

# Grådige algoritmer

# Grådige algoritmer

Et generelt algoritme-konstruktionsprincip (“paradigme”) for optimeringsproblemer.

# Grådige algoritmer

Et generelt algoritme-konstruktionsprincip (“paradigme”) for optimeringsproblemer.

Ideen er simpel:

- ▶ Opbyg løsningen bid for bid ved hele tiden af vælge hvad der lige nu ser ud som “bedste valg” (uden at tænke på resten af løsningen).

Dvs. man håber på at **lokal** optimering giver **global** optimering.

# Grådige algoritmer

Et generelt algoritme-konstruktionsprincip (“paradigme”) for optimeringsproblemer.

Ideen er simpel:

- ▶ Opbyg løsningen bid for bid ved hele tiden af vælge hvad der lige nu ser ud som “bedste valg” (uden at tænke på resten af løsningen).

Dvs. man håber på at **lokal** optimering giver **global** optimering.

Mere præcist kræver metoden en **definition** af “bedste valg” (også kaldet “grådig valg”), samt et **bevis** for gentagen brug af dette ender med en optimal løsning.

# Grådige algoritmer

Et generelt algoritme-konstruktionsprincip (“paradigme”) for optimeringsproblemer.

Ideen er simpel:

- ▶ Opbyg løsningen bid for bid ved hele tiden af vælge hvad der lige nu ser ud som “bedste valg” (uden at tænke på resten af løsningen).

Dvs. man håber på at **lokal** optimering giver **global** optimering.

Mere præcist kræver metoden en **definition** af “bedste valg” (også kaldet “grådig valg”), samt et **bevis** for gentagen brug af dette ender med en optimal løsning.

Mange grådige algoritmer kan forklares som specialtilfælde af begrebet “Matroide” (afsnit 16.4, ikke pensum), men vi vil nøjes med at tænke på grådige algoritmer som et løst princip, hvor “bedste valg” og korrekthedsbevis skal opfindes individuelt fra problem til problem.

## Eksempel: et simpelt skeduleringsproblem

Input: en samling aktiviteter, hver med en starttid og sluttid.

Output: en størst mulig mængde af ikke-overlappende aktiviteter.

# Eksempel: et simpelt skeduleringsproblem

Input: en samling aktiviteter, hver med en starttid og sluttid.

Output: en størst mulig mængde af ikke-overlappende aktiviteter.

Grådigt forslag:

Sorter aktiviteter efter stigende sluttid

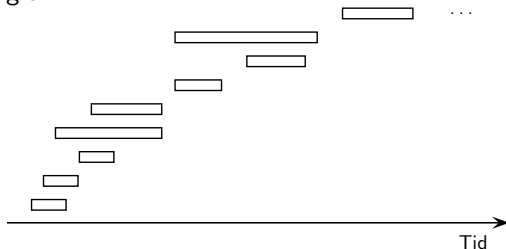
**For** hver aktivitet  $a$  taget i den rækkefølge:

**If**  $a$  overlapper allerede valgte aktiviteter:

Skip  $a$

**Else**

Vælg  $a$



# Analyse

Vi vil vise følgende invariant:

Der findes en optimal løsning  $OPT$  som indeholder de af algoritmen indtil nu valgte aktiviteter.



# Analyse

Vi vil vise følgende invariant:

Der findes en optimal løsning OPT som indeholder de af algoritmen indtil nu valgte aktiviteter.

Når algoritmen er færdig, følger korrekthed af denne invariant:

Algoritmens valgte aktiviteter ligger inden i en optimal løsning OPT. Pga. algoritmens virkemåde vil alle ikke-valgte aktiviteter overlappende en af de valgte. Altså kan algoritmens løsning ikke udvides og stadig være en løsning. Specielt kan OPT ikke være større end algoritmens valgte løsning, som derfor er lig med OPT.

# Analyse

Bevis for invariant ved induktion:

**Basis:**

Klart før første iteration af **for**-løkken (da ingen aktiviteter er valgt).

