Notater: INF2080

Veronika Heimsbakk
veronahe@student.matnat.uio.no

1. juni 2013

Innhold

1 Intro

INF2080 - Logikk og Beregninger tar jeg våren 2013. Notatene er basert delvis på pensumboka (Sipser), egne forelesningsnotater, gruppeundervisning og foiler på kurssiden. Under semesteret oppdateres disse notatene når jeg gidder. Hovedsakelig i bruk for eksamensrepetisjon. Om noen ønsker LaTeXmarkup som er brukt i notatene, er det bare å sende meg en e-post.

2 Terminologi

2.1 Mengder

En mengde er en endelig eller uendelig samling av objekter der rekkefølgen ignoreres. Objektene i en mengde kalles elementer.

Notasjon $A = \{a, b, c\}, B = \{d, a, e\}$, den tomme mengden $= \emptyset$

- I: a $\in$ {a, b, c}
- Ikke i: d $\notin$ {a, b, c}
- Delmengde: {a} $\subseteq$ {a, b, c}
- Union: A $\cup$ B = {a, b, c, d, e}
- Snitt: A $\cap$ B = {a}
- Differanse: A / B = {b, c}

2.2 Boolsk logikk

		og	eller	ikke	xor	ekvivalens	implikasjon
		$\vee$	$\wedge$	$\neg$	$\oplus$	$\leftrightarrow$	$\rightarrow$
0	0	0	0	1	0	1	1
0	1	0	1	0	1	0	1
1	0	0	1		1	0	0
1	1	1	1		0	1	1

2.3 Nyttige definisjoner

Symmetri En binær relasjon R på mengden S er *symmetrisk* hvis det for alle x, y er slik at hvis $\langle x, y\rangle \in$ R, så $\langle y, x\rangle \in$ R.

Refleksiv En binær relasjon R på mengden S er *refleksiv* hvis det for alle x i S er slik at $\langle x, x\rangle \in$ R.

Transitiv En binær relasjon R på mengden S er *transitiv* hvis det for alle x, y, z er slik at hvis $\langle x, y\rangle \in$ R og $\langle y, z\rangle \in$ R, så $\langle x, z\rangle \in$ R.

Ekvivalensrealsjon En binær relasjon på mengden S er en *ekvivalensrelasjon* hvis den er refleksiv, symmetrisk og transitiv.

Injektiv En funksjon $f : A \to B$ er *injektiv* hvis for alle $x, y \in A$ så impliserer $x \neq y$ at $f(x) \neq f(y)$. Sier at f er en-til-en.

Surjektiv En funksjon $f : A \to B$ er *surjetiv* hvis for alle $y \in B$ så fins $x \in A$ slik at $f(x) = y$. Vi sier at f er på.

Bijektiv En funksjon er *bijektiv* hvis den er injektiv og surjektiv. Vi sier også at funksjonen er en en-til-en korrespondanse.

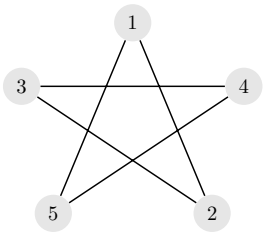
Oppfyllbarhet

Tautologi

Falisifiserbarhet

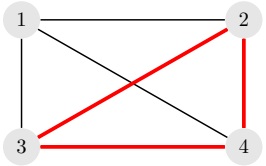
Kontradiksjon

2.4 Grafer



Figur 1: Eksempel på graf.

En **urettet graf**, er en mengde linjer som kobler sammen noen punkter. Punktene kalles **noder** og linjene kalles **kanter**. Antall kanter til en node kalles **graden** til den noden. I Fig. 1 har alle nodene grad 2.



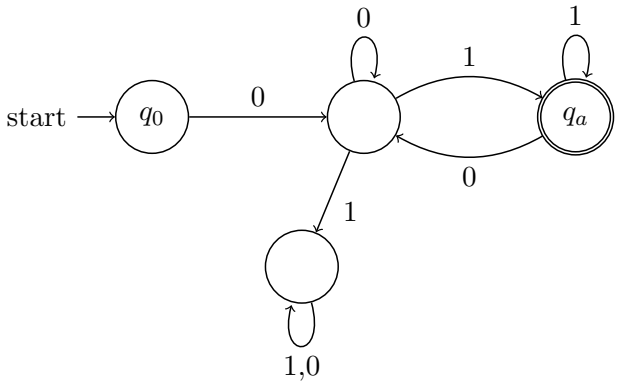
Figur 2: Eksempel på graf med subgraf.

I Fig. 2 er den rød streken en **subgraf** av hele grafen i figuren. De røde kantene representere også en **sti** gjennom grafen. En sti er en vei over kantene i en graf. En **rettet graf** er en graf med piler på kantene.

2.5 Språk og alfabeter

3 Automater

3.1 Deterministiske



Figur 3: Eksempel på en deterministic finite automata (DFA).

- Tilstander: ◯, automaten over har fire tilstander. En automat består av én eller flere tilstander.
- Starttilstand: en automat har nøyaktig en starttilstand q_0.
- Overganger: $\longrightarrow$, disse er markert med elementer fra alfabetet Σ= {0,1} (i eksempelet).
- Akseptert tilstand: ◎, det er x-antall slike aksepterte tilstander.

Eksempel på input og utfall for automaten i Figur 3.

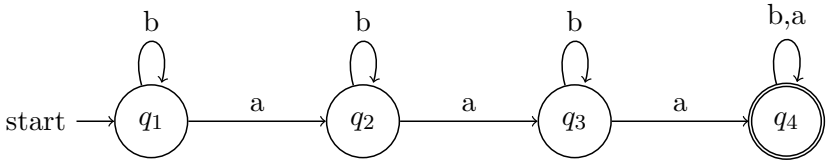
Input	Utfall
01	akseptert
010	avvist
0011	akseptert
00110	avvist

La automaten i Figur 3 hete M, M kjenner igjen språket: $L(M) = \{w|$ w starter med 0 og slutter med 1$\}$. $L(M)$ er språket med en samling stringer w som M aksepterer. Eksempel: 0101 er et element i $L(M)$, $0101 \in L(M)$.

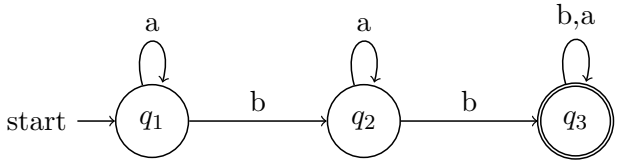
Definisjon 3.1. *En **deterministic finite automata** er et 5-tuppel* $(Q, \Sigma, \delta, q_0, F)$*, hvor:*

- Q *er et sett tilstander.*
- Σ *er input alfabetet.*
- δ*:* $Q \times \Sigma \to Q$ *er pilenes funksjon.*
- q_0 *er* Q *som starttilstand.*
- $F \subseteq Q$ *er de x-antall aksepterte.*

Eksempel Over språket $\Sigma = \{a, b\}$



Figur 4: Gjenkjenner om et ord har minst tre a-er.



