Algdat sammendrag

Kapittel 1 Kompleksitet Analyse

Hva vi beregner

Med kompleksitetsanalyse prøver vi ikke å forutsi kjøretid i sekunder direkte. Det er ikke bare unødig komplisert, men også forskjellig for ulike maskiner. I stedet prøver vi å finne sammenhenger av typen “kjøretid T for en algoritme er proporsjonal med f(n), der n er størrelsen på datamengden”.

Enkle operasjoner

I stedet for å telle sekunder, ser vi på algoritmen og teller opp hvor mange enkle operasjoner den utfører for en datamengde av en gitt størrelse. For vår maskin uavhengige analyse antar vi at alle slike enkel operasjoner tar like lang tid. Virkeligheten er mer komplisert, men dette gir en god tilnærming. Vi teller rett og slett opp hvor mange enkle operasjoner programmet vårt gjør.

Algo 1.2

```
Int algo(int t[], int n){
        Int sum = 0;
        for(int i = 0; i< n; i++) sum += t[i];
        Return sum;
}
```

Algo 1.2 Inneholder 3 operasjoner som alltid utføres en gang, på linje 3,4 og 7 Løkka på linjene 4-6 inneholder 3 operasjoner som utføres n ganger. Dermed blir kjøretiden T(n) = 3n+3.
Når vi under opptelling støter på et metodekall, legger vi til antall operasjoner som utføres i metoden.

Kompleksitet

Når vi beregner kjøretid ser vi først og fremst på kompleksiteten for kjøretid funksjonen. Reel kjøretid for en algoritme kan f.eks være $T(n) = 2n^3 + 8n^2 - 5n$. Dette runder opp til n^3:

Asymptotisk notasjon

Når vi forenkler kjøretid funksjonen for å se på kompleksiteten, for eksempel ($4n^2 + 2n \Rightarrow n^2$), sier vi at vi studerer den asymptotiske effektiviteten. Den algoritmen som har best asymptotisk effektivitet er best for store datamengder.

O - en øvre grense for kjøretid

O-notasjon brukes for å angi øvre grense for kjøretid. Kjøretiden kan altså aldri bli verre enn det som angis med O-notasjon.

Definisjon

En funksjon f(n) tilhører mengden O(g(n)) hvis det eksisterer positive konstanter c og n0 slik at cg(n) er større enn eller lik f(n) for alle n>= n0. Hvis dette er tilfelle, skriver vi f(n) E O(g(n)), og sier g(n) er en asymptitusj øvre grense for f(n).

- en nedre grense for kjøretid

-notasjon brukes for å angi beste mulig kjøretid. Den sier noe om hvor raskt en algoritme kan utføres, hvis vi er maksimalt heldige med inndata. Bedre blir det garantert ikke!

Definisjon

En funksjon f(n) tilhører mengden (g(n)) hvis det eksisterer positive konstanter c og n0 slik at cg(n) er mindre enn eller lik f(n) for alle n>=n0. Hvis dette er tilfelle, skriver vi f(n) $\in$ (g(n)), og sier at g(n) er en asymptotisk nedre grense for f(n).

Θ - en øvre og nedre grense

Θ-notasjon brukes for å angi både øvre og nedre grense for kjøretid. Kjøretiden er alltid proporsjonal med g(n), både i de beste og verste tilfeller vi kan tenke oss.

Definisjon

En funksjon f(n) tilhører mengden Θ(g(n)) hvis det eksisterer positive konstanter c1, c2 og n0, slik at c1g(n) er mindre enn eller lik f(n), og c2g(n) er større enn eller lik f(n) for alle n>=n0. Hvis dette er tilfelle skriver vi f(n) $\in$ Θ(g(n)), og sier at g(n) begrenser f(n) asymptotisk.

Kapittel 2 Rekursjon

Hva er rekursjon?

Definisjon: En algoritme som kaller seg selv, kalles en rekursiv algoritme.
Noen ganger har vi indirekte rekursjon. Det betyr at en algoritme kaller en annen algoritme og den andre kaller den første igjen. Eventuelt kan vi ha flere algoritmer mellom den første og den som kaller den. Vi har også begrepet rekursiv datastruktur. Det er en datatype der variablene inneholder en peker eller referanse til en annen variabel av samme type lenket liste og tre er eksempler på dette.

Rekursjon og iterasjon

Alle algoritmer som bare har et rekursivt kall i algoritmen , og at vi bare minker med 1 for hvert rekursivt kall nivå. Alle slike algoritmer kan og bør omformes på denne måten. Alle metodekall, både rekursive og vanlige, fører til a argument verdier, lokale variabler og andre ting må gjemmes bort i maskinens minne mens kallet foregår. Har vi mange kall inni hverandre, slik som vi ofte får når vi bruker rekursjon, vil det kunne føre til at mye plass blir brukt. I verste fall bruker vi opp all plassen, slik at program utføringen blir avbrutt. Dessuten tar det noe tid å gjemme bort og hente fram igjen disse dataene. Derfor er det lurt å ikke bruke rekursjon i utrengsmål. Er det enkelt å lage algoritmer ikke-rekursive, så gjør vi det, men er det mye enklere lage den rekursiv så gjør vi det.

Induksjon, rekursjon og rekursive definisjoner

Rekursjon er nært knyttet til begrepet induksjon. Induksjonsbevis kan ofte brukes til å bevise at en rekursiv algoritme er riktig og at den har en bestemt tid kompleksitet. Ideen bak induksjonsbevis er følgende Vi ønsker å bevise at noe gjelder uansett hvor stor n vi har, og starter med vise det for et enkelt tilfelle basis. Deretter antar vi at vi har vist der for en bestemt

verdi, n-1 og beviser at da må det gjelde for n også. Dette er induksjonstrinnet. Og når induksjonstrinnet gjelder for n, må det gjelde for n+1 også, og generelt for alle heltall. Rekursjon induksjon er to sider av samme sak. Begge deler krever at vi kan løse et lett tilfellet, basis og deretter antar vi at vi har løst problemet for n-1. Så gjenstår bare å bruke løsningen for n-1 til å finne løsningen for n. Forskjellen er at i rekursjon starter vi på en bestemt n, jobber oss ned til basis og opp til n igjen mens i induksjon starter vi på basis og går opp til uendelig.

Analyse av lineær rekursjon

La T(n) være tiden vi bruker på en algoritme når vi har datamengden n. Da il tiden vi bruker med datamengden n-1, være T(n-1). Da kan vi lage et rekursivt uttrykk for tidsforbruket. For fakultets algoritmen og andre lineære algoritmer får vi når vi lar tidsforbruket av ting som er O(1) være 1 da kan vi lage en tabells om viser tidsforbruket for ulike verdier av n.

Bevis: Basis er OK siden T(1) = 1. For induksjonstrinnet har vi at vi skal at påstanden gjelder for n-1, dvs at T(n-1) = n -1. Da kan vi bruke likningen for å finne at T(n) = T(n-1) = (n-1)+1 = n. Noe som stemmer med påstanden. Dermed er påstanden bevist.

Split og hersk!

Splitt og hersk-teknikk går ut på å splitte opp problemet i under problem, løse disse, og sette sammen disse løsningen til en løsning for det opprinnelige problemet. Veldig ofte består splittingen av å dele problemet på midten, og få ett eller to nye problemer som er bare halvparten så store som det opprinnelig problemet. I andre tilfeller får vi ikke redusert problem størrelsen så mye i hvert nivå, kanskje bare fra n til n-1. Dette er det samme som i lineær regresjon, men til forskjell fra det lineære tilfellet, kan vi ha mer enn et rekursivt kall i hvert nivå, slik at program utføringen blir mye mer komplisert.

En generell metode for å finne tidskompleksiteten (Master metoden).

I mange tilfeller får vi et uttrykk som kan beskrives med følgende likning T(n) = aT(n/b) + cn^k der a >= 1, b>1, c og k>=0 er konstanter. Forholdet med tids kompleksitet i Θ-notasjon blir:

b^k < a => T(n) E Θ(n^logb a)

b^k = a => T(n) E Θ(n^k log n)

b^k > a => T(n) E Θ(n^K)

Prøve og feile- algoritmer

Noen ganger vet vi ikke på forhånd nøyaktig hvordan vi skal finne den riktige løsningen, men må prøve oss fram og kanskje gå et skritt eller flere tilbake. Og prøve nye veier hvis det viser seg at det vi prøve rikke fører fram. Slike problemer er godt egent for rekursjon. HVert skritt blir ett nivå i rekruksjons og vi går oppover og nedover i refusjons nivåene inntil vi finner en løsning eller alle løsningen er prøvd. Har vi fortsatt ikke funnet noen løsning når alle mulige er prøvd, kan vi fastslå at det ikke finnes noen løsning. Som regel vil denne løsningsmetode føre til mye arbeid for datamaskinen fordi det er mange muligheter som må prøves. Derfor ser man seg ofte om etter måter å kutte vekk noen av mulighetene på.

Kapittel 3

Sorterings Problemet

Det er viktig å ha gode sorteringsalgoritmer for enhver anledning. En sorteringsalgoritmer tar en uordnet mengde tall og plasserer dem i stigende rekkefølge. Alternativt kan de sorteres i synkende rekkefølge, til tider kalt omvendt orden eller baklengs sortering. Vi kaller ikke alt vi sorterer tall. Men for datamaskiner og algoritmer er også tekst en spesiell form for tall basert på ascii, unicode eller liknende. Sortering er et grunnleggende problem og det finnes mange algoritmer. Hvilken som er best avhenger av problemet: hvor stor datamengde vi skal sortere, og i hvilken grad den er sortert allerede. Andre sider ved datamengden, som datatype fordeling av soteringsnøkkler., kan til tider også ha noe å si.

Innsettingstorering

Innsettingssortering er en enkel algoritme for små datamengder den sorterer tallene på samme måte som mange av oss sorterer kortene i kortspill:

- Start med et kort på hånda
- Plukk opp ett og ett kort om gangen
- Sett hvert av kortene inn på rett plass etter hvert som de plukkes opp etter som datamaskinen ikke ser alle kortene samtidig slik vi gjør flytter den tallet en og en plass forbi de ferdig sorterte inntil den finner rett plass.

Analyse

Algoritmen består av to løkker inni hverandre; den totale kjøretiden blir dermed produktet av de kjøretiden for de to. Den ytre økka teller fra 1 til n, og er opplagt Θ(n). Den indre er mer dataavhengig. I beste fall slår ikke betingelsen t[i]> flytt til i det hele tatt slik at den indre løkka ikke blir kjørt. Innsettingssortering er altså **Ω**(n), Dette skjer hvis tabellen er korrekt sortert fra før av. I verste fall slår denne betingelsen til hver eneste gang slik at den indre løkka går fra j-1 til -1 jver gang. Vi får j iterasjon , den totale kjøretiden blit. O(n^2). Dette skjer hvis tabellen er sortert i omvendt orden.

Boblesorting

Boblesortering fungerer ved å gå gjennom tabellen n-1 ganger. Ved hvert gjennomløp sjekkes alle tall som står etter hverandre og bytter plass hvis de står feil innbyrdes. Det største tallet “synker” dermed fram til siste plass i tabellen på første gjennomløp, mens små tall “bobler” en posisjon opp. I neste omgang er det derfor n-1 omganger er alle tall fra t[n-1] til t[1] plassert rett og t[0] er også korrekt ettersom alle “gale” posisjoner er opptatt.

Analyse

Boblesortering har to løkker inni hverandre. I den ytre teller i ned fra n-1 til og er dermed Θ(n). I den indre løkka teller j fra 0 til i. Denne dobbeltløkka er nesten som den i en vanlig dobble løkke, eneste forskjellen er at den ytre løkka teller baklengs, men det har ingenting å si for antall operasjoner. Boblesortering er dermed Θ(n^2).

Velgesortering

Velg Sortering kalles så fordi algoritmen i første omgang velger seg de største tallet og plasserer det rett dvs. På den siste plassen i tabellen. Velgsorting har akkurat den samme dobbeltløkka som bobblesortering. Dermed er også velg sortering Θ(n^2). Velgsortering gjør like

mange sammenlikninger som boblesortering, men færre ombyttinger. Velgsortering bytter om tall maksimalt en gang for hvert gjennomløp av den ytre løkka, for å sette tall nr på rett plass.

Om kvadratisk sorterings algoritmer

Algoritmer vi har sett på hitill, har alle kvadratisk kjøretid, O(n^2). Dette er ikke tilfeldig. Vi skal se litt på hvorfor det blir slik.

Definisjon

En invertering er et par elementer i en tabell som står i feil rekkefølge i forhold til hverandre. Altså t[i] > t[j] mens i < j. I tabellen {4, 1, 3, 2} er det der med fire inverteringer gitt ved følgende tallpar. (4,1), (4,3), (4,2), (3,2). En sortert tabell har ingen inverteringer.

Teorem 3.1 gjennomsnitt iverteringer

En uordnet tabell av størrelse n innehoder i gjennomsnitt n(n-1)/4 inverteringer.

