Uke 9,
Forelesning 2

Korteste vei algoritmer:

- Uvektet graf
- Vektet graf

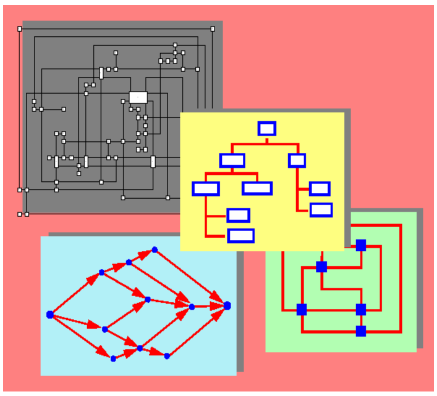
Begge kan være i en rettet eller urettet versjon

Vi fortsetter med korteste vei, én-til-alle for:

- Vektet graf med negative kanter (kapittel 9.3.3)

Vi tar opp:

- Activity graphs (9.3.4, 9.3.5)
- Depth-first search (9.6, 9.6.1)
- Finding cycles



Fortsetter med Grafer…

Korteste vei i en uvektet graf (repetisjon)

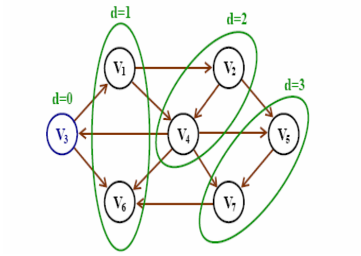
Korteste vei fra s til t i en uvektet graf er lik veien som bruker færrest antall kanter (alle kanter har vekt=1)

Vi finner den korteste veien fra s til alle andre noder ved først å markere at veien fra s til s har lengde 0.

Deretter markerer vi at etterfølgerne til s er på avstand 1, at de umarkerte etterfølgerne til disse er på avstand 2, at de umarkerte etterfølgerne til disse er på avstand 3, osv.

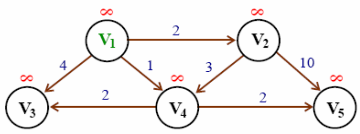
Dette er en såkalt **bredde-først** algoritme fordi vi først markerer alle som er på avstand 1, deretter alle som er på avstand 2, osv.

Det lønner seg å plassere etterfølgerne til noden vi markerer i en **kø**, i stedet for å å søke sekvensielt gjennom alle nodene for å finne neste node som skal behandles.

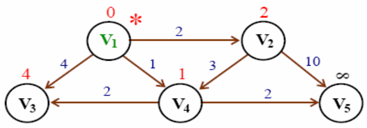


Korteste vei i en vektet graf (repetisjon)

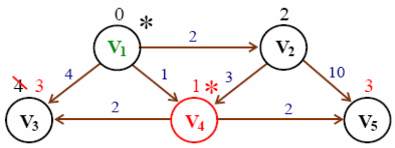
- Vi bruker de samme ideene som i en graf uten vekter.
- Initielt har alle noder avstand 'uendelig'.



- Vi starter med å makrere s som kjent og med avstand lik 0. Vi markerer at alle etterfølgere w til s har avstand $c_{s,w}$.



Deretter velger algoritmen i hvert steg den noden v som har **minst distanse og som er ukjent**. v markeres som 'kjent', og vi undersøker alle utgående kanter fra v til ukjente noder: Distansene (fra startnoden) til disse nodene oppdateres dersom veien via v gir kortere lengde enn den gamle "besteveien" (som bruker kjente noder, men ikke v).



Hva hvis grafen har negative kanter?

- Dersom den vektede grafen har negative kanter, fungerer ikke Dijkstras algoritme, se oppg. 9.7a denne uka!
- Problemet er at når vi erklærer at v er kjent, så kan det være en **veldig negativ** kant fra en ukjent node til v.
- Dermed kan vi ikke garantere at d_v er den ekte korteste distansen fra s til v.

- En mulig løsning på problemet:
  - — Ikke tenk på 'kjente' eller 'ukjente' noder lenger.
  - — Vi har i stedet en **kø** som inneholder noder som har fått forbedret distanseverdien sin.
  - — Løkken i algoritmen gjør følgende:
    - ∗ Ta ut v fra køen.
    - ∗ For hver etterfølger w, sjekk om $d_v + c_{v,w}$ er en forbedring.
    - ∗ I så fall oppdater d_w og plasser w på køen (hvis den ikke er der allerede).

