Strängar

Teckenkodning

ASCII - *American Standard Code for Information Interchange*

- Tecken kodas med 7 bitar
- Exempel:

$$a = 97_{10} = 1100001_2$$

ASCII

- Det enda språket som ASCII kan koda är Engelska (a-z)
- 7 bitar eftersom åttonde biten användes för feldetektering i överföringar (parity bit)
- Många kontrolltecken som inte används längre (33 stycken totalt)
- Vissa kontrolltecken har bytt betydelse
- T.ex. NUL (0) användes bl.a. för att reservera utrymme på hålkort som kunde fyllas i senare

Andra teckenkodningar

- Flera 8-bitars teckenkodningar har utvecklats för att utöka ASCII, men de var sällan kompatibla med varandra
- T.ex. i ISO/IEC 646 finns å, ä och ö med
- 1980 påbörjades arbetet med *Unicode-standard*

Unicode

Unicode

- Unicode-standarden har som mål att kunna koda alla världens språk
- Definierar över 150000 *codepoints*, t.ex:

å = U+00E5, 指 = U+6307, 😃 = U+1F603

- Numera standard på internet

Unicode

- Standarden definierar även flera olika kodningar
- UTF-8, UTF-16 och UTF-32
- Specificerar hur enskilda codepoints ska kodas binärt

UTF-8

- UTF-8 är den vanligast kodningen av Unicode
- Det är en *längvarierande* textkodning, 1-4 *oktetter* (bytes)
- Bakåtkompatibel med ASCII (första 127 kodningarna är identiska)
- Enkel att iterera över
- Enkelt att hitta början på ett tecken

U+0000 - U+007F	0yyyzzzz			
U+0080 - U+07FF	110xxxyy	10yyzzzz		
U+0800 - U+FFFF	1110wwww	10xxxxyy	10yyzzzz	
U+010000 - U+10FFFF	11110uvv	10vvwwww	10xxxxyy	10yyzzzz

UTF-16 & UTF-32

- UTF-16 liknar UTF-8 men använder en eller två dubbeloktetter
- UTF-32 kodas i 32 bitar och motsvarar codepoints direkt
- Har fördelen att det går att indexera

Versaler/gemener i Unicode

- Unicodestandarden definierar även hur man omvandlar versaler till gemener
- T.ex. ß → SS

Sortering i Unicode

- Hur ska vi sortera följande lista i alfabetisk ordning?

