TDT4120

Algoritmer & Datastrukturer

Kompendium

Forord

I faget Algoritmer og Datastrukturer handler det enkelt sagt å finne på smarte måter å løse gitte problemer på. En algoritme er en veldefinert prosedyre som tar inn en eller flere verdier og gir tilbake en eller flere nye verdier. Det blir på en måte en "kokebok" som forteller hvordan man steg for steg skal gå frem for å løse problemet. Et eksempel på dette (som dere ofte vil møte på i faget) er sortering. De verdiene som kommer inn kan være et gitt antall kort, og ved å bruke en sorteringsalgortme på disse kortene vil vi kunne få en hånd med sorterte kort.

En datastruktur er på sin side en måte å lagre og organisere data på for å forenkle tilgang og modifiseringer. Faget tar dermed sikte på å lære hvordan man kan implementere gode algoritmer med fornuftige datastrukturer. Selv om selve algoritmene er uavhengige av programmeringsspråk, så er det lagt opp til at de skal implementeres på en datamaskin. Derfor skrives algoritmene stort sett i pseudokode. Det vil si at at koden i struktur skrives som en generell datakode, men med flere ord (og litt matematiske symboler). Dette gjør at man fort skal kunne "oversette" pseudokoden til det programmeringsspråket man ønsker å bruke.

Hensikten med kompendiet er i hovedsak at den vil gi en lettfattelig presentasjon av pensum. Samtidig ønsker vi at den skal være enkel å slå opp i, og den vil fungere som en fin repetisjon av faget. Men den vil ikke være en erstatning for læreboken. For det første vil vi ikke kunne være i nærheten av å gi samme dybde i pensum. Dessuten er det av plasshensyn ikke detaljerte pseudokoder her (selv om det ofte vil være kortere versjoner av dette som letter forståelsen av algoritmen). Men vi håper at kompendiet vil være et nyttig suplement ved siden av læreboken!

Vi vil takke Marius Thaule og Vegard Bertelsen for deres innsats som korrekturlesere.

Lykke til med Alg.Dat!!!

Med vennlig hilsen

Rikke Amilde Løvlid & Johan L. Rambech Dahl

2003

Innhold

Kapittel 1

KJØRETIDER og KJEKKE FORMLER

For å vite hvor effektiv en algoritme er, ønsker man å vite hvor kjapp den er i forhold til hvor mye innformasjon man sender inn. Noen ganger vil en dobling i antall inn-verdier bare øke kjøretiden med en konstant tid, andre ganger vil den doble kjøretiden, og ofte vil den øke veldig mye mer. Man prater ofte om **asymptotisk kjøretid** for algoritmer. Det vil, enkelt sagt, si at man finner en (teoretisk) matematisk funksjon for kjøretiden med antall inn-verdier som parameter. Deretter ser man på det leddet som dominerer mest i uttrykket når inn-verdiene blir store nok, og bruker dette (uten noe konstantledd) som et mål på hvor rask algoritmen er.

Eksempel 1.0.1. Å lete gjennom en enkel følge med tall.
Anta at du har mange forskjellige tall, og du skal finne det største tallet. Du sjekker da det første tallet, og setter denne som en midlertidig mulighet for at det er det største tallet. Deretter går du igjennom resten av tallene og sammenligner hver og en av dem med det tallet du antar er det største. Hvis du plukker opp et større tall enn du har fra før, setter du det nye til å være en mulighet for å være det største. Når du har gått gjennom alle tallene, vil det tallet du satte av som en mulig kandidat sist være det største tallet. Slik må det være, fordi da du plukket opp det største tallet satte du dette som en mulig kandidat og ingen andre tall var større. Dette gjelder selvsagt også om flere tall er like i verdi. Så lenge du ikke kunne ane hvor i bunken det største er da du begynte, vet du at du er nødt til å gå gjennom alle tallene en gang.

På den annen side trenger du ikke gå gjennom tallene flere ganger heller, for ved å bruke algoritmen beskrevet ovenfor, vet du at svaret er korrekt etter nøyaktig en gjennomgang (så lenge du ikke gjør noen feil underveis, vel og merke!). Hvis mengden med tall dobles og du antar at du alltid vil bruke like lang tid på hver sammenligning uansett hvor mange tall du har, vil arbeidstiden også dobles. Det kan f.eks se slik ut:

Du har tallene 2 3 1 7 5

1. steg: Du sjekker tallet 2, 2 er kandidat til å være det største tallet.
2. steg: Du sjekker tallet 3, 3 er kandidat til å være det største tallet.
3. steg: Du sjekker tallet 1, 3 er fortsatt kandidat til å være det største tallet.
4. steg: Du sjekker tallet 7, 7 er kandidat til å være det største tallet.
5. steg: Du sjekker tallet 5, 7 er fortsatt kandidat til å være det største tallet.
Og nå som du har gått gjennom alle tallene, vet du at 7 er det største tallet!

❑

Når det gjelder kjøretider er det viktig å ha noen matematiske symboler som beskriver dette.

1.1 Θ , O og Ω -notasjon

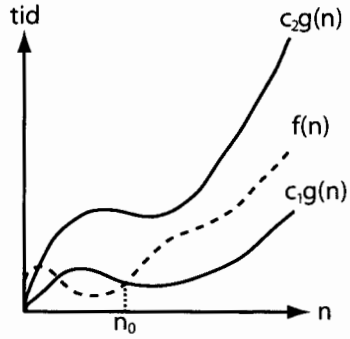
Disse symbolene må man bli kjent med bruken av. Vi vil her gi en definisjon for hver og en av dem, og så forklare hva det faktisk betyr.

1.1.1 Θ -notasjon

Denne notasjonen er den mest nøyaktige av de tre. Har du denne, har du også de to andre! Her er $f(n)$ en beskrivelse av kjøretiden, hvor n er størelsen på inn-verdi. Selve funksjonen til kjøretiden kaller vi $g(n)$. Nå har vi at $\Theta(g(n))$ defineres som følger:

Definisjon. $\Theta(g(n)) = \{\ f(n)$: slik at det finnes positive konstanter c_1, c_2 og n_0 så vi har $0 \leqslant c_1 g(n) \leqslant f(n) \leqslant c_2 g(n)$for alle $n_0 \leqslant n\ \}$

Dette vil si at $f(n)$ blir skvist mellom to kurver som begge bare har en konstant forskjell, så lenge input-verdien n er stor nok. Forvirrende? Les gjennom eksemplet som følger og se på den flotte grafen i figur 1.1!



Figur 1.1: $f(n) = \Theta(g(n))$

Eksempel 1.1.1.
Anta at du vet at kjøretiden til en algoritme er gitt ***nøyaktig*** *til å være* $g(n) = \frac{1}{2}n^2 + 3n$. *Da vil denne sies å ha kjøretid* $\Theta(n^2)$ *fordi ved å velge* c_1 *mindre enn* $\frac{1}{2}$ *og* c_2 *større enn* $\frac{1}{2}$ *kan vi finne en eller annen stor nok n slik at* $c_1 n^2$ *alltid er mindre enn* $\frac{1}{2}n^2 + 3$ *og* $c_2 n^2$ *alltid vil være større enn* $\frac{1}{2}n^2 + 3$.

❑

Det vil sjelden være aktuelt å finne konstantene c_1, c_2 eller n_0. Poenget er bare å se hvor kjapp algoritmen er i forhold til andre algoritmer når inn-verdiene blir store nok. Ofte er det lett å se hvordan kjøretiden til en algoritme vil bli!

Eksempel 1.1.2.

- $\Theta(log(n))$*: Denne kalles logaritmisk kjøretid, og er fin-fin! Dette skjer blant annet hvis du kan halvere problemstørrelsen din ved å teste ett element. F.eks kan du tenke deg at du vet at du har en* ***stigende tallfølge*** *og skal finne et bestemt tall. Da kan du sjekke det midterste. Hvis det er for lite, kan du se bort fra alle de tallene du vet er mindre. Dermed har du allerede omtrent halvert problemstørrelsen din! Om tallet du ser på først er større, ser du bort fra alle tallene som er større enn dette. Dersom tallet du leter etter eksisterer i tallfølgen, finner du det fort. Og hvorfor er kjøretiden* $\Theta(log(n))$ *mon tro? Jo,* $log(2n) = log(2) + log(n)$*, som gir at en dobling i problemstørrelse kun gir et konstant tillegg til kjøretiden.*

- $\Theta(n)$*: Dette er en polynomisk kjøretid. Se eksempel 1.0.1. Hvis du har en input på størrelse n, og er nødt til å gå gjennom alle tallene én, men bare én gang, har vi enkelt og greit* $\Theta(n)$*!*

- $\Theta(n^2)$*: Nok et eksempel på en polynomisk kjøretid. Gitt at du har en input på n. Det forekommer ofte at man må lete gjennom en matrise som har n rader og n kolonner. I dette tilfelle vil en dobling av input gi fire ganger så mange elementer å lete igjennom!*

- $\Theta(2^n)$*: Her er vi inne på eksponentiell kjøretid. Denne er ikke morsom! Bare ett lite tillegg på input fra f.eks en million til en million og én vil øke kjøretiden med det dobbelte. Dette kan forekomme ved at du for hvert element du tester, springer det ut to nye valg som du begge må teste. Dette er lettere å skjønne når du lærer om trær (se eget kapittel 4).*

❑

1.1.2 O-notasjon

Til forskjell fra Θ -definisjonen vil O-notasjonen ta for seg den øvre begrensningen til funksjonen:

Definisjon. $O(g(n))$ = { $f(n)$: slik at det finnes positive konstanter c og n_0 så vi har $0 \leqslant f(n) \leqslant cg(n)$ for alle $n_0 \leqslant n$ }

Dette er ikke så ulikt Θ -notasjonen. $2n^2 + 100n$ er både $\Theta(n^2)$ og $O(n^2)$. Men den er også $O(n^3)$, $O(n^4)$, $O(2^n)$ og alt som værre er! For hvis en eller annen konstant ganget med n^2 alltid vil være større enn kjøretiden, så gjelder det også alle andre funksjoner som vokser raskere enn det igjen. Altså, hvis du vet at kjøretiden til en av algoritmene dine er $O(n^2)$ så vet du at den vil være maksimalt $\Theta(n^2)$, men den kan også være $\Theta(n \cdot log(n))$ eller $\Theta(n)$ eller $\Theta(log(n))$. Derimot kan den ikke være værre, f.eks $\Theta(n^3)$.

1.1.3 Ω-notasjon

Der O-notasjonen ga en øvre begrensning, gir Ω -notasjon den nedre begrensingen. Derfor brukes ikke Ω så veldig ofte, for den vil aldri kunne si hvor bra en algoritme er, bare hvor dårlig den er! Hvis du f.eks viser at en algoritme har en kjøretid på Ω (n), så kan det godt hende at den egentlige kjøretiden er Θ (2^n), som er fryktelig dårlig. Den formelle definisjonen er som følger.

Definisjon. $\Omega(g(n))$ = { $f(n)$: slik at det finnes positive konstanter c og n_0 så vi har $0 \leqslant cg(n) \leqslant f(n)$ for alle $n_0 \leqslant n$ }

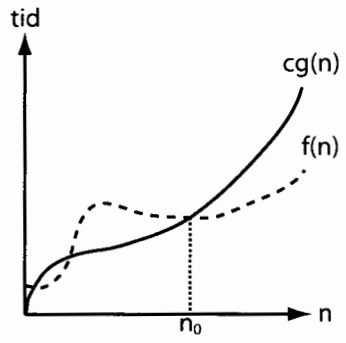
1.1.4 Bruken av Θ , O og Ω -notasjon

Disse begrepene brukes hele tiden, og det er derfor viktig å kjenne til dem. Sett deg derfor godt inn i begrepene! Merk at av og til brukes Θ og O-notasjonen om hverandre. Men hvis du har at en algoritme har en kjøretid på $O(n^2)$ mens den kanskje er $\Theta(n)$ så er det jo riktig, selv om det jo er kjekt å vite at den er enda raskere enn det $\Theta(n^2)$ er! Studér figur 1.2, 1.3 og 1.4 for å få en god følelse av hva disse uttrykkene virkelig betyr.

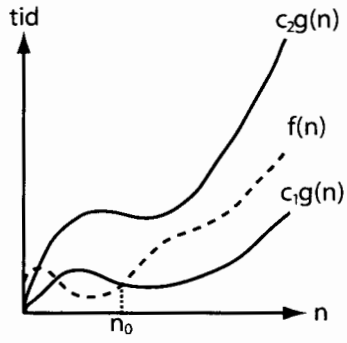
I praksis brukes Θ-notasjon når det er et fast antall elementer du må undersøke hver gang. Dvs. det er ikke snakk om å ha flaks eller uflaks med tallene. I eksempel 1.0.1 f.eks. må vi gå gjennom alle tallene uansett rekkefølgen de står i.

-notasjon brukes mest hensiktsmessig for å vise verste tilfelle, mens Ω brukes for å illustrere beste tilfelle. Legg merke til at når vi veit hvor mange elementer vi må undersøke, som i eksempel 1.0.1, er disse like, og vi bruker Θ-notasjon.

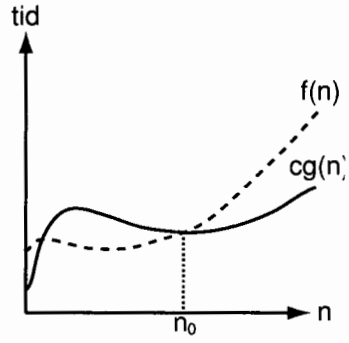
Noen ganger snakker vi om "worst case", "average case" og "best case" og bruker denne notasjonen i disse forskjellige tilfellene. Da må du ikke la deg lure til å tro at notasjonen gjelder kjøretiden til algoritmen generelt.



Figur 1.2: $f(n) = O(g(n))$



Figur 1.3: $f(n) = \Theta(g(n))$



Figur 1.4: $f(n) = \Omega(g(n))$

1.2 Kjøretiden til rekurensligninger

En rekurensligning er en ligning som beskriver en funksjon ved dennes verdi for mindre inn-verdier. Følgende eksempel beskriver godt en typisk rekurensligning.

Eksempel 1.2.1. Fibonacci-tallene
Fibonacci-tallene kan defineres som:

$$f(0) = f(1) = 1$$

$$f(n) = f(n-1) + f(n-2)$$

For å finne det n-te Fibonacci-tallet må man altså kjenne de to foregående tallene osv. De fem første Fibonacci-tallene blir for øvrig 1 1 2 3 5.

Dette er en typisk rekurensligning som ofte forekommer i eksempler (men det skal nevnes at det er mulig å finne et direkte uttrykk for det n-te Fibonacci-tallet. Dog vil jo da eksemplet miste all sin sjarm og pedagogiske misjon)

❑

Men selv om Fibonacci-tallene kan løses som rekurensligning, vil dette bli en eksponentiell kjøretid! Heldigvis finnes det langt smartere måter å løse denne på enn ved ren rekursjon. Det kommer vi tilbake til senere.

Det er i hovedsak tre metoder som brukes for finne kjøretiden til rekurensligninger. De er:

1.2.1 Substitusjonsmetoden

Denne metoden ordner biffen i to steg:
1. Gjett på en løsning
2. Bruk **matematisk induksjon** til å verifisere eller forkaste den mulige løsningen.

Det kan virke vanskelig å gjette på riktig løsning, men dette er faktisk ikke så veldig stort problem. Med litt trening ser man sånn noenlunde hvor den vil ligge hen. Og mulighetene er egentlig ikke så veldig mange. Det blir som regel Θ av n, n^2 eller n^3, eventuelt multiplisert med $log(n)$.

Eksempel 1.2.2.
Anta at rekurensen er beskrevet ved:

$$T(n) = 2T(\lfloor n/2 \rfloor) + \frac{2}{3}n, \quad \lfloor ... \rfloor \textit{betyr å runde ned til nærmeste heltall}$$

Vi kan da gjette på at løsningen er $O(n \cdot log(n))$. Det vi da må vise, er at $T(n) \leq c \cdot n \cdot log(n)$ for en eller annen konstant $c > 0$. Ved å sette inn dette forslaget, vil man finne ut at for $c \geq 1$ fungerer det. Se [1] side 64. Hadde vi mislykkes, måtte vi ha prøvd med en dårligere kjøretid. Og dessuten må man finne ut om det ikke finnes enda bedre kjøretider. Finn noen eksempler og regn dem ut!

❑

1.2.2 Rekurenstre

Om du fortsatt ikke liker tanken på å gjette i vildens sky, kommer rekurenstre-metoden som kallet. I rekurenstreet vil hver node representere kostnaden av et enkelt underproblem. Vi summerer kostnadene på hvert nivå av treet for å finne kostnaden per nivå, og deretter summerer vi alle kostnadene per nivå for å finne total kostnad til alle nivåene. Dette høres kanskje komplisert ut men er ganske greit. Lager man treet svært nøye, kan det holde som et bevis i seg selv. Eller man kan finne noen tall og så se hva slags **asymptotisk oppførsel** de har, for deretter å verifisere gjettingen ved substitusjonsmetoden.

Eksempel 1.2.3.
Anta at du har en tallfølge som representerer kjøretiden til en algoritme for hver gang inn-verdien øker med en enhet. Den er gitt ved:

$$1, 3, 7, 15, 31$$

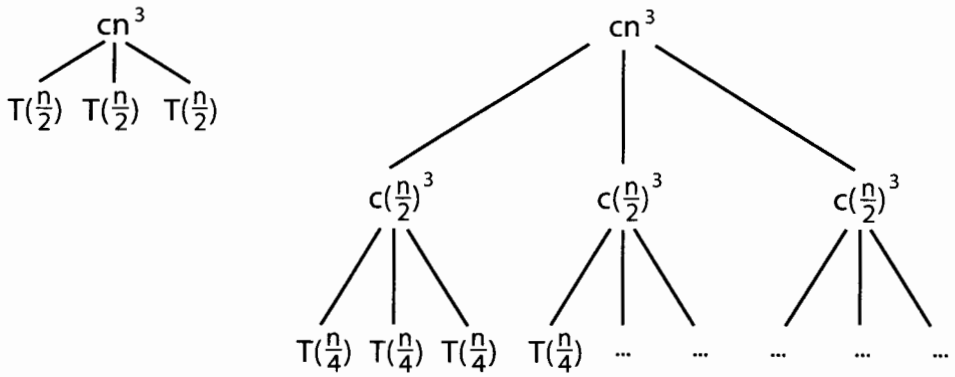
Vi ser at hvis vi legger til tallet 1 i hver del her, så får vi:

$$2, 4, 8, 16, 32$$

og dette er eksponentielt økende! Vi kan dermed konkludere med at kjøretiden er $\Theta(2^n)$ (men dette må jo sjekkes, så klart!)

❑

Eksempel 1.2.4.
Rekurenstreet til $T(n) = 3T(n/2) + cn^3$ er vist i figur 1.5.



Figur 1.5: Til venstre, 1.steg i treet. Til høyre, 2.steg i treet.

❑

Se ellers side 69 i [1] for en bedre oversikt!

