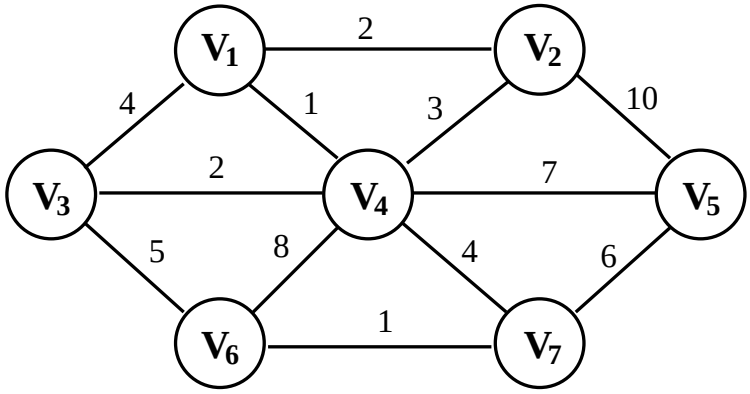

# GRAFER

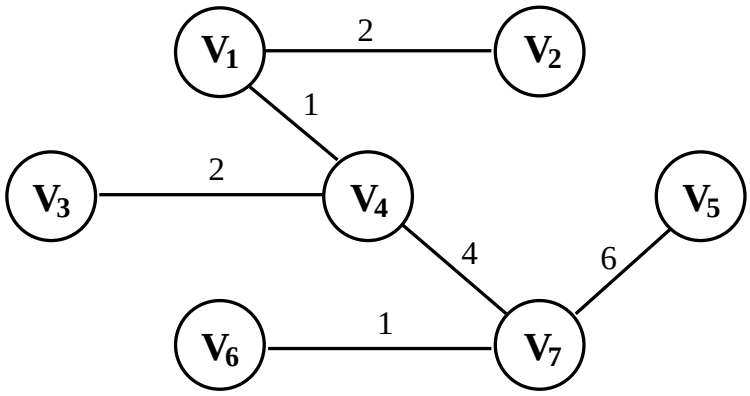
Dagens plan:

- Minimale spenntrær
  - — Prim (Kapittel 9.5.1)
  - — Kruskal (Kapittel 9.5.2)
- Dybde-først søk (Kapittel 9.6.1)
  - — Løkkeleting
- Dobbeltsammenhengende grafer (Kapittel 9.6.2)
  - — Å finne ledd-noder (articulation points)

# Minimalt spenntre

- 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.

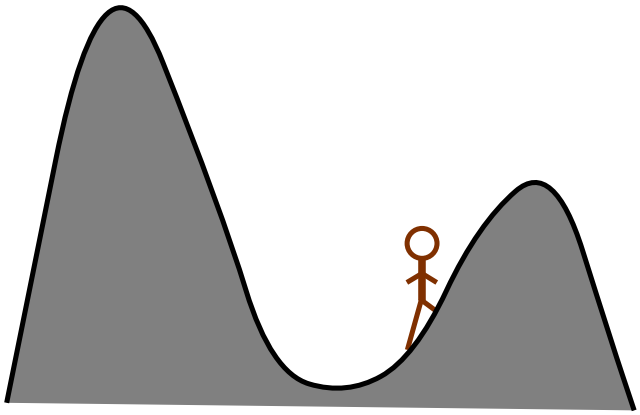



??? *Hvor mange kanter får spenntreet i det generelle tilfellet?*

# 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:

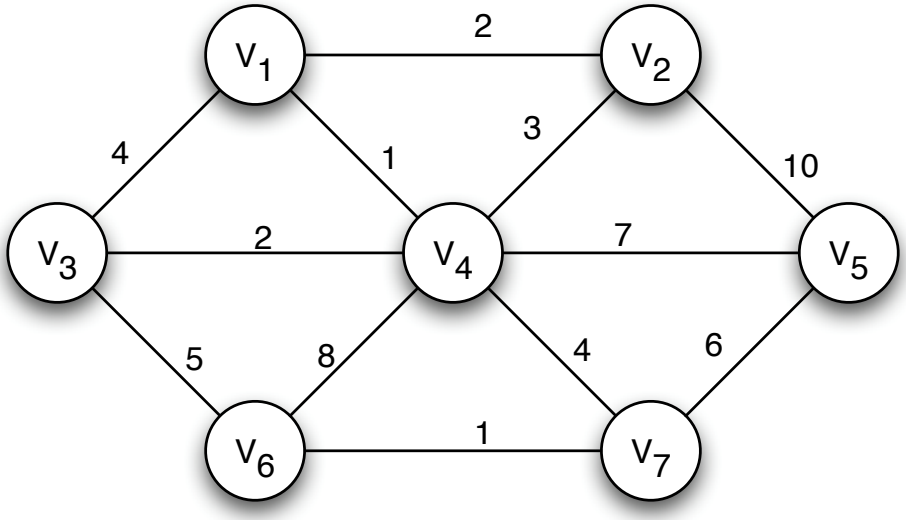


Finn det høyeste punktet!

