Grafer

I. GRAF

definisjon / terminologi

noen eksempler på graf-problemer

II. NOEN ENKLE GRAFALGORITMER

III. GRAF TRAVERSERING

DFS og BFS

IV. GRAF ADT OG IMPLEMENTASJON

Kant-Liste og Nabo-Liste

Nabo-Matrise

V. RETTEDE GRAFER

topologisk sorterting, transitiv tillukking

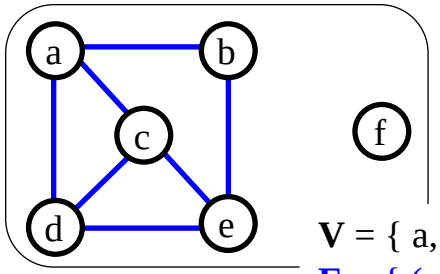
VI. VEKTEDE GRAFER

kortest sti, minste utspennede tre

Kap. 12 (kursorisk 12.4.4, 12.7.2)

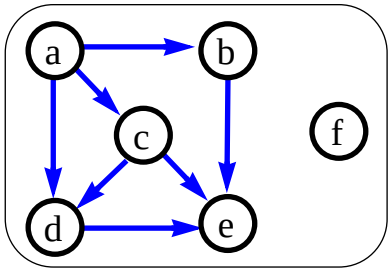
I. En graf G

er gitt ved to mengder V og E; G=(V,E)
V er noder (boka: vertices)
E er kanter (boka: edges)
en kant e ε E er et (uordnet)
par {u,v} av noder u ε V og v ε V

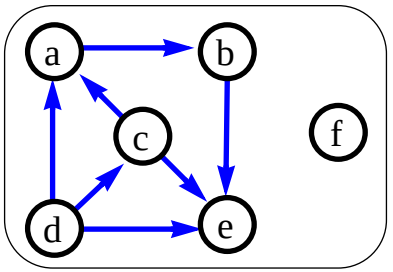


V = { a, b, c, d, e, f }
E = { (a,b), (a,c), (a,d), (b,e),
(c,d), (c,e), (d,e) }
= { (b,a), (c,a), (d,a), (e,b),
(d,c), (e,c), (e,d) }

En **rettet graf** (diGraf)
V av noder
E av kanter
der en kant e ε E er et ordnet
par (u,v), u ε V, v ε V, dvs u → v



E = { (a,b), (a,c), (a,d), (b,e),
(c,d), (c,e), (d,e) }



E = { (a,b), **(c,a)**, **(d,s)**, (b,e),
(d,c), (c,e), (d,e) }

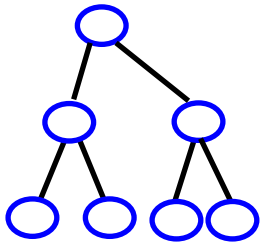
En **ikke-rettet graf** kan sees på som en (rettet) graf der relasjonen E er **symmetrisk** u → v ε E *hviss* v → u ε E.

En (rettet) **graf** er gitt ved
en mengde V av noder
en **binær relasjon** E ⊆ V × V

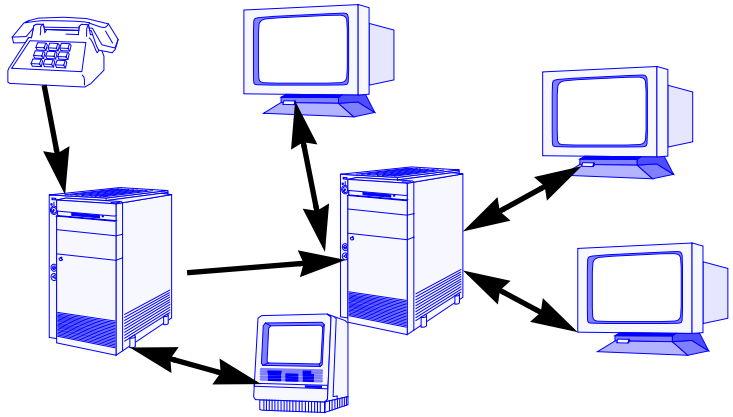
Anvendelser ...

I. Binære relasjoner

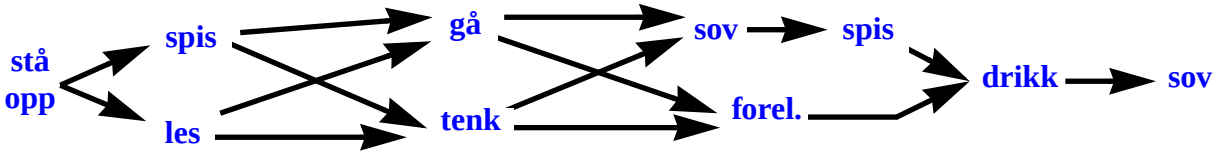
II. Trær



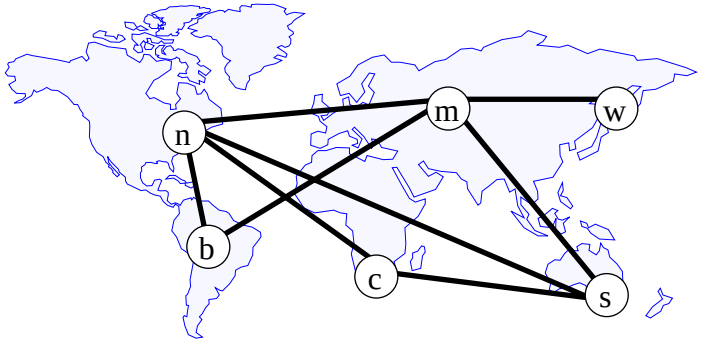
III. Nettverk

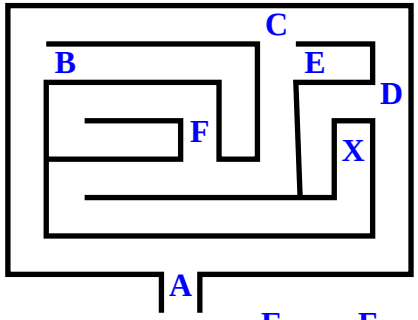


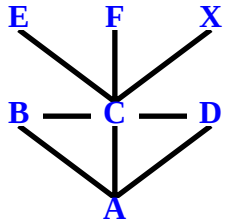
IV. Planlegging



V. Forbindelser







. . .

Graf-terminologi

nabonoder : 2 noder som er forbundet med en kant — a-c, c-e

gradtall til en node : antall nabonoder (kanter): — deg(c)=3, deg(f)=0

$$\sum_{v \in V} deg(v) = 2(\text{ \# kanter })$$

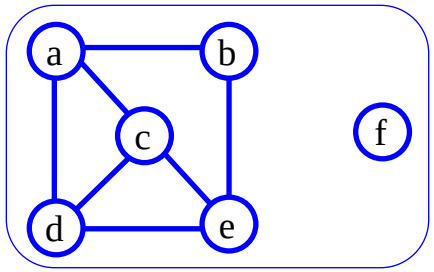
inngrad/utgrad : antall innkommende/utgående kanter (rettede grafer)

sti : en sekvens n_1, n_2 ... n_k av noder slik at $(n_i, n_{i+1}) \in E$

acaca, bec, abedc, edce

enkel sti : en sti hvor ingen node forekommer 2 ganger

sykel : en sti hvor $n_1 = n_k$

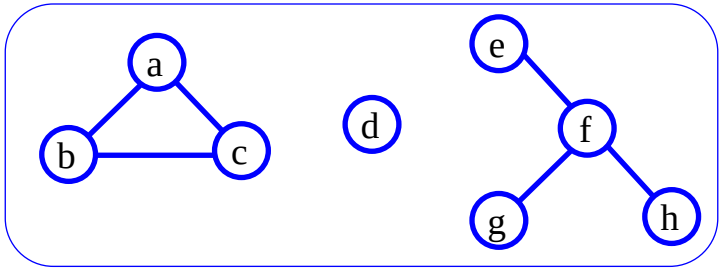


sammenhengende graf : graf hvor det finnes en sti mellom alle par av noder

delgraf : en delmengde av V og E som er en graf

sammenhengende komponent : en maksimal sammenhengende delgraf

komplett graf : om det for hvert par av noder u, v ε V, finnes en kant (u,v) ε E

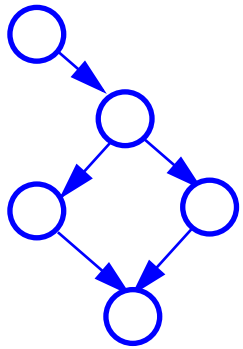


(ikke-rotet) **tre** : sammenhengende graf uten sykler

utspennende tre for en graf G : en delgraf av G som er et tre og inneholder alle G's noder

skog : samling av trær

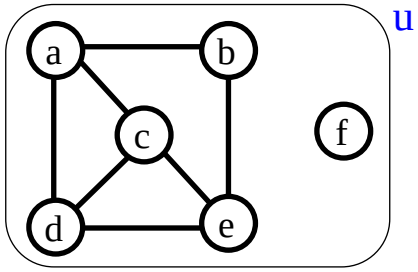
DAG :rettet graf uten sykler (directed acyclic graph)



Graf

URETTET:

en ikke-rettet kant (u,v)
= to rettede kanter
u → v og v → u



sti : en sekvens n_1, n_2 ... n_k av noder slik at (n_i,n_{i+1}) ε E

syklus: enkel sti (hver node 1 gang) men $n_1=n_k$

sammenhengende graf : det finnes en sti mellom alle par av noder

Er v oppnåelig fra u ?

Finn alle v oppnåelige fra u.

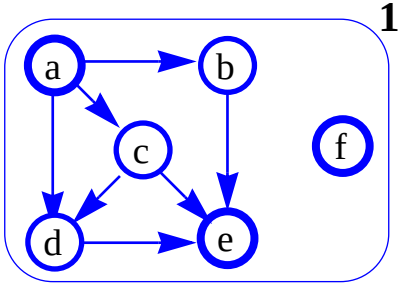
Er G sterkt sammenhengende ?

Er G asyklisk ?

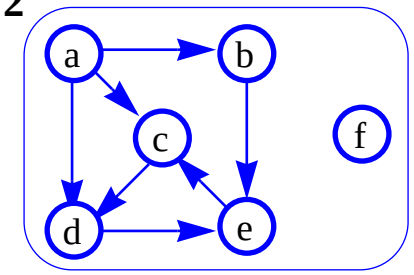
RETTET (DiGraph):

hver kant (u, v) er et ordnet (rettet) par u → v.

1



2



sti : en sekvens av fra-til kanter ... : ade (ikke eda)

rettet syklus: rettet enkel sti ... **2**: cde

kilde/sluk : en node uten noen inngående / utgående kanter **1**: a/e

oppnåelig (eng: reachable) :en node v kan nåes fra u dersom det finnes en rettet sti fra u til v
e oppnåelig fra a, men ikke omvendt

sterkt sammenhengende graf : enhvernodeueroppnåeligfraenhverannennodev hverken **1** eller **2**

DAG : rettet, asyklisk graf – ingen (rettede) sykler
1 er DAG, men ikke **2**

transitiv tillukning G* av G : Har kant u → v hviss G har en sti fra u til v

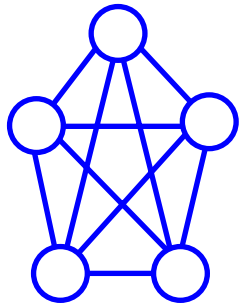
II. Antall noder og kanter

n = # noder i grafen, ***k*** = # kanter i grafen

- G er **komplett** **hviss** hver node har (***n*** –1) naboer

  dvs. ***k*** = 1/2 * $\sum_{v \in V} deg(v)$ = 1/2 * ***n*** * (***n*** –1)

n =5
k =10



- G **ikke komplett** **hviss** ***k*** < 1/2 * ***n*** * (***n*** –1)

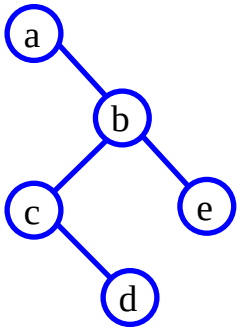
- **Hvis** G **er et tre så er** ***k*** = ***n*** –1

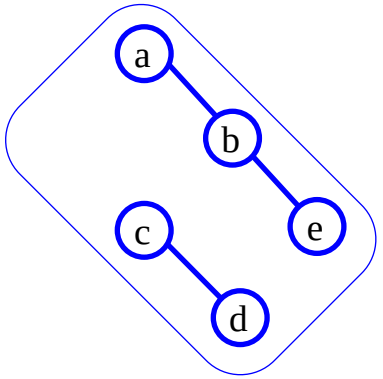
  - dette er det minste antall kanter som kan gi en sammenhengende graf på ***n*** noder. Bevis?

  - fjerning av en vilkårlig kant, gjør treet usammenhengende.

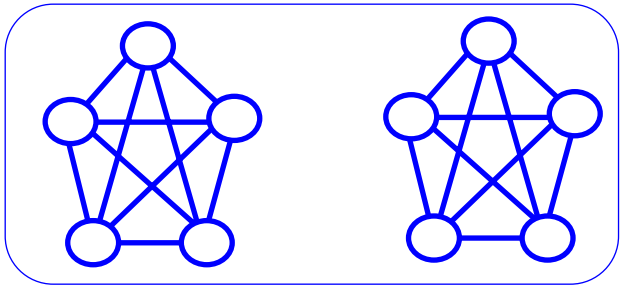
  - G trenger ikke å være et tre selv om ***k*** = ***n*** –1 (G kan være usammenhengende)

n =5
k =4





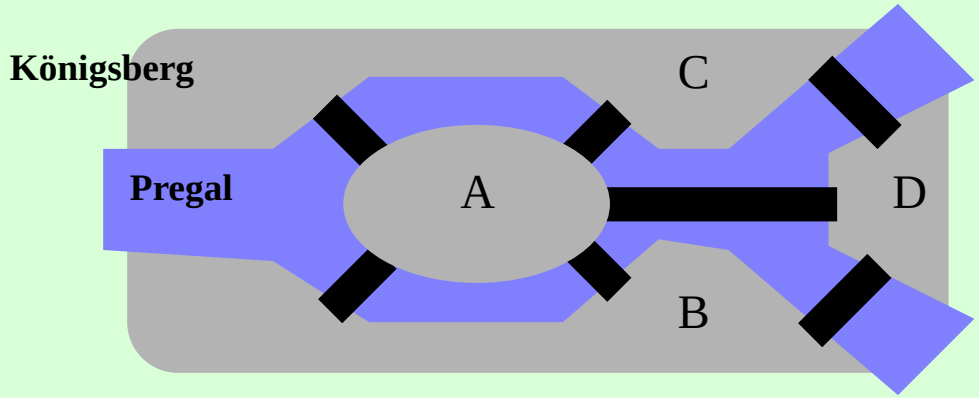
- **Hvis** ***k*** < ***n*** –1 **så er ikke** G **sammenhengende**
  men ikke omvendt. Hvorfor ikke?



n =10
k =20 > 9

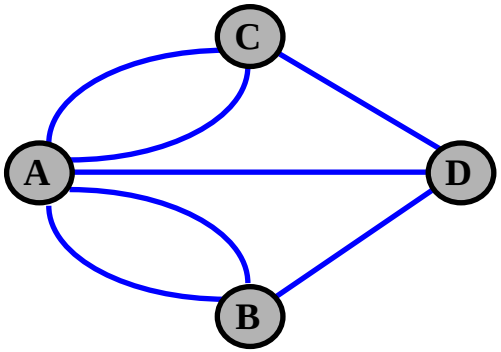
II. *Eulers tur i en vilkårlig graf*

Euler:
Kan jeg under min aftentur krysse hver bro nøyaktig én gang og returnere til startpunktet?



Euler (1736) :
Nei – og jeg kan bevise det v.hj.a. grafer !

Bruk multigrafer (som under), eller la broer være noder med gradtall 2.



Eulers tur : en sykel som traverserer hver kant i grafen nøyaktig én gang

Eulers teorem : En (sammenhengende) graf G har en Eulers tur hvis og bare hvis G har 0 eller 2 noder hvor gradtallet er et oddetall. Hvorfor? 'Lett' å finne:**O(n+k)**

Hamiltonsk sykel : en sykel som traverserer hver node nøyaktig en gang. Vanskelig å finne: **O(n!)**

... **...**

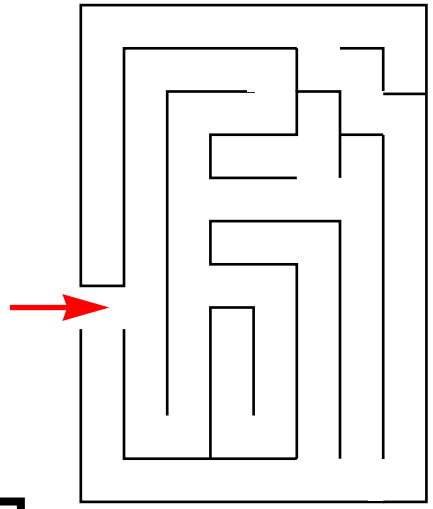
III. *Ariadnes (t)råd*

Minos :

OK, Theseus, but you must first find and kill Minotaur hidding in the Labirynth.

Theseus :

I'm good at killing but how the heck am I going to find it and then get out of there **?**



Ariadne :

(she felt in love with Theseus in the meantime...)

Ta denne tråden og fest den ved inngangen til Labirynten.

(**k**) Hver gang du kommer til et nytt kryss **u**, merk **u** '**visited**'.

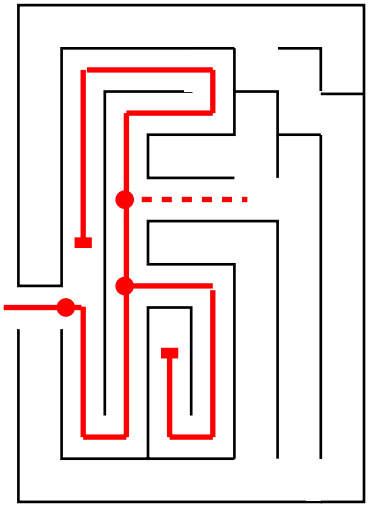
(**r**) Velg en vilkårlig vei – merk den og følg men **dra tråden** etter deg

Når du så kommer til et nytt kryss **v** : dersom det er

- en blind gate, gå tilbake **langs tråden** (**nøst** den igjen)
- et kryss merket '**visited**', gå tilbake **langs tråden** (**nøst** den igjen)
- et umerket kryss, **gjenta** det hele (**k**)

Etter hvert vil alle veier (**r**) fra krysset **u** bli merket

- gå da tilbake **langs tråden** til forrige krysset og fortsett å utforske umerkede veier derfra.



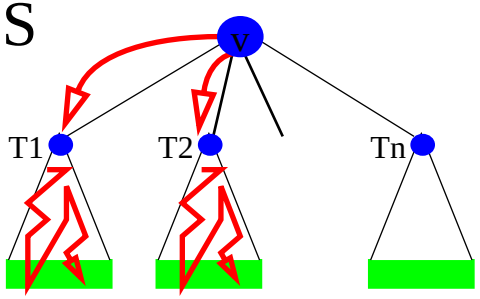
Du vil på denne måten kunne utforske hele labirynten og returnere til inngangen

Chorus :

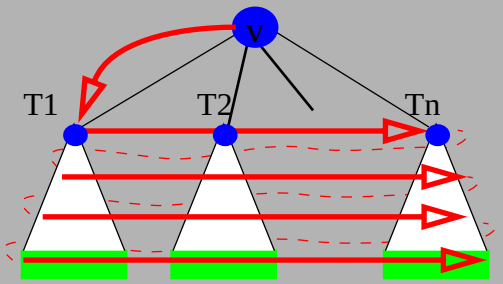
Smart Girl!

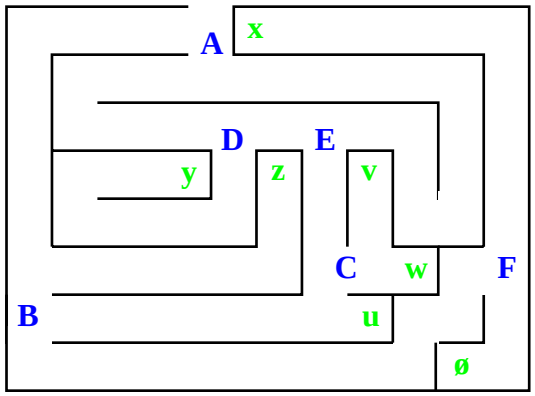
III. Graf traversering

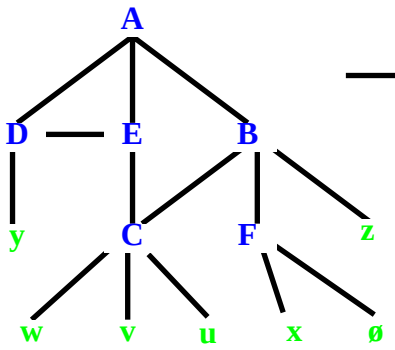
a) DFS



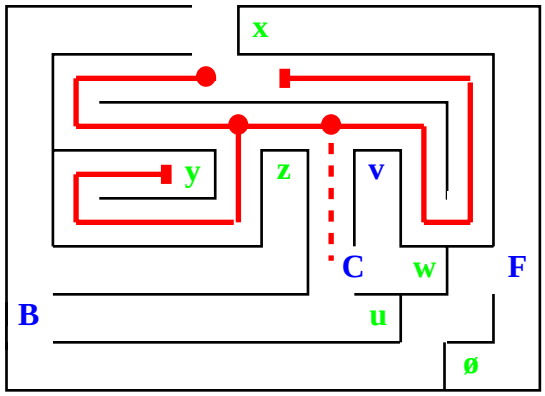
b) BFS

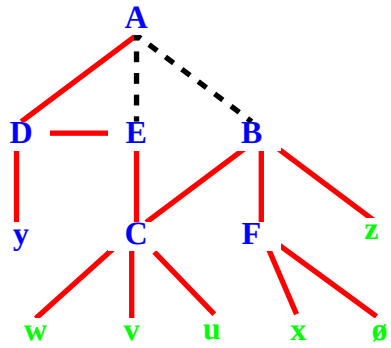







```
DFS(A)
  merk A visited
  for hver kant e = (A, X)
    if ( ! merket X )  // ! e er rød
        // merk–e–rød
        DFS(X)
    // else merk–e–svart
```





Kanter merket med rød
under DFS traversering gir et
utspennende tre for grafen
– roten til treet kan velges
vilkårlig !

III.a) DFS traversering av grafer

```
DFS(s)
  merk s visited
  for hver kant e = (s,u)
      if ( ! merket–u )
          // merk–e–rød
          DFS(u)
```

12.12 DFS traversering av en ikke-rettet graf G fra en node s:

a) besøker *alle* noder i en *sammenhengende komponent* til s

b) merkete kanter gir et *utspennende tre*, DFS tre, for denne komponenten til s

Begrunnelse :

a) Anta, kontrapositivt, at det finnes en ubesøkt node v.

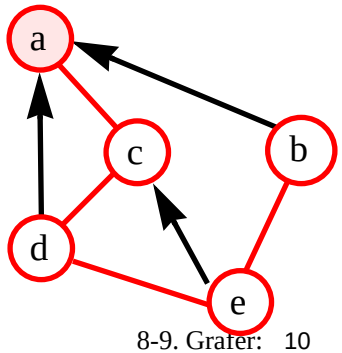
Siden komponenten er sammenhengende, finnes det en sti fra s til enhver node,

så anta at v er den første ubesøkte noden på en sti fra s:

- Siden v er den første slike, finnes det en nabo node u ("like før") som ble besøkt
- Men da – mens vi besøkte u – måtte vi også ha sett på kanten (u,v) og, siden v ikke var merket, måtte den ha blitt besøkt.

b) Vi merker kanter (u,v) kun når vi går til endenoden v for første gang

- Derfor danner vi aldri en sykel – vi fåren asyklisk graf, dvs. et tre
- Treet er utspennende fordi alle komponentens noder er med (a)



III.a) DFS traversering av grafer

```
DFS(s)
  merk s visited
  for hver kant e = (s,u)
      if ( ! merket–u )
          // merk–e–rød
          DFS(u)
```

Kjøretid av DFS:

DFS kalles 1 gang for hver node og ser hver kant 2 ganger :
$O(n_s+k_s)$ hvis

- gitt en kant, kan man aksessere dens ende-noder i $O(1)$
- merking av noder/kanter og sjekking om de er merket tar $O(1)$
- for hver node v, kan alle dens kanter akssesere 1 gang i $O(k(v))$

DFS kan brukes for å lage $O(n+k)$ algoritmer for å :

- sjekke om G er sammenhengende og finne sammenhengende komponenter
- finne utspennende tre for G
- sjekke om det finnes en sti mellom to noder;
- se om G inneholder sykler

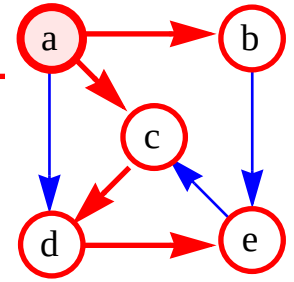
III.a) DFS på en rettet graf

12.16 (12.12) DFS traversering av en rettet graf G fra en node a:

a) besøker alle noder *oppnåelige* fra a

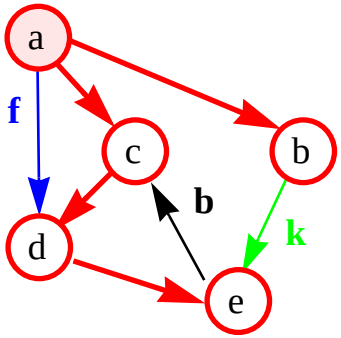
b) gir et utspennende tre, DFS-treet, for delgrafen oppnåelig fra a

```
DFS(u)                    // opptil n rekursive kall
  merk–u
  for hver kant e ε outIncidentEdges(u)
      v = opposite(u,e)
      if ( ! merket(v) )
          // merk e rød
          ..... DFS(v)
```

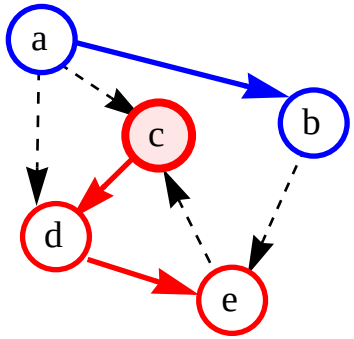


– Traverserte kanter som ikke er med i DFS-treet kan deles i tre grupper:

- **fram**-kanter fra v til en etterfølger node i DFS-treet
- **tilbake**-kanter fra v til en forgjenger node i DFS-treet
- **kryss**-kanter fra v til en urelatert node i DFS-treet



– Iterert DFS: 'for hver node v utfør DFS(v)' kan gi en skog:



– BFS for rettede grafer har tilsvarende egenskaper til BFS for ikke-rettede grafer (etterlater kun tilbake- og kryss-kanter)

kompleksitet	kant-liste	nabo-liste	nabo-matrise
outIncidentEdges(v)	O(**m**)	O(**deg**)	O(**n**)
DFS	**O(n * m)**	**O(n + m)**	**O(n * n)**
siden m=O(n*n) får vi	O($\mathbf{n^{3)}}$	**O(n^2)**	**O(n^2)**

DFS på rettet graf gir opphav til ***O*(n+k)** algoritme for å :