**Skridt:**

Lad OPT være den optimale løsning fra induktionsudsagnet før iterationen. Vi skal vise at der findes en optimal løsning OPT' i induktionsudsagnet efter iterationen.

**If-case:** Her vælger algoritmen intet nyt, så OPT kan bruges som OPT'.

# Analyse

## Else-case:

Lad input efter sortering være aktiviteterne  $a_1, a_2, \dots, a_n$ , lad  $a_i$  være algoritmens senest valgte aktivitet og lad  $a_j$  være aktiviteten, som vælges i denne iteration.

Pga. invarianten indeholder OPT  $a_i$ . I OPT kan  $a_i$  ikke være den sidste (for så kunne OPT udvides med  $a_j$ ). Lad  $a_k$  være den næste i OPT efter  $a_i$ . Da  $a_j$  er den første aktivitet efter  $a_i$  som ikke overlapper  $a_i$ , må  $j \leq k$ .

Hvis  $j = k$  kan OPT genbruges som OPT'. Ellers skifter vi  $a_j$  i OPT ud med  $a_i$ . Dette giver ikke overlap med andre aktiviteter i OPT (de stopper enten før  $a_i$  eller starter efter  $a_k$  og dermed efter  $a_j$ ) og bevarer størrelsen. Vi har derfor en ny optimal løsning OPT' som opfylder invarianten.

[NB: I første iteration findes  $a_i$  ikke, men vi kan sætte  $j = 1$  og sige noget tilsvarende.]

# Analyse

## Else-case:

Lad input efter sortering være aktiviteterne  $a_1, a_2, \dots, a_n$ , lad  $a_i$  være algoritmens senest valgte aktivitet og lad  $a_j$  være aktiviteten, som vælges i denne iteration.

Pga. invarianten indeholder OPT  $a_i$ . I OPT kan  $a_i$  ikke være den sidste (for så kunne OPT udvides med  $a_j$ ). Lad  $a_k$  være den næste i OPT efter  $a_i$ . Da  $a_j$  er den første aktivitet efter  $a_i$  som ikke overlapper  $a_i$ , må  $j \leq k$ .

Hvis  $j = k$  kan OPT genbruges som OPT'. Ellers skifter vi  $a_j$  i OPT ud med  $a_i$ . Dette giver ikke overlap med andre aktiviteter i OPT (de stopper enten før  $a_i$  eller starter efter  $a_k$  og dermed efter  $a_j$ ) og bevarer størrelsen. Vi har derfor en ny optimal løsning OPT' som opfylder invarianten.

[NB: I første iteration findes  $a_i$  ikke, men vi kan sætte  $j = 1$  og sige noget tilsvarende.]

Køretid:

# Analyse

## Else-case:

Lad input efter sortering være aktiviteterne  $a_1, a_2, \dots, a_n$ , lad  $a_i$  være algoritmens senest valgte aktivitet og lad  $a_j$  være aktiviteten, som vælges i denne iteration.

Pga. invarianten indeholder OPT  $a_i$ . I OPT kan  $a_i$  ikke være den sidste (for så kunne OPT udvides med  $a_j$ ). Lad  $a_k$  være den næste i OPT efter  $a_i$ . Da  $a_j$  er den første aktivitet efter  $a_i$  som ikke overlapper  $a_i$ , må  $j \leq k$ .

Hvis  $j = k$  kan OPT genbruges som OPT'. Ellers skifter vi  $a_j$  i OPT ud med  $a_i$ . Dette giver ikke overlap med andre aktiviteter i OPT (de stopper enten før  $a_i$  eller starter efter  $a_k$  og dermed efter  $a_j$ ) og bevarer størrelsen. Vi har derfor en ny optimal løsning OPT' som opfylder invarianten.

[NB: I første iteration findes  $a_i$  ikke, men vi kan sætte  $j = 1$  og sige noget tilsvarende.]

Køretid: Sortering +  $O(n)$ .

