Textkompression: Burrows-Wheeler-Transformation

Proseminar „Algorithmen der Bioinformatik“

Uli Köhler

12. November 2012

Aufbau dieser Präsentation

- Kompression in der Bioinformatik
- Die Burrows-Wheeler-Transformation
  - Algorithmus der Hintransformation
  - Algorithmus der Rücktransformation
  - Probleme der BWT
- Diskussion verschiedener Resultate
  - Burrows & Wheeler
  - Cox & Bauer
  - Eigene Resultate

Große Datenmengen in der Bioinformatik I

- High-Throughput-Sequencing erzeugt durch n-fache Abdeckung große Datenmengen
- Oft Speicherung der Rohdaten gewünscht
- Speicherplatz ist teuer, teils langsamer Zugriff

Große Datenmengen in der Bioinformatik II

- **Lösung:** Verlustfreie Datenkompression
  $\rightarrow$ Daten unter Einsatz von weniger Speicherplatz darstellen
- Prinzip: Eliminierung von Redundanzen
- Anforderungen je nach Anwendung:
  - Schnelle Kompression/Dekompression
  - Wenig Speicherplatzverbrauch

Ein naiver Kompressionsalgorithmus I

- Eingabestring $S := AAAAAAAAATTT$
  $\rightarrow |S| = 12$
- Der Algorithmus fasst gleiche aufeinanderfolgende Zeichen (*Runs*) zusammen
- Resultat: $S' = 9A3T \rightarrow |S'| = 4$
  $\rightarrow$ 75% Speicherplatzeinsparung
- Problem: Viele Strings (z.B. ATATATAT) können nicht komprimiert werden

Ein naiver Kompressionsalgorithmus II

- Aufeinanderfolgende gleiche Zeichen können von diesem Algorithmus besser komprimiert werden
- Einige reale Kompressionsalgorithmen können davon ebenfalls profitieren
- Ist es möglich, einen String *reversibel* so umzuordnen, dass möglichst viele, möglichst lange *Runs* auftreten?
  $\rightarrow$ **Burrows-Wheeler-Transformation**

BWT - Kompression I

- Eingabestring
  $S := \texttt{aabrac}$
- Initialisierung einer Matrix M der Dimensionen $|S| \times |S|$
- Bildung aller zyklischen Rotationen des Eingabestrings

	M
0	aabrac
1	abraca
2	bracaa
3	racaab
4	acaabr
5	caabra

BWT - Kompression II

- Lexikographische Sortierung der Rotationen des Eingabestrings

M
aabrac
abraca
acaabr
bracaa
caabra
racaab

BWT - Kompression III

- Resultat:
  Das Tupel (L, I), wobei L die letzte Spalte der Matrix ist und I der Index des Eingabestrings in der Matrix ist
- $(L, I) = (caraab, 1)$

	M
0	aabrac
1	abraca
2	acaabr
3	bracaa
4	caabra
5	racaab

BWT – Rücktransformation I

- Eingabetupel: (L, I)
- Initialisierung einer Matrix M der Dimensionen $|L| \times |L|$
- Bereits bekannt: L ist die letzte Spalte von M

M

_ _ _ _ _ c
_ _ _ _ _ a
_ _ _ _ _ r
_ _ _ _ _ a
_ _ _ _ _ a
_ _ _ _ _ b

BWT – Rücktransformation II

- Die erste Spalte ist immer sortiert $\rightarrow$ Kann durch sortieren von L ausgefüllt werden

M

a_____c
a_____a
a_____r
b_____a
c_____a
r_____b

BWT – Rücktransformation III

- Die entstandene Matrix wird um ein Zeichen rotiert

M

_ _ _ _ c a
_ _ _ _ a a
_ _ _ _ r a
_ _ _ _ a b
_ _ _ _ a c
_ _ _ _ b r

BWT – Rücktransformation IV

- Die Matrix wird lexikographisch sortiert

M

_____aa
_____ab
_____ac
_____br
_____ca
_____ra

BWT – Rücktransformation V

- Die Schritte
  - Schreiben von L in die rechteste freie Spalte
  - Lexikographisches Sortieren

  werden wiederholt bis die Matrix voll ist

BWT – Rücktransformation VI

M
___caa
___aab
___rac
___abr
___aca
___bra

Sortieren →

M
___aab
___abr
___aca
___bra
___caa
___rac

BWT – Rücktransformation VII

M
__caab
__aabr
__raca
__abra
__acaa
__brac

Sortieren →

M
__aabr
__abra
__acaa
__brac
__caab
__raca

BWT – Rücktransformation VIII

M
_caabr
_aabra
_racaa
_abrac
_acaab
_braca

Sortieren →

M
_aabra
_abrac
_acaab
_braca
_caabr
_racaa

BWT – Rücktransformation IX

M
caabra
aabrac
racaab
abraca
acaabr
bracaa

Sortieren →

M
aabrac
abraca
acaabr
bracaa
caabra
racaab

BWT – Rücktransformation X

- M wurde vollständig rekonstruiert
- Zeile I ist der Originaltext

	M
0	aabrac
1	abraca
2	acaabr
3	bracaa
4	caabra
5	racaab

BWT – Rücktransformation (Original)

- Besseres Laufzeitverhalten - keine mehrfache Sortierung notwendig
- Berechnung von $C[ch] :=$ Anzahl der Zeichen in L, die im Alphabet vor ch vorkommen
- Berechnung von $P[i] :=$ Anzahl der Vorkommen von $L[i]$ im Präfix von L der Länge i
- Rekursive Berechnung des Originaltextes S

$$i[|L|] := I \tag{1}$$
$$i_{j-1} := P[i_j] + C[L[i_j]] \tag{2}$$
$$S[j] := L[i_{j+1}] \tag{3}$$

Effiziente Kompression

- Die Kompression kann effizient mit **Suffix Trees** implementiert werden
- Aber: Suffix Trees sind auf realer Hardware vergleichsweise langsam
  $\rightarrow$ Lösung: **Suffix Arrays** $\rightarrow$ Vortrag am 19.11.2012

Komprimierbarkeit

- Einige Zeichenkombinationen tauchen häufiger in Texten auf als andere, z.B. *an* in *and*
- Alle Zeilen von M, die mit *nd* beginnen, tauchen nacheinander auf (Sortierung)
- Durch Rotation ist bei nahezu allen Zeilen, die mit *nd* beginnen, das letzte Zeichen *a*
  $\rightarrow$ L enthält viele aufeinanderfolgende *a*

Skalierbarkeit

- Komplexität der BWT-Transformation ist $\mathcal{O}(n^2)$
  $\rightarrow$ Anwendung der BWT auf sehr große Datensätze (z.B. Genome) skaliert nicht
- Zeitverbrauch hauptsächlich durch Sortierung
  $\rightarrow$ Wahl des Sortieralgorithmus wichtig
- Burrows und Wheeler stellen einen für englische Texte optimierten Quicksort vor
- Lösung: Unterteilung des Datensatzes in kleinere *Blöcke* - Anwendung der BWT auf diese Blöcke

