INF2220 - Algoritmer og datastrukturer

HØSTEN 2016

Institutt for informatikk, Universitetet i Oslo

Forelesning 6:

Grafer II

Dagens plan:

Dijkstra fort.
Minimale spenntrær

- Prim
- Kruskal

Dybde-først søk

- Løkkeleting
- DFS Spenntre

Dijkstra (fort.)

Dijkstras algoritme

1. For alle noder:

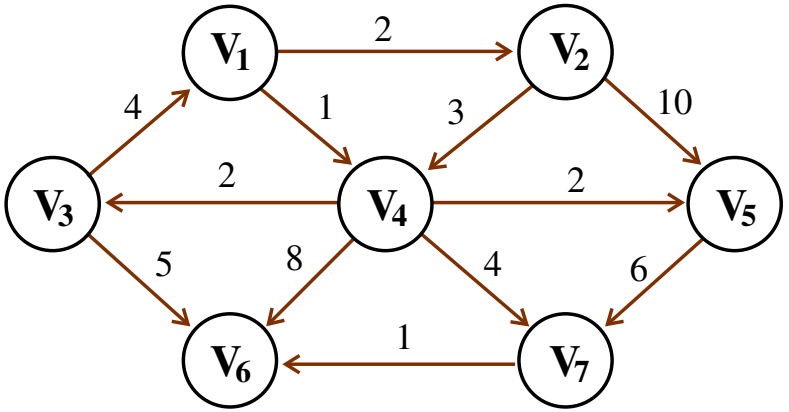
   Sett avstanden fra startnoden **s** lik ∞. Merk noden som **ukjent**
2. Sett avstanden fra **s** til seg selv lik **0**
3. Velg en ukjent node **v** med minimal (aktuell) avstand fra **s** og marker **v** som **kjent**
4. For hver ukjent nabonode **w** til **v**:

   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. Akkurat som for uvektede grafer, ser vi bare etter potensielle forbedringer for naboer (w) som ennå ikke er valgt (kjent)

$$\begin{array}{lllll} \text{uvektet:} & d_w & = & d_v + 1 & \text{hvis} \quad d_w = \infty \\ \text{vektet:} & d_w & = & d_v + c_{v,w} & \text{hvis} \quad d_v + c_{v,w} < d_w \end{array}$$

