Grafalgoritmer og Algoritmisk Problemløsningsteknik

Erik Meineche Schmidt

Indhold

Introduktion **1**

1 Basale grafteoretiske begreber **2**

2 Repræsentation af grafer **4**

3 Ikke-orienterede grafer **6**

3.1 Grafgennemløb 6

3.2 Anvendelse af grafgennemløb 12

3.3 Letteste udspændende træ 15

3.3.1 Transitionssystem for letteste udspændende træ . . 17

3.3.2 Realisationer af transitionssystemet 20

4 Orienterede grafer **29**

4.1 Gennemløb 29

4.1.1 Træer og rekursive gennemløb 30

4.1.2 Rekursivt dybde-først gennemløb 32

4.2 Sammenhængsegenskaber 33

4.2.1 Acykliske grafer 35

4.3 Korteste veje 38

4.3.1 Enkelt-kilde korteste veje 38

4.3.2 Alle korteste veje 41

5 Del-og-kombiner **45**

5.1 Skabelonen 46
5.2 Hanois Tårne 48
5.3 Quicksort 49
5.4 Flettesortering 53
5.5 Selektion 57
5.6 Heltalsmultiplikation 59
5.7 Sammenfatning 68

6 Dynamisk Programmering 72

6.1 Knapsack 72
6.2 Skabelonen 77
6.3 Binomialkoefficienter 79

7 Kombinatorisk søgning 84

7.1 0-1 Knapsack 84
7.2 Skabelonen 92
7.3 Dronningeproblemet 94
7.4 Rekursive løsninger 97

8 Sammenfatning 101

9 Referencer 102

Indledning

Denne bog består af to halvdele, omhandlende hhv. grafalgoritmer og algoritmisk problemløsningsteknik.

Grafalgoritmernes berettigelse ligger først og fremmest i, at grafer har en meget bred anvendelsesflade inden for beskrivelser af kombinatoriske problemer, hvorfor det er vigtigt at kende effektive algoritmer til løsning af de oftest forekommende grafproblemer. Grafalgoritmer er også gode eksempler på systematisk anvendelse af ikke-trivielle datastrukturer.

Problemløsningsteknikkernes berettigelse ligger i, at de er bredt anvendelige i forbindelse med løsning af en lang række forskellige algoritmiske problemer. Vi understreger dette ved at karakterisere tre af de vigtigste teknikker i v.hj.a. *algoritmeskabeloner*, dvs. abstrakte standardalgoritmer, der omfatter alle de konkrete algoritmer af de pågældende typer.

Kapitel 1 og 2 introducerer grafer, deres repræsentation og den mest nødvendige terminologi.

Kapitel 3 behandler ikke-orienterede grafer med hovedvægt på algoritmer til grafgennemløb og udspændende træer.

Kapitel 4 gentager historien fra kapitel 3 for orienterede grafer, dog udvidet med behandling af acykliske grafer og med korteste veje i stedet for udspændende træer.

I kapitel 5 introduceres den første problemløsningsteknik, del-og-kombiner, som er den vigtigste og den, der har de bredeste anvendelsesområder. Den belyses bl.a. ved hjælp af tre centrale algoritmiske problemer, hvor den fører til effektive løsninger.

De to øvrige teknikker, dynamisk programmering og kombinatorisk søgning, som præsenteres i kapitel 6 og 7 kan opfattes som en slags 1. og 2. reserver, der evt. kan bringes i anvendelse, når del-og-kombiner teknikken svigter.

1 Basale grafteoretiske begreber

En graf er et par $G = (V, E)$ bestående af *knuder* V og *kanter* E. Vi skal for simpelheds skyld antage, at knudemængden er

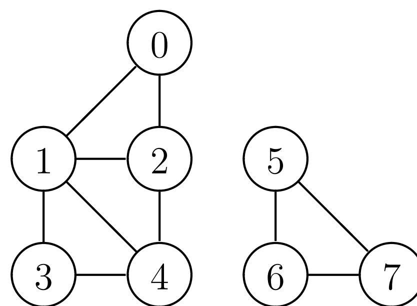
$$V = \{0, 1, \dots, n-1\}$$

Hvis G er en *orienteret* graf, består E af en mængde af *ordnede par* af formen $[v, w]$, hvor $v, w \in V$. Hvis G er en *ikke-orienteret* graf, består E af en mængde af *uordnede par* $\{v, w\}$ hvor $v, w \in V$. Vi skal i begge tilfælde altid antage, at $v \neq w$.

Når vi tegner grafer, vil kanterne i de orienterede grafer være pile, medens kanterne i de ikke-orienterede grafer blot er "streger". Knuderne tegnes i begge tilfælde som cirkler.

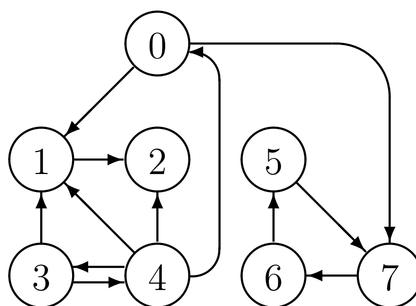
Følgende tegning repræsenterer således den ikke-orienterede graf

$$G_{io} = (\{0, 1, \dots, 7\}, \quad \{\{0,1\}, \{0,2\}, \{1,2\}, \{1,3\}, \{1,4\}, \{2,4\}, \{3,4\}, \{5,6\}, \{5,7\}, \{6,7\}\})$$



medens følgende viser den orienterede graf

$$G_o = (\{0, 1, \dots, 7\}, \quad \{[0,1], [0,7], [1,2], [3,1], [3,4], [4,0], [4,1], [4,2], [4,3], [5,7], [6,5], [7,6]\})$$



Da det som regel vil fremgå af sammenhængen, om der er tale om en orienteret eller en ikke-orienteret graf, skal vi bruge fællesbetegnelsen (u, v) for $[u, v]$ eller $\{u, v\}$ og lade situationen afgøre, hvilken der er tale om.

Hvis (u, v) er en kant, siges u og v at være *incidente* med (u, v). (u, v) er også incident med u og v.

Hvis $\{u, v\}$ er kant i en ikke-orienteret graf, er u og v naboer. En orienteret kant $[u, v]$ *går fra* u *til* v.

Hvis v er en knude i en ikke-orienteret graf, er v's *grad* antallet af naboknuder. Hvis v er knude i en orienteret graf, er *indgraden* (*udgraden*) antallet af kanter $[u, v]$ ($[v, u]$).

En *vej* fra v_0 til v_k er en liste af knuder $(v_0, v_1, \ldots, v_k)$, hvor $k \geq 0$, og hvor der for alle $i \in 0..k$ gælder at (v_i, v_{i+1}) er en kant. En vej er *simpel*, hvis alle knuderne er forskellige.

I en orienteret graf kaldes en vej for en *cykel*, hvis $k > 0$ og $v_0 = v_k$; vejen er en *simpel* cykel, hvis der yderligere gælder at $v_0, v_1, \ldots, v_{k-1}$ er indbyrdes forskellige.

I en ikke-orienteret graf er en vej en cykel, hvis $k > 0$, $v_0 = v_k$, og ingen kant gentages; vejen er igen en simpel cykel, hvis der yderligere gælder, at $v_0, \ldots, v_{k-1}$ alle er indbyrdes forskellige.

Hvis $G_1 = (V_1, E_1)$ og $G_2 = (V_2, E_2)$ er grafer, er G_1 en *delgraf* af G_2, såfremt $V_1 \subseteq V_2$ og $E_1 \subseteq E_2$. G_1 er en *udspændende* delgraf af G_2, hvis der yderligere gælder $V_1 = V_2$. G_1 er den delgraf, der *induceres* af V_1, hvis E_1 består af alle kanter $(u, v) \in E_2$ for hvilke $u \in V_1$ og $v \in V_1$.

2 Repræsentation af grafer

Når grafer skal manipuleres af en algoritme, skal de naturligvis repræsenteres på en eller anden måde. Dette kan i alt væsentligt gøres på to måder, nemlig v.hj.a. *kantlisterepræsentation* eller som *incidensmatrix*.

I en kantlisterepræsentation angives for hver knude, hvilke knuder den har som naboer. Den relevante TRINE type er

Type Graf = **List** (Vector)

og udseendet vil for de to eksempelgrafer ovenfor være som følger

$G_{io} =$ ((1,2),
(3,4,2,0),
(0,1,4),
(4,1),
(2,1,3),
(6,7),
(7,5),
(5,6),
)

$G_o =$ ((1,7),
(2),
( ),
(4,1),
(2,1,3),
(7),
(5),
(6),
)

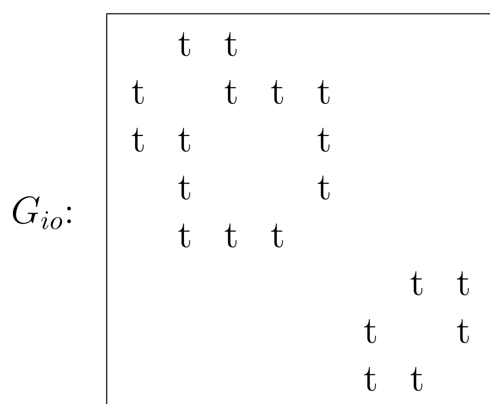
hvor altså $G_{io}.(v)$ ($G_o.(v)$) er en liste bestående af de knuder w, for hvilke $\{v, w\}$ ($[v, w]$) er en kant. $G_{io}.(v)$ og $G_o.(v)$ kan naturligvis også være kædede (pointer) lister.

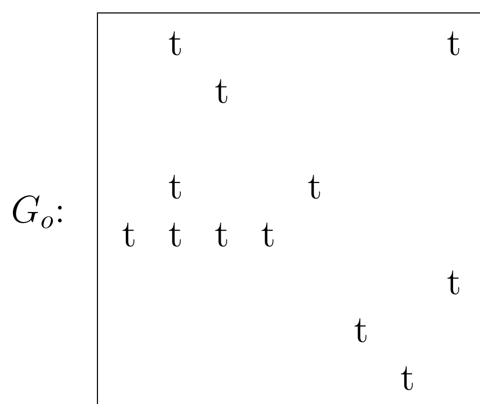
En *incidensmatrix* er en $n \times n$ Boolsk matrix, hvor den (v, w)'te indgang

er **true**, hvis $\{v, w\}$ eller $[v, w]$ er en kant og **false** ellers. Den relevante TRINE type er her

Type Graf = **List** (Bits)
Type Bits = **List** (Bool)

og de to eksempler ser ud som følger (vi bruger **t** for **true** og undlader at angive **false**-værdier).





Bemærk, at da kanterne i ikke-orienterede grafer er uordnede par, vil incidensmatricen for en sådan graf altid være symmetrisk.

Det skulle være klart, at kantlisterepræsentationen er den mest økonomiske m.h.t. pladskrav, idet den fylder $O(m)$, hvor m er antallet af kanter i grafen. Incidensmatricen fylder altid $O(n^2)$ uanset antallet af kanter – til gengæld kan man afgøre om et par (v, w) er en kant i tid $O(1)$.

I det følgende skal vi altid antage – med mindre det modsatte eksplicit nævnes – at grafer repræsenteres v.hj.a. kantlister, og vi skal også systematisk betegne antal knuder med n og antal kanter med m.

3 Ikke-orienterede grafer

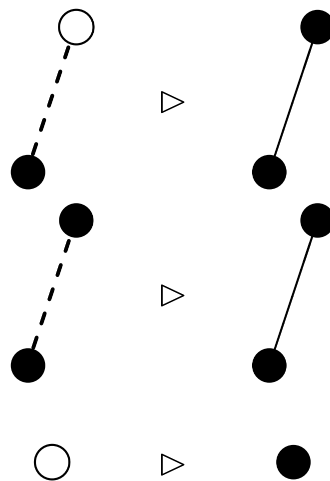
I dette kapitel betragtes algoritmer, der behandler følgende problemstillinger for ikke-orienterede grafer: gennemløb, sammenhæng, udspændende træer.

3.1 Grafgennemløb

En af de simpleste problemstillinger er at konstruere en effektiv algoritme til gennemløb af en graf, hvor der med gennemløb menes, at samtlige kanter og knuder har været "besøgt" mindst én gang.

Algoritmen tager udgangspunkt i det transitionssystem for grafgennemløb, der blev introduceret i afsnit 5.4 i [Transitionssystemer]. Systemet ser ud som følger.

Transitionssystem: Graffarvning
Konfigurationer: Danske grafer
Transitioner :



hvis der ikke findes lyserøde knuder.

hvor en lyserød knude er en rød knude (●) med mindste én hvid kant ( ).

Systemets egenskaber er karakteriseret i sætning 5.6 i [Transitionssystemer].

Vi ønsker nu at skrive en algoritme, der kan realisere dette transitionssystem, og vi ønsker at gøre det på en sådan måde, at en graf gennemløbes

under anvendelse af højst $O(n+m)$ primitive operationer, hvor en primitiv operation svarer til valg af en transitionsregel.

Algoritmen konstrueres omkring en datastruktur, der indeholder de knuder, der har mulighed for at være lyserøde. Ideen er den simple, at udførelsen af en af de to første regler i transitionssystemet modsvares af at udvælge en knude fra datastrukturen, farve alle udgående hvide kanter røde og indsætte de tilhørende naboer i datastrukturen, hvis de da ikke allerede er der. I følgende algoritme er intuitionen, at H betegner de hvide knuder og PLR de **p**otentielt **l**yse**r**øde knuder.

Algoritme: Skitse af Graffarvning
Stimulans: $G = (V, E)$ ikke orienteret "hvid" graf
Respons : G, hvor alle knuder og kanter er farvet røde
Metode : $PLR, H := \emptyset, V$
  do $\{I\}$
    $(H \neq \emptyset) \vee (PLR \neq \emptyset) \rightarrow$
      if $PLR = \emptyset \rightarrow \ll$ vælg og fjern v fra $H \gg$
        $\ll$ farv v rød $\gg$
        $\ll$ indsæt v i $PLR \gg$
      $|PLR \neq \emptyset \rightarrow$ $\ll$ vælg og fjern v fra $PLR \gg$
        for $w : (v, w) \in E$ **do**
          if $w \in PLR \rightarrow \ll$ farv (v, w) rød $\gg$
          $|w \in H \rightarrow \ll$ fjern w fra $H \gg$
            $\ll$ farv (v, w) og w rød $\gg$
            $\ll$ indsæt w i $PLR \gg$
          & true $\rightarrow$ **skip** $\{(v, w)$ er rød$\}$
          fi
        od
      fi
  od

Algoritmens opførsel karakteriseres af følgende invariant:

I: (alle knuder i Her hvide) og
(alle knuder i $V - H$ er røde) og
(alle lyserøde knuder tilhører PLR) og
(alle kanter der er incidente med to knuder i PLR er hvide)

At algoritmen er korrekt, vises ved som sædvanligt at bevise, at invarianten er gyldig, at beregningen terminerer og at invarianten implicerer responskravet, når beregningen er standset. Vi overlader det til læseren at vise, at invarianten er gyldig, og nøjes med at anføre resten af korrekthedsbeviset.

Som termineringsfunktion bruger vi to gange antallet af hvide knuder plus antallet af knuder i PLR, dvs.

$$\mu = 2|H| + |PLR|$$

At alle kanter og knuder i grafen er farvede, når beregningen standser, følger af, at den kontrollerende betingelse er falsk, dvs. at $H = PLR = \emptyset$. Dette betyder, at alle knuder er røde ($H = \emptyset$), men så er alle kanter også røde, for ellers ville der være en lyserød knude, hvilket er umuligt da $PLR = \emptyset$.

Når denne algoritme skal implementeres, får vi brug for datastrukturer til H og PLR, der understøtter følgende operationer.

H:	Init $[H](n)$	**sat**	$H' = \{0, 1, \ldots, n-1\}$
	Empty $[H]$	**sat**	$(\text{Empty}' = (H = \emptyset)) \wedge (H' = H)$
	Member $[H](v)$	**sat**	$(\text{Member}' = v \in H) \wedge (H' = H)$
	Delete $[H](w)$	**sat**	$H' = H - \{w\}$
	DeleteSome $[H, v]$	**sat**	$H \neq \emptyset \rightarrow (v' \in H) \wedge (H' = H - \{v'\})$
PLR:	Init $[PLR](n)$	**sat**	$PLR' = \emptyset$
	Empty $[PLR]$	**sat**	som ovenfor
	Member $[PLR](v)$	**sat**	som ovenfor
	Insert $[PLR](v)$	**sat**	$PLR' = PLR \cup \{v\}$
	DeleteSome $[PLR, v]$	**sat**	som ovenfor.

En datastruktur som H kalder vi en *monoton mængde* (fordi der aldrig tilføjes elementer til den), og en struktur som PLR betegner vi en *nondeterministisk mængde*, fordi vi kun kan fjerne elementer nondeterministisk fra den.

Hvis H (PLR) er en monoton (nondeterministisk) mængde, der realiseres v.hj.a. en box MS (NS), kan vi skrive følgende algoritme.

Algoritme: Graffarvning

```
Stimulans:  som ovenfor
Respons  :  som ovenfor
Metode   :  NS'Init[PLR](n)
            MS'Init[H](n)
            do ¬ (MS'Empty[H] ∧ NS'Empty[PLR]) →
                if NS'Empty[PLR] →
                    MS'DeleteSome[H, v]
                    ≪farv v rød≫
                    NS'Insert[PLR](v)
                & true →
                    NS'DeleteSome[PLR, v]
                    for (v, w) in E do
                        if NS'Member[PLR](w) →
                            ≪farv (v, w) rød≫
                         | MS'Member[H](w) →
                            MS'Delete[H](w)
                            ≪farv (v, w) og w rød≫
                            NS'Insert[PLR](w)
                        & true → skip
                        fi
                       od
                fi
            od
```

Det skulle være nemt at se, at vi har konstrueret en algoritme med den ønskede egenskab, at antallet af kald til datastrukturens operationer er proportionalt med $O(n + m)$.

Mere præcist har vi følgende udtryk for algoritmens udførelsestid

$$
\begin{aligned}
&T[\![\text{Graffarvning}]\!](n,m) = \\
&\quad T[\![NS' \text{ Init } [PLR](n)]\!] + T[\![MS' \text{ Init } [H](n)]\!] + \\
&\quad n * (\, T[\![NS' \text{ Empty } [PLR]]\!] + T[\![MS' \text{ Empty } [H]]\!] + \\
&\qquad\quad T[\![NS' \text{ Insert } [PLR](v)]\!] + \\
&\qquad\quad T[\![NS' \text{ DeleteSome } [PLR, v]]\!] + \\
&\qquad\quad T[\![MS' \text{ Delete } [H](w)]\!]) + \\
&\quad m * (T[\![NS' \text{ Member } [PLR](w)]\!] + T[\![MS' \text{ Member } [H](w)]\!]) + \\
&\quad T[\![\text{samtlige kald af formen } MS' \text{ DeleteSome } [H, v]]\!]
\end{aligned}
$$

Hvis vi kan implementere monotone og nondeterministiske mængder, så alle operationerne har udførelsestid $O(1)$, har vi opnået en optimal algoritme. Dette kan (næsten) gøres på følgende måde.

Den nondeterministiske mængde repræsenteres på *to* måder, dels som en bitvektor og dels som en liste. Bitvektoren tjener til at gøre Insert og Member effektive, medens listen tager sig af DeleteSome.

Den monotone mængde repræsenteres som et par bestående af en bitvektor og en tæller, hvor bitvektoren gør Delete og Member effektive, medens tælleren gør Empty effektiv. DeleteSome realiseres ved lineær søgning efter "det første element" i bitvektoren; hertil anvendes pegepinden start, som angiver hvor den sidste søgning sluttede. Dette gør ikke nødvendigvis det enkelte kald af DeleteSome effektivt, men i amortiseret forstand er sagen i orden, fordi de samlede omkostninger ved alle kald af DeleteSome tilhører $O(n)$. Ideen er den samme som for boxen PS i programmet Erathostenes i [P&D].

De to boxe ser ud som følger.

```
Box MS
    Type Mset = Prod(size: Int, cont: Bits, start: Int)
    Type Bits = List(Bool)

    Proc Init[ms: Mset] (n: Int)
      ms := Mset(n, Bits(true|n), 0)
    end Init

    Proc Empty[ms: Mset] → (Bool)
      return ms.size = 0
    end Empty

    Proc DeleteSome[ms: Mset, i: Int]
      i := ms.start
      do ¬ ms.cont.(i) → i := i+1 od
      ms.start, ms.cont.(i) := i, false
      ms.size := ms.size−1
    end DeleteSome

    Proc Delete[ms: Mset] (i: Int)
      if ms.cont.(i) →
        ms.cont.(i) := false
        ms.size := ms.size−1
      fi
    end Delete

    Proc Member[ms: Mset] (i: Int) → (Bool)
      return ms.cont.(i)
    end Member
  end MS

  Box NS
    Type Nset = Prod(size: Int, Bcont: Bits, Lcont: Vector)
    Type Bits = List(Bool)

    Proc Init[ns: Nset] (n: Int)
      ns := Nset(0, Bits(false|n), Vector(0|n))
    end Init

    Proc Empty[ns: Nset] → (Bool)
      return ns.size = 0
    end Empty

    Proc DeleteSome[ns: Nset, i: Int]
      i := ns.Lcont.(ns.size−1)
      ns.size := ns.size−1
      ns.Bcont.(i) := false
    end DeleteSome

    Proc Insert[ns: Nset] (i: Int)
      if ¬ ns.Bcont.(i) →
            ns.Lcont.(ns.size) := i
            ns.size := ns.size+1
            ns.Bcont.(i) := true
      fi
    end Insert

    Proc Member[ns: Nset] (i: Int) → (Bool)
      return ns.Bcont.(i)
    end Member
  end NS
```

3.2 Anvendelse af grafgennemløb

I dette afsnit betragtes følgende tre anvendelser af algoritmen Graffarvning:

- sammenhængskomponenter
- dybde-først og bredde-først nummerering
- udspændende træer

Sammenhængskomponenter

En ikke-orienteret grafs sammenhængskomponenter er en opdeling af grafens knuder i klasser, hvor hver klasse består af samtlige knuder, der parvist er forbundet af en vej i grafen. I eksempelgrafen G_{io} side 2 er der således to sammenhængskomponenter bestående af hhv. knuderne $\{0, 1, 2, 3, 4\}$ og $\{5, 6, 7\}$. En mere præcis definition af sammenhængskomponenter er som følger. Hvis vi definerer en relation F (for Forbundet) ved, at v står i relationen F til w, såfremt der findes en vej fra v til w, så er det nemt at se, at F er en ækvivalensrelation. Sammenhængskomponenterne er nu simpelthen ækvivalensklasserne for F.

En algoritme, der finder sammenhængskomponenterne, kunne virke på følgende måde: Den tildeler, startende med 1, alle knuder i en sammenhængskomponent det samme nummer. I G_{io} vil knuderne $\{0, 1, 2, 3, 4\}$ således få nummer 1 og knuderne $\{5, 6, 7\}$ vil få nummer 2.

Hvis vi betragter algoritme Graffarvning er det klart, at hver gang PLR er tom, begynder vi på en ny sammenhængskomponent. Men så kan vi finde numrene på sammenhængskomponenterne v.hj.a. følgende konkretisering. CN er en vektor, der indeholder komponentnumrene, og N er en heltalsvariabel, der holder styr på det løbende nummer.

