TECHNISCHE UNIVERSITÄT WIEN

DIPLOMARBEIT

Die probabilistische Methode in der Kombinatorik

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Finanz- und Versicherungsmathematik

eingereicht von

Daniel Linzmayer BSc
Matrikelnummer 01025807

ausgeführt am Institut für Stochastik und Wirtschaftsmathematik
der Fakultät für Mathematik und Geoinformation der Technischen Universität Wien

Betreuung
Betreuer: Ao.Univ.-Prof. DI. Dr.techn. Karl Grill

Wien, 22.03.2018 _____ (Unterschrift Verfasser) _____ (Unterschrift Betreuer)

Zusammenfassung

Ziel dieser Arbeit ist die Lösung kombinatorischer Probleme unter Einsatz von probabilistischen Methoden, insbesondere Simulated Annealing. Probabilistische Algorithmen verwenden einen Zufallsgenerator, um das Vorgehen zu bestimmen, und sind somit bei Problemen schneller, deren Definitionsraum zu groß für eine erschöpfende Suche ist. Es werden Analysen durchgeführt, unter welchen Bedingungen diese Methoden eine Lösung approximieren. Simulated Annealing ist am Abkühlprozess aus der Metallurgie angelehnt, bei dem ein Stoff stark erhitzt und dann langsam abgekühlt wird, um eine möglichst optimale Beschaffenheit zu erzielen. Anhand von zwei kombinatorischen Fragestellungen werden diese Methoden auch in Laufzeit und Güte der Lösung untersucht. Die Erste ist das Property B Problem. Von einer Grundmenge mit endlicher Anzahl an Objekten hat eine Familie von Teilmengen Property B, falls diese Familie mit jeder beliebigen Partition der Grundmenge nichtleeren Durchschnitt hat. Gesucht wird die kleinstmögliche Familie, sodass diese Eigenschaft nicht gilt. Das zweite Problem ist die Suche nach Covering Codes. Hierbei wird in einem Coderaum ein Code gesucht, sodass alle Codewörter des Coderaumes höchstens Abstand R von einem Codewort dieses Codes haben. Dazu wurden zwei Lösungsverfahren durch Simulated Annealing in C realisiert.

Inhaltsverzeichnis

1 Einführung

Diese Arbeit untersucht, wie kombinatorische Probleme anhand von zwei Fragestellungen mittels probabilistischen Methoden gelöst werden können. Probabilistische Algorithmen finden heutzutage großen Einsatz in vielen Problemen der Praxis. Ein Algorithmus ist eine Vorschrift, die vorgibt wie ein Problem gelöst werden kann. Bei einem deterministischen Algorithmus ist das Ergebnis stets gleich, wenn an den Anfangsbedingungen nichts geändert wird. Probabilistische Algorithmen verwenden einen Zufallszahlgenerator, um das Vorgehen zu randomisieren. Die Kombinatorik beschäftigt sich mit Fragen wie: "Wie viele Farben werden zur Einfärbung einer beliebigen Landkarte benötigt?", "Wie kann der Transport von unterschiedlich großen Kisten mit möglichst geringem Aufwand organisiert werden." und viele weitere. Die klassische Herangehensweise an diese Probleme sind deterministische Programme, die stets die optimale Lösung suchen und dabei nicht vom vorgegebenen Weg abweichen. Dies bedeutet aber, dass ein Problem mit gleicher Startkonfiguration immer derselben Lösung zugeführt wird. Im Gegensatz dazu stehen die probabilistischen Methoden, die mittels Zufallsvariablen nach einer Lösung suchen. Das Grundprinzip ist, dass nicht der ganze Definitionsraum abgetastet wird, sondern zufällig Kandidaten für eine Lösung erstellt und getestet werden. So werden Lösungen gefunden, welche den optimalen Wert approximieren. Viele der heutigen Problemstellungen werden mit probabilistischen Methoden gelöst, um den Rechenaufwand zu reduzieren, da es in der Praxis meist ausreichend ist, eine annehmbar gute Lösung zu haben. Zwei Probleme aus der Kombinatorik werden in dieser Arbeit mithilfe eines probabilistischen Algorithmus gelöst. Bei dem Algorithmus handelt es sich um Simulated Annealing, welches durch die Metallurgie inspiriert wurde. Ein Metall wird stark erhitzt, sodass alle Moleküle in freier Bewegung sind und den energieärmsten Zustand erreichen können. Anschließend wird der Stoff sehr langsam abgekühlt um den Zustand zu erhalten. Dieses Verfahren kann in der Mathematik zur Approximation eines Minimums oder Optimums eingesetzt werden. Die zwei kombinatorischen Probleme, welche mit Simulated Annealing behandelt werden, sind zum einen die Suche nach Covering Codes, zum anderen die Suche nach einer Mengenfamilie, welche Property B nicht erfüllt. Für diese beiden Probleme wird ein Simulated Annealing Algorithmus in C realisiert, die Ergebnisse verglichen und analysiert.

Kapitel 2 befasst sich mit der mathematischen Präzisierung von Algorithmen und Komplexität. Intuitiv sind diese Begriffe sehr schnell klar, aber eine genaue Definition erweist sich als durchaus komplex. Hierbei ist das Konzept der Turingmaschine hilfreich. Weiters werden die Komplexitätsklassen $\mathcal{P}$, $\mathcal{NP}$ und $\mathcal{BPP}$ definiert.

Kapitel 3 sammelt die wahrscheinlichkeitstheoretischen und stochastischen Grundlagen, auf denen der Rest der Arbeit aufbaut.

In Kapitel 4 werden verschiedene probabilistische Algorithmen vorgestellt. Beginnend mit dem Einfachsten, der Brute Force Suche, werden die Verfahren zunehmend verfeinert, und spannen so einen intuitiven Bogen über die Lokale Suche und den MCMC-Algorithmen hin zum Simulated Annealing.

Kapitel 5 ist völlig dem Simulated Annealing gewidmet. Zunächst wird die Motivation des Verfahrens und das Analogon aus der Werkstoffkunde beschrieben. Danach wird das Verfahren mathematisch formuliert und Bedingungen und Voraussetzungen definiert, unter welchen der Algorithmus auch eine gewünschte Lösung approximiert. Abschließend werden Methoden und Ansätze präsentiert, mit denen das Verfahren praktisch umgesetzt werden kann.
Kapitel 6 bietet eine Einführung in die Covering Codes und legt alle theoretischen Grundlagen fest, mit denen später die Suche nach Covering Codes durch Simulated Annealing erklärt werden.

Kapitel 7 dient der kurzen Definition der wichtigsten graphentheoretischen Aussagen.
Kapitel 8 formuliert anschließend mithilfe dieser Definitionen die Property B und leitet einige Schranken her.

Kapitel 9 beschreibt den in C realisierten Simulated Annealing Algorithmus für Covering Codes und seine genaue Funktionsweise. Ferner werden für einige Dimensionen Berechnungen angestellt und die Ergebnisse verglichen. Dabei werden besonders Laufzeit, sowie die Approximationsgüte untersucht und für verschiedene Fälle verglichen.

Kapitel 10 verläuft analog zu Kapitel 9 für Property B. Auch hier wird der realisierte Algorithmus und seine Arbeitsweise detailliert beschrieben. Anschließend werden für den Fall $k = 5$ einige Spezialfälle untersucht und bestehende Schranken verbessert. Das Kapitel schließt ähnlich dem Kapitel 9 mit einem Vergleich der Laufzeiten des Programms unter verschiedenen Eingabeparameter.

2 Algorithmen und Berechenbarkeit

Ein Algorithmus ist eine in Schritten strukturierte Vorschrift, welche ein gestelltes Problem löst. Als Problem wird hier stets eine Diskrepanz zwischen dem aktuellen Zustand und einem Zielzustand bezeichnet. Ein klassisches Beispiel für einen Algorithmus ist das Kochrezept. Hier führt eine klar nacheinander abzuarbeitende Abfolge von Tätigkeiten zu einem fertigen Gericht. Philosophische Probleme wie: "Was ist der Sinn des Lebens?", seien hier ausgeklammert, da weder gesichert ist, ob eine Antwort existiert, noch nachprüfbar ist, ob sie zutrifft, da die richtige Antwort auch völlig Subjektiv sein kann.
In dieser Arbeit ist ein Problem immer die Suche nach einer Lösung, die als solche identifizierbar und auch nachprüfbar ist. Für solche Probleme lässt sich also ein Algorithmus konstruieren, der das Problem in endlich vielen Schritten lösen kann. Diese Probleme nennt man berechenbar (die saubere mathematische Definition folgt im Verlauf des Kapitels). Im Gegensatz zu den oben genannten philosophischen Problemen stehen sehr klar definierte und konkret nachvollziehbare Probleme, die nachweislich nicht berechenbar sind, also ein klein Algorithmus gefunden werden kann.
Dieses Kapitel soll diese intuitiven Konzepte strenger definieren und gibt somit einen Überblick über Berechenbarkeitstheorie und Komplexitätsklassen. Der Begriff "Komplexität eines Problems" soll hier quantifizierbar werden. Ferner sollen deterministische und probabilistische Algorithmen untersucht und die Leistungsfähigkeit solcher Algorithmen gegenübergestellt werden. Für detailliertere Ausführungen der hier präsentierten Konzepte sei auf [Win02] [DK00] und [Rot08] verwiesen.

2.1 Berechenbarkeitstheorie

Um mit Algorithmen auf einer theoretischen Grundlage arbeiten zu können, wurde von Alan Turing das Konzept der Turingmaschine entwickelt. Die Turingmaschine konkretisiert das abstrakte Konzept eines Algorithmus. Die Turingmaschine besteht aus einem unendlich langen Band mit Feldern, auf denen jeweils ein Symbol aufgedruckt werden kann. Das Band kann beliebig oft überschrieben werden. Dieses Band ist in einem Mechanismus eingefasst, der das Band um ein Feld vorwärts oder zurück bewegen kann. In der Mitte dieses Mechanismus ist ein Lese- und Schreibkopf befestigt, der ein Feld des Bandes auslesen kann und es mit einem Symbol beschreiben kann.

Definition 2.1.1. *[Win02] S. 6*
Die Turingmaschine
Eine Turingmaschine $\mathbb{T} = [\Sigma, Q, f, g, h, q_0, q_E]$ *besteht aus:*

- $\Sigma \neq \emptyset$ *einem endlichen Alphabet mit Leerzeichen*
- $Q \neq \emptyset$ *einer endlichen Zustandsmenge mit* $q_0 \in Q$ *als Anfangszustand und* $q_E \in Q$ *als Endzustand.*
- *f:* $Q \times \Sigma \rightarrow Q$ *der Zustandsüberführungsfunktion*
- *g:* $Q \times \Sigma \rightarrow \Sigma$ *der Schreibfunktion*
- *h:* $Q \times \Sigma \rightarrow \{-1, 0, 1\}$ *einer Transportfunktion*

f,g und h bilden zusammen die Überführungsfunktion
$\delta : Q \times \Sigma \rightarrow Q \times \Sigma \times \{-1, 0, 1\}$

Die Funktion f bestimmt in jedem Schritt, in welchem Zustand der Kopf sein soll, g definiert, welches Symbol geschrieben wird, und h bestimmt, wie das Band verschoben wird.
In den meisten Fällen liest eine Turingmaschine das Symbol auf dem aktuellen Feld aus, beschreibt es, falls gefordert und wechselt dann zum nächsten bzw. vorherigen Feld. Dort liest es wieder das aktuelle Symbol aus und überschreibt es. Dieser Prozess wird solange weitergeführt, bis die Maschine hält. Wann sie welche Felder ausliest, beschreibt und zu welchem Zeitpunkt sie hält, ist von der Überführungsfunktion abhängig.

Eine Turingmaschine kann auch lesen ohne zu schreiben bzw. mehrere Felder beschreiben bevor sie wieder ein Feld ausliest. Die Überführungsfunktion für die Turingmaschine gleichsam ein Programm.

Definition 2.1.2. *[Win02] S.9*
Eine Funktion ist ***berechenbar****, falls eine Turingmaschine konstruiert werden kann, welche die Funktion in endlich vielen Schritten berechnen kann.*

Bei einer berechenbaren Funktion wird gefordert, dass die Turingmaschine irgendwann zu einem Ende kommt (hält) und nicht unendlich lang rechnet. Die Frage, ob und bei welcher Eingabe eine Turingmaschine hält, hängt mit dem Begriff der Entscheidbarkeit zusammen.
Das Alphabet Σ beinhaltet alle Symbole, mit denen die Eingabe und Ausgabe bewältigt wird. Klassische Beispiele für ein Alphabet sind die binären Zahlen, mit denen auch ein moderner Computer operiert. Mit Σ^* wird die Menge aller Wörter die aus Σ konstruierbar sind bezeichnet. Dies bedeutet, dass jedes Problem eine Teilmenge dieser Menge ist, also $P \subseteq \Sigma^*$.

Definition 2.1.3. *[Win02] S.12*

Ein Problem $P \subseteq \Sigma^*$ heißt entscheidbar, falls die Funktion

$$1_P(p) = \begin{cases} 0 & \textit{falls } p \in P \\ 1 & \textit{falls } p \notin P \end{cases} \tag{2.1.1}$$

berechenbar ist.

Im Zusammenhang mit diesen Definition steht die Church-Turing-These, welche besagt:

Satz 2.1.1. *[Win02] S.17*
Die Menge der berechenbaren Funktionen entspricht der Menge der intuitiv berechenbaren Funktionen

Das Problem an dieser These ist der Begriff intuitiv, der nicht näher definiert wird, wodurch diese Aussage auch nicht beweisbar ist. Trotzdem ist diese These hilfreich, da sie die Menge an berechenbaren Problemen bereits gut einschränkt und Fragen wie: "Was ist der Sinn des Lebens", ausgeschlossen werden. Damit werden Begriffe wie Algorithmus, Problem und Lösung mathematisch fassbar.
Eine wichtige Frage der Berechenbarkeitstheorie ist das

Definition 2.1.4. *[Win02] S.53*
Halteproblem:
Zu einer vorgegebenen Turingmaschine und Eingabe soll bestimmt werden, ob die Turingmaschine hält.

Dies wird beantwortet mit

Satz 2.1.2. *[Win02] S.54*
Das Halteproblem ist nicht entscheidbar.

Der Beweis ist in [Win02] S.54-55 zu finden.
Die Aussage ist weitreichend: Es kann keine Turingmaschine geben, die für eine beliebige Turingmaschine und Eingabe bestimmen kann, ob sie hält oder nicht.
Die bislang verwendete Turingmaschine wird auch deterministische Turingmaschine genannt, also eine Maschine, die nach einer Vorschrift alle Schritte seriell abarbeitet.
Dazu steht im Kontrast die nichtdeterministische Turingmaschine, die gleichzeitig in mehreren Zuständen sein kann und mehrere Symbole gleichzeitig lesen und/oder schreiben kann und dies gleichzeitig auf mehreren Positionen des Bandes.

Eine nichtdeterministische Turingmaschine kann also mehrere Operationen parallel durchführen.

Definition 2.1.5. *[Win02] S.18*
Eine nichtdeterministische Turingmaschine $\mathbb{T} = [\Sigma, Q, \delta, q_0, q_E]$ *besteht aus*

- *Dem endlichen Alphabet* $\Sigma \neq \emptyset$
- *Der endlichen Zustandsmenge* $Q \neq \emptyset$ *mit* q_0 *als Anfangszustand und* q_E *als Endzustand.*
- $\delta : Q \times \Sigma \to \mathfrak{P}^*(Q \times \Sigma \times \{-1, 0, 1\})$ *der Zustandsüberführungsrelation*

Wobei $\mathfrak{P}^*$ *die Potenzmenge ohne leere Menge bezeichnet.*

Unterscheidungsmerkmal der nichtdeterministischen Turingmaschine von der deterministischen ist die Fähigkeit Probleme schneller zu berechnen, und zwar aufgrund der Möglichkeiten mehrere Befehle gleichzeitig durchzuführen. Interessant und auf den ersten Blick vielleicht überraschend ist die Tatsache, dass jedes Problem, das eine nichtdeterministische Turingmaschine berechnen kann, auch von einer deterministischen berechnet werden kann. Dafür konstruiert man eine deterministische Turingmaschine, die alle parallel ausgeführten Befehle seriell ausführt. Für diese Konstruktion ist lediglich höhere Rechenzeit und mehr Speicherplatz am Band erforderlich. Die genaue Konstruktion ist in [Win02]S.20-21 zu finden. Bedingt durch die Definition gilt, dass alle durch eine deterministische Turingmaschine berechenbaren Probleme auch nichtdeterministisch berechenbar sind.
Die möglicherweise erstaunlichste Erkenntnis der Berechenbarkeitstheorie ist, dass jeder moderne PC genau diese Probleme berechnen kann die auch eine deterministische Turingmaschine berechnen kann. Auch für diesen Beweis sei auf [Win02] S.50 verwiesen. Es zeigt sich also, dass die deterministische Turingmaschine eine gute Basis ist, um Berechenbarkeits- und Komplexitätsanalysen durchzuführen.
In [Win02] S.61-62 wird die Existenz einer universellen Turingmaschine nachgewiesen, die alle Turing-berechenbaren Probleme lösen kann. Damit entfällt der Aufwand, für jedes Problem eine eigene Maschine zu konstruieren. Der moderne Computer kann als eine Realisierung dieser universellen Turingmaschine verstanden werden.

2.2 Komplexität

Nachdem in 2.1 geklärt wurde, was eine Maschine berechnen kann, ist nun die nächste Frage, wie "schwer" ein Problem ist. Der Schweregrad eines Problems besteht aus der Menge an Daten, die gespeichert werden müssen und Anzahl der Rechenoperationen. Somit wird zwischen Zeitkomplexität und Speicherplatzkomplexität unterschieden.
Für die Zwecke dieser Arbeit genügt es, lediglich die Zeitkomplexität zu untersuchen, da diese für die hier behandelten Probleme im Gegensatz zum mangelnden Speicherplatz die größere Schwierigkeit darstellt. Für den Rest des Kapitels wird also mit dem Begriff Komplexität stets die Zeitkomplexität bezeichnet.

Definition 2.2.1. *[Win02] S.158*
Zeitkomplexität T(n) ist die Anzahl an Schritten, auch Takte genannt, die eine Turingmaschine benötigt, um für ein Problem mit Länge $n \in \mathbb{N}$ eine Lösung zu finden bzw. eine Menge zu entscheiden.

Der Definitionsraum der Zeitkomplexität ist das Intervall $[1, \infty]$. Die Extremfälle sind zum einen das sofortige Halten der Turingmaschine und zum anderen, dass die Maschine nie hält. Da jedes Problem eine eigene Komplexität hat, empfiehlt es sich, alle bekannten Probleme in Komplexitätsklassen einzuteilen.

Definition 2.2.2. *Eine Komplexitätsklasse ist die Menge aller Probleme, die von einer Turingmaschine mit Komplexität $T(n) \leq f(n), n \in \mathbb{N}$ für hinreichend großes n gelöst werden können.*

Diese Schranke soll die asymptotische Entwicklung des Problems bei zunehmender Wortlänge eingrenzen. Bei heuristischer Betrachtung kann es also durchaus vorkommen, dass ein Problem für kleinere n den Eindruck erwecken kann, einer anderen Komplexitätsklasse anzugehören. Aus der Definition der Komplexitätsklasse folgt eine Kette an monoton wachsenden Komplexitätsklassen, bei der jede Komplexitätsklasse alle kleineren vollständig enthält und selbst in der nächst größeren vollständig enthalten ist.

Satz 2.2.1. *Seien $\mathcal{X}$ und $\mathcal{Y}$ 2 Komplexitätsklassen mit den beschränkenden Funktionen f und g. Sei $f(n) \leq g(n)$ für $n \geq n_0$ wobei n_0 hinreichend groß. Dann gilt:*
$\mathcal{X} \subseteq \mathcal{Y}$

Beweis. P ein beliebiges Problem aus $\mathcal{X}$. Sei ferner T(n) die Komplexität von P. Es gilt auf Grund der Voraussetzung:

$$T(n) \leq f(n) \leq g(n)$$

Wodurch P auch ein Problem aus $\mathcal{Y}$ ist. □

Bedingt durch die Definition gibt es also sehr viele Komplexitätsklassen, für praktische Betrachtungen kommen jedoch einige wenige in Frage. Die beiden wichtigsten sind:

Definition 2.2.3. *$\mathcal{P}$ mit Funktion $f(n) = c \cdot n^k, k, c = konst.$ ist die Komplexitätsklasse aller Probleme, die von einer deterministischen Turingmaschine in polynomieller Zeit gelöst werden können.*

Definition 2.2.4. *$\mathcal{NP}$ mit Funktion $f(n) = c \cdot n^k, k, c = konst.$ ist die Komplexitätsklasse aller Probleme, die von einer nichtdeterministischen Turingmaschine in polynomieller Zeit gelöst werden können.*

Ein Problem in polynomieller Zeit zu lösen wird oft synonym zu dem Begriff, ein Problem "effizient" zu lösen, gebraucht. Die Definition von $\mathcal{NP}$ und $\mathcal{P}$ lassen folgenden Schluss zu:

$$\mathcal{P} \subseteq \mathcal{NP}$$

Ein Problem aus $\mathcal{P}$ kann auch durch eine nichtdeterministische Turingmaschine in polynomieller Zeit gelöst werden.
Die Frage nach der Umkehrung ist eines der großen ungelösten Probleme der Mathematik.

Die Lösung der Frage

$$\mathcal{P} \stackrel{?}{=} \mathcal{NP}$$

wird als so wichtig und relevant für die gesamte theoretische Informatik als auch die Lehre von Algorithmen generell angesehen, dass sie vom Clay Mathematics Institute in Cambridge, Massachusetts, USA mit zu den Milleniumproblemen aufgenommen wurde.
Die Milleniumprobleme sind eine Liste von sieben mathematischen Problemen, von denen zum Stand der Arbeit (2018) lediglich die Poincaré-Vermutung gelöst wurde. Für die Lösung eines dieser Probleme wird ein Preis von 1 Million US-Dollar ausgezahlt. Ironischerweise hat Grigori Perelman, dem die Lösung der Poincaré-Vermutung zugeschrieben wird, den Preis bis heute nicht angenommen.
Es wurde im Laufe der Zeit für viele Probleme nachgewiesen, dass sie in $\mathcal{NP}$ liegen. Das berühmteste Beispiel dafür ist

Definition 2.2.5. ***Das Problem des Handelsreisenden.***
Ein Händler will n Städte mit seinem Auto besuchen. Gibt es eine Route, die eine vorgegebene Länge N unterschreitet?

Eine strengere Variante ist auch die Suche nach der optimalen Route für n Städte. Die Lösung dieses Problems ist für wenige Städte noch überschaubar, der Aufwand steigt jedoch exponentiell mit jeder hinzukommenden Stadt, da es für n Städte n! viele Möglichkeiten gibt, eine Route zu planen.
Wird ein Vorschlag zu einer Route gemacht, ist sehr leicht nachzuprüfen, ob diese den Kriterien genügt. Dazu müssen lediglich die Teilstrecken zwischen den Städten aufaddiert und mit N vergliechen werden. Somit liegt das Nachprüfen einer Strecke in $\mathcal{P}$.
Weitere Probleme dieser Art sind das Erstellen eines Stundenplans für einen Studenten (schwierig zu berechnen, da Überschneidungen vermieden werden müssen, die jedoch beim Nachprüfen leicht identifiziert werden können), die Suche nach Covering Codes sowie dem 2-Färbe-Problem bzw. Property B. Die Covering Codes und Property B werden in dieser Arbeit behandelt.
Würde der Nachweis $\mathcal{P} = \mathcal{NP}$ gelingen, bedeutet dies, dass es für all die vorher genannten Probleme eine Lösungsmethode gibt, welche in der selben Komplexitätsklasse wie das Nachprüfen derselben liegt. Dies würde eine enorme Verbesserung aller momentanen Lösungsverfahren für $\mathcal{NP}$ in Aussicht stellen, wobei erwähnt sei, dass diese besseren Algorithmen erst gefunden werden müssen. Momentaner Stand der Forschung ist, dass deterministische Turingmaschinen Probleme aus $\mathcal{NP}$ in exponentieller Zeit lösen können.

