Rettede og Vektede Grafer

I. Graf

II. Graf Traversering

III. Graf ADT og Implementasjon

IV. Rettede Grafer (diGraphs)

terminologi
DiGraph ADT og implementasjon
DFS av diGraph
transitiv tillukking
DAG og topologisk sortering

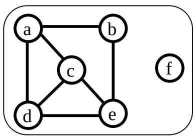
V. Vektede Grafer

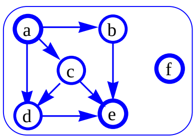
kortest sti
minimalt utspennende tre

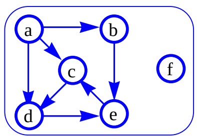
Kap. 9 (kursorisk 9.4.5)
Kap.10 (untatt 10.1.5, 10.2.2–10.2.4, 10.3)

Rettede grafer (diGraphs)

hver kant (u, v) betraktes som ordnet (rettet) par u → v.
(en ikke-rettet kant (u,v) = to rettede kanter u → v og v → u)







sammenhengende

graf : det finnes en sti mellom alle par av noder

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

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

kilde / sluk : en node uten noen inngående / utgående kanter

oppnålig : en node v kan nåes fra u dersom det finnes en rettet sti med n_1=u og n_k=v ade (ikke eda)

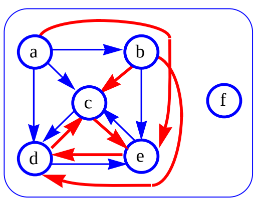
sterkt sammenhengende : hver node u er oppnåelig fra hver annen node v

rettet sykel : rettet enkel sti (hver node 1 gang) med $n_1 = n_k$

DAG : rettet, asyklisk graf – ingen (rettede) sykler

transitiv tilluking G* av G : V*=V og en kant u → v hvis G har en sti fra u til v

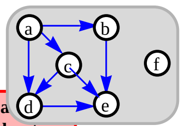
Er v oppnålig fra u ?
Finn alle v oppnålige fra u.
Er G sterkt sammenhengende ?
Er G asyklisk ?
Finn transitiv tilluking til G.



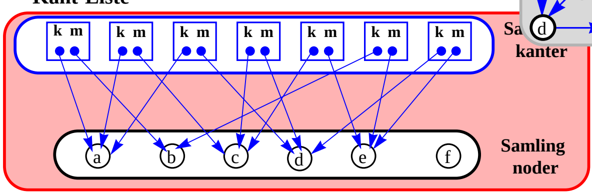
DiGraf ADT

```
// har kun rettede kanter
public interface DiGraph extends Graph {
//  int numEdges();
//  int numVertices();

/** hver node har både inn- og utkanter */
    int inDegree(Position v);                              //degree(v)
    int outDegree(Position v);

    PosSequence inIncidentE(Position v);                   //incidentE(v)
    PosSequence outIncidentE(Position v);

/** nabo noder til v kan ligge på inn- eller utkanter */
    PosSequence inAdjacentV(Position v);                   //adjacentV(v)
    PosSequence outAdjacentV(Position v);

/** kanter har mål og kilde noder */
    Position destination(Position e);                      //endV(e)
    Position origin(Position e);                           //opposite(v,e)

// Oppdatering:
/** en ny kant rettet fra v til u, med Object o */
//  Position insertE(Position v, Position u, Object o);
//  Position insertV(Object o);
//  Object removeE(Position e);
/** fjern v og alle dens ut-/innkanter */
//  Object removeV(Position v);

/** rett kanten e mot v */
    void setDirectionTo(Position e, Position v);

/** snu kanten e i motsatt retning */
    void reverse(Position e);
}
```

