Aufgabe 1 b: Schmucknachrichten ILP Lösung

Teilnahme-ID: 74993

Bearbeiter dieser Aufgabe:
Mathieu de Borman

28. April 2025

Inhaltsverzeichnis

1 Lösungsidee

Diese Lösung optimiert die Länge des codierten Texts mithilfe eines Integer Linear Program (ILP). Bei dieser werden Gleichungen aufgestellt, die sicherstellen, dass die Codierung präfixfrei und optimal ist.

1.1 Präfixfreie Codes

Sei d die Durchmesser der Perlen, p die Anzahl an verschiedenen Perlen und d_i der Durchmesser der i-kleinsten Perle. D.h. z.B., dass d_p der Durchmesser der größten Perle ist. Sei a_i die Anzahl an Codewörtern der Länge i. Sei b_i die Anzahl an Codewortpräfixen der Länge i.

Es gibt nur eine einzige Möglichkeit, einen Präfix der Länge 0 zu konstruieren, nämlich als keine Perle. Außerdem muss jedes Codewort aus mindestens einem Zeichen bestehen. Es gibt also kein Codewort der Länge 0. Wir wissen also, dass $a_0 = 0$ und $b_0 = 1$ sind.

Ausgehend von einem Codewort c können bis zu p neue Codewörter konstruiert werden, indem das Codewort jeweils durch die einzelnen Perlen verlängert wird. Dann darf c allerdings nicht mehr als Codewort verwendet werden, da c sonst potentiell Präfix eines anderen Codewortes wäre. Wenn ein Präfix der Länge i nicht verwendet wird, um ein Zeichen mit einem Code der Länge i zu codieren, dann werden daraus bis zu p andere Präfixe konstruiert. Um zu berechnen, wie viele Präfixe der Länge i existieren, muss also geschaut werden, wie viele ungenutzte Präfixe der Länge i minus die Länge der letzten Perle im Präfix existieren. Ein Präfix ist genutzt, wenn ein Codewort existiert welches identisch zu dem Präfix ist. Mathematisch formuliert:

$$b_i = \sum_{\substack{j=1 \\ i-d_j \geq 0}}^{p} (b_{i-d_j} - a_{i-d_j}) \, \forall \, i \in \mathbb{N}$$

Das $i - d_j \geq 0$ sagt aus, dass negative Präfixlängen nicht betrachtet werden. Damit es keine negativen Anzahlen an ungenutzten Präfixen gibt, muss außerdem gelten:

$$a_i \leq b_i \, \forall \, i \in \mathbb{N}$$

Allerdings müssen wir i irgendwie sinnvoll begrenzen, da wir nicht unendlich viele Gleichungen betrachten können. Mithilfe der Heuristik aus Abschnitt 1.5, wird eine Begrenzung k ermittelt. k sagt aus, wie lang das längste Codewort in einer optimalen Codierung maximal sein wird.

Mithilfe dieser Gleichungen ist also festgelegt, wie viele Codewörter der welcher Länge existieren dürfen, während es nur präfixfreie Codes gibt.

1.2 Optimale Zuweisung der Zeichen

Dieser Teil ist wie ein klassisches Zuweisungsproblem. Jedes Zeichen kann genau einer der k verschiedenen Codewortlängen zugewiesen werden. Jeder Codewortlänge i werden genau a_i Zeichen zugewiesen. Wenn ich sage, dass ein Zeichen einer Codewortlänge i zugewiesen wird, dann meine ich damit, dass das Zeichen mit einem Codewort der Länge i codiert wird.

Sei $|s|$ die Anzahl an verschiedenen Zeichen und $s_{ij} \in \{0, 1\}$ die Variable, die aussagt, ob das i-te Zeichen der j-ten Codewortlänge zugewiesen wird. Wenn $s_{ij} = 0$ ist, wurde es nicht zugewiesen. Wenn $s_{ij} = 1$ ist, wurde es zugewiesen. Sei h_i die Häufigkeit des i-ten Zeichens. Wir können nun Mathematisch ausdrücken, dass jedes Zeichen maximal einmal Zugewiesen wird:

$$1 = \sum_{j=1}^{k} s_{ij} \, \forall \, i$$

Außerdem werden jeder Codewortlänge genau a_i Zeichen zugewiesen.

$$a_i = \sum_{j=1}^{|s|} s_{ji} \, \forall \, i$$

Jetzt muss noch ein Ziel definiert werden. Wenn ein Zeichen i einer Codewortlänge j zugewiesen wurde, dann nimmt das Zeichen $h_i \cdot j$ Millimeter im codierten Text ein. Die zu minimierende Gesamtlänge z ist Mathematisch formuliert:

$$\min z = \sum_{i=1}^{|s|} \sum_{j=1}^{k} h_i \cdot j \cdot s_{ij}$$

Es ist auch mit dieser Bedingung linear, da $h_i \cdot j$ sich nicht ändert, also ein konstanter Faktor ist.