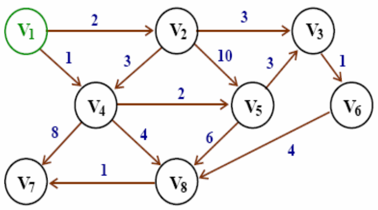
- Tidsforbruket blir da $\mathcal{O}(|E| \cdot |V|)$ som er mye verre enn Dijkstras algoritme.

Algoritmen fungerer ikke hvis det er **negative løkker.**

I dag: Rettede, ikke-sykliske grafer (DAG)

- Dersom vi vet at grafen er ikke-syklisk, kan vi lage en forbedret versjon av Dijkstras algoritme ved å forandre metoden for å velge neste kjente node.
- Den nye regelen er at vi velger nodene i en **topologisk ordning**.
- Når vi velger en node v, vet vi at den har riktig korteste distanse d_v:
  — Distansen d_v kan jo ikke lenger forandres, siden den topologiske ordningen garanterer at noden ikke har inngående kanter fra ukjente noder.
- Algoritmen trenger bare en enkelt gjennomgang av nodene og kantene fordi utvelgelse av nodene og oppdatering av distansene kan utføres samtidig med den topologiske sorteringen.
- Dermed blir tidsforbruket $\mathcal{O}(|V| + |E|)$ fordi utvelgelsen av neste node tar konstant tid ("plukk en node fra boksen med noder uten forgjengere") og hver kant bare blir undersøkt en gang.

Eksempel:



v	kjent	d_v	p_v
v_1			
v_2			
v_3			
v_4			
v_5			
v_6			
v_7			
v_8			

Dersom vi ønsker å ha f.eks. v_2 som startnode, må vi først fjerne v_1 (som ikke kan nås fra v_2) og alle utkanter fra v_1 fra grafen.

Aktivitetsgrafer

- En veldig viktig anvendelse av DAG-er er i **prosjektplanlegging**.
- Vi har en mengde aktiviteter som hver tar en viss tid å gjennomføre.
- I tillegg er aktivitetene avhengige av hverandre, på den måten at noen aktiviteter ikke kan startes før visse andre aktiviteter er avsluttet.

Eksempel: Husbygging

Aktiviteter:

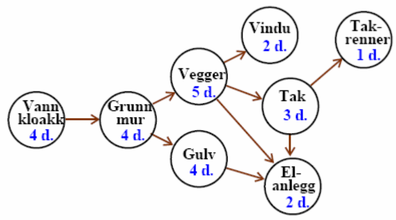
- Grunnmuren tar 4 dager
- Veggene tar 5 dager
- Gulvet tar 2 dager
- Taket tar 3 dager
- Vinduene tar 2 dager
- Det elektrisk anlegget tar 2 dager
- Vann og kloakk tar 4 dager
- Takrennene tar 1 dag

Avhengigheter:

- Grunnmuren avhenger av vann og kloakk
- Veggene avhenger av grunnmuren
- Gulvet avhenger av grunnmuren
- Taket avhenger av veggene
- Vinduene avhenger av veggene
- El-anlegget avhenger av gulv, vegger og tak
- Takrennene avhenger av taket

Mer husbygging

- Disse opplysningene kan overføres til en såkalt **aktivitetsgraf**:
  - Hver aktivitet blir et nodeobjekt.
  - Aktivitetens varighet (lengde) blir en variabel (vekt) i nodeobjektet.
  - Avhengighetene modeleres ved rettede, uvektede kanter:
    - Det går en kant fra node v til node w dersom aktivitet w er direkte avhengig av aktivitet v.



Mer prosjektplanlegging

- Vi ønsker typisk å få svar på følgende spørsmål:
  - Kan prosjektet gjennomføres?
  - Hva er minste totale gjennomførelsestid?
  - Hvilke aktiviteter kan bli forsinket, og med hvor lenge, uten at totaltiden for prosjektet øker?
  - Hvilke aktiveteter er **kritiske**, i betydningen at de **må** gjennomføres på fastsatt tid hvis prosjektet skal bli ferdig i tide?

