AD Noter

Magnus Goltermann (xzb187)

March 2023

Indhold

1 Beviser

1.1 Direkte bevis

Bevis ved blot at komme frem til sætningen ved brug af fx simpel algebra.

1.1.1 Eksempel

Hvis n er lige, så er n^2 lige:

$$n^2 = (2a)^2 = 4a^2 = 2\left(2a^2\right)$$

1.2 Bevis ved kontraposition

I stedet for at bevise at p medfører q ($p \to q$), så er det ækvivalent at bevise den negerede version: $\sim q \to \sim p$.

1.2.1 Eksempel

Hvis n er lige, så er n^2 lige, negeret: Hvis n er ulige, så er n^2 ulige. et ulige tal er defineret ved $n = 2a - 1$:

$$n^2 = (2a-1)^2 = 4a^2 - 4a + 1 = 2\left(2a^2 - 2a\right) + 1$$

1.3 Bevis ved modstrid

For at bevise S, så antager vi først at det omvendte er korrekt, $\sim S$. Hvis $\sim S$ fører til noget umuligt, så er S korrekt.

1.3.1 Eksempel

Bevis at $\sqrt{2}$ er et irrationelt tal. Det omvendte er derfor, at $\sqrt{2}$ er et rationalt tal, som kan skrives ved $\frac{a}{b}$, som er en fuldt reduceret brøk, og derved kan a og b ikke begge være lige (for så kan der blot divideres med 2 for at reducere yderligere).

$$\sqrt{2} = \frac{a}{b} \iff 2 = \frac{a^2}{b^2} \iff 2b^2 = a^2$$

Og derved er a^2 et lige tal ($a = 2c$) og b et ulige tal.

$$2b^2 = (2c)^2 = 4c^2 \iff b^2 = 2c^2$$

Fra før vidste vi at b var ulige, men b^2 er $2c^2$ og derfor lige (kommer fra, at hvis n er lige så er n^2 også lige). Dette er umuligt, og derfor er det modsatte bevist, og $\sqrt{2}$ er derfor irrationel.

1.4 Bevis ved induktion

Givet et udsagn $p(n)$. Hvis $p(b = 1)$ er korrekt, så antag at $p(n)$ er korrekt, og så lav induktion trin $p(n + 1)$.

1.4.1 Eksempel

$p(n) = 1 + 2 + 3 + ... + n = n(n + 1)/2$.

$$p(1) = 1 = 1 \cdot 2/2 = 1$$

Vi antager derfor at $p(n)$ er korrekt, og laver et induktions trin:

$$p(n + 1) = 1 + 2 + 3 + ... + n + n + 1 = (n + 1)(n + 2)/2$$

$$\iff$$

$$n(n + 1)/2 + n + 1 = (n + 1)(n + 2)/2$$

Herfra vises at venstre og højre siden er lig. Der kan stoppes når det er trivielt, at de er ens.

1.4.2 Stærk induktion

Vis at $p(b)$ holder, $p(b + 1)$ og op til n, så holder $p(n)$ for alle $n \geq b$

1.5 Bevis af algoritme

Når man skal bevise en algoritme er korrekt, skal der bevises at den altid er korrekt, og at den altid vil stoppe på et tidspunkt. At den er korrekt vises ved en invariant, som er en del af problemet, som ved hver nye iteration altid er korrekt. Fx ved insertion sort, så er det de første i elementer, da de altid er sorteret. Helt formelt kan det sættes op i:
Initielt: Er invarirant korrekt til at starte med?
Løbende: Er invariant korrekt igennem algortimen?
Til sidst: Er invarianten korrekt i sidste iteration?

2 Tidskompleksitet og asymptotisk notation

For at bestemme tidskomplexiteten, ser vi oftest på hvor hurtig algoritmen er asymptotisk, herunder værste, bedste og gennemsnitlig køretid. Herfra kan man bestemme den øvre/nedre grænse asymptotisk, og heraf benytte asympototisk notationerne Ω (nedre), O (øvre) og Θ ("oppe og nedre").

2.1 O notation

Hvis man siger at $f(n)$ er $O(g(n))$ angiver det, at asymptotisk, så vil $g(n)$ (ganget med en konstant) altid være større eller lig $f(n)$. Formelt som

$$O(g(n)) = f(n)|0 \leq f(n) \leq cg(n) \forall n \geq n_0$$

2.2 Ω notation

Samme som ved O notation, blot begrænset nedadtil. Dvs. Asymptotisk er en funktion $f(n)$ begrænset nedadtil af $cg(n)$

$$\Omega(g(n)) = f(n)|0 \leq cg(n) \leq f(n) \forall n \geq n_0$$

2.3 Θ notation

Når $f(n)$ er begrænset opadtil og nedadtil af samme $c_1 g(n)$ og $c_2 g(n)$

$$\Theta(g(n)) = O(g(n)) = \Omega(g(n)) = f(n)|0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n) \forall n \geq n_0$$

2.4 Lille o notation

Hvis den er skarpt begrænset opadtil. Dvs. samme som store O, men uligheden er skarp, og det skal gælde hvor enhver c større end 0.

$$\Omega(g(n)) = f(n)|0 \leq cg(n) \leq f(n) \forall n < n_0, \forall c > 0$$

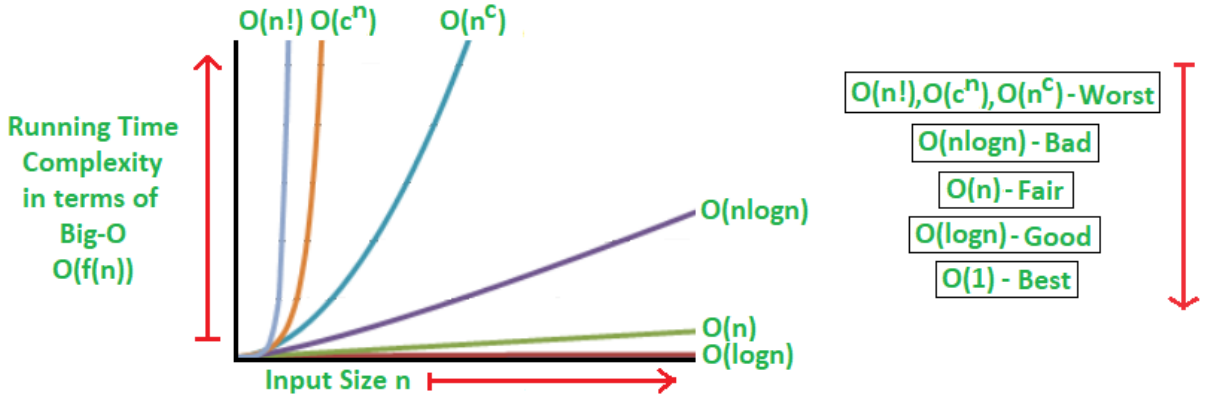
2.5 Lille ω notation

Hvis den er skarpt begrænset nedadtil. Dvs. samme som store Ω, men uligheden er skarp, og det skal gælde hvor enhver c større end 0.

$$\Omega(g(n)) = f(n)|0 \leq cg(n) < f(n) \forall n \geq n_0, \forall c > 0$$

2.6 Ekstra regler

$$n! = nlog(n)$$
$$n^x > log(n), x > 0$$
$$log(n)^3 = log(n + n + n) = O(log(n))$$
$$x^{log(n)} > nlog(n)$$
$$n^x < y^{log(n)}$$



Figur 1: Hierarki for asymptotisk notation

3 Del og hersk (rekursion)

En del og hersk algoritme deler problemet op i små delproblemer, som herefter løses rekursivt ved enten igen at dele op eller ved brute force. Når delproblerme er løst, så kombineres de til den fulde løsning.

3.1 Merge sort

Merge sort opdeler listen rekursivt, indtil længden af listen er 1. Herefter benyttes merge til at flette alle listerne sammen.

3.1.1 Merge hjælpefunktion

Merge fletter to sorterede lister samme til én sorteret liste i $O(k_1 + k_2)$.

