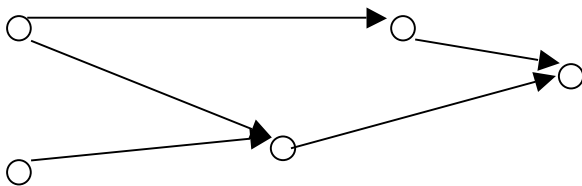
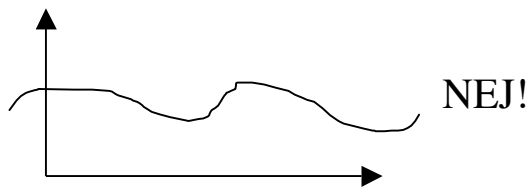


Grafer og grafalgoritmer

Hvad mener vi med en graf?



Graf: En matematisk abstraktion over ting som er “logisk” forbundet med hinanden

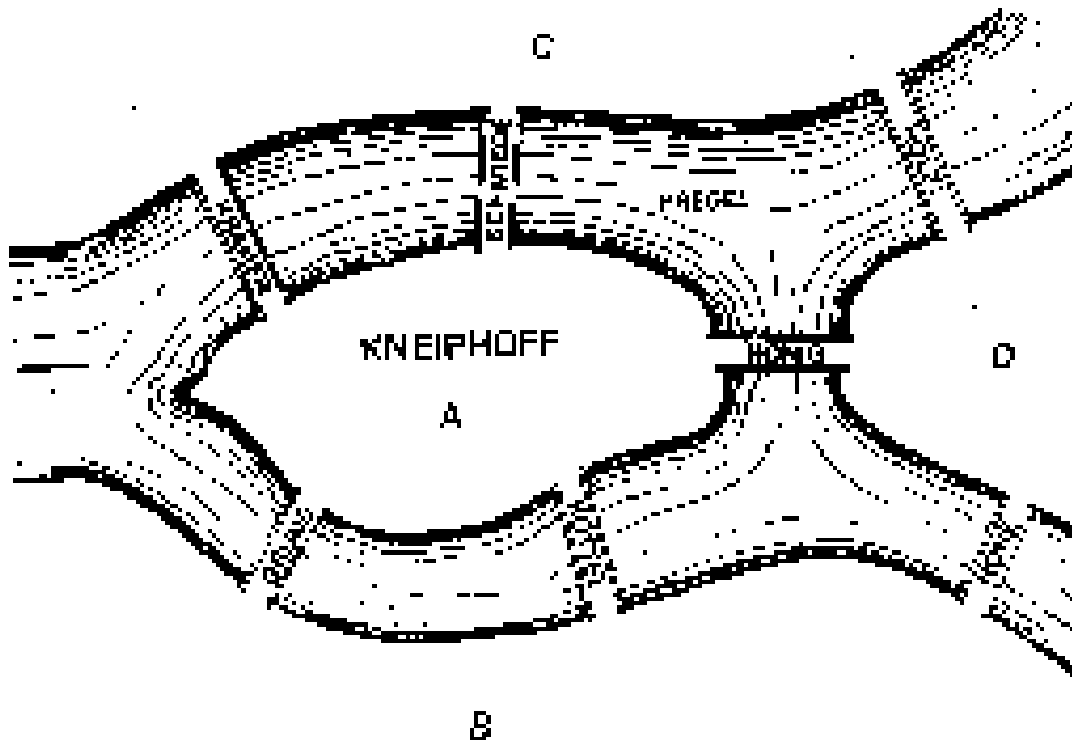
Dagens program:

- Terminologi og anvendelser
- Repræsentationer af grafer
- Algoritme til korteste vej
- Algoritme til korteste vej — med vægte (=omkostninger)
- Kort om netværksplanlægning
 - Princippet
 - Skitser af algoritmer, topologisk sortering, kritisk vej m.v.

Grafer, terminologi og baggrund

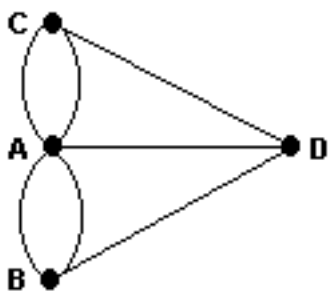
Udspringer af matematisk disciplin Grafteori

Leonard Euler 1736: “De 7 broer i Königsberg” (nu Kaliningrad)



“Er det muligt at gå en tur, så man passerer alle broer, men ikke mere end én gang”?

Svaret er nej... beviset baseret på abstraktion = grafer:



(Interesserede kan finde bevis på nettet eller i en bog på biblioteket)

Grafteori: Stort og spændende område, mange spændende sætninger — og anvendelser!

Terminologi:

En graf består af

- *knuder (vertices)*; en eller anden (abstrakt mængde)
 $\{1,2,3,4\}$
- *kanter (edges)*, en binær relation over knuder.
 $\{(1,2), (2,3), (2,2), \dots\}$

Forskellige typer grafer:

Ikke-orienteret graf:

Kanter uden retning, dvs. $(1,2) \approx (2,1)$

Orienteret graf:

Kanter har retning, dvs. $(1,2) \neq (2,1)$, dvs. Pile

Vi taler om ind-kanter og ud-kanter for given knude

Vægtet graf (typisk orienteret, men også ikke-orienteret):

Hver kant har en vægt (\approx pris \approx omkostning \approx ...)

Tætte og tynde grafer:

Tæt: Alle kan se alle,

dvs. antal kanter $\approx O(\text{antal-knuder}^2)$

Tynd: Hver knude har lille antal kanter til/fra

dvs. antal kanter $\approx O(\text{antal-knuder})$

Eksempel 2D-strukturer uden krydsende kanter

Fænomener i grafer:

En *vej (path)* fra K_1 til K_n :

En sekvens af kanter $(K_1, K_2), (K_2, K_3), \dots, (K_{n-1}, K_n)$.