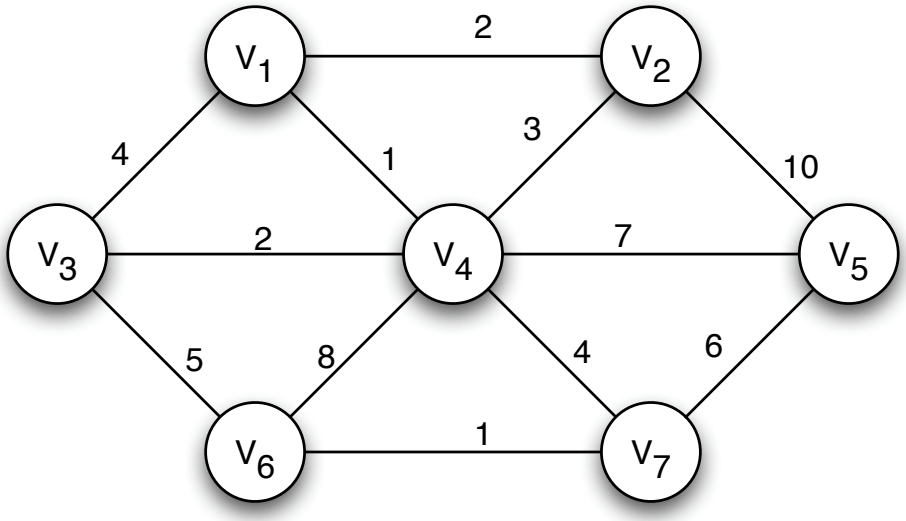
Vi skal se på to ulike grådige algoritmer for å finne minimale spenntrær.

# Prims algoritme

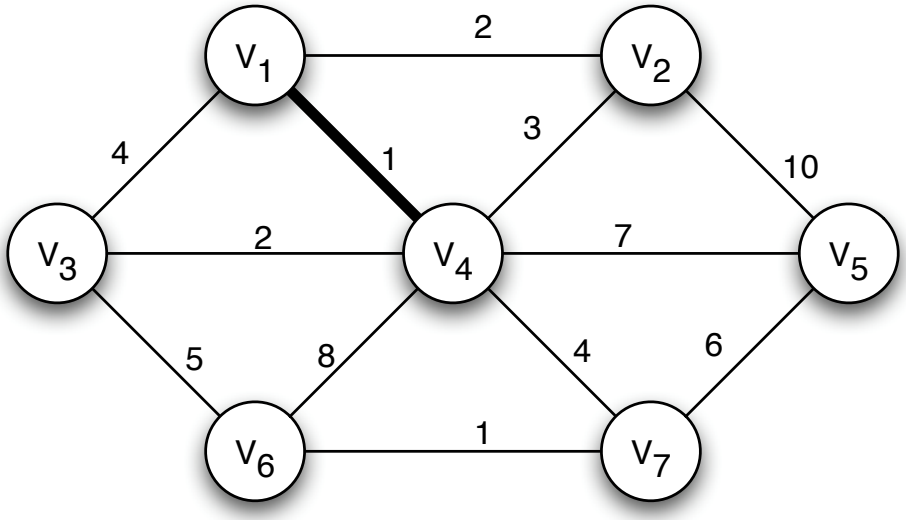
- Treet bygges opp trinnvis. I hvert trinn legges en kant (og dermed en tilhørende node) til treet.
- På ethvert tidspunkt har vi to typer noder: De som er med i treet, og de som ikke er det.
- Nye noder legges til ved å velge en kant $(u, v)$ med **minst** vekt slik at $u$ er med i treet, og $v$ ikke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