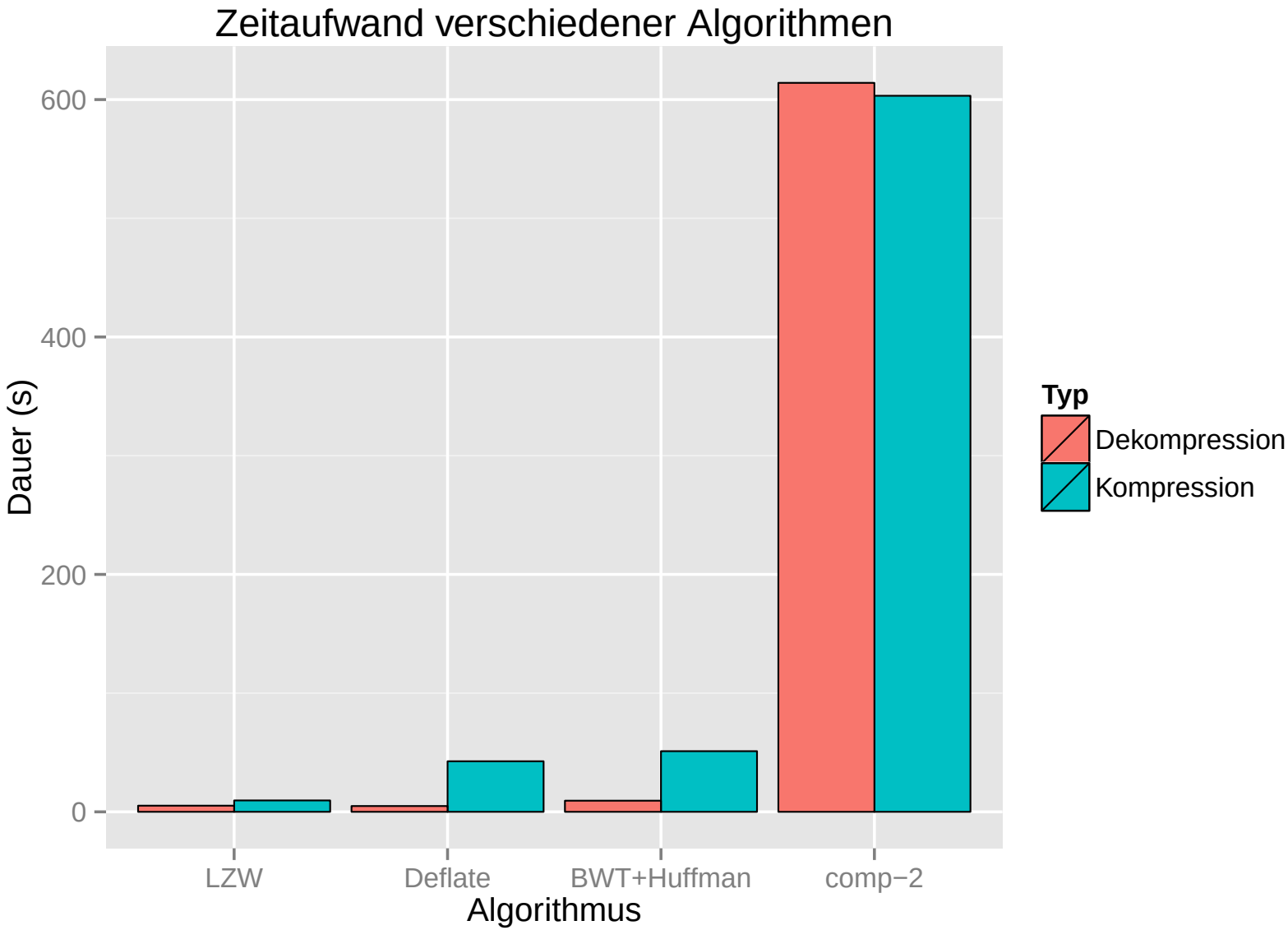
Overhead

- Die Länge des Tupels (L, I) ist insgesamt größer als $|S|$
- Genauer Längenunterschied hängt von der Repräsentation von (L, I) ab
- Lösung: Anwendung eines Kompressionsalgorithmus, z.B. der *Huffman-Kodierung* auf (L, I)
- Je nach Kompressionsalgorithmus oft vorherige Anwendung einer Codierung wie der Move-To-Front-Codierung (MTF)

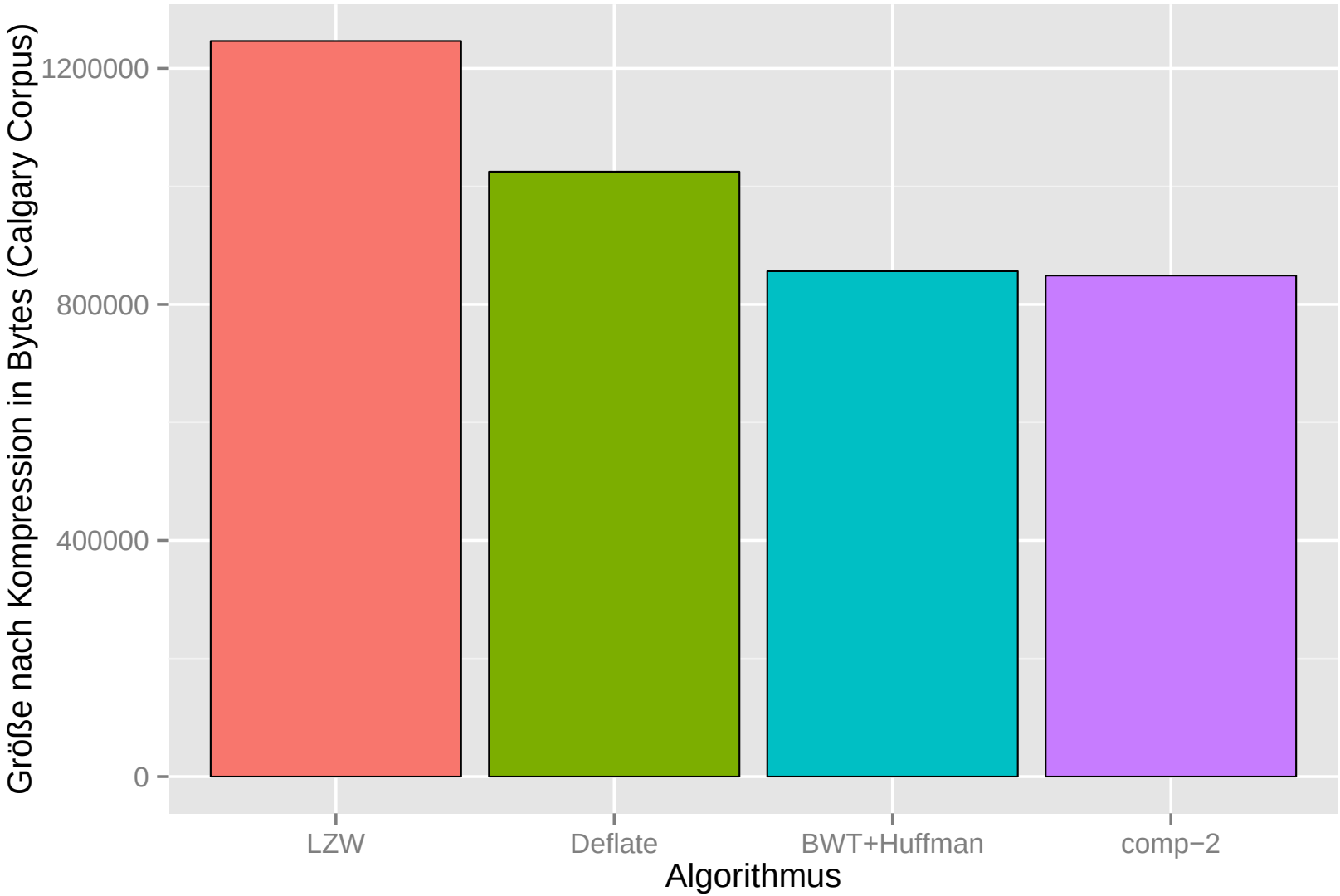
Vergleich mit anderen Algorithmen

- Kompression ist meist ein Kompromiss zwischen Geschwindigkeit und Platzeinsparung
- Einige Algorithmen sind für spezielle Datentypen (z.B. Bilder) besser geeignet
- Einige Algorithmen (z.B. *bzip2*) benutzen bereits die BWT
  $\rightarrow$ Kein Vergleich möglich

