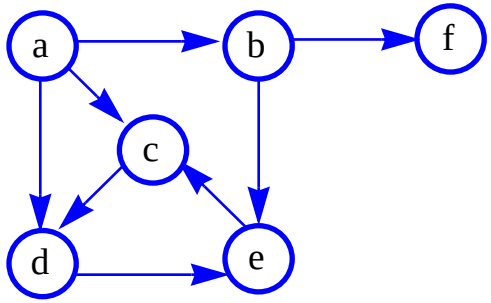
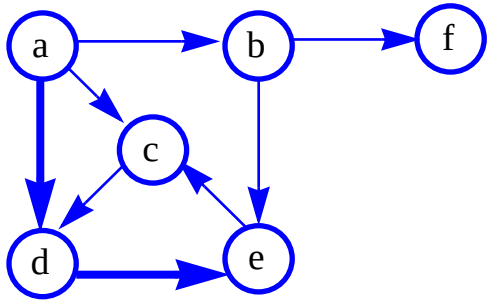
V.a) Transitiv tillukning



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

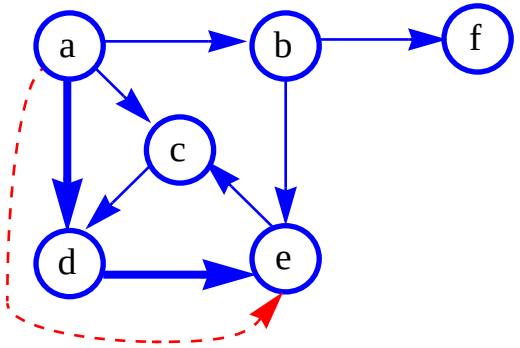
V.a) Transitiv tillukning



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
a	b c d

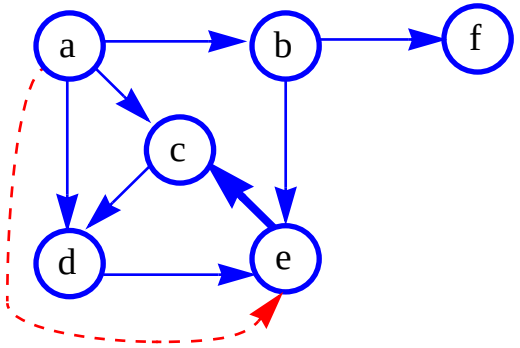
V.a) Transitiv tillukning



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

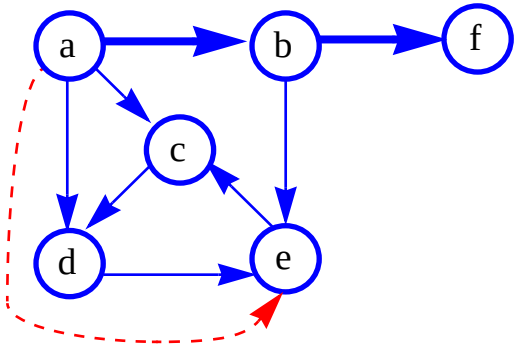
V.a) Transitiv tillukning



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

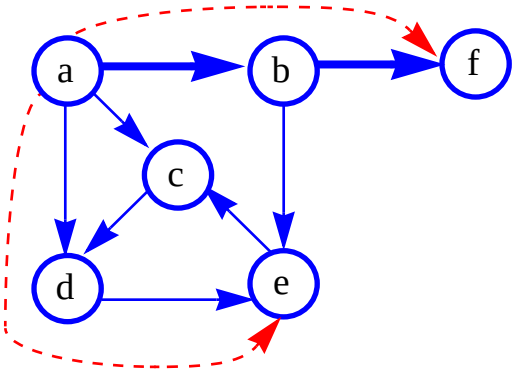
V.a) Transitiv tillukning



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

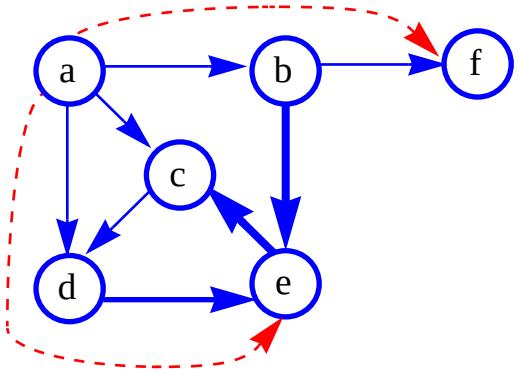
V.a) Transitiv tillukning



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

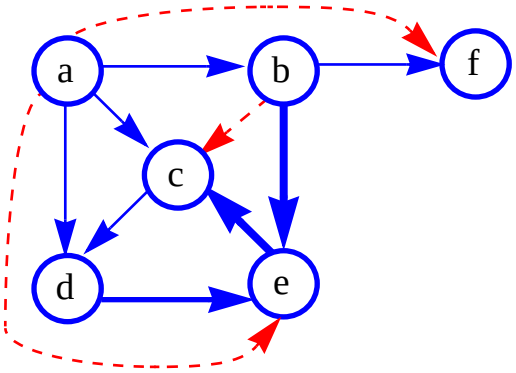
V.a) Transitiv tillukning



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	

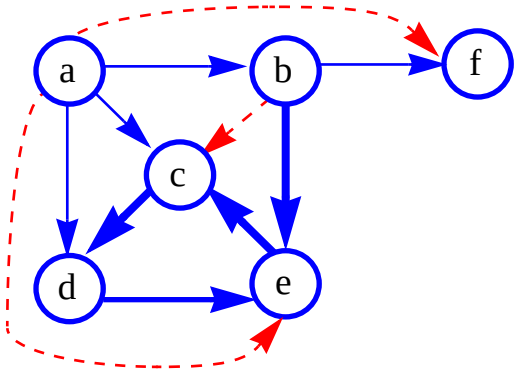
V.a) Transitiv tillukning



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

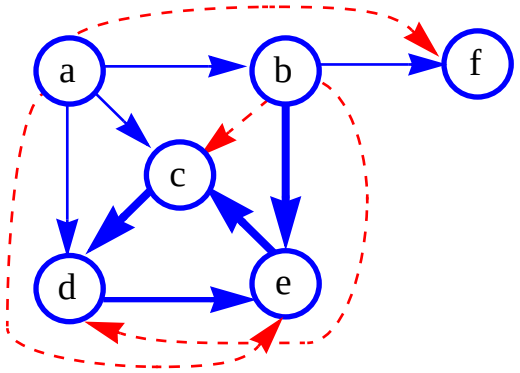
V.a) Transitiv tillukning



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

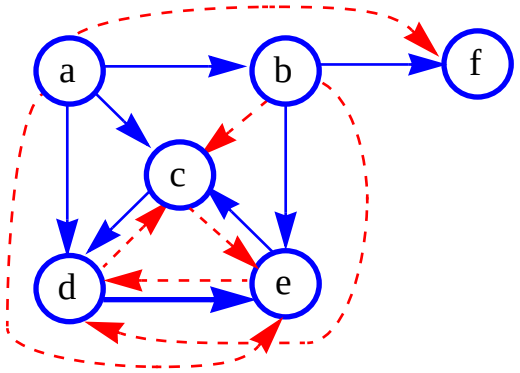
V.a) Transitiv tillukning



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

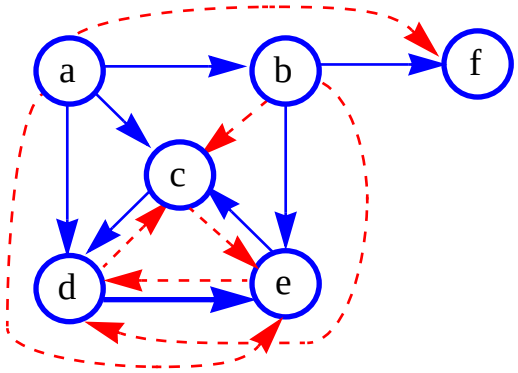
V.a) Transitiv tillukning



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		

V.a) Transitiv tillukning

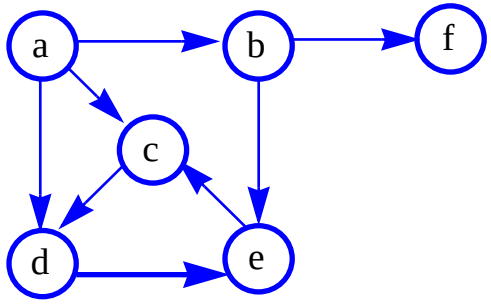


IterertDFS(Graph G) O**(n * DFS)**
  for hver node v ε V
    DFS(v) – utvid G med kant (v,u)
    for hver u i DFS-tre til v

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		

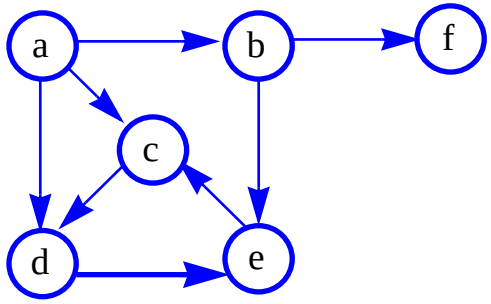
Kant-Liste	$O(n^2 * k)$
Nabo-Liste	$O(n^2 + nk)$
Nabo-Matrise	$O(n^3)$

V.a) Transitiv tillukning



FloydWarshall(Graph G)
  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

V.a) Transitiv tillukning

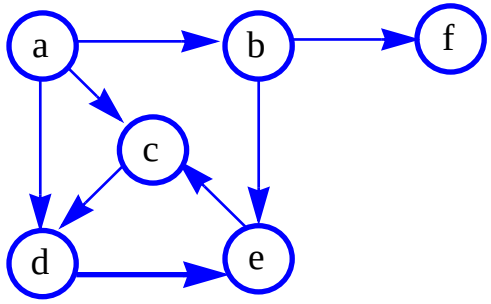


FloydWarshall(Graph G)
  enummerer V : $v_1,v_2,...,v_n$ (vilkårlig)
  $G_0 = G$
  for k = 1,2,...,n
    $G_k = G_{k-1}$
    for hvert tallpar a ≠ b, a,b ≠ k
      if G_{k-1}.**areAdjacent**(v_a,v_k) og G_{k-1}.**areAdjacent**(v_k,v_b)
        legg kant (v_a,v_b) til G_k

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

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

V.a) Transitiv tillukning



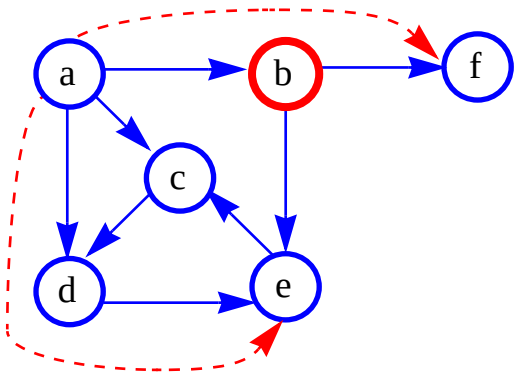
FloydWarshall(Graph G)
  enummerer V : $v_1,v_2,...,v_n$ (vilkårlig)
  $G_0 = G$
  for k = 1,2,...,n
    $G_k = G_{k-1}$
    for hvert tallpar a ≠ b, a,b ≠ k
      if G_{k-1}.**areAdjacent**(v_a,v_k) og G_{k-1}.**areAdjacent**(v_k,v_b)
        legg kant (v_a,v_b) til G_k

i en rettet graf:
G_{k-1} har kant (v_a,v_k) og (v_k,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 $\}$

V.a) Transitiv tillukning



FloydWarshall(Graph G)
  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

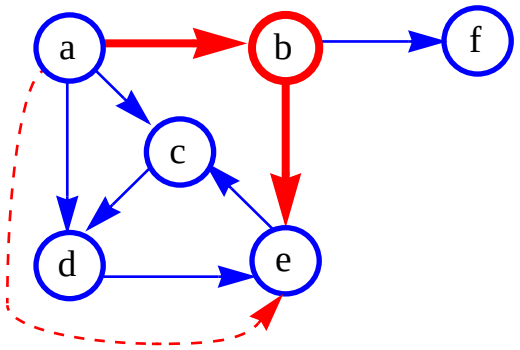
i en rettet graf:
G_{k-1} har kant (v_a,v_k) og (v_k,v_b)

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 $\}$

V.a) Transitiv tillukning



FloydWarshall(Graph G)
  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

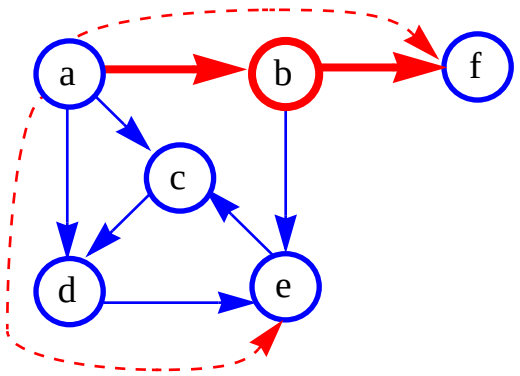
i en rettet graf:
G_{k-1} har kant (v_a,v_k) og (v_k,v_b)

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,

V.a) Transitiv tillukning



FloydWarshall(Graph G)
  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

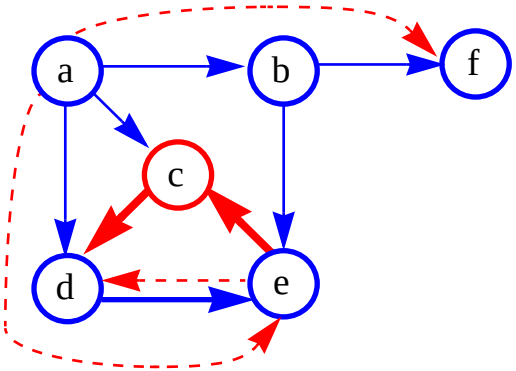
i en rettet graf:
G_{k-1} har kant (v_a,v_k) og (v_k,v_b)

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 $\}$

V.a) Transitiv tillukning



FloydWarshall(Graph G)
  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_k) og (v_k,v_b)

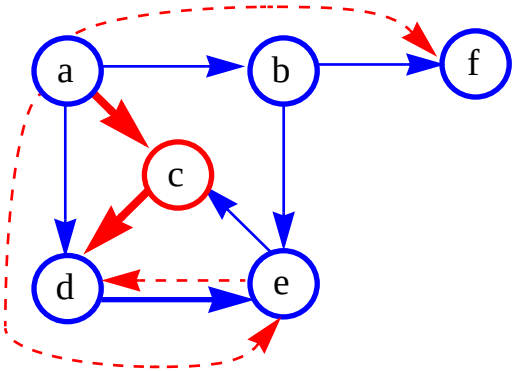
1	2	3	4	5	6
a	b	c	d	e	f

$G_a = G_0 = \{$ ab, ac, ad, cd, de, ec, bf $\}$

$G_b = G_a \cup \{$ ae, af $\}$

$G_c = G_b \cup \{$ ed $\}$ (ad)

V.a) Transitiv tillukning



FloydWarshall(Graph G)
  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_k) og (v_k,v_b)

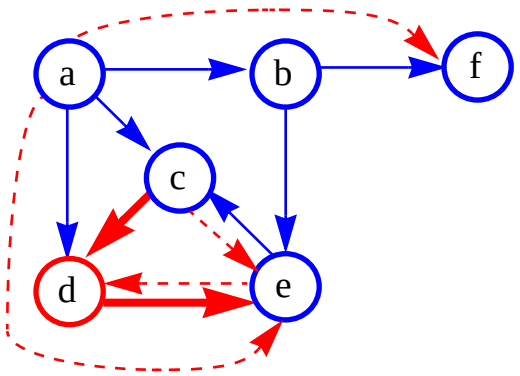
1	2	3	4	5	6
a	b	c	d	e	f

$G_a = G_0 = \{$ ab, ac, ad, cd, de, ec, bf $\}$

$G_b = G_a \cup \{$ ae, af $\}$

$G_c = G_b \cup \{$ ed $\}$ (ad)

V.a) Transitiv tillukning



FloydWarshall(Graph G)
  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_k) og (v_k,v_b)

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

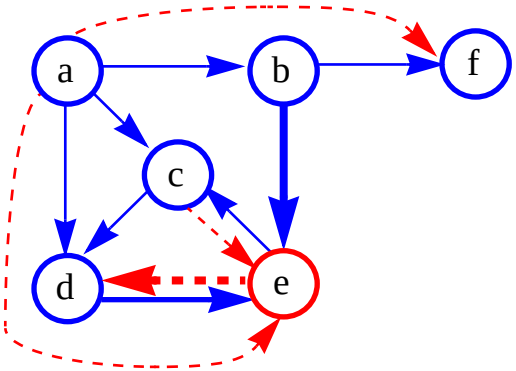
$G_a = G_0 = \{$ ab, ac, ad, cd, de, ec, bf $\}$

$G_b = G_a \cup \{$ ae, af $\}$

$G_c = G_b \cup \{$ ed $\}$ (ad)

$G_d = G_c \cup \{$ ce $\}$

V.a) Transitiv tillukning



FloydWarshall(Graph G)
  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_k) og (v_k,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 $\}$

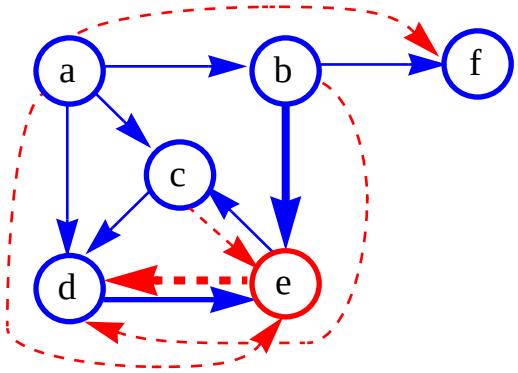
$G_b = G_a \cup \{$ ae, af $\}$

$G_c = G_b \cup \{$ ed $\}$ (ad)

$G_d = G_c \cup \{$ ce $\}$

$G_e = G_d \cup \{$ bc, bd, dc $\}$ (ac)

V.a) Transitiv tillukning



FloydWarshall(Graph G)
  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_k) og (v_k,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 $\}$

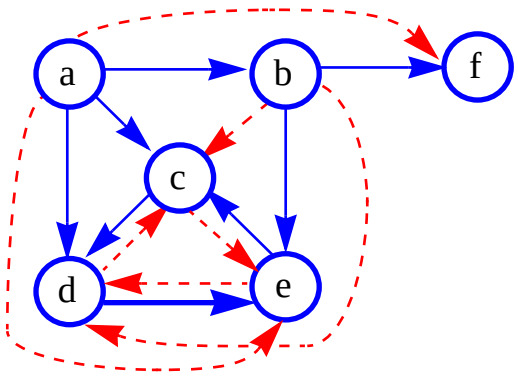
$G_b = G_a \cup \{$ ae, af $\}$

$G_c = G_b \cup \{$ ed $\}$ (ad)

$G_d = G_c \cup \{$ ce $\}$

$G_e = G_d \cup \{$ bc, **bd**, dc $\}$ (ac)

V.a) Transitiv tillukning



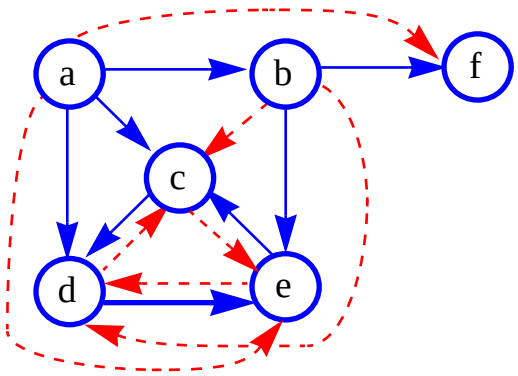
FloydWarshall(Graph G)
  enummerer V : $v_1,v_2,...,v_n$ (vilkårlig)
  $G_0 = G$
  for k = 1,2,...,n
    $G_k = G_{k-1}$
    for hvert tallpar a ≠ b, a,b ≠ k
      if G_{k-1}.**areAdjacent**(v_a,v_k) og G_{k-1}.**areAdjacent**(v_k,v_b)
        legg kant (v_a,v_b) til G_k

i en rettet graf:
G_{k-1} har kant (v_a,v_k) og (v_k,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)
$G_f = G_e$

V.a) Transitiv tillukning



FloydWarshall(Graph G)
  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_k) og (v_k,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)

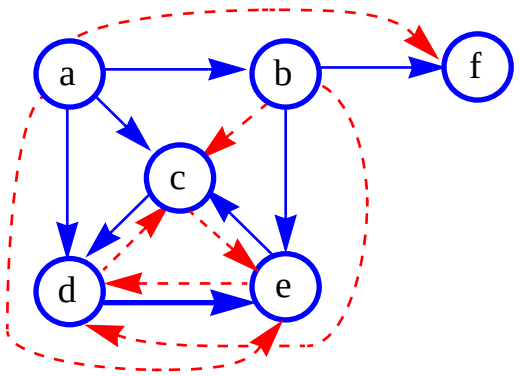
$G_d = G_c \cup \{$ ce $\}$

$G_e = G_d \cup \{$ bc, bd, dc $\}$ (ac)

$G_f = G_e$

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

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	$O(n^2 * k)$
Nabo-Liste	$O(n^2 + nk)$
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_k) og (v_k,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)

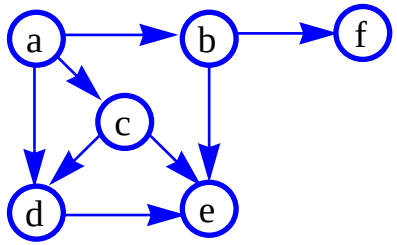
$G_f = G_e$

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

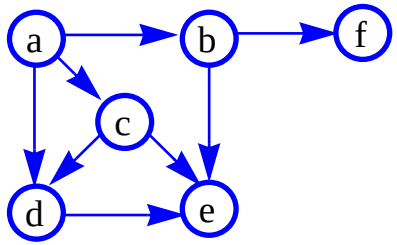
V.b) DAG - Directed Acyclic Graph

rettet asyklisk graf

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



V.b) DAG - Directed Acyclic Graph

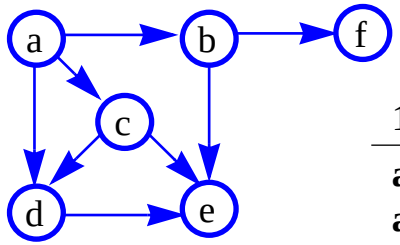


rettet asyklisk graf

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

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 $i < j$)

V.b) DAG - Directed Acyclic Graph



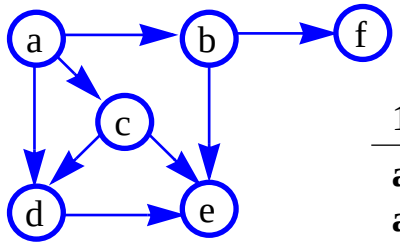
1	2	3	4	5	6
a	**b**	**f**	**c**	**d**	**e**
a	**c**	**d**	**b**	**e**	**f**
....					
a	d	c	e	b	f

rettet asyklisk graf

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

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 $i < j$)

V.b) DAG - Directed Acyclic Graph



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

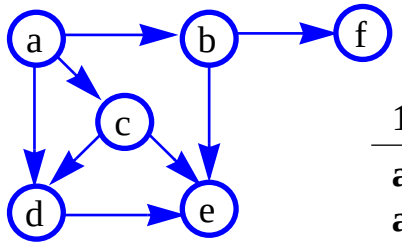
rettet asyklisk graf

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

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.

V.b) DAG - Directed Acyclic Graph



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

rettet asyklisk graf

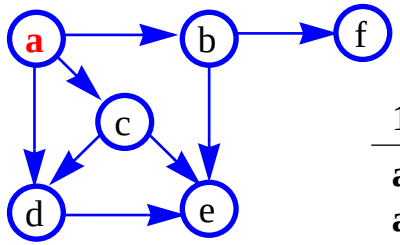
- *arv i et OO-språk*
- *forkrav til kurs*
- *planlegging av avhengige aktiviteter*

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)
  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.outAdjacentVertices(h)
     {  in(v) = in(v) – 1
        if (in(v)==0) Q.enqueue(v)  }
     R.enqueue(h)  }
  return R
```

V.b) DAG - Directed Acyclic Graph



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

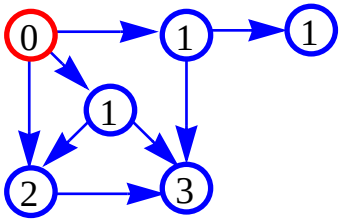
rettet asyklisk graf

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

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 $i < j$)

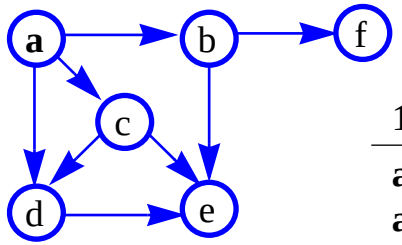
12.21. En rettet graf kan sorteres topologisk hvis og bare hvis den er asyklisk.

```
Queue TS(Graph G)
  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.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

V.b) DAG - Directed Acyclic Graph



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

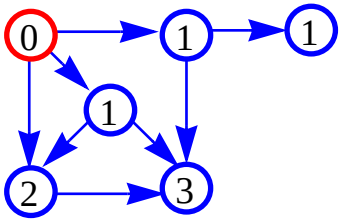
rettet asyklisk graf

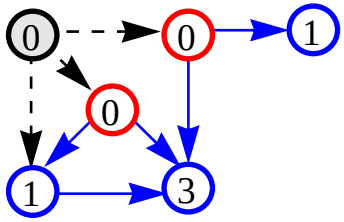
- *arv i et OO-språk*
- *forkrav til kurs*
- *planlegging av avhengige aktiviteter*

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.

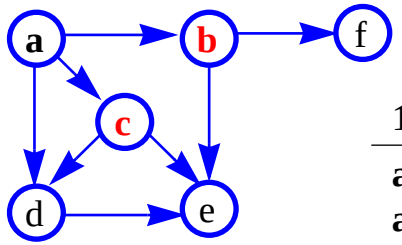
```
Queue TS(Graph G)
  Q, R = empty Queue
  for hver node v ε V
  {  in(v) = G.inDegree(v)
     if (in(v) == 0) Q.enqueue(v)  }
  while (! Q.isEmpty() )
  {  h = Q.dequeue()
     for hver v ε G.outAdjacentVertices(h)
     {  in(v) = in(v) – 1
        if (in(v)==0) Q.enqueue(v)  }
     R.enqueue(h) 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





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

V.b) DAG - Directed Acyclic Graph



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

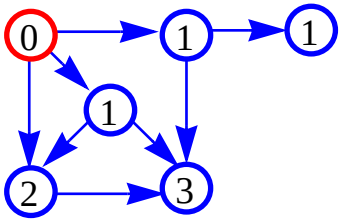
rettet asyklisk graf

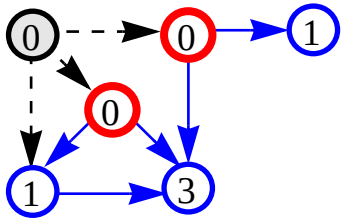
- *arv i et OO-språk*
- *forkrav til kurs*
- *planlegging av avhengige aktiviteter*

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.

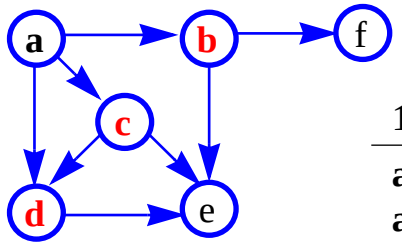
```
Queue TS(Graph G)
  Q, R = empty Queue
  for hver node v ε V
  {  in(v) = G.inDegree(v)
     if (in(v) == 0) Q.enqueue(v)  }
  while (! Q.isEmpty() )
  {  h = Q.dequeue()
     for hver v ε G.outAdjacentVertices(h)
     {  in(v) = in(v) – 1
        if (in(v)==0) Q.enqueue(v)  }
     R.enqueue(h) 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

V.b) DAG - Directed Acyclic Graph



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

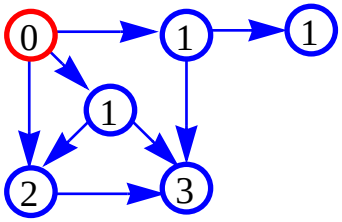
rettet asyklisk graf

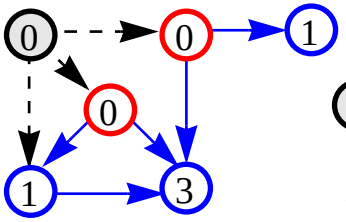
- *arv i et OO-språk*
- *forkrav til kurs*
- *planlegging av avhengige aktiviteter*

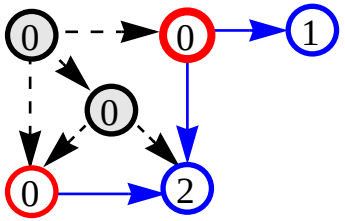
Topologisk ordning av en graf G er en enummerering av noder $v_1,v_2...v_n$ slik at hvis (v_i,v_j) ε E så er i < j. (dermed, hvis det finnes en sti v_i ... v_j så er i < j)

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

```
Queue TS(Graph G)
  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.outAdjacentVertices(h)
     {  in(v) = in(v) – 1
        if (in(v)==0) Q.enqueue(v)  }
     R.enqueue(h)  }
  return R
```

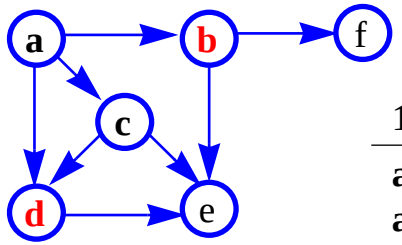






R	a	b	c	d	e	f	Q
	0	1	1	2	3	1	a
a	x	**0**	**0**	**1**	3	1	c b
a	x	0	x	**0**	**2**	1	c b **d**

V.b) DAG - Directed Acyclic Graph



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

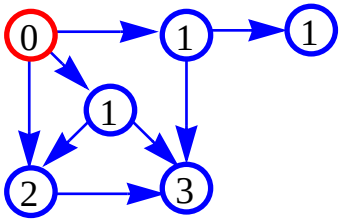
rettet asyklisk graf

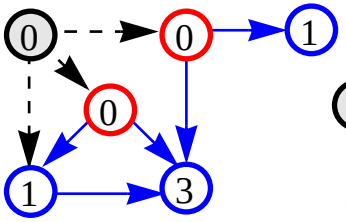
- *arv i et OO-språk*
- *forkrav til kurs*
- *planlegging av avhengige aktiviteter*