Hier nochmal das Problem in rein mathematischer Formulierung:

$$
\begin{aligned}
\min \quad & z = \sum_{i=1}^{|s|} \sum_{j=1}^{k} h_i \cdot j \cdot s_{ij} \\
\text{unter den Nebenbedingungen} \quad & 1 = \sum_{j=1}^{k} s_{ij} \,\forall\, i \\
& a_j = \sum_{i=1}^{|s|} s_{ij} \,\forall\, j \\
& b_j = \sum_{\substack{i=1 \\ j-d_i \geq 0}}^{p} (b_{j-d_i} - a_{j-d_i}) \,\forall\, j \\
& a_j \leq b_j \,\forall\, i \\
& a_0 = 0,\, b_0 = 1 \\
& s_{ij} \in \{0,1\} \,\forall\, i,j \\
& a, b \in \mathbb{N}_0^k
\end{aligned}
$$

1.3 Optimierung

Es gibt häufig Zeichen mit gleichen Häufigkeiten. Statt für jedes einzelne Zeichen eine Variable zu verwenden, kann man für alle Zeichen gleicher Häufigkeit die gleiche Variable benutzen. Dadurch zählt jede Zuweisungsvariable nicht mehr, ob das Zeichen zugewiesen wurde, sondern wie viele Zeichen dieser Häufigkeit zugewiesen werden.

1.4 Dekomprimieren der Lösung

Das Ergebnis des ILP besteht aus der Länge des optimal codierten Texts und den Werten für a, b und s. Allerdings muss nun noch eine Codetabelle konstruiert werden.

Basierend auf der Idee aus Abschnitt 1.1 können wir bei der Präfixlänge 0 anfangen und dort eintragen, dass es einen Präfix mit keiner Länge gibt. Anschließend wird dieser Präfix durch jede Perle verlängert und bei der jeweils neuen Präfixlänge eingetragen. Nun kann für alle folgenden Präfixlängen gleiches für alle nun dort eingetragenen Präfixe getan werden. In dem Fall, dass in a eingetragen ist, dass es in der optimalen Lösung Codewörter der Länge i gibt, dann muss man für diese Länge a_i Präfixe den Zeichen zuordnen, statt sie zu verlängern.

Wir wissen, dass die häufigeren Zeichen die kürzeren Codewörter haben müssen damit der codierte Text eine optimale Länge hat. Das häufigste Zeichen bekommt also das kürzeste Codewort das zweithäufigste das zweit kürzeste und so weiter.

Die Codetabelle ist zwar nicht identisch zu dem, was das ILP ausgibt, aber dafür gleich optimal.

1.5 Ermittlung der maximalen Codewortlänge

In dem Fall, dass der Eingabetext nur aus einmal vorkommenden Zeichen besteht, ist bei der optimalen Codierung des Texts die Differenz zwischen dem längsten und kürzesten Codewort kleiner gleich d_p. Mit anderen Worten: die Codewörter haben in dem Fall alle eine ähnliche Länge. Der Beweis dazu erfolgt in Abschnitt 1.1 der greedy Lösung.

Das längste Codewort der Lösung wird nicht länger, wenn die Anzahl an verschiedenen Zeichen kleiner wird, aber der Text gleich lang bleibt. Zumindest habe ich für letzteres bisher noch kein anderes Verhalten beobachten können. Dennoch habe ich dafür leider keinen Beweis.

Auf Basis dieser Annahmen können wir nun einen ziemlich guten Wert für k ermitteln. Dazu konstruieren wir eine optimale Lösung für einen Text gleicher Länge, der ausschließlich aus unterschiedlichen Zeichen besteht, und ermitteln daraus das längste Codewort. Dieser Methode kann man sich annähern, indem man betrachtet, ab welcher Codewortlänge es genügend verschiedene Codewörter gibt, um alle Zeichen mit einem Codewort dieser Länge zu codieren.