1.2.3 Master-metoden

Selve kokebokmetoden! Hvis du har en rekurensligning på formen:

$$T(n) = aT(n/b) + f(n)$$

hvor $a \geq 1$ og $b > 1$ er konstanter og $f(n)$ er en asymptotisk, positiv funksjon, så er det bare å bruke Masterteoremet rett fram. Den står på side 73 i [1]. I korte trekk sier den at om du lar n/b bety enten $\lfloor n/b \rfloor$ eller $\lceil n/b \rceil$ så har du:

1. Hvis $f(n) = O(n^{log_b a - \epsilon})$ for en konstant $\epsilon > 0$, $\Rightarrow T(n) = \Theta(n^{log_b a})$
2. Hvis $f(n) = \Theta(n^{log_b a})$, $\Rightarrow T(n) = \Theta(n^{log_b a} \cdot log(n))$
3. Hvis $f(n) = \Omega(n^{log_b a + \epsilon})$ for en konstant $\epsilon > 0$ og hvis $af(n/b) \leq cf(n)$ for en konstant $c < 1$ og en n stor nok , $\Rightarrow T(n) = \Theta(f(n))$

Dette kan se litt kinkig ut, men er egentlig ganske greit. Hold tunga rett i munnen!

Eksempel 1.2.5.
Anta at du har:

$$T(n) = 9T(n/3) + n$$

Her er $a = 9$, $b = 3$ *og* $f(n) = n$. *Vi ser at*

$$n^{log_b a} = n^{log_3 9} = n^2 = \Theta(n^2)$$

Siden $f(n) = O(n^{log_3 9 - \epsilon})$ *med* $\epsilon = 1$, *så gir første del av Masterteoremet at* $T(n) = \Theta(n^2)$.

❑

Vi vil anbefale å trene seg litt i denne metoden. I [1] står det eksempler på alle tilfellene fra Masterteoremet.

Kapittel 2

SORTERING

Sortering er en av de viktigste operasjonene man har. Og det brukes også mye som en del av andre operasjoner. Tenk for eksempel hvis du spiller kortspillet bridge. Da får du utdelt 13 kort, og for å ha oversikt over hvor bra hånden din er sorterer du den etter farge og kortenes valør innad i hver farge.

I dataprogrammering er det ikke vanskelig å forestille seg at små sorteringer ofte vil ligge i bakgrunnen for det man gjør. Når man oppretter en ny mappe, vil denne (ofte) settes på plass etter alfabetisk rekkefølge, altså må det ha skjedd en sortering! Om du ikke er overbevist om viktigheten av sortering allerede, bør det også nevnes at dette er svært eksamensrelevant. I kapitlet vil vi anta at i sortering med tall vil vi sortere fra minste til største, mens sortering med bokstaver vil gå etter standard, alfabetisk rekkefølge. Vi skriver ikke pseudokodene her i kompendiet, men vi anbefaler at man går igjennom dem. Her følger mer en oversikt over hva de forskjellige sorteringene gjør. Se på detaljene i boka. La oss gå igang!

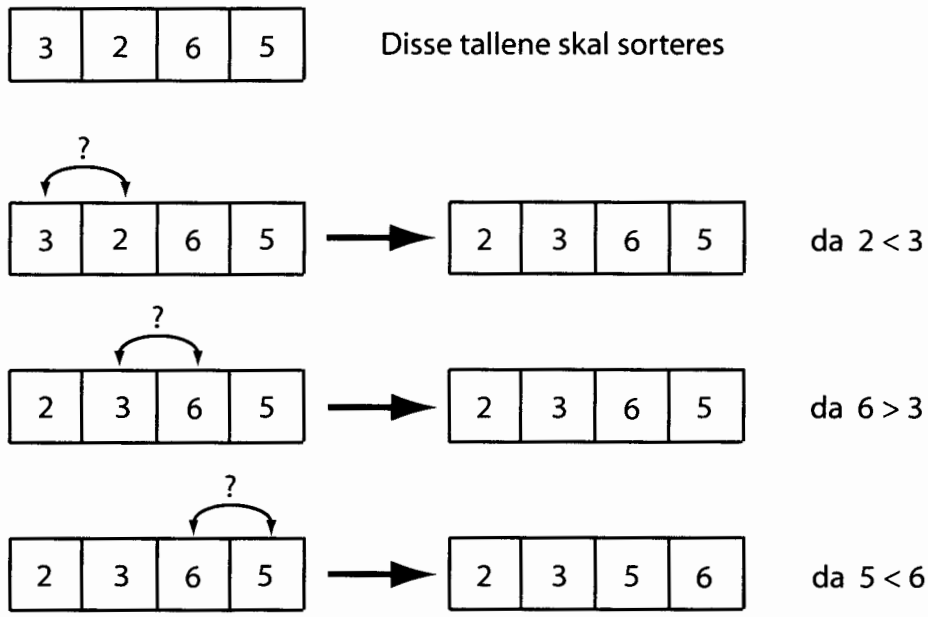
2.1 Sortering ved sammenligning

Dette er den ene av to grupper sorteringer. Felles i denne gruppen er at sorteringen skjer ved sammenligning av to og to elementer. De fleste av disse metodene trenger ikke bruke mer plass enn antallet elementer som kommer i input, og derfor sies disse å være sorteringer "in place".

2.1.1 Bubble Sort (Boblesortering)

Dette kan sies å være den enkleste og mest "naive" av alle sorteringer. Den tester to og to naboelementer, og hvis den finner at det første elementet er større enn det neste, bare bytter den de to. Boblesortering foregår "in place".

Kjøretiden til Boblesortering er $\Theta(n^2)$ for både gjennomsnittlig- og værste tilfelle. Det er



Figur 2.1: Boblesortering in action!

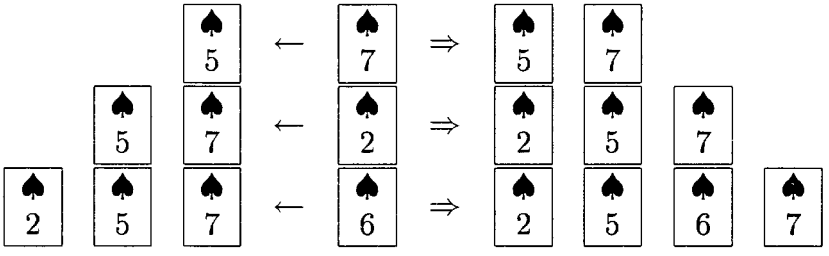
ikke så vanskelig å se for seg. Hvert eneste element kan risikere å måtte flytte seg opptil n plasser under sorteringen. Boblesortering er nok den minst effektive metoden dere kommer innom, men det er fin å kjenne til. Dessuten har den et lite konstantledd på hver operasjon som gjør at den er effektiv på veldig små mengder.

2.1.2 Insertion Sort (Sortering ved Innsetting)

Denne sorteringen er nok den mest intuitive sorteringene av dem alle, og foregår også "in place". Den tar utgangspunkt i et element, og sjekker så et nytt element. Deretter settes de slik at de er sortert i forhold til hverandre. Så tar den et element til og ser hvor dette skal plasseres i forhold til de som er der. Slik fortsetter det. Når n-1 elementer er sortert ferdig og det n'te kommer, finner det lett sin plass i rekken. Dette er den måten de aller fleste sorterer en korthånd på. Denner metoden er også meget effektiv på små datamengder. Dette gjør den også fin å kombinere med andre metoder som vi kommer til senere.

Kjøretiden her er også $\Theta(n^2)$ både for gjennomsnittelig kjøretid og i værste tilfelle. Men det er i praksis noe bedre enn boblesortering. Et veldig viktig poeng er: hvis du har en hel del elementer som nesten er sortert ferdig, er denne metoden helt suveren! Sett at du har en ganske stor korthånd og så får et kort til. Da vil denne metoden bare sette inn kortet på riktig plass! De andre metodene (som du selvsagt aldri ville brukt i praksis i dette tilfelle) ville da ha stykket opp de allerede ferdig sorterte kortene, og du ville ikke ha kunnet utnytte det du hadde. Dette kan for eksempel brukes i en stor database over adresser i tilfeller der det av og til vil skje små forandringer. Dessuten er denne også bra på veldig små mengder input.

Figur 2.2: Sortering ved innsetting



2.1.3 Merge Sort (Flettesortering)

Nå kommer en sortering som er bedre enn de to foregående (så sant størrelsen på inn-data ikke er meget liten)! I flettesortering deles problemet opp i stadig mindre biter, og når bitene er tilstrekkelig små, er det lett å sortere dem hver for seg.

Tenk for eksempel at du deler input-verdiene i to like deler (om du har et oddetall av elementer, vil den ene delen bare bli en større enn den andre). Deretter fortsetter du, med hver del for seg, til du har kun to eller tre verdier i hver lille del. Disse er lette å sortere (f.eks ved å bruke metoden til boblesortering!) hver for seg. Når dette er gjort, tar man for seg to og to deler. Disse flettes sammen ganske enkelt ved å se på den minste verdien i hver av de to nye delene/bunkene, og skille ut den minste av dem. Så gjentar man prosedyren med de elementene som er igjen og ender med en større del sorterte elementer. Slik fortsetter det til alt er sortert.

Altså, man splitter opp problemet, ordner og fikser litt, og samler sammen alt igjen. Virket det vanskelig? Se på eksempel 2.1.1!

Eksempel 2.1.1. Å sortere en tallfølge med flettesortering.
Anta at du har tallene 8 3 9 13 12 2 14 10 5

1. steg: *Del tallene rundt midten:* *8 3 9 13 / 12 2 14 10 5*
2. steg: *Fortsett i samme dur:* *8 3 / 9 13 / 12 2 / 14 10 5*
3. steg: *Nå sorterer vi hver lille delmengde og får:* *3 8 / 9 13 / 2 12 / 5 10 14*
4. steg: *Fletter så sammen to og to delmengder:* *3 8 9 13 / 2 5 10 12 14*
5. steg: *Etter siste fletting er alle tallene sortert!* *2 3 5 8 9 10 12 13 14*

Bingo!

❑

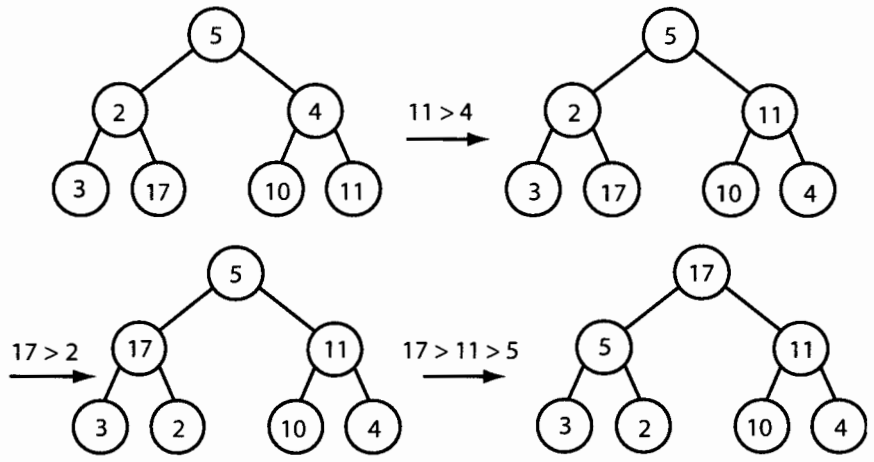
Kjøretiden til flettesortering er $\Theta(n \cdot log(n))$ i gjennomsnittelig og værste kjøretid. Hvor kommer logaritmen inn i bildet? Jo, den kommer inn når du deler opp problemet. Vi

forklarer dette intuitivt. For et dobbelt så stort antall input-elementer trenger du kun dele en eneste gang i to deler, slik at de to delene hver for seg tar like lang tid som for den opprinnelige mengden av problemer. Slik vil du kun trenge en ekstra deling av problemet når det blir dobbelt så stort! (Husk fra matematikken at $log(2n) = log(2) + log(n) = 1 + log(n)$ når log er 2-logaritmen). Multiplikasjonen med n kommer av at når du skal flette sammen de to siste delene, må du sjekke alle elementene en gang mot et annet.

2.1.4 Heap Sort (Haugsortering)

Her benytter vi oss igjen av deling av problemet slik at vi oppnår en god kjøretid. Heapsort er nok et eksempel på en sortering "in place".

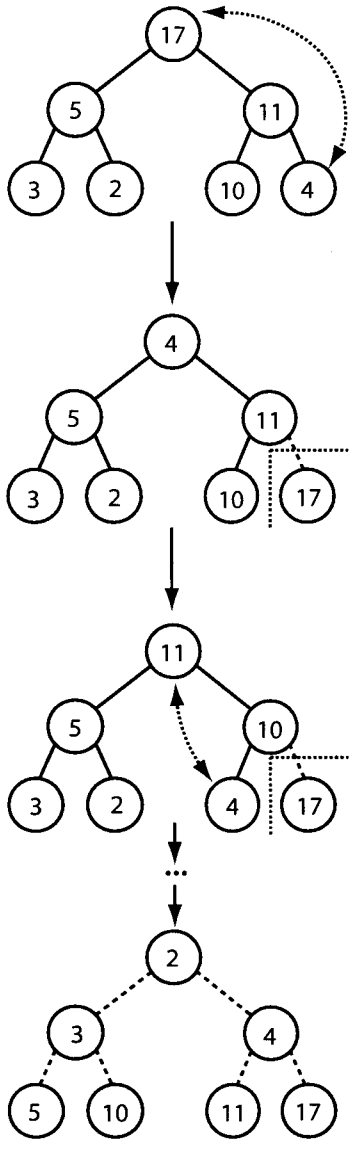
1.steg: Heapify
Dette består i å lage en slags "haug" av elementene først. En haug (eller en "heap") er et eksempel på en datastruktur. Det foregår slik at du setter elementene i et **tre** slik at ethvert element er det **største av alle som er under det**. Se på figur 2.3. Dette innebærer blant annet at det øverste elementet er det største av alle de som er der.



Figur 2.3: Heapify

2.steg: Sorter
Når haugen først er ferdig, går det fort å sortere. Bare plukk ut det øverste elementet i haugen, som er det største av samtlige. Deretter går det fort å ordne de resterende elementene i haugen igjen for deretter å plukke ut det største av dem igjen.

Kjøretiden her er også $\Theta(n \cdot log(n))$. Det som er så fint med denne metoden, er prinsippet



Dette er vår haug. Siden 17 er på toppen, veit vi at dette er det største tallet.

Vi lar 17 bytte plass med løvnoden lengst til høyre på det nederste nivået.

17 er nå sortert, og vi kjører heapify på resten av treet.

11 er nå det høyeste tallet. Vi lar 11 bytte plass med 4, løvnoden legst til høyre på det nederste nivået. Deretter kjøres heapify.

Prosedyren med å bytte høyeste tall med løvnoden lengst til venstre i det nederste nivået og kjør heapify uten dette tallet, gjentas helt til alle nodene er sortert.

Den sorterte rekkefølgen får vi ved å lese treet nivå for nivå, fra venstre mot høyre,Som da vi startet å bygge treet.

Figur 2.4: Sorteringen av haugen i Heapsort

med haugen. Dette kommer frem om du har med en **prioritetskø** å gjøre. Dette vil vi dog ikke ta med i kompendiet.

2.1.5 QuickSort

Quiksort er en av de desidert mest populære sorteringene i bruk. Den bruker, som flette-sortering, en form for splitt-og-herskmetode. Denne algoritmen kan deles i tre trinn:

Splitt:
Velg ut et element i mengden du vil sortere. La oss kalle dette **pivotelementet** for a. Så deler vi mengden i to mindre mengder slik at **alle elementer mindre eller lik** a danner den ene mengden, mens resten (som da er større enn a) danner den andre mengden.

Eksempel 2.1.2. Å splitte en mengde
Anta at du har tallene 2 8 7 1 3 6 4
Vi velger nå pivotelementet a til å være det siste tallet 4. Tall som tilhører 1.partisjon, altså som er mindre eller lik a, understreker vi. De tallene som hører til 2.partisjon, setter vi i ***fet skrift****. Algoritmen for splitting blir som følger:*

- *2* < 4, *2 tilhører 1.partisjon:* 2 8 7 1 3 6 4
- *8* > 4, *8 tilhører 2.partisjon:* 2 **8** 7 1 3 6 4
- *7* > 4, *7 tilhører 2.partisjon:* 2 **8** **7** 1 3 6 4
- *1* < 4, *1 tilhører 1.partisjon:* 2 1 **7** **8** 3 6 4; *1 bytter med 8*
- *3* < 4, *3 tilhører 1.partisjon:* 2 1 3 **8** **7** 6 4; *3 bytter med 7*
- *6* > 4, *6 tilhører 2.partisjon:* 2 1 3 **8** **7** **6** 4

Pivotelementet finner sin plass (4 bytter med 8), så partisjonen gir:

2 1 3 4 7 6 8

❑

Det er for så vidt andre måter å splitte på enn den i eksemplet! Dere har kanskje sett en metode i forelesning hvor man søker fra begge sider og bytter om de par hvor det til venstre er større enn pivotelementet og det til høyre er mindre eller lik. Denne siste fremgangsmåten er nok noe lettere å implementere. Husk: Det er flere veier til Rom!

Hersk:
De to delmengdene sorteres ved å bruke quicksort igjen på hver av de to delene. Det er dette som kalles å gjøre et **rekursivt kall**.

Kombiner: Sett sammen delmengdene når du er ferdig med alle rekursive kall! Dette er lett, da Quicksort sorterer "in place".

Dette kan virke greit, men to spørsmål melder seg. Hvordan velge a, og når skal man avslutte de rekursive kallene?

For å velge a, kan man ganske enkelt bare bruke den første verdien man har tilgjengelig. F.eks om man har en liste med tall, bruker man det øverste tallet. Dette kan gå fint, men om tallmengden nesten er ferdig sortert vil denne metoden få en veldig stor kjøretid. Dette kommer vil tilbake til når vi diskuterer kjøretiden for denne metoden.

Det er også et poeng at man ikke trenger å splitte helt til alle delmengdene bare består av ett enkelt element. På små datamengder er ikke Quicksort så veldig effektiv. Derfor er det ofte slik at når en delmengde har under et visst antall element, så sorterer man delmengden med en annen metode. Det er vanlig å sette grensen litt under 10 og så bruke sortering ved innsetting.

Hva så med **kjøretiden** til Quicksort? Gitt av vi klarer å velge elementet a slik at den deler mengden i to nesten helt like store deler. Ved samme argument som for kjøretiden til flettesortering, så vil Quicksort har kjøretid $\Theta(n \cdot log(n))$. Å få valgt verdien a til å tilfredstille dette er derimot nesten umulig om man ikke har god kunnskap om mengden. Men det viser seg at om man kan velge a helt **tilfeldig**, vil den gjennomsnittelige kjøretiden likevel bli $\Theta(n * log(n))$. Hvis alle elementene ligger helt tilfeldig på forhånd, så vil metoden for å velge a beskrevet tidligere være det samme som å plukke helt tilfeldige verdier, og vi er i mål. Men hva om mengden vår er nesten ferdig sortert allerede? Da vil den ene delmengden bare bestå av ett element, nemlig a selv, mens alle andre element havner i den andre delmengden. Hvis vi har n elementer i den opprinnelige mengden vår, så vil vi nesten måtte foreta n delinger og kjøretiden blir $\Theta(n^2)$. For Quicksort sier vi at **gjennomsnittelig kjøretid** er $\Theta(n \cdot log(n))$, mens den i verste tilfelle er $\Theta(n^2)$. Derfor er det viktig å få randomisert når vi skal plukke ut vår a. Det kan også nevnes at Quicksort (med god randomisering) er raskere enn flettesortering og haugsortering selv om de to sistnevnte har både gjennomsnittelig og en verste kjøretid på $\Theta(n * log(n))$. Dette skjer fordi konstantleddet er veldig lite i Quicksort.