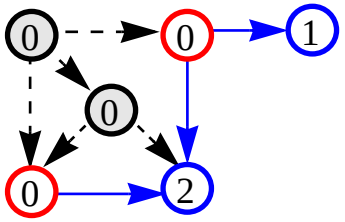
Topologisk ordning av en graf G er en enummerering av noder $v_1,v_2...v_n$ slik at hvis (v_i,v_j) ε E så er i < j. (dermed, hvis det finnes en sti v_i ... v_j så er i < j)

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

```
Queue TS(Graph G)
  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.outAdjacentVertices(h)
     {  in(v) = in(v) – 1
        if (in(v)==0) Q.enqueue(v)  }
     R.enqueue(h)  }
  return R
```

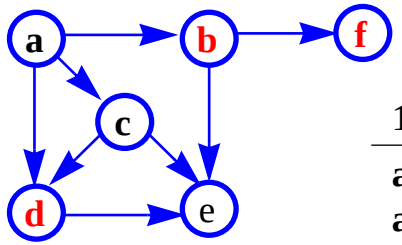






R	a	b	c	d	e	f	Q
	0	1	1	2	3	1	a
a	x	**0**	**0**	**1**	3	1	c b
a **c**	x	0	x	**0**	**2**	1	b d

V.b) DAG - Directed Acyclic Graph



rettet asyklisk graf

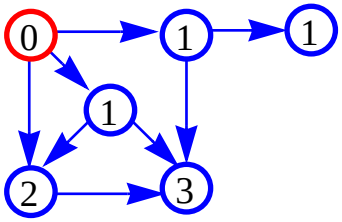
- *arv i et OO-språk*
- *forkrav til kurs*
- *planlegging av avhengige aktiviteter*

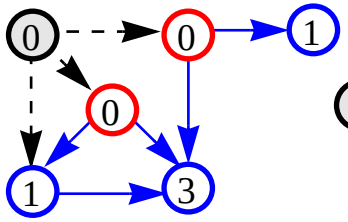
1	2	3	4	5	6
a	**b**	**f**	**c**	**d**	**e**
a	**c**	**d**	**b**	**e**	**f**
....					
a	d	c	e	b	f

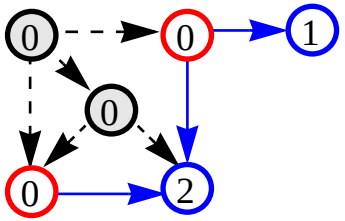
Topologisk ordning av en graf G er en enummerering av noder $v_1,v_2...v_n$ slik at hvis (v_i,v_j) ε E så er i < j. (dermed, hvis det finnes en sti v_i ... v_j så er i < j)

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

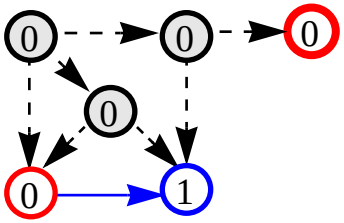
```
Queue TS(Graph G)
  Q, R = empty Queue
  for hver node v ε V
  {  in(v) = G.inDegree(v)
     if (in(v) == 0) Q.enqueue(v)  }
  while (! Q.isEmpty() )
  {  h = Q.dequeue()
     for hver v ε G.outAdjacentVertices(h)
     {  in(v) = in(v) – 1
        if (in(v)==0) Q.enqueue(v)  }
     R.enqueue(h) 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



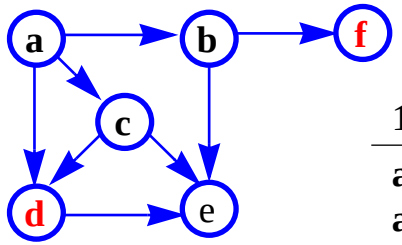




R	a	b	c	d	e	f	Q
	0	1	1	2	3	1	a
a	x	**0**	**0**	**1**	3	1	c b
a c	x	0	x	**0**	**2**	1	b d
a c	x	x	x	0	**1**	**0**	b d **f**



V.b) DAG - Directed Acyclic Graph



rettet asyklisk graf

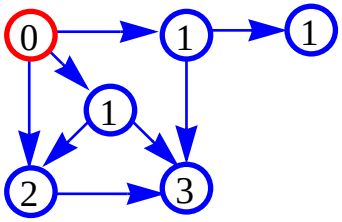
- *arv i et OO-språk*
- *forkrav til kurs*
- *planlegging av avhengige aktiviteter*

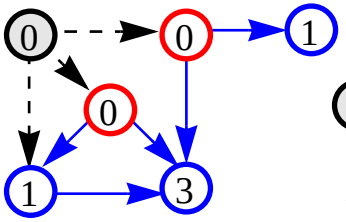
1	2	3	4	5	6
a	**b**	**f**	**c**	**d**	**e**
a	**c**	**d**	**b**	**e**	**f**
....					
a	d	c	e	b	f

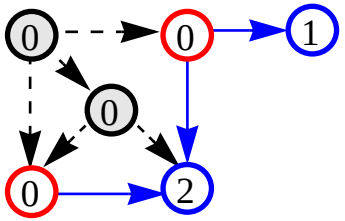
Topologisk ordning av en graf G er en enummerering av noder $v_1,v_2...v_n$ slik at hvis (v_i,v_j) ε E så er i < j. (dermed, hvis det finnes en sti v_i ... v_j så er i < j)

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

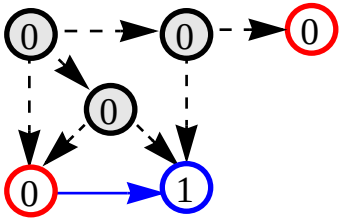
```
Queue TS(Graph G)
  Q, R = empty Queue
  for hver node v ε V
  {  in(v) = G.inDegree(v)
     if (in(v) == 0) Q.enqueue(v)  }
  while (! Q.isEmpty() )
  {  h = Q.dequeue()
     for hver v ε G.outAdjacentVertices(h)
     {  in(v) = in(v) – 1
        if (in(v)==0) Q.enqueue(v)  }
     R.enqueue(h) 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



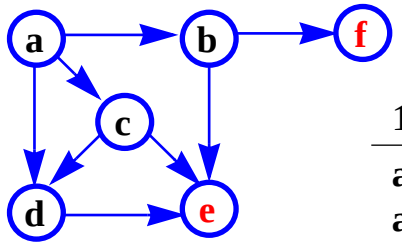




R	a	b	c	d	e	f	Q
	0	1	1	2	3	1	a
a	x	**0**	**0**	**1**	3	1	c b
a c	x	0	x	**0**	**2**	1	b d
a c **b**	x	x	x	0	**1**	**0**	d f



V.b) DAG - Directed Acyclic Graph



rettet asyklisk graf

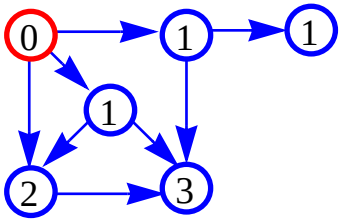
- *arv i et OO-språk*
- *forkrav til kurs*
- *planlegging av avhengige aktiviteter*

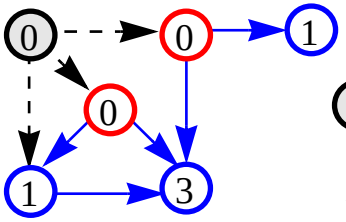
1	2	3	4	5	6
a	**b**	**f**	**c**	**d**	**e**
a	**c**	**d**	**b**	**e**	**f**
....					
a	d	c	e	b	f

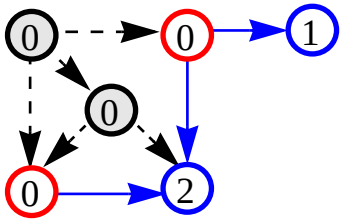
Topologisk ordning av en graf G er en enummerering av noder $v_1,v_2...v_n$ slik at hvis (v_i,v_j) ε E så er i < j. (dermed, hvis det finnes en sti v_i ... v_j så er i < j)

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

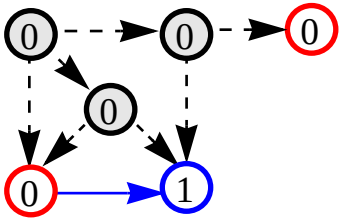
```
Queue TS(Graph G)
  Q, R = empty Queue
  for hver node v ε V
  {  in(v) = G.inDegree(v)
     if (in(v) == 0) Q.enqueue(v)  }
  while (! Q.isEmpty() )
  {  h = Q.dequeue()
     for hver v ε G.outAdjacentVertices(h)
     {  in(v) = in(v) – 1
        if (in(v)==0) Q.enqueue(v)  }
     R.enqueue(h) 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

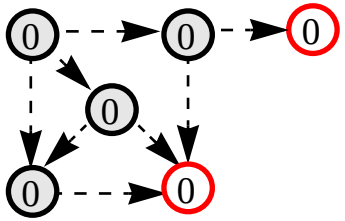




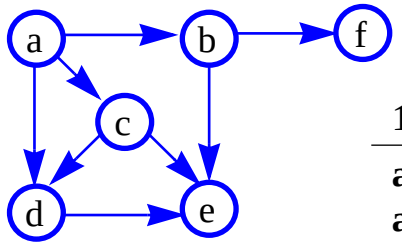


R	a	b	c	d	e	f	Q
	0	1	1	2	3	1	a
a	x	**0**	**0**	**1**	3	1	c b
a c	x	0	x	**0**	**2**	1	b d
a c b	x	x	x	0	**1**	**0**	d f
a c b d	x	x	x	x	**0**	0	f e
a c b d f e							





V.b) DAG - Directed Acyclic Graph



rettet asyklisk graf

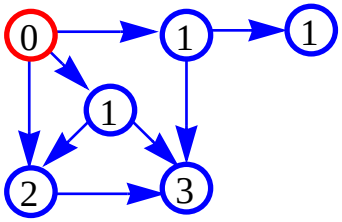
- *arv i et OO-språk*
- *forkrav til kurs*
- *planlegging av avhengige aktiviteter*

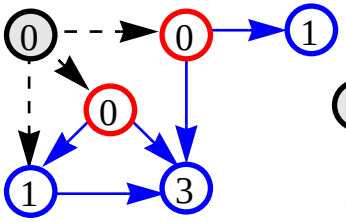
1	2	3	4	5	6
a	**b**	**f**	**c**	**d**	**e**
a	**c**	**d**	**b**	**e**	**f**
....					
a	d	c	e	b	f

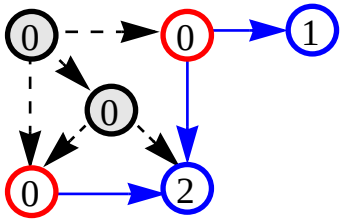
Topologisk ordning av en graf G er en enummerering av noder $v_1,v_2...v_n$ slik at hvis (v_i,v_j) ε E så er i < j. (dermed, hvis det finnes en sti v_i ... v_j så er i < j)

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

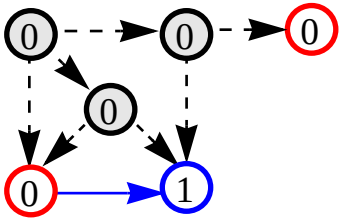
```
Queue TS(Graph G)                          O(nk)
  Q, R = empty Queue                       O(n+k)
  for hver node v ε V                      O(n²)
  {  in(v) = G.inDegree(v)
     if (in(v) == 0) Q.enqueue(v)  }
  while (! Q.isEmpty() )
  {  h = Q.dequeue()
     for hver v ε G.outAdjacentVertices(h)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

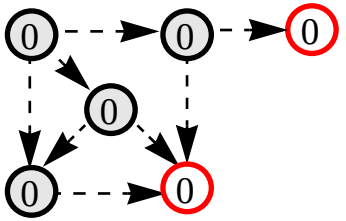




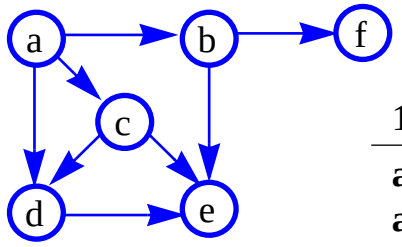


R	a	b	c	d	e	f	Q
	0	1	1	2	3	1	a
a	x	**0**	**0**	**1**	3	1	c b
a c	x	0	x	**0**	**2**	1	b d
a c b	x	x	x	0	**1**	**0**	d f
a c b d	x	x	x	x	**0**	0	f e
a c b d f e							





V.b) DAG - Directed Acyclic Graph



rettet asyklisk graf

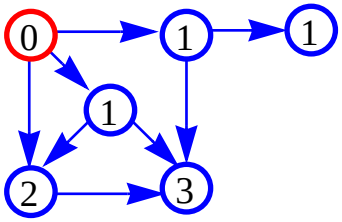
- *arv i et OO-språk*
- *forkrav til kurs*
- *planlegging av avhengige aktiviteter*

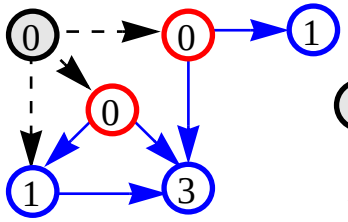
1	2	3	4	5	6
a	**b**	**f**	**c**	**d**	**e**
a	**c**	**d**	**b**	**e**	**f**
....					
a	d	c	e	b	f

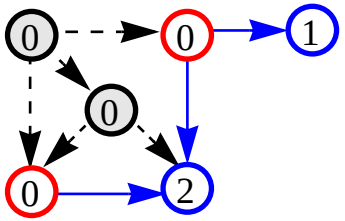
Topologisk ordning av en graf G er en enummerering av noder $v_1,v_2...v_n$ slik at hvis (v_i,v_j) ε E så er i < j. (dermed, hvis det finnes en sti v_i ... v_j så er i < j)

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

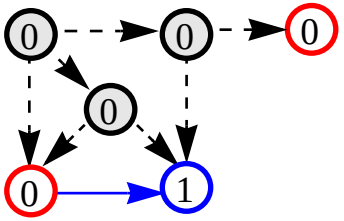
```
Queue TS(Graph G)                          O(nk)
  Q, R = empty Queue                       O(n+k)
  for hver node v ε V                      O(n^2)
  {  in(v) = G.inDegree(v)
     if (in(v) == 0) Q.enqueue(v)  }
  while (! Q.isEmpty() )
  {  h = Q.dequeue()
     for hver v ε G.outAdjacentVertices(h)
     {  in(v) = in(v) – 1
        if (in(v)==0) Q.enqueue(v)  }
     R.enqueue(h)  }
  return R
```

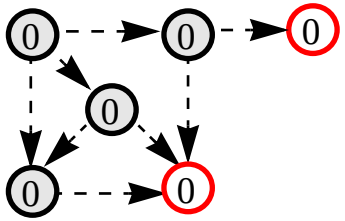






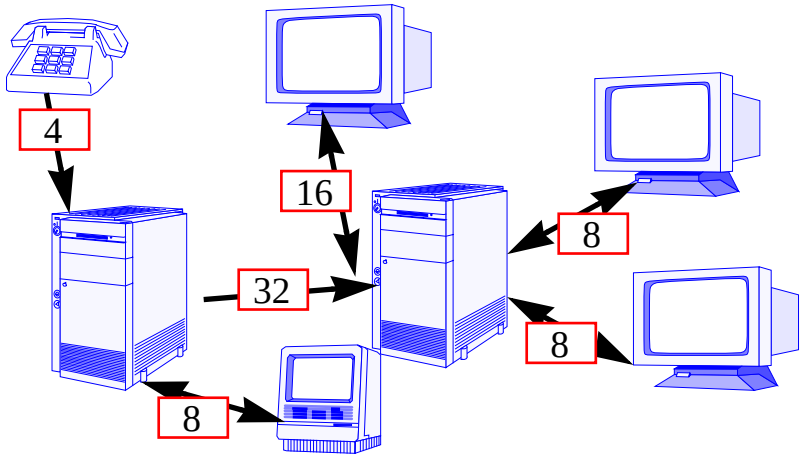
R	a	b	c	d	e	f	Q
	0	1	1	2	3	1	a
a	x	**0**	**0**	**1**	3	1	c b
a c	x	0	x	**0**	**2**	1	b d
a c b	x	x	x	0	**1**	**0**	d f
a c b d	x	x	x	x	**0**	0	f e
a c b d f e							



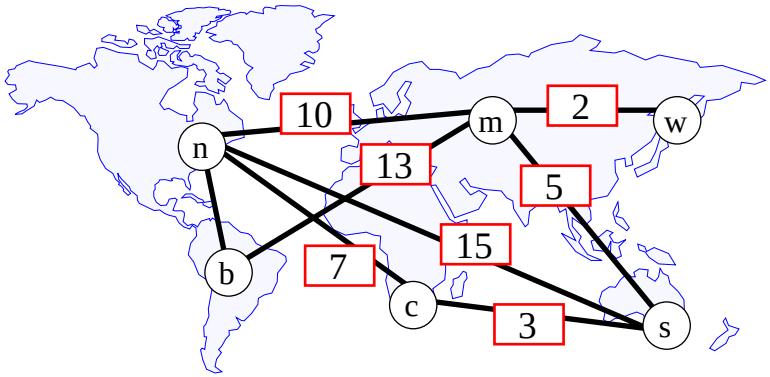


12.22 Har grafen en rettet sykel vil TS oppdage dette. Hvordan?.
-> TS gir en O(n+k) algoritme som sjekker om en rettet graf er asyklisk

NETTVERK KAPASITET



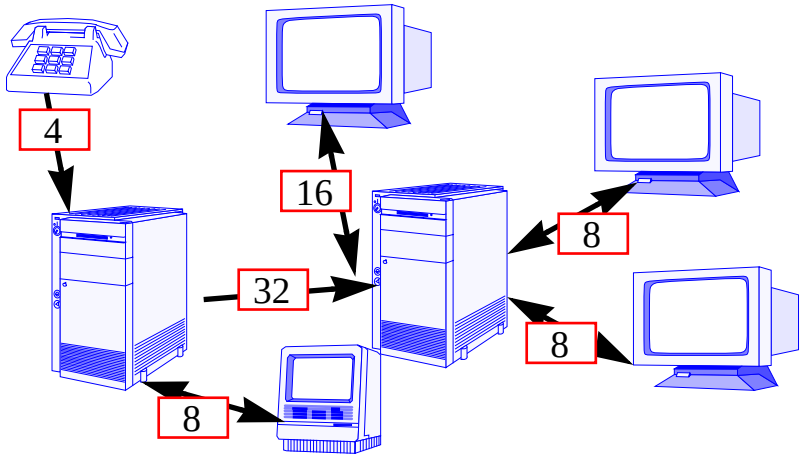
AVSTAND



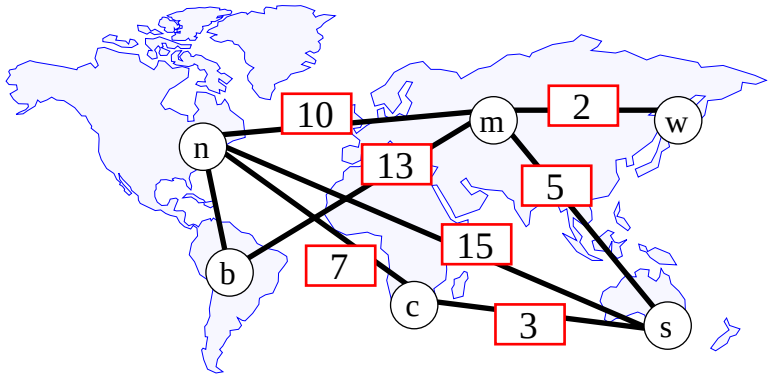
VI. Vektede Grafer

en graf der hver kant har et vekt-attributt

NETTVERK KAPASITET



AVSTAND

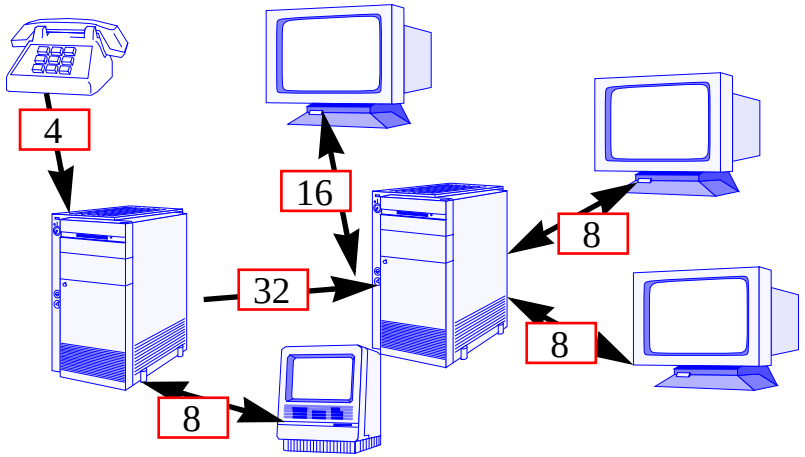


VI. Vektede Grafer

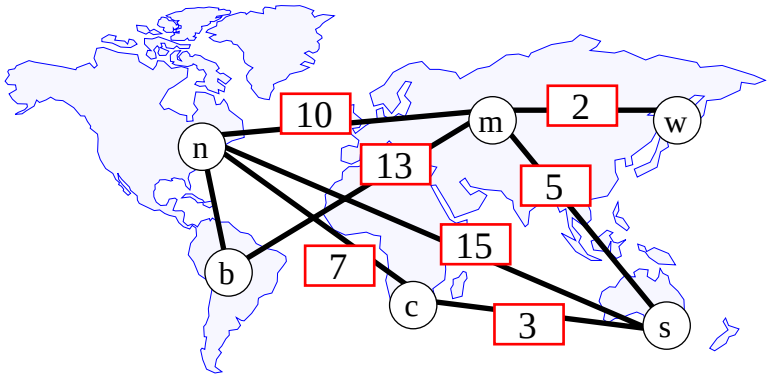
en graf der hver kant har et vekt-attributt

- 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



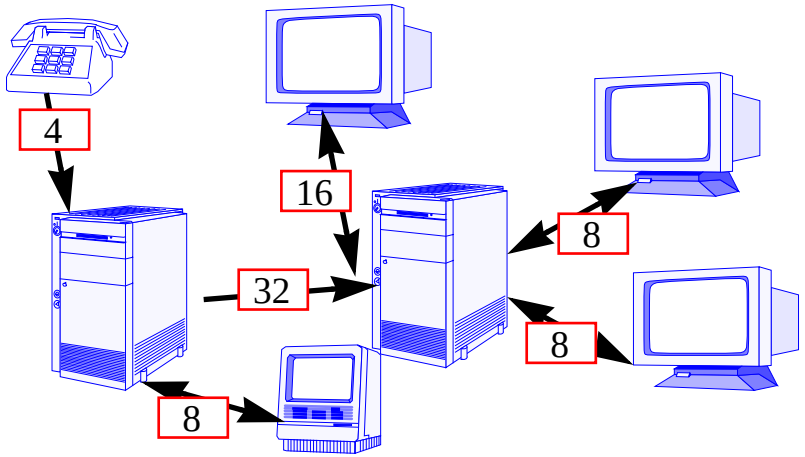
- Implementasjon bruker det faktum at 'Edge implements Position' – kanter lagrer objekter med vekt

VI. Vektede Grafer

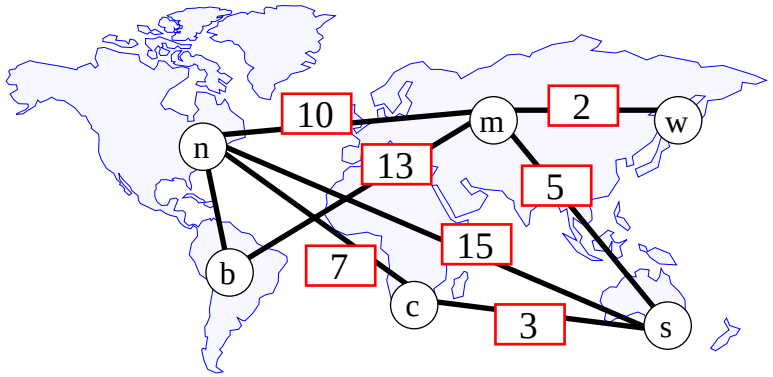
en graf der hver kant har et vekt-attributt

- 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

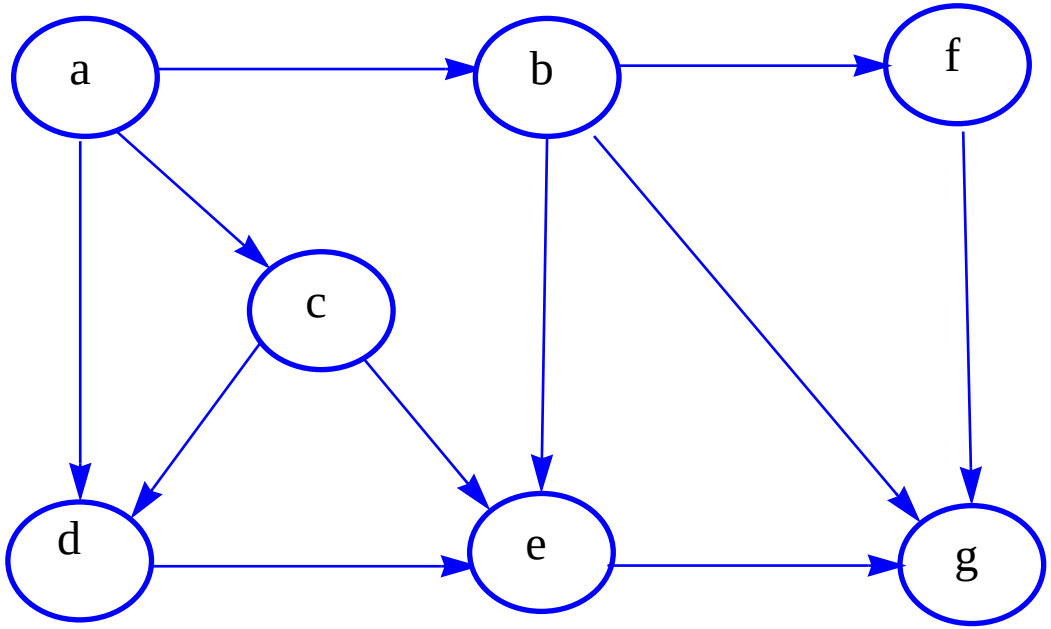
- I tillegg til vanlige graf-problemer, spør man i forbindelse med vektede grafer for eksempel om
  - *hva er **korteste** sti fra u til v, dvs. sti fra u til v med minste sum av vekter?*
  - *hva er **minste** utspennende tre, dvs. hvor sum av vekter på kanter i treet er minst?*
  - *..... minste /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

Korteste sti : ikke-vektet graf

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
    for each kant (u,v)
        if ( D(u) + 1 < D(v) )   D(v) = D(u) + 1
```

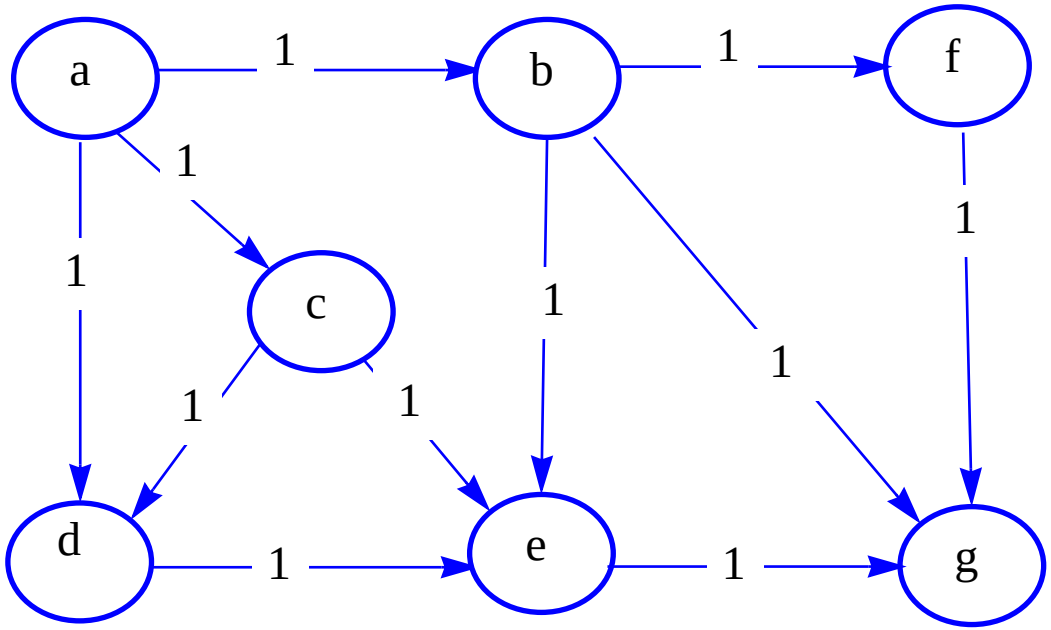


Korteste sti : ikke-vektet graf

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
    for each kant (u,v)
        if ( D(u) + 1 < D(v) )   D(v) = D(u) + 1
```

