FRIEDRICH-SCHILLER-
UNIVERSITÄT
JENA

Kodierung beliebiger endlicher gerichteter Graphen in jeweils eine möglichst kurze DNA-Zeichensequenz und ihre Dekodierung

Ausarbeitung im Fach:

Molekulare Algorithmen

Sommersemester 2021

FRIEDRICH-SCHILLER-UNIVERSITÄT JENA

Fakultät für Mathematik und Informatik

eingereicht von

Maximillian Enderling und Max Möbius

Jena, 25. Juli 2021

Inhaltsverzeichnis

Abbildungsverzeichnis

1 Einleitung

1.1 Aufgabenstellung

Ein beliebiger endlicher gerichteter Graph lässt sich mathematisch durch ein Konstrukt $G = (V, E)$, bestehend aus einer endlichen Knotenmenge V und einer Kantenmenge $E \subseteq V \times V$ beschreiben. Ein Graph mit $|V| = n$ Knoten kann bis zu n^2 Kanten besitzen. Jeder Knoten wird durch eine beliebige eindeutige Zeichenkette wie a oder $knoten5$ symbolisiert.
Ein Beispiel für einen Graphen ist $G = (\{a, b, c\}, \{(a, b), (a, c), (b, b), (b, c), (c, b)\})$.

- Entwickeln und implementieren Sie in Java einen Algorithmus zur Kodierung eines beliebigen gegebenen Graphen G in eine möglichst kurze Zeichensequenz, die ausschließlich die vier Zeichen A, C, G und T verwendet. Knoten- und Kantenmenge sollen in einer gemeinsamen Textdatei durch Aufzählung ihrer Elemente vorgegeben werden. Die Anzahl und Benennung der Knoten ist frei wählbar.
- Entwickeln und implementieren Sie in Java einen Algorithmus zur Dekodierung einer DNA-Zeichensequenz in die einzelnen Elemente der Knoten- und Kantenmenge des zugrunde gelegten Graphen G.
- Welcher Zusammenhang besteht zwischen Anzahl Knoten, Anzahl Kanten und der Länge der DNA-Zeichensequenz?

1.2 Erklärung und Schlussfolgerung der Aufgabenstellung

1.2.1 Was bedeutet es einen Graphen zu kodieren?

Bei der Kodierung bringt man einen Graphen in eine andere Form, ohne dass dabei Information verloren geht. Das wird meist genutzt, um die Information des Graphen mit geringem Platzverbrauch zu speichern und/oder transportieren zu können. Wichtig hierbei ist, dass Knotenbezeichnungen (z.B. a, 2 oder $knoten3$) nicht erhalten bleiben müssen. Andererseits ist es wichtig, dass, wenn ein Graph kodiert und wieder dekodiert wird, das Ergebnis die selbe Struktur wie der Anfangsgraph haben muss. Die in der Aufgabenstellung erwähnten Graphen sind ist deshalb ebenfalls kodiert.

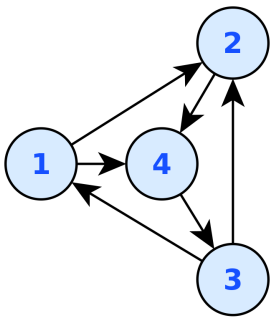


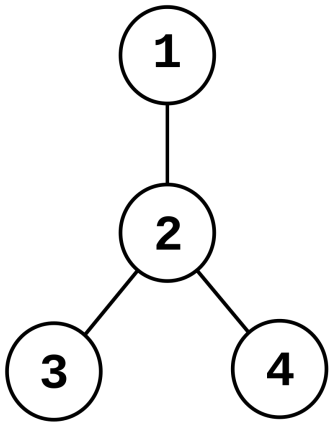
Abbildung 1: Beispiel eines gerichteten Graphen

Quelle: https://de.wikipedia.org/wiki/Gerichteter_Graph

Da Graphen ein abstraktes Konstrukt sind, gibt es keine natürliche Darstellung. Man kann sie zum Beispiel grafischer darstellen (siehe Abbildung [1]), wobei die Knoten durch die Kanten miteinander verbunden sind. Es ist aber auch möglich, dass Knoten keine Verbindung haben.

1.2.2 Welche Methoden zur Kodierung von Graphen gibt es?

Die am häufigsten verwendeten Kodierungsmethoden sind **Adjazenzmatrizen** und **Adjazenzlisten**. Beide Varianten sind Datenstrukturen, die auch in Form eines Strings dargestellt werden können.



(a) ungerichteter Graph

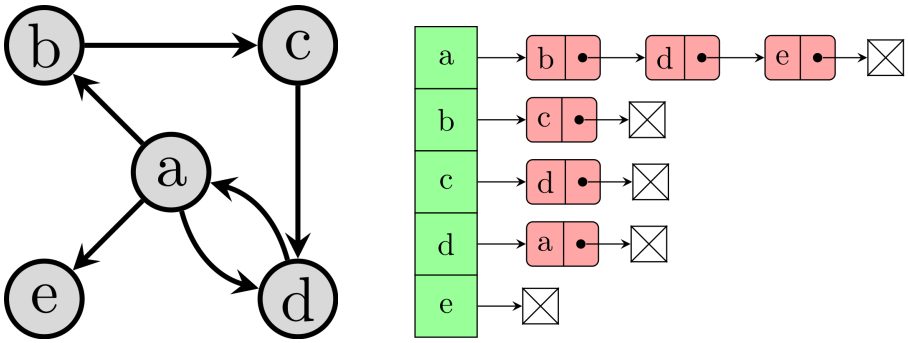
	1	2	3	4
1	0	1	0	0
2	1	0	1	1
3	0	1	0	0
4	0	1	0	0

(b) Adjazenzmatrix

Abbildung 2: Graph als Adjazenzmatrix kodieren

Quelle: https://de.wikipedia.org/wiki/Adjazenzmatrix

Sei V die Knotenmenge des Graphen. Adjazenzmatrizen sind Matrizen der Größe $|V| \times |V|$. Wie in Abbildung [2] zu sehen besitzt jeder Knoten eines Graphens in der Adjazenzmatrix eine Spalte und eine Zeile. Die Werte in der Matrix geben an, welche Knoten eine Kanten besitzen. Kanten werden durch Einsen markiert. In diesem Beispiel wird nur ein ungerichteter Graph gezeigt. Bei gerichteten Graphen funktioniert die Kodierung genau so, die Kanten verlaufen jedoch immer vom Knoten der Spalte zum Knoten der Kante (müssen also auch so gelesen werden) [1, S. 589-593].



