Alg.dat ninja på 8 timer - et utfordrene projekt

Frederik M.J.V MTDT

29. november 2008

(øvings/eksamensforelesning av Åsmund)

1 Kap 1

- Introkapittel, poeng: Hvorfor trenger vi raskere maskiner, dårlig algoritmer så inneffektive at maskinene ikke tar dem igjen

2 Kap 2&3 - Kjøretid

- Introduksjon til analyse: Kommer garantert på eksamen. F.eks:
- Kjøretid for kjente algoritmer
- Kodeanalyse: Ubruklige variabel og funksjonsnavn: Ofte: Hva gjør koden (hvilken algoritme) og hva er kjøretiden
- Rene kompleksitetsanalyseoppgaver. f.eks finne ut hvilken O, Ω eller θ en funksjon hører til.
- $m \lg n$ er den $\theta(n \lg n), \theta(n), \theta(n^2)$, og tilsvarende for O og Ω
- Mulig å lage slemme funksjoner som ikke er O og Ω av den andre. F.eks: n og $n^{1+\sin n}$
- Obs på forskjel om gyldige og tette grenser
- $\theta = O\&\Omega$
- Stryk også konstantledd ol.
- $f(n) = O(g(n))$ og $f(n) \leq cg(n)$ når $n \geq n_0$
- Formell utregning/bevis av en kjøretidsberegning
  $f(n) = 3n^2 - 2n + 4$, $g(n) = n^2$
  $f(n) = 3n^2 - 3n + 4 \underbrace{\leq}_{n \geq 0} 3n^2 + 4 \underbrace{\leq}_{n^1} 3n^2 + 4n^2 = 7n^2$
  Da har vi greid å bevise det med $n_0 = 1$ og $c = 7$
  I foilene er det en tilsvarende greie for Ω
- Analyse av lineær kode:

```
for i=1...n:
  for i=1...m:
    foo();
```

  $n \cdot m$
- Kan bli knot dersom man ikke vet størrelsene på for-løkkene i.e. for djikstra:

```
1. while not Q.empty: -- |V|
2.  u=extract-min(Q) -- lg |V|
3.  for v in v.neighbours: --
4.    relax(u,v)
```

  Tar linje 1 og to for seg og 3 og fire for seg. Totalt sett vil innholdet i den indre løkka kjøres for hver kant dvs relax kjøres E ganger totalt. (ikke gange med V). Releax tar $\log V$ tid dersom man bruker en heap. derfor får man total kjøretid $|V| \lg |V| + |E| \lg |V| = E \lg |V|$ (siden den første er mindre)

3 Kap 4 - rekurensanalyse

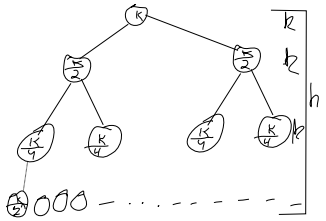
- Ser på tårnet i hanoi (standardeksempel).
- Se også diskretnotater for hvordan man løser rekurensrelasjoner.

3.1 Rekurensligninger

- $T(n) = 2T(n-1) + 1$, trenger basetilfelle: $T(1) = 1$
  $T(k) = 2TK(k-1) + 1 = 2(2T(k-2)+1) + 1 = 2^2 \cdot T(k-2) + 2 + 1 = 2^2(2T(k-3)+1) + 2 + 1 = 2^3 T(k-3) + 2^2 + 2^1 + 2^0 = \cdots = 2^x(T(k-x) + \cdots + 2^2 + 2^1 + 2^0 = 2^{k-1}T(1) + \cdots + 2^2 + 2^1 + 2^0 = 2^{k-1} + \cdots + 2^2 + 2^1 + 2^0 = 2^k - 1$ dvs når man slutter $k - 1 = 1 \Rightarrow x = k - 1$
- Huskeformler: $2^{k-1} + \cdots + 2^2 + 2^1 + 2^0 = 2^k - 1$, $n + (n-1) + \cdots + 3 + 2 + 1 = 0 = \frac{n(n+1)}{2}$

3.2 Rekusjonstrær

- ```
  def MS(list[1...n]):
    a=MS(list[1...n/2]);
    b=MS(list{n/2...n]);
    list=merge(a,b)
  ```
  (ikke in-place)
- $T(n) = 2T(\frac{n}{2})$
  Ser på liste med k elementer



