INF2220 - Algoritmer og datastrukturer

HØSTEN 2017

Ingrid Chieh Yu
Institutt for informatikk, Universitetet i Oslo

Forelesning 5:

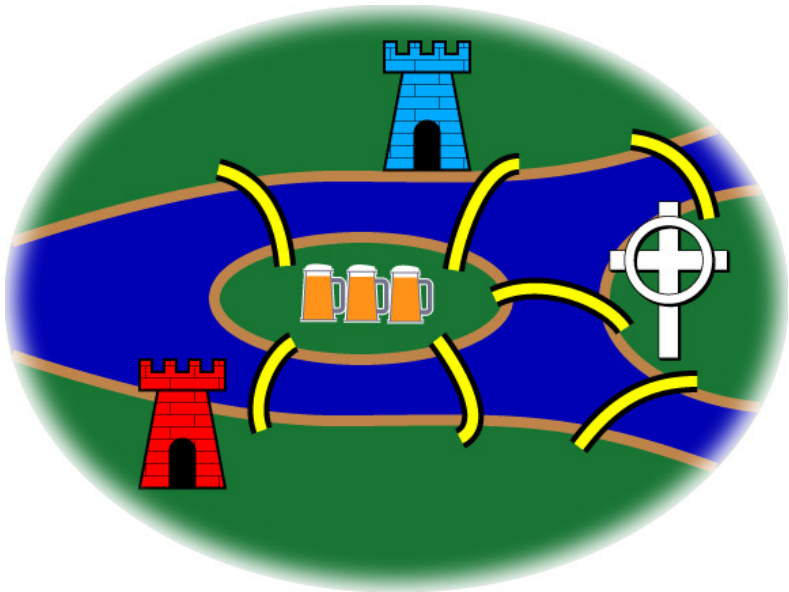
Grafer I

Dagens plan:

- Definisjon av en graf
- Grafvarianter
- Intern representasjon av grafer
- Topologisk sortering
- Korteste vei en-til-alle uvektet graf
- Korteste vei en-til-alle vektet graf
- M.A.W. Data Structures and Algorithm Analysis kap. 9

Det første grafteoretiske problem: Broene i Königsberg

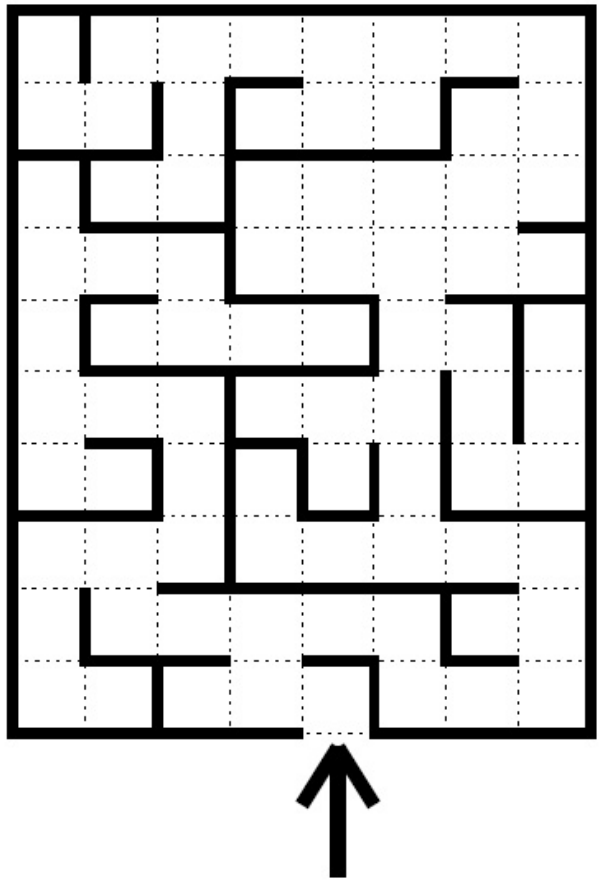
Er det mulig å ta en spasertur som krysser hver av broene nøyaktig en gang?



Dette problemet ble løst av Euler allerede i 1736!

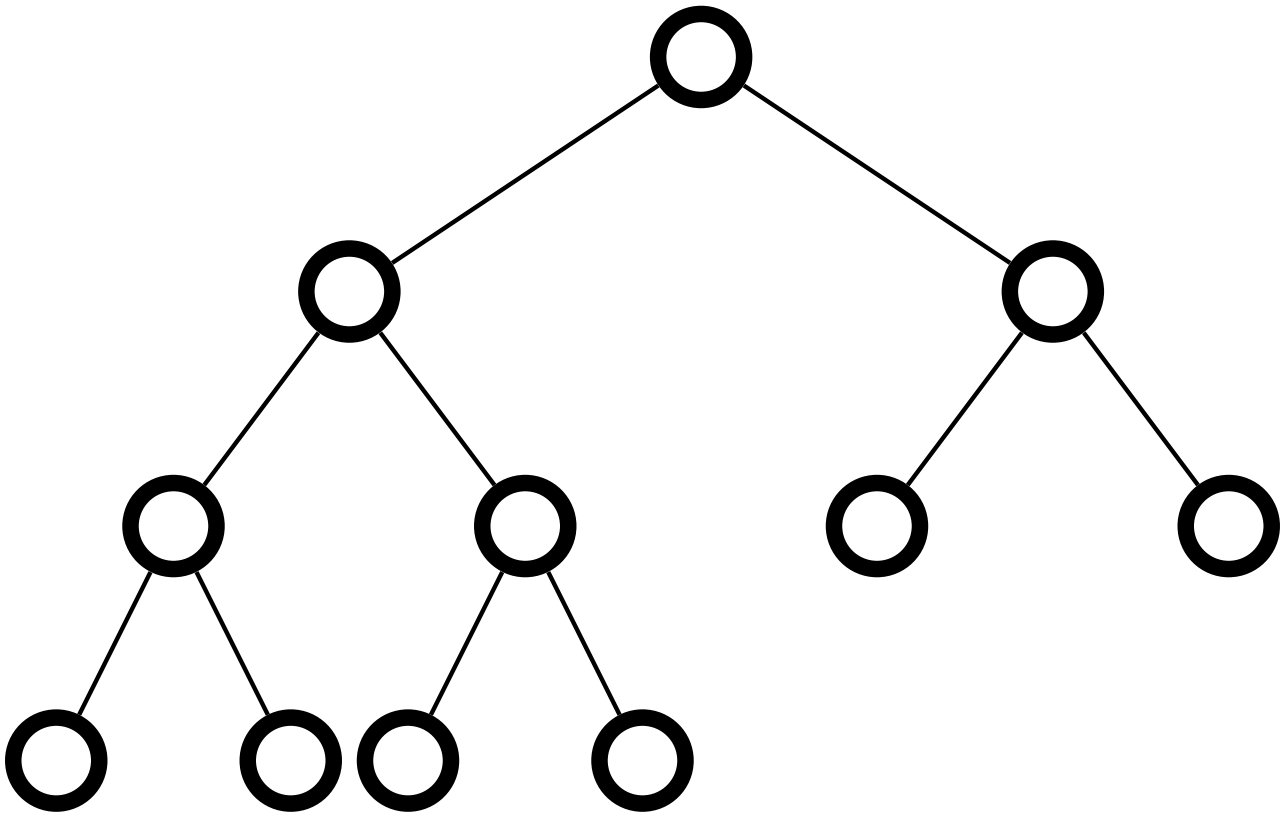
Grafer vi har sett allerede

- Labyrint



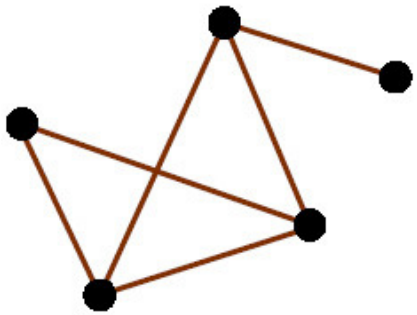
Grafer vi har sett allerede

- Trær

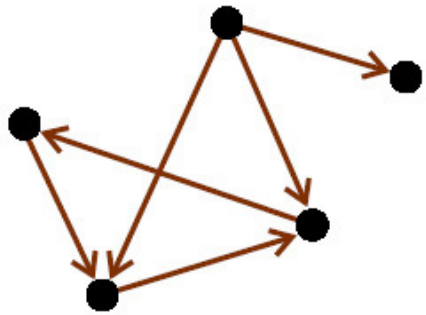


Hva er en graf?

