F11 - Grafalgoritmer

5DV149 Datastrukturer och algoritmer
Kapitel 17

Niclas Börlin
niclas.borlin@cs.umu.se

2024-04-23 Tis

Innehåll

1. Traversering
   - Bredden-först
   - Djupet-först
2. Finna kortaste vägen
   - Från en nod till alla andra noder:
     - Dijkstras algoritm
   - Från alla noder till alla andra noder:
     - Floyds algoritm
3. Konstruera ett (minsta) uppspännande träd
   - Kruskals algoritm
   - Prims algoritm

1. Traversering av grafer

Blank

Bredden-först-traversering

Bredden-först-traversering

- Man undersöker först noden, sedan dess grannar, grannarnas grannar, osv.
- Grafen kan innehålla cykler — risk för oändlig loop
  - Markera om noden har setts
- En kö hjälper oss hålla reda på grannarna
- Endast noder till vilka det finns en väg från utgångsnoden kommer att besökas

Algoritm, bredden-först-traversering av graf

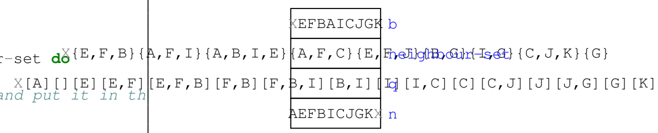
```
Algorithm g=Traverse-bf-order(n: Node, g: Graph)
// Input: A node n in a graph g to be traversed

 // Mark the starting node as seen
(n, g) ← Set-seen(n, g)
// Put it in an empty queue
q ← Enqueue(n, Queue-empty())

while not Isempty(q) do
  // Pick first node from queue
  n ← Front(q)
  q ← Dequeue(q)
  // Get its neighbours
  neighbour-set ← Neighbours(n, g)
  for each neighbour b in neighbour-set do
    if not Is-seen(b,g) then
      // Mark unseen node as seen and put it in the queue
      (b, g) ← Set-seen(b, g)
      q ← Enqueue(b,q)
```

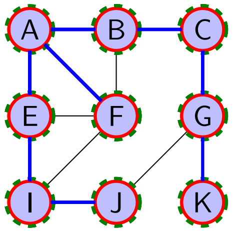

Visualiseringssymboler

- Aktuell nod markeras med röd ring
- Ljusblå färg betyder sedd (*seen*) nod
- Noder i kön markeras med grönstreckad cirkel
- Bågar som motsvarar hur vi ”upptäckte” en nod markeras med tjock blå linje
- Grannar markeras med orange ring under iterationen

Traverse-breadth-first(A,g)

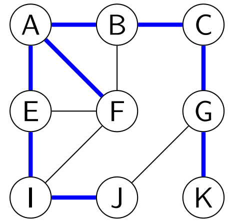
```
Algorithm g=Traverse-bf-order(n: Node, g: Graph)
// Input: A node n in a graph g to be traversed

 // Mark the starting node as seen
(n, g) <- Set-seen(n, g)
// Put it in an empty queue
q <- Enqueue(n, Queue-empty())

while not Isempty(q) do
  // Pick first node from queue
  n <- Front(q)
  q <- Dequeue(q)
  // Get its neighbours
  neighbour-set <- Neighbours(n, g)
  for each neighbour b in neighbour-set do
    if not Is-seen(b,g) then
      // Mark unseen node as seen and put it in th
      (b, g) <- Set-seen(b, g)
      q <- Enqueue(b,q)
```

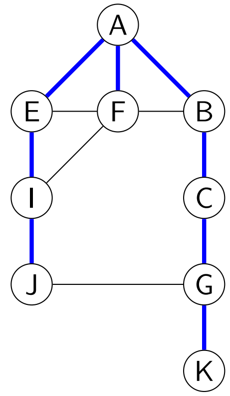




Traverse-breadth-first(A,g)

- Klar!
- Notera att de blå bågarna utgör ett *uppspännande träd*





Djupet-först-traversering

Djupet-först-traversering

- Ansats:
  1. Starta i en utgångsnod
  2. Besök dess grannar djupet-först, rekursivt
- Grafen kan innehålla cykler — risk för oändlig loop
  - Lösning: Håll reda på om noden är besökt eller ej
  - Gör rekursivt anrop endast för icke besökta noder
  - Motsvarar att undersöka en labyrint genom att markera de vägar man gått med färg
- Endast de noder man kan nå från utgångsnoden kommer att besökas

