9. Appendix

Alle Inhalte in diesem Abschnitt stammen von mir, daher bitte mit Vorsicht genießen!

String-Matching

Problemstellung und Begriffe

Problemstellung:
Finden von Textmustern P der Länge $lenPat$ in einem Text T der Länge $lenTxt$, beides sind Zeichenketten, können also auch als Buchstaben-Arrays aufgefasst werden.
Logischerweise gilt $lenPat \leq lenTxt$, die Zeichen von P und T sind alle aus demselben endlichen Alphabet Σ.
Gesucht sind nun alle gültigen Verschiebungen in, mit denen P in T auftaucht, diese sollen in einer Liste/einem Array zurückgegeben werden.
Gesucht sind also alle $sft \in \mathbb{N}$, für die $T[sft, \ldots, sft + lenPat - 1] = P$ gilt, woraus wiederum folgt, dass $T[sft + j] = P[j]$ f.a. $j \in \{0, 1, \ldots, lenPat - 1\}$ gelten muss.

Naives String-Matching

```
NaiveStringMatching(T,P) // P pattern to look for in text T
        lenTxt = length(T);
        lenPat = length(P);
        L = []; // list of matches
        FOR sft=0 TO lenTxt-lenPat DO
                isValid = true;
                FOR j=0 TO lenPat-1 DO
                        IF P[j] != T[sft+j] THEN isValid=false;
                IF isValid THEN
                        L = append(L,sft);
        return L;
```

- Laufzeit: $O((lenTxt - lenPat + 1) \cdot lenPat)$
- Jeder Index in T, für den potentiell ein Match gefunden werden könnte, wird untersucht, indem jeweils die nächsten $lenPat$ Zeichen untersucht werden
- Wenn es nach dem Durchlauf der Schleife nicht zu einer Unstimmigkeit kam, wird der entsprechende Index zu L hinzugefügt, sonst nicht
- Problem: Informationen aus Bearbeitung des Index sft werden bei Index $sft + 1$ nicht weitergegeben

String-Matching mit endlichen Automaten

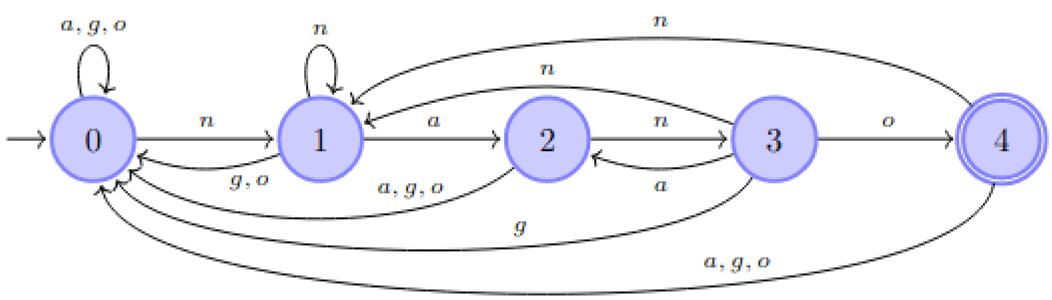
Nun werden deterministische, endliche Automaten (DFA) genutzt, um das Problem des naiven String-Matchings zu überwinden und deutlich bessere Laufzeiten zu erzielen.
Siehe auch AFE #TODO Verweis auf DFA einfügen
Hier wird eine vereinfachte, auf das Problem angepasste Version der DFAs verwendet.
Sei im Folgenden P ein gegebenes Muster der Länge $lenPat$.

- Zuerst muss eine Vorverarbeitungs-/Preprocessingphase stattfinden, in der der DFA konstruiert wird.
- Der DFA hat genau $lenPat + 1$ interne Zustände, st ist die Variable, die immer den aktuellen Zustand speichert.
- Der DFA hat als einzigen akzeptierenden Zustand $lenPat$, Startzustand ist 0.
- Die Übergangsfunktion δ gibt für jeden Zustand st und jedes eingelesene Zeichen $w \in \Sigma$ den nächsten Zustand $\delta(st, w)$ aus.
- Für den Algorithmus wird am Ende nur die Übergangsfunktion δ und die $lenPat$ als Eingabe benötigt, da der Rest nicht relevant ist, wenn der Automat wie oben konstruiert wurde.

```
FSMMatching(T,δ,lenPat)
        lenTxt = length(T);
        L = []; // list of matches
        st = 0; // first state
        FOR sft=0 TO lenTxt-1 DO
                st=δ(st,T[sft]); // get next state
                IF st=lenPat THEN // accepting state
                        L = append(L,sft-lenPat+1);
        return L;
```

