FRIEDRICH-SCHILLER-
UNIVERSITÄT
JENA

- Projektarbeit -

Kodierung Trinärer Bäume in DNA Sequenzen

für das Modul Molekulare Algorithmen

Friedrich-Schiller-Universität Jena
Fakultät für Mathematik und Informatik

eingereicht von Dominic Wild und Hanno Barschel

Betreuer: Dr.-Ing. habil. Thomas Hinze

Jena, 1. Juli 2023

Inhaltsverzeichnis

1 Einleitung

Wir haben uns die Aufgabe 18 “Kodierung und Dekodierung beliebiger trinärer Bäume in möglichst kurzen DNA-Zeichensequenzen“ ausgesucht. In diesem Kapitel werden wir zunächst die Details der Aufgabenstellung genauer analysieren und anschließend eine Übersicht der Kapitel in dieser Projektarbeit geben.
Zuerst muss sich natürlich die Frage gestellt werden, was ein trinärer Baum überhaupt ist:

Definition 1. *Ein **trinärer Baum** T ist ein zyklenfreier Graph dessen Knoten hierarchisch angeordnet sind. Jeder Knoten hat einen Namen und eine Liste an Nachfolgerknoten (sogenannte Kinder), die sich auf der nächstunteren Ebene befinden. Ein Knoten hat entweder kein Kind (Blattknoten) oder genau 3 Kinder (innerer Knoten).*

In Abb. 1 befindet sich ein einfacher trinärer Baum. Er kann formal durch die Zeichenkette

$$a(b(e(), f(k(), l(), m()), g()), c(), d(h(), i(), j()))$$

beschrieben werden. Im Folgenden werden wir leere Klammern weglassen, um eine bessere Übersichtlichkeit zu ermöglichen. Obige Zeichenkette verändert sich also zu

$$a(b(e, f(k, l, m), g), c, d(h, i, j)).$$

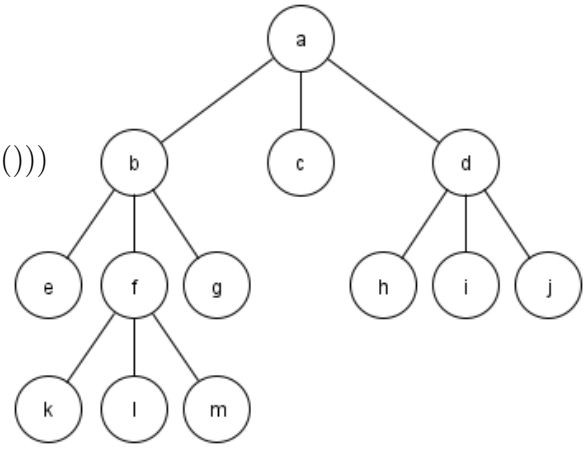

Abbildung 1: Einfacher Beispielbaum

Es ist zunächst sinnvoll, eine Datenstruktur zur Kodierung trinärer Bäume zu implementieren. Die Eingabe des Nutzers sei dabei die oben beschriebene Struktur mit beliebigen, aber eindeutigen Knotennamen, die aus kleingeschriebenen englischen Buchstaben oder Ziffern bestehen. Nun soll ein Algorithmus entworfen werden, der den Baum in eine Zeichenkette zerlegt, die ausschließlich die Zeichen **A**, **C**, **T** und **G** verwendet. Diese Menge wird im Folgenden mit $\mathcal{K}$ bezeichnet. Dabei sollen die Namen der Knoten durch ein globales Wörterbuch kodiert werden. Das Wörterbuch übersetzt Buchstaben und Ziffern zu DNA Sequenzen. Die letzte Aufgabe ist es schließlich, eine Dekodie-

rungsfunktion zu schreiben, die eine DNA Sequenz wieder in den dazugehörigen Baum übersetzt.

Im nächsten Kapitel werden drei mögliche Kodierungen genauer beschrieben. Es wird genauer auf Idee und Komplexität der Algorithmen eingegangen. Anschließend werden wir die Algorithmen an verschiedenen Bäumen testen und Laufzeiten vergleichen. Außerdem vergleichen wir unsere Algorithmen für Bäume mit anderen Verzweigungsgraden und verschiedene Größen von $\mathcal{K}$. Weiterhin gehen wir auf Implementierungsdetails ein und im letzten Teil werden die Ergebnisse schließlich noch analysiert, diskutiert und zusammengefasst.

