UPPSALA UNIVERSITET

Föreläsningsanteckningar

Automatateori

Rami Abou Zahra

Inlämningsdatum
September 3, 2022

Contents

1. Mål med kursen

Målet med kursen är att studera matematiska modeller för beräkningar/beräkningsbarhet
På detta vis vill vi kunna förstå vilka problem som överhuvudtaget är beräkningsbara samt nå en förståelse om hur svåra olika beräkningsbara problem är att lösa. Vi vill även bli bekanta med matematiska modeller för beräkningar som tillämpas inom datavetenskap och andra ämnen. Vi kommer även kika lite på användbara algoritmer inom programkonstruktion.

1.1. **Några tillämpningar av teorin.**

De bästa metoderna för översättning av hög-nivå språk såsom Python till maskinkod bygger på teorin om *sammanhangsfria grammatiker*

Det visar sig även att data som följer ett visst mönster kan metoder som är beroende på *ändliga automater* och *reguljära uttryck* användas för att exempelvis söka i datan/databaser.

Det går även att tillämpa teorin för att bestämma om ett konkret problem ens är lösbart algoritmiskt (dvs beräkningsbar)

Fler tillämpningsområden:

- Lingvistik
- Bio-informatik

2. Alfabet och Strängar

2.1. Terminologi.

Definition/Sats 2.1: Alfabet

En ändlig (icke-tom) mängd av tecken/symboler. Brukar betecknas stora sigma (Σ)

Exempel:

Mängden $\{0, 1\}$ är ändlig, icke-tom, och består av tecken/symboler. Alltså är det ett korrekt alfabet enligt vår definition

Mängden $\{\perp, \vdash, \wedge\}$ är ändlig, icke-tom, och består av symboler. Alltså är även detta ett korrekt alfabet

Mängden $\mathbb{N}$ däremot, är ej ändlig, den är icke-tom och består av tecken/symbler, men ej ändlig. Alltså är de naturliga talen ej ett korrekt alfabet.

Definition/Sats 2.2: Sträng

En sträng definieras som en kombination av karaktärerna i alfabetet. En intuitiv fråga som kanske dyker upp är om det finns något supremum (max-längd) på strängarna och svaret är nej. Vi ska kika mer på det snart

Låt $\Sigma = \{0, 1\}$ vara ett alfabet. Då är 0 en valid sträng, men även 1 och 01 och 00 och 11 och $\cdots$

I definitionsrutan sade vi att strängarna kan bli hur stora som helst, och eftersom vi bara kan lägga till fler symboler från vårt alfabet in i strängen så kan mängden av strängar bli oändligt stor.
Men, hur oändligt stor?

Definition/Sats 2.3: Uppräknelig mängd

En mängd A kallas för en **uppräknelig mängd** om det finns en *surjektiv* funktion $f : \mathbb{N} \to A$

Mycket ord att ta in, vi påminner oss om vad surjektion betyder:

Definition/Sats 2.4: Surjektiv

En **surjektiv** funktion är en funktion vars målmängd träffas helt.
Alltså, givet en avbildning (funktion) $f : A \to B$ kommer funktionen att evalueras till alla tal i mängden B. Detta betyder inte nödvändigtvis att hela A används

Ett exempel på en surjektiv funktion är en funktion $f(x)$ så att givet 2 olika x-värden ges samma y-värde. $f(x) = x^2$ är en sådan funktion

Anmärkning:
Om en mängd A är uppräknelig så kan A skrivas som $A = \{a_n : n \in \mathbb{N}\}$

Anmärkning:
Om en mängd A är ändlig så är A uppräknelig. Exempelvis är vår mängd Σ alltid ändlig och därmed alltid uppräknelig.

Kuriosa/Anmärkning:
Varje program kan betraktas som en ändlig sträng över alfabetet $\{0, 1\}$, alltså är mängden av program uppräknelig!

Påstående:
Mängden av funktioner $g : \mathbb{N} \to \{0, 1\}$ är *inte* uppräknelig

Proof 2.1: Bevis av påstående

Antag för motsägelse att mägnden av funktioner $g : \mathbb{N} \to \{0, 1\}$ är uppräknelig.
Då kan vi göra en lista $g_0, g_1, g_2, \cdots$ av alla sådana funktioner.
Defniera nu en funktion h enligt följande:

$$h(n) = \left.\begin{matrix} 0 \text{ om } g_n(n) = 1 \\ 1 \text{ om } g_n(n) = 0 \end{matrix}\right\}$$

