TDT4120 Algoritmer og datastrukturer kompendium

Martboi

27. november 2018

Innhold

1. Problemer og algoritmer

1.1 Introduksjon

Problem: relasjon mellom input og output
Instans: bestemt input for et problem
Problemstørrelse n: lagringsplassen til en instans
Algoritme: prosedyre som løser et problem, altså for et bestemt input gir oss riktig output i henhold til problemet.
Random-access machine(RAM): enkel, abstrakt maskin for å analysere algoritmer. Den er helt sekvensiell, støtter integres og flyttall, og kan kun gjøre enkle operasjoner som aritmetikk, kontrollstrukturer, minneoperasjoner etc.

1.2 Kjøretid

1.2.1 Asymptotisk kjøretid

Θ: kjøretiden ligger mellom en øvre og nedre asymptotisk grense definert av Θ.
Ω: kjøretiden har en øvre asymptotisk grense definert av Ω.
O: kjøretiden har en nedre asymptotisk grense definert av O.
Vi kan bruke ω og o for en strengere grense, i stedet for $\geq$ har vi $>$ og motsatt.

1.2.2 Best, average og worst case

Best-case: beste mulige kjøretid for en gitt størrelse
Worst-case: verste mulige
Average-case: forventet, gitt en sannsynlighetsfordeling

1.2.3 Amortisert analyse

Se på gjennomsnitt per operasjon etter at mange operasjoner har blitt utført. Finne øvre grense på kostnad $T(n)$ for en sekvens med n operasjoner, finner så amortisert arbeid som $T(n)/n$.

1.3 Splitt og hersk

Generelt i algdat bruker vi splitt og hersk teknikken: del opp problemet i delproblemer, løs disse delproblemene og kombiner til en fullstendig løsning av problemet. Dette kan gjøres ved hjelp av rekursiv dekomponering eller løkkeinvarianter. Vi trenger en invariant, noe som er sant så langt, en initialisering og en terminering. Vi kan da bruke induksjon til å vise at løsning av delproblemer $\rightarrow$ løsning av problemet.

1.4 Insertion sort

Insertion sort er en enkel sorteringsalgoritme som går ut på å ta hvert element i lista og flytte det oppover til det er på riktig plass. Insertion sort har average og worst-case kjøretid på n^2 og best-case n.

1.5 Noen viktige summer

$$\sum_{i=0}^{n-1} i = \frac{n(n-1)}{2} \tag{1}$$

$$\sum_{i=0}^{n-1} 2^i = 2^n - 1 \tag{2}$$

2. Datastrukturer

2.1 Stakker og køer

Stakk: kun adgang øverst a.k.a. sist inn, først ut(LIFO) , funksjoner: isEmpty(), push(), pop(). Kø: først inn, først ut(FIFO). Implementeres som ringbuffer, altså at vi holder styr på head og tail, og når tail eller head treffer enden av arrayet så setter vi den til toppen igjen. Funksjoner: enqueue(), dequeue(), isEmpty().

2.2 Lenkede lister

Består av noder som peker på neste(og eventuelt forrige). tar lineær tid å slå opp, konstant tid å sette inn/slette elementer hvis vi har noden. Funksjoner: listSearrch() som itererer fra head til vi finner ønsket element, listInsert() som setter inn element som ny head, listDelete() som setter nodens neste til å peke på nodens forrige og motsatt, så den forsvinner fra lista.
Listene kan enten implementeres som flere arrays for key, next, prev eller single-array der ting kommer etter hverandre.

2.3 Hashtabeller

Vi har et array av lenka lister. Dersom vi bruker direkte adressering har vi at nøkkel=indeks. Med hashtabeller er modifisert nøkkel indeksen, i.e. slår opp på h(k), ikke k, der h er hash-funksjonen og k er nøkkelen. Kjeding: hver posisjon i tabellen har en lenket liste, ved kollisjon fra hashfunksjonen legger vi bare til i lista på den indeksen. Alternativt kan vi bruke flere hashfunksjoner for å deale med kollisjoner. Med mange kollisjoner får vi lineære lange lister. Med lineært stor tabell som er jevnt fordelt får vi plutselig konstant forventet kjøretid, med få kollisjoner. Viktig å lage uniformt fordelte hashfunksjoner. For statiske datasett kan vi lage en custom hashfunksjon som har worst-case konstant tid. Grunnleggende hashfunksjoner: k mod(m), floor(m(kA - floor(kA)).

2.4 Dynamiske tabeller

Det er kjipt å allokere nytt minne, det tar lineær tid... Derfor, når for eks. en stakk eller lenket liste er full, så allokkerer vi masse nytt minne.

3. Splitt og hersk

3.1 Rekurrenser

En rekurrens er en rekursiv likning, som brukes til å finne kjøretiden til en rekursiv algoritme. Vi finner løsning vha. iterasjonsmetoden, rekurrenstrær, masterteoremet og verifiserer med substitusjon.
Iterasjonsmetoden: Eks. T(n) = T(n-1) + 1 = T(n-2) + 2 = ... = n (når T(1) = 1, dette er vanlig å anta for grunntilfellet). Dette kan visualiseres med rekurrenstrær.
Verifikasjon med substitusjon: gitt for eks. løsningen vår T(n-1)=n-1, så viser vi at T(n) = T(n-1) + 1 = n - 1 + 1 = n. Må også vise grunntilfellet.

3.2 Maximum-subarray-problemet

Finne et subarray som har maks endring fra start til slutt. Bruker D&C gjennom binærsøk for å løse problemet. Først lager vi et nytt array der hvert element er endringen fra forrige verdi til verdi nå. Så bruker vi rekursjon til å dele opp problemet slik at vi finner max subarray i left og right rekursivt. Da er det bare å slå sammen left og right til å finne max subarray i et subarray. Løser dette med rekursjon i linearitisk kjøretid baby.

3.3 Binærsøk

Sortert sekvens, deler da sekvensen rekursivt i to, og leter i riktig del. Dette gir oss logaritmisk kjøretid, dobling av element øver kjøretid med 1. Iterasjonsmetoden: $T(n) = T(n/2) + 1 = T(n/4) + 2 = ... = \lg(n) + 1$.

3.4 Mergesort

Sorter venstre og høyre halvdel rekursivt, flett sammen fra bunnen og opp.
Prosedyre: del opp sekvensen i minste deler, og sorter hver av de, så sorteres et nivå opp osv. helt til hele sekvensen er sortert.
Rekurrens: $T(n) = 2T(n/2)+n = 4T(n/4)+2n = ... = 2+\lg(n)n = \lg(n)n+n$, altså linearitmisk kjøretid.

3.5 Quicksort

Ta alle høye verdier i en halvdel, alle lave verdier i andre halvdel, gjør så det samme for hver del rekursivt helt til hele sekvensen er sortert. Samme best/avg som merge, men worst case: $T(n) = T(n-1) + n = ... = n + (n-1) + ... + 1 = n(n+1)/2$, altså kvadratisk :(

3.6 Randomisering

Så ingen input alltid er kjip. Dette så vi var et problem med quicksort. Dersom vi i stedet velger et tilfeldig splitteelement er vi sikra at sortert liste gir fucked kjøretid.