(a) gerichteter Graph

(b) Adjazenzliste

Abbildung 3: Graph als Adjazenzliste kodieren

Quelle: https://de.wikipedia.org/wiki/Adjazenzliste

Eine Adjazenzliste (siehe Abbildung [3]) ist eine Liste, die für einen Knoten eines Graphens alle seine Nachbarn (alle Knoten mit denen ein Knoten über eine Kante verbunden ist) enthält. Ein Graph besitzt somit $|V|$ viele Adjazenzliste, um für jeden Knoten seine Nachbarn zu speichern. Meist wird dazu ein Array mit einfach verketteten Listen verwendet. Diese Listen als String darzustellen ist im Beispiel in der Aufgabenstellung gezeigt; dort werden erst die Knoten aufgelistet und dann die Kanten anhand der Bezeichnungen zweier Knoten dargestellt [1, S. 589-593].

2 Konzept

2.1 Konzeptliste

Aus der Aufgabenstellung entnehmen wir die direkten Anforderungen:

K.1 Ein Graph als String soll in eine Liste von Knoten und in eine Liste von Kanten umgewandelt werden können.

K.2 Kodierung eines Graphens mithilfe der Zeichen A, C, G und T zu einer möglichst kurzen DNA-Zeichensequenz.

K.3 Dekodierung einer DNA-Zeichensequenz zu Knoten und Kanten.

K.4 Der Zusammenhang zwischen der Anzahl der Knoten, Anzahl der Kanten und der Länge der DNA-Zeichensequenz muss gemessen werden. Wir verwenden dafür die Einheit DNA-Bits „dbits“, wobei eine Base genau einem dbit entspricht.

Wir spezifizieren K.1 noch weiter, indem wir die gespeicherte Form von Graphen konkretisieren:

K.5 Die Knoten eines Graphen mit n haben genau die interne Darstellung $\{0, ..., n-1\}$.

Aus K.2 und K.3 ergibt sich weiterhin die Anforderung:

K.6 Die Kodierung soll die Information eines Graphens erhalten

Wichtig sind uns hier aber nicht die Bezeichnungen der Knoten, stattdessen nutzen wir die Eigenschaft der Graphen-Isomorphie:

> Two graphs $G = (V, E)$ and $G' = (V', E')$ are isomorphic if there exists a bijection $f : V \to V'$ such that $(u, v) \in E$ if and only if $(f(u), f(v)) \in E'$ In other words, we can relabel the vertices of G to be vertices of G', maintaining the corresponding edges in G and G' [1, S. 1171].

Darauf basierend stellen wir die Anforderung:

K.7 Eine Kodierungsfunktion K und eine dazugehörige Dekodierungsfunktion D sollen die Isomorphie erhalten, insbesondere sind beliebige Graphen G1 und G2 genau dann isomorph, wenn auch $D(K(G1))$ und G2 isomorph sind.

Weiterhin besteht ein Spezialfall von K. 7, wenn $G1 = G2$ ist: $D(K(G))$ muss zu G isomorph sein. Damit wir das überprüfen können, stellen wir auf:

K.8 Man muss die Isomorphie zwischen zwei Graphen testen können.

Die Knoten und Kanten sind als Menge definiert, haben also keine feste Reihenfolge. Trotzdem haben wir uns entschieden

K.9 Man soll entscheiden können, ob die Reihenfolge der Knoten erhalten bleiben soll

zu fordern. Die Reihenfolge bezieht sich hier auf die natürlichen Zahlen die wir laut K.5 zur Verarbeitung nutzen.

2.2 Graph-Kodierung

In diesem Abschnitt werden verschiedene Algorithmen dargestellt, um Graphen zu DNA-Zeichensequenzen zu kodieren und dekodieren.

2.2.1 Summen-Code

Eine Möglichkeit zur Kodierung eines Graphen in der Form von Adjazenzlisten ist der **Summen-Code**. Dabei handelt es sich um eine Kodierung eines Graphen, die auf einfachen Regeln beruht:

- A, C und G sind Kodierungen für Zahlen $\rightarrow 1 = A, 2 = C$ und $5 = G$
- T ist das Trennzeichen (wird also bei der eigentlichen Kodierung nicht verwendet)

Grundidee
Wenn die Reihenfolge relevant ist bzw. beibehalten werden soll, wird jeder Knoten wie folgt kodiert: für jeden Knoten wird gezählt, wie oft die folgenden Zahlen in den Knotenindex passen.

1. Zuerst aufsummiert die 5.
2. Dann die 2.
3. Dann die 1.

Wichtig ist hierbei, die Summe beginnend mit der größten Zahl zu bilden und dann kleiner zu werden, um die Summe mit den wenigsten Zahlen, bestehend aus den drei Zahlen, zu bilden. Beispiele: $5 \rightarrow 5 \rightarrow GC, 9 \rightarrow 5 + 2 + 2 \rightarrow GCC, 13 \rightarrow 5 + 5 + 2 + 1 \rightarrow GGCA$.
Je nach Wertebereich können die drei Zahlen natürlich angepasst werden, um optimal zu kodieren. Wenn man z.B. weiß, dass sich in einem Graphen immer einhundert Knoten befinden und diese dann alle gleich wahrscheinlich sind, dann ist es sinnvoll statt 1, 2 und 5 die Zahlen 1, 5 und 22 (oder 23) zu wählen (siehe [5]).

Ist nun die Reihenfolge irrelevant, müssen Knoten, die keine Kanten zu anderen Knoten haben, nur gezählt und nicht direkt kodiert werden. Außerdem ist die Anzahl der Knoten ohne Kanten wichtig, um die Gesamtzahl der Knoten zu kennen und um diese in der Liste der Knoten bei der Dekodierung wieder eintragen zu können. Bei irrelevanter Reihenfolge fallen die Knoten ohne Kanten weg, wodurch die anderen Knoten ihre Bezeichnungen ändern, je nachdem wie weit sie in der Liste nach vorne rücken. Nun werden alle Knoten in der Liste zu den erwähnten Zeichensequenzen kodiert. Für die Abbildung: Sei $K_{a,b}$ der b. Knoten der Kante a, also Kantenmenge = $\{(K_{0,1}, K_{0,2}), (K_{1,1}, K_{1,2}), ...\}$.