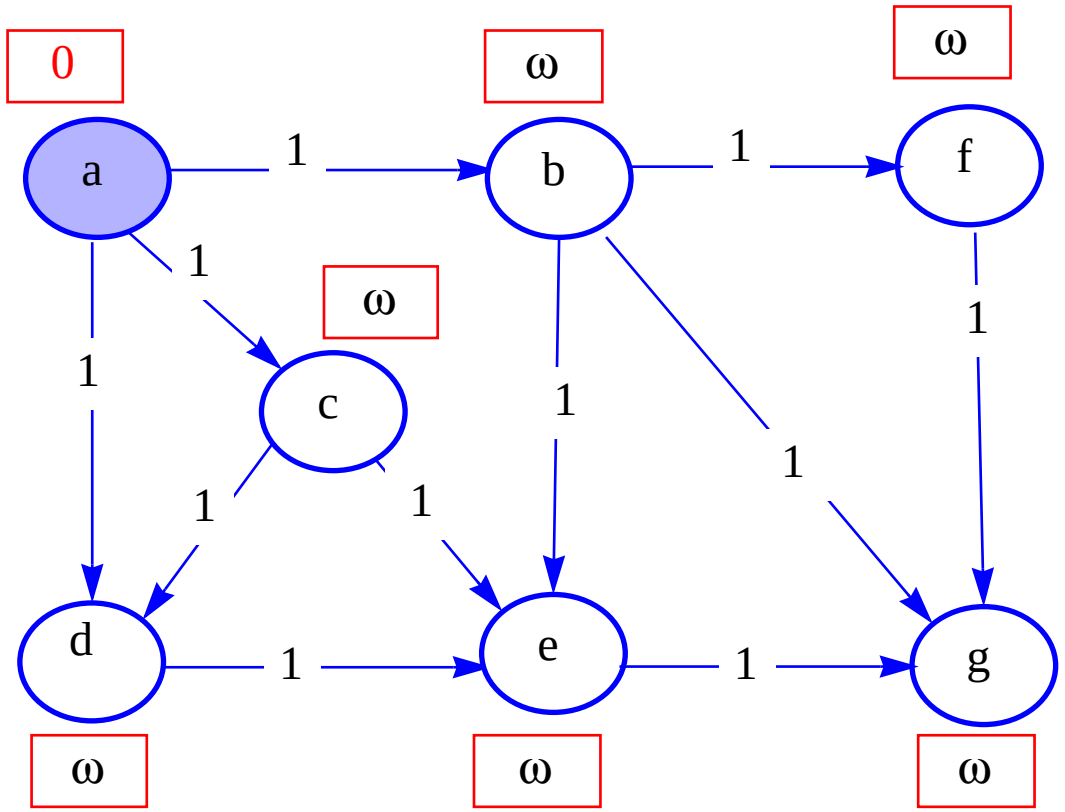


Korteste sti : ikke-vektet graf

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
    for each kant (u,v)
        if ( D(u) + 1 < D(v) )   D(v) = D(u) + 1
```

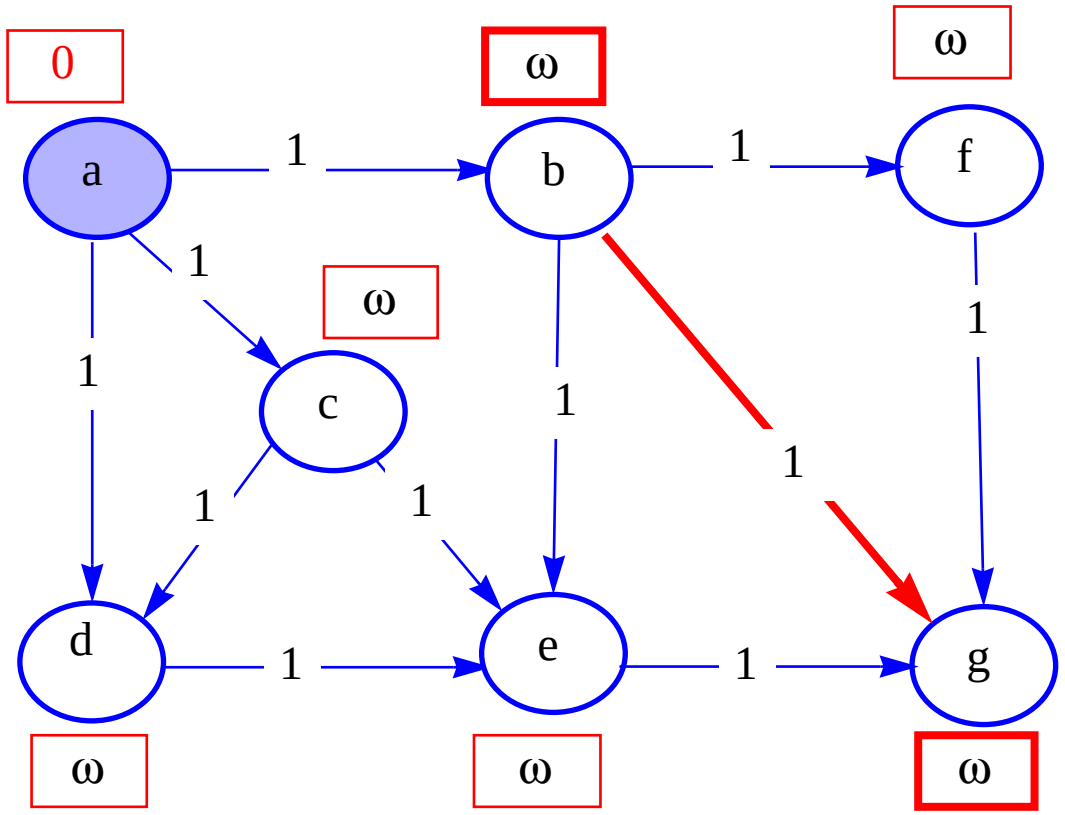


Korteste sti : ikke-vektet graf

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
    for each kant (u,v)
        if ( D(u) + 1 < D(v) )   D(v) = D(u) + 1
```

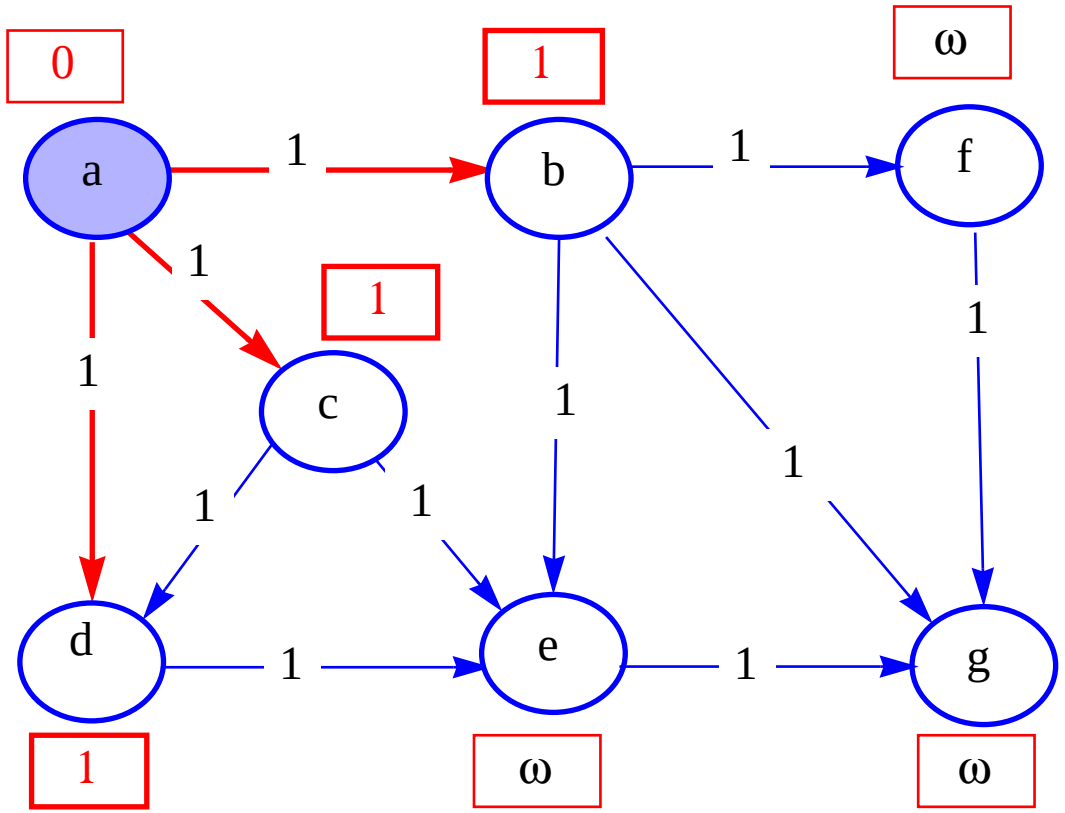


Korteste sti : ikke-vektet graf

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
    for each kant (u,v)
        if ( D(u) + 1 < D(v) )   D(v) = D(u) + 1
```

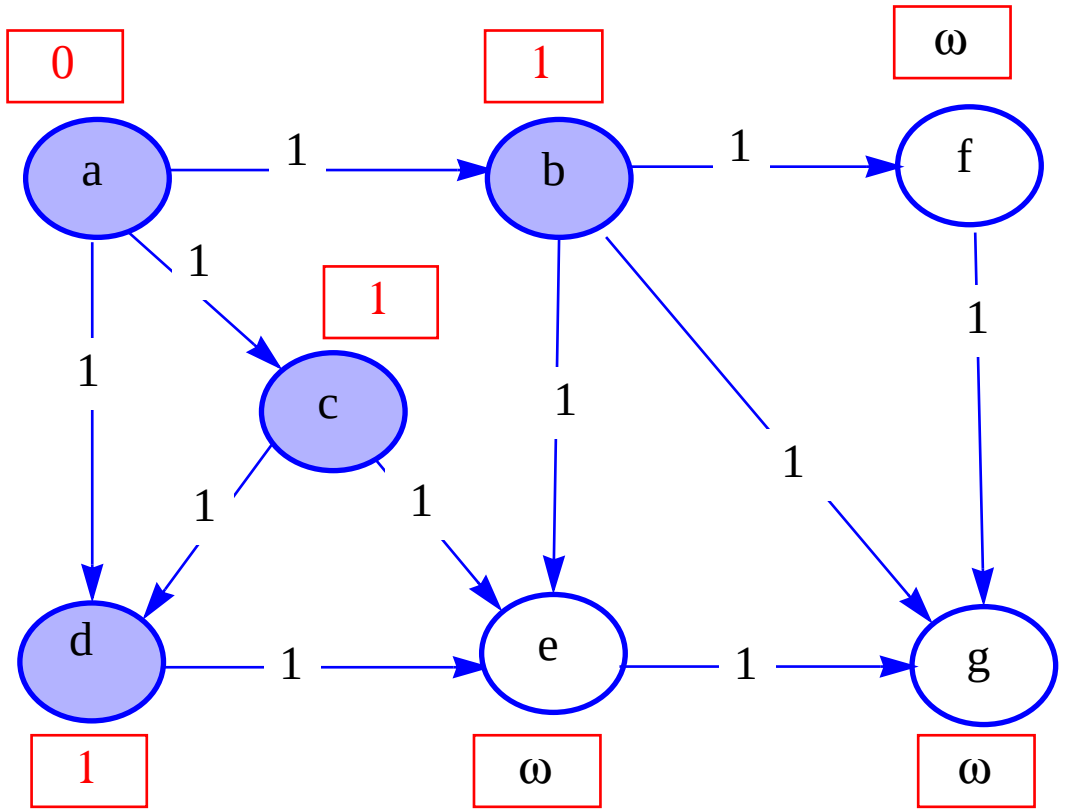


Korteste sti : ikke-vektet graf

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
    for each kant (u,v)
        if ( D(u) + 1 < D(v) )   D(v) = D(u) + 1
```

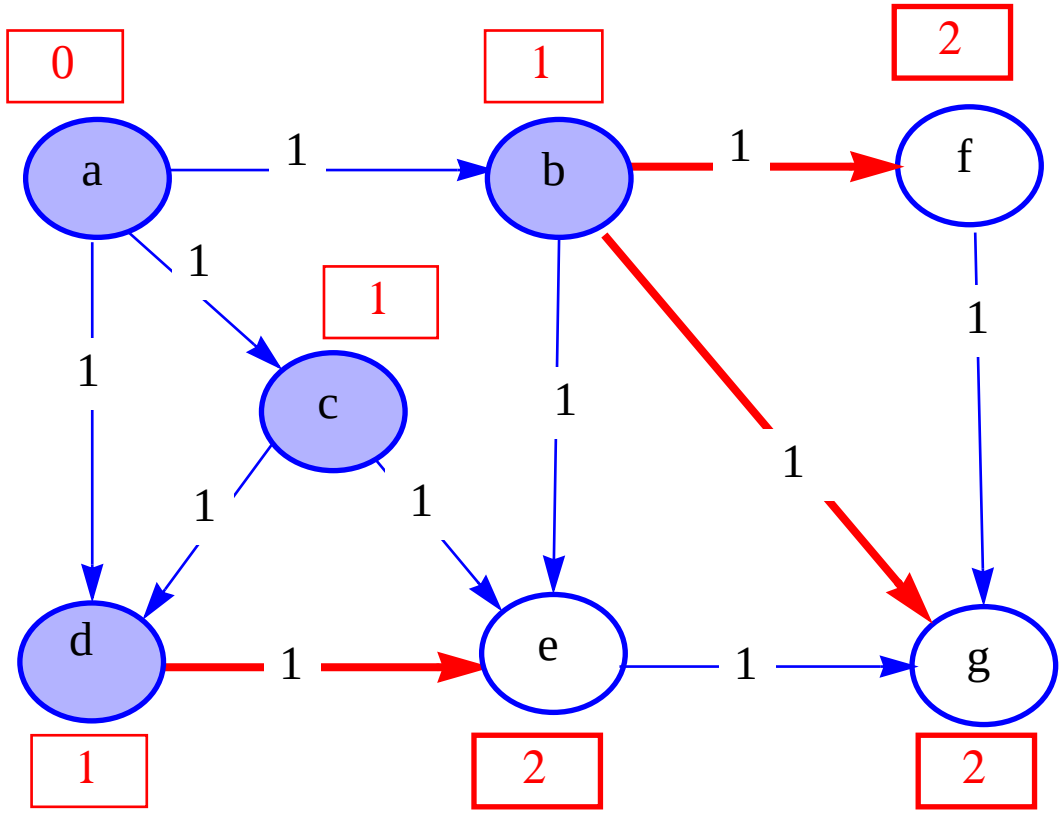


Korteste sti : ikke-vektet graf

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
    for each kant (u,v)
        if ( D(u) + 1 < D(v) )   D(v) = D(u) + 1
```

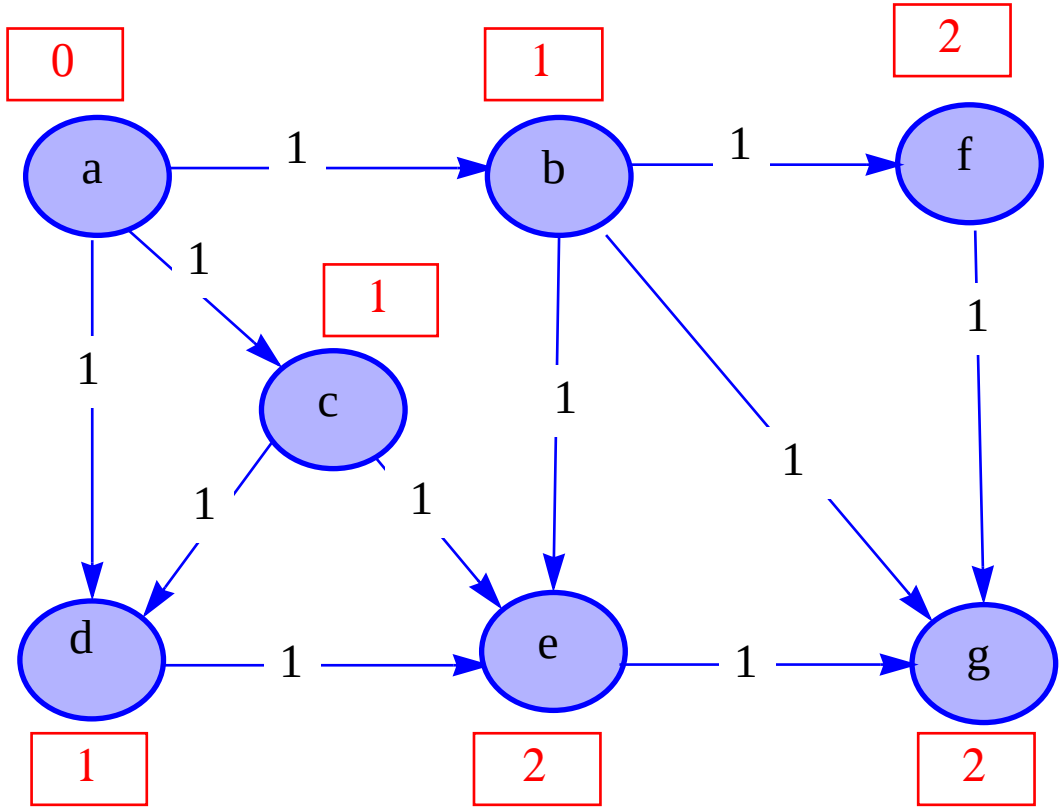


Korteste sti : ikke-vektet graf

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
    for each kant (u,v)
        if ( D(u) + 1 < D(v) )   D(v) = D(u) + 1
```

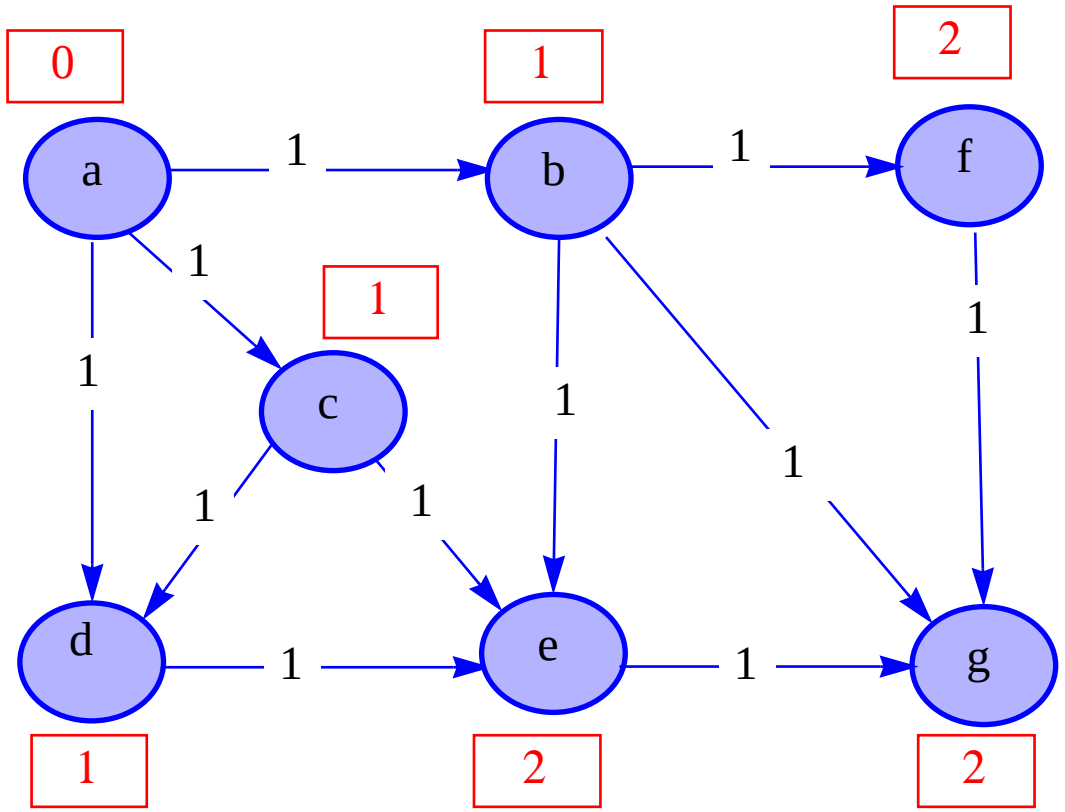


Korteste sti : ikke-vektet graf

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*k)
    for each kant (u,v)
        if ( D(u) + 1 < D(v) )   D(v) = D(u) + 1
```

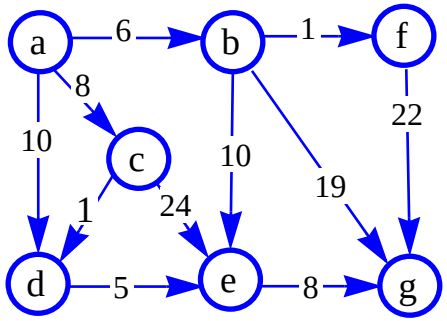


Korteste sti (single-source shortest-paths) :

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*k)
    for each kant (u,v)
        if ( D(u) + vekt(u,v) < D(v) )   D(v) = D(u) + vekt(u,v)
```

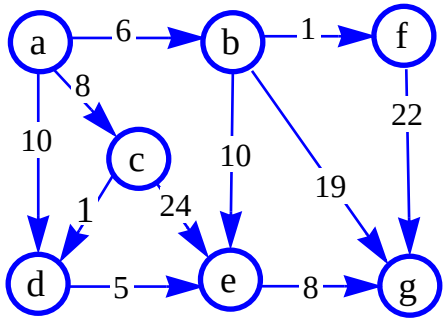


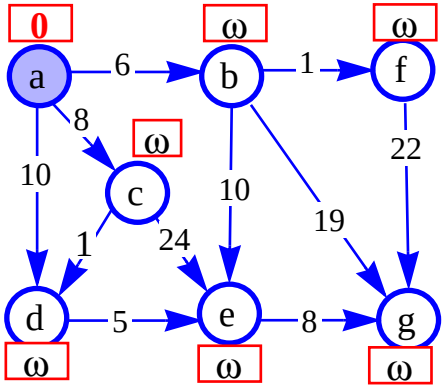
Korteste sti (single-source shortest-paths) :

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*k)
    for each kant (u,v)
        if ( D(u) + vekt(u,v) < D(v) )   D(v) = D(u) + vekt(u,v)
```



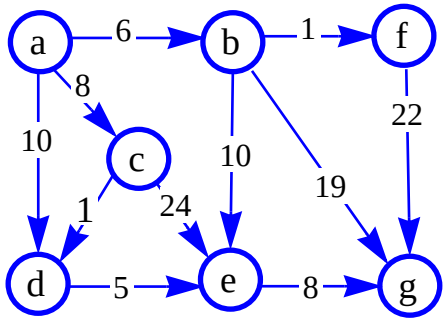


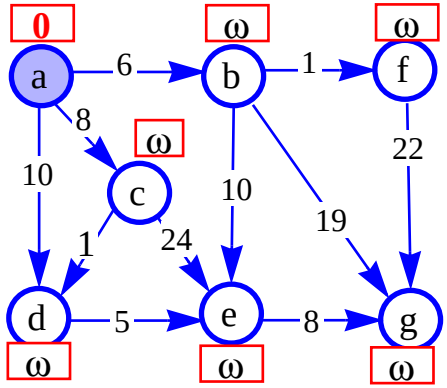
Korteste sti (single-source shortest-paths) :

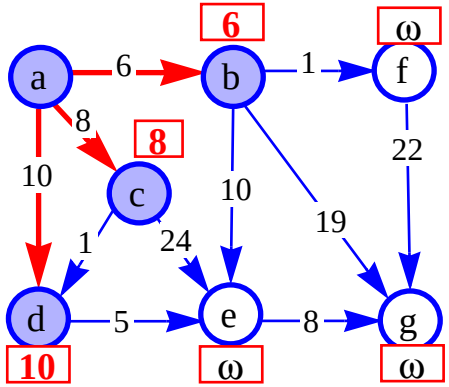
Finn (lengden av) korteste sti fra a til en bestemt/alle andre node(r)

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*k)
    for each kant (u,v)
        if ( D(u) + vekt(u,v) < D(v) )   D(v) = D(u) + vekt(u,v)
```





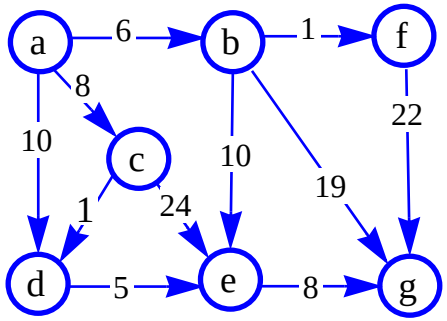


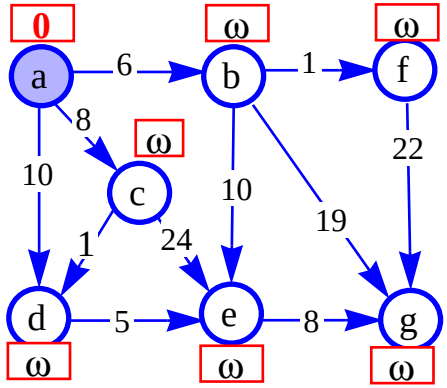
Korteste sti (single-source shortest-paths) :

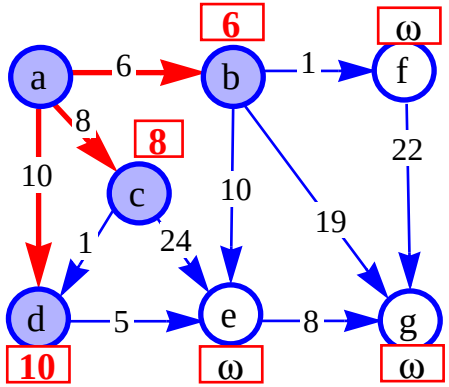
Finn (lengden av) korteste sti fra a til en bestemt/alle andre node(r)

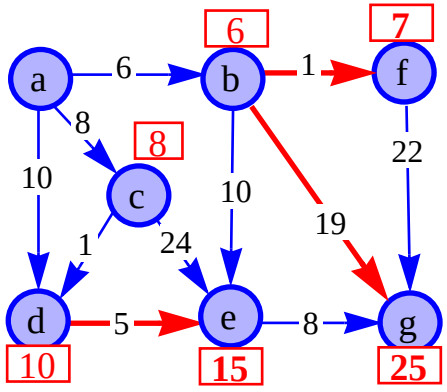
Ford-Bellman **SS-SP** :

```
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*k)
    for each kant (u,v)
        if ( D(u) + vekt(u,v) < D(v) )   D(v) = D(u) + vekt(u,v)
```







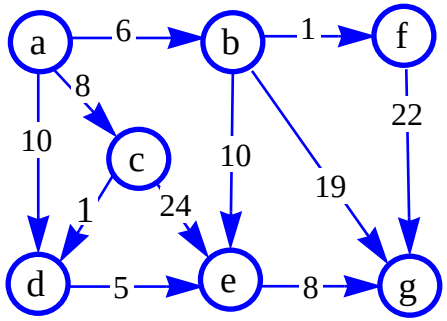


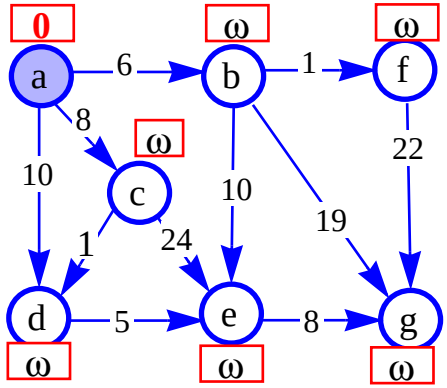
Korteste sti (single-source shortest-paths) :

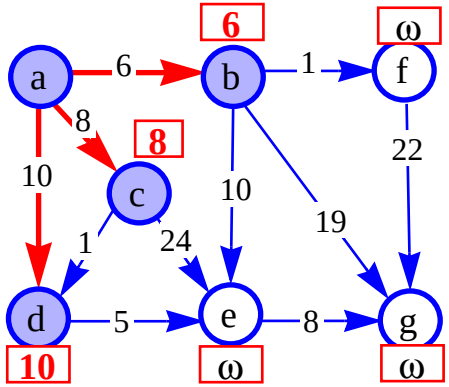
Finn (lengden av) korteste sti fra a til en bestemt/alle andre node(r)

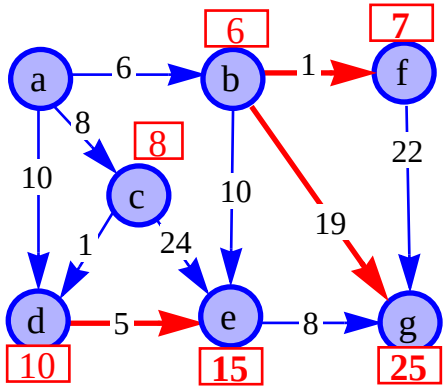
Ford-Bellman **SS-SP** :

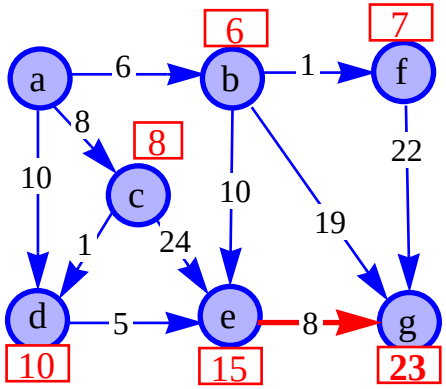
```
for each node v : D(v)= ω        // ω er et maksimalt tall (her ω > n)
D(s) = 0;                        // s er startnoden
for (i=1; i<n; i++)              // n antall noder, m antall kanter      O(n*k)
    for each kant (u,v)
        if ( D(u) + vekt(u,v) < D(v) )   D(v) = D(u) + vekt(u,v)
```









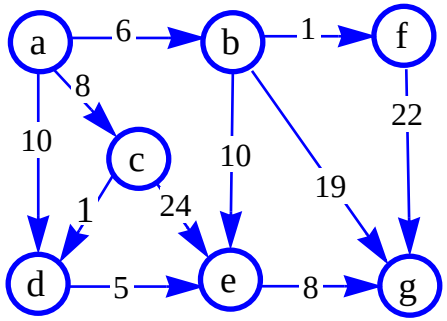


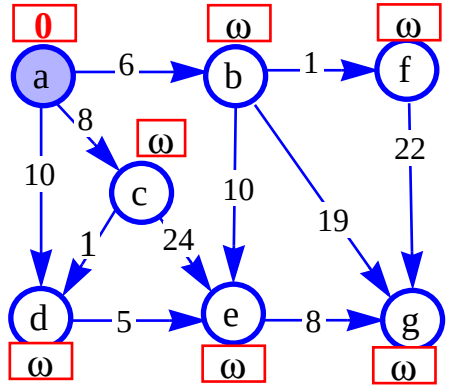
Korteste sti (single-source shortest-paths) :

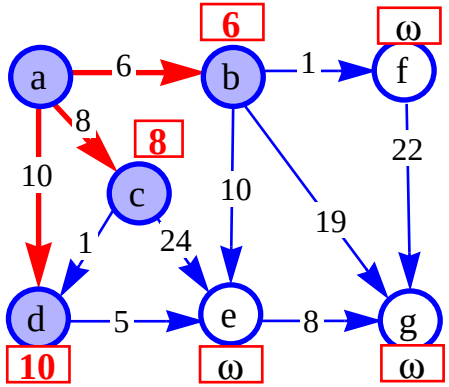
Finn (lengden av) korteste sti fra a til en bestemt/alle andre node(r)