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

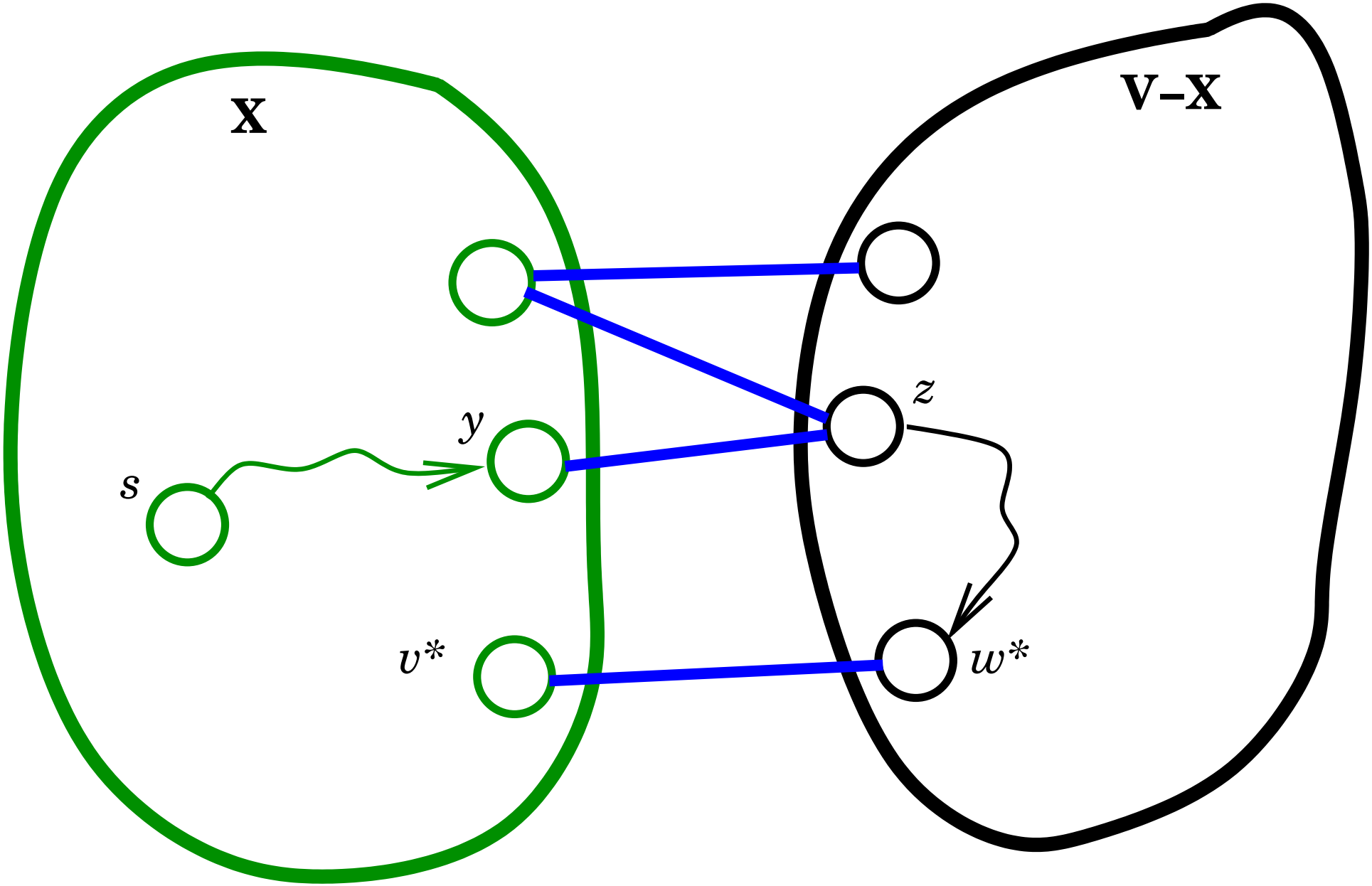
v	kjent	avstand	vei
v_1			
v_2			
v_3			
v_4			
v_5			
v_6			
v_7			

Hvorfor virker algoritmen?

Invariant

- Ingen kjent node har større avstand til **s** enn en ukjent node
- Alle kjente noder har riktig korteste vei satt

- 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 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!


X
V–X
s
y
z
v*
w*

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 $\mathcal{O}(|\mathbf{E}|)$

Totalt tidsforbruk

$$\mathcal{O}(|\mathbf{E}| + |\mathbf{V}|^2) = \mathcal{O}(|\mathbf{V}|^2)$$

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 bruker $\mathcal{O}(\log |\mathbf{V}|)$ tid (kap. 6)

Totalt tidsforbruk

Hva 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:
  1. Ta ut en node **v** fra køen
  2. For hver etterfølger **w**, sjekk om vi får en forbedring ($\mathbf{d_w} > \mathbf{d_v} + \mathbf{c_{v,w}}$)
  3. Oppdater i så fall avstanden, og plasser **w** (tilbake)! i køen (hvis den ikke er der allerede)
- Tidsforbruket blir $\mathcal{O}(|\mathbf{E}| \cdot |\mathbf{V}|)$
- 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!

Hva med asykliske grafer?

- Lineær tid ved å behandle nodene i en topologisk rekkefølge $\mathcal{O}(|\mathbf{E}| + |\mathbf{V}|)$
- når en node $\mathbf{v}$ er valgt, kan $\mathbf{d_v}$ ikke lenger senkes siden det er ingen innkommende kanter som kommer fra ukjente noder