2.1.6 Hvor fort kan det gå?

Det kan være naturlig å spørre seg hvor kjapt det er mulig å sortere ved sammenligning. Ved å se på side 167 i [1], så står der et flott teorem som sier:

Teorem. *Enhver sortering ved sammenligning krever* $\Omega(n*log(n))$ *antall sammenligninger i verste tilfelle.*

Dette teoremet beviser lærebokforfatterne også. Dermed har vi vist at haugsortering og flettesortering er optimale sammenligningssorteringer (dette gjelder **ikke** Quicksort fordi den i verste tilfelle vil ligge på $\Theta(n^2)$).

2.2 Sortering i lineær tid

Hvis man kjenner til en del egenskaper ved mengden man skal sortere på forhånd, så kan man senke kjøretiden betraktelig!

2.2.1 Counting Sort (Tellesortering)

Anta at vi har en mengde med n heltall med verdier fra 0 til k. Da kan vi rett og slett lage oss en liste fra 0 til k, og så setter vi hvert element inn på sin plass i listen. Og voilà, sorteringen faller ut av seg selv. Studér pseudokoden i læreboka [1] nøye.

Eksempel 2.2.1. Sortering av tall med tellesortering
Anta at vi har 13 tall mellom 1 og 9

3 9 8 9 4 1 5 4 7 8 2 4 7

Vi lager så en tabell hvor vi teller opp hvor mange vi har av hvert tall.

De mulige tall	1	2	3	4	5	6	7	8	9
Antall av hvert tall	1	1	1	3	1	0	2	2	2

Nå er det bare å hente ut den sorterte tallfølgen! Den blir da enkelt som følger:

1 2 3 4 4 4 5 7 7 8 8 9 9

❑

Kjøretiden til tellesortering avhenger av verdien k. Hvis den ikke øker for mye i forhold til størrelsen, nærmere bestemt $k = O(n)$, så blir kjøretiden $\Theta(k + n)$. Om du vet at alle (hel)tallene ligger i et bestemt område, f.eks mellom 1 og 10, så er det tellesortering som er tingen! For om k er konstant, så blir kjøretiden $\Theta(n)$. Hvert element sjekkes for sin verdi én gang og settes i sin bås. Nå skal man ikke la seg lure til å tro at man alltid kan bruke tellesortering. Har man ikke kontroll på verdien av k, er man på ville veier. Og tenk om man skal sortere tallene 2 , 1 og 1.000.000.000. Den vanligvis så trege boblesorteringen klarer dette på rekordtid, mens tellesorteringen må sette av en milliard plasser å plassere tallene i før de sorteres overhode. Det er ikke lønnsomt!

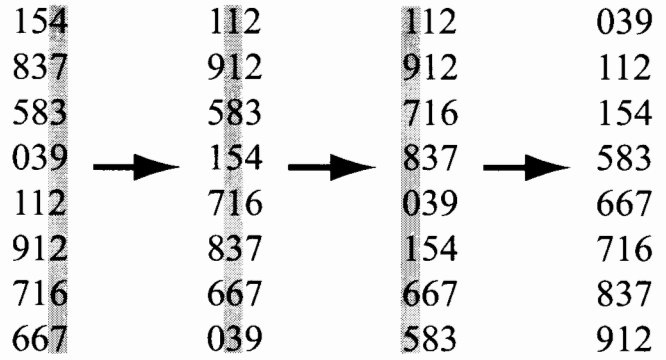
2.2.2 Radix Sort

Når man skal sortere en mengde heltall (f.eks i det vanlige titallsystemet) kan det være et smart triks å først sortere etter det mest signifikante sifferet, deretter det nest mest signifikante osv til man er i mål. Det som ikke er så gunstig da, er at for hver gang man gjør dette vil man ha 9 delmengder med tall som man ikke gjør noe med før den siste delmengden er ferdig sortert, og disse må man holde styr på.

Dette er ikke så veldig vanskelig, men radix-sortering bruker noe av samme idé som er enda smartere. Du vil se at hvis man derimot sorterer etter det **minst** signifikante sifret

i alle tall, vil man få sortert alle tallene uten det lille problemet som oppstår ved å ta fatt det mest signifikante sifferet. Det er kanskje ikke så intuitivt at denne metoden faktisk fører frem, men se på eksemplet under her. Ellers har radix-sorteringen den desidert korteste pseudokoden av alle, bare to linjer! Man kan bruke en hvilken som helst (sammenlignings)sortering i prosessen, og dette må gjentas like mange ganger som det er siffer i det største tallet.

Hvis ikke tallenes størrelse vokser over alle proposjoner, er kjøretiden til radix-sortering $\Theta(n)$.



Figur 2.5: Radixsort. Det er de tallene i grå felt man sorterer etter.

Legg merke til at det er et krav vi må ha på den midlertidige sorteringen vår. Hvis to tall er like, så må det tallet som ligger bak det andre også gjøre et etter sorteringen vår! Hvis ikke, blir det bare rot. Se på tallene 112 og 154 i figur 2.5. Etter andre sortering har vi funnet at 12 er mindre enn 54. Når så tallet 1 testes mot 1, så kan vi ikke tillate at deres rekkefølge etter sorteringen blir vilkårlig, for da risikerer vi å få at 154 er mindre enn 112!

2.2.3 Bucket Sort (Bøttesortering)

Den siste sorteringen vi tar for oss, er bøttesortering. Den er nesten helt lik tellesortering, men er forskjellig på ett vesentlig område. Der tellesortering krever heltall og lager en liste fra 0 til verdien av det største tallet, så kan bøttesortering ta for seg andre rasjonale tall også. Trikset er at man lager "bøtter", eller egentlig tallintervaller, i listen i stedet for bare enkelttall. F.eks kan man sette $[3, 4)$ til å være et intervall, og da vil alle tall som er større eller lik 3 men mindre enn 4 havne i dette intervallet. Når man har intervallene, sorterer man disse etter en kjent og kjær sammenligningssortering. Ellers er bøttesortering helt likt det vi kjenner fra tellesorteringen.

Det vises i [1] at forventet kjøretid for bøttesortering er $\Theta(n)$.

2.3 Finne i-te Verdi

Hvordan skal man gå frem hvis man i en tallmengde vil finne den i-te største (eller vice versa den minste) verdien? For å finne den største verdien av alle tallene kan man gå frem slik som i eksempel 1.0.1 i 1.kapittel. Vi viste der at man trenger nøyaktig n sammenligninger, så kjøretiden er $\Theta(n)$. Men å finne akkurat den i-te verdien krever mer kløkt. En mulighet er jo å bruke en av de sorteringene vi har gått igjennom og så bare plukke ut det tallet vi vil ha. Men trenger vi å sortere alle tallene våre når vi bare er på jakt etter dette ene? Anta at vi sorterer ved sammenligning. Vi har lært at beste kjøretid da er $\Theta(n \cdot log(n))$. En annen mulighet, er å finne største verdi for deretter å eliminere denne. Ved å fortsette slik finner man det nest største, tredje største osv helt til man kommer til det i-te største. Men om det i-te største ligger rundt midten, så er vi nødt til å finne største tall et antall ganger som er proposjonalt med antallet tall. Dermed blir kjøretiden i dette tilfelle $\Theta(n^2)$, som er dårligere enn det fikk i sted. Er det mulig å finne det i-te største elementet med en kjøretid på $\Theta(n)$? Vi ser lett at dette gjelder for det største elementet. Også for det i-te største er svaret ja!

Trikset er å bruke splitt-metoden fra **Quicksort**. Vi velger en tilfeldig verdi, kall den p, og lager to delmengder av den opprinnelige mengden vår. Dermed har vi at hvis posisjonen til p er mindre enn i, så holder det å lete i **øvre** halvdel, og er den mindre leter vi i **nedre** del. I Quicksort kjørte vi rekursive kall i begge delmengder. Her gjør vi et rekursivt kall bare i den ene delmengden, så kjøretiden for denne metoden er $\Theta(n)$. Det er ikke så vanskelig å se for seg om man tenker seg om litt. Vi henviser til læreboka for en mer detaljert beskrivelse av denne kjøretiden. Studér følgende eksempel.

Eksempel 2.3.1. Finn 3.minste verdi
Anta at vi har samme tallfølge som i eksempel 2.1.2, altså:

2 8 7 1 3 6 4

Vi ønsker å finne det tredje minste tallet. Med 4 som pivotelement har vi allerede at første splitt gir oss:

2 1 3 4 *7 6 8*

Alle verdier til venstre for 4 *er mindre eller lik, alle til høyre er større. Det vi nå vet, er at* 4 *er det fjerde minste elementet. Dermed kan vi se bort fra tallene 4, 7, 6 og 8, og vi står igjen med å finne det tredje minste tallet av:*

2 1 3

Velger vi så 2 *som pivotelement, så gir splitten oss:*

1 2 3

2 er det nest minste elementet, vi kan se bort fra det og tallet 1. Dermed er det bare tallet 3 igjen, og det er det tredje minste tallet i følgen vår!

❑

Ellers finnes det enda bedre metoder, men disse vil vi ikke gå inn på.

2.4 Hashing

Vanligvis brukes **direkte adressering** når man skal slå opp i en liste. Dette fungerer så lenge vi har råd til å ha en liste som har én posisjon for hver mulig verdi. Men dette kan gi mange ubrukte plasser! For å ikke sløse slik med plassen, benytter man seg av **hashing**. Dette er ikke så ulikt prinsippet for bøttesortering. Man kan tenke seg at en hashtabell består av en mengde bøtter, og i hver av bøttene er det rom for et par posisjoner fra listen. Slik vil mye plass kunne spares om man klarer å lage hashtabellen liten. Men det er en risiko for at det faller flere verdier i en og samme bøtte. Det er dette som kalles **kollisjoner**.

For å få bukt med kollisjoner, er det et godt triks å bruke **lenka liste**. Men man risikerer at noen av listene blir veldig lange, slik at det fungerer like bra å kun benytte seg av en lenket liste uten å drive med hashing i det hele tatt. Et annet gyllent triks er det som kalles **åpen adressering**. Enkelt sagt vil dette si at hvis man vil legge til et element og dennes plass er opptatt, så legger man rett og slett elementet inn i den første ledige plassen! Dessverre er det en god del problemer som kan oppstå, så dette er en ganske komplisert affære.

Et svært aktuelt spørsmål er hvordan man skal lage hashtabellen. Man snakker om en hash-funksjon h som tilordner enhver posisjon i listen til en posisjon i hashtabellen. Men denne må velges med kløkt, så elementene blir spredd mest mulig.

- **Divisjonsmetoden**

La k være nummeret til posisjonen i listen. Da kan man sette hashfunksjonen til å ta resten av k når det deles med tallet m. Altså:

$$h(k) = k \ mod \ m$$

hvor tallet m må velges med omhu. Et godt valg er å la m være et primtall og holde det langt unna en potens av tallet 2.

- **Multiplikasjonsmetoden**

Denne metoden går i to trinn. Først multipliserer vi k med et tall A som ligger mellom 0 og 1. Deretter ta desimalene (eller resten av divisjon med tallet 1!) og så gange med en verdi m og så runde av nedover. Matematisk blir dette:

$$h(k) = \lfloor m(k \cdot Amod1) \rfloor$$

Når det gjelder kjøretider til hashing, så går det stort sett fort. Å slå opp i en vanlig liste går på $O(1)$ tid. Hashing går som oftest mye raskere i praksis, selv om den også har $O(1)$ i gjennomsnittelig kjøretid. Konstantleddet er dog mye mindre i hashing! Men husk at hvis hashfunksjonen er dårlig stilt slik at veldig mange elementer havner på samme sted i hashtabellen (og vi fikser kollisjoner med en lenket liste) så vil hashingen oppføre seg nesten som en lenket liste og kjøretiden blir $O(n)$ i verste tilfelle. Likevel går det an å få bukt med dette problemet ved å benytte seg av såkalt **perfekt hashing**. Men akkurat det er hardcore type über, så det lar vi ligge...

2.5 Diverse

For å få en følelse med hvordan de forskjellige sorteringene fungerer, anbefaler vi å gå inn på http://www.cs.ubc.ca/spider/harrison/Java/sorting-demo.html. Her vil man se forløpet av sorteringene. Å se på dette og sammenligne med algoritmene er svært lærerikt!

Kapittel 3

LISTER, KØER og STAKKER

Dette kapittelet handler om tre enkle abstrakte datatyper (ADT), lenka lister, køer og stakker. Da dette bør være kjent stoff, har vi valgt å gjøre dette kapitlet veldig kort. For nærmere beskrivelse av hvordan disse kan implementeres se [4].

3.1 Liste

Vi har forskjellige varianter av lenka lister:

- enkle eller doble
- sortert eller ikke sortert
- sirkulær eller ikke sirkulær

De som kan litt enkel kombinatorikk, vet at dette blir 8 mulige kombinasjoner..

3.2 Kø

Hovedprinsipp: FIFO - "first-in, first-out"

En kø er akkurat det du tror det er, bortsett fra at her forekommer ingen sniking.

En kø kan implementeres som ei lenka liste eller som en array.

3.3 Stakk

Hovedprinsipp: LIFO - "last-in, first-out"

En stakk kan assosieres med en stabel tallerkner. Når en er nyvaska, legges den på toppen,

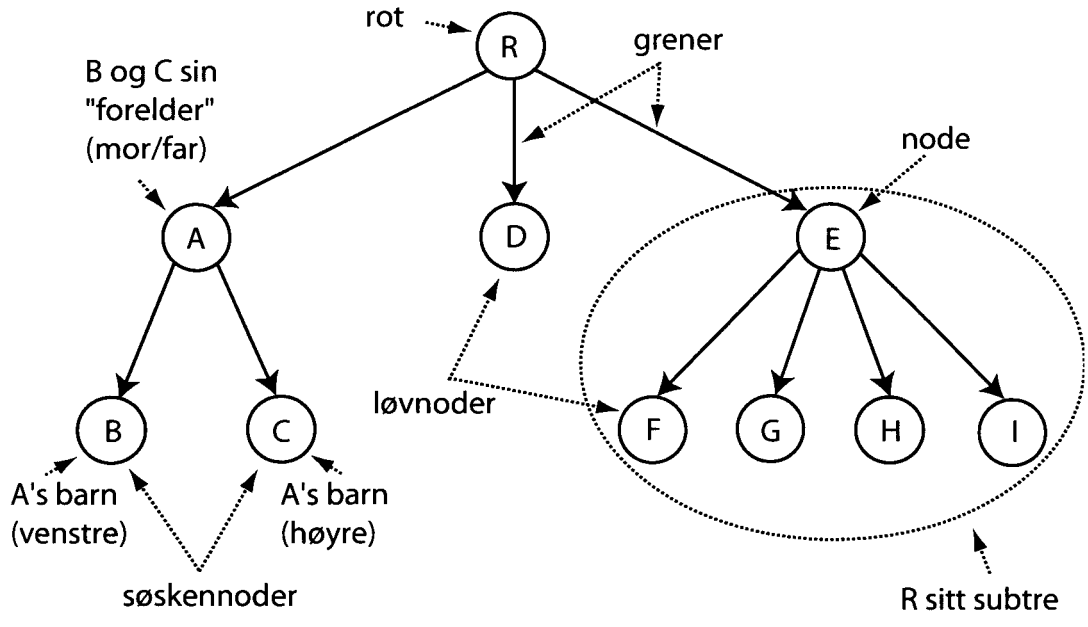
og når du trenger en, tar du den øverste i stabelen. Akkurat som vi ikke tar med snikere i vår kø, regner vi her ikke med de som stokker om på tallerkene i blant slik at ingen skal bli mer slitt enn andre.

Feil kan oppstå dersom du prøver å ta ut et element fra en tom stakk eller sette inn et element i en full stakk.

Stakker implementeres også vanligvis ved hjelp av en array eller ei lenka liste.

Kapittel 4

TRÆR



Figur 4.1: Figuren viser hvordan et tre er bygd opp og hva de forskjellige elementene er.

- **Nivå for node:** Det er antall grener som må passeres f.o.m. rot t.o.m. noden.
- **Nodegrad:** Det er antall barn, som er det samme som antall subtrær/undertrær, noden har.
- **Fritt tre:** Grenene har ikke retning, dvs alle nodene kan oppfattes som rotnode.
- **Rettet tre:** Grenene har retning (som på figur 4.1).
- **Ordnet tre:** Hvordan subtrærne/barna ordne i forhold til hverandre har betydning.

- **Trehøyden:** Maks antall grener som kan passeres f.o.m. rot t.o.m. løvnode.
- **k-grad-tre:** Et tre der hver node kan ha maks k barn. Posisjonen til barna er viktige.
- **komplett k-grad-tre:** Et k-grad-tre der alle nodene har k barn. Dvs et fullt k-grad-tre.
  Antall noder på dybde h er k^h. Trehøyden er $\log_k n$ der n er antall løvnoder.

4.1 Implementasjon

Hvordan vi implementerer en trestruktur avhenger av om vi har fast antall barn (f.eks. binære trær, 2-3 trær) eller variabelt antall barn (f.eks. B-trær og mer generelle trær).

For hver av variantene har vi tatt med kjøretiden til noen vanlige operasjoner:

- finne forelder (far/mor, heretter kalt far) til en node
- finne barn i til en node
- hente verdien til en node
- finne rotnoden

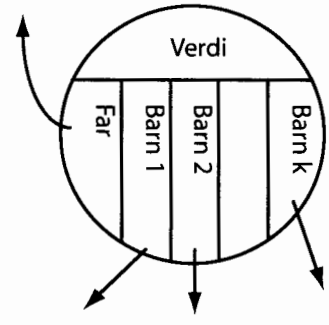
4.1.1 Generelle Trær med Fast Antall Barn

Hver node er et objekt med en verdi, en peker til hver av barna og en peker til far. Se figur 4.2.

Kjøretider:

- finne far:
- finne barn i:
- hente verdien:
- rot:

$\Big\}$ $= O(1)$



Figur 4.2: Slik ser en node ut.

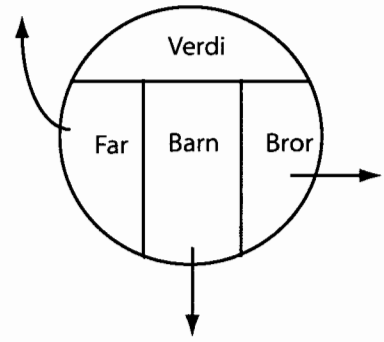
4.1.2 Generelle Trær med Variablet Antall Barn

Alternativ 1
Hver node er et objekt med en verdi, en peker til sitt første barn (lengst til venstre), en peker til far og en til sin nærmeste bror til høyre for seg. Se figur 4.3.

