Uke 9,
Forelesning 1

HUSK – *Hittil…*

- Forutsetninger for og essensen i faget
- Metodekall, rekursjon, permutasjoner
- Analyse av algoritmer
- Introduksjon til ADT'er
- De første ADT'er: Lister, stabler og køer

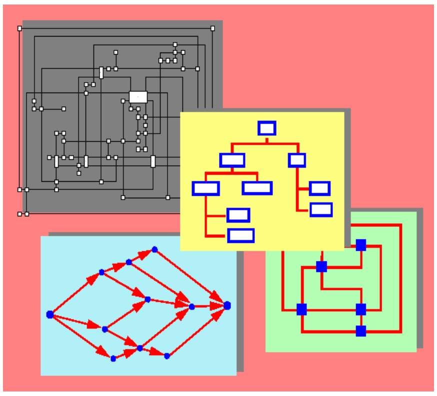
Uke 1, Uke 2 og Uke 3

- Flere ADT'er: Generelle trær, binære trær og binære søketrær

Uke 4 og Uke 5

- Flere ADT'er: Hashing, hash-tabeller
- ADT'er for disk-datastrukturer introduseres: Utvidbar hashing, B-Trær
- Prioritetskøer & Heap-implemantasjonen
- Disjunkte mengder
- Introduksjon til grafer og topologisk sortering

Uke 6, Uke 7 og Uke 8

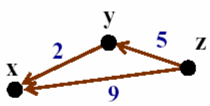


Fortsetter med Grafer…

Plan for i dag: korteste vei, én-til-alle for:

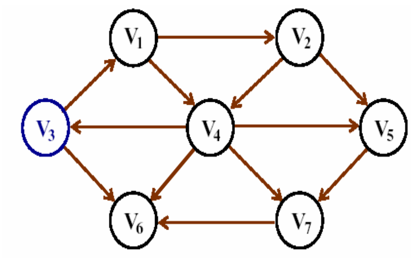
- Uvektet graf (kapittel 9.3.1)
- Vektet rettet graf uten negative kanter (kapittel 9.3.2)
- Vektet rettet graf med negative kanter (kapittel 9.3.3)

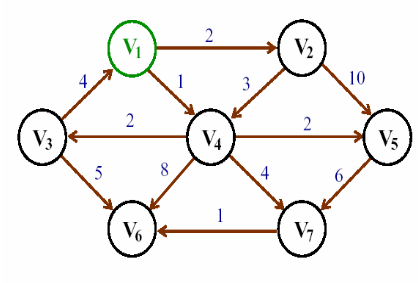
Problemstilling: **gitt en rettet graf G (vektet eller uvektet), finn korteste vei fra én gitt node til alle andre noder**



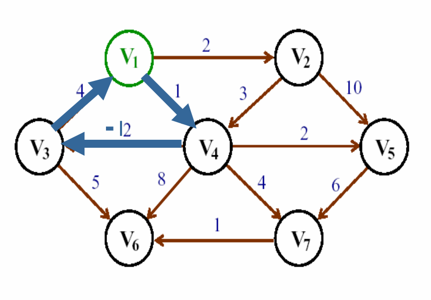
- Korteste vei fra z til x uten vekt er 1.
- Korteste vei fra z til x med vekt er 7 (via y).

Uvektet rettet graf kan betraktes som spesiel tilfele av vektet variant av problemet, hvor hver kant har vekt lik 1.

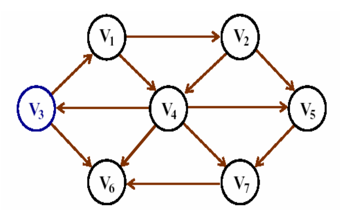




- Negative kanter må ikke være vanskelige
- Negativ kost løke (problemet som kan oppstå pga negative kanter)



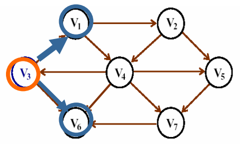
Problemstilling

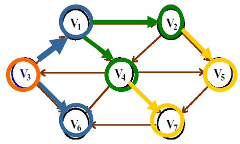


Korteste vei i en uvektet graf

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

- Følgende **bredde-først** algortime løser problemet:
  1. Marker at lengden fra s til s er lik 0.
  2. Se etter noder som er på avstand 1 fra s ved finne etterfølgere til s, og som ikke har fått markert noen distanse. Marker disse.
  3. Se etter noder som er på avstand 2 fra s, osv.
  4. Forsett inntil alle noder er markert, eller vi har kommet til distanse lik $N - 1$.

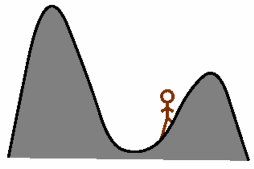




- Vi kan finne den korteste veien ved å sette **bakoverpekere** til den noden som "oppdaget" oss.
- Tidsforbruket til algoritmen er $\mathcal{O}(|V|^2)$.
- Vi sparer tid ved å plassere etterfølgerne til noden vi behandler på en **kø**. Så tar vi ut første node i køen og behandler denne.
- Da blir alle noder i avstand 1 behandlet før alle i avstand 2 før alle i avstand 3 . . .
- Denne strategien ligner på bredde-først traversering av trær (først rotnoden, så alle noder på nivå 1, så alle noder på nivå 2, osv.).
- Tidsforbruket blir da $\mathcal{O}(|E| + |V|)$ fordi køoperasjoner tar konstant tid og hver kant og hver node bare blir behandlet én gang.