Bevis

For enhver tabell t, kan vi lage en tabell tomv som inneholder de samme tallene i omvendt orden, f.eks t = 1, 3, 2, 4 ⇔ tomv = 4,2,3,1. Hvis vi velger to tall x og y fra y, der y>x vil paret (x,y) være en investering i enten t eller tomv. Det totale antall slike par i t og tomv er n(n-1)/2. En gjennomsnitt tabell, hvor vi antar at tallene er tilfeldig ordnet i, inneholder dermed halvparten så mange investeringer.

Teorem 3.2

Enhver algoritme som sorterer ved å bare bytte om tabellelmenter som står ved siden av hverandre, trenger **Ω**(n^2) operasjoner i gjennomsnitt.

Bevis

Teorem 3,1 forteller oss at gjennomsnitts tabellen har n(n-1)/4 inverteringer å bytte om to tall som står ved siden av hverandre fjerner kun en invertering algoritmer som bare bytter om nabotall, trenger i gjennomsnitt n(n-1)/4 slike ombyttinger.

Både boble og innsettingssortering sorterer ved å bytte om naboer, og er dermed **Ω**(n^2) i gjennomsnitt. Velg Sortering byter elementer som ligger lengre fra hverandre og bytter dermed O(n) ganger. Men antallet sammenligninger får likevel denne algoritmen opp i Θ(n^2) kjøretid.

Shellsort

Shellsort er en sub kvadratisk sorterings algoritmer. Den har altså lavere kompleksitet enn algoritmene vi har sett hittil. Algoritmen sammenligner og bytter nødvendig om tall står langt fra hverandre i tabellen. Shellsort har en ytre løkke hvor variablene går fra n/2 til 1. S bestemmer hvor store steg som brukes i sorteringen innenfor. Mens starter på n/2. I) den omgangen sammenliknes og sortere tall med innbyrdes avstand n/2. Så deles n med 2.2 og vi får en ny sortering av tall som ligger nærmere hverandre. Så lenge s er relativt høyt tall, flyttes tallene med lange , men få hopp. Når s til slutt blir 1 , får vi en standard innsettingssortering. Alle lange flyttene er gjort allerede,m noe som er ideelt for innsettingssortering.

Analyse

Tidkompleskitenten for shell sort avhenger veldig av hvordan s senkes fra n/2 til 1. Shells opprinnelige ide var å halvere s for hver omgang av den ytre løkka. Det gir en merkbar forbedring i forhold til innsettingssortering, men har fortsatt et verste tilfelle som er O(n^2). Hvis vi i stedet halverer s og legger til 1 når den nye s blir et partall, får vi i stedet en kompleksitet på O(n^3/2). I programeksempler bruker vi Gonnets sekvens, som går ut på å dele s med 2,2 i stedet for 2. Kompleksiteten for denne varianten av shell sort er det ingen som har klart å beregne, men målinger på testdata tyder på noe i område O(n^7/6) - O(n^5/4).

Flettesortering

Flettesortering er en splitt og hers-algoritme for sortering. Vi har en tabell med n tall som skal sorteres. Denne deles i to tabeller med n/2 tall hver. Dermed har vi to mindre underproblemer, som kan løses hver for seg rekursiv med fletesortering. PÅ dette viset deler vi opp i mindre og

mindre usorterte tabeller, inntil vi får mini tabeller med bare ett tall hver. En tabell med bare ett element er ferdig sortert.

Analyse fletting

Vi lar n = h-v+1, altså antall tall som skal flettes. Løkka kopierer data inn i hjelpe tabellen. Den for seg når en av deltabellene er tømt og kan dermed ikke repetere mer enn n ganger. Den er altså O(n). I beste fall kopieres bare den ene deltabellen, som ahr størrelse n/2. Denne løkka er dermed Θ(n) Neste løkke kopierer gjenværende tall i venstre del tabell, i verste fall n/2. Den er altså O(n) neste løkke kopierer sorterte data tilbake fra hjelpetabellen, iverste fall n tall. Dn er dermed O(n). Legger vi sammen tidsbruk finner vi at fletting er Θ(n).

Analyse flettesortering

Algoritmen flettesort er rekursiv. Den deler datasettet i to like store deler, og kaller seg selv på hver av dem. Til slutt slås de to sorterte delene sammen med metoden flett, som er Θ(n) da kan vi sette opp T(n) = 2T(n/2)+ Θ(n) som blir til O(nlogn).

Quicksort

Quicksort er den raskeste kjente sorteringsalgoritme for generell data. Den er derfor mye brukt og viktig å forstå. Ettersom det er den mest brukte algoritmen, er det utviklet mange triks for å få den enda raskere, og vi skal på noen av dem her. Quicksort er , akkurat som flettesortering, en splitt og hersk-algoritme. Tabellen som skal sorteres, deles i to deler, hvor alle tall i den laveste delen har lavere verdi enn alle tall i den høyre delen. Tabellen deles ikke nødvendigvis på midten, delingen foregår ved at vi velger et tall fra tabellen som delingstall og flyttermiddre tall mot venstre og større tal mot høyre. Delings Posisjonen blir stedet hvor delingstallet havner. Quick sort har altså en kjøretid på O(nlogn) bevis på side 64 i boken eller https://www.khanacademy.org/computing/computer-science/algorithms/quick-sort/a/analysis-of-quicksort

Metoden median3sort - enkel forbedring av quicksort

Denne metoden er ikke nødvendig for å implementere quicksort, men er tatt med som et eksempel på enkel forbedring. Npr quicksort skal velge et delingstall, gjelder det å dele tabellen jevnest mulig men uten å bruke for lang tid på å finne et passende delingstall. Hvis vi antar at vi

får til en jeven deling, blir kjøretiden for quicksort O(nlogn) hvis tabelldelingen er O(n). Det kan lett vises ved å sette opp rekurrenslikningen for en quicksort med jevn deling
T(n) = 2T(n/2) + Tsplitt(n). Som blir O(nlogn), med median3sort, median3sort i seg selv er O(1) fordi den bare har if-setninger i seg.

Kapittel 4

Liste som abstrakt datatype

Ei liste er en samling med data som representerer ulike objekter av samme type. Det kan være alt fra liste over antall biler registrert per år de siste 50 årene til liste over arkiverte verv i en bedrift til liste med navn og fødselsnummer på studentene på et bestemt studium. Og uendelig mye mer.
Definisjon
En abstrakt datatype er et begrep som vi knytter bestemte operasjoner til, men vi sier ingenting som hvordan den er implementert. Hvert objekt kaller vi et element i lista, og vi må kunne:

- Opprette lista
- Finne ut hvor mange leementer det er in lista
- Sette inn elementer i lista
- Fjerne elementer fra lista
- Tømme
- Finne et element i lista når vi vet hvor det står (oppslag i lista)
- Finne et element i lista når vi vet hvilken verdi det har (søking i lista)
- Gå gjennom alle elementene i lista sekvensielt og gjøre noe med hvert enkelt
- Finne største og minste
- Eventuelt sortere listeelmenetene med hensyn på et eller annet kriterium.

Liste implementert ved hjelp av en tabell

Opprette lista, finne lengden av den og tømme den

Int tab[10] eller int [] tab = new int [10] lager da en tabell med ti elementer. Det å finne lengden og kapasiteten er en O(1) operasjon. Hvis vi ønsker å tømme lista, gjør vi det ved å sette

lengden til 0. Bi behøver ikke viske ut de elementene som står der, når lengden er satt til 0, skrives det over de gamle når vi legger inn nye elementer derfor er det å tømme lista en O(1) operasjon.

Sette inn elementer og fjerne elementer fra lista

Der å sette inn elementer bakerst i lista er enkelt så lenge lengden er mindre enn kapasiteten. Da er det bare å sette inn det nye elementet direkte og øke lengden, og vi får en O(1) operasjon. Men straks vi når kapasitet, får vi et problem. For å få plass til flere elementer, må vi lage en ny og større tabell, og kopiere over alle de elementene vi allerede har til den nye tabellen, før vi setter inn det nye elementene hvis vi har n elementer blir dette en O(n) operasjon. Ønsker vi f.eks at det nye elementet skal komme først, må alle de andre flyttes et hakk, noe som gir en O(n) operasjon. Hvis vi antar at vi skal sette inn element nummer n+1 på posisjon i , vil antall elementer vi må flytte i, være (n-1). Hvis alle posisjoner er like sannsynlige, vil vi i gjennomsnitt flytte på ½(n+1) elementer. Dette viser at også en gjennomsnittlig innsetting er en O(n) operasjon i slike tilfeller. Når vi skal fjeren et element fra en usortert liste, kan vi gjøre det ved å flytte siste element inn på plassen til det elementet som skal fjernes og minske lengden med 1. Dette blir en O(1)-operasjon. Ønsker vi derimot at rekkefølgen av elementene skal være uforandret, må vi i stedet flytte alle de etterfølgende elememtneen et ahkk fremover, slik at vi får en O(n)-operasjon.

Lenket liste

Enkel liste

Definisjon
En node er en datastruktur som består av et element samt en eller flere lenker til en eller flere andre noder. En lenket liste er en samling noder der hver node leder videre til neste node lista. Første node kalles lenkas hode. I praksis vil vi vanligvis ha en lenke til første node, og da får gjene denne lenka navnet hode.

Vi bruker to klasser/strukturer, Node og enkelliste. Den består av et felt som inneholder de dataene vi ønsker å lagre same en lenke til neste node som vi kaller neste. Siden nodene inneholder felter av samme type som seg selv, får vi en rekursiv datastruktur. Siden nodene ikke har egne navn, er den eneste måten å få tak i en node på, å gå via neste-lenka. Så når vi

skal til slutten av lista må vi starte i begynnelsen og gå gjennom alle nodene til vi finner ei lenke som er null. I en enkel liste har vi to variabler, hode som er lenket til første node, og antall elementer, som holder styr på hvor mange noder vi har. Det siste er egentlig ikke nødvendig, vi kunne ha traversert lista og telt nodene hvis vi fikk bruk for å vite hvor mange det var, men ved å ha en egen variabel blir det å finne antall noder en O(1) operasjon i stedet for en O(n) operasjon. Når vi skal legge inn en ny node bakerst i lista kan vi traverse lista ved hjelp av en løkke konstruksjon. For hver runde i løkka sjekker vi om neste er null i så fall er vi kommet til siste node. Denne operasjonen blir dermed av størrelsesorden O(n). Hvis det ikke spiller noen rolle hvor den nye noden blir satt inn, er det lurere å plassere den først i lista. Da slipper vi å traversere lista før vi setter inn noden. Slik at vi får en O(1) operasjon i stedet for en O(n)-operasjon. Hvis vi derimot gjerne vil at nodene skal havne i samme rekkefølge som vi har lagt dem inn, slik at nyte noder må legge bakerst, kan vi bruke ei ekstern lenke for å få til O(1) kjøretid. Vanligvis har det liten hensikt å sortere en lenket liste siden vi ikke kan få noe bedre enn O(n) når vi skal finne igjen noder likevel. Men hvis man ønsker en sortert liste, vil det være enklest å sørge for å legge inn nodene sortert med det samme de plasseres i lista, noe som fører til en innsetning ordenene blir O(n). Alt i alt kan vi si at sammenliknet med tabell implementasjon, gir denne lenket liste implementasjonen følgende fordeler:

- Vi slipper å ta hensyn til at lista kan bli full.
- Vi bruker litt ekstra plass, nærmere bestemt plass til O(n) ekstra lenker.
- Vi har klart å få innsetting til å bli O(1) i alle tilfeller (bortsett fra hvis de nye nodene skal settes inn andre steder enn første eller sist).
- Oppslag er blitt vesentlig mindre effektivt, vi får O(n) i stedet for I(1). Sljing i usortert liste gir samme tidskompleksitet O(n) i begge tilfeller og vi kan ikke spare noe på å sortere lista.
- Sletting er O(n), mens det i tabell implementasjonen kan være O(1)

Dobbellenket liste

Noen ganger trenger vi å få tak i noden foran den noden vi ser på i øyeblikket. Med ei enkelt lenket liste må vi da starte på begynnelsen av lista og traversere den fram til vi finner den noden vi skal ha tak i, slik det er vist i metoden for å fjerne en node dette kan effektiviseres hvis vi kan traversere den linked elista fra begge veier. Halslenker er brukt for å få innsetting bakerst til å bli en O(1) operasjon En lenket liste med lenker i begge retingnerene kalle seg dobbel lenkeliste. Å fjerne en node fra den dobbeltlenke lista krever også en del flytting av lenker. Først tester vi om den noden som skal fjernes, er fremst. Hvis den ikke er det settes neste-lenka til noden foran til

å lenke til noidneenba og immostsatt fall er det hodet som skal lenke til noden ba. Deretter tester vi om den noden som skal fjernes er bakerst og setter enten neste nodes forrige eller halen til å lene til forrige node. I en dobbel lenket liste er det mulig å implementere quicksort slik at vi kan få sorteringtil å bli en O(nlogn). Men det har fortsatt liten hensikt. Ved å lage lista sirkulær kan vi sløyfe hale lenke. Da lar vi hele tiden siste node sin neste lenke til første node, og første node sin forrige lenke til siste node. Dermed kan vi finne siste node ved å be om hode sin forrige.