Figur 5: Gjenkjenner om et ord har minst to b-er.

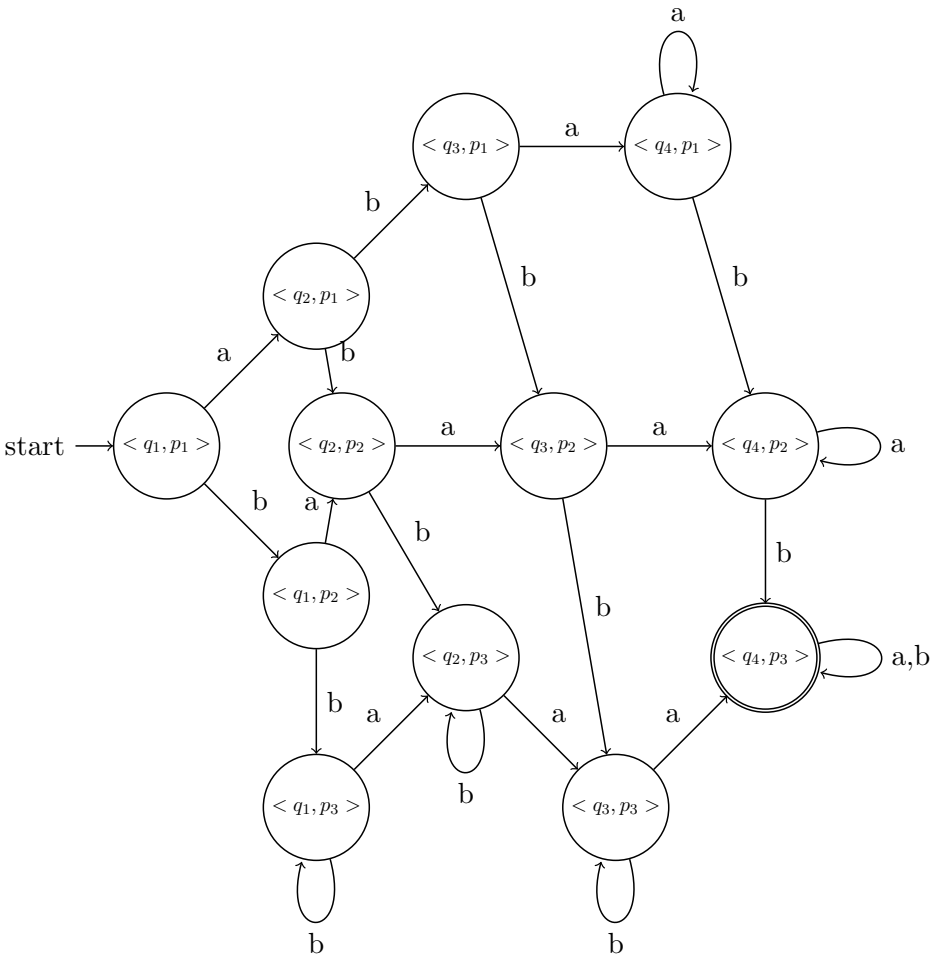
Hva nå om vi vil slå sammen disse to, og ha en automat som gjenkjenner minst tre a-er og minst to b-er?

3.2 Nondeterministiske

Hos en DFA er neste tilstand bestemt av inputen. I en NFA har man flervalg, man kan også gå over ε.

Definisjon 3.1. *En* ***nondeterministic finite automata*** *er et 5-tuppel* $(Q, \Sigma, \delta, q_0, F)$*, hvor:*

1. Q *er tilstandene.*
2. Σ *er alfabetet.*
3. $\delta : Q \times \Sigma_\epsilon \longrightarrow P(Q)$
4. $q_0 \in Q$ *er starttilstanden.*
5. $F \subseteq Q$ *er de aksepterte tilstandene.*

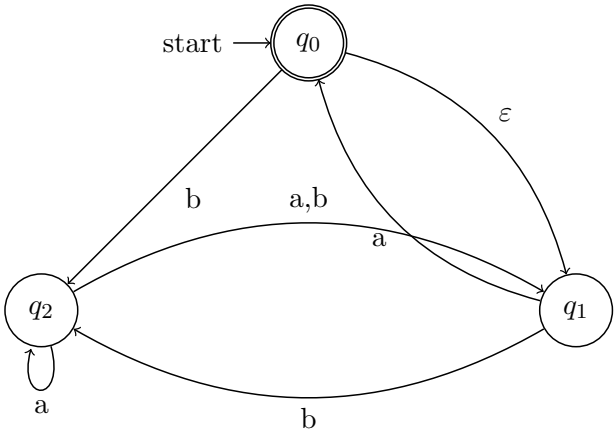


Figur 6: Gjenkjenner om et ord har minst to b-er og minst tre a-er.

3.3 Pumpelemma

1. Stringen lager en sti mellom tilstandene.
2. Anta at antall tilstander er p.
3. Enhver string lengere enn p må gå n loop.
4. Treffer tilstand vi har vært borte i fra før.

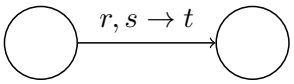
Definisjon 3.1. *Anta at vi har en DFA D med d tilstander og et ord w av lengde n som blir akseptert av D. Om $n > d$, så kan vi dele w opp i $w = xyz$ med lengden av $xy \leq n+1$, y ikke tom og der samtlige $x(y)*z$ blir akseptert.*



Figur 7: Eksempel på en NFA.

3.4 Pushdown automata - PDA

En PDA består av tilstander, en start, transisjoner og aksepterende. Starten er en starttilstand og en tom stakk. Transisjoner er som følger:



Hvor r, s er forutsetninger og t er aksjonen. Deler dette opp i to:

1. Vokter: symbol lest, symbol øverst på stakken/tas av på stakken.
2. Aksjon: nytt symbol på stakken.

Bruker også ε. Akseptering skjer med aksepterende tilstand eller en tom stakk.

3.4.1 Chomskys normalform

- Enhver CFG kan skrives ved bruk av regler: $A \longrightarrow BC$, $D \longrightarrow d$
- Forgrening for variable.
- Unært (rett fram) for terminaler.

4 Regulære Uttrykk

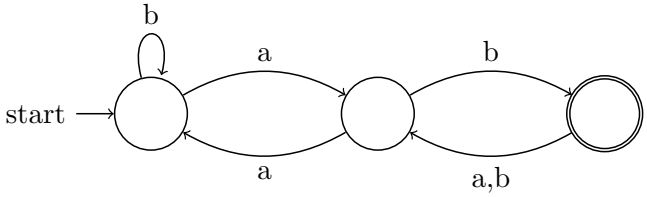
Ta $\Sigma = \{a, b\}$. De regulære uttrykkene gir delmengder av stringer i $\{a, b\}$.

- $\emptyset$ – tom delmengde.