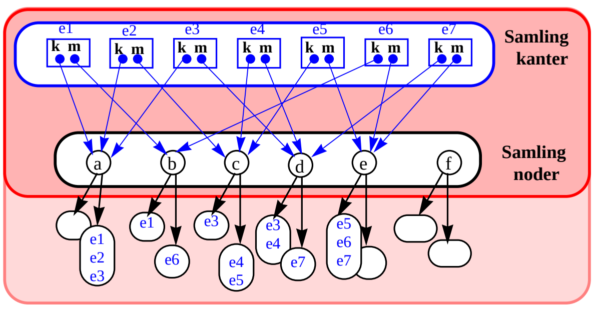

Implementasjon av DiGraph



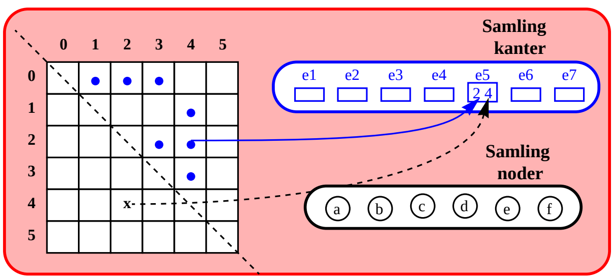
Kant-Liste



Nabo-Liste

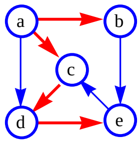


Nabo-Matrise



DFS på rettet graf

```
DFS(u)
  merk–u
  for hver kant e=(u,v) // rettet !
     if ( ! merk-et–v )
        merk–e–rød
        DFS(v)
```



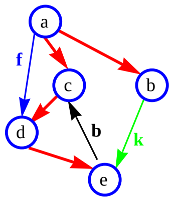
9.16 (9.12) DFS traversering av en rettet graf G fra en node s:

a) besøker alle noder *oppnålige* fra s

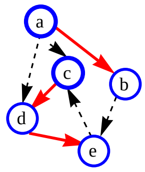
b) gir et utspennende, så kalt DFS, tre for delen oppnålig fra s

Kanter fra G som ikke er med i DFS kan deles i tre grupper:

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



Kan gi en skog selv om grafen er sammenhengende



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

- finne en delgraf oppnålig fra en gitt node

samt, ved iterasjon over alle noder, O**(n(n+k))** algoritmer for å :

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

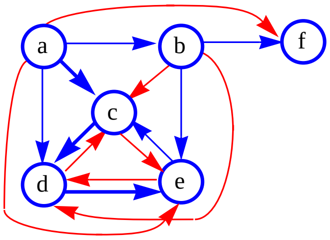
(BFS for rettede grafer har tilsvarende egenskaper til BFS for ikke-rettede grafer (etter-later kun bak- og kryss-kanter))

O(DFS)

operasjon	**Kant-Liste**	**Nabo-Liste**	**Nabo-Matrise**
PosSamling			
size	1	1	1
isEmpty	1	1	1
elements	n+k	n+k	n+k
positions	n+k	n+k	n+k
replace, swap	1	1	1
(di)Graph			
vertices/edges	n/k	n/k	n/k
endV, opposite, degree...	1	1	1
incidentE, adjacentV	**k**	**deg(v)**	**n**
areAdjacent	**k**	**min deg(v, u)**	**1**
insertE	1	1	1
insertV	1	1	n^2
removeE	1	1	1
removeV	k	deg(v)	n^2
DFS	**n * k**	**n + k**	**n * n**
hvis G er tett: k = $O(n^2)$	$\mathbf{n^3}$	$\mathbf{n^2}$	$\mathbf{n^2}$

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

Transitiv tilluking



TC(Graph G) *O*(**n * DFS**)
for hver node v ε V
DFS'(v) – legg til kant (v,u)
for hver besøkt node

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	$O(n^2 * k)$
Nabo Liste	$O(n^2 + nk)$
Nabo Matrise	$O(n^3)$

FloydWarshall(Graph G) $O(n^3$ *** areAdjacent)**
enumerer V : $v_1, v_2, \ldots, v_n$ (vilkårlig)
$G_0 = G$
for i = 1,2,...,n
$G_k = G_{k-1}$
for hver $a \neq b$, $a,b \neq i$
if G_{k-1}.areAdjacent(v_a,v_i) og G_{k-1}.areAdjacent(v_i,v_b)
legg (v_a,v_b) til G_k

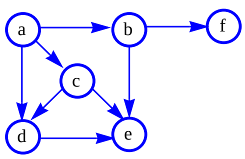
1 2 3 4 5 6
a b c d e f

$G_a = G_0 = \{$ 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	$O(n^3 * k)$
Nabo Liste	$O(n^3 * deg)$
Nabo Matrise	$O(n^3)$

