Diplomarbeit

Thema:
„Aufzählen von DNA-Codes “

Daniel Bärmann
706255

Themenersteller und 1. Gutachter	2. Gutachter
Prof. Dr. Jürgensen	Dr. Henning Bordihn

Werder/Havel, 24. Juli 2006

INHALTSVERZEICHNIS

1. EINLEITUNG

Das Ziel der vorliegenden Arbeit ist das Finden von DNA-Codes mit bestimmten Eigenschaften. Diese Eigenschaften und weitere formale Definitionen werden in Kapitel 2 eingeführt. Das dritte Kapitel ist der Entwicklung eines Modells zum Aufzählen von DNA-Codes gewidmet. Es werden Überlegungen angstellt, die ein möglichst effizientes Testen von Eigenschaften erlauben sollen. In diesem Kapitel wird auch ein im Rahmen dieser Arbeit entstandenes Werkzeug vorgestellt, dass das (interaktive) Aufzählen von Codes mit beliebigen Kombinationen von Eigenschaften erlaubt. Im vierten Kapitel werden Betrachtungen zu speziellen DNA-Codes gemacht. Insbesondere die Optimalität von Codes bezüglich ihrer Informationsrate steht hier im Vordergund. Im Anhang finden sich Tabellen von Codes mit ausgewählten Eigeschaften, die von besonderem Interesse sein können. Es handelt sich um solide Codes, die bei der Konstruktion fehlererkennender DNA-Codes von Bedeutung sind.

2. DEFINITIONEN UND NOTATION

In dieser Arbeit werden einsträngige DNA-Moleküle als Wörter über dem Alphabet $\Delta = \{a, c, g, t\}$ und Mengen von Molekülen als Codes aufgefasst. Daher ist es notwendig zunächst eine Notationen festzulegen.

Ein Alphabet ist eine nicht-leere, endliche Menge von Symbolen. Im folgenden bezeichnet das Symbol Σ ein beliebiges Alphabet.
Die Menge aller Wörter über Σ wird mit Σ^* bezeichnet. Diese Menge schließt das leere Wort λ ein.

Mit Σ^+ wird die Menge aller Wörter ohne λ bezeichnet.
Ein Code ist eine Teilmenge von Σ^+.
Die Länge eines Wortes w wird mit $|w|$ notiert. $|w|_x$ bezeichnet die Anzahl der Vorkommen von x in w für $w \in \Sigma^*$ und $x \in \Sigma^+$.
Das Wort w^n besteht aus n konkatenierten Kopien des Wortes w für $n \geq 0$, wobei w^0 das leere Wort λ ist.
Das Spiegelbild eines Wortes w wird mit w^R notiert.
Ein von λ verschiedenes Wort u heißt Infix von w, wenn $w = xuy$ für $x, y \in \Sigma^*$
u heißt echtes Infix von w, falls $|x| + |y| > 0$.
Analog heißt u Präfix (Suffix) von w, falls $w = ux$ ($w = xu$) mit $x \in \Sigma^*$, bzw. echtes Präfix (Suffix) von w, falls $|x| > 0$.
Die Menge aller Infixe eines Wortes w wird mit $\mathit{Infix}(w)$ notiert. Zur Bezeichnung der Menge aller k-buchstabigen Infixe von w ($k > 0$), wird diese Schreibweise verwendet: $\mathit{Infix}_k(w)$.
Die Mengen aller Prä- bzw. Suffixe werden mit $\mathit{Prefix}(w)$ bzw. $\mathit{Suffix}(w)$ bezeichnet, die entsprechenden Mengen der k-buchstabigen Prä- und Suffixe mit $\mathit{Prefix}_k(w)$ bzw. $\mathit{Suffix}_k(w)$.

Eine Teilmenge L von Σ^* heißt Sprache über Σ.
Mit L^n, $L \subseteq \Sigma$, ($n \geq 0$) wird die Sprache aller n-buchstabigen Wörter über

Σ bezeichnet, d.h. für alle $w \in L$ gilt: $w = a_1 \dots a_n$.
Mit $L^R = \{w^R | w \in L\}$ wird das Spiegelbild von L bezeichnet.
Infix(L) bezeichnet die Menge aller Infixe von L, d.h. *Infix*$(L) = \bigcup_{w \in L}$ *Infix*(w).

Sei A eine beliebige Menge. Eine Involution ist eine Abbildung $\theta : A \to A$, so dass für alle Elemente $x \in A$ gilt: $\theta(\theta(x)) = x$.

Bezogen auf das DNA-Alphabet kann jeder DNA-Strang als Wort über Δ aufgefasst werden.
Die Enden eines DNA-Strangs unterscheiden sich chemisch voneinander. Das gemäß der chemischen Nomenklatur als 5'-Ende bezeichnete Ende eines Strangs entspricht dem Anfang des Wortes, das als 3'-Ende bezeichnete Ende dem Ende des Wortes.
Die natürliche Komplementarität der Basen kann durch die Involution $\tau : \Delta \to \Delta$ mit $\tau(a) = t, \tau(c) = g, \tau(g) = c$ und $\tau(t) = a$ wiedergegeben werden.
Durch Erweitern dieser Involution von Δ zu einem Antimorphismus von Δ^* wird die Komplementarität ganzer DNA-Stränge modelliert:
Sei $w \in \Delta^*, w = a_1 \dots a_n$. Dieses Wort repräsentiert den DNA-Strang $5' - a_1 a_2 \dots a_n - 3'$.
Das zu w komplementäre Wort ist $\tau(w) = \tau(a_1 \dots a_n) = \tau(a_n) \dots \tau(a_1)$, bzw. der DNA-Strang $5' - \tau(a_n) \dots \tau(a_1) - 3'$.
Demnach sind zwei einzelne Stränge w_1 und w_2 genau dann komplementär, wenn $w_1 = \tau(w_2)$.

2.1 Erwünschte Eigenschaften von DNA-Codes

Das Rechnen mit DNA-Molekülen basiert auf der chemischen Interaktion der Moleküle. Durch die Komplementarität der Basenpaare können sich zwei einsträngige DNA-Moleküle zu einem Doppelstrang verbinden (Hybridisierung). Treten komplementäre Sequenzen in einem einzelnen Strang auf, kann sich dieser zu einer haarnadelartigen Struktur verschlingen. Derartige Rekationen sind für das Rechnen mit DNA-Molekülen unerwünscht, da die entstehenden Produkte für den weiteren Rechenprozess nicht geeignet sind.
Um sicherzustellen, dass Reaktionen in einer Art und Weise ablaufen, die zu korrekten Ergebnissen führt, wurden in [HKK03, KKT00, KKLW03] Vorschläge gemacht, die im Folgenden wiedergegeben werden.

In den folgenden Definitionen sei θ eine beliebige Involution.

Definition 1. *Eine DNA-Sprache $L \subseteq \Sigma^+$ heißt θ-**überlappungsfrei**, falls $L \cap \theta(L) = \emptyset$.*

Dies bedeutet, dass eine entsprechende Menge von DNA-Strängen keine zueinander komplementären Stränge enthält.

Definition 2. *Eine DNA-Sprache $L \subseteq \Sigma^+$ heißt θ-**konform**, falls für alle $w \in L$ und alle $x, y \in \Sigma^*$ gilt: sind w und $x\theta(w)y$ in L, so ist $xy = \lambda$.*

In diesem Fall enthält die entsprechende Menge von DNA-Strängen keine Stränge, deren echte Infixe komplementär zu einem anderen Strang sind.

Definition 3. *Eine DNA-Sprache $L \subseteq \Sigma^+$ heißt θ-**p-konform**, falls für alle $w \in L$ und alle $y \in \Sigma^*$ gilt: sind w und $\theta(w)y$ in L, so ist $y = \lambda$.*

Diese Eigenschaft erlaubt keine Stränge, deren Präfixe komplementär zu anderen Strängen sind.

Definition 4. *Eine DNA-Sprache $L \subseteq \Sigma^+$ heißt θ-**s-konform**, falls für alle $w \in L$ und alle $x \in \Sigma^*$ gilt: sind w und $x\theta(w)$ in L, so ist $x = \lambda$.*

Analog zu Definition 3 verhindert diese Eigenschaft das Vorkommen von Strängen, deren Suffixe komplementär zu anderen Strängen sind.

Definition 5. *Eine DNA-Sprache $L \subseteq \Sigma^+$ heißt* ***streng θ-konform****, falls L θ-konform und θ-überlappungsfrei ist.*

Die folgende Eigenschaft unterbindet die Hybridisierung von DNA-Strängen an Konkatenationsstellen:

Definition 6. *Eine DNA-Sprache $L \subseteq \Sigma^+$ heißt* ***θ-kommafrei****, falls für alle $w \in L$ gilt: $L^2 \cap \Sigma^+\theta(L)\Sigma^+ = \emptyset$.*

Definition 7. *Eine DNA-Sprache $L \subseteq \Sigma^+$ heißt* ***θ-sticky-frei****, falls für alle $w \in \Sigma^+$ und alle $x, y \in \Sigma^*$ gilt: sind wx und $y\theta(w)$ in L, so ist $xy = \lambda$.*

Diese Eigenschaft erlaubt keine Stränge deren 3'- und 5'-Ende komplementär zueinander ist.

Definition 8. *Eine DNA-Sprache $L \subseteq \Sigma^+$ heißt* ***θ-3'-überhangfrei****, falls für alle $w \in \Sigma^+$ und alle $x, y \in \Sigma^*$ gilt: sind wx und $\theta(w)y$ in L so ist $xy = \lambda$.*

Diese Eigenschaft verhindert die Hybridisierung an den 5'-Enden zweier Stränge, so dass die 3'-Enden unhybridisiert bleiben.

Die folgende Definition verhindert den selben Vorgang am 3'-Ende zweier Stränge.

Definition 9. *Eine DNA-Sprache $L \subseteq \Sigma^+$ heißt* ***θ-5'-überhangfrei****, falls für alle $w \in \Sigma^+$ und alle $x, y \in \Sigma^*$ gilt: sind xw und $y\theta(w)$ in L, so ist $xy = \lambda$.*

Definition 10. *Eine DNA-Sprache $L \subseteq \Sigma^+$ heißt* ***θ-überhangfrei****, falls L θ-5'- und θ-3'-überhangfrei ist.*

Definition 11. *Seien k, m, m_1 und m_2 natürliche Zahlen mit $k > 0$ und $m_1 \leq m \leq m_2$.*
Eine DNA-Sprache $L \subseteq \Sigma^+$ heißt ***$\theta(k, m_1, m_2)$-teilwortkonform****, falls für alle $u \in \Sigma^k$ gilt: $\Sigma^* u \Sigma^m \theta(u) \Sigma^* \cap L = \emptyset$.*

Mit dieser Eigenschaft wird eine intramolekulare Hybridisierung verhindert, in dem nur Stränge, in denen zu einander komplementäre k lange Teilstücken weniger als m_1 oder mehr als m_2 Basen von einander entfernt sind, zugelassen sind.

Definition 12. *Sei $k > 0$. Eine DNA-Sprache $L \subseteq \Sigma^+$ ist ein θ-k-**Code**, falls $Infix_k(L) \cap Infix_k(\theta(L)) = \emptyset$.*

In diesem Fall gibt es zu keinem k langen Teilstück eines Strangs ein komplementäres in einem anderen Strang.

Definition 13. *Sei L eine DNA-Sprache und sei w ein nicht leeres Wort über dem Alphabet Σ. Eine Zerlegung des Wortes heißt L-Zerlegung, falls $w = x_1y_1 \dots x_ny_ny_{n+1}$ mit $Infix(x_i) \cap L = \emptyset$ und $y_j \in L$ für $i = 1, 2, \dots, n+1$ und $j = 1, 2, \dots, n$.*
L heißt solider DNA-Code, falls es zu jedem Wort aus Σ^+ genau eine L-Zerlegung gibt.

Dies bedeutet, dass kein Codewort Teil eines anderen sein darf und dass der Beginn eines Wortes mit dem Ende keines anderen übereinstimmen darf und umgekehrt.
Bezogen auf eine beliebige Sequenz von Basen bedeutet dies, dass in ihr enthaltene Codewörter eindeutig erkannt werden können.

Definition 14. *Eine DNA-Sprache L heißt kommafrei, falls $L^2 \cap \Sigma^+L\Sigma^+ = \emptyset$.*

Dies bedeutet, dass kein Codewort Teil eines anderen Codeworts oder der Konkatenation zweier Codewörter sein darf.

Die folgenden Abbildungen zeigen die durch die verschiedenen Codeeigenschaften verbotenen Strukturen.

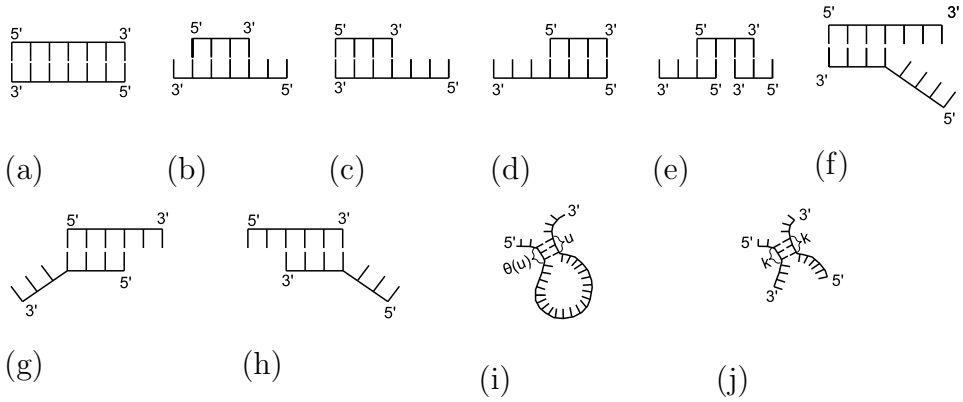

Fig. 2.1: (a) Überlappungsfreiheit, (b) θ-Konformität, (c) θ-p-Konformität, (d) θ-s-Konformität, (e) θ-Kommafreiheit, (f) θ-Sticky-Freiheit, (g) θ-3'-Überhangfreiheit, (h) θ-5'-Überhangfreiheit, (i) θ-Teilwortkonformität, (j) θ-k-Code

3. AUFZÄHLEN VON DNA-CODES

Das Ziel dieser Arbeit ist es, DNA-Codes mit bestimmten Eigenschaften aufzuzählen. Um Codes aufzählen zu können ist ein Modell notwenig, welches die dafür erforderlichen Mittel und mathematischen Formalismen bereitstellt. Ein derartiges Modell wurde bereits in [Her05] für binäre Codes geschaffen. Dieses Modell soll hier erweitert werden.

Die Grundlage des genannten Modells ist eine Ordnung auf dem verwendeten Alphabet, der Menge der möglichen Codewörter sowie der Menge der Codes.
Zunächst wird daher eine Ordnung auf dem verwendeten Alphabet definiert. Für das DNA-Alphabet $\Delta = \{a, c, g, t\}$ mit der Ordnungsrelation $<$ soll gelten:

$$a < c < g < t$$

Diese Beziehung dient als Grundlage für das Einführen einer Ordnung auf der Menge der möglichen Codewörter Δ^*.

Mithilfe der Ordnung auf Δ und eines ordnungserhaltenden Isomorphismus $\beta : \Delta \to \{1, 2, 3, 4\}$ bzw. seiner Erweiterung $\beta^* : \Delta^* \to \{1, 2, 3, 4\}^*$ wird nun eine Abbildung der Wörter aus Δ^* auf folgende Darstellung definiert:

$$\beta^*(b_{n-1} \dots b_0) = \beta(b_{n-1}) \dots \beta(b_0)$$

wobei

$$\begin{aligned} \beta(a) &= 1 \\ \beta(c) &= 2 \\ \beta(g) &= 3 \\ \beta(t) &= 4 \end{aligned}$$

Der Isomorphismus $\gamma : \{1, 2, 3, 4\}^* \to \mathbb{N}$ bildet diese Darstellung in die Menge der natürlichen Zahlen ab:

$$\gamma(\beta_{n-1} \dots \beta_0) = \sum_{i=0}^{n-1} 4^i \beta_i$$

Durch Verkettung der Isomorphismen wird jedem Codewort eine natürliche Zahl zugeordnet und umgekehrt. Diese Zahl kann als Position des Codewortes in der Aufzählung aufgefasst werden.

Die folgende Tabelle zeigt die ersten 10 Codewörter:

Position	Codewort
1	a
2	c
3	g
4	t
5	aa
6	ac
7	ag
8	at
9	ca
10	cc

Die so definierte pseudolexikographische Ordnung auf den Codewörtern wird benutzt, um eine Ordnung auf der Menge der Codes zu definieren. Hierzu wird ein Code als gemäß der Relation $<$, wie sie für Codewörter definiert ist, geordnete Menge aufgefasst. Diese Menge wird mit dem größten Codewort beginnend absteigend sortiert.

Die Menge der Codes kann so als Baum dargestellt werden. An der Wurzel des Baumes beginnend, liefern die Knotenbezeichnungen eines jeden Pfades genau einen Code. Dabei entspricht jedem Knoten genau ein Codewort.
Den Wurzelknoten ausgenommen entspricht die Anzahl der Knoten entlang eines Pfades der Codelänge.
Der erste Knoten eines Pfades gibt das größte Codewort des jeweiligen Codes

an. Die Position des ersten Codewortes ist gleichzeitig die Länge, die der vom Wurzelknoten ausgehende und über das erste Codewort bis zu einem Blattknoten führende Pfad hat.
Daher ist die Tiefe des Baumes für ein gegebenes Startcodewort beschränkt. Die folgende Abbildung zeigt den Baum der ersten 31 Codes.

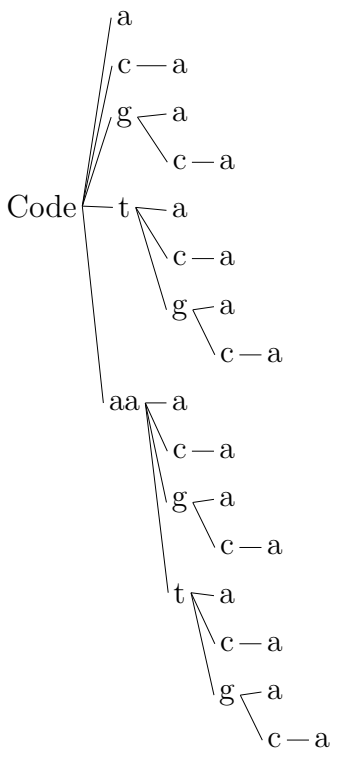

Fig. 3.1: Codebaum der ersten 31 Codes

Für unendliche Codes ist diese Art der graphischen Darstellung eher ungeignet, da es in diesem Fall kein Codewort gibt, welches größer als alle anderen ist und welches als erstes Codewort einen Pfad beginnt.

3.1 Testen von Codeeigenschaften

Um eine möglichst optimale Laufzeit beim Finden von Codes mit bestimmten Eigenschaften zu erhalten, müssen verschiedene Methoden der Codeaufzählung hinsichtlich ihrer Laufzeit bewertet werden.
Hier ergeben sich zunächst zwei Ansätze:

Der erste zählt alle Codes nacheinander auf und testet jeden generierten Code auf seine Eigenschaften. Dabei wird jedes mögliche Codewortpaar auf die Eigenschaften getestet. Bei einem Code der Länge n wären dies mindestens n^2 Tests pro Eigenschaft.

Bei einer rekursiven Aufzählung hingegen werden Codes schrittweise aus bereits positiv getesteten Codes generiert.

Die folgende Definition zeigt, wann ein Code aus einem anderen generierbar ist:

Definition 15. *Sei $C = \{w_1, \ldots, w_n\}, n \in \mathbb{N}$ ein Code mit $w_1 < w_2 < \cdots < w_n$.*
Jeder Code $C' = \{w'_1, \ldots, w'_m\}$ mit $m \geq n$ und $w'_1 < w'_2 < \cdots < w'_n$ heißt aus C generiert, falls $C' = C \cup C''$ und für alle $w'' \in C''$ gilt: $w'' < w_n$.

Satz 1. *Sei $C = \{w_1, \ldots, w_n\}$ ein Code mit $w_1 < w_2 < \cdots < w_n$.*
Sei E eine Menge von Eigenschaften.
Erfüllt C die Eigenschaften in E nicht, so erfüllen auch alle aus C generierbaren Codes E nicht.

Beweis. Unter der Vorraussetzung ein Code C erfülle eine Menge von Eigenschaften E nicht, sei angenommen der aus C generierbare Code C' erfülle die Eigenschaften. Dann erfüllen alle Teilcodes von C' diese Eigenschaften. Da nach Definition $C \subseteq C'$ würden die Eigenschaften auch in C erfüllt sein.
Dies ist jedoch ein Widerspruch zur Vorraussetzung. □

Dieser Zusammenhang ermöglicht ein effizienteres Testen der Eigenschaften.
Sei C ein aus C' generierter Code. Dann gibt es einen Code C'', so dass $C = C' \cup C''$. C' erfülle die Eigenschaften. Um zu ermitteln, ob auch C die

Eigenschaften erfüllt, genügt es alle neu hinzukommenden Codewortpaare zu testen. Jeder Rekursionsschritt ermittelt C'' so, dass $|C''| = 1$. Hat C die Länge n und C' somit die Länge $n-1$, so kommen $2(n-1)+1$ neue Codewortpaare hinzu. Für einen Code der Länge n müssen also höchstens $2(n-1)+1 = 2n-1$ Tests durchgeführt werden.

Das Hauptaugenmerk liegt nun auf der Optimierung der eigentlichen Eigenschaftstests. Die folgenden Betrachtungen sollen daher zwei Fragen beantworten:

- Bei welchen Tests ist am ehesten mit einem negativen Ergebnis zu rechnen, so dass die betreffenden Codes früher ausgeschlossen werden können?
- Bestehen Beziehungen zwischen den Eigenschaften, so dass Tests überflüssig werden?

3.1.1 Laufzeitbetrachtungen

In den folgenden Laufzeitbetrachtungen sei n die Länge eines beliebigen geordneten DNA-Codes C.

- θ-Überlappungsfreiheit

  Zum Testen der θ-Überlappungsfreiheit muss festgestellt werden, ob mindestens ein Codewort ein Komplement im Code besitzt.
  Die Wahrscheinlichkeit ein solches Codewort zu finden ist am größten, wenn alle Codewörter die gleiche Länge haben, da Codewörter mit unterschiedlicher Länge nicht komplementär zueinander sein können.
  Der worst case betrifft also Codes mit konstanter Codewortlänge. In diesem Fall muss jedes Codewort mit all seinen (kleineren) Vorgängern ($n-1$ Vergleiche) und mit sich selbst (1 Vergleich) verglichen werden. Dies ergibt n Vergleiche.

- θ-(p, s)-Konformität

  Ein Code heißt $\theta-(p, s)$-konform, falls es kein Codewort gibt, dessen Komplement Infix (Präfix, Suffix) in einem anderen ist.
  Da ein neu hinzugekommenes Codewort nur in seinen Vorgängern ein Infix (Präfix, Suffix) sein kann, müssen nur $n-1$ Paare getestet werden.

- θ-Kommafreiheit

  Ein Code heißt θ-kommafrei, falls das Komplement keines Codeworts echtes Infix in einer Konkatenation zweier Codewörter ist.
  Für ein neu hinzugekommenes Codewort muss getestet werden, ob sein Komplement Infix in einer der $(n-1)^2$ Konkatenationen von je zwei Codewörtern des generierenden Codes ist. Zusätzlich müssen $2(n-1)$ mögliche Konkatenationen des neuen Codeworts mit je einem bereits vorhandenem Codewort sowie die Konkatenation des neuen Codeworts

mit sich selbst berücksichtigt werden.
Dies ergibt $(n-1)^2 + 2(n-1) + 1 = n^2$ Vergleiche.

- θ-sticky-Freiheit

  Ein Code heißt θ-sticky-frei, wenn das Komplement keines echten Präfixes eines jeden Codewortes gleichzeitig echtes Suffix eines Codewortes ist.
  In diesem Fall müsste jede mögliche Paarung von Codewörtern getestet werden. Dies ergibt $2n - 1$ Vergleiche.

- θ-3'(5')-Überhangfreiheit

  Ein Code heißt θ-3'(5')-überhangfrei, wenn das Komplement keines echten Präfixes (Suffixes) eines jeden Codewortes gleichzeitig echtes Präfix (Suffix) eines anderen Codewortes ist.
  Hier genügt das Testen von $n - 1$ Paaren, bestehend aus dem neuen Codewort und jeweils einem der bereits vorhandenen Codewörter. Hinzu kommt ein weiterer Test des neuen Codeworts mit sich selbst. Dies sind n Tests.

- $\theta(k, m_1, m_2)$-Konformität

  Ein Code heißt $\theta(k, m_1, m_2)$-konform, falls kein Codewort zwei k Zeichen lange komplementäre Teilstücken besitzt, die zwischen m_1 und m_2 Zeichen voneinander entfernt sind. Dieser Test muss nur für das neu hinzukommende Codewort ausgeführt werden. Es ist also nur ein Test erforderlich.

- θ-k-Code

  Ein Code heißt θ-k-Code, falls kein k Zeichen langes Teilwort eines jeden Codeworts Komplement in einem anderen Codewort ist. Dies muss

für das neu hinzukommende Codewort und jeweils ein vorhandenes getestet werden. Dazu kommt der Test des neuen Codeworts mit sich selbst. Es ergeben sich n Tests.

Die folgende Tabelle liefert einen Überblick über die angestellten Betrachtungen:

Eigenschaft	Aufwand $O(n)$
θ-Überlappungsfreiheit	n
$\theta-(p,s)$-Konformität	$n-1$
θ-Kommafreiheit	n^2
θ-sticky-Freiheit	$2n-1$
θ-3'(5')-Überhangfreiheit	n
$\theta(k,m_1,m_2)$-Teilwortkonformität	1
θ-k-Code	n

Das Ergebnis dieser Untersuchung ist eindeutig. Offenbar ist es sinnvoll die Eigenschaft der θ-Konformität vor allen anderen zu testen während die θ-Kommafreiheit erst nach allen anderen Eigenschaften getestet werden sollte.

3.1.2 Beziehungen zwischen Eigenschaften

Zwischen vielen der genannten Eigenschaften bestehen Inklusionsbeziehungen, wie sie in Abbildung 3.2 wiedergegeben sind. Ein Pfeil weist dabei auf die jeweils eingeschlossene Eigenschaft.

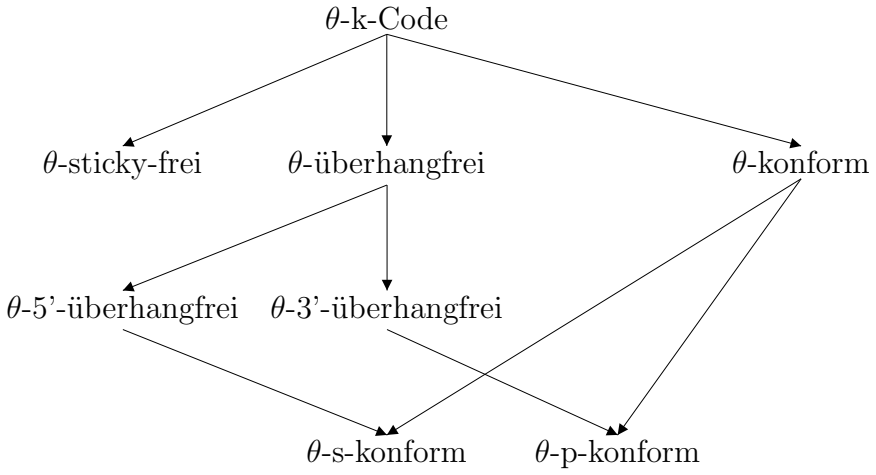

Fig. 3.2: Beziehungen zwischen Codeeigenschaften

Anhand dieser Beziehungen können Tests gespart werden, wenn sie durch den Test einer übergeordneten Eigenschaft behandelt werden.

Soll beispielsweise ein θ-überhangfreier und θ-p-konformer Code gefunden werden, muss die θ-p-Eigenschaft nicht nachgewiesen werden, da jeder θ-überhangfreie Code θ-3'-überhangfrei ist und letzterer stets θ-p-konform ist.

3.2 Aufzählungsprogramm

Das mit dieser Arbeit entstande Programm ist ein Werkzeug zum Aufzählen von DNA-Codes mit den beschriebenen Eigenschaften. Es findet Codes gemäß den in 3.1 durchgeführten Betrachtungen in polynomieller Zeit (worst case). Das Programm ist in der Sprache C++ geschrieben und wurde auf einer Unix- und Windows-Plattform getestet.

3.2.1 Arbeitsweise

Das zentrale Element des Aufzählungsprogramms ist eine rekursive Funktion, die den Codebaum in der Breite durchläuft. Der Vorgang beginnt in der Hilfsfunktion `StartRecursion()` (Listing 3.1). Diese Funktion zählt linear alle Codewörter, beginnend mit dem kleinsten, auf und untersucht die von ihnen generierten Teilbäume rekursiv.
Ist die Funktion beim generierenden Codewort w angelangt, so untersucht sie zunächst den von ihm generierten trivialen Code $\{w\}$ auf die gewünschten Eigenschaften (Zeile 1077). Sind sie nicht erfüllt untersucht `StartRecursion()` das nächste Generatorcodewort. (Zeile 1096)
Treffen alle gewünschten Eigenschaften zu, wird mittels der Funktion `Recurse()` der von w erzeugte Teilbaum untersucht. (Zeile 1090)
Letztere Funktion erhält beim Aufruf einen Parameter `tiefe`, der angibt, bis zu welcher Tiefe der Teilbaum untersucht werden soll.
`Recurse()` erzeugt nun schrittweise aus dem bestehenden Code Codes der angegebenen Tiefe und untersucht sie auf die Eigenschaften. Treffen diese bei einem Code nicht zu, endet die Rekursion für diesen Ast des Baumes. Alle aus diesem Code generierbaren Codes müssen nun nicht mehr betrachtet werden und der nächste Code derselben Tiefe wird betrachtet (Zeile 1058).
Treffen die Eigenschaften jedoch zu, so werden zunächst durch einen weiteren Aufruf der Funktion `Recurse()` mit einer um 1 größeren Tiefe alle weiteren aus dem Code generierbaren Codes untersucht (Zeile 1054). Anschließend wird mit dem nächsten Code der aktuellen Tiefe fortgefahren. (Zeile 1058)

Listing 3.1: CCode::StartRecursion(), CCode.cpp

```
1067 int CCode::StartRecursion(OnCodeFound onCodeFound)
1068 {
1069   int goOn = 1;
1070   codesFound = 0;
1071
1072
1073   while ((goOn >= 0) && (!restrictCodes ||
1074     ((codesFound<maxCodes) && (restrictCodes))))
1075   {
1076
1077     int ok = CheckProperties(this->word[1],this->word[1],false);
1078     if (ok == 1)
```

```
1079        {
1080          if ((((restrictCodeLength) && (count>=minCodeLength) &&
1081            (count<=maxCodeLength)) || (!restrictCodeLength)) &&
1082            ((infRate && (getInformationRate()>=((minInfRate<0)?this->
                  getMaxInfRate(getCount()):minInfRate))) || (!infRate)))
1083          {
1084            codesFound++;
1085            if (onCodeFound!=NULL)
1086              onCodeFound((void *)this,codesFound,0);
1087          }
1088          if (((restrictCodeLength) && (count<maxCodeLength) && (word[1].
                getPosition()>=minCodeLength)) ||
1089            (!restrictCodeLength))
1090            goOn = this->Recurse(2,onCodeFound);
1091        } else
1092          if (ok<0) {
1093            goOn = -1;
1094            break;
1095          }
1096      this->word[1].Next();
1097    }
1098    if (goOn<0)
1099    {
1100      return -1;
1101    }
1102    return 1;
1103  }
```

Listing 3.2: CCode::Recurse(int), CCode.cpp

```
970  int CCode::Recurse(int tiefe,OnCodeFound onCodeFound)
971  {
972    int a = 1;
973    bool b = true;
974
975    if (((restrictCodeLength) && (count>=maxCodeLength))) return 0;
976
977    CCodeWord* temp = new CCodeWord[ (tiefe+1) ];
978    // bereits gefundende Codewörter sichern
979    for (int k=1; k<tiefe; k++)
980    {
981      temp[k] = this->word[k];
982    }
983
984    // Codewort-Feld neu anlegen
985    delete[] this->word;
986    this->word = new CCodeWord[ (tiefe+1) ];
987
988    // gesicherte Codewörter in Codewort-Feld ablegen
989    for (int k=1; k<tiefe; k++)
```

```
990     {
991       this->word[k] = temp[k];
992     }
993     this->count = tiefe;
994
995     delete[] temp;
996     delete[] this->word[tiefe].letter;
997     // Wort der Tiefe tiefe initialisieren
998       this->word[tiefe].letter = new int[(1+1)];
999     this->word[tiefe].letter[1] = MIN_LETTER;
1000      this->word[tiefe].setPosition(1);
1001      this->word[tiefe].setLength(1);
1002
1003    for (int i=1; i<this->word[tiefe-1].getPosition(); i++)
1004    {
1005      bool hasMinLength = true;
1006      bool hasInfRate = true;
1007
1008
1009      if (infRate && restrictCodeLength)
1010      {
1011        int ir = CheckInfRate(tiefe,this->maxCodeLength);
1012        if (ir<0) break;
1013        if (ir==0)
1014          hasInfRate = false;
1015      }
1016
1017      if (restrictCodeLength)
1018      {
1019        if (word[tiefe].getPosition()+(tiefe-1)<this->minCodeLength)
1020        {
1021          hasMinLength = false;
1022        }
1023      }
1024
1025      // Codewörter auf Eigenschaften testen
1026      // zunächst neues Wort mit sich selbst testen
1027      if ((CheckProperties(this->word[tiefe],this->word[tiefe],true) == 1) &&
            (hasMinLength) && (hasInfRate))
1028      {
1029        // dann jedes Paar von neuem Codewort mit einem Vorgänger
1030        for ( int l=1; l<tiefe; l++)
1031        {
1032          a = CheckProperties(this->word[l],this->word[tiefe],false);
1033          if (a == 0)
1034          {
1035            b = false;
1036            break;
1037          } else
```

```
1038            if (a < 0)
1039              return -1;
1040        }
1041        if (b)
1042        {
1043          if ((((restrictCodeLength) && (count>=minCodeLength) &&
1044            (count<=maxCodeLength)) || (!restrictCodeLength)) &&
1045            ((infRate && (getInformationRate()>=((minInfRate<0)?this->
                  getMaxInfRate(getCount()):minInfRate))) || (!infRate)))
1046          {
1047            codesFound++;
1048            if (onCodeFound!=NULL)
1049              onCodeFound((void *)this,codesFound,0);
1050            if ((restrictCodes) && (codesFound>=maxCodes))
1051              return -1;
1052
1053          }
1054          int s = this->Recurse(tiefe+1,onCodeFound);
1055          if (s < 0) return s;
1056        }
1057      }
1058      this->word[tiefe].Next();
1059      a = 1;
1060      b = true;
1061    }
1062    // Nach der Rekursion sollte die count wieder um eins verringert werden.
1063    this->count = tiefe-1;
1064    return 1;
1065  }
```

3.2.2 Bedienung

Das Programm kann auf zwei kombinierbare Arten bedient werden. Zum einen kann der Anwendung eine Reihe von Parametern übergeben werden, die die Eigenschaften der zu suchenden Codes spezifizieren. Zum anderen kann das Programm in einem interaktiven Konsolenmodus gestartet werden, der eine bequeme Eingabe von Codeeigenschaften erlaubt. Im folgenden ist eine Liste der zulässigen Parameter und ihre Bedeutung aufgeführt.

Tab. 3.1: Parameterliste

Parameter	Bedeutung
-i	Startet das Programm im interaktiven Modus.
-l=minLength,maxLength	Sucht Codes deren Codewörter mindestens minLength und höchstens maxLength Zeichen lang sind. Ohne Angabe dieses Parameters ist die Codewortlänge nicht beschränkt.
-n=min,max	Sucht Codes mit mindestens min und höchstens max Codewörtern. Ohne Angabe dieses Parameters wird die Codelänge nicht eingeschränkt.
-count=n	Sucht höchstens n Codes. Warnung: ohne Angabe dieses Parameters terminiert das Programm unter Umständen nicht, da es ohne Beschränkung der Codelänge und/oder der Codewortlänge unendlich viele Codes mit den gewünschten Eigenschaften gibt.
-o Dateiname	Gibt die Liste gefundener Codes als Textdatei im CSV-Format aus.

-theta-non-overlapping, -strict	Sucht θ-überlappungsfreie Codes.
-theta-compliant	Sucht θ-konforme Codes.
-theta-prefix-compliant	Sucht θ-p-konforme Codes.
-theta-suffix-compliant	Sucht θ-s-konforme Codes.
-theta-sticky-free	Sucht θ-sticky-freie Codes.
-theta-3-prime-overhang-free	Sucht θ-3'-überhangfreie Codes.
-theta-5-prime-overhang-free	Sucht θ-5'-überhangfreie Codes.
-theta-overhang-free	Sucht θ-überhangfreie Codes. Schließt die Parameter -theta-3-prime-overhang-free und -theta-5-prime-overhang-free ein.
-theta-comma-free	Sucht θ-kommafreie Codes.
-theta-subword-compliant=k,m1,m2	Sucht $\theta(k, m_1, m_2)$-teilwortkonforme Codes.
-theta-code=k	Sucht θ-k-Codes.
-comma-free	Sucht kommafreie Codes.
-solid	Sucht solide Codes.
-rate=minRate	Sucht Codes mit einer relativen Informationsrate von mindestens minRate bits/Buchstabe.

Bei Start im interaktiven Modus (Abb. 3.2.2) wird eine Liste mit möglichen Eigenschaften angezeigt. Der Anwender kann eine Eigenschaft durch Druck auf die entsprechende Buchstabentaste ein- oder ausschalten.
Eingeschaltete Eigenschaften werden mit einem $\times$ markiert.
Wurden neben dem Parameter -i andere Codeeigenschaften betreffende Parameter übergeben, werden die dazugehörigen Eigenschaften als aktiv markiert.
Durch Druck auf 's' wird der Suchvorgang gestartet, mit 't' wird das Programm beendet.

```
(A) x theta-non-overlapping
(B)   theta-compliant
(C)   theta-p-compliant
(D)   theta-s-compliant
(E)   theta-free
(F)   theta-sticky-free
(G)   theta-overhang-free
(H)   theta-3'-overhang-free
(I)   theta-5'-overhang-free
(J)   theta(k, m1, m2)-subword-compliant
(K)   theta-k-code
(L) x strict
(M) x solid
(N) x comma-free
(O)   restrict code word length
(P)   restrict code length
(Q) x restrict number of codes to 200
(R)   limit information rate
(S)   Run
(T)   Quit
```

Fig. 3.3: Interaktiver Modus

4. UNTERSUCHUNGEN

4.1 Binäre DNA-Codes

Unerwünschte Hybridisierungen von DNA-Molekülen können das Ergebnis einer Berechnung verfälschen und unbrauchbar machen. Die Klasse der θ-1-Codes schließt derartige Hybridisierungsmöglichkeiten für alle in ihr enthaltenen Codes aus.
Ist C ein beliebiger DNA-Code und bezeichne P die θ-1-Eigenschaft, so gilt gemäß Definition 12:

$$P(C) \Leftrightarrow \mathit{Infix}_1(C) \cap \mathit{Infix}_1(\theta(C)) = \emptyset$$

Da die Menge der Teilwörter der Länge 1 eines Codes der Menge der Buchstaben entspricht, aus denen seine Codewörter aufgebaut sind, gilt auch

$$\mathit{Infix}_1(C) \subseteq \Delta \text{ und } \mathit{Infix}_1(\theta(C)) \subseteq \Delta.$$

Wie muß $\mathit{Infix}_1(C)$ aufgebaut sein, damit $\mathit{Infix}_1(C) \cap \mathit{Infix}_1(\theta(C)) = \emptyset$?
Angenommen $x \in \mathit{Infix}_1(C)$. Dann ist $\theta(x) \in \mathit{Infix}_1(\theta(C))$. Weiterhin sei $y = \theta(x)$. Ist $y \in \mathit{Infix}_1(C)$, dann ist $\theta(y) \in \mathit{Infix}_1(\theta(C))$. Da aber $\theta(y) = \theta(\theta(x)) = x$, ist $x \in \mathit{Infix}_1(\theta(C))$. Somit ist aber $\mathit{Infix}_1(C) \cap \mathit{Infix}_1(\theta(C)) \neq \emptyset$, was der Bedingung für die θ-1-Eigenschaft widerspricht.
Folglich darf ein Code, welcher ein θ-1-Code ist, nur aus Codewörtern bestehen, deren Buchstaben nicht komplementär zueinander sind. Da es jedoch nur je zwei nicht komplementäre Buchstaben im DNA-Alphabet gibt, sind derartige Codes binär. Ein DNA-Code mit dieser Eigenschaft heißt daher binärer DNA-Code. Das Codealphabet ist hier auf nur zwei nicht-komplementäre Zeichen eingeschränkt:

Definition 16. *Sei* $\Delta' = \{x, y\}$ *ein Alphabet mit* $x \in \Delta, y \in \Delta - \{x, \theta(x)\}, x < y$.
Ein DNA-Code C *heißt binärer DNA-Code, falls* $C \subseteq \Delta'^*$.

Da dies Codes über einem binären Alphabet sind, liegt die Vermutung nahe, dass sie auf herkömmliche binäre Codes über dem binären Alphabet $\mathbb{B} = \{0, 1\}$ abgebildet werden können.
In der Tat kann dies durch eine geeignete injektive Abbildung $\delta : \Delta' \to \mathbb{B}$ mit $\mathbb{B}$ bewirkt werden. Für diese Abbildung gilt:

$$\delta(x) = \begin{cases} 0, \text{ falls } x = \min \Delta' \\ 1 \text{ sonst} \end{cases}$$

Durch Erweitern von δ auf Wörter über Δ' können ganze Codewörter auf binäre Codewörter über $\mathbb{B}$ abgebildet werden. Jeder θ-1-Code kann somit auf genau einen binären Code abgebildet werden.
Die Abbildung ist somit injektiv.
Allerdings kann ein binärer Code auf vier verschiedene θ-1-DNA-Codes abgebildet werden, da es genau vier Möglichkeiten gibt, zwei nicht-komplementäre Basen in einer Menge Δ' zusammenzufassen. Ein Beispiel:

$C_1 = \{acacaa, acca, aac\}, \delta(a) = 0, \delta(c) = 1$
Der C_1 entsprechende binäre Code ist $C_{1,b} = \{010100, 0110, 001\}$.
Der folgenden Tabelle können die vier möglichen inversen Abbildungen von $C_{1,b}$ entnommen werden:

$\delta^{-1}(0)$	$\delta^{-1}(1)$	Code
a	c	$\{acacaa, acca, aac\}$
a	g	$\{agagaa, agga, aag\}$
c	t	$\{ctctcc, cttc, cct\}$
g	t	$\{gtgtgg, gttg, ggt\}$

Die Abbildung ist somit nicht surjektiv bzw. bijektiv.

Da binäre DNA-Codes auf konventionelle binäre Codes abbildbar sind, unterliegen sie auch den selben Eigenschaften und Gesetzmäßigkeiten wie letztere.

4.2 Optimale Codes

Eine weitere Untersuchung im Rahmen dieser Arbeit zielte auf das Finden von optimalen Codes bezüglich ihrer Informationsrate.

Für die relative Informationsrate $r(n)$ einen n-stelligen Codes über einem vierbuchstabigen Alphabet gilt bekanntlich:

$$r(n) = \frac{n \log_4 n}{\sum_{w \in C} |w|}$$

Da diese Größe von der Anzahl der Codewörter und der durchschnittlichen Codewortlänge beeinflusst wird, werden hier Codes mit fester Länge n betrachtet.
Die Frage lautet daher, bei welcher durchschnittlichen Codewortlänge die relative Informationsrate einen maximalen Wert annimmt.
Da n als fest angenommen wird, hängt $r(n)$ nun mehr maßgeblich von der Summe aller Codewortlängen ab. Diese Summe wird im Folgenden mit $s(n) = \sum_{w \in C} |w|$ bezeichnet.

Da ein Quotient genau dann maximal ist, wenn sein Divisor bei festem Dividenden minimal ist, ist $r(n)$ genau dann maximal, wenn $s(n)$ minimal ist.
Die nächste Frage lautet daher: wie kann $s(n)$ minimiert werden?
Eine beliebige Summe über einer Menge von Summanden ist minimal, wenn die jeweils kleinsten Summanden der Menge Teil dieser Summe sind.
Um also die kleinstmögliche Summe von Codewortlängen für eine gegebene Codelänge zu erhalten, muss der betreffende Code aus den kürzestmöglichen Codewörtern bestehen.

Die kürzesten Codewörter haben die Länge 1. Daher sind in jedem optimalen Code der Länge $n < 4$ genau n Codewörter der Länge 1 enthalten. Für $n \geq 4$ sind dies genau vier Codewörter, da es nur vier verschiedene Codewörter der Länge 1 gibt. Für $n > 4$ werden entsprechend Codewörter der Länge 2 hinzugefügt, bis alle möglichen $4^2 = 16$ Codewörter der Länge 2 verbraucht wurden. Dann kommen Codewörter der Länge 3 an die Reihe usw.

Definition 17. *Ein Code $C = C_1 \cup C_2 \cup \cdots \cup C_n$ heißt optimal bezüglich seiner Codewortlängen genau dann, wenn $\forall i : |C_i| = 4^i$ und $\forall w \in C_i : |w| = i$*

Die untenstehende Tabelle zeigt eine Übersicht über die Längen der Codewörter für die beschriebenen Codes.

n	$s(n) = \sum_{w \in C} \lvert w \rvert$	
1	$1 \cdot 1$	$= 1$
2	$2 \cdot 1$	$= 2$
3	$3 \cdot 1$	$= 3$
4	$4 \cdot 1$	$= 4$
5	$4 \cdot 1 + 1 \cdot 2$	$= 6$
6	$4 \cdot 1 + 2 \cdot 2$	$= 8$
7	$4 \cdot 1 + 3 \cdot 2$	$= 10$
$\vdots$	$\vdots$	
20	$4 \cdot 1 + 16 \cdot 2$	$= 36$
21	$4 \cdot 1 + 16 \cdot 2 + 1 \cdot 3$	$= 39$
22	$4 \cdot 1 + 16 \cdot 2 + 2 \cdot 3$	$= 42$
$\vdots$	$\vdots$	

Offenbar wird die Summe der Codewortlängen $s(n)$ gerade durch die endliche Reihe $b_k = \sum_{i=1}^{k} i \cdot 4^i$ und einem Restterm $c(n,k)$ der Form $c(n,k) = (n - n'_k) \cdot (k+1)$, wobei $n'_k = \sum_{i=1}^{k} 4^i$ beschrieben.
Somit lautet die allgemeine Formel zur Berechnung der minimalen Codewortlänge für ein gegebenes n zunächst

$$s(n,k) = \underbrace{\sum_{i=1}^{k} i \cdot 4^i}_{b_k} + \underbrace{(n - \underbrace{\sum_{i=1}^{k} 4^i}_{n'_k}) \cdot (k+1)}_{c(n,k)}$$

Da dieser Term von einer weiteren Variablen k abhängt, soll diese nun eliminiert werden.
Die Reihe b_k kann auch explizit notiert werden:

$$\begin{aligned}
b_k &= 4^1 + 2 \cdot 4^2 + \cdots + k \cdot 4^k \\
4b_k &= 4^2 + 2 \cdot 4^3 + \cdots + n \cdot 4^{n+1} \\
b_k - 4b_k &= 4 + 4^2 + 4^3 + \cdots + 4^n - n \cdot 4^{n+1} \\
-3b_k &= \frac{4}{3}(4^k - 1) - k \cdot 4^{k+1} \\
b_k &= \frac{1}{3}k \cdot 4^{k+1} - \frac{4}{9}(4^k - 1)
\end{aligned}$$

(Beweis durch vollständige Induktion)
Ebenso kann die geometrische Reihe $n'_k = \sum_{i=1}^{k} 4^i$ in der expliziten Form

$$n'_k = \tfrac{4}{3}(4^k - 1)$$

formuliert werden.
Desweiteren gilt für jedes $n \geq n'_k$:

$$\tfrac{n}{n'_k} \geq 1$$

Durch Umstellen dieser Ungleichung nach einem natürlichen k, kann die Länge desjenigen Codes festgestellt werden, der kürzer als n ist aber dessen Codewortlängensumme um den Term $c(k, n)$ vermindert ist.
Zusammenfassend gilt also:

$$\forall C, |C| = n : \exists C', |C| = n' \leq n : s(n, k) = b_k + c(k, n), s(n', k) = b_k$$

Mit $\frac{n}{n'_k} \geq 1$ ergibt sich k wie folgt:

$$\begin{aligned}
\frac{n}{\frac{4}{3}(4^k - 1)} &\geq 1 \\
\frac{3n}{4(4^k - 1)} &\geq 1 \\
\frac{3}{4}n &\geq 4^k - 1 \\
\frac{3}{4}n + 1 &\geq 4^k \\
k &\leq \log_4(\frac{3}{4}n + 1)
\end{aligned}$$

Da $k \in \mathbb{N}$ ist

$$k \leq \left\lfloor \log_4(\frac{3}{4}n + 1) \right\rfloor$$

Somit kann die Abhängigkeit von k in der Ausgangsgleichung entfernt werden und die Summe der Codewortlängen eines beliebigen Codes der Länge n beträgt

$$\begin{aligned} s(n) &= b_k + c(k) \\ &= \frac{1}{3}k \cdot 4^{k+1} - \frac{4}{9}(4^k - 1) + (n - n'_k) \cdot (k+1) \\ &= \frac{1}{3}k \cdot 4^{k+1} - \frac{4}{9}(4^k - 1) + [n - \frac{4}{3}(4^k - 1)] \cdot (k+1) \\ &= -\frac{16}{9}4^k + \frac{16}{9} + nk + n + \frac{4}{3}k \end{aligned}$$

mit

$$k = \left\lfloor \log_4(\tfrac{3}{4}n + 1) \right\rfloor$$

Durch Einsetzen dieses Terms in die Formel der relativen Informationsrate ergibt sich:

$$r(n) = \frac{9n \cdot \ln n}{2 \cdot \ln 2\,(-16 \cdot 4^k + 16 + 9kn + 12k + 9n)}$$

(k wie oben)

ANHANG

A. CODE-KATALOGE

A.1 Codes mit maximaler relativer Informationsrate

Die folgenden Tabellen zeigen optimale Codes mit Längen von 10 bis 20 Codewörtern.
Es sind jeweils höchstens 10 Codes je Tabelle aufgeführt.

n bezeichnet die Länge der Codes, r gibt die relative Informationsrate an.

Nr.	Code
1	cc, ca, at, ag, ac, ac, aa, t, g, c, a
2	cg, ca, at, ag, ac, ac, aa, t, g, c, a
3	cg, cc, ac, at, ag, ac, ac, aa, t, g, c, a
4	cg, cc, ca, ag, ac, ac, aa, t, g, c, a
5	cg, cc, ca, at, ac, ac ,aa, t, g, c, a
6	cg, cc, ca, at, ag, aa, t, g, c, a
7	cg, cc, ca, at, ag, ac, t, g, c, a
8	ct, ca, at, ag, ac, ac, aa, t, g, c, a
9	ct, cc, ac ,at, ag, ac, ac ,aa, t, g, c, a
10	ct, cc, ca, ag, ac, ac, aa, t, g, c, a

Tab. A.1: $n = 10$, $r = 1,038$ Bits/Buchstabe
(Anzahl gesamt: 8008)

Nr.	Code
1	cg, cc, ca, at, ag, ac, aa, t, g, c, a
2	ct, cc, ca, at, ag, ac, aa, t, g, c, a
3	ct, cg, ca, at, ag, ac, aa, t, g, c, a
Fortsetzung auf nächster Seite	

Nr.	Code
4	ct, cg, cc, at, ag, ac, aa, t, g, c, a
5	ct, cg, cc, ca, ag, ac, aa, t, g, c, a
6	ct, cg, cc, ca, at, ac, aa, t, g, c, a
7	ct, cg, cc, ca, at, ag, aa, t, g, c, a
8	ct, cg, cc, ca, at, ag, ac, t, g, c, a
9	ga, cc, ca, at, ag, ac, aa, t, g, c, a
10	ga, cg, ca, at, ag, ac, aa, t, g, c, a

Tab. A.2: $n = 11$, $r = 1,057$ Bits/Buchstabe
(Anzahl gesamt: 11440)

Nr.	Code
1	ct, cg, cc, ca, at, ag, ac, aa, t, g, c, a
2	ga, cg, cc, ca, at, ag, ac, aa, t, g, c, a
3	ga, ct, cc, ca, at, ag, ac, aa, t, g, c, a
4	ga, ct, cg, ca, at, ag, ac, aa, t, g, c, a
5	ga, ct, cg, cc, at, ag, ac, aa, t, g, c, a
6	ga, ct, cg, cc, ca, ag, ac, aa, t, g, c, a
7	ga, ct, cg, cc, ca, at, ac, aa, t, g, c, a
8	ga, ct, cg, cc, ca, at, ag, aa, t, g, c, a
9	ga, ct, cg, cc, ca, at, ag, ac, t, g, c, a
10	gc, cg, cc, ca, at, ag, ac, aa, t, g, c, a

Tab. A.3: $n = 12$, $r = 1,0755$ Bits/Buchstabe
(Anzahl gesamt: 12870)

Nr.	Code
1	ga, ct, cg, cc, ca, at, ag, ac, aa, t, g, c, a
2	gc, ct, cg, cc, ca, at, ag, ac, aa, t, g, c, a
3	gc, ga, cg, cc, ca, at, ag, ac, aa, t, g, c, a
Fortsetzung auf nächster Seite	

Nr.	Code
4	gc, ga, ct, cc, ca, at, ag, ac, aa, t, g, c, a
5	gc, ga, ct, cg, ca, at, ag, ac, aa, t, g, c, a
6	gc, ga, ct, cg, cc, at, ag, ac, aa, t, g, c, a
7	gc, ga, ct, cg, cc, ca, ag, ac, aa, t, g, c, a
8	gc, ga, ct, cg, cc, ca, at, ac, aa, t, g, c, a
9	gc, ga, ct, cg, cc, ca, at, ag, aa, t, g, c, a
10	gc, ga, ct, cg, cc, ca, at, ag, ac, t, g, c, a

Tab. A.4: $n = 13$, $r = 1,0933$ Bits/Buchstabe
(Anzahl gesamt: 11440)

Nr.	Code
1	gc, ga, ct, cg, cc, ca, at, ag, ac, aa, t, g, c, a
2	gg, ga, ct, cg, cc, ca, at, ag, ac, aa, t, g, c, a
3	gg, gc, ct, cg, cc, ca, at, ag, ac, aa, t, g, c, a
4	gg, gc, ga, cg, cc, ca, at, ag, ac, aa, t, g, c, a
5	gg, gc, ga, ct, cc, ca, at, ag, ac, aa, t, g, c, a
6	gg, gc, ga, ct, cg, ca, at, ag, ac, aa, t, g, c, a
7	gg, gc, ga, ct, cg, cc, at, ag, ac, aa, t, g, c, a
8	gg, gc, ga, ct, cg, cc, ca, ag, ac, aa, t, g, c, a
9	gg, gc, ga, ct, cg, cc, ca, at, ac, aa, t, g, c, a
10	gg, gc, ga, ct, cg, cc, ca, at, ag, aa, t, g, c, a

Tab. A.5: $n = 14$, $r = 1,1105$ Bits/Buchstabe
(Anzahl gesamt: 8008)

Nr.	Code
1	gg, gc, ga, ct, cg, cc, ca, at, ag, ac, aa, t, g, c, a
2	gt, gc, ga, ct, cg, cc, ca, at, ag, ac, aa, t, g, c, a
3	gt, gg, ga, ct, cg, cc, ca, at, ag, ac, aa, t, g, c, a
Fortsetzung auf nächster Seite	

Nr.	Code
4	gt, gg, gc, ct, cg, cc, ca, at, ag, ac, aa, t, g, c, a
5	gt, gg, gc, ga, cg, cc, ca, at, ag, ac, aa, t, g, c, a
6	gt, gg, gc, ga, ct, cc, ca, at, ag, ac, aa, t, g, c, a
7	gt, gg, gc, ga, ct, cg, ca, at, ag, ac, aa, t, g, c, a
8	gt, gg, gc, ga, ct, cg, cc, at, ag, ac, aa, t, g, c, a
9	gt, gg, gc, ga, ct, cg, cc, ca, ag, ac, aa, t, g, c, a
10	gt, gg, gc, ga, ct, cg, cc, ca, at, ac, aa, t, g, c, a

Tab. A.6: $n = 15$, $r = 1,127$ Bits/Buchstabe
(Anzahl gesamt: 4368)

Nr.	Code
1	gt, gg, gc, ga, ct, cg, cc, ca, at, ag, ac, aa, t, g, c, a
2	ta, gg, gc, ga, ct, cg, cc, ca, at, ag, ac, aa, t, g, c, a
3	ta, gt, gc, ga, ct, cg, cc, ca, at, ag, ac, aa, t, g, c, a
4	ta, gt, gg, ga, ct, cg, cc, ca, at, ag, ac, aa, t, g, c, a
5	ta, gt, gg, gc, ct, cg, cc, ca, at, ag, ac, aa, t, g, c, a
6	ta, gt, gg, gc, ga, cg, cc, ca, at, ag, ac, aa, t, g, c, a
7	ta, gt, gg, gc, ga, ct, cc, ca, at, ag, ac, aa, t, g, c, a
8	ta, gt, gg, gc, ga, ct, cg, ca, at, ag, ac, aa, t, g, c, a
9	ta, gt, gg, gc, ga, ct, cg, cc, at, ag, ac, aa, t, g, c, a
10	ta, gt, gg, gc, ga, ct, cg, cc, ca, ag, ac, aa, t, g, c, a

Tab. A.7: $n = 16$, $r = 1,1429$ Bits/Buchstabe
(Anzahl gesamt: 1820)

Nr.	Code
1	ta, gt, gg, gc, ga, ct, cg, cc, ca, at, ag, ac, aa, t, g, c, a
2	tc, gt, gg, gc, ga, ct, cg, cc, ca, at, ag, ac, aa, t, g, c, a
3	tc, ta, gg, gc, ga, ct, cg, cc, ca, at, ag, ac, aa, t, g, c, a
Fortsetzung auf nächster Seite	

Nr.	Code
4	tc, ta, gt, gc, ga, ct, cg, cc, ca, at, ag, ac, aa, t, g, c, a
5	tc, ta, gt, gg, ga, ct, cg, cc, ca, at, ag, ac, aa, t, g, c, a
6	tc, ta, gt, gg, gc, ct, cg, cc, ca, at, ag, ac, aa, t, g, c, a
7	tc, ta, gt, gg, gc, ga, cg, cc, ca, at, ag, ac, aa, t, g, c, a
8	tc, ta, gt, gg, gc, ga, ct, cc, ca, at, ag, ac, aa, t, g, c, a
9	tc, ta, gt, gg, gc, ga, ct, cg, ca, at, ag, ac, aa, t, g, c, a
10	tc, ta, gt, gg, gc, ga, ct, cg, cc, at, ag, ac, aa, t, g, c, a

Tab. A.8: $n = 17$, $r = 1,158$ Bits/Buchstabe
(Anzahl gesamt: 560)

Nr.	Code
1	tc, ta, gt, gg, gc, ga, ct, cg, cc, ca, at, ag, ac, aa, t, g, c, a
2	tg, ta, gt, gg, gc, ga, ct, cg, cc, ca, at, ag, ac, aa, t, g, c, a
3	tg, tc, gt, gg, gc, ga, ct, cg, cc, ca, at, ag, ac, aa, t, g, c, a
4	tg, tc, ta, gg, gc, ga, ct, cg, cc, ca, at, ag, ac, aa, t, g, c, a
5	tg, tc, ta, gt, gc, ga, ct, cg, cc, ca, at, ag, ac, aa, t, g, c, a
6	tg, tc, ta, gt, gg, ga, ct, cg, cc, ca, at, ag, ac, aa, t, g, c, a
7	tg, tc, ta, gt, gg, gc, ct, cg, cc, ca, at, ag, ac, aa, t, g, c, a
8	tg, tc, ta, gt, gg, gc, ga, cg, cc, ca, at, ag, ac, aa, t, g, c, a
9	tg, tc, ta, gt, gg, gc, ga, ct, cc, ca, at, ag, ac, aa, t, g, c, a
10	tg, tc, ta, gt, gg, gc, ga, ct, cg, ca, at, ag, ac, aa, t, g, c, a

Tab. A.9: $n = 18$, $r = 1,173$ Bits/Buchstabe
(Anzahl gesamt: 120)

Nr.	Code
1	tg, tc, ta, gt, gg, gc, ga, ct, cg, cc, ca, at, ag, ac, aa, t, g, c, a
2	tt, tc, ta, gt, gg, gc, ga, ct, cg, cc, ca, at, ag, ac, aa, t, g, c, a
3	tt, tg, ta, gt, gg, gc, ga, ct, cg, cc, ca, at, ag, ac, aa, t, g, c, a
Fortsetzung auf nächster Seite	

Nr.	Code
4	tt, tg, tc, gt, gg, gc, ga, ct, cg, cc, ca, at, ag, ac, aa, t, g, c, a
5	tt, tg, tc, ta, gg, gc, ga, ct, cg, cc, ca, at, ag, ac, aa, t, g, c, a
6	tt, tg, tc, ta, gt, gc, ga, ct, cg, cc, ca, at, ag, ac, aa, t, g, c, a
7	tt, tg, tc, ta, gt, gg, ga, ct, cg, cc, ca, at, ag, ac, aa, t, g, c, a
8	tt, tg, tc, ta, gt, gg, gc, ct, cg, cc, ca, at, ag, ac, aa, t, g, c, a
9	tt, tg, tc, ta, gt, gg, gc, ga, cg, cc, ca, at, ag, ac, aa, t, g, c, a
10	tt, tg, tc, ta, gt, gg, gc, ga, ct, cc, ca, at, ag, ac, aa, t, g, c, a

Tab. A.10: $n = 19$, $r = 1,1896$ Bits/Buchstabe
(Anzahl gesamt: 16)

Nr.	Code
1	tt, tg, tc, ta, gt, gg, gc, ga, ct, cg, cc, ca, at, ag, ac, aa, t, g, c, a

Tab. A.11: $n = 20$, $r = 1,201$ Bits/Buchstabe
(Anzahl gesamt: 1)

A.2 Solide Codes

Nr.	r	Code
1	$0,4966$	acgt acgg acgc acct accg accc acat acag aatt aatg aatc aagt aagg aagc act
2	$0,4884$	acgt acgg acgc acct accg accc acat acag aatt aatg aatc aagt aagg aagc aact
3	$0,4884$	actc acgt acgg acgc acct accg accc acat acag aatt aatg aatc aagt aagg aagc
4	$0,4884$	actg acgt acgg acgc acct accg accc acat acag aatt aatg aatc aagt aagg aagc

Fortsetzung auf nächster Seite

Nr.	r	Code
5	$0,4884$	actg actc acgg acgc acct accg accc acat acag aatt aatg aatc aagt aagg aagc
6	$0,4884$	actg actc acgt acgc acct accg accc acat acag aatt aatg aatc aagt aagg aagc
7	$0,4884$	actg actc acgt acgg acct accg accc acat acag aatt aatg aatc aagt aagg aagc
8	$0,4884$	actg actc acgt acgg acgc accg accc acat acag aatt aatg aatc aagt aagg aagc
9	$0,4884$	actg actc acgt acgg acgc acct accc acat acag aatt aatg aatc aagt aagg aagc
10	$0,4884$	actg actc acgt acgg acgc acct accg acat acag aatt aatg aatc aagt aagg aagc
11	$0,5141$	actg actc acgt acgg acgc acct accg accc acag aagt aagg aagc att atg atc
12	$0,5052$	actg actc acgt acgg acgc acct accg accc acag aatc aagt aagg aagc att atg
13	$0,5052$	actg actc acgt acgg acgc acct accg accc acag aatg aagt aagg aagc att atc
14	$0,4966$	actg actc acgt acgg acgc acct accg accc acag aatg aatc aagt aagg aagc att
15	$0,5052$	actg actc acgt acgg acgc acct accg accc acag aatt aagt aagg aagc atg atc
16	$0,4966$	actg actc acgt acgg acgc acct accg accc acag aatt aatc aagt aagg aagc atg
17	$0,4966$	actg actc acgt acgg acgc acct accg accc acag aatt aatg aagt aagg aagc atc
18	$0,4884$	actg actc acgt acgg acgc acct accg accc acag aatt aatg aatc aagt aagg aagc
19	$0,5141$	actg actc acgt acgg acgc acct accg accc acat aatt aatg aatc agt agg agc

Fortsetzung auf nächster Seite

Nr.	r	Code
20	$0,5052$	actg actc acgt acgg acgc acct accg accc acat aatt aatg aatc aagc agt agg

Tab. A.12: Solide Codes, n=15

Nr.	r	Code
1	0.5	actg actc acgt acgg acgc acct accg accc acat acag aatt aatg aatc aagt aagg aagc
2	$0,5$	actt actc acgt acgg acgc acct accg accc acat acag aatt aatg aatc aagt aagg aagc
3	$0,5$	actt actg acgt acgg acgc acct accg accc acat acag aatt aatg aatc aagt aagg aagc
4	$0,5$	actt actg actc acgg acgc acct accg accc acat acag aatt aatg aatc aagt aagg aagc
5	$0,5$	actt actg actc acgt acgc acct accg accc acat acag aatt aatg aatc aagt aagg aagc
6	$0,5$	actt actg actc acgt acgg acct accg accc acat acag aatt aatg aatc aagt aagg aagc
7	$0,5$	actt actg actc acgt acgg acgc accg accc acat acag aatt aatg aatc aagt aagg aagc
8	$0,5$	actt actg actc acgt acgg acgc acct accc acat acag aatt aatg aatc aagt aagg aagc
9	$0,5$	actt actg actc acgt acgg acgc acct accg acat acag aatt aatg aatc aagt aagg aagc
10	$0,5246$	actt actg actc acgt acgg acgc acct accg accc acag aagt aagg aagc att atg atc
11	$0,5161$	actt actg actc acgt acgg acgc acct accg accc acag aatc aagt aagg aagc att atg
12	$0,5161$	actt actg actc acgt acgg acgc acct accg accc acag aatg aagt aagg aagc att atc

Fortsetzung auf nächster Seite

Nr.	r	Code
13	$0,5079$	actt actg actc acgt acgg acgc acct accg accc acag aatg aatc aagt aagg aagc att
14	$0,5161$	actt actg actc acgt acgg acgc acct accg accc acag aatt aagt aagg aagc atg atc
15	$0,5079$	actt actg actc acgt acgg acgc acct accg accc acag aatt aatc aagt aagg aagc atg
16	$0,5079$	actt actg actc acgt acgg acgc acct accg accc acag aatt aatg aagt aagg aagc atc
17	$0,5$	actt actg actc acgt acgg acgc acct accg accc acag aatt aatg aatc aagt aagg aagc
18	$0,5246$	actt actg actc acgt acgg acgc acct accg accc acat aatt aatg aatc agt agg agc
19	$0,5161$	actt actg actc acgt acgg acgc acct accg accc acat aatt aatg aatc aagc agt agg
20	$0,5161$	actt actg actc acgt acgg acgc acct accg accc acat aatt aatg aatc aagg agt agc

Tab. A.13: Solide Codes, n=16

Nr.	r	Code
1	$0,5109$	actt actg actc acgt acgg acgc acct accg accc acat acag aatt aatg aatc aagt aagg aagc
2	$0,5345$	agat actt actg actc acgt acgg acgc acct accg accc acat aatt aatg aatc agt agg agc
3	$0,5264$	agat actt actg actc acgt acgg acgc acct accg accc acat aatt aatg aatc aagc agt agg
4	$0,5264$	agat actt actg actc acgt acgg acgc acct accg accc acat aatt aatg aatc aagg agt agc
5	$0,5186$	agat actt actg actc acgt acgg acgc acct accg accc acat aatt aatg aatc aagg aagc agt

Fortsetzung auf nächster Seite

Nr.	r	Code
6	$0,5264$	agat actt actg actc acgt acgg acgc acct accg accc acat aatt aatg aatc aagt agg agc
7	$0,5186$	agat actt actg actc acgt acgg acgc acct accg accc acat aatt aatg aatc aagt aagc agg
8	$0,5186$	agat actt actg actc acgt acgg acgc acct accg accc acat aatt aatg aatc aagt aagg agc
9	$0,5109$	agat actt actg actc acgt acgg acgc acct accg accc acat aatt aatg aatc aagt aagg aagc
10	$0,5264$	agcc agat actt actg actc acgt acgg acgc acct accg accc acat aatt aatg aatc agt agg
11	$0,5186$	agcc agat actt actg actc acgt acgg acgc acct accg accc acat aatt aatg aatc aagg agt
12	$0,5186$	agcc agat actt actg actc acgt acgg acgc acct accg accc acat aatt aatg aatc aagt agg
13	$0,5109$	agcc agat actt actg actc acgt acgg acgc acct accg accc acat aatt aatg aatc aagt aagg
14	$0,5264$	agcg agat actt actg actc acgt acgg acgc acct accg accc acat aatt aatg aatc agt agg
15	$0,5186$	agcg agat actt actg actc acgt acgg acgc acct accg accc acat aatt aatg aatc aagg agt
16	$0,5186$	agcg agat actt actg actc acgt acgg acgc acct accg accc acat aatt aatg aatc aagt agg
17	$0,5109$	agcg agat actt actg actc acgt acgg acgc acct accg accc acat aatt aatg aatc aagt aagg
18	$0,5264$	agcg agcc actt actg actc acgt acgg acgc acct accg accc acat aatt aatg aatc agt agg
19	$0,5186$	agcg agcc actt actg actc acgt acgg acgc acct accg accc acat aatt aatg aatc aagg agt
20	$0,5186$	agcg agcc actt actg actc acgt acgg acgc acct accg accc acat aatt aatg aatc aagt agg

Fortsetzung auf nächster Seite

Nr.	r	Code

Tab. A.14: Solide Codes, n=17

Nr.	r	Code
1	$0,536133$	agcg agcc agat actt actg actc acgt acgg acgc acct accg accc acat aatt aatg aatc agt agg
2	$0,528582$	agcg agcc agat actt actg actc acgt acgg acgc acct accg accc acat aatt aatg aatc aagg agt
3	$0,528582$	agcg agcc agat actt actg actc acgt acgg acgc acct accg accc acat aatt aatg aatc aagt agg
4	$0,521241$	agcg agcc agat actt actg actc acgt acgg acgc acct accg accc acat aatt aatg aatc aagt aagg
5	$0,536133$	agct agcc agat actt actg actc acgt acgg acgc acct accg accc acat aatt aatg aatc agt agg
6	$0,528582$	agct agcc agat actt actg actc acgt acgg acgc acct accg accc acat aatt aatg aatc aagg agt
7	$0,528582$	agct agcc agat actt actg actc acgt acgg acgc acct accg accc acat aatt aatg aatc aagt agg
8	$0,521241$	agct agcc agat actt actg actc acgt acgg acgc acct accg accc acat aatt aatg aatc aagt aagg
9	$0,536133$	agct agcg agat actt actg actc acgt acgg acgc acct accg accc acat aatt aatg aatc agt agg
10	$0,528582$	agct agcg agat actt actg actc acgt acgg acgc acct accg accc acat aatt aatg aatc aagg agt
11	$0,528582$	agct agcg agat actt actg actc acgt acgg acgc acct accg accc acat aatt aatg aatc aagt agg
12	$0,521241$	agct agcg agat actt actg actc acgt acgg acgc acct accg accc acat aatt aatg aatc aagt aagg
13	$0,536133$	agct agcg agcc actt actg actc acgt acgg acgc acct accg accc acat aatt aatg aatc agt agg

Fortsetzung auf nächster Seite

Nr.	r	Code
14	$0,528582$	agct agcg agcc actt actg actc acgt acgg acgc acct accg accc acat aatt aatg aatc aagg agt
15	$0,528582$	agct agcg agcc actt actg actc acgt acgg acgc acct accg accc acat aatt aatg aatc aagt agg
16	$0,521241$	agct agcg agcc actt actg actc acgt acgg acgc acct accg accc acat aatt aatg aatc aagt aagg
17	$0,536133$	agct agcg agcc agat actg actc acgt acgg acgc acct accg accc acat aatt aatg aatc agt agg
18	$0,528582$	agct agcg agcc agat actg actc acgt acgg acgc acct accg accc acat aatt aatg aatc aagg agt
19	$0,528582$	agct agcg agcc agat actg actc acgt acgg acgc acct accg accc acat aatt aatg aatc aagt agg
20	$0,521241$	agct agcg agcc agat actg actc acgt acgg acgc acct accg accc acat aatt aatg aatc aagt aagg

Tab. A.15: Solide Codes, n=18

Nr.	r	Code
1	$0,545342$	agct agcg agcc agat actt actg actc acgt acgg acgc acct accg accc acat aatt aatg aatc agt agg
2	$0,538071$	agct agcg agcc agat actt actg actc acgt acgg acgc acct accg accc acat aatt aatg aatc aagg agt
3	$0,538071$	agct agcg agcc agat actt actg actc acgt acgg acgc acct accg accc acat aatt aatg aatc aagt agg
4	$0,530991$	agct agcg agcc agat actt actg actc acgt acgg acgc acct accg accc acat aatt aatg aatc aagt aagg
5	$0,538071$	aggc agct agcg agcc agat actt actg actc acgt acgg acgc acct accg accc acat aatt aatg aatc agt
6	$0,530991$	aggc agct agcg agcc agat actt actg actc acgt acgg acgc acct accg accc acat aatt aatg aatc aagt

Fortsetzung auf nächster Seite

Nr.	r	Code
7	$0,538071$	aggg agct agcg agcc agat actt actg actc acgt acgg acgc acct accg accc acat aatt aatg aatc agt
8	$0,530991$	aggg agct agcg agcc agat actt actg actc acgt acgg acgc acct accg accc acat aatt aatg aatc aagt
9	$0,538071$	aggg aggc agcg agcc agat actt actg actc acgt acgg acgc acct accg accc acat aatt aatg aatc agt
10	$0,530991$	aggg aggc agcg agcc agat actt actg actc acgt acgg acgc acct accg accc acat aatt aatg aatc aagt
11	$0,538071$	aggg aggc agct agcc agat actt actg actc acgt acgg acgc acct accg accc acat aatt aatg aatc agt
12	$0,530991$	aggg aggc agct agcc agat actt actg actc acgt acgg acgc acct accg accc acat aatt aatg aatc aagt
13	$0,538071$	aggg aggc agct agcg agat actt actg actc acgt acgg acgc acct accg accc acat aatt aatg aatc agt
14	$0,530991$	aggg aggc agct agcg agat actt actg actc acgt acgg acgc acct accg accc acat aatt aatg aatc aagt
15	$0,538071$	aggg aggc agct agcg agcc actt actg actc acgt acgg acgc acct accg accc acat aatt aatg aatc agt
16	$0,530991$	aggg aggc agct agcg agcc actt actg actc acgt acgg acgc acct accg accc acat aatt aatg aatc aagt
17	$0,538071$	aggg aggc agct agcg agcc agat actg actc acgt acgg acgc acct accg accc acat aatt aatg aatc agt
18	$0,530991$	aggg aggc agct agcg agcc agat actg actc acgt acgg acgc acct accg accc acat aatt aatg aatc aagt
19	$0,538071$	aggg aggc agct agcg agcc agat actt actc acgt acgg acgc acct accg accc acat aatt aatg aatc agt
20	$0,530991$	aggg aggc agct agcg agcc agat actt actc acgt acgg acgc acct accg accc acat aatt aatg aatc aagt

Tab. A.16: Solide Codes, n=19

Nr.	r	Code
1	$0,54708$	aggg aggc agct agcg agcc agat actt actg actc acgt acgg acgc acct accg accc acat aatt aatg aatc agt
2	$0,540241$	aggg aggc agct agcg agcc agat actt actg actc acgt acgg acgc acct accg accc acat aatt aatg aatc aagt
3	$0,54708$	aggt aggc agct agcg agcc agat actt actg actc acgt acgg acgc acct accg accc acat aatt aatg aatc agt
4	$0,540241$	aggt aggc agct agcg agcc agat actt actg actc acgt acgg acgc acct accg accc acat aatt aatg aatc aagt
5	$0,54708$	aggt aggg agct agcg agcc agat actt actg actc acgt acgg acgc acct accg accc acat aatt aatg aatc agt
6	$0,540241$	aggt aggg agct agcg agcc agat actt actg actc acgt acgg acgc acct accg accc acat aatt aatg aatc aagt
7	$0,54708$	aggt aggg aggc agcg agcc agat actt actg actc acgt acgg acgc acct accg accc acat aatt aatg aatc agt
8	$0,540241$	aggt aggg aggc agcg agcc agat actt actg actc acgt acgg acgc acct accg accc acat aatt aatg aatc aagt
9	$0,54708$	aggt aggg aggc agct agcc agat actt actg actc acgt acgg acgc acct accg accc acat aatt aatg aatc agt
10	$0,540241$	aggt aggg aggc agct agcc agat actt actg actc acgt acgg acgc acct accg accc acat aatt aatg aatc aagt
11	$0,54708$	aggt aggg aggc agct agcg agat actt actg actc acgt acgg acgc acct accg accc acat aatt aatg aatc agt
12	$0,540241$	aggt aggg aggc agct agcg agat actt actg actc acgt acgg acgc acct accg accc acat aatt aatg aatc aagt
13	$0,54708$	aggt aggg aggc agct agcg agcc actt actg actc acgt acgg acgc acct accg accc acat aatt aatg aatc agt
14	$0,540241$	aggt aggg aggc agct agcg agcc actt actg actc acgt acgg acgc acct accg accc acat aatt aatg aatc aagt
15	$0,54708$	aggt aggg aggc agct agcg agcc agat actg actc acgt acgg acgc acct accg accc acat aatt aatg aatc agt

Fortsetzung auf nächster Seite

Nr.	r	Code
16	$0,540241$	aggt aggg aggc agct agcg agcc agat actg actc acgt acgg acgc acct accg accc acat aatt aatg aatc aagt
17	$0,54708$	aggt aggg aggc agct agcg agcc agat actt actc acgt acgg acgc acct accg accc acat aatt aatg aatc agt
18	$0,540241$	aggt aggg aggc agct agcg agcc agat actt actc acgt acgg acgc acct accg accc acat aatt aatg aatc aagt
19	$0,54708$	aggt aggg aggc agct agcg agcc agat actt actg acgt acgg acgc acct accg accc acat aatt aatg aatc agt
20	$0,540241$	aggt aggg aggc agct agcg agcc agat actt actg acgt acgg acgc acct accg accc acat aatt aatg aatc aagt

Tab. A.17: Solide Codes, n=20

A.3 Solide binäre Codes

Nr.	r	Code
1	$0,180346$	aaaccaac aaacacac aaaccac aaacacc aaacaac aaaccc
2	$0,176247$	aaaccaac aaacacac aaaccac aaacacc aaacaac aaaaccc
3	$0,176247$	aaaccaac aaacacac aaacccc aaaccac aaacacc aaacaac
4	$0,172331$	aaaccaac aaacacac aaaaaccc aaaccac aaacacc aaacaac
5	$0,176247$	aaaccaac aaacacac aaaacaac aaaccac aaacacc aaaccc
6	$0,172331$	aaaccaac aaacacac aaaacaac aaaccac aaacacc aaaaccc
7	$0,172331$	aaaccaac aaacacac aaaacaac aaacccc aaaccac aaacacc
8	$0,168585$	aaaccaac aaacacac aaaacaac aaaaaccc aaaccac aaacacc

Fortsetzung auf nächster Seite

Nr.	r	Code
9	$0,176247$	aaaccaac aaacacac aaaacacc aaaccac aaacaac aaaccc
10	$0,172331$	aaaccaac aaacacac aaaacacc aaaccac aaacaac aaaaccc
11	$0,172331$	aaaccaac aaacacac aaaacacc aaaccccc aaaccac aaacaac
12	$0,168585$	aaaccaac aaacacac aaaacacc aaaaaccc aaaccac aaacaac
13	$0,172331$	aaaccaac aaacacac aaaacacc aaaacaac aaaccac aaaccc
14	$0,168585$	aaaccaac aaacacac aaaacacc aaaacaac aaaccac aaaaccc
15	$0,168585$	aaaccaac aaacacac aaaacacc aaaacaac aaaccccc aaaccac
16	$0,164998$	aaaccaac aaacacac aaaacacc aaaacaac aaaaaccc aaaccac
17	$0,176247$	aaaccaac aaacacac aaaaccac aaacacc aaacaac aaaccc
18	$0,172331$	aaaccaac aaacacac aaaaccac aaacacc aaacaac aaaaccc
19	$0,172331$	aaaccaac aaacacac aaaaccac aaaccccc aaacacc aaacaac
20	$0,168585$	aaaccaac aaacacac aaaaccac aaaaaccc aaacacc aaacaac

Tab. A.18: Solide binäre Codes, n=6

Nr.	r	Code
1	$0,188957$	aaacccac aaaccaac aaacacac aaaccccc aaaccac aaacacc aaacaac
Fortsetzung auf nächster Seite		

Nr.	r	Code
2	$0,185391$	aaacccac aaaccaac aaacacac aaaacaac aaacccc aaaccac aaacacc
3	$0,185391$	aaacccac aaaccaac aaacacac aaaacacc aaacccc aaaccac aaacaac
4	$0,181958$	aaacccac aaaccaac aaacacac aaaacacc aaaacaac aaacccc aaaccac
5	$0,185391$	aaacccac aaaccaac aaacacac aaaaccac aaacccc aaacacc aaacaac
6	$0,181958$	aaacccac aaaccaac aaacacac aaaaccac aaaacaac aaacccc aaacacc
7	$0,181958$	aaacccac aaaccaac aaacacac aaaaccac aaaacacc aaacccc aaacaac
8	$0,17865$	aaacccac aaaccaac aaacacac aaaaccac aaaacacc aaaacaac aaacccc
9	$0,185391$	aaacccac aaaccaac aaacacac aaaaaccc aaaccac aaacacc aaacaac
10	$0,181958$	aaacccac aaaccaac aaacacac aaaaaccc aaaacaac aaaccac aaacacc
11	$0,181958$	aaacccac aaaccaac aaacacac aaaaaccc aaaacacc aaaccac aaacaac
12	$0,17865$	aaacccac aaaccaac aaacacac aaaaaccc aaaacacc aaaacaac aaaccac
13	$0,181958$	aaacccac aaaccaac aaacacac aaaaaccc aaaaccac aaacacc aaacaac
14	$0,17865$	aaacccac aaaccaac aaacacac aaaaaccc aaaaccac aaaacaac aaacacc
15	$0,17865$	aaacccac aaaccaac aaacacac aaaaaccc aaaaccac aaaacacc aaacaac
16	$0,17546$	aaacccac aaaccaac aaacacac aaaaaccc aaaaccac aaaacacc aaaacaac

Fortsetzung auf nächster Seite

Nr.	r	Code
17	$0,185391$	aaacccac aaaccaac aaacacac aaacaacc aaacccc aaaccac aaacacc
18	$0,181958$	aaacccac aaaccaac aaacacac aaacaacc aaaacacc aaacccc aaaccac
19	$0,181958$	aaacccac aaaccaac aaacacac aaacaacc aaaaccac aaacccc aaacacc
20	$0,17865$	aaacccac aaaccaac aaacacac aaacaacc aaaaccac aaaacacc aaacccc

Tab. A.19: Solide binäre Codes, n=7

Nr.	r	Code
1	$0,196721$	aaccacac aacaaccc aacaacac aaacccac aaacacac aaccacc aaacccc aaacacc
2	$0,193548$	aaccacac aacaaccc aacaacac aaacccac aaacacac aaaacacc aaccacc aaacccc
3	$0,193548$	aaccacac aacaaccc aacaacac aaacccac aaacacac aaaaccccc aaccacc aaacacc
4	$0,190476$	aaccacac aacaaccc aacaacac aaacccac aaacacac aaaaccccc aaaacacc aaccacc
5	$0,193548$	aaccacac aacaaccc aacaacac aaacccac aaacaccc aaacacac aaccacc aaacccc
6	$0,190476$	aaccacac aacaaccc aacaacac aaacccac aaacaccc aaacacac aaaaccccc aaccacc
7	$0,193548$	aaccacac aacaaccc aacaacac aaacccac aaaccacc aaacacac aaacccc aaacacc
8	$0,190476$	aaccacac aacaaccc aacaacac aaacccac aaaccacc aaacacac aaaacacc aaacccc
9	$0,190476$	aaccacac aacaaccc aacaacac aaacccac aaaccacc aaacacac aaaaccccc aaacacc

Fortsetzung auf nächster Seite

Nr.	r	Code
10	$0,1875$	aaccacac aacaaccc aacaacac aaacccac aaaccacc aaacacac aaaacccc aaaacacc
11	$0,190476$	aaccacac aacaaccc aacaacac aaacccac aaaccacc aaacaccc aaacacac aaaccccc
12	$0,1875$	aaccacac aacaaccc aacaacac aaacccac aaaccacc aaacaccc aaacacac aaaaccccc
13	$0,193548$	aaccacac aacaaccc aacaacac aaaccccc aaacccac aaacacac aaccacc aaacacc
14	$0,190476$	aaccacac aacaaccc aacaacac aaaccccc aaacccac aaacacac aaaacacc aaccacc
15	$0,190476$	aaccacac aacaaccc aacaacac aaaccccc aaacccac aaacaccc aaacacac aaccacc
16	$0,190476$	aaccacac aacaaccc aacaacac aaaccccc aaacccac aaaccacc aaacacac aaacacc
17	$0,1875$	aaccacac aacaaccc aacaacac aaaccccc aaacccac aaaccacc aaacacac aaaacacc
18	$0,1875$	aaccacac aacaaccc aacaacac aaaccccc aaacccac aaaccacc aaacaccc aaacacac
19	$0,2$	aaccacac aacaccac aacaaccc aaacccac aaccacc aacaccc aacacac aaaccccc
20	$0,196721$	aaccacac aacaccac aacaaccc aaacccac aaaaccccc aaccacc aacaccc aacacac

Tab. A.20: Solide binäre Codes, n=8

B. TABELLEN

TABELLENVERZEICHNIS

ABBILDUNGSVERZEICHNIS

C. QUELLCODES

Listing C.1: CCodeWord.h

```
1  #ifndef CODEWORD_H
2  #define CODEWORD_H
3
4  #include <iostream>
5
6  #define MIN_LETTER 0
7  #define MAX_LETTER 3
8
9  class CCodeWord
10 {
11   public:
12     // Konstruktoren
13     CCodeWord();
14     CCodeWord(char *seq);
15     CCodeWord(const CCodeWord &);
16
17     // Destruktor
18     ~CCodeWord();
19
20     // Operatoren
21     CCodeWord& operator=(const CCodeWord&);
22
23     bool operator==(const CCodeWord &);
24
25     bool operator!=(const CCodeWord &);
26
27     bool operator>(const CCodeWord &);
28
29     bool operator<(const CCodeWord &);
30
31     friend std::ostream& operator << (std::ostream&, const CCodeWord&);
32
33     // Konkateniert das this-Codewort mit dem erste Codewort
34     // und liefert das Ergebnis im zweiten.
35     void Concat(const CCodeWord&, CCodeWord &) const;
36
37     // Erzeugt ein Präfix des this-Codewortes mit der angegebenen Länge.
```

```
38      // Gibt FALSE zurück, wenn kein Präfix mit der gewünschten Länge
            existiert,
39      // andernfalls TRUE
40      bool Prefix(int,CCodeWord&) const;
41
42      // Erzeugt ein Suffix des this-Codewortes mit der angegebenen Länge.
43      // Gibt FALSE zurück, wenn kein Suffix mit der gewünschten Länge
            existiert,
44      // andernfalls TRUE
45      bool Suffix(int,CCodeWord&) const;
46
47      // Erzeugt ein Infix des this-Codewortes ab gegebener Position mit der
            angegebenen Länge.
48      // Gibt FALSE zurück, wenn kein Infix mit den gewünschten Parametern
            existiert,
49      // andernfalls TRUE
50      bool Infix(int start, int length, CCodeWord &res) const;
51
52      // Ausgabe der CodeWortLänge
53      int getLength() const;
54
55      // Ausgabe der CodeWortLänge
56      int getPosition() const;
57
58      // Eingabe der CodeWortLänge
59      void setLength(int);
60
61      // Eingabe der CodeWortLänge
62      void setPosition(int);
63
64      // Nächstes Codewort
65      void Next();
66
67      // Vorheriges Codewort
68      void Back();
69
70      // Speichert die Buchstaben des Codeworts
71      int* letter;
72
73      // Bildet das Komplement des Codeworts gemäß der
74      // Watson-Crick-Komplementarität
75      void Complement(CCodeWord& ) const;
76
77      // Bestimmt, ob die Länge des Codeworts zwischen den
78      // gegebenen Werten liegt.
79      bool hasLength(int , int) const;
80
81    private:
82      int length;
```

```
83      int position;
84      static const char beta[4];
85      // Berechnet die Position des Codewortes
86      int CalcPosition() const;
87      // Entfernt alle Vorkommen von c in src und liefert das
88      // Ergebnis in dest
89      static int strrm(const char c, char *src,char *dest);
90
91  };
92
93  #endif
```

Listing C.2: CCodeWord.cpp

```
1   #include <iostream>
2   #include "CCodeWord.h"
3
4   const char CCodeWord::beta[4] = {'a','c','g','t'};
5
6   // Konstruktor
7   // Erstellt das Codewort 'a'.
8   CCodeWord::CCodeWord():length(1),position(1)
9   {
10    this->letter = new int[2];
11    this->letter[1] = MIN_LETTER;
12  }
13
14  // Erzeugt ein Codewort aus dem übergeben String.
15  // seq darf nur aus den Buchstaben a,c,t oder g bestehen.
16  // Leerzeichen werden ignoriert.
17  // Groß- und Kleinschreibung ist irrelevant.
18  CCodeWord::CCodeWord(char *seq)
19  {
20    int l = (int)strlen(seq);
21
22    char *b = (char*)malloc(l+1);
23    memset(b,0,l+1);
24    strcpy(b,seq);
25    strrm(' ',b,b);
26    l = (int)strlen(b);
27    this->letter = new int[l+1];
28    for (int i=1;i<=l;i++)
29    {
30      switch(tolower(b[i-1]))
31      {
32        case 'a': letter[i] = 0; break;
33        case 'c': letter[i] = 1; break;
34        case 'g': letter[i] = 2; break;
35        case 't': letter[i] = 3; break;
36        default: break;
```

```
37        }
38      }
39      this->length = l;
40      this->position = CalcPosition();
41      free(b);
42    }
43
44    // Copy-Konstruktor
45    CCodeWord::CCodeWord(const CCodeWord &cw)
46    {
47      this->letter = new int[cw.getLength()+1];
48      for (int i=1;i<=cw.getLength();i++)
49        this->letter[i] = cw.letter[i];
50      this->length = cw.getLength();
51      this->position = cw.getPosition();
52    }
53
54    // Destruktor
55    CCodeWord::~CCodeWord()
56    {
57      delete[] this->letter;
58    }
59
60    // Weist dem this-Codewort das übergebene Codewort zu.
61    // Es wird eine Tiefenkopie erstellt.
62    CCodeWord& CCodeWord::operator=(const CCodeWord& cw)
63    {
64      this->length = cw.getLength();
65      this->position = cw.getPosition();
66      if (this->letter!=NULL)
67        delete[] this->letter;
68      this->letter = new int[(this->length+1)];
69      for(int i=1; i<=this->length; i++)
70      {
71        this->letter[i] = cw.letter[i];
72      }
73      return *this;
74    }
75
76    // Konkatniert das this-Codewort mit cw und
77    // lierfert das Ergebnis in res
78    //
79    void CCodeWord::Concat(const CCodeWord& cw, CCodeWord &res) const
80    {
81      delete[] res.letter;
82      res.letter = new int[cw.getLength()+getLength()+1];
83      for (int i=1;i<=this->getLength();i++)
84        res.letter[i] = letter[i];
85      for (int i=1;i<=cw.getLength();i++)
```

```
86        res.letter[getLength()+i] = cw.letter[i];
87
88      res.setLength(this->getLength()+cw.getLength());
89      res.setPosition(res.CalcPosition());
90    }
91
92    // Erzeugt ein Präfix des this-Codewortes mit der Länge length
93    // und gibt das Ergebnis in res zurück.
94    // Gibt FALSE zurück, wenn kein Präfix mit der gewünschten Länge existiert,
95    // andernfalls TRUE
96    bool CCodeWord::Prefix(int length,CCodeWord& res) const
97    {
98      if ((length>this->getLength()) || (length<1)) return false;
99      delete[] res.letter;
100     res.letter = new int[length+1];
101     for (int i=1;i<=length;i++)
102       res.letter[i] = this->letter[i];
103     res.setLength(length);
104     res.setPosition(CalcPosition());
105     return true;
106   }
107
108   // Erzeugt ein Suffix des this-Codewortes mit der Länge length
109   // und gibt das Ergebnis in res zurück.
110   // Gibt FALSE zurück, wenn kein Suffix mit der gewünschten Länge existiert,
111   // andernfalls TRUE
112   bool CCodeWord::Suffix(int length,CCodeWord& res) const
113   {
114     if ((length>this->getLength()) || (length<1)) return false;
115     delete[] res.letter;
116     res.letter = new int[length+1];
117     for (int i=1;i<=length;i++)
118       res.letter[i] = this->letter[this->getLength()-length+i];
119     res.setLength(length);
120     res.setPosition(CalcPosition());
121     return true;
122   }
123
124   // Erzeugt ein Infix des this-Codewortes ab der Position start mit der Länge
        length
125   // und gibt das Ergebnis in res zurück.
126   // Gibt FALSE zurück, wenn kein Infix mit den gewünschten Parametern
        existiert,
127   // andernfalls TRUE
128   bool CCodeWord::Infix(int start, int length, CCodeWord &res) const
129   {
130     if (start<1) return false;
131     if (start+length>this->getLength()+1)
132       return false;
```

```
133     if (length <= 0)
134       return false;
135     delete[] res.letter;
136     res.letter = new int[length+1];
137     for (int i=start;i<start+length;i++)
138     {
139       res.letter[i-start+1] = this->letter[i];
140     }
141     res.setLength(length);
142     res.setPosition(res.CalcPosition());
143     return true;
144   }
145
146   // Bestimmt, ob die Länge des Codeworts zwischen den
147   // gegebenen Werten min und max liegt.
148   bool CCodeWord::hasLength(int min, int max) const
149   {
150     return (getLength() >=min) && (getLength()<=max);
151   }
152
153   // Liefert die Länge des Codeworts.
154   int CCodeWord::getLength() const
155   {
156     return this->length;
157   }
158
159
160   // Liefert die Position des Codeworts.
161   int CCodeWord::getPosition() const
162   {
163     return this->position;
164   }
165
166
167   // Legt die Länge des Codeworts fest.
168   void CCodeWord::setLength(int l)
169   {
170     this->length = l;
171   }
172
173
174   // Legt die Position des Codeworts fest.
175   void CCodeWord::setPosition(int p)
176   {
177     this->position = p;
178   }
179
180
181   // Berechnet das nächste Codewort in der Aufzählung.
```

```
182 void CCodeWord::Next()
183 {
184     bool tmp2 = 1;
185     for(int i=this->length; i>=1; i--)
186     {
187         if (tmp2)
188         {
189             if ((this->letter[i]) == MAX_LETTER)
190                 this->letter[i] = MIN_LETTER;
191             else
192             {
193                 this->letter[i]++;
194                 tmp2 = 0;
195             }
196         }
197     }
198     if (tmp2)
199     {
200         this->length++;
201     delete[] this->letter;
202         this->letter = new int[this->length+1];
203         for(int l=1; l<=this->length; l++)
204         {
205             this->letter[l] = MIN_LETTER;
206         }
207     }
208
209     this->position++;
210 }
211
212
213 // Berechnet das vorheriges Codewort.
214 void CCodeWord::Back()
215 {
216     bool tmp2 = 1;
217   if (this->position != 1)
218   {
219       for(int i=this->length; i>=1; i--)
220       {
221           if (tmp2)
222           {
223               if ((this->letter[i]) == MIN_LETTER)
224                   this->letter[i] = MAX_LETTER;
225               else
226               {
227                   this->letter[i]--;;
228                   tmp2 = 0;
229             }
230       }
```

```
231         }
232         if (tmp2)
233             this->length--;
234
235         this->position--;
236     }
237 }
238
239
240
241 // Bildet das Komplement des this-Codeworts gemäß der
242 // Watson-Crick-Komplementarität und liefert das Ergebnis in
243 // comp.
244 void CCodeWord::Complement(CCodeWord & comp) const
245 {
246     delete[] comp.letter;
247     comp.letter = new int[getLength()+1];
248     comp.setLength(getLength());
249
250     for (int i=1;i<=getLength();i++)
251     {
252         comp.letter[getLength()-i+1] = 3-this->letter[i];
253     }
254     comp.setPosition(comp.CalcPosition());
255 }
256
257 // Vergleicht das this-Codewort mit dem übergebenen.
258 // (Führt keinen Tiefenvergleich durch)
259 // Rückgabe:  TRUE, falls this = cw
260 //            FALSE sonst
261 bool CCodeWord::operator==(const CCodeWord &cw)
262 {
263     return cw.getPosition()==getPosition();
264 }
265
266 // Vergleicht das this-Codewort mit dem übergebenen.
267 // (Führt keinen Tiefenvergleich durch)
268 // Rückgabe:  TRUE, falls this vor cw in der Aufzählung erscheint.
269 //            FALSE sonst
270 bool CCodeWord::operator>(const CCodeWord &cw)
271 {
272     return cw.getPosition()>getPosition();
273 }
274
275 // Vergleicht das this-Codewort mit dem übergebenen.
276 // (Führt keinen Tiefenvergleich durch)
277 // Rückgabe:  TRUE, falls this nach cw in der Aufzählung erscheint.
278 //            FALSE sonst
279 bool CCodeWord::operator<(const CCodeWord &cw)
```

```
280 {
281   return cw.getPosition()<getPosition();
282 }
283
284 // Vergleicht das this-Codewort mit dem übergebenen.
285 // (Führt keinen Tiefenvergleich durch)
286 // Rückgabe:  TRUE, falls this ungleich cw ist.
287 //          FALSE sonst
288 bool CCodeWord::operator!=(const CCodeWord &cw)
289 {
290   return !(*this == cw);
291 }
292
293 // Gibt das Codewort in lesbarer Form auf dem übergebenen Stream aus.
294 std::ostream& operator << (std::ostream& os, const CCodeWord& cw)
295 {
296   for(int i=1; i<=cw.length; i++)
297     os << CCodeWord::beta[cw.letter[i]];
298   return os;
299 }
300
301 // Berechnet die Position des Codewortes
302 int CCodeWord::CalcPosition() const
303 {
304   int pos = 1+letter[getLength()];
305   for (int i=1;i<getLength();i++)
306   {
307     pos += (1 << (1 << i))*(letter[getLength()-i]+1);
308   }
309   return pos;
310 }
311
312 // Entfernt alle Vorkommen von c in src und liefert das
313     // Ergebnis in dest
314 int CCodeWord::strrm(const char c, char *src,char *dest)
315 {
316    char *buf = (char *)malloc(strlen(src)+1);
317    memset(buf,0,strlen(src)+1);
318    int corr=0;
319    for (int i=0;i<(int)strlen(src);i++)
320    {
321      if (src[i]!=c)
322        buf[i-corr]=src[i];
323      else
324        corr++;
325    }
326    strcpy(dest,buf);
327    return corr;
328 }
```

Listing C.3: CCode.h

```
1  #ifndef CODE_H
2  #define CODE_H
3
4  #include "CCodeWord.h"
5  #include <iostream>
6
7  // Callback-Funktionszeiger
8  // Parameter:
9  //
10 // 1. void * Codeobjekt
11 // 2. int    laufende Nummer des gefundenen Codes
12 // 3. int    Flag, 0 = ok, 1 = keine weiteren Codes
13 typedef void (*OnCodeFound)(void *,int,int);
14
15 class CCode
16 {
17   public:
18     // Konstruktoren
19     CCode();
20     CCode(CCodeWord *,int);
21     CCode(char *code);
22     CCode(const CCode &);
23
24     // Destruktor
25     ~CCode();
26
27     friend std::ostream& operator << (std::ostream&, const CCode&);
28
29     // N?hster Code
30     void Next();
31
32     // Fgt dem Code ein Codewort hinzu
33     void AddCodeWord(const CCodeWord&);
34
35     void RemoveCodeWord(int);
36
37     // Bestimmt ob ein Codewort im Code enthalten ist
38     bool Contains(const CCodeWord &);
39
40     // Legt die Anzahl der Codew?ter fest
41     void setCount(int);
42
43     // Legt die Position des Codes fest
44     void setPosition(int);
45
46     // Liefert die Position des Codes
47     int getPosition() const;
48
```

```
49      // Liefert die Anzahl der Codew?ter des Codes
50      int getCount() const;
51
52      // Testet ob alle Codew?ter zwischen min und max Zeichen lang sind
53      bool hasCodeWordLength(int min, int max);
54
55      // Testet, ob die bergebenen Codew?ter berlappungsfrei bezglich
56      // der DNA-Involution sind
57      bool isNonOverlappingCodeWord(const CCodeWord&, const CCodeWord&);
58
59      // Testet, ob der Code berlappungsfrei bezglich der DNA-Involution
60      // ist
61      bool isNonOverlapping();
62
63      // Testet, ob die bergebenen Codew?ter Theta-konform sind
64      bool isThetaCompliantCodeWord(const CCodeWord&, const CCodeWord&);
65
66      // Testet, ob der Code Theta-konform ist
67      bool isThetaCompliant();
68
69      // Testet, ob die bergebenen Codew?ter Theta-p-konform sind
70      bool isThetaPCompliantCodeWord(const CCodeWord&, const CCodeWord&);
71
72      // Testet, ob der Code Theta-p-konform ist
73      bool isThetaPCompliant();
74
75      // Testet, ob die bergebenen Codew?ter Theta-s-konform sind
76      bool isThetaSCompliantCodeWord(const CCodeWord&, const CCodeWord&);
77
78      // Testet, ob der Code Theta-p-konform ist
79      bool isThetaSCompliant();
80
81      // Testet ob das erste bergebene Codewort Infix im zweiten ist
82      bool isInfix(const CCodeWord &, const CCodeWord &, bool);
83
84      // Testet ob das erste bergebene Codewort Pr?ix des zweiten ist
85      // gibt die count der bereinstimmenden lettern zurck
86      int isPrefix(const CCodeWord &, const CCodeWord &);
87
88      // Testet ob das erste bergebene Codewort Suffix des zweiten ist
89      // und gibt die count der bereinstimmenden lettern zurck
90      int isSuffix(const CCodeWord &, const CCodeWord &);
91
92      // Testet, ob die bergebenen Codew?ter Theta-frei bezglich der
93      // DNA-Involution sind
94      bool isThetaFreeCodeWord(const CCodeWord&, const CCodeWord&);
95
96      // Testet, ob der Code Theta-frei ist
97      bool isThetaFree();
```

```
98
99      // Testet, ob die bergebenen Codew?ter Theta-sticky-frei bezglich
100     // der DNA-Involution sind
101     bool isThetaStickyFreeCodeWord(const CCodeWord&, const CCodeWord&);
102
103     // Testet, ob der Code 3'-?erhang-frei ist
104     bool is3PrimeOverhangFree();
105
106     // Testet, ob die bergebenen Codew?ter 3'-?erhangfrei bezglich
107     // der DNA-Involution sind
108     bool is3PrimeOverhangFreeCodeWord(const CCodeWord&, const CCodeWord&);
109
110     // Testet, ob der Code 5'-?erhang-frei ist
111     bool is5PrimeOverhangFree();
112
113     // Testet, ob die bergebenen Codew?ter 5'-?erhangfrei bezglich der
114     // DNA-Involution sind
115     bool is5PrimeOverhangFreeCodeWord(const CCodeWord&, const CCodeWord&);
116
117     // Testet, ob die bergebenen Codew?ter berhangfrei bezglich der
118     // DNA-Involution sind
119     bool isOverhangFreeCodeWord(const CCodeWord&, const CCodeWord&);
120
121     // Testet, ob der Code Theta-sticky-frei ist
122     bool isThetaStickyFree();
123
124     // Testet, ob das Codewort Theta(k, m1, m2)-Teilwort-konform ist.
125     bool isSubwordCompliantCodeWord(const CCodeWord&, int k, int m1, int m2)
            ;
126
127     // Testet, ob der Code Theta(k, m1, m2)-Teilwort-konform ist.
128     bool isSubwordCompliant(int k, int m1, int m2);
129
130     // Testet, ob die beiden Codew?ter solide sind.
131     bool isSolidCodeWord(const CCodeWord& w1, const CCodeWord& w2);
132
133     // Testet, ob die beiden Codew?ter solide sind.
134     bool isSolid();
135
136     // Testet, ob die beiden Codew?ter kommafrei sind.
137     bool isCommaFreeCodeWord(const CCodeWord& w1, const CCodeWord& w2);
138
139     // Testet, ob die Codew?ter den Bedingungen fr einen
140     // Theta-k-Code gengen
141     bool iskCodeCodeWord(int k,const CCodeWord &cw1,const CCodeWord & cw2);
142
143     // Testet, ob der Code ein Theta-k-Code ist.
144     bool iskCode(int k);
145
```

```
146     // Testet zwei Codew?ter auf Eigenschaften
147     int CheckProperties(const CCodeWord & cw1,const CCodeWord &cw2,bool);
148
149     // Startet die Rekursion
150     int StartRecursion(OnCodeFound onCodeFound);
151
152     // Die Rekursion zur Aufz?lung der Codes
153     // int Recurse(OnCodeFound onCodeFound,CRecursionParams &params);
154     int Recurse(int, OnCodeFound onCodeFound);
155
156     // Liefert die mittlere Codewortl?ge
157     float getAvgCodewordLength() const;
158
159     float getInformationRate() const;
160
161     bool Compare(const CCodeWord& , const CCodeWord& , int);
162
163     // Speichert ein Array von Codew?tern
164     CCodeWord* word;
165
166     // Speichern zu testende Eigenschaften und deren Parameter
167     bool nonOverlapping;
168     bool thetaCompliant;
169     bool thetaPrefixCompliant;
170     bool thetaSuffixCompliant;
171     bool thetaFree;
172     bool thetaStickyFree;
173     bool restrictCodeWordLength;
174     bool restrictCodeLength;
175     bool restrictCodes;
176     bool theta3OverhangFree;
177     bool theta5OverhangFree;
178     bool thetaOverhangFree;
179     bool thetaSubwordCompliant;
180     bool thetaCode;
181     bool solid;
182     bool commaFree;
183     bool infRate;
184     int k,m1,m2;
185     int kCode;
186     int maxCodes;
187     int minCodeWordLength;
188     int maxCodeWordLength;
189     int minCodeLength;
190     int maxCodeLength;
191     float minInfRate;
192   private:
193     int count;
194     int codesFound;
```

```
195     int position;
196
197     int CalcPosition();
198     float getMinInfRateForBranch();
199     float getMaxInfRateForBranch();
200     float getMaxInfRateForBranch(int depth,int maxLength);
201     int CheckInfRate();
202     int CheckInfRate(int tiefe, int maxLength);
203     float getMaxInfRate(int n);
204     int getMinSumOfCodewordLenghts(int n);
205
206 };
207
208 #endif
```

