UPPSALA UNIVERSITET

Föreläsningsanteckningar

Automatateori

Rami Abou Zahra

Inlämningsdatum
September 7, 2022

Contents

1. Mål med kursen 2
1.1. Några tillämpningar av teorin 2
2. Alfabet och Strängar 3
2.1. Terminologi 3
2.2. Operationer på strängar 4
3. Språk 6
3.1. Operationer på språk 6
3.2. Kleenestjärnatillslutning 6
4. Reguljära språk och uttryck 8
5. Determiniska finita (ändliga) automater [DFA] 9
5.1. Grafisk beskrivning av en DFA 9
6. Icke-Determiniska Finita (åndliga) automater [NFA] 11
6.1. Omvandling av en NFA till en DFA som accepterar samma språk 13
7. Övning 18
8. Generaliserade Finita Automater [GFA] 19
8.1. Tillståndseliminationsalgoritmen 19
9. Analys av reguljära uttryck 30
9.1. Minimering av DFA 30

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 $\{\bot, \vdash, \wedge\}$ 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:

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

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 □

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. □

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 (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:

$$(L_1)^* = \{(aa)^n : n \in \mathbb{N}\} = \{\varepsilon, aa, aaaa, aaaaaa, \cdots\}$$
$$(L_1)^+ = \{aa, aaaa, aaaaaa, \cdots\}$$
$$(L_2)^* = \{\varepsilon, aa, bb, aaaa, aabb, bbaa, bbbb, \cdots\}$$

4. Reguljära språk och uttryck

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:

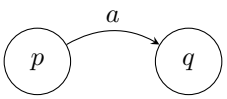

Figure 1.

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:

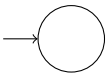

Figure 2.

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

Figure 3.

Exempel:

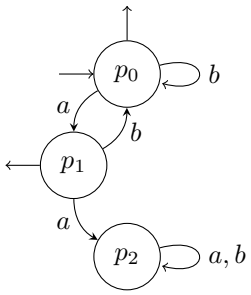

FIGURE 4.

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 $Q\text{x}\Sigma \to Q$
- $s \in Q$ (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-Determiniska 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:

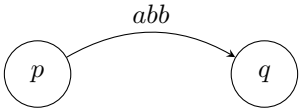

Figure 5.

- En NFA kan göra en tillståndsövergång utan att läsa något (läser tecknet ε)
- En NFA kan ha flera valmöjligheter i en given situation (icke-determinism), dvs vi kan ha följande situation:

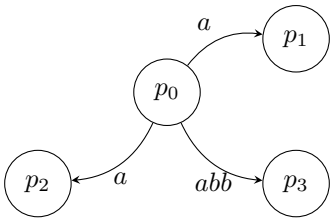

Figure 6.

- Det följer att en NFA (i allmänhet) kan ”hänga sig”

Exempel: (På en NFA)

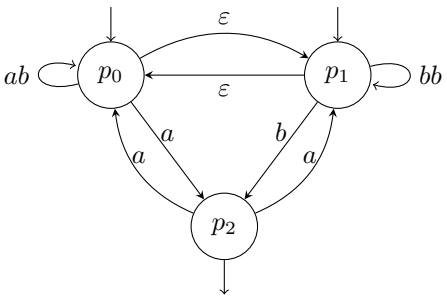

Figure 7.

Definition/Sats 6.1: NFA Acceptans

En NFA accepterar en sträng w om vi kan börja i något starttillstånd och gå från tillstånd till tillstånd genom att avläsa en del av strängen w och slutligen hamna i ett accepterande tillstånd. Det vill säga, om w kan delas upp i delsträngar $w = v_1 v_2 \cdots v_n$ där vi tillåter $v_k = \varepsilon$ *och* det finns tillstånd $p_0, p_1, \cdots, p_n$ där p_0 är ett starttillstånd och p_n är ett accepterande tillstånd samt följande övergångar:

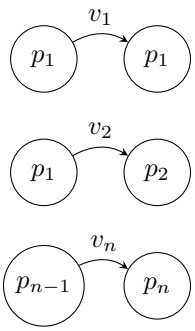

FIGURE 8.

Exempel:
Betrakta föregående exempel och strängarna:

$$aaa$$
$$aaaa$$
$$aabbaa$$
$$abbba$$

Vilka strängar accepteras av NFA:n?

Definition/Sats 6.2

Mängden av strängar som accepteras av en NFA M kallas för M:s språk, och betecknas precis som i DFA fallet med $L(M)$

6.1. **Omvandling av en NFA till en DFA som accepterar samma språk.**

Betrakta följande NFA:

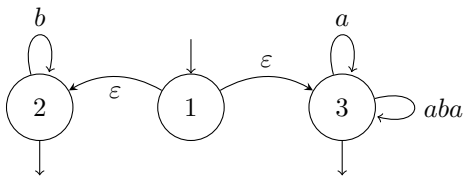

FIGURE 9.

Vi konstruerar först en "icke-glupsk" NFA M' som avlöser högst ett tecken i taget (försöker gå från generaliseringen tillbaks till en DFA) och accepterar samma strängar som M.
$M' =$

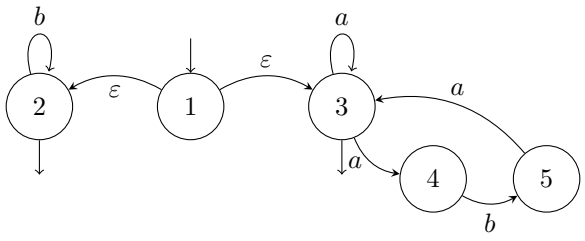

FIGURE 10.

Sedan använder vi den så kallade "delmängdsalgoritmen" för att konstruera en DFA M'' som accepterar samma strängar som M'.
M'':s starttillstånd är mängden av alla tillstånd i M' som kan nås uitan att avläsa några tecken alls.