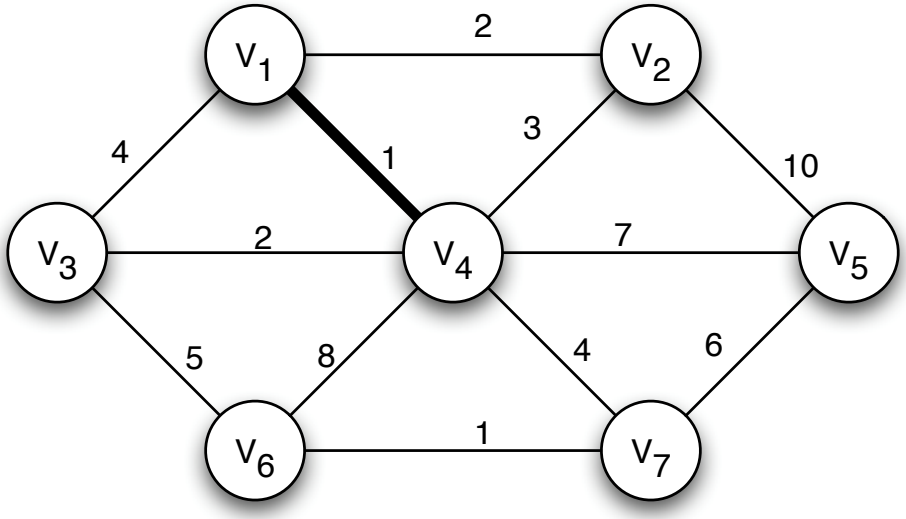



- Vi begynner med å velge en vilkårlig node. La oss begynne med $v_1$.

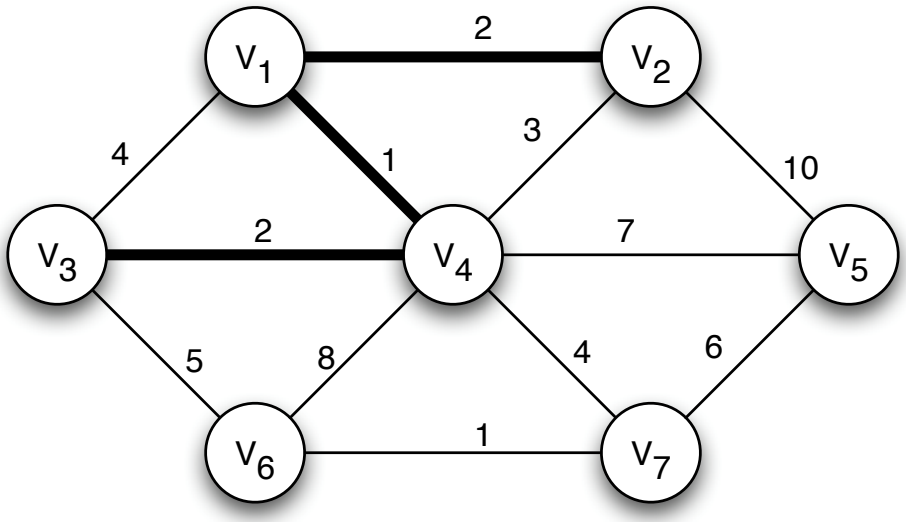


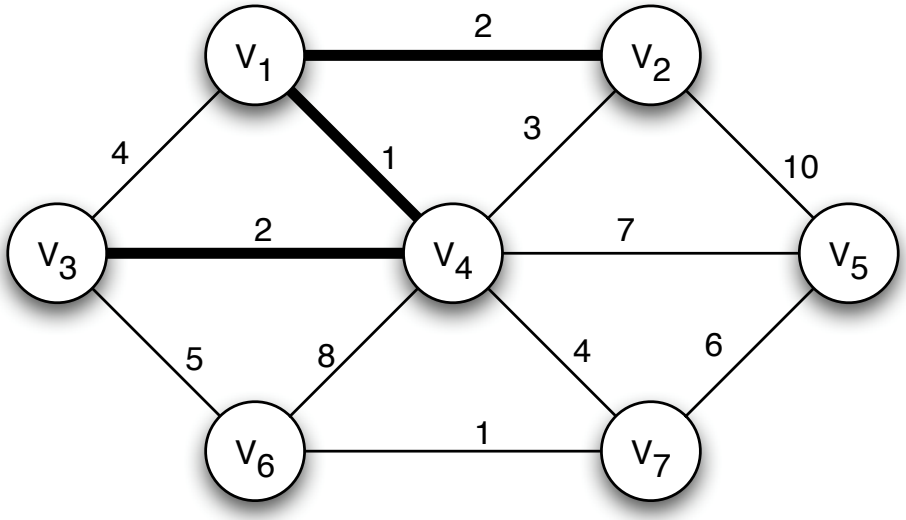

- Minste kant ut fra $v_1$ går til $v_4$, så vi legger den inn i spenntreet.



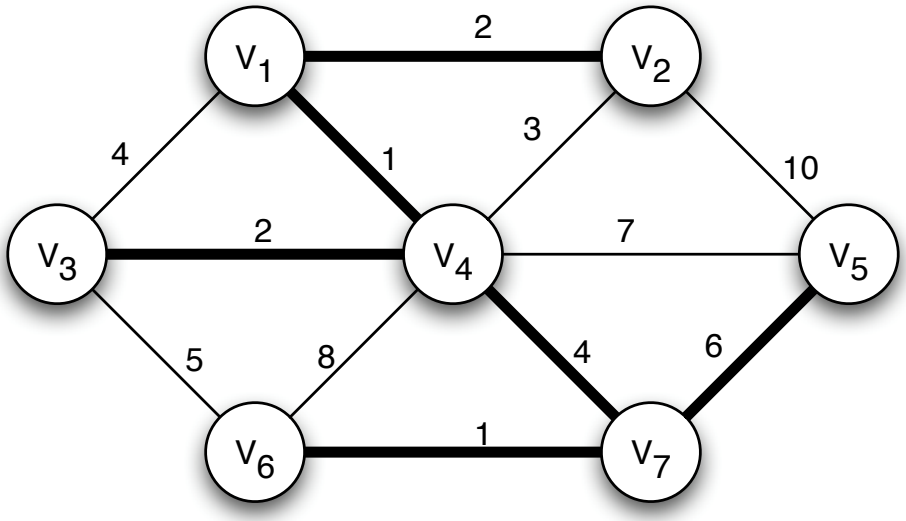



- Vi har nå to mulige forsettelser:
  - kanten fra $v_1$ til $v_2$,
  - eller kanten fra $v_4$ til $v_3$.
- Samme hvilken vi velger, vil den andre av dem bli neste kant, så vi legger dem begge inn i spenntreet.

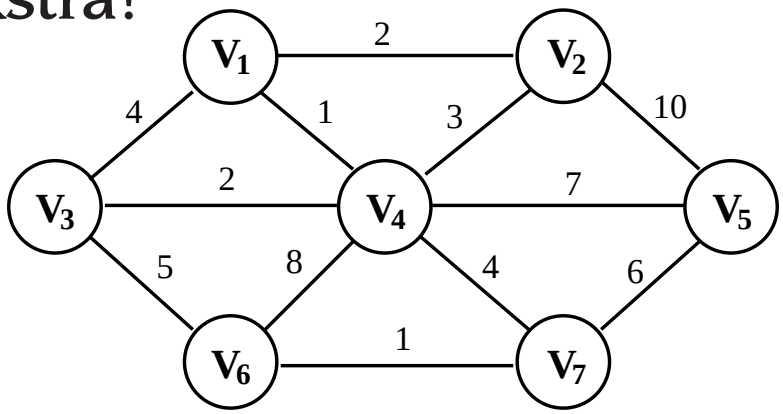




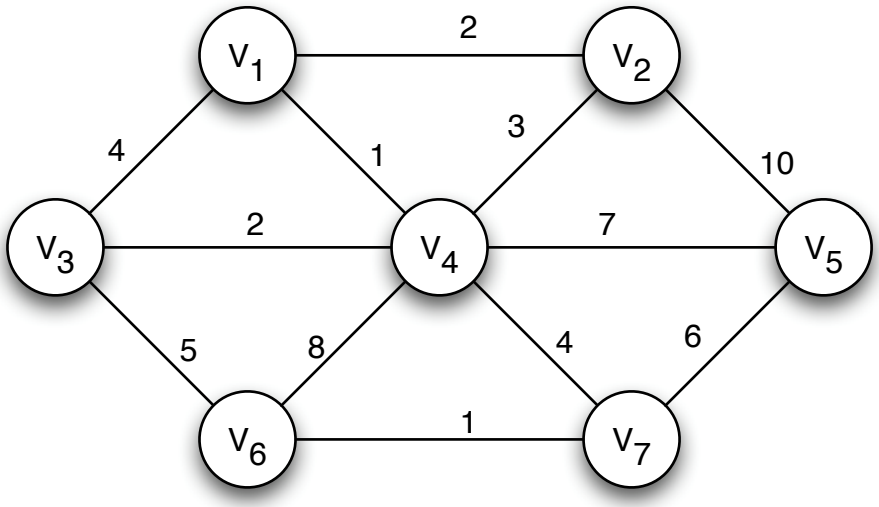

- Minste kant ut fra spenntreet går nå fra $v_4$ til $v_7$.
- Så får vi kanten fra $v_7$ til $v_6$.
- Endelig får vi kanten fra $v_7$ til $v_5$.
- Det ferdige spenntreet blir:



- Prims algoritme er essensielt lik Dijkstras algoritme for å finne korteste vei!
  - «avstanden» til en node $v$: den minste vekten til en kant som forbinder $v$ med en **kjent** node.
- Husk at vi har urettede grafer, slik at hver kant befinner seg i **to** nabolister.
- Samme kjøretidsanalyse som for Dijkstra!




| 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 |

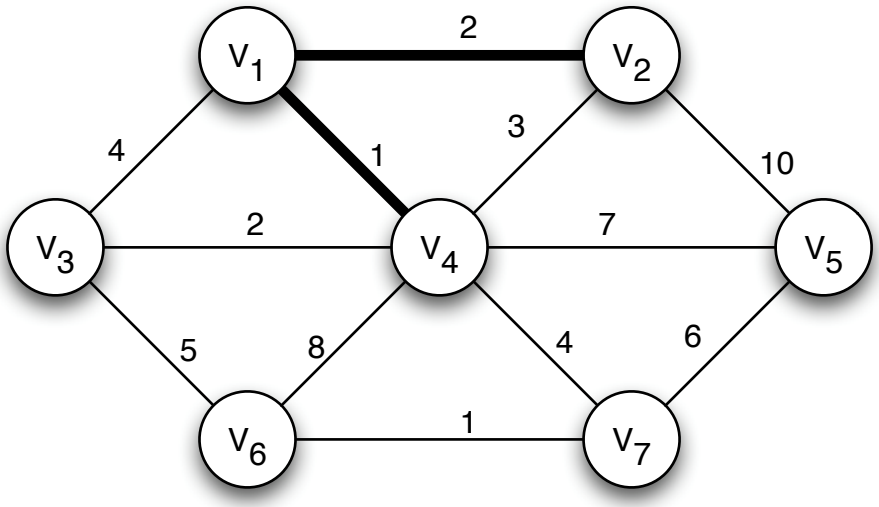
$V_1$ er lagt inn

| 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

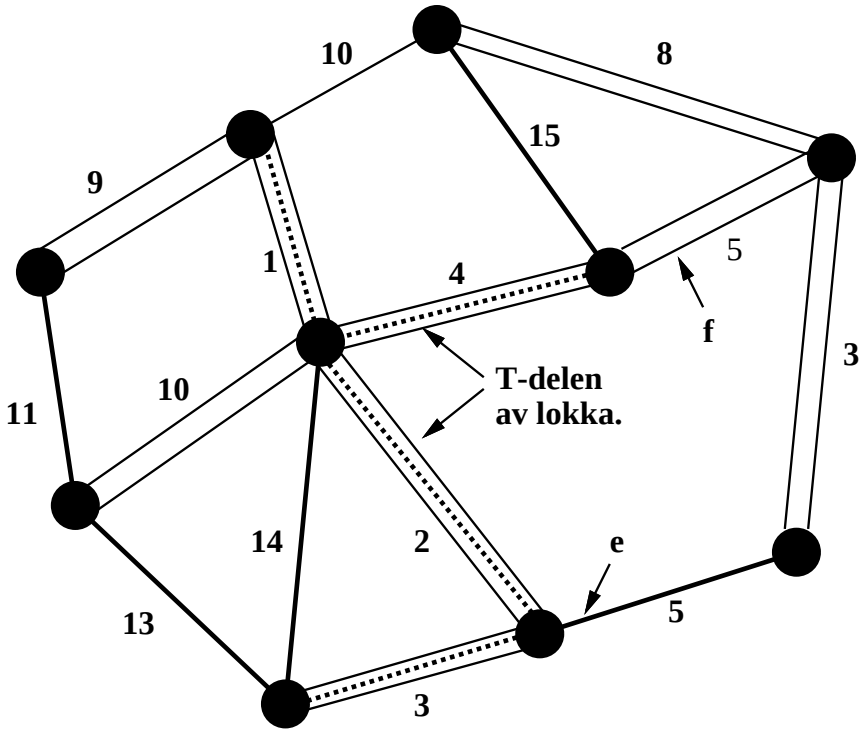
| node | kjent | avst. | fra |
|---|---|---|---|
| $V_1$ | T | 0 | 0 |
| $V_2$ | T | 2 | $V_1$ |
| $V_3$ | T | 2 | $V_4$ |
| $V_4$ | T | 1 | $V_1$ |
| $V_5$ | T | 6 | $V_7$ |
| $V_6$ | T | 1 | $V_7$ |
| $V_7$ | T | 4 | $V_4$ |

$V_5$ er lagt inn. Ferdig!

# Hvorfor virker Prim?

## Løkke-lemmaet:

Anta at $U$ er et spenntre for en graf, og at kanten $e$ ikke er med i treet $U$.
Om vi legger kanten $e$ til treet $U$, vil det 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.




## Prim-invarianten:

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

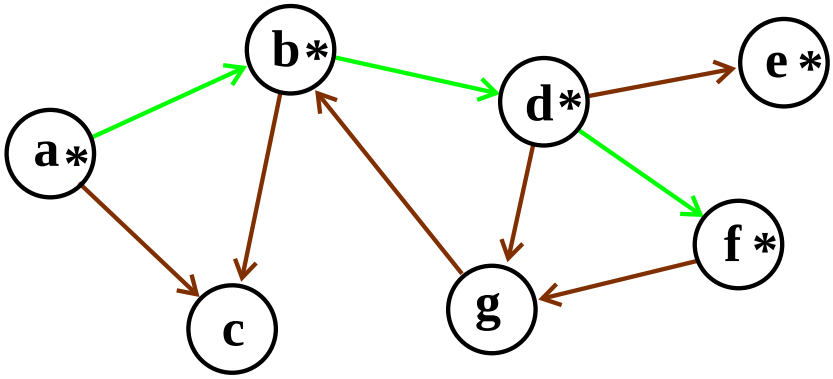
- Ser på kantene en etter en, sortert etter minst vekt:
  - En kant aksepteres hvis den ikke fører til noen løkke.
- Kruskals algoritme opprettholder dermed en skog, (en **samling** trær):
  - Initielt: $|V|$ trær med en node hver.
  - Legger til en kant: To trær slås sammen.
  - Ved terminering: Bare ett tre.
- Bruker disjunkte sett (kapittel 8):
  - Invariant: To noder tilhører samme sett hviss de er sammenhengende i den nåværende spenn-skogen.
  - Å velge en kant $(u, v)$ tilsvarer å gjøre en union på $u$ og $v$.
- Sorteringen av kantene gjøres mest effektivt ved å bruke en prioritetskø. Gjentatte **deleteMin** gir da kantene i den rekkefølgen de skal testes.

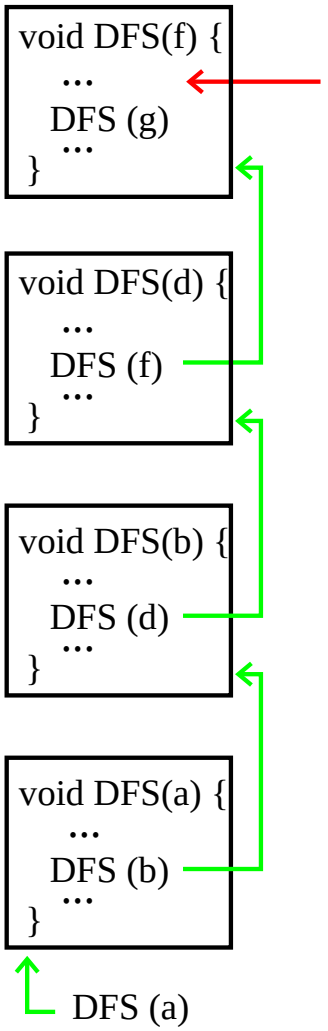
# Dybde-først søk

- Generalisering av prefiks traversering for trær.
- Vi starter i en node $v$ og traverserer rekursivt alle nabonodene.
- 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 ser kanskje kjeden av rekursive metodekall og besøkt-merkene i nodene slik ut:



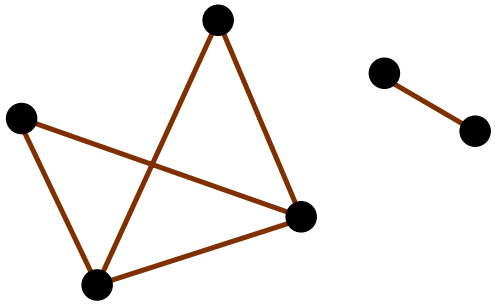




- Noder merket '*' er besøkte.
- Node e er allerede behandlet ferdig.
- Nå behandles node f
- Den kommer til å kalle DFS(g)
- DFS(g) har ingen ikke-besøkte etterfølgere.
- Dermed må algoritmen "trekke seg tilbake".
- Først i DFS(b) finnes det en ikke-besøkt etterfølger.

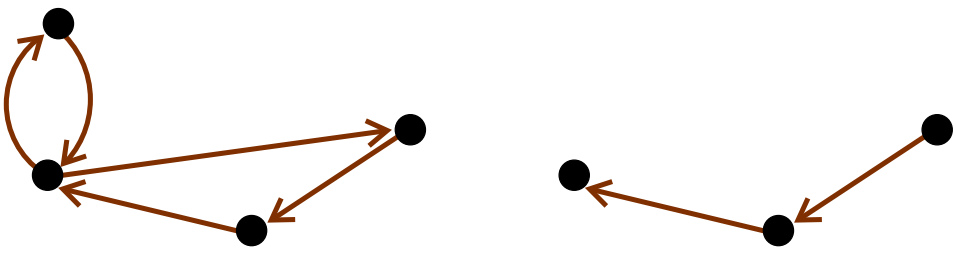
# Er grafen sammenhengende?