- Laufzeit ohne Preprocessing: $O(lenTxt)$

Beispiel-DFA



- Hier ein DFA, der für $\Sigma = \{a, g, n, o\}$ das Muster $P = [n, a, n, o]$ erkennt
- Sämtliche DFA, die hier konstruiert werden, werden ungefähr diese Struktur haben, man kann sich bei der Konstruktion also grob hieran orientieren
- Insbesondere seien angemerkt:
  - Der Startzustand mit entsprechender Schleife für alle Buchstaben aus dem Alphabet, die nicht dem ersten Buchstaben des Musters entsprechen

- Der akzeptierende Zustand, der für $P[0]$ als nächsten Zustand 1 hat
- Der eindeutige Pfad des gesuchten Musters, der bei einem falschen Zeichen entsprechend auf Zustand 0, bzw. 1 für $P[0]$ zurückführt

Rabin-Karp

Idee

- Das Alphabet Σ mit $|\Sigma| = d$ wird durch die Zahlen $\{0, 1, \ldots, d-1\}$ identifiziert, hier wird zur Einfachheit $d = 10$ verwendet, für das lateinische Alphabet wäre $d = 26$ nötig
- Nehme nun p, Dezimaldarstellung des Musters P, und vergleiche diese mit entsprechend langen Abschnitten aus T mit $t_{sft} := T[sft, \ldots, sft + lenPat - 1]$, dann gilt für ein Match an der Stelle sft, wenn $t_{sft} = p$ gilt

Einfacher Algorithmus

```
RabinKarpMatchBasic(T,P)
        n = T.length; m = P.length;
        h = 10^(m-1); // biggest power of 10
        p = 0; t_0 = 0; L = []; // initialise variables
        FOR i=0 TO m-1 DO
                p = 10p + P[i]; // calculate decimal representation of P
                t_0 = 10t_0 + T[i]; // calculate initial value for t_sft with
sft=0
        FOR sft=0 TO n-m DO
                IF p==t_sft THEN // match found
                        L = append(L,sft);
                IF sft<n-m THEN // iff there is another iteration of the loop
                        t_(sft+1) = 10(t_sft - T[sft]h) + T[sft+m]; // next
t_sft value
        return L;
```

- Die Berechnung von t_{sft+1} erfolgt folgendermaßen:
  1. Die höchste Stelle wird abgezogen, dafür ist die Berechnung von `h` notwendig, das die höchste Zehnerpotenz in dem Muster ist
  2. Die nun verbleibende Zahl wird mit 10 multipliziert, um sie zu "verschieben"
  3. Der nächste Eintrag in T wird addiert, er füllt die in 2. frei gewordene Stelle
- Problem:
  Mit wachsender Länge des Musters werden die arithmetischen Berechnungen zu groß, um sie als konstant anzusehen

Weniger einfacher Algorithmus

```
RabinKarpMatch(T,P,q) // q is a prime number
        n = T.length; m = P.length;
        h = (10^(m-1)) (mod q);
        p = 0; t_0 = 0; L = [];
        FOR i=0 TO m-1 DO
                p = (10p + P[i]) (mod q);
                t_0 = (10t_0 + T[i]) (mod q);
        FOR sft=0 TO n-m DO
                IF p==t_sft THEN // potential match
                        b = true;
                        FOR j=0 TO m-1 DO // check for match
                                IF P[j] != T[sft + j] THEN
                                        b = false;
                                        break;
                        IF b THEN
                                L = append(L,sft);
                IF sft<n-m THEN // iff there is another iteration of the loop
                        t_(sft+1) = (10(t_sft - T[sft]h) + T[sft+m]) (mod q);
        return L;
```

- Lösung für Problem: modulo-Rechnung mit einer Primzahl bei t_{sft} und p
- Nun kann es jedoch false positives geben, daher muss im Falle eines potentiellen Matches nochmal überprüft werden, ob es sich wirklich um ein Match handelt

Rekursionsbäume

Grobe Struktur eines Rekursionsbaums:

- Wurzel ist Initialaufruf
- Für jeden rekursiven Aufruf in einer Ausführung erhält der Knoten des Aufrufs einen Kindknoten
- Gibt es keine rekursive Aufrufe, z.B. beim Anker, so ist der Knoten ein Blatt