Primfaktorzerlegung mit Simulated Annealing

Bachelorarbeit
zur Erlangung des akademischen Grades
Bachelor of Science

vorgelegt von
Fabian Köhler
geboren in Varel

Lehrstuhl für Theoretische Physik 1
Fakultät Physik
Technische Universität Dortmund
2014

Abstract
Prime factorization is an interesting problem as it is not efficiently solvable on a classical computer with any known algorithm. This fact is used in modern cryptograpy methods. In this thesis, a method based on simulated annealing will be examined. It could be shown that one factorization step can be done in polynomial time.

Zusammenfassung
Die Primfaktozerlegung ist ein interessantes Problem, weil es auf einem klassischen Computer mit keinem bekannten Algorithmus effizient lösbar ist. Diese Tatsache wird zum Beispiel in modernen Verschlüsselungsverfahren benutzt. In dieser Arbeit wird eine auf Simulated Annealing basierende Methode untersucht [1]. Dabei kann unter anderem bestätigt werden, dass damit ein Schritt der Zerlegung einer Zahl in polynomieller Laufzeit möglich ist.

Inhaltsverzeichnis

1 Einleitung

Die Zerlegung einer Zahl in ihre Primfaktoren lässt sich auf einem klassischen Computer mit keinem bisher bekanntem Algorithmus effizient, d.h. in polynomieller Laufzeit, berechnen. Das schnellste klassische Verfahren, das Zahlkörpersieb, hat eine exponentielle Laufzeit [5].
Die Probe hingegen, ob mehrere Zahlen die Primfaktoren einer Zahl sind, lässt sich effektiv durch einfache Multiplikation durchführen. Auf diesem Prinzip basiert eine Vielzahl aktueller Verschlüsselungsmethoden, z.B. das RSA-Kryptosystem [7].
Jenseits der klassischen Informatik gibt es jedoch schon Verfahren, die eine effektive Behandlung des Problemes verheißen. So schlug P. W. Shor 1994 einen Algorithmus vor [8], der auf einem Quantencomputer in polynomieller Laufzeit Primfaktoren einer Zahl berechnet. Die Implementierung eines entsprechenden Quantencomputers ist schwierig und bisher nur für relativ kleine Systeme realisiert worden. Die prinzipielle Funktion dieses Algorithmus ist für die Zahl 15 experimentell bestätigt worden [10].
Ein weiterer Ansatz nutzt Adiabatic Quantum Computing [4] in einem NMR-System. Mit diesem Verfahren konnte die Zahl 143 faktorisiert werden [11].
Hier soll nun ein von E. L. Altschuler und T. J. Williams entwickeltes Verfahren [1] beschrieben und untersucht werden, das mit Simulated Annealing eine aus der statistischen Physik motivierte und in vielen Bereichen genutzte Optimierungsmethode verwendet. Dieses Verfahren ist nicht deterministisch und führt nicht immer zum gesuchten Ergebnis, was allerdings auf den Shor-Algorithmus ebenfalls zutrifft.
Zunächst wird in Kapitel 2 ein knapper Überblick über die Methode des Simulated Annealings gegeben und anschließend die Anwendung dieser Methode auf das Problem der Primfaktorzerlegung beschrieben. Des Weiteren wird die Laufzeit des Algortihmus abgeschätzt und mit der des Zahlkörpersiebs verglichen.
In Kapitel 3 wird die Implementierung der Programme erläutert. Außerdem werden die durchgeführten Untersuchungen beschrieben und ausgewertet.
In Kapitel 4 werden die Resultate noch einmal zusammengefasst und weitere Untersuchungen vorgestellt, die zu weiteren Erkenntnissen führen könnten.

2 Beschreibung des Verfahrens

2.1 Primfaktorzerlegung

Das hier beschriebene Verfahren zur Faktorisierung basiert größtenteils auf der Arbeit von Altschuler und Williams [1].
Die Primfaktorzerlegung bezeichnet die eindeutige Darstellung einer Zahl $N \in \mathbb{N}$ durch

$$N = \prod_{i=1}^{M} p_i^{m_i},$$

wobei die $p_i \in \mathbb{N}$ mit $p_i > 1$ und $p_i \neq p_j$ für $i \neq j$ Primzahlen und die $m_i \in \mathbb{N}$ die zugehörigen Exponenten sind.
Diese Zerlegung kann rekursiv aufgebaut werden. Zunächst wird die Zahl N in zwei Faktoren A und B zerlegt. Danach werden A und B zerlegt sofern sie nicht prim sind.
Es soll ein Verfahren entwickelt werden, welches die Zerlegung $N = A \cdot B$ mit $A \geq B$ berechnet und dabei die Methode des Simulated Annealing anwendet.
Im folgenden sollen zunächst einige Eigenschaften der binären Repräsentation der Zahlen A und B abgeleitet werden.

2.2 Abschätzung der Wertebereiche von a, a_1, b, b_1

Die Zahlen N, A, B werden im Dualsystem dargestellt. a, b, n sollen dabei angeben, wie viele Stellen dafür unter Vernachlässigung führender Nullen erforderlich sind. a_1 bzw. b_1 geben dabei an, wie häufig die Ziffer 1 in A bzw. B vorkommt.
Der Algorithmus durchläuft alle möglichen Tupel (a, a_1, b, b_1) (vgl. Abschnitt 2.4). Deshalb sollte geprüft werden, inwieweit der Suchbereich bei gegebenem N eingeschränkt werden kann.
Die binäre Darstellung von A bzw. B kann maximal so lang sein wie die von N, d.h. $a \leq n \wedge b \leq n$. Wegen $p_i > 1$ ist $a > 1 \wedge b > 1$. Es gibt allerdings noch weiteres Potential, die Wertebereiche für a und b zu reduzieren.
Durch a, b, n sind die Wertebereiche der Zahlen A, B, N auf

$$\begin{aligned} 2^{a-1} \leq & A \leq 2^a - 1 \\ 2^{b-1} \leq & B \leq 2^b - 1 \\ 2^{n-1} \leq & N \leq 2^n - 1 \end{aligned}$$

eingeschränkt. Es ist $A \cdot B = N$, sodass bei maximalem N

$$(2^a - 1)(2^b - 1) = 2^{a+b} - 2^a - 2^b + 1 < 2^{a+b}$$

gilt und $a+b$ Bits zur Darstellung von N genügen. Für den Fall eines minimalen N gilt analog

$$\begin{aligned} 2^{a-1} \cdot 2^{b-1} = 2^{a+b-2} &\stackrel{!}{=} 2^{n-1} \\ \Rightarrow a+b &= n+1 \end{aligned}$$

