \mid t \in r\}$$

Een database instantie I (een verzameling van verschillende relaties) van dimensie ℓ kan op zijn beurt ook worden gerepresenteerd door aan elke relatie het overeenkomstige sticker complex te koppelen. We definiëren $complex(I)$, met I

een database instantie (met database schema D) die aan elke relatienaam een relatie instantie koppelt, dan als:

$$complex(I) = \{(R, complex(I(R)) \mid R \in D\}$$

3.6 Operaties op complexen

In deze sectie bespreek ik de formele definities van de verschillende operaties op sticker complexen. Deze operaties komen overeen met de standaard operaties die men in DNA computing gebruikt, behalve misschien de operatie 'difference'. Een bijkomend voordeel is dat sticker complexen zo zijn gedefinieerd dat deze operaties altijd resulteren in een sticker complex.

Als een algemene regel veronderstellen we dat we na elke operatie een finale minimalisatie stap uitvoeren op het resultaat van die operatie om zo een mathematisch deterministische operatie te bekomen. De minimalisatie bestaat uit het verwijderen van alle redundante componenten uit een sticker complex. Deze finale minimalisatie stap vermeld ik niet expliciet bij elke formele definitie om de leesbaarheid te behouden. Bovendien wordt er ook verondersteld dat het resultaat van elke operatie tot op isomorfisme na uniek gedefinieerd is. We beschouwen nu de formele definitie van volgende operaties op complexen: union, difference, hybridize, ligate, flush, split, blocking en cleanup.

3.6.1 Union *(unie)*

Zij $C_1 = \big(V_1,\ E_1,\ \lambda,\ \mu_1,\ immob_1,\ blocked_1\big)$ en $C_2 = \big(V_2,\ E_2,\ \lambda,\ \mu_2,\ immob_2,\ blocked_2\big)$ twee sticker complexen. Zonder verlies van algemeenheid veronderstellen we dat V_1 en V_2 disjunct zijn. Als dit niet het geval is dan hernoemen we de gemeenschappelijke elementen in V_2. De unie van C_1 en C_2 is dan gedefinieerd als:

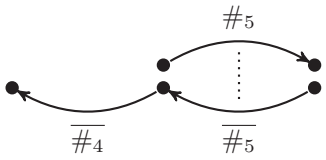
$$\begin{aligned} C_1 \cup C_2 = &(V_1 \cup V_2,\ E_1 \cup E_2,\ \lambda_1 \cup \lambda_2,\ \mu_1 \cup \mu_2, \\ &immob_1 \cup immob_2,\ blocked_1 \cup blocked_2) \end{aligned}$$

De werkelijke implementatie in DNA zal voor deze operatie eenvoudigweg bestaan uit het samenvoegen van de twee oplossingen waarin de twee complexen zich (gescheiden) bevinden.

3.6.2 Difference *(verschil)*

Zij $C_1 = \big(V_1,\ E_1,\ \lambda,\ \mu_1,\ immob_1,\ blocked_1\big)$ en $C_2 = \big(V_2,\ E_2,\ \lambda,\ \mu_2,\ immob_2,\ blocked_2\big)$ twee sticker complexen die voldoen aan volgende voorwaarden:

1. $\mu_1 = immob_1 = blocked_1 = \emptyset$ en $\mu_2 = immob_2 = blocked_2 = \emptyset$, al de componenten van C_1 en C_2 zijn dus enkelvoudige strengen;



Figuur 3.3: Het sticker complex `rightboot` voor de operatie difference

2. Alle strengen van C_1 en C_2 zijn positief, niet-circulair en hebben allemaal dezelfde lengte[2];
3. Elke streng van C_2 eindigt met $\#_4$ en bevat niet $\#_5$.

Het verschil tussen C_1 en C_2, genoteerd met $C_1 - C_2$, wordt nu gegeven door de unie van al de strengen in C_1 die geen isomorfe tegenhanger in C_2 hebben. Als C_1 of C_2 niet voldoen aan bovenstaande voorwaarden, dan is het verschil tussen hen ongedefinieerd.

De voorwaarden die hierboven worden opgelegd, zijn nodig om een effectieve implementatie in DNA mogelijk te maken. Deze implementatie gaat als volgt: We veronderstellen dat C_1 zich in een proefbuis p_1 bevindt en dat C_2 zich in een proefbuis p_2 bevindt. Omdat alle strengen van C_2 in p_2 eindigen met $\#_4$ (zie voorwaarde 3) kunnen we er makkelijk de tag $\#_5$ aan vastkoppelen. Dit doen we door het complex `rightboot` in figuur 3.3 toe te voegen waarna het vrije $\#_4$ uiteinde zal hybridiseren aan het complementaire $\overline{\#_4}$ uiteinde van `rightboot`. Hierna passen we ligase en vervolgens denaturatie toe wat het gewenste resultaat geeft. Na het aankoppelen van $\#_5$, voegen we aan p_2 een overvloed van geïmmobiliseerde korte primers $\overline{\#_5}$ toe. Gebruikmakend van polymerase en denaturatie (zie sectie 1.3 en 1.2) kunnen we de complementen van alle strengen in p_2 bekomen die bovendien nog steeds geïmmobiliseerd zijn zodat we ze gemakkelijk kunnen uitfilteren. Met behulp van deze geïmmobiliseerde complementaire strengen kunnen we nu uit p_1 alle strengen filteren die voorkomen in p_2. Deze strengen zullen immers hybridiseren aan hun complement. Omdat alle strengen dezelfde lengte hebben, kunnen we bovendien partiële hybridisatie voorkomen door gebruik te maken van een zeer nauwkeurig bepaald smeltpunt dat gebaseerd is op de precieze lengte en de AT-GC verhouding van de strengen.

3.6.3 Hybridize *(hybridiseren)*

Zij $C_1 = \big(V_1,\ E_1,\ \lambda_1,\ \mu_1,\ immob_1,\ blocked_1\big)$ en $C_2 = \big(V_2,\ E_2,\ \lambda_2,\ \mu_2,\ immob_2,\ blocked_2\big)$ twee sticker complexen. We noemen C_2 een *hybridisatie extensie* van C_1 als $V_1 = V_2$, $E_1 = E_2$, $\lambda_1 = \lambda_2$, $immob_1 = immob_2$, $blocked_1 = blocked_2$ en $\mu_1 \subseteq \mu_2$. De verzameling μ_2 duidt met andere woorden extra bogen aan die ook gehybridiseerd zijn. Bovendien moet een hybridisatie extensie voldoen aan al de voorwaarden uit de definitie van een sticker complex (zie sectie 3.4.3). C_2 heeft bovendien een *maximale matching* als de enige hybridisatie extensie van C_2, C_2 zelf is.

[2]Merk op dat de strengen zowel hetzelfde aantal bogen als basen hebben. De labels op de bogen bestaan immers uit een gelijk aantal basen.

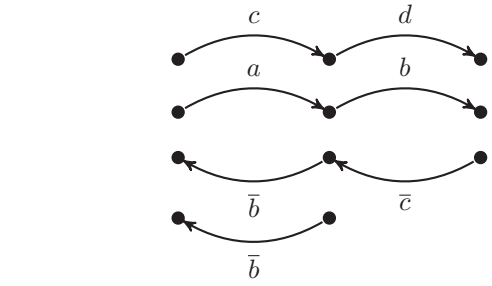
Deze notie van hybridisatie extensie is echter niet voldoende. Zoals aangegeven in de definitie van het sticker complex (zie sectie 3.4.4) staat elke component in een sticker complex voor meerdere voorkomens van het gerepresenteerde DNA complex. Deze extra (redundante) DNA complexen zullen ook deelnemen aan de hybridisatie en mogelijk nieuwe, van elkaar verschillende DNA complexen vormen. Deze nieuwe DNA complexen moeten een equivalente representatie krijgen in het sticker complex datamodel. Hiervoor zullen we extra bogen en knopen moeten toevoegen aan ons sticker complex. Om dit gedrag te formaliseren, noemen we C_2 een *"multiplying hybridization extension" (MHE)* (in het Nederlands: een vermenigvuldigende hybridisatie extensie) van C_1 als C_2 een hybridisatie extensie met maximale matching is van een redundante extensie C_1' (die extra componenten bevat) van C_1 (zie ook sectie 3.4.4). Voor het construeren van een MHE van een sticker complex C_1 mogen we dus extra isomorfe kopieën van bestaande componenten in C_1 toevoegen zodat ze eveneens kunnen deelnemen aan de hybridisatie.

Een component D van een MHE C^i noemen we *onvoltooid* als we uit C^i een andere MHE C^{i+1} kunnen construeren waarbij D gebonden is aan een grotere component. Een MHE is *verzadigd* als deze geen onvoltooide componenten bevat (zie figuur 3.4). Tenslotte stellen we formeel dat een sticker complex C *recursie-vrije hybridisatie* heeft als er slechts een eindig aantal MHE's van C bestaan. In dat geval definiëren we `hybridize`(C) als de unie van al de verzadigde MHE's van C. In het andere geval, als C geen recursie-vrije hybridisatie heeft, is `hybridize`(C) ongedefinieerd.

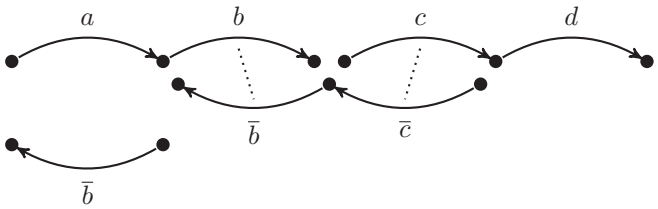
Een moeilijkheid bij hybridisatie is dat deze kan uitlopen in een ongecontroleerde kettingreactie. Dit is ongewenst in het sticker-complex datamodel omdat we immers streven naar een eerste-orde/recursie-vrij model voor DNA computing en niet naar een model op basis van 'self-assembly DNA computations'. We willen dus de situaties waarin er oneindig veel niet-equivalente MHE's bestaan, kunnen uitsluiten. Deze situaties kunnen zich zeer zeker voordoen. Veronderstel bijvoorbeeld een sticker complex C dat bestaat uit twee niet-circulaire strengen die de woorden ab en $\overline{a}\overline{b}$ vormen. Omdat we, wegens de definitie van sticker complexen, over verschillende instanties van ab en $\overline{a}\overline{b}$ beschikken, kunnen we oneindig veel niet-equivalente MHE van C met een willekeurige lengte construeren. Een voorbeeld met drie instanties voor ab en voor $\overline{a}\overline{b}$ is gegeven in figuur 3.5. Het sticker complex datamodel (zoals beschreven in [GV10]) legt geen restricties of voorwaarden op bij het gebruik van hybridisatie om recursie-vrije hybridisatie te garanderen. Het is dus de verantwoordelijkheid van de gebruiker van deze operatie om ervoor te zorgen dat `hybridize` enkel wordt toegepast op complexen met recursie-vrije hybridisatie, wil hij een zinvol resultaat bekomen.

Recent toonde men echter aan dat het beslisbaar is (in polynomiale tijd) dat voor een gegeven sticker complex de hybridisatie recursie-vrij (eindig) is [BGV11]. Intuïtief kan men dit als volgt bekijken[3]. Bij een hybridisatie vertrekken we altijd van een eindig aantal componenten, die we echter onbeperkt kunnen kopiëren. Zij $C = \big(V,\ E,\ \lambda,\ \mu,\ \mathit{immob},\ \mathit{blocked}\big)$ een sticker complex. We beschouwen de verzameling μ' van alle koppels $\{e, e'\}$, met e en $e' \in E$, die niet in μ zitten maar wel in μ zouden kunnen zitten na een hybridisatie en waarbij de componenten van zowel e als e' geen geïmmobiliseerde boog bevatten. Met

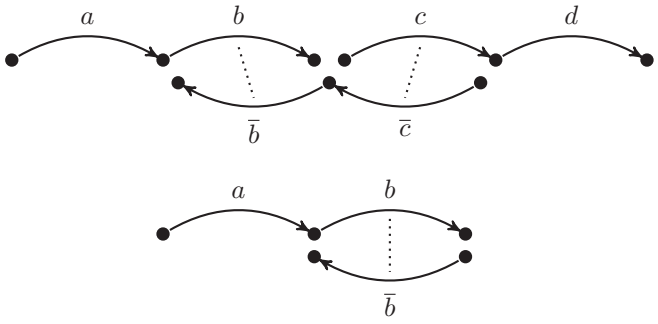
[3]Mijn aanpak verschilt van deze in de paper, maar het principe komt op hetzelfde neer.



(a) Een sticker complex C

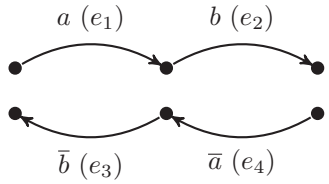


(b) Een niet verzadigde hybridisatie extensie van C

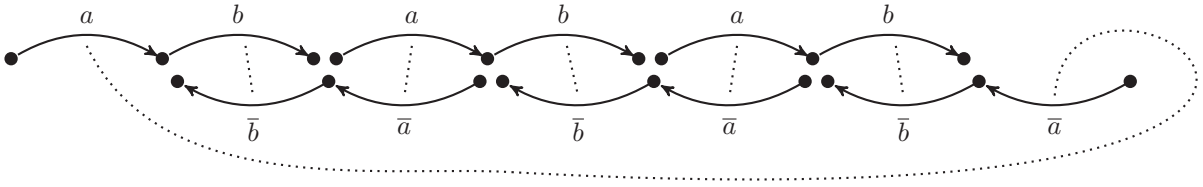


(c) MHE van C die wel verzadigd is

Figuur 3.4: Hybridisatie extensies bij sticker-complexen



(a) Een sticker complex C



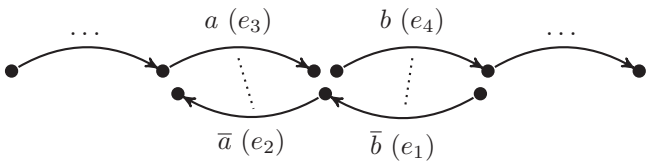
(b) Een niet-verzadigde hybridisatie extensie van C

Figuur 3.5: Sticker complex met oneindig aantal niet-equivalente MHE's

andere woorden, zowel e als e' zijn open en hun labels zijn elkaars complement. We beschouwen alleen de componenten zonder geïmmobiliseerde boog omdat de componenten met geïmmobiliseerde boog niet met elkaar kunnen hybridiseren en dus nooit oneindig lange ketens vormen. Een hybridisatie is nu *"niet-recursie vrij"* als er *"recursie"* mogelijk is op een van de koppels in μ'.

We bekijken hiervoor de hybridisatie bij een open boog e_1 in een component c_1. Omdat e_1 open is, kunnen we uit μ' een koppel $\{e_1, e_2\}$ kiezen. Zij c_2 de component waarin e_2 zich bevindt (merk op dat c_1 en c_2 dezelfde component kunnen zijn). Binnen c_2 kiezen we opnieuw een open boog om de hybridisatie verder te zetten en we noemen deze boog e'_2. Bovendien moet $e'_2 \neq e_2$ omdat we e_2 binnen de component c_2 zojuist gebruikt hebben voor de hybridisatie en dus niet meer open is. Omdat e'_2 open is, kunnen we uit μ' opnieuw een koppel $\{e'_2, e_3\}$ kiezen en heel de procedure herhalen voor e_3. Op deze manier bekomen we uiteindelijk een sequentie van koppels $\{e_1, e_2\}, \{e'_2, e_3\}, \{e'_3, e_4\}, ...$ uit μ'. Indien we nu voor een vrij te kiezen e_1 een hybridisatiecyclus $\{e_1, e_2\}, \{e'_2, e_3\}, \{e'_3, e_4\}, ..., \{e'_i, e'_1\}$, $\{e_1, e_2\}$ kunnen construeren, dan is de hybridisatie *"recursief"* en niet eindig. We kunnen dan immers vertrekkend van het laatste koppel opnieuw heel de sequentie herhalen en dit een oneindig aantal keer. Het koppel $\{e_1, e_2\}$ kan inderdaad opnieuw voorkomen omdat we werken met (oneindig veel) kopieën van de componenten c_1 en c_2. Indien we voor alle mogelijke open bogen niet in staat zijn om zo een hybridisatiecyclus te construeren, dan is de hybridisatie *"niet-recursief"* en dus wel eindig. Als we het voorbeeld in figuur 3.5(a) bekijken dan kunnen we de sequentie $\{e_2, e_3\}, \{e_4, e_1\}, \{e_2, e_3\}$ construeren waardoor we zien dat de hybridisatie niet eindig is. De bovenstaande werkwijze is bovendien gemakkelijk om te zetten naar een polynomiaal algoritme.

De DNA implementatie van de `hybridize` operatie gaat vanzelfsprekend via



Figuur 3.6: Een voorbeeld van een gat voor de `ligate` operatie

het natuurlijk proces hybridisatie (zie sectie 1.2). Het activeren (of voorkomen) van hybridisatie kunnen we realiseren door het verlagen (of verhogen) van de temperatuur. Een bijzonderheid hierbij is het geval van geïmmobiliseerde componenten. Deze mogen immers maar één geïmmobiliseerde boog bevatten. We veronderstellen echter dat er een bepaalde minimale afstand is tussen de verschillende geïmmobiliseerde componenten zodat het waarschijnlijk is dat de overgrote meerderheid van de hybridisatie reacties zal gebeuren tussen vrij rondzwevende strengen en tussen vrij rondzwevende en geïmmobiliseerde strengen. Dit is nodig omdat we anders niet meer voldoen aan de definitie van een sticker complex

3.6.4 Ligate *(ligeren)*

De `ligate`-operatie verbindt twee strengen die bij elkaar gehouden worden door een sticker. We definiëren een *gat* als een verzameling van vier bogen $\{e_1, e_2, e_3, e_4\}$ zodat $\{e_1, e_4\} \in \mu$ en $\{e_2, e_3\} \in \mu$. Hierbij zijn e_1 en e_2 (in die volgorde) opeenvolgende bogen op een negatieve streng, is e_3 de laatste boog van een positieve streng en is e_4 de eerste boog van zijn positieve streng. De situatie wordt geschetst in figuur 3.6. We kunnen dit *gat* nu *opvullen* door de eind-knoop van e_3 te versmelten met de start-knoop van e_4. We definiëren `ligate`(C) als het sticker complex bekomen uit C door al de gaten op te vullen.

De implementatie van deze operatie in DNA gebeurt door het toevoegen van het enzym ligase waarvan de werking beschreven staat in sectie 1.4. Als we in het voorbeeld van figuur 3.6 het label a gelijkstellen aan de basensequentie TCA (en $\overline{a}$ dus aan AGT) en het label b aan TGC (en $\overline{b}$ dus aan ACG), dan bekomen we dezelfde situatie als in figuur 1.3 op pagina 8 waar de ligase wordt uitgevoerd.