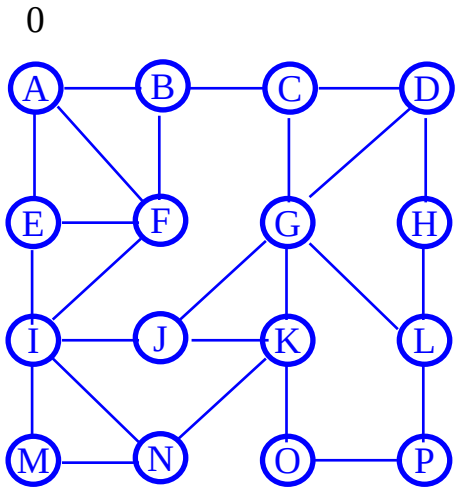
- finne alle noder oppnåelig fra en gitt node

Iterert DFS gir ***O*(n(n+k))** algoritmer for å :

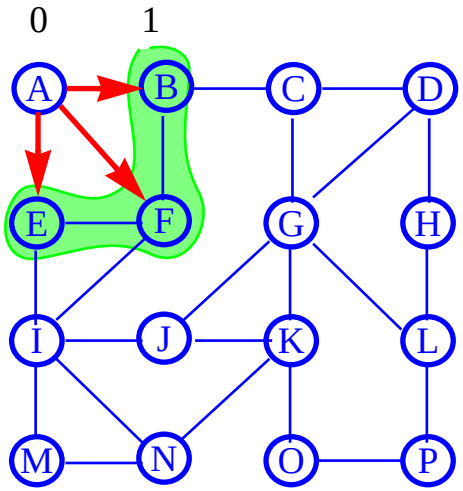
- avgjøre om G er sterkt sammenhengende; (mulig også i O(n+m))
- lage transitiv tilluking G* av G

III.b) BFS traversering av grafer

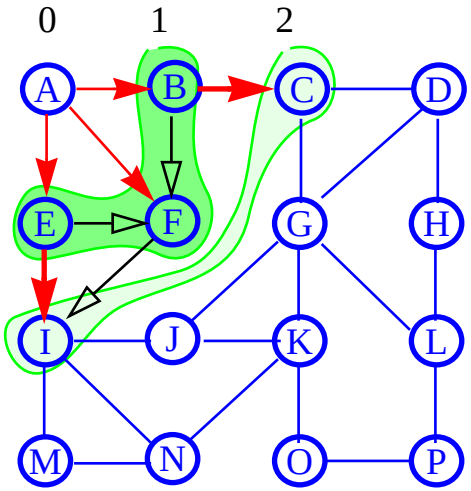
Hva er lengden av korteste vei fra A *til* P *?*



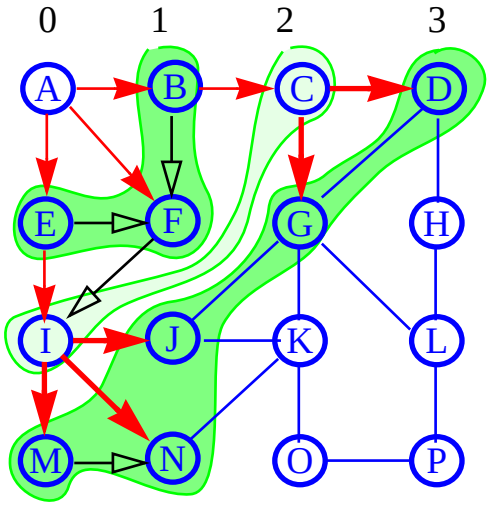
b)



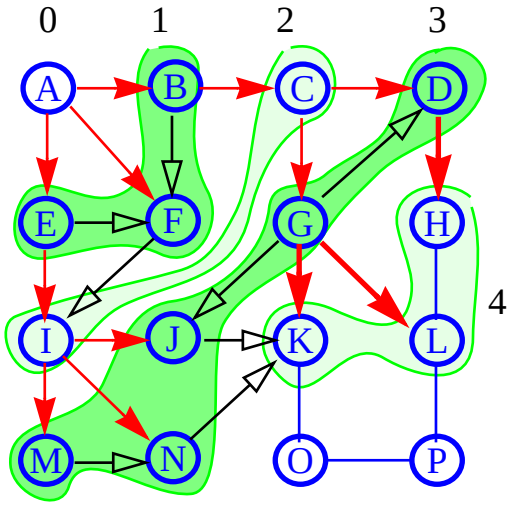
c)



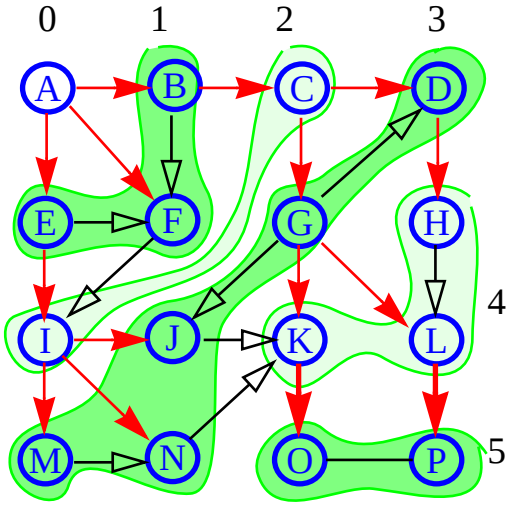
d)



e)



f)



III.b) BFS graf traversering

Tree DFS

```
/*DFS(Tree T,Position s)
  *   Stack S = new StackIM()
  *   S.push(s)
  *   while (!S.isEmpty())
  *      p= S.pop()
  *      for each p's child c
  *           S.push(c)
  */
```

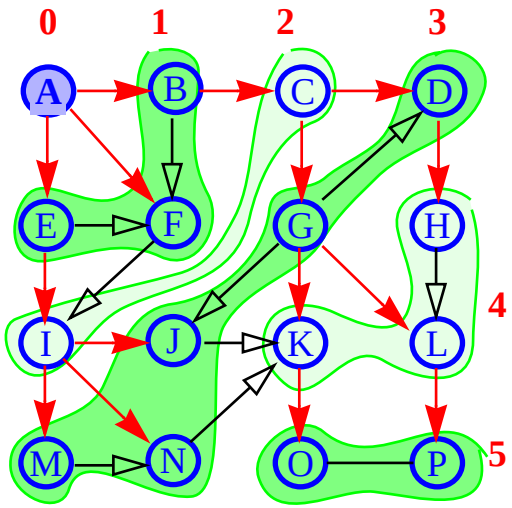
Tree BFS

```
/*BFS(Tree T,Position s)
  *   Queue S = new QueueIM()
  *   S.enqueue(s)
  *   while (!S.isEmpty())
  *      p= S.dequeue()
  *      for each p's child c
  *           S.enqueue(c)
  */
```

Graph BFS

```
// initielt er alle noder umerket

BFS(Graph G, Position s)
    Queue S = new QueueIM()
    merk(v, 0) – merker med nivå
    S.enqueue(s)
    while (!S.isEmpty())
      k= S.dequeue()
      for each kant e=(k,m)
        if (! merket(m))
            merk(m, k.merke+1)
            merk(e, rød)
            S.enqueue(m)
```



12.14. BFS traversering av en ikke-rettet graf G fra en node s:

a) besøker alle noder i en sammenhengende komponent til s

b) røde kanter danner et utspennende tre, så kalt BFS tre, for G

c) korteste stien fra **s** til hver node på nivå ***i*** har ***i*** kanter og enhver annen sti har minst ***i*** kanter

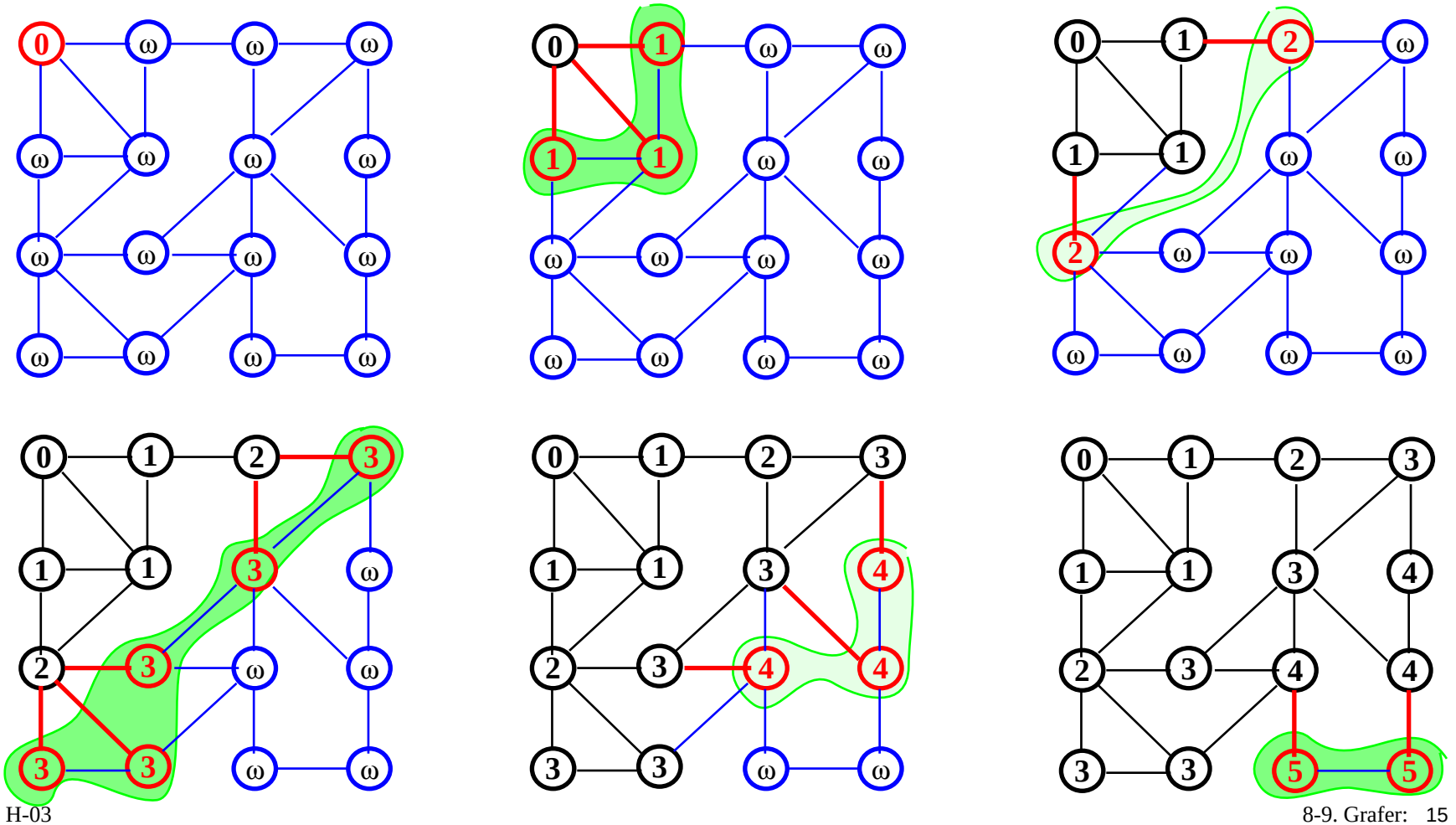
d) er ikke kant **(u,v)** med i BFS tre, så er nivå forskjellen mellom **u** og **v** høyst ***1***

BFS gir opphav til $O(n+k)$ algoritmer for å

- sjekke om G er sammenhengende
- finne sammenhengende komponenter i G
- finne utspennende tre for G
- beregne minimalt antall kanter mellom to noder (korteste sti)

III.b) BFS: Ford-Bellman

```
for each node v : D[v]= ω      // ω er et maksimalt tall (her ω > n)
D[s] = 0;                      // s er startnoden
for (i=1; i<n; i++)            // n er antall noder i grafen G                    O(n*k)
   for each kant (k,m)         // for en ikke-rettet G : (k,m) ε G hviss (m,k) ε G
      if ( D[k] + 1 < D[m])    D[m] = D[k] + 1
```



IV. Graf ADT *(ikke-rettet utsnitt)*

package jdsl.graph.api;

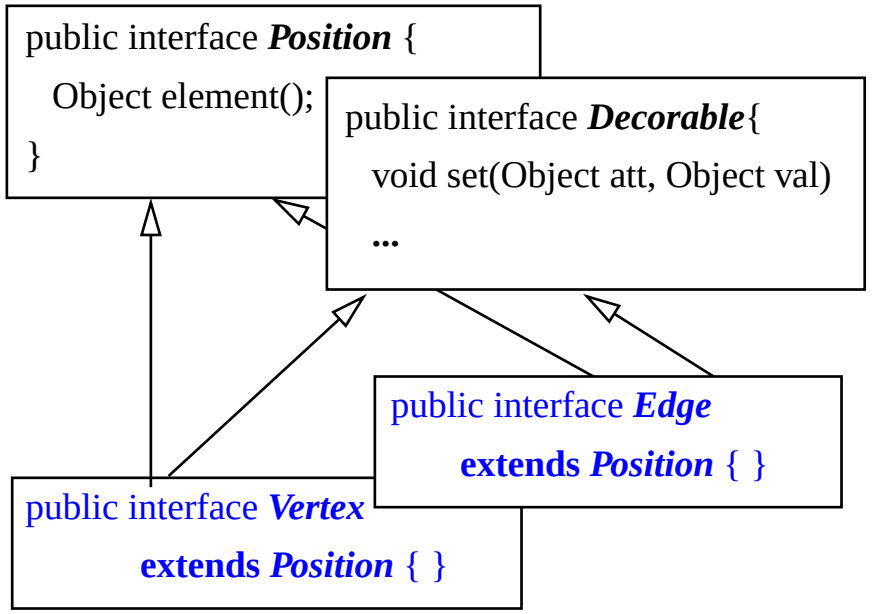
```
public interface InspectableGraph extends PositionalContainer {

  int numVertices();  int numEdges();
  VertexIterator vertices();
  EdgeIterator edges();
/** # rettede og ikke-rettede nabokanter */
  int degree(Vertex v);
  Vertex[] endVertices(Edge e)
  Vertex opposite(Vertex v, Edge e);
/** nabonoder langs alle kanter
    inn–, utgående samt ikke-rettede */
  VertexIterator adjacentVertices(Vertex v)
/** rettede og ikke-rettede nabokanter */
  EdgeIterator incidentEdges(Vertex v)
}
```

```
public interface Graph extends ModifiableGraph {
   Vertex insertVertex(Object o);
   Edge insertEdge(Vertex u, Vertex v, Object o);

   Object removeEdge(Edge e) ;
/** fjern noden v og ALLE kanter med v */
   Object removeVertex(Vertex v) ; !!!

}
```