Bei der Multiplikation $A \cdot B = N$ gilt also $a+b = n \ \vee \ a+b = n+1$. Mit Hilfe dieser Bedingungen kann der Parameterbereich deutlich genauer eingegrenzt werden.
Zunächst kann eine untere Schranke für a gewonnen werden, es gilt:

$$a = n-b \ \vee \ a = n-b+1$$

Diese Ausdrücke werden minimal für $a = b$, also das maximal zulässige $b \leq a$, sodass der minimale Wert für a durch

$$a_{\min} = \begin{cases} 2 & \text{falls } \lfloor \frac{n}{2} \rfloor = 1 \\ \lfloor \frac{n}{2} \rfloor & \text{sonst} \end{cases} \tag{2.1}$$

gegeben ist. Dies legt zu gegebenem a die untere Schranke

$$b_{\min} = \begin{cases} 2 & \text{falls } n-a = 1 \\ n-a & \text{sonst} \end{cases} \tag{2.2}$$

für b fest.
Die binären Darstellungen von A bzw. B müssen mindestens eine 1 enthalten, sonst wäre die Zahl 0. Maximal können alle Stellen 1 sein, sodass die Wertebereiche für a_1 bzw. b_1 durch die Ungleichungen

$$\begin{aligned} 1 \leq a_1 \leq a \\ 1 \leq b_1 \leq b \end{aligned}$$

gegeben sind.
Mit diesen Überlegungen wird der Parameterraum von

$$\begin{aligned} 2 \leq a \leq n \qquad & 1 \leq a_1 \leq a \\ 2 \leq b \leq a \qquad & 1 \leq b_1 \leq b \end{aligned} \tag{2.3}$$

auf

$$\begin{aligned} a_{\min} \leq a \leq n \qquad & 1 \leq a_1 \leq a \\ b_{\min} \leq b \leq a \qquad & 1 \leq b_1 \leq b \end{aligned} \tag{2.4}$$

reduziert. Die Auswirkungen auf die Laufzeit werden in Abschnitt 2.5 betrachtet.
Im nächsten Abschnitt wird zunächst ein Überblick über die Methode des Simulated Annealing gegeben.

2.3 Simulated Annealing

Simulated Annealing ist eine Methode, um Optimierungsprobleme zu lösen, die sehr komplex sind [6]. Dies ist der Fall wenn für die Parameter ein großer Wertebereich zulässig ist, die Dimensionalität des Problems, d.h. die Anzahl der Parameter, hoch ist oder es neben dem gesuchten globalen Extremum viele lokale Extrema gibt.
Zu einer gegebenem Temperatur wird eine adäquate Anfangskonfiguration gewählt, z.B. bei einer hohen Temperatur eine zufällige Konfiguration. Dann werden Annealing Schritte ausgeführt, d.h. die Temperatur sukzessive gesenkt. Nach jeder Temperatursenkung wird eine ausreichende Anzahl an Metropolisschritten durchgeführt. Dazu wird jeweils eine leichte Modifikation am aktuellen Systemzustand vorgenommen. Verringert die neue Konfiguration die Energie, respektive eine analog dazu definierte Kostenfunktion, so wird die Änderung akzeptiert, ansonsten nur mit einer Wahrscheinlichkeit von $p = \exp\left(-\frac{\Delta E}{k_\mathrm{B} T}\right)$. Auf diese Art und Weise wird versucht, das System zu äquilibrieren.

2.4 Anwendung von Simulated Annealing auf die Primfaktorzerlegung

Damit Simulated Annealing auf das das Problem der Primfaktorzerlegung angewendet werden kann, wird zunächst eine Energiedefinition eingeführt. Hier wird

$$E\left(A, B, N\right) = \sum_{i=1}^{n} \begin{cases} f\left(i\right) & \text{falls } \{A \cdot B\}_i = \{N\}_i \\ 0 & \text{sonst} \end{cases}$$

gewählt. Dabei bezeichne die Schreibweise $\{Z\}_i$ die i-te Stelle der binären Repräsentation einer Zahl Z, wobei $i = 0$ das niederwertigste Bit ist. $f\left(i\right)$ ist eine monoton steigende Funktion, z.B. $f\left(i\right) = i$ oder $f\left(i\right) = i^2$. Die Energie wächst mit der Übereinstimmung des Produktes $A \cdot B$ mit N, wobei die höherwertigen Bits ein größeres Gewicht haben. Bei dieser Energiedefinition gilt es also ein Maximum zu finden und kein Minimum.
Für das Simulated Annealing wird für jedes 4-Tupel (a, b, a_1, b_1) mit einer Temperatur $T_0 = 1$ begonnen, um dann N_a Annealing-Schritte durchzuführen. Dabei wird das System jeweils um einen Faktor $F_c < 1$ abgekühlt, sodass $T_{i+1} = F_c \cdot T_i$. Vor jedem Abkühlen werden N_c Konfigurationen getestet. Dies geschieht mit Hilfe des Metropolis-Algorithmus [2].
Zunächst sollen einige notwendige, grundlegende Operationen zur Modifikation von binären Zahlen eingeführt werden. Diese lassen die Länge der Binärzahl und die Anzahl der auf 1 bzw. 0 gesetzten Bits invariant und werden verwendet, um Variationen einer Konfiguration zu generieren.

- **swap:** Es werden zwei Bits zufällig vertauscht, wobei darauf zu achten ist, dass eines auf 1 gesetzt ist und das andere auf 0.
- **slide:** Es wird eine durchgängige Sequenz an Bits zufällig ausgewählt und das Bit ganz rechts entfernt. Danach werden alle anderen Bits nach rechts durchgeschoben und das entfernte Bit links wieder eingefügt.

- **reverse:** Es wird eine zufällige Sequenz an Bits ausgewählt und ihre Reihenfolge invertiert.
- **random:** Es werden Bits zufällig ausgewählt und zufällig permutiert.

Algorithmus 1 Metropolis-Schritt

```
1:  procedure METROPOLIS(A, B, E, N)
2:      if randomInt (0, 1) = 0 then
3:          A' ← randomOperation (A)
4:      else
5:          B' ← randomOperation (B)
6:      end if
7:      if A · B = N then
8:          Exit                                  ▷ Faktoren A, B wurden gefunden
9:      end if
10:     E' = E (A', B', N)
11:     if E' > E then
12:         A ← A'
13:         B ← B'
14:         E ← E'
15:     else
16:         p ← randomFloat (0.0, 1.0)
17:         if p < exp(−(E'−E)/(k_B T)) then
18:             A ← A'
19:             B ← B'
20:             E ← E'
21:         end if
22:     end if
23: end procedure
```

Der Ablauf eines Metropolisschrittes ist in Algorithmus 1 dargestellt. Es wird mit der Funktion *randomOperation*() zufällig eine der Operationen ausgewählt und durchgeführt. Danach wird die Energie E' der daraus resultierenden neuen Konfiguration berechnet. Die Akzeptanzwahrscheinlichkeit für die neue Konfiguration ist

$$p = \begin{cases} 1 & \text{falls } E' > E \\ \exp\left(\frac{E'-E}{k_\mathrm{B} T_i}\right) & \text{sonst} \end{cases} \tag{2.5}$$

Eine neue Konfiguration wird also auf jeden Fall akzeptiert, wenn sie die Energie vergrößert, ansonsten mit einer über einen Boltzmann-Faktor gegebene Wahrscheinlichkeit. Dabei ist k_B ein Analogon zur physikalischen Boltzmann-Konstanten. Hier wird sie je nach Problem unterschiedlich gewählt, um das Wachsen der Energieskala mit zunehmendem n auszugleichen, damit eine geeignete Akzeptanzwahrscheinlichkeit ungünstigerer

Konfigurationen gewährleistet ist. Dies wird im Abschnitt 3.2 genauer untersucht.