Kjøretider:

- finne far:
- finne barn i:
- hente verdien:
- rot:

$\Big\}$ $= O(1)$

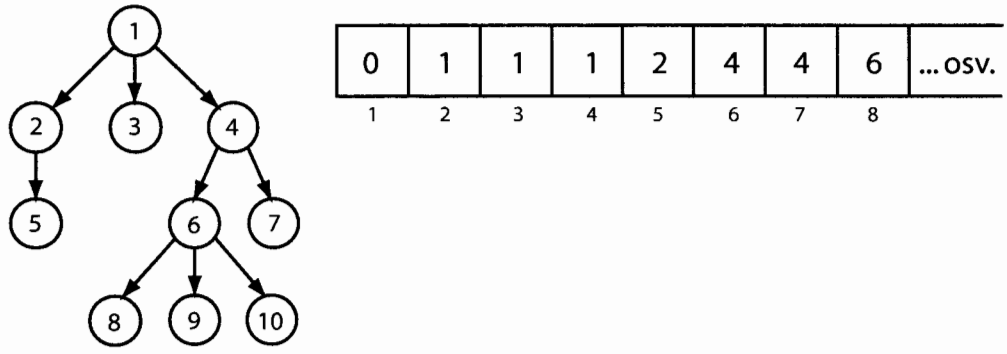


Figur 4.3: Slik ser en node ut.

(Vi antar et fast, gjennomsnittlig antall barn på hvert nivå.)

Alternativ 2

Her utnytter vi at alle noder i et tre har nøyaktig en far. Vi lager en array. På plass nr i står faren til node nummer i, eller en peker til denne. Det blir da lett å finne faren, men for å finne barna til node i, må vi søke gjennom arrayen etter tallet i.



Figur 4.4: Figuren viser et tre og den tilsvarende "fedrearrayen". Arrayen vil ha like mange elementer som det er noder.

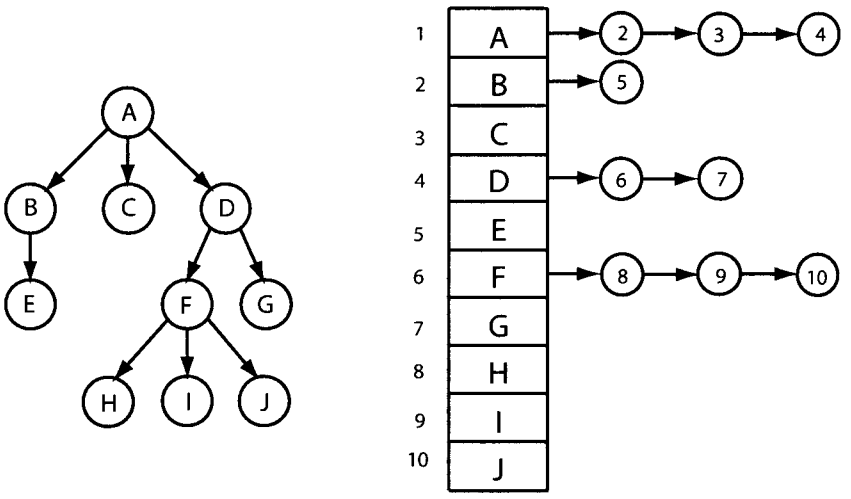
Kjøretider:

- finne far: $= O(1)$
- finne barn i: $= O(N)$
- hente verdien: $= O(1)$
- rot: $= O(1)$

(Vi har N elementer/noder i alt.)

Alternativ 3

Vi lagrer nodene i en array. Hvert element i arrayen inneholder et objekt med verdien til noden og pekere til ei lenka liste. Den lenka lista innholder pekere til barna.



Figur 4.5: Figuren illustrerer dette alternativet.

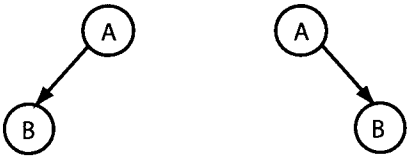
Kjøretider:

- finne far: $= O(h)$
- finne barn 1: $= O(1)$
- finne barn $i > 1$: $= O(h)$
- hente verdien: $= O(1)$
- rot: $= O(1)$

(Her er h dybden eller nivået.)

4.2 Binære Trær

Binære Trær er egentlig et 2-grad-tre, men dette navnet blir aldri brukt. Hver node har altså maks to barn og ordningen av høyre- og venstrebarn er viktig, se figur 4.6

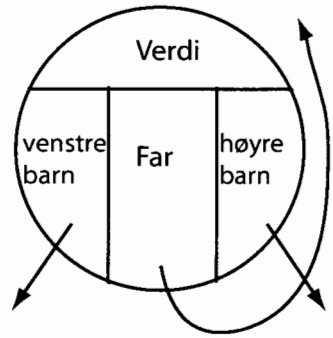


Figur 4.6: Disse to trærne er forskjellige!

Hver node er et objekt med en verdi, en peker til hver av barna og en peker til faren. Se figur 4.7

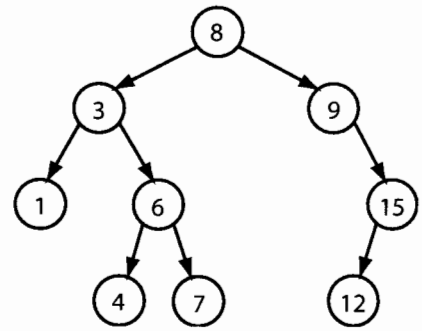
4.2.1 Binære Søketrær

Et binært søketre er bygget som et binært tre, men det er i tillegg organisert slik at verdien til venstre barn er mindre eller lik verdien til far mens verdien til høyre barn er større. Det



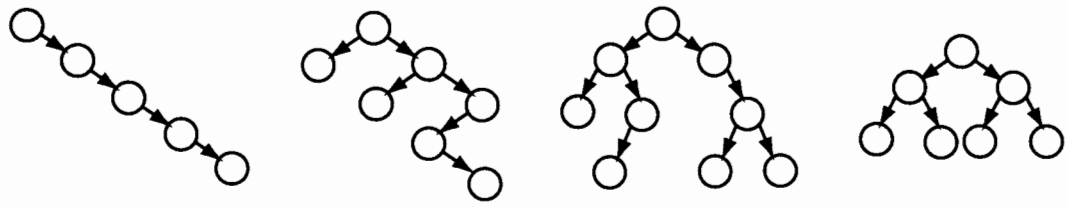
Figur 4.7: Figuren viser en node i et binærtre.

betyr at for alle node gjelder at alle verdiene til nodene i venstre subtre er lavere og alle i høyre er høyere enn verdien til noden selv. Se figur 4.8



Figur 4.8: Figuren viser et binært søketre.

Poenget er at med denne organiseringa er det vanligvis ganske kjapt å finne fram til ønsket node. Hvor lang tid det tar avhenger imidlertid av hvor heldig vi er med rekkefølgen nodene settes inn i. Figur 4.9 viser de mulige tilfellene. Hvor lang tid det tar å finne et element avhenger direkte av høyden til treet.

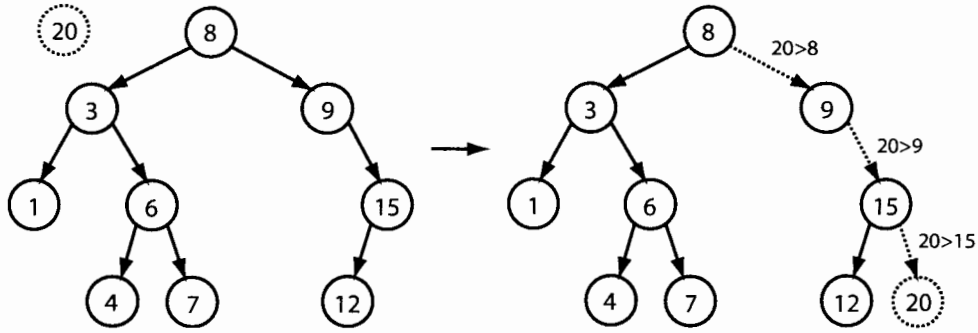


Figur 4.9:

Innsetting av Node

Innsetting av noder i et binært tre er enkelt. Det er bare å passe på å overholde kriteriene for et binærtre. Vi starter alltid ved roten. Dersom treet ikke har noen rot, dvs treet ikke

er påbegynt, settes vår node til å være rotnoden. Ellers sjekker vi for hver nye node vi kommer til om verdien til noden vi skal sette inn er større eller mindre enn verdien til denne noden. Dersom den er mindre, går vi til venstre, ellers går vi til høyre. Vår node kan settes inn på første tomme plass vi kommer til.



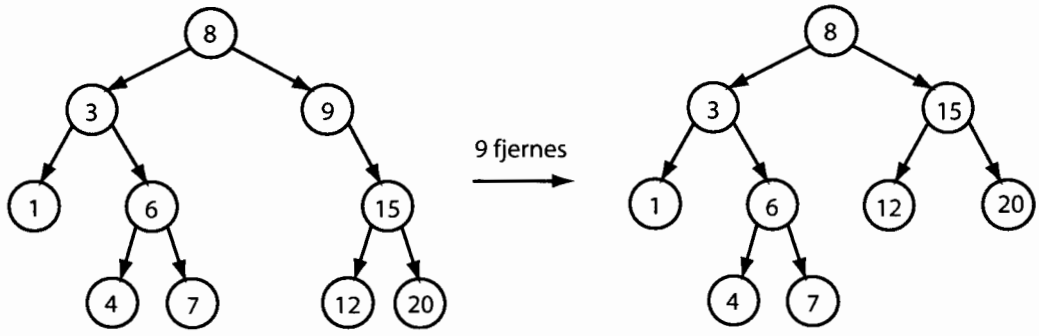
Figur 4.10: Figuren viser hvordan vi setter inne en node i et binærtre

Fjerning av Node

Vi har tre muligheter:

1. **Noden vi skal fjerne har ingen barn.**
   Vi kan da bare fjerne noden.

2. **Noden vi skal fjerne har bare et barn.**
   Vi kan da bare ta ut noden, og la barnet overta nodens far. Dette er illustrert i figur 4.11.



Figur 4.11: Figuren viser hvordan vi kan fjerne en node med et barn i et binærtre.

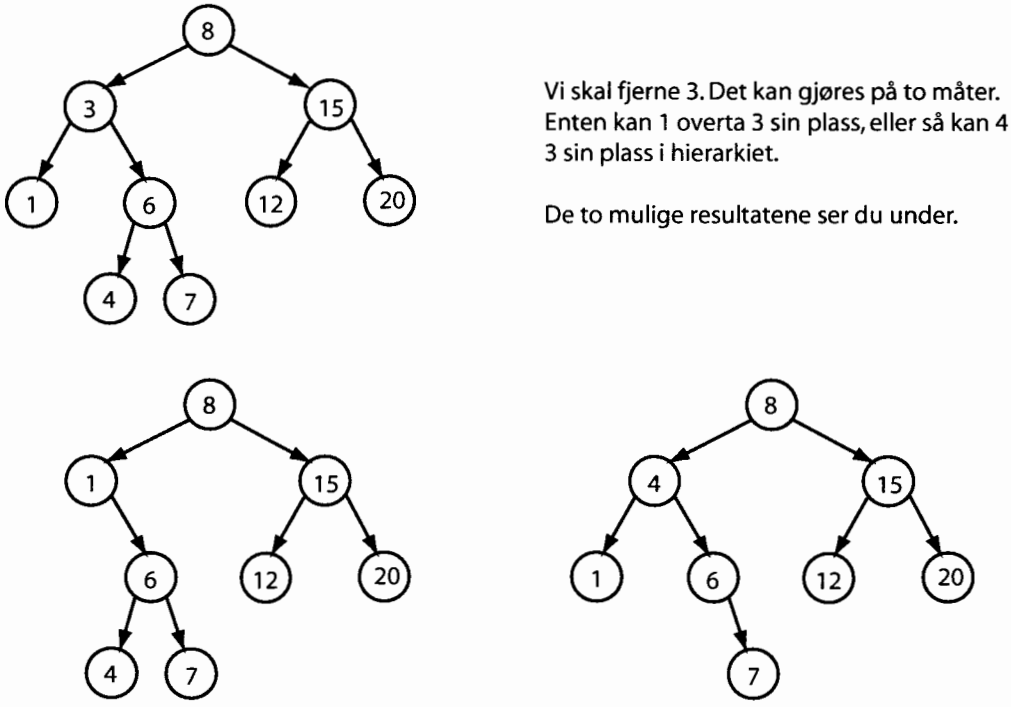
3. **Noden vi skal fjerne har to barn.** De to foregående tilfellene var rimelig enkle. Det er denne også, hvis man bare ser trikset. Spørsmålet vi må stille oss er: Hvilken annen node kan erstatte vår node, dvs ta dens plass i treet? Det må være en med verdi større eller lik alle verdier i venstre subtre eller en med verdi mindre enn alle i

høyre subtre.

En løsning er å velge den noden med størst verdi i venstre subtre. Den finner du ved å gå til venstre barn og holde til høyre i hvert kryss videre nedover. Når du kommer til løvnoden har du funnet den du leter etter.

En annen løsning er å velge den noden med minste verdi i høyre subtre. Helt tilsvarende går du da til høyre barn og holder til venstre i alle kryss helt til du når denne løvnoden.

En av disse nodene kan klippes av og settes inn i steden for den noden vi vil fjerne. Figur 4.12 viser et eksempel.



Figur 4.12: Figuren viser hvordan vi kan fjerne en node med to barn i et binærtre.

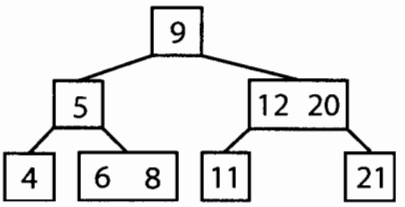
Traversering

Å traversere et tre eller en graf er et fint ord for å gå gjennom treet eller grafen. Her gir vi dere noen rekkursive forklaringer på noen systematiske måter dette kan gjøres på. Bruk figur 4.13 så forstår dere poenget!

- **Prefiks:** Først utføres operasjonen på rotnoden, deretter prefikstraverseres venstre subtre (hvis det eksisterer) og til slutt prefikstraverseres høyre subtre (hvis det eksisterer).

Det er altså et krav at hver node må være halvfull. I en annen variant av B-trær, B*-trær, er kravet at nodene må være minst 2/3 fulle.

- **Alle løvnodene ligger på samme nivå, som er høydens tre** $h \leq log_t \frac{n+1}{2}$.



Figur 4.14: Figuren viser et eksempel på et B-tre med $t = 2$.

I vanlige B-trær lagres det informasjon (satelittdata) til hver nøkkelverdi. Dvs alle nodene inneholder informasjon, enten lagret direkte med nøkkelverdiene, eller, mer vanlig, nøkklene peker til stedet på disken hvor denne informasjonen er lagret. Vi har også noe som kalles B^+-trær. Dette er en variant av B-trær som lagrer all informasjon i løvnodene. De øvrige nodene inneholder da bare nøkkler og pekere til barna.

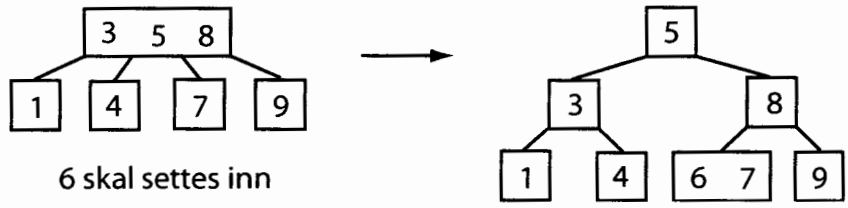
Vi kommer ikke til å snakke så mye om selve objektene eller pekerene til disse, men det ligger under at når vi flytter rundt på nøkler, er det egentlig disse objektene eller pekerene til disse vi reorganiserer, sletter eller setter inn.

4.3.1 Innsetting

Dette er litt mer komplisert enn innsetting i et binærtre da vi må passe på at treet fortsetter å være balansert. Vi kan altså ikke bare gå nedover i treet og hekte på en ny node nederst der den passer inn.

Løsningen er å sette inn den nye nøkkelen i en allerede eksisterende løvnode. Men vi kan ikke sette inn nye nøkler i allerede fulle noder. Disse må splittes. Den metoden læreboka, [1], bruker venter ikke og ser om splitting er helt nødvendig. I steden splitter den alle fulle noder den kommer til mens den vandrer nedover på vei mot rett plass i grafen. (Hvilken vei den velger nedover bestemmes ved å sammenligne nøkkelverdiene som skal settes inn med nøkkelverdiene nedover etter regelen i første punkt av definisjonen på B-trær.)

Splittingen foregår på denne måten: Vi finner medianen av nøkkelverdiene i den fulle noden og flytter det tilhørende objektet på rett plass hos nodens far. Den fulle noden vil da splittes i to nye, like store noder.
Legg merke til at eneste måten å få treet til å vokse i høyden er å splitte rota.



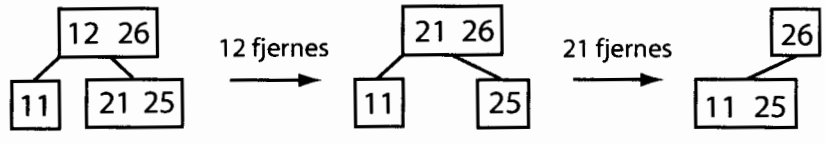
Figur 4.15: Figuren viser et eksempel på hvordan sette inne en node i et b-tre.

4.3.2 Fjerning

Å fjerne en nøkkel fra et B-tre er analogt med å sette inn en, men fjerning blir litt mer komplisert da vi kan slette nøkler fra alle noder, ikke bare løvnodene. Når vi sletter fra en indre node, må barna til noden reorganiseres.

Vi ønsker å fjerne nøkkel k fra node x. Vi skal se på de tilfellene du kan støte borti. Husk at hver node kan ha minst t-1 nøkler (tilsvarer t barn) og maks $2t$-1 nøkler (tilsvarer $2t$ barn).

1. Hvis x er en løvnode, kan vi bare slette k fra x.
2. Hvis x er en indre node, har vi tre muligheter:
   (a) Forgjengeren til k er den nøkkelen i et av x sine barn som har verdi nærmest k sin verdi men litt lavere. Hvis det barnet til x som inneholder k sin forgjenger, k', har minst t nøkler, kan k' slettes fra sin plass (rekursivt) og k' ta k sin plass. Bruk figur 4.16 til å forstå dette!
   (b) Helt tilsvarende kan du bytte ut k med nøkkelen med verdi litt større enn k sin verdi og slette denne rekursivt. Nøkkelen som menes er altså en som ligger i et av x sine barn, og ingen andre nøkler i noe av disse barna har en verdi som ligger mellom k og denne nøkkelens verdi. Dette kan bare gjøres dersom barnet har minst t nøkler, ellers kan vi risikere at barnet får nøkler enn det har lov til.
   (c) Hvis begge disse barna, både det som inneholder forgjengeren og det som inneholder etterfølger, har t-1 barn, må disse gå sammen til en node og k settes inn på rett plass (i midten) i denne nye noden. Vi prøver så igjen å slette k. Se figur 4.16



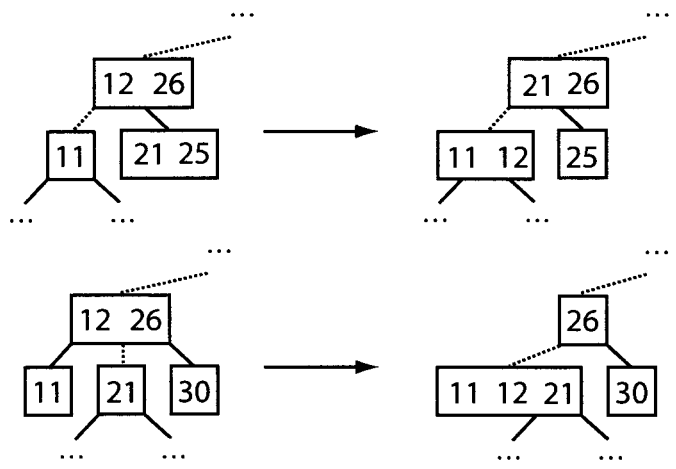
Figur 4.16: Figuren viser enkle eksempler av henholdsvis 2a og 2c.