2.3 BPP

Die Algorithmen, die in dieser Arbeit zum Einsatz kommen, sind alle probabilistischer Natur.
Eine Turingmaschine kann zu einer probabilistischen Turingmaschine ausgebaut werden, indem eine weitere Funktion eingeführt wird. Diese Funktion gibt vor jedem neuen Schritt ein zufälliges Symbol aus. Der Lese- und Schreibkopf geht nun nicht allein nach den Vorgaben des Programms und dem Symbol auf dem Band vor, sondern bindet auch das Zufallssymbol ein. Damit kann eine probabilistische Turingmaschine bei identischen Startbedingungen bei zwei Durchläufen zwei verschiedene Ergebnisse produzieren, im Gegensatz zur deterministischen Turingmaschine.

Definition 2.3.1. *$\mathcal{BPP}$ mit Funktion $f(n) = c \cdot n^k, k, c = konst.$ ist die Komplexitätsklasse aller Probleme, die von einer probabilistischen Turingmaschine in polynomieller Zeit mit einer Fehleranfälligkeit kleiner als 1/2 berechnet werden.*

Die Fehlerschranke kann willkürlich gewählt werden, solange sie echt kleiner als $1/2$ ist. Dadurch kann bei einem Problem aus $\mathcal{BPP}$ die probabilistische Turingmaschine beliebig oft angewendet werden, um die Fehleranfälligkeit zu reduzieren. Dies entspricht auch der praktischen Interpretation von Monte-Carlo-Algorithmen, deren Ergebnisse mit wiederholter Anwendung bzw. größerer Stichprobe zunehmend genauer werden.

3 Stochastische Grundlagen

3.1 Wahrscheinlichkeitstheorie

Definition 3.1.1. *[Gri18] S.75*
Es sei

$$S_n = \sum_{i=0}^{n} X_i$$

die Partialsumme der X_i bis n. Dann ist

$$\bar{X}_n = \frac{S_n}{n}$$

das n-te Stichprobenmittel

Satz 3.1.1. *[Gri18] S.75-76*

Schwaches Gesetz der großen Zahlen
Sind die Zufallsvariablen $(X_n, n \in \mathbb{N})$ unabhängig und identisch verteilt mit endlicher Varianz, dann gilt:

$$\bar{X}_n \to \mathbb{E}[X_1] \tag{3.1.1}$$

Satz 3.1.2. *[Gri18] S.81*

Starkes Gesetz der großen Zahlen
X_n seien unabhängige Zufallsvariable mit:

$$\sum_{n\in\mathbb{N}} \frac{\mathbb{V}(X_n)}{n^2} < \infty$$

Dann gilt

$$\lim_{n\to\infty} \frac{1}{n} \sum_{k\leq n} (X_k - \mathbb{E}(X_k)) = 0 \tag{3.1.2}$$

fast sicher.

3.2 Markov Ketten

In diesem Abschnitt werden die Grundlagen von Markov Ketten erläutert. Details siehe [MGNR12] [Nor97] [Ros10].

Definition 3.2.1. *[Nor97]S.1-2*
Sei I eine abzählbare Menge, genannt Zustandsraum. $i \in I$ wird Zustand genannt. Für eine Zufallsvariable $X : \Omega \to I$ sei $\lambda_i := P(X = i)$ definiert. Falls $\sum_{i=0}^{\infty} \lambda_i = 1$ gilt, dann ist $\lambda = (\lambda_i : i \in I)$ eine Wahrscheinlichkeitsverteilung.

Die Zufallsvariable X nimmt also mit Wahrscheinlichkeit λ_i den Zustand i an.

Definition 3.2.2. *[Nor97]S.2*
Eine Matrix $P = (p_{ij} : i, j \in I)$ mit $p_{ij} \geq 0 \ \forall i, j$ wird stochastisch genannt, falls alle Zeilensummen 1 ergeben: $\sum_{j \in I} p_{ij} = 1$

Damit lässt sich bereits die Markov Kette definieren.

Definition 3.2.3. *[Nor97]S.2*
Eine Zufallsvariable $(X_n)_{n \geq 0}$ heißt Markov Kette mit Anfangsverteilung λ und stochastischer Matrix P (Übergangsmatrix genannt), falls gilt:

1. $P(X_0 = i_1) = \lambda_{i_1}$
2. $P(X_{n+1} = i_{n+1} | X_0 = i_1, \ldots, X_n = i_n) = p_{i_n i_{n+1}}$

Satz 3.2.1. *[Nor97]S.2-3*
Ein zeitdiskreter stochastischer Prozess $(X_n)_{0 \geq n \geq N}$ ist eine Markov Kette genau dann, wenn $\forall i_1, \ldots, i_N \in I$

$$P(X_0 = i_1, X_1 = i_2, \ldots, X_N = i_N) = \lambda_{i_1} p_{i_1 i_2} p_{i_2 i_3} \ldots p_{i_{N-1} i_N} \tag{3.2.1}$$

Eine Eigenschaft von Markov Ketten ist die Gedächtnislosigkeit:

Satz 3.2.2. *[Nor97]S.3*
Sei $(X_n)_{n\geq 0}$ eine Markov Kette mit Startverteilung λ und Übergangsmatrix P. Dann ist $(X_{m+n})_{n\geq 0}$ eine Markov Kette mit Startverteilung δ_i und Übergangsmatrix P, falls $X_m = i$. $(X_{m+n})_{n\geq 0}$ ist unabhängig von $X_0, \ldots, X_m$

Die Zukunft einer Markov Kette hängt nur von der Gegenwart ab, und nicht von der Vergangenheit.

Definition 3.2.4. *[LA87]S.13*
Sei $a_i(k)$ die Wahrscheinlichkeit, dass die Markov Kette nach k Schritten in den Zustand i gelangt. $a_i(k)$ kann rekursiv berechnet werden:

$$a_i(k) = \sum_l a_l(k-1) \cdot p_{li} \tag{3.2.2}$$

Definition 3.2.5. *[LA87]S.13*
Eine Markov Kette, deren Übergangswahrscheinlichkeiten zeitunabhängig sind, wird homogen genannt.
Falls sie nicht homogen ist, wird sie inhomogen genannt.

Eine Markov Kette kann in Kommunikationsklassen unterteilt werden.

Definition 3.2.6. *[Nor97]S.10*
Es führt i zu j, (im Zeichen: $i \to j$) falls

$$P_i(X_n = j \text{ für ein } n \geq 0) > 0$$

Also die Wahrscheinlichkeit, aus dem Zustand i in endlich vielen Schritten in den Zustand j zu kommen, positiv ist.
j und i kommunizieren miteinander, (im Zeichen $i \leftrightarrow j$) falls sowohl $i \to j$ als auch $j \to i$.

Definition 3.2.7. *[Nor97]S.11*
Eine Menge C von Zuständen wird Kommunikationsklasse genannt, wenn $\forall i, j \in C$ gilt $i \leftrightarrow j$.
Eine Kommunikationsklasse wird geschlossen genannt, falls sie nicht mehr verlassen werden kann, wenn sie einmal betreten wurde.

Definition 3.2.8. *[Nor97]S.24*
Ein Zustand i ist rekurrent, falls $P_i(X_n = i \text{ für unendlich viele } n) = 1$ gilt.
Ein Zustand i ist transient, falls $P_i(X_n = i \text{ für unendlich viele } n) = 0$ gilt.

Eine Markov Kette wird immer wieder zu einem rekurrenten Zustand zurückkehren, und einen transienten Zustand irgendwann nicht mehr erreichen.

Satz 3.2.3. *[Nor97]S.24*
Sei C eine Kommunikationsklasse. Dann sind in C entweder alle Zustände transient oder alle Zustände sind rekurrent.

Satz 3.2.4. *[Nor97]S.27*
Jede rekurrente Klasse ist geschlossen und jede endliche geschlossene Klasse ist rekurrent.

Definition 3.2.9. *[Nor97]S.11*
Eine Übergangsmatrix P, die aus einer einzigen Klasse besteht, wird irreduzibel genannt.
Eine Markovkette wird irreduzibel genannt, falls die zugehörige Übergangsmatrix irreduzibel ist.

Definition 3.2.10. *[MGNR12]S.194*
Eine Übergangsmatrix P wird aperiodisch genannt, falls

$$\forall i \in I : ggT(\{n > 0 | (P^n)_{i,i} > 0\}) = 1 \tag{3.2.3}$$

Eine Markovkette wird aperiodisch genannt, falls die zugehörige Übergangsmatrix irreduzibel ist.
P und die zugehörige Markovkette werden periodisch genannt, falls sie nicht aperiodisch sind und obiger $ggT > 1$ ist.

In einer irreduziblen Markovkette können keine kleineren geschlossenen Klassen gefunden werden, also kommuniziert jeder Zustand mit jedem anderen.
In einer periodischen Markovkette kann man einige Aussagen über die Markovkette treffen, wenn man den Startpunkt kennt. Bei einer aperiodischen Markovkette ist dies nicht möglich.

Definition 3.2.11. *[MGNR12]S.197-198*
Sei $P \in \mathbb{R}^{IxI}$ eine Übergangsmatrix. Ein Wahrscheinlichkeitsvektor $\mu \in \mathbb{R}^I$ mit

$$\mu = \mu \cdot P \tag{3.2.4}$$

heißt Stationäre Verteilung der Matrix P.

Definition 3.2.12. *[MGNR12]S.199*
Eine stochastische Matrix $P = (p_{i,j})$ heißt reversibel, falls ein Wahrscheinlichkeitsvektor $\mu \in \mathbb{R}^I$ existiert, der

$$\forall i, j \in I : \mu_i \cdot p_{i,j} = \mu_j \cdot p_{j,i} \tag{3.2.5}$$

erfüllt.

Diese Gleichung wird auch detailed-balance-Gleichung genannt.

Satz 3.2.5. *[MGNR12]S.200*
Zu jeder irreduziblen und aperiodischen Markov-Kette existiert eine eindeutige positive stationäre Verteilung

Der letzte und auch wichtigste Satz, der Ergodensatz, ist das Fundament auf dem die Theorie der MCMC-Verfahren aufbaut. Sei dazu $V_i(n) = \sum_{k=0}^{n-1} 1_{\{X_k=1\}}$ die Anzahl, wie oft X den Zustand i bis zum Zeitpunkt n besucht.

Satz 3.2.6. *[Nor97]S.53-54 Sei P irreduzibel und λ eine beliebige Verteilung. Falls $(X_n)_{n\geq 0}$ eine Markovkette mit Startverteilung λ und Übergangsmatrix P ist, dann gilt:*

$$\mathbb{P}\left(\frac{V_i(n)}{n} \to \frac{1}{m_i} \text{ für } n \to \infty\right) = 1 \tag{3.2.6}$$

mit $m_i = \mathbb{E}_i(T_i)$ als die erwartete Rückkehrzeit zum Zustand i. Im Fall dass m_i endlich ist, gilt für beliebige beschränkte Funktion $f : I \to \mathbb{R}$

$$\mathbb{P}\left(\frac{1}{n}\sum_{k=0}^{n-1} f(X_k) \to F \text{ für } n \to \infty\right) = 1 \tag{3.2.7}$$

mit $F = \sum_{i\in I} \pi_i f_i$ wobei $(\pi_i : i \in I)$ die eindeutige invariante Verteilung ist.

Der Satz garantiert, dass aperiodische und irreduzible Markov-Ketten gegen eine stationäre Verteilung konvergieren.

4 Stochastische Suchalgorithmen

Im diesem Kapitel werden Verfahren vorgestellt, mit denen ein Optimum einer Menge bzw. einer Funktion gefunden werden kann. Zur besseren Lesbarkeit wird bei der Vorstellung der Verfahren immer davon ausgegangen, dass ein Minimum gefragt ist. Es sei angemerkt, dass eine Maximumssuche analog erfolgt. Hierzu müssen nur die auftretenden Ungleichungen umgekehrt werden.

4.1 Brute Force Suche

Die Brute Force Suche ist eine universell einsetzbar, einfach zu implementierende, aber auch sehr ineffiziente Möglichkeit der Optimierung.
Sei $D \subset \mathbb{R}^n$ die Definitionsmenge einer Funktion $f : D \to \mathbb{R}^n$. Gesucht ist ein Minimum oder mehrere Minima der Funktion f. Bei der Brute Force Suche werden für alle Punkte $x \in D$ die Funktionswerte f(x) berechnet und verglichen. Wenn D komplett abgetastet wurde, kann der Punkt $x_u \in D$ identifiziert werden, für den $f(x_u)$ minimal ist. Damit diese Suche erfolgreich sein kann, muss der Definitionsbereich eine beschränkte Teilmenge von $\mathbb{N}$ oder $\mathbb{Z}$ sein. Ansonsten können nicht alle Punkte aus D in endlicher Zeit abgearbeitet werden und kein Resultat erzielt werden. Unter diesen Voraussetzungen findet die Brute Force Methode immer ein Minimum, sofern eines existiert.
Das große Problem dieser Methode ist die Laufzeit. Da relevante Probleme der Mathematik und Informatik große Definitionsbereiche haben, stößt dieses Verfahren sehr schnell an seine Grenzen. Im Kapitel Covering Codes wird beispielsweise versucht, aus 2048 Codes 68 zu finden, die gewisse gewünschte Eigenschaften erfüllen. Dies ist mit Brute Force vollkommen unrealistisch, da $\binom{2048}{68} \sim 1.9387 \cdot 10^{128}$ Kombinationen getestet werden müssten. Trotzdem wird die Brute Force Suche in vielen Fällen, in denen kein besseres Verfahren bekannt ist, verwendet.
Eine Alternative ist die stochastische Brute Force Suche. Hierbei wird der Raum nicht nach einer Ordnung abgetastet, sondern die Punkte werden völlig zufällig gewählt. Dies stellt besonders in den Fällen eine Verbesserung dar, in denen der Ausgangspunkt eine vorbestimmte Konfiguration ist und versucht wird diese dann iterativ zu verbessern. Anstatt alle möglichen Konfigurationen zu testen, wird die Vorhandene leicht verändert (zum Beispiel in einem Punkt) und danach verglichen. Auf diese Art kann das Verfahren unter gewissen Umständen die Lösung schneller dem Minimum annähern. Mit der stochastischen Brute Force Suche kann das globale Minimum angenähert werden. Das tatsächliche Finden eines solchen globalen Minimums stellt ein Problem dar, da dafür die exakte Kombination vom Zufallsgenerator erraten werden müsste.

4.2 Lokale Suche

Die Lokale Suche gehört zu den iterativen Verfahren. Auch hier wird für eine Funktion $f : D \to \mathbb{R}^n$ mit $D \subset \mathbb{R}^n$ jenes x_u (bzw. mehrere $x_{u,1}, \ldots, x_{u,n}$ für $n \in \mathbb{N}$) gesucht, für das f minimal ist. Anders als bei Brute Force wird ein spezieller Punkt x_0 ausgezeichnet, der als Startpunkt dient. Von diesem Startpunkt aus wird die Lösung dann iterativ verbessert. Dies geschieht wie folgt:
Berechne für den Startwert x_0 den Funktionswert $f(x_0)$. Nun wähle in einer ϵ-Umgebung von x_0 einen neuen Wert, also $x_1 = x_0 + d$ wobei $d \in (-\epsilon, \epsilon)$ liegt. Berechne nun $f(x_1)$. Es werden zwei Fälle unterschieden:

- $f(x_1) < f(x_0)$
  Der neue Funktionswert ist kleiner als der Aktuelle womit x_1 als neuer Vorschlag für das Minimum akzeptiert wird.

- $f(x_1) > f(x_0)$
  Der neue Funktionswert ist größer als der Aktuelle, dadurch bleibt x_0 als Vorschlag für das Minimum.

Hier iteriert nun das Verfahren, ausgehend von dem aktuellen Vorschlag wird wieder in einer ϵ-Umgebung ein neuer Wert gesucht und abermals verglichen.
Der Vorgang wird beendet, wenn für eine festgelegte Anzahl an Schritten keine Verbesserung mehr eintritt. Dann wird davon ausgegangen, dass ein Minimum gefunden wurde.
Die Lokale Suche ist ein Sammelbegriff für alle Algorithmen, die iterativ vom aktuellen Wert in der unmittelbaren Umgebung einen geeigneten Nachfolger wählen, um so approximativ eine Lösung zu generieren. Die vielen Abwandlungen der Lokalen Suche entstehen durch verschiedene Methoden der Nachfolgerwahl, als auch der Akzeptanz- und Verwerfungskriterien für einen Kandidaten.
Bei dem hier präsentierten Vorgehen, kann nicht garantiert werden, dass es sich bei dem gefundenen Minimum um ein globales Minimum handelt. Ausgehend vom Startpunkt, kann die Methode in ein lokales Minimum geraten, welches nicht mehr verlassen werden kann. Es ist also Zufall, ob das globale oder nur ein lokales Minimum gefunden wurde.
Demonstrativ kann die Lokale Suche mit einem Bergsteiger verglichen werden. Dabei wird von einem Bergsteiger ausgegangen, der aufgrund von extrem dichtem Nebel mit eingeschränkter Sicht einen Berg erklimmen möchte. Dabei geht der Bergsteiger davon aus, dass er den Berg erklommen hat, falls er einen Punkt erreicht, von dem es nur mehr bergab geht. Dem Bergsteiger kann es also passieren, dass er nur eine kleine Erhöhung erreicht hat, und auf ihr genauso verharrt wie die Lokalen Suche bei einem globalen Minimum.

Simulated Annealing toleriert zu einem gewissen Grad ein verschlechtern der Lösung, also einen Abstieg des Bergsteigers, damit eventuell doch der echte Gipfel erreicht werden kann.
Abschließend sei gesagt, dass bei der Lokale Suche das Problem mit dem lokalen Minimum vermieden werden kann, wenn der Algorithmus häufig genug mit verschiedenen Startpunkten durchgeführt wird. Hierbei kann die Beschaffenheit der Funktion, die im allgemeinen Unbekannt ist, große Schwierigkeiten bei der Wahl geeigneter Startpunkte bereiten.

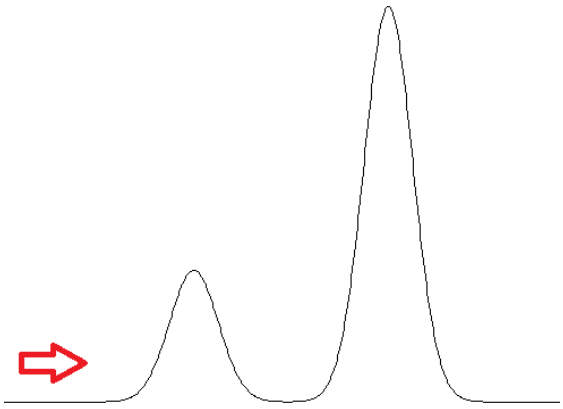

Abbildung 1: Der Bergsteiger wird stets im kleineren Hügel verharren, wenn er von links startet.

4.3 Markov Chain Monte Carlo

Markov Chain Monte Carlo Algorithmen gehören zu den Methoden der Lokalen Suche. Auch Simulated Annealing gehört dieser Klasse an. Bei einer Monte Carlo Simulation werden Stichproben aus einer Verteilung gezogen um anschließend Berechnungen anzustellen. Dies ist hilfreich, falls eine analytische Berechnung mithilfe der Verteilung nicht möglich oder extrem aufwendig ist.
Eine häufige Anwendung von Monte Carlo Simulationen sind hochdimensionale Integrale. Möglich ist diese Herangehensweise durch das Gesetz der großen Zahlen Dies bedeutet, dass Monte Carlo Simulationen genauer werden, je größer die Stichprobenmenge wird und dementsprechend auch mehr Berechnungen angestellt werden. Das Problem hierbei ist, dass die benötigte Verteilung simulierbar sein muss, damit Stichproben gezogen werden können. Oft ist jedoch entweder die Verteilung schwer zu simulieren oder gänzlich unbekannt.

Mit MCMC Verfahren kann dieses Problem umgangen werden. Dazu wird eine aperiodische und irreduzible Markov-Kette konstruiert. Anstelle des Gesetzes der großen Zahlen nutzt man nun den Ergodensatz, der bei einer solchen Markov-Kette sowohl die Existenz und Eindeutigkeit einer stationären Verteilung garantiert, als auch die Konvergenz der Markov-Kette gegen diese Verteilung. Damit ist es obsolet, die Zielverteilung direkt zu berechnen, sondern es muss lediglich eine passende Markov-Kette erzeugt werden, deren stationäre Verteilung hinreichend nahe an der gesuchten Verteilung ist. Dieses Vorgehen ist in der Praxis meistens wesentlich einfacher.
Eine wichtige Einschränkung dieser Vorgehensweise ist, dass nicht klar ist, ab wann das Verfahren eine gewisse Fehlerschranke unterschreitet. Diese Phase wird "burn-in"genannt. In dieser Phase sucht die Markov-Kette gewissermaßen zunächst die Zielverteilung, und erst danach werden die Ergebnisse brauchbar. Die Dauer dieser Phase hängt einerseits von der Zielverteilung, aber auch von dem gewählten Startpunkt ab, und kann bereits nach wenigen Schritten verlassen werden, aber auch tausende Iterationsschritte andauern. Details zu diesem Unterkapitel sind in [Ros10] [MGNR12] zu finden
Als Beispiel für dieses abstrakte Konzept dient der Metropolis-Hastings Algorithmus.

Definition 4.3.1. *[Ros10] S.261*
Sei Q eine irreduzible und symmetrische Übergangsmatrix. Dann lässt sich wie folgt eine Markov-Kette erzeugen:
Falls $X_n = i$, dann erzeuge eine Zufallsvariable Y sodass $P(Y = j) = q(i,j), j = 1, 2, \ldots$. Falls $Y = j$, dann setze $X_{n+1} = j$ mit Wahrscheinlichkeit $\alpha(i,j) \in (0,1]$ und falls $Y = i$, dann setze $X_{n+1} = i$ mit Gegenwahrscheinlichkeit $1 - \alpha(i,j)$

Die so konstruierte Markov-Kette hat eine Übergangsmatrix P, die wie folgt aussieht:

$$\begin{aligned} P_{i,j} &= q(i,j)\alpha(i,j), \text{falls } j \neq i \\ P_{i,i} &= q(i,i) + \sum_{k \neq i} q(i,k)(1 - \alpha(i,k)) \end{aligned} \tag{4.3.1}$$

Diese Markov-Kette ist irreduzibel und reversibel mit stationärer Verteilung, falls

$$\pi(i)P_{i,j} = \pi(j)P_{j,i} \text{ für } j \neq i \tag{4.3.2}$$

Die Matrix Q wird Vorschlagsmatrix genannt. Sie schlägt Werte vor, welche die Markov-Kette im nächsten Schritt annehmen soll. $\lambda(i, j)$ sind die Akzeptanzwahrscheinlichkeiten, die den vorgeschlagenen Wert zulassen. Falls abgelehnt wird, verbleibt die Markov-Kette im aktuellen Zustand $X_{nn+1} = X_n$. Die generierten Werte simulieren eine Wahrscheinlichkeitsverteilung, woraus bei Bedarf auch ein Erwartungswert berechnet werden kann.
Im folgenden soll die praktische Anwendung eines MCMC-Verfahrens anhand eines einfachen Beispiels erläutert werden:
Es wird versucht die Standardnormalverteilung (N(0,1)) mittels MCMC anzunähern. Die Dichte dieser Verteilung sei hier f(x). Als Startpunkt wird 0.5 gewählt. Die Vorschlagsdichte sei N(0,0.25). Die Akzeptanzwahrscheinlichkeiten werden wie folgt ausgewählt: Falls der Funktionswert an dem vorgeschlagenen Punkt höher ist als vorher, dann soll der vorgeschlagene Wert auf jeden Fall angenommen werden. Falls der neue Funktionswert den Alten unterschreitet, dann soll der neue Wert mit Wahrscheinlichkeit $\frac{f(X_{n+1})}{f(X_n)}$ angenommen werden.
Konkret funktioniert der Algorithmus nun wie folgt:

1. Nehme den Startpunkt $X_0 = 0.5$ (f(X_0)=0.3520653) und ziehe eine Zufallszahl aus N(0,0.25), z.B. 0.015

2. Betrachte den neuen Wert Y = 0.515. f(Y) = 0.3493954.

3. Da f(Y) < f(X_0), setze $X_1 = 0.515$ nur, falls U(0,1) < $\frac{f(Y)}{f(X_0)} = 0.9924$, ansonsten setze $X_1 = X_0$.