3.6.5 Flush *(afspoelen)*

De operatie `flush`(C) geeft als resultaat een sticker complex C' bekomen uit C door al de componenten die geen geïmmobiliseerde boog bevatten, te verwijderen. Zoals eerder besproken zijn er twee methoden om DNA-strengen te verankeren (zie sectie 1.6). Indien er gebruik gemaakt wordt van een bindingsoppervlak, dan zal men het bindingsoppervlak letterlijk afspoelen zodat alleen de gebonden moleculen overblijven. Bij het gebruik van magnetische korrels kan men met behulp van een magneet de gebonden moleculen vasthouden terwijl men de inhoud van de proefbuis vervangt door een nieuwe lege oplossing.

3.6.6 Split *(splitsen)*

Beschouw een label $\sigma \in \{\#_2, \#_3, \#_4, \#_6, \#_8\}$ waarbij σ een label is van minstens één boog in een sticker complex $C = \big(V,\ E,\ \lambda,\ \mu,\ \mathit{immob},\ \mathit{blocked}\big)$. We beschouwen bovendien ook de waarden in tabel 3.1.

Label	Open	Place
$\#_2$	*true*	before
$\#_3$	*true*	before
$\#_4$	*true*	after
$\#_6$	*false*	after
$\#_8$	*false*	before

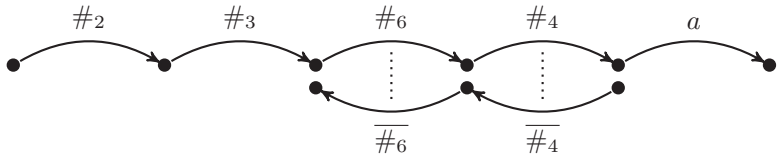
Tabel 3.1: Alle mogelijke splitpoints

Elk tripel *(label, open, place)* in deze tabel wordt een *splitpoint* genoemd. We noemen een boog *open* als deze niet voorkomt in *blocked* en niet in μ. Als we C splitsen op σ, genoteerd met $\texttt{split}(C, \sigma)$, moeten we eerst het splitpoint $\big(\sigma,\ \text{open}_\sigma,\ \text{place}_\sigma\big)$ voor σ bepalen. Vervolgens bepalen we de verzameling B van alle bogen in C met label σ en die wel of niet open zijn afhankelijk van de waarde open_σ uit het splitpoint. Daarna bepalen we al de knopen waar de splitsing moet plaatsvinden. Indien de waarde van place_σ "before" is, beschouwen we alle startknopen van de bogen in B. Indien de waarde "after" is, dan beschouwen we al de eindknopen. We noemen de verzameling van deze knopen S. Voor elke knoop $u \in S$ doen we nu volgende stappen:

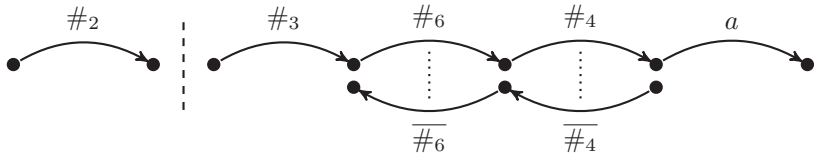
1. Als u een inkomende boog heeft, noemen we deze e_1. Als u een uitgaande boog heeft, noemen we deze e_2.
2. Als zowel e_1 als e_2 bestaan, dan vervangen we u door twee knopen u_1 en u_2 waarbij we e_1 laten toekomen in u_1 en e_2 laten vertrekken vanuit u_2. Indien e_1 en e_2 niet beiden bestaan, dan hoeven we verder niets te doen en gaan we verder met de volgende knoop (er valt immers niets te splitsen).
3. Indien er een knoop u' bestaat met een inkomende boog e_4 én een uitgaande boog e_3 en waarbij bovendien $\{e_1, e_3\} \in \mu$ en $\{e_2, e_4\} \in \mu$, dan splitsen we u' op analoge wijze in u'_1 en u'_2.

Nadat we deze stappen voor alle knopen in S voltooid hebben, hebben we het resultaat van $\texttt{split}(C, \sigma)$ berekend. Laten we een voorbeeld uitwerken op het sticker complex C dat gegeven is in figuur 3.7(a). Het resultaat van de operatie $\texttt{split}(C, \#_2)$ is gelijk aan het sticker complex C. Dit komt omdat we volgens het splitpoint van $\#_2$ (zie tabel 3.1), de startknoop van de boog moeten nemen. Deze startknoop heeft echter enkel een uitgaande boog en geen inkomende boog dus moeten we, zoals stap 2 hierboven aangeeft, niets splitsen.

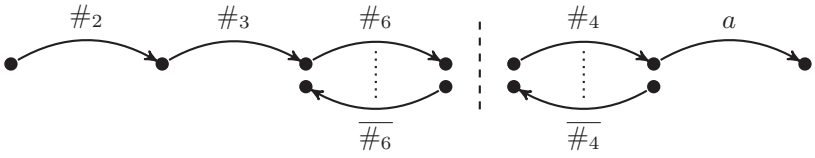
We gaan verder met de operatie $\texttt{split}(C, \#_3)$. Volgens het splitpoint van $\#_3$ moeten de bogen met label $\#_3$ open zijn en moeten we splitsen op de startknoop. De enige boog in C met als label $\#_3$ is open en de startknoop heeft zowel een uitgaande als een inkomende boog. Bij deze bewerking zal er dus wel gesplitst moeten worden. Het resultaat hiervan wordt getoond in figuur 3.7(b).



(a) Het sticker complex C



(b) Resultaat van de operatie $\texttt{split}(C, \#_3)$



(c) Resultaat van de operatie $\texttt{split}(C, \#_6)$

Figuur 3.7: Voorbeelden van de `split` operatie

Voor $\texttt{split}(C, \#_6)$ kunnen we een analoge redenering volgen. Volgens het splitpoint mogen de bogen niet open zijn en moeten we splitsen op de eindknoop van de boog. De enige boog in C met label $\#_6$ is niet open en de eindknoop heeft zowel een inkomende als een uitgaande boog. We kunnen dus splitsen op deze eindknoop. Merk op dat deze eindknoop een tegenoverliggende knoop u' heeft, en we hier ook moeten splitsen. Knoop u' heeft immers zowel een inkomende als een uitgaande boog (zie stap 3 hierboven). Het resultaat van de operatie ziet u in figuur 3.7(c).

Tenslotte beschouwen we de operatie $\texttt{split}(C, \#_4)$. Volgens het splitpoint moeten de bogen open zijn en moeten we splitsen op de eindknoop. Maar al de bogen in C met label $\#_4$ zijn niet open, dus kunnen we nergens splitsen en is het resultaat gewoon gelijk aan C. Merk op dat hoewel in dit voorbeeld slechts op één plaats gesplitst werd (als we konden splitsen), dit in andere gevallen ook op meerdere plaatsen tegelijkertijd kan gebeuren ondanks dat we maar één `split`-operatie gebruiken.

Bij werkelijk DNA zal deze operatie worden gerealiseerd met behulp van restrictie enzymen (zie sectie 1.7). Omdat we enkel kunnen splitsen op de labels in Θ (weergeven in tabel 3.1) hebben we slechts vijf split-basen-sequenties. We gebruiken dan ook maar vijf restrictie-enzymen. Elk symbool in Θ zal gecodeerd worden door de restrictie-enzymherkenningsplaats sequentie van een bepaald restrictie enzym. Deze wordt aangevuld met basen totdat de totale sequentie dezelfde lengte heeft als die van de andere symbolen in Σ. Door de definitie van deze operatie kunnen we echter enkel restrictie enzymen gebruiken die zogenaamde *blunt* uiteinden als resultaat geven (zie figuur 1.7(b) op pagina 11).

3.6.7 Block en Blockfrom *(blokkeren)*

Er zijn twee blocking operaties. Bij beide operaties veronderstellen we dat het sticker complex $C = \big(V,\ E,\ \lambda,\ \mu,\ \mathit{immob},\ \mathit{blocked}\big)$ verzadigd is, met andere woorden dat $\texttt{hybridize}(C) = C$. Als C niet verzadigd is dan zijn beide operaties ongedefinieerd.

De eenvoudigste operatie is $\texttt{block}(C, \sigma)$ met $\sigma \in \Sigma$ (let op, enkel Σ en niet $\overline{\Sigma}$). Het resultaat van deze operatie is het complex $C' = \big(V,\ E,\ \lambda,\ \mu,\ \mathit{immob},\ \mathit{blocked}'\big)$ bekomen uit C door al de open bogen met label σ aan $\mathit{blocked}'$ toe te voegen. Formeel kunnen we dit noteren met:

$$\mathit{blocked}' = \mathit{blocked} \cup \{e \mid e \in E \wedge \lambda(e) = \sigma \wedge \neg(\exists e' \in E \colon \{e, e'\} \in \mu)\}$$

De andere operatie is $\texttt{blockfrom}(C, \sigma)$ met C een sticker complex zoals hierboven beschreven en $\sigma \in \Sigma$. Voor deze operatie beschouwen we bovendien een continue deelstreng s in C. We noemen s een *σ-blocking range* als het voldoet aan volgende drie voorwaarden:

1. Al de bogen in s zijn open (zoals gedefinieerd hierboven).
2. Ofwel bevat de deelstreng s de eerste boog van het gehele streng, ofwel is de boog voorgaand (in de richting van de bogen) aan s geblokkeerd (komt voor in μ of in *blocked*).
3. De laatste boog van s is gelabeld met σ.

We definiëren $\texttt{blockfrom}(C, \sigma)$ dan als het sticker complex bekomen uit C waarbij al de bogen die voorkomen in een σ-blocking range, zijn toegevoegd aan *blocked*. Een voorbeeld van deze operatie ziet u in figuur 3.8. Ondanks dat er in dit voorbeeld slechts één σ-blocking range is, kunnen er dit in andere gevallen meerderen zijn.

(a) Het sticker complex C

(b) Het resultaat van $\texttt{blockfrom}(C, b)$

Figuur 3.8: Voorbeeld van de `blockfrom` operatie

Zoals reeds aangeven bij de definitie van precomplexen (zie sectie 3.4.2) wordt de eenvoudige $\texttt{block}(C, \sigma)$-operatie in werkelijkheid gerealiseerd door het toevoegen van een oligo die complementair is aan σ. Deze oligo zal op al de DNA sequenties equivalent aan σ hybridiseren waardoor al de σ-bogen "geblokkeerd" zijn. Het 3'-uiteinde van deze primer is echter veranderd naar zijn dideoxy-variant. Een dideoxynucleotide verschilt van een gewone deoxynucleotide door het OH-uiteinde dat vervangen is door een H-uiteinde [Rus10]. Het OH-uiteinde van de gewone nucleotide kan je in figuur 1.1 op pagina 4 linksonder en rechtsboven zien. Door deze kleine aanpassing zal het polymerase enzym stoppen bij

het dideoxy-uiteinde en de DNA-streng niet meer verder aanvullen. Dit is nodig om ongewenste effecten van latere `blockfrom`-operaties (die gebruik maken van polymerase) te voorkomen. Bij de `blockfrom`(C, σ)-operatie zullen we eerst (normale) primers toevoegen die complementair zijn aan σ waarna we DNA polymerase laten starten op deze primers om zo de daarop volgende basen aan te vullen.

3.6.8 Blockexcept *(blokkeer behalve)*

We introduceren ook nog een extra operatie op ℓ-complexen (zie sectie 3.5) die eigenlijk slechts een afkorting is van verschillende `block` en `blockfrom`-operaties. De definitie gaat als volgt: Zij n een natuurlijk getal en zij C een sticker complex dat voldoen aan volgende voorwaarden:

1. C is een ℓ-complex met $\ell \geq n$ (zie sectie 3.5);
2. in elk ℓ-vector streng binnen C zijn ofwel alle bogen geblokkeerd[4] ofwel is geen enkele boog geblokkeerd;
3. C is verzadigd, met andere woorden $C =$ `hybridize`(C).

`blockexcept`(C, n) is dan gedefinieerd als het sticker complex bekomen uit C door voor elke ℓ-vector $(e_0,\ e_1,\ ...,\ e_l,\ e_{l+1})$ die nog niet geblokkeerd is, alle bogen behalve e_n te blokkeren. Indien C of n niet voldoen aan de bovengenoemde voorwaarden, dan is `blockexcept`(C, n) ongedefinieerd.

De `blockexcept`-operatie wordt in DNA geïmplementeerd met behulp van een extra index alfabet $\Phi = \{\phi_1, \phi_2, ..., \phi_l\}$ zodat we de implementatie van een ℓ-vector $\#_3 v_1 v_2 ... v_l \#_4$ met $v_i \in \Lambda$ voor $i = 1, 2, ..., l$ kunnen aanpassen naar $\#_3 \phi_1 v_1 \phi_2 v_2 ... v_l \#_4$ zodat ϕ_i de 'index' aangeeft van karakter v_i. De symbolen in Φ hebben elk ook een eigen (unieke) DNA sequentie. Vervolgens definiëren we `blockexcept`(C, n) als de samenstelling van de operaties `block`$(C, \#_3)$; `blockfrom`(C, ϕ_{n-1}); `block`(C, ϕ_{n+1}); `blockfrom`$(C, \#_4)$.

3.6.9 Cleanup *(opschonen)*

De `cleanup`-operatie maakt matchings en blokkeringen in een sticker complex C ongedaan en verwijdert alle strengen behalve de langste positieve streng. Er wordt bij deze operatie echter verondersteld dat er een positieve streng in C bestaat met een lengte groter dan twee[5] en dat alle positieve strengen minstens één open boog hebben. Als C niet aan deze veronderstellingen voldoet, dan is `cleanup`(C) ongedefinieerd. Anders is `cleanup`(C) gelijk aan de unie van alle positieve strengen in C met maximale lengte waarbij er geen gematchte of geblokkeerde bogen meer zijn.

Deze operatie wordt in werkelijkheid gerealiseerd door eerst te denatureren waarna de losgekomen geïmmobiliseerde strengen uit de proefbuis verwijderd worden. Hierna wordt de langste DNA-streng gescheiden van de anderen via gel- of

[4] De boog bevindt zich in μ of in *blocked*

[5] In de paper [GV10] vereist men voor *alle* positieve strengen een lengte groter dan twee. De reden waarom ik dit veranderd heb, kan u lezen in sectie 5.1.

capillaire elektroforese (zie sectie 1.5). Door de voorwaarden in de definitie van sticker complexen, is het resultaat van deze operaties ofwel leeg, ofwel een verzameling van positieve DNA-strengen. We vereisen immers dat er minstens één positieve streng aanwezig is met een lengte groter of gelijk aan drie zodat het resultaat zeker geen sticker (die een maximale lengte van twee hebben) kan zijn.

3.7 DNAQL

DNAQL is een querytaal voor het sticker complex datamodel zoals SQL dat is voor het relationeel model. Het is een beperkte functionele programmeertaal voor het beschrijven van functies (algoritmen) van ℓ-complexen naar ℓ-complexen. Een belangrijke eigenschap hierbij is dat een DNAQL programma onafhankelijk van ℓ werkt en herbruikt kan worden voor verschillende dimensies. Momenteel is het nog niet bewezen dat DNAQL al dan niet Turing-compleet is, maar voor het gebruik als database querytaal is dit ook niet noodzakelijk. Een groot deel van de operaties uit DNAQL komen overeen met de operaties uit het sticker complex datamodel. Hier zijn dan extra's zoals een emptiness test, variabelen, tellers die optellen tot de dimensie ℓ, een beperkte for-lus voor het itereren over een teller en de mogelijkheid om functies te definiëren, aan toegevoegd. Dit laatste heb ik zelf aan de programmeertaal toegevoegd zodat het mogelijk is om afkortingen te introduceren alsook gemakkelijker de input parameters te bepalen.

3.7.1 Syntax

De syntax van DNAQL is gegeven in de grammatica in figuur 3.9. Een uitdrukking die voldoet aan deze grammatica is een DNAQL *programma*. Een uitdrukking die enkel voldoet aan de regel bij het niet-terminale symbool $\langle$*expression*$\rangle$, noemen we een DNAQL *expressie*. We kunnen twee soorten variabelen onderscheiden. Enerzijds zijn er variabelen die een sticker complex vertegenwoordigen en anderzijds variabelen die de functie van teller vervullen (van 1 tot ℓ). Complex-variabelen worden geïnitialiseerd (gebonden) door `let`-constructies en parameterdefinities. De teller-variabelen worden gebonden door for-lussen. De vrije (ongebonden) complex-variabelen in een DNAQL expressie staan voor input-variabelen en bevatten de complexen die als input aan de expressie worden meegegeven. Formeel definiëren we een DNAQL *programma* nu als een uitdrukking die voldoet aan deze grammatica en waar bovendien geen vrije teller- of complex-variabelen in voorkomen en die een functiedefinitie voor de functie `main` bevat. Elke teller-variabele moet dus een overeenkomstige for-lus hebben en elke complex-variabele een overeenkomstige let-expressie of parameterdefinitie.