En *kreds/cyklus (cycle)*: En vej fra K til K .

Klassifikation af grafer:

- *Acyklisk*: ingen kredse
- *Sammenhængende*: For vilkårlige knuder K_1, K_2 , altid vej fra K_1 til K_2 eller omvendt
- *Skov*: Aldrig mere end én vej mellem to knuder
- *Træ*: en sammenhængende skov :)
 - NB: et træ har altid én "*top-knude*"
(burde hedde dets rod, men nu er det altså etableret)

Anvendelser

Infrastruktur/forsyningslinjer

Knuder = Byer/Forbrugere-Producenter
/Computere-Servere-Printere

Kanter = Veje/Lastbilsruter/Togstrækninger/Vandrør
/el-ledninger/datanetforbindelser

Interessante spørgsmål:

- Er grafen sammenhængende (ellers øer)
- Billigste vej mellem to knuder
- Billigste vej som involverer givet punktmængde

Sammenhængsgrad \approx et mål for forsyningssikkerhed:

- hvor mange forskellige veje mellem to vilkårlige punkter

Rejseplanlægning:

Knuder = stationer/stoppesteder/adresser

Kanter = trafikforbindelser; pris = rejsetid

Ekstra komplikation: Køreplan => ventetid

Planlægning af projekter:

Knuder = Produkter (=aktivitet afsluttet)

Kanter = Hvilke ting forudsætter hvilke andre

Hvor lang tid tager aktivitet X?

Programstrukturer:

Kaldegtrafer: Hvilke metoder kalder hvilke?

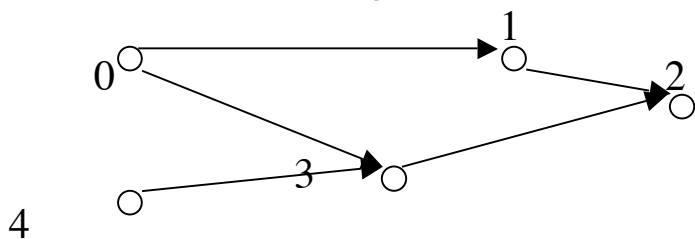
- Fejlsøgning, optimering på compiletime

UML: Hvilke klasser benytter hvilke og hvordan (f.eks. nedarver fra, indeholder objekter fra, ...)

- Vedligeholdelse af programmer...

Repræsentation af grafer

Matrice: bool [5][5] graf



0	1	0	1	0
0	0	1	0	0
0	0	0	0	0
0	0	1	0	0
0	0	0	1	0

Lister:

For hver knude, en liste over dens ud-kanter

0: [1, [3, null]]

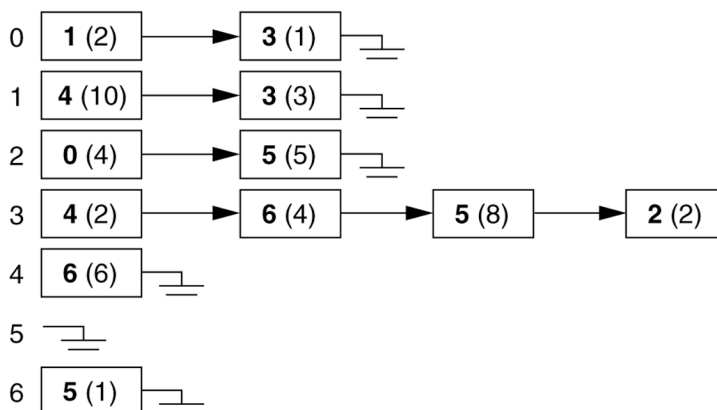
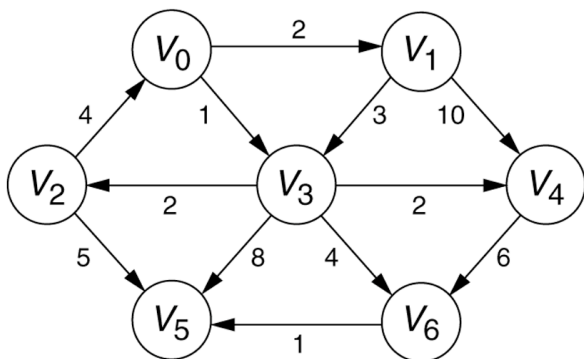
1: [2, null]

2: null

3: [2, null]

4: [4, null]

Eksempel på listerepræsentation af vægtede grafer:



Hvordan var det nu med `java.util.LinkedList`

Constructors

`LinkedList` ()

Constructs an empty list.

boolean `add` (`Object` o)

Appends the specified element to the end of this list.

`Object` `getFirst` ()

Returns the first element in this list.

`Object` `getLast` ()

Returns the last element in this list.

int `indexOf` (`Object` o)

Returns the index in this list of the first occurrence of the specified element, or -1 if the List does not contain this element.

int `lastIndexOf` (`Object` o)

Returns the index in this list of the last occurrence of the specified element, or -1 if the list does not contain this element.

`ListIterator` `listIterator` (int index)

Returns a list-iterator of the elements in this list (in proper sequence), starting at the specified position in the list.

`Object` `remove` (int index)

Removes the element at the specified position in this list.

boolean `remove` (`Object` o)

Removes the first occurrence of the specified element in this list.

`Object` `removeFirst` ()

Removes and returns the first element from this list.

`Object` `removeLast` ()

Removes and returns the last element from this list.

Osv. osv.