Die Heuristik fängt also bei $k = 0$ an und erhöht k solange, bis die Anzahl an verschiedenen Codewörtern der Länge k größer gleich der Textlänge ist.

2 Umsetzung

Die Lösung wurde in JavaScript Umgesetzt. Das Zählen der Zeichenhäufigkeiten wird mithilfe einer Hashmap bewerkstelligt. Anschließend werden die Perlendurchmesser und die Zeichenhäufigkeiten mithilfe von der Funktion *compactList* in eine verdichtete Datenstruktur konvertiert. Bei dieser werden nämlich Doppelte Zahlen Zusammengefasst, sodass sich nur gemerkt wird, wie viele Zahlen mit dem jeweiligen Wert existieren. Nachdem mithilfe der Funktionen *heuristic* und *solveWithILP* eine Lösung gefunden wurde, berechnet die Funktion *decompress* die Codetabelle.

2.1 Heuristik

In der Funktion *heuristic* wird nun k ermittelt. Hierfür wird die Liste *wordLens* verwendet, um zu speichern, wie viele Wörter der jeweiligen Länge existieren. Anfangs gibt es nur ein Codewort bei der Länge 0. Dann wird die while schleife solange wiederholt, bis es mindestens so viele verschiedene Codewörter gibt, wie es verschiedene Zeichen im Eingabetext gibt. In der while Schleife werden immer die Codewörter um jede Perle verlängert. In der Liste wird jedoch nur gespeichert, wie viele Codewörter es für die jeweilige Länge gibt. Außerdem wird die Länge der Liste konstant gehalten, da wieder an den Anfang der Liste gesprungen wird, wenn man das Ende der Liste erreicht. Dadurch muss die Liste nur um eins größer als die Größte Perle sein. Hier ein Beispiel für einen Text der Länge 31 und den Perlen mit den Durchmessern 1,1,3 und 6. Die Darstellung Zeigt, was für Werte *wordLens* und k vor der jeweiligen Iteration haben. Die Farben zeigen, welche Werte von *wordLens* sich seit der Letzten Iteration verändert haben.

k	*wordLens*						
0	1	0	0	0	0	0	0
1	0	2	0	1	0	0	1
2	2	0	4	1	2	0	1
3	2	4	0	9	2	4	1
4	2	4	9	0	20	4	10
5	22	4	9	20	0	44	10

Da 44 größer gleich 31 ist, gibt die Funktion in dem Beispiel 5 zurück.

2.2 ILP

Als ILP-Löser verwende ich glpk.js. glpk.js kann leicht in Projekte eingebunden werden, indem man es als weitere Datei dem Projekt anfügt (oder ins gleiche Skript einbaut). Trotzdem, dass es offline funktioniert (ich also garantieren kann, dass es auch auf Ihrem Gerät funktioniert) und einen sehr kleinen Speicherplatzbedarf (245 KB) hat, löst es die ILP erstaunlich schnell. Dafür werden die Gleichungssysteme allerdings nicht immer optimal gelöst (alle Beispielausgaben sind dennoch optimal).

Das ILP wird in der Funktion *solveWithILP* gelöst. Dort wird das Problem zuerst als Objekt gespeichert, die verschiedenen Gleichungen eingetragen und dann mithilfe von glpk.js gelöst. Wenn die Gleichungen offensichtlich mit einem zu kleinen k aufgestellt wurden, wird k solange verdoppelt und das Problem nochmals gelöst, bis die Ausgabe sinnvolle Ergebnisse enthält. Es gibt zwei Kriterien, an denen erkannt wird, dass k zu klein war.

1. In der Ausgabe steht, dass die Textlänge des codierten Texts 0 ist.
2. Für die Gleichungen wurde $k+1$ verwendet, um einen Puffer zu haben. Wenn also $a_k > 0$ ist, wissen wir, dass k zu klein war.

