MAT1030 – Forelesning 29

Kompleksitetsteori

Roger Antonsen - 13. mai 2009

(Sist oppdatert: 2009-05-17 22:38)

Forelesning 29: Kompleksitetsteori

Oppsummering

- Forrige gang startet vi på kapitlet om kompleksitetsteori.
- Vi er interessert i å kunne si noe om hvor lang tid det tar å følge en algoritme.
- Målet er at vi skal kunne sammenlikne tidsbruken til forskjellige algoritmer, for å vurdere hvilken som er mest tidseffektiv.
- I tillegg skal vi kunne vurdere hvorvidt et program basert på en algoritme kan forventes å terminere for de ønskede input innen akseptabel tid.
- Vi følger boka og er i ferd med å se på fire viktige aspekter, eller *tilnærminger*, for å vurdere effektiviteten av en algoritme.
- Første tilnærming: Tell bare de mest tidkrevende operasjonene.
- Andre tilnærming: Hvis tidsbruken varierer for forskjellige input av samme størrelse, ta utgangspunkt i det verste tilfellet.
  - Vi ønsker å finne et *generelt* svar og ikke måtte kjøre algoritmen for alle mulige input.
  - For mange algoritmer avhenger tiden ofte av om vi er heldige med valg av input eller ikke.
  - Når vi skal vurdere kompleksiteten til en algoritme, så er det derfor hensiktsmessig å vurdere tidsbruken i de *verste* tilfellene.
- Det norske ordet "tilnærming" er normalt en grei oversettelse av det engelske "approximation", men det vil gi en riktigere intuisjon om vi erstattet det med forenkling.
- Målet med disse tilnærmingene er at det skal bli mulig å sammenlikne algoritmer, og da viser det seg at det er enkelte forenklinger som gir det mest nyttige bildet.
- Hvis vi oppfatter ordet tilnærming slik at det står for en tilnærmet beskrivelse av kompleksiteten til en algoritme, er dette noenlunde dekkende.
- Vi avsluttet forrige gang midt i introduksjonen til den tredje tilnærmingen og fortsetter der vi slapp.

Tredje tilnærming (fortsettelse)

- Vi kan lage en "dum" algoritme som regner ut $x = 0$ ved rekursivt å multiplisere 0 med 2^n, på følgende måte.

Eksempel.

1 *Input* n [n naturlig tall]

2 $x \leftarrow 0$

3 **For** $i = 1$ **to** n **do**

3.1 $x \leftarrow 2x$

4 *Output* x

- Vi kan finne en annen "dum" algoritme som beregner den samme funksjonen.

Eksempel (Fortsatt).

1 *Input* n

2 $x \leftarrow \frac{3 \cdot 5 - 15}{n \cdot (n+1)}$

3 *Output* x.

- I det siste eksemplet må vi foreta fem regneoperasjoner, mens i det første eksemplet er antall regneoperasjoner avhengig av n.
- For små n vil den første algoritmen faktisk gi raskere svar, også fordi vi der kan arbeide med hele tall, mens vi må arbeide med flytende reelle tall i den andre algoritmen.
- For store input er imidlertid den andre, direkte metoden raskere enn den første.
- Ved å følge tredje tilnærming, stopper all diskusjon om hvilken av to dumme algoritmer som er best.
- Hvis input er lite, vil de fleste algoritmer gi oss et svar innen rimelig tid, og det spiller ikke så stor rolle hvilken algoritme vi velger hvis det er flere mulige.
- Hvis input er stort, kan en ineffektiv algoritme bruke ødeleggende mye mer tid enn en effektiv algoritme.
- Det er derfor at tidsbruken for store inputverdier er det mest interessante.
- Dette er samlet i tredje tilnærming:

Anta at input er stort

Måle kompleksitet med funksjoner

- Vi har sammenliknet algoritmer, og vi har drøftet kompleksitet i visse tilfeller, men vi har ikke sagt så mye om hva slags funksjoner vi vil bruke til å måle kompleksitet med.
- Data er gitt på digital form, og det er naturlig å måle størrelsen på input ut fra hvor mange bit som brukes til å representere input.
- Vi antar fra nå av at størrelsen på input måles i antall bit som brukes i representasjonen.

Eksempel.

- La oss gå tilbake til eksemplet om grafer og problemet om å avgjøre om en graf er sammenhengende eller ikke.