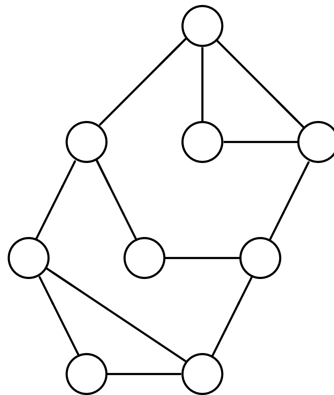
$$
\begin{array}{lll}
\ll \text{farv } v \text{ rød} \gg & \textbf{is} & N := N + 1 \\
 & & CN.(v) := N \\
\ll \text{farv } (u, w) \text{ og } w \text{ rød} \gg & \textbf{is} & CN.(w) := N
\end{array}
$$

Herudover skal følgende initialiseringer tilføjes

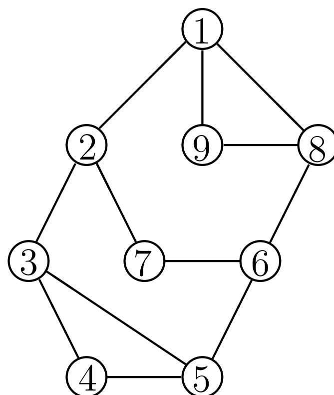
$$CN, N := \text{Vector}(0|n), 0$$

Dybde-først og bredde-først nummerering

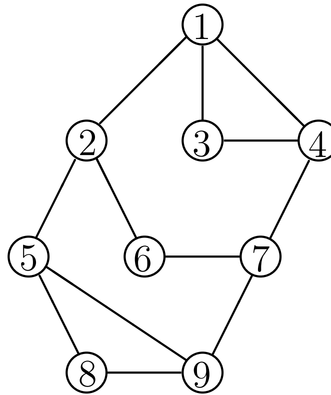
I dette eksempel bruges Graffarvning til at nummerere grafens knuder i overensstemmelse med den rækkefølge i hvilken potentielt lyserøde knuder udvælges fra PLR, dvs. i overensstemmelse med hvordan DeleteSome realiseres. Der er to hovedmetoder hertil: man kan vælge den sidst indsatte knude eller man kan vælge den først indsatte knude. I det ene tilfælde fungerer PLR som en *stak* og i det andet som en *kø*, jfr. [P&D]. Når det er stakdisciplinen der anvendes taler man om *dybde-først* nummerering, og når det er kødisciplinen om *bredde-først* nummerering. Navnene stammer fra at man forsøger hhv. at komme så dybt og bredt i grafen som muligt. I følgende graf



giver en dybde-først nummerering således



medens en bredde-først nummerering resulterer i



Selve realiseringen af Graffarvning går som følger (vi betragter kun dybde-først og overlader bredde-først til læseren).

Initialiseringen udvides med

$$DFN, N := \text{Vector }(0|n), 1$$

og da nummereringen som sagt skal foregå, når knuderne *fjernes* fra PLR, forbliver farvningerne uændret, medens der efter kaldet
NS' DeleteSome$[PLR, v]$ indsættes sætningen

$$DFN.(v), N := N, N+1$$

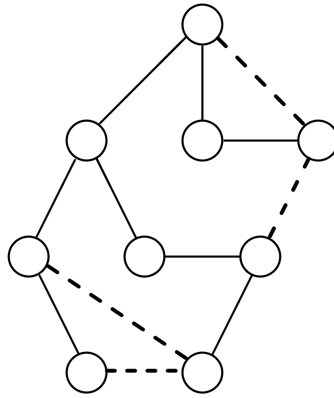
Herudover skal den monotone mængde PLR implementeres, så DeleteSome anvender stakdisciplinen, men det gør implementationen af NS side 11 faktisk allerede.

Udspændende træer

Et træ T kaldes et *udspændende træ* for en graf G, hvis T er en udspændende delgraf for G, dvs. (jfr. side 3) hvis

- T har de samme knuder som G
- T's kanter også er kanter i G

Følgende er et udspændende træ for grafen ovenfor (de af grafens kanter der ikke tilhører T er stiplede).



Det skulle være klart, at enhver sammenhængende graf har et udspændende træ. Hvis grafen ikke er sammenhængende, har den et antal træer, der tilsammen udspænder den – en sådan kaldes en *udspændende skov*.

Det er særdeles nemt at modificere Graffarvning til at finde en udspændende skov. Hvis vi betragter realiseringen

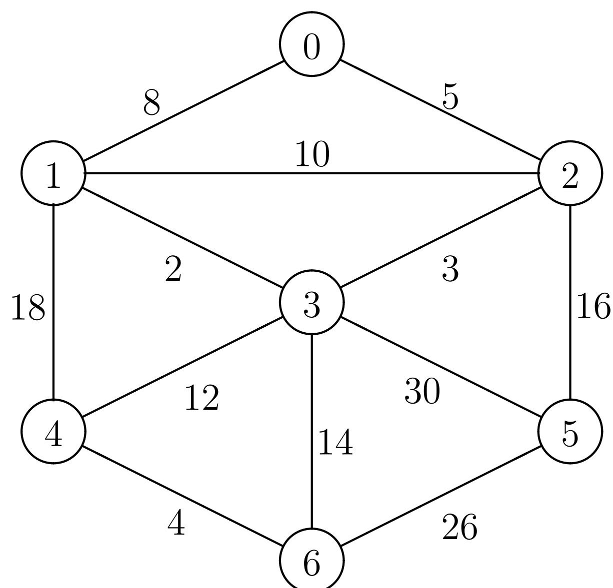
$\ll$ farv (v, w) rød $\gg$ **is skip**

gælder det, at når algoritmen standser, udgør de røde knuder og kanter tilsammen en udspændende skov for grafen.

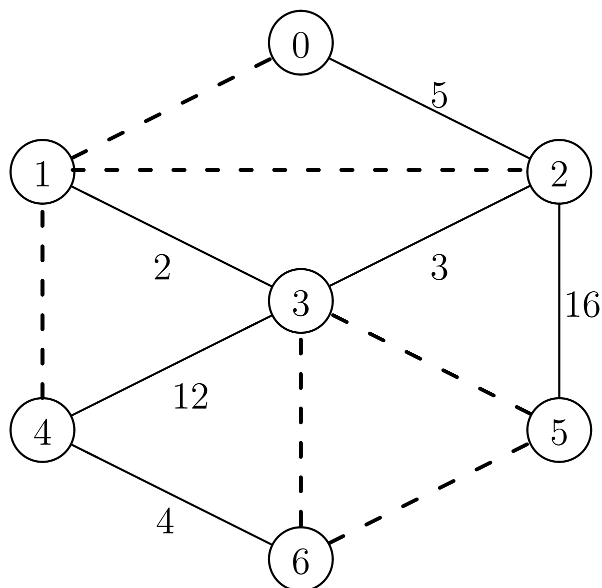
Det er klart, at forskellige valg af gennemløbsstrategier fører til forskellige udspændende træer. Man taler således både om dybde-først og bredde-først træer, som er de udspændende træer, der fremkommer som resultat af at anvende hhv. stak- og kø-disciplinen i DeleteSome. Det er også klart, at uanset hvilken strategi der bruges, så har vi en optimal $O(n+m)$ algoritme til at finde udspændende træer.

3.3 Letteste udspændende træ

I dette afsnit betragtes også udspændende træer, men nu for *vægtede grafer*, dvs. grafer hvor der til hver kant er knyttet en vægt. Følgende er et eksempel på en vægtet graf med heltallige vægte.



Vægtede grafer har adskillige anvendelser, hvoraf en af de mere oplagte er som *afstandskort*. Her er knuderne byer, kanterne er veje, og vægtene er længden af vejene. En af de klassiske problemstillinger for vægtede grafer er at finde *letteste udspændende træer*, som er udspændende træer, hvor summen af kanternes vægte er mindst mulig. Grafen ovenfor har følgende letteste udspændende træ med en kantsum på 42.



Man kan finde sådanne træer på en måde, der minder om hvad vi så i sidste afsnit, blot skal man (naturligvis) vælge træ-kanterne med større omhu. I stedet for at "gå direkte på" en modifikation af Graffarvning til

også at håndtere letteste udspændende træ, skal vi i næste afsnit præsentere et generelt transitionssystem, der kan konkretiseres til flere forskellige algoritmer for letteste udspændende træer.

3.3.1 Transitionssystem for letteste udspændende træ

Til beskrivelse af transitionssystemet får vi brug for endnu et grafteoretisk begreb.

Et *snit* i en graf $G = (V, E)$ er en opdeling af V i to disjunkte mængder, dvs. et par (V_1, V_2), hvor $V_1 \cap V_2 = \emptyset$ og $V_1 \cup V_2 = V$. En kant $\{v, w\}$ siges at *skære* snittet, hvis $v \in V_1$ og $w \in V_2$ (eller omvendt).

Graferne vil i det følgende have kanter der er enten blå, røde eller hvide, hvorfor vi kalder dem *norske grafer*. Et snit i en norsk graf, der skæres af mindst én hvid kant og ikke af nogen blå kant, kaldes et *dansk snit*. En simpel cykel i en norsk graf, der indeholder mindst én hvid kant og ingen røde kanter, kaldes en *finsk cykel*.

Intuitionen bag transitionssystemet er, at da et udspændende træ for en graf skal indeholde mindst en kant, der skærer ethvert snit i grafen, så skal det letteste udspændende træ indeholde den letteste sådanne kant. Da træer ydermere ikke kan indeholde cykler, må det letteste udspændende træ ikke indeholde den tungeste kant på nogen cykel. Dette fører til transitionssystemet, der er konstrueret omkring følgende egenskab, som vi skal vise er en invariant.

> I: Der findes et letteste udspændende træ for grafen som indeholder alle de blå kanter og ingen af de røde.

Transitionssystem: Letteste udspændende træ

Konfigurationer: Norske grafer

Transitioner : Vælg et dansk snit og farv den letteste hvide kant, der skærer snittet, blå.

Vælg en finsk cykel og farv dens tungeste hvide kant i cyklen rød.

Dette transitionssystem har følgende egenskab, som er i overensstemmelse med intuitionen ovenfor:

> Enhver proces for systemet, som starter med en hvid sammenhængende vægtet graf G, er endelig, og slutkonfigurationens blå kanter udgør et letteste udspændende træ for G.

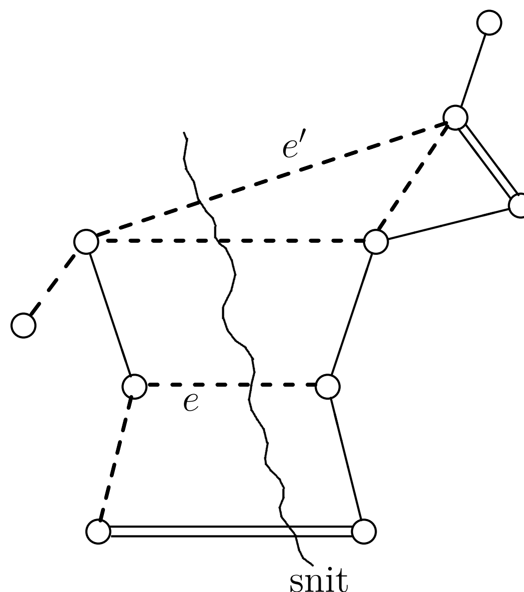
Vi beviser denne påstand på sædvanlig vis, dvs. vi viser, at der findes en hensigtsmæssig termineringsfunktion, at invarianten er gyldig, og at invarianten implicerer, at slutkonfigurationen har de ønskede egenskaber.

a) At alle processer er endelige følger af, at antallet af hvide kanter formindskes med 1 hver gang en transition udføres.

b) At invarianten er gyldig ses på følgende måde (for simpelheds skyld antager vi, at alle kantvægte er parvis forskellige).

   Iflg. invarianten eksisterer der et letteste udspændende træ for G, der indeholder alle de blå kanter og ingen af de røde. Lad os kalde det T.

   Betragt nu en udførelse af den "danske" transitionsregel, som farver en hvid kant blå. Lad os kalde denne kant e. Umiddelbart før udførelsen er situationen som følger (hvide kanter er stiplede, blå kanter er fuldt optrukne og røde kanter er dobbelte).



Vi påstår, at e må være med i T, og at vi altså roligt kan farve den blå. Argumentet er et indirekte bevis, dvs. vi antager at e ikke er med i T og etablerer en modstrid.

Betragt e's endepunkter. Disse er forbundet af en vej i T (ellers ville T ikke være noget udspændende træ), og da de tilhører hver sin del af det valgte snit, må der være en kant e' i T, der også skærer snittet. e' har imidlertid større vægt end e, fordi den er hvid, og e er den letteste hvide kant der skærer dette snit. Vi betragter nu følgende træ T_0. T_0 er lig med T bortset fra at e' er erstattet af e. T_0 er også et udspændende træ, og dets vægt er lavere end T's, fordi e har lavere vægt end e'. Men det er i modstrid med at T er et letteste udspændende træ.

Altså kan vi konkludere, at e tilhører T, hvorfor invarianten også er opfyldt efter at e er farvet blå.

Hvis vi i stedet betragter en udførelse af den "finske" transition, hvor e males rød, skal vi vise, at e ikke kan tilhøre T. Beviset er igen indirekte, dvs. vi antager at e tilhører T og etablerer en modstrid.

Hvis vi fjerner e fra T, falder T i to stykker, hvis knuder repræsenterer et snit i grafen. Den finske cykel vi har valgt, må foruden e indeholde endnu en kant e', der skærer dette snit. Iflg. invarianten er e' hvid, men så er den lettere end e, fordi vi farvede den tungeste hvide kant i cyklen. Hvis vi igen betragter træet T_0, som er lig med T bortset fra at e er udskiftet med e', har vi atter en modstrid.

Altså kan e ikke tilhøre T, hvorfor invarianten også er opfyldt efter at e er farvet rød.

c) At slutkonfigurationen har de ønskede egenskaber følger af invarianten, hvis vi kan vise, at alle grafens kanter er enten blå eller røde. Dette viser vi ved at bevise, at så længe der er hvide kanter tilbage, kan en af transitionerne udføres. Mere præcist beviser vi, at enhver hvid kant i en graf, der tilfredsstiller invarianten enten tilhører et dansk snit eller en finsk cykel. Argumentet er som følger.

Mængden af blå træer (et træ der kun indeholder en enkelt knude betragtes også som blåt) udgør en udspændende skov for grafen. Hvis e er en hvid kant, er der én af følgende muligheder

- e's endepunkter tilhører forskellige blå træer

- e's endepunkter tilhører samme blå træ.

I det første tilfælde skærer e det danske snit, hvis knuder består af hhv. det ene træ og resten af grafen. I det andet tilfælde udgør e, sammen med den vej i det blå træ, der forbinder e's endepunkter, en finsk cykel.

Alt i alt beviser a), b) og c) tilsammen, at transitionssystemet har de ønskede egenskaber.

3.3.2 Realisationer af transitionssystemet

Som omtalt i indledningen til sidste afsnit, findes der flere måder at realisere transitionssystemet Letteste udspændende træ. I dette afsnit betragter vi de to mest kendte metoder som leder til hhv. *Kruskals* og *Prims* algoritmer.

Kruskals algoritme

Metoden minder en del om beviset for, at enhver hvid kant enten tilhører et dansk snit eller en finsk cykel. Mere præcist går algoritmen ud på at lade en udspændende skov "gro op nedefra". Dette sker ved successivt at kombinere to mindre blå træer til et større ved at farve en af de hvide kanter der forbinder dem blå. For at sikre, at man altid betragter den letteste sådanne hvide kant behandles kanterne i voksende rækkefølge efter vægt. Givet en hvid kant, skal man da afgøre, om dens endepunkter ligger i to forskellige blå træer eller om de tilhører det samme blå træ. Hertil kan man passende benytte typen ækvivalensrelation (jfr. [P&D]), idet det skulle være klart, at relationen *tilhører samme blå træ* er en ækvivalensrelation over grafens knuder.

I nedenstående algoritme er P en prioritetskø, der indeholder grafens kanter, og R er en variabel af type ækvivalensrelation. Kanterne er af følgende type

Type Edge = **Prod**(n1,n2,p: Int)

og T er en repræsentation af den letteste udspændende skov. Vi kommer

ikke nærmere ind på denne repræsentation, men opfatter blot T som en graf, dvs. et par bestående af en mængde knuder og en mængde kanter. Den anførte $\oplus$ operation er følgende operation på par af mængder

$$(V_1, E_1) \oplus (V_2, E_2) = (V_1 \cup V_2, E_1 \cup E_2)$$

Algoritme: Kruskal

Stimulans: $G = (V, E)$: vægtet, sammenhængende
ikke-orienteret graf

Respons: T: letteste udspændende træ for G

Metode:

```
≪Indsæt G's kanter i P ≫
≪ T := ({0, 1, ..., n − 1}, ∅) ≫
Init [R](n)
do ¬ Empty [P] →
      DelMin [P, e]
      if ¬ Equiv [R](e.n1, e.n2) →
            (* e forbinder to forskellige blå træer
               og kan farves blå *)
            ≪ T := T ⊕ (∅, {e}) ≫
            Join [R](e.n1, e.n2)
      & true → skip
               (*  e's ender tilhører samme træ.
                   e kan farves rød *)
      fi
od
```

Det er klart, at algoritmen er korrekt. Med hensyn til dens effektivitet får vi følgende

$$\begin{aligned} T[\![\text{Kruskal}]\!](n, m) \approx\; & m * \log(m) + \\ & T[\![\text{Init}[R](n)]\!] + \\ & m * (T[\![\text{Empty}[P]]\!] + \\ & \quad T[\![\text{DelMin}[P, e]]\!] + \\ & \quad T[\![\text{Equiv}[R](e.n1, e.n2)]\!]) + \\ & n * (\, T[\![\text{Join}[R](e.n1, e.n2)]\!] + \\ & \quad T[\![\ll T := T \oplus (\emptyset, \{e\}) \gg]\!]) \end{aligned}$$

Da vi ved, at P og R kan implementeres så samtlige operationer har logaritmiske udførelsestider (og da det er nemt at se, at vi kan antage, at

$T[\![\ll T := T \oplus (\emptyset, \{e\}) \gg]\!] \in O(1))$ giver dette alt i alt

$$T[\![\text{Kruskal}]\!](n, m) \in O(m * (\log(n) + \log(m)) + n * \log(n))$$

hvilket, i det normale tilfælde, hvor $m \geq n$, reduceres til

$$T[\![\text{Kruskal}]\!](m) \in O(m \log(m)).$$

Følgende program viser en implementation af Kruskal's algoritme i TRINE. De relevante Boxe findes i [TRINE] og [P&D]. Da vi aldrig ændrer på T's knudemængde, kan vi nøjes med at interessere os for kanterne i T, hvorfor T kan implementeres som en følge af kanter.

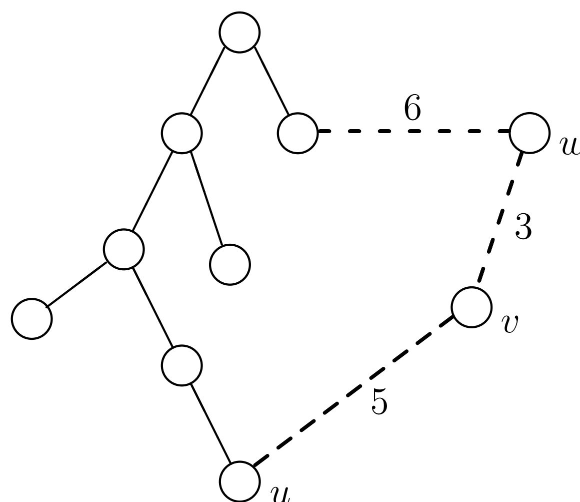
```
Process Kruskal
    (+ @"sequence.tri"
       @"polypri.tri"
       @"equiv.tri"

       Type Edge = Prod(n1, n2, p: Int)

       Box ES
           Sequence(Edge)
       end ES

       Box EP
           PolyPri(Edge)
       end EP

       Var T: ES'Seq
       Var P: EP'Queue
       Var R: Eq'Rel
       Var e: Edge
       Var n, m: Int

       write("Antal knuder: ")
       read[n]
       Eq'Init[R](n)
       ES'Init[T]
       write("Antal kanter: ")
       read[m]
       EP'Init[P](m, true)
       ≪ Indlæs kanter ≫
       do ¬ EP'Empty[P] →
             (+ Var p: Int
                EP'DeleteBest[P, e, p]
             +)
             if ¬ Eq'Equiv[R](e.n1, e.n2) →
                   ES'Rpush[T](e)
                   Eq'Join[R](e.n1, e.n2)
             fi
       od
       write("Træet er: ")
       ES'Print[T]
    +)
end Kruskal
where ≪ Indlæs kanter ≫ is
           write("Angiv kanterne: ")
           (+ Var i: Int
```

```
        i := 0
        do i ≠ m →
              read [e]
              EP'Insert [P] (e, e.p)
              i := i+1
        od
    +)
```

Prims Algoritme

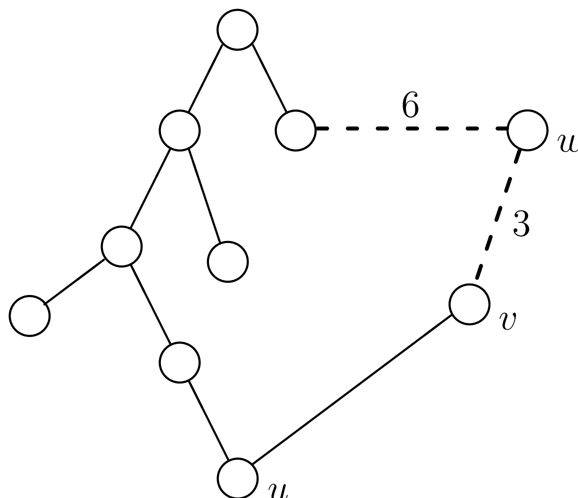
Dette er den tidligere omtalte realisering, der minder om Graffarvning. Her er der nu kun ét blåt træ, som vokser skridt for skridt ved at det udvides med den letteste kant, der forbinder det med resten af grafen. Hvis vi sammenligner med Graffarvning, svarer det til, at det er den letteste "lyserøde" kant der tilføjes.

Algoritmen konstrueres ved, at de lyserøde kanter organiseres i en prioritetskø, hvor de udvælges efter stigende prioritet. En sådan organisering kan imidlertid betyde, at prioritetskøen kommer til at indeholde helt op til $O(m)$ kanter, og hvis m er større end n betyder det, at én og samme knude er forbundet til det blå træ med mere end én kant. Det er ikke nødvendigt at have alle disse kanter i prioritetskøen, den letteste er fuldt tilstrækkelig. Vi lader derfor i stedet køen indeholde de *knuder*, der er forbundet til træet, hver med en prioritet der er lig med vægten af den letteste kant, der forbinder knuden til træet. Vi kan illustrere det som følger (kanterne i det blå træ er igen fuldt optrukket).



Her tilhører v og w prioritetskøen med prioriteter hhv. 5 og 6. Hvis v fjernes

og kanten $\{u, v\}$ males blå, bliver situationen



w tilhører stadig prioritetskøen, men nu med prioritet 3, fordi $\{v, w\}$ nu er den letteste kant, der forbinder w til træet.