- En graf G =(V,E) har en mengde noder, V, og en mengde kanter, E
- |V| og |E| er henholdsvis antall noder og antall kanter i grafen
- Hver kant er et par av noder, dvs. (u, v) slik at $u, v \in V$
- En kant (u, v) modellerer at u er relatert til v
- Dersom nodeparet i kanten (u, v) er ordnet (dvs. at rekkefølgen har betydning), sier vi at grafen er rettet, i motsatt fall er den urettet



Urettet graf



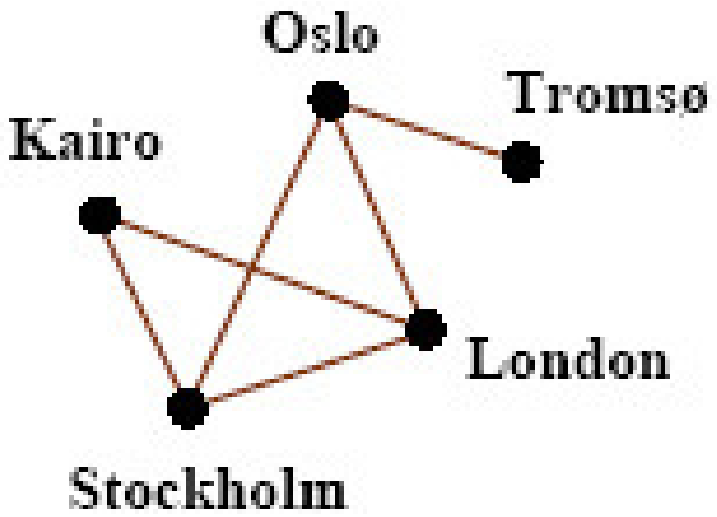
Rettet graf

- Grafer er den mest fleksible datastrukturen vi kjenner ("alt" kan modelleres med grafer)

Hvorfor grafer?

- De dukker opp i veldig mange problemer i hverdagslivet:

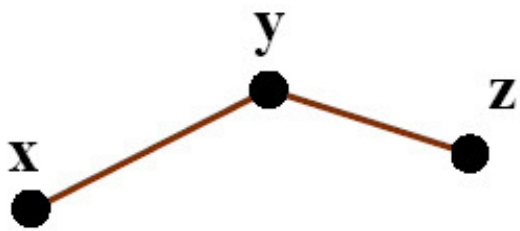
  - Flyplassystemer
  - Datanettverk
  - Trafikkflyt
  - Ruteplanlegging
  - VLSI (chip design)
  - og mange flere . . .



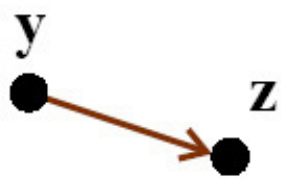
- Grafalgoritmer viser veldig godt hvor viktig valg av datastruktur er mhp. tidsforbruk
- Det finnes grunnleggende algoritmeteknikker som løser mange ikke-trivielle problemer raskt

Grafer: Definisjoner og varianter

- Node y er nabo-node (eller etterfølger) til node x dersom $(x, y) \in E$

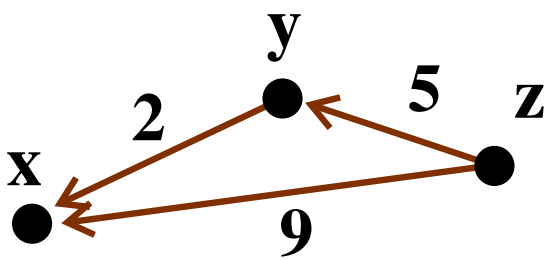


x og y er naboer,
y og z er naboer,
men x og z er ikke naboer

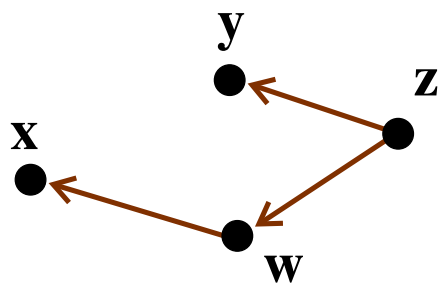


z er nabo-node til y,
men y er ikke
nabo-node til z

- En graf er vektet dersom hver kant har en tredje komponent, kalt kost eller vekt

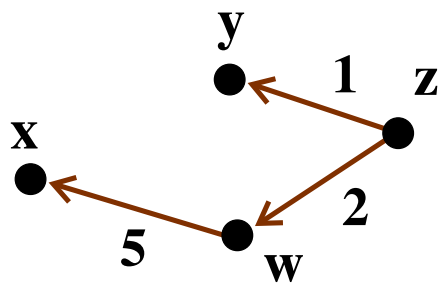


- En vei (eller sti) i en graf er en sekvens av noder $v_1, v_2, v_3, \ldots, v_n$ slik at $(v_i, v_{i+1}) \in E$ for $1 \leq i \leq n-1$
- Lengden til veien er lik antall kanter på veien, dvs. $n-1$



<z, w, x> er en vei med lengde 2

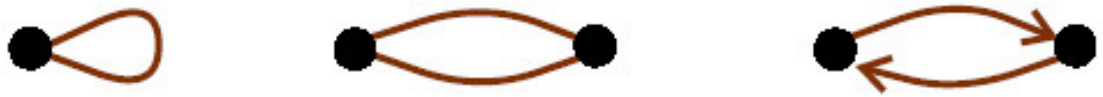
- Kosten til en vei er summene av vektene langs veien



<z, w, x> er en vei med kost 7

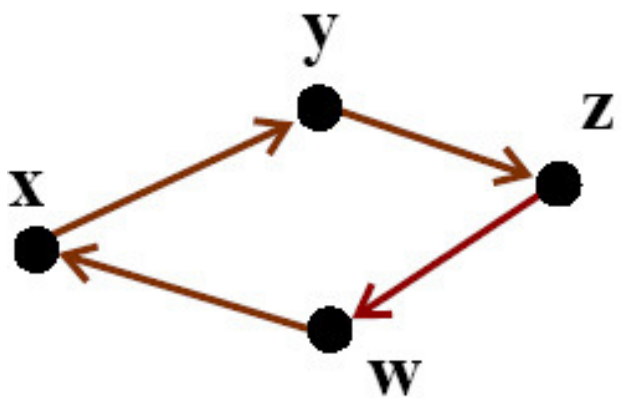
- En vei er enkel dersom alle nodene (untatt muligens første og siste) på veien er forskjellige

- Våre grafer har vanligvis ikke "loops", (v, v), eller "multikanter" (to like kanter):



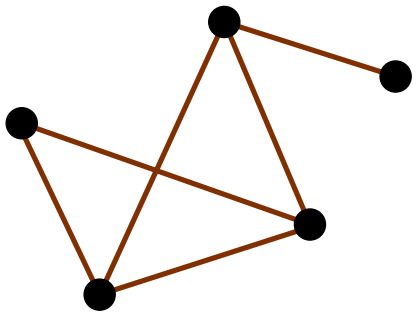
loop multikant vanlig rettet graf

- En løkke (sykel) i en rettet graf er en vei med lengde ≥ 1 slik at $v_1 = v_n$. Løkken er enkel dersom stien er enkel
- I en urettet graf må også alle kanter i løkken være forskjellige

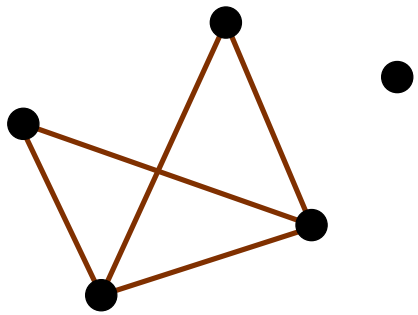


<x, y, z, w, x> er en enkel løkke

- En rettet graf er asyklisk dersom den ikke har noen løkker
- En rettet, asyklisk graf blir ofte kalt en DAG (Directed, Acyclic Graf)
- En urettet graf er sammenhengende dersom det er en vei fra hver node til alle andre noder