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) sammenhengende 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.
- Vi trenger da tre verdier til tilstandsvariablen:
  usett, igang og ferdig (besøkt).

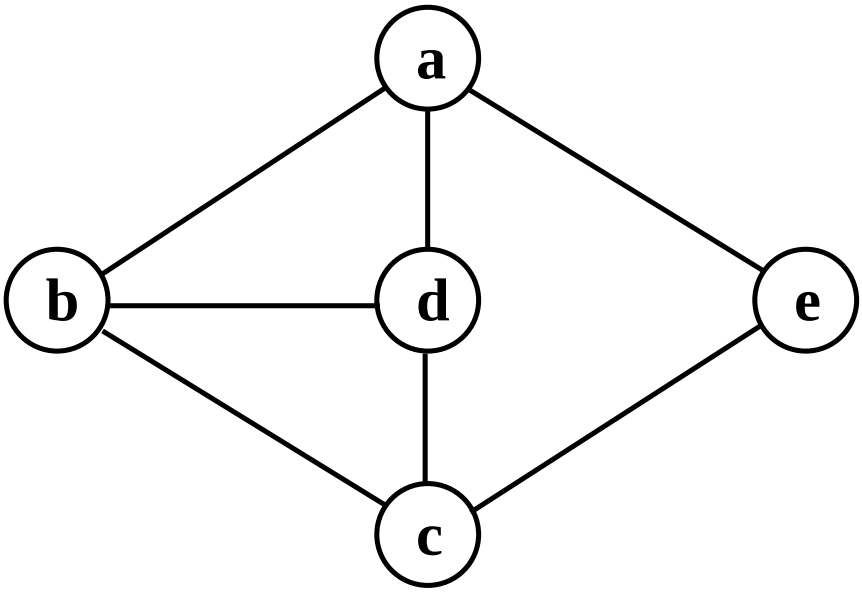
```
void løkkeLet(Node v) {
    if (v.tilstand == igang) {
        < Løkke er funnet >
    } else if (v.tilstand == usett) {
        v.tilstand = igang;
        for < hver nabo w til v > {
            løkkeLet(w);
        }
        v.tilstand = ferdig;
    }
}
```

- Metoden bygger på at de nodene der kall er i gang, alltid ligger på en rett vei fra startnoden.
- Må passe på å gjøre nye startkall inntil metoden er kalt i alle nodene.
- 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 ...

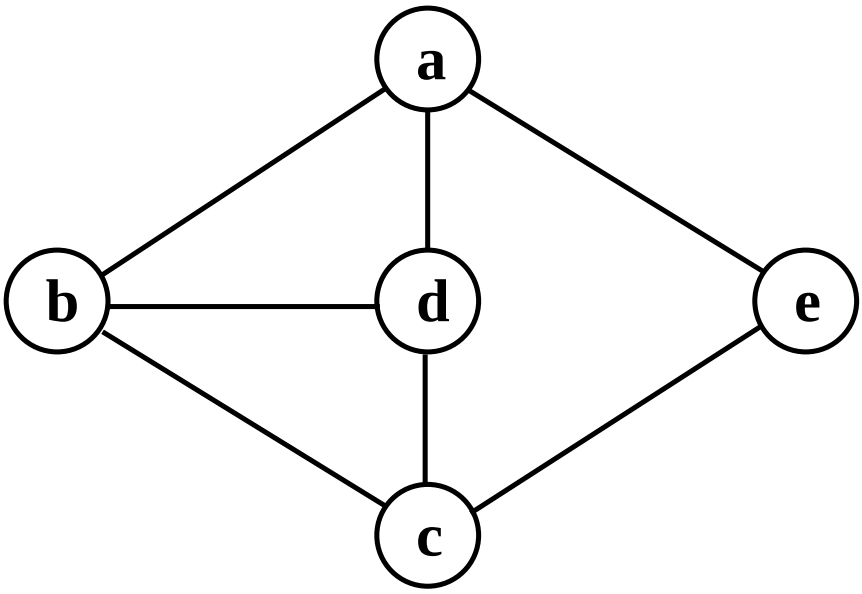
# Dobbeltsammenheng (biconnectivity)