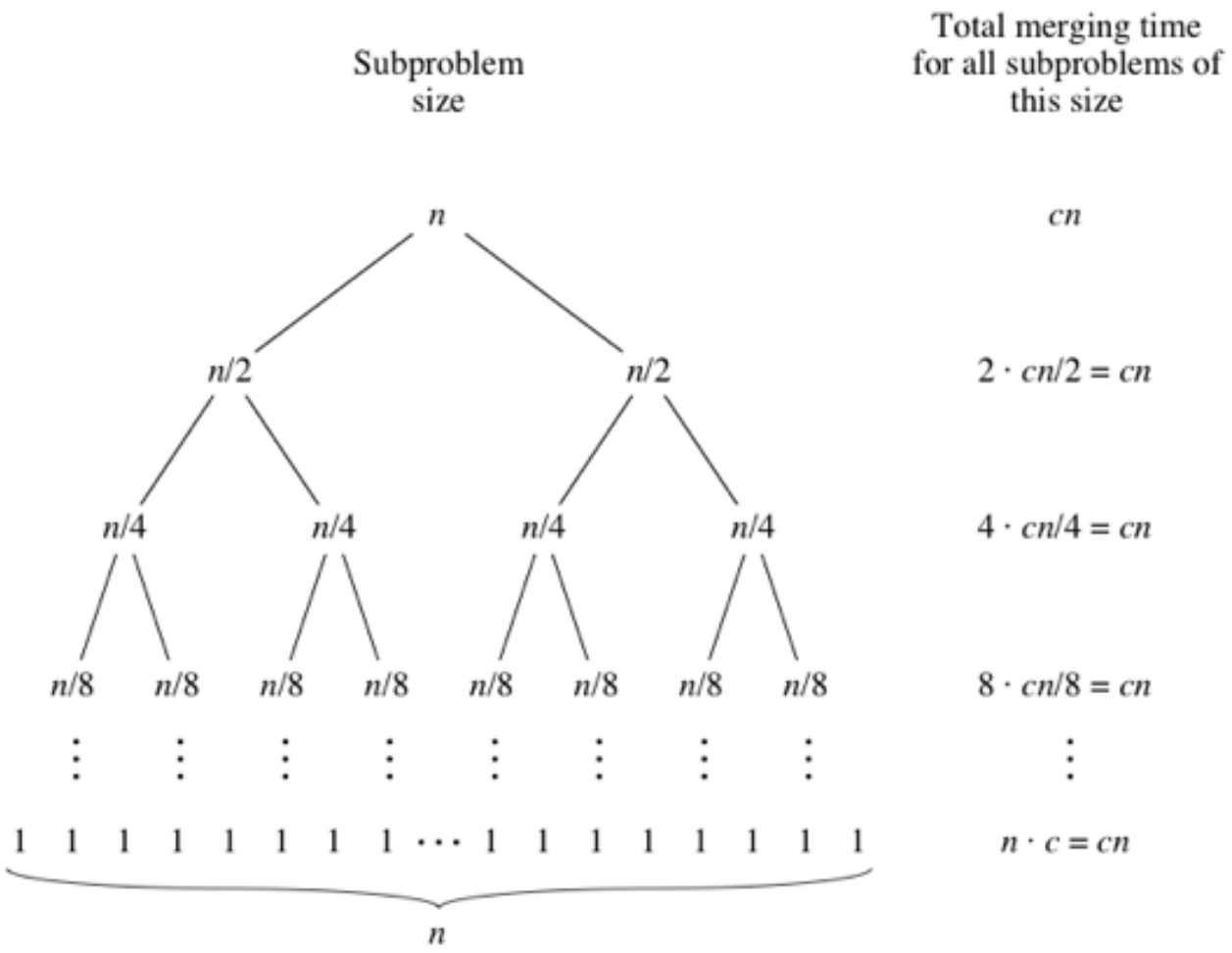
3.2 Rekursionsligning

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1 \\ T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + \Theta(n) & \text{if } n > 1 \end{cases} \tag{1}$$

Her ses den rekursive del af ligningen, at den hver kald deler listen op i to lige store dele, som ved hver kald har en "cost"på $\Theta(n)$. Som kan omskrives til

$$T(n) \leq 2T(n/2) + cn$$

For at bestemme køretiden ud fra rekursionsligningen kan et rekursionstræ opstilles (figur 2).



Figur 2: Merge sort rekursionstræ

For at opstille et rekursionstræ, så skal være knude have omkostningen i sig, og opdeles i nye knuder ift. de rekursive kald der er defineret. For merge sort deles hver knude op i yderligere 2 nye knuder, som hver har halvdelen af den forrige, samt en omkostning på n, som følge af halveringen også bliver halveret ved hvert kald.
Herefter summeres omkostningerne for hver række, som ved sidste niveau blot bliver cn. Herefter bestemmes højden af træet, som er $log(n)$, da den halveres hver gang. Og derfor gætter vi på en køretid på $O(nlog(n))$

3.3 Substitutionsmetoden

3.3.1 Gæt

Først gættes på en øvre grænse. Oftest ved brug af fx rekursionstræ.

3.3.2 Induktionsbevis

Først antages tidsomkostningerne for når n nærmer sig 0, fx $T(n) = 1$ når $n = 1, 2..., k$.
Herefter sættes ligningen op $T(n) = O(\text{gæt(n)})$, som føre til at vi skal vise $T(n) \leq$ gæt(n) for en given $n \geq n_0$ og for en konstant $c > 0$ og $n_0 > 0$.
Basecase:
For at opstille et basecase, bliver vi nødt til at være sikre på, at vi ikke ender i $T(k)$, som var "predefineret"omkostninger. Derfor laver vi stærk induktionstrin, og løser derfor manuelt de første par n-værdier, så vi er sikre på de aldrig kommer ned til k.

Induktionstrin:
Herefter substitueres den rekursive opdeling ind i gættet, som herefter skal bevises er mindre eller lig det oprindelige gæt.

3.3.3 Merge sort bevis

Vi har vores gæt fra tidligere $O(nlog(n))$, samt at $T(1) = 1$. Og derfra ligningerne

$$T(n) = 2T(\lfloor n/2 \rfloor) + n, n > 1 \tag{2}$$

$$T(n) \leq cnlog(n) \tag{3}$$

Basecase:
Da ligningen kun er defineret når $n > 1$, bliver vi nødt til manuelt at beregne, så vi er sikre på ikke at få problemer med dem. Og derfor skal vi beregne $n = 2$ pg $n = 3$ ud, da de begge vil give $T(1)$.

$$T(2) = 4 \text{ og } T(3) = 5$$

Hvilket begge er gældende i (3) for $c \geq 2$.
Induktionstrin:
Herfra indsættes rekursionsligningen i gættet, og derfra omskrives (3). Og herfra skal det blot bevises at det er mindre eller lig $cnlog(n)$

$$\begin{aligned} T(n) &= 2T(\lfloor n/2 \rfloor) + n \\ &\leq 2c\lfloor n/2 \rfloor \lg(\lfloor n/2 \rfloor) + n \\ &\leq cn \lg(n/2) + n \\ &= cn \lg n - cn \lg 2 + n \\ &= cn \lg n - cn + n \\ &\leq cn \lg n \end{aligned} \tag{4}$$

3.4 Master metode

Hvis en rekursionsligning har formen

$$T(n) = aT(n/b) + f(n) \tag{5}$$

med $a \geq 1$ og $b > 1$, så kan master metoden bruges.

3.4.1 Case 1

If $f(n) = O\left(n^{\log_b a - \epsilon}\right)$ for a constant $\epsilon > 0$ then $T(n) = \Theta\left(n^{\log_b a}\right)$

3.4.2 Case 2

If $f(n) = \Theta\left(n^{\log_b a}\right)$ then $T(n) = \Theta\left(n^{\log_b a} \lg n\right)$

3.4.3 Case 3

If $f(n) = \Omega\left(n^{\log_b a + \epsilon}\right)$ for some constant $\epsilon > 0$ and if $af(n/b) \leq cf(n)$ for some constant $c < 1$ and all sufficiently large n then $T(n) = \Theta(f(n))$

4 Dynamisk programmering

Ideen med dynamisk programmering er, at når vi udregner et delproblem, så gemmer vi svaret. Således at hvis vi kommer til samme delproblem igen, så har vi allerede svaret.

4.1 Memoization: top down

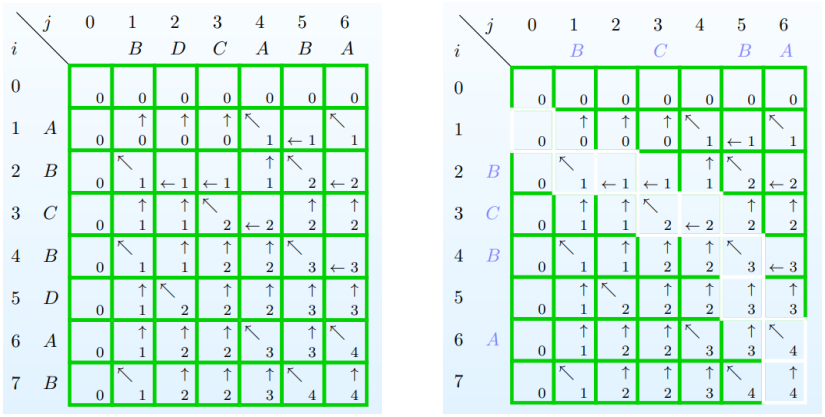
Løs problemet "oppefra"ved rekursion. Gem hver løsning til et subproblem i fx en tabel/array, og benyt den hvis det kommer igen.

4.2 Bottom up

Sorter delproblerme efter størrelse, og løs herefter problemerne efter størrelse først og gem dem i en tabel/array. Da de større problemer oftest kræver løsningerne til de små delproblemer, så kan de bare bruge de tidligere svar. Dårlig løsning, hvis et problem ikke kræver alle mindre delproblemer løst, for så udregner den "for mange"delproblemer.

4.3 Længste fælles delsekvens

Givet to strenge, bestem den længste fælles delsekvens mellem de to strenge. For at læse det, opstilles et array $c[i, j]$ til at have alle løsninger til delproblemer. Herefter udfyldes det array, ved at se på 3 cases
Case 1: $i = 0$ or $j = 0 \Rightarrow c[i, j] = 0$
Case 2: $i, j > 0$ and $x_i = y_j \Rightarrow c[i, j] = c[i-1, j-1] + 1$
Case 3: $i, j > 0$ and $x_i \neq y_j \Rightarrow c[i, j] = \max\{c[i, j-1], c[i-1, j]\}$
For at håndkøre nemt / visualisere det, kan en tabel opstilles, se figur 3.



(a) c arrayet fyldt ud

(b) En løsningen fundet: BCBA

Figur 3: LCS med strengene ABCBDAB og BDCABA. Pilene angiver hvor værdien kommer fra (hvis case 0, vælger den altid bare pil op).

5 Grådige algoritmer

En grådig algoritme løser et problem ved altid at vælge det nuværende bedste valg, og tager altså ikke mindre delproblemer i betragtning.

5.1 Grådige valg egenskab

Et problem har det grådige valg egenskab, hvis der findes en optimal løsning til et problem, som inkluderer det grådige valg. Hvis dette ikke gælder, vil en grådig algoritme ikke finde den korrekte løsning.

5.2 Optimale delproblemer

Hvert grådigt valg vil herefter kunne give en korrekt løsning, og evt. et delproblem som med endnu et grådigt valg igen vil være korrekt og evt. skabe et nyt delproblem. Problemet bliver derfor mindre og mindre i takt med hvert valg.

Hvis vi kan vise at et problem har det grådige valg egenskab og optimale delproblemer, så kan vi bevise at en grådig algoritme vil finde en optimal løsning til et problem.

5.3 Eksempel: Aktivitet valg

Givet en liste af aktiviteter som er sorteret efter slutstidspunkt, angiv hvor mange aktiviteter maksimalt kan planlægge, hvori de ikke overlappe.

5.3.1 Optimalt delproblem

Hvis vi har en optimal løsning til S_d, som er en del af S_A, så vil løsningen til S_A indholde løsningen til S_d. Kort sagt, hvis der kan planlægges 3 aktiviteter i S_d, så vil S_A mindst have 3 aktiviteter plus de andre delproblemers løsninger.