3 Komplexitätsanalyse

Ich verwende das Wort Komplexität stellvertretend für Laufzeit- und Speicherkomplexität, um die Lesbarkeit zu erhöhen. Wenn sich die Laufzeitkomplexität von der Speicherkomplexität unterscheidet, dann spezifiziere ich das. Die Komplexität wird in Abhängigkeit von d, k und der Länge des Eingabetexts l formuliert. Das Zählen der Zeichenhäufigkeiten hat eine Komplexität von $\mathcal{O}(l)$. Das Komprimieren der Perlengrößen hat eine Komplexität von $\mathcal{O}(p \log p)$, da die Perlen vorher sortiert werden müssen. Das Ermitteln von k durch die Heuristik hat eine Laufzeitkomplexität von $\mathcal{O}(pk + d_p)$ und eine Speicherkomplexität von $\mathcal{O}(d_p)$. Das Erstellen der Codetabelle hat eine Komplexität von $\mathcal{O}(lk + l \log l)$. Ich möchte anmerken, dass die Form der Lösung in gerade mal $\mathcal{O}(k)$ Speicherplatz, nämlich als a gespeichert wird!

Durch die Optimierung, bei der gleich häufige Zeichen zusammengefasst werden, gibt es nur noch $\sqrt{l}$ statt l Zuweisungen für jede Codewortlänge. Wenn wir nämlich betrachten, wie Lang ein Text mindestens sein muss um n verschiedene Zeichenhäufigkeiten zu haben, wissen wir anhand der Gaußschen Summenformel $\frac{n^2+n}{2}$, dass die Textlänge Quadratisch zur Anzahl an verschiedenen Zeichenhäufigkeiten wächst. Dementsprechend muss die Anzahl an verschiedenen Zeichenhäufigkeiten wurzelförmig zur Textlänge anwachsen. Für jede Codewortlänge müssen also $\sqrt{l}$ Zeichenhäufigkeiten zugewiesen werden. Das ILP hat also $\mathcal{O}(k\sqrt{l})$ Variablen und $\mathcal{O}(k + \sqrt{l})$ Gleichungen. Das Erstellen der Gleichungen hat eine Komplexität von $\mathcal{O}(k\sqrt{l} + kp)$. Die Komplexität von glpk.js ist $\mathcal{O}(2^n)$, wenn n die Anzahl an verschiedenen Variablen ist. Das Lösen eines Gleichungssystems hat also eine Komplexität von $\mathcal{O}(2^{k\sqrt{l}})$. Es könnte allerdings sein, dass k mehrmals verdoppelt werden muss. Das ändert jedoch nichts an der Komplexität, da $2^{k\sqrt{l}} + 2^{\frac{k}{2}\sqrt{l}} + 2^{\frac{k}{4}\sqrt{l}} + ... < 2^{2k\sqrt{l}}$ ist. Insgesamt hat das Programm also eine Komplexität von $\mathcal{O}(p \log p + l \log l + pk + d_p + 2^{k\sqrt{l}})$.

Durch näheres Betrachten von k kann dieses durch andere Variablen ersetzt werden.
Schlimmstenfalls ist k gleich $d_p l$, da das die absolut maximale Länge eines Codeworts ist. Die Komplexität ist also $\mathcal{O}(p \log p + p d_p l + 2^{d_p l\sqrt{l}})$. Unter der Annahme, dass die Idee der Heuristik aus Abschnitt 1.5 richtig ist, ist k schlimmstenfalls gleich $d_2 \log l$. Das liegt daran, dass die Anzahl an möglichen Codewörtern sich mindestens verdoppelt, wenn diese die Länge $i + d_2$ statt i haben. Man könnte also vermuten, dass die Komplexität $\mathcal{O}(p \log p + p d_2 \log l + d_p + l^{d_2\sqrt{l}})$ ist.

In Anbetracht dessen, dass viele andere Zuweisungsprobleme mit zusätzlichen Beschränkungen auch NP-schwer sind, vermute ich, dass dieses Problem auch NP-schwer ist.

Die tatsächliche Laufzeit könnte noch stark reduziert werden, indem einige der Zuweisungsvariablen der Gleichungen durch Heuristiken eliminiert werden.

4 Beispiele

Wir rufen nun das JavaScript-Programm mit den verschiedenen BwInf-Eingabedateien auf. Die Beispiele "schmuck10.txt" und "schmuck11.txt" zeigen, dass das Programm auch mit sehr großen und vielen Perlen umgehen kann. Diese Dateien sind in demselben Ordner wie die Programmdatei.

