Algoritmer og Sandsynlighed
Kompendium

Kevin Vinther

7. januar 2024

Indhold

1 Basic Counting Problems

- Pigeonhole (inkl Generalized)
- Permutationer og Kombinationer
- Subsets med repetition
- Pascal's Trekant
- Binomialkoefficienter
- Bevis for binomialsætning vha kombinatorisk argument
- Bevis $n^2 + 1$ delsekvenser med mindst $n + 1$ der er strikt nedad- eller opadgående.

1.1 Pigeonhole

Dueslagsprincippet (pigeonhole principle) er et simpelt princip, men kan bruges til meget i beviser.

Theorem 1 (Dueslagsprincippet)**.**

2 Inclusion Exclusion

3 Discrete Probability

4 Randomized Algorithms

5 Probabilistic Analysis

6 Indicator Random Variables

7 Universal Hashing

I hashing har vi et univers der er meget større end størrelsen på tabellen hvortil hash funktionen finder et index. Dvs. $|U| >> m$, hvor m er størrelsen på tabellen.

Problem med normal hashing: Der kan blive lavet angreb hvor en person der kender hash funktionens værdi kan lave en masse argumenter der alle hasher til det samme index. Vi fikser dette ved at tilfældigt og uafhængigt

af nøglerne, vælge en universal hash funktion. På en universal hash funktion er sandsynligheden for at to værdier hasher til det samme $\leq 1/m$.

Vi kalder det en **kollision**, hvis to værdier hasher til den samme værdi. Vi ved naturligvis fra dueslagsprincippet (Teorem 1) at hvis der er mere end m værdier der bliver hashet, så vil der være mindst én kollision.

7.1 Universal Hashing

Lad $\mathcal{H}$ være en endelig kollektion af hash funktioner således at $h : U \to [m]$ for hver $h \in \mathcal{H}$.

Theorem 2. *Hash kollektionen $\mathcal{H}$ er **universal** hvis følgende holder:*

Lad $h \in \mathcal{H}$ være valgt tilfældigt. Så $\forall k, l \in U$ med $k \neq l \quad p(h(k) = h(l)) \leq \frac{1}{m}$

Theorem 3. *Antag at h er valgt tilfældigt fra en universal kollektino $\mathcal{H}$ af hash funktioner fra $U \to [m]$.*

Antag at vi har brugt h til at hashe et sæt $s \subseteq U$ med $|S| = n$ hvor vi bruger chaining til at løse kollisioner.

Lad $T = T[0], T[1], \ldots, T[m-1]$ være tabellen af linked lister vi får når $T[i]$ er en linked list med de elementer $x \in S$ hvorfra $h(x) = i$.

Derfra gælder følgende:

- *Hvis $k \notin S$, så $E[n_{h(k)}] \leq \frac{n}{m} = \alpha$ hvor $n_{h(k)}$ er længden af $T[h(k)]$*
- *Hvis $k \in S$ så $E(n_{h(k)}) \leq \alpha + 1$*

Bevis. $\forall k, l \in U, k \neq l$ definerer vi

$$X_{kl} = \begin{cases} 1 & \text{hvis } h(k) = h(l) \\ 0 & \text{hvis } h(k) \neq h(l) \end{cases}$$

For et fixed $k \in U$ definerer vi $Y_k = |\{l \in S \setminus \{k\} | h(k) = h(l)\}|$, altså er Y_k antallet af nøgler i $S \setminus \{k\}$ der hasher til samme værdi som k.

Derfor ved vi at $Y_k = \sum_{l \neq k, l \in S} X_{kl}$

$$\begin{aligned} E(Y_k) &= E(\sum_{l \neq k, l \in S} X_{kl}) \\ &= \sum_{l \neq k, l \in S} E(X_{kl}) \\ &\leq \sum_{l \neq k, l \in S} \frac{1}{m} \end{aligned} \tag{1}$$

- Hvis $k \notin S$ så $n_{h(k)} = Y_k$ og $|\{l \in S | l \neq k\}| = |S| = n$ så

$$E(n_{h(k)}) = E(Y_k) \leq \sum_{l \neq k, l \in S} \frac{1}{m} = \frac{|S|}{m} = \frac{n}{m} = \alpha$$

- Hvis $k \in S$, så $n_{h(k)} = Y_k + 1$ og $|\{l \in S | l \neq k\}| = |S| - 1 = n - 1$ dermed

$$E(n_{h(k)}) = 1 + E(Y_k) \leq 1 + \sum_{l \neq k, l \in S} \frac{1}{m} = 1 + \frac{n-1}{m} \leq 1 + \alpha$$

□

Corollary 4. *Ved brug af Universal hashing + chaining, ved at starte fra en tom tabel med m pladser, tager det forventet tid $O(n)$ til at lave en sekvens af `INSERT`, `SEARCH` og `DELETE` operations når $O(m)$ af dem er `INSERT`*

