Rettede og Vektede Grafer

I. Graf

II. Graf Traversering

III. Graf ADT og Implementasjon

IV. Rettede Grafer (DiGraphs)

terminologi
ADT og implementasjoner
DFS/BFS av DiGraph
transitiv tillukking
DAG og topologisk sortering

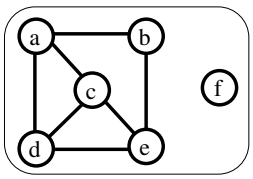
V. Vektede Grafer

kortest sti
minimalt utspennende tre

Kap. 9 (kursorisk 9.4.5)
Kap.10.1 og 10.2.1 (untatt 10.2.2–10.2.4, 10.3)

Rettet Graf (DiGraph)

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



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

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 ?

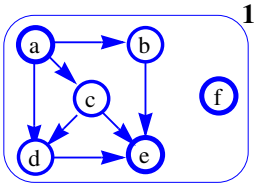
Finn alle v oppnålige fra u.

Er G sterkt sammenhengende ?

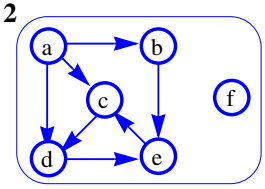
Er G asyklisk ?

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

1



2



sti : en sekvens av noder ... : ade (ikke eda)

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

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

oppnåelig : 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 graf : hver node u er oppnåelig fra hver annen node v

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

transitiv tilluking

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

Graf ADT

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

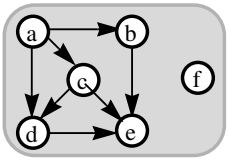
```
package jdsl.core.api;

public interface InspectableGraph extends PositionalContainer {
                                                    // throws
  int numVertices();  int numEdges();
  Enumeration vertices();
  Enumeration edges();
/** # rettede og ikke-rettede nabokanter */
  int degree(Vertex v);                             InvalidPos
  Vertex[] endVertices(Edge e)                      InvalidPos
  Vertex opposite(Vertex v, Edge e);                InvalidPos
/** nabonoder langs alle kanter
    inn–, utgående samt ikke-rettede */
  Enumeration adjacentVertices(Vertex v)            InvalidPos
/** rettede og ikke-rettede nabokanter */
  Enumeration incidentEdges(Vertex v)               InvalidPos
}
```

```
                                                    // throws
  Enumeration unDirectedEdges();
  Enumeration directedEdges();
  int outDegree(Vertex v);
  int inDegree(Vertex v);
  Vertex origin(Edge e)                             InvalidPos, InvalidEdge
  Vertex destination(Edge e)                        InvalidPos, InvalidEdge
  boolean isDirected(Edge e)                        InvalidPos
  Enumeration outAdjacentVertices(Vertex v)         InvalidPos
  Enumeration inAdjacentVertices(Vertex v)          InvalidPos
  Enumeration outIncidentEdges(Vertex v)            InvalidPos
  Enumeration inIncidentEdges(Vertex v)             InvalidPos
```

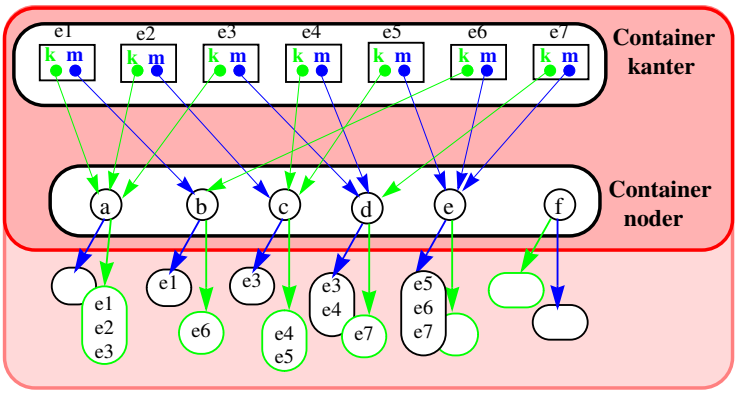
```
public interface Graph extends InspectableGraph {
  Vertex insertVertex(Object o);
  Edge insertEdge(Vertex u, Vertex v, Object o);
                                                    InvalidPos
  Object removeEdge(Edge e)                         InvalidPos
  Object removeVertex(Vertex v)                     InvalidPos
}
```

