



Noter

Adit (hjg708)

AD 2023

Indhold

1 Divide and Conquer	4
1.1 1. Hvad er divide-and-conquer (del-og-hersk)?	4
1.2 2. Merge Sort — kørsel med [5, 3, 13, 10, 14]	4
1.3 3. Asymptotisk analyse	5
1.3.1 i. Rekursionsrelationer og -træer	5
1.3.2 ii. Substitutionsmetoden	5
1.3.3 iii. Master theorem (kort)	6
1.4 4. Lower bound for sorterings (kort)	7
2 Dynamic Programming	8
2.1 1. Hvad er Dynamic Programming?	8
2.2 2. Fibonacci — Divide and conquer vs. Dynamic Programming	8
2.2.1 i. Gentagelse af sub-problemer	8
2.2.2 ii. Effekt på køretid	9
2.3 3. Longest common subsequence (LCS)	9
2.3.1 i. Beskrivelse af problemet	9
2.3.2 ii. Håndkørse af algoritme — LCS(ABC, ACD)	9
2.3.3 iii. Bevis af optimal delstruktur (Theorem 15.1)	10
3 Greedy Algorithms	10
3.1 1. Hvad er greedy algorithms?	10
3.2 Huffman	11
3.2.1 i. Mål og notation	11
3.2.2 ii. Håndkørsel med a:12, b:13, c:10, d:5	11
3.2.3 iii. Greedy choice property	11
3.2.4 iv. Kort om Lemma om $B(T)$	12
3.2.5 v. Optimal delstruktur	12
3.2.6 vi. Kort om køretid	12
4 Amortized Analysis	13
4.1 1. Hvad er Amortized Analysis, og hvorfor bruge det?	13
4.2 2. Stak med MultiPop	13
4.3 3. Aggregeret analyse	13
4.4 4. Accounting method	13
4.5 5. Potentiale metode	14
4.5.1 i. Om metoden	14
4.5.2 ii. Eksempel med stack	14
5 Fibonacci Heaps	15
5.1 1. Hvad er det?	15
5.2 2. Struktur	15
5.2.1 i. Egenskaber	15
5.2.2 ii. Attributter	15
5.2.3 iii. Operationer	15
5.3 3. Operationen Extract-Min	16
5.3.1 i. Håndkørsel med ovenstående input	16
5.3.2 ii. Potentialefunktionen Φ	17
5.3.3 iii. Amortiseret køretid	17
6 Balanced Binary Search Trees	18
6.1 1. Hvad er Binary Search Trees?	18
6.1.1 i. Insert, Delete, Search	18
6.1.2 ii. Struktur og BST egenskab — eksempel $\langle 11, 2, 1, 7, 5, 8, 14, 15 \rangle$	18
6.2 2. Red-black trees	18
6.2.1 i. Hvorfor?	18
6.2.2 ii. Struktur og 5 egenskaber	18
6.2.3 iii. Bevis af lemma	19

6.2.4	iv. Bevis af $h = O(\dots)$	19
6.2.5	v. Kort om opretholdelse af egenskaber — Rotationer	20
7	Minimum Spanning Trees	20
7.1	1. Beskrivelse af problemet	20
7.2	2. Generisk algoritme	21
7.2.1	i. Termonologi	21
7.2.2	ii. Algoritmen	21
7.2.3	iii. Bevis af korrekthed	21
7.3	3. Kruskals algortime	22
7.3.1	i. Implementation med disjoint sets	22
7.3.2	ii. Håndkørsel med ovenstående input	23
7.3.3	iii. Køretid (kort)	23
7.4	4. Prims algoritme	24
7.4.1	i. Håndkørsel med samme graf som før — start nederst til højre	24
7.4.2	ii. Køretiden er $O(\dots)$	24
8	Shortest Paths	24
8.1	1. Beskrivelse af problemet	24
8.2	2. Generelt for algoritmerne — initialisering og relaxation	24
8.3	3. Bellman-Ford algortime	25
8.3.1	i. Håndkørsel med ovenstående input	25
8.3.2	ii. Køretid (kort)	25
8.4	4. Dijkstra's algortime	26
8.4.1	i. Håndkørsel med ovenstående input	26
8.4.2	ii. Køretiden er $O(\dots)$	26
8.4.3	iii. Bevis af korrekthed	26

1 Divide and Conquer

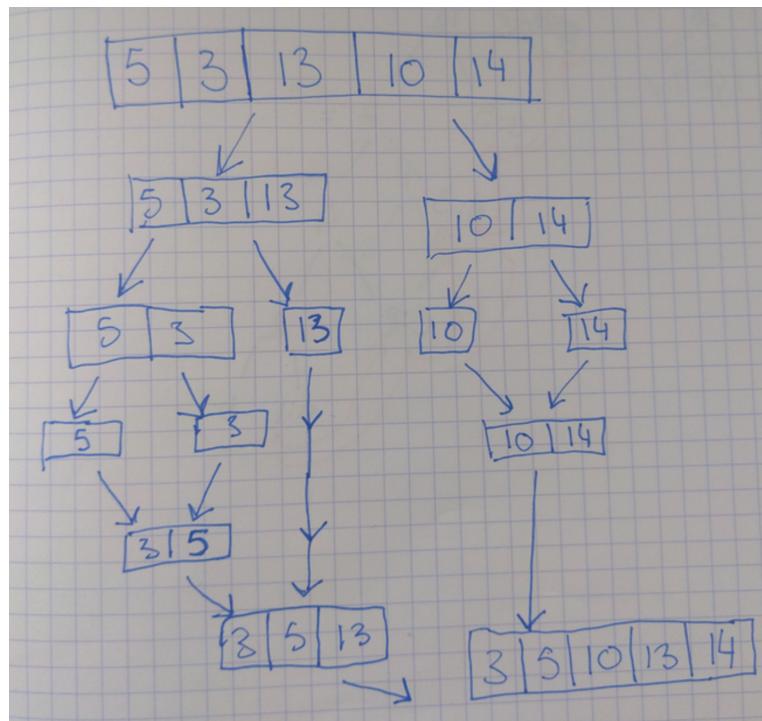
1.1 1. Hvad er divide-and-conquer (del-og-hersk)?

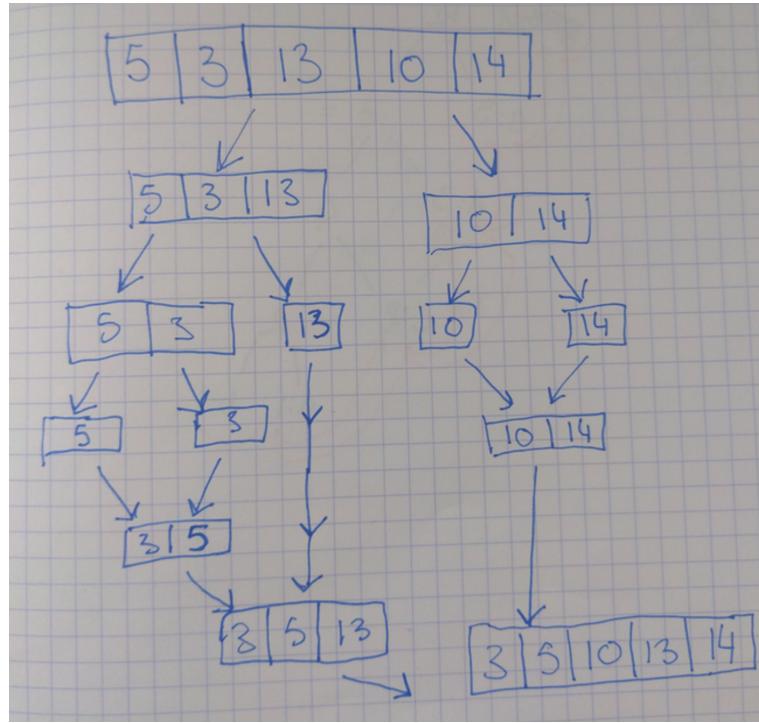
- Del problemet ind i en mængde af delproblemer som er mindre instanser af det samme problem.
- Hersk (løs) delproblemerne ved at løse dem rekursivt. Hvis delproblemerne er tilpas små, så løs dem bare "direkte".
- Kombinér løsningerne til delproblemerne sammen til én løsning til det oprindelige problem.
- Man kan sammenligne med Dynamisk Programmering, men problemerne har ikke altid optimal delstruktur, altså der er ikke overlappende delproblemer.
- Eksempler på algoritmer der benytter det er QuickSort og MergeSort