Algoritm för djupet-först-traversering av graf

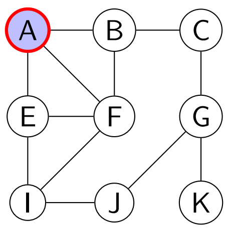
```
Algorithm Traverse-depth-first(n: Node, g: Graph)
// Input: A node n in a graph g to be traversed
// Output: The modified graph after traversal

// Mark the start node as visited.
(n, g) ← Set-visited(n, g)
// Get all its neighbours
neighbour-set ← Neighbours(n, g)
for each neighbour b in neighbour-set do
  if not Is-visited(b, g) then
    // Visit unless visited
    g ← Traverse-depth-first(b, g)
return g
```


Visualiseringssymboler

- Aktuell nod n markeras med röd ring
- Ljusblå färg betyder besökt (*visited*) nod
- Överstrukna noder i grannmängden N illustrerar noder redan behandlade i `for`-loopen
- Vid rekursivt anrop läggs aktuell nod n och grannmängden N på en stack
- Bågarna som motsvarar rekursiva anrop markeras med tjock blå linje

$g \leftarrow$Traverse-depth-first(A,g)

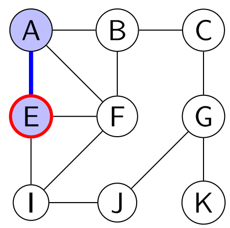
- $n \leftarrow$ A, markera som besökt
- Grannar: {E,F,B}
- E ej besökt $\rightarrow$ anropa Traverse-depth-first(E,g).



(n=A, {E,F,B})

$g \leftarrow$Traverse-depth-first(E,g)

- $n \leftarrow$ E, markera som besökt
- Grannar: {I,F,A}
- I ej besökt $\rightarrow$ anropa Traverse-depth-first(I,g).

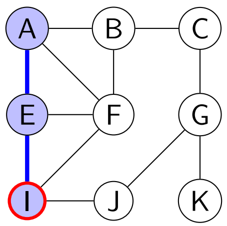


(n=E, {I,F,A})

(n=A, {E,F,B})

g ←Traverse-depth-first(I,g)

- n ← I, markera som besökt
- Grannar: {E,J,F}
- E redan besökt → Grannar: {$\cancel{E}$,J,F}
- J ej besökt → anropa Traverse-depth-first(J,g).



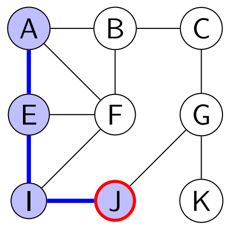
(n=I, {$\cancel{E}$,J,F})

(n=E, {I,F,A})

(n=A, {E,F,B})

g ←Traverse-depth-first(J,g)

- n ← J, markera som besökt
- Grannar: {G,I}
- G ej besökt → anropa Traverse-depth-first(G,g).



(n=J, {G,I})

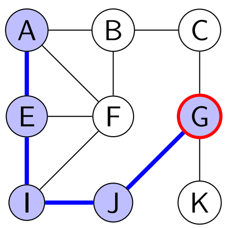
(n=I, {$\cancel{E}$,J,F})

(n=E, {I,F,A})

(n=A, {E,F,B})

g ←Traverse-depth-first(G,g)

- n ← G, markera som besökt
- Grannar: {C,K,J}
- C ej besökt → anropa Traverse-depth-first(C,g).



(n=G, {C,K,J})

(n=J, {G,I})

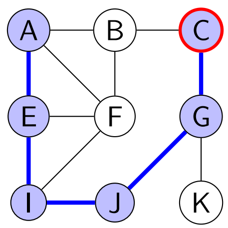
(n=I, {$\cancel{E}$,J,F})

(n=E, {I,F,A})

(n=A, {E,F,B})

g ←Traverse-depth-first(C,g)

- n ← C, markera som besökt
- Grannar: {G,B}
- G redan besökt → Grannar: {$\cancel{G}$,B}
- B ej besökt → anropa Traverse-depth-first(B,g).



(n=C, {$\cancel{G}$,B})

(n=G, {C,K,J})

(n=J, {G,I})

(n=I, {$\cancel{E}$,J,F})

(n=E, {I,F,A})

(n=A, {E,F,B})

$g \leftarrow$Traverse-depth-first(B,g)

- $n \leftarrow$ B, markera som besökt
- Grannar: {A,F,C}
- A redan besökt → Grannar: {~~A~~,F,C}
- F ej besökt → anropa Traverse-depth-first(F,g).