Iteratorer

Definisjon
En iterator er et objekt som brukes til å traverse en datastruktur på en bestemt måte.
Den har gjeren en opersjonfor å starte traverseringen for å finne gjeldende element i datastrukturen en for å flytte til neste element og en for å finne utom man er kommet itl sliutten for å få til dette må vi ha variabler som forteller hvilken datastruktur iteratoren er knyttet til og hvor i datastrukturen befinner seg.

Kapittel 5 Kø og Stakk

Kø

Definisjon
En kø er en lineær datastruktur der innsetting alltid skjer bakerst og uttak alltid fremst.
Dette er akkurat som i vanlig køer du stiller deg bakerst og venter til alle foran deg er ekspedert.
En annen betegnelse for det er FIFO “first in first out”.

Implementasjon

Når slike køsystemer skal programmeres bruker vi ei liste til å lagre de ventende elementene i. Da vet du fra orrgie kappitel at det er to måter implementere den på, enten som en tabell eller som en lenket liste. Å legge inn nye elementer bakerst i en tabell er enkelt , og er en O(1)-operasjon. Men å fjerne elementer fremst i tabellen er ikke så enkelt når vi må beholde rekkefølgen. Da må vi flytte alle elementene, og får en O(n)-operasjon. Men følgende lille triks gjør at man kan få en O(1)-operasjon. Vi later som tabellen er i en sirkel. I stedet for å flytte de

andre elementene når de først blir tatt ut, holder vi rede på hvor det fremste befinner seg. Innsetting av nytt element finner vi neste posisjon ved å skrive slutt = (slutt+1)%n der n er antall elementer vi har plass til. O når vi har hentet ut første element, oppdaterer vi start-referansen ved å skrive start = (start+1)%n. Vi kan bruke ei enkel lenket liste med hode og hale og alle operasjoner blir O(1)-operasjon.

Stakk

Definisjon
En stakk er en lineær datastruktur der både innsetting og utak skjer fremst. Her er det en som kommer sist, som blir ekspedert først urettferdig, men effektivt i noen sammenhenger. En annen betegnelse på stakk er LIFO " last inn først out". Kjennetegnet på en stakk er at de som legges i stakken først, havner nederst og kommer sist ut. For en stakk må vi ha følgende operasjoner. Hente ut verdien av øverste element fjern øverste element, legge inn et nytt element, finne ut om stakken er tom og finne ut om stakken er full. Det å legge et nytt element på staken kalles push og det å fjerne det øverste kalles pop.

Implementasjon

Når vi bruker en tabell, vil det første elementet vi pusher på stakken, legges i tabell element 0. Deretter fyller vi tabellen, fortløpende, slik at toppen på stakken flytter seg utover i tabellen. Tilsvarende når vi popper et element, får vi ut det elementet som ligger bakerst i tabellen, og lengden på tabellen minsker med en. Hvis vi bruker en variabel til å holde rede på hvor toppen til enhver tid befinner seg, vil begge disse bli O(1) operasjoner. Også å hente ut verdien av toppen samt det å finne ut om stakken er eller full er O(1).

Kapittel 6 Trær

Trær og grafer

Toppen av et tre kalles roten, og er punktet hvor treet vokser ut ifra. Under rota har vi noder på flere nivåer og noder i ulike nivå representerer ulike generasjoner i stamtavla eller familietreet. I stamtavla har hver node forbindelse til akkurat to andre noder på nivået under, mens for familie

treet kan hver node ha forbindelse til mange noder på nivået under. Et tre er et spesialtilfelle av datastrukturen graf. En graf er en samling noder med forbindelse mellom nodene. Forbindelsene kaller vi kanter. En vei mellom nodene x og y er en oppramsing av nodene som vi kan passere for å komme oss fra x til y når vi følger kantene. En graf er sammenhengende hvis det finnes minst en vei fra enhver node til enhver annen node. Et fritt tre er en sammenhengende asyklisk graf. Et tre med rot er en sammenhengende, asyklisk graf der en av nodene er utpekt til å kalles rota. En nodes subtær er de trærene vi får når vi fjerner kanten mellom noden og dens barn. Subtrær består av alle noden s. Etterkommere. En nodes forgjengere er alle noder på veien mellom noden og rota. Noder uten barn kalles ytre noder eller løvnoder, noder med barn er indre noder. Antall barn en node har, kalles nodens grad. Et tomt tre er et tre der rota er null. En nodes dybde er veilengden fra rota til den spesifikke noden. Et ordnet tre er et tre der det er bestemt rekkefølge mellom barna.

Binærtrær

Definisjon
Et binærttre er et posisionstre der hver node kan ha maksimalt to barn.
Dette er den mest vanlig typen tre. Et binærttre er perfekt hvis alle løvnodene befinner seg på nederste nivå, og alle indre noder har to barn. Et binært tre er komplett hvis det er perfekt bortsett fra at det kan mangle noen noder lengst til høyre i nederste nivå. Et binært tre er fullt hvis alle indre noder har akkurat to barn, de vil si at det er perfekt bortsett fra at det kan finne løvnoder andre steder i treet enn på nederste nivå. De vanligste operasjonene på et binærtre er å sette inn nye noder, ta vekk noder, finne node med en bestemt verdi samt traversere treet. Siden binærtreet er ordnet, må vi holde orden på rekkefølgen av barna, men i stedet for å kalle dem første og andre barn kaller vi dem venstre og høyre barn.

Høyde og dybde

Å finne høyde og dybde i treet hører ikke med til de operasjonene brukere av trestrukturen trenger så ofte. Men disse operasjonene er kjekke å ha for å lett implementasjon av noen andre operasjoner. Dybde = summen av av dybden til forgjengerne. node 4 = 1+ node2 = 2+node1=3+nodenull = 3-1 = 2. Høyde er det samme bare med en liten forskjell man tar det største tallet av høyden til nodene på samme nivå høyde(treet) = høydenode1 = 1+maks(node2, node3).... Osv

Traversering

Preorder node left right 1 2 4 7 6 3 5
Inorder left node right 7 4 2 6 1 5 3
Postorder left right node 7 4 6 2 5 3 1.

Binære søketrær

Definisjon
Et binært søketre er enten et tomt tre eller et binært tre der nodene har nøkler. Alle noder må tilfredsstille at nodens nøkkel er større enn nøklene til alle noder i venstre subtre, og mindre enn alle nøkler i nordens høyre subtre.

Innsetting

Når en node skal settes inn i det binære søketrær, sjekker vi først om rota er null. Er den det settes noden her, og vi er ferdige. I motsatt fall må vi begynne å sammenlikne nøkkelverdier. Hvis den nye nodens nøkkelverdi er mindre enn nøkkelverdier til den vi sammenlikner med, skal vi gå til venstre, er den større, går vi til høyre.

Søking

Å finne en node med en gitt nøkkelverdi skjer på omtrent samme måte som innsetting. Den eneste forskjellen er at nå må vi også teste på om nøkkelverdier er like, ikke bare mindre eller større. Hvis vi finner nøkkelverdien, returner vi den noden. Hvis vi kommer helt ned til en node uten det barnet vi skulle gått vidre til, kan vi fast slå at nøkkelverdien ikke finnes og returnerer null.

Sletting

Den vanskeligste operasjon på et binært søketre er å slett noder i det. Her må vi dele i tre ulike tilfeller:

1) Noden som skal slettes, har ingen barn Da kan den lenka fra foreldrene den som lenker til noden settes lik null.
2) Noden som slettes har ett barn. Da kan den lenka fra foreldrenode som lener til noden settes lik å lenke til nodens barn.

3) Noen som skal slettes har to barn. Da erstatter vi nodens element med element i den noden som kommer like etter i soteringsrekkfølen, som er den minste i høyre subtre. Denne noden vil oppfylle kravet som at nøkkelverdien er større enn alle nøkkelverdiene i venstre subtre. Og den oppfyller også at nøkkelverdier er mindre enn alle nøkkelverdier i høyre subtre.

Tidskompleksitet for operasjoner på binærtre

Teorem I et perfekt binærtre med høyde h, har vi n = 2^(h+1)-1 noder.
Bevis basis: Når h= 0 har vi bare rota alstå 1 node dette stemmer med formelen.
Induksjonstrinnet: La rota ha to subtær, hver med høyde h-1. Hvert av subtrærene har da 2^h -1 noder så hele treet får 1+2*(2^h -1) = 1+2^(h+1) -2 = 2^(h+1)-1 noder.
Dette viser at et perfekt tre med n noder har høyde h = log2(n+1) -1 E Θ(logn). Et perfekt tre har så liten høyde som det er mulig å ha, så alle andre binærtrær har h E □(logn). I verste fall kan et generelt binærtre være slik at alle noder bare har et barn. Slik at høyden bli n-1. I det generelle tilfellet vil høyden være et sted mellom O(logn) og O(n) men vanligvis mye nærere O(logn) enn O(n). Derfor vil vi vanligvis regne med at høyden er O(logn) når vi skal finne kompleksiteten til ulike operasjoner.

Dybde, høyde og traversering.

For å beregne treets høyde må vi starte i rota og gå nedover begge subtrærne. Kaller vi kompleksiteten for å beregne høyden tiltre med n noder for T(n), får vi t(n) = 2t(n/2)+1 da får vi Θ(n).

Innsetting, søking og sletting i binært søketre.

Hvis treet har høyden O(logn), vil derfor både innsetting og søking få kompleksitet O(logn) slettinger litt mer komplisert. I tilfelle 1 og 2 har vi ingen løkke og får O(1). I tilfelle 3 har vi løkke og et rekursivt kall. Antall runder i løkka vil være. o(h). Mens det rekursive kallet vil være O(1) siden det alltid er tilfelle 1 eller 2. Derfor ser vi at sletting blir O(logn) hvis h er det.

I verste fall får man O(n) AVl-trær er et tre hvor vi kan garantere O(logn).

B-trær

B-tre er en datastruktur som brukes for å få effektiv bruk av disker og andre sekundærlager. Den er derfor mye brukt i for eksempel datasystemer. B-trær er ikke binære; de kan ha hundrevis av barn. Hvor mange barn et bestemt tre kan ha, forteller vi ved å angi at det for eksempel er et 100 veis tre. Generelt gjelder at et mbsi tre er et tre som kan ha maksimalt m barn.

Definisjon

Et søk tre er et tre der nodene har en eller flere nøkkelverdier n1 til nk og nøklene er ordnet slik at n1<n2<n3..<nk. Nodene har k+1 lenker til eventuelle barn og nøklene o bar bi(o<i<k) ligger mellom ni og ni+1. Nøklene i b0 er mindre enn n1 og nøklene i bk er større enn nk.

Et B-tre er et søketre der alle løvnoder er på samme nivå og alle noder unntatt rota har mellom n-1 og 2n-1 nøkler for en eller annen n>1. Rota har mellom 1 og 2n-1 nøkler.

B-trær brukes for det meste når vi har så mye data at de må lagres på disk. Da er det et poeng å få så få diskaksesser som mulig, siden dette tar mye mer tid enn å finne data i primærlageret. Når vi skal søke i treet, trenger vi å hente fram en node på hvert nivå, så et tre med lav gøye er å foretrekke. Det kan vi oppnå ved at hver node har mange barn og dermed også mange nøkler. Vi velger n slik at en node tar et helt antall sider på disken, noe som gir effektiv framhenting.

Søking og innsetting

Søking i et B-tre gjøres omtrent samme måte som i et binært søketre. Vi starter med å smaneligne den nøkkelverdien vi søker med de nøklene som finne si rota. Siden nøklene er sortert, kan vi eventuelt bruke binærsøk for å finne ut om nøkkelen finnes i node eller hvilket barn vi må gå videre til . Dette barnet hentes fram og vi gjentar prosessen til vi enten finner nøkkelen eller kan fastslå at den ikke finnes siden vi har kommet til en løvnode.

Innsetting skjer alltid i en løvnode. Først må vi finne ut hvilken node den skal inn i, på samme måte som når vi søker etter en verdi. Hvis noden ikke er full plasserer vi den nye verdien på riktig plass i noden og er ferdig hvis noen derimot er full må den splittes i to nyenoder då m foreldrenoden få ei lenke ekstra og dermed også en nøkkel ekstra.

Sletting

For dette tret får vi følgende regler:

1) Hvis nøkkelen ikke finnes i den vi står i , og dette er en løvnode, kan vi fastslå at nøkkelen ikke finnes.
2) Hvis nøkkelen ikke finnes i den noden vi står i og dette er en indre node, må vi finne det barnet som vi skal gå videre med. Anta at dette er barn nr i
    a) Hvis i har minst n nøkler fortsetter vi videre nedover.
    b) Hvis i bare har n-1 nøkler ser vi på barnets forrige søken og sjekker om det har minst n nøkler. Hvis det er tilfellet flytter vi nøkkel i fra foreldre noden og sørger samtidig for å flytte det siste subtreet i i-1 over i i. Tilsvarende kan vi også sjekke neste søsken. Hvis ingen av søsknene har mer enn n-1 nøkler, må barn i slås sammen med et av sine søsken, og nøkkel i må flyttes fra foreldrene den ned i det nye barnet. Dette dp da 2n-1 nøkler, og vi kan fortsette nedover i treet.
3) Hvis nøkkelen finnes i den noden vi står i og dette er en løvnode, kan nøkkelen slettes direkte, siden vi nå har sørget for at det er nok nøkler til at en kan slettes.
4) Hvis nøkkelen finnes i den noden vi står i, og dette er en indre node, gjør vi følgende:
    a) Hvis vi enten kan flytte opp den største av de nøklene som er mindre eller den minste av de som er større gjør vi det, og sletter den nøkkelen i stedet.Dette krever at den noden vi henter nøkkelen fra har minst n nøkler før vi flytter opp en.
    b) Hvis vi ikke kan gjøre noen av delene, bety det at begge de aktuelle barna har n-1 nøkler. Da kan de slås sammen, og den nøkkelen som skal slettes, flyttes også ned til dette barnet. Så fortsetter vi videre med dette barnet.

Regler

1) Det må være (m eller n)/2 barn
2) Root kan bare min 2 barn
3) Alle løv noder må være på samme nivå
4) Bottom up oppretting
5) Når en node er full så sender vi en av nøkklene til foreldre noden

Kapittel 7 Heap Struktur og prioritetskø

Hva er en Heap?

I en max-heap har hver node en nøkkelverdi som er større enn eller lik begge barnodenes. Rota har dermed den største nøkkelen. I en min heap har hver node en nøkkelverdi som er mindre enn eller lik barnenodenes, dermed har orta den minste nøkkelverdien.

Anvendelser

En prioritetskø er en datastruktur hvor man kan legge inn og ta ut elementer som har ulik prioritet, Prioritetene kan justeres opp og ned. Forskjellen på en prioritetskø og andre lagringsstrukturer er at vi lett kan plukke ut elementet med den høyeste prioriteten. Vi ser at dette er enekelt når en max-heap brukes for å implementere prioritetskø. En heap er ikke den eneste måten å implementere en prioritetskø på. MAn kan bruke en lenket liste, en sortert liste eller en uordnet tabell. Disse metodene har imidlertid ulempen at noen metoder, som å sette inn eller ta ut en node, blir O(n). En fibonacci-heap er et alternativ med lavere kompleksitet, men koden og datastrukturen er såpass komplisert at det sjelden lønner seg for praktiske formål. En heap fylles opp fra venstre til høyre og bytter plass på rot noden hvis den finner noe som er enten større eller mindre etter hva heapen er sortert etter. Hvis et tall som er større/mindre en et annet blir satt under hverandre eller i lavere prio så bytter de plass.

Datastruktur og metoder

Metoden fiks_heap

Dette er en viktig metode som brukes av andre heap metoder. Den kalles for å korrigere en node muligens står for høyt opp i treet. Når vi bruker fiks_heap, antar vi at høyre og venstre subtre under noden er korrekte heaper, mens noden selv muligens har for lav nøkkelverdi i forhold til barne nodene. Hvis ikke er metoden ferdig, men hvis den har det, strider dette mot heap egenskapene. Så fiks_heap bytter noden med den av barne nodene som har høyest verdi for å rette på dette. Noden synker altsånedover i treet. Deretter kaller metoden seg selv på den nye nodeprosidjonen, i fall noden treger å synke enda lengre ned.

Analyse

ALgoritmen inneholder ingen løkker, men den kaller seg selv rekursivt når noden må flyttes. Bortsett fra reksushjbner er algoritmen Θ(1), så kjøretiden blir proporsjonal med antall metodekall som er en mer enn antall flytt. Noden flyttesd bare en vei nedover i treet. Antall flytt og dermed mmetodejall, begrenses dermed av nodens høyde i treet, h. Koplkisteten for fiks_heap blir dermed O(h). O(h) er O(logn).

Metoden lag_heap

Denne metoden brukes for å lage en heap av en uordnet mengde tall. Dette er for eksempel aktuelt når vi skal lage en prioritetskø og allerede har en del elementer som skal være med. En uordent tallmengde form av en tabell kan sees på som et komplett binærtre, akkurat som en heao. Men tallene i en silk mendel tilfrestiller vanglivis ikke heap egenskaper.

Analyse

Kallet til fiks_heap er O(logn), sp en skulle to at lag_heap ble O(logn). Men slik er det ikke mange av nodene vi bruker fiks_heap på, ha lavere høyde enn log n, så kompleksiteten for lag_heap blir faktisk Θ(n) i stedet.

Bevis

Fiks_heap har komplektistet O(h), hvor h er nodens høyde i treet. En heap med n noder har høyden log n, og det er maksimalt n/2^h+1 nodeer med en gitt høyde h. Dermed kan vi skirve tidsforbuket som

T(n) E O(sum(h=0, log n, (n/2^h+1)*h)) = O(n*sum(h=0, log n, h/2^2)) (1)

Hvis vi setter inn h for k og x = ½ får vi

Sum(h=0, inf (½)^h) = sum(h=0, inf , h/2^h) = 1/2/(1-½)^2 = 2 (2)

Etter som vi arbeider med asymptotisk notasjon, kan vi kombinere ligningene 1 og 2 og får t(n) E O(n*2)) O(n) selv om den øvre grensen på summeformelen ikke passer helt. I dette tilfellet fører det bare til at 2 blir litt før høy konstant, men den fjerner vi jo uansett. While delen av løkka er Ω(n) uansett hva slags kompleksitet fiks_heap får. Dermed har vi samme grense ovenfra og nedenfra og t(n) E Θ(n).

Metoden hent_maks

Å finne største tallet i en maks-heap er enkelt. Takket være heapegenskapen er det bestandig rotnoden, som vi kan finne med en komplexitet på Θ(1). I prioriteres køer har vi til tider bruk for å fjerne det største tallet f.eks fordi det brukes opp da ønsker vi at det største tallet skal rykke opp og bli rotnoden. Mesteparten av algoritmen er Θ(1), bortsett fra kallet til fiks_heap som gir en kompleksitet på O(log n)

Metoden sett inn

Det er ikke bare å sette i noden nederst når vi setter inn, det kan jo hende den har høyere prioritet enn foreldre noden. Derfor bruker vi prio_opp med et tillegg på 0. Prioriten forandres ikke, men noden flyttes oppover hvis det trengs. Kallet til prio_opp gir denne metoden en kompleksitet på O(log n).

Metoden heapsort

Til nå har vi brukt heapen som en delvis sortert strucktur. Med heapsort kan vi også få en fullstendig sortering. Metoden begynner med et kall til lag_heap som lager en heap av usorterte dta. Dette kallet er O(n). Deretter tar metoden vare på lengden av heapen. En løkke plukker ut det største tallet og setter det inn igjen i tabellen bakenfor heapen. Dermed blir heapen mindre og mindre mens en større og større del av tabellen blir sortert. Vær oppmerksom på at hent_maks trekker 1 fra heaplengden sp metoden trenger ikke selv gjøre dette. Til slutt setter vi tilbake lagrede heap lengden. Vi kaller hent_maks n-1 ganger. Hvert kall er O(log n) og Ω(1), det siste fore kommer hvis alle tallene er like. Løkka og dermed heapsort , har kompleksitet O(n log n) og Ω(n).

Kapittel 8 Hashtabeller

Hva er en hashtabell?

Tabeller er velkjent fra grunnleggende programmering. Tabellelemtenee kan være hva som helst og indekserer med heltall. Oppslag har kompleksitet Θ(1) og plass forbruket er Θ(n) der n er antall. Dette fungerer ikke hvis vi skal bruke indekser som er større enn ints eller som ikke er ints. Det er her hashtabeller kommer inn, i en hashtabell så gir vi hvert element en nøkkel som blir hashet til en plass i tabellen, dette gir også samme kompleksitet som en vanlig tabell.

Lastfaktor

Definisjon

For en hashtabell med størrelse m som inneholder n elementer er lastfaktoren gitt ved: a=n/m
En dårlig utnyttet hashtabell har dermed en lastfaktor a nær 0, mens en full tabell har a = 1. Overflyte tabeller kan ha a over 1.

Problemer med hashtabeller

Hashtabeller har et problem og det er at to poster med ulike nøkler kan få samme verdi av hash funksjonen. Men en tabellposisjon kan bare lagre en post. Dette kalles en kollisjon. Kollisjoner er nesten umulig å unngå fordi hash funksjonen konvertere nøkler med mer en om mulige nøkkelverdier til heltall innenfor. [0, m-1]. En mulig løsning er å si at førstemann får plassen, mens nestemann som haser til en opptatt posisjon får ikke være med. Det kan imidlertid være upraktisk og vil virke rart hvis tabellen fremdeles har massevis av ledig plasser.

Hashfunksjoner

Hash Funksjoner tar en nøkkel som parameter, og bergner en posisjon i hashtabellen. Den er uavhengig av hva tabellen inneholder fra før. Beregning skal skje raskt, på Θ(1) tid. En god hashfunksjon skal gi god spredning slik at postene fordeles jevent utover tabellen. En dårlig hashfunksjon vil gi unødvendig mange kollisjoner og håndtering av disse tar tid. I ekstreme tilfeller kan kollisjonene også øke kompleksiteten.

Hash Funksjoner basert på rest divisjon

En hashfunksjon etter divisjons metoden omformer en nøkkel k til en av m poisjoner i hashtabellen ved hjelp av følgende formel: h(k) = k mod m
Ikke alle verdier for m gir like godt resultat. Å bruke rest divisjon gir dermed føringer for hvilke størrelser har tabellene våre kan ha. Hvis feks m = 256 får vo hashfunskjone h(k) = k mod 256. Denne funksjonen avhenger imidlertid bare av de åtte siste bitene i nøkkelen resten av nøkkelen påvirker ikke resultatet i det hele tatt. Alle verdiene for m = 2^x ahr dette problemet. Liknende problemer oppstår for m= 10^x hvis nøklene våre er vanlige tall i titallsystemet. Gode

verdier for m er primtall, som ikke bær ligge for nær noen toeråoptens. Med en slik m vil h(k= kunne variere uansett hvilken del av k som varier og vi får dermed større sjanse for god spredning.

Hash Funksjoner basert på multiplikasjon

Multiplikasjon er en annen måte å lage hash funksjoner på med denne måten gagner vi første nøkkelen k med et tall a der 0<A<1 produktet består av en heltallsdel og en desimal del. Vi ser bort fra heltallsdelen og ganger desimal delen med størrelsen på hashtabellen m matematisk kan det skrives slik: h(k) = [m*(kA-[kA])]. Denne hash funksjonen fungerer for alle verdier av m, den ligger altså ingen føringer for størrelsen på hashtabellen., Men den generelle varianten beskrevet her bruker desimaltall og beregner på desimaltall er så å si alltid trege sammenlignet med heltall operasjoner.

Kollisjons Håndtering

Lenkende lister

Lenkede lister er en enkel løsning på kollisjons problemet. Hver post kan ha en lenke til neste post og hash tabellen blir en tabell med liste hoder. nÅr vi setter inn en post i tabellen, lenker vi den smelte inn først i den aktuelle listen. O(n) for look up fordi kan hende vi må gå gjennom en lang liste med noder for å finne riktig node i lenken.

Åpen adressering

Åpen adressering er en annen måte å håndtere kollisjoner på. Med denne måten lagres alle elementene i selve hash tabellen det er ingen lister eller andre eksterne strukturer. I denne metoden finner vi en nye plass til elementet ved en kollisjon.