Vi får derfor brug for en datastruktur, der tillader flere operationer end normale prioritetskøer. Hertil kommer at datastrukturen udover elementet (knuden) og dets prioritet (vægten) også skal indeholde selve kanten, for ellers kan vi ikke *konstruere* det udspændende træ.

Følgende *prioritetsmængde* er brugbar. Prioritetsmængden kombinerer egenskaber fra såvel prioritetskøer som mængder. Dens elementer er af formen

$$(v, a, p)$$

hvor v er selve elementet, a er dets *attribut*, og p er dets *prioritet*. Vi beskriver prioritetsmængden som en box på følgende måde

Box PrioritySet (A)
  Type Pset = ≪mængde af elementer af formen (v, a, p) ≫
  Proc Init[ps: Pset]
  Proc Empty[ps: Pset] → (Bool)
  Proc Insert[ps: Pset] (v: Int, a: A, p: Int)
  Proc DelMin[ps: Pset, v: Int, a: A, p: Int]
  Proc Member[ps: Pset] (v: Int) → (Bool)
  Proc Change[ps: Pset] (v: Int, a: A, p: Int)
  Proc Prty [ps: Pset] (v: Int) → (Int)
end PrioritySet.

Init, Empty, Insert og DelMin virker på samme måde som for prioritets-køer; Member undersøger om et element tilhører prioritetsmængden; Prty giver elementets prioritet og Change bruges til at ændre både prioritet og attribut for et element. Formelt opfylder procedurerne følgende specifikationer

Init[ps] **sat** $ps' = \emptyset$
Empty[ps] **sat** (Empty$' = (ps = \emptyset)) \wedge (ps' = ps)$
Insert[ps](v, a, p) **sat** $\neg$ Member[ps]$(v) \rightarrow ps' = ps \cup \{(v, a, p)\}$
DelMin[ps, v, a, p] **sat** $ps \neq \emptyset \rightarrow ((v', a', p') \in ps) \wedge$
$(ps' = ps \setminus \{(v', a', p')\}) \wedge$
$(\forall (w, b, q) \in ps : p' \leq q)$
Member[ps](v) **sat** (Member$'$= $(\exists\, b, q : (v, b, q) \in ps)) \wedge (ps' = ps)$
Change[ps](v, a, p) **sat** $\exists\, b, q : (v, b, q) \in ps \rightarrow$
$ps' = ps \setminus \{(v, b, q)\} \cup \{(v, a, p)\}$
Prty[ps](v) **sat** Member[ps]$(v) \rightarrow (\exists\, b : (v, b, Prty') \in ps) \wedge (ps' = ps)$

Vi kan nu realisere Prim's algoritme på følgende måde, de tre variabler U, L og T er af følgende typer.

U : MS' Mset
L : PS' Pset
T : ≪repræsentation af træet≫

hvor PS er følgende Box

Box *PS*
PrioritySet (Edge)
end *PS*

og U i analogi med Graffarvning er en monoton mængde.

```
Algoritme: Prim
Stimulans:  som ovenfor
Respons  :  som ovenfor
Metode   :  ≪T:= (∅,∅)≫
            PS'Init[L]
            MS'Init[U](n)
            do ¬ (PS'Empty[L] ∧ MS'Empty[U]) →
                if PS'Empty[L] →
                        MS'DeleteSome[U,v]
                        ≪T:=T⊕({v},∅)≫
                        for (v,w,p) in E do
                                MS'Delete[U](w)
                                PS'Insert[L](w,Edge(v,w,p),p)
                        od
                & true →
                        PS'DelMin[L,v,a,p]
                        ≪T:=T⊕({v},{a})≫
                        for (v,w,p) in E do
                                if PS'Member[L](w) →
                                        if PS'Prty[L](w) > p →
                                                PS'Change[L](w,Edge(v,w,p),p)
                                        fi
                                 | MS'Member[U](w) →
                                        MS'Delete[U](w)
                                        PS'Insert[L](w,Edge(v,w,p),p)
                                fi
                        od
                fi
            od
```

Korrektheden er igen oplagt, og med hensyn til tidskompleksiteten haves følgende (vi ignorerer alle de lineære led, som vi kender fra forudgående analyser af monotone mængder o.l.).

$$
\begin{aligned}
T[\![\text{Prim}]\!](n,m) \approx n * (\ & T[\![PS' \text{ Empty}[L]]\!]+ \\
& T[\![PS' \text{ DelMin}[L,v,a,p]]\!]+ \\
& T[\![PS' \text{ Insert}[L](w,\text{Edge}(v,w,p),p)]\!]) \\
m * (& T[\![PS' \text{ Member}[L](w)]\!]+ \\
& T[\![PS' \text{ Prty}[L](w)]\!]+ \\
& T[\![PS' \text{ Change}[L](w,\text{Edge}(v,w,p),p)]\!])
\end{aligned}
$$

Vi skal nu overveje, hvordan en prioritetsmængde kan implementeres effektivt.

Det er klart, at operationerne Init, Empty, Insert og Delmin kan understøttes af en bunke på sædvanlig vis (jfr. [P&D]). Det er imidlertid også klart, at bunken ikke kan understøtte Member, Change og Prty, fordi de alle involverer en form for søgning. Member, Change og Prty kan deri-

mod implementeres effektivt som en “bitvektor”, hvor listen, i stedet for sandhedsværdier, indeholder elementernes prioriteter og attributter.

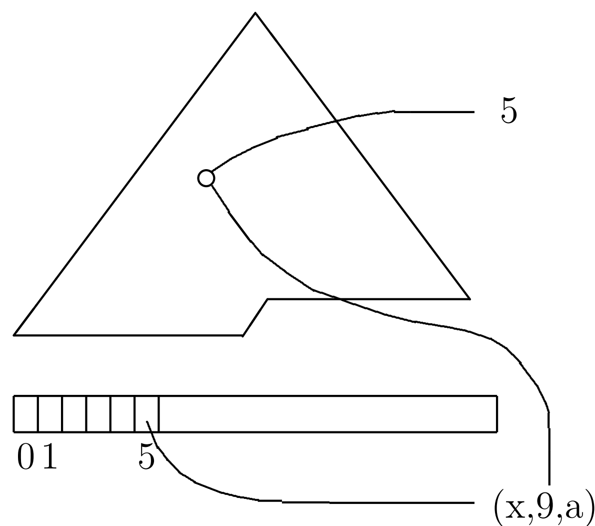
Baseret på disse overvejelser konstruerer vi nu en *hybrid* datastruktur, der består af såvel en bunke som en “bitvektor”, forbundet på passende vis. Mere præcist indrettes datastrukturen som følger.

Bunken indeholder elementerne på sædvanlig vis, og “bitvektoren”s v’te indgang indeholder en værdi af formen

$$(x, p, a)$$

hvor x er indeks for v’s placering i bunken, og p og a er hhv. prioritet og attribut for v.

Elementet 5 med prioritet 9 og attribut a vil således være repræsenteret på følgende måde (hvor x er dets indeks i bunken).



Det er ikke svært at indse, at man kan vedligeholde *begge* datastrukturerne med nogenlunde de samme omkostninger som i deres “rene” form. Hvis man f.eks. under Insert skal ombytte et element i bunken med dets far, så skal man blot sørge for også at ombytte de to indices i bitvektoren, men det er en operation, der også tager konstant tid.

Den eneste operation, der er dyrere i den hybride datastruktur, er Change. Det skyldes, at når bitvektoren er opdateret med en ny prioritet og attribut for et element, så skal elementet bringes “på plads i bunken”, dvs. placeres

i overensstemmelse med sin nye prioritet. Dette kan imidlertid gøres ved at starte i den knude, elementet befinder sig i, og så skubbe op eller ned på sædvanlig vis afhængig af den nye prioritet. Da der aldrig skal skubbes mere end bunkens højde, bliver udførelsestiden $O(\log(n))$.

Vi kan altså med denne repræsentation opnå følgende udførelsestider.

$$\begin{array}{lll} T[\![\text{Init}[ps]]\!] & \in & O(n) \\ T[\![\text{Empty}[ps]]\!] & \in & O(1) \\ T[\![\text{Insert}[ps](e,a,p)]\!] & \in & O(\log(n)) \\ T[\![\text{DelMin}[ps,e,a,p]]\!] & \in & O(\log(n)) \\ T[\![\text{Member}[ps](e)]\!] & \in & O(1) \\ T[\![\text{Change }[ps](e,a,p)]\!] & \in & O(\log(n)) \\ T[\![\text{Prty }[ps](e)]\!] & \in & O(1) \end{array}$$

Hvis dette indsættes i udtrykket ovenfor, bliver resultatet

$$T[\![\text{Prim}]\!](n,m) \in O((n+m)*\log(n))$$

dvs. en udførelsestid, der i almindelighed er af samme størrelsesorden som for Kruskals algoritme.

Det er imidlertid værd at bemærke, at Prims algoritme under visse omstændigheder kan "tunes". Hvis der nemlig er tale om en graf med mange kanter, f.eks. så mange at $m \approx n^2$, så følger det ved indsættelse af $m = n^2$, at

$$T[\![\text{Prim}]\!](n) \in O(n^2 \log(n))$$

Dette kan forbedres ved at observere, at der nu er råd til at bruge tid $O(n)$ på DelMin og Insert *forudsat* at Change får udførelsestid $O(1)$. Men det kan naturligvis klares ved blot at smide bunken væk og foretage lineær søgning i bitvektoren, hver gang der indsættes og fjernes et element. Med en sådan implementation bliver kompleksiteten af Prims algoritme for en tæt graf

$$T[\![\text{Prim}]\!](n) \in O(n^2)$$

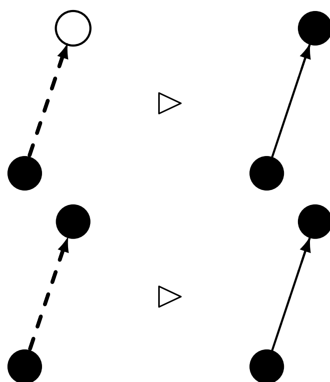
dvs. algoritmen bliver *lineær*.

4 Orienterede grafer

Dette kapitel omhandler nogenlunde de samme problemstillinger som kapitel 3, men denne gang for orienterede grafer. Da sammenhængsbegrebet er mere kompliceret for orienterede grafer end for de ikke-orienterede, er letteste udspændende træer (eller skove) ikke af helt samme interesse i denne sammenhæng. I stedet betragter vi det tilsvarende problem, *korteste veje*, som går ud på at finde korteste afstande (eller veje) i vægtede orienterede grafer.

4.1 Gennemløb

Det er særdeles simpelt at konstruere et transitionssystem til farvning af orienterede grafer; vi kan simpelthen bruge det samme som på side 6, bortset fra, at kanterne nu skal være orienterede. Vi får følgende alternativer til de to øverste transitioner.



En rød knude kaldes nu lyserød, hvis den har mindst én *udadgående* hvid kant.

Med denne udvidelse er det nemt at se, at vi kan bruge den *samme* graffarvningsalgoritme for orienterede grafer som for ikke-orienterede grafer, jfr. side 7 og side 8. Heraf følger så, at vi også for orienterede grafer kan tale om dybde-først og bredde-først gennemløb, nummerering osv.

4.1.1 Træer og rekursive gennemløb

En særlig interessant delklasse af de orienterede grafer er *træerne*. Et træ, mere præcist et *rodtræ*, er en acyklisk orienteret graf, hvor præcis én knude (roden) har indgrad 0 og alle andre knuder har indgrad 1. Et gennemløb af et sådant træ kan naturligvis foretages v.hj.a. graffarvningsalgoritmen, men da der er tale om et træ, er det (ikke overraskende) mere hensigtsmæssigt at formulere gennemløbsalgoritmen rekursivt. Følgende algoritme er den rekursive analogi til algoritmen Skitse af Graffarvning side 7.

Algoritme: Farvning af træ

Stimulans : $T = (V, E)$, hvidt træ med rod r
Respons : T, med alle knuder og kanter farvet røde
Metode : **Proc** Farv $[T:$ Træ$](v:$ Int$)$
  for $w : (v, w) \in E$ **do**
    $\ll$ farv (v, w) og w røde $\gg$
    Farv $[T](w)$
  od
end Farv

$\ll$ farv r rød $\gg$
Farv $[T](r)$

I et træ kan vi på entydig vis knytte grafens kanter til dens knuder ved at associere hver kant med den knude, den går ind i. Hvis vi nu vedtager, at kanter farves samtidig med "deres" knuder, kan vi nøjes med at interessere os for at farve knuderne, dvs. vi kan erstatte

$\ll$ farv (v, w) og w røde $\gg$

med

$\ll$ farv w rød $\gg$

Den resulterende algoritme farver alle grafens knuder røde – men det gør følgende procedure også, blot i en anden rækkefølge (farvningen af roden og kaldet Vraf$[T](v)$ skal også byttes om).

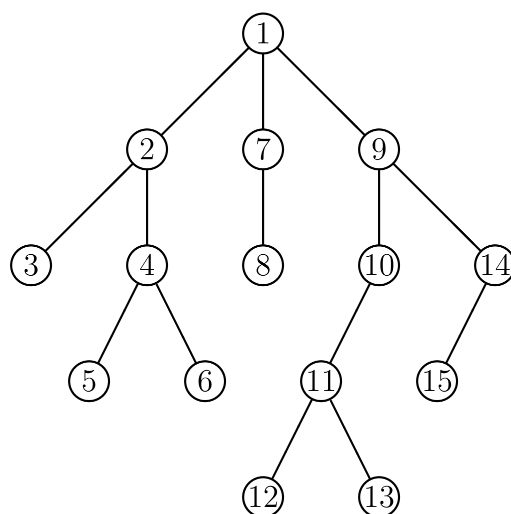
```
Proc Vraf [T : Træ](v : Int)
  for w : (v, w) ∈ E do
    Vraf [T](w)
    ≪ farv w rød ≫
  od
end Vraf
```

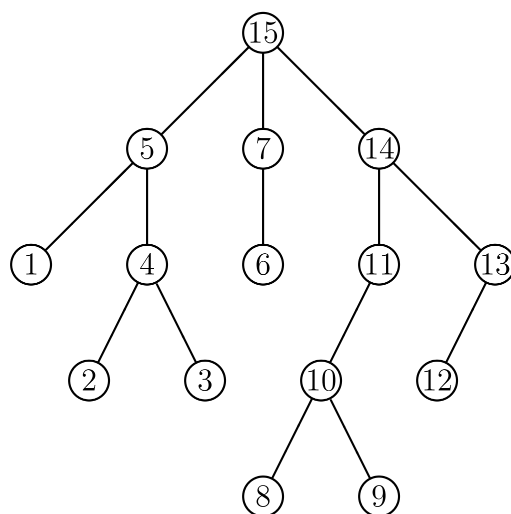
Disse forskellige metoder til farvning af træets knuder kan samles i følgende mere generelle algoritme, hvor vi taler om *forfarve* og *efterfarve* som betegnelser for de farver vi tildeler knuderne på de to tidspunkter den rekursive algoritme besøger dem.

Algoritme: Tofarvning af træ

```
Stimulans : T = (V, E), hvidt træ med rod r
Respons   : T, med alle knuder farvet
Metode    : Proc ToFarv [T : Træ](v : Int)
              ≪ forfarv v ≫
              for w : (v, w) ∈ E do
                ToFarv [T : Træ](w)
              od
              ≪ efterfarv v ≫
            end ToFarv

            ToFarv [T](r)
```

Et eksempel på en anvendelse kan være nummerering af et træ's knuder. Følgende repræsenterer hhv. en fornummerering og en efternummerering af et træ.





Disse to rækkefølger til besøg af et træ's knuder benævnes også hhv. *pre-order* og *postorder*.

4.1.2 Rekursivt dybde-først gennemløb

Det skulle være klart, at den rækkefølge hvori et træ's knuder forfarves minder meget om et dybde-først gennemløb. Dette skyldes, at den kontrol der er indbygget i rekursionsmekanismen er en stakdisciplin, og det kan derfor næppe overraske, at man også kan give en rekursiv formulering af dybde-først gennemløb af generelle grafer. Metoden fungerer såvel for orienterede som for ikke-orienterede grafer, og følgende Trine program viser hvordan. Programmet indlæser en kantlisterepræsentation af en graf

med n knuder og foretager en dybde-først nummerering i analogi med den metode, der er beskrevet side 13.

```
Process DfsRec
   (+ Type Graph = List(Vector)
      Type Colour = Sum(red: Int, white: Unit)
      Type NodeColours = List(Colour)

      Proc Dfs[G: Graph, NC: NodeColours, N: Int](v: Int)
         NC.(v), N := Colour(red: N), N+1
         (+ Var w: Int
            w := 0
            do w< | G.(v) | →
                  if is(NC.(G.(v).(w)), white) → Dfs[G, NC, N](G.(v).(w)) fi
                  w := w+1
            od
         +)
      end Dfs

      Var DFN: NodeColours
      Var G: Graph
      Var N: Int

      write("Indlæs grafen: ")
      read[G]
      DFN, N := NodeColours(Colour(white: #) | | G |), 1
      (+ Var u: Int
         u := 0
         do u< | G | →
               if is(DFN.(u), white) → Dfs[G, DFN, N](u) fi
               u := u+1
         od
      +)
      write("Dybde først orden: ", DFN, eol)
   +)
end DfsRec
```

4.2 Sammenhængsegenskaber

Der findes flere forskellige definitioner på, hvad det vil sige, at en orienteret graf er sammenhængende. Vi anfører blot følgende to.

En orienteret graf G er *svagt* sammenhængende, hvis den *tilsvarende* ikke-orienterede graf $\bar{G}$ er sammenhængende. Med den tilsvarende graf menes den ikke-orienterede graf $\bar{G}$, hvor $\{v, w\}$ er en kant hvis enten $[v, w]$ eller $[w, v]$ er en kant i G.

En orienteret graf G er *stærkt* sammenhængende, hvis der for ethvert par af knuder v og w gælder, at der både findes en vej fra v til w og en vej fra w til v.

Det er et interessant faktum, at der findes *lineære* algoritmer til at afgøre såvel svag som stærk sammenhæng. Vi skal dog nøjes med at se på det simpleste problem, nemlig svag sammenhæng. Her skal man blot finde en metode til at konstruere den til G svarende graf $\bar{G}$ i lineær tid, for så kan man bruge den lineære algoritme Graffarvning på $\bar{G}$.

Følgende algoritme viser hvordan grafen $\bar{G}$ kan konstrueres. (H er en hjælpevariabel.)

Algoritme: G til $\bar{G}$
Stimulans : G: kantliste for den orienterede graf (V, E)
Respons : $\bar{G}$: kantliste for den til G svarende ikke-orienterede graf.
Metode : H := Vector (Vector()$|n$)

```
for v ∈ V do
   for w i G.(v) do
      if v < w →≪ tilføj w til H.(v) ≫
      | w < v →≪ tilføj v til H.(w) ≫
      fi
   od
od
for v ∈ V do
   ≪fjern eventuelle dubletter fra H.(v) ≫
od
Ḡ := H
for v ∈ V do
   for w i H.(v) do
      ≪ tilføj v til Ḡ.(w) ≫
   od
od
```

Det overlades til læseren at argumentere for at algoritmen er korrekt, og at

≪ tilføj v til $H.(w)$ ≫
≪fjern eventuelle dubletter fra $H.(v)$ ≫

kan implementeres på en sådan måde, at algoritmen får lineær udførelsestid.

4.2.1 Acykliske grafer

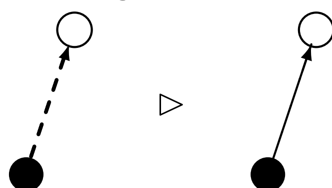
Blandt de orienterede grafer har vi allerede udpeget én interessant delklasse, nemlig træerne. Disse er igen en del af en større interessant klasse, de *acykliske grafer*. En orienteret graf er acyklisk, hvis den ikke indeholder nogen cykel.

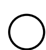
Et eksempel på en anvendelse af acykliske grafer er som følger. Hvis $S = (K, T)$ er et transitionssystem, kan vi betragte grafen G_S, hvis knuder er konfigurationerne i K, og hvor (k, k') er en kant, hvis (k, k') tilhører transitionsrelationen T. At G_S er acyklisk betyder, at transitionsystemet er gentagelsesfrit, dvs. at man ikke kan komme tilbage til en konfiguration, man har været i før.

Vi skal nu betragte en algoritme til at afgøre om en graf er acyklisk. Vi skal igen opfatte algoritmen som en farvningsproces, og vi formulerer den først i termer af et transitionssystem.

Farveterminologien er som i afsnit 3.1, og ideen i transitionssystemet er enkel: en hvid knude uden indgående hvide kanter kan farves rød, og en udgående hvid kant fra en rød knude kan farves rød.

Transitionssystem: Acyklisk farvning
Konfigurationer: Danske grafer
Transitioner :



 **hvis** ○'s hvide indgrad er 0

Transitionssystemet har følgende relevante egenskaber:

- Enhver proces der starter med en hvid graf G er endelig.
- Slutkonfigurationen er rød hvis og kun hvis G er acyklisk.

Det er klart, at alle processer er endelige. Det er også nemt at se, at hvis

G indeholder en cykel, så kan de knuder der ligger på cyklen ikke blive farvet. Dette følger af, at forudsætningen for at en knude kan farves er, at alle dens forgængere er røde, men så er forudsætningen for at farve en knude der ligger på en cykel, at den allerede er farvet.

Tilbage står at vise, at hvis G er acyklisk, så kan vi blive ved med at anvende transitioner, indtil alle knuder og kanter er farvet. Antag, at G på et tidspunkt indeholder en hvid knude v. Vi starter med at anvende den først transitionsregel (den der farver kanter) så længe som muligt. Hvis v nu ikke kan farves, er det fordi den har en indgående hvid kant, som heller ikke kan farves. Men det betyder at denne kant udgår fra en anden *hvid* knude w. Hvis w heller ikke kan farves, kan vi gentage argumentet og finde en ny hvid kant og knude. Dette kan vi blive ved med indtil vi finder en hvid knude, der kan farves – dette må ske, for ellers er grafen cyklisk. Vi farver nu denne knude og har altså vist, at så længe der findes hvide knuder, kan en af dem farves.

Det er nemt at skrive en algoritme, der realiserer transitionssystemet, og som dermed kan bruges til at afgøre om en orienteret graf er cyklisk. Vi overlader dette til læseren og skriver i stedet en variant af algoritmen, som foretager en såkaldt *Topologisk Sortering* af en acyklisk graf. En topologisk sortering er en nummerering af grafens knuder, som er i overensstemmelse med grafens struktur på følgende måde

> TopSort.$(v) <$ TopSort.(w) medfører at der ikke findes nogen vej fra w til v i G.

Følgende algoritme sorterer grafen G topologisk. Variablen R, som indeholder de røde knuder, er en nondeterministisk mængde. Indegree er en vektor, der indeholder knudernes hvide indgrader.