5.3.2 Grådige valg egenskab

Vi vil gerne vælge den aktivitet som slutter først (intuitivt, så vil vi gerne gøre mere plads til andre aktiviteter). Vi angiver den aktivitet som slutter først a_1 for problemet S. Antag der er en anden optimal løsning, hvori a_j er en aktivitet som er før andre aktiviteter, men overlapper med a_1. Her kan a_1 nemt ombyttes med a_j, og stadig være en optimal løsning (se her, at det grådige valg blot skal give en løsning, og ikke altid den eneste).

5.3.3 Grådige algoritme

Da vi har vist optimale delproblemer og det grådige valg egenskab, kan vi nemt angive en algoritme, som altid bare vælger den aktivitet som har det tidligste sluttidspunkt.

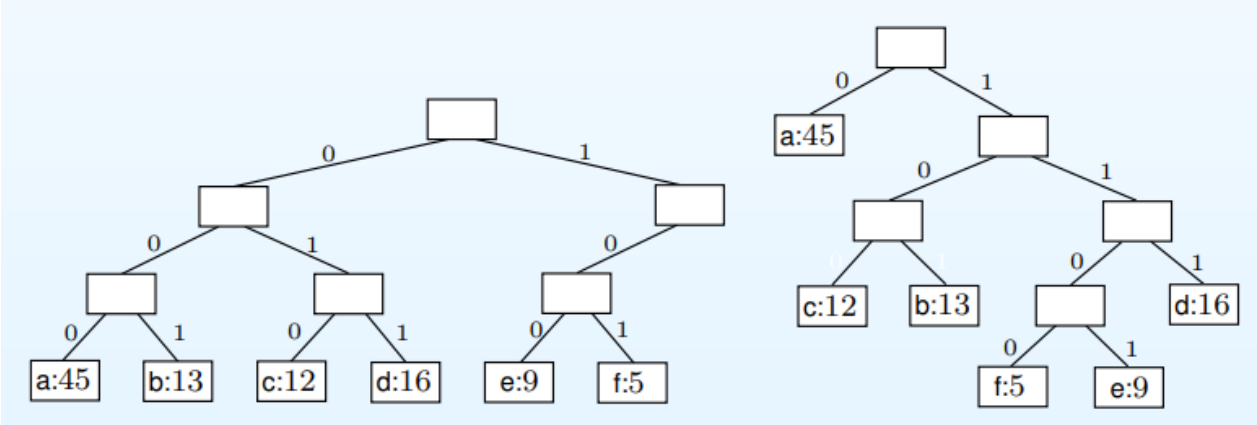
5.4 Huffman codes

Givet en streng, eller en liste med frekvenser af bogstaver der er brugt, angiv her en mere kompakt måde at nedskrive den binært. Givet 6 bogstaver: a, b, c, d, e og f og deres frekvenser, se figur 4.

Letters:	a	b	c	d	e	f
Frequency:	45	13	12	16	9	5
3 bits:	000	001	010	011	100	101
Variable length:	0	101	100	111	1101	1100

Figur 4: Bogstaver og deres frekvens og "kode"

For at gøre parsing deterministisk, så må ingen kode være et prefix for en anden. Fx må vi ikke have både "00"og "001"eller "0"og "01". De tilhørende parse træer kan ses på figur 5



Figur 5: Venstre er det originale parsetræ, til højre er parsetræ for koderne

5.4.1 Omkostning ved parsetræ

Givet en liste af karakterer C, så for hver karakter $c \in C$, så er f_c dens frekvens og $d_T(c)$ dens dybde i parsetræet. Omkostningen for et parsetræ kan derved gives som

$$B(T) = \sum_{c \in C} f_c \cdot d_T(c) \tag{6}$$

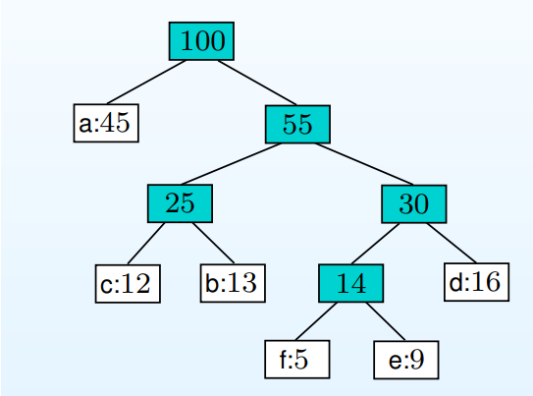
$B(T)$ kan derfor ses som den samlede omkostning for at "gemme"den givne streng. Vi vil derfor gerne gøre $B(T)$ så lille som muligt.

5.4.2 Huffman algoritmen

Givet en liste med bogstaver og deres frekvens, sammensæt da de to med lavest frekvens, og dan en forældre til dem med summen af deres frekvens. Fortsæt indtil alle bogstaver er med i træet, og træet vil da angive en parsekode til hvert bogstav med "0"til venstre og "1"til højre. For at holde styr på mindste element, kan en min-heap bruges (kan laves i n tid og extract min kan bruges i logn tid).

Eksempel

Givet 6 bogstaver og deres frekvens: a:45, d:16, b:13, c:12, f:5 og e:9. Herefter sammensætter vi dem i par, efter lavest frekvens. Fx bliver f og e et par, som får en forældre der er summen af deres frekvens, altså $5 + 9 = 14$. Herefter gøres det samme, men hvor 14 er med i "puljen". Så sammensættes c og b, med en forældre $12+13 = 25$. Så sammensættes d og subtræet med f og e, som får en fælles foreældre på $14+16 = 30$. Så sammensættes træet med c og b med træet for f, e, og d med forældren $25 + 30 = 55$. Som så til sidst sammensættes med a og får den samlede forældre (træroden) til $45+55 = 100$. Se figur 5 for visuelt eksempel.



Figur 6: Eksempel af parsetræ lavet af Huffman algoritmen

5.4.3 bevis

Grådige valg
Optimale delstruktur
Se at omkostningen af et parsetræ kan omskrives til summen af alle knuder (bortset fra roden) angivet som W.

$$B(T) \stackrel{\text{def}}{=} \sum_{c \in C} f_c \cdot d_T(c) = \sum_{v \in W} f_v \tag{7}$$

Vores grådige valg vælger altid de to laveste frekvenser, og danner en ny psdoudo karakter som deres foreældre med frekvens som summen af de to, angivet vec z. Herfra ses det, at antallet af karakterer må falde med 1 ved hver iteration (mister 2 pga. sammensætning og får en pga. z).

6 Amortiseret analyse

Den samlede tidsomkostning for en række operationer, og derved mulighed for at bestemme den gennemsnitlige tidsomkostning.

6.1 Aggrageret analyse

Givet n operationer, så er $T(n)$ den totale worst-case tidsomkostning, hvorfra den gennemsnitlige er $T(n)/n$.

6.1.1 Eksempel: Stack med multipop

Givet en datastruktur (stack), som har push, pop, multipop og is-empty. De er alle $\Theta(1)$ på nær multipop som er $\Theta(\min(s,k))$ hvor s er antal elementer i stacken og k er antal elementer til som skal poppes (fjernes). Hvad er så den gennemsnitlige omkostning for denne struktur.
Se her, at multipop blot er en masse pop operationer, og at der maks kan poppes så mange gange som der er blevet pushed. Dvs. der kan max pushes n gange, og derfra max poppes n gange, så altså i værste fald $O(n)$. Derfra er den amortiserede omkostning $O(n)/n = O(1)$.

6.2 Accounting metoden

Givet en sekvens af n operationer, hvori den i'te operation koster c_i. I accounting metoden benytter vi kunstige omkostninger angivet ved $\hat{c}_i$, som angiver den amortiserede omkostning. Vi tager "for meget"for nogle operationer således at $\hat{c}_i > c_i$ med en mængde på $\hat{c}_i - c_i$, så vi noget i banken til de dyre operationer senere. Omvendt kan vi også betale for lidt for nogle operationer $\hat{c}_i < c_i$ med en mængde på $c_i - \hat{c}_i$, som kan gøres ved at bruge de "kreditor"vi har på hvert objekt for dem vi betalte for meget ved. Vi har derfor uligheden

$$\sum_{i=1}^{n} c_i \leq \sum_{i=1}^{n} \hat{c}_i \tag{8}$$

Vi gerne vil finde en øvre grænse på (8), da det kan være svært at finde en for c_i, gør vi det for $\hat{c}_i$. Det eneste krav er, at vi mindst betaler ligeså meget eller mere i gennemsnit end de originale omkostninger.

6.2.1 Eksempel: Stack med multipop

Angiv da nye omkostninger til operationerne. Således at vi betaler for meget for push med en omkostning på 2, så kan vi trække det fra på pop, så den koster 0. Og da multipop blot we flere pop, så får den også en omkostningen på 0.
Se da, at vi aldrig kan poppe flere end vi har pushed, og derfor vil vi aldrig kunne få negative krediter i alt. Summen for $\hat{c}_i$ som angivet i (8) vil derfor være højest $2n$ (hvis der kun poppes). Hvilket giver os en gennemsnitlig omkostning for en operation til $O(1)$.

6.3 Potentiale metoden

Potentiale metoden gemmer kredit i en "bank", hvori den nuværende antal af krediter kan angives med en potentiale funktion Φ. Omkostningen for en operation er angivet som c_i, og den amortiserede omkostning som $\hat{c}_i$ defineret ved

$$\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1}) \tag{9}$$

Hvis differencen mellem $\Phi(D_i) - \Phi(D_{i-1}) > 0$, så betaler vi for meget, og kommer kredit ind i banken. Hvis omvendt $\Phi(D_i) - \Phi(D_{i-1}) < 0$, så betaler vi for lidt, og trækker derfor kredit ud af banken. Igen gælder (8). Vi skal derfor vise, at

$$\sum_{i=1}^{n} \hat{c}_i = \left(\sum_{i=1}^{n} c_i\right) + \Phi(D_n) - \Phi(D_0) \geq \sum_{i=1}^{n} c_i \tag{10}$$

gælder. Her er D_n den sidste operation og D_0 er den første operation. Hvorfor vi skal vise, at $\Phi(D_n) - \Phi(D_0)$ aldrig kan være negativ. Oftest vælges $\Phi(D_i) = 0$, og $\Phi(D_i) \geq 0$ for alle i, således den aldrig kan blive negativ.