DAG

rettet asyklisk graf
- arv i et OO-språk
- forkrav til kurs
- planlegging av avhengige aktiviteter

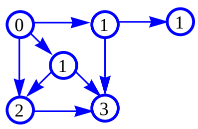


Topologisk ordning av en DiGraph G er en enumerering av noder $v_1, v_2 \ldots v_n$ slik at hvis (v_i,v_j) ε E så $i < j$.
(dermed, hvis det finnes en sti $v_i \ldots v_j$ så $i < j$)

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

9.21 DiGraph kan sorteres topologisk hvis og bare hvis den er asyklisk.

```
Queue TS(DiGraph G)
  Q, R = empty Queue
  for hver node v ε V
    in(v) = G.inDegree(v)
    if (in(v)==0) Q.enqueue(v)
  while (! Q.isEmpty() )
    h = Q.dequeue()
    for hver v ε G.outAdjacentV(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
ø	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							

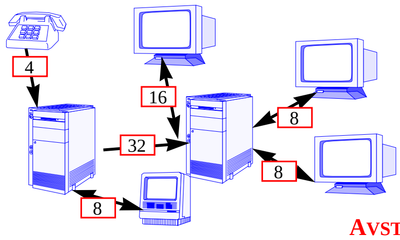
O(nk), O(n+k), $O(n^2)$
plass-behov O(n)

9.22 Er grafen syklisk vil TS returnere en ekte delmengde av noder.
-> TS gir en O**(n+k) algoritme for å se om en DiGraph er asyklisk**

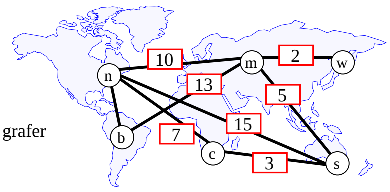
Vektede Grafer

- en graf der hver kant har et **vekt-attributt**
  - **vekter skal være TO (typisk heltall) og man designer en** Comparator **for sam-menlikning av kanter mht. vekt**

NETTVERK KAPASITET



AVSTAND



- I tillegg til vanlige graf-problemer, spør man i forbindelse med vektede grafer
  – hva er **korteste** sti fra u til v ?
  – hva er **minste** utspennende tre ?
  – ..... minste/korteste/billigste ..... ?

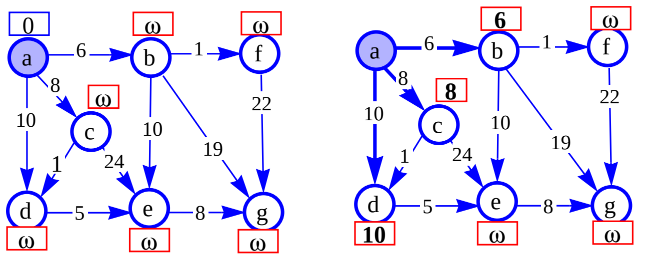
- public interface VGraph extends Graph
  // public interface VdiGraph extends DiGraph
  { Object **vekt**(Position e);
  void **setVekt**(Position e, Objekt k); }

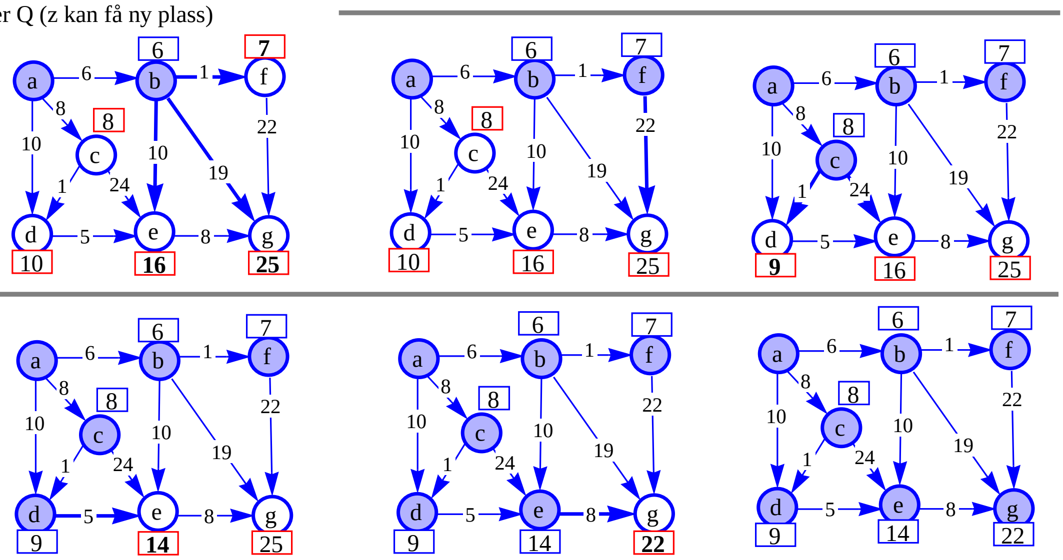
- Implementasjon er en rett-fram utvidelse av tilsv. implementasjon av (di)Graph der Kant-Posisjoner lagrer vekt-Objekter