(n=B, {~~A~~,F,C})

(n=C, {~~G~~,B})

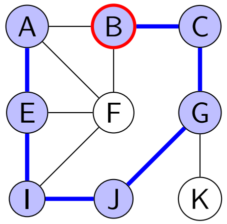
(n=G, {C,K,J})

(n=J, {G,I})

(n=I, {~~E~~,J,F})

(n=E, {I,F,A})

(n=A, {E,F,B})



$g \leftarrow$Traverse-depth-first(F,g)

- $n \leftarrow$ F, markera som besökt
- Grannar: {B,A,E,I}
- B besökt → Grannar: {~~B~~,A,E,I}
- A besökt → Grannar: {~~B~~,~~A~~,E,I}
- E besökt → Grannar: {~~B~~,~~A~~,~~E~~,I}
- I besökt → Grannar: {~~B~~,~~A~~,~~E~~,~~I~~}
- Färdig med F, återvänd

(n=F, {~~B~~,~~A~~,~~E~~,~~I~~})

(n=B, {~~A~~,F,C})

(n=C, {~~G~~,B})

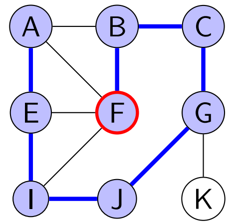
(n=G, {C,K,J})

(n=J, {G,I})

(n=I, {~~E~~,J,F})

(n=E, {I,F,A})

(n=A, {E,F,B})



$g \leftarrow$Traverse-depth-first(B,g)

- $n \leftarrow$ B, markera som besökt
- Grannar: {A,F,C}
- A redan besökt → Grannar: {~~A~~,F,C}
- F ej besökt → anropa Traverse-depth-first(F,g).
- F färdig → Grannar: {~~A~~,~~F~~,C}
- C besökt → Grannar: {~~A~~,~~F~~,~~C~~}
- Färdig med B, återvänd

(n=B, {~~A~~,~~F~~,~~C~~})

(n=C, {~~G~~,B})

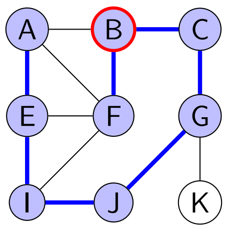
(n=G, {C,K,J})

(n=J, {G,I})

(n=I, {~~E~~,J,F})

(n=E, {I,F,A})

(n=A, {E,F,B})



$g \leftarrow$Traverse-depth-first(C,g)

- $n \leftarrow$ C, markera som besökt
- Grannar: {G,B}
- G redan besökt → Grannar: {~~G~~,B}
- B ej besökt → anropa Traverse-depth-first(B,g).
- B färdig → Grannar: {~~G~~,~~B~~}
- Färdig med C, återvänd

(n=C, {~~G~~,~~B~~})

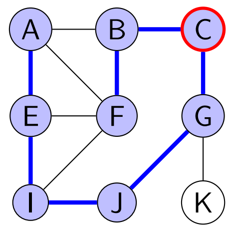
(n=G, {C,K,J})

(n=J, {G,I})

(n=I, {~~E~~,J,F})

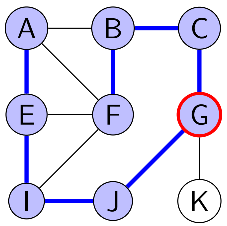
(n=E, {I,F,A})

(n=A, {E,F,B})



g ←Traverse-depth-first(G,g)

- n ← G, markera som besökt
- Grannar: {C,K,J}
- C ej besökt → anropa Traverse-depth-first(C,g).
- C färdig → Grannar: {$\cancel{C}$,K,J}
- K ej besökt → anropa Traverse-depth-first(K,g).