I vårat fall når vi naturligtvis noden 1 när vi startar, men genom att använda den tomma strängen kan vi nå nod 2 och 3, alltså blir denna mängd $\{1, 2, 3\}$.
Notera att de element från vårat alfabet som vi använder är a och b, därför kommer vi nu följa var vi hamnar med just dessa två bokstäver.

Vi ställer oss frågan, vilka noder som nås från $\{1, 2, 3\}$ genom att *bara* avläsa a. Börjar vi i nod 1 kan vi ta ε till 3, loopa runt till 3 igen, och sedan nå nod 4 genom att avläsa a. Vi kan inte nå nod 5 och avläsa a för att gå tillbaks till nod 3, ty det kräver att vi avläser b. Mängden här blir alltså $\{3, 4\}$.

Vårat diagram ser just nu ut på följande:

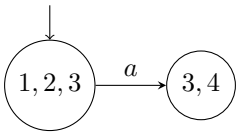

FIGURE 11.

Vi undersöker nu villka tillstånd vi kan nå genom att avläsa b. Vi kan avläsa ε och komma till nod 2, varpå vi kan avläsa b för att återigen komma till nod 2. Mängden blir därför $\{2\}$ och vårat diagram ser ut på följande sätt:

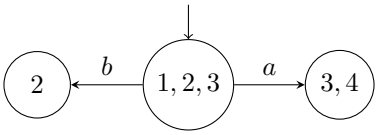

FIGURE 12.

Vi utför denna frågeställning rekursivt tills dess att mängden vi når är Ø, det vill säga nu när vi är på nod 2 så frågar vi oss ”vilka noder når vi härifrån om vi bara använder a” och samma sak för b.
Vi får då följande diagram:

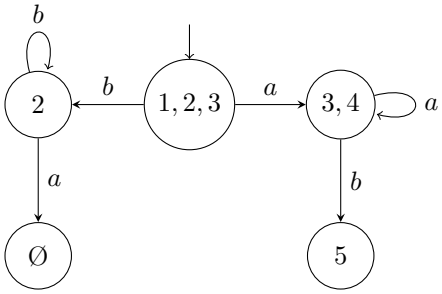

FIGURE 13.

Givetvis fortsätter vi med Ø och 5:

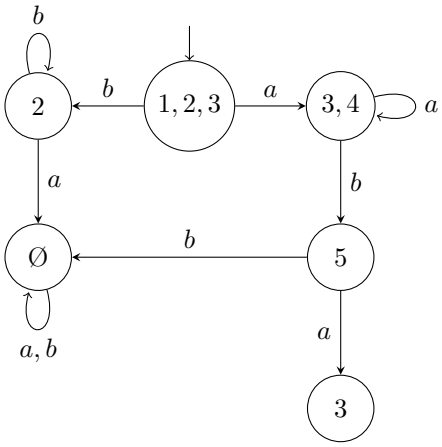

FIGURE 14.

Forstätter vi med 3 får vi slutligen:

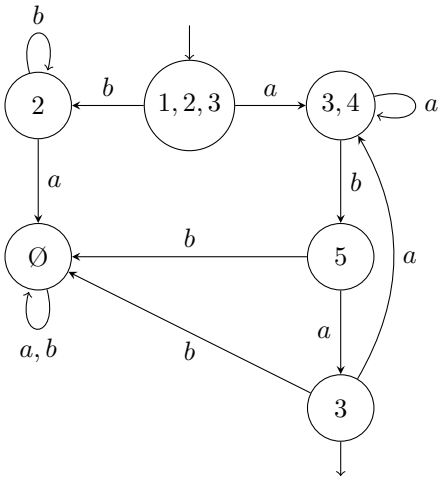

FIGURE 15.

Eftersom inga nya tillstånd (noder) har tillkommit, så är vi klara. Vi kan fråga oss varför denna algoritm alltid terminerar, och då får vi inte glömma att tillståndsmängden bara har ändligt många delmängder.

Notera att de tillstånd i M'' som innehåller något accepterande tilltånd från M' blir de accepterande tillstånden för M'' (det vill säga, i M' (Figure 10) hade nod 3 accepterande tillstånd, så de accepterande tillstånden i M'' blir de noder med enbart mängden 3).

Varför accepterar M'' då samma strängar som M'?
Jo, enligt konstruktion för en sträng w så gäller att:

- Det finns en avläsningsväg av w igenom M' så att man till slut hamnar i ett accepterande tillstånd $\Leftrightarrow$ Det finns en (unik) avläsningsväg av w igenom M'' så att man till slut hamnar i ett accepterande tillstånd.

Exempel:
Beskriv en avläsningsväg för $w = aaabaa$ igenom M' och igenom M'' som slutar med acceptans i båda fallen.

Enligt den beskrivna omvandlingsmetoden så har vi:

Definition/Sats 6.3

För varje NFA M_1 så finns en DFA M_2 som accepterar samma språk som M_1

Definition/Sats 6.4

Fr varje NFA M så finns ett reguljärt uttryk som beskriver $L(M)$

Vi visar nu:

Definition/Sats 6.5

Om L är ett reguljärt språk så finns en NFA (och alltså även en DFA) som accepterar L

Proof 6.1

Induktion över uppbyggnaden av reguljära språk:

- Om $L = \emptyset$ så accepteras L av startnoden men ingen accepterande nod, och vice versa (se Figur 16)
- Om $L = \{\sigma\}$ (godtycklig tecken) så accepteras L av NFA:n med 2 noder, en start och en accepterande, där det krävs just σ för att ta sig igenom (se Figure 17)
- Antag att $L = L_1 \cup L_2$ där L_1, L_2 accepteras av M_1, M_2 respektive. Då accepteras L av (Figure 18)
- Antag att $L = L_1 L_2$ och att L_1, L_2 accepteras av M_1, M_2 respektive. Då accepteras L av (Figure 19)
- Antag att $L = L_1^*$ och att L_1 accepteras av M_1. Då accepteras L av (Figure 20)
- Fallet $L = L^+$ behandlas på liknande sätt som föregående punkt
- Fallet $L = \varepsilon$ behandlas enligt Figure 21

□

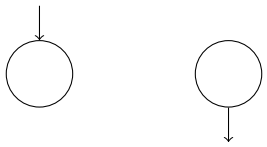

FIGURE 16.

