GRAFER

Dagens plan:

- Kort repetisjon om grafer
- Korteste vei, en-til-alle, for:
  - uvektede grafer (repetisjon)
  - vektede rettede grafer uten negative kanter (Dijkstra, kapittel 9.3.2)
  - vektede rettede grafer med negative kanter (kapittel 9.3.3)
- Minimalt spenntre
  - Prim (Kapittel 9.5.1)
  - Kruskal (Kapittel 9.5.2)

Siden forelesningen forrige uke måtte avlyses, vil følgende sentrale temaer ikke bli gjennomgått på forelesning:

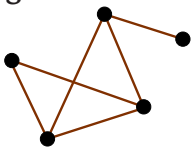
- Aktivitetsgrafer (kapittel 9.3.4)
- Dybde-først søk (kapittel 9.6.1)
  - Løkkeleting

Dette vil istedenfor bli tatt ekstra grundig på gruppene.

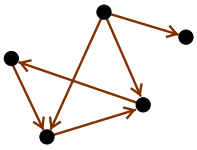
Det anbefales også sterkt at dere gjør den frivillige innleveringen!

Noen grafdefinisjoner

- En graf G består av:
  - En mengde noder, V
  - En mengde kanter, E
- Hver kant er et par av noder, (u, v), og modellerer at u er relatert til v.



Urettet graf

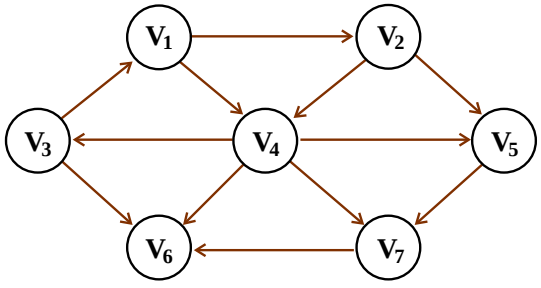


Rettet graf

- $|V|$ er antall noder i grafen
- $|E|$ er antall kanter i grafen
- Node y er nabo-node/etterfølger til node x dersom $(x, y) \in$ E.
- En graf er vektet dersom hver kant har en tredje komponent, kalt kost eller vekt.
- En vei (eller sti) i en graf er en sekvens av noder $v_1, v_2, v_3, \ldots, v_n$ slik at $(v_i, v_{i+1}) \in$ E for $1 \leq i \leq n$.

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).



Bredde-først algoritme:
(MAW side 303, figur 9.18)

```
void uvektet(Node s) {
   Kø k = new Kø;
   Node v;

   k.settInn(s);
   s.avstand = 0;

   while (!k.isEmpty()) {
      v = k.taUt();
      v.kjent = true;

      for < hver nabo w til v > {
         if (w.avstand = UENDELIG) {
            w.avstand = v.avstand + 1;
            w.vei = v;
            k.settInn(w);
         }
      }
   }
}
```

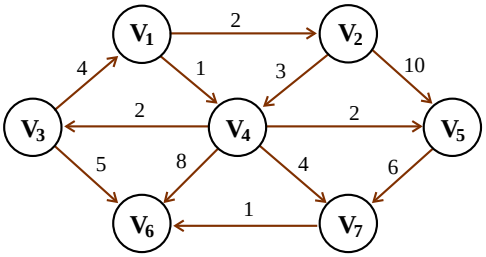
Tidsforbruk: $O(|E| + |V|)$.

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 med minst avstand fra startnoden.
- Den samme hovedideen kan brukes hvis vi har en graf med vekter.
- Akkurat som for uvektede grafer ser vi bare etter potensielle forbedringer for naboer som enda ikke er valgt (kjent).

Vi får da en algoritme kjent som Dijkstras algoritme.

Eksempel



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			

Dijkstras algoritme

1. Sett avstanden fra startnoden s til seg selv lik 0.
2. Velg ukjent node v med minst avstand, og marker v som "kjent".
3. For hver ukjente nabonode w til v: Dersom avstanden vi får ved å følge veien gjennom v er kortere enn den gamle avstanden:
   (a) Reduserer avstanden for w.
   (b) Sett bakoverpekeren i w til v.
4. Så lenge det finnes ukjente noder, gå til punkt 2.

Hvorfor virker Dijkstras algoritme?

Algoritmen har følgende invariant:
Ingen ukjente noder har mindre avstand enn noen av de kjente nodene.

MAW side 309, figur 9.32

```
void dijkstra(Node s) {
   s.avstand = 0;
   for ( ; ; ) {
      v = < ukjent node med minst avstand >;
      if (v == null) {
         return;
      }

      v.kjent = true;
      for < hver nabo w til v > {
         if (!w.kjent) {
            if (v.avstand + avstand(v,w) < w.avstand) {
               w.avstand = v.avstand + avstand(v,w);
               w.vei = v;
            }
         }
      }
   }
}
```


Tidsforbruk

- Hvis vi leter sekvensielt etter den ukjente noden med minst avstand tar dette $O(|V|)$ tid. Dette gjøres $|V|$ ganger, så total tid for å finne minste avstand blir $O(|V|^2)$.
- I tillegg oppdateres avstandene, maksimalt en oppdatering per kant, dvs. til sammen $O(|E|)$.
- Total tid: $O(|E| + |V|^2) = O(|V|^2)$.