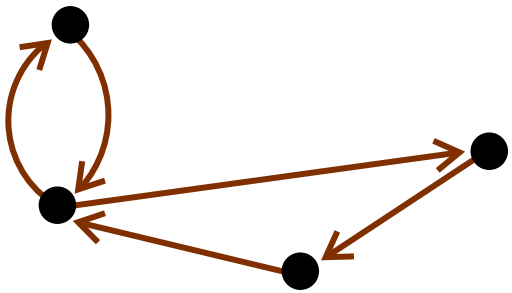


sammenhengende

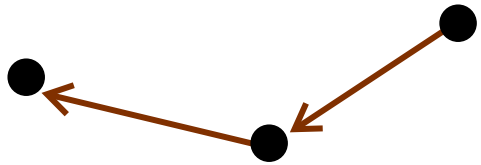


ikke sammenhengende

- En rettet graf er sterkt sammenhengende dersom det er en vei fra hver node til alle andre noder
- En rettet graf er svakt sammenhengende dersom den underliggende urettede grafen er sammenhengende

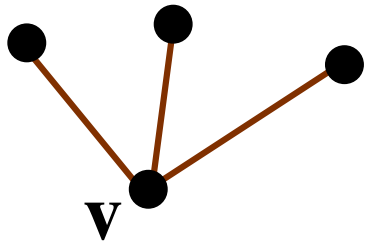


sterkt sammen–
hengende

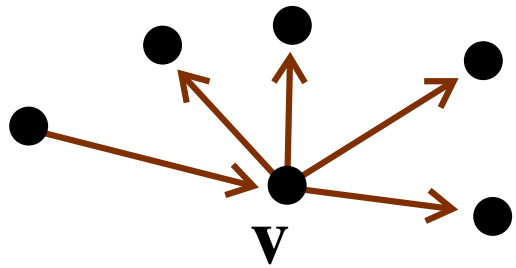


svakt sammen–
hengende

- Graden til en node i en urettet graf er antall kanter mot noden
- Inngraden til en node i en rettet graf er antall kanter inn til noden
- Utgraden til en node i en rettet graf er antall kanter ut fra noden.

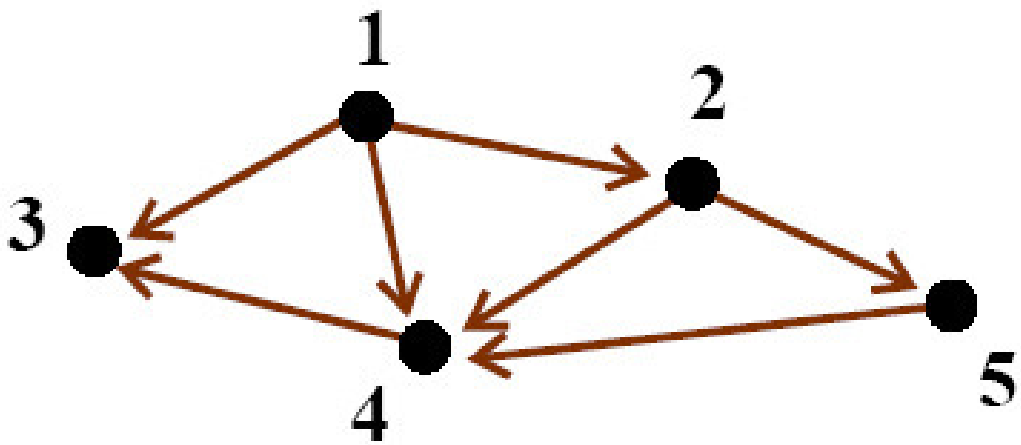


Grad(v)=3

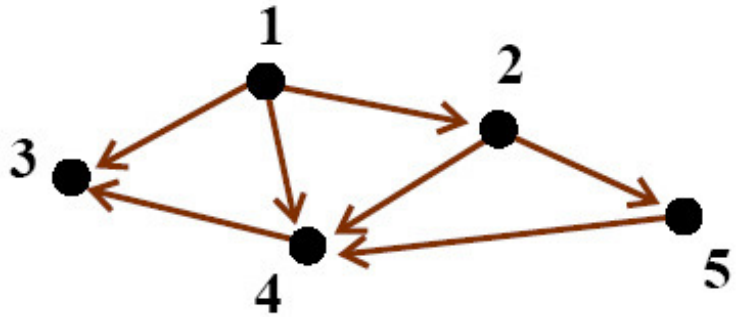


Inngrad(v)=1
Utgrad(v)=4

Hvordan representere grafer?



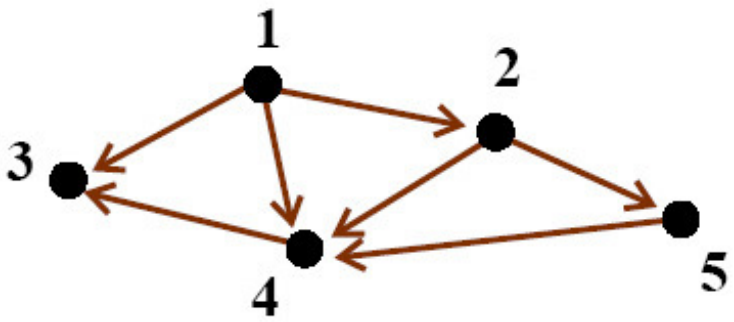
Nabo-matrise (adjacency matrix)

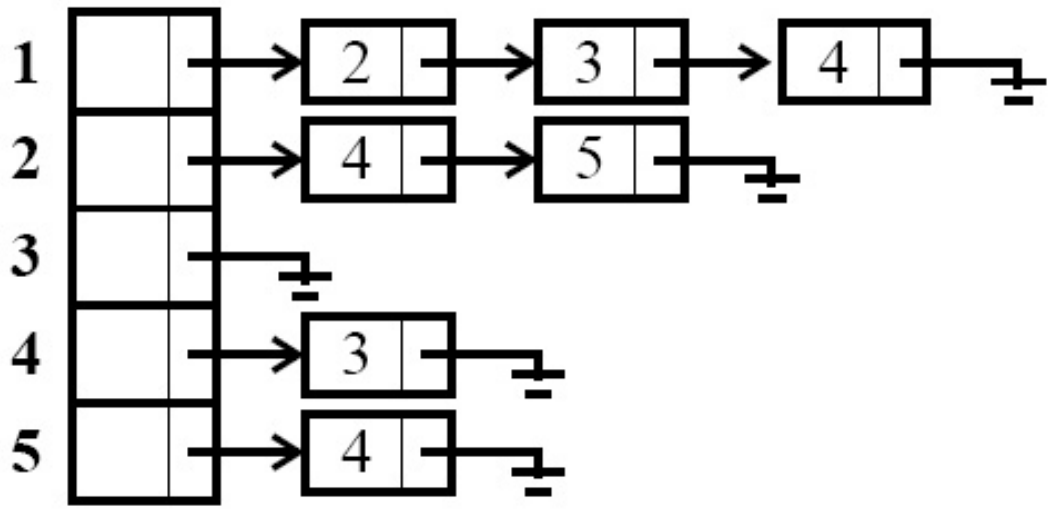


	1	2	3	4	5
1	0	1	1	1	0
2	0	0	0	1	1
3	0	0	0	0	0
4	0	0	1	0	0
5	0	0	0	1	0

- Bra hvis "tett" graf, dvs. $|E| = \Theta(|V|^2)$
- Det tar $\mathcal{O}(|V|)$ tid å finne alle naboer

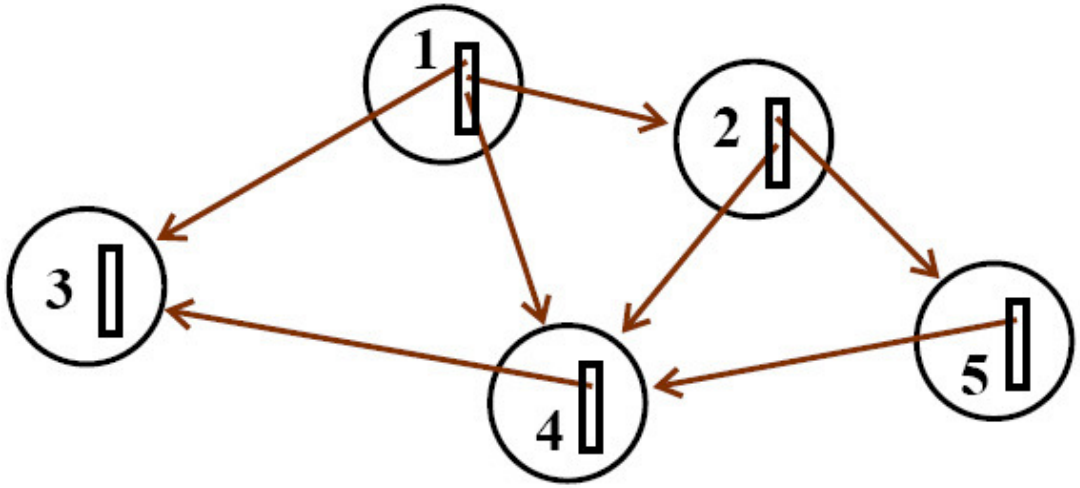
Nabo-liste (adhacency list)





- Bra hvis "tynn" ("sparse") graf
- Tar $\mathcal{O}(\text{Utgrad}(v))$ tid å finne alle naboer til v
- De fleste grafer i det virkelige liv er tynne!