Algoritmen vi bruker må også ha med noe som garanterer at kravet om minste antall barn alltid blir oppfylt. Dette kan gjøres på denne måte:

3. Hvis vi støter på en node med t-1 nøkler når vi går fra rota mot løvnodene på jakt etter x, utfører vi en av følgende operasjoner:

   (a) Hvis en av nodens nærmeste brødre har t eller flere nøkler kan vi gi vår node en ekstra nøkkel ved å flytte en fra faren ned til denne og en fra broren opp til faren. Merk at det er absolutt ikke tilfeldig hvem vi flytter. Studer figur 4.17.

   (b) Dersom begge brødrene til noden har t-1 nøkler, slår vi sammen noden og en av søsknene. Husk at en nøkkel fra faren også flyttes ned! Se figur 4.17.



Figur 4.17: Figuren viser 3a (øverst) og 3b (nederst). Den stiplede linjen viser veien vi har fulgt for å finne nøkkelen vi ønsker å slette.

Kapittel 5

GRAFER

Ofte har vi ikke bare et sett med elementer som skal behandles men også forbindelser mellom disse. Dette kan representeres ved hjelp av en graf.

En graf, G=(V,E), består av noder (V for vertices) og kanter (E for edges). Begge deler kan inneholde informasjon. For eksempel kan nodene være hus og kantene veier mellom husene. Veiene kan ha ulik lengde eller andre egenskaper som gjør at hvilken vei vi velger ikke er avhengig av hvor mange hus vi må innom. Vi sier at kantene har forskjellige kostnader. Dessuten kan veiene være enveiskjørte.

Noen spørsmål som kan dukke opp ved behandling av grafer:

- Vi har to hus. Finnes det en vei mellom de?
- Hvor mange hus kan vi nå fra et annet hus ved bare å følge veiene på lovlig vis?
- Hva er korteste vei?

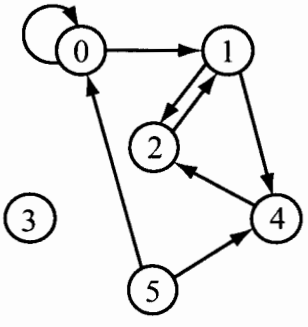
5.1 Noen Begreper

- **Naboer:** To noder (hus) er naboer dersom det går en kant i mellom dem.
- **Rettet:** Vi har en rettet graf dersom kantene har retning (symboliseres med piler).
- **Urettet:** En graf er urettet dersom kantene ikke har noen retning.
- **Sykel:** Vi har en sykel i en rettet graf dersom vi kan gå fra en node, via minst en annen og tilbake til den samme noden igjen. (Vi beveger oss bare i pilens retning.) En graf er **syklisk** dersom den inneholder sykler.
- **Vektet:** Vi har en vektet graf dersom kantene har forskjellige kostnader. Hva vi mener med kostnader, blir klarere utover i kompendiet.

- **K-fargbar:** En graf er k-fargbar dersom nodene kan farges i k forskjellige farger uten at to nabonoder må ha samme farge.
- **DAG (Directed Asyclic Graph):** En rettet graf som ikke inneholder noen sykler. Trær er DAGer. Disse kan sorteres topologisk.
- **Komplett:** En urettet graf er komplett dersom alle nodene er forbundet med hverandre.
- **Kritisk punkt/bro:** Kutting av denne kanten vil etterlate en ikke-sammenhengende graf.
- **Sti (path):** En sti er en vei gjennom grafen eller deler av den. En **enkel sti (simple path)** er en vei som er innom hver node maks en gang. To noder er forbundet (connected) hvis det går en sti mellom dem.
- **Hamilton-sti/sykel:** Nodene besøkes en og bare en gang langs en sammenhelgende vei gjennom grafen (langs kantene). Dvs vi har en enkel sykel gjennom alle nodene i grafen.
- **Euler-sti:** Kantene benyttes en og bare en gang når vi traverserer/går gjennom grafen (uten å hoppe). Euler viste at en slik sti bare finnes dersom vi har 0 eller 2 noder med odde antall naboer.

5.2 Implementasjon

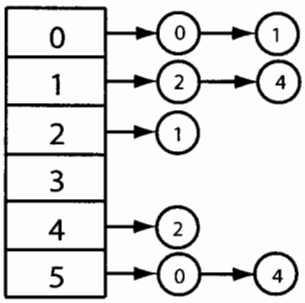
En graf kan implementeres på flere måter.Det vanligste er å implementere den som en **nabomatrise**.



fra \ til	0	1	2	3	4	5
0	1	1	0	0	0	0
1	0	0	1	0	1	0
2	0	1	0	0	0	0
3	0	0	0	0	0	0
4	0	0	1	0	0	0
5	1	0	0	0	1	0

Figur 5.1: Vi setter altså 1 i matrise[a][b] dersom det går en kant fra a til b ellers nuller. Legg merke til at dersom vi har en urettet graf, trenger vi bare halve matrisa.

Der finnes selvsagt flere måter å implementere grafen på, for eksempel kan den reperesenteres som ei lenka liste, som illustrert i figur 5.2.



Figur 5.2: Her er samme graf som i figur5.1 illustrert med listerepresentasjon

5.3 Traversering

Her skal vi gå igjennom to nyttige måter å gå systematisk gjennom en graf på. Disse metodene er det veldig viktig å kunne og forstå!

5.3.1 Dybde - Først - Søk (DFS)

Prinsipp: Grafen skal utforskes i dybden. Det betyr at den neste kanten du skal utforske, går fra den noden du sist oppdaget. Dersom denne noden ikke har noen kanter som går til noder du ikke har oppdaga før, går du tilbake til forrige node og gjør det samme der, osv.

Dette prinsippet kan illustreres ved hjelp av en stakk (LIFO). Det elementet som ligger på toppen er det neste som utforskes. Hvis dette har en kant til ett annet element som ikke allerede ligger i stakken, legges dette nye elementet øverst. Hvis det elementet som ligger på toppen ikke har noen kanter til et uutforsket element, tas elementet ut av stakken og legges i en liste over ferdig-utforskede elementer.

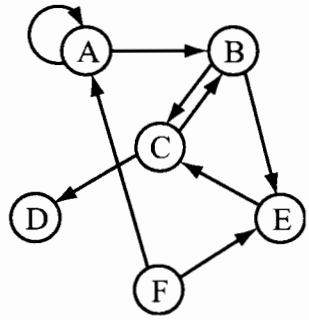
Legg merke til at dersom det går flere kanter ut fra den noden du skal undersøke til andre uoppdagede noder, spiller det i prinsippet ingen rolle hvilken du velger. Når du skal implementere dette, er det imidlertid greit å bestemme seg for en eller annen regel, eller velge vei "random".

Før du kan avslutte må du sjekke at alle nodene har blitt besøkt, for det er ikke alltid det er forblindelse mellom alle nodene i en graf. Dersom det er flere uoppdagede igjen, hopper du til en av dem og starter et nytt DFS. Dette gjentar du til alle nodene er oppdaget.

Eksempel 5.3.1. DFS

Vi oppretter listene oppdaget *og* ferdig *hvor vi legger henholdsvis nodene etterhvert som vi oppdager dem og de ferdige utforskede nodene. Husk at* oppdaget *er en stakk!*

Vi begynner på A og traverserer gjennom grafen i følge prinsippene over. Følg med på hvordan vi beveger oss i grafen!



Figur 5.3: Dette er grafen vi skal traversere.

1. oppdaget: A ferdig:
2. oppdaget: B ferdig: A
3. oppdaget: E,C ferdig: A,B *(E kjenner ingen nye noder. Vi går derfor tilbake til B og utforsker B sin andre nabo, C.)*
4. oppdaget: C ferdig: A,B,E
5. oppdaget: D ferdig: A,B,E,C *(C kjenner også B, men siden B allerede ligger i* ferdig, *blir ikke denne lagt i* oppdaget.*)*
6. oppdaget: ferdig: A,B,E,C,D *(Lista* oppdaget *er nå tom. Nå må vi søke i matrisa eller arrayen og se om det er flere noder igjen i grafen, og det er det!)*
7. oppdaget: F ferdig: A,B,E,C,D

❑

Du vil garantert komme over problemer hvor det kan være nyttig å benytte DFS. I dette kurset brukes DFS blant annet i topologisk sortering og i korteste-vei-problemer.

5.3.2 Bredde - Først - Søk (BFS)

Prinsipp Grafen skal utforskes i bredden. Det betyr at du utforsker alle kantene ut i fra den noden du står før du beveger deg til neste node.

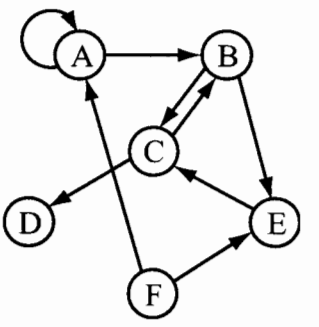
Dette prinsippet kan illustreres ved hjelp av en kø (FIFO). Det elementet som ligger først i køen er det første neste som skal utforskes. De elementene denne noden har kanter til

vil legges bakerst i køen. (Rekkefølgen spiller i prinsippet ingen rolle her heller.) Deretter fjernes elementet fra køen og legges i en liste med ferdig utforskede elementer.

Også her må du tenke på at alle nodene ikke nødvendigvis er forbundet. Du må alltid forsikre deg om at du har fått med alle nodene i grafen.

Eksempel 5.3.2. BFS
Dette eksempelet er helt analogt eksempel 5.3.1, bare at her bruker vi BDF.

Vi oppretter listene oppdaget *og* ferdig *hvor vi legger henholdsvis nodene etterhvert som vi oppdager dem og de ferdige utforskede nodene. Husk at* oppdaget *er en stakk!*



Figur 5.4: Dette er grafen vi skal traversere. Denne er helt lik den vi brukte i eksempel 5.3.1.

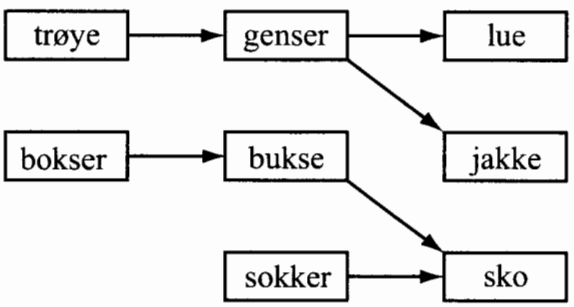
1. oppdaget:A ferdig:
2. oppdaget:B ferdig:A
3. oppdaget:C,E ferdig:A,B
4. oppdaget:E,D ferdig:A,B,C *(C kjenner også B, men siden B allerede ligger i* ferdig, *blir ikke denne lagt i* oppdaget.*)*
5. oppdaget:D ferdig:A,B,C,E
6. oppdaget: ferdig:A,B,C,E,D *(Lista* oppdaget *er nå tom. Nå må vi søke i matrisa eller arrayen og se om det er flere noder igjen i grafen, og det er det!)*
7. oppdaget:F ferdig:A,B,C,E,D,F

❑

5.4 Topologisk Sortereing

La oss si at nodene representerer oppgaver som skal gjøres. Noen oppgaver må gjøres før andre. For eksempel må du lage middagen før du kan spise den. Andre kan gjøres i valgfri rekkefølge. Dette kan illustreres med en rettet graf.

Eksempel 5.4.1. Å kle på seg i rett rekkefølge.



Figur 5.5: Et klassisk eksempel er rekkefølgen du kan kle på deg. I grafen har vi tatt med noen klesplagg, og pilene viser rekkefølgen de må tas på. Det kan selvsagt diskuteres om du må vente med å ta på deg lua til du har tatt på genseren, men av erfaring blir det gjerne ekstraarbeid dersom du forsøker en annen rekkefølge..

❑

Topologisk sortering går ut på å finne en mulig rekkefølge oppgavene kan gjøres og tar utgangspunkt i en DAG. Det er opplagt at topologisk sortering bare har mening når vi har å gjøre med en rettet graf, og den kan ikke inneholde sykler. Dersom vi har sykler eksisterer ingen gyldig løsning.

5.4.1 Algoritme

Topologisk sortering er veldig enkelt når du har forstått DFS. Du kjører rett og slett bare et DFS og printer ut oppgavene i den rekkefølgen de besøkes for siste gang. Du kan da uføre oppgavene i motsatt rekkefølge av det de blei printet ut i!

Eksempel 5.4.2. Løsning på kleseksempelet.
Vi må begynne traverseringen i en node som ikke har noen piler inn mot seg. Minst en slik node må finnes da grafen ikke kan ha noen sykler. Vi kan altså begynne å ta på trøye, bokser eller sokker.

Vi velger tilfeldigvis å begynne med sokkene. Disse legges øverst i stakken. Vi har bare

en vei å gå videre, nemlig til skoa. Ingen piler peker fra skoa, så vi kan være sikre på å ikke besøke dette noden i grafen igjen, og legger derfor skoa først i lista besøktForSisteGang. *Vi går så tilbake til sokker, og oppdager at heller ikke denne noden kjenner noen noder vi ikke har besøkt enda. Sokker legges så til i lista besøktForSisteGang. Nå er lista* oppdaget *tom igjen, og vi må søke for å se om det er flere noder i grafen. Det er det, men vi må passe på å velge en som ikke har noen kanter inn mot seg. Vi velger bokser. Det fører til at bukse og bokser blir puttet inn i besøktForSisteGang i den rekkefølgen. Vi finner så trøya, går videre til genseren, og velger tilfeldig lue før jakke. Lue kjenner ingen flere og blir lagt til i besøktForSisteGang. Vi går tilbake til genser som også kjenner jakke. Jakke, genser og trøye blir lagt til i lista i denne rekkefølgen.*

Etter at traverseringa er ferdig, ser lista besøktForSisteGang slik ut: sko, sokker, bukse, bokser, lue, jakke, genser, trøye. Vips har du en mulig rekkefølge å ta på deg klærne i, nemlig først trøye, så genser, jakke, lue, bokser, bukse, sokker og til slutt sko. Dette er kanskje ikke den rekkefølgen du velger til vanlig, men det er ingen praktisk grunn til at du ikke skulle kunne kle på deg i denne rekkefølgen, i alle fall ikke ut i fra den miniverden grafen gir oss. Du kan selvsagt argumentere for at du ikke vil ta på deg jakke og lue før frokost men vil gjerne ha på deg en bokser først. Dette er det imidlertid umulig å se ut i fra grafen.

❑

5.5 Minimale Spenntrær

Som vi vet er et tre en forbundet graf uten sykler. Et **spenntre** er en subgraf av en graf G som inneholder alle nodene til G og er et tre. Et **minimalt spenntre** av en vektet graf G er et spenntre av G, der summen av kostnadene til kantene inngår i treet, er minimal.

Vi skal se på to ideer som finner et minste spenntre. (Dette treet trenger ikke være unikt.)

5.5.1 Prims Algoritme

Prinsipp: Begynn med en tilfeldig node. I hvert steg velges den kanten med minst kostnad som forbinder de nodene som er med i treet med de som ikke er med. Når alle nodene er med i treet, har du funnet et minimalt spenntre.

5.5.2 Kruskals Algoritme

Prinsipp: Velg hele tiden den kanten med minst kostnad som ikke lager noen sykler. Dersom den kanten som har minst kostnad av de som er igjen vil lage en sykel, ignoreres denne. Fortsett til det ikke er flere kanter som kan legges til.

Kapittel 6

KORTESTE VEI-PROBLEMER

Denne typen problemer er ekstremt nyttig i mange sammenhenger. Hvis man skal kjøre bil fra et sted til et annet, vil man gjerne kjøre den korteste veien for å spare bensin og tid. I dette tilfelle holder det å se på **veilengden** som en **kostnad** for hvert delproblem (her vil delproblemene bli veiene mellom de forskjellige kryssene hvor vi må foreta veivalg). I dette tilfelle blir kostnaden alltid positiv, men det trenger den ikke bli i andre eksempler. Hvis man skal dra rundt og selge ting, kan det hende man vet at det på enkelte strekninger vil bli tap. Men det er fortsatt mulig å finne ut hvilken vei det lønner seg å ta for å tjene mest mulig penger.

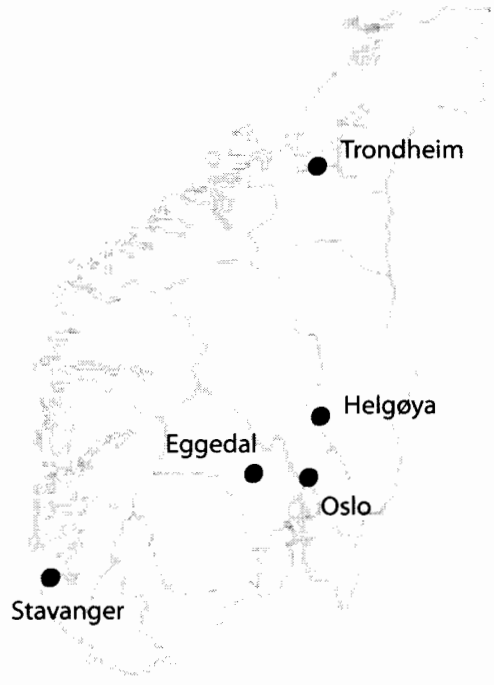
Det er dessuten også mange problemer som lett kan "oversettes" til et korteste-vei-problem. Det gjelder å være kreativ! Etter å ha oversatt et problem, er det bare å bruke en kjent algoritme og løse problemet. Men det gjelder også å holde tunga rett i munnen. Det er ikke alltid at tilsynelatende lignende problem så lett kan oversettes. Prøv selv å se på noen lengste-vei-problemer!

Korteste vei-problemer er ganske omfattende, så vi deler opp kapitlet i noen hensiktsmessige klasser:

6.1 Korteste vei, én-til-alle

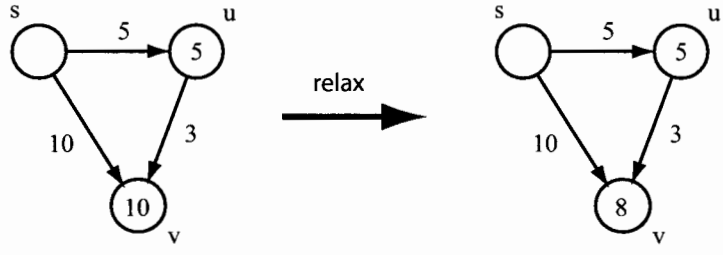
Her vil vi konsentrere oss om de problemene som har ett startpunkt, og som går til alle tenkelige sluttpunkt i den mengden man ser på. Jamfør bileksemplet ovenfor. Anta at vi er fire studenter som deler kjøkken på Moholt og alle har sin egen bil. Førstemann bor i Oslo, den andre i Stavanger, nummer tre i Eggedal og fjerdemann på Helgøya. Da holder det at man kjører en algoritme for korteste vei, én-til-alle-problem (her kan man også benytte seg av det gunstige faktum at alle kostnader er positive, noe som dere etterhvert vil se gir enklere og raskere algoritme) Med Trondheim, Oslo, Stavanger, Eggedal og Helgøya som mengden av steder, bruker man Trondheim som startpunkt og regner ut. Dermed kan alle ta hver sin bil (ikke så veldig sosialt eller miljøvennlig, men eksemplets intensjon helliger

midlet) og kjøre korteste vei til sin hjemby. Poenget i dette kapittelet er å se hvordan vi kan finne korteste vei.