- Siden løkker og parallelle kanter ikke kan gjøre en graf mer sammenhengende, kan vi godt begrense dette problemet til enkle grafer, det vil si grafer uten løkker og parallelle kanter.
- Uten å gå i detalj, kan vi si at for å representere en enkel graf med n noder, trenger vi et antall bit begrenset av $k \cdot n^2$ hvor k er et tall uavhengig av n men avhengig av hvordan vi velger å representere grafen digitalt.

Eksempel (fortsatt).

- Snur vi dette, ser vi at hvis m er antall bit i input, er antall noder i grafen begrenset av et tall $a \cdot \sqrt{m}$ hvor a er en konstant uavhengig av m.
- Da vi lagde en prosedyre for å bestemme om en graf med n noder er sammenhengende eller ikke, forestilte vi oss en prosess i følgende trinn:
  1. Velg ut en node.
  2. I $n - 1$ runder, utvid noden til en maksimal sammenhengende delgraf, ved i hvert trinn å legge til de nye nodene som kan nås fra delgrafen bygget opp så langt ved å legge til en kant.
  3. Undersøk om det fins noder som ikke er med i sammenhengskomponenten.

Eksempel (fortsatt).

- I hvert skritt i hovedløkka, gikk vi gjennom alle kantene, for å se om en av endenodene lå i grafen konstruert så langt.
- Hvis input er på m bit, har vi ca. $m^{\frac{1}{2}}$ trinn i hovedløkka og vi må (i verste tilfelle) teste ca. $\frac{1}{2} \cdot m$ kanter.
- Siden vi opererer med cirkatall, vi skal se på de verste tilfellene og bare på den mest tidkrevende delen av algoritmen, får vi at tidsbruken er omtrent $m^{\frac{3}{2}}$ hvor m er antall bit i input.
- Vi skal etterhvert være litt mer presis i hva vi mener med “cirka”.

Definisjon.

En polynomfunksjon er en funksjon på formen

$$f(n) = a_k n^k + a_{k-1} n^{k-1} + \cdots + a_1 n + a_0$$

Vi antar normalt at $a_k \neq 0$, og da er k graden til funksjonen.

- I noen tilfeller er det viktig å skille mellom polynomfunksjonen og polynomet, som er det definerende uttrykket.
  - Dette er ikke så viktig for oss.
- Hvis graden til en polynomfunksjon f er større enn graden til en annen funksjon g, vil $f(n) > g(n)$ bare n er stor nok.
- Det betyr at hvis kompleksiteten til to algoritmer er gitt ved polynomfunksjoner, kan vi bruke tilnærming 3 og bestemme hvilken som er den raskeste hvis gradene er forskjellige.

Eksempel.

- Vi har gitt et stort tall på binær form og vil undersøke om tallet er et av Fibonacci-tallene.
- Det gitte tallet er representert ved n bit.
- Vi setter av fire n-bits områder R_1, R_2, R_3 og R_4 hvor det gitte tallet ligger i R_1.
- Vi starter med å laste binærkoden til 1 i R_2 og binærrepresentasjonen til 2 i R_3
- Dette tar $n + n$ enkeltoperasjoner (siden vi vi må rydde R_2 og R_3 for søppel).

Eksempel (fortsatt).

- Deretter starter vi en løkke hvor vi
  1. Laster summen av tallene i R_2 og R_3 inn i R_4. Dette tar ca $2n$ regneskritt, siden vi må holde orden på eventuell mente.
  2. Sammenlikner verdien av R_1 og R_4. Er de like, svarer vi JA, er tallet i R_4 størst, svarer vi NEI og er tallet i R_1 fortsatt størst, fortsetter vi prosessen.
  3. Laster tallet i R_3 over i R_2 og deretter tallet i R_4 over i R_3 Dette tar ca $2n$ regneskritt.
- Antall ganger vi må gjennomføre denne løkka er tilnærmet proporsjonal med n ettersom Fibonaccitallene øker tilnærmet eksponensielt.
- Det betyr at vi kan bruke en annengradsfunksjon til å beskrive den omtrentlige tidsbruken, $a \cdot n$ løkker som hver bruker ca $b \cdot n$ regneskritt.