6.3.1 Eksempel: Stack med multipop

D_0 er vores initielle stack og D_i er stacken efter den i'te operation. Vi angiver $\Phi(D_i)$ til at være antal elementer i stacken D_i, da den opretholder $\Phi(D_0) = 0$. Dvs. $\Phi(D_i)$ angiver antal elementer når operationen er lavet, og $\Phi(D_{i-1})$ angiver antal elementer lige før operationen blev udført.
Herfra kan vi gennemgå de forskellige operationer:
Push

$$\Phi(D_i) - \Phi(D_{i-1}) = 1 \tag{11}$$

$$\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1}) = 1 + 1 = 2 \tag{12}$$

Pop:

$$\Phi(D_i) - \Phi(D_{i-1}) = -1 \tag{13}$$

$$\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1}) = 1 - 1 = 0 \tag{14}$$

Multipop

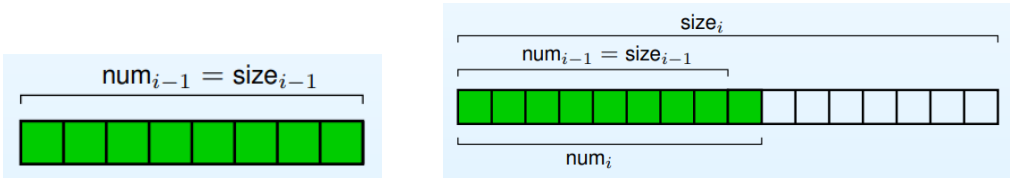
$$\Phi(D_i) - \Phi(D_{i-1}) = -k' \tag{15}$$

$$\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1}) = k' - k' = 0 \tag{16}$$

Hvor k' er antallet af elementer den popper.
Her ses det, at alle elementer er mindst 2, og derfra er summen af alle operationer mindst 2n, hvilket giver vores gennemsnitlige amortiserede køretid til $O(n)/n = O(1)$

6.4 Dynamisk tabel

En dynamisk tabel T er en abstrakt datatype, som har operationerne insert og delete (begge $O(n)$). T er tom til at starte med, og hukommelse som T bruger er gemt som et array af slots. Ideen med dynamisk er, at den plads den tager i hukommelsen ændrer sig i takt med den bliver fyldt/tømt. num_i angiver antal elementer i T efter den i'te operation og $size_i$ angiver den størrelse som T tager op efter den i'te operation. $\alpha_i = \frac{num_i}{size_i}$, som kaldes load factor, og er hvor stor en andel af T der er fyldt op efter den i'te operation.
Tabel udvidelse sker når $num_{i-1} = size_{i-1}$ (eller $\alpha_{i-1} = 1$), så bliver $size_i = 2sizei - 1$. altså når tabellen bliver fyldt ud, så allokerer vi et nyt array der er dobbelt så stort, se figur 7.



(a) Arrayet fyldt op

(b) Arrayet efter udvidelse

Figur 7: Eksempel på dynamisk tabel udvidelse.

6.4.1 Amortiseret analyse for insert

En operation der kræver udvidelse af tabellen er $c_i = num_i$ og eller bare $c_i = 1$. Vi vil nu bestemmeden amortiserede køretid ved brug af potentiale metoden.
Lad D_0 være den initielle tomme tabel T, og lad D_i være T efter den i'te opdatering.
Givet en potentiale funktion

$$\Phi(D_i) = 2num_i - size_i \tag{17}$$

Først er det vigtigt at $\Phi(D_0) = 0$ og $\Phi(D_i) \geq 0$. Det er klart at $\Phi(D_0) = 0$, da vi starter med T tom. Og siden T altid er mindst halv fyldt må $\Phi(D_i) \geq 0$ også gælde. Og da den derfor er gyldig, så må vores amortiserede omkostning altid være større end vores "almindelig", og derfor vil (10) gælde.

Herfra kan vi bestemme en øvre grænse, da den så vil begrænse c_i. Vi har 3 cases:
Ingen tabeludvidelse:

$$\begin{aligned}\hat{c}_i &= c_i + \Phi(D_i) - \Phi(D_{i-1}) \\ &= 1 + (2 \text{ num }_i - \text{ size }_i) - (2 \text{ num }_{i-1} - \text{ size }_{i-1}) \\ &= 1 + (2 \text{ num }_i - \text{ size }_i) - (2(\text{ num }_i - 1) - \text{ size }_i) \\ &= 3\end{aligned} \tag{18}$$

Tabel udvides og $i = 1$:

$$\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1}) = 2 \tag{19}$$

Tabel udvides og $i > 1$**:**

$$\begin{aligned}\hat{c}_i &= c_i + \Phi(D_i) - \Phi(D_{i-1}) \\ &= \text{num}_i + (2\,\text{num}_i - \text{size}_i) - (2\,\text{num}_{i-1} - \text{size}_{i-1}) \\ &= \text{num}_i + (2\,\text{num}_i - 2(\text{ num }_i - 1)) - (2(\text{num}_i - 1) - (\text{num}_i - 1)) \\ &= 3\end{aligned} \tag{20}$$

I udledningen ses det at $size_i = 2(num_i - 1)$ og at $num_{i-1} = num_i - 1$ for at få det til kun at afhænge af num_i, så det hele går ud og giver 3.

6.4.2 Amortiseret analyse for delete

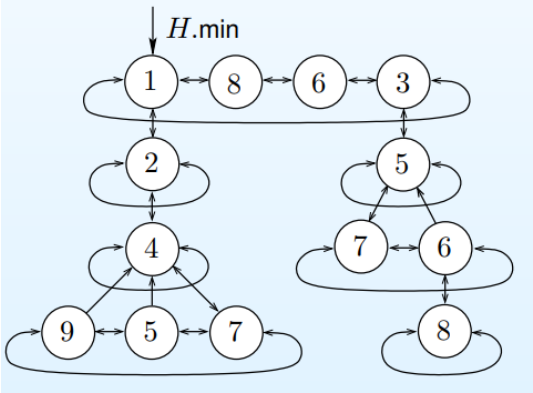
Hvis vi følger samme princip når vi mindsker tabellen, ved at gøre det når den er halv fyldt, så får vi et problem i vores amortiserede analyse (negativt potentiale). Vi bliver nødt til først at halvere tabellen når det er en fjerdel fyldt, altså når $\alpha = \frac{1}{4}$, og så omformulerer vores potenitale funktion til en stykvis funktion når vi gør tabellen mindre:

$$\Phi(D_i) = \begin{cases} 2 \text{ num }_i - \text{ size }_i & \text{if } \alpha_i \geq 1/2 \\ \text{size}_i/2 - \text{ num }_i & \text{if } \alpha_i < 1/2 \end{cases} \tag{21}$$

Det kan vises at $\hat{c}_i \geq 3$, og derfor får vi en amortiseret gennemsnitlig køretid på $O(1)$ for både delete og insert. Se fulde bevis i slides.

6.5 Fibonacci Heaps

Et fibonacci heao er en samling a træer, som alle har min-heap egenskab, dvs. enhver forældre er mindre eller lig dens børn. Dvs. det mindste element i hele heapen må være blandt rodknuderne.
Alle rodknuder er forbundet cirkulært med en dobbelt linked liste. Hver knude har en pointer til dens forældre. Hver knude har kun en pointer til et barn, og søskende knuder er så forbundet cirkulært med en dobbelt linked liste. Der er en pointer H.min til det mindste element. Se figur 8 for visuelt.



Figur 8: Eksempel på fibonacci heao

Hver knude har 3 properties: x.key (dens værdi), x.deg (antal børn) og x.mark (hvis den er markeret).

6.5.1 Potentiale funktion

For at beregne den amortiserede køretid, er potentiale funktionen angivet til

$$\Phi(H) = t(H) + 2m(H) \tag{22}$$

$t(h)$ angiver antal rødder i rodlisten. $m(H)$ angiver antal markerede knuder.

6.5.2 Insert

Sæt blot knuden fast på roden.

6.5.3 Extract min

Fjern roden der har pointeren H.min. Sæt herefter alle børn op på roden, og kør consolidate for at "rydde op". Og returner min knuden.

6.5.4 Consolidate

Lad $D(n)$ være det maksimum antal børn for enhver knude. Consolidate laver herefter et array med NIL pointers $A[0, ..., D[n]]$, men n som antal knuder i heapen (H). Herefter benytter den A til at finde rodknuder med samme antal børn (degree / x.deg). To træer/rodknuder r_1 og r_2 som har samme x.deg, bliver sat sammen ved at sætte r_1 som et barn af r_2 således at $key(r_1) \geq key(r_2)$. Når alle rodknuder med samme antal børn er sammensat, så er der ingen to rodknuder med samme antal børn. Rodknuderne ses herefter igennem for at finde en evt. ny H.min.

6.5.5 Markerede knuder

Alle knuder starter som umarkerede. Hvis en knude får en forældre i consolidate, så bliver den umarkeret. Hvis en knude mister et barn, så bliver den markeret. Hvis en knude der allerede er markeret mister endnu et barn, så rykkes den op som rod, og sættes som umarkeret. "Når en knude mister sit første barn, bliver den ked ad det, og derfor markeret. Når en knude mister sit andet barn, så starter den forfra."

6.5.6 Decrease key

Decrease key reducerer x.key for en knude, men det giver problemer med min heap egenskaberne. Dvs. hvis den nye værdi er større end sin forældre, så gør den ikke mere. Men hvis den er mindre, bliver knuded sat op som rod, og foreældren bliver markeret. Flere knuder kan rammes af dette, som kaldes "cascading cut", hvori flere knuder bliver påvirket op igennem træet af en decrease key.

6.5.7 Union

Union slår blot rodknuderne sammen for de to træer, hvilket nemt kan gøres, da det blot er en dobbelt linked liste, så ændres pointerne bare i enderne. Og H.min sættes til den mindste fra de to træer.

6.5.8 Delete

Sæt x.key til minus uendelig, og kør herefter extract min.