- Vi skal først se på korteste vei fra én node til alle andre noder i en vektet graf *uten* negative kanter.
- Vi bruker de samme ideene som i en graf uten vekter.
- Algortimen vi da kommer fram til er kjent som **Dijkstras** algoritme.
- Den er et godt eksempel på en **grådig** algoritme:
  - Grådige algoritmer prøver i hvert steg å gjøre det som ser best ut der og da.
  - Algoritmene blir raske, men ikke alle problemer lar seg løse med grådige algoritmer:

Eksempel: Finn det høyeste punktet.



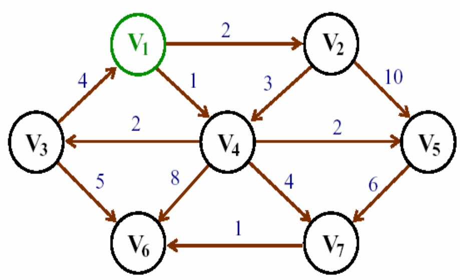
Dijkstras algoritme

1. Kall startnoden for s. Sett d_s lik 0 og marker s 'kjent'.
2. Sett distansen til alle nabonoder w lik kosten fra s til w, dvs. $d_w := c_{s,w}$.
3. Sett bakoverpekerne for nabonodene lik p_s.
4. Velg ukjent node v med minst distanse, og marker v som 'kjent'.
5. Se på alle ukjente nabonoder w:
   (a) Reduserer distansen for w dersom lengden vi får ved å følge stien gjennom v er kortere enn den gamle lengden: $d_w := \min(d_w, d_v + c_{v,w})$.
   (b) Hvis lengden ble redusert, så sett bakoverpekeren lik p_v.
6. Så lenge det finnes ukjente noder, gå til punkt 3.

http://www-b2.is.tokushima-u.ac.jp/~ikeda/suuri/dijkstra/DijkstraApp.shtml?demo1

Algoritmen velger altså i hvert steg den noden v som har minst distanse og som er ukjent. v markeres som 'kjent', og deretter undersøker algoritmen 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).

Eksempel:



v	kjent	d_v	p_v
v_1			
v_2			
v_3			
v_4			
v_5			
v_6			
v_7			

Hvorfor virker algoritmen?

- Algoritmen har følgende **innvariant**: *Alle kjente noder har mindre distanse enn de ukjente nodene.*
- Det medfører at alle kjente noder har riktig korteste vei satt (distansen er faktisk den korteste distansen).
- Vi plukker ut den ukjente noden v med minst distanse og markerer den som kjent.
- Dermed påstår vi at distansen til v er riktig.

- Den 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.

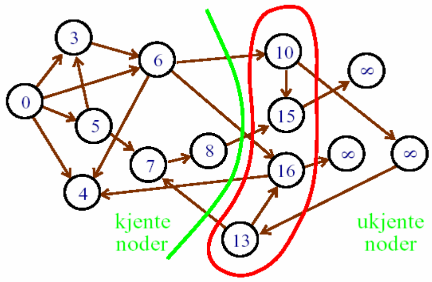
Tidsforbruk

- Se implementasjon i pseudokode på side 308 i MAW.
- Algoritmen leter sekvensielt etter den ukjente noden med minst distanse. Det tar $\mathcal{O}(|V|)$ tid.
- Dette gjøres $|V|$ ganger, så total tid for å finne minste distanse blir $\mathcal{O}(|V|^2)$.
- I tillegg bruker algoritmen konstant tid på å oppdatere distansene.

- Det er ikke mer enn en oppdatering per kant, så total tid for å oppdatere distansene blir $\mathcal{O}(|E|)$.
- Total tid for hele algoritmen blir dermed $\mathcal{O}(|E| + |V|^2)$
- Hvis grafen er **tett**, dvs. at $|E| = \Theta(|V|^2)$, så er algoritmen optimal (alle andre algoritmer må også bruke minst $\mathcal{O}(|V|^2)$ tid.
- Hvis grafen er **tynn**, dvs. at $|E| = \Theta(|V|)$, så kan vi klare det bedre!

En raskere implementasjon for tynne grafer

- Idéen er å bruke en **prioritetskø** for å finne minste distanse i **sublineær** tid.



- Vi plasserer på priortetskøen de noder som er ukjente og som har distanse mindre enn ∞.
- Prioritetskøen er ordnet på distanse, slik at vi får ut noden med den korteste veien fra de kjente nodene.
- Ved starten av algoritmen setter vi startnoden inn i prioritetskøen med distanse=0.
- Vi må ta hensyn til at prioriteten til en ukjent node forandres hvis vi finner en kortere vei til noden.
- **DeleteMin** og **DecreaseKey** tar $\mathcal{O}(\log|V|)$ tid, så totalt tidsforbruk blir $\mathcal{O}(|V|\log|V| + |E|\log|V|) = \mathcal{O}(|E|\log|V|)$.

Hva med negative kanter?

- Dersom den vektede grafen har negative kanter, fungerer ikke Dijkstras algoritmen.
- 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 **FIFO-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.

ALMIRA KARABEG foreleser

- Vi avslutter korteste vei algoritmer ved å se på tilfeller med negative kanter
- Activity graphs
- Depth-first search
- Finding cycles