3.7 Masterteoremet

Vi har at $a^{\log n} = n^{\log a}$. Lage generell rekurrens: $T(n) = aT(n/b) + f(n) = f(n) + af(n/b) + a2f(n/b2) + a2T(n/b2) + ... + a^{\log_b n}$. Dette gir følgende resultater:
Dersom $f(n) = O(n^{\log_b a - \epsilon}$ har vi $T(n) = \Theta(n^{\log_b a})$.
Dersom $f(n) = \Theta(n^{\log_b a}$ har vi $T(n) = \Theta(n^{\log_b a} \lg n)$.
Ellers er $f(n) = \Omega(n^{\log_b a + \epsilon}$, og vi får $T(n) = \Theta(f(n))$.

3.8 Variabelskifte

Akkurat det det høres ut som, bare bytt ut stygge ting med nye variable for å gjøre rekurrenser lettere å løse, for eks. m = lg(n).

4. Sortering i lineær tid

Vi har en sorteringsgrense på worst-case $\Omega(n \log n)$, hvordna kan vi bryte den? Jo, vi antar noe mer...

4.1 Tellesortering

Maksverdi k til elementer er input til funksjonen, 0 er antatt minimum. Vi itererer gjennom lista og teller antall av hvert element i en ny tabell. Vi itererer igjennom telletabellen og gjør den kumulativ, altså sier at hvert element er summen av alle bak + current. Så itererer vi gjennom den kumulative telletabellen fra toppen og setter inn riktig tall i en ny sortert tabell hele veien ned. Vi må dekrementere den kumulative tabellen når vi setter inn et nytt tall på riktig sted i den sorterte tabellen. Dette gir oss $T(n) = \Theta(n + k)$.

4.2 Radikssortering

Vi itererer fra 1 til d og sorterer tabellen etter sifferet i. Krav: sorteringsalgoritmen må være stabil, altså må ikke bytte om rekkefølge på like verdier. Dette gir oss $T(n) = \Theta(d(n + k))$.

4.3 Bucketsort

Antar uniformt fordelt tabell A. Vi starter med å lage en ny tabell B for bøttene"våre. Så itererer vi gjennom tabellen A og legger til hvert element i B[floor(nA[i])]. Så sorterer vi hver bøtte som da er lenka lister med forventet konstant lengde. Så slår vi sammen alle bøttene i

riktig rekkefølge. Dette blir en form for hashing. Dette blir en $\Theta(n)$ average case, men $\Theta(n^2)$ worst case.

4.4 Randomized select

Antar distinkte verdier. Vi bryr oss bare om å finne plass k i den sorterte lista, for eks. median eller max/min. Blanding av quicksort og binærsøk". Vi partisjonerer lista som i quicksort, og leter enten til venstre eller høyre som i binærsøk. Da sorterer vi ikke mer enn det vi trenger! $\Theta(n)$ for average case, men $\Theta(n^2)$ worst case når pivotposisjonen står alene.

4.5 Select

Hvis vi vil garantere $\Theta(n)$ worst-case må vi finne en god pivot. Enten kan vi ha med pivot som input, eller så deler vi tabellen i 5 grupper, finner medianen i hver gruppe ved å sortere først, bruker så select funksjonen for å finne medianen av tabellen av medianene. Da har vi funnet en ganske rimelig pivot, og kan kjøre select med denne pivoten.

5. Rotfaste trestrukturer

5.1 Trær

Urettet graf $G = (V, E)$: mengde av noder og kanter
Fritt tre: urettet graf som er sammenhengende og asyklisk
Rotfast tre: fritt tre med angitt rotnode
Ordnet tre: barna har en ordning
Posisjonstre: hvert barn har en posisjon, barn kan dermed mangle
Binærtre: Posisjonstre der hver node har to barneposisjoner
Komplett binærtre: Alle løvnoder har akkurat samme dybde
Fullt binærtre: alle interne noder har to barn
Balansert binærtre: alle løvnoder har ca. samme dybde
Dybde: antall kanter fra rota
Høyde: maksimal dybde
N løvnoder $\rightarrow lg(N)$ dybde, $(2N - 1)$ noder totalt når treet er komplett

5.2 Hauger

Hauger har de største elementene på toppen, node er større enn barna sine (dette er da maks-haug, kan også ha min-haug). Implementeres som arrays: $[0, 1_l, 1_r, 2_{ll}, 2_{lr}, 2_{rl}, 2_{rr}, ...]$, må da gange med to for å få barna.
Hvordan vedlikeholde (heapify) treet? Se på venstre og høyre node, sjekke om noen av barna er større, og så bytte om det barnet med foreldernode. Så går vi videre ned rekursivt og forsetter å vedlikeholde. Vil være behov for både en heapify som går nedover og en som går oppover (bubble og sink).
Hvordan bygge treet? Itererer fra length/2 ned til 1 (altså over alle interne noder fra bunnen)

og kjører heapify på hver node. Dette er en lineær operasjon.
Hauger som prioritetskøer: Vi vet at maksimum er rotnoden. Hvordan fjerne maks? Sette inn siste element som rota, og så reparere rotnoden.
Hvordan øke nodeverdi? Sette inn ny verdi, og kjøre heapify oppover, ikke nedover.
Hvordan sette inn node? Sett inn node i bunnen, og så reparere oppover.
Alt dette tar logaritmisk tid...

5.3 Heapsort

Selection sort med en haug! Får inn tabell → Build heap → Iterer fra bunnen til toppen, bytt ut indeks med rota, heapify rota. Det som i praksis skjer da er at vi sender det største elementet til rota, plukker det ut, og gjentar til vi har hele tabellen sortert. Dette gir oss nlgn worst case og average case, n best case.

5.4 Binære søketrær

Binærsøk som datastruktur ish. Her er ordninga at venstre barn er mindre enn noden, høyre barn er større enn noden. Traversering (kan ha pre-order, in-order, post-order, her in-order): ganske enkelt walk(left), print(node), walk(right). Søk: hvis mindre søk venstre barn, hvis større søk høyre barn and repeat...
Innsetting: søk til vi finner der den skal være i bunnen, og så er det bare å sette inn der (i et array må da alle senere elementer flyttes et hakk...) Vi finner maks/min ved å iterere nedover kun til venstre/høyre.

6. Dynamisk programmering

6.1 Dekomponering

Optimal delstruktur: det finnes optimal løsning bestående av optimale delløsninger.
Velfunderthet: enhver avhengighetskjede ender med et grunntilfelle, vi krever velfunderte relasjoner for at rekursjonskjeden skal terminere. Det betyr ikke at vi ikke kan ha forgreininger som samles igjen, bare ikke sykliske grener eller grener som ikke termineres.

6.2 wtf er DP?