# Rygsæksproblemet

Rygsæk som kan bære  $W$  kg.

Ting med værdi og vægt.

Ting nr. $i$	1	2	3	4	5	6	7
Vægt $w_i$	4	6	2	15	7	4	5
Værdi $v_i$	45	32	12	50	23	9	15

# Rygsæksproblemet

Rygsæk som kan bære  $W$  kg.

Ting med værdi og vægt.

Ting nr. $i$	1	2	3	4	5	6	7
Vægt $w_i$	4	6	2	15	7	4	5
Værdi $v_i$	45	32	12	50	23	9	15

Mål: tag mest mulig værdi med uden at overskride vægtgrænsen.

# Rygsæksproblemet

“Fractional” version af problemet (dele af ting kan medtages i rygsækken) kan løses med en grådig algoritme: vælg tingene efter aftagende “nytte” = værdi/vægt. Et simpelt udskiftningsargument viser, at den optimale løsning kun kan være som den af algoritmen valgte.



# Rygsæksproblemet

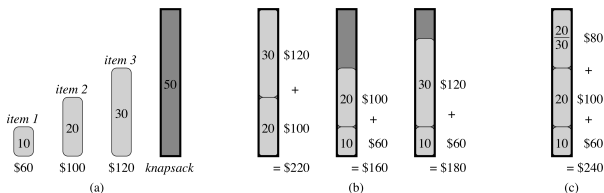
“Fractional” version af problemet (dele af ting kan medtages i rygsækken) kan løses med en grådige algoritme: vælg tingene efter aftagende “nytte” = værdi/vægt. Et simpelt udskiftningsargument viser, at den optimale løsning kun kan være som den af algoritmen valgte.

NB: Denne grådige algoritme virker IKKE for 0-1 versionen af problemet (hvor kun hele ting kan medtages):

# Rygsæksproblemet

“Fractional” version af problemet (dele af ting kan medtages i rygsækken) kan løses med en grådig algoritme: vælg tingene efter aftagende “nytte” = værdi/vægt. Et simpelt udskiftningsargument viser, at den optimale løsning kun kan være som den af algoritmen valgte.

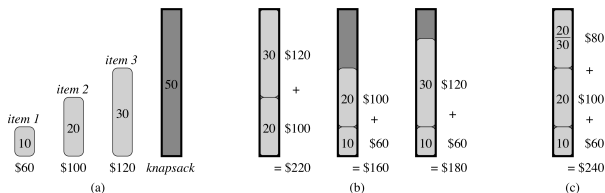
NB: Denne grådige algoritme virker IKKE for 0-1 versionen af problemet (hvor kun hele ting kan medtages):



# Rygsæksproblemet

“Fractional” version af problemet (dele af ting kan medtages i rygsækken) kan løses med en grådige algoritme: vælg tingene efter aftagende “nytte” = værdi/vægt. Et simpelt udskiftningsargument viser, at den optimale løsning kun kan være som den af algoritmen valgte.

NB: Denne grådige algoritme virker IKKE for 0-1 versionen af problemet (hvor kun hele ting kan medtages):



Eksempel på at “grådige valg” ikke bare kan antages at virke for alle problemer (lokal optimering giver ikke altid global optimering).

# Bitmønstre

`01101011`0001100101011011...

Bitmønstre skal **fortolkes** for at have en betydning:

- ▶ Bogstaver
- ▶ Tal (heltal, kommatall)
- ▶ Computerinstruksjon (program)
- ▶ Pixels (billedfil)
- ▶ Amplitude (lydfil)
- ▶ ⋮

# Bitmønstre

`01101011`0001100101011011...

Bitmønstre skal **fortolkes** for at have en betydning:

- ▶ Bogstaver
- ▶ Tal (heltal, kommatall)
- ▶ Computerinstruksjon (program)
- ▶ Pixels (billedfil)
- ▶ Amplitude (lydfil)
- ▶ ⋮

Fokus i dag: bogstaver (og andre tegn).