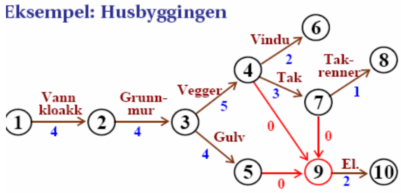
- I den andre obligatoriske oppgaven skal dere implementere algoritmer som svarer på disse spørsmålene for en tilfeldig aktivitetsgraf.
- Vi skal se på en annen metode som går ut på å gjøre om aktivitetsgrafen til en **hendelsesgraf**.
- Hovedpoenget er å flytte vektene fra nodene til kantene, slik at vi kan bruke varianter av korteste-vei algoritmen.

Hendelsesgrafer

- Hver node i en **hendelsesgraf** representerer at en aktivetet (og alle dens forgjengere) i aktivitetsgrafen er avsluttet.
- Hendelser som kan nås fra en node v kan ikke settes i gang før node v er ferdig.
- Aktivtetene er nå representert ved kanter i stedet for ved noder.
- Oversettelsen fra aktivitetsgrafer til hendelsesgrafer kan foregå automatisk eller manuelt (for hånd).

- Dersom en aktivetet er avhengig av flere andre aktiveteter, kan det være nødvendig å legge til **hjelpekanter** (dummy edges) og **hjelpenoder** (dummy nodes) for at grafene skal uttrykke akkurat de samme avhengighetene.

Eksempel: Husbyggingen



- For å finne tidligste avslutningstid for prosjektet, trenger vi bare å finne den **lengste** veien fra den første hendelsen.
- For en generell graf er lengste-vei problemet meningsløst pga. av **positiv-kost-løkker**.

- Hvis det eksisterer positive løkker, kan vi spørre om den lengste **enkle** stien, men vi kjenner ikke noen algoritme som alltid løser det problemet (kalt **Hamiltonicity**) raskt, dvs. i polynomisk tid.
- Siden hendelsesgrafer er ikke-sykliske, behøver vi ikke bekymre oss om positive-løkker.
- Vi kan bruke en modifisert variant av korteste-vei algoritmen.

- Dersom EC_i er **tidligste avslutningstid** for node i, så bruker vi følgende regler for å oppdatere EC-verdiene i de forskjellige nodene:

$$EC_1 = 0$$
$$EC_w = \max_{(v,w)\in E}(EC_v + c_{v,w})$$

- EC_w er lik den lengste veien fra startnoden til w.
- EC-verdiene kan beregnes ved å gå gjennom nodene i topologisk rekkefølge.
- Vi kan også beregne det **seneste tidspunktet**, LC_i, som en node i kan bli ferdig, uten at den forsinker prosjektet:

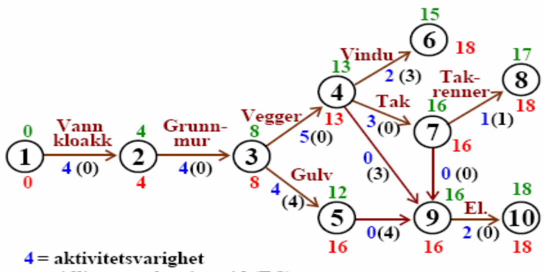
$$LC_n = EC_n$$
$$LC_v = \min_{(v,w)\in E}(LC_w - c_{v,w})$$

- LC-verdiene kan beregnes ved å gå gjennom nodene i omvendt topologisk rekkefølge.
- **Slakken** til en kant i hendelsesgrafen forteller hvor mye den tilhørende aktiviteten kan bli forsinket uten at totaltiden til prosjektet øker. Vi har at:

$$\text{Slack}(v,w) = LC_w - EC_v - c_{c,w}$$

- Formelen sier at slakken til en kant (aktivitet) (v,w) er lik seneste avslutningstidspunkt til w minus tidligste avslutningstidspunkt for v minus tiden som det tar å utføre aktiviteten.