Das Programm wird mit Hilfe des Browsers ausgeführt. Mithilfe des "Datei Hochladen" Knopfs kann man Beispieleingaben auswählen. Die Ausgabe erscheint auf der Webseite. Bei der Codetabelle steht immer der Index der jeweiligen Perle statt der Länge der Perle. Unter "Solution Data" stehen die komprimierten Lösungsdaten, d.h. die Werte von a. Das Programm terminiert für alle getesteten Eingaben in weniger als 200ms auf einem gewöhnlichen PC. Für "schmuck1.txt" und "schmuck5.txt" bis "schmuck11.txt" habe ich die Codetabelle weggelassen, um diesen Abschnitt kurz zu halten. Alle Ergebnisse sind auch im Ordner der Programmdatei zu finden.

Eingabedatei	Nachrichtenlänge	Laufzeit
schmuck1.txt	191	0.047s
schmuck2.txt	135	0.041s
schmuck3.txt	279	0.037s
schmuck4.txt	137	0.034s
schmuck5.txt	3162	0.051s
schmuck6.txt	234	0.075s
schmuck7.txt	134559	0.151s
schmuck8.txt	3287	0.039s
schmuck9.txt	36597	0.089s
schmuck10.txt	37230	0.088s
schmuck11.txt	84886	0.131s

schmuck1.txt

```
Runtime: 0.047s
Pearllengths in mm: 1,1,2
Messagelength in mm: 191
Solution Data: 0,0,1,6,10,8,0
```

schmuck2.txt

```
Runtime: 0.041s
Pearllengths in mm: 1,5
Messagelength in mm: 135
Solution Data: 0,1,0,0,0,0,0,0,0,0,0,3,1,1,1,2,0,0
a: [0]
 : [1,1,0]
b: [1,0,0,0,0,0,0]
c: [1,0,1]
d: [1,0,0,1]
e: [1,0,0,0,1]
f: [1,0,0,0,0,1]
g: [1,1,1]
h: [1,0,0,0,0,0,1]
```

schmuck3.txt

```
Runtime: 0.037s
Pearllengths in mm: 1,2,3
Messagelength in mm: 279
Solution Data: 0,0,2,1,1,2,2,1,0,0,0
a: [0,0]
b: [1]
c: [0,1]
 : [2,0]
d: [0,2,0]
e: [2,1]
f: [0,2,1]
g: [2,2]
h: [0,2,2]
```

schmuck4.txt

```
Runtime: 0.034s
Pearllengths in mm: 1,5
Messagelength in mm: 137
Solution Data: 0,0,0,0,0,0,0,1,4,1,2,3,3,0
a: [0,1,0]
b: [0,0,1,0]
c: [1,0,0,0]
d: [0,0,0,0,0,0,0,0]
e: [0,0,0,1]
f: [0,0,0,0,1]
g: [1,1]
h: [0,0,0,0,0,1]
i: [0,1,1]
j: [0,0,0,0,0,0,1]
k: [1,0,1]
l: [0,0,1,1]
m: [1,0,0,1]
n: [0,0,0,0,0,0,0,1]
```

schmuck5.txt

```
Runtime: 0.051s
Pearllengths in mm: 1,1,2,3,4,5,6
Messagelength in mm: 3162
Solution Data: 0,0,2,9,6,5,6,10,3,0
```

schmuck6.txt

```
Runtime: 0.075s
Pearllengths in mm: 1,2,3
Messagelength in mm: 234
Solution Data: 0,0,0,0,1,9,13,9,2,0,0,0,0,0,0,0,0
```

schmuck7.txt

```
Runtime: 0.151s
Pearllengths in mm: 1,1,1,1,1,1,1,2,3,4
Messagelength in mm: 134559
Solution Data: 0,4,19,23,20,15,1,0
```

schmuck8.txt

```
Runtime: 0.039s
Pearllengths in mm: 1,1,2,2,3
Messagelength in mm: 3287
Solution Data: 0,0,0,2,11,71,143,94,0
```

schmuck9.txt

```
Runtime: 0.089s
Pearllengths in mm: 1,2,3,4
Messagelength in mm: 36597
Solution Data: 0,0,0,0,0,4,14,11,20,27,74,98,181,148,97,0
```

schmuck10.txt

```
Runtime: 0.088s
Pearllengths in mm: 1,2,2,3,3,3,4,4,4,4,5,5,5,5,5,6,6,6,6,6,6,7,7,7,7,7,7,7
Messagelength in mm: 37230
Solution Data: 0,1,0,0,1,21,55,107,141,182,207,3,0
```

schmuck11.txt

```
Runtime: 0.131s
Pearllengths in mm: 1,5,23,27,56,171,198,298,723,823,1971,8723,18723,29823,30289,238623,923981
Messagelength in mm: 84886
Solution Data: 0,0,0,0,0,0,0,0,0,0,0,0,0,1,8,4,8,3,4,10,8,13,12,24,33,22,40,56,82,87,73,72,64,39,11,0
```