Bevis. Vi indsætter $O(m)$ elementer, hvilket betyder at $|S| \in O(m)$. Dermed $\alpha = \frac{n}{m} \in O(1)$, fordi det er mindre end m. Den forventede længde af hver liste i tabellen er $O(1)$, så hver operation tager $O(1)$ forventede tid, så $O(n)$ for alle operationer. □

7.2 Design af Universal Class

7.2.1 Cormen

Følgende er Cormen's metode til at lave en universal class:

- Vælg et primtal $p \geq |U|$ og antag at $U \subseteq \{0, 1, 2, \ldots, p-1\}$, $\mathbb{Z}_p = \{0, 1, 2, \ldots, p-1\}, \mathbb{Z}_p^* = \{1, 2, \ldots, p-1\}$
- Fordi p er et primtal kan vi løse ligninger $\mod p$.
- $p \geq |U| > m$ så $p > m$.
- For $a \in \mathbb{Z}_p^*$ og $b \in Z_p$ definér $h_{ab}(k) = ((ak + b) \mod p) \mod m$, $h_{ab} : \mathbb{Z}_p \to \mathbb{Z}_m$
- Sæt $\mathcal{H} = \mathcal{H}_{pm} = \{h_{ab} | a \in \mathbb{Z}_p^*, b \in \mathbb{Z}_p\}$

Theorem 5. *Klassen $\mathcal{H}_{pm}$ er universal.*

7.2.2 KT

I KT's metode identificerer vi universet U med tupler af formen $(x_1, x_2, \ldots, x_r)$ for et haltal når $0 \leq x_i < p$ for $i = 1, 2, \ldots, r$. Derudover antager vi at universet er $U \subseteq \{0, 1, 2, \ldots, N-1\}$. Antag ydermere at n er størrelsen af hashtabellen, i den tidligere metode fra Cormen var dette m.

Da vi antager at universet er lavet af tal, kan vi konvertere de tal til binære tal á $log_2 p$ bits. D.v.s, at hvis $p = 11$, og du har tallet 85, som du gerne vil hashe. $log_2(11) = 3.45 \approx 4$. Vi tager 85 i binær: 1010101. Længden af det binære tal er kun 7 ciffrer, derfor tager vi og tilføjer et 0 først (da dette ikke ændrer på tallet). Dermed bliver det 01010101 som har 8 ciffrer. Vi kan dermed dele det op i 2, så vi har en vektor der hedder $(0101, 0101)$. Vi konverterer dette tilbage til heltal, $(5, 5)$

Hvor mange bits skal vi bruge til at repreæsenterer et tal af størrelse N? $\log_2 N$. Hvor mange stykker af længde $log_2 P$ kan du lave? $\frac{\log_2 N}{\log_2 p} \approx r \approx \frac{\log N}{\log n}$. Det er stadig log 2, jeg er bare doven.

Lad $A = \{(a_1, a_2, \ldots, a_r) | a_i \in \{0, 1, 2, \ldots, p-1\} \forall i \in [r]\}$

For $a \in A$ lad

$$h_a(x) = \left(\sum_{i=1}^{r} a_i x_i \right) \mod p$$

$$\mathcal{H} = \{h_a | a \in A\}$$

Theorem 6. *$\mathcal{H}$ er en universal hashfamilie.*

Bevis. Lad $x = (x_1, x_2, \ldots, x_r)$ og $y = (y_1, y_2, \ldots, y_r)$ være distinkte elementer fra U.

Hvad vi nu vil vise er at når $a = (a_1, a_2, \ldots, a_r) \in A$ er valgt tilfældigt, så $p(h_a(x) = h_a(b)) \leq \frac{1}{p}$. Hvis det er højest $1/p$ er det også højest $1/n$, da $p > n$.

Da $x \neq y$ er der et $j \in [r]$ således at $x_j \neq y_j$, altså, der **må** være en koordinat hvorpå de er uenige.

Vi bruger følgende måde at vælge et tilfældigt $a \in A$ på: Først vælg alle $a_i, i \neq j$. Så vælg a_j.

Vi vil nu bevise at for hvert valg af a_i hvor $i \neq j$, så er sandsynligheden for at det sidste valg af a_j ender med $h_a(x) = h_a(y)$ er præcis $\frac{1}{p}$.

$$
\begin{aligned}
h_a(x) &= h_a(y) \\
\sum_{q=1}^{r} a_q x_q &= \sum_{q=1}^{r} a_q y_q \mod p \\
\sum_{q=1}^{r} a_q (x_q - y_q) &= 0 \mod p \\
\sum_{q \neq j} a_q (x_q - y_q) + a_j (x_j - y_j) &= 0 \mod p \\
\sum_{q \neq j} a_q (x_q - y_q) &= a_j (y_j - x_j) \mod p
\end{aligned} \tag{2}
$$

Grunden til vi skriver $\mod p$ til sidst, er fordi, hvis $a \mod p = b \mod p$ så $a = b \mod p$.

Efter vi har fikset a_i for $i \neq j$ har vi $\sum_{q \neq j} a_q(x_q - y_q) = s \mod p$ for some $s \in \{0, 1, 2, \ldots, p-1\}$

Dermed $h_a(x) = h_a(y)$ hvis og kun hvis $a_j(y_j - x_j) = s \; mod p$ da $z = y_j - x_j \neq 0$ siden vi har sikret os at $x_j \neq y_j$ har ligningen $a_j(y_j - x_j) = s \mod p$ en unik løsning $a_j = s(y_j - x_j)^{-1} \mod p$ Vi ved at z ikke er 0, da vi antog at $i \neq j$.

a_j får en tilfældig værdi i $\{0, 1, 2, \ldots, p-1\}$ når $a = \{a_1, a_2, \ldots, a_r\}$ bliver konstrueret. Dermed er sandsynligheden at $a_j = s \cdot (y_j - x_j)^{-1} \mod p$ holder (og dermed $h_a(x) = h_a(y)$) $1/p$. Q.E. FUCKING D. □

7.3 Perfect Hashing

Målet med Perfect Hashing er at få en virkelig god worst case behavior. Vi kan dog kun få det når nøglerne er statiske, så når tabellen er blevet lavet, kan nøglerne ikke ændres. Dette kan eksempelvis bruges i CD/DVD.

Vi ønsker at få $O(1)$ memory access i worst case (dvs. konstant tid selv i worst case), og et lavt memory use.

Perfect hashing bruger to niveauer af hashing hvor begge bruger universal hashing. Så først hashes der til tabel 1, og derefter hashes der til tabel 2, frem for en linked list.

Ved level et finder vi et nøjagte valgt hash funktion $h \in \mathcal{H}$ når h er universal.

I stedet for at bruge en linked list bruger vi en sekundær hashtabel S_j sammen med en associeret hashfunktion h_j for at undgå kollisioner i level 2. Størrelsen af n_j vil være n_j^2 hvor $n_j = |\{x \in S | h(x) = j\}|$

Ved level 1 bruger vi $h \in \mathcal{H}_{pm}$ når $p > |S|$ $(S \subseteq \{0, 1, 2, \ldots, p-1\})$ (dette er familien af hash funktioner defineret fra Cormen)

Nøgler hvor $h(x) = j$ bliver hashet til tabellen S_j af størrelsen m_j ved brug af $h_j \in \mathcal{H}_{pm_j}$

Vores første mål er at vi skal blive sikre på at der er ingen collisions på level 2. Derefter skal vi vise at de forventede hukkommelsesbrug er $O(n), n = |S|$.

Theorem 7. *Antag at vi lagrer n distinkte nøgler i en hashtabel af størrelse $m = n^2$ ved brug af en tilfældig $h \in \mathcal{H}$, når $\mathcal{H}$ er universal. Så er sandsynligheden at der er* ***ingen*** *kollisioner mindst* $1/2$.

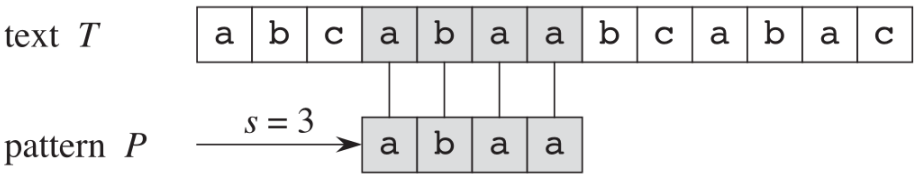
Bevis. □

8 String Matching

8.1 Notation

Jeg tænker ikke at der skal snakkes om det her til eksamen, men følgende er en liste af notation der er nødvendige for forforståelse:

- **Strenge**: Arrays med karakterer (ligesom i programmeringssprog)
- **Shift**: Hvor langt inde i en streng
- $P[1..m]$: Mønster med længde m
- $T[1..n]$: Tekst med længde n
- $T[1..n-m]$: Den tekst vi leder efter. Vi er ikke interesseret i de sidste m, da de er længere end mønsterstrengen.
- **P forekommer med shift s**: Du finder mønstret s karakterer inde i teksten.
- **Validt shift** et shift hvor mønsteret P forekommer
- **Invalidt shift** et shift hvor mønsteret P **ikke** forekommer.