$k = 2^h$, $\lg k = \lg 2^h = h \lg 2 = h$, $h \cdot k = k \lg k$

3.3 Masterteoremet

- $aT(\frac{n}{b}) + f(n)$

Ex Merge sort $T(n) = 2T(\frac{n}{2}) + n$

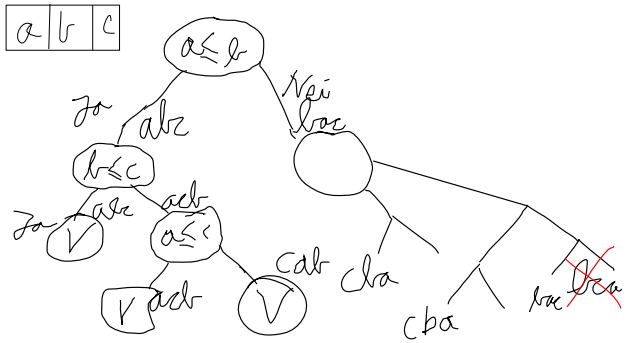
- Tre tilfeller:
  1. $f(n) = O(n^{\log_b^a - \epsilon})$, $\epsilon > 0$
  2. $f(n) = \theta(n^{\lg_b a + \epsilon})$, $\epsilon = 0$
  3. $f(n) = \omega(n^{\lg_b a + \epsilon})$, $\epsilon > 0$; $af(\frac{n}{b}) \leq cf(n)$

4 Ka 6 - heap, heapsort

- Krav til heap: At subnodene skal være mindre en den over. Legger heap i en array
- Sette inn: Setter inn nederst til høyre og skyver tallet ned og setter tallet på plass i stedet.
- Fjerne: Tar det siste elementet og flytter opp (i stedet for rota du har fjernet). Så sjekker vi om heap-egenskapen er oppfylt og lar tallet synke ned igjen.
- Heapsort: (bruke max heap) Først heapyfy: Starter med node en som har barn og bobler nedover til heapegenskapen er oppfylt. Gjentatte ganger fjerner vi ting fra heapen men ikke slette dem fra arrayet. Setter det elementet vi tok ut til slutt i arrayet.

5 Sortering

- Quicksort: Lage et pivotelement og flytte alle elementene over/under elementet om de er større/mindre enn elementet. Worst case gir n^2
- Raskeste: Sortering er i værste tilfellet $n \log n$ (dersom man ikke kan anta noe om dataene)
- Bevise at raskeste er i værste tilfellet $n \log n$.