Repræsentation med lister benytte i lærebogen

(her rensset for diverse ekstrainformation og fejlcheck ved indlæsning)

- Attributter har her tekstlige navne som benyttes ved indlæs og udskriv

```
class Edge // start vertex given by position in list (below)
{public Vertex    dest; // Second vertex in Edge
  public double   cost; // Edge cost
  public Edge( Vertex d, double c ) {dest = d; cost = c;}}
```

```
class Vertex
{ public String   name; // Vertex name
  public List     adj;  // Adjacent vertices //liste over ud-kanter
  public Vertex(String nm){name=nm;adj=new LinkedList( );}}
```

```
public class Graph
{private Map vertexMap = new HashMap( ); // String to Vertex

  private Vertex getVertex( String vertexName )
  {Vertex v = (Vertex) vertexMap.get( vertexName );
   if(v==null)
     {v=new Vertex(vertexName); vertexMap.put(vertexName,v);}
   return v;}
```

```
public void addEdge( String sourceName, String destName, double cost)
{Vertex v=getVertex(sourceName); Vertex w=getVertex(destName);
  v.adj.add( new Edge( w, cost ) );}
```

```

public static void main( String [ ] args )
{ Graph g = new Graph( );
  FileReader fin = new FileReader( args[0] ); //file name
  BufferedReader graphFile = new BufferedReader( fin );
  // Read the edges and insert
  String line;
  while( ( line = graphFile.readLine( ) ) != null )
  {StringTokenizer st = new StringTokenizer( line );
    String source = st.nextToken( ); String dest = st.nextToken( );
    int cost = Integer.parseInt( st.nextToken( ) );
    g.addEdge( source, dest, cost );};
  System.out.println( "File read..." );
  System.out.println( g.vertexMap.size( ) + " vertices" );
  .....
}

```

Korteste vej mellem to knuder

- beregner korteste veje fra startknode til alle andre knuder
a la princippet i dynamisk programmering;
vi har måske brug for at gå via andre knuder

Version 1: Orienterede grafer uden vægte; vejlængde = antal kanter.

Datastruktur:

hver knude tilføjes hjælpevariable:

```
double dist; //hidtil bedst fundne afstand fra start; init=INFINITY  
Vertex previous; // Previous vertex on shortest path; init=null
```

Metode clearAll, som initialiserer alle hjælpevariable i graf

global kø “q” over knuder som skal besøges.

Princip i algoritme:

Givet knude kalde **start**.

Først besøges start;

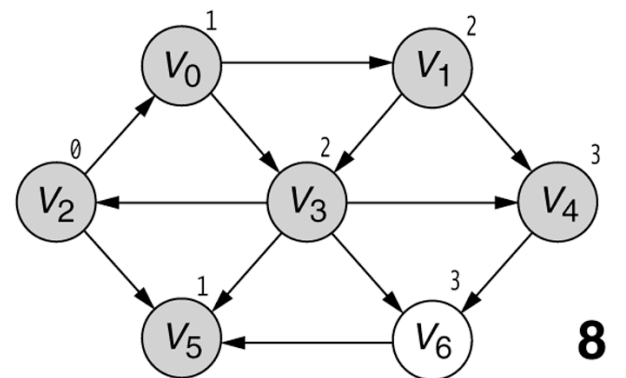
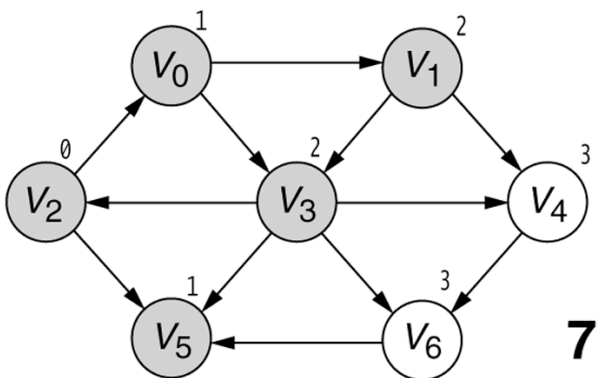
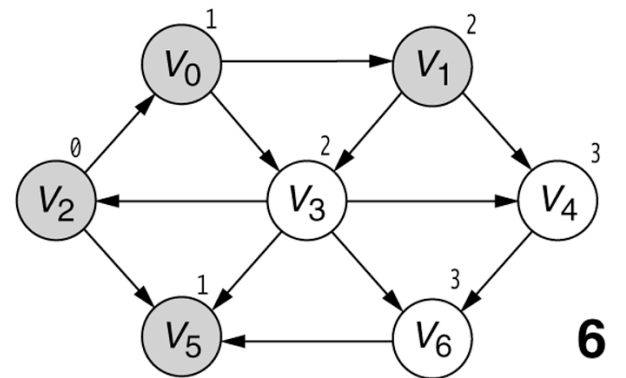
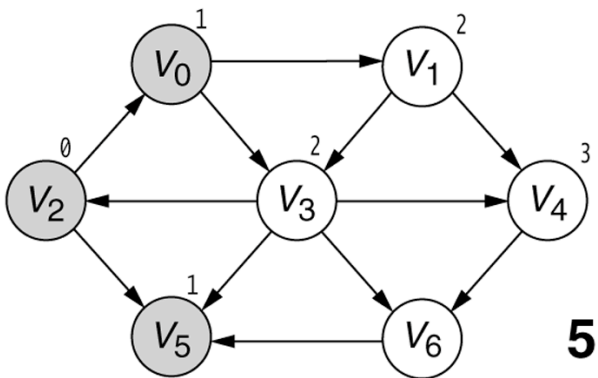
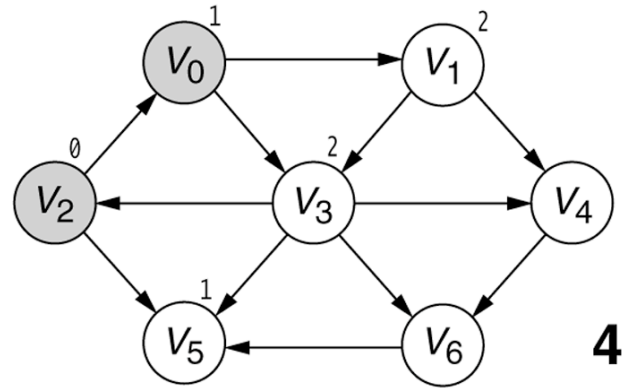
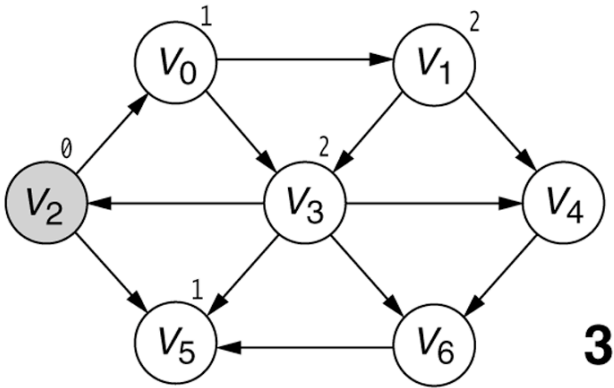
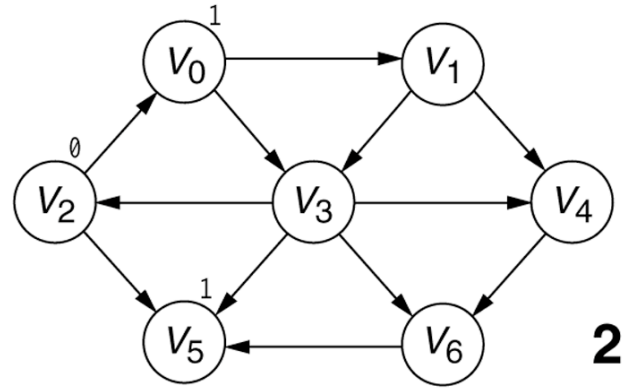
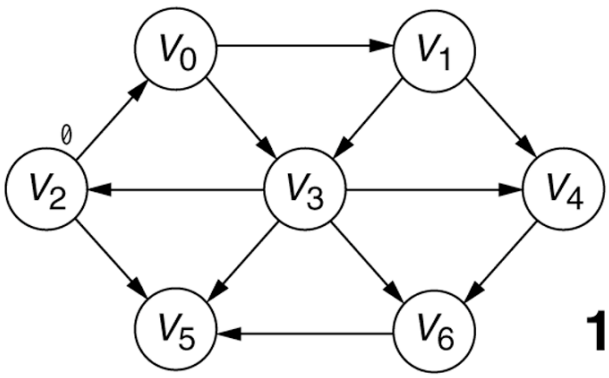
Så besøges alle knuder i afstand 1 fra start