De betekenissen van de regels bij het niet-terminale symbool $\langle$*constant*$\rangle$ in de grammatica zijn:

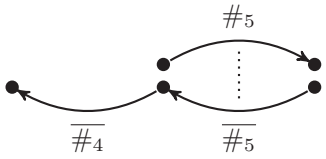
- Een woord $w \in \Sigma^+$ representeert een enkelvoudige, lineaire, positieve streng die het woord w vormt door al de symbolen van de bogen achter elkaar te plaatsen (in de richting van de bogen);

$$
\begin{array}{rcl}
\langle program \rangle & \rightarrow & \langle functionlist \rangle \\
\langle functionlist \rangle & \rightarrow & \langle functiondefinition \rangle \mid \langle functiondefinition \rangle\ \langle functionlist \rangle \\
\langle functiondefinition \rangle & \rightarrow & \texttt{fun}\ \langle identifier \rangle (\ \langle parameterlist \rangle\ ) := \langle expression \rangle \\
\langle parameterlist \rangle & \rightarrow & \langle \rangle \mid \langle complexvar \rangle \mid \langle complexvar \rangle, \langle parameterlist \rangle \\
\langle expression \rangle & \rightarrow & \langle complexvar \rangle \mid \langle foreach \rangle \mid \langle if \rangle \mid \langle let \rangle \mid \langle operator \rangle \mid \\
 & & \langle functioncall \rangle \mid \langle constant \rangle \\
\langle foreach \rangle & \rightarrow & \texttt{for}\ \langle complexvar \rangle := \langle expression \rangle\ \texttt{iter}\ \langle counter \rangle\ \texttt{do}\ \langle expression \rangle \\
\langle if \rangle & \rightarrow & \texttt{if empty}(\langle complexvar \rangle)\ \texttt{then}\ \langle expression \rangle\ \texttt{else}\ \langle expression \rangle \\
\langle let \rangle & \rightarrow & \texttt{let}\ x := \langle expression \rangle\ \texttt{in}\ \langle expression \rangle \\
\langle operator \rangle & \rightarrow & ((\langle expression \rangle) \cup (\langle expression \rangle)) \\
 & \mid & \texttt{union}(\langle expression \rangle, \langle expression \rangle) \\
 & \mid & ((\langle expression \rangle) - (\langle expression \rangle)) \\
 & \mid & \texttt{difference}(\langle expression \rangle, \langle expression \rangle) \\
 & \mid & \texttt{hybridize}(\langle expression \rangle) \\
 & \mid & \texttt{ligate}(\langle expression \rangle) \\
 & \mid & \texttt{flush}(\langle expression \rangle) \\
 & \mid & \texttt{split}(\langle expression \rangle, \langle splitpoint \rangle) \\
 & \mid & \texttt{split}(\langle expression \rangle, \langle complexvar \rangle) \\
 & \mid & \texttt{block}(\langle expression \rangle, \Sigma) \\
 & \mid & \texttt{block}(\langle expression \rangle, \langle complexvar \rangle) \\
 & \mid & \texttt{blockfrom}(\langle expression \rangle, \Sigma) \\
 & \mid & \texttt{blockfrom}(\langle expression \rangle, \langle complexvar \rangle) \\
 & \mid & \texttt{blockexcept}(\langle expression \rangle, \langle counter \rangle) \\
 & \mid & \texttt{cleanup}(\langle expression \rangle) \\
\langle functioncall \rangle & \rightarrow & \langle identifier \rangle (\ \langle expressionlist \rangle\ ) \\
\langle expressionlist \rangle & \rightarrow & \langle \rangle \mid \langle expression \rangle \mid \langle expression \rangle, \langle expressionlist \rangle \\
\langle constant \rangle & \rightarrow & \Sigma^+ \mid \left(\overline{\Sigma} - \overline{\Lambda}\right)\left(\overline{\Sigma} - \overline{\Lambda}\right) \mid \texttt{immob}(\overline{\Sigma}) \\
 & \mid & \texttt{leftboot} \mid \texttt{rightboot} \mid \texttt{empty} \\
\langle splitpoint \rangle & \rightarrow & \#_2 \mid \#_3 \mid \#_4 \mid \#_6 \mid \#_8
\end{array}
$$

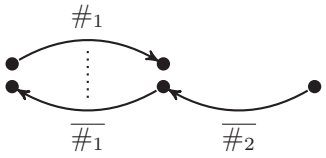
Figuur 3.9: De syntax van DNAQL

- Een tweeletterwoord $\overline{a}\overline{b}$ met $a, b \in \Sigma - \Lambda$, staat voor een enkelvoudige, lineaire, negatieve streng met lengte twee van de vorm:

- `immob`$(\overline{a})$ met $a \in \Sigma$, representeert een enkelvoudige, lineaire, geïmmobiliseerde boog met label $\overline{a}$;
- `leftboot` en `rightboot` zijn geïllustreerd in figuur 3.10;
- `empty` representeert het lege complex $(\emptyset,\ \emptyset,\ \emptyset,\ \emptyset,\ \emptyset,\ \emptyset)$.



(a) Sticker complex `rightboot`



(b) Sticker complex `leftboot`

Figuur 3.10: De `left-` en `rightboot` complexen

3.7.2 Semantiek

De evaluatie van een DNAQL expressie e over een ℓ-complex maakt gebruik van een *ℓ-complex toekenning* β en een *ℓ-teller toekenning* γ. Een ℓ-complex toekenning kent ℓ-complexen toe aan complex-variabelen. Je kan dit vergelijken met een tabel van variabele namen en toegekende waarden (complexen). Een ℓ-teller toekenning kent op dezelfde wijze een integer waarde uit het bereik $[1, ..., \ell]$ toe aan elke teller-variabele. β moet bovendien gedefinieerd zijn voor alle vrije complex-variabelen van e en γ moet alle vrije teller-variabelen van e bevatten. Als aan deze voorwaarden voldaan is, dan evalueert e tot een ℓ-complex wat we noteren met $[\![e]\!]^{\ell}(\beta, \gamma)$. De evaluatie van de verschillende DNAQL expressies leiden we nu inductief af op volgende manier:

Complex-variabele: We beschouwen eerst de eenvoudigste situatie waarbij e de expressie x is, met x een complex-variabele. De evaluatie van e bestaat dan uit de waarde van x die gegeven wordt door de ℓ-complex toekenning β. Dit wordt geïllustreerd in figuur 3.11. In dergelijke figuren staan de voorwaarden voor de evaluatie boven de horizontale lijn. Het uiteindelijke resultaat van de expressie, uitgedrukt in operaties van het sticker complex datamodel, staat onder

de horizontale lijn. De operaties van DNAQL en van het sticker complex datamodel hebben vaak dezelfde naam en om beiden goed te kunnen onderscheiden zullen we in deze sectie de operaties van het sticker complex datamodel noteren met een Sans serif lettertype. Indien de complex-variabele evalueert tot een

$$\frac{x \text{ is a complex variable}}{[\![x]\!](\beta,\gamma) = \beta(x)}$$

Figuur 3.11: Semantiek van een DNAQL complex-variabele

sticker complex met slechts één boog, dan kan deze ook gebruikt worden als tweede parameter bij de `split`, `block` of `blockfrom` operaties.

Union: We gaan verder met het geval dat e de union operatie bevat. Deze zal het resultaat van twee expressies samenvoegen in één sticker complex. De semantiek wordt gegeven in figuur 3.12. Deze is analoog voor $\texttt{union}(e_1, e_2)$.

$$\frac{[\![e_1]\!](\beta,\gamma) = C_1 \qquad [\![e_2]\!](\beta,\gamma) = C_2}{[\![e_1 \cup e_2]\!](\beta,\gamma) = C_1 \cup C_2}$$

Figuur 3.12: Semantiek van de DNAQL union-operatie

Difference: De difference operatie, in het geval $e = e_1 - e_2$, zal op de resultaten van de twee expressie e_1 en e_2 (die een gedefinieerd resultaat moeten hebben) de sticker complex operatie difference toepassen. Dit ziet u in figuur 3.13, waarbij dit voor de alternatieve notatie $\texttt{difference}(e_1, e_2)$ volledig analoog gaat.

$$\frac{[\![e_1]\!](\beta,\gamma) = C_1 \qquad [\![e_2]\!](\beta,\gamma) = C_2 \qquad C_1 - C_2 \text{ is gedefinieerd}}{[\![e_1 - e_2]\!](\beta,\gamma) = C_1 - C_2}$$

Figuur 3.13: Semantiek van de DNAQL difference-operatie

Hybridize: $e = \texttt{hybridize}(e')$ zal de sticker complex operatie hybridize toepassen op het resultaat van de expressie e'. Dit is geïllustreerd in figuur 3.14.

$$\frac{[\![e']\!](\beta,\gamma) = C'}{[\![\texttt{hybridize}(e')]\!](\beta,\gamma) = \mathsf{hybridize}(C')}$$

Figuur 3.14: Semantiek van de DNAQL hybridize-operatie

Ligate: In het geval dat e de `ligate`-operatie bevat, passen we de sticker complex operatie ligate toe op het resultaat van het argument e'. De semantiek is gegeven in figuur 3.15.

$$\frac{[\![e']\!](\beta,\gamma) = C'}{[\![\texttt{ligate}(e')]\!](\beta,\gamma) = \mathsf{ligate}(C')}$$

Figuur 3.15: Semantiek van de DNAQL ligate-operatie

Flush: Ook bij $e = \texttt{flush}(e')$ is er een rechtstreekse overeenkomst tussen de DNAQL operatie `flush` en de operatie flush van het sticker complex datamodel. Dit ziet u in figuur 3.16.

$$\frac{[\![e']\!](\beta,\gamma) = C'}{[\![\texttt{flush}(e')]\!](\beta,\gamma) = \mathsf{flush}(C')}$$

Figuur 3.16: Semantiek van de DNAQL flush-operatie

Split: De `split`-operatie is eveneens een rechtstreekse toepassing van de sticker complex operatie split. Ook hier wordt er gewerkt met splitpoints. De volledige semantiek ziet u in figuur 3.17 waarbij $e = \texttt{split}(e', \sigma)$. Indien de tweede parameter een complex-variabele is, dan moet deze evalueren tot een sticker complex met maar één boog en wordt het label van die boog als splitpoint gebruikt.

$$\frac{[\![e']\!](\beta,\gamma) = C' \qquad \sigma \in \{\#_2, \#_3, \#_4, \#_6, \#_8\}}{[\![\texttt{split}(e', \sigma)]\!](\beta,\gamma) = \mathsf{split}(C', \sigma)}$$

Figuur 3.17: Semantiek van de DNAQL split-operatie

Block en Blockfrom: Ook de operaties `block` en `blockfrom` zijn een triviale toepassing van hun overeenkomstige sticker complex operaties (zie figuur 3.18). Ook hier geldt dat indien de tweede parameter een complex-variabele is, deze moet evalueren tot een sticker complex met slechts één boog. Het label van die boog wordt dan gebruikt als label voor de blokkering.

Blockexcept: Voor het geval $e = \texttt{blockexcept}(e', i)$ gebruiken we een tellervariabele i die bovendien gebonden is. Deze teller-variabele zal aangeven welke boog we niet zullen blokkeren zoals gedefinieerd is in de blockexcept-operatie van het sticker complex datamodel. De semantiek is gegeven in figuur 3.19.

$$\frac{[\![e']\!](\beta,\gamma) = C' \qquad \mathsf{block}(C',\sigma) \text{ is gedefinieerd}}{[\![e]\!](\beta,\gamma) = [\![\texttt{block}(e',\sigma)]\!](\beta,\gamma) = \mathsf{block}(C',\sigma)}$$

$$\frac{[\![e']\!](\beta,\gamma) = C' \qquad \mathsf{blockfrom}(C',\sigma) \text{ is gedefinieerd}}{[\![e]\!](\beta,\gamma) = [\![\texttt{blockfrom}(e',\sigma)]\!](\beta,\gamma) = \mathsf{blockfrom}(C',\sigma)}$$

Figuur 3.18: Semantiek van de DNAQL block- en blockfrom-operaties

$$\frac{[\![e']\!](\beta,\gamma) = C' \qquad i \text{ is a counter} \qquad \mathsf{blockexcept}(C',\gamma(i)) \text{ is gedefinieerd}}{[\![\texttt{blockexcept}(e',i)]\!](\beta,\gamma) = \mathsf{blockexcept}(C',\gamma(i))}$$

Figuur 3.19: Semantiek van de DNAQL blockexcept-operatie

Cleanup: Indien $e = \texttt{cleanup}(e')$, spreekt de semantiek van de `cleanup`-operatie voor zich daar deze enkel een toepassing is van de overeenkomstige sticker complex operatie (zie figuur 3.20).

$$\frac{[\![e']\!](\beta,\gamma) = C' \qquad \mathsf{cleanup}(C') \text{ is gedefinieerd}}{[\![\texttt{cleanup}(e')]\!](\beta,\gamma) = \mathsf{cleanup}(C')}$$

Figuur 3.20: Semantiek van de DNAQL cleanup-operatie

Let: De `let`-operatie zorgt voor de introductie van een complex-variabele[6] x en kent aan deze variabele de waarde bekomen uit een expressie e_1 toe. Daarna zal deze variabele gebruikt worden in een expressie e_2 (waar x als vrije variabele in voorkomt) waarvan het resultaat C_2 ook als resultaat voor de `let`-operatie dient. De notatie $\beta[x := C_1]$ betekent dat de toekenning β een extra record krijgt voor x met bijhorende waarde C_1. Formeel wordt de semantiek gegeven in figuur 3.21.

$$\frac{[\![e_1]\!](\beta,\gamma) = C_1 \qquad [\![e_2]\!](\beta[x := C_1],\gamma) = C_2}{[\![\texttt{let } x := e_1 \texttt{ in } e_2]\!](\beta,\gamma) = C_2}$$

Figuur 3.21: Semantiek van de DNAQL let-operatie

If-test: De if-test in DNAQL controleert of een gegeven variabele het lege complex bevat. Indien dit zo is, zal deze het resultaat van de expressie na het

[6] x kan hier een willekeurige naam zijn.

`then` keyword geven. In het andere geval wordt het resultaat van de expressie na het `else` gedeelte teruggegeven. Dit is geïllustreerd in figuur 3.22 met e als de expressie `if empty`(x) `then` e_1 `else` e_2.

$$\frac{[\![e_1]\!](\beta,\gamma) = C_1 \qquad \beta(x) \text{ is het lege complex}}{[\![\texttt{if empty}(x)\ \texttt{then}\ e_1\ \texttt{else}\ e_2]\!](\beta,\gamma) = C_1}$$

$$\frac{[\![e_2]\!](\beta,\gamma) = C_2 \qquad \beta(x) \text{ is } \textit{niet} \text{ het lege complex}}{[\![\texttt{if empty}(x)\ \texttt{then}\ e_1\ \texttt{else}\ e_2]\!](\beta,\gamma) = C_2}$$

Figuur 3.22: Semantiek van de DNAQL if-test

For-lus: Indien e een for-lus expressie is, introduceert deze zowel een complex-variabele x als een teller-variabele i. De initiële waarde van x is gelijk aan het resultaat van een expressie e_1 en i heeft de waarde 1. Hierna worden x en i gebruikt voor de evaluatie van e_2 wat als resultaat C_i geeft. Vervolgens wordt de waarde van i met één verhoogd. Als i dan een waarde heeft kleiner als ℓ, dan stellen we x gelijk aan C_{i-1} (de waarde van i is met één verhoogd) en evalueren we e_2 met deze nieuwe waarden voor x en i. Als i groter is als ℓ, is $(C_{i-1} =)\ C_\ell$ de output van de gehele for-lus expressie. Deze semantiek is formeel beschreven in figuur 3.23. De uitdrukkingen $\beta[x := C_{n-1}]$ en $\gamma[i := n]$ geven aan dat de toekenningen β en γ een nieuwe waarde krijgen voor x en i.

$$\frac{[\![e_1]\!](\beta,\gamma) = C_0 \qquad [\![e_2]\!](\beta[x := C_{n-1}], \gamma[i := n]) = C_n \text{ for } n = 1, \ldots, \ell}{[\![\texttt{for}\ x := e_1\ \texttt{iter}\ i\ \texttt{do}\ e_2]\!](\beta,\gamma) = C_\ell}$$

Figuur 3.23: Semantiek van de DNAQL for-lus

Functies: Er rest ons nog alleen de semantiek van het geval dat e een functie-aanroep is, te definiëren. De formele beschrijving ziet u in figuur 3.24. Intuïtief komt de semantiek neer op het toevoegen van alle toekenningen van waarden aan hun corresponderende parameter om vervolgens de expressie in de functiedefintie hiermee te evalueren. Het resultaat van de functie aanroep is dan het bekomen sticker complex.

$$\frac{\begin{array}{c}\exists(\texttt{fun}\ f(\ x_1, x_2, ..., x_i\ ) := e_0) \qquad [\![e_1]\!](\beta,\gamma) = C_1 \qquad [\![e_2]\!](\beta,\gamma) = C_2 \\ (...) \qquad [\![e_i]\!](\beta,\gamma) = C_i \qquad [\![e_0]\!](\beta[x_1 := C_1, x_2 := C_2, ..., x_i := C_i], \gamma) = C_0\end{array}}{[\![f(e_1, e_2, ..., e_i)]\!](\beta,\gamma) = C_0}$$

Figuur 3.24: Semantiek van de DNAQL functie aanroep

De dimensie ℓ werd in de figuren 3.11 tot 3.24 weggelaten om de regels overzichtelijk te houden. Omdat sommige operaties op complexen niet altijd een resultaat geven, kan de evaluatie van $[\![e]\!]^{\ell}(\beta, \gamma)$ falen en is het resultaat ongedefinieerd.

De semantiek van een DNAQL programma e, wat we noteren met $[\![e]\!](\beta, \emptyset)$ of eenvoudigweg door $[\![e]\!](\beta)$, is de evaluatie van de functie `main` binnen het programma. Deze functie moet immers gedefinieerd zijn wil e een geldig DNAQL programma zijn. De waarden voor de parameters van de `main`-functie moeten door de gebruiker worden aangeleverd als inputwaarden bij het uitvoeren van het programma.

Hoofdstuk 4

Implementatie

In dit hoofdstuk wil ik het implementatie gedeelte van mijn bachelorproef bespreken. Het implementatie gedeelte bestond uit het ontwikkelen van een interpreter voor de DNAQL programmeertaal. Dit omvatte onder meer het implementeren van datastructuren voor een abstracte syntaxboom, de verschillende operaties van DNAQL, het opstellen van een bestandsformaat dat kan worden ingelezen en uitgeschreven, het visualiseren van sticker complexen... Om dit alles te realiseren, heb ik gebruik gemaakt van de programma's *Flex* en *Bison* voor het opbouwen van de abstracte syntaxboom en het *Qt* framework. De implementatie is gemaakt in de programmeertaal C++ om juist deze programma's te kunnen gebruiken en om overdraagbaar te zijn naar verschillende besturingssystemen. De applicatie werd ontwikkeld in het Engels.

Tijdens de ontwikkeling van de applicatie heb ik ook veel aandacht geschonken aan robuustheid. Voor de belangrijkste operaties in het programma zijn er unittests geschreven en werd er telkens gecontroleerd op geheugenlekken. De gehele communicatie tussen de (G)UI en het model gebeurt via het Model-View-Controller design pattern. De datastructuren voor het representeren van een sticker complex vormen hierbij het 'model', de datastructuren voor de abstracte syntaxboom zijn de 'controller' en de klassen voor de communicatie met de gebruiker (zowel grafisch als via de commandline) de 'view'. Hiernaast werd ook veel aandacht besteed aan foutafhandeling en documentatie (in doxygen stijl). Het project bestaat uiteindelijk uit 140 bestanden met een totaal aan ongeveer 6700 regels code en 1700 regels commentaar.

Als eerst bespreek ik kort het gebruik van de simulator in een korte handleiding. Daarna ga ik dieper in op bestandsformaat dat ik gebruik voor het inlezen van sticker complexen. Vervolgens geef ik een korte bespreking van de verschillende geïmplementeerde datastructuren. Op het einde van dit hoofdstuk behandel ik uitgebreid de belangrijkste ontworpen algoritmen.