Eksempel: Husbyggingen



4 = aktivitetsvarighet
0 = tidligste avslutningstid (EC)
0 = seneste avslutningstid (LC)
(0) = slakk

- Vi setter LC til 18 for alle "endenoder" (noder uten etterfølgere).

Dybde-først søk

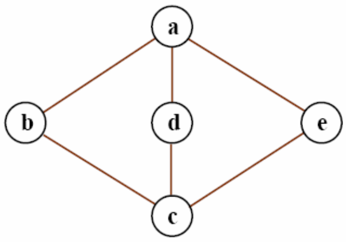
- **Dybde-først søk** er en teknikk for å systematisk undersøke alle nodene i en graf:
  - Vi starter i en node v.
  - Vi markerer v som besøkt og foretar de beregningene vi ønsker å gjøre i noden.
  - Deretter undersøker vi rekursivt alle ikke-besøkte etterfølgere til v.
- Rekursjonsteknikken medfører at vi undersøker alle noder som kan nåes fra første etterfølger til v, før vi undersøker neste etterfølger til v.
- Vi går altså i **dybden først**, i motsetning til ved **bredde-først søk** der alle etterfølgerne blir behandlet før vi behandler noen av etterfølgernes etterfølgere.

•Depth-first Search – se på! Har topologisk sortering, minimum span tre, grådige algoritmer….

- Dybde-først søk ligner på **preorder**-traversering av et tre.
- I dybde-først traversering av en graf må vi passe på å unngå å gå evig rundt og rundt i en løkke i grafen.
- Vi markerer derfor nodene som besøkt etterhvert som vi behandler dem.

- Dersom grafen er urettet og usammenhengende, eller rettet og ikke sterkt sammenhengende, er det ikke sikkert at vi klarer å besøke alle nodene i et enkelt dybde-først søk.
- Vi kan da foreta nye dybde-først søk fra noder som ikke er besøkte, inntil alle noder er behandlet.
- Dybde-først søk er en effektiv måte å gå gjennom alle nodene i en graf fordi hver node bare blir besøkt en gang.

Eksempel fra boka: finn DFT



Er grafen sammenhengende?

Dybde-først søk kan brukes til å sjekke om en graf er sammenhengende:

- 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 for hver eneste node v klarer å besøke alle andre noder i grafen når vi gjør dybde-først søk fra v.

Løkkeleting og topologisk sortering

- Vi kan bruke dybde-først søk til å sjekke om en graf har løkker (sykler), og til å angi en omvendt topologisk sortering dersom grafen er løkkefri.
- Vi trenger da tre verdier til tilstandsvariablen: usett, igang og ferdig (besøkt).

```
proc. Løkkeleting(v); ref(node) v;
begin
   if v.tilstand = igang then <Løkke er funnet>
   else if v.tilstand = usett then
   begin
      v.tilstand := igang;
      for w:= <hver etterfølger til v> do
         Løkkeleting(w);
      v.tilstand := ferdig;
   <Skriv ut noden v, for omvendt topologisk sortering>
   end
end;
```

- Prosedyren bygger på at de noder der kall er i gang alltid ligger på en rett vei fra startnoden.
- Dersom man fra et kall i en node har en utgående kant til en node der kall allerede er i gang, så har vi funnet en løkke.
- Vi må passe på å gjøre nye startkall på denne prosedyren inntil den er kalt i alle noder. (Det er nok å starte nye søk fra alle usette noder uten forgjengere.)
- Denne prosedyren vil alltid finne en løkke dersom det finnes en.
- Det kan vises ved et **bevis-ved-selvmotsigelse**.

- Løkkeleting-prosedyren kan 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 (er ferdig med kallet).
- Dette er riktig tidspunkt å skrive ut noden fordi:
  — Vi har besøkt alle etterfølgerne til noden.
  — Etterfølgerne var enten 'usett' (og da har vi skrevet dem ut i det vi trakk oss tilbake fra dem), eller 'ferdig' (og da var de skrevet ut tidligere).

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

ALMIRA KARABEG foreleser!

Vi introduserer minimum span tre problemet

- Prim's algoritme (kapittel 9.5.1)
- Kruskal's algoritme (kapittel 9.5.2)

Og en grådig algoritme til

- Huffman kode (kapittel 10.1.2.)