- For sortering må sorteringsalgoritmene kunne lage permutasjoner på alle måte for en gitt input. dette gir $n!$
- Man vil ha et så balansert tre som mulig. Derfor får man en idél algoritme med et balansert bestemmelsestre med $n!$ løvnoder. høyden blir 2^h
  $n! = 2^h \Rightarrow h = \lg n! \rightsquigarrow \theta(n \lg n)$
- Lineære metoder:
- Radix: Tellesortering på hvert siffer, begynner på det minst signifikante sifferet (det til høyre)
- Stabile sorteringsalgoritmer: Bytter ikke på den innbyrdes rekkefølgen på innbydres like elementer Opp til d siffere i hvert tall, n tall, k tall i sifferet (grunntallet i tallsystemet)) $\theta(d(n+k))$
- Slem oppgave: $[0, n^2>$, f.eks $[0, 10000]$ (n tall) sortere i lineær tid utifra n. Greit nok, så $[0, n^x>, nx$. Jobber heller i n-talls systemet. $\theta(x(n+n) = \theta(xn)$
- Pseudopolynomialitet: $\theta(n)$ måler størrelsen på tallet i logaritmen $n = 2^{\lg n} = 2^w$, $w^{\lg} n$
  ```

- I forbindelse med 0-1 napsack: $\theta(nk)$ der n er antall elementer, mens k er størrelsen på sekken og dermed må gjøres om
- Når man jobber med lister antar man at tallene er konstant endelig store.

6 Ordensstatistikk (bl.a. select)

- Finne det n-te (største/minste etc) elementet i en liste.
- I stedet for å sortere først modifiserer man quicksort ved å ta partition og ser på den delen med riktig antall. $n + (1 + \frac{1}{2} + \frac{1}{4} \dots) = 2n$

7 Hashfunksjoner

- Gjøre en mengde tall med store verdier til få tall. f.eks med modulo. Dersom man får kolisjoner så må man bruke en lenka liste. Finnes andre metoder for kollisjoner som hopper videre i tabellen. (dårlig for sletting)
- Kjøretid $\theta(1 + \frac{n}{m})$ $\alpha = \frac{n}{m}$ = n antall innlagte elementer, m størrelsen på tabell.

8 Binære søketre

- Sette inn 1 element n operasjon dersom treet er ubalansert, 1 elemnt bestcase: 1
- Sette inn n elementer: $1 + 2 + 3 + 4 + \dots = n^2$ worstcase, $n \log n$ (balansert tre)

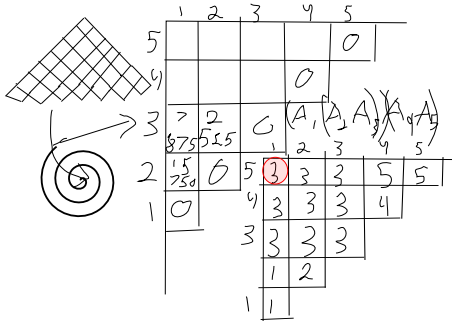
9 Dynamisk programmering

- Memoisering: Hindre at man løser overlappende delproblemer omigjen ved å legge inn tidligere løsninger av delproblem i en tabell også slå opp den dersom man trenger svarene på delproblemene.
- Fyller ut tabellen nedenfra og opp. Dropper hele rekusjonen og regner ut tabellen nedenifra og opp i stedet.
- Krav:
  - Rekursiv struktur
  - Optimal delstruktur: Løsningen består av de mindre løsningene.
  - Plass til tabellen

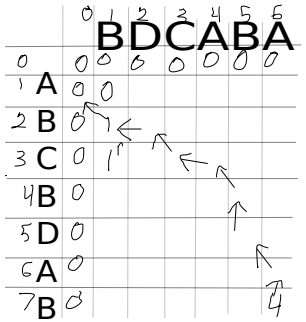
Ex $1, 4, 5, 10, 8 = 1+1+\dots+1 = 1+1+1+5 = 4+4 = 4+1+1+1+1$. Flere muligheter og må sjekke alle kombinasjoner siden man ikke vet hvem som er best.

9.1 DP-algoritmer fra pensum