Figur 6.1: De fem nevnte steder i Norge

Et viktig prinsipp i alle korteste-vei-problemene, er det som kalles **relaxation**. Dette **relax-prinsippet** er ikke noe mer skummelt enn at man sjekker en tilfeldig vei fra en node til en annen, og skulle denne være bedre enn den som allerede finnes, oppdaterer man den nye veien til å være den korteste. Skulle den veien man prøver være lenger, lar man tingenes tilstand være som de er.



Figur 6.2: Relaxation. Tallet i noden angir funnet avstand fra kilden

En viktig ting å ha i mente, er at en korteste vei **ikke kan ha sykler**. Hvis det eksisterer en **negativ sykel**, så vil man kunne gå rundt denne negative syklen så mange ganger man vil og få stadig kortere vei. Dermed vil det ikke kunne være en bestemt korteste vei, da

man alltid kan ta en ny runde i den negative syklen og få en enda kortere vei. Har man en sykel med **verdi 0**, så blir denne helt uinteressang og man kan like godt droppe den. Og har man en **positiv sykel**, så vil man ved å fjerne denne syklen fortsatt ha en vei mellom sine to noder som også blir kortere enn om man brukte syklen.

La oss nå se på noen algoritmer som løser slike problemer.

6.1.1 Bellman-Ford

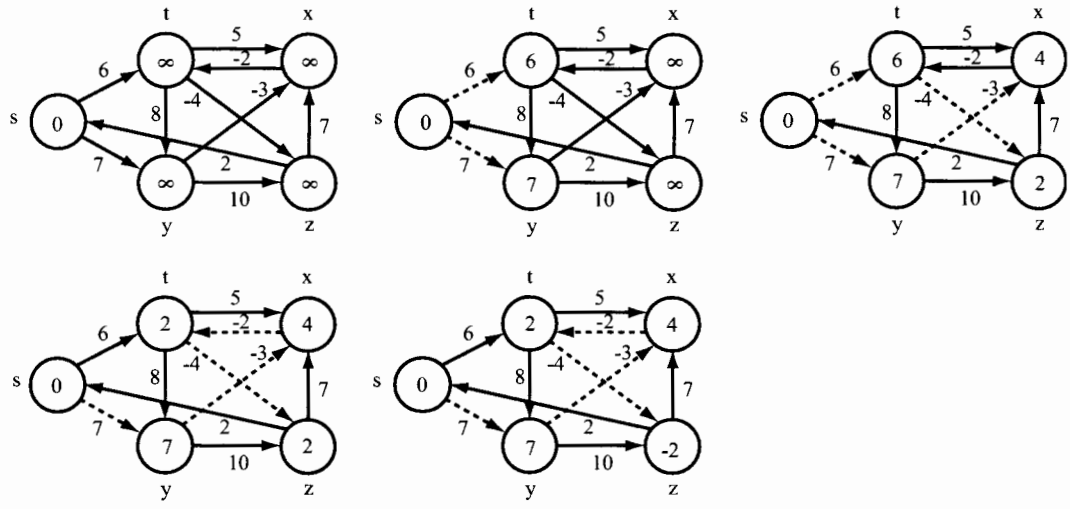
Denne algoritmen tar for seg de problem hvor kostnadene på **kantene** til grafen kan være **negative**. Dermed blir den ganske generell. Hvis det eksister **negative sykler**, returnerer Bellman-Ford "FALSE". Men om slike negative sykler ikke forekommer (vel og merke holder det at de ikke kan nås fra utgangspunktet vårt), så returnerer algoritmen den beste vei.

Algoritmen fungerer som følger (se også på pseudokoden i [1] side 588):

1. Man tar for seg én og én node i grafen. For hver node sjekker man avstanden fra **startnoden** til **alle de andre nodene** via denne noden, og bruker relaxation-prisnsippet. Når man er ferdig med dette, vil man faktisk ha funnet den korteste veien! Vel å merke om man ikke har en negativ sykel. Studér kommende eksempel, og prøv å forstå hvorfor det må bli slik. Alle veier mellom nodene vil være utprøvd, og den korteste veien blir funnet takket være relaxation-prinsippet.

2. Deretter gjelder det å finne ut om vi faktisk har en negativ sykel. Dette gjør man ved å sjekke samtlige kanter i grafen. La oss si at vi har en kant fra node u til node v, kall denne kanten (u, v). Hvis summen av kostnaden fra startnoden til node u og kostnaden til (u, v) blir mindre enn verdien vi fant for kostnaden fra startnoden til v, vil dette bety at man har en negativ sykel. Dette bevises for så vidt i [1] på side 590.

Hvis vi kaller antallet noder for n og antallet kanter for k, blir kjøretiden til Bellman-Ford-algoritmen ganske enkelt $O(nk)$. Dette ser man lett, da man går gjennom alle nodene en gang og for hver gang man tar for seg en node så vil man måtte sjekke et antall kanter som er proposjonalt med antallet kanter som eksisterer. For å sjekke negative sykler tar det bare $\Theta(k)$ i tid, så denne vil ikke påvirke den asymptotiske kjøretiden fra første del av algoritmen.



Figur 6.3: Bellman-Ford

6.1.2 Dijkstra

Kravet for å bruke Dijkstras algoritme, er at alle kantene har **positive** verdier (eventuelt også verdien 0). Da kan man bruke en smartere algoritme enn Bellman-Ford. Dijkstra er faktisk en **grådighetsalgoritme** (se neste kapittel). Pseudokoden står på side 595 i [1]

Det faktum at grådighetsprinsippet fungerer her, gjør at man kan kjøre gjennom en algoritme som ikke tester alt det som Bellman-Ford måtte ha testet. Hvorfor grådighet vil fungere, er ikke så vanskelig å se for seg. Se på illustrasjonen nedenfor. Hvis du bruker 10 liter bensin bort til Ola og 15 liter til Mogens, vil det aldri i verden kunne være lønnsomt å dra via Mogens til Ola hvis vi ser på bensinforbruket. For vi må nødvendigvis svi av noe bensin mellom Ola og Mogens, så uansett hva vi forbruker mellom disse to fyrene vil det fra et bensin-perspektiv være ulønnsomt å dra via Mogens (ja, det finnes selvsagt argumenter for å besøke Mogens. Det kan jo være direkte hyggelig, og dessuten gir han deg kanskje bensin gratis. Men i dette tilfelle vil denne bensinen Mogens gir deg regnes som et negativt forbruk, og vi kan ikke bruke Dijkstra!).

Ellers kan pseudokodene til Dijkstra og Bellman-Ford se veldig forskjellige ut, men det er ikke så stor forskjell. I Dijkstra trenger man heller ikke den delen av Bellman-Ford hvor man søker etter negative sykler.

Kort foralt tar Dijkstra og plukker ut nodene en etter en ettersom hvor nær de er startnoden hvis vi følger korteste vei. Den første noden som plukkes ut er startnoden (husk at dette er en én-til-alle-algoritme). Deretter kjøres relaxation til alle andre noder for å finne neste node som skal plukkes ut, dvs den noden som nå ligger nærmest startnoden. De nodene vi har plukket ut tas ikke med i neste runde med relaxation. Dette gjentas til vi har funnet korteste vei til alle nodene. Eksempel 6.1.1 viser kanskje tankemåten enda

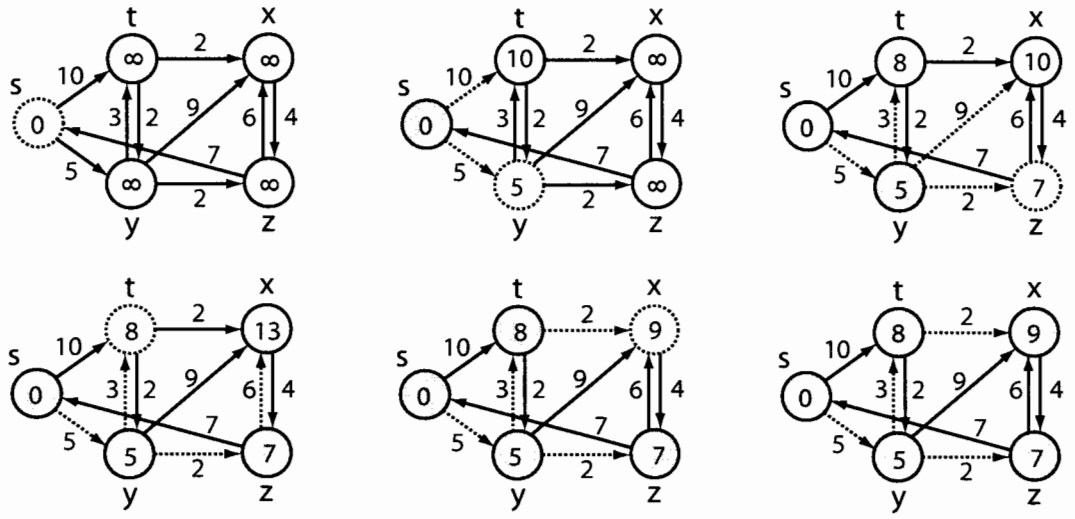
klarere.

Eksempel 6.1.1. Dijkstra i "praktisk"!
En gjeng syklister er samlet på startstreken (startnoden) til et sykkelritt. Alle er like sprekе og sykler nøyaktig like fort hele tiden. Vi lar kostnaden mellom to noder være tiden det tar for en syklist å sykle fra den ene til den andre. Målet i et tradisjonelt sykkelritt er selvsagt å komme først til mål, men her er det snakk om å finne korteste vei til alle nodene.

Det er mange veivalg underveis, men vi antar at det er så mange syklister at det alltid er minst en til hvert veivalg. Slik vil den eller de syklisten(e) som besøker en node først, ha funnet den korteste veien til denne noden og tiden han/de brukte illustrerer kostnaden til denne veien.

Det er akkurat slik Dijkstra finner de korteste veiene; så fort en syklist når fram til en hittil ubesøkt node, legges den til i mengden med noder vi har funnet korteste vei til.

❑



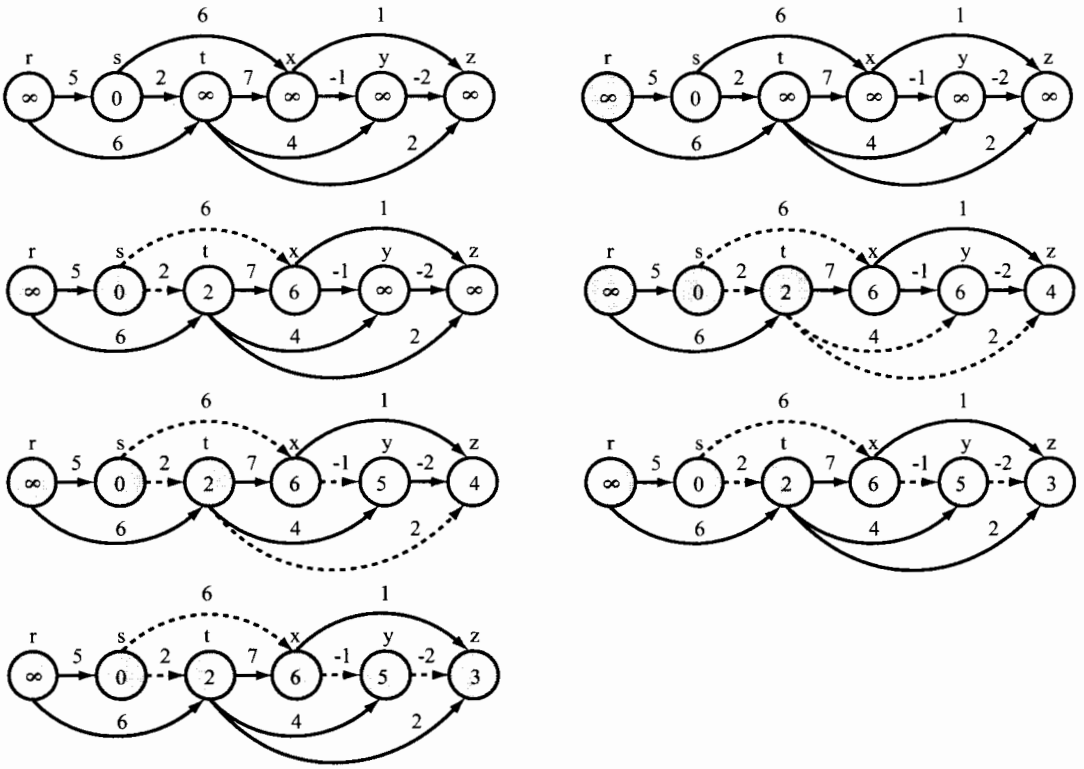
Figur 6.4: Dijkstra

Det kan vises at kjøretiden til Dijkstra er $O(n^2)$ i de fleste tilfeller, men den kan gjøres bedre om man har god kunnskap om grafen man ser på. Merk at $O(n^2)$ er bedre enn $O(nk)$, fordi antallet kanter k minst må være $n-1$ mens den øvre grensen faktisk kan være asymptotisk med n^2.

6.1.3 Hva om man har en DAG?

Har man en DAG, vil det ikke oppstå noen vanskeligheter selv om man har negative kostnader på kantene, for i en DAG vil det ikke eksistere noen sykler i alle tilfelle!

Trikset for å finne korteste vei er i første rekke å gjøre en **topologisk sortering** av DA-Gen. Slik oppnår vi en lineær ordning av nodene. Deretter trenger vi bare besøke nodene en gang i den topologisk sorterte rekkefølgen og kjøre en relaxation for hver gang til de noder som ligger foran. Om dette skulle være uklart, så skulle det holde å se på figurene nedenfor. Denne algoritmen er enkel!



Figur 6.5: Sortering av DAG

Kjøretiden vil avhenge av den topologiske sorteringen, og en slik sortering har som kjøretid $\Theta(n+k)$. Ingen andre ledd i algoritmen for korteste vei i en DAG vil bli større enn den topologiske sorteringen, så kjøretiden blir $\Theta(n+k)$.

6.2 Korteste vei, alle-til-alle

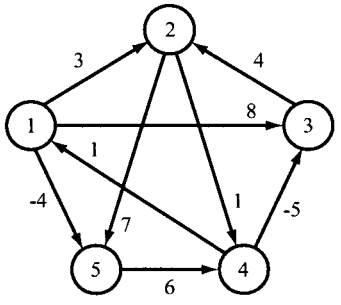
En soleklar måte å takle dette problemet på, er rett og slett å bruke en god algoritme for et én-til-alle-problem, og så bruke denne på samtlige noder som startnoder. Dette er ikke så dumt som det kan høres ut som. Hvis alle kostnadene er ikke-negative, fungerer det meget bra å bruke Dijkstra. Kjøretiden her vil bli $O(n^3)$, som man ser ved å multiplisere kjøretiden til en gjennomgang av Dijkstra med antallet noder n. Hvis vi har negative kostnader og kjører samme trikset med Bellman-Ford, blir kjøretiden $O(n^2k)$, som ikke er så veldig bra fordi k er av størrelsesorden n^2. Nå skal det sies at med meget smarte fremgangsmåter her,

vil disse kjøretidene kunne senkes betraktelig. Men vi bruker heller en algoritme som tar for seg problemet med alle-til-alle uten å skulle gå omveien om mange gjennomkjøringer av en én-til-alle-algoritme.

6.2.1 Floyd-Warshall

Ved å benytte oss av prinsippet om **dynamisk programmering**, skal vi her vise hvordan Floyd-Warshall-algoritmen fungerer.

Pseudokoden for Floyd-Warshall er relativt grei. Man lager en **nabomatrise** til nodene, men gjør litt om på den. Hvis det går en vei fra en node til en annen, setter vi verdien til å være **kostnaden på kanten** mellom dem. Går det ikke en direkte kant mellom de to nodene, settes kostnaden til å være **uendelig**. Deretter velger man seg en node, kall noden a, som man har lov til å gå innom. Så tester man veien mellom to og to noder, kall dem u og v. Hvis det koster mindre å gå via a, så legger man inn denne veien til å være den korteste. Etter å ha latt a være alle nodene, har man fått en nabomatrise som viser minste kostnad mellom nodene. Det er heller ikke så veldig vanskelig å lage enda en nabomatrise som holder styr på hvilke noder man skal gå innom for å oppnå denne ultimate veien. Det er ikke så rent ulogisk at dette vil fungere, men det gjelder å få et godt grep om hva som skjer. Studér eksemplet som følger.



Figur 6.6: Graf som vi kjører Floyd-Warshall på. Tallene i nodene betyr her nummeret på noden! Ikke avstand som det av og til gjør

Figuren viser grafen til problemet vårt. Om vi bruker algoritmen på side 630 i [1], vil matrisene under utregningen bli slik du ser under her. Merk at indeksen i hver matrise betyr hvilken node vi har gått innom. Den første, med indeks 0, er startmatrisen.

$$D^{(0)} = \begin{bmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 1 & \infty & -5 & 0 & \infty \\ \infty & \infty & \infty & 6 & 0 \end{bmatrix}$$

$$D^{(1)} = \begin{bmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 1 & 4 & -5 & 0 & -3 \\ \infty & \infty & \infty & 6 & 0 \end{bmatrix}$$

$$D^{(2)} = \begin{bmatrix} 0 & 3 & 8 & 4 & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & 5 & 11 \\ 1 & 4 & -5 & 0 & -3 \\ \infty & \infty & \infty & 6 & 0 \end{bmatrix}$$

$$D^{(3)} = \begin{bmatrix} 0 & 3 & 8 & 4 & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & 5 & 11 \\ 1 & -1 & -5 & 0 & -3 \\ \infty & \infty & \infty & 6 & 0 \end{bmatrix}$$

$$D^{(4)} = \begin{bmatrix} 0 & 3 & -1 & 4 & -4 \\ 2 & 0 & -4 & 1 & -2 \\ 6 & 4 & 0 & 5 & 3 \\ 1 & -1 & -5 & 0 & -3 \\ 7 & 5 & 1 & 6 & 0 \end{bmatrix}$$

$$D^{(5)} = \begin{bmatrix} 0 & 1 & -3 & 2 & -4 \\ 2 & 0 & -4 & 1 & -2 \\ 6 & 4 & 0 & 5 & 3 \\ 1 & -1 & -5 & 0 & -3 \\ 7 & 5 & 1 & 6 & 0 \end{bmatrix}$$

Kjøretiden til Floyd-Warshall er ganske enkelt $O(n^3)$. Det er n noder man skal gå innom, og for hver gang kan man variere startnoden med $n-1$ muligheter og sluttnoden med $n-2$ muligheter. Om man ser på pseudokoden i [1], så ser man at den består av tre greie for-løkker. Nå skal det sies at også Dijkstra på alle nodene gir $O(n^3)$, men operasjonen per ledd i Floyd-Warshall er så mye mindre at denne stort sett vil lønne seg selv om alle kantene er positive.

