MAT1030 – Diskret matematikk

Forelesning 30: Kompleksitetsteori

Dag Normann

Matematisk Institutt, Universitetet i Oslo

14. mai 2008

Informasjon

- Det er lagt ut program for orakeltjenestene i MAT1030 denne våren på semestersiden.
- Det blir ikke ordinære gruppetimer fra og med neste uke.
- Oppgaveregningen i morgen blir ren tavleregning, ettersom Roger er bortreist, uten tilgang til e-post, og vikaren har ikke tilgang til styringsfilen for oppgavefoilene.

Oppsummering

- Sist onsdag 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.

Oppsummering

- Kompleksitetsteori er en presis matematisk disiplin, men vi skal ikke drive den så langt.
- Vi vil følge boka, og finne frem til fire aspekter vi kan se på når vi skal vurdere effektiviteten av en algoritme.
- Det første aspektet var at vi skal konsentrere oss om de delene av algoritmen som tar lengst tid.
- Dette kan innebære å se på hvilke enkeltoperasjoner, hvor vi setter en verdi på en variabel, det er som er mest tidkrevende.
- Det viktigste er imidlertid å se på de forskjellige løkkene som skal gjennomkjøres under utførelsen av algoritmen, og å se på hvor mange regnetrinn de består av.
- Vi sammenfattet denne tilnærmingen med

  Tell bare de mest tidkrevende operasjonene.

Oppsummering

Det norske ordet "tilnærming" er normalt en grei oversettelse av det engelske "approximation", men det vil gi en riktigere intuisjon om vi ertattet 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.

La oss så fortsette utforskningen av kompleksitetsteoriens verden.

Kompleksitetsteori

Eksempel

1. *Input* n [n naturlig tall]
2. *Input* $x_{n-1}, \ldots, x_1$ [Hver x_i lik 0 eller 1]
3. $x_n \leftarrow 0$
4. $i \leftarrow 1$
5. **While** $x_i = 1$ **do**
   - 5.1 $x_i \leftarrow 0$
   - 5.2 $i \leftarrow i + 1$
6. $x_i \leftarrow 1$
7. *Output* $x_n \cdots x_1$

Kompleksitetsteori

Eksempel (Fortsatt)

- Denne pseudokoden gir en algoritme for å legge 1 til det binære tallet $x_n \cdots x_1$.
- Hvis vi starter med $n = 20$ og det binære tallet 11111111111111111111, vil **while**-løkka gjentaes nitten ganger, og vi tester om den skal brukes 20 ganger.
- Hvis vi starter med det binære tallet 11111111111111111110 utfører vi testen for **while**-løkka bare en gang.
- Siden den eneste kontrollen vi har over hvor mange ganger denne løkka må gjentas er antall siffre i det binære tallet, lar vi det være målet på hvor lang tid vi bruker.

Kompleksitetsteori

For endel algoritmer vil tiden vi bruker kunne avhenge av om vi er heldige med valg av input eller ikke.

Når vi skal vurdere kompleksiteten til en algoritme, kan det ofte være hensiktsmessig å vurdere tidsbruken i de verste tilfellene.

Det er dette læreboka setter opp som tilnærming nr 2, etter at man har vurdert hvilken del av programmet det er som overskygger de andre delene i tidsbruk:

Hvis tidsbruken varierer for forskjellige input av samme størrelse,

ta utgangspunkt i det verste tilfellet.

Kompleksitetsteori

Eksempel

- Vi har gitt en sammenhengende graf og skal avgjøre om grafen har en Eulerkrets eller ikke.
- Når vi skal vurdere kompleksiteten av en algoritme, er det viktig hvordan vi representerer input.
- Her vil vi anta at grafen er gitt som en symmetrisk matrise, hvor tallet i rad i og kolonne j angir hvor mange kanter det er mellom nodene i og j.
- Tallene på diagonalen skal være det dobbelte av antall løkker ved den tilsvarende noden.
- Graden til en node er da summen av alle tallene langs tilsvarende rad (eller søyle).

Kompleksitetsteori

Eksempel (Fortsatt)

- Vi bestemmer om grafen har en Eulerkrets ved å summere tallene i hver rad til vi finner et oddetall.
- Har grafen en Eulerkrets, må vi summere tallene i alle radene, så hvis n er antall noder, må vi utføre $n(n-1)$ addisjoner og sjekke at n tall er partall.
- Hvis grafen ikke har en Eulerkrets kan vi slippe billig fra det og utføre bare $n-1$ addisjoner.
- Den dominerende prosessen i det verste tilfellet er det å summere tallene i alle radene, så det er de operasjonene vi legger til grunn når vi vurderer kompleksiteten.