2 Algorithmen

Zum Kodieren des trinären Baumes in DNA-Basensequenzen muss man sich zuerst ein Wörterbuch überlegen, welches normalsprachliche Zeichen zu Basensequenzen übersetzt, sowie ein inverses Alphabet, das die Basensequenzen zurück übersetzt. Da die trinäre Baumstruktur mit den Symbolen "(", ")" und "," festgelegt wird und die einzigartigen Knotennamen aus kleinen lateinischen Buchstaben und Ziffern bestehen dürfen, müssen wir somit insgesamt 39 verschiedene Zeichen kodieren.
Da wir das Ziel verfolgen, eine möglichst kurze Kodierung zu erreichen, ist es sinnvoll, das Symbol ")" nicht mitzukodieren, da dieses keinen zusätzlichen Informationsgehalt in trinären Bäumen realisiert und nur die Lesbarkeit der Baum-Zeichenkette erhöht. Das begründet sich aus der Tatsache, dass wir bereits wissen, wie viele Kindknoten pro Knoten existieren: 3 oder 0. Befindet sich also hinter einem Knoten eine offene Klammer, so hat der Knoten genau 3 Kindknoten - ist das nicht der Fall, handelt es sich um einen Blattknoten. Dadurch kann eine geschlossene Klammer rekonstruiert werden, indem einfach durch eine Zeichenkette ohne geschlossene Klammern iteriert wird und die Anzahl der bereits gesehenen Knoten bzw. Kommas gezählt wird.
Somit benötigen wir für unsere Kodierung nur 38 verschiedene Symbole. Das Komma kann nicht weggelassen werden, da Knotennamen eine variable Länge haben können. Die offene Klammer ist wichtig, um Strukturinformation zu kodieren. Damit haben wir mit 38 kodierten Zeichen einen minimalen Zeichensatz erreicht. Diese Menge an Zeichen wird im Folgenden mit $\mathcal{A}$ bezeichnet.
Mit den 4 Basennamen lassen sich für eine Wortlänge n genau 4^n verschiedene Wörter

erstellen. Dementsprechend braucht man für das Kodieren von s Symbolen eine Wortlänge von

$$n = \lceil log_4(s) \rceil.$$

Für $m = 38$ Symbole ergibt sich entprechend eine Wortlänge von $n = 3$, wenn davon ausgegangen wird, dass die Länge der Wörter gleich ist.

Um den benötigten Speicherplatz einer Kodierung auszurechnen, müssen wir zunächst den Zusammenhang zwischen der Anzahl an Knoten und der Anzahl an Klammern beziehungsweise Kommas quantifizieren. Sei dazu B ein trinärer Baum mit k Knoten. Für jeden Knoten gibt es zwei Möglichkeiten: Entweder er hat kein Kind oder er hat genau 3 Kinder. Im ersten Fall gibt es weder eine Klammer noch ein Komma. Im zweiten Fall gibt es für den Knoten genau eine offene Klammer und zwei Kommas zur Trennung der 3 Kinder. Doch wie viele Elternknoten gibt es in einem trinären Baum?

Satz 1. *Die Anzahl der Knoten mit genau 3 Kindern ist $\frac{k-1}{3}$ für einen trinären Baum mit k Knoten ($k \geq 1$).*

Beweis. Beweis über vollständige Induktion. Statt aber die Anzahl der Knoten immer um eins zu inkrementieren gilt folgende Aussage: Sei T ein beliebiger trinärer Baum mit k Knoten. Wir wollen die Aussage für einen Baum mit mehr Knoten zeigen, also müssen wir Knoten hinzufügen. Dazu nehmen wir einen beliebigen Blattknoten aus dem Baum und fügen 3 Kinder hinzu. Das heißt k muss immer um 3 inkrementiert werden (beginnend bei 1).
Ind.Anfang: $k = 1$. Das heißt es gibt nur den Wurzelknoten ohne Kinder, also ist die Formel korrekt.
Ind.Behauptung: Die Aussage ist für einen trinären Baum mit k Knoten korrekt.
Ind.Beweis: Sei T' ein Baum mit $k + 3$ Knoten. Nach obiger Aussage ist er aus einem Baum T mit k Knoten entstanden. In diesem gibt es nach der Induktionsbehauptung genau $\frac{k-1}{3}$ Knoten mit 3 Kindern. T' ist aus T entstanden, indem an einem Blattknoten 3 Kinder angehängt wurden. Das heißt es gibt genau einen neuen Knoten mit 3 Kindern:

$$\frac{k-1}{3} + 1 = \frac{k-1}{3} + \frac{3}{3} = \frac{k+2}{3}.$$

□

Für jeden inneren Knoten muss es in der Kodierung genau eine offene Klammer geben und zwei Kommas. Also ergibt sich die Anzahl der Klammern $A(\text{"("})$ und die Anzahl der Kommas $A(\text{","})$ zu

$$A(\text{"("}) = \frac{k-1}{3} \text{ und } A(\text{","}) = \frac{2k-2}{3}.$$

Zur Kodierung und Dekodierung haben wir uns drei verschiedene Ansätze überlegt. Im Folgenden geben wir eine detailliertere Beschreibung der 3 Algorithmen und analysieren und vergleichen den jeweils benötigten Speicherplatz.