Korteste sti (single-source shortest-paths) : vekt(e) ≥ 0 ...BFS

Finn korteste sti fra a til alle / en node(r)

initialiser **D**(a)=0 og **D**(v)=ω for alle v≠a
sett alle noder i en **PrQueue Q** mht. **D**
while (! Q.isEmpty())
  v = **Q.remMin()** *// Greedy*
  for hver z ε G.**outAdjacentV**(v)
    if (**D**(v)+**vekt**(v,z) < **D**(z))
      D(z)= **D**(v) + **vekt**(v,z)
      oppdater Q (z kan få ny plass)





Dijkstra's SS-SP

1. initialiser D(a)=0 og D(v)=ω for alle v≠a
2. sett alle noder i en PrQueue Q mht. D // **LI**
3. while (! Q.isEmpty()) // **LI**→
4. v = Q.remMin()
5. for hver z ε G.**outAdjacentV**(v) // **N-L**
6. if (D(v)+vekt(v,z) < D(z))
   D(z)= D(v) + vekt(v,z)
7. **oppdater Q** (z kan få ny plass) // → **LI** // **heap+**

LI : alle noder x som har blitt fjernet fra Q har D(x) = d(a,x)

– holder før inngangen siden ingen node ble fjernet.

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

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

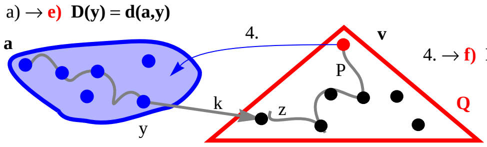
a) Anta ikke og la **v** være **den første** node for hvilken **D(v) > d(a,v)** ved 4.

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 **k = (y,z)**

a) → **e) D(y) = d(a,y)**



4. → **f) D(v) ≤ D(z)**

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

siden k=(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(k) = D(z)

Men da:

D(v) $\leq^{f}$ D(z) $=^{h}$ d(a,z) ≤ d(a,z) + d(z,v) $=^{c}$ d(a,v) – **motsier a)** D(v) > d(a,v)

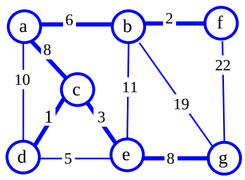
for N-L/heap har vi totalt **n+k** iterasjoner á **log n** oppdatering (inne i heap!): ***O*((n+k)log n)**

MST

Gitt avstander mellom forskjellige byer, strek ledningene slik at

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

2. total lengde av brukt ledning er minimal

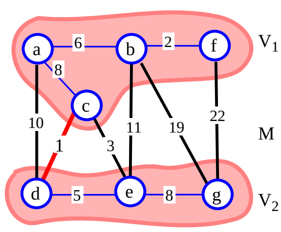