# Repræsentation af tegn

En klassisk repræsentation: ASCII.

```
⋮  
a: 01100001  
b: 01100010  
c: 01100011  
d: 01100100  
⋮
```

Alle tegn fylder 8 bits (fixed-width codes).

# Huffman-koder

Er ASCII-kodning den kortest mulige repræsentation af en fil af tegn?

# Huffman-koder

Er ASCII-kodning den kortest mulige repræsentation af en fil af tegn?

Det kommer an på filens indhold!



# Huffman-koder

Er ASCII-kodning den kortest mulige repræsentation af en fil af tegn?

Det kommer an på filens indhold! Eksempel:

	a	b	c	d	e	f
Frequency (in thousands)	45	13	12	16	9	5
Fixed-length codeword	000	001	010	011	100	101
Variable-length codeword	0	101	100	111	1101	1100

# Huffman-koder

Er ASCII-kodning den kortest mulige repræsentation af en fil af tegn?

Det kommer an på filens indhold! Eksempel:

	a	b	c	d	e	f
Frequency (in thousands)	45	13	12	16	9	5
Fixed-length codeword	000	001	010	011	100	101
Variable-length codeword	0	101	100	111	1101	1100

Fixed-width version:

$$3 \cdot (45.000 + 13.000 + \dots + 5.000) = 300.000 \text{ bits}$$

Variable-width version:

$$1 \cdot 45.000 + 3 \cdot 13.000 + \dots + 4 \cdot 5.000 = 224.000 \text{ bits}$$

# Huffman-koder

Er ASCII-kodning den kortest mulige repræsentation af en fil af tegn?

Det kommer an på filens indhold! Eksempel:

	a	b	c	d	e	f
Frequency (in thousands)	45	13	12	16	9	5
Fixed-length codeword	000	001	010	011	100	101
Variable-length codeword	0	101	100	111	1101	1100

Fixed-width version:

$$3 \cdot (45.000 + 13.000 + \dots + 5.000) = 300.000 \text{ bits}$$

Variable-width version:

$$1 \cdot 45.000 + 3 \cdot 13.000 + \dots + 4 \cdot 5.000 = 224.000 \text{ bits}$$

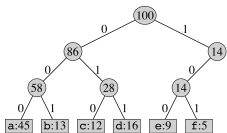
Muligt ønske: kort(est mulig?) repræsentation af en fil. Sparer plads på disk, tid på transport over netværk.

# Prefix-kode = træer

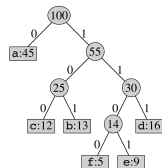
Kodeord = sti i binært træ: 0 ~ gå til venstre, 1 ~ gå til højre

Prefix(-fri) kode: ingen kode for et tegn er starten (prefix) af koden for et andet tegn ( $\Rightarrow$  dekodning utvetydig). Så tegn svarer til knuder med nul børn (blade).

	a	b	c	d	e	f
Frequency (in thousands)	45	13	12	16	9	5
Fixed-length codeword	000	001	010	011	100	101
Variable-length codeword	0	101	100	111	1101	1100



(a)



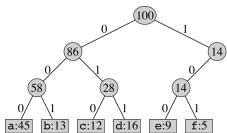
(b)

# Prefix-kode = træer

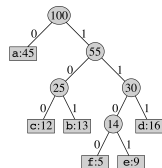
Kodeord = sti i binært træ: 0 ~ gå til venstre, 1 ~ gå til højre

Prefix(-fri) kode: ingen kode for et tegn er starten (prefix) af koden for et andet tegn ( $\Rightarrow$  dekodning utvetydig). Så tegn svarer til knuder med nul børn (blade).

	a	b	c	d	e	f
Frequency (in thousands)	45	13	12	16	9	5
Fixed-length codeword	000	001	010	011	100	101
Variable-length codeword	0	101	100	111	1101	1100



(a)



(b)

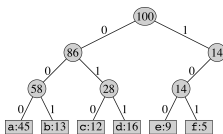
For en givet fil (tegn og deres frekvenser), find bedste variable-width prefix-kode. Dvs. for  $\text{Cost}(\text{tree}) = |\text{kodet fil}|$ , find træ med lavest cost.