Listing C.4: CCode.cpp

```
1  #include <iostream>
2  #include <string>
3  #include "CCode.h"
4  #include <math.h>
5  using namespace std;
6  // Default-Konstruktor
7  //
8  // Erzeugt den Code {a}
9  CCode::CCode()
10 {
11   word = new CCodeWord[2];
12   CCodeWord * cw = new CCodeWord();
13   this->word[1] = *cw;
14   count = 1;
15   nonOverlapping = false;
16   position = 1;
17   theta3OverhangFree=false;
18   thetaCompliant=false;
19   theta5OverhangFree=false;
20   thetaFree=false;
21   thetaPrefixCompliant=false;
22   thetaOverhangFree=false;
23   thetaSuffixCompliant=false;
24   thetaStickyFree=false;
25   minCodeWordLength=0;
26   maxCodeWordLength=0;
27   restrictCodeWordLength=false;
28   restrictCodeLength=false;
29   minCodeLength=0;
30   maxCodeLength=0;
31   restrictCodes=false;
32   thetaSubwordCompliant=false;
33   k=0;
```

```
34     m1=0;
35     thetaCode=false;
36     m2=0;
37     solid=false;
38     kCode=0;
39     commaFree=false;
40     infRate = false;
41     minInfRate = 0.0f;
42     codesFound = 0;
43  }
44
45  // Erzeugt einen Code aus einem String kommagetrennter
46  // Codewörter
47  CCode::CCode(char *code)
48  {
49     word = new CCodeWord[1];
50     CCodeWord * cw = new CCodeWord();
51     this->word[1] = *cw;
52     count = 0;
53     nonOverlapping = false;
54     position = 1;
55     theta3OverhangFree=false;
56     thetaCompliant=false;
57     theta5OverhangFree=false;
58     thetaFree=false;
59     thetaPrefixCompliant=false;
60     thetaOverhangFree=false;
61     thetaSuffixCompliant=false;
62     thetaStickyFree=false;
63     minCodeWordLength=0;
64     maxCodeWordLength=0;
65     restrictCodeWordLength=false;
66     restrictCodeLength=false;
67     minCodeLength=0;
68     maxCodeLength=0;
69     restrictCodes=false;
70     thetaSubwordCompliant=false;
71     k=0;
72     m1=0;
73     thetaCode=false;
74     m2=0;
75     solid=false;
76     kCode=0;
77     commaFree=false;
78     infRate = false;
79     minInfRate = 0.0f;
80     char *tok;
81     char *c = new char[strlen(code)+1];
82     strcpy(c,code);
```

```
83
84    tok = strtok(c,",");
85
86    while (tok!=NULL)
87    {
88      CCodeWord cw(tok);
89      this->AddCodeWord(cw);
90      tok = strtok(NULL,",");
91    }
92    delete[] c;
93  }
94
95
96  CCode::CCode(const CCode & code)
97  {
98    word = new CCodeWord[1];
99    CCodeWord * codew = new CCodeWord();
100   this->word[1] = *codew;
101   count = 0;
102   nonOverlapping = code.nonOverlapping;
103   position = 1;
104   theta3OverhangFree=code.theta3OverhangFree;
105   thetaCompliant=code.thetaCompliant;
106   theta5OverhangFree=code.theta5OverhangFree;
107   thetaFree=code.thetaFree;
108   thetaPrefixCompliant=code.thetaPrefixCompliant;
109   thetaOverhangFree=code.thetaOverhangFree;
110   thetaSuffixCompliant=code.thetaSuffixCompliant;
111   thetaStickyFree=code.thetaStickyFree;
112   minCodeWordLength=code.minCodeWordLength;
113   maxCodeWordLength=code.maxCodeWordLength;
114   restrictCodeWordLength=code.restrictCodeWordLength;
115   restrictCodeLength=code.restrictCodeLength;
116   minCodeLength=code.minCodeLength;
117   maxCodeLength=code.maxCodeLength;
118   restrictCodes=code.restrictCodes;
119   thetaSubwordCompliant=code.thetaSubwordCompliant;
120   k=code.k;
121   m1=code.m1;
122   thetaCode=code.thetaCode;
123   m2=code.m2;
124   solid=code.solid;
125   kCode=code.kCode;
126   commaFree=code.commaFree;
127   infRate = code.infRate;
128   minInfRate = code.minInfRate;
129
130   for (int i=1;i<code.count;i++)
131   {
```

```
132        AddCodeWord(code.word[i]);
133     }
134  }
135  // Erzeugt einen Code aus einem Array von Codewörtern
136  //
137  // cw - Array von Codewörtern
138  // l  - Anzahl der Elemente in cw
139  CCode::CCode(CCodeWord *cw,int l)
140  {
141     word = new CCodeWord[1];
142     CCodeWord * codew = new CCodeWord();
143     this->word[1] = *codew;
144     count = 0;
145     nonOverlapping = false;
146     position = 1;
147     theta3OverhangFree=false;
148     thetaCompliant=false;
149     theta5OverhangFree=false;
150     thetaFree=false;
151     thetaPrefixCompliant=false;
152     thetaOverhangFree=false;
153     thetaSuffixCompliant=false;
154     thetaStickyFree=false;
155     minCodeWordLength=0;
156     maxCodeWordLength=0;
157     restrictCodeWordLength=false;
158     restrictCodeLength=false;
159     minCodeLength=0;
160     maxCodeLength=0;
161     restrictCodes=false;
162     thetaSubwordCompliant=false;
163     k=0;
164     m1=0;
165     thetaCode=false;
166     m2=0;
167     solid=false;
168     kCode=0;
169     commaFree=false;
170     infRate = false;
171     minInfRate = 0.0f;
172
173     for (int i=0;i<l;i++)
174     {
175        AddCodeWord(cw[i]);
176     }
177  }
178
179  // Prüft, ob das bergebene Codewort Element des
180  // Codes ist.
```

```
181 //
182 // Rckgabe: TRUE, falls cw Element von this ist,
183 //          sonst FALSE
184 bool CCode::Contains(const CCodeWord &cw)
185 {
186   for (int i=1;i<=getCount();i++)
187   {
188     if (word[i]==cw) return true;
189   }
190   return false;
191 }
192
193 // Fügt dem Code ein Codewort hinzu
194 // Bei gltigem Codewort wird cw gemäß definierter
195 // Ordnung in this einsortiert.
196 void CCode::AddCodeWord(const CCodeWord& cw)
197 {
198   if (Contains(cw)) return;
199   int i=1;
200   CCodeWord *temp = new CCodeWord[count+2];
201   while ((word[i]<cw) && (i<=count))
202   {
203     temp[i] = *(new CCodeWord(word[i]));
204     i++;
205   }
206   temp[i] = *(new CCodeWord(cw));
207   while (i<=count)
208   {
209     temp[i+1] = word[i];
210     i++;
211   }
212
213   delete[] this->word;
214   this->word = temp;
215   this->count++;
216   this->setPosition(this->CalcPosition());
217 }
218
219 void CCode::RemoveCodeWord(int number)
220 {
221   CCodeWord *temp = new CCodeWord[count+1];
222   for (int i=1;i<number;i++)
223     temp[i] = *(new CCodeWord(word[i]));
224   for (int i=number+1;i<=count;i++)
225     temp[i-1] = *(new CCodeWord(word[i]));
226
227   delete[] this->word;
228   this->word = temp;
229   this->count--;
```

```
230     this->setPosition(this->CalcPosition());
231   }
232
233   // Destruktor
234   CCode::~CCode()
235   {
236     delete[] this->word;
237   }
238
239   // Überläd den Stream-Operator.
240   // Schreibt den Code in den bergebenen Stream.
241   std::ostream& operator << (std::ostream& os, const CCode& c)
242   {
243     for (int i=1; i<=c.count; i++)
244       {
245           os << c.word[i];
246           os << " ";
247       }
248     return os;
249   }
250   // Berechnet den gemäß definierter Ordnung auf den this-Code
251   // folgenen Code
252   void CCode::Next()
253   {
254       int x = this->count;
255       bool done = false;
256       if ( this->word[x].getPosition() == 1 )
257       {
258           while(!done)
259           {
260               if ( x > 1 )
261               {
262                   if ( (this->word[(x-1)].getPosition()-this->word[x].
                        getPosition()) > 1 )
263                   {
264                       this->word[x].Next();
265                       this->count = x;
266                       done = true;
267                   }else
268                   {
269                       if ( this->word[x-1].getPosition()-this->word[x].
                            getPosition() == 1 )
270                       {
271               delete[] this->word[x].letter;
272                           this->word[x].letter = new int[(1+1)];
273               this->word[x].letter[1] = MIN_LETTER;
274                           this->word[x].setPosition(1);
275                           this->word[x].setLength(1);
276                           x--;
```

```
277                     }
278                 }
279             }else
280             {
281                 if ( x == 1)
282                 {
283                     this->word[x].Next();
284                     this->count = x;
285                     done = true;
286                 }
287             }
288         }
289     }else
290     {
291         if ( this->word[x].getPosition() > 1)
292         {
293       CCodeWord* temp = new CCodeWord[((x+1)+1)];
294       for (int k=1; k<=(x); k++)
295       {
296         temp[k] = this->word[k];
297       }
298
299       delete[] this->word;
300       this->word = new CCodeWord[((x+1)+1)];
301
302       for (int k=1; k<=(x); k++)
303       {
304         this->word[k] = temp[k];
305       }
306       delete[] temp;
307       delete[] this->word[x+1].letter;
308             this->word[x+1].letter = new int[(1+1)];
309       this->word[x+1].letter[1] = MIN_LETTER;
310             this->word[x+1].setPosition(1);
311             this->word[x+1].setLength(1);
312             x++;
313             this->count = x;
314         }
315     }
316   this->count = x;
317   this->setPosition(this->getPosition()+1);
318 }
319
320 // Legt die count der Codewörter fest
321 void CCode::setCount(int anz)
322 {
323   this->count = anz;
324 }
325
```

```
326 // Legt die Position des Codes fest
327 void CCode::setPosition(int pos)
328 {
329   this->position = pos;
330 }
331
332 // Liefert die Position des Codes
333 int CCode::getPosition() const
334 {
335   return this->position;
336 }
337
338 // Liefert die Anzahl der Codewörter des Codes
339 int CCode::getCount() const
340 {
341   return this->count;
342 }
343
344
345 // Testet, ob die beiden Codewörter solide sind.
346 bool CCode::isSolidCodeWord(const CCodeWord& codew1, const CCodeWord& codew2
      )
347 {
348   int start = (codew2.getLength() * (-1)) + 1;
349   int ende = codew1.getLength() - 1;
350   for (int t=start; t<=ende; t++)
351   {
352     if (! ((codew1.getPosition()==codew2.getPosition()) && (t==0)) )
353       if ( this->Compare(codew1,codew2,t) )
354       {
355         return false;
356       }
357   }
358   return true;
359 }
360
361 // Testet zwei Codewörter auf Eigenschaften
362 //
363 // cw1, cw2 - zu testende Codew?ter
364 // unaryOnly - TRUE, wenn nur ein einzelnes Codewort
365 //         betreffende Eigenschaften getestet werden sollen.
366 //         In diesem Fall ist cw2 bedeutungslos.
367 //         FALSE, wenn alle Eigenschaften getestet werden sollen.
368 //
369 // Rückgabe:  -1, falls mind. eine Eigenschaften nicht erfllbar ist
370 //         0, falls mind. eine Eigenschaft nicht erfllt ist
371 //         1, falls alle Eigenschaften erfllt sind
372 int CCode::CheckProperties(const CCodeWord & cw1,const CCodeWord &cw2,bool
      unaryOnly)
```

```
373 {
374   if (restrictCodeWordLength)
375   {
376     if (cw1.getLength()>this->maxCodeWordLength)
377       return -1;
378     if (!cw1.hasLength(this->minCodeWordLength,this->maxCodeWordLength))
379       return 0;
380   }
381
382   if ((this->thetaSubwordCompliant) && (unaryOnly))
383     if (!isSubwordCompliantCodeWord(cw1,this->k,this->m1,this->m2))
384       return 0;
385
386   if (unaryOnly) return true;
387
388   if ((this->thetaPrefixCompliant) && !(this->theta3OverhangFree) && !(this
        ->thetaCompliant) && (!this->thetaOverhangFree))
389     if (!isThetaPCompliantCodeWord(cw1,cw2))
390       return 0;
391
392   if ((this->thetaSuffixCompliant) && !(this->thetaCompliant) && !(this->
        theta5OverhangFree)  && (!this->thetaOverhangFree))
393     if (!isThetaSCompliantCodeWord(cw1,cw2))
394       return 0;
395
396   if (this->thetaCompliant)
397     if (!isThetaCompliantCodeWord(cw1,cw2))
398       return 0;
399
400   if (this->nonOverlapping)
401     if (!isNonOverlappingCodeWord(cw1,cw2))
402       return 0;
403
404   if ((this->thetaOverhangFree) || ((theta3OverhangFree) && (
        theta5OverhangFree)))
405     if (!isOverhangFreeCodeWord(cw1,cw2))
406       return 0;
407
408   if ((this->theta3OverhangFree) && (!thetaOverhangFree))
409     if (!is3PrimeOverhangFreeCodeWord(cw1,cw2))
410       return 0;
411
412   if ((this->theta5OverhangFree) && (!thetaOverhangFree))
413     if (!is5PrimeOverhangFreeCodeWord(cw1,cw2))
414       return 0;
415
416   if (this->thetaCode)
417     if (!iskCodeCodeWord(this->kCode,cw1,cw2))
418       return 0;
```

```
419
420   if (this->thetaStickyFree)
421     if (!isThetaStickyFreeCodeWord(cw1,cw2))
422       return 0;
423
424   if (this->solid)
425     if (!isSolidCodeWord(cw1,cw2))
426       return 0;
427
428   if (this->commaFree)
429     if (!isCommaFreeCodeWord(cw1,cw2))
430       return 0;
431
432   if (this->thetaFree)
433     if (!isThetaFreeCodeWord(cw1,cw2))
434       return 0;
435
436   return 1;
437 }
438
439 // Testet, ob die bergebenen Codewörter überlappungsfrei bezglich
440 // der DNA-Involution sind
441 //
442 // Rückgabe:  0, falls w1 und w2 nicht Theta-überlappungsfrei sind.
443 //          1 sonst
444 bool CCode::isNonOverlappingCodeWord(const CCodeWord& w1, const CCodeWord&
      w2)
445 {
446   CCodeWord cw;
447   w2.Complement(cw);
448   return cw!=w1;
449 }
450
451 // Testet, ob der this-Code überlappungsfrei bezglich der DNA-Involution ist
      .
452 //
453 // Rückgabe:  0, falls this nicht Theta-überlappungsfrei ist
454 //          1 sonst
455 bool CCode::isNonOverlapping()
456 {
457   for (int i=1; i<=this->count; i++)
458     for (int l=i; l<=this->count; l++)
459     {
460       if (!isNonOverlappingCodeWord(this->word[l],this->word[i]))
461         return false;
462     }
463   return true;
464 }
465
```

```
466 // Testet, ob die bergebenen Codewörter berhangfrei bezglich
467 // der DNA-Involution sind
468 //
469 // Rückgabe:  0, falls w1 und w2 nicht Theta-überhangfrei sind.
470 //          1 sonst
471 bool CCode::isOverhangFreeCodeWord(const CCodeWord&cw1, const CCodeWord& cw2
        )
472 {
473   return this->is3PrimeOverhangFreeCodeWord(cw1,cw2) &&
          is5PrimeOverhangFreeCodeWord(cw1,cw2);
474 }
475
476 // Testet, ob die bergebenen Codewörter konform bezüglich
477 // der DNA-Involution sind
478 //
479 // Rückgabe:  0, falls w1 und w2 nicht Theta-konform sind.
480 //          1 sonst
481 bool CCode::isThetaCompliantCodeWord(const CCodeWord& w1, const CCodeWord&
        w2)
482 {
483   bool conf = true;
484
485   int diff = abs(w1.getLength()-w2.getLength());
486   if (diff<1)
487     return conf;
488
489   CCodeWord comp;
490   if (w1.getLength()<w2.getLength())
491   {
492     w1.Complement(comp);
493     return !isInfix(comp,w2,false);
494   } else
495   {
496     w2.Complement(comp);
497     return !isInfix(comp,w1,false);
498   }
499 }
500
501 // Testet, ob der this-Code konform bezüglich der DNA-Involution ist.
502 //
503 // Rückgabe:  0, falls this nicht Theta-konform ist
504 //          1 sonst
505 bool CCode::isThetaCompliant()
506 {
507   for (int i=1; i<=this->count; i++)
508     for (int l=i; l<=this->count; l++)
509     {
510       if (!isThetaCompliantCodeWord(this->word[l],this->word[i]))
511         return false;
```

```
512       }
513     return true;
514   }
515
516   // Testet, ob die bergebenen Codewörter präfixkonform bezüglich
517   // der DNA-Involution sind
518   //
519   // Rückgabe:  0, falls w1 und w2 nicht Theta-p-konform sind.
520   //            1 sonst
521   bool CCode::isThetaPCompliantCodeWord(const CCodeWord& w1, const CCodeWord&
        w2)
522   {
523     bool conf = true;
524     if (w1.getLength()==w2.getLength())
525       return conf;
526
527     CCodeWord comp;
528     if (w1.getLength()<w2.getLength())
529     {
530       w1.Complement(comp);
531       return !(isPrefix(comp,w2) == w1.getLength());
532     } else
533     {
534       w2.Complement(comp);
535       return !(isPrefix(comp,w1) == w2.getLength());
536     }
537   }
538
539   // Testet, ob der this-Code präfixkonform bezglich der DNA-Involution ist.
540   //
541   // Rückgabe:  0, falls this nicht Theta-p-konform ist
542   //            1 sonst
543   bool CCode::isThetaPCompliant()
544   {
545     for (int i=1; i<=this->count; i++)
546       for (int l=i; l<=this->count; l++)
547       {
548         if (!isThetaPCompliantCodeWord(this->word[l],this->word[i]))
549           return false;
550       }
551     return true;
552   }
553
554   // Testet, ob die übergebenen Codewörter thetafrei sind.
555   //
556   // Rückgabe:  0, falls w1 und w2 nicht Theta-überhangfrei sind.
557   //            1 sonst
558   bool CCode::isThetaFreeCodeWord(const CCodeWord& w1, const CCodeWord& w2)
559   {
```

```
560     for (int c=1;c<=this->count;c++)
561     {
562       CCodeWord conc1,conc2;
563       CCodeWord comp;
564       this->word[c].Complement(comp);
565       w1.Concat(w2,conc1);
566       w2.Concat(w1,conc2);
567       if ((conc1.getLength()-2) < comp.getLength())
568         return true;
569       if (isInfix(comp,conc1,true)) return false;
570       if (isInfix(comp,conc2,true)) return false;
571     }
572     return true;
573   }
574
575
576   // Testet, ob der this-Code Theta-frei ist.
577   //
578   // Rückgabe:  0, falls this nicht Theta-frei ist
579   //            1 sonst
580   bool CCode::isThetaFree()
581   {
582     for (int i=1; i<=this->count; i++)
583       for (int l=i; l<=this->count; l++)
584       {
585         if (!isThetaFreeCodeWord(this->word[l],this->word[i]))
586           return false;
587       }
588     return true;
589   }
590
591   // Testet, ob die bergebenen Codewörter suffixkonform bezüglich
592   // der DNA-Involution sind
593   //
594   // Rückgabe:  0, falls w1 und w2 nicht Theta-s-konform sind.
595   //            1 sonst
596   bool CCode::isThetaSCompliantCodeWord(const CCodeWord& w1, const CCodeWord&
        w2)
597   {
598     bool conf = true;
599     if (w1.getLength()==w2.getLength())
600       return conf;
601
602     CCodeWord comp;
603     if (w1.getLength()<w2.getLength())
604     {
605       w1.Complement(comp);
606       return !(isSuffix(comp,w2) == w1.getLength());
607     } else
```

```
608     {
609       w2.Complement(comp);
610       return !(isSuffix(comp,w1) == w2.getLength());
611     }
612   }
613
614   // Testet, ob der this-Code suffixkonform bezüglich der DNA-Involution ist.
615   //
616   // Rückgabe:  0, falls this nicht Theta-s-konform ist
617   //            1 sonst
618   bool CCode::isThetaSCompliant()
619   {
620     for (int i=1; i<=this->count; i++)
621       for (int l=i; l<=this->count; l++)
622       {
623         if (!isThetaSCompliantCodeWord(this->word[l],this->word[i]))
624           return false;
625       }
626     return true;
627   }
628
629   // Testet, ob der this-Code solide ist.
630   //
631   // Rückgabe:  0, falls this nicht solide ist
632   //            1 sonst
633   bool CCode::isSolid()
634   {
635     for (int i=1; i<=this->count; i++)
636       for (int l=i; l<=this->count; l++)
637       {
638         if (!isSolidCodeWord(this->word[l],this->word[i]))
639           return false;
640       }
641     return true;
642   }
643
644   // Testet, ob die bergebenen Codewörter sticky-frei bezüglich der
645   // DNA-Involution sind.
646   //
647   // Rückgabe:  0, falls w1 und w2 nicht Theta-sticky-frei sind.
648   //            1 sonst
649   bool CCode::isThetaStickyFreeCodeWord(const CCodeWord& w1, const CCodeWord&
        w2)
650   {
651     CCodeWord comp;
652     w1.Complement(comp);
653     int r = isSuffix(comp,w2);
654
655     if (r != 0)
```

```
656     {
657       if ((r<w1.getLength()) || (r<w2.getLength()))
658         return false;
659     }
660
661     w2.Complement(comp);
662     r = isSuffix(comp,w1);
663
664     if (r != 0)
665     {
666       if ((r<w1.getLength()) || (r<w2.getLength()))
667         return false;
668     }
669     return true;
670   }
671
672   // Testet, ob der this-Code stickyfrei bezüglich der DNA-Involution ist.
673   //
674   // Rückgabe:  0, falls this nicht Theta-sticky-frei ist
675   //            1 sonst
676   bool CCode::isThetaStickyFree()
677   {
678     for (int i=1; i<=this->count; i++)
679       for (int l=i; l<=this->count; l++)
680       {
681         if (!isThetaStickyFreeCodeWord(this->word[l],this->word[i]))
682           return false;
683       }
684     return true;
685   }
686
687   // Testet, ob die bergebenen Codewörter 3'-überhangfrei bezüglich der
688   // DNA-Involution sind.
689   //
690   // Rückgabe:  0, falls w1 und w2 nicht Theta-3'-überhang-frei sind.
691   //            1 sonst
692   bool CCode::is3PrimeOverhangFreeCodeWord(const CCodeWord& w1, const
        CCodeWord& w2)
693   {
694     int l,r;
695     CCodeWord comp,pre;
696
697     l = (w1.getLength()<=w2.getLength())?w1.getLength():w2.getLength();
698
699     for (int i=1;i<=l;i++)
700     {
701       w1.Prefix(i,pre);
702       pre.Complement(comp);
703       r = isPrefix(comp,w2);
```

```
704       if (r == i) {
705         if ((r<w1.getLength()) || (r<w2.getLength()))
706           return false;
707       }
708
709       w2.Prefix(i,pre);
710       pre.Complement(comp);
711       r = isPrefix(comp,w1);
712       if (r != i) continue;
713       if ((r<w1.getLength()) || (r<w2.getLength()))
714         return false;
715     }
716
717     return true;
718   }
719
720
721   // Testet, ob der this-Code 3'-berhangfrei bezüglich der DNA-Involution ist.
722   //
723   // Rückgabe:  0, falls this nicht Theta-3'-überhangfrei ist
724   //            1 sonst
725   bool CCode::is3PrimeOverhangFree()
726   {
727     for (int i=1; i<=this->count; i++)
728       for (int l=i; l<=this->count; l++)
729       {
730         if (!is3PrimeOverhangFreeCodeWord(this->word[l],this->word[i]))
731           return false;
732       }
733     return true;
734   }
735
736   // Testet, ob die übergebenen Codewörter 5'-überhangfrei bezglich der
737   // DNA-Involution sind.
738   //
739   // Rückgabe:  0, falls w1 und w2 nicht Theta-5'-überhang-frei sind.
740   //            1 sonst
741   bool CCode::is5PrimeOverhangFreeCodeWord(const CCodeWord& w1, const
        CCodeWord& w2)
742   {
743
744     int l,r;
745     CCodeWord comp,suff;
746
747     l = (w1.getLength()<=w2.getLength())?w1.getLength():w2.getLength();
748
749     for (int i=1;i<=l;i++)
750     {
751       w1.Suffix(i,suff);
```

```
752       suff.Complement(comp);
753       r = isSuffix(comp,w2);
754       if (r == i) {
755       if ((r<w1.getLength()) || (r<w2.getLength()))
756         return false;
757       }
758
759       w2.Suffix(i,suff);
760       suff.Complement(comp);
761       r = isSuffix(comp,w1);
762       if (r != i) continue;
763       if ((r<w1.getLength()) || (r<w2.getLength()))
764         return false;
765     }
766
767     return true;
768   }
769
770
771   // Testet, ob der this-Code 5'-überhangfrei bezüglich der DNA-Involution ist
        .
772   //
773   // Rückgabe:  0, falls this nicht Theta-5'-berhangfrei ist
774   //            1 sonst
775   bool CCode::is5PrimeOverhangFree()
776   {
777     for (int i=1; i<=this->count; i++)
778       for (int l=i; l<=this->count; l++)
779       {
780         if (!is5PrimeOverhangFreeCodeWord(this->word[l],this->word[i]))
781           return false;
782       }
783     return true;
784   }
785
786   // Testet, ob die übergebenen Codewörter Theta(k,m1,m2)-teilwortkonform
787   // sind.
788   //
789   // Rückgabe:  0, falls w1 und w2 nicht Theta(k,m1,m2)-teilwortkonform sind.
790   //            1 sonst
791   bool CCode::isSubwordCompliantCodeWord(const CCodeWord& cw, int k, int m1,
        int m2)
792   {
793     if (cw.getLength()<(k*2+m1)) return true;
794     for (int m=m1;m<=m2;m++) // über die Länge des Mittelstücks
795     {
796       for (int myK=1;myK<=cw.getLength()-2*k-m+1;myK++)
797       {
798         CCodeWord u;
```

```
799         CCodeWord cu1,cu2;
800         cw.Infix(myK,k,u);
801         cw.Infix(myK+k+m,k,cu1);
802         u.Complement(cu2);
803         if (cu1==cu2) return false;
804       }
805     }
806     return true;
807   }
808
809   // Testet, ob der this-Code Theta(k,m1,m2)-teilwortkonform ist.
810   //
811   // Rückgabe:  0, falls this nicht Theta(k,m1,m2)-teilwortkonform ist.
812   //            1 sonst
813   bool CCode::isSubwordCompliant(int k, int m1, int m2)
814   {
815     for (int i=1;i<=this->getCount();i++)
816     {
817       if (!isSubwordCompliantCodeWord(this->word[i],k,m1,m2))
818         return false;
819     }
820     return true;
821   }
822
823   // Testet, ob die bergebenen Codewörter einen kommafreien
824   // Code bilden.
825   //
826   // Rückgabe:  0, falls w1 und w2 keinen kommafreien Code bilden.
827   //            1 sonst
828   bool CCode::isCommaFreeCodeWord(const CCodeWord& w1, const CCodeWord& w2)
829   {
830     bool conf = true;
831     if (w1.getLength()==w2.getLength())
832       return conf;
833
834     CCodeWord comp;
835     if (w1.getLength()<w2.getLength())
836     {
837       return !(isPrefix(w1,w2) == w1.getLength());
838     } else
839     {
840       return !(isPrefix(w2,w1) == w2.getLength());
841     }
842   }
843
844   // Testet, ob die übergebenen Codewörter einen k-Code bilden.
845   //
846   // Rückgabe:  0, falls w1 und w2 keinen k-Code bilden.
847   //            1 sonst
```

```
848 bool CCode::iskCodeCodeWord(int k,const CCodeWord &cw1,const CCodeWord & cw2
      )
849 {
850   CCodeWord comp;
851   cw2.Complement(comp);
852
853   for (int i=1;i<=cw1.getLength()-k+1;i++)
854   {
855     CCodeWord inf;
856     cw1.Infix(i,k,inf);
857     if (isInfix(inf,comp,false))
858       return false;
859   }
860
861   cw1.Complement(comp);
862   for (int i=1;i<=cw2.getLength()-k+1;i++)
863   {
864     CCodeWord inf;
865     cw2.Infix(i,k,inf);
866     if (isInfix(inf,comp,false))
867       return false;
868   }
869   return true;
870 }
871
872 // Testet, ob der this-Code ein k-Code ist.
873 //
874 // Rückgabe:  0, falls der this-Code kein k-Code ist.
875 //          1 sonst
876 bool CCode::iskCode(int k)
877 {
878 for (int i=1; i<=this->count; i++)
879     for (int l=i; l<=this->count; l++)
880     {
881       if (!iskCodeCodeWord(k,this->word[l],this->word[i]))
882         return false;
883     }
884   return true;
885 }
886
887
888
889 // Testet ob alle Codewörter zwischen min und max Zeichen lang sind
890 //
891 // Rückgabe: TRUE, falls alle Codewörter des this-Codes zwischen
892 // min und max Zeichen lang sind.
893 bool CCode::hasCodeWordLength(int min, int max)
894 {
895   return (word[1].hasLength(min,max) && word[count].hasLength(min,max));
```

```
896 }
897
898 // Testet, ob w1 Infix in w2 ist.
899 //
900 // w1, w2 - Codewörter
901 // strict - TRUE, falls getestet werden soll, ob w1 echtes Infix ist
902 //        FALSE, falls getestet werden soll, ob w1 Infix ist
903 //
904 // Rückgabe:  0, fallse w1 nicht (echtes) Infix in w2 ist
905 //          1, sonst
906 bool CCode::isInfix(const CCodeWord &w1, const CCodeWord &w2, bool strict)
907 {
908   if (w1.getLength()>w2.getLength()) return false;
909   if ((strict) && (w1.getLength()==w2.getLength())) return false;
910
911   for (int i2=1+strict;i2<=w2.getLength()-w1.getLength()+1-strict;i2++)
912   {
913     bool infix = true;
914     for (int i1=1;i1<=w1.getLength();i1++)
915     {
916       if (w1.letter[i1]!=w2.letter[i2+i1-1])
917       {
918         infix = false;
919         break;
920       }
921     }
922     if (infix) return true;
923   }
924   return false;
925 }
926
927 // Testet, ob w1 Präfix von w2 ist.
928 //
929 // w1, w2 - Codewörter
930 //
931 // Rückgabe:  die Anzahl der bereinstimmenden Zeichen.
932 //          = w1.getLength(), falls w1 Präfix von w2
933 //          < w1.getLength() und > 0, falls ein Präfix von w1 Präfix von w2
        ist
934 //          = 0, falls w1 kein Präfix von w2 ist.
935 int CCode::isPrefix(const CCodeWord &w1, const CCodeWord &w2)
936 {
937   int l = (w1.getLength()<=w2.getLength())?w1.getLength():w2.getLength();
938
939   for (int i=1;i<=l;i++)
940   {
941     if (w1.letter[i]!=w2.letter[i])
942     {
943       return i-1;
```

```
944       }
945     }
946     return l;
947   }
948
949   // Testet, ob w1 Suffix von w2 ist.
950   //
951   // w1, w2 - Codewörter
952   //
953   // Rückgabe:  die Anzahl der bereinstimmenden Zeichen.
954   //         = w1.getLength(), falls w1 Suffix von w2
955   //         < w1.getLength() und > 0, falls ein Suffix von w1 Suffix von w2
        ist
956   //         = 0, falls w1 kein Suffix von w2 ist.
957   int CCode::isSuffix(const CCodeWord &w1, const CCodeWord &w2)
958   {
959     int l = (w1.getLength()<=w2.getLength())?w1.getLength():w2.getLength();
960
961     for (int i=1;i<=l;i++)
962     {
963       if (w1.letter[w1.getLength()-i+1]!=w2.letter[w2.getLength()-i+1])
964         return i-1;
965     }
966     return l;
967   }
968
969
970   int CCode::Recurse(int tiefe,OnCodeFound onCodeFound)
971   {
972     int a = 1;
973     bool b = true;
974
975     if (((restrictCodeLength) && (count>=maxCodeLength))) return 0;
976
977     CCodeWord* temp = new CCodeWord[ (tiefe+1) ];
978     // bereits gefundende Codewörter sichern
979     for (int k=1; k<tiefe; k++)
980     {
981       temp[k] = this->word[k];
982     }
983
984     // Codewort-Feld neu anlegen
985     delete[] this->word;
986     this->word = new CCodeWord[ (tiefe+1) ];
987
988     // gesicherte Codewörter in Codewort-Feld ablegen
989     for (int k=1; k<tiefe; k++)
990     {
991       this->word[k] = temp[k];
```

```
992     }
993     this->count = tiefe;
994
995     delete[] temp;
996     delete[] this->word[tiefe].letter;
997     // Wort der Tiefe tiefe initialisieren
998       this->word[tiefe].letter = new int[(1+1)];
999     this->word[tiefe].letter[1] = MIN_LETTER;
1000      this->word[tiefe].setPosition(1);
1001      this->word[tiefe].setLength(1);
1002
1003    for (int i=1; i<this->word[tiefe-1].getPosition(); i++)
1004    {
1005      bool hasMinLength = true;
1006      bool hasInfRate = true;
1007
1008
1009      if (infRate && restrictCodeLength)
1010      {
1011        int ir = CheckInfRate(tiefe,this->maxCodeLength);
1012        if (ir<0) break;
1013        if (ir==0)
1014          hasInfRate = false;
1015      }
1016
1017      if (restrictCodeLength)
1018      {
1019        if (word[tiefe].getPosition()+(tiefe-1)<this->minCodeLength)
1020        {
1021          hasMinLength = false;
1022        }
1023      }
1024
1025      // Codewörter auf Eigenschaften testen
1026      // zunächst neues Wort mit sich selbst testen
1027      if ((CheckProperties(this->word[tiefe],this->word[tiefe],true) == 1) &&
            (hasMinLength) && (hasInfRate))
1028      {
1029        // dann jedes Paar von neuem Codewort mit einem Vorgänger
1030        for ( int l=1; l<tiefe; l++)
1031        {
1032          a = CheckProperties(this->word[l],this->word[tiefe],false);
1033          if (a == 0)
1034          {
1035            b = false;
1036            break;
1037          } else
1038            if (a < 0)
1039              return -1;
```

```
1040        }
1041        if (b)
1042        {
1043          if ((((restrictCodeLength) && (count>=minCodeLength) &&
1044            (count<=maxCodeLength)) || (!restrictCodeLength)) &&
1045            ((infRate && (getInformationRate()>=((minInfRate<0)?this->
                  getMaxInfRate(getCount()):minInfRate))) || (!infRate)))
1046          {
1047            codesFound++;
1048            if (onCodeFound!=NULL)
1049              onCodeFound((void *)this,codesFound,0);
1050            if ((restrictCodes) && (codesFound>=maxCodes))
1051              return -1;
1052
1053          }
1054          int s = this->Recurse(tiefe+1,onCodeFound);
1055          if (s < 0) return s;
1056        }
1057      }
1058      this->word[tiefe].Next();
1059      a = 1;
1060      b = true;
1061    }
1062    // Nach der Rekursion sollte die count wieder um eins verringert werden.
1063    this->count = tiefe-1;
1064    return 1;
1065  }
1066
1067  int CCode::StartRecursion(OnCodeFound onCodeFound)
1068  {
1069    int goOn = 1;
1070    codesFound = 0;
1071
1072
1073    while ((goOn >= 0) && (!restrictCodes ||
1074      ((codesFound<maxCodes) && (restrictCodes))))
1075    {
1076
1077      int ok = CheckProperties(this->word[1],this->word[1],false);
1078      if (ok == 1)
1079        {
1080          if ((((restrictCodeLength) && (count>=minCodeLength) &&
1081            (count<=maxCodeLength)) || (!restrictCodeLength)) &&
1082            ((infRate && (getInformationRate()>=((minInfRate<0)?this->
                  getMaxInfRate(getCount()):minInfRate))) || (!infRate)))
1083          {
1084            codesFound++;
1085            if (onCodeFound!=NULL)
1086              onCodeFound((void *)this,codesFound,0);
```

```
1087          }
1088          if (((restrictCodeLength) && (count<maxCodeLength) && (word[1].
                getPosition()>=minCodeLength)) ||
1089            (!restrictCodeLength))
1090            goOn = this->Recurse(2,onCodeFound);
1091        } else
1092          if (ok<0) {
1093            goOn = -1;
1094            break;
1095          }
1096      this->word[1].Next();
1097    }
1098    if (goOn<0)
1099    {
1100      return -1;
1101    }
1102    return 1;
1103  }
1104
1105
1106
1107  // Berechnet die Position des Codes innerhalb der Aufzählung
1108  int CCode::CalcPosition()
1109  {
1110    int pos = 0;
1111    for (int i=1;i<=count;i++)
1112    {
1113      pos += (1 << (word[i].getPosition()-1));
1114    }
1115    return pos;
1116  }
1117
1118
1119  float CCode::getAvgCodewordLength() const
1120  {
1121    float res = 0.0f;
1122    for (int i=1;i<=getCount();i++)
1123      {
1124        res+=(float)word[i].getLength();
1125      }
1126    return res/(float)getCount();
1127  }
1128
1129  // Vergleicht die, sich berschneidenden, Teile zweier Codewörter
1130  // Die 3. Variable gibt die Verschiebung des 2. Codewortes an
1131  bool CCode::Compare(const CCodeWord& codew1, const CCodeWord& codew2, int
        versch)
1132  {
1133    int start;
```

```
1134    int ende;
1135
1136    if (versch > 0)
1137      start = versch+1;
1138    else
1139      start = 1;
1140
1141    if (codew1.getLength() < (codew2.getLength()+versch))
1142      ende = codew1.getLength();
1143    else
1144      ende = codew2.getLength()+versch;
1145
1146    for (int i=start; i<=ende; i++)
1147      if (codew1.letter[i] != codew2.letter[(i-versch)])
1148      {
1149        return false;
1150      }
1151
1152    return true;
1153  }
1154
1155
1156  // Berechnet die relative Informationsrate des this-Codes
1157  float CCode::getInformationRate() const
1158  {
1159    return (log((float)getCount())/log(4.0f))/getAvgCodewordLength();
1160  }
1161
1162  // Berechnet die minimale relative Informationsrate für den aktuellen
        Codebaumzweig
1163  float CCode::getMinInfRateForBranch()
1164  {
1165    return (log((float)getCount())/log(4.0f))/(float)this->word[1].getLength()
          ;
1166  }
1167
1168  // Berechnet die maximale relative Informationsrate für den aktuellen
        Codebaumzweig.
1169  float CCode::getMaxInfRateForBranch()
1170  {
1171    return ((float)getCount()*(log((float)getCount())/log(4.0f)))/((float)
          getMinSumOfCodewordLenghts(getCount()-1)+(float)word[1].getLength());
1172  }
1173
1174  // Berechnet die maximale relative Informationsrate für den aktuellen
        Codebaumzweig bis zur
1175  // Tiefe maxlength
1176  float CCode::getMaxInfRateForBranch(int depth, int maxLength)
1177  {
```

```
1178    int s = 0;
1179    for (int i=1;i<=depth;i++)
1180      s+=word[i].getLength();
1181    s+=getMinSumOfCodewordLenghts(maxLength-depth);
1182    return ((float)maxLength*(log((float)maxLength)/log(4.0f)))/((float)s);
1183  }
1184
1185  // Prüft, ob im aktuellen Codebaumzweig die gewünschte relative
        Informationsrate
1186  // erreicht werden kann.
1187  //
1188  // Rückgabe:   1, falls die gewünschte relative Informationsrate erreicht
        werden kann.
1189  //           -1, falls die gewünschte relative Informationsrate nicht erreicht
         werden kann.
1190  int CCode::CheckInfRate()
1191  {
1192    if ((((minInfRate==-1.0)?getMaxInfRate(getCount()):minInfRate)<=
          getMaxInfRateForBranch()))
1193      return 1;
1194    return -1;
1195  }
1196
1197  // Prüft, ob im aktuellen Codebaumzweig die gewünschte relative
        Informationsrate
1198  // erreicht werden kann, wenn bis zur Tiefe maxLength gesucht werden würde
1199  //
1200  // Rückgabe:   1, falls die gewünschte relative Informationsrate erreicht
        werden kann.
1201  //           -1, falls die gewünschte relative Informationsrate nicht erreicht
         werden kann.
1202  int CCode::CheckInfRate(int tiefe,int maxLength)
1203  {
1204    float max = getMaxInfRateForBranch(0,maxLength);
1205    float r = getMaxInfRateForBranch(tiefe,maxLength);
1206
1207    if (r>max) return 0;
1208
1209    if ((((minInfRate==-1.0)?getMaxInfRate(maxLength):minInfRate)<=r))
1210      return 1;
1211    return 0;
1212  }
1213
1214  // Berechnet die maximale Informationsrate eines Codes für eine gegebene
        Codelänge
1215  float CCode::getMaxInfRate(int n)
1216  {
1217    float k = floor(log(0.75f*(float)n+1.0f)/log(4.0f));
1218    return (9.0f*(float)n*log((float)n))/(2.0f*log(2.0f)*(-16.0f*pow(4.0f,k)
```

```
        +16.0f+9.0f*k*(float)n+12.0f*k+9.0f*(float)n));
}

// Berechnet die minimale Summe der Codewortlänge für eine gegebene
    Codelänge
int CCode::getMinSumOfCodewordLenghts(int n)
{
  float k = floor(log(0.75f*(float)n+1.0f)/log(4.0f));
  return (int)(-(16.0/9.0)*pow(4.0f,k)+(16.0/9.0)+(float)n*k+(float)n
      +(4.0/3.0)*k);
}
```