Finn strukturen til optimal løsning, definer verdien av optimal løsning rekursivt, kalkuler verdien av optimal løsning, konstruer optimal løsning fra kalkulert informasjon.
Generalisering av divide and conquer der vi potensielt møter samme delproblemer eksponentielt mange ganger, altså at vi har overlappende delproblemer - derfor memoisere løsninger på delproblemer. Memoisering: gi funksjoner hukommelse: "har jeg fått disse inputene før? Da har jeg dette lagret, og trenger ikke regne ut på nytt". Dette er basically det vi har gjort hele tiden med D&C, men nå har vi overlappende delproblemer i tillegg. Merk at vi bruker memoisering i top-down, og iterasjon i bottom-up. I bottom-up løser vi de enkleste delproblemene først, så

bruker vi bare alltid de enklere løsningene direkte fra tabell oppover.
En delproblemgraf viser hvordan de ulike delproblemene er avhengig av hverandre.

6.3 Eksempel I: stavkutting

Input: lengde n, priser p_i for lengder $i = 1..n$
Output: lengder $l_1 + ... + l_k$ blir n med maksimal totalpris $p_l1 + ... + p_lk$
Vi kan løse dette ved å iterere gjennom fra 1 til n, og se om prisen for i + prisen av de tidligere gir en bedre maks pris. Vi kaller da Cut-Rod rekursivt nedover for å finne beste løsning. Vi kan bruke DP til å lagre prisene av delstavene vi finner underveis. Ved å lagre løsningene underveis får vi lineær kjøretid!

6.4 Eksempel II: least common subsequence

Input: to sekvenser X og Y
Output: en felles subsekvens Z og indekser $i_1...$ og $j_1...$
Itererer gjennom de to sekvensene fra enden, dersom vi har like bokstaver klipper vi begge av og vi har +1 lengde i Z, ellers blir det to nye delproblemer der vi enten klipper av en bokstav fra enden på X eller på Y. Vi fyller inn i et 2D regneark for alle kombinasjoner etterhvert som vi regner de ut. Ved å fylle ut regnearket kan vi da finne lengste subsekvens, hvis vi også lagrer hvert steg for hvert element i tabellen.

6.5 Eksempel III: 0-1 ryggsekk

Input: verdier $v1$,..., vekter w_1,..., kapasitet W
Output: hvilke objekter gir mest verdi gitt kapasiteten W
Lett å løse den fraksjonelle varianten: bare ta med deg så mye som mulig av den dyreste gjenstanden, og forsetter nedover. I 0-1 varianten derimot er det enten eller. Dekomp: ta med objekt eller ikke. Da to nye delproblemer fra dette: ta med objekt, løs problemet for de resterende gjenstandene med redusert kapasitet, eller ikke ta med og løs delproblemet uten redusert kapasitet. Dette kan så implementeres med rekursjon+memoisering eller iterativt med bottom-up. Dette er et NP-hardt problem!!

7. Grådige algoritmer

7.1 Grådighet

Tar det beste valget lokalt, hva som er best akkurat nå.
Løser det mest lovende delproblemet rekursivt og bygger løsningen på denne delløsningen.
Vil identifisere globalt og lokalt optimeringskriterie, samt hvordan vi konstruerer løsningen.
Grådighetskriteriet: dersom vi gjør et lokalt optimalt valg nå, så hindrer ikke dette oss i å få den globalt optimale løsningen. Grådig valg + optimal delstruktur gir oss optimal løsning.

7.2 Eksempel I: ryggsekk

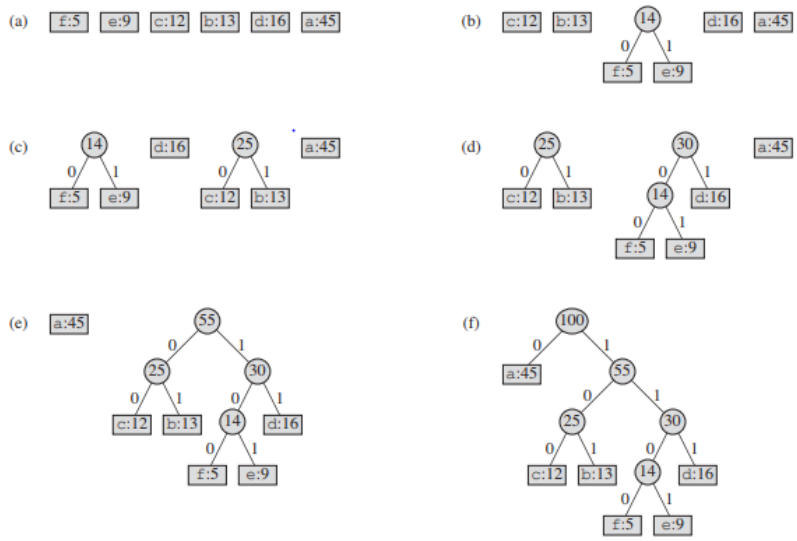
Vi har objekter med verdi v og vekt w, samt en total kapasitet W. Ønsker å finne indekser for objekter + en brøkdel av et objekt som maksimerer verdi så lenge summen av vektene er mindre eller lik W.
Grådig valg: velg alltid det med høyest kilopris (dette gir ikke optimal delstruktur uten betingelsen om at vi kan ta med en brøkdel av det siste objektet, dette forenkler altså problemet veldig).

7.3 Eksempel II: aktivitetsvalg

Vi har n intervaller, og ønsker å finne flest mulig ikke-overlappende intervaller.
Delproblem: intervaller innenfor et gitt område, valget kan da bli et intervall vi skal ta med. Da får vi to delproblemer vi må løse rekursivt. Men vi må ikke løse alle delproblemene, fordi det alltid vil lønne seg å ta med intervallet som slutter først. Dette er altså et grådig valg der optimal delstruktur holder.

7.4 Eksempel III: Huffmann koder

Vi har et alfabet med frekvenser for hver bokstav, vi vil minimere forventet kodelengde.
Vi kan slå sammen løvnoder ved å la én bit skille mellom dem, grådig valg blir da å slå sammen de sjeldneste nodene, siden den ekstra bit-en da koster minst. Bevis med fortrinn: Betrakt en vilkårlig løsning og transformér den gradvis til en grådig løsning, uten å senke kvaliteten. Den grådige løsningen må da være minst like god som enhver annen.



Figur 1: Huffman kode steg

8. Traversering av grafer

8.1 Grafrepresentasjoner

Vi ser på nabomatriser og nabolister. Generelt sett er det raskere med nabomatriser, men de tar med plass, men det kommer jo så klart an på problemet.

Nabomatriser: boolsk matrise A der kanten (u, v) er i matrisa som [u, v] = 1, ellers 0. Urettet graf → symmetrisk matrise A. Ikke godt egnet til traversering, men raskt å sjekke om en kant eksisterer eller ikke.

Naboliste: tabell av noder, der hvert element i lista er en lenket liste av alle nabonodene til noden. Ikke godt egnet til raskt oppslag, men godt egnet til traversering.