- ϵ – $\{\epsilon\}$
- a – {a}
- b – {b}
- union
- konkatinering
- kleene stjerne

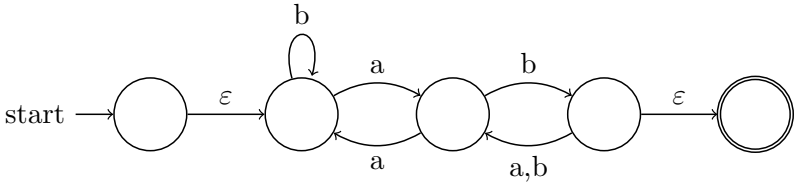
Et regulært språk er en delmengde av stringen som en DFA kan akseptere. Regulære uttrykk = regulære språk.

4.1 DFA til regulært uttrykk

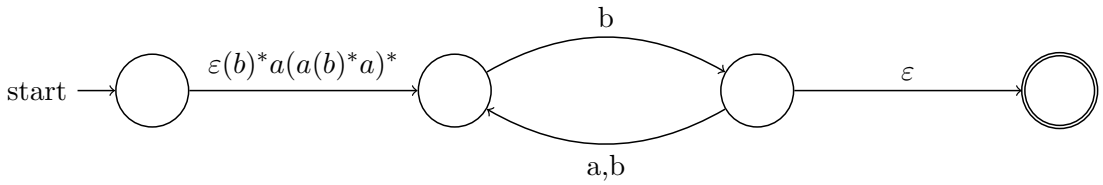


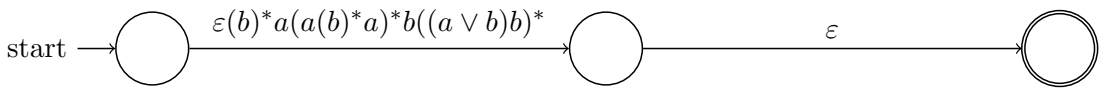
Trenger fire steg:

Steg 1 legger til starttilstand og aksepterende over ε.



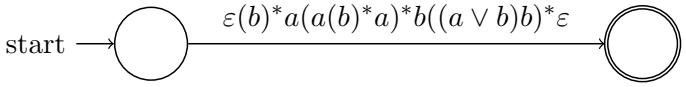
Steg 2 bruker GNFA (reg.exp på pilene) og tar vekk en tilstand, så reparer.





Steg 3 tar vekk en ny tilstand.

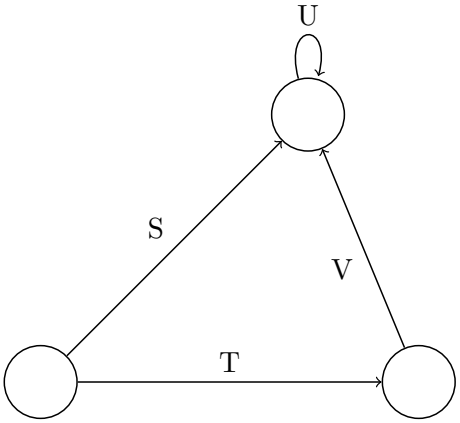
Steg 4 tar bort den siste tilstanden.



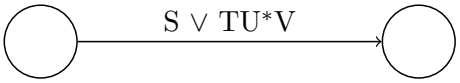
Vi får da det hele regulære uttrykket:

$$\epsilon(b)^*a(a(b)^*a)^*b((a \vee b)b)^*\epsilon$$

4.2 Fra NFA til reg.ex.



Figur 8: En NFA.



Figur 9: Figur 8 som regex.

5 Kontekstfrie språk

1. Gramatikk – CFG – kontekstfire språk.
2. Automat med stakk – PDA – pushdown automata.

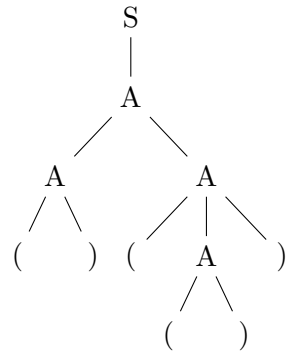
Eksempel på CFG Språk: ((()))

- Variable: S og A.
- Terminaler: (og).
- Startsymbol: S.
- Regler (omskrivningsregler): $S \to A$, $A \to AA$, $A \to (A)$, $A \to ()$
- Utledning:

$$\frac{\frac{\frac{\frac{\frac{S}{A}}{AA}}{()A}}{()(A)}}{()(())}$$

  Lager uttrykk som ()(()).

Regler: $v.s. \to h.s.$ Venstresiden inneholder nøyaktig ett symbol. Dette er begrunnelsen for at språk er kontekstfri. Symbolet er en variabel. Høyresiden er en string av symboler. Utledningen fortsetter omskrivningene til vi bare har terminale symboler.



Figur 10: Utledningen i treform.

Begynner med S. Variablene er interne noder, og terminalene er løvnoder.

6 Turing maskiner

6.1 Turing analyse

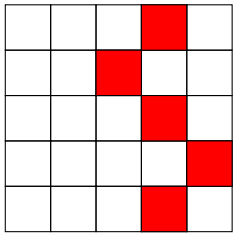
Beregninger på rutepapir. Trenger ekstra blanke ruter. Tillater flere tilstander. Klarer seg med én aktiv rute. Beregning gjøres på tape. Bevege aktiv

rute høyre/ventre/stopp. Klarer seg med to symboler i alfabetet. Kan utføre hvilken som helst beregning slik.

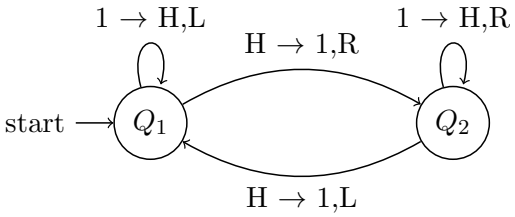
6.2 Turing maskinen

Forskjellen mellom en finite automata og en turing maskin er:

- En turing maskin kan både lese og skrive til og fra en tape.
- Lese-/skrivehodet kan bevege seg til både høyre og venstre.
- Tapen er uendelig.
- Aksepterende og avvisende tilstand trår i kraft umiddelbart.



Tapen er den horisontale raden og aktiv rute er markert med rød. Tid er x-aksen og rom er y-aksen.



Figur 11: Eksempel på en Turing maskin.

I eksempel 10 ser vi en TM med to tilstander: Q_1 og Q_2. Denne tar enten inn 1 eller H (blank). Den kan lese 1, da skriver den H og går til venstre (left, L), da havner den i samme tilstand Q_1. Så kan den lese H, skrive 1 og gå til høyre (right, R), og havner i tilstand Q_2.

DEFINISJON 6.1. *En **TM** er et 7-tuppel* $(Q, \Sigma, \Gamma, \delta, q_0, q_{accept}, q_{reject})$

1. Q *er tilstandene,*
2. Σ *er input alfabetet som ikke holder blanke,*
3. Γ *er alfabetet på tapen – inkludert blanke,*
4. $\delta : Q \times \Gamma \to Q \times \Gamma \times \{L, R\}$ *er funksjonen for transisjonene,*

5. $q_0 \in Q$ *er starttilstanden,*

6. $q_{accept} \in Q$ *er aksepterende tilstand, og*

7. $q_{reject} \in Q$ *er avvist tilstand. Hvor* $q_{reject} \neq q_{accept}$.