- dvs. alle som kan nås via én kant fra start

Dernæst alle knuder i afstand 2 fra start;

- dvs. alle som kan nås via én kant fra dem i forrige skridt,
dog ikke dem som allerede har været besøgt!

osv.



Implementation i Java

```
public void unweighted( String startName )
{ clearAll( );
  Vertex start = (Vertex) vertexMap.get( startName );
  LinkedList q = new LinkedList( );
  q.addLast( start ); start.dist = 0;

  while( !q.isEmpty( ) )
  { Vertex v = (Vertex) q.removeFirst( );
    for( Iterator itr = v.adj.iterator( ); itr.hasNext( ); )
    { Edge e = (Edge) itr.next( );
      Vertex w = e.dest;
      if( w.dist == INFINITY )
      { w.dist = v.dist + 1;
        w.prev = v;
        q.addLast( w );}}}}}
```

Sluttilstand:

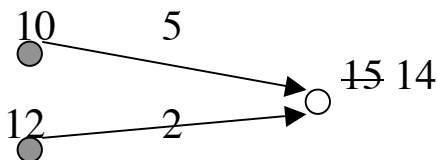
Alle knuder med påskrevet afstand fra start.

Incl. dem som ikke kan nås fra start: "INFINITY"

Korteste vej med vægte, “Dijkstras algoritme”

Princip, som før men ekstra komplikation i og med

- ikke nok med besøgt/ikke besøgt
- vægt for knude kan tælles ned flere gange



Vi skelner mellem

- færdigbehandlede knuder (“visited”)
- ufærdige naboer til færdigbehandlede knuder (“unseen”)

For at generalisere algoritmen fra før bruger vi en *prioritetskø*, som indeholder den sidste slags.

Alle knuder tildeles fra start afstand fra startknude = uendelig.

Abstrakt algoritme:

indsæt start-knude med afstand=0 i kø;

while(der-er-flere-kø) {

 lad v = knuden i køen med mindst afstand... v fjernes fra køen;

 for alle v 's naboer w , som ikke allerede er færdigbehandlede,

 { hvis v .afstand + længde($v \rightarrow w$) < w .afstand så

 { w .afstand = v .afstand + længde($v \rightarrow w$);

 sæt w i kø (hvis den ikke allerede er der); }

 notér v som færdigbehandlet } }

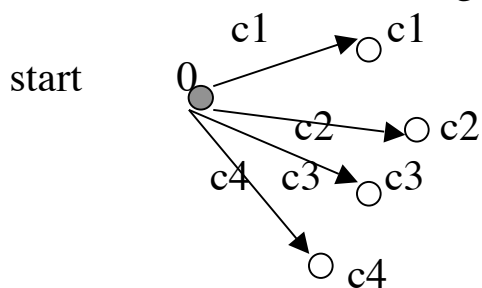
Hvorfor virker den?