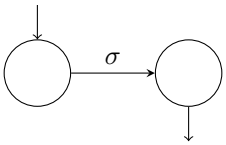

FIGURE 17.

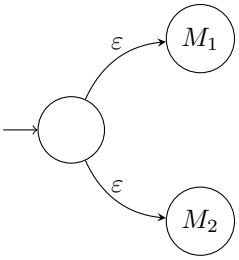

FIGURE 18. "Parallellkoppling"

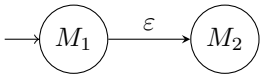

FIGURE 19. "Seriekoppling"

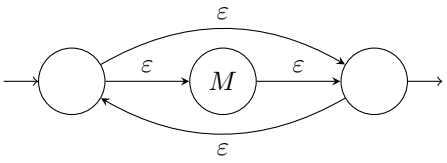

FIGURE 20.

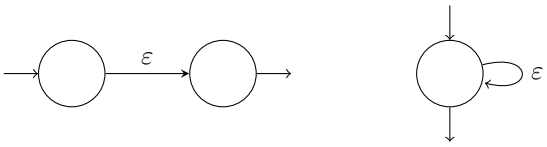

FIGURE 21.

Definition/Sats 6.6: Finit/ändlig automat

Med en **finit/ändlig automat** menar vi en NFA eller en DFA (vilken som)

7. Övning

En NFA för språket som beskrivs av $a(aba \cup aa)^*$:

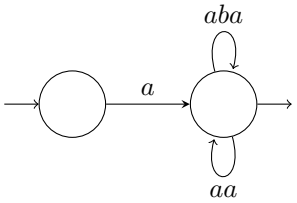

Figure 22.

Skapa en DFA för samma språk, gör först NFA:n icke-glupsk (och namnge tillstånden):

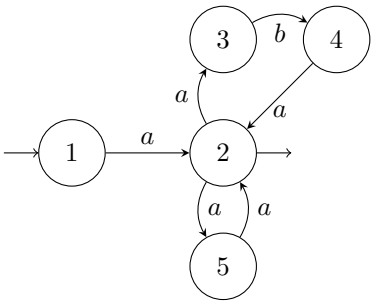

Figure 23.

Med delmängdsalgoritmen får vi följande DFA:

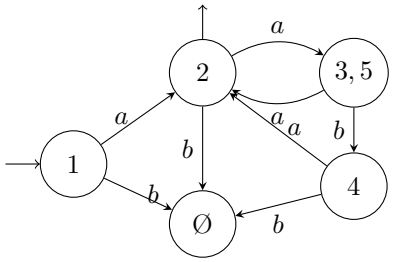

Figure 24.

8. Generaliserade Finita Automater [GFA]

En GFA ser ut (grafiskt) som en NFA *förutom* att vi tillåter att pilarna bär reguljära uttryck (istället för endast strängar)

Exempel:

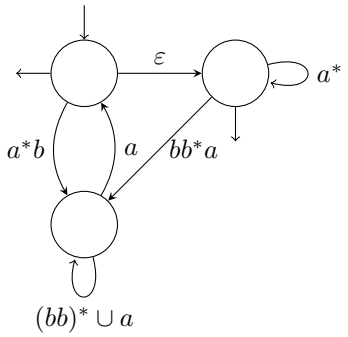

Figure 25.

Om α är ett reguljärt uttryck så tolkar vi:

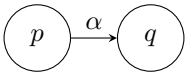

Figure 26.

som att man kan gå från tillstånd p till tillstånd q genom att avläsa en sträng i språket som α beskriver.

Vi använder GFA:er för att omvandla en given NFA/DFA M till ett reguljärt uttryck som beskriver $L(M)$, och då följer att $L(M)$ är reguljärt.

8.1. **Tillståndseliminationsalgoritmen.**

Detta långa namn är namnet på den algoritm som omvandlar en NFA/DFA M till ett reguljärt uttryck för $L(M)$:

Antag att en NFA/DFA M är given, om M saknar starttillstånd eller accepterande tillstånd så $L(M) = \emptyset$ och då beskrivs $L(M)$ av det reguljära uttrycket $\emptyset$ (och vi är klara)

(Fråga om det gäller att om en NFA/DFA inte har start/acc. tillstånd att den är isomorf med en annan NFA/DFA som inte heller har start/acc. tillstånd)

Därför kan vi anta att M har minst ett starttillstånd och minst ett accepterande tillstånd.