6.2.1 Konfigurasjon

Når en Turing maskin beregner vil endringer skje i gjeldene tilstand, gjeldene innhold på tapen og gjeldende posisjon på lokasjonen til hodet. Disse tre gjenstandene kalles en **konfigurasjon** av Turing maskinen.

Konfigurasjonen er skrevet som: uqv, hvor q er tilstanden, u og v er to stringer i Γ. q er gjeldende tilstand og uv er gjeldende innhold på tapen. Gjeldende posisjon for hodet er v.

Spesielle hendelser kan oppstå når hodet er på en av endelsene i konfigurasjonen. For en hendelse til venstre: q_ibv gir q_jcv hvis transisjonen beveger seg til venstre. Og den gir cq_jv for en transisjon som går til høyre. For den som slutter til høyre i konfigurasjonen uaq_i er ekvivalent med $uaq_i\textvisiblespace$, fordi vi antar at resten av tapen er blank.

Startkonfigurasjonen for $\mathcal{M}$ på input w er konfigurasjonen q_0w, dette indikerer at maskinen er i starttilstanden q_0 med hodet lengst til venstre på tapen. I en aksepterende konfigurasjon, er tilstanden q_{accept}. Og for en avvisende konfigurasjon er den q_{reject}. Aksepterende og avvisende konfigurasjoner er stagnerende konfigurasjoner (halting) og gir ingen flere konfigurasjoner.

En TM $\mathcal{M}$ aksepterer input w hvis en sekvens av konfigurasjoner $C_1, C_2, \ldots, C_k$ eksisterer, hvor:

1. C_1 er startkonfigurasjonen på input w,

2. hver C_i gir C_{i+1}, og

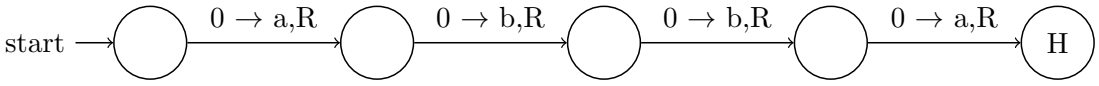
3. C_k er en aksepterende konfigurasjon.

6.2.2 Grunnleggende Turing maskiner

Skrive ord

Alfabet : a, b, 0 – 0 er blank.

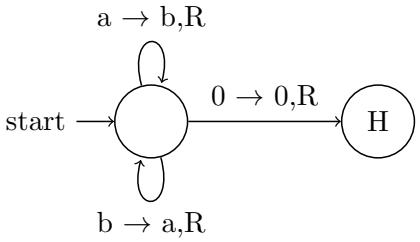
Spesifikasjon : skriver *abba* og stopper.



Bytte bokstaver

Alfabet : a, b, 0 – 0 er blank.

Spesifikasjon : bytter b mot a og motsatt til den treffer en blank.

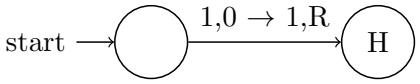


6.3 Flittige bevere

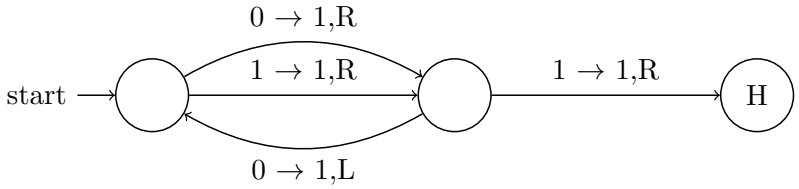
Maskiner i alfabetet $0, 1$ – 0 er blank. En bever stopper.
N-bever: bever med N tilstander + stopptilstand.

- For Turing maskiner i 0,1 med N tilstander + stopp:
  - $2N$ voktere.
  - Hver transisjon kan utføre $2(N + 1)2$ handlinger.
  - Det er $(4N + 4)^{2N}$ slike maskiner.
  - Noen av disse stopper, andre stopper ikke.
- Flittig N-bever: N-bever som produserer flest antall 1ere.

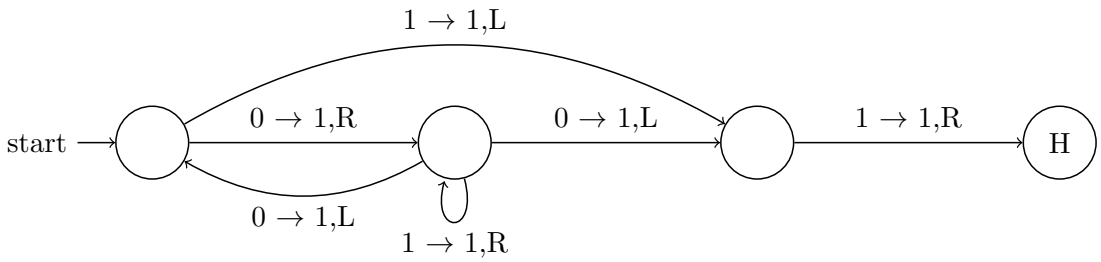
Bever funksjonen: $\beta(1) = 1, \beta(2) = 4, \beta(3) = 6, \beta(4) = 13, \beta(5) \geq 4098$



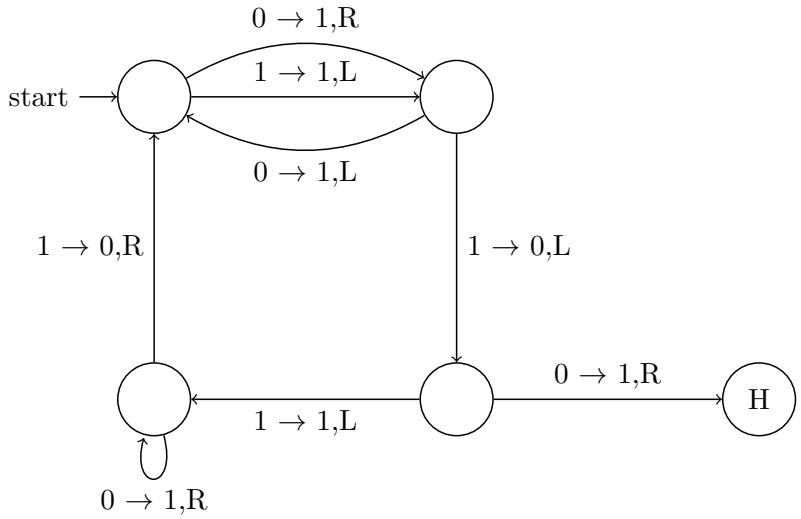
Figur 12: Flittig 1-bever – $\beta(1) = 1$



Figur 13: Flittig 2-bever – $\beta(2) = 4$



Figur 14: Flittig 3-bever – $\beta(3) = 6$



Figur 15: Flittig 4-bever – $\beta(4) = 13$

Beverfunksjonen er ikke beregnbar La $f(n)$ være en beregnbar funksjon.