Lad os formulere et matematisk induktionsbevis over antal gennemløb af while-løkken:

Hypotesen:

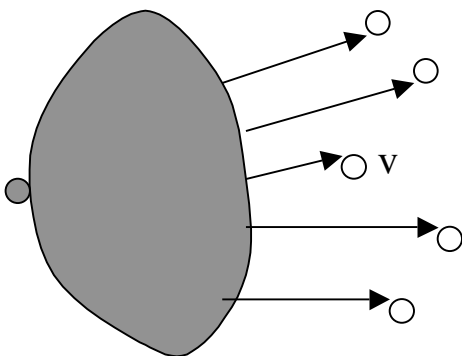
- Afstand for alle færdigbehandlede er korrekt, og
- Afstand for “ufærdige nabo” $w = \text{lgd. af korteste vej i delgraf som udover } w \text{ kun indeholder færdigbehandlede.}$

Induktionsstart, lad os vælge når startknuden er færdigbehandlet:



OK!

Antag nu at algoritmen har kørt korrekt op til skridt k :



Skridt $k+1$: Lad v være “hvid” knude med mindst vægt:

- v 's naboers vægte justeres (måske)
- v forfremmes til færdigbehandlet

1. *Redegøre for at v 's afstand D er den mindst mulige. Men kan der være en anden vej fra start til v , som er billigere? Per induktionshypotese del 2, kan der ikke være en anden vej grå->grå->...->grå-> v . Ergo må en sådan vej P være af formen ... -> hvid -> grå. Kald denne hvide v_1 og dens noterede afstand D_1 . Udfra valget af v må vi have $D_1 \geq D$. Kan længden af $P = D_1 + \text{noget-positivt}$ så være mindre end D ?? Nej vel? Modstrid!*
2. *Redegøre for at de nye afstande for v 's naboer er korteste vej i delgrafnen bestående af de grå – nu udvidet med v . Hvis en sådan nabo w får talt sin vægt ned, så er det netop fordi ny vej via v er kortere end tidligere grå->grå->...->grå-> w .*

Q.E.D.

Bogens algoritme i Java:

Prioritetskøen implementeres lidt snusket med BinaryHeap... (NB: ikke i Javas std. pakker)

BinaryHeap er en datastruktur,

- med underligt navn,
- hvis indmad vi ikke kender

men som implementerer metoderne

- insert(Comparable x)
- Comparable deleteMin()

hvor “Min” er bestemt ved objekternes compareTo.

Problem ved denne BinaryHeap: Objekternes rækkefølge bestemmes på indsættelsestidspunktet.

Dvs. hvis objekt ændres undervejs, så det påvirker resultat af compareTo, opfører deleteMin sig ikke korrekt!!!

Teknik (læs: hæsligt bøjet søm):

- I algoritmen ændres knudernes “afstand” altid nedad
- Vi vælger altid knude med mindst “afstand”
- Når knude er valgt, sættes den “scratch” felt = 1 (init=0).
- Hvis en knude med “scratch =1” fremkommer ved deleteMin, ignoreres den.

Følgende “Path” objekter indsættes i BinaryHeap’en_

```
class Path implements Comparable
{ public Vertex    dest; // w
  public double    cost; // d(w)
  public Path( Vertex d, double c ) {dest = d; cost = c;}
  public int compareTo( Object rhs )
  { double otherCost = ((Path)rhs).cost;
    return cost < otherCost ? -1 : cost > otherCost ? 1 : 0;}}
```

Dijkstras algoritme for korteste vej i vægtede grafer ved brug af “BinaryHeap”:

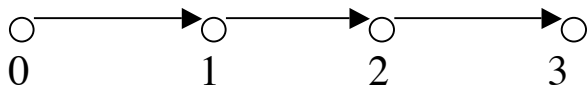
```
public void dijkstra( String startName )
{ PriorityQueue pq = new BinaryHeap( );
  Vertex start = (Vertex) vertexMap.get( startName );
  clearAll( );
  pq.insert( new Path( start, 0 ) ); start.dist = 0;
  int nodesSeen = 0;

  while( !pq.isEmpty( ) && nodesSeen < vertexMap.size( ) )
  { Path vrec = (Path) pq.deleteMin( );
    Vertex v = vrec.dest;
    if( v.scratch != 0 ) continue; // already processed v
    v.scratch = 1; nodesSeen++;

    for( Iterator itr = v.adj.iterator( ); itr.hasNext( ); )
    { Edge e = (Edge) itr.next( );
      Vertex w = e.dest;
      double cvw = e.cost;
      if( w.dist > v.dist + cvw )
      { w.dist = v.dist + cvw;
        w.prev = v;
        pq.insert( new Path( w, w.dist ) );}}}}}
```

Acykliske grafer

Typisk anvendelse: Planlægning



0: tagspær

1: lægter

2: tagsten

3: rejsegilde

A --> B: A forudsætter B

Acykliske grafer nemmere at programmere om: man kommer aldrig tilbage til samme sted!

Eksempel: Topologisk sortering

at ordne knuderne i en graf i rækkefølge, så afhængigheder er overholdt.