Anzahl der Knoten	Trenner	Kantenliste								
$\lvert V\rvert$	'T'	$K_{0,1}$	'T'	$K_{0,1}$	'T'	$K_{1,1}$	'T'	$K_{1,2}$	'T'	$\cdots$
$\frac{\lvert V\rvert}{5}$ **dbits**	**1 dbit**	**? dbits**	**1 dbit**	**? dbits**	**1 dbit**	**? dbits**	**1 dbit**	**? dbits**	**1 dbit**	$\cdots$

Abbildung 4: Summen-Code

Aufbau der DNA-Zeichensequenz

Wenn für jeden Knoten in der Liste eine Zeichensequenz gebildet wurde, kann die DNA-Zeichensequenz erstellt werden. Es wird zuerst die Zeichensequenz des letzten Knotens, also mit der höchsten Zahl, gespeichert, da dieser vorgibt wie viele Knoten im Graphen sind. Dann folgt ein Trennzeichen T, woraufhin die Zeichensequenzen der Knoten aus den Kanten hintereinander in die DNA-Zeichensequenz eingetragen werden, wobei ein T zwei Knoten voneinander trennt.

Dekodierung der DNA-Zeichensequenz
Die Zeichensequenz bis zum ersten T gibt die Gesamtanzahl der Knoten des Graphens an. Die Zahlen 1 bis Gesamtanzahl werden zu Zeichensequenzen kodiert (und in die Liste der Knoten gespeichert), wie in der Grundidee erklärt, und invertiert genutzt, um die DNA-Zeichensequenz zu dekodieren. Beim restlichen String wird nun jede Zeichensequenz zwischen den Trennzeichen T zu Zahlen dekodiert. Von Anfang an zählend ergeben zwei Zahlen immer eine Kanten und werden als diese auch in der Liste der Kanten gespeichert.

Wurde beim Kodieren und Dekodieren die Reihenfolge beachtet, liegt der Graph nun in der selben Form vor, in der er eingegeben wurde. Ist die Reihenfolge jedoch irrelevant gewesen, verändern sich die Knotenbezeichnungen je nachdem, wie viele Knoten keine Kanten hatten. Die Graphen bleiben dabei immer isomorph.

Länge der DNA-Zeichensequenz
Seien V die Knoten und E die Kanten. Wir schätzen den worst case grob ab: Die Anzahl der Knoten benötigt, wie in Abbildung 4 angegeben, $\frac{|V|}{5}$ dbits. Außerdem benötigt jedes K auch maximal $\frac{|V|}{5}$ dbits. Mit den Trennzeichen zusammen benötigt man also, im worst case, $\frac{|V|}{5} + 2(|E| \cdot \frac{|V|}{5} + 1)$ dbits.

2.2.2 Fixe-Länge-Code

Grundidee
Eine weitere Möglichkeit, einen Code auf Basis von Adjazenzlisten zu erstellen, ist der Fixe-Längen-Code. Die Kantenliste wird dabei, wie beim Summen-Code, gespeichert, indem beide Knoten einzeln kodiert werden. Für die Kodierung eines einzelnen Knotens berechnen wir zuerst den Knotenindex in Basis 4: $(12)_{10} = (30)_4$. Dann können wir die Bezeichnungen für die vier verfügbaren Basen nutzen, um diese Zahl als DNA-Sequenz zu speichern, indem wir z.B. einfach „0“ durch „A“, „1“ durch „C“, „2“ durch „G“ und „3“ durch „T“ ersetzen. Der Knoten mit Index 12 wird dementsprechend durch „TA“ repräsentiert. Dadurch, dass wir eine bestimmte Breite verwenden, können wir die Knoten ohne Trennzeichen hintereinander schreiben.

Mit Erhaltung der Reihenfolge
Sei $K_{a,b}$ der b. Knoten der Kante a, also Kantenmenge= $\{(K_{0,1}, K_{0,2}), (K_{1,1}, K_{1,2}), ...\}$. Zuerst halten wir die feste Länge k der Knoten in der Kantenliste fest. Sei $g = \max_{a>0, b\in 1,2} K_{a,b}$, also der letzte in einer Kante vorkommende Knoten. Dann ist $k =$ die Anzahl der dbits, die man zum Kodieren von g benötigt. Wegen K.5 ist klar, dass wir alle Knoten x mit $0 \leq x \leq g$ benutzen. Die Anzahl der Knoten nach g schreiben wir einfach in l. Damit wir wissen, wo l beginnt, setzen wir davor einen Trenner $k \times' A'$. Damit wir den Trenner später auch als solchen erkennen, müssen wir bei der Kodierung der Knoten in der Kantenliste darauf achten, dass wir diese erst bei 1 beginnen lassen. Auf diese Weise wird ein 0-Knoten in der Kantenliste illegal und kann als Trenner benutzt werden. Das beschriebene Format entspricht der Abbildung 5

Länge	Trenner	Kantenliste					Trenner	l
$k\times$'A'	'C'	$K_{0,1}$	$K_{0,2}$	$K_{1,1}$	$K_{1,2}$	$\cdots$	$k\times$'A'	l
k dbits	**1 dbit**	**k dbits**	**k dbits**	**k dbits**	**k dbits**	$\cdots$	**k dbits**	$\log_4 l$ **dbits**

Abbildung 5: Fixe-Länge-Code mit Ordnungserhaltung

Ohne Erhaltung der Reihenfolge
Auch wenn die Reihenfolge keine Rolle spielt, funktioniert der Code ganz ähnlich wie mit Reihenfolgenerhaltung. Jetzt können wir die Bezeichnungen für die Knoten aber umordnen. Betrachten wir das Beispiel $G = (\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10\}, \{(9, 10)\})$. Auch wenn wir nur die Knoten 9 und 10 in der Kantenliste kodieren müssten, wäre ihre Bezeichnung zwei Zeichen lang: $9 = GC$ und $10 = GG$ (hier wurde noch nicht $+1$ gerechnet, um die Knotenkodierung $k \times' A'$ auszuschließen). Wenn wir die verwendeten Knoten nach vorn sortieren, können wir diese möglicherweise mit weniger Basen speichern. Den Beispielgraphen können wir umschreiben zu $S = (\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10\}, \{(0, 1)\})$, dadurch haben wir ein kleineres k und benötigen weniger Zeichen für die Kantenliste. Sei also $S = K$ mit unbenutzten Knotenbezeichnungen nach hinten sortiert, $g = \max_{a>0,b\in 1,2} S_{a,b}$ und $k =$ die Anzahl der dbits, die man zum Kodieren von g benötigt. Die Anzahl der nicht benötigten Knoten, also derer die nach hinten sortiert wurden, wird in u gespeichert.