- TM med k tilstander i alfabetet 0, 1.
- Starter beregningen til venstre for n 1ere – ellers bare 0.
- Stopper beregningen til venstre for $f(n)$ 1ere – ellers bare 0.
- Antar at $f(n) > n^2$ – f vokser raskere enn lineært.
- Antar f er strengt voksende – $f(n+1) > f(n)$.

$f(f(n))$ kan beregnes med $2k$ tilstander. Tallet $f(f(n))$ kan beregnes med $n + 2k$ tilstander fra blank. $\beta(n + 2k) \geq f(f(n)) \geq f(n^2) \succ f(n + 2k)$

6.4 Nondeterministisk Turing maskin

På hvilket som helst punkt i beregningen kan maskinen fortsette i forskjellige retninger. Transisjonsfunksjon for en NTM:

$$\delta : Q \times \Gamma \to \mathcal{P}(Q \times \Gamma \times \{L, R\}).$$

Beregningen av en NTM er et tre med grener som tilsvarer de forskjellige mulighetene for maskinen.

Teorem Hver NTM har en ekvivalent deterministisk TM.

Simulerer en NTM $\mathcal{N}$ med en deterministisk TM $\mathcal{D}$. Tanken bak simuleringen er at $\mathcal{D}$ må prøve alle mulige grener av $\mathcal{N}$. Hvis $\mathcal{D}$ noen gang finner en aksepterende tilstand i en av grenene, aksepter. Ellers vil $\mathcal{D}$ sin simulering ikke terminere.

Bevis Simulerende $\mathcal{D}$ har tre taper. Dette er ekvivalent med å ha en tape. Tape 1 har alltid input string og stopper aldri. Tape 2 er en kopi av $\mathcal{N}$ sin tape av en gren i dens nondeterministiske beregning. Tape 3 holder styr på lokasjonen til $\mathcal{D}$ i $\mathcal{N}$ sitt tre.

Først vurderer vi data-representasjonen på tape 3. Hver node i treet kan ha max. b barn, hvor b er størrelsen av det største settet som mulig av valg gitt av $\mathcal{N}$ sin transisjonsfunksjon. For hver node i treet gir vi en adresse som er en string fra alfabetet $\Gamma_b = \{1, 2, \ldots, b\}$. Gir adressen 231 til noden vi finner ved å starte ved roten, går så til 2. barn, så til den nodens 3. barn, og til slutt dens 1. barn. Tape 3 inneholder en string over Γ_b. Den tomme stringen adresserer roten i treet.

Beskrivelse av $\mathcal{D}$:

1. Tape 1 inneholder input w, tape 2 og tape 3 er tomme.
2. Kopierer tape 1 til tape 2 og initialiserer stringen på tape 3 til ε.
3. Bruker tape 2 til å simulere $\mathcal{N}$ med input w på en gren av den nondeterministiske beregningen. Før hvert steg av $\mathcal{N}$, konstanter neste symbol på tape 3 for å bestemme hvilket valg blant de som $\mathcal{N}$ tillater i sin transisjonsfunksjon. Hvis det er ingen symboler igjen på tape 3 eller om det nondeterministiske valget ikke er gyldig, aborter grenen og gå til steg 4. Gå også til steg 4 hvis en avvisende konfigurasjon inntreffer. Hvis en aksepterende konfigurasjon inntreffer, aksepter.
4. Erstatt stringen på tape 3 med neste. Simuler neste gren ved å gå til steg 2.

7 Avgjørbar

Vi starter med beregningsproblemer knyttet til finite automata. Bruker algoritmer for å teste om automaten aksepterer en string, om språket er tomt og om to automater er ekvivalente.

7.1 For DFA

Eksempel Aksepteringsproblemet for DFA er ved testing om en spesifik string kan bli uttrykt som språket A_{DFA}. Dette språket inneholder enkoding for alle DFAer tilsammen med stringer som blir akseptert av DFAene. La

$$A_{DFA} = \{\langle \mathcal{B}, w\rangle | \mathcal{B} \text{ er en DFA som aksepterer input string } w\}.$$

Problemet med å teste om en DFA $\mathcal{B}$ aksepterer input w er det samme som å teste om $\langle \mathcal{B}, w\rangle$ er et medlem av språket A_{DFA}. En TM $\mathcal{M}$ bestemmer A_{DFA}

$\mathcal{M}$ = på input $\langle \mathcal{B}, w\rangle$, hvor $\mathcal{B}$ er en DFA og w er en string:

1. Simuler $\mathcal{B}$ på input w.
2. Hvis simuleringen ender i aksepterende tilstand, aksepter. Ellers, avvis.

Bevis Undersøke $\langle \mathcal{B}, w\rangle$. Det er en representasjon av en DFA $\mathcal{B}$ sammen med en string w. Når $\mathcal{M}$ får dens input, $\mathcal{B}$ velger om den representerer en DFA $\mathcal{B}$ og en tring w. Hvis ikke avviser $\mathcal{M}$. $\mathcal{M}$ følger simuleringen direkte. Holder styr på $\mathcal{B}$ sin gjeldende tilstand og $\mathcal{B}$ sin gjeldende posisjon i input w ved å skrive informasjon på tapen.

7.2 For NFA

Vi kan også bevise det samme for en NFA. La

$$A_{NFA} = \{\langle \mathcal{B}, w\rangle | \mathcal{B} \text{ er en NFA som aksepterer på input string } w\}.$$

Teorem A_{NFA} er et bestembart språk.

Bevis En TM $\mathcal{N}$ bestemmer A_{NFA}. La $\mathcal{N}$ bruke $\mathcal{M}$ som en sub-rutine. Fordi $\mathcal{M}$ er designet for å jobbe med DFAer, $\mathcal{N}$ konverterer først NFAen til en DFA, før den gir den til $\mathcal{M}$.

$\mathcal{N}$ = på input $\langle \mathcal{B}, w\rangle$, hvor $\mathcal{B}$ er en NFA og w er en string:

1. Koverter NFA $\mathcal{B}$ til en ekvivalent DFA $\mathcal{C}$, ved å bruke prosedyre for konvertering.
2. Kjør TM $\mathcal{M}$ på input $\langle \mathcal{C}, w\rangle$.

3. Hvis $\mathcal{M}$ aksepterer, aksepter. Ellers, avvis.

Ved å kjøre TM $\mathcal{M}$ i steg 2 betyr å i steg 2 betyr å inkorporere $\mathcal{M}$ til designet $\mathcal{N}$ som en sub-prosedyre.

7.3 For Regulært Uttrykk

Likedan kan vi se om et reg.ex. genererer en gitt string. La

$$A_{REX} = \{\langle \mathcal{R}, w\rangle | \mathcal{R} \text{ er et reg.ex. som genererer string } w \ \}.$$

Bevis TM $\mathcal{P}$ bestemmer A_{REX}.
$\mathcal{P}$ = på input $\langle \mathcal{R}, w\rangle$, hvor $\mathcal{R}$ er et reg.ex. og w en string:

1. Konverter reg.ex. $\mathcal{R}$ til en ekvivalent NFA $\mathcal{A}$.
2. Kjør TM $\mathcal{N}$ på input $\langle \mathcal{A}, w\rangle$.
3. Hvis $\mathcal{N}$ aksepterer, aksepter. Ellers, avvis.