1.2 2. Merge Sort — kørsel med [5, 3, 13, 10, 14]

- Lav kun den ene side

- Lad være med at tegne bokse rundt om tallene



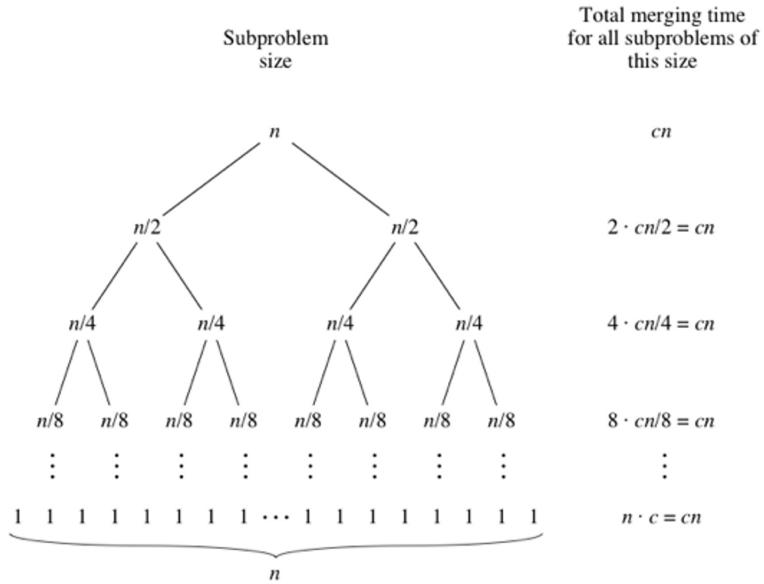


1.3 3. Asymptotisk analyse

1.3.1 i. Rekursionsrelationer og -træer

Rekursionstræer: Kan i sig selv bruges som bevis hvis man er meget forsigtig. Men jeg vil bruge det til at komme med et kvalificeret gæt.

Tegn dette træ:



Rekursionsrelationer — eksempel for MergeSort:

$$T(n) = 2T(\lfloor n/2 \rfloor) + cn$$

1.3.2 ii. Substitutionsmetoden

- Metode:

1. Gæt på en løsning
 2. Bevis din løsning er sand ved hjælp af induktion (OBS: se ikke bort fra konstanter)
 3. Hvis gæt ikke virker, så kom med nyt gæt
- Induktionsbevis for MergeSort
 - Vi gætter at $T(n) = O(n \lg n)$
 - Vis: $T(n) \leq cn \lg n$ for $n \geq n_0$
 - Vi vælger $n_0 = 2$ (bemærk: vi kan ikke vælge $n_0 = 1$, da $\lg 1 = 0$)
 - Vi antager at $T(1) = 1$
 - Vi antager at n er en toerpotens, for at gøre beviset lettere (ellers ville vi have to base case, $n = 2, 3$)
 - Base Case, $n = 2$:
 - * $T(2) = 2T(2/2) + 2 = 2T(1) + 2 = 4 \leq c \cdot 2 \lg 2$
 - * Vi ser at $T(n) \leq cn \lg n$ for $c \geq 2$
 - Induktionsskridt:
 - * Induktionsantagelsen: $T(m) \leq cm \lg m$ for $m = 2, 3, \dots, n - 1$. Derfor:
$$\begin{aligned} T(n) &= 2T(n/2) + n \\ &\leq 2c \frac{n}{2} \lg\left(\frac{n}{2}\right) + n \\ &= cn \lg\left(\frac{n}{2}\right) + n \end{aligned}$$
 - * Brug logaritmeregler og vi får:
$$\begin{aligned} &= cn \lg(n) - cn \lg(2) + n \\ &= cn \lg(n) - cn + n \\ &\leq cn \lg n \end{aligned}$$
 - * QED: Vi har $T(n) \leq cn \lg n$

1.3.3 iii. Master theorem (kort)

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

Ud fra konstanerne a, b samt hvordan funktionen $f(n)$ er asymptotisk, kan man finde den asymptotiske $T(n)$

Note. Det ses her (ikke så vigtigt præcis hvad det er)

Theorem 4.1 (Master theorem)

Let $a \geq 1$ and $b > 1$ be constants, let $f(n)$ be a function, and let $T(n)$ be defined on the nonnegative integers by the recurrence

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

where we interpret n/b to mean either $\lfloor n/b \rfloor$ or $\lceil n/b \rceil$. Then $T(n)$ has the following asymptotic bounds:

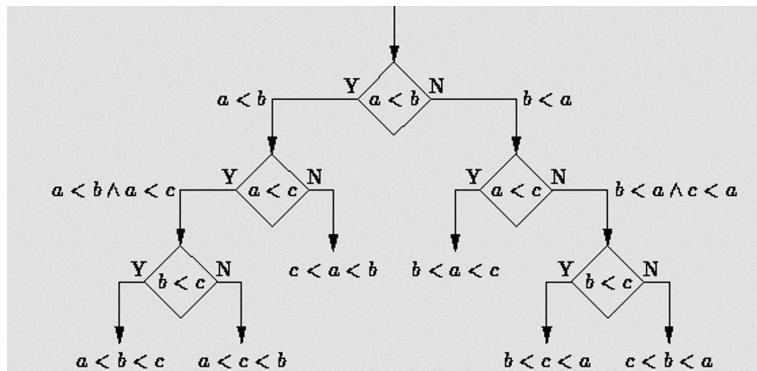
1. If $f(n) = O(n^{\log_b a - \epsilon})$ for some constant $\epsilon > 0$, then $T(n) = \Theta(n^{\log_b a})$.
2. If $f(n) = \Theta(n^{\log_b a})$, then $T(n) = \Theta(n^{\log_b a} \lg n)$.
3. If $f(n) = \Omega(n^{\log_b a + \epsilon})$ for some constant $\epsilon > 0$, and if $af(n/b) \leq cf(n)$ for some constant $c < 1$ and all sufficiently large n , then $T(n) = \Theta(f(n))$. ■

1.4 4. Lower bound for sorteringsalgoritme (kort)

- Lad S være en sorteringsalgoritme, der kun kan tjekke om $a < b$. Den må altså ikke bruge anden information (counting sort bruger f.eks. information om største/mindste værdier).
- For at sortere n elementer vil den bedste køretid være:

$$S = \Omega(n \lg n)$$

- Vi kan opstille dette som et decision tree:

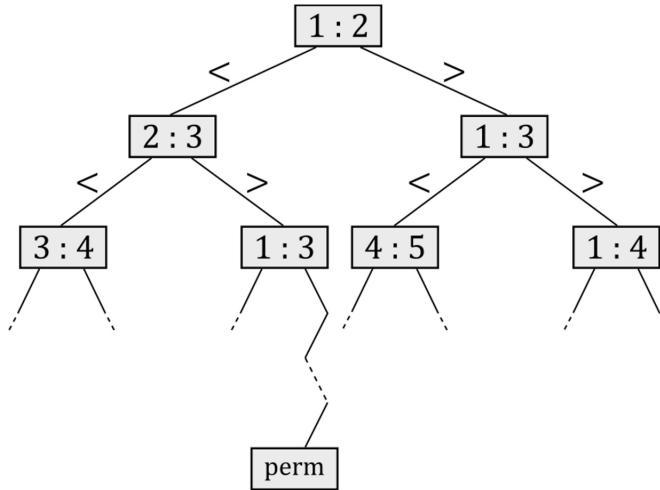


- Vi kan vise, at den højden på dette træ: $h \geq \lg n! = n \lg n$
- Længere beskrivelse:

Vi antager for at gøre det nemmere, at der ikke er elementer som er ens. Derudover vil vi kun bruge operationen $a < b$ til at få informationer om to elementers relation, da alle andre giver ækvivalent information.

Antag vi får et input array A med n elementer. Nu anvender vi en comparison sort S på denne. Det S gør, er at permuttere elementer i A til en sorteret liste.

Denne permutation er unikt defineret ud fra de sammenligninger algoritmen har foretaget. Alle disse udfald kan vi repræsentere i et beslutningstræ. Alle simple veje fra rodknuden til et blad er alle de forskellige permutationer der findes.



Lad nu antallet af blade være b og højden h . Såfremt træet er perfekt balanceret kan vi presse 2^h blade ind i træet. Og vi ved også, at alle permutationer forekommer i bladene, derfor må $b \geq n!$ for ellers ville der ikke være plads til alle permutationer i bladene. Derfor får vi:

$$2^h \geq b \geq n!$$

Vi kan tage 2tals-logaritmen hvorved vi får:

$$h \geq \lg(n!) = \Omega(n \lg n)$$

Herved har vi altså bevist, at der er et input af længde n som kræver at der bliver foretaget $h = \Omega(n \lg n)$ sammenligner.

2 Dynamic Programming

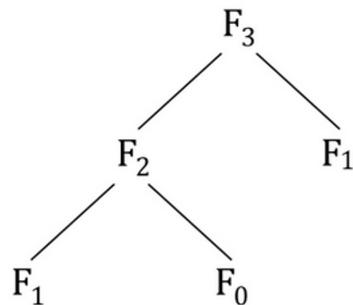
2.1 1. Hvad er Dynamic Programming?

- Minder om og nærmest udvidelse af del og hersk-paradigmet: Del problemet ind i mindre delproblemer, løs delproblemerne rekursivt og kombiner herefter løsningerne.
- Problemerne skal have optimal delstruktur. Det vil sige delproblemerne på en eller anden måde overlapper. Altså når flere delproblemer har de samme deldelproblemer.

2.2 2. Fibonacci — Divide and conquer vs. Dynamic Programming

2.2.1 i. Gentagelse af sub-problemer

- Tegn træet herunder



- Det kan her ses at vi løser F_1 flere gange. Derfor er det hurtigere hvis vi husker resultatet (undgå føre rekursive kald)

2.2.2 ii. Effekt på køretid

- Vi antager at vi kan addere i konstant tid.
- Så får vores algoritme linær køretid i stedet for exponentiel

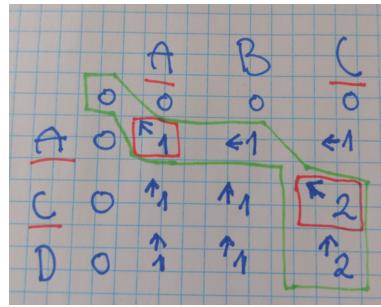
2.3 3. Longest common subsequence (LCS)

2.3.1 i. Beskrivelse af problemet

- Vi vil gerne finde Longest Common Subsequence af f.eks. to DNA-strenge.
- Eksempelvis: LCS(ABC, ACD) \rightarrow AC

2.3.2 ii. Håndkørse af algoritme — LCS(ABC, ACD)

- Tegn tabel og fyld ud
- Bemærk at hver enkelt celle tager konstant tid, da den kun tjekker de omkringliggende celler



- Se algoritmen

LCS-LENGTH(X, Y)

```

1   $m = X.length$ 
2   $n = Y.length$ 
3  let  $b[1..m, 1..n]$  and  $c[0..m, 0..n]$  be new tables
4  for  $i = 1$  to  $m$ 
5     $c[i, 0] = 0$ 
6  for  $j = 0$  to  $n$ 
7     $c[0, j] = 0$ 
8  for  $i = 1$  to  $m$ 
9    for  $j = 1$  to  $n$ 
10   if  $x_i == y_j$ 
11      $c[i, j] = c[i - 1, j - 1] + 1$ 
12      $b[i, j] = "\nwarrow"$ 
13   elseif  $c[i - 1, j] \geq c[i, j - 1]$ 
14      $c[i, j] = c[i - 1, j]$ 
15      $b[i, j] = "\uparrow"$ 
16   else  $c[i, j] = c[i, j - 1]$ 
17      $b[i, j] = "\leftarrow"$ 
18 return  $c$  and  $b$ 

```

2.3.3 iii. Bevis af optimal delstruktur (Theorem 15.1)

- Fortæl Theorem.

Theorem 15.1 (*Optimal delstruktur af en LCS*)

Lad $X = \langle x_1, x_2, \dots, x_m \rangle$ og $Y = \langle y_1, y_2, \dots, y_n \rangle$ være sekvenserne (f.eks. DNA-strenge), og lad $Z = \langle z_1, z_2, \dots, z_k \rangle$ være en hvilken som helst LCS af X og Y :

- 1: Hvis $x_m = y_n$, så er $z_k = x_m = y_n$ og Z_{k-1} er en LCS af X_{m-1} og Y_{n-1} .
- 2: Hvis $x_m \neq y_n$ og $z_k \neq x_m$, så medfører det at Z er en LCS af X_{m-1} og Y .
- 3: Hvis $x_m \neq y_n$ og $z_k \neq y_n$, så medfører det at Z er en LCS af X og Y_{n-1} .

- Alle er beviser ved modstrid

1. De ender på det samme

$$\begin{aligned} X &= * * * * A \\ Y &= * * * * A \\ Z &= * * * A \end{aligned}$$

(a) Bevis for $z_k = x_m = y_n$:

- - Antag at $x_m \neq z_k$ (her: $Z = * * * B$)
- Så kunne vi sætte $x_m = y_n$ (her: A) bagpå Z (her: $W = * * * B A$)
- Nu har vi en W med længde $k + 1$, men dette er modstid, da Z med længde k var en LCS

(b) Bevis for Z_{k-1} er en LCS af X_{m-1} og Y_{n-1} :

- Antag, at Z_{k-1} (her: $* * *$) er en CS af X_{m-1} og Y_{n-1} , men ikke den længste
- Der må dermed findes en sekvens $W = * * * *$ som er længere end Z_{k-1} .
- Vi kan sætte $x_m = y_n$ (her: A) bagpå W^{**} (her: $W = * * * * A$) og få en sekvens længere end Z
- Dette er modstrid, da Z var en LCS

2. De ender ikke på det samme

$$\begin{aligned} X &= * * * * * A \\ Y &= * * * * A \quad < -- O \quad ! = A, \text{ potentelt } O = B \\ Z &= * * * A \end{aligned}$$

Bevis ved modstrid

- Antag at $Z \notin \text{LCS}(X_{m-1}, Y)$
- Eftersom $x_m \neq z_k$ (her: $A \neq B$), vil Z stadig være en CS; vi kan altså fjerne A
- Der må dermed findes en sekvens $W \in \text{LCS}(X_{m-1}, Y)$
- Vi har $W > Z$ (her: $* * * * B$)
- Vi kan sætte x_m bagpå uden det gør noget (siden $x_m \neq z_k$, her: $A \neq B$) Dermed: $W \in \text{LCS}(X, Y)$
- Dette er modstrid, da $Z \in \text{LCS}(X, Y)$

3. Symmetrisk med nummer 2

3 Greedy Algorithms

3.1 1. Hvad er greedy algorithms?

- Vi har en række delproblemer der afhænger af løsningen på andre delproblemer.

- Grådig algoritme laver vi altid den løsning som på nuværende tidspunkt ser bedst ud, uden at kigge på fremtidige delproblemer.
- Greedy choice property: Man kan kun bruge greedy algoritms hvis en lokal optimal løsning vil lede til en global optimal løsning.
- Optimal delstruktur: hvis der fandtes en optimal løsning der indeholder det grådige valg, så består den optimale løsning P af en sammensætning af, det grådige valg (x) og, en løsning til det delproblem vi har tilbage når vi har lavet det grådige valg (P').
- Kan sammenlignes med optimal delstruktur fra Dynamic Programming, men ikke formuleret ens
- Knapsack problem er et eksempel hvor man ikke kan bruge Greedy Algorithms
- Activity selection kan løses med en grådig algoritme, og det kan Huffman også.