8.2 Traversering: bredde først

Naboer til noden vi besøker stiller seg bakerst i køen. Gir korteste vei med en FIFO-kø(uten vekter).

8.3 Traversering: dybde først

Flood-fill: fyll rekursivt nord, øst, sør, vest. Samme prinsipp som BFS, men med LIFO-kø. Nå blir kallstakken i praksis huskelista vår.

8.4 Kantklassifisering

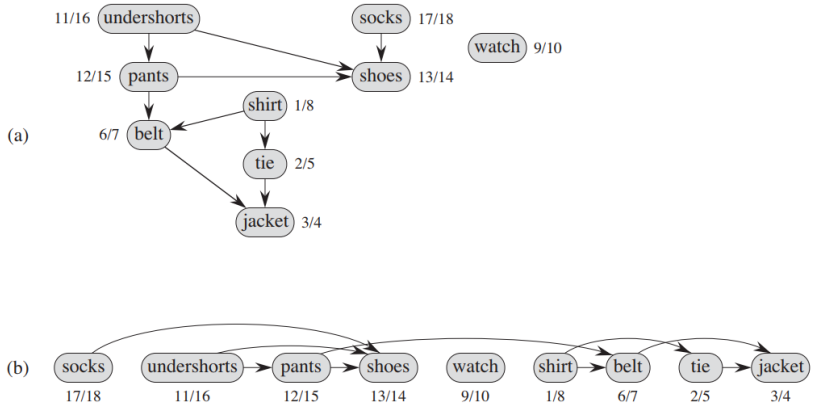
Tre-kanter: kanter i dybde-først-skogen (kanter til hvit node)
Bakoverkanter: kanter til en forgjenger i DF-skogen (kanter til grå node)
Foroverkanter: kanter utenfor DF-skogen til en etterkommer (kanter til svart node)
Krysskant: andre kanter (svart node)

Et DFS har en parentesstruktur.
Parentesteoremet: i et DFS for (u, v) har vi at intervalene til u og v er disjunkte og hverken u eller v er etterkommere av den andre i dybde-først-skogen, eller at intervallet til u er inne i intervallet til v og u da er en etterkommer av v i dybde-først-treet eller motsatt, altså at intervallet til v er inne i intervallet til u og v da er en etterkommer av u i dybde-først-treet. Dette er egentlig bare en logisk konsekvens av hvordan DFS funker, altså som en greie med parentesstruktur.

Kjøretid for DFS: $\Theta(V + E)$, kjøretid for BFS: $\Omega(V + E)$.

8.5 Topologisk sortering

Gir nodene en rekkefølge, foreldre før barn. Hvis vi utforsker u → v så vil v alltid være ferdig før u. Dette gjør finish time en nyttig egenskap for topologisk sortering. Altså er synkende finish time trygt å sortere på. Krever en DAG, altså en rettet, asyklisk graf. I DP med memoisering gjør vi basically det samme, når et kall allerede er regnet ut er det en svart node. Algoritmen her blir egentlig bare å DFSe G, og så sette inn v i en lenka lista etterhvert som hver node v er ferdig. Da er lista sortert på finish times. Vi gjør en topologisk sortering i $\Theta(V + E)$.



Figur 2: Topologisk sortering

9. Minimale spenntrær

9.1 Disjunkte mengder

Mengder representeres som trær, rota peker på seg selv.
Rang: største avstand til løvnode
Make-set(x) lager en mengde med bare x. Union(x, y) setter sammen mengdene med x og y med union. Dette gjøres enkelt ved å sette rota av den ene mengden under rota på den andre, som da har størst rang. Find-set(x) returnerer en peker til mengden som x er i.
Finner da rotnoden rekursivt opp treet til den finner noden som peker på seg selv. Samtidig dytter den nodene den kommer til opp så de peker direkte på rota, funksjonen flater altså ut treet.
M operasjoner: $O(ma(n))$, der $a(n)$ basically er ¡= 4, vokser mye tregere enn log.

9.2 Generisk om minimale spenntrær

Spenngraf: delgraf som inneholder alle nodene.
Spennskog: spenngraf som er asyklisk.
Spenntre: sammenhengende spennskog.
Vi ønsker å lage et spenntre som minimerer vektsummen til kantene i den opprinnelige grafen, altså et minimalt spenntre. Input: en urettet graf $G = (V, E)$ og en vektfunksjon w, output: en asyklisk delmengde som kobler sammen nodene i V og minimeer vektsummen av w. Vi utvider

en kantmengde (partiell løsning) gradvis.
Invariant: kantmangden utgjør en del av et minimalt spenntre.
Trygg kant: kant som bevarer invarianten.

Generisk løsning:
`så lenge A ikke er et spenntre: finn kant som er trygg og legg til i A, returner A.`

Kan bruke snitt til å finne overgangen fra en mengde til en annen. Da går det an å finne trygge kanter over snittet hvis alle andre kanter over snittet har høyere vekt.

Kruskal: en kant med minimal vekt blant de gjenværende er trygg så lenge den ikke danner sykler.

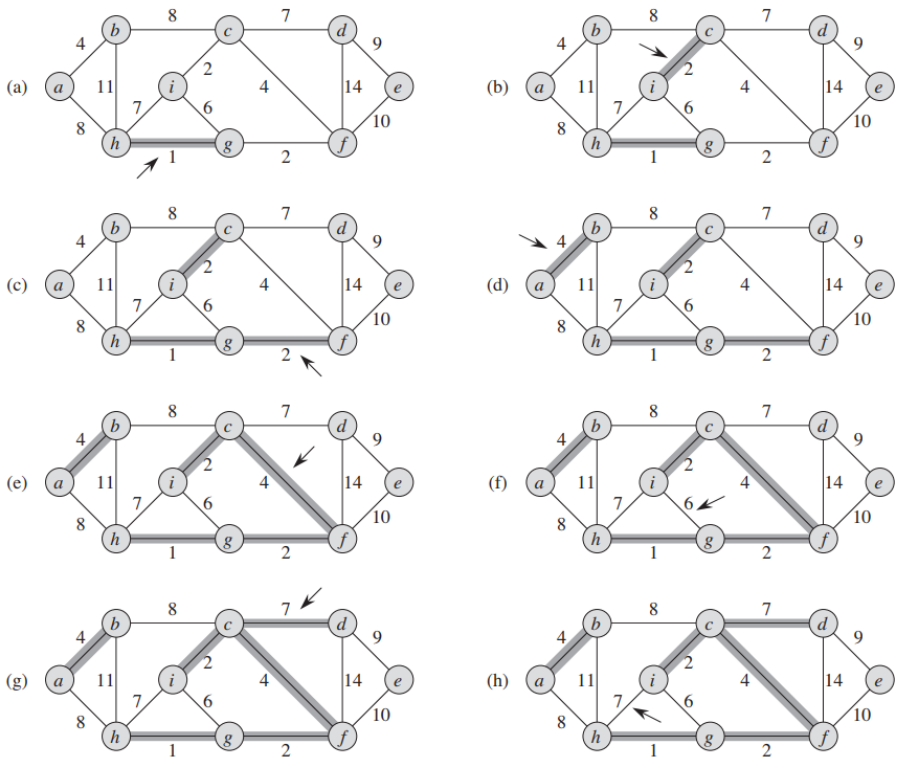
Prim: bygger tre gradvis, en lett kant over snittet rundt treet er alltid trygg.