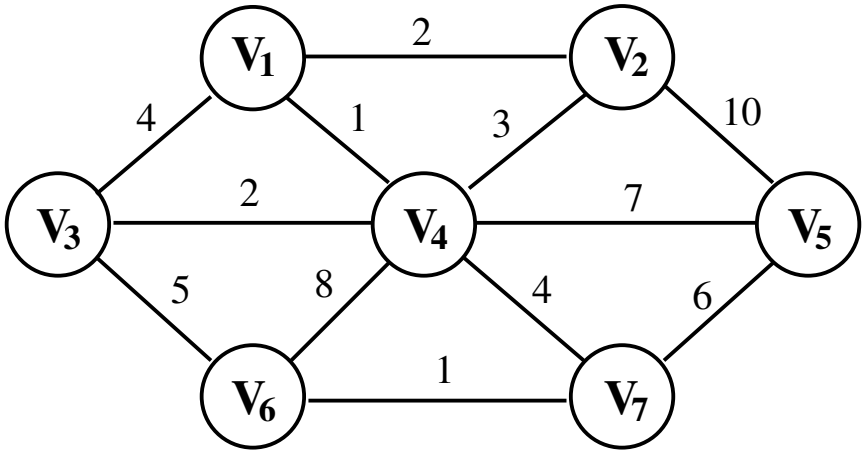
Minimale spenntrær

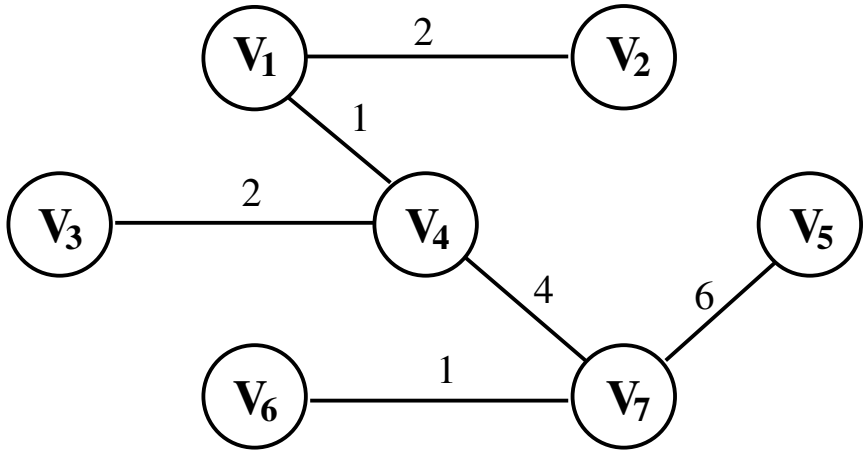
Minimale spenntrær

Definition

Et minimalt spenntre for en **urettet** graf **G** er et tre med kanter fra grafen, slik at alle nodene i **G** er forbundet til lavest mulig kostnad

- eksisterer bare for sammenhengende grafer
- Generelt: flere minimale spenntrær for samme graf



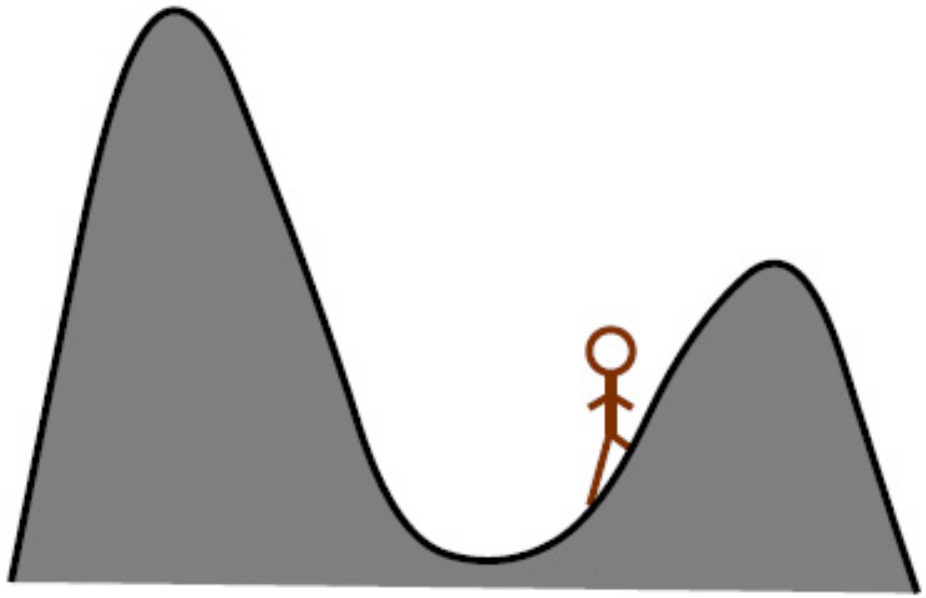


Hvor mange kanter får spenntreet?

