Contents

0 Indledning

Dette hæfte introducerer grafer, de mest fundamentale begreber fra grafteorien, samt nogle af de vigtigste grafalgoritmer.

Grafer har en bred anvendelsesflade inden for beskrivelser af kombinatoriske problemer, og det er derfor vigtigt at kende effektive algoritmer til løsning af de oftest forekommende grafproblemer. Grafalgoritmer er også gode eksempler på systematisk anvendelse af ikke-trivielle datastrukturer.

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.

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

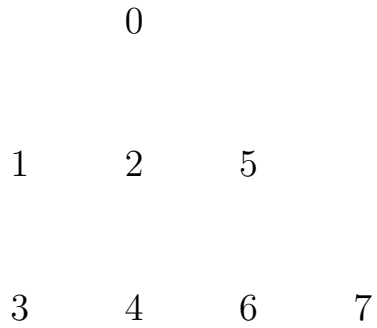
$$V = \{0, 1, \ldots, n-1\}$$

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.

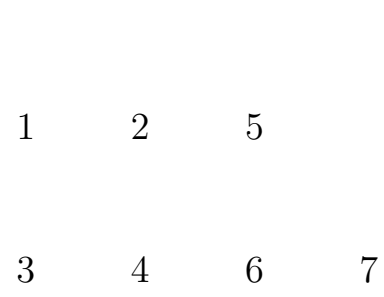
Følgende tegning repræsenterer således den ikke-orienterede graf

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



medens følgende viser den orienterede graf

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



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.

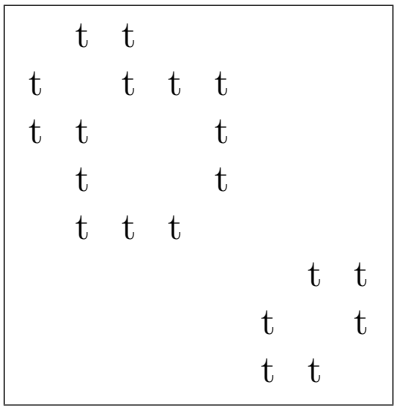
En *incidensmatrix* er en $n \times n$ Boolsk matrix, hvor den (v, w)'te indgang er **true**, hvis $\{v, w\}$ eller $[v, w]$ er en kant og **false** ellers. Den relevante

TRINE type er her

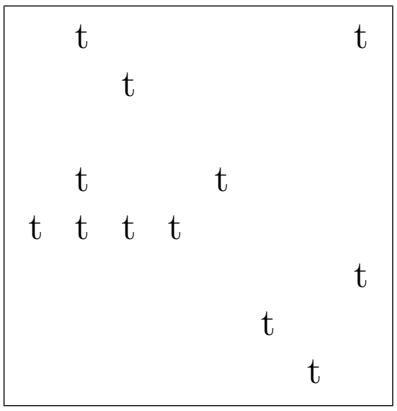
Type Graf = **List** (Bits)
Type Bits = **List** (Bool)

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

G_{io}:



G_o:



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 respon-

skravet, 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

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

$$m * (T[\![NS' \text{ Member } [PLR](w)]\!] + T[\![MS' \text{ Member } [H](w)]\!])+$$
$$T[\![\text{samtlige kald af formen } MS' \text{ DeleteSome } [H, v]]\!]$$

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 [Programmeringsteori].

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 *F*orbundet) 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.

$$\ll \text{farv } v \text{ rød} \gg \quad \textbf{is} \quad \begin{array}{l} N := N + 1 \\ CN.(v) := N \end{array}$$

$$\ll \text{farv } (u, w) \text{ og } w \text{ rød} \gg \quad \textbf{is} \quad CN.(w) := N$$

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. [Datastrukturer]. 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 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. [Datastrukturer]), 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: $\ll$Indsæt G's kanter i $P \gg$
$\ll T := (\{0, 1, \ldots, n-1\}, \emptyset) \gg$
Init $[R](n)$
do $\neg$ Empty $[P] \rightarrow$
DelMin $[P, e]$
if $\neg$ Equiv $[R](e.n1, e.n2) \rightarrow$
(* e forbinder to forskellige blå træer
og kan farves blå *)

$$\ll T := T \oplus (\emptyset, \{e\}) \gg$$
Join $[R](e.n1, e.n2)$
& **true** $\rightarrow$ **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 [Datastrukturer]. 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 prioritetskøer; 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** $(\text{Empty}' = (ps = \emptyset)) \wedge (ps' = ps)$
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** $(\text{Member}' = \ (\exists\, b, q : (v, b, q) \in ps)) \wedge (ps' = ps)$
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 ef-

fektivt.

Det er klart, at operationerne Init, Empty, Insert og Delmin kan understøttes af en bunke på sædvanlig vis (jfr. [Datastrukturer]). 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 derimod 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.

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

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 9. 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 8.

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) ∈ E do
        ≪ farv (v, w) og w røde ≫
        Farv [T](w)
    od
end Farv

≪ farv r rød ≫
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

≪ farv (v, w) og w røde ≫

med

≪ farv w rød ≫

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 $v \in V$ **do**
for w i $G.(v)$ **do**
if $v < w \rightarrow \ll$ tilføj w til $H.(v) \gg$
| $w < v \rightarrow \ll$ tilføj v til $H.(w) \gg$
fi
od
od
for $v \in V$ **do**
$\ll$fjern eventuelle dubletter fra $H.(v) \gg$
od
$\bar{G} := H$

for $v \in V$ **do**
  for w i $H.(v)$ **do**
    $\ll$ tilføj v til $\bar{G}.(w) \gg$
  od
od

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

$\ll$ tilføj v til $H.(w) \gg$
$\ll$fjern eventuelle dubletter fra $H.(v) \gg$

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 :

$\triangleright$

$\triangleright$ **hvis** $\bigcirc$'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.

Algoritme: Topologisk Sortering
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: (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 $D.(v)$'s variant 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** $(D.(v), \text{blue}) \Rightarrow D.(v).\text{blue}$
D: er længden af den korteste vej fra 0 til v
is $(D.(v), \text{white}) \Rightarrow v$ 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 $PS'\text{Prty}[L](v)$ 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 40. Vi kan repræsentere grafen v.hj.a. en særlig incidensmatrix på følgende form

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

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 $u \neq n \rightarrow$
≪farv u grøn≫
for $(v,w) \in V \times V$ **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 Referencer

[Datastrukturer] Michael I. Schwartzbach: *Datastrukturer.* DAIMI FN-38, Februar 1994.

[Programmeringsteori I] Erik Meineche Schmidt: *Programmeringsteori I.* DAIMI FN-51, Januar 1994.

[Programmeringsteori II] Erik Meineche Schmidt: *Programmeringsteori II.* DAIMI FN-52, Marts 1994.

[Transitionssystemer] Erik Meineche Schmidt: Introduktion til transitionssystemer, Dat 1 – nr. 23, Februar 1994.

[Trine] Erik Meineche Schmidt & Michael I. Schwartzbach: *Programmering og programmeringssproget Trine.* DAIMI FN-36, August 1993.