INF2220 - Algoritmer og datastrukturer

HØSTEN 2007

Institutt for informatikk, Universitetet i Oslo

INF2220, forelesning 7:
Grafer II

Dagens plan:

Korteste vei, en-til-alle, for:

- Vektet rettet graf uten negative kanter (kap. 9.3.2) (Dijkstras algoritme)
- Vektet rettet graf med negative kanter (kap. 9.3.3)

Minimale spenntrær

- Prim (kap. 9.5.1)
- Kruskal (kap. 9.5.2)

Dybde først søk (kap. 9.6.1)

- Finne sammenhengskomponenter
- Løkkeleting

Dobbeltsammenhengende grafer (kap. 9.6.2)

- Å finne ledd-noder (articulation points)

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 hovedidéen 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

Dijkstras algoritme

1. For alle noder:

   Sett avstanden fra startnoden **s** lik ∞. 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?

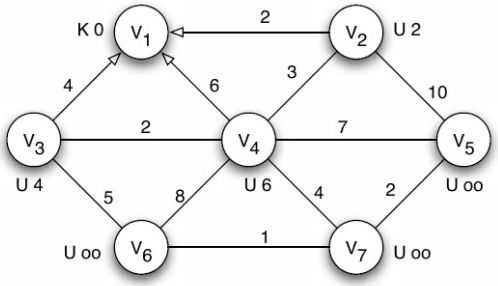
- Algoritmen har følgende *invariant*:

  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 ($\mathbf{d_v}$), markerer den som kjent og påstår at avstanden til **v** er riktig
- Denne påstanden holder fordi
  - $\mathbf{d_v}$ 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 $\mathbf{d_v}$, må nødvendigvis forlate mengden av kjente noder et sted,
    men $\mathbf{d_v}$ er allerede den korteste veien fra kjente noder til **v**
- Dette argumentet holder fordi vi ikke har negative kanter

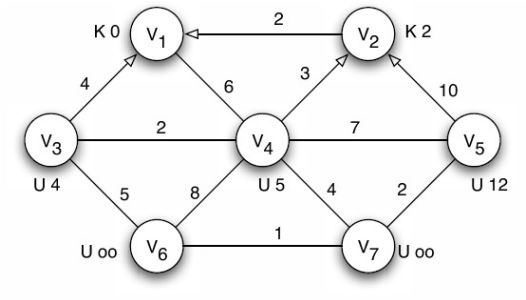
Eksempel

Den første noden som velges, er startnoden $\mathbf{V_1}$

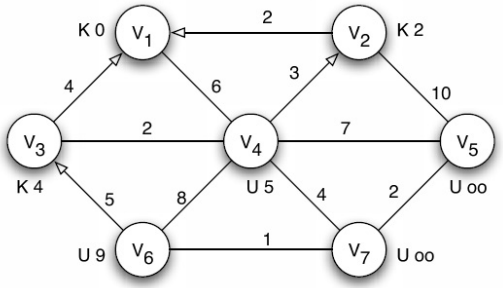


Naboene til $\mathbf{V_1}$ har fått endret sin avstand og fått tilbakepekere til $\mathbf{V_1}$

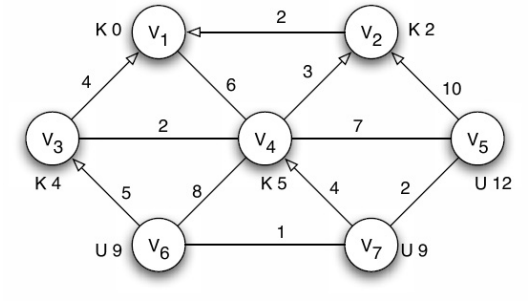
Nå er V_2 nærmeste ukjente node



$\mathbf{V_4}$ har fått ny avstand og tilbakepeker,
mens $\mathbf{V_3}$ er nærmeste ukjente node