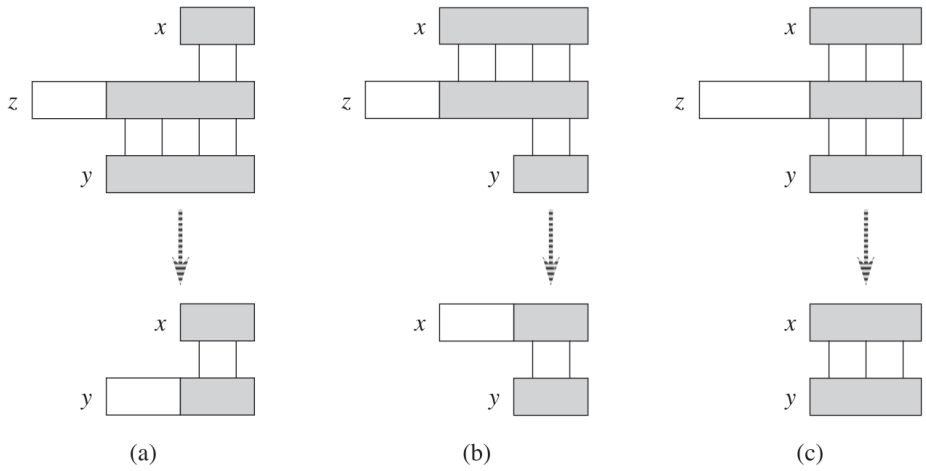


- Σ^* (Sigma-Stjerne) er sættet af alle endelige strenge der kan bliver lavet fra karaktererne i Σ.
- ε, den **tomme streng**, er strengen uden noget indhold. Den er også en del af Σ^*.
- $|x|$ er længden af streng x.
- **Concatenation** af to strenge x og y, skrevet xy har længde $|x| + |y|$ og er karaktererne i x efterfulgt af karaktererne i y.

- **Præfiks** af streng x, denoted $w \sqsubset x$, gælder hvis $x = wy$ hvor $y \in \Sigma^*$, altså, w er en del af streng x i starten af strængen. y er den resterende del af streng x, som ikke er w.
- **Suffiks**: denoted $w \sqsupset x$ omvendt.

Lemma 8 (31.1 (Overlapping-suffix lemma) (Cormen))**.** *Suppose that x, y, and z are strings such that $x \sqsupset z$ and $y \sqsupset z$. If $|x| \leq |y|$, then $x \sqsupset y$. If $|x| \geq |y|$, then $y \sqsupset x$. If $|x| = |y|$ then $x = y$.*

Bevis. Se Figur 1 □



Figur 1: Vi antager at $x \sqsupset z$ og $y \sqsupset z$. De tre dele af figuren illustrerer de tre cases af lemmaet. **(a)** Hvis $|x| \leq |y|$, så $x \sqsupset y$. **(b)** Hvis $|x| \geq |y|$, så $y \sqsupset x$. **(c)** Hvis $|x| = |y|$, så er $x = y$.

Vi antager at tiden det tager for at finde ligheden mellem to strenge er $\Theta(t + 1)$ hvor t er størrelsen af den længste streng. $+1$, til hvis $t = 0$.

8.1.1 Køretids Overview

8.2 Naive Algoritme

- Hvad er den?
- Hvorfor er den dårlig?

Algorithm	Preprocessing Time	Matching Time
Naive	0	$O((n-m+1)m)$
Rabin-Karp	$\Theta(m)$	$O((n-m+1)m)$
Finite Automaton	$O(m\|\Sigma\|)$	$\Theta(n)$
Knuth-Morris-Pratt	$\Theta(m)$	$\Theta(n)$

- Hvad er worst-case?

Den naive algoritme er virkelig det, naiv. **Source Code**:

```
Naive-String-Matcher(T,P)
n = T.length
m = P.length
for s = 0 to n - m
    if P[1..m] == T[s+1..s+m]
        print "Pattern occurs with shift " s
```

8.2.1 Køretid

Den er virkelig skrald. Køretiden er $O((n-m+1)m)$. Dens worst case sker hvis teksten er a^n og mønsteret der ledes efter er a^n (begge er mængder af a'er, på længde hhv. m og n. I dette tilfælde finder den matches hver gnag, og der tager dermed $O(n^2)$ tid.

Der er **ingen** preprocessing tid, da der ikke skal gøres noget før algoritmen kører.

8.3 Rabin-Karp

- Hvad er hovedidéen?
- Hvorfor er den bedre end naive?

Trods at Rabin-Karp har en worst-case køretid på $\Theta((n-m+1)m)$ er dens gennemsnitlige køretid bedre.