# Prefix-kode = træer

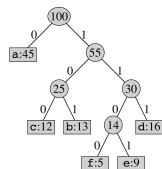
Kodeord = sti i binært træ: 0 ~ gå til venstre, 1 ~ gå til højre

Prefix(-fri) kode: ingen kode for et tegn er starten (prefix) af koden for et andet tegn ( $\Rightarrow$  dekodning utvetydig). Så tegn svarer til knuder med nul børn (blade).

	a	b	c	d	e	f
Frequency (in thousands)	45	13	12	16	9	5
Fixed-length codeword	000	001	010	011	100	101
Variable-length codeword	0	101	100	111	1101	1100



(a)



(b)

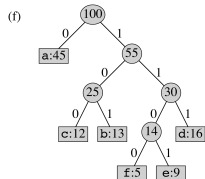
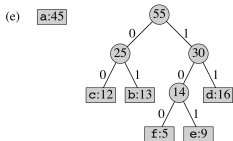
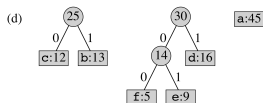
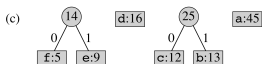
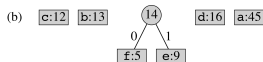
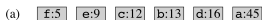
For en givet fil (tegn og deres frekvenser), find bedste variable-width prefix-kode. Dvs.  $\text{for Cost}(\text{tree}) = |\text{kodet fil}|$ , find træ med lavest cost.

Optimale træer kan ikke have knuder med kun ét barn (alle tegn i undertræet for en sådan knude kan forkortes med en bit, jvf. (a) ovenfor). Så kun knuder med to eller nul børn findes.

# Huffmans algoritme

David Huffman, 1952

Byg op nedefra (fra mindste til største frekvenser) ved hele tiden at lave flg. "grådige valg": slå de to deltræer med de to mindste samlede frekvenser sammen:



# Køretid

Givet en tabel med  $n$  tegn og deres frekvenser laver Huffmans algoritme

- ▶  $n - 1$  iterationer.

(Der er  $n$  træer til start, ét træ til slut, og hver iteration mindsker antallet med præcis én.)



# Køretid

Givet en tabel med  $n$  tegn og deres frekvenser laver Huffmans algoritme

- ▶  $n - 1$  iterationer.

(Der er  $n$  træer til start, ét træ til slut, og hver iteration mindsker antallet med præcis én.)

Ved at bruge en (min-)prioritetskø, f.eks. implementeret ved en heap, kan hver iteration udføres med:

- ▶ to ExtractMin-operationer
- ▶ én Insert-operation
- ▶  $O(1)$  andet arbejde.

# Køretid

Givet en tabel med  $n$  tegn og deres frekvenser laver Huffmans algoritme

- ▶  $n - 1$  iterationer.

(Der er  $n$  træer til start, ét træ til slut, og hver iteration mindsker antallet med præcis én.)

Ved at bruge en (min-)prioritetskø, f.eks. implementeret ved en heap, kan hver iteration udføres med:

- ▶ to ExtractMin-operationer
- ▶ én Insert-operation
- ▶  $O(1)$  andet arbejde.

Hver prioritetskø-operation tager hver  $O(\log n)$  tid.

# Køretid

Givet en tabel med  $n$  tegn og deres frekvenser laver Huffmans algoritme

- ▶  $n - 1$  iterationer.

(Der er  $n$  træer til start, ét træ til slut, og hver iteration mindsker antallet med præcis én.)

Ved at bruge en (min-)prioritetskø, f.eks. implementeret ved en heap, kan hver iteration udføres med:

- ▶ to ExtractMin-operationer
- ▶ én Insert-operation
- ▶  $O(1)$  andet arbejde.