Listing C.5: DNACode.cpp

```
// DNACode.cpp : Definiert den Einstiegspunkt fr die Konsolenanwendung.
//
#include "CCode.h"
#include <iomanip>
#ifdef WIN32
#include <conio.h>
#else
// #include <curses.h>
#include <stdlib.h>
#include <unistd.h>
//#include <conio.h>
#endif
#include <iostream>
#include <fstream>
#include <time.h>

using namespace std;

char *outfile = NULL;
ofstream *outstream = NULL;

void Interactive(CCode& c)
{
  bool done = false;
  while (!done)
  {
    std::cout << "(A)" << (c.nonOverlapping?" x ":"   ") << "theta-non-
        overlapping" << std::endl;
    std::cout << "(B)" << (c.thetaCompliant?" x ":"   ") << "theta-compliant
        " << std::endl;
    std::cout << "(C)" << ((c.thetaPrefixCompliant)?" x ":"   ") << "theta-p
        -compliant" << std::endl;
    std::cout << "(D)" << ((c.thetaSuffixCompliant)?" x ":"   ") << "theta-s
        -compliant" << std::endl;
    std::cout << "(E)" << ((c.thetaFree)?" x ":"   ") << "theta-comma-free"
        << std::endl;
```

```
32      std::cout << "(F)" << ((c.thetaStickyFree)?" x ":"   ") << "theta-sticky
            -free" << std::endl;
33      std::cout << "(G)" << ((c.thetaOverhangFree || (c.theta5OverhangFree &&
            c.theta3OverhangFree))?" x ":"   ") << "theta-overhang-free" << std
            ::endl;
34      std::cout << "(H)" << ((c.theta3OverhangFree)?" x ":"   ") << "theta-3'-
            overhang-free" << std::endl;
35      std::cout << "(I)" << ((c.theta5OverhangFree)?" x ":"   ") << "theta-5'-
            overhang-free" << std::endl;
36      std::cout << "(J)" << ((c.thetaSubwordCompliant)?" x ":"   ") << "theta(
            ";
37        if (c.thetaSubwordCompliant)
38          std::cout << c.k;
39        else
40          std::cout << "k";
41
42        std::cout << ", ";
43
44        if (c.thetaSubwordCompliant)
45          std::cout << c.m1;
46        else
47          std::cout << "m1";
48        std::cout << ", ";
49        if (c.thetaSubwordCompliant)
50          std::cout << c.m2;
51        else
52          std::cout << "m2";
53        std::cout << ")-subword-compliant" << std::endl;
54      std::cout << "(K)" << ((c.thetaCode)?" x ":"   ") << "theta-";
55      if (c.thetaCode)
56        std::cout << c.kCode;
57      else
58        std::cout << "k";
59      std::cout << "-code" << std::endl;
60      std::cout << "(L)" << ((c.nonOverlapping)?" x ":"   ") << "strict" <<
            std::endl;
61      std::cout << "(M)" << ((c.solid)?" x ":"   ") << "solid" << std::endl;
62      std::cout << "(N)" << ((c.commaFree)?" x ":"   ") << "comma-free" << std
            ::endl;
63      std::cout << "(O)" << ((c.restrictCodeWordLength)?" x ":"   ") << "
            restrict code word length";
64      if (c.restrictCodeWordLength)
65      {
66        std::cout << " to [" << c.minCodeWordLength << ", " << c.
              maxCodeWordLength << "]";
67      }
68      std::cout << std::endl;
69      std::cout << "(P)" << ((c.restrictCodeLength)?" x ":"   ") << "restrict
            code length";
```

```
70      if (c.restrictCodeLength)
71      {
72        std::cout << " to [" << c.minCodeLength << ", " << c.maxCodeLength <<
            "]";
73      }
74      std::cout << std::endl;
75      std::cout << "(Q)" << ((c.restrictCodes)?" x ":"   ") << "restrict
          number of codes";
76      if (c.restrictCodes)
77      {
78        std::cout << " to " << c.maxCodes;
79      }
80      std::cout << std::endl;
81      std::cout << "(R)" << ((c.infRate)?" x ":"   ") << "limit information
          rate";
82      if (c.infRate)
83      {
84        std::cout << " to at least " << c.minInfRate;
85      }
86      std::cout << std::endl;
87      std::cout << "(S)   Run" << std::endl;
88      std::cout << "(T)   Quit" << std::endl;
89      switch (tolower(getchar()))
90      {
91        case 'a': c.nonOverlapping = !c.nonOverlapping; break;
92        case 'b': { c.thetaCompliant = !c.thetaCompliant; }; break;
93        case 'c': { c.thetaPrefixCompliant = !c.thetaPrefixCompliant;  }break;
94        case 'd': { c.thetaSuffixCompliant = !c.thetaSuffixCompliant; }break;
95        case 'e': c.thetaFree = !c.thetaFree; break;
96        case 'f': c.thetaStickyFree = !c.thetaStickyFree; break;
97        case 'g': { c.thetaOverhangFree = !c.thetaOverhangFree; c.
            theta3OverhangFree = !c.theta3OverhangFree; c.theta5OverhangFree =
             !c.theta5OverhangFree; }; break;
98        case 'h': { c.theta3OverhangFree = !c.theta3OverhangFree; c.
            thetaOverhangFree = c.theta3OverhangFree && c.theta5OverhangFree;
            } break;
99        case 'i': { c.theta5OverhangFree = !c.theta5OverhangFree; c.
            thetaOverhangFree = c.theta3OverhangFree && c.theta5OverhangFree;
            }break;
100       case 'j':
101         {
102           c.thetaSubwordCompliant = !c.thetaSubwordCompliant;
103           if (c.thetaSubwordCompliant)
104           {
105             std::cout << " k,m1,m2 = ";
106             bool err = (scanf("%d,%d,%d",&c.k,&c.m1,&c.m2)!=3);
107             err |= ((c.k<=0) | (c.m1<1) | (c.m2<c.m1));
108             //unget
109             if (err)
```

```
110             {
111               std::cerr << "Invalid values." << std::endl <<
112                 "Enter values as 'k,m1,m2'." << std::endl <<
113                 "Note: k > 0, m1>0, m2>=m1" << std::endl;
114               c.thetaSubwordCompliant = false;
115               break;
116             };
117           }
118         } break;
119       case 'k': {
120           c.thetaCode = !c.thetaCode;
121           if (c.thetaCode)
122           {
123             std::cout << " k = ";
124             bool err = (scanf("%d",&c.kCode)!=1);
125             err |= (c.kCode<=0);
126             //unget
127             if (err)
128             {
129               std::cerr << "Invalid value." << std::endl <<
130                 "Note: k > 0" << std::endl;
131               c.thetaCode = false;
132               break;
133             };
134           }
135           }break;
136       case 'l': c.nonOverlapping= !c.nonOverlapping; break;
137       case 'm': c.solid= !c.solid; break;
138       case 'n': c.commaFree= !c.commaFree; break;
139       case 'o': {
140           c.restrictCodeWordLength = !c.restrictCodeWordLength;
141           if (c.restrictCodeWordLength)
142           {
143             std::cout << "Enter minimum and maximum code word length as '
                  min_length,max_length'." << std::endl;
144             std::cout << "l = ";
145             bool err = (scanf("%d,%d",&c.minCodeWordLength,&c.
                  maxCodeWordLength)!=2);
146             err |= (c.minCodeWordLength<=0) | (c.maxCodeWordLength<c.
                  minCodeWordLength);
147             //unget
148             if (err)
149             {
150               std::cerr << "Invalid values." << std::endl <<
151                 "Note: min_length > 0, max_length >= min_length" << std::
                      endl;
152               c.restrictCodeWordLength = false;
153               break;
154             };
```

```
155             }
156             }break;
157         case 'p': {
158             c.restrictCodeLength = !c.restrictCodeLength;
159             if (c.restrictCodeLength)
160             {
161               std::cout << "Enter minimum and maximum number of code word as '
                      min_count,max_count'." << std::endl;
162               std::cout << "c = ";
163               bool err = (scanf("%d,%d",&c.minCodeLength,&c.maxCodeLength)!=2)
                      ;
164               err |= (c.minCodeLength<=0) | (c.maxCodeLength<c.minCodeLength);
165               //unget
166               if (err)
167               {
168                 std::cerr << "Invalid values." << std::endl <<
169                   "Note: min_count > 0, max_count >= min_count" << std::endl;
170                 c.restrictCodeLength = false;
171                 break;
172               };
173             }
174             }break;
175         case 'q': {
176             c.restrictCodes = !c.restrictCodes;
177             if (c.restrictCodes)
178             {
179               std::cout << "Enter number of codes to be found at most." << std
                      ::endl;
180               std::cout << "n = ";
181               bool err = (scanf("%d",&c.maxCodes)!=1);
182               err |= (c.maxCodes<0);
183               //unget
184               if (err)
185               {
186                 std::cerr << "Invalid value." << std::endl <<
187                   "Note: n >= 0" << std::endl;
188                 c.restrictCodes = false;
189                 break;
190               };
191             }
192           }break;
193         case 'r': {
194             c.infRate = !c.infRate;
195             if (c.infRate)
196             {
197               std::cout << "Enter minimum information rate." << std::endl;
198               std::cout << "Enter -1 for optimal information rate." << std::
                      endl;
199               std::cout << "rate >= ";
```

```
200             bool err = (scanf("%f",&c.minInfRate)!=1);
201             err |= !((c.minInfRate==-1.0) || (c.minInfRate>=0.0));
202             //unget
203             if (err)
204             {
205               std::cerr << "Invalid value." << std::endl <<
206                 "Note: rate >= 0.0 or rate = -1" << std::endl;
207               c.infRate = false;
208               break;
209             };
210           }
211         }break;
212       case 's': done = true; break;
213       case 't': _exit(0);
214       default: break;
215     }
216     std::cout << std::endl;
217   }
218
219 }
220
221 int ParseCmdLine(int argc, char* argv[], CCode &c, int &n, char * &outfile)
222 {
223   bool interactive = false;
224   outfile = NULL;
225   for (int i=1;i<argc;i++)
226   {
227
228     if (strcmp(argv[i],"-i")==0)
229     {
230       std::cout << "entering interactive mode" << std::endl;
231       interactive = true;
232     } ;
233     if ((strstr(argv[i],"-o")==argv[i]))
234     {
235       if (!(i+1<argc))
236         {
237     std::cerr << "Missing file argument." << endl <<
238     "Usage: -o filename" << endl;
239     return -1;
240         }
241       outfile = argv[i+1];
242
243     }
244
245     if ((strstr(argv[i],"-l")==argv[i]))
246     {
247       bool err = (sscanf(argv[i],"-l = %d,%d",&c.minCodeWordLength,&c.
            maxCodeWordLength)!=2);
```

```
248         err |= ((c.minCodeWordLength<0) || (c.maxCodeWordLength<c.
                minCodeWordLength));
249         if (err)
250         {
251           std::cerr << "Invalid code word length specified." << std::endl <<
252             "Usage: -l = minLength,maxLength" << std::endl;
253           std::cerr << "(minLength <= maxLength)" << std::endl;
254           return -1;
255         }
256         c.restrictCodeWordLength = !err;
257       }
258       if ((strstr(argv[i],"-n")==argv[i]))
259       {
260         bool err = (sscanf(argv[i],"-n = %d,%d",&c.minCodeLength,&c.
                maxCodeLength)!=2);
261         err |= ((c.minCodeLength<0) || (c.maxCodeLength<c.minCodeLength));
262         if (err)
263         {
264           std::cerr << "Invalid code length specified." << std::endl <<
265             "Usage: -n = minLength,maxLength" << std::endl;
266           std::cerr << "(minLength <= maxLength)" << std::endl;
267           return -1;
268         }
269         c.restrictCodeLength = !err;
270       }
271       if (strstr(argv[i],"-count")==argv[i])
272       {
273         bool err = (sscanf(argv[i],"-count = %d",&n)!=1);
274         err |= (n<0);
275         if (err)
276         {
277           std::cerr << "Invalid number of codes." << std::endl <<
278             "Usage: -count = n" << std::endl;
279           std::cerr << "(n >= 0)" << std::endl;
280           return -1;
281         }
282         c.restrictCodes = !err;
283         c.maxCodes = n;
284       }
285       if (strstr(argv[i],"-rate")==argv[i])
286       {
287         bool err = (sscanf(argv[i],"-rate = %f",&c.minInfRate)!=1);
288         err |= (c.minInfRate<0.0f);
289         if (err)
290         {
291           std::cerr << "Invalid information rate." << std::endl <<
292             "Usage: -rate = value" << std::endl;
293           std::cerr << "(value >= 0.0)" << std::endl;
294           return -1;
```

```
295         }
296         c.infRate = !err;
297       }
298       if ((strcmp(argv[i],"-theta-non-overlapping")==0) ||
299         (strcmp(argv[i],"-strict")==0))
300       {
301         c.nonOverlapping = true;
302       }
303       if (strcmp(argv[i],"-theta-compliant")==0)
304       {
305         c.thetaCompliant = true;
306       }
307       if (strcmp(argv[i],"-solid")==0)
308       {
309         c.solid = true;
310       }
311       if (strcmp(argv[i],"-comma-free")==0)
312       {
313         c.commaFree = true;
314       }
315       if (strcmp(argv[i],"-theta-prefix-compliant")==0)
316       {
317         c.thetaPrefixCompliant = true;
318       }
319       if (strcmp(argv[i], "-theta-suffix-compliant")==0)
320       {
321         c.thetaSuffixCompliant = true;
322       }
323       if (strcmp(argv[i] ,"-theta-comma-free")==0)
324       {
325         c.thetaFree = true;
326       }
327       if (strcmp(argv[i], "-theta-sticky-free")==0)
328       {
329         c.thetaStickyFree = true;
330       }
331       if ((strcmp(argv[i],"-theta-3-prime-overhang-free")==0) ||
332         strcmp(argv[i], "-theta-overhang-free")==0)
333       {
334         c.theta3OverhangFree = true;
335       }
336       if ((strcmp(argv[i],"-theta-5-prime-overhang-free")==0) ||
337         (strcmp(argv[i], "-theta-overhang-free")==0))
338       {
339         c.theta5OverhangFree = true;
340       }
341       if ((strstr(argv[i],"-theta-subword-compliant")==argv[i]))
342       {
343         bool err = (sscanf(argv[i],"-theta-subword-compliant =%d,%d,%d",&c.k
```

```
            ,&c.m1,&c.m2)!=3);
344         err |= (c.m1>c.m2) || (c.m1<0) || (c.k<=0);
345         if (err)
346         {
347           std::cerr << "Invalid compliance parameters specified." << std::endl
                <<
348             "Usage: -theta-subword-compliant = k,m1,m2" << std::endl;
349           std::cerr << "(0 <= m1 <= m2 and k > 0)" << std::endl;
350           return -1;
351         }
352         c.thetaSubwordCompliant = !err;
353       }
354       if ((strstr(argv[i],"-theta-code")==argv[i]))
355       {
356         bool err = (sscanf(argv[i],"-theta-code = %d",&c.kCode)!=1);
357         err |= (c.kCode<=0);
358         if (err)
359         {
360           std::cerr << "Invalid parameters specified." << std::endl <<
361             "Usage: -theta-code = k" << std::endl;
362           std::cerr << "(k > 0)" << std::endl;
363
364           return -1;
365         }
366         c.thetaCode = !err;
367       }
368       if (strcmp(argv[i],"-strict")==0)
369       {
370         c.nonOverlapping= true;
371       }
372       // TODO: solid
373
374     }
375     return interactive;
376   }
377
378   void EnumerateCode(void *code,int number, int flag)
379   {
380     if (flag==1)
381       {
382         cout << "No more codes." << endl;
383       } else {
384       CCode *c = (CCode *)code;
385     cout << " | " << setw(6) << number << " | " << setw(6) << c->getCount() <<
          " | " << setprecision(4) << setw(8) << c->getAvgCodewordLength() << "
          | " << setprecision(4) << setw(8) << c->getInformationRate() << " | "
          << *c <<  endl;
386       if (outstream!=NULL)
387         *outstream << number << "," << c->getCount() << "," << c->
```

```
            getAvgCodewordLength() << "," << "," << c->getInformationRate() <<
             "," << *c << endl;
388     }
389   }
390
391
392   int main(int argc, char* argv[])
393   {
394     CCode c;
395     int n = 0;
396
397
398     int interactive = ParseCmdLine(argc,argv,c,n,outfile);
399
400     if (outfile!=NULL)
401     {
402         cout << "Dumping to " << outfile <<"." << endl;
403     }
404
405
406     if (interactive == 1)
407       Interactive(c);
408
409     if (outfile!=NULL)
410     {
411        try
412        {
413       outstream = new ofstream(outfile,ios::out);
414        }
415        catch(...)
416        {
417       cerr << "Error creating " << outfile << "." << endl;
418       return -1;
419        }
420     }
421
422     cout << "?---------?---------?----------?----------?" <<  endl;
423     cout << "| ####### | Length | Avg.cw len | Inf.rate |" <<  endl;
424     cout << "?---------?---------?----------?----------?" <<  endl;
425
426     if (outstream!=NULL)
427         *outstream << ";number,length,avgCodeWordLength,infRate,code" << endl;
428
429     clock_t start,ende;
430     start = clock();
431     int res = c.StartRecursion(EnumerateCode);
432     ende = clock();
433     cout << "?---------?---------?----------?----------?" <<  endl;
434     if (res<0)
```

```
435       cout << "No more codes." << endl;
436     else
437       cout << "Done." << endl;
438     if (outstream!=NULL)
439       {
440         outstream->flush();
441         outstream->close();
442         delete outstream;
443       }
444     std::cout << "Running time: " << ((double)(ende-start))/CLOCKS_PER_SEC << " s" << std::endl;
445     //getch();
446
447
448     return 0;
449   }
```