Algorithmus 2 Ablauf des Annealing

```
1: procedure ANNEAL(A, B, E, N)
2:     T ← 1.0
3:     for i ← 1 to N_a do
4:         for j ← 1 to N_c do
5:             Metropolis (A, B, E, N) 1
6:         end for
7:         T ← T · F_c
8:     end for
9: end procedure
```

Der Metropolis-Algorithmus 1 wird im Rahmen des in Algorithmus 2 dargestellten Annealing-Verfahrens aufgerufen. Es wird mit einer Temperatur von $T_0 = 1$ begonnen. Dann werden N_a Annealing-Schritte durchgeführt und dabei jeweils N_c Metropolis-Schritte durchgeführt.
Diese Prozedur wird für jedes erlaubte 4-Tupel (a, a_1, b, b_1) durchgeführt. Diesem Zweck dienen die 4 Schleifen in Algorithmus 3.
Für jedes Tupel (a, a_1) wird eine zufällige binäre Zahl A mit a Stellen gezogen, von denen a_1 auf 1 gesetzt sind. Analog wird für jedes Tupel (b, b_1) eine entsprechende Zahl B erstellt. Anschließend wird die anfängliche Energie berechnet. Außerdem wird das Produkt $A \cdot B$ berechnet, sodass das Programm abgebrochen werden kann, falls zufällig direkt eine Zerlegung der Zahl N in zwei Faktoren gefunden wird. Ansonsten wird das in Algorithmus 2 beschriebene Annealing durchgeführt.

Algorithmus 3 Zerlegung in zwei Faktoren

```
1: procedure FACTORIZE(N)
2:     if ⌊n/2⌋ then
3:         a_min ← 2
4:     else
5:         a_min ← ⌊n/2⌋
6:     end if
7:     for a ∈ [a_min, n] do
8:         for a_1 ∈ [1, a] do
9:             A ← randomBitset(a, a_1)
10:            if n − a = 1 then
11:                b_min ← 2
12:            else
13:                b_min ← n − a
14:            end if
15:            for b ∈ [b_min, a] do
16:                for b_1 ∈ [1, b] do
17:                    B ← randomBitset(b, b_1)
18:                    E ← E(A, B, E, N)
19:                    if A · B = N then
20:                        Exit                ▷ Faktoren A, B wurden gefunden
21:                    end if
22:                    anneal(A, B, E, N) 2
23:                end for
24:            end for
25:        end for
26:    end for
27: end procedure
```

2.5 Abschätzung der Laufzeit

Bei der Untersuchung des Algorithmus ist die Laufzeit in Abhängigkeit von der Länge n der zu faktorisierenden Zahl interessant. Hierzu kann die Anzahl der im ungünstigsten Fall benötigten Metropolis-Schritte betrachtet werden.
Schränkt man den Suchbereich nicht wie in Abschnitt 2.2 beschrieben ein, d.h. wählt man den Parameterbereich gemäß der Gleichungen (2.3), werden maximal

$$R_1(n) = \sum_{a=2}^{n} \sum_{a_1=1}^{a} \sum_{b=2}^{a} \sum_{b_1=1}^{b} N_a \cdot N_c \tag{2.6}$$

Metropolis-Schritte durchgeführt. Mit dem kleineren Suchbereich gemäß Gleichungen (2.4) ergibt sich eine Laufzeit von

$$R_2(n) = \sum_{a=a_{\min}}^{n} \sum_{a_1=1}^{a} \sum_{b=b_{\min}}^{a} \sum_{b_1=1}^{b} N_a \cdot N_c. \tag{2.7}$$

In beiden Fällen ist grob eine Laufzeit von $\mathcal{O}\left(n^4 \cdot N_a \cdot N_c\right)$ zu erwarten, weil die Anzahl der Summanden in jeder Summe linear mit n wächst.

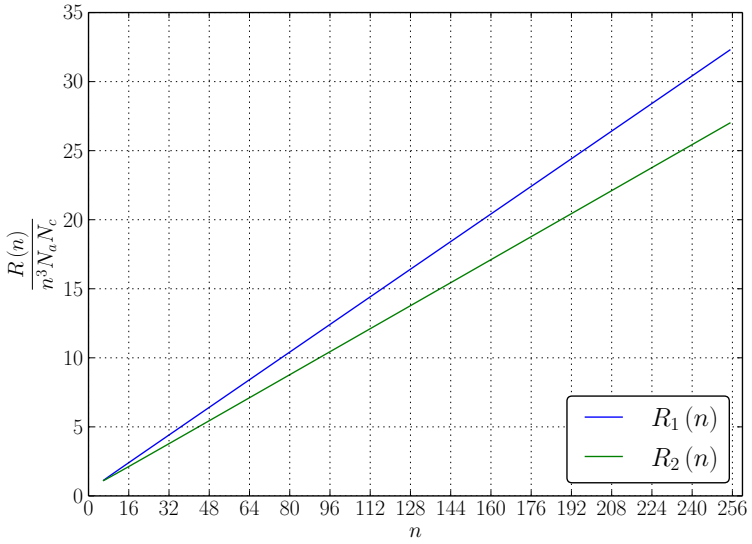

Abbildung 2.1: Theoretische Laufzeiten nach Gln. (2.6) und (2.7)

Die Gleichungen (2.6) und (2.7) wurden numerisch ausgewertet und in Abb. 2.1 graphisch dargestellt.
Außerdem wurde die Abweichung

$$1 - \frac{R_2(n)}{R_1(n)}$$

berechnet und in Abb. 2.2 aufgetragen, um die Verbesserung der Laufzeit durch den kleineren Parameterbereich darzustellen. Dabei ist festzustellen, dass durch die Einschränkungen im ungünstigsten Fall 16 % weniger Metropolisschritte erforderlich sind.
Nun vergleichen wir dieses Verhalten mit dem Zahlkörpersieb, welches eine Laufzeit von

$$r_{\text{sieve}} \approx \exp\left(c \cdot (\log N)^{\frac{2}{3}} (\log\log N)^{\frac{1}{3}}\right) \tag{2.8}$$

mit $c = 64/9$ (allgemeines Zahlkörpersieb) hat [5]. Diese wurde im Vergleich zu der maximalen Laufzeit der Simulated Annealing Methode in 2.3 aufgetragen. Dabei ist zu berücksichtigen, dass das Zahlkörpersieb ein deterministischer Algorithmus ist, also immer das korrekte Ergebnis liefert. Das Simulated Annealing ist spätestens bei der maximalen Laufzeit beendet, muss aber nicht zwangsläufig ein Ergebnis geliefert haben.