Då gäller att $h : \mathbb{N} \to \{0, 1\}$, alltså finns h med i mängden av funktionerna.
Men! Enligt definition av h så finns det inte en funktion g_n så att $g_n = h$ (vi tar ju motsatt värde till g_n) för varje $n \in \mathbb{N}$, men detta betyder att h *inte* finns med i lista, vilket är en motsägelse $\square$

Definition/Sats 2.5

Det finns någon funktion $g : \mathbb{N} \to \{0, 1\}$ som *inte* kan beräknas av något program

Proof 2.2

Låt A vara mängden av funktioner $g : \mathbb{N} \to \{0, 1\}$ och låt P vara mängden av program.
Antag att för varje $g \in A$ så finns $p_g \in P$ som beräknar g.
Låt $g' \in A$ vara godtycklig funktion ur mängden A.

Eftersom P är uppräknelig så finns en surjektiv avbildning $f : \mathbb{N} \to P$.
Men då kan vi definiera en surjektiv funktion $h : \mathbb{N} \to A$ genom att låta $h(n)$ vara g om $f(n) = p_g$ för något $g \in A$ och g' annars

Detta motsäger att A inte var uppräknelig. $\square$

2.2. **Operationer på strängar.**

Låt $v = a_1 \cdots a_n$ och $w = b_1 \cdots b_n$ vara strängar. Vi vill, likt hur man definierar plus och gånger med tal, definiera operationer för strängar:

Definition/Sats 2.6: Sammanfogning

Sammanfogning (sträng1 + sträng2) definieras enligt följande: $vw = a_1 \cdots a_n b_1 \cdots b_n$
Notera! Ordning spelar roll ($a_1 b_1 \neq b_1 a_1$)

Definition/Sats 2.7: Repetition n-gånger

Detta fungerar ungefär som att ta $(\text{sträng})^n$:
$v^n = \underbrace{vvv \cdots v}_{n \text{ gånger}}$

Definition/Sats 2.8: Reversering

Här vill vi härma ”ta ordet baklänges” fast med strängar, alltså:
$v^{rev} = a_n \cdots a_1$ istället för $v = a_1 \cdots a_n$

Givetvis finns det strängar där $v = v^{rev}$, låt $v =$ racecar och se vad som händer om vi reverserar den!

En ganska central grej med både addition och multiplikation inom matematik är ”enheten”, det vill säga 1-1 = 0 (där 0 är enheten för addition) och $2 \cdot \frac{1}{2} = 1$ (där 1 är enheten för multiplikation).
Vi vill försöka härma enheten, det vill säga, finna en sträng sådant att den inte påverkas av operationerna vi tidigare har definierat.
En egenskap hos dessa enheter som vi vill bevara är deras ”längd”, tänk exempelvis enhetsvektorn som har längd 1 för att åstadkomma att vi ej förflyttar oss. Med addition och multiplikation får vi tänka oss tallinjen, 0 i addition betyder att vi inte rör oss i tallinjen på samma sätt som 1an gör med multiplikation.

Definition/Sats 2.9: Längd av sträng

Längden av en sträng $v = a_1 \cdots a_n$ betecknas $|v| = n$ och betyder ”hur många tecken från alfabetet Σ har jag använt?”

Definition/Sats 2.10: Tomma strängen

Vi definierar den **tomma strängen** (ε)
Under operationer beter sig ε på följande sätt:

$$\varepsilon^{\text{rev}} = \varepsilon$$
$$\varepsilon\varepsilon = \varepsilon$$
$$v\varepsilon = v = \varepsilon v$$

Varför kallas den för tomma strängen om det är en enhet? Jo! Vi testar att evaluera längden av ε:

$$|\varepsilon| = 0$$

Det som kan vara ointuitivt är att den tomma strängen *inte* finns i något alfabet omm (om och endast om) det inte specifieras i definitionen, men ε den är en valid sträng i alla alfabet.
Den beter sig lite som tomma mängden, i den att följande gäller för en godtycklig mängd A:

$$\emptyset \notin A \qquad \varepsilon \notin \Sigma$$
$$\emptyset \subseteq A \qquad \varepsilon \subseteq \Sigma$$

Låt x, y vara strängar, då gäller följande:

Definition/Sats 2.11: Prefix

Vi säger att x är en **prefix** till y om det finns en sträng z så att $xz = y$

Exempel:
Låt $y =$ sportbil, $x =$ sport, då är $z =$ bil och ”sport” (x) är prefixet.

Definition/Sats 2.12: Suffix

Vi kallar x för **suffix** till y om det finns en sträng z så att $zx = y$

Exempel:
I föregående exempel med strängen ”sportbil”, så är ”bil” suffixet till ”sportbil”.