- Matrisemultiplikasjon: (prøver å multiplisere de minste matrisene først) $A \underbrace{BC}_{AX} \underbrace{DE}_{Y}$ $\underbrace{AX \underbrace{DE}_{Y}}_{AX_{ZY}}$ eller noe sånt. Vi ser at vi har en rekursiv substruktur. Optimal substruktur: Siden den beste måten å splitte ABC er også bra for å splitte $ABCDE$
  $A_1A_2A_3A_4A_5$ Ser på de forskjellige multipliseringspermutasjonene. Med basetilfellet at man skal multiplisere 1 matrise (som er gratis). Da får man en 3.kantet DP-tabell.

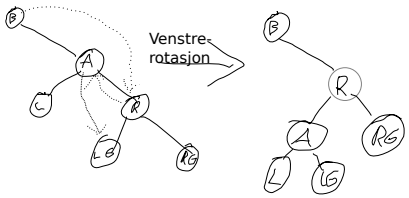


- Longest common subsequence: To strenger **B**D**CA**B**A**, A**BCB**D**A**B Sjekker de siste og stryker dem og husker at de skal være med. Dersom vi har to forskjellige bokstaver vet vi at ett av dem må kastes. Prøver hva som skjer hvis man stryker den ene og prøver å finne den lengste. Så velger man å stryke den andre og ser hvor bra det går. Ser også på hvordan det går å kappe bort en tom streng. Dersom det ikke er noen forbedring kopierer du det forrige tilfellet.
  Spare minne ved å ta vare på raden over, men du trenger pilene for å komme deg opp.



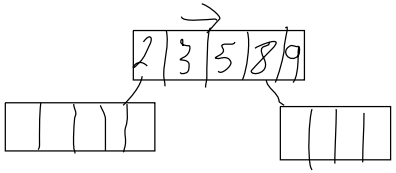
10 Trær

10.1 Rotasjoner



Skal ikke kunne hvordan red-black trær gjør rotasjoner. Bare vite hvordan rotasjoner foregår og at Red-black trær er selvbalanserende.

10.2 B-trær



Bruker hovedsaklig splitting for å balansere. Fint når du jobber med noe på disk. Bruker f.eks en sektor som lengde på en node. Bør lese mer selv i boka.

10.3 Div DP

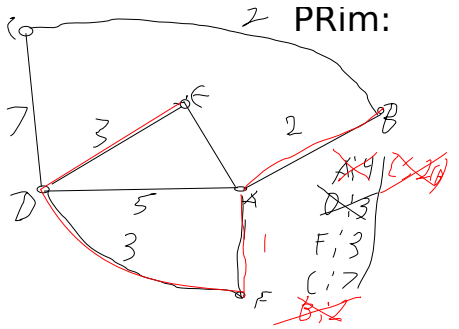
- Vanslikge oppgaver: DP,LP, Flyt/sirkulasjon. Skjelden (men ikke aldri) grådige algoritmer.
- Grådig: Optimal substruktur og grådig-valg egenskapen. Man bør prøve å finne et bevis dersom man prøver med en grådig algortime. evt. prøve å finne moteksempler. Les også Activity selection (dukker av og til opp vrier). Mer avanserte vil lese beviset for huffmanntre for å vise at grådige er korrekte: Enten beviser man at det grådige valget ikke er feil eller ekvivalent med en optimal løsning, 16.4 er ikke pensum.
- De siste prasisoppgavene er trening på DP. Tvilling-DNA: Vri på LCS, Edit distance. Seddeltrykkeiet er DP. (ripp-off fra programmeringskonkuranse, hevn over oss fra Åsmund).

11 Grafer ol

- Kjenne (og kunne bevise) basisobservasjon for konstruksjon av minimale spenntrær: Teorem 7.1.3 s363
- Respekterer subsettet hvis et kutt ikke krysser noen deler av et MST. Dersom du har et repekterende kutt så er det garantert trygt å legge til den med den minste kantvekta.

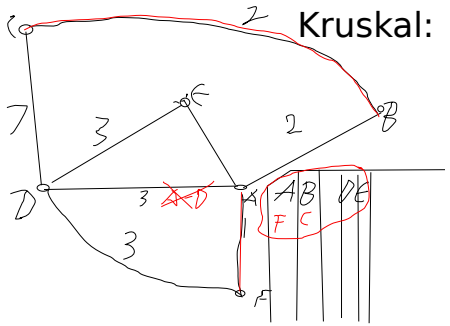
11.1 Prims og kruskals algoritme for minimale spenntrær

- Prims algoritme:



(Til høyre, liste over hvilke noder vi har, bruker prioritetskø på de gjennværende nodene)

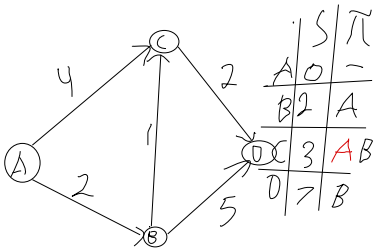
- Kruskals algoritme: Sorterer alle kantene og behandeler dem etter tur (den minste først)