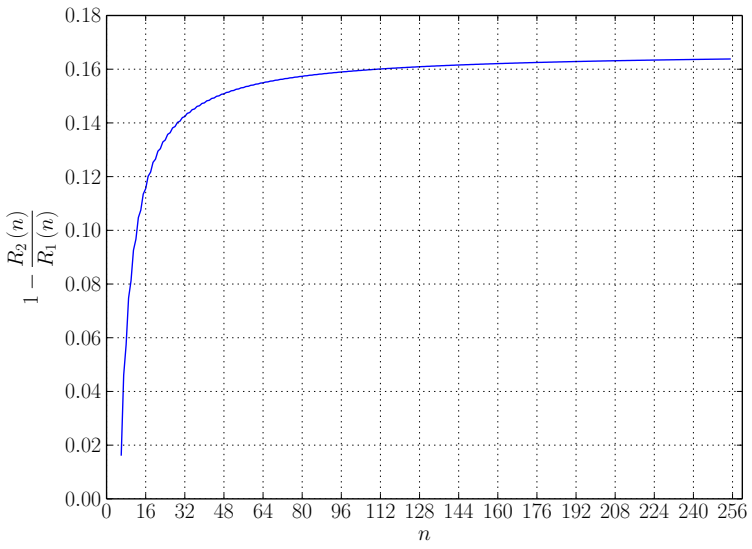

Abbildung 2.2: Verbesserung der Laufzeit durch Einschränkung des Parameterbereiches

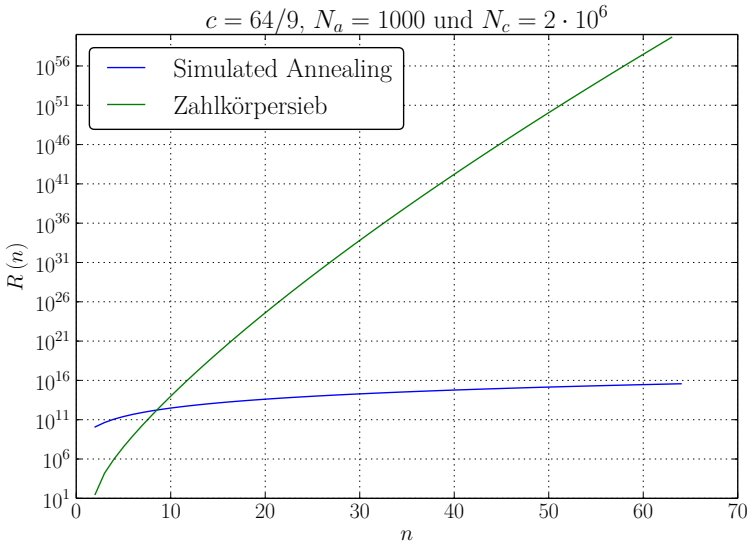

Abbildung 2.3: Laufzeitverhalten des allgemeinen Zahlkörpersiebs und der Simulated Annealing Methode

3 Untersuchung des Verfahrens

3.1 Implementierung

Für die Untersuchungen wurden 3 Programme (*onestep*, *semiprime* und *factorize*) implementiert.
Das Programm *onestep* führt einen Schritt der Faktorisierung einer Zahl durch, bildet also Algorithmus 3 ab.
Mit dem Programm *factorize* kann die komplette Faktorisierung einer Zahl berechnet werden. Es führt also den Algorithmus 3 an der Zahl N aus. Anschließend wird durch Vergleich mit einer Liste vom Primzahlen überprüft, ob die erhaltenen Faktoren prim sind. Ist dies nicht der Fall wird der Algorithmus auf die entsprechenden Faktoren angewendet, bis die komplette Zerlegung ermittelt wurde. Dieses Programm wählt den Wert der Boltzmann-Konstanten immer automatisch (siehe Abschnitt 3.2), weil sie für die entsprechenden Zerlegungsschritte angepasst werden muss.
Das Programm *semiprime* bekommt zwei Primfaktoren A und B übergeben und berechnet das Produkt $N = A \cdot B$, welches folglich eine Semiprimzahl ist. Die a, a_1, b, b_1 sind durch die übergebenen Zahlen bereits bekannt. Dann werden N_a Annealing-Schritte durchgeführt mit jeweils N_c Metropolis-Schritten. Von allen Iterationen in Algorithmus 3 wird somit nur diejenige durchgeführt, die das gesuchte Ergebnis liefern könnte. Mit diesem Programm soll hauptsächlich der Einfluss der Wahl der Boltzmann-Konstanten auf das Verfahren untersucht werden.
Bei der Ausführung der einzelnen Programme werden jeweils die Laufzeit und die Erfolgsrate protokolliert.
Die angewendete Methode ist nicht-deterministisch und führt somit nicht immer auf die gesuchten Primfaktoren. Da die Probe bei allen 3 Programmen effizient durchführbar ist, wird bei jedem Durchlauf der Erfolg oder Misserfolg gespeichert. Daraus kann die Erfolgsrate durch Division der Anzahl der erfolgreichen Versuche durch die Anzahl der Aufrufe berechnet werden.
Weil bei allen Programmen nur die tatsächliche Laufzeit und nicht die Anzahl der Metropolisschritte protokolliert wird, ist eine etwas anderes Wachstum als in 2.5 beschrieben zu erwarten. Bei der Betrachtung der erlaubten Operationen stellt sich heraus, dass diese alle linear in n also in $\mathcal{O}(n)$ skalieren. Die Anzahl der Bits in den Sequenzen kann maximal n sein und für die swap-Operation wird im ungünstigsten Fall über alle Bits iteriert. Für einen einzelnen Zerlegungsschritt ist mit einem Verhalten $\mathcal{O}(n^5 \cdot N_c \cdot N_a)$ zu rechnen, wenn die reale Laufzeit betrachtet wird.
Für die Implementierung des Verfahrens wurde die Programmiersprache C++ verwendet. Alle Programme bieten die Möglichkeit zur Verwendung von Threads, also der parallelen Programmausführung, vgl. Abschnitt 3.3. Dabei kommt die Thread-Bibliothek

aus dem aktuellen C++11 Standard zum Einsatz.
Zur Auswertung und für die grafische Darstellung wurde die Programmiersprache Python mit den Paketen numpy, scipy und matplotlib verwendet.

3.2 Abschätzung von k_B

Bei allen Untersuchungen wurde hier $f(i) = i^2$ gewählt, sodass die zu maximierende Kostenfunktion

$$E(A, B, N) = \sum_{i=1}^{n} \begin{cases} i^2 & \text{falls } \{A \cdot B\}_i = \{N\}_i \\ 0 & \text{sonst} \end{cases}$$

ist. Der maximale Wert dieser Funktion ist die quadratische Pyramidalzahl [9]

$$E_{\max}(n) = \sum_{i=1}^{n} i^2 = \frac{1}{3}n^3 + \frac{1}{2}n^2 + \frac{1}{6}n,$$

sodass die Energie mit $\mathcal{O}(n^3)$ skaliert.
Es ist wichtig k_B geeignet zu wählen, weil dieser Parameter die Wahrscheinlichkeit bestimmt, mit der der Metropolisalgorithmus energetisch ungünstigere Konfigurationen akzeptiert. Kennt man für ein $\tilde{n}$ einen Wert $\tilde{k}_\mathrm{B}$, der zu einer hohen Erfolgsrate führt, so kann nach