1. utspennende tre: DFS eller BFS (O(n+k))

2. ... ?

Finne alle mulige og sammenlikne deres vekter – *don't even think about it*!

10.5 La G=(V,E) være vektet og sammenhengende og V_1, V_2 være en partisjonering av V ($V_1 \cap V_2 = \emptyset$ og $V_1 \cup V_2 = V$). La M være en delmengde av E av alle kanter med en ende i V_1 og andre i V_2 og la e = min(M). Det finnes en MST som inneholder e.

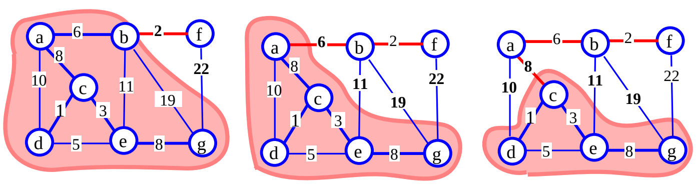


Begrunnelse :

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

Legger vi til en kant e, får vi en sykel 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: har alle kanter forskjellige vekter, er MST entydig bestemt



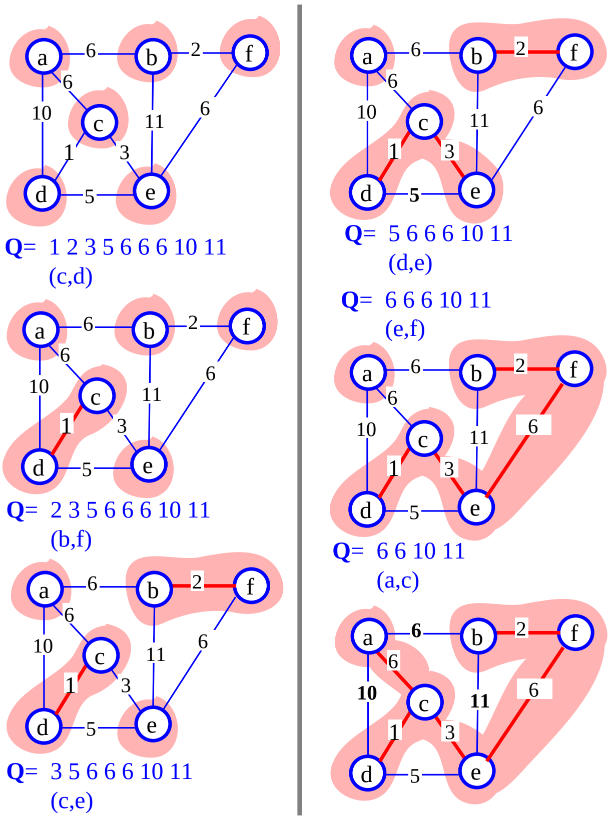
Kruskal algoritme MST

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

LI: T inneholder nøyaktig MST$_V$ 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)
  • hvis C(v) != C(u) : 10.5

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

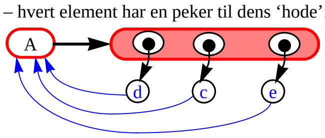


Implementasjon av Kruskal

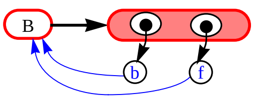
Kruskal(Graph G=(V,E))	
for hver v ε V C(v) = {v}	**n +**
Q = PrQueue med alle kanter mht. vekt	**k * log k**
T = Ø	+
while (! **Q**.isEmpty())	**k * (**
(v,u) = **Q.remMin**();	**log k +**
if **C**(v) != **C**(u)	**1. ???**
legg (v,u) til **T**	+
C(v) = **C**(u) = **C**(v) ∪ **C**(u)	**2. ???)**

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

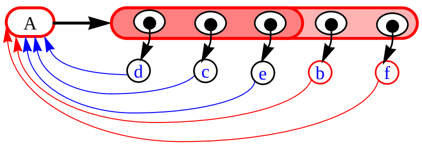
C(d) = C(c) = C(e)



1. C(d) == C(b) er ***O*(1)**



2. C(d) ∪ **C**(f)



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

O(n + k * log k + k * log(k + n)**)** = ***O***(n + k* log(k+n) **)**

(C(v) ∪ C(u)) utføres inntil alle n noder er i samme C(i)

O(n + k * log k + k * log k + n) = ***O*(n + k * log k)**

10.6. k ≤ (n^2–n)/2 ≤ n^2 : log k ≤ 2 log n = O(log n) .... ***O*(k log n)**