Algoritmen går därmed ut på följande sätt:

- Lägg till ett *nytt* starttillstånd och pilar som bär ε från detta nya starttillstånd till alla de gamla starttillstånd som nu förlorar sin status som starttillstånd (här förändras *inte* vilka strängar som accepteras)
- Lägg till ett *nytt* accepterande tillstånd och pilar som bär ε till detta nya accepterande tillstndet **från** alla de gamla accepterande tillstånden som nu förlorar status som accepterande tillstånd
- Eliminera alla de gamla tillstånden, ett för ett, samt ersätt gamla pilar med nya enligt följande mönster (där krysset markerar det tillstånd som elimineras):

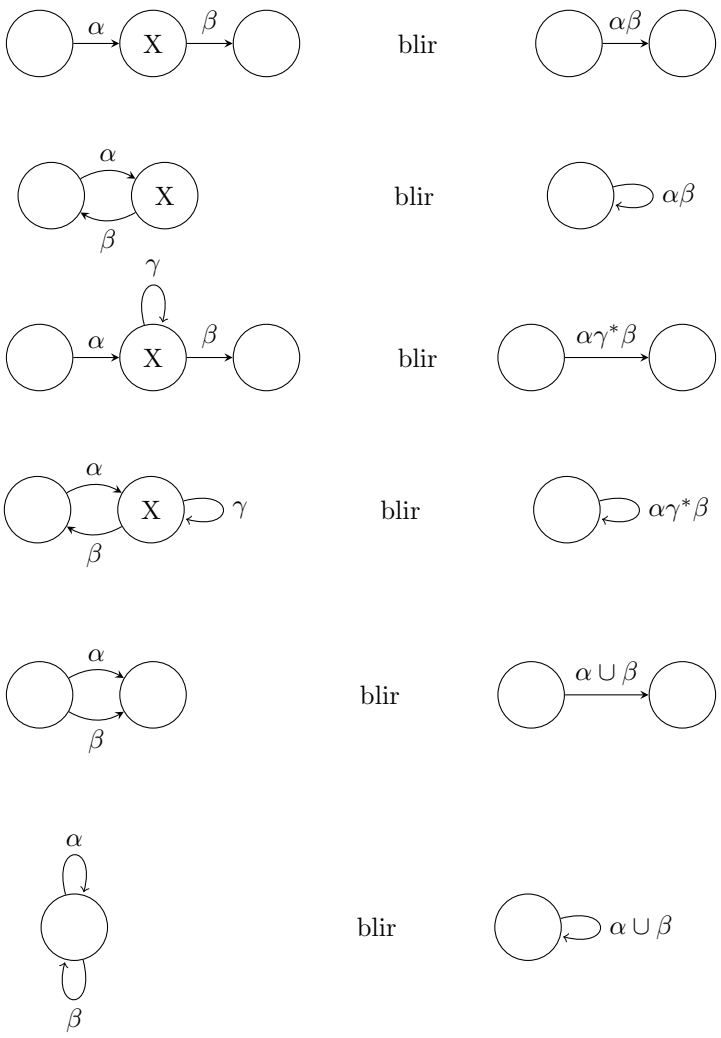

FIGURE 27.

När alla elimineringar har upprepats tills det inte går att förenkla mer, så har vi en GFA på formen:

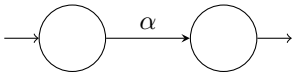

FIGURE 28.

Där α är ett reguljärt uttryck som beskriver $L(M)$

Från algoritmen följer det:

Definition/Sats 8.1

Varje språk som accepteras av någon finit automata är reguljärt.

Exempel:
Betrakta följande NFA samt omvandla till ett reguljärt uttryck:

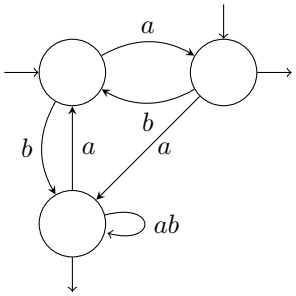

FIGURE 29.

Vi börjar med att lägga till ett nytt starttillstånd samt ett nytt accepterande tillstånd och kopplar de till de redan existerande med ε:

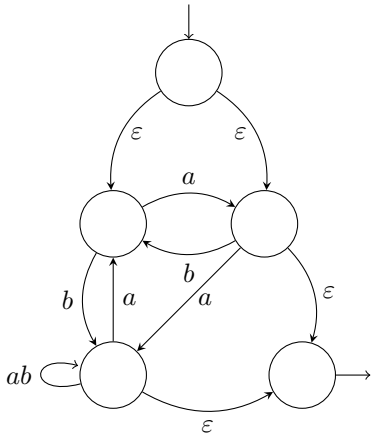

FIGURE 30.

Det finns lite olika strategier för att välja vilken nod man kan börja med att elimineras, personligen föredrar jag att välja den nod med minst antal anslutningar, det gör saker lite lättare.
Om det är så att det endast finns en nod som *inte* har en direkt koppling till det accepterande tillståndet, föredras det eftersom det blir lättare att se vilka kopplingar som måste ersättas. Väljer man en nod som är direkt kopplad till det accepterande tillståndet så finns det flera andra pilar som kommer från andra noder som man kan råka blanda ihop.

Det är många små steg som görs åt gången under elimineringsprocessen, vi ska försöka köra detta exempel med detalj.

Vi börjar med att markera alla noder, detta kommer göra saker lättare att förklara och är inget måste.

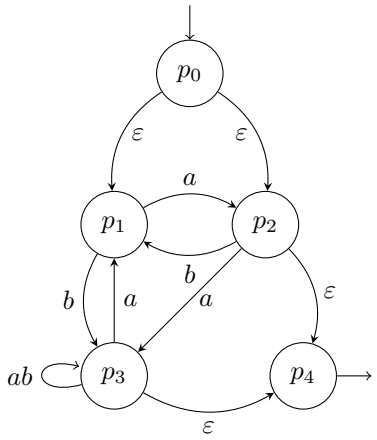

FIGURE 31.

Notera att ”den längsta vägen” till det accepterande tillståndet är via p_0, p_1, p_3, p_4
Därför väljer vi att eliminera den noden i den vägen som inte är direkt kopplad till det accepterande tillståndet, det vill säga p_1:

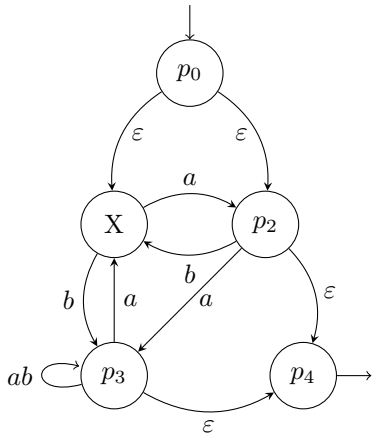

FIGURE 32.