2.1 Trivialer Ansatz

2.1.1 Konzept

Für diesen Ansatz nutzen wir die ersten 38 Wörter des 3-fachen kartesischen Produktes unserer Basen-Zeichenmenge $\mathcal{K}$: $\mathcal{K} \times \mathcal{K} \times \mathcal{K}$.
Die Kodierung in DNA-Sequenzen erfolgt durch schlichtes Ersetzen der Symbole aus $\mathcal{A}$ durch ihre Basenkodierung und das Auslassen der geschlossenen Klammern.
Die Dekodierung realisieren wir, indem wir durch die DNA-kodierte Sequenz iterieren, und für je 3 Symbole die Übersetzungen ins Normalsprachliche, gemäß des inversen Wörterbuchs, durchführen. Anschließend rekonstruieren wir die geschlossenen Klammern, wie es oben bereits beschrieben wurde, und übersetzen das in einen Baum.

2.1.2 Komplexität

Hier werden wir den benötigten Speicherplatz berechnen: Sei dazu B ein trinärer Baum mit k Knoten, deren Namen insgesamt m Buchstaben enthalten. Das impliziert das Vorhandensein von $\frac{2k-2}{3}$ Kommas im Baum, wie oben beschrieben. Außerdem impliziert es das Vorhandensein von $\frac{k-1}{3}$ offenen Klammern. Entsprechend ergibt sich eine Gesamtzeichenlänge von

$$3 \cdot m + 3 \cdot \frac{2 \cdot (k-1)}{3} + 3 \cdot \frac{k-1}{3} = 3 \cdot m + 3 \cdot k - 3 = 3 \cdot (m + k - 1)$$

der DNA-kodierten Sequenz.

2.2 Klammeransatz

2.2.1 Konzept

Die Idee bei diesem Ansatz ist es, die Klammer und das Komma separat zu kodieren. Das heißt, es wird ein Alphabet erstellt, das das Komma durch A und die Klammer durch AA kodiert. Damit man wieder eindeutig dekodieren kann, wird der Kodierungsbuchstabe A sonst nicht im Alphabet verwendet. Dadurch wird die Kodierung der Buchstaben zwar eventuell länger, aber die Kodierung des Kommas ist deutlich kürzer. Das heißt, dass diese Variante besser ist, wenn die Gesamtzahl m an Buchstaben relativ gering ist.
Analog zu der Überlegung am Anfang ergibt sich die Länge einer Kodierung durch

$$n = \lceil log_3(s) \rceil.$$

s ist 36, weil Klammer und Komma extra kodiert wurden. Der Logarithmus wird zur Basis 3 genommen, weil wir nur noch die 3 Symbole $\{C, T, G\}$ verwenden können. Damit ist n gleich 4, also schlechter als oben.

2.2.2 Komplexität

Sei B ein trinärer Baum analog zu Kapitel 2.1.2 mit k Knoten und einer Gesamtzahl von m Buchstaben. Die Kommas werden nur noch durch ein Symbol kodiert, die Klammern durch zwei. Jeder normale Buchstabe wird jetzt mit 4 Symbolen kodiert. Damit ergibt sich die Gesamtlänge der kodierten DNA Sequenz zu

$$4 \cdot m + \frac{2 \cdot (k-1)}{3} + 2 \cdot \frac{k-1}{3} = 4 \cdot m + \frac{4}{3} \cdot (k-1) = 4 \cdot \left(m + \frac{k-1}{3} \right).$$

2.3 Verbesserter Klammeransatz

2.3.1 Konzept

Für diesen Algorithmus wurde die Idee des Klammeransatzes noch weiter verbessert. Auch hier wählen wir separat eine Kodierung für Klammern und Kommas. Analog wie oben soll ein Komma durch A und eine Klammer durch AA kodiert werden.
Im Unterschied zum normalen Klammeransatz kann das Symbol A im Alphabet noch vorkommen, es darf nur nicht an der ersten Stelle stehen. Das heißt die Kodierung $a = BAC$ ist möglich, aber $a = ABC$ nicht, weil das Symbol A an der ersten Stelle steht.

Es ist nicht mehr ganz so klar erkennbar, dass die Kodierung eindeutig dekodierbar ist: Sei s ein kodierter Baum. In s steht zunächst die Kodierung der Wurzel in Dreierblöcken. Wenn einer dieser Blöcke mit einem A startet, wissen wir, dass es eine Klammer ist, weil ein Buchstabe nicht mit A starten darf und ein Komma an der Stelle nicht vorkommen kann. Nach dem zweiten A für die Klammerkodierung folgt die Kodierung des ersten Kindes in Dreierblöcken. Startet einer dieser Blöcke mit A so analysieren wir das nächste Symbol: Ist es auch ein A, so ist es eine Klammer, ansonsten ein Komma. Durch dieses Verfahren können wir den Baum rekonstruieren.
Die Berechnung der Kodierungslänge ist jetzt etwas komplizierter. Es gilt

$$s = 3 \cdot 4^{n-1} \Longleftrightarrow n = \lceil \log_4(\frac{s}{3}) \rceil + 1$$

Mit $s = 36$ ergibt sich die Länge $n = 3$. Offensichtlich muss diese Kodierung damit besser sein als alle anderen bisher betrachteten Kodierungen, da sie die gleiche Wortlänge wie der triviale Ansatz mit kürzeren Komma- und Klammerkodierungen hat.