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


Implementasjon av Graph



Kant-Liste

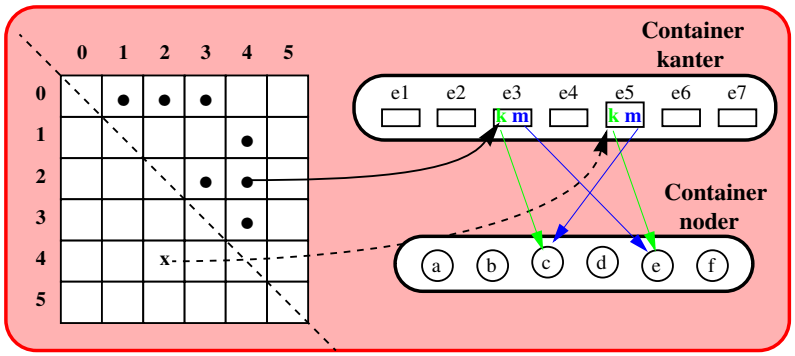


utvid tidligere implementasjoner slik at kant-klassen

- *skiller mellom* **Vertex origin** *og* **Vertex destination**, *og*
- *har et atributt* **boolean isDirected**

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

DFS på en rettet graf

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

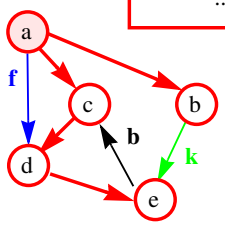
a) besøker alle noder *oppnåelige* fra a

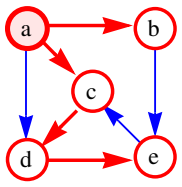
b) gir et utspennende, DFS tre for delen oppnåelig fra a

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

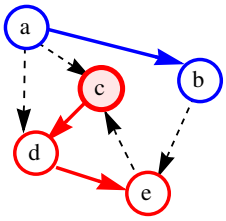
– Kanter fra G som ikke er med i DFS kan deles i tre grupper:

- **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



kompleksistet	kant-liste	nabo-liste	nabo-matrise
outIncidentEdges(v)	**k**	**deg v**	**n**
DFS	**n * k**	**n + k**	**n * n**
tett graf: k=O(n^2)	**n^3**	**n^2**	**n^2**

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

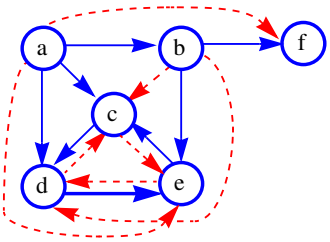
- finne en delgraf oppnåelig fra en gitt node

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 (etterlater kun bak- og kryss-kanter)

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 : $v_1,v_2,...,v_n$ (vilkårlig)
  $G_0 = G$
  for k = 1,2,...,n
    $G_k = G_{k-1}$
    for hvert par a ≠ b, a,b ≠ k (av tall 1...n, dvs. av noder)
      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

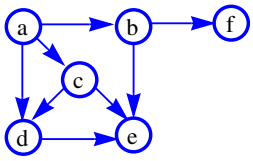
i en rettet graf:
G_{k-1} har kant (v_a,v_i) og (v_i,v_b)

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

DAG



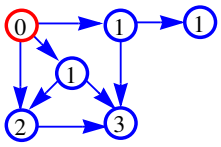
rettet asyklisk graf
- *arv i et OO-språk*
- *forkrav til kurs*
- *planlegging av avhengige aktiviteter*

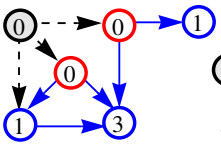
1	2	3	4	5	6
a	**b**	**f**	**c**	**d**	**e**
a	**c**	**d**	**b**	**e**	**f**
....					
a	d	c	e	b	f

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

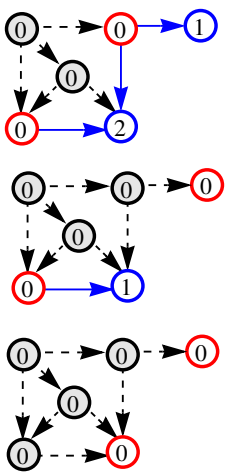
9.21. En rettet graf kan sorteres topologisk hvis og bare hvis den er asyklisk.