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*k)
    for each kant (u,v)
        if ( D(u) + vekt(u,v) < D(v) )   D(v) = D(u) + vekt(u,v)
```



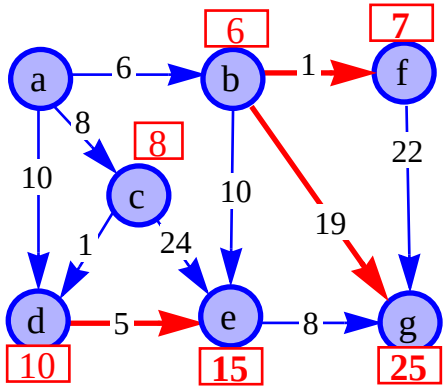


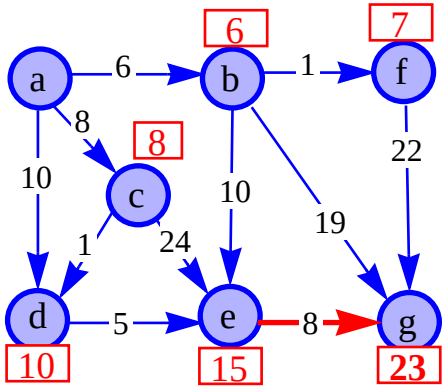


graf	vekter
ikke-rettet	1
rettet	vilkårlige

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

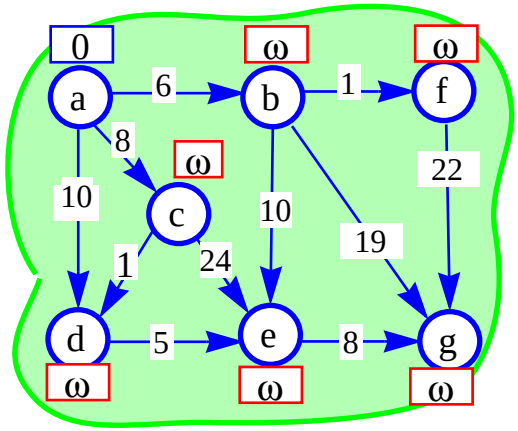
- hvis det finnes en kant (u,v) med
  D(u) + vekt(u,v) < D(v),
  så har G en negativ sykel
- ellers er D(v) korrekt for alle noder v





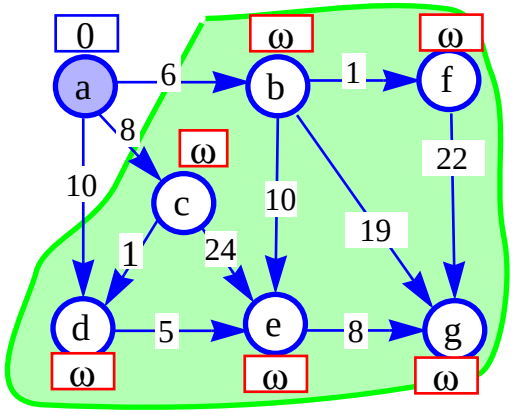
Korteste sti (SS-SP) : Dijkstra algoritme (vekt(e) ≥ 0)

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



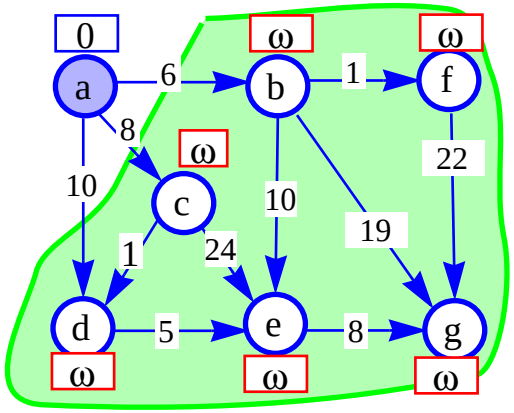
Korteste sti (SS-SP) : Dijkstra algoritme (vekt(e) ≥ 0)

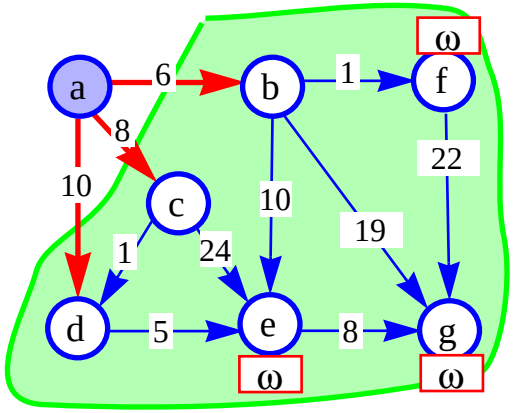
initialiser D(a)=0 og D(v)=ω for alle andre v
sett alle noder i en PriorityQueue Q mht. D
while (! Q.isEmpty())
  v = Q.removeMinElement() // Greedy
  for hver z ε G.outAdjacentVertices(v)
    if (D(v)+vekt(v,z) < D(z))
      D(z)= D(v) + vekt(v,z)
      oppdater Q // z kan få ny nøkkel



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

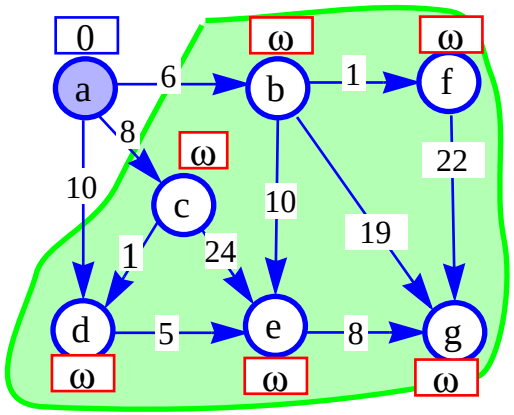
initialiser D(a)=0 og D(v)=ω for alle andre v
sett alle noder i en PriorityQueue Q mht. D
while (! Q.isEmpty())
  v = Q.removeMinElement() // Greedy
  for hver z ε G.outAdjacentVertices(v)
    if (D(v)+vekt(v,z) < D(z))
      D(z)= D(v) + vekt(v,z)
      oppdater Q // z kan få ny nøkkel

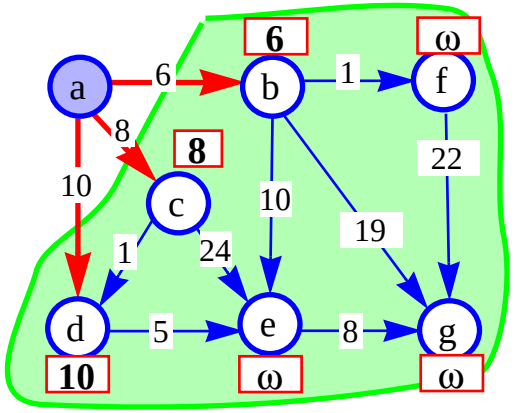




Korteste sti (SS-SP) : Dijkstra algoritme (vekt(e) ≥ 0)

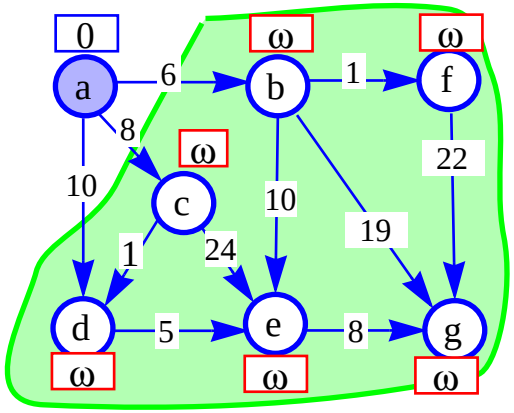
initialiser D(a)=0 og D(v)=ω for alle andre v
sett alle noder i en PriorityQueue Q mht. D
while (! Q.isEmpty())
  v = Q.removeMinElement() // Greedy
  for hver z ε G.outAdjacentVertices(v)
    if (D(v)+vekt(v,z) < D(z))
      D(z)= D(v) + vekt(v,z)
      oppdater Q // z kan få ny nøkkel

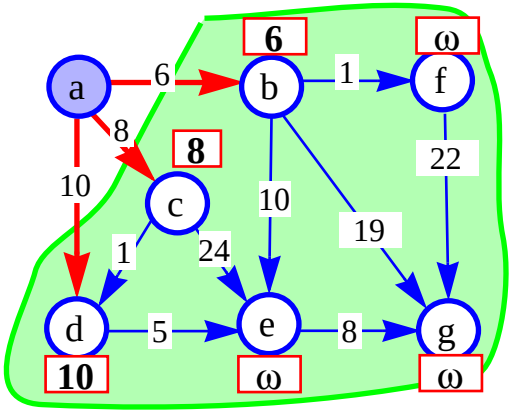


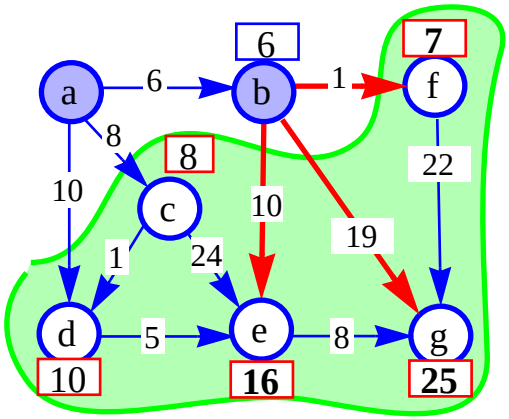


Korteste sti (SS-SP) : Dijkstra algoritme (vekt(e) ≥ 0)

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

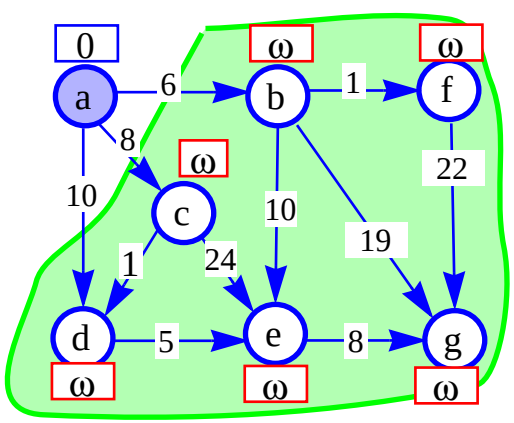


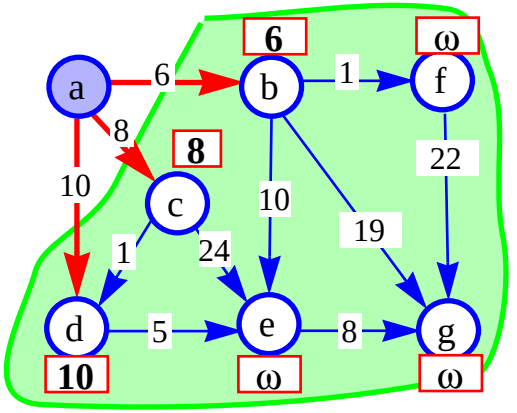


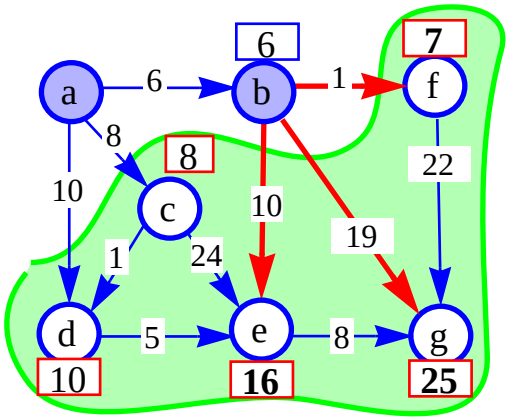


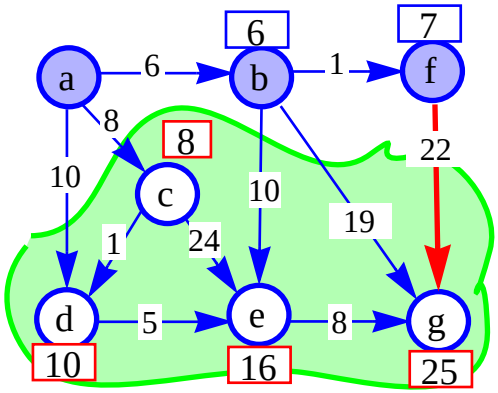
Korteste sti (SS-SP) : Dijkstra algoritme (vekt(e) ≥ 0)

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



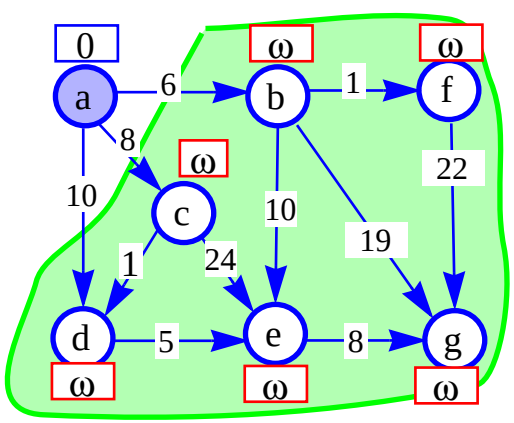


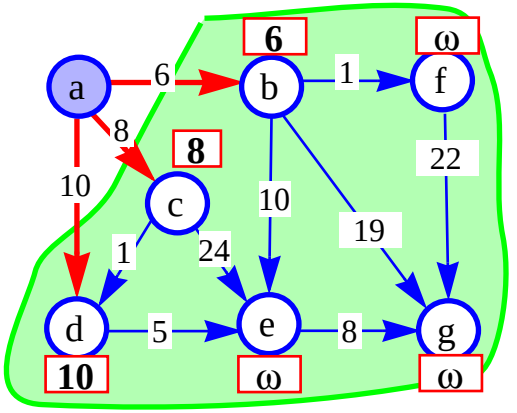


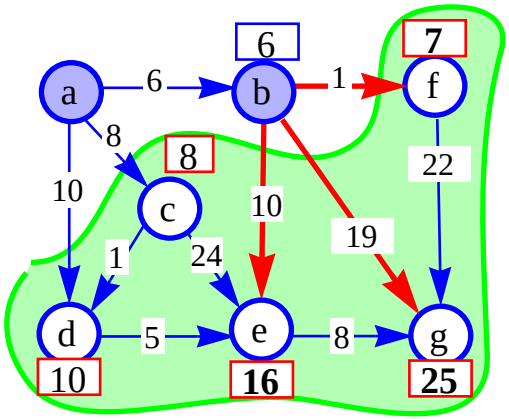


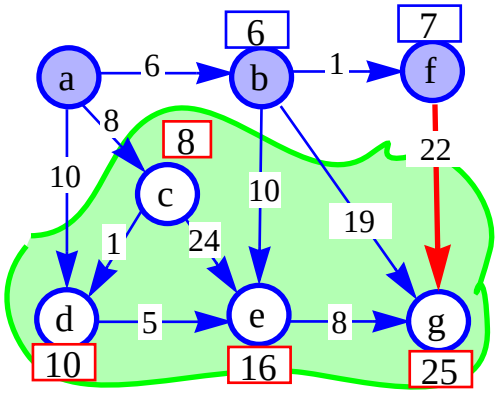
Korteste sti (SS-SP) : Dijkstra algoritme (vekt(e) ≥ 0)

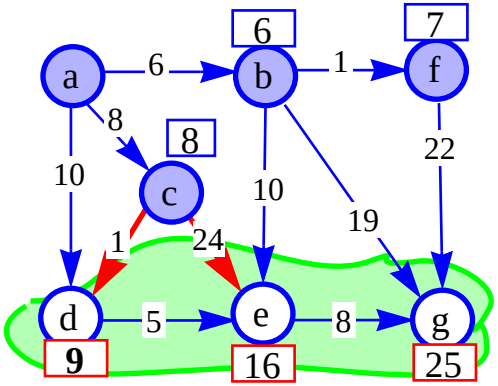
initialiser D(a)=0 og D(v)=ω for alle andre v
sett alle noder i en PriorityQueue Q mht. D
while (! Q.isEmpty())
  v = Q.removeMinElement() // Greedy
  for hver z ε G.outAdjacentVertices(v)
    if (D(v)+vekt(v,z) < D(z))
      D(z)= D(v) + vekt(v,z)
      oppdater Q // z kan få ny nøkkel





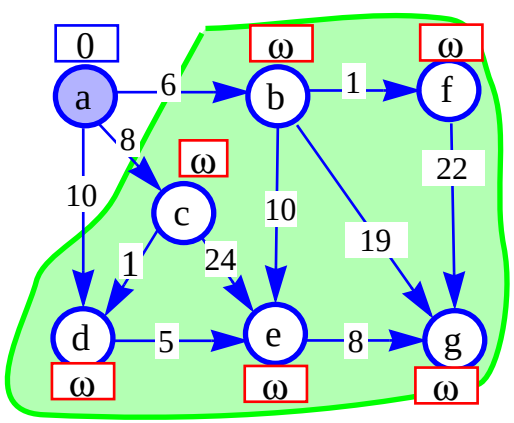


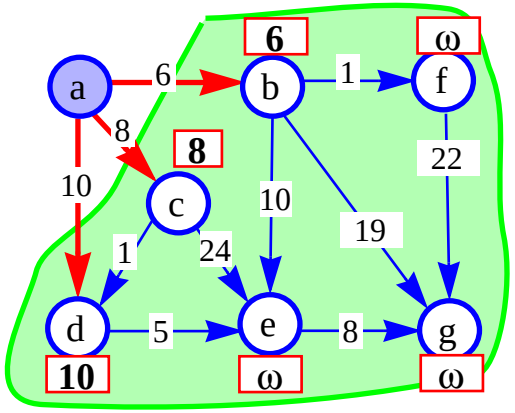


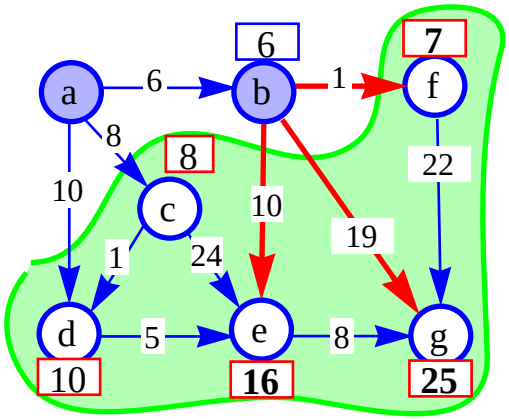


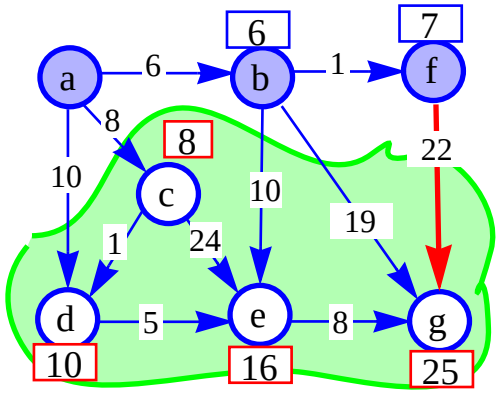
Korteste sti (SS-SP) : Dijkstra algoritme (vekt(e) ≥ 0)

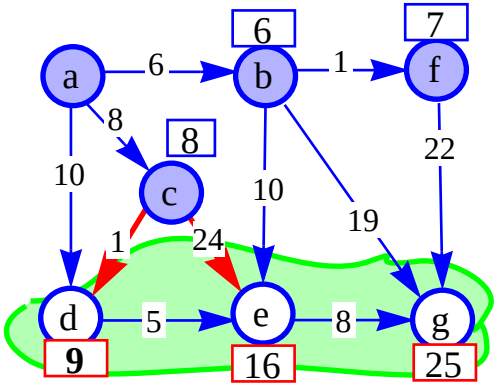
initialiser D(a)=0 og D(v)=ω for alle andre v
sett alle noder i en PriorityQueue Q mht. D
while (! Q.isEmpty())
  v = Q.removeMinElement() // Greedy
  for hver z ε G.outAdjacentVertices(v)
    if (D(v)+vekt(v,z) < D(z))
      D(z)= D(v) + vekt(v,z)
      oppdater Q // z kan få ny nøkkel

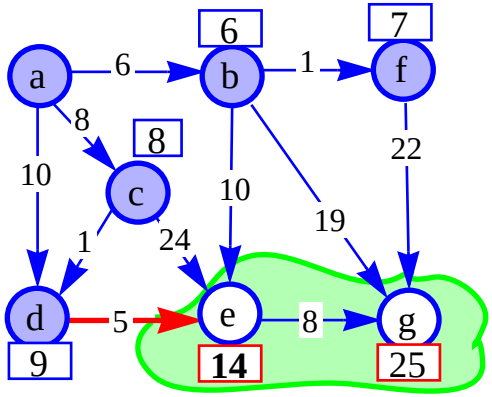






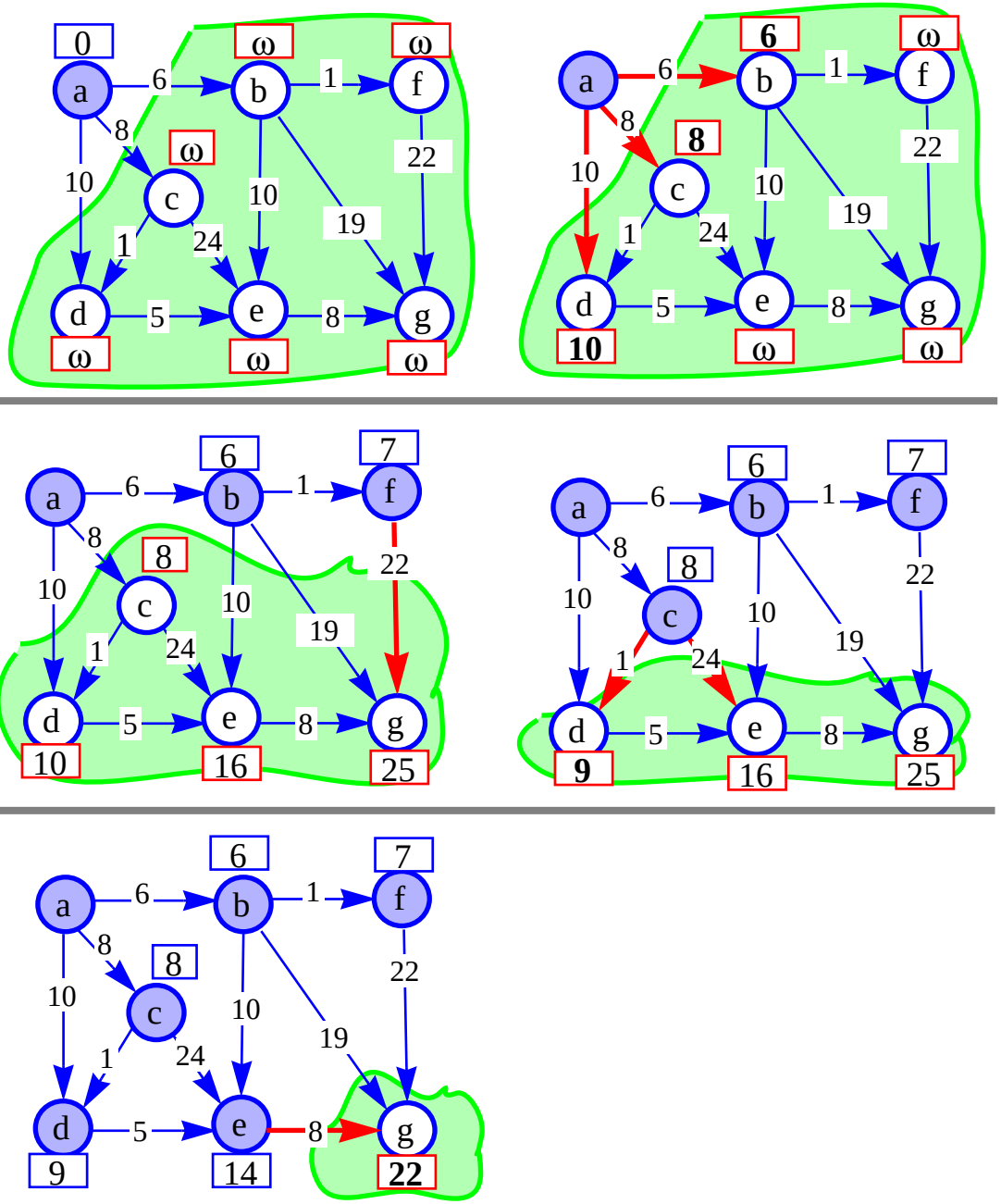






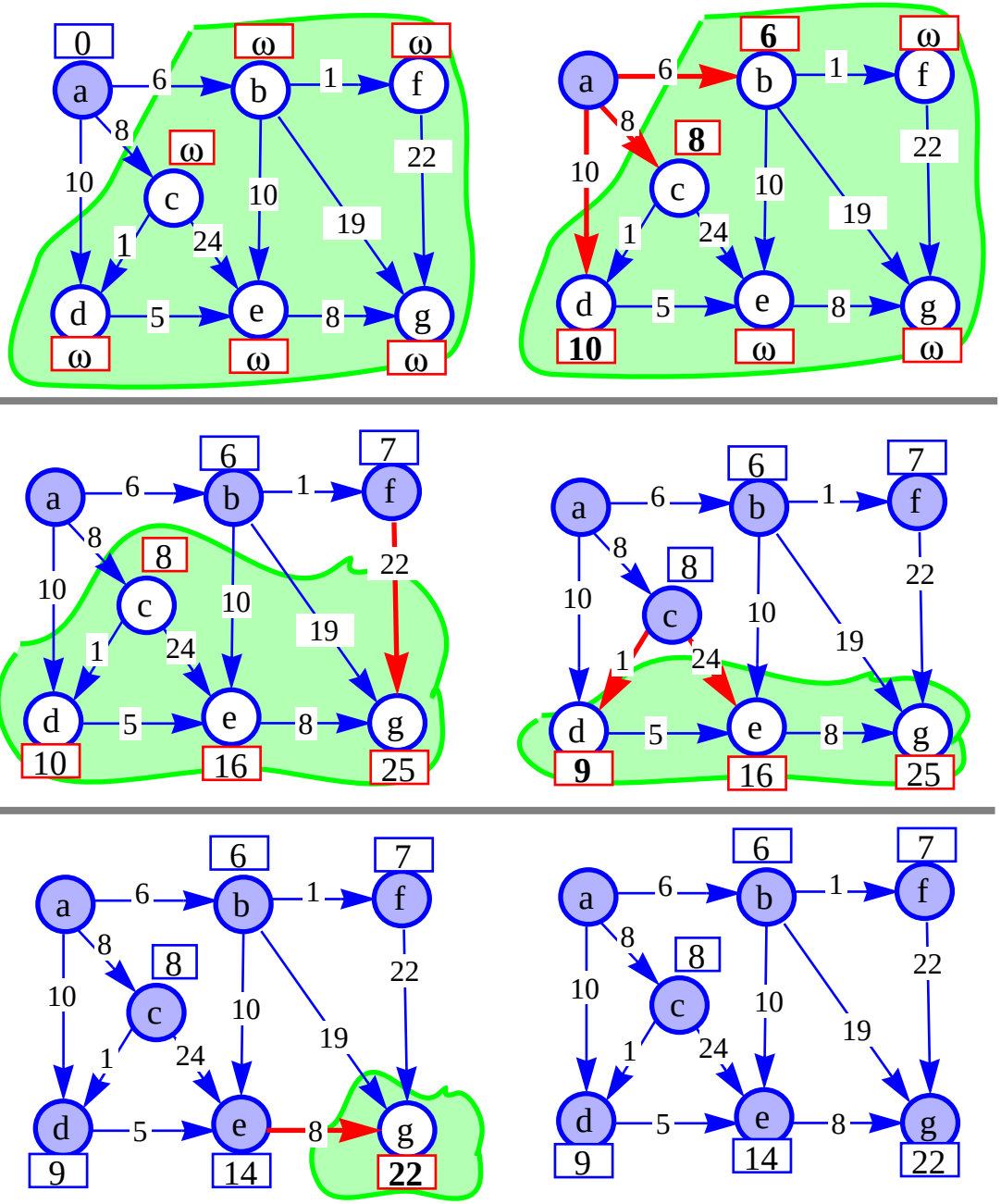
Korteste sti (SS-SP) : Dijkstra algoritme (vekt(e) ≥ 0)

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



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())
4. v = Q.**removeMinElement**()
5. for hver z ε G.**outAdjacentVertices**(v)
6. if (D(v)+vekt(v,z) < D(z))
7. Q.**replaceKey**(z, D(v) + vekt(v,z)) ;

Løkke Invariant :

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())
4. v = Q.**removeMinElement**()
5. for hver z ε G.**outAdjacentVertices**(v)
6. if (D(v)+vekt(v,z) < D(z))
7. Q.**replaceKey**(z, D(v) + vekt(v,z)) ;

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

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())
4. v = Q.**removeMinElement**()
5. for hver z ε G.**outAdjacentVertices**(v)
6. if (D(v)+vekt(v,z) < D(z))
7. Q.**replaceKey**(z, D(v) + vekt(v,z)) ;

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.

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())
4. v = Q.**removeMinElement**()
5. for hver z ε G.**outAdjacentVertices**(v)
6. if (D(v)+vekt(v,z) < D(z))
7. Q.**replaceKey**(z, D(v) + vekt(v,z)) ;

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

– holder før inngangen siden ingen node ble fjernet.

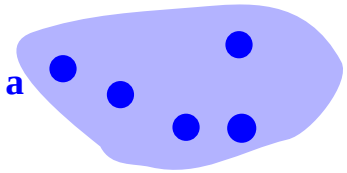
– for å vise at den opprettholdes i løkken, må vise at den holder etter 4.

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.

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())
4. v = Q.**removeMinElement**()
5. for hver z ε G.**outAdjacentVertices**(v)
6. if (D(v)+vekt(v,z) < D(z))
7. Q.**replaceKey**(z, D(v) + vekt(v,z)) ;



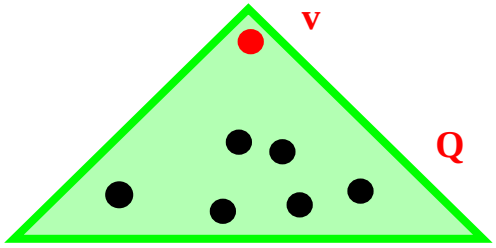
Løkke Invariant : **alle noder x som har blitt fjernet fra Q har korrekt D(x)**

– 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.



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())
4. v = Q.**removeMinElement**()
5. for hver z ε G.**outAdjacentVertices**(v)
6. if (D(v)+vekt(v,z) < D(z))
7. Q.**replaceKey**(z, D(v) + vekt(v,z)) ;

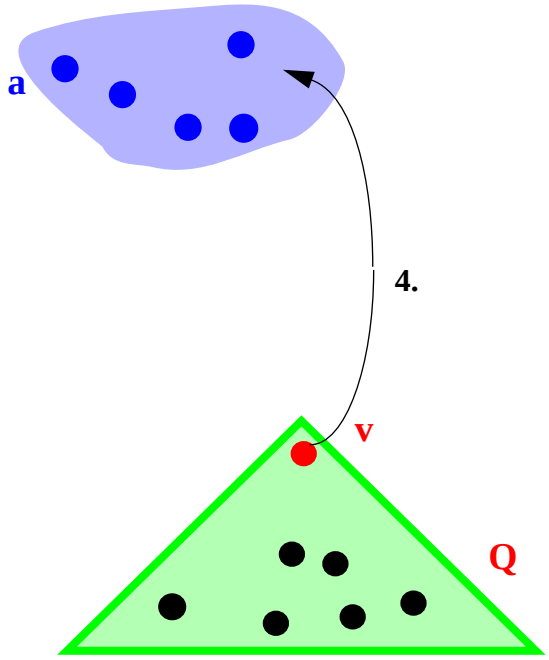
Løkke Invariant : **alle noder x som har blitt fjernet fra Q har korrekt D(x)**

– 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.



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())
4. v = Q.**removeMinElement**()
5. for hver z ε G.**outAdjacentVertices**(v)
6. if (D(v)+vekt(v,z) < D(z))
7. Q.**replaceKey**(z, D(v) + vekt(v,z)) ;

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

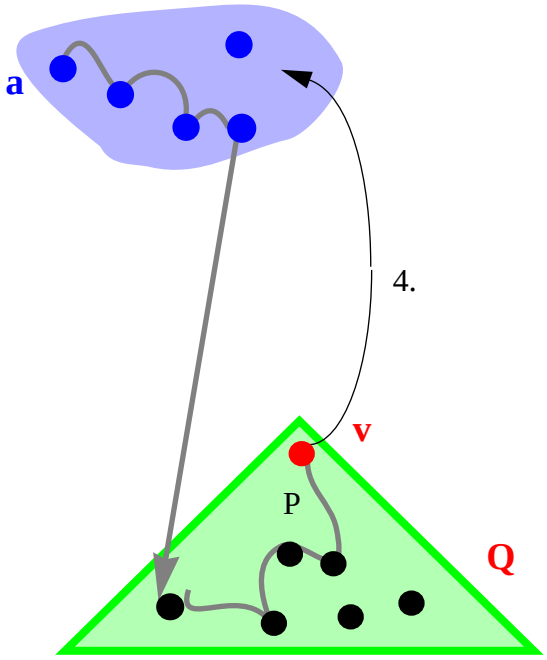
– holder før inngangen siden ingen node ble fjernet.

– 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)**



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())
4. v = Q.**removeMinElement**()
5. for hver z ε G.**outAdjacentVertices**(v)
6. if (D(v)+vekt(v,z) < D(z))
7. Q.**replaceKey**(z, D(v) + vekt(v,z)) ;

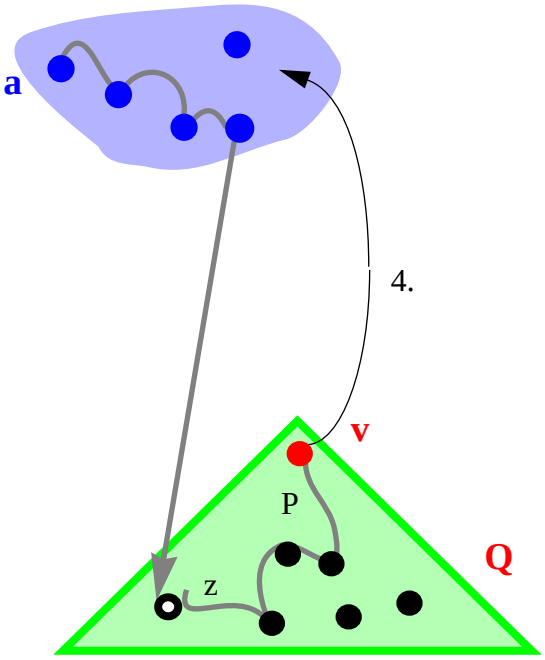
Løkke Invariant : **alle noder x som har blitt fjernet fra Q har korrekt D(x)**

– 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)**)



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())
4. v = Q.**removeMinElement**()
5. for hver z ε G.**outAdjacentVertices**(v)
6. if (D(v)+vekt(v,z) < D(z))
7. Q.**replaceKey**(z, D(v) + vekt(v,z)) ;

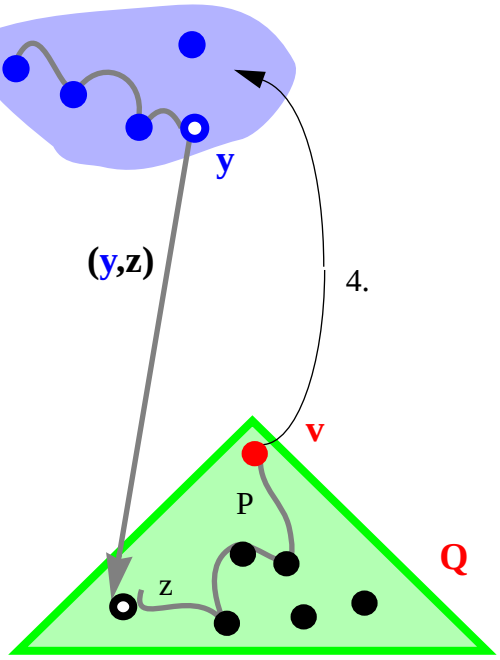
Løkke Invariant : **alle noder x som har blitt fjernet fra Q har korrekt D(x)**

– 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)**



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())
4. v = Q.**removeMinElement**()
5. for hver z ε G.**outAdjacentVertices**(v)
6. if (D(v)+vekt(v,z) < D(z))
7. Q.**replaceKey**(z, D(v) + vekt(v,z)) ;

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.

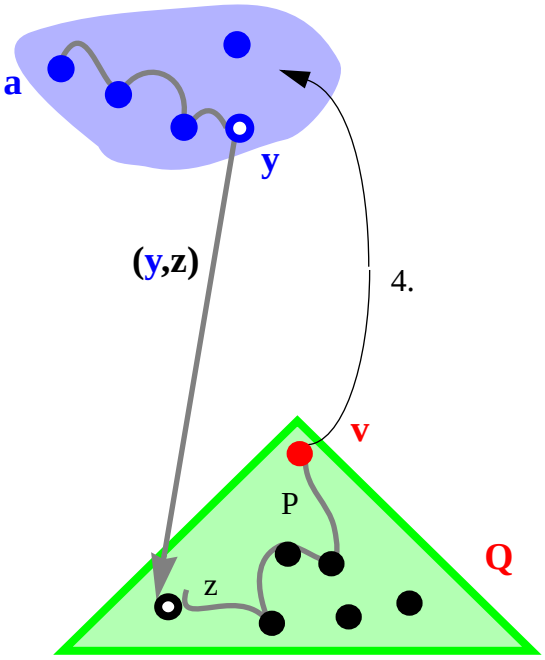
a) Anta ikke og la **v** være **den første** node for hvilken **D(v)** > **d(a,v)** ved 4.

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)**



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())
4. v = Q.**removeMinElement**()
5. for hver z ε G.**outAdjacentVertices**(v)
6. if (D(v)+vekt(v,z) < D(z))
7. Q.**replaceKey**(z, D(v) + vekt(v,z)) ;

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

– holder før inngangen siden ingen node ble fjernet.

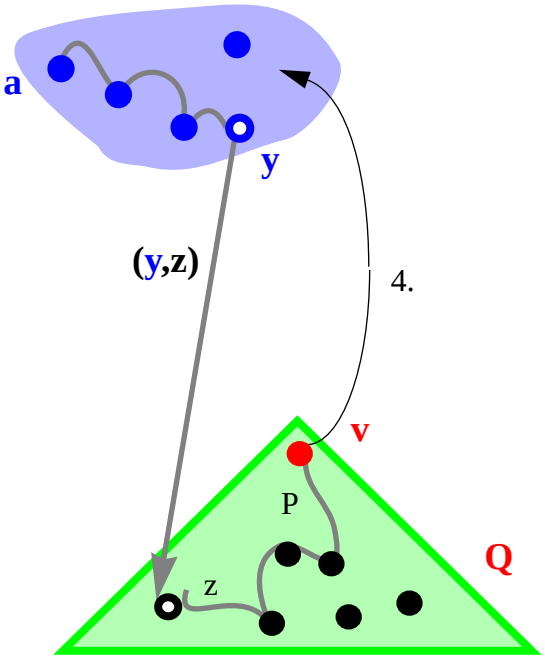
– for å vise at den opprettholdes i løkken, må vise at den holder etter 4.

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)**



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())
4. v = Q.**removeMinElement**()
5. for hver z ε G.**outAdjacentVertices**(v)
6. if (D(v)+vekt(v,z) < D(z))
7. Q.**replaceKey**(z, D(v) + vekt(v,z)) ;

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.

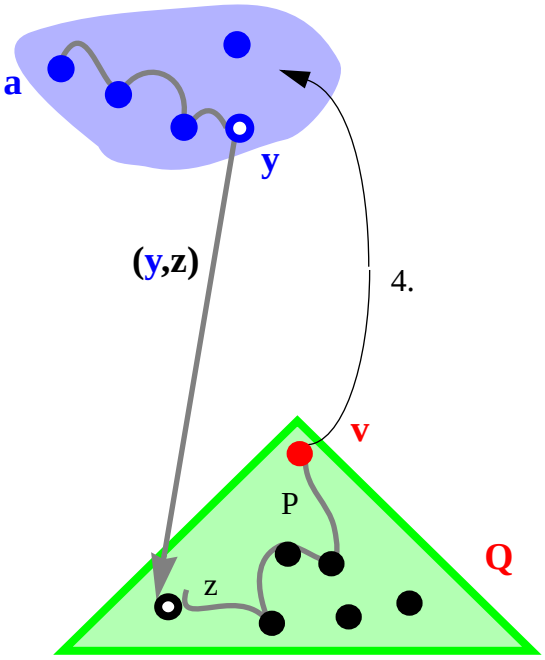
12.23 Ved 4. er D(v) = d(a,v) – lengden av korteste sti fra a til v.

- **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)**



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())
4. v = Q.**removeMinElement**()
5. for hver z ε G.**outAdjacentVertices**(v)
6. if (D(v)+vekt(v,z) < D(z))
7. Q.**replaceKey**(z, D(v) + vekt(v,z)) ;

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)**

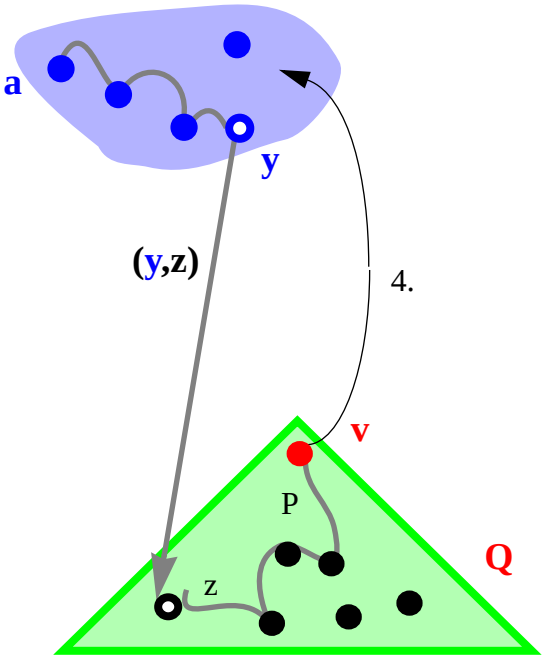
a) → **e)** **D(y) = d(a,y)**

4. → **f)** **D(v) ≤ D(z)**

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

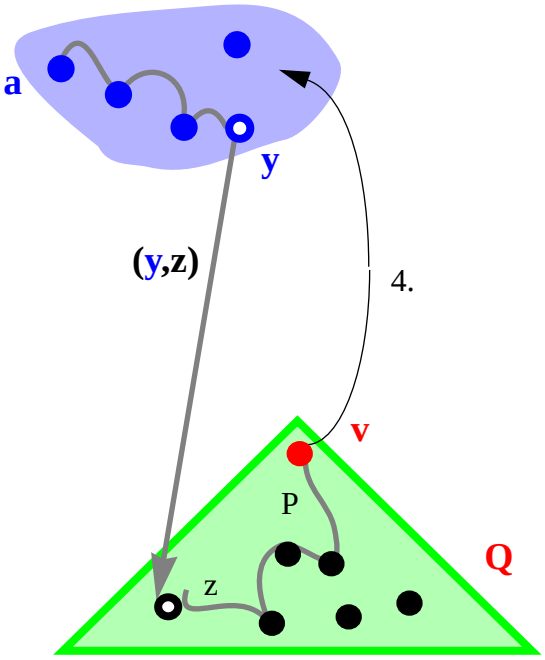
siden (y,z) er med i korteste sti P a—v, finns det ikke en kortere sti a—z enn

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



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())
4. v = Q.**removeMinElement**()
5. for hver z ε G.**outAdjacentVertices**(v)
6. if (D(v)+vekt(v,z) < D(z))
7. Q.**replaceKey**(z, D(v) + vekt(v,z)) ;



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

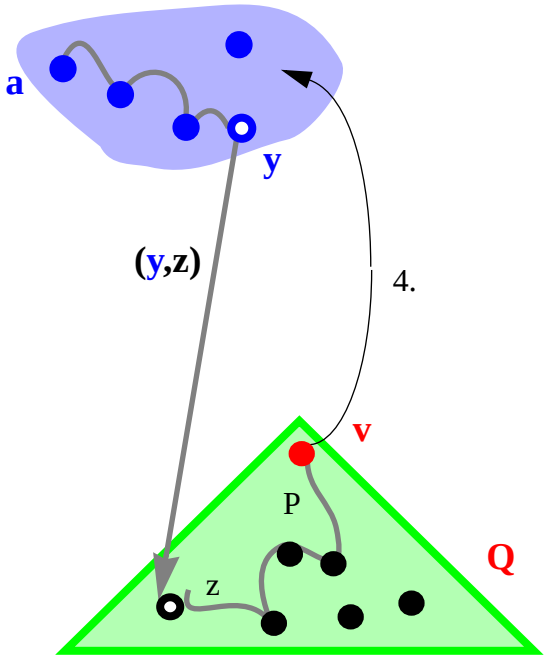
h) **d(a,z) = D(y) + vekt(y,z) = D(z)**

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)

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)) ;



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)**

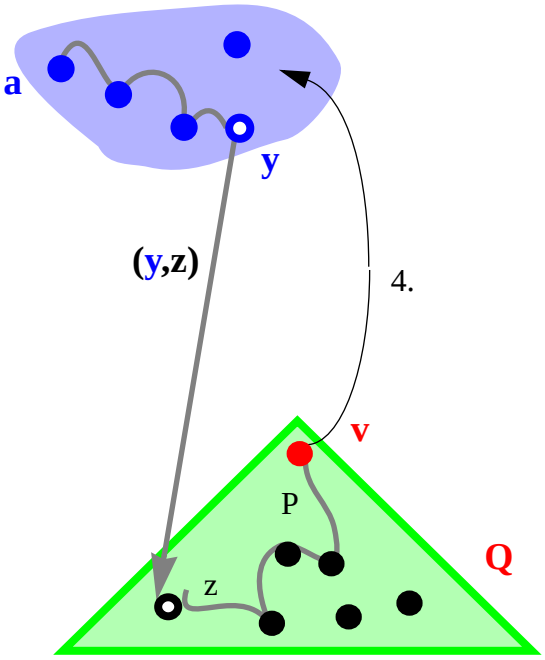
Men da:

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)

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) ) ;
```