9.3 Kruskalboi

En skog er fragmenter av et MST, den andre skogen er disjoint-set forest.

`Lag V individuelle partielle spenntrær, sorter kantene etter vekt, for hver kant (u,v): legg til kant i A og slå sammen u og v dersom det ikke er en sti mellom u og v, return A.`

Make-set: $O(1)$, sortering: $O(E)$, find-set: $O()$, union: $O()$, totalt: $O(E)$.



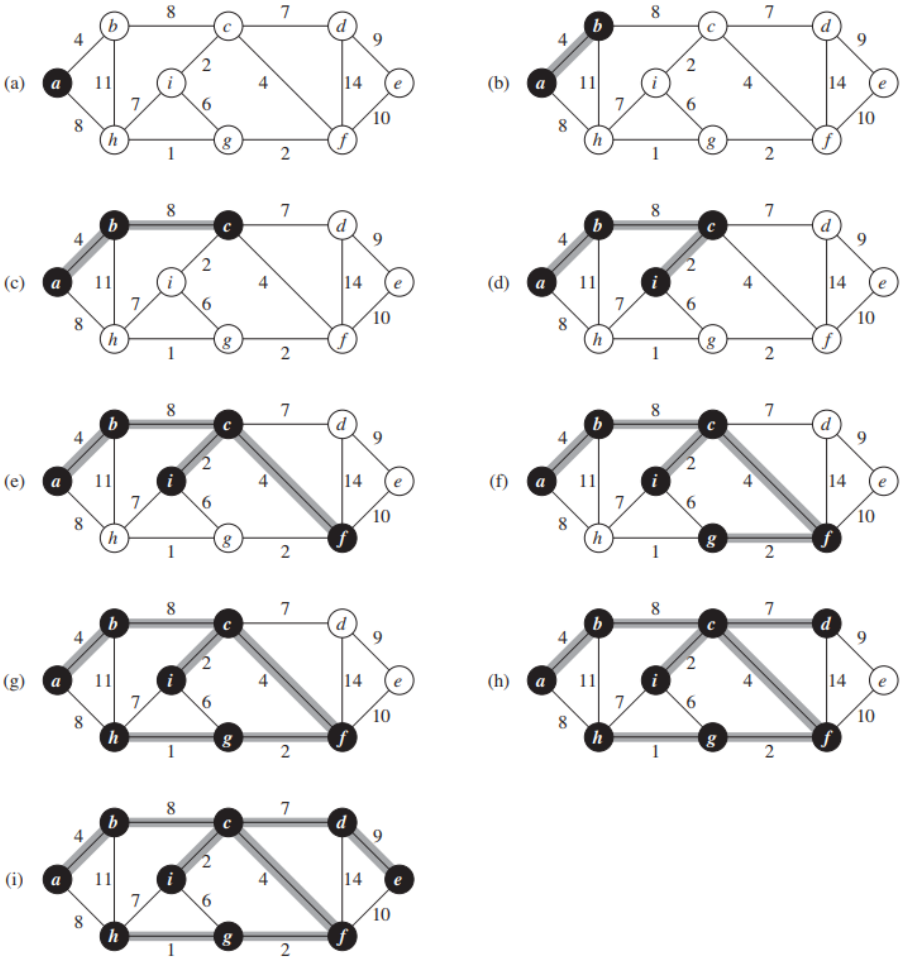
Figur 3: Kruskalboi

9.4 Primboi

Implementeres vha. traversering med min-prioritets-kø(i stedet for LIFO eller FIFO). Prioriteten er vekten på den letteste kanten mellom noden til treet. Vi starter med å legge alle noder inn i køen fra starten med uendelig dårlig prioritet (kan også traversere, blir helt ekvivalent).

```
Sett alle noder inn i køen med uendelig dårlig prioritet, startnoden har prioritet
0. Så lenge køen ikke er tom: ta ut node fra min-prioritets-køen, for hver kant
til noden: dersom den ikke er brukt enda og har lavere vekt enn prioriteten til
noden kanten går til, legg stien.
```

$O(E)$ med binærhaug.



Figur 4: Primboi in effect

10. Korteste vei fra én til alle

10.1 Dekomponering

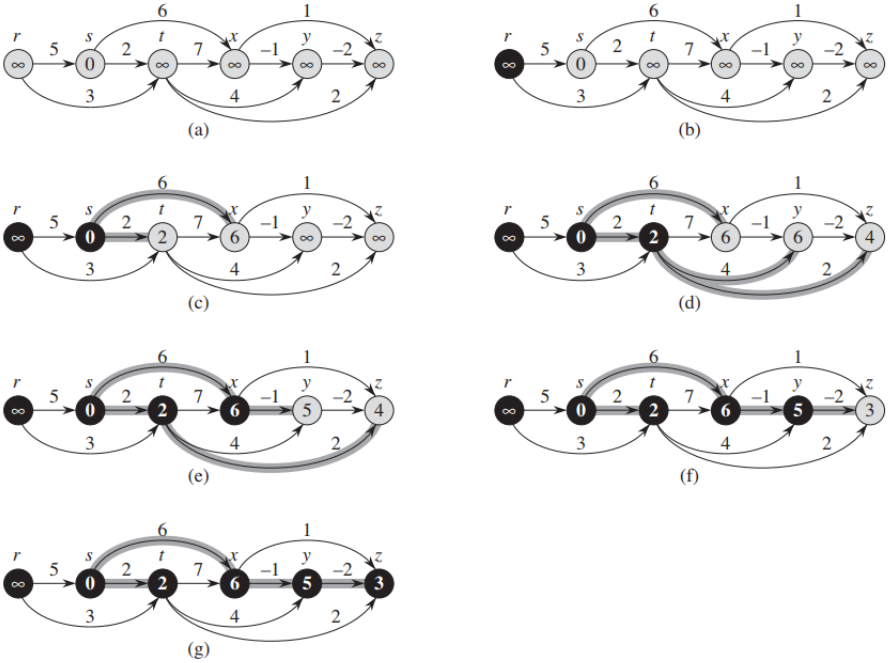
Input: rettet graf $G = (V, E)$ med vektfunksjon w og startnode s.
Output: sti p for hver node v fra s til v som gir minimal vektsum.
Merk at det er lett å gå fra én til alle til alle til én, og vi har ikke noe bedre enn én til alle for å finne én til én.
Dekomponering: funksjon som sjekker korteste vei mellom inputs, antar at delstiene allerede er regnet ut. Hvis s.d = 0 (grunntilfellet) og G er sortert topologisk får vi optimal delstruktur.