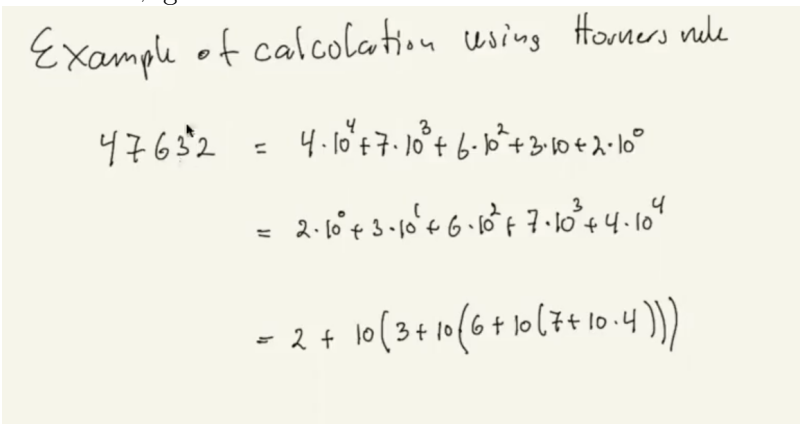
Algoritmen konverterer bogstaverne til tal, i radix-d notation, hvor d er størrelsen på alfabetet, $|\Sigma|$.

I følgende eksempler vil vi gå ud fra at $d = 10$, og $\Sigma = \{0, 1, \ldots, 9\}$. Husk at $P[1..m]$ er mønsteret vi leder efter. Ved rabin-karp skelner vi mellem $P[1..m]$ og p, hvor p er dets decimalværdi. Dvs., hvis $P[1..m] = 1372$, så er $p = 1372$ i decimalværdi. Eksemplet virker forsimplet idet vores alfabet også er tal, men tænk hvis alfabetet var $\Sigma = \{a, b, ...j\}$, i dette tilfælde

ville p ikke være ændret, men $P[1..m] = acgb$. Ydermere er teksten $T[1..n]$'s decimal counterpat t_s. Den bliver udregnet på samme måde. Hvis $t_s = p$ så $T[s+1..s+m] = P[1..m]$.

Vi vil gerne have en måde hvorpå vi kan lave alfabetet om til tal, som vi kan regne på. Hvis vi kan konverterer mønsteret $P[1..m]$ til p på $\Theta(m)$ tid, så kan vi konvertere t_s på $\Theta(n-m+1)$ tid. Til at gøre dette bruger vi **Horner's Rule**, som er meget vigtig at kende, se Definition 1.

Definition 1 (Horner's Rule)**.** Horner's Rule er en regel hvorpå du hurtigt (specielt for computere) kan udregne decimaltal. Dette gør du ved at tage det sidste tal der skal udregnes først, derefter tager du tallet på 10'ernes plads, ganger det med 10^1, etc. indtil du er ved d'ende plads, og ganger det med 10^{d1}. Se følgende billede.



Noget af det smarteste med Horner's Rule, er at, når du går til næste værdi, så kan du udregne det hurtigt uden at tage det hele om igen. Dette giver køretid $\Theta(n-m)$:

$$t_{s+1} = 10(t_s - 10^{m-1}T[s+1]) + T[s+m+1]$$

Forklaring 9. *Skip dette hvis du ikke har meget tid.* $10^{m-1} \cdot T[s+1]$ *fjerner det højeste ciffer. Ved at gange det med 10 skifter du tallet til venstre med en cifferposition. Ved at tilføje* $T[s+m+1]$ *får du det nye, laveste ciffer.*

Problem! p og t_s er muligvis for storre til at de kan være i et computer `word`. Hvis dette er tilfældet, og P indeholder m karakterer, så tager vi tallet **modulo** q. $p \mod q$ bliver udregnet på $\Theta(m)$ tid (størrelsen af p.) Alle t_s værdier i $\Theta(n-m+1)$ tid.

[1] Dette gælder kun i base-10. Rabin-karp kører i base-b. Konverter dette til b^d

Hvilken q skal vi dog vælge? Simpelt! Vælg et primtal således der er plads til $10q$ i én computer `word`. Derefter kan vi udføre alle udregninger simpelt. Ved at bruge modulo-udregning, ændrer Horner's udregning sig til at blive: $t_{s+1} = (d(t_s - T[s+1]h) + T[s+m+1]) \mod q$, hvor $h \equiv d^{m-1} (\mod q)$ er værdien af ciffret 1 i højeste position.

Problem igen! Hvad hvis $p \equiv t_s$, men $P[1..m] \neq T[s+1..n-m]$? Altså, tallene er ens, men de er strengene ikke grundet modulo? Dette kalder vi et **spurious hit**, og er pisse irriterende, men desværre end nødvendig onde. Derfor, når vi finder et **hit** om det er spurious eller ej, så tjekker vi også strengene.

```
RABIN-KARP-MATCHER(T, P, d, q)
 1  n = T.length
 2  m = P.length
 3  h = d^(m-1) mod q
 4  p = 0
 5  t_0 = 0
 6  for i = 1 to m                    // preprocessing
 7       p = (dp + P[i]) mod q
 8       t_0 = (dt_0 + T[i]) mod q
 9  for s = 0 to n - m                // matching
10       if p == t_s
11            if P[1..m] == T[s + 1..s + m]
12                 print "Pattern occurs with shift" s
13       if s < n - m
14            t_{s+1} = (d(t_s - T[s + 1]h) + T[s + m + 1]) mod q
```

Figur 2: Rabin Karp Algoritmen

Worst-case er i samme situation som ved den naive algoritme. Hvis teksten er en del a'er, og det samme med mønstret, så vil vi få en masse hits.