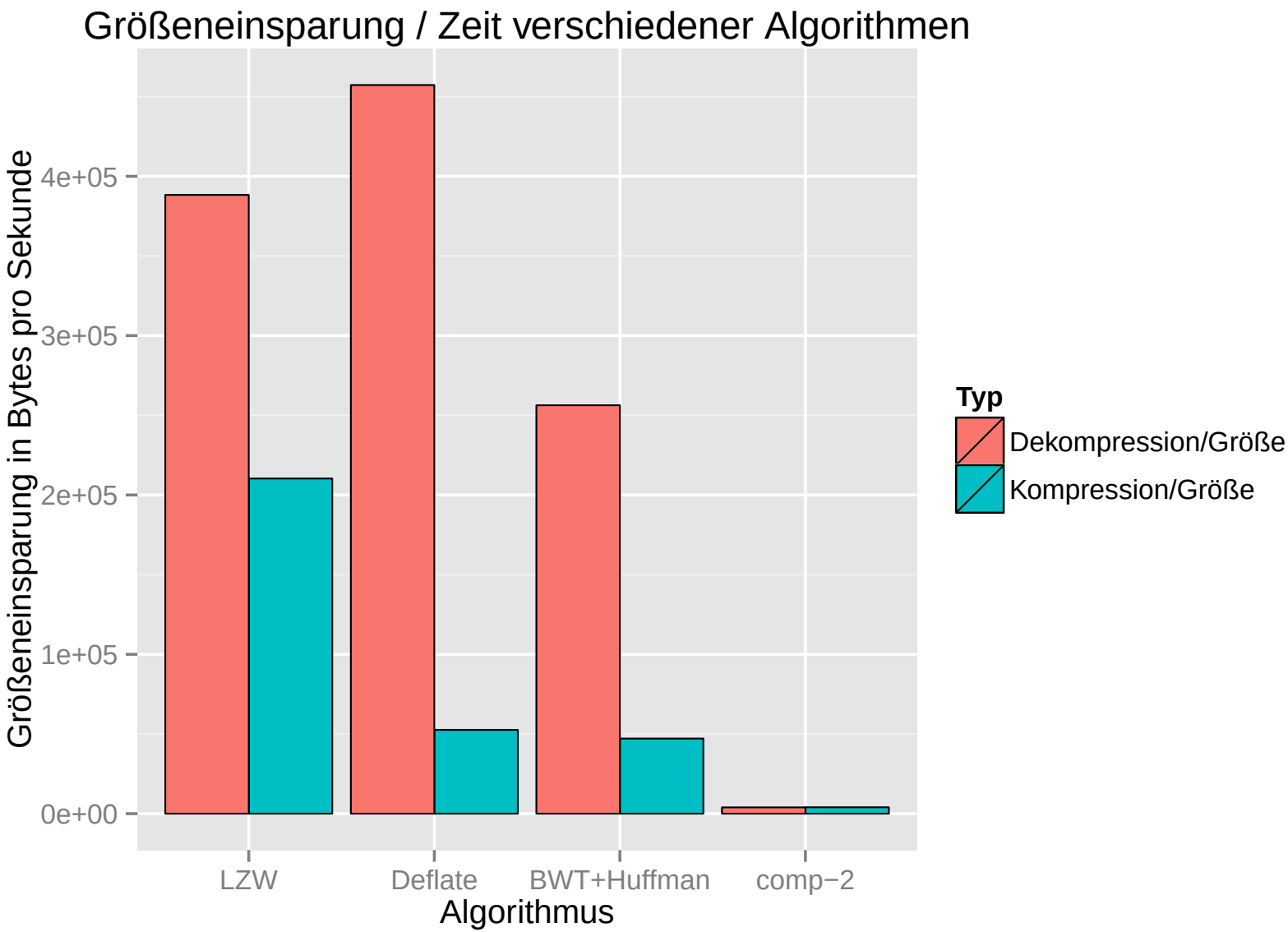
Resultate von Burrows & Wheeler I



Resultate von Burrows & Wheeler II



Resultate von Burrows & Wheeler III



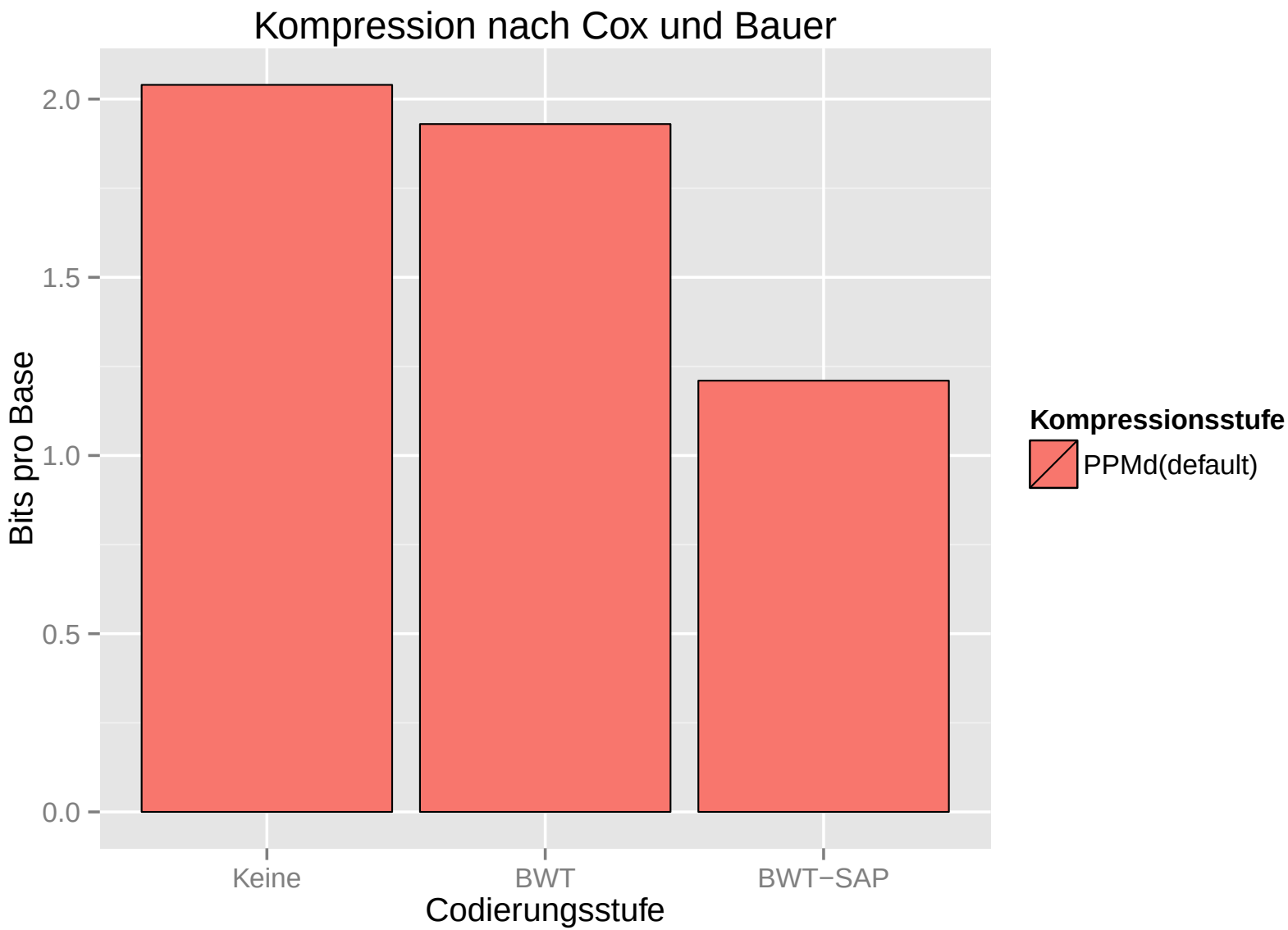
Diskussion der Resultate von Burrows & Wheeler