$$k_\mathrm{B} = \frac{E_{\max}(n)}{E_{\max}(\tilde{n})}\tilde{k}_\mathrm{B}$$

eine Abschätzung für die zu wählende Boltzmann-Konstante in Abhängigkeit der Zahlenlänge n getroffen werden.
Das Experimentieren mit verschiedenen Werten zeigte, dass bei $n = 33$ eine Boltzmann-Konstante $\tilde{k}_\mathrm{B} \approx 8^3 = 512$ geeignet ist.
Außerdem wurde das Programm *semiprime* verwendet, um gezielt nach einem optimalen Wert für k_B zu suchen. Dabei wurden zwei Primzahlen $A = 66889$ und $B = 104723$ gewählt, welche die Semiprimzahl $N = A \cdot B = 7004816747$ mit $n = 33$, ergeben und das Programm für verschiedene Werte $1^3 \leq k_\mathrm{B} \leq 16^3$ aufgerufen. Jeder Durchlauf wurde dabei 4800-mal wiederholt. Als Parameter wurden $N_a = 1000$, $N_c = 80000$ und $F_c = 0.997$ gewählt.
Es wurde noch eine zweite Reihe von Simulationen mit den selben Parametern durchgeführt, allerdings mit Werten $4^3 \leq k_\mathrm{B} \leq 28^3$.
Die ermittelte Erfolgsrate bzw. die Programmlaufzeiten sind in Abb. 3.1 bzw. 3.2 grafisch dargestellt. Die Erfolgsraten wurden jeweils durch Mittelung über alle Laufzeiten zu jedem k_B unabhängig davon, ob diese erfolgreich waren, errechnet.
Es ist deutlich erkennbar, dass ein zu niedriger Wert von k_B zu Erfolgsraten nahe 0 führt, also nur die wenigsten Programmaufrufe die Primfaktoren A, B finden. Für den Erfahrungswert $k_\mathrm{B} \approx 8^3$ ergibt sich bereits eine Erfolgsrate von $\approx 30\%$. Besser scheint es jedoch beispielsweise $k_\mathrm{B} = 17^3$ oder auch größer zu wählen, was hier einer Erfolgsrate von $> 45\,\%$ entspricht.

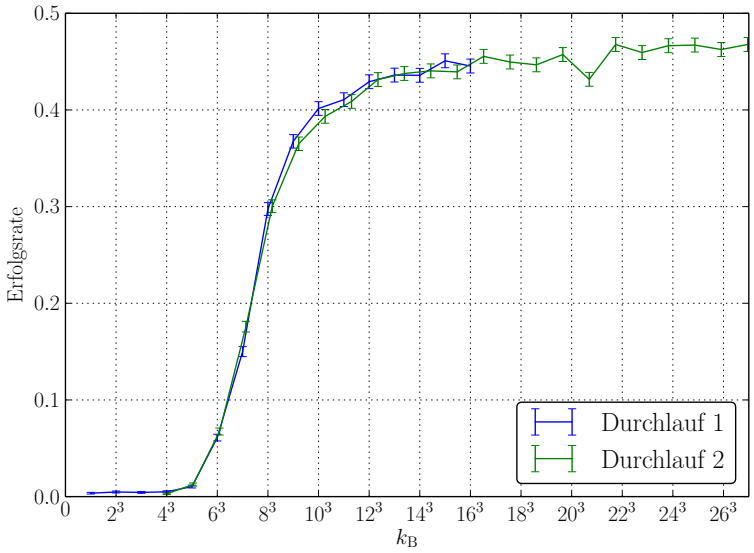

Abbildung 3.1: Erfolgsrate bei der Suche nach einem optimalen Wert für k_B

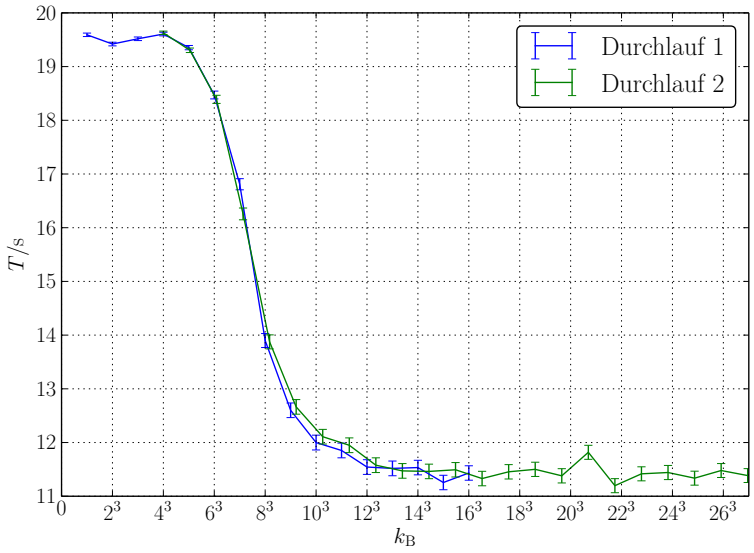

Abbildung 3.2: Laufzeiten bei der Suche nach einem optimalen Wert für k_B

Dieses Verhalten spiegelt sich auch in der Programmlaufzeit wieder. Zu Beginn ist diese nahezu konstant, da kaum Aufrufe die gesuchten Faktoren finden und dadurch alle $N_c \cdot N_a$ Schritte durchlaufen werden. Mit steigendem k_B sinkt die Laufzeit des Programmes. Dies spiegelt die zunehmende Erfolgsrate wieder. Wenn mehr Durchläufe erfolgreich sind, führen weniger Programmaufrufe dazu, dass der gesamte Parameterbereich durchlaufen werden muss. Dies passiert bei den nicht erfolgreichen Durchläufen.
Betrachtet man die Akzeptanzwahrscheinlichkeit, so kann man dieses Verhalten erklä-

ren. Ein zu kleines k_B führt dazu, dass so gut wie keine ungünstigen Konfigurationen akzeptiert werden. Dies widerspricht der grundsätzlichen Idee hinter dem Metropolisalgorithmus, da auch schlechtere Konfigurationen teilweise akzeptiert werden müssen, damit das Verfahren in kein lokales Extremum läuft. Es ist auch zu erwarten, dass k_B nicht zu groß gewählt werden sollte, weil dies dazu führen würde, dass die Akzeptanzwahrscheinlichkeit für schlechtere Zahlen A, B sehr hoch ist. Der Algorithmus würde sich hier nicht der gesuchten Lösung nähern, sondern die Zahlen A, B beliebig ändern.
Es wäre also günstiger gewesen in der Formel (3.2) beispielsweise $\tilde{k}_\mathrm{B} = 17^3 = 4913$ für $\tilde{n} = 33$ zu wählen. In den nachfolgenden Untersuchungen konnte dies jedoch nicht berücksichtigt werden, da diese Erkenntnis erst sehr spät gemacht wurde. In allen weiteren Simulationen wurde $\tilde{k}_\mathrm{B} = 8^3 = 512$ gesetzt, um automatisch Werte der Boltzmann-Konstanten in Abhängigkeit der Länge n der zu faktorisierenden Zahl zu ermitteln.