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)



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)**

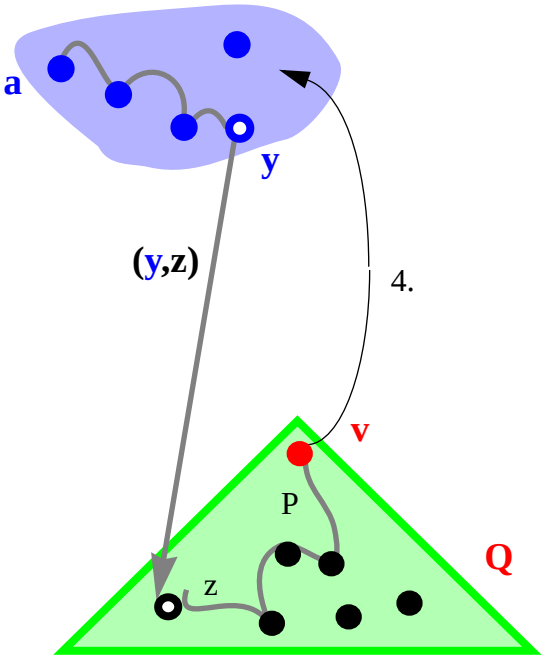
Men da:

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)

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+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)



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:

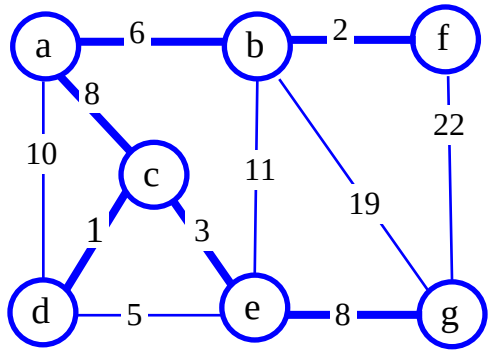
D(v) ≤ [f] D(z) = [h] d(a,z) ≤ d(a,z) + d(z,v) = [c] d(a,v) – motsier a) D(v) > d(a,v)

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



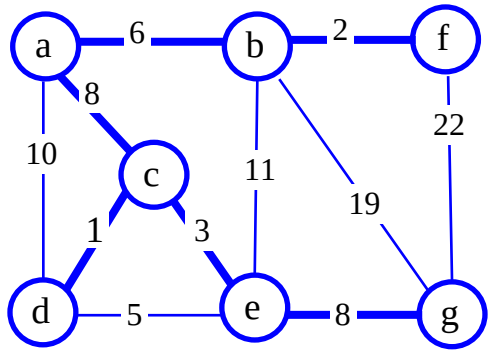
MST (Minimal Spanning Tree)

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*



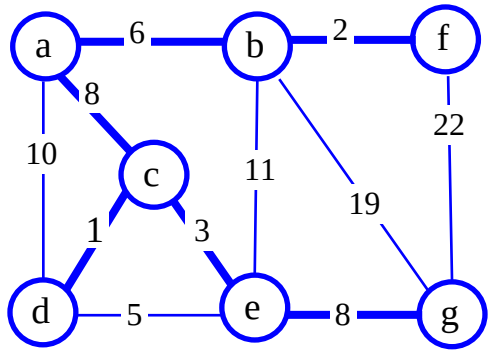
MST (Minimal Spanning Tree)

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 sammenlikn deres vekter ? – *don't even think about it !*

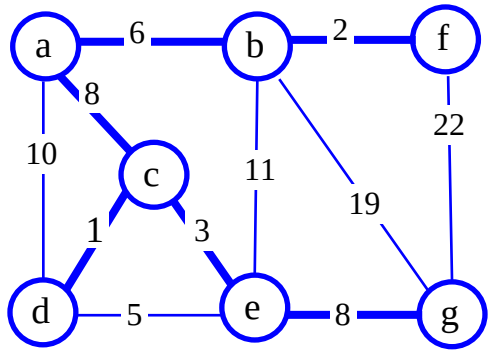


MST (Minimal Spanning Tree)

Gitt avstander mellom forskjellige byer, strek ledningene slik at

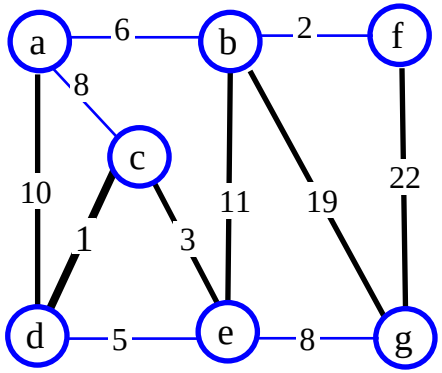
1. hvert par av byer er koblet sammen (en sti) og

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 sammenlikn deres vekter ? – *don't even think about it !*

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

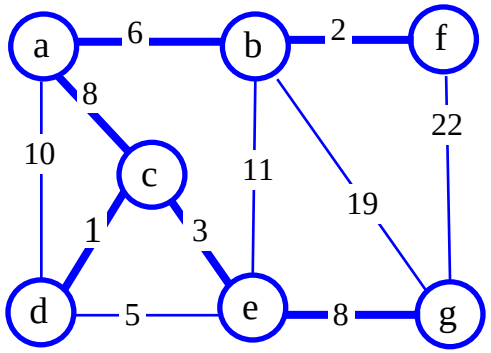


MST (Minimal Spanning Tree)

Gitt avstander mellom forskjellige byer, strek ledningene slik at

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

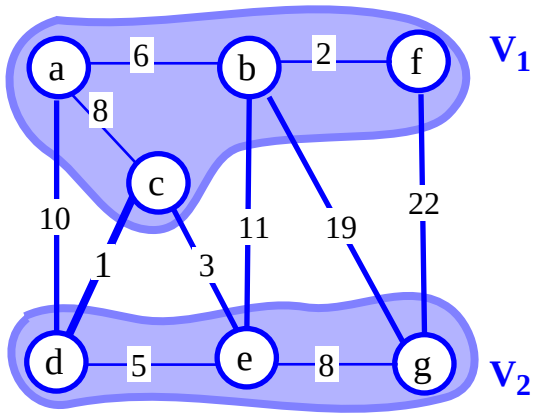
2. total lengde av brukt ledning er minimal



1. utspennende tre: DFS eller BFS (O(n+k)) – *lite sannsynlig at det blir minste*
2. finn alle trær og sammenlikn 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 $V_1 \cup V_2 = V$).**

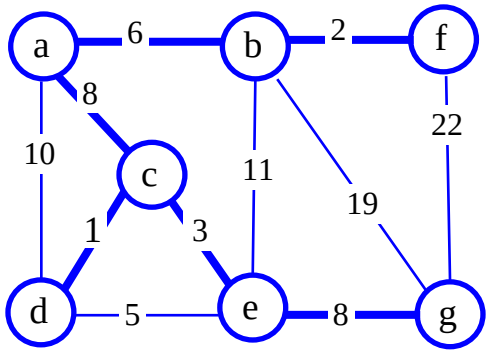


MST (Minimal Spanning Tree)

Gitt avstander mellom forskjellige byer, strek ledningene slik at

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

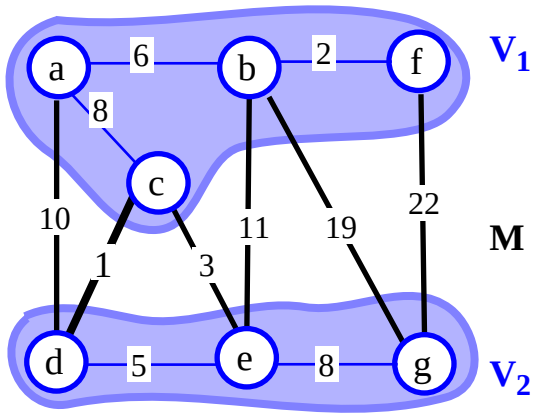
2. total lengde av brukt ledning er minimal



1. utspennende tre: DFS eller BFS (O(n+k)) – *lite sannsynlig at det blir minste*
2. finn alle trær og sammenlikn 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.**

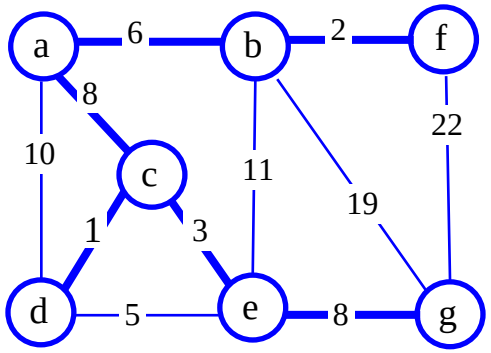


MST (Minimal Spanning Tree)

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

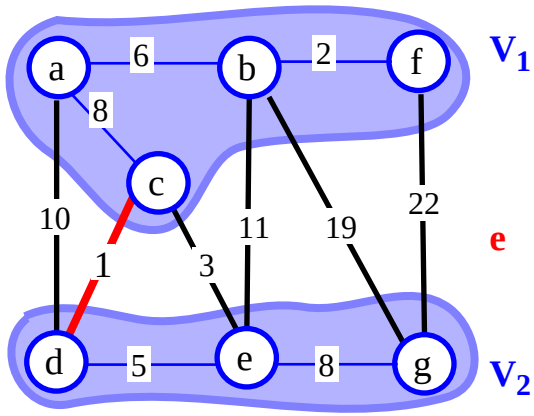


1. utspennende tre: DFS eller BFS (O(n+k)) – *lite sannsynlig at det blir minste*
2. finn alle trær og sammenlikn deres vekter ? – *don't even think about it !*

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).

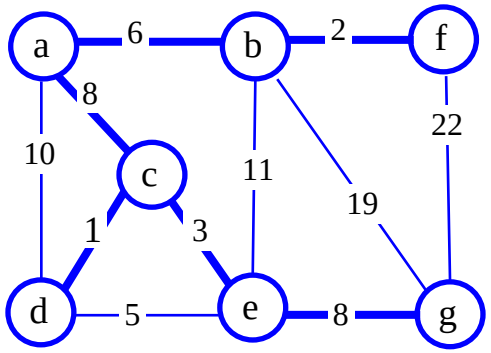


MST (Minimal Spanning Tree)

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

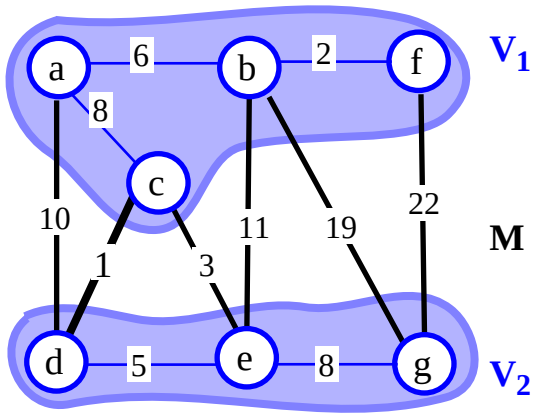


1. utspennende tre: DFS eller BFS (O(n+k)) – *lite sannsynlig at det blir minste*
2. finn alle trær og sammenlikn deres vekter ? – *don't even think about it !*

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 :

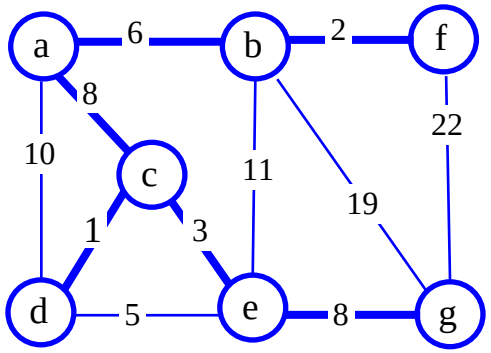
Et MST X må binde sammen V_1 og V_2 med en kant **k** : **v(e)** ≤ **v(k)**.

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

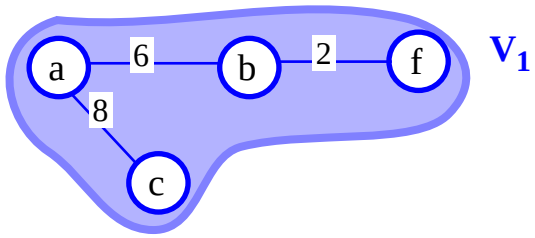


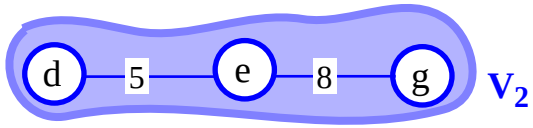
1. utspennende tre: DFS eller BFS (O(n+k)) – *lite sannsynlig at det blir minste*
2. finn alle trær og sammenlikn deres vekter ? – *don't even think about it !*

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 :

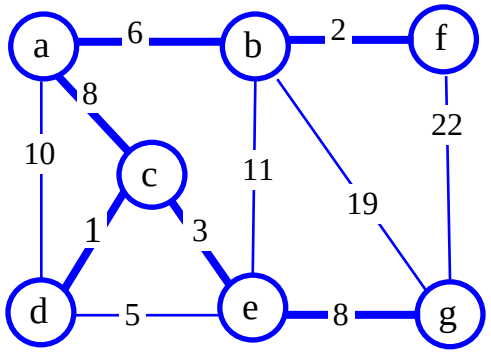
Et MST X må binde sammen V_1 og V_2 med en kant **k** : **v(e)** ≤ **v(k)**.