4.1 Een korte handleiding

Alvorens aan de eigenlijke bespreking van de implementatie te beginnen, overloop ik eerst kort de handleiding van mijn applicatie *DNAQL Simulator*. Zo zal u de bespreking van de algoritmen en datastructuren beter in hun context kunnen plaatsen. De interactie met het programma verloopt grotendeels via de commandline interface. Bij het aanroepen van het programma geeft u daarvoor enkele vlaggen mee afhankelijk van het resultaat dat u wilt bereiken. Een vlag begint altijd met één liggend streepje. Na de vlaggen moet u ook de nodige bestanden aan het programma leveren in overeenstemming met de gekozen vlaggen. Deze bestanden kunnen zowel een DNAQL programma bevatten als een sticker complex beschrijven.

4.1.1 Het weergeven van een sticker complex

De meest eenvoudige functie van het programma DNAQL Simulator is het weergeven van een sticker complex op basis van een sticker complex bestand. Hiervoor voert u het programma uit met de vlag `-disp` met daarachter het bestand dat het sticker complex beschrijft zoals dit gedefinieerd is in sectie 4.2. Het programma zal vervolgens het opgegeven sticker complex uittekenen in het venster *Sticker Complex Visualizer* waarna u het kan manipuleren. Indien u de `-disp` vlag gebruikt, werkt het programma enkel als u het als volgt opstart:

```
./DNAQL_Simulator -disp complex-bestand.sc
```

4.1.2 Het simuleren van een DNAQL programma

De hoofdfunctionaliteit van het programma is uiteraard het simuleren (interpreteren) van een DNAQL programma. Hiervoor geeft u bij het starten van het programma eerst het bestand op dat het DNAQL programma bevat en daarna al de bestanden die de input sticker complexen beschrijven. Indien u het programma gebruikt zonder vlaggen zal het DNAQL programma wel geïnterpreteerd worden, waarbij er eventuele foutboodschappen worden uitgeprint, maar zal u niets van het resultaat zien. U kan echter gebruik maken van de volgende vlaggen:

`-v :` De visualisatie vlag zal ervoor zorgen dat het resultaat van het opgegeven DNAQL programma, wordt getekend in het *Sticker Complex Visualizer* venster.

`-debug :` Deze vlag start het programma in debug-modus. Hierbij zal er na elke operatie gepauzeerd worden en zal het tot dan toe berekende sticker complex weergeven worden. Indien de vlag `-v` ook gebruikt werd, zal het sticker complex grafisch worden weergegeven. Anders zal het worden uitgeprint in de console, in het formaat dat beschreven staat in sectie 4.2. De uitvoering van het programma kan opnieuw verder gezet worden door het drukken op de entertoets. Daarenboven zal er ook debug-informatie worden uitgeprint naar de console, bijvoorbeeld welke functie opgeroepen wordt, welke operatie wordt uitgevoerd, hoeveel componenten er in de hybridisatie zitten...

`-debug-hybrid :` Met deze vlag wordt het programma gestart in een speciale debug-modus voor de hybridisatie operatie. Deze modus is hetzelfde als de gewone debug-modus maar waarbij er in een hybridisatie operatie ook na elke hybridisatie-stap wordt gepauzeerd. Op deze manier kan men de hybridisaties zelf ook debuggen en bovendien ook mooi de werking van het hybridisatie algoritme volgen.

`-o :` Met behulp van deze vlag kan de gebruiker aangeven waar hij het resultaat van zijn DNAQL programma in het sticker complex bestandsformaat wil opslaan. De bestandsnaam moet onmiddellijk volgen na deze vlag en moet voor het programma toegankelijk zijn.

`-tree :` De abstracte syntaxboom van het DNAQL programma zal in het bestand na deze vlag worden opgeslagen in het *Graphviz dot*-formaat[1]. Dit bestand kan men dan makkelijk omzetten naar een afbeelding met behulp van het Graphviz programma `dot`. Indien men de abstracte syntaxboom bewaard heeft in het bestand *absyntree.dot* en hiervan een afbeelding wil maken in het bestand *afbeelding.png*, dan kan men het volgende commando gebruiken:

```
dot -Tpng -o afbeelding.png absyntree.dot
```

`-help :` Hiermee toont u de handleiding van het programma in de console. De handleiding omschrijft het gebruik van het programma en de functie van elke vlag.

4.1.3 Het gebruik van de Sticker Complex Visualizer

De positieve strengen van een sticker complex worden door de *Sticker Complex Visualizer* in een groene kleur getekend. De stickers (de negatieve strengen) worden in het rood getekend. Twee bogen die gehybridiseerd zijn, worden aan elkaar verbonden door een stippellijn. Net zoals de tekeningen die ik in dit proefschrift gebruik, zijn bogen die geblokkeerd zijn getekend met een gestippelde pijl en bogen die geïmmobiliseerd zijn met een dikkere pijl.

Een sticker complex wordt dynamisch getekend. Dit wil zeggen dat de posities van de knopen voortdurend worden aangepast totdat een optimale lay-out is gevonden. Dit dynamisch tekenen kan men pauzeren door op de *spatiebalk* te drukken. Bij het visualiseren van grote sticker complexen is dit soms wel aangewezen.

De verschillende knopen kunnen ook manueel verplaatst worden door er op te klikken en ze te verslepen. Op deze manier kan men, als men het dynamisch tekenen laat aanstaan, handmatig de componenten herschikken. Het is ook mogelijk om in en uit te zoomen. Om het beeld te vergroten dient men de controltoets ingedrukt te houden en naar voor te scrollen met de muis. Om het beeld te verkleinen, houdt men de controltoets ingedrukt en scroll je naar achter. Dit kan ook mogelijk met de plus- en mintoets.

[1] `http://www.graphviz.org/content/dot-language`

4.1.4 De voorbeeldbestanden

Bij de applicatie staan in de map `examples` enkele voorbeeldbestanden. Onder deze bestanden vindt men zowel DNAQL programma's als sticker complex bestanden. De DNAQL programma's zijn een implementatie van de simulaties van enkele operaties van de relationele algebra. Ze werden gebaseerd op het bewijs dat het sticker complex datamodel de relationele algebra kan simuleren [GV10].

Ik heb enkele van deze DNAQL expressies moeten aanpassen omdat ze anders niet (correct) werkten (met name deze voor het cartesisch product). Ook zijn er enkele overbodige `cleanup` en `ligate` operaties weggelaten. De simulatie van de *rename* operatie is ook meegeleverd, maar deze geeft helaas niet het gewenste resultaat.

4.1.5 Compileren van het programma

Om het DNAQL Simulator programma te compileren, dient men de *Qt toolkit* en de programma's *Flex* en *Bison* geïnstalleerd te hebben. Voor het compileren zelf voert men eerst het commando `'qmake DNAQL_Simulator.pro'` uit gevolgd door het commando `'make'` in de map met de broncode bestanden. Het is ook mogelijk om het bestand `DNAQL_Simulator.pro` te open in het programma *Qt Creator* en van daaruit het programma te compileren.

Voor het compileren en uitvoeren van de unittests dient men in het bestand `main.cpp` de regel `'#if 1'` te veranderen naar `'#if 0'`. Vervolgens dient men in het bestand `mainTestSuite.cpp` het omgekeerde te doen. Hierna compileert men het project zoals hierboven beschreven staat en kan men de unittests uitvoeren. Ik wil hierbij tenslotte nog opmerken dat de unittests gemaakt zijn met de *QTest namespace* van het *Qt framework*.

4.2 Het sticker complex bestandsformaat

Om sticker complexen makkelijk te kunnen inlezen en opslaan, heb ik een bestandsformaat gedefinieerd. Met dit bestandsformaat kan men per bestand, één sticker complex definiëren. Dit bestandsformaat wordt ook gebruikt om de output van een DNAQL programma op te slaan. De meegeleverde sticker complex bestanden eindigen allemaal op de extensie `.sc`, maar dit is geen vereiste. Het formaat bestaat uit twee grote secties, de "positieve strengen" sectie en de "stickers" sectie die van elkaar gescheiden worden door de string '%%'. Om de complexiteit van het bestandsformaat te beperken, heb ik de alfabetten Λ, Ω en Θ vastgelegd met volgende waarden:

$$\begin{aligned}\Lambda =&\{0,1,2,3,4,5,6,7,8,9\}\\ \Omega =&\{A,B,C,D,E,F,G,H,I,J,K,L,M,\\ &N,O,P,Q,R,S,T,U,V,W,X,Y,Z\}\\ \Theta =&\{\#1,\#2,\#3,\#4,\#5,\#6,\#7,\#8,\#9\}\end{aligned}$$

In de “positieve strengen” sectie worden, zoals de naam al doet vermoeden, alle positieve strengen van het sticker complex beschreven. Hierbij hanteer ik het volgend formaat: Per positief streng geeft men eerst een streng-naam op. Deze streng-naam dient om later het streng uniek te identificeren. De streng-naam moet dus per bestand uniek zijn. Verder liggen er geen restricties op de naam. Na de streng-naam volgt een sequentie van symbolen die afgesloten wordt met een puntkomma. Dit zijn de symbolen die op de bogen van het streng zullen komen, in dezelfde opgegeven volgorde. Deze symbolen moeten uit het alfabet Σ ($= \Lambda \cup \Omega \cup \Theta$) komen. Indien een bepaalde boog geblokkeerd is, dan moet zijn corresponderend symbool gevolgd worden door het karakter ‘|’. De symbolen mogen willekeurig van elkaar gescheiden worden door een witruimte[2]. Indien het streng circulair is, moet het laatste symbool van de sequentie ‘>’ zijn. Bij het inlezen van de positieve strengen wordt automatisch de dimensie van het sticker complex bepaald, en dit op basis van de eerste opgegeven ℓ-vector. Indien er positieve strengen aanwezig zijn met verschillende dimensies, dan zal het programma een foutboodschap geven. Op het einde van deze sectie moet de string ‘%%’ staan.

De “stickers” sectie definieert de stickers in het sticker complex. Merkwaardig aan deze sectie is dat de bogen in een volgorde worden opgegeven die tegengesteld is aan hun oriëntatie. Dit heb ik gedaan zodat men zowel de positieve strengen als de stickers van links naar rechts kan lezen en gemakkelijker met elkaar kan verbinden. U zal zien dat dit intuïtief het gemakkelijkst is. Bovendien is in het sticker complex datamodel de oriëntatie van de stickers toch niet van belang[3]. Per sticker geeft men dus eerst het symbool van de ‘laatste’ boog gevolgd door een witruimte. Indien de boog open is, laat men hierop het woord “open” volgen. Bij een gehybridiseerde boog, geeft men de streng-naam met daarachter het boog-nummer, tussen vierkante haakjes, van de boog waarop hij gehybridiseerd is. Tussen de streng-naam en het boog-nummer mag geen witruimte zijn. Het boog-nummer van de eerste boog in een streng is nul, en wordt daarna telkens met één verhoogd. Merk op dat dit lijkt op het adresseren van een element in een array. Hierna volgt eventueel terug een witruimte en een tweede boog die hetzelfde formaat volgt. De sticker moet ook worden afgesloten met een puntkomma. Hierna kan dan een volgende sticker volgen. Indien de boog van een sticker geblokkeerd is, dan wordt zijn symbool ook gevolgd door een ‘|’. Bij een geïmmobiliseerde sticker-boog, wordt het symbool gevolgd door een punt. Een voorbeeld van een sticker complex bestand ziet u in listing 4.1 waarbij de visualisatie van het gedefinieerde sticker complex te zien is in figuur 4.1.

4.3 Datastructuren

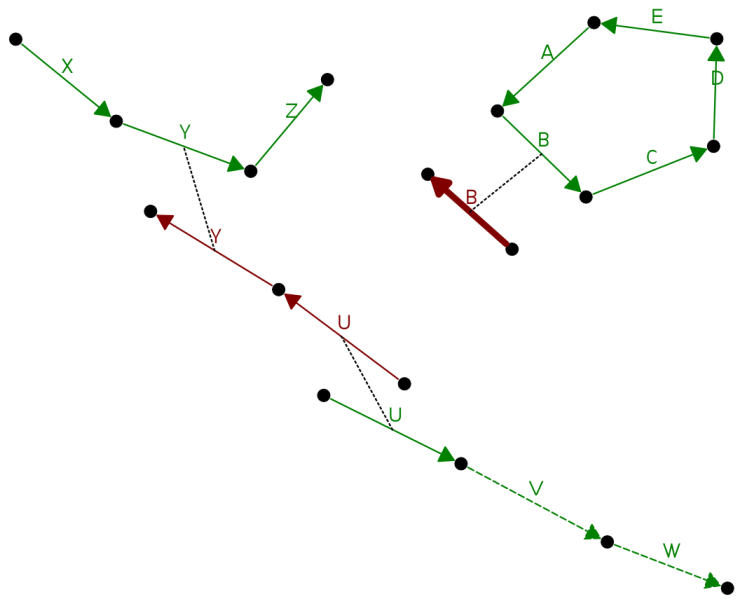
Bij het implementeren van de datastructuren van de interpreter (de ‘model’ en ‘controller’ klassen), heb ik ervoor gekozen om enkel gebruik te maken van standaard C++ code en de STL library. Op deze manier zijn deze datastructuren onafhankelijk van een bepaald framework en kunnen ze gemakkelijk hergebruikt

[2] Dit is een spatie, een tab of een newline

[3] Dit is niet helemaal waar, zie sectie 5.1

```
s1 ABC
   DE>;
s2 XYZ;
s3 U V| W| ;
 %%
B. s1[1];
Y s2[1] U s3[0];
```

Listing 4.1: Een voorbeeld sticker complex bestand

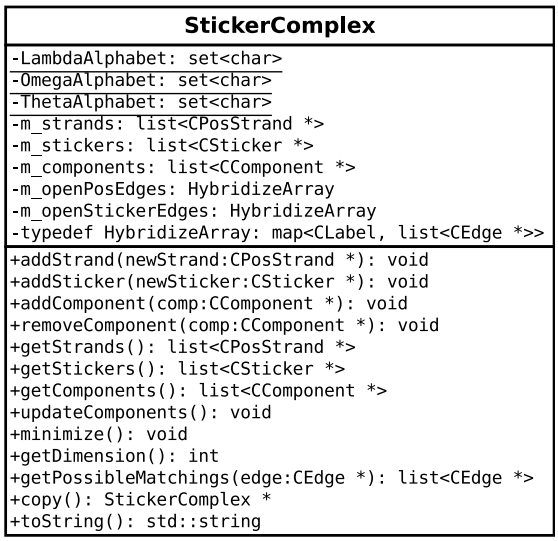


Figuur 4.1: Visualisatie van het sticker complex in listing 4.1

worden voor een andere applicatie. Dit komt bovendien ook de begrijpelijkheid van de code ten goede. Een nadeel is echter dat de veel gebruikte `set` en `map` containers enkel logaritmische toegangstijd toelaten. Als de volgende C++ standaard (C++0x) officieel beschikbaar is, kan men deze beter vervangen met de `unordered_set` en `unordered_map` containers. De datastructuren in de 'view'-klassen maken wel gebruik van een specifiek framework, namelijk het *Qt* framework. Hierna volgt een bespreking van de belangrijkste datastructuren en hun functie binnen de applicatie.

4.3.1 De klasse StickerComplex

De klasse `StickerComplex` implementeert de eigenlijke representatie van een sticker complex. Een schematische UML weergave van deze klasse ziet u in figuur 4.2. Let op, de UML diagrammen die u hier vindt, zijn telkens maar een beperkte weergave van de gehele klassenstructuur. Dit heb ik gedaan om het overzicht te behouden.



Figuur 4.2: UML weergave van de klasse `StickerComplex`

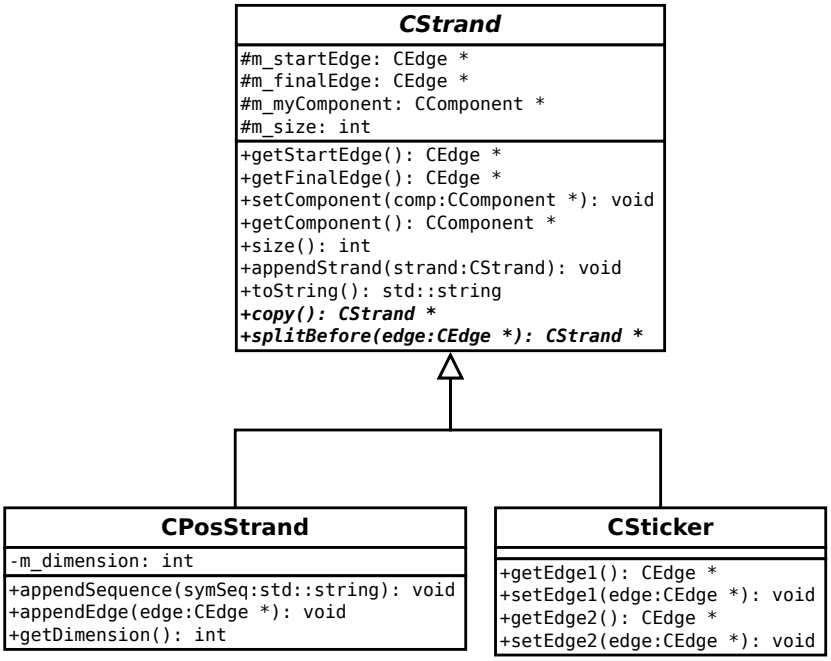
De statische variabelen `LambdaAlphabet`, `OmegaAlphabet` en `ThetaAlphabet` bevatten zoals de naam reeds doet vermoeden, de symbolen van de respectievelijke alfabetten Λ, Ω en Θ. Ze worden gebruikt voor het controleren op correcte invoer. Positieve strengen, stickers en componenten worden opgeslagen in de variabelen `m_strands`, `m_stickers` en `m_components`. U vraagt zich misschien af waarom ik niet enkel een lijst van componenten bijhoud, maar ook nog eens lijsten van positieve strengen en stickers. Ik heb dit gedaan omdat ik voor verschillende algoritmen soms enkel de stickers of de positieve strengen nodig heb. Bovendien veranderen de componenten bij nagenoeg elke operatie omdat er strengen verdwijnen of bijkomen. Aparte lijsten voor positieve strengen en stickers maken het geheugenbeheer hierdoor veel gemakkelijker. Een `StickerComplex` object zal in zijn destructor zelf al zijn strengen en componenten opruimen. De membervariabelen `m_openPosEdges` en `m_openStickerEdges` zijn een mapping van een bepaald label op een lijst van respectievelijk positieve of negatieve open bogen met dat label. Deze mapping wordt in de applicatie een `HybridizeArray` genoemd omdat deze vooral van belang zijn in de `hybridize` operatie.

Eén van de belangrijkste methoden van deze klasse is `updateComponents`. Deze methode zal op basis van de positieve strengen en stickers in `m_strands` en `m_stickers` opnieuw alle componenten berekenen en alle gerelateerde variabelen opnieuw instellen. Ze wordt telkens aangeroepen nadat er grote veranderingen in het `StickerComplex` object hebben plaatsgevonden. Een andere belangrijke methode is `minimize` die al de redundante componenten uit het sticker complex verwijdert (zie sectie 3.4.4). Ook de functie `getPossibleMatchings` is van groot belang. Deze zal, gegeven een boog (met een label), een lijst van open bogen teruggeven die mogelijk kunnen hybridiseren met de gegeven boog. Deze lijst wordt opgehaald uit `m_openPosEdges` of `m_openStickerEdges` afhankelijk van het al dan niet negatief zijn van de boog. De functie `copy` maakt een nieuw `StickerComplex` object aan dat identiek is aan het oorspronkelijke object. Deze

twee objecten hebben elk hun eigen set van strengen en componenten. De functie `toString` tenslotte, geeft een string representatie van het `StickerComplex` object terug zoals die ook gebruikt wordt in het sticker complex bestandsformaat (zie sectie 4.2). Zoals u kan zien, heb ik ervoor gekozen om de typische operaties van DNAQL niet in deze klasse als methodes onder te brengen. Ik vond dat dit de klasse te groot zou maken. De implementatie van deze operaties komt later terug in de klassen van de abstracte syntaxboom.

4.3.2 De klassen CStrand, CPosStrand en CSticker

Deze klassen dienen voor het representeren van een positieve of negatieve streng. Een streng wordt gerepresenteerd door een (dubbel gelinkte) lijst van `CEdge` objecten waarbij er telkens een pointer bijhouden wordt naar de kop (`m_startEdge`) en de staart (`m_finalEdge`) van de lijst. Dit wordt gerealiseerd in de abstracte klasse `CStrand` (zie figuur 4.3). Deze klasse heeft ook nog een variabele voor het bijhouden van een pointer naar de component waartoe hij behoort en een variabele waarin de lengte wordt bijgehouden.



Figuur 4.3: UML weergave van de klasse `CStrand` en afgeleide klassen

Een belangrijke operatie is `appendStrand` die een gegeven `CStrand` object achteraan de eigen lijst toevoegt. Er worden dus twee strengen aan elkaar geplakt. De `toString` operatie geeft een string representatie van de streng terug die overeenkomt met de specificatie van het sticker complex bestandsformaat. De abstracte functie `copy` geeft een identieke kopie van het originele object terug met een eigen lijst van bogen. De functie `splitBefore`, die eveneens abstract is, zal de eigen lijst opsplitsen juist voor het als parameter meegegeven `CEdge`

object. De gerepresenteerde streng zal dus worden gesplitst in twee strengen. De nieuwe afgesplitste streng wordt dan als return waarde teruggegeven.

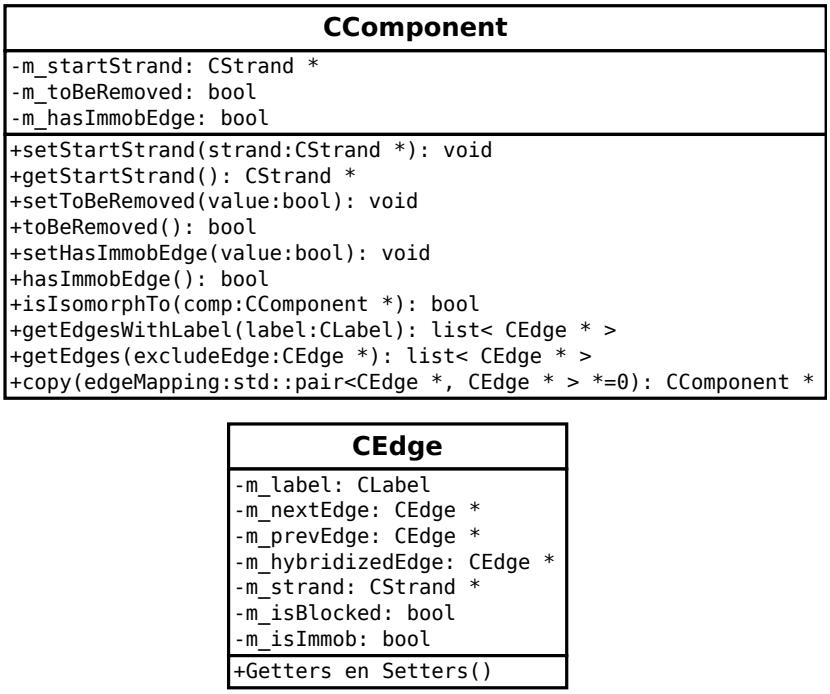
De klasse `CPosStrand` erft van de klasse `CStrand` over en implementeert deze abstracte functies en dient specifiek voor positieve strengen. Bovendien heeft deze klasse extra methoden voor het toevoegen van bogen achteraan de lijst. Dit kan zowel voor een afzonderlijk `CEdge` object als voor een sequentie van symbolen (een string) die dan worden omgezet naar bogen. Om stickers te kunnen voorstellen gebruik ik de afgeleide klasse `CSticker`. Deze klasse kan hoogstens twee bogen bevatten, die kunnen worden ingesteld of opgevraagd via de functies `getEdge1`, `getEdge2`, `setEdge1` en `setEdge2`. Let wel, hierbij is net zoals in het sticker complex bestandsformaat, de nummering van de bogen omgekeerd aan de oriëntatie van de sticker maar dit heeft verder weinig invloed op de applicatie.

4.3.3 De klassen CEdge en CComponent

Een `CEdge` object vertegenwoordigt een boog in een streng van een sticker complex. De UML weergave ziet u in figuur 4.4. Het label van de boog wordt opgeslagen in de membervariabele `m_label`. De informatie of dat de boog in het predikaat *immob* of in het predikaat *blocked* zit, wordt weergegeven door `m_isImmob` en `m_isBlocked`. Verder bevat een `CEdge` object ook nog pointers naar de vorige boog (`m_prevEdge`) en de volgende boog (`m_nextEdge`). Zo ontstaat er als het ware een dubbel gelinkte lijst structuur. De relaties die aangegeven worden door de matching-functie μ weerspiegelen zich in de pointer `m_hybridizedEdge` die het adres van de gekoppelde boog bevat. Indien onze boog niet voorkomt in μ, dan is deze pointer nul. De `CLabel` klasse bevat enkel een string die het eigenlijke label bevat en een boolean die aangeeft of het om een positief symbool gaat (uit Σ) of om een negatief (complementair) symbool (uit $\overline{\Sigma}$).

De klasse `CComponent` representeert een component van het sticker complex datamodel. De belangrijkste membervariabele is `m_startStrand`. Deze variabele is een pointer naar een willekeurige streng (een `CStrand` object) van de component dat als startpunt voor al de operaties gebruikt zal worden. Dit streng kan zowel positief als negatief zijn. Verder zijn er ook nog membervariabelen die aangeven of een component verwijderd moet worden (`m_toBeRemoved`) en of de component ergens een geïmmobiliseerde boog heeft (`m_hasImmobEdge`). Deze laatste twee variabelen zijn vooral van belang bij de hybridisatie operatie. Een beknopte UML representatie kan u zien in figuur 4.4.

De belangrijkste methoden van `CComponent` zijn `isIsomorphTo`, `getEdgesWithLabel`, `getEdges` en `copy`. De functie `isIsomorphTo` controleert of dat de opgegeven component isomorf is aan de eigen component. Het algoritme hiervoor wordt besproken in sectie 4.4.2. De functies `getEdgesWithLabel` en `getEdges` geven een lijst van bogen terug die in de component aanwezig zijn waarbij deze bogen eventueel een bepaald label hebben of waarbij eventueel een bepaalde boog wordt uitgesloten. De `copy` functie maakt een identieke kopie van de gehele component. Hierbij kan men optioneel een `std::pair<CEdge *, CEdge *>` pointer meegeven als `edgeMapping` parameter. In deze `edgeMapping` moet



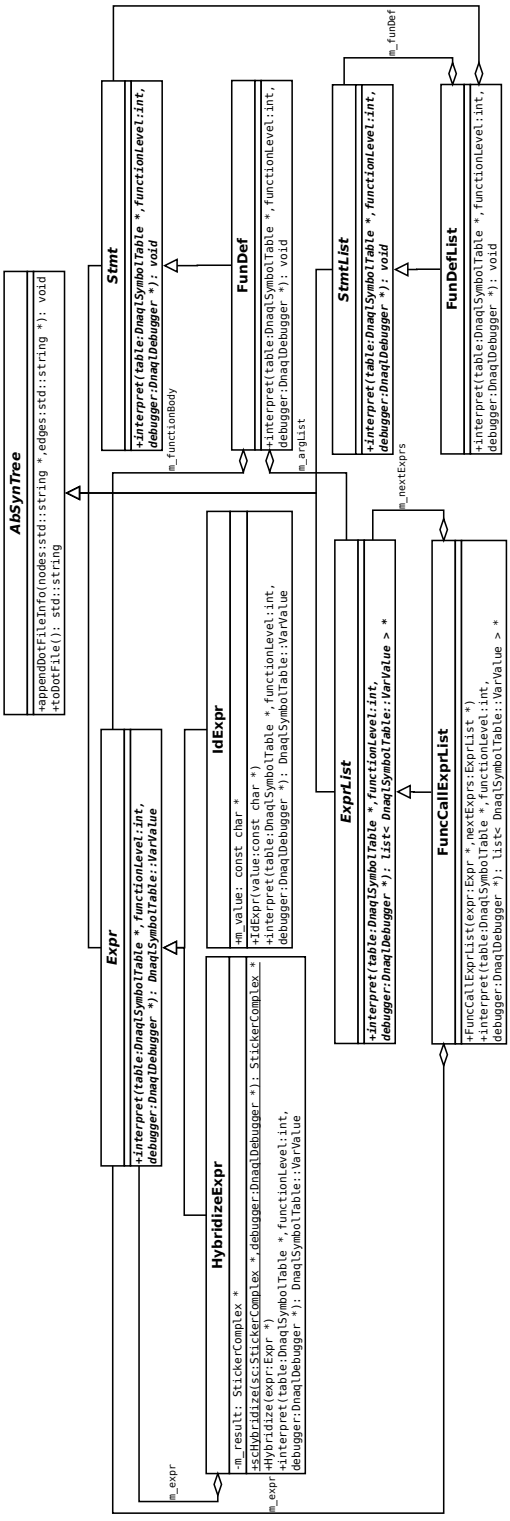
Figuur 4.4: UML weergave van de klassen `CComponent` en `CEdge`

men het eerste element invullen met een adres van een `CEdge` object uit het oorspronkelijke `CComponent` object, en het tweede element initialiseren op nul. Na het uitvoeren van de `copy` functie, zal men in het tweede element van de `edgeMapping` het adres van het corresponderende `CEdge` object in de nieuwe component vinden. Deze functie is handig indien men met dezelfde boog wil verder werken, maar dan op een nieuwe kopie (zoals bijvoorbeeld bij hybridisatie).

4.3.4 De klassen voor de abstracte syntaxboom

De datastructuren voor het opbouwen van de abstracte syntaxboom en het uiteindelijk interpreteren van een DNAQL programma werden opgesteld volgens het 'Interpreter' design pattern [GHJV09]. Er werd hierbij echter een onderscheid gemaakt tussen expressies, statements, expressielijsten en statementlijsten. In figuur 4.5 ziet u een beknopte weergave van de abstracte klassen en enkele afgeleide klassen.

Met behulp van de code gegenereerd door de programma's *Flex* en *Bison*, wordt er met deze datastructuren een abstracte syntaxboom opgebouwd. Voor elke regel in de grammatica van DNAQL (zie sectie 3.7.1) is er een overeenkomstige datastructuur waarvan de `interpret` functie de semantiek van die regel implementeert. Verschillende afgeleide `Expr` klassen hebben een extra membervariabele `m_result`. In deze gevallen (vooral bij expressie-klassen die operaties voorstellen) wordt er een kopie genomen van het input sticker complex en wordt

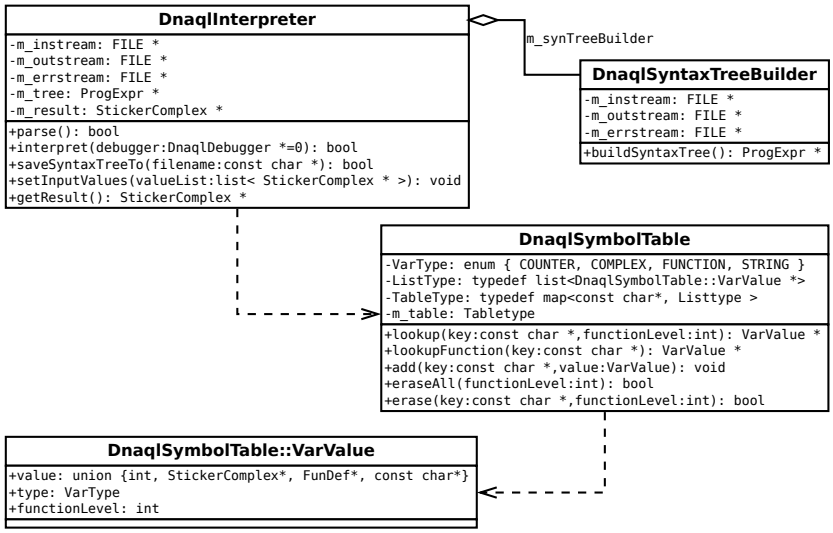


Figuur 4.5: UML weergave van de klassen voor de abstracte syntaxboom

de expressie op deze kopie uitgevoerd. Het resultaat daarvan wordt dan opgeslagen in de variabele `m_result`. De operaties die we besproken hebben in sectie 3.6, hebben elk een implementatie in een statische functie in een overeenkomstige expressie klasse. Op deze manier is de operatie minder afhankelijk van zijn abstracte syntaxboom klasse en kan hij gemakkelijker afzonderlijk gebruikt en getest worden.

4.3.5 DnaqlInterpreter en DnaqlSyntaxTreeBuilder

De interface tussen de abstracte syntaxboom en de andere klassen wordt gevormd door de klasse `DnaqlInterpreter` (zie figuur 4.6). De klasse zal de syntaxboom laten opbouwen door een `DnaqlSyntaxTreeBuilder` object. Met de functie `saveSyntaxTreeTo` kan de syntaxboom eventueel worden opgeslagen naar een bestand in het *Graphviz dot*-formaat. Na het parsen en interpreteren van een DNAQL programma, kan men aan deze klasse het resulterende sticker complex opvragen.



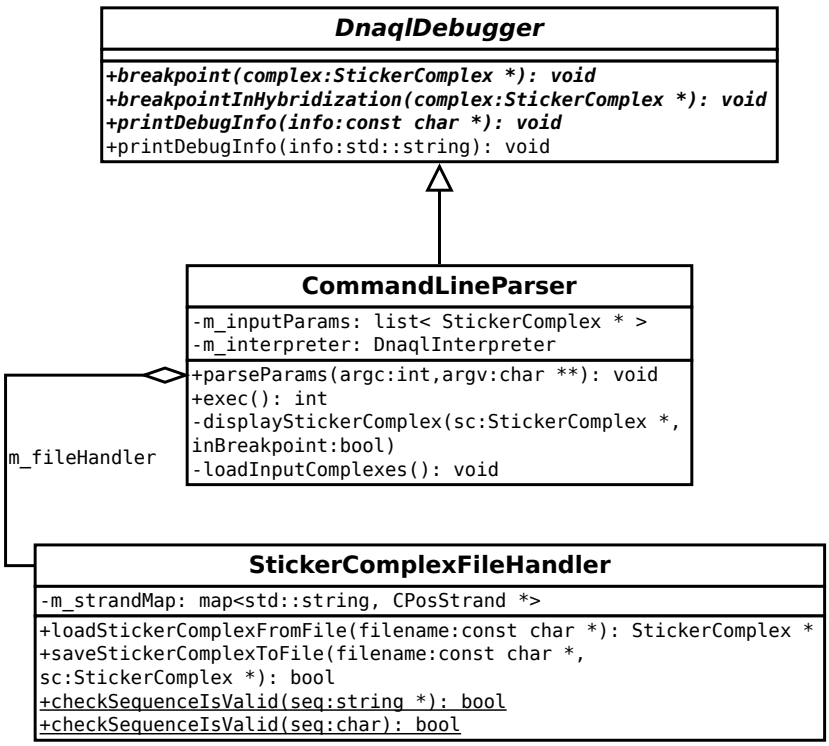
Figuur 4.6: UML weergave van de DnaqlInterpreter klasse

`DnaqlInterpreter` maakt bij het interpreteren gebruik van de klasse `DnaqlSymbolTable` voor het bijhouden van de verschillende variabelen (identifiers) en hun overeenkomstige waarde, in de vorm van de struct `VarValue`. Het type van een `VarValue` is `COUNTER` in het geval van een teller-variabelen, `COMPLEX` in het geval van een complex-variabele, `FUNCTION` bij een functienaam en `STRING` indien er een variabele naam wordt teruggegeven. Het `STRING` type wordt vooral gebruikt bij de parameterdefinities van functies in de klasse `ArgList`. De datastructuur `DnaqlSymbolTable` bevat een map die een variabele naam afbeeldt op een lijst van `VarValue`'s. Dit is een lijst omdat we werken met functies en omdat eenzelfde variabele naam in meerdere functies kan voorkomen. Met behulp van de variabele `functionLevel` in de `VarValue` struct kan een functie enkel de variabele op zijn 'level' gebruiken. Deze waarde wordt ook telkens meegege-

ven via de `interpret` functies (zie figuur 4.5). Bij de aanroep van een functie wordt dit level met één verhoogd (in de klasse `FuncCallExpr`) zodat we als het ware een nieuwe 'laag' op onze symbol table creëren. Als de functie afgelopen is, wordt deze 'laag' met al zijn variabelen terug verwijderd met behulp van de functie `eraseAll`. Voor het opzoeken van 'gewone' variabelen (op een bepaald function level) wordt de functie `lookup` gebruikt. Voor het opzoeken van een functiedefinitie (nodig bij het uitvoeren van een function call), is de functie `lookupFunction` aangewezen.

4.3.6 CommandLineParser en StickerComplexFileHandler

De interactie met de gebruiker gebeurt voor een groot deel via de command line. De `CommandLineParser` klasse is verantwoordelijk voor het verzorgen van deze communicatie en de gevraagde acties uit te voeren. Omdat deze klasse de communicatie met de gebruiker stuurt, erft ze over van de abstracte interface klasse `DnaqlDebugger` om zo debug informatie aan de gebruiker te kunnen geven. De `exec` functie van de `CommandLineParser` klasse zal in de `main` functie worden aangeroepen. Een schematische weergave ziet u in figuur 4.7.



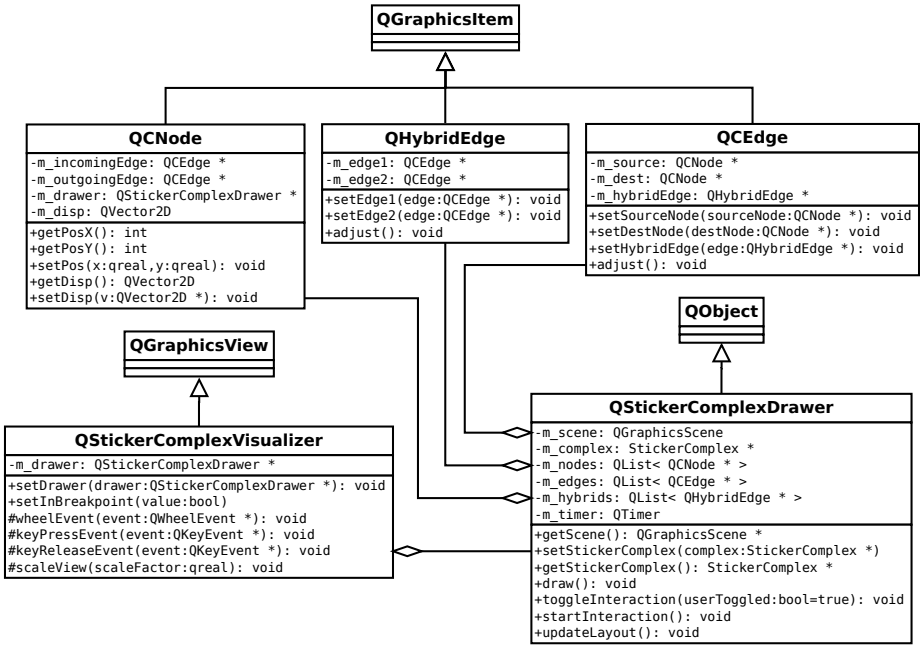
Figuur 4.7: UML weergave van de CommandLineParser klasse

Voor het inladen van sticker complexen uit een bestand en voor het opslaan van het berekende resultaat, maakt de `CommandLineParser` klasse gebruik van de `StickerComplexFileHandler` klasse. Deze klasse heeft ook twee publieke

statische functies die controleren of een string een geldige sequentie van symbolen over Σ is. Dit gebeurt met behulp van de alfabetten gedefinieerd in de `StickerComplex` klasse (zie sectie 4.3.1). De variabele `m_strandMap` wordt gebruikt om de namen van strengen uit het bestand te koppelen aan `CPosStrand` objecten, zodat deze bij het inladen van de stickers gemakkelijk terug te vinden zijn (zie ook sectie 4.2).

4.3.7 De visualisatie klassen

De klassen die gebruikt worden voor het visualiseren van een sticker complex kan u zien in figuur 4.8. Hierbij zijn `QCNode`, `QCEdge` en `QHybridEdge` `QGraphicsItem` klassen die op een bepaalde positie in een `QGraphicsScene` getekend kunnen worden. Ze representeren respectievelijk een knoop, een boog en een (gestippelde) hybridisatie boog. De klasse `QStickerComplexVisualizer` is een `QGraphicsView` die de 'scene' uittekent op het scherm en eventuele acties van de gebruiker opvangt. Het aanmaken en in de scène plaatsen van alle knoop- en boog-objecten op basis van een gegeven `StickerComplex` object gebeurt door de klasse `QStickerComplexDrawer`. Deze berekent via een *force directed graph drawing* algoritme de optimale positie voor elke knoop en boog.



Figuur 4.8: UML weergave van de klassen voor de visualisatie

Omdat alleen de positie van de `QCNode` objecten kan veranderen, ofwel door het teken-algoritme ofwel door de gebruiker (door ze te verslepen), bevatten zij pointers naar hun aangrenzende `QCEdge` objecten. Op deze manier kunnen zij bij het veranderen van positie, via de functie `adjust` de positie van hun bogen aanpassen, die op hun beurt ook weer de positie van eventuele `QHybridEdge` objecten aanpassen.

4.3.8 De Lex en Yacc bestanden

De bestanden *dnaql.lex* en *dnaql.y* implementeren de DNAQL grammatica zoals die gegeven is in sectie 3.7.1. Ook de regels in verband met variabelen en functienamen, de verschillende alfabetten en tokens zijn in deze bestanden opgenomen. Omdat er in de grammatica ook negatieve strengen als constanten aanwezig zijn, worden deze aangegeven door ze vooraf te laten gaan met het symbool ˆ gevolgd door een sequentie van symbolen (waarbij de volgorde van de symbolen tegengesteld is aan de oriëntatie van de sticker). De alfabetten Λ, Ω en Θ zijn voor DNAQL programma's hetzelfde als in het gedefinieerde bestandsformaat (zie sectie 4.2). Hierdoor kunnen we constanten van variabelenamen onderscheiden omdat deze laatsten steeds met een kleine letter moeten beginnen. Deze bestanden bevatten bovendien ook de nodige code voor het opbouwen van de abstracte syntaxboom. Deze bestanden worden door de programma's *Flex* en *Bison* omgezet naar een `yyparse` functie die door de klasse `DnaqlSyntaxTreeBuilder` gebruikt wordt voor het omzetten van een DNAQL bestand naar een abstracte syntaxboom.

4.4 Algoritmen

In deze sectie wil ik de belangrijkste algoritmen die ik geïmplementeerd heb, bespreken. Het zijn vooral algoritmen die te maken hebben met de operaties van het sticker complex datamodel. Bij het lezen van de algoritmen houdt u best de sectie over de datastructuren bij de hand, omdat ik soms naar specifieke functies of membervariabele van een klasse verwijs. De verschillende algoritmen worden telkens beschreven in een pseudo-code die dicht aanleunt bij de programmeertalen C++ en Java. Omdat de meeste informatici vertrouwd zijn met (één van) deze talen, zou het begrijpen van deze pseudo-code geen moeilijkheid mogen vormen.

4.4.1 Bepaling van de componenten

Een van de belangrijkste algoritmen is het algoritme voor het bepalen van de afzonderlijke componenten. Dit algoritme is geïmplementeerd in de functie `updateComponents` in de `StickerComplex` klasse. Deze functie gaat tijdens het bepalen van de componenten, ook de hybridize array's invullen. Een beschrijving in pseudo-code ziet u in listing 4.2 en listing 4.3.

Aan de start van het algoritme wordt voor elke streng de component variabele op nul gezet en worden de componentlijst en de hybridize array's leeggemaakt. Hierna overlopen we eerst al de positieve strengen. Voor elk positief streng waarvan de component variabele nul is, maken we een nieuw `CComponent` object aan en roepen we de functie `addStrandToComponent` op. Na de positieve strengen overlopen we al de stickers van het sticker complex. Voor elke sticker waarvan de component variabele nul is, maken we ook een nieuw `CComponent` object aan en geven we de sticker als `startStrand` mee.

```
void StickerComplex::updateComponents() {
    resetComponents();
    foreach(strand in m_strands) {
        if(strand.getComponent() == 0) {
            CComponent comp = new CComponent();
            comp.setStartStrand(strand);
            m_components.push_back(comp);
            addStrandToComponent(strand, comp);
        }
    }
    foreach(sticker in m_sticker) {
        if(sticker.getComponent() == 0) {
            CComponent comp = new CComponent();
            comp.setStartStrand(sticker);
            m_components.push_back(comp);
            addStrandToComponent(sticker, comp);
        }
    }
}
```

Listing 4.2: De functie `updateComponents`

```
void StickerComplex::addStrandToComponent(CStrand strand,
      CComponent comp) {
    if(strand != 0 && strand.getComponent() == 0) {
        strand.setComponent(comp);
        foreach(edge in strand.edges() ) {
            if(edge.getHybridizedEdge() != 0)
                addStrandToComponent(
                    edge.getHybridizedEdge().getStrand(),comp);
            else if(not edge.isBlocked())
                addEdgeToHybridizeArray( edge );

            if(edge.isImmob())
                comp.setHasImmobEdge( true );
        }
    }
}
```

Listing 4.3: De functie `addStrandToComponent`

De functie `addStrandToComponent` zal alle bogen van een streng overlopen. Indien een boog een gehybridiseerde boog heeft, wordt de streng van deze gehybridiseerde boog ook aan de component toegevoegd met een recursieve oproep van `addStrandToComponent`.

4.4.2 Isomorfie bij componenten en minimisatie

Om de operatie difference en het minimisatie algoritme te kunnen implementeren, moest ik eerst een algoritme hebben om te controleren of twee componenten isomorf zijn. Dit algoritme werd mij aangereikt door mijn begeleider. Het isomorfie algoritme werd geïmplementeerd als de functie `isIsomorfphTo` in de klasse `CComponent`. Een beschrijving in pseudo-code vindt u in listing 4.4 en 4.5.

```
bool CComponent::isIsomorphTo(CComponent comp) {
   CEdge startEdge = m_startStrand.getStartEdge();
   //haal alle mogelijke passende 'start' bogen op
   CLabel label = startEdge.getLabel()
   list<CEdge> posMatch = comp.getEdgesWithLabel(label);

   //controleer isomorfisme bij elke mogelijke startboog
   map<CEdge,CEdge> checked;
   foreach(edge in posMatch) {
      //maak de 'bezochte bogen' leeg
      checked.clear();

      //controleer op isomorfisme
      if(verifyIsomorf(startEdge, edge, checked))
         return true;
   }
   //er werd bij geen enkele boog isomorfisme gevonden
   return false;
}
```

Listing 4.4: De functie `isIsomorphTo`

De intuïtie achter dit algoritme is dat het probeert om de twee componenten op zo een manier op elkaar te leggen, dat alle bogen exact overeenkomen. In een eerste stap moeten we dus alle bogen uit de tweede component ophalen, die mogelijk kunnen matchen met de startboog in de eerste component. Voor al deze mogelijke *'matchings'* controleren we of we de twee componenten op elkaar kunnen afbeelden vertrekkende van deze twee bogen. Dit gebeurt in de functie `verifyIsomorph`.

De functie `verifyIsomorph` is een recursieve functie die controleert of twee bogen isomorf zijn (indien ze beiden nog niet bezocht zijn) en of dat hun buren en eventueel gehybridiseerde bogen isomorf zijn. Zo worden uiteindelijk alle bogen van beide componenten gecontroleerd. Indien alle bogen van de ene component een equivalente boog in de andere component gevonden hebben, zijn de componenten isomorf en zal de functie `true` teruggeven.

Het algoritme voor minimisatie werd ondergebracht in de functie `minimize` van de klasse `StickerComplex`. Deze zal voor alle componenten c_1 in het sticker

```
bool CComponent::verifyIsomorph( CEdge e1, CEdge e2,
                                 map<CEdge,CEdge> checked){
   //lege bogen zijn triviaal isomorf
   if(e1 == 0 and e2 == 0)
      return true;
   //beide bogen zijn niet leeg en alle twee reeds bezocht
   else if(e1 != 0 and e2 != 0
         and checked.contains(e1)
         and checked.contains(e2)
         and checked.find(e1) == e2
         and checked.find(e2) == e1)
            return true;
   //de bogen zijn nog niet bezocht maar wel isomorf
   else if(e1 != 0 and e2 != 0
         and not checked.contains(e1)
         and not checked.contains(e2)
         and e1.getLabel() == e2.getLabel()
         and e1.isBlocked() == e2.isBlocked()
         and e1.isImmob() == e2.isImmob()) {
            //controleer de buren op isomorfisme
            checked.insert(e1, e2);
            checked.insert(e2, e1);
            return   verifyIsomorph(e1.getPreviousEdge(),
                                    e2.getPreviousEdge(),
                                    checked) and
                     verifyIsomorph(e1.getNextEdge(),
                                    e2.getNextEdge(),
                                    checked) and
                     verifyIsomorph(e1.getHybridizedEdge(),
                                    e2.getHybridizedEdge(),
                                    checked);
   } else
      return false;
}
```

Listing 4.5: De functie `verifyIsomorph`

complex, c_1 vergelijken met alle in de lijst volgende componenten c_2 en controleren op isomorfie met het bovenstaand algoritme. Indien c_1 isomorf is aan c_2, dan verwijdert het de component c_2 van het sticker complex. Na het uitvoeren van de operatie `minimize` zullen er geen redundante componenten in het sticker complex meer overblijven.

4.4.3 Kopiëren van een sticker complex

Het volledig kopiëren van een sticker complex lijkt in eerste instantie niet zo eenvoudig omdat men immers ook al de connecties tussen de verschillende bogen moet kopiëren zodat deze naar de juiste nieuwe gekopieerde bogen wijzen. Het algoritme, dat geïmplementeerd is in de functie `copy` van de `StickerComplex` klasse, berust echter volledig op de kopieer-functionaliteit van zijn componenten. Het algoritme zal dus een nieuw sticker complex aanmaken, en dan een kopie van elk van zijn componenten aan dit nieuwe sticker complex toevoegen.

Het kopiëren van een component is iets ingewikkelder. Het algoritme zal, beginnende vanaf de startstreng, elk streng kopiëren en daarna de bogen van het originele streng overlopen. Indien één van de bogen gehybridiseerd is aan een andere boog, dan zal de streng van deze gehybridiseerde boog ook (recursief) gekopieerd worden. Om dit te realiseren is het nodig om een mapping bij te houden tussen de originele strengen en de nieuwe strengen. Zo kunnen we aan de nieuwe streng de juiste nieuwe boog opvragen zodat we hierbij de juiste hybridisatie kunnen instellen. Het algoritme is beschreven in psuedo-code in listing 4.6.

De `copy` functie van de klasse `CComponent` maakt gebruik van de `copy` functie van de klasse `CStrand`. Het kopiëren van een streng is vrij rechttoe rechtaan. Er wordt vertrokken van een leeg streng en hieraan worden dan kopieën van alle bogen aan toegevoegd (in dezelfde volgorde als deze voorkomen in het originele streng). In het geval van circulaire strengen, stoppen we als we de start-boog een tweede maal tegen komen.

4.4.4 Bepalen van de eindigheid van een hybridisatie

Het algoritme om te bepalen of een hybridisatie eindig is, gebeurt op gelijkaardige wijze zoals beschreven is in sectie 3.6.3. Het verschil is echter dat ik in mijn implementatie geen koppels van bogen uit μ' bijhoud, maar enkel de bogen waar een hybridisatie heeft plaatsgevonden. Dus telkens enkel het eerste element van elk koppel. De pseudo-code voor dit algoritme vindt u in listing 4.7 en listing 4.8.

Om er zeker van te zijn dat er nergens een cyclus mogelijk is, gaat het algoritme voor elke negatieve (sticker) boog controleren of er van daaruit een hybridisatie cyclus kan gevormd worden. Als zo een cyclus gevonden werd, dan stopt het algoritme en geeft het aan dat de hybridisatie niet zal stoppen. Indien geen cyclus gevonden werd, gaat hij verder met een volgende negatieve boog totdat alle bogen onderzocht zijn. Als het algoritme bij geen enkele negatieve boog een

```
CComponent CComponent::copy(){
    CComponent output = new CComponent();
    map<CStrand, CStrand> checked;
    CStrand startStrandCopy = copyStrand(m_startStrand, output,
                                         checked);
    output.setStartStrand(startStrandCopy);
    return output;
}

CStrand CComponent::copyStrand(CStrand strand, CComponent comp,
                               map<CStrand, CStrand> checked){
    if( strand != 0 ) {
        if( not checked.contains(strand) ) {
            CStrand newStrand = strand.copy();
            //add strand to map
            checked.insert(strand, newStrand);

            newStrand.setComponent( comp );
            CEdge edge = strand.getStartEdge();
            CEdge edgeN = newStrand.getStartEdge();
            CEdge temp = 0;

            while(edge != 0 and temp != strand.getStartEdge()){
                if(edge.getHybridizedEdge() != 0) {
                    CStrand hybStrand = edge.getHybridizedEdge().
                                              getStrand();
                    CStrand hybStrCopy
                        = copyStrand(hybStrand, comp, checked);
                    int index = hybStrand.getIndex(
                                edge.getHybridizedEdge());
                    CEdge hEdgeCopy = hybStrCopy.getEdge(index);
                    edgeN.setHybridizedEdge( hEdgeCopy );
                }
                if(edgeN.isImmob())
                    comp.setHasImmobEdge( true );
                edge = edge.getNextEdge();
                edgeN = edgeN.getNextEdge();
                temp = edge;
            }
            return newStrand;
        } else {
            return checked.value(strand);
        }
    } else
        return 0;
}
```

Listing 4.6: Het kopiëren van een component

```
bool HybridizeExpr::hybridizationIsTerminating(StickerComplex sc){
    list<CSticker> stickerList = sc.getStickers();
    bool cyclePresent = false;
    set<CEdge> usedEdges;
    foreach(sticker in stickerList){
        usedEdges.clear();
        cyclePresent = lookForHybridizationCycle(
                            sticker.getEdge1(), sc, usedEdges);
        if(not cyclePresent) {
            usedEdges.clear();
            cyclePresent = lookForHybridizationCycle(
                            sticker.getEdge2(), sc, usedEdges);
        }
    }
    return not cyclePresent;
}
```

Listing 4.7: De functie `hybridizationIsTerminating`

cyclus kan vinden, dan kunnen we besluiten dat de hybridisatie eindig is. Dit gebeurt in de functie `hybridizationIsTerminating`.

Voor het controleren of er vanaf een bepaalde boog een hybridisatie cyclus kan ontstaan, gebruikt het algoritme de recursieve functie `lookForHybridization-Cycle`. Deze functie geeft de waarde `false` terug indien de aangeboden boog niet open is, of de component van die boog een geïmmobiliseerde boog bevat. Bogen van een geïmmobiliseerde component kunnen immers nooit meerdere malen voorkomen en onmogelijk een cyclus geven. Vervolgens controleert het algoritme of we de huidige boog al eens gebruikt hebben voor hybridisatie. Indien dit het geval is, kunnen we spreken van een cyclus en stoppen we de functie met de waarde `true`. Merk op dat dit zelfs niet de originele start-boog hoeft te zijn en dat er in dit geval een sub-cyclus wordt aangegeven. Indien we de boog nog niet gebruikt hebben voor hybridisatie, dan voegen we hem toe aan de 'gebruikte bogen'-lijst (`usedEdges`) en veronderstellen we dat hij gehybridiseerd is. Vervolgens halen we alle mogelijke bogen op waarmee onze huidige boog zou kunnen hybridiseren om daarna voor elke van deze bogen, alle andere open bogen in hun component toe te voegen aan onze todo-lijst. De todo-lijst is een lijst van bogen waarlangs we mogelijk een cyclus kunnen vinden. De andere bogen van de component worden opgehaald via de functie `getEdges` waarbij de parameter aangeeft welke boog moet worden uitgesloten in de teruggegeven lijst.