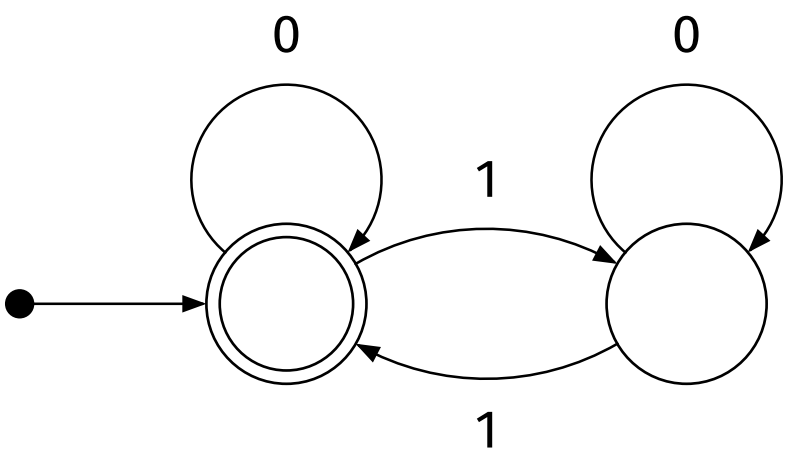
  - Stockholm
  - Aarhus
  - Zürich

Automater

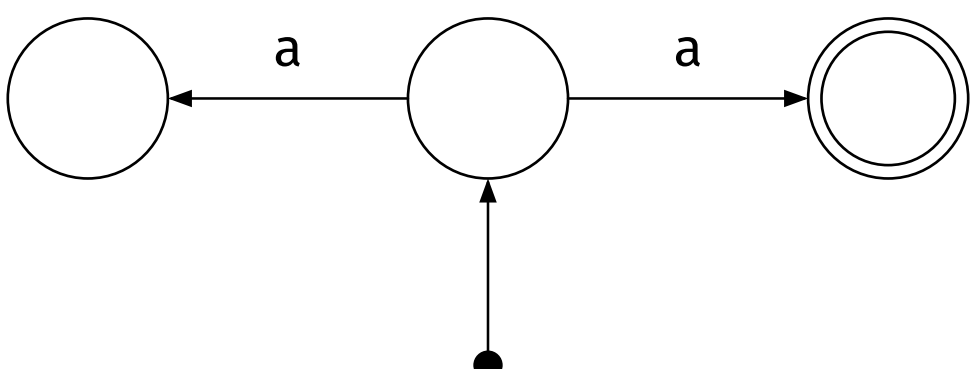
Automater

- Automater består av olika tillstånd med övergångar
- Vissa tillstånd är *accepterande*
- Kan användas för att känna igen *reguljära språk*

- För varje övergång konsumeras en symbol
- Exempel: automat som känner igen det binära språket med ett jämnt antal ettor:



- Föregående automat var *deterministisk* (DFA)
- För varje tillstånd och varje symbol finns högst en övergång att välja
- En automat kan också vara *icke-deterministisk* (NFA). Exempel:



Reguljära uttryck

(regex)

Reguljära uttryck

- Reguljära uttryck är ett sätt att beskriva reguljära språk
- De grundläggande operationerna är:
  - **Konkatenering**: `abc` matchar `abc`
  - **Alternativ**: `dag|natt` matchar `dag`, `natt`
  - **Upprepning** (Kleene closure): `ab*` matchar `a`, `ab`, `abb`...
  - **Gruppering**: `a(b|c)d` matchar `abd`, `acd`
- Dessa operationer räcker för att beskriva *alla* reguljära språk...
- ...men i praktiken har fler operationer tillkommit (`?`, `+`, `[...]`)

Reguljära uttryck

- När reguljära uttryck implementeras i programvara omvandlas de oftast till ändliga automater
- Det sker vanligtvis i följande steg:
    1. Omvandla uttrycket till en NFA (*Thompsons construction*)
    2. Omvandla NFA:n till en DFA (*Subset/powerset construction*)
    3. Minimera DFA:n (*Hopcrofts algoritm*)
- Det sista steget är en valfri optimering som vi inte går igenom

