

# INF2220 - Algoritmer og datastrukturer

HØSTEN 2008

Institutt for informatikk, Universitetet i Oslo

## INF2220, forelesning 7: Grafer II

### Dijkstras algoritme

#### Korteste vei i en vektet graf uten negative kanter

- ▶ Graf uten vekter:
  - ▶ Velger først alle nodene med avstand 1 fra startnoden, så alle med avstand 2 osv
  - ▶ Mer generelt: Velger hele tiden en ukjent node blant dem med minst avstand fra startnoden
- ▶ Den samme hovedideen kan brukes hvis vi har en graf med vekter
- ▶ Akkurat som for uvektede grafer, ser vi bare etter potensielle forbedringer for naboer som ennå ikke er valgt (kjent)

Denne algoritmen ble publisert av Dijkstra i 1959 og har fått navn etter ham

### Dagens plan:

#### Korteste vei, en-til-alle, for:

- ▶ Vektet rettet graf uten negative kanter (Dijkstras algoritme)
- ▶ Vektet rettet graf med negative kanter

#### Minimale spenntrær

- ▶ Prim
- ▶ Kruskal

#### Dybde først søk

- ▶ Finne sammenhengskomponenter
- ▶ Løkkeleting

#### Dobbeltsammenhengende grafer

- ▶ Å finne ledd-noder (articulation points)

### Dijkstras algoritme

#### 1. For alle noder:

Sett avstanden fra startnoden  $s$  lik  $\infty$ . Merk noden som «ukjent»

#### 2. Sett avstanden fra $s$ til seg selv lik 0

3. Velg en ukjent node  $v$  med minimal avstand fra  $s$  (det kan være flere med samme avstand), og marker  $v$  som «kjent»

#### 4. For hver ukjent nabonode $w$ til $v$ :

Dersom avstanden vi får ved å følge veien gjennom  $v$ , er kortere enn den gamle avstanden til  $s$

- ▶ reduserer avstanden til  $s$  for  $w$
- ▶ sett bakoverpekeren i  $w$  til  $v$

#### 5. Så lenge det finnes ukjente noder, gå til punkt 3

**Merk:** Dijkstras algoritme virker både på rettede og urettede grafer

## Hvorfor virker algoritmen?

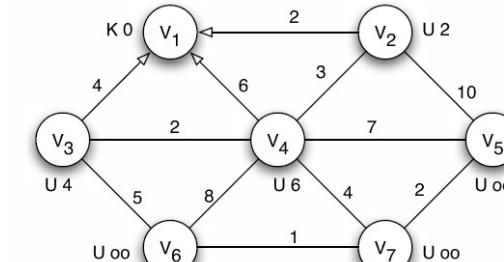
- Algoritmen har følgende *invariant*:

*Ingen kjent node har større avstand til s enn en ukjent node*

- Følgelig har alle kjente noder riktig korteste vei satt (registrert avstand er lavest mulig kost av en vei til s)
- Vi plukker ut en ukjent node  $v$  med minst avstand ( $d_v$ ), markerer den som kjent og påstår at avstanden til  $v$  er riktig
- Denne påstanden holder fordi
  - $d_v$  er den korteste veien som finnes ved å bruke bare kjente noder
  - de kjente nodene har riktig korteste vei satt
  - en vei til  $v$  som er kortere enn  $d_v$ , må nødvendigvis forlate mengden av kjente noder et sted,  
men  $d_v$  er allerede den korteste veien fra kjente noder til  $v$
- Dette argumentet holder fordi vi ikke har negative kanter

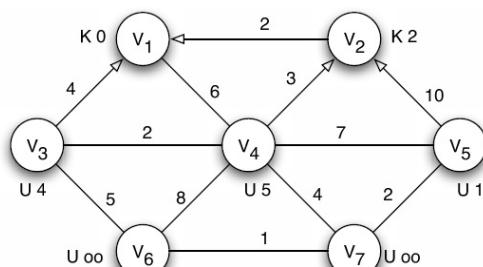
## Eksempel

Den første noden som velges, er startnoden  $V_1$

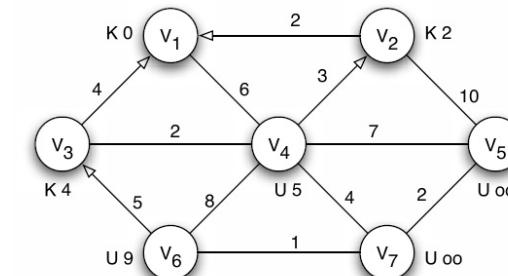


Naboene til  $V_1$  har fått endret sin avstand og fått tilbakepekkere til  $V_1$