4. Kehre zu Schritt 1 zurück, nun mit X_1 statt X_0

Die so generierten Werte werden abgespeichert und mittels Histogramme kann dann die zugehörige Verteilung visualisiert werden. Dieser Algorithmus nennt sich auch Metropolis-Hastings Algorithmus, und gehört zu den geläufigeren MCMC-Verfahren.
Im Folgenden sind einige Durchläufe mit zunehmender Iterationsanzahl abgebildet.

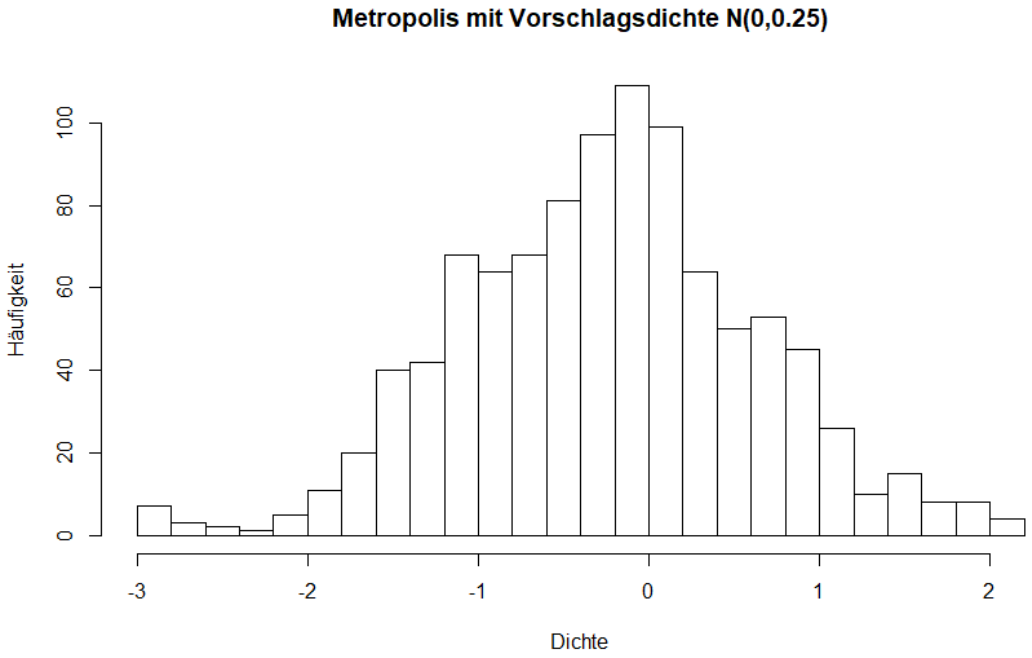

Abbildung 2: N(0,1) wird mit einer Vorschlagsdichte von N(0,0.25) in 1000 Schritten approximiert

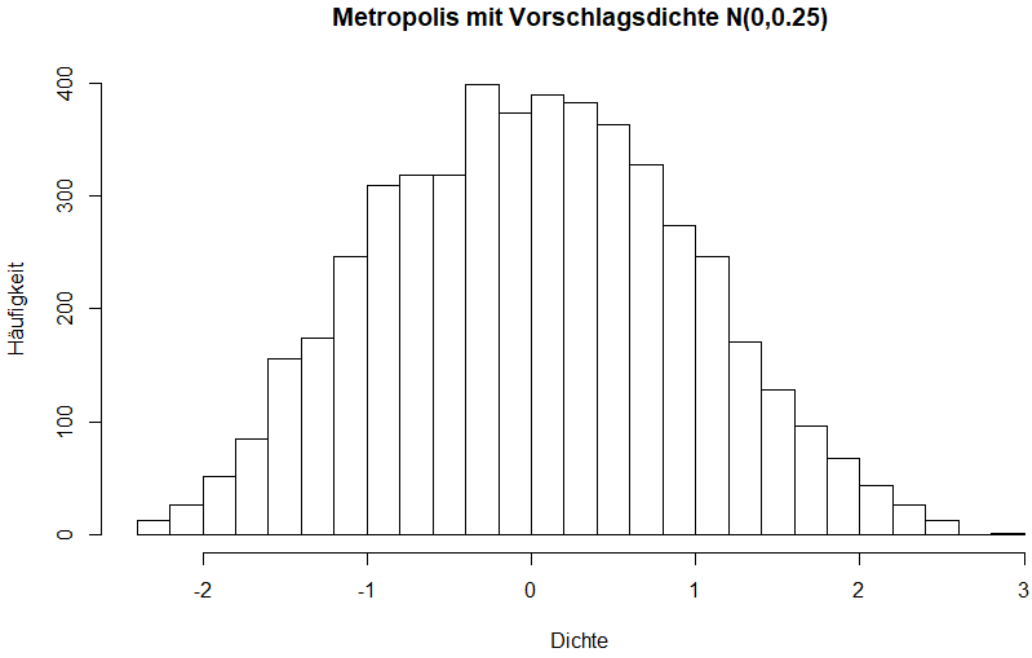

Abbildung 3: N(0,1) wird mit einer Vorschlagsdichte von N(0,0.25) in 5000 Schritten approximiert

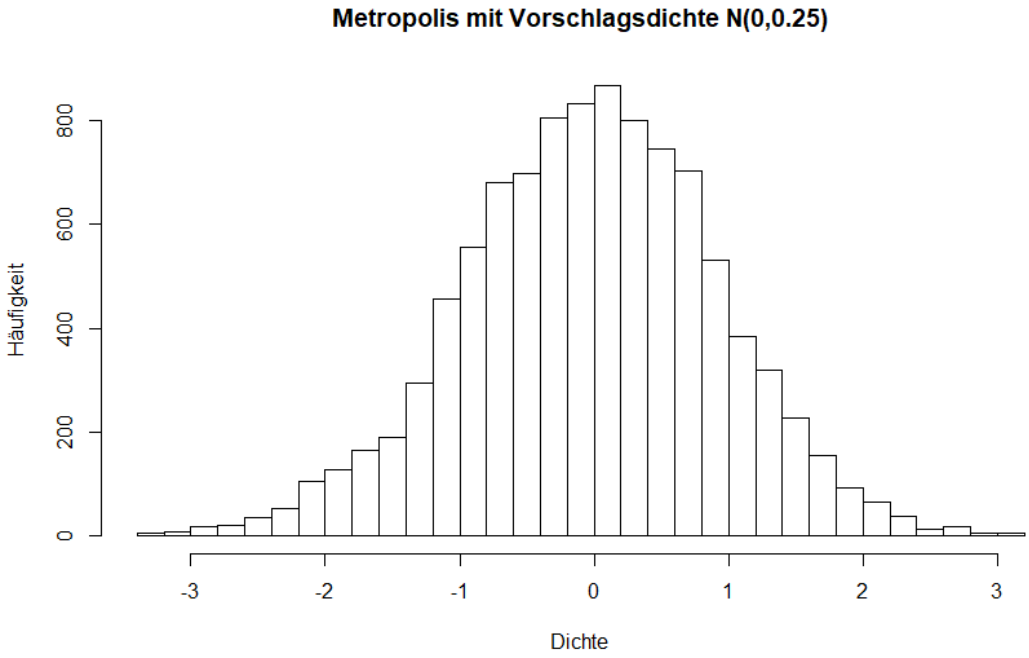

Abbildung 4: N(0,1) wird mit einer Vorschlagsdichte von N(0,0.25) in 10000 Schritten approximiert

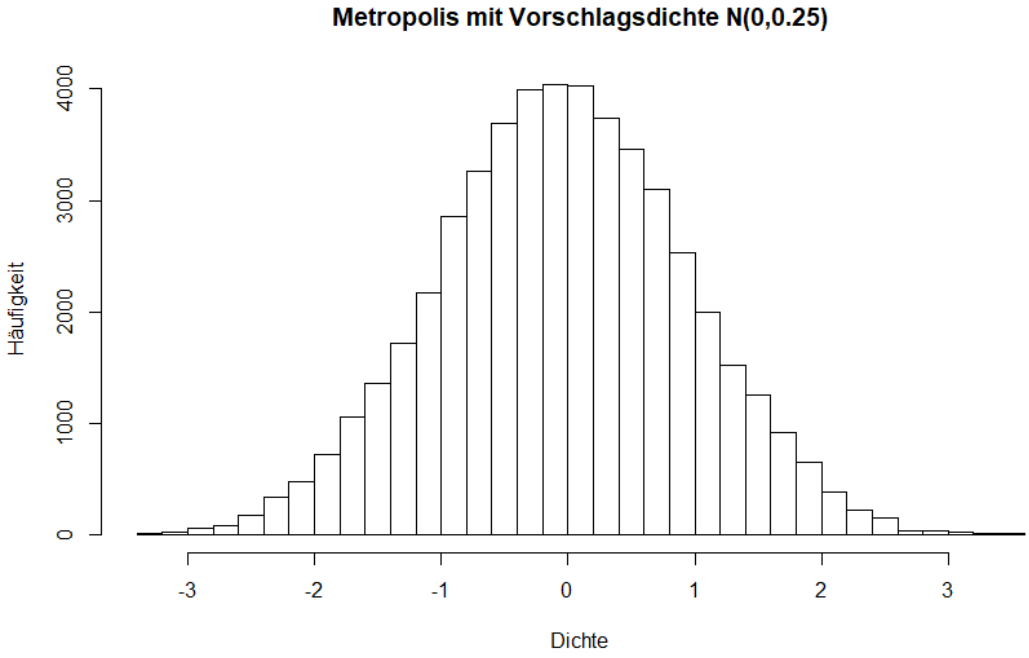

Abbildung 5: N(0,1) wird mit einer Vorschlagsdichte von N(0,0.25) in 50000 Schritten approximiert

5 Simulated Annealing

Das folgende Kapitel bezieht sich in weiten Teilen auf [LA87].

5.1 Motivation

Simulated Annealing ist eine mathematische Analogie zu einem physikalischen Phänomen, welches die Erhitzung eines Stoffes in der Metallurgie begleitet. Hierbei wird ein Stoff in einem Wärmebad so stark erhitzt, sodass sich alle Moleküle frei ausrichten können. Gemäß dem zweiten Hauptsatz der Thermodynamik sucht jedes Teilchen den für sich energieärmsten Zustand. Durch vorsichtiges und langsames Abkühlen wird die optimale Ausrichtung der Teilchen erhalten und setzt sich fort bis die Abkühlung beendet ist. Bei jedem Temperaturschritt T lässt man den Stoff ein sogenanntes thermales Gleichgewicht erreichen. Dieser Energiezustand wird beschrieben durch:

$$\mathbb{P}[\mathbf{E} = E] = \frac{1}{Z(T)} \cdot exp(-\frac{E}{k_B T}) \tag{5.1.1}$$

wobei Z(T) ein Normalisierungsfaktor und k_B die Boltzmann Konstante ist.
$exp(-\frac{E}{k_B T})$ nennt man auch den Boltzmann-Faktor. Die in 5.1.1 beschriebene Verteilung wird Boltzmann Verteilung genannt. Aus der Definition der Verteilung geht hervor, dass mit sinkender Temperatur die Masse der Boltzmann Verteilung zunehmend auf einen kleineren Bereich konzentriert wird, bis schließlich nur mehr die optimalen (=minimalen) Energiezustände positive Wahrscheinlichkeit haben. Problematisch ist, dass bei zu schneller und starker Abkühlung Ungleichgewichte aus vorherigen Temperaturzuständen mitgenommen werden, da dem Stoff zu wenig Zeit gelassen wurde, überall sein thermales Gleichgewicht zu erreichen. Der Name Simulated Annealing leitet sich aus dem Englischen to anneal, abkühlen, ab. Da die Abkühlung bei Algorithmen nur simuliert ist, ergibt sich daraus der Begriff Simulated Annealing.
Für jeden Temperaturschritt kann die Annäherung an das Thermale Gleichgewicht durch einen Metropolis-Algorithmus simuliert werden. Dazu wird die Position eines der Teilchen verändert, wobei die neue Position meistens in einer Umgebung des Teilches zu finden ist. Nun wird die Energie des Momentanzustands mit dem Ursprungszustand verglichen. Falls dieses ΔE negativ ist, so bedeutet dies, dass die Veränderung einen energieärmeren Zustand produziert hat. Von diesem wird nun wieder ein Teilchen verändert. Falls $\Delta \mathrm{E} \geq 0$, dann wird der neue Zustand mit der Wahrscheinlichkeit $exp(-\frac{\Delta E}{k_B T})$ angenommen.

Folglich kann Simulated Annealing als eine Folge von Metropolis Algorithmen mit abnehmender Akzeptanzwahrscheinlichkeit verstanden werden.
Dazu sei $C : I \to \mathbb{R}$ eine Kostenfunktion und c der Kontrollparameter. Modellhaft steht C hier für die Energie und c für die Temperatur. I ist ein abzählbarer Zustandraum. Zunächst ist c sehr hoch und ein Übergangsverfahren wird konstruiert, welches das System von der jetzigen Konfiguration i in eine andere Konfiguration j überführt. Diese Konfiguration ist in einer Umgebung von i zu finden. Sei nun $\Delta C_{ij} = C(j) - C(i)$. Die Wahrscheinlichkeit von der Konfiguration i in die Konfiguration j zu wechseln ist 1, falls $\Delta C_{ij} \leq 0$ und $\exp(-\frac{\Delta C_{ij}}{c})$ falls $\Delta C_{ij} > 0$. Dieser Vorgang wird fortgesetzt, bis folgendes gilt:

$$\mathbb{P}(\text{Konfiguration} = i) = q_i(c) = \frac{1}{Q(c)} \cdot \exp(-\frac{C(i)}{c}) \tag{5.1.2}$$

Hierbei ist Q eine Normalisierungskonstante, welche das Äquivalent des Normalisierungsfaktor Z(T) darstellt.
Der Kontrollparameter wird stets nach Beendigung des obigen Ablaufes um einen Schritt gesenkt, bis ein sehr kleiner Wert für c erreicht wird, zu dem nahezu keine Änderungen mehr akzeptiert werden. Diese Konfiguration ist gewissermaßen "eingefroren" und kann nun als Lösung dienen. Eine optimale Konfiguration, also eine welche die Kostenfunktion minimiert, wird $i_0 \in I$ genannt. Es gilt:

$$C_{opt} = C(i_0) = \min_{i \in I} C(i) \tag{5.1.3}$$

Im Vergleich ist die Lokale Suche ein Spezialfall des Simulated Annealing mit konstanter Temperatur 0.

5.2 Definition

Simulated Annealing ist der wiederholte Vorgang, die momentane Konfiguration in eine andere Konfiguration, welche in einer Umgebung der aktuellen Konfiguration liegt, zu überführen. Damit wird versucht, eine Konfiguration zu finden, welche einen energieärmeren Zustand ermöglicht. Welche Konfiguration als nächste gewählt wird, hängt dabei ausschließlich von der aktuellen Konfiguration ab. Diese Eigenschaft wird auch Markov-Eigenschaft genannt und ist eine Eigenschaft der Markov Kette. Deshalb erweist es sich als zielführend, Simulated Annealing mathematisch mit einer Markov Kette zu beschreiben.
Die Wahrscheinlichkeit p_{ij} ist somit die Wahrscheinlichkeit aus der Konfiguration i im nächsten Schritt die Konfiguration j anzunehmen.

Diese Übergangswahrscheinlichkeiten hängen von einem Kontrollparameter c ab, welcher an der Temperatur im Abkühlprozess angelehnt ist. Bleibt dieser Parameter c konstant, so handelt es sich um eine homogene Markov Kette und die Übergangswahrscheinlichkeiten können wie folgt definiert werden:

$$p_{ij}(c) = \begin{cases} G_{ij}(c)A_{ij}(c) & \forall j \neq i \\ 1 - \sum_{l=1, L \neq i}^{|I|} G_{il}(c)A_{il}(c) & j = i \end{cases} \tag{5.2.1}$$

Die Übergangswahrscheinlichkeit ist somit das Produkt der Vorschlagswahrscheinlichkeit $G_{ij}(c)$ und der Akzeptanzwahrscheinlichkeit $A_{ij}(c)$. Die so konstruierte Matrix ist tatsächlich auch eine stochastische nach Definition.
Bei der simulierten Abkühlung wird die Temperatur schrittweise gesenkt. Dies entspricht dem Senken des Kontrollparameters c. Dieser Tatsache kann mathematisch nun mit zwei Möglichkeiten begegnet werden:

- Der homogene Algorithmus:
  In jedem Temperaturschritt wird eine neue homogene Markov Kette wie beschrieben konstruiert. Simulated Annealing ist somit eine Verkettung mehrerer homogener Markovketten.

- Der inhomogene Algorithmus:
  Der gesamte Abkühlprozess wird durch eine einzige inhomogene Markov Kette modelliert.

5.3 Konvergenzverhalten

Simulated Annealing erreicht ein globales Minimum, falls nach einer entsprechend großen Anzahl an Übergängen K

$$\mathbb{P}(X_K \in I_{opt}) = 1 \tag{5.3.1}$$

gilt. Hierbei ist I_{opt} eine Menge von globalen minimalen Konfigurationen.
Im Folgenden wird gezeigt, dass sowohl der homogene als auch der inhomogene Algorithmus unter bestimmten Voraussetzungen asymptotisch 5.3.1 erfüllt.
Für den homogenen Algorithmus muss folgendes gelten:

- Jede homogene Markovkette ist unendlich lang.

- Bestimmte Bedingungen an $A(c_l)$ und $G(c_l)$ sind erfüllt.

- $$\lim_{l \to \infty} c_l = 0 \tag{5.3.2}$$

wobei A und G die entsprechenden Akzeptanz- und Vorschlagsmatrizen sind und c_l der Kontrollparameter der l-ten Markovkette ist.
Für den inhomogenen Algorithmus gilt:

- Bestimmte Bedingungen an $A(c_k)$ und $G(c_k)$ sind erfüllt.
- Die Konvergenzrate der Folge c_k ist durch $\mathcal{O}(|\log k|^{-1})$ beschränkt.
- $$\lim_{k\to\infty} c_k = 0 \tag{5.3.3}$$

Der homogene Algorithmus wird hier genau untersucht, für den aufwändigeren inhomogenen Fall sei auf [LA87] verwiesen:
Im Folgenden wird die stationäre Verteilung der Markov Kette relevant. Diese sei hier $\mathbf{q}$, wobei die Komponenten dieses Vektors wie folgt aussehen:

$$q_i = \lim_{k\to\infty} \mathbb{P}(X_k = i | X_0 = j) \tag{5.3.4}$$

wobei j beliebig ist
Insgesamt wird gefragt, ob die folgende Gleichung hält:

$$\lim_{c\searrow 0} \mathbf{q}(c) = \pi \tag{5.3.5}$$

wobei π ein I-Vektor mit den folgenden Koordinaten ist:

$$\pi_i = \begin{cases} |I_{opt}|^{-1} & \text{falls } i \in I_{opt} \\ 0 & \text{sonst} \end{cases} \tag{5.3.6}$$

Es folgt

$$\lim_{c\searrow 0}(\lim_{k\to\infty} \mathbb{P}(X_k = i)) = \pi_i \tag{5.3.7}$$

und somit gilt

$$\lim_{c\searrow 0}(\lim_{k\to\infty} \mathbb{P}(X_k \in I_{opt})) = 1 \tag{5.3.8}$$

Zunächst stellt sich die Frage, ob überhaupt eine stationäre Verteilung existiert. Diesbezüglich ist der Ergodensatz hilfreich, der unter bestimmten Voraussetzungen an die Markovkette nicht nur die Existenz und Eindeutigkeit einer solchen Verteilung garantiert, sondern auch die Konvergenz der Markov Kette gegen diese Verteilung. Die Bedingungen sind Aperiodizität und Irreduzibilität der Markov Kette.

Für die Akzeptanzmatrix gilt, dass alle Einträge stets positiv sind, da die Annahmewahrscheinlichkeit durch den Boltzmannfaktor bestimmt wird. An die Vorschlagsmatrix wird folgende Bedingung gestellt:

$$\forall i,j \in \mathcal{R}\ \exists p \geq 1\ \exists l_0, l_1, \ldots, l_p \in I\ (l_0 = i \wedge l_p = j) : \\ G_{l_k l_{k+1}}(c) > 0, k = 0, 1, \ldots, p-1 \tag{5.3.9}$$

Insgesamt folgt, dass die einzelnen Übergangswahrscheinlichkeiten strikt positiv sind. Daraus folgt, dass die Markov Kette irreduzibel ist, da kein Übergang Wahrscheinlichkeit 0 hat.
Die Aperiodizität lässt sich aus der Irreduzibilität folgern, falls die folgende Bedingung gilt [LA87]

$$\forall c > 0\ \exists i_c \in I : p_{i_c i_c}(c) > 0 \tag{5.3.10}$$

Insgesamt reicht es also, von den Matrizen folgendes zu fordern [LA87]

$$\forall c > 0 \exists i_c, j_c \in I : A_{i_c j_c}(c) < 1 \wedge G_{i_c j_c} > 0 \tag{5.3.11}$$

Mit der Konstruktion der $p_i j$ und den Bedingungen an die Matrizen A und G folgt also die Existenz einer stationären Verteilung. Für den weiteren Verlauf sei sie $\mathbf{q}(c)$ genannt. Damit folgt auch die Konvergenz der Markov Kette gegen diese Verteilung.
Da im homogenen Fall für jeden Wert des Kontrollparameters c eine eigene homogene Markov Kette konstruiert wurde, bleibt nun die Frage zu klären, ob diese so gewonnene Folge von stationären Verteilungen auch gegen die gewünschte Zielverteilung konvergiert. Dazu wird eine neue Funktion benötigt:

Definition 5.3.1. *[LA87]S.21*
Sei $\psi(\gamma, c)$ eine Funktion die folgende Bedingungen erfüllt:

- $\forall i \in I, c > 0 : \psi(C(i), c) > 0$
- $\forall j \in I :$
  $\sum_{i=1, i\neq j}^{|I|} \psi(C(i), c) G_i j(c) A_{ij}(c) = \psi(C(j), c) \sum_{i=i, i\neq j}^{|I|} G_{ji}(c) A_{ji}(c)$

Die zweite Bedingung wird auch global balance Bedingung genannt, sie ist eine weniger strenge Variante der detailed balance Gleichung. Die stationären Verteilungen können nun wie folgt angeschrieben werden:

$$q_i(c) = \frac{\psi(C(i), c)}{\sum_j \psi(C(j), c)} \tag{5.3.12}$$

Für $\lim_{c \searrow 0} \mathbf{q}(c) = \pi$ muss π noch die folgenden Bedingungen erfüllen:

$$\lim_{c \searrow 0} \psi(\gamma, c) = \begin{cases} 0 & \text{falls } \gamma > 0 \\ \infty & \text{falls } \gamma < 0 \end{cases} \tag{5.3.13}$$

$$\frac{\psi(\gamma_1, c)}{\psi(\gamma_2, c)} = \psi(\gamma_1 - \gamma_2, c) \tag{5.3.14}$$

$$\forall c > 0 : \psi(0, c) = 1 \tag{5.3.15}$$

All diese Bedingungen garantieren, dass die q_i's gegen genau das definierte π konvergieren. Es gibt viele Ansätze, eine solche Markov Kette zu konstruieren, und einer soll hier vorgestellt werden:

Satz 5.3.1. *[LA87]S.22 Falls für die Funktion $\psi(C(i) - C_opt, c) = A_{i_0 i}(c)$ gilt, wobei $i_0 \in I_{opt}$ und $G(c)$ von c unabhängig ist, und folgende Bedingungen für $A(c)$ und G erfüllt sind:*

$$\forall i, j \in I : G_{ji} = G_{ij} \tag{5.3.16}$$

$$\forall i, j, k \in I : C(i) \leq C(j) \leq C(k) \Rightarrow A_{ik}(c) = A_{ij}(c) A_{jk}(c) \tag{5.3.17}$$

$$\forall i, j \in I : C(i) \geq C(j) \Rightarrow A_{ij}(c) = 1 \tag{5.3.18}$$

$$\forall i, j \in I, c > 0 : C(i) < C(j) \Rightarrow 0 < A_{ij}(c) < 1 \tag{5.3.19}$$

dann sieht die stationäre Verteilung wie folgt aus:

$$\forall i \in I : q_i(c) = \frac{A_{i_0 i}(c)}{\sum_{j \in I} A_{i_0 j}(c)} \tag{5.3.20}$$

Es folgt dabei aus

$$\forall i, j \in I : C(i) < C(j) \Rightarrow \lim_{c \searrow 0} A_{ij}(c) = 0 \tag{5.3.21}$$

dass die stationären Verteilungen $\boldsymbol{q}(c)$ gegen π konvergieren.