8 Redusering

En reduksjon er en måte å konvertere et problem til et annet, slik at løsningen på det andre problemet kan brukes til å løse det første.

8.1 Stoppeproblemet

Ta for eksempel stoppeproblemet. Bruker da ikke-avgjørbarheten til A_{TM} for å vise ikke-avgjørbarheten til $HALT_{TM}$. La

$$HALT_{TM} = \{\langle \mathcal{M}, w\rangle | \mathcal{M} \text{ er en TM og } \mathcal{M} \text{ stagnerer på input } w \ \}.$$

Teorem $HALT_{TM}$ er ikke-avgjørbart.

Idé Anta at man har en TM $\mathcal{R}$ som avgjør $HALT_{TM}$. Med $\mathcal{R}$ kan man teste om $\mathcal{M}$ stopper på w eller ikke. Hvis $\mathcal{R}$ indikerer at $\mathcal{M}$ ikke stopper på w – avvis, fordi $\langle \mathcal{M}, w\rangle$ er ikke i A_{TM}. Hvis $\mathcal{R}$ indikerer at $\mathcal{M}$ stopper på w, kan man simulere uten fare for loop. Hvis TM $\mathcal{R}$ eksisterer kan vi avgjøre A_{TM}, men vi vet at A_{TM} er ikke-avgjørbart. Ved denne motsigelsen kan vi konkludere med at $\mathcal{R}$ ikke eksisterer – derfor er $HALT_{TM}$ ikke-avgjørbart.

Bevis For motsigelsens skyld anta at TM $\mathcal{R}$ avgjør $HALT_{TM}$. Konstruerer TM $\mathcal{S}$ for å avgjøre A_{TM}.

$\mathcal{S}$ = på input $\langle \mathcal{M}, w \rangle$, en enkoding av TM $\mathcal{M}$ og en string w:

1. Kjør TM $\mathcal{R}$ på input $\langle \mathcal{M}, w \rangle$.
2. Hvis $\mathcal{R}$ avviser, avvis.
3. Hvis $\mathcal{R}$ aksepterer, simuler $\mathcal{M}$ på w til den stopper.
4. Hvis $\mathcal{M}$ aksepterer, aksepter. Hvis $\mathcal{M}$ avviser, avvis.

Hvis $\mathcal{R}$ avgjør $HALT_{TM}$, så vil $\mathcal{S}$ avgjøre A_{TM}. Siden vi vet at A_{TM} er ikke-avgjørbar, så må $HALT_{TM}$ også være ikke-avgjørbar.

9 Kompleksitet i tid

Eksempel Gitt språket $A = \{0^k 1^k | k \geq 0\}$. A er et avgjørbart språk. Hvor mye tid bruker en single-tape TM på å avgjøre A? La TM hete M_1.

M_1 = på intput string w:

1. Søker på tapen og avviser om 0 blir funnet til høyre for 1.
2. Gjentar om både 0 og 1 er igjen på tapen.
3. Søker gjennom tapen og krysser bort enkel 0 og enkel 1.
4. Hvis det er flere 0er igjen etter alle 1ere er krysset av, eller motsatt, avvis. Hvis hele inputen er krysset av, aksepter.

I en **worst-case analyse**, antar vi lengst kjøretid for alle inputs av en gitt lengde. I en **gjennomsnittsanalyse** ser vi på snittet av kjøretiden for inputs med en gitt lengde.

Definisjon 9.1. *La M være en deterministisk TM som stopper for alle inputs.* ***Kjøretiden*** *for M er funksjonen*

$$f : \mathcal{N} \to \mathcal{N}$$

hvor $f(n)$ er max. lengde på steg M bruker for hvilken som helst input av lengde n. Hvis $f(n)$ er kjøretiden til M, er M en $f(n)$-tid TM.

9.1 P-klassen

Definisjon 9.1. *P er klassen av språk som er avgjort ved* ***polynom-tid*** *for en deterministisk single-tape TM.*

$$P = \bigcup_k TIME(n^k)$$

1. *P er en invariant for alle modeller for beregning som er polynom ekvivalent til den deterministiske single-tape Turing maskinen, og*

2. *P svarer til gruppen av problemer som realistisk sett kan løses på en datamaskin.*

9.1.1 Problemer i P

Når man skal analysere en algoritme for å vise at den kjører i polynom-tid, gjør vi to ting:

1. Vi må gi en stor O-notasjon på antall trinn algoritmen bruker når den kjører en input med lengde n.

2. Undersøke de individuelle stegene i beskrivelsen av algoritmen for å være sikker på at hvert steg kan bli implementert i polynom-tid ved en fornuftig deterministisk modell.

Når begge steg er gjennomført kan man konkludere med at algoritmen kjører i polynom-tid. Fordi vi har demonstrert at den kjører i polynomt antall steg, hvor hver kan kjøres i polynom-tid.

9.1.2 Grafer

En av disse er en **nabomatrise**[1], hvor (i, j) er 1 om det er en kant fra node i til node j, og 0 hvis ikke. Når man analyserer grafer blir kjøretiden beregnet av antall noder i stedet for størrelsen på grafen.

Problem En rettet graf $\mathcal{G}$ inneholder noder s og t. $PATH$-problemet er å bestemme om det fins en rettet sti fra s til t. La:

$$PATH = \{\langle \mathcal{G}, s, t\rangle | \mathcal{G} \text{ er en rettet graf som har en rettet sti fra } s \text{ til } t\}.$$

Bevis Vi kan bevise dette ved å presentere en polynom-tid algoritme som bestemmer $PATH$. En brute-force algoritme for $PATH$: Undersøke alle potensielle stier i $\mathcal{G}$ og bestemme om det er noen rettet sti fra s til t. En mulig sti er en sekvens av noder i $\mathcal{G}$ som har lengde på max. m, hvor m er antall noder $\mathcal{G}$. Men antallet slike potensielle stier er ca. m^m, som er eksponentiell i antall noder i $\mathcal{G}$. Derfor ser vi at brute-force algoritem bruker eksponentiell tid.

For å få en *polynom-tid* algoritme for $PATH$ må vi gjøre noe som unngår brute-force. En måte er å bruke graf-søk metode. Her markerer vi alle nodene i $\mathcal{G}$ som er tilgjengelige fra s ved rettede stier av lengde 1, til 2, til 3, til m.

[1] Egentlig **adjacency matrix**, men jeg finner ikke noe godt norskt ord for det.

Bevis En polynom-tid algoritme M for $PATH$.
M = på input $\langle \mathcal{G}, s, t \rangle$, hvor $\mathcal{G}$ er en rettet graf med noder s og t:

1. Marker node s.
2. Gjenta til ingen tilleggsnoder er markert:
   - Søk gjennom alle kanter i $\mathcal{G}$. Hvis en kant (a, b) er funnet, som går fra markert node a til ikke-markert node b, marker node b.
3. Hvis t er markert, aksepter. Ellers, avvis.