- Zeitdaten sind kritisch zu bewerten - Hardware von 1994
- BWT+Huffman: Gute Kompression, aber dennoch akzeptables Verhältnis *Kompression / Zeit*
- Beispiele beziehen sich nicht auf Bioinformatik-Datensätze, sondern auf einen Kompressionstestkorpus

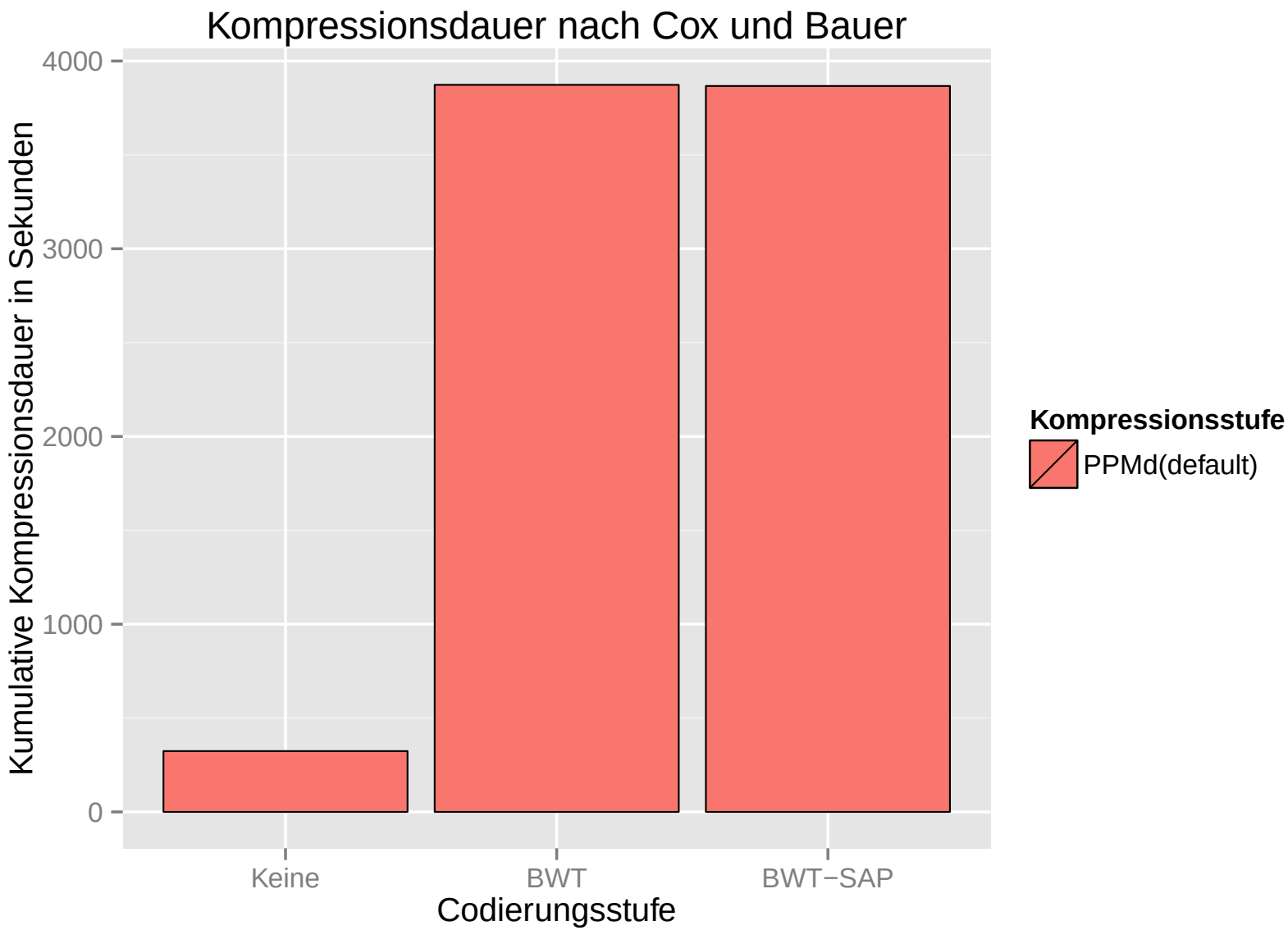
Resultate von Cox & Bauer I

- BWT muss zur Anwendung von Sequenzdaten generalisiert werden
- Cox & Bauer definieren dafür den BWT-SAP-Algorithmus
- Ein Algorithmus von Bauer zur Reduktion des BWT-Speicherverbrauches ohne stark erhöhten CPU-Zeitverbrauch wird verwendet