5 Quelltext

```
//funktion um die Listendaten zu konvertieren
function compactList(list, method) {
 //sortiere die Werte
 list.sort((a, b) => method(a) - method(b))
 let numAmount = []
 let numSize = []
 let lastSize = 0
 //gehe durch jede Zahl durch und speichere dir
 //wie viele von der jeweiligen Zahl existieren
 for (let i = 0; i < list.length; i++) {
  if (lastSize == method(list[i])) {
   numAmount[numAmount.length - 1]++
  } else {
   lastSize = method(list[i])
   numSize.push(lastSize)
   numAmount.push(1)
  }
 }
 return [numAmount, numSize]
}
//funktion um k erstmals zu bestimmen
function heuristic([pearlAmount, pearlSize], message) {
 //erstell die Liste mit den Codewortlängen
 let len = pearlSize[pearlSize.length - 1] + 1
 let wordLens = Array(len).fill(0)
 wordLens[0] = 1
 //gehe durch alle Codewortlängen durch, bis es genügend verschiedene Codewörter einer Länge gibt
 let k = 0
 while (wordLens[k % len] < message.length) {
  for (let i = 0; i < pearlAmount.length; i++) {
   wordLens[(k + pearlSize[i]) % len] += wordLens[k % len] * pearlAmount[i]
  }
  wordLens[k % len] = 0
  k++
 }
 return k
}
//funktion um die Gleichungen ins ILP einzufügen,
//die garantieren, dass kein Codewort Präfix eines anderen ist
function praefixConditions([pearlAmount, pearlSize], k, problem, glpk) {
 //füge die Gleichungen für b_i = die ungenutzen Präfixe der präfixe ohne die letzte Perle
 for (let i = 1; i <= k; i++) {
  let variables = [{ name: `_b${i}`, coef: -1 }]
  problem.subjectTo.push({
   name: `b${i}`,
   vars: variables,
   bnds: { type: glpk.GLP_FX, ub: 0, lb: 0 }
  })
  for (let j = 0; j < pearlAmount.length; j++) {
   if (i - pearlSize[j] >= 0) {
    variables.push({ name: `_b${i - pearlSize[j]}`, coef: pearlAmount[j] })
    variables.push({ name: `_a${i - pearlSize[j]}`, coef: -pearlAmount[j] })
   }
  }
 }
 //füge die Gleichungen ein, die Aussagen,
```

```
 //dass es keine negative Anzahl an ungenutzen Präfixlängen gibt
 for (let i = 1; i <= k; i++) {
  let variables = [{ name: `_b${i}`, coef: 1 }, { name: `_a${i}`, coef: -1 }]
  problem.subjectTo.push({
   name: `a${i}b${i}`,
   vars: variables,
   bnds: { type: glpk.GLP_LO, ub: 0, lb: 0 }
  })
 }
}
//funktion um die Gleichungen ins ILP einzufügen,
//die die Zeichen optimal den Codewortlängen zuweisen
function assignmentConditions([charAmount, charFrequency], k, problem, glpk) {
 //füge die Gleichungen ein, die machen, dass jedes Zeichen nur einmal zugewiesen wird
 for (let i = 0; i < charAmount.length; i++) {
  let variables = []
  problem.subjectTo.push({
   name: "s" + i,
   vars: variables,
   bnds: { type: glpk.GLP_FX, ub: charAmount[i], lb: charAmount[i] }
  })
  for (let j = 1; j <= k; j++) {
   variables.push({ name: i + "|" + j, coef: 1 })
   //füge Variable ein, nach der auch minimiert werden soll
   problem.objective.vars.push({ name: i + "|" + j, coef: charFrequency[i] * j })
  }
 }
 //füge die Gleichungen ein, die machen,
 //dass jeder Codewortlänge genau a_i Zeichen zugewiesen werden
 for (let i = 1; i <= k; i++) {
  let variables = [{ name: `_a${i}`, coef: -1 }]
  problem.subjectTo.push({
   name: `a${i}`,
   vars: variables,
   bnds: { type: glpk.GLP_FX, ub: 0, lb: 0 }
  })
  for (let j = 0; j < charAmount.length; j++) {
   variables.push({ name: j + "|" + i, coef: 1 })
  }
 }
}
//funktion um das ILP aufzustellen und lösen zu lassen