<u>Algoritme: Topologisk Sortering</u>

Stimulans : $G = (V, E)$ orienteret acyklisk graf
Respons : TopSort: Vector, indeholder topologisk sortering af G
Metode : ≪indlæs grafen i G≫

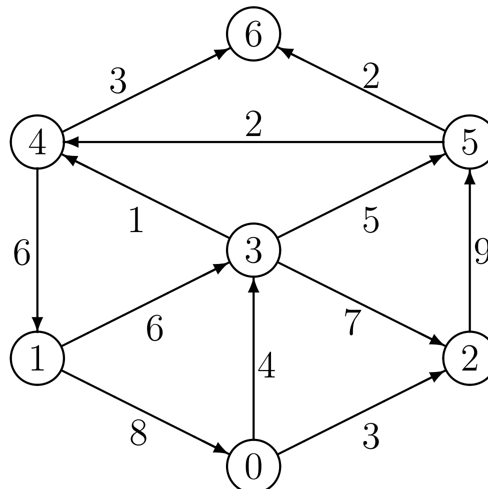
```
Indegree:=Vector(0|n)
for (v,w) in E do
        Indegree.(w):=Indegree.(w)+1
od
NS'Init[R](n)
for v in V do
       if Indegree.(v) = 0 → NS'Insert[R](v) fi
od
TopSort,N:=Vector(0|n),1
do ¬ NS'Empty[R] →
       NS'DeleteSome[R,v]
       TopSort.(v),N:=N,N+1
       for (v,w) in E do
              Indegree.(w):=Indegree.(w)-1
              if Indegree.(w) = 0 → NS'Insert[R](w) fi
       od
od
```

Da udførelsestiden for operationerne i den non-deterministiske mængde er $O(1)$, er det klart at vi har

$$T[\![\text{Topologisk Sortering}]\!](n, m) \in O(m + n)$$

4.3 Korteste veje

I dette afsnit skal vi igen betragte *vægtede* grafer, men nu er graferne naturligvis orienterede. Følgende er et eksempel på en vægtet orienteret graf med heltallige vægte.



Intuitionen bag anvendelser af vægtede grafer kan igen være, at vægtene betegner afstande, men nu er vejene "ensrettede", og det problem vi skal interessere os for er at finde korteste orienterede veje. Vi skal først betragte det såkaldte *enkelt-kilde* (eng.: single source) problem, der går ud på at beregne længden af den korteste vej fra en given knude (kilden) til alle de øvrige knuder i grafen. Vi skal antage, at knude 0 er kilden, og skal altså beregne længden af den korteste vej fra 0 til de øvrige knuder. I grafen ovenfor er længderne af de korteste veje som følger:

$$D\text{: } (0{,}11{,}3{,}4{,}5{,}9{,}8)$$

Der findes flere metoder til at beregne korteste veje, vi skal vælge én, der ligger meget tæt på Prims algoritme til at finde letteste udspændende træer.

4.3.1 Enkelt-kilde korteste veje

Algoritmen opbygger en mængde af blå knuder ved en for en at udvælge kandidater fra en passende prioritetsmængde. Alle knuder starter med at

være hvide, og de farves blå, efterhånden som deres korteste afstand fra 0 bliver kendt.

Algoritmen ser ud som følger. D er af typen

Type DistList = **List** (**Sum** (blue: Int, white: Unit)

og L er en prioritetsmængde, hvis elementer er af formen (v, w, p).

En knude v bliver blå, når $D.(v)$ skifter fra white til blue. Attributterne i prioritetsmængden er *knuder* (hvor der i Prims algoritme var tale om *kanter*). Denne ændring skyldes udelukkende ønsket om en passende simplificering, idet vi blot har erstattet den lidt redundante repræsentation $(v,\ \text{Edge}(w, v, p), p)$ med den simplere (v, w, p). Det skulle være klart, at vi stadig har samme information til rådighed.

```
Algoritme: Dijkstra
Stimulans : G = (V, E) orienteret graf med ikke-negative vægte
Respons   :      is (D.(v), blue) ⇒ D.(v).blue
            D:      er længden af den korteste vej fra 0 til v
                 is (D.(v), white) ⇒ v kan ikke nås fra 0
Metode    : PS'Init [L]
            MS'Init [U] (n)
            D := DistList (Dist (white: #) |n)
            MS'Delete [U] (0)
            D.(0) := Dist (blue: 0)
            for (0, w, p) in E do
                    MS'Delete [U] (w)
                    PS'Insert [L] (w, 0, p)
            od
            do ¬ PS'Empty [L] →
                          PS'DelMin [L, v, a, p]
                          D.(v) := Dist (blue: p)
                          for (v, w, p) in E do
                                  if PS'Member [L] (w) →
                                          if PS'Prty [L] (w) > p+D.(v).blue →
                                                  PS'Change [L] (w, v, p+D.(v).blue)
                                          fi
                                  | MS'Member [U] (w) →
                                          MS'Delete [U] (w)
                                          PS'Insert [L] (w, v, p+D.(v).blue)
                                  fi
                          od
            od
```

Som det fremgår, minder algoritmen meget om Prims algoritme, og det er da også nemt at se, at den har samme udførelsestid, dvs.

$$T[\![\text{Dijkstra}]\!](n, m) \in O((n + m) * \log(n))$$

hvilket, for tætte grafer hvor $m \approx n^2$, igen kan forbedres til det lineæare

$$T[\![\text{Dijkstra}]\!](n) \in O(n^2)$$

At algoritmen er korrekt, indses ved hjælp af følgende invariant (en *lyseblå* vej er en vej der starter ved kilden, passerer et antal blå knuder, og slutter ved en hvid knude i L).

I: (for alle blå knuder v gælder, at $D.(v)$.blue er længden af den korteste vej fra 0 til v)

og

(for alle v i L gælder, at PS'Prty$[L](v)$ er lig med længden af den korteste lyseblå vej til v)

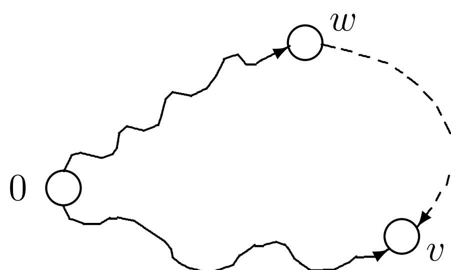
og

(for enhver hvid knude v gælder, at hvis der findes en vej fra 0 til v, så er der et præfiks af vejen, der er lyseblå).

Det er klart, at gyldighed af I (samt terminering som er oplagt) medfører korrekthed af algoritmen.

Det er ikke vanskeligt at indse, at I er gyldig, og det meste af beviset overlades til læseren. Vi skal dog vise kernen i argumentet, som er følgende: for den knude i L, hvis lyseblå vej er kortest, gælder, at denne lyseblå vej også er den korteste blandt alle de veje i grafen, der forbinder kilden til knuden. Dette medfører umiddelbart, at vi kan farve denne knude blå, og herefter følger gyldigheden af I.

Lad v være den knude i L, der har lavest prioritet, dvs. hvis "lyseblå" afstand fra 0 er mindst. Antag nu, at der findes en anden vej fra 0 til v som er kortere. Da v er hvid, starter denne vej med et lyseblåt præfiks, dvs. den indeholder en (anden) knude w fra L. Vi har følgende situation



hvor både v og w tilhører L, og de fuldt optrukne veje er lyseblå. Nu er w's "lyseblå" afstand imidlertid større end eller lig med v's, så vejen til v via w kan kun være kortest, hvis den stiplede vej indeholder en kant med negativ vægt. Men det er umuligt, fordi grafens vægte er ikke-negative. Altså er v's afstand korrekt og beviset er ført.

4.3.2 Alle korteste veje

I dette afsnit ser vi på en algoritme, der finder den korteste vej mellem alle par af knuder i en orienteret graf. Det er klart, at problemet kan løses ved at udføre Dijkstras Algoritme n gange, hvilket giver en total udførelsestid på

$$T[\![n \text{ gange Dijkstra}]\!](n, m) \in O(n * (n + m) * \log(n))$$

Vi skal imidlertid se på en anden metode, som, udover at være effektiv, også illustrerer et interessant princip til approximativ løsning af opgaven.

Betragt igen den orienterede graf fra side 38. Vi kan repræsentere grafen v.hj.a. en særlig incidensmatrix på følgende form

$$g: \begin{array}{ccccccc} 0 & \infty & 3 & 4 & \infty & \infty & \infty \\ 8 & 0 & \infty & 6 & \infty & \infty & \infty \\ \infty & \infty & 0 & \infty & \infty & 9 & \infty \\ \infty & \infty & 7 & 0 & 1 & 5 & \infty \\ \infty & 6 & \infty & \infty & 0 & \infty & 3 \\ \infty & \infty & \infty & \infty & 2 & 0 & 2 \\ \infty & \infty & \infty & \infty & \infty & \infty & 0 \end{array}$$

hvor

$$g.(v,w) = \begin{cases} p & \text{hvis } (v,w,p) \in E \\ 0 & \text{hvis } v = w \\ \infty & \text{ellers} \end{cases}$$

Vi starter som sædvanligt i en situation, hvor alle grafens knuder er hvide, og vi vil nu farve dem *grønne* én efter én. Samtidig vil vi opretholde følgende invariant (vi kalder en vej *lysegrøn*, hvis alle dens knuder, pånær evt. den første og den sidste, er grønne)

I: $g.(v,w)$ er lig med længden af den korteste lysegrønne vej fra v til w

(Det skal bemærkes, at vi tildeler en ikke-eksisterende vej længden ∞.)

Algoritme: Floyd
Stimulans : $G = (V,E)$ orienteret, hvid graf med ikke-negative vægte
Respons : $g : g.(v,w) =$ længden af den korteste vej fra v til w
Metode : ≪initialiser g som ovenfor≫

```
u := 0
do u ≠ n →
    ≪farv u grøn≫
    for (v, w) ∈ V × V do
        g.(v, w) := min {g.(v, w),
                         g.(v, u) + g.(u, w)}
    od
    u := u + 1
od
```

Det er klart, at der gælder

$$T[\![\text{Floyd}]\!](n) \in \Theta(n^3)$$

og det er også klart, at gyldigheden medfører korrekthed, fordi alle veje i grafen er lysegrønne, når algoritmen standser.

At invarianten er gyldig følger af, at de eneste nye lysegrønne veje, der opstår når u farves grøn, består af sammensætningen af to eksisterende lysegrønne veje, hvor den første ender i u og den anden begynder i u. Men disses længder findes jo i hhv. $g.(v, u)$ og $g.(u, w)$.

Følgende TRINE program realiserer algoritmen. Programmet indlæser først antallet af knuder, og dernæst et antal linier af formen

$$v \;\; w \;\; p$$

som hver repræsenterer kanter $[v, w]$ med vægt p. Indlæsningen afsluttes med "kanten" $[0, 0]$.

```
Process Floyd
    (+ Type EInt = Int
       Type Matrix = List(List(EInt))

       Proc Eplus(x, y: EInt) → (EInt)
           if ¬ (is(x) ∨ is(y)) → return ?-EInt
           & true → return x+y
           fi
       end Eplus

       Proc Emin(x, y: EInt) → (EInt)
           if ¬ is(x) → return y
           | ¬ is(y) → return x
           & true →
                if x ≤ y → return x
                | x ≥ y → return y
                fi
           fi
       end Emin

       Var G: Matrix
       Var n: Int

       write("Antal knuder: ")
       read[n]
       G := Matrix(List(?-EInt | n) | n)
       ≪ Indlæs kanter og initialiser G ≫
       (+ Var u, v, w: Int
          u := 0
          do u ≠ n →
               v := 0
               do v ≠ n →
                    w := 0
                    do w ≠ n →
                         G.(v, w) := Emin(G.(v, w), Eplus(G.(v, u), G.(u, w)))
                         w := w+1
                    od
                    v := v+1
               od
               u := u+1
          od
       +)
       ≪ Udskriv G ≫
    +)
end Floyd
where ≪ Indlæs kanter og initialiser G ≫ is
          (+ Var u, v, p: Int
             read[u, v, p]
             do (u ≠ 0) ∨ (v ≠ 0) →
                  G.(u, v) := p
                  read[u, v, p]
             od
             u := 0
             do u ≠ n →
                  G.(u, u) := 0
                  u := u+1
             od
          +)

where ≪ Udskriv G ≫ is
          (+ Var i: Int
             i := 0
             do i ≠ n →
                  write(G.(i), eol)
                  i := i+1
             od
          +)
```

5 Del-og-kombiner

Del-og-kombiner[1] teknikken bygger på den simple observation, at hvis et problem kan opdeles i et antal simplere problemer, hvis løsninger kan kombineres til en løsning af det oprindelige problem, så er en sådan opdeling et skridt på vejen mod en løsning af selve problemet. Sagt lidt mere præcist kan del-og-kombiner teknikken anvendes i situationer, hvor et problem p kan opdeles i et antal delproblemer $p_1, \ldots, p_k$ på en sådan måde at

1. hvert delproblem p_i er simplere end p
2. løsningerne til $p_1, \ldots, p_k$ kan kombineres til en løsning af p.

Det skulle være klart, at man ved gentagen anvendelse af denne teknik får reduceret sit problem til et (muligvis stort) antal simple problemer, som man så antages at kunne løse direkte. I konkrete anvendelser af denne teknik vil man ofte realisere løsningerne af delproblemerne $p_1, \ldots, p_k$ i form af kald af én eller flere rutiner, som hver løser et delproblem.

Hvis disse delproblemer er af *samme slags* som det oprindelige problem, kan man introducere en procedure til at løse det oprindelige problem, som vil kunne skrives (ofte meget elegant og overskueligt) under anvendelse af *rekursion*.

Procedurerne `reverse1` og `reverse2` s. 35 i [P&D] er et udmærket eksempel herpå, idet ligningen

$$\tilde{L} = L(i..|L|) ++ L(0..i)$$

på meget direkte måde repræsenterer følgende anvendelse af del-og-kombiner:

> Problemet *spejling af en følge* L løses ved at opdele L i to mindre problemer, *spejling af* $L(0..i)$ og *spejling af* $L(i..|L|)$, hvis løsninger, $\widetilde{L(0..i)}$ og $\widetilde{L(i..|L|)}$, kombineres til $\tilde{L}$ ved simpel omvendt sammensætning.

[1] Det engelske navn for teknikken er *divide-and-conquer*, som vel nærmest burde oversættes ved del-og-hersk, men del-og-kombiner er en mere dækkende betegnelse.

Det skulle være nemt at se, at proceduren `Kochline` s. 159 i [TRINE] også er skrevet under anvendelse af del-og-kombiner teknikken. Her er problemet at tegne en kurve af en vis længde i en vis retning. Dette deles op i det (simple) problem at tegne en streg, efterfulgt af fire delproblemer, der består i at tegne mindre kurver af passende længde i passende retninger.

5.1 Skabelonen

Da del-og-kombiner teknikken er anvendelig i en lang række forskellige situationer, præciserer vi den i form af følgende såkaldte *algoritme-skabelon*, som er en (abstrakt) algoritmebeskrivelse, i hvilken der udover stimulans, respons og metode også optræder et såkaldt *univers*, som er en beskrivelse af egenskaberne ved de elementer metode-delen manipulerer samt af de operatorer, der anvendes på dem.

Skabelon: Del-og-kombiner

Univers : En klasse af *problemer* P, som har *løsninger* i en klasse L. P består af *simple* problemer S og *komplekse* problemer K, dvs. $P = S \cup K$. Problemer $p \in S$ har en *direkte* løsning, medens problemer $p \in K$ kan *deles* i et antal mindre problemer $p_1, \ldots, p_k \in P$, hvis løsninger $l_1, \ldots, l_k \in L$ kan *kombineres* til $l \in L$, hvor l er en løsning til P.

Stimulans : p: P, problem der ønskes løst

Respons : l: L, løsning af p

Metode :

```
proc løs[p : P, l : L]
    if ≪ p er i S ≫→
        ≪find løsning l direkte≫
    & true →
        (+ Var p1, ..., pk : P
           Var l1, ..., lk : L
           ≪del p i p1, ..., pk ≫
           løs[p1, l1]
           ⋮
           løs[pk, lk]
           ≪kombiner l1, ..., lk til l ≫
        +)
    fi
end løs

løs[p, l]
```

Eksemplet med spejling af tegnfølger repræsenterer en konkretisering af skabelonen, hvor problemerne P består i at spejle tegnfølger af vilkårlig længde. De simple problemer er dem, hvor længden af følgerne er mindre end eller lig med 1, og løsningen består i at "gøre ingenting". Tegnfølger med længde større end 1 deles i to kortere tegnfølger, og kombinationen af løsninger består i at sætte løsningsfølgerne efter hinanden.

Vi overlader det til læseren at overbevise sig om, at programmet Kochline også repræsenterer en konkretisering af del-og-kombiner skabelonen. Be-

mærk i den forbindelse, at *løsninger* ikke altid repræsenteres i form af eksplicitte parametre i del-og-kombiner procedurerne. Løsninger kan, afhængigt af situationen, f.eks. også repræsenteres af følger af udskrifter (i grafikpakken). Når vi i det følgende konkretiserer skabelonen, vil det altid fremgå af sammenhængen, hvordan løsninger repræsenteres.

5.2 Hanois Tårne

Hanois Tårne [Harel, p. 31] er også et eksempel på en anvendelse af del-og-kombiner, hvor skabelonen er konkretiseret på følgende måde.

P: flytning af n ($n > 0$) skiver fra én stang til én af de to tomme stænger under anvendelse af den anden tomme stang som mellemstation.

L: følger af enkeltflytninger, hvor en enkeltflytning består i at flytte den øverste skive fra én stang til en anden. En enkeltflytning angives ved

$$\alpha \rightarrow \beta$$

hvor α og β er navnet på en stang.

K: problemer med mindst 2 skiver.

Det er naturligvis trivielt at løse de simple problemer. Med hensyn til de komplekse problemer i K, løses de som bekendt ved, at flytning af n skiver fra α til δ via β kan opdeles i tre delproblemer på følgende måde

p_1: flyt $n-1$ skiver fr α til β via δ
p_2: flyt 1 skive direkte fra α til δ
p_3: flyt $n-1$ skiver fra β til δ via α

Hvert af disse problemer er mindre end det oprindelige, og deres løsninger udgør tilsammen en løsning af p. Det er imidlertid ikke umiddelbart klart, at p_1, p_2 og p_3 tilhører klassen P, fordi vi her har forudsat at de to af stængerne er *tomme*, og det er δ-stangen ikke, når vi løser p_3. Det er imidlertid oplagt, at da δ-stangen indeholder den *største* af skiverne, som alle andre

skiver kan ligge oven på, gør det ikke noget, at den ikke er tom. Vi kan klare dette formelle problem ved at *generalisere* problemklassen P til

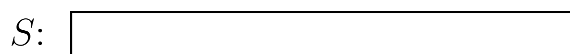
> P: flytning af de øverste $n(n \geq 0)$ skiver fra én stang til én af to andre stænger, som begge kun indeholder skiver, der er større end de n skiver, der skal flyttes

og så er det nemt at se, at såvel p som p_1, p_2 og p_3 alle tilhører P.

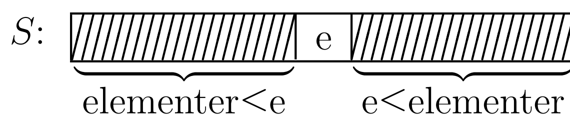
Sortering er en anden problemstilling, hvor del-og-kombiner finder anvendelse og kan føre til hensigtsmæssige algoritmer. Vi skal se på to eksempler, som går under navnene hhv. Quicksort og Flettesortering.

5.3 Quicksort

Quicksort sorterer elementerne i en vektor i ikke-aftagende orden v.hj.a. omflytninger. Ideen i algoritmen kan illustreres på følgende måde. Hvis elementerne i en vektor



opdeles på følgende måde



kan S sorteres ved at sortere de to "halvdele", som hver repræsenterer et mindre problem af samme slags. Til selve opdelingen kan vi bruge algoritmen Opdel s. 22 i [P&D], idet vi som deleelement vælger et tilfældigt element i vektoren.

Idet vi repræsenterer delvektorer v.hj.a. to "pegepinde" venstre og højre, foregår sorteringen mere præcist på følgende måde.

Algoritme: Quicksort

Stimulans : S: vector

Respons : S: sorteret i ikke-aftagende orden

Metode : **proc** opdel [S: Vector, r: Int]($venstre,højre,e$: Int)

```
    ≪Udfør algoritmen Opdel med stimulans
        S(venstre..højre) og e, og med respons r ≫
end Opdel

proc quicksort [S: Vector](venstre,højre: Int)
    if venstre < højre−1 →
        (+ var i, r: int
            ≪vælg i så venstre ≤ i < højre≫
            opdel [S, r](venstre,højre,S.(i))
            { S(venstre..r) ≤ e ≤ S(r..højre) og
                S.(i) er ikke blevet flyttet}
            if i < r → r := r − 1 fi
            S.(i) :=: S.(r)
            {S.(r) står rigtigt }
            quicksort[S](venstre,r)
            quicksort[S](r+1,højre)
        +)
    fi
end quicksort

quicksort[S](0, |S|)
```

Følgende komplette TRINE program realiserer algoritmen:

```
Process Quicksort
    (+ Proc Opdel [S: Vector, r: Int] (venstre, højre, e: Int)
            (+ Var lav, høj: Int
               lav, høj := venstre, højre
               do lav<høj →
                     if S.(lav) ≤ e → lav := lav+1
                     | (S.(høj-1)<e) ∧ (e<S.(lav)) →
                           S.(lav) :=: S.(høj-1)
                           lav, høj := lav+1, høj-1
                     | e ≤ S.(høj-1) → høj := høj-1
                     fi
               od
               r := lav
            +)
        end Opdel

        Proc Quicksort [S: Vector] (venstre, højre: Int)
            if venstre<højre-1 →
                  (+ Var r, i: Int
                     i := random(venstre, højre)
                     Opdel [S, r] (venstre, højre, S. (i))
                     if i<r → r := r-1 fi
                     S.(i) :=: S.(r)
                     Quicksort [S] (venstre, r-1)
                     Quicksort [S] (r+1, højre)
                  +)
            fi
        end Quicksort

        Var u: Vector

        write("Indlæs vektor: ")
        read [u]
        Quicksort [u] (0, | u |)
        write("Resultat: ", u, eol)
    +)
end Quicksort
```

Læseren opfordres til at overbevise sig om, at quicksort er en korrekt konkretisering af del-og-kombiner skabelonen, herunder især at de i algoritmen anførte udsagn er gyldige.

Da algoritmen bærer navnet *quick* sort, ville man forvente, at den var effektiv. Det er den imidlertid ikke altid, hvilket kan indses på følgende måde.

Det er klart, at der for algoritmen Opdel gælder

$$T[\![\text{Opdel}]\!](n) \in \Theta(n)$$

hvor n nu er længden af vektoren $S(venstre..højre)$, dvs. $n = højre\text{-}venstre$.