Für den Beweis siehe [LA87]

5.4 Praktische Überlegungen

Die wichtigsten Mechanismen mit denen das Verfahren näher definiert und auch sein Konvergenzverhalten manipuliert werden kann, sind:

- Starttemperatur T_0
- Endtemperatur T_f (finale Temperatur)
- Die Länge der Markovkette des Metropolisalgorithmus
- Die Vorschrift, nach der die Abkühlung stattfindet.

Die Gesamtheit dieser Parameter kann auch vereinfacht Abkühlplan (von englischen cooling schedule) genannt werden.
Da nur endlich viele Schritte gemacht werden können und somit das Temperatur Gleichgewicht nicht erreicht werden kann, wird stattdessen ein Quasi-Gleichgewicht angestrebt. Quasi-Gleichgewicht wird der Zustand eines Systems genannt, welcher sich hinreichend nahe dem echten Gleichgewicht befindet. Hierbei handelt es sich mehr um ein Konzept als um eine tatsächliche Metrik, womit auch die Entscheidung, wie nahe das Quasi-Gleichgewicht dem echten sein muss, bei der Konstruktion eines Lösungsverfahrens individuell gefällt werden muss. Folgende Überlegungen finden sich häufig bei der Erstellung eines Abkühlplans:

- Die Starttemperatur sollte so hoch gewählt werden, dass jeder mögliche Zustand gleich wahrscheinlich angenommen wird.
- Die finale Temperatur sollte so angesetzt sein, dass in diesem Schritt nur mehr sehr wenige Zustandswechsel akzeptiert werden. Allerdings sollte auch darauf geachtet werden, dass nicht zu viele der Temperaturschritte kurz vor der finalen Temperatur kaum Änderungen bringen und damit den Algorithmus unnötig verlangsamen.
- Die Länge der Markovkette und die Vorschrift zur Reduktion der Temperatur sollten in Balance gehalten werden. Dies liegt daran, dass zu hohe Temperaturschritte mehr Übergänge in den einzelnen Temperaturstadien notwendig machen, um das erwünschte Quasi-Gleichgewicht zu erreichen, da sonst Defekte weitergegeben werden und nicht mehr korrigiert werden können. Kleine Temperaturschritte lassen weniger Übergänge nötig werden. Die optimale Balance ist für das vorliegende Problem individuell zu bestimmen.

Ein naheliegender und intuitiv verständlicher Kühlplan kann wie folgt aussehen:
Der Startwert kann empirisch durch eine Versuchsreihe ermittelt werden. Dazu sei ein beliebiges T_0 fixiert. Anschließend wird ein Testlauf durchgeführt, mit dem die Akzeptanzrate getestet wird, mit der neue Vorschläge angenommen werden. Falls diese Akzeptanzrate einen gewünschten Wert unterschreitet (bspw. 0.8, 0.9 oder 0.95), dann wird die Starttemperatur verdoppelt und erneut die Akzeptanzrate überprüft. Wiederholtes Durchführen dieses Vorgangs produziert schlussendlich eine vernünftige Starttemperatur.
Die finale Temperatur ergibt sich meistens aus der Anzahl der geplanten Temperaturschritte, kann jedoch ähnlich der Starttemperatur anhand einer gewünschten Akzeptanzrate ermittelt werden. In diesem Fall sollte die Akzeptanzrate einen positiven Wert nahe 0 unterschreiten.
Als Wahl für die Länge der Markovkette eignet sich ein deterministischer Wert der sich an der Größe des Problems orientiert. Eine andere Möglichkeit ist eine dynamische Länge, die sich an jeden Temperaturschritt anpasst. Beispielsweise können eine Mindestzahl oder Höchstgrenze an Übergängen gefordert werden, und die Länge der Markovkette dementsprechend angepasst werden. Dabei müssen jedoch Probleme, die damit einhergehen beachtet werden, wie z.B. eine viel zu lange Markovkette bei der Starttemperatur oder bei der finalen Temperatur.
Zur Absenkung der Temperatur eignet sich als Vorschrift

$$T_{k+1} = \alpha \cdot T_k, k \in \mathbb{N} \tag{5.4.1}$$

α kann so gewählt werden, dass entsprechend größere oder kleinere Temperaturschritte das Resultat sind, bspw. $\alpha = 0.95$ oder $\alpha = 0.99$
Dieser Ansatz birgt das Problem, dass die Temperaturschritte zunehmend kleiner werden, je näher die Temperatur gegen 0 geht. Als Alternative kann eine Vorschrift gewählt werden, welche die gleichen Temperaturschritte garantiert. Dazu muss die Gesamtanzahl der Schritte zunächst fixiert werden, und sei hier mit K bezeichnet. Dann gilt:

$$T_k = \frac{K-k}{K} \cdot T_0, k = 1, \ldots, K \tag{5.4.2}$$

Es gibt noch andere Ansätze für Kühlpläne, die weitaus komplizierter sind.
Alle Kühlplänen sind ein Gleichgewicht zwischen der Länge der Markovkette und der Vorschrift zur Abkühlung wodurch sie in zwei Klassen unterteilt werden:

- Klasse A: Eine variable Länge der Markovkette bei fixen Abständen zwischen den Temperaturen
- Klasse B: Ein fixe Länge der Markovkette bei variablen Abständen zwischen den Temperaturen

Wie bereits erwähnt sind Klasse A Kühlpläne aufgrund der einfachen Kühlvorschrift leichter und überschaulicher zu konstruieren.

Zur Performanceanalyse betrachtet man zwei Punkte

- Die **Qualität** der Lösung
- Die **Laufzeit** des Algorithmus

In der klassischen Performanceanalyse für deterministische Algorithmen unterscheidet man zwischen dem worst-case und dem average-case. Der worst-case ist die längstmögliche Dauer des Algorithmus oder die schlechteste Lösung die generiert wird, also die Lösung, die den größten Fehler aufweist. Für Laufzeit und Qualität sind es meistens verschiedene Startbedingungen, welche für das jeweilige Kriterium den worst-case produzieren. Der average-case ist die erwartete Laufzeit und Qualität bei einer Vielzahl von Startkonfigurationen. Für Simulated Annealing gestaltet sich die Angelegenheit schwieriger, da es sich hier um einen probabilistischen Algorithmus handelt, weil nicht nur die Menge aller möglichen Probleme unbekannt ist, sondern nun auch jedes Problem einer zufälligen Lösung zugeführt wird.

6 Covering Codes

Die Suche nach Covering Codes ist eingebettet in das generelle Problem der nichtlinearen Optimierung.
Das Covering Code Problem ist ein spezialisierter Fall der generellen Suche nach der Anzahl disjunkter n-Sphären, die benötigt werden, um einen vorgegebenen Raum optimal zu füllen. Dieses Problem kann zu Fragen der Praxis spezialisiert werden, um beispielsweise nach der optimalen Anzahl und Platzierung von Sendemasten zu suchen.
Bei Covering Codes geht es konkret darum, wie viele Codes der Länge N notwendig sind, sodass der N-dimensionale Coderaum von Sphären mit Radius R und den gesuchten Codes als Mittelpunkt überdeckt werden kann. Der Distanzbegriff in diesem Raum wird definiert als die Summe der paarweisen Abstände der Bits von 2 Codes. Die Codes, welche Abstand 1 vom Code 1010 haben, sind die folgenden: 0010, 1110, 1000, 1011. Dies heißt, dass eine Sphäre mit Radius 1 um den Punkt 1010 diese 4 Punkte abdecken kann.
Alternativ zur Frage der Überdeckung ist auch die Suche nach der maximalen Anzahl an disjunkten Sphären mit Radius R, sodass jeder Punkt des Raumes von genau einer Sphäre abgedeckt ist. Diese Art der Probleme werden auch Packing problem genannt. Ferner sei erwähnt, dass man nicht nur nach binären, sondern auch im allgemeinen nach q-ären codes, wobei $q \geq 2$, suchen kann. Im Rahmen dieser Arbeit wird sich die Suche jedoch auf binäre Codes beschränken, wobei die grundlegende Theorie relativ deckungsgleich für beliebige q angewendet werden kann.
Vergleiche auch [CHLL97].

6.1 Definition

Definition 6.1.1. *[CHLL97]S.15*
Sei $\mathbb{Q}$ eine endliche Menge mit q Elementen. Eine nichtleere Menge C aus $\mathbb{Q}^n = \bigotimes_{i=1}^{n} \mathbb{Q} = \mathbb{Q} \times \mathbb{Q} \times \cdots \times \mathbb{Q}$ wird q-ärer Code der Länge n genannt.

Diese Mengen, auch Codewörter genannt, können zu einem Code zusammengefasst werden. Ein Code, der nur aus einem Codewort besteht, wird trivial genannt. $\mathbb{Q}$ wird Alphabet genannt. $\mathbb{Q}^n$ wird q-ärer Hamming-Raum genannt.
Für die weitere Betrachtung sei $\mathbb{Q} = \{0, 1\}$, also nur den Raum der binären Codes.
Zunächst sei die Hamming-Distanz definiert:

Definition 6.1.2. *[CHLL97]S.16*
Der Abstand zweier Vektoren $\mathbf{x} = (x_1, x_2, \ldots, x_n)$, $\mathbf{y} = (y_1, y_2, \ldots, y_n)$ *in* $\mathbb{Q}^n$ *definiert durch*

$$d(\mathbf{x}, \mathbf{y}) = |\{i : x_i \neq y_i\}|. \tag{6.1.1}$$

wird Hamming-Distanz genannt. Für eine Menge $V \subset \mathbb{Q}^n$ *ist die Hamming-Distanz*

$$d(\mathbf{x}, V) = \min_{\mathbf{v} \in V} d(\mathbf{x}, \mathbf{v}). \tag{6.1.2}$$

Die Hamming-Distanz erfüllt die Dreiecksungleichung und ist eine Metrik. Abhängig davon kann man das Hamming-Gewicht definieren:

Definition 6.1.3. *[CHLL97]S.16*
Das Hamming-Gewicht $\omega(\mathbf{x})$ *eines Vektors* $\mathbf{x} = (x_1, x_2, \ldots, x_n) \in \mathbb{Q}^n$ *ist*

$$\omega(\mathbf{x}) = d(\mathbf{x}, \mathbf{0}) \tag{6.1.3}$$

wobei $\mathbf{0}$ *den n-dimensionalen Nullvektor bezeichnet.*

Definition 6.1.4. *[CHLL97]S.16*
Die minimale Distanz eines Codes C ist die kleinste Distanz zwischen den Codewörtern.

$$d = d(C) = \min_{\mathbf{a}, \mathbf{b} \in C, \mathbf{a} \neq \mathbf{b}} d(\mathbf{a}, \mathbf{b}) \tag{6.1.4}$$

Definition 6.1.5. *[CHLL97]S.16*
Eine Hamming-Sphäre (auch Hamming-Kugel) $B_r(\mathbf{x})$ *mit Radius r und Mittelpunkt* $\mathbf{x} \in \mathbb{Q}^n$ *ist:*

$$B_r(\mathbf{x}) = \{\mathbf{y} \in \mathbb{Q}^n : d(\mathbf{y}, \mathbf{x}) \leq r\} \tag{6.1.5}$$

mit Karidnalzahl

$$V_q(n,r) = \sum_{i=0}^{r} \binom{n}{i} (q-1)^i \tag{6.1.6}$$

Wobei im binären Fall q nicht angeschrieben werden muss.

Definition 6.1.6. *[CHLL97]S.17*
Der Überdeckungsradius (covering radius) eines Codes $C \subset \mathbb{Q}^n$ ist die kleinste ganzzahlige Zahl R sodass jeder Vektor $\mathbf{x} \in \mathbb{Q}^n$ durch zumindest ein Codewort R-überdeckt (Hamming-Distanz $\leq$ R) wird.

$$R = R(C) = \max_{x \in \mathbb{Q}^n} d(\mathbf{x}, C) = \max_{x \in \mathbb{Q}^n} \min_{\mathbf{c} \in C} d(\mathbf{x}, \mathbf{c}) \tag{6.1.7}$$

Der Überdeckungsradius ist die Distanz zwischen dem Code und dem entferntesten Vektor im Coderaum bzw. der kleinste Radius, sodass die Hamming-Kugeln mit den Codewörtern als Mittelpunkt den gesamten Coderaum abdecken.
Ein q-ärer Code C mit K Codewörtern, minimaler Distanz d und Überdeckungsradius R wird kurz $(\mathbf{n}, \mathbf{K}, \mathbf{d})_\mathbf{q}\mathbf{R}$ **Code** geschrieben. Auch hier kann das q bei binären Codes ausgespart werden.

Definition 6.1.7. *[CHLL97]S.18*
Es sei

$$K_q(n,R) = \min\{K : \exists (n,K)_q R \ code)\} \qquad \textit{für } 0 \leq R \leq n \tag{6.1.8}$$

$$t_q(n,K) = \min\{R : \exists (n,K)_q R \ code)\} \qquad \textit{für } 0 \leq R \leq n \tag{6.1.9}$$

$$A_q(n,R) = \max\{K : \exists (n,K,d)_q R \ code)\} \qquad \textit{für } 0 \leq R \leq n \tag{6.1.10}$$

Satz 6.1.1. *[CHLL97]S.18*

$$K_q(n,R) \geq \frac{q^n}{V_q(n,R)} \tag{6.1.11}$$

Beweis. Sei C ein $(n, K)_q R$ Code. Da jedes Codewort auf jeden Fall $V_q(n, R)$ Vektoren R-überdeckt, gilt $K \cdot V_q(n, R) \geq q^n$ □

Daraus lässt sich die Dichte μ von C definieren:

Definition 6.1.8. *[CHLL97]S.18*

$$\mu(C) = \frac{K \cdot V_q(n, R)}{q^n} \tag{6.1.12}$$

Definition 6.1.9. *[CHLL97]S.15*
Ein Code wird perfekt genannt, falls die Hamming-Kugeln disjunkt sind und alles Punkte im Raum abdecken. In diesem Fall hat der Code die Dichte $\mu(C) = 1$.

In einem perfekten Code wird jeder Punkt im Coderaum von exakt einer Hamming-Kugel abgedeckt.

6.2 Rechenvorschriften

Es wird nun der Coderaum definiert und mit den nötigen Operationen versehen. Vergleiche dazu [CHLL97]S.19

$$\mathbb{Z}_q = \text{Die Menge der Ganzzahlen modulo q} \tag{6.2.1}$$

Diese Menge ist ein Körper, falls q eine Primzahl ist. Ein Körper ist eine Menge in dem alle Grundrechnungsarten wohldefiniert sind und jedes Element eine multiplikative Inverse hat.

$$\mathbb{F}_q = \text{Endlicher Körper von q Elementen} \tag{6.2.2}$$

Im speziellen binären Fall:

$$\mathbb{F} = \mathbb{F} = \{0, 1\} = \text{Der endliche Körper von 0 und 1.} \tag{6.2.3}$$

Hier wird hauptsächlich $\mathbb{F} = \mathbb{F}_2$ betrachtet. $\overline{\mathbf{x}}$ ist das Komplement von $\mathbf{x} \in \mathbb{F}^n$. Dazu tauscht man alle Koordinaten von $\mathbf{x}$.

Nun folgen einige Konventionen und Rechenregeln, wobei $\mathbf{x}, \mathbf{y}, \in \mathbb{F}^n$, $\alpha \in \mathbb{F}$ und $A, B \subseteq \mathbb{F}$:

$$\mathbf{x} + \mathbf{y} = (x_1 + y_1, x_2 + y_2, \ldots, x_n + y_n) \tag{6.2.4}$$

$$\mathbf{x} - \mathbf{y} = (x_1 - y_1, x_2 - y_2, \ldots, x_n - y_n) \tag{6.2.5}$$

$$\mathbf{x} \cdot \mathbf{y} = (x_1 y_1, x_2 y_2, \ldots, x_n y_n) \tag{6.2.6}$$

$$\langle \mathbf{x}, \mathbf{y} \rangle = x_1 y_1 + x_2 y_2 + \cdots + x_n y_n \tag{6.2.7}$$

$$d(\mathbf{x}, \mathbf{y}) = \omega(\mathbf{x} - \mathbf{y}) = \omega(\mathbf{x} + \mathbf{y}). \tag{6.2.8}$$

$$A + B = \{\mathbf{a} + \mathbf{b} : \mathbf{a} \in A, \mathbf{b} \in B\}. \tag{6.2.9}$$

$$\mathbf{x} + A = \{\mathbf{x} + \mathbf{a} : \mathbf{a} \in A\} \tag{6.2.10}$$

$$\alpha \cdot \mathbf{x} = (\alpha x_1, \alpha x_2, \ldots, \alpha x_n) \tag{6.2.11}$$

Die skalare Multiplikation ist im binären Fall wegen $\alpha \in \{0, 1\}$ weniger interessant, dennoch ist sie für den nächsten Abschnitt wichtig.

6.3 Lineare Codes

Definition 6.3.1. *[CHLL97]S.19-20*
Ein Code $C \subseteq \mathbb{F}^n$ wird **linear** *genannt, falls die paarweisen Summen und skalaren Vielfachen der Codewörter ebenfalls im Code enthalten sind. C bildet in diesem Fall einen linearen Unterraum von $\mathbb{F}^n$.*

Mithilfe eines linearen Codes kann eine Basis aus k linear unabhängigen Codewörtern $\mathbf{g}_1, \mathbf{g}_2, \ldots, \mathbf{g}_k$ konstruiert werden. Die $k \times n$ Matrix

$$\mathbf{G} = \mathbf{G}(C) = \begin{pmatrix} \mathbf{g}_1 \\ \mathbf{g}_2 \\ \vdots \\ \mathbf{g}_k \end{pmatrix}. \tag{6.3.1}$$

Diese wird auch Generatormatrix von C genannt. Die Codewörter von C konstruieren sich aus den q^k Linearkombinationen aus $\mathbf{G}$. Mit einer $k \times n$ Matrix wird so ein Code mit Dimension k erzeugt.

Ein linearer Code $C \subseteq \mathbb{F}_q^n$ mit Dimension k, Minimalabstand d und Überdeckungsradius R wird $[n, k, d]_q R$ Code genannt. Wie gewohnt verbleibt der Fokus auf den binären Fall, womit q = 2 fixiert sei und q als Index entfallen kann. Analog zum nichtlinearen Fall sei definiert:

$$t[n,k] = \min\{R : \text{ Es gibt einen } [n,k]R \text{ Code}\} \tag{6.3.2}$$
$$k[n,R] = \min\{k : \text{ Es gibt einen } [n,k]R \text{ Code}\} \tag{6.3.3}$$
$$a[n,d] = \max\{k : \text{ Es gibt einen } [n,k,d] \text{ Code}\} \tag{6.3.4}$$
$$\tag{6.3.5}$$

Die Begriffe linear und nichtlinear müssen einander nicht ausschließen. Ein linearer Code ist ein Code, der zwingend die Definition erfüllt. Ein nichtlinearer Code kann durchaus linear sein, aber muss nicht zwingend die Definition erfüllen. Für lineare Codes existiert eine weitere Rechenregel:

$$\omega(\mathbf{x} + \mathbf{y}) = \omega(\mathbf{x}) + \omega(\mathbf{y}) - 2\omega(x \cdot y) \tag{6.3.6}$$

In einem linearen Code sind also entweder alle oder genau die Hälfte aller Codewörter gerade.

6.4 Konstruktionen

Um die mit Simulated Annealing konstruierten Codes bewerten zu können, werden in diesem Abschnitt deterministische Methoden präsentiert.
Eine einfache Methode ist die Punktion. Es wird in allen Codewörtern eine fixe Koordinate gleichsam ausgestochen.

Satz 6.4.1. *[CHLL97]S.62*
Sei C ein (n,K)R Code, dann ist der punktierte Code $C* = \{(c_1, c_2, \ldots, c_{n-1}) : (c_1, c_2, \ldots, c_n) \in C\}$ *ein Covering Code mit Radius R - 1 oder R.*

Eine weitere Möglichkeit ist das Hinzufügen eines Paritätbits:

Satz 6.4.2. *[CHLL97]S.62*
Sei C ein Code mit Überdeckungsradius R, dann ist der Code $\widehat{C} = \{(\mathbf{c}, \pi(\mathbf{c}) : \mathbf{c} \in C\}$ *ein Covering Code mit Radius R + 1.*

$$\pi(\mathbf{c}) = \begin{cases} 0 & \textit{falls } \omega(\mathbf{c}) \textit{ gerade ist} \\ 1 & \textit{falls } \omega(\mathbf{c}) \textit{ ungerade ist} \end{cases} \tag{6.4.1}$$

Eine weitere Konstruktionsmöglichkeit ist die direkte Summe:

Satz 6.4.3. *[CHLL97]S.63*
Seien C_1, C_2 Covering Codes mit ihren jeweiligen Radien R_1 und R_2, dann hat die direkte Summe

$$C_1 \oplus C_2 = \{(c_1, c_2) : c_1 \in C_1, c_2 \in C_2\} \tag{6.4.2}$$

Überdeckungsradius $R_1 + R_2$

Daraus lassen sich folgende Eigenschaften ableiten:

Satz 6.4.4. *[CHLL97]S.63*
Es gilt

$$K(n+1, R) \leq 2K(n, R) \tag{6.4.3}$$

und

$$t[n+1, k+1] \leq t[n, k] \tag{6.4.4}$$

und für fixes R existiert eine Konstante $n(R)$ sodass $\forall n \geq n(R)$ gilt:

$$K(n+2, R+1) \leq K(n, R) \tag{6.4.5}$$

und

$$k[n+2, R+1] \leq k[n, R] \tag{6.4.6}$$

Eine wichtige Alternative zur direkten Summe ist das $(\mathbf{u}, \mathbf{u}+\mathbf{v})$ Verfahren.