Raskere implementasjon (for tynne grafer):

- Bruker en prioritetskø til å ta vare på ukjente noder med avstand mindre enn ∞.
- Vi må ta hensyn til at prioriteten til en ukjent node forandres hvis vi finner en kortere vei til noden.
- DeleteMin og DecreaseKey tar $O(\log |V|)$ tid.
- Totalt tidsforbruk blir $O(|V| \log |V| + |E| \log |V|) = O(|E| \log |V|)$.

Hva med negative kanter?

- Dersom den vektede grafen har negative kanter, fungerer ikke Dijkstras algoritme (se oppgave 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:
    - Ta ut en node v fra køen.
    - For hver etterfølger w, sjekk om vi får en forbedring.
    - Oppdater i så fall avstanden, og plasser w på køen (hvis den ikke er der allerede).
- Tidsforbruket blir da $O(|E| \cdot |V|)$ som er mye verre enn Dijkstras algoritme.
- Algoritmen fungerer ikke hvis det er **negative løkker**.

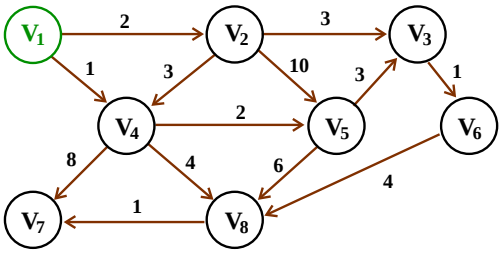
Rettede, ikke-sykliske grafer (DAG)

- Dersom vi vet at grafen ikke inneholder løkker, 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 avstand:

  Avstanden kan jo ikke lenger forandres, siden den topologiske ordningen garanterer at noden ikke har inngående kanter fra ukjente noder.

Tidsforbruk:

Eksempel

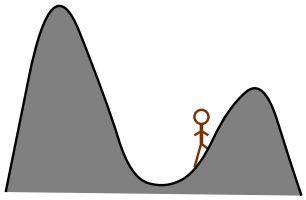


v	kjent	avstand	vei
v_1			
v_2			
v_3			
v_4			
v_5			
v_6			
v_7			
v_8			

Hva hvis vi ønsker å ha v_2 som startnode?

Grådige algoritmer

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

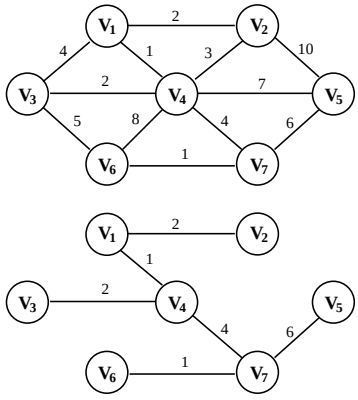


Finn det høyeste punktet!

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

Minimalt spenntre

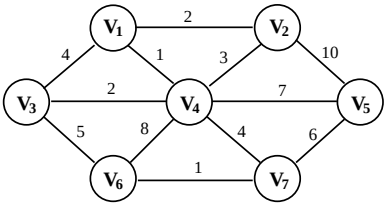
- Et minimalt spenntre for en **urettet** graf G er et tre bestående av kanter fra grafen, slik at alle nodene i G er forbundet til lavest mulig kostnad.
- Minimale spenntrær eksisterer bare for sammenhengende grafer.
- Generelt finnes det flere minimale spenntrær for samme graf.

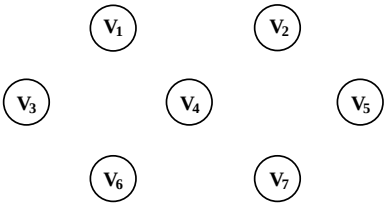


Hvor mange kanter får spenntreet i det generelle tilfellet?

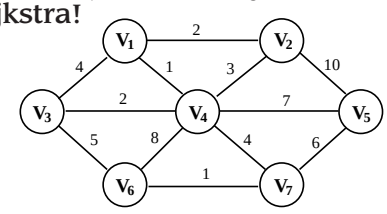
Prims algoritme

- Treet bygges opp stegvis. I hvert steg 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.





- 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!

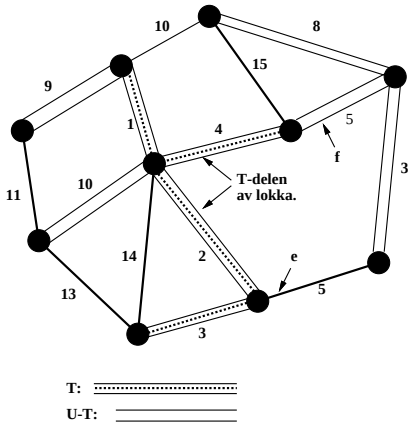


v	kjent	avstand	vei
V_1	F	0	0
V_2	F	∞	0
V_3	F	∞	0
V_4	F	∞	0
V_5	F	∞	0
V_6	F	∞	0
V_7	F	∞	0

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.
- Tidsforbruk: