Udo Feldkamp

Ein DNA-Sequenz-Compiler

Diplomarbeit

Diplomarbeit

am Fachbereich Informatik

der Universität Dortmund

Lehrstuhl XI

29.10.1999

Betreuer:

Prof. Dr. Wolfgang Banzhaf

Dipl.-Biol. Hilmar Rauhe

Dank

Ich möchte Prof. Dr. Wolfgang Banzhaf und Dipl.-Biol. Hilmar Rauhe für die Betreuung und die fachliche und organisatorische Unterstützung danken. Weiter danke ich Jens Niehaus für Gespräche, L. M. Adleman für die von ihm zur Verfügung gestellten Sequenzen und Ruth Gehrmann für Kritik, Diskussion und viel Geduld.

Inhaltsverzeichnis

1 Einleitung

Das Moore'sche Gesetz, demzufolge sich die Rechenleistung von Microchips alle 18 Monate verdoppelt, wird schätzungsweise 2017 das Ende seiner Gültigkeit erreichen. Zum einen stößt die Verkleinerung der Transistoren auf physikalische Grenzen, wenn die Breite der Leiterbahnen in den Größenbereich von Atomen gerät, zum anderen läßt sich auch die Taktfrequenz nicht beliebig erhöhen, u. a. weil die Verlustleistung und die damit entstehende Wärmeentwicklung schon bei heutigen Prozessoren ein nicht zu vernachlässigendes Problem darstellt.

Die Ansprüche der Benutzer bzw. der Software an die Leistung der Computer dagegen, die bisher mit der Leistungssteigerung der Hardware gewachsen sind, werden wohl in absehbarer Zukunft keine Stagnation erleben. In der Tat gibt es schon heute Probleme, für deren Lösung auch die Computer des Jahres 2017 nicht leistungsfähig genug sein werden: die Probleme der Klasse NP, für die die zu deren Lösung benötigte Rechenzeit exponentiell mit der Größe der Eingabe wächst. Die Suche nach alternativen Rechnerkonzepten ist also ein aktuelles und wichtiges Problem.

R. Feynman hat 1953 in seiner berühmten Rede über die Miniaturisierung „*There's Plenty of Room at the Bottom*" bereits die Idee angesprochen, Moleküle zur Informationsverarbeitung zu benutzen. 1994 hat L. M. Adleman zum ersten Mal DNA-Moleküle verwendet, um eine kleine Instanz eines Standardproblems der Informatik, des Hamilton-Pfad-Problems, zu lösen [Adleman94, Adleman98]. Dadurch inspiriert hat es seitdem eine Vielzahl theoretischer Untersuchungen, Vorschläge zu Rechnermodellen mit DNA und einige tatsächlich im Labor durchgeführte Experimente gegeben. Motivation für das wachsende Interesse am *DNA-Computing* ist die hohe Parallelität der Berechnung durch die Handhabung großer Mengen von DNA in kleinen Volumina.

Das teilweise nur schwer zu kontrollierende und bzgl. der gewünschten Berechnung fehlerbehaftete Verhalten von DNA, vor allem aber die oft umständliche Umsetzung des konventionellen, durch die von-Neumann-Rechnerarchitektur geprägten algorithmischen Denkens in Konzepte des DNA-Computing lassen es zweifelhaft erscheinen, ob jemals ein DNA-Computer den Platz des Silizium-basierten Rechners einnehmen wird. Allerdings gibt es Anwendungsbereiche, in denen der Einsatz von DNA zur Informationsverarbeitung durchaus seine Berechtigung hat, sei es, weil DNA-Computing für bestimmte Probleme Silizium-Rechnern ebenbürtig oder überlegen sein kann, oder weil es Anwendungen jenseits des Schreibens und Ausführens von Programmen gibt. In diesen „*computational niches*" [Adleman95] kann das DNA-Computing konventionelle Computer sinnvoll ergänzen.

Einen vielversprechenden Bereich bilden die Suchprobleme, für deren Lösung auch die meisten DNA-Computing-Modelle entworfen wurden. Hierbei wird ausgenutzt, daß man mit einer entsprechenden Abbildung von Lösungen auf DNA-Sequenzen in einem Volumen, das z. B. in ein Reagenzglas paßt, einen großen Lösungraum eines Problems bilden und durchsuchen kann. So haben z. B. Boneh et al. in der Theorie einen DNA-Computing-Algorithmus entworfen, mit dem man den *Data Encryption Standard* (DES) brechen kann [Boneh95b].

Andere Ansätze verzichten auf die Nachbildung logischer oder arithmetischer Operationen oder Algorithmen mit DNA, sondern konzentrieren sich auf die Rolle der DNA als Informations-*träger*. Beispiele hierfür sind die Markierung von Produkten oder die Steganographie [Rauhe99, LeierRichter99] (s. Abschnitt 3.3.2). Eine wichtige Rolle spielt insbesondere hierbei die kontrollierte Produktion von syntaktisch korrekten Ausdrücken in der DNA-Darstellung. Dazu dient das Konzept des programmierten *self-assembly*, bei dem sich eine große Anzahl solcher Ausdrücke „von selbst“ bilden.

Um DNA-Computing praktisch einsetzbar zu machen, sind insbesondere zwei Aspekte wichtig: Zum einen müssen Verfahren und Werkzeuge entwickelt werden, die flexibel einsetzbar sind, anstatt nur auf ein bestimmtes Problem zugeschnitten zu sein. Zum anderen muß die Wahrscheinlichkeit für das Auftreten von Fehlern bei der Handhabung von DNA minimiert werden.

Das in dieser Arbeit vorgestellte Werkzeug, ein *DNA-Sequenz-Compiler*, ist im Hinblick auf diese beiden Zielsetzungen entwickelt worden. Es dient zur Übersetzung von regulären Grammatiken in DNA-Sequenzen für verschiedene Anwendungen und trägt bereits im Vorfeld, d. h. schon vor der eigentlichen Anwendung im Labor, zur Verringerung der Fehlerwahrscheinlichkeit bei.

Nachdem in Kapitel 2 einige biochemische Grundlagen erläutert werden und ein kurzer Überblick über bisherige Ansätze des DNA-Computing gegeben wird, folgt in Kapitel 3 eine Darstellung des Konzepts des programmierten *self-assembly*, der Aufgabe des Compilers in diesem Konzept sowie möglicher Anwendungen.

In Kapitel 4 wird die Implementierung des Compilers beschrieben, die wichtigsten Merkmale und Methoden des Programms vorgestellt, sowie eine Beschreibung der Parameter und einige Hinweise zur Wahl derselben gegeben.

Einige Beispiele für Übersetzungen mit dem Compiler werden in Kapitel 5 vorgestellt und analysiert.

In Kapitel 6 werden schließlich der Compiler und das programmierte *self-assembly* diskutiert und ein Ausblick auf mögliche zukünftige Entwicklungen gegeben.

2 DNA-Computing

2.1 Biochemische Grundlagen

2.1.1 Was ist DNA?

[Knippers95, Gassen96, Hennig95]

Alle Lebewesen tragen in ihren Zellen genetische Informationen, die als Bauplan für ihren Körper dienen, in einem Makromolekül, der *Desoxyribonucleinsäure* (*DNS*, gebräuchlicher: *DNA* von *desoxyribonucleic acid*). Dieses Makromolekül ist eine Kette von sogenannten *Nucleotiden*, die aus einem Fünf-Kohlenstoff-Zucker (Desoxyribose), einem Phosphatrest und einer Base bestehen.

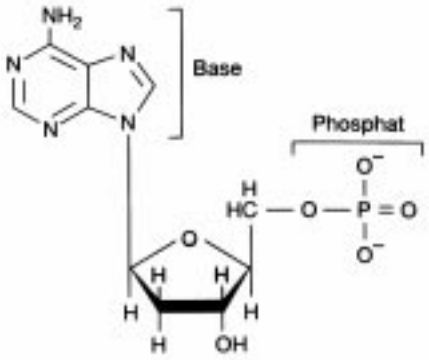


Abbildung 2.1: Nucleotid, bestehend aus Desoxyribose (Mitte), Phosphatrest und Base (hier: Adenin). Quelle: [Hennig95]

In der DNA gibt es vier Arten von Basen: *Adenin* und *Guanin* (die Purinbasen) sowie *Thymin* und *Cytosin* (die Pyrimidinbasen). Sie werden meist nur mit ihren Anfangsbuchstaben (A, G, T, C) bezeichnet, wobei diese Abkürzungen sowohl für die Basen als auch für die gesamten Nucleotide verwendet werden.

Jedes Nucleotid hat zwei unterschiedliche Enden, die als 5’ und 3’ bezeichnet werden. Diese Bezeichnungen stammen vom Phosphatrest, welcher am fünften Kohlenstoffatom (dem 5’-C-Atom) des Zuckers gebunden ist, und der OH-Gruppe am 3’-C-Atom. In einer *Polymerisation* genannten Reaktion verbinden sich jeweils Phosphatrest und OH-Gruppe zweier Nucleotide, so daß sich Ketten bilden, die ebenfalls eindeutige 5’- und 3’-Enden besitzen, nämlich den freien Phosphatrest bzw. die freie OH-Gruppe an den äußersten beiden Nucleotiden. Da die Verbindung zwischen benachbarten Nucleotiden nur durch die Ribose und das Phosphat

hergestellt wird, nicht aber durch die Basen, bezeichnet man die Zucker-Phosphat-Ketten auch als das *Rückgrat* der DNA.

Per Konvention liest man diese Ketten vom 5'-Ende in Richtung 3'-Ende und schreibt z. B. `5′-AACGAT-3′`. Ist die Orientierung unmißverständlich, können die Bezeichnungen der Enden auch weggelassen werden.

Zwei ssDNA-Sequenzen heißen zueinander *revers*, wenn sie aus den gleichen Nucleotidketten mit entgegengesetzter Orientierung bestehen, z. B. `5′-AACGAT-3′` und `3′-AACGAT-5′` bzw. `5′-TAGCAA-3′`.

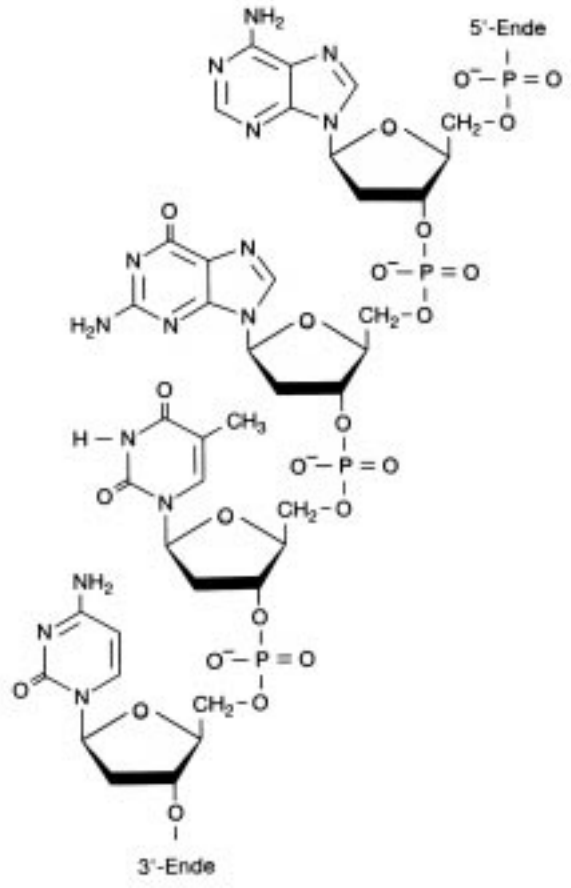


Abbildung 2.2: Nucleotidkette der Sequenz `5′-AGTC-3′`. Quelle: [Hennig95]

In der Natur (d. h. in den Zellen der Lebewesen, *in vivo*) sind diese Ketten (auch Stränge oder Sequenzen genannt) mehrere Tausende oder Millionen Nucleotide (nt) lang, sie werden dann *Polymere* genannt. Im Rahmen des DNA-Computing bewegen sich die Sequenzlängen im ein- bis dreistelligen Bereich, man nennt Sequenzen dieser Länge auch *Oligomere* oder *Oligonucleotide* (oder kurz *Oligos*). Diese werden synthetisch hergestellt und nicht in Zellen, sondern in Reagenzgläsern (*in vitro*) gehandhabt.

Watson und Crick entdeckten 1953, daß DNA in der Natur meist doppelsträngig vorliegt, d. h. zwei DNA-Einzelstränge (ssDNA von *single stranded DNA*) haben sich zu einer gemeinsamen Struktur verbunden, der bekannten Doppelhelix (dsDNA von *double stranded DNA*).

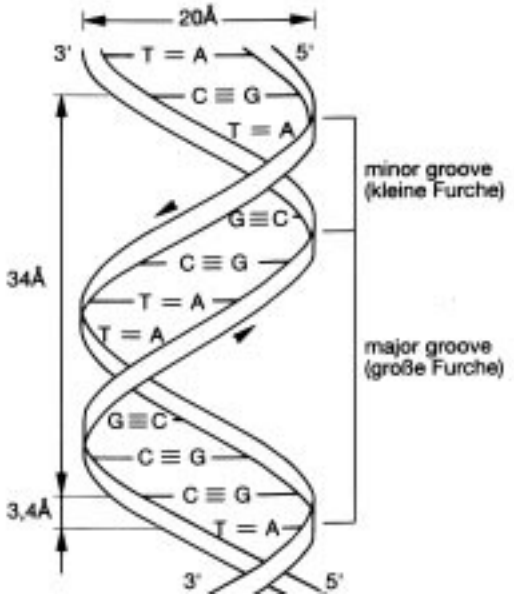


Abbildung 2.3: Watson-Crick-Doppelhelix der DNA. Quelle: [Hennig95]

Hierbei verbinden sich jeweils Pyrimidinbasen des einen Strangs mit Purinbasen des anderen, genauer: jeweils Adenin mit Thymin (per zwei Wasserstoffbrücken) und Guanin mit Cytosin (per drei Wasserstoffbrücken). Daher werden A und T sowie G und C als zueinander (Watson-Crick-) *komplementär* bezeichnet. Diese Verbindung ist nicht covalent und damit instabiler als die Verbindung zwischen benachbarten Nucleotiden im Rückgrat der Einzelstränge.

Die Verbindung zweier Einzelstränge zu einem Doppelstrang ist immer antiparallel, d. h. daß sich z. B. `5′-ACGT-3′` mit `3′-TGCA-5′` verbinden würde, aber nicht mit `5′-TGCA-3′`. Man nennt dann auch diese Einzelstränge zueinander komplementär. Ein Einzelstrang, der mit seinem komplementären Strang identisch ist, heißt *selbstkomplementär*[1] (z. B. ist `ACGT` selbstkomplementär).

Doppelstränge notiert man so, daß der obere Strang in 5'-3'-Richtung gelesen wird, z. B.:

```
5′-ACGT-3′
3′-TGCA-5′
```

[1] Ich vermeide in dieser Arbeit den Begriff Palindrom, der einen Doppelstrang von selbstkomplementären Sequenzen bezeichnet, um Mißverständnisse mit der klassischen Auffassung des Wortes zu vermeiden (`AGGA` z.B. ist kein Palindrom).

In dsDNA kann es vorkommen, daß einzelne sich gegenüberliegende Basen nicht Watson-Crick-komplementär sind (sogenannte *mismatches*). Diese *mismatches* verringern die Schmelztemperatur und damit die Stabilität der dsDNA (s. u.).

2.1.2 Grundlegende Mechanismen und Werkzeuge

Hier sollen nur diejenigen Mechanismen vorgestellt werden, die zur Zeit im DNA-Computing Verwendung finden.

2.1.2.1 Denaturierung und Hybridisierung

Als *Denaturierung* (*Schmelzen*, *Dissoziation*) bezeichnet man das Lösen der Wasserstoffbrücken zwischen den Basen, welches die Teilung eines Doppelstrangs in zwei Einzelstränge zur Folge hat. Am einfachsten ist die Denaturierung durch Erhitzen zu erzielen. Die Stabilität eines Doppelstrangs hängt aber von vielen Faktoren ab, z. B. von der Salzkonzentration der die DNA umgebenden Lösung, so daß sich auch durch Änderung dieser anderen Einflüsse eine Denaturierung erzielen läßt (siehe auch Kapitel 4.1.3.1).

Zu beachten ist, daß zwar der Begriff des Schmelzens für die Denaturierung gebräuchlich ist, aber kein Phasenübergang stattfindet.

Umgekehrt kann man z. B. durch Verringerung der Temperatur aus einzelsträngigen DNA-Sequenzen Doppelstränge erzeugen. Diesen Vorgang nennt man *Hybridisierung*, *Renaturierung* (für vorher denaturierte natürliche DNA), *Reassoziation* oder *annealing* (womit eher das kontrollierte Abkühlen betont wird).

2.1.2.2 Restriktionsendonuclease

Restriktionsendonucleasen sind bakterielle Enzyme, die (meist doppelsträngige) DNA an einer kurzen, für das jeweilige Enzym charakteristischen Subsequenz (der Restriktionsschnittstelle oder *restriction site*) angreift und dort durch Hydrolyse zerschneidet. In der Natur dient dieser Mechanismus dazu, fremde DNA zu zerstören[1]. 1993 waren ca. 2300 verschiedene Restriktionsenzyme für über 200 verschiedene spezifische Restriktionsschnittstellen (sowohl ssDNA als auch dsDNA) bekannt [Maley98].

[1] Genauer gesagt die DNA von Bakteriophagen, das sind Viren, die Bakterien befallen.

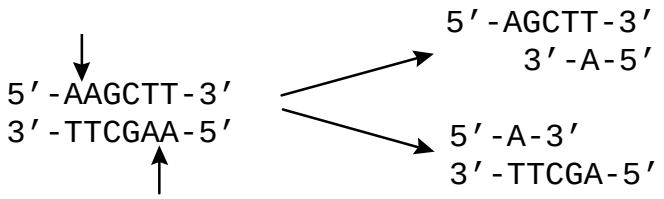


Abbildung 2.4: Spezifisches Schneiden des Restriktionsenzyms HindIII. [Knippers95]

2.1.2.3 Exonuclease

Exonucleasen sind ebenfalls Enzyme zum Abbau von DNA, greifen aber nicht mitten im Strang, sondern an den Enden an. Jede Exonuclease hat eine bestimmte Abbaurichtung (5' $\rightarrow$ 3' oder 3' $\rightarrow$ 5'), viele haben weitere Beschränkungen wie z. B. Abbau nur von einzelsträngiger oder nur von doppelsträngiger DNA.

2.1.2.4 Ligation

Hybridisieren TTTT, GGGG und CCAA zu $\begin{smallmatrix}\mathrm{TTTTGGGG}\\ \mathrm{AACC}\end{smallmatrix}$, so verbinden sich nicht auch automatisch die benachbarten Nucleotide T und G miteinander, beim nächsten Schmelzvorgang würde dieser Doppelstrang also wieder in drei Einzelstränge zerfallen. Das Enzym Ligase schließt diese Lücke, indem es die freie 5'-Phosphatgruppe des Guanin und die ebenfalls freie 3'-OH-Gruppe des Thymin verbindet. Somit wird aus den beiden oberen Strängen ein einzelner Strang.

Oft umfaßt der Begriff der *Ligation* nicht nur die Wirkung der Ligase, sondern auch die vorhergehende Hybridisierung.

2.1.2.5 PCR

Bestimmte Enzyme, Polymerasen genannt, erweitern einsträngige DNA zu zweisträngiger DNA. Dazu muß am 3'-Ende dieser ssDNA (des sog. *templates*) ein kurzes Oligonucleotid, *Primer* genannt, hybridisiert sein, so daß dort bereits ein kleines Stück doppelsträngig ist. Die Polymerase verlängert den Primer, beginnend an dessen 3'-Ende, komplementär zum *template*, dabei bewegt sich die Polymerisation auf dem *template* von 3' nach 5'.

Eine wichtige Anwendung dieses Mechanismus ist die *Polymerase-Kettenreaktion* oder *PCR* (*polymerase chain reaction*). Hierbei wird eine doppelsträngige DNA-Sequenz zunächst denaturiert. Dann läßt man die beiden Einzelstränge mit den jeweiligen Primern hybridisieren, wobei die Primer im Überschuß angeboten werden, um eine Hybridisierung mit einem Primer wahrscheinlicher zu machen als die Reassoziation der beiden Einzelstränge. Anschließend

werden die Einzelstränge mit Hilfe der Polymerase zu Doppelsträngen vervollständigt, so daß nun zwei Kopien des Original-Doppelstrangs vorliegen. Wiederholt man diese Schritte, so kann man mit jedem Durchlauf die Anzahl der Doppelstränge verdoppeln und somit auch kleinste Mengen DNA zu nachweisbaren und für das DNA-Computing verwendbaren Mengen vervielfältigen (*amplifizieren*)[1]. Notwendig dazu ist allerdings die Kenntnis der Primersequenzen.

2.1.2.6 Gelelektrophorese

Bei diesem Verfahren bringt man die DNA auf ein Gel auf und legt ein elektrisches Feld an. Die Sequenzen wandern dann durch das Gel auf die Anode zu, dabei ist ihre Geschwindigkeit umgekehrt proportional zum Logarithmus ihrer Größe, sie bewegen sich also um so schneller je kürzer sie sind. Nach einer gewissen Zeit erhält man so eine Längensortierung der Sequenzen im Gel; Sequenzen gleicher Länge gruppieren sich zu sog. *Banden*. Mit einer anschließenden Färbung der DNA kann man diese Banden sichtbar machen.

Läßt man auf einem reservierten Teil des Gels DNA-Stränge mit bekannten Längen laufen, bilden diese Banden hinterher einen Maßstab für die Größe der zu untersuchenden oder zu sortierenden Banden.

2.1.2.7 Sequenzierung

Die momentan gebräuchlichste Methode, um DNA zu sequenzieren, d. h. um die Basenfolge eines DNA-Strangs auszulesen, ist die Didesoxymethode von Sanger bzw. darauf basierende modifizierte Verfahren.

Didesoxynucleotide sind Moleküle, die sich wie Desoxynucleotide per Polymerase an DNA-Stränge anhängen lassen, sie besitzen jedoch keine 3'-OH-Gruppe, so daß kein weiteres Nucleotid an diesen Strang angehängt werden kann, die Polymerisationsreaktion bricht hier ab. Dies nutzt man zur Sequenzierung, indem man in vier getrennten Reaktionsansätzen Polymerisationen mit der zu sequenzierenden DNA als *template* durchführt und außer den vier Desoxynucleotiden zu jedem Ansatz Didesoxynucleotide mit jeweils einer der vier Basen hinzugibt. Dadurch enden die neu synthetisierten Stränge in einem Ansatz mit einer bestimmten Base, die der hinzugegebenen Didesoxynucleotide. Trennt man nun die Produkte der vier Ansätze in einer Gelelektrophorese mit einem hochauflösenden Polyacrylamidgel nach der

[1] Genauer gesagt wird der Bereich zwischen den Primern amplifiziert, die nicht unbedingt an den Enden der *templates* hybridisieren.

Länge auf, so läßt sich aus den Längen und damit den Positionen, an denen die Polymerisation abgebrochen ist, die Sequenzfolge ablesen.

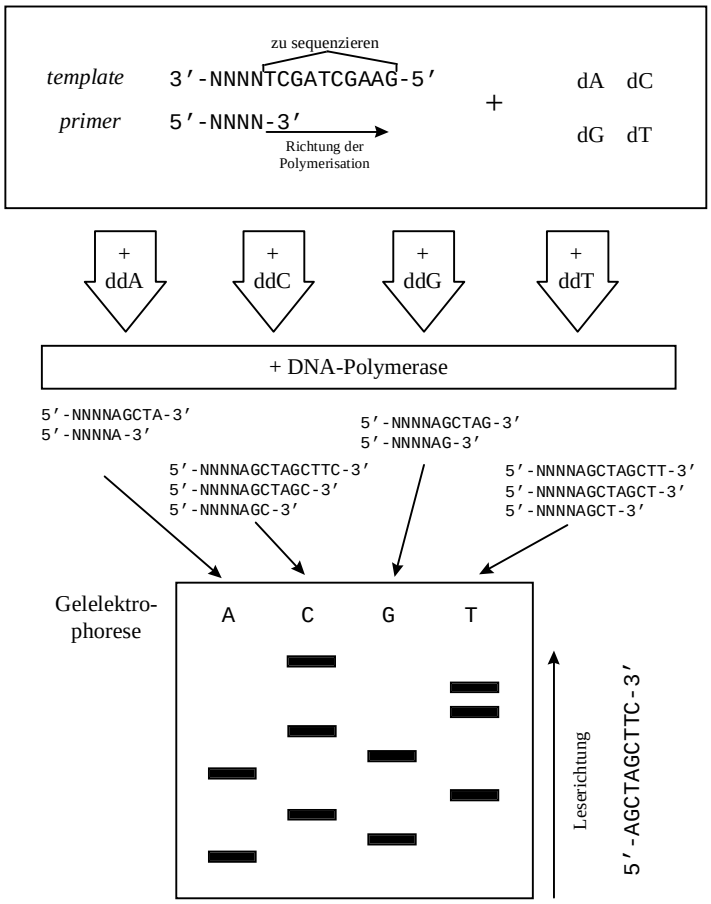


Abbildung 2.5: Sequenzierung mit der Didesoxymethode nach Sanger. dA, dC, dG und dT stehen für Desoxynucleotide, ddA, ddC, ddG und ddT für Didesoxynucleotide. [Gassen96]

Notwendig ist hierzu natürlich die Kenntnis eines Primers, an dem die Polymerisation starten kann, für die Sequenzierung einer völlig unbekannten Sequenz ist diese Methode ungeeignet. Im Bereich des DNA-Computing, in dem die verwendeten DNA-Sequenzen gezielt konstruiert werden, ist das Verfahren nach Sanger jedoch ausreichend.

2.2 Rechnen mit DNA

L. M. Adleman löste 1994 ein kleines Hamilton-Pfad-Problem mit DNA-Molekülen [Adleman94, Adleman98]. Seitdem wurden auch andere Probleme mit DNA gelöst bzw. es wurde beschrieben, wie man diese Probleme mit DNA lösen kann. Im Bestreben, eine allgemeinere, d. h. nicht nur auf ein Problem zugeschnittene Verfahrensweise zum Rechnen mit DNA zu finden, wurden verschiedene Modelle aufgestellt, die hier kurz vorgestellt werden. Für eine ausführlichere Diskussion der Modelle siehe [Niehaus98].

In den meisten Modellen (die zur Lösung von Suchproblemen entworfen wurden) läßt sich das Verfahren in zwei Phasen zerlegen:

- Eine Initialisierungsphase, in der DNA-Sequenzen erzeugt werden, die Lösungskandidaten für das zu lösende Problem kodieren.
- Eine Berechnungsphase, in der die „schlechten" Kandidaten, d. h. diejenigen, die keine gültige Lösung darstellen, aussortiert werden.

In der Berechnungsphase werden Operatoren verwendet wie die Vereinigung von DNA-Mengen, das Extrahieren von Sequenzen mit bestimmten Teilsequenzen, die Sortierung der Sequenzen nach Länge per Gelelektrophorese usw.

Um zu verdeutlichen, wie ein solches Verfahren funktioniert, folgt zunächst als Beispiel der DNA-Computing-„Klassiker": Adlemans Lösung eines Hamilton-Pfad-Problems.

2.2.1 Adlemans Lösung des Hamilton-Pfad-Problems

Das *Hamilton-Pfad-Problem* (*HPP*) besteht darin zu entscheiden, ob es einen gerichteten Weg durch einen Graphen von einem Startknoten v_s zu einem Endknoten v_e gibt, auf dem jeder Knoten des Graphen genau einmal besucht wird (ein solcher Weg heißt *Hamilton-Pfad*). Dieses Problem ist NP-vollständig, d. h. jedes andere Problem der Klasse NP kann auf das HPP reduziert werden und es existiert kein effizienter Lösungsalgorithmus für das HPP.

Adlemans Beispielgraph [Adleman94, Adleman98] hatte 7 Knoten, 14 Kanten und genau einen Hamilton-Pfad. Sein (nicht-deterministischer) Lösungsalgorithmus sah wie folgt aus:

1. Erzeuge zufällige Pfade durch den Graphen.
2. Behalte nur die Pfade, die mit v_s beginnen und mit v_e aufhören.
3. Behalte nur die Pfade, die genau |V| Knoten enthalten.
4. Behalte nur die Pfade, die alle Knoten besuchen.
5. Falls mindestens ein Pfad übrig bleibt, antworte „Ja", sonst „Nein".

Um diesen Algorithmus mit DNA-Sequenzen durchzuführen, kodierte Adleman die Knoten und Kanten als Sequenzen der Länge 20. Wenn die Sequenzen für zwei Knoten v und w aus je zwei 10-meren v_l und v_r bzw. w_l und w_r bestehen, so besteht die Sequenz für die Kante $v \rightarrow w$ aus den Komplementären zu w_l und v_r.

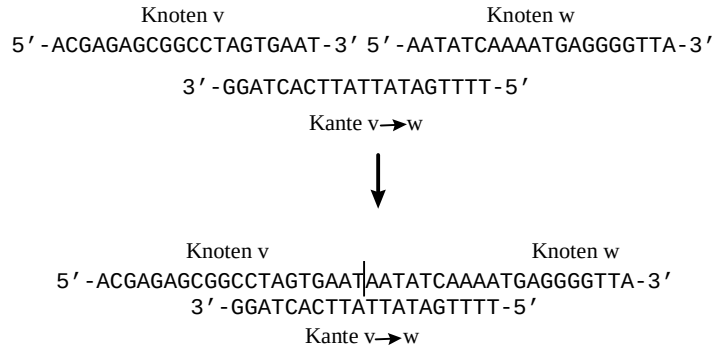


Abbildung 2.6: Kanten- und Knotensequenzen nach Adleman

Somit hybridisiert eine Kantensequenz überlappend mit den Sequenzen der beiden Knoten, die die Kante verbindet, über ein Länge von jeweils 10 Nucleotiden. Diese Hybridisierung führt zu langen Doppelsträngen, die Pfade durch den Graphen repräsentieren, implementiert also Schritt 1 des Algorithmus. Adleman erzeugte ca. 10^{14} Sequenzen, konnte also ziemlich sicher sein, auch den Hamilton-Pfad darunter zu finden.

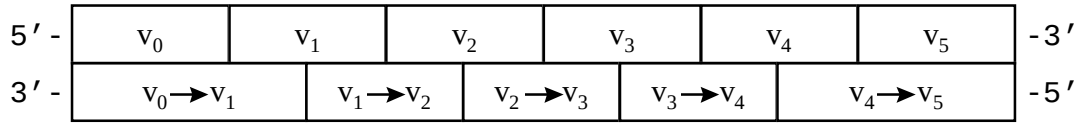


Abbildung 2.7: Ligation von Kanten- und Knotensequenzen zu einer Pfadsequenz. Die Kantensequenzen $v_0 \rightarrow v_1$ und $v_4 \rightarrow v_5$ sind länger gewählt worden, um glatte Enden zu erhalten.

Schritt 2 wurde durch eine PCR implementiert, in der die Sequenz des Startknotens und die Komplementärsequenz des Endknotens als Primer verwendet wurden, so daß nur die Sequenzen (Pfade) verstärkt wurden, die mit eben diesen Knoten anfingen bzw. endeten.

Für Schritt 3 verwendete Adleman eine Gelelektrophorese. Mit ihr konnte er genau die Sequenzen aussortieren, die 140 Basenpaare lang waren, also Pfade der richtigen Länge repräsentierten (7 Knoten á 20 bp).

Mit Hilfe von mikroskopisch kleinen Eisenkügelchen, auf deren Oberfläche Sondensequenzen (die Komplementärsequenzen eines Knotens v) angebracht waren und die man mit einem Magneten mechanisch handhaben konnte, war Adleman in der Lage, diejenigen Sequenzen an die Kügelchen anzulagern und zu extrahieren, die den entsprechenden Knoten v enthielten. Sukzessive Ausführung dieses Verfahrens für jeden Knoten ergaben die Extraktion genau der Sequenzen, die jeden Knoten enthalten (Schritt 4). Da diese die Länge |V| haben, müssen es die Sequenzen sein, die Hamilton-Pfade darstellen.

Der letzte Schritt (nachzuprüfen, ob überhaupt noch Sequenzen vorhanden sind) bestand aus einer weiteren PCR mit anschließender Gelelektrophorese. Adleman fand in seinem Experiment Sequenzen vor, die tatsächlich diejenigen waren, die den richtigen Pfad repräsentierten.

2.2.2 Das beschränkte Modell

Eine erste Formalisierung eines Modells nach Adlemans Experiment fand 1995 durch Lipton statt [Lipton95]. Er entwickelte das beschränkte Modell, das seinen Namen dem Umstand verdankt, daß die verwendete DNA-Menge in der Berechnungsphase nicht mehr zunimmt, also durch die Initialisierungsmenge beschränkt ist. Dies schließt die Anwendung von PCR aus.

In der Initialisierungsphase werden Bitvektoren kodierende Sequenzen erzeugt. Hierzu erstellte Lipton einen Graphen, dessen Knoten Nullen und Einsen enthalten. Mit einer Kodierung des Graphen, wie Adleman sie vorgeschlagen hat (eine Sequenz pro Knoten, also zwei Sequenzen für jede Bitposition), können nun Pfade von einem Start- zu einem Zielknoten erzeugt werden, die Bitvektoren einer bestimmten Länge repräsentieren[1].

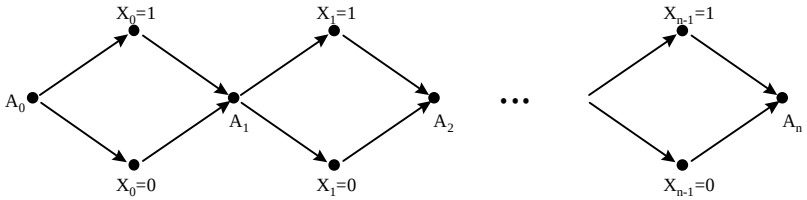


Abbildung 2.8: Liptons Graph zur Erzeugung von Bitvektoren. [Lipton95]

Der Vorteil dieser Kodierung liegt darin, daß man die Bitstrings auch zur Lösung anderer Probleme wiederverwenden kann, es muß nur die Berechnungsphase entsprechend geändert werden.

In der Berechnungsphase werden folgende Operatoren verwendet:

Extraction: Sequenzen, die eine bestimmte Subsequenz enthalten, und solche, die sie nicht enthalten, werden in zwei verschieden Reagenzgläser getrennt. Dies wird i. a. ähnlich durchgeführt wie Schritt 4 in Adlemans Lösungsalgorithmus für das HPP.

Merge: Die Vereinigung zweier Mengen von DNA. Sie wird durch Vereinigung der Inhalte der beiden Reagenzgläser implementiert.

[1] In Liptons Aufsatz waren diese Bitvektoren Belegungen der Variablen eines Boole'schen Ausdrucks in konjunktiver Normalform und wurden zur Lösung des Erfüllbarkeitsproblems SAT verwendet.

Detection: Es wird getestet, ob sich überhaupt noch DNA im Reagenzglas befindet. Hierzu kann man z. B. eine PCR mit anschließender Gelelektrophorese durchführen.

2.2.3 Das unbeschränkte Modell

Adleman [Adleman96] erweiterte das beschränkte Modell um folgenden Operator:

Amplify: Vervielfältigt den Inhalt eines Reagenzglases. Hierzu verwendet man i. a. die PCR.

Durch diesen zusätzlichen Operator für die Berechnungsphase fällt die Mengenbeschränkung weg. Dadurch werden ggf. geringere Initialmengen von DNA benötigt, die PCR wird aber als eine wesentliche Fehlerquelle angesehen (s. u.).

2.2.4 Das Generator-Modell

Da im Laufe der Berechnung im beschränkten Modell die Menge der verwendeten DNA immer mehr abnimmt und das unbeschränkte Modell zur Amplifizierung die fehlerbehaftete PCR verwendet, schlugen Bach et al. vor, bereits die Initialisierungsmenge effizienter zu nutzen [Bach96]. Indem man gezielt Wissen um die Problemlösungen in die Erzeugung der Initialisierungssequenzen einfließen läßt, kann man den Suchraum von vornherein einschränken und die vorhandene DNA-Menge vermehrt für sinnvolle Lösungskandidaten verwenden.

Zu diesem Zweck gibt es folgende Operatoren für die Initialisierungsphase:

Split: Eine DNA-Menge wird so auf zwei Reagenzgläser verteilt, daß beide Gläser ca. die gleiche Menge enthalten. Dabei ist die Zuordnung einer Sequenz zu einem der beiden Gläser natürlich rein zufällig.

Append: Die in einem Reagenzglas enthaltenen Sequenzen werden um eine bestimmte Sequenz verlängert (durch Hybridisierung und Ligation entsprechend der Adleman'schen Kodierung).

Merge: Wie im beschränkten Modell werden hier die DNA-Mengen vereinigt.

Mit iterativer Anwendung der Operatorensequenz **Split** - **Append** - **Merge** können so ebenfalls zufällige Bitvektoren erzeugt werden, aber durch Weglassen von **Split** und **Merge** an bestimmten Positionen haben alle Sequenzen an der entsprechende Position dieselbe Bitbelegung, der Suchraum hat damit eine Dimension weniger.

2.2.5 Das Sticker-Modell

Das Sticker-Modell [Roweis96] bietet eine alternative Kodierung von Bitvektoren. Die Sequenz, die den Vektor repräsentieren soll, ist zunächst einsträngig. Für jede Bitposition gibt

es nur eine Sequenz (statt zwei wie in den vorhergehenden Modellen). Ist diese Subsequenz im Vektor einsträngig, so hat das entsprechende Bit den Wert 0, ist sie doppelsträngig, so hat das Bit den Wert 1. Die Komplemente der Bitsequenzen, die sich an den Einserstellen anlagern, heißen *sticker*.

Eine zufällige (gleichverteilte) Initialisierung einer Bitposition läßt sich erreichen, indem man halb so viele *sticker* wie Vektorensequenzen in ein Reagenzglas gibt und eine Hybridisierung durchführt.

In der Berechnungsphase gibt es außerdem zusätzliche Operatoren:

Separate: Trennt die Sequenzen in einem Reagenzglas in zwei Mengen auf: In einer sind alle Sequenzen, bei denen ein bestimmtes Bit gesetzt ist, in der anderen alle Sequenzen, bei denen dieses Bit gelöscht ist. Dieser Operator wird ähnlich implementiert wie **Extraction**.

Set, Clear: Setzt bzw. löscht bei allen Sequenzen ein bestimmtes Bit. Während sich **Set** einfach durch Hybridisierung mit dem entsprechenden *sticker* implementieren läßt, ist die Umsetzung von **Clear** nicht so offensichtlich. Ein Erwärmen zur Dissoziation eines *stickers* würde auch die anderen *sticker* (an den anderen Bitpositionen) lösen, ist also ungeeignet. Eine vorgeschlagene Methode ist die Verwendung von Strängen, die nur aus Pyrimidinbasen bestehen und sich spezifisch an bestimmte *sticker*-Stränge zu Dreifach-Helices anlagern, welche instabiler sind als die doppelsträngigen Bereiche und daher bereits bei geringeren Temperaturen dissoziieren [Bach96].

Ein Nachteil dieses Modells neben der schwierigen Verwirklichung des Operators **Clear** ist die Beschränkung auf Vektoren von binären Variablen, Variablen mit größeren Wertebereichen können so nicht kodiert werden.

2.2.6 Das Restriktionsenzym-Modell

Da ein wesentlicher Operator der bisher vorgestellten Modelle, nämlich **Extraction** (bzw. **Separate** im Sticker-Modell), sehr fehlerbehaftet ist, haben Amos et al. eine Alternative zum Aussortieren schlechter Kandidaten vorgeschlagen [Amos96]. Auch hier sind die Kandidaten zunächst einsträngig und enthalten Restriktionsschnittstellen an bestimmten Stellen. Statt **Extraction** wird dieser Operator verwendet:

Remove: Die schlechten Kandidaten werden markiert, indem sie mit den Komplementärsequenzen der Restriktionsschnittstellen hybridisieren. Anschließend werden sie (und nur sie) durch Restriktionsenzyme zerschnitten, die spezifisch dsDNA schneiden, die Teile lassen sich dann anhand der geringeren Länge per Gelelektrophorese aussortieren.

2.2.7 Das Surface-Modell

In diesem Modell bewegen sich die DNA-Sequenzen nicht frei in einer Lösung, sondern sie sind auf einer Oberfläche fixiert [Liu96, Cai96]. Somit sind die Vorgänge während der Initialisierungs- und Berechnungsphase wesentlich einfacher zu kontrollieren, allerdings engt man den Suchraum dadurch ein, daß man nur noch zwei statt drei Dimensionen für die Anordnung der Sequenzen zur Verfügung hat. Hauptsächlich soll dieses Modell auch eher dazu dienen, in Experimenten Erkenntnisse über die Vorgänge bei der Ausführung der Operatoren und die Fehler, die dabei auftreten, zu gewinnen, um damit auch die lösungsbasierten Ansätze verbessern zu können.

Verwendet werden hier folgende Operatoren:

Mark: Die eigentlich einsträngigen Sequenzen, die eine bestimmte Bitbelegung haben, werden markiert, indem sie (mit Anlagerung und Polymerisation) doppelsträngig gemacht werden.

Unmark: Alle Markierungen werden gelöscht, indem die dsDNA denaturiert wird. Da die Kandidatenstränge fixiert sind, lassen sich die Markierungsstränge problemlos entfernen (abspülen).

Destroy-marked, -unmarked: Zerstört alle markierten bzw. alle unmarkierten Sequenzen. Dies wird mit Exonucleasen realisiert, die nur dsDNA bzw. nur ssDNA zerstören.

Append-marked, -unmarked: An die markierten bzw. unmarkierten Sequenzen wird eine bestimmte Sequenz angehängt. Dies wird mit verschiedenen Arten von Ligation erreicht.

Erase-marked: Schneidet den Teil aller markierten Sequenzen ab, der mit einer vorhergegangenen **Append**-Operation angehängt wurde. Hierzu werden Restriktionsenzyme verwendet.

2.2.8 Turing-Maschinen-Modelle

Eine Turing-Maschine ist ein (theoretisches) Rechnermodell, das auf einem unendlich langem Band Zeichen lesen kann. Abhängig vom Zustand der Maschine und dem gelesenen Zeichen kann ein Zeichen geschrieben, der Schreib-/Lesekopf nach rechts oder links bewegt sowie der Steuerungsautomat in einen neuen Zustand überführt werden.

Aufgrund der sehr primitiven Operationen (ganz zu schweigen vom Mangel an unendlich langen Bändern) werden Turing-Maschinen nicht zum Rechnen benutzt (sie werden nicht einmal praktisch gebaut), obwohl sie die gleiche Berechnungskraft besitzen wie ein Computer, der mit einer Hochsprache programmiert wird. Sie dienen vielmehr der theoretischen Informatik

als Modell, an dem sich gerade wegen ihrer Einfachheit Berechenbarkeit und Komplexität von Problemen untersuchen lassen.

Beaver hat ein theoretisches DNA-Computing-Modell beschrieben, das einer Turing-Maschine entsprechen soll [Beaver95]. Dieses ist ebenfalls nicht zur wirklichen Anwendung geeignet, sondern soll nur zeigen, daß das DNA-Computing eine gleiche Berechnungskraft besitzt wie konventionelle Computer. Daß diese Analogie nicht nur umständlich, sondern auch sehr gewagt (streng genommen sogar inkorrekt) ist, hat Niehaus ausgeführt [Niehaus98].

Auch Rothemund hat ausführlich gezeigt, daß das DNA-Computing die gleiche Berechnungskraft besitzt wie die Turing-Maschine [Rothemund96]. Shapiro hat sogar ein dreidimensionales Modell einer molekularen Turing-Maschine entworfen, dessen Umsetzung jedoch auch fraglich ist [Shapiro99].

2.2.9 Weitere Ansätze

Guarnieri et al. haben einen Ansatz *in vitro* realisiert, der es erlaubt, Addition von Binärzahlen mit DNA zu implementieren [Guarnieri96]. Wesentlicher Nachteil der vorgestellten Methode ist die unterschiedliche Kodierung von Ein- und Ausgabezahlen in DNA-Sequenzen, so daß sich das Ergebnis einer Addition nicht für eine weitere Addition verwenden läßt.

Gupta et al. schlagen in [Gupta97] ein Modell vor, um einfache logische und arithmetische Operationen durchzuführen. Es beruht u. a. auf dem originellen Prinzip, die Ausgabesequenz je nach Wert der Eingabe und anzuwendendem Operator anders zu interpretieren.

Rose et al. haben einen Ansatz entworfen, um einen endlichen Automaten mit DNA zu simulieren [Rose97]. Hierbei kodieren die zu bearbeitenden Sequenzen jeweils den aktuellen Zustand des Automaten und das zuletzt gelesene Zeichen der Eingabe. Allerdings muß in diesem Modell jede Transition von Hand durchgeführt werden (wenn auch für viele Instanzen des Automaten parallel), was die Effizienz nachteilig beeinflußt. Auch bleibt unklar, wie das Erreichen eines Endzustands erkannt werden soll. Eine Implementierung *in vitro* steht noch aus.

Amos et al. entwickelten (zumindest in der Theorie) ein Verfahren, mit dem man die Ausgabe von Schaltkreisen aus NAND-Gattern mit DNA berechnen kann [Amos97]. Hier sind zeitaufwendige Schritte für jede Stufe des Schaltkreises notwendig.

Boneh et al. beschreiben ein Verfahren, mit dem es möglich ist, innerhalb von etwa vier Monaten die Verschlüsselung des DES (*Data Encryption Standard*) zu brechen (d. h. den 56-Bit-Schlüssel zu finden), wenn man eine Nachricht sowohl verschlüsselt als auch als Klartext

vorliegen hat. Dies funktioniert auch, wenn man nur weiß, daß der Klartext einer von mehreren Kandidaten ist [Boneh95b].

Da DNA sowie deren Mutations- und Rekombinationsverhalten das Vorbild für die Genetischen Algorithmen[1] ist, liegt es nah, diese mit Hilfe von DNA zu implementieren [Maley98]. Die größten Schwierigkeiten liegen hierbei bei der Berechnung der Fitneß und der Selektion.

2.2.10 Diskussion

Als wesentliche Vorteile des DNA-Computing werden angesehen [Adleman94]:

- Die hohe Parallelität der Berechnung. Für die Initialisierungsphase seines HPP-Experiments hat Adleman 10^{14} Ligationen pro Sekunde veranschlagt, wies aber darauf hin, daß sich bis zu 2^{70} DNA-Stränge in einem Reagenzglas befinden können, die parallel bearbeitet werden können.
- Der geringe Platzbedarf für die Speicherung von Informationen. Für die Speicherung von 1Bit wird nur etwa 1 nm^3 benötigt.
- Die Energieeffizienz. Mit Aufwendung von 1 Joule können $2*10^{19}$ Operationen (Ligationen) durchgeführt werden. Das theoretische Maximum liegt bei $34*10^{19}$ Operationen pro Joule.

Allerdings stellen sich auch einige wesentlichen Probleme:

- Die große Rechenkraft des DNA-Computing ist auf einige Abschnitte eines Algorithmus beschränkt. Viele Operatoren der Berechnungsphase sind dagegen sehr zeitaufwendig. Insbesondere sind Ein- und Ausgabe (Kodierung der Kandidaten in Sequenzen, Auslesen der übriggebliebenen Sequenzen) noch sehr zeitintensiv.
- NP-Probleme bleiben auch in der Architektur des DNA-Computing NP-Probleme. Der exponentielle Aufwand zur Lösung solcher Probleme verlagert sich zwar von der Rechenzeit zur Menge der verwendeten DNA, aber da diese eine zwar sehr kleine, doch von Null verschiedene Ausdehnung hat, sind auch dem DNA-Computing Grenzen bzgl. der Größe der lösbaren Probleme gesetzt. Z. B. würde man zur Lösung eines HPP mit 70 Knoten mit Adlemans Verfahren ca. 10^{25} kg DNA brauchen [Maley98] (zum Vergleich: die Erdmasse beträgt ca. $6*10^{24}$ kg).

[1] Genetische Algorithmen gehören zu den Evolutionären Algorithmen (s. 6.2 für eine Beschreibung).

- Die verwendeten Operatoren arbeiten nicht perfekt, sondern sind fehlerbehaftet; manche Verfahren, wie die PCR oder die Implementierung von **Extraction**, sogar sehr. Darüber hinaus gibt es nur wenige fundierte Abschätzungen zur Quantifizierung dieser Fehler.

2.2.11 Fehler im DNA-Computing

Da die meisten bisherigen Ansätze auf der Trennung von falschen und richtigen Lösungen (oder besser: von schlechten und guten) beruhen, werden die Fehler, die bei den Berechnungen auftreten können, auf der Ebene des Algorithmus, der mit DNA-Computing durchgeführt werden soll, unterschieden in *falsche Negative* (eine gute Lösung wurde nicht gefunden oder als schlecht aussortiert) und *falsche Positive* (schlechte Lösungen wurden als gute ausgegeben).

Auf der Implementierungsebene dagegen betrachtet man die Unvollkommenheiten der Operationen. So arbeitet z. B. die u. a. in der PCR verwendete Polymerase nicht völlig fehlerfrei, sondern hin und wieder (mit einer Wahrscheinlichkeit von ca. 10^{-5} bis 10^{-6}) wird eine falsche Base statt der zum *template* komplementären angefügt.

Wesentlich folgenreicher sind *Fehlhybridisierungen*, d. h. Hybridisierungen zwischen (Sub-) Sequenzen, die gemäß des zu berechnenden Algorithmus keinen Doppelstrang bilden sollten. Dieser Fehler kommt dadurch zustande, daß für die Anlagerung keine perfekte Komplementarität zwischen den beiden Einzelsträngen bestehen muß. Auch wenn *mismatches* bestehen, können zwei Sequenzen hybridisieren.

An fehlerhaften Anlagerungen können außerdem Verschiebungen der Einzelstränge oder Teile dieser gegeneinander auftreten (*shifts*), sowie Schlaufen (*loops*) und Ausbuchtungen (*bulges*). Auch müssen nicht immer genau zwei ssDNA-Stränge beteiligt sein (*hairpin loops, junctions*).

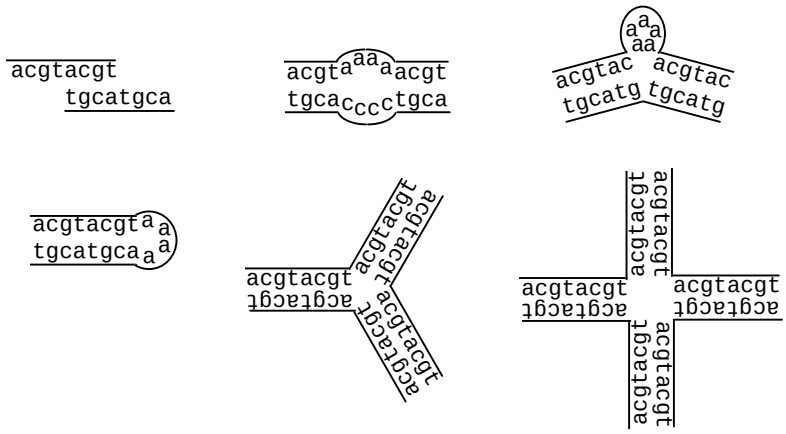


Abbildung 2.9: Fehlerhafte Anlagerungen. Oben (v.l.n.r.): *shift, internal loop, bulge*, unten (v.l.n.r.): *hairpin loop, junction* aus drei bzw. vier Einzelsträngen

Um alle diese Fehlermöglichkeiten zu minimieren, sollten die Implementierungen der Operatoren *in vitro* verbessert werden. Jedoch auch bereits im Vorfeld der eigentlichen Berechnung, nämlich bei der Wahl der zu verwendenden Sequenzen, kann man die Wahrscheinlichkeit für das Auftreten von Fehlern verringern.

3 Die Aufgabe des DNA-Sequenz-Compilers

Für das im Rahmen dieser Arbeit entwickelte Programm, den DNA-Sequenz-Compiler, galten folgende allgemeine Überlegungen und Vorgaben:

- Um das DNA-Computing allgemeiner einsetzbar zu machen, muß eine Programmierschnittstelle entwickelt werden, mit der man DNA-Programme auf höherer Abstraktionsebene formulieren kann als auf der molekularen Ebene. Ein erster Schritt hierzu waren die in 2.2 vorgestellten Operatoren. Neben einer Automatisierung der Ausführung dieser Operatoren, die momentan noch vom Menschen manuell erfolgt, ist eine Automatisierung der Kodierung von Daten sowie deren gewünschtem Verhalten bei der Ausführung der Operatoren wünschenswert.
- Da die Operatoren des DNA-Computing nicht fehlerfrei arbeiten, müssen Maßnahmen getroffen werden, um die Wahrscheinlichkeit für das Auftreten von Fehlern zu minimieren sowie die Auswirkungen von bestimmten Fehlern abzuschwächen. Dies kann nicht nur durch Verbesserung der Implementierung der Operatoren geschehen, sondern bereits im Vorfeld durch eine geschickte Wahl der zu verwendenden DNA-Sequenzen.
- Für die Lösung von NP-Problemen ist es auch im DNA-Computing sinnvoll, den Rechenaufwand (hier im wesentlichen die Menge der benötigten DNA) zu verringern, indem man den zu durchsuchenden Lösungsraum einschränkt. Dies wurde im Generator-Modell verwirklicht. Anstatt aber wie dort die Einschränkung durch mehrfache Anwendung der Operatoren **Split**, **Append** und **Merge** vorzunehmen, wäre es bequemer, die Sequenzen von vornherein (d. h. bevor die Berechnung *in vitro* angesetzt wird) so zu wählen, daß die Einschränkung in der Kodierung inhärent ist. Boneh et al. schlagen in diesem Zusammenhang die Verwendung von regulären Ausdrücken in der „*initial soup*“, also der Initialmenge von DNA, vor [Boneh96].
- Neben der Lösung von NP-Problemen sind noch andere Anwendungen für das DNA-Computing denkbar. Diese sollten aber auch von der oben erwähnten Automatisierung profitieren können.
- Insbesondere das in 3.3 vorgestellte Verfahren des programmierten *self-assembly* soll durch den Compiler unterstützt und eine Automatisierung ermöglicht werden.

Eine interessante Arbeit in diesem Zusammenhang ist die von Amos et al., die einen Compiler zur Übersetzung von CREW-PRAM-Programmen (*concurrent-reading-exclusive-writing parallel random access machine*, ein Standardmodell für Parallelrechner) geplant haben.

Dabei wird zunächst ein in einer Assembler-ähnlichen Sprache geschriebenes Programm in einen NAND-Gatter-basierten Schaltkreis übersetzt, und dieser dann in DNA-Sequenzen, mit denen man diesen Schaltkreis simulieren kann [Amos97, Amos98]. Es ist jedoch fraglich, ob es Sinn macht, Berechnungen im DNA-Computing auf der Architektur Silizium-basierter Rechner aufzubauen, da die Art der Berechnung im DNA-Computing nicht auf Transistoren und deren Schalterfunktion beruht.

Winfree, Yang und Seeman haben sich ohne den Umweg über die Boole'sche Logik mit der universellen Anwendbarkeit des Verhaltens von DNA beschäftigt, und dabei die Berechnungskraft des *self-assembly* theoretisch untersucht [Winfree96]. Neben der erfreulichen Entdeckung eines auf einem DNA-Gitter basierenden Modells mit universeller Rechenkraft ist ein interessantes Zwischenergebnis ihrer Arbeit, daß sich mit dem sog. linearen *self-assembly* unter Ausschluß von *hairpin loops* genau die regulären Sprachen erzeugen lassen. An dieser Stelle verfolgen Winfree et al. diesen Gedanken jedoch nicht weiter, da ihnen die Anwendungsmöglichkeiten des linearen *self-assembly* unzureichend erscheinen, sondern konzentrieren sich auf ihr Gittermodell mit universeller Rechenkraft. Insbesondere wird nicht einmal ein mögliches Ausleseverfahren aufgezeigt. Inzwischen wurde aber gezeigt, daß das lineare *self-assembly*, insbesondere in Verbindung mit anderen Techniken wie z. B. PCR, durchaus weiterreichende Anwendungsmöglichkeiten bietet, die über die Initialisierung von DNA-Mengen von Suchalgorithmen hinausgehen [Rauhe99, LeierRichter99] (s. auch 3.3).

Das in dieser Arbeit vorgestellte Konzept des programmierten *self-assembly* ist ein lineares *self-assembly*, zur Vereinfachung wird hier meist der Zusatz der Linearität weggelassen.

Der im Rahmen dieser Arbeit entwickelte DNA-Sequenz-Compiler unterstützt die Vielfältigkeit und Einsatzfähigkeit des programmierten *self-assembly* durch seine Eigenschaft als Programmierwerkzeug, die Automatisierung der Code-Erzeugung und die Minimierung von Fehlerwahrscheinlichkeiten im eigentlichen *self-assembly*-Schritt sowie in anschließend angewandten weiterführenden Techniken.

3.1 Die Eingabe: Reguläre Grammatiken

[Wegener93]

Regelsysteme, mit denen genau die Wörter einer bestimmten Sprache erzeugt werden, heißen Grammatiken. Z. B. kann man mit einer Grammatik syntaktisch korrekte Programme einer bestimmten Programmiersprache erzeugen.

Der Linguist Chomsky formulierte 1953 die gebräuchliche Definition von Grammatiken:

Eine *Grammatik* G ist ein Quadrupel (Σ, V, S, P) mit

- Σ eine endliche Menge von Symbolen (*Terminalen*), das Alphabet
- V eine endliche, zu Σ disjunkte Menge von Hilfszeichen (*Variablen*)
- $S \in V$ das *Startsymbol*
- P eine endliche Menge von *Regeln* bzw. *Produktionen*. Eine solche Regel ist ein Paar (l, r) mit $l \in (V \cup T)^+$ und $r \in (V \cup T)^*$ (A^+ und A^* bezeichnen die Mengen der möglichen Ketten von Symbolen aus A, wobei in A^* die Kette der Länge Null vorkommen darf, in A^+ nicht). Kommt l als Teilwort in einem Wort vor, so wird bei Anwendung der Regel l durch r werden. Man schreibt auch $l \rightarrow r$.

Man nennt die Menge aller Ausdrücke (Wörter), die sich mit den Regeln aus P in endlich vielen Schritten aus S ableiten lassen, die von der Grammatik G erzeugte Sprache L(G).

Sei z. B. G = ({a}, {S}, S, {$S \rightarrow aS$, $S \rightarrow a$}), so ist L(G) = {a, aa, aaa, ...}.

Weiterhin hat Chomsky die Grammatiken in eine Hierarchie mit vier Klassen eingeteilt:

0. Grammatiken ohne weitere Einschränkungen heißen Grammatiken vom Typ Chomsky-0.
1. Grammatiken, bei denen alle Regeln die Form $u \rightarrow v$ mit $u \in V^+$, $v \in ((V \cup T) - \{S\})^+$ und $|u| \leq |v|$ oder $S \rightarrow \varepsilon$ haben, heißen *kontextsensitiv* oder Grammatiken vom Typ Chomsky-1. (ε repräsentiert hierbei das leere Wort, d. h. das Wort der Länge 0)
2. Grammatiken, bei denen alle Regeln die Form $A \rightarrow v$ mit $A \in V$ und $v \in (V \cup T)^*$ haben, heißen *kontextfrei* oder Grammatiken vom Typ Chomsky-2.
3. Grammatiken, bei denen alle Regeln die Form $A \rightarrow v$ mit $A \in V$ und $v = \varepsilon$ oder $v = aB$ mit $a \in T$ und $B \in V$ haben, heißen *rechtslinear*, *regulär* oder Grammatiken vom Typ Chomsky-3.

Da im Rahmen dieser Arbeit nur die Chomsky-3-Grammatiken von Interesse sind, werde ich auch nur diese hier etwas genauer betrachten. Zu den anderen Grammatiken siehe [Wegener93].

Die Bezeichnung rechtslinear läßt sich einfach aus den erlaubten Produktionsregeln ersehen. Mit den Ersetzungen wachsen die Ausdrücke am rechten Ende weiter, während der Rest fixiert bleibt. Die Bezeichnung regulär bezieht sich darauf, daß die von Chomsky-3-Grammatiken erzeugten Sprachen genau die sind, die von endlichen Automaten akzeptiert werden.

Obwohl reguläre Grammatiken zu schwach sind, um arithmetische Ausdrücke zu bilden oder eine Programmiersprache zu beschreiben (Rekursionen, wie sie z. B. für Klammerausdrücke notwendig sind, sind nicht möglich), haben sie dennoch eine Bedeutung in der Übersetzung von Programmiersprachen. In der sog. lexikalischen Analyse müssen Teilstrings des Programmcode-Strings erkannt werden, die einen Variablennamen, eine Zahl o. ä. darstellen. Um diese Teilstrings zu erkennen und mit einem ihrem Typ entsprechenden Token zu versehen, wird ein endlicher Automat verwendet, da sie einfach genug sind, um mit einer regulären Grammatik dargestellt werden zu können.

Im Ansatz des programmierten *self-assembly* geht es umgekehrt darum, reguläre Ausdrücke zu erzeugen anstatt sie zu erkennen. Aber auch hier ist eine der einfachen Anwendungen die Erzeugung von Zahlen bzw. Bitstrings. Jedoch können reguläre Grammatiken im Rahmen des programmierten *self-assembly* auch für andere Anwendungen genutzt werden, wie in den Kapiteln 3.3 und 4.2.7 beschrieben.

Die hier verwendeten regulären Grammatiken unterliegen im Hinblick auf die Verarbeitung der Ausdrücke im DNA-Computing gewissen Einschränkungen:

- Die Startvariable S darf nur auf der linken Seite einer Regel, also als zu ersetzende Variable vorkommen.
- In den Startregeln (Regeln, in denen S vorkommt), darf nur ein ausgezeichnetes Terminal, das Startterminal s, vorkommen.
- Anstatt die Ausdrucksbildung durch eine Regel der Form $A \rightarrow \varepsilon$ zu beenden, geschieht dies durch Regeln der Form $A \rightarrow e$ mit $A \in V$ und $e \in T$. Dadurch können keine leeren Wörter gebildet werden.
- Es gibt ein weiteres ausgezeichnetes Terminal, das Endterminal e. Dieses ist das einzige Terminal, das in Regeln ohne Variable auf der rechten Seite (Endregeln) vorkommen darf.

Damit ist der kleinste mögliche Ausdruck „se“, der in der Anwendung durch die besondere Rolle von s und e als leeres Wort gilt.

Terminale, die weder Start- noch Endterminal sind, heißen in dieser Arbeit auch *echte Terminale*. Sollten S, s oder e für andere Zwecke in der Grammatik bestimmt sein, so kann man im Compiler auch andere Zeichen für diese Rolle wählen.

Auf die Hintergründe dieser Einschränkungen wird im nächsten Abschnitt genauer eingegangen.

Dem DNA-Sequenz-Compiler genügt für die Eingabe einer Grammatik die Regelmenge (und ggf. die Zeichen, die statt S, s und e als Sonderzeichen gelten sollen), da aufgrund der einfachen Form der Regeln klar ist, welche Zeichen darin Terminale und welche Variablen sind.

3.2 Die Ausgabe: Algomere

Terminale und Variablen einer (regulären) Grammatik werden beim programmierten *self-assembly* durch Oligomere repräsentiert. Die Regeln werden durch die besondere Anordnung dieser Oligomere zu doppelsträngigen Strukturen, den *Algomeren*, implementiert.

Eine Regel einer regulären Grammatik besteht aus einer zu ersetzenden Variablen, dem neuen Terminal und ggf. einer weiteren, neuen Variablen. Dem entspricht der Aufbau der Algomere.

Das einer Regel A $\rightarrow$ xB entsprechende Algomer besitzt in der Mitte einen doppelsträngigen Teil, der das Terminal x darstellt, sowie an den Enden einsträngige Überhangsequenzen (sog. *sticky ends*), die die Variablen repräsentieren. Dabei steht die linke Überhangsequenz für die zu ersetzenden Variable A und die rechte (am unteren Strang befindliche und daher komplementäre) für die neue Variable B. Diese Variablen können natürlich auch identisch sein.

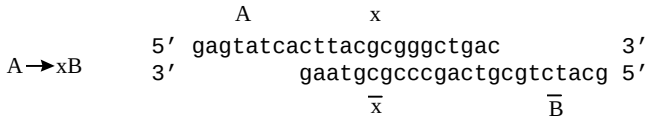


Abbildung 3.1: Regel und deren Repräsentation als Algomer. Die doppelsträngige Kernsequenz bildet das Terminal, die Überhänge stellen die Variablen dar. Überstrichene Buchstaben bezeichnen Komplementärsequenzen.

Gemäß der in Kapitel 3.1 beschriebenen Einschränkungen der hier verwendeten regulären Grammatiken, haben die Start- und Endregeln die Form S $\rightarrow$ sA bzw. A $\rightarrow$ e (A steht hierbei für eine beliebige Variable). Die entsprechenden Algomere haben ebenfalls eine besondere Form:

Startregeln werden zu Algomeren übersetzt, bei denen der linke einzelsträngige Teil nicht die zu ersetzende Variable S kodiert[1], sondern eine spezielle Überhangsequenz. Diese stellt kein Element der Grammatik dar, sondern kann z. B. eine Restriktionsschnittstelle sein, mit der man die vollständigen Sequenzen in größere Sequenzen (z. B. Klonierungsvektoren[2]) einfügen und

[1] Daher, und weil S nirgendwo anders vorkommt, erzeugt der Compiler auch keine Sequenz, die die Startvariable repräsentiert.

[2] Fügt man die Logomere z. B. in bakterielle Plasmid-DNA ein und bringt diese wieder in Bakterien ein, so kann man die Vermehrung der Bakterien zur Vervielfältigung der Logomere nutzen.

später mit Restriktionsenzymen wieder ausschneiden kann. Der rechte Einzelstrang repräsentiert die Variable A, der doppelsträngige Teil das Startterminal s.

Algomere, die Endregeln repräsentieren, haben ebenfalls auf beiden Seiten einzelsträngige *sticky ends*. Hier ist die rechte ssDNA-Sequenz eine spezielle Überhangsequenz, während die linke die Variable A darstellt und der doppelsträngige Teil wiederum das Endterminal e.

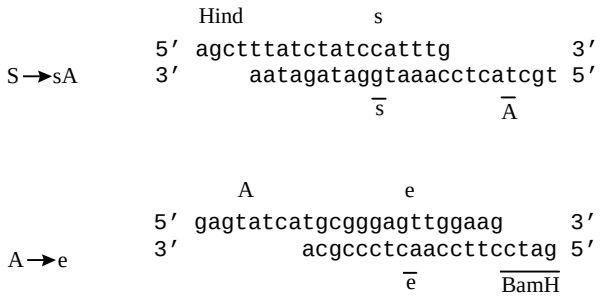


Abbildung 3.2: Start- und Endregel, entsprechende Algomere. Jeweils eine Überhangsequenz ist keine Variable, sondern z. B. eine Restriktionsschnittstelle, hier für die Enzyme HindIII und BamHI. Überstrichene Buchstaben bezeichnen Komplementärsequenzen.

Gibt man nun die Algomere aller Regeln in ein Reagenzglas und kühlt die Lösung unter die Schmelztemperatur der Variablensequenzen ab, so hybridisieren diese *sticky ends*. Es bilden sich längere dsDNA-Sequenzen, die ganze Ausdrücke (oder Wörter) der Grammatik repräsentieren und daher *Logomere* (von griech. $\lambda o \gamma o \varsigma$ = Wort) genannt werden.

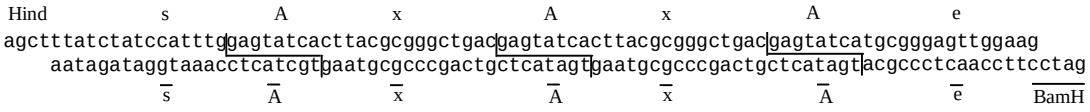


Abbildung 3.3: Hybridisierung der Algomere zum Logomer, der den Ausdruck „xx“ darstellt.

Algomere, die das Start- oder Endterminal (bzw. die entsprechende Sequenz) enthalten, heißen *Terminatoren*, und bilden Anfang und Ende der Logomere.

Die Start- und Endterminalsequenzen selber haben verschiedene praktische Bedeutungen:

- Sie können für einen Extraktionsschritt als Identifikator für Logomere einer bestimmten Grammatik dienen.
- In der PCR können sie als Primer verwendet werden, um die Logomere einer bestimmten Grammatik zu vervielfältigen. Neben der einfachen Vermehrung der Sequenzen für weitere Rechenschritte im DNA-Computing kann auch dies zur Identifikation und sogar zum Auslesen von Logomeren verwendet werden [Rauhe99] (s. Kap. 3.3).
- Auch in der Sequenzierung können sie als Primer verwendet werden.

Der DNA-Sequenz-Compiler erlaubt auch die Erzeugung mehrerer verschiedener Sequenzen zur Repräsentation von s bzw. e, so daß gleiche Ausdrücke verschiedene Rollen in der Anwendung zugewiesen bekommen können. Für ein Beispiel siehe Kapitel 3.3.

Die übrigen Algomere heißen *Elongatoren*, da sie die Regeln repräsentieren, die den Ausdruck jeweils um ein Terminal verlängern.

Eine wesentliche Eigenschaft der Logomere besteht darin, daß die Variablensequenzen nicht wirklich ersetzt werden wie die Variablen einer Grammatik, sondern sie bleiben zwischen den Terminalsequenzen erhalten.

Tatsächlich sind sie nämlich für eine korrekte Ausdrucksbildung unverzichtbar. Eine Kodierung ohne Variablen, wie sie z. B. von Adleman verwendet wurde[1], ist nicht mächtig genug, beliebige reguläre Grammatiken zu repräsentieren. Sei z. B. R = {S $\rightarrow$ aA, A $\rightarrow$ xB, B $\rightarrow$ b, S $\rightarrow$ cC, C $\rightarrow$ xD, D $\rightarrow$ d}, so ist L(G) = {axb, cxd}. Verwendet man nun in der Adleman'schen Kodierung die Knoten als Terminale und die Kanten als Variablen[2], so können an der x-Knoten-Sequenz rechts sowohl xb-Kanten als auch xd-Kanten anlagern, unabhängig davon, ob links eine ax-Kante oder eine cx-Kante angelagert ist. Es würden also auch Sequenzen für die Ausdrücke axd und cxb entstehen, die nicht in L(G) enthalten sind.

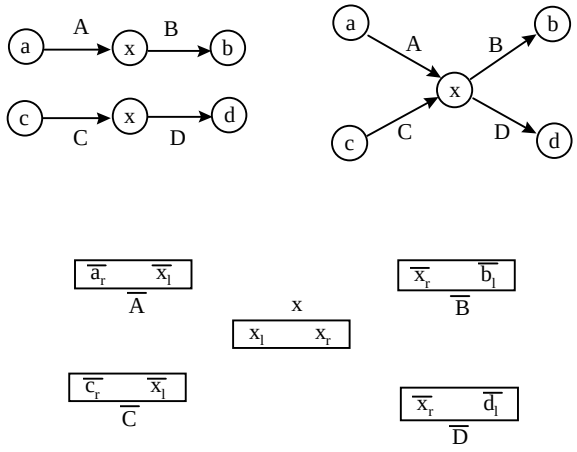


Abbildung 3.4: Oben: Graph für die zu übersetzende Grammatik G (links), Graph der *in vitro* tatsächlich erzeugten Grammatik (rechts). Unten: Sequenzen für die Kanten (Variablen) und den Knoten für das Terminal x. Überstrichene Buchstaben bezeichnen Komplementärsequenzen.

[1] Gerichtete Pfade durch einen Graphen lassen sich auch durch eine reguläre Grammatik beschreiben (s. 3.3 und 5.4).

[2] Kaplan et al. haben in [Kaplan96] die Knoten als Wörter und die Kanten als Regeln einer Grammatik, die diese Wörter zusammensetzt, bezeichnet. Zu ihren Gunsten nehme ich an, daß sie damit keine formale Grammatik im Chomsky'schen Sinn meinten.

Die Verwendung von Kanten als Terminale und Knoten als Variablen ändert nichts an dem Problem, da dieser Graph dem Zustandsgraphen des akzeptierenden Automaten mit den Variablen als Zuständen entspräche und man hier zwei verschiedene Übergänge (Kanten) mit der Eingabe x hätte, es gäbe dann also zwei verschiedene Sequenzen zur die Kodierung des Symbols x.

Um das Analogon der endlichen Automaten heranzuziehen: die Variablen sind notwendig, da das zuletzt eingelesene Terminal alleine nicht ausreicht, um einen Zustand des akzeptierenden Automaten eindeutig zu identifizieren.

An die Sequenzen, die die Terminale und Variablen repräsentieren, werden bestimmte Anforderungen gestellt. Sie sollten nicht nur im lexikalischen Sinne eindeutig sein, sondern einander möglichst unähnlich, sie sollten bei der Hybridisierung ein bestimmtes Verhalten zeigen usw. Diese Anforderungen und die Methoden, ihnen gerecht zu werden, werden in Kapitel 4.1 näher beschrieben.

Die Algomere werden jeweils in Form der beiden Einzelstränge ausgegeben, aus denen sie sich zusammensetzen, so daß diese Oligonucleotide direkt synthetisiert bzw. bei einem Synthetisierer bestellt werden können.

3.3 Anwendung: Programmiertes self-assembly

Die Bezeichnung „programmiertes *self-assembly*“ bezieht sich darauf, daß sich die Algomere „selbständig“ im Reagenzglas zu Logomeren zusammenfügen. Die Programmierung erfolgt durch die Angabe einer regulären Grammatik, deren Regeln bestimmen, in welcher Weise sich die Algomere aneinanderfügen dürfen.

3.3.1 Überblick über das Verfahren

Das Verfahren läßt sich in folgende Schritte aufteilen:

- **Grammatik**. Zunächst muß eine reguläre Grammatik, die die gewünschten Ausdrücke erzeugt, erstellt werden. Dies ist die Programmierung des *self-assembly*-Vorgangs.
- **Übersetzung**. Die Regeln der Grammatik werden durch den DNA-Sequenz-Compiler in Algomere übersetzt. Diese müssen anschließend synthetisiert werden.
- **Die Hybridisierung der Algomere**. Die synthetisierten bzw. vom Synthetisierer gelieferten ssDNA-Stränge müssen zunächst zu dsDNA-Algomeren hybridisieren. Hierbei müssen die Algomere getrennt erzeugt werden, da es sonst zu falschen Anlagerungen und

damit zur Bildung nicht vorgegebener Regeln kommen kann (z. B. könnten aus den Algomeren für die Regeln A → aB und C → aD Algomere für A → aD und C → aB werden).

- **Der eigentliche *self-assembly*-Schritt**. Die Algomere werden in einem Reagenzglas vermengt und abgekühlt, so daß sie sich per Hybridisierung der *sticky ends* zu Logomeren verbinden. Gleichzeitig werden die Lücken im Rückgrat der Logomere mit Hilfe von Ligase geschlossen.
- **Weitere Verarbeitungsschritte**. Nach dem *self-assembly* kann man die Logomere weiterverarbeiten, wie für die jeweilige Anwendung nötig. Z. B. können die Operationen eines der in 2.2 vorgestellten Modelle durchgeführt werden, um für die Lösung eines Problems ungeeignete Sequenzen auszusortieren. Im folgenden werden einige Anwendungsmöglichkeiten vorgestellt.

3.3.2 Anwendungen

Rauhe hat einige Anwendungen für Logomere, die Bitstrings darstellen, vorgestellt [Rauhe99]:

3.3.2.1 Zufallszahlengenerator

Mit der Regelmenge R = {S → sA, A → 0A, A → 1A, A → e} läßt sich ein Zufallszahlengenerator implementieren.

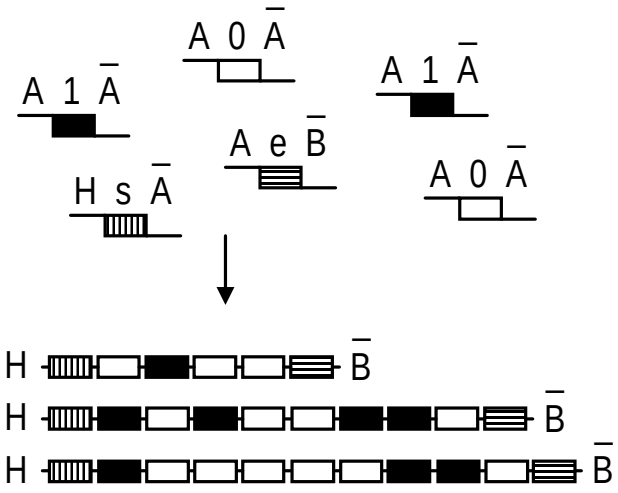


Abbildung 3.5: Zufallszahlengenerator. Die Elongatoren, die Nullen und Einsen darstellen, bilden zufällige Bitsequenzen, die mit Terminatoren abgeschlossen werden.

Im *self-assembly*-Schritt werden Bitstrings verschiedener Länge gebildet. Die Leistung läßt sich durchaus mit der konventioneller Pseudozufallszahlengeneratoren auf modernen PCs vergleichen und hängt (in vernünftigen Grenzen) von der Menge der verwendeten DNA ab.

Es lassen sich nicht Bitstrings beliebiger Länge erzeugen. Grob kann man die Wahrscheinlichkeiten für das Auftreten der Logomerlängen durch das Mengenverhältnis Terminatoren zu Elongatoren beeinflussen. Je mehr Terminatoren vorhanden sind, desto höher ist die Wahrscheinlichkeit, daß ein Elongator einen Terminator statt eines weiteren Elongators als Hybridisierungspartner bekommt, die zu erwartende durchschnittliche Länge der Logomere wird also geringer.

3.3.2.2 Bytes

Man kann natürlich auch Bitstrings einer festen Länge generieren, z. B. Bytes. Die Regelmenge hierfür ist

$R = \{S \rightarrow sS_0,$

$S_0 \rightarrow 0S_1,\ S_0 \rightarrow 1S_1,$

$S_1 \rightarrow 0S_2,\ S_1 \rightarrow 1S_2,$

...

$S_7 \rightarrow 0S_8,\ S_7 \rightarrow 1S_8,$

$S_8 \rightarrow e\}.$

Wie leicht zu sehen ist, sind die Ausdrücke aus L(G) genau acht Terminale lang (plus Start- und Endterminal), an jeder Bitposition steht also eine 0 oder 1.

Gegenüber der Bitstring-Kodierung von Lipton [Lipton95] (s. 2.2.2) hat diese Kodierung zwei wesentliche Vorteile:

- Man benötigt für die Darstellung der Bits nur zwei Sequenzen (anstatt zwei für jede Bitposition, also 16 Sequenzen), man ist also nur auf eine wesentlich geringere Ausbeute des Generators angewiesen.
- Diese Kodierung erlaubt ein einfaches Auslesen der Bitbelegung per PCR. Verteilt man die Byte-Logomere auf zwei Reagenzgläser und gibt in das eine die Komplementärsequenz zur 0, in das andere die Komplementärsequenz zur 1, in beide die Startterminalsequenz als Primer, und führt mit beiden eine PCR durch, so verstärkt man verschieden lange Logomerteile, die jeweils mit einer 0 bzw. im anderen Glas mit einer 1 enden. Trägt man die Sequenzen nun in zwei Bahnen auf ein Gel auf und führt eine Gelelektrophorese durch, so werden sie nach der Länge getrennt. Da die Länge der Logomerteile mit der Bitposition des letzten Bits korreliert, läßt sich für jede Bitposition ablesen, ob dort eine 0 oder eine 1 steht, je nachdem in welcher Bahn sich die entsprechende Bande befindet.

Die Byte-Grammatik läßt sich leicht zu einer n-Byte-Grammatik erweitern. Man muß dazu nicht für jedes Byte 16 neue Regeln in R aufnehmen (zum Anhängen von 0 und 1 für jeweils acht Positionen), sondern man kann den DNA-Sequenz-Compiler n verschiedene Sequenzen für das Startterminal s (und, je nach weiterer Anwendung, auch für das Endterminal e) erzeugen lassen. Dadurch haben die n Byte-Logomere den gleichen Aufbau mit den gleichen Sequenzen für Bits und Variablen, können aber anhand der Startsequenzen eindeutig der entsprechenden Byteposition zugeordnet werden.

Gegenüber der Erweiterung von R um neue Regeln hat diese Vorgehensweise zwei wesentliche Vorteile:

- Da die Variablensequenzen in den verschiedenen Bytes gleich sind, wird V nicht vergrößert, es ist also nicht notwendig, weitere Sequenzen für Variablen zu erzeugen, wie es bei neuen Regeln der Fall wäre. Dies spart nicht nur Kosten für die Synthetisierung von Oligomeren, sondern ist vor allem für eine erfolgreiche Übersetzung wichtig, da die Ausbeute an Sequenzen einer bestimmten Länge geringer wird, je höher die Anforderungen (wie *uniqueness*, Schmelztemperatur usw.) an diese sind.
- Kürzere Logomere sind im Labor einfacher zu handhaben. So bilden z. B. zu lange Sequenzen sog. Sekundärstrukturen, d. h. anstatt annähernd linearer Stränge bilden sich dreidimensionale Strukturen, die in den Operatoren des DNA-Computing hinderlich und somit neue Fehlerquellen sein können. Außerdem können lange DNA-Sequenzen „brechen“ und somit unbrauchbar werden, oder die Lösung zu viskos machen.

Außer für klassische DNA-Computing-Anwendungen kann man diese n-Byte-Sequenzen auch anderweitig verwenden. Z. B. lassen sie sich als eine Art Seriennummer sowohl flüssigen Stoffen wie Lack, Tinte, Öl usw. beimischen als auch auf feste Stoffe wie Dokumente oder Papiergeld aufbringen und später wieder zur Identifikation extrahieren und auslesen. Hierzu können die Bytes auch Zeichen kodieren, wie z. B. mit dem ASCII.

3.3.2.3 Steganographie

Mit *Steganographie* bezeichnet man das Verstecken geheimer Nachrichten in „unverfänglichen“ Umgebungen.

In [Clelland99] und [LeierRichter99] werden steganographische Verschlüsselungsverfahren unter Einsatz von DNA-Sequenzen gezeigt. Der DNA-Sequenz-Compiler kann verwendet werden, um die von [LeierRichter99] benutzte Binärkodierung der Nachrichten zu erzeugen,

insbesondere die für das Steganographieverfahren benötigten unterschiedlichen Start- und Endterminalsequenzen.

3.3.2.4 „Klassische“ Anwendungen

Natürlich kann man das programmierte *self-assembly* auch für die klassischen Ansätze des DNA-Computing zur Lösung NP-harter Probleme verwenden. So kann man z. B. in Adlemans Verfahren zur Lösung des HPP Logomere verwenden, um Pfade in einem gerichteten Graphen herzustellen (s. 5.4). Hierzu sieht man z. B. die Knoten des Graphen als Terminale und die Kanten als Variablen an. Damit hat jeder Knoten für jedes Paar aus ein- und ausgehender Kante eine Regel. Man kann sogar Schritt 2 aus Adlemans Lösungsalgorithmus (s. 2.2) einsparen, indem man die Grammatik so wählt, daß das erste bzw. letzte Terminal (das nicht s oder e ist) eines jeden Ausdrucks gerade dem Start- bzw. Endknoten des zu suchenden Hamilton-Pfades entspricht. Somit wird der Lösungsraum schon bei der Initialisierung (Schritt 1) eingeschränkt.

4 Die Implementierung des Compilers

4.1 Der Sequenzgenerator

Der Sequenzgenerator ist ein im Rahmen dieser Arbeit entwickeltes Softwaremodul, das DNA-Sequenzen erzeugt, die die im Folgenden erläuterten Anforderungen erfüllen, um die Fehlerwahrscheinlichkeit des DNA-Computing möglichst gering zu halten. Diese Anforderungen werden z. B. von zufällig generierten Sequenzen nicht erfüllt. Daher verwendet der Generator das in diesem Kapitel beschriebene Verfahren zur gezielten Konstruktion geeigneter Sequenzen.

Neben seiner allgemeinen Verwendbarkeit als eigenständiges Werkzeug im DNA-Computing bildet der Generator ein wesentliches Modul des Compilers, da er die DNA-Sequenzen liefert, die der Compiler den Variablen und Terminalen zuordnet.

4.1.1 Uniqueness

Eine große Fehlerquelle im DNA-Computing sind Fehlhybridisierungen, d. h. Hybridisierungen von Sequenzen, die gar nicht hybridisieren sollen. Möglich ist dies dadurch, daß für eine Hybridisierung keine perfekte Komplementarität notwendig ist. Werden zwei Operanden durch Sequenzen repräsentiert, die sich zwar an mindestens einer Stelle unterscheiden (im klassischen Sinne also eindeutig sind), aber nur an wenigen Stellen, so daß sie einander sehr ähnlich sind, so sind sie unter Umständen beim Extraktionsvorgang, bei der Anlagerung von Primern zur PCR oder auch beim programmierten *self-assembly* für die Hybridisierung nicht unterscheidbar.

Für die Minimierung von Fehlern im DNA-Computing ist es daher unerläßlich, daß die verwendeten DNA-Sequenzen sowie deren Komplementärsequenzen einander möglichst unähnlich sind und somit die Wahrscheinlichkeit für Fehlhybridisierungen zumindest möglichst gering ist. Diese Eigenschaft ist es auch, die im DNA-Computing als Eindeutigkeit (*uniqueness*) bezeichnet wird.

Als Maß für die (Un-)Ähnlichkeit ist die (für Informatiker naheliegende) Hamming-Distanz[1] ungeeignet, da sie keine *shifts*, *hairpin loops* oder *bulges* berücksichtigt. Man benötigt also einen *uniqueness*-Begriff, der auch diese Besonderheiten des DNA-Computing einschließt.

[1] Die Hamming-Distanz zwischen zwei Zeichenketten ist gleich der Anzahl der Stellen, an denen sie verschiedene Zeichen besitzen.

Im Rahmen dieser Arbeit wird die *uniqueness* von Sequenzen im folgenden, von Niehaus [Niehaus98] vorgeschlagenen Sinne verstanden:

Anstatt die Nucleotide A, C, G und T als Alphabet für die Sequenzen zu verwenden, benutzt man sogenannte *Basissequenzen*. Diese sind selbst DNA-Sequenzen einer festen Länge *BL*. Um eine gesamte Sequenz zu bilden, werden sie aber nicht einfach hintereinander gesetzt, sondern je zwei aufeinanderfolgende Basissequenzen überlappen sich an *BL* - 1 Stellen.

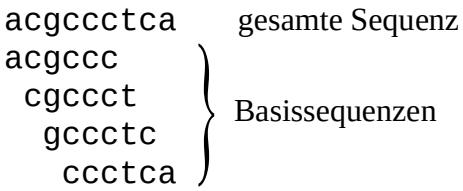


Abbildung 4.1: Vier Basissequenzen der Länge 6 bilden durch Überlappen eine Gesamtsequenz der Länge 9. Allgemein besteht ein Sequenz der Länge SL aus *n* Basissequenzen der Länge *BL* mit $n = SL - BL + 1$.

Fordert man nun, daß jede Basissequenz einer bestimmten Länge *BL* höchstens einmal in allen verwendeten Sequenzen vorkommt, so erhält man eine *uniqueness* in dem Sinne, daß zwei beliebige Sequenzen gemeinsame Subsequenzen höchstens der Länge *BL* - 1 haben können. Außerdem bildet jede Basissequenz mit seinem Komplement eine Äquivalenzklasse, so daß auch die Komplemente verwendeter Basissequenzen nicht vorkommen dürfen. Als logische Konsequenz wird die Verwendung selbstkomplementärer Basissequenzen verboten, also solcher Basissequenzen, die mit ihrem Komplement identisch sind.

Somit sind Sequenzbereiche, in denen es zu Fehlhybridisierungen (ohne *mismatches*) kommen kann, kürzer als die Basissequenzlänge, was nicht nur die Wahrscheinlichkeit für das Zusammentreffen dieser Subsequenzen, sondern vor allem die thermodynamische Stabilität dieser Anlagerungen verringert, so daß sie sich erst gar nicht bilden oder aber die beabsichtigten Hybridisierungen energetisch deutlich günstigere Zustände darstellen und daher bevorzugt werden.

Diese *uniqueness* (bzw. der entscheidende Parameter *BL*) läßt sich nicht nur einfach messen, sie bietet vor allem für die Konstruktion geeigneter Sequenzen ein praktisches Werkzeug (s. 4.1.2). Neben der Basissequenzlänge ist eine weitere wichtige Größe der Grad, mit dem man mehrfaches Vorkommen von Basissequenzen toleriert (s. 4.2.4).

Neben der unzulänglichen Hamming-Distanz bzw. der darauf aufbauenden H-Distanz[1] ist das Vorkommen gemeinsamer Subsequenzen das häufigste Maß für eindeutige Sequenzen in der Literatur [z. B. Seeman90, Deaton97, Baum95]. Niehaus hat exemplarisch am HPP gezeigt, daß die mit dem im Folgenden beschriebenen Verfahren erzeugten Sequenzen in der Simulation weniger unerwünschte Anlagerungen zeigen als zufällig gewählte (wie in Adlemans Experiment) oder mit maximaler H-Distanz gewählte Sequenzen (wie in der Wiederholung des Experiments von Deaton et al.) [Niehaus98].

Allerdings hat auch dieser *uniqueness*-Begriff einen Nachteil: Betrachtet man z. B. die beiden Sequenzen

S_1 = ACGTGAGTGCA und

S_2 = ACGTGCGTGCA,

so sind diese bezüglich einer Basissequenzlänge von 6 *unique*, alle Basissequenzen dieser Länge von S_1 unterscheiden sich in genau einer Stelle (der unterstrichenen) von denen von S_2. Trotzdem sind sich die Sequenzen sehr ähnlich, und die Komplementärsequenz zu S_1 (TGCACTCACGT) würde sich sehr wahrscheinlich auch an S_2 anlagern.

Dennoch ist diese Auffassung von *uniqueness* z. Z. diejenige, die den möglichen Fehlhybridisierungen in der Implementierung *in vitro* am ehesten gerecht wird, weshalb sie in dieser Arbeit verwendet wird. Um das geschilderte Problem zu berücksichtigen, kann dem Compiler auch eine maximale Homologie der Sequenzen zueinander angegeben werden (s. 4.1.3.2).

4.1.2 Das Niehaus-Verfahren

Jens Niehaus hat in [Niehaus98] beschrieben, wie man den *uniqueness*-Begriff mit Basissequenzen konstruktiv zur Erzeugung uniquer Sequenzen verwenden kann[2].

Bei diesem Verfahren sind die Basissequenzen einer bestimmten Länge *BL* die Knoten eines gerichteten Graphen. Jeder Knoten *k* hat als Nachfolger genau die vier Basissequenzen, die in einer DNA-Sequenz auf diese Basissequenz folgen können, d. h. die Basissequenzen, deren *BL* - 1 ersten Nucleotide mit den *BL* - 1 letzten Nucleotiden von *k* identisch sind.

---

[1] Die H-Distanz ist die maximale Hamming-Distanz der gegenüberliegenden Teilsequenzen im doppelsträngigen Bereich über alle möglichen Verschiebungen zweier Sequenzen zueinander [Deaton97].

[2] Seeman hat ein anderes, heuristisches Verfahren entwickelt, das auf einem ähnlichen *uniqueness*-Begriff basiert, das aber anstatt Sequenzen zu konstruieren vorgegebene Sequenzen so lange repariert, bis sie die *uniqueness*-Anforderung erfüllen [Seeman83, Seeman90].

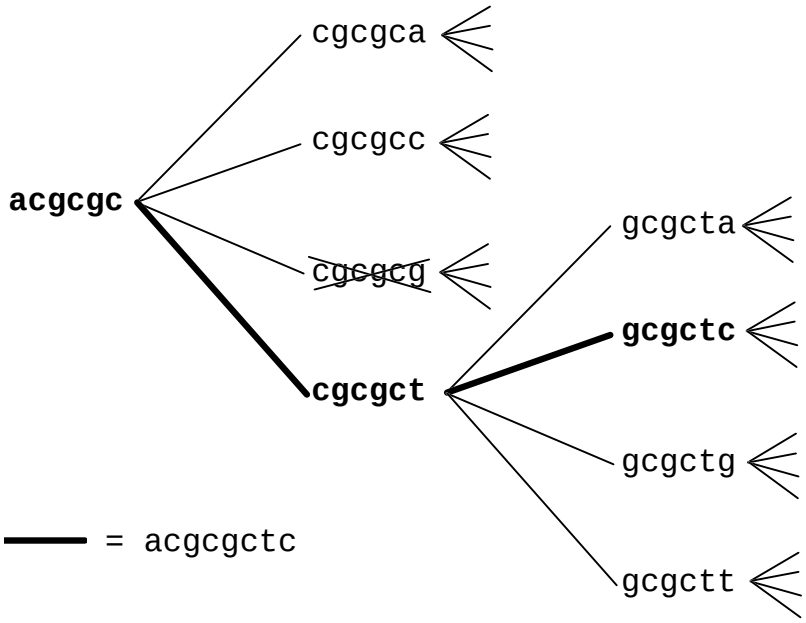


Abbildung 4.2: Ausschnitt aus dem Graph der Basissequenzen der Länge 6. Jede Basissequenz hat die vier Basissequenzen als Nachfolger, die in einer Gesamtsequenz auf sie folgen können. Sequenzen bilden somit einen Pfad durch den Graphen. Als Beispiel ist der Pfad für die Sequenz `ACGCGCTC` hervorgehoben. Die Basissequenz `CGCGCG` ist selbstkomplementär und daher deren Verwendung verboten.

Eine DNA-Sequenz der Länge *SL* kann somit als Pfad der Länge (*SL* - *BL* +1) durch den Graph der Basissequenzen betrachtet werden. *Unique* Sequenzen sind somit Pfade, die keine gemeinsamen Knoten haben.

Um eine Menge solcher Pfade zu finden, hat Niehaus folgenden Backtracking-Algorithmus vorgestellt:

1. Erzeuge den Graphen aus allen Basissequenzen, die nicht selbstkomplementär sind.
2. Kennzeichne alle Knoten als unbenutzt und als unbenutzte Anfangsknoten.
3. Solange Knoten j existiert, der noch nicht als benutzter Anfangsknoten markiert ist
   - Markiere Knoten j als benutzten Anfangsknoten und sowohl j als auch dessen Komplementärknoten als benutzt.
   - i := 0, Element[0] := j
   - Solange i < Pfadlänge - 1 und i ≥ 0 gilt:
     - Existiert kein unbenutzter Nachfolgeknoten m von Knoten j, so markiere Knoten j und dessen Komplementärknoten als unbenutzt und setze i := i - 1, j := Element[i].

  - Sonst wähle per Zufall einen unbenutzten Nachfolgeknoten m von Knoten j aus und markiere m und dessen Komplementärknoten als benutzt. Setze zusätzlich i := i + 1, j := m, Element[i] := j.

- Wenn i = Pfadlänge - 1 gilt, steht der fertige Strang in Element[0] bis Element[Pfadlänge - 1]; übernehme Strang in Ausgabemenge.

Das Backtracking, das ausgelöst wird, wenn kein unbenutzter Nachfolgeknoten mehr zur Verfügung steht, arbeitet nur pfadweise, d. h. ein bereits fertiggestellter Pfad der gewünschten Länge wird nicht wieder aufgelöst, wodurch die Rechenzeit in Grenzen gehalten wird. Der Algorithmus terminiert also, wenn aus den noch unbenutzten Knoten kein Pfad der gewünschten Länge mehr konstruiert werden kann. (Bei der Verwendung im Compiler wird er abgebrochen, wenn genügend Sequenzen erzeugt wurden.)

Die jeweilige Wahl des nächsten Startknotens bzw. des nächsten unbenutzten Nachfolgers ist zufällig, um eine Neigung zu bestimmten Basen bzw. Basissequenzen zu vermeiden. In Experimenten wurde nie die rechnerisch maximal mögliche Anzahl von Pfaden gefunden, es blieben also immer unbenutzte Knoten übrig, die aber zu „verstreut“ waren, um noch einen weiteren Pfad bilden zu können. Immerhin wurden aber meist über 80 % der maximal möglichen Pfade gefunden (s. 4.2.5.2), was für diese Anwendung reicht, zumal es nicht sicher ist, ob ein anderer Algorithmus (außer vielleicht der indiskutablen vollständigen Enumeration aller Pfadmengen) alle möglichen Pfade oder auch nur mehr Pfade als der hier vorgestellte Algorithmus finden würde. Insbesondere ist zu beachten, daß die hier und in [Niehaus98] verwendete Abschätzung (s. 0) maximal möglicher Sequenzen eher zu hoch greift, da sie nicht berücksichtigt, daß die Basissequenzen, die eine Sequenz bilden, der Einschränkung unterliegen, einander überlappen zu müssen.

4.1.3 Weitere Anforderungen

Über die *uniqueness* hinaus gibt es noch weitere Anforderungen an die zu verwendenden Sequenzen, um Fehler bei der Berechnung *in vitro* zu minimieren (übrigens nicht nur für die Verwendung der Sequenzen im *self-assembly*, sondern auch für andere Anwendungen). Diese Anforderungen, die durch sie vermiedenen Fehler sowie die entsprechenden Methoden des Generators zu ihrer Gewährleistung werden hier beschrieben.

Andere zusätzliche Methoden des Generators, die speziell aufgrund seiner Aufgabe innerhalb des Compilers, Sequenzen für Algomere herzustellen, hinzugefügt werden mußten, werden in 4.2.4 erläutert.

4.1.3.1 Schmelztemperatur

Bei der Hybridisierung der Einzelstränge zu Algomeren, vor allem aber beim anschließenden *annealing* der Algomere zu Logomeren sollten alle Stränge möglichst gleichzeitig, d. h. bei möglichst derselben Temperatur hybridisieren. Abgesehen davon, daß dies das *annealing* vereinfacht und verkürzt, würden die für einen Teil der Sequenzen nötigen tieferen Temperaturen, die für die anderen Sequenzen tiefer als nötig sind, dazu führen, daß sich bei diesen die Wahrscheinlichkeit für die instabileren, fehlerhaften Anlagerungen erhöht.

Weiterhin können unterschiedliche Schmelztemperaturen zu Verschiebungen der Wahrscheinlichkeiten beim *self-assembly* führen. Wenn z. B. in der Grammatik zur Erzeugung von binären Zufallszahlen (s. 3.3.2.1) die *sticky ends* der Algomere, die Einsen anfügen, eine höhere Schmelztemperatur haben als die der Algomere, die Nullen anfügen, so würden diese beim *annealing* auch zuerst hybridisieren und es würden sich Einserketten bilden, an die erst anschließend Nullen angefügt würden.

Schließlich sollten Variablensequenzen deutlich geringere Schmelztemperaturen besitzen als Terminalsequenzen, um die Stabilität der Algomere beim *self-assembly*-Schritt zu gewährleisten.

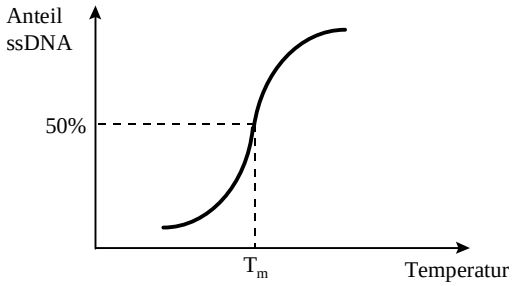


Abbildung 4.3: Verlauf der Denaturierung von dsDNA mit steigender Temperatur. Bei der Schmelztemperatur T_m liegt 50 % der DNA in einsträngiger Form vor.

Die Hybridisierung (und umgekehrt auch die Denaturierung) einer Menge von DNA findet bei Abkühlung (bzw. Erwärmung) in einem gewissen Temperaturbereich statt. Die Temperatur, bei der 50 % der DNA in einsträngiger Form vorliegt, ist die *Schmelztemperatur* (*melting temperature*) T_m. Diese hängt außer von äußeren Einflüssen wie Salzkonzentration usw. insbesondere von der Länge und der Zusammensetzung der DNA ab und soll für die verwendeten Sequenzen möglichst gleich sein. Die Hybridisierungstemperatur ist gleich dieser Schmelztemperatur, da zwischen Schmelz- und Hybridisierungsvorgang keine Hysterese existiert.

Zur Vorhersage der Schmelztemperatur[1] einer DNA-Sequenz gibt es verschiedene Verfahren:

GC-Gehalt

Die einfachste und ungenaueste Methode orientiert sich nur am GC-Gehalt der Sequenz:

$$Ratio_{GC} = \frac{\#GC - Basenpaare}{Sequenzlänge}$$

Auch ohne wirklich die Schmelztemperaturen zu berechnen, kann man recht grob schätzen, daß die Schmelztemperaturen von Sequenzen, die gleichen GC-Gehalt haben, recht nahe beieinander liegen.

Wallace-Methode

Die einfachste Formel, um aus dem GC-Gehalt einer Sequenz ihre Schmelztemperatur abzuschätzen, ist die Wallace-Formel [Genset, Rychlik89]:

T_m = (1 - $Ratio_{GC}$) * Sequenzlänge * 2 ° C + $Ratio_{GC}$ * Sequenzlänge * 4 ° C

oder

T_m = *#AT-Basenpaare* * 2 ° C + *#GC-Basenpaare* * 4 ° C

Tatsächlich ist sie allerdings nur für sehr kurze Oligonucleotide (≤ 20 nt) brauchbar.

%GC-Formel

Eine andere, für längere Oligomere genauere Formel, die die Schmelztemperatur ebenfalls proportional zum GC-Gehalt schätzt, lautet:

$$T_m = A + 41 * Ratio_{GC} - \frac{B}{Sequenzlänge}$$

Leider sind sich die verschiedenen Quellen nicht einig, wie die Werte der Konstanten A und B zu wählen sind. Hier eine Auswahl:

Quelle	[Deaton95]	[www]	[Oligocalc]	[Genosys]
A	81,5	62,3	100,5	81,5
B	600	500	820	500

Tabelle 4.1: Beispielwerte für die Parameter A und B der %GC-Formel.

[1] Zur Vereinfachung der Sprache wird im Folgenden die Schmelztemperatur nicht von der Dissoziationstemperatur T_d unterschieden, die sich aus der Schmelztemperatur und den Korrekturen für Salzkonzentration, Filterbindung usw. ergibt.

***nearest neighbor*-Methode**

Die Schmelztemperatur einer Sequenz hängt tatsächlich nicht nur vom GC-Gehalt ab, sondern auch davon, welche Basen jeweils zueinander benachbart sind, d. h. von der Reihenfolge der Basen in der Sequenz [Breslauer86, Rychlik89, Sugimoto96]. Genauer gesagt werden aufgrund der Reihenfolge der Basen zunächst Enthalpie- und Entropieänderungen ΔH und ΔS beim Dissoziationsvorgang abgeschätzt und aus diesen anschließend die Schmelztemperatur berechnet.

Für jedes der 10 möglichen Paare benachbarter Basenpaare gibt es einen bestimmten ΔH- und einen ΔS-Wert. Diese werden einfach über alle Nachbarpaare einer Sequenz aufsummiert, anschließend werden noch ein Initiierungswert (für das letzte zu lösende Basenpaar) und ggf. eine weitere Entropieänderung für selbstkomplementäre Sequenzen addiert.

Der Compiler kann hierfür wahlweise die Werte aus [Breslauer86] oder [Sugimoto96] verwenden; letztere liefern eine exaktere Approximation von gemessenen Werten. Um neue Sequenzen „kompatibel“ zu alten, mit Werten aus [Breslauer86] erzeugten oder gewählten halten zu können, stehen auch diese Werte noch zur Verfügung.

Nachbarpaar	ΔH [kcal/mol]	ΔS [cal/(K*mol)]
AA TT	-8,0	-21,9
AT TA	-5,6	-15,2
TA AT	-6,6	-18,4
CA GT	-8,2	-21,0
CT GA	-6,6	-16,4
GA CT	-8,8	-23,5
GT CA	-9,4	-25,5
CG GC	-1,8	-29,0
GC CG	-10,5	-26,4
GG CC	-10,9	-28,4
Initiierung	0,6	-9,0
selbstkompl.	0,0	-1,4

Tabelle 4.2: Enthalpien (ΔH) und Entropien (ΔS) für *nearest neighbor*-Paare sowie für Initiierung und selbstkomplementäre Oligomere. Die dsDNA-Paare können auch um 180° gedreht gelesen werden. Quelle: [Sugimoto96]

Bsp.: ΔH und ΔS für die Sequenz GGAATTCC berechnen sich wie folgt:

$\Delta H = \Delta H(GG) + \Delta H(GA) + \Delta H(AA) + \Delta H(AT) + \Delta H(TT) + \Delta H(TC) + \Delta H(CC) + \Delta H_{init}$

= -10,9 - 8,8 - 8,0 - 5,6 - 8,0 - 8,8 - 10,9 + 0,6 kcal / mol

= - 60,4 kcal / mol

$\Delta S = \Delta S(GG) + \Delta S(GA) + \Delta S(AA) + \Delta S(AT) + \Delta S(TT) + \Delta S(TC) + \Delta S(CC) + \Delta S_{init} + \Delta S_{sc}$

= -28,4 - 23,5 - 21,9 - 15,2 - 21,9 - 23,5 - 28,4 - 9,0 - 1,4 cal / (mol*K)

= - 173,2 cal / (mol*K)

Kennt man nun die Werte für Entropie und Enthalpie, so berechnet sich die Schmelztemperatur wie folgt:

$$T_m = \frac{\Delta H[cal/mol]}{\Delta S[cal/(K \cdot mol)] + R \cdot \ln(c/4)} - 273{,}15^\circ C \quad \text{mit}$$

$$R = 1{,}987 \frac{cal}{K \cdot mol} \quad \text{(Gaskonstante) und}$$

c = Konzentration der DNA-Probe.

Man beachte die unterschiedlichen Einheiten von ΔH in der Tabelle und in der T_m-Formel. Für selbstkomplementäre Sequenzen wird c/4 durch c ersetzt.

Um vernünftige Werte zu erhalten, müssen die Sequenzen mindestens 8 nt lang sein, als gut (ca. 6 % Abweichung [Sugimoto96]) gelten Ergebnisse für 12 bis 70 nt lange Sequenzen.

Korrekturterme

Die tatsächliche Schmelztemperatur (die Dissoziationstemperatur) von DNA hängt nicht nur von den Sequenzen selbst ab, sondern auch von Eigenschaften der Umgebung, wie dem pH-Wert der Lösung und verschiedenen Konzentrationen. Außer der bereits in der *nearest neighbor*-Methode eingegangenen Probenkonzentration werden im Rahmen des Generators berücksichtigt:

- *Salzkonzentration* der Lösung. Die Anwesenheit von Salz (i. a. NaCl) wirkt sich stabilisierend auf die dsDNA aus. Wasser schwächt die Wasserstoffbrücken zwischen komplementären DNA-Sequenzen, da die Wassermoleküle mit den Basen um die Bindungen konkurrieren (daher kann man auch destilliertes Wasser verwenden, um dsDNA zu dissoziieren). In Anwesenheit von Salz bilden die Wassermoleküle aber bevorzugt Hydrathüllen um die Salzionen und konkurrieren somit nicht mehr um die Bindungen der Wasserstoffbrücken [Stryer94]. Mit der Salzkonzentration erhöht sich also die Schmelztemperatur.

Der an die jeweilige verwendete Formel für T_m anzuhängende Term lautet [Genset, Genosys, Oligocalc]:

$+ 16{,}6 * \log (c_{salt})$

(SantaLucia et al. schlagen $+ 12{,}5 * \log (c_{salt})$ vor [SantaLucia96], der oben genannte Term ist aber gebräuchlicher.)

- *Formamidkonzentration*. Formamid dient dazu, die Schmelztemperatur künstlich zu verringern, da es noch stärker als Wasser um die Bindungen der Basen konkurriert. Der an die jeweilige Formel für T_m anzuhängende Term lautet [Genosys]:

$- 0{,}62 * c_F$

Mismatches verringern ebenfalls die Stabilität und damit die Schmelztemperatur von DNA (etwa um 1 ° C pro Prozent *mismatches* [Deaton95]). Da die generierten Sequenzen aber perfekt hybridisieren sollen, werden *mismatches* hier nicht berücksichtigt.

Einfluß auf die Schmelztemperatur hat auch der pH-Wert der Lösung. Die hier aufgeführten Formeln beziehen sich auf pH 7, einen Korrekturterm für abweichende pH-Werte war in der Literatur nicht zu finden.

Ein Korrekturterm für Filterhybridisierung wird hier ebenfalls nicht beachtet, da die Anwendungen i. a. lösungsbasiert sind. Soll die Hybridisierung auf einem Filter stattfinden, ist ein entsprechender Term hinzuzufügen. Vorgeschlagen wird - 7,6 °C [Rychlik89, Genosys].

Wahl für die Implementierung

Aus den verschiedenen möglichen Formeln und Parametern wurde für die Implementierung des Generators folgende Auswahl getroffen:

- Konzentrationen: $c = 2*10^{-7}$ M, $c_{salt} = 0{,}05$ M, $c_F = 0$ M.

Dies sind Standardwerte für eine PCR. Die Werte können vom Benutzer geändert werden.

- Autor: ΔH- und ΔS-Werte aus [Sugimoto96], da diese die genaueren Ergebnisse liefern. Auch dieser Wert ist vom Benutzer änderbar.
- Methode: Wallace für Sequenzen mit Längen < 13 nt, %GC-Formel für Sequenzen mit Längen > 50 nt, *nearest neighbor*-Methode sonst.
- Parameter für %GC-Formel: A = 81,5, B = 500 [Genosys]. Da keine Vergleiche der Eignung der verschiedenen Parameterpaare gefunden wurde, ist diese Entscheidung relativ willkürlich. Genosys erscheint als Anbieter von synthetisierten Oligonucleotiden jedoch als vertrauenswürdige Quelle.

- Korrekturterme: + 16,6 * log (c_{salt}) - 0,62 * c_F

Der Benutzer kann einen Wertebereich für die Schmelztemperatur bzw. den GC-Gehalt der Sequenzen vorgeben. Hat der Niehaus-Algorithmus eine Sequenz (einen Pfad) vollständig erzeugt, so wird deren Schmelztemperatur bzw. GC-Gehalt berechnet. Liegt der ermittelte Wert nicht im vorgegebenen Intervall, so wird auch hier Backtracking ausgelöst, um eine bessere Sequenz zu finden.

4.1.3.2 Homologievergleich

Bisher wurde bei der *uniqueness* nur die Vermeidung gemeinsamer Subsequenzen berücksichtigt (s. o.).

Um zu große Ähnlichkeiten zwischen den Sequenzen auch im Sinne der H-Distanz zu vermeiden, kann der Benutzer einen Maximalwert für die *Homologie* der Sequenzen angeben. Um die Homologie zweier Sequenzen zu berechnen, werden diese gegeneinander verschoben und für jede dieser theoretisch möglichen Anlagerungen die Anzahl der komplementären Basenpaare gezählt. Die Homologie ist der Quotient aus dem Maximum dieser Anzahlen und der kleineren der beiden Sequenzlängen.

Auch hier wird nach Fertigstellung einer kompletten Sequenz die Homologie dieser Sequenz sowie deren Komplement gegenüber den anderen vorliegenden Sequenzen berechnet, und falls einer dieser beiden Werte das vorgegebene Maximum überschreitet, Backtracking ausgelöst.

4.1.3.3 *Fraying*

Da GC-Basenpaare mit drei Wasserstoffbrücken verbunden sind, AT-Basenpaare dagegen nur mit zwei Wasserstoffbrücken, sind GC-Basenpaare die stabileren. Um ein „Ausfransen“ (*fraying*) der Sequenzen, d. h. ein teilweises Aufschmelzen an den Enden, zu verhindern, kann man mit dieser Option dafür sorgen, daß das jeweils erste und das letzte Basenpaar der zu erzeugenden Sequenzen GC-Basenpaare sind.

Dies ist besonders dann wichtig, wenn man in der Implementierung *in vitro* möglichst nah an das Modell der *two state-transition* annähern möchte [Breslauer86, SantaLucia96]. Dieses Modell geht davon aus, daß die dsDNA entweder über die ganze Länge hybridisiert oder komplett denaturiert vorliegt, Zwischenzustände gibt es nicht. Von dieser vereinfachten Annahme wird insbesondere bei der Ermittlung der ΔH- und ΔS-Werte für die Berechnung der Schmelztemperatur mit der *nearest neighbor*-Methode ausgegangen. Schränkt man also die

Schmelztemperatur der vom Compiler zu erzeugenden Sequenzen mit dieser Methode ein, ist ein Anwählen dieser Option sinnvoll.

4.1.3.4 Kein GGG

Folgen drei oder mehr Guanin-Basen direkt aufeinander, so stellen sich *in vitro* ungewöhnliche und unerwünschte Phänomene ein:

- Die mit der *nearest neighbor*-Methode vorhergesagten ΔH-Werte weichen ungewöhnlich stark von den gemessenen ab [Breslauer86, SantaLucia96].
- Es bilden sich GG-Basenpaare [Seeman83, Seeman90, Sundquist89, Sen90].
- Es können sich Vierfachstränge bilden [Sundquist89, Sen90].

Um diese Effekte zu vermeiden, kann man mit der Option „Kein GGG“ dafür sorgen, daß keine Basissequenzen verwendet werden, die drei oder mehr aufeinander folgende Guanin-Basen enthalten.

4.1.4 Analyse von Sequenzen

Der Generator bietet eine Funktion an, mit der man Sequenzen auf ihre *uniqueness* untersuchen kann. Dies kann interessant sein, wenn man Sequenzen, die nicht mit dem Generator erzeugt wurden, untersuchen möchte, aber auch wenn Verletzungen der *uniqueness* bei der Erzeugung toleriert werden mußten (s. 0), oder wenn man wissen möchte, ob eine *uniqueness* auch für kleinere Basissequenzlängen als die bei der Erzeugung verwendete besteht.

Hierzu kann der Benutzer der Analysefunktion einen Bereich von Basissequenzlängen vorgeben. Der Generator extrahiert für jede Länge dieses Bereichs jeweils alle Basissequenzen dieser Länge aus den zu untersuchenden Sequenzen und gibt sie in eine Datei (`AnalyzeUniqueness.txt`) aus. Außerdem wird ausgegeben, wie oft diese Basissequenz oder ihr Komplement aufgetreten ist, an welchen Positionen in welchen Sequenzen, und schließlich ob die Basissequenz selbstkomplementär ist.

4.2 Der Compiler

In diesem Kapitel folgen auf eine kurze Übersicht über Ein- und Ausgaben des Programms Beschreibungen der wichtigsten Verfahren und Werkzeuge.

4.2.1 Eingabe

4.2.1.1 Die reguläre Grammatik

Da die eigentlich zu übersetzenden Elemente die Regeln der Grammatik sind und diese im Fall der regulären Grammatiken nur über eine mögliche Form verfügen (mit Ausnahme der Endregeln), läßt sich aus ihnen eindeutig bestimmen, welche Zeichen Terminale und welche Variablen sind. Es genügt also die Eingabe der Regeln, um die Grammatik zu bestimmen, sowie ggf. noch die Eingabe von Startvariable und Start- und Endterminal, falls diese von den Voreinstellungen (S, s und e) abweichen sollten.

Es ist zwar in der Theorie formaler Sprachen üblich, daß Terminale mit kleinen und Variablen mit großen Buchstaben benannt werden, der Benutzer ist jedoch natürlich nicht darauf festgelegt, er kann insbesondere auch Zeichen verwenden, die keine Buchstaben sind.

Für die Korrektheit einer eingegebenen Regelmenge sind neben der korrekten Form der Regeln, die durch eine entsprechende Benutzerführung in der Eingabemaske unterstützt wird, folgende Punkte zu beachten:

- Es ist mindestens eine Start- und mindestens eine Endregel vorhanden.
- Das Start- bzw. Endterminal wird nur in Start- bzw. Endregeln verwendet.
- In Start- bzw. Endregeln wird nur das Start- bzw. Endterminal verwendet.
- Die Startvariable taucht nur auf der linken Seite von Startregeln auf.

4.2.1.2 Restriktionsschnittstellen

Die spezifischen Sequenzen für die spätere Anwendung von Restriktionsenzymen (z. B. für die spätere Klonierung) stehen nach dem *self-assembly* an Anfang und Ende der Logomere. Eingegeben werden der jeweils obere Strang dieser Sequenzen sowie die Positionen, an denen das entsprechende Enzym diese dsDNA-Sequenz im oberen und unteren Strang schneidet.

Es können natürlich auch, je nach weiterer Verwendung der Logomere, andere Sequenzen als Überhänge angegeben werden.

4.2.1.3 Wiederzuverwendende Sequenzen

Soll eine bestehende und bereits übersetzte Grammatik erweitert werden, so können deren Terminal- und Variablensequenzen im neuen Übersetzungsvorgang wiederverwendet werden, um somit die neuen Algomere zu den alten kompatibel zu halten. So können z. B. die Sequenzen für Nullen und Einsen wiederverwendet werden, wenn eine Bitstring-Grammatik

geändert werden soll, so daß das Auslesen der Bitstrings per PCR (s. 3.3.2) mit alten und neuen Logomeren gemeinsam erfolgen kann, ohne neuen Primer für die Bitsequenzen zu benötigen.

Eingegeben wird jeweils der obere Strang sowie das Zeichen, dem diese Sequenz zugeordnet werden soll.

Der Compiler geht davon aus, daß diese Sequenzen sowie die ggf. bereits bestehenden Übergänge zwischen Terminal- und Variablensequenzen korrekt sind im Sinne der *uniqueness*, was der Fall ist, wenn sie mit diesem Compiler (unter Verwendung der gleichen Basissequenzlänge wie in der neuen Übersetzung) generiert worden sind.

4.2.1.4 Parameter

Hierunter fallen essentielle Parameter wie die Länge der zu generierenden Sequenzen und der Basissequenzen, aber auch Optionen wie z. B. das Verbot der Verwendung von drei oder mehr Guanin-Basen hintereinander.

Für die Definition der Parameter sowie Hinweisen zu ihrer Wahl siehe 4.2.5.

4.2.1.5 Kompatible Sequenzen

Diese Sequenzen haben keine Bedeutung für die Grammatik oder die Algomere, aus ihnen werden nur vor jedem Generatoraufruf die Basissequenzen der jeweiligen Länge extrahiert und markiert, so daß die erzeugten Sequenzen auch diesen Sequenzen gegenüber *unique* sind. Dies kann nötig sein, falls sich bei der Anwendung noch andere Sequenzen außer den Algomeren im Reagenzglas befinden.

4.2.2 Ausgabe

Bei erfolgreicher Übersetzung werden die vier Gruppen von Sequenzen (Variablen, Start-, End- und echte Terminale) in folgende Dateien ausgegeben:

- `Variables.txt`: In dieser Datei sind die Variablen, die entsprechenden Sequenzen sowie einige zusätzliche Angaben wie GC-Gehalt und Schmelztemperatur aufgelistet.
- `Terminals.txt`: In dieser Datei sind die Terminalsequenzen in gleicher Weise wie oben die Variablen aufgelistet, nach Start-, End- und echten Terminalen gruppiert.
- `Algomers_ss.txt`: Hier sind für jede Regel der obere und untere Strang des entsprechenden Algomers aufgelistet, wiederum mit den „technischen Daten“.

Wurden mehrere Start- bzw. Endterminale erzeugt, so tauchen dementsprechend die Start- bzw. Endregeln mehrfach auf. Die Restriktionsschnittstellen am Anfang bzw. Ende der Terminatorsequenzen sind so angefügt, wie sie vom jeweiligen Restriktionsenzym geschnitten werden.

- `Algomers_ds.txt`: Diese Datei enthält ebenfalls die oberen und unteren Stränge der Algomere, allerdings ohne die „technischen Daten“ und so verschoben, wie sie später in der Anwendung hybridisieren sollen.

Konnte der Compiler die Übersetzung nicht erfolgreich durchführen, so wird zurückgegeben:

- Entweder der Fehler in der Regelmenge, der vom Parser entdeckt wurde,
- oder die Gruppe von Sequenzen, für die nicht genügend Sequenzen mit den vorgegebenen Anforderungen generiert werden konnten.

4.2.3 Der Parser

Vor der Erzeugung der Sequenzen werden die Regeln der zu übersetzenden regulären Grammatik auf Korrektheit überprüft und vorverarbeitet.

Da die Regeln einer regulären Grammatik eine sehr feste Form haben, können aus ihnen unmißverständlich die Terminale und Variablen extrahiert werden, so daß der Benutzer sie nicht extra angeben muß.

Desweiteren überprüft der Parser die Regeln auf Korrektheit. Siehe dazu auch Kapitel 4.2.1.

4.2.4 Compilerstrategien

Grob lassen sich die Sequenzen in zwei Gruppen einteilen: in die für Variablen und die für Terminale. In einem Logomer wechseln sich Sequenzen aus diesen beiden Gruppen jeweils ab. Wie in 4.2.4.1 beschrieben, geben die Sequenzen der zuerst generierten Gruppen den Rahmen für die Sequenzen der zweiten Gruppe vor, welche durch das Ausfüllen der Lücken zwischen den Sequenzen der ersten Gruppe erzeugt werden.

Die Terminalsequenzen lassen sich selbst wieder in drei Gruppen aufteilen: „echte“ Terminale (die in Elongatoren vorkommen), Start- und Endterminale. Letztere beiden werden zwar für die Grobeinteilung zu den Terminalen gezählt, müssen aber zur Generierung getrennt betrachtet werden, da sie an einer Seite mit Restriktionsschnittstellen verbunden sind statt mit Variablensequenzen und mehrere Kodierungen für ein Terminal möglich sind.

Bei der Übersetzung von Regeln in Algomere treten Probleme auf, die über die Perspektive des bloßen Generators hinausgehen. Für die Darstellung einer regulären Grammatik genügt es nicht, eine Menge von Sequenzen zur Repräsentation der Terminale und Variablen zu erzeugen, wie in Kapitel 4.1 beschrieben.

- Die Konkatenation der Sequenzen kann zu *uniqueness*-Verletzungen im Übergang von einer Sequenz zur anderen führen.
- Die Sequenz einer Variable geht ggf. nicht nur in eine Terminalsequenz pro Seite über, sondern mehrere, und umgekehrt.
- Unter Umständen kann es erforderlich sein, Verletzungen der *uniqueness* hinzunehmen, um die Regeln einer Grammatik überhaupt auf DNA-Sequenzen abbilden zu können.
- Die Reihenfolge, in der die verschiedenen Gruppen von Sequenzen erzeugt werden, hat Auswirkungen sowohl auf die *uniqueness* als auch auf die Ausbeute.
- Die Aufteilung der Terminalgruppe in drei Teilgruppen, die getrennt erzeugt werden, aber ähnliche Rollen in den Logomeren übernehmen, bringt weitere Probleme mit sich.

Es war für die Entwicklung eines DNA-Sequenz-Compilers also notwendig, Strategien zu entwerfen, mit denen der Compiler diesen Problemen begegnet.

4.2.4.1 Sequenzerzeugung (*uniqueness revisited*)

„Zusammenwachsen" statt Konkatenation

Ein einfaches Verfahren, das sich alleine mit dem Generator umsetzen läßt, ist es, mit Hilfe des Generators Sequenzen für die Variablen und Terminale zu erzeugen, diese dann zu Algomeren zusammenzubauen und auszugeben. Leider ist dieses Verfahren fehlerträchtig, denn an den Verbindungsstellen zweier Sequenzen würden wieder (BL[1] – 1) neue Basissequenzen entstehen.

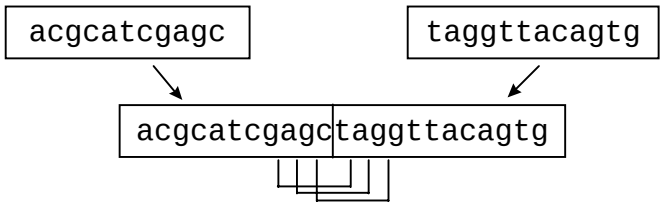


Abbildung 4.4: Bei der Konkatenation von Sequenzen entstehen im Übergangsbereich *BL* - 1 neue Basissequenzen der Länge *BL* (hier für *BL* = 4).

[1] *BL* = Basissequenzlänge

Diese wären aber im Konstruktionsprozeß des Generators nicht berücksichtigt worden und könnten daher auch an anderer Stelle vorkommen und somit die *uniqueness* verletzen. Tatsächlich könnte sich schlimmstenfalls die maximale Länge mehrfach auftretender Subsequenzen auf 2 * *BL* – 2 verdoppeln.

Um dies zu verhindern, betrachtet der Compiler zur Erzeugung nicht nur Terminal- und Variablensequenzen einzeln, sondern jeweils den gesamten oberen Strang eines Algomers <Variable|Terminal|Variable>[1].

Hierzu verwendet der Compiler eine besondere, im Folgenden beschriebene Funktion des Generators, die es erlaubt, Lücken zwischen zwei Teilsequenzen aufzufüllen. Somit kann man die Sequenzen für die Variablen zuerst generieren, dann die Terminalsequenzen durch Auffüllen der Lücken zwischen zwei Variablensequenzen erzeugen. Unter Umständen werden die Sequenzen im umgekehrter Reihenfolge generiert (s. 4.2.4.2), für die Beschreibungen in diesem Abschnitt sei aber diese Reihenfolge gewählt.

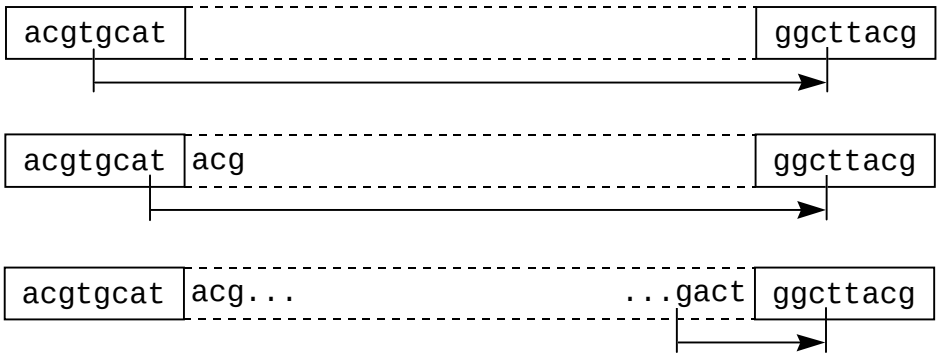


Abbildung 4.5: Auffüllen der Lücke zwischen zwei Sequenzen (hier für Basissequenzlänge 4). Oben: Die Startbasissequenz für den zu suchenden Pfad wird durch die letzte Basissequenz der vorderen Rahmensequenz vorgegeben. Mitte: Der Übergang ist erzeugt, der nächste Nachfolger ist die erste Basissequenz der Extension. Unten: Die Extension ist fertig erzeugt, die drei Basissequenzen des hinteren Übergangs müssen noch nach Verfügbarkeit überprüft werden.

Das Auffüllen funktioniert auch nach dem Prinzip der Pfadsuche im Basissequenz-Graphen. Allerdings ist hier der Startknoten durch die letzte Basissequenz der vorderen Teilsequenz vorgegeben[2]. Die ersten *BL* – 1 Nachfolgeknoten gehören damit noch nicht zur zu erzeugenden Sequenz (geben aber bereits die ersten *BL* – 1 Nucleotide an), sondern bilden den Übergang zwischen den Sequenzen. Nach Fertigstellung der Sequenz (der *Extension*) wird das Pfadwachstum wiederum *BL* – 1 Schritte weitergeführt, um die Basissequenzen im hinteren

[1] Für Terminatoren ist entsprechend eine Variable durch eine Restriktionsschnittstelle zu ersetzen.

[2] Sollte die vordere Sequenz kürzer sein als die Basissequenzlänge, so wird sie zufällig (aber unter Beachtung der *uniqueness*) aufgefüllt.

Übergang zu berücksichtigen. Hierbei sind diese *BL* – 1 Nachfolger bereits durch die ersten Nucleotide der hinteren Sequenz vorgegeben. Dadurch, daß der Pfad auch die Übergangsbereiche einschließt, wird gewährleistet, daß auch diese Basissequenzen nur einmal vorkommen.

Gleiche Sequenz hat verschiedene Übergänge

Hierbei taucht die Schwierigkeit auf, daß ein Terminal in mehreren Regeln vorkommen kann, seine DNA-Kodierung also zwischen verschiedenen Paaren von Variablensequenzen zu liegen kommt und damit verschiedene Übergangsbereiche entstehen. Daher sammelt der Compiler vor der Erzeugung der entsprechenden Terminalsequenz erst alle Paare von Variablensequenzen, die diese Terminalsequenz einrahmen sollen, und übergibt diese dem Generator. Dieser startet nun gleichzeitig von den verschiedenen durch die vorderen Variablensequenzen vorgegebenen Startknoten und läßt die Pfade zusammenwachsen, so daß sie über die Strecke der eigentlichen Terminalsequenz hinweg gleich sind. Anschließend divergieren sie wieder im Übergangsbereich zu den hinteren Variablensequenzen.

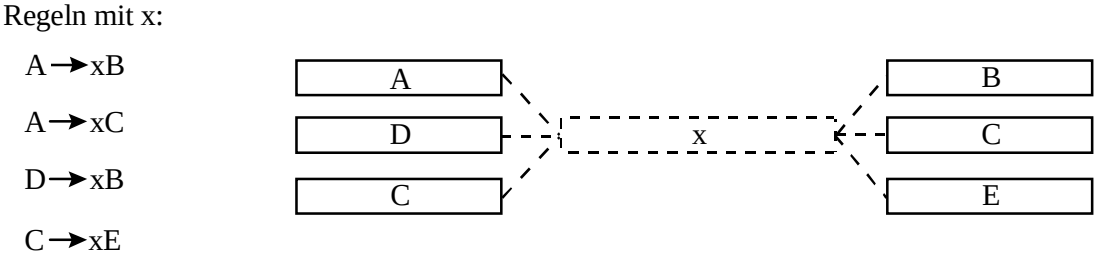


Abbildung 4.6: Durch die vier Regeln mit dem Terminal x ergeben sich vier verschiedene Pfade, die sich über die Länge der Terminalsequenz für x überschneiden. Weiter überschneiden sich auch einige Sequenzen für mehrfach vorkommende Variablen, so daß vorne und hinten je drei Pfade zusammenlaufen.

Notwendige Verletzung der *uniqueness*

Ein weiteres Problem ergibt sich dadurch, daß es durchaus Grammatiken geben kann, in denen z. B. eine Variable mit mehr als vier verschiedenen Terminalen benachbart ist. In diesem Fall ist eine Mehrfachverwendung von Basissequenzen und damit eine Verletzung der *uniqueness* unvermeidlich.

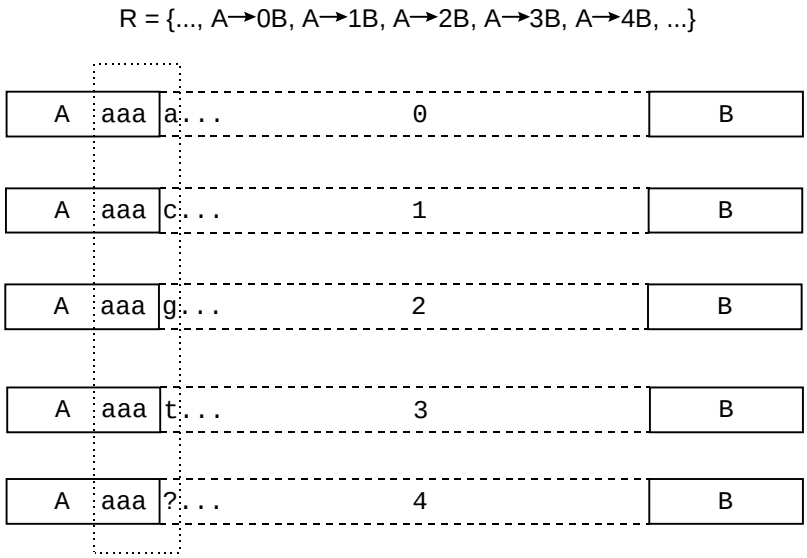


Abbildung 4.7: Haben mehr als vier verschiedene Terminalsequenzen Übergänge zu der gleichen Variablensequenz, so muß mindestens eine Basissequenz mehrfach verwendet werden. Der gepunktete Rahmen zeigt die Spalte an, in der eine Verletzung der *uniqueness* toleriert werden muß, um alle Regeln aus R übersetzen zu können.

Daher kann der Generator derartige Verletzungen innerhalb einer Spalte, d. h. an der gleichen Position für alle Pfade, tolerieren. Hierzu werden die Sequenzen der lückenfüllenden Gruppe parallel erzeugt und die eventuell auftretenden Verletzungen der *uniqueness* zwischen deren Pfaden toleriert. Der Benutzer kann angeben, wie groß der Bereich an Anfang und Ende der zu erzeugenden Sequenz ist, in der derartige Verletzungen toleriert werden sollen. Diese Längenangabe bezieht sich hier allerdings aus der Sicht der Extension auf Basen anstatt auf Basissequenzen, d. h. daß eine Länge des Toleranzbereichs von 1 bedeutet, daß die erste bzw. letzte Basissequenz des Übergangs mehrfach vorkommen darf, nicht die erste bzw. letzte Basissequenz der eigentlichen Sequenz.[1]

Natürlich kann diese notwendige Verletzung auch durch verschiedene einrahmende Sequenzen der bereits generierten Gruppe verursacht werden, nicht nur durch verschiedene Sequenzen der zu erzeugenden Gruppe.

Eine Besonderheit stellt bei der parallelen Pfadsuche das Backtracking dar. Wird für einen Pfad Backtracking ausgelöst (durch fehlende Nachfolger, unerwünschte Schmelztemperatur o. ä.), so wird dieses auch nur für diesen Pfad durchgeführt, die anderen Extensionen werden so lange fixiert, bis die Pfadsuche für die „schuldige“ Sequenz wieder auf gleicher Höhe mit ihnen ist.

[1] s. Errata

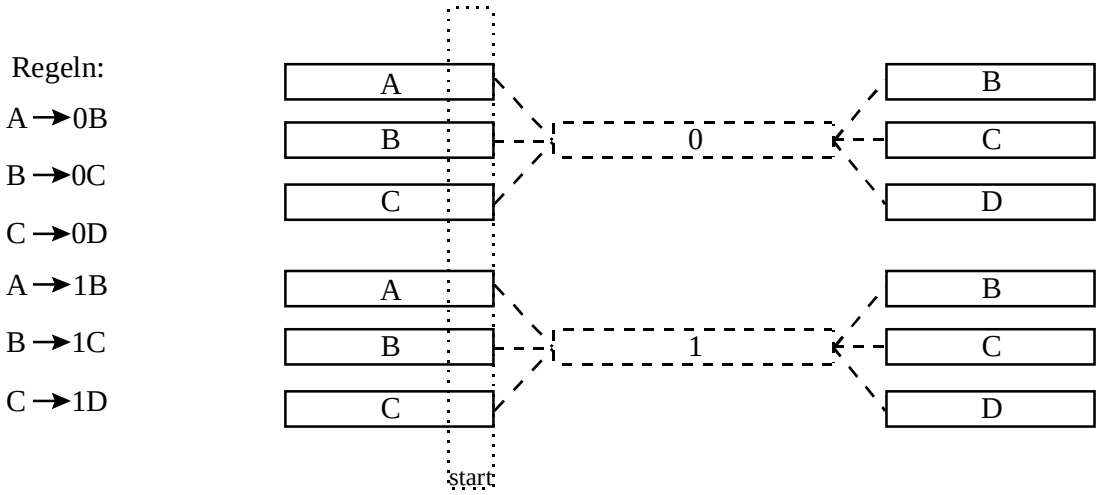


Abbildung 4.8: Paralleles Auffüllen zweier Extensionen für die Terminale 0 und 1. Der gepunktete Rahmen zeigt die Spalte, in der die Startbasissequenzen für die Pfadsuche liegen. Ab der nächsten Spalte können ggf. Verletzungen der *uniqueness* toleriert werden.

Start- und Endterminale

Auch wenn die Sequenzen für Start- und Endterminale vor den Variablensequenzen erzeugt werden (s. u.) und daher bei ihrer Erzeugung keine Übergänge zu diesen berücksichtigt werden müssen, wird für sie die Methode des „Zusammenwachsens“ verwendet, da die Übergänge zu den Restriktionsschnittstellen berücksichtigt werden müssen. Die einrahmenden Bereichen für die Variablensequenzen bleiben hierbei leer, dementsprechend werden dort natürlich auch keine Übergänge generiert.

4.2.4.2 Reihenfolge der Erzeugung

Die Erzeugung der Sequenzen dieser vier Gruppen in vier Schritten erlaubt außerdem die Verwendung von vier verschiedenen Graphen und gestattet somit für jede Gruppe eine eigene Wahl der Basissequenzlänge. Zu beachten ist, daß es zur Zeitersparnis kein gruppenübergreifendes Backtracking gibt, d. h. eine erzeugte Gruppe wird nicht wieder aufgelöst, wenn die Erzeugung der nachfolgenden Gruppe fehlschlägt. Dies mag zu unnötigen Fehlschlägen führen, ist aber zur Beschränkung der Rechenzeit äußerst sinnvoll, insbesondere im Falle einer erfolglosen Übersetzung, bei der das Ende der Suche sonst erst spät erfolgen würde.

Vorzug der kürzeren Sequenzen

Angenommen, man möchte zunächst Sequenzen einer Gruppe 1 erzeugen und wählt dazu die Basissequenzlänge BL_1, anschließend erzeugt man die Sequenzen einer Gruppe 2 mit Basissequenzlänge BL_2.

Wenn man die Sequenzen der Gruppe 1 erzeugt hat, muß man natürlich bei der Erzeugung der Sequenzen der Gruppe 2 die verwendeten Basissequenzen der ersten Gruppe als benutzt berücksichtigen. Dies ist aber problematisch, wenn $BL_1 \neq BL_2$.

Daher werden vor der Erzeugung von Gruppe 2 Basissequenzen der Länge BL_2 aus den Sequenzen der Gruppe 1 extrahiert und diese dann als benutzt markiert. D. h. daß die Sequenzen der Gruppe n *unique* sind

- untereinander bzgl. der Basissequenzlänge BL_n,
- zu Sequenzen einer Gruppe m mit m < n ebenfalls bzgl. BL_n,
- zu Sequenzen einer Gruppe m mit m > n dagegen bzgl. BL_m.

Falls $BL_1 < BL_2$ gilt, so gälte die stärkere *uniqueness* der Sequenzen aus Gruppe 1 also nur innerhalb dieser Gruppe. Dreht man andererseits die Reihenfolge der Erzeugung um (vertauscht also die Sequenzen der Gruppen 1 mit denen der Gruppe 2), so daß $BL_1 > BL_2$ gilt, wird die Ausbeute der erzeugten Sequenzen aus Gruppe 2 geringer, da pro verwendeter längerer Basissequenz mehrere kürzere Basissequenzen als benutzt markiert werden müssen.

Testläufe haben gezeigt, daß die Ausbeute bei halbwegs hohen (aber *in vitro* nötigen) Ansprüchen an die Eigenschaften der Sequenzen so gering sein kann, daß die Übersetzung versagt. Zudem werden die Hybridisierungen von Variablen- und Terminalsequenzen in getrennten Schritten vorgenommen, so daß man die Gefahr von Fehlhybridisierungen zwischen den verschiedenen Gruppen als weniger wichtig ansehen kann als die Gefahr von Fehlhybridisierungen innerhalb einer Gruppe. Daher wurde die Entscheidung getroffen, die kürzeren Sequenzen (soweit möglich, s. u.) zuerst zu generieren.

Damit beim *annealing* im *self-assembly*-Schritt, in dem die Algomere sich per Ligation zu Logomeren verbinden, die Variablensequenzen (die als *sticky ends* dienen) hybridisieren, ohne daß die Terminalsequenzen denaturieren (und so die Algomere zerfallen), muß das *self-assembly* bei einer Temperatur stattfinden, die unter der Schmelztemperatur (genauer: dem Schmelztemperaturbereich) der Terminalsequenzen liegt. Das erfordert, daß die Schmelztemperatur der Variablensequenzen geringer sein muß als die der Terminalsequenzen. Da die Sequenzlänge eine maßgebliche Einflußgröße für die Schmelztemperatur ist, sollte man also die Variablensequenzen kürzer wählen als die Terminalsequenzen.

Aus diesen Gründen werden normalerweise die Variablensequenzen zuerst generiert. Dies ist allerdings nicht immer möglich (bzw. sinnvoll).

Wiederverwendung von Sequenzen

Der DNA-Sequenz-Compiler bietet auch die Möglichkeit, fertige Sequenzen in späteren Übersetzungen wiederzuverwenden, um Sequenzen verschiedener Grammatiken zueinander kompatibel zu halten. Sollen z. B. einige Terminalsequenzen wiederverwendet werden, so ist es sinnvoller, zunächst die restlichen, noch fehlenden Terminalsequenzen zu erzeugen, und anschließend erst die Variablensequenzen durch Auffüllen der Lücken zu generieren. Würde man zuerst die Variablen übersetzen (frei, d. h. ohne die fertigen Terminalsequenzen zu beachten) und dann erst dazu passend die Terminale, so bestünde die Möglichkeit, daß man mit der freien Wahl der Variablensequenzen bereits die *uniqueness* verletzt hat, da die Übergangsbereiche zu den fertigen Terminalsequenzen nicht in der Konstruktion enthalten waren.

Sind also wiederzuverwendende Sequenzen vorgegeben, so werden zunächst die noch fehlenden Sequenzen der entsprechenden Gruppe erzeugt. Sind sowohl Variablen- als auch Terminalsequenzen vorgegeben[1], so werden wieder aus oben genannten Gründen die Variablen bevorzugt.

Echte, Start- und Endterminale

Die drei Teilgruppen der Terminalsequenzen können auch nicht immer konfliktfrei generiert werden, ohne daß später erzeugte Sequenzen die bereits erzeugten anderer Gruppen beachten, denn wenn die Variablensequenzen zu diesem Zeitpunkt bereits erzeugt (oder vorgegeben) wurden, so erzeugt die erste Terminalgruppe bereits Übergänge zu Variablen, die die nachfolgenden Terminalgruppen beachten müssen, insbesondere im Hinblick auf die ggf. nötige Verletzung der *uniqueness* innerhalb einer Spalte. Daher werden die vorliegenden Sequenzen bereits bearbeiteter Terminalgruppen ähnlich wie wiederzuverwendende Sequenzen der eigenen Gruppe behandelt (s. u.).

Innerhalb der Terminalgruppe werden die echten Terminale den Start- und Endterminalen vorgezogen, unabhängig davon, welche Terminalsequenzen vorgegeben wurden. Dadurch wird die Anzahl der Fallunterscheidungen (und der Programmcode zur Beachtung fertiger Terminalsequenzen, s. u.) klein gehalten. Konflikte zwischen den Terminalgruppen können auftreten, wenn Sequenzen für Variablen und Start-/Endterminale vorgegeben sind, so daß die Erzeugung der echten Terminalgruppe die Sequenzen für s und e und deren Übergänge zu den Variablensequenzen nicht berücksichtigt. Dieser Fall dürfte aber eher selten auftreten, da wohl hauptsächlich Sequenzen für Variablen und echte Terminale wiederverwendet werden. Falls er doch

[1] Hierbei können die beschriebenen *uniqueness*- Verletzungen zwischen den verschiedenen Gruppen ggf. unvermeidlich sein, wenn die Sequenzen nicht mit dem Compiler generiert wurden.

einmal auftreten sollte, so kann man sich dadurch behelfen, daß man die beiden Gruppen getrennt in zwei Compilerläufen erzeugt.

In Tabelle 4.3 sind die Bearbeitungsreihenfolgen der Gruppen noch einmal dargestellt, in Abhängigkeit davon, welche Sequenzen vorgegeben wurden.

Vorliegende Sequenzen				**Bearbeitungs-reihenfolge**
V	**T**	**s**	**e**	
-	-	-	-	Vf Tp sp ep
-	mindestens ein x			Tf sf ef Vp
x	*	*	*	Vf Tp sp ep

Tabelle 4.3: Bearbeitungsreihenfolge der Gruppen. V = Variablen, T = echte Terminale, s = Start-terminale, e = Endterminale, f = freie Erzeugung, p = passende Erzeugung (durch Auffüllen), - = keine Sequenzen vorgegeben, x = Sequenzen vorgegeben, * = - oder x.

Späte Erzeugung kurzer Sequenzen

Werden die Variablen zuletzt generiert, so ergibt sich wieder das oben erklärte Problem der zu geringen Ausbeute bei kleiner Basissequenzlänge. Daher wendet der Compiler hier einen Trick an, der die Rechenzeit evtl. etwas erhöht, aber auch die Chancen für eine erfolgreiche Übersetzung verbessert.

Die Variablen werden, passend zu den Terminalsequenzen, mit der Basissequenzlänge der echten Terminalsequenzen erzeugt. Somit sind sie gegenüber den Terminalsequenzen zwar bzgl. einer anderen Basissequenzlänge als ihrer eigenen *unique*, aber die Wahrscheinlichkeit für eine ausreichende Ausbeute an Sequenzen ist wesentlich höher.

Um nun aber die *uniqueness* innerhalb der Variablengruppe bzgl. ihrer eigenen Basissequenz-länge zu gewährleisten, werden die Variablensequenzen einzeln erzeugt, jeweils anschließend auf ihre *uniqueness* zu den anderen bereits generierten Variablensequenzen (bzgl. der richtigen Basissequenzlänge) geprüft und, falls diese *uniqueness* verletzt wird, verworfen und neu erzeugt. Dies kann u. U. einige Zeit in Anspruch nehmen, bis genügend Sequenzen generiert sind oder der Compiler einen Mißerfolg meldet, insbesondere, da das Prüfen und Verwerfen außerhalb des Generators stattfindet und somit kein Backtracking ausgelöst wird, sondern ein neuer Generierungslauf gestartet wird, und somit auch ein und dieselbe Sequenz mehrmals generiert werden kann[1]. Da man aber bei einem (aufgrund der sonst geringen Ausbeute an Sequenzen) fehlgeschlagenen Übersetzungsversuch einen neuen Lauf (oder sogar mehrere

[1] Eine Endlosschleife wird vermieden, indem die Anzahl der Generierungsversuch begrenzt wird.

Läufe) starten muß, ist es fraglich, ob ein Verzicht auf diesen „Trick“ eine wirkliche Zeitersparnis wäre. Außerdem wäre bei den weiteren Läufen eine nötige Erhöhung der Basissequenzlänge und damit eine Einbuße an uniqueness zu erwarten.

4.2.4.3 Wiederverwendung von Sequenzen

Werden Sequenzen für Variablen oder Terminale wiederverwendet, so werden diese automatisch als kompatible Sequenzen beachtet und ihre Basissequenzen zu jedem Generatoraufruf extrahiert und als benutzt markiert. Sollen die restlichen Sequenzen der entsprechenden Gruppe durch Auffüllen von Lücken erzeugt werden, so werden diese „alten“ Sequenzen bei der Erzeugung mitgeführt, um die ggf. nötige spaltenweise Verletzung der *uniqueness* zu erlauben.

Hierzu werden auch für diese Variablen bzw. Terminale die Sequenzen der jeweils anderen Gruppe gesammelt, die die vorgegebene Sequenz im Logomer einrahmen werden. Die parallele Suche der Pfade zum Auffüllen der Lücken wird gestartet, hierbei werden die jeweiligen Nachfolgeknoten für die vorgegebenen Pfade aber nicht zufällig gewählt, sondern aus den wiederzuverwendenden Sequenzen gelesen.

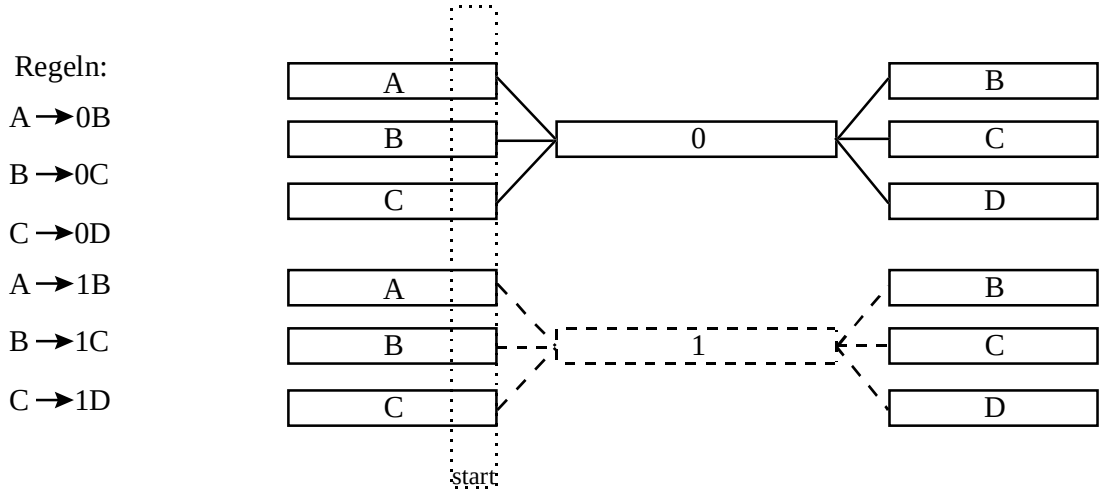


Abbildung 4.9: Die Terminalsequenz für 0 wird wiederverwendet, die für 1 muß noch erzeugt werden. Auch hier wachsen die Pfade parallel, um ggf. notwendigen Verletzungen der *uniqueness* in einer Spalte zulassen zu können. Die Pfade, die über die 0 verlaufen, sind dabei durch die wiederzuverwendende Sequenz vorgegeben und werden nachverfolgt.

Eine besondere Art der Wiederverwendung ergibt sich durch die Terminalsequenzen, die immer in der Reihenfolge echte Terminale – Startterminale – Endterminale erzeugt werden. Werden die Sequenzen aller drei Gruppen passend zu bereits fertigen Variablensequenzen erzeugt, so muß man bei der Erzeugung der Start- und Endterminale auch die Übergänge der echten Terminale zu den Variablen beachten.

Dies wird dadurch erreicht, daß bei der Erzeugung der Start- bzw. Endterminale zunächst die Variablen gesammelt werden, die die hintere Rahmensequenzen für Start- bzw. die vordere für Endterminale bilden. Entsprechend werden die Sequenzen der echten Terminale gesammelt, für die es Regeln gibt, in denen sie diese Variablen als hintere bzw. vordere Rahmensequenz haben. Diese Terminalsequenzen werden dann wie wiederzuverwendende Sequenzen an den Generator übergeben, nur mit dem Unterschied, daß sie auf der Seite, auf der s bzw. e in die Restriktionsschnittstelle übergehen, keine einrahmenden Sequenzen haben.

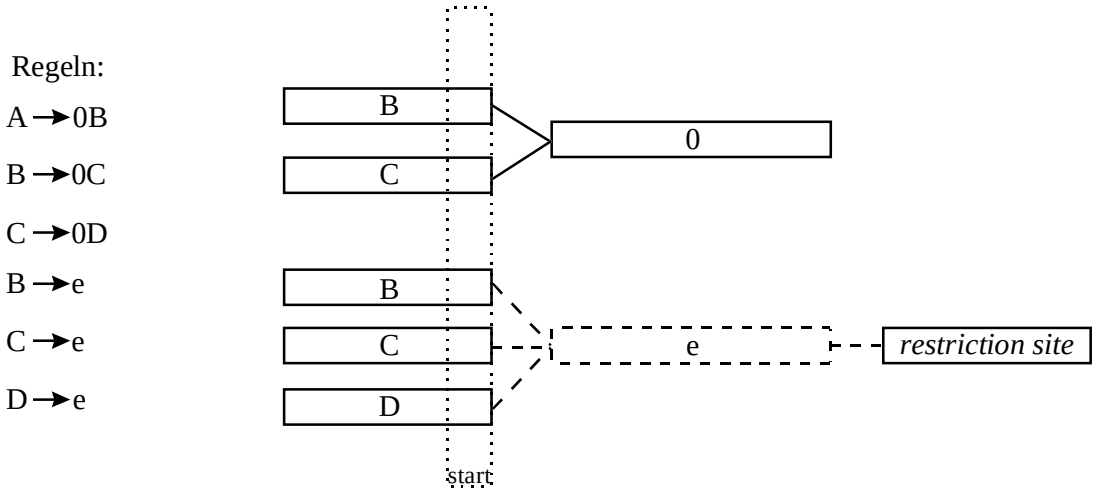


Abbildung 4.10: Bei der Erzeugung der Endterminalsequenz(en) müssen die Übergänge von Variablen zu echten Terminalen berücksichtigt werden. Dazu werden die echten Terminale wie wiederzuverwendende Sequenzen behandelt. Die erste Regel wird hier nicht beachtet, da kein Übergang von A nach e generiert werden muß, hier also auch keine Verletzung der *uniqueness* toleriert wird.

Ggf. verschiedene Längen von echten und anderen Terminalen stellen insofern kein Problem dar, als daß nur die Übergänge an einem Ende gruppenübergreifend verglichen werden müssen und somit eine parallele Pfadsuche für Endterminale problemlos über die Länge der fixen echten Terminalsequenzen hinausgehen oder auch bereits vor dieser enden kann. Da der Generator die Sequenzen von links nach rechts (von 5' nach 3') bearbeitet und für Startterminale die hinteren (rechten) Übergänge zu betrachten sind, wäre hier die Situation für die Behandlung verschieden langer Extensionen komplizierter. Die Verfolgung der vorgegebenen Pfade für die echten Terminale würde entweder vor den Startbasissequenzen der Startterminale beginnen oder, was problematischer wäre, würde erst später einsetzen. Daher werden nicht die Startterminalsequenzen selber, sondern ihre Komplementäre erzeugt und dementsprechend auch die anderen Sequenzen in komplementärer Form vorgegeben. Dadurch liegen die Variablensequenzen (bzw. deren Komplementäre) nun auch hier vor den Extensionen, so daß die gruppenübergreifend zu beachtenden Übergänge wiederum die vorderen sind. Die *uniqueness* wird durch den Wechsel zu Komplementärsequenzen nicht

beeinträchtigt, da die verwendeten Basissequenzen und ihre Komplementäre als zu einer Äquivalenzklasse gehörend betrachtet werden. GC-Gehalt und Schmelztemperatur werden natürlich ebenfalls nicht beeinflußt.

4.2.5 Parameter des Compilers

Zur Bedeutung einiger der Parameter siehe auch 4.1.2 und 4.1.3.

4.2.5.1 Definition der Parameter

Die zu erzeugenden Sequenzen lassen sich in vier Gruppen einteilen: Variablen-, echte Terminal-, Startterminal- und Endterminalsequenzen. Für jede dieser vier Gruppen gibt es folgende Parameter:

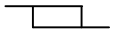
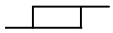
Name	Beschreibung	Typ	Werte-bereich	Einheit	Default-wert
SequenceLength	Länge der zu erzeugenden Sequenzen	integer	> 0	nt	-
BaseLength	Länge der zu verwendenden Basissequenzen	integer	[1, 16]	nt	-
GCLower, GCUpper	Untere und obere Grenze für den GC-Gehalt der Sequenzen	floating point	[0, 1]	-	0.0, 1.0
TmLower, TmUpper	Untere und obere Grenze für die Schmelztemperatur der Sequenzen	floating point	IR	° C	0.0, 100.0
NoFraying	GC-Basenpaare an den Enden, um Ausfransen zu verhindern	bool	{true, false}	-	false
NoGGG	nicht mehr als zwei Guaninbasen hintereinander	bool	{true, false}	-	false
BSGCLower, BSGCUpper	Untere und obere Grenze für den GC-Gehalt der zu verwendenden Basissequenzen	floating point	[0, 1]	-	0.0, 1.0

Um gleiche Parameter verschiedener Gruppen zu unterscheiden, werden ggf. die Buchstaben V, T, S und E angehängt. Also heißt z. B. die Länge der Variablensequenzen SequenceLengthV.

Die beiden Gruppen der Start- und Endterminalsequenzen haben außerdem den Parameter

Name	Beschreibung	Typ	Werte-bereich	Einheit	Default-wert
SequenceNumber	Anzahl der verschiedenen Start- bzw. Endterminal-sequenzen	integer	> 0	-	1

Weiter gibt es folgende gruppenunabhängige Parameter:

Name	Beschreibung	Typ	Werte-bereich	Einheit	Default-wert
RandomSeed	Startwert des Pseudo-zufallszahlengenerators	long integer	IN	-	12345
MaxHomology	Maximal zulässige Homologie zwischen zwei erzeugten Sequenzen	floating point	[0, 1]	-	1.0
ViolationZoneFront, ViolationZoneEnd	Länge der Bereiche an Anfang und Ende der Sequenzen, in denen die ggf. notwendige Verletzung der Uniqueness toleriert werden soll (s. 4.2.4)	integer	[0, Sequence-Length]	nt	0, 0
AlgomerForm	Form der Algomere: 0 =  1 = 	integer	{0, 1}	-	0
SampleConcentration	Probenkonzentration der DNA (für die T_m-Berechnung)	floating point	> 0	M	$2*10^{7}$
SaltConcentration	Salzkonzentration der Lösung (für die T_m-Berechnung)	floating point	> 0	M	0.05
FormamideConcentration	Formamidkonzentration der Lösung (für die T_m-Berechnung)	floating point	≥ 0	M	0.0
Author	Aus wessen Arbeit stammen die für die T_m-Berechnung verwendeten ΔH- und ΔS-Werte?	-	{Breslauer, Sugimoto}	-	Sugimoto

4.2.5.2 Zur Wahl der Parameter

Da die Parameter einigen Einfluß auf die Erfolgswahrscheinlichkeit sowohl des Compilers als auch der weiteren Verwendung der Algomere haben, eine sinnvolle Wahl ihrer Werte aber nicht auf den ersten Blick ersichtlich ist, folgen an dieser Stelle einige Hinweise zu den Auswirkungen der Parameter und ihrer Werte.

Zu beachten ist, daß die Überlegungen zu der Ausbeute an erzeugten Sequenzen nur als Schätzungen zu verstehen sind, da nicht nur das Erzeugungsverfahren zufallsgesteuert ist, sondern durch die zu übersetzende Grammatik weitere Einschränkungen der Ausbeute erfolgen können. Insbesondere gelten diese Abschätzungen nur für einen Generatoraufruf und damit für eine Sequenzgruppe. Die Beachtung der *uniqueness* zwischen den Gruppen verringert die Ausbeute natürlich weiter.

Siehe auch Kapitel 5 für beispielhafte Parameterwerte.

Sequenzlänge, Basissequenzlänge

Die wichtigsten Parameter sind die Längen der zu erzeugenden Sequenzen und der dazu zu verwendenden Basissequenzen. Sie haben Einfluß auf das Hybridisierungsverhalten der Algomere, auf die *uniqueness*, aber auch auf die maximal mögliche Anzahl von Sequenzen.

Die Sequenzlängen sollten so gewählt werden, daß sich die Schmelztemperaturen in „vernünftigen“ Bereichen befinden, d. h. sie sollten mit vertretbarem technischem Aufwand im Labor zu erreichen sein. Bei langen Sequenzen (und somit langen Logomeren) besteht die Gefahr, daß sie brechen oder Sekundärstrukturen bilden, was sie ggf. für die weitere Verwendung unbrauchbar macht.

Die Längen von Terminal- und Variablensequenzen sollten sich deutlich unterscheiden, insbesondere sollten die Variablensequenzen kürzer sein. Dadurch erzielt man für die *sticky ends* der Algomere eine (möglichst deutlich) geringere Hybridisierungstemperatur als für die doppelsträngigen Terminalteile, so daß die Algomere während des *annealing* im *self-assembly*-Schritt stabil bleiben. Somit kann i. a. auch die Basissequenzlänge für die Variablen kürzer gewählt werden.

Die Basissequenzlängen sollten natürlich möglichst kurz gewählt werden, um eine möglichst stringente *uniqueness* zu erzielen. Andererseits sinkt mit geringerer Länge auch die Anzahl der verfügbaren Basissequenzen und damit die Ausbeute an Sequenzen.

Die maximale Anzahl möglicher Sequenzen in Abhängigkeit von Sequenzlänge *SL* und Basissequenzlänge *BL* läßt sich wie folgt bestimmen:

Anzahl Basissequenzen: $N_{bs}(BL) = 4^{BL}$

Falls *BL* gerade ist, so werden die selbstkomplementären Basissequenzen nicht verwendet. Es gibt keine selbstkomplementären Basissequenzen ungerader Länge, da das mittlere Nucleotid nicht zu sich selbst komplementär sein kann. Außerdem halbiert sich die Anzahl der erlaubten Basissequenzen dadurch, daß mit jeder verwendeten Basissequenz auch deren Komplement nicht mehr zur Verfügung steht. Also gilt für die Anzahl brauchbarer Basissequenzen:

Falls *BL* gerade: $N_{useful}(BL) = \frac{N_{bs}(BL) - 4^{(BL/2)}}{2}$,

sonst $N_{useful}(BL) = \frac{N_{bs}(BL)}{2}$

Eine Sequenz der Länge *SL* besteht aus *SL* - *BL* + 1 Basissequenzen. Damit ist die maximale Anzahl erzeugbarer Sequenzen

$$N_{seqs}(SL) = \left\lfloor \frac{N_{useful}(BL)}{SL - BL + 1} \right\rfloor$$

Wie in 4.1.2 erläutert, wird diese maximale Anzahl durch das Niehaus-Verfahren nicht erreicht, da unbenutzte Knoten im Graphen übrig bleiben, die keinen vollständigen Pfad mehr bilden können. Es bleibt zu untersuchen, ob die Topologie des Graphen es überhaupt erlaubt, diesen „Verschnitt“ zu vermeiden, da die hier aufgeführte Abschätzung nicht die Einschränkung der Freiheitsgrade der Basissequenzen berücksichtigt, die durch die gegenseitige Überlappung bei der Bildung von Sequenzen entsteht.

In Tabelle 4.4 sind einige Beispielwerte für maximal mögliche und tatsächlich erreichte Sequenzausbeuten in Abhängigkeit von *SL* und *BL* angegeben.

Um das Niehaus-Verfahren zu beschleunigen, werden die Basissequenzen intern durch 32-Bit-Wörter dargestellt. Mit einer Kodierung von zwei Bit pro Base sind so Basissequenzlängen von maximal 16 nt möglich. Die Anzahl der dadurch zur Verfügung stehenden Basissequenzen dürfte für die in den nächsten Jahren realistischen Anwendungen ausreichend sein, außerdem würden längere gemeinsame Subsequenzen bereits zu massiven Fehlhybridisierungen führen.

	SequenceLength										
BaseLength	**10**	**11**	**12**	**13**	**14**	**15**	**16**	**17**	**18**	**19**	**20**
4	13,2 von 17 (77,6%)	11,9 von 15 (79,3%)	10,4 von 13 (80,0%)	9,3 von 12 (77,5%)	8,4 von 10 (84,0%)	8,1 von 10 (81,0%)	7,3 von 9 (81,1%)	6,8 von 8 (85,0%)	6,1 von 8 (76,3%)	5,8 von 7 (82,9%)	5,6 von 7 (80,0%)
5	70,6 von 85 (83,1%)	59,8 von 73 (81,9%)	53,1 von 64 (83,0%)	47,1 von 56 (84,1%)	42,5 von 51 (83,3%)	38,6 von 46 (83,9%)	35,6 von 42 (84,8%)	33,0 von 39 (84,6%)	30,7 von 36 (85,3%)	28,4 von 34 (83,5%)	26,6 von 32 (83,1%)
6	327,8 von 403 (81,3%)	274,5 von 336 (81,7%)	233,5 von 288 (81,1%)	206,3 von 252 (81,9%)	184,0 von 224 (82,1%)	165,4 von 201 (82,3%)	150,5 von 183 (82,2%)	138,5 von 168 (82,4%)	127,3 von 155 (82,1%)	118,4 von 144 (82,2%)	111,4 von 134 (83,1%)
7	0 von 2048 (0,0%)	0 von 1638 (0,0%)	0 von 1365 (0,0%)	0 von 1170 (0,0%)	0 von 1024 (0,0%)	757,8 von 910 (83,3%)	682,1 von 819 (83,3%)	623,4 von 744 (83,8%)	573,3 von 682 (84,1%)	531,1 von 630 (84,3%)	494,8 von 585 (84,6%)

	SequenceLength									
BaseLength	**21**	**22**	**23**	**24**	**25**	**26**	**27**	**28**	**29**	**30**
4	5,1 von 6 (85,0%)	5,0 von 6 (83,3%)	4,9 von 6 (81,7%)	4,2 von 5 (84,0%)	4,0 von 5 (80,0%)	4,0 von 5 (80,0%)	4,0 von 5 (80,0%)	3,7 von 4 (92,5%)	3,4 von 4 (85,0%)	3,3 von 4 (82,5%)
5	25,1 von 30 (83,7%)	24,1 von 28 (86,1%)	22,3 von 26 (85,8%)	21,7 von 25 (86,8%)	20,8 von 24 (86,7%)	19,8 von 23 (86,1%)	19,0 von 22 (86,4%)	18,2 von 21 (86,7%)	17,4 von 20 (87,0%)	16,9 von 19 (88,9%)
6	103,6 von 126 (82,2%)	97,7 von 118 (82,8%)	92,9 von 112 (82,9%)	88,6 von 106 (83,6%)	84,5 von 100 (84,5%)	80,1 von 96 (83,4%)	76,6 von 91 (84,2%)	73,7 von 87 (84,7%)	70,5 von 84 (83,9%)	67,8 von 80 (84,8%)
7	463,0 von 546 (84,8%)	433,5 von 512 (84,7%)	409,6 von 481 (85,2%)	387,6 von 455 (85,2%)	367,4 von 431 (85,2%)	350,8 von 409 (85,8%)	334,1 von 390 (85,7%)	320,3 von 372 (86,1%)	307,1 von 356 (86,3%)	294,0 von 341 (86,2%)

	SequenceLength									
BaseLength	**31**	**32**	**33**	**34**	**35**	**36**	**37**	**38**	**39**	**40**
4	3,0 von 4 (75,0%)	3,0 von 4 (75,0%)	3,0 von 4 (75,0%)	3,0 von 3 (100,0%)	2,8 von 3 (93,3%)	3,0 von 3 (100,0%)	2,9 von 3 (96,7%)	2,9 von 3 (96,7%)	2,4 von 3 (80,0%)	2,0 von 3 (66,7%)
5	16,4 von 18 (91,1%)	15,7 von 18 (87,2%)	15,0 von 17 (88,2%)	14,9 von 17 (87,6%)	14,2 von 16 (88,8%)	13,9 von 16 (86,9%)	13,3 von 15 (88,7%)	12,8 von 15 (85,3%)	12,5 von 14 (89,3%)	12,2 von 14 (87,1%)
6	65,4 von 77 (84,9%)	62,8 von 74 (84,9%)	60,6 von 72 (84,2%)	58,8 von 69 (85,2%)	56,6 von 67 (84,5%)	54,7 von 65 (84,2%)	53,0 von 63 (84,1%)	51,7 von 61 (84,8%)	50,1 von 59 (84,9%)	48,2 von 57 (84,6%)
7	283,4 von 327 (86,7%)	272,5 von 315 (86,5%)	262,8 von 303 (86,7%)	253,4 von 292 (86,8%)	245,1 von 282 (86,9%)	237,7 von 273 (87,1%)	230,0 von 264 (87,1%)	223,0 von 256 (87,1%)	217,1 von 248 (87,5%)	211,1 von 240 (88,0%)

Tabelle 4.4: Ausbeute der Generierung in Abhängigkeit von *SequenceLength* und *BaseLength*. Angegeben sind die tatsächlich generierten und die maximal möglichen Sequenzanzahlen sowie in Klammern die prozentuale Rate der generierten an den möglichen. Die tatsächlichen Werte sind über zehn Generatorläufe mit verschiedenen Startwerten des Zufallszahlengenerators gemittelt.

NoGGG

Die Option *NoGGG* verringert recht deutlich die Ausbeute an Sequenzen, da alle Basissequenzen, die drei oder mehr aufeinanderfolgende Guaninnucleotide besitzen, ausgeschlossen werden. Die Anzahl der hierdurch nicht mehr verfügbaren Basissequenzen berechnet sich wie folgt:

Die Anzahl der Basissequenzen der Länge BL, die *k* aufeinanderfolgende Gs enthalten, ist

$$N_{GGG}(BL,k) = 4^{BL-k} \cdot (BL - k + 1),$$

da jede der *BL* - *k* übrigen Nucleotide eine beliebige der vier Basen besitzen kann und die G-Subsequenz sich an *BL* - *k* + 1 Positionen in der Basissequenz befinden kann.

Damit ist die Anzahl aller ausgeschlossenen Basissequenzen

$$N_{GGG}(BL) = \sum_{k=3}^{BL} N_{GGG}(BL,k) = \sum_{k=3}^{BL} 4^{BL-k} \cdot (BL - k + 1)$$

Tabelle 4.5 enthält hierfür einige Beispielwerte.

	BaseLength								
	4	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**
N_{bs}	256	1024	4096	16384	65536	262144	1048576	4194304	16777216
N_{useful}	120	512	2016	8192	32640	131072	523776	2097152	8386560
N_{GGG}	9	57	313	1593	7737	36409	167481	757305	3378745
Ratio	7,5%	11,1%	15,5%	19,4%	23,7%	27,8%	32,0%	36,1%	40,3%

Tabelle 4.5: Durch *NoGGG* verursachter Verlust an Basissequenzen in Abhängigkeit von ihrer Länge. Angegeben sind jeweils gesamte und brauchbare Anzahlen von Basissequenzen, die durch *NoGGG* ausgeschlossenen Basissequenzen sowie deren Anteil an den brauchbaren.

Dieser teilweise erhebliche Verlust an Basissequenzen sollte nur in Ausnahmefällen gerechtfertigt sein. Die Gefahr der Bildung von Vierfachsträngen läßt sich durch hohe Salzkonzentration (oder *potassium*) verringern [Sen90]. Die Länge der Subsequenzen, an denen es durch GG-Basenpaarbildung zu Fehlhybridisierungen kommen kann, liegt im durch den hier gewählten *uniqueness*-Begriff tolerierten Bereich[1]. Wesentlich bleibt der zu treffende Kompromiß zwischen der Verringerung der Ausbeute und dem Fehler bei der Berechnung der Schmelztemperatur, wenn diese mit der *nearest-neighbor*-Methode approximiert wird. Leider konnte ich in der Literatur keine Quantifizierung dieses Fehlers finden.

Eine zusätzliche Erschwernis für den Compiler kann die Wahl von *NoGGG* in Verbindung mit *NoFraying* sein (s. u.).

GC-Gehalt der Basissequenzen

Ein weiterer Verlust an verwendbaren Basissequenzen kann durch die Einschränkung ihres GC-Gehalts entstehen.

Die Anzahl der Sequenzen der Länge *BL* mit dem absoluten GC-Gehalt *GC* ($0 \le GC \le BL$) ist

$$N_{seqs}(BL, GC) = 2^{GC} \cdot 2^{BL-GC} \cdot \binom{BL}{GC} = 2^{BL} \cdot \binom{BL}{GC},$$

da man für jedes GC-Nucleotid sowie für jedes Nicht-GC-Nucleotid nur noch zwei Basen zur Auswahl hat. Der Binomialkoeffizient gibt die Anzahl der möglichen Anordnungen der GC-Nucleotide in der Basissequenz an.

Mit $Ratio_{GC} = \frac{GC}{BL}$ folgt

$$N_{seqs}(BL, Ratio_{GC}) = 2^{BL} \cdot \binom{BL}{BL \cdot Ratio_{GC}}$$

[1] Die Ausnahme ist die Basissequenz, die nur aus Guaninnucleotiden besteht und durch die GG-Basenpaarbildung als selbstkomplementär betrachtet werden kann.

und damit für die Anzahl der Sequenzen mit einem GC-Gehalt zwischen $LRatio_{GC}$ und $URatio_{GC}$

$$N_{seqs}(BL, LRatio_{GC}, URatio_{GC}) = \sum_{R=LRatio_{GC}}^{URatio_{GC}} N_{seqs}(BL, R)$$

Beispielwerte hierfür sind in Tabelle 4.6 zu finden.

	BaseLength								
GC	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**
0	16 (6,250%)	32 (3,125%)	64 (1,563%)	128 (0,781%)	256 (0,391%)	512 (0,195%)	1024 (0,098%)	2048 (0,049%)	4096 (0,024%)
1	64 (25,000%)	160 (15,625%)	384 (9,375%)	896 (5,469%)	2048 (3,125%)	4608 (1,758%)	10240 (0,977%)	22528 (0,537%)	49152 (0,293%)
2	96 (37,500%)	320 (31,250%)	960 (23,438%)	2688 (16,406%)	7168 (10,938%)	18432 (7,031%)	46080 (4,395%)	112640 (2,686%)	270336 (1,611%)
3	64 (25,000%)	320 (31,250%)	1280 (31,250%)	4480 (27,344%)	14336 (21,875%)	43008 (16,406%)	122880 (11,719%)	337920 (8,057%)	901120 (5,371%)
4	16 (6,250%)	160 (15,625%)	960 (23,438%)	4480 (27,344%)	17920 (27,344%)	64512 (24,609%)	215040 (20,508%)	675840 (16,113%)	2027520 (12,085%)
5		32 (3,125%)	384 (9,375%)	2688 (16,406%)	14336 (21,875%)	64512 (24,609%)	258048 (24,609%)	946176 (22,559%)	3244032 (19,336%)
6			64 (1,563%)	896 (5,469%)	7168 (10,938%)	43008 (16,406%)	215040 (20,508%)	946176 (22,559%)	3784704 (22,559%)
7				128 (0,781%)	2048 (3,125%)	18432 (7,031%)	122880 (11,719%)	675840 (16,113%)	3244032 (19,336%)
8					256 (0,391%)	4608 (1,758%)	46080 (4,395%)	337920 (8,057%)	2027520 (12,085%)
9						512 (0,195%)	10240 (0,977%)	112640 (2,686%)	901120 (5,371%)
10							1024 (0,098%)	22528 (0,537%)	270336 (1,611%)
11								2048 (0,049%)	49152 (0,293%)
12									4096 (0,024%)

Tabelle 4.6: Anzahl sowie prozentualer Anteil der Basissequenzen mit einem bestimmten absoluten GC-Gehalt GC an allen Basissequenzen dieser Länge BaseLength.

GC-Gehalt und Schmelztemperatur der Sequenzen

Für die Anzahl kompletter Sequenzen mit GC-Gehalt in einem bestimmten Bereich gelten die selben Gleichungen wie oben, nur ist die Basissequenzlänge *BL* durch die Sequenzlänge *SL* zu ersetzen. Allerdings berücksichtigt diese Gleichung nicht die *uniqueness* der Sequenzen.

Selbstverständlich sollte der Benutzer auf eine konsistente Wahl der Werte achten. So können z. B. Sequenzen ungerader Länge ebensowenig einen GC-Gehalt von genau 50 % haben wie Sequenzen, die aus Basissequenzen mit einem GC-Gehalt von 10 % bestehen.

Die Wahl des Schmelztemperaturbereichs sollte natürlich in einem realistischen Rahmen liegen, insbesondere die Sequenzlänge ist hier bereits ein maßgeblicher Faktor. Siehe hierzu auch Kapitel 4.2.6.

Liegt die Sequenzlänge in dem Bereich, in dem die *nearest neighbor*-Methode zur Berechnung der Schmelztemperatur verwendet wird, so bietet sich die gleichzeitige Wahl der Option *NoFraying* an, da die verwendeten Werte für Entropie und Enthalpie auf dem *2-state-transition*-Modell beruhen, d. h. es wird vereinfachend angenommen, daß die DNA entweder vollständig einzel- oder vollständig doppelsträngig vorliegt, Zwischenzustände sind ausgeschlossen. In der Realität kann man diese Modellvorstellung annähern, indem man die Enden der dsDNA durch ein GC-Basenpaar stabilisiert.

NoFraying

Die Wahl dieser Option sollte die Ausbeute nur unwesentlich einschränken, da nur die Enden der Sequenzen auf GC-Basenpaare beschränkt werden. Allerdings sollte man beachten, daß, wenn man sowohl Variablen- als auch Terminalsequenzen mit dieser Option generiert, in den Algomeren dann diese GC-Basenpaare (und evtl. sogar zwei gleiche Basen) nebeneinander zu liegen kommen. Sollte außerdem die Option *NoGGG* gewählt worden sein, so kann der Compiler hier ggf. scheitern, wenn er für viele Übergänge Basissequenzen mit genau zwei aufeinanderfolgenden Guaninnucleotiden benötigt.

Sinnvoll ist die Verwendung dieser Option nicht nur, um die Doppelstränge zu stabilisieren, was vor allem für die Terminalsequenzen interessant ist, sondern auch dadurch, daß die mit Hilfe der *nearest neighbor*-Methode approximierten Schmelztemperaturen für diese „nicht-ausfransenden“ Sequenzen geringere Abweichungen von den realen Werten zeigen (s. o.).

Maximale Homologie

Erzeugt der Generator nur eine Menge von Sequenzen, ohne die Verwendung in Algomeren zu beachten, so läßt sich die maximale Homologie auf 0,5 senken, ohne nennenswerte Einbußen bei der Ausbeute zu erhalten. Allerdings steigt die benötigte Rechenzeit sehr stark an, zum einen aufgrund der vielen teuren Homologie-Berechnungen, zum anderen weil es mit jeder neu hinzugekommenen zu vergleichenden Sequenz für die nächste zu erzeugende schwieriger wird, noch die Homologie-Beschränkung gegenüber allen bisher erzeugten Sequenzen zu erfüllen, wodurch das Backtracking immer länger suchen muß, bis eine weitere Sequenz akzeptiert wird.

Im Rahmen der Verwendung im Compiler kommen noch andere Beschränkungen hinzu, so daß die Einschränkung der Homologie auf Werte unter 0,7 in Verbindung mit der Einschränkung

des GC-Gehalts oder der Schmelztemperatur leicht zu Laufzeiten von mehreren Stunden führen kann. Anstatt diesen Parameter allzu restriktiv zu wählen, sollte der Benutzer eher darauf achten, daß die Sequenzlänge deutlich größer ist als die Basissequenzlänge, wodurch eine hohes Maß an *uniqueness* gewährleistet wird.

Größe der *violation zone*

Die Bereiche, in denen Verletzungen der *uniqueness* toleriert werden, weil sie ggf. unvermeidlich sind, sollten natürlich möglichst klein sein. Andererseits ist eben eine gewisse Mindestgröße oft notwendig.

Angenommen, man hat n mal dieselbe Basissequenz als Ausgangspunkt für die Sequenzerzeugung durch „Zusammenwachsen“. Dann hat man im besten Fall (jeder der vier möglichen Nachfolgerknoten wird gleich oft gewählt) im nächsten Schritt vier Gruppen mit (durchschnittlich) jeweils $n/4$ gleichen Basissequenzen. Führt man diese Betrachtung weiter, so muß man die Mehrfachverwendung gleicher Basissequenzen für mindestens $\lceil \log_4(n) \rceil$ Schritte tolerieren.

Tatsächlich wird dieser Wert eher noch höher liegen, da eine völlige Gleichverteilung der Nachfolger nicht sicher ist (auch wenn sie gleichverteilt gezogen werden), insbesondere dann nicht, wenn bestimmte Nachfolger nicht zur Verfügung stehen, weil sie bereits woanders verwendet oder von vornherein ausgeschlossen wurden.

Ein Höchstwert für diese Parameter ist prinzipiell nur durch die Sequenzlänge gegeben, allerdings erscheint es sinnvoll, sie jeweils auf maximal *BaseLength* - 1 zu beschränken (wenn möglich). Somit ist die Verwendung der Basissequenzen, die mehrfach vorkommen dürfen, nur auf die Übergangsbereiche beschränkt, die nächste Basissequenz wäre bereits vollständig in der zu erzeugenden Sequenz enthalten und würde damit die Wahrscheinlichkeit für Fehlhybridisierungen bei der Bildung der Algomere oder bei der Ligation zu Logomeren erhöhen.

4.2.6 Vorgeschlagene Vorgehensweise

Um zunächst eine Kombination von Sequenz- und Basissequenzlängen zu finden, die eine ausreichende Ausbeute liefert, sollte man, ggf. nach Abschätzungen wie oben beschrieben, zunächst einen Übersetzungslauf mit möglichst wenig Einschränkungen durchführen. In einem zweiten Schritt sollte man dann Einschränkungen wie GC-Gehalt, *NoFraying* und *NoGGG* vornehmen.

Ist dies erfolgreich, sollte man in den beiden Ausgabedateien `Variables.txt` und `Terminals.txt` nachsehen, in welchem Bereich sich die Schmelztemperaturen für die Sequenzen befinden. Diese sollten als Anhaltspunkte für die anschließende Einschränkung der Schmelztemperaturbereiche betrachtet werden.

Wenn der Benutzer entsprechend viel Zeit hat, sollte er die Schmelztemperaturintervalle zunächst noch großzügig wählen und dann über einige Compilerläufe immer weiter verkleinern.

Sollte die Übersetzung scheitern, so kann man diesen Schritt zunächst mit anderen Werten für *RandomSeed* wiederholen, bevor man die Ansprüche an die Sequenzen wieder verringert, indem man die Intervalle für GC-Gehalt oder Schmelztemperatur vergrößert oder die Basissequenzlänge erhöht.

Wenn man einen der anderen Parameter (z. B. *NoFraying*) nachträglich stringenter wählt, so ist es u. U. sinnvoll, die Schmelztemperaturbeschränkung zunächst wieder zu lockern.

4.2.7 Analyse der Algomere

Um festzustellen, welchen Umfang die ggf. tolerierten Verletzungen der *uniqueness* tatsächlich haben, kann man die Algomere nach der Übersetzung analysieren. Diese Funktion kann der Benutzer natürlich auch verwenden, um Algomere, die nicht mit dem Compiler erzeugt wurden, zu untersuchen.

Hierzu werden die oberen Stränge der Algomere mit der Analysefunktion des Generators (s. 4.1.4) untersucht und die Ergebnisse in die Datei `AnalyzeUniqueness.txt` geschrieben. Der zur Analyse verwendete Bereich von Basissequenzlängen ist frei wählbar, die der echten Terminale sollte aber sinnvollerweise enthalten sein.

5 Beispiele

In diesem Kapitel werden einige Beispielgrammatiken und ihre Übersetzungen in Algomere vorgestellt. Es werden dabei nur diejenigen Parameter angegeben, die von den Default-Werten abweichen.

Eine Parameterbelegung wird als ungeeignet betrachtet, wenn sie für keine der zehn Belegungen für RandomSeed aus {12345, 54321, 98765, 4, 589129348, 346, 9182736, 667, 23, 361401} zu einer erfolgreichen Übersetzung führte. Übersetzungsversuche, die (auf einem Pentium 166 mit 32 MB Hauptspeicher) mehr als zehn Minuten dauerten, wurden aus Zeitgründen abgebrochen und für gescheitert erklärt. Längere Wartezeiten könnten hier allerdings durchaus zu Erfolgen führen.

In den Ausgabedateien `Terminals.txt` und `Variables.txt` wurde folgendes Ausgabeformat verwendet: Zeichen - Sequenz - Länge - GC-Gehalt - Schmelztemperatur.

5.1 Zufallszahlen

Die Grammatik zur Erzeugung von Zufallszahlen hat die Regelmenge

$R = \{S \rightarrow sA, A \rightarrow 0A, A \rightarrow 1A, A \rightarrow e\}$

Gruppenspezifische Parameter:

Parameter	Variablen	echte Terminale	Startterminal	Endterminal
SequenceLength	10	20	20	20
BaseLength	4	6	6	6
GCLower	0.5	0.5	0.5	0.5
GCUpper	0.5	0.5	0.5	0.5
TmLower	0	50	50	50
TmUpper	100	51	51	51
NoGGG	true	true	true	true

Allgemeine Parameter:

Parameter	Wert
RandomSeed	12345

verwendete Restriktionsschnittstellen:

vor s: HindIII (AAGCTT, wird im oberen und unteren Strang zwischen den Adeninen geschnitten)

nach e: BamHI (GGATCC, wird jeweils zwischen den Guaninen geschnitten)

Ausgaben:

```
Variables.txt
A    aaatcgtcgg 10    0.500000   8.402902

Terminals.txt
Start terminator sequences
s    acggcctatactagctctac  20    0.500000   50.535416
Elongator sequences
0    tacagagtccggtttggtga  20    0.500000   50.764376
1    cagatcgagtgtatgaggag  20    0.500000   50.100650
End terminator sequences
e    ataacctcgttggaccacct  20    0.500000   50.375568

Algomers_ds.txt
S -> sA
agcttacggcctatactagctctac
    atgccggatatgatcgagatgtttagcagcc

A -> 0A
aaatcgtcggtacagagtccggtttggtga
          atgtctcaggccaaaccacttttagcagcc

A -> 1A
aaatcgtcggcagatcgagtgtatgaggag
          gtctagctcacatactcctctttagcagcc

A -> e
aaatcgtcggataacctcgttggaccacctg
          tattggagcaacctggtggacctag
```

Bemerkungen:

Da für diese Grammatik nur wenige Sequenzen nötig sind, konnten die Parameter recht restriktiv gewählt werden. Die Basissequenzlänge für Terminalsequenzen kann bei Verzicht auf die Einschränkungen (bzw. bei Lockerung der Einschränkungen) für GC-Gehalt und Schmelztemperatur sogar auf 5 verringert werden.

Die Schmelztemperatur für die Variablensequenzen wurde nicht explizit eingeschränkt, da ihre Länge in dem Bereich liegt, in dem die geschätzte Schmelztemperatur mit der Wallace-Methode ermittelt wird und damit nur vom GC-Gehalt abhängt.

Die Wahl der Option NoFraying ließ die Übersetzung selbst ohne NoGGG und bei Erweiterung des Schmelztemperaturbereichs für Terminalsequenzen auf 50 - 52 °C scheitern. Erst ein

Vergrößern der ViolationZone auf 2 erlaubte hier wieder erfolgreiche Übersetzungen (z. B. mit RandomSeed = 98765). In der Ausgabe sind hier die dadurch tolerierten *uniqueness*-Verletzungen hervorgehoben.

```
Algomers_ds.txt
S -> sA
agcttctctacgtgttagggactgg
    agagatgcacaatccctgaccgtatccttac

A -> 0A
catag gaatgc ttgctaactaaccggca tc
          g aacgattgattggccgt aggtatc cttac

A -> 1A
catag gaatgc agagtttacgaggatct tc
          g tctcaaatgctcctaga aggtatc cttac

A -> e
cataggaatggcgcatttcttgccgtcgagg
          cgcgtaaagaacggcagctccctag
```

Verzichtet man auf die Einschränkung der Schmelztemperatur und begnügt sich mit der Festlegung des GC-Gehalts auf 50 % für alle Sequenzgruppen, so läßt sich der GC-Gehalt der Basissequenzen auf 30 - 70 % einschränken (z. B. RandomSeed = 12345, ohne NoFraying und NoGGG). Eine Einschränkung auf 50 % führte nicht zum Erfolg.

5.2 Bytes

Die Grammatik zur Erzeugung von Bytes hat die Regelmenge

$R = \{S \rightarrow sA,$
$A \rightarrow 0B,\ A \rightarrow 1B,$
$B \rightarrow 0C,\ B \rightarrow 1C,$
$C \rightarrow 0D,\ C \rightarrow 1D,$
$D \rightarrow 0E,\ D \rightarrow 1E,$
$E \rightarrow 0F,\ E \rightarrow 1F,$
$F \rightarrow 0G,\ F \rightarrow 1G,$
$G \rightarrow 0H,\ G \rightarrow 1H,$
$H \rightarrow 0I,\ H \rightarrow 1I,$
$I \rightarrow e\}$

Gruppenspezifische Parameter:

Parameter	Variablen	echte Terminale	Startterminal	Endterminal
SequenceLength	10	20	20	20
BaseLength	4	6	6	6
GCLower	0.5	0.5	0.5	0.5
GCUpper	0.5	0.5	0.5	0.5

TmLower	0	50	50	50
TmUpper	100	51	51	51
NoFraying	true	true	true	true

Allgemeine Parameter:

Parameter	Wert
RandomSeed	54321
ViolationZoneFront	5
ViolationZoneEnd	5

verwendete Restriktionsschnittstellen:

vor s: HindIII (`AAGCTT`, wird im oberen und unteren Strang zwischen den Adeninen geschnitten)

nach e: BamHI (`GGATCC`, wird jeweils zwischen den Guaninen geschnitten)

Ausgaben:

```
Variables.txt
A     cggaaacatc 10    0.500000   8.402902
B     cttttagccc 10    0.500000   8.402902
C     ggagattacc 10    0.500000   8.402902
D     ccgcaaatag 10    0.500000   8.402902
E     cagagcatac 10    0.500000   8.402902
F     cgtagaactg 10    0.500000   8.402902
G     gacggttatc 10    0.500000   8.402902
H     ctgaagtgac 10    0.500000   8.402902
I     gtcttgtgtc 10    0.500000   8.402902


Terminals.txt
Start terminator sequences
s     gtaattgagagcctgtcagc  20    0.500000   50.406172
Elongator sequences
0     ggatttggcaacaacctgag  20    0.500000   50.033737
1     caaccaggattaagccatgc  20    0.500000   50.593991
End terminator sequences
e     cgaaggttaagttttcgggg  20    0.500000   50.579711


Algomers_ds.txt
S -> sA
agcttgtaattgagagcctgtcagc
    acattaactctcggacagtcggcctttgtag

A -> 0B
cggaaacatcggatttggcaacaacctgag
          cctaaaccgttgttggactcgaaaatcggg
```

```
A -> 1B
cggaaacatccaaccaggattaagccatgc
          gttggtcctaattcggtacggaaaatcggg

B -> 0C
cttttagcccggatttggcaacaacctgag
          cctaaaccgttgttggactccctctaatgg

B -> 1C
cttttagccccaaccaggattaagccatgc
          gttggtcctaattcggtacgcctctaatgg

C -> 0D
ggagattaccggatttggcaacaacctgag
          cctaaaccgttgttggactcggcgtttatc

C -> 1D
ggagattacccaaccaggattaagccatgc
          gttggtcctaattcggtacgggcgtttatc

D -> 0E
ccgcaaatagggatttggcaacaacctgag
          cctaaaccgttgttggactcgtctcgtatg

D -> 1E
ccgcaaatagcaaccaggattaagccatgc
          gttggtcctaattcggtacggtctcgtatg

E -> 0F
cagagcatacggatttggcaacaacctgag
          cctaaaccgttgttggactcgcatcttgac

E -> 1F
cagagcataccaaccaggattaagccatgc
          gttggtcctaattcggtacggcatcttgac

F -> 0G
cgtagaactgggatttggcaacaacctgag
          cctaaaccgttgttggactcctgccaatag

F -> 1G
cgtagaactgcaaccaggattaagccatgc
          gttggtcctaattcggtacgctgccaatag

G -> 0H
gacggttatcggatttggcaacaacctgag
          cctaaaccgttgttggactcgacttcactg

G -> 1H
gacggttatccaaccaggattaagccatgc
          gttggtcctaattcggtacggacttcactg

H -> 0I
ctgaagtgacggatttggcaacaacctgag
          cctaaaccgttgttggactccagaacacag

H -> 1I
ctgaagtgaccaaccaggattaagccatgc
          gttggtcctaattcggtacgcagaacacag
```

```
I -> e
gtcttgtgtccgaaggttaagttttcggggg
          gcttccaattcaaaagccccctag
```

Bemerkungen:

Die Schmelztemperatur für die Variablensequenzen wurde wie bei den Zufallszahlen auch hier nicht eingeschränkt, da sie nur vom GC-Gehalt abhängt.

Eine Verringerung der ViolationZone-Parameter führte zu Mißerfolgen[1].

Die Option NoGGG (u. a. zur Vermeidung der langen G-Sequenz im Algomer zur Regel I → e) führte nur dann zu erfolgreichen Übersetzungen, wenn man keine Einschränkung der Schmelztemperaturen vornahm (aber mit 50 % GC-Gehalt für alle Sequenzgruppen). Der relativ hohe Bedarf an Basissequenzen ließ auch keine Einschränkung des GC-Gehalts der Basissequenzen zu.

Die Verringerung von MaxHomologie auf Werte unter 0,8 führte zu zu langen Suchzeiten, die Ausführung wurde abgebrochen.

5.3 4-Byte-Wörter

Die Regeln sind hier dieselben wie die für die Byte-Grammatik. Die Parameter wurden ebenfalls gleich gewählt, nur daß hier die Anzahl zu erzeugender Startterminale auf 4 erhöht wurde.

Ausgabe:

```
Variables.txt
A    cggaaacatc 10   0.500000   8.402902
B    cttttagccc 10   0.500000   8.402902
C    ggagattacc 10   0.500000   8.402902
D    ccgcaaatag 10   0.500000   8.402902
E    cagagcatac 10   0.500000   8.402902
F    cgtagaactg 10   0.500000   8.402902
G    gacggttatc 10   0.500000   8.402902
H    ctgaagtgac 10   0.500000   8.402902
I    gtcttgtgtc 10   0.500000   8.402902

Terminals.txt
Start terminator sequences
s    caacacatggagttacacgc  20   0.500000   50.208030
s    gaaaaaattggactcggggc  20   0.500000   50.182982
s    gctcctagaagtctacaagc  20   0.500000   50.116172
s    cttctgccatacaactaggc  20   0.500000   50.337490
```

[1] s. Errata

```
Elongator sequences
0    ggatttggcaacaacctgag  20    0.500000   50.033737
1    caaccaggattaagccatgc  20    0.500000   50.593991
End terminator sequences
e    cttgtttaatacaggggcgc  20    0.500000   50.869243
```

Bemerkungen:

Es gelten auch hier die Bemerkungen zur Byte-Grammatik. Wesentlicher Unterschied zu dieser ist eine etwas längere Laufzeit bis zur erfolgreichen Übersetzung.

Der Parameter MaxHomology konnte erfolgreich auf 0,8 gesenkt werden, weitere Verringerungen führten zum Abbruch wegen zu langer Laufzeit.

5.4 Hamilton-Pfad-Problem (Adleman)

Der von Adleman 1994 gelösten Instanz des HPP lag folgender Graph zugrunde:

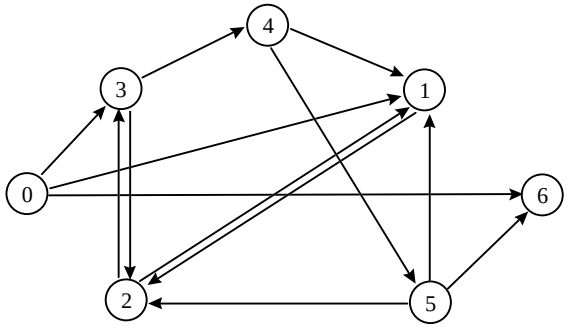


Abbildung 5.1: Graph aus [Adleman94], der einzige Hamilton-Pfad von 0 nach 6 folgt der Numerierung der Knoten.

Er hat 7 Knoten, 14 Kanten und genau einen Hamilton-Pfad von Knoten 0 nach Knoten 6:
$0 \rightarrow 1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 5 \rightarrow 6$

Es folgt zunächst eine kurze Analyse von Adlemans Sequenzen, dann werden einige Varianten vorgestellt, das Problem mit einer regulären Grammatik zu beschreiben. Die dort nicht angegebenen Parameter haben dabei die jeweiligen Default-Werte, als Restriktionsschnittstellen wurden wieder HindIII und BamHI verwendet (s. o.).

Adlemans Sequenzen

Adleman wählte für die Darstellung der Knoten zufällige Sequenzen der Länge 20 nt, deren Hälften als *sticky ends* für die zu dem jeweiligen Knoten führenden bzw. die von ihm wegführenden Kanten dienten (s. 2.2.1). Diese *sticky ends* waren:

Von 0 `GGCTGACGATACATGCATAC`

Nach 1 `ACGAGAGCGG`

Von 1 CCTAGTGAAT

Nach 2 AATATCAAAA

Von 2 TGAGGGGTTA

Nach 3 CTAAAGGACA

Von3 CACGATAAGA

Nach 4 GATGTTGTAC

Von 4 AAGACTAGAG

Nach 5 GTCAAGCAGC

Von 5 ACACGGCTGG

Nach 6 GGACTTCGAGCTAGTTAGAA

Die Sequenzen für „Von 0“ und „Nach 6“ wurden mit doppelter Länge gewählt, da keine Kante zu Knoten 0 führt und keine Kante von Knoten 6 ausgeht und die einen Pfad darstellende dsDNA-Sequenz hier dementsprechend „glatte Enden“, d. h. Enden ohne weitere *sticky ends*, hat.

Bzgl. der *uniqueness* hatte Adleman mit seiner zufälligen Auswahl eine glückliche Hand. Die *sticky ends* sind bei einer Basissequenzlänge von 7 *unique*, bei einer Basissequenzlänge von 6 kommt auch nur eine Basissequenz (ACGATA) doppelt vor, eine weitere (ATGCAT) ist selbstkomplementär.

Bei der Konkatenation der *sticky ends* zu Knotensequenzen kommen bei Basissequenzlänge 6 nur eine weitere doppelt auftretende und eine weitere selbstkomplementäre Basissequenz hinzu. Ab Basissequenzlänge 7 sind auch diese Sequenzen *unique*.

Durch die Bildung der Kantensequenzen ergibt sich nicht nur wiederum eine zusätzliche selbstkomplementäre, sondern auch zwei weitere doppelt vorkommende Basissequenzen der Länge 6. Diese beiden *uniqueness*-Verletzungen entstehen durch die häufige Verwendung der *sticky ends* „Nach 1“ bzw. „Von 5“. Da „Nach 1“ viermal verwendet wird und „Von 5“ dreimal, erscheint eine *violation zone* der Länge 1 tolerabel.

Auch die Homologie der *sticky ends* untereinander und gegenüber ihren Komplementärsequenzen ist recht gut. Der Höchstwert liegt bei 0,6, die meisten Paare haben sogar Werte unter 0,5.

Die Schmelztemperaturen sind allerdings eher schlecht gewählt. Der GC-Gehalt der *sticky ends* liegt zwischen 10 und 70 %, die mit der *nearest neighbor*-Methode approximierten Schmelztemperaturen liegen zwischen 43 und 70 °C für die Sequenzen der Länge 10. „Von 0“ und „Nach 6“ haben sogar Schmelztemperaturen von ca. 82 °C.

Weiterhin sind die Sequenzen nicht gegen *fraying* geschützt. An einer Stelle treten vier Guaninbasen hintereinander auf.

Knoten als Teminale

Wählt man die Knoten des Graphen als Terminale der Grammatik und die Kanten als Variablen, so erhält man eine sehr große Regelmenge, da man für jedes Paar von ankommender und ausgehender Kante eine eigene Regel benötigt. Dies sind beim Beispielgraphen 32 Regeln (incl. Start- und Endregel).

Mit den Parametern

Parameter	Variablen	echte Terminale	Startterminal	Endterminal
SequenceLength	10	20	20	20
BaseLength	5	6	6	6

und keinen weitere Einschränkungen waren bereits ViolationZone-Längen von 5 nötig, um eine erfolgreiche Übersetzung zu erreichen. Dieser Ansatz wurde zugunsten geschickterer Kodierungen mit kleineren Regelmengen verworfen.

Alle Kanten als ein Terminal

Stellt man die Kanten durch Terminale dar, so erhält man deutlich weniger Regeln. Da eine Identifikation der Kanten nicht mehr notwendig ist, sondern es nur auf die Reihenfolge der Variablen (der Knoten) ankommt, welche die „aktiven“ Elemente im *self-assembly* sind, genügt es, ein Terminal für alle Kanten zu verwenden.

Da diese Terminalsequenz allerdings der Mittelteil für 14 Algomere ist, gibt es viele notwendige *uniqueness*-Verletzungen. Daher mußten auch hier die ViolationZone-Parameter auf 5 gesetzt werden, damit die Übersetzung erfolgreich sein konnte.

Die Basissequenzlänge für Variablen ließ sich auf 4 senken. Die Basissequenzlänge für Terminale hätte man ebenfalls senken können, was aber angesichts der großen *violation zones* nicht sinnvoll erscheint.

Ein Terminal pro Kante

Bekommt jede Kante (und damit jede Regel) ein eigenes Terminal zugewiesen, so ist eine Übersetzung mit den Parametern

Parameter	Variablen	echte Terminale	Startterminal	Endterminal
SequenceLength	10	20	20	20
BaseLength	4	6	6	6

auch bei striktem Verbot von Verletzungen (ViolationZoneFront = ViolationZoneEnd = 0) erfolgreich.

Eine Einschränkung des GC-Gehalts der Sequenzen auf 50 % hatte keinen Erfolg (die Läufe wurden wegen zu langer Laufzeit abgebrochen). Dies gilt auch, wenn diese Einschränkung nur für die echten Terminale gemacht wird. Erst, wenn die Längen der *violation zones* auf 1 erhöht wird, gelingt die Übersetzung (dann auch in deutlich unter einer Minute). Dabei ist z. B. für RandomSeed = 12345 die einzige wirklich erfolgte Verletzung die hier hervorgehobene dreifache Verwendung einer Basissequenz:

```
5 -> l1
ttgtg atgccg ctcagtgctccatcagata
          cgagtcacgaggtagtctataatcatcccg

5 -> m2
ttgtg atgccg atctagtcggagagtcgaa
          ctagatcagcctctcagctttgtaaggggt

5 -> n6
ttgtg atgccg tcatggacggtgacatgat
          cagtacctgccactgtactatcggtgacat
```

Der Mißerfolg bei Einschränkung des GC-Gehalts ist bemerkenswert, da der Generator ohne die Verwendung der Sequenzen in einer Grammatik mit den gewählten Längen problemlos ca. 100 Sequenzen mit einem GC-Gehalt von 50 % erzeugen kann. Daß er hier bei 14 Sequenzen bereits scheitert, bzw. sehr lange für die Erzeugung braucht, verdeutlicht, daß die durch die einrahmenden Variablensequenzen und die Übergänge zu diesen entstehende Randbedingung stark beschränkend wirkt. Dies zeigt einerseits, wie wichtig es ist, die Bildung von korrekten Übergängen in der Konstruktion der Sequenzen zu integrieren, andererseits aber auch die Notwendigkeit, ggf. Zugeständnisse zu machen.

Besteht man darauf, daß ViolationeZoneFront = ViolationZoneEnd = 0 gilt, so läßt sich der GC-Gehalt der Variablen auf 50 %, der der Terminale auf 45 - 55 % beschränken. Eine Einschränkung der Schmelztemperaturen der Terminale auf einen Bereich von 2 °C (z. B. 51 - 53 °C) erfordert allerdings wieder eine Erweiterung der *violation zones* auf 1, eine Wahl der Schmelztemperaturen von 51,5 - 52,5 °C sogar eine Lockerung der GC-Gehalt-Einschränkung.

Ein Terminal pro linker Variable

Ein Kompromiß zwischen den vorigen beiden Ansätzen ist die Wahl von einem Terminal für jeden Knoten, von dem eine Kante ausgeht. Dadurch benötigt man einerseits weniger Sequenzen als in der zweiten Variante (6 statt 14), erzwingt andererseits nicht soviele *uniqueness*-Verletzungen wie in der ersten Variante.

Tatsächlich war auch hier eine Wahl der ViolationZone-Längen von 5 notwendig, allerdings wurden weniger Verletzungen begangen als in der ersten Variante, der gleich große Toleranzrahmen wurde also weniger ausgeschöpft[1].

Alle Sequenzen ließen sich auf einen GC-Gehalt von 50 % und die Terminalsequenzen auf eine Schmelztemperatur von 50 - 52 °C einschränken. Eine engere Einschränkung der Schmelztemperaturen war nicht erfolgreich, ebenso die Wahl der Option NoFraying für die Terminale ohne Vergrößerung des Schmelztemperaturintervalls.

Vergleich: zufällige versus gezielt konstruierte Sequenzen

Wie gezeigt wurde, kann mit einer geschickt gewählten Grammatik (Kanten als Terminale, ein Terminal für jede Kante) nicht nur eine ebenso gute *uniqueness* erzeugt werden wie mit der Adleman'schen Kodierung, es ist auch sicher zu erwarten, daß bei größeren Instanzen die Sequenzen des Compilers den zufällig gewählten überlegen sein werden. Desweiteren konnten GC-Gehalt und Schmelztemperatur durch den Compiler wesentlich schärfer begrenzt werden.

Allerdings zeigen die Ergebnisse der anderen Grammatik-Varianten, daß die Wahl der Regeln sehr entscheidend für die Qualität der Algomere ist. Hier z. B. hat erst die - von der reinen Darstellung eines Graphen aus betrachtet redundante - Vergrößerung der Terminalmenge eine Erzeugung von sehr guten Sequenzen ermöglicht. Bei größeren Graphen müßte hier allerdings auch ein Kompromiß bzgl. der Größe der Terminalmenge gefunden werden.

[1] s. Errata

6 Diskussion und Ausblick

6.1 Diskussion

Es wurde gezeigt, daß der entwickelte DNA-Sequenz-Compiler in verschiedenen Anwendungen des DNA-Computing zum Einsatz kommt. Dabei wird jedoch nicht das klassische Konzept eines Compilers verfolgt, der wie z. B. bei Amos et al. einen Programmiersprachen-Quelltext in einen DNA-„Maschinencode“ (der einen NAND-Gatter-Schaltkreis repräsentiert) übersetzen soll, der anschließend mit den in 2.1.2 und 2.2 vorgestellten Operationen schrittweise von einem Prozessor (z. B. einem Labortechniker) verarbeitet wird [Amos98]. Die Programmierung liegt viel mehr in der Erzeugung der Algomere mit bestimmten *sticky ends*, während der damit programmierte Ablauf in der Ligation des *self-assembly* stattfindet, also in einem Schritt, der die hohe Parallelität der *durch* DNA ausgeführten Berechnung ausnutzt, ohne dem DNA-Computing das Rechenkonzept Silizium-basierter Rechner aufzuzwingen.

Allerdings läßt sich das *self assembly* incl. Programmierung und Übersetzung auch als Teilprogramm oder Unterroutine einer größeren DNA-Computing-Anwendung einsetzen. Insbesondere wurden verschiedene Verwendungsmöglichkeiten der Logomere außerhalb der klassischen DNA-Computing-Modelle aufgezeigt, aber auch ihr Einsatz bei der Lösung von Suchproblemen wie dem HPP.

Da das HPP NP-vollständig ist, also jedes NP-Problem auf das HPP reduziert werden kann, ist es also (zumindest theoretisch) möglich, das programmierte *self-assembly* Verfahren zur Lösung beliebiger NP-Probleme zu verwenden.

Doch auch, wenn eine solche Reduktion des Problems i. a. keine Reduktion des zur Lösung (bzw. Approximation der Lösung) nötigen Aufwands bedeutet, kann doch sicherlich das programmierte *self-assembly* in vielen Fällen ein sinnvolles Initialisierungswerkzeug sein, sowohl weil sich bei geschickter Wahl der Grammatik der Suchraum einschränken läßt, als auch weil die mit dem DNA-Sequenz-Compiler erzeugten Sequenzen mit dem Ziel der Minimierung von Fehlern in DNA-Computing-Operationen konstruiert werden.

Ein Problem ist die Abdeckung des Lösungsraums, d. h. die Beantwortung der Frage, wieviel DNA-Sequenzen man benötigt, um mit einer gewissen Wahrscheinlichkeit alle möglichen Ausdrücke in der Initialmenge enthalten zu haben. Werden Grammatiken übersetzt, deren Ausdrücke beliebig lang sein können, so kann selbstverständlich nicht die gesamte Sprache der Grammatik in Logomeren erzeugt werden. Selbst endliche, aber lange Logomere können unwahrscheinlich werden, dies kann man aber zu einem gewissen Grad durch ein geschickt

gewähltes Mengenverhältnis von Terminatoren zu Elongatoren sowie durch die spätere Hinzugabe der Terminatoren zu den z. T. bereits ligierten Elongatoren ausgleichen. Auch bei Grammatiken, die eine endliche Sprache mit Ausdrücken relativ kurzer aber verschiedener Länge erzeugen, läßt sich die Menge an zu verwendender DNA nicht so einfach abschätzen wie im Falle von gleich langen Sequenzen, wo sich die Abschätzung auf das *coupon-collector*-Problem reduzieren läßt [Maley98].

In vielen Anwendungen genügt jedoch die Beschränkung auf Grammatiken mit Ausdrücken fester Länge, diese sind theoretisch wie praktisch leichter zu handhaben. Für andere Fälle müssen Untersuchungen zeigen, wie weit man mit Änderungen und Verfeinerungen des *self-assembly*-Verfahrens eine Abdeckung des zu betrachtenden Lösungsraums gewährleisten kann.

6.2 Ausblick

Der Bereich des DNA-Computing steckt noch in den Kinderschuhen. Es wird noch reichlich Forschung, vor allem auch die praktische Umsetzung von Modellen im Labor, erfordern, um dieses neue Konzept der Informationsverarbeitung wirklich nutzbar zu machen. Aber auch in dem Teilbereich des DNA-Computing, in dem das programmierte *self-assembly* und damit der DNA-Sequenz-Compiler zur Anwendung kommt, gibt es noch viele Ansatzpunkte für weitere Entwicklungen.

Eine interessante Alternative zu dem in dieser Arbeit vorgestellten Verfahren zur Erzeugung von DNA-Sequenzen, an die verschiedene Anforderungen gestellt werden, bieten die Evolutionären Algorithmen. In diesen der natürlichen Evolution nachempfundenen Verfahren stehen Lösungskandidaten für ein Problem miteinander in Konkurrenz. Während schlechtere Lösungen aussterben, können die besseren überleben und sich vermehren, wobei die Nachkommen leichte Änderungen gegenüber den Eltern aufweisen, um so den Lösungsraum weiter zu durchsuchen.

Denkbar ist die Verwendung eines Evolutionären Algorithmus im Generator, wobei die Güte (die sog. *Fitneß*) der Lösungskandidaten (die hier Mengen von Sequenzen wären) sich aus der *uniqueness* und der Einhaltung von Schmelztemperaturen errechnen würde. Auch die unter den Compilerstrategien vorgestellten Anforderungen an die Sequenzen, die über die Generierung einer Menge von Sequenzen hinausgehen, ließen sich ggf. in diese Fitneßberechnung mit einbeziehen. Es ließe sich so evtl. die Ausbeute des Generators, vor allem aber auch die Laufzeit verbessern. Im Rahmen dieser Arbeit wurden bereits Versuche hierzu unternommen, die jedoch aus Zeitgründen zugunsten des Niehaus-Verfahrens wieder verworfen wurden.

Auch der hier verwendete Sequenzgenerator läßt sich noch erweitern. Denkbar ist z. B. eine Art „Feinregulierung“ der Schmelztemperatur von bestimmten Sequenzen, die aus Gründen der *uniqueness* wünschenswert sind, aber nicht die gewünschte Schmelztemperatur besitzen. Eine Möglichkeit ist hierzu die Permutation der vier Basen, die die *uniqueness* der Sequenzen nicht beeinträchtigt [Seeman83, Seeman90], sehr wohl aber ihre Schmelztemperatur. Eine andere Möglichkeit, die allerdings die anschließende Synthetisierung der Oligonucleotide erschwert und möglicherweise nicht effizient zu berechnen ist, ist die gezielte Ersetzung von bestimmten DNA-Nucleotiden gegen RNA-Nucleotide [Hoheisel96].

Neben der Schmelztemperatur kann es auch noch von Interesse sein, die *Hybridisierungskinetik* zu berücksichtigen, d. h. auch das zeitliche Verhalten der DNA-Sequenzen *in vitro* schon im Vorfeld während der Generierung zu bestimmen.

Um dem Problem der Homologie bei dem hier gewählten *uniqueness*-Begriff zu begegnen, kann man zur Sequenzerzeugung die Menge der zu verwendenden Basissequenzen auf eine Teilmenge beschränken, deren Basissequenzen einen bestimmten minimalen Hamming-Abstand (oder eine H-Distanz) untereinander aufweisen.

Eine andere, im Hinblick auf Anwendungsmöglichkeiten sinnvolle Erweiterung ist die Möglichkeit zum Einfügen bestimmter Subsequenzen in die Algomere unter Berücksichtigung der *uniqueness*. Dies können z. B. *stop codons* sein, die ein Auslesen und Verarbeiten der DNA-Informationen *in vivo* abbrechen und die Sequenzen somit auch in biologischen Systemen unbedenklich machen. Andere, für Anwendungen *in vitro* interessante Subsequenzen sind Restriktionsschnittstellen und Proteinbindungsstellen.

Um die Benutzerfreundlichkeit des Compilers zu erhöhen, sollte der Benutzer bei der Parameterwahl unterstützt werden. So sollte er z. B. bei der Wahl von wenig sinnvollen Parametern gewarnt werden. Denkbar ist auch eine Hilfsfunktion, die eine geeignete Parameterwahl vorschlägt oder die in 4.2.6 beschriebene Vorgehensweise automatisiert.

Hierzu müssen auch noch weitere Untersuchungen über die Auswirkungen der Parameterwahl gemacht werden, insbesondere praktische.

Weitere mögliche Erweiterungen der Implementierung umfassen eine Online-Hilfe, eine *multithreading*-Architektur sowie eine Portierung der Software auf andere Plattformen, z. B. UNIX oder Apple Macintosh.

Das programmierte (lineare) *self-assembly* sollte in größerem Umfang theoretisch wie praktisch untersucht werden. Speziell das Ausmaß der Auswirkungen der tolerierten *uniqueness*-Verletzungen auf das Hybridisierungsverhalten und das Problem der vollständigen Abdeckung

eines Lösungsraums für Suchprobleme sind Aspekte, in denen praktische Erfahrung und / oder theoretische Abschätzungen nötig sind.

Eine Erweiterung im größeren Maßstab wäre die Übersetzung von allgemeineren Grammatiken, die damit auch nichtlineares *self-assembly* ermöglichen würde, wozu man aber die Form der Algomere entsprechend anpassen müßte. Läßt man z. B. Verzweigungen (Dreier-*junctions*) und *hairpin loops* zu, so lassen sich Grammatiken vom Typ Chomsky-2 darstellen, kompliziertere, zweidimensionale DNA-Strukturen erlauben angeblich sogar universelle Berechnungen [Winfree96]. Hier sind auch noch Anwendungen für die so hergestellten DNA-Strukturen zu finden, insbesondere geeignete Verfahren zum Auslesen.

Eine sicherlich lohnenswerte Anstrengung ist schließlich der Versuch, sich von der durch die Wirkungsweise von Transistoren als Schalter geprägte Sicht der Informationsverarbeitung zu lösen. Anstatt zu versuchen, Rechenkonzepte von Silizium-basierten Computern in DNA umzusetzen, kann man das der DNA eigene Verhalten und ihre Eigenschaften direkter nutzen, wie in dieser Arbeit gezeigt wurde.

7 Literatur

[Adleman94] L. M. Adleman (1994), „Molecular Computation of Solutions to Combinatorial Problems“, *Science*, Vol. **266**, 1021-1024.

[Adleman96] L. M. Adleman (1996), „On Constructing a Molecular Computer“, *DNA Based Computers*, (E. B. Baum und R. J. Lipton, Hrsg.), DIMACS: Series in Discrete Mathematics and Theoretical Computer Science, American Mathematical Society, Vol. **27.**

[Adleman98] L. M. Adleman (1998), „Rechnen mit DNA“, *Spektrum der Wissenschaft*, November 1998, 70-77.

[Amos96] M. Amos, A. Gibbons, D. Hodgson (1996), „Error-resistant Implementation of DNA Computations“, Research Report CS-RR-298, Department of Computer Science, University of Warwick, Coventry CV4 7AL, England.

[Amos97] M. Amos, P. E. Dunne (1997), „DNA Simulation of Boolean Circuits“, Technical Report CTAG-97009, Department of Computer Science, University of Liverpool.

[Amos98] M. Amos, P. E. Dunne, A. Gibbons (1998), „Efficient Time and Volume DNA Simulation of CREW PRAM Algorithms“, Technical Report CTAG-98006, Department of Computer Science, University of Liverpool.

[Bach96] E. Bach, A. Condon, E. Glaeser, C. Tanguay (1996), „DNA Models and Algorithms for NP-complete Problems“, *Proceedings of the 11th Conference on Computational Complexity*, IEEE Computer Society Press, 290-299. Zitiert in [Niehaus98].

[Baum95] E. B. Baum (1995), „DNA Sequences Useful for Computation“, *Second Anuual Meeting on DNA Based Computers*, Princeton University 1996, DIMACS: Series in Discrete Mathematics and Theoretical Computer Science, American Mathematical Society, 122-127.

[Beaver95] D. Beaver (1995), „Molecular Computing“, Technical Report TR95-001, Penn State University.

[Boneh95a] D. Boneh, R. J. Lipton (1995), „Making DNA Computers Error Resistant“, Technical Report 491-95, Princeton University.

[Boneh95b] D. Boneh, C. Dunworth, R. J. Lipton (1995), „Breaking DES Using a Molecular Computer“, Technical Report CS-TR-489-95, Princeton University.

[Boneh96] D. Boneh, C. Dunworth, R. J. Lipton, J. Sgall (1996), „On The Computational Power of DNA“, *Discrete Applied Mathematics*, Special Issue on Computational Molecular Biology, Vol. **71**, 79-94.

[Breslauer86] K. J. Breslauer, R. Frank, H. Blöcker, L. A. Marky (1986), „Predicting DNA duplex stability from the base sequence“, *Proc. Natl. Acad. Sci.*, Vol. **83**, 3746-3750.

[Cai96] W. Cai, A. E. Condon, R. M. Corn, E. Glaser, Z. Fei, T. Frutos, Z. Guo, M. G. Lagally, Q. Liu, L. M. Smith, A. Tiehl (1996), „The Power of Surface-Based DNA Computation“, unveröffentlicht, Vorabversion erhältlich unter: ftp://corninfo2.chem.wisc.edu/Papers/powerDNA.ps

[Clelland99] C. T. Clelland, V. Risca, , C. Bancroft, (1999), „Hiding messages in DNA microdots“, *Nature*, Vol. **399**, 533-534.

[Deaton95] R. Deaton, R. C. Murphy, M. Garzon, D. T. Franceschetti, S. E. Stevens Jr. (1995), „Good Encodings for DNA-based Solutions to Combinatorial Problems“, *Second Annual Meeting on DNA Based Computers*, Princeton University 1996, DIMACS: Series in Discrete Mathematics and Theoretical Computer Science, American Mathematical Society, 159-171.

[Deaton97] R. Deaton, R. C. Murphy, M. Garzon, D. R. Franceschetti, S. E. Stevens Jr., P. Neathery (1997), „A New Metric for DNA Computing“, *Genetic Programming 97: Proceedings of the Second Annual Conference*, (J. R. Koza, K. Deb, M. Dorigo, D. B. Fogel, M. Garzon, H. Iba, R. L. Riolo, Hrsg.), Stanford University, San Francisco, CA: Morgan Kaufmann, 472-478.

[Gassen96] H. G. Gassen, K. Minol (1996), *Gentechnik*, 4. Auflage, Gustav Fischer Verlag, Stuttgart, UTB 1290.

[Genosys] URL: http://www.genosys.com/tec_mel.htm

[Genset] URL: http://www.genset.fr/Tech_Info/tech_info.html

[Gray96] J. M. Gray, T. G. Frutos, A. M. Berman, A. E. Condon, M. G. Lagally, L. M. Smith, R. M. Corn (1996), „Reducing Errors in DNA Computing by Appropriate Word Design“, unveröffentlicht, Vorabversion erhältlich unter ftp://corninfo2.chem.wisc.edu/Papers/wdesign.ps

[Guarnieri96] F. Guarnieri, M. Fliss, C. Bancroft (1996), „Making DNA Add“, *Science*, Vol. **273**, 220-223.

[Gupta97] V. Gupta, S. Parthasarathy, M. J. Zaki (1997), „Arithmetic and Logic Operations with DNA“, *Proceedings of the 3rd DIMACS Workshop on DNA Based Computers*, DIMACS: Series in Discrete Mathematics and Theoretical Computer Science, American Mathematical Society, 212-220.

[Hennig95] W. Hennig (1995), *Genetik*, Springer-Verlag, Berlin, Heidelberg.

[Hoheisel96] J. D. Hoheisel (1996), „Sequence-independent and linear variation of oligonucleotide DNA binding stabilities“, *Nucleic Acids Research*, Vol. **24**, No. 3, 430-432.

[Kaplan96] P. D. Kaplan, G. Cecchi, A. Libchaber (1996), „DNA based molecular computation: template-template interactions in PCR“, *Second Annual Meeting on DNA Based Computers*, Princeton University 1996, DIMACS: Series in Discrete Mathematics and Theoretical Computer Science, American Mathematical Society.

[Kari96] L. Kari, G. Paun, G. Rozenberg, A. Salomaa, S. Yu (1996), „DNA Computing, Matching Systems, and Universality“, Technical Report No. 49, Turku Centre for Computer Science.

[Knippers95] R. Knippers (1995), *Molekulare Genetik*, 6. Auflage, Georg Thieme Verlag, Stuttgart, New York.

[LeierRichter99] A. Leier, C. Richter, H. Rauhe (1999), „Cryptography with DNA binary strands“, eingereicht.

[Lipton95] R. J. Lipton (1995), „DNA Solution of Hard Computational Problems“, *Science*, Vol. **268**, 542-545.

[Liu96] Q. Liu, Z. Guo, A. E. Condon, R. M. Corn, M. G. Lagally, L. M. Smith (1996), „A Surface-Based Approach to DNA Computation“, *Genetic Programming 97: Proceedings of the Second Annual Conference*, (J. R. Koza, K. Deb, M. Dorigo, D. B. Fogel, M. Garzon, H. Iba, R. L. Riolo, Hrsg.), Stanford University, San Francisco, CA: Morgan Kaufmann.

[Marathe99] A. Marathe, A. E. Condon, R. M. Corn (1999), „On Combinatorial DNA Word Design“, unveröffentlicht, Vorabversion erhältlich unter http://corninfo.chem.wisc.edu/writings/DNAcomputing.html#Papers oder ftp://corninfo2.chem.wisc.edu/Papers/combo.ps

[Maley98] C. C. Maley (1998), „DNA Computation: Theory, Practice, and Prospects“, *Evolutionary Computation*, Vol. **6**, No. 3, 201-229.

[Niehaus98] J. Niehaus (1998), „DNA-Computing: Bewertung und Simulation“, Diplomarbeit am Fachbereich Informatik der Universität Dortmund, Lehrstuhl XI.

[Oligocalc] URL: http://www.basic.nwu.edu/biotools/oligocalc.html

[Rauhe99] H. Rauhe, G. Vopper, W. Banzhaf, J. C. Howard (1999), „Programmable Polymers“, eingereicht.

[Rose97] J.A. Rose, Y. Gao, M. Garzon, R. C. Murphy, R. Deaton, D. R. Franceschetti, S. E. Stevens Jr. (1997), „DNA Implementation of Finite-State Machines“, *Genetic Programming 97: Proceedings of the Second Annual Conference*, (J. R. Koza, K. Deb, M. Dorigo, D. B. Fogel, M. Garzon, H. Iba, R. L. Riolo, Hrsg.), Stanford University, San Francisco, CA: Morgan Kaufmann, 479-487.

[Rothemund96] P. W. K. Rothemund (1996), „A DNA and restriction enzyme implementation of Turing machines“, *DIMACS: Series in Discrete Mathematics and Theoretical Computer Science*, Vol. **27**: *DNA based Computers*, American Mathematical Society.

[Roweis96] S. Roweis, E. Winfree, T. Burgoyne, N. V. Chalyapov, M. F. Goodman, P. W. K. Rothmund, L. M. Adleman (1996), „A Sticker Based Model for DNA Computation“, *Second Annual Meeting on DNA Based Computers*, Princeton University 1996, DIMACS: Series in Discrete Mathematics and Theoretical Computer Science, American Mathematical Society, unveröffentlicht. Zitiert in [Niehaus98].

[Rychlik89] W. Rychlik, R. E. Rhoads (1989), „A computer program for choosing optimal oligonucleotides for filter hybridization, sequencing and in vitro amplification of DNA“, *Nucleic Acids Research*, Vol. **17**, No. 21, 8543-8551.

[SantaLucia96] J. SantaLucia Jr., H. T. Allawi, P. A. Seneviratne (1996), „Improved Nearest-Neighbor Parameters for Predicting DNA Duplex Stability“, *Biochemistry*, Vol. **35**, No. 11, 3555-3562.

[Seeman83] N. C. Seeman, N. R. Kallenbach (1983), „Design of immobile Nucleic Acid Junctions“, *Biophys. J.*, Vol. **44**, 201-209.

[Seeman90] N. C. Seeman (1990), „De Novo Design of Sequences for Nucleic Acid Structural Engineering“, *Journal of Biomolecular Structure & Dynamics*, Vol. **8**, No. 3, 573-581.

[Sen90] D. Sen, W. Gilbert (1990), „A sodium-potassium switch in the formation of four-stranded G4-DNA“, *Nature*, Vol. **344**, 410-414.

[Shapiro99] E. Shapiro (1999), „A mechanical Turing Machine: Blueprint for a Biological Computer“, Vortrag bei *DIMACS: 5th International Meeting on DNA Based Computers*. Umfangreiche Informationen unter http://www.wisdom.weizmann.ac.il/users/udi/public_html/DNA5/index.html

[Stryer94] L. Stryer (1994), *Biochemie*, 2. korr. Nachdruck, Spektrum Akademischer Verlag GmbH, Heidelberg, Berlin, Oxford.

[Sugimoto96] N. Sugimoto, S. Nakano, M. Yoneyama, K. Honda (1996), „Improved thermodynamic parameters and helix initiation factor to predict stability of DNA duplexes“, *Nulceic Acids Research*, Vol. **24**, No. 22, 4501-4505.

[Sundquist89] W. I. Sundquist, A. Klug (1989), „Telomeric DNA dimerizes by formation of guanine tetrads between hairpin loops“, *Nature*, Vol. **342**, 825-829.

[Wegener93] I. Wegener (1993), *Theoretische Informatik*, Teubner, Stuttgart.

[Winfree96] E. Winfree, X. Yang, N. C. Seeman (1996), „Universal Computation via Self-assembly of DNA: Some Theory and Experiments“, *Second Annual Meeting an DNA Based Computers*, Princeton University 1996, DIMACS: Series in Discrete Mathematics and Theoretical Computer Science, American Mathematical Society.

[www] Seite des World Wide Web mit inzwischen nicht mehr gültiger URL.

8 Anhang

8.1 Dokumentation

8.1.1 Allgemeines

Für die Entwicklung des DNA-Sequenz-Compilers wurde ein objekt-orientierter Ansatz gewählt. Der Compiler wurde auf einem PC unter MS Windows 95 entwickelt, unter Benutzung von MS Visual C++ und MS Visual SourceSafe. Verwendet wurde außerdem die Klassenbibliothek SYSTEMS von Hilmar Rauhe.

Die Module SYSUniqueDNAGen und SYSDNACompiler sind gemäß des Konzepts der Bibliothek SYSTEMS plattformunabhängig geschrieben, eine Portierung auf andere Plattformen würde also nur eine Portierung der grundlegenden Klassen von SYSTEMS erfordern. Alleine die Benutzeroberfläche und deren Module sind spezifisch für MS Windows.

8.1.2 Module

Neben den Windows- und MFC(Microsoft Foundation Classes)-spezifischen Modulen für die Oberfläche und denen der Klassenbibliothek SYSTEMS von Hilmar Rauhe sind die wichtigsten Module die folgenden:

8.1.2.1 SYSssDNA

Diese Objektklasse kapselt einsträngige DNA-Sequenzen in Form von Zeichenketten, wobei nur A, a, C, c, G, g, T und t legale Zeichen sind.

Die wichtigste im Rahmen dieser Arbeit vorgenommene Erweiterung dieser zu SYSTEMS gehörenden Klasse ist die Funktion **MeltingTemperature()**, mit der wie in 4.1.3.1 beschrieben die Schmelztemperatur der Sequenz approximiert wird.

8.1.2.2 SYSUniqueDNAGen

Diese Objektklasse stellt den Sequenzgenerator dar. Die wichtigsten Funktionen sind:

GenerateDNA()

Mit dieser Funktion kann man eine Menge von DNA-Sequenzen erzeugen, wobei die *uniqueness* im Sinne der höchstens einmal zu verwendenden Basissequenzen beachtet wird. Es

gibt zwei Überladungen dieser Funktion, die jeweils auch durch den Funktionaloperator des Objekts aufgerufen werden können, da sie anhand ihrer Parameterlisten unterscheidbar sind.

Die erste bekommt als Parameter die Länge der zu erzeugenden Sequenzen, die Basissequenzlänge sowie optional eine maximale Anzahl zu erzeugender Sequenzen übergeben. Sie generiert die Sequenzen nach dem Niehaus-Verfahren wie in 4.1.2 beschrieben.

Die zweite Überladung erlaubt die Erzeugung von Sequenzen verschiedener Länge, für die ggf. auch verschiedene Basissequenzlängen verwendet werden. Die Längen von Sequenzen und Basissequenzen werden in Form von Vektoren als Parameter übergeben. Außerdem werden in dieser Überladung Einschränkungen bzgl. GC-Gehalt, Schmelztemperatur und Homologie berücksichtigt.