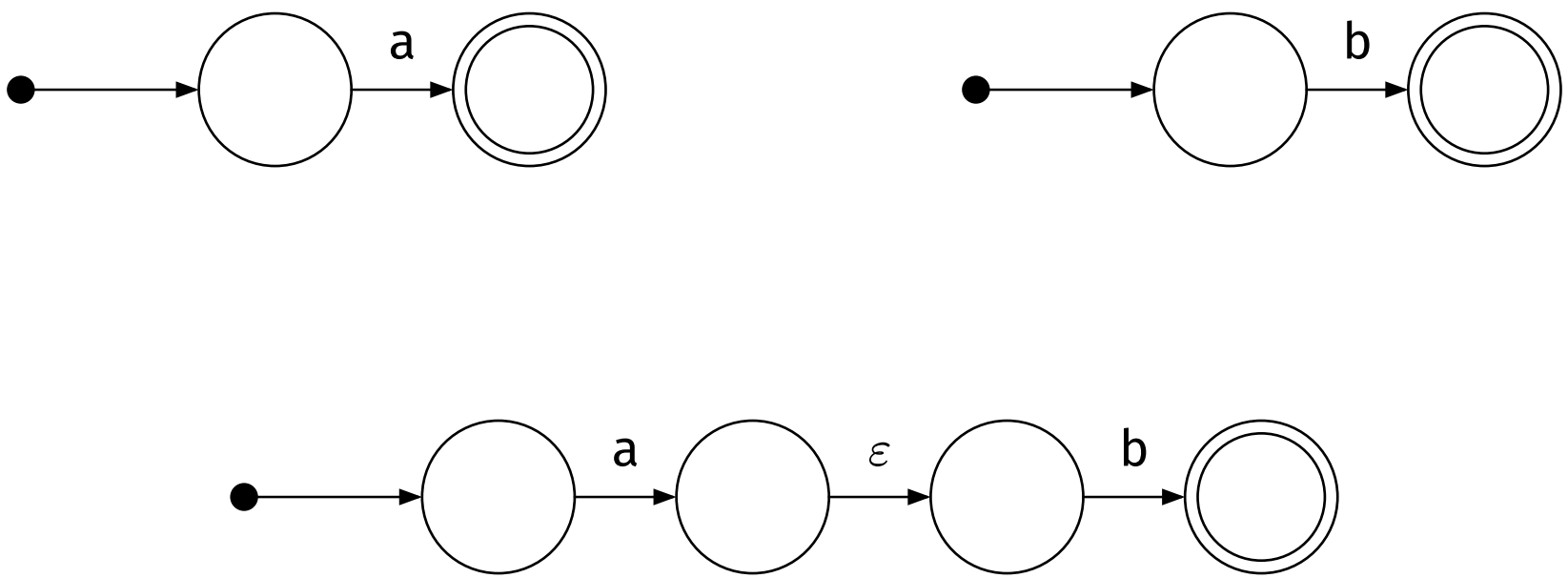
Thompson’s construction

Thompson's construction – ε-övergångar

- När vi konstruerar en NFA tillåter vi *ε-övergångar*.
- ε-övergångar konsumerar ingen symbol