Länge	Trenner	Kantenliste					Trenner	u
$k\times$'A'	'C'	$S_{0,1}$	$S_{0,2}$	$S_{1,1}$	$S_{1,2}$	$\cdots$	$k\times$'A'	u
k dbits	**1 dbit**	**k dbits**	**k dbits**	**k dbits**	**k dbits**	$\cdots$	**k dbits**	$\log_4 u$ **dbits**

Abbildung 6: Fixe-Länge-Code ohne Ordnungserhaltung

Länge der DNA-Zeichensequenz
Seien V die Knoten und E die Kanten. Wir schätzen den worst case grob ab: Sei $k = \lceil \log_4 |V| \rceil$ die Kodierung der Länge, dann benötigt k dbits + 1 für den Trenner. Jedes K bzw S benötigt dann k dbits. Der zweite Trenner benötigt wieder k dbits. l und u sind beide $\leq$ als die Anzahl der nicht verwendeten Knoten, sind also nach oben durch $|V|$ beschränkt. Insgesamt haben wir damit im worst case $k + 2 \cdot |E| \cdot k + |V|$

2.2.3 Huffman-Code

Der Huffman-Code ist eine weitere Möglichkeit, einen Graphen auf Basis der Adjazenzlisten-Form darzustellen. Der originale Huffman-Code hat zwei für uns wichtige Eigenschaften:

1. Er ist ein Präfixcode: kein gültiges Codewort ist der Beginn eines anderen Codewortes.
2. Häufig benutzte Symbole (in unserem Fall Knoten) bekommen kürzere Kodierungen als weniger häufig genutzte.
3. Es gibt keinen kürzeren Präfix-Code.

Den originalen Code, den Huffman entworfen hatte, erzeugen wir für unseren Fall, indem ein Baum mit maximalem Verzweigungsgrad 4 (wegen 4 Basen) erzeugt wird, in welchem häufig genutzte Knoten eine kleinere Tiefe haben.[3] Daraus wird dann direkt das sogenannte Kodierungsbuch erstellt, in dem die konkreten Übersetzungen von Knotenbezeichnung zu Basensequenz festgehalten werden. Damit wir aber nicht für jeden Knoten auch noch dessen Häufigkeit (die prinzipiell beliebig groß werden kann) mit kodieren müssen, setzen wir hier den kanonischen Huffman-Code ein. Dieser kann, nur aus den Tiefen der Knoten im Huffman-Baum, das Kodierungsbuch erstellen. (Vgl. [4])

Mit Erhaltung der Reihenfolge
Wenn wir die Reihenfolge erhalten sollen, ist es wichtig die genaue Zuordnung der Knoten zu ihrer Baumtiefe mit zu kodieren. Sei D_i die Anzahl der Knoten mit Tiefe i im Huffman-Baum. Dann können wir z.B. wie in Abbildung 7 erst die Länge a jedes Zeichens, also die maximal benötigte Länge, kodieren. Anschließend wird ein Trennzeichen gesetzt, daraufhin folgen alle D_i, nach deren Verwendung erneut ein Trennzeichen eingefügt wird. Wenn wir für die Kodierung der D_i und von a das Zeichen T nicht verwenden, also diese nur mit der Basis 3 darstellen, dann können wir 'T' als Trennzeichen nutzen. Nachdem wir die Tiefe für jeden Knoten kodiert haben, also genug Information zur Konstruktion des dazugehörigen kanonischen Huffman-Codes zur Verfügung gestellt, können wir die Kanten wie in Abbildung 5 Fixe-Längen-Code alle hintereinander notieren. Wir definieren S also wie dort. Die Kodierung von S nehmen wir diesmal aber aus dem kanonischen Huffman-Code, also haben die Knoten keine fixe Länge.

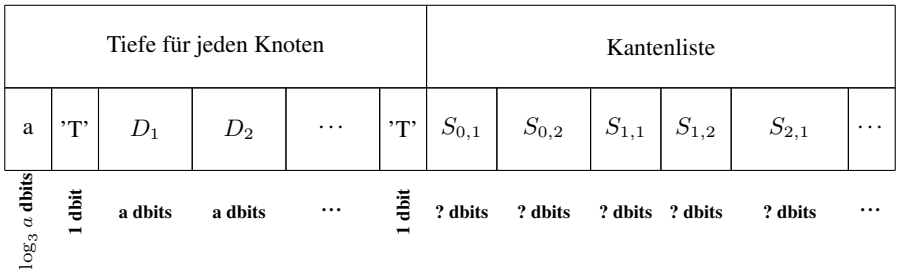


Abbildung 7: Huffman-Code mit Ordnungserhaltung

Ohne Erhaltung der Reihenfolge
Wenn die Reihenfolge keine Rolle spielt, können wir die Tiefen auch anders darstellen: Wir kodieren nur die Anzahl der Knoten pro Baumtiefe. Sei C_i = die Anzahl der Knoten im Huffman-Baum mit Tiefe i und $\sum_i C_i$ = Anzahl der Knoten im Graphen. Typischerweise ist deshalb diese Kodierung der Tiefen kürzer als die Kodierung mit Erhaltung der Reihenfolge, trotzdem entstehen ohne die selben Kodierungsschlüsselwörter wie mit, wobei die Knotenbezeichnungen untereinander aber vertauscht sein können. Der Marker am Anfang zeigt an, dass der Graph ohne Erhaltung der Reihenfolge kodiert wurde.