Kompleksitetsteori

Eksempel (Fortsatt)

- Anta nå at vi ikke visste at grafen var sammenhengende.
- Er det ødeleggende for kompleksiteten av problemet hvorvidt grafen har en Eulerkrets at vi må undersøke om den er sammenhengende?
- Vi kan uformelt beskrive en prosedyre som undersøker om en graf er sammenhengende på følgende måte:
  - Vi vil finne sammenhengskomponenten til node 1:
  - La A være $n \times n$-matrisen til G hvor $a_{i,j}$ er tallet i rad i og søyle j.
  - La $X_1 = \{1\}$
  - Ved rekursjon for $k < n$, la $X_{k+1} = \{j \leq n \mid \exists i \in X_k(a_{i,j} > 0)\}$.
  - G er sammenhengende hvis $X_n = \{1, \ldots, n\}$.

Kompleksitetsteori

Eksempel (Fortsatt)

- I denne algoritmen har vi en hovedløkke i n trinn.
- Hvert trinn i løkka består av en gjennomløpning av alle par av noder, for å se om det finnes en kant som forbinder den ene noden med sammenhengskomponenten bygget opp så langt.
- Det å undersøke om en graf er sammenhengende krever altså flere operasjoner enn det å undersøke om den har en Eulerkrets, når vi gjør det på denne måten.

Kompleksitetsteori

Eksempel

- Det neste eksemplet som skal belyse tilnærming 2 er Euklids algoritme.
- Euklids algoritme er en selvkallende algoritme som finner det største felles mål for to tall.
- Det største felles målet er det samme som den største felles faktoren.
- Hvis $n \geq m$ er to naturlige tall vil $Euklid(n, m)$ være
  - m hvis m er en faktor i n.
  - $Euklid(m, k)$ hvor k er resten når vi deler n på m når m ikke er en faktor i n.
- Euklids algoritme er rask, selv for store tall.

Kompleksitetsteori

Eksempel (Fortsatt)

- Hvis vi følger Euklids algoritme for to tallpar som ligger nær hverandre ser vi at det likevel kan være forskjeller i hvor raskt algoritmen gir et svar.
  1. $(80, 32) \to (32, 16)$ som gir svar 16.
  2. $(81, 32) \to (32, 17) \to (17, 15) \to (15, 2) \to (2, 1)$ som gir svaret 1
- Hvordan skal vi så kunne finne de verste tilfellene?
- Følg med på den overraskende fortsettelsen!

Kompleksitetsteori

Eksempel (Fortsatt)

- Det minste par av forskjellige tall som gir oss svaret med en gang er $(2, 1)$
- Det minste tallet > 2 som gir 1 som rest når vi deler det med 2 er $1 + 2 = 3$
- Det minste tallet > 3 som gir 2 som rest når vi deler det med 3 er $3 + 2 = 5$.
- Det minste tallet > 5 som gir 3 som rest når vi deler det med 5 er $5 + 3 = 8$

Kompleksitetsteori

Eksempel (Fortsatt)

- Hvis vi begynner med et par av Fibonaccitall (F_{n+1}, F_n) vil Euklids algoritme gi oss paret (F_n, F_{n-1}) i neste omgang.
- Dette er de verste tilfellene, det vil si de tilfellene hvor vi bruker lengst tid i forhold til hvor store tallene er.
- Dette var neppe en anvendelse Fibonacci hadde i tankene, men hvem vet?

Kompleksitetsteori

- Når vi skal vurdere om en algoritme er raskere enn en annen, er det ikke sikkert at det er relevant for alle input.
- Det kan lønne seg å benytte en algoritme som arbeider raskere for store input, der tiden vi bruker faktisk kan ha økonomisk betydning, selv om en annen algoritme er bedre for små input.
- Vi skal først illustrere dette ved å gå gjennom et eksempel i boka, ettersom dette eksemplet i seg selv er viktig.
- Det dreier seg om effektiv eksponensiering, det vil si, om en metode for raskt å kunne beregne store potenser av et tall.
- Eksemplet har samme verdi om vi regner potenser av reelle tall, naturlige tall eller hele tall, så det presiserer vi ikke.

Kompleksitetsteori

Eksempel

- Vi kan definere funksjonen $f(x, n) = x^n$ ved rekursjon som følger:
  - $x^0 = 1$
  - $x^{n+1} = x^n \cdot x$
- Skal vi bruke denne til å beregne 3^8 får vi følgende beregning:

Kompleksitetsteori

Eksempel (Fortsatt)

1. $3^0 = 1$
2. $3^1 = 1 \cdot 3 = 3$
3. $3^2 = 3 \cdot 3 = 9$
4. $3^3 = 9 \cdot 3 = 27$
5. $3^4 = 27 \cdot 3 = 81$
6. $3^5 = 81 \cdot 3 = 243$
7. $3^6 = 243 \cdot 3 = 729$
8. $3^7 = 729 \cdot 3 = 2187$
9. $3^8 = 2187 \cdot 3 = 6561$

Kompleksitetsteori

Eksempel (Fortsatt)

- Som mennesker utfører vi de første multiplikasjonene raskere enn de siste, men for en maskin som arbeider med fullstendige binære representasjoner er en multiplikasjon en multiplikasjon, og tar omtrent like lang tid uansett hvordan faktorene ser ut.
- I realiteten må vi utføre seks mulltiplikasjoner for å beregne 3^8 på denne måten.

