Aufgabe 3

33.Bundeswettbewerb Informatik 2014/'15 2.Runde

Tim Hollmann

15. April 2015

1 Lösungsidee

1.1 Regulärer Ausdruck

Zunächst dachte ich, diese Aufgabe durch Formulieren eines regulären Ausdruckes leicht lösen zu können; der gierige „greedy“-Operator „`,`“ sollte für Maximalität der ermittelten Teilstrings sorgen. Ein solcher regulärer Ausdruck wäre zum Beispiel

```
|[[ACGT]{l,}]{k,}|
```

Ein solches Suchmuster erscheint auf den ersten Blick auch logisch, da man in dieser Aufgabe im Prinzip nach einem Teilstring mit besonderen Eigenschaften in einem anderen String sucht.

Die RegEx-Engine selbst setzte diesem Lösungsansatz aber schnell ein Ende, da sie nicht in der Lage ist, rekursiv zu suchen; d.h. sie hätte zum Beispiel **`AGGA`** in **`AGGAGGA`** nicht zweimal gefunden, da sie nach Finden des ersten **`AGGA`**s an dessen Ende weiter gesucht hätte und dessen letztes **`A`** nicht mitnehmen würde. Dieser Ansatz war also hinfällig.

1.2 Naiver Algorithmus

Zielführender war der darauf folgende Algorithmus:

1. Durchgehen und Speichern aller theoretisch möglichen Teilstrings
2. Zählen ihrer Häufigkeiten
3. Maximale Teilstrings filtern nach Häufigkeit und Länge

Schritt 1 etwa so: (Listing 1)

```
// str: String mit DNS
// l,k: Minimale Länge und Häufigkeit
vector<string> substrings; // Vorläufige Lösungen

for(unsigned int start = 0; start < str.length(); start++){
    for(unsigned int length = l; length < str.length(); length++){
        substrings.push_back(str.substr(start, length));
    }
}
```

Listing 1: Naiver Algorithmus

Dieser Algorithmus wird auch „naiver Algorithmus“ genannt; er ist extrem unoptimiert und wird im Hinblick auf die gigantische Menge an Basen zunehmend ineffizient. Es entstehen auch unverhältnismäßig viele und nicht zielführende Zwischenergebnisse. Die Lösungen einiger anderer Teilnehmer scheinen mit Laufzeiten von 800 Sekunden bis

zu 2 Stunden[1] auf der Idee des naiven Algorithmus zu basieren, alle theoretisch möglichen Teilstrings durchzugehen. Theoretisch bringt dieser Algorithmus auch korrekte Lösungen, mein Anspruch war es aber trotzdem, eine für die Datenquelle angemessenere Laufzeit zu erreichen. Deshalb besteht das grundlegende Ziel der weiteren Aufgabenbearbeitung in der Optimierung des Programmes auf eine im Verhältnis zur gegebenen DNS annehmbare Laufzeit.

Ich erweitere also das Programm um eine optimierte Datenstruktur.

1.3 Optimierte Datenstruktur: Der Suffix-Baum

Mein finales Programm konstruiert als Datenstruktur einen sogenannten „Suffix-Baum“. Ein Suffix-Baum besteht - wie der Name schon sagt - aus den Suffixen seines Original-Strings. Ein Suffix ist das Ende des Strings nach einem bestimmten Buchstaben. So hat der String `CAGGAGGATTA` (das Beispiel aus der Angabe) folgende Suffixe: (Abb.1). Alle Suffixe des Original-Strings mit dem selben Anfangsbuchstaben werden übereinander gelegt. Wo Zeichen unterschiedlich sind, werden für jede Lösung abgehende „Zweige“ erstellt. So ergibt sich aus dem Beispiel der Angabe folgender Suffix-Baum: (Abb.2).

Startpunkt	Suffix
0	CAGGAGGATTA
1	AGGAGGATTA
2	GGAGGATTA
3	GAGGATTA
4	AGGATTA
5	GGATTA
6	GATTA
7	ATTA
8	TTA
9	TA
10	A

Abbildung 1: Suffixe von `CAGGAGGATTA`

Dabei bedeutet `$`[2] das Ende des Original-Strings. Dies ist nötig, da sonst z.B. das Suffix `AGGA` in `AGGAGGA` komplett untergehen würde; mit `$` gibt es nach `AGGA` eine Verzweigung zum Ende und `AGGA` hat damit eine Häufigkeit von 2.

In Abbildung 3 noch ein anderer Suffix-Baum vom String `BANANA`. Die Suffixe befinden sich immer zwischen der Wurzel und jedem der Knoten.

[1] siehe EI-Communitiy-Forum

[2] Auch „Wächter“ oder *sentinel* genannt

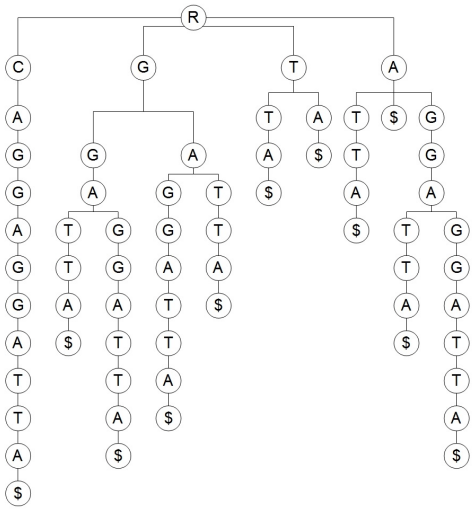

Abbildung 2: Suffix-Tree von **CAGGAGGATTA** (genauer: eigentlich ein „Suffix-Trie“)

1.4 Warum ist die Verwendung eines Suffix-Baumes von Vorteil?

Die Verwendung eines Suffix-Baumes hat gegenüber dem naiven Algorithmus mehrere Vorteile:

- Um einen Suffix-Baum zu konstruieren, benötigt man lediglich die Suffixe eines Strings, wodurch der Algorithmus zur Konstruktion eines Suffix-Baumes bei einem String der Länge n eine Laufzeitkomplexität von $\mathcal{O}(n)$ hat (linear), wo hingegen der naive Algorithmus $\mathcal{O}(n^2)$ Zeit benötigt, was ihn für längere DNS nahezu untauglich macht.

- Bei der Auswertung eines Suffix-Baumes behandelt man nicht - wie beim naiven Algorithmus - jeden möglichen Teilstring einzeln. Stattdessen betrachtet man nur die Knoten[3], also die Veränderungen der Suffixe im Verhältnis zueinander. Allein durch die Betrachtung der Knoten vermeidet man Unmengen von nicht zielführenden Lösungen; sich deckende Lösungen der selben Häufigkeit wie z.B. **AG** und **AGG** (siehe Beispiel der Angabe) kommen so überhaupt nicht als Lösung in Frage, da sie sich alle im Suffix **AGGA** der selben Häufigkeit vollständig decken; der Knoten kommt deshalb erst nach AGGA (siehe Abb. 2).