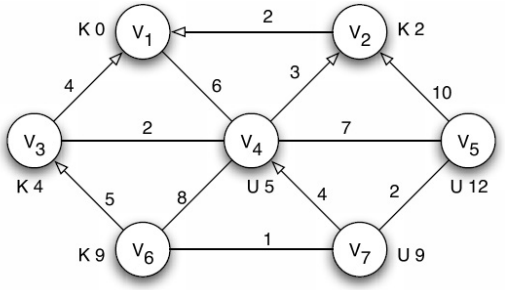


$\mathbf{V_6}$ har fått endret sin avstand og fått tilbakepeker til $\mathbf{V_3}$
Nå er $\mathbf{V_4}$ nærmeste ukjente node
Merk at den aldri senere kan få endret sin avstand

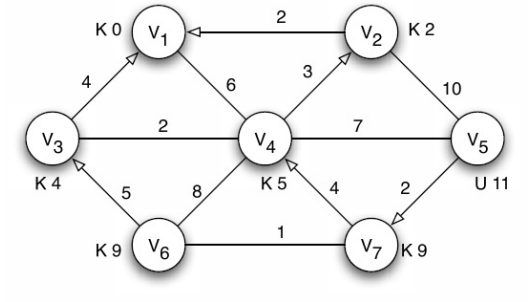


$\mathbf{V_7}$ har fått ny avstand og tilbakepeker,
og $\mathbf{V_6}$ og $\mathbf{V_7}$ er nærmeste ukjente noder

Vi velger $\mathbf{V_6}$ som blir kjent



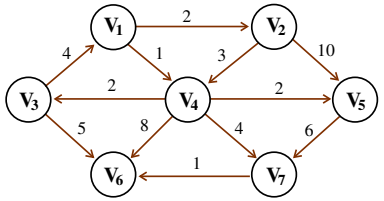
Nå er $\mathbf{V_7}$ nærmest og blir kjent



$\mathbf{V_5}$ får ny avstand og tilbakepeker
Vi kan nå avslutte med å gjøre $\mathbf{V_5}$ kjent

Oppgave

Bruk Dijkstras algoritme, og fyll ut tabellen nedenfor!



Initielt:

v	kjent	avstand	vei
v_1	F	0	0
v_2	F	∞	0
v_3	F	∞	0
v_4	F	∞	0
v_5	F	∞	0
v_6	F	∞	0
v_7	F	∞	0

v	kjent	avstand	vei
v_1			
v_2			
v_3			
v_4			
v_5			
v_6			
v_7			

Tidsforbruk

- Hvis vi leter sekvensielt etter den ukjente noden med minst avstand tar dette $\mathcal{O}(|\mathbf{V}|)$ tid, noe som gjøres $|\mathbf{V}|$ ganger, så total tid for å finne minste avstand blir $\mathcal{O}(|\mathbf{V}|^2)$
- I tillegg oppdateres avstandene, maksimalt en oppdatering per kant, dvs. til sammen $\mathcal{O}(|\mathbf{E}|)$
- Total tid: $\mathcal{O}(|\mathbf{E}| + |\mathbf{V}|^2) = \mathcal{O}(|\mathbf{V}|^2)$
- 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 ∞
- Prioriteten til ukjent node forandres hvis vi finner kortere vei til noden
- DeleteMin og DecreaseKey tar $\mathcal{O}(\log |\mathbf{V}|)$ tid
- Totalt tidsforbruk blir $\mathcal{O}(|\mathbf{V}| \log |\mathbf{V}| + |\mathbf{E}| \log |\mathbf{V}|) = \mathcal{O}(|\mathbf{E}| \log |\mathbf{V}|)$

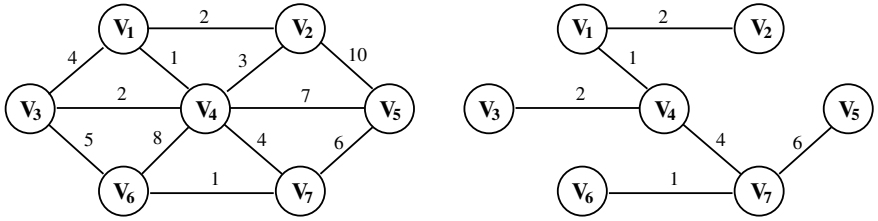
Hva med negative kanter?

- Dijkstras algoritme fungerer ikke med negative kanter (oppg. 9.7a)
- **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:
  1. Ta ut en node **v** fra køen
  2. For hver etterfølger **w**, sjekk om vi får en forbedring
  3. Oppdater i så fall avstanden, og plasser **w** (tilbake) i køen (hvis den ikke er der allerede)
- Tidsforbruket blir $\mathcal{O}(|\mathbf{E}| \cdot |\mathbf{V}|)$, 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 $|\mathbf{V}|$ ganger.

  Da må vi terminere algoritmen!

Minimale spenntrær

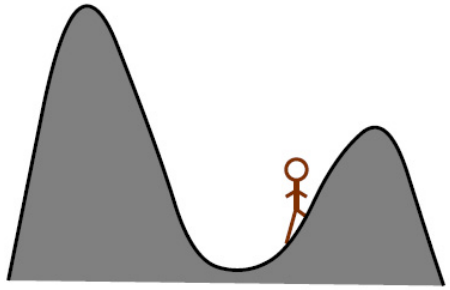
- Et minimalt spenntre for en **urettet** graf **G** er et tre bestående av kanter fra grafen, slik at alle nodene i **G** er forbundet til lavest mulig kostnad
- Minimale spenntrær eksisterer bare for sammenhengende grafer
- Generelt kan det finnes flere minimale spenntrær for samme graf



Hvor mange kanter får spenntreet i det generelle tilfellet?

Grådige algoritmer

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



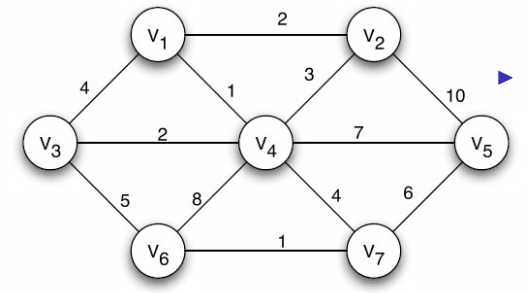
Finn det høyeste punktet!

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

Prims algoritme

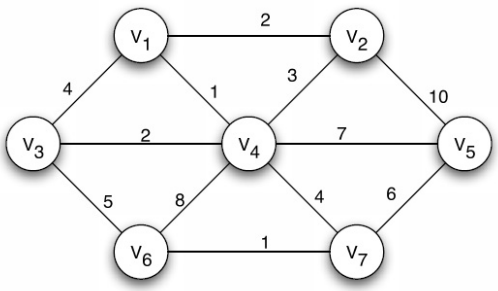
- Treet bygges opp trinnvis

  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 $(\mathbf{u}, \mathbf{v})$ med **minst** vekt slik at $\mathbf{u}$ er med i treet, og $\mathbf{v}$ ikke er det.

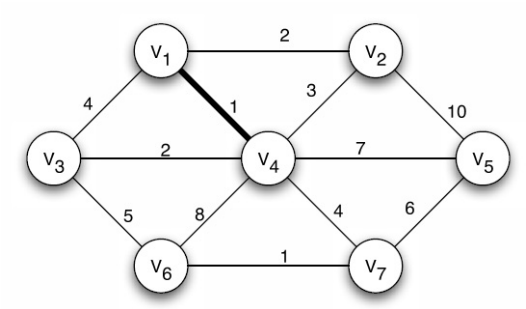


- Algoritmen begynner med å velge en vilkårlig node

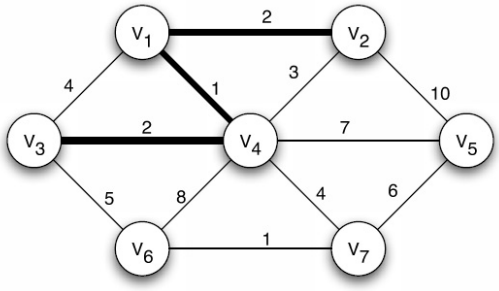
Eksempel: La oss velge $\mathbf{v_1}$



- Minste kant ut fra $\mathbf{v_1}$ går til $\mathbf{v_4}$, så vi legger den inn i spenntreet



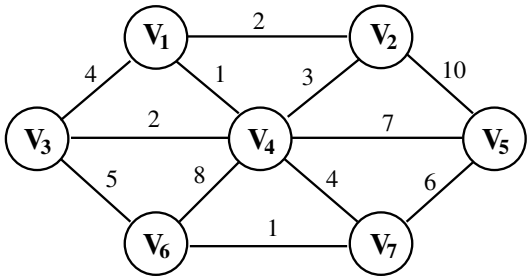
- Vi har nå to mulige fortsettelser:
  - Kanten fra $\mathbf{v_1}$ til $\mathbf{v_2}$
  - Kanten fra $\mathbf{v_4}$ til $\mathbf{v_3}$
- Samme hvilken vi velger, vil den andre av dem bli neste kant, så vi legger dem begge inn i spenntreet



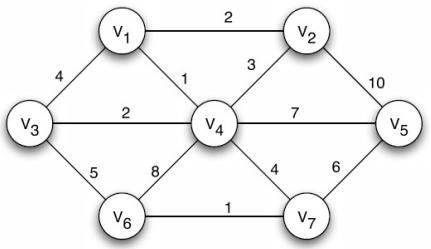
- Minste kant ut fra spenntreet går nå fra $\mathbf{v_4}$ til $\mathbf{v_7}$
- Så får vi kanten fra $\mathbf{v_7}$ til $\mathbf{v_6}$
- Til slutt får vi kanten fra $\mathbf{v_7}$ til $\mathbf{v_5}$

- 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_1	F	0	0
v_2	F	∞	0
v_3	F	∞	0
v_4	F	∞	0
v_5	F	∞	0
v_6	F	∞	0
v_7	F	∞	0



node	kjent	avst.	fra
V_1	F	0	0
V_2	F	oo	0
V_3	F	oo	0
V_4	F	oo	0
V_5	F	oo	0
V_6	F	oo	0
V_7	F	oo	0

Initialtilstanden

node	kjent	avst.	fra
V_1	T	0	0
V_2	F	2	V_1
V_3	F	4	V_1
V_4	F	1	V_1
V_5	F	oo	0
V_6	F	oo	0
V_7	F	oo	0

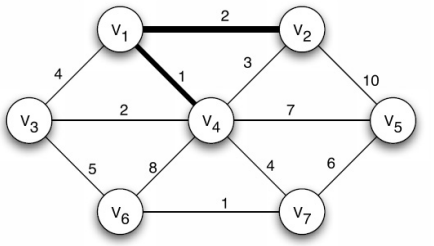
V_1 er lagt inn

node	kjent	avst.	fra
V_1	T	0	0
V_2	F	2	V_1
V_3	F	2	V_4
V_4	T	1	V_1
V_5	F	7	V_4
V_6	F	8	V_4
V_7	F	4	V_4

V_4 er lagt inn

node	kjent	avst.	fra
V_1	T	0	0
V_2	T	2	V_1
V_3	F	2	V_4
V_4	T	1	V_1
V_5	F	7	V_4
V_6	F	8	V_4
V_7	F	4	V_4

V_2 er lagt inn



node	kjent	avst.	fra
V_1	T	0	0
V_2	T	2	V_1
V_3	T	2	V_4
V_4	T	1	V_1
V_5	F	7	V_4
V_6	F	5	V_3
V_7	F	4	V_4

V_3 er lagt inn

node	kjent	avst.	fra
V_1	T	0	0
V_2	T	2	V_1
V_3	T	2	V_4
V_4	T	1	V_1
V_5	F	6	V_7
V_6	F	1	V_7
V_7	T	4	V_4

V_7 er lagt inn

node	kjent	avst.	fra
V_1	T	0	0
V_2	T	2	V_1
V_3	T	2	V_4
V_4	T	1	V_1
V_5	F	6	V_7
V_6	T	1	V_7
V_7	T	4	V_4