8.3.1 Forventede antal hits

Vi vil gerne finde det forventede antal hits. Vi antager at $\mod q$ agerer som en tilfældig mapping (funktion) fra alfabetet til heltal base q, altså $\Sigma^* \to \mathbb{Z}_q$.

Ydermere antager vi at alle værdierne modulo q er lige sandsynlige, i.e, $p(t_s \equiv p \mod q) = \frac{1}{q}$. Det vil sige at antallet af falske hits er $\frac{O(n)}{q} = O(\frac{n}{q})$. Dette får vi fra antal af hvor mange der modulerer til samme værdi. Hvis der er 10 forskellige bogstaver, og vi er i base-3, så $\frac{10}{3} = 3.\overline{3}$ ca. tal der mapper til det samme.

Den forventede køretid bliver derfor $O(n)+O(m(v+\frac{n}{q}))$ hvor v er antallet af korrekte hits, det's køretid er $O(1)$ og $q \geq m$. Dermed bliver den totale køretid $O(n+m) = O(n)$ da $n \geq m$.

8.4 Finite Automaton Based

- Hvordan laver man en DFA?

Jeg tænker **ikke** du behøver at forklare hvad en DFA er osv. Du får her en kort introduktion, som du bare kan springe over, givet at du forstår finite automata fint.

8.5 Introduktion

Mange algoritmer bygger en Finite Automata (herfra forkortet som DFA), da den er utroligt hurtig til at finde matches. Hver karakter bliver kigget på præcis én gang, og bruger tid $O(1)$ per gang den bliver kigget på. Efter maskinen bliver bygget er matching tiden $\Theta(n)$. Dog kan tiden der bruges til at bygge maskinen være meget stor hvis Σ er stort.

Definition 2 (Finite Automata)**.** A **finite automaton** M, is a 5-tuple $(Q, q_0, A, \Sigma, \delta)$, where

- Q is a finite set of **states**
- $q_0 \in Q$ is the **start state**
- $A \subseteq Q$ is a disntinguished set of **accepting states**,
- Σ is a finite **input alphabet**,
- δ is a function from $Q \times \Sigma$ into Q called the **transition function** of M.

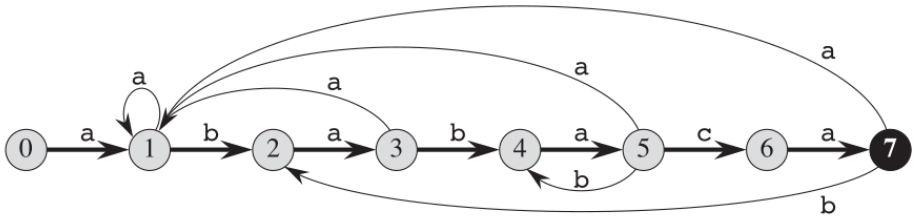
Ved hvert state, læser maskinen et input, og går fra den state, q, til den næste defineret state, $\delta(q, a)$. Hvis q er en del af A, og, efter strengen er blevet "spist", ender maskinen i $q \in A$, så er strengen "accepteret", ellers er den ikke.

Ydermere bliver funktionen ϕ defineret som **Final-state funktion** fra Σ^* til Q således at $\phi(w)$ er den state som M ender op i, efter den scanner strengen w. Så, M accepterer streng w hvis **og kun hvis**, $\phi(w) \in A$.

$$\begin{aligned} \phi(\varepsilon) &= q_0, \\ \phi(wa) &= \delta(\phi(w), a) \text{ for } w \in \Sigma^*, a \in \Sigma \end{aligned} \tag{3}$$

8.6 Streng-matchende automat

For at givet mønster P, vil vi lave en streng-matchende automat som pre-processing skridt før den bruges til at søge efter strengen. Se figur 3 for hvordan vi konsturerer automaten for mønstret $P = ababaca$.