- I det forrige eksemplet så vi at hvis m er et tall gitt på binær form med n sifre, fines det en konstant c slik at antall regneskritt som skal til for å avgjøre om m er et Fibonaccitall eller ikke er begrenset av
$$f(n) = c \cdot n^2.$$
- Vi var ikke spesielt ivrige etter å finne en konkret verdi på c, av forskjellige grunner:
  1. c vil avhenge av hvilket språk vi bruker og faktisk av hvilken maskin vi bruker.
  2. Den virkelige tiden avhenger vel så mye av hvor kraftig maskinvare vi disponerer som hvor liten vi kan få verdien på c til å bli.
  3. Den teknologiske utviklingen gjør at selv store verdier for c er uten betydning for effekten av denne algoritmen.
- Det som ville hjulpet var om vi kunne bringe kompleksiteten ned fra, si $40 \cdot n^2$ til $1.000 \cdot n$.

Fjerde tilnærming og O-notasjon

- Fjerde tilnærming lyder:

  Vi skiller ikke mellom to tidskompleksiteter hvis vekstraten til den ene er et konstant multiplum av vekstraten til den andre.

- Etter at vi nå har innført fire prinsipper for tilnærminger, hvorav tre av dem er mer å betrakte som tommelfingerregler enn matematisk presise regler, skal vi innføre den såkalte O-notasjonen (ikke "null", men bokstaven O).
- Ved hjelp av den blir faktisk bruk av første, tredje og fjerde tilnærming presise.
- Den vil også gjøre det mer presist å avgjøre hva som faktisk er de verste tilfellene (andre tilnærming).

Definisjon.
Tidskompleksiteten til en algoritme er en funksjon f fra $\mathbb{N}$ til $\mathbb{N}$, slik at når argumentet n er størrelsen på input (målt i antall bit), så er verdien $f(n)$ er *det maksimale antallet av mest tidskrevende operasjoner utført* når størrelsen på input er n.

Definisjon.

La f og g være tidskompleksiteter, det vil si, funksjoner fra $\mathbb{N}$ til $\mathbb{N}$.

Vi sier at f er $O(g)$ hvis det fins en positiv konstant c slik at

$$f(n) \leqslant c \cdot g(n)$$

for alle tilstrekkelig store n.

- Med "tilstrekkelig store" mener vi at det fins en n_0 slik at ulikheten holder for alle $n \geqslant n_0$.
- Skulle vi gitt denne definisjonen mer presist, måtte vi bruke kvantorene vi lærte om tidligere i semesteret.
- Da ser definisjonen av at f er $O(g)$ slik ut:

$$\exists c > 0 \exists n_0 \forall n \geqslant n_0 (f(n) \leqslant c \cdot g(n)).$$

- Med denne notasjonen, og i lys av et eksempel vi har sett på før, kan vi si at tidskompleksiteten for den naturlige algoritmen som undersøker om en graf er sammenhengende og har en Eulerkrets er $O(n^{\frac{3}{2}})$, når n er antall bit vi trenger for å representere grafen.
- I motsetning til tidligere formuleringer som "størrelsesorden er...", er dette et presist matematisk utsagn, og dekker alle reelle implementeringer av algoritmen vår.
- Bruken av denne notasjonen er så viktig at vi skal spandere på oss endel eksempler for å få litt intuisjon rundt den.
- Vi minner om at en polynomfunksjon er en funksjon

$$f(n) = a_k n^k + \cdots + a_1 n + a_0.$$

- Vi skal anta at alle koeffisientene er i $\mathbb{N}_0$, det vil si ikkenegative hele tall.
- Videre vil vi normalt anta at $a_k > 0$, og polynomfunksjonen har da grad k.
- Vi skal se på sammenhengen mellom O-notasjonen og polynomfunksjoner.

Eksempel.

- La $f(n) = 3n + 2$ og $g(n) = 2n$.
- Da er f $O(g)$ fordi $f(n) \leqslant 2g(n)$ når $n \geqslant 2$.

Eksempel.

- La $f(n) = 10^6 \cdot n$ og la $g(n) = n^2$.
- Er f $O(g)$?
- Vi kan lett finne en verdi av c som viser dette veldig enkelt:
- $f(n) \leqslant 10^6 \cdot g(n)$ for alle n.
- Vi har også at $f(n) \leqslant g(n)$ for alle $n \geqslant 10^6$.

Eksempel.

- La $f(n) = n^2$ og la $g(n) = 10^6 \cdot n$.
- Er f $O(g)$?
- I dette tilfellet er svaret negativt.
- For å vise det, må vi vise at det ikke fins noen c som duger.
- For å vise at c ikke duger, må vi vise at det fins vilkårlig store n slik at $c \cdot g(n) < f(n)$.
- La $n > 10^6 \cdot c$.
- Da er $f(n) = n^2 > 10^6 \cdot c \cdot n = c \cdot g(n)$.
- Dette viser at svaret er negativt.