(n=G, {$\cancel{C}$,K,J})

(n=J, {G,I})

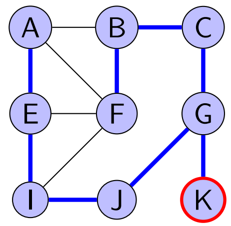
(n=I, {$\cancel{E}$,J,F})

(n=E, {I,F,A})

(n=A, {E,F,B})

g ←Traverse-depth-first(K,g)

- n ← K, markera som besökt
- Grannar: {G}
- G besökt → Grannar: {$\cancel{G}$}
- Färdig med K, återvänd



(n=K, {$\cancel{G}$})

(n=G, {$\cancel{C}$,K,J})

(n=J, {G,I})

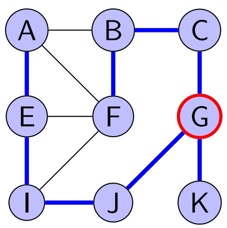
(n=I, {$\cancel{E}$,J,F})

(n=E, {I,F,A})

(n=A, {E,F,B})

g ←Traverse-depth-first(G,g)

- n ← G, markera som besökt
- Grannar: {C,K,J}
- C ej besökt → anropa Traverse-depth-first(C,g).
- C färdig → Grannar: {$\cancel{C}$,K,J}
- K ej besökt → anropa Traverse-depth-first(K,g).
- K färdig → Grannar: {$\cancel{C}$,$\cancel{K}$,J}
- J besökt → Grannar: {$\cancel{C}$,$\cancel{K}$,$\cancel{J}$}
- Färdig med G, återvänd



(n=G, {$\cancel{C}$,$\cancel{K}$,$\cancel{J}$})

(n=J, {G,I})

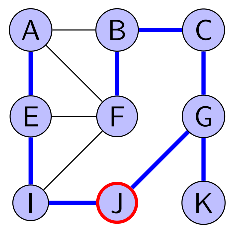
(n=I, {$\cancel{E}$,J,F})

(n=E, {I,F,A})

(n=A, {E,F,B})

g ←Traverse-depth-first(J,g)

- n ← J, markera som besökt
- Grannar: {G,I}
- G ej besökt → anropa Traverse-depth-first(G,g).
- G färdig → Grannar: {$\cancel{G}$,I}
- I besökt → Grannar: {$\cancel{G}$,$\cancel{I}$}
- Färdig med J, återvänd



(n=J, {$\cancel{G}$,$\cancel{I}$})

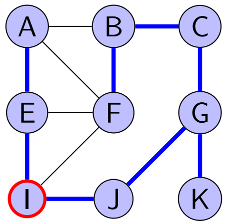
(n=I, {$\cancel{E}$,J,F})

(n=E, {I,F,A})

(n=A, {E,F,B})

g ←Traverse-depth-first(I,g)

- n ← I, markera som besökt
- Grannar: {E,J,F}
- E redan besökt → Grannar: {~~E~~,J,F}
- J ej besökt → anropa Traverse-depth-first(J,g).
- J färdig → Grannar: {~~E~~,~~J~~,F}
- F besökt → Grannar: {~~E~~,~~J~~,~~F~~}
- Färdig med I, återvänd



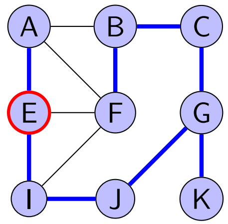
(n=I, {~~E~~,~~J~~,~~F~~})

(n=E, {I,F,A})

(n=A, {E,F,B})

g ←Traverse-depth-first(E,g)

- n ← E, markera som besökt
- Grannar: {I,F,A}
- I ej besökt → anropa Traverse-depth-first(I,g).
- I färdig → Grannar: {~~I~~,F,A}
- F besökt → Grannar: {~~I~~,~~F~~,A}
- A besökt → Grannar: {~~I~~,~~F~~,~~A~~}
- Färdig med E, återvänd

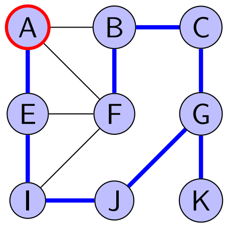


(n=E, {~~I~~,~~F~~,~~A~~})

(n=A, {E,F,B})

g ←Traverse-depth-first(A,g)

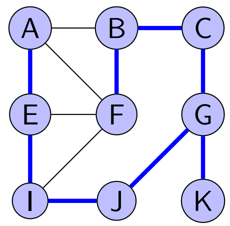
- n ← A, markera som besökt
- Grannar: {E,F,B}
- E ej besökt → anropa Traverse-depth-first(E,g).
- E färdig → Grannar: {~~E~~,F,B}
- F besökt → Grannar: {~~E~~,~~F~~,B}
- B besökt → Grannar: {~~E~~,~~F~~,~~B~~}
- Färdig med A, återvänd