2.3.2 Komplexität

Sei B der trinäre Baum analog zu den anderen Komplexitätsbetrachtungen. Die Gesamtlänge der kodierten DNA Sequenz ergibt sich zu

$$3 \cdot m + \frac{2 \cdot (k-1)}{3} + 2 \cdot \frac{k-1}{3} = 3 \cdot m + \frac{4}{3} \cdot (k-1).$$

Also ist diese Kodierung echt besser als die bisher betrachteten Kodierungen.

2.3.3 Zusatz

Für diesen Ansatz gibt es eine Alternative: Statt Klammer mit AA zu kodieren, kodieren wir sie durch B. Wenn die Länge einer Kodierung n gleich bleibt, dann gibt es für jede Klammer einen Buchstaben weniger in der Kodierung. Leider wird n aber 4 und die Länge einer kodierten DNA Sequenz ergibt sich als

$$4 \cdot m + \frac{2 \cdot (k-1)}{3} + \frac{k-1}{3} = 4 \cdot m + k - 1$$

Das heißt dieser Ansatz ist sinnvoll, wenn jeder Knoten nur wenig Buchstaben hat und es viele Klammern und Kommas gibt. Außerdem hängt es von der Anzahl s der benötigten Buchstaben aus $\mathcal{A}$ ab: Wäre diese beispielsweise nur 32, würde eine Dreierkodierung

ausreichen und die Variante wäre besser als die originale Variante.

2.4 Kodierung mit variabler Wortlänge

Für Ansatz 2.2 und 2.3 haben wir schon eine variable Wortlänge genutzt, indem wir das Komma bspw nur mit A kodiert haben. Durch das Nutzen variabler Wortlängen kann das noch verbessert werden. Wie effektiv diese Kodierung ist, hängt von der Länge des Alphabets ab. Außerdem nutzt der Algorithmus Wahrscheinlichkeiten: Kommt ein Buchstabe besonders häufig vor, so sollte er eine kurze Kodierung haben. Dieses Prinzip haben wir bereits angewandt indem wir die Kommakodierung besonders kurz gemacht haben.
Um eine effiziente Kodierung für $\mathcal{A}$ zu erreichen schlagen wir folgendes vor:
Für das Komma wähle die Kodierung A; für die Klammer die Kodierung AA. Sortiere die Buchstaben nach absteigender Wahrscheinlichkeit (je nachdem wie oft sie im Baum erwartet werden). Für die ersten 4 Buchstaben wähle jeweils eine der Kodierungen $\{CC, CA, CT, CG\}$. Der Rest wird mit beliebigen Dreierkombinationen kodiert, die mit G oder T starten. Damit haben wir $2 + 4 + 2 \cdot 4 \cdot 4 = 38$ mögliche Kodierungen, also genau ausreichend.
Ein kodierter Baum kann nun folgendermaßen dekodiert werden:
Startet die Zeichenfolge mit einem A, so schauen wir, ob danach noch ein A kommt. Wenn ja, dann wähle eine Klammer, ansonsten ein Komma. Finden wir ein C, so wissen wir, dass die Kodierung die Länge 2 hat, weil nur Kodierungen der Länge 2 mit C starten. Analog können wir beim Finden von G oder T auf eine Kodierung der Länge 3 schließen. Klammer und Komma müssen extra kodiert werden. Um das zu sehen betrachten wir den Fall, dass sowohl die Kodierung A als auch AA ein Buchstabe ist: In diesem Fall können wir die Zeichenkette AAA nicht mehr eindeutig dekodieren.
Die Kodierung kann auch weiter abgewandelt werden, indem A gelöscht und $\{AA, AC, AG, AT\}$ hinzugefügt wird. Das kann sinnvoll sein, weil es jetzt 8 Kodierungen der Länge 2 gibt und nicht nur 5. Das Komma und die Klammer können in diesem Fall beliebig kodiert werden. Ob diese Kodierung sinnvoll ist, hängt von den Häufigkeiten der Buchstaben ab. Diese Kodierung werden wir nicht weiter betrachten, da sie von den Knotennamen im Baum abhängt. Damit ist sie im worst-case nicht besser, als der Algorithmus aus 2.3. In der realen Welt sollte die Kodierung dennoch sinnvoll sein.