De eneste sætninger i kroppen af quicksort, der har ikke-konstant omkostning, er kaldet af opdel samt de to rekursive kald af quicksort. Vi har

derfor

$$\begin{aligned} T[\![\textbf{proc} \text{ quicksort}]\!](n) \approx\ & T[\![\textbf{proc} \text{ opdel}]\!](n) + \\ & T[\![\textbf{proc} \text{ quicksort}]\!](m) + \\ & T[\![\textbf{proc} \text{ quicksort}]\!](n-m-1) \end{aligned}$$

hvor $m = r\text{-}venstre$. Størrelsen af m afhænger altså af, hvor heldige vi er med valget af det element $S.(i)$, der bruges til at opdele efter. Hvis uheldet er ude, kan vi risikere hver gang at vælge det mindste element i S. Det betyder at $r = venstre$, dvs. at $m = 0$. Nu er det klart at

$$T[\![\textbf{proc} \text{ quicksort}]\!](0) \in O(1)$$

hvorfor ligningen ovenfor bliver

$$T[\![\textbf{proc} \text{ quicksort}]\!](n) \approx n + T[\![\textbf{proc} \text{ quicksort}]\!](n-1)$$

Men så følger det at

$$T[\![\textbf{proc} \text{ quicksort}]\!](n) \in \Theta(n^2)$$

Worst case tidskompleksiteten af quicksort er således ikke bedre end de simple sorteringsalgoritmer.

Betragter man derimod den *forventede* udførelsestid, ser sagen anderledes ud. Dette følger af, at det element der udvælges til at dele på, $S.(i)$, vælges *tilfældigt* blandt elementerne i $S(venstre..højre)$. Idet vi for resten af denne analyse antager, at alle elementer i S er parvis forskellige, betyder dette, at værdien af r (dvs. placeringen af delepunktet) også er tilfældig. Men så kan vi finde den forventede tidskompleksitet af quicksort (som vi skal betegne med T_E) på følgende måde

$$\begin{aligned} & T_E[\![\textbf{proc} \text{ quicksort}]\!](n) \approx \\ & \quad n + \sum_{m=0}^{n-1} \frac{1}{n}(T_E[\![\textbf{proc} \text{ quicksort}]\!](m) + T_E[\![\textbf{proc} \text{ quicksort}]\!](n-m-1)) \end{aligned}$$

idet der er samme sandsynlighed, $\frac{1}{n}$, for alle værdier af m.

Man kan, ved induktion efter n, bevise, at når $T_E[\![\textbf{proc} \text{ quicksort}]\!]$ tilfredsstiller denne ligning, så findes der en positiv konstant k, således at

$$T_E[\![\textbf{proc} \text{ quicksort}]\!](n) \le k * n * \log(n)$$

hvoraf vi kan konkludere, at

$$T_E[\![\textbf{proc}\ \text{quicksort}]\!](n) \in O(n \log(n))$$

Dette indikerer, og praksis bekræfter da også, at Quicksort faktisk er en effektiv sorteringsmetode.

5.4 Flettesortering

Det næste eksempel på del-og-kombiner er som nævnt flettesortering. Ideen i denne algoritme er, at man kan sortere en vektor ved at dele den i to halvdele, som sorteres hver for sig, og derefter flette de to halvdele sammen til én sorteret følge.

Algoritme: Flettesortering

Stimulans : S: vector

Respons : S: sorteret

Metode : **proc** flet [S_1, S_2, S: Vector]

```
proc flet [S₁, S₂, S: Vector]
    ≪udfør algoritmen Fletning med
       stimulans S₁ og S₂ og respons S ≫
end flet

proc sorter [S: Vector]
    if |S| > 1 →
       (+ var S1, S2: vector
          S1, S2 := S(0..|S|/2), S(|S|/2..|S|)
          sorter[S1]
          sorter [S2]
          flet [S1, S2, S]
       +)
    fi
end sorter

sorter [S]
```

Det skulle være klart, at algoritmen er korrekt, og m.h.t. dens kompleksitet er det også klart, at der gælder

$$
\begin{aligned}
T[\![\textbf{proc}\ \text{sorter}]\!](n) \;\approx\;& n + \\
& 2 * T[\![\textbf{proc}\ \text{sorter}]\!](\frac{n}{2}) + \\
& T[\![\textbf{proc}\ \text{flet}]\!](n)
\end{aligned}
$$

hvor n angiver længden af S. Hvis flettealgoritmen kan implementeres så den får kompleksitet $O(n)$, giver dette anledning til ligningen

$$T[\![\textbf{proc}\ \text{sorter}]\!](n) \approx 2 * (n + T[\![\textbf{proc}\ \text{sorter}]\!](\frac{n}{2}))$$

Dette er den samme ligning som opstod under analysen af proceduren `reverse2`, og løsningen er

$$T[\![\textbf{proc}\ \text{sorter}]\!](n) \in \Theta(n \log(n))$$

Følgende TRINE program viser en implementation af flettesortering, hvor kompleksiteten af fletningen er $O(n)$. Læseren opfordres til at studere hvordan dette foregår.

```
Process Flettesortering
    (+ Proc flet [S1, S2, S: Vector]
          (+ Var i1, i2, i: Int
             S := Vector(0 | |S1| + |S2|)
             i1, i2, i := 0, 0, 0
             do ¬(i1 = |S1|) ∧ (i2 = |S2|) →
                   S.(i) := S1.(i1)
                   i1, i := i1+1, i+1
             | (i1 = |S1|) ∧ ¬(i2 = |S2|) →
                   S.(i) := S2.(i2)
                   i2, i := i2+1, i+1
             | ¬(i1 = |S1|) ∧ ¬(i2 = |S2|) →
                   if S1.(i1) ≤ S2.(i2) →
                         S.(i) := S1.(i1)
                         i1, i := i1+1, i+1
                   | S2.(i2) ≤ S1.(i1) →
                         S.(i) := S2.(i2)
                         i2, i := i2+1, i+1
                   fi
             od
          +)
    end flet
```

```
        Proc sorter [S: Vector]
            if | S | >1 →
                (+ Var S1, S2: Vector
                    S1, S2 := S(0 .. | S |/2), S(| S |/2 .. | S |)
                    sorter [S1]
                    sorter [S2]
                    flet [S1, S2, S]
                +)
            fi
        end sorter

        Var u: Vector

        write("Indlæs vektor: ")
        read [u]
        sorter [u]
        write("Resultat: ", u, eol)
    +)
end Flettesortering
```

Det skal bemærkes, at Flettesortering, i modsætning til Quicksort, *ikke* er et eksempel på en omflytningsalgoritme, således som dette er defineret i [P&D]. Årsagen er, at fletningen *kopierer* elementerne fra listerne S1 og S2 til listen S, dvs. den indskrænker sig ikke til blot at bruge ombytninger. Dette er af betydning for procedurens *lagerforbrug*, hvilket kan ses på følgende måde.

Hvis vi, i analogi med betegnelsen for tidskompleksitet, betegner procedurens pladsforbrug *eksklusive* hvad parametrene fylder, med $P[\![\textbf{proc}\ \text{sorter}]\!]$ får vi

$$
\begin{aligned}
P[\![\textbf{proc}\ \text{sorter}]\!](n) &\approx n + \\
&\quad 2 * P[\![\textbf{proc}\ \text{sorter}]\!](\frac{n}{2}) + \\
&\quad P[\![\textbf{proc}\ \text{flet}]\!](n) \\
P[\![\textbf{proc}\ \text{flet}]\!](n) &\approx 1
\end{aligned}
$$

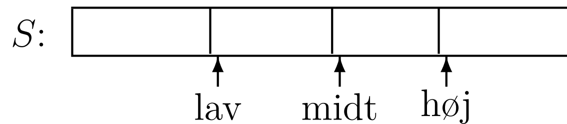
fordi sorter indeholder to lokale variabler S_1, S_2, hvis samlede længde er n, medens proceduren flet kun indeholder simple lokale variabler. (Bemærk, at udsagnet i procedure flet angiver, at der for ethvert kald flet[S_1,S_2,S] gælder, at $|S| = |S_1| + |S_2|$.)

Denne ligning for $P[\![\textbf{proc}\ \text{sorter}]\!]$ er igen velkendt og har løsning

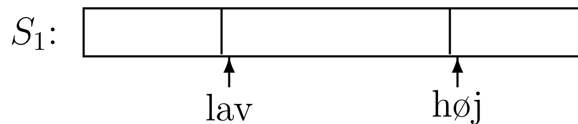
$$P[\![\textbf{proc}\ \text{sorter}]\!](n) \in \Theta(n \log(n))$$

Der bruges med andre ord lige så meget ekstra lager som der bruges tid.

Dette kan være kritisk i praktiske anvendelser, men heldigvis kan man klare sig med mindre. Der kræves nemlig blot én hjælpevektor S af længde n, idet man kan flette skiftevis fra S til S_1 og fra S_1 til S. De to stumper, der flettes, er altid naboer og deres afgrænsning kan derfor angives på følgende måde



Efter fletningen vil S_1 have følgende udseende



hvor S_1(lav..høj) indeholder fletningen af S(lav..midt) og S(midt..høj).

Et program, der fungerer på denne måde, ser ud som følger. Variablen p af type Position bruges til at angive, om data befinder sig på S_1 eller S_2.

```
Process FlettesorteringEnTo
    (+ Proc FletEnTo [S1, S2: Vector] (lav, midt, høj: Int)
           (+ Var i1, i2, i: Int
              i1, i2, i := lav, midt, lav
              do ¬ (i1 = midt) ∧ (i2 = høj) →
                   S2.(i) := S1.(i1)
                   i1, i := i1+1, i+1
              | (i1 = midt) ∧ ¬ (i2 = høj) →
                   S2.(i) := S1.(i2)
                   i2, i := i2+1, i+1
              | ¬ (i1 = midt) ∧ ¬ (i2 = høj) →
                   if S1.(i1) ≤ S1.(i2) →
                        S2.(i) := S1.(i1)
                        i1, i := i1+1, i+1
                   | S1.(i2) ≤ S1.(i1) →
                        S2.(i) := S1.(i2)
                        i2, i := i2+1, i+1
                   fi
              od
           +)
       end FletEnTo

       Type Position = Sum(en, to: Unit)
```

```
Proc SorterEnTo [S1, S2: Vector, p: Position] (lav, høj: Int)
    if høj-lav>1 →
        (+ Var midt: Int
           Var p1, p2: Position
           midt, p1, p2 := (lav+høj)/2, p, p
           SorterEnTo [S1, S2, p1] (lav, midt)
           SorterEnTo [S1, S2, p2] (midt, høj)
           if p1 ≠ p2 → ≪S1(midt..høj):=:S2(midt..høj≫ fi
           if is(p1, en) →
                  FletEnTo [S1, S2] (lav, midt, høj)
                  p := Position (to: #)
           | is(p1, to) →
                  FletEnTo [S2, S1] (lav, midt, høj)
                  p := Position (en: #)
           fi
        +)
    fi
end SorterEnTo

Var u, v: Vector
Var p: Position

write("Indlæs vektor: ")
read [u]
v, p := Vector (0 | | u |), Position (en: #)
SorterEnTo [u, v, p] (0, | u |)
write("Resultat: ")
if is(p, en) → write(u, eol)
| is(p, to) → write(v, eol)
fi
  +)
end FlettesorteringEnTo
where ≪S1(midt..høj):=:S2(midt..høj≫ is
        (+ Var i: Int
           i := midt
           do i<høj →
                  S1.(i) :=: S2.(i)
                  i := i+1
           od
        +)
```

5.5 Selektion

Et problem, der er beslægtet med sortering, er det såkaldte selektions-problem, som går ud på at skrive en procedure

Proc Select (S: Vector, k: Int) → (Int)

som finder det k'te element (efter størrelse) i S. Mere præcist skal Select tilfredsstille specifikationen

Select(S,k) **sat** $1 \leq k \leq |S| \rightarrow SL(Select',k,S)$

hvor SL er prædikatet

$$\text{SL(e,k,S): } |\{i \in 0..|S| \mid S.(i)<e\}| < k \leq |\{i \in 0..|S| \mid S.(i) \leq e\}|$$

Det er klart, at man kan løse problemet ved at sortere S og dernæst returnere S.(k-1). Dette giver imidlertid en udførelsestid på $\Theta(n \log(n))$, og vi er ude efter noget bedre.

Det viser sig, at man kan bruge samme grundlæggende idé som i Quicksort, dvs. udvælge et tilfældigt element e i S, opdele S's elementer efter e og så undersøge, hvor tæt delepunktet er på k. Nedenstående algoritme viser hvordan.

Proceduren Split gør lidt mere end opdel i Quicksort, idet den flytter rundt på elementerne i S, så de der er mindre end e står til venstre; de der er lig med e i midten; og de der er større end e står til højre. Den tilfredsstiller følgende specifikation

Split[S,m,s](e) **sat**
$(S'(0..m') < e = S'(m'..s') < S'(s'..|S|)) \wedge$
$(0 \leq m' \leq s' \leq |S|) \wedge \text{perm}(S,S')$

hvor perm(S,S′) betyder, at S′ er en permutation af S.

Algoritme: Selektion
Stimulans: S,k: $1 \leq k \leq |S|$
Respons : e: SL(e,k,S)
Metode :

```
Proc Select(S: Vector, k: Int) → (Int)
  (+ Var e, m, s: Int
     e := S.(random(0, |S|))
     Split [S, m, s] (e)
     if k ≤ m → return Select(S(0 .. m), k)
      | m < k ≤ s → return e
      | s < k → return Select(S(s .. |S|), k-s)
     fi
  +)
end Select
e := Select(S,k)
```

Med hensyn til udførelsestid er det nemt at se, at hvis uheldet er ude, opfører algoritmen sig nogenlunde lige som Quicksort, dvs.

$$T[\![\textbf{Proc}\ \text{Select}]\!](\text{n}) \in \Theta(\text{n}^2)$$

Omvendt kan man (igen i lighed med Quicksort) indse, at hvis alle elementer i S er forskellige, dvs. $m = s - 1$, så er den forventede udførelsestid løsning til ligningen

$$T_E[\![\textbf{Proc}\text{Select}]\!](\text{n}) \approx n + \frac{1}{n}(1 + \sum_{s=1}^{n} \max(T_E[\![\textbf{proc}\ \text{Select}]\!](s-1),\ T_E[\![\ \textbf{proc}\ \text{Select}]\!](n-s)))$$

Det kan, igen ved induktion efter n, vises at der findes en positiv konstant c så

$$T_E[\![\textbf{proc}\ \text{Select}]\!](n) \leq cn$$

dvs.

$$T_E[\![\textbf{proc}\ \text{Select}]\!](n) \in O(n)$$

5.6 Heltalsmultiplikation

Betragt som sidste eksempel følgende samling procedurer, der understøtter manipulation af lange (decimale) heltal.

De mest centrale er Plus, Minus; CMult; TDiv, TMult og TMod, som hhv. adderer og subtraherer to lange tal; multiplicerer et langt tal med et ciffer; dividerer, multiplicerer og tager modulus af et langt heltal med en 10'er potens. Procedurerne tilfredsstiller følgende specifikationer

$\text{Plus}(t_1, t_2)$	**sat**	$\text{Plus}' = t_1 + t_2$
$\text{Minus}(t_1, t_2)$	**sat**	$t_1 \geq t_2 \rightarrow \text{Minus}' = t_1 - t_2$
$\text{CMult}(t, d)$	**sat**	$\text{CMult}' = t * d$
$\text{TDiv}(t, p)$	**sat**	$\text{TDiv}' = t/10^p$
$\text{TMult}(t, p)$	**sat**	$\text{TMult}' = t * 10^p$
$\text{TMod}(t, p)$	**sat**	$\text{TMod}' = t\ \textbf{mod}\ 10^p$

```
(+ @"sequence.tri"

   Box N
       Sequence(Int)
   end N

   Type Tal = N'Seq

   Proc Konst(v: Vector) → (Tal)
       (+ Var t: Tal
           N'Con[t](v)
           return t
       +)
   end Konst

   Proc Plus(t1, t2: Tal) → (Tal)
       (+ Var t: Tal
           Var m, i, s1, s2: Int
           N'Init[t]
           s1, s2 := N'Len[t1], N'Len[t2]
           if s1 >s2 →
                   s1 :=: s2
                   t1 :=: t2
           fi
           i := s1
           do i ≠ s2 →
                   N'Rpush[t1](0)
                   i := i+1
           od (* t1 og t2 er lige lange *)
           m, i := 0, 0
           do i ≠ s2 →
                   m := m+N'Sel[t1](i)+N'Sel[t2](i)
                   N'Rpush[t](m mod 10)
                   m, i := m/10, i+1
           od
           if m>0 → N'Rpush[t](m) fi
           return t
       +)
   end Plus

   Proc Minus(t1, t2: Tal) → ( Tal) (* t2 forudsættes at være mindre end eller lig t1 *)
       (+ Var t: Tal
           Var i, s1, s2, s: Int
           s1, s2 := N'Len[t1], N'Len[t2]
           i := 0
           do i ≠ s2 →
                   N'Sel[t2](i) := -N'Sel[t2](i)
                   i := i+1
           od
           t := Plus(t1, t2)
           s := N'Len[t]
           i := 0
           do i ≠ s →
                   if N'Sel[t](i)<0 →
                           N'Sel[t](i) := N'Sel[t](i)+10
                           N'Sel[t](i+1) := N'Sel[t](i+1)-1
                   fi
                   i := i+1
           od
           return t
       +)
   end Minus

   Proc Length(t: Tal) → (Int)
       return N'Len[t]
   end Length
```

```
Proc CMult(t: Tal, d: Int) → (Tal)
    (+ Var i, s, m: Int
       s := N'Len[t]
       m, i := 0, 0
       do i ≠ s →
            m := m+N'Sel[t](i)*d
            N'Sel[t](i), m := m mod 10, m/10
            i := i+1
       od
       if m>0 → N'Rpush[t](m) fi
       return t
    +)
end CMult

Proc TDiv(t: Tal, p: Int) → (Tal)
    (+ Var i: Int
       i := 0
       do i ≠ p →
            N'Lpop[t]
            i := i+1
       od
       return t
    +)
end TDiv

Proc TMult(t: Tal, p: Int) → (Tal)
    (+ Var i: Int
       i := 0
       do i ≠ p →
            N'Rpush[t](0)
            i := i+1
       od
       i := N'Len[t]
       do i ≠ p →
            i := i-1
            N'Sel[t](i) := N'Sel[t](i-p)
       od
       do i ≠ 0 →
            i := i-1
            N'Sel[t](i) := 0
       od
       return t
    +)
end TMult

Proc TMod(t: Tal, p: Int) → (Tal)
    (+ Var i: Int
       Var r: Tal
       i := 0
       N'Init[r]
       do i ≠ p →
            N'Rpush[r](N'Sel[t](i))
            i := i+1
       od
       return r
    +)
end TMod

Proc Skriv(t: Tal)
    (+ Var i: Int
       i := N'Len[t]
       do i>0 →
            i := i-1
            write(N'Sel[t](i))
       od
       if i = N'Len[t] → write("0") fi
```

```
    +)
end Skriv

Proc Læs[t: Tal] (* Forudsætter at der læses en ikke-tom følge af cifre *)
    (+ Var x: Text
       Var i: Int
       read[x]
       i:=0
       do (i< | x |) ∧ (x.(i) = '0') → i:=i+1 od
       x:=x(i..| x |-1)
       N'Init[t]
       i:=| x |
       do i ≠ 0 →
             i:=i-1
             N'Rpush[t](ci(x.(i))-ci('0'))
       od
    +)
end Læs

Proc Mult₁(x, y: Tal) → (Tal)
    (+ Var t: Tal
       Var i: Int
       i:=0
       t:=Konst(Vector())
       do i ≠ Length(y) →
             t:=Plus(t, TMult(CMult(x, N'Sel[y](i)), i))
             i:=i+1
       od
       return t
    +)
end Mult₁

Proc Mult₂(x, y: Tal) → ( Tal)
    (+ Var t: Tal
       Var p, n: Int
       Var u, v, w, z: Tal
       if Length(x) ≤ Length(y) → n:=Length(x)
       | Length(x) ≥ Length(y) → n:=Length(y)
       fi
       if n ≤ 2 → return Mult₁(x, y) fi
       p:=n/2
       u, v:=TDiv(x, p), TMod(x, p)
       w, z:=TDiv(y, p), TMod(y, p)
       return Plus(TMult(Mult₂(u, w), 2*p),
              Plus(TMult(Plus(Mult₂(u, z), Mult₂(w, v)), p), Mult₂(v, z)))
    +)
end Mult₂

Proc Mult₃(x, y: Tal) → (Tal)
    (+ Var t: Tal
       Var p, n: Int
       Var u, v, w, z, r, uw, vz: Tal
       if Length(x) ≤ Length(y) → n:=Length(x)
       | Length(x) ≥ Length(y) → n:=Length(y)
       fi
       if n ≤ 2 → return Mult₁(x, y) fi
       p:=n/2
       u, v:=TDiv(x, p), TMod(x, p)
       w, z:=TDiv(y, p), TMod(y, p)
       r:=Mult₃(Plus(u, v), Plus(w, z))
       uw:=Mult₃(u, w)
       vz:=Mult₃(v, z)
       return Plus(Plus(TMult(uw, 2*p), TMult(Minus(Minus(r, uw), vz), p)), vz)
    +)
end Mult₃
```

```
    ≪Program≫
+)
```

Det skulle være nemt at se, at alle procedurerne har udførelsestid $O(n)$, hvor n er længden af argumenterne.

Vi ønsker nu at tilføje en procedure, der *multiplicerer* to tal. Følgende procedure anvender den sædvanlige skolealgoritme.

```
Proc Mult₁(x, y: Tal) → (Tal)
    (+ Var t: Tal
       Var i: Int
       i := 0
       t := Konst(Vector())
       do i ≠ Length(y) →
             t := Plus(t, TMult(CMult(x, N'Sel[y ](i)), i))
             i := i+1
       od
       return t
    +)
end Mult₁
```

Det skulle være klart, at

$$T[\![\textbf{proc } \mathrm{Mult}_1]\!](n, m) \in \Theta(n * m)$$

hvor n og m er længden af x og y.

Man kan konstruere en multiplikationsalgoritme v.hj.a. del-og-kombiner ved at opdele tallene x og y på følgende måde.

x :	u	v
y :	w	z
	$\lceil n/2 \rceil$	$\lfloor n/2 \rfloor$

Her er n maksimum af længderne af x og y, og $\lceil r \rceil$ ($\lfloor r \rfloor$) betegner det mindste (største) hele tal, der er større (mindre) end eller lig med r ($\lceil\ \rceil$ og $\lfloor\ \rfloor$ betegnes også hhv. som *loft* og *gulv*).

Hvis vi sætter $p = \lfloor n/2 \rfloor$, er det klart, at

$$\begin{aligned} x &= u * 10^p + v \\ y &= w * 10^p + z \end{aligned}$$

hvoraf

$$(+) \qquad x * y = u * w * 10^{2p} + (u * z + w * v) * 10^{p} + v * z$$

Dette giver umiddelbart anledning til følgende alternative procedure

```
Proc  Mult₂(x, y: Tal) → (Tal)
  (+ Var  t: Tal
     Var  p, n: Int
     Var  u, v, w, z: Tal
     if Length(x) ≤ Length(y) → n := Length(x)
      | Length(x) ≥ Length(y) → n := Length(y)
     fi
     if n ≤ 2 → return Mult₁(x, y) fi
     p := n/2
     u, v := TDiv(x, p), TMod(x, p)
     w, z := TDiv(y, p), TMod(y, p)
     return Plus(TMult(Mult₂(u, w), 2*p),
                 Plus(TMult(Plus(Mult₂(u, z), Mult₂(w, v)), p),
                 Mult₂(v, z)))
  +)
end  Mult₂
```

Hvis n igen betegner længden af det længste argument, får vi nu

$$T[\![\text{Mult}_2]\!](n) \approx n + 4 * T[\![\text{Mult}_2]\!](\frac{n}{2})$$

hvilket følger af, at (+) udtrykker multiplikation af to n-cifrede tal ved 4 multiplikationer af $n/2$-cifrede tal samt et antal lineære operationer (additioner og multiplikationer med 10'er potenser).