- Passt in der Auswertung des Suffix-Baumes zum Beispiel ein Ast nicht zu den Lösungs-Eigenschaften, verwirft man diesen Ast und damit alle davon abgehenden Lösungen gleich mit. So erspart man sich das Verarbeiten dieser Unter-Äste. Man „stutzt“ also den Suffix-Baum an der Stelle, an der es für die Lösung nicht mehr passt und überprüft die Suffixe nicht einzeln; dies ist erheblich effizienter.

[3] engl. „nodes“

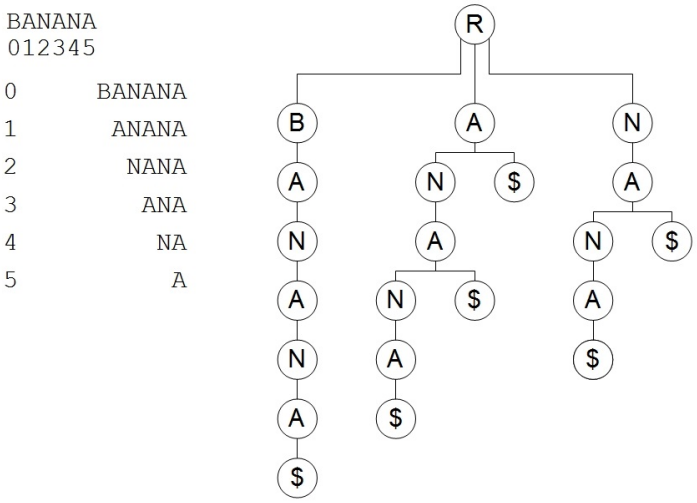

Abbildung 3: Suffix-Tree von **BANANA**

- Im Gegensatz zum naiven Algorithmus kann man, wenn man den Suffix-Baum komplett konstruiert hat, wichtige Eigenschaften wie Länge und Häufigkeit der Suffixe direkt ablesen. Die Länge eines Suffixes entspricht dem Abstand des aktuellen Knotens zur Wurzel. Die Häufigkeit eines Suffixes entspricht der Summe aller von ihm abgehenden Äste und deren Unter-Äste. Das Ablesen dieser Eigenschaften ist von erheblichem Vorteil, da sich hierdurch die Lösungen definieren (siehe **1.5**).

1.5 Kriterien für eine Lösung und Definition am Suffix-Baum

Eine Lösung nach Definition der Aufgabenstellung (unbeachtet der Maximalität, hierzu siehe **1.6**) ist ein Suffix mit Mindest-Länge l und Mindest-Häufigkeit k. Übertragen auf den Suffix-Baum heißt das, dass der Knoten mindestens l Buchstaben von der Wurzel entfernt sein und in Summe mindestens k Unterknoten haben muss.

Als Beispiel hier das Beispiel aus der Angabe mit $l = 2$,$k = 2$: (Abb. 4). Ermittelt werden die potentiellen Lösungen **GA**($l = 2, k = 2$), **GGA**(3,2), **AGGA**(4,2).

Die Maximalität der Lösungen ist hier noch nicht gegeben; alle ermittelten Lösungen sind gleich häufig und **GA** und **GGA** sind in **AGGA** vorhanden).

1.6 Erreichen der Maximalität

Maximalität ist erreicht, wenn kein Suffix einer Häufigkeit in einem anderen der selben Häufigkeit vorkommt. Hierzu muss man die Suffixe einer Häufigkeit miteinander vergleichen und Übereinstimmungen prüfen und ggf. Lösungen entfernen. Alle Suffixe einer Häufigkeit miteinander zu vergleichen ist aber ineffizient. Allein aus Gründen der Logik muss ein String, der den zu überprüfenden String beinhalten soll, mindestens dessen Länge besitzen. Wenn beide Strings die selbe Länge besitzen, reicht es für den

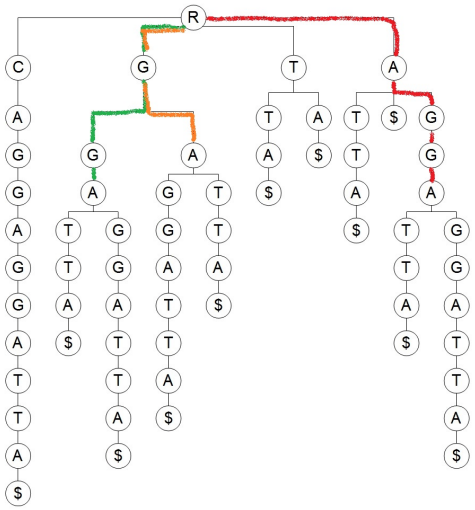

Abbildung 4: Potentielle Lösungen für $l = 2$,$k = 2$: GA, GGA, AGGA. Von der Wurzel bis zu jedem Knoten, der mindestens 2 Buchstaben von ihr entfernt ist und in Summe mindestens 2 untergeordnete Suffixe hat.

Negativ-Beweis aus, zunächst nur die erste Stelle zu überprüfen.

Diese Vergleichskriterien senken die Anzahl der verglichenen Buchstaben und steigern die Effizienz.

Wie erhält man nun aber die Lösungen einer Häufigkeit? Für jede Häufigkeit die komplette Lösungsmenge nach anderen der selben Häufigkeit zu durchsuchen ist ineffizient. Ich sortiere die Lösungen nach Häufigkeit und Länge. Dann gehe ich von oben nach unten, bis sich die Häufigkeit verändert. Dann habe ich zwischen Start- und Endpunkt die Lösungen einer Häufigkeit, die ich dann von oben nach unten miteinander vergleiche, da sich die längste Lösung dieser Häufigkeit ganz oben befindet.

2 Umsetzung

2.1 Implementierung des Suffix-Baumes

Das Hauptproblem bestand in der Implementierung eines Suffix-Baumes. Zunächst musste eine Klasse definiert werden, aus der sich jeder erdenkliche Suffix-Baum konstruieren lassen könnte. In zweiten Schritt musste das Programm in die Lage versetzt werden, einen Suffix-Baum als Summe von Instanzen der definierten Klasse zu jedem beliebigen DNS-String konstruieren zu können. Gleichzeitig sollte die Konstruktionszeit des Suffix-Baumes nicht viel mehr als linear mit der Länge seiner Quelle zunehmen.

Zur Leistungsoptimierung verwendet mein Programm im Suffix-Baum keine Buchstaben, sondern lediglich Integer-Pointer auf den Original-String.

2.1.1 Die Suffix-Klasse

In der Logik meines Programmes besteht ein Suffix-Baum aus Elementen der Klasse `suffix`, die Einen Suffix repräsentieren, von denen an bestimmten Stellen andere Suffixe abgehen. Jede Instanz der Klasse `suffix` hat einen Stack aus anderen Instanzen der Klasse `suffix`, die von ihm abgehen; jede dieser untergeordneten Suffixe hat einen Integer-Pointer auf die Stelle des Mutter-Suffixes, von der er abgeht. Siehe Abbildung 5.