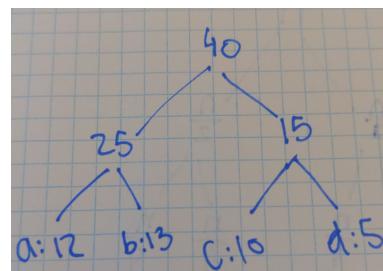
3.2 Huffman

3.2.1 i. Mål og notation

- Vi har f.eks. en tekst, som vi vil repræsentere med binære tal. Nogle bogstaver forekommer oftere end andre
- Komprimering: Vi vil gerne bruge færrest muligt bits
- Problem: $a = 0$, $b = 1$, $c = 01$. Hvad betyder 001 (aab eller ac?)
- Løsning: Vi bruger et træ — stop ved alle blade
- Notation:
 - T — et parsetræ
 - C — sættett af karakterer
 - f_c — frekvensen af karakter c
 - $d_T(c)$ — dybden af c i T
 - $B(T) = \sum_{c \in C} f_c \cdot d_T(c)$ — omkostingen af træet. Siger hvor mange bits vi bruger til den komprimerede tekst. Skal minimeres

3.2.2 ii. Håndkørsel med a:12, b:13, c:10, d:5

- Tag det mindste element og sæt det sammen med det næstmindste



3.2.3 iii. Greedy choice property

- Overordnet: det grådige valg er med i en optimal løsninger.
- Lad
 - x og y — de symboler med mindst frekvens
 - T — et optimalt parsetræ
 - a og b — de to blade med størst dybde i T

- Påstand: Vi kan bytte om på a og x , og stadig have et optimal træ T'
- Bevis
 - Vi omskriver med de nye d_T

$$\begin{aligned} B(T') &= B(T) - f_a \cdot d_T(a) - f_x \cdot d_T(x) + f_x \cdot d_T(a) + f_a \cdot d_T(x) \\ &= B(T) + (f_a - f_x)(d_T(x) - d_T(a)) \end{aligned}$$

- Da x er et mindste element, har at vi: $f_a - f_x \geq 0$
- Da a er et nederste element har vi: $d_T(x) - d_T(a) \leq 0$
- Negativ gange positiv = negativ
- Derfor har vi $B(T') \leq B(T)$, og T' er dermed optimal
- Vi kan dermed altid finde en optimal løsning, hvor x og y er søskende

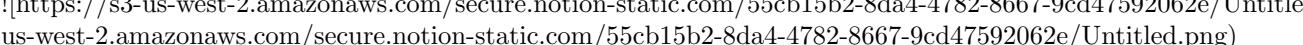
3.2.4 iv. Kort om Lemma om $B(T)$

- Lad W være alle knuder pånær rodknuden.
- Vi har:

$$B(T) = \sum_{c \in C} f_c \cdot d_T(c) = \sum_{w \in W} f_w$$

- Bevise ikke. Giver mening ift. tegningen — frekvens $f : 5$ er inkluderet i 14, 30, 55

3.2.5 v. Optimal delstruktur

- I det første valg, sætter vi x og y sammen — den bliver erstattet af en ”virtuel” knude z
- Vi har $f_z = f_x + f_y$
- Lad C' være det resterende problem: $C' = (C \setminus \{x, y\}) \cup \{z\}$
- Påstand: Hvis det grådige valg er i et optimal træ T_1 for C , så er $T'_1 = T_1 \setminus \{x, y\}$ et optimalt træ for C'
- Tegn:

- Bevis:
 - Lad T'_2 være en optimal løsning for C' .
 - Vi skal derfor vise at $B(T'_1) \leq B(T'_2)$ (da T'_2 per definition er optimal)
 - Pga. Lemma fra før — når vi fjerner to knuder, trækkes disse knuder fra $B(T)$:

$$B(T'_1) = B(T) - f_x - f_y$$

- Lad $T_2 = T'_2 \cup \{x, y\}$ være en løsning for $C \rightarrow$ dermed er $B(T_1) \leq B(T_2)$

3.2.6 vi. Kort om køretid

- Vi bruger Fibonacci Heap (Extract-Min og Insert) — hver operation er $O(\lg n)$
- Der vil maks være $O(n)$ operationer
- Derfor: $O(n \lg n)$

4 Amortized Analysis

4.1 1. Hvad er Amortized Analysis, og hvorfor bruge det?

- Kigge på køretid over mange operationer i stedet for isoleret for hver operation
- Kan give bedre og mere retvisende køretid — gennemsnitlig worst-case
- Følgende eksempler er lidt tænkte, men bruges også i f.eks. analyse af Fibonacci Heaps og disjoint-set forests

4.2 2. Stak med MultiPop

- Vil bruge som eksempel i følgende
- Push(S, x) — tilføjer element — $\Theta(1)$
- Pop(S) — fjerner element; kaldes kun hvis stakken ikke er tom — $\Theta(1)$
- Stack-Empty(S) — finder ud af om stakken er tom — $\Theta(1)$
- MultiPop(S, k) — popper $\hat{k} = \min(|S|, k)$ elementer — benytter Pop og Stack-Empty

4.3 3. Aggregeret analyse

- Vi betrager n operationer, finder den samlede køretid $T(n)$ og den armotiserede køretid er dermed $O\left(\frac{T(n)}{n}\right)$
- Dårlig bound: Man kunne jo sige, at MultiPop tager worst-case $O(n)$ så hvis alle operationer var MultiPop ville $T(n) = n^2$ og armotiseret $O(n)$
- Godt bound: Nu udnytter vi, at der højst kan være n Push-operationer og der kan ikke være flere Pop-operationer (inklusiv dem som bliver kaldt i MultiPop) end Push-operationer. Dermed er $T(n) = n$ og den armotiserede køretid $O(1)$.

4.4 4. Accounting method

- Intuition: Vi lægger penge på et element ved nogle operationer, og ved andre operationer tager vi penge fra elementet. Et element må aldrig blive negativ
- Vi har den faktiske cost c_i (cost isoleret set for i 'te operation) og den armotiserede cost \hat{c}_i (det vi betaler til banken). F.eks.:

Operation	Faktisk cost	Amotiseret cost
Push	1	2
Pop	1	0
MultiPop	$\min(s, k)$	0

- Det gælder, da man ved Push betaler for den tilsvarende Pop
- Det skal gælde at summen af de faktiske omkostninger ikke er større end den armotiserede cost:

$$\sum_{i=0}^n c_i \leq \sum_{i=0}^n \hat{c}_i$$

- Dermed ser for n operationer vi at worst-caser, at betale $2n$. Derfor er armotised cost $\frac{2n}{n} = O(1)$

4.5 5. Potentiale metode

4.5.1 i. Om metoden

- Minder om accounting method — men i stedet for at gemme overkud på elementer, gemmer vi det i "banken"
- Vi har n operationer og strukturen D_i efter den i 'te operation. Potentialefunktionen $\Phi(D_i)$ beskriver pengene der er i banken på dette tidspunkt.
- Den armotiserede cost \hat{c}_i er givet ved den faktiske cost c_i plus forskellen i banken:

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

- $\Phi(D_i) - \Phi(D_{i-1}) > 0$ betyder putte penge i banken
- Ligesom før skal de faktiske cost ikke blive større end den armotiserede cost:

$$\sum_{i=0}^n c_i \leq \sum_{i=0}^n \hat{c}_i \quad (1)$$

- Vi kan skrive summen af armotiserede cost som:

$$\begin{aligned} \sum_{i=1}^n \hat{c}_i &= \sum_{i=1}^n (c_i + \Phi(D_i) - \Phi(D_{i-1})) \\ &= c_1 + \Phi(D_1) - \Phi(D_0) \\ &\quad + c_2 + \Phi(D_2) - \Phi(D_1) \\ &\quad + \vdots \\ &\quad + c_n + \Phi(D_n) - \Phi(D_{n-1}) \\ &= \left(\sum_{i=1}^n c_i \right) + \Phi(D_n) - \Phi(D_0) \end{aligned}$$

- Derfor ser vi at (1) er opfyldt, når $\Phi(D_n) \geq \Phi(D_0)$

4.5.2 ii. Eksempel med stack

- Vi vil først finde en $\Phi(D_i)$ og derefter vise at $\Phi(D_n) \geq \Phi(D_0)$
- Vi vælger $\Phi(D_i) = |D_i|$ hvor $|D_i|$ er antallet af elementer på stakken
- Vi starter med ingen elementer, så $\Phi(D_0) = 0$. Vi kan aldrig få under nul elementer, så $\Phi(D_n) \geq \Phi(D_0)$
- Vi skal nu finde upper bound

– Push: der tilføjes et element til stacken, så:

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

– Pop: der fjernes et element så:

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

– MultiPop: vi popper $\hat{k} > 0$ elementer fra stacken. Så:

$$\Phi(D_i) - \Phi(D_{i-1}) = -\hat{k}$$
 (vi fjerner \hat{k} elementer)

$$c_i = \hat{k}$$
 (multipop vil bruge \hat{k} Pop operationer)

Dermed:

–

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

- Vi kan dermed sige at $\hat{c}_i \leq 2$. Så den amortiserede køretid er $\frac{2n}{n} = O(1)$

5 Fibonacci Heaps

5.1 1. Hvad er det?