Nå er  $V_2$  nærmeste ukjente node



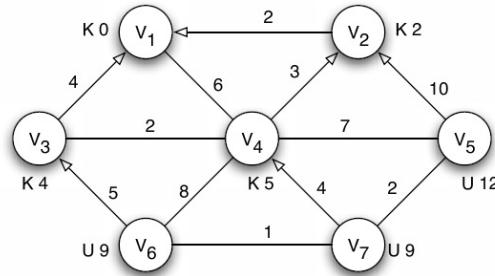
$V_4$  har fått ny avstand og tilbakepeker,  
mens  $V_3$  er nærmeste ukjente node



$V_6$  har fått endret sin avstand og fått tilbakepeker til  $V_3$

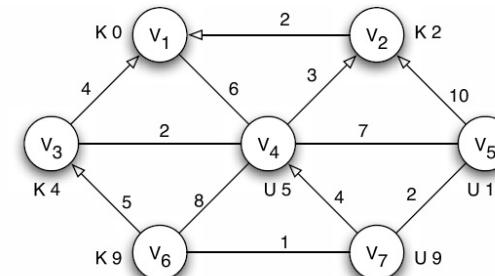
Nå er  $V_4$  nærmeste ukjente node

Merk at den aldri senere kan få endret sin avstand

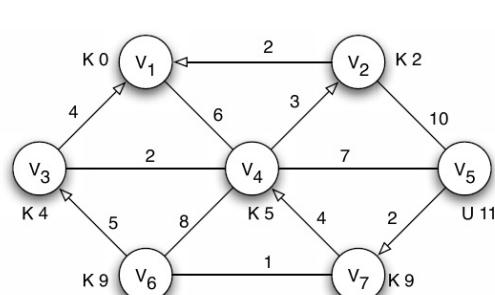


V<sub>7</sub> har fått ny avstand og tilbakepeker,  
og V<sub>6</sub> og V<sub>7</sub> er nærmeste ukjente noder

Vi velger V<sub>6</sub> som blir kjent



Nå er V<sub>7</sub> nærmest og blir kjent



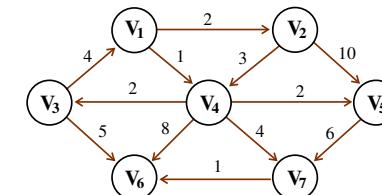
V<sub>5</sub> får ny avstand og tilbakepeker

Vi kan nå avslutte med å gjøre V<sub>5</sub> kjent

## Dijkstras algoritme

### Oppgave

Bruk Dijkstras algoritme, og fyll ut tabellen nedenfor!



Initielt:

v	kjent	avstand	vei
V <sub>1</sub>	F	0	0
V <sub>2</sub>	F	$\infty$	0
V <sub>3</sub>	F	$\infty$	0
V <sub>4</sub>	F	$\infty$	0
V <sub>5</sub>	F	$\infty$	0
V <sub>6</sub>	F	$\infty$	0
V <sub>7</sub>	F	$\infty$	0

v	kjent	avstand	vei
V <sub>1</sub>			
V <sub>2</sub>			
V <sub>3</sub>			
V <sub>4</sub>			
V <sub>5</sub>			
V <sub>6</sub>			
V <sub>7</sub>			

## Tidsforbruk

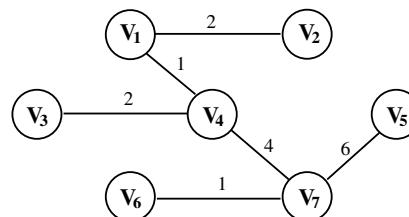
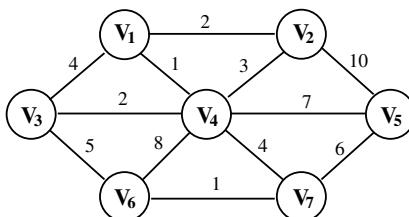
- Hvis vi leter sekvensielt etter den ukjente noden med minst avstand tar dette  $\mathcal{O}(|V|)$  tid, noe som gjøres  $|V|$  ganger, så total tid for å finne minste avstand blir  $\mathcal{O}(|V|^2)$
- I tillegg oppdateres avstandene, maksimalt en oppdatering per kant, dvs. til sammen  $\mathcal{O}(|E|)$
- Total tid:  $\mathcal{O}(|E| + |V|^2) = \mathcal{O}(|V|^2)$
- Bra for tette grafer: ( $|E| = |V|^2$ )