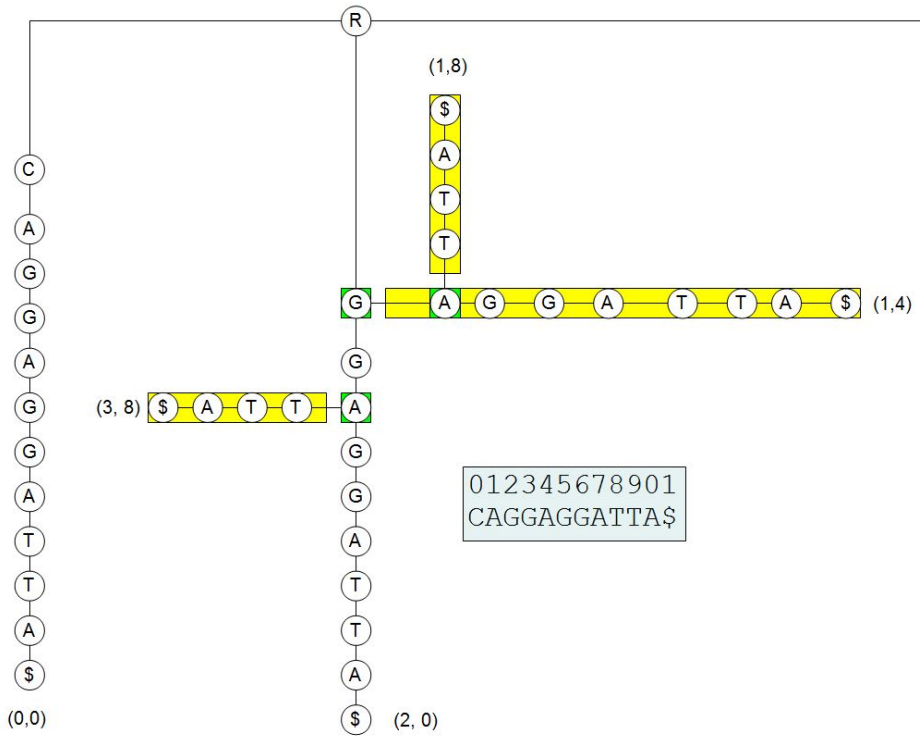

Abbildung 5: Suffix-Baum in der Logik eines Programmes: Von jedem Suffix (hier weiß) gehen Suffixe ab (gelb), die einen relativen Pointer auf die Stelle halten, von der sie abgehen (hier jeweils grün). Von diesen abgehenden Suffixen können genauso Suffixe abgehen (siehe **`G-A`**).

Jede Instanz der Klasse `suffix` hat zusätzlich noch einen „absoluten“ Zeiger, der im Gegensatz zum relativen Zeiger nicht auf den Mutter-Suffix zeigt, sondern auf die Stelle am Original-Quellstring, an der dieser Suffix beginnt. (Dies wird später für die Konstruktion des Suffix-Baumes benötigt). So hat zum Beispiel in Abbildung 5 der Suffix **`TTA$`** den relativen Pointer 3, weil er sich an die dritte Stelle des darüberliegenden Suffixes **`GGAGGATTA$`** angehängt hat, und einen absoluten Pointer 8, da er sich im Original-String an Stelle 8 befindet (siehe blauer Kasten Abb. 5).

Dies ist alles nur möglich, wenn es Stellen geben kann, an die die Suffixe angehängt werden können; man muss die längsten Suffixe zuerst behandeln. Mein Programm bearbeitet die DNS von vorne, sodass die längsten Suffixe zuerst bearbeitet werden.

Ich definiere einen Suffix also wie folgt: (Listing 2)

```
28 // ---------- Suffix-Klasse ---------- //
29 class suffix{
30 public:
31     int relPointer; // relativeer Zeiger auf die Stelle, an der er sich am ↩
       Mutter-Suffix andockt
32     int absPointer; // absoluter Zeiger auf die Stelle am original-String
33
34     vector<suffix> subSuffixes; // Stack mit Sub-Suffixen
35
36     suffix(int relStart, int absStart){ // Konstruktor
37         this->relPointer = relStart;
38         this->absPointer = absStart;
39     }
40
41     [...]
```

Listing 2: Definition der Suffix-Klasse (noch unvollständig)

Schlussendlich besteht dann der Suffix-Baum aus vier Instanzen der Klasse `suffix` (den längsten Suffixen mit unterschiedlichen Anfangsbuchstaben). Diese vier Instanzen sind die Haupt-/Urknoten und bilden die Wurzel des Suffix-Baumes; mein Programm speichert sie als Vector (Listing 3).

```
212 vector<suffix> root; //Suffixbaum-Wurzel
```

Listing 3: Suffix-Baum-Wurzel

2.1.2 Konstruieren eines Suffix-Baumes durch das Programm

Die Suffixe des Original-Strings werden sukzessive von vorne abgearbeitet. Das Programm prüft zuerst, ob es in `root` einen Suffix gibt, der mit dem Anfangsbuchstaben des aktuellen Suffixes beginnt (genauer: dessen absoluter Zeiger auf den selben Anfangsbuchstaben im Originalstring zeigt). Wenn ein solcher Hauptknoten gefunden wurde, werden ihre Buchstaben progressiv verglichen. Wird eine Nicht-Übereinstimmung gefunden („missmatch“), wird zuerst im `subSuffix`-Stack nach einem Abgehenden Suffix gesucht, dessen reltiver Zeiger auf die Stelle des Missmatches zeigt. Existiert einer, wird dieser angewiesen, das Suffix abzüglich der bereits verglichenen Buchstaben aufzunehmen. Existiert keiner, wird eine neue Instanz der Klasse `suffix` mit relativem Zeiger auf die Stelle des Missmatches erstellt, die dann in den `subSuffix`-Stack des übergeordneten Suffixes hinzugefügt wird.

Da der gesamte Prozess des Suffix-Aufnehmens logischer Weise dynamisch und beliebig oft und tief ablaufen muss, habe ich der Klasse `suffix` eine weitere Memberfunktion hinzugefügt: `suffix::takeUp`. Diese Funktion nimmt als Argument einen absoluten Zeiger; den Starpunkt eines Suffixes, der in diese Instanz aufgenommen werden soll. (Listing: 4).

```
49  bool takeUp(int startPointer){ // Dieser Suffix soll einen anderen Suffix ↩
        mit Startpunkt startPointer aufnehmen
50  //Der Suffix von startPointer bis String-Ende(length) soll diesem Suffix ↩
        hinzugefügt werden
51
52      //sukzessiven Vergleich starten
53      for (unsigned int deep = 1; deep < (length - startPointer); deep++){
54          if (line[this->absPointer + deep] != line[startPointer + deep]){
55              //Ungleichheit entdeckt. Suche nach Subsuffix an dieser Stelle
56              bool found = false;
57              for (unsigned int i = 0; i < this->subSuffixes.size(); i++){
58                  if (this->subSuffixes[i].relPointer == deep && ↩
                        line[this->subSuffixes[i].absPointer] == ↩
                        line[startPointer + deep]){
59                      this->subSuffixes[i].takeUp(startPointer + deep);
60                      found = true;
61                      break;
62                  }
63              }
64              if (!found){
65                  //Kein geeignetes Sub-Suffix an dieser Stelle gefunden -> ↩
                        Suffix hinzufügen
66                  this->subSuffixes.push_back(*new suffix(deep, startPointer ↩
                        + deep));
67              }
68              break;
69          }
70      }
71      return true;
72  }
```