async function solveWithILP(chars, pearls, k) {
 let glpk = await GLPK()
 //erstelle ein Problem und setze a_0 auf 0 und b_0 auf 1
 let problem = {
  name: "A1_ILP",
  objective: { direction: glpk.GLP_MIN, vars: [] },
  subjectTo: [{
   name: "b0",
   vars: [{ name: "_b0", coef: 1 }],
   bnds: { type: glpk.GLP_FX, ub: 1, lb: 1 }
  }, {
   name: "a0",
   vars: [{ name: "_a0", coef: 1 }],
   bnds: { type: glpk.GLP_FX, ub: 0, lb: 0 }
  }],
  generals: [],
```

```
  binaries: []
 }
 //Deklariere alle Werte von a und b als integer
 for (let i = 0; i <= k; i++) {
  problem.generals.push(`_a${i}`, `_b${i}`)
  for (let j = 0; j < chars[0].length; j++) {
   //deklariere alle Werte von s als binäre oder integer Werte
   if (chars[0][j] == 1) {
    problem.binaries.push(j + "|" + i)
   } else {
    problem.generals.push(j + "|" + i)
   }
  }
 }
 //füge die Gleichungen ins ILP ein
 praefixConditions(pearls, k, problem, glpk)
 assignmentConditions(chars, k, problem, glpk)
 //lasse das ILP von glpk.js lösen
 let solution = (await glpk.solve(problem)).result
 let a = []
 //gebe a und die länge des codierten Texts zurück
 for (let i = 0; i <= k; i++) {
  a.push(solution.vars[`_a${i}`])
 }
 return [solution.z, a]
}
//funktion um die Codetabelle mithilfe der komprimierten Daten zu erstellen
function decompress(chars, diameters, wordLens) {
 let table = ""
 //sortiere die Zeichen absteigend nach Größe
 chars.sort((a, b) => b[1] - a[1])
 let charInd = 0
 //initialisiere die Liste in der die Präfixe der jeweiligen Längen gespeichert werden
 let prefixes = []
 for (let i = 0; i < wordLens.length; i++) {
  prefixes.push([])
 }
 prefixes[0].push("")
 //für jede Codewortlänge i
 for (let i = 0; i < wordLens.length; i++) {
  //weise a_i Codewörtern Zeichen zu
  for (let j = 0; j < wordLens[i]; j++) {
   table += `\n${chars[charInd][0]}: [${prefixes[i].pop()}]`
   charInd++
  }
  //Verlängere die anderen Präfixe der Länge i um die verschiedenen Perlen
  while (prefixes[i].length > 0) {
   let old = prefixes[i].pop()
   for (let j = 0; j < diameters.length; j++) {
    if (i + diameters[j] < wordLens.length) {
     prefixes[i + diameters[j]].push(old.length == 0 ? j + "" : `${old},${j}`)
    }
   }
  }
 }
 //gebe die Codetabelle zurück
 return table
}
```

```
//funktion um eine optimale Codierung zu finden
async function main(diameters, message) {
 //Zähle wie häufig jedes Zeichen im Text vorkommt
 let charSymbols = {}
 for (let i = 0; i < message.length; i++) {
  charSymbols[message.codePointAt(i)] = charSymbols[message.codePointAt(i)] + 1 || 1
 }
 charSymbols = Object.entries(charSymbols)
 for (let i = 0; i < charSymbols.length; i++) {
  charSymbols[i][0] = String.fromCodePoint(charSymbols[i][0])
 }
 //komprimiere die Perlendaten
 let pearls = compactList(diameters, a => a)
 //komprimiere die Zeichendaten
 let chars = compactList(charSymbols, a => a[1])
 //ermittle k und füge einen puffer hinzu
 let k = heuristic(pearls, message) + 1
 //löse das Problem
 let solution = await solveWithILP(chars, pearls, k)
 //verdopple k solange, bis die Lösung nicht offensichtlich nicht optimal ist
 while (solution[0] == 0 || solution[1][k] != 0) {
  k *= 2
  solution = await solveWithILP(chars, pearls, k)
 }
 //gebe die Länge des codierten Texts, die Codetabelle und die Werte für a zurück
 return [solution[0], decompress(charSymbols, diameters, solution[1]), solution[1]]
}
```