Husk også at det å tegne opp grafen er et meget bra triks i denne typen oppgaver.

Kapittel 7

MAKS FLYT-PROBLEMER

I forrige kapittel, under Floyd-Warsall-algoritmen, brukte vi en slags nabomatrise med den forskjell at verdiene mellom to noder ikke var 1, men kostnaden på kanten mellom dem. Gikk det ingen direkte vei, ga vi verdien uendelig. Dette konseptet kan utvides en hel del. Istedenfor å ha kostnad på kanten mellom to noder, så er det ofte gunstig å operere med en **flyt** mellom dem. Denne flyten kan være meget generell, så maks-flyt-problemer spenner over et stort område. I et maks-flyt-problem ønsker vi å finne den største flyten fra en gitt kilde til et gitt sluk uten å overskride noen kapasitetskrav.

7.1 Flytnettverk

Et flytnettverk $G = (V, E)$ (hvor V står for nodene og E for kantene) er en **rettet graf** hvor hver kant (u, v) har en ikke-negativ **kapasitet** $c(u, v) \geq 0$. Eksisterer ingen kant mellom to noder, setter vi kapasiteten til å være null. Det er to noder som er av essensiell betydning. Det er **kilden** s, og det er **sluket** t. Vi kan nå definere flyt på følgende måte:

Definisjon. En **flyt** i G er en funksjon

$$f : V \times V \to \mathbb{R}$$

slik at følgende tre krav er oppfylt:

1. **Kapasitet:** For alle $u, v \in V$, så krever vi $f(u, v) \leq c(u, v)$.
2. **Symmetri:** For alle $u, v \in V$, så krever vi $f(u, v) = -f(v, u)$.
3. **Bevaring av flyt:** For alle $u \in V - \{s, t\}$, så krever vi

$$\sum_{u \in V} f(u, v) = 0$$

Vi sier at $f(u, v)$ er **flyten** fra kilden til sluket, og dens verdi er rett og slett summen av flyten til alle veiene ut fra kilden. Som igjen er det samme som summen av flyten inn i

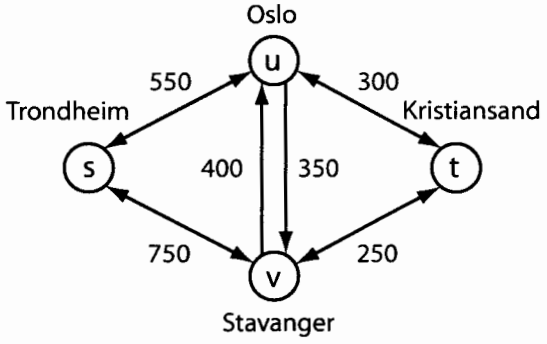
sluket. Mer presist:

$$|f| = \sum_{v \in V} f(s, v)$$

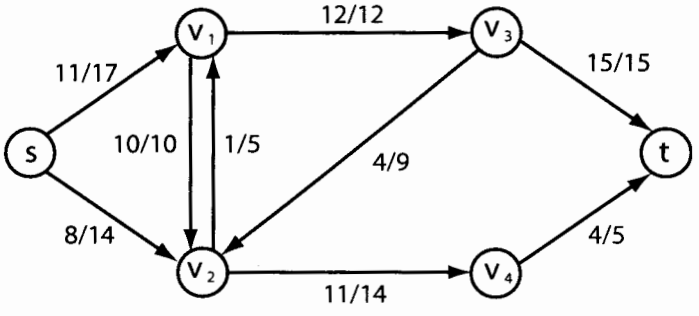
Det første kravet sier at vi alltid må rekspektere kapasiteten. Det andre kravet er egentlig bare en formalitet som gjør det mulig å regne med negativ flyt, som generaliserer teorien en hel del. Hvis jeg sender hundre kroner til din konto, vil det gå en positiv flyt av penger fra meg til deg, mens det går en negativ flyt fra deg til meg. Bevaringsloven til sist sier bare at flyt ikke uten videre kan forsvinne eller oppstå. Dette kan sammenlignes med diverse bevaringslover fra fysikken, som f.eks energibevaring.

Legg spesielt merke til en ting. De tre kravene gir at **total flyt** i en node må være null. Det som kommer inn blir sendt ut igjen med samme rate, det er ingenting som hoper seg opp. Dette kan sammenlignes med Kirchoffs lover for elektrisk strøm. Ellers er den **totale positive flyten** definert til å være summen av de positive flytene inn i en node, eller:

$$\sum_{\substack{u \in V \\ f(u,v)>0}} f(u, v).$$

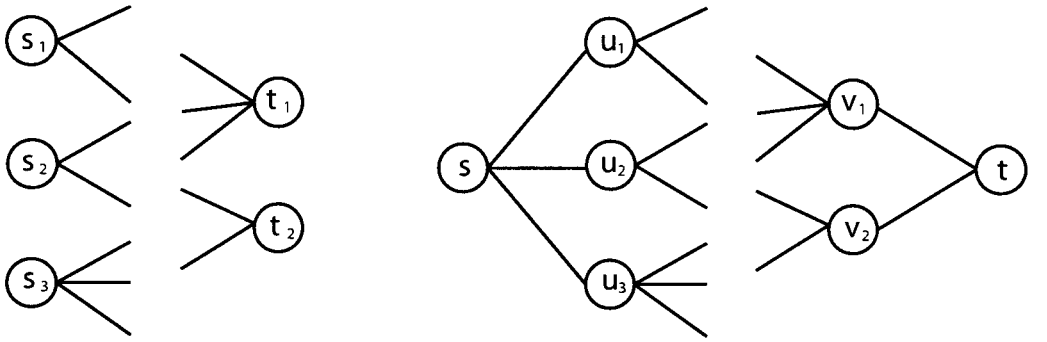


Figur 7.1: Gjennomsnittlig bensinforbruk mellom byer i Norge



Figur 7.2: Et typisk flytnettverk. Flyten er i telleren og kapasiteten er i nevneren.

Hvis vi har flere kilder og sluk, er det et godt triks å legge til en **superkilde** og et **supersluk**, slik at vi oversetter problemet til det vi er vant til.



Figur 7.3: Til venstre, tre kilder og to sluk. Til høyre har vi en superkilde og et supersluk!

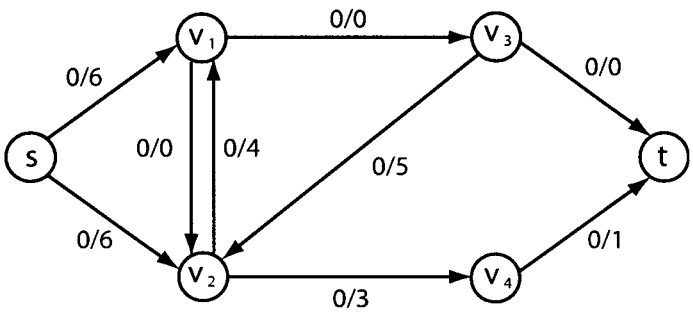
7.2 Ford-Fulkerson

Ford-Fulkerson er en god algoritme for å løse maks-flyt-problemet. Dette er en iterativ metode som tar i bruk tre essensielle ideér, residual-nettverk, flytforøkende vei og snitt.

7.2.1 Residual-nettverk

Som uttrykket kanskje vil tilsi, for et flytnettverk med en gitt flyt, består residual-nettverket av de kantene som tillater mer flyt!

Eksempel 7.2.1.
Vi beskriver ofte flyten og kapasiteten ved å skrive en brøk med flyt-verdien øverst og kapasitetsverdien nederst. Man får residual-nettverket ved å ta de opprinnelige kapasiteter og trekke fra den flyten som går i hver kant. Med flyten fra figur 7.2 får vi residual-nettverket til å bli slik som i figur 7.4.



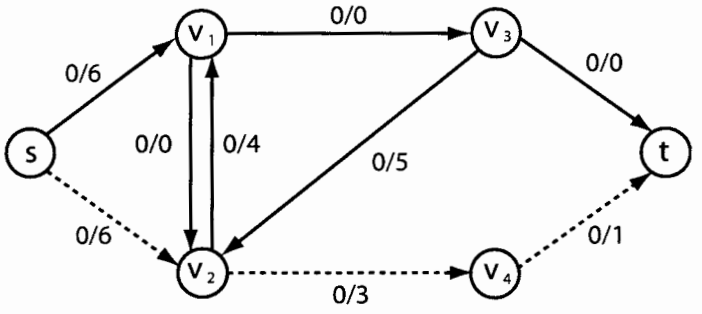
Figur 7.4: Residualnettverket

❑

Gitt et flytnettverk G og en flyt f. Da skriver vi G_f for residual-nettverket.

7.2.2 Flytforøkende vei

En flytforøkende vei er rett og slett en enkel vei fra kilden s til sluket t i residual-nettverket G_f. I figur 7.5 er den stiplede veien en flytforøkende vei. Vi bruker residualnettverket fra figur 7.4.



Figur 7.5: Flytforøkende vei

Vi ser at vi kan sende en enhet i den flytforøkende veien.

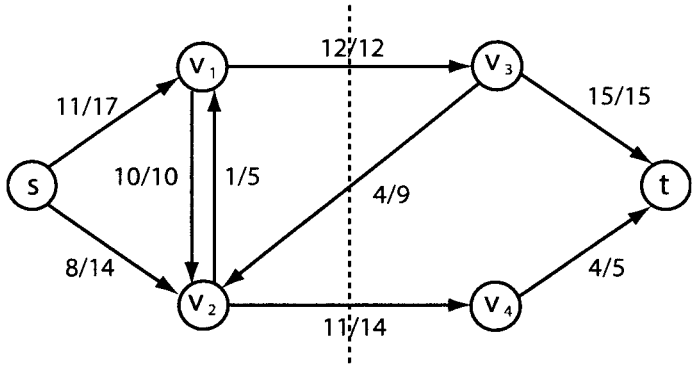
7.2.3 Snitt

Vi definerer et snitt som følger:

Definisjon. Et **snitt** (S, T) til et flytnettverk $G = (V, E)$ er en **partisjon** av nodene V i S og $T = V - S$ slik at kilden s ligger i S og sluket t ligger i T.

Med andre ord er snittet en inndeling av alle nodene i to deler hvor kilden og sluket er i hver sin del. Vi snakker om **nettoflyt** og **kapasitet** over et snitt, hvor det førstnevnte er all flyten gjennom snittet fra kilden til sluket minus flyt fra sluket til kilden, og kapasiteten over snittet er kapasiteten fra kilde til sluk minus kapasitet fra sluk til kilde. Et **minimalt snitt** er et snitt hvis kapasitet er den minste oppnålige.

Eksempel 7.2.2.
Ved å bruke grafen fra figur 7.2, kan vi dele et snitt som i figur 7.6.



Figur 7.6: Snitt. S er til venstre, og T er til høyre.

❑

Det vises i [1] at flyten gjennom et snitt (S, T) rett og slett er $|f|$. Om man skjønner hva dette står for, er resultatet temmelig trivielt. Det sier bare at den flyten som sendes fra kilden og går inn i sluket, i sin helhet må krysse snittet! Ved å se på figuren over, ser man at dette er tilfellet. Tenker man litt etter, så vil man innse at dette enkle resultatet medfører det faktum at flyten f i et flytnettverk G har en **øvre begrensning** ved kapasiteten til ethvert snitt. Av dette følger et meget viktig teorem:

Teorem. ***Maks flyt, minimalt snitt***
Hvis f er en flyt i et flytnettverk $G = (V, E)$ med kilde s og sluk t, så har vi:

1. *f er en maksimal flyt i G.*
2. *Residualnettverket G_f har ingen flytforøkende vei.*
3. *Verdien på flyten fra kilde til sluk, $|f|$ må være lik kapasiteten til et eller annet snitt.*

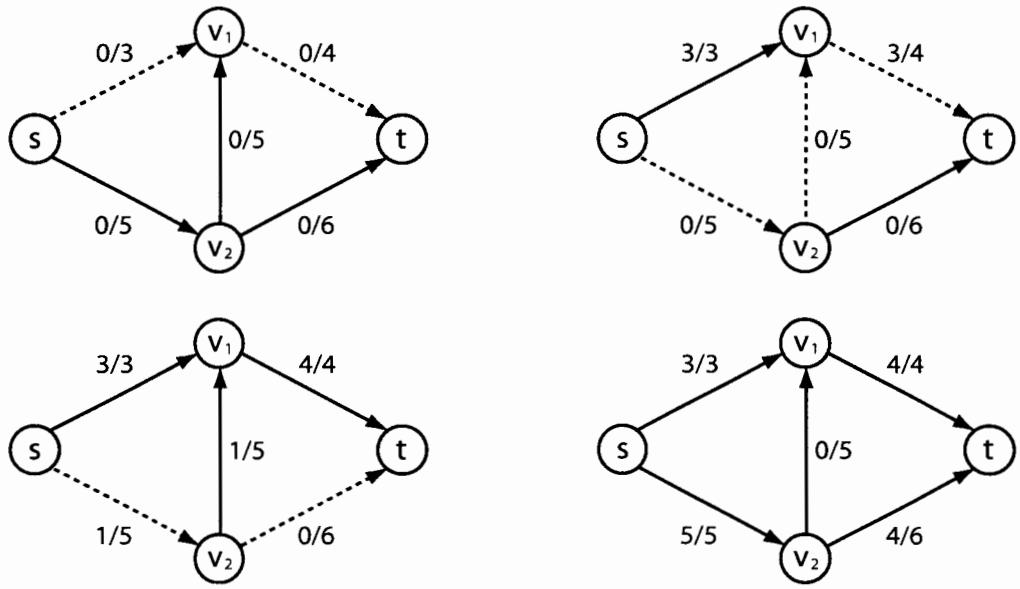
For å si det enkelt, det å finne det minimale snitt er det samme som å finne den maksimale flyten! Ved å tenke seg om, er dette ganske greit. Når vi har maksimal flyt, så vil det bli "kork" der hvor kapasiteten er minst, og ingenting mer kan slippe gjennom denne flaskehalsen. Skal man få større flyt, **må** man øke kapasiteten i flaskehalsen. Det hjelper for eksempel ikke uten videre å legge 3-felts motorvei fra Oslo mot Svinesund når veien ut fra Oslo består av én fil (vel og merke for en stakkars Oslobeboer som har bopel i sentrum eller lenger vest).

7.2.4 Ford-Fulkerson-algoritmen

Nå har tiden kommet til selve algoritmen. Ideén her er å finne en eller annen flytforøkende vei p og øke flyten f på hver kant til p med residualkapasiteten. Algoritmen er så enkel som at man rett og slett finner en flytforøkende vei, og når man har gjort det setter man på all den flyten som veien kan tåle. Deretter leter man etter en ny flytforøkende vei, og gjør det sammen. Når man oppdager at det ikke er mer flytforøkende veier, har man oppnådd maksimal flyt!

Det som kan være urovekkende, er hvordan man skal sjekke om det eksisterer en flytforøkende vei. Et av de bedre triksene, er å bruke et **bredde-først-søk**. Men denne kan i enkelte tilfeller gi en lang kjøretid, og det eksisterer bedre måter å søke på i dette tilfellet. De vil vi derimot ikke gå inn på her.

Eksempel 7.2.3. En kjøring av Ford-Fulkerson
Vi starter med et nettverk uten noe flyt. Det som er uthevet i grått er en flytforøkende vei. Gangen blir da som følger:



Figur 7.7: Ford-Fulkerson

❑

Bruker man et bredde-først-søk, så vil man ha at kjøretiden til Ford-Fulkerson er $O(E|f^*|)$, hvor $|f^*|$ er den maksimale flyten funnet ved algoritmen, og E som vanlig er antallet kanter. Dette ser man greit, ved at man for hvert søk risikerer å måtte gå igjennom alle kantene, og man vil sende minst en enhet av flyten hver gang. Legg merke til at Ford-Fulkerson ikke hadde fungert dersom vi hadde hatt irrasjonale kapasiteter. Men implementert på en PC, så vil man i praksis bare operere med **rasjonale tall**, så saken vil bli biff.

Kapittel 8

DYNAMISK PROGRAMMERING OG GRÅDIGHETSALGORITMER

To viktige begreper i dette kapittelet:

Optimal substruktur: Den optimale løsningen på problemet består av optimale løsninger på underproblemer. Det vil si at du kan bruke løsninger på delproblemer for å finne løsningen på hele problemet.

Eksempel 8.0.4. Fibonacci-Tallene
Fibonacci-tallene er definert i 1.2.1.

Å finne det n'te fibonacci-tallet er et eksempel på et problem med optimal substruktur. Hvis vi vet f(n-1) og f(n-2) kan vi bruke dette til å finne f(n).

❑

Overlappende delproblemer: Se opp for problemer med denne egenskapen! En naiv splitt og hersk algoritme vil løse samme problem mange ganger. I dette kapittelet vil du se hvordan slike probemer kan løses effektivt.

Eksempel 8.0.5. Fibonacci-Tallene forts.
For å finne f(n), må vi først finne f(n-1) og f(n-2). En naiv rekursiv metode vil regne ut disse hver for seg:

$f(n\text{-}1) = f(n\text{-}2) + f(n\text{-}3)$
$f(n\text{-}2) = f(n\text{-}3) + f(n\text{-}4)$

Allerede her kommer det fram to overlappende delproblemer, f(n-2) og f(n-3). Disse vil her regnes ut flere ganger.

❑

8.1 Dynamisk Programmering

Hvis du kommer over et problem med optimal substruktur og overlappende delproblemer, er det stor mulighet for at dette er en oppskrift du kan følle for en effektiv algoritme:

1. Bli sikker på at problemet har optimal substruktur og overlappende delproblemer
2. Løs problemet for den eller de mest trivielle situasjonen(e) Bruk så disse til å finne de nest vanskeligste osv.
3. Hvilke valg har du? Hva bør du velge i de forskjellig tenkelige situasjonene?
4. Sett opp et skjema og fyll inn for noen tall.
5. Programmér:)

Eksempel 8.1.1.
Per er vedlig dårlig i rettskriving. Han vet ikke om han skal skrive "gullerot" eller "gulrot". Per er ikke alene, blant annet har de som designer søkemotorer på internett oppdaget at det er mange som han. I steden for at han får null treff på "gullerot"er det et ønske om at han skal få opp et spørsmål om han ikke heller vil søke på "gulrot" eller andre lignende ord. Denne funksjonen er også nyttig for de som skriver touch litt fortere enn de klarer..

Søkemotoren trenger da blant annet et program som sammenligner ord. Det er dette vi skal lage.

Som eksempel bruker vi "gullerot" og "gulrot". Vi ønsker å finnet ut hvor stor forskjell det er på disse to skrivemåtene, dvs hvor mange editeringer vi må gjøre for å få "gullerot" til å bli "gulrot".

Har du misstanke om at vi kan bruke dynamisk programmering?