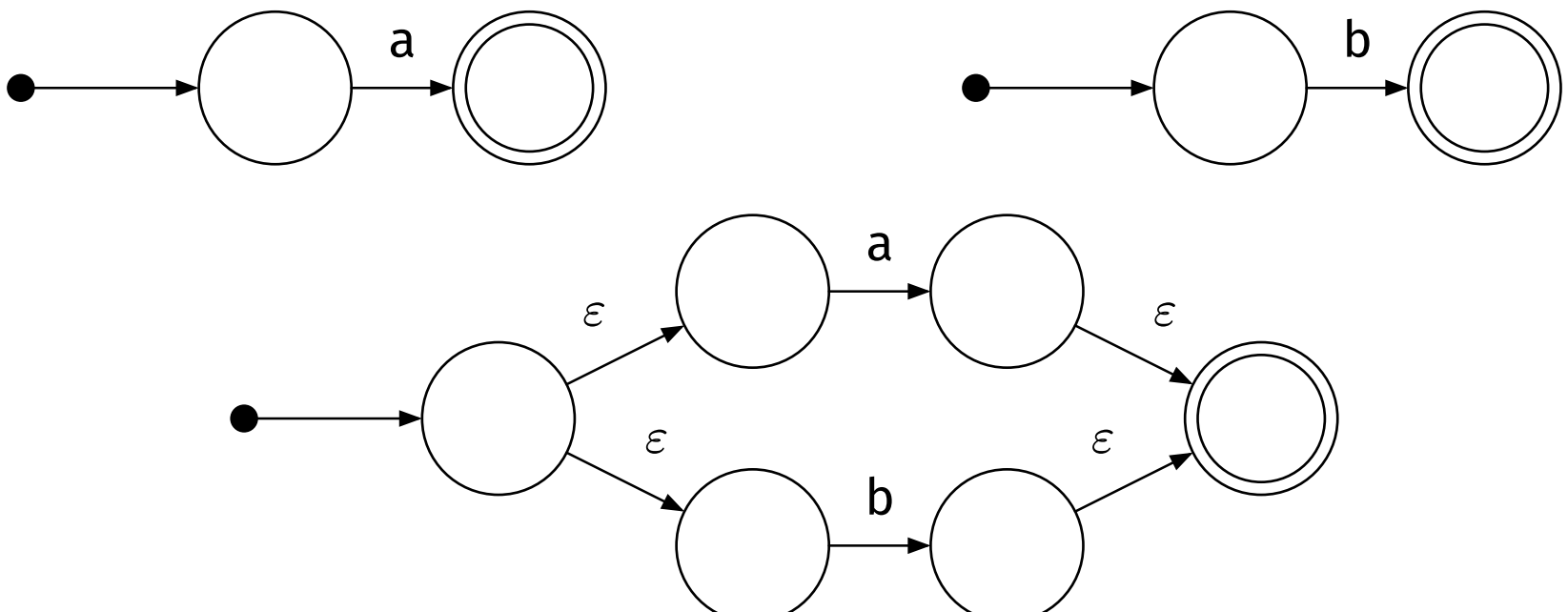
Thompson's construction – konkatenering

- Exempeluttryck: ab



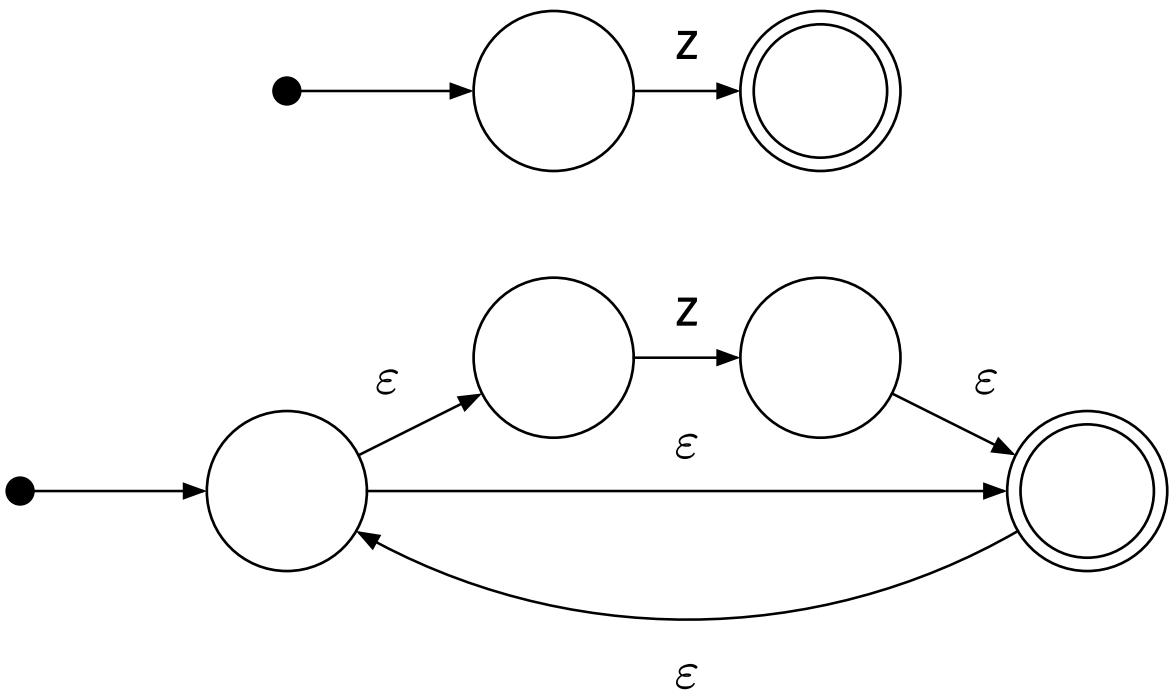
Thompson's construction – alternativ

- Exempeluttryck: a | b



Thompson's construction – upprepning

- Exempeluttryck: $z*$

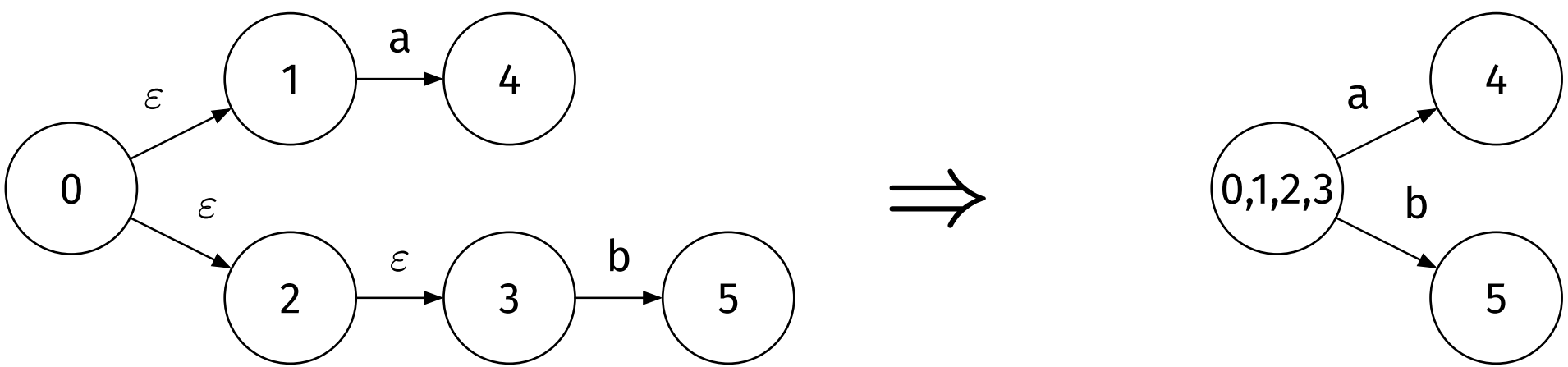


Exempel på tavlan:

`(ab)*|cd`

Subset construction

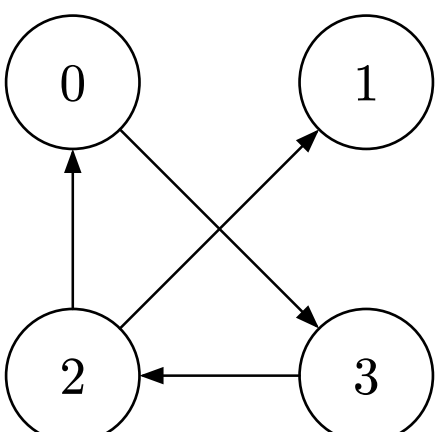
- Tillstånd i DFA:n motsvarar delmängder av tillstånden i NFA:n
- Vi slår ihop alla tillstånd som kan nås med en eller fler ε-övergångar