Løsning af denne ligning giver

$$T[\![\text{Mult}_2]\!](n) \in \Theta(n^2)$$

dvs. proceduren er ikke bedre end skolealgoritmen.

Nu kan vi imidlertid observere, at vi kan erstatte en multiplikation med to subtraktioner på følgende måde.

Hvis vi sætter

$$\begin{aligned} r &= (u + v) * (w + z) \\ &= u * w + (u * z + v * w) + v * z \end{aligned}$$

kan vi skrive ligningen (+) som

$$x * y = u * w * 10^{2p} + (r - u * w - v * z) * 10^{p} + v * z$$

og da vi kun behøver at udregne $u * w$ og $v * z$ én gang, er vi nede på tre multiplikationer af $n/2$-cifrede tal. Dette fører til følgende procedure

```
Proc Mult3(x, y: Tal) → (Tal)
  (+ Var t: Tal
     Var p, n: Int
     Var u, v, w, z, r, uw, vz: Tal
     if Length(x) ≤ Length(y) → n := Length(x)
      | Length(x) ≥ Length(y) → n := Length(y)
     fi
     if n ≤ 2 → return Mult1(x, y) fi
     p := n/2
     u, v := TDiv(x, p), TMod(x, p)
     w, z := TDiv(y, p), TMod(y, p)
     r := Mult3(Plus(u, v), Plus(w, z))
     uw := Mult3(u, w)
     vz := Mult3(v, z)
     return Plus(Plus(TMult(uw, 2*p),
               TMult(Minus(Minus(r, uw), vz), p)), vz)
  +)
end Mult3
```

hvis analyse giver

$$T[\![\textbf{proc}\ \mathrm{Mult}_3]\!](n) \approx n + 3 * T[\![\mathrm{Mult}_3]\!](n/2)$$

Denne ligning har løsningen

$$\begin{aligned}
T[\![\textbf{proc}\ \mathrm{Mult}_3]\!](n) &\approx n + 3(\frac{n}{2} + 3(\frac{n}{4} + \cdots + 3))\cdots) \\
&= n(1 + \frac{3}{2} + (\frac{3}{2})^2 + \cdots + (\frac{3}{2})^{\log(n)}) \\
&= n * \frac{1 - (\frac{3}{2})^{\log(n)+1}}{1 - \frac{3}{2}} \\
&= 2n((\frac{3}{2})^{\log(n)+1} - 1) \\
&\approx n * \frac{3^{\log(n)+1}}{2^{\log(n)+1}} \\
&\approx 3^{\log(n)+1} \\
&= n^{\log(3)} \\
&\approx n^{1.59}
\end{aligned}$$

Vi ser altså at del-og-kombiner, sammen med "tricket" med at spare en multiplikation fører til en asymptotisk forbedring i forhold til standard-metoden.

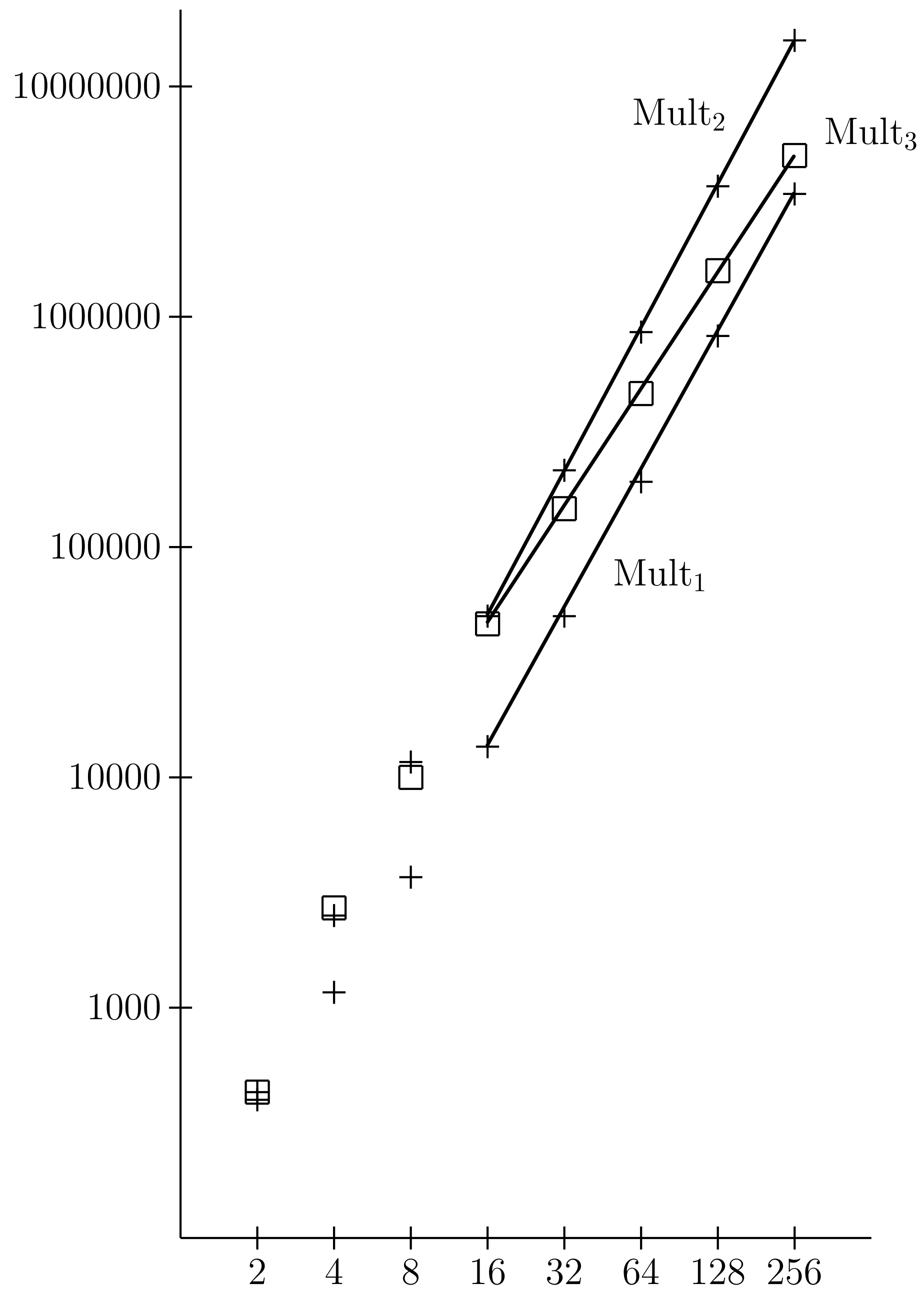
Nedenstående rette linier (tegnet i et dobbeltlogaritmisk hoordinatsystem) viser resultatet af et antal kørsler med de tre metoder. Da analysen kun gælder for store værdier af n, er de tre rette linier kun tilpasset punkter, hvis abcisse er større end eller lig med 16, svarende til multiplikation af 16-cifrede tal. De tre liniers hældningskoefficienter er som følger

$$\begin{array}{lcl} L_1 & : & 1.95 \\ L_2 & : & 2.01 \\ L_3 & : & 1.64 \end{array}$$

Hvis dette sammenlignes med de af analysen forudsagte værdier

$$\begin{array}{lcl} L_1 & : & 2.00 \\ L_2 & : & 2.00 \\ L_3 & : & 1.59 \end{array}$$

er der altså tale om ganske god overensstemmelse.


10000000
1000000
100000
10000
1000
2
4
8
16
32
64
128
256
Mult2
Mult3
Mult1

5.7 Sammenfatning

Del-og-kombiner er som nævnt i indledningen en generel algoritmisk problemløsningsteknik. Når man skal anvende skabelonen i en konkret situation, skal man fastlægge

- problemklassen P
- de simple problemer S
- hvordan man *deler* et problem i *mindre* delproblemer
- hvordan man *kombinerer* løsninger til delproblemer.

Her har man naturligvis en del frihed, men det er altid en god idé at søge at gøre delproblemerne disjunkte og nogenlunde lige store, samt at holde udgifterne til deling og kombination under kontrol, helst så de er $O(n)$. Dette skyldes, at ligningen for tidskompleksiteten af proceduren løs i skabelonen da får følgende udseende

$$\begin{aligned} T[\![\textbf{proc}\ \text{løs}]\!](1) &\approx 1 \\ T[\![\textbf{proc}\ \text{løs}]\!](n) &\approx n + \\ & k * T[\![\textbf{proc}\ \text{løs}]\!](\frac{n}{k}) + \\ & n \end{aligned}$$

hvor vi har antaget, at simple problemer har størrelse 1 og kan løses i konstant tid. Denne ligning kan løses på præcis samme måde som tidligere og resultatet bliver

$$T[\![\textbf{proc}\ \text{løs}]\!](n) \in \Theta(n \log(n))$$

Hvis man modsætningsvis kommer til at dele et problem i (f.eks. 2) problemer, hvoraf det ene er meget lille og det andet er meget stort, bliver resultatet en ligning af formen

$$\begin{aligned} T[\![\textbf{proc}\ \text{løs}]\!](n) &\approx n + \\ & T[\![\textbf{proc}\ \text{løs}]\!](1) + \\ & T[\![\textbf{proc}\ \text{løs}]\!](n-1) + \\ & n \end{aligned}$$

hvilket som bekendt giver

$$T[\![\textbf{proc}\ \text{løs}]\!](n) \in \Theta(n^2)$$

Sammenfattende kan det altså siges, at anvendelser af del-og-kombiner, hvor

- simple problemer løses i tid $O(1)$
- deling og kombination har tidskompleksitet $O(n)$
- delproblemerne er disjunkte og nogenlunde lige store

fører til effektive løsninger med tidskompleksitet $\Theta(n \log(n))$.

Der findes faktisk flere eksempler på problemer, hvor del-og-kombiner fører til *optimale* algoritmer, dvs. algoritmer der beviseligt er lige så gode som alle andre algoritmer, der løser det pågældende problem.

Vi kan betragte Binær søgning som en anvendelse af del-og-kombiner hvor opdelingen består i at dele søgedomænet i to lige store dele, og hvor man i kombinationen ignorerer den halvdel, målelementet ikke kan befinde sig i. Betragtet på denne måde fås følgende ligning

$$T[\![\text{Binær søgning}]\!](n) \approx 1 + T[\![\text{Binær søgning}]\!](\frac{n}{2})$$

som har løsningen

$$T[\![\text{Binær søgning}]\!](n) \in \Theta(\log(n))$$

Det er klart, at Binær søgning udelukkende baserer sig på *sammenligninger*, og man kan vise, at den er optimal blandt sammenligningssøgealgoritmer.

Situationen er den samme m.h.t. flettesortering. Det er ikke svært at se, at argumentet for at enhver sammenligningssøgning må udføre $\Omega(\log(n))$ sammenligninger [Harel, p. 146], kan udvides til at vise, at enhver sammenligningssorteringsalgoritme må udføre $\Omega(n \log(n))$ sammenligninger. Da flettesortering kun baserer sig på sammenligninger, er den derfor optimal inden for denne klasse.

Det er ofte sådan i del-og-kombiner algoritmer, at delingen og kombinationen er "omvendt proportionale" i den forstand, at hvis der "sker noget" i den ene, er den anden "triviel". Dette ses tydeligt i de to sorteringseksempler. I quicksort er situationen således

deling: Opdel

kombination: **skip**

og i flettesortering

deling: ≪del listen i to halvdele≫
kombination: Fletning

Sagt på en anden måde, laver Quicksort arbejdet på vejen "ned" i rekursionen, medens flettesortering gør det på vejen "tilbage". Der er mange eksempler på del-og-kombiner algoritmer, der udviser en tilsvarende asymmetri imellem deling og kombination.

Som afslutning på dette afsnit opsamler vi de forskellige løsninger til del-og-kombiner rekursionsligninger i følgende nyttige sætning.

Sætning
Rekursionsligningen

$$\begin{aligned} T(1) &\approx 1 \\ T(n) &\approx k * T(\frac{n}{d}) + n^p \text{ for } n > 1 \end{aligned}$$

hvor k, d og p er positive tal, har løsning

$$T(n) \in \begin{cases} \Theta(n^p) & \text{for} \quad k < d^p \\ \Theta(n^p \log(n)) & \text{for} \quad k = d^p \\ \Theta(n^{\log_d(k)}) & \text{for} \quad k > d^p \end{cases}$$

Bevis
Vi anvender den sædvanlige udfoldning og får

$$\begin{aligned} T(n) &\approx n^p + k * ((\frac{n}{d})^p + k * ((\frac{n}{d^2})^p + \cdots + k * 1)) \cdots) \\ &= n^p * (1 + \frac{k}{d^p} + (\frac{k}{d^p})^2 + \cdots + (\frac{k}{d^p})^{\log_d(n)}) \end{aligned}$$

Vi betragter nu de tre tilfælde.

$k < d^p$: Da $(\frac{k}{d^p}) < 1$ er den uendelige kvotientrække $\sum_{i=0}^{\infty}(\frac{k}{d^p})^i$ konvergent. Men så er den endelige delrække det også, dvs. $T(n) \approx n^p$.

$k = d^p$: Summen indeholder $\log_d(n)$ led, dvs. $T(n) \approx n^p * \log(n)$.

$k > d^p$: Vi bruger sumformelen for en kvotientrække (jfr. side 65) og får

$$
\begin{aligned}
T(n) &\approx n^p * \frac{1 - (\frac{k}{d^p})^{\log_d(n)+1}}{1 - (\frac{k}{d^p})} \\
&\approx n^p * (\frac{k}{d^p})^{\log_d(n)} \\
&= n^{\log_d(k)}
\end{aligned}
$$

□

Denne sætning giver nu direkte udførelsestiden for del-og -kombiner løsninger, hvor et (stort) problem deles op i k (små) problemer, som hvert er d gange så småt, og hvor opdelings- og kombinationstiderne er et polynomium i størrelsen af det store problem.

6 Dynamisk Programmering

Der er mange interessante problemer, der ikke umiddelbart lader sig løse v.hj.a. del-og-kombiner strategien på en sådan måde, at antallet af delproblemer er en passende (lille) konstant. Hvis man i sådanne tilfælde fristes til blot at generere så mange delproblemer som nødvendigt, ender man let med helt uacceptabelt ineffektive algoritmer. Dette illustreres af følgende vigtige *optimeringsproblem*, det såkaldte *Knapsackproblem*.

6.1 Knapsack

I dette problem er der givet et antal objekter $o_1, \ldots, o_n$, som hver har en størrelse $s_1, \ldots, s_n$ og en værdi $v_1, \ldots, v_n$. Problemet går ud på at pakke en rygsæk (engelsk: knapsack) af kapacitet C på en sådan måde, at indholdet har den størst mulige værdi (man kan vælge vilkårligt mange objekter af hver slags). En mere matematisk formulering af problemet er, at man skal maksimere summen

$$\sum_{i=1}^{n} x_i v_i$$

under randbetingelsen

$$\sum_{i=1}^{n} x_i s_i \leq C$$

hvor $s_1, \ldots, s_n$ er positive heltal og C, $v_1, \ldots, v_n$, $x_1, \ldots, x_n$ er ikke-negative heltal. Lad os betegne denne maksimale værdi med $MV(C)$.

Følgende rekursive algoritme løser problemet

Algoritme: Knapsack

Stimulans : $s_1, \ldots, s_n \geq 1; C, v_1, \ldots, v_n \geq 0$
Respons : $V_{opt} = MV(C)$
Metode : **proc** knap(c: Int) $\rightarrow$ (Int)

```
    (+ var j, mv: Int
        mv := 0
        for j := 1 to n do
            if s_j ≤ c → mv := max{mv, v_j+ knap(c − s_j)} fi
        return mv
    +)
end knap

V_opt := knap(C)
```

Proceduren virker som følger. Hvis kapaciteten c i et kald er mindre end størrelsen af det mindste objekt, min$\{s_j\}$, så er $MV(c)$ lig med 0. Ellers er $MV(c)$ lig med maximum af $v_j + MV(c - s_j)$ over de s_j'er for hvilke $s_j \leq c$, fordi vi ved at medtage et eksemplar af objektet o_j forøger værdien med v_j mod til gengæld at bruge plads s_j. Det følger heraf, at algoritmen er korrekt, hvis den ellers standser, men det gør den, fordi argumenterne til rekursive kald aftager (alle s_j'erne er positive).

Til gengæld er algoritmen næppe særlig effektiv, fordi det må forventes, at der udføres en hel række af kald knap(c) for samme værdi af c, dvs. at én gang udregnede værdier beregnes igen og igen. Dette bekræftes af følgende analyse (for nemheds skyld ser vi på det udartede tilfælde, hvor alle s_j'erne er lig med 1). I dette tilfælde fås

$$\begin{aligned} T[\![\textbf{proc}\text{ knap}]\!](0) &\approx O(n) \\ T[\![\text{ knap}]\!](c) &\approx T[\![\textbf{proc}\text{ knap}]\!](c) \\ &\approx n * T[\![\text{ knap}(c-1)]\!] \\ &\approx n * T[\![\textbf{proc}\text{ knap}]\!](c-1) \end{aligned}$$

som har løsningen

$$T[\![\textbf{proc}\text{ knap}]\!](c) \in \Theta(n^{c+1})$$

Der er altså tale om eksponentiel udførelsestid. Det kan indses (men regnerierne er lidt mere kompliceret), at dette også gælder, når s_j'erne er mere "normale".

Algoritmens udførelsestid kan imidlertid effektiviseres dramatisk ved at observere, at der ikke forekommer mere end $C + 1$ *forskellige* kald af proceduren knap, og at man derfor med fordel kan gå over til *genbrug*, dvs. gemme én gang udregnede værdier i en passende datastruktur, og så blot returnere disse næste gang, der er brug for dem. For Knapsack skal der bruges en vektor H(0..C), hvis elementer initialiseres til " at være ukendte", og hvor H.(c) sættes til $MV(c)$ første gang denne udregnes.

Disse overvejelser resulterer i følgende alternative algoritme – typen Huk (for hukommelse) er givet ved

```
Type Huk   = List(Celle)
Type Celle = Sum(kendt: Int, ukendt: Unit)
```

Algoritme: Tabel knapsack

Stimulans : $s_1, \ldots, s_n \geq 1; C, v_1, \ldots, v_n \geq 0$
Respons : $V_{opt} = MV(C)$
Metode :

```
proc Tknap[H: Huk](c: Int) → (Int)
   if is (H.(c),kendt) → skip
   |  is (H.(c),ukendt) → (+ Var j, mv: Int
                             mv := 0
                             for j := 1 to n do
                                if s_j ≤ c → mv :=
                                  max{mv, v_j + Tknap[H](c-s_j)}
                                fi
                             H.(c):= Celle(kendt: mv)
                           +)
   fi
   return H.(c).kendt
end Tknap

H := Huk(Celle(ukendt: #)|C + 1)
V_opt := Tknap[H](C)
```

Det skulle være klart, at Tabel knapsack er ækvivalent med Knapsack, og at den dermed er korrekt. Med hensyn til analysen af dens effektivitet kan vi bruge bogføringsteknikken fra s. 31 i [P&D] på følgende måde:

Det er klart, at hver gang der udføres et kald af formen Tknap[H](c), hvor H.(c) er kendt, koster kaldet kun 1 krone, fordi der blot er tale om et tabelopslag. Vi påstår nu, at algoritmen (næsten) kan finansieres i sin helhed, hvis vi fra begyndelsen deponerer $2n$ kroner i hvert element i H, dvs. ialt deponerer $2(C+1)n$ kroner.

Argumentet er et induktionsargument, som består i at vise, at det på et vilkårligt tidspunkt af udførelsen gælder, at hvis alle ukendte elementer i H besidder $2n$ kroner (de kendte elementer har ingen penge), så kan vi finansiere et kald af formen Tknap[H](c), hvor H.(c) er ukendt, (næsten) udelukkende v.hj.a. de midler, der befinder sig i H.

Til brug for beviset skal vi kalde tabellen H *velsitueret*, hvis der gælder følgende

is (H.(c),kendt) $\Rightarrow$ H.(c) = MV(c) og H.(c) har ingen penge
is (H.(c),ukendt)$\Rightarrow$ H.(c) besidder $2n$ kroner

Beviset føres ved induktion efter c med følgende induktionsantagelse

I(c): Hvis H er velsitueret, så er H stadig velsitueret efter kaldet Tknap[H](c), og udgiften til kaldet kan, pånær eventuelt 1 krone, betales af H selv.

Selve beviset går som følger

<u>Basis</u>

$c < min\{s_j\}$: Hvis H.(c) er kendt, bruger vi den ekstra krone til at betale for tabelopslaget. Hvis H.(c) er ukendt, udføres der ingen rekursive kald. Det fås derfor umiddelbart, at $T[\![\textbf{proc}\ \text{Tknap}]\!](c) \in O(n)$, og det kan rigeligt finansieres med de $2n$ kroner, der er deponeret i H.(c).

Induktionsskridt

Vi antager, at $I(c')$ er opfyldt for alle $c' < c$. Beviset for $I(c)$ går som følger.
Hvis H.(c) er kendt, bruger vi igen den ekstra krone til at betale for tabelopslaget. Ellers starter vi med at hæve de $2n$ kroner i H.(c). De n kroner dækker udgiften til udførelsen af procedurekroppen , og iflg. induktionsantagelsen kan de øvrige n kroner sammen med H selv finansiere de højst n rekursive kald. Dette *forudsætter*, at H er velsitueret ved indgangen til disse kald. Det er H imidlertid ikke, for vi har jo hævet pengene i H.(c), uden at ændre værdien af H.(c). Det gør imidlertid ikke noget, fordi der for samtlige rekursive kald Tknap[H](c′) gælder, at $c' < c$, og derfor kommer vi ikke under kaldet af Tknap[H](c) til at mangle de $2n$ kroner, der blev hævet.
Efter afslutningen af de rekursive kald sættes H.(c) lig den netop beregnede værdi MV(c), dvs. H bliver igen velsitueret.

Det følger af ovenstående, at

$$T[\![\text{Tabel knapsack}]\!](c) \in O(c * n)$$

hvilket er en meget konkret demonstration af forbedringen i forhold til $O(n^{c+1})$.

Bemærk, at vi her har set et eksempel på, at der på en og samme tid argumenteres for *korrekthed*, *terminering* og *kompleksitet* af en rekursiv procedure.

Det skal også bemærkes, at de her viste egenskaber for proceduren Tknap (naturligvis) også kunne have været vist v.hj.a. bevisprincippet for rekursive procedurer (jfr. [P&D]). Mere præcist kunne vi have vist, at følgende specifikation af Tknap er korrekt

$$
\begin{array}{l}
\text{Tknap[H](c) } \textbf{sat} \\
\quad (\text{velsitueret(H)} \rightarrow \\
\qquad (\text{velsitueret(H}')) \wedge \textbf{ is } (\text{H}'.\text{(c),kendt}) \wedge \\
\qquad (\text{Tknap}' = \text{MV(c)}) \wedge \\
\qquad (\text{T}[\![\textbf{proc} \text{ Tknap}]\!](\text{c}) \in \text{O}(\Phi(\text{H}) - \Phi(\text{H}') + 1)
\end{array}
$$

hvor Φ er en potentialfunktion, der angiver “H’s rigdom”, dvs.

$$\Phi(\mathrm{H}) = \alpha n * |\{c \mid \textbf{ is } (\mathrm{H.(c),ukendt})\}|$$

hvor α er en passende konstant (jfr. [P&D]).