Listing 4: `suffix::takeUp`

Diese Funktion wird an der Suffixbaum-Wurzel für jedes Suffix aufgerufen, sofern schon ein passender Hauptknoten besteht (Listing 5).

```
219 // Suffixe von Anfang an sukzessive abarbeiten
220 for (unsigned int i = 0; i < length; i++){
221     //Neues Suffix: i bis String-Ende (line[length-1])
222
223     //Passenden Root-Knoten ermitteln
224     bool found = false; //Knoten gefunden?
225     for (unsigned int n = 0; n < root.size(); n++){
226         //Gleicher Anfangsbuchstabe?
227         if (line[root[n].absPointer] == line[i]){
228             root[n].takeUp(i); // Aufnehmen des Suffixes
```

```
229             found = true;
230             break;
231         }
232     }
233
234     //Wenn Urknoten noch nicht existiert, hinzufügen, wenn nicht ↩
            Escape-Zeichen
235     if (!found && line[i] != escapeSign){
236         root.push_back(*new suffix(0, i));
237     }
238
239 }
```

Listing 5: Verarbeiten der Suffixe des Original-Strings: Suchen nach Hauptknoten und Aufruf der `suffix::takeUp()`-Funktion (`line` ist die DNS-Quelle und `length` ihre Länge (aus Effizienzgründen ausgelagert))

Da sich die Suffixe dann gegenseitig aufrufen und vergleichen, ist zur Konstruktion des Suffix-Baumes nichts weiteres mehr nötig.

Ein von meinem Programm generierter Suffix-Baum zum Beispiel-String grafisch ausgegeben: (Abb. 6)

2.2 Auswerten des Suffix-Baumes nach l und k

Die Suffixe des Original-Strings sind dank des Wächters `$` im Quellstring einzigartig; sie haben deshalb standardmäßig die Häufigkeit 1. Überall dort, wo zwei oder mehr Suffixe teilweise übereinstimmen, gibt es dank der Wächter immer spätestens beim vorletzten Buchstaben einen Knoten.

Zwischen jedem Knoten und der Wurzel befindet sich deshalb ein Suffix mit einer Häufigkeit ≥ 2. Mein Programm geht deshalb alle Knoten durch und überprüft, ob diese mindestens l Buchstaben von der Suffix-Baum-Wurzel entfernt sind. Daraufhin wird die Häufigkeit ermittelt; dafür müssen die Häufigkeit des vom aktuellen Knoten abgehenden Astes und aller am Suffix darunterliegenden abgehenden Äste addiert werden. Um die Häufigkeit eines Astes ermitteln zu können, reicht es aber nicht, 1 + die Anzahl der Sub-Suffixe im Stack zu rechnen, da diese selbst noch Sub-Suffixe haben. Um die Häufigkeit eines beliebigen Astes ermitteln zu können, erweitere ich die Klasse `suffix` um eine weitere Memberfunktion: `suffix:getFrequency()`.

```
41 unsigned int getFrequency(void){ //Ermitteln der Häufigkeit
42     unsigned int counter = 1;
43     for (auto &suffixes : this->subSuffixes){
44         counter += suffixes.getFrequency();
45     }
46     return counter;
47 }
```

Listing 6: Häufigkeits-Funktion und dynamisches Ermitteln der Häufigkeit eines Astes

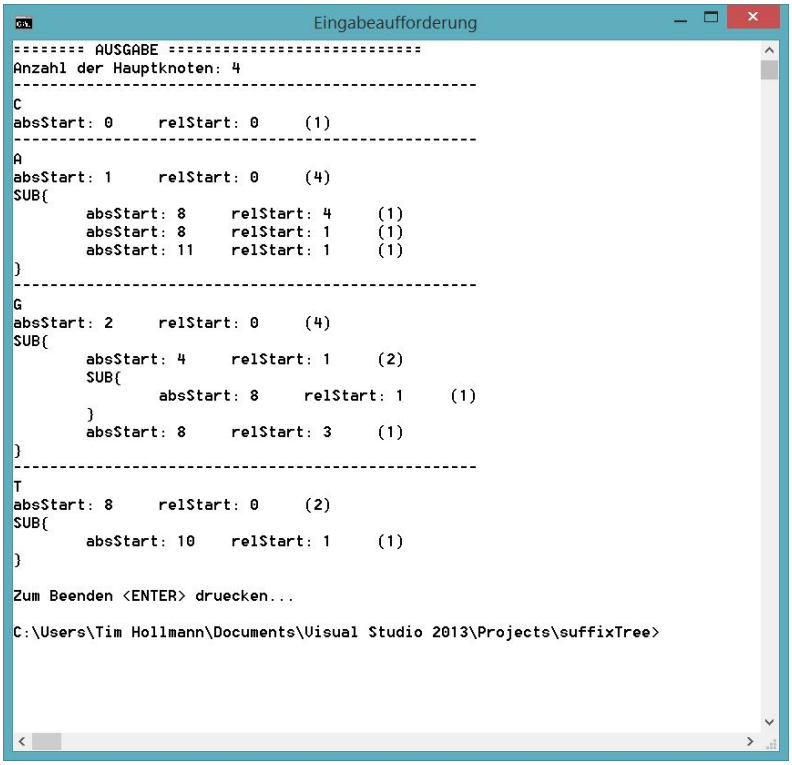

Abbildung 6: Vom Programm konstruierter und semi-grafisch ausgegebener Suffix-Baum des Beispiel-Strings **CAGGAGGATTA**. Kontrolle: Es gibt die selben absoluten und relativen Zeiger wie der per Hand konstruierte Suffix-Baum in Abb. 5

Diese Funktion gibt die Häufigkeit des Suffixes sowie die der abgehenden Suffixe zurück, indem sie sich selbst im Kontext des Sub-Suffixes aufruft (Listing 6) und kann so die Häufigkeit eines beliebig langen Astes ermitteln.