- Vi slår även ihop tillstånd som nås med samma symbol



Exekvering av NFA

Exekvering av NFA

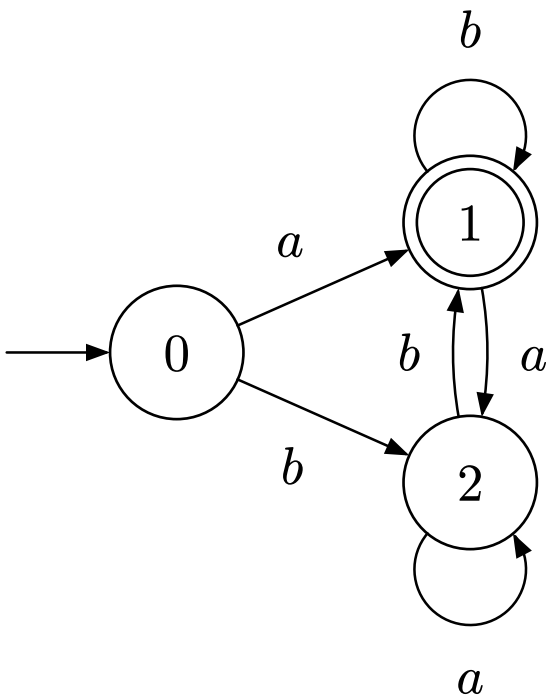
Vi representerar kanter mellan tillstånd med en grannmatris



$$
\begin{array}{c} \\ 0 \to \\ 1 \to \\ 2 \to \\ 3 \to \end{array}
\begin{array}{c}
\begin{array}{cccc} 0 & 1 & 2 & 3 \\ \uparrow & \uparrow & \uparrow & \uparrow \end{array} \\
\begin{bmatrix} 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix}
\end{array}
\iff
\begin{array}{l} 0 \to 3 \\ 2 \to 0 \\ 2 \to 1 \\ 3 \to 2 \end{array}
$$