### Raskere implementasjon (for tynne grafer):

- Bruker en **prioritetskø** til å ta vare på ukjente noder med avstand mindre enn  $\infty$
- Prioriteten til ukjent node forandres hvis vi finner kortere vei til noden
- removeMin** og **decreaseKey** bruker  $\mathcal{O}(\log |V|)$  tid (jfr. kap. 8)
- Totalt tidsforbruk blir  $\mathcal{O}(|V| \log |V| + |E| \log |V|) = \mathcal{O}(|E| \log |V|)$

## Minimale spennrør

- Et minimalt spennrør for en **urettet** graf  $G$  er et tre med kanter fra grafen, slik at alle nodene i  $G$  er forbundet til lavest mulig kostnad
- Minimale spennrør eksisterer bare for sammenhengende grafer
- Generelt kan det finnes flere minimale spennrør for samme graf



Hvor mange kanter får spennrøret i det generelle tilfellet?

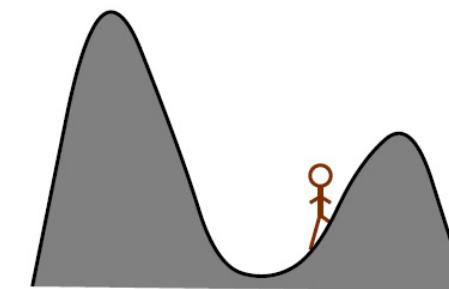
## Hva med negative kanter?

- Dijkstras algoritme fungerer ikke med negative kanter
  - En mulig løsning:**
    - Nodene er ikke lenger «kjente» eller «ukjente»
    - Vi har i stedet en **kø** som inneholder noder som har fått forbedret avstandsverdien sin
    - Løkken i algoritmen gjør følgende:
      - Ta ut en node **v** fra køen
      - For hver etterfølger **w**, sjekk om vi får en forbedring
      - Oppdater i så fall avstanden, og plasser **w** (tilbake) i køen (hvis den ikke er der allerede)
    - Tidsforbruket blir  $\mathcal{O}(|E| \cdot |V|)$ , dvs. mye verre enn Dijkstras algoritme
    - Det finnes ingen korteste vei med **negative løkker** i  $G$ . Det er det hvis og bare hvis samme node blir tatt ut av køen mer enn  $|V|$  ganger.
- Da må vi terminere algoritmen!

## Minimale spennrør Grådige algoritmer

### Grådige algoritmer

- Prøver i hvert trinn å gjøre det som ser best ut der og da
- Typisk eksempel: **Gi vekselpenger**
- Raske algoritmer, men kan ikke løse alle problemer:

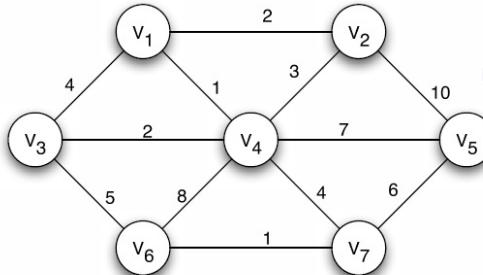


Finn det høyeste punktet!

Vi skal se på to ulike grådige algoritmer for å finne minimale spennrør

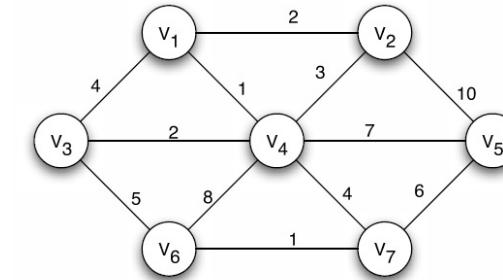
## Prims algoritme

- ▶ Treet bygges opp trinnvis
  - I hvert trinn legges en kant (og dermed en tilhørende node) til treet
- ▶ Vi har til enhver tid to typer noder:
  - ▶ Noder som er med i treet
  - ▶ Noder som ikke er med i treet
- ▶ Nye noder legges til ved å velge en kant  $(u, v)$  med **minst** vekt slik at  $u$  er med i treet, og  $v$  ikke er det.