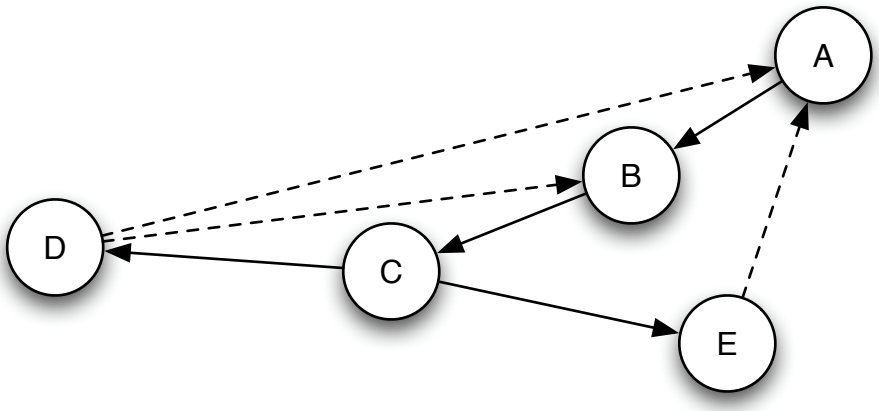

**Figur 9.60, side 326 i MAW**

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

Dette er en viktig egenskap i mange nettverk (som strøm- , vei- , data- og internett)




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

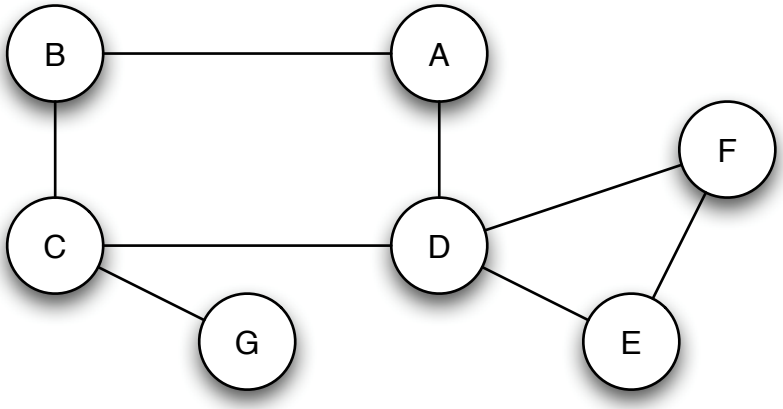



Heltrukne linjer viser vanlige rekursive kall. De danner et spenntre for grafen.

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 $T$ (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.

- Ledd-nodene kan nå leses ut av verdiene av $Nr(v)$ og $Lav(v)$ i nodene i $T$

  Vi har to tilfeller:

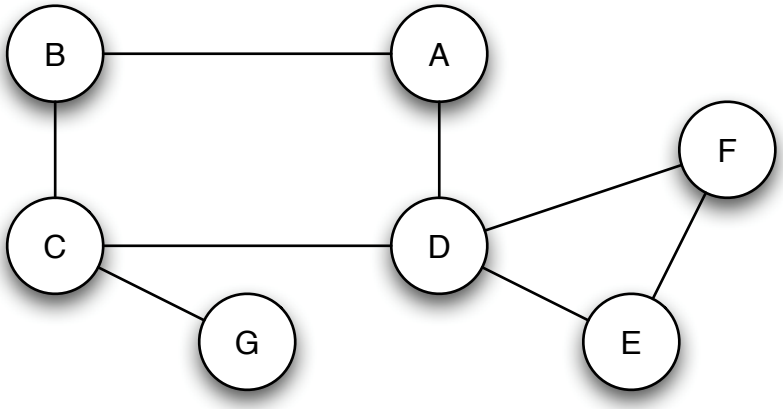
  — Rotnoden er en ledd-node hvis, og bare hvis, den har mer enn ett barn.

  — Øvrige noder $v$ er ledd-noder hvis, og bare hvis, $v$ har et barn $w$ med $Lav(w) \geq Nr(v)$.

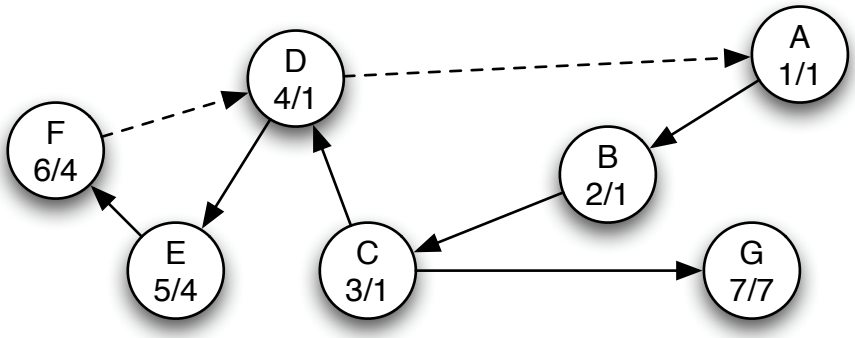
  Rotnoden tilfredsstiller alltid det andre kriteriet, så den må behandles spesielt.

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

Eksemplet på neste lysark viser hvordan og hvorfor algoritmen virker.




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




Vi ser på de to tilfellene:

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

Se også figur 9.64 i MAW.