“tagspærene skal på før lægterne”

“hvorvidt du sætter vinduer i før eller efter tagstenene er ligegyldigt”

Algoritme som udskriver knuderne i topologisk sorteret rækkefølge

G = vor graf;

```
while(flere-knuder-tilbage) {
```

```
    find knude V med nul indgående kanter; //findes fordi G acyklisk  
    udskriv V;
```

```
    fjern v og alle dens udgående kanter fra G; }
```

Bogens algoritme i Java (udsnit af fig. 14.32)

(fig. 14.32 blander top.sort. med en kortestevej algoritme)

Princip: hver knudes “scratch” benyttes som tæller for antal ind-kanter
der bruges en kø til at holde de knuder som på givet tidspunkt
har 0 ind-kanter

```
LinkedList q = new LinkedList( );
```

```
// Compute the indegrees
```

```
Collection vertexSet = vertexMap.values( );
```

```
for( Iterator vsitr = vertexSet.iterator( ); vsitr.hasNext( ); {
```

```
    Vertex v = (Vertex) vsitr.next( );
```

```
    for( Iterator witr = v.adj.iterator( ); witr.hasNext( ); )
```

```
        ( (Edge) witr.next( ) ).destscratch++;}
```

```
// Enqueue vertices INITIALLY of indegree zero
```

```
for( Iterator vsitr = vertexSet.iterator( ); vsitr.hasNext( ); ) {
```

```
    Vertex v = (Vertex) vsitr.next( );
```

```
    if( v.scratch == 0 ) {q.addLast( v ); UDSKRIV v; } ;
```

```
// Loop: Remove 0-nodes from queue and count down in-degrees
```

```
while(!q.isEmpty( )) {
```

```
    Vertex v = (Vertex) q.removeFirst( );
```

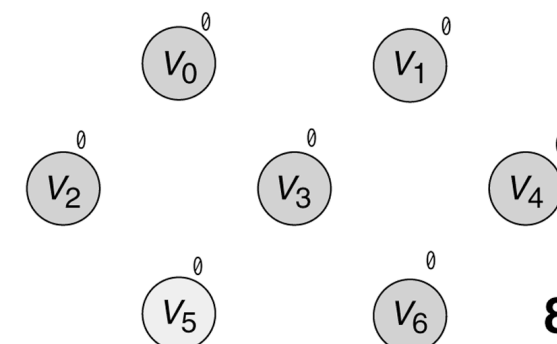
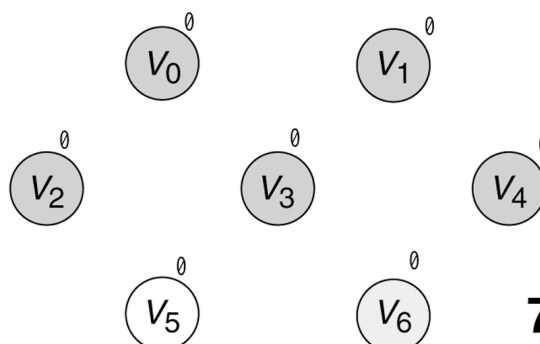
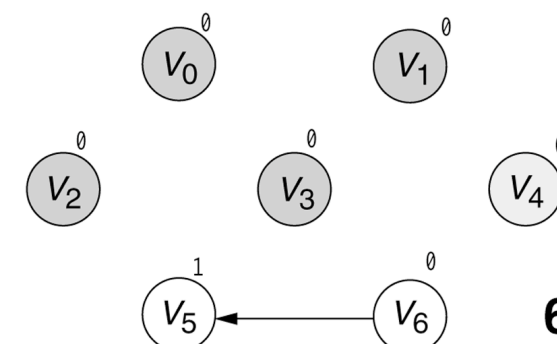
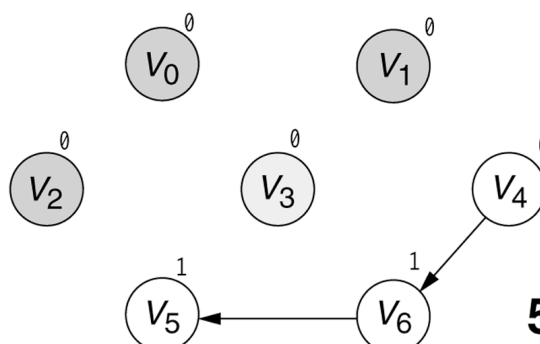
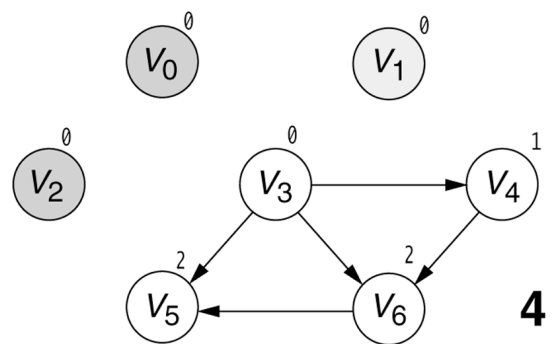
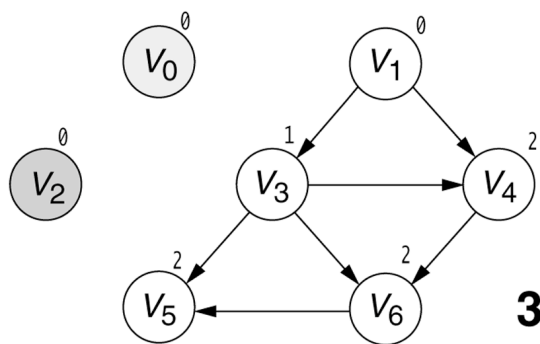
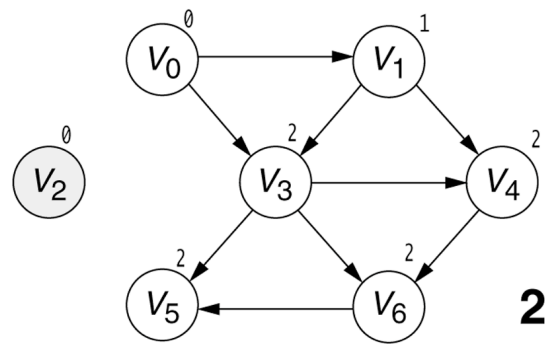
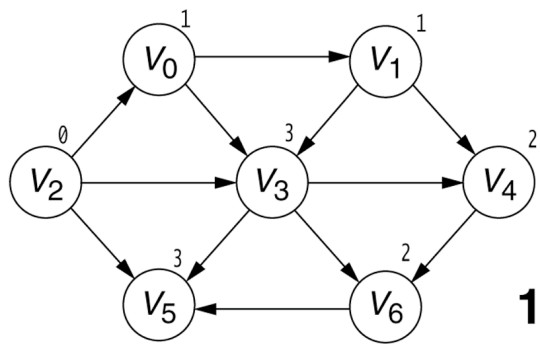
```
    for( Iterator itr = v.adj.iterator( ); itr.hasNext( ); ) {
```

```
        Edge e = (Edge) itr.next( );
```

```
        Vertex w = e.dest;
```

```
        if( --w.scratch == 0 ) {q.addLast( w );; UDSKRIV v; }}}
```

