Verwendung von Überdeckungscodes zur Lösung von SAT–Problemen

Diplomarbeit im Fach Mathematik

am Institut für
Numerische und Angewandte Mathematik
der Mathematischen Fakultät
der Georg-August-Universität
zu Göttingen

von Percy–Constantin von Samson–Himmelstjerna

1. Referent:	Prof. Dr. C. Damm
2. Referent:	Prof. Dr. S. Waack
eingereicht am:	03. Mai 2007

Hiermit erkläre ich, diese Arbeit eigenständig erstellt und nur die angegebenen Quellen verwendet zu haben.

Göttingen, den 03. Mai 2007 _____

Kurzdarstellung

Evgeny Dantsin, Andreas Goerdt, Edward A. Hirsch *et al.* veröffentlichten 2002 in der Theoretical Computer Science 289/1 einen Artikel [4] mit dem Titel „A Deterministic $\left(2 - \frac{2}{k+1}\right)^n$ Algorithm for k–SAT Based on Local Search“[1]. Der dort vorgestellte Algorithmus löst (binäre) Erfüllbarkeitsprobleme deterministisch, indem er eine Überdeckung des Suchraums erstellt und in den einzelnen Mengen der Überdeckung lokal nach der Lösung des Problems sucht. Als gute Möglichkeit den $\mathbb{F}_2^n$ zu überdecken bieten sich Überdeckungscodes über einem binären Alphabet an.

In dieser Arbeit wird gezeigt, dass dieser Algorithmus auch für q–näre Alphabete geeignet und auf mehrwertige Logik anwendbar ist. Sowohl die vorgestellten Möglichkeiten einen Überdeckungscode zu bestimmen als auch die Methode der lokalen Suche sind ein Weg, deterministisch mehrwertige Erfüllbarkeitsprobleme zu lösen. Darüber hinaus werden Möglichkeiten zur Verbesserung der Überdeckungseigenschaft eines gegebenen Codes erläutert. Außerdem wurde der Algorithmus für binäre k–SAT Probleme implementiert und getestet.

Danksagungen

Bedanken möchte ich mich an dieser Stelle bei Professor Dr. C. Damm für seine konstruktive Kritik und insbesondere für seine Anregungen bei der Bearbeitung des Themas. Dank gebührt auch Herrn Professor Dr. St. Waack dafür, dass er sich als Korreferent zur Verfügung gestellt hat.
Auch bei den vielen „kleinen Helferlein“, die während der Phase der Programmierung immer wieder ein zweites Paar Augen hatten, möchte ich mich bedanken, allen voran Frau Dipl.–Inf. Edith Werner vom Institut für Informatik, die mir die Verwendung von `valgrind` bei brachte.

Ein besonderer Dank richtet sich an meine Frau, die mir in dieser Zeit stets eine Stütze war und den nötigen Freiraum gegeben hat, um diese Arbeit zu erstellen.

[1] Informationen zu diesem Artikel, auch zu seiner Geschichte, finden sich auf der Homepage von Edward A. Hirsch unter `http://logic.pdmi.ras.ru/~hirsch/`

Inhaltsverzeichnis

Kapitel 1

Einleitung

Es gibt schon mehrere Algorithmen um Probleme in mehrwertiger Logik randomisiert zu lösen, bzw. deren Unerfüllbarkeit zu zeigen. Evgeny Dantsin, Andreas Goerdt, Edward A. Hirsch *et al.* veröffentlichten 2002 in der Theoretical Computer Science 289/1 einen Artikel [4], in dem ein Algorithmus vorgestellt wurde, der deterministisch Probleme in Bool'sche Algebra löst. Das Schöne an diesem Algorithmus ist, dass er trotz deterministischer Vorgehensweise schneller als die so genannte „vollständige Suche" — Laufzeit $\mathcal{O}(q^n)$ bei einem q–nären Alphabet — ist. Dieser Algorithmus und die dafür nötigen codierungstheoretischen Grundlagen sollen im Rahmen dieser Arbeit von Bool'scher Logik auf mehrwertige Logik übertragen und erweitert werden.

Zuerst erfolgt eine Einführung in die mehrwertige Logik mit anschließender Betrachtung der Überdeckungscodes und Möglichkeiten ihrer Konstruktion. Im folgenden Kapitel wird der „Greedy Algorithmus" für die Bestimmung eines Überdeckungscodes vorgestellt. Anschließend wird der Algorithmus von Dantsin, Goerdt, Hirsch *et al.* auf mehrwertige Logik übertragen und gezeigt, dass er auch diese lösen kann, sowie seine Laufzeit bestimmt. Die für eine Implementation benötigten Formate werden erläutert, gefolgt von der Beschreibung der Implementation. Schließlich erfolgt die Auswertung der Testdurchläufe.

Der Arbeit beigefügt ist eine CD–ROM, die die Arbeit im pdf–Format, die Quellcodedateien, die für die Programmtests verwendeten Shell Skripte einschließlich Formeldateien[1] und die Auswertung der Programmtests enthält.

[1] siehe `http://www.cs.ubc.ca/~hoos/SATLIB/benchm.html`

Kapitel 2

Grundlagen

2.1 Einführung in mv–SAT

Schon seit Ende der 80er Jahre ist die mehrwertige Logik (*engl. multivalued logic, Abkürzung mv–SAT*) bekannt und wird erforscht. Zahlreiche Anwendungsmöglichkeiten gibt es im Bereich der „Electronic Design Automation“ (EDA) (zum Beispiel *Model–Checking*) oder bei SAT–Problemen mit einer hohen Anzahl an Variablen, wie beispielsweise dem „Taubenschlag–Problem“.

Einer der größten Vorteile der mehrwertigen Logik ist, dass sich bei geeigneter Basiswahl die Anzahl der benötigten Variablen logarithmisch verringert. Betrachtet man die Menge an Information, die in einem Byte abgespeichert werden kann, so werden acht Variablen zur Basis 2 benötigt. Wird eine größere Basis gewählt, so verringert sich die Zahl der benötigten Variablen nach folgender Formel. Dabei sei n die Zahl der Bool'schen Variablen, q die Basis der mehrwertigen Logik und m die Zahl der mehrwertigen Variablen:

$$\lceil n \log_q 2 \rceil = m.$$

Wie entsteht diese Formel? Man will 2^n durch q^m möglichst genau annähern, wobei $2^n \leq q^m$ gilt, da andernfalls ein Informationsverlust auftrete. Außerdem soll m so klein wie möglich sein. Folglich wird auf

$$2^n \leq q^m$$

der Logarithmus zur Basis q und die obere Gaußklammer angewendet.

Algebraisch betrachtet ergeben sich keine großen Unterschiede zwischen mehrwertiger Logik und Bool'scher Logik. Die Grundoperationen AND, OR und XOR werden auf dieselbe Art und Weise angewendet. Auch die Negation ist möglich. Dies wird ab Lemma 2.10 noch näher ausgeführt.

2.2 Grundlegende Definitionen

Definition 2.1 Sei $\mathcal{Q}$ ein Alphabet mit q Elementen. Dann bezeichnet $x_i \in \mathcal{X}$ eine mehrwertige Variable (*multivalued variable*) mit Werten aus $\mathcal{Q}$. Mit $\mathcal{P}(\mathcal{Q})$ sei die *Teilmenge der Potenzmenge* von $\mathcal{Q}$ bezeichnet, für die gilt:

$$\forall\ v \subseteq \mathcal{P}(\mathcal{Q}) :\ \#(v) \leq q.$$

Definition 2.2 Eine Zuweisung einer Wertemenge zu einer mehrwertigen Variable nennt man mehrwertiges Literal (*multivalued literal*). Dadurch wird der möglichen Wertebereich der Variable eingeschränkt.
Eine Wertemenge $v_i \in \mathcal{P}(\mathcal{Q})$ mit Elementen aus $\mathcal{Q}$ heißt zu einer Variable x_i zugeordnet, wenn x_i als Werte alle Elemente von v_i annehmen kann, jedoch keine Werte aus $\mathcal{Q} \setminus v_i$.
Als Schreibweise für ein Literal ist $x_i^{v_i}$ üblich, mit $\mathcal{L}(\mathcal{X})$ wird die Menge der Literale über $\mathcal{X}$ bezeichnet.

Definition 2.3 Gilt $\#(v_i) = 1$, so wird die Zuweisung als *vollständig* (*complete*) bezeichnet, andernfalls als *unvollständig* (*incomplete*). Sind alle Zuweisungen der Variablen $x_i \in \mathcal{X}$ vollständig, so wird $\mathcal{X}$ als *vollständig zugewiesen* (*full assignment*) bezeichnet, andernfalls als *unvollständig zugewiesen* (*partial assignment*).

Definition 2.4 Die Auswertung eines mehrwertigen Literals ist als Bool'sche Funktion definiert:

$$x_i^{v_i} \equiv (x_i = \sigma_1) \vee \cdots \vee (x_i = \sigma_k).$$

Dabei sind die $\sigma_j \in v_i \subseteq \mathcal{P}(\mathcal{Q}),\ j = 1, 2, \ldots, k.$ v_i bezeichnet die *Literal–Wertemenge.*

Bemerkung 2.5 $x_1^{\{1,4\}}$ hat den Wert „`true`“, falls x_1 den Wert 1 oder 4 annimmt. Offensichtlich gilt:

$$x_i^{\emptyset} \equiv 0, \quad x_i^{\mathcal{Q}} \equiv 1.$$

Definition 2.6 Unter einer *mehrwertigen Logik–Funktion (multivalued logic function)* versteht man eine logische Funktion, deren Argumentbereich mehrwertige Variablen sind und deren Wertebereich aus $\mathcal{Q}$ ist:

$$f :\ \mathcal{L}(\mathcal{X}) \longrightarrow \mathcal{Q}$$

Ist der Wertebereich einer mehrwertigen Logik–Funktion $\{0, 1\}$, so wird von einer *mehrwertigen Funktion (multivalued function)* gesprochen:

$$f :\ \mathcal{L}(\mathcal{X}) \longrightarrow \{0, 1\}$$

Definition 2.7 Die Auswertung eines mehrwertigen Literals $x_i^{v_i}$ eingeschränkt auf eine Wertemenge w_j zu x_i ist eine mehrwertige Funktion. Sie ist wie folgt definiert:

$$f(x_i^{v_i})_{|w_j} : \mathcal{L}(\mathcal{X}) \longrightarrow \{0,1\} \cup \mathcal{P}(\mathcal{Q})$$

$$f(x_i^{v_i})_{|w_j} = \begin{cases} 1 \ , \ w_j \cap v_i = w_j & (i) \\ 0 \ , \ w_j \cap v_i = \emptyset & (ii) \\ y_1 \ , \ w_j \cap v_i = v_i & (iii) \\ y_2 \ , \ \text{sonst} & (iv) \end{cases}$$

Dabei bedeutet $f = 0$ „`false`“ und $f = 1$ „`true`“. $f = y_1$ und $f = y_2$ bezeichnen einen unbekannten Literalwert, der gewöhnlich speziell behandelt werden muss.

Bemerkung 2.8 Die Unterscheidung zwischen den letzten beiden Fällen obiger Definition liefert einige wertvolle Einblicke in die Beziehung der Mengen w_j und v_i. Die Auswertung und Verwendung wurde in [1] beschrieben.

Lemma 2.9 Ist eine mehrwertige Variable vollständig zugewiesen, so ist die Auswertung des zugeordneten mehrwertigen Literals eindeutig bestimmt und hat die Werte „`true`“ oder „`false`“.

Beweis: Ist einer mehrwertigen Variable ein spezieller Wert zugeordnet, dann enthält das zugehörige mehrwertige Literal entweder den Wert in seiner Wertemenge oder nicht. Somit wird entweder Fall (i) oder Fall (ii) aus Definition 2.7 erfüllt. □

Lemma 2.10 Die Negation eines mehrwertigen Literals ist die Bildung des Komplements seiner zugewiesenen Wertemenge:

$$\neg x_i^{v_i} = x_i^{\mathcal{Q} \setminus v_i}.$$

Beweis: Nach 2.4 gilt:

$$x_i^{v_i} = 1 \ \forall \ \sigma \in v_i.$$

Ebenso gilt:

$$x_i^{v_i} = 0 \ \forall \ \hat{\sigma} \notin v_i$$

und es ist $\{\hat{\sigma} \mid \hat{\sigma} \notin v_i\} = \mathcal{Q} \setminus v_i$. Bildet man die Negation von $x_i^{v_i} = 1$, ergibt sich $\neg x_i^{v_i} = 0$ und mit Definition 2.7 gilt: $(x_{i \mid \mathcal{Q} \setminus v_i}^{v_i}) = 0 = \neg x_i^{v_i}$. □

Die nächsten drei Definitionen behandeln die Verknüpfung von mehrwertigen Variablen mit den Operatoren AND, OR und XOR. Die Verknüpfung wird analog einer mehrwertigen logischen Funktion ausgewertet.

Definition 2.11 Der AND–Operator, auch *logische Konjunktion* zweier mehrwertiger Literale genannt, ist wie folgt definiert:

$$x_1^{v_1} \text{ AND } x_2^{v_2} = x^{v_1 \cap v_2}.$$

Sind x_1 und x_2 Werte zugewiesen, berechnet sich der AND–Operator so:

$$x_1^{v_1} \text{ AND } x_2^{v_2} \quad = \quad \begin{cases} 1 \ , \ x_1^{v_1} = 1 \text{ und } \quad x_2^{v_2} = 1 \\ 0 \ , \ \text{sonst} \end{cases}$$

Definition 2.12 Der OR–Operator, auch *logische Disjunktion* zweier mehrwertiger Literale genannt, ist wie folgt definiert:

$$x_1^{v_1} \text{ OR } x_2^{v_2} = x^{v_1 \cup v_2}.$$

Sind x_1 und x_2 Werte zugewiesen, berechnet sich der OR–Operator so:

$$x_1^{v_1} \text{ OR } x_2^{v_2} \quad = \quad \begin{cases} 1\ , & x_1^{v_1} = 1 \text{ und} \quad x_2^{v_2} = 1 \\ 1\ , & x_1^{v_1} = 1 \text{ oder} \quad x_2^{v_2} = 1 \\ 0\ , & x_1^{v_1} = 0 \text{ und} \quad x_2^{v_2} = 0 \end{cases}$$

Definition 2.13 Der XOR–Operator ist wie folgt definiert:

$$\begin{aligned} x_1^{v_1} \text{ XOR } x_2^{v_2} &= (x_1^{v_1} \text{ AND } \neg x_2^{v_2}) \text{ OR } (\neg x_1^{v_1} \text{ AND } x_2^{v_2}) \\ &= \begin{cases} 1\ , & x_1^{v_1} = 1 \text{ und} \quad x_2^{v_2} = 0 \\ 1\ , & x_1^{v_1} = 0 \text{ und} \quad x_2^{v_2} = 1 \\ 0\ , & \text{sonst} \end{cases} \end{aligned}$$

Im weiteren werden die auch üblichen Schreibweisen $\wedge$ für AND, $\vee$ für OR und $\oplus$ für XOR benutzt.