Lineær probing

Dette er den enkleste formen for åpen adressering. Hvis nytt element hasher til en plass som allerede er opptatt, prøver vi simpelthen neste posisjon i tabellen. Er den også opptatt går vi ett steg videre og gir oss ikke før vi eventuelt har prøvd alle plassene. Hvis siste posisjon i tabellen er opptatt, fortsetter vi å lete etter ledige plasser fra den første plassen og utover. På dette viset er vi sikre på at vi alltid får lagt inn nye elementer så sant tabellen ikke er helt fullt. Lineær probing er probe metoden definert slik: probe (h,1,m) = (h+1) mod m

Kvadratisk probing

Kvadratisk probing er en forbedring i forhold til lineær probing, ved at elementer som kolliderer på forskjellige steder, prøver ut forskjellige alternativer. Dermed vil en haug elementer som kolliderer i posisjon 1, ikke komme i veien for elementer som kolliderer i en annen posisjon. Probe Sekvensen beregnes slik: probe(h,i,m) = (h + k1*i + k2*i) mod m

Dobbel hashing

Vi har sett at kvadratisk probing gir ulike probe sekvenser for elementer som hasher til forskjellige posisjoner. Men det er fortsatt problemer for elementer som hasher til samme posisjon hvis den er opptatt, vil begge prøve på nytt på samme sted. Dobbel hashing unngår dette ved å bruke to hash funksjoner i probe sekvensen, en som angir den første posisjonen vi skal prøve, og en annen som angir hvor langt vi skal hoppe hvis det blir kollisjon. Formelen for probesekvensen ser slikt ut: probe(h1,h2,i,m = (h1+i*h2) mod m

Dobbelt hashing stiller krav til hash funksjoner h2(k):

1. Funksjonen h2(k) må aldri får verdien 0.
2. h2(k) og tabellstørrelsen m må være relativt prime. Det vil si at de ikke må ha felles faktor for noen verdi av k. Hvis de har en felles faktor, vil ikke probe sekvensen gå innom alle posisjonene , og det kan bli umulig å plassere et element. Hvis f.eks m = 50, h1(k) = 3 og h2(k) = 10, vil probesekvensen are prøver pos 3,13,23,33, og 43. Hvis disse er opptatt, hjelper det ikke om alle de andre er ledige.
3. For å minimere antall kollisjoner bær også h2(k) mod m i største mulig grad gi ulike tall der h1(k) gir like tall. Dette kravet er ikke absolutt da hashing vil virke også om det ikke oppfylles,men vi i så fall ikke få noen forbedring i forhold til dobbelt hashing.

Kravet om ingen felles faktor kan oppnås på flere måter:

- Hvis m er et primtall, kan vi la h2(k) produsere tall i intervallet [1,m] f.eks h2(k) = k mod(m-1) +1 dette oppfyller også krav 3 ovenfor.
- Hvis m er en toerpotens kan vi ha h2(k) som bare produserer oddetall, f.eks h2(k) = (2|k|+) mod m
- Vi kan bruke en hvilken som helst funksjon til å lage et forslag til tall for deretter å tilpasse det hvis det viser seg å ha en felles faktor med m tilpassing kan f.eks gjøres med en løkke som legger til en passende verdi og tester på nytt. Løkker har imidlertid lett for å øke kompleksiteten.

Kapittel 9 Grafteori

Innledning

Hva en graf er

En graf er en datastruktur som kan lagre mange typer informasjon. Grafen består av en mengde noder og kanter mellom disse. En slik datastruktur egner seg for å lagre mange slags informasjon som kan brukes for å løse praktiske problemer. Både noder og kanter kan inneholde ekstra informasjon, avhengig av hva slags problem det gjelder. En graf i denne sammenhengen har ingenting å gjøre med kurver av typen y = f(x), kjent fra matematikken. Når vi har kant(n,m), ser vi at m er nabo til n og det skrives n->m. Det omvendte er ikke nødvendigvis tilfelle i en rettet graf. En kan som begynner og slutter i samme node kalles en selvløkke. En vei er en kjede av noder som henger sammen med kanter og lengden er antall

kanter den inneholder. En rundtur er en vei som begynner og slutter i samme node. En enkel vei eller rundtur er en som ikke inneholder den samme noden flere ganger, bortsett fra at rundturer naturligvis slutter der de begynner.

Sammenlikning av naboliste og tabell

Mange graf algoritmer itererer over kantene ut fra en node. Med ei naboliste er tidsforbruket proporsjonalt med antall kanter ut fra noden, og en løkke over alle kanter ut fra alle noder O(K). Med tabell representasjonen er tidsforbruket for å iterere over kanter ut fra en node O(N), og en løkke over alle kanter ut fra alle noder blir $O(N^2)$. N^2 er maksimal verdi for K, men for praktiske grafer er K ofte mye lavere enn det. I en graf hvor nodene representerer veikryss og kantene veiene dem, vil det typisk være 3-4 kanter per node. K vil altså være proporsjonal med N, ikke N^2. Tabellen har den fordelen over nabolista at man kan slå opp en bestemt kant direkte. Med naboliste må man lete gjennom kantene til den aktuelle noden for å finne en bestemt kant. Det tar mer tid, men er ikke noe problem for de mange algoritmene som uansett itererer over alle kanter ut fra den enkelte noden. Naboliste gir lavere kompleksitet for mange av graf algoritmene.

Bredde først-søk (BFS)

Søket starter i en startnode s , og finner alle noder som kan nås fra den. Det kalles bredde først-søk fordi det går i bredden: BFS finner alle noder med avstand c fra start noden så det finner noen med avstand x +1. BFS konstruere et bredde først-tre som viser korteste vei fra s til alle noder som kan nås fra. BFS kan dermed brukes for løse en enkel variant av korteste vei problemet. Det brukes også som en komponent i andre grafalgorimter. Alle noder får uendelig avstand og ingen forgjenger. Unntaket er startnoden som får avstand 0. Det begynner med et kall til init, deretter legges start noden inn i køen. Den ytre løkka plukker en og en node ut av køen, og legger den i variabelen n for å undersøke kantene dens, ved å traversere kantlisten. Hvis vi finner en node hvis avstand er uendelig, er den ikke funnet fra før. Da setter vi dens avstand lik mer enn n-avstanden, ettersom veien til den går via n. Forgjengeren setter vi lik n, så vi senere vet hvor den ble funnet fra. Noden legges dessuten i køen. Slik at vi med tiden kommer til å undersøke kantene som går ut fra den også.

Analyse

Init setter en avstand på hver av nodene, den er dermed O(N). Den ytre løkka går så lenge det finnes noder i køen. I verste fall kommer etter hvert alle nodene innom køen, det skjer alltid hvis alle noder kan n fra start noden. Noder kan bare havne i køen hvis de har uendelig avstand, men da får de også satt avstanden til en endelig verdi. Dermed kan hver node maksimalt havne i køen en gang, så den ytre løkka i BFS har O(N) kompleksitet. Den indre løkka undersøker alle kanter ut fra en node, og avhenger dermed av hvor mange kanter hver node har. Derfor kan vi ikke si noe om kompleksiteten for en omgang men til sammen kan vi komme til å undersøke alle kanter ut fra aller noder og får dermed en kompleksitet på O(K) for den indre løkka. Til sammen her BFS kompleksitet O(N+K).

Dybde først-søk (DFS)

Dybde først-søk går dypere i grafen så sant det er mulig. Hver gang algoritmen finner en node, prøver den straks å gå videre til en annen ved å følge den første kanten ut. Dermed har vi lett for å finne lange sammenhengende kjeder med noder. HVis kanten imidlertid fører til en node vi har funnet før, prøver algoritmen neste kant. Når alle kanter ut fra en bestemt node er undersøkt, går algoritmen tilbake noden som ble funnet før den, og fortsetter med andre kanter ut fra den. I sin enkleste form bruker vi DFS på samme måte som BFS. Vi starter søkealgoritmen fra en startnode, den finner alle noder som kan nås fra den, og lager et BFS-tre med mulige veier gjennom grafen. Til forskjell fra BFS er ikke disse veiene nødvendigvis kortest mulig.

Analyse

Metoden dfs_init itererer over all nodene og er dermed $\Theta(N)$. Metoden dfs_sok itererer over alle kanter ut fra en node og kaller seg selv rekursivt hver gang en kant leder til en node som ikke er undersøkt ennå. Kompleksitet for denne metoden er dermed $O(N+K)$.

Topologisk sortering

Topologisk sortering er aktuelt når vi planlegger aktiviteter som avhenger av hverandre, det vil si at noe må være ferdig før andre kan begynne. Betingelsene kan representeres med en rettet graf med en node per aktivitet. Når en aktivitet må være ferdig før en annen begynner, legger vi inn en kant fra den første til den andre. Topologisk sortering går ut på å ta en slik graf og sette nodene i en mulig rekkefølge. Legg merke til at det kan være mange riktige rekkefølger for topologisk sortering.

Litt mer forklaring

Algoritmen kan virke litt tilfeldig. Rekkefølgen på sidene avhenger av hvordan nodene er nummerert, enn hvordan de faktisk henger sammen, og DFS velger veier gjennom grafen basert på rekkefølgen i kantlisten som egentlig ikke har noe med problemet å gjøre. Nøkkelen til å forstå topologisk sortering ligger i at DFS legger nodene i resultatlista i samme rekkefølge som de blir ferdige, altså når alle kantene ut fra dem er ferdig undersøkt.

Analyse

Den første løkka i topologisortering er $\Theta(N)$. Den andre løkka repeterer N ganger, og kaller df_topo hver gang. DFS er i utgangspunktet $O(N+K)$, men her gjør vi N søk på den samme grafen. Df_topo sørger for at vi aldri undersøker de samme nodene flere ganger. Dermed blir den totale kompleksitet $\Theta(N+K)$.

Sammenhengende og sterkt sammenhengende grafer

Definisjon

En urettet graf kan deles inn i sammenhengende deler. I en sammenhengende del finnes det en vei fra enhver node til enhver annen , og hele grafen sies å henge sammen hvis det bare er en slik del. En rettet graf eller en del av en slik, er sterkt sammenhengende hvis det finnes vei fra enhver node til enhver annen. Dette må gjelde begge veier, vi trenger både a-b g b-a. Algoritmen for å finne sterkt sammenhengende deler er som følger:

1. Kjør DFS på alle nodene i grafen, så du får et eller flere DFS tær. DFS beregner ferdig tider for alle nodene.
2. Sorter nodene i grafen på synkende ferdig-tid
3. Lag den omvendte grafen, G^T
4. Kjør DFS på alle nodene i den omvendte grafen. Start med den som fikk høyest ferdig-tid, og fortsett nedover. Det skulle være enkelt etter etter som nodene er sortert slik.
5. Hvert av DFS trærne etter punkt 4 vil nå være en sterkt sammenhengende del av grafen.

Analyse

DFS har på samtlige noder i grafen kompleksitet Θ(N+K). Å sortere nodene kan gjøres på Θ(N) tid med tellesortering. Den omvendte grafen kan framstilles på Θ(N+K) tid for en graf implementert som naboliste. Kompleksiteten for hele operasjonen blir dermed Θ(N+K).

Vektede Grafer

I grafene vi har sett på så langt, finnes bare en slags informasjon om kantene, om de finnes eller ikke. I en vektet graf har vi data, vanligvis et tall, for hver kant, som generelt kalles vekten, v. Hva vektene representerer, avhenger av hva grafen brukes til. Når grafen implementerer med naboliste, lagrer vi vektene i kantene, når vi bruker nabo tabell lagres vektene i tabellen istedenfor bare lage 1 og 0.

Korteste vei-problemet

Korteste vei-problemet dukker opp i mange sammenhenger. Foruten den korteste veien mellom to steder kan de samme algoritmene brukes til å finne billigste eller raskeste rute. Problemet kan løses på mange måter. En metode som vil føre fram er å prøve mulige veier og se hvilken som er kortest. Dette er imidlertid upraktisk, da det fort blir uhorvelig mange kombinasjoner av tilgjengelige veier.

Varianter

1. *En til alle*
   Resultatet er korteste vei-tre med start noden som rot samt informasjon om de andre

nodenes avstander fra startnoden. Å finn korteste vei til bare en annen node har ikke lavere kompleksitet, ettersom veien i verste fall går innom alle de andre nodene.

2. *Alle til en*
   Her finner vi den korteste vei fra alle andre noder inn til en målnode. For en urettet graf er det samme problemet, for en rettet graf kan vi beregne den omvendte grafen G^T, og deretter bruke en vanlig korteste vei-algoritme med mål noden som startnode.
3. *Alle til alle*
   Her finner vi den korteste veien fra hver node til hver av de andre. Det kan gjøres ved å bruke en en til alle algoritme en gang for hver node. Det finnes imidlertid mer effektiv algoritmer for dette.

Kanter med negativ vekt

En til alle kan videre deles inn i algoritmer som håndterer kanter med negativ vekt, og de som ikke gjør det. Negative vekter på kantene er selvsagt ikke aktuelt når vi snakke rom, reisetid og geografiske distanser; men de kan forekomme når det er snakk om pris. Dermed tjener vi på å kjøre disse strekningene med negativ kostnad. Under slike forhold kan turen med lavest pris inneholde en lang omvei for å ta med passasjerer.

Rundturer

En rundtur begynner og slutter i samme node. Korteste vei fra en node til en annen kan ikke inneholde en rundtur, fordi rundturen har en lengde som vi unngår ved ikke å ha den med i veien vår. Når vi har kanter med negativ vekt, kan det hende vi har en rundtur hvor summen av kantene i runden er negativ. Hvis denne runden kan nås fra startnoden har ikke korteste vei-problemet noen løsning.