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

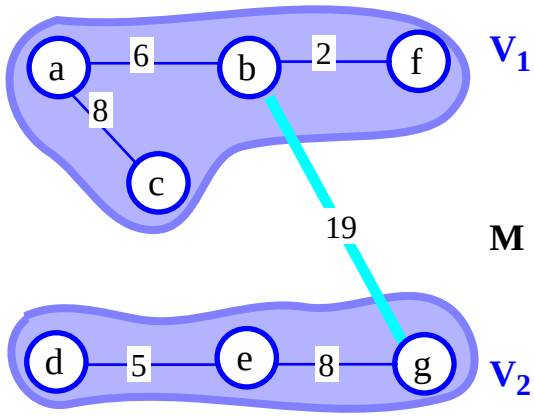


1. utspennende tre: DFS eller BFS (O(n+k)) – *lite sannsynlig at det blir minste*
2. finn alle trær og sammenlikn deres vekter ? – *don't even think about it !*

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 :

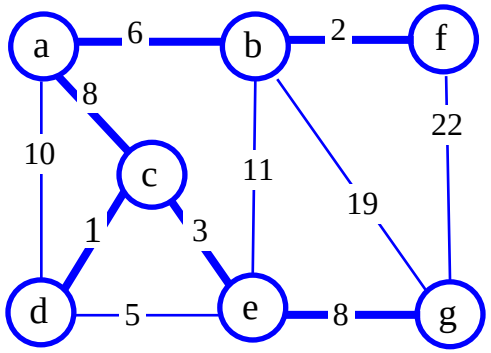
Et MST X må binde sammen V_1 og V_2 med en kant **k** : **v(e)** ≤ **v(k)**.

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

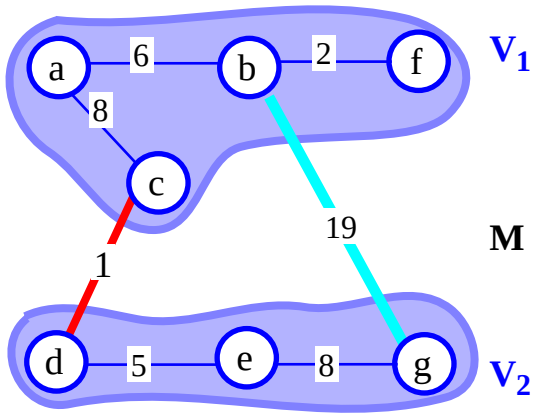


1. utspennende tre: DFS eller BFS (O(n+k)) – *lite sannsynlig at det blir minste*
2. finn alle trær og sammenlikn deres vekter ? – *don't even think about it !*

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)**.

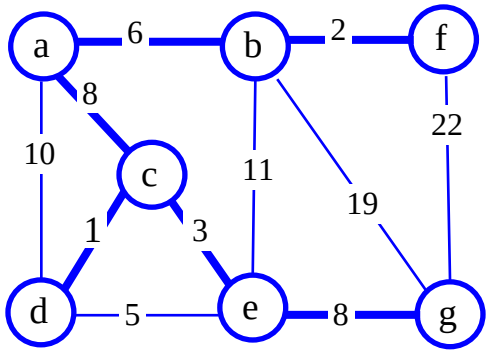
Legger vi kant **e** til X, får vi en syklus men kun på formen **k**–V_1–**e**–V_2.

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

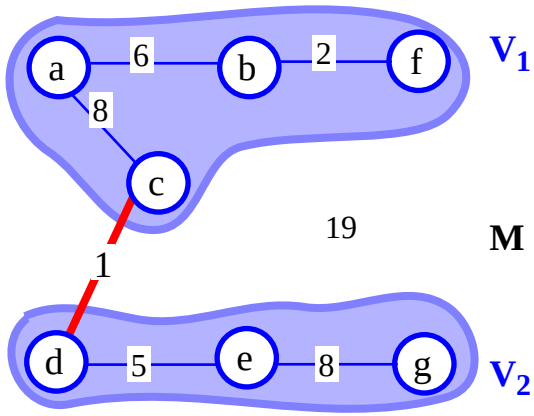


1. utspennende tre: DFS eller BFS (O(n+k)) – *lite sannsynlig at det blir minste*
2. finn alle trær og sammenlikn deres vekter ? – *don't even think about it !*

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.

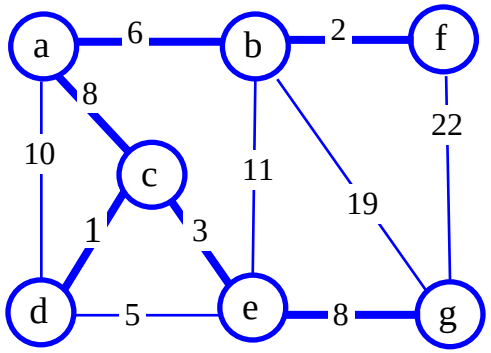
Fjerner vi **k** får vi et tre med høyst samme vekt, dvs et MST.

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

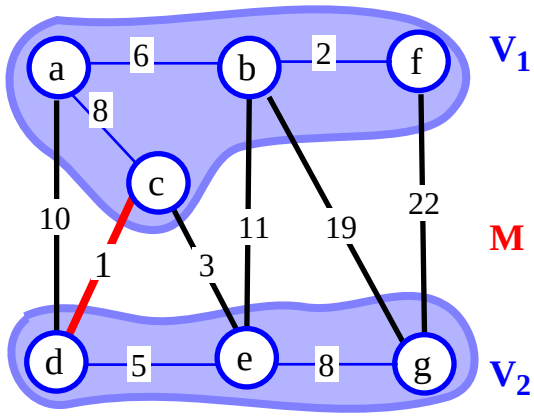


1. utspennende tre: DFS eller BFS (O(n+k)) – *lite sannsynlig at det blir minste*
2. finn alle trær og sammenlikn deres vekter ? – *don't even think about it !*

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.

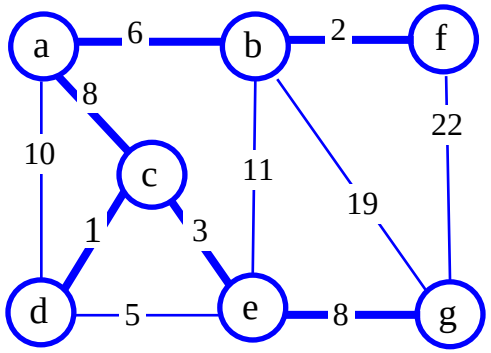
Dermed : hvis alle kanter har forskjellige vekter, er MST entydig bestemt

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

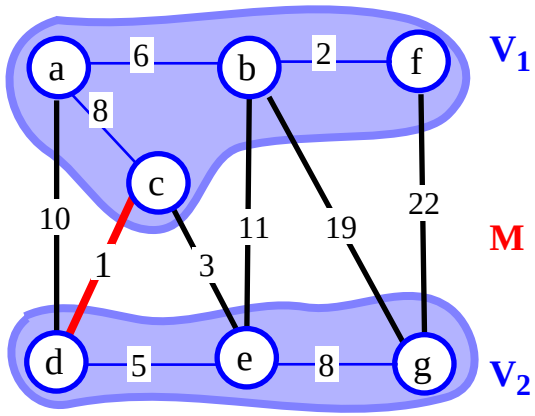


1. utspennende tre: DFS eller BFS (O(n+k)) – *lite sannsynlig at det blir minste*
2. finn alle trær og sammenlikn deres vekter ? – *don't even think about it !*

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).

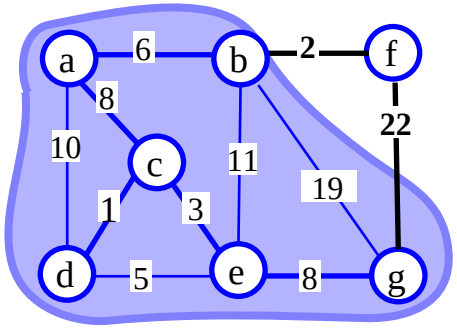


Begrunnelse :

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.



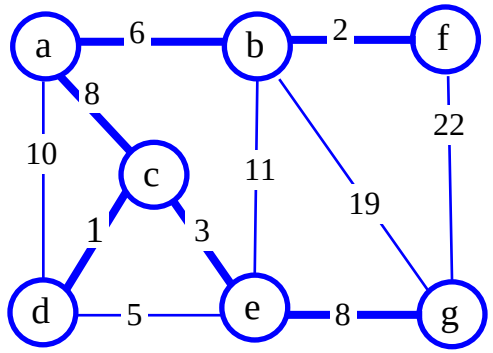
Dermed : hvis alle kanter har forskjellige vekter, er MST entydig bestemt

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

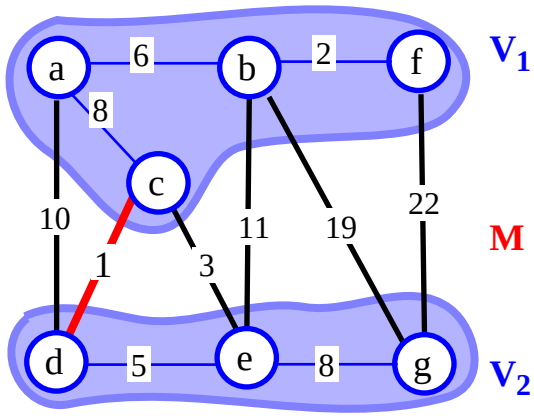


1. utspennende tre: DFS eller BFS (O(n+k)) – *lite sannsynlig at det blir minste*
2. finn alle trær og sammenlikn deres vekter ? – *don't even think about it !*

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).

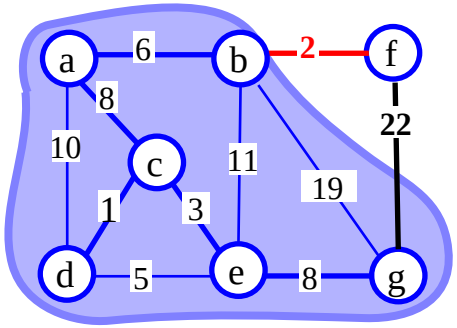


Begrunnelse :

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.



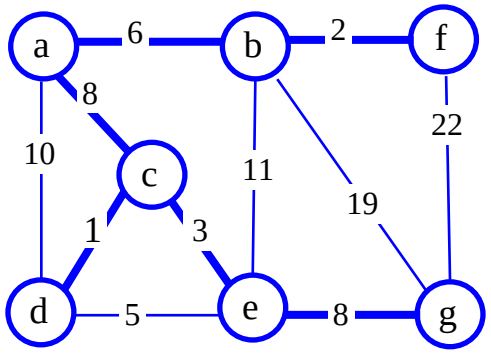
Dermed : hvis alle kanter har forskjellige vekter, er MST entydig bestemt

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

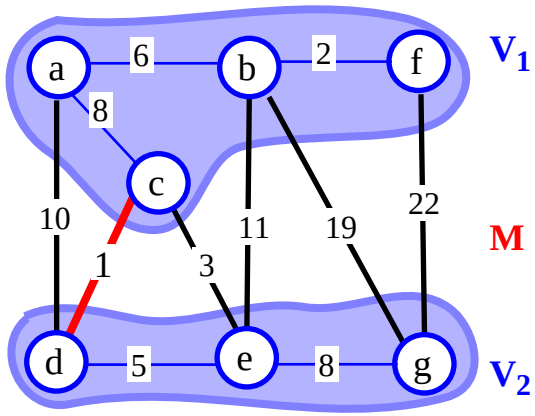


1. utspennende tre: DFS eller BFS (O(n+k)) – *lite sannsynlig at det blir minste*
2. finn alle trær og sammenlikn deres vekter ? – *don't even think about it !*

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).

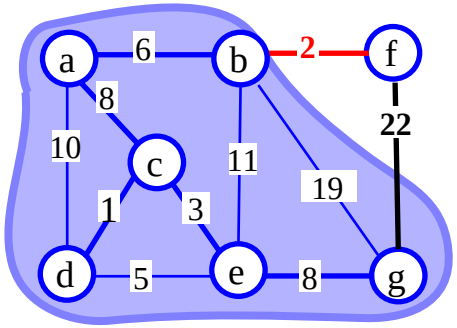


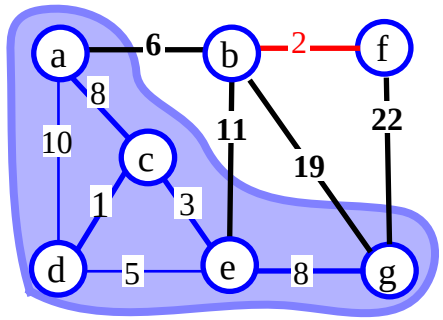
Begrunnelse :

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.





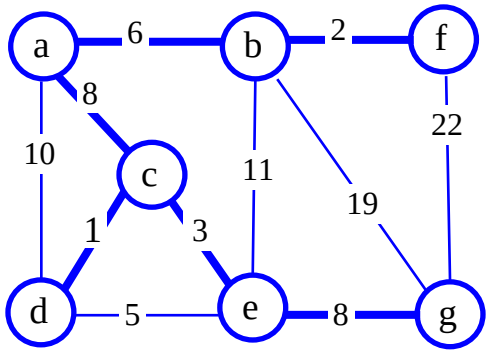
Dermed : hvis alle kanter har forskjellige vekter, er MST entydig bestemt

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

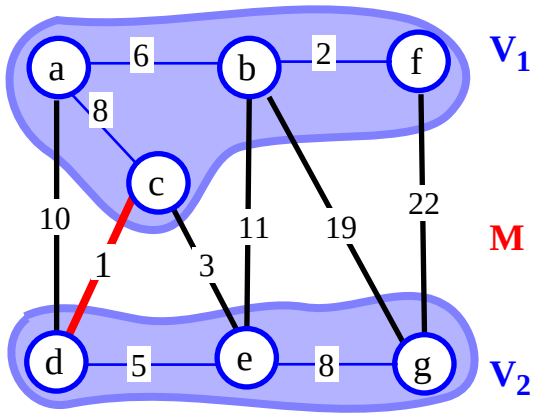


1. utspennende tre: DFS eller BFS (O(n+k)) – *lite sannsynlig at det blir minste*
2. finn alle trær og sammenlikn deres vekter ? – *don't even think about it !*

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).

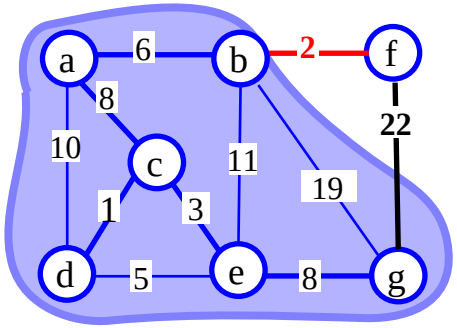


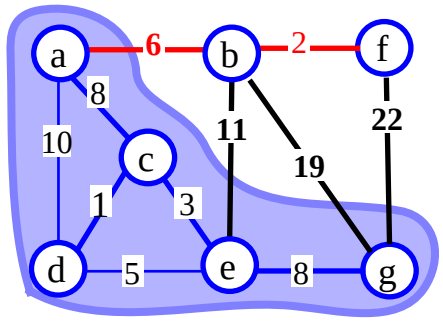
Begrunnelse :

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.





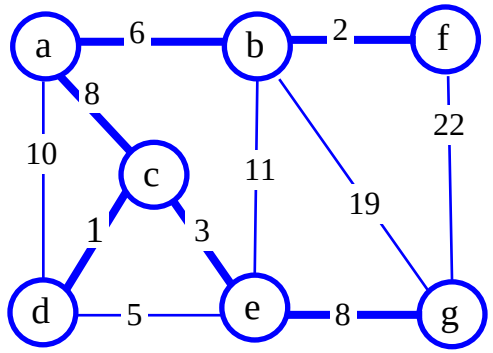
Dermed : hvis alle kanter har forskjellige vekter, er MST entydig bestemt

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

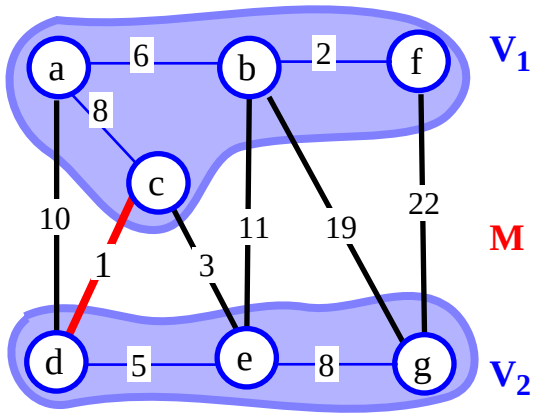


1. utspennende tre: DFS eller BFS (O(n+k)) – *lite sannsynlig at det blir minste*
2. finn alle trær og sammenlikn deres vekter ? – *don't even think about it !*

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).

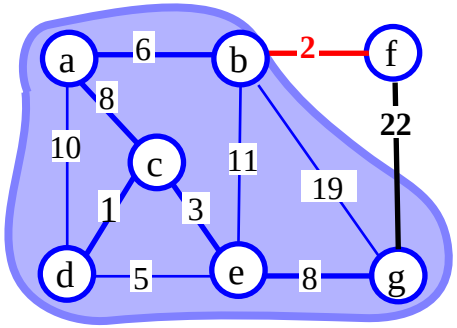


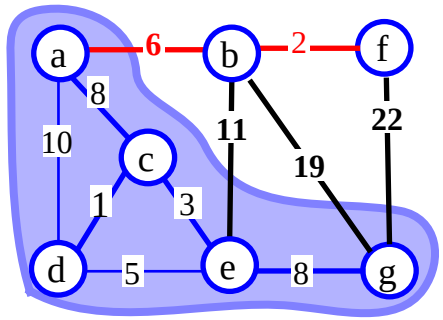
Begrunnelse :

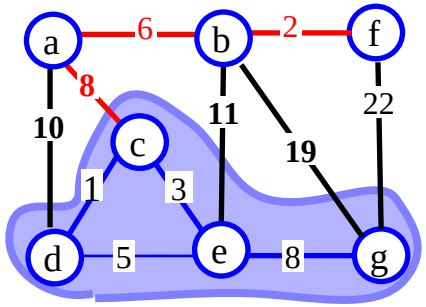
Et MST X må binde sammen V_1 og V_2 med en kant **k** : **v(e)** ≤ **v(k)**.

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.



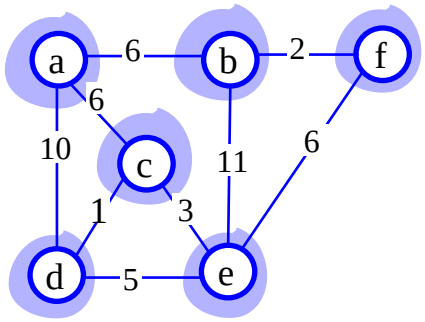




Dermed : hvis alle kanter har forskjellige vekter, er MST entydig bestemt

Kruskal algoritme MST

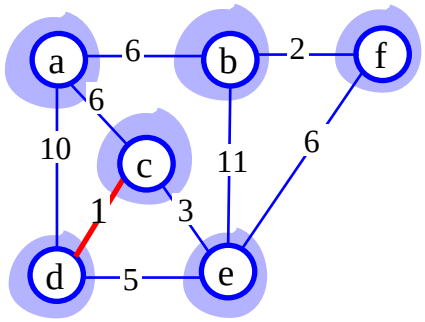
```
Kruskal(Graph G=(V,E))
   // sammenhengende, vektet
   for hver node v ε V : C(v) = {v}
   Q = PriorityQueue med alle kanter mht. vekt
   T = Ø
   while ( ! Q.isEmpty() )
      (v,u) = Q.removeMinElement();// Greedy
      if C(v) != C(u)
         legg (v,u) til T
         C(v) = C(u) = C(v) ∪ C(u)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



Q= 1 2 3 5 6 6 6 10 11

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)
```

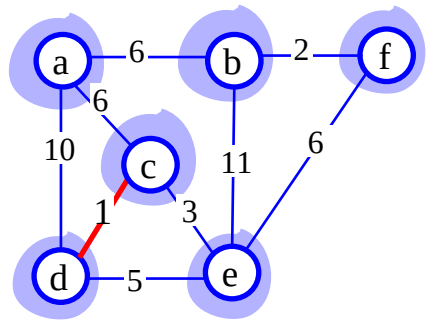


Q= **1** 2 3 5 6 6 6 10 11

(c,d)

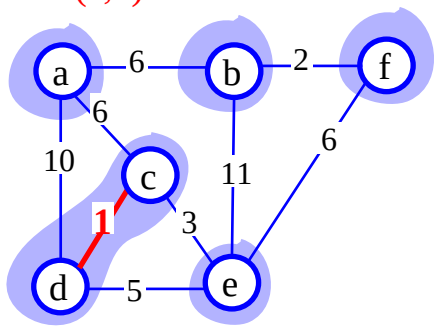
Kruskal algoritme MST

```
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)
```



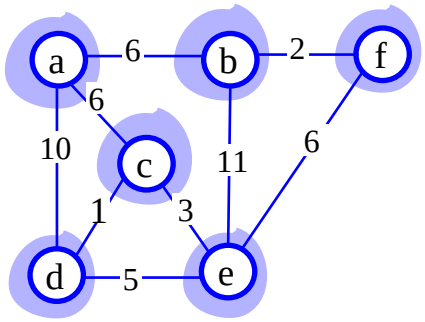
Q= **1** 2 3 5 6 6 6 10 11

(c,d)



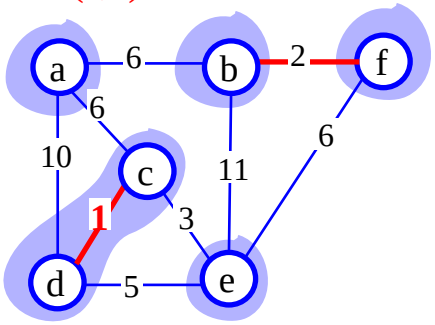
Kruskal algoritme MST

```
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)
```



Q= 1 2 3 5 6 6 6 10 11

(c,d)

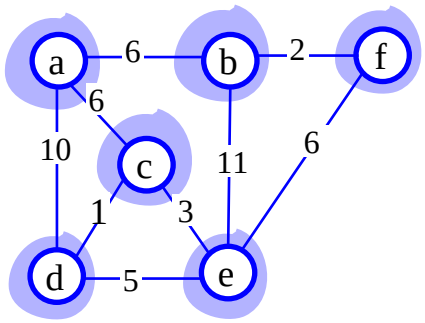


Q= 2 3 5 6 6 6 10 11

(b,f)

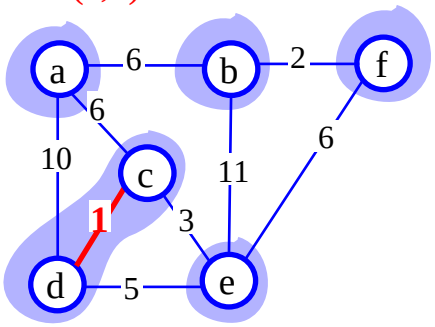
Kruskal algoritme MST

```
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)
```



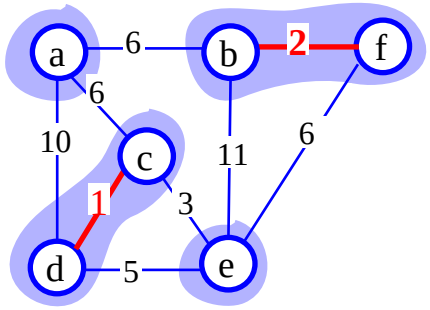
Q= 1 2 3 5 6 6 6 10 11

(c,d)



Q= 2 3 5 6 6 6 10 11

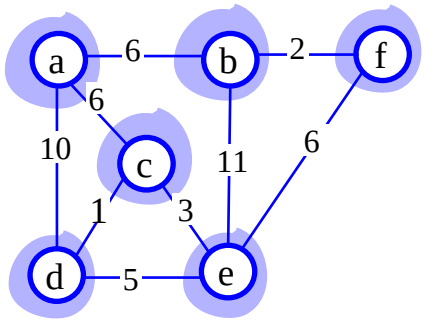
(b,f)



Q= 3 5 6 6 6 10 11

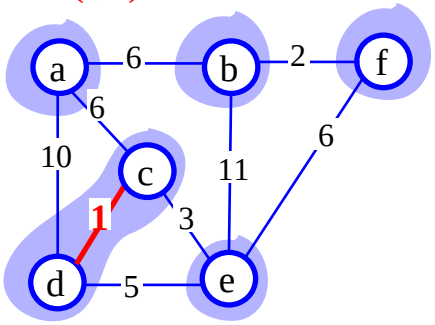
Kruskal algoritme MST

```
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)
```



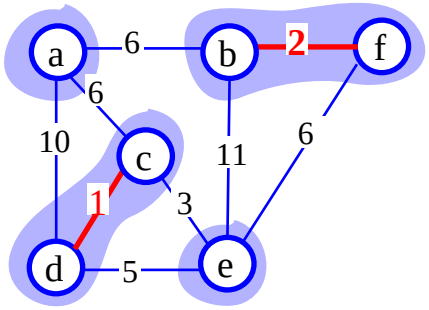
Q= 1 2 3 5 6 6 6 10 11

(c,d)



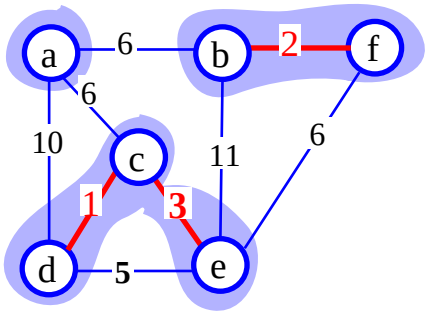
Q= 2 3 5 6 6 6 10 11

(b,f)



Q= 3 5 6 6 6 10 11

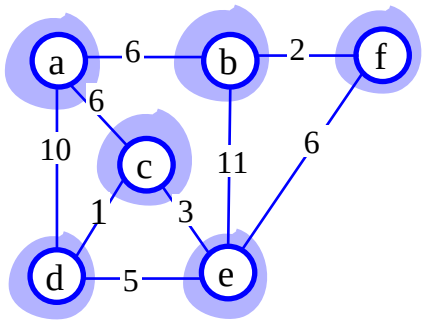
(c,e)



Q= 5 6 6 6 10 11

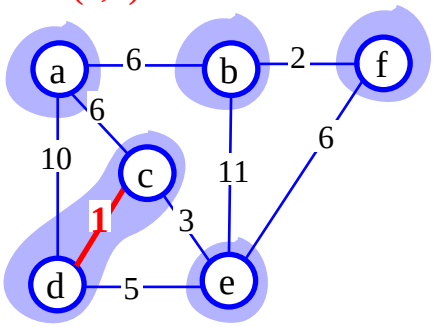
Kruskal algoritme MST

```
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)
```



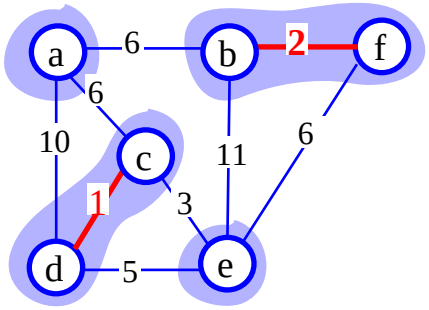
Q= 1 2 3 5 6 6 6 10 11

(c,d)



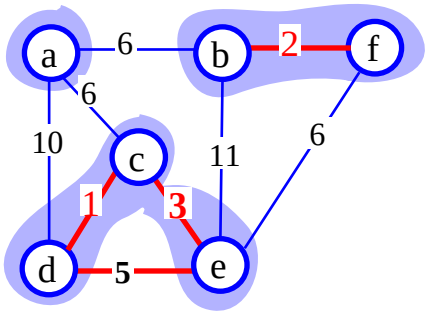
Q= 2 3 5 6 6 6 10 11

(b,f)



Q= 3 5 6 6 6 10 11

(c,e)

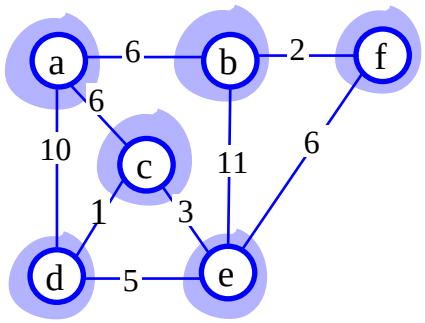


Q= 5 6 6 6 10 11

(d,e)

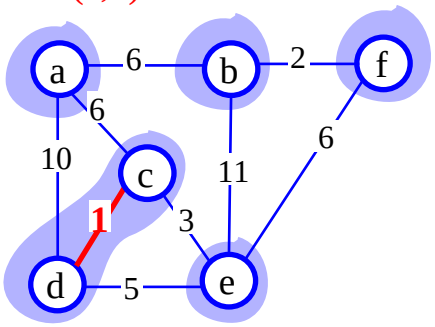
Kruskal algoritme MST

```
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)
```



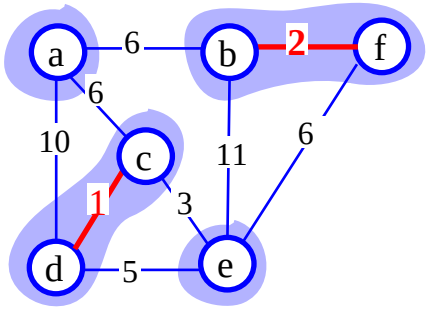
Q= 1 2 3 5 6 6 6 10 11

(c,d)



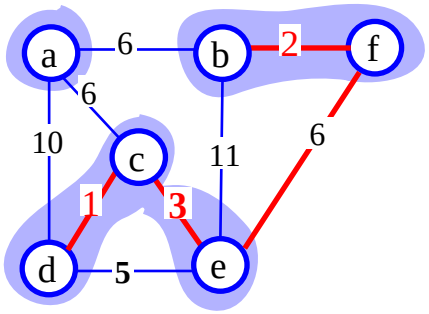
Q= 2 3 5 6 6 6 10 11

(b,f)



Q= 3 5 6 6 6 10 11

(c,e)



Q= 5 6 6 6 10 11

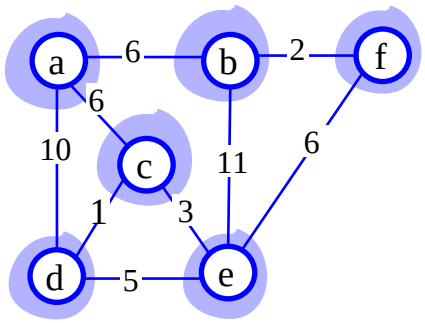
(d,e)

Q= 6 6 6 10 11

(e,f)

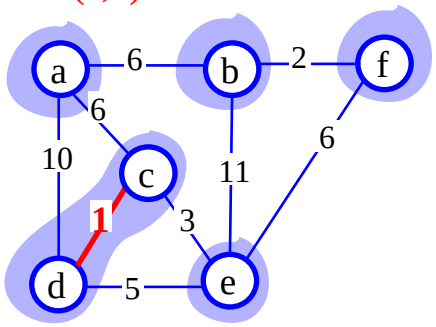
Kruskal algoritme MST

```
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)
```