IV. Graf ADT

(*kan ha **både** rettede og ikke-rettede kanter*)

package jdsl.graph.api;

```
public interface InspectableGraph extends PositionalContainer {

  int numVertices();  int numEdges();
  VertexIterator vertices();
  EdgeIterator edges();
/** # rettede og ikke-rettede nabokanter */
  int degree(Vertex v);
  Vertex[] endVertices(Edge e)
  Vertex opposite(Vertex v, Edge e);
/** nabonoder langs alle kanter
    inn–, utgående samt ikke-rettede */
  VertexIterator adjacentVertices(Vertex v)
/** rettede og ikke-rettede nabokanter */
  EdgeIterator incidentEdges(Vertex v)
}
```

```
  EdgeIterator unDirectedEdges();
  EdgeIterator directedEdges();
  int outDegree(Vertex v);
  int inDegree(Vertex v);
  Vertex origin(Edge e)
  Vertex destination(Edge e)
  boolean isDirected(Edge e)
  VertedIterator outAdjacentVertices(Vertex v)
  VertedIterator inAdjacentVertices(Vertex v)
  EdgeIterator outIncidentEdges(Vertex v)
  EdgeIterator inIncidentEdges(Vertex v)
```

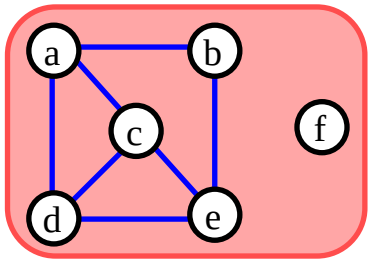
```
public interface Graph extends ModifiableGraph {
  Vertex insertVertex(Object o);
  Edge insertEdge(Vertex u, Vertex v, Object o);

  Object removeEdge(Edge e)
  /** fjern noden v og ALLE kanter med v */
  Object removeVertex(Vertex v)

}
```

```
  void makeUndirected(Edge e)
  Edge insertDirectedEdge( Vertex u,
                           Vertex v, Object o);
  void reverseDirection(Edge e)
  /** gir retning til en ikke-rettet kant */
  void setDirectionTo/From(Edge e, Vertex v)
```

1. Implementasjon av Graph *med Kant-Liste*



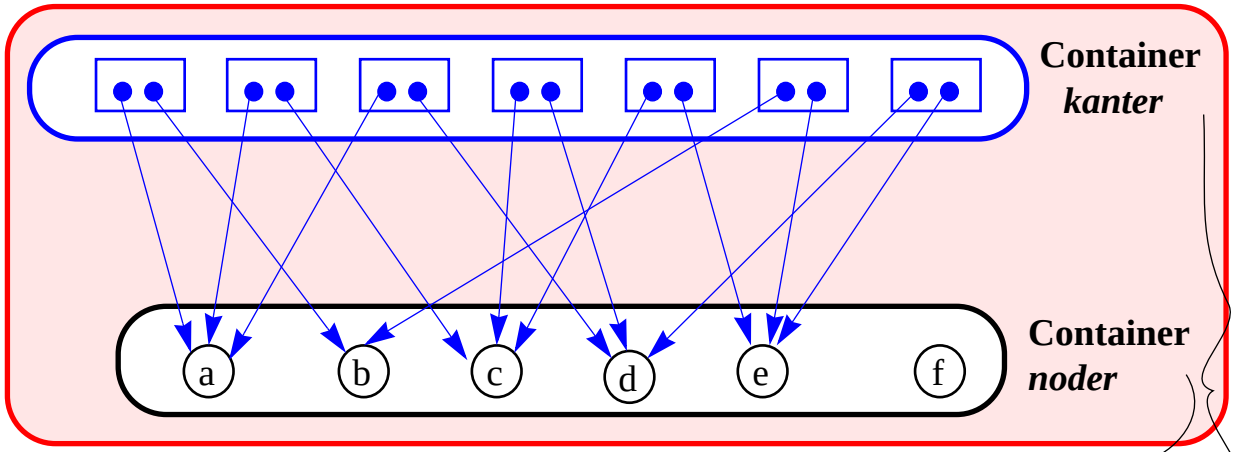
endV(e), opposite(v,e),
degree(v) ..... $O(1)$

insertV(o), insertE(v,u,o),
removeE(e) ..... $O(1)$

incidentE(v), adjacentV(v) .. $O(k)$

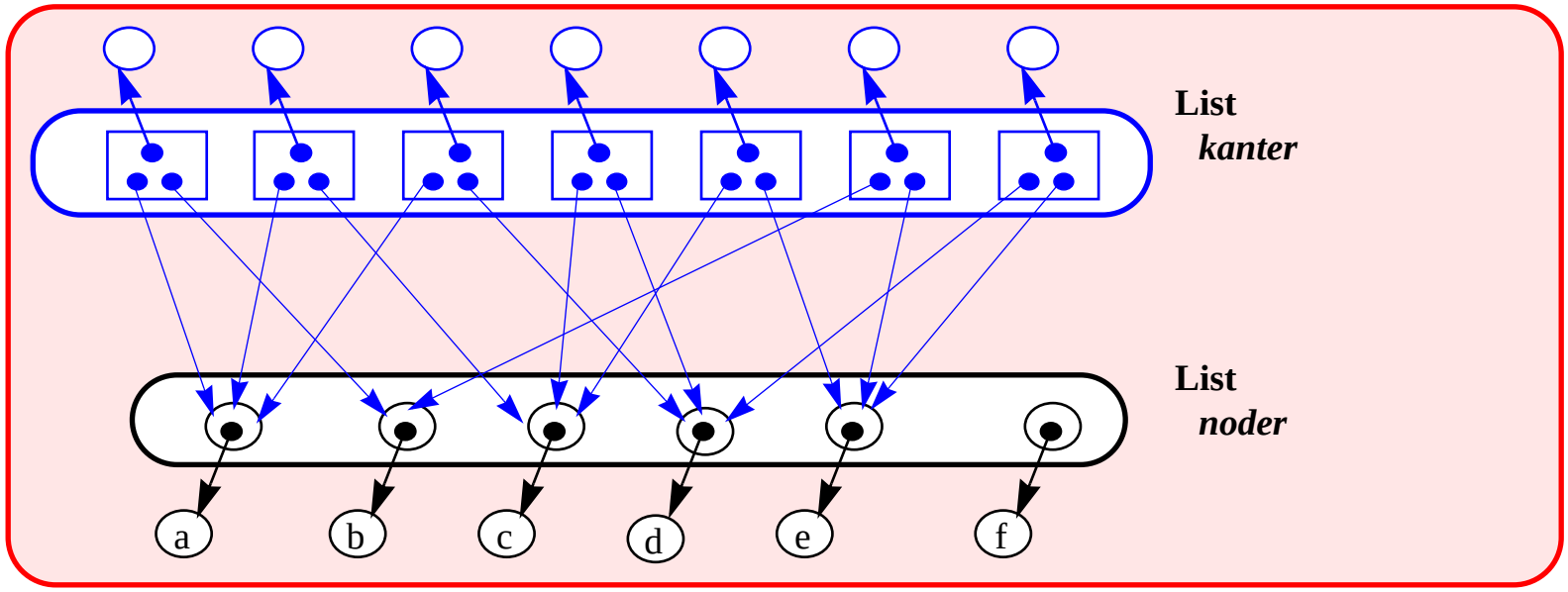
removeV(v) ..... $O(k)$

areAdjacent(v,u) ..... $O(k)$



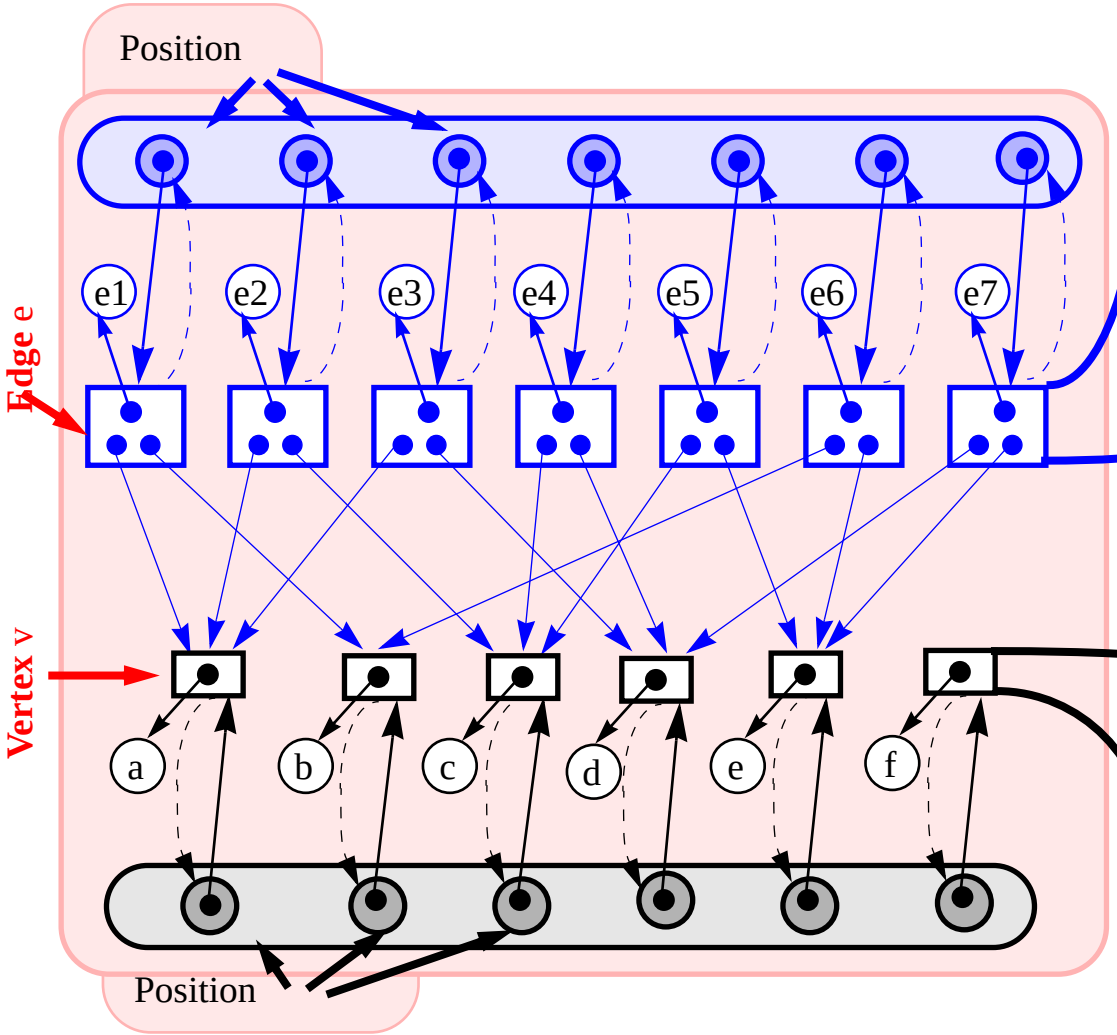
plass forbruk $O(k+n)$

Sequence
List
Dictionary



public class *GraphKL* implements *Graph* {

```
private List kanter, noder;
public GraphKL(List k, n)   { kanter = k; noder = n; }
```




```
public int numEdges() { kanter.size(); }
public int numVertices() { noder.size(); }
. . .
```

```
class KLEdge implements Edge {
  protected Object elem;
  protected Position kp;
  protected Vertex[] ee=new Vertex[2];
  public KLEdge (Vertex a,Vertex b,
                 Object o)  {
    elem=o; ee[0]=a; ee[1]=b; }
  public Vertex[] endV() { return ee; }
  public boolean has(Vertex v) {
    return (ee[0]==v || ee[1]==v) ; }
  public Position iKanter() { return kp; }
  public void setPos(Position p) { kp = p; }
  ... }
```