Hver prioritetskø-operation tager hver  $O(\log n)$  tid.

Så samlet køretid for de  $n$  iterationer er  $O(n \log n)$ .

# Korrekthed

Resumé:

Huffmans algoritme vedligeholder en samling træer  $F$ . Vægten af et træ er summen af hyppigheden i dets blade. I hvert skridt slår Huffman to træer med mindste vægte sammen, indtil der kun er ét træ.

Vi vil bevise følgende invariant:

Træerne i  $F$  kan slås sammen til et optimalt træ.

Når algoritmen stopper, indeholder  $F$  kun ét træ, som derfor ifølge invarianten må være et optimalt træ.

# Korrekthed

Vi vil bevise følgende invariant:

Træerne i  $F$  kan slås sammen til et optimalt træ.

Beviset er via induktion over antal skridt i algoritmen.

Basis: Ingen skridt er taget. Da består  $F$  af  $n$  træer, som hvert kun er et blad. Disse er netop bladene i ethvert optimalt træ, så invarianten er oplagt opfyldt.

Induktionsskridt: antag invarianten er opfyldt før et skridt i algoritmen, og lad os vise at den er opfyldt efter skridtet.

# Korrekthed

Lad træerne i  $F$  (før skridtet) være

$$t_1, t_2, t_3, \dots, t_k.$$

hvor algoritmen slår  $t_1$  og  $t_2$  sammen. Dvs. at med hensyn til vægt gælder følgende for træerne:

$$t_1 \leq t_2 \leq t_3 \leq \dots \leq t_k,$$

Ifølge induktionsantagelsen kan træerne slås sammen til et optimalt træ. Lad  $T$  være toppen af dette træ—dvs. et træ, hvis blade er rødderne i  $t_1, t_2, t_3, \dots, t_k$ .

# Korrekthed

Hvis rødderne i  $t_1$  og  $t_2$  er søskende i  $T$ , er det klart at invarianten igen gælder efter næste skridt i Huffman.

Da det samlede træ er det samme, er det stadig et optimalt træ.

# Korrekthed

Hvis rødderne i  $t_1$  og  $t_2$  *ikke* er søskende i  $T$ , finder vi et andet top-træ  $T'$  (dvs. en anden sammenlægning af træerne i  $F$ ) som også giver et optimalt træ, og hvor  $t_1$  og  $t_2$  er søskende:

Se på et blad i  $T$  af størst dybde. Da  $k \geq 2$ , har bladet en forælder. Denne forælder har mindst ét undertræ, altså har den to (i optimale træer har ingen knuder ét undertræ, som bemærket tidligere). Dens andet undertræ må være et blad, ellers er det første blad ikke et dybeste blad.

Altså findes to blade i  $T$  som er søskende og begge er af størst dybde. Lad disse indeholde rødderne af  $t_i$  og  $t_j$ , med  $i < j$ .



# Korrekthed

Mulige situationer:

1	2	3...
$i$	$j$	
$i$		$j$
	$i$	$j$
		$i, j$

*Handling som giver  $T'$  fra  $T$ :*

Ingen

Byt  $t_2$  og  $t_j$

Byt  $t_1$  og  $t_j$

Byt  $t_1$  og  $t_i$ , samt  $t_2$  og  $t_j$

# Korrekthed

Mulige situationer:

1	2	3...
$i$	$j$	
$i$		$j$
	$i$	$j$
		$i, j$

Handling som giver  $T'$  fra  $T$ :

Ingen

Byt  $t_2$  og  $t_j$

Byt  $t_1$  og  $t_j$

Byt  $t_1$  og  $t_i$ , samt  $t_2$  og  $t_j$

For et byt af  $t_1$  og  $t_i$  gælder, at eftersom  $t_i$ 's rod mindst har samme dybde som  $t_1$ 's rod, og den samlede frekvens af  $t_i$  er mindst lige så stor som den samlede frekvens af  $t_1$ , vil der være flere tegn i filen, som kan få kortere kodeord, end der er tegn som kan få længere kodeord. Desuden er forandringerne i længde af kodeord den samme for både forlængelse og forkortelse (nemlig forskellen i dybde mellem  $t_1$  og  $t_i$ ).