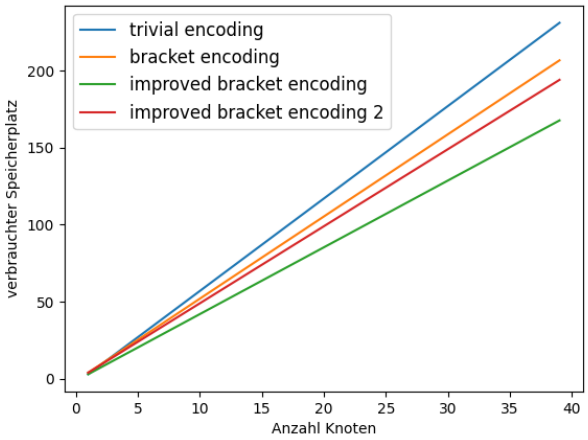

Abbildung 2: Vergleich des Speicheraufwands aller Algorithmen, falls $m = k$.

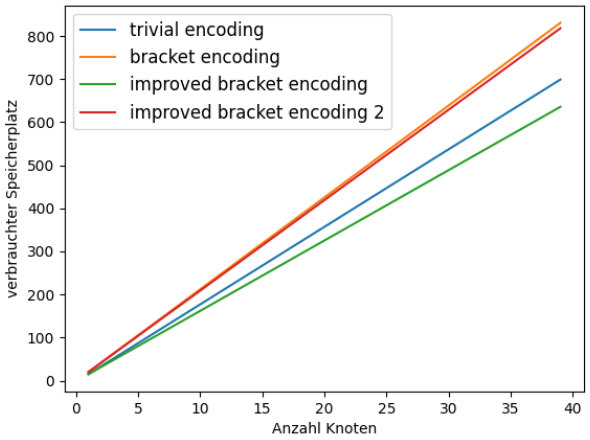

Abbildung 3: Vergleich des Speicheraufwands aller Algorithmen, falls $m = 5 \cdot k$.

2.5 Speichervergleich

Nun haben wir den Speicherbedarf der einzelnen Algorithmen berechnet. In diesem Kapitel werden wir sie vergleichen und genauer analysieren.
Die Funktionen für die Laufzeiten sind zwar linear, aber haben trotzdem die 2 Variablen k und m. Um eine 2-dimensionale grafische Betrachtung zu ermöglichen, wählen wir m in Abhängigkeit von k. Im ersten Fall (Abb. 2) ist $m = k$, das heißt jeder Knotenname besteht nur aus einem Buchstaben. Im zweiten Fall (Abb. 3) gehen wir von $m = 5 \cdot k$ aus, also dass jeder Knotenname aus 5 Buchstaben besteht.

Die Abbildungen spiegeln im Prinzip genau unsere Erwartungen wider: Im ersten Fall sind Knoten nur kurz kodiert, also ist jede Kodierung besser, als die triviale Kodierung. Das liegt daran, dass das triviale Alphabet Klammer und Komma jeweils mit 3 Symbolen kodiert. Die anderen Alphabete nutzen weniger Symbole. Steigt die Anzahl der Buchstaben pro Knotennamen aber auf 5 so werden sowohl die Klammerkodierung als auch der Zusatz der verbesserten Klammerkodierung schlechter, weil ein Buchstabe mit 4 Symbolen kodiert wird und nicht, wie im trivialen Alphabet, mit 3 Symbolen. Der verbesserte Klammeralgorithmus ist trotzdem besser, weil normale Buchstaben analog durch 3 Symbole kodiert werden, Klammer und Komma aber durch 2 bzw. 1 Symbol. Der Zusatz der verbesserten Klammerkodierung wird für 38 Zeichen immer schlechter sein, als der originale Algorithmus zur verbesserten Klammerkodierung, weil ein Buchstabe mit 4 Symbolen statt nur mit 3 kodiert wird. Allerdings hängt das von der Länge der Zeichen ab, die kodiert werden müssen.

3 Weitere Komplexitätsbetrachtungen

3.1 Laufzeitbetrachtung

Zum Testen der Laufzeiten verwenden wir Bäume mit unterschiedlichen Mengen an Knoten: Bäume mit maximaler Tiefe und Bäume mit maximaler Breite. Die Ergebnisse der Lautzeitbetrachtung sind in Abb. 4 dargestellt.
Klare Unterschiede in der Laufzeit lassen sich lediglich beim Dekodieren feststellen, da alle Algorithmen dieselbe Kodierungsfunktion (nur mit anderen Alphabeten) verwenden. Auch die Art des Baumes hat keinen Einfluss auf die Laufzeit beim Kodieren.
Beim Dekodieren ist erkennbar, dass das triviale Dekodierungsverfahren eindeutig schneller als alle anderen Verfahren ist, da dieses nicht mit unterschiedlichen Wortlängen umgehen muss. Alle anderen Klammeransätze haben ähnliche Laufzeiten. Die Art des Baumes scheint kaum einen Einfluss auf die Laufzeit zu haben.
Letztendlich sind die Funktionen zum Kodieren und Dekodieren jedoch so schnell, dass auch größere Graphen mit 5000 Knoten im Bruchteil von Sekunden kodiert und dekodiert werden können. Dementsprechend erscheint das Betrachten der Laufzeit als eher nebensächlich.