V_6 er lagt inn

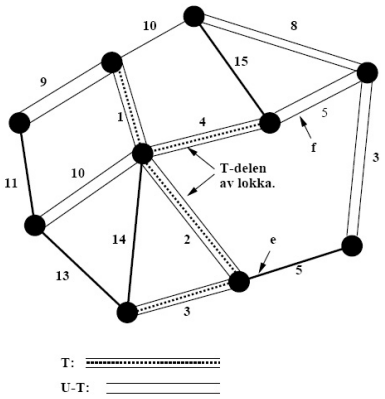
node	kjent	avst.	fra
V_1	T	0	0
V_2	T	2	V_1
V_3	T	2	V_4
V_4	T	1	V_1
V_5	T	6	V_7
V_6	T	1	V_7
V_7	T	4	V_4

V_5 er lagt inn. Ferdig!

Hvorfor virker Prim?

Løkke-lemma for spenntrær

Anta at **U** er et spenntre 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 spenntre 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 spenntre **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:

- Initielt plasseres kantene i en prioritetskø og nodene i hver sin disjunkte mengde

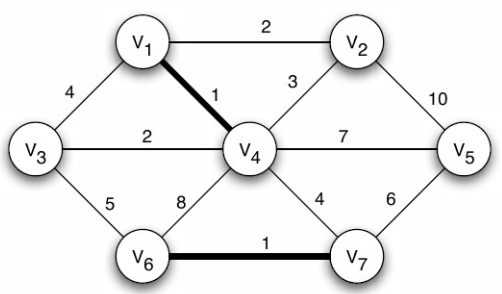
  Invariant:

  De disjunkte mengdene er subtrær av det endelige spenntreet

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

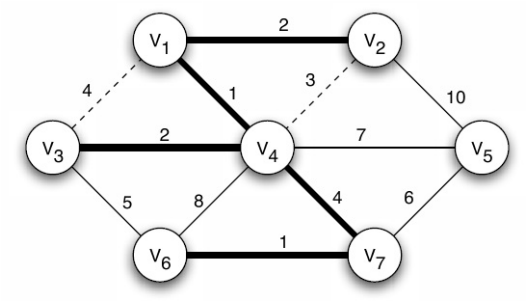
Eksempel

Utgangspunkt for Kruskals algoritme:



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

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

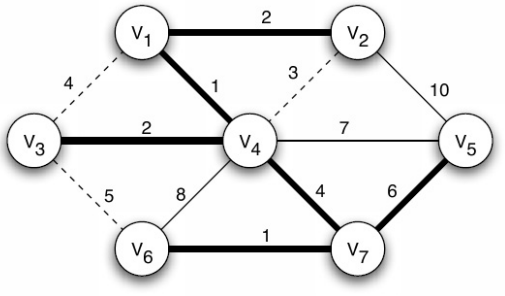


Vi har en kant med vekt 3 — Den vil lage en løkke og må forkastes

Vi har to kanter med vekt 4 hvor den mellom $\mathbf{v_1}$ og $\mathbf{v_3}$ 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 $|\mathbf{V}| - \mathbf{1}$ kanter i spenn-treet og kan slutte

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

Tidsanalyse:

- Hovedløkken går $|\mathbf{E}|$ ganger
- I hver iterasjon gjøres en **deleteMin**, to **find** og en **union**, med samlet tidsforbruk

$$\mathcal{O}(\log|\mathbf{E}|) + \mathbf{2}\cdot\mathcal{O}(\log|\mathbf{V}|)+\mathcal{O}(\mathbf{1}) = \mathcal{O}(\log|\mathbf{V}|)$$

(fordi $\log|\mathbf{E}| < \mathbf{2}\cdot\log|\mathbf{V}|$)