7 Binære søgetræer

Et søgetræ S, er en datastruktur som har nogle objekter med en nøgle. Den har følgene operationer:

- acces(k, S): returnerer objekt med nøgle k

- insert(i, S): indsætter objekt i ind i S
- delete(k, S): sletter objekt med nøglen k i S
- make(): returnerer et nyt tomt S

Alle nøgler er unikke, og et objekt kan ikke tilgås direkte.
Fra en knude er alle knuder til venstre mindre og alle til højre større. Hver knude har to pointere L(x) til knuden til venstre og R(x) til den til højre.

7.1 Acces(k,S)

Sammenlign k med knuden, hvis den er mindre gå til venstre, hvis større gå til højre. $O(n)$

7.2 insert(i, S)

Lad som om vi søger efter i, og så indsæt når vi når et blad. $O(n)$

7.3 delete(k, S)

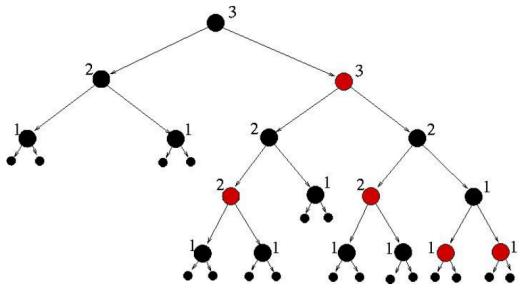
Hvis k ikke har børn slettes den blot. Hvis den har et barn, så hægtes barnet bare på forældren i stedet for k. Hvis k har 2 børn, så skal en direkte forgænger findes. Forgængeren skal være mindre end k, og derfor går vi et skridt ned til venstre. Men da forgængeren også skal være større end den knude, går vi så lang mod højre som muligt for at finde den største i dette undertræ. Den knude bytter vi om med k. Dvs. Vi går et skridt til venstre, så lang til højre som muligt, bytter den med k. Omvendt kunne dens fortsætter også findes, som er en gang til højre, og så til venstre så langt som muligt, og byt den med k. Når der er byttet rundt, så kan k blot fjernes, og dens barn hægtes opad. $O(n)$.

7.4 Rød-sorte binære søgetræer

Vi vil gerne opnå et balanceret binært søgetræ, og derfor reducere højden fra $O(n)$ til $O(log(n))$, og derfor laver i et rød-sort binært søgetræ. Herfra laver vi nogle forskellige knuder.

- Dummy knuder: Hvis en knude ikke har 2 børn, så tilføjer vi dummy knuder så den har 2.
- Røde knuder: Hver rød knude har 2 sorte knuder som børn

Roden skal altid være sort. Antal af sorte knuder man møder for at nå et blad i træet vil være det samme for alle blade. Den sorte højde angives som $bh(x)$. Se figur 9.



Figur 9: Rød-sort binært søgetræ

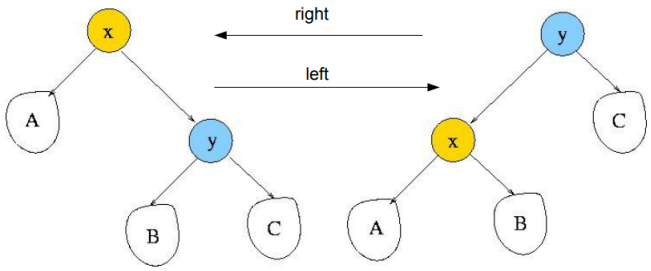
7.4.1 Bevis for højde af rød-sort træ

7.4.2 Access

Samme som med almindeligt søgetræ, dog er højden nu begrænset og derfor $O(log(n))$

7.4.3 Venstre/højre rotation

For at rebalancere et træ, kan rotationer være nødvendigt. Se figur 10.

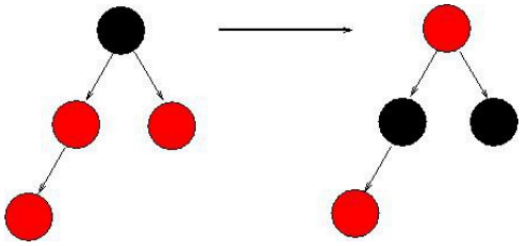


Figur 10: Venstre og højre rotation

Hvis vi ser på venstre rotationen, så kan det ses at x bliver et barn af y til venstre for, da y må være større end x. A forbliver på x, da den er mindre end alt andet. B var til højre fra x til at starte med, og bliver derfor sat som barn til højre fra x. C forbliver på y til højre, da den startede der og er størst. Det omvendte gælder den anden vej med højre rotation.

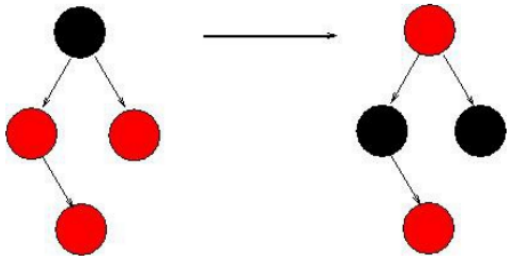
7.4.4 Insert

Vi indsætter på samme måde som almindeligt søgetræ. Men herefter kan der opstå ubalance i den sorte højde, eller med en rød knude der får endnu et rødt barn (venstre eller højre), vi har derfor 4 cases. $O(log(n))$
Case 1:
Hvis vi har to røde efter hinanden til venstre, kan vi ændre farven er midter knuderne til sorte, og den øverste til rød. Dette kan dog propergere opad, da den øverste forældre i dette delproblem også kan være rød. Vi fortsætter med at farve de midterste sorte, indtil forældren til den øverste er sort, eller vi møder roden (hvis vi møder roden, skal den farves sort). Dvs. vi enten løser problemet på stedet, eller flytter det opad. Se figur 11.



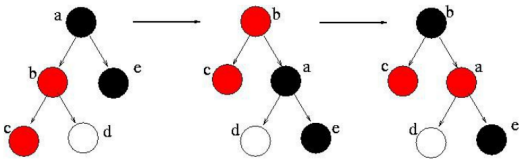
Figur 11: Case 1: 2 røde knuder efter hinanden med venstre barn

Case 2:
Samme løsning som case 1. Se figur 12.



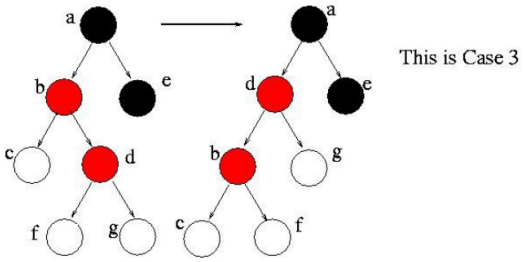
Figur 12: Case 2: 2 røde knuder efter hinanden med højre barn

Case 3:
2 røde efter hinanden mod venstre, men med en sort onkel. Dette kan løses lokalt, og behøver altså ikke propergere opad. Se figur 13, da den øverste igen ender med at være sort.
Først højre roteres a og b. Så bytter de farve, så b bliver rød og a sort.



Figur 13: Case 3: 2 røde knuder efter hinanden mod venstre, men med en sort onkel.

Case 4: 2 røde efter hinanden mod højre. Se figur 14. Her ses det, at vi blot lavet er venstre rotation på b og d, hvorfra vi får svarende til hvordan case 3 var, som vi kender løsningen til.



Figur 14: Case 4: 2 røde knuder efter hinanden mod højre, men med en sort onkel.

7.4.5 Delete

For at slette og bibeholde balance er der i alt 8 cases, men det generelle koncept nævnet kun. Vi gør som i almindelig delete. Dvs. vi finder dens direkte efterfølger, bytter dem og fjerner.
Hvis efterfølgeren (b) er:

- rød: gør ikke noget
- sort og L(b) rød: gør L(b) sort
- sort og L(b) sort: lav L(b) fed (tæller for 2 sorte) og propager.

Det vigtigste er at bibeholde højden og balancen. Alle de 8 cases er kun vigtige hvis vi skal implementere dette.

8 Disjunkte mængder

8.1 Vektor repræsentation

Givet nogle disjunkte mængder, så kan de repræsenteres ved at kende 6 "ting"for dem:

- SET repræsentativ (kanonisk element - samme som FIRST)
- SET: Hvilket set hvert element er i
- NEXT: Hvilket element der kommer efter elementet
- FIRST: Hvad der er først (kanonisk element)
- LAST: Hvad der er det sidste element
- SIZE: Størrelsen på det givne set.

8.1.1 Make set

Nem og lave i konstant tid $O(1)$, makeset(x) skal blot lave operationerne: set(x)=x, first(x)=x, next(x)=0, size(x)=1

8.1.2 Find

find(x) skal blot returnerer set(x), $O(1)$

8.1.3 Link

Sammensætter to sets, givet to kanoniske elementer x og y, så bliver den mindre mængde tilføjet til den større, og alle elementer i den lille mængde pointer til det kanoniske element bliver opdateret.

8.2 Træ repræsentation

Hvert set er et nyt træ. Roden er det kanoniske element. Hver knude har en pointer til sin forældre, roden peget til sig selv.

8.2.1 Makeset

Lav en et-knude træ i $O(1)$

8.2.2 Find

Følg foreældre pointere fra x indtil roden er fundet, som er fundet når pointeren går til sig selv. Dvs. det er det kanoniske element der returneres.

8.2.3 Link

Ved link sættes de to træer blot sammen ved at sætte den pointer der går til sig til over til det andet træs rod (fint, da det ikke er et binært træ). Oftest laves link ved størrelse, så man sætter det "lille"træ på det større træ.

8.2.4 Problemer med træ repræsentationen

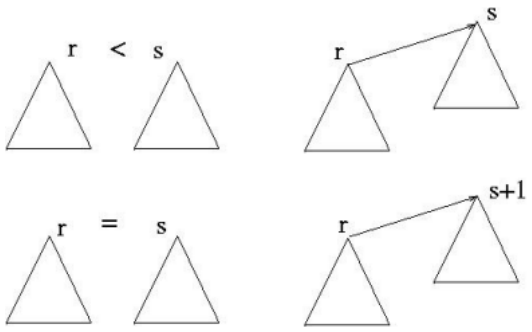
Man kan risikere, at man laver en dårlig kombination af makeset og link, således at man får et træ med kun en lang vej. Dette vil gøre, at vores find kører i $O(n)$. Derfor skal man fx linke med størrelse som prioritet eller rank.