- God amortiseret køretid, mange af operationerne er i konstant tid.
- Bruges især til at implementere prioritetskører. Smart til Dijkstra's algoritme og Prim's algoritme
- Værd at bemærke at Fibonacci heaps ofte er langsommere i praksis
- Jeg vil tale om en Min-Heap, men man kunne på samme måde lave en Max-Heap

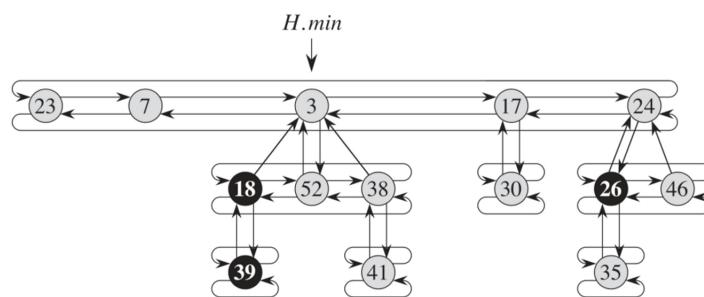
5.2 2. Struktur

5.2.1 i. Egenskaber

- Tegn heap (fra håndkørsel-eksempel)
- Min-Heap egenskab: $x.key \geq x.p.key$
- $H.\min$ er en pointer til rodknuden.
- En tom Fibonacci Heap har $H.\min = NIL$

5.2.2 ii. Attributter

- Hver knude x har følgende attributter:
 - $x.p$ — knudens forælder
 - $x.child$ — et vilkårligt barn
 - $x.left$ og $x.right$ — brugt til cirkulær, double linked list af søskende (gælder også rodknuder). Sådan her:



- $x.degree$ — antallet af børn i listen af børn
- $x.mark$ — fortæller om knuden har mistet ét barn siden den sidst blev gjort til et barn af en anden knude. Nye knuder er umarkerede.

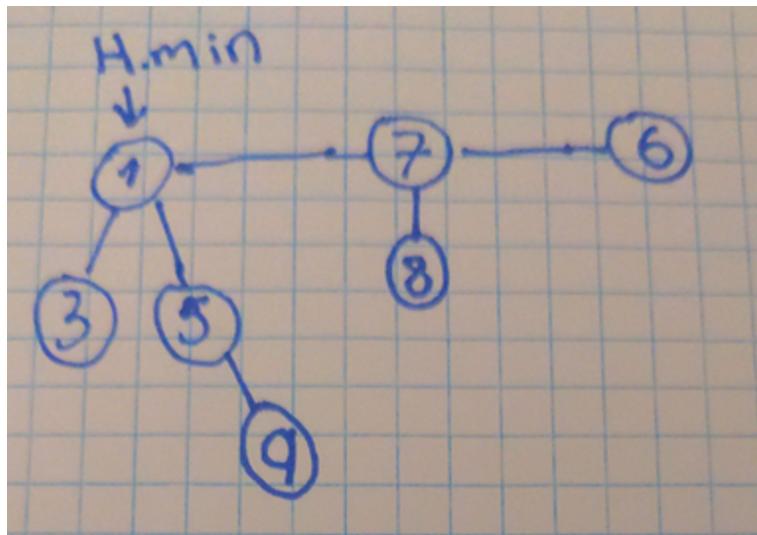
5.2.3 iii. Operationer

- Nævn operationerne og hvad de gør
- Alle operationer er $O(1)$ på nær Extract-Min og Delete som er $O(\lg n)$

Operation	Amortiseret køretid
Make-Heap()	$\Theta(1)$
Insert(H, x)	$\Theta(1)$
Minimum(H)	$\Theta(1)$
Union(H_1, H_2)	$\Theta(1)$
Decrease-Key(H, x, k)	$\Theta(1)$
Extract-Min(H)	$\Theta(\lg n)$
Delete(H, x)	$\Theta(\lg n)$

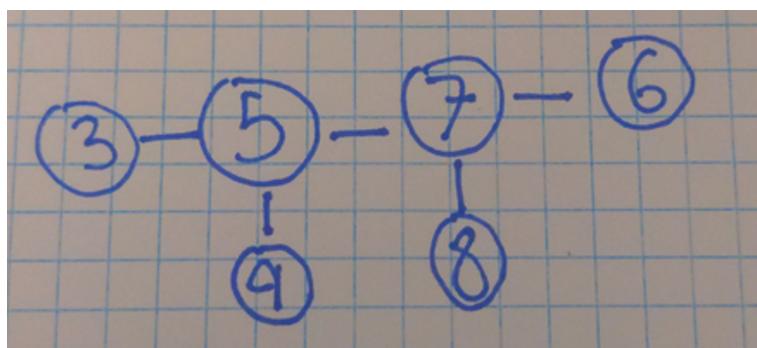
- H = heap, x = knude og k = ny key

5.3 3. Operationen Extract-Min



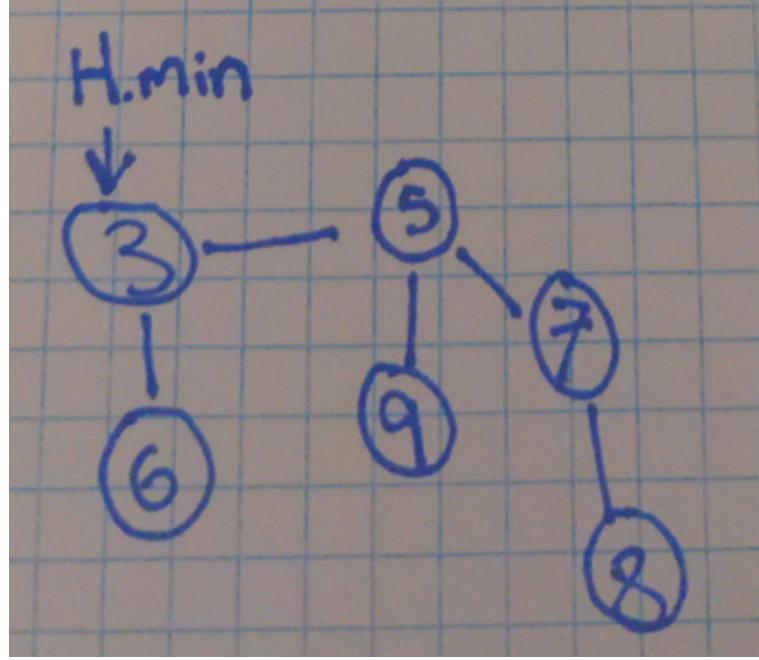
5.3.1 i. Håndkørsel med ovenstående input

- Når vi kører Extract-Min skal vi uanset hvad finde den nye mindste rodknude til at være $H.min$. Derfor er det oplagt at "rydde op" her.
- Extract-Min fungerer på følgende måde:
 1. Fjern $H.min$ fra rodlisten (returneres til sidst)
 2. Fly alle børn op i rodlisten



3. Benyt Consolidate — rydder op så hvert rodelement har unik degree

- Consolidate fungerer på følgende måde:
 1. Find $D(n)$ — det maksimale antal børn en knude kan have. Den er $\Theta(\lg n)$
 2. Lav et array $A = [0 \dots D(n)]$. Felt i bruges som en pointer til en knude $x.degree = i$
 3. For hver knude i rodlisten slår vi den sammen med en anden rodknude af samme degree



5.3.2 ii. Potentialefunktionen Φ

- Vi definerer potentialefunktionen således:

$$\Phi(H) = t(H) + 2m(H)$$

Hvor $t(H)$ er antallet af rodknuder og $m(H)$ er antallet af markerede knuder.