Queue **TS**(Graph G) O(**nk**)
  Q, R = empty Queue O(**n+k**)
  for hver node v ε **V** $O(n^2)$
  { 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							



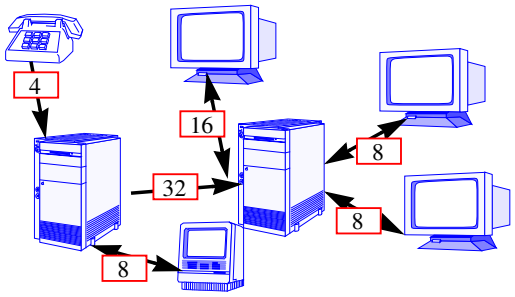
9.22 Er grafen syklisk vil TS returnere en ekte delmengde av noder.
  -> TS gir en O(n+k) algoritme for å se om en rettet graf er asyklisk

Vektede Grafer

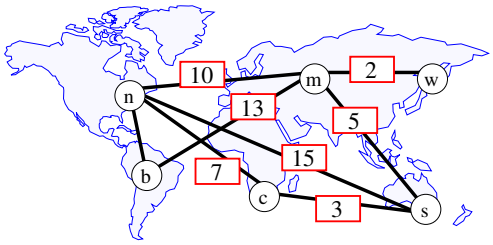
en graf der hver kant har et vekt-attributt

- vekter skal være TO (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
  - *hva er **korteste** sti fra u til v ?*
  - *hva er **minste** utspennende tre ?*
  - *..... minste / korteste / billigste ..... ?*

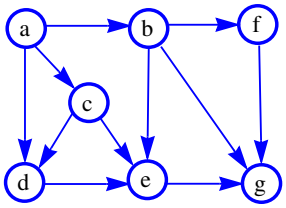
Kortest sti

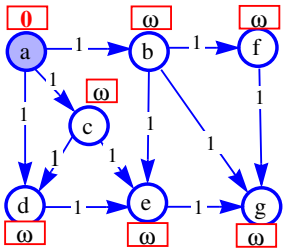
...BFS

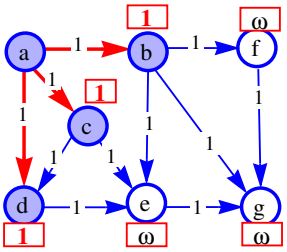
Finn (lengden av) korteste sti fra a til en bestemt/alle andre node(r)

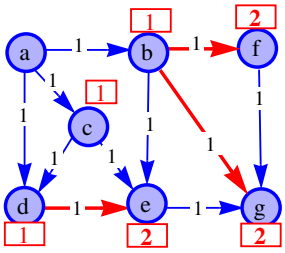
Ford-Bellman **BFS** :

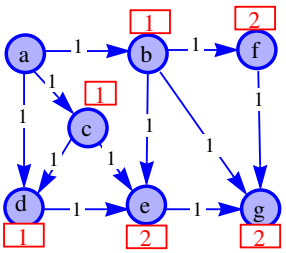
```
for each node v : D(v)= ω          // ω er et maksimalt tall (her ω > n)
D(s) = 0;                          // s er startnoden
for (i=1; i<n; i++)                // n er antall noder i grafen G        O(n*k)
    for each kant (k,m)
        if ( D(k) + 1 < D(m) )   D(m) = D(k) + 1
```









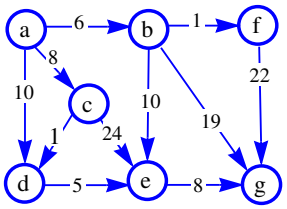


Kortest sti (single-source shortest-paths) :

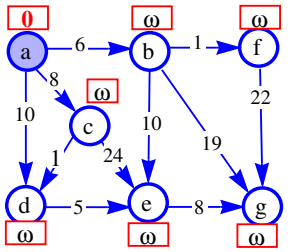
Finn (lengden av) korteste sti fra a til en bestemt/alle andre node(r)

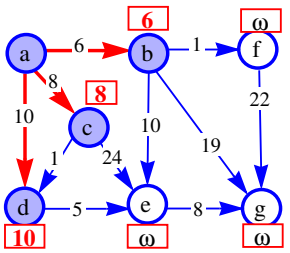
Ford-Bellman **SS-SP** :

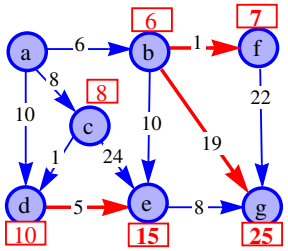
```
for each node v : D(v)= ω          // ω er et maksimalt tall (her ω > n)
D(s) = 0;                          // s er startnoden
for (i=1; i<n; i++)                // n er antall noder i grafen G        O(n*k)
    for each kant (k,m)
        if ( D(k) + vekt(k,m) < D(m) )   D(m) = D(k) + vekt(k,m)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

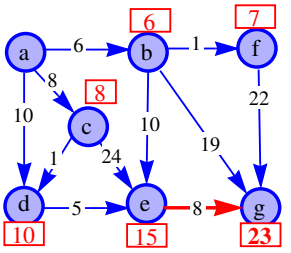


graf	vekter
ikke-rettet	0
rettet	vilkårlige







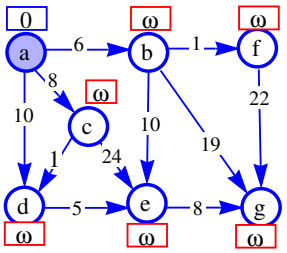


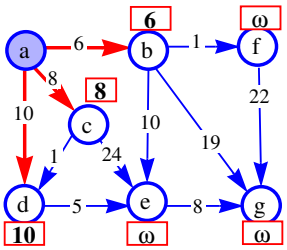
10.3 Etter Ford-Bellman (G,a):

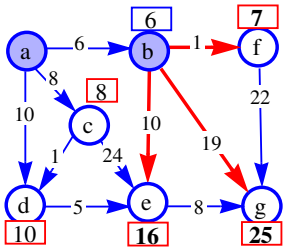
- hvis det finnes en kant (k,m) med
  D(k) + vekt(k,m) < D(m),
  så har G en negativ sykel
- ellers D(v) = d(a,v) for alle noder v

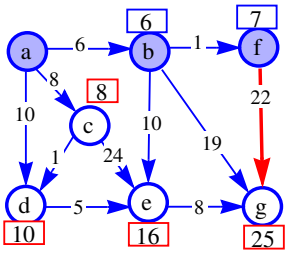
Kortest sti (SS-SP) : Dijkstra algoritme (vekt(e) ≥ 0)

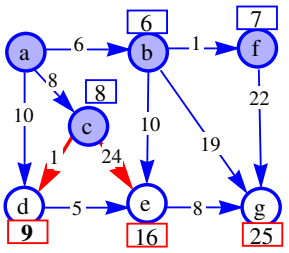
```
initialiser D(a)=0 og D(v)=ω for alle v a
sett alle noder i en PriorityQueue Q mht. D
while ( ! Q.isEmpty() )
  v = Q.removeMinElement()    // Greedy
  for hver z ε G.outAdjacentVertices(v)
    if ( D(v)+vekt(v,z) < D(z) )
      D(z)= D(v) + vekt(v,z)
      oppdater Q    // z kan få ny nøkkel
```

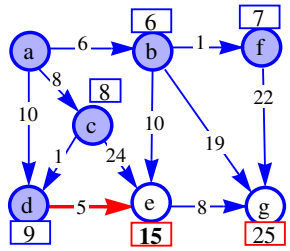


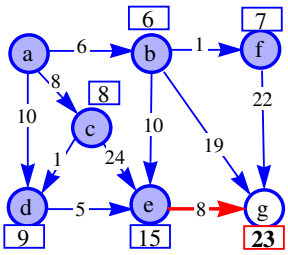


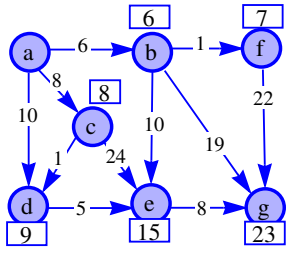








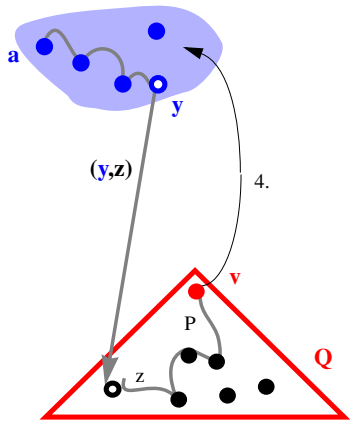




Dijkstra's SS-SP

1. initialiser D(a)=0 og D(v)=ω for alle v a
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+k) log n)	**O**(n^2 log n)
usortert sekvens	**O**(n*n+k)	**O**(n^2)
sortert sekvens	**O**(n +k*n)	**O**(n^3)



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 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:

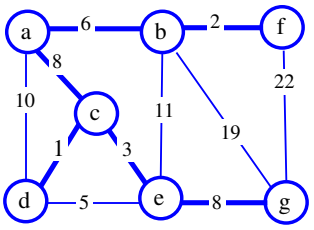
D(v) ≤ ^{f} D(z) =^{h} d(a,z) ≤ d(a,z) + d(z,v) =^{c} d(a,v) – motsier a) D(v) > d(a,v)

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