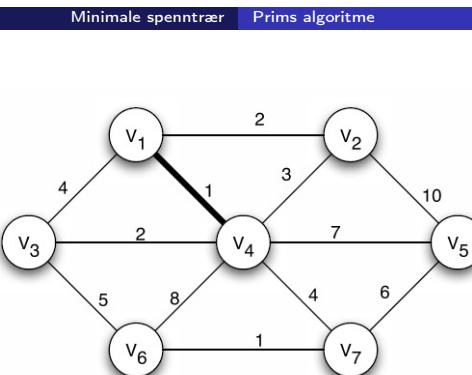


▶ Algoritmen begynner med å velge en vilkårlig node

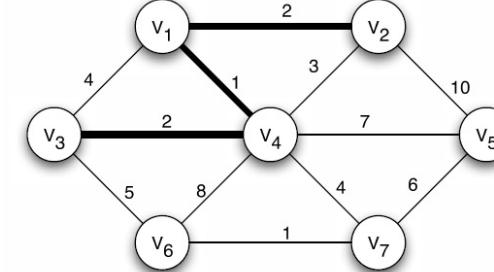
**Eksempel:** La oss velge v<sub>1</sub>



- ▶ Minste kant ut fra v<sub>1</sub> går til v<sub>4</sub>, så vi legger den inn i spennetreeet



- ▶ Vi har nå to mulige fortsettelse:
  - ▶ Kanten fra v<sub>1</sub> til v<sub>2</sub>
  - ▶ Kanten fra v<sub>4</sub> til v<sub>3</sub>
- ▶ Samme hvilken vi velger, vil den andre av dem bli neste kant, så vi legger dem begge inn i spennetreeet

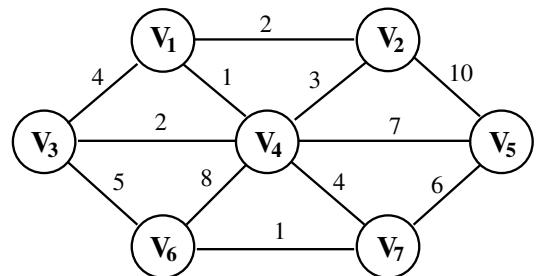


- ▶ Minste kant ut fra spennetreeet går nå fra v<sub>4</sub> til v<sub>7</sub>
- ▶ Så får vi kanten fra v<sub>7</sub> til v<sub>6</sub>
- ▶ Til slutt får vi kanten fra v<sub>7</sub> til v<sub>5</sub>

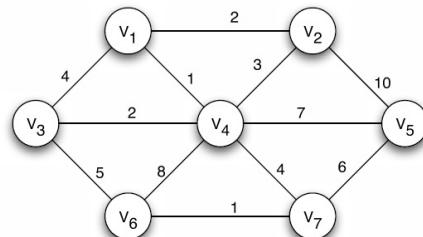
- Prims algoritme er essensielt lik Dijkstras algoritme for korteste vei!

I Prims algoritme er «avstanden» til en ukjent node  $v$  den minste vekten til en kant som forbinder  $v$  med en **kjent** node

- Husk at vi har urettede grafer, så hver kant befinner seg i **to** nabolister
- Kjøretidsanalysen er den samme som for Dijkstras algoritme



V	kjent	avstand	vei
V <sub>1</sub>	F	0	0
V <sub>2</sub>	F	$\infty$	0
V <sub>3</sub>	F	$\infty$	0
V <sub>4</sub>	F	$\infty$	0
V <sub>5</sub>	F	$\infty$	0
V <sub>6</sub>	F	$\infty$	0
V <sub>7</sub>	F	$\infty$	0