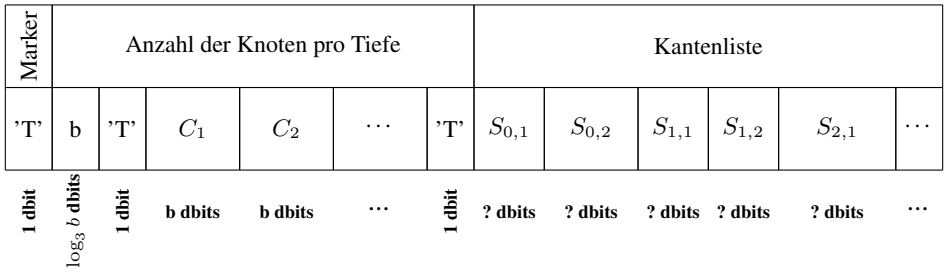


Abbildung 8: Huffman-Code ohne Ordnungserhaltung

Länge der DNA-Zeichensequenz

Seien V die Knoten und E die Kanten. Wir schätzen den worst case grob ab: Die Tiefe für jeden Knoten zu kodieren erzeugt eine längere DNA Sequenz als die Anzahl der Knoten pro Tiefe. Die maximale Tiefe eines Huffman-Baums ist $< |V|$. Also haben wir eine obere Schranke $|V|^2$ für die Kodierung der Baumtiefen in beiden Varianten. Die Kodierung für jedes S benötigt maximal $|V|$ (= maximiale Tiefe des Huffman-Baums) dbits. Insgesamt haben wir im worst case also $|V|^2 + 2 \cdot |E| \cdot |V|$ dbits, was zunächst schlechter ist als der Fixe-Längen-Code aus Abschnitt 2.2.2.

3 Implementierung

Die Implementierung zur Lösung der gegebenen Aufgabenstellung liegt als eigenständiges Java-Paket `de.unijena.DNAGraphUtils` vor und kann deshalb einfach in ein Projekt eingebunden werden.

3.1 Umsetzung der Konzeptliste

Das zentrale Element des Paketes ist die Klasse **Graph**, die Graphen in einem internen Format speichert. Ein Objekt der Klasse Graph enthält dabei eine Liste der Knoten eines Graphen, sowie eine Liste der Kanten, die in dem Graphen bestehen (entsprechend K.5). Neue Instanzen der Klasse kann man auch aus Strings in unterstützten Formen oder Kodierungen laden. Darunter zählt insbesondere die form, die in der Aufgabenstellung vorkommt (z.B. $G = (\{a, b, c\}, \{(a, b), (a, c)\})$). Wir nennen das die „natürliche Form“ und implementieren diese in der Klasse `NaturalGraphEncoding`, von der eine Instanz mit übergeben werden muss, um ein String aus der Form in ein Graphen-Objekt umzuwandeln (erfüllt K.2). Mit der Methode `Graph::toString()` können Graphen-Objekte auch wieder in Strings umgewandelt werden (erfüllt K.3).
Wie in K.8 verlangt, stellt das Paket die Funktion `Graph::isIsomorphicTo(Graph other)` zur Verfügung.

3.2 Implementierung des Codes

Die bereits erwähnte Klasse `NaturalGraphEncoding` ist eine Implementation der Schnittstelle `GraphEncoding`. Dieses Interface beschreibt alle benötigten Methoden, um weitere Kodierungen einzufügen:

- `String GraphEncoding::toString(Graph g, boolean preserveOrder)` ist zum Kodieren eines Graphens `g` aus dem internen Format in eine Zeichenkette (nach K. 2) bei optionaler Ordnungserhaltung (durch `preserveOrder` angezeigt, nach K.9)
- `void GraphEncoding::load(Graph g, String repr)` dekodiert den in der Zeichenkette `repr` kodierten Graphen und speichert den Wert in den gegebenen Graphen `g`. (erfüllt K.3)

Bei der Implementierung beider Funktionen muss darauf geachtet werden K.6 und K.7 einzuhalten.

3.2.1 Summen-Code

Die Implementierung beruht auf der Idee für den Code, welche schon in Abschnitt 2.2.1 erklärt wurde. Hierbei wird nicht wie nach K. 5 mit der internen Darstellung gearbeitet, sondern mit $1, ..., n$, da der Code nur mit Zahlen beginnend bei 1 rechnen kann. Bei der Kodierung werden so die Bezeichnungen alle um 1 erhöht und beim Dekodieren vor der Rückgabe wieder um 1 verringert. Es wäre auch möglich mit 0 zu starten; dafür wird die Zahl einfach als leere Zeichensequenz kodiert und tritt auf, wenn zwei Trennzeichen hintereinander auftauchen. Dies könnte man als Optimierung einfügen, da die generell DNA-Zeichensequenz verkürzt wird.

Kodierung des Graphen
Wenn `preserveOrder` wahr ist, wird `convertToDNASequence(graph)` ausgeführt, ansonsten `convertToDNASequenceWithoutUnusedVerts(graph)`. Beide Funktionen sind grundsätzlich gleich aufgebaut und sollen das eingebende Graphen-Objekt kodieren. Dazu wird für jeden Knotenindex die zugehörige Zeichensequenz bestimmt und beides wird in eine `Map`, die als Kodierungsbuch dient, eingetragen. Dann wird die DNA-Zeichensequenz aufgebaut, wobei hier die `Map` zum Umwandeln der Zahlen dient. Bei `convertToDNASequenceWithoutUnusedVerts(graph)` werden lediglich, vor dem Aufbau der `Map`, alle Knoten ohne Kanten aussortiert und die Knotenbezeichnungen dementsprechend angepasst. Am Ende wird die DNA-Zeichensequenz als String zurückgegeben.

Dekodierung des Graphen
Zuerst wird die DNA-Zeichensequenz bei den Trennzeichen aufgeteilt. Die erste Sequenz wird ausgelesen, um die Anzahl an Knoten im Graphen zu bestimmen. Dann wird wieder die `Map` anhand der Knotenanzahl erstellt. Die `Map` muss aber invertiert werden, damit Zeichensequenzen zu Zahlen umgewandelt werden können (und nicht andersherum). Nun werden alle Zahlen zwischen 1 bis zur Knotenanzahl in die Liste der Knoten des neuen Graphen-Objekts eingetragen. Außerdem werden die Sequenzen im String mit der `Map` zu Zahlen übersetzt und wie im Konzept erklärt als Kanten des neuen Graphen-Objekts eingetragen.