3.3 Parallelisierbarkeit

Betrachtet man Algorithmus 3, so erkennt man, dass dieses Verfahren zur Primfaktorzerlegung Potential zur Parallelisierung bietet. So kann das Annealing für jedes Tupel (a, a_1, b, b_1) unabhängig ausgeführt werden. Man generiert für jedes Tupel neue Anfangskonfigurationen A und B und führt dann jeweils das Annealing aus. Dies kann dann auf verschiedenen Prozessorkernen oder Rechnern erfolgen. Dadurch ist eine enorme Leistungssteigerung zu erwarten.
Dieses Vorgehen unterscheidet sich jedoch leicht von Algorithmus 3. Dort wird für jedes Tupel (a, a_1) eine Anfangskonfiguration A generiert und dann über alle Werte (b, b_1) iteriert. Dabei werden dann Anfangskonfigurationen B erzeugt. Da A jedoch zufällig gewählt wird und innerhalb der Schleifen zufällig modifiziert wird, kann für jedes Tupel (a, a_1, b, b_1) eine neues A zufällig generiert werden.
Die Implementierung der Programme erlaubt es, eine beliebige Anzahl von Threads festzulegen, auf die die Tupel (a, a_1, b, b_1) gleichmäßig verteilt werden. So wurden bereits die Simulationen in Abschnitt 3.2 mit 8 Threads ausgeführt, genauso wie alle späteren Simulationen mit der Ausnahme der in diesem Abschnitt zu findenden.
Es soll durch Aufruf des Programms *onestep* mit verschiedenen Anzahlen von Threads untersucht werden, wie effizient die Parallelisierung gelungen ist. Theoretisch halbiert sich die Laufzeit, wenn man die Anzahl der Threads verdoppelt. Dieses Vorgehen führt jedoch zu einem gewissen Verwaltungsaufwand beim Erstellen der Threads und dem Verteilen der Arbeit, sodass die Parallelisierungseffizienz nicht bei 100 % liegen wird. Dafür müsste sich die Programmlaufzeit halbieren, wenn man die Anzahl der Threads verdoppelt.
Um die Parallelisierbarkeit des Verfahrens zu untersuchen wird das Programm *onestep* verwendet. Die zu zerlegende Semiprimzahl ist $N = 783061$ ($n = 20$). Die weiteren Parameter betrugen $N_a = 500$, $N_c = 200$ und $F_c = 0.997$. Die Simulation wurde 200-mal wiederholt und zwar je für $1, 2, \ldots, 8$-Threads. Im Anschluss wurde der Speedup τ_1/τ_i berechnet, wobei τ_1 die Laufzeit mit einem Thread und τ_i mit i-Threads ist. Diese Größe wurde in Abb. 3.3 grafisch gegen die Anzahl der Threads aufgetragen.

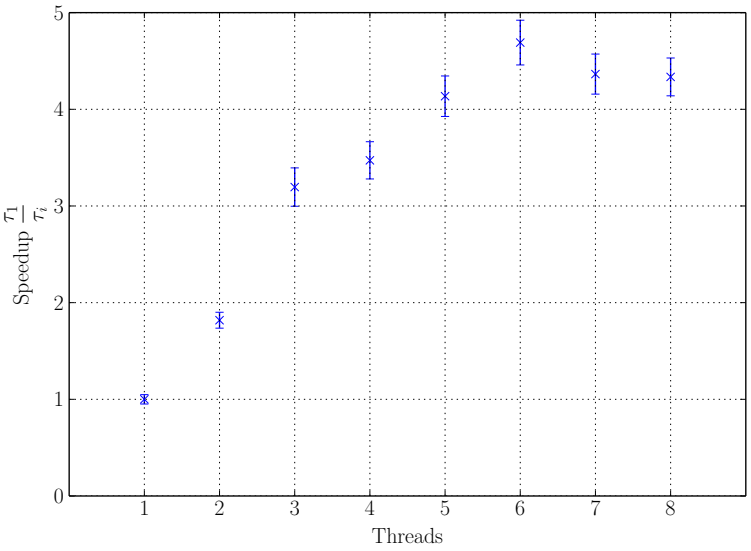

Abbildung 3.3: Beschleunigung des Programmes durch die Verwendung mehrerer Threads

Es ist erkennbar, dass für 2 und 4 Threads fast der maximal mögliche Speedup (Faktor 2 bzw. 4 erreicht wird), während 8 Threads nur ein Speedup von unter 5 erreicht wird. Im Idealfall würden alle Punkte auf einer Ursprungsgeraden mit der Steigung 1 liegen. Für 1 bis 6 Threads scheint die Parallelisierung gut zu funktionieren. Dass der Speed für 7 bis 8 geringer ist, könnte an statistischen Fluktuationen liegen oder daran, dass der Verwaltungsaufwand für mehr als 6 Threads bei $n = 20$ zu hoch ist. Diese Threadanzahlen rentieren sich erst bei längeren Zahlen.

3.4 Analyse eines einzelnen Zerlegungsschrittes

Mit dem Programm *onestep* wurden einzelne Zerlegungsschritte untersucht und dafür zunächst zwei Sätze von Zahlen gewählt. Zum einen wurden manuell für $n \in \{6, 8, \ldots, 24\}$ je 10 Semiprimzahlen konstruiert, sowie die drei Möglichen für $n = 4$. Der andere Satz von Zahlen besteht aus beliebigen Zahlen, wobei für jedes $n \in \{8, 9, \ldots, 32\}$ jeweils 15 solcher Zahlen zufällig ausgesucht wurden. Die Boltzmann-Konstante wurde dabei wie in Abschnitt 3.2 beschrieben automatisch gewählt.
Die Parameter waren $N_a = 500$, $N_c = 1000$ und $F_c = 0.997$. Bei den Semiprimzahlen wurde jede Zerlegung 20-mal durchgeführt, bei den beliebigen Zahlen auf Grund des größeren Datensatzes nur 10-mal.
Für jedes n wurde die Erfolgsrate bestimmt. Dabei wurden die Anzahl der erfolgreichen Zerlegungen der Zahlen der Länge n aufsummiert und durch die Gesamtzahl aller Zerlegungen dividiert. Die Laufzeit wurde auf zwei Arten bestimmt. Zum einen wurden Analog zur Erfolgsrate einfach alle Laufzeiten zu jedem n gemittelt, zum anderen wurde nur über die erfolgreichen Durchläufe gemittelt. Der Grund dafür ist, dass beide Mitte-

lungen interessant sein können. Die eine gibt an, welche Zeit das Programm im Mittel benötigt, bis es unabhängig vom Ergebnis beendet wird, die andere, wie lange die erfolgreichen Zerlegungen im Mittel brauchen.

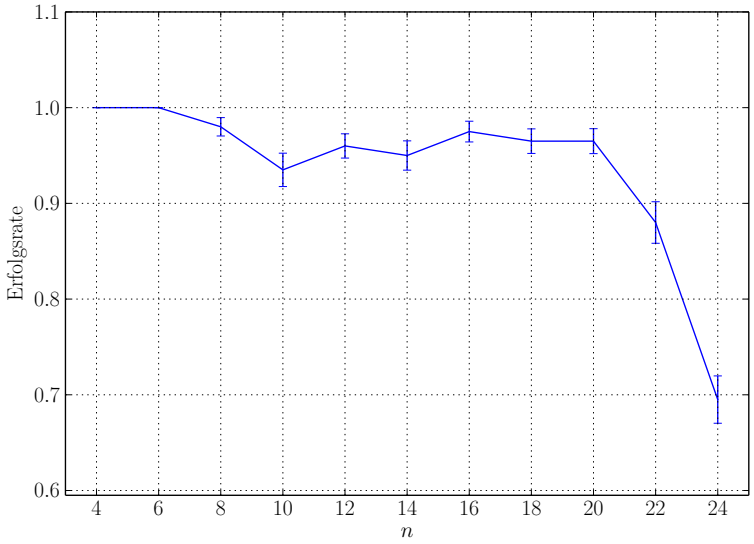

Abbildung 3.4: Erfolgsrate bei der Zerlegung von Semiprimzahlen

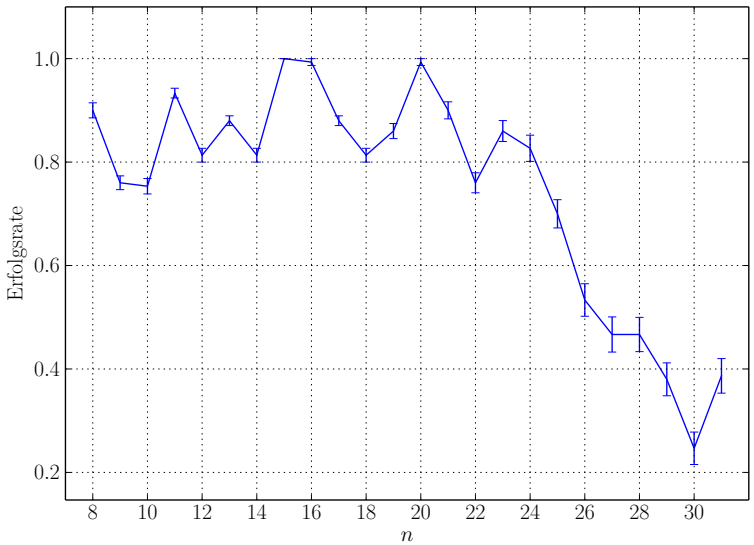

Abbildung 3.5: Erfolgsrate bei der Zerlegung von beliebigen Zahlen

Die dabei ermittelten Erfolgsraten wurden in Abb. 3.4 und Abb. 3.5 dargestellt.
In beiden Fällen erkennt man, dass die Erfolgsrate mit zunehmendem n merklich abnimmt. Dies liegt daran, dass N_a und N_c für alle N gleich belassen wurden. Der Be-

reich in dem (a, a_1, b, b_1) liegen können wächst jedoch mit zunehmendem n, genauso wie die oberen Grenzen für a und b. Für große a und b gibt es viel mehr mögliche Zahlen A, B mit gegebenen a_1 und b_1. Dies bedeutet, dass bei großen n ein geringerer Anteil der möglichen Zahlen A und B durchlaufen wird als bei kleinen n, somit sinkt auch die Wahrscheinlichkeit, die Faktoren zu finden.

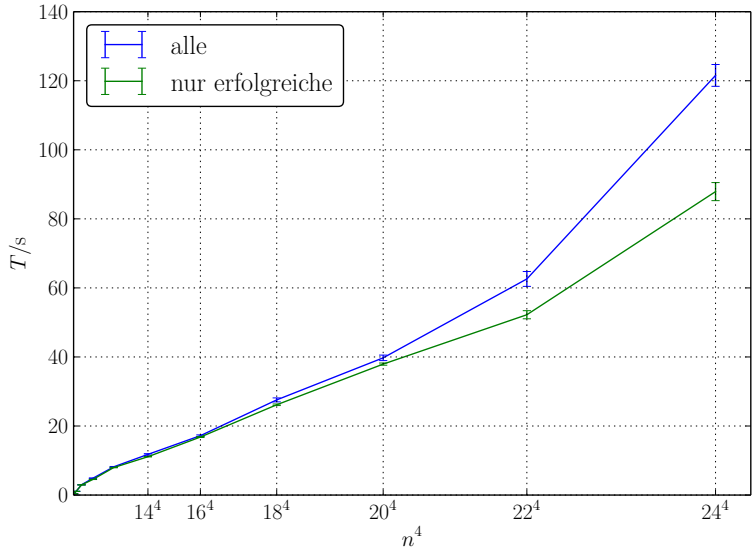

Abbildung 3.6: Laufzeiten bei der Zerlegung von Semiprimzahlen

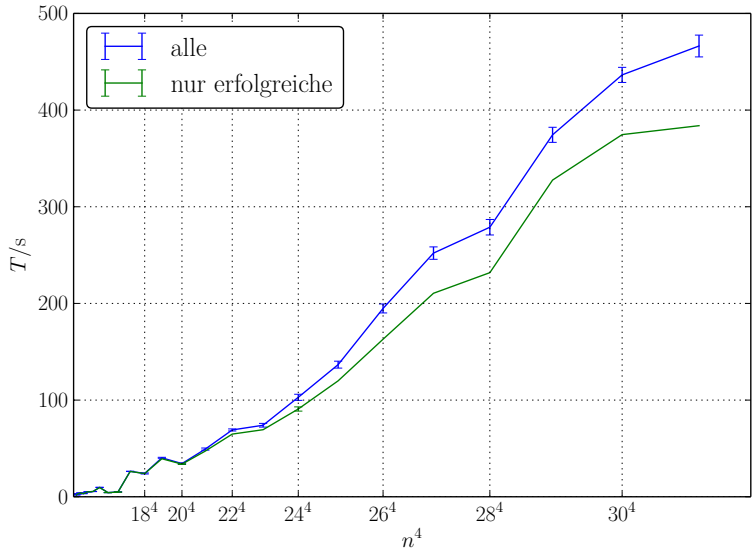

Abbildung 3.7: Laufzeiten bei der Zerlegung von beliebigen Zahlen

Bei der Simulation wurden die gleichen Werte N_a und N_c nicht geändert und stellen somit konstante Faktoren dar. Die Laufzeit wächst nach einem $\mathcal{O}\left(n^5\right)$-Gesetz.

3.5 Laufzeitverhalten bei kompletter Zerlegung

Mit dem Programm *factorize* wurde für $n = 8, 9, \ldots, 15$ die Zerlegung von Zahlen in alle ihre Primfaktoren untersucht. Dazu wurden für jedes n zufällig 5 Zahlen gezogen und jede davon 10-mal zerlegt. Als Parameter wurden $N_a = 1000$, $N_c = 1000$ und $F_c = 0.997$ gewählt.
Die Werte für N_a und N_c wurden scheinbar recht hoch angesetzt, da alle Durchläufe erfolgreich waren. Die ermittelte Lauzeit ist in Abb. 3.8. Auch hier ist grob ein Wachstum mit $\mathcal{O}\left(n^5\right)$ erkennbar, dies würde darauf hindeuten, dass der erste Zerlegungsschritt den dominanten Beitrag zur Laufzeit liefert. Dies kann jedoch in künftigen Untersuchungen genauer überprüft werden. Es ist ein komplizierterer Zusammenhang zu erwarten, da die vollständige Zerlegung noch von der Zahl der Primfaktoren so wie deren Längen abhängt.

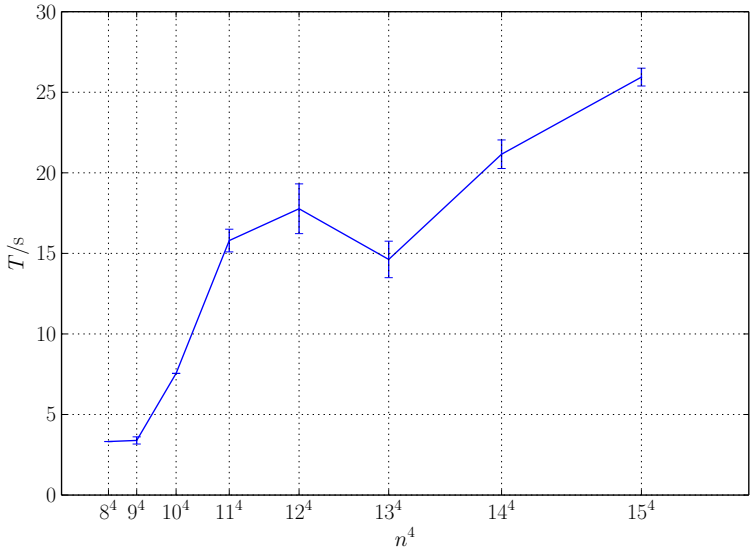

Abbildung 3.8: Laufzeitverhalten bei kompletter Zerlegung in Abhängigkeit der Zahlenlänge

4 Fazit und Ausblick

Im Rahmen dieser Arbeit konnte gezeigt werden, dass die von Altschuler und Williams vorgeschlagene Methode [1] grundsätzlich geeignet ist, Zahlen in ihre Primfaktoren zu zerlegen.
Es konnte das Laufzeitverhalten $\mathcal{O}\left(n^5 \cdot N_a \cdot N_c\right)$ eines einzelnen Zerlegungsschrittes mit Hilfe von Simulationen bestätigt werden. Dies deutet darauf hin, dass die Anzahl der Metropolisschritte tatsächlich mit $\mathcal{O}\left(n^4 \cdot N_a \cdot N_c\right)$ skaliert. Um dies zu zeigen, könnten die benötigten Schritte gezählt werden. Außerdem konnte der Einfluss der Boltzmann-Konstanten und damit der Energieskala auf die Erfolgsrate und die Laufzeit untersucht werden. Mit den Erkenntnissen aus der Simulation ist es möglich, eine automatische Abschätzung von k_B in Abhängigkeit der Zahlenlänge vorzunehmen.
Das Verfahren bietet zudem Möglichkeiten für weitere Untersuchungen. So könnte untersucht werden, wie sich Erfolgsrate und Laufzeit entwickeln, wenn zu kleine Zahlen N_a und N_c gewählt werden und somit zunächst nur wenige Programmdurchläufe eine erfolgreiche Zerlegung liefern.
Außerdem wurde ausschließlich die quadratische Energiedefinition getestet. Der genaue Einfluss der Energiefunktion auf die Laufzeit und die Erfolgsrate wurde nicht untersucht. Dabei könnte versucht werden, Funktionen zu ermitteln, die eine schnellere Konvergenz der Faktoren A und B gegen die Faktoren von N bewirken. Zusätzlich bietet es sich an zu studieren, welcher Anteil des Parameterraumes im Durchschnitt überhaupt durchlaufen wird, bis ein erfolgreiches Ergebnis vorliegt. Dabei könnte auch ermittelt werden, wie viel schneller die erfolgreichen Durchläufe terminiert sind.
Bei der Implementierung des Programmes *factorize* könnte man noch erweiterte Methoden zum Test, ob die erhalten Faktoren prim sind, verwenden. So existiert zum Beispiel der Test nach Miller und Rabin, der zwar eigentlich probabilistisch ist, aber bei geschickter Implementierung deterministisch ist [3]. Außerdem wäre es noch interessant, inwiefern der erste Zerlegungsschritt dominant für die Laufzeit ist und wie lange die weiteren Zerlegungen brauchen.
Des Weiteren kann noch überprüft werden, welche Primzahlen man in der Praxis maximal zerlegen könnte. Dazu könnte man das Programm mit Techniken wie MPI (Message Passing Interface) auf mehrere Knoten eines Clusters verteilen, da sich der Code gut parallelisieren lässt.

Der Programm-Code der erstellten Programme, die erzeugten Daten und die Arbeit selbst sind zu finden unter:
`https://github.com/f-koehler/bachelor-thesis`

Literatur

[1] E. L. Altschuler und T. J. Williams. *Using Simulated Annealing to Factor Numbers.* 17. Feb. 2014. arXiv: 1402.1201v2.

[2] N. Metropolis, A. W. Rosenbluth, M. N. Rosenbluth, A. H. Teller und E. Teller. „Equation of State Calculations by Fast Computing Machines". In: *Journal of Chemical Physics* (1953), S. 1087–1092. DOI: 10.1063/1.1699114.

[3] G. L. Miller. „Riemann's Hypothesis and tests for primality". In: *Proceedings of seventh annual ACM symposium on Theory of computing* (1975), S. 234–239. DOI: 10.1145/800116.803773.

[4] X. Peng, Z. Liao, N. Xu, G. Qin, X. Zhou, D. Suter und J. Du. „Quantum Adiabatic Algorithm for Factorization and Its Experimental Implementation". In: *Physical Review Letters* 101 (2008). DOI: 10.1103/PhysRevLett.101.220405.

[5] C. Pomerance. „A Tale of Two Sieves". In: *Notices of the AMS* 43 (1996), S. 1473–1485. URL: http://www.ams.org/notices/199612/pomerance.pdf (besucht am 13. 06. 2014).

[6] W. H. Press, S. A. Teukolsky, W. T. Vetterling und B. P. Flannery. „Numerical Recipes: The Art of Scientific Computing (3rd edition)". In: Cambridge University Press, 2007. Kap. 10.12, S. 549–555.

[7] R. L. Rivest, A. Shamir und L. Adleman. „A method for obtaining digital signatures and public-key cryptosystems". In: *Communications of the ACM* 21 (1978), S. 120–126. DOI: 10.1145/359340.359342.

[8] P. W. Shor. „Polynomial-Time Algorithms for Prime Factorization and Discrete Logarithms on a Quantum Computer". In: *SIAM Journal on Computing* 26 (1996), S. 1484–1509. DOI: 10.1137/S0097539795293172.

[9] N. J. A. Sloane. *Online Encyclopedia of Integer Sequences (OEIS): Square pyramidal numbers.* 11. März 2010. URL: http://oeis.org/A000330 (besucht am 13. 06. 2014).

[10] L. M. K. Vandersypen, M. Steffen, G. Breyta, C. S. Yannoni, M. H. Sherwood und I. L. Chuang. „Experimental realization of Shor's quantum factoring algorithm using nuclear magnetic resonance". In: *Nature* 414 (2001), S. 883–887. DOI: 10.1038/414883a.

[11] N. Xu, J. Zhu, D. Lu, X. Zhou, X. Peng und J. Du. „Quantum Factorization of 143 on a Dipolar-Coupling NMR system". In: *Physical Review Letters* 108 (2012). DOI: 10.1103/PhysRevLett.108.130501.

Danksagung

Ich danke Herrn Prof. Dr. Stolze für die freundliche Unterstützung und sehr gute Betreuung bei der Erstellung dieser Arbeit. Er hat sich auch außerhalb unserer wöchentlichen Treffen Zeit für mich genommen und mir aus so mancher Situation, in der ich mich verzettelt hatte, herausgeholfen. Des Weiteren hat er mir, mit freundlicher Unterstützung von Herrn Dr. Raas, Zugang zum Phido-Cluster der Fakultät Physik verschafft, ohne den die numerischen Teile der Arbeit nicht möglich gewesen wären.
Des Weiteren danke ich Frau Priv.-Doz. Dr. Löw dafür, dass sie sich die Zeit nimmt, diese Arbeit als Zweitkorrektorin zu begutachten.
Meiner Familie danke ich für die bisherige Unterstützung während meiner Schullaufbahn und meines Studiums. Außerdem möchte ich meiner Freundin Judith Storb für den moralischen Beistand danken.
Des Weiteren bedanke ich bei meinen Kommilitonen Sonja Bartkowski, Kay Schönwald, Nils Ziegeler und Felix Wieland, die sich freundlicherweise die Zeit genommen haben, einen ersten Blick auf die Arbeit zu werfen und Verbesserungsvorschläge zu liefern.