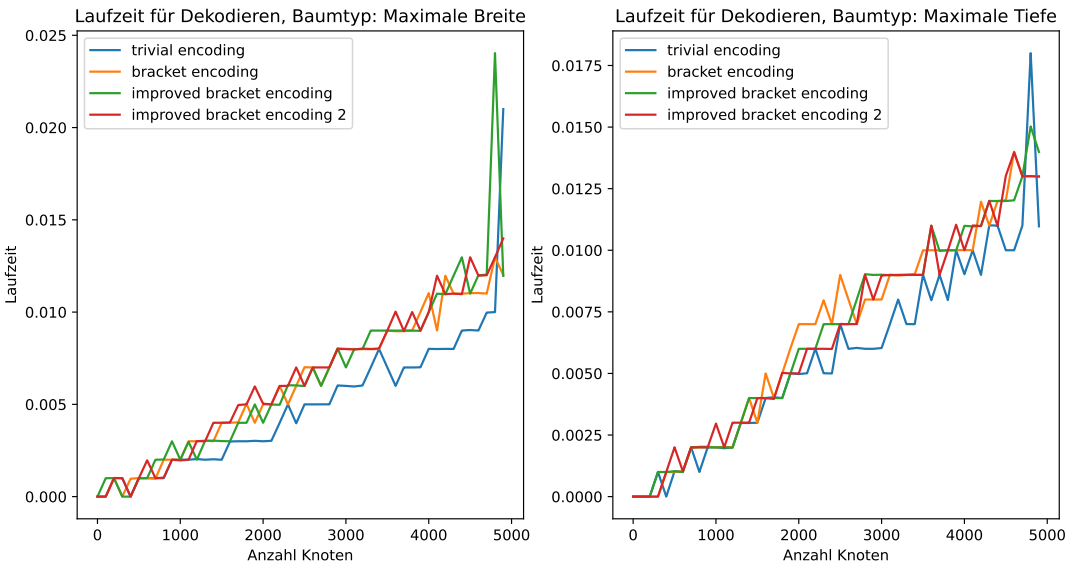

Abbildung 4: Laufzeiten bei der Dekodierung

3.2 Speicherkomplexität bei anderem Zeichensatz

Nun betrachten wir die Abhängigkeit des Speicheraufwandes von der Länge des Alphabets $\mathcal{A}$. Das ist sinnvoll, weil die Speicherkomplexität der Algorithmen stark davon abhängt, wie in 2.3.3 beschrieben. In Abb. 5 und Abb. 6 ist dieser Zusammenhang für trinäre Bäume mit $m = k$ und $m = 5 \cdot k$ für $k = 500$ dargestellt:

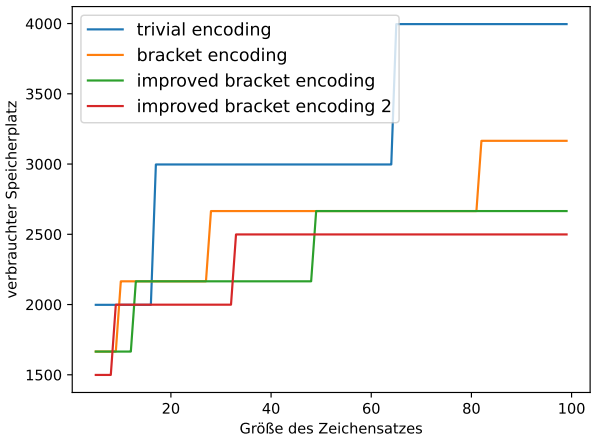

Abbildung 5: Vergleich des Speicheraufwands aller Algorithmen, falls $m = k$.

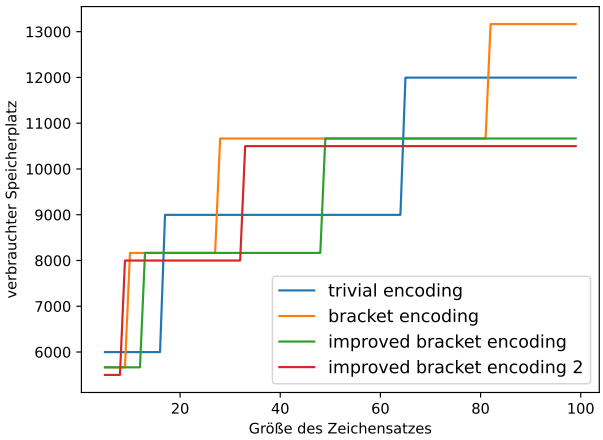

Abbildung 6: Vergleich des Speicheraufwands aller Algorithmen, falls $m = 5 \cdot k$.