3.2.2 Fixe-Länge-Code

Die Implementierung des Codes in der Klasse `FixedLengtGraphEncoding`, der in Abschnitt 2.2.2 beschrieben wird, folgt direkt den dort definierten Bausteinen.

3.2.3 Huffman-Code

Der Code, der in Abschnitt 2.2.3 beschrieben wird, wird in der der Klasse `HuffmanGraphEncoding` implementiert. Dinge, die im Konzept nicht konkret beschrieben wurden, sind der Aufbau des Huffman-Baums und die Erstellung des kanonischen Huffman-Codes aus den Knotentiefen. Dazu hier die beiden relevanten Ausschnitte:

```
private static Map<Integer, String>
    generateVertMap(ArrayList<TreeSet<Integer>> depthToNodes){
      Map<Integer, String> decodingMap = new HashMap<>();
      int code = 0;
      for (TreeSet<Integer> depthToNode : depthToNodes) {
         for (var node : depthToNode) {
            decodingMap.put(node, toDNA(code, 4));
            code += 1;
         }
         code *= 4;
      }
      return decodingMap;
   }
```

Der Eingabeparameter `depthToNodes` enthält für jede Baumtiefe die Menge von darin liegenden Knoten. `toDNA` ist eine Hilfsfunktion, die jede beliebige Zahl zuerst in eine beliebige Basis (hier 4) und dann in DNA-Basen umwandelt. Der Algorithmus stammt direkt aus [4].

```
private static TreeNode
    generateHuffman4aryTree(ArrayList<ComparablePair<Integer, Integer>>
    frequencies) {
```

```
        var probabilityList = new PriorityQueue<TreeNode>();
        for (ComparablePair<Integer, Integer> probability : frequencies) {
            probabilityList.add(new TreeNode(probability.getV1(),
                probability.getV2()));
        }

        int usableSymbols = 4; // since we construct a 4-ary tree
        var initial_count = probabilityList.size() == 1 ? 1 : 2 +
            (probabilityList.size() - 2) % (usableSymbols - 1);
        mergeLeastFrequentHuffmanNodes(probabilityList, initial_count);
        while (probabilityList.size() != 1) {
            mergeLeastFrequentHuffmanNodes(probabilityList, usableSymbols);
        }

        return probabilityList.poll();
    }
```

Der Eingabeparameter `depthToNodes` enthält an der Position i die Anzahl der Vorkommen (=Frequenz) des Knotens i in der Kantenliste. So bekommen Knoten, die häufiger in Kanten vorkommen, kürzere Codes. `mergeLeastFrequentHuffmanNodes` kombiniert die vier Knoten mit den geringsten Frequenzen unter einem neuen Elternknoten und fügt diesen wieder in die `probabiltyList` ein. Der Code implementiert den Algorithmus aus [3].

3.2.4 Isomorphie-Implementierung

Ob zwei Graphen `g1` und `g2` isomorph sind, ist bei Reihenfolgenerhaltung trivial, da wir dann direkt die `Graph::toString()` Methode verwenden können. Ansonsten kann sich die Bezeichnung der Knoten ändern. Die Implementierung dient also dazu herauszufinden, ob die Graphen `g1` und `g2` isomorph sind. Die Grundidee dabei ist, zu prüfen, ob die Kanten die selben Bezeichnungen haben. Wenn dies nicht der Fall ist, werden Übersetzungen für die Knoten des Graphen `g2` gesucht, die die Knotenbezeichnungen so übersetzt, dass sie identisch zu denen in `g1` werden; wir suchen also die in der Graphen-Isomorphie-Definition beschriebene Bijektion f. Anhand der Häufigkeit der Knoten in den Kanten von `g1` und `g2` finden wir heraus, welche Knotenbezeichnungen überhaupt ineinander übersetzt werden könnten. Wir erstellen für jeden Knoten aus `g2` eine Liste an möglichen Übersetzungen. Nun werden die Werte aus den Listen so miteinander kombiniert, dass mehrere Listen entstehen. Jede Liste muss dabei für jeden Knoten in `g2` eine Übersetzung beinhalten. Aussortiert werden dabei Kombinationen, in denen die selbe Bezeichnung mehrmals vor kommt. Diese Kombinationen werden nun genutzt, um die Bezeichnungen in `g2` zu übersetzen. Für jede Übersetzung wird geprüft, ob dadurch die Kantenmenge aus `g1` und `g2` äquivalent sind. Wenn eine entsprechende Kombination gefunden wurde, sind `g1` und `g2` isomorph.

Problem
Dieser Algorithmus besitzt keine gute Laufzeit. Für eine Knotenanzahl $n \in \mathbb{N}$ gibt es im schlimmsten Fall (alle Knoten kommen gleich oft in Kanten vor) $(n-1)^n$ viele Kombinationen. Bei vier Knoten sind es nur 81 Kombination, die maximal getestet werden; bei zehn Knoten sind es schon $3\,486\,784\,401$ Kombinationen. Diese Zahl steigt exponentiell. Es ist keine andere Funktion implementiert, die Graphen mit hoher Knotenanzahl auf Isomorphie testet. Man könnte aber etwa den Algorithmus aus [2] nachrüsten. Bei zu großen Graphen bricht die Implementierung deshalb auch ab.

4 Benchmark

Im Paket wird die Klasse `GraphBenchmark` mitgeliefert, in der zwei Funktionen definiert sind. Mit `testIsomorphismPreservation(GraphEncoding code, Random rand)` wird ein GraphEncoding auf Isomorphieerhaltung überprüft. Der Zufallsgenerator `rand` kann, muss aber nicht, mitgegeben werden (darf auch `null` sein). Mit der Funktion `sampleEncodings(GraphEncoding[] encodings, int maxVerticesNumber, Random rand)` werden für zufällige Graphen die DNA-Zeichensequenzen für alle mitgegebenen Kodierungen erzeugt. Dabei werden für alle $0 < i <$ `maxVerticesNumber` Graphen mit i Knoten und $\frac{i^2}{4}$ Kanten erzeugt, wobei bei der Kantenerzeugung die Knoten normalverteilt ausgewählt wurden.

Im Folgenden sind Grafiken zum Benchmark-Test der drei implementierten GraphEncodings dargestellt. Es ist zu erwähnen, dass in der Legende die Name für die jeweiligen Verfahren stehen. + steht dabei für das Einhalten der Reihenfolge und bei − ist die Reihenfolge irrelevant.

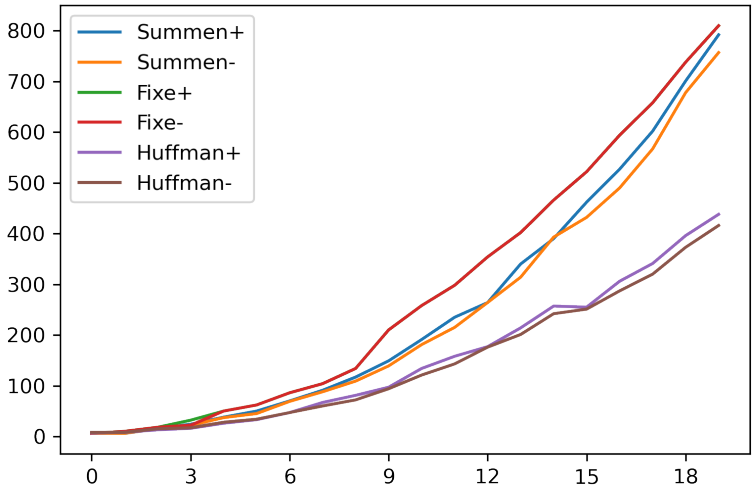


Abbildung 9: Länge der Kodierung von zufälligen Graphen bis zu 20 Knoten

In der Abbildung 9 werden die implementierten GraphEncodings auf die Länge ihrer kodierten DNA Sequenzen überprüft. Bei allen Verfahren ist zu erkennen, dass die Beachtung der Reihenfolge keinen wirkliche Unterschied macht. Bei Graphen mit bis zu drei Knoten sind eigentlich noch keine Unterschiede zwischen den Verfahren zu erkennen; dort werden jeweils weniger als 30 Zeichen benötigt. Danach zeichnet sich Huffman deutlich ab und kommt maximal knapp über 400 Zeichen. Das Summen- und Fixe-Länge-Verfahren steigen dahingegen stark an und landen jeweils ca. bei einer doppelten Anzahl von rund 750 bis 800 Zeichen. Bei Summen zeichnet sich außerdem dahingegen ein Trend ab, dass Graphen mit mehr Knoten sogar noch schneller steigen als bei Fixe.

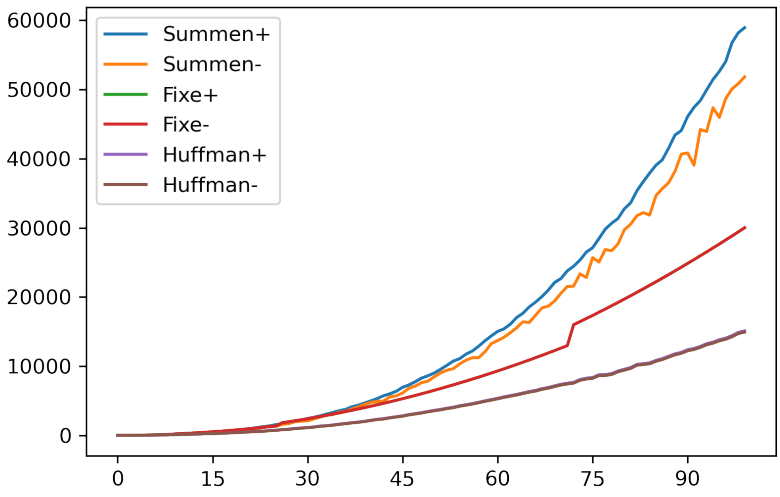


Abbildung 10: Länge der Kodierung von zufälligen Graphen bis zu 100 Knoten

Abbildung 10 erweitert Abbildung 9. Huffman hat noch immer die merkbar kleinste Anzahl an Zeichen und steigt, im Gegensatz zu den anderen Verfahren, auch nur minimal weiter an. Fixe hat nach den Anzeichen eines steilen Anstiegs ein doch eher langsames Wachstum im Hinblick auf die Zeichenanzahl. Das Verfahren hat bei 100 Knoten mit rund 300.000 Zeichen noch immer drei mal so viel, wie die 100.000 Zeichen bei Huffman. Summen scheint den Trend mit dem steilen Anstieg fortzusetzen; anscheinend steigt die Anzahl der Zeichen mit den Knoten in einem Graphen über-linear weiter. Das Verfahren mit irrelevanter Reihenfolge zeigt scheint sogar noch kürzere Zeichenketten zu generieren. Mit Reihenfolgenerhaltung liegt die Zeichenanzahl bei knapp 600.000 und bei irrelevanter Reihenfolge bei knapp über 500.000 (also beachtlich weniger). Die Frage ist, ob die beiden Verfahren weiterhin eine signifikant divergierende Anzahl an Zeichen bei der Kodierung verwenden.

Interessant ist es auch, die Shannon-Entropie [6] der entstandenen Kodierung zu untersuchen. Diese schätzt den Informationsgehalt pro Zeichen und wird in $\frac{bits}{Symbol}$ angegeben. Wir arbeiten mit vier Zeichen ('A','C','G','T') und können deshalb auch maximal 2 Bits pro Symbol erreichen. Eine hohe Entropie steht nicht notwendigerweise für eine gute Kodierung, da man durch unnötiges Hinzufügen von weißem Rauschen auch eine hohe Entropie erreichen könnte. Eine niedrige Entropie deutet aber immer auf eine geringe Informationsdichte, und damit eine suboptimale Kodierung hin.

