Grådige algoritmer vs. dynamisk programmering

Åsmund Eldhuset

asmunde *at* stud.ntnu.no
folk.ntnu.no/asmunde/algdat/graadighet-dp.ppt

Matrisemultiplikasjon

- Eksempel: Regne ut $A_1A_2A_3A_4A_5$
- Dimensjonene må passe ($a \times b$, $b \times c$, $c \times d$, $d \times e$, $e \times f$)
- Matrisemultiplikasjon er ikke kommutativt (du kan ikke bytte om på rekkefølgen)
- Men det er assosiativt (du kan velge i hvilken rekkefølge du utfører multiplikasjonene)
- Vi kan angi multiplikasjonsrekkefølgen ved å sette på parenteser:
  $A_1A_2A_3A_4A_5 = A_1((A_2A_3)(A_4A_5)) = ((A_1A_2)A_3)(A_4A_5) = A_1(A_2(A_3(A_4A_5))) = \ldots$

Matrisemultiplikasjon

- Multiplikasjon av to matriser med dimensjoner $a \times b$ og $b \times c$ gir abc skalar-multiplikasjoner (multiplikasjoner av tall)
- Forskjellige parentessettinger av et større uttrykk fører som regel til forskjellig totalt antall skalarmultiplikasjoner
- Hvilken parentessetting er best?

Matrisemultiplikasjon

- *Fullt parentessatt uttrykk (FPU)*: enten en enkelt matrise eller et produkt av to utrykk som er fullt parentessatt
- Alle måter å multiplisere sammen matriser på kan skrives fullt parentessatt
- Den beste måten må derfor være et FPU – altså et produkt av to andre FPU, som selv må være optimale

Matrisemultiplikasjon

- Den ytterste multiplikasjonen kan settes på en av følgende måter:
  - $A_1(A_2A_3A_4A_5)$
  - $(A_1A_2)(A_3A_4A_5)$
  - $(A_1A_2A_3)(A_4A_5)$
  - $(A_1A_2A_3A_4)A_5$
- For hver av disse mulighetene må vi (rekursivt) undersøke matrisegruppene vi får og finne optimal parentessetting for dem
- Vi har $\Theta(n^2)$ forskjellige matrisegrupper, så antallet delproblemer er overkommelig

Grådige algoritmer

"*The point is, ladies and gentlemen, that greed, for lack of a better word, is good. Greed is right, greed works.*"
- Gordon Gekko, "Wall Street"

- Grådige algoritmer brukes på optimaliseringsproblemer: vi vil ha den beste løsningen på noe
- Veldig enkelt konsept: gjør det som ser best ut her og nå!
- Dessverre fører ikke dette alltid frem: en grådig algoritme gir ikke alltid optimal løsning

Krav for grådige algoritmer

- Greedy-choice property: Det grådige valget er en del av den optimale løsningen
- Optimal substruktur: Etter at vi har tatt det grådige valget står vi igjen med en mindre utgave av den originale problemstillingen

Myntutdelingsproblemet

- DP-løsning: Du har x kroner igjen å dele ut, og før du kan finne ut hvilken mynt du bør dele ut er du nødt til å prøve alle myntene og se hvilket resultat det gir
- Myntsett hvor grådig algoritme fungerer: Ut fra x kan du med en gang se hvilken mynt det lønner seg å dele ut

Fractional knapsack

- Kan ikke løses med DP, siden antallet forskjellige delproblemer blir uendelig!
- DP krever som regel løsning av alle delproblemer for å kunne velge det best egnede, mens en grådig algoritme gjør et grådig valg og produserer ett delproblem
- Fractional knapsack har greedy-choice property: det er optimalt å ta så mye som mulig av det stoffet som er dyrest per vekt, så det nest dyreste osv.