8.2.5 Link ift. størrelse

Her linkes der blot således at det træ med færrest elementer sættes på det træ med flere elementer.

8.2.6 Link ift. rank

Alle rodknuder med et element har ranken 1. Hvis 2 træer skal sættes sammen, så sættes det træ med mindst rank på den med større. Hvis de har lige rank, vælg blot en og øg dens rank. Se figur 15.



Figur 15: Link ift. rank

8.2.7 Vej kompressioner

Når find bliver kaldt på en knude, så sætter den automatisk alle knuder den går igennem til at være direkte børn af roden.

	Makeset	Find	Link	Total
Vector	O(1)	O(1)	O(n)	O(m+nlogn)
Tree	O(1)	O(n)	O(1)	O(mn)
Link by rank	O(1)	O(logn)	O(1)	O(n+mlogn)
Compression	O(1)	O(logn)	O(1)	O(n+mlog*n)

Figur 16: Tidskompleksitet af forskellige repræsentationer af disjunkte set. $log*$ er "bedre"end log

```
MAKE-SET(x)
1  x.p = x
2  x.rank = 0

UNION(x, y)
1  LINK(FIND-SET(x), FIND-SET(y))

LINK(x, y)
1  if x.rank > y.rank
2       y.p = x
3  else x.p = y
4       if x.rank == y.rank
5            y.rank = y.rank + 1
```

The FIND-SET procedure with path compression is quite simple:

```
FIND-SET(x)
1  if x ≠ x.p
2       x.p = FIND-SET(x.p)
3  return x.p
```

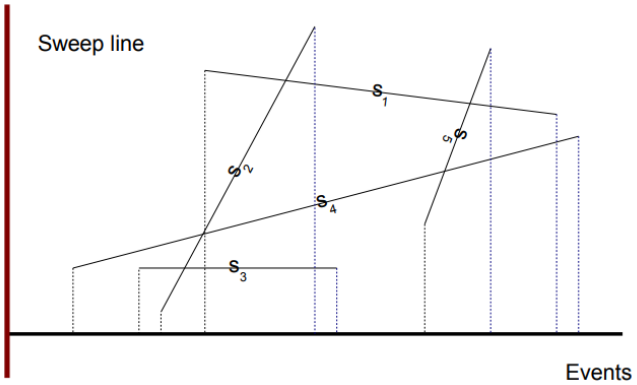
Figur 17: Kode til disjoint sets. Bemærk Find-Set laver stikomprimering. Samt at Union også laver stikomprimeringer, hvis det sæt der angives ikke er roden (eller et direkte barn af roden), så bliver den sat direkte på roden sammen med andre sæt på vejen til roden.

9 Plane sweep: skæringer mellem linjer

Problemet er, at givet en mængde af linjer, bestem hvor mange gange de skærer hinanden. Den trivielle måde med blot at se på alle linjer tager $O(n^2)$, men det kan gøres i $O((n+k)log(n))$ ved brug af plane sweep (k er antal skæringer).
Antagelser (algoritmen kan ændres, uden at få en værre tidskompleksitet, til at håndtere dem):

- Ingen vertikale linjer
- Ingen linjer overlapper
- 3 linjer kan ikke overlappe i samme punkt

Først angives start- og slutsteder for linjerne på x-aksen, dvs. vi kender de to "events"for hver linje, men vi mangler at kende deres eventuelle skæringer. Herefter danner vi en "sweeper", som kan anses som at være en lodret linje, som skal "feje"hen over vores linjer. Se figur 18 for visuelt at se situationen.



Figur 18: Plane sweep situation fra start.

Ved en hver x-koordinat, kan fejelinjen angive rækkefølgen af linjerne oppefra og ned, kaldet fejelinjens status.
Når fejelinjen kommer til et startsted for en linje s:

- s indsættes på fejelinjens status
- Kontroller om s skærer linjen over sig på fejelinjen. Hvis den gør, tilføj skæringen som event.
- Kontroller om s skærer linjen under sig på fejelinjen. Hvis den gør, tilføj skæringen som event.

Når fejelinjen kommer til et slutsted for en linje s:

- Kontroller om linjer over eller under s skærer til højre for s. Hvis de gør, og ikke tidligere fundet, tilføj som event.
- Fjern s fra fejelinjens status.

Når fejelinjen ankommer til et skæringspunkt p mellem s_1 og s_2, som er blevet tilføjet ved de to "cases"nævn ovenfor

- Byt s_1 og s_2 på fejelinjens status
- Kontroller om s_2 og linjen over den skærer til højre for p. Hvis den gør, og ikke tidligere fundet, tilføj som event.
- Kontroller om s_2 og linjen under den skærer til højre for p. Hvis den gør, og ikke tidligere fundet, tilføj som event.

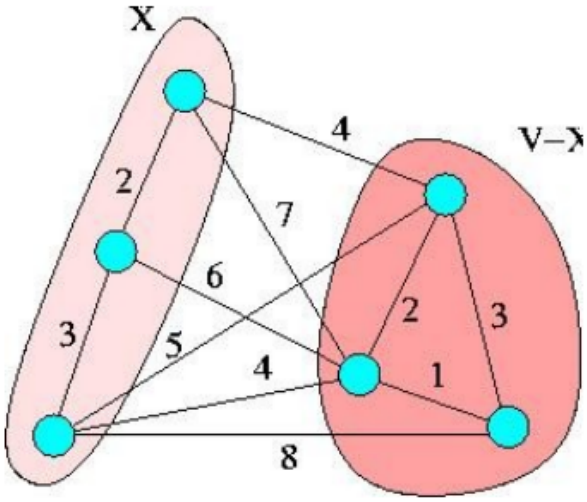
Algoritmen events kræver at kunne få den mindste element, indsætte et nyt på korrekt position og kontrollere for medlemsskab. Dette kan gøres ved brug af et binært søgetræ, så alle operationerne kræver $O(log(n))$. Status linjen kræver at kunne indsætte, fjerne, bytte to elementer, returnere et element lige før et andet og et lige efter. Kan også gøres med binært søgetræ med samme tid: $O(log(n))$

10 Mindste udspændende træ (minumum spanning tree - MST)

Et træ, er en forbundet graf hvori er ikke er nogle cycles (kredse). Et Udspændende træ har alle knuder i grafen. En graf med n knuder har i sit udspændende træ $n-1$ kanter ("streger"). Et mindste udspændende træ, er et træ hvori summen af vægte (fra kanterne) er minimeret.

10.1 Snit

$C = \{X, V - X\}, \varnothing \subset X \subset V$ er en opdeling af knuder i to delmængder, som kaldes et snit af G, se figur 19. En kant krydser et snit hvis den har en knude i X og en anden i $V - X$. De kanter der krydser et snit som har den mindste vægt kaldes en let kant (light edge), fra figur 19 vil 4 være en let kant.



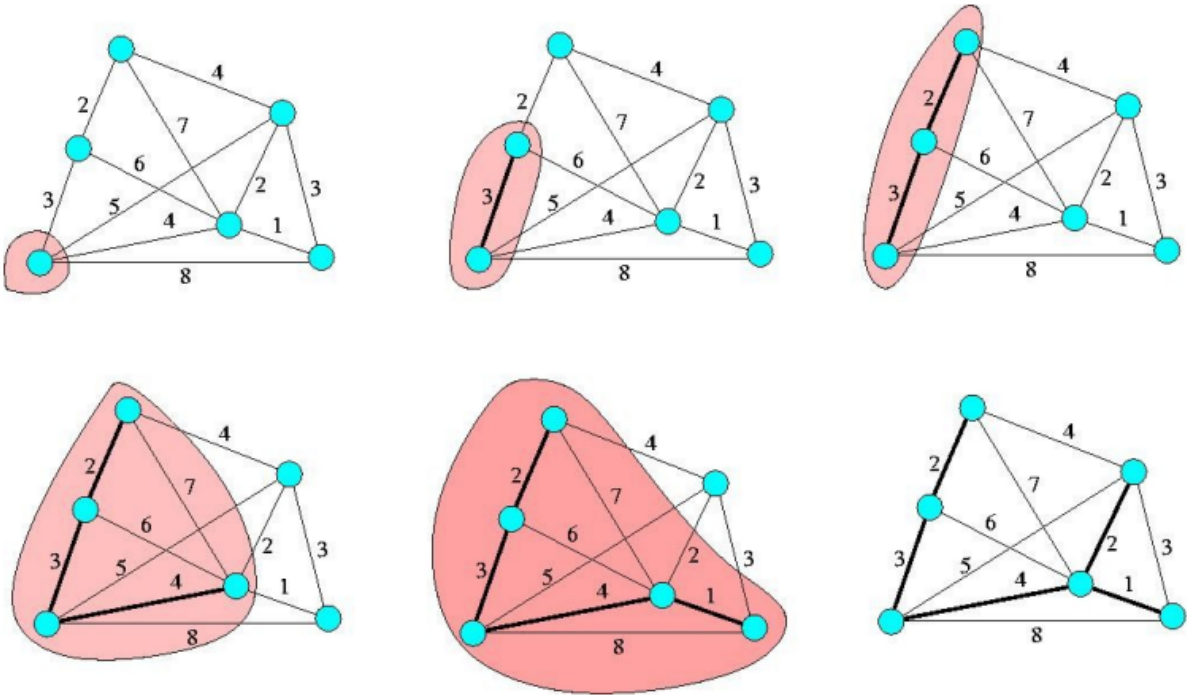
Figur 19: Eksempel på et snit mellem X og $V - X$

10.2 Let kant (light edge)

En let kant fra et hvilket som helst snit tilhører et mindste udspændende træ (ikke nødvendigvis unikt). Denne egenskab kan nemt vises ved modstrid. Se blot på en let kant e, som ikke er indeholdt i noget MST. Se da på snittet som e er i, og se da at det gamle MST totale sum af vægtene må blive mindre eller forblive i størrelse når e bliver valgt i stedet for den gamle kant. Se slides for mere formelt bevis.

10.3 Prims algoritme

Vælg en arbitrær knude. Lav herefter et snit som respekterer den knude. Vælg så den lette kant fra det snit. Dan herfra endnu et snit som respekterer de to knuder og fortsæt sådan. Se figur 20 for eksempel.