Objekter & array



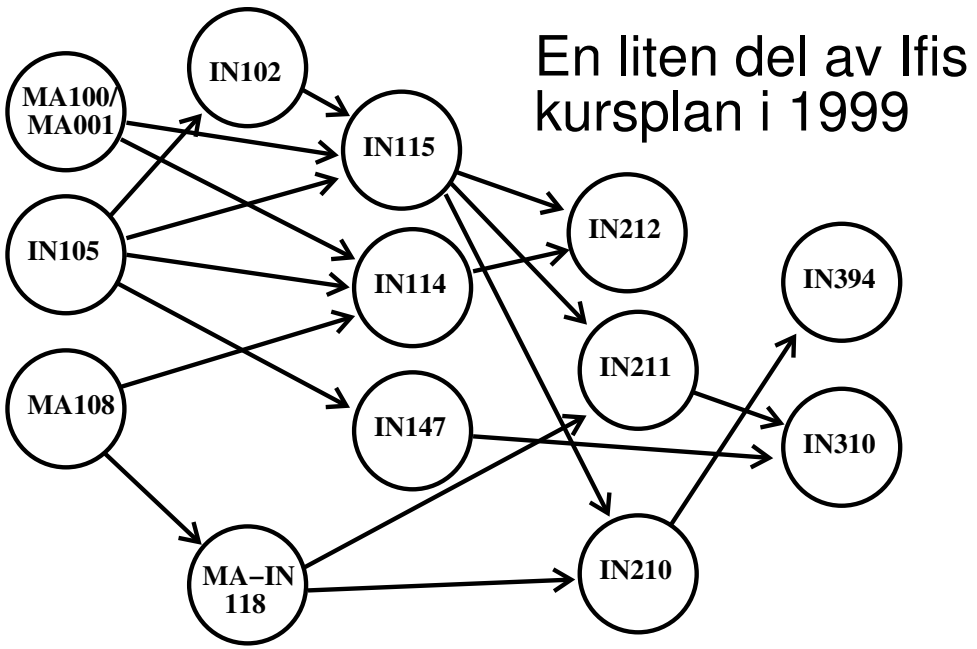
- I Java kan grafer også representeres ved en kombinasjon av node-objekter og etterfølgerarrayer
- Arraylengden kan være en parameter til node-klassens constructor

```
class Node {
    int antallNaboer;
    Node[ ] etterf;    double[ ] vekt;

    Node(int kapasitet) {
        etterf = new Node[kapasitet];
        vekt   = new double[kapasitet];
        antallNaboer = 0;}
}
```

- Da må vi vite antall etterfølgere når vi genererer noden
- Eventuelt kan vi *estimere* en øvre grense og la siste del av arrayen være tom
- Vi trenger da en variabel som sier hvor mange etterfølgere en node *faktisk* har

Topologisk sortering

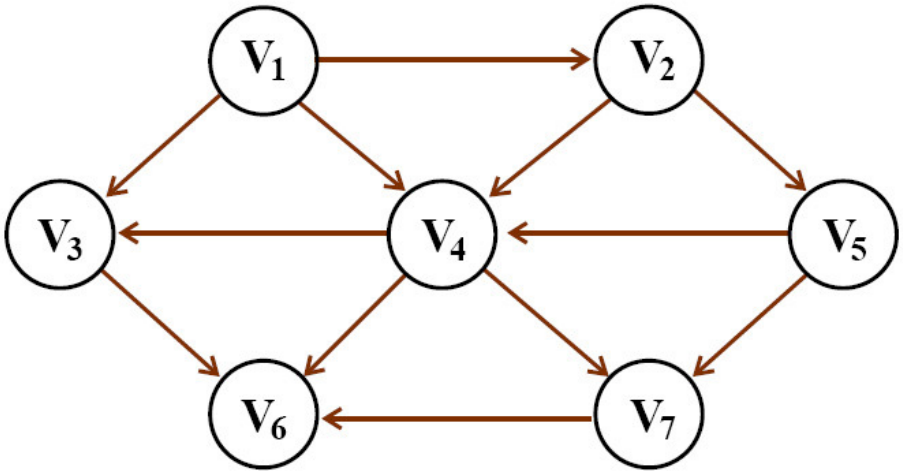
- En topologisk sortering er en ordning (rekkefølge) av noder i en DAG slik at dersom det finnes en vei fra v_i til v_j, så kommer v_j **etter** v_i i ordningen
- Topologisk sortering er umulig hvis grafen har en løkke



Følgende enkle algoritme finner en topologisk sortering (hvis det er noen):

1. Finn en node med inngrad = 0
2. Skriv ut noden, og fjern noden og utkantene fra grafen (marker noden som ferdig og reduser inngraden til nabonodene)
3. Gå tilbake til punkt 1

Eksempel:



Algoritme for topologisk sortering

```
void topsort() {
    Node v;

    for (int teller = 0; teller < ANTALL_NODER; teller++) {
        v = finnNyNodeMedInngradNull();

        if (v == null) {
            error("Løkke funnet!");

        } else {
            < Skriv ut v som node 'teller' >
            for < hver nabo w til v > {
                w.inngrad--;
          }}}}
```

- Denne algoritmen er $\mathcal{O}(|V|^2)$ siden finnNyNodeMedInngradNull ser gjennom hele node/inngrad-tabellen hver gang
- Unødvendig: bare noen få av verdiene kommer ned til 0 hver gang

En forbedring er å holde alle noder med inngrad=0 i en *boks*.

Boksen kan implementeres som en stakk eller en kø:

1. Plasser alle nodene med inngrad=0 i boksen.
2. Ta ut en node v fra boksen.
3. Skriv ut v.
4. Fjern v fra grafen og reduserer inngraden til alle etterfølgerne.
5. Dersom noen av etterfølgerne får inngrad=0, settes de inn i boksen.
6. Gå tilbake til punkt 2.

Forbedret algoritme

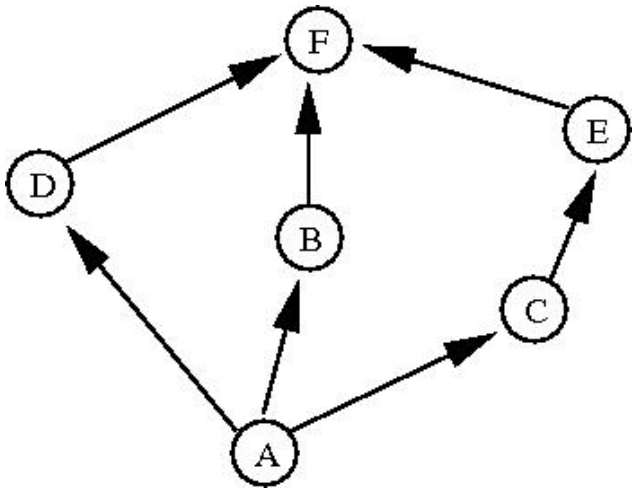
```
void topsort() {
    Kø k = new Kø();
    Node v;
    int teller = 0;

    for < hver node v >
        if (v.inngrad == 0) k.settInn(v);

    while (!k.isEmpty()) {
        v = k.taUt();
        < Skriv ut v >;
        teller++;

        for < hver nabo w til v > {
            w.inngrad--;
            if (w.inngrad == 0) k.settInn(w);
        }
    }
    if (teller < ANTALL_NODER) error("Løkke funnet!");
}
```

- Forutsatt at vi bruker nabolister, er denne algoritmen $\mathcal{O}(|V| + |E|)$.