Im Fall $m = k$ wechseln sich der verbesserte Klammeransatz und der verbesserte Klammeransatz mit Zusatz immer wieder ab als speichersparendster Ansatz. Die beiden anderen Ansätze sind immer schlechter bis gleich gut.
Der Fall $m = 5 \cdot k$ ist etwas interessanter: Tatsächlich gibt es Zeichensätze $\mathcal{A}$, bei welchen der triviale Kodierungsalgorithmus die beste Speicherkomplexität aufweist - das ist immer der Fall wenn dieser eine kleinere Wortlänge n als die anderen Ansätze hat. Ansonsten wechseln sich wieder der verbesserte Klammeransatz und der verbesserte Klammeransatz mit Zusatz als speichersparendster Algorithmus ab.

3.3 Speicherkomplexität bei anderem Verzweigungsgrad

Bisher haben wir nur trinäre Bäume betrachtet jedoch wäre auch eine Betrachtung von n-ären Bäumen, also Bäumen mit einem anderen Verzweigungsgrad, interessant.
Wie wir bereits in Kapitel 2.5 gesehen haben, steigt der Speicheraufwand aller Funktionen linear mit der Anzahl k an Knoten im Baum. Für einen qualitativen Vergleich reicht es also aus, k auf einen festen Wert wie hier im Beispiel auf 500 zu setzen. Trotzdem ist eine Differenzierung in $m = k$ oder $m = 5 \cdot k$ natürlich weiterhin relevant. In

Abb. 7 und Abb. 8 ist der Speicheraufwand in Abhängigkeit vom Verzweigungsgrad dargestellt:

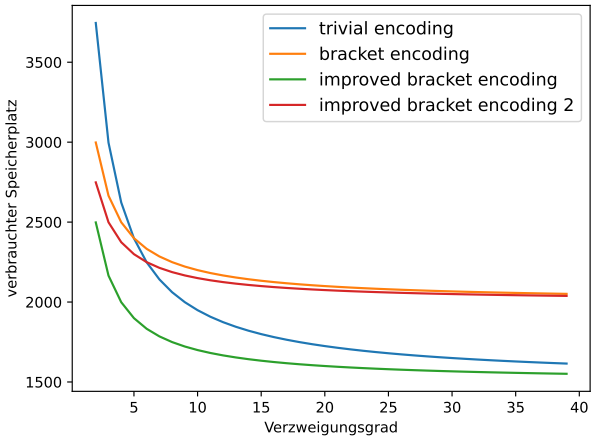

Abbildung 7: Vergleich des Speicheraufwands aller Algorithmen, falls $m = k$.

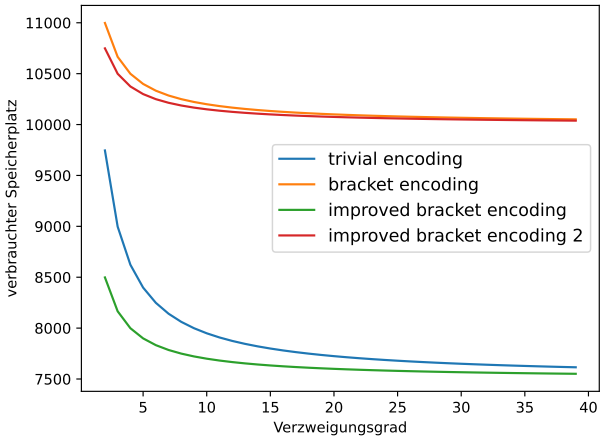

Abbildung 8: Vergleich des Speicheraufwands aller Algorithmen, falls $m = 5 \cdot k$.

Es ist erkennbar, dass der verbesserte Klammeransatz weiterhin bei jeglichen Verzweigungsgraden besser als alle anderen Ansätze ist. Interessant ist jedoch, dass der triviale Ansatz im Fall $m = k$ bei kleinem Verzweigungsgrad schlechter und bei höherem Verzweigungsgrad besser als der Klammeransatz und der verbesserte Klammeransatz mit Zusatz ist. Im Fall $m = 5 \cdot k$ ist das bei jedem Verzweigungsgrad der Fall. Das war auch zu erwarten, da der triviale Ansatz besser wird, je weniger Klammern verwendet werden.