Eksempel fra bogen



Korteste vej for acykliske grafer

Tricket:

Knuderne besøges i rækkefølge svarende til topologisk sortering
Dvs. når en knude besøges, er alle dens forgængere besøgt
(og med garanti ikke ændrer sin korteste vej)

Algoritmen for topologisk sortering med 4 ekstra linjer i stedet for udskrift:

```
LinkedList q = new LinkedList( );
```

```
// Compute the indegrees
```

```
Collection vertexSet = vertexMap.values( );  
for( Iterator vsitr = vertexSet.iterator( ); vsitr.hasNext( ); {  
    Vertex v = (Vertex) vsitr.next( );  
    for( Iterator witr = v.adj.iterator( ); witr.hasNext( ); )  
        ( (Edge) witr.next( ) ).dest.scratch++;}
```

```
// Enqueue vertices INITIALLY of indegree zero
```

```
for( Iterator vsitr = vertexSet.iterator( ); vsitr.hasNext( ); ) {  
    Vertex v = (Vertex) vsitr.next( );  
    if( v.scratch == 0 ) q.addLast( v );
```

```
// Loop: Remove 0-nodes from queue and count down in-degrees
```

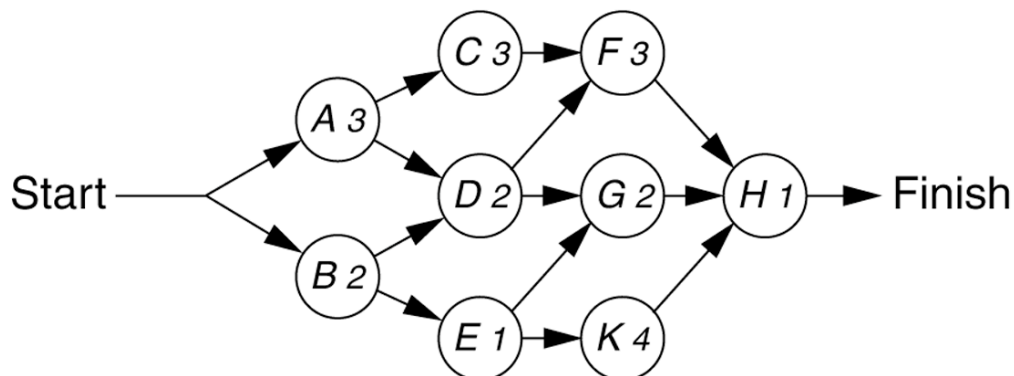
```
while(!q.isEmpty( )) {  
    Vertex v = (Vertex) q.removeFirst( );  
    for( Iterator itr = v.adj.iterator( ); itr.hasNext( ); ) {  
        Edge e = (Edge) itr.next( );  
        Vertex w = e.dest;  
        if( --w.scratch == 0 ) q.addLast( w );  
        if( v.dist == INFINITY ) continue; // on "island"; avoid overflow  
        if( w.dist > v.dist + cvw )  
            { w.dist = v.dist + cvw;  
              w.prev = v;}}
```

Skitse af netværksplanlægning

Udgangspunkt: En *aktivitetsgraf*

Knuder: Aktivitet m. forventet tidsforbrug

Kanter: Forudsætningsforhold

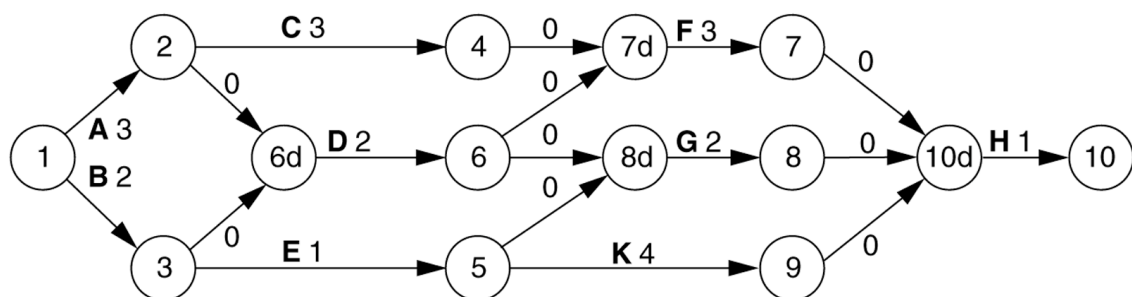


Obs: Omkostninger på knuder passer ikke ind i vores hidtidige model.