Definition 2.14 Eine *mehrwertige Klausel (multivalued clause)* ist eine logische Disjunktion eines oder mehrerer mehrwertiger Literale:

$$C = \sum_i x_i^{v_i}$$

Beispiel:

$$\mathcal{Q} = \{0, 1, 2, 3, 4, 5\}, \qquad C = (x_1^{\{0,2,4\}} \vee x_2^{\{1,3\}} \vee x_3^{\{5\}}).$$

Lemma 2.15 Eine mehrwertige Klausel nimmt den Wert „`true`“ an, wenn mindestens eines der mehrwertigen Literale den Wert „`true`“ annimmt. Sie nimmt den Wert „`false`“ an, wenn alle mehrwertigen Literale den Wert „`false`“ annehmen.

Beweis: folgt aus Definition 2.14. □

Definition 2.16 Eine Formel in *mehrwertiger konjunktiver Normalform (mv–CNF)* ist die logische Konjunktion einer Menge von mehrwertigen Klauseln.

Im Folgenden soll stets davon ausgegangen werden, dass alle Formeln in mv–CNF gegeben sind.

Lemma 2.17 Eine mv–CNF–Formel nimmt den Wert „`true`“ an, wenn alle mehrwertigen Klauseln den Wert „`true`“ annehmen.

Beweis: folgt aus Definition 2.16. □

Definition 2.18 Ein mv–SAT–Problem heißt *erfüllbar*, wenn eine *vollständige Zuweisung* existiert, für die die mv–CNF–Formel den Wert „`true`“ annimmt. Die zugehörige vollständige Zuweisung wird als *Lösung* des Problems bezeichnet. Das Problem heißt *unerfüllbar*, wenn es keine solche Lösung gibt.

Kapitel 3

Konstruktion von Überdeckungscodes über $\mathcal{Q}^n$

3.1 Grundlagen

Sei $\mathcal{Q}$ ein Alphabet mit q Elementen. Folglich hat der n–dimensionale Raum $\mathcal{Q}^n$ q^n Elemente. Diese haben beispielsweise folgendes Aussehen:

$$\sigma = \sigma_1\sigma_2\sigma_3 \ldots \sigma_n$$

Definition 3.1 Eine Überdeckung des $\mathcal{Q}^n$ ist eine Menge von Untermengen $\mathcal{U}_i \subset \mathcal{Q}^n$, für die gilt:

$$\bigcup_{\forall i} \mathcal{U}_i \supseteq \mathcal{Q}^n$$

Eine Überdeckung des $\mathcal{Q}^n$ kann durch

$$\mathcal{C}' = \{B_r(a) \mid B_r(a) = \{b \in \mathcal{Q}^n | \; d_H(a,b) \leq r, \; a \in \mathcal{Q}^n \text{ fest, } r\text{–Radius}\}\}$$

gebildet werden. Dabei entspricht $d_H(a,b)$ der folgenden

Definition 3.2 Seien $a, b \in \mathcal{Q}^n$. Dann bezeichnet $d_H(a,b) = \#\{i \in \mathbb{N} \mid a_i \neq b_i\}$ die *Hamming–Distanz* zwischen a und b.

Definition 3.3 Sei $\mathcal{C}$ eine Menge von Wörtern aus dem $\mathcal{Q}^n$. Dann ist

$$\min\{d_H(a_1,a_2) \text{ mit } a_1, a_2 \in \mathcal{C}, \; a_1 \neq a_2\} \tag{3.1}$$

die *Minimaldistanz* von $\mathcal{C}$.

In Definition 3.1 eingeführten Mengen $B_r(a)$ werden als *Kugeln* oder *Bälle* bezeichnet. Je nach Kontext werden dabei beide Bezeichnungen verwendet — Kugeln im Zusammenhang mit der Anzahl der enthaltenen Wörter, Bälle, wenn die Menge an Wörtern als solche gemeint ist.

Lemma 3.4 Sei $V_q^n(a,r) = \#\{b \in \mathcal{Q}^n \mid d_H(a,b) \leq r\}$ das Volumen der einzelnen Kugeln um die ausgezeichneten Elemente a. Dann gilt:

$$V_q^n(a,r) = \sum_{i=0}^{r} \binom{n}{i} (q-1)^i \tag{3.2}$$

Definition 3.5 Die q–näre Entropiefunktion $H_q(\rho)$ ist definiert als:

$$\begin{aligned} H_q &: \{0\} \cup (0; 1 - \frac{1}{q}] \longrightarrow [0;1] \\ H_q(0) &:= 0 \\ H_q(\rho) &:= \rho \log_q \frac{q-1}{\rho} + (1-\rho) \log_q \frac{1}{1-\rho} \end{aligned}$$

q bezeichnet dabei die Größe der Grundmenge.

Lemma 3.6 Sei $0 \leq \rho \leq 1 - \frac{1}{q},\ r = \lfloor \rho * n \rfloor \in \mathbb{N}$. Dann kann $V_q(a,r)$ abgeschätzt werden durch:

$$\begin{aligned} V_q^n(a,r) &\approx q^{n(\rho \log_q \frac{q-1}{\rho} + (1-\rho)\log_q \frac{1}{1-\rho})} \\ &= q^{n \cdot H_q(\rho)} \end{aligned} \tag{3.3}$$

Beweis: Da $r = \lfloor \rho n \rfloor$ gilt, ist der letzte Term der rechten Seite von (3.2) der größte. Somit gilt:

$$\binom{n}{\rho n}(q-1)^{\rho n} \ \leq\ V_q^n(a,r)\ \leq\ (1+\rho n)\binom{n}{\rho n}(q-1)^{\rho n} \tag{3.4}$$

Dabei ist das rechte Ungleichheitszeichen erfüllt, da die Summe, mit der das Volumen berechnet wird, $(1+\rho n)$ Terme hat, von denen jeder kleiner ist als $\binom{n}{\rho n}(q-1)^{\rho n}$; das linke Ungleichheitszeichen ist erfüllt, da $\binom{n}{\rho n}(q-1)^{\rho n}$ lediglich der letzte Term der Summe ist.

Wende nun auf (3.4) $\log_q$ an:

$$\begin{aligned} &\log_q \tbinom{n}{\rho n} + \rho n \log_q (q-1) \\ &\quad\leq\ \log_q V_q^n(a,r) \\ &\leq \log_q \Big((1+\rho n) \tbinom{n}{\rho n} (q-1)^{\rho n} \Big) \\ &\quad\Longleftrightarrow \\ &\log_q \tbinom{n}{\rho n} + \rho n \log_q (q-1) \\ &\quad\leq\ \log_q V_q^n(a,r) \\ &\leq \log_q \rho n + \log_q \tbinom{n}{\rho n} + \rho n \log_q (q-1) \end{aligned} \tag{3.5}$$

Nachfolgend wird

$$\begin{aligned}
\binom{n}{m} &\leq \frac{n^n}{m^m(n-m)^{n-m}} \\
&\iff \\
n^n &= (m+(n-m))^n \\
&= \sum_{i=0}^{n} \binom{n}{i} m^i (n-m)^{n-i} \\
&\geq \binom{n}{m} m^m (n-m)^{n-m}
\end{aligned}$$

benötigt und in (3.5) eingesetzt:

$$\begin{gathered}
n\log_q n - \rho n \log_q \rho n - n(1-\rho)\log_q\Big(n(1-\rho)\Big) + \rho n \log_q(q-1) \\
\leq \\
\log_q V_q^n(a, \rho n) \\
\leq \\
\log_q \rho n + n\log_q n - \rho n \log_q \rho n - n(1-\rho)\log_q\Big(n(1-\rho)\Big) + \rho n \log_q(q-1)
\end{gathered}$$

Mit n^{-1} multiplizieren

$$\begin{aligned}
&\Longrightarrow \\
&\quad \log_q n - \Big(\rho \log_q n + (1-\rho)\log_q n\Big) - \rho\log_q\rho - \\
&\qquad (1-\rho)\log_q(1-\rho) + \rho\log_q(q-1) \\
&\qquad \leq \\
&\quad n^{-1}\log_q V_q^n(a,\rho n) \\
&\qquad \leq \\
&\quad n^{-1}\log_q \rho n + \log_q n - \Big(\rho\log_q n + (1-\rho)\log_q n\Big) - \rho\log_q\rho - \\
&\qquad (1-\rho)\log_q(1-\rho) + \rho\log_q(q-1) \\
&\Longleftrightarrow \\
&\quad \rho\log_q\frac{q-1}{\rho} + (1-\rho)\log_q\frac{1}{1-\rho} \\
&\qquad \leq \\
&\quad n^{-1}\log_q V_q^n(a,\rho n) \\
&\qquad \leq \\
&\quad n^{-1}\log_q \rho n + \rho\log_q\frac{q-1}{\rho} + (1-\rho)\log_q\frac{1}{1-\rho}
\end{aligned}$$

Somit gilt:

$$V_q^n(a,r) \approx q^{nH_q(\rho)} \tag{3.6}$$

□

3.2 Codekonstruktionen

Nun soll gezeigt werden, dass man mit zufällig gewählten Codes bzw. Codewörtern — im Folgenden soll die Bezeichnung „randomisierte Codes“ verwendet werden — den $\mathcal{Q}^n$ überdecken kann. Dabei bildet jedes Codewort a den Mittelpunkt eines Balls mit Radius r und Kugelvolumen $V_q^n(a,r)$ wie im obigen Abschnitt.

Lemma 3.7 Für jedes $n \geq 1$ und jedes r existiert ein randomisierter Überdekkungscode $\mathcal{C}'$ mit Länge n und Überdeckungsradius r mit Größe höchstens

$$\left\lceil \frac{n \cdot q^n}{V_q^n(a,r)} \right\rceil \tag{3.7}$$

Beweis: Wähle zufällig mit Zurücklegen $\frac{n \cdot q^n}{V_q^n(a,r)} = \#\mathcal{C}'$ Elemente.
zz: $\mathcal{C}'$ bildet mit hoher Wahrscheinlichkeit einen Überdeckungscode mit den gewünschten Eigenschaften:
Wähle $b \in \mathcal{Q}^n$ fest, $a \in \mathcal{C}'$ beliebig. Dann hat „$b \in B_r(a)$“ die Wahrscheinlichkeit

$$\frac{V_q^n(b,r)}{q^n}$$

„$b \notin B_r(a)$“ hat Wahrscheinlichkeit

$$1 - \frac{V_q^n(a,r)}{q^n}$$

und „$b \notin B_r(a)\ \forall\ a \in \mathcal{C}'$“ hat Wahrscheinlichkeit

$$\left(1 - \frac{V_q^n(a,r)}{q^n}\right)^{\#\mathcal{C}'}$$

Benutze nun $(1 - \frac{1}{t})^{mt} \overset{t\to\infty}{\leq} e^{-m}$:

$$\rightsquigarrow \left(1 - \frac{V_q^n(a,r)}{q^n}\right)^{\frac{n \cdot q^n}{V_q^n(a,r)}} \leq e^{-n} \overset{n\to\infty}{\longrightarrow} 0 \tag{3.8}$$

Somit geht die Wahrscheinlichkeit dafür, dass mit der zufällig gewählten Überdekkung kein Überdeckungscode gebildet wird, gegen 0. □

Für die nächste Folgerung muss erst Lemma 4.7.1 aus [5] auf den q–nären Fall verallgemeinert werden:

Lemma 3.8 Sei $0 < \rho < 1$ und $\rho n \in \mathbb{N}$, dann gilt:

$$\sqrt{8n\rho(1-\rho)}^{-1} \cdot q^{nH_q(\rho)} \leq \binom{n}{\rho n}(q-1)^{\rho n} \leq \sqrt{2\pi n\rho(1-\rho)}^{-1} \cdot q^{nH_q(\rho)}$$

Beweis: Nach der *Stirlingschen Formel* gilt:

$$n! = \sqrt{2\pi n}\ n^n e^{-n} \exp\left(\frac{1}{12n} - \frac{1}{360n^3} + \dots\right)$$

Wird dabei bei exp kein Term bzw. der 2., 4., usw. genommen, wird $n!$ unterschätzt, wird der 1., 3., usw. genommen, überschätzt:

$$\binom{n}{\rho n} = \frac{n!}{(\rho n)!(n\cdot(1-\rho))!} \geq \frac{\sqrt{2\pi n}\; n^n e^{-n}\; \exp\left(-\frac{1}{12\rho n} - \frac{1}{12n(1-\rho)}\right)}{\sqrt{2\pi\rho n}\;(\rho n)^{\rho n}e^{-\rho n}\;\sqrt{2\pi n(1-\rho)}\;(n(1-\rho))^{n(1-\rho)}e^{-n(1-\rho)}}$$

Dabei ist $n!$ über- und $(\rho n)!$ sowie $(n(1-\rho)!$ unterschätzt worden.
Annahme: $\rho n \geq 1$ und $n(1-\rho) \geq 3$
Aufgrund der Symmetrie in ρ und $1-\rho$ von

$$\sqrt{8n\rho(1-\rho)}^{-1} q^{nH_q(\rho)} \leq \binom{n}{\rho n}(q-1)^{\rho n} \leq \sqrt{2\pi n\rho(1-\rho)}^{-1} q^{nH_q(\rho)}$$

bleiben nur die Fälle $\rho n = 1$, $n(1-\rho) = 1$; $\rho n = 1$, $n(1-\rho) = 2$ und $\rho n = 2$, $n(1-\rho) = 2$ gesondert zu betrachten, was direkt durch Substitution geschehen kann.