(a)

state	input a	b	c	P
0	1	0	0	a
1	1	2	0	b
2	3	0	0	a
3	1	4	0	b
4	5	0	0	a
5	1	4	6	c
6	7	0	0	a
7	1	2	0	

(b)

i	—	1	2	3	4	5	6	7	8	9	10	11
$T[i]$	—	a	b	a	b	a	b	a	c	a	b	a
state $\phi(T_i)$	0	1	2	3	4	5	4	5	6	7	2	3

(c)

Figur 3: For mere information om figuren, se pp. 997 i Cormen

Definition 3 (Suffiks Funktion)**.** Givet et mønster $P[1..m]$, definerer vi funktionen σ, kaldet **suffiks funktion** korresponderende til P. Funktionen σ mapper Σ^* til $\{0, 1, \ldots, m\}$, således at $\sigma(x)$ er længden af det længste præfix af P som også er et suffiks af x:

$$\sigma(x) = \max\{k : P_k \sqsupset x\}$$

Suffiks Funktionen er **well-defined** (hvert element mapper til noget), da $P_0 = \varepsilon$ er et suffiks er hver streng.

Example 1 (Eksempler på Suffiks Funktionen)**.** Givet mønstret $P = ab$, har vi $\sigma(\varepsilon) = 0$, $\sigma(ccaca) = 1$ og $\sigma(ccab) = 2$. Givet mønstret med længde m har vi $\sigma(x) = m$ hvis og kun hvis $P \sqsupset x$. Fra definitionen af suffiksfunktionen betyder $x \sqsupset y$ også at $\sigma(x) \leq \sigma(y)$.

Vi definerer en streng-matchende automat som korresponderende til et mønster $P[1..m]$ som følger:

- Sættet af states Q er $\{0, 1, \ldots, m\}$. Start staten q_0, og staten m er de eneste accepteret states.
- Transition function δ er defineret ved følgende ligning, for hver state q og karakter a:

$$\delta(q, a) = \sigma(P_q a) \tag{4}$$

Det vil sige, at givet en karakter a, vil vi gå fra state q til længden af det længste præfiks af P, som også er et suffiks af x. Så, altså, hvis a får dig én længere, vil du også gå én state tilbage. Men, hvis du går tilbage til kun at være 2 inde, så er du tilbage på state 2.

Forklaring 10 (Yderligere forklaring)**.** *Vi definerer $\delta(q, a) = \sigma(P_q a)$ fordi vi vil holde fast i det længste præfix af mønsteret P der har matchet strengen T indtil videre.*

*Antag at $p = \phi(T_i)$, så, efter at have læst T_i, så er automatonet i state q. Vi designer δ således at state q fortæller os længden af det længste præfiks af P der er en suffiks af T_i. Det vil sige, i state q, $P_q \sqsupset T_i$ og $q = \sigma(T_i)$. Dette vil også sige at **hvis** $q = m$**, så har vi fundet et match!** Dermed, siden $\phi(T_i)$ og $\sigma(T_i)$ begge er lig q, ser vi at automaten holder følgende invariant:*

$$\phi(T_i) = \sigma(T_i) \tag{5}$$

Dermed, hvis vi er i state q, og automaten læser karakter $T[i+1] = a$, så skal vores transition lede til det korresopnderende længste præfiks af P som er et suffiks af $T_i a$. Den state er $\sigma(T_i a)$.

Fordi P_q er det længste præfiks af P som er et suffiks af T_i, så er det længste præfiks af P som er et suffiks af $T_i a$ ikke kun $\sigma(T_i a)$, men også $\delta(P_q a)$. (Dette bliver bevist senere)

Der er to states vi skal kigge på, den første, $a = P[q+1]$, så er $\delta(q, a) = q + 1$. Den næste, $a \neq P[q+1]$, så skal vi finde et mindre præfiks af P som også er et suffiks af T_i.

Lad os kigge på et eksempel. Streng-matching automaten fra Figur 3 har $\delta(5, c) = 6$, som så er first case, hvor vi bare går videre. Et sekmepl på second case er $\delta(5, b) = 4$. Vi laver denne transition fordi, hvis automaten læser et b når $q = 5$, så $P_q b = ababab$, og det længste præfiks af P som også er et suffiks af $ababab$ er $P_4 = abab$.

Følgende er algoritmen for at lave en finite automata til streng-matching. Sættet af states $Q = \{0, 1, \ldots, m\}$, start staten $q_0 = 0$, den eneste accepting state er m, $m \in A$.

```
Finite-Automaton-Matcher(T, d, m):

n = T.length
q = 0
for i = 1 to n
    q = d(q, T[i])
    if q == m
        print "Pattern occurs with shift" i - m
```

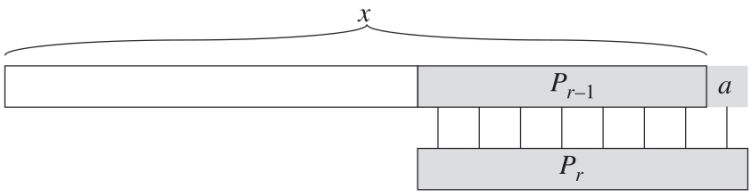
Figur 4:

Transition function er som beskrevet tidligere. Hvis dette ikke er klart, se Forklaring 9.