Rückgabewert sind jeweils die Anzahl der erzeugten Sequenzen.

CommonExtension()

Diese Funktion erzeugt Sequenzen, indem Lücken zwischen einrahmenden Sequenzen aufgefüllt werden. Insbesondere beachtet werden dabei die Basissequenzen der Übergänge zwischen den Rahmensequenzen und der Extension, mehrere Rahmensequenzenpaare pro Extension, die parallele Erzeugung mehrerer Extensionen, die ggf. zu tolerierenden spaltenweisen *uniqueness*-Verletzungen und die Beachtung vorgegebener Extensionen (s. 4.2.4.1).

Die Parameter dieser Funktion sind neben den Rahmensequenzen die Länge der zu erzeugenden Extensionen, die Basissequenzlänge sowie die Längen der *violation zones*, in denen die spaltenweise *uniqueness*-Verletzung toleriert werden soll.

Rückgabewert ist die Anzahl der erzeugten Extensionen.

AnalyzeUniqueness()

Um die *uniqueness* von Sequenzen zu analysieren, extrahiert diese Funktion für jede Basissequenzlänge in einem bestimmten Bereich alle Basissequenzen dieser Länge aus den Sequenzen und gibt diese mit Häufigkeit und Position(en) ihres Vorkommens in eine Datei `AnalyzeUniqueness.txt` aus.

Parameter dieser Funktion sind obere und untere Grenze der Basissequenzlängen.

8.1.2.3 SYSDNACompiler

Diese Objektklasse enthält den DNA-Sequenz-Compiler. Die wichtigsten Funktionen sind die folgenden:

CheckRules()

Diese Funktion überprüft die als Parameter übergebene Regelmenge, ob sie korrekt ist, d. h. ob alle Regeln regulär sind, es mindestens eine Start- und eine Endregel gibt usw. Sie wird bei einer Übersetzung von **ParseRules()** aufgerufen, kann aber auch direkt verwendet werden. Zur Korrektheit der Regeln siehe auch 3.1 und 4.2.1.1.

Der Boole'sche Rückgabewert ist TRUE, wenn die Regelmenge korrekt ist, FALSE sonst.

ParseRules()

Diese Funktion ruft zunächst **CheckRules()** auf. Falls dort keine Fehler gefunden werden, extrahiert sie aus den Regeln Terminal- und Variablenmenge. Sie wird bei einer Übersetzung von **Compile()** aufgerufen.

Als Parameter bekommt sie die Regelmenge übergeben. Der Rückgabewert entspricht dem von **CheckRules()**.

Compile()

Dies ist die eigentliche Übersetzungsfunktion. Sie kann auch durch den Funktionaloperator des Objekts aufgerufen werden.

Nach einem Aufruf von **ParseRules()** werden zunächst die Überhangsequenzen (i. a. Restriktionsschnittstellen) sowie die wiederzuverwendenden Terminal- und Variablensequenzen zur Menge der kompatiblen Sequenzen hinzugefügt. Anschließend werden die Funktionen zur Generierung der Terminal- und Variablensequenzen gemäß den in 4.2.4 beschriebenen Strategien aufgerufen. Zuletzt werden aus den Sequenzen die Algomere gebildet und die Ausgabedateien geschrieben.

Als Parameter werden die zu übersetzende Regeln, Referenzen der Sequenzmengen für obere und untere Stränge der Algomere sowie die gewünschte Form der Algomere übergeben.

Der Rückgabewert ist FALSE, wenn ein bei der Übersetzung ein Fehler aufgetreten ist, TRUE sonst. Außerdem befinden sich in den Sequenzmengen, deren Referenzen übergeben wurden, die Einzelstränge der Algomere in der Reihenfolge der Regeln, die sie repräsentieren. Falls mehrere Start- oder Endterminale erzeugt wurden, wurden entsprechend viele Kopien der Start- bzw. Endregeln zur Regelmenge hinzugefügt.

GenerateFreeXxxx(), GenerateMatchingXxxx()

Das **Xxxx** dient als Platzhalter für die Zeichenketten **Variables**, **Terminals**, **StartTerminals** und **EndTerminals**.

Diese Funktionen generieren die Sequenzen der verschiedenen Gruppen. Bei den mit **Matching** bezeichneten Funktionen werden hierbei die Sequenzen der jeweils anderen Gruppe als Rahmensequenzen an **SYSUniqueDNAGen::CommonExtension()** übergeben und die zu erzeugenden Sequenzen als Extensionen generiert.

GenerateFreeVariables() und **GenerateFreeTerminals()** verwenden die zweite Überladung von **SYSUniqueDNAGen::GenerateDNA()**, da hier keine Übergänge zu bereits generierten Sequenzen beachtete werden müssen. **GenerateFreeStartTerminals()** und **GenerateFreeEnd-Terminals()** verwenden wiederum **SYSUniqueDNAGen::CommonExtension()**, da die Übergänge zu den Überhangsequenzen beachtet werden müssen.

Zu den Besonderheiten der Generierung siehe 4.2.4.1, zur Reihenfolge der Aufrufe durch **Compile()** siehe 4.2.4.2.

Als Parameter wird ein intern verwendeter Index der zuletzt hinzugekommenen Sequenzen in der Menge der kompatiblen Sequenzen übergeben, die mit **Matching** bezeichneten Funktionen bekommen außerdem noch die Regelmenge.

Der Rückgabewert ist TRUE, falls genug Sequenzen generiert werden konnten, FALSE sonst.

AnalyzeUniqueness()

Um zu analysieren, in welchem Umfang tatsächlich bei der Übersetzung tolerierte *uniqueness*-Verletzungen aufgetreten sind, ruft diese Funktion **SYSUniqueDNAGen::Analyze-Uniqueness()** auf. Dabei werden die oberen Stränge der Algomere als zu analysierende Sequenzmenge übergeben. Die Ergebnisse werden in die Datei `AnalyzeUniqueness.txt` geschrieben (s. o.).

8.1.3 Die Benutzeroberfläche

Der Compiler wird über Menüs gesteuert. Abbildung 8.1 zeigt das Hauptfenster mit der Menüleiste und der Werkzeugleiste. Letztere bietet schnelleren Zugriff auf einige Menü-einträge. Z. Z. sind nur die Menüs „File“, „Rules“, „Input Sequences“ und „Compiler“ von Bedeutung, die Menüs „Edit“, „View“, „Window“ und „Help“ haben noch keine Funktion.

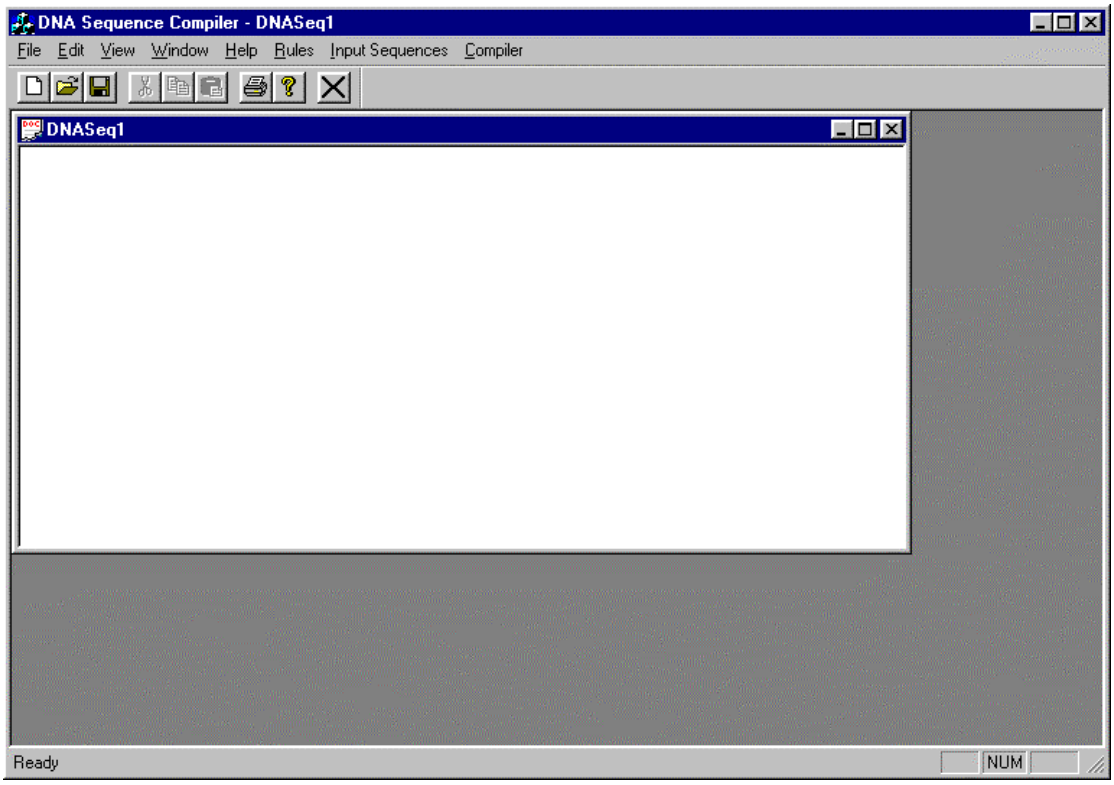


Abbildung 8.1: Das Hauptfenster der Benutzeroberfläche.

8.1.3.1 Das Menü „Files“

Open

Öffnet einen Dateiauswahldialog, um eine Datei mit Compiler-Eingaben zu laden. Diese ist eine Textdatei mit der Endung `.dsc`, die i. a. zuvor mit dem Menüeintrag „Save“ geschrieben wurde. In ihr befinden sich sowohl die Regeln als auch die verschiedenen Eingabesequenzen und die Compileroptionen.

Save

Öffnet einen Dateiauswahldialog, um die Regeln, die Eingabesequenzen sowie die Compiler-optionen in eine Textdatei mit der Endung `.dsc` zu schreiben.

8.1.3.2 Das Menü „Rules“

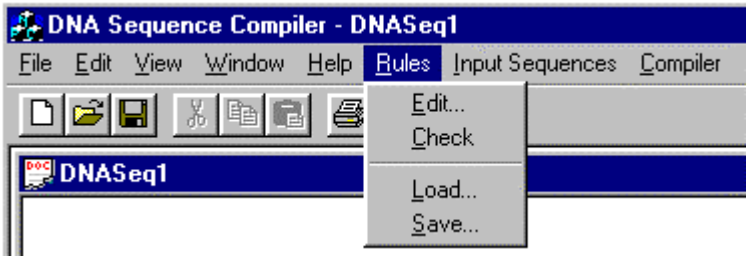


Abbildung 8.2: Das Menü „Rules“.

Edit

Öffnet ein Dialogfenster, in dem die Regelmenge der regulären Grammatik eingegeben werden kann.

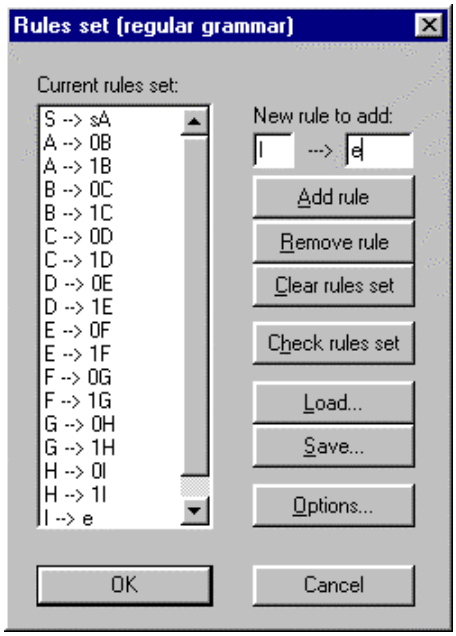


Abbildung 8.3: Dialog zur Eingabe der Regelmenge

In der Liste auf der linken Seite wird die aktuelle Regelmenge angezeigt.

In den beiden Textfeldern oben rechts kann eine neue Regel eingegeben werden, nach linker und rechter Seite der Regel getrennt. Mit dem Taster „Add rule“ wird die Regel in die Liste übernommen. Ist dort bereits ein Eintrag ausgewählt (hervorgehoben), so wird die neue Regel vor diesem Eintrag eingefügt. Ist kein Eintrag ausgewählt, so wird die Regel an das Ende der Liste angehängt.

Mit dem Taster „Remove rule“ wird eine in der Liste ausgewählte Regel entfernt. Mit „Clear rules set“ wird die gesamte Liste gelöscht.

Durch Betätigen des Tasters „Check rules set“ kann der Benutzer durch Aufruf von **SYSDNACompiler::CheckRules()** überprüfen, ob die in der Liste gezeigte Regelmenge korrekt ist.

Der Taster „Load“ öffnet einen Dateiauswahldialog, um Regeln aus einer Textdatei zu laden. Dabei geht die aktuelle Regelmenge verloren. Mit dem Taster „Save“ schreibt man die Regeln der Liste in eine Textdatei, die ebenfalls mit einem Dateiauswahldialog anzugeben ist.

Mit dem Taster „Options“ öffnet man ein weiters Dialogfenster, in dem man die Optionen zur Regelmenge einstellen kann. Hier sind die Zeichen für Startvariable sowie Start- und Endterminal auszuwählen. Die Voreinstellung ist S, s und e.

Mit „OK“ werden die gemachten Änderungen übernommen und der Dialog geschlossen. „Cancel“ schließt den Dialog, ohne die Änderungen zu übernehmen.

Check

Durch diesen Menüeintrag wird **SYSDNACompiler::CheckRules()** aufgerufen, um die aktuelle Regelmenge auf Korrektheit zu überprüfen.

Load

Dieser Menüeintrag öffnet einen Dateiauswahldialog, um eine Regelmenge aus einer Textdatei zu laden.

Save

Dieser Menüeintrag schreibt die aktuelle Regelmenge in eine Textdatei, die zuvor mit einem Dateiauswahldialog ausgewählt wird.

8.1.3.3 Das Menü „Input Sequences“

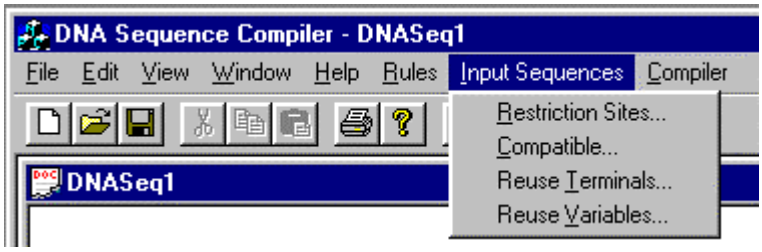


Abbildung 8.4: Das Menü „Input Sequences“.

Restriction sites

Dieser Menüeintrag öffnet ein Dialogfenster, in dem die Restriktionsschnittstellen (bzw. sonstige Überhangsequenzen) für die Terminatoren eingegeben werden können.

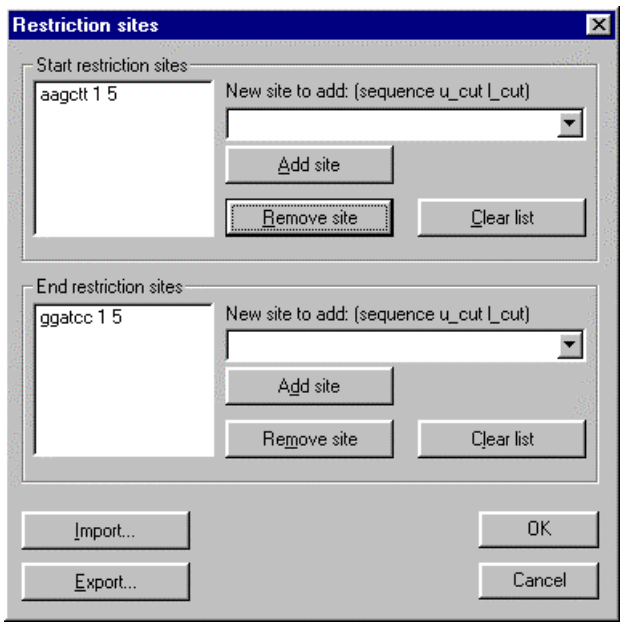


Abbildung 8.5: Dialog zur Eingabe der Restriktionsschnittstellen.

Es gibt jeweils für Start- und Endterminatoren einen eigenen Eingabebereich. In diesen ist links jeweils eine Liste mit den aktuellen Restriktionsschnittstellen. Angezeigt wird jeweils der obere Strang sowie die Positionen, an denen der obere und der untere Strang geschnitten werden, gezählt von 5‘-Ende des oberen Strangs aus.

Im Textfeld rechts oben kann eine neue Restriktionsschnittstelle eingegeben werden. Dabei ist die gleiche Formatierung zu verwenden wie bei der Anzeige in der Liste. Mit dem Taster „Add site“ wird die eingegebene Restriktionsschnittstelle in die Liste übernommen. Ist ein Eintrag in der Liste ausgewählt, wird der neue Eintrag vor diesem eingefügt. Ist kein Eintrag ausgewählt, wird der neue ans Ende der Liste angehängt.

Mit dem Taster „Remove site“ wird ein vorher ausgewählter Eintrag aus der Liste entfernt. „Clear list“ löscht die gesamte Liste.

Der Taster „Import“ öffnet einen Dateiauswahldialog, um Restriktionsschnittstellen (für Start- und Endterminatoren) aus einer Textdatei zu laden. Dabei geht die aktuelle Liste verloren. Mit dem Taster „Export“ werden die Einträge der beiden Listen in eine Textdatei geschrieben, die mit einem Dateiauswahldialog auszuwählen ist.

Mit „OK“ werden die gemachten Änderungen übernommen und der Dialog geschlossen. „Cancel“ schließt den Dialog, ohne die Änderungen zu übernehmen.

Compatible

Dieser Menüeintrag öffnet ein Dialogfenster, in dem die Sequenzen eingegeben werden können, zu denen die neuen, zu erzeugenden kompatibel sein sollen, d. h. die Basissequenzen dieser Sequenzen (und deren Komplementärsequenzen) dürfen nicht zur Erzeugung der Algomere verwendet werden. Restriktionsschnittstellen und von Compiler wiederzuverwendende Sequenzen müssen hier nicht eingegeben werden, da sie automatisch beachtet werden.

Links im Dialog ist eine Liste der aktuellen kompatiblen Sequenzen.

Im Textfeld oben rechts kann man eine neue Sequenz eingeben. Mit dem Taster „Add sequence“ wird diese in die Liste übernommen. Ist eine Sequenz in der Liste ausgewählt, so wird die neue Sequenz vor dieser eingefügt, anderenfalls wird sie an die Liste angehängt.

Mit dem Taster „Remove sequence“ wird eine zuvor ausgewählte Sequenz aus der Liste entfernt. Der Taster „Clear list“ löscht die gesamte Liste.

Der Taster „Import“ öffnet einen Dateiauswahldialog, um Sequenzen aus einer Textdatei zu laden. Dabei geht die aktuelle Liste verloren. Mit „Export“ werden die Sequenzen der Liste in eine Textdatei geschrieben, die in einem Auswahldialog auszuwählen ist.

Mit „OK“ werden die gemachten Änderungen übernommen und der Dialog geschlossen. „Cancel“ schließt den Dialog, ohne die Änderungen zu übernehmen.

Reuse Terminals

Dieser Menüeintrag öffnet ein Dialogfenster mit drei Seiten, zwischen denen man über eine Reiterleiste am oberen Rand des Fensters wählen kann. Auf diesen Seiten kann der Benutzer wiederzuverwendende Sequenzen für echte Terminale, Start- und Endterminale eingeben. Die drei Seiten sind gleich aufgebaut.

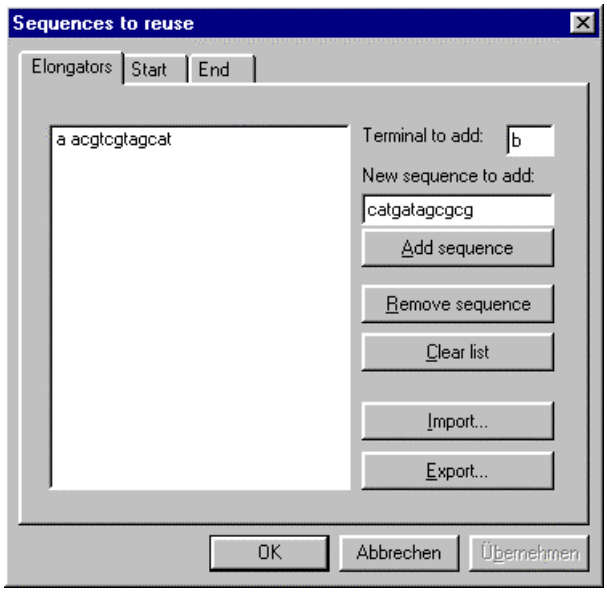


Abbildung 8.6: Dialog zur Eingabe der wiederzuverwendenden Terminalsequenzen.

Links ist eine Liste der wiederzuverwendende Sequenzen der aktuell gewählten Terminalgruppe. Es werden jeweils das Terminal und die entsprechende Sequenz angezeigt.

In den Textfeldern rechts kann ein neuer Eintrag eingegeben werden, wobei im oberen das übersetzte Terminal und im unteren die Sequenz eingegeben wird. Auf den Seiten für Start- und Endterminale ist das obere Feld grau (nicht auswählbar), da klar ist, welches Terminal durch die Sequenz dargestellt werden soll. Mit dem Taster „Add sequence“ wird aus dem Inhalt der beiden Textfelder ein neuer Eintrag in die Liste eingefügt.

Für die weitere Funktionalität der Taster siehe den Menüeintrag „Compatible“.

Wird mit den Reitern von einer Seite zu einer anderen geblättert, so werden die auf der alten Seite gemachten Änderungen übernommen.

Reuse Variables

Dieser Menüeintrag öffnet einen Dialog, in dem die wiederzuverwendenden Variablensequenzen eingegeben werden können. Abgesehen davon, daß er keine verschiedenen Seiten hat, ist dieser Dialog gleich aufgebaut wie die unter dem Menüeintrag „Reuse Terminals“ beschriebenen Dialogseiten.

8.1.3.4 Das Menü „Compiler“

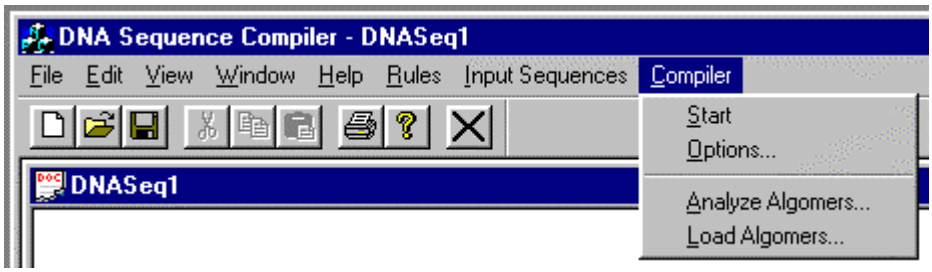


Abbildung 8.7: Das Menü „Compiler“.

Start

Startet den Compiler durch Aufruf von **SYSDNACompiler::Compile()** mit den gewählten Eingaben. Ein Nachrichtenfenster meldet entweder die erfolgreiche Übersetzung sowie anschließend die dafür benötigte Zeit oder aber den Grund für einen Mißerfolg, z. B. welcher Fehler in der Regelmenge vorliegt oder für welche Sequenzgruppe nicht genügend Sequenzen erzeugt werden konnten.

Bei einer erfolgreichen Übersetzung werden die Ergebnisse in die Textdateien `Variables.txt`, `Terminals.txt`, `Algomers_ds.txt` und `Algomers_ss.txt` geschrieben (s. 4.2.2).

Options

Dieser Menüeintrag öffnet ein Dialogfenster in dem die Parameter und Optionen des Compilers eingegeben werden können.

Im oberen Bereich sind die gruppenspezifischen Parameter nach Gruppen geordnet aufgelistet. Darunter befinden sich die globalen Parameter. Zu der Bedeutung der Parameter sowie zur Wahl derselben siehe 4.2.5.

Der Taster „Load“ öffnet einen Dateiauswahldialog, um Optionen aus einer Textdatei zu laden. Mit „Save“ werden die aktuellen Optionen in eine Textdatei geschrieben, die in einem Auswahldialog auszuwählen ist.

Mit „OK“ werden die gemachten Änderungen übernommen und der Dialog geschlossen. „Cancel“ schließt den Dialog, ohne die Änderungen zu übernehmen.

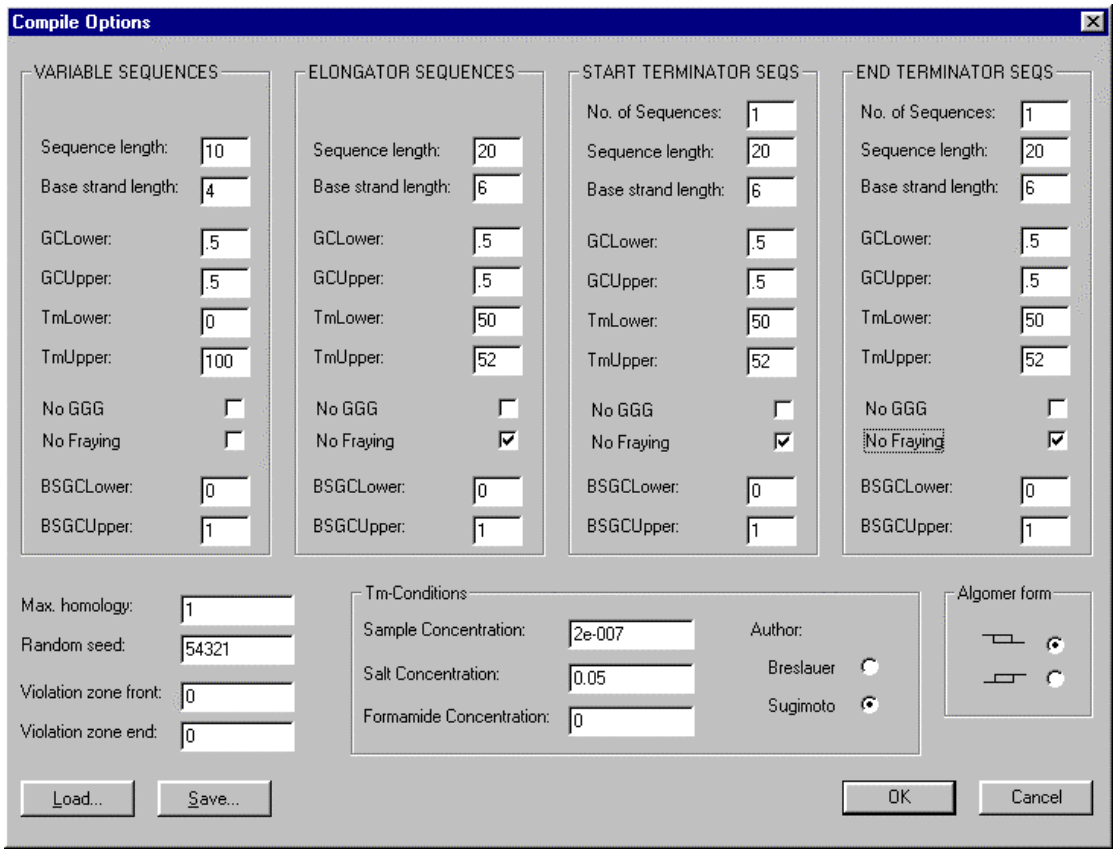


Abbildung 8.8: Dialog zur Eingabe der Parameter.

Analyze Algomers

Mit diesem Menüeintrag werden die Algomere auf ihre *uniqueness* untersucht. Dazu wird **SYSDNACompiler::AnalyzeUniqueness()** aufgerufen. Die untersuchten Algomere stammen entweder aus einer erfolgreichen Übersetzung oder wurden aus einer Datei geladen.

Load Algomers

Dieser Menüeintrag öffnet einen Dateiauswahldialog, um Algomere aus einer Textdatei zu laden. In dieser müssen die Algomere in einem Format angegeben sein, wie es in der Ausgabedatei `Algomers_ss.txt` verwendet wird.

Diese Funktion erlaubt es, bereits erzeugte Algomere auch später noch auf ihre *uniqueness* zu untersuchen.

8.2 Glossar

Algomer: doppelsträngige DNA-Sequenz mit Überhängen an jedem Ende, die eine Regel einer regulären Grammatik repräsentiert.

Basissequenz: Kurze DNA-Sequenz. Mehrere sich überlappende Basissequenzen bilden eine längere Sequenz. *Unique* DNA-Sequenzen bestehen aus Basissequenzen, die höchstens einmal in allen Sequenzen vorkommen.

echte Terminale: die Terminale, die in anderen Regeln als Start- oder Endregel vorkommen.

Elongator: Algomer, das eine andere Regel repräsentiert als Start- oder Endregel.

Endterminal: besonderes Terminal, das nur in Endregeln vorkommt und damit das letzte Terminal eines Ausdrucks der Grammatik ist. Als Default ist e das Endterminal.

Extension: die Sequenz, die als Mittelteil zwischen zwei vorgegebenen Sequenzen durch Auffüllen erzeugt wird.

Gruppe: Die Sequenzen, die die Symbole der Grammatik repräsentieren, werden in Variablen- und Terminalsequenzen unterschieden. Die Terminalsequenzen werden wiederum in die drei Teilgruppen echte, Start- und Endterminale unterteilt.

Homologie: Maß für die Ähnlichkeit zweier ssDNA-Sequenzen, Quotient aus maximaler Anzahl gleicher Basen über alle Verschiebungen und der Länge der kürzeren Sequenz. Liegt zwischen 0 (völlig verschieden) und 1 (identisch bzw. Subsequenz).

kompatible Sequenzen: Sequenzen, zu denen die zu erzeugenden *unique* sein sollen. Dies wird erreicht, indem die Basissequenzen der kompatiblen Sequenzen extrahiert und als benutzt markiert werden.

Logomer: dsDNA-Sequenz, die einen Ausdruck einer regulären Grammatik repräsentiert. Besteht aus Algomeren.

Niehaus-Verfahren: Graph-basiertes Verfahren zur Konstruktion von Sequenzen aus Basissequenzen, so daß jede Basissequenz (incl. ihre Komplementärsequenz) höchstens einmal vorkommt.

Rahmensequenzen: vordere und hintere vorgegebene Sequenzen, zwischen denen durch Auffüllen eine weitere Sequenz (die Extension) erzeugt wird, wobei die Übergänge zwischen Extension und Rahmensequenzen für die *uniqueness* berücksichtigt werden.

Spalte, spaltenweise Verletzung der *uniqueness*: Eine *Spalte* bezeichnet die Menge der aktuellen (zuletzt hinzugefügten) Basissequenzen aller parallel erzeugten Sequenzen zu einem Zeitpunkt der Erzeugung. In einer Spalte kann ggf. mehrfaches Vorkommen von Basissequenzen toleriert werden, da dies zur Übersetzung der Regeln notwendig sein kann.

Startterminal: besonderes Terminal, das nur in Startregeln vorkommt und damit das erste Terminal eines Ausdrucks der Grammatik ist. Als Default ist s das Startterminal.

Terminator: Algomere, die Start- oder Endregeln repräsentieren und die Ligation von Algomeren zu Logomeren beenden.

Terminalgruppen: Die Gruppe der Terminalsequenzen ist wiederum unterteilt in drei Terminalgruppen mit Sequenzen für echte, Start- und Endterminale.

Übergang: die Basissequenzen, die durch Konkatenation zweier Sequenzen entstehen, aber zu keiner der beiden einzelnen Sequenzen gehören.

Überhang: einsträngiges Ende einer dsDNA-Sequenz, dient als *sticky end*, also um die Anlagerung einer anderen Sequenz mit komplementärem Überhang zu ermöglichen.

***uniqueness*:** Eindeutigkeit (möglichst große Unähnlichkeit) von Sequenzen, hier: jede Basissequenz (incl. deren Komplementärsequenz) darf höchstens einmal in allen Sequenzen vorkommen.

***violation zone*:** Bereiche an Anfang und Ende von Extensionen, in denen spaltenweise Verletzung der *uniqueness* toleriert wird. Die Länge gibt die Anzahl von Spalten an, in denen solche Verletzungen auftreten dürfen, gezählt ab der jeweils äußersten Basissequenz des Übergangs.

8.3 Errata

Wenn die spaltenweise Verletzung der *uniqueness* durch gleiche Extensionen mit mehreren verschiedenen Rahmensequenzen nötig wird, so muß die *violation zone* selbstverständlich an den inneren Enden der Übergänge (also an der Extension) beginnen. Leider wurde dies durch einen Programmierfehler bei der Implementierung nicht berücksichtigt, was erst zu spät bemerkt wurde. Dies erklärt allerdings die nötigen Längen der *violation zones* von Basissequenzlänge - 1, wenn der geschilderte Fall eintrat. Wegen der Reproduzierbarkeit der vorgestellten Ergebnisse wird dieser Fehler erst in einer späteren Programmversion behoben werden.