Dann gilt:

$$\frac{1}{12\rho n} + \frac{1}{12n(1-\rho)} \leq \frac{1}{12} + \frac{1}{36} = \frac{1}{9}$$

und

$$\exp\left(-\frac{1}{12\rho n} - \frac{1}{12n(1-\rho)}\right) \geq e^{-\frac{1}{9}} \approx 0,895 \geq \frac{1}{2}\sqrt{\pi}$$

Somit ist

$$\begin{aligned}
\binom{n}{\rho n}(q-1)^{\rho n} &\geq (8n\rho(1-\rho))^{-\frac{1}{2}}\rho^{-\rho n}(1-\rho)^{-n(1-\rho)}(q-1)^{\rho n} \\
&= (8n\rho(1-\rho))^{-\frac{1}{2}}\;(q-1)^{\rho n}\;q^{-n(\rho\log_q\rho+(1-\rho)\log_q(1-\rho))} \\
&= (8n\rho(1-\rho))^{-\frac{1}{2}}\;q^{n(\rho\log_q\frac{q-1}{\rho}+(1-\rho)\log_q\frac{1}{1-\rho})} \qquad (3.9)
\end{aligned}$$

Für die rechte Ungleichheit wird wie folgt $n!$ überschätzt und $(\rho n)!$, $(n(1-\rho))!$ unterschätzt:

$$\binom{n}{\rho n}(q-1)^{\rho n} \leq \frac{\sqrt{2\pi n}\,n^n\,e^{-n}\,e^{\frac{1}{12n}}\exp\left(\frac{1}{12\rho n} - \frac{1}{360(\rho n)^3} + \frac{1}{12n(1-\rho)} - \frac{1}{360(n(1-\rho))^3}\right)}{\sqrt{2\pi\rho n}\,(\rho n)^{\rho n}e^{-\rho n}\;\sqrt{2\pi n(1-\rho)}\,(n(1-\rho))^{n(1-\rho)}e^{-n(1-\rho)}}$$

Da $\rho n \geq 1$ gilt, ist $\frac{1}{360(\rho n)^3} \leq \frac{1}{360\rho n}$ und für $(1-\rho)$ analog. Außerdem sei $\rho \geq 1-\rho$.
Somit ist

$$\begin{aligned}
&\frac{1}{12n} - \frac{1}{12\rho n} - \frac{1}{12n(1-\rho)} + \frac{1}{360(\rho n)^3} + \frac{1}{360(n(1-\rho))^3} \\
&\leq\quad -\frac{1}{12n(1-\rho)} + \frac{1}{360\rho n} + \frac{1}{360n(1-\rho)} \\
&\overset{\rho\geq 1-\rho}{\leq} -\frac{1}{12n(1-\rho)} + \frac{1}{180n(1-\rho)} \\
&\leq\quad 0
\end{aligned}$$

Und

$$\begin{aligned}
\binom{n}{\rho n}(q-1)^{\rho n} &\leq \sqrt{(2\pi n\rho(1-\rho))}^{-1}\rho^{-\rho n}(1-\rho)^{-n(1-\rho)}(q-1)^{\rho n} \\
&= \sqrt{(2\pi n\rho(1-\rho))}^{-1} q^{nH_q(\rho)}
\end{aligned}$$

Wenn die obere Schranke in ρ und $1-\rho$ symmetrisch ist, kann die Einschränkung $\rho \geq 1-\rho$ fallen gelassen werden. □

Folgerung 3.9 Sei $0 < \rho < 1 - \frac{1}{q},\ \beta_q(n) = \sqrt{n\rho(1-\rho)}$. Dann existiert für jedes n ein Überdeckungscode der Länge n, Radius höchstens ρn und Größe höchstens

$$n\beta_q(n)q^{n(1-H_q(\rho))}$$

.

Beweis:

$$\frac{1}{\sqrt{n\rho(1-\rho)}}\, q^{nH_q(\rho)} \ \leq\ V_q^n(a,r)\ \leq\ q^{nH_q(\rho)} \tag{3.10}$$

aus (3.7) und dem Kehrwert von (3.10) folgt:

$$\begin{aligned} \beta_q(n)q^{-nH_q(\rho)} &\geq V_q^n(a,r)^{-1} \\ &\Longleftrightarrow \\ n\beta_q(n)q^{n(1-H_q(\rho))} &\geq \frac{nq^n}{V_q^n(a,r)} \end{aligned} \tag{3.11}$$

□

Das folgende Lemma zeigt, dass es möglich ist, einen Code in annehmbarer Größe in polynomieller Laufzeit zu bestimmen:

Lemma 3.10 Sei $n \geq 1,\ 0 < \rho \leq 1 - \frac{1}{q},\ \beta_q(n) = \sqrt{8n\rho(1-\rho)}$. Dann kann ein Überdeckungscode der Länge n, Radius bis zu ρn und Größe maximal $n^2 \cdot \beta_q(n) \cdot q^{n(1-H_q(\rho))}$ gebildet werden in Laufzeit $p(n)q^{3n}$, p–Polynom.

Beweis: Der Code ist größer als derjenige aus Lemma 3.9, dieser Teil der Aussage ist also erfüllt. Wird sogar für die Bestimmung des Codes der Greedy Algorithmus aus Kapitel 4 verwendet, erhält man mit Lemma 4.1 einen noch kleineren Code. Für die Berechnung der Laufzeit sei hier auf Lemma 4.3 verwiesen. □

Lemma 3.11 Sei $d \geq 2$ ein Teiler von $n \geq 1$ und $0 < \rho < 1 - \frac{1}{q}$. Dann gibt es ein Polynom p_d so, dass ein Überdeckungscode der Länge n, Radius nahezu ρn und Größe maximal $p_d(n) \cdot q^{n(1-H_q(\rho))}$ mit einer Laufzeit von $p_d(n)(q^{3\frac{n}{d}} + q^{n(1-H_q(\rho))})$ gebildet werden kann.

Beweis: Der $\mathcal{Q}^n$ besteht aus n–stelligen Wörtern, die in d Blöcke der Länge n/d unterteilt werden. Mit Lemma 3.10 wird einen Überdeckungscode $\mathcal{C}'$ mit Radius nahezu $\frac{\rho n}{d}$ für $\mathcal{Q}^{\frac{n}{d}}$ konstruiert. Der Überdeckungscode $\mathcal{C}$ ist dann als direkte Summe von d Instanzen von $\mathcal{C}'$ definiert, z.B. die Menge aller Aneinanderreihungen von Wörtern aus $\mathcal{C}'$.
Bleibt zu zeigen, dass der so gebildete Überdeckungscode $\mathcal{C}$ Radius ρn hat:
Für jedes $a \in \mathcal{Q}^n$ kann eine Zerlegung $a_1, \ldots, a_d$ mit Länge $\frac{n}{d}$ pro Block gebildet werden. Zu jedem a_i wird ein $w_i \in \mathcal{C}'$ mit $d(a_i, w_i) \leq \frac{\rho n}{d}$ assoziiert. Die Aneinanderreihung $w = w_1 w_2 \ldots w_d$ ist dann ein Element aus $\mathcal{C}$ mit $d(a,w) \leq \rho n$. Die Konstruktion von $\mathcal{C}'$ erfolgt mit Laufzeit $O(p(n)q^{3\frac{n}{d}})$. Die Größe von $\mathcal{C}$ ist bis zu

$$\left(n^2 \cdot \beta_q(n) \cdot q^{\frac{n}{d}(1-H_q(\rho))}\right)^d = n^{2d} \cdot \beta_q(n)^d \cdot q^{n(1-H_q(\rho))}.$$

□

3.3 Verbesserungen der Codegröße

Die in Lemma 3.7 zuerst beschriebene Schranke für die Größe des Überdeckungscodes lautet $n \cdot \frac{q^n}{V_q^n(a,r)}$. Die zweite aus der Folgerung 3.9 ist aufgrund der anderen Konstruktionsmethode etwas größer.

Im Folgenden sollen noch einige gängige und bekannte Schranken vorgestellt werden.

Definition 3.12 Die *Hamming–* oder auch *Kugelpackungsschranke* beschränkt die Größe eines Codes $\mathcal{C}$ der Länge n über einem q–nären Alphabet wie folgt:

$$\#\mathcal{C} \cdot V_q^n(a,r) \leq \#\mathcal{Q}^n \tag{3.12}$$

Dabei ist der Radius $r = \left\lfloor \frac{d-1}{2} \right\rfloor$ mit d– Minimaldistanz der Codewörter.

Definition 3.13 Die *Kugelüberdeckungsschranke* beschränkt die Größe eines Überdeckungscodes $\mathcal{C}$ nach unten:

$$\#\mathcal{C} \geq \frac{q^n}{V_q^n(a,r)} \tag{3.13}$$

Bemerkung 3.14 In der Hamming–Schranke gilt das Gleichheitszeichen genau dann, wenn es zu jedem Wort $b \in \mathcal{Q}^n$ ein Codewort $a \in \mathcal{C}$ gibt, so dass

$$d_H(a,b) = r = \left\lfloor \frac{d-1}{2} \right\rfloor \tag{3.14}$$

Einen solchen Code nennt man *perfekten Code*.

Lemma 3.15 Ein perfekter Code erfüllt auch die Kugelüberdeckungsschranke mit Gleichheit und es gilt:

$$\forall\ b \in \mathcal{Q}^n\ \exists!\, a \in \mathcal{C} :\ d_H(a,b) \leq r \tag{3.15}$$

Beweis: Der erste Teil des Lemmas folgt durch Überführen der einen Gleichung in die andere.
Für den Beweis des zweiten Teils sei $\mathcal{C}$ eine Überdeckung des $\mathcal{Q}^n$. Folglich gilt:

$$\bigcup_{\forall i} \mathcal{U}_i = \mathcal{Q}^n$$

Seien nun a_1 und a_2 Codewörter aus $\mathcal{C}$ und b ein beliebiges Wort aus $\mathcal{Q}^n$ mit $d_H(a_1,b) \leq r$ und $d_H(a_2,b) \leq r$. Dann wird b sowohl bei der Berechnung von $V_q^n(a_1,b)$ als auch bei $V_q^n(a_2,b)$ „gezählt“. Daraus folgt, dass $\#\mathcal{C} \cdot V_q^n(a,r) \geq \#\mathcal{Q}^n + 1$ ist. □

Bemerkung 3.16 Dieser argumentative Beweis kann auch über die Dreiecksungleichung geführt werden:
Die Minimaldistanz zweier Codewörter eines perfekten Codes beträgt $2 * r + 1$. Der

Abstand eines Codewortes a_1 oder a_2 vom (beliebigen) Wort b ist kleiner als r. Somit können folgende Ungleichungen aufgestellt werden:

$$\begin{aligned} \#\{a_i^1 = b_i\} &\geq \#\{a_i^1 = a_i^2\} \\ \#\{a_i^2 = b_i\} &\geq \#\{a_i^1 = a_i^2\} \\ &\text{und} \\ \#\{a_i^1 = b_i\} &\geq n - r \\ \#\{a_i^2 = b_i\} &\geq n - r \\ \#\{a_i^1 = a_i^2\} &\leq n - (2 * r + 1) \end{aligned}$$

Daraus folgt, dass $\#\{a_i^1 = a_i^2\} \leq r - 1$. □

Eine weitere Möglichkeit die Codegröße nach unten abzuschätzen — also die minimal benötigte Anzahl an Codewörtern zu berechnen — ist die *Methode, die Überdeckungen zu zählen (method of counting excess)*. Die Idee dahinter ist, die Anzahl an Wörtern aus dem $\mathcal{Q}^n$ zu zählen, die von mehreren Codewörtern überdeckt werden. Diese Methode wurde von G. J. M. van Wee in [2] eingeführt und von G. D. Cohen *et al.* in ihrer Zusammenfassung über Überdeckungsradien [3] erneut beschrieben. Jedoch wurde diese Schranken nur für den binären Fall berechnet, so dass das erste von mehreren Hauptresultaten an dieser Stelle auf den q–nären Fall erweitert werden soll.

Lemma 3.17 Sei q die mehrwertige Basis des Codes $\mathcal{C}$, n, $r \in \mathbb{N}$ die Dimension des überdeckten Raumes und der Radius. Dann gilt:

$$\#\mathcal{C} \geq \frac{(n - r + \epsilon)q^n}{(n - r)V_q^n(r) + \epsilon V_q^n(r - 1)} \tag{3.16}$$

wobei

$$\epsilon = (r + 1)\left\lceil \frac{n + 1}{r + 1} \right\rceil - (n + 1)$$

ist.

Beweisskizze: Sei $\mathcal{C}$ wie im Lemma und $\mathcal{B}$ die Menge der Wörter aus dem $\mathcal{Q}^n$, die den genauen Abstand r zu $\mathcal{C}$ haben ($\Leftrightarrow d_H(a, b) = r \ \forall a \in \mathcal{C},\ b \in \mathcal{B}$).
Offensichtlich gilt

$$\#\mathcal{C} \cdot V_q^n(r - 1) + \#\mathcal{B} \geq q^n \tag{3.17}$$

Sei $\mathcal{Z}_i = \{z \in \mathcal{Q}^n, z \text{ wird von genau } i + 1 \text{ bedeckt}\}$ mit $i = 0, 1, \ldots$ und
$\mathcal{Z} = \bigcup_{i>0} \mathcal{Z}_i = \{z \in \mathcal{Q}^n, z \text{ wird von mindestens zwei Codewörtern überdeckt}\}$. Sei X eine Untermenge des $\mathcal{Q}^n$. Die Überdeckung von $\mathcal{C}$ über X ist definiert durch $E(X) = \sum_{i \geq 0} i|\mathcal{Z}_i \cap X|$ (ein Element, dass $i + 1$ mal überdeckt wird, gehört zu $E(X)$).

Die Idee ist, eine untere Schranke von $E(B_1(b))$ für ein b aus $\mathcal{B}$ anzugeben:
Wenn $b \in \mathcal{B}$, dann ist

$$E(B_1(b)) \equiv -|B_1(b)| \mod (r + 1)$$

und damit

$$E(B_1(b)) = \sum_{i\geq 0} i|\mathcal{Z}_i \cap B_1(b)| \geq \epsilon \quad \text{für alle } b \in \mathcal{B} \ . \tag{3.18}$$

Für $z \in \mathcal{Z}$ gibt es mindestens zwei Codewörter a_1 und a_2, so dass $d_H(z, a_1) \leq r$ und $d_H(z, a_2) \leq r$. Demzufolge ist

$$|\mathcal{B} \cap B_1(z)| \leq n + 1 - \lambda \ , \ \lambda = |B_1(z) \cap (B_{r-1}(a_1) \cup B_{r-1}(a_2))| \ .$$

Der nächste Schritt beruht auf der Tatsache, dass $\lambda \geq r + 1$. Der Fall $r = 1$ ist trivial, für $r \geq 2$ ist $(B_{r-1}(a_1) \cup B_{r-1}(a_2))$ nicht leer. Sei nun $d_1 = d_H(z, a_1)$, $d_2 = d_H(z, a_2)$. Ohne Beschränkung der Allgemeinheit tritt einer der vier Fälle ein:

1. $d_1 = d_2 = r$
2. $d_1 = r - 1$, $d_2 = r$
3. $d_1 = d_2 = r - 1$
4. $d_1 \leq r - 2$, $d_2 \leq r$

Dann kann für die Fälle 1), 2) und 3) gesagt werden, dass $d(a_1, a_2) = 2$ oder 1 ist, wobei 2 der schlechtere Fall wäre. Für diese drei Konstellationen von z, a_1 und a_2 ergibt sich $\lambda \geq r + 1$. Im vierten Fall ist $B_1(z) \subseteq B_{r-1}(a_1)$ und somit $\lambda \geq n + 1 \geq r + 1$.