Grådige algoritmer

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:



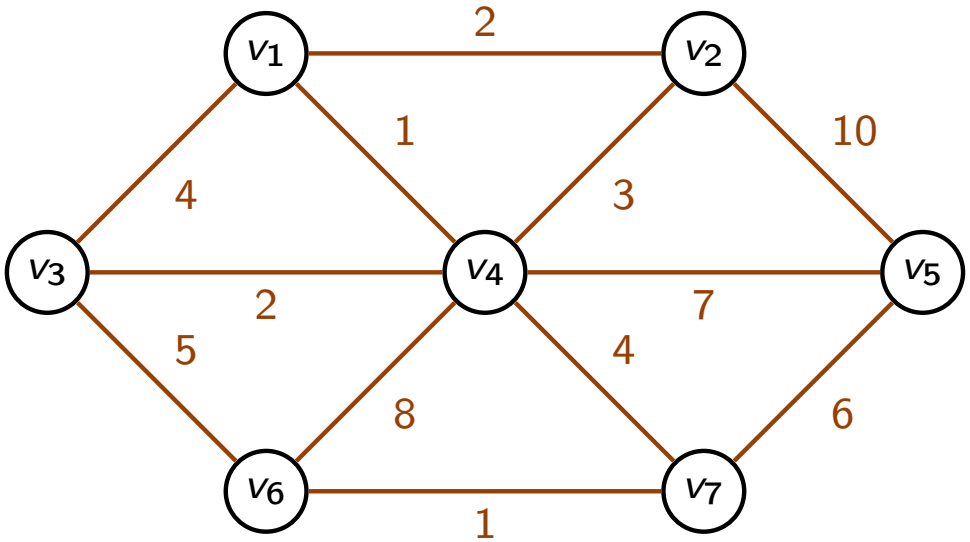
Finn det høyeste punktet!

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

Prims algoritme

- Treet bygges opp trinnvis

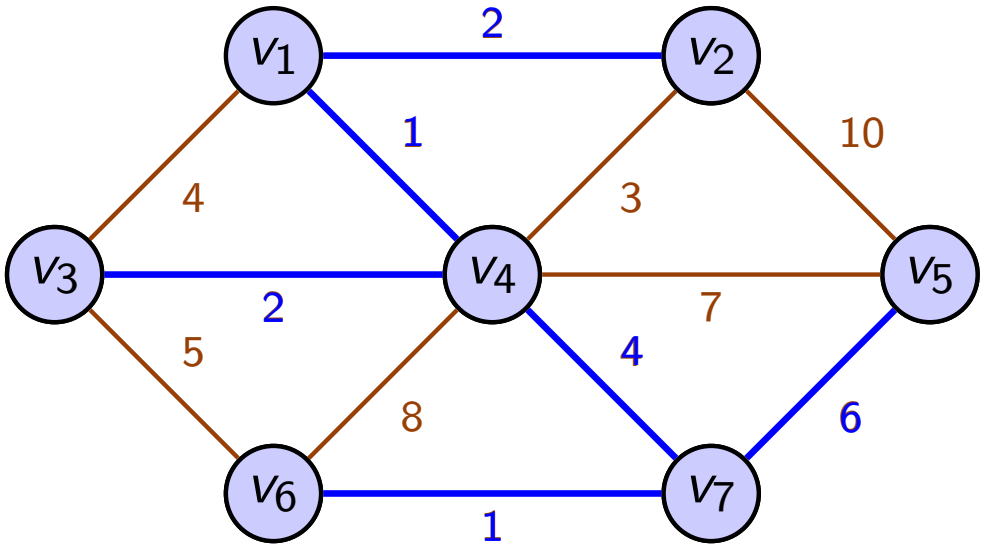
  I hvert trinn: pluss en kant (og dermed en tilhørende node) til treet
- 2 typer noder:
  - Noder som er med i treet
  - Noder som ikke er med i treet
- Nye noder: velge en kant $(\mathbf{u}, \mathbf{v})$ med minst vekt slik at $\mathbf{u}$ er med i treet, og $\mathbf{v}$ ikke er det.



- Algoritmen begynner med å velge en vilkårlig node

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

Prim: growing a tree (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: essensielt lik Dijkstras algo 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: vi har urettede grafer, så hver kant befinner seg i **2** nabolister
- Kjøretidsanalysen er den samme som for Dijkstras algoritme

Hvorfor virker Prim?

Lemma (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 vi fjerner en vilkårlig kant i denne løkken, vil vi igjen ha et spenntre for grafen

Invariant

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 (slik at **find**(**v**) gir mengden til (**v**)).

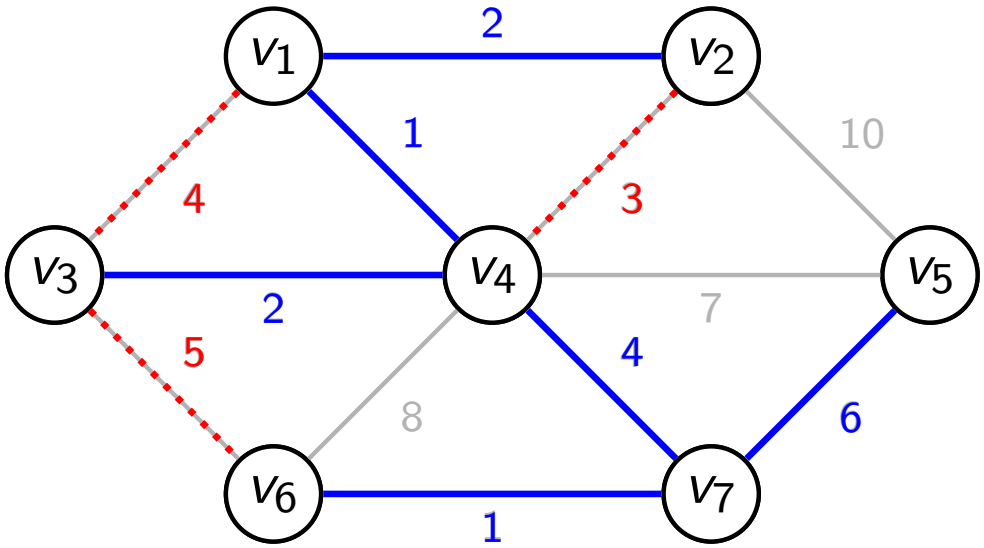
  Invariant:

  De disjunkte mengdene er subtrær av det endelige spenntreet

- **deleteMin** 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 $|\mathbf{V}| - \mathbf{1}$ kanter

Growing a forest (1)

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 kant med vekt 3: — Den vil lage en løkke og må forkastes 2 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 kant med vekt 5: løkke kant med vekt 6: knytter den siste noden til treet

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

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 sammenhengskomponent i grafen

Dybde-først søk

Dybde-først søk

- klassisk graf traversering
- generalisering av prefiks traversering for trær
- gitt: start node v: rekursivt traverserer alle nabonodene
- rekursjon $\Rightarrow$ vi undersøker alle noder som kan nåes fra første etterfølger til **v**, før vi undersøker *neste* etterfølger til **v**
- for vilkårlige grafer: pass på å terminerer rekursjon!

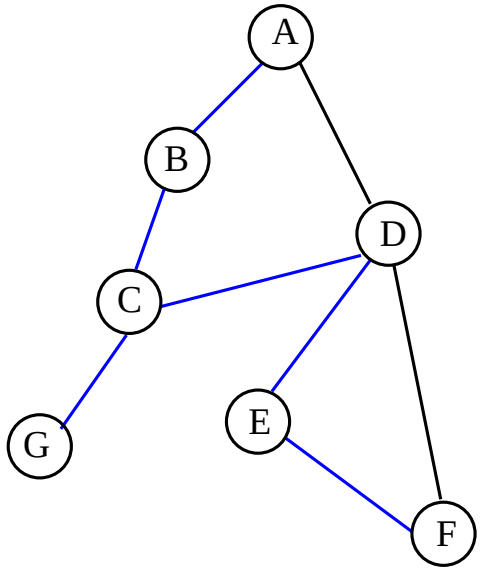
$\Rightarrow$ unvisited $\leftrightarrow$ visited nodes.

DFS

Initialize: all nodes "unvisited".

Recur:
- when visited: return immidiately
- when unvisited
  - set to "visited"
  - recur on all neighbors

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

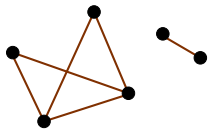


Løkkeleting

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

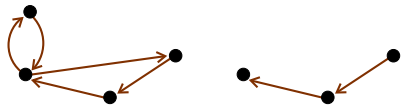
```
void løkkeLet(Node v) {
    if (v.tilstand == igang) {
        < Løkke er funnet >
    } else if (v.tilstand == usett) {
        v.tilstand = igang;
        for < hver nabo w til v > {
            løkkeLet(w);
        }
        v.tilstand = ferdig;
    }
}
```

Er grafen sammenhengende?



urettet graf & DFS

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



rettet graf & DFS

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**

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

Urettet grafer

Dybde-først-spenntre

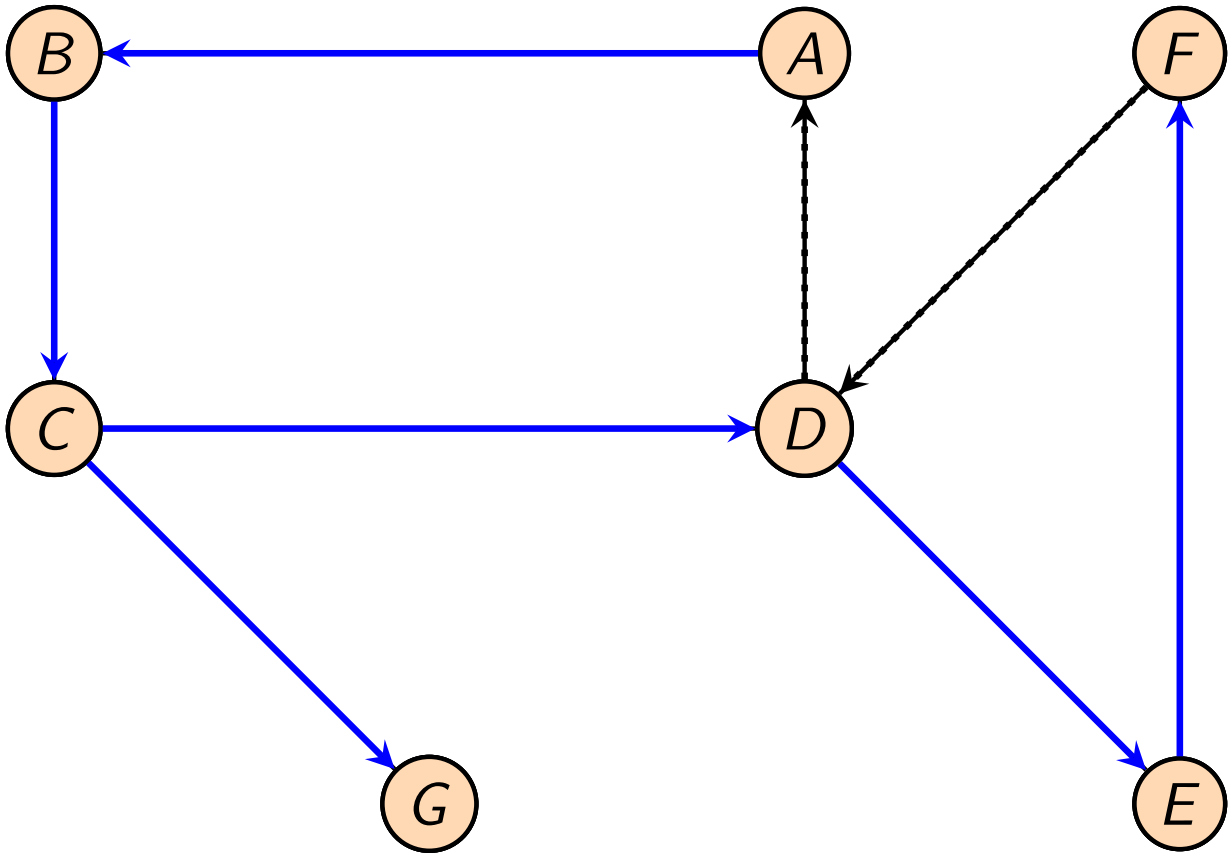
- for urettet sammenhengende grafer
- *ikke sammenhengende* grafer: dfs spanning forest
- huske "back-pointers"

Ulike type kanter

gitt en bestemt kjøring av dfs

1. tree edges
2. back edges

DFS, urettet graf (1)



Neste forelesning: 5. oktober

Grafer III:

- Biconnectivity
- Stongly connected components