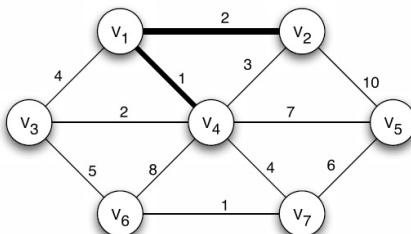


node	kjent	avst.	fra
V <sub>1</sub>	F	0	0
V <sub>2</sub>	F	oo	0
V <sub>3</sub>	F	oo	0
V <sub>4</sub>	F	oo	0
V <sub>5</sub>	F	oo	0
V <sub>6</sub>	F	oo	0
V <sub>7</sub>	F	oo	0

node	kjent	avst.	fra
V <sub>1</sub>	T	0	0
V <sub>2</sub>	F	2	V <sub>1</sub>
V <sub>3</sub>	F	2	V <sub>4</sub>
V <sub>4</sub>	T	1	V <sub>1</sub>
V <sub>5</sub>	F	7	V <sub>4</sub>
V <sub>6</sub>	F	8	V <sub>4</sub>
V <sub>7</sub>	F	4	V <sub>4</sub>

Initialtilstanden

V<sub>1</sub> er lagt inn



node	kjent	avst.	fra
V <sub>1</sub>	T	0	0
V <sub>2</sub>	T	2	V <sub>1</sub>
V <sub>3</sub>	T	2	V <sub>4</sub>
V <sub>4</sub>	T	1	V <sub>1</sub>
V <sub>5</sub>	F	7	V <sub>4</sub>
V <sub>6</sub>	F	5	V <sub>3</sub>
V <sub>7</sub>	F	4	V <sub>4</sub>

node	kjent	avst.	fra
V <sub>1</sub>	T	0	0
V <sub>2</sub>	T	2	V <sub>1</sub>
V <sub>3</sub>	T	2	V <sub>4</sub>
V <sub>4</sub>	T	1	V <sub>1</sub>
V <sub>5</sub>	F	7	V <sub>4</sub>
V <sub>6</sub>	F	6	V <sub>7</sub>
V <sub>7</sub>	F	1	V <sub>7</sub>

V<sub>3</sub> er lagt inn

V<sub>7</sub> er lagt inn

node	kjent	avst.	fra
V <sub>1</sub>	T	0	0
V <sub>2</sub>	T	2	V <sub>1</sub>
V <sub>3</sub>	T	2	V <sub>4</sub>
V <sub>4</sub>	T	1	V <sub>1</sub>
V <sub>5</sub>	F	6	V <sub>7</sub>
V <sub>6</sub>	T	1	V <sub>7</sub>
V <sub>7</sub>	T	4	V <sub>4</sub>

node	kjent	avst.	fra
V <sub>1</sub>	T	0	0
V <sub>2</sub>	T	2	V <sub>1</sub>
V <sub>3</sub>	T	2	V <sub>4</sub>
V <sub>4</sub>	T	1	V <sub>1</sub>
V <sub>5</sub>	T	6	V <sub>7</sub>
V <sub>6</sub>	T	1	V <sub>7</sub>
V <sub>7</sub>	T	4	V <sub>4</sub>

V<sub>6</sub> er lagt inn

V<sub>5</sub> er lagt inn. Ferdig!

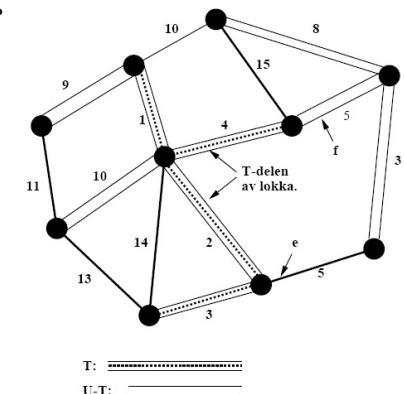
## Hvorfor virker Prim?

### Løkke-lemma for spenntrær

Anta at  $U$  er et spenntrær for en graf, og at kanten  $e$  ikke er med i treet  $U$ . Hvis vi legger kanten  $e$  til treet  $U$ , dannes en entydig bestemt enkel løkke. Hvis, og bare hvis, vi fjerner en vilkårlig kant i denne løkken, vil vi igjen ha et spenntrær for grafen

### Invariant for Prims algoritme

Det treet  $T$  som dannes av de kantene (og deres endenoder) vi til nå har plukket ut, er slik at det finnes et minimalt spenntrær  $U$  for grafen som inneholder (alle kantene i)  $T$



## Kruskals algoritme:

Se på kantene en etter en, sortert etter minst vekt

Kanten aksepteres hvis, og bare hvis, den ikke fører til noen løkke

Algoritmen implementeres vha. en prioritetskø og disjunkte mengder:

- ▶ Initiert plasseres kantene i en prioritetskø og nodene i hver sin disjunkte mengde (slik at **find(v)** gir mengden til **(v)**).