6.2 Skabelonen

Vi kan også beskrive dynamisk programmeringsteknikken lidt mere abstrakt i form af en skabelon. Vi bruger følgende modifikation af del-og-kombiner skabelonen, hvor vi udover begreberne fra dennes univers skal bruge en hukommelse

$$H : P \to R$$

som er en funktion, der til hvert problem i problemklassen P knytter dets løsning fra L (eller er udefineret). Som R kan vi bruge typen

Sum (kendt: L, ukendt: Unit)

Skabelonen ser ud som følger

Skabelon: Dynamisk Programmering

Univers : Universet fra del-og-kombiner skabelonen udvidet med funktionen $H : P \to R$

Stimulans: $p : P$, problem der ønskes løst

Respons : $l : L$, løsning af p

Metode :

```
proc Tløs[H: Huk, p : P, l : L]
  if is (H(p),kendt) → l := H(p).kendt
  & true → if ≪ p er i S ≫ →
               ≪find løsning l direkte≫
           & true →
               (+ Var p1, . . . , pk : P
                  Var l1, . . . , lk : L
                  ≪del p i p1, . . . , pk ≫
                  Tløs[p1, l1]
                  ⋮
                  Tløs[pk, lk]
                  ≪kombiner l1, . . . , lk til l ≫
               +)
           fi
           H(p) := R(kendt: l)
  fi
end Tløs

≪sæt H(q) til R(ukendt: #) for alle q der er mindre end p ≫
Tløs[H, p, l]
```

Vi kan nu finde udførelsestiden T⟦**Proc** Tløs⟧(k, s), hvor k er antallet af små problemer et stort problem opdeles i, og hvor s er lig med det samlede antal problemer, der er mindre end p. Vi skal antage, at der for de ubestemte stumper i skabelonen gælder

$$T[\![\ll \text{p er i S} \gg]\!] \in O(k)$$
$$T[\![\ll \text{find løsning l direkte} \gg]\!] \in O(k)$$
$$T[\![\ll \text{del p i } p_1, \ldots, p_k \gg]\!] \in O(k)$$
$$T[\![\ll \text{kombiner } l_1 \ldots l_k \text{ til l} \gg]\!] \in O(k)$$

Analysen er identisk med den, der blev gennemført i forrige afsnit og giver

følgende udførelsestid

$$\mathrm{T}[\![\textbf{Proc}\ \mathrm{Tløs}]\!](k, s) \in O(k * s)$$

At dette passer med analysen af Tabel knapsack følger af, at vi i det tilfælde havde $k = n$ og $s = c$.

6.3 Binomialkoefficienter

At princippet bag dynamisk programmering – introduktion af en tabel til at indeholde resultatet af én gang beregnede værdier – er velkendt fra andre sammenhænge, illustreres på udmærket vis af algoritmer til beregning af *binomialkoefficienter*.

Binomialkoefficienten $\binom{n}{m}$ angiver som bekendt antallet af måder, hvorpå man kan udtage m elementer fra en mængde med n elementer. Da såvel den tomme mængde som hele mængden kun kan udtages på én måde, er det klart, at der for alle $n \geq 0$ gælder

$$\binom{n}{0} = \binom{n}{n} = 1$$

Det er også let at se, at man for $0 < m < n$ kan udtrykke $\binom{n}{m}$ ved

$$\binom{n}{m} = \binom{n-1}{m} + \binom{n-1}{m-1}$$

Dette skyldes, at antallet af måder hvorpå man kan udtage m elementer (naturligvis) er lig med antallet af forskellige delmængder, der indeholder præcis m elementer (lad os kalde en sådan for en *m-delmængde*). Betragt nu et tilfældigt element e. Antallet af m-delmængder er lig med antallet af m-delmængder hvor *e er med*, plus antallet af m-delmængder, hvor *e ikke er med*. Det første af disse antal er

$$\binom{n-1}{m-1}$$

fordi der skal vælges $m - 1$ ud af de resterende $n - 1$ (e skal jo være med), og det andet er

$$\binom{n-1}{m}$$

fordi der nu skal vælges m ud af de resterende $n-1$ (e skal jo ikke være med).

Sammenfattende har vi altså for $n \geq 0$

$$\binom{n}{m} = \begin{cases} 1 & \text{for } m = 0 \\ \binom{n-1}{m-1} + \binom{n-1}{m} & \text{for } 0 < m < n \\ 1 & \text{for } m = n \end{cases}$$

Denne definition fører umiddelbart til følgende procedure

```
Proc Bin(n, m: Int) → (Int)
    if (m < 0) ∨ (n < m) → abort("Forkerte argumenter")
    &(m = 0) ∨ (n = m) → return 1
    & true → return Bin(n-1, m)+Bin(n-1, m-1)
    fi
end bin
```

Da værdierne i "bunden" af rekursionen alle er lig med 1, er det klart, at der ved et kald af formen $Bin(n,m)$ udføres $\binom{n}{m} - 1$ additioner, dvs. der gælder

$$T[\![Bin(n,m)]\!] \in \Omega(\binom{n}{m})$$

Nu vides det, at for $m = n/2$ er

$$\binom{n}{m} \geq \frac{2^n}{n+1}$$

hvorfor vi igen står med en håbløs procedure.

Imidlertid kan det igen observeres, at procedure Bin højst kaldes med $(n+1)(m+1)$ forskellige parametre. Vi kan derfor introducere en tabel på præcis samme måde som før, og får da følgende program

```
Process Binomial
    (+ Type Huk = List(Række)
       Type Række = List(Celle)
       Type Celle = Sum(kendt: Int, ukendt: Unit)

       Proc Tbin[H: Huk](n, m: Int) → (Int)
           if (m<0) ∨ (n<m) → abort("Forkerte argumenter")
           & true →
                if is(H.(n, m), ukendt) →
                     (+ Var r: Int
                        if (m = 0) ∨ (n = m) → r := 1
                        & true → r := Tbin[H](n-1, m)+Tbin[H](n-1, m-1)
                        fi
                        H.(n, m) := Celle(kendt: r)
                     +)
                fi
                return H.(n, m).kendt
           fi
       end Tbin

       Var n, m: Int

       ≪Indlæs n og m≫
       do n ≥ 0 →
           (+ Var H: Huk
              H := Huk(Række(Celle(ukendt: #) | m+1) | n+1)
              write("Tbin(n,m)= ", Tbin[H](n, m), eol)
           +)
           ≪Indlæs n og m≫
       od
    +)
end Binomial
where ≪Indlæs n og m≫ is
           write("n= ")
           read[n]
           write("m= ")
           read[m]
```

Dette er en realisering af skabelonen, hvor $k = 2$ og $s = n * m$. Udførelsestiden bliver derfor

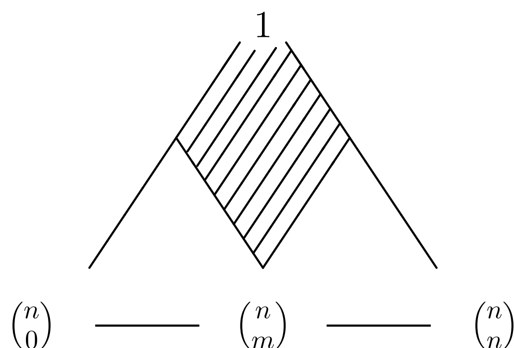
$$T[\![Tbin(n, m)]\!] \in \Theta(n * m)$$

Den rolle tabellen spiller her kan illustreres v.hj.a. den såkaldte *Pascals Trekant*, som er betegnelsen for følgende opstilling af binomialkoefficienterne

```
            1
          1   1
        1   2   1
      1   3   3   1
    1   4   6   4   1
  1   5   10  10  5   1
1   6   15  20  15  6   1
```

hvor hver række indeholder samtlige binomialkoefficienter for en bestemt værdi af n, mere præcist indeholder den n'te række koefficienterne $\binom{n}{m}$ for $0 \leq m \leq n$. Trekanten er konstrueret ud fra den rekursive definition af binomialkoefficienterne, idet hvert element fremkommer som summen af de to elementer, der står lige oven over.

Ved at studere programmet Binomial nøjere, kan man indse, at den del af tabellen H, der udfyldes af proceduren `Tbin`, er lig med følgende skraverede del af Pascals Trekant:



Det er klart, at en alternativ metode til beregning af $\binom{n}{m}$ er følgende iterative algoritme, som beregner hele Pascals Trekant.

Algoritme: Pascals Trekant

Stimulans : $n, m : n, m \geq 0$

Respons : $r = \binom{n}{m}$

Metode : $T, i :=$ List(Vector(1)), 1

do $\{T$ indeholder rækkerne $0..i$ i Pascals trekant$\}$

$i \leq n \rightarrow T, j := T ++$ List(Vector($1|i+1$)), 1

do $j \neq i \rightarrow T.(i,j) := T.(i-1, j-1) + T.(i-1, j)$

$j := j + 1$

od

$i := i + 1$

od

$r := T.(n, m)$

Det er naturligvis klart, at vi igen har

$$T[\![\text{Pascals Trekant}]\!](n, m) \in \Theta(n * m)$$

Dette eksempel illustrerer en anden generel pointe vedrørende dynamisk programmering. Det er nemt at se, at man i stedet for at tilføje en tabel H til en rekursiv procedure, hvor H udfyldes i overensstemmelse med denne procedures opførsel, ligeså godt kunne skrive et *iterativt* program, der udfylder H systematisk fra bunden af i lighed med Pascals Trekant. Denne alternative formulering af dynamisk programmering er den, der oftest mødes i litteraturen. Vi skal imidlertid tænke på dynamisk programmering som “at smide en tabel efter en rekursiv procedure”, fordi dette normalt er mere intuitivt og lettere at forstå.

7 Kombinatorisk søgning

Som det fremgik af forrige kapitel, kan man bruge dynamisk programmering til at løse et problem p, såfremt det *totale antal problemer, der er mindre end* p, ikke er for stort. Hermed menes, at det ikke må være større, end at man kan introducere en hukommelsesfunktion H, hvis domæne har denne størrelse. Dette betyder i praksis, at antallet af problemer der er mindre end p (som er en øvre grænse for antallet af forskellige kald i den rekursive løsning) skal være et polynimium af lav grad i p's størrelse.

Der findes imidlertid mange problemer, hvor dette ikke er tilfældet, og hvor man følgelig ikke kan klare sig med den ligefremme systematik, der præger dynamisk programmering (eller del-og-kombiner). I sådanne situationer er man henvist til at vælge de delproblemer, man løser – og rækkefølgen hvori man løser dem – med større omtanke. Kombinatorisk søgning dækker over en sådan problemløsningsteknik, hvor man udvælger delproblemer v.hj.a. en prioritetsmekanisme, som styres af en prioritetskø. Vi illustrerer teknikken v.hj.a. to eksempler, det såkaldte *0-1 Knapsack Problem* (som er en variant af Knapsack problemet fra sidste kapitel) og det såkaldte *Dronninge Problem*, som er en klassiker inden for kombinatoriske søgeproblemer.

7.1 0-1 Knapsack

Vi betragter igen Knapsack problemet fra kapitel 6, men denne gang med den yderligere restriktion, at der højst må medtages ét objekt af hver slags. Sagt med andre ord, skal vi finde maksimum for

$$\sum_{i=1}^{n} x_i v_i$$

under randbetingelserne

$$\begin{array}{l} x_i \in \{0,1\} \\ \sum_{i=1}^{n} x_i s_i \leq C \end{array}$$

En oplagt løsning er følgende modifikation af algoritme Knapsack s. 73, hvor parameteren M er en delmængde af $\{1, 2, \ldots, n\}$.

Algoritme: 0-1 Knapsack

Stimulans : $s_1, \ldots, s_n \geq 1; C, v_1, \ldots, v_n \geq 0$
Respons : $V_{opt} = MV(C)$
Metode : **proc** Knap01(M: Mængde, c: Int) $\rightarrow$ (Int)
(+ **var** j, mv: Int
$mv := 0$
for $j : j \notin M$ **do**
if $s_j \leq c \rightarrow$
$mv := max\{mv, v_j +$ Knap01$(M \cup \{j\}, c - s_j)\}$ **fi**
return mv
+)
end Knap01

$V_{opt} :=$ Knap01$(\emptyset, C)$

Hvis vi med m betegner antallet af objekter, der *ikke* er med i M, dvs. $m = n - |M|$ får vi

$$T[\![\textbf{proc}\ \text{Knap01}]\!](m, c) \approx m * T[\![\textbf{proc}\ \text{Knap01}]\!](m-1, c-1)$$

hvoraf (idet vi gør den rimelige antagelse, at $n \leq C$)

$$T[\![\textbf{proc}\ \text{Knap01}]\!](n, C) \in O(n!)$$

hvilket igen er en uantagelig udførelsestid.

Hvis vi nu prøver at bruge skabelonen for dynamisk programmering, kan vi se, at problemklassen P består af *par* af formen (M, c), hvor M angiver de objekter, vi har udvalgt, og c er rest-kapaciteten. Nu udgøres mængden af problemer, der er mindre end (M, c), af de par (M', c') for hvilke der gælder

$$M' \supseteq M \text{ og } c' < c$$

Men da startproblemet er parret $(\emptyset, C)$ følger det, at det totale antal delproblemer er af størrelsesordenen $C * 2^n$, hvilket selv for moderate værdier af n (og C) umuliggør anvendelse af dynamisk programmering. Vi er derfor henvist til at "prøve os frem med omtanke", og til den ende er det nyttigt med følgende reformulering af 0-1 Knapsack problemet.

Betragt transitionssystemet

Transitionssystem: 0-1 Knapsack

Konfigurationer : $\{[M, c] \mid M \subseteq \{1, 2, \ldots, n\}, 0 \leq c \leq C\}$
Transitioner : $[M, c] \triangleright [M \cup \{i\}, c - s_i]$ **hvis** $(i \notin M) \ \wedge \ (s_i \leq c)$

Hvis vi definerer værdien af mængden M som summen af værdierne af de objekter den "indeholder", dvs.

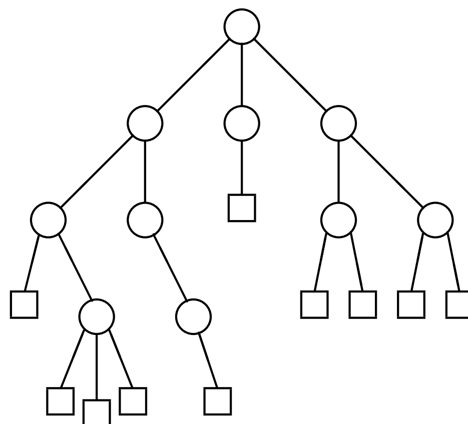
$$V(M) = \sum_{i \in M} v_i$$

er det klart, at 0-1 Knapsack problemet går ud på at finde den proces for transitionssystemet

$$[\emptyset, C] \triangleright \ldots \triangleright [M, c]$$

hvor værdien af M i slutkonfigurationen er maksimal.

Teknikken i kombinatorisk søgning består nu i at lede efter en sådan optimal proces for transitionssystemet ved på ethvert tidspunkt at udvælge den konfiguration, der er mest "lovende", dvs. som ser ud til at være den næste i den optimale proces. Det er klart, at vi kan repræsentere mængden af følger, der starter i startkonfigurationen $[\emptyset, C]$ v.hj.a. et *konfigurationstræ* på følgende måde



Her er roden startkonfigurationen $[\emptyset, C]$ og for enhver knude k gælder, at enten er k et blad, eller også har k en datter k' for hver konfiguration k', for hvilken $k \triangleright k'$.

Det følger, at ikke alene er hver vej fra roden til et blad en følge for transitionssystemet, men der er også en af disse der kan forlænges til den optimale

proces vi leder efter. Ideen er nu, at vi på ethvert tidspunkt holder styr på bladene i konfigurationstræet og i hvert skridt vælger at *ekspandere* det blad, der i en eller anden forstand er mest lovende.

Vi har altså brug for en datastruktur, hvis elementer udgøres af træets blade og som på et vilkårligt tidspunkt kan levere det "bedste" blad. En datastruktur med disse egenskaber er en prioritetskø, hvor prioriteten af et element (et blad) er udtryk for elementets (bladets) "godhed".

Vi kan nu skrive følgende abstrakte skitse til en løsning. Q er en prioritetskø, hvis mere detaljerede indretning vi skal se på i det følgende.

Algoritme: Skitse til kombinatorisk 0-1 Knapsack

Stimulans : $s_1, \ldots, s_n \geq 1; C, v_1, \ldots, v_n \geq 0$
Respons : $V_{opt} = MV(C)$
Metode : init$[Q]$
  insert$[Q]([\emptyset, C])$
  $V_{opt} := 0$
  do $\neg$ empty$[Q] \rightarrow$ delbest$[Q, k]$
    $V_{opt} := \max\{V_{opt}, \text{Værdi}(k)\}$
    for $k' : k \triangleright k'$ **do**
      insert$[Q](k')$
    od
  od

Det skulle være nemt at indse, at denne algoritme er korrekt (den fungerer faktisk ligesom den rekursive løsning). Den lider imidlertid af den skavank, at én og samme konfiguration kan blive indsat i prioritetskøen flere gange. Dette skyldes, at transitionssystemet har den egenskab, at de fleste konfigurationer kan nås på mange måder fra startkonfigurationen. Problemet kan klares på flere måder. Én er at indføre en mekanisme, der husker hvilke konfigurationer der allerede er set, men så risikerer man igen at pladskravet bliver for stort.

En bedre metode er at generere konfigurationerne mere systematisk. Det kan gøres ved at finde et andet transitionssystem, hvor en konfiguration kun optræder én gang i konfigurationstræet.

Følgende transitionssystem har denne egenskab.

Transitionssystem: 0-1 Knapsack

Konfigurationer : $\{[M,i,c] \mid M \subseteq \{1,2,\ldots,i\}, 0 \le i \le n, 0 \le c \le C\}$

Transitioner : $[M,i,c] \rhd [M,i+1,c]$ **hvis** $i < n$

$[M,i,c] \rhd [M \cup \{i+1\}, i+1, c - s_{i+1}]$

hvis $(i < n) \ \wedge \ (s_{i+1} \le c)$

Konfigurationerne $[M,i,c]$ indeholder nu parameteren i, som angiver nummeret på det objekt, der sidst er betragtet – og de to transitioner angiver muligheden for hhv. at udelukke og medtage det $(i+1)$'te objekt i M.

Vi er nu interesseret i optimale processer af formen

$$[\emptyset, 0, C] \rhd \ldots \rhd [M, n, c]$$

og de kan findes på samme måde som i skitsen ovenfor. Denne skitse er imidlertid for "liberal" med hensyn til hvilke konfigurationer den indsætter i prioritetskøen. Man bør kun indsætte konfigurationer, der har mulighed for at føre til den søgte optimale proces, dvs. blindgyder skal elimineres så tidligt som muligt. Afskæring af blindgyder sker v.hj.a. en såkaldt *afskæringsmekanisme*, og det er den anden karakteristiske egenskab ved kombinatorisk søgning. Principielt udmønter den sig i at sætningen

for $k' : k \rhd k'$ **do**
    insert$[Q](k')$
od

erstattes med

for $k' : k \rhd k'$ **do**
    if $\ll k'$ er lovende$\gg \rightarrow$
            insert$[Q](k')$
    fi
od

hvor afskæringsmekanismen, her repræsenteret ved

$$\ll k' \text{ er lovende} \gg$$

udtrykker, at det kun er de “interessante” efterfølgere vi genererer. Det er konkretiseringen af $\ll k'$ er lovende $\gg$ som, sammen med prioritetsmekanismen, bestemmer effektiviteten af en kombinatorisk søgealgoritme.

Vi skal nu fastlægge disse to parametre for vores konkrete algoritme (da vi taler om et *maksimerings*problem, vil vores prioritetskø levere det element der har den *største* prioritet).

Vi definerer prioriteten af en konfiguration $[M, i, c]$ som den største værdi M potentielt kan udvides til uden hensyntagen til kapacitetsbegrænsninger, dvs. vi definerer

$$prt([M, i, c]) = V(M) + \sum_{i<j\leq n} v_j$$

Vi kalder endvidere en konfiguration lovende hvis dens potentiel (som altså i dette tilfælde er lig med dens prioritet) er større end det hidtil fundne maksimum. Dette giver anledning til følgende algoritme, hvor vi bruger en prioritetskø af “type” PolyPri(Int), dvs. en prioritetskø der indeholder proceduren

$$\textbf{proc} \text{ DeleteBest}[Q : \text{Pkø}, x : E, p : \text{Int}]$$

som fjerner og returnerer det “bedste” element x i Q og som samtidig angiver dettes prioritet p.

Algoritme: Kombinatorisk 0-1 Knapsack

Stimulans : $s_1, \ldots, s_n \geq 1; C, v_1, \ldots, v_n \geq 0$
Respons : $V_{opt} = MV(C)$
Metode : Init[Q, false]
Insert[Q]($[\emptyset, 0, C], \sum_{i=1}^{n} v_i$)
$V_{opt} := 0$
do $\neg$ empty[Q] $\rightarrow$ DeleteBest[Q, k, p]
if $k.i = n \rightarrow V_{opt} := max\{V_{opt}, p\}$
& true $\rightarrow$ **for** $k' : k \rhd k'$ **do**
if $prt(k') > V_{opt} \rightarrow$
Insert[Q]($k', prt(k')$)
fi
od
fi
od

Med hensyn til algoritmens effektivitet, kan man godt risikere at komme til at generere en konfiguration for hver eneste delmængde af $\{1, 2, \ldots, n\}$, dvs. udførelsestiden bliver i værste fald eksponentiel i n. Der er imidlertid gode chancer for at den såkaldte *heuristik*, som afskæringsmekanismen og prioritetsmekanismen repræsenterer, kan begrænse omkostningerne i praktiske tilfælde.