- Gyldighed: $\Phi(H_o) = 0$, da der ikke er nogen knuder (dermed ingen markerede eller rodknuder)
 $\Phi(H_o) \geq 0$, da der ikke vil kunne være et negativt antal knuder

5.3.3 iii. Amortiseret køretid

- Faktisk cost c_i :
 - Flytte børn op i rodlisten — $O(D(n))$
 - Initialisere A — $O(D(n))$
 - Finde ny $H.\min$ — $O(D(n))$
 - Gennemløb af knuder (opflyttede børn + gamle rodknuder) i Consolidate — $O(D(n) + t(H))$
 - Samlet set:
- Armodiseret cost \hat{c}_i :
 - Intuition: Hver gang vi reducerer antallet af rodknuder, bliver det betalt af et fald i $t(H)$ i potentialet.
 - Vi skal finde følgende:

$$\hat{c}_i = c_i + \Phi(H') - \Phi(H) = c_i + [t(H') + 2m(H')] - [t(H) + 2m(H)]$$

- Vi ved fra (1) at $c_i \leq t(H) + D(n)$
- Vi har $t(H') \leq D(n) + 1$ da alle rodknuder har forskellig degree
- Vi har $m(H') = m(H)$ da vi ikke ændrer nogle markeringer
- Dermed kan vi omskrive ovenstående:

$$\begin{aligned} &\leq D(n) + t(H) + [(D(n) + 1) + 2m(H)] - [t(H) + 2m(H)] \\ &= 2D(n) + 1 \\ &= O(D(n)) \end{aligned}$$

- Man kan bevise at $D(n) = \Theta(\lg n)$, så derfor er den armotiserede køretid $O(\lg n)$

6 Balanced Binary Search Trees

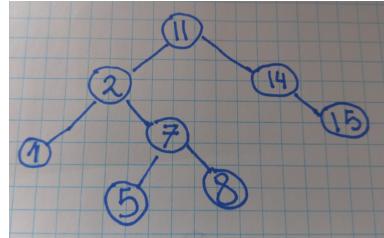
6.1 1. Hvad er Binary Search Trees?

6.1.1 i. Insert, Delete, Search

- En effektiv datastruktur, som kan Insert, Delete og Search i $O(h)$. I visse tilfælde vil $h = O(\lg n)$
 - ved tilfældige indsættelser og ved Red-Black trees
- Bliver f.eks. brugt i plane-sweep algoritmen (der ser om linjesegmenter skærer hinanden)

6.1.2 ii. Struktur og BST egenskab — eksempel $\langle 11, 2, 1, 7, 5, 8, 14, 15 \rangle$

- Det er et træ
- Hver knude x har attributterne key , $left$, $right$ og p (parent)
- Vi sørger for at $x.key \geq x.left.key$ og $x.key < x.right.key$



6.2 2. Red-black trees

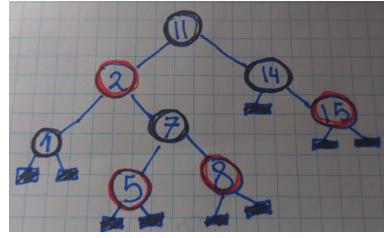
6.2.1 i. Hvorfor?

- Worst-case højde for normalt BST: $h = O(n)$
- Vi vil ned på $h = O(\lg n)$

6.2.2 ii. Struktur og 5 egenskaber

- Ligesom BST — med ekstra attribut $color$ (rød eller sort)
- Skal opfylde følgende 5 egenskaber:
 1. Hver knude er enten rød eller sort
 2. Rodknuden er sort
 3. Alle blade er NIL , som vi betragter som sort
 4. Hvis en knude er rød, så er begge dens børn sorte
 5. Enhver knude x har samme antal sorte knuder i simple veje ned til dens NIL -efterkommere. Denne kalder vi black-height, $bh(x)$

- Tegn videre på træet fra før — nævn $bh(root) = 2$



6.2.3 iii. Bevis af lemma

- Dette lemma skal vi bruge til at bevise højden senere
- Lad $s(x)$ betegne antallet af interne knuder for (del)træet med rodknude x .
- Påstand: Vi vil ved stærk induktion vise at antallet af interne knuder er mindst:

$$s(x) \geq 2^{bh(x)} - 1 \quad (3)$$

- Base case — $h(x) = 0$
 - Venstre side af (1): $s(x) = 0$, da x i dette tilfælde må være et *NIL*-blad
 - Højre side af (1): *NIL*-blad har $bh(x) = 0$ og dermed $2^{bh(x)} - 1 = 0$
 - Vi ser at (1) er overholdt
- Induktionsskridt — $h(x) = k$
 - Dermed kan højden af børne-træerne være maks $k - 1$: $h(x.left) \leq k - 1$ og $h(x.right) \leq k - 1$
 - Antallet af elementer er givet ved:

$$s(x) = s(x.left) + s(x.right) + 1$$

– Pga. vores induktionsantagelse ved vi, at (1) gælder for børnene. Vi kan dermed skrive:

$$\geq [2^{bh(x.left)} - 1] + [2^{bh(x.right)} - 1] + 1 = 2^{bh(x.left)} + 2^{bh(x.right)} - 1$$

– Hvis et barn er sort, vil det have højde $bh(x.child) = bh(x) - 1$. Hvis det er rødt, vil det have $bh(x.child) = bh(x)$. Derfor får vi:

$$\geq 2^{bh(x)-1} + 2^{bh(x)-1} - 1 = 2^{bh(x)} - 1$$

– Vi har dermed vist lemmaet (1)

6.2.4 iv. Bevis af $h = O(\dots)$

- Påstand: $h = O(\lg n)$ (RB-tree med n interne knuder)
- Bevis:
 - Husk egenskab nummer 4: Hvis en knude er rød, så er begge dens børn sorte
 - Dermed kan antallet af røde knuder fra rodknuden r ikke være mere end $\frac{h}{2}$. Derfor må antallet af sorte knuder være de resterende, altså minimum $\frac{h}{2}$. Dvs:

$$bh(r) \geq \frac{h}{2} \quad (4)$$

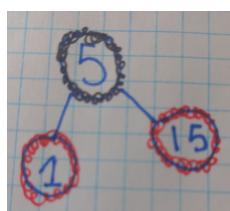
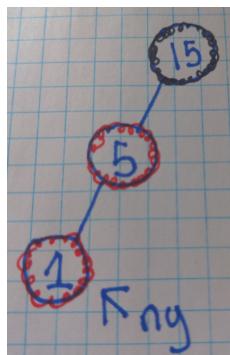
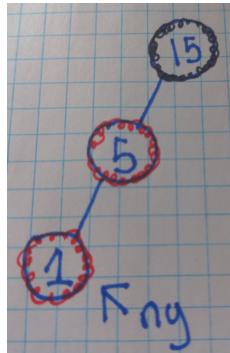
– Vi samler (1) og (2) — og omskriver:

$$n = s(r) \geq 2^{bh(r)} - 1 \geq 2^{h/2} - 1n + 1 \geq 2^{h/2}2\lg(n+1) \geq h$$

– Vi har dermed vist at $h = O(\lg n)$

6.2.5 v. Kort om opretholdelse af egenskaber — Rotationer

- Vi skal altid overholde de 5 egenskaber — men efter Insert og Delete vil de kunne være brudt.
- I visse tilfælde skal vi derfor lave om i strukturen på træet ved at omarangere deltræer
- Vi vil gerne have store deltræer så langt op som muligt
- Der er to typer rotationer — Left og Right. Der er et sæt regler for hvordan man bruger dem op igennem træet.
- Simpelt eksempel på Right rotate:



7 Minimum Spanning Trees

7.1 1. Beskrivelse af problemet

- Tegn grafen som skal bruges til håndkørsel
- Givet en vægtet graf $G = (V, E, w)$, skal man finde et subset $T \subseteq E$ der forbinder hele grafen. Der må ikke være kredse.
- Grafen er sammenhængende og den er ikke-orienteret (undirected)
- Man skal minimere summen af vægtene i T .
- Man kan f.eks. bruge det til at lave en approximation af traveling salesman problemet