4 Implementierung

Wir haben unsere Algorithmen und die Datenstruktur des trinären Baumes in Python implementiert[1]. Dazu haben wir eine allgemeine n-äre Baum-Klasse *TriTree* erstellt, welche uns das Erstellen und Abspeichern von Baumstrukturen ermöglichte. *TriTree* vererbt ihre Eigenschaften an die Kindklasse *DNA_Encoder*. Diese Klasse hat verschiedene Funktionalitäten: Sie kann den kodierten Baum in Form einer Zeichenkette ausgeben. Außerdem kann ein Baum aus einer solchen Zeichenkette wieder aufgebaut werden. Welches Alphabet genutzt wird, hängt von der genutzten Version ab. Die 4 genutzten Versionen sind: *log*, *bracket*, *bracket_improved* und *bracket_improved*2. Des

[1] https://github.com/Hanno1/MolekulareAlgorithmen

Weiteren haben wir uns gedacht, dass es ganz sinnvoll sein kann, einen nicht kodierten Baum einzulesen. Um diese Funktionalität zu nutzen, muss das Attribut *initial_value* auf die Baum-Zeichenkette gesetzt werden. Beispielsweise kann der Code folgendermaßen genutzt werden:

```
# trinaerer Baum aus initialen String und logarithmisches Alphabet
t1 = DNA_Encoder(version="log",
                 initial_value="a(b(c,d(f,g,h),e),l,i)")
# Kodierung in DNA String
s1 = t1.tree_to_dna()
# Initialisierung des Baums mit DNA String
t2 =  DNA_Encoder(version="log", dna_value=s1)
# Ausgabe des Baums, wie in der Aufgabenstellung beschrieben
print(t2.get_tree_string())
```

Ein ausführlicheres Beispiel befindet sich im Anhang. Außerdem sind weitere Beispiele im Github in der Datei main.py.

5 Zusammenfassung

In diesem Projekt ist es uns gelungen, besonders speichereffiziente Algorithmen zur Kodierung und Dekodierung trinärer Bäume zu realisieren. Für den uns vorgegebenen Zeichensatz, bestehend aus kleinen englischen Buchstaben und Ziffern, erreichen wir mit unserem besten Ansatz - dem verbesserten Klammeransatz - eine Speicherkomplexität von $3 \cdot m + \frac{4}{3} \cdot (k-1)$ für k Knoten und m Buchstaben. Außerdem haben wir gezeigt, wie mit variabler Kodierungslänge experimentiert werden kann.
Weiterhin haben wir die Speicherkomplexität unserer Algorithmen für verschiedene Zeichensatzgrößen und Verzweigungsgrade betrachtet, wobei besonders die Betrachtung der Zeichensatzgrößen interessant für die Wahl des Algorithmus ist.
Alle unserer Algorithmen sind effizient und leicht für den Nutzer zu verwenden.

6 Anhang

```
from DNA_Encoder import DNA_Encoder
from AlphabetFunctions import initialize_alphabet

# initialize Alphabet
# Hier kann das Alphabet veraendert werden.
# Es muss aber mindestens 3 Zeichen geben!
initialize_alphabet(["A", "C", "T", "G"])

# Erstellung des Baums aus der Einleitung
# mit Encoding aus 2.1 (logarithmisch)
# Fuer version gibt es die Moeglichkeiten
# "log" (Kapitel 2.1), "bracket" (Kapitel 2.2),
# "bracket_improved" (Kapitel 2.3.1) und
# "bracket_improved2" (Kapitel 2.3.3)
t1 = DNA_Encoder(version="log",
                 initial_value="a(b(e,f(k,l,m),g),c,d(h,i,j))")
# triviale Kodierung erstellen
e1 = t1.tree_to_dna()
# e1=AATAACAAGAACACTAAAACGAACAGAAAAAGCAAAAGTAAAATAAAA
#    ACAAAAACCAACATCAAAATTAAAATG

# Erstellung des Baums aus dem kodierten String (gleiches Encoding)
t2 = DNA_Encoder(version="log", dna_value=e1)
# Ausgabe des dekodierten Baums
print(t2.get_tree_string())
# Ergebnis: a(b(e,f(k,l,m),g),c,d(h,i,j))

# Erstellung des Baums aus der Einleitung
# mit Encoding aus 2.3.1 (verbesserter Klammeransatz)
t3 = DNA_Encoder(version="bracket_improved",
                 initial_value="a(b(e,f(k,l,m),g),c,d(h,i,j))")
e3 = t3.tree_to_dna()
# e3=AAAGGAACGGACAGACCGGATTGATGGAGAGACTGAATGAAGGGACGGATAGATC
```

```
# e3 ist deutlich kuerzer als e1, obwohl es den gleichen Baum kodiert

# Erstellung des Baums aus dem kodierten String (gleiches Encoding!)
t4 = DNA_Encoder(version="bracket_improved", dna_value=e3)
print(t4.get_tree_string())
# Ergebnis: a(b(e,f(k,l,m),g),c,d(h,i,j))

# Erstellung eines Baums mit Verzweigungsgrad 2 und Encoding 2.3.3
t5 = DNA_Encoder(version="bracket_improved2", branching_degree=2,
                 initial_value="a(b,c(d(e,f),g))")
e5 = t5.tree_to_dna()
# e5=AAAAGAAACTAAATGAAAGGAACATAACCTAACT
t6 = DNA_Encoder(version="bracket_improved2", branching_degree=2,
                 dna_value=e5)
print(t6.get_tree_string())
# Ergebnis: a(b,c(d(e,f),g))
```