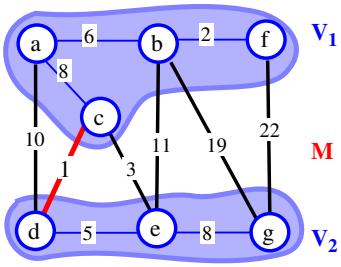


1. utspennende tre: DFS eller BFS (O(n+k)) – *lite sannsynlig at det blir minste*
2. finn alle trær og sammenlik deres vekter ? – *don't even think about it !*

10.5 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).

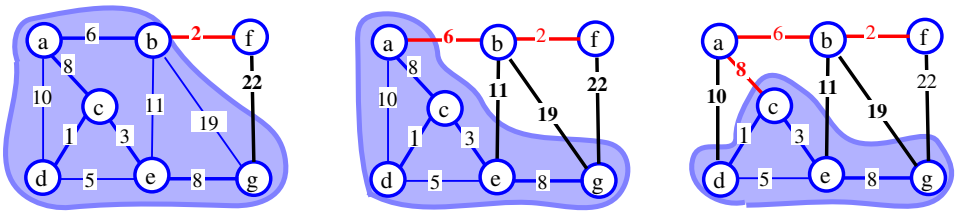


Begrunnelse :

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.



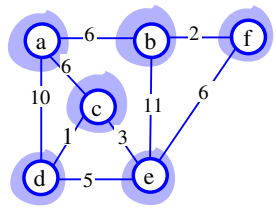
Dermed : hvis alle kanter har forskjellige vekter, er MST entydig bestemt

Kruskal algoritme MST

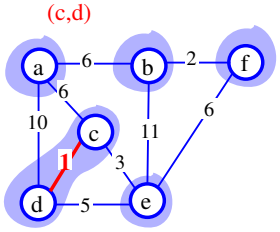
```
Kruskal(Graph G=(V,E))
  // sammenhengende, vektet
  for hver node v ε V : C(v) = {v}
  Q = PriorityQueue med alle kanter mht. vekt
  T = Ø
  while ( ! Q.isEmpty() )
     (v,u) = Q.removeMinElement();// Greedy
     if C(v) != C(u)
        legg (v,u) til T
        C(v) = C(u) = C(v) ∪ C(u)
```

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**

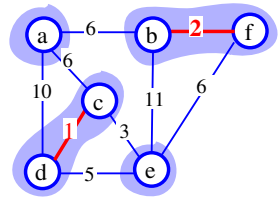
O(n +
  k log k +
  k *(log k + // heap
      C(v) != C(u) +
      C(v) ∪ C(u))
)



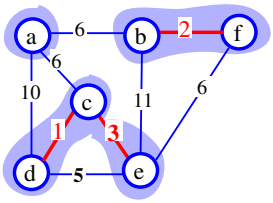
Q= 1 2 3 5 6 6 6 10 11
(c,d)



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

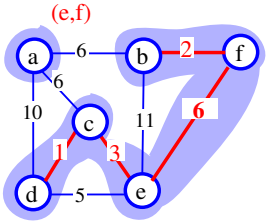


Q= 3 5 6 6 6 10 11
(c,e)

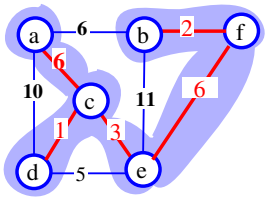


Q= 5 6 6 6 10 11
(d,e)

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



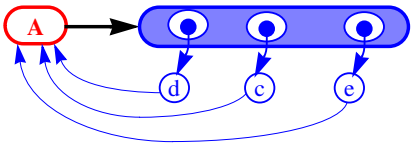
Q= 6 6 10 11
(a,c)



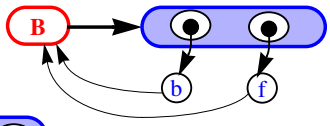
Implementasjon av Kruskal (finn/summer mengder)

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

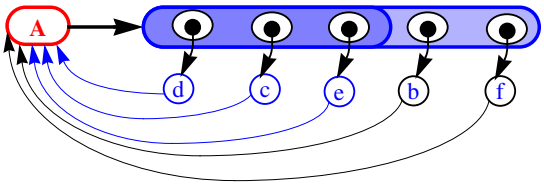
C(d) = **C**(c) = **C**(e) = **A**



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

- *Rettede Grafer*
  - *Graf ADT*
  - *implementasjoner*
- ***Algoritmer***

  DFS *– forskjeller fra ikke-rettet tilfelle*

  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*