7.2 2. Generisk algoritme

7.2.1 i. Termonologi

- Cut: En opdeling af grafen i to sæt af knuder, $C = (X, V - X)$
- Krydse et cut: En kant (u, v) hvor $u \in X$ og $v \in V - X$ (eller omvendt)
- Let kant: En kant (u, v) der krydser et snit, og har minimal vægt
- Et cut respekterer et sæt kanter A : Når et cut ikke resulterer i, at nogle af kanterne i sættet A krydser cuttet.
- Sikker kant for A : En kant vi kan tilføje til subsettet A , som vedligeholder at A stadig er et subset af et Minimum Spanning Tree

7.2.2 ii. Algoritmen

- Vi laver en tom mængde A
- Så længe A ikke er et Spanning Tree, så putter vi en sikker kant i A .
 - Se pseudokode

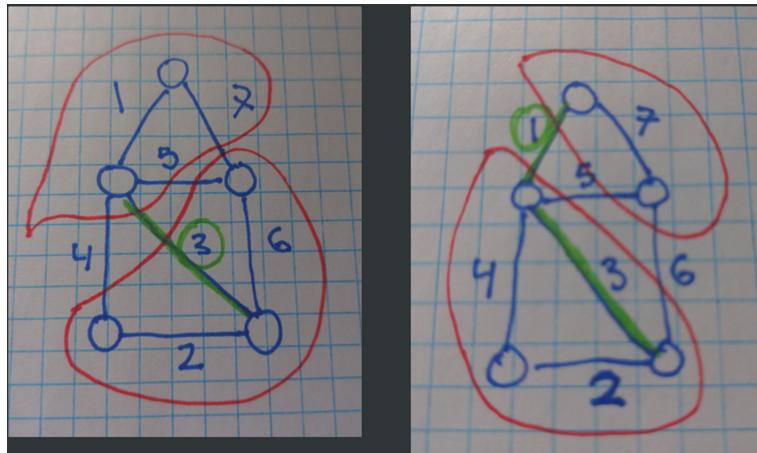
Algorithm 1: Generic-MST

```

1 Generic-MST( $G, w$ )
2    $A = \emptyset$ 
3   while  $A \neq$  et Spanning Tree
4     | find en kant  $(u, v)$  sikker for  $A$ 
5     |  $A = A \cup \{(u, v)\}$ 
6   return  $A$ 

```

- Hvordan vælger vi cut og sikker kant? Det afhænger af algoritme



7.2.3 iii. Bevis af korrekthed

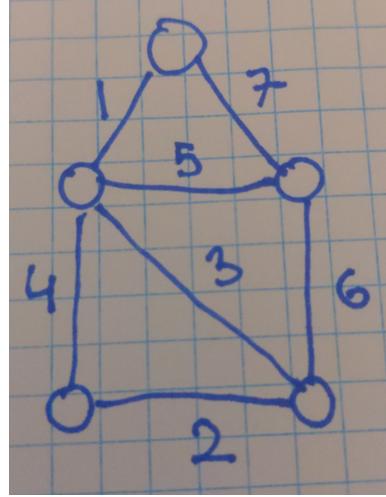
- Vi vil bevise løkkevarianten.
- Lad:
 - $A \subseteq T$ hvor T er et eller andet MST.
 - C være et cut der respekterer A .
 - $e = (u, v)$ være en let kant der krydser C .
- Påstand:

- $(A + e) \subseteq T'$ hvor T' er et MST (ikke nødvendigvis samme som T)

- Bevis ved modstrid:

- Antag at der ikke findes et T' der indeholder $(A + e)$
- Det er givet, at T indeholder A , men vil så ikke indeholde $e = (u, v)$
- T indeholde en sti $u \rightsquigarrow v$ — og denne må krydse C , der blev brugt til at vælge e
- T må dermed indeholde en anden kant f der krydser C .
- Da e var en let kant, har vi $|e| \leq |f|$.
- Vi kan erstatte f med e : $T' = T - f + e$
- Der er modstrid, da vi antog at der ikke fandtes et T' der indeholdt e

7.3 3. Kruskals algoritme



7.3.1 i. Implementation med disjoint sets

1. Lav et tomt sæt A
2. Lav et sæt for hver knude
3. Gennemløb alle kanter (u, v) , fra mindst vægt til størst
 - Er u og v er i samme sæt?
 - Hvis ikke, så $Union(u, v)$ og tilføj (u, v) til A
4. Se pseudokode

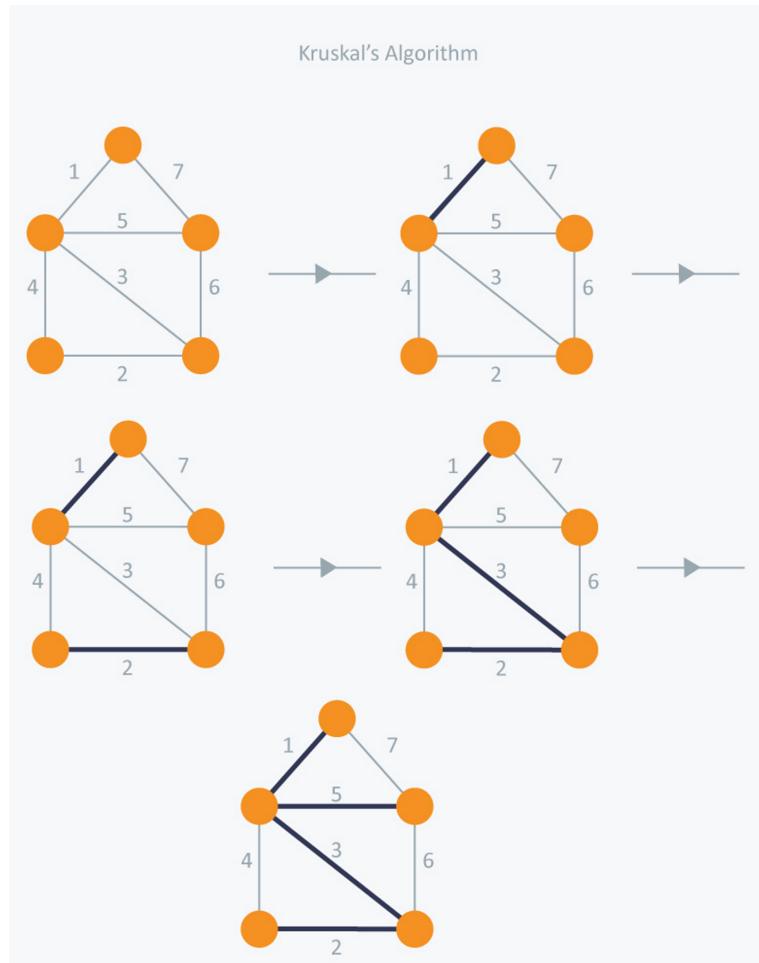
Algorithm 2: MST-Kruskal

```

1 MST-Kruskal( $G, w$ )
2    $A = \emptyset$ 
3   foreach vertex  $v \in G.V$ 
4     Make-Set( $v$ )
5   sort the edges of  $G.E$  into increasing order by weight  $w$ 
6   foreach edge  $(u, v) \in G.E$  in sorted order
7     if  $\text{Find-Set}(u) \neq \text{Find-Set}(v)$ 
8        $A = A \cup \{(u, v)\}$ 
9        $\text{Union}(u, v)$ 
10  return  $A$ 

```

7.3.2 ii. Håndkørsel med ovenstående input

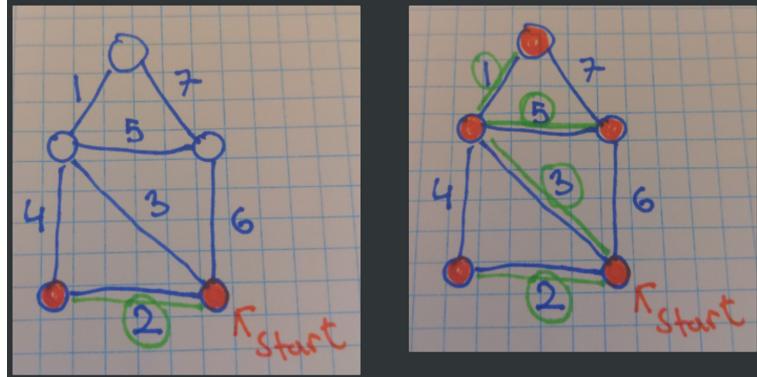


7.3.3 iii. Køretid (kort)