3. Språk

Vi har definierat vårt alfabet (exvis siffror), bildat strängar m.h.a operationer (operationer med tal). Kombinerar vi dessa bör vi rimligtvis få ut något som påminner om vektorrum, det vill säga en mängd med tal med några operationer (i vårat fall är mängden tal = mängden strängar, och operationerna de operationer vi definierat överst).

Definition/Sats 3.1: Språk

Ett **språk** över ett alfabet Σ är en mängd strängar över Σ

Några speciella språk:
- $\emptyset$ (tomma mängden)
- $\{\varepsilon\}$
- Σ
- Σ^* (mängden av alla strängar över Σ)

Med liknande resonemang som tidigare kan man visa att:
- Σ^* är uppräknelig
- Mängden av alla språk över Σ är inte uppräknelig

3.1. Operationer på språk.

Låt Σ vara ett alfabet och låt L, L_1, L_2 vara språk över Σ.
Nya språk (över Σ) kan bildas genom:
- Union $L_1 \cup L_2$
- Snitt $L_1 \cap L_2$
- Differens $L_1 - L_2$
- Komplement i Σ^* är $\Sigma^* - L$
- Sammanfogning $L_1L_2 = \{wv : w \in L_1 \wedge v \in L_2\}$
- Repetition:
  - $L^0 = \{\varepsilon\}$
  - $L^{n+1} = L^nL$

3.2. Kleenestjärnatillslutning.

Vi har använt notationen "upphöjt till en stjärna" för att på sätt och vis indikera mängden av alla kombinationer av element. Exempelvis har vi använt Σ^* för att beteckna alla möjliga konstruerbara strängar i ett alfabet.
Av dessa strängar kunde vi skapa språk, detta betecknades med L, och med dessa språk hade vi operationer, precis som vi hade operationer med våra strängar där vi kunde konstruera nya språk givet andra språk, då är frågan om vi kan konstruera mängden av alla språk!

Kleenestjärnatillslutning går ut på följande princip:

$$\sum_{i=0}^{\infty} x^i = x^0 + x^1 + x^2 + x^3 \cdots$$

Där vi på något sätt vill sammanfoga alla språk L^i. Detta gör vi på följande sätt:

$$L^* = \{w : w \in L^n \text{ för något } n \in \mathbb{N}\} = \bigcup_{i=0}^{\infty} L^i = L^0 \cup L^1 \cup L^2 \cup L^3 \cdots$$

L^* uttalas "L-stjärna". Notera att vi har med mängden ε eftersom vi har L^0, vi kan välja att omittera denna genom att använda L-plus istället (L^+).

Exempel:

Låt $\Sigma = \{a, b\}$, $L_1 = \{aa\}$ och $L_2 = \{aa, bb\}$. Då gäller följande:

$$(L_1)^* = \{(aa)^n : n \in \mathbb{N}\} = \{\varepsilon, aa, aaaa, aaaaaa, \cdots\}$$

$$(L_1)^+ = \{aa, aaaa, aaaaaa, \cdots\}$$

$$(L_2)^* = \{\varepsilon, aa, bb, aaaa, aabb, bbaa, bbbb, \cdots\}$$

Låt Σ vara ett alfabet. Vi definierar följande:

Definition/Sats 4.1: Reguljära uttryck

- $\emptyset$ är ett reguljärt uttryck för språket $\emptyset$
- ε är ett reguljärt uttryck för språket $\{\varepsilon\}$
- För varje $\sigma \in \Sigma$ så är σ ett reguljärt uttryck för språket $\{\sigma\}$
- Om α och β är reguljära uttryck för språken L_1 och L_2 så är följande även reguljära uttryck för språken $L_1 \cup L_2$, $L_1 L_2$, L_1^* och L_1^+:
  - $(\alpha \cup \beta)$
  - $(\alpha\beta)$
  - (α^*)
  - (α^+)

Definition/Sats 4.2: Reguljära språk

Om L är ett språk och det finns ett reguljärt uttryck för L så säger vi att L är reguljärt

Exempel:
Låt $\Sigma = \{0, 1\}$
Då är $(0 \cup (1(0^*)))$ ett reguljärt uttryck för språket $L = \{0\} \cup (\{1\}(\{0\}^*)) = \{0\} \cup \{10^n : n \in \mathbb{N}\}$
Eftersom det finns ett reguljärt uttryck för L, så är även språket L reguljärt i detta fall.