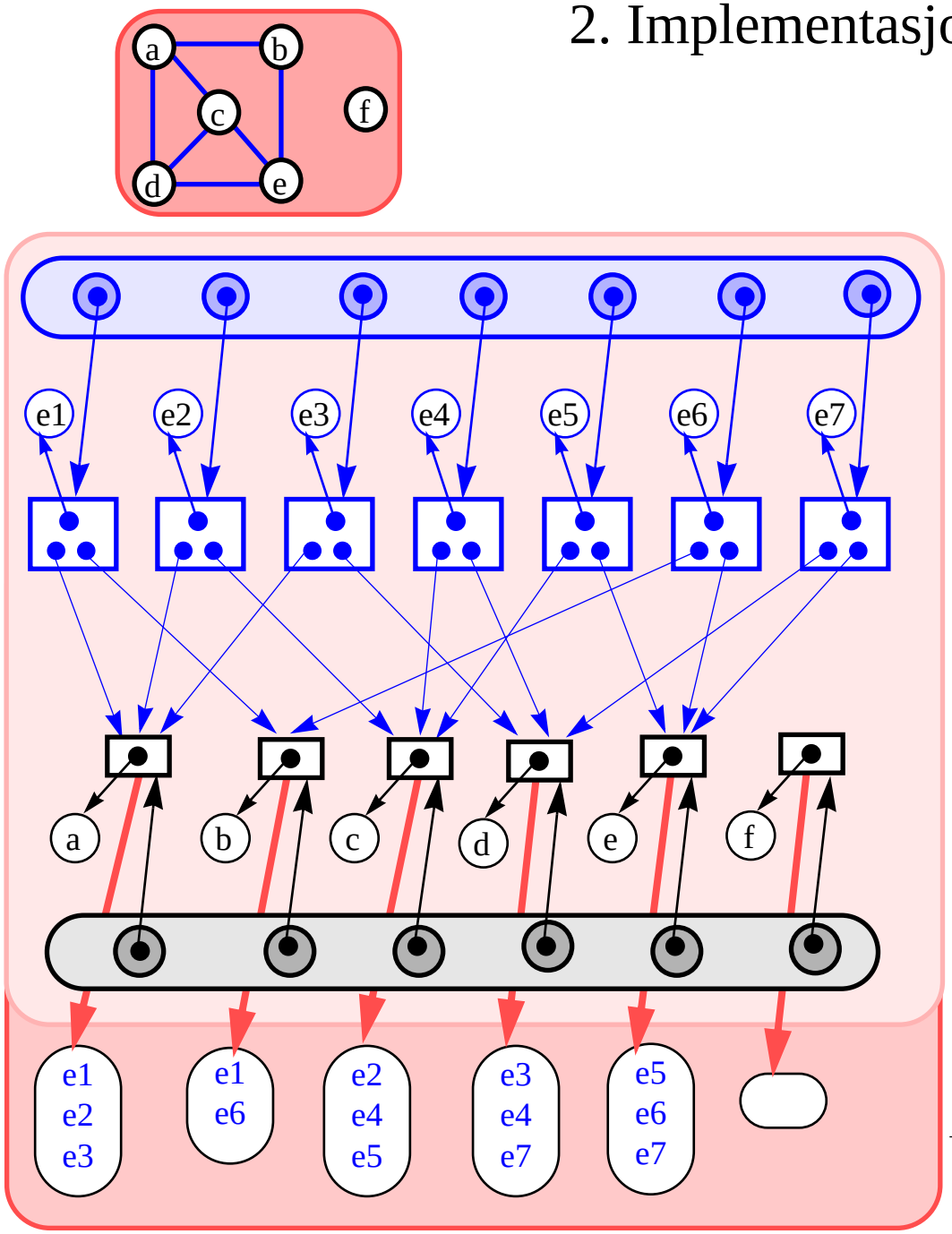
DataInvariant:

e . iKanter() . element() == e

n . iNoder() . element() == n

```
class KLNode implements Vertex {
  protected Object elem; int deg=0;
  protected Position np;
  public KLNode(Object o)  { elem=o;}
  public Position iNoder() {return np; }
  public void setPos(Position p) { np = p; }
  public int degree() { return deg; }
  public void inc() { deg = deg+1; }
  public void dec() { deg = deg–1; }
... }
```

2. Implementasjon av Graph *med Nabo-Liste*



er som Kant-Liste men i tillegg

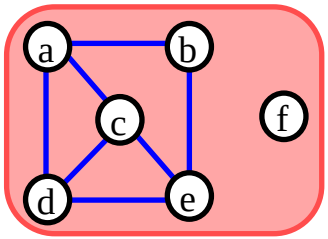
- hver node v har en samling Inc(v) med sine (incident) kanter
- (hver kant (u,v) holder en referanse til sin posisjon i Inc(v) og Inc(u))

endV(e), opposite(v,e), degree(v)	*O*(**1**)	*O*(**1**)
insertV(o), insertE(v,u,o), removeE(e)	*O*(**1**)	*O*(**1**)
incidentE(v), adjacentV(v)	*O*(**k**)	*O*(deg **v**)
removeV(v)	*O*(**k**)	*O*(deg **v**)
areAdjacent(v,u)	*O*(**k**)	*O*(deg **v**|**u**)

```
Object removeVertex(Vertex v) {
  if (ok(v)) {  Object ret= v.element();
    Iterator k= Inc( (NLNode) v ) ;
    while (k.hasMoreElements() ) {      O(deg v)
      NLNode e= (NLNode)k.nextElemet();
      if (e.has(v)) removeEdge(e);  }
    noder.remove( ((NLNode)v).iNoder() );
    return ret;
  } else throw new InvalidPosExc(“”);  }
```

- Inc(v) har ofte kun nabo-noder

3. Implementasjon av Graph *med Nabo-Matrise*

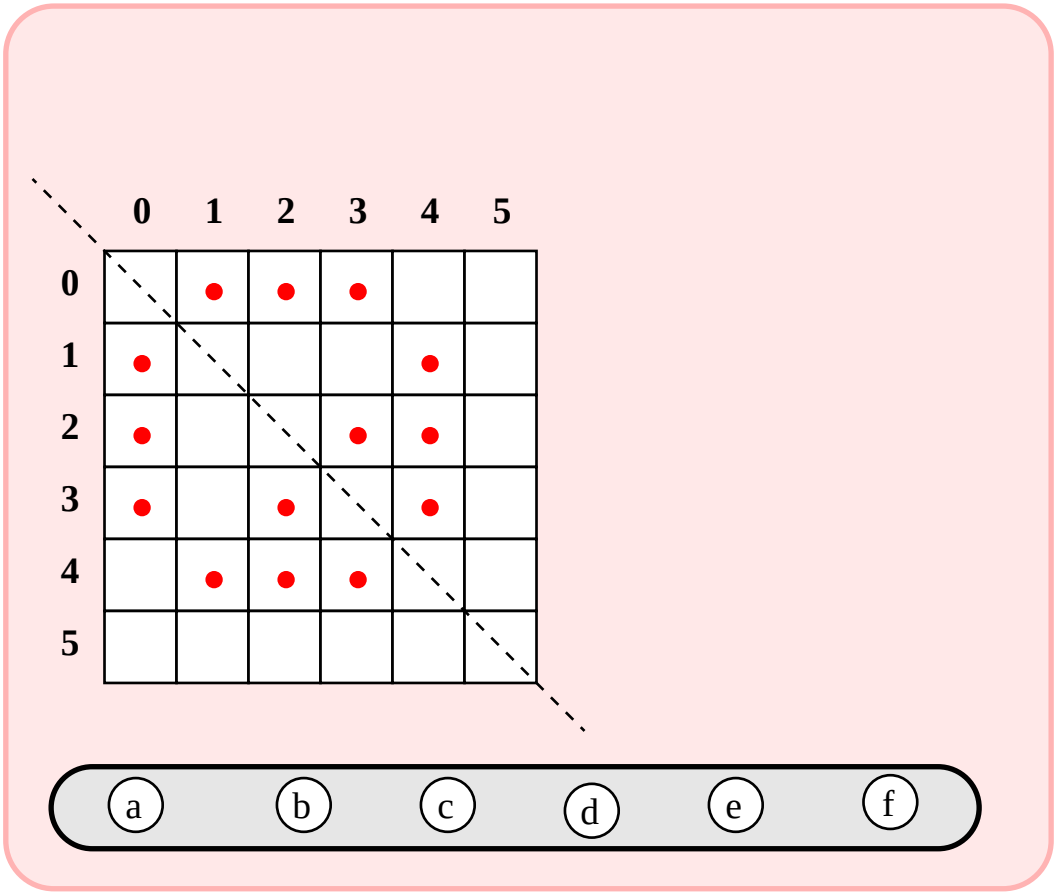


Enumerer alle noder fra 0...n-1:

a	b	c	d	e	f	
|	|	|	|	|	|	no
0	1	2	3	4	5	

i en **n** x**n** matrise **boolean A[n][n]**

- **A[i][j]** == **true** hviss G har en kant e=(i,j)
- **A[i][j]** == **false** hviss (i,j) ikke er en kant i G



plass forbruk $O(\mathbf{n^2})$

endV(e), opposite(v,e), degree(v)	$O(\mathbf{1})$	$O(\mathbf{1})$
insertE(v,u,o), removeE(e) ..	$O(\mathbf{1})$	$O(\mathbf{1})$
insertV(o)	$O(\mathbf{1})$	$O(\mathbf{n^2})$
incidentE(v), adjacentV(v)	$O(\deg \mathbf{v})$	$O(\mathbf{n})$
removeV(v)	$O(\deg \mathbf{v})$	$O(\mathbf{n^2})$
areAdjacent(v,u)	$O(\deg \mathbf{v}\|\mathbf{u})$	$O(\mathbf{1})$

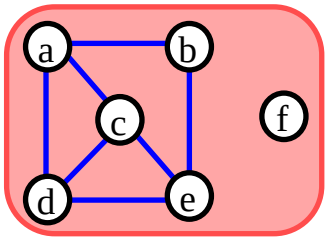
```
boolean areAdjacent(Vertex v, Vertex u)
  { if ( ok(v) && ok(u) )
    return A [no(v)] [no(u)]  ; }
```

```
void insertEdge(Vertex v,u) {
  if ( ok(v) && ok(u) )
    A [no(v)] [no(u)]  = true ;
  } else if ( areAdjacent(v,u) )  throw ...
  else throw new InvalidPosExc(“”);  }
```

DataInvariant for ikke-rettet graf:

A [**i**] [**k**] == **A** [**k**] [**i**]

3. Implementasjon av Graph *med Nabo-Matrise*

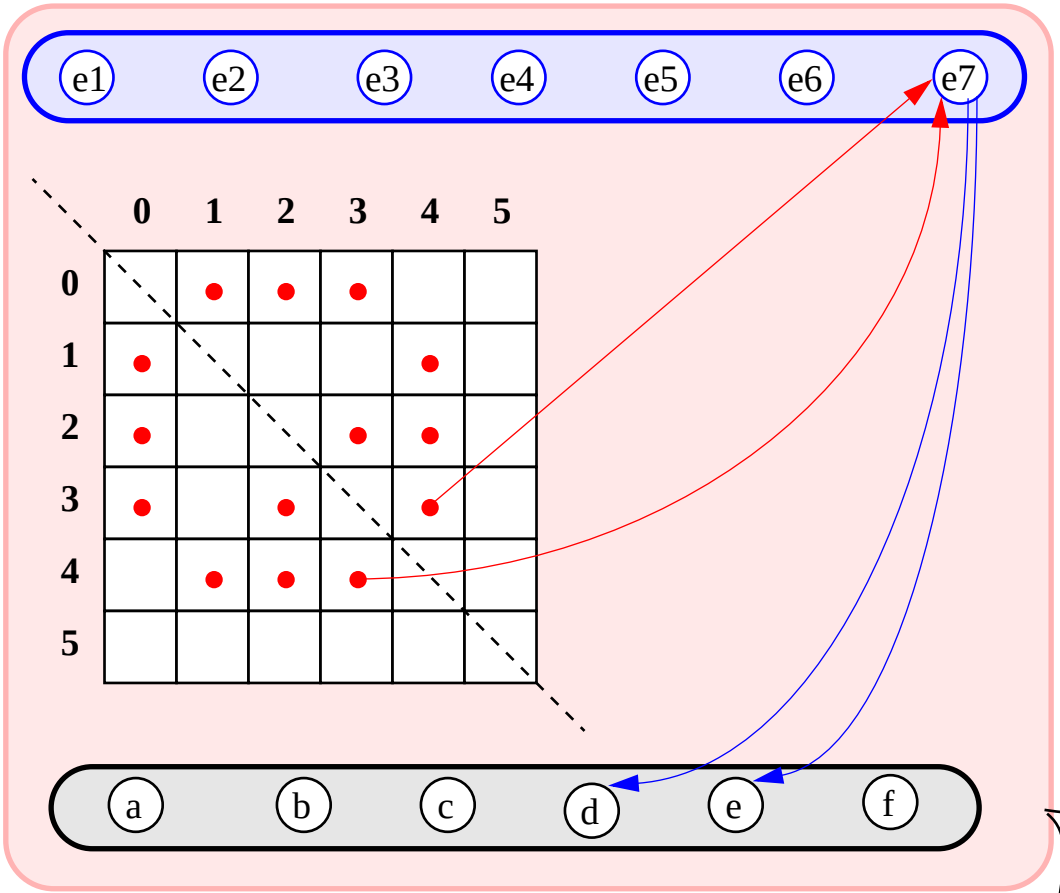


Enumerer alle noder fra 0...n-1:

a	b	c	d	e	f	
|	|	|	|	|	|	no
0	**1**	**2**	**3**	**4**	**5**	

i en **n** x**n** matrise **A**

- **A[i][j]** == **e** (**true**) hviss G har en kant e=(i,j)
- **A[i][j]** == **null** (**false**) hviss (i,j) ikke er en kant i G



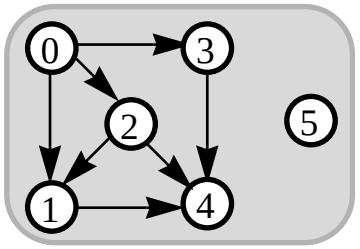
plass forbruk $O(\mathbf{n^2})$

endV(e), opposite(v,e), degree(v)	$O(\mathbf{1})$	$O(\mathbf{1})$
insertE(v,u,o), removeE(e) ..	$O(\mathbf{1})$	$O(\mathbf{1})$
insertV(o)	$O(\mathbf{1})$	$O(\mathbf{n^2})$
incidentE(v), adjacentV(v)	$O(\deg \mathbf{v})$	$O(\mathbf{n})$
removeV(v)	$O(\deg \mathbf{v})$	$O(\mathbf{n^2})$
areAdjacent(v,u)	$O(\deg \mathbf{v}\|\mathbf{u})$	$O(\mathbf{1})$