De volgende stap bij het zoeken naar een hybridisatie cyclus is het leegmaken van de todo-lijst en zo alle mogelijke paden te doorzoeken. Hiervoor nemen we telkens de eerste boog uit de lijst en onderzoeken we recursief of er een cyclus mogelijk is. Indien er een cyclus gevonden werd, stoppen we de functie met de waarde `true`. Anders nemen we de volgende boog uit de todo-lijst en proberen we hier opnieuw. Als de todo-lijst uiteindelijk leeg is en er geen cyclus gevonden werd, dan stoppen we de functie met de waarde `false`.

```
bool HybridizeExpr::lookForHybridizationCycle
        (CEdge edge, StickerComplex sc, set<CEdge> usedEdges){
   /* we negeren bogen van geimmobiliseerde componenten en
    * bogen die geblokkeerd zijn */
   if(edge != 0 and
      not edge.isBlocked() and
      edge.getHybridizedEdge() == 0 and
      not edge.getComponent().hasImmobEdge()){

      if(usedEdges.contains(edge))
         return true;
      else{
         usedEdges.insert(edge);
         list<CEdge> todoList;
         list<CEdge> possibleMatchings
               = sc.getPossibleMatchings(edge);
         foreach(matchEdge in possibleMatchings){
            CComponent currentComp = matchEdge.getComponent();
            //haal alle andere bogen van currentComp op
            list<CEdge> edgesList
                  = currentComp.getEdges(matchEdge);
            //voeg ze toe aan de lijst
            todoList.push_back(edgeList);
         }
         bool foundCycle = false;
         while( not todoList.empty() and not foundCycle ) {
            CEdge e = todoList.front();
            todoList.pop_front();
            foundCycle = lookForHybridizationCycle(e, sc,
                                                   usedEdges);
         }
         usedEdges.erase(edge);
         return foundCycle;
      }
   } else
      return false;
}
```

Listing 4.8: De functie `lookForHybridizationCycle`

4.4.5 Hybridisatie algoritme

Voor het implementeren van de hybridisatie operatie zelf, heb ik gekozen voor een *brute force* aanpak. Het voordeel van deze aanpak is dat het algoritme eenvoudig te begrijpen is en makkelijk implementeerbaar is met mijn datastructuren. Het nadeel is dat het algoritme niet in staat is efficiënt om te gaan met het exponentiële karakter van hybridisatie, maar hierover later meer. In mijn implementatie is dit algoritme gerealiseerd in de statische functies `scHybridize`, `doAlternativeHybridization` en `doHybridizationStep` in de `HybridizeExpr` klasse. Deze functies zijn in pseudo-code beschreven in de listings 4.9, 4.10 en 4.11.

```
StickerComplex HybridizeExpr::scHybridize(StickerComplex sc){
    StickerComplex output = sc.copy();
    //houd per sticker boog bij welke bogen we al gebruikt hebben
    map<CEdge *, set<CEdge *>> usedEdges;
    bool foundOne = true;
    /* Voor elke mogelijke open sticker boog (in het originele
     * complex), 'plak' hem op elke mogelijke 'matching' */
    while(foundOne){
        list<CSticker> stickerList = output.getStickers();
        foundOne = false;
        foreach(sticker in stickerList){
            if( doHybridizationStep( sticker.getEdge1(), output,
                                                    usedEdges) )
                foundOne = true;
            if( doHybridizationStep( sticker.getEdge2(), output,
                                                    usedEdges) )
                foundOne = true;
        }
        if(foundOne)
            output.minimize();
    }
    output.cleanupMarkedComponents();
    output.minimize();
    return output;
}
```

Listing 4.9: De functie `scHybridize`

Het algoritme werkt als volgt: Voor elke sticker in het sticker complex, kijken we of één van zijn bogen open is. Voor elke open sticker-boog halen we alle mogelijke (positieve) streng-bogen op waarmee deze sticker-boog zou kunnen hybridiseren (via de functie `getPossibleMatchings`). Vervolgens overlopen we deze lijst van mogelijke 'matchende'-bogen. Veronderstel dat onze sticker-boog s zich in component c_1 bevindt en onze huidige streng-boog e in component c_2. We controleren dan eerst of c_1 en c_2 niet beiden geïmmobiliseerd zijn (tenzij $c_1 = c_2$). Twee (verschillende) geïmmobiliseerde componenten kunnen immers niet aan elkaar hybridiseren. Indien dat het geval is, dan voegen we de boog e toe aan de lijst van gebruikte bogen van s zodat we vermijden dat we in een latere iteratie niet hetzelfde opnieuw doen. Daarna maken we een kopie van c_1 en c_2 waarbij deze aan elkaar gehybridiseerd zijn via de bogen s en e en markeren we de originele componenten c_1 en c_2 voor verwijdering. In het uiteindelijke sticker complex zullen zij immers enkel gehybridiseerd kunnen

```
void HybridizeExpr::doAlternativeHybridization(CEdge edge,
      CComponent strandComp, CEdge stickerEdge,
      StickerComplex output){
   //maak een mapping zodat we de juiste nieuw boog krijgen
   pair<CEdge, CEdge> strandEdgeMapping;
   strandEdgeMapping.first = edge;
   strandEdgeMapping.second = 0;

   //maak een extra kopie van de component
   CComponent newStrandComp
                        = strandComp.copy(strandEdgeMapping);
   newStrandComp.setToBeRemoved(true);
   CEdge newEdge = strandEdgeMapping.second;

   //stel de hybridisatie tijdelijk in
   newEdge.setHybridizedEdge(stickerEdge);
   stickerEdge.setHybridizedEdge(newEdge);

   //Maak een kopie van de component (met de hybridisatie)
   CComponent newComp2 = stickerComp.copy();

   //Maak de hybridisatie in de originele bogen ongedaan
   stickerEdge->setHybridizedEdge(0);
   strandEdgeMapping.second->setHybridizedEdge(0);

   //voeg de componenten toe
   output.addComponent( newComp );
   output.addComponent( newStrandComp );
}
```

Listing 4.10: De functie `doAlternativeHybridization`

voorkomen. De nieuwe kopie wordt dan toegevoegd aan het sticker complex. Er is echter nog het speciale geval als $c_1 = c_2$. In dat geval kan de component zowel op zichzelf hybridiseren als op een exacte kopie. Deze laatste situatie moet dus nog extra worden toegevoegd aan het sticker complex. Hiervoor maken we in de functie `doAlternativeHybridization` een extra kopie van de component waarbij we de juiste nieuwe boog ophalen die overeenkomt met onze originele boog. Hiervan maken we nog eens een extra kopie waarin de twee bogen aan elkaar gehybridiseerd zijn en voegen deze toe aan het sticker complex. Deze laatste extra kopie is nodig zodat we niet opnieuw alle componenten moeten berekenen.

Hierna gaan we verder met de volgende open sticker-boog. Nadat we op deze manier alle sticker-bogen overlopen hebben, kijken we of er een component werd toegevoegd. Indien dit het geval is, minimaliseren we het tussentijds sticker complex en beginnen we terug opnieuw door alle stickers in de (nieuwe) stickerlijst te overlopen. Dit blijven we herhalen tot er geen componenten meer werden toegevoegd en de MHE dus verzadigd is. Daarna verwijderen we al de componenten die gemarkeerd zijn voor verwijdering en minimaliseren we het bekomen sticker complex. Dit is dan ook de output voor ons algoritme.

Een probleem bij dit algoritme, en bij hybridisatie in het algemeen, is dat het niet stopt indien de hybridisatie *recursief* is. Indien er oneindig veel niet-equivalent MHE's bestaan, zal dit algoritme ook proberen om ze allemaal te

```
bool HybridizeExpr::doHybridizationStep(CEdge stickerEdge,
      StickerComplex output, map<CEdge *, set<CEdge *>> usedEdges){
   bool foundOne = false;
   if(stickerEdge != 0 and
      not stickerEdge.isBlocked() and
      stickerEdge.getHybridizedEdge() == 0){

      list<CEdge> possibleMatchings
                        = output.getPossibleMatchings(stickerEdge);
      list<CEdge> usedEdgesList = usedEdges.find(stickerEdge);
      CComponent stickerComp = stickerEdge.getComponent();
      foreach(edge in possibleMatchings){
         if( not usedEdgesList.contains(edge) ) {
            CComponent strandComp = edge.getComponent();
            //twee geimmobiliseerde componenten mag niet
            if(strandComp == stickerComp or
               not strandComp.hasImmobEdge() or
               not stickerComp.hasImmobEdge() ) {
               //insert edge bij de gebruikte bogen
               usedEdgesList.insert(edge);
               foundOne = true;

               //markeer de component voor verwijdering
               stickerComp.setToBeRemoved(true);
               strandComp.setToBeRemoved(true);

               //stel de hybridisatie tijdelijk in
               stickerEdge.setHybridizedEdge(edge);
               edge.setHybridizedEdge(stickerEdge);

               //Maak een kopie van de component (met hybridisatie)
               CComponent newComp = stickerComp.copy();

               //Maak de hybridisatie in de originele bogen ongedaan
               stickerEdge.setHybridizedEdge(0);
               edge.setHybridizedEdge(0);

               //voeg de component toe
               output.addComponent(newComp);

               /* Als de componenten gelijk zijn,
                * doen we ook de alternatieve hybridisatie.*/
               if(strandComp == stickerComp and
                  not strandComp.hasImmobEdge()) {
                  doAlternativeHybridization(edge, strandComp,
                                             stickerEdge, output);
               }
            }
         }
      }
   }
   return foundOne;
}
```

Listing 4.11: De functie `doHybridizationStep`

genereren en dus nooit stoppen. Voor het aanroepen van dit algoritme (via de functie `scHybridize`), wordt er echter steeds gecontroleerd of de hybridisatie eindig is en dit met behulp van het algoritme uit sectie 4.4.4. Als dat algoritme aangeeft dat de hybridisatie wel degelijk eindig is, dan pas wordt het echte hybridisatie algoritme uitgevoerd om zo te voorkomen dat het programma oneindig lang blijft doorgaan.

Is dit algoritme nu correct? Hiervoor nemen we even de definitie van de operatie `hybridize` uit sectie 3.6.3 erbij:

> In het geval dat er slechts een eindig aantal MHE's van C bestaan, definiëren we `hybridize`(C) als de unie van al de verzadigde MHE's van C.

We gaan nu even na of het hierboven beschreven algoritme voldoet aan deze definitie. Bij het aanvangen van het algoritme weten we dat er slechts een eindig aantal MHE's bestaan omdat dit gecontroleerd werd door de functie `hybridizationIsTerminating`. Hierdoor weten we ook dat het algoritme zal stoppen. Bovendien zijn al de MHE's op het einde van het algoritme verzadigd, omdat het algoritme pas stopt als er geen nieuwe hybridisatie meer kon plaatsvinden. Op dat moment kunnen er dus geen grotere, nieuwe componenten geconstrueerd worden.

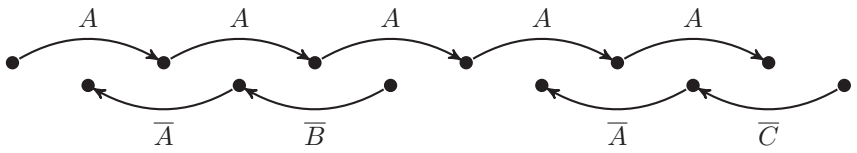
Een minder triviaal aspect van de definitie is dat we *al* de verzadigde MHE's moeten geconstrueerd hebben. We weten intussen al dat de MHE's die we bekomen verzadigd zijn, maar nog niet dat we ze *allemaal* hebben. We bekijken dit eerst op het niveau van de "gewone" hybridisatie extensies met maximale matching. Deze worden correct geconstrueerd omdat we een gegeven boog laten hybridiseren met elke mogelijke complementaire boog en dit ook met eventuele nieuwe componenten in latere iteraties.

Nu we weten dat onze hybridisatie correct verloopt, kunnen we vanuit het ongerijmde aantonen dat we ook alle mogelijke MHE's bekomen. Zij $C = \big(V,\ E,\ \lambda,\ \mu,\ \mathit{immob},\ \mathit{blocked}\big)$ een sticker complex en stel dat `hybridize`(C) een verzadigde MHE m oplevert die niet door het algoritme berekend kan worden. In dat geval moet m een (gewone) hybridisatie extensie met maximale matching zijn van een redundante extensie C'. Omdat we weten dat de gewone hybridisatie extensies door het algoritme correct berekend worden, moet C' een redundante extensie zijn die niet door het algoritme kan gesimuleerd worden. Er zouden dus componenten in C moeten zijn die niet als redundante kopieën zouden kunnen worden toegevoegd. Het is echter onmogelijk dat er een redundante extensie C' bestaat die niet door het algoritme kan worden gesimuleerd. Dit enerzijds omdat er bij elke hybridisatiestap een kopie wordt genomen van de te hybridiseren componenten en de originelen behouden blijven; en anderzijds omdat we, indien een component op zichzelf hybridiseert, ook telkens het geval beschouwen dat die component op een identieke kopie van zichzelf zou hybridiseren. Bijgevolg kan de MHE m niet bestaan en moet het algoritme alle mogelijke MHE's genereren.

Uit de voorgaande redeneringen kunnen we besluiten dat het algoritme wel degelijk correct werkt. Het is echter niet het meest efficiënte algoritme om een hybridisatie te berekenen. Omdat het algoritme een sticker op alle mogelijke

plaatsen plakt, ontstaan er in elke hybridisatiestap veel overbodige en isomorfe componenten. De isomorfe componenten worden tijdens het algoritme wel weggewerkt met een minimalisatie, maar toch is dit een belangrijke vertragende factor. Bovendien houdt het algoritme alle berekende tussenresultaten bij zodat ook deze bij een volgende iteratie nog zouden kunnen hybridiseren. Deze tussenresultaten worden enkel op het einde van het algoritme verwijderd en niet tijdens omdat het zeer moeilijk te bepalen is wanneer een bepaalde (tussenresultaats-) component niet meer gebruikt zal worden in de hybridisatie.

Hybridisatie op zich is al exponentieel, wat het maken van een efficiënt algoritme nog bemoeilijkt. We bekijken hiervoor even het voorbeeld in figuur 4.9. Zoals u kan nagaan, zal de hybridisatie van dit sticker complex uiteindelijk resulteren in 32 ($= 2^5$) verschillende componenten. Bij het uitvoeren van het algoritme op dit voorbeeld, zullen er vlak voor het verwijderen van de gemarkeerde componenten en de minimalisatie 258 componenten in het sticker complex aanwezig zijn waarbij het maximale aantal componenten tijdens het uitvoeren 916 was[4].



Figuur 4.9: Voorbeeld van een sticker complex met een eindige, exponentiële hybridisatie

4.4.6 De andere operaties

De algoritmes achter de andere operaties van het sticker complex datamodel zijn veel eenvoudiger dan die van de `hybridize` operatie. Hun bespreking hoeft dan ook niet zo uitvoerig te gebeuren. Al de operaties werden geïmplementeerd als statische functie binnen hun overeenkomstige klasse van de abstracte syntaxboom datastructuur. Op het einde van elke operatie wordt ook een minimalisatie stap uitgevoerd met de functie `minimize`, behalve bij de functies `scDifference` en `scFlush`. Deze operaties passen immers de componenten zelf niet aan maar zullen enkel gehele componenten verwijderen. Hierdoor kunnen er geen redundante componenten geïntroduceerd worden.

De Union operatie

De implementatie van de `union` operatie is vrij eenvoudig en is te vinden in de functie `scUnion` van de klasse `UnionExpr`. Het algoritme vertrekt van een kopie van het eerste sticker complex en voegt hieraan kopieën van de componenten van het andere sticker complex toe.

[4]Dit kan je zien door bij het opstarten van de applicatie de vlag `-debug` mee te geven.

De Difference operatie

In het algoritme voor de `difference` operatie (in de functie `scDifference` in de klasse `DiffExpr`) worden eerst de voorwaarden die bij deze operatie gelden gecontroleerd (zie sectie 3.6.2) door over de componenten van de twee sticker complexen te itereren. Indien aan de voorwaarden voldaan is, start het algoritme van een leeg sticker complex en voegt hier via een dubbele for-lus alle componenten van het eerste sticker complex aan toe die geen isomorfe tegenhanger hebben in het tweede sticker complex.

De Ligate operatie

Bij de `ligate` operatie vertrekken we van een kopie van het input sticker complex. Daarna overlopen we al de stickers van lengte twee en kijken we of beide bogen van de sticker gehybridiseerd zijn. We noemen de boog die gehybridiseerd is op de eerste boog van de sticker e_1 en die van de tweede sticker-boog e_2. Indien e_1 geen volgende boog heeft en e_2 geen vorige boog, dan verbinden we de strengen van e_1 en e_2 via de functie `appendStrand` (zie sectie 4.3.2). In de implementatie vindt u deze operatie terug in de functie `scLigate` van de klasse `LigateExpr`.

De Flush operatie

De implementatie van de `flush` operatie (in de functie `scFlush` van de klasse `FlushExpr`) vertrekt van een leeg sticker complex. Aan dit sticker complex worden dan kopieën van de componenten van het input sticker complex toegevoegd die een geïmmobiliseerde boog bevatten.

De Split operatie

Ook het algoritme achter de `split` operatie vertrekt van een kopie van het originele input sticker complex. Vervolgens itereert het algoritme over alle positieve strengen en kijkt bij elk streng of er een boog is die matcht met het opgegeven splitpoint. Indien er zo een boog is, dan wordt eerst de eventuele gehybridiseerde sticker gesplitst en daarna het positieve streng, beiden met behulp van de functie `splitBefore` (zie sectie 4.3.2). Indien we twee verschillende strengen bekomen (dus geen circulaire streng die doorgeknipt werd), dan wordt het tweede streng ook aan het sticker complex toegevoegd. De implementatie hiervan vindt u in de functie `scSplit` van de klasse `SplitExpr`.