LITERATURVERZEICHNIS

[Adl94] Adleman, Leonard M.: *Molecular Computation of Solutions to Combinatorial Problems.* Science, 266:1021–1024, 11, 1994.

[FH06] Freeland, S. J. und L. D. Hurst: *Der raffinierte Code des Lebens.* Spektrum der Wissenschaft Dossier, 1:18–25, 2006.

[Her05] Herrmann, Christian: *Entwicklung von Methoden zur Aufzählung und Untersuchung von soliden Codes, Hyper-Codes und soliden Hyper-Codes.* Diplomarbeit, Universität Potsdam, 2005.

[HKK03] Hussini, S., L. Kari und S. Konstantinidis: *Coding properties of DNA languages.* Theoretical Computer Science, 290/3:1557–1579, 2003.

[KE00] Kathrin Erk, Lutz Priese: *Theoretische Informatik.* Springer, Januar 2000.

[KKLW03] Kari, L., S. Konstantinidis, E. Losseva und G. Wozniak: *Sticky-free and overhang-free DNA languages.* Acta Informatica, 40:119–157, 2003.

[KKT00] Kari, L., R. Kitto und G. Thierrin: *Codes, involutions and DNA encoding. In W. Brauer, H. Ehrig, J. Karhumäki, A. Salomaa (eds.).* Lecture Notes in Computer Science, 2300:376–393, 2000.

[Shy01] Shyr, H. J.: *Free Monoids and Languages.* Hon Min Book Company, 2001.

Eidesstattliche Erklärung

Ich versichere, dass ich die Diplomarbeit selbständig ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertig habe und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat oder veröffentlicht wurde. Alle Ausführungen der Arbeit, die wörtlich oder sinngemäß übernommen wurden, wurden als solche gekennzeichnet.

Werder/Havel, 24. Juli 2006