Følgende program, der nøje følger algoritmen, bruger en instans af den polymorfe prioritetskø PolyPri side 55 i [P&D]. Som det fremgår af initialiseringen PQ'Init[Q](n^3, *false*) er det *bedste* element det *største* element. Bemærk i øvrigt, at da hver konfiguration i transitionssystemet højst har to efterfølgere, kan **for**-sætningen

for $k' : k \rhd k'$ **do**
$\ll$Håndter k' $\gg$
od

realiseres som

$\ll$Håndter k'_1 $\gg$
$\ll$Håndter k'_2 $\gg$

hvor k'_1 og k'_2 er de to efterfølgere for k.

```
Process Knapsack
    (+ @"PolyPri.tri"

        Type Sel = List(Bool)
        Type Config = Prod(M: Sel, i, c: Int)

        Box PQ
            PolyPri(Config)
        end PQ

        Var Q: PQ'Queue
        Var C, Vopt, n, Vsum: Int
        Var Kopt: Config
        Var S, V: Vector

        ≪Indlæs n, C, S og V≫
        ≪Beregn Vsum≫
        PQ'Init[Q](n*n*n, false)
        PQ'Insert[Q](Config(Sel(false | n), 0, C), Vsum)
        Vopt := 0
        do ¬ PQ'Empty[Q] →
            (+ Var k: Config
               Var p: Int
               PQ'DeleteBest[Q, k, p]
               if k.i = n →
                    if p>Vopt → Vopt, Kopt := p, k fi
               & true →
                    (+ Var ki: Int
                       Var kny: Config
                       ki := k.i
                       if (S.(ki) ≤ k.c) ∧ (p>Vopt) →
                            kny := k
                            kny.M.(ki), kny.i, kny.c := true, ki+1, k.c−S.(ki)
                            PQ'Insert[Q](kny, p)
                       fi
                       if p−V.(ki)>Vopt →
                            kny := k
                            kny.i := ki+1
                            PQ'Insert[Q](kny, p−V.(ki))
                       fi
                    +)
               fi
            +)
        od
        write("Løsning: ", Kopt, eol)
        write("Værdi: ", Vopt, eol)
    +)
end Knapsack
where ≪Indlæs n, C, S og V≫ is
            write("Indlæs antal objekter: ")
            read[n]
            write("Indlæs kapacitet: ")
            read[C]
            write("Indlæs størrelser: ")
            read[S]
            write("Indlæs værdier: ")
            read[V]

where ≪Beregn Vsum≫ is
            (+ Var i: Int
               i, Vsum := 0, 0
               do i ≠ n → i, Vsum := i+1, Vsum+V.(i) od
            +)
```

7.2 Skabelonen

Vi kan også angive en skabelon for kombinatorisk søgning.

Kravene til afskærings- og sammenligningsmekanismerne er angivet indirekte ved, at de skal sørge for gyldigheden af en invariant, der udtrykker at

- alle konfigurationer i Q kan nås fra startkonfigurationen k_0
- enhver konfiguration, der kan nås fra k_0, og som er bedre end det aktuelle bud på den optimale konfiguration, kan også nås fra en konfiguration i Q.

Dette er et præcist udtryk for, at prioritetskøen indeholder alle de "interessante" konfigurationer.

Skabelon: Kombinatorisk Søgning

Univers : $S = (K, T)$ er et gentagelsesfrit transitionssystem, hvori alle processer er endelige. $prt : K \to I\!R$ og $val : K \to I\!R$ er konfigurationernes *prioritet* og *værdi*. Q er en prioritetskø med prt som prioritetsmekanisme. $\ll k$ er lovende$\gg$ og $\ll k_1$ er bedre end $k_2 \gg$ er en afskærings- og sammenligningsmekanisme, der gør algoritmen gyldig. Invarianten er

$$I(Q, k_0)\text{: } (\forall \bar{k} \in Q : k_0 \overset{*}{\triangleright} \bar{k}) \wedge$$
$$\forall k \in K\text{:}(k_0 \overset{*}{\triangleright} k \wedge \ll k \text{ er bedre end } k_{opt} \gg) \Rightarrow$$
$$\exists \bar{k} \in Q : \bar{k} \overset{*}{\triangleright} k$$

Stimulans: k_0: startkonfiguration

Respons : $k_{opt}, V_{opt} : (V_{opt} = val(k_{opt})) \wedge$
$$\neg(\exists k \in K : k_0 \overset{*}{\triangleright} k \wedge \ll k \text{ er bedre end } k_{opt} \gg)$$

```
Metode  : Init[Q]
          Insert[Q](k_0, prt(k_0))
          k_opt, V_opt := k_0, val(k_0)
          do {I(Q, k_0)}
            ¬ Empty[Q] →
                DeleteBest[Q, k, p]
                if ≪ k er død≫ ∧ ≪ k er bedre end k_opt ≫→
                      k_opt, V_opt := k, val(k)
                & true → for k' : k ▷ k' do
                            if ≪ k' er lovende≫→
                                    Insert[Q](k', prt(k'))
                            fi
                         od
                fi
          od
```

Når man skal anvende denne skabelon i en konkret situation, skal man naturligvis først fastlægge transitionssystemet. Dernæst vælges de fire "parametre"

val
prt
$\ll k$ er lovende$\gg$
$\ll k_1$ er bedre end $k_2 \gg$

så hensigtsmæssigt som muligt for det givne problem. Valget af $\ll k_1$ er bedre end $k_2 \gg$ giver normalt sig selv, afhængigt af om der er tale om et maksimerings- eller minimeringsproblem. Naturen af val er som regel givet ved problemformuleringen, så den egentlige udfordring ligger i valget af, og samspillet mellem, prioritets- og afskæringsmekanismerne. (I nogle tilfælde, f.eks. det skal vi se i næste afsnit, er man kun interesseret i at finde en død konfiguration, og så kan man helt ignorere værdifunktionen.)

Valget af prioritets- og afskæringsmekanisme kan der ikke siges ret meget om i almindelighed. Det er et spørgsmål om erfaring og intuition og skal overvejes i hver enkelt konkret situation.

7.3 Dronningeproblemet

I dette afsnit betragtes et klassisk kombinatorisk problem, som kun kan løses v.hj.a. kombinatorisk søgning. Problemet hører til i “legetøjskategorien”, men det er velegnet til at illustrere en konkretisering af skabelonerne, hvor værdier ikke spiller nogen rolle. Problemet går ud på at finde en måde at placere 8 dronninger på et skakbræt så ingen af dronningerne er i *konflikt* med (dvs. kan slå) nogen af de andre. To dronninger må altså ikke stå i samme søjle, række eller diagonal. Følgende er et eksempel på en løsning.

*							
				*			
							*
					*		
		*					
						*	
	*						
			*				

I det følgende udvikler vi et program, der kan finde en (alle) løsning(er) til det generaliserede dronningeproblem, der består i at betragte skakbrætter af vilkårlig størrelse. Fremgangsmåden er den samme som i sidste afsnit, dvs. vi definerer først et passende transitionssystem og konstruerer derefter programmet v.hj.a. skabelonen.

Konfigurationerne i transitionssystemet beskriver dronningernes positioner på skakbrættet. Vi skal med det samme gøre transitionssystemet gentagelsesfrit, hvilket opnås ved at placere dronningerne række for række startende fra toppen. En konfiguration beskriver således en situation, hvor et antal af de øverste rækker alle indeholder en dronning, mens de resterende rækker er tomme. Vi kan derfor lade konfigurationerne være følger af tal, der angiver positionerne for dronningerne i de øverste rækker. Følgende situation på et 8×8 bræt

*							
			*				
	*						
				*			

beskrives således af talfølgen

$$(0, 3, 1, 4)$$

hvor det i'te element indeholder positionen for dronningen i den i'te række (bemærk at nummereringen starter med 0).

Givet en konfiguration k kan denne nu udvides til konfigurationen $k \cdot p$, hvor $0 \leq p < n$, forudsat at position p i den første tomme række ikke er i konflikt med nogen af dronningerne i k. Vi definerer prædikatet $\text{fred}(k, p)$ ved

$\text{fred}(k, p)$: en dronning i position p i række $|k|$ er ikke i konflikt med nogen dronning i k

Vi bemærker, at betydningen af dette prædikat jo defineres præcist v.hj.a. reglerne for skak.

Følgende transitionssystem er nu relevant.

Transitionssystem: Dronninger på $n \times n$ bræt	
Konfigurationer	: $\{[k] \mid k \in N_0^* \wedge |k| \leq n\}$
Transitioner	: $[k] \triangleright [k \cdot p]$ **hvis** $(|k| < n) \wedge \text{fred}(k, p)$

Det skulle være klart, at hvis

$$[\lambda] \triangleright \ldots \triangleright [k]$$

er en proces, så er $|k| = n$, dvs. k repræsenterer en løsning til problemet.

Konstruktionen af programmet foregår nu v.hj.a. skabelonen. Som prioritetsmekanisme bruger vi konfigurationens *længde*, dvs. vi videreudvikler

først de konfigurationer, der allerede indeholder mange dronninger. Da vi kun er interesseret i at finde døde konfigurationer, har vi ingen brug for skabelonens værdifunktion, dvs. den kan ignoreres.

Følgende program finder en enkelt løsning til $n \times n$ problemet (et program der finder alle løsninger opnås ved blot at fjerne **stop** sætningen). For simpelheds skyld udskrives løsningen(erne) straks de findes.

```
Process Dronninger
    (+ @"PolyPri.tri"
        Type Config = Vector
        Box PQ
            PolyPri(Config)
        end PQ
        Var Q: PQ'Queue
        Var n: Int
        Var løsning: Bool
        write("Indlæs antal dronninger: ")
        read[n]
        løsning := false
        PQ'Init[Q](n*n*n, false)
        PQ'Insert[Q](Config(), 0)
        do ¬ PQ'Empty[Q] →
            (+ Var k: Config
               Var p: Int
               PQ'DeleteBest[Q, k, p]
               if | k | = n →
                     write("Løsning: ", k, eol)
                     løsning := true
                     stop
               & true →
                     (+ Proc fred[k: Config](p: Int) → (Bool)
                            (+ Var i: Int
                               i := 1
                               do i ≤ | k | →
                                     if (k.(| k | -i) = p) ∨ (k.(| k | -i) = p-i) ∨
                                        (k.(| k |-i) = p+i) → return false fi
                                     i := i+1
                               od
                               return true
                            +)
                        end fred
                        Var p: Int
                        p := 0
                        do p ≠ n →
                              if fred[k](p) → PQ'Insert[Q](k++Config( p), | k |+1) fi
                              p := p+1
                        od
                     +)
               fi
            +)
        od
        if ¬ løsning → write("Ingen løsninger!", eol) fi
    +)
end Dronninger
```

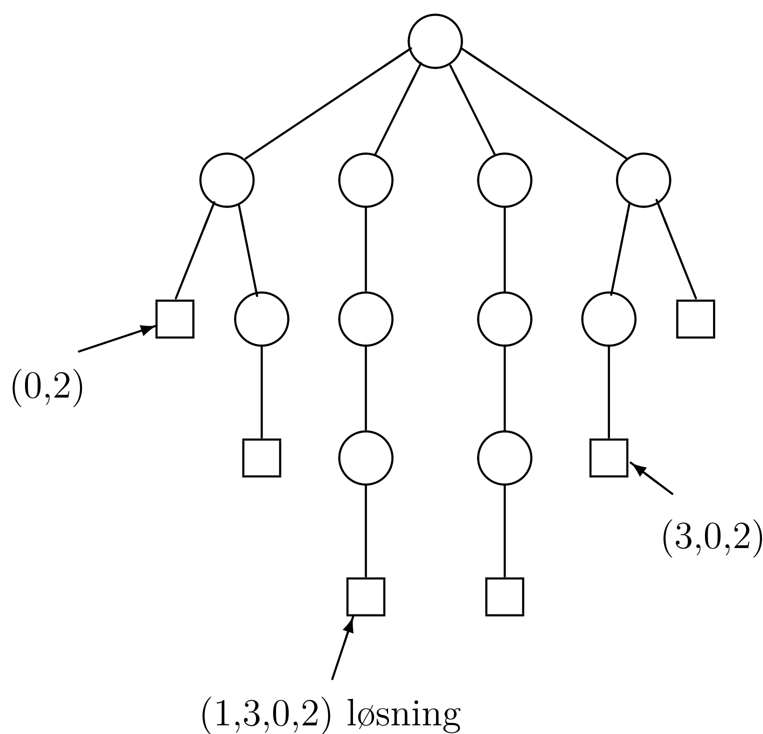
7.4 Rekursive løsninger

Den normale måde at præsentere løsninger til kombinatoriske søgeproblemer er i form af rekursive procedurer, der gennemløber konfigurationstræet på mere systematisk vis end den “springen fra blad til blad”, der repræsenteres af prioritetskø algoritmerne. Baggrunden herfor er den observation, at hvis man er villig til at opgive den eksplicitte kontrol (v.hj.a. prioriteterne) over den rækkefølge hvori konfigurationstræets blade ekspanderes, så behøver man heller ikke en eksplicit prioritetskø, idet denne da kan repræsenteres implicit i form af forskellige inkarnationer af en passende rekursiv procedure.

Vi illustrerer teknikken ved hjælp af følgende løsning til dronningeproblemet.

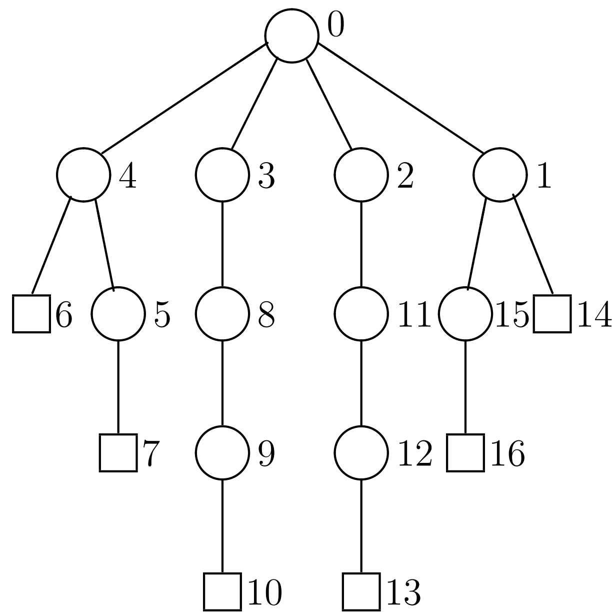
```
Process RecDronninger
    (+ Type Config = Vector

       Proc fred [k: Config] (p: Int) → (Bool)
           (+ Var j: Int
              j := 1
              do j ≤ | k | →
                     if (k.(| k |-j) = p) ∨
                        (k.(| k |-j) = p-j) ∨ (k.(| k |-j) = p+j) → return false fi
                     j := j+1
              od
              return true
           +)
       end fred

       Proc Dronning [n: Int] (k: Config)
           if | k | = n →
                  write("Løsning: ", k, eol)
                  stop
           & true →
                  (+ Var p: Int
                     p := 0
                     do p ≠ n →
                            if fred [k] (p) → Dronning [n] (k++Config(p)) fi
                            p := p+1
                     od
                  +)
           fi
       end Dronning

       Var n: Int

       write("Indlæs antal dronninger: ")
       read [n]
       Dronning [n] (Config())
       write("Ingen løsninger!", eol)
    +)
end RecDronninger
```

Hvis vi betragter det komplette konfigurationstræ med rod [λ] for 4×4 problemet



er det nemt at se, at proceduren Dronning gennemsøger træet ved (rekursivt) at undersøge undertræerne fra venstre mod højre (et såkaldt *dybde-først venstre-til-højre* gennemløb). Dette fortsætter indtil den finder det første blad i dybde 4, idet dette repræsenterer den første løsning.

Vi kan nu også se sammenhængen med prioritetskø-løsningen. Hvis vi nummererer træets knuder som følger



og tildeler konfigurationerne disse tal som prioriteter i en udførelse af prioritetskø-algoritmen, så vil denne behandle konfigurationerne i *præcis samme rækkefølge* som den rekursive procedure. Dette er en meget håndgribelig illustration af, hvordan de forskellige inkarnationer af den rekursive procedure definerer en implicit prioritetskø.

Vi slutter dette kapitel med (for fuldstændighedens skyld) også at angive en rekursiv løsning til 0-1 Knapsack problemet.

```
Process Rec01Knapsack
    (+ Type Sel = List(Bool)
       Type Config = Prod(M: Sel, i, c: Int)

       Proc Rec01Knap[S, V: Vector, n: Int, Kopt: Config, Vopt: Int](k: Config, Vpot: Int)
           if k.i = n →
                 if Vpot>Vopt → Vopt, Kopt := Vpot, k fi
           & true →
                 (+ Var ki: Int
                    Var nyk: Config
                    ki := k.i
                    if (S.(ki) ≤ k.c) ∧ (Vpot>Vopt) →
                          nyk := k
                          nyk.M.(ki), nyk.i, nyk.c := true, ki+1, k.c-S.(ki)
                          Rec01Knap[S, V, n, Kopt, Vopt](nyk, Vpot)
                    fi
                    if Vpot-V.(ki)>Vopt →
                          nyk := k
                          nyk.i := ki+1
                          Rec01Knap[S, V, n, Kopt, Vopt](nyk, Vpot-V.(ki))
                    fi
                 +)
           fi
       end Rec01Knap

       Var C, Vopt, n, Vsum: Int
       Var Kopt: Config
       Var S, V: Vector

       ≪Indlæs n, C, S og V≫
       ≪Beregn Vsum≫
       Kopt, Vopt := Config(Sel(false | n), 0, C), 0
       Rec01Knap[S, V, n, Kopt, Vopt](Config(Sel(false | n), 0, C), Vsum)
       write("Løsning: ", Kopt, eol)
       write("Værdi: ", Vopt, eol)
    +)
end Rec01Knapsack
where ≪Indlæs n, C, S og V≫ is
          write("Indlæs antal objekter: ")
          read[n]
          write("Indlæs kapacitet: ")
          read[C]
          write("Indlæs størrelser: ")
          read[S]
          write("Indlæs værdier: ")
          read[V]

where ≪Beregn Vsum≫ is
          (+ Var i: Int
             i, Vsum := 0, 0
             do i ≠ n → i, Vsum := i+1, Vsum+V.(i) od
          +)
```

8 Sammenfatning

De tre generelle problemløsningsteknikker, der er præsenteret i denne bog, kan opfattes som beregnet til at løse kombinatoriske problemer af varierende vanskelighed.

Den første, del-og-kombiner, er god, når hvert problem kan opdeles i et *lille* antal delproblemer, der ikke lapper for meget over hinanden.

Den anden, dynamisk programmering, kan anvendes selv om opdelingen af nogle problemer fører til et stort antal evt. overlappende delproblemer, blot der i den totale samling delproblemer ikke er for mange, der er forskellige.

Den tredie, kombinatorisk søgning, kan bruges i tilfælde, hvor det samlede antal delproblemer er for stort til at deres løsninger eksplicit kan lagres samtidigt. Der skal imidlertid findes en mekanisme, der styrer den rækkefølge hvori delproblemer genereres og løses.

Den varierende kompleksitet af de tre teknikker ses også af deres udførelsestider. Del-og-kombiner løsninger har ofte udførelsestider i $O(n \log n)$, dynamisk programmering i $O(n^\alpha)$, hvor α er et lille tal, medens kombinatorisk søgning normalt tilhører $O(2^{n^\alpha})$ hvor $0 < \alpha \leq 1$.

Det er ikke unormalt, i bøger om algoritmisk problemløsningsteknik, at omtale en fjerde teknik, den såkaldte *grådige algoritme*. Denne dækker over de situationer, hvor man kan beskrive løsningen til sit problem som en proces for et transitionssystem, hvor transitionssystemet har den egenskab, at det altid er "let" at beregne næste konfiguration i den optimale proces. En grådig algoritme er nu en simpel iteration, der ud fra startkonfigurationen successivt beregner konfigurationerne i den optimale proces og standser ved den første døde konfiguration.

Begrebet grådighed er ikke særlig godt karaktetiseret og vi skal derfor ikke omtale det nærmere. Det skal dog bemærkes, at en lang række af grafalgoritmerne, f.eks. Letteste udspændende træ, Korteste veje, og Topologisk sortering, er eksempler på grådige algortimer, hvor man v.hj.a. hensigtsmæssige konstruerede datastrukturer effektivt kan finde og udføre næste skridt i beregningen.

9 Referencer

[Harel] David Harel: ALGORITHMICS, The Spirit of Computing, 2. Edition, 1992.

[P&D] Erik Meineche Schmidt & Michael I. Schwartzbach: *Programmeringsteori og Datastrukturer*. DAIMI FN-56, Januar 1995.

[Transitionssystemer] M. Nielsen, E. M. Schmidt, M. I. Schwartzbach: Introduktion til transitionssystemer, Dat 1 – nr. 23, Februar 1995.

[TRINE] Erik Meineche Schmidt & Michael I. Schwartzbach: *Programmering og programmeringssproget Trine*. DAIMI FN-36, August 1994.

Algoritmer

0-1 Knapsack, 84
Del-og-kombiner, 47
Dijkstra, 39
Dynamisk Programmering, 78
Farvning af træ, 30
Flettesortering, 53
Floyd, 42
G til $\overline{G}$, 34
Graffarvning, 7, 8
Knapsack, 73
Kombinatorisk 0-1 Knapsack, 87, 90
Kombinatorisk Søgning, 92
Kruskal, 21
Pascals Trekant, 83
Prim, 26
Quicksort, 50
Selektion, 58
Tabel knapsack, 74
Tofarvning af træ, 31
Topologisk Sortering, 37

Programmer

Binomial, 81
Box MS, 11
Box NS, 11
DfsRec, 33
Dronninger, 96
Flettesortering, 54
FlettesorteringEnTo, 56
Floyd, 44
Knapsack, 91
Kruskal, 22
Lange heltal, 60
Mult_1, 63
Mult_2, 64
Mult_3, 65
Quicksort, 51
Rec01Knapsack, 100
RecDronninger, 97

Transitionssytemer

0-1 Knapsack, 86, 88
Acyklisk farvning, 35
Dronninger, 95
Graffarvning, 6
Letteste udspændende træ, 17

Stikordsregister

0-1 knapsack, 84

acykliske grafer, 35
afskæringsmekanisme, 88
algoritme-skabelon, 46
alle korteste veje, 41

binomialkoefficienter, 79
bredde-først, 13

CMult, 59
cykel, 3

dansk snit, 17
del-og-kombiner, 45
delgraf, 3
divide-and-conquer, 45
dronningeproblemet, 94
dybde-først, 13
dybde-først venstre-til-højre gennemløb, 98

efterfarve, 31
ekspandere, 87
enkelt-kilde problem, 38

finsk cykel, 17
flettesortering, 53
forfarve, 31
forventede udførelsestid, 52
fred(k, p), 95

genbrug, 74
grad, 3
graf, 2
grafgennemløb, 6

Hanois Tårne, 48
heltalsmultiplikation, 59
heuristik, 90
hybrid datastruktur, 27

ikke-orienteret, 2
incidensmatrix, 4
incidente, 3
indgraden, 3
induceres, 3

kanter, 2
kantliste, 4
knapsackproblem, 72
knuder, 2
kombinatorisk søgning, 84
konfigurationstræ, 86
korteste veje, 29
Kruskals algoritme, 20

lange heltal, 59
letteste udspændende træer, 16

m-delmængde, 79
Minus, 59
monoton mængde, 8

naboer, 3
nondeterministisk mængde, 8
norske grafer, 17

optimeringsproblem, 72
orienteret, 2

Pascals trekant, 81
Plus, 59
postorder, 32
preorder, 32
Prims algoritme, 23
prioritetsmængde, 24

quicksort, 49

randbetingelsen, 72
rekursionsligning, 70
rodtræ, 30

sammenhængskomponenter, 12
selektionsproblem, 57
simpel, 3
simpel cykel, 3
skolealgoritme, 63
snit, 17
stærkt sammenhængende, 33
svagt sammenhængende, 33

TDiv, 59
TMod, 59
TMult, 59
topologisk sortering, 36

udgraden, 3
udspændende delgraf, 3
udspændende skov, 15
udspændende træ, 14
univers, 46

vej, 3
velsitueret, 75
vægtede grafer, 15, 38