10.2 DAG-shortest-path

`Topologisk sorter G.`
`Initialiser noder ved å sette` $v.d = \infty$, $v.\pi = nil$ `for alle noder, husk å sette` $s.d = 0$ `etterpå.`

`For hver fra-node for hver til-node: sett sti fra fra-node til til-node dersom det er bedre enn det vi har. Når vi er ferdig med en node blir den satt til svart, dette skaper problemer med sykler, da må vi kjøre prosedyren flere ganger.` Vi kaller denne siste prosessen for slakking.

Kjøretid: $\Theta(V + E)$



Figur 5: DAG style

10.3 Kantslakking

Oppspalting av minimums-operasjonen fra dekomponeringen. Starter med uendelig estimat på avstanden d, går trinnvis ned (slakker") lengden til vi ender opp med minimal avstand.

Sti-slakkings-egenskapen: om p er en kortest vei fra s til v og vi slakker kantene til p i rekkefølge, så vil v få riktig avstandsestimat. Dette gjelder uavhengig av om andre slakkinger forekommer imens.

Enkel sti: sti uten sykler

La oss ta noen kjempespennende egenskaper før vi går videre:
Triangle inequality: for alle kanter (u, v) har vi at $\delta(s, v) \leq \delta(s, u) + w(u, v)$
Upper-bound property: Vi har alltid $v.d \geq \delta(s, v)$, og når $v.d = \delta(s, v)$ endrer den seg aldri.
No-path property: Hvis det ikke er en sti mellom s og v, så er alltid $v.d = \delta(s, v) = \infty$.
Convergence property: Hvis s-u-v er en raskerste vei for en (u, v), og $u.d = \delta(s, u)$ ved et tidspunkt før (u, v) er slakket, så er $v.d = `delta(s, v)$ for alle tidspunkt etter.
Path-relaxation property: Hvis p er en raskeste vei og vi slakker kantene til p i rekkefølgen vi besøker de fra s til v, så er $v.d = \delta(s, v)$.
Predecessor-subgraph property: når $v.d = \delta(s, v)$ for alle v, er forgjengergrafen et raskeste vei tre med rot i s.

Vi begrenser til at kortest sti er alltid enkel, da positiv sykel kan droppes og negativ sti kan alltid legge til en sykel til. Men den kan ikke finnes effektivt (NP-hardt).
Vi vil finne en smart rekkefølge for å slakke kantene. Naiv slakking: slakk alle kanter i korteste vei k-1 ganger. Da garanterer vi korteste vei. Da får vi Bellman-Ford:

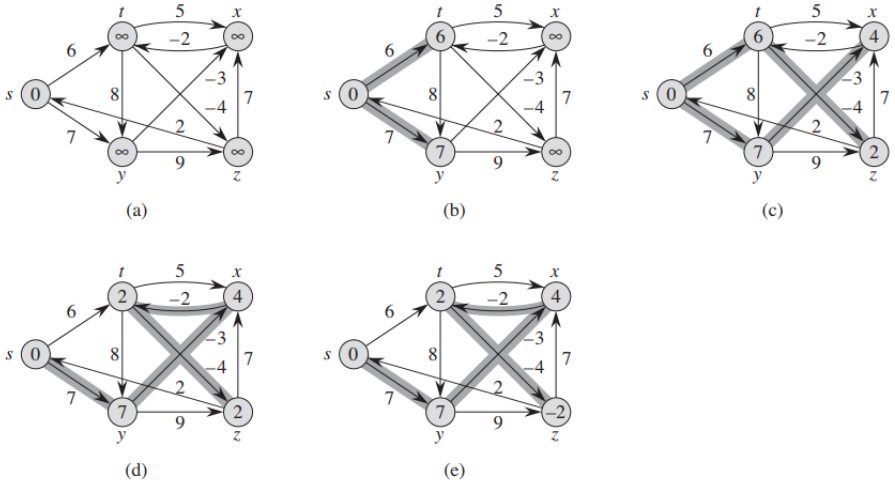
10.4 Bellman-Ford

Slakk alle kanter til det bare må bli rett.

```
Initialiser nodene. Slakk alle kanter k-1 ganger. Slakk en gang til for å sjekke
at vi ikke har negative sykler.
```

Dette gir kjøretid på O(VE).

Om et estimat endres, så var tidligere slakking bortkastet. Altså bare slake kanter fra v når v.d ikke kan forbedres for å effektivisere.
Strategi: velg den gjenværende med lavest estimat, funker siden det laveste estimatet kan ikke forbedres når vi ikke har negative kanter. Da får vi Dijkstra:



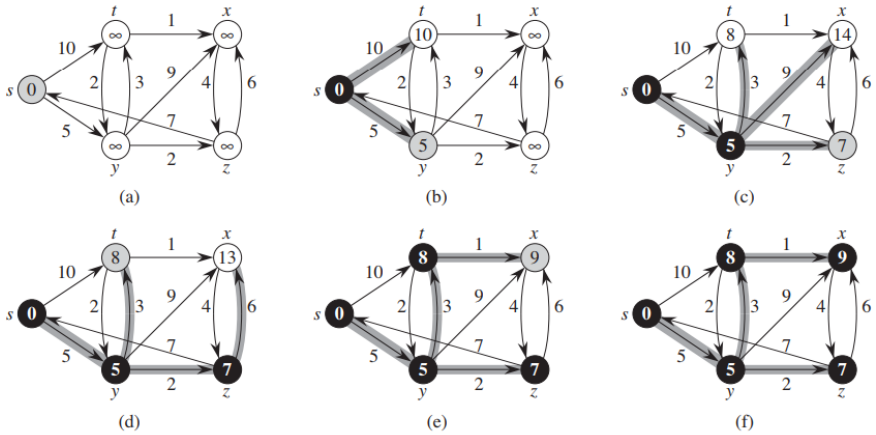
Figur 6: BF in effect

10.5 Dijkstraboi

Prioritetskø som i Prim.

`Initialiser noder og prioritetskø Q. Så lenge Q ikke er tom: hent node med korteste estimat, for hver ut-nabo: slakk ut-kanten.`

Nå slakker vi bare fra hver kant én gang, men på bekostning av at negative kanter er no go. Gir kjøretid $O(V + E)$.



Figur 7: Dijkstraboi in full effect

11. Korteste vei fra alle til alle

11.1 Johnsons algoritme

Input: vektet, rettet graf (uten negative sykler) $G = (V, E)$, med n noder og vekter w.
Output: $n \times n$ matrise D med avstander fra alle noder til alle noder.

For spinkle grafer er det helt ok å kjøre Djikstra for hver node. Men hva med negative kanter? Fast økning av hver kant for å gjøre alle negative kanter positive funker ikke, siden lange stier og korte stier ikke blir vekta like mye.
I stedet får hver node en verdi h, og kantvekten økes med differansen mellom verdiene til de to nodene til kanten. Alle nodene utenom første og siste går da mot hverandre. Ved så å trekke fra verdiene til første og siste node får vi stilengden.
For å garantere positive kanter kan vi velge $h(v)$ som korteste vei fra en startnode s til v. Men da må vi sikre at vi når alle nodene fra s. Vi gjør dette ved å legge til en ny node s i grafen, som går til alle de andre nodene.