Anmärkning:
För notationens enkelhet så reducerar vi paranteser enligt följande paranteskonvention:

- Först * eller $^+$
- Sedan Sammanfogning
- Sist union

Jämför med paranteskonvention för aritmetiska uttryck som ser ut på följande vis:

- Först exponent
- Sedan multiplikation
- Sedan addition

Exempel:
Vi skriver $0 \cup 10^*$ istället för $(0 \cup (1(0^*)))$
(Notera! Det är inte siffran tio, utan siffran ett sammanfogat med noll).

5. Deterministiska finita (ändliga) automater [DFA]

Förr i tiden programmerades datorer genom att man hade ”punch-cards”, det vill säga kort av styvt papper som hade ”hål” som agerade som dagens transistorer gör.
DFA:er påminner lite om hur dessa fungerar, det vill säga att vi ska tänka oss en slags textremsa som inmatning:

a	b	a	b	b	$\cdots$

Där ändligt många inledande rutor innehåller tecken från en sådan textremsa med alfabet Σ.

Vi har också en ”kontrollmekanism”, det vill säga själva DFA:n som vi kan tänka oss har ett ”läshuvud” som avläser en ruta i taget (det är viktigt att precisera, DFA:er läser EN ruta i taget).
DFA:n befinner sig alltid i ett tillstånd, av ändliga antal möjliga. När den avläser en ruta så övergår den till ett nytttillstånd samt flyttar läshuvudet ett steg till höger.
Det nya tillståndet beror (endast) på det tidigare tillståndet samt det just avlästna tecknet (i föregående ruta).

De två ”viktigaste” tillstånd kallas för **starttillståndet** och **accepterande tillståndet**.
Vi antar att en DFA alltid befinner sig i starttillståndet då den sätts igång.
Om en DFA befinner sig i ett accepterande tillstånd då en sträng har avlästs (det vill säga, läshuvudet befinner sig på den första blanka rutan till höger om strängen) så säger vi att DFA:n **accepterar** strängen.
Om strängen inte accepteras, så avvisas strängen.

5.1. Grafisk beskrivning av en DFA.

Vi väljer att representera tillstånden med hjälp av noder, och tillståndsövergångar med pilar:

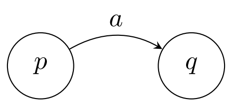

Figure 1.

Figur 1 betyder att om DFA:n befinner sig i tillstånd p och avläser a på textremsan så övergår DFA:n till tillstånd q (och flyttar därmed läshuvudet ett steg till höger).

Anmärkning:
För varje tillstånd och varje tecken från input-alfabetet skall det finnas *preci* en utgående pil som bär detta tecken.

Starttillstånd markeras med en pil mot sig:

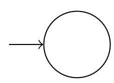

Figure 2.

Accepterande tillstånd markeras med en pil från sig *eller* dubbelrand:

Figure 3.

Exempel:

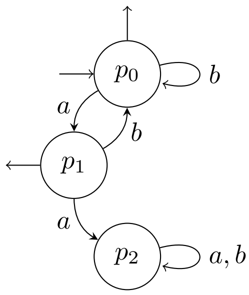

FIGURE 4.

Kom ihåg att en sträng accepteras om vi befinner oss i ett accepterande tillstånd då hela strängen är avläst.
Vilka strängar accepterar av ovanstående DFA? $((b \cup ab)^*(\varepsilon \cup a))$

Definition/Sats 5.1

Antag att M är en DFA.
Mängden av strängar som M accepterar kallas för M**:s språk** och betecknas $L(M)$

Definition/Sats 5.2: Formell definition av DFA

En **DFA** är en 5-tupel $M = (Q, \Sigma, \delta, s, F)$ där:

- Q är en ändlig mängd (tillståndsmängden)
- Σ är en ändlig mängd (inputalfabet)
- δ är en funktion (övergångsfunktion) från Qx$\Sigma \to Q$
- $s \in Q$ (s är Starttillstånd)
- $F \subseteq Q$ (accepterande tillstånden)

För att formellt kunna definiera *acceptans* av en sträng behöver vi en **utvidgad övergångsfunktion**. Denna kan vi löst definiera på följande vis:

Definition/Sats 5.3: Utvidgad övergångsfunktion

En funktion $T(\text{nuvarande tillstånd}, \text{nuvarande symbol på läshuvu}) \to \text{nästa tillstånd}$

6. Icke-Deterministiska Finita (åndliga) automater [NFA]

En NFA är en generalisering av en DFA på följande sätt:

- En NFA kan ha flera Starttillstånd
- En NFA tillåts kunna läsa flera tecken på en gång, dvs vi kan tillståndsövergångar på formen:

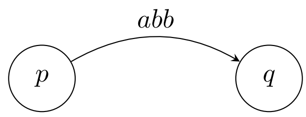

Figure 5.