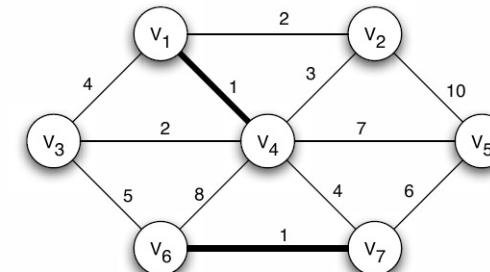
**Invariant:**

De disjunkte mengdene er subtrærer av det endelige spennetreet

- ▶ **removeMin** gir neste kant **(u, v)** som skal testes
  - ▶ Hvis **find(u) ! = find(v)**, har vi en ny kant i treet og gjør **union(u, v)**
  - ▶ Hvis ikke, ville **(u, v)** ha dannet en løkke, så kanten forkastes
- ▶ Algoritmen terminerer når prioritetskøen er tom, eventuelt når vi har lagt inn  $|V| - 1$  kanter

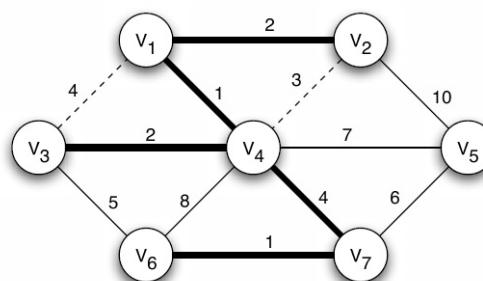
## Eksempel

Utgangspunkt for Kruskals algoritme:



Vi har to kanter med **vekt 1** — De blir til to subtrærer i spenn-treet

Vi har to kanter med **vekt 2** — De blir del av det øverste subtreet

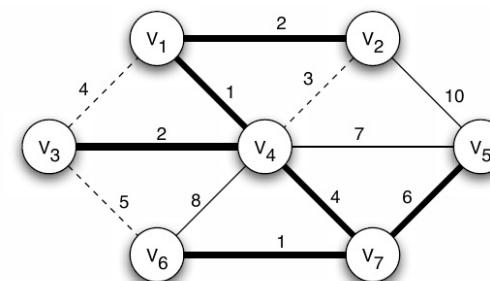


Vi har en kant med **vekt 3** — Den vil lage en løkke og må forkastes

Vi har to kanter med **vekt 4** hvor den mellom **v1** og **v3** danner en løkke, mens den andre binder de to subtrærne sammen

Vi har en kant med **vekt 5** som lager løkke

Vi har en kant med **vekt 6**: Den knytter den siste noden til treet



Nå har vi lagt inn  $|V| - 1$  kanter i spenn-treet og kan slutte

Hvis vi fortsetter med resten av kantene, vil alle danne løkker og bli forkastet

## Tidsanalyse:

- Hovedløkken går  $|E|$  ganger
- I hver iterasjon gjøres en **removeMin**, to **find** og en **union**, med samlet tidsforbruk
 
$$\mathcal{O}(\log |E|) + 2 \cdot \mathcal{O}(\log |V|) + \mathcal{O}(1) = \mathcal{O}(\log |V|)$$

(fordi  $\log |E| < 2 \cdot \log |V|$ )
- Totalt tidsforbruk er  $\mathcal{O}(|E| \cdot \log |V|)$

## Prim vs. Kruskal

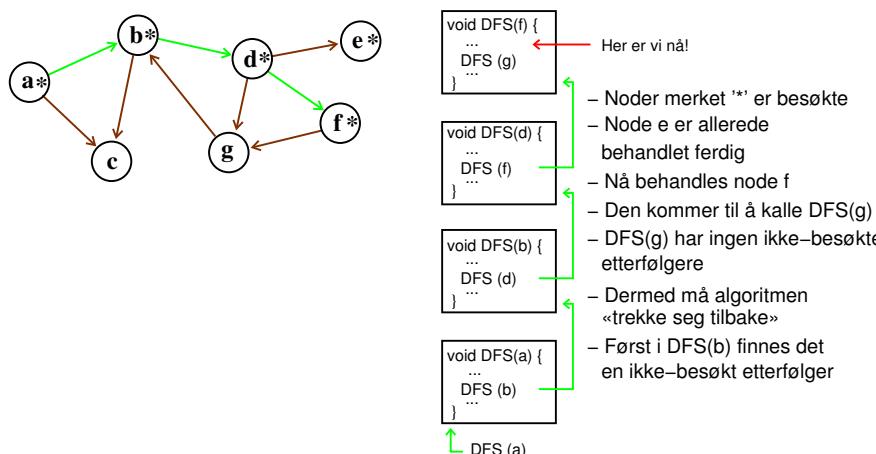
- Prims algoritme er noe mer effektiv enn Kruskals, spesielt for tette grafer
- Prims algoritme virker bare i sammenhengende grafer
- Kruskals algoritme gir et minimalt spenn-tre i hver sammenhengskomponent i grafen