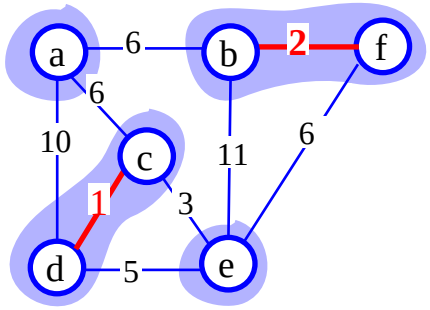
Q= 1 2 3 5 6 6 6 10 11

(c,d)



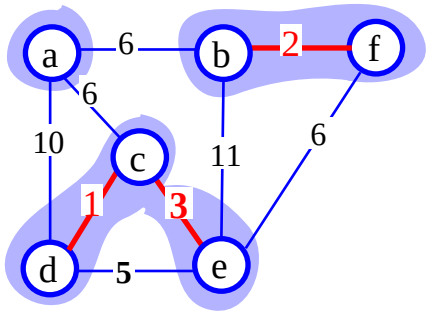
Q= 2 3 5 6 6 6 10 11

(b,f)



Q= 3 5 6 6 6 10 11

(c,e)

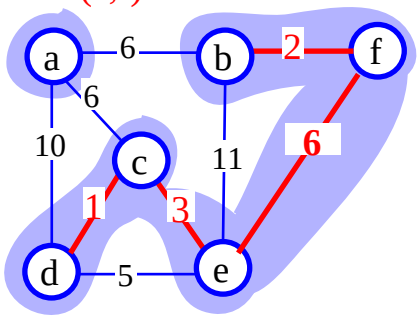


Q= 5 6 6 6 10 11

(d,e)

Q= 6 6 6 10 11

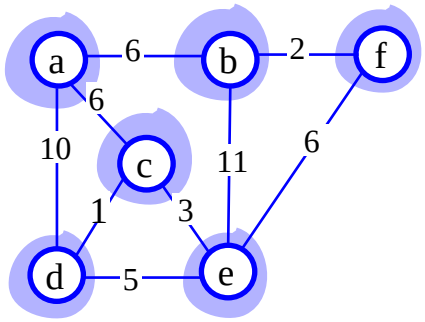
(e,f)



Q= 6 6 10 11

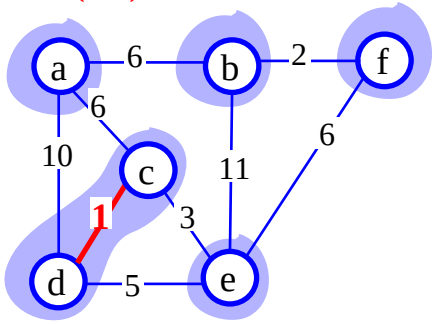
Kruskal algoritme MST

```
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)
```



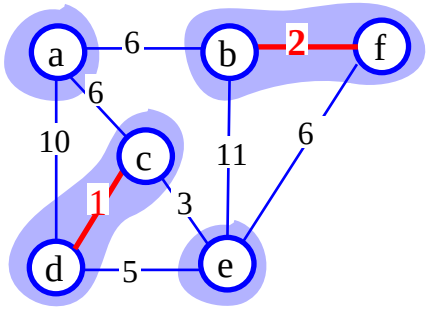
Q= 1 2 3 5 6 6 6 10 11

(c,d)



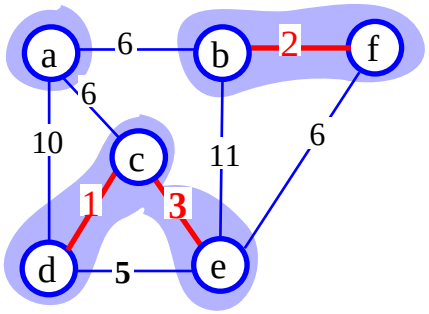
Q= 2 3 5 6 6 6 10 11

(b,f)



Q= 3 5 6 6 6 10 11

(c,e)

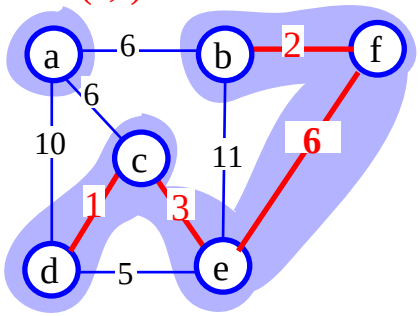


Q= 5 6 6 6 10 11

(d,e)

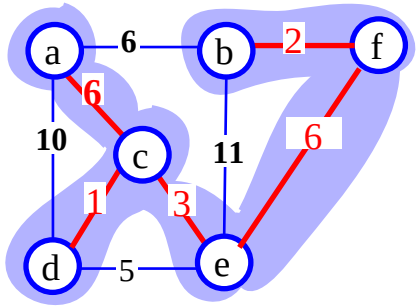
Q= 6 6 6 10 11

(e,f)



Q= 6 6 10 11

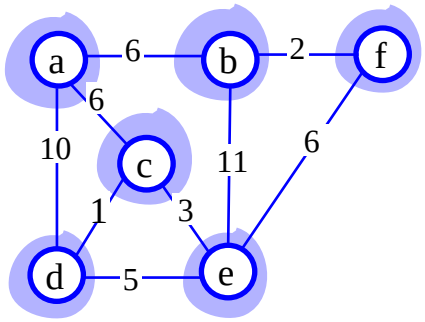
(a,c)



Kruskal algoritme MST

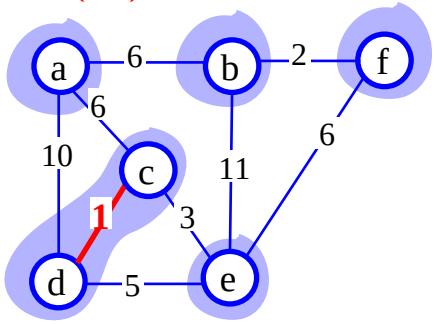
```
Kruskal(Graph G=(V,E))
  // sammenhengende, vektet
  for hver node v ε V : C(v) = {v}
  Q = PriorityQueue med alle kanter mht. vekt
  T = Ø
  while ( ! Q.isEmpty() )
     (v,u) = Q.removeMinElement();// Greedy
     if C(v) != C(u)
        legg (v,u) til T
        C(v) = C(u) = C(v) ∪ C(u)
```

LI: T inneholder MSTv for hver C(v)



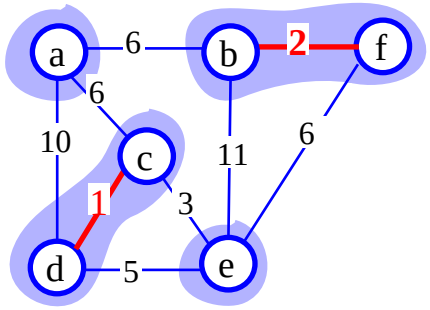
Q= 1 2 3 5 6 6 6 10 11

(c,d)



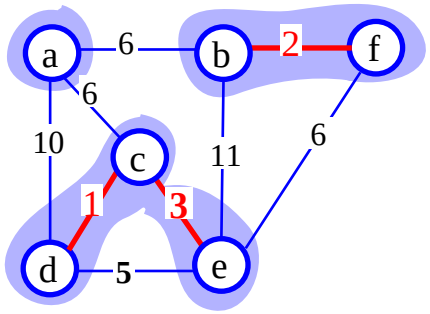
Q= 2 3 5 6 6 6 10 11

(b,f)



Q= 3 5 6 6 6 10 11

(c,e)

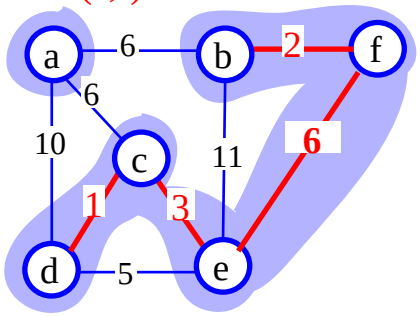


Q= 5 6 6 6 10 11

(d,e)

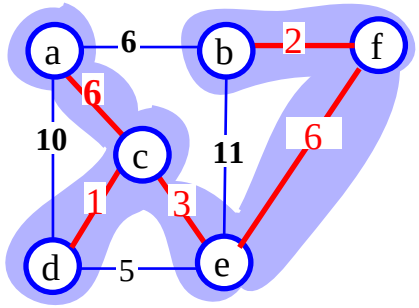
Q= 6 6 6 10 11

(e,f)



Q= 6 6 10 11

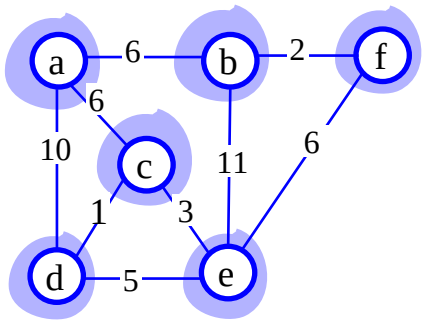
(a,c)



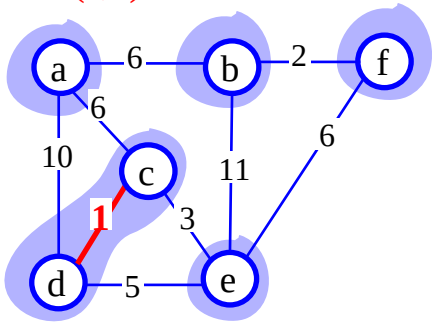
Kruskal algoritme MST

```
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)
```

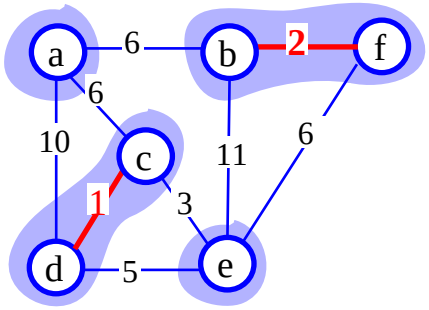
LI: T inneholder MSTv for hver C(v)
– før inngangen i løkka - trivielt
– rundgang : 12.25



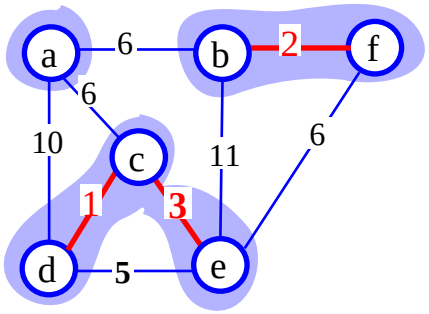
Q= 1 2 3 5 6 6 6 10 11
(c,d)



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

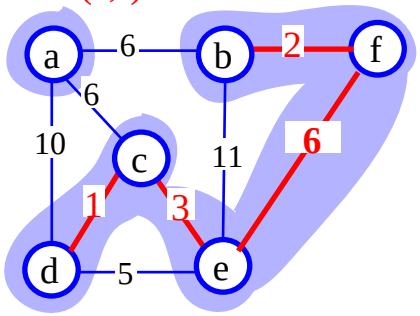


Q= 3 5 6 6 6 10 11
(c,e)

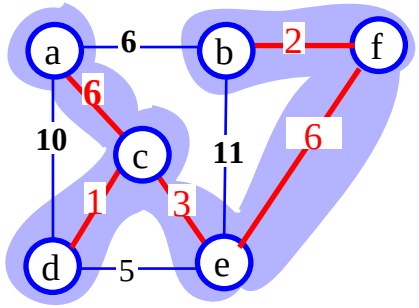


Q= 5 6 6 6 10 11
(d,e)

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



Q= 6 6 10 11
(a,c)



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)
```

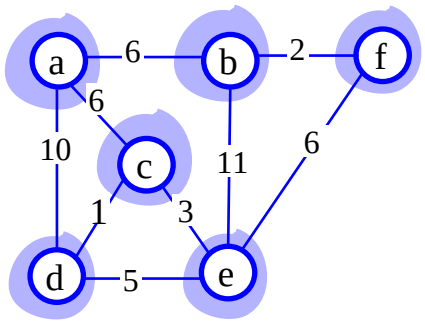
LI: T inneholder MSTv for hver C(v)

– før inngangen i løkka - trivielt

– rundgang : 12.25

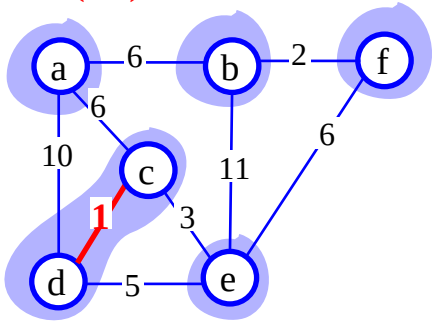
• hvis C(v) == C(u) :

vekt(v,u) vekt(k) for alle k i C(v): 12.25



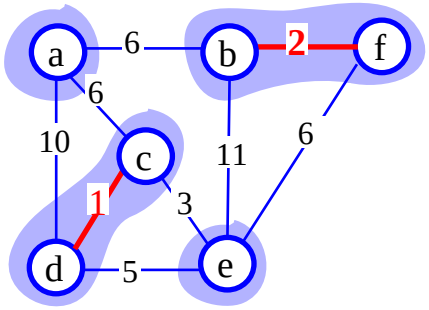
Q= 1 2 3 5 6 6 6 10 11

(c,d)



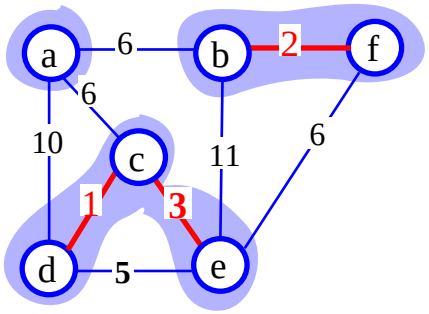
Q= 2 3 5 6 6 6 10 11

(b,f)



Q= 3 5 6 6 6 10 11

(c,e)

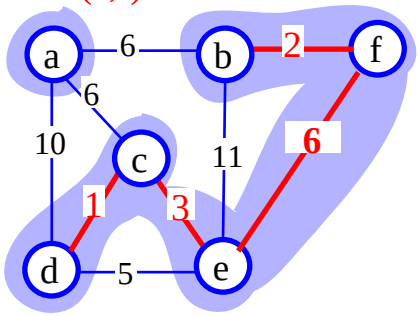


Q= 5 6 6 6 10 11

(d,e)

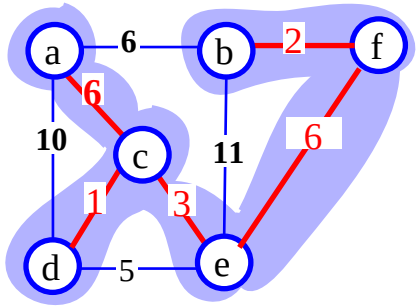
Q= 6 6 6 10 11

(e,f)



Q= 6 6 10 11

(a,c)



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)

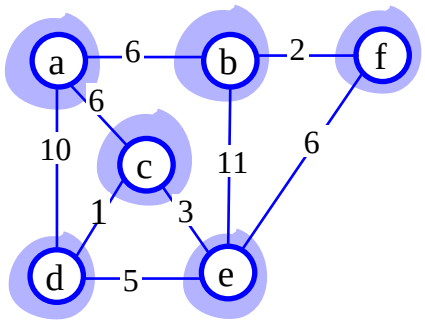
– før inngangen i løkka - trivielt

– rundgang : 12.25

• hvis C(v) == C(u) :

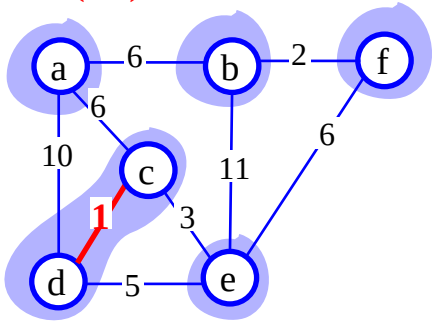
vekt(v,u) ≥ vekt(k) for alle k i C(v): 12.25

• hvis C(v) != C(u) : 12.25



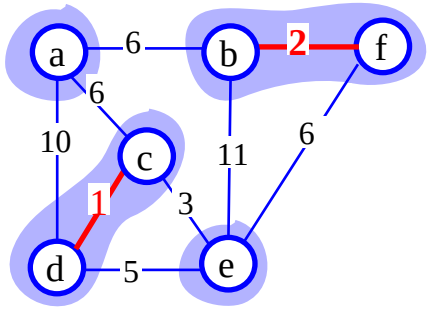
Q= 1 2 3 5 6 6 6 10 11

(c,d)



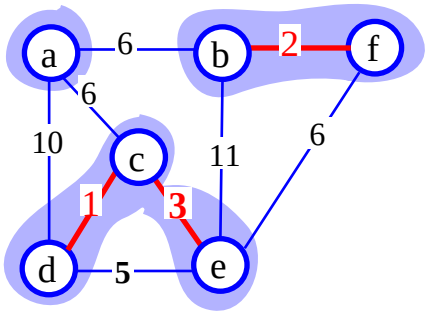
Q= 2 3 5 6 6 6 10 11

(b,f)



Q= 3 5 6 6 6 10 11

(c,e)

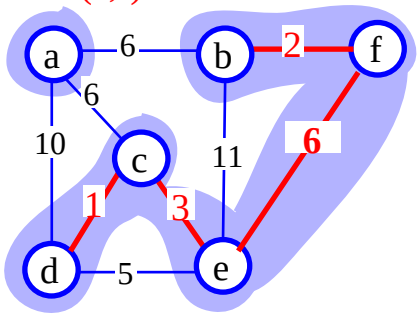


Q= 5 6 6 6 10 11

(d,e)

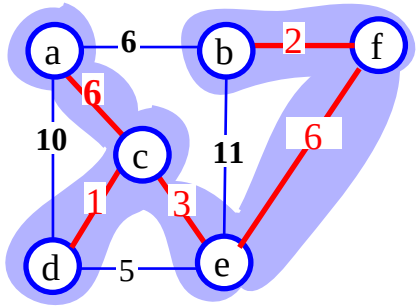
Q= 6 6 6 10 11

(e,f)



Q= 6 6 10 11

(a,c)

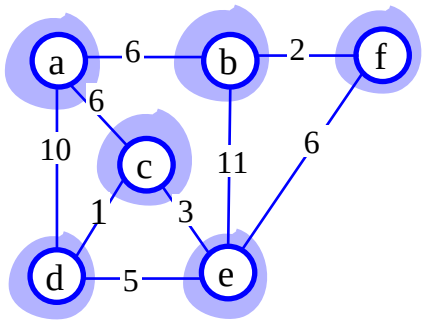


Kruskal algoritme MST

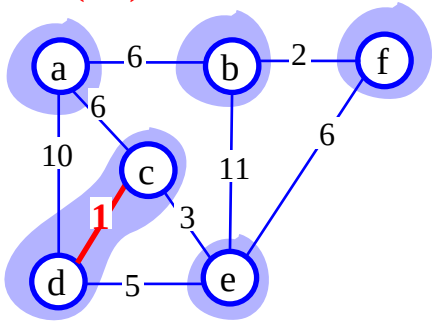
```
Kruskal(Graph G=(V,E))
  // sammenhengende, vektet
  for hver node v ε V : C(v) = {v}
  Q = PriorityQueue med alle kanter mht. vekt
  T = Ø
  while ( ! Q.isEmpty() )
     (v,u) = Q.removeMinElement();// Greedy
     if C(v) != C(u)
        legg (v,u) til T
        C(v) = C(u) = C(v) ∪ C(u)
```

LI: T inneholder MSTv for hver C(v)
– før inngangen i løkka - trivielt
– rundgang : 12.25
  • hvis C(v) == C(u) :
   vekt(v,u) vekt(k) for alle k i C(v): 12.25
  • hvis C(v) != C(u) : 12.25

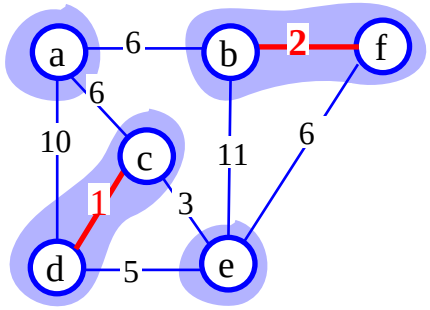
O(n +



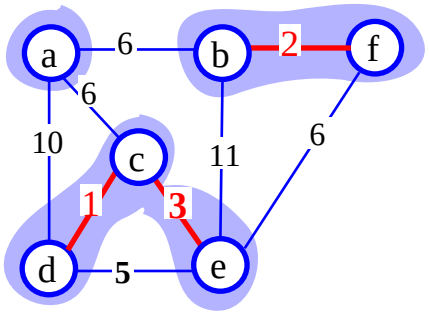
Q= 1 2 3 5 6 6 6 10 11
(c,d)



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

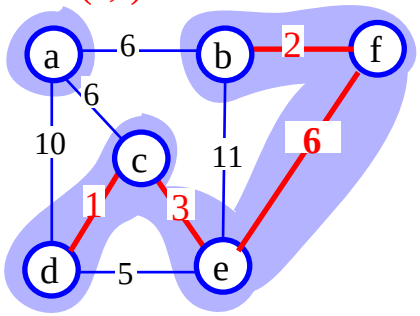


Q= 3 5 6 6 6 10 11
(c,e)

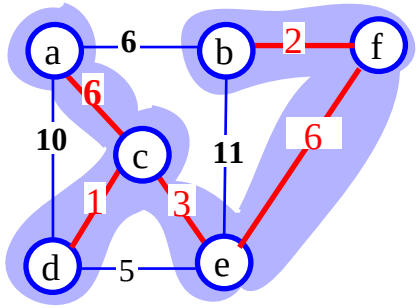


Q= 5 6 6 6 10 11
(d,e)

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



Q= 6 6 10 11
(a,c)

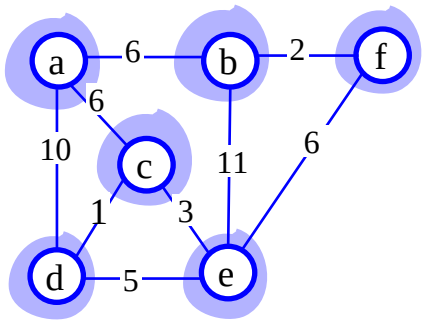


Kruskal algoritme MST

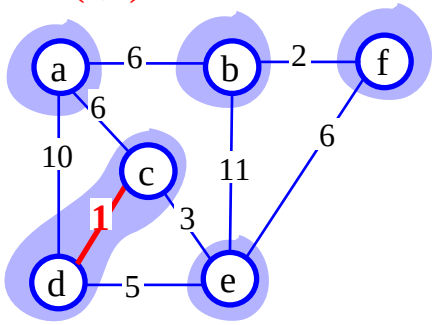
```
Kruskal(Graph G=(V,E))
  // sammenhengende, vektet
  for hver node v ε V : C(v) = {v}
  Q = PriorityQueue med alle kanter mht. vekt
  T = Ø
  while ( ! Q.isEmpty() )
     (v,u) = Q.removeMinElement();// Greedy
     if C(v) != C(u)
        legg (v,u) til T
        C(v) = C(u) = C(v) ∪ C(u)
```

LI: T inneholder MSTv for hver C(v)
– før inngangen i løkka - trivielt
– rundgang : 12.25
  • hvis C(v) == C(u) :
   vekt(v,u) vekt(k) for alle k i C(v): 12.25
  • hvis C(v) != C(u) : 12.25

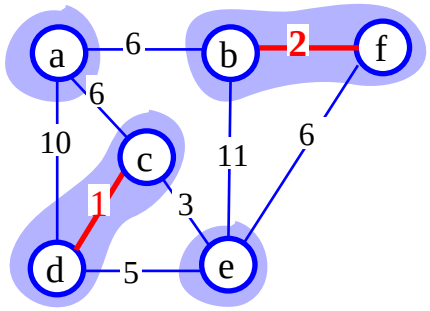
O(n +
  k log k +



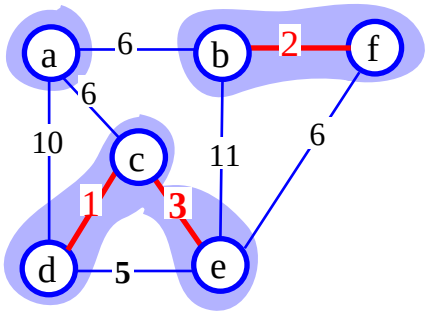
Q= 1 2 3 5 6 6 6 10 11
(c,d)



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

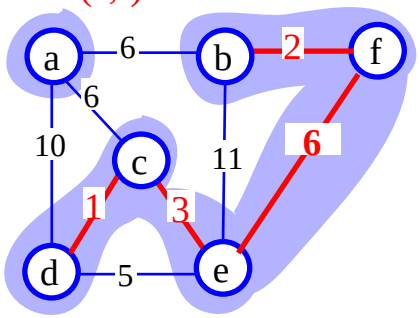


Q= 3 5 6 6 6 10 11
(c,e)

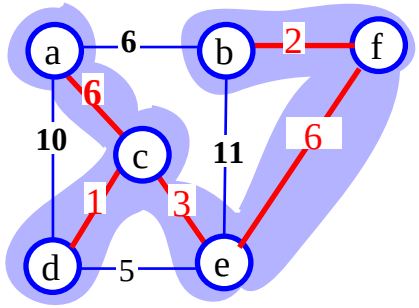


Q= 5 6 6 6 10 11
(d,e)

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



Q= 6 6 10 11
(a,c)



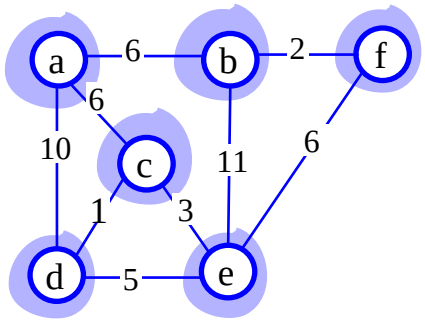
Kruskal algoritme MST

```
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)
```

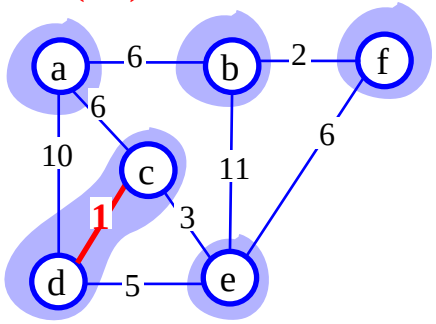
LI: T inneholder MSTv for hver C(v)
– før inngangen i løkka - trivielt
– rundgang : 12.25
  • hvis C(v) == C(u) :
   vekt(v,u) vekt(k) for alle k i C(v): 12.25
  • hvis C(v) != C(u) : 12.25

O(n +
  k log k +
  k *(

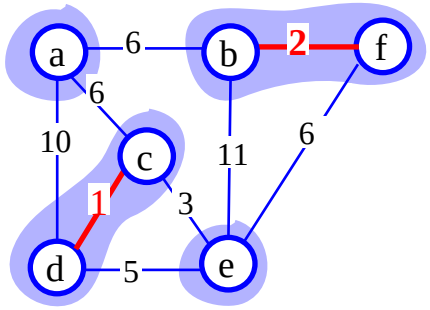
)



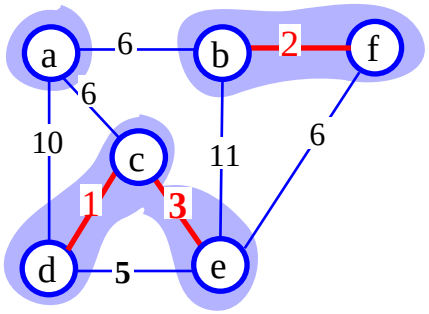
Q= 1 2 3 5 6 6 6 10 11
(c,d)



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

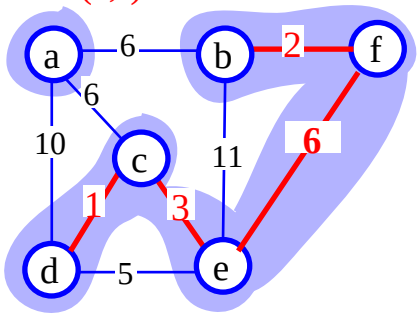


Q= 3 5 6 6 6 10 11
(c,e)

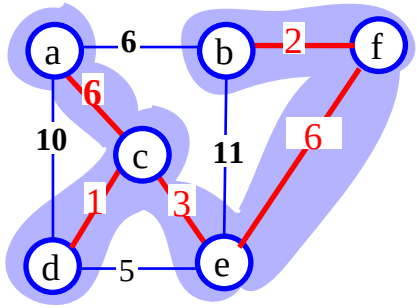


Q= 5 6 6 6 10 11
(d,e)

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



Q= 6 6 10 11
(a,c)

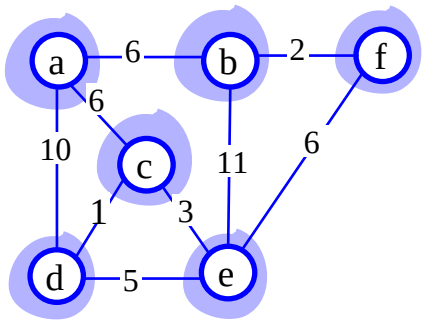


Kruskal algoritme MST

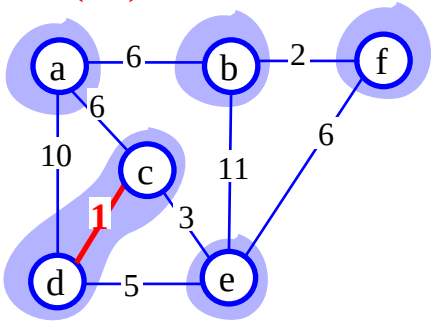
```
Kruskal(Graph G=(V,E))
   // sammenhengende, vektet
   for hver node v ε V : C(v) = {v}
   Q = PriorityQueue med alle kanter mht. vekt
   T = Ø
   while ( ! Q.isEmpty() )
      (v,u) = Q.removeMinElement();// Greedy
      if C(v) != C(u)
         legg (v,u) til T
         C(v) = C(u) = C(v) ∪ C(u)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