Dijkstra's algorithm

Dijkstras algoritme er en fler algoritmer som løser korteste vei-problemet. Den forutsetter at ingen av kantene har negativ vekt. Hvis slike kanter likevel forekommer, får vi et galt resultat. Dijkstras algoritme bruker prioritetskø for å holde orden på hvilken node som har lavest distanse estimat blant de som den ikke er ferdig med. Algoritmen må plukke ut den noden som til enhver tid har kortest distanse, fordi denne kan bidra til kortere veier til andre noder. Hvis vi hadde plukket noder med lengre avstand først, ville vi risikert at noder vi var ferdige med fikk kortere avstander senere, og dermed måtte vi sett på dem om igjen. Dette er grunnen til at Dijkstras algoritme ikke håndterer negative kanter, for en negativ kant som går tilbake til en node vi er ferdig med kan gi denne en kortere avstand enn den hadde fra før.

Analyse

Kjøretiden avhenger av hvordan prioritetskøen implementeres. Dijkstras opprinnelige algoritme brukte en tabell, og fant minimums noden med et lineært søk. Init er $\Theta(N)$, dette gir en kompleksitet på $\Theta(N^2)$. Den indre løkka behandler alle kantene ut fra en node. Den kjøres en gang for hver node i grafen, dermed får vi en kompleksitet på $\Theta(K)$. Hele algoritmen får dermed

en kompleksitet på $\Theta(N^2+K)$ som kan forenkles til $\Theta(N^2)$. Hvis vi bruker en heap så får vi en kompleksitet på $\Theta((N+K) \log N)$ fordi innsetting og oppdatering av heapen skjer nå i metoden og begge disse er log n og de kjøres for både N og K.

Bellman Ford-algorithm

Til forskjell fra Dijkstras algoritme klarer denne algoritmen grafer hvor noen av kantene har negative vekter. Bellman Ford-algortimen bruker også metoden forkort() for å komme fram til de korteste veiene, men der Dijkstras algrotime greier seg meg ett kall til forkort() per kant, bruker Bellman-Ford en løkke som gir N-1 omganger med et kall til forkort() for hver kant. Testen inni løkka er den samme som testen i forkort(), hvis det finnes ytterligere potensial for forkortning etter de N-1 omgangene må det være en negative løkke. Etter init har startnoden avstand 0, en avstand som ikke vil endre seg i en graf uten negative rundturer. Etter første omgang med forkort() på alle kantene har algoritmen nødvendigvis funnet den neste noden etter startnoden i korteste vei-treet. Vi vet ikke nødvendigvis ennå hvem av dem det er, men minst en har fått sin endelige lengde. I neste omgang finner vi ferd med minst en node til som får sin endelige lengde, og slik fortsetter det. I verste fall trenger vi N-1 omganger, det skjer hvis korteste vei-treet er en lang kjede med noder. En slik kjede kan imidlertid ikke inneholde mer enn N-1 kanter uten å inneholde rundturer, og rundturer forekommer ikke ettersom veien alltid er kortere uten. Dermed vet vi at de korteste veiene må være funnet etter N-1 omganger. Hvis imidlertid det forekommer en negativ rundtur, vil vi kunne bruke forkort() uendelig mange gange og likevel få forbedriner for hver gang.

Analyse

Init er $\Theta(N)$. Trippleløkka går K kanter N-1 ganger deretter går løkka den siste løkka gjennom de K kanten en gang til. Kompleksiteten for Bellman Ford-algoritmen blir dermed $\Theta(NK)$.

Minimale spenntrær

Definisjoner

Når vi har en sammenhengende vektet urettet graf med N noder, er et spenntre definert som et tre som kobler sammen alle nodene ved hjelp av N-1 kanter valgt blant de K kantene i grafen.

Kostnader for et spenntre er summen av vektene på alle kantene som er med i spenntrær.

Et minimalt spenntre er spenntrær med lavest mulig kostnad for den gitte grafen. Når flere kanter har samme kostnad, kan det være flere ulike minimale spenntrær.

Kruskal's algoritme

Kruskals algoritme begynner med å se på hver node som et frittstående tre. Deretter bruker ei løkke som slår sammen trær ved å ta i bruk passende kanter. Algoritmen velger alltid den lavest

vektede kanten som fører til at vi slår sammen to forskjellige trær til et. Dermed unngår vi rundturer, og treet blir minimalt fordi vi alltid bruker den lavest vektede kanten når vi slår sammen. For å kunne finne den korteste kanten i hele grafen trengs en ekstra datastruktur. Metoden kurskal bygger opp spenntreet. Det inneholder ei løkke som går gjennom den sorterte kant tabellen. Algoritmen sjekke om til og fra noden er i samme mengde. Hvis de er det er de allerede i det samme treet. I så fall kan ikke kanten brukes, da det vil være til en rundtur. Hvis derimot nodene er i hver sin mengde, er de i hvert sitt tre, og da tar vi kanten med i spenntrær. Når vi gjør det, må mengdene oppdateres for at fremtidige tester skal vite hvilke noder som er i samme tre.

Analyse

Tidsforbruket avhenger av hvordan mengdeoperasjonene lag_mengde, finn_mengde og mengdeunion implementers. Det går ikke boka inn på, men de beste kjente implementasjonene av disse er O(ma(n)),der m er totale antallet mengdeoperasjoner (lag,finn og union) mens n er antall ganger vi bruker metoden lag_mengde a(n) er meget langsomt stigende funksjon , med lavere orden enn logaritmefunksjonen. I praksis er det bare tallet 4 siden det ikke stiger over det. Den ytre løkka i init utføres N ganger. Vi får dermed N kall til lag_mengde. Den indre løkka ser på nodens kanter, ytre og indre løkke til sammen blir dermed Θ(K).Til slutt sorterer vi kant tabellen, med tidsforbruk Θ(KlogK). Metoden kruskal har en løkke som er Θ(K). Den gjør 2K kall til finn_mengde og n-1 kall til mengde union. Flere kan det ikke bli, ettersom spenntreet inneholder akkurat N-1 kanter. Dette blir til omtrent O(N+K).

Prims algoritme

Prims algoritme velger en startnode og bygger opp spenntreet ved å koble en og en enslig node til treet, inntil alle er med. Noder som ikke er tatt med ennå, ligger i en prioritetskø ordnet på avstand fra resten av treet. En nodes avstand fra treet er lengden på den korteste kanten mellom noden og treet. Noder som ikke har dirkete forbinelse har uendelig avstand, i startfasen har alle det utenom startnoden som har avstand 0. Når algoritmen tar med en ny node n, ser den på alle kantene som går fra n til mx. Hvis mx ikke er med i treet ennå og har større avstand enn vekten på kx oppdateres mx med den lavere distansen og forgjengeren settes til n. Dette gjør det mulig alltid å velge noden som har kortest avstand fra resten av treet. Prims algoritme trenger å holde orden på hvilke noder som til enhver tid er med i spenntreet.

Analyse

Som for Djikstras algoritme avhenger også her kjøretiden av hvordan prioritetskøen implementeres. Med tabell får vi $O(N^2)$ og med heap får vi O((N+K)logN)

Maksimal flyt

Flytnettverk er nok en praktisk anvendelse av grafer. Vi flytter et eller annet fra node til node, langs forbindelser som har en maksimal kapasitet som ikke kan overskrides. Materialer kan ikke

samles opp i en node, alt som kommer inn må straks videre gjennom utgående forbindelser. Det finnes to spesielle noder; *kilden*, hvor materialet kommer inn i grafen, og *sluket* hvor de forsvinner ut.

Definisjon

Maksimal flyt-problemet går ut på å finne den maksimale flyten vi kan ha gjennom et slikt nettverk, fra kilde til sluk, uten å gå over kapasitetsgrensen noe sted.

Noen regler for flyt

Et flytnettverk er en rettet graf hvor hver kant (n,m) har en kapasitet k(n,m) flyten gjennom kanten kalles f(n,m). Vi regner med at alle noder ligger på en eller annen vei mellom kilden og sluket.
Kapasitetsregelen: f(n,m) <= k(n,m). Flyen må være mindre enn eller lik kapasiteten.

Symmetriregelen: f(n,m) = -f(m,n) Flyt i en retning sees på som negativ flyt i motsatt retning.

Flytkonservering: Summen av alle flyt inn og ut fra noder utenom kilde og sluk må være null. Nodene har altså ingen lagringskapasitet, alt må videre straks.

Ford-Fullkerson-metoden for maksimal flyt

Ford Fullkerson-metoden bruker flyt økende veier. En slik vei er en hvilken som helst vei fra kilde til sluk hvor det finnes kapasitet i riktig retning langs hele veien.
Algoritmen er slik:

1. Initialiser flyten til 0 for alle kanter
2. Øke flyten langs flyt økende veier, så lenge vi kan finne slike. Flyten økes så mye som mulig, altså like mye som kapasiteten i den kanten langs veien som har lavest kapasitet. Dermed oppfylles reglene for kapasitet og flyt konservering.

Restnett

For å bruke flyt økende veier må vi finne dem først. En flyt økende vei må gå fra kilden til sluket, og den må gå via kanter som enten har kapasitet til å transportere mer enn de allerede gjør, eller kanter hvor vi kan kansellere eksisterende flyt som går i feil retning. For å finne veier beregner vi et restnett, som er en graf med oversikt over restkapasitet og muligheter for forandring i flyten i hele grafen. For enhver kant er restkapasiteten gitt ved følgende formel kr(n,m) = k(n,m) - f(n,m). Legg merke til at restkapasiteten kan være større enn kapasiteten når flyten er negativ, fordi vi kan kansellere feil flyt i tillegg til å bruke den eksisterende flyten. Når vi har restnettet, kan vi finne en flyt økende vei ved å starte i kilden, og finne en vei gjennom restnettet til sluket. Ettersom restnettet består av kanter som alle har kapasitet for å øke flyten, vil denne veien være en flyt økende vei. Neste steg blir å finne ut hvilken kant i veien som har lavest restkapasitet, og øke flyten langs hele veien akkurat så mye. Deretter må vi beregne nytt restnett for å lete etter neste flyt økende vei.

Edmonds karp-algorithm

Ford Fullkorn-metoden sier ikke noe om hvordan vi skal finne en flyt økende vei gjennom rest nettet. Dette kan gjøres på mange måter f.eks med et BFS eller DFS søk. Vi starter søket i kildekoden, og avslutter så snart det finner sluket. Hvilken måte en velger har imidlertid overraskende mye å si for kjøretiden, velger vi BFS får vi O(K|f|), forutsatt at vektene er heltall. DFS er ikke nødt til å gjøre så dårlig valg men kan det og kan derfor enten får O(K|f|) eller O(|f|). Kompleksitet blir $O(NK^2)$.

Kapittel 10

Ulike optimalisering metoder

Rå kraft-algoritmer

En måte å finne den optimale løsningen på er å beregne alle muligheter og velge den beste. Dette forutsetter at det ikke er uendelig mange muligheter. Det er gjerne enkelt å programmere, men vil ofte gi veldig mye arbeid for datamaskinen. Av den grunn omtales slike algoritmer gjerne som rå kraft-algoritmer, altså algoritmer der vi bruker mye datamaskin-kraft.

Splitt og hersk-algoritmer

Da prøver vi å finne optimale sub problemer som kan løses og settes sammen til en optimal løsning av det opprinnelige problemet. Avhengig av hvordan sub problemene avhenger av hverandre, har vi tre ulike teknikker, vanlig splitt og hersk der vi bruker rekursjon, og to teknikker der vi ikke bruker rekursjon. Disse teknikkene kjennetegn ved at de har overlappende subproblem, dvs. At de samme sub problemene dukker opp flere ganger.

Probailistiske algoritmer

Hvis vi kan nøye oss med å få en ganske god løsning, ikke nødvendigvis den aller beste, kan vi bruke statistiske metoder til å finne en løsning. Dette er teknikker der vi bruker loddtrekning som et middel til å finne løsningen. Da får vi sjekket ut et variert utvalg av punkter i løsningsrommet, og vil ofte kunne lokalisere et optimum relativt raskt.

Dynamisk programmering

Dynamisk programmering likner på splitt og hersk-teknikken i det at teknikken går ut på å dele problemet i mindre deler som løses og deretter settes sammen til en løsning på hele problemet. Men der splitt og hersk starter på toppen og bruker rekursjon til å komme seg nedover til basis og deretter tilbake til toppen, starter dynamisk programmering nederst og finner og lagrer løsningen som settes sammen til stadig nye løsninger på veien opp til toppen. Løsningsmetoden egner seg når de samme del problemene dukker opp mange ganger. Når

løsningen på disse delproblemer er beregnet og lagret, kan disse løsningene brukes på nytt neste gang samme del problemet dukker opp. Vi har allerede sett på et eksempel på denne teknikken i kapittel 2, ved beregning av fibonacci-tallene . Den rekursive løsningen av dette problemet beregnet de samme Fibonacci-tallene mange ganger, og vi kunne effektivisere programmet ved å ta vare på løsningene etter hvert som vi fant dem. I fibonacci-eksemplet trengte vi bare å ta vare på de to siste løsningene for å finne neste, men normalt trenger vi alle eller mange av de løsningene vi har produsert tidligere. Da er det naturlig å bruke en tabell til å lagre dem i. De neste eksemplene viser dette. Begge disse eksemplene er optimalisering problems. Men dynamisk programmering kan også brukes i andre sammenhenger.