Eine Funktion `process` (Listing 9) verarbeitet den Suffix-Baum, indem sie alle Knoten des Suffix-Baumes durchgeht, die mindestens l Buchstaben von der Wurzel entfernt sind. Haben alle abgehenden gleichen und daruntergelegenen Suffixe zusammen die Mindesthäufigkeit k, wird der Suffix in die Lösungsmenge `loesungen` (Listing 7) der Struktur `loesung` (Listing 8) aufgenommen.

```
88 vector<loesung> loesungen;
```

Listing 7: Lösungsmenge

```
76 // Lösungs-Klasse
77 class loesung{
78 public:
79     string str;
80     unsigned int haeufigkeit;
```

```
81      loesung(string a, int b){
82          this->str = a;
83          this->haeufigkeit = b;
84      }
85      loesung(void){} // Standard-Konstruktor
86  };
```

Listing 8: Lösungs-Klasse

```
99  // Process: Suffix-Baum nach l und k auswerten
100 int process(suffix suf, string suffixSoFar){
101
102     if (k <= 1){
103         //Loesung gefunden: Aktueller String: suf.absPointer -> Stirng-Ende
104         if (suf.absPointer != length){ //Pointer darf nicht auf letzte ↩
                Stelle zeigen (-> $)
105             string currentSuffix = suffixSoFar + ↩
                    line.substr(suf.absPointer, length - suf.absPointer);
106             if (currentSuffix.length() >= l){
107                 loesungen.push_back(*new loesung(currentSuffix, 1));
108             }
109         }
110     }
111
112     for (auto &suffix : suf.subSuffixes){
113
114         // Suffix: suffixSoFar + line.substr(suf.absPointer, ↩
                suffix.relPointer)
115         // Suffix zwischen suf und suffix
116
117         string currentSuffix = suffixSoFar + line.substr(suf.absPointer, ↩
                suffix.relPointer);
118
119         // Stimmt die Längendifferenz + suffixSoFar.size() >= l?
120         if (currentSuffix.length() >= l){
121             //Häufigkeit ermitteln
122
123             unsigned int counter = 1;
124
125             // Häufigkeiten gleicher und darunterliegender Suffixe addieren
126             for (auto &s : suf.subSuffixes){
127                 if (s.relPointer >= suffix.relPointer){
128                     counter += s.getFrequency();
129                 }
130             }
131
132             if (counter >= k){ loesungen.push_back(*new ↩
                    loesung(currentSuffix, counter)); }
133
134         }
135         process(suffix, suffixSoFar + line.substr(suf.absPointer, ↩
                suffix.relPointer));
```

```
136
137     }
138
139     return 1;
140
141 }
```

Listing 9: Funktion `process`

2.3 Maximalisieren

Die nun in `loesungen` enthaltenen potentiellen Lösungen können mehrmals die exakte selbe Lösung enthalten; dies zu verhindern hätte bedeutet, alle bereits ermittelten Lösungen bei jeder Ermittlung einer neuen Lösung in `process` durchgehen zu müssen - dies wäre extrem ineffizient gewesen. Das Herausfiltern dieser doppelten Lösungen geschieht in einem Rutsch mit dem Maximalisieren der Lösungen.

Nach **1.6** werden die ermittelten Lösungen im vector `loesungen` zunächst nach Häufigkeit und Länge sortiert (Zeile 264). Daraufhin werden die Array-Bereiche ermittelt, in denen Lösungen der selben Häufigkeit liegen. (Zeile 270-277). Dann werden diejenigen Lösungen miteinander verglichen, die

- Noch nicht aussortiert wurden (`loesung::haeufigkeit`/`myFreq` $\neq 0$)
- Deren String kürzer oder gleich lang mit dem String der zu vergleichenden Lösung sind
- Bei gleich langen Lösungen werden zunächst nur die ersten Stellen verglichen, bevor der komplette String verglichen wird.

```
263 // Lösungen nach Häufigkeit und Länge sortieren
264 sort(loesungen.begin(), loesungen.end(), sortLoesungen);
265
266 vector<loesung> loesungenGefiltert; // Schlussendliche Lösungen
267
268 for (unsigned int i = 0; i < loesungen.size(); i++){
269
270     unsigned int myFrequency = loesungen[i].haeufigkeit;
271
272     //Range ermitteln
273     unsigned int e; // End-Zeiger
274
275     for (e = i + 1; e < loesungen.size(); e++){
276         if (loesungen[e].haeufigkeit != myFrequency){ break; }
277     }
278
279     // Range vergleichen
280     for (unsigned int a = i; a < e; a++){
281         unsigned int myFreq = loesungen[a].haeufigkeit;
```

```
282         string myStr = loesungen[a].str;
283
284         if (myFreq != 0){
285             for (unsigned int b = i; b < e; b++){
286                 if (a != b && loesungen[b].haeufigkeit != 0 && myFreq == ↩
                        loesungen[b].haeufigkeit){
287
288                     if (myStr.size() == loesungen[b].str.size()){
289                         if (myStr[0] == loesungen[b].str[0] && myStr == ↩
                                loesungen[b].str){
290                             loesungen[b].haeufigkeit = 0;
291                         }
292                     }
293                     else if (myStr.size() > loesungen[b].str.size()){
294                         if (myStr.find(loesungen[b].str) != string::npos){
295                             loesungen[b].haeufigkeit = 0;
296                         }
297                     }
298
299                 }
300             }
301         }
302     }
303
304     i = e-1; // Zur nächsten Range springen
305
306 }
```

Listing 10: Vorläufige Lösungen filtern

Alle doppelten oder nicht maximalen vorher ermittelten Lösungen haben nun die Häufigkeit 0 zugewiesen bekommen; diese müssen aussortiert werden. Dies ist trivial und erfolgt im Zuge der Ausgabe.