  Kø/stakk-operasjoner tar konstant tid, og hver node og hver kant blir bare behandlet en gang.

Oppgave

Hvilke av disse er lovlige topoligiske sorteringer av Grafen?



1. A,D,C,B,E,F
2. A,C,E,B,D,F
3. A,D,B,C,E,F
4. Alle de over
5. Ingen av de over

Oppgave

Anta at du har en avhengighetsgraf med noder: {Q, B, J, P, A, Z}. Grafen har 4 lovlige topologiske sorteringer, det finnes **ingen** annen sortering av nodene som tilfredstiller avhengighetene:

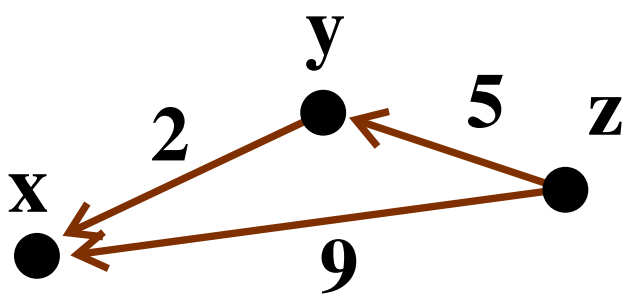
1. Q, A, B, J, Z, P
2. Q, B, A, J, Z, P
3. Q, A, B, Z, J, P
4. Q, B, A, Z, J, P

Tegn en graf som oppfyller kriteriene.

Korteste vei, en-til-alle

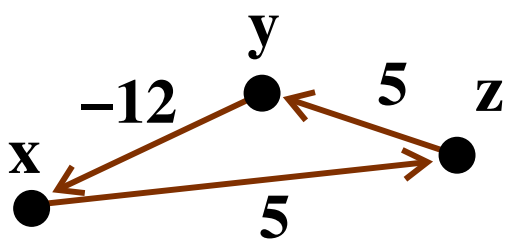
I korteste vei problemet (en-til-alle) har vi gitt en (muligens vektet) graf G=(V,E) og en node *s*.

Vi ønsker å finne den korteste veien (evt. med vekter) fra *s* til alle andre noder i G.



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

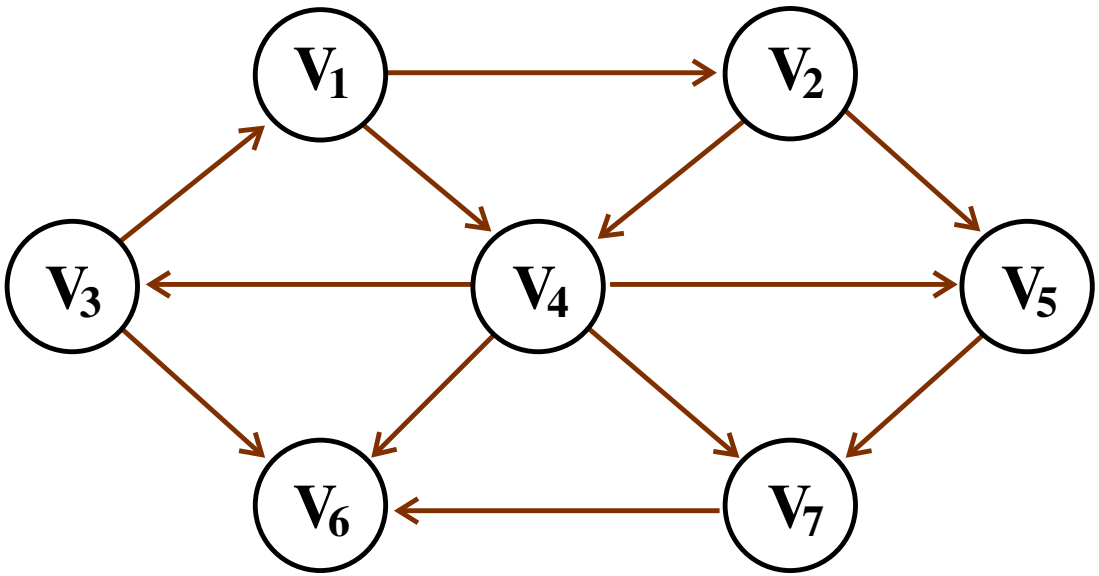
- Negative vekter (kost) i løkker kan skape problemer:



Hvor mye koster korteste vei fra x til z?

Korteste vei i en uvektet graf

Korteste vei fra s til t i en uvektet graf er lik veien som bruker færrest antall kanter.



(Det tilsvarer at alle kanter har vekt=1)

- Følgende bredde-først algoritme løser problemet for en node s i en uvektet graf G:

  1. Marker at lengden fra s til s er lik 0.
     (Merk at s foreløpig er den eneste noden som er markert.)
  2. Finn alle etterfølgere til s.
     Marker disse med avstand 1.
  3. Finn alle umarkerte etterfølgere til nodene som er på avstand 1.
     Marker disse med avstand 2.
  4. Finn alle umarkerte etterfølgere til nodene som er på avstand 2.
     Marker disse med avstand 3.
  5. Fortsett til alle noder er markert, eller vi ikke har noen umarkerte etterfølgere.

- Finnes det fortsatt umarkerte noder, kan ikke hele G nåes fra s.
- Hvis G er urettet, skjer dette hvis og bare hvis G er usammenhengende.

Vi kan finne den korteste veien ved å sette bakoverpekere ("vei") til den noden som "oppdaget" oss

```
void uvektet(Node s) {
    for < hver node v > {
        v.avstand = UENDELIG;
        v.kjent = false;
    }
    s.avstand = 0;

    for (int dist = 0; dist < ANTALL_NODER; dist++) {
        for < hver node v > {
            if (!v.kjent && v.avstand == dist) {
                v.kjent = true;
                for < hver nabo w til v > {
                    if (w.avstand == UENDELIG) {
                        w.avstand = dist + 1;
                        w.vei = v;
                    }}}}}}
```

Hovedløkken vil som oftest fortsette etter at alle noder er merket, men den vil terminere selv om ikke alle noder kan nåes fra s. Tidsforbruket er $\mathcal{O}(|V|^2)$.

- Vi sparer tid ved å benytte en *kø* av noder.
- Vi begynner med å legge s inn i køen.
- Så lenge køen ikke er tom, tar vi ut første node i køen, behandler denne og legger dens etterfølgere inn bakerst i køen.
- Da blir s behandlet først. Så 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 $\mathcal{O}(|E| + |V|)$ fordi køoperasjoner tar konstant tid og hver kant og hver node bare blir behandlet en gang.