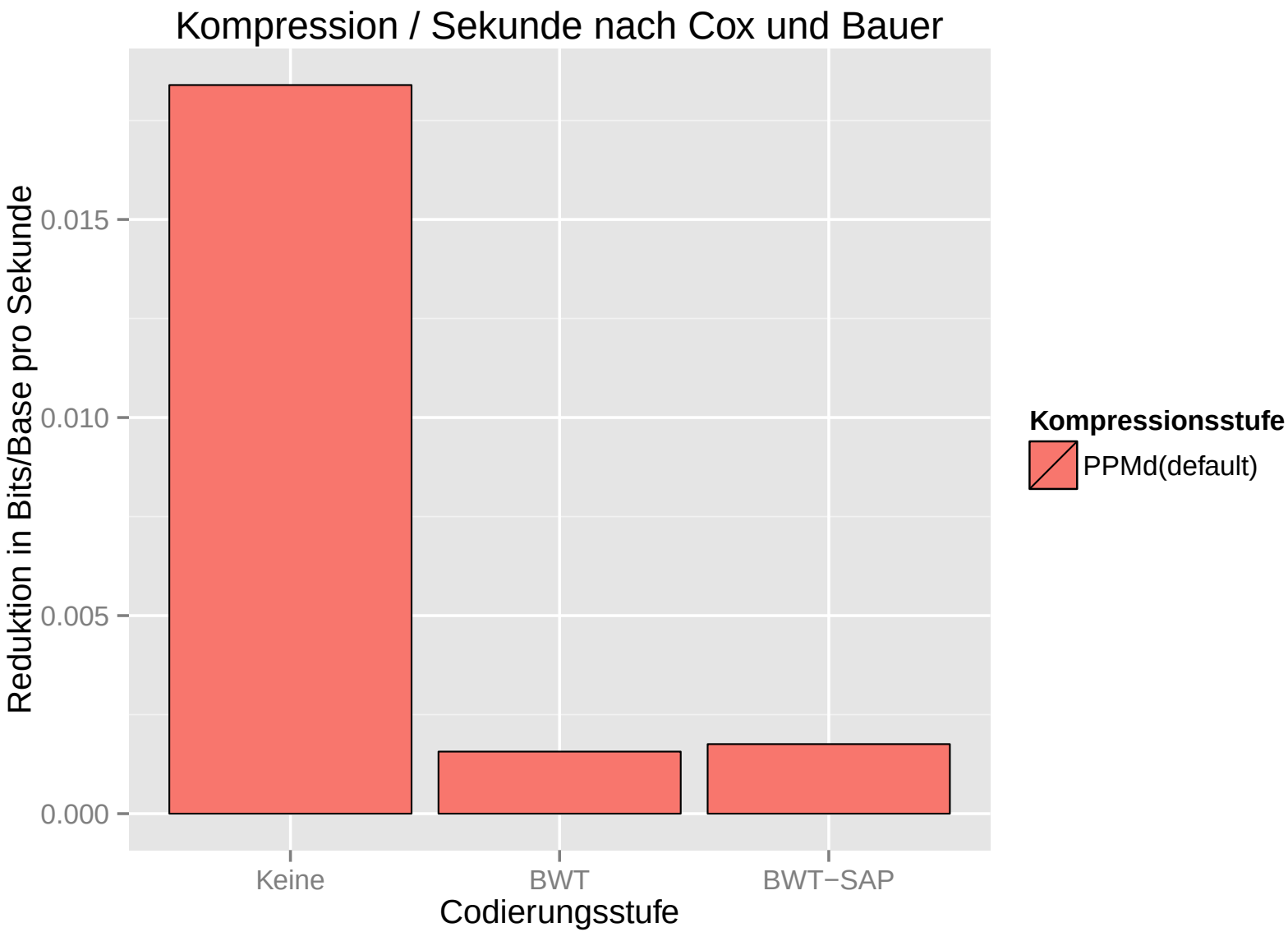
Resultate von Cox & Bauer II



Resultate von Cox & Bauer III



Resultate von Cox & Bauer IV



Resultate von Cox & Bauer V

- Im Vergleich zu früheren Verfahren konnte die Kompressionsrate um 10% erhöht werden
- Die Kompressionszeit wurde von 14 Stunden auf 1 Stunde gesenkt (auf schnellerer Hardware)

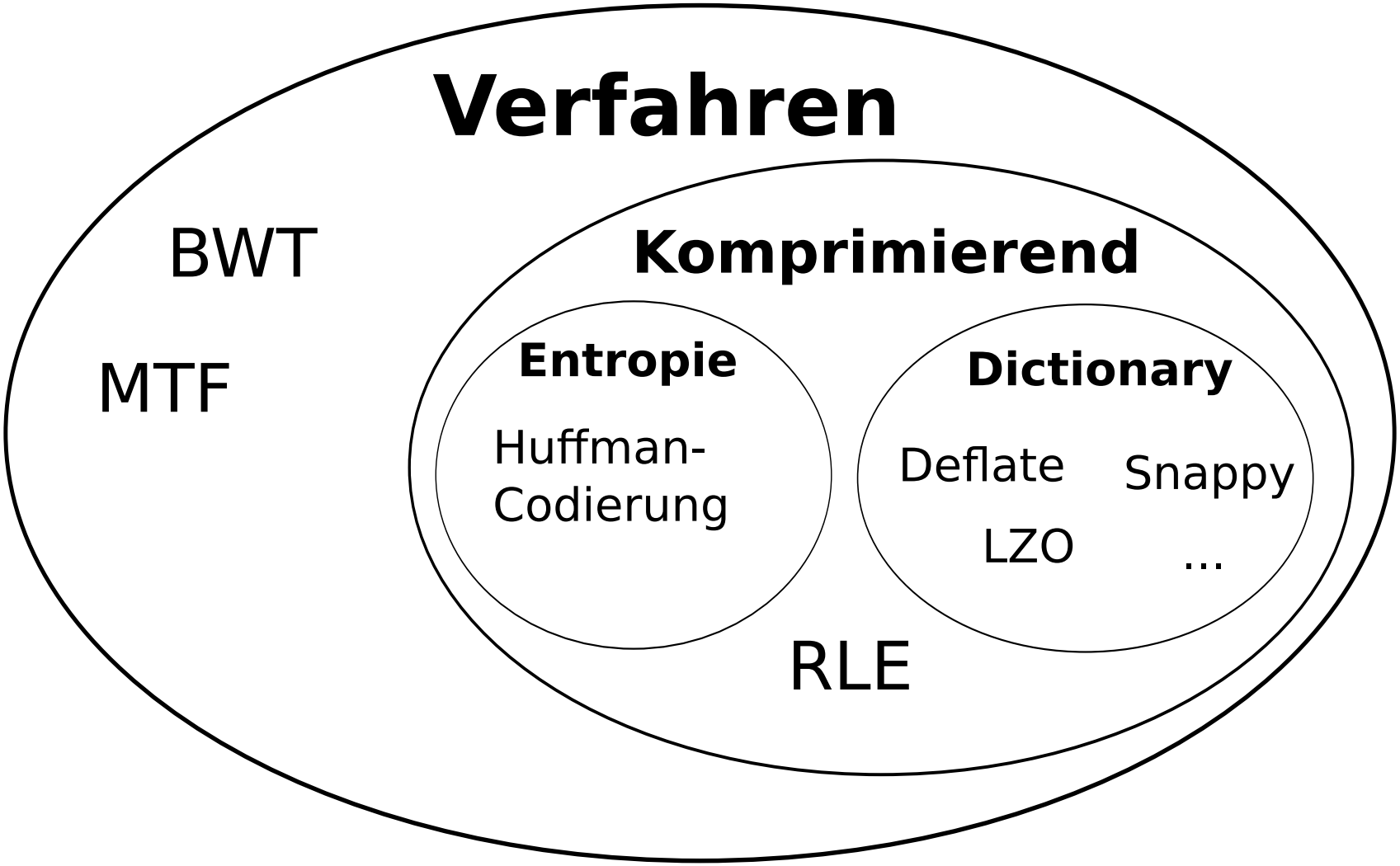
Warum eine eigene Implementierung?

- Alle vorgestellten Resultate verwenden die BWT zusammen mit anderen Verfahren
- Offene Fragen:
  - Wie verhält sich die BWT alleine und zusammen mit anderen Verfahren
  - Welche Kombination von anderen Verfahren erzielt für welche Datensätze die besten/schnellsten Resultate?
  - Sind moderne (nicht-BWT-basierte) Kompressionsalgorithmen der BWT für bestimmte Anforderungen überlegen?
  - Welche Blockgrößen sind für die BWT optimal?
  - Sind Bioinformatikdaten mit den bekannten Kompressionstestkorpora vergleichbar?