Exekvering av NFA

- I en automat gäller olika kanter för olika indatasymboler
- En grannmatris för varje indatasymbol

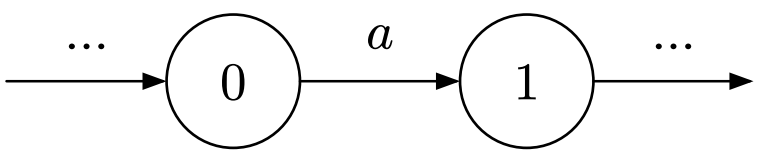


$$M(a) = \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & 1 \end{bmatrix}$$

$$M(b) = \begin{bmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ 0 & 1 & 0 \end{bmatrix}$$

Exekvering av NFA

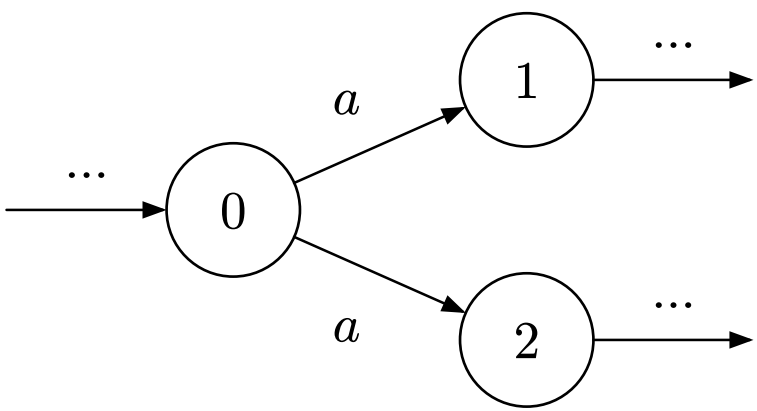
I en DFA stegar vi genom grafen från ett tillstånd till nästa




```
read 'a' in state 0
 →  go to state 1
```

Exekvering av NFA

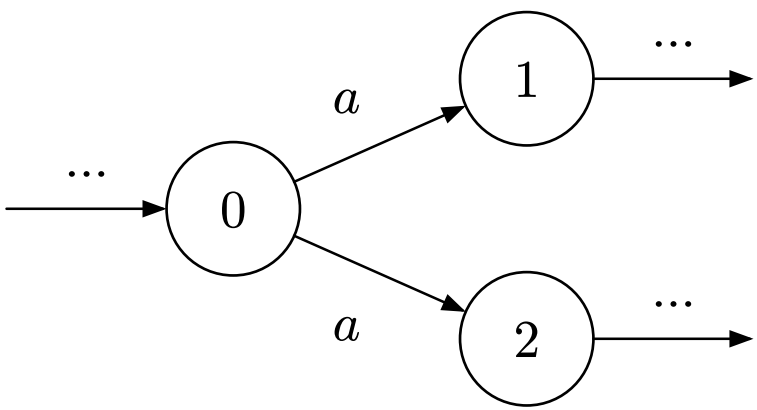
Hur gör vi med en NFA?




```
read 'a' in state 0
→ ?
```

Exekvering av NFA

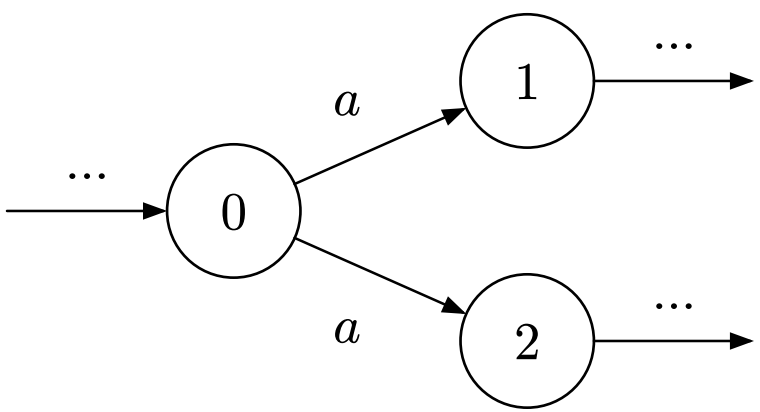
- Istället för ett enda state har vi en ”superposition”
- En vektor som räknar antalet positioner för varje tillstånd




```
read 'a' in state [1, 0, 0, ...]
 →  go to state [0, 1, 1, ...]
```