Å kombinere rekursjon med dynamisk programmering

Dynamisk programmering brukes altså når de samme del problemene dukker opp mange ganger. Som regel er det enklest å beregne delproblemer systematisk nedenfra og opp, men noen ganger kan det være lurere å starte på toppen. Da bruker vi rekursjon til å komme ned til basis og deretter jobbe oss oppover igjen, samtidig som vi lagrer resultatene fra de rekursive kallene etter hvert som de produseres, Dette er hensiktsmessig hvis det er bare noen av de mange mulige del problemene som faktisk må beregnes, og vi ikke på forhånd vet hvilke det blir behov for. Før hvert rekursive kall sjekker vi om resultatet allerede er beregnet. Hvis det er det bruker vi dette resultatet, hvis ikke, setter vi i gang rekursjonen. Denne teknikken kalles på engelsk memorization.

Grådige algoritmer

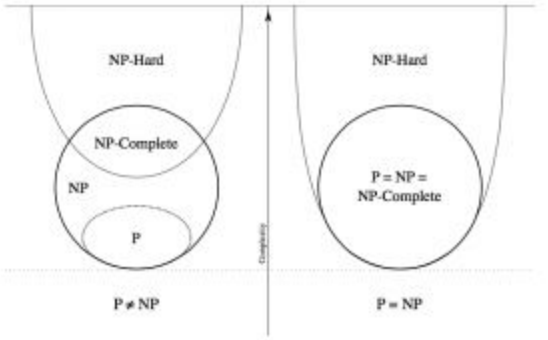
En grådig algoritme er en algoritme som alltid gjør det som synes best i øyeblikket uten tanke på helheten. Noen ganger gir dette en optimal løsning, mens andre ganger kan det lønne seg bedre å være noe mindre grådig. Men selv om vi ikke alltid finner den aller beste løsningen, vil vi ofte finne en løsning som er bra nok. Det kan av og til være vanskelig å finne ut om vi trenger en grådig algoritme eller dynamisk programmering. Da må vi se på om vi kan gjøre et endelig valg her og nå, eller om vi bare kan gjøre foreløpige valg som kanskje må vurderes senere. Hvis vi kan gjøre endelige valg, bør vi prøve med en grådig algoritme, i motsatt fall vil dynamisk programmering være nødvendig.

Evolusjonære algoritmer

Evolusjonære algoritmer baserer seg på utviklingslæren. Vi ser på ulike mulige løsninger som individer i en befolkning. De beste lar vi leve og formere seg og optimal løsning. Som alltid ved optimalisering må vi ha en måte å avgjøre om en løsning er bedre enn en annen. Dette kaller vi målfunksjonen. Målfunksjonen kan være alt fra en enkel matematisk funksjon til en komplisert algoritme, simulering eller fysisk realisering av det som løsningen representerer, med måling av egenskapene. Men siden målfunksjonen skal regnes ut mange ganger, prøver vi alltid å lage den så enkel som mulig.

NP-Komplette problemer

NP ⇒ Nondeterministic Polynomial
NPC ⇒ NP-Complete
P ⇒ Polynomial
Et problem er NPC hvis det er et “decisionproblem"og NP-Hard hvis det er et “optimalisering problem".
Sammenhengen mellom alle NP-problemer:
P ⊆ NP
NP C ⊆ NP
Hvis et problem A ∈ NP C, så vil også A ∈ NP. Dette er fordi NP = P + NP C. Hvis noe ikke er i P eller NPC, er det heller ikke i NP, fordi P ∩ NP C = ∅.



Figur: Venndiagramet, med med P 6= NP og P = NP

Du vet at problem A er i NP og problem B er i NPC. Du vil vise at A også er i NPC. Da reduserer du fra B til A.

Du står overfor de tre problemene A, B og C. Alle tre befinner seg i mengden NP. Du vet at A er i mengden P og at B er i mengden NPC. Anta at du skal bruke polynomiske reduksjoner mellom disse problemene til å vise . . . :
1. . . . at C er i P må C ≤ A (C reduseres til A)
2. . . . at C er i NPC må B ≤ C (B reduseres til C)
3. . . . hvis B kan reduseres til A er P = NP (NB: ikke løst enda)
4. Alle disse reduksjonene skjer i polynomisk tid
To ikke helt legitime, men greie huskeregeler: reduser fra litt til mer // fra litt . . . ≤ . . . til mer
reduser nedover: NPC → (NP) → P (forsiktig med denne)

Liste/strukturen til NPC-problemer redusert fra CIRCUT-SAT:

- CIRCUT-SAT
- SAT
- 3-CNF-SAT
    - SUBSET-SUM
    - CLIQUE

- VERTEX-COVER
- HAM-CYCLE
- TSP (Traveling Salesman Problem)

Oppsumering

Kompletthet

Et problem er komplett for en gitt klasse og en gitt type reduksjon dersom det er maksimalt for redusibilitets relasjonen.

Maksimalitet

Et element er maksimalt dersom alle andre er mindre eller lik. For reduksjoner: Q er maksimalt dersom alle problemer i klassen kan reduseres til Q.

NPC

De komplette språkene i NP, under polynomiske reduksjoner.

NP-hardhet

Et problem Q er NP-hardt dersom alle problemer i NP kan reduseres til det. Et problem er altså NP-komplett dersom det er NP-hard og er i NP.

Hvis man får til å redusere en NPC problem til enten NP eller P så har man gjort noe stort ta får du likheten P=NP=NPC.

Reduksjon

Det å redusere problem A til problem B betyr kort at vi sier at B er vanskeligere eller likt problem A så vi får er ulikhet som ser sånn ut A<=B. Dette betyr for eksempel hvis A er et P problem så blir B et NP problem eller hvis A er NPC så er B NPC grunnen til det site og at NPC ikke blir NP-hard er fordi alle NPC problemer kan reduseres til hverandre.

Kompleksitet og forklaring

Grovinndeling i stigende rekkefølge:

1. Konstant: 1
2. Polylogaritmisk: (logb n)^k
3. Polynomisk: n^k
4. Eksponentiell: 2^n
5. Faktoriell: n!
6. Alt som er enda verre, f.eks. n^n

Pseudo Polynomial Itet

Gitt en algoritme som tar tallet n som input og har kjøretid O(n) – hvilken kompleksitetsklasse er dette? n betegner her ikke størrelsen på input, men er selv en del av inputen. Det vil si at størrelsen til n = antall bits som kreves = lg(n). n = 2lg(n) = 2w ⇒ NB: Eksponentiell kjøretid! Dukker ofte opp som lurer oppgave på eksamen!

Kjøretider på pensumalgoritmer

Problem	**Algoritme**	**Kjøretid**		
		BC	**AC**	**WC**
Sortering	Insertion Sort	$\Theta(n)$	$\Theta(n^2)$	$\Theta(n^2)$
	Bubble Sort	$\Theta(n)$	$\Theta(n^2)$	$\Theta(n^2)$
	Merge Sort	$\Theta(n \lg(n))$	$\Theta(n \lg(n))$	$O(n \lg(n))$
	Heap Sort	$O(n \lg(n))$	$O(n \lg(n))$	$O(n \lg(n))$
	Quick Sort	$\Theta(n \lg(n))$	$O(n \lg(n))$	$\Theta(n^2)$
	Counting Sort	-	$\Theta(n)$	$\Theta(k+n)$ hvis $k = O(n)$
	Radix Sort	-	$\Theta(d(n+k))$	-
	Bucket Sort	-	$\Theta(n)$	-
Grafer/Trær	Toplogisk sortering	-	$\Theta(\|V\| + \|E\|)$	-
	DFS	-	$\Theta(\|V\| + \|E\|)$	-
	BFS	-	$O(\|V\| + \|E\|)$	-
	Prim	$O(E \lg(V)$	$\Theta(E \lg(V))$	-
	Kruskal	$O(E \lg(V)$	$\Theta(E \lg(V))$	-
	Binærsøk	$O(\lg n)$	-	$O(\lg n)$
Korteste vei	Bellman-Ford	-	$O(\|V\| \times \|E\|)$	-
	Dijkstra	$O(\|E\| \lg(\|V\|)$ (bin-heap)	$O(\|V\|^2)$ (array)	-
	DAG-Shortest Path	-	$\Theta(\|V\| + \|E\|)$	-
	Floyd-Warshall	-	$\Theta(\|V\|^3)$	-
Flyt	Ford-Fulkerson	-	$O(E\|f^*\|)$	f^* = maks flyt i G
	Edmonds-Karp	-	$O(VE^2)$	-
Grådig	Huffmann	$O(n \lg(n))$	-	-

Noen rekursive kompleksiteter

Rekursive kall	Størrelse, delproblem	Arbeid i hvert kall	Rekurrens	Kjøretid
Ett	Redusert med 1	Konstant	$T(n) = T(n-1) + \Theta(1)$	$\Theta(n)$
Ett	Halvert	Konstant	$T(n) = T(\frac{n}{2}) + \Theta(1)$	$\Theta(\lg n)$
Ett	Redusert med 1	Lineært	$T(n) = T(n-1) + \Theta(n)$	$\Theta(n^2)$
Ett	Halvert	Lineært	$T(n) = T(\frac{n}{2}) + \Theta(n)$	$\Theta(n)$
To	Redusert med 1	Konstant	$T(n) = 2T(n-1) + \Theta(1)$	$\Theta(2^n)$
To	Halvert	Konstant	$T(n) = 2T(\frac{n}{2}) + \Theta(1)$	$\Theta(n)$
To	Redusert med 1	Lineært	$T(n) = 2T(n-1) + \Theta(n)$	$\Theta(2^n)$
To	Halvert	Lineært	$T(n) = 2T(\frac{n}{2}) + \Theta(n)$	$\Theta(n \lg n)$

Binærsøk:

Θ(log n)

Innsettingsort:

O(n^2) men kan være raskere for nesten sorterete elementer eller under 10 elementer. Insertion Sort er en enkel sorteringsalgoritme. Den tar de to første elementene og plasserer i forhold til hverandre. Deretter plasserer den det neste i forhold til de to forrige, osv. Veldig effektiv å bruke på små mengder. Kan feks. brukes i slutten på en Quick Sort algoritme.

```
for j = 0 to A.length do
        key = A[j] i = j − 1 while i > 0 and A[i] > key do
                A[i + 1] = A[i]
                i = i − 1
        end while
        A[i] + 1 = key
end for
```

Bobblesort:

Θ(n^2)
Bubble Sort testet to og to naboelementer. Dersom den første er større bytter de plass. Effektiv på små datamengder.

```
for i = 0 to A.length − 1 do
        for j = A.length downto i + 1 do
                if A[j] < A[j − 1] then
                exchange A[j] with A[j − 1]
                end if
```

```
        end for
end for
```

Velgesort:

Θ(n^2)

Shellsort:

O(n^2) men kan være O(n^3/2) eller O(n^7/6) - O(n^5/4) kommer ann på hva man deler s med for hver runde.

flettesort :

O(nlog n)

Quicksort:

O(nlogn) men i noen tilfeller kan den være O(n^2)
Quick Sort er enda en “split-og-hersk”-algoritme. Man starter gjerne Quick Sort ved å randomisere lista. Den starter med å velge et pivotelement. Den deler deretter lista i to partisjoner: en med elementene som er mindre eller lik pivoten, og en med elementene som er større en pivot. Deretter kaller den seg selv rekursivt på de to partisjonene. Deretter fletter man sammen de to partisjonene.

```
function QUICKSORT(A, p, r)
        if p < r then
                q = PARTITION(A, p, r)
                QUICKSORT(A, p, q − 1)
                QUICKSORT(A.q + 1, r)
        end if
end function
function PARTITION(A, p, r)
        x = A[r]
        i = p − 1
        for j = p to r − 1 do
                if A[j] ≤ x then
                        i = i + 1 10
                        exchange A[i] with A[j]
                end if
        end for
        exchange A[i + 1] with A[j] return i + 1
end function
```

Tellesort: Intern tellsort

O(nlogn) Θ(n+k)

Counting Sort tar et heltall N mellom 0 og k. Lager en liste med verdier fra 0 til k og setter inn tallene på sin plass i lista. Fungerer best når verdiene på tallene som sorteres ligger tett etter hverandre og k ikke er for høy. function

```
COUNTINGSORT(A, B, k)
        let C[0 . . . k] be a new array
        for i = 0 to k do
                C[i] = 0
        end for
        for j = 1 to A.length do
                C[A[j]] = C[A[j]] + 1
        end for
        for i = 0 to k do
                C[i] = C[i] + C[i − 1]
         end for
        for j = A.length down to 1 do
                 B[C[A[j]]] = A[j] C[A[j]] = C[A[j]] − 1
        end for
end function
```

Radiasort:

O(nlogn)

Tabell list

operasjon	usortert	sortert
Lage	O(1)	O(1)
Finne antall	O(1)	O(1)
Sette inn elementer	O(1)	O(n)
Fjerne elementer	O(1)	O(n)
Tømme	O(1)	O(1)
Finne elementer på angitt plass	O(1)	O(1)
Søke etter element	O(n)	O(logn)

Trastevere	O(n)	O(n)
Finne største/minste	O(n)|	O(1)
Sortere	O(nlogn)	-

Lenket liste

operasjon	enkelt lenket	double lenket
Lage	O(1)	O(1)
Finne antall	O(1)	O(1)
Sette inn elementer	O(1) eller O(n)	O(1)
Fjerne elementer	O(n)	O(1)
Finne elementer på angitt plass	O(n)	O(n)
Søke etter element	O(n)	O(n)
Trastevere	O(n)	O(n)
Finne største/minste	O(n)	O(n)
Sortere	O(n^2)	O(nlogn)

Kø med enkelt lenket liste med hode og alle variabler

O(1) på alle kø operasjoner
Kø er en FIFO, “Fist In, First Out”-datastruktur. En kø implementeres som oftest som et array (gjerne et dynamisk array, likt Java sin ArrayList. En kø har egenskapene ENQUEUE og DEQUEUE.