Korteste uvektet vei fra node s

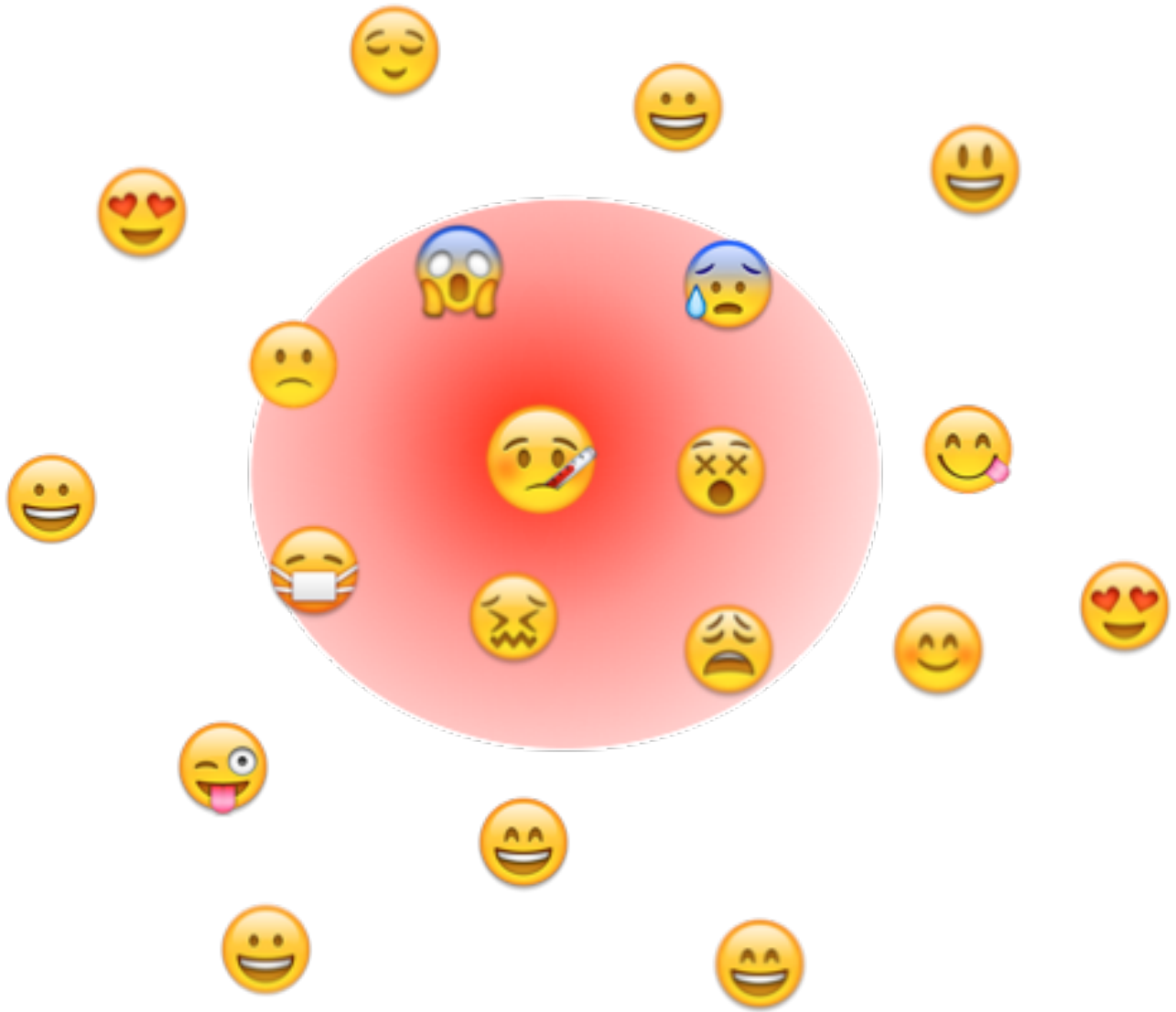
```
void uvektet(Node s) {
    Kø k = new Kø();
    Node v;
    for < hver node n > n.avstand = UENDELIG;
    s.avstand = 0;
    k.settInn(s);
    while (!k.isEmpty()) {
        v = k.taUt();
        for < hver nabo w til v > {
            if (w.avstand == UENDELIG) {
                w.avstand = v.avstand + 1;
                w.vei = v;
                k.settInn(w);
            }}}}
```

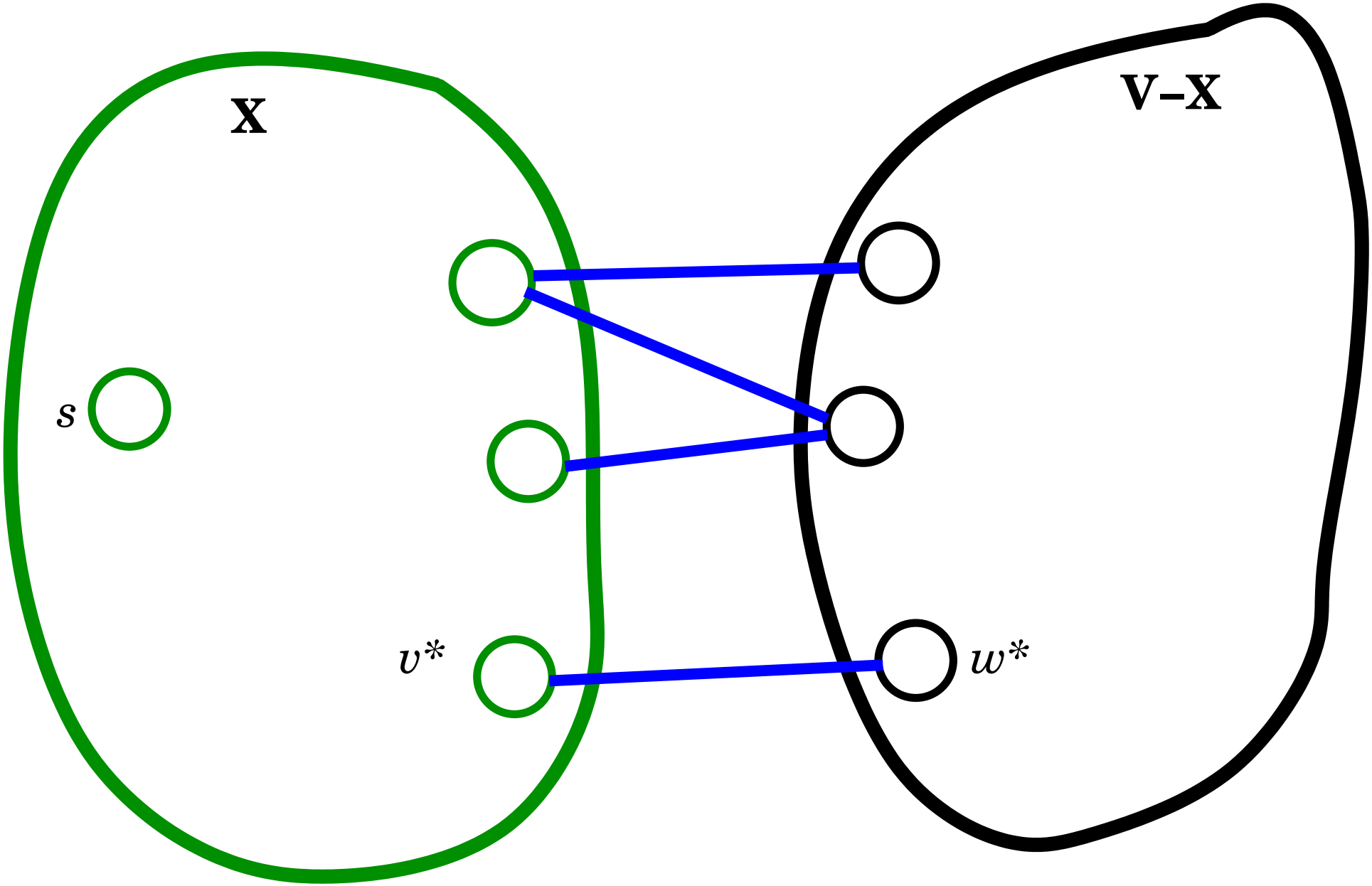
- Bruken av kø gjør attributtet kjent overflødig.
- Forutsatt at vi bruker nabolister, er denne algoritmen $\mathcal{O}(|V| + |E|)$.
- Kø-operasjoner tar konstant tid, og hver node og hver kant blir behandlet bare en gang.

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 blant dem med minst avstand fra startnoden
- Den samme hovedideen kan brukes hvis vi har en graf med vekter