# Korrekthed

Mulige situationer:

1	2	3...
$i$	$j$	
$i$		$j$
	$i$	$j$
		$i, j$

Handling som giver  $T'$  fra  $T$ :

Ingen

Byt  $t_2$  og  $t_j$

Byt  $t_1$  og  $t_j$

Byt  $t_1$  og  $t_i$ , samt  $t_2$  og  $t_j$

For et byt af  $t_1$  og  $t_i$  gælder, at eftersom  $t_i$ 's rod mindst har samme dybde som  $t_1$ 's rod, og den samlede frekvens af  $t_i$  er mindst lige så stor som den samlede frekvens af  $t_1$ , vil der være flere tegn i filen, som kan få kortere kodeord, end der er tegn som kan få længere kodeord. Desuden er forandringerne i længde af kodeord den samme for både forlængelse og forkortelse (nemlig forskellen i dybde mellem  $t_1$  og  $t_i$ ).

Så den kodede fils længde stiger ikke ved byt af  $t_1$  og  $t_i$ , dvs. træet kan ikke blive dårlige ved byt af  $t_1$  og  $t_i$ .

# Korrektthed

Mulige situationer:

1	2	3...
$i$	$j$	
$i$		$j$
	$i$	$j$
		$i, j$

Handling som giver  $T'$  fra  $T$ :

Ingen

Byt  $t_2$  og  $t_j$

Byt  $t_1$  og  $t_j$

Byt  $t_1$  og  $t_i$ , samt  $t_2$  og  $t_j$

For et byt af  $t_1$  og  $t_i$  gælder, at eftersom  $t_i$ 's rod mindst har samme dybde som  $t_1$ 's rod, og den samlede frekvens af  $t_i$  er mindst lige så stor som den samlede frekvens af  $t_1$ , vil der være flere tegn i filen, som kan få kortere kodeord, end der er tegn som kan få længere kodeord. Desuden er forandringerne i længde af kodeord den samme for både forlængelse og forkortelse (nemlig forskellen i dybde mellem  $t_1$  og  $t_i$ ).

Så den kodede fils længde stiger ikke ved byt af  $t_1$  og  $t_i$ , dvs. træet kan ikke blive dårlige ved byt af  $t_1$  og  $t_i$ .

Tilsvarende kan vises for et byt af  $t_1$  og  $t_j$ , og for et byt af  $t_2$  og  $t_j$ .

## Korrektthed

Mulige situationer:

1	2	3...
$i$	$j$	
$i$		$j$
	$i$	$j$
		$i, j$

Handling som giver  $T'$  fra  $T$ :

Ingen

Byt  $t_2$  og  $t_j$

Byt  $t_1$  og  $t_j$

Byt  $t_1$  og  $t_i$ , samt  $t_2$  og  $t_j$

For et byt af  $t_1$  og  $t_i$  gælder, at eftersom  $t_i$ 's rod mindst har samme dybde som  $t_1$ 's rod, og den samlede frekvens af  $t_i$  er mindst lige så stor som den samlede frekvens af  $t_1$ , vil der være flere tegn i filen, som kan få kortere kodeord, end der er tegn som kan få længere kodeord. Desuden er forandringerne i længde af kodeord den samme for både forlængelse og forkortelse (nemlig forskellen i dybde mellem  $t_1$  og  $t_i$ ).

Så den kodede fils længde stiger ikke ved byt af  $t_1$  og  $t_i$ , dvs. træet kan ikke blive dårlige ved byt af  $t_1$  og  $t_i$ .

Tilsvarende kan vises for et byt af  $t_1$  og  $t_j$ , og for et byt af  $t_2$  og  $t_j$ .

Da træet var optimalt før byt, er det også efter. Og  $t_1$  og  $t_2$  er nu søskende. □