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

Prim vs. Kruskal

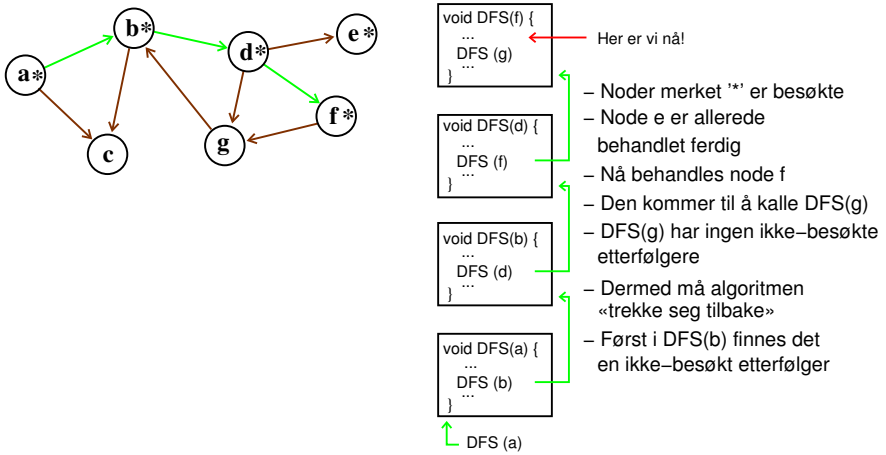
- Prims algoritme er noe mer effektiv enn Kruskals, spesielt for tette grafer
- Prims algoritme virker bare i sammenhengende grafer
- Kruskas algoritme gir et minimalt spenn-tre i hver sammenhengs-komponent i grafen

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

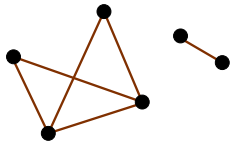

Midtveis i en dybde-først traversering kan kjeden av rekursive metodekall og besøkt-merkene i nodene se slik ut:



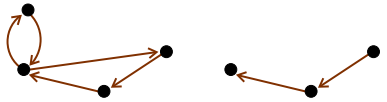
Er grafen sammenhengende?

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

- En urettet graf er sammenhengende hvis og bare hvis et dybde-først søk som starter i en tilfeldig node, besøker alle nodene i grafen



- En rettet graf er sterkt sammen-hengende hvis og bare hvis vi fra hver eneste node **v** klarer å besøke alle de andre nodene 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)

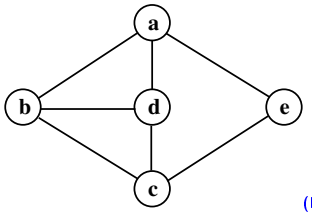
```
void løkkeLet(Node v) {
    if (v.tilstand == igang) {
        < Løkke er funnet >
    } else if (v.tilstand == usett) {
        v.tilstand = igang;
        for < hver nabo w til v > {
            løkkeLet(w);
        }
        v.tilstand = ferdig;
    }
}
```

- 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!

- 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 ...

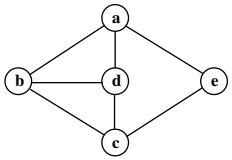
Dobbeltsammenheng (biconnectivity)



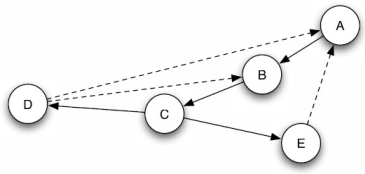
(Figur 9.60, side 357 i MAW)

En *dobbeltsammenhengende graf* er en graf som fortsatt er sammenhengende selv om en vilkårlig node blir fjernet fra grafen

Viktig egenskap i mange nettverk (som strøm- , vei- , data- og internett)



En dybde-først traversering av grafen i leksikografisk ordning ser slik ut:

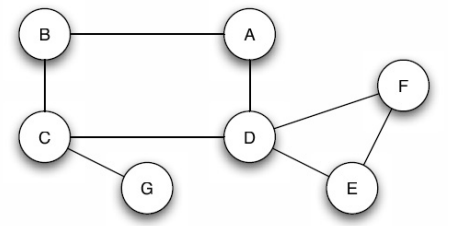


Vanlige rekursive kall (heltrukne linjer) som danner et spenntre for grafen

Stiplede linjer viser «bakoverkall», dvs. forsøk på kall av merkede noder
Disse kalles *bakoverkanter* i grafen