Stack

O(1)
Stack er en LIFO, “Last In, First Out”-datastruktur. Stack implementeres oftest med et array, eller en lenket liste. En stack har egenskapene INSERT, PUSH og POP (DELETE).

Binære søke Trær

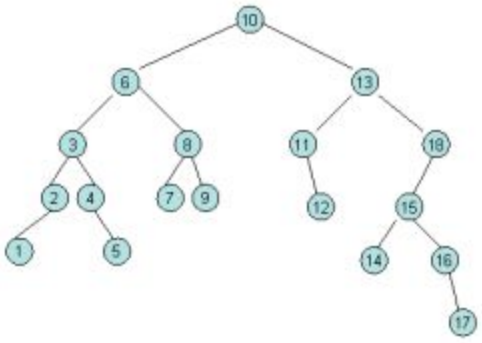
Høyde = H = O(logn)

Operasjon	Average	Worst
Plass (bit)	$O(\log n)$	$O(n)$
Søk	$O(\log n)$	$O(n)$
Sett inn	$O(\log n)$	$O(n)$
Slett	$O(\log n)$	$O(n)$

La x være en gitt node i søketreet. Hvis y er en node i det venstre subtreet til x må y sin verdi være mindre eller lik (≤) x sin verdi. Tilsvarende for høyre subtre. Her må y sin verdi være større enn eller lik (≥) x sin verdi.

Traversering av binæretrær

Vi benytter oss av tre måter å traversere et binært tre på, disse er: Preorder Her printer man ut nodens verdi før dens barn, venstre og deretter høyre. Inorder Her printer man venstre barn, noden, og deretter høyre barn (om ikke det er noe venstre barn, print noden før høyre barn) Postorder Her printer man nodens verdi etter man har printet venstre og høyre barn
Et eksempel fra figur 5:



Preorder 10, 6, 3, 2, 1, 4, 5, 8, 7, 9, 13, 11, 12, 18, 15, 14, 16, 17
Inorder 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18
Postorder 1, 2, 5, 4, 3, 7, 9, 8, 6, 12, 11, 14, 17, 16, 15, 18, 13, 10

B-trær

O(h) = O(logn)

Heap

En heap er en spesiell trestruktur, som tilfredstiller heap-egenskapen. Det finnes ingen regler for hvordan søsken er ordnet i heapen. En heap er ofte implementert med et array. Det er viktig å legge merge til at første indeks skal være tom. Det vil si at i et 0-indeksert array vil plass 0 være tom, mens første node står på indeks 1.

Operasjon	Tidskompleksitet
Finn min/max	$\Theta(1)$
Slett min/max	$\Theta(\log n)$
Sett inn	$\Theta(\log n)$
Omstrukturere heap	$\Theta(\log n)$
Merge	$\Theta(n)$

Kommentar: Min/max avhenger om det er min/max heap

Heap-egenskapen

I en heap må hele treet være fullstendig fylt ut, bortsett fra muligens det laveste nivået, som fylles ut fra venstre.

- Rotnode: i = 1
- Foreldrenode(i): i/2
- Høyre-barn: (2i + 1), Venstre-barn: (2i)

Max-Heap:

For hver node i, bortsett fra rot-noden, må verdien til barnenode være mindre eller lik foreldrenoden. A[P ARENT(i)] ≥ A[i]

Min-Heap:

For hver node i, bortsett fra rot-noden, må verdien til en barnenode være større eller lik foreldre noden. A[PARENT(i)] ≤ A[i]

HashTabell

Hashing er en effektiv måte å lagre (key, value)-pairs. Istedenfor at indeksen er nøkkelen, er heller indeksen basert på nøkkelen med en hashfunksjon.

Operasjon	WC	BC
Søke	$O(1)$	$O(n)$
Sett inn	$O(1)$	$O(n)$
Slette	$O(1)$	$O(n)$
Plass	$O(n)$	$O(n)$

Kommentar: Usortert array

Direct-address tables

Fungerer best/optimalt når mengen mulige nøkler (n) er forholdsvis lav. Ved å lage en tabell med n felter, kan hver indeks representere en nøkkel

Hash-tables

En mer effektiv måte å lage en slik tabell på. Man har en mindre tabell enn antall elementer. Hvert felt har derimot en egen nøkkel. Disse nøklene blir generert av hashfunksjonen, h(k). På grunn av at tabellen er mindre enn antall elementer kan kollisjoner oppstå (flere nøkler, gir samme hash

BFS

O(N+K)
BFS implementeres med en kø. BFS utforsker grafen i bredden. Man starter på foreldrenoden og legger inn alle dens barn i køen. Når alle naboer til node x er oppdaget, fjernes den fra køen og man tar den neste noden i køen, og legger alle dens barn inn i køen. Når køen er tom, sjekker man ikke videre om det er ubesøkte noder.

```
function BFS(G, v) .
        lag en kø Q
        legg v inn i Q
        while Q.notEmpty() do
                v = Q.dequeue()
                for each edge e adjacent to v do
                        if e not marked then
                                mark w
                                Q.enqueue(e)
                        end if
                end for
        end while
end function
```

DFS

O(N+K)
DFS implementeres med en stack. DFS utforsker grafen i dybden. Den fyller stacken med noder den støter på. Kan man ikke går videre vil den “backtrace” til forrige node, og se etter en mulig vei videre. Hvis stacken er tom, sjekker man om alle noder er besøkt. Hvis ikke, starter man på nytt fra ubesøkte noder.
function DFS(G, v) .

```
        v er startnode initaliser en tom stack, S
        for each vertex u in G do
                set visited[u] → false
        end for
        S.push(v)
        while S.notEmpty() do
                u = S.pop()
                for all w adjacent to u do
                        if not visited[w] then
                                visited[w] → true
                                S.push(w)
                        end if
                end for
        end while
end function
```

Topologisksort

Θ(N+K)
Topologisk sortering bruker en DAG(Rettet asyklisge grafer eller rettede grafer uten sykler) til å finne en rekkefølge gjennom alle elementene i grafen. Det tillates ikke sykler. Man begynner i noden som ikke har noen kanter inn til seg. Hvis en graf er topologisk sortert betyr det at den har bare enkelt noder som sterkt sammenhengende komponenter, altså alle nodene i grafen er en sterkt sammenhengende komponent alene .

```
function TOPOLOGICALSORT(G)
        DFS(G) to compute finish times v.f
        for each vertex v. as each vertex is finished,
        insert it onto the end of the list
        return the list of verticies
end function
```

Sterk Sammenhengende komponenter

Θ(N+K)

Relax/forkort

```
function RELAX(u, v, w)
         if v.d > u.d + w(u, v) then
                v.d = u.d + w(u, v)
                v.π = u
                Cormen bruker π til å representere foreldrenode.
```

```
    end if
```

Dijkstras

O((N+K)log N)
Dijkstra er en korteste vei, en-til-alle algoritme. Tillater ikke negative kanter. Den velger noder en etter en fra hvor nærme de er startnoden. Dijkstra er en grådig algoritme.

```
function DIJKSTRA(G, w, s)
    INITIALIZE-SINGLE-SOURCE(G, s)
    S = ∅
    Q = G.V
    while Q 6= ∅ do
        u = EXTRACT-MIN(Q)
        S = S ∪ {u}
        for each vertex v ∈ G.Adj[u] do
            RELAX(u, v, w)
        end for
    end while
end function
```

Bellman Ford

Θ(NK)
Bellman-Ford er en korteste vei, en-til-alle algoritme. Den tillater negative kanter. Returnerer f alse dersom det finnes en negativ sykel i grafen. Går igjennom alle kantene og bruker RELAX på hver av dem |V | − 1 ganger. Deretter sjekker den etter negative sykler ved å sjekke om veien fra startnoden til node w via node v blir mindre enn den vi fant under søket.

```
function BELLMANN-FORD(G, w, s)
    INITIALIZE-SINGLE-SOURCE(G, s)
    for i = 1 to |G.V | − 1 do
        for each edge (u, v) ∈ G.E do
            RELAX(u, v, w)
        end for
    end for
    for each edge (u, v) ∈ G.E do
        if v.d > u.d + w(u, v)
            then return false
        end if
    end for
    return true
end function
```

Minimale spenntrær

Er et tre som bruker alle de opprinnelige nodene og er en delmengde av den opprinnelige kant mengden. I en minimal spenntre algoritme så er input en urettet graf og output er en asyklisk delmengde som kobler sammen nodene i grafen og minimerer vektsummen. Her har vi to algoritmer i pensum. Kurskal og prims.

Flyt Problemer

Flytt in = Flytt ut. Mengde flytt på en kant er summen av alt som går inn og ut altså summen av alt som går inn minus alt som går ut. Vi kan sette inn noder hvor vi vill så lenge flyten opprettholdes. I rest nettet så er Framover kanter ledig kapasitet og bakover kanter er flyt. Augmented path er en sti fra kilde til sluket i rest nettet. Langs framover kanter kan Flyten økes og langs bakover kanter kan Flyten omdirigeres.

Vi kan sende flyt baklengs langs kanter der det allerede går flyt. Vi opphever da flyten, så den kan omdirigeres til et annet sted. Det er dette bakover kantene i rest nettet representerer, men vi sender ikke flytt baklengs i et flyt nett det er bare en representasjon i et rest nett. Når vi traverser gjennom flytt nettet lagrer vi all flytt i rest nettet framover betyr mulighet for å gå mens bakover betyr mulighet for å stjele. Når vi traverserer nettet så kan vi bare sende minste flyt mulig gjennom nodene, altså den minste vekten i veien er maksimalt flyt vi kan sende gjennom alle kantene fra start til slutt.

Kruskals

O(N+KlogK) og for sammenhengende grafer O(KlogK)
Kruskals algoritme sorterer alle kanter etter kostnaden og velger den billigste tilgjengelige kanten , og legger til denne med noder i treet. Dette kun hvis 13 den ikke allerede er brukt og vil danne en sykel. Fortsetter til det ikke finnes kanter som kan legges til i treet.

```
function KRUSKAL(G, w)
        A = ∅
        for each vertex v ∈ G.V do
                MAKE-SET(v)
        end for
        sort edges of G.E into nondecreasing order by weight w
        for each edge (u, v) ∈ G.E, taken in nondecreasing order by weight do
                if FIND-SET(u) 6= FIND-SET(v) then
                        A = A ∪ {(u, v)}
                        UNION(u,v)
                end if
        end for
        return A
end function
```

Prims

O((N+K)logN)
Prims algoritme velger en tilfeldig node, og legger alle kantene inn i en prioritetskø etter vekt. Velger den billigste kanten og legger den til i treet. Det vil si at den legger til den billigste kanten som er mulig å legge til fra treet den bygger.

```
function PRIM(G, w, r)
        for each u ∈ G.V do
                u.key = ∞
                u.π = NIL
        end for
        r.key = 0
        Q = G.V
        while Q != ∅ do
                u = EXTRACT-MIN(Q)
                for each v ∈ G.Adj[u] do
                        if v ∈ Q and w(u, v) < v.key then
                                v.π = u v.key = w(u, v)
                        end if
                end for
        end while
end function
```

Edmonds Karp

O(NK^2)
Endret en bokstav på Ford-Fulkerson metoden. De benytter seg av BFS. Edmonds-Karp bruker Ford-Fulkerson og BFS til å finne flytforøkende stier.

Max Flow/Min Cut

Max Flow/Min Cut Minimum-snitt (min-cut) pa et flytnettverk: det snittet som har lavest kapasitet av alle snitt. Det vil si min-cut angir en flaskehals i flytnettverket. Det vil si at det ikke kan sendes mer flyt gjennom nettverket enn det vi kan sende gjennom flaskehalsen. Man kan da ikke finne noen flytforøkende sti over flaskehalsen. Det vil da være maksimal-flyt, max-flow.