3 Quelltext

```
8  #include <iostream>     // std::cout
9  #include <string>       // std::string
10 #include <vector>       // std::vector
11 #include <fstream>      // std::fstream
12 #include <ctime>        // std::clock()
13 #include <algorithm>    // std::sort()
14 #include <sstream>      // std::sstream_basic
15 #include <map>          // std::map
16
17 using namespace std;
18
19 // --- Globale Variablen ----
20
21 string line;
```

```
22  unsigned int length;
23
24  unsigned int l, k;
25  string filename;
26  char escapeSign = '$';
27
28  // ---------- Suffix-Klasse ---------- //
29  class suffix{
30  public:
31      int relPointer; // relativeer Zeiger auf die Stelle, an der er sich am ↩
            Mutter-Suffix andockt
32      int absPointer; // absoluter Zeiger auf die Stelle am original-String
33
34      vector<suffix> subSuffixes; // Stack mit Sub-Suffixen
35
36      suffix(int relStart, int absStart){ // Konstruktor
37          this->relPointer = relStart;
38          this->absPointer = absStart;
39      }
40
41      unsigned int getFrequency(void){ //Ermitteln der Häufigkeit
42          unsigned int counter = 1;
43          for (auto &suffixes : this->subSuffixes){
44              counter += suffixes.getFrequency();
45          }
46          return counter;
47      }
48
49      bool takeUp(int startPointer){ // Dieser Suffix soll einen anderen ↩
            Suffix mit Startpunkt startPointer aufnehmen
50          //Der Suffix von startPointer bis String-Ende(length) soll diesem ↩
                Suffix hinzugefügt werden
51
52          //sukzessiven Vergleich starten
53          for (unsigned int deep = 1; deep < (length - startPointer); deep++){
54              if (line[this->absPointer + deep] != line[startPointer + deep]){
55                  //Ungleichheit entdeckt. Suche nach Subsuffix an dieser ↩
                        Stelle
56                  bool found = false;
57                  for (unsigned int i = 0; i < this->subSuffixes.size(); i++){
58                      if (this->subSuffixes[i].relPointer == deep && ↩
                            line[this->subSuffixes[i].absPointer] == ↩
                            line[startPointer + deep]){
59                          this->subSuffixes[i].takeUp(startPointer + deep);
60                          found = true;
61                          break;
62                      }
63                  }
64                  if (!found){
65                      //Kein geeignetes Sub-Suffix an dieser Stelle gefunden ↩
                            -> Suffix hinzufügen
66                      this->subSuffixes.push_back(*new suffix(deep, ↩
                            startPointer + deep));
67                  }
```

```
68                  break;
69              }
70          }
71          return true;
72      }
73
74  };
75
76  // Lösungs-Klasse
77  class loesung{
78  public:
79      string str;
80      unsigned int haeufigkeit;
81      loesung(string a, int b){
82          this->str = a;
83          this->haeufigkeit = b;
84      }
85      loesung(void){} // Standard-Konstruktor
86  };
87
88  vector<loesung> loesungen;
89
90  // Vergleichsfunktion, um die Lösungen nach Häufigkeit und Länge zu sortieren
91  bool sortLoesungen(const loesung& x, const loesung& y){
92      if (x.haeufigkeit > y.haeufigkeit) return false;
93      if (x.haeufigkeit < y.haeufigkeit) return true;
94      if (x.str.length() < y.str.length()) return false;
95      if (x.str.length() > y.str.length()) return true;
96      return false;
97  }
98
99  // Process: Suffix-Baum nach l und k auswerten
100 int process(suffix suf, string suffixSoFar){
101
102     if (k <= 1){
103         //Loesung gefunden: Aktueller String: suf.absPointer -> Stirng-Ende
104         if (suf.absPointer != length){ //Pointer darf nicht auf letzte ↩
                Stelle zeigen (-> $)
105             string currentSuffix = suffixSoFar + ↩
                    line.substr(suf.absPointer, length - suf.absPointer);
106             if (currentSuffix.length() >= 1){
107                 loesungen.push_back(*new loesung(currentSuffix, 1));
108             }
109         }
110     }
111
112     for (auto &suffix : suf.subSuffixes){
113
114         // Suffix: suffixSoFar + line.substr(suf.absPointer, ↩
                suffix.relPointer)
115         // Suffix zwischen suf und suffix
116
117         string currentSuffix = suffixSoFar + line.substr(suf.absPointer, ↩
                suffix.relPointer);
```

```
118
119         // Stimmt die Längendifferenz + suffixSoFar.size() >= l?
120         if (currentSuffix.length() >= l){
121             //Häufigkeit ermitteln
122
123             unsigned int counter = 1;
124
125             // Häufigkeiten gleicher und darunterliegender Suffixe addieren
126             for (auto &s : suf.subSuffixes){
127                 if (s.relPointer >= suffix.relPointer){
128                     counter += s.getFrequency();
129                 }
130             }
131
132             if (counter >= k){ loesungen.push_back(*new ↩
                    loesung(currentSuffix, counter)); }
133
134         }
135         process(suffix, suffixSoFar + line.substr(suf.absPointer, ↩
                suffix.relPointer));
136
137     }
138
139     return 1;
140
141 }
142
143 /* Funktion für Dateneingabe */
144 bool input(int argc, char *argv[]){
145
146     if (argc == 4){
147
148         stringstream sstream;
149         sstream.clear();
150
151         /* Datei öffnen und auslesen */
152
153         sstream << argv[1];
154
155         filename = "";
156         sstream >> filename;
157
158         fstream fin(filename);
159         getline(fin, line);
160         fin.close();
161
162         /* l und k*/
163
164         sstream.clear();
165         sstream << argv[2];
166         sstream >> l;
167
168         sstream.clear();
169         sstream << argv[3];
```

```
170         sstream >> k;
171
172         /* Überprüfung */
173
174         if (line.size() == 0 || l == 0 || k == 0){
175             cout << "\nFehler - Fehlerhafte Dateneingabe. (" << argc << " ↩
                    Parameter).\nExistiert die Quell-Datei? k und l dürfen ↩
                    nicht 0 sein. \nMuster: mississippi.exe <DNS-Quelldatei> ↩
                    <l> <k>\n\n";
176             return false;
177         }
178
179         return true;
180
181     }else{
182         cout << "\nFehler - Nicht ausreichende Dateieingabe (" << argc << ↩
                " Argumente). \nMuster: mississippi.exe <DNS-Quelldatei> <l> ↩
                <k>\n\n";
183         return false;
184     }
185
186 }
187
188 /* Hauptfunktion */
189 int main(int argc, char *argv[]){
190
191     cout << "\n+-----------------------------------------------+ ";
192     cout << "\n| Mississippi v3 - DNS-Sequenzierungs-Programm  | ";
193     cout << "\n| Tim Hollmann @ 33.BwInf 2.Runde 3.Aufgabe     | ";
194     cout << "\n+-----------------------------------------------+ \n";
195
196     // Eingabe
197     if (!input(argc, argv)) return 0;
198
199     //Terminalwort anfügen
200     line += escapeSign;
201
202     //String-Länge extra speichern; ist effizienter
203     length = line.length();
204
205     /* Übersicht über eingelesene Daten */
206     cout << "\n" << "Eingelesene Zeichenketten-L[ae]nge:" << length-1 << " ↩
            (" << filename << ")\n";
207     cout << "\n" << "l:\t" << l;
208     cout << "\n" << "k:\t" << k;
209
210     // ========== Verarbeitung 1 : Suffix-Baum erstellen ========== //
211
212     vector<suffix> root; //Suffixbaum-Wurzel "pflanzen"
213
214     cout << "\n\n=== Verarbeitungsschritt 1 : Erstellen des Suffix-Baumes ↩
            ===\nBitte warten...";
215
216     //Zeit nehmen: vorher
```

```
217     clock_t start = clock();
218 
219     // Suffixe von Anfang an abarbeiten
220     for (unsigned int i = 0; i < length; i++){
221         //Neues Suffix: i bis String-Ende (line[length-1])
222 
223         //Passenden Root-Knoten ermitteln
224         bool found = false; //Knoten gefunden?
225         for (unsigned int n = 0; n < root.size(); n++){
226             //Gleicher Anfangsbuchstabe?
227             if (line[root[n].absPointer] == line[i]){
228                 root[n].takeUp(i);
229                 found = true;
230                 break;
231             }
232         }
233 
234         //Wenn Urknoten noch nicht existiert, hinzufügen, wenn nicht ↩
                Escape-Zeichen
235         if (!found && line[i] != escapeSign){
236             root.push_back(*new suffix(0, i));
237         }
238 
239     }
240 
241     //Zeit nehmen: nachher
242     cout << "fertig.\nBen[oe]tigte Zeit: " << ((clock() - start) / ↩
            (double) CLOCKS_PER_SEC) << " Sekunde(n).";
243 
244     // ========== Verarbeitung 2 : Suffix-Baum auswerten ========== //
245     cout << "\n\n=== Verarbeitungsschritt 2 : Auswerten des erstellten ↩
            Suffix-Baumes ===\nBitte warten...";
246 
247     //Zeit nehmen: vorher
248     start = clock();
249 
250     for (auto &r : root) { process(r, ""); }
251 
252     cout << "fertig.\nBen[oe]tigte Zeit: " << ((clock() - start) / ↩
            (double)CLOCKS_PER_SEC) << " Sekunde(n).";
253 
254     cout << "\n" << loesungen.size() << " vorl[ae]ufige L[oe]sungen ↩
            gefunden.";
255 
256     // ========== Verarbeitung 3 : Lösungen filtern -> maximal ========== //
257 
258     cout << "\n\n=== Verarbeitungsschritt 3 : Filtern der L[oe]sungen ↩
            ===\nBitte warten...";
259 
260     //Zeit nehmen: vorher
261     start = clock();
262 
263     // Lösungen nach Häufigkeit und Länge sortieren
264     sort(loesungen.begin(), loesungen.end(), sortLoesungen);
```

```
265
266     vector<loesung> loesungenGefiltert;
267
268     for (unsigned int i = 0; i < loesungen.size(); i++){
269
270         unsigned int myFrequency = loesungen[i].haeufigkeit;
271
272         //Range ermitteln
273         unsigned int e; // End-Zeiger
274
275         for (e = i + 1; e < loesungen.size(); e++){
276             if (loesungen[e].haeufigkeit != myFrequency){ break; }
277         }
278
279         // Range vergleichen
280         for (unsigned int a = i; a < e; a++){
281             unsigned int myFreq = loesungen[a].haeufigkeit;
282             string myStr = loesungen[a].str;
283
284             if (myFreq != 0){
285                 for (unsigned int b = i; b < e; b++){
286                     if (a != b && loesungen[b].haeufigkeit != 0 && myFreq ↩
                        == loesungen[b].haeufigkeit){
287
288                         if (myStr.size() == loesungen[b].str.size()){
289                             if (myStr[0] == loesungen[b].str[0] && myStr ↩
                                == loesungen[b].str){
290                                 loesungen[b].haeufigkeit = 0;
291                             }
292                         }
293                         else if (myStr.size() > loesungen[b].str.size()){
294                             if (myStr.find(loesungen[b].str) != ↩
                                string::npos){
295                                 loesungen[b].haeufigkeit = 0;
296                             }
297                         }
298
299                     }
300                 }
301             }
302         }
303
304         i = e-1; // Zur nächsten Range springen
305
306     }
307
308     for (auto &l : loesungen){
309         if (l.haeufigkeit != 0) loesungenGefiltert.push_back(l);
310     }
311
312     cout << "fertig.\nBen[oe]tigte Zeit: " << ((clock() - start) / ↩
        (double)CLOCKS_PER_SEC) << " Sekunde(n).";
313
314     // Ausgabe der Lösungen
```

```
315     cout << "\n\n\n=== L[oe]sungen (" << loesungenGefiltert.size() << ") :";
316     cout << "\nNr.\tH[ae]ufigkeit\tL[ae]nge\tRepetition";
317     cout << "\n-----------------------------------------------------";
318
319     for (unsigned int i = 0; i < loesungenGefiltert.size(); i++){
320
321         stringstream sstream;
322         sstream.clear();
323
324         string number = "";
325         sstream << i+1;
326         sstream >> number;
327
328         while (number.size() < 3){ number = "0" + number; }
329
330         cout << "\n" << number << "\t" << ↩
                loesungenGefiltert[i].haeufigkeit << "\t\t" << ↩
                loesungenGefiltert[i].str.size() << "\t\t" << ↩
                loesungenGefiltert[i].str;
331
332     }
333
334     return 0;
335 }
```