Nodene må gå mellom forskjellige trær for å hindre sykler. (må sjekke alle nodene i begge trærne for å se om det er noen felles noder) Union find er vanskelig å lage effektivt. Å slå sammen trær er også slitsom.

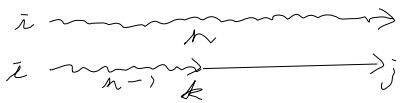
12 Dijkstra & bellman ford

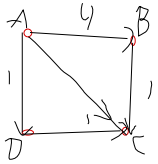
- Bellman ford: kjør releax på alle kantene sinnsynkt mange $(V-1)(+1$ for sykler) ganger. Bevis man har funnet korteste vei til nodene som er like mange kanter fra startnoden som ganger vi har kjørt releax
- Tåler negative kanter, kan finne dem ved å kjøre siste gangen og sjekker at den er konsistent. (at det ikke er mulig å finne en enda kortere vei)
- ITK (på Samfundet) kupper tavla og definerer releax i Lisp som ($defunreleaxsleep$)
- Dikstra: Mer effektiv



Kjøretid $V \lg V + E \lg V$, antar at grafen er connected og får $E \lg V$ (med heap) eller Antar at det kun finnes en kant mellom alle noder derfor vil en arrayimlementasjon kjøre på $O(V^2)$

13 Alle til alle korteste vei





- Man kan dele opp en lang vei i en kortre vei pluss resten (den siste kanten).

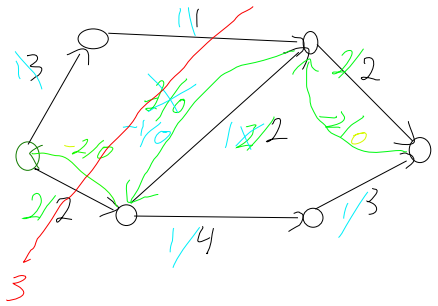
$$\begin{bmatrix} & A & B & C & D \\ A & 0 & & 3 & \\ B & 0 & 0 & 0 & \\ C & & & 0 & \\ D & & & & 0 \end{bmatrix}$$

Betrakter en vei med en bestemt lengde og ser så på hvordan den kommer til neste kant.

- Floyd warshall ser i stedet på stier via en viss node.
- Djigstra er raskere på en spearse graf.

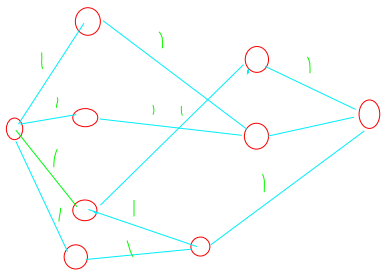
14 Flyt

- Løsing og egenskaper

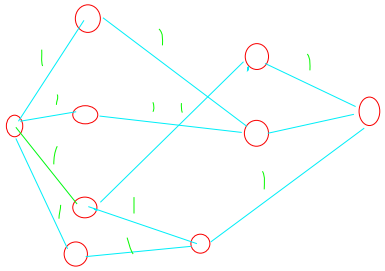


Egenskaper: Ikke overskriv flyt, skew symmetry dvs. flyt en vei er -flyt andre veien. Flytkonserveringsegenskap kun kide og sluk kan skape og sluke flyt. kapasitet c, flyt f, residual (rest) r: $c - f = r$
Ford fulkerson: Så lenge det finnes flytforøkende stier skal du sende så mye flyt som mulig i denne stien. (bruk residualgrafen)
Edmons-karp: Kjøre Ford fulkerson med bredde først søk: Bruk BFS, avansert bevis tar lang tid.
Snitt - en linje som går over alle noder mellom kilde og sluk.
To måter å bevise at man ikke kan få større flyt: Max-flow min cut 7.6.6 (657): Maksimal flyt - ingen flytforøkende stier - ingen flaskehalser med ekstra kapasitet.

- Bipartite matching



I edmonds karp er flytene alltid heltall og dersom man setter kapasitet 1 vil kantene enten være opptatt eller ikke.
Dersom man har vekting av de forskjellige tilfellene får man et sirkulasjonsproblem. Man har i stedet nedre grenser for flyt og en konstnad ved kantene. Finne billigst mulig løsning. Hver kant har: nedre grense l, øvre grense u og kostnad c.

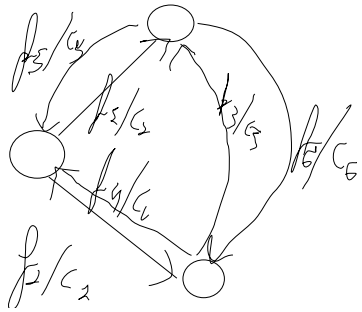