Om vi nu ska ta bort denna nod, så måste vi fortfarande kunna avläsa samma strängar för at komma till ett accepterande tillstånd.
Strategin här är att notera vilka vägar som är möjliga att ta till de noder som är kopplade till p_1.

Tidigare kunde jag använda εa för att gå från p_0, p_1, p_2. Eftersom ε är den tomma strängen, så räcker det alltså med att säga ”tidigare kunde jag läsa a för att komma till p_2”.
Tar vi bort p_1 måste vi kunna gå från p_0 till p_2 genom att också läsa a. Vi kan lägga till den egenskapen genom följande:

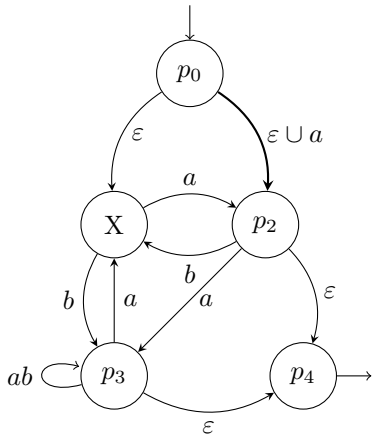

FIGURE 33.

En relevant fråga man kanske ställer sig är ”varför tar vi inte bor vägen a” från p_2 till p_1? Detta eftersom vi tidigare kunde avläsa εba för att ”loopa” runt p_2, och detta måste vi naturligtvis behålla:

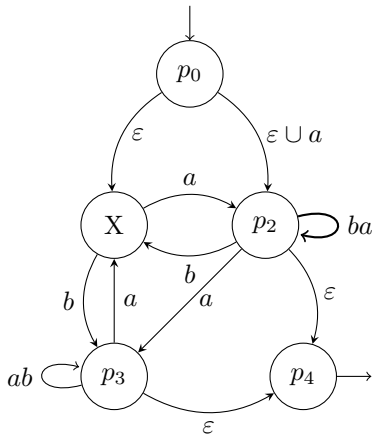

FIGURE 34.

Är vi redo att ta bort de vägarna nu? Nästan. Tidigare kunde vi komma till p_3 via vägen p_0, p_2, p_1, p_3 med strängen εbb eller εa. Vi måste alltså kunna gå från p_2 till p_3 med den strängen, vi gör därför följande:

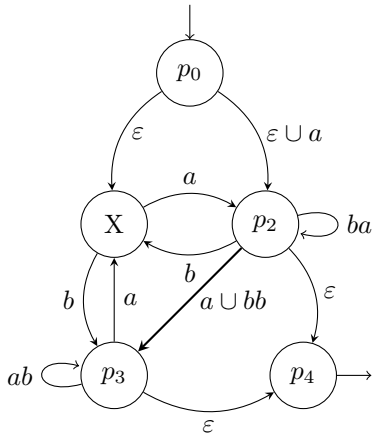

FIGURE 35.

Nu kan vi plocka bort b-pilen, eftersom vi har återskapat alla vägar:

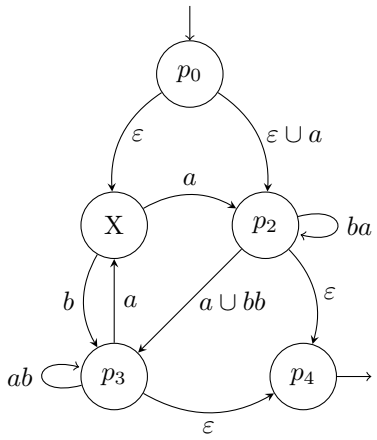

FIGURE 36.

Nu kan vi alldeles strax få bort a. Notera att vi kunde från p_3 nå p_2 genom p_1 genom att avläsa aa (gå upp ett steg, sedan till höger), eftersom vi tar bort p_1 behöver vi alltså ha en pil från p_3 till p_2 med aa. För er som undrar varför vi inte ska ta hänsyn till det a som redan står på en pil mellan p_1 och p_2 så har vi redan gjort det när vi bytte till $\varepsilon \cup a$

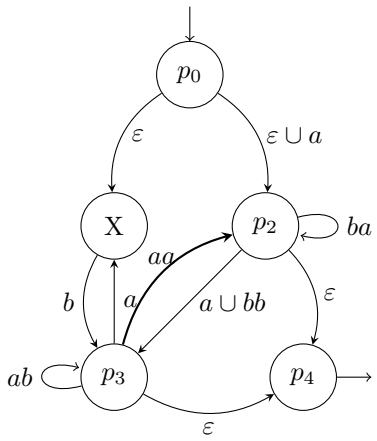

FIGURE 37.

Nästan där! Vi har kunnat läsa εb för att ta oss från p_0 till p_3 genom p_1, vilket är samma sak som att läsa b. Vi behöver alltså en koppling från p_0 till p_3 med bara b:

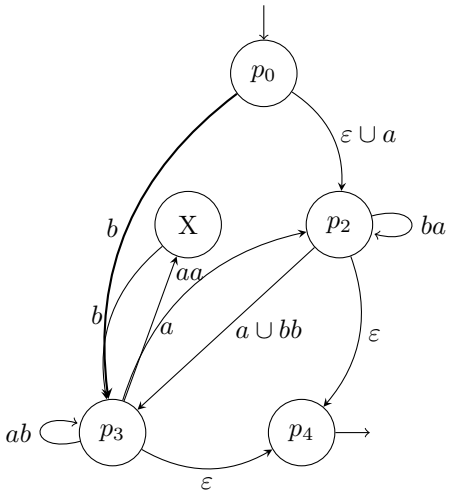

FIGURE 38.