I stedet: En begivenhedsgraf:

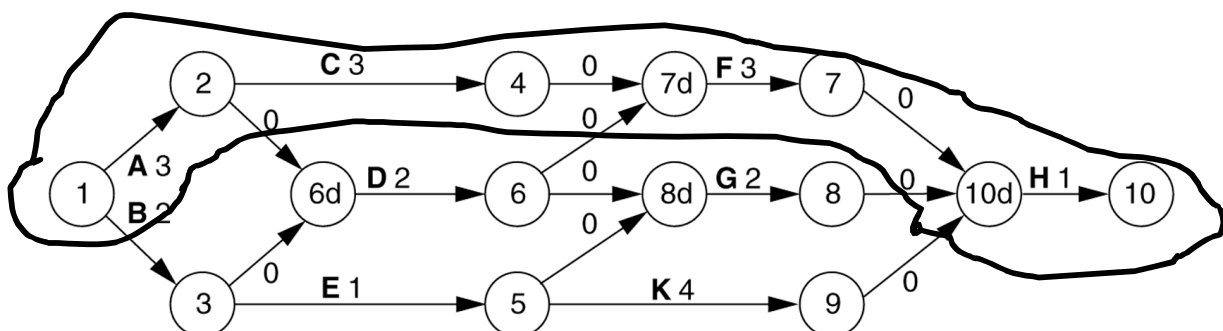
Knuder: aktivitet-start, aktivitet-slut

Kanter. Forudsætningsforhold, Aktivitet m. tidsforbrug

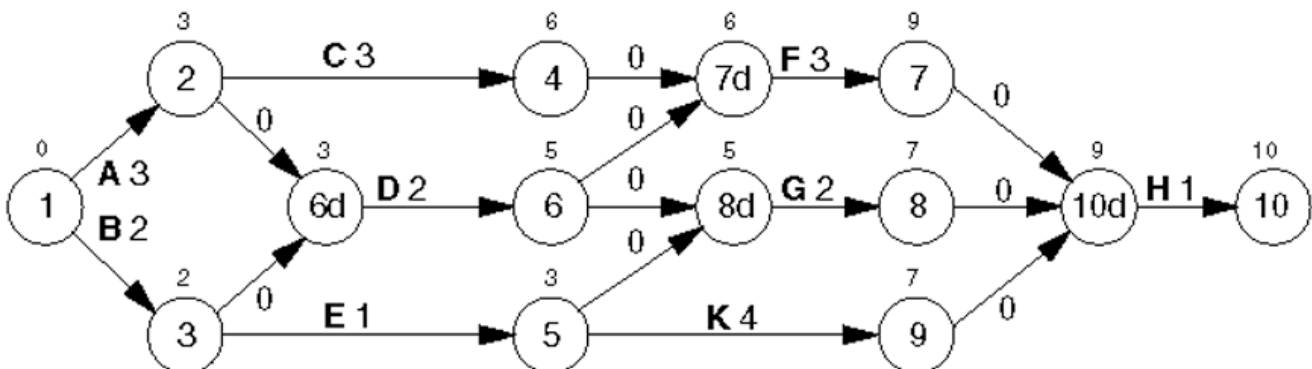


Hvor lang tid tager projektet hvis arbejdet planlægges optimalt?

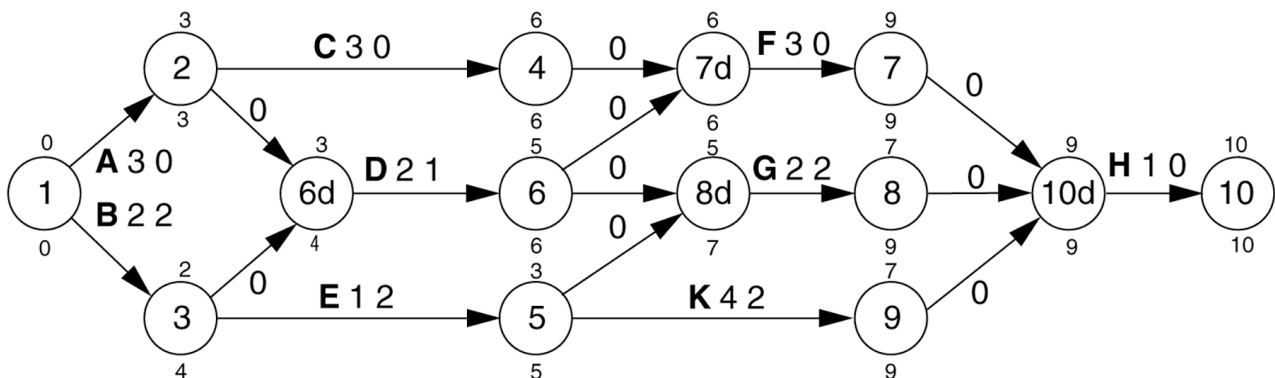
Kritiske vej = længste vej fra start til slut



Algoritmer a la korteste vej kan tilpasses "længste vej" \approx
for hver aktivitet, tidligst mulige slutpunkt



Yderligere algoritmer til:
Senest mulige starttidspunkt
Tidligst mulige sluttidspunkt
Hvor meget der kan slækkes



Yderligere forfining af modellen med ressourcer:
Aktivitet A kræver 1 gravko m. fører, 1 rende-graver m. fører
Aktivitet E kræver 1 rende-graver m. fører, 1 bobcat, 5 gartnere,
3 river, 2 skovle

Vi har ialt følgende ressourcer:
1 gravko, 2 rende-graver, 1 byggekran, ...

Alt dette kan behandles med algoritmer
små forbedringer = store penge
heuristikker til at behandle usikkerheder,
robusthed for forudset og uforudsete ændringer