De Block operatie

De implementatie van de `block` operatie (in de functie `scBlock` van de klasse `BlockExpr`) kopieert eerst het input sticker complex. Daarna overloopt het al de bogen van al de positieve strengen en controleert of hun label overeenkomt met het opgegeven label. Indien dat zo is, dan wordt de boog geblokkeerd.

De Blockfrom operatie

De `blockfrom` operatie (zie functie `scBlockFrom` van de klasse `BlockFromExpr`) is zeer gelijkaardig aan die van de `block` operatie. Indien er echter bij het overlopen van de bogen van de positieve strengen een match wordt gevonden, dan itereert het algoritme vanaf daar achterwaarts terug over de bogen van dat streng. Het blokkeert dan al deze bogen totdat het een niet-open boog tegenkomt.

De Blockexcept operatie

Ook het algoritme in de functie `scBlockExcept` van de klasse `BlockExceptExpr` voor de `blockexcept` operatie is zeer gelijkaardig aan de vorige twee. Hierbij itereren we eveneens over alle bogen van alle positieve strengen, maar we blokkeren enkel de bogen (met een atomisch waardesymbool) die zich binnen een ℓ-vector bevinden. Hierbij houden we ook een teller bij zodat we de $n$de boog kunnen oplaten.

De Cleanup operatie

Bij de implementatie van de `cleanup` operatie (in de functie `scCleanup` in de klasse `CleanupExpr`) overlopen we eerst alle positieve strengen om zo de maximale lengte te bepalen. Hierbij controleren we ook of er een positieve streng is met een minimale lengte van drie en op de aanwezigheid van minstens één open boog bij alle positieve strengen. Er wordt tevens een lijst bijgehouden van de langste strengen. Nadien worden kopieën van deze 'langste strengen' toegevoegd aan een nieuw leeg sticker complex dat achteraf geminimaliseerd wordt.

4.4.7 Het tekenen van een sticker complex

Het algoritme voor het tekenen van een sticker complex is gebaseerd op een *force-directed graph drawing* algoritme [BETT98, Kob05]. Het basisidee achter dit algoritme is dat knopen van dezelfde component elkaar afstoten en knopen met een boog ertussen elkaar ook aantrekken. Je kan dit vergelijken met de aantrekkings- en afstotingskrachten tussen elektronen en protonen of met de krachten van trekveren (de bogen) en drukveren (tussen alle knopen) volgens de Wet van Hooke. De afstotende kracht heeft een grootte van $C * k^2/d$, de aantrekkende een grootte van $2 * C * d^2/k$. Voor twee knopen a en b op posities (x_1, y_1) en (x_2, y_2), zijn k en d gelijk aan:

$$d = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

$$k = \sqrt{\frac{graaf\ \ oppervlakte}{aantal\ \ knopen}}$$

Experimenteel heb ik ook vastgesteld dat voor mijn applicatie $C = 0,02$ het beste resultaat geeft. Om ervoor te zorgen dat ook de componenten min of meer los van elkaar getekend worden, stoten knopen van verschillende componenten

elkaar ook af maar hierbij is $C = 0,0032$ en is de afstotende kracht dus veel kleiner.

Het algoritme dat ik gebruik is een iteratief algoritme dat zal blijven itereren totdat de maximale verplaatsing kleiner is dan één. Tijdens een iteratie houden we voor elke knoop een displacement-vector bij die geïnitialiseerd wordt op nul. Hierna vergelijken we elke knoop v met elke andere knoop u en maken we de berekening voor de afstotende krachten:

$$\begin{aligned} \delta &= v.pos - u.pos \\ force &= k^2/|\delta| * C \\ v.disp &= v.disp + \delta * force/|\delta| \end{aligned}$$

In deze formules geeft $v.pos$ een tweedimensionale vector terug die de positie (x, y) van knoop v bevat. Merk ook op dat δ dan ook een vector is en dat $d = |\delta|$. Vervolgens berekenen we de aantrekkende krachten. Hiervoor overlopen we al de bogen e en maken we de berekening:

$$\begin{aligned} v &= e.sourceNode \\ u &= e.destinationNode \\ \delta &= v.pos - u.pos \\ force &= |\delta|^2/k * C * 2 \\ v.disp &= v.disp - \delta * force/|\delta| \\ u.disp &= u.disp + \delta * force/|\delta| \end{aligned}$$

We beschouwen bovendien ook aantrekkende krachten tussen twee gehybridiseerde bogen e_1 en e_2. Hiervoor herhalen we de berekening voor de aantrekkende krachten maar nu eenmaal met $v = e_1.sourceNode$ en $u = e_2.sourceNode$ en een andermaal met $v = e_1.destinationNode$ en $u = e_2.destinationNode$. Tenslotte stellen we de nieuwe berekende posities in door voor elke knoop v het volgende uit te voeren:

$$\begin{aligned} x &= v.pos.x + max(min(v.disp.x, 25), -25) \\ y &= v.pos.y + max(min(v.disp.y, 25), -25) \\ v.pos &= (x, y) \end{aligned}$$

Hierbij beperken we de absolute verplaatsing dus tot 25. Bovendien wordt er bij het instellen van de positie ook voor gezorgd dat de knopen steeds binnen een bepaald kader blijven. Indien er een verplaatsing groter dan één plaatsvond, begint het algoritme terug van vooraf aan. Anders stopt het algoritme en blijven de knopen statisch staan. Alle verplaatsingen worden ook 'live' op het scherm getekend zodat de gebruiker gemakkelijk de voortgang van het algoritme kan volgen.

Hoofdstuk 5

Conclusie

Ik wil dit proefschrift graag beëindigen met het bespreken van enkele conclusies. Ik deel deze op in twee secties. In de eerste sectie wil ik enkele bemerkingen formuleren bij het sticker complex datamodel en bij mijn implementatie. In de daarna volgende sectie geef ik mijn algemene conclusies.

5.1 Bemerkingen en verder onderzoek

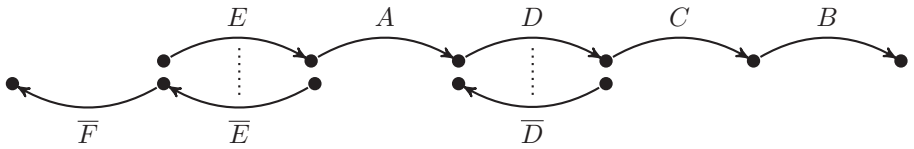
Tijdens het implementeren van het sticker complex datamodel en haar operaties, ben ik soms op problemen gestuit die nog niet helemaal opgehelderd zijn. Een eerste probleem dat ik ben tegengekomen was bij de operatie `blockfrom` (zie sectie 3.6.7). De DNA implementatie van deze operatie vertrouwt erop dat de `block` operatie gebruik maakt van een primer waarvan het 3'-uiteinde aangepast is naar een dideoxy-uiteinde [GV10]. Dit zodat de polymerase van de `blockfrom` operatie op tijd zou stoppen. Het voorbeeld in figuur 5.1 toont echter aan dat het ook nodig is dat stickers een 3' dideoxy-uiteinde nodig hebben, iets wat momenteel nog in het model ontbreekt. Indien ze geen 3' dideoxy-uiteinde hebben, kan DNA polymerase gewoon over de stickers 'lopen' en andere bogen ook blokkeren en zelfs het positieve streng verlengen. Het is uiteraard de bedoeling dat in het voorbeeld van figuur 5.1 de `blockfrom` operatie stopt bij de boog met label d. In de geïmplementeerde simulator gebeurt dit wel op de veronderstelde manier.

Een volgende probleem bevindt zich bij de operatie `ligate`. Om dit probleem te illustreren beschouw ik het sticker complex gegeven in figuur 5.2. Zoals u kan zien, voldoet dit sticker complex niet aan de voorwaarden van een *gat* voor de `ligate` operatie (zie sectie 3.6.4). We zien immers dat $\{e_1, e_4\} \notin \mu$ en $\{e_2, e_3\} \notin \mu$ waardoor de `ligate` operatie het positieve streng niet circulair zal maken. Dit levert ons twee problemen op. Enerzijds is dit in strijd met de werkelijke DNA ligase die wel een circulair streng zou vormen. Anderzijds is dit voorbeeld in tegenspraak met de stelling dat de volgorde van de bogen bij stickers niet van belang is (zie sectie 3.4.3) [GV10]. Ik laat het aan de lezer over om na te gaan dat moesten we de bogen van de sticker omwisselen, de operatie

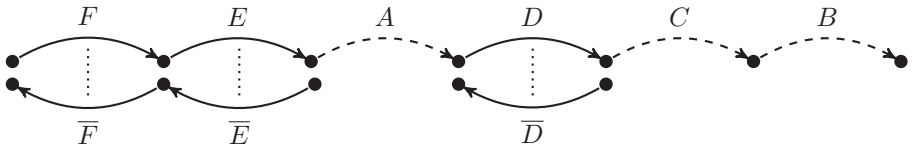
`ligate` wel een circulair streng zal construeren waardoor de volgorde van de bogen bij deze operatie wel van belang is.

Ook met de operatie `cleanup` was er een probleem. Tijdens het schrijven van het DNAQL programma voor de simulatie van het cartesisch product, viel het mij op dat de operatie `cleanup` ook toepasbaar moest zijn op positieve strengen met een lengte kleiner als drie. Er moet echter wel minstens één positieve streng aanwezig zijn met een lengte van minstens drie, wil de DNA implementatie van `cleanup` correct werken. Ik heb de definitie van `cleanup` hiervoor aangepast naar de versie zoals ze in sectie 3.6.9 beschreven staat. U kan gemakkelijk nagaan waarom dit nodig was door de DNAQL Simulator in debug modus uit te voeren op het meegeleverde simulatie programma van het cartesisch product.

Een ander probleem, dat ik reeds aangehaald heb in sectie 4.4.5, is het feit dat er momenteel nog geen efficiënt algoritme gevonden is voor het simuleren van de `hybridize` operatie. Ondanks dat we vrij gemakkelijk kunnen bepalen of een hybridisatie eindig is of niet, is het veel moeilijker om deze effectief uit te rekenen. Het voornaamste probleem zit hem hier in de natuurlijke exponentialiteit van hybridisatie. Het is hierbij van belang geen extra componenten te introduceren die later tot hetzelfde resultaat leiden. Een efficiënt hybridisatie algoritme is duidelijk een punt voor verder onderzoek.

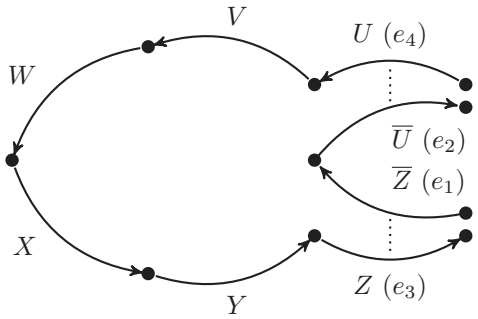


(a) Het sticker complex C



(b) Het resultaat van `blockfrom`(C, B)

Figuur 5.1: Gevolg van het gebrek aan een 3' dideoxy-uiteinde bij stickers



Figuur 5.2: Sticker complex dat niet door `ligate` herkend wordt

Een ander aspect dat verder onderzocht zal moeten worden is het gebruik van datatypes. Momenteel is het met het huidige model niet mogelijk om aan een bepaald attribuut of gecodeerde waarde een type toe te kennen. Een typensysteem is een belangrijk element binnen een database systeem. Het aantonen dat DNAQL al dan niet Turing-compleet is, is eveneens een topic voor verder onderzoek. Ook het zoeken naar andere toepassingen van het sticker complex datamodel binnen DNA computing, kan men aan deze lijst toevoegen.

5.2 Algemene conclusies

Ik ben aan deze bachelorproef begonnen met een beperkte (middelbare school) kennis over DNA en geen kennis over DNA computing of het sticker complex datamodel. Na mij de afgelopen vijf maanden in dit onderwerp verdiept te hebben, kan ik zeggen dat ik enorm veel geleerd heb. In eerste instantie heb ik natuurlijk veel geleerd over DNA en over DNA computing, maar ook over algemenere zaken zoals hoe formele systemen tot stand komen en ontwikkeld worden. Ik heb ook veel ervaring opgedaan bij het lezen van wetenschappelijke artikels en papers om deze daarna kritisch te verwerken. Het schrijven van deze verhandeling heeft me veel bijgebracht over hoe je een groter werk aanpakt en welke schrijfstijl je best hanteert.

Persoonlijk ben ik van mening dat ik de doelstellingen die in deze bachelorproef voorop waren gesteld, gehaald heb. Ik heb een begrijpbare (theoretische) uiteenzetting geschreven over DNA, DNA computing en het sticker complex datamodel. Ik heb eveneens een volledig werkende 'interpreter' voor de DNAQL programmeertaal ontwikkeld. De ontwikkelde applicatie is echter niet op alle vlakken even efficiënt en dit vooral bij de hybridisatie operatie. Maar net zoals bij alle projecten, zou men dit project ook verder kunnen blijven uitbreiden. Ik denk hierbij bijvoorbeeld aan een "compiler" gedeelte dat een DNAQL programma omzet naar een aantal uit te voeren "laboratoriumstappen" zodat men deze programma's ook eens werkelijk zou kunnen testen. Een uitgebreidere debugger of een DNAQL ontwikkelomgeving zijn ook mogelijke, nuttige uitbreidingen.

DNA computing is een onderzoeksgebied dat nog in de kinderschoenen staat maar waarin de laatste jaren reeds veel vooruitgang is geboekt. Het is duidelijk dat DNA computing zich ook uitstekend leent voor databasesystemen. Het sticker complex datamodel is een uitstekende aanzet om dit te realiseren. Er is hiermee al een hele weg afgelegd, maar er is ook nog een hele weg te gaan.

Bibliografie

[Adl94] Leonard M. Adleman. Molecular computation of solutions to combinatorial problems. *Science*, 266(5187):1021 – 1024, Nov 1994.

[Amo05] Martyn Amos. *Theoretical and Experimental DNA Computation*. Springer, 2005.

[Amo09] Martyn Amos. DNA computing. In Robert A. Meyers, editor, *Encyclopedia of Complexity and Systems Science*, volume 4, pages 2089 – 2104. Springer New York, 2009. `http://www.doc.mmu.ac.uk/STAFF/M.Amos/pubs.html`.

[Bal08] Coco Ballantyne. Longest piece of synthetic DNA yet. Internet, January 2008. `http://www.scientificamerican.com/article.cfm?id=longest-piece-of-dna-yet`.

[BETT98] Giuseppe Di Battista, Peter Eades, Roberto Tamassia, and Ioannis G. Tollis. *Graph Drawing: Algorithms for the Visualization of Graphs*. Prentice Hall, 1998.

[BGV11] Robert Brijder, Joris J.M. Gillis, and Jan Van den Bussche. Graph-theoretic formalization of hybridization in DNA sticker complexes. Technical report, Hasselt University and transnational University of Limburg, 2011.

[BPEA+01] Yaakov Benenson, Tamar Paz-Elizur, Rivka Adar, Ehud Keinan, Zvi Livneh, and Ehud Shapiro. Programmable and autonomous computing machine made of biomolecules. *Nature*, 414(6862):430, 2001.

[CRU+08] Neil A. Campbell, Jane B. Reece, Lisa A. Urry, Michael L. Cain, Steven A. Wasserman, Peter V. Minorsky, and Robert B. Jackson. *Biology*. Pearson Benjamin Cummings, 8 edition, 2008.

[dD87] Christian de Duve. *De levende cel*, volume 10 of *De Wetenschappelijke Bibliotheek*. Natuur & Techniek, 1987.

[Dij11] Arjen Dijkgraaf. DNA-sequenser op een chip. *Mens & Molecule*, (2):25, February 2011.

[Fey61] Richard P. Feynman. There's plenty of room at the bottom. In D. Gilbert, editor, *Miniaturization*, pages 282 – 296. Reinhold Publishing Corporation, New York, 1961.

[GHJV09] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns Elements of Reusable Object-Oriented Software*. Addison Wesley professional computing series, 2009.

[GV10] Joris J.M. Gillis and Jan Van den Bussche. A formal model for databases in DNA. Technical report, Hasselt University, 2010. `http://alpha.uhasselt.be/joris.gillis/pubs/anb10.pdf`.

[Kob05] Stephen G. Kobourov. Force-directed drawing algorithms. In Roberto Tamassia, editor, *Handbook of Graph Drawing and Visualization*. CRC Press, 2005. `http://www.cs.brown.edu/~rt/gdhandbook/chapters/force-directed.pdf`.

[LFDR87] Marie Leblond-Francillard, Marc Dreyfus, and FranÃ§ois Rougeon. Isolation of DNA-protein complexes based on streptavidin and biotin interaction. *European Journal of Biochemistry*, 166(2):351 – 355, 1987.

[Mul90] Kary B. Mullis. The unusual origin of the polymerase chain reaction. *Scientific American*, 262(4):56 – 65, 1990.

[OR96] Mitsunori Ogihara and Animesh Ray. Simulating boolean circuits on a DNA computer. In *In Proceedings of 1st International Conference on Computational Molecular Biology*, pages 326 – 331. ACM Press, 1996.

[QL00] Liu Qinghua and Wang Liman. DNA computing on surfaces. *Nature*, 403(6766):175 – 179, 2000.

[QW11] Lulu Qian and Erik Winfree. A simple DNA gate motif for synthesizing large-scale circuits. *J. R. Soc. Interface*, February 2011. `http://rsif.royalsocietypublishing.org/content/early/2011/02/03/rsif.2010.0729.full`.

[Rei11] John H. Reif. Scaling up DNA computation. *Science*, 332(6034):1156 – 1157, June 2011.

[Rus10] Peter J. Russell. *iGenetics A Molecular Approach*. Pearson Education, third edition, 2010. International edition.

[SG90] Harold Swerdlow and Raymond Gesteland. Capillary gel electrophoresis for rapid, high resolution DNA sequencing. *Nucleic Acids Research*, 18(6), 1990.

[Sip06] Michael Sipser. *Introduction to the Theory of Computation*. Course Technology Cengage Learning, second edition, 2006. International edition.

[SKK+99] Kensaku Sakamoto, Daisuke Kiga, Ken Komiya, Hidetaka Gouzu, Shigeyuki Yokoyama, Shuji Ikeda, Hiroshi Sugiyama, and Masami Hagiya. State transitions by molecules. *Biosystems*, 52(1-3):81 – 91, 1999.

[SSZW06] Georg Seelig, David Soloveichik, David Yu Zhang, and Erik Winfree. Enzyme-free nucleic acid logic circuits. *Science*, 314(5805):1585 – 1588, dec 2006.

[Wri99] Robert Wright. Watson & Crick. (cover story). *Time*, 153(12):172, 1999.