Satz 6.4.5. *[CHLL97]S.66*
Sei $C_1 \subseteq \mathbb{F}^n$ ein Covering Code mit Radius R_1 und analog $C_2 \subseteq \mathbb{F}^n$ mit R_2. Dann ist auch $\{(\mathbf{u}, \mathbf{u}+\mathbf{v}) : \mathbf{u} \in C_1, \mathbf{v} \in C_2\}$ ein Covering Code mit einem Radius von höchstens $R_1 + R_2$

und allgemeiner:

Satz 6.4.6. *[CHLL97]S.67*
Sie $C_1 \subseteq \mathbb{F}^{n_1}$ ein Covering Code mit Radius R_1 und analog $C_2 \subseteq \mathbb{F}^{n_2}$ mit R_2 und $f : \mathbb{F}^{n_1} \to \mathbb{F}^{n_2}$ Dann ist auch $\{(\mathbf{u}, f(\mathbf{u})+\mathbf{v}) : \mathbf{u} \in C_1, \mathbf{v} \in C_2\}$ ein Covering Code mit einem Radius von höchstens $R_1 + R_2$

Die beiden Sätze garantieren die Existenz eines solchen Codes, jedoch kann lediglich eine obere Schranke für den Radius angegeben werden. Genauer kann es im speziellen Fall R = 1 angegeben werden:

Satz 6.4.7. *Sei C ein (n,K)1 code. Der Code*

$$C' = \{(\pi(\mathbf{u}), \mathbf{u}, \mathbf{u} + \mathbf{v}) : u \in \mathbb{F}^n, v \in C\} \tag{6.4.7}$$

mit Paritätsbit $\pi(\mathbf{u})$ ist ein $(2n + 1, 2^n K)1$ Code.

Mit viel Aufwand kann dieser Spezialfall generalisiert werden. Der folgende Satz ist eine wichtige Stütze bei der Suche von Covering Codes:

Satz 6.4.8. *[CHLL97]S.68*
Sei W eine nichtleere Menge mit

$$W = \{\omega_1, \omega_2, \ldots, \omega_s\} \subseteq \{0, 1, 2, \ldots, n + 1\}$$

wobei $\omega_1 < \omega_2 < \cdots < \omega_s$. $R \leq n$ sei nichtnegativ und erfülle

$$\begin{aligned} \omega_1 &\leq \frac{1}{2}(R + 1) \\ \omega_s &\geq n + 1 - \frac{1}{2}(R + 1) \\ \omega_{i+1} - \omega_i &\leq R + 1 \quad \textit{für } i = 1, 2, \ldots, s - 1 \end{aligned} \tag{6.4.8}$$

Sei S_ω die Menge aller Vektoren der Länge n + 1 mit Gewicht ω. Mit

$$C_1 = \bigcup_{\omega \in W} S_\omega$$

und C_2 ein (n,K)R Code, dann ist

$$C = \{(x_0, \mathbf{x}, \mathbf{x}+\mathbf{y}) : x_0 \in \mathbb{F}, \mathbf{x} \in \mathbb{F}^n, (x_0, \mathbf{x}) \in C_1, \mathbf{y} \in C_2\}$$

ein $(2n+1, K\sum_{i=1}^{s}\binom{n+1}{\omega_i})R$ Code.

6.5 Schranken

In diesem Abschnitt wird erläutert, wie Schranken für $K_q(n, R)$ gefunden werden können. Es soll untersucht werden, wie ganz allgemein mit möglichst wenig Kugeln mit Radius R Codes der Länge n überdeckt werden können. Dies dient als Vorbereitung, damit anschließend mit Simulated Annealing konkrete Ergebnisse produziert werden können, die unter Umständen auch die Schranken optimieren. Ein interessanter Ansatz ist die Suche nach p(n,K,r), die maximale Anzahl an Punkten in $\mathbb{F}^n$, welche von Hamming-Kugeln mit Radius r überdeckt werden können. Allgemein enthält eine Hamming-Kugel in $\mathbb{F}^n$ mit Radius r genau $\sum_{i=0}^{r}\binom{n}{i}$ Punkte, woraus folgt:

$$p(n, K, r) \leq K\sum_{i=0}^{r}\binom{n}{i} \tag{6.5.1}$$

Gleichheit gilt nur, falls die Kugeln disjunkt sind. Fortan sei $V(n, r) = \sum_{i=0}^{r}\binom{n}{i}$. Klarerweise ist

$$p(n, K, r) \leq 2^n \tag{6.5.2}$$

Satz 6.5.1. *[CHLL97]S.146*

$$K(n, R) \geq \frac{2^n}{V(n, R)} \tag{6.5.3}$$

Da bei $K > A(n, 2r+1)$ K disjunkte Hamming-Kugeln mit Radius r in $\mathbb{F}^n$ nicht mehr möglich sind, folgt:

Satz 6.5.2. *[CHLL97]S.146*
Falls $n \geq 2r+1$, dann gilt:

$$p(n, K, r) \leq KV(n, r) - (K - A(n, 2r+1))\binom{2r}{r} \tag{6.5.4}$$

Daraus folgt:

Satz 6.5.3. *[CHLL97]S.147*
Falls $R > 0$ und $n \geq 2R+1$, dann gilt:

$$K(n, R) \geq \frac{2^n - A(n, 2R+1)\binom{2R}{R}}{V(n, r) - \binom{2R}{R}} \tag{6.5.5}$$

Es folgt nun ein weiterer Satz für p(n,K,r) mit einer Folgerung für K(n,R)

Satz 6.5.4. *[CHLL97]S.148*
Falls $n \geq 2r+3$

$$p(n, K, r) \leq K(V(n, r) - 2\binom{2r}{r} + 1) + A(n, 2r+1)(3\binom{2r}{r} - 2) \tag{6.5.6}$$

Satz 6.5.5. *[CHLL97]S.148*
Falls $R > 0$ und $n \geq 2R+3$, dann gilt:

$$K(n, R) \geq \frac{2^n - A(n, 2R+1)(3\binom{2r}{R} - 2)}{V(n, R) - 2\binom{2R}{R} + 1} \tag{6.5.7}$$

Für den nächsten Satz bezeichnet $K_i(a)$, wie oft a in der i-ten Koordinate eines Codewortes C auftritt. Analog dazu $K_{ij}(ab)$ wie oft das Paar in der i-ten und j-ten Koordinate auftritt.

Satz 6.5.6. *[CHLL97]S.149*
Sei C ein (n,K)R Code und $n > R$, dann gilt für a = 0,1

$$K_i(a) \geq \frac{2^{n-1} - KV(n-1, R-1)}{\binom{n-1}{R}} \tag{6.5.8}$$

Satz 6.5.7. *[CHLL97]S.149*
Für alle R= 0,1,... gilt

$$K(2R+2, R) = 4 \tag{6.5.9}$$

Definition 6.5.1. *[CHLL97]S.149*
Falls in allen Koordinaten der Codewörter eines Codes alle Paare 00, 01, 10 und 11 zumindest einmal vorkommen, so nennt man den Code 2-surjektiv.

Satz 6.5.8. *[CHLL97]S.150*
Falls C ein nicht 2-surjektiver (n + 2,K)R+1 Code ist, dann gilt $K \geq K(n, R)$

Satz 6.5.9. *[CHLL97]S.150*
Für alle R = 1,2, ... gilt

$$K(2R+3, R) = 7 \tag{6.5.10}$$

Satz 6.5.10. *[CHLL97]S.150*
Für alle R = 1,2, ... gilt

$$K(2R+4, R) \geq 8 \tag{6.5.11}$$

Nun einige Resultate für Covering Codes mit Radius 1:

Satz 6.5.11. *[CHLL97]S.152*
Falls n gerade ist, gilt

$$K(n,1) \geq \frac{2^n}{n} \tag{6.5.12}$$

Satz 6.5.12. *[CHLL97]S.152*
Für alle m = 1, 2, ... gilt

$$K(2^m, 1) = 2^{2^m - m} \tag{6.5.13}$$

Satz 6.5.13. *[CHLL97]S.153*
Falls n + 1 durch s + 1 teilbar ist, wobei s + 1 eine Primzahl ≠ 2 ist, dann gilt

$$K(n,1) \geq \frac{(V(n,s)+s)2^n}{V(n,s)(n+1)} \tag{6.5.14}$$

Satz 6.5.14. *[CHLL97]S.154*
Für alle n = 5 (mod 6) gilt

$$K(n,1) \geq \frac{V(n,2)+5)2^{n+1} - 9A(n,3)(n+1)}{(2V(n,2)-3)(n+1)} \tag{6.5.15}$$

Diese Resultate lassen sich verallgemeinern.

Satz 6.5.15. *[CHLL97]S.157*
Falls $n > R$, dann gilt

$$K(n,R) \geq \frac{(n-R+\epsilon)2^n}{(n-R)V(n,R)+\epsilon V(n,R-4)} \tag{6.5.16}$$

wobei $\epsilon = (R+1)\lceil\frac{(n+1)}{(R+1)}\rceil - (n+1)$

Dieser Satz kann auch etwas verbessert werden:

Satz 6.5.16. *[CHLL97]S.157*
Falls $R \geq 2$, $n \geq 2R+1$ und

$$\epsilon = (R+1)\lceil\frac{(n+1)}{(R+1)}\rceil - (n+1) \leq R-1 \tag{6.5.17}$$

dann

$$K(n,R) \geq \frac{(\rho+\epsilon)2^n}{\rho V(n,R)+\epsilon V(n,R-1)} \tag{6.5.18}$$

wobei

$$\rho = \begin{cases} n-3+\frac{2}{n} & \textit{falls } R=2 \\ n-R-1 & \textit{falls } R \geq 3 \end{cases} \tag{6.5.19}$$

Das finale Thema dieses Abschnittes sind weiter Schranken, die mithilfe von linearen Ungleichungen hergeleitet werden. Dazu sei C ein (n,K)R Code.

Definition 6.5.2. *[CHLL97]S.158*

$$\mathcal{A}_i(\mathbf{x}) = |\{\mathbf{c} \in C : d(\mathbf{c},\mathbf{x}) = i\}| \tag{6.5.20}$$

und

$$\mathcal{A}_i = \mathcal{A}_i(\mathbf{0}) \tag{6.5.21}$$

Falls nun betrachtet wird, wie die Vektoren auf $\mathbb{F}^n$ mit Gewicht i überdeckt sind, so ist im Fall R = 1 ersichtlich, dass ein Codewort mit Gewicht i + 1 genau i +

1 Vektoren überdeckt, ein Codewort mit Gewicht i überdeckt nur sich selber, und ein Codewort mit Gewicht i - 1 überdeckt n - i + 1. Damit folgt:

$$(n-i+1)\mathcal{A}_{i-1} + \mathcal{A}_i + (i+1)\mathcal{A}_{i+1} \geq \binom{n}{i} \tag{6.5.22}$$

und für eine beliebige Translation C + $\mathbf{x}$:

$$(n-i+1)\mathcal{A}_{i-1}(\mathbf{x}) + \mathcal{A}_i(\mathbf{x}) + (i+1)\mathcal{A}_{i+1}(\mathbf{x}) \geq \binom{n}{i} \tag{6.5.23}$$

Satz 6.5.17. *[CHLL97]S.158*
Erfüllt ein Code C mit K Codewörtern für alle $\mathbf{x} \in \mathbb{F}^n$ *die Ungleichung*

$$\lambda_0\mathcal{A}_0(\mathbf{x}) + \lambda_1\mathcal{A}_1(\mathbf{x}) + \cdots + \lambda_n\mathcal{A}_n(\mathbf{x}) \geq \beta \tag{6.5.24}$$

mit $\lambda_0, \lambda_1, \ldots, \lambda_n \in \mathbb{R}$*, dann*

$$K \geq \frac{\beta 2^n}{\sum_{i=0}^{n} \lambda_i \binom{n}{i}} \tag{6.5.25}$$

Satz 6.5.18. *[CHLL97]S.159*
Sei $C \subseteq \mathbb{F}^n$ *mit Radius R. Falls* $\mathcal{A}_0 = \mathcal{A}_1 = \cdots = \mathcal{A}_{R-2} = 0$ *und* $\mathcal{A}_{R-1} + \mathcal{A}_R = i$*, dann gilt:*

$$\mathcal{A}_{R+1} + \mathcal{A}_{R+2} \geq F(n - iR + 1, R + 2) \tag{6.5.26}$$

wobei

$$F(v,s) = \begin{cases} 1 & \textit{falls } 2 \leq v \leq s \\ 0 & \textit{falls } v < 2 \end{cases} \tag{6.5.27}$$

Mit diesem Satz können nun Ungleichungen für die $\mathcal{A}_i$ aufgestellt werden. Sei $m_1 > 0$ und

$$m_0 = \min_{i \geq 1}\{m_1 i + F(n - iR + 1, R + 2)\} \tag{6.5.28}$$

Für einen beliebigen Code $C \subseteq \mathbb{F}^n$ mit Radius R gilt

$$\sum_{i=0}^{R-2} m_0\mathcal{A}_i + m_1(\mathcal{A}_{R-1} + \mathcal{A}_R) + (\mathcal{A}_{R+1} + \mathcal{A}_{R+2}) \geq m_0 \tag{6.5.29}$$

und für beliebige Translation

$$\sum_{i=0}^{R-2} m_0 \mathcal{A}_i(\mathbf{x}) + m_1(\mathcal{A}_{R-1}(\mathbf{x}) + \mathcal{A}_R(\mathbf{x})) + (\mathcal{A}_{R+1}(\mathbf{x}) + \mathcal{A}_{R+2}(\mathbf{x})) \geq m_0 \quad (6.5.30)$$

weiters ist

$$K(n,R) \geq \frac{m_0 2^n}{\sum_{i=0}^{R-2} m_0 \binom{n}{i} + m_1(\binom{n}{R-1} + \binom{n}{R}) + (\binom{n}{R+1} + \binom{n}{R+2})} \quad (6.5.31)$$

Es gilt

Satz 6.5.19. *[CHLL97]S.160*
Für $C \subseteq \mathbb{F}^n$ mit Überdeckungsradius R und $\forall \mathbf{x} \in \mathbb{F}^n$

$$\sum_{i=0}^{R-2} m_0 \mathcal{A}_i(\mathbf{x}) + m_1(\mathcal{A}_{R-1}(\mathbf{x}) + \mathcal{A}_R(\mathbf{x})) + (\mathcal{A}_{R+1}(\mathbf{x}) + \mathcal{A}_{R+2}(\mathbf{x})) \geq m_0 \quad (6.5.32)$$

wobei

$$m_1 = max_{i \geq 2} \frac{F(n-R+1, R+2) - F(n-iR+1, R+2)}{i-1} \quad (6.5.33)$$

und

$$m_0 = m_1 + F(n-R+1, R+2) \quad (6.5.34)$$

Mithilfe dieser Konstruktionen und Suchverfahren wurde eine umfangreiche Tabelle, welche als Table für Covering Codes gespeichert ist. Zu finden ist jene auf dieser Website: http://old.sztaki.hu/∼keri/codes/ (Zugriff am 22.03.2018 um 15:10)
Für R = 1,2,3 sind hier bis N = 14 alle Werte dieser Seite entnommen und aufgelistet. Die fett gedruckten Werte zeigen an, dass für diese Dimension ein optimaler Code gefunden wurde.

n \ R	**1**	**2**	**3**
1	**1**		
2	**2**	**1**	
3	**2**	**2**	**1**
3	**2**	**2**	**1**
4	**4**	**2**	**2**
5	**7**	**2**	**2**
6	**12**	**4**	**2**
7	**16**	**7**	**2**
8	**32**	**12**	**4**
9	62	**16**	**7**
10	107-120	24-30	**12**
11	180–192	37–44	15–16
12	342–380	62–78	18–28
13	598–704	97–128	28–42
14	1172–1408	159–248	44–64

Tabelle 1: Schranken für Covering Codes

7 Graphentheorie

In diesem Kapitel werden Grundlagen der Graphentheorie angeführt, sodass die Property B sinnvoll definiert werden kann.
$[X]^r = \{Y : Y \subseteq X, |Y| = r\}$
Es sei V eine Menge von Knotenpunkten.

Definition 7.0.3. *[ES74]S.15*
Ein Menge $G \subseteq [V]^2$ wird Graph genannt, die Elemente von G werden Kanten genannt.

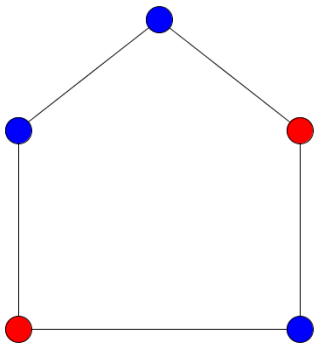

Abbildung 6: Beispiel für einen einfachen Graphen mit 5 Knoten und 5 Kanten.

Die Kanten werden als $\{x, y\}$ angeschrieben, wobei $x, y \in V$, also definiert sich eine Kante durch ihre beiden Endpunkte. Es sei für $x \in G$ weiters:

Definition 7.0.4. *[ES74]S.15*
$(x) = |\{y \in V : \{x, y\} \in G\}|$

Also bezeichnet (x) die Anzahl der Knoten, die mit x durch eine Kante aus G verbunden sind.

Definition 7.0.5. *[ES74]S.15*
Falls $W \subseteq V$ dann ist $G|W = G \cap [W]^2$ der durch W erzeugte Teilgraph.

Ein Teilgraph H eines Graphen G ist eine Teilmenge von G.

Definition 7.0.6. *[ES74]S.15*
Eine Menge $W \subseteq V$ wird unabhängig genannt, falls $G \cap [W]^2 = \emptyset$
Eine Menge $W \subseteq V$ wird Clique genannt, falls $G \cap [W]^2 = [W]^2$

Definition 7.0.7. *[ES74]S.15*
Die Cliquenzahl $\omega(G)$ eines Graphen ist die maximale Anzahl an Knoten die eine Clique von G haben kann.

Ein Graph ist ein Spezialfall eines k-Graphen, wobei k = 2 gilt. In diesem Fall hat eine Kante genau zwei Eckpunkte. Im Allgemeinen kann man einen Hypergraph wie folgt definieren:

Definition 7.0.8. *[ES74]S.15*
Ein Hypergraph G auf einer Knotenmenge V ist eine Teilmenge von 2^V. Die Elemente von G werden Hyperkanten genannt. Ein k-Graph ist eine Teilmenge von $[V]^k$. Ein 2-Graph wird Graph genannt.

Die restlichen Definitionen für Hypergraphen verlaufen analog zu einem Graphen.

Definition 7.0.9. *[ES74]S.15*
Eine r-Färbung eines (Hyper)graphen ist eine Funktion $h : G \to [r]$ wobei für alle (Hyper)kanten $e \in G$ es $x, y \in e$ gibt, sodass $h(x) \neq h(y)$. Das kleinste r, für welches so eine r-Färbung gefunden werden kann wird chromatische Zahl $\chi(G)$ genannt.

Die Suche nach einer r-Färbung eines k-Graphen ist der Versuch, jeden Knoten eines k-Graphen mit r Farben so zu färben, dass er sich von einem seiner Nachbarn unterscheidet. Als Nachbarn seien hier Knoten bezeichnet, die durch eine Kante verbunden sind. Im Falle eines Graphen bedeutet dies, dass kein Knoten dieselbe Farbe wie eine seiner Nachbarn hat. Häufig werden 2-Färbungen betrachtet, also $h : G \to \{+1, -1\}$ Der vorherige Graph hat keine zulässige 2-Färbung. Ein Beispiel für einen 2-gefärbten Graph:

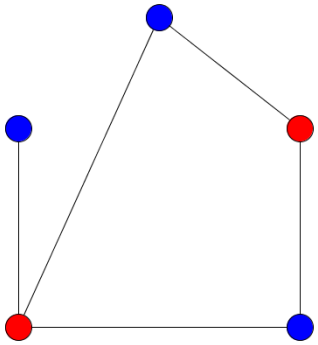

Abbildung 7: Dieser Graph hat eine zulässige 2-Färbung

8 Property B

Mit diesen Grundlagen kann nun die Property B für Graphen definiert werden.

8.1 Definition

Definition 8.1.1. *[ES74]S.19*
Sei G ein Hypergraph. G hat die Property B, falls $\chi(G) \leq 2$ gilt.

Definition 8.1.2. *[ES74]S.19*
m(n) ist das kleinste |G| welches ein n-Graph G ohne Property B haben kann.

Konkret heißt dies: "Wie viele Knoten muss ein n-Graph zumindest haben, sodass eine 2-Färbung nicht möglich ist?"
Dieses Problem kann auch für beliebiges $\chi(G)$ gestellt werden. Die Tatsache, dass die Property B auch völlig ohne Graphentheorie mit Hilfe der Mengenlehre formuliert werden kann ist interessant.

Definition 8.1.3. *[ES74]S.19*
Property B für Mengen:
Sei X eine Menge mit N Elementen und sei C eine Familie von k-Elementigen Teilmengen von X. C hat die Property B, falls X in 2 disjunkte Mengen Y und Z zerlegt werden kann, sodass jede Menge in C einen nichtleeren Schnitt mit Y und Z hat. Hier bezeichnet m(n) die minimale Anzahl von Elementen in den Submengen von C (also das kleinste k), sodass ein C zusammengestellt werden kann, welches die Property B nicht hat.

Dieses Problem ist zum Stand dieser Arbeit lediglich bis zum 4-Graphen bzw. zu 4-elementigen Untermengen eindeutig beantwortet. Die Schwierigkeit beim Nachweis, dass die Property B nicht hält, liegt darin, dass explizit nachgewiesen werden muss, dass eine 2-Färbung nicht möglich ist. Dies kann aus der Existenz einer 3-Färbung nicht geschlossen werden. Dies bedeutet, dass in erschöpfender Suche jede Möglichkeit, einen Graphen mit beliebigen Kanten in 2 Farben einzufärben, darauf getestet werden muss, ob die Nachbarschaftsbedingung verletzt ist. Im mengentheoretischen Fall übersetzt sich das Problem auf die Frage, ob für eine Familie C keine Partition von X möglich ist, sodass alle Mengen aus C nichtleeren Durchschnitt mit Y und Z haben.