Listing 11: Kompletter Programm-Quelltext

4 Mögliche Probleme

4.1 Arbeitsspeicher

Die Speicherplatzkomplexität meines Algorithmus ist mit $\mathcal{O}(n \cdot (s + l))$[4] linear, was eigentlich sehr gut ist. Aber selbst das stößt (natürlich) an seine Grenzen, wenn z.B. zu viele Lösungen entstehen. Bei meinem Computer (64-Bit, 16 GB RAM) passiert dies bei `E.coli.100000.txt` bei $l = 1, k = 1$ (inwieweit $k = 1$ beim Suchen nach Repetitionen sinvoll ist, sei dahingestellt). Es entstehen einfach zu viele Lösungen, als dass diese gespeichert werden könnten. Ab $k = 2$ ist die Bearbeitung wieder erfolgreich. Dies ist meiner Meinung nach auch völlig ausreichend. Ab $k = 3, l = 3$ kann mein Programm DNS bis zur Länge 550.000 in 200 Sekunden sequenzieren. Wer eine DNS mit einer Länge von über 1.000.000 sequenzieren wil, muss schon mit mehr kommen, als einem Desktop-Rechner. (Der Anspruch, beliebig Lange DNS am Stück und an einem Computer sequenzieren zu wollen, besteht auch in der Realität nicht. Man teilt lange DNS gewöhnlich in kleine Stücke auf und sequenziert diese separat[5].)

[4] $l \hat{=}$ Speicherplatzbedarf für eine Instanz der Klasse **`suffix`**, $l \hat{=}$ Speicherplatzbedarf für eine Instanz der Klasse **`loesung`** (worst-case-Schätzung; natürlich führt nicht jedes Suffix auch zu einer Lösung)

[5] Stichwort: Shotgun-Sequencing

4.2 Uniforme DNS

Ein weiteres (sehr künstliches) Problem kann entstehen, wenn die DNS vollständig oder zu einem großen Teil aus einer einzigen Base besteht. Dies ist der worst-Case und zwingt mein Programm schon nach kurzer Zeit und nicht allzu langer DNS in die Knie, da der zur Verfügung stehende Arbeitsspeicher durch die ungeheure Anzahl entstehender Lösungen vollständig verbraucht wird. Zu diesem Problem kann auch ein fehlerhafter Zufallszahlen-Generator führen; mein ursprüngliches PHP-Programm zur Generierung zufälliger DNS beliebiger Länge hat lange Zeichenfolgen häufig wiederholt und mein Programm damit zum Abstürzen gebracht. So lange man gut generierte oder reale DNS verwendet, wird dieser Fehler nie auftreten. Unnötig zu erwähnen, dass eine DNS aus einer einzigen Base nicht existiert; welche Information sollte darin kodiert sein? Das Lebewesen wäre nicht in der Lage, eigene funktionsfähige Proteine bilden zu können.