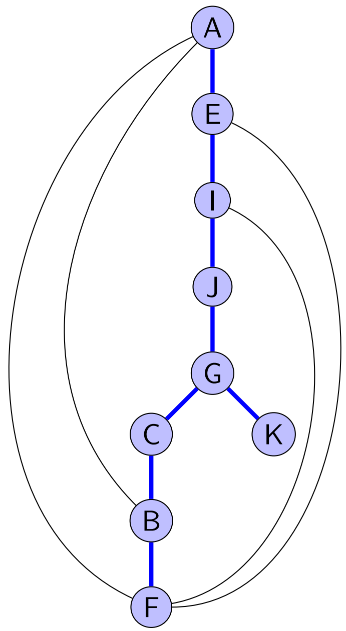


(n=A, {~~E~~,~~F~~,~~B~~})

Klart!

- Vi fick ett uppspännande träd





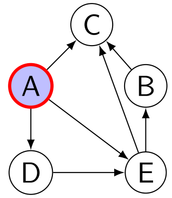
Fråga

```
Algorithm Traverse-depth-first(n: Node, g: Graph)
// Input: A node n in a graph g to be traversed
// Output: The modified graph after traversal

// Mark the start node as visited.
(n, g) ← Set-visited(n, g)
// Get all its neighbours
neighbour-set ← Neighbours(n, g)
for each neighbour b in neighbour-set do
  if not Is-visited(b, g) then
    // Visit unless visited
    g ← Traverse-depth-first(b, g)
return g
```

- Hur behöver algoritmen modifieras för att fungera på en riktad graf?
  - Inte alls!
  - Funktionen `Neighbours` hanterar det

g ←Traverse-depth-first(A,g) för riktad graf

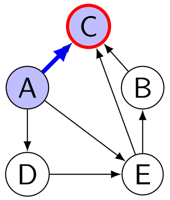
- n ← A, markera som besökt
- Grannar: {C,E,D}
- C ej besökt → anropa Traverse-depth-first(C,g).



(n=A, {C,E,D})

g ←Traverse-depth-first(C,g) för riktad graf

- n ← C, markera som besökt
- Inga grannar.
- Färdig med C, återvänd

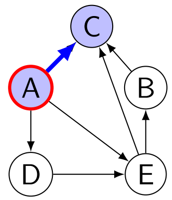


(n=C, {})

(n=A, {C,E,D})

g ←Traverse-depth-first(A,g) för riktad graf

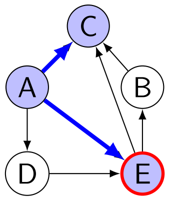
- n ← A, markera som besökt
- Grannar: {C,E,D}
- C ej besökt → anropa Traverse-depth-first(C,g).
- C färdig → Grannar: {$\cancel{C}$,E,D}
- E ej besökt → anropa Traverse-depth-first(E,g).



(n=A, {$\cancel{C}$,E,D})

$g \leftarrow$Traverse-depth-first(E,g) för riktad graf

- $n \leftarrow$ E, markera som besökt
- Grannar: {B,C}
- B ej besökt $\rightarrow$ anropa Traverse-depth-first(B,g)

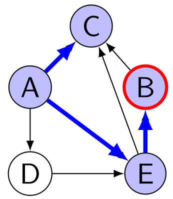


(n=E, {B,C})

(n=A, {$\cancel{C}$,E,D})

$g \leftarrow$Traverse-depth-first(B,g) för riktad graf

- $n \leftarrow$ B, markera som besökt
- Grannar: {C}
- C besökt $\rightarrow$ Grannar: {$\cancel{C}$}
- Färdig med B, återvänd



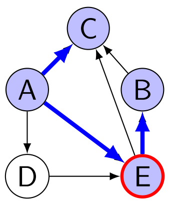
(n=B, {$\cancel{C}$})

(n=E, {B,C})

(n=A, {$\cancel{C}$,E,D})

$g \leftarrow$Traverse-depth-first(E,g) för riktad graf

- $n \leftarrow$ E, markera som besökt
- Grannar: {B,C}
- B ej besökt $\rightarrow$ anropa Traverse-depth-first(B,g)
- B färdig $\rightarrow$ Grannar: {$\cancel{B}$,C}
- C besökt $\rightarrow$ Grannar: {$\cancel{B}$,$\cancel{C}$}
- Färdig med E, återvänd