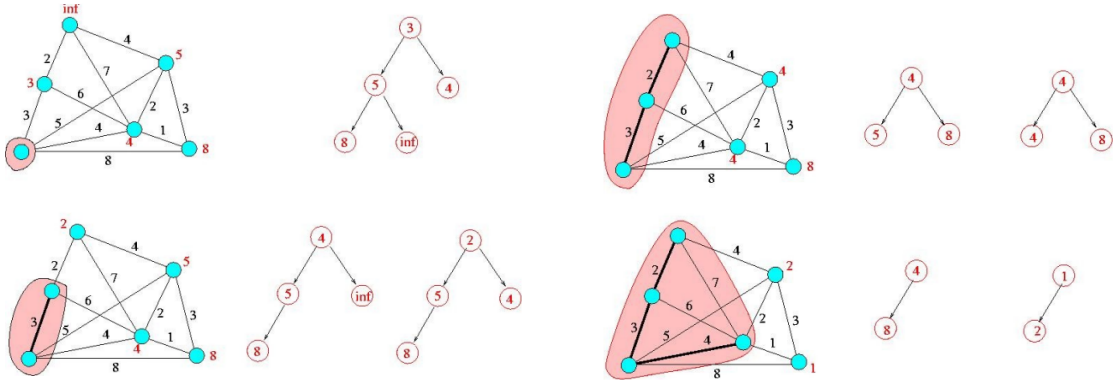


Figur 20: Eksempel af Prims algoritme

Implementation af prims:

- Kanter ikke i træet T er i en min-heap
- Enhver af sådan en kant har en nøgle med længden af den korteste kant der forbinder den med T.
- Den knude der skal tilføjes til T er roden i min-heapen.
- Hver gang en kant tilføjes skal min-heapen opdateres, da nøglerne fra kanterne kan blive mindre (da afstanden til T kan blive mindre).

Med en min-heap som kø, så får vi en samlet tidskompleksitet på $O(mlog(n))$ hvor m er antal kanter i den sammenhængene graf.



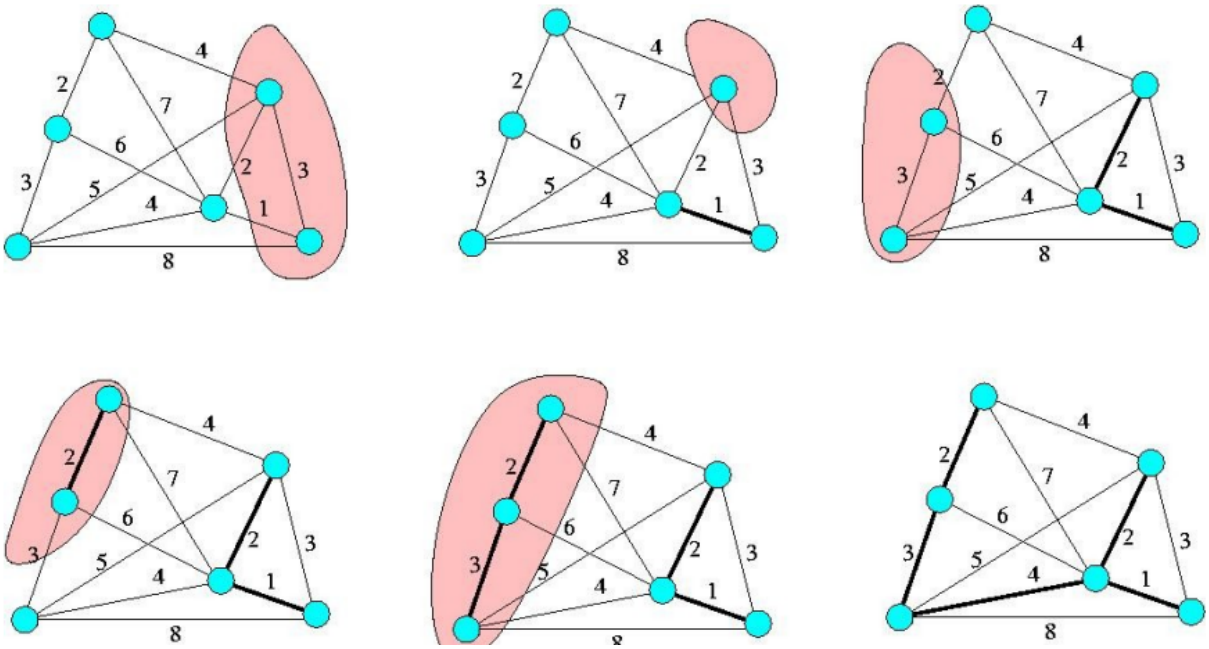
(a) Heapen ved de 2 første iterationer

(b) Heapen ved de næste 2 iterationer

Figur 21: Eksempel på prims algoritme med min-heap som prioritetskø

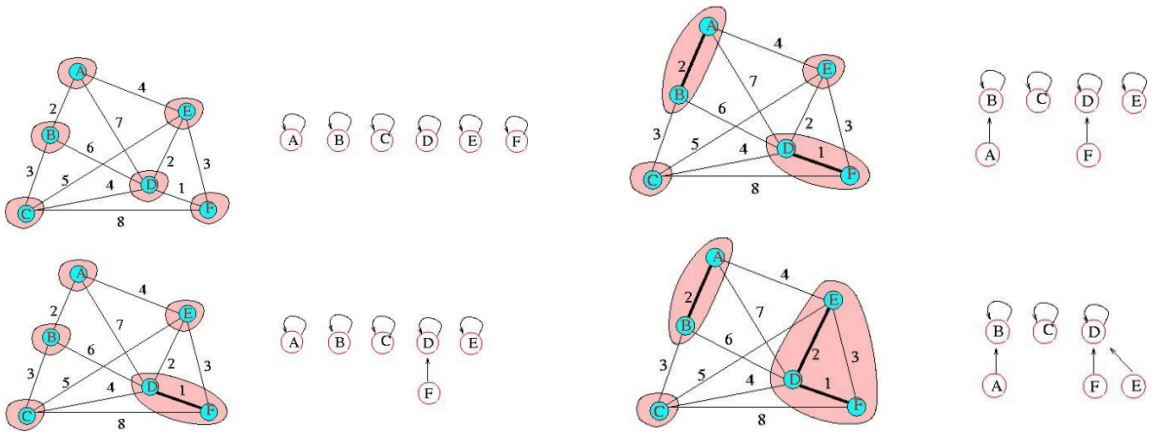
10.4 Kruskals algoritme

Vælg altid bare den mindste kant som ikke danner en kreds, se figur 22.



Figur 22: Eksempel på kruskals algoritme

Implementation af kruskal kan gøres ved først at sortere kanterne efter størrelse. Herefter skal vi blot holde styr på, at der ikke dannes en kreds. Dette kan gøres ved brug af disjunkte mængder. Hver gang en knude tilføjes, så bliver de to disjunkte mængder sat sammen ved brug af link. Dvs. hver gang vi skal lave en ny kant, så kigger vi blot på om de to knuder er i hver sin disjunkte mængde. Se figur 23 for visuelt eksempel.



(a) De disjunkte mængder ved første tilføjelse

(b) De disjunkte mængder ved de næste 2 tilføjelser

Figur 23: Eksempel på kruskals algoritme med disjunkte mængder for at holde styr på kredse.

Tidskompleksitet af kruskal er $O(mlog(m))$ (m er antal kanter).

11 Korteste vej

Givet en graf, så kan der bestemmes den kortete vej mellem en knude og alle, korteste vej mellem alle knuder eller lignende. Problemet er veldefineret hvis der ikke findes kredse med en negativ sum.

11.1 Relaxation

Hvis der findes en ny kortere vej til en knude, så opdater dens afstand med hjælpefunktionen relaxation(u,v). Dvs hvis

$$d[v] > d[u] + w(u, v)$$

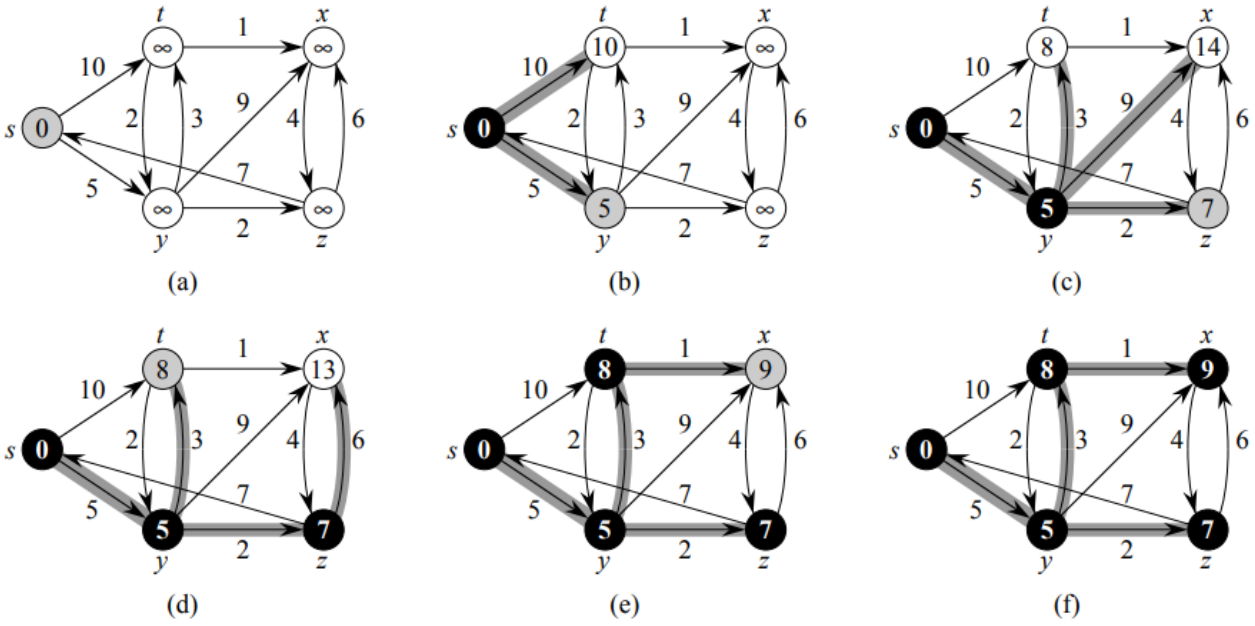
så sæt $d[v] = d[u] + w(u, v)$ og pointeren sættes til u.