## Dybde-først søk

- Generalisering av prefiks traversering for trær.
- Vi starter i en node **v** og traverserer alle nabonodene rekursivt
- Rekursjonen gjør at vi undersøker alle noder som kan nås fra første etterfølger til **v**, før vi undersøker neste etterfølger til **v**
- For en vilkårlig graf må vi passe på å **unngå løkker**.
  - Markerer nodene som besøkt etterhvert som de behandles, og kaller rekursivt videre bare for umerkede noder

```
void dybdeFørstSøk(Node v) {
    v.merk = true;
    for < hver nabo w til v > {
        if (!w.merk) {
            dybdeFørstSøk(w);
        }
    }
}
```

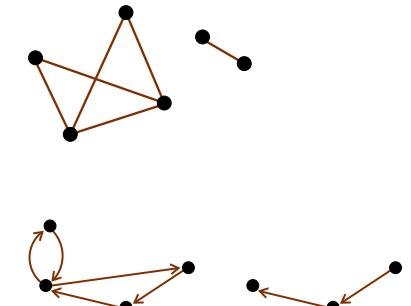
Midtveis i en dybde-først traversering kan kjeden av rekursive metodekall og besøkt-merkene i nodene se slik ut:



## Er grafen sammenhengende?

Hvis grafen ikke er sammenhengende, kan vi foreta nye dybde-først søk fra noder som ikke er besøkte, inntil alle noderne er behandlet

- En urettet graf er sammenhengende hvis og bare hvis et dybde-først søk som starter i en tilfeldig node, besøker alle noderne i grafen
- En rettet graf er sterkt sammenhengende hvis og bare hvis vi fra hver eneste node **v** klarer å besøke alle de andre noderne i grafen ved et dybde-først søk fra **v**



## Løkkeleting

- ▶ Vi kan bruke dybde-først søk til å sjekke om en graf har løkker
- ▶ Bruker tre verdier til tilstandsvariablen: **usett**, **igang** og **ferdig** (besøkt)

```
void løkkeLet(Node v) {
    if (v.tilstand == igang) {
        < Løkke er funnet >
    } else if (v.tilstand == usett) {
        v.tilstand = igang;
        for < hver nabo w til v > {
            løkkeLet(w);
        }
        v.tilstand = ferdig;
    }
}
```

- ▶ Metoden for løkkeleting kan også gi oss en topologisk sortering (med nodene skrevet ut i omvendt rekkefølge) dersom grafen ikke har løkker
- ▶ Det får vi til ved å skrive ut noden like før vi trekker oss tilbake (idet vi er ferdig med kallet)
- ▶ Dette er riktig tidspunkt å skrive ut noden fordi:
  - ▶ Vi har sjekket alle etterfølgerne til noden
  - ▶ Disse var enten usette (og da har vi skrevet dem ut i det vi trakk oss tilbake fra dem), eller ferdige (og da var de skrevet ut tidligere)

Dersom vi fant en node som var igang, har vi funnet en løkke ...

- ▶ Noder der kall er i gang ligger alltid på en rett vei fra startnoden
- ▶ Vi må passe på å gjøre nye startkall inntil metoden er kalt i alle nodene
- ▶ Metoden vil alltid finne en løkke dersom det eksisterer en!

## Grafer: Oppsummering

- ▶ Implementasjon av grafer
- ▶ Topologisk sortering
- ▶ Korteste vei, uvektet graf
- ▶ Dijkstras alg. (korteste vei, vektet graf)
- ▶ Prim, Kruskal (minimale spenntrær)
- ▶ Dybde-først søk