Tidigare har vi kunnat avläsa $\varepsilon bababababab$ för att loopa mellan p_1 och p_3 (trots att p_3 redan har en loop för ab), vi har redan replikerat att vi kan ta b för att komma till p_3, och eftersom loopen ab redan finns, behöver vi inte replikera den.
Vi kan därför nu plocka bort pilarna a, b:

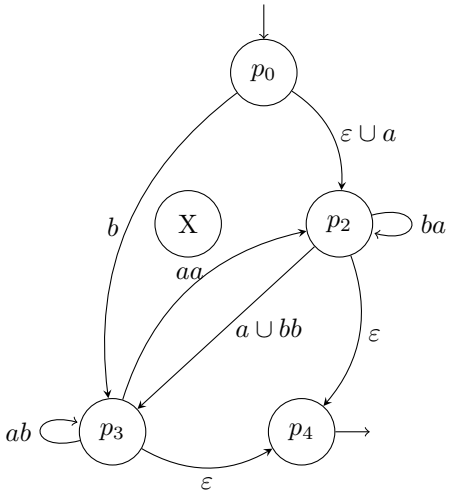

FIGURE 39.

Notera att noden vi vill ta bort är fri! Den har inga kopplingar till andra noder, och vi kan därför ta bort den:

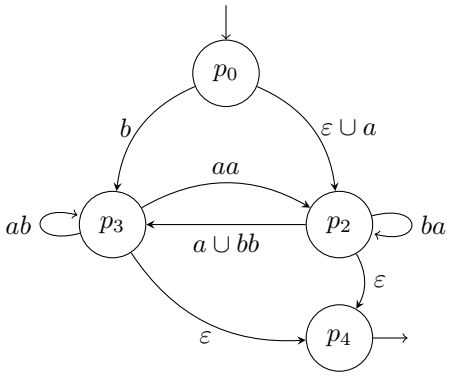

FIGURE 40.

Vi repeterar nu processen för antingen p_3 eller p_2 (det spelar ingen roll, de har precis samma antal kopplingar och kopplas med samma avstånd till det accepterande tillståndet). Vi väljer p_3.

Det som gör denna nod lite speciell är att den har en loop, och detta måste vi ta hänsyn till varje gång vi kopplar bort en koppling. Fördelen är att vi kan notera detta med $(ab)^*$ och därmed inkludera de fall då vi inte alls väljer att snurra i loopen (eftersom $\varepsilon \subseteq (ab)^*$)

Vi noterar att vi från p_2 kan avläsa $((a \cup bb)aa)^*$ för att loopa p_2. Men! Eftersom vi når p_3 (och därmed kan avläsa ab och loopa p_3) så måste vi ha med det i vår nya loop:

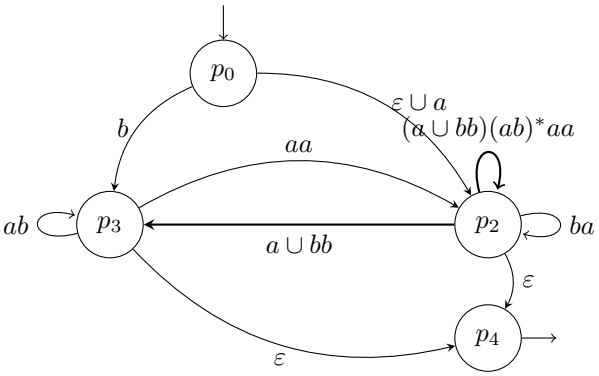

FIGURE 41.

Eftersom vi inte kan nå något mer än p_4 via ε efter att ha avläst $(a \cup bb)(ab)^*$ kan vi nu ta bort den pilen (fetmarkerad i Figure 41) och skapa en pil från p_2 till p_4:

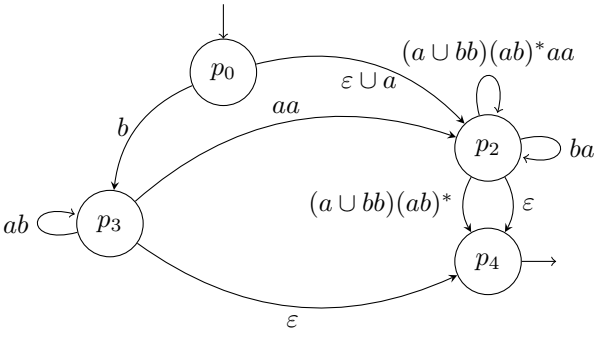

FIGURE 42.

Vi vill nu försöka få bort pilen över den vi just tog bort, den går från p_3 till p_2 så vi måste se till att de strängar vi kan läsa för att komma till p_3 leder till p_2
Vi noterar att vi kan avläsa baa för att komma till p_2 från p_3, men vi kan ju också läsa $babaa$ och oändligt antal fler loopar runt ab, så vi skapar en pil från p_0 till p_2 med $b(ab)^*aa$:

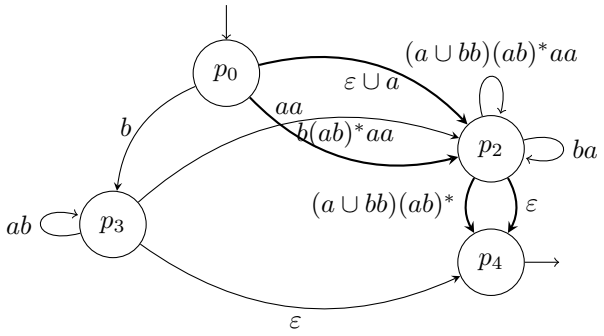

FIGURE 43.