11.2 Belman-Ford algoritme

Gå igennem alle kanter (i en tilfældig rækkefølge), og relaxer hvis der findes en mindre afstand, dette gøres $n - 1$ gange (dvs. den går igennem alle n kanter n-1 gange). Herefter kontrolleres det for om der findes en vej hvori $d[v] > d[u] + w(u, v)$ gælder, og hvis ja, så må der findes en negativ kreds og derfor udefineret. $O(n \cdot m)$.

11.2.1 Dijkstras algoritme

Kun defineret for ikke negative vægte. Ved hver iteration, vælg den knude med den korteste vej u til start. Relaxer herefter alle knuder fra u. Fortsæt indtil alle knuder er "kigget"fra. Se figur 24.



Figur 24: Eksempel på Dijkstras algoritme

For at holde styr på den knude med lavest afstand, kan en min-heap benyttes.

12 Beregningsgeometri (computational geometry)

12.1 Konvekse hylstre

Givet et sæt Z af n punkter, find det mindste konvekse sæt der indeholder Z. Et sæt er konveks, hvis der for ethvert 2 punkter i sættet kan laves en lige linje mellem, hvori linjen også er i sættet.

Højre og venstre drej for 3 punkter kan beregnes ved at opstille en matrice med x-koordinater i første kolonne og y i anden og 1 i tredje. Hvis determinanten er positiv er det et venstre drej.

12.1.1 Punkt Beskærings metode

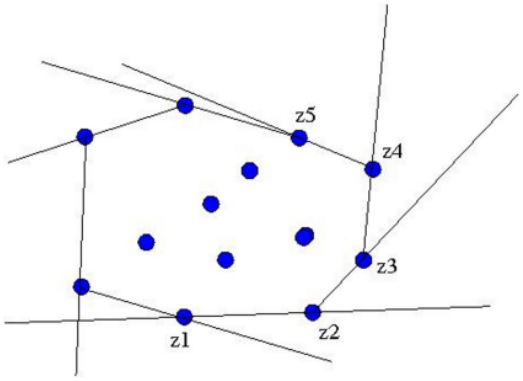
Lav en trekant mellem 3 punkter, og smid da de indre punkter væk, da de umuligt kan være hjørner i det konvekse sæt. For at bestemme om punkterne er i trekanten, så skal alle to punkter i trekanten kunne lave samme drej for at komme til den. Fortsæt med at lave trekanter indtil alle er hjørner (dvs. ingen trekanter har punkter i).

12.1.2 Kant beskærings metode

Lav linjer mellem to punkter, hvis der er punkter både til højre og venstre for linjen, så kan det ikke være en ydre kant. Hvis det er en ydre kant, så må de to punkter være hjørnepunkter.

12.1.3 Jarvis March

En forbedring af kant beskærings metoden.
Når en hjørne kant er blevet fundet, så drej linjen om det om det ene punkt indtil den rammer et nyt punkt, dette må også være et hjørnepunkt. Og fortsæt da fra det punkt af, se figur 25.

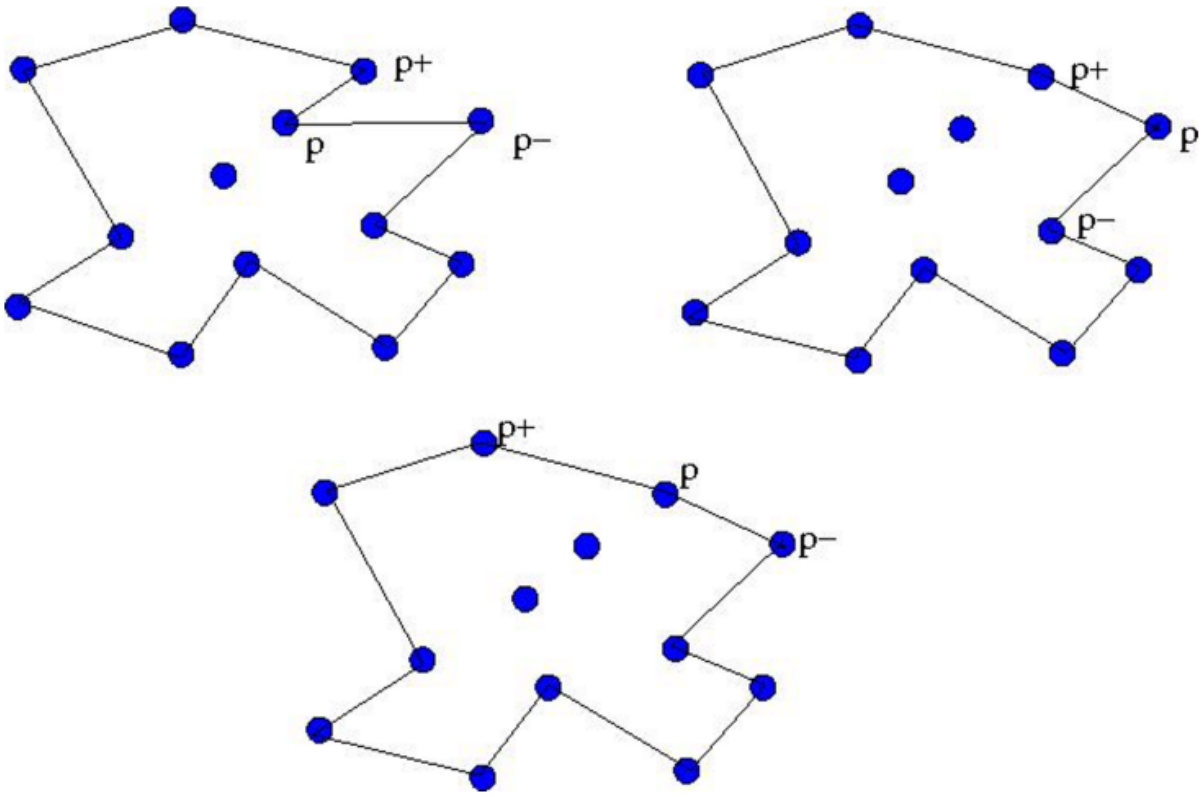


Figur 25: Eksempel på Jarvis March

Kører i $O(hn)$, hvor h er antal af hjørnepunkter. Implementeres på en lidt anden måde, se da slides.

12.1.4 Grahams scan

Bestem først centroiden af alle punkter (gennemsnitlige punkt: x-koordinatet er den gennemsnitlige x for alle punkter, samme med y). Sorter herefter punkterne efter polær koordinat, med vinkel først og så afstand hvis 2 har samme vinkel. Herefter angiver man et punkt p, den forgænger p- og den eftergænger p+. Hvis p-, p og p+ laver et højre drej, så må p være inde i mængden og altså ikke et hjørne og smides da væk. Se eksempel på figur 26.



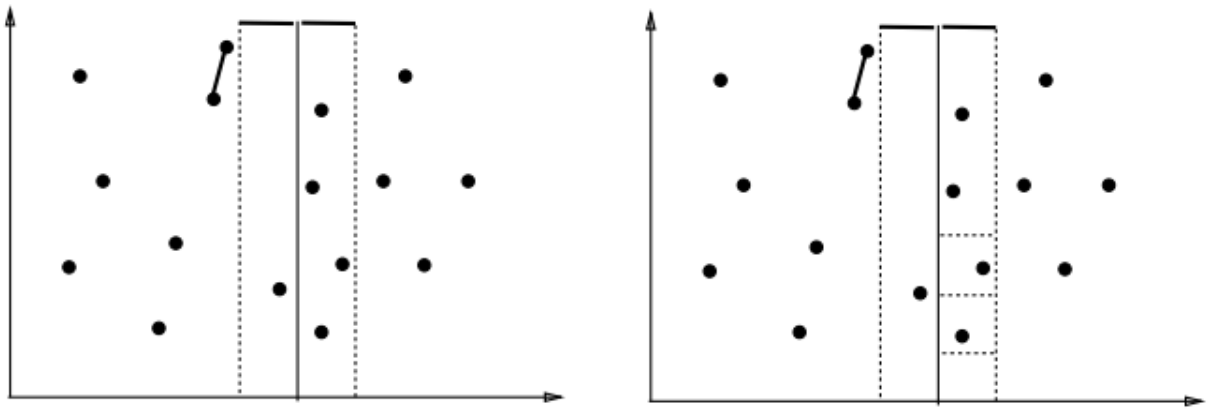
Figur 26: Grahams scan. Figuren oppe til venstre smides p væk, da de laver et højre drej. Oppe til højresmides p ikke væk, da de laver et venstre drej. Og figuren nede bliver p også.

12.2 Tætteste par problem

Givet n punkter, find da det tætteste par. Det naive ville være bare at tage et punkt, og så bestemme alle afstande for hvert punkt med $O(n^2)$. Dette kan nemt forbedres i første dimension, for der er alle punkterne bare på en linje, som kan sorteres, og derfor behøves der kun at kontrollere n punkter, og derfor $O(nlog(n))$ (pga. sortering).

12.2.1 Del og hersk metode

Del op i to delmængder rekursivt, og beregn da afstanden når der kun er 2 punkter tilbage. Dog skal der også kontrolleres for tætteste punkter mellem to delmængder. Denne kører i $O(n^2)$. For at løse problemet med at sammenligne fra to delmængder, så bestemmes først den mindste afstand i de to mængder. Tag den afstand i x-retningen på begge sider af dælmængderne, og sammenlign da blot de punkter der ligger der med de punkter der i y-retningen max er den mindste afstand væk, se figur 27. Og da der maks kan være 4 punkter uden at have en mindre afstand end den mindste i sådan en firkant der dannes, så skal hvert punkt kun sammenlignes med 4 andre punkter.



Figur 27: Sammenlingen af de tætteste punkter af to delmængder med del og hersk.

Denne metoder kører i $O(nlog(n))$