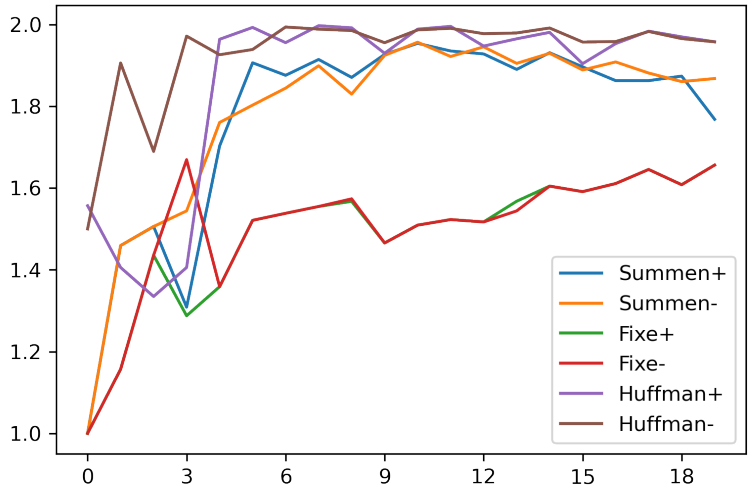


Abbildung 11: Entropie der Kodierung von zufälligen Graphen bis zu 20 Knoten

In Abbildung 11 kann man für kleine Graphen interessante Informationen ablesen: Die Huffman-Kodierung hat, unabhängig von der Ordnungserhaltung, eine sehr gute Entropie nahe 2. Auch die Summen-Kodierung hat hier, mit 1.9, eine gute Entropie. Allein die Fixe-Längen-Kodierung ist mit rund 1.5 eher suboptimal.

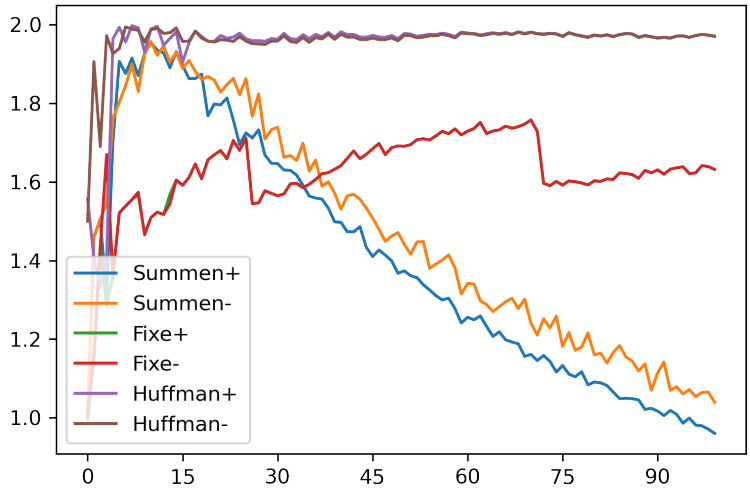


Abbildung 12: Entropie der Kodierung von zufälligen Graphen bis zu 100 Knoten

In 12 erkennt man klar, dass die Huffman-Kodierung die Entropie durchgehend nahe bei 2 hält. Die Summen-Kodierung verliert für größere Graphen ihre Informationsdichte; vermutlich da größere Knotenbezeichnungen durch lange 'G'-Ketten dargestellt werden. Die Fixe-Längen-Kodierung hält ihre Entropie

bei etwa 1.6. Vermutlich liegt es auch daran, dass kleinere Zahlen zur benötigten Breite mit 'A' aufgefüllt werden. In der Nähe der 4er-Potenzen gibt es bei dem Verfahren auch immer Entropie-Sprünge, da immer dort eine höhere Zeichenbreite für die Kodierung notwendig wird.

5 Zusammenfassung und Ausblick

Das Paket zur Lösung der Aufgabenstellung wurde erfolgreich erstellt. Alle angesetzten, implementierungsrelevanten Konzeptpunkte wurden darin umgesetzt.

Der Benchmark-Test wurde wie geplant durchgeführt und hat gezeigt, wie sich die implementierten GraphEncodings bei der Kodierung von DNA Sequenzen unterschiedlich großer Graphen geschlagen haben. Das Huffman-Verfahren hat dabei die besten Ergebnisse gezeigt; sowohl bei kleinen als auch bei großen Graphen (mit hoher Knotenanzahl). Die Anzahl an Zeichen, die das Verfahren benötigt, steigt, trotz des schlechten worst case, mit der Anzahl der Knoten immer nur minimal weiter und zeigt keine Anzeichen dafür, dass die Zeichenanzahl irgendwann stärker steigen sollte. Das Summen-Verfahren und Fixe-Länge-Verfahren haben jedoch mit steigender Knotenanzahl ein starkes Wachstum in der Anzahl der Zeichen gezeigt. Bis zu Graphen mit 45 Knoten haben beide eine ähnliche Anzahl an Zeichen in der DNA Sequenz, bei höheren jedoch ist beim Summen-Verfahren ein immer stärker steigendes Wachstum zu erkennen. Das Fixe-Länge-Verfahren ist bis zu Graphen mit ca. 70 Knoten noch verwendbar, sollte aber bei größeren Graphen nicht mehr angewandt werden. Das Summen-Verfahren hingegen sollte bei Graphen ab 45 Knoten schon nicht mehr eingesetzt werden und insbesondere nur in kleineren Graphen genutzt werden.

Grundsätzlich zeigt sich erst bei großen Graphen wirklich (ab 40-45 Knoten) wie unterschiedlich gut die Verfahren zur Kodierung der Graphen sind. Bei Graphen mit einer Größe von bis zu 20 Knoten kann man noch jedes Verfahren nutzen (natürlich unterscheiden sich da trotzdem schon die Längen der DNA Sequenzen), aber bei größeren Graphen sollte nur noch das Huffman-Verfahren verwendet werden, da es zuverlässig möglichst kurze Zeichensequenzen erstellt.

6 Literaturverzeichnis

[1] Thomas H. Cormen u. a. *Introduction to Algorithms, Third Edition*. 3rd. The MIT Press, 2009. ISBN: 0262033844.

[2] Harald Andrés Helfgott, Jitendra Bajpai und Daniele Dona. „Graph isomorphisms in quasi-polynomial time“. In: *arXiv preprint arXiv:1710.04574* (2017).

[3] David A Huffman. „A method for the construction of minimum-redundancy codes“. In: *Proceedings of the IRE* 40.9 (1952), S. 1098–1101.

[4] Eugene S Schwartz und Bruce Kallick. „Generating a canonical prefix encoding“. In: *Communications of the ACM* 7.3 (1964), S. 166–169.

[5] Jeffrey Shallit. „What this country needs is an 18c piece“. In: *Mathematical Intelligencer* 25.2 (2003), S. 20–23.

[6] Claude Elwood Shannon. „A mathematical theory of communication“. In: *The Bell system technical journal* 27.3 (1948), S. 379–423.