Dies führt zu

$$|\mathcal{B} \cap B_1(z)| \leq n - r \quad \text{für alle } z \in \mathcal{Z} \ . \tag{3.19}$$

Mit (3.17), (3.18) und (3.19) sowie $\#\mathcal{C} \cdot V_q^n(r) - \sum_{i\geq 0} i|\mathcal{Z}_i| = q^n$ erhält man

$$\begin{aligned} \epsilon(q^n - \#\mathcal{C} \cdot V_q^n(r)) \leq \epsilon\#\mathcal{B} &\leq \sum_{b\in\mathcal{B}} \sum_{i\geq 0} i|\mathcal{Z}_i \cap B_1(b)| \leq \sum_{i\geq 0} i \sum_{z\in\mathcal{Z}_i} |\mathcal{B} \cap B_1(z)| \\ &\leq \sum_{i\geq 0} i(n-r)|\mathcal{Z}_i| = (n-r)(\#\mathcal{C} \cdot V_q^n(r) - q^n) \end{aligned}$$

und schließlich

$$\#\mathcal{C} \geq \frac{(n - r + \epsilon)q^n}{(n-r)V_q^n(r) + \epsilon V_q^n(r-1)} \ .$$

□

Beispiel: Nimmt man den 3^6 als $\mathcal{Q}^n$ und überdecken ihn mit dem Radius 3. Dann werden nach (3.7)

$$\frac{6 * 3^6}{V_3^6(3)} \approx 18,773$$

und somit 19 Codewörter benötigt. Nach (3.17) genügen

$$\frac{\Big(6 - 3 + \big((3+1)\lceil\frac{6+1}{3+1}\rceil - (6+1)\big)\Big)q^n}{(6-3)V_q^n(3) + \big((3+1)\lceil\frac{6+1}{3+1}\rceil - (6+1)\big)V_q^n(3-1)} \approx 3,777$$

also 4 Codewörter.

Die Kugelüberdeckungsschranke liefert als Wert $\approx 3,129$. Verwendet man Lemma 3.17 für die Berechnung der Codegröße, nähert man sich somit dicht an die Größe eines perfekten Codes an.

3.4 Heuristik der Codekonstruktion

Im Teil 3.2 dieses Kapitels wurde die Größe eines randomisierten Codes vorgestellt. Eine Heuristik mit zwei besonders markanten Schwächen: Zum einen wird nicht geprüft, ob ein Codewort mehrmals bestimmt wird, und zum anderen wird nicht geprüft, ob die Minimaldistanz des Codes größer als der doppelte Radius ist.

Diese Schwächen sind die Hauptgründe dafür, dass die Überdeckungseigenschaften des randomisiert bestimmten Überdeckungscodes eingeschränkt sind. Dieses Problem wird gelöst, indem der angegebene oder genauer gesagt verwendete Radius vergrößert (im Folgenden *expandiert* genannt) wird.

An dieser Stelle wird der Beweis dafür geliefert, dass das Expandieren des Radius auch stochastisch betrachtet die Überdeckungseigenschaft des randomisiert bestimmten Codes $\mathcal{C}$ verbessert.

Ziel ist jetzt zu zeigen, dass die Überdeckungseigenschaft eines randomisierten Codes bzw. zufällig gewählter Codewörtern verbessert werden kann, indem man den für die weiteren Rechnungen verwendeten Überdeckungsradius um den Faktor α vergrößert. Dabei bildet jedes Codewort a den Mittelpunkt einer Kugel mit Radius r und Volumen $V_q^n(a,r)$ wie im obigen Abschnitt.

Lemma 3.18 Sei $\mathcal{C}$ ein randomisiert bestimmter Code über dem $\mathcal{Q}^n$ mit $n \geq 1$, $r \in \mathbb{N}$ und mit Größe

$$\left\lceil \frac{n \cdot q^n}{V_q^n(a,r)} \right\rceil.$$

Dann ist derselbe Code mit expandiertem Radius αr, $\alpha \in \mathbb{R}$, $\alpha \geq 1$ eine bessere Überdeckung als mit nicht expandiertem Radius.

Beweis: Zu unserem jetzigen Code $\mathcal{C}$ und dem Code $\mathcal{C}'$ aus Lemma (3.7) gibt es in der Größe und Bestimmung keine Unterschiede.
Wähle $b \in \mathcal{Q}^n$ fest, $a \in \mathcal{C}$ beliebig. Dann hat „$b \in B_{\alpha r}(a)$“ die Wahrscheinlichkeit

$$\frac{V_q^n(a,\alpha r)}{q^n}$$

„$b \notin B_{\alpha r}(a)$“ hat Wahrscheinlichkeit

$$1 - \frac{V_q^n(a,\alpha r)}{q^n}$$

und „$b \notin B_{\alpha r}(a)\ \forall\ a \in \mathcal{C}$“ hat Wahrscheinlichkeit

$$\left(1 - \frac{V_q^n(a,\alpha r)}{q^n}\right)^{\#\mathcal{C}}.$$

Da $\frac{n \cdot q^n}{V_q^n(a,r)} \geq \frac{n \cdot q^n}{V_q^n(a,\alpha r)}$ erfüllt ist, gilt auch

$$\left(1 - \frac{V_q^n(a,\alpha r)}{q^n}\right)^{\frac{n \cdot q^n}{V_q^n(a,r)}} \geq \left(1 - \frac{V_q^n(a,\alpha r)}{q^n}\right)^{\frac{n \cdot q^n}{V_q^n(a,\alpha r)}}$$

und aus $\frac{V_q^n(a,\alpha r)}{q^n} \geq \frac{V_q^n(a,r)}{q^n}$ folgt:

$$1 - \frac{V_q^n(a,\alpha r)}{q^n} \leq 1 - \frac{V_q^n(a,r)}{q^n}$$

Und somit ist

$$\Big(1 - \frac{V_q^n(a,\alpha r)}{q^n}\Big)^{\frac{n\cdot q^n}{V_q^n(a,r)}} \leq \Big(1 - \frac{V_q^n(a,r)}{q^n}\Big)^{\frac{n\cdot q^n}{V_q^n(a,r)}}$$

Benutze nun $(1-\frac{1}{t})^{mt} \overset{t\to\infty}{\leq} e^{-m}$:

$$\rightsquigarrow \Big(1 - \frac{V_q^n(a,\alpha r)}{q^n}\Big)^{\frac{n\cdot q^n}{V_q^n(a,r)}} \leq \Big(1 - \frac{V_q^n(b,r)}{q^n}\Big)^{\frac{n\cdot q^n}{V_q^n(a,r)}} \leq e^{-n} \overset{n\to\infty}{\longrightarrow} 0 \qquad (3.20)$$

Somit ist ein Code $\mathcal{C}$ mit expandiertem Radius eine deutlich bessere Überdeckung als derselbe Code mit normalem Radius. □

Kapitel 4

Greedy Codes

Eine deterministische Art und Weise den $\mathcal{Q}^n$ zu überdecken ist der Greedy Algorithmus. Er geht nach folgender Methode vor: „Suche Dir stets den größtmöglichen Ball um den Raum zu bedecken, bis alles überdeckt ist.“

4.1 Greedy Algorithmus für Greedy Codes

Im Folgenden sei der Algorithmus im Pseudocode beschrieben:

- $U := \mathcal{Q}^n$;
  $C := \emptyset$;
- $c_1 := (00\ldots0)$;
  $i := 1$;
- $U := U \setminus B_r(c_1)$;
  $C := C \cup B_r(c_1)$;
- `while` $(U \neq \emptyset)$ `do`
  - $i := i + 1$;
  - $c_i := \arg\max_{c \in U} \#(B_r(c) \cap U)$;
  - $U := U \setminus B_r(c_i)$;
  - $C := C \cup B_r(c_i)$;
- $\{c_1, \ldots, c_i, \ldots, c_s\}$ bildet den gewünschten Code.

4.2 Größe des Greedy Codes

Die Größe dieses Überdeckungscodes beträgt nach [6] $\log_{\frac{1}{1-\epsilon}} q^n$, wobei hier ϵ aufgrund der gegebenen Einschränkungen gleich $\frac{V_q^n(a,r)}{q^n} \leq 1$ ist. Hier soll nun die Größe für $q \geq 2$ und Radien $r < n$ bestimmt werden.

Lemma 4.1 Für $q > 2$ und $r = \lfloor \rho * n \rfloor \in \mathbb{N},\ 0 \leq \rho \leq 1 - \frac{1}{q}$ beträgt die Größe des Greedy Codes näherungsweise

$$n \ln q \, \frac{V_q^n(a,r)}{q^n} \tag{4.1}$$

Beweis: Nach Einsetzen von $\epsilon = \frac{V_q^n(a,r)}{q^n}$ in $\log_{\frac{1}{1-\epsilon}} q^n$ erhält man:

$$\begin{aligned}
\log_{\frac{q^n}{q^n - V_q^n(a,r)}} q^n &= \frac{\log_q q^n}{\log_q \frac{q^n}{q^n - V_q^n(a,r)}} \\
&\approx \frac{n}{\log_q q^n - \log_q(q^n - q^{nH_q(\rho)})} \\
&= \frac{n}{n - \log_q(q^n(1 - q^{n(H_q(\rho)-1)}))} \\
&= \frac{n}{-\log_q(1 - q^{n(H_q(\rho)-1)})} \\
&= \frac{-n}{\ln(1 - q^{n(H_q(\rho)-1)}) / \ln q}
\end{aligned}$$

Nun wird die Potenzreihe von $\ln(1-x) = \sum_{k=1}^{\infty} -\frac{x^k}{k}$ für die Abschätzung angewendet. Die Voraussetzung $|x| < 1$ ist erfüllt, da $q^{n(H_q(\rho)-1)} < 1$ für Radius $r < n$ gilt.

$$\frac{-n}{\ln(1 - q^{n(H_q(\rho)-1)}) / \ln q} \approx \frac{-n \ln q}{\left(-q^{n(H_q(\rho)-1)} - o\left(\frac{(V_q^n(a,r))^2}{2q^{2n}}\right)\right)}$$

Mit dem $\mathcal{O}$–Kalkül entfällt $o\left(\frac{(V_q^n(a,r))^2}{2q^{2n}}\right)$:

$$\begin{aligned}
\frac{-n \ln q}{\ln(1 - q^{n(H_q(\rho)-1)})} &\approx \frac{n \ln q}{q^{n(H_q(\rho)-1)}} \\
&= n \ln q \cdot \frac{q^n}{V_q^n(a,r)}
\end{aligned}$$

□

Bemerkung 4.2 Für einen Radius $r = n$ beträgt die Codegröße 1.

4.3 Laufzeit des Greedy Algorithmus

Eine ausführliche Berechnung der Laufzeit dieses Algorithmus findet sich in [7]. Dort wird der Algorithmus verwendet, um das Mengenüberdeckungsproblem (*set cover problem*) zu lösen. Hier soll die Laufzeitberechnung speziell für eine Überdeckung des $\mathcal{Q}^n$ angepasst werden.

Lemma 4.3 Die Laufzeit des Greedy Algorithmus beträgt bis zu

$$p(n) \cdot q^{3n}$$

mit p–Polynom.

Beweis: Mit jeder Überdeckung wird die Anzahl der bisher nicht überdeckten Elemente assoziiert. In jeder Iteration des Algorithmus wird die Überdeckung ausgewählt, die die meisten Elemente überdeckt. Anschliessend wird die Zahl der nicht überdeckten Elemente aktualisiert. Dies geschieht mit Laufzeit $p(n) \cdot q^{2n}$, p–Polynom. Insgesamt werden bis zu q^n Iterationen benötigt. □

4.4 Speicherbedarf des Greedy Algorithmus

Die Schwierigkeit, den Greedy Algorithmus praktisch einzusetzen, liegt darin, dass er einen großen Speicherbedarf hat. Dieser soll nun bestimmt werden.

Lemma 4.4 Der Greedy Algorithmus hat bei der Erstellung eines Überdeckungscodes einen Speicherplatzbedarf in der Größenordnung von

$$q^n + n + n \ln q \frac{q^n}{V_q^n(a,r)} \ .$$

Beweis: Der Greedy Algorithmus führt stets eine Liste der schon bedeckten Elemente des $\mathcal{Q}^n$ sowie der noch nicht bedeckten. Dazu bedarf es Speicher in der Größenordnung q^n, da jedes Wort des $\mathcal{Q}^n$ nur in einer Liste vorkommt. Um ein Wort von einer Liste in die andere zu kopieren benötigt wird Speicherplatz in der Größenordnung der Dimension n benötigt. Da der Greedy Code die in Lemma 4.1 bewiesene Größe hat, braucht man für den Code noch Speicherplatz in der Größenordnung $n \ln q \frac{q^n}{V_q^n(a,r)}$. □

Lemma 4.5 Mit dem Greedy Algorithmus kann deterministisch ein Überdeckungscode der Größe

$$n \ln q \frac{q^n}{V_q^n(a,r)}$$

und Speicherbedarf

$$q^{n/d} + n/d + n \ln q \frac{q^n}{V_q^n(a,r)}$$

erstellt werden.

Beweis: Um einen Greedy Code mit dem genannten Speicherbedarf zu erstellen wird erneut Lemma 3.11 und die Methodik aus dem Beweis von jenem Lemma angewendet. Die Größe des Codes ändert sich dadurch nicht, was schon bewiesen wurde. □

Kapitel 5

Algorithmus zur Lösung von mv–SAT–Problemen

In diesem Kapitel werden der Hauptalgorithmus um erfüllende Belegungen eines mv–SAT–Problems zu finden und die Funktion `localSearch` vorgestellt.