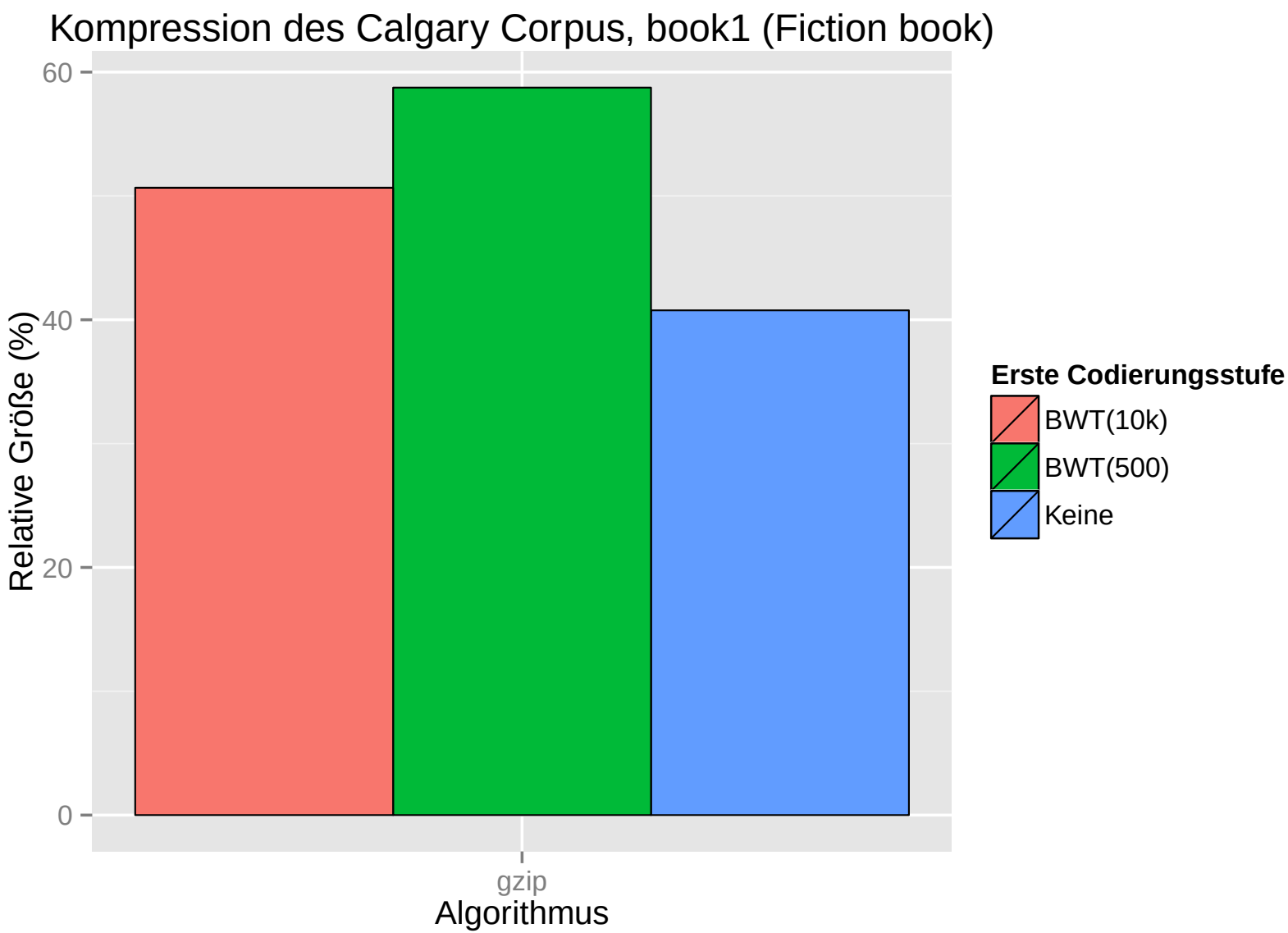
Methodik

- Erstellung eines Bioinformatik-Kompressionstestkorpus
- Software in *C++* zur Kompression der Datensätze mit verschiedenen Kombinationen von Algorithmen
- Teilweise wird auf Speicherung von kleinen Datenmengen verzichtet, um Verzerrung zu vermeiden
- Messung von:
  - Kompressionszeit
  - Resultierende Dateigröße