Kompleksitetsteori

Eksempel (Fortsatt)

- En alternativ måte å beregne 3^8 på kan være:
  1. $3^2 = 3 \cdot 3 = 9$
  2. $3^4 = 3^2 \cdot 3^2 = 9 \cdot 9 = 81$
  3. $3^8 = 3^4 \cdot 3^4 = 81 \cdot 81 = 6561$
- Her bruker vi bare tre multiplikasjoner i motsetning til seks.
- Skulle vi beregnet 3^{16} ville vi etter den første metoden måtte utføre 8 nye multiplikasjoner, mens vi etter den nye metoden klarer oss med en til:

  $$3^{16} = 3^8 \cdot 3^8 = 6561 \cdot 6561 = 43046721$$

- Dette går faktisk fortere, selv for hånd.

  (Eller gjør det det?)

Kompleksitetsteori

Med utgangspunkt i siste eksempel, skal vi nå beskrive to algoritmer for eksponensiering, og sammenlikne dem.

Vi har sett på hvordan vi kan beregne x^1, x^2, x^4, x^8 og så videre ved gjentatt kvadrering.

Hvordan skal vi for eksempel kunne utnytte dette til å beregne x^{13}?

Vi vet at $x^{13} = x^8 \cdot x^4 \cdot x$

Vi vet at 13, representert som binært tall, er 1101_2

En strategi kan derfor være at vi beregner x, x^2, x^4 og x^8 samtidig som vi ser på binærrepresentasjonen av 13 for å se hvilke av disse tallene som skal inngå som et produkt i x^{13}.

Siden 13 faktisk er gitt ved sin binære representasjon i en datamaskin, er dette veldig gunstig.

Vi skal gi en fullstendig pseudokode for å beregne x^n når n er gitt på binær form, men først skal vi se på et eksempel:

Kompleksitetsteori

Eksempel

- Vi vil beregne 3^{22}
- $22 = 16 + 4 + 2$ så binærformen til 22 er 10110
- Vi vil beregne to følger:
  1. Den ene er 3, 3^2, 3^4, 3^8 og 3^{16} slik vi har sett før.
  2. Den andre er produktet av de tallene i den første følgen som inngår i 3^{22} etterhvert som vi kommer til dem.
- Vi ser på hvilke tallpar vi får underveis, og hvordan vi kommer frem til dem:
  1. $y_1 = 3 = 3^{2^1}$ og $z_1 = 1$ fordi siste siffer i 10110 er 0.
  2. $y_2 = 3 \cdot 3 = 9$ og $z_2 = 9 \cdot 1 = 9$
  3. $y_3 = 9 \cdot 9 = 81$ og $z_3 = 81 \cdot 9 = 279$
  4. $y_4 = 81 \cdot 81 = 6561$ og $z_4 = 279$
  5. $y_5 = 6561 \cdot 6561 = 43046721$ og $z_5 = 43046721 \cdot 279 = 12010035159$
- Svaret er 12010035159.

Kompleksitetsteori

1 *Input* x [x et reelt tall]

2 *Input* k [k antall siffre i binærrepresentasjonen av n]

3 *Input* $b_k \cdots b_1$ [Binærrepresentasjonen av n]

4 $y \leftarrow x$

5 $z \leftarrow 1$

6 **For** $i = 1$ **to** k **do**

  6.1 **If** $b_i = 1$ **then**

    6.1.1 $z \leftarrow y \cdot z$

  6.2 $y \leftarrow y \cdot y$

7 *Output* z

- Denne pseudokoden er litt anderledes enn den som står i boka.
- Skal vi beregne x^2 tar denne prosedyren litt mer tid enn den definert ved rekursjon, ettersom vi her får både å regne ut x^2 og $x^2 \cdot 1$, men for store n er denne algoritmen vesentlig raskere.

Kompleksitetsteori

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

Det vi gjør her er å regne ut $x = 0$ ved rekursivt å multiplisere 0 med 2^n.

Kompleksitetsteori

Vi kan finne en annen 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.

Kompleksitetsteori

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.

Kompleksitetsteori

- 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

Kompleksitetsteori

- 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 bits som brukes til å representere input.
- 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 bits begrenset av $k \cdot n^2$ hvor k er et tall uavhengig av n men avhengig av hvordan vi velger å representere grafen digitalt.

Kompleksitetsteori

- Snur vi dette, ser vi at hvis m er antall bits 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 finnes noder som ikke er med i sammenhengskomponenten.

Kompleksitetsteori

- 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 bits, 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 bits i input.
- Vi skal etterhvert være litt mer presis i hva vi mener med "cirka".

Kompleksitetsteori

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 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 polynommfunksjoner, kan vi bruke tilnærmeing 3 og bestemme hvilken som er den raskeste hvis gradene er forskjellige.

Kompleksitetsteori

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 bits.
- 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).

Kompleksitetsteori

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.

Kompleksitetsteori

- I det forrige eksemplet så vi at hvis m er et tall gitt på binær form med n siffre, finnes 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 $1.000 \cdot n$.