Sammenhengende grafer som ikke er dobbeltsammenhengende, må ha minst en «ledd-node» ("articulation point"), dvs. en node vi ikke kan fjerne uten å gjøre grafen ikke-sammenhengende

Eksempel (fig. 9.62 i MAW):



Her har vi to ledd-noder: C og D

Vi ser at ved å fjerne C mister G sin forbindelse til grafen, og ved å fjerne D, får vi to sammenhengskomponenter, $\{A, B, C, G\}$ og $\{E, F\}$

Algoritme for å finne ledd-noder

Forutsetning: Vi har en sammenhengende graf G

- Gjør et dybde-først søk i G og gi hver node v et besøksnummer $Nr(v)$ (prefix-ordning)

  Vi har nå et spenntre T for G (med bakoverkanter) hvor $Nr(v)$ øker når vi går nedover i T
- For hver node v i T definerer vi $Lav(v)$ som den lavest nummererte node som kan nåes fra v ved å gå null eller flere kanter utover i T og til slutt muligens tilbake langs én bakoverkant

  Mer presist er $Lav(v)$ definert som minimum av
  - $Nr(v)$
  - Minste $Nr(w)$ der (v, w) er bakoverkant i G (dvs. en kant i $G \setminus T$)
  - Minste $Lav(w)$ der (v, w) er kant i T

  Merk at idet vi er ferdig med v, kjenner vi $Lav(v)$ slik at vi får merket nodene med $Lav(v)$ i postfix-orden

- Fra verdiene av $Nr(v)$ og $Lav(v)$ kan vi nå avgjøre hvilke av i nodene i T som er ledd-noder

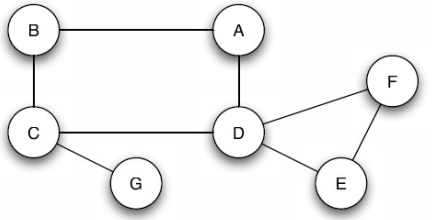
  Vi har to tilfeller:
  1. Rotnoden er en ledd-node hvis, og bare hvis, den har mer enn ett barn
  2. Øvrige noder v er ledd-noder hvis, og bare hvis, v har et barn w med $Lav(w) \geq Nr(v)$

  Rotnoden tilfredsstiller alltid det andre kriteriet,
  så den må behandles spesielt

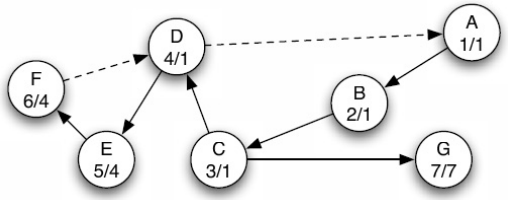
Merk at vi bare trenger én traversering for å utføre hele algoritmen, så tidsordenen blir $\mathcal{O}(|V| + |E|)$

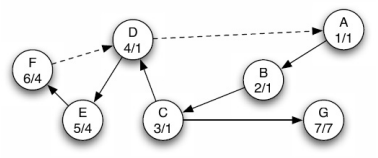
Det følgende eksemplet viser hvordan og hvorfor algoritmen virker

Eksempel



Vi utfører algoritmen og merker hver node v med $Nr(v)/Lav(v)$





Vi ser på de to tilfellene:

1. Hvis roten har flere barn, kan vi ikke fjerne roten uten å splitte grafen

   Det er ikke tilfelle her

2. Vi ser at node D har et barn E med $Lav(E) \geq Nr(D)$

   Det betyr at vi ikke kan komme fra E til noen node v med lavere nummer enn D uten å gå gjennom D

   Altså er D en ledd-node

   Tilsvarende har node C et barn G med $Lav(G) \geq Nr(C)$

   Altså er også C en ledd-node

Se også figur 9.64 på side 361 i MAW