Exekvering av NFA

- I början har vektorn bara en etta i starttillståndet: $[1 \;\; 0 \;\; 0 \;\; \cdots]$
- Ett steg görs med multiplikation av tillståndvektorn och grannmatrisen för nästa indatasymbol

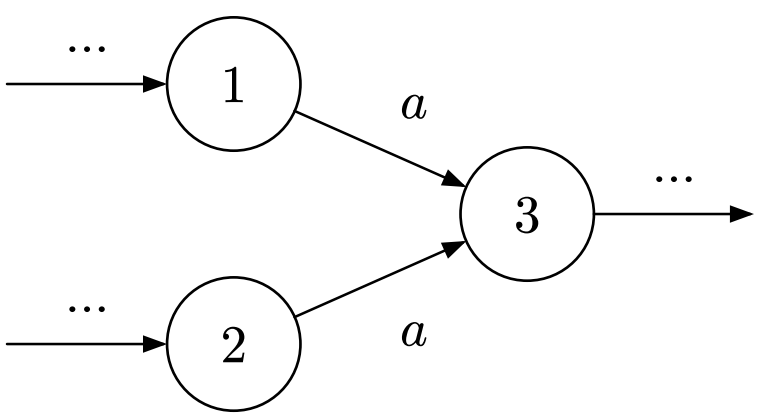


$$\vec{s}M(a) = [1 \;\; 0 \;\; 0 \;\; \cdots] \begin{bmatrix} 0 & 1 & 1 & \cdots \\ \vdots & \ddots & & \\ & & & \end{bmatrix}$$

$$= [0 \;\; 1 \;\; 1 \;\; \cdots]$$

Exekvering av NFA

- Vi kan ha flera positioner i samma tillstånd
- Egentligen vill vi bara veta om vi är i ett tillstånd eller inte



$$\vec{s}M(a) = \begin{bmatrix} 0 & 1 & 1 & 0 & \cdots \end{bmatrix} \begin{bmatrix} & & \cdots & & \\ 0 & 0 & 0 & 1 & \cdots \\ 0 & 0 & 0 & 1 & \cdots \\ \vdots & & & \ddots & \end{bmatrix}$$

$$= \begin{bmatrix} 0 & 0 & 0 & 2 & \cdots \end{bmatrix}$$

- Vi kan göra en enklare *max-min-matrismultiplikation*

Vanlig skalärprodukt med addition och multiplikation:

$$\boldsymbol{a} \cdot \boldsymbol{b} = a_1 b_1 + a_2 b_2 + \ldots$$

max-min-skalärprodukt:

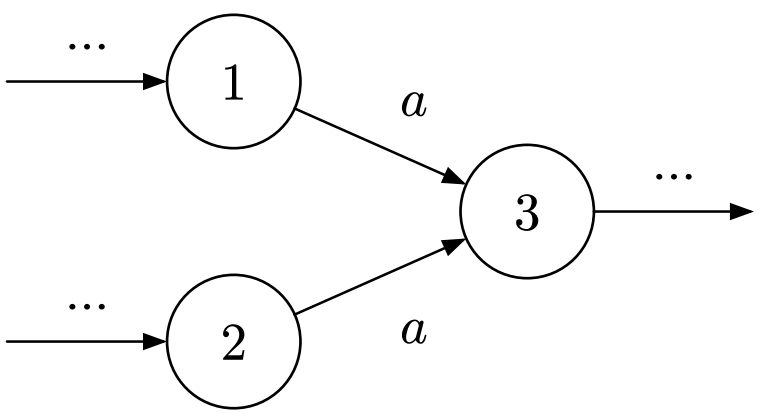
$$\boldsymbol{a} \cdot \boldsymbol{b} = \max(\min(a_1, b_1), \min(a_2, b_2), \ldots)$$