1. ***Har problemet optimal substurktur?*** *Svaret er ja. For eksempel er et delproblem å finne ut hvor mange editeringer du trenger for å forandre* gulro *til* gulle. *Hvis vi veit svaret på dette, er oppgaven vår litt enklere, altså optimal substruktur.*
   Har problemet overlappende delproblemer? *La oss si at vi har funnet ut hvor mange editeringer som skal til for å forandre* gul *til* gulro. *Dette vil vi bruke både når vi skal finne ut hvor mange editeringer som skal til for å forandre* gul *til* gulrot, gull *til* gulro *og* gull *til* gullrot. *Dvs dette problemet, som vi nå har løsningen på er et underproblem til alle disse tre. (Men ingen av disse tre er delproblemer av hverandre.) Altså har vi overlappende delproblemer.*

Du kan altså bruke dynamisk programmering :-) Vi går videre til punkt 2 i oppskriften over.

2. *De mest trivielle delproblemene er antall editeringer for å forandre* g *til* g, gu ... *og* gulrot *og andre veien;* g, gu, gul, gull ... *og* gullerot *til* g.

   Dette kan settes inn i en matrise. Vi kaller matrisen c. *Verdien i f.eks rute* $c(2,3)$ *(nummerering som i java, begynner å telle fra 0) er minste antall editeringer som skal til for å forandre* gul *til* gulr. *I figur 8.1 er de trivielle verdiene fylt inn.*

		g	u	l	r	o	t
		0	1	2	3	4	5
g	1						
u	2						
l	3						
l	4						
e	5						
r	6						
o	7						
t							

Figur 8.1:

3. *Spørsmålet er hvordan vi finner verdien i* $c(i,j)$.

   La oss si at vi har regnet ut de tre nærmeste delproblemene A, B og C, se figur 8.2. Vi har da tre muligheter for verien i $c(i,j)$*:*

   (a) **bokstav i = bokstav j:** *Vi trenger ikke gjøre noen nye editeringer, dvs* $c(i,j) = A$.

   (b) **bokstav i ≠ bokstav j:** *Her er det to muligheter:*

      i. Vi kan ta verdien i B og legge til 1. Dette tilsvarer å erstatte bokstav i med bokstav j.

      ii. Vi kan ta verdien i C og legge til 1. Dette tilsvarer å slette bokstav i.

   Vi velger selvsagt det som gir alternativet som tilsvarer ferrest editeringer (lavest verdi).

A	B
C	c(i,j)

Figur 8.2:

4. *Vi har allerede tyvstartet på dette punktet. Skjemaet har vi klart. Det er bare å fortsette å fylle inn. Figur 8.3 viser tabellen delvis utfylt. Fyll ut resten selv!*

		g	u	l	r	o	t
		0	1	2	3	4	5
g	1	0	1	2	3	4	
u	2	1	0	1	2	3	
l	3	2	1	2	3	4	
l	4						
e	5						
r	6						
o	7						
t							

Figur 8.3:

5. *Klarer du å programmere dette? Selvsagt gjør du det :-) (Hint: Først må du opprette en tabell som har størresle "lengdePåFeilOrd" * "lengdePåRettOrd". Deretter må du fylle inn de trivielle verdiene. Husk at verdien i c(0,0) avhenger av om første bokstav i de to ordene er like eller ikke. Tenk logisk her, dette er ikke vanskelig! Deretter må du lage en dobbel for-løkke som går gjennom hele matrisen, venstre mot høyre, ovenfra og ned, og fyll inn verdier ut i fra reglene i punkt 3.*

❑

8.2 Memoisering

Her benytter vi vanlig traversering, men i steden for å løse samme problem flere ganger lagres verdiene i en tabell etter hvert. For hver verdi som skal beregnes kikker vi altså først i tabellen for å se om vi har gjort dette før, hvis ikke må vi regne.

Vi velger memoisering og ikke dynamisk programmering når vi ikke trenger å regne ut alle underproblemene for å løse hovedproblemet.

8.3 Grådighetsalgoritmer

Problemene vi bruker grådighetsalgoritmer på ligner ofte veldig på problemene som er beskrevet tidligere i dette kapittelet. Vi har fortsatt optimal substruktur og vi har valg. Forskjellen ligger i at valgene er mye enklere. Vi tviler ikke lenger på hva som er best, en ting skiller seg klart ut og vi kan velge det hver gang uten å trenge å vurdere de andre.

I neste avsnitt får du et eksempel, nemlig Huffmannkode.

8.3.1 Huffmankode

Huffman-kode er en særdeles effektiv teknikk for å komprimere data. Idéen er å ha en **variabel** lengde på kodebitene. For eksempel i en enkel tekstfil lar man de bokstavene som

forekommer ofte ha en kort kode, og de man bruker sjelden gir man en lang kode.

En **prefikskode** er slik at intet kodeord også er et prefiks til et annet kodeord. Dette gjør dem meget lett å dekode, for man trenger da ikke vite hvor et ord slutter og et annet begynner! En dekodingsprosess trenger en hendig representasjon for prefiks-koden slik at det opprinnelige kodeordet lett blir funnet. En god representasjon er rett og slett et binærtre hvor bladene er de gitte bokstavene. Huffman fant opp en **grådighetsalgoritme** som konstruerer en **optimal** prefikskode. Huffman-algortimen bruker frekvensen til de forskjellige bokstavene og lager et binærtre.

Eksempel 8.3.1. Huffmankode
Anta at vi sender noe informasjon som kun innehar bokstavene e, r, s og t. Vi vet også den relative frekvensen til bokstavene. Dette er vist i tabellen under:

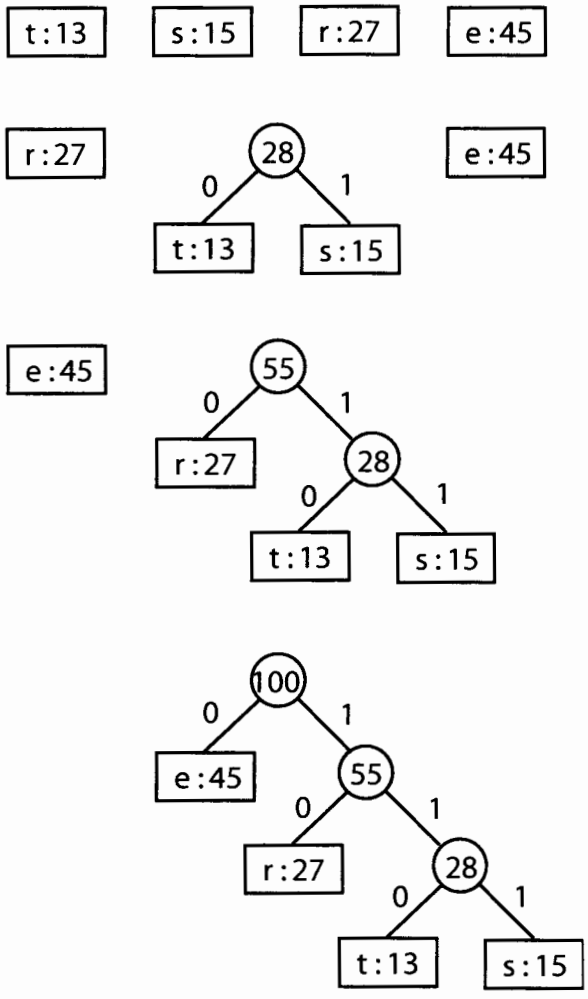
Bokstav	e	r	s	t
Frekvens	45	27	15	13

Nå bruker vi algoritmen til Huffman steg for steg og kommer til slutt fram til det optimale binærtreet for disse fire bokstavene.
Dermed har vi koden til de forskjellige bokstavene, som blir:

Bokstav	e	r	s	t
Kode	0	10	111	110

Legg merke til at dette er en prefikskode. Ordet "se" blir 1110, ordet "erterester" blir da 0101100100111110010 (hva nå enn det skulle bety).

❑



Figur 8.4: Huffman-treet slik det blir fra eksemplet

Kapittel 9

NPC

Det finnes mange klasser av problemer. P, NP, NPC og NP-Hard er fire slike klasser.

NP ("Non-deterministic Polynomial"): Mengden av de problemene der gyldigheten av en gjettet løsning kan testes i polynomisk tid.

P ("Polynominal"): Mengden av de problemene vi vet kan løses i polynimisk tid. Det vil si de som har "worst case"kjøretid på O(nk).Det er opplagt at disse er med i mengden NP. Det er slike problemer dette kurste går ut på å løse.

NPC ("NP-komplett"): Hvis det finnes en polynomisk løsning for et problem i NPC, finnes det polynomike løsninger for alle problemer i NP. Dette er definisjonen på et NP-komplett problem. Dersom vi finner løsningen på et av disse problemene, har vi løst dem alle og NP=P. Det motsatte gjelder også; hvis det viser seg å være uløsbart, er alle det.

I likhet med andre "komplette" problemer i andre problem-klasser er altså NPC-problemene de vanskeligste som finnes i klassen. I prinsippet er de også bare et problem, da alle kan reduseres til det samme problemet i polynomisk tid. (Det ville også vært menigsløst å si at flere lignende problemer er "det "vanskeligste")

Foreløpig er det ikke funnet noen algoritme som kan løse NPC-problemer i poynomisk tid, men det er heller ikke bevist at en slik løsning ikke finnes. Det er disse problemene dette kapittelet skal gi dere en liten smakebit på.

NP-hard ("NP-vanskelig"): Mengden av de problemene som alle problemer i NP kan reduseres til i polynomisk tid. Disse problemene trenger ikke nødvendigvis ligge i NP. NP-Hard er ikke pensum, men begrepet vil sikkert dukke opp, og vi ønsker å unngå forvirring.

9.1 SAT-problemet

Men hvordan vise at et problem er NPC? Det høres i utgangspunktet umulig ut, men heldigvis finnes det teoremer som sier at SAT-problemet er NPC. Det vi må vise er at alle problemer i NP kan reduseres til SAT-problemet i polynomisk tid. I dette kurset aksepteres beviset helt uten videre. For spesielt interesserte står beviset i [1].

Når vi har et NPC-problem, er det enkelt å vise at andre problemer er NPC. Det holder å vise at vi kan transformere, i polynomisk tid, et kjent NPC-problem til vårt problem. Vi tenker gjerne at vi viser at vårt problem er minst like vanskelig som det kjente NPC-problemet.

Men før vi gir konkrete eksempler på slike beviser, vil vi presentere noen kjente NPC problemer. Vi begynner selvsagt med SAT-problemet.

Vi har en bolsk krets eller et bolsk nettverk med AND, OR og NOT porter. SAT-problemet ("the circuit-satisfiability problem") går kort ut på å finne ut om det eksisterer et sett med bolske uttrykk (0-ere og 1-ere) som gjør at outputen blir 1 (true).

9.2 Andre Kjente NPC-problemer

SAT-problemet er ikke veldig enkelt å jobbe med. Det er ofte enklere å sammenligne problemet med andre kjente NPC-problem. Vi skal presentere noen for dere her. Hvis dere blir spurt om å vise at et problem er NPC, kan dere bruke en av disse å sammenligne med.

Lengste vei: Vi har sett på algoritmer som finner korteste vei i polynomisk tid, men å finne lengste vei er verre. Det er nemlig et NPC-problem.

Klikk: En klikk er en komplett subgraf av grafen G. Dvs dersom en delmengde av nodene i G har forblindelse til alle andre i denne delmengden, har vi en klikk (alle kjenner alle i en klikk). Størrelsen på klikken er antall noder som er med. Spørsmålet "Eksisterer det en klikk av en gitt størrelse k "gir oss et NPC-problem.

Hamiltonsykel: Vi kan finne en Euler-sti (Kantene benyttes en og bare en gang når vi traverserer/går gjennom grafen, uten å hoppe.) i polynomisk tid, men å bestemme om en graf har en Hamilton sykel er et NPC-problem. (I en Hamiltonsykel besøkes nodene en og bare en gang langs en sammenhelgende vei gjennom grafen, dvs vi har en enkel sykel gjennom alle nodene i grafen.)

Travelling Salesman Problem (TSP): Vi har en handelsmann som skal besøke n byer. Han skal bare innom hver by en gang og ønsker å gjøre rundturen så billig som mulig. Poenget her er altså å finne en minimal Hamiltonsykel. Eller for å være helt korrekt er

problemet å finne ut om det eksisterer en rute som er $\leq$ en gitt verdi.

9.3 Hvordan Bevise at et Problem er P, NP, NPC?

For å bevise at et problem er med i P (dvs kan løses i polynomisk tid), må du finne en algoritme som løser problemet i polynomisk tid.

For å bevise at et problem er med i NP, må du finne en algoritme som tester om en gjettet løsning er korrekt. Denne må kjøre i polynomisk tid.

For å bevise at et problem er med i NPC, må du vise at det er med i NP, og at det er minst like vanskelig som et annet problem du vet er i NPC. Det finnes bøker med lister over kjente NPC-problemer. Her kan du finne noen [3]. Måten du gjør det på er å transformere det kjente problemet til ditt problem eller en enklere utgave av ditt problem. Transformasjonen må kunne gjøres i polynomisk tid. Det er på tide med et eksempel.

Eksempel 9.3.1. Bevis for at TSP er NPC
Vi må først vise at TSP er med i NP. Problemet er å finne ut om det eksisterer en rute som er $\leq$ x km. På mystisk vis gjettes vi på det beste veivalget. Det er innlysende at vi med letthet (dvs i poynomisk tid) kan finne ut om denne veien er kortere eller like lang som x. (Vi trenger bare å slå opp avstandene og addere de sammen.) Dette viser at TSP er med i NP.

Nå kan vi vise at TSP er med i NPC. Vi tar utgangspunkt i Hamiltonsykel-problemet. La oss si at nodene i grafen er byer og kantene er veier. Vi setter kostnaden på kantene til å være 1. Eksisterer det en tur som har lengde $\leq$ antall noder? Dette er et Travelling Salesman Problem"av enkleste sort. Poenget er at spørsmålet kunne like gjerne vært: Finnes det en Hamilltonsykel i grafen? Dette er et problem vi gikk ut i fra er NPC.

Av sammenligningen skjønner vi at dersom vi finner en algoritme for å løse vårt problem (TSP), vil denne også kunne løse Hamiltonsykel-problemet. Altså kan vi konkludere med at TSP er NPC.

❑

Kapittel 10

LINEÆRPROGRAMMERING

10.1 Enkel lineærprogrammering

Lineærprogrammering er spesielt nyttig i det som kalles **optimeringsteori**. Dette går ut på å enten minimere eller maksimere en lineær funksjon med en hel del krav.

Eksempel 10.1.1. Oppnå ståkarakter
Anta at Gløshaugen har komponert et nytt fag som kalles ***"part.kont"*** *som består av 50% partikkelfysikk og 50% kontorlandskap. Dette faget er obligatorisk for deg, og du skal levere to semesteroppgaver, en for hver del. Det gis karakter for begge semesteroppgavene, og for å bestå faget må du ha fått ståkarakter på begge. Men fordi du synes dette faget er tullete og du har mye annet å gjøre, ønsker du å bruke så lite tid på faget som mulig samtidig som du skal bestå. La oss kalle tiden du bruker på partikkelfysikk for x_1, og tiden på kontorlandskap for x_2. Du vet at det vil være to sensorer, en tørr professor og en økonom. Du regner med at du får 5 poeng per arbeidstime fra den tørre professoren og 2 fra økonomen for oppgaven i partikkelfysikk, mens for kontorlandskap får du 1 poeng per arbeidstime fra den tørre professoren og økonomen gir 4. Vi oppsummerer dette i en tabell.*

Tabell 10.1: Tabell over poeng pr arbeidstime

	Tørr professor	Økonom
Partikkelfysikk	5	2
Kontorlandskap	1	4

Poenget er nå at for å stå må du få minst 40 poeng for hver oppgave. For å sikre deg litt, krever du at du skal ha stått hos begge sensorene. Da kan vi sette opp et ligningsystem som ser slik ut:

Minimér:

$$x_1 + x_2$$

Hvor:

$$5x_1 + x_2 \geq 40$$

$$2x_1 + 4x_2 \geq 40$$

$$x_1, x_2 \geq 0$$

Den siste ligningen viser bare at tiden vi bruker nøvendigvis er positiv. Dette er et lignings-ystem som kan løses med lineærprogrammering.

❑

For å løse ligningen i 10.1.1, kan man bruke ganske enkel matematikk fra Matematikk 2 (TMA4105), men når vi får sinnsykt mange variable trengs det nye metoder!

Når man skal løse et problem av den typen vi så i eksempel 10.1.1(men ofte med mange fler variable!), så brukes det som kalles **simplex-metoden**. Den tar først og oversetter alle probleme til sin egen standardform, slik at det generelt kan skrives som:

Maksimér

$$\sum_{j=1}^{n} c_j x_j$$

hvor

$$\sum_{j=1}^{n} a_{ij} x_j \leq b_i \qquad \text{for} \qquad i = 1, 2, ...m$$

$$x_j \geq 0 \qquad \text{for} \qquad j = 1, 2, ..., n$$

Som man kanskje aner, så er det ekstremt nyttig å løse slike former for problemer. Vi skal ikke her vise hvordan simplex-metoden fungerer, men interesserte anbefales å se på dette.

10.2 Å formulere problem som lineære program

Med en gang man kan skrive om et problem til et lineært program, finnes det en rekke glupe algoritmer for å løse dette på. Simplex-metoden er en måte å gjøre det på, men den er gammel og idag brukes langt mer raffinerte metoder. Vi skal her vise lett to problemer vi har vært borte i som kan oversettes til til lineærprogrammering.

10.2.1 Korteste vei

For å forenkle litt, ser vi på et én-til-én-problem.

Vi har en graf $G = (V, E)$ med kostnader på kantene. Som vanlig kaller vi kildenoden for s, og sluk-noden for t. Vi ønsker å regne ut kostnaden til den korteste veien fra s til t, og kaller denne verdien $d[t]$. For å kunne bruke lineærorogrammering, trenger vi å finne en mengde variable og krav til disse som definerer når vi har en kosrteste vei fra s til t. Og det er faktisk akkurat det **Bellman-Ford** gjør! Når algoritmen avsluttes har den regnet ut en verdi $d[v]$ for enhver node v slik at for enhver kant $(u, v) \in E$, så har vi $d[v] \leq d[u] + w(u, v)$ hvor $w(u, v)$ er kostnaden til kanten mellom u og v. Kildenoden vil selvsagt alltid ha verdien $d[s] = 0$. Da kan vi skrive problemet på formen:

Minimér $d[t]$

Hvor

$$d[v] \leq d[u] + w(u, v) \quad \text{for enhver kant} \quad (u, v) \in E,$$
$$d[s] = 0$$

Her vil vi ha like mange variable som det er noder. Antall krav blir $k + 1$, et krav per kant samt kravet om at kildenoden alltid har verdi 0.

10.2.2 Maks flyt

Husk at en maksimal flyt er en flyt som tilfredstiller tre spesifikke krav, og som har maksimal flytverdi. En flyt tilfredstiller de "lineære kravene", og verdien til flyten er en lineær funksjon. Se på definisjonen i kapittel 7.1. Maks-flyt-problemet kan nå skrives som:

Maksimér

$$\sum_{v \in V} f(s, v)$$

hvor

$$\begin{aligned} f(u, v) &\leq c(u, v) \quad \text{for alle} \quad u, v \in V \\ f(u, v) &= -f(v, u) \quad \text{for alle} \quad u, v \in V \\ \textstyle\sum_{v \in V} f(u, v) &= 0 \quad \text{for alle} \quad u, v \in V - \{s, t\} \end{aligned}$$

Og nå kan dette problemet løses med flotte algoritmer!