Johnson:
`Lag G' med startnode s og kjør Bellman-Ford på G'. Så setter vi distansen fra s til v som h(v) for alle v i G. Så setter vi ny vekt som gammel vekt pluss differansen mellom nodene for hver e. Så gjør vi djikstra på hver node u og setter inn avstanden fra u til v i D for alle noder u og v.`

11.2 Transitiv lukning

Får inn rettet graf G, skal returnere graf G' der kantene bare eksisterer hvis stien i G eksisterer.
Korteste stier har felles delstier –¿ DP?
Dekomponering: finnes det en vei fra i til j via nodene $1, 2, ..., k$? Kaller dette delproblemet t_{ij}^{k}.
Kan deles opp i $t_{ij}^{(k-1)}$, $t_{ik}^{(k-1)}$ og $t_{kj}^{(k-1)}$. Grunntilfellet blir da bare om dette er en kant eller ikke (k=0).
Koden under ser ganske stygg ut, men den første delen er bare å initialisere $T^(0)$, deretter går vi igjennom T k ganger, og sjekker for hvert element om stien mellom i og j eksisterer ved å sjekke om den eksisterer fra før eller om den går innom noden k. Badabim badabom. Dette gir $\Theta(n^3)$ uff.

```
TRANSITIVE-CLOSURE(G)
 1  n = |G.V|
 2  let T^(0) = (t_ij^(0)) be a new n × n matrix
 3  for i = 1 to n
 4      for j = 1 to n
 5          if i == j or (i, j) ∈ G.E
 6              t_ij^(0) = 1
 7          else t_ij^(0) = 0
 8  for k = 1 to n
 9      let T^(k) = (t_ij^(k)) be a new n × n matrix
10      for i = 1 to n
11          for j = 1 to n
12              t_ij^(k) = t_ij^(k−1) ∨ (t_ik^(k−1) ∧ t_kj^(k−1))
13  return T^(n)
```

11.3 Floyd-Warshall

Vi ønsker lavere kjøretid enn Bellman-Ford og vi vil tillate negative kanter...
Dekomponering: korteste vei fra i til j innom $1, 2, ..., k$. Samme som i sted, bare at vi ser på forgjengere π og avstander d, ikke t. Siden det er alle til alle har hver node nå V forgjengere, ikke én. Da blir $d_{ij}^k = \min(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)})$ eller w_{ij} i grunntilfellet(k=0). Da blir forgjengeren $\pi_{ij}^k = \pi_{ij}^{(k-1)}$ hvis $d_{ij}^{(k-1)}$ er mindre, ellers så blir forgjengeren $\pi_{kj}^{(k-1)}$.

Når D matrisa og matrisa er generert kan vi bruke matrisa til å finne en vilkårlig optimal sti fra i til j ved bare å se på hva hver forgjenger er rekursivt fra π_{ij} til vi ender opp der vi starta i π_{ii}. Easy.
Trippel løkke $\rightarrow \Theta(n^3)$.

```
FLOYD-WARSHALL'(W)
1  n = W.rows
2  initialize D and Π
3  for k = 1 to n
4      for i = 1 to n
5          for j = 1 to n
6              if d_ij > d_ik + d_kj
7                  d_ij = d_ik + d_kj
8                  π_ij = π_kj
9  return D, Π
```

12. Maksimal flyt

12.1 el problemo

Flytnett: rettet graf $G = (V, E)$, med kapasitet $c(u, v) \geq 0$, kildenode s og sluknode t.

Antar at alle noder i V er på en sti fra s til t, og at vi ikke har noen løkker (kan godt ha sykler, løkker er kanter (u, u), altså som går til seg selv), samt at vi ikke tillater antiparallelle kanter(altså at dersom (u, v) er i E er ikke (v, u) i E). Kanter som ikke finnes i E har kapasitet lik 0.

Flyt: funksjon f fra alle nodepar til $\mathbb{R}$, der $0 \leq f(u, v) \leq c(u, v)$.
Antar at flyt inn i node er flyt ut av node (Kirchoff 1.)

Flytverdi: $|f| = \sum f(s, v) - \sum f(v, s)$
Merk at $|f|$ er notasjon for flytverdien av f, ikke absoluttverdi.

Input: flytnett G
Output: flyt f for G med maks flytverdi $|f|$

Dersom vi har antiparallelle kanter kan vi splitte den ene kanten med en node. Da bevarer vi flyten i nettverket og fjerner alle antiparallelle kanter.
Dersom vi har flere kilder og sluk er det bare å legge til en super-kilde og et super-sluk som dekker alle kilder og sluk.

Restnett: rettet graf med fremoverkanter ved ledig kapasitet, bakoverkant ved flyt. Merk at vi ikke trenger å lage en ny graf som representerer restnettet i en implementasjon, vi kan hente informasjonen vi trenger rett fra flytnettet.

Forøkende sti: sti fra kilde s til sluk t i restnettet. Langs fremoverkant $\rightarrow$ flyten kan økes, langs bakoverkant $\rightarrow$ flyten kan omdirigeres .

Flytoppheving: sende flyt baklengs langs kanter der det allerede går flyt, flyt langs bakoverkant i restnett representerer dette.

$_f(u, v) = c(u, v) - f(u, v)$ dersom $u, v)$ er i E, $f(v, u)$ dersom (v, u) er i E, 0 ellers.

Grunnen til at vi setter $c_f(u, v)$ lik flyten i (v, u) dersom (v, u) er i grafen, er at det sier hvor mye vi maksimalt kan redusere flyten bakover, som er flyten som er der nå.

12.2 Ford-Fulkerson

Finn forøkende stier så lenge det går. Når vi ikke lenger kommer fram til t er det ingen forøkende stier igjen, og vi har maksimal flyt (om vi bruker BFS kaller vi det Edmonds-Karp).

Høynivå implementasjon:

Finn forøkende sti først, finn så flaskehalsen i stien, altså laveste restkapasitet. Oppdater flyt langs stien med denne verdien. Repeat...

Litt mer detaljert:

```
FORD-FULKERSON(G, s, t)
1  for each edge (u, v) ∈ G.E
2        (u, v).f = 0
3  while there is a path p from s to t in G_f
4        c_f(p) = min {c_f(u, v) : (u, v) is in p}
5        for each edge (u, v) in p
6              if (u, v) ∈ E
7                    (u, v).f = (u, v).f + c_f(p)
8              else (v, u).f = (v, u).f − c_f(p)
```

Alternativ: flett inn BFS. Vi finner flaskehalser underveis. Holder styr på hvor mye flyt vi får frem til hver node. Traverserer bare noder vi ikke enda har nådd.
Mulig økning: v.a, gitt av flaskehalsen vi har funnet som er på stien mellom s og v.