LI: T inneholder MSTv for hver C(v)
– før inngangen i løkka - trivielt
– rundgang : 12.25
  • hvis C(v) == C(u) :
   vekt(v,u) vekt(k) for alle k i C(v): 12.25
  • hvis C(v) != C(u) : 12.25

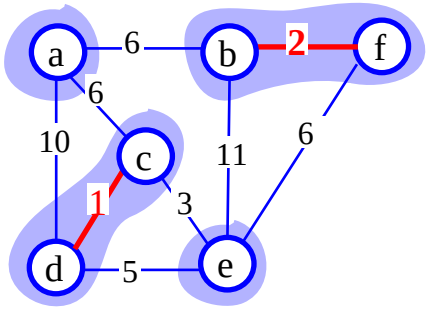
O(n +
  k log k +
  k *(log k + // heap
    C(v) != C(u) +
    C(v) ∪ C(u))
)



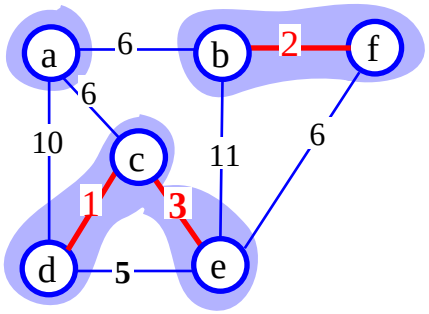
Q= 1 2 3 5 6 6 6 10 11
(c,d)



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

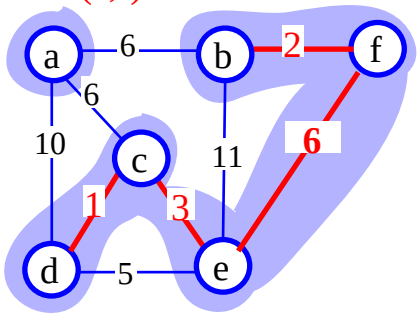


Q= 3 5 6 6 6 10 11
(c,e)

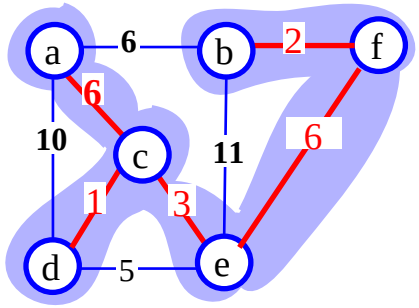


Q= 5 6 6 6 10 11
(d,e)

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



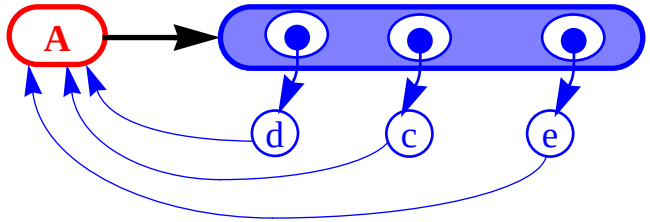
Q= 6 6 10 11
(a,c)



Implementasjon av Kruskal (finn/summer mengder)

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

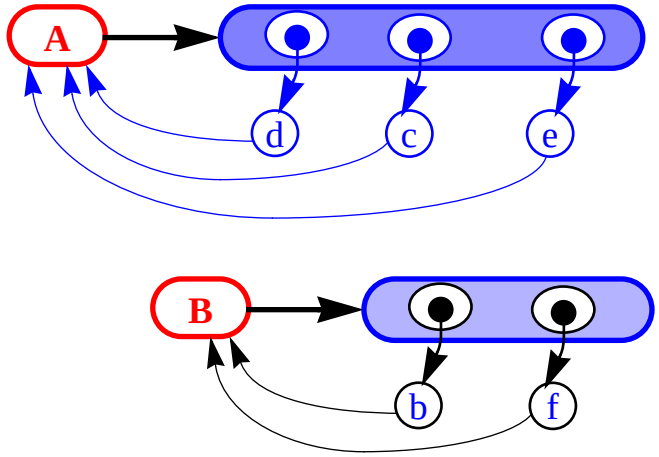
C(d) = **C**(c) = **C**(e) = **A**



Implementasjon av Kruskal (finn/summer mengder)

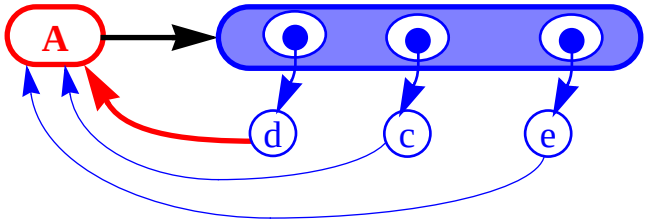
C(v) er en sekvens – hvert element har en peker til dens 'hode':

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



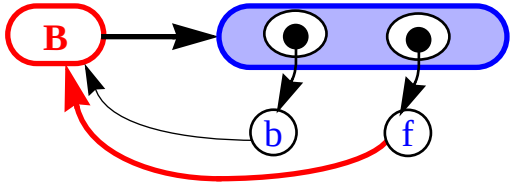
Implementasjon av Kruskal (finn/summer mengder)

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



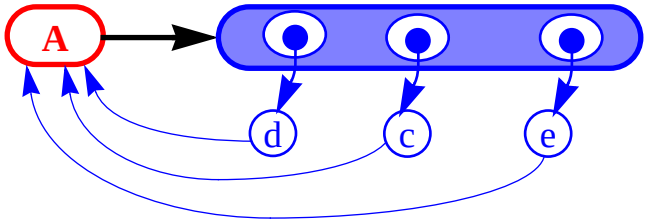
C(d) = **C**(c) = **C**(e) = **A**

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



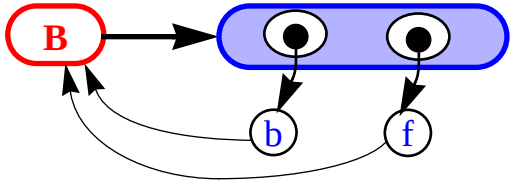
Implementasjon av Kruskal (finn/summer mengder)

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)**



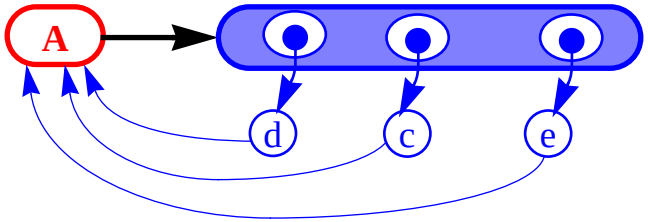
2. **C**(d) ∪ **C**(f)

for hvert element v i **C**(f)

C(d) . insertLast(v)

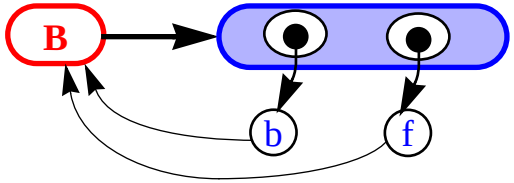
Implementasjon av Kruskal (finn/summer mengder)

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



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

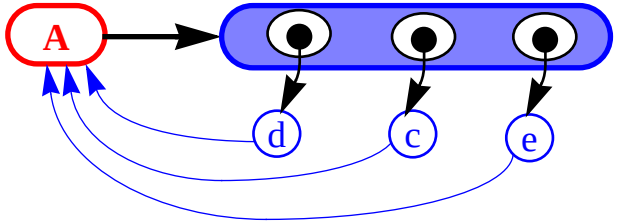
1. **C**(d) != **C**(f) er ***O*(1)**



2. **C**(d) ∪ **C**(f)

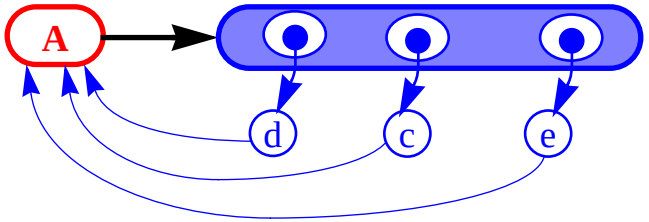
for hvert element v i **C**(f)

C(d) . insertLast(v)



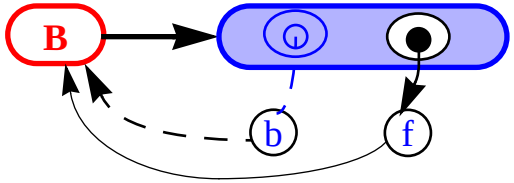
Implementasjon av Kruskal (finn/summer mengder)

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



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

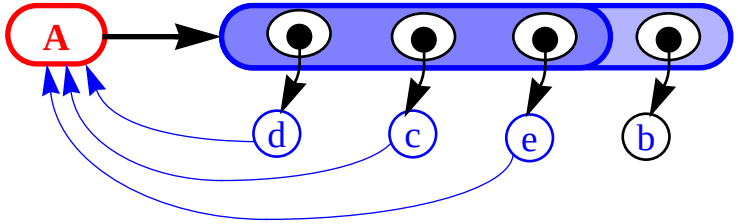
1. **C**(d) != **C**(f) er ***O*(1)**



2. **C**(d) ∪ **C**(f)

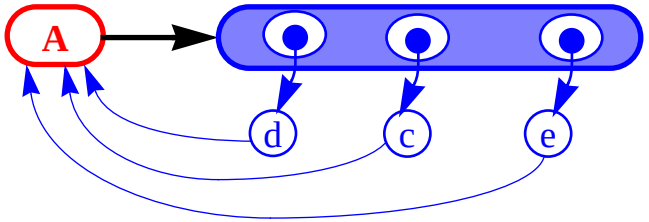
for hvert element v i **C**(f)

C(d) . insertLast(v)



Implementasjon av Kruskal (finn/summer mengder)

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



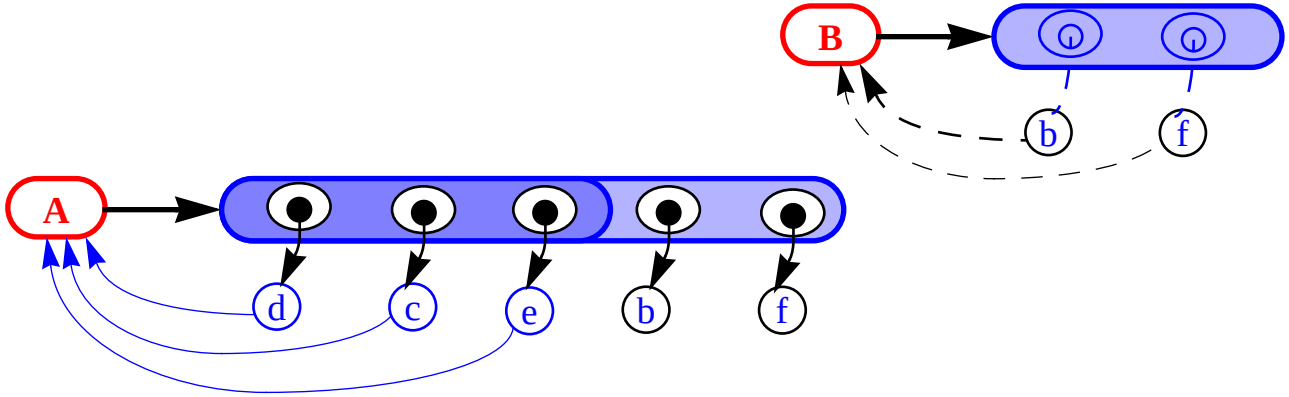
C(d) = **C**(c) = **C**(e) = **A**

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

2. **C**(d) ∪ **C**(f)

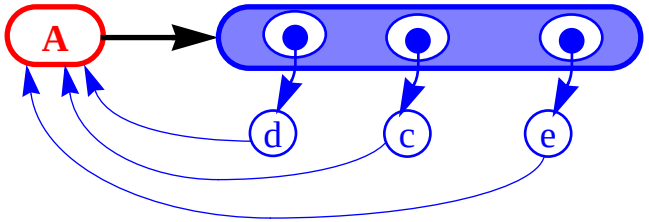
for hvert element v i **C**(f)

C(d) . insertLast(v)



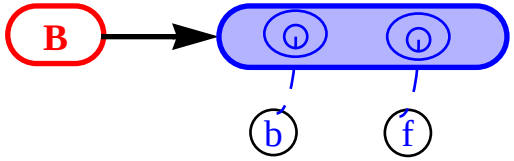
Implementasjon av Kruskal (finn/summer mengder)

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



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

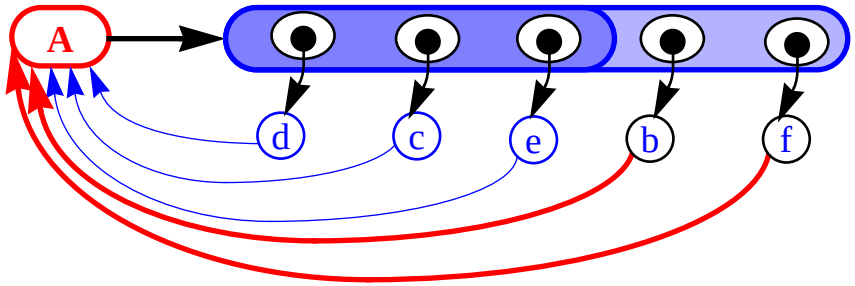
1. **C**(d) != **C**(f) er ***O*(1)**



2. **C**(d) ∪ **C**(f)

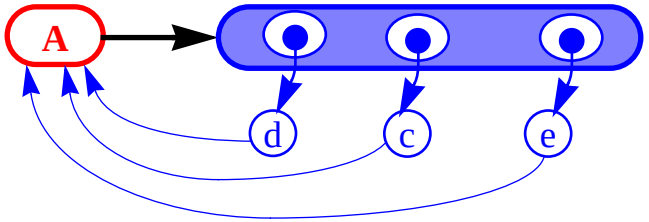
for hvert element v i **C**(f)

C(d) . insertLast(v)



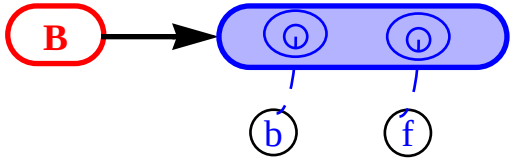
Implementasjon av Kruskal (finn/summer mengder)

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

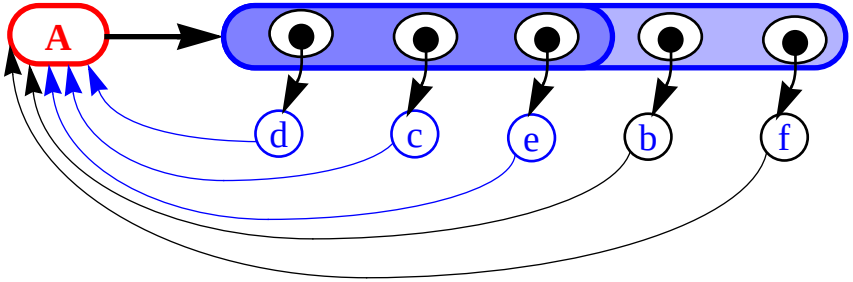


C(d) = **C**(c) = **C**(e) = **A**

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



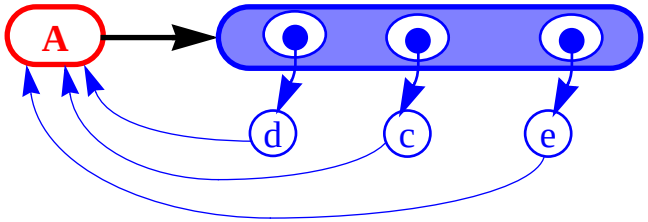
2. **C**(d) ∪ **C**(f)
for hvert element v i **C**(f)
C(d) . insertLast(v)



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

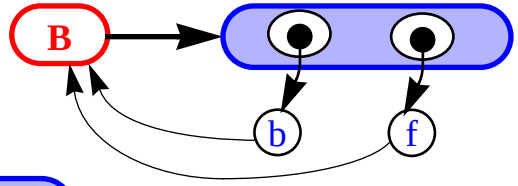
Implementasjon av Kruskal (finn/summer mengder)

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

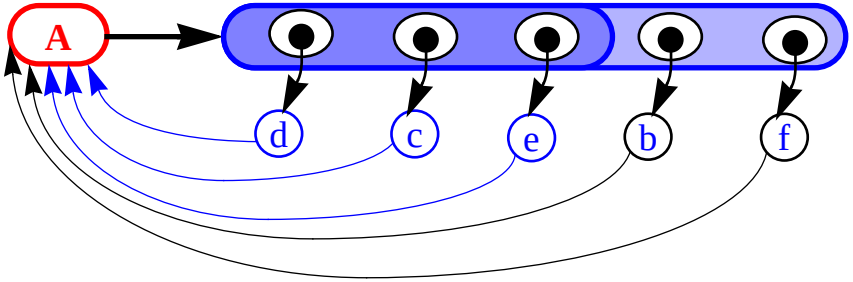


C(d) = **C**(c) = **C**(e) = **A**

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



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

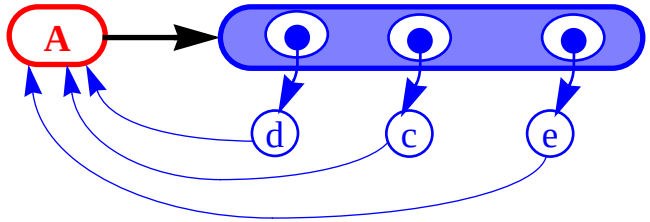


er ***O*(** min(**C**(d),**C**(f)) **)**

```
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)                   ?
```

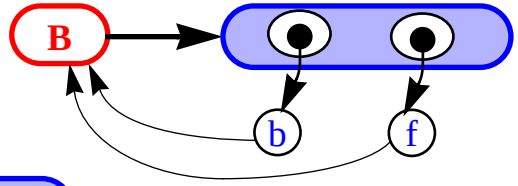
Implementasjon av Kruskal (finn/summer mengder)

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

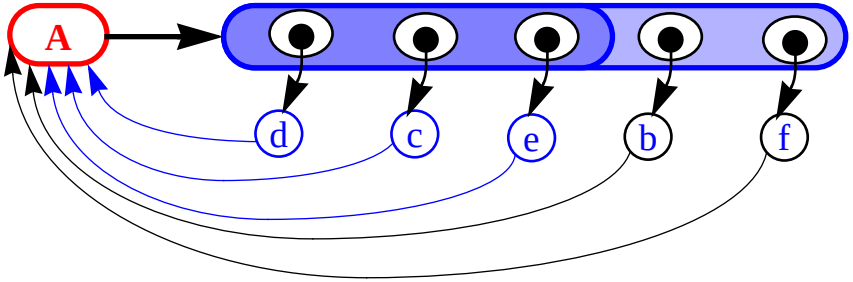


C(d) = **C**(c) = **C**(e) = **A**

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)) **)**

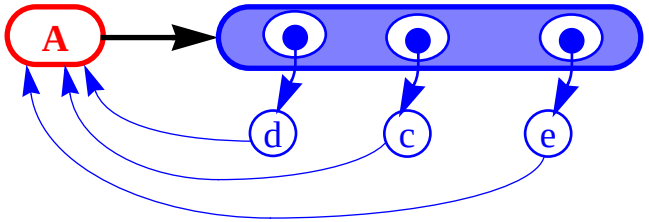
for hver node v ε V : C(v) = {v}	n
Q = PriorityQueue med alle kanter	k*logk
T = Ø	
while (! Q.isEmpty())	k*.....
(v,u) = Q.removeMinElement();	logk
if C(v) != C(u)	1
legg (v,u) til T	1
C(v) = C(u) = C(v) ∪ C(u)	**?**

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

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

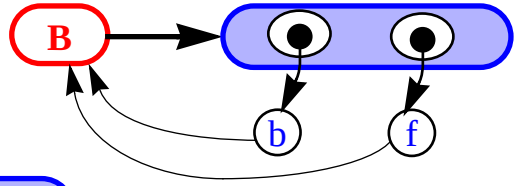
Implementasjon av Kruskal (finn/summer mengder)

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

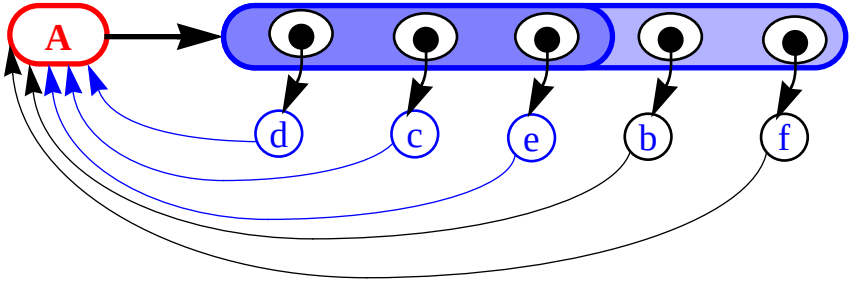


C(d) = **C**(c) = **C**(e) = **A**

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)) **)**

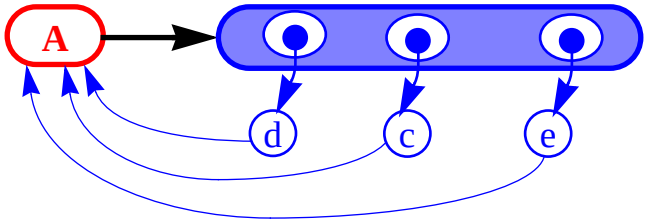
for hver node v ε V : C(v) = {v}	n
Q = PriorityQueue med alle kanter	k*logk
T = Ø	
while (! Q.isEmpty())	k*.....
(v,u) = Q.removeMinElement();	logk
if C(v) != C(u)	1
legg (v,u) til T	1
C(v) = C(u) = C(v) ∪ C(u)	**?**

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

? C(v) ∪ C(u) utføres inntil alle n noder er i samme C(i)
hver gang en node v flyttes, fordobles C(v)

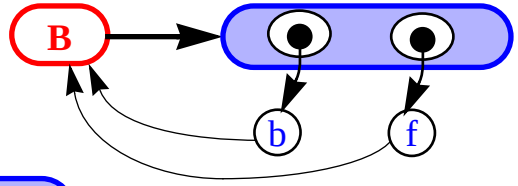
Implementasjon av Kruskal (finn/summer mengder)

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

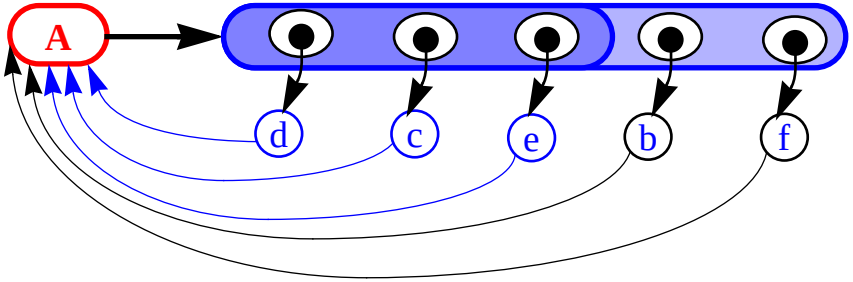


C(d) = **C**(c) = **C**(e) = **A**

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)                   ?
```

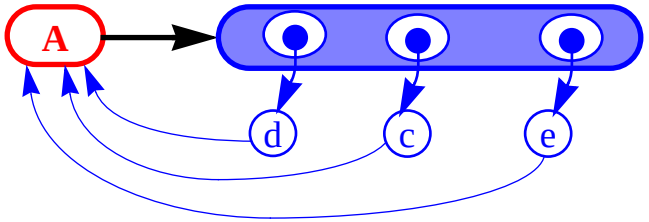
n + k*log k + k*log k + 2*k + **?** = *O*(n + k*logk + **?**)

? C(v) ∪ C(u) utføres inntil alle n noder er i samme C(i)
hver gang en node v flyttes, fordobles C(v)

= ***O*(n * log n)**

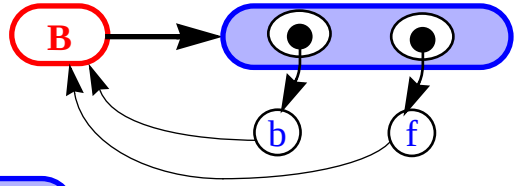
Implementasjon av Kruskal (finn/summer mengder)

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

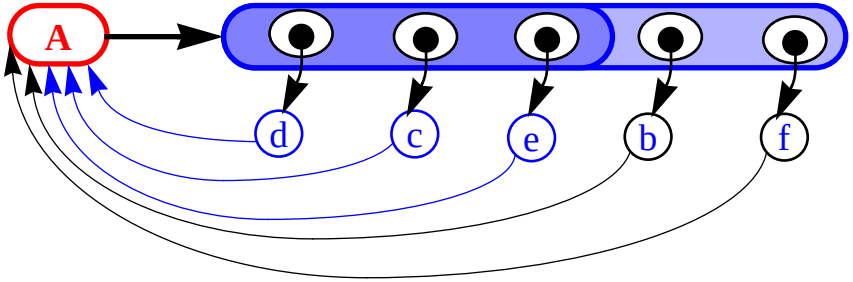


C(d) = **C**(c) = **C**(e) = **A**

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

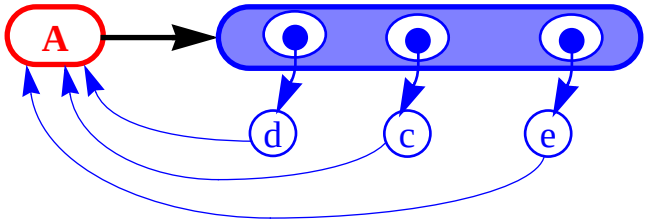
? C(v) ∪ C(u) utføres inntil alle n noder er i samme C(i)
hver gang en node v flyttes, fordobles C(v)

= *O*(n * log n)

k = *O*(n^2)

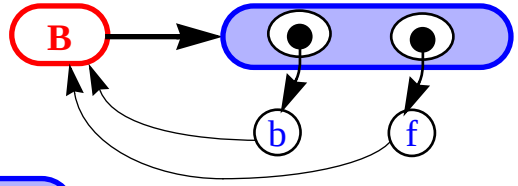
Implementasjon av Kruskal (finn/summer mengder)

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

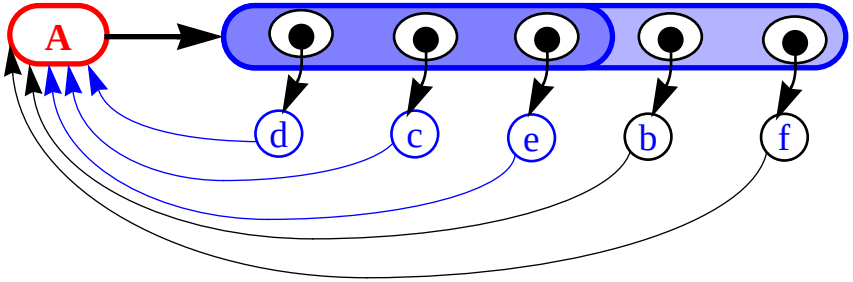


C(d) = **C**(c) = **C**(e) = **A**

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

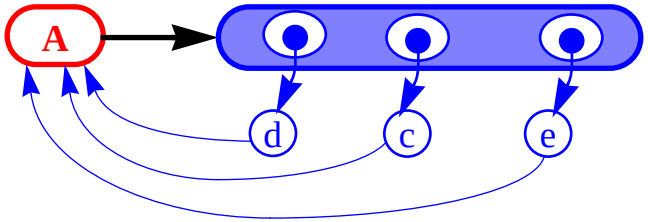
? C(v) ∪ C(u) utføres inntil alle n noder er i samme C(i)
hver gang en node v flyttes, fordobles C(v)

= *O*(n * log n)

k = *O*(n^2) -> **log k** = *O*(log n^2) = *O*(2log n) = ***O*(log n)**

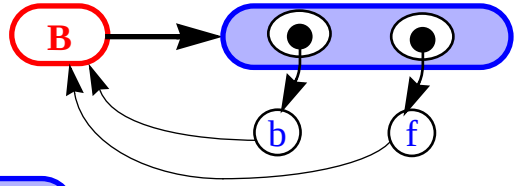
Implementasjon av Kruskal (finn/summer mengder)

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

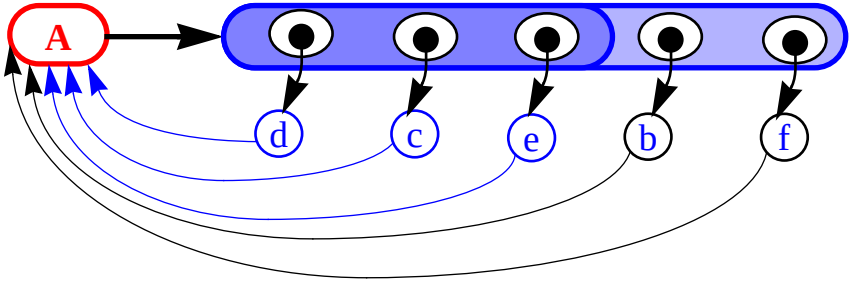


C(d) = **C**(c) = **C**(e) = **A**

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

? C(v) ∪ C(u) utføres inntil alle n noder er i samme C(i)
hver gang en node v flyttes, fordobles C(v)

= ***O*(n * log n)**

k = *O*(n^2) -> log k = *O*(log n^2) = *O*(2log n) = *O*(log n)

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

Oppsummering

- *Grafer*
  - *Graf (rettet vs. ikke-rettet), terminologi*
  - *ADT og implementasjoner (kant-liste, nabo-liste, nabo-matrise)*

Oppsummering

- *Grafer*
  - *Graf (rettet vs. ikke-rettet), terminologi*
  - *ADT og implementasjoner (kant-liste, nabo-liste, nabo-matrise)*
- ***Traversering*** *(generalisering fra Trær)*

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

  BFS

Oppsummering

- *Grafer*
  - *Graf (rettet vs. ikke-rettet), terminologi*
  - *ADT og implementasjoner (kant-liste, nabo-liste, nabo-matrise)*
- ***Traversering** (generalisering fra Trær)*

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

  BFS
- ***Algoritmer***

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

  topologisk sortering

  DAG

Oppsummering

- *Grafer*
  - *Graf (rettet vs. ikke-rettet), terminologi*
  - *ADT og implementasjoner (kant-liste, nabo-liste, nabo-matrise)*
- ***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*