8.2 Schranken

Ab 5-Graphen sind nur mehr Schranken bekannt. Schranken für Property B sind Konfigurationen von Hypergraphen (also Anzahl an Knoten und Kanten eines n-Graphes), sodass die Property B gilt. Spencer und Erdös [ES74] leisteten einen wichtigen Beitrag bei der analytischen Suche von Schranken:

Satz 8.2.1. *[ES74]S.19*
$m(n) \geq 2^{n-1}$

Beweis. Sei G ein beliebiger aber fixer Hypergraph, $|G| = f$. Sei $C = (K_1, K_2)$ eine 2-Färbung des Graphen, sodass $K_1 \cup K_2 = M$ und $K_1 \cap K_2 = \emptyset$. Sei $\beta(C)$ die Anzahl der einfärbigen $A \in G$ unter der Färbung C. Sei M die Vereinigung aller Knotenpunkte von G.

$$\beta(C) = \sum_{A \in G} \sum_{i=1}^{2} \sum_{A \subset K_i} 1 \tag{8.2.1}$$

Sei nun $R = (R_1, R_2)$ eine beliebige 2-Färbung. Dann gilt

$$\mathbb{E}[\beta(R)] = \sum_{A \in G} \sum_{i=1}^{2} \mathbb{P}[A \subseteq R_i] = \sum_{A \in G} \sum_{i=1}^{2} 2^{-n} = f \cdot 2^{-n+1} \tag{8.2.2}$$

Falls G nun nicht die Property B für alle hat, dann gilt für alle R: $\beta(R) \geq 1$. womit

$$1 \leq \mathbb{E}[\beta(R)] = f \cdot 2^{-n+1} \tag{8.2.3}$$

und damit

$$2^{n-1} \leq f \tag{8.2.4}$$

□

Dieses Ergebnis wurde ein wenig verbessert:

Satz 8.2.2. *[ES74]S.20*
$m(n) \geq 2^n(1+2n^{-1})^{-1}$

Beweis. Sei G ein n-graph ohne property B. Sei C abermals eine zufällige Färbung von M. Falls ein $N \in G$ einfärbig ist, dann wechsle die Farbe eines Punktes $x_N \in N$. Jedem $N \in G$ wird ein $x_N \in N$ zugeordnet, sodass

$$|\{L \in G, L \cap N = \{x_N\}\}| < fn^{-1} \tag{8.2.5}$$

Es gibt einige $x \in N$ welche diese Gleichung erfüllen, da sonst

$$|G| > |\{L \in G : |L \cap N| = 1\}| \geq n(fn^{-1}) \tag{8.2.6}$$

Für jede beliebige Färbung $C = (K_1, K_2)$ gilt entweder $\beta(C) \geq 2$ oder es gibt ein eindeutiges $N \in G$ i = 1 oder 2, $N \subseteq K_i$. Sei $C' = (K_1 \Delta\{x_N\}, K_2 \Delta\{x_N\})$. Da $\beta(C') \geq 1$ existiert ein einfärbiges $L \in G$ in C'. Dies tritt nur ein, sofern $L \cap K_i = \{x_N\}$ gilt. Es gilt somit für alle Färbungen C:

$$\frac{\beta(C)}{2} + \sum_{N \in G} \sum_{\substack{L \in G \\ L \cap N = \{x_N\}}} \sum_{i=1}^{2} \sum_{\substack{N \subseteq K_i \\ L \cap K_i = \{x_N\}}} \frac{1}{2} \geq 1 \tag{8.2.7}$$

Betrachte nun eine Färbung $C = (K_1, K_2)$. $\mathbb{E}[\beta(C)] = f2^{-n+1}$ gilt nach wie vor. Es gibt weniger als $f(fn^{-1})$ Möglichkeiten für N, L und i. Für jede Möglichkeit gilt

$$\mathbb{P}[N \subseteq K_i, L \cap K_i = \{x_N\}] = 2^{-(2n-1)} \tag{8.2.8}$$

Womit folgt

$$\mathbb{E}\left[\sum_{N \in G} \sum_{\substack{L \in G \\ L \cap N = \{x_N\}}} \sum_{i=1}^{2} \sum_{\substack{N \subseteq K_i \\ L \cap K_i = \{x_N\}}} \frac{1}{2}\right] < f(fn^{-1})2 \cdot 2^{-(2n-1)}2^{-1} \tag{8.2.9}$$

Benutze nun die beiden berechneten Erwartungswerte in 8.2.7, so folgt

$$f2^{-n} + f^2 n^{-1} 2^{-(2n-1)} \geq 1 \tag{8.2.10}$$

mit der Satz bewiesen ist. □

Die momentan beste Abschätzung für die untere Schranke wurde 2000 von Radhakrishnan, J.; Srinivasan, A. [RS00] verbessert:

Satz 8.2.3. *Sei n hinreichend groß und H ein Hypergraph wobei alle Kanten genau Dimension n haben. Dann ist H 2-färbbar, falls die Anzahl der Kanten nicht* $0.7 \cdot \sqrt{\frac{n}{ln(n)}} \cdot 2^n$ *überschreitet.*

Der vollständige Beweis ist Inhalt von [RS00], die Beweisidee soll jedoch skizziert werden:
Sei H ein Hypergraph, V seine Knoten und E die Kanten. Sei

$$\mathcal{M}(\chi) = \{f \in E : f\text{ist einfärbig in } \chi\} \tag{8.2.11}$$

und für $v \in V$ sei:

$$\mathcal{M}(v, \chi) = \{f \in \mathcal{M} : v \in f\} \tag{8.2.12}$$

Die Idee hinter dem Beweis ist nun der Versuch, einen 2-färbigen Graphen zu konstruieren. Dazu wird zunächst eine zufällige Einfärbung gewählt:

$$\chi_0 : V \to \{Rot, Blau\} \tag{8.2.13}$$

Hierbei ist $\chi_0(v)$ für jedes $v \in V$ jeweils rot oder blau mit Wahrscheinlichkeit 1/2. Anschließend werden zufällig Knoten ausgesucht, die Färbung dieses Knotens mit all seinen Nachbarn verglichen, und falls eine Kante einfärbig ist, dann wird der Knoten umgefärbt. In [RS00] wird eine neue Form der Färbung vorgeschlagen. Anstatt alle Knoten simultan zu färben und anschließend das Ergebnis zu untersuchen, werden zwei neue Funktionen eingeführt.
Die Funktion $delay : V \to [0, 1]$ ordnet jedem Knoten einen Verzögerungsfaktor aus dem Intervall [0,1] zu. Sie bestimmt die zeitliche Abfolge, in der die Knoten umgefärbt werden.
Die zweite Funktion $b : V \to \{0, 1\}$ wobei b(v) = 1 mit Wahrscheinlichkeit p und b(v) = 0 mit Wahrscheinlichkeit 1-p ist.

Nun kann V schrittweise umgefärbt werden:

- Schritt 1:
  Wenn $\mathcal{M}(v_1, \chi_0) \neq \emptyset$ und $b(v_1) = 1$, dann färbe v_1 um. Die neue Färbung wird χ_1 genannt.
- Schritt 2:
  Falls eine Kante aus $\mathcal{M}(v_2, \chi_0)$ auch unter χ_1 einfärbig ist und $b(v_2) = 1$ gilt, dann färbe v_2 um. Die neue Färbung wird χ_2 genannt.
  $\vdots$
- Schritt i:
  Falls eine Kante aus $\mathcal{M}(v_i, \chi_0)$ auch unter χ_{i-1} einfärbig ist und $b(v_i) = 1$ gilt, dann färbe v_i um. Die neue Färbung wird χ_i genannt.

Sei χ^* die resultierende Färbung, nachdem alle Knoten überprüft wurden. Dieses langsame Einfärben verhindert, dass bereits erzielte Verbesserungen nicht wieder durch weitere Umfärbungen zunichte gemacht werden.
In [RS00] wird weiters eine Analyse der Wahrscheinlichkeiten des Umfärbens durchgeführt, welche die Aussage des Theorems bestätigt.
Es folgt nun noch eine Abschätzung in die Gegenrichtung. Hier kommt die stochastische Brute Force Suche zum Einsatz:

Satz 8.2.4. *[ES74]S.20*
$m(n) \leq (1 + o(1))e(log(2))n^2 2^{n-2}$

Beweis. Sei $r = [\frac{n^2}{2}]$ und $A_1, A_2, \ldots$ unabhängige Zufallszahlen aus $[r]^n$. Sei $F = F_f = \{A_1, \ldots, A_f\}$. Für beliebige Färbung $C = (K_1, K_2)$ und beliebiges i gilt

$$\begin{aligned}
\mathbb{P}[A_i \text{ ist einfärbig}] &= \frac{\left[\binom{|K_1|}{n} + \binom{|K_2|}{n}\right]}{\binom{r}{n}} \\
&> \frac{2 \cdot \binom{r/2}{n}}{\binom{r}{n}} \\
&> 2e^{-1}2^{-n}(1 + o(1))
\end{aligned}$$

Da die A_i unabhängig sind, gilt

$$\mathbb{P}[\text{F ist durch C eingefärbt}] < (1 - 2e^{-1}2^{-n}(1 + o(1)))^f \tag{8.2.14}$$

und somit

$$\mathbb{E}[|\{\text{F ist durch C zweigefärbt}\}|] < (1 - 2e^{-1}2^{-n}(1 + o(1)))^f \tag{8.2.15}$$

Setze $f = (1 + o(1))e(log2)n^2 2^{n-2}$. Die erwartete Anzahl an 2-Färbungen für F ist kleiner als die Vereinigung. Somit existiert ein $F_f, |F_f| \leq f$, welches nicht 2-gefärbt werden kann. □

9 Simulated Annealing für Covering Codes

9.1 Beschreibung des Algorithmus

Der Algorithmus wurde in C realisiert. Am Beginn des Programmes steht eine Benutzereingabe. Gefragt werden die Werte

- R ... der Radius der Kugeln, mit denen der Raum überdeckt wird.
- K ... die Anzahl der Kugeln die den Raum überdecken sollen.
- L ... die Anzahl der Inneren Schleifen
- N ... die Länge der Codes im Coderaum.

Die Innere Schleife wird in der Programmbeschreibung noch detailliert ausgeführt. Nach der Benutzereingabe wird die Variable $nc = 2^N$ gesetzt, die Anzahl der Codewörter die nicht überdeckt sind. Da das Programm ausschließlich binäre Codes berechnet, entspricht dies genau der Anzahl an Codewörtern im Coderaum mit Länge N. Nun werden K zufällige Punkte als Startpunkte gewählt. Die add Funktion berechnet nun anhand von N und R für jeden Startpunkt, welche Codewörter durch die Kugeln mit Radius R überdeckt werden, und zieht die Anzahl von nc ab. Nun fängt die Suche mittels Simulated Annealing an. Als Starttemperatur wird T = 0.01 gewählt. In der Inneren Schleife wird einer der vorhandenen Punkte zufällig gewählt, aus der Liste der Covering Codes mittels der Funktion rem entfernt und durch einen neuen zufälligen Punkt ersetzt. Es wird die Anzahl der überdeckten Codewörter miteinander verglichen, und wie folgt weiterverfahren:

- Fall 1: Es werden mehr Codewörter als vorher überdeckt.
  Dies stellt eine Optimierung der alten Codes dar womit der neu gewählte Punkt den alten als Codewort im Covering Code ersetzt.
- Fall 2: Es werden weniger Codewörter als vorher überdeckt.
  Es sei $d < 0$ die Differenz zwischen der Überdeckung durch das alte Codewort und dem Neuen. Da es sich bei dem neuen Covering Code um einen weniger optimalen Code handelt, wird der neue Punkt mit einer Wahrscheinlichkeit von $exp(-T * d)$ angenommen. Dazu wird von der rand() Funktion (Teil des C Compiler) eine gleichverteilte Zufallsvariable generiert und geprüft, ob sie kleiner als $exp(-T * d)$ ist. Falls dies zutrifft, wird das neue Codewort trotz der Suboptimalität gewählt.

In beiden Fällen wird am Ende die neue Anzahl der überdeckten Codewörter berechnet und in die Konsole sowie in die Datei geschrieben. Diese Innere Schleife wird nun L-fach wiederholt.

Sobald der letzte Schritt in der inneren Schleife beendet ist, wird die Temperatur mit 1.01 multipliziert, also um den Faktor 0.01 geändert. Dies ist zwar eine Erhöhung, beachtet man jedoch das negative Vorzeichen in der Exponentialfunktion, so bewirkt diese Erhöhung letzten Endes ein Absinken der Temperatur, da die Akzeptanzschwelle wie gewünscht immer geringer wird. Dies wurde so gestaltet, um eine Division durch einen Wert nahe 0 zu vermeiden, da dies numerisch problematisch ist und vermieden werden sollte. Anschließend wird abermals mit der Inneren Schleife begonnen. Diese Prozedur wird solange fortgesetzt, bis die Temperatur den Wert 10 überschreitet. Es gibt nun zwei Optionen, wie das Programm zu seinem Ende findet:

- Fall 1: nc = 0
  Das Programm prüft vor jedem Abschluss eines Schrittes in der Inneren Schleife, ob nc den Wert 0 erreicht hat. Falls dies der Fall ist, sind alle Codewörter überdeckt und das Programm beendet mit dem gefundenen Covering Code.

- Fall 2: Die Temperatur überschreitet 10
  Falls diese Schwelle erreicht wird, bricht das Programm erfolglos ab.

In beiden Fällen kann in der Konsole sowie der abgespeicherten Datei nun der letzte Wert für nc eingesehen werden und der Covering Code, der zu diesem Wert geführt hat.

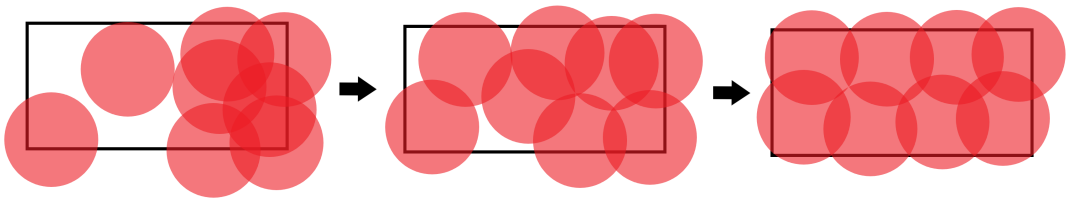

Abbildung 8: Die Codewörter (=Kreise) werden so lange verschoben, bis der ganze Coderaum (=Rechteck) überdeckt ist.

9.1.1 add()

Die Funktion add(i,x) dient dazu an der Stelle i den Punkt x als Teil des Covering Codes zu speichern und gleichzeitig alle Codewörter die nun überdeckt sind zu speichern und die Gesamtanzahl der Überdeckungen als Ganzzahl zurückzugeben. Für allgemeines R untersucht die Funktion jedes Codewort im Raum (eine Schleife die 1 bis 2^N durchläuft), wendet die XOR-Operation auf das Codewort und den Punkt x an, und berechnet dann mittels __builtin_popcount die Anzahl an 1er des XOR-Produktes. Falls diese Anzahl $\leq R$ ist, so wird dieser Punkt als überdeckt markiert. Für die Fälle R = 1,2,3 wurde ein spezielleres Verfahren angewendet:
In allen drei Fällen wird zunächst der Punkt x selber als überdeckt markiert, da der Mittelpunkt der Kugel immer in der Kugel ist, unabhängig vom Radius. Anschließend wird das XOR Produkt von x und dem Codewort 000 ... 001 gebildet. Dieses Produkt ist das Codewort, welches sich ausschließlich in der ersten Zahl von x unterscheidet. Da somit die Distanz 1 ist, wird dieser Punkt in allen drei Fällen als überdeckt eingetragen. Nun wird der 1er um eine Stelle nach links geschoben und abermals das XOR Produkt mit x gebildet. Diese Vorgehensweise wird solange weitergeführt, bis alle Codewörter der Länge N mit einem 1 Bit abgearbeitet sind. Für den Fall R = 1 wäre die Funktion add fertig. Im Fall 2 müssen in analoger Weise alle Codewörter der Länge N mit zwei 1er Bits mit x per XOR verknüpft werden und die generierten Codewörter als überdeckt markiert werden. Der Fall R = 3 wird in ähnlicher Art bearbeitet. Zuletzt wird die Gesamtzahl der neu überdeckten Punkte als Ganzzahl von der Funktion zurückgegeben. Hierbei sei darauf hingewiesen, dass ein Punkt nur dann die Gesamtzahl erhöht, falls er vorher nicht überdeckt war. Da die Kugeln bei einer Überdeckung nicht disjunkt sein müssen, können einzelne Codewörter von mehreren Kugeln überdeckt sein. Jeder Punkt wird trotzdem nur einmal gezählt.

9.1.2 rem()

Die Funktion rem(i) nimmt lediglich die Stelle i an. Anschließend nimmt sie das i-te Codewort im Covering Code und verfährt nahezu gleich der add Funktion, mit der Ausnahme dass die gefundenen Codewörter eben aus der Liste der überdeckten Codewörter entfernt werden. Auch gibt sie die Anzahl der nun nicht mehr überdeckten Wörter als Ganzzahl zurück. Auch hier wird diese G nur erhöht, falls eine Codewort nicht noch von weiteren Kugeln überdeckt wird.

9.2 Ergebnisse

Es wird versucht, mit dem so konstruierten Programm für verschiedene Dimensionen Covering Codes zu finden. Um die Ergebnisse vergleichbar zu machen, wird nc nicht als absolute Zahl ausgegeben, sondern als Anteil der nicht überdeckten Codes in Prozent. Getestet wurde:

- Für R = 1:
  (1,32,5000,8),(1,120,5000,10),(1,380,5000,12),(1,1408,5000,14)
- Für R = 2:
  (2,12,5000,8),(2,30,5000,10),(2,78,5000,12),(2,248,5000,14)
- Für R = 3:
  (3,4,5000,8),(3,12,5000,10),(3,28,5000,12),(3,64,5000,14)

Die gesuchte Anzahl der Codes ist an den momentan bekannten oberen Schranken orientiert. Damit ist garantiert, dass keine unmöglichen Codes gesucht werden, und die Laufzeiten und Ergebnisse vergleichbar sind.

Berechnungen für Covering Codes:

R \ N	**8**	**10**	**12**	**14**
1	0.626s	1.377s	6.4723s	67.9563s
2	1.694s	3.8696s	6.4553s	11.3273s
3	0.005s	7.4113s	19.6456s	36.5873s

Tabelle 2: Laufzeiten für K = 5000
Prozessor: Intel Core i7-4790K @ 4.00 GHZ (8 CPUs) und 8GB Ram

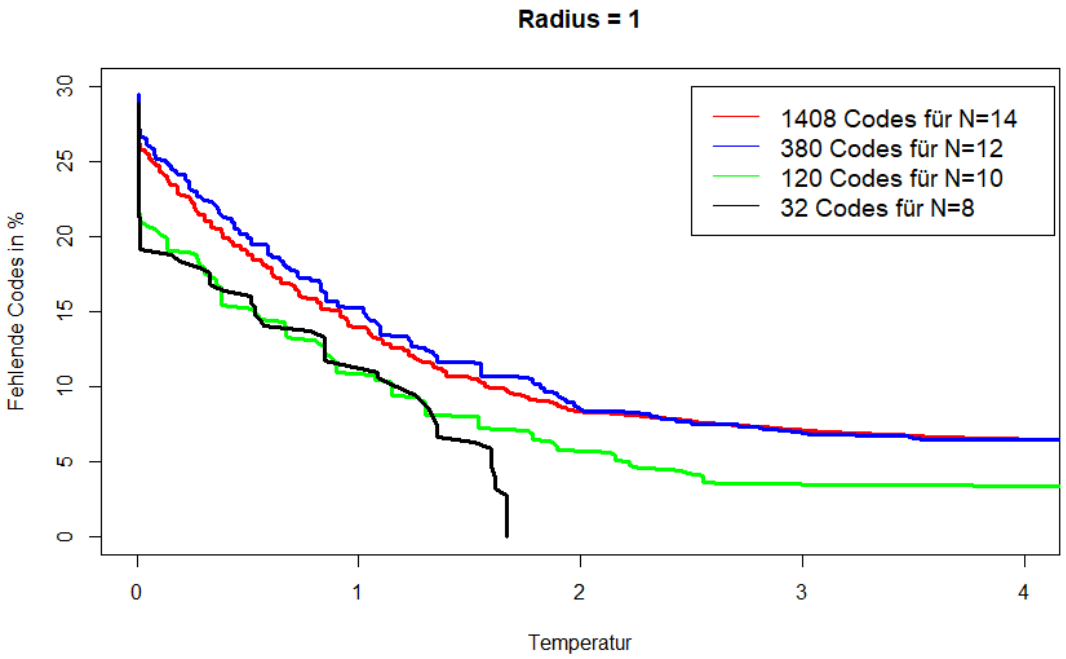

Abbildung 9: Covering Codes für Radius 1

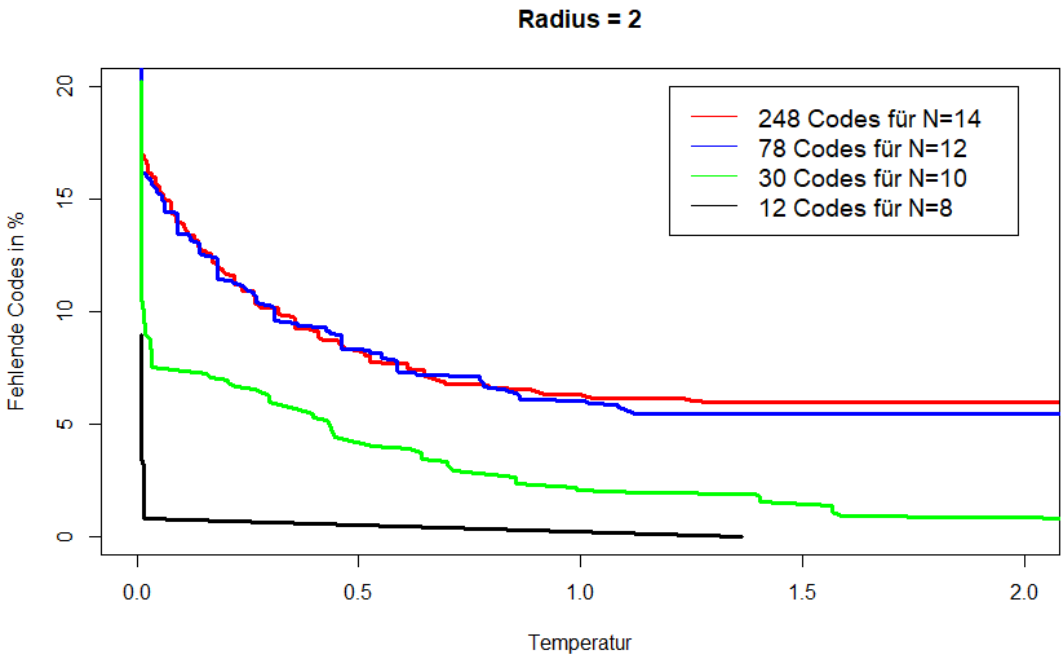

Abbildung 10: Covering Codes für Radius 2

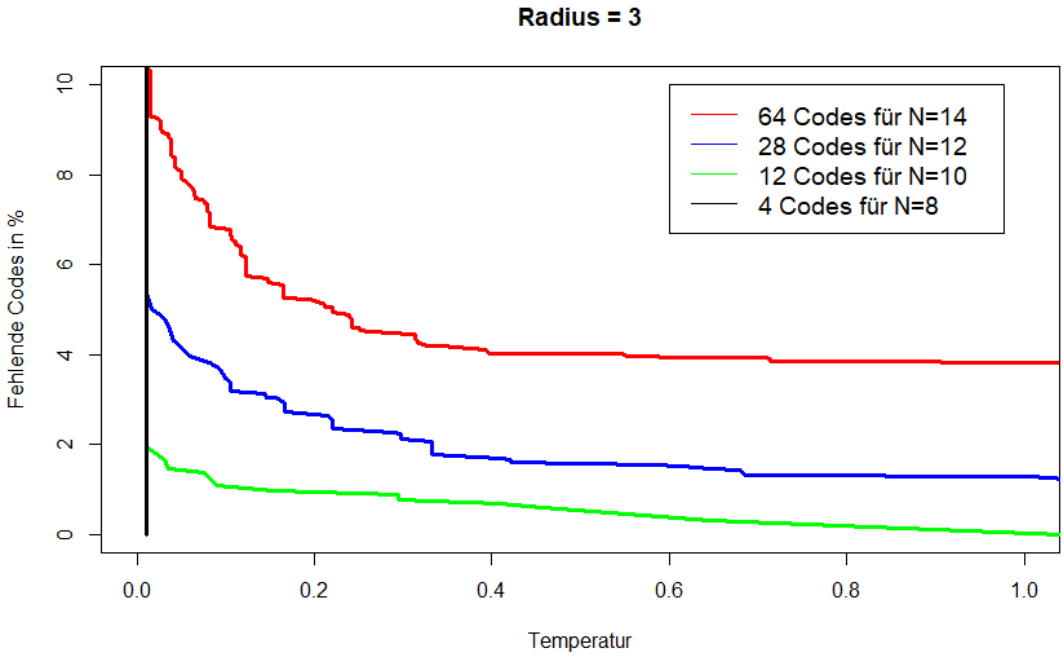

Abbildung 11: Covering Codes für Radius 3

Bei den Laufzeiten fällt auf, dass mit zunehmender Anzahl der Codewörter auch die Laufzeit dementsprechend steigt. Die durchaus nicht immer der Intuition entsprechenden Zeiten können wie folgt erklärt werden:
Je größer N ist, desto mehr Codewörter beinhaltet der Coderaum, dadurch müssen bei jedem Aufruf der add oder rem Funktion mehr Codewörter überprüft und gegebenenfalls markiert werden. Hier wirkt der angegebene Radius entgegen. Zwar verursacht ein größerer Radius R mehr Rechenzeit bei add und rem, allerdings werden durch jedes Codewort auch wesentlich mehr Codewörter überdeckt. Dadurch sind weniger Codewörter im Covering Code notwendig, und Verbesserungen können bei gleichem N schneller erzielt werden.
Die Plots stellen die Entwicklung von der Überdeckungsrate anhand der Abkühlung dar. Wenn 0 erreicht wird, hat das Programm einen Covering Code gefunden. Bei der Analyse sei darauf hingewiesen, dass zur besseren Lesbarkeit die Temperatur und die fehlenden Codes in verschiedenen Skalen aufgetragen sind.

- N = 8
  Hier konnte für alle Radien ein Covering Code gefunden werden. Dies drückt sich dadurch aus, dass die schwarze Linie stets 0 trifft. Bei Radius 1 konnten die ersten 32 zufällig gewählten Codes bereits ca. 70% des Coderaumes abdecken. Danach stellte sich schnell eine Verbesserung ein, bis eine vollständige Überdeckung gefunden werden konnte. Bei R = 2 konnten die ersten 12 Codewörter bereits eine Überdeckung von 92% erzielen, und schnell auf 98% verbessert werden. Danach war jedoch noch eine vergleichsweise lange Abkühlung nötig, um auch die letzten 2% zu erreichen. Für R = 3 hatten die Startcodewörter eine Abdeckung von 92%, die letzten konnten jedoch bereits im ersten Temperaturschritt ebenfalls abgedeckt werden.

- N = 10
  Für alle 3 Radien stellte sich zunächst eine ähnlich rapide Verbesserung wie bei N = 8 ein. Es konnte jedoch nur für R = 3 ein Covering Code gefunden werden. Bei R=1 konnten ca. 3% des Raumes nicht bedeckt werden, bei R = 2 blieben 1% unbedeckt. Es ist jedoch auffällig, dass ungefähr bei der Temperatur, in der für N = 8 eine Lösung gefunden wurde, die Verbesserungsrate bei N = 10 stark abnahm.

- N = 12
  Hier stellte sich für alle Radien eine logarithmisch abnehmende Verbesserungsrate ein. Das Verfahren nähert sich bei R = 1 94%, für R = 2 95% und für R = 3 99% Abdeckung an. Interessant ist, dass 90% Abdeckung in allen 3 Fällen schnell erreicht werden konnte. Dennoch bleibt die Rate über der des Falles N = 10.

- N = 14
  Dieser Fall war sowohl von der Laufzeit her als auch von der Genauigkeit der schlechteste. Auffallend ist hier, dass in den Fällen R = 1,2 die Verbesserungrate und die Qualität des Endergebnis der für N = 12 sehr ähnelt. Für Radius 1 war die Rechenzeit jedoch im Schnitt 10 Mal so lange. Im Fall R = 2 immerhin noch doppelt so lang. Für Radius R = 3 ist jedoch die Verbesserung deutlich schlechter als im Fall N = 10.

Zusammenfassend kann gesagt werden, dass vorallem die Dimension N des Coderaumes relevant für Qualität und Laufzeit sind. Dies hängt auch damit zusammen, dass ein größerer Raum bei gleichem Radius mehr Kugeln zur Überdeckung benötigt. Trotzdem konnte in allen Fällen eine Überdeckungsrate von zumindest 95% erreicht werden.
Mit der Einstellung (1,62,10000,9) konnte ein Covering Code gefunden werden. Nach der zur Verfügung stehenden Liste würde es sich hierbei um einen optimalen Covering Code handeln. Für diese Berechnung wurde jedoch beim Kühlplan die Starttemperatur $T_0 = 0.001$ gesetzt und $\alpha = 1.001$ gesetzt. Dies ist neben einer höheren Starttemperatur auch ein wesentlich langsameres Abkühlen. Der vollständige Code ist im Anhang B.1 zu finden.
Da die Schranken für R = 1 und N = 9 nach oben und unten 62 sind, ist der vorliegende Code ein perfekter.

10 Simulated Annealing für Property B

10.1 Beschreibung des Algorithmus

Auch dieser Algorithmus wurde in C realisiert. In seiner grundlegenden Funktionsweise ähnelt er der Suche nach Covering Codes, unterscheidet sich jedoch in einigen wichtigen Details. Als Eingabe wird verlangt:

- K ... Mächtigkeit der Submengen bzw. Dimension des K-Graphen
- n ... Anzahl der Submengen bzw. Anzahl der Knoten
- L ... die Anzahl der Inneren Schleifen
- N ... Mächtigkeit der Grundmenge bzw. Anzahl der Kanten.

Das Programm bedient sich einer speziellen Codierung, die es ermöglicht bitweise mit Mengen zu operieren. Die Kanten bzw. Elemente der Mengen können nummeriert werden. Mithilfe von Bitverschiebung kann nun je nach Nummerierung die Zahl 1 um die Nummer nach links verschoben werden. So produzieren die Zahlen 0,1,2,3 die Codes 1, 10, 100 und 1000. Dies wiederum entspricht den Mengen $\{1\}, \{2\}, \{3\}, \{4\}$. Zum Bespiel kann die Menge $\{1, 3, 4\}$ mit dem Binärcode 11010 dargestellt werden. So werden nun anhand von K und N alle möglichen Mengen ermittelt und in der Variable sets abgespeichert. Es wird ein Zeiger t generiert. Dieser Zeiger wird wie folgt befüllt:
Drei Schleifen arbeiten alle Mengen in sets und alle möglichen Partitionen der Grundgesamtheit durch. Dabei wird stets verglichen, ob es eine Menge gibt, die mit einer der Partitionshälften leeren Durchschnitt hat. Dazu wird der Code der Partition mit dem Code der Menge bitweise verglichen. Falls in keinem Bit eine Übereinstimmung besteht, liegen zwei disjunkte Mengen vor. In diesem Fall wird die binäre Codierung der Partition an der ersten Stelle des Zeigers gespeichert, der Zeigerindex um 1 erhöht und weitergesucht. Dies wird solange weitergeführt, bis für alle Mengen aus sets und alle Partitionsmöglichkeiten alle disjunkten Paare gefunden wurden. Schlussendlich wird $nc = 2^N$ gesetzt. Dies ist die Anzahl aller möglichen Partitionen.
Der Algorithmus versucht nun, für die angegebenen Dimensionen der Grundmenge und der Submengen die Property B nachzuweisen. Dazu wird die Familie C konstruiert, also n Mengen mit Mächtigkeit K gewählt. Rechnerisch wird für jede Menge eine Binärzahl kleiner 2^N mit Hamming-Gewicht K per Zufall gezogen. Dies ist notwendig, damit nur genau die Binärzahlen gezogen werden, die Mengen mit K Elementen darstellen. Dann werden für jede Menge alle möglichen Partitionen markiert, welche die Menge in einer der beiden Hälften omplett enthält.

Für diese Partitionen wird die Zählvariable um die Partitionen, die zum ersten Mal auftreten, reduziert. Es beginnt der Annealing-Vorgang. Beginnend bei $T_0 = 0.01$ wird jetzt eine Menge der Familie C mit einer zufälligen neuen mit Hamming-Gewicht K ausgetauscht. Nun werden die Partitionen neu markiert. Falls mehr Partitionen als vorher abgedeckt sind, so wird die neue Menge angenommen. Falls es zu einer Verschlechterung kommt, wird wie gewohnt die neue Menge mit der Wahrscheinlichkeit $exp(-T * d)$, wobei $d > 0$ den Verschlechterungsgrad darstellt. Dieser Vorgang wird L mal wiederholt, anschließend wird mit der Temperatur $T_1 = T_0 * 1.01$ fortgesetzt, bis die Endtemperatur $T_f = 10$ überschritten wird. Auch hier wird T eigentlich erhöht, dennoch handelt es sich um eine Abkühlung, da sich die Temperatur im Zähler der Boltzmannkonstante befindet. Auch hier ist dies numerisch vorteilhafter, da Divisionen generell fehleranfälliger als Multiplikationen sind. Auch hier gibt es zwei Optionen für den Algorithmus, wie er zu seinem Ende findet:

- Fall 1: nc = 0
  Das Programm prüft vor jedem Abschluss eines Schrittes in der Inneren Schleife, ob nc den Wert 0 erreicht hat. In diesem Fall sind alle Partitionen, die mit Mengen der Mächtigkeit K einen leeren Durchschnitt haben abgedeckt. Das bedeutet, dass keine Partition mehr übrig ist mit der eine Situation, in der die Property B hält, konstruiert werden kann. Somit wurde für die angegebenen Dimensionen eine Familie C gefunden, die mit der Grundgesamtheit Property B verletzt.

- Fall 2: Die Temperatur überschreitet 10
  Das Programm bricht erfolglos ab.

Auch hier gibt die Konsole in beiden Fällen die Familie C codiert in Dezimalzahl, die wiederum als Code für die vorher erläuterte Binärdarstellung dient, aus und speichert sie zusammen mit nc in einer Datei ab.

10.1.1 add()

add(i,x) nimmt als Eingabe einen Index i und den Code x einer Menge. Anschließend werden alle Partitionen, bei denen die Menge x zu einer Partitionshäflte disjunkt ist, als überdeckt markiert. Diese Information ist schon bekannt, da sie wie erwähnt zu Beginn der Programmroutine ermittelt wurde. Für alle Partitionen, die zum ersten Mal überdeckt werden, wird nc um 1 reduziert.

10.1.2 rem()

Die Funktion rem(i) löscht die i-te Menge aus der Liste. Hierbei wird bei allen Partitionen, die von dieser Menge überdeckt wurden, die Markierung entfernt, und für alle Partitionen, die nur von der aktuellen Menge überdeckt wurden, wird nc um die entsprechende Anzahl erhöht.

10.2 Suche nach Schranken

Dieses Programm versucht, für die Eingabe $[k, n, L, N]$ eine Konstellation zu finden, in denen Property B nicht hält. Sei nun $m(k, N)$ die Zahl n mit der eine Familie von k-Elementigen Mengen, zusammengestellt aus einer N-Elementigen Grundmenge, ohne Property B gefunden wurde. m(k) unterscheidet sich also von m(k,N) insofern, als m(k) eine Schranke für beliebiges N ist, und m(k,N) für jedes N einen eigenen Wert hat. Es gilt:

Satz 10.2.1. *Sei m(K) die kleinste Anzahl an Mengen, für die bei einer Familie mit K-Elementigen Mengen die Property B nicht gilt. Sei $m(k, N)$ wie oben definiert. Dann gilt:*

$$m(k) \leq m(k, N) \tag{10.2.1}$$

Dies folgt direkt aus der Definition von m(k). Dies bedeutet, dass mit Simulated Annealing konsequent die obere Schranke verkleinert werden kann, solange neue Familien ohne Property B gefunden werden.
Die Schranke m(K) ist nur von der Mächtigkeit der Submengen (Mengen der Familie) abhängig. Im Gegensatz dazu muss für $m^*(k, N)$ auch die Mächtigkeit der Grundmenge angegeben werden.

10.2.1 m(5)

Die Property B ist für m(K), K = 1, ..., 4 geklärt. Für m(K) mit $K \geq 5$ sind nur Schranken bekannt. Nach den letzten Erkenntnissen von [AASS16] gilt:

$$29 \leq m(5) \leq 51 \tag{10.2.2}$$

Im weiteren Verlauf werden einige Überlegungen angestellt, wie man für K = 5 bei vorgegebenem N diese allgemeinen Schranken etwas verschärfen kann.

Satz 10.2.2. *Es gelte $N \geq 2k+1$. Sei m(k,N) das kleinste m, sodass es m k-Elementige Mengen aus einer Grundmenge mit Mächtigkeit N gibt, für die Property B nicht gilt. Dann gilt:*

$$m \geq \frac{\binom{N}{k}}{\binom{N-\lfloor n/2 \rfloor}{k} + \binom{\lfloor n/2 \rfloor}{k}} \tag{10.2.3}$$

Beweis. Sei eine k-Elementige Menge X fixiert. Betrachte nun eine l-elementige Menge A, welche mit A^c eine Partition von $\{1, 2, \ldots, N-1, N\}$ bildet. Die Anzahl der Möglichkeiten A so zu wählen, dass A oder A^c leeren Schnitt mit X hat ist:

$$\binom{N-k}{l} + \binom{N-k}{l-k}$$

Für m Mengen folgt damit:

$$m\left(\binom{N-k}{l} + \binom{N-k}{l-k}\right) \geq \binom{N}{l}$$

Dies ist das Schubfachprinzip. Die Anzahl der möglichen Partitionen A, A^c der Grundmenge kann nicht größer sein, als die Anzahl aller Mengen die mit einem beliebigen A bzw. seinem Komplement leeren Schnitt haben.
Angenommen es gibt eine solche Partition A, A^c, für die es keine k-Elementige Menge X gibt, welche disjunkt zu A oder A^c ist. Da $N \geq 2k+1$, gilt zumindest für eine Partitionshälfte, dass sie mehr als k Elemente hat. Sei dies o.B.d.A. A. Es lässt sich nun iterativ solange ein beliebiges Element aus A entfernen, bis eine Menge mit k Elementen generiert wurde, die eine Teilmenge von A ist. Nach Definition

ist sie somit disjunkt zu A^c. Dies wiederspricht der Annahme.
Einige Umformungsschritte führen zu:

$$\begin{aligned} m &\geq \frac{\binom{N}{l}}{\binom{N-k}{l} + \binom{N-k}{l-k}} \\ &= \frac{\frac{N!}{l!(N-l)!}}{\frac{(N-k)!}{l!(N-k-l)!} + \frac{(N-k)!}{(l-k)!(N-l)!}} \\ &= \frac{\frac{N!}{k!(N-k)!}}{\frac{(N-l)!}{k!(N-k-l)!} + \frac{l!}{k!(l-k)!}} \\ &= \frac{\binom{N}{k}}{\binom{N-l}{k} + \binom{l}{k}} \end{aligned}$$

Dieser Term ist bei $l = \lfloor N/2 \rfloor$ am kleinsten. □

Für allgemeines $N = 2k + d$ gilt für den Zähler:

$$\binom{N}{k} = \binom{2k+d}{k} = \frac{(2k+d)!}{k!(2k+d-k)!} = \frac{(2k+d)!}{k!(k+d)!} = \frac{\prod_{i=1}^{d+1}(k+i)}{\prod_{i=1}^{d}(2k+d+i)} \frac{(2(k+d))!}{(k+d+1)!(k+d)!} = \frac{\prod_{i=1}^{d+1}(k+i)}{\prod_{i=1}^{d}(2k+d+i)} C_{k+d}$$

Wobei $C_n = \frac{(2n)!}{(n+1)!n!}$ die Catalan-Zahlen sind.

Es folgt somit für k = 5:

$$m(5,11) \geq 66 \tag{10.2.4}$$
$$m(5,12) \geq 66 \tag{10.2.5}$$
$$m(5,13) \geq 47.\dot{5}$$
$$m(5,14) \geq 47.\dot{5}$$
$$m(5,15) \geq 39$$
$$m(5,16) \geq 39$$
$$m(5,17) \geq 34$$
$$m(5,18) \geq 34$$
$$m(5,19) \geq 30.76$$
$$m(5,20) \geq 30.76$$

Ab N = 21 unterschreitet m(k,N) die bekannte Schranke m(k) und ist deswegen nicht mehr brauchbar.
Der Abkühlplan wurde hier wie folgt gewählt:

- $T_0 = 0.01$
- $T_f = 10$
- $T_{n+1} = T_n \cdot \alpha$ mit $\alpha = 1.001$

Mit den Parametern [5,66,100,11] wurde für N = 11 eine Familie gefunden. Diese ist im Anhang A.1 zu finden.
Weiters konnte für N = 12 mit [5,66,2000,12] eine Familie generiert werden, welche im Anhang A.2 zu finden ist.
Damit folgt:

$$m(5,11) \leq 66$$
$$m(5,12) \leq 66$$

In Kombination mit 10.2.4 und 10.2.5 führt dies zu

$$\mathbf{m(5,11) = 66}$$
$$\mathbf{m(5,12) = 66}$$

Dadurch sind für $K = 5$ und $N = 11, 12$ die Anzahl der Submengen auf 66 festgelegt. Diese Ergebnisse lassen zusammen mit dem Wissen, dass $m(5) \leq 51$, den Schluss zu, dass für m(5) die Submengen aus einer Grundgesamtheit mit Mächtigkeit $N \geq 13$ gebildet werden müssen.

10.2.2 Laufzeitanalyse

Abschließend wird die Laufzeit dieses Algorithmus für verschiedene Eingaben getestet. Da das Verfahren probabilistisch vorgeht, wurde jede Eingabe öfter getestet und anschließend gemittelt. Folgende Eingaben wurden getestet:

- (5,66,K,11) für $K \in \{100, 2000, 5000, 7500, 10000\}$
- (5,66,K,12) für $K \in \{2000, 5000, 7500, 10000\}$
- (5,66,K,13) für $K \in \{5000, 7500, 10000\}$

Die kleinsten K wurden jeweils so gewählt, dass noch ein Covering Code gefunden werden konnte. Diese wurden nur heuristisch getestet und stellen keine absoluten Schranken dar. Der im vorherigen Abschnitt verfeinerte Kühlplan wird auch hier eingesetzt. Die Ergebnisse lauten:

N \ K	**100**	**2000**	**5000**	**7500**	**7500**
11	0.783s	10.143s	23.910s	36.134s	47.394s
12	-	28.790s	56.241s	114.152s	115.287s
13	-	-	153.570s	220.899s	309.251s

Tabelle 3: Laufzeiten bei der Suche nach Familien mit k = 5
Prozessor: Intel Core i7-4790K @ 4.00 GHZ (8 CPUs) und 8GB Ram

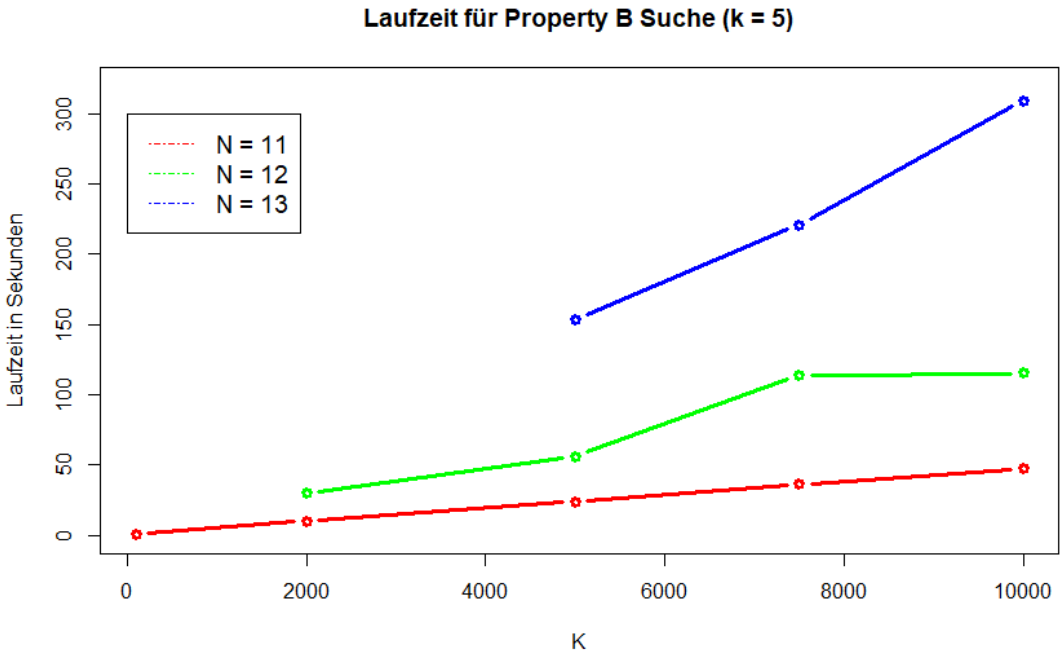

Abbildung 12: Laufzeiten für die Suche nach Property B

Es ist zu erkennen, dass mit zunehmender Dimension N die Laufzeit wächst. Während die Erhöhung von K eher lineares Wachstum erkennen lässt, weist die Erhöhung von N eine exponentielle Zunahme auf. Die wird deutlich, wenn man die Struktur des Algorithmus analysiert. Eine Erhöhung von K auf K + 1 verursacht lediglich einen zusätzlichen Mengentausch und einen weiteren Vergleich. K + d sind demnach nur d-mal mehr Mengenvorschläge und -vergleiche. Bei der Erhöhung der Dimension N auf N + 1 wird jedoch der ganze Raum vergrößert. Dadurch entstehen wesentlich mehr mögliche Partitionen, die geprüft werden müssen, mehr mögliche Submengen und damit auch mehr Kombinationsmöglichkeiten, die in Frage kommen. Mit einem gröberen Kühlplan könnte dem entgegengewirkt werden, davon ist aber abzuraten, da mit $\alpha = 1.01$ bereits die Mengenfamilie für k = 5 und N = 12 nicht mehr gefunden werden konnte. Auch eine massive Erhöhung von K (getestet bis 100000) konnte dem nicht entgegenwirken. Eine langsame Abkühlung ist auf jeden Fall notwendig, um Defekte zu vermeiden.

11 Conclusio

Simulated Annealing hat sich als ein leistungsfähiges Werkzeug erwiesen, mit welchem Probleme, bei denen der Definitionsbereich nicht konsequent abgearbeitet werden kann, gut behandelbar sind. Bei den Covering Codes konnten für Radius R = 1,2 und 3 sowie Dimension N = 8, 10, 12, 14 Fällen zumindest eine Genaugkeit von 95% erzielt werden. In vielen Fällen war eine Überdeckungsrate von 99% möglich. Für den Fall R = 1 und N = 9 konnte ein perfekter Covering Code mit 62 Codewörtern gefunden werden. Hierbei konnte die Rechenzeit in einem annehmbaren Rahmen gehalten werden. Größere Rechenkapazitäten können die hier vorgelegten Ergebnisse verbessern. Bei der Analyse von Property B konnte für den Fall des 5-Graphen gezeigt werden, dass bei 11 und 12 Knoten genau 66 Kanten nötig sind, um einen Hypergraphen zu konstruieren, der nicht 2-färbbar ist. All diese Resultate wären aufgrund der enormen Größe des abzutastenden Raumes ohne Simulated Annealing nicht erzielbar gewesen.

Anhang

A Mengen ohne Property B

A.1 k = 5 und N = 11

{1,3,4,7,10} {2,7,8,9,11} {3,4,8,9,10} {1,4,5,7,9} {2,3,4,5,10} {1,2,4,7,8} {2,3,4,8,11} {1,4,6,7,11} {1,4,8,9,11} {2,3,5,6,11} {1,2,3,5,8} {1,3,4,6,8} {1,6,7,8,10} {1,3,7,8,9} {3,4,5,7,8} {1,2,5,7,10} {1,7,9,10,11} {1,3,6,9,11} {3,4,6,10,11} {2,3,4,6,7} {5,7,8,9,10} {2,5,6,7,8} {2,5,8,10,11} {1,3,5,6,7} {2,4,5,7,11} {1,5,7,8,11} {1,4,5,8,10} {1,2,4,5,6} {4,7,8,10,11} {2,3,9,10,11} {4,6,7,8,9} {3,6,7,9,10} {2,4,7,9,10} {4,5,6,7,10} {1,5,6,8,9} {1,2,3,6,10} {3,4,5,6,9} {1,5,6,10,11} {3,5,7,10,11} {1,2,3,7,11} {1,3,8,10,11} {2,6,7,10,11} {3,5,6,8,10} {2,3,7,8,10} {2,4,5,8,9} {4,5,9,10,11} {2,4,6,9,11} {3,4,7,9,11} {1,2,3,4,9} {2,3,5,7,9} {4,5,6,8,11} {2,4,6,8,10} {1,3,5,9,10} {1,4,6,9,10} {1,2,4,10,11} {6,8,9,10,11} {3,5,8,9,11} {1,2,6,8,11} {1,2,5,9,11} {1,2,8,9,10} {5,6,7,9,11} {1,2,6,7,9} {3,6,7,8,11} {1,3,4,5,11} {2,3,6,8,9} {2,5,6,9,10}

A.2 k = 5 und N = 12

{2,4,5,9,10} {1,2,3,7,12} {1,2,5,6,7} {2,3,4,6,12} {1,9,10,11,12} {3,6,9,11,12} {2,3,4,9,11} {2,3,9,10,12} {3,4,5,10,12} {5,7,9,10,12} {3,4,6,10,11} {1,2,4,9,12} {2,4,10,11,12} {1,3,5,9,12} {1,2,5,10,12} {2,4,6,7,9} {3,6,7,9,10} {1,6,7,10,11} {1,4,7,10,12} {3,4,7,9,12} {1,2,7,9,10} {1,6,7,9,12} {1,2,5,9,11} {1,3,4,9,10} {1,4,5,10,11} {3,4,5,6,9} {1,4,6,9,11} {4,6,9,10,12} {5,6,10,11,12} {4,5,6,7,10} {5,6,7,9,11} {1,3,6,10,12} {1,2,6,11,12} {2,6,7,10,12} {1,3,5,6,11} {3,5,6,7,12} {1,3,5,7,10} {1,2,3,4,5} {1,3,7,9,11} {2,3,5,7,9} {1,5,7,11,12} {2,3,5,11,12} {2,3,5,6,10} {2,7,9,11,12} {1,4,5,6,12} {1,2,3,10,11} {2,5,7,10,11} {2,4,5,7,12} {1,4,5,7,9} {2,5,6,9,12} {3,4,5,7,11} {3,5,9,10,11} {1,2,3,6,9} {4,6,7,11,12} {1,2,4,6,10} {1,3,4,11,12} {3,7,10,11,12} {2,3,4,7,10} {1,5,6,9,10} {4,5,9,11,12} {1,3,4,6,7} {2,3,6,7,11} {2,6,9,10,11} {2,4,5,6,11} {4,7,9,10,11} {1,2,4,7,11}

B Covering Codes

B.1 R = 1 und N = 9

010000000,011110010,001101000,001010101,100001101,011100001,111111101
001100100,111000101,010111000,010000111,111010110,011101111,101000000
101110000,111101010,000010100,001111011,100101001,000011010,011001100
110101111,111100001,101011111,001111100,100110101,001000011,000101101
110110011,111001011,001001110,010101011,100100010,000001101,000100010
110010011,110010001,110111000,011110110,010101110,101101100,101100111
100010011,100000110,000110111,111110110,100111110,011010110,101111011
101010000,010110101,011011001,000110001,111001010,000001001,110001010
100001001,110100100,110011100,101011000,011010010,010011111

C Quellcode

C.1 Property B Quellcode

```
#include<stdio.h>
#include<stdlib.h>
#include<math.h>
#include<time.h>

int cv[1<<20];
int nc;
int x[20000];
int N;
int K;
int ns;
int sets[1<<20];
int L;
int n;
int *t;
double T,T0=0.01,T1=10,Tfact=1.001;
int bestn;
char fn[40];
FILE *f;
int count = 1;
int r(void)
{ //Generiere eine Zufallszahl, welche in die angegebenen Dimensionen passt.
  return (int)(((double)rand()/(RAND_MAX+1.0))*ns);
}

int add(int i,int xx)        //Die add-Funktion, wie in Kapitel 9 beschrieben
{
 int res=0;
 int j;

 for(j=0; j<(1<<(N-K)); j++)
 {
  cv[t[(xx << (N - K)) + j]]++;
```

```
  //Arbeite alle Partitionen durch, bei denen xx komplett in eine Partitionshälfte fällt.
  if (cv[t[(xx << (N - K)) + j]] == 1)        //Wenn die Partition nur durch xx bedeckt ist
  {
   res++;        //dann zähle diese Partition zu den bedeckten.
  }
 }
 x[i]=xx;        //Speichere die Menge xx am Index i
 return res;        //Gib zurück, wie viele Partitionen neu überdeckt wurden.
}

int rem(int i)
{
 int res=0;
 int j;

 for(j=0; j<(1<<(N-K)); j++)
 {
  cv[t[(x[i] << (N - K)) + j]]--;
  //Arbeite alle Partitionen durch, bei denen xx komplett in eine Partitionshälfte fällt.
  if (cv[t[(x[i] << (N - K)) + j]] == 0) //Wenn die Partition nur durch die Menge an der Stelle i bedeckt ist
  {
   res++;        //dann zähle diese Partition zu den unbedeckten.
  }
 }
 return res;        //Gib zurück, wie viele Partitionen nicht mehr überdeckt sind.
}

void w(void)
{
 int i;
 double coverage;
 coverage = 100*((double)bestn/(1 << N));  //Berechne den Prozentsatz der Mengen, die NICHT überdeckt sind.

 for (i = 0; i < n; i++)
 {
  printf("%d\n", sets[x[i]]);        //Gib alle Mengen in der Konsole aus.
 }

 //printf("bestn=%d\n",bestn);
 printf("nc=%f\n",coverage);        //Gib die fehlenden Partitionen in Prozent aus.

 for (i = 0; i < n; i++)
 {
  fprintf(f, "%d\n", sets[x[i]]);        //Schreibe alle Mengen in eine Datei.
 }

 fprintf(f,"%d,",count);   //Schreibe die aktuelle Ausgabenummer in die Datei (Jede Ausgabe erhöht count um 1)
 fprintf(f,"%f,",T);        //Schreibe die aktuelle Temperatur in die Datei
 //fprintf(f,"%d\n",bestn);
 fprintf(f,"%f\n",coverage);        //Schreibe die fehlenden Partitionen (in %) in die Datei
 count++;
}

int main(int argc, char** argv)
{
 int i,j,k,l,m,d;
 int counter;

 if(argc!=5)
 {
  printf("usage: sa K n L N\n");        //Brich ab falls keine Eingabe der Parameter
  exit(1);
```

```
 }

 K=atoi(argv[1]);
 n=atoi(argv[2]);
 L=atoi(argv[3]);        //Wandle die Parameter von string auf int um.
 N=atoi(argv[4]);

sprintf(fn,"s2-%d-%d-%d-%d",K,n,L,N);
f=fopen(fn,"w");
printf("K=%d n=%d L=%d N=%d\n",K,n,L,N);        //Bereite die Datei und Ausgabe vor

clock_t begin = clock();        //Starte die Uhr zur Zeitmessung

//#############################################################################################
//Konstruiere alle möglichen Mengen und speichere sie.
//#############################################################################################

for (i = 0; i < K; i++)
{
 x[i] = i;
}

for(ns=0; ;ns++)
{
 for (i = 0; i < K; i++)
 {
  sets[ns] += 1 << x[i];
 }

 for (j = K - 1; j >= 0 && x[j] == j + N - K; j--)
 {
 }
 if (j < 0) break;
 x[j]++;
 for (i = j + 1; i < K; i++)
 {
  x[i] = x[i - 1] + 1;
 }
}

ns++;

printf("ns=%d\n",ns);         //Schreibe die Überschriften
fprintf(f,"Index,");         //die ausgegebene Datei kann in R geladen warten,
fprintf(f,"Temperatur,",T);
        //falls die Mengen nicht in die Datei geschrieben werden und die Laufzeit am Ende entfernt wird.
fprintf(f,"nc\n",bestn);        //Mit diesen Daten wurden auch die Grafiken erstellt.

t=malloc(ns*sizeof(int)*(1<<(N-K)));        //Allokiere den Zeiger für add und rem

for (i = k = 0; i < ns; i++)
{
 for (j = 0; j < (1 << (N - 1)); j++)
 {
  if ((sets[i] & j) == 0 || (sets[i] & ~j) == 0)
  {
   t[k++] = j; //Speichere alle Partitionen, die für Property B und den Mengen von oben in Frage kommen.
  }
 }
}
srand(time(0));
nc = (1 << (N - 1));        //nc die Anzahl der nicht überdeckten Mengen.
```

```
for (i = 0; i < n; i++)
{
 nc -= add(i, r());        //Füge die ersten n Mengen zur Liste.
}

bestn=nc;
w();                              //Schreibe zum ersten Mal.

for (T = T0; T < T1; T *= Tfact)        //Der Abkühlplan
{
 for (l = 0; l < L; l++)        //Die innere Schleife
 {

  i = rand() % n;        //Bestimme einen zufälligen Index i
  k = x[i];        //Speichere die aktuelle Menge an i
  m = r();        //Erstelle neue zufällige Menge
  d = rem(i);        //Entferne die aktuelle Menge aus der Liste
  d -= add(i, m);        //Füge die neue Menge ein.

  if (d <= 0 || rand() < RAND_MAX *exp(-T*d))        //Bestimme ob akzeptiert wird.
  {
   nc += d;        //Vergleiche neuen Überdeckungsstand
   if (nc < bestn)        //Falls besser
   {
    bestn = nc; w();        //Speichere die neue Überdeckungszahl und rufe erneut w() auf.
   }
   if (nc == 0)        //Falls alles überdeckt ist
   {
    clock_t end=clock();                                                    //Stoppe die Uhr
    printf("Time taken:%lf",(double)(end-begin)/CLOCKS_PER_SEC);        //Schreibe die Laufzeit in die Konsole
    fprintf(f, "Time taken:%lf",(double)(end-begin)/CLOCKS_PER_SEC);        //und die Datei
    exit(0);                                                                //Beende das Programm
   }
  }
  else        //Wenn abgelehnt wurde
  {
   rem(i);        //entferne die neue Menge wieder
   add(i, k);        //und setze die alte Menge wieder ein.
  }
 }
}        //Wenn die finale Temperatur überschritten wurde
clock_t end=clock();        //Stoppe die Uhr
printf("Time taken:%lf",(double)(end-begin)/CLOCKS_PER_SEC);        //Schreibe die Laufzeit in die Konsole
fprintf(f, "Time taken:%lf",(double)(end-begin)/CLOCKS_PER_SEC);        //und die Datei
}
```


C.2 Covering Codes Quellcode

```
#include<stdio.h>
#include<stdlib.h>
#include<math.h>
#include<time.h>

double T,T0=0.01,T1=10,Tfact=1.01;

int nc;
```

```
int bestn;
int x[20000];
int cv[1 << 20];

int R;
int K;
int L;
int N;

char fn[40];
FILE *f;
int count = 1;

int r(void)
{
 return (int)(rand() % ((1 << N)-1));        //Wähle zufälligen Punkt für Überdeckungskugel
}

int add(int i,int xx)
{
 int res = 0;
 int hlp = xx;
 int j, k, l;

 if(R > 3)
 {

 }
 else
 {
  cv[hlp]++;
  if (cv[hlp] == 1)
  {
   res++;                //Im Fall R = 1,2,3 zähle den Mitelpunkt der Kugel zu den überdeckten Codes.
  }
 }

 if (R == 1)
 {
  for (j = 0; j < N; j++)
  {        //Laufe alle Codes durch, die sich nur in einem Bit von dem Covering Codewort unterscheidet.
   hlp ^= (1 << j);
   cv[hlp]++;                //Markiere sie als überdeckt.
   if (cv[hlp] == 1)
   {
        res++;                //Und zähle sie zu den überdeckten fall sie vorher nicht überdeckt waren.
   }
   hlp = xx;
  }
 }
 if (R == 2)
 {
  for (j = 0; j < N; j++)
  {
   hlp ^= (1 << j);
   cv[hlp]++;
   if (cv[hlp] == 1)
   {
    res++;
   }
   for (k = (j + 1); k < N; k++)
   {
```

```
    hlp ^= (1 << k);
    cv[hlp]++;
    if (cv[hlp] == 1)
    {
     res++;                //Hier passiert das selbe wie bei R = 1 für R = 2
    }
    hlp ^= (1 << k);
   }
   hlp = xx;
  }
 }
 if (R == 3)
 {
  for (j = 0; j < N; j++)
  {
   hlp ^= (1 << j);
   cv[hlp]++;
   if (cv[hlp] == 1)
   {
    res++;
   }
   for (k = (j + 1); k < N; k++)
   {
    hlp ^= (1 << k);
    cv[hlp]++;
    if (cv[hlp] == 1)
    {
     res++;
    }
    for (l = (k + 1); l < N; l++)
    {
     hlp ^= (1 << l);
     cv[hlp]++;
     if (cv[hlp] == 1)
     {
      res++;               //Hier passiert das selbe wie bei R = 1 und R = 2 für R = 3
     }
     hlp ^= (1 << l);
    }
    hlp ^= (1 << k);
   }
   hlp = xx;
  }
 }
 if (R > 3) //Bei R > 3 arbeite den ganzen Coderaum durch und sieh bei jedem Code nach, ob er überdeckt wird.
 {
  for (j = 0; j < (1 << N); j++)        //Überprüfe alle Codes im N-dimensionalen Coderaum
  {
   if (__builtin_popcount(j ^ xx) <= R)
   {//Zähle die Einser, falls <= R ist, dann wird j von der Kugel mit Zentrum xx überdeckt.
    cv[j]++;
    if (cv[j] == 1)
    {
     res++;
    }
   }
  }
 }
 x[i] = xx;                          //Speichere das neue Codewort
 return res;                          //Gib die Anzahl der neu überdeckten Codewörter zurück.
}
```

```
int rem(int i)
{
 int res = 0;
 int hlp = x[i];
 int j, k, l;

 if(R > 3)
 {

 }
 else
 {
 cv[hlp]--;
 if (cv[hlp] == 0)
 {
  res++;                          //Im Fall R = 1,2,3 zähle den Mitelpunkt der Kugel zu den überdeckten Codes.
 }
}

if (R == 1)
{
 for (j = 0; j < N; j++)
 {
  hlp ^= (1 << j); //Laufe alle Codes durch, die sich nur in einem Bit von dem Covering Codewort unterscheidet.
  cv[hlp]--;                //Markiere sie als überdeckt.
  if (cv[hlp] == 0)
  {
   res++;                //Und zähle sie zu den überdeckten fall sie vorher nicht überdeckt waren.
  }
  hlp = x[i];
 }
}
if (R == 2)
{
 for (j = 0; j < N; j++)
 {
  hlp ^= (1 << j);
  cv[hlp]--;
  if (cv[hlp] == 0)
  {
   res++;
  }
  for (k = (j + 1); k < N; k++)
  {
   hlp ^= (1 << k);
   cv[hlp]--;
   if (cv[hlp] == 0)
   {
    res++;                //Hier passiert das selbe wie bei R = 1 für R = 2
   }
   hlp ^= (1 << k);
  }
  hlp = x[i];
 }
}
if (R == 3)
{
 for (j = 0; j < N; j++)
 {
  hlp ^= (1 << j);
  cv[hlp]--;
  if (cv[hlp] == 0)
```

```
  {
   res++;
  }
  for (k = (j + 1); k < N; k++)
  {
   hlp ^= (1 << k);
   cv[hlp]--;
   if (cv[hlp] == 0)
   {
    res++;
   }
   for (l = (k + 1); l < N; l++)
   {
    hlp ^= (1 << l);
    cv[hlp]--;
    if (cv[hlp] == 0)
    {
     res++;        //Hier passiert das selbe wie bei R = 1 und R = 2 für R = 3
    }
    hlp ^= (1 << l);
   }
   hlp ^= (1 << k);
  }
  hlp ^= (1 << j);
 }
}
if (R > 3) //Bei R > 3 arbeite den ganzen Coderaum durch und sieh bei jedem Code nach, ob er überdeckt wird.
{
 for (j = 0; j < (1 << N); j++)        //Überprüfe alle Codes im N-dimensionalen Coderaum
 {
  if (__builtin_popcount(j ^ x[i]) <= R)        //Betrachte ob code j von betreffender Kugel überdeckt wurde.
  {
   cv[j]--;
   if (cv[j] == 0)
   {
    res++;
   }
  }
 }
}
return res;                //Gib die Anzahl der nicht mehr überdeckten Codewörter zurück
}

void w()
{
 int i;
 double coverage;
 coverage = 100*((double)bestn/(1 << N));//Berechne die Anzahl der Codes die NICHT überdeckt sind in Prozent
 for (i = 0; i < K; i++)
 {
 printf("%d\n", x[i]);               //Schreibe den aktuellen Covering Code in die Konsole
 }

 //printf("bestn=%d\n",bestn);
 printf("nc=%f\n",coverage);        //Schreibe die fehlende Überdeckung in Prozent

 for (i = 0; i < K; i++)
 {
  fprintf(f, "%d\n", x[i]);        //Schreibe den aktuellen Covering Code in die Datei
 }

 //fprintf(f,"bestn=%d\n",bestn);
```

```
 fprintf(f,"%d,",count);
 //Schreibe die Nummer des aktuellen Eintrages in die Konsole (wie oft w schon aufgerufen wurde)
 fprintf(f,"%f,",T); //Schreibe die aktuelle Temperatur in die Konsole
 //fprintf(f,"%d\n",bestn);
 fprintf(f,"%f\n",coverage); //Schreibe die fehlende Überdeckung in Prozent in die Konsole
 count++;
}

int main(int argc, char** argv)
{
 int i, k, l, m, mhelp, d, j, p, q;

 if (argc != 5)
 {
  printf("usage: sa R K L N\n");                        //Brich ab falls keine Eingabe vorhanden.
  exit(1);
 }

 R = atoi(argv[1]);        //Radius der Kugel
 K = atoi(argv[2]);        //Anzahl der Kugel
 L = atoi(argv[3]);        //Innere Schleife
 N = atoi(argv[4]);        //Länge der Codes.

 sprintf(fn, "s2-%d-%d-%d-%d", R, K, L, N);
 f = fopen(fn, "w");  //Bereite die Datei und die Eingabe vor
 printf("K=%d n=%d L=%d N=%d\n", R, K, L, N);

 nc = (1 << N);        //Anzahl der Codes die noch nicht überdeckt sind, das sind am Anfang alle.

 printf("nc=%d\n", nc);               //Schreibe die Überschriften
 fprintf(f,"Index,");                //die ausgegebene Datei kann in R geladen warten,
 fprintf(f,"Temperatur,",T);//falls die Codewörter und Laufzeit nicht in die Datei geschrieben werden.
 fprintf(f,"nc\n",bestn);        //Mit diesen Daten wurden auch die Grafiken erstellt.

 srand(time(0));

 clock_t begin = clock();                                        //Starte die Uhr

 for (i = 0; i < K; i++)
 {
  nc -= add(i, r());        //Füge nun Sphären ein mit Mittelpunkt r(). Radius ist vorgegeben.
 }
 bestn = nc;

 for (T = T0; T < T1; T *= Tfact)                         //Der Abkühlplan
 {
  for (l = 0; l < L; l++)                                         //Die Innere Schleife
  {

   i = rand() % n;                                  //Bestimme einen zufälligen Index i
   k = x[i];                                         //Speichere die aktuelle Menge an i
   m = r();                                         //Erstelle neue zufällige Menge
   d = rem(i);                                         //Entferne die aktuelle Menge aus der Liste
   d -= add(i, m);                                  //Füge die neue Menge ein.

   if (d <= 0 || rand() < RAND_MAX *exp(-T*d))               //Bestimme ob akzeptiert wird.
   {
    nc += d;                         //Vergleiche neuen Überdeckungsstand
    if (nc < bestn)                 //Falls besser
    {
     bestn = nc; w();        //Speichere die neue Überdeckungszahl und rufe erneut w() auf.
    }
```

```
    if (nc == 0)                        //Falls alles überdeckt ist
    {
     clock_t end=clock();  //Stoppe die Uhr
     printf("Time taken:%lf",(double)(end-begin)/CLOCKS_PER_SEC); //Schreibe die Laufzeit in die Konsole
     fprintf(f, "Time taken:%lf",(double)(end-begin)/CLOCKS_PER_SEC); //und die Datei
     exit(0);
    }
   }
   else                                              //Wenn abgelehnt wurde
   {
    rem(i);                                      //entferne die neue Menge wieder
    add(i, k);                                 //und setze die alte Menge wieder ein.
   }
  }
 }                                               //Wenn die finale Temperatur überschritten wurde
 clock_t end=clock();                                                  //Stoppe die Uhr
 printf("Time taken:%lf",(double)(end-begin)/CLOCKS_PER_SEC);         //Schreibe die Laufzeit in die Konsole
 fprintf(f, "Time taken:%lf",(double)(end-begin)/CLOCKS_PER_SEC);          //und die Datei
}
```

Literatur

[AASS16] Sachin Aglave, V. A. Amarnath, Saswata Shannigrahi, and Shwetank Singh. Improved bounds for uniform hypergraphs without property b. arXiv:1602.00218, 2016.

[CHLL97] G. Cohen, I. Honkala, S. Litsyn, and A. Lobstein. *Covering Codes*. North-Holland, 1 edition, 1997.

[DK00] Ding-Zhu Du and Ker-I Ko. *Theory of Computational Complexity*. John Wiley and Sons, 2000.

[ES74] Paul Erdös and Joel Spencer. *Probabilistic Methods in Combinatorics*. Akademiai Kiado, 1974.

[Gol10] Oded Goldreich. *P, NP and NP-Completeness, The Basic of Computational Complexity*. Cambridge University Press, 1 edition, 2010.

[Gri18] Karl Grill. Maß- und wahrscheinlichkeitstheorie. Vorlesungsskript, https://institute.tuwien.ac.at/fileadmin/t/mathstoch/upload/mw1.pdf (Zuletzt geöffnet am 12.03.2018 17:32), 2018.

[LA87] P.J.M van Laarhoven and E.H.L. Aarts. *Simulated Annealing: Theory and Applications*. D. Reidel Publishing Company, Dordrecht, Holland,, 1987.

[MGNR12] Thomas Müller-Gronbach, Erich Novak, and Klaus Ritter. *Monte Carlo-Algorithmen*. Springer, 2012.

[MPS15] Jithin Mathews, Manas Kumar Panda, and Saswata Shannigrahi. On the construction of non-2-colorable uniform hypergraphs. *Discrete Applied Mathematics*, 180:181–187, 2015.

[Nor97] J.R. Norris. *Markov Chains*. Cambridge University Press, 1997.

[Öst14] Patric R.J. Östergård. On the minimum size of 4-uniform hypergraphs without property b. *Discrete Applied Mathematics*, 163(2):199–204, 2014.

[Ros10] Sheldon M. Ross. *Introduction to Probability Models*. Elsevier, 10 edition, 2010.

[Rot08] Jörg Rothe. *Komplexitätstheorie und Kryptologie*. Springer, 2008.

[RS00] Jaikumar Radhakrishnan and Aravind Srinivasan. Improved bounds and algorithms for hypergraph 2-coloring. *Random Structures and Algorithms*, 16:4–32, 2000.

[Wil96] L.T. Wille. New binary covering codes obtained by simulated annealing. *IEEE TRANSACTIONS ON INFORMATION THEORY*, 42:300–302, 1996.

[Win02] Renate Winter. *Theoretische Informatik*. Oldenbourg Wissenschaftsverlag, 2002.

Abbildungsverzeichnis

Tabellenverzeichnis