12.3 Edmundson-Karp

Kan beskrives som følger: så lenge det er mulig å øke flyten: initialiser noder og kø Q med s i for BFS. Gjør så BFS til vi finner t eller vi er tom for noder å besøke, sjekker potensielle nabonoder mens vi søker for å finne noen noder med positiv restkapasitet. Dersom vi gjør det legger vi til denne flyten. Prøver så å finne forøkende sti, ellers er vi good.

```
EDMONDS-KARP(G, s, t)
 1  for each edge (u, v) ∈ G.E
 2      (u, v).f = 0
 3  repeat › until t.a == 0
 4      for each vertex u ∈ G.V
 5          u.a = 0 › reaching u in G_f
 6          u.π = NIL
 7      s.a = ∞
 8      Q = ∅
 9      ENQUEUE(Q, s)

10      while t.a == 0 and Q ≠ ∅
11          u = DEQUEUE(Q)
12          for all edges (u, v), (v, u) ∈ G.E
13              if (u, v) ∈ G.E
14                  c_f(u, v) = c(u, v) − (u, v).f
15              else c_f(u, v) = (v, u).f
16              if c_f(u, v) > 0 and v.a == 0
17                  v.a = min(u.a, c_f(u, v))
18                  v.π = u
19                  ENQUEUE(Q, v)

20      u, v = t.π, t › at this point, t.a = c_f(p)
21      while u ≠ NIL
22          if (u, v) ∈ G.E
23              (u, v).f = (u, v).f + t.a
24          else (v, u).f = (v, u).f − t.a
25          u, v = u.π, u
26  until t.a == 0
```

Finne forøkende sti tar $O(E)$, og vi gjør $O(|f*|)$ (maks flytverdi) antall slike operasjoner, altså $O(E|f*|)$ kjøretid - uten BFS.
Med BFS blir nå antall operasjoner $O(VE)$, slik at kjøretiden er $O(VE^2)$!

12.4 Minimalt snitt

Dette er dualen til maks flyt, finne den partisjonen S, T av V med minimal kapasitet. Maks flyt = min snitt! Merk at flyten over et snitt er summen av flyten over den ene veien minus summen av flyten andre veien, men kapasiteten til et snitt går bare over snittet, altså fra s-siden til t-siden. Og et minimalt snitt er da snittet med minimal kapasitet.

12.5 Matching

En matching M er en delmengde av en urettet graf E der ingen av kantene i M deler noder.
Bipartitt matching: M matcher maksimalt fra en mengde til en annen.

Kan løses med maks flyt ved å legge til kilde og sluk i E, med kapasitet lik 1 på hver kant.
Heltallsteoremet: For heltallskapasiteter gir Ford-Fulkerson heltallsflyt.

13. NP-kompletthet

13.1 NP

Et problem er en relasjon mellom input og output: $X \to Y$
Et konkret problem har input og output som bitstrenger: 0110000101 $\to$ 11010010001
Optimeringsproblem vs. beslutningsproblem. Vi reduserer fra optimeringsproblemet til et beslutningsproblem.
Konkrete beslutningsproblemer tilsvarer formelle språk (mengder av strenger).
En algoritme aksepterer x dersom $A(x) = 1$, avviser om $A(x) = 0$.
Den avgjør et språk L dersom x i $L \to A(x) = 1$, x ikke i $L \to A(x) = 0$.

En verifikasjonsalgoritme sjekker om løsningen er gyldig, tar inn sertifikat y og instans x. Sertifikat er en streng y som bevis for et ja-svar på beslutningsproblem. A verifiserer x hvis det eksisterer et sertifikat y slik at $A(x, y) = 1$. Denne definisjonen er asymmetrisk; det finnes ikke motbevis, bare en vei.

Kompleksitetsklasse: en mengde språk
P: språkene som kan avgjøres i polynomisk tid
Cobhams tese: det er disse problemene vi kan løse i praksis
NP: språkene som kan verifiseres i polynomisk tid (nondeterministic polynomial)
Co-NP: språkene som kan falsifiseres i polynomisk tid

HAM-CYCLE: språket for Hamilton-sykel-problemet, lett å verifisere i polynomisk tid (ikke nødvendigvis lett å falsifisere). Hamilton-sykel-problemet går ut på å avgjøre om det finnes en sti gjennom en graf som besøker hver node akkurat én gang.

P vs. NP: om vi kan løse problemet, så kan vi verifisere det med samme algoritme, og bare ignorere sertifikatet, altså er P i NP og co-NP. Men vi vet ikke om snittet mellom co-NP og NP!

13.2 Reduksjoner

Reduksjoner: la oss si vi har to problemer A og B. Hvis vi kan løse A ved å løse B kan ikke A være vanskeligere enn B.

Vi vil redusere fra Q til Q', fra språk L1 til L2. Reduksjonen har input bitstreng x og output bitstreng $f(x)$, der x er i L1 og $f(x)$ er i L2. Om vi kan avgjøre at $f(x)$ er i L2 kan vi avgjøre at x er i L1 (ikke motsatt!) Så hvis vi vil løse x kan vi redusere til $f(x)$, løse om den er i L2 og slik vite at x er i L1. Dette er samme scenario som A og B over. Så hvis løsningen til $f(x)/B$

er polynomisk er også løsningen til x/A polynomisk.

Redusibilitet: hvis A kan reduseres til B i polynomisk tid skriver vi $A \leq_p B$. Denne relasjonen utgjør en preordning. Så vi kan vise at B er vanskelig ved å redusere fra et vanskelig problem A, altså etablere $A \leq_p B$.

13.3 NPC

Kompletthet: et problem C er komplett dersom alle problemene X kan løses ved å løse C. Da er alle X like vanskelig eller ikke like vanskelig som C, X kan ikke være vanskeligere!
Et element er maksimalt dersom alle andre problemer i klassen kan reduseres til Q. Et problem er komplett for en gitt klasse og en gitt type reduksjoner dersom det er maksimalt for redusibilitetsrelasjonen.

NP-kompletthet: viser at L er NPC ved å vise at L er i NP, og ved å redusere problemet til et kjent NP-komplett språk L' vha. en reduksjonsalgoritme f som mapper instanser av L' til instanser av L og viser at for alle x i L' er det ekvivalent med at $f(x)$ er i L og at f har polynomisk kjøretid.

NP-hard: problemer som er minst like vanskelig som NP. Alt i NP kan reduseres til alt i NPH.

NPC er snittet av NP og NP-hard (P er i NP).

Dermed kan alle problemene i NPC reduseres til hverandre. Problemer i P kan reduseres til alt i NP. Hvis vi finner en reduksjon fra NP til P er NP=P. NP reduseres til NPC -¿ problemet er NPC.

Noen NP-komplette problemer: CIRCUIT-SAT, SAT, 3-CNF-SAT, SUBSET-SUM, CLIQUE, VERTEX-COVER, HAM-CYCLE, TSP