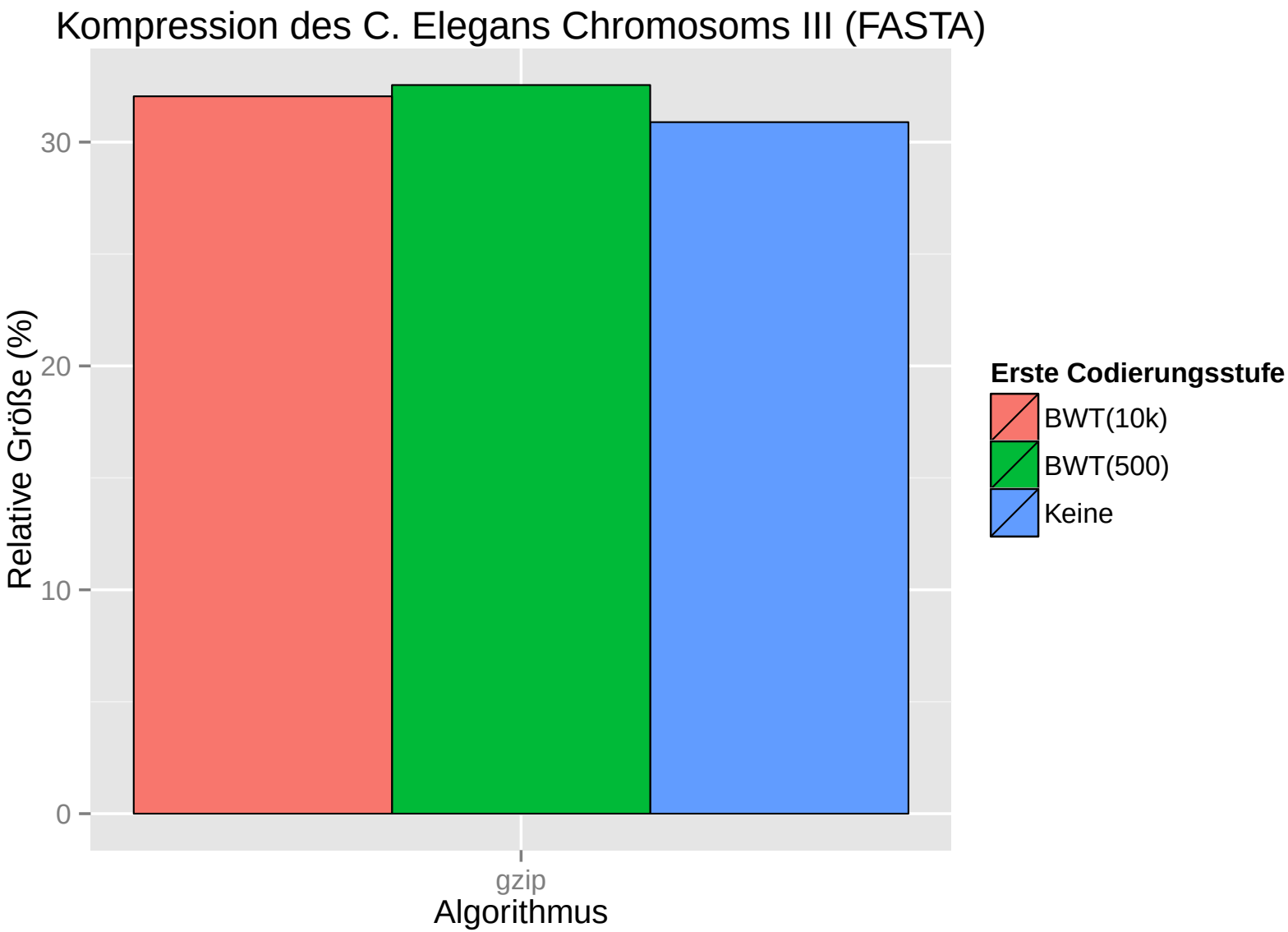
Implementierte Verfahren



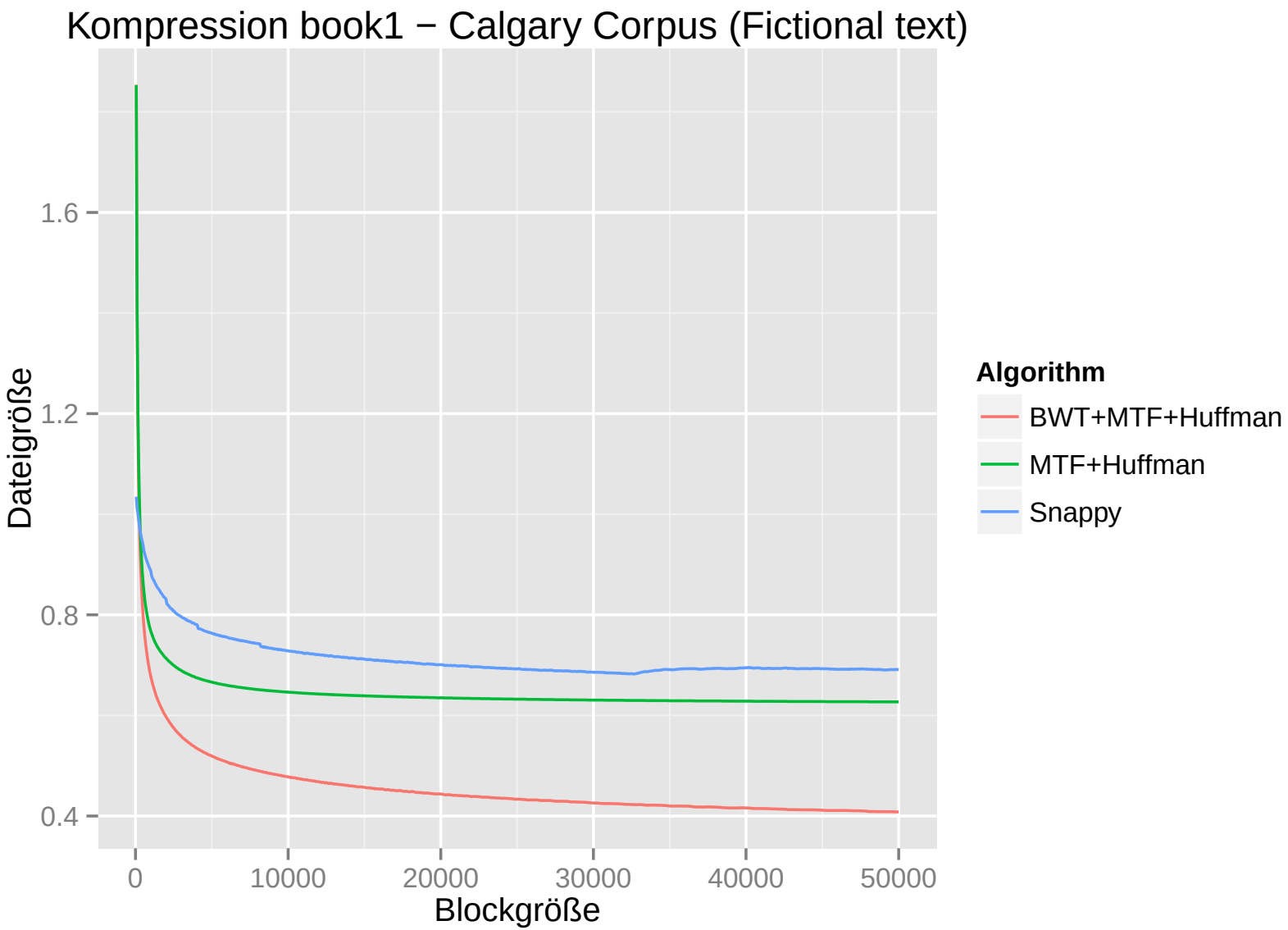
Resultate der Eigenimplementierung I



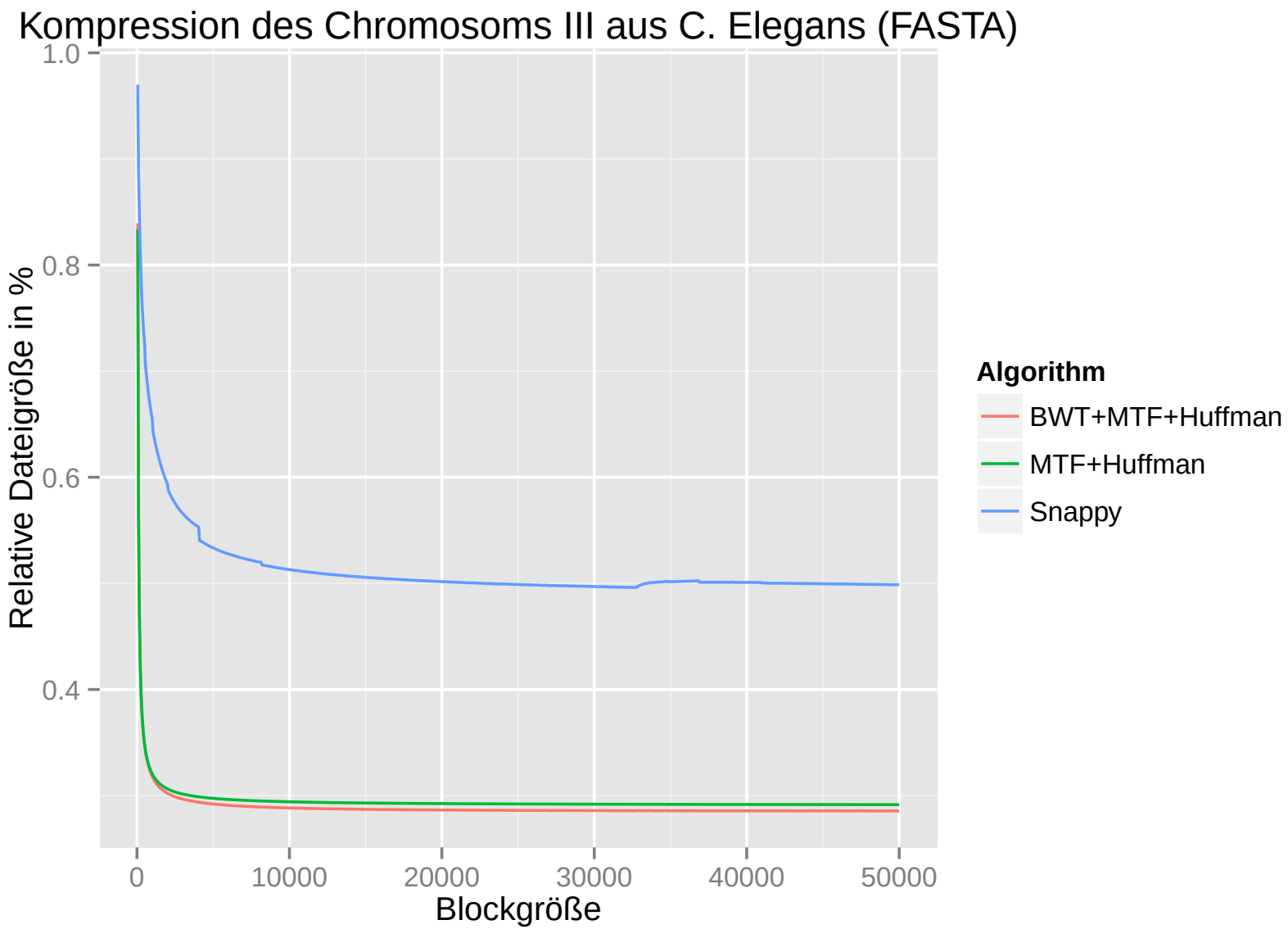
Resultate der Eigenimplementierung II



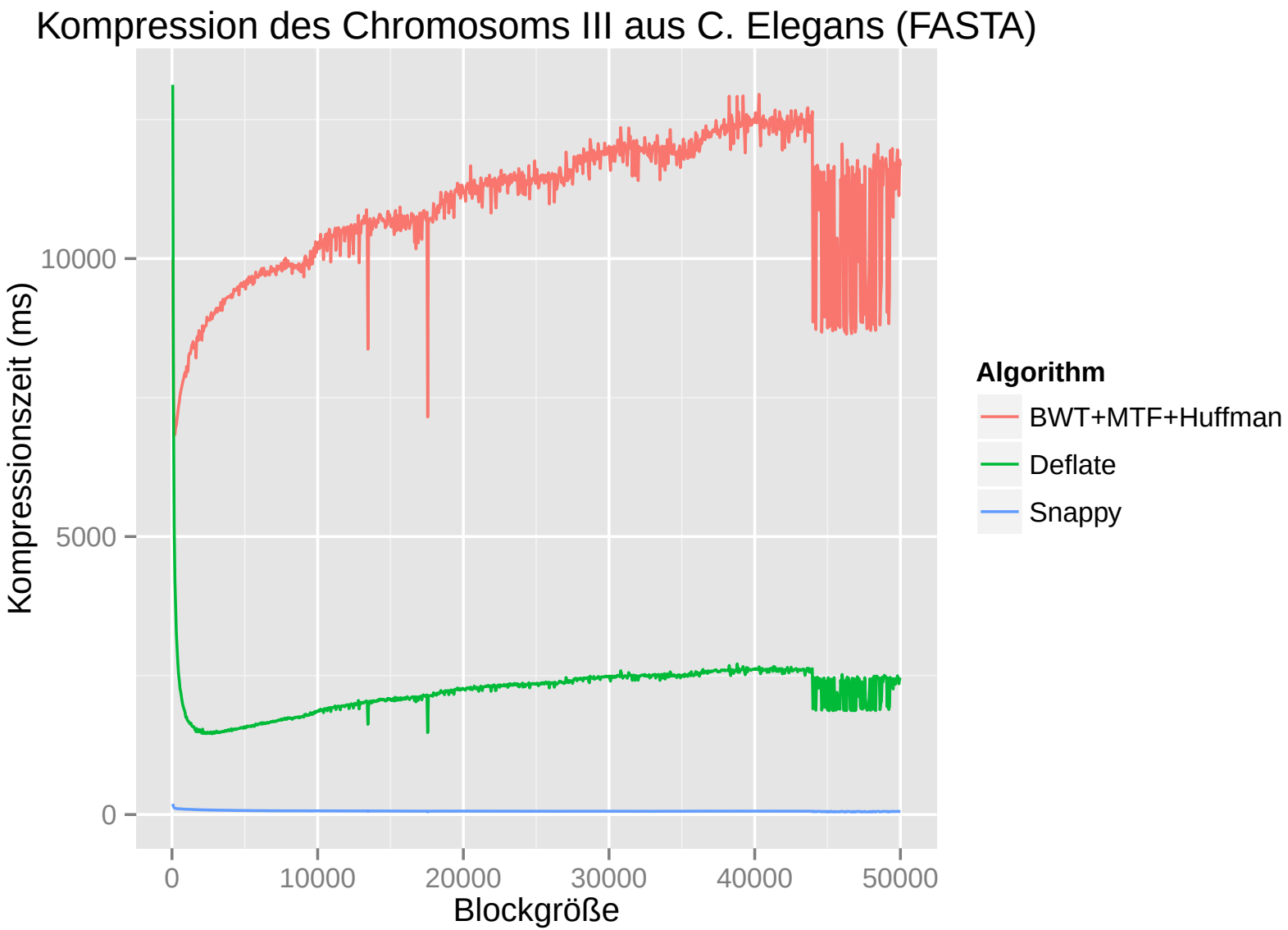
Resultate der Eigenimplementierung III



Resultate der Eigenimplementierung IV



Resultate der Eigenimplementierung V



Diskussion der eigenen Resultate I

- Nur ein kleiner Teil der Resultate konnte hier diskutiert werden
- Verbreitete Dictionary-basierte Verfahren wie *Deflate* erzeugen mit BWT größere Dateien als ohne BWT
- Kompression von *BWT+MTF+Huffman* erzeugt unter den BWT-Verfahren die besten Größeneinsparungen
- **Aber:** Langsame Kompression
  $\rightarrow$ Implementierung langsam? Stark Optimierter Encoder in C++

Diskussion der eigenen Resultate II

- BWT+MTF+Huffman erzeugt bei Genomdatensätzen nur unwesentlich kleinere Dateien als MTF+Huffman
- Andere Datensätze - auch Bioinformatische - zeigen diesen Effekt nicht