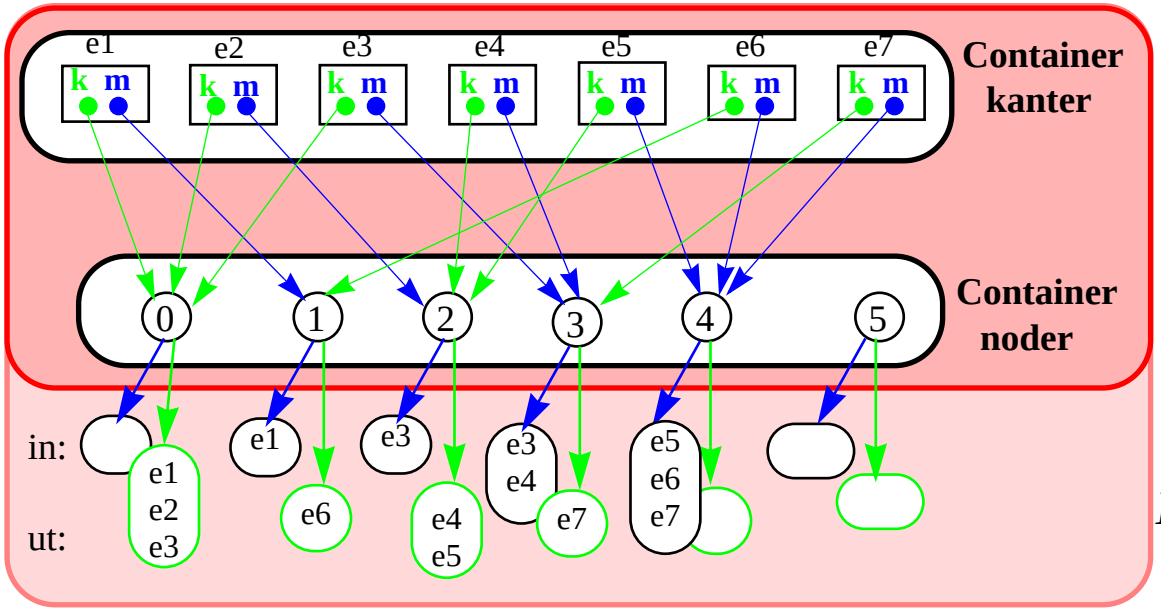
```
Edge insertEdge(Vertex v,u, Object o) {
  if (ok(v) && ok(u) && !areAdjacent(v,u))
  { NMEdge e = new NMEdge(v,u,o,this);
    e . setPos ( kanter . insertLast (e) );
    ((KLNode) v).inc(); ((KLNode) u).inc();
    A[no(v)] [no(u)]  = e ;
    return e;
  } else if (areAdjacent(v,u))  throw ...
  else throw new InvalidPosExc(“”);  }
```

Utmerket for **stabile**, nesten **komplette** grafer (ikke for traversering)

4. Implementasjon av rettet Graph



Kant-Liste

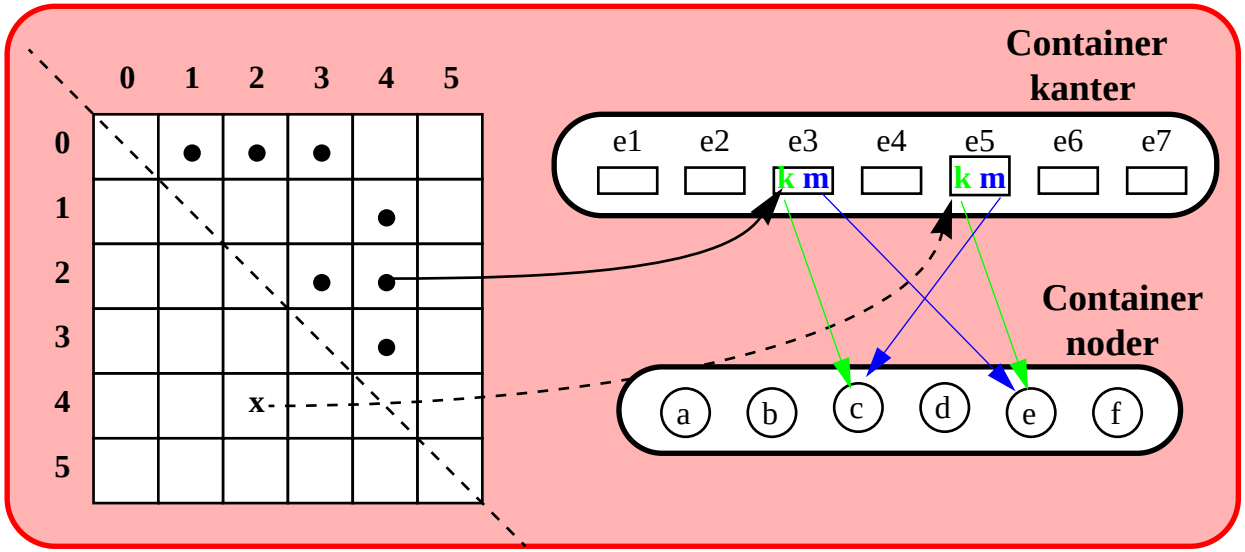


urettet implementasjon trenger ikke

- *å skille mellom* **Vertex origin** *og* **Vertex destination** *i Kant-klassen, eller*
- *mellom* **in- og ut-** *samlinger i Node-klassen*

Nabo-Liste

Nabo-Matrise

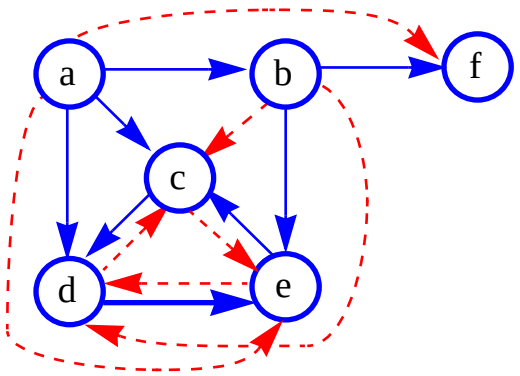


Implementasjoner av Graph

n : antall noder **k** : antall kanter

kompleksitet operasjon		**Kant-Liste**	**Nabo-Liste**	**Nabo-Matrise**
	numVertices(), numEdges()	1	1	1
vertices() / edges()	un/directedEdges()	n / k	n / k	n / k
degree(v)	in/outDegree(v)	1	1	1
endVertices(e), opposite(v,e)	origin(e), destination	1	1	1
adjacentVertices(v)	in/outAdjacentVertices(v)	k	deg v	n
incidentEdges(v)	in/outIncidentEdges(v)	k	deg v	n
insertVertex(o)		1	1	n^2
removeVertex(v)		k	deg v	n^2
insertEdge(v,u,o)	inserDirectedEdge(v,u,o)	1	1	1
removeEdge(e)		1	1	1
reverseDirection(e), makeUndirected(e), ...		1	1	1
areAdjacent(v,u)		k	min(deg u,v)	1

V.a) Transitiv tillukning



IterertDFS(Graph G) $O(n * DFS)$
for hver node v ε V
DFS(v) – utvid G med kant (v,u)
for hver u i DFS-tre til v

node	kant til	lagt til
a	b c d	e f
b	e f	c d
c	d	e
d	e	c
e	c	d
f		

Kant-Liste $O(n^2 * m)$
Nabo-Liste $O(n^2 + nm)$
Nabo-Matrise $O(n^3)$

FloydWarshall(Graph G) **$O(n^3)$ med nabomatrise**
enummerer V : $v_1,v_2,...,v_n$ (vilkårlig)
$G_0 = G$
for k = 1,2,...,n
$G_k = G_{k-1}$
for hvert tallpar a ≠ b, a,b ≠ k
if G_{k-1}.**areAdjacent**(v_a,v_k) og G_{k-1}.**areAdjacent**(v_k,v_b)
legg kant (v_a,v_b) til G_k

i en rettet graf:
G_{k-1} har kant (v_a,v_i) og (v_i,v_b)

1 2 3 4 5 6
a b c d e f

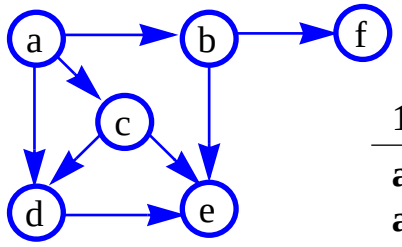
$G_a = G_0 = \{ ab, ac, ad, cd, de, ec, bf \}$
$G_b = G_a \cup \{ ae, af \}$
$G_c = G_b \cup \{ ed \}$ (ad)
$G_d = G_c \cup \{ ce \}$
$G_e = G_d \cup \{ bc, bd, dc \}$ (ac)
$G_f = G_e$

Kant-Liste $O(n^3 * m)$
Nabo-Liste $O(n^3 * deg)$
Nabo-Matrise $O(n^3)$

V.b) DAG-Directed Acyclic Graph

rettet asyklisk graf

- *arv i et OO-språk*
- *forkrav til kurs*
- *planlegging av avhengige aktiviteter*

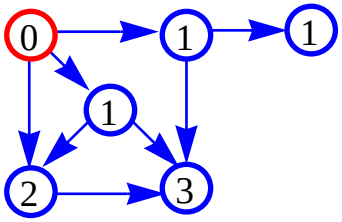


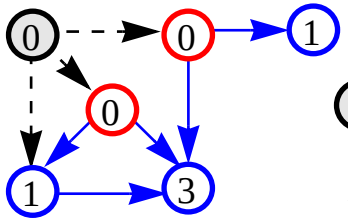
1	2	3	4	5	6
a	**b**	**f**	**c**	**d**	**e**
a	**c**	**d**	**b**	**e**	**f**
....					
a	d	c	e	b	f

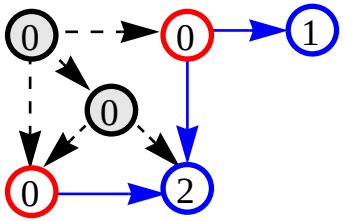
Topologisk ordning av en graf G er en enummerering av noder $v_1,v_2...v_n$ slik at hvis (v_i,v_j) ε E så er i < j. (dermed, hvis det finnes en sti v_i ... v_j så er i < j)

12.21. En rettet graf kan sorteres topologisk hvis og bare hvis den er asyklisk.

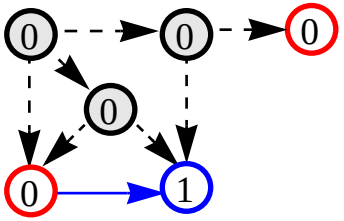
```
Queue TS(Graph G)                          O(nk)
  Q, R = empty Queue                       O(n+k)
  for hver node v ε V                      O(n²)
  {  in(v) = G.inDegree(v)
     if (in(v) == 0) Q.enqueue(v)  }
  while (! Q.isEmpty() )
  {  h = Q.dequeue()
     for hver v ε G.outAdjacentVertices(h)
     {  in(v) = in(v) – 1
        if (in(v)==0) Q.enqueue(v)  }
     R.enqueue(h)  }
  return R
```

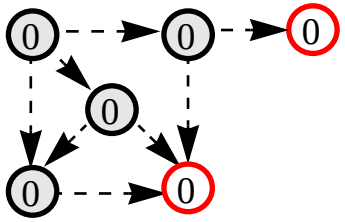






R	a	b	c	d	e	f	Q
	0	1	1	2	3	1	a
a	x	**0**	**0**	**1**	3	1	c b
a c	x	0	x	**0**	**2**	1	b d
a c b	x	x	x	0	**1**	**0**	d f
a c b d	x	x	x	x	**0**	0	f e
a c b d f e							





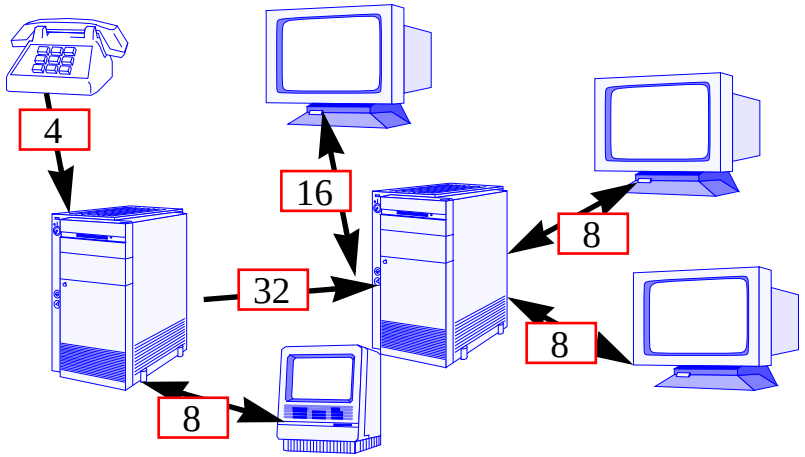
12.22 Har grafen en rettet sykel vil TS oppdage dette. Hvordan?.
-> TS gir en O(n+m) algoritme som sjekker om en rettet graf er asyklisk

VI. Vektede Grafer

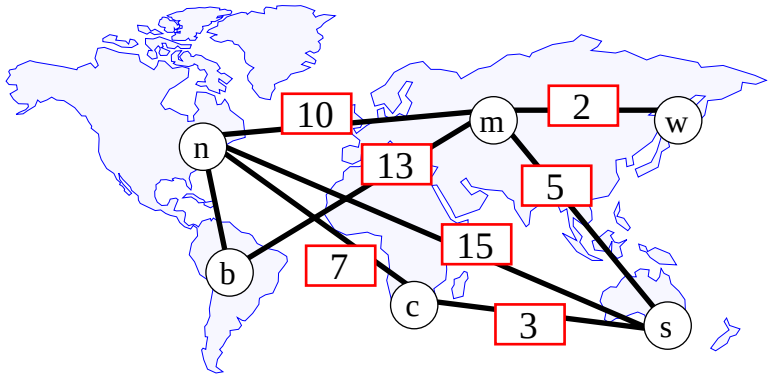
en graf der hver kant har et vekt-attributt

- vektene er vanligvis Totalt Ordnet (typisk heltall)
  - man designer en Comparator for sammenlikning av kanter mht. vekt
  - tilleggs antakelser om vekter (f.eks. > 0, 0, etc.)