Eksempel.

- La $f(n) = 3n^4 + 10n^3 + 2n + 20$ og la $g(n) = n^4$.
- Er f $O(g)$?
- Svaret er *JA*, og vi skal gi et argument som er så generelt at det tjener som bevis for neste teorem.

- La $c = 3 + 10 + 2 + 20 = 35$, dvs., summen av alle koeffisientene i f.
- Husk at $n \geqslant 1$ her.

$$\begin{aligned} f(n) &= 3n^4 + 10n^3 + 2n + 20 \\ &\leqslant 3n^4 + 10n^4 + 2n^4 + 20n^4 \\ &= (3 + 10 + 2 + 20)n^4 \\ &= c \cdot g(n) \end{aligned}$$

Denne metoden kan brukes til å vise følgende teorem.

Teorem.
Hvis f er en polynomfunksjon med grad $\leqslant k$ vil f være $O(n^k)$.

- Hva hvis graden til f er større enn graden til g?
- Vi har sett et eksempel på dette hvor f ikke er $O(g)$.
- Det gjelder helt generelt, og vi skal se på et eksempel som illustrerer det.

Eksempel.
- La $f(n) = n^3$ og la $g(n) = 2n^2 + 4n + 6$.
- La c være en vilkårlig positiv konstant.
- Vi vil vise at det fins vilkårlig store n slik at $c \cdot g(n) < f(n)$.
- Velger vi $n > (2 + 4 + 6)c = 12c$ får vi

$$f(n) = n^3 > c \cdot (2 + 4 + 6)n^2 \geqslant c \cdot g(n)$$

(som i beviset for teoremet).

Vi kan oppsummere dette med følgende observasjon:

Korollar.
a) Hvis f og g er to polynomfunksjoner og f er $O(g)$, vil graden til f være mindre eller lik graden til g.
b) Omvendt, hvis f og g er to polynomfunksjoner slik at graden til f er mindre eller lik graden til g vil f være $O(g)$.

- Vi har definert relasjonen

  $$f \text{ er } O(g)$$

  og det ville vært dumt å ikke benytte anledningen til å repetere litt om relasjoner i denne forbindelse.
- Vi husker at en relasjon R er transitiv hvis

  $$aRb \wedge bRc \Rightarrow aRc.$$

- Er O-notasjonstrelasjonen transitiv?
- La oss drive litt undersøkende matematikk og anta at f er $O(g)$ og at g er $O(h)$.
- Da fins det $c > 0$ og n_0 slik at hvis $n \geqslant n_0$ vil

  $$f(n) \leqslant c \cdot g(n).$$

- Videre fins det $d > 0$ og n_1 slik at hvis $n \geqslant n_1$ vil

  $$g(n) \geqslant d \cdot h(n).$$

- Hvis vi nå lar $n \geqslant max\{n_0, n_1\}$ har vi at

  $$f(n) \leqslant c \cdot g(n) \leqslant c \cdot d \cdot h(n),$$

  så konstanten $c \cdot d > 0$ kan brukes til å vise at f er $O(h)$.
- Dette viser at relasjonen er transitiv.
- Vi husker også at en relasjon R kalles refleksiv hvis aRa for alle a i grunnmengden.
- Er relasjonen

  $$f \text{ er } O(g)$$

  refleksiv?
- For alle funksjoner f og for alle tall n er $f(n) \leqslant 1 \cdot f(n)$, så f er $O(f)$ for alle f.
- Det viser at relasjonen er refleksiv.
- På generelt grunnlag kan vi da definere relasjonen

  f og g har samme kompleksitet

  ved f er $O(g)$ og g er $O(f)$.
- Siden vi tar utgangspunkt i en relasjon som er transitiv og refleksiv, får vi en ekvivalensrelasjon på denne måten.
- Ekvivalensklassene til denne relasjonen kaller vi ofte kompleksitetsklasser og de svarer til mengder av funksjoner hvor alle har samme kompleksitet ut fra forenklingene 1, 3 og 4.
- Dette er et eksempel på hvordan man kan bruke teorien for relasjoner til å gjøre et upresist begrep "vokser omtrent like fort" til et presist begrep.
- To polynomfunksjoner tilhører samme ekvivalensklasse nøyaktig når graden er den samme.
- Med dette avslutter vi innføringen i O-notasjonen.