  algoritmen: publisert av Dijkstra i 1959 og har fått navn etter ham



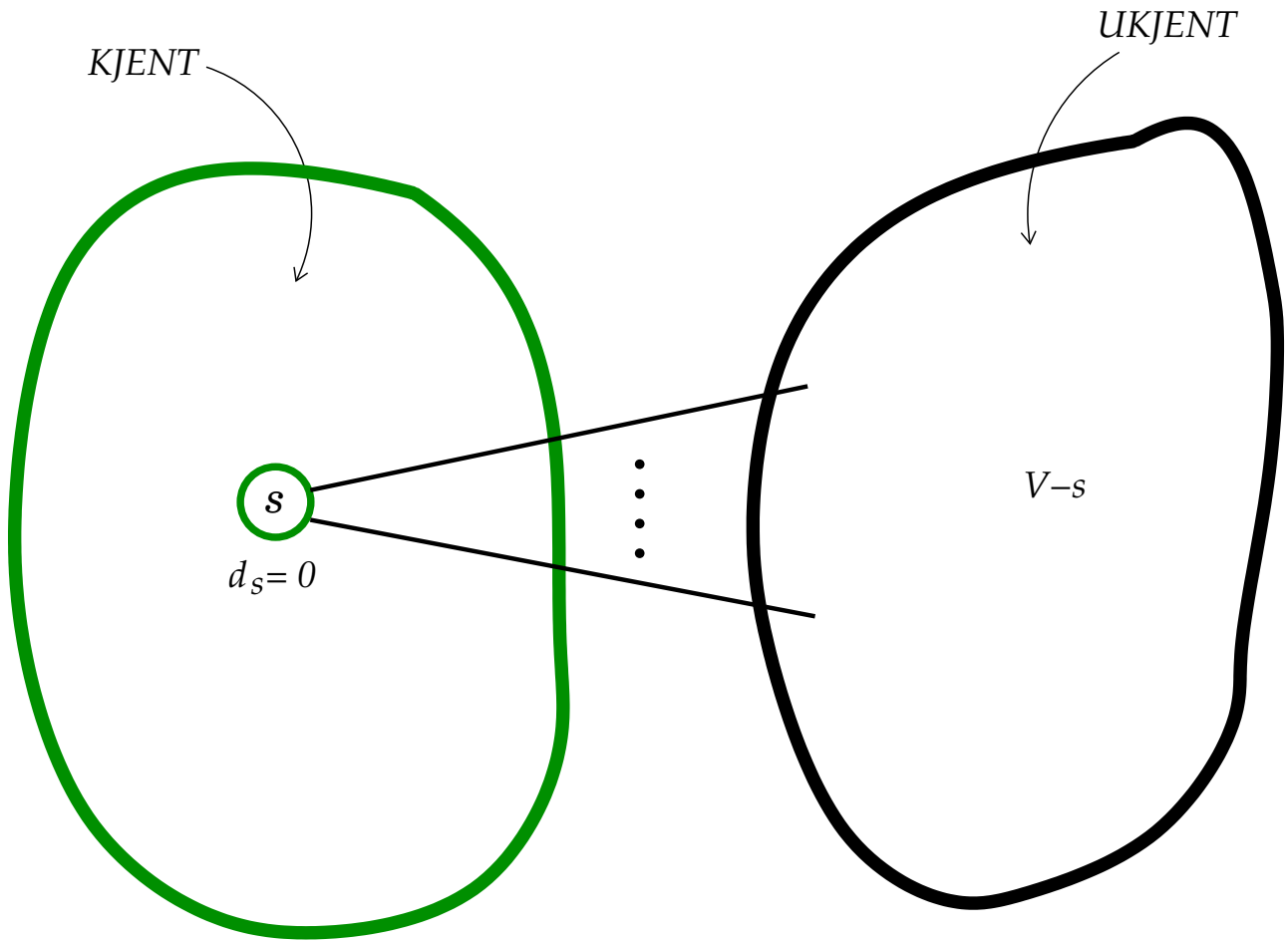

X
V–X
s
v*
w*

Nodene er enten KJENT eller UKJENT

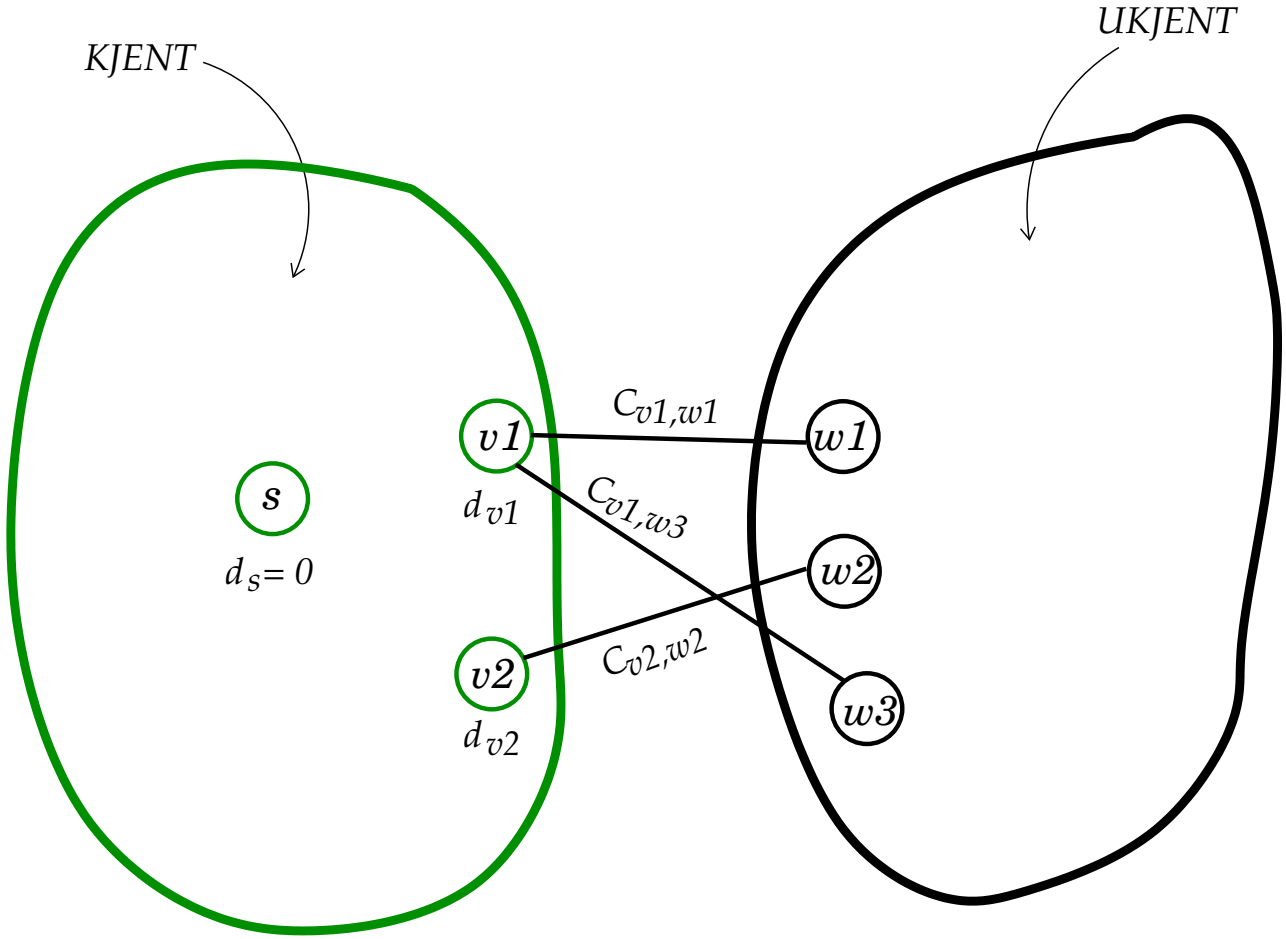
KJENT = korteste vei fra s funnet!

UKJENT = korteste vei er under beregning

i starten:

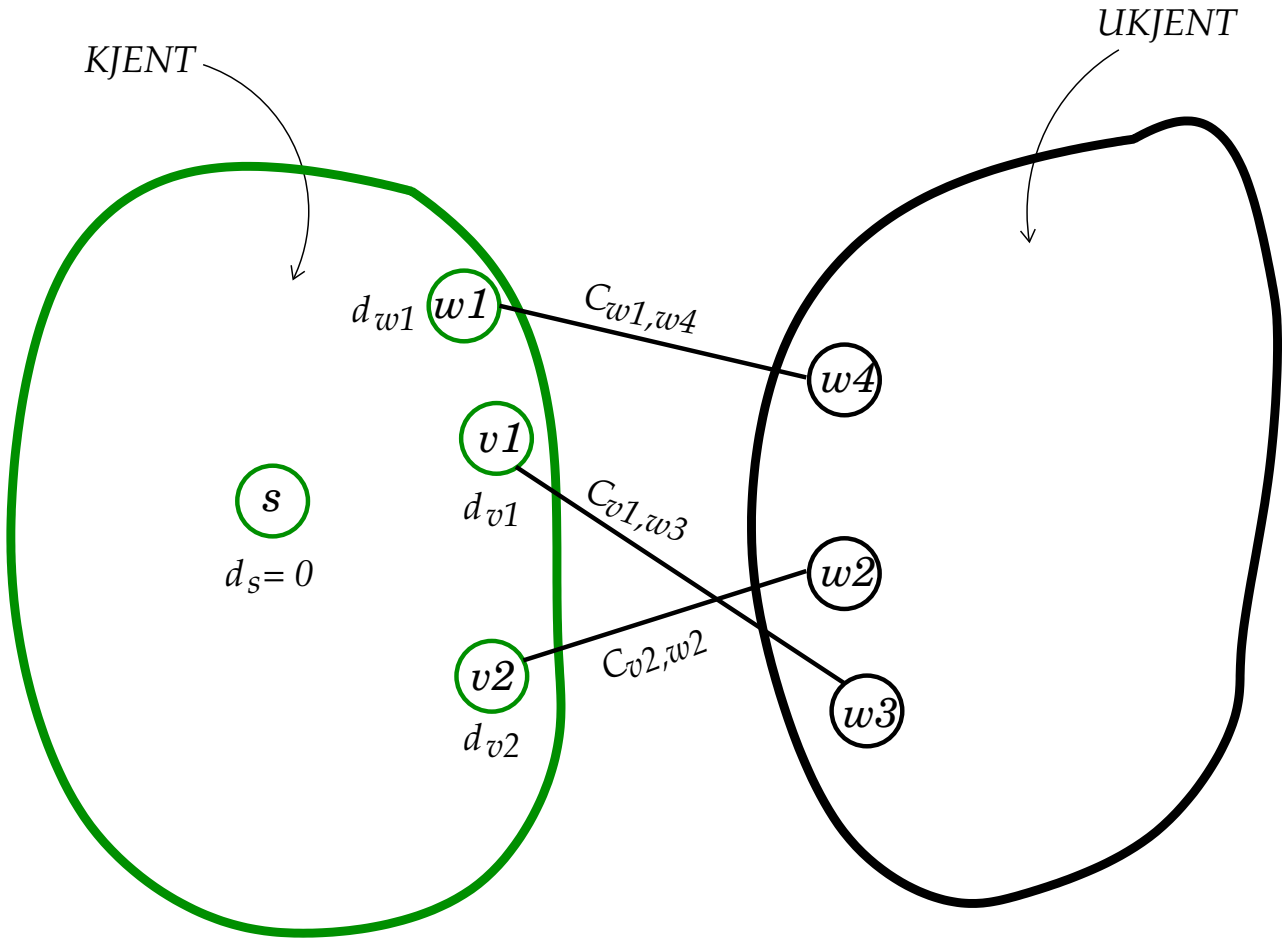


For hvert steg øker algo. mengden av kjente noder med 1. Hvilken node velger den??



Velg den noden som har minst avstand til s: Dijkstra grådig kriterier

$$MIN(\underbrace{d_{v1} + C_{v1,w1}}_{d_{w1}}, \underbrace{d_{v2} + C_{v2,w2}}_{d_{w2}}, \underbrace{d_{v1} + C_{v1,w3}}_{d_{w3}})$$


KJENT
UKJENT
d_{w1}
w1
$C_{w1,w4}$
w4
v1
$C_{v1,w3}$
s
d_{v1}
w2
$d_s = 0$
$C_{v2,w2}$
v2
w3
d_{v2}

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**:

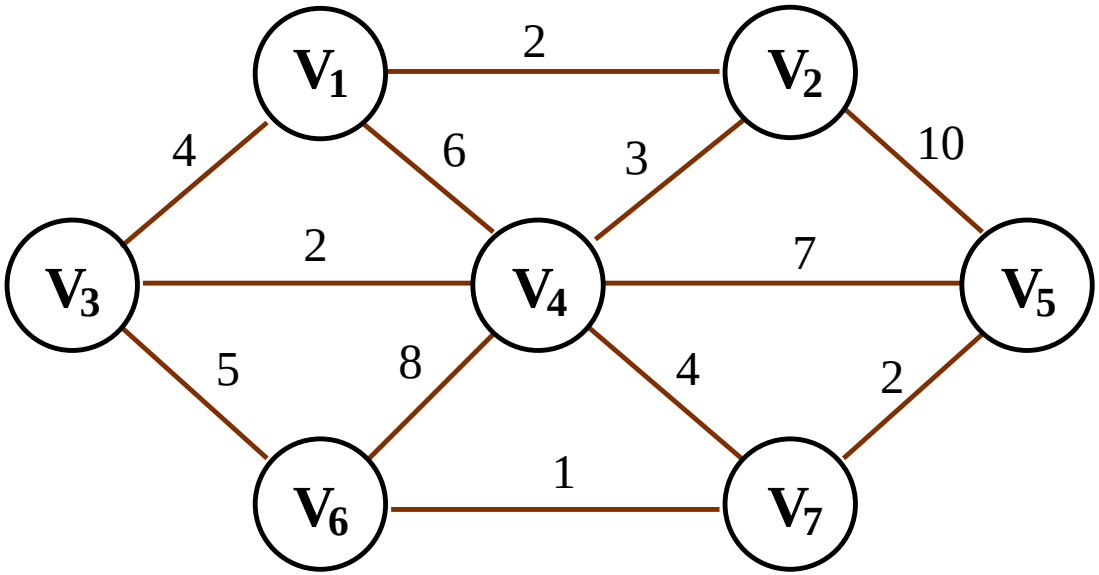
   Dersom avstanden vi får ved å følge veien gjennom **v**, er kortere enn den gamle avstanden til **s**
   - reduserer avstanden til **s** for **w**
   - sett bakoverpekeren i **w** til **v**
5. Akkurat som for uvektede grafer, ser vi bare etter potensielle forbedringer for naboer (w) som ennå ikke er valgt (kjent)

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

Eksempel

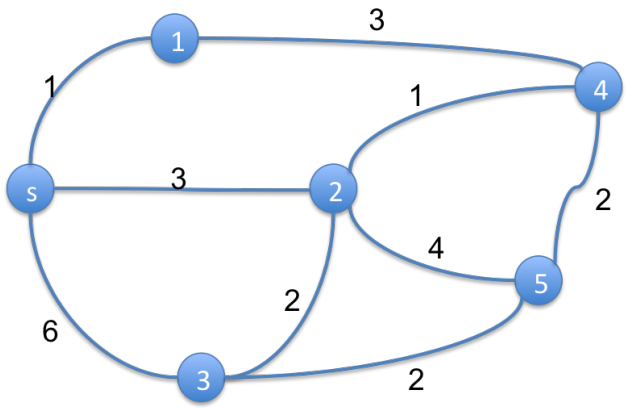
Input:

$s = V_1$



Oppgave

Consider the following graph. Assume that we use Dijkstra algorithm to find shortest paths from vertex s to the other vertices in the graph. In which order are the shortest paths computed?



1. s, 1, 2, 3, 4, 5
2. s, 1, 2, 4, 5, 3
3. s, 1, 4, 2, 3, 5
4. s, 1, 2, 4, 3, 5
5. s, 1, 4, 3, 2, 5

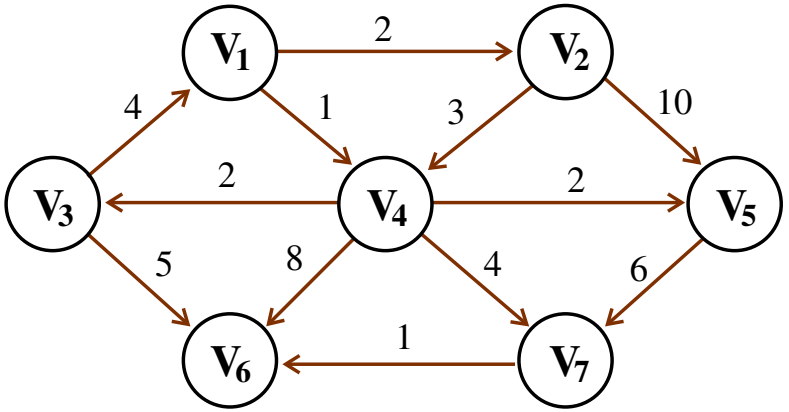
Oppgave

Given an undirected graph with negative edge weights, will Dijkstra's algorithm find the shortest path between two nodes correctly?

1. Yes
2. No

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			

Oppsummering

- Graf: Noder + Kanter
- Begrep: Rettet, urettet, inngrad, utgrad, sti...
- DAG: Rettet asyklisk graf (topologisk sortering)
- Avstand: Korteste vei en-til-alle

Neste forelesning: 28. september

Grafer II: mer om Dijkstra, Prim, Kruskal, Dybde-først søk