NETTVERK KAPASITET



AVSTAND



- Implementasjon bruker det faktum at 'Edge implements Position' – kanter lagrer objekter med vekt

- I tillegg til vanlige graf-problemer, spør man i forbindelse med vektede grafer for eksempel om
  - *hva er **korteste** sti fra u til v, dvs sti fra u til v med minste sum av vekter?*
  - *hva er **minste** utspennende tre, dvs hvor sum av vekter på kanter i treet er minst?*
  - *..... minste / korteste / billigste ..... ?*

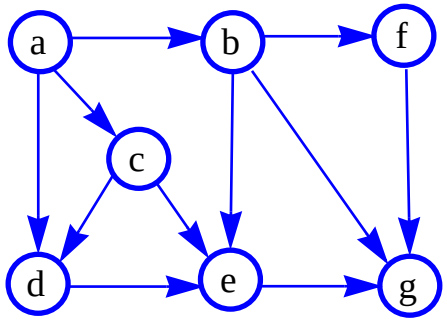
Korteste sti : ikke-vektet graf

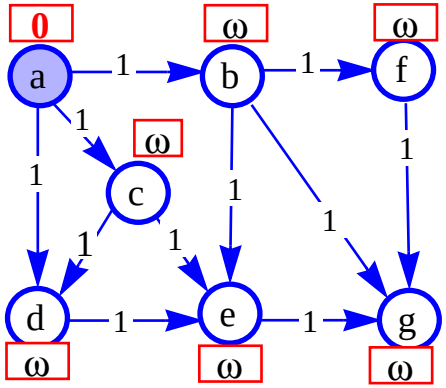
...BFS

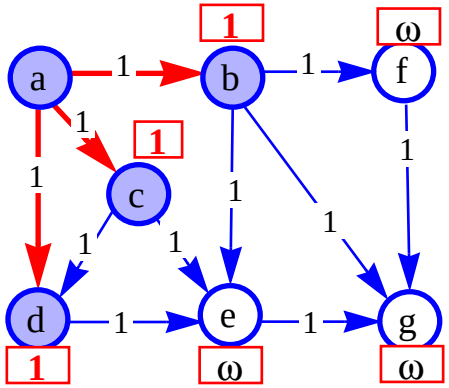
Finn sti fra a til enhver annen node, som har minst antall kanter (alternativt: kantvekter=1)

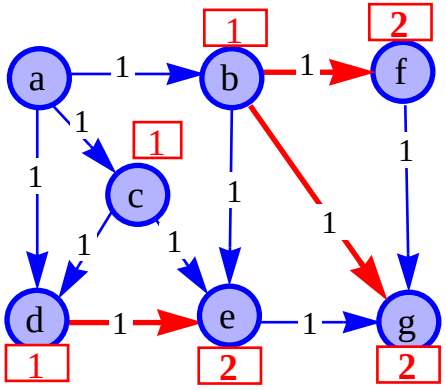
Ford-Bellman **BFS** :

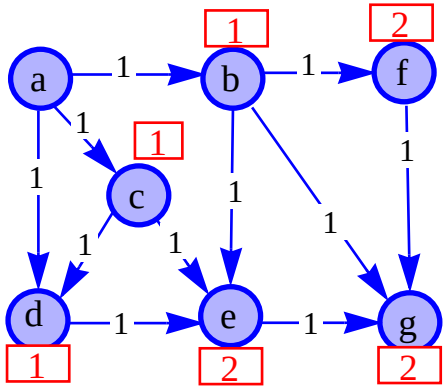
```
for each node v : D(v)= ω        // ω er et maksimalt tall (her ω > n)
D(s) = 0;                        // s er startnoden
for (i=1; i<n; i++)              // n er antall noder i grafen G      O(n*m)
    for each kant (u,v)
        if ( D(u) + 1 < D(v) )   D(v) = D(u) + 1
```









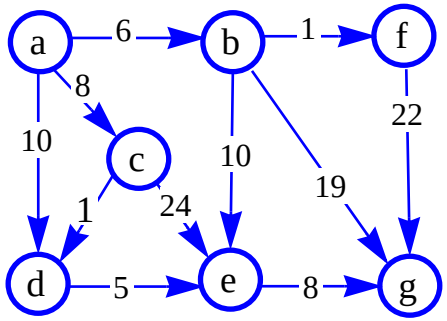


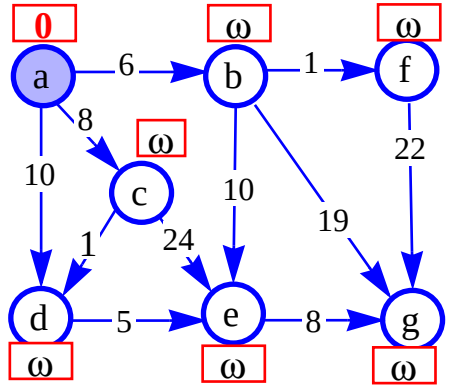
Korteste sti (single-source shortest-paths) :

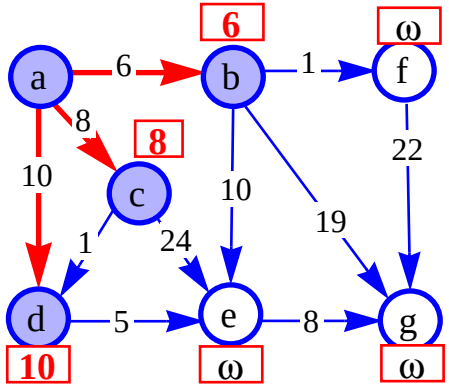
Finn (lengden av) korteste sti fra a til en bestemt/alle andre node(r)

Ford-Bellman **SS-SP** :

```
for each node v : D(v)= ω        // ω er et maksimalt tall (her ω > n)
D(s) = 0;                        // s er startnoden
for (i=1; i<n; i++)              // n antall noder, m antall kanter      O(n*m)
    for each kant (u,v)
        if ( D(u) + vekt(u,v) < D(v) )   D(v) = D(u) + vekt(u,v)
```



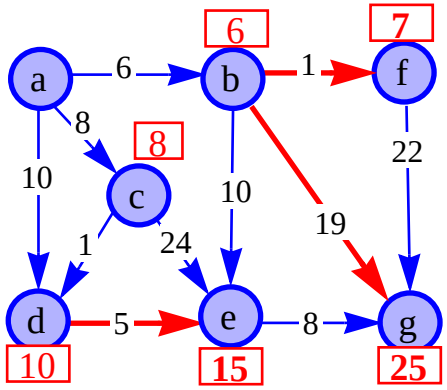


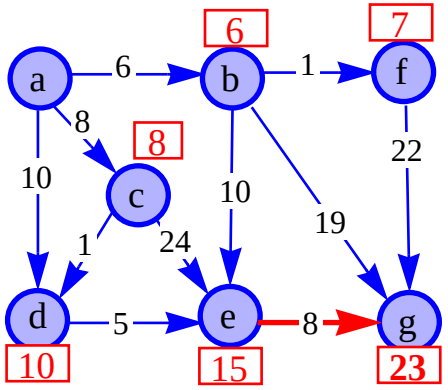


graf	vekter
ikke-rettet	1
rettet	vilkårlige

Etter Ford-Bellman SS-SP(G,a):

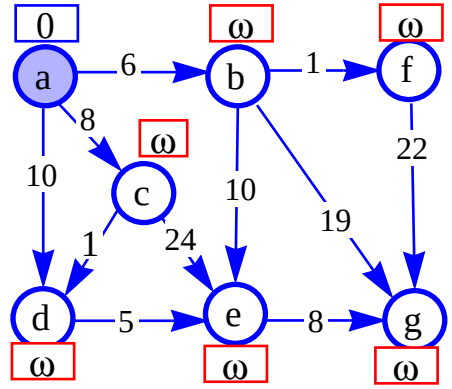
- hvis det finnes en kant (u,v) med
  D(u) + vekt(u,v) < D(v),
  så har G en negativ sykel
- ellers er D(v) korrekt for alle noder v

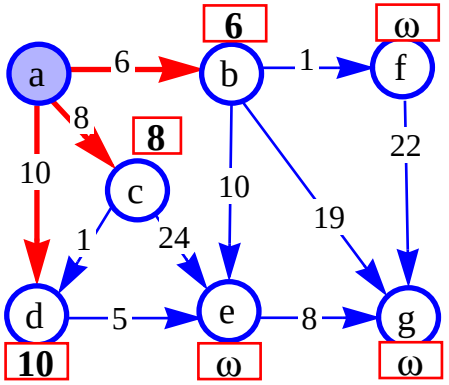


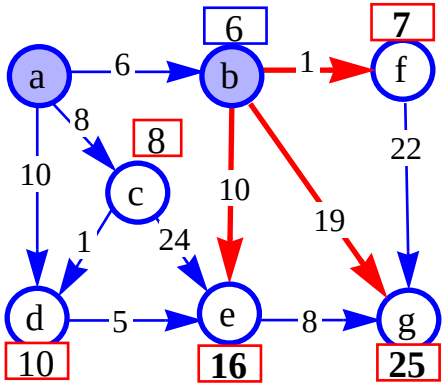


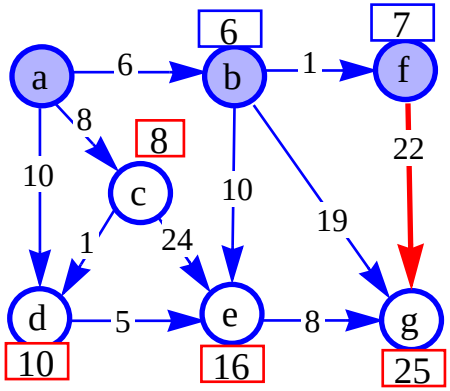
Korteste sti (SS-SP) : Dijkstra algoritme (vekt(e) ≥ 0)

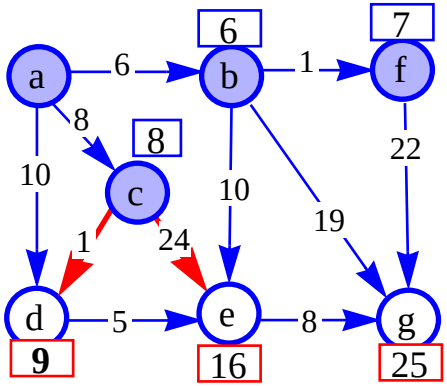
initialiser D(a)=0 og D(v)=ω for alle andre v
sett alle noder i en PriorityQueue Q mht. D
while (! Q.isEmpty())
  v = Q.removeMinElement() // Greedy
  for hver z ε G.outAdjacentVertices(v)
    if (D(v)+vekt(v,z) < D(z))
      D(z)= D(v) + vekt(v,z)
      oppdater Q // z kan få ny nøkkel

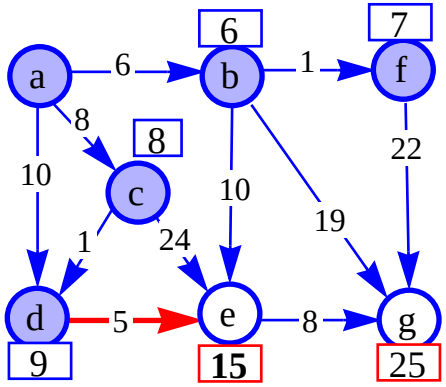


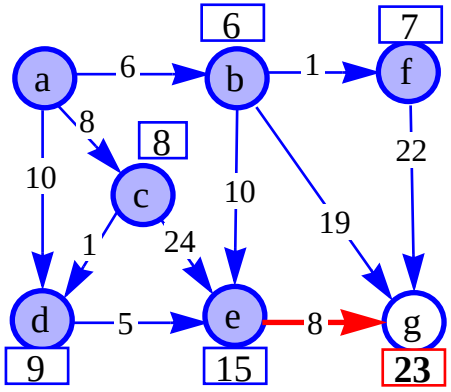


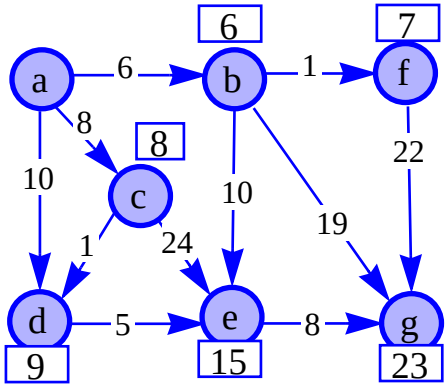








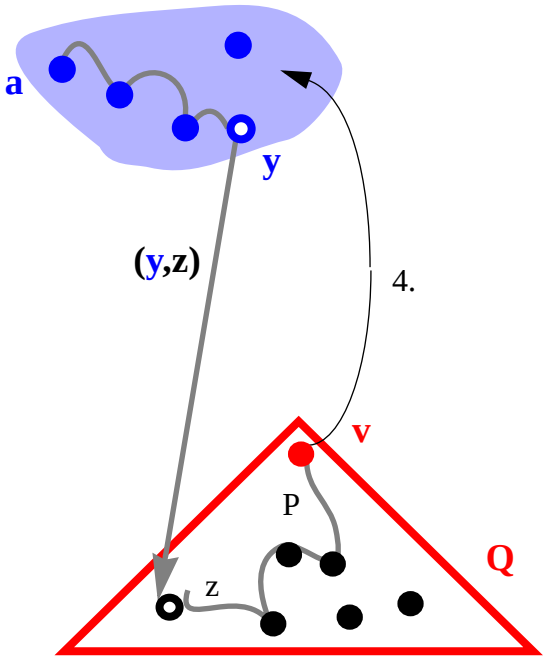




Dijkstra's SS-SP

1. initialiser D(a)=0 og D(v)=ω for alle andre v
2. sett alle noder i en PriorityQueue Q mht. D
3. while (! Q.isEmpty()) // n
4. v = Q.**removeMinElement**()
5. for hver z ε G.**outAdjacentVertices**(v) // k: Nabo-Liste
6. if (D(v)+vekt(v,z) < D(z))
7. Q.**replaceKey**(z, D(v) + vekt(v,z))

PriorityQueue	skal bruke Locator !!!	
heap	**O**((n+m) log n)	**O**(n^2 log n)
usortert sekvens	**O**(n*n+m)	**O**(n^2)
sortert sekvens	**O**(n +m*n)	**O**(n^3)



Løkke Invariant : alle noder x som har blitt fjernet fra Q har korrekt D(x)

– holder før inngangen siden ingen node ble fjernet.