- Große Blockgrößen (über etwa 50 kiBytes) erhöhen die Laufzeit bei der BWT, erhöhen die Kompressionsrate aber nur unwesentlich

Diskussion der eigenen Resultate III

- **Konklusion:** Unmodifizierte BWT ist oft ungeeignet, wenn die Kompressions- oder Dekompressionszeit eine übergeordnete Rolle spielt
  $\rightarrow$ BWT-SAP
- Mögliche Alternative:
  Asymmetrische Kompressionsalgorithmen
  $\rightarrow$ Einzelfallentscheidung

Zusammenfassung I

- Datenkompression ist aufgrund der enormen Datenmengen, die durch Methoden wie High-Throughput-Sequencing entstehen, oftmals notwendig
- Verschiedene Algorithmen erreichen verschiedene Kompressionsraten und benötigen unterschiedlich viel Rechenzeit für Kompression und Dekompression

Zusammenfassung II

- Die Burrows-Wheeler-Transformation ist ein Algorithmus, um Zeichenketten besser komprimierbar zu machen
- Im Unix-Programm *bzip2* wird die BWT eingesetzt
- BWT-basierte Algorithmen sind nicht für alle Anwendungsfälle

Vielen Dank für eure Aufmerksamkeit!

Alle Quellen unter
https://github.com/ulikoehler/Proseminar