(en matrisprodukt är en matris med skalärprodukter av rader och kolumner)

Exekvering av NFA

- Om vi bara använder 0 och 1 i våra matriser är detta ekvivalent med *boolesk matrismultiplikation* ($\boldsymbol{A} \odot \boldsymbol{B}$)
- $\max$ är ekvivalent med `OR`, $\min$ är ekvivalent med `AND`

$$\boldsymbol{a} \cdot \boldsymbol{b} = (a_1 \wedge b_1) \vee (a_2 \wedge b_2) \vee \ldots$$



$$\vec{\boldsymbol{s}} \odot M(a) = \begin{bmatrix} 0 & 1 & 1 & 0 & \cdots \end{bmatrix} \begin{bmatrix} & & \cdots & & \\ 0 & 0 & 0 & 1 & \cdots \\ 0 & 0 & 0 & 1 & \cdots \\ \vdots & & & \ddots & \end{bmatrix}$$

$$= \begin{bmatrix} 0 & 0 & 0 & 1 & \cdots \end{bmatrix}$$

Exekvering av NFA

- Vi kan lagra tillståndsvektorer och matriskolumner som bitvektorer
- t.ex. $\begin{bmatrix} 0 & 1 & 1 & 0 & 0 \end{bmatrix} = 01100_2$
- Med högst 64 tillstånd kan en vektor eller matriskolumn vara en u64
  - För fler tillstånd kan vi stränga ihop flera ints eller använda ett bignum-bibliotek/språk
- Matrisprodukt görs med bitoperationer

```
def step(state, m):
  new_state = 0
  for i in range(len(m)):
    if state & m[i] != 0:
      new_state |= (1 << i)
  return new_state
```

```
def run(start, accept, Ms):
  state = start
  for sym in sys.stdin.read():
    state = step(state, Ms[sym])
  return state & accept != 0
```

Exekvering av NFA

- Kan tolkas som deterministisk vandring genom graf av tillståndsvektorer
- Den grafen är *implicit*; vi lagrar inte alla möjliga tillståndsvektorer eller kanter mellan dem
- Om vi konstruerar en *explicit* graf har vi omvandlat vår NFA till en DFA
  - Varje NFA har därmed en mostvarande DFA; NFA och DFA är ekvivalenta
  - För en NFA med n tillstånd kan den explicita grafen av tillståndsvektorer ha upp till 2^n noder

Exekvering av NFA

Varför?

- Enkelt att implementera
- Snabbare att kompilera än en DFA
- Kan kräva mindre minne, mostvarande DFA har i värsta fall 2^n noder
- I vissa fall även snabbare att köra
  - Optimeringar såsom state caching (typ JIT för regex) kan läggas till
- Flexibelt, lättare att lägga till utökad regex
- Behöver dock vara ε-fri precis som DFA

Läxa

- Skriv ett en regex-implementation med stöd för UTF-8
- Kan vara ett kommandoradsverktyg (som `grep`)
- Annars (typ bibliotek): skriv *tester*!
- Ska ha stöd för konkatenering, alternativ, upprepning och gruppering
- *Valfritt*: +, ?, \, [a-z] och case insensitive
- UTF-8 ska avkodas av er
- Det går bra att samarbeta i par
- Lägg koden i ett repo med namnet `<kth_id>-regex`

BNF-notation för reguljära uttryck


```
<regex> ::= <alternation> ;

<alternation> ::= <concatenation> | <concatenation> "|" <alternation> ;

<concatenation> ::= <kleene_closure> | <kleene_closure> <concatenation> ;

<kleene_closure> ::= <atom> | <atom> "*" ;

<atom> ::= CHARACTER | "(" <alternation> ")" ;
```