– for å vise at den opprettholdes i løkken, må vise at den holder etter 4.

12.23 Ved 4. er D(v) = d(a,v) – lengden av korteste sti fra a til v.

a) Anta ikke og la **v** være **den første** node for hvilken **D(v)** > **d(a,v)** ved 4.

b) Dvs. **korteste sti P** a–v er kortere enn **D(v)**

c) La **z** være første noden på P som fortsatt er i Q (**d(a,v)** = **d(a,z)**+**d(z,v)**)

d) og la **y** være **z**'s umiddelbar forgjenger på P med en P-kant = **(y,z)**

a) → **e)** **D(y) = d(a,y)**

4. → **f)** **D(v) ≤ D(z)**

d) → **g)** **D(z) ≤ D(y) + vekt(y,z) = d(a,y) + vekt(y,z)**

siden (y,z) er med i korteste sti P a—v, finns det ikke en kortere sti a—z enn

h) **d(a,z) = D(y) + vekt(y,z) = D(z)**

Men da:

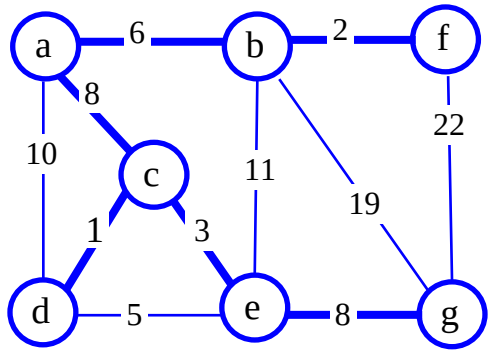
D(v) ≤ [f] D(z) =[h] d(a,z) ≤ d(a,z) + d(z,v) =[c] d(a,v) – motsier a) D(v) > d(a,v)

MST (Minimal Spanning Tree)

Gitt avstander mellom forskjellige byer, strek ledningene slik at

1. hvert par av byer er koblet sammen (en sti) og

2. total lengde av brukt ledning er minimal

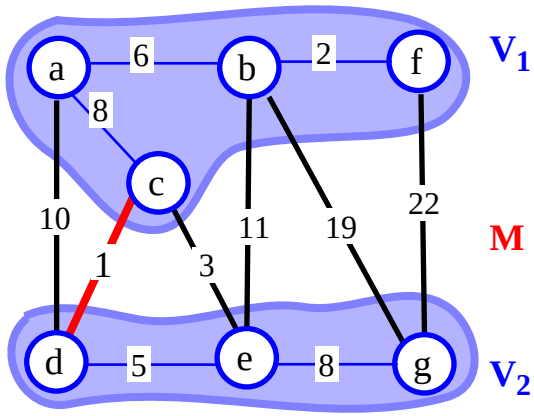


1. utspennende tre: DFS eller BFS (O(n+k)) – *lite sannsynlig at det blir minste*
2. finn alle trær og sammenlik deres vekter ? – *don't even think about it !*

12.25 La G = (V,E) være vektet og sammenhengende og la

- **V_1, V_2 være en partisjonering av V ($V_1 \cap V_2 = \emptyset$ og $V_1 \cup V_2 = V$).**
- **M – en delmengde av E av kanter med en ende i V_1 og andre i V_2.**

Det finnes en MST som inneholder e = min(M).

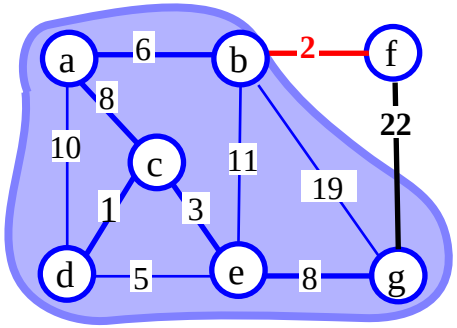


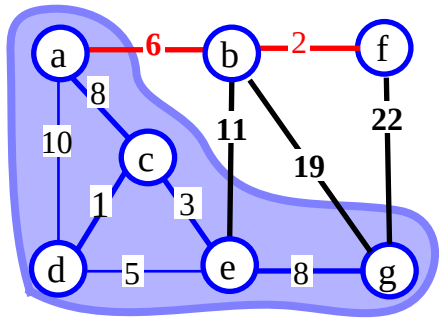
Begrunnelse :

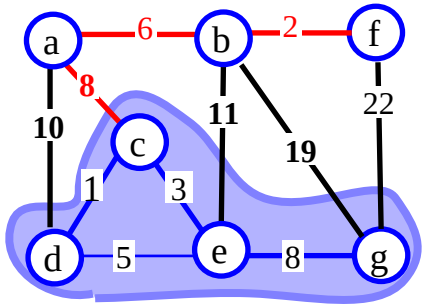
Et MST X må binde sammen V_1 og V_2 med en kant **k** : **v(e)** ≤ **v(k)**.

Legger vi kant **e** til X, får vi en syklus men kun på formen **k**–V_1–**e**–V_2.

Fjerner vi k får vi et tre med høyst samme vekt, dvs et MST.







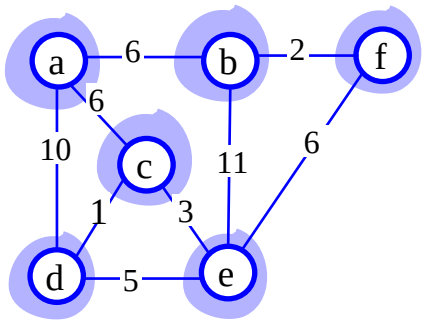
Dermed : hvis alle kanter har forskjellige vekter, er MST entydig bestemt

Kruskal algoritme MST

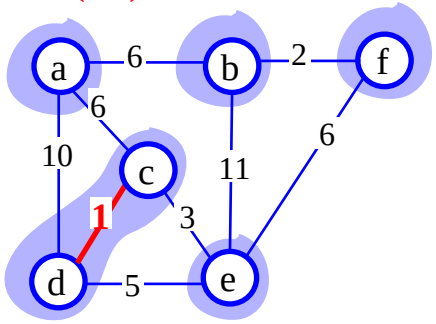
```
Kruskal(Graph G=(V,E))
  // sammenhengende, vektet
  for hver node v ε V : C(v) = {v}
  Q = PriorityQueue med alle kanter mht. vekt
  T = Ø
  while ( ! Q.isEmpty() )
     (v,u) = Q.removeMinElement();// Greedy
     if C(v) != C(u)
        legg (v,u) til T
        C(v) = C(u) = C(v) ∪ C(u)
```

LI: T inneholder MSTv for hver C(v)
– før inngangen i løkka - trivielt
– rundgang
  • hvis C(v) == C(u) :
    vekt(v,u) vekt(k) for alle k i C(v): **10.5**
  • hvis C(v) != C(u) : **10.5**

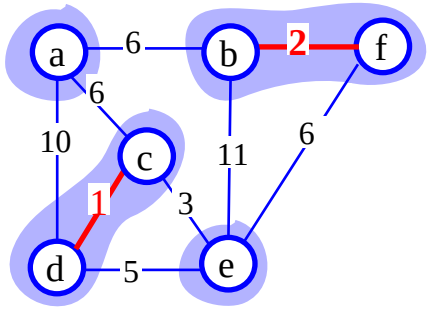
O(n +
  k log k +
  k *(log k + // heap
    C(v) != C(u) +
    C(v) ∪ C(u))
)



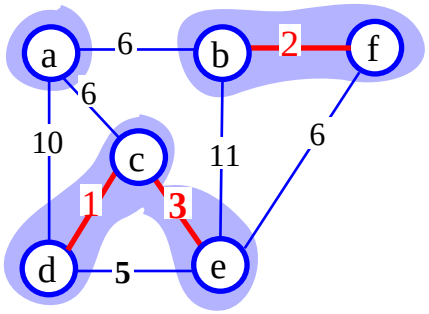
Q= 1 2 3 5 6 6 6 10 11
(c,d)



Q= 2 3 5 6 6 6 10 11
(b,f)

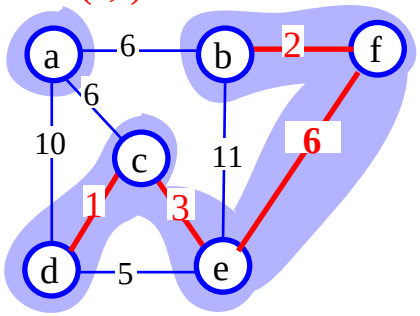


Q= 3 5 6 6 6 10 11
(c,e)

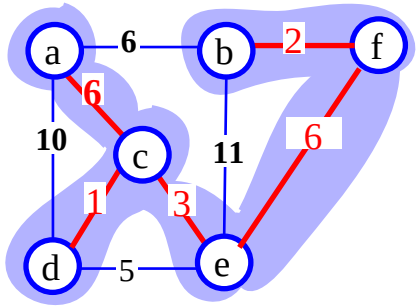


Q= 5 6 6 6 10 11
(d,e)

Q= 6 6 6 10 11
(e,f)

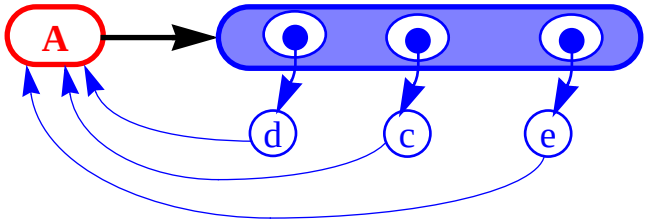


Q= 6 6 10 11
(a,c)



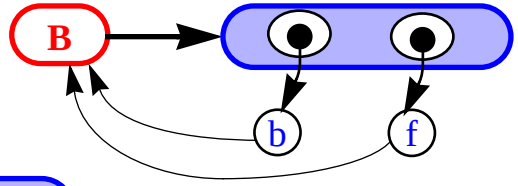
Implementasjon av Kruskal (finn/summer mengder)

C(v) er en sekvens – hvert element har en peker til dens 'hode':

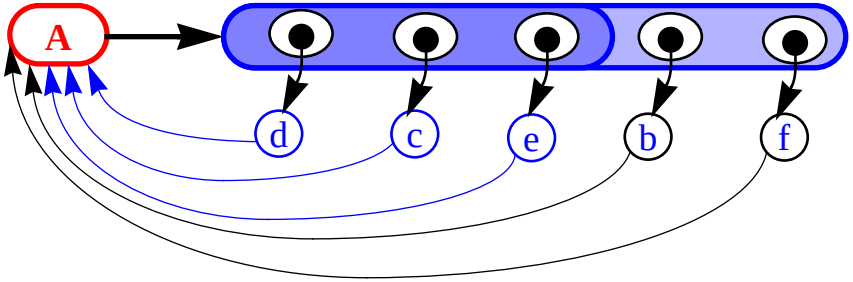


C(d) = **C**(c) = **C**(e) = **A**

1. **C**(d) != **C**(f) er ***O*(1)**



2. **C**(d) ∪ **C**(f)
for hvert element v i **C**(f)
C(d) . insertLast(v)



er ***O*(** min(**C**(d),**C**(f)) **)**

for hver node v ε V : C(v) = {v}	n
Q = PriorityQueue med alle kanter	k*logk
T = Ø	
while (! Q.isEmpty())	k*.....
(v,u) = Q.removeMinElement();	logk
if C(v) != C(u)	1
legg (v,u) til T	1
C(v) = C(u) = C(v) ∪ C(u)	**?**

n + k*log k + k*log k + 2*k + k***?** = ***O*(**n + k*logk + k***?)**

? C(v) ∪ C(u) utføres inntil alle n noder er i samme C(i)
dvs. k***?** = ***O*(**n)

n + k * log k + **n** = ***O*(k * log k)** da k = $O(n^2)$

10.6: .... ***O*(k log n)**

Oppsummering

- *Grafer*
  - *Graf ADT (rettet vs. ikke-rettet)*
  - *implementasjoner*
- ***Traversering** (generalisering fra Trær)*

  ***DFS** – forskjeller rettet vs. ikke-rettet tilfelle*

  BFS
- ***Algoritmer***

  transitiv tillukking
  - *n * DFS*
  - *Floyd-Warshall : nabo-matrise!*

  topologisk sortering

  DAG
- *Vektede Grafer*

  kortest sti
  - *Ford-Bellman algoritme : O(n*k)*
  - *Dijkstra algoritme : O((n+k) log n)*

  MST
  - *Kruskal algoritme*
  - *implementasjon : find/union*