Men vi har nu 2 olika pilar som går från samma till samma nod, vi kan slå ihop dessa till en enda:

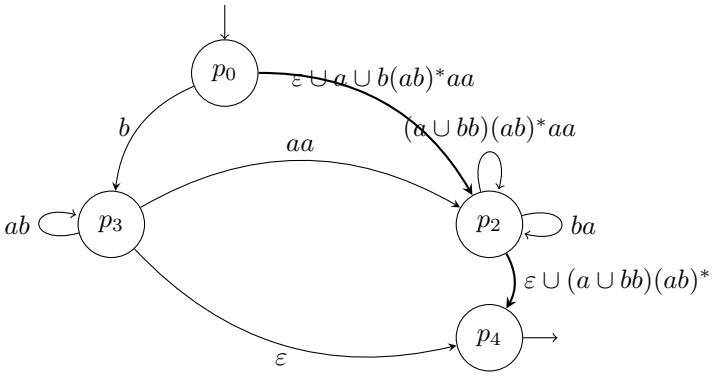

FIGURE 44.

Notera att vi kan ta vägen $b\varepsilon$ (men också givetvis loopa runt p_3 med ab hur mycket som helst eller hur lite som helst), detta bevarar vi genom att dra en pil från p_0 till p_4 med $b(ab)^*$.
Vi kan även ta bort pilarna från p_0 till p_3, pilarna från p_3 till p_2, samt pilarna från p_3 till p_4:

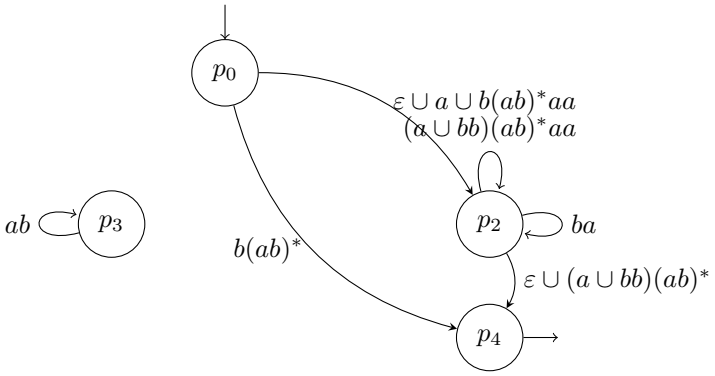

FIGURE 45.

p_3 är nu ensamt och vi kan ta bort den:

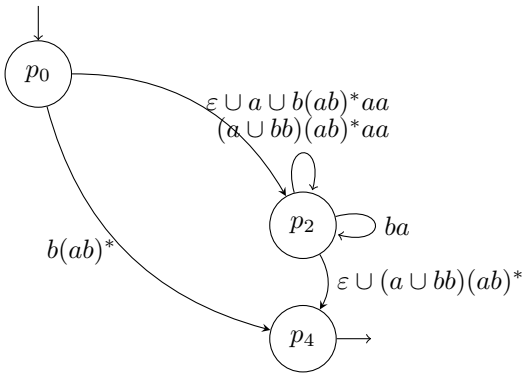

FIGURE 46.

Notera att vi har två loopar på p_2, detta kan vi förenkla till en genom att använda union:

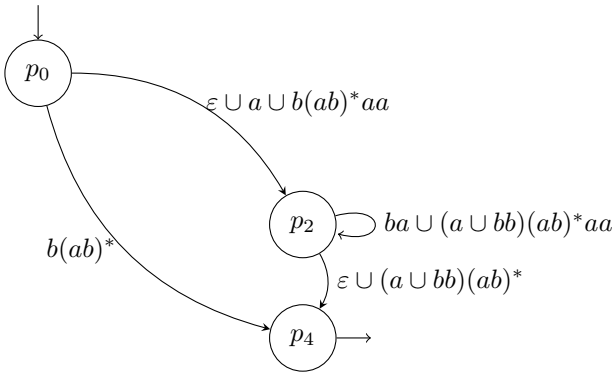

FIGURE 47.

Dags att börja eliminera p_2! De pilar som går in i p_2 är en pil från p_0, och de pilar som går ut är en pil till p_4, alltså bör vi kunna avläsa strängen $\varepsilon \cup a \cup b(ab)^*aa\varepsilon \cup (a \cup bb)(ab)^*$ för att ta oss från p_0 till p_4 via p_2, men vi har ju även en loop, alltså måste vi slänga in $(ba \cup (a \cup bb)(ab)^*aa)^*$:

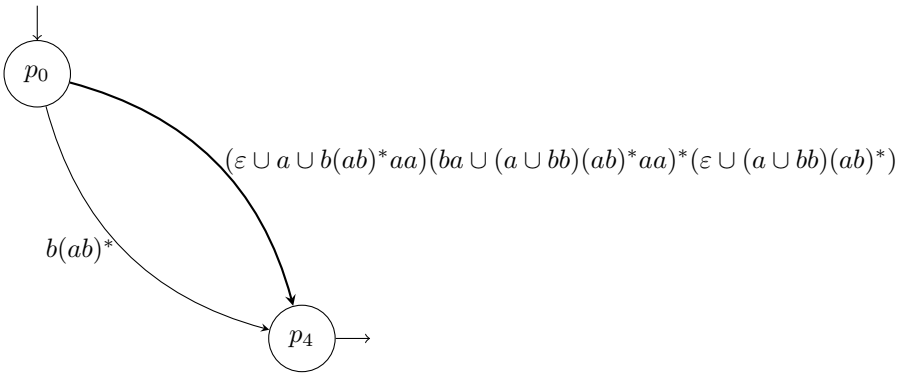

FIGURE 48.