Dersom man bruker flyt risikerer man at flyt på avveiet og det ødelegger seg. Max flyt er et spesialtilfellet for sirkulasjonsproblemet. Maks flyt til sirkulasjon: Nedre grenser: nedre grense er minus kanten motsatt vei: nedre grense er minus kapasiteten andre veien (i den orginale grafen). Alle får samme kostnad, 0.Lager loop fra source til sink med flyt $[0, \infty]$ med kostnad $-\infty$ (der ∞ kan være veldig mye dersom du vet at det ikke kommer mer flyt gjennom grafen).

15 Lineær programmering

- $x_1 + 4x_2 - 8x_3$ med krav: $\begin{cases} x_1 > 0 \\ x_2 + x_1 \leq 5 \\ x_3 - x_2 + x_1 \leq 3 \end{cases}$ (der programmet tilsvarer en tabell med ulikheter/ligninger)
- Hvordan formulere problemer som lineære program:

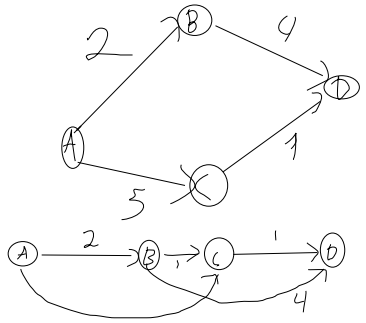
– Flyt: ikke overskride kapasiteten, flyt inn er flyt ut, flyt en vei er minus flyt den andre veien.



Maksimer $f_1 + f_2$ $\begin{cases} f_1 \leq c_1 & \text{kapasitet} \\ f_2 \leq c_2 & \\ \vdots & \\ f_1 = -f_4 & \text{skew symmetry} \\ f_2 = -f_5 & \\ f_1 + f_6 = 0 & \text{flyt bevart} \forall \in V - \{src, snk\} \end{cases}$

– For sirkulasjonsproblemer: Legge til krav om at flyen må være større eller lik nedre grense., i funksjonen er målet å minimere eller maksimere hhv. konstnaden eller inntektene i grafen $f_1 \cdot c_1 + f_1 \cdot p_2 + \ldots$

– Korteste vei:

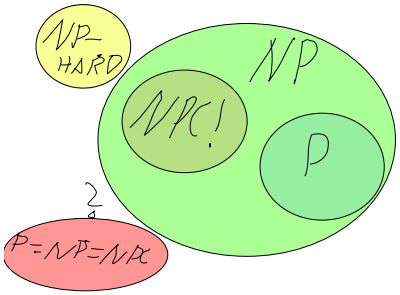


d_B, d_C, d_D, Krav: $\begin{cases} d_A = 0 \\ d_A + 5 \geq d_C \\ d_A + 2 \geq d_B \\ d_C + 1 \geq d_D \\ d_B + 4 \geq d_D \end{cases}$ Vi skal ikke minimalisere avstanden til d (d_D), men maksimere. Prøver å trekke grafen lengst mulig men blir holdt igjen av kra-

vene. sirkulasjon = minimum cost flow - source of sink.
multicommodityflow - typer varer

16 NP-kompletthet

- Krav: Definere P,NP,NPC,NP-hard.
- Algoritmene våre er ofte optimaliseringsproblemer, kan ofte gjøres om til bestemmelsesproblemer ved å si "eksisterer det en... bedre enn ...".
- Man kan svare på optimaliseringsproblem i logaritmsik tid ved binærsøk med en bestemmelsesproblemer.



NP=nondeterministisk polynomial, kan verifisere en måte å løse det på i en polynomisk tid dersom løsningen er gyldig.
P=et problem som kan bekreftest på polynomisk tid.
NPC= de vanskeligste problemene i NP
Div eksponensielle kjøretiden: $2^n \leq n! \leq n^n$
NP-Hard - ikke-verifiseringsproblemer som er like vanskelige som NP komplette problemer. s986 - Et problem som er like vanskelig som er like vanskelig som et problem i np men som ikke nødvendigvis ligger i np er np hard (dvs NP$\in$ HP-hard)
En eller annen gal mann har bevist at alle problemer i NP kan skives om til Cirquit-SAT.
p $A \rightarrow B$ (lite jobb) f.eks Flyt$\rightarrow$ sirk som kan løse flyt har vi bevist at sirk er minst like vanskelig som flyt: flyt$\leq_p$ sirk.
Bruker det for å bevise at noe er NP-komplett. Huske å redusere riktig vei kjent $\rightarrow$ ukjent (siden det kjente er vanskelig og hvis man kunne løse det kjent via ukjent og ukjent var lett er det ikke vanskelig lengere)