Bevor jedoch der Algorithmus beschrieben wird, ist noch folgendes Lemma notwendig:

Lemma 5.1 Sei $x_i^{v_i^C}$ ein mehrwertiges Literal in einer Klausel C. Sei a ein Wort aus dem $\mathcal{Q}^n$ mit $x_i^{v_i^C} = 0$ unter a. Dann ist $x_i^{v_i^C} := 1$ unter a gleichbedeutend mit „setze $\overline{v_i^C} = a_i$“.

Beweis: $x_i^{v_i^C} = 0$ unter a ist gleichbedeutend mit $a_i \notin v_i^C$. Somit ist $a_i \in \mathcal{Q}^n \setminus v_i^C$. Wenn $x_i^{\overline{v_i^C}} = 1$ sein soll, muss $a_i \in \overline{v_i^C}$ erfüllt sein. Deshalb setze $\overline{v_i^C} = a_i$.

5.1 Beschreibung des Hauptalgorithmus

Der Algorithmus bekommt als Eingabe eine mv–SAT–Formel in k–CNF mit n Variablen. Die Idee ist, dass der $\mathcal{Q}^n$ mit Bällen des Radius ρn überdeckt wird. Für die Konstruktion dieser Überdeckung kann Lemma 3.11 oder eine andere der vorgestellten Möglichkeiten verwendet werden.

Hauptalgorithmus

- `Setze` $\rho := \frac{1}{k+1}$
- `Setze` $d := 6$
- `Bilde Überdeckungscode` $\mathcal{C}$ `der Länge` n `und Radius höchstens` ρn

- Für jedes Codewort a in $\mathcal{C}$ führe *localSearch*(F, a, r) aus
  Gibt *localSearch*$(F, a, r) ==$ 'true' zurück, so gib 'true' zurück
  andernfalls gib 'false' zurück

Bemerkung 5.2 Die Schritte 1 bis 3 können von einem anderen Programm bzw. Algorithmus übernommen werden. Insbesondere dann, wenn viele Formeln überprüft werden müssen, sollte ein Überdeckungscode für den $\mathcal{Q}^n$ nur einmal erstellt werden. Im Gegenzug dafür werden die Schritte 1 bis 3 durch Einlesen und Validieren des Codes ersetzt.

5.2 Lokale Suche mit der Funktion `localSearch`

Für die lokale Suche bekommt die Funktion `localSearch` als Eingabe die Formel, das Codewort und den Radius übergeben.

Function localSearch(F, a, r)

1. Ist $F(a) =$ 'true', gib 'true' zurück
2. Ist $(r \leq 0)$, gib 'false' zurück
3. Enthält F eine leere Klausel, so gib 'false' zurück
4. Nimm deterministisch eine Klausel C_j aus F mit $C_j(a) ==$ 'false'
   Setze Variable ret auf 'false'
   Für jedes Literal $x_i^{v_i^{C_j}} \in C_j$ prüfe:
   ist $v_i^{C_k} \subseteq v_i^{C_j},\ k \neq j$, so entferne C_k aus F.
   ist $v_i^{C_k} \subseteq \mathcal{Q}^n \setminus v_i^{C_j},\ k \neq j$, so entferne x_i aus C_k.
   ist $v_i^{C_k} \cap v_i^{C_j} \neq \emptyset$ und $v_i^{C_k} \cap \mathcal{Q}^n \setminus v_i^{C_j} \neq \emptyset,\ k \neq j$, dann
   setze $v_i^{C_k}$ auf $v_i^{C_k} \setminus (v_i^{C_k} \cap (\mathcal{Q}^n \setminus v_i^{C_j}))$.
   ergibt Search$(F_{\text{neu}}, a, (r-1))$ 'true', so setze ret auf 'true'
5. gib ret zurück

Bemerkung 5.3 In der Implementation des Algorithmus hat es sich bewährt, `localSearch` einen weiteren Parameter zu übergeben, in dem gespeichert wird, welche a_i auf welchen Wert gesetzt werden müssen. So erhält man am Ende der Rekursion ein Lösung der Formel.

Lemma 5.4 `localSearch`(F, a, r) hat Laufzeit $poly(n) \cdot ((q-1)k)^r$

Beweis: Die Rekursionstiefe von `localSearch`(F, a, r) ist maximal r. Wenn F eine mv–SAT–Formel im k–CNF Format ist, gibt es k Variablen, die maximal $(q-1)$ Werte in ihrer Wertemenge haben können – andernfalls wäre die Klausel erfüllt. Somit überprüft `localSearch`(F, a, r) maximal $(q-1) \cdot k$ Zuweisungsmöglichkeiten mit Rekursionstiefe r. □

Satz 5.5 Sei $F(a) = 0$, C eine beliebige Klausel in F mit $C(a) = 0$. Dann gilt: Existiert in C ein Literal x_i mit $F_{|_{v_i^C}}(b') = 1$, wobei $b' \in B_{r-1}(a)$, dann existiert ein $b \in B_r(a)$ mit $F(b) = 1$.

Beweis: F hat in $B_r(a)$ eine erfüllende Belegung $:\Longleftrightarrow$

$$\exists x_i : \; v_i^C \neq \emptyset, \exists b : \; b_i \in v_i^C \text{ und } d(a_{|a_i=b_i}, b) \leq r-1$$

Dieser Schritt wird wiederholt bis alle unerfüllten Stellen von a neu mit b_i's aus $B_r(a)$ belegt sind. □

Lemma 5.6 Mit diesem Algorithmus kann man auch erfüllende Belegungen einer mehrwertigen Formel finden.

Beweis: Setze zu Beginn das Lösungswort $s := s_1 s_2 \dots s_i \dots s_n$ mit $s_i = \mathcal{Q}$ für alle i.
Vor jedem Rekursionsaufruf über Klausel C und Literal x_i setze $s_i := s_i \cap v_i^C$; merke altes s_i.
Wenn der Rekursionsast nicht zum Erfolg führt, stelle altes s_i wieder her.
Zum Schluß ist $s = s_1 s_2 \dots s_n$ mit $s_i = \{\dots\} \subseteq \mathcal{Q}$.
Die Werte aus den s_i sind in Kombinationen mit den Werten der anderen s_i erfüllende Belegungen von F. □

Kapitel 6

Analyse der Laufzeit

Es soll nun die Laufzeit für den eben beschriebenen Algorithmus bestimmt werden. Dies wird für nicht expandierte Radien und dann für expandierte geschehen.

6.1 Berechnung mit nicht expandiertem Radius

Die Überdeckung besteht aus $B_q(n, \rho, d) = p_{d\ 2}(n)q^{n(1-H_q(\rho))}$ Bällen und die Konstruktion benötigt Laufzeit $T_1(n, \rho, d) = p_{d\ 1}(n)(q^{3n/d} + q^{n(1-H_q(\rho))})$. In jedem Ball wird lokale Suche verwendet, die pro Ball Laufzeit $T_2(n, \rho) = p_{d\ 3}(n)\left((q-1)k\right)^{\rho n}$ hat. Die Laufzeit des gesamten Algorithmus beträgt folglich

$$T_1(n, \rho, d) + B_q(n, \rho, d) \cdot T_2(n, \rho) \tag{6.1}$$

Um den exponentiellen Teil des Produkts zu minimieren wird $\rho = \frac{1}{k+1}$ fest gewählt. Durch geschickte Wahl von d, beispielsweise $d = 6$, wird der erste Term kleiner als der zweite.

Satz 6.1 Der Algorithmus löst mv–SAT–Formeln mit k Variablen pro Klausel mit Laufzeit $p_d(n) \cdot \left(q - \frac{q}{k+1}\right)^n$, wobei q die Größe des Alphabets ist und n die Anzahl der Variablen der eingegebenen Formel.

Beweis: (6.1) berechnet sich wie folgt:

$$
\begin{aligned}
& T_1(n,\rho,d) + B_q(n,\rho,d)\cdot T_2(n,\rho) \\
&= \; p_{d\ 1}(n)(q^{3n/d} + q^{n(1-H_q(\rho))}) \;+\; p_{d\ 2}(n) q^{n(1-H_q(\rho))}\cdot p_{d\ 3}(n)\left((q-1)k\right)^{\rho n} \\
&= \; p_d(n)\cdot\ (q-1)^{\rho n}\cdot\ \left(q^{n(1-H_q(\rho))+\rho n\log_q k}\right) \\
&= \; p_d(n)\cdot\ (q-1)^{\rho n}\cdot\ \left(q^{n(1-\frac{1}{k+1}\log_q(k+1)(q-1)-\frac{k}{k+1}\log_q\frac{k+1}{k}+\frac{1}{k+1}\log_q k)}\right) \\
&= \; p_d(n)\cdot\ (q-1)^{\rho n}\cdot \\
&\qquad \left(q^{n(1-\frac{1}{k+1}\log_q(k+1)-\frac{k}{k+1}\log_q(k+1)+\frac{1}{k+1}\log_q k+\frac{k}{k+1}\log_q k-\frac{1}{k+1}\log_q(q-1))}\right) \\
&= \; p_d(n)\cdot\ (q-1)^{\rho n}\cdot\ \left(q^{n(1-\frac{1}{k+1}\log_q(q-1)+\log_q\frac{k}{k+1})}\right) \\
&= \; p_d(n)\cdot\ (q-1)^{\rho n}\cdot\ \left(q\,\frac{k}{k+1}\,(q-1)^{-\frac{1}{k+1}}\right)^n \\
&= \; p_d(n)\cdot\ (q-1)^{\rho n}\cdot\ (q-1)^{-\frac{1}{k+1}n}\cdot\ \left(q-\frac{q}{k+1}\right)^n \\
&= \; p_d(n)\cdot\ \left(q-\frac{q}{k+1}\right)^n \qquad (6.2)
\end{aligned}
$$

□

6.2 Berechnung mit expandiertem Radius

In Kapitel 3.3 wurde eine Verbesserung der Überdeckungseigenschaften des randomisierten Überdeckungscodes durch eine „künstliche“ Vergrößerung des Überdekkungsradius bei gleich bleibender Anzahl an Codewörtern vorgestellt.

Hier soll nun die Berechnung der Laufzeit erfolgen.

Satz 6.2 Der Algorithmus löst mv–SAT–mit k Variablen pro Klausel und Faktor α zur Vergrößerung des Überdeckungsradius mit Laufzeit

$$p_d(n)\cdot\ \left(((q-1)k)^{\frac{\alpha-1}{k+1}}\cdot\ (q-\frac{q}{k+1})\right)^n,$$

wobei q die Größe des Alphabets ist und n die Anzahl der Variablen der eingegebenen Formel.

Beweis: Die Anzahl der Überdeckungsbälle sowie die Laufzeit der Konstruktion des Überdeckungscodes bleiben unverändert. Die Laufzeit der lokalen Suche verändert sich durch den Faktor α dahingehend, dass der Radius noch mit α multipliziert wird, also $\alpha\rho n$ groß wird. Somit berechnet sich

$$T_1(n,\rho,d) + B_q(n,\rho,d)\cdot T_2(n,\alpha\rho) \qquad (6.3)$$

wie folgt:

$$
\begin{aligned}
& T_1(n,\rho,d) + B_q(n,\rho,d)\cdot T_2(n,\alpha\rho) \\
&= p_{d\ 1}(n)(q^{3n/d} + q^{n(1-H_q(\rho))}) + p_{d\ 2}(n) q^{n(1-H_q(\rho))}\cdot p_{d\ 3}(n)\,((q-1)k)^{\alpha\rho n} \\
&= p_d(n)\cdot(q-1)^{\alpha\rho n}\cdot\frac{(q-1)^{(1-\alpha)\rho n}}{(q-1)^{(1-\alpha)\rho n}}\cdot\left(q^{n(1-H_q(\rho))+\alpha\rho n\log_q k}\cdot\frac{q^{(1-\alpha)\rho n\log_q k}}{q^{(1-\alpha)\rho n\log_q k}}\right) \\
&= p_d(n)\cdot\ (q-1)^{\rho n}\cdot\ (q-1)^{(\alpha-1)\rho n}\cdot\ q^{(\alpha-1)\rho n\log_q k}\cdot \\
&\qquad \left(q^{n(1-\frac{1}{k+1}\log_q(k+1)(q-1)-\frac{k}{k+1}\log_q\frac{k+1}{k}+\frac{\alpha}{k+1}\log_q k)}\right) \\
&= p_d(n)\cdot\ ((q-1)k)^{(\alpha-1)\rho n}\cdot\ (q-1)^{\rho n}\cdot \\
&\qquad \left(q^{n(1-\frac{1}{k+1}\log_q(k+1)-\frac{k}{k+1}\log_q(k+1)+\frac{1}{k+1}\log_q k+\frac{k}{k+1}\log_q k-\frac{1}{k+1}\log_q(q-1))}\right) \\
&= p_d(n)\cdot\ ((q-1)k)^{(\alpha-1)\rho n}\cdot\ (q-1)^{\rho n}\cdot\ \left(q^{n(1-\frac{1}{k+1}\log_q(q-1)+\log_q\frac{k}{k+1})}\right) \\
&= p_d(n)\cdot\ \big((q-1)k\big)^{(\alpha-1)\rho n}\cdot\ (q-1)^{\rho n}\cdot\ \left(q\,\frac{k}{k+1}\,(q-1)^{-\frac{1}{k+1}}\right)^n \\
&= p_d(n)\cdot\ \big((q-1)k\big)^{(\alpha-1)\rho n}\cdot\ (q-1)^{\rho n}\cdot\ (q-1)^{-\frac{1}{k+1}n}\cdot\ \left(q-\frac{q}{k+1}\right)^n \\
&= p_d(n)\cdot\ \left(\big((q-1)k\big)^{\frac{\alpha-1}{k+1}}\cdot\ \big(q-\frac{q}{k+1}\big)\right)^n \qquad (6.4)
\end{aligned}
$$

□

Beispiel: Diese Formel ist auf den ersten Blick nicht sonderlich aussagekräftig. Darum soll hier ein Beispiel zur Berechnung mit konkreten Zahlen gegeben werden. Dabei sei $q = 2$, $k = 3$ und n unbestimmt.

1. $\alpha = 1,5$
   Einsetzen von q und k führt zu:
$$
\begin{aligned}
& (3^{\frac{0,5}{4}}\cdot 1,5)^n \\
\approx\ & 1.721^n
\end{aligned}
$$

2. $\alpha = 2$
   Einsetzen von q und k ergibt:
$$
\begin{aligned}
& (3^{\frac{1}{4}}\cdot 1,5)^n \\
\approx\ & 1.974^n
\end{aligned}
$$

Die Laufzeit bleibt somit immer noch unter der Marke der vollständigen Suche, die eine Laufzeit von $O(2^n)$ hat. Diese Laufzeit ist die schlechteste bekannte.