Analyse For å vise polynom-tid: Steg 1 og 3 i M er enkelt implementert i polynom-tid i hvilken som helst fornuftig deterministisk modell. Understeget i 2 involverer et søk av inputen og en test på markerte noder, som også kan implementeres i polynom-tid. Så M er en polynom-tid algoritme for $PATH$.

9.2 NP-klassen

Noen forsøk på å unngå brute-force i problemer har ikke vært suksessfult, og polynom-tid algoritmer for å løse disse vet man ikke om eksisterer. Hvorfor? Det vet man ikke. Kanskje det rett og slett ikke kan løses i polynom-tid? Men om man har en polynom-tid algoritme for ett slikt problem, kan den bli brukt til å løse en hel klasse slike problemer.

Eksempel En Hamilton-sti i en rettet graf $\mathcal{G}$ er en rettet sti som går gjennom en node nøyaktig én gang. La

$$HAMPATH = \{\langle \mathcal{G}, s, t \rangle | \mathcal{G} \text{ er en rettet graf med en Hamilton-sti fra } s \text{ til } t\}.$$

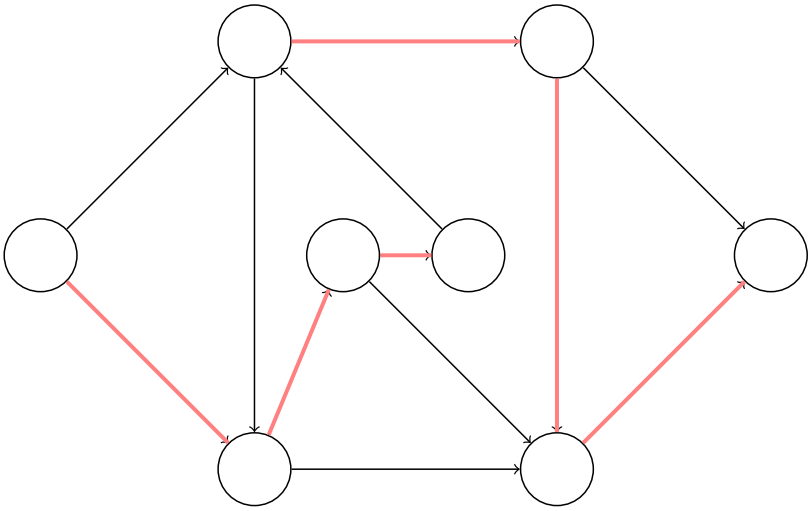
Vi kan enkelt få til eksponentiell-tid algoritme for $HAMPATH$-problemet ved å modifisere brute-force algoritmen for $PATH$. Vi trenger bare å bekrefte at stien er en Hamilton-sti. Ingen vet om den er løselig i polynom-tid.

$HAMPATH$-problemet har en egenskap kalt **polynom verifiserbarhet**, som er viktig i forståelsen av kompleksitet. Bekrefte eksistensen av en Hamilton-sti er mye lettere enn å bestemme dens eksistens.

DEFINISJON 9.1. *En **verifikator** for et språk A er en algoritme $\mathcal{V}$, hvor*

$$A = \{w | \mathcal{V} \text{ aksepterer } \langle w, c \rangle \text{ for en string } c\}.$$

Vi måler tiden til en verifikator ved lengden av w, så en polynom-tid verifikator kjører i polynom-tid for lengden av w. Språket A er polynom verifer-serbart hvis det har en polynom-tid verifikator.



Figur 16: Graf med Hamilton-sti.

En verifikator bruker ekstra informasjon, representert ved symbolet c for å verifisere at en string w er medlem av A. Denne informasjonen er kalt **sertifikat** eller **bevis** for medlemsskap i A.

For $HAMPATH$-problemet er sertifikatet for en string $\langle \mathcal{G}, s, t \rangle \in HAMPATH$ enkelt nok Hamtilonstien fra s til t.

DEFINISJON 9.1. *NP er klassen av språk som har polynom-tid verifikatorer.*

NP-klassen er viktig fordi den inneholder mange problemer med praktisk interesse. $HAMPATH$ er en del av NP. NP betyr nondeterministisk polynom-tid. Problemer i NP blir ofte kalt NP-problemer.

NTM Følgende er en **nondeterministisk Turing maskin (NTM)** som bestemmer $HAMPATH$-problemet i nondeterministisk polynom-tid. Vi definerer tiden til en nondeterministisk maskin til å være tiden brukt av lengste beregningsgren.

N_1 = på input $\langle \mathcal{G}, s, t \rangle$, hvor $\mathcal{G}$ er en rettet graf med noder s og t:

1. Skriv en liste med m nummer, $p_1, \ldots, p_m$, hvor m er antall noder i $\mathcal{G}$. Hvert nummer i lista er nondeterministisk valgt til å være mellom 1 og m.
2. Sjekke om det er repetisjoner i lista. Hvis så, avvis.
3. Sjekke om $s = p_1$ og $t = p_m$. Hvis en av de ikke stemmer, avvis.
4. For hver i mellom 1 og $m - 1$, sjekk om (p_i, p_{i+1}) er en kant i $\mathcal{G}$. Hvis ingen er, avvis. Ellers, om alle tester er blitt suksessfulle, aksepter.

Analyse Studere hvert steg. I steg 1, nondeterministisk seleksjon er i polynom-tid. I steg 2 og 3, hver del er en enkel sjekk, så til sammen kjøres de i polynom-tid. Steg 4 kjøres også i polynom-tid. **Merk**: denne algoritmen kjører i nondeterministisk polynom-tid.

Et språk er i NP, hvis og bare hvis, det kan bestemmes av en nondeterministisk polynom-tid Turing maskin.

9.2.1 Konvertering av polynom-tid verifikator til NTM

Problem Hvordan konvertere en polynom-tid verifikator til en ekvivalent polynom-tid NTM og vice versa. NTM simulerer verifikatoren ved å gjette sertifikatet. Verifikatoren simulerer NTM ved bruk av de aksepterende grenene som sertifikat.

Bevis La $A \in NP$ og vis at A bestemmes av en polynom-tid NTM $\mathcal{N}$. La $\mathcal{V}$ være polynom-tid verifikator for A. Anta at $\mathcal{V}$ er en TM som skjører i n^k tid og konstruerer $\mathcal{N}$ som følger:

$\mathcal{N}$ = på input w av lengde n:

1. Nondeterministisk velg en string c med lengde max. n^k.
2. Kjør $\mathcal{V}$ på input $\langle w, c\rangle$.
3. Hvis $\mathcal{V}$ aksepterer, aksepter. Ellers, avvis.

Og for verifikator: $\mathcal{V}$ = på input $\langle w, c\rangle$ hvor w og c er stringer:

1. Simuler $\mathcal{N}$ på input w, og behandler hvert symbol av c som beskrivelse av nondeterministisk valg for hvert steg.
2. Hvis denne grenen av $\mathcal{N}$ sin beregning aksepterer, aksepter. Ellers, avvis.

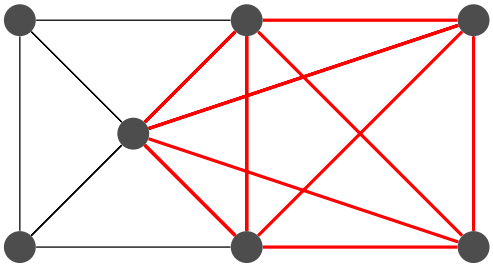
9.2.2 Problem i NP

En **clique** i en urettet graf er en subgraf. En k-clique er en clique som inneholder k noder.

Clique problemet avgjør om en graf inneholder en clique av en spesiell størrelse. La

$$CLIQUE = \{\langle \mathcal{G}, k\rangle | \mathcal{G} \text{ er en urettet graf med en } k\text{-clique}\}.$$

$CLIQUE$ er i NP. Er cliquen sertifikatet?



Figur 17: 5-clique

Bevis Følgende er en verifikator $\mathcal{V}$ for $CLIQUE$.
$\mathcal{V}$ = på input $\langle\langle \mathcal{G}, k\rangle, c\rangle$:

1. Test om c er i en subgraf med k noder i $\mathcal{G}$.
2. Test om $\mathcal{G}$ inneholder alle kanter koblet med nodene i c.
3. Hvis begge er suksessfulle, aksepter. Ellers, avvis.

9.3 P vs NP

P = klassen med språk hvor medlemsskap kan bli *bestemt* raskt. NP = klassen med språk hvor medlemsskap kan bli *verifisert* raskt.

$HAMPATH$ og $CLIQUE$ er i NP, men de er ikke kjent i P. Selv om det er vanskelig å forestille seg, kan P og NP være ekvivalente. Men de kan *ikke bevises* eksistens av et enkelt språk i NP som ikke er i P. Spørsmålet om P = NP er en av de største uløste problemer i historien.

9.4 NP-kompletthet

Hvis problemer i NP krever mer enn polynom-tid, så vil også et NP-komplett (NPC) problem gjøre det. NP-komplette problemer kan reduseres til problemer i NP. Et eksempel på et problem i NPC er $KNFSAT$.

Teorem $KNFSAT$ er NP-komplett.

Bevis Vise at $KNFSAT$ er polynom-tid reduserbart til $3SAT$. Skal da konvertere ϕ til ϕ', hvor ϕ er på knf og er tilfredstillbar $\Leftrightarrow$ ϕ' er på knf og er tilfredstillbar.

La $C_1, C_2, \ldots, C_k$ være klausulene i ϕ. Hvis ϕ er på $3SAT$ settes ϕ til å være ϕ'. Ellers:

1. For hver klausul som har én literal (l_1) endres denne til ($l_1 \vee l_1 \vee l_1$). Denne klausulen må være sann.

2. For hver klausul som har to literaler $(l_1 \vee l_2)$ endres denne til $(l_1 \vee l_2 \vee l_1)$. En av disse literalene må være sanne for at klausulen skal være sann.

3. For hver klausul som har tre literaler gjør man ingen endringer. Klausulen må være sann.

4. For hver klausul som har flere enn tre literaler $(l_1 \vee l_2 \vee \cdots \vee l_m)$. Legger til en ny variabel z_i og erstatter klausulen med $(l_1 \vee l_2 \vee z_1) \wedge (\neg z_1 \vee l_3 \vee z_2) \wedge (\neg z_2 \vee l_4 \vee z_3) \wedge \cdots \wedge (\neg z_{m-3} \vee l_{m-1} \vee l_m)$. Verdien for disse må være sann.

Analyse Konverteringen til $3SAT$ vil gå i polynom-tid. Derav er $KNFSAT \leq_p 3SAT$, og $KNFSAT \in NPC$.

10 Kompleksitet i rom

Definisjon 10.1. *La $\mathcal{M}$ være en deterministisk Turing maskin som stagnerer på alle input. Kompleksiteten i rommet til $\mathcal{M}$ er funksjonen $f : \mathcal{N} \to \mathcal{N}$, hvor $f(n)$ er maksimum antall av tape-celler som $\mathcal{M}$ scanner på hvilket som helst input av lengde n. Hvis kompleksiteten til rommet til $\mathcal{M}$ er $f(n)$, sier vi at $\mathcal{M}$ kjører i rom $f(n)$.*

Hvis $\mathcal{M}$ er en nondeterministisk Turing maskin hvor alle grener stagnerer på alle input, definerer vi kompleksiteten i rommet $f(n)$ til å være maksimum antall tape-celler som $\mathcal{M}$ scanner på hvilken som helst gren av dens beregning for hvilken som helst input av lengde n.

Definisjon 10.1. *La $f : \mathcal{N} \to \mathcal{R}^+$ være en funksjon. Kompleksiteten i rom-klassene, $SPACE(f(n))$ og $NSPACE(f(n))$, er definert slik:*

$SPACE(f(n)) = \{L \mid L$ er et språk som er bestemt av en $O(f(n))$-rom $DTM\}$.

$NSPACE(f(n)) = \{L \mid L$ er et språk som er bestemt av en $O(f(n))$-rom $NTM\}$.

Eksempel SAT kan bli løst med en lineær-rom algoritme. Antar at SAT ikke kan løses med en polynom-tid algoritme, mye mindre med lineær-tid algoritme, fordi SAT er NP-komplett. Rom ser ut til å være mer kraftfult enn tid fordi rom kan reduseres, mens tid kan ikke.

$\mathcal{M}_1 =$ på input $\langle \phi \rangle$, hvor ϕ er en boolsk formel:

1. For hver sanne tilordning til variablene $x_1, \ldots, x_m$ av ϕ:
   - Evaluer ϕ på den tilordningen.
2. Hvis ϕ noen gang blir evaluert til 1, aksepter. Ellers, avvis.

Maskinen $\mathcal{M}_1$ kjører i lineært-rom fordi hver interaksjon av loopen kan gjenbruke samme porsjon av tapen. Maskinen trenger bare å lagre den gjeldene sanne tilordningen, og kan bli utført med $O(m)$ rom. Antall variabler m er maksimum n, lengden av inputen, så denne maskinen kjører i rom $O(n)$.

10.1 Savitchs teorem

Dette teoremet viset at deterministiske maskiner kan simulere nondeterministiske maskiner ved å bruke overraskende lite rom. For kompleksitet i tid vil en slik simulering trenge eksponentiell økning i tid. For kompleksitet i rom viser Savitch teorem at hvilken som helst NTM som bruker $f(n)$ rom kan bli konvertert til en DTM som bruker bare $f^2(n)$ rom.

Savitchs teorem For hvilken som helst funksjon $f : \mathcal{N} \to \mathcal{R}^+$, hvor $f(n) \geq n$, $NSPACE(f(n)) \subseteq SPACE(f^2(n))$.