Det er her nemt at se at køretiden på en tekst-streng af længde n er $\Theta(n)$. Før vi viser pre-processing tid, kigger vi på et bevis for at algoritmen kører som forventet.

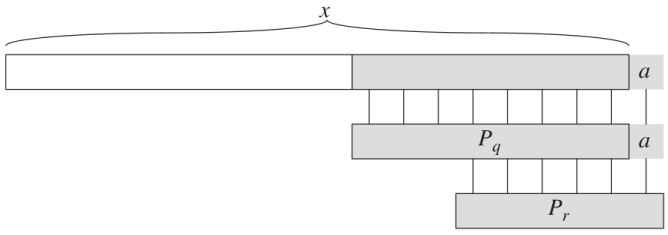
Lemma 11 (Suffix-Function Inequality)**.** *For hver streng x og karakter a, har vi at $\sigma(xa) \leq \sigma(x) + 1$.*

Bevis. Se Figur 5. Hvis $r = 0$, så $\sigma(xa) = r \leq \delta(x) + 1$ er trivielt løst, da $\sigma(x)$ er nonnegativt. Antag at $r > 0$, så $P_r \sqsupset xa$ per definition af σ.



Figur 5: En illustration til beviset af Lemma 10. FIguren viser $r \leq \delta(x) + 1$, hvor $r = \delta(xa)$

Dermed, $P_{r-1} \sqsupset x$, ved at fjerne a fra enden af P_r og fra enden af xa. Dermed $r - 1 \leq \sigma(x)$, siden $\sigma(x)$ er det største k således $P_k \sqsupset x$, og således $\sigma(xa) = r \leq \sigma(x) + 1$ □



Figur 6: Illustration af beviset for Lemma 11. Figuren fiser at $r = \delta(P_q a)$, hvor $q = \sigma(x)$ og $r = \sigma(xa)$

Lemma 12 (Suffix-Function Recursion Lemma)**.** *For enhver streng x og karakter a, hvis $q = \sigma(x)$, så $\sigma(xa) = \sigma(P_q a)$.*

Bevis. Vi ved fra definitionen af σ at $P_q \sqsupset x$. Som vist i figur 6, har vi også $P_q a \sqsupset xa$. Hvis $r = \sigma(xa)$, så $P_r \sqsupset xa$ og, gennem Lemma 10, $r \leq q + 1$. Dermed har vi at $|P_r| = r \leq q + 1 = |P_q a|$. Derfor, $r \leq \sigma(P_q a)$, dermed $\sigma(xa) \leq \sigma(P_q a)$. Vi har dog også $\sigma(P_q a) \leq \sigma(xa)$, siden $P_q a \sqsupset xa$. Dermed $\sigma(xa) = \sigma(P_q a)$ □

Tid til det vigtigste skridt. Vi skal vise at automaten vedligeholder invarianten i ligning 4.

Theorem 13. *If ϕ is the final-state function of a string-matching automaton for a given pattern P and $T[1..n]$ is an input text for the automaton, then $\phi(T_i) = \sigma(T_i)$ for $i = 0, 1, \ldots, n$.*

Bevis. Vi beviser gennem induktion på i.

Ved $i = 0$ er teoremet sandt, da $T_0 = \varepsilon$ dermed $\phi(T_0) = 0 = \sigma(T_0)$. □

Vi antager nu at $\phi(T_i) = \sigma(T_i)$ og beviser at $\phi(T_{i+1}) = \sigma(T_{i+1})$. Lad q være $\phi(T_i)$, og lad a være $T[i+1]$. Så:

$$\begin{aligned}
\phi(T_{i+1}) &= \phi T(_i) \text{ (fra definition på} \mathrm{T}_{i+1} \text{ og a)} \\
&= \delta(\phi(T_i), a) \text{ (fra definitionen på} \phi) \\
&= \delta(q, a) \text{ (af defintiion på q)} \\
&= \sigma(P_q a) \text{ (fra definitionen tidligere)} \\
&= \sigma(T_i a) \\
&= \sigma(T_{i+1})
\end{aligned}$$

8.7 Find Transition Function

Vi vil gerne finde transition funktion. Vi har allerede defineret den tidligere, men vi vil have en algoritmisk metode hvorpå vi kan gøre det.

```
COMPUTE-TRANSITION-FUNCTION(P, Σ)
1  m = P.length
2  for q = 0 to m
3       for each character a ∈ Σ
4            k = min(m + 1, q + 2)
5            repeat
6                 k = k − 1
7            until P_k ⊐ P_q a
8            δ(q, a) = k
9  return δ
```

Figur 7: Algoritmen for at finde transition function på.

Køretiden på algoritmen er $O(m^3|\Sigma|)$.

9 Flows

10 Min-Cut