5 Beispiele

5.1 Anwendung auf gegebene Beispiele

Datei	**Basen**	l	k	**Laufzeit [Sekunden]**	**Gef. Repetitionen**
chrM.fa.txt	16.571				
		1	1	1,766	9.180
		3	2	0,281	9.159
		5	2	0,296	8.839
		5	4	0,047	3.831
		6	1	1,765	7.874
		7	5	0,031	713
		15	9	0,015	0
E.coli.100000.txt	100.000				
		1	2	9,463	54.544
		3	5	0,781	18.780
		6	3	3,25	33.734
		7	7	0,277	7.508
		9	4	0,14	1.585
		10	5	0,125	36
		12	3	0,109	23
CAGGAGGATTA.txt	11				
		1	2	0	4
		2	2	0	1

5.2 Beispielhafte Bearbeitung

Eine beispielhafte Ausgabe zum oben angegebenen Beispiel $l = 12, k = 3$ auf `E.coli.100000.txt`:

```
+-----------------------------------------------+
| Mississippi v3 - DNS-Sequenzierungs-Programm  |
| Tim Hollmann @ 33.BwInf 2.Runde 3.Aufgabe     |
+-----------------------------------------------+

Eingelesene Zeichenketten-L[ae]nge:100000 (E.coli.100000.txt)

l:  12
k:  3

=== Verarbeitungsschritt 1 : Erstellen des Suffix-Baumes ===
Bitte warten...fertig.
Ben[oe]tigte Zeit: 0.062 Sekunde(n).

=== Verarbeitungsschritt 2 : Auswerten des erstellten Suffix-Baumes ===
Bitte warten...fertig.
Ben[oe]tigte Zeit: 0.047 Sekunde(n).
82 vorl[ae]ufige L[oe]sungen gefunden.

=== Verarbeitungsschritt 3 : Filtern der L[oe]sungen ===
Bitte warten...fertig.
Ben[oe]tigte Zeit: 0 Sekunde(n).


=== L[oe]sungen (23) :
Nr.     H[ae]ufigkeit   L[ae]nge    Repetition
-----------------------------------------------------
001     3               42          ↩
    TGCCGGATGCGCTTTGCTTATCCGGCCTACAAAATCGCAGCG
002     3               19          CAGCGTCGCATCAGGCGTT
003     3               18          GTAGGCCTGATAAGACGC
004     3               15          TTTCAATATTGGTGA
005     3               14          CCCGGACGGTGCTA
006     3               13          CGCCAGCGTCGCA
007     3               12          TCCAGCACTTTC
008     3               12          CCACACCGGTGA
009     3               12          GCTGGCGCTGGC
010     3               12          CGCTGGCGCTGG
011     3               12          ATGGGCGGCGGC
012     3               12          ACGCCGCATCCG
013     3               12          TGCCGCAGTTAA
014     3               12          GGATAAGGCGTT
015     3               12          TTCTGGCTGGCG
016     3               12          GCAGCGCCAGCA
017     3               12          ATGACGCTGGCG
018     3               12          CTGCCGGATGCG
019     4               16          CAGCGTCGCATCAGGC
020     4               15          CTTATCCGGCCTACA
021     4               12          GCCGGATGCGCT
022     5               14          TTATCCGGCCTACA
023     5               12          CTTATCCGGCCT
```

Die Ausgaben meines Programmes zu allen anderen oben angegebenen Beispielen finden Sie im Ordner `/Aufgabe3/Ausgaben` im Format `QuelldateiName_l_k.txt`.

6 Laufzeitkomplexität

Bei diesem Programm besonders interessant ist natürlich das Verhalten bei zunehmender Länge der eingegebenen DNS. Grundsätzlich besitzt der von mir implementierte Algorithmus zur Konstruktion eines Suffix-Baumes eine lineare Laufzeitkomplexität $\mathcal{O}(n)$. Dies bezieht sich aber nur auf die Konstruktion des Suffix-Baumes (den ersten Verarbeitungsschritt). Betrachtet man Abbildung 7, bemerkt man die Zunahme der Laufzeit bei Zunahme der DNS-Länge. Dies liegt daran, dass die beiden anderen Verarbeitungsschritte (Auswerten und Maximalisieren) nicht linear sind. Die Zunahme der Laufzeit ist aber erst bei sehr großen Strings (dem dreifachen des Maximalbeispieles) wirklich spürbar. Das Maximalbeispiel mit 100.000 Zeichen wird in ca. 9 Sekunden verarbeitet; vom Beispiel chM.fa mit 16.000 Zeichen erst gar nicht zu sprechen.

Auf die Speicherplatzkomplexität und die eventuellen Probleme bei $k = 1$ bei langen Strings bin ich in 4.1 bereits eingegangen.

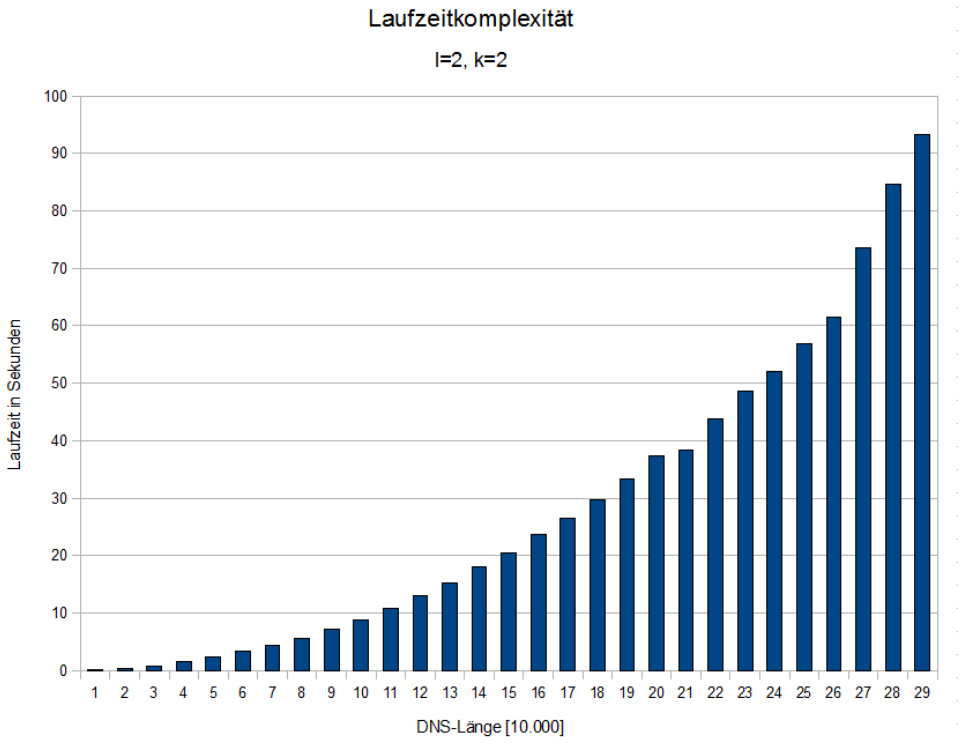

Abbildung 7: Zwar ist die Konstruktionszeit des Suffix-Baumes linear, die beiden anderen Verarbeitungsschritte zwar leider nicht, aber die Zunahme ist erst bei sehr großen Strings wie 300.000 wirklich spürbar.