- Vi bruger disjoint set forrests (f.eks. implementeret med link by rank)
- Vi skal
 - sortere kanterne — $O(E \lg E)$
 - $O(E)$ Find-Set — $O(1)$ per operation
 - $O(E)$ Link — $O(\lg V)$ per operation
 - V Make-Set — $O(1)$ per operation
 - Bemærk at $O(\lg E) = O(\lg V^2) = O(2 \lg V) = O(\lg V)$
 - Derfor er køretiden $O(E \lg E) = O(E \lg V)$

7.4 4. Prims algoritme

7.4.1 i. Håndkørsel med samme graf som før — start nederst til højre



MST-PRIM(G, w, r)

```

1  for each  $u \in G.V$ 
2     $u.key = \infty$ 
3     $u.\pi = \text{NIL}$ 
4   $r.key = 0$ 
5   $Q = G.V$ 
6  while  $Q \neq \emptyset$ 
7     $u = \text{EXTRACT-MIN}(Q)$ 
8    for each  $v \in G.Adj[u]$ 
9      if  $v \in Q$  and  $w(u, v) < v.key$ 
10         $v.\pi = u$ 
11         $v.key = w(u, v)$ 
```

7.4.2 ii. Køretiden er $O(\dots)$

- Køretiden hvis man bruger Fibonacci heaps, er $O(E + V \lg V)$
- Den er derfor især god hvis man har mange kanter

8 Shortest Paths

8.1 1. Beskrivelse af problemet

- Single Source Shortest Path: Givet en graf $G = (V, E, w)$, skal vi finde den korteste vej fra en knude til alle de andre
- Kan bruges til mange ting, f.eks. netværk og Google Maps

8.2 2. Generelt for algoritmerne — initialisering og relaxation

- Tegn grafen fra håndkørsel, uden de røde tal

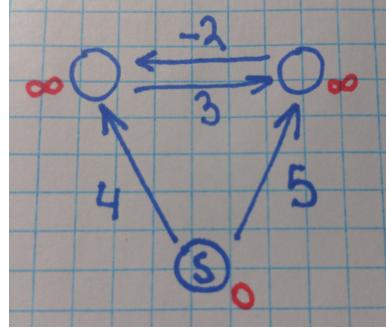
- Initialisering:

- $s.d = 0$
- For de resterende $v \in V$:
- Koreste vej estimat $v.d = \infty$
- Forgænger $v.\pi = \text{NIL}$

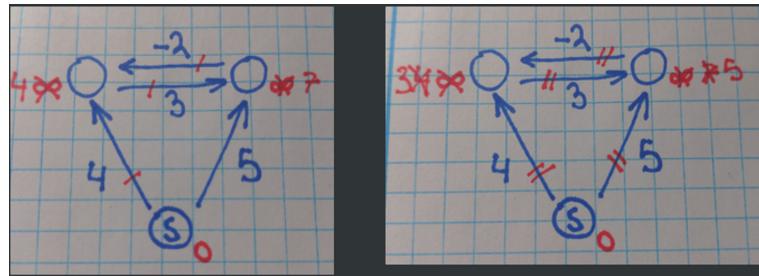
- Relaxation: Vi ser om vi kan få en bedre $v.d$

- Betragt en kant (u, v)
- Hvis $v.d > u.d + w(u, v)$
 - * sæt $v.d = u.d + w(u, v)$
 - * sæt $v.\pi = u$

8.3 3. Bellman-Ford algoritme



8.3.1 i. Håndkørsel med ovenstående input



Se pseudokode

Algorithm 1: MST-Kruskal

```

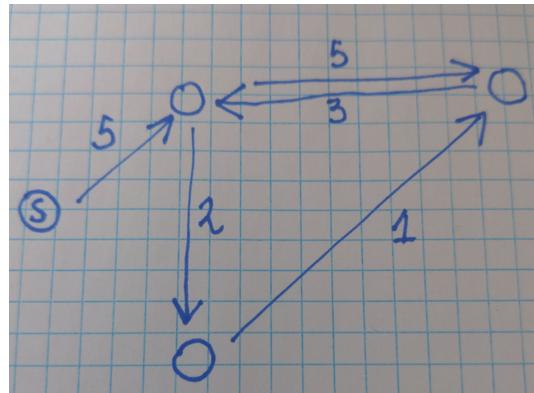
1 Bellman-Ford( $G, w, s$ )
2   Initialize-Single-Source( $G, s$ )
3   for  $i = 1$  to  $|G.V| - 1$ 
4     foreach edge  $(u, v) \in G.E$ 
5       Relax( $u, v, w$ )
6   foreach edge  $(u, v) \in G.E$ 
7     if  $v.d > u.d + w(u, v)$ 
8       return FALSE
9   return TRUE

```

8.3.2 ii. Køretid (kort)

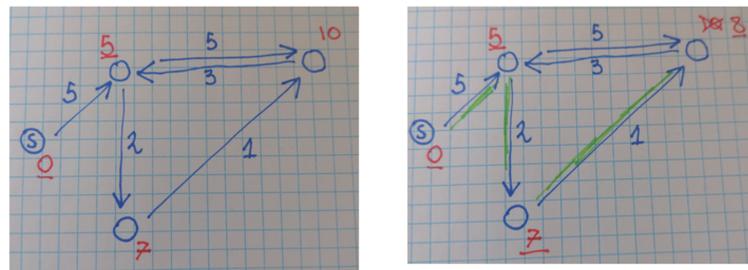
- Vi skal besøge alle $|E|$ kanter $|V| - 1$ gange.
- Derfor er køretiden $O(EV)$
- Bemærk at $E = O(V^2)$, så køretiden er $O(V^3)$

8.4 4. Dijkstra's algoritme



8.4.1 i. Håndkørsel med ovenstående input

- Det er en grådig algoritme
- Vi kan ikke have negative vægte

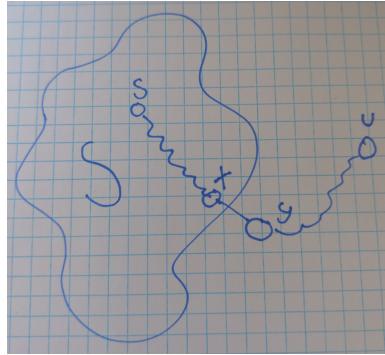


8.4.2 ii. Køretiden er $O(\dots)$

- $|V|$ Extract-Min i hver $O(\lg V)$
- $|E|$ Decrease-Key i hver $O(1)$ amortiseret
- Samlet: $O(V \lg V + E)$

8.4.3 iii. Bevis af korrekthed

- Efter at have besøgt en knude, så tilføjer vi den til mængden S .
- Påstand: Når $u \in S$, så har vi en optimal løsning $u.d = \delta(s, u)$
- Bevis ved induktion:
 - Base case:
 - * Trivialt, at $s.d = \delta(s, s) = 0$
 - Induktionsskridt:
 - * Visualtiseret:



- * Stærk induktion — alle knuder der allerede er i S , er en korteste vej
- * Antag for modstrid, at $u.d$ ikke er optimal: $u.d > \delta(s, u)$
- * Der må derfor findes en korteste vej: $P : s \rightsquigarrow u$ (ellers må $u.d = \delta(s, u) = \infty$)
- * Lad $(x, y) \in P$ hvor x er den sidste knude i S (sidste knude vi kender kortest vej til)
- * Da x blev tilføjet til S , blev (x, y) relaxed

$$y.d = \delta(s, x) + w(x, y)$$

- * Da (x, y) er en del af den korteste vej:

$$= \delta(s, y)$$

- * Da y ligger før u på vejen, og alle vægte er positive, har vi:

$$\leq \delta(s, u)$$

- * På grund af vores antagelse at $u.d$ ikke er optimal:

$$< u.d$$

- * Vi har dermed $y.d < u.d$
- * Når algoritmen vælger at tilføje u til S , så er det efter en Extract-Min. Derfor må vi have $u.d \leq y.d$
- * Dermed har vi modstrid, og vi har vist at $u.d = \delta(s, u)$