0-1 Knapsack

- Når du har i elementer igjen å velge blant og w kg igjen i sekken, kan du enten ta med element i eller ikke
  - Hvis du tar den med, har du tjent v_i, men du har bare $w - w_i$ kg igjen
  - Hvis du ikke tar den med, har du ikke tjent noe på det, men du har fremdeles w kg igjen
- Dette er et DP-problem hvor det kan lønne seg å bruke memoisering
  - Alle tallene i siste rad, unntatt det lengst til høyre, er uinteressante
  - Vi trenger ikke alle svarene vi har funnet
  - Memoisering angriper problemet top-down og løser kun de delproblemene som er nødvendige for å finne det endelige svaret

Repetisjon: krav for DP

- Optimal substruktur (innebærer rekursiv struktur)
- Overlappende delproblemer (hvis ikke: splitt og hersk-algoritme)
- Løsning av ett delproblem avhenger av løsningene av andre delproblemer (hvis ikke: grådig algoritme)

Grådig vs. DP

- Begge utnytter optimal substruktur
- Grådig
  - Lokalt optimale valg er globalt optimale; optimal løsning avgjøres ut fra hva som virker best der og da
  - Løses som regel ovenfra og ned; man tar et valg og ender opp med et mindre delproblem
- DP
  - Optimal løsning avgjøres ved å se på optimale løsninger av delproblemer
  - Løses nedenfra og opp; man løser større og større delproblemer ut i fra de mindre delløsningene

Grådig vs. DP

- På grunn av den optimale substrukturen vil man på et visst stadium alltid ha et valg som må tas; de forskjellige valgene fører frem til forskjellige delproblemer
- Med DP må man alltid undersøke *alle* delproblemene som *alle* de mulige valgene fører frem til, før man kan ta en avgjørelse om hva som er best
- Med grådig trenger man ikke å gjøre det: pga. greedy-choice property har man en regel for hvilket valg som skal tas – man tar da det valget og arbeider videre med det ene delproblemet man får

Hvordan bevise korrekthet av en grådig algoritme?

- Trinn 1: bevis at problemet har "greedy choice property": at det grådige valget som tas er med i den optimale løsningen
- Trinn 2: bevis at problemet har optimal substruktur, dvs. at etter at du har gjort det grådige valget vil du stå igjen med et problem som har samme form som det opprinnelige

Hvordan bevise korrekthet av en grådig algoritme?

- Eksempel med Huffman:
- 1. Bevis at i en optimal prefikskoding vil alltid de to tegnene med lavest frekvens stå lengst nede i treet (så det grådige valget er gyldig)
- 2. Bevis at ved å bytte ut de to bokstavene med en node som inneholder summen av frekvensene og finne en optimal prefikskode for dét, får man en optimal prefikskode for det opprinnelige alfabetet

Huffmantrær

- Algoritme for å generere en optimal prefikskoding
- Prefikskoder: binære koder av variabel lengde som er slik at ingen kode er noe prefiks av en annen kode
  - F.eks.: 01 er prefiks av 010, men ikke av 001
  - Med prefikskoder kan vi kode en tekst til en sammenhengende binærstreng uten fare for misforståelser

Huffmantrær

- Huffmantrær kan brukes til mer enn prefikskoder
- Kan brukes på alle problemer hvor:
  - Man har en samling objekter som skal settes opp som *løvnoder* i et *fullt binærtre*
  - Hvert objekt har en kostnad p_i, og man ønsker å minimere $p_1d_1 + p_2d_2 + ... + p_nd_n$, hvor d_i er dybden til objekt i
- Eksempelproblemer:
  - Merging av mange lister (Kortstokker-øvingen)
  - Oppgave 6 fra eksamen i desember 2004

Hvorfor ikke Huffman for matrisemultiplikasjon?

- Man vil kanskje være fristet til å løse matrisemultiplikasjonsproblemet ved hjelp av huffmanntrær
- Parentessettingen kan modelleres slik at matrisene blir løvnoder i et fullt binærtre, slik at vi får samme struktur som et huffmantre
- Dessverre er uttrykket for det vi vil optimalisere forskjellig fra det som huffmantrær optimaliserer

Anbefalte eksamensoppgaver

- Desember 1999: 1i, 2e
- (Januar 1996: 3)
- Desember 1996: 6
- August 1997: 3
- Desember 2004: 6

Spørsmål?