Kapitel 7

Speicherformate für mv–SAT–Probleme und Überdeckungscodes

7.1 Allgemeines

Am DIMACS Center der Rutgers University in New Jersey[1] wurde ein Format entwickelt, mit dem SAT–Probleme abgespeichert werden können. Vorgestellt wurde es in [8]. Die Idee ist, jeder Zeile ein Schlüsselzeichen an den Anfang zu setzen, das den Inhalt eindeutig kennzeichnet; die Art der Informationsdarstellung ist auch genau beschrieben. Außerdem ist genau festgelegt, in welcher Reihenfolge die Informationen dem parsenden Programm zur Verfügung gestellt werden.

Um mv–SAT–Probleme automatisiert effektiv lösen zu können bzw. auf Erfüllbarkeit zu testen, braucht es ebenfalls ein Format, in dem die Formeln abgespeichert werden können. Das Format des DIMACS Center soll hier auf mv–SAT–Probleme erweitert werden. Ebenso wurde das Speicherformat für die Überdeckungscodes, das auch in diesem Kapitel vorgestellt werden soll, ist auf Basis obigen Formats entwikkelt. Ziel dabei ist es, unabhängig der verwendeten Plattformen oder Programme leicht zu parsende Dateien[2] zu verwenden.

[1] `http://www.dimacs.rutgers.edu`

[2] Allerdings wird in der Implementation bisher nicht zwischen den diversen Betriebssystemen unterschieden. Das heißt, dass nur auf ein „Line Feed“ (’\n’) geparst wird, bzw. ein ’\n’ geschrieben wird. Somit ist eine Kompatibilität mit Unix–Systemen und eingeschränkt mit Windows–Systemen gegeben – mit Mac OS jedoch nicht, da dort ein „Carriage Return“ (’\r’) als Zeilenendezeichen gilt.

7.2 mv–SAT–Probleme

Die Dateien enthalten mehrere Zeilen mit ASCII–Zeichen. Ein- und Ausgabedateien enthalten verschiedene Arten von Zeilen, unten näher beschrieben. Jede Zeile endet mit einem Zeilenendezeichen. Einträge innerhalb einer Zeile werden durch ein Leerzeichen voneinander getrennt.

Jede Datei besteht aus zwei Hauptteilen: der *Präambel* und den *Klauseln*.

Präambel

Die Präambel enthält Informationen über den vorliegenden Fall. Diese Information steht in Zeilen, die wiederum mit speziellen Buchstaben, gefolgt von einem Doppelpunkt und einem Leerzeichen, beginnen. Folgende Arten von Informationen dürfen vorkommen:

- **Kommentare:** Kommentarzeilen enthalten Informationen für „menschliche“ Leser und werden vom Programm ignoriert. Sie stehen am Anfang der Präambel. Jede Kommentarzeile beginnt mit dem Kleinbuchstaben „c“.

  ```
  c: Dies ist ein Beispiel für eine Kommentarzeile
  ```

- **Problemzeile:** Pro Eingabedatei existiert eine Problemzeile. Sie muss vor allen anderen das Problem spezifizierenden Informationen stehen. Für mv–CNF–Formeln hat sie folgendes Format:

  ```
  p: FORMAT BASIS DIMENSION KLAUSELN
  ```

  Der Kleinbuchstabe „p“ kennzeichnet die Problemzeile. Die „`FORMAT`“–Eingabe erlaubt es dem Programm das erwartete Format zu bestimmen und sollte „`mvcnf`“ enthalten. „`BASIS`“ bezeichnet die Anzahl der möglichen Variablenwerte, „`DIMENSION`“ die der Variablen und „`KLAUSELN`“, wie viele Klauseln die Formel enthält. `BASIS`, `DIMENSION` und `KLAUSELN` enthalten vorzeichenlose Integer–Werte.

  ```
  p: mvcnf 3 6 3
  ```

  Diese Zeile muss als letzte der Präambel erscheinen.

Bemerkung 7.1 An dieser Stelle ist erwähnenswert, dass das von DIMACS vorgeschlagene Format keine Doppelpunkte nach den Zeilenbezeichnern vorsieht. Um aber das maschinelle Einlesen besser gegen Fehler schützen zu können, verlangt die vorliegende Implementation den Doppelpunkt. Fehler können hierbei leicht entstehen, wenn ein Überdeckungscode mittels eines gepufferten Datenstroms übergeben wird.

Klauseln

Die Klauseln werden sofort nach der Problemzeile aufgeschrieben. Die enthaltenen Literale werden dabei aus einer Zahl von 1 bis n gebildet, dem in „`DIMENSION`“ gespeicherten Wert, gefolgt von ihrer Wertemenge v_i in einfachen Klammern. Die Elemente der Wertemenge werden auch durch Leerzeichen getrennt. Umfasst v_i ganz $\mathcal{Q}$, so kann die Angabe entfallen, ist v_i gleich der leeren Menge, wird dies durch leeren Klammerinhalt dargestellt. Die Negation wird durch ein der Variable vorgestelltes „$-$“ gekennzeichnet.
Jede Klausel wird wie im DIMACS–Format durch den Wert 0 abgeschlossen.

Die Werte in der Wertemenge werden durch Leerzeichen getrennt.

Beispiel:

```
c: Dies ist ein Beispiel für eine mv--SAT--Formel
c: Die Klausellaenge betraegt 3

p: mvcnf 3 6 3

1(1 3) 3(2) 5 0 2(1) -4(2 3) 6() 0 3(1 2) 6(1 3) 4(2 3) 0
3 1() 5() 0 4(1 2) -3 2 0 3(1 3) 4(2) 5(1 2) 0
```

7.3 Überdeckungscodes

Das Speicherformat für Überdeckungscodes baut auf dem für mv–SAT–Probleme auf. So wird der Inhalt der Zeilen durch ein „Schlüsselzeichen“ gekennzeichnet. Jede neue Information bekommt auch eine neue Zeile.

Die Schlüsselzeichen lauten:

- `c:` Wie bei den Formeldateien stehen in mit einem „c“ gekennzeichneten Zeilen Kommentare.
- `b:` Die Basis wird mit ASCII-Ziffern als *unsigned long* aufgeschrieben.
- `d:` Die Dimension wird analog zur Basis aufgeschrieben.
- `r:` Der Radius des Überdeckungscodes wird genauso wie Basis und Dimension aufgeschrieben.
- `s:` Die Größe des Codes, ebenfalls angegeben wie die Basis.

Die einzelnen Codewörter werden im Anschluss zeilenweise in die Datei geschrieben. Jede Zeile beginnt hierbei mit einem `w` gefolgt von einem Doppelpunkt. Die Codewörter selbst sind in einem modifizierten Stellenwertsystem aufgeschrieben. Dabei wird jede Stelle im 10er–System notiert, die einzelnen Stellen durch Punkte getrennt.

Die Angabe der Codegröße ist notwendig, da diese von der Art der Bestimmung des Codes abhängt. Andernfalls müsste das einlesende Programm die Berechnung der Codegröße selbst durchführen. Dies funktioniert jedoch nur dann fehlerfrei, wenn der Typ des Codes mit übergeben wird.

Beispiel:

```
c: Dies ist ein Beispiel für einen Ueberdeckungscode
c: Erstellt am: 2007-04-01
c: Randomisiert bestimmter Ueberdeckungscode mit folgenden Parametern:


b: 3
d: 6
r: 3
s: 19


w: 2.0.1.1.1.2
w: 0.1.2.1.1.0
w: 1.0.1.1.2.2
w: 2.2.0.0.1.1
w: 2.1.2.0.0.1
w: 1.2.2.0.1.2
w: 0.0.0.0.1.1
w: 0.0.0.2.2.0
w: 2.1.0.0.2.1
w: 2.2.2.2.2.2
w: 1.2.0.0.2.2
w: 2.0.2.0.1.1
w: 2.2.2.0.2.1
w: 0.2.2.0.0.2
w: 2.2.2.2.1.1
w: 2.2.1.2.1.1
w: 0.1.2.2.2.0
w: 1.2.0.0.2.2
w: 2.0.2.1.1.0
```

Kapitel 8

Die Programme `createCode` und `solveSAT`

Der vorgestellte Algorithmus mit der Methodik der lokalen Suche[1] wurde in der Sprache `C` implementiert. Hierfür wurden zwei Programme geschrieben:

- Das Programm **createCode**, das einen randomisiert bestimmten Überdekkungscode erstellt, und
- das Programm **solveSAT**, das ein SAT–Problem mit gegebenem Überdekkungscode löst.

Der Vorteil dieser Aufteilung liegt darin, dass es unerheblich ist woher der Überdekkungscode stammt oder wie er erstellt wurde. Angegeben werden muss er jedoch auf jeden Fall. Diese Aufteilung kann auch zu einem späteren Zeitpunkt genutzt werden um den Überdeckungscode durch andere Programme oder einem *Computer–Algebra–System (CAS)*, zum Beispiel MuPAD, zu erstellen.

Die Sprache `C` wurde gewählt, da sie besonders hardwarenah ist und somit gute Laufzeitergebnisse erzielt werden können.

Die Quellcodedateien befinden sich auf der CD–ROM im Verzeichnis „Programme“.

8.1 Benutzung der Programme

Das Lösen von SAT–Problemen mit den Programmen `createCode` und `solveSAT` erfolgt durch Aufruf zuerst von `createCode` mit unten aufgeführten, teilweise optionalen Parametern. Anschließend kann der so erzeugte (randomisierte) Überdekkungscode an das Programm `solveSAT` übergeben werden, damit dieses ihn verwendet, um Lösungen der Formel zu suchen.

[1] Derzeit ist jedoch lediglich die Implementation der Funktion `localSearch` für den binären Fall umgesetzt.

Es ist auch möglich, den Überdeckungscode von `createCode` formatiert über einen gepufferten Datenstrom (*Pipe*) an `solveSAT` zu übergeben. Damit kann zum Beispiel mit einem Shell–Skript für jede Formel einfach ein neuer Überdeckungscode erzeugt werden. Hier ist jedoch eine Einschränkung zu beachten: Der gepufferte Datenstrom einer bash–Shell schließt die Ausgabe mit einem „`EOF`“ ab. Das hat zur Folge, dass für den weiteren Programmablauf der Datenstrom `stdin` „geschlossen“ wird, und das Programm `solveSAT` keine Eingaben über die Konsole mehr lesen kann. Darum wird bei dieser Art und Weise, die Programme zu verwenden, die Benutzerkommunikation bei `solveSAT` automatisch ausgeschaltet. Dies betrifft die Ausgabe des Überdeckungscodes auf die Konsole sowie eine Überprüfung einiger weniger Hamming–Distanzen der Codewörter. Die Ausgabe der Formel und der gefundenen Lösungen ist hiervon nicht betroffen. Der verwendete Überdeckungscode kann jedoch in der von `createCode` geschriebenen Datei[2] nachvollzogen werden.

Aufrufparameter der Programme

- **createCode:**

```
createCode [-H] |
           [-S] [-A] [-F <Ausgabedatei>]
           (-U | -C -b <unsigned int> -d <unsigned int>
                   (-r | -h) <unsigned int> |
           <Eingabedatei>)
```

Die Eingabeparameter haben folgende Bedeutung:

-H Gibt eine Hilfe für den Aufruf des Programms aus.

-S „silent“, unterdrückt Ausgaben auf den Monitor, genauer gesagt, auf die Konsole.

-U Eingabe der Parameter des Codes durch den User über die Tastatur, eine Angabe von „-S“ wird dabei überschrieben.

-C Eingabe der Parameter des Codes durch die Tastatur beim Programmaufruf. Eine mit angegebene Eingabedatei wird hierbei ignoriert, ebenso wie „S“ und „U“.

- -b Bezeichnet die Basis des Codes.
- -d Bezeichnet die Dimension des Codes.
- -r Bezeichnet den Radius des Codes – kann durch die Angabe von „k“ ersetzt werden.
- -k Bezeichnet die maximale Anzahl an Variablen pro Klausel des zu lösenden (mv) SAT–Problems. Kann die Angabe von „-r“ ersetzen.

-A Der erzeugte Code wird formatiert auf `stdout` geschrieben.

-F Der erzeugte Code wird in die angegebene Datei geschrieben. Der Dateiname muss im Anschluss folgen.

[2] Sofern kein anderer Dateiname angegeben wurde, erstellt `createCode` eine Datei mit Namen `ausgabeRandomCoveringCode_YYYY-MM-DD.rcc`. YYYY-MM-DD steht für das aktuelle Datum. Bei Problemen mit einer Datei diesen Namens wird ersatzweise die Datei `ausgabe.rcc` verwendet.

Eine Angabe von „r“ oder „k“ ist für die Berechnung der Größe des Codes unerlässlich. Wird „k“ angegeben, erfolgt die Berechnung von „r“ nach der Formel $r = d/(k+1)$.
Ebenso ist die Angabe von „U“, „C“ oder als letztem Parameter ein Dateiname Pflicht. Ist keiner der Parameter angegeben, bricht das Programm mit einer Fehlermeldung ab.
Werden die Codeparameter mittels einer Datei an `createCode` übergeben, so muss diese Eingabedatei an letzter Stelle stehen.
Soll mit „A“ der Überdeckungscode formatiert auf `stdout` geschrieben werden und mit einem gepuffertem Datenstrom an `solveSAT` übergeben werden, so wird die Verwendung des Parameters „S“ empfohlen. Andernfalls kann es passieren, dass `solveSAT` mit einer Fehlermeldung abbricht. Auf ein automatisches Abschalten von Ausgaben wurde hierbei bewusst verzichtet, damit der Benutzer die Möglichkeit hat, sich den Überdeckungscode formatiert auf der Konsole zu betrachten.
Dem Benutzer ist die Reihenfolge der Parameter nicht vorgeschrieben. Er kann sie beliebig permutieren. Die einzige Einschränkung ist, dass der Name der Ausgabedatei nach dem Parameter „F“ folgen muss, und die Eingabedatei an letzter Stelle stehen muss. Ebenso müssen Zahlwerte im Anschluss an den zugehörigen Parameter übergeben werden.

- **solveSAT:**

```
solveSAT [-H] |
         [-S] [-A] [-T] [-L <Logfile>]
         [-R <unsigned int>] [-D <unsigned int>]
         -F <Formeldatei> (-U | -C <Codedatei>)
```

Die Eingabeparameter haben folgende Bedeutung:

-H Gibt eine Hilfe für den Aufruf des Programms aus.

-S „silent“, unterdrückt Ausgaben auf den Monitor, genauer gesagt, auf die Konsole.

-U Formatierte Eingabe des Codes über `stdin` – dient für die Eingabe des Codes mittels eines gepufferten Datenstroms.

-C Bezeichnet die Datei, in der ein einzulesender Code gespeichert ist. Der Dateiname muss im Anschluss folgen.

-F Bezeichnet die Datei, in der die Formel gespeichert ist. Der Dateiname muss im Anschluss folgen.

-A Die Ergebnisse des Algorithmus werden auf `stdout` ausgegeben.

-L Bezeichnet das Logfile, in dem die Ergebnisse der Funktion `localSearch` protokolliert werden.

-R Faktor, um den der Überdeckungsradius künstlich erweitert werden kann. Dient der Verbesserung der Überdeckungseigenschaften.

-D Bestimmt die maximale Anzahl an falschen Klauseln, die pro Rekursionsteilbaum überprüft werden.

-T Es wird ein Rekursionsbaum in der Datei `localSearchRecTree.txt` gespeichert. Nach jedem Aufruf des Programms sollte diese Datei gesichert werden, da sie bei einem neuen Aufruf mit diesem Parameter überschrieben wird.

Die Parameter „-U“ und „-C“ können sich gegenseitig ersetzen, wenngleich einer von ihnen gesetzt werden muss. Ansonsten bricht das Programm mit einer Fehlermeldung ab. Des weiteren führt eine Nichtangabe der Formeldatei zum sofortigen Programmabbruch.

Der Parameter „T“ kann zu einem Programmabbruch durch das Betriebssystem führen, wenn die Formel sehr viele Variablen und Klauseln hat. Dann kann die Datei mit dem Rekursionsbaum sehr groß werden und die vom Dateisystem festgelegte Maximalgröße einer Datei überschreiten. Insbesondere bei einer Kombination aus „T“ und „D“ kann dies leicht passieren. Ein Neuaufruf ohne diesen Parameter lässt das Programm dann wieder normal arbeiten.

Auch bei `solveSAT` ist der Benutzer frei, in welcher Reihenfolge er die notwendigen Parameter angibt. Allerdings gilt auch hier, dass Dateinamen und Zahlwerte im Anschluss an den zugehörigen Parameter folgen müssen.

8.2 Parameter mit Laufzeitbeeinflussung

Zwei Aufrufparamter, `-R` und `-D`, nehmen unmittelbaren Einfluss auf den Ablauf und die Laufzeit des Programms `solveSAT`. Ihre Funktion soll im Folgenden erläutert werden.

Der Aufrufparameter –R des Programms `solveSAT`

`solveSAT` kann mit dem Parameter „–R“, gefolgt von einem Fließkommawert, aufgerufen werden. Dieser bewirkt eine „künstliche“ Ausdehnung der Überdeckungsbälle der einzelnen Codewörter. Wie in 3.3 beschrieben, erhöht dies die Wahrscheinlichkeit dafür, dass der angegebene Code den $\mathcal{Q}^n$ überdeckt.

Dies hat Auswirkung auf die Laufzeit der Funktion `localSearch`, da die Rekursionstiefe um diesen Faktor α erhöht wird. Die Berechnung der Laufzeit wurde in 6.2 vorgestellt.

Der Aufrufparameter –D des Programms `solveSAT`

Die Implementation des Algorithmus beinhaltet noch einen weiteren „Steuerungswert“ für den Programmablauf. So wird mit dem Parameter –D, gefolgt von einem `unsigned int` Wert, die Anzahl der Klauseln gesteuert, mit deren Variablen neue Rekursionsaufrufe erfolgen. Wird das Programm mit Standardwerten, also ohne die entsprechende Option, aufgerufen, so ist der Wert gleich 1.

Lemma 8.1 Die Gesamtgröße des Rekursionsbaumes pro Codewort berechnet sich wie folgt:

$$\#\{\text{zu überprüfende Klauseln}\}^{\text{Überdeckungsradius}}$$

Beweis: Die Rekursionstiefe ist gleich dem Überdeckungsradius, und in jeder Rekursion werden nacheinander entsprechend viele unerfüllte Klauseln auf Erfüllbarkeit hin überprüft. □

Eine binäre 3–SAT–Formel mit 91 Klauseln, Dimension 20 und einem Überdekkungscode mit Radius 4, der nicht expandiert wird, hat somit einen „worst case"–Rekursionsbaum der Größe

$$91^4 * 3^4 = 273^4 = 5\,554\,571\,841$$

für jedes einzelne Codewort, sofern jede falsche Klausel überprüft werden soll.

8.3 Technische Einzelheiten der Programmierung

In diesem Unterkapitel soll auf die wichtigsten Details der Programmierung eingegangen werden. Dabei handelt es sich um die Konstrukte, mit denen Überdeckungscodes und binäre SAT–Formeln beschrieben werden. Diese Elemente der Programmierung werden von allen Modulen benötigt und durch Einbindung der entsprechenden Header–Datei bzw. durch Deklaration als externe Variablen importiert. Da ist es von Vorteil, dass `createCode` und `solveSAT` dieselben Module verwenden.

Basis, Dimension und Radius des Codes

Diese drei Werte sind naturgegeben existenziell für die Erstellung und Benutzung der Überdeckungscodes. Sie werden von allen Modulen und den Hauptprogrammen benötigt. Aus diesem Grund wurden sie global im Modul `defineCode` deklariert und initialisiert. Die Hauptprogramme und die anderen Module binden sie als externe Variablen ein.

Ebenso sind sie für die Suche nach Lösungen der (binären) k–SAT Formeln von grundlegender Bedeutung. Stimmen Basis und Dimension des Codes nicht mit Basis und Dimension der Formel überein, ist das Problem mit den gegebenen Bedingungen nicht lösbar.

Strukturen für Überdeckungscodes

Überdeckungscodes werden in der Struktur `Code` gespeichert. Ihre Vereinbarung steht im Modul `defineCode` in der Header–Datei. In der Struktur werden die Größe des Codes als `unsigned int` und die Codewörter in einem dynamischen Array gespeichert. Sie lautet:

```
typedef struct
{
  /*
   * Hier wird ein Code abgelegt
   */
  /*Groesse des Codes*/
  unsigned int size;
  /*Liste der Codewoerter*/
  codeWord *wordList;
} Code;
```

Der Wert `size` ist zwar auch von großer Bedeutung, aber nur im Zusammenhang mit dem Überdeckungscode an sich. Er hat nur untergeordnete Relevanz für die Formeln. Deswegen wurde er nicht global deklariert.

In der Struktur `codeWord` werden die einzelnen Codewörter abgelegt, insgesamt `size` viele. Ihre Vereinbarung steht auch in der Header–Datei des Moduls `defineCode`:

```
typedef struct
{
  /*
   * Hier wird ein Codewort abgelegt.
   */
  unsigned int *word;
} codeWord;
```

Alternativ könnten die Codewörter direkt in der Struktur `Code` als zweidimensionales Array der Größe `size`×`dimension` gespeichert werden. Dennoch wurde den Codewörtern bewusst eine eigene Struktur deklariert. Dies vereinfachte die Verarbeitung eines einzelnen Wortes im Ablauf der Programme.

Strukturen für (binäre) k–SAT Formeln

SAT–Formeln werden in der Struktur `binFormula` gespeichert. Die Vereinbarung dieser Struktur steht in der Header–Datei des Moduls `binarySAT`. Gespeichert werden die Anzahl der Klauseln dieser Formel, die maximale Anzahl an Variablen pro Klausel und die Klauseln selbst in einem dynamischen Array:

```
typedef struct
{
  /*
   * Hier wird eine Formel abgelegt.
   */
  /*Groesse der Formel*/
  unsigned int numberClauses;
  /*maximale Anzahl der Literale pro Klausel*/
  unsigned int k;
  /*Liste der Klauseln*/
  binClause *formula;
} binFormula;
```

In der Struktur `binClause` werden die einzelnen Klauseln der Formel gespeichert. Ihre Anzahl beträgt `numberClauses` viele. Die Vereinbarung der Struktur erfolgt ebenfalls in der Header–Datei des Moduls `binarySAT`:

```
typedef struct
{
  /*
   * Hier wird eine Klausel abgelegt.
   */
  char *clause;
} binClause;
```

Die Größe einer Klausel beträgt stets `dimension`. Literale, die positiv in der Klausel vertreten sind, werden durch ein `'+'` gekennzeichnet, negierte durch ein '-'. Alle anderen haben einen davon verschiedenen Wert.

Die Deklaration einer eigenen Struktur für die Klauseln erfolgte nach denselben Überlegungen wie bei den Codewörtern.

Enum–Werte für den Programmablauf

Im Modul `systemSignals` sind insgesamt 3 Enum–Werte für den Programmablauf hinterlegt:

- `statusAblauf` dient dazu, den Funktionen die Möglichkeit zu geben, einen genauen Grund für das Ende der Funktion anzugeben.
- `inputParameters` dient dazu, die Eingabeparameter der Programmaufrufe einfach zu parsen und zu erfassen.
- `boolean` ist eine „low–level“ Umsetzung Bool'scher Werte speziell für das Programm `solveSAT`.

Bei außergewöhnlichen Programmabbrüchen wird dem Benutzer eine Fehlermeldung mit den entsprechenden Fehlercodes auf der Konsole ausgegeben. Mit Hilfe der Datei `systemSignals.h` kann er dann den Grund noch näher spezifizieren.

Kapitel 9

Test der Implementation

9.1 Testansatz

Um den Algorithmus und die Implementation zu testen, wurden zwei Ansätze verfolgt. Bei dem ersten wurde nach jeder erfolglosen Suche ein neuer Überdeckungscode erzeugt, und bei dem zweiten wurde der Überdeckungsradius sukzessive erhöht. Diese Vorgehensweisen wurden unabhängig voneinander an identischen Formeln und mit unterschiedlichen, jeweils zur Laufzeit erstellten Überdeckungscodes getestet.

Hierfür wurden mehrere Shell Skripte geschrieben:

- Je ein Skript, das in einer Schleife insgesamt 50 Wiederholungen des Lösungsskripts aufruft.
- Je ein Lösungsskript, das zu Beginn einen Überdeckungscode erzeugt und diesen in eine Datei schreibt. Anschließend wird in einer Schleife für jede gespeicherte Formel — insgesamt 1000 Stück — mit dem vorher erstellten Überdeckungscode nach Lösungen gesucht. Die Ergebnisse werden in Logfiles geschrieben, die für jeden Aufruf des Lösungsskripts neu erstellt[1] werden.
- Skripte, die die Logfiles für die Auswertung in `csv`–Dateien[2] umwandeln.

9.2 Verwendeter Rechner

Getestet wurden die Programme auf dem Rechner `s241.math.uni-goettingen.de` der Mathematischen Fakultät der Universität Göttingen. Er ist für Inhaber eines

[1] Der Dateiname setzt sich aus einer 12-stelligen Zahl gefolgt von einem Unterstrich und dem Namen des Testskripts zusammen. Die Zahl wird gebildet aus Jahr, Monat, Tag, Stunde, Minute und Sekunden. Sollte ein Testdurchlauf innerhalb einer Sekunde fertig gestellt sein, so würde der nächste an die Datei hinten angehängt werden — ein Datenverlust kommt somit nicht vor.

[2] `csv` steht für „comma separated values“. `csv`–Dateien sind Textdateien, die einfach strukturierte Daten enthalten.

entsprechenden CIP–Accounts des Instituts für Numerische und Angewandte Mathematik zugänglich und benutzbar. Technisch ist diese Maschine mit 4 „Dual Core AMD Opteron(tm) Processor 275“ ausgerüstet. Diese Prozessoren haben 1800 MHz, einen 1024 KB Cache sowie 64bit Adressierungsbreite. Der Arbeitsspeicher ist 4GB groß.

9.3 Auswertung

Die Testläufe haben gezeigt, dass der Algorithmus

1. Lösungen findet,
2. und somit die zur Laufzeit erstellten Überdeckungscodes hinreichend gut sind, um mindestens eine Lösung zu finden.

Dass tatsächlich eine Lösung gefunden wurde, überprüft das Programm `solveSAT`, nachdem die Rekursion mit `localSearch` abgeschlossen wurde, indem es das durch `localSearch` gefundene Lösungswort an der Formel testet. Erst wenn dieser Test positiv ausfällt, wird die Lösung auch als solche gewertet und in das Logfile geschrieben.

Dass die zur Laufzeit erstellten Überdeckungscodes hinreichend gut sind, ist einleuchtend, da sowohl bei Testansatz 1 als auch bei Testansatz 2 für jede Formel nur einmal nach Lösungen gesucht wurde. Es war somit weder bei Testansatz 1 mehr als ein Überdeckungscode notwendig noch bei Testansatz 2 ein Expandieren des Radius.

Beispieldateien für die Logfiles der Testansätze sowie die Tabellenkalkulationen der Auswertung sind auf der CD-ROM gespeichert. Erstere aufgrund ihrer Anzahl und Größe als `zip`-Dateien.

9.4 Zeitumfang der Tests

Da die Testdurchläufe zwar gleichzeitig gestartet wurden, der verwendete Rechner jedoch über mehrere Prozessoren und Hyperthreading–Technologie verfügt, war die einzige von den Tests geteilte Systemressource der Arbeitsspeicher. Insofern kann von einer nahezu „Share–Nothing–Architektur“ für die Tests gesprochen werden.

Der Zeitumfang belief sich auf insgesamt 31 Stunden, 51 Minuten und 46 Sekunden für den Test mit expandiertem Radius, und 31 Stunden, 51 Minuten und 42 Sekunden für den Test mit stets neu bestimmtem Code. Der Unterschied von 4 Sekunden kommt mit Sicherheit daher, dass einerseits die Programme gleichzeitig auf dem selben Rechner gestartet wurden, andererseits aber auf die selben Formeldateien zugriffen. Deshalb kann er wohl vernachlässigt werden.

Der Zeitunterschied spielt bei der Bewertung der Zeitdauer auch deshalb keine große Rolle, da beide Testansätze eine Durchschnittszeit von 38 Minuten und 13 Sekunden pro Durchlauf benötigt haben. Das ergibt eine Durchschnittsdauer von 2,293 Sekunden für jede Formel. Ein zeitlicher Rahmen, der dem Programm bei den vorliegenden Gegebenheiten „Alltagstauglichkeit“ bescheinigt.

Der gleiche Test wurde noch einmal durchgeführt, jedoch mit nicht „geteilten“ Formeldateien. Dies hat keine Beschleunigung gebracht und soll auch nicht weiter Beachtung finden.

Kapitel 10

Zusammenfassung und Ausblick

10.1 Zusammenfassung

Codierungstheoretische Ergebnisse

Am häufigsten werden in der Codierungstheorie binäre Eingabealphabete untersucht und verwendet. Mit dieser Arbeit wurde gezeigt, dass viele Forschungsergebnisse für binäre Alphabete auch für q–näre Alphabete Gültigkeit haben.

Insbesondere seien hier die Ergebnisse bezüglich der Erstellung und Größe der Überdeckungscodes aus Kapitel 3 hervorgehoben. Gerade die Methode, die Überdeckungen zu zählen, kann auf dieser Arbeit aufbauend noch weiter generalisiert werden.

Algorithmus

Diese Arbeit hat gezeigt, dass der eingangs erwähnte Algorithmus nicht nur bei binären SAT–Problemen einsetzbar ist, sondern sich auch dafür eignet, Lösungen mehrwertiger Logik zu suchen und vorhandene zu finden.

Bei den Recherchen hat sich ergeben, dass die wenigsten Algorithmen mehrwertige Logik deterministisch lösen; dieser Algorithmus ist ein Weg deterministisch mit adäquatem Zeitaufwand vorzugehen.

Programmierung

Die Programmierung des Algorithmus für binäre k–SAT–Formeln erfolgte in der Sprache `C`. Dabei wurden nur Standardbibliotheken für die Ein- und Ausgabe auf

den Bildschirm, für das Lesen und Schreiben von Dateien, für mathematische Funktionen, Zeitangaben und Zufallsalgorithmen verwendet. Das Parsen des Programmaufrufs oder der Eingabedateien sowie das Schreiben der Inhalte der Dateien wurden vollständig selbst entwickelt, dem gegebenen Algorithmus angepasst und umgesetzt.

Bekannte Einschränkungen

Mehrere Einschränkungen der Implementation sind bekannt und könnten bei späteren Erweiterungen aufgehoben werden:

- Sind Dimension und Radius zu groß, erreichen die Binomialkoeffizienten bei der Volumenberechnung Werte, die größer sind als die durch die Rechnerarchitektur vorgegebenen Schranken. Dies führt zu einem erzwungenen und kontrollierten Programmabbruch mit entsprechender Fehlermeldung.
- Das Programm `solveSAT` ist nur geeignet, um k–SAT–Formeln einzulesen und zu lösen. Andere Bool'scheFormeln werden nicht unterstützt.
- Der Algorithmus ist prinzipiell nicht geeignet um *alle* Lösungen einer SAT–Formel zu finden. Er ist lediglich in der Lage, $\#\mathcal{C}$–viele zu finden und zu verifizieren. Zusätzlich prüft das Programm nicht, ob eine gefundene Lösung nicht schon in einem anderen Rekursionslauf gefunden wurde. Es wird davon ausgegangen, dass die Zahl der mehrfach überdeckten Wörter[1] aus dem $\mathcal{Q}^n$ entsprechend gering ist.
- Die tatsächliche Minimaldistanz eines Codes berechnet sich mit Laufzeit $\mathcal{O}((\#\mathcal{C})^2)$, wenn $\mathcal{C}$ der Überdeckungscode ist. Es gibt zwar Algorithmen, die effizienter sein mögen, doch wurde dieser Aspekt als für diese Arbeit nebensächlich angesehen und nicht weiter verfolgt.

Ein weiteres Problem liegt darin, dass der von `createCode` erzeugte Überdeckungscode formatiert nach `stdout` ausgegeben und von `solveSAT` direkt mittels eines gepufferten Datenstroms eingelesen werden kann. Dabei ist jedoch eine weitere Kommunikation zwischen Benutzer und Programm ausgeschlossen, da der gepufferte Datenstrom mit einem `EOF` abschließt und `solveSAT` somit auf `stdin` anschließend nicht mehr zugreifen kann. Dieses Problem liegt aber nicht an der Implementierung der Programme an sich, sondern beim gepufferten Datenstrom . Dies hat zur Folge, dass die Benutzerkommunikation im Programm `solveSAT` anschließend ausgeschaltet ist. Ausgaben sind allerdings in gewohnter Art und Weise möglich.

Stärke der Programmierung

Das Programm `createCode` erstellt Überdeckungscodes beliebiger Basis und Dimension. Diese Werte müssen lediglich vom Benutzer übergeben werden. Dies kann entweder durch Übergabe der Werte beim Programmaufruf, durch eine Eingabedatei

[1] siehe Lemma 3.17 aus Kapitel 3.3

oder durch Kommunikation des Benutzers mit dem Programm zur Laufzeit erfolgen. Der erstellte Überdeckungscode wird anschließend in eine Datei geschrieben, die entweder automatisiert erzeugt wird, oder beim Programmaufruf vorgegeben wurde.

Das Programm `solveSAT` ist unabhängig von der Art und Weise, wie der Überdeckungscode erstellt wurde. Wichtig ist, dass es den Code in dem in Kapitel 7 beschriebenen Format übergeben bekommt. Im Anschluss kann durch eine einfache Abfrage der Basis entschieden werden, ob die Funktion `localSearch` für binäre oder q–näre Formeln verwendet wird. Derzeit ist nur die Unterstützung für binäre k–SAT–Formeln implementiert.

10.2 Ausblick

Weitere codierungstheoretische Forschungsansätze

Diese Arbeit behandelt einige Aspekte der Codierungstheorie bezüglich Überdekkungscodes. Insgesamt ist der Stand der Forschung weit umfangreicher. Hier könnte die Arbeit auf jeden Fall noch weitergeführt und ergänzt werden. So wurde zum Beispiel bei der Methode, die Überdeckungen zu zählen („method of counting excess“), nur eine untere Schranke für Überdeckungscodes betrachtet und vom binären auf den q–nären Fall übertragen.

Auch im Bereich der Konstruktion des Überdeckungscodes gibt es noch zahlreiche Ansätze, die auf den q–nären Fall übertragen werden könnten.

Weiterentwicklungen der Programme `createCode` und `solveSAT`

Generelle Ergänzung

Die Programmierung von `solveSAT` setzt voraus, dass ein Überdeckungscode entweder schon formatiert in eine Datei geschrieben wurde oder formatiert über `stdin` eingelesen werden kann.

Das Programm `createCode` erstellt mit gegebenen Daten einen randomisierten Überdeckungscode. Dessen Güte bezüglich der tatsächlichen Raumüberdeckung hängt sehr stark von der Dimension und dem Radius ab. Das bedeutet, je größer die Dimension bei eher niedrigem Radius, umso besser die Überdeckung. Folglich ist ein Programm, das einen bestmöglichen Überdeckungscode zur Verfügung stellt, hilfreich. Da es für die Kompatibilität mit `solveSAT` auf die formatierte Ausgabe der Codewörter ankommt, kann hierfür auch ein Computer Algebra System (*CAS*), wie zum Beispiel MuPAD, genutzt werden.

Erweiterung von `localSearch` auf q–näre Alphabete

`localSearch` ist für das Lösen binärer k–SAT–Formeln implementiert. Der Algorithmus aus Kapitel 5 ist programmiertechnisch noch nicht umgesetzt. Hier sind noch Möglichkeiten, das Programm `solveSAT` weiter auszubauen.

Literaturverzeichnis

[1] Cong Liu, Andreas Kuehlmann, Mattew W. Moskewicz
CAMA: A Multi-Valued Satisfiability Solver
ICCAD '03, November 11–13, 2003, San Jose, California, USA

[2] G. J. M. van Wee
Improved sphere bounds on the covering radius of codes.
IEEE Trans. Inform. Theory 34, 237–245 (März 1988)

[3] G. D. Cohen, S. N. Litsyn, A. C. Lobstein, H. F. Mattson Jr.
Covering Radius 1985 – 1994
AAECC 8, 173–239 (1997)

[4] Evgeny Dantsin, Andreas Goerdt, Edward A. Hirsch, Ravi Kannan
Jon Kleinberg, Christos Papadimitriou, Prabhakar Raghavan, Uwe Schöning
A Deterministic $(2 - \frac{2}{k+1})^n$ Algorithm for *k*-SAT Based on Local Search
Theoretical Computer Science 289/1, 2002, pp. 69-83

[5] Robert Ash
Information Theory
1965

[6] Marek Karpinsky, Alexander Zelikovsky
Approximating Dense Cases of Covering Problems
Bonn, 1997

[7] Dorit S. Hochbaum
Approximation algorithms for NP-hard problems
1997

[8] DIMACS – The Center for Discrete Mathematics and Theoretical Computer Science, Rutgers University
Satisfiability Suggested Format
8. Mai 1993,
ftp://dimacs.rutgers.edu/pub/challenge/satisfiability/doc/satformat.dvi

Anhang A

Notationen und Begriffe

In diesem Anhang sollen die verwendeten Notationen und Begriffe erklärt bzw. aufgelistet werden.

$\mathcal{Q}$: $\mathcal{Q}$ steht für das gegebene Alphabet. Seine Elemente sind die möglichen Werte der einzelnen multivalued Variablen und werden mit σ bzw. σ_i bezeichnet.

$\mathcal{P}(\mathcal{Q})$: $\mathcal{P}(\mathcal{Q})$ bezeichnet die Teilmenge der Potenzmenge von $\mathcal{Q}$, deren Elemente die möglichen Wertemenge der Literale sind.

$\mathcal{X}$: $\mathcal{X}$ bezeichnet die Menge der mehrwertigen Variablen. Mit x_i wird eine einzelne Variable aus $\mathcal{X}$ bezeichnet. (Dabei handelt es sich um die i–te Variable eines n–dimensionalen Raumes.)

Literalwertemenge: Als *Literalwertemenge* v_i bezeichnet man die Menge an Elementen des Alphabets $\mathcal{Q}$, deren Werte die zugehörige Variable annehmen kann. Kommt dieselbe Variable in mehreren Klauseln einer Formel vor, so kann die Literalwertemenge von Klausel zu Klausel unterschiedlich sein.

mehrwertiges Literal: Eine Zuweisung einer Literalwertemenge an eine einzelne multivalued Variable x_i bezeichnet man als *mehrwertiges Literal* $x_i^{v_i}$.

$\mathcal{L}(\mathcal{X})$: $\mathcal{L}(\mathcal{X})$ bezeichnet die Menge der mehrwertigen Literale mit Variablen aus $\mathcal{X}$ und Wertemengen aus $\mathcal{P}(\mathcal{Q})$.

$\mathcal{Q}^n$: $\mathcal{Q}^n$ bezeichnet den n–dimensionalen Raum über dem Alphabet $\mathcal{Q}$. Seine Elemente, auch Wörter genannt, haben die Gestalt $\sigma = \sigma_1\sigma_2\sigma_3 \ldots \sigma_n$ und bilden so vollständige Zuweisungen („*full assignments*“) für die Variablen aus $\mathcal{X}$.

Hamming–Distanz: Mit $d_H(a,b) = \#\{i \mid a_i \neq b_i\}$ bezeichnet man die Hamming–Distanz von a und b aus $\mathcal{Q}^n$.

q-näre Entropiefunktion: $H_q(\rho)$ bezeichnet die q-näre Entropiefunktion. Sie misst in der Codierungstheorie den Informationsgehalt einer Nachricht.

$B_r(a)$: $B_r(a)$ ist eine Umgebung um ein ausgewähltes Wort $a \in \mathcal{Q}^n$ mit Radius r.

Volumen einer Kugel / eines Balls: $V_q^n(a,r)$ ist das Volumen einer Kugel oder eines Balls um einen Punkt a aus dem $\mathcal{Q}^n$ mit Radius r. Für die Elemente b innerhalb der Kugel gilt: $d_H(a,b) \leq r$.

$\mathcal{C}$: $\mathcal{C}$ bezeichnet einen Code. Dieser kann auf verschiedene Art und Weise gebildet werden. Beispielsweise als Zufallscode oder mit dem Greedy Algorithmus.

Minimaldistanz: Die Minimaldistanz eines Codes $\mathcal{C}$ ist die minimale Hamming–Distanz zweier verschiedener Codewörter aus $\mathcal{C}$.

Kugelpackungsschranke: Siehe Hammingschranke.

Hammingschranke: Die Hammingschranke ist die maximale Anzahl an Codewörtern über einem Alphabet bei gegebener Minimaldistanz.

Kugelüberdeckungsschranke: Die Kugelüberdeckungsschranke ist die minimale Anzahl an Codewörtern, die benötigt wird, um einen Such– bzw. Eingaberaum mit gegebenem Radius zu überdecken.

perfekter Code: Codes, die die Hammingschranke mit Gleichheit erfüllen, werden perfekte Codes genannt.

method of counting excess: Möglichkeit, die Anzahl der Wörter aus dem $\mathcal{Q}^n$ zu bestimmen, die von mehr als einem Codewort aus $\mathcal{C}$ mit gegebenem Überdekkungsradius r überdeckt werden.

Anhang B

Die CD–ROM

Dieser Arbeit ist eine CD–ROM mit folgendem Inhalt beigelegt:

- der Arbeit als pdf–Dokument,
- den Quellcodedateien einschließlich eines Makefiles für die Kompilierung und eines Moduldiagramms,
- den Shell Skripten um die Programme zu testen,
- Ergebnissen der Testdurchläufe,
- Shell Skripten um die Logfiles der Testdurchläufe in weiterverarbeitbare `csv`–Dateien umzuwandeln und
- den verwendeten 3–SAT–Formeln als `tar`–Datei.

Bei den Shell Skripten ist zu beachten, dass Pfadangaben explizit angegeben sind und von jedem Benutzer persönlich angepasst werden müssen. Dies ist durch die Benutzung von entsprechenden Variablen pro Skript nur an einer Stelle am Anfang notwendig.

Außerdem ist die `tar`–Datei vor Benutzung zu entpacken. Dies geschieht nicht automatisiert zur Laufzeit. Alternativ können sie mit entsprechenden `tar`–Befehlen auch direkt aus der Datei ausgelesen werden. Dies wurde jedoch bisher nicht getestet. Es kann nicht ausgeschlossen werden, dass dabei „Artefakte“ entstehen, die zu Fehlern beim Parsen der Datei durch `solveSAT` führen.

Die Diplomarbeit und alle dazu gehörigen Dateien und Informationen sind auch unter `www.mtpcvsh.de/mvsat` zu finden. Weiterentwicklungen und zusätzliche Informationen werden auch dort veröffentlicht werden.