Vi har nu 2 alternativa vägar från p_0 till p_4, antingen vägen med det långa reguljära uttrycket, eller den korta med $b(ab)^*$. Har vi en nod som går till en annan med 2 olika pilar kan dessa kombineras med union. Man kan tänka på union som ett ”eller”, det vill säga $b(ab)^* \cup Q$ kan översättas till ”$b(ab)^*$ eller Q”

Gör vi detta får vi:

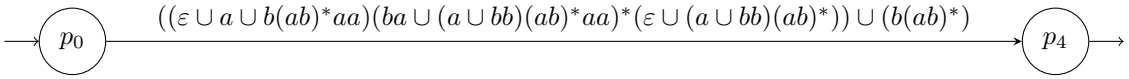

FIGURE 49.

Vi har nu ett starttillstånd, ett accepterande tillstånd, och inga andra noder. Bara en pil som kopplar de 2 tilltånden, och texten över den är det reguljära uttrycket för den ursprungliga NFA:n

9. Analys av reguljära uttryck

Om L är ett reguljärt uttryck $\Leftrightarrow$ L accpeteras av någon NFA $\Leftrightarrow$ L accpeteras av någon DFA. Detta väcker frågan, vad är den minsta DFA:n vi kan skapa givet ett reguljärt uttryck?

9.1. **Minimering av DFA.**

Definition/Sats 9.1

En DFA M är *minimal* om det inte finns en DFA som accepterar samma som M och har färre tillstånd än M

Definition/Sats 9.2

Låt M vara en DFA och w en sträng (w är en sträng över M:s alfabet).
Vi säger att w *driver* DFA:n M från ett tillstnd p till ett annat q om när M startas i tillstånd P med w på tapen så stannar M i tillstånd q

Exempel:

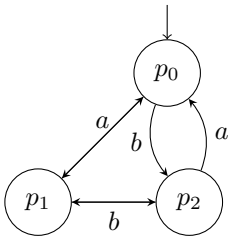

Figure 50.

Här driver strängen aaa M från p_2 till p_0. Strängen ab driver M från p_2 till p_2

Om ingen sträng driver M från starttillstånd till q så kallas q för *isolerat* tillstånd.

Exempel:

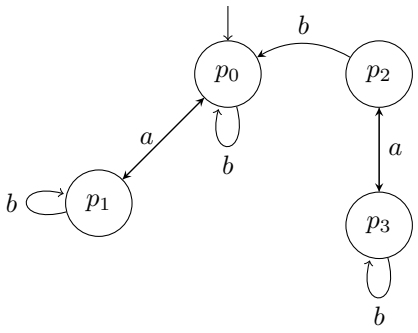

Figure 51.

Notera här att p_2 och p_3 är isolerade tillstånd! Då kan vi lika gärna plocka bort de utan att det påverkar vilka strängar som accepteras.

Definition/Sats 9.3: Reduktion av ett tillstånd (om möjligt)

Låt M vara en DFA (utan isolerade tillstånd) och antag att p och q är olika tillstånd i M.
Antag att följande villkor gäller för varje sträng w (även ε):

- w driver M från p till ett accepterande tillstånd om och endast om w driver M från q till ett accepterande tillstånd

Låt nu M' vara som M utom att q tas bort, och varje övergång i Figure 52 ersätts med Figure 53.
Då gäller $L(M') = L(M)$ och M' har färre tillstånd än M

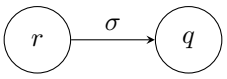

FIGURE 52.

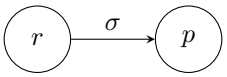

FIGURE 53.

Detta påminner lite om tillståndeliminering, men vi vill inte skapa reguljära uttryck utan det är ok att bara koppla bort pilarna från en nod till en annan.

Definition/Sats 9.4: Särskiljandealgoritmen för minimering av en DFA

Exempel:
Låt M vara följande DFA: (**KOPIERA FRÅN ANTECKNINGAR**)

Här är det enklare att skriva en tabell över tillståndsövergångarna:

	1	2	3	4	5	6	7
a	3	3	3	3	3	3	7
a	1	1	6	4	4	5	7

Nivå	Sönderdelningar
1	$\{7\}\,\{1,2,3,4,5,6\}$ (initial & accept. tillst)
2	$\{7\}\,\{1,2,3,4,5\}\,\{6\}$
3	$\{7\}\,\{1,2,4,5\}\,\{3\}\,\{6\}$
4	$\{7\}\,\{1,2,4,5\}\,\{3\}\,\{6\}$

Hur bildas nivå $n+1$ efter att nivå n är bildad?

- Om två tillstånd tillhör olika delar på nivå n, så gör de även det på nivå $n+1$
- Antag att p och q är tillstånd som tillhör samma del på nivå n, p och q ska särskiljas, dvs placeras i olika delar på nivå $n+1$ **om** det finns ett $\sigma \in \Sigma$ som driver DFA:n från p till en annan del på nivå n än σ som driver DFA:n från q

Efter att vi kom till nivå 4 såg vi att vi inte fick några ändringar, nu kan vi konstruera en DFA M' som är minimal.
M':s tillstånd är delarna på den sista nivån (4). (**KOPIERA FRÅN ANTECKNINGAR**)
Starttillståndet hos M' är den del/mängd som innehåller M:s starttillstånd. Samma gäller för accepterande tillstånd.
Givet en del/mängd P och tecken σ så lägg till en övergång $P \to Q$ via σ där Q är den unika del sp att det finns $p \in P$ och $q \in Q$ så att övergången från $p \to q$ via σ finns i M

Övning:
Betrakta följande tillståndsövergångstabell:

	1	2	3	4	5	6	7	8
a	1	1	2	3	5	7	8	2
b	1	4	6	5	5	5	6	6

Tillstånd $4, 6, 7$ är accepterande och tillstånd 4 är starttillståndet. Vi utför algoritmen:

Nivå	Sönderdelningar
1	$\{1, 2, 3, 5, 8\}\,\{4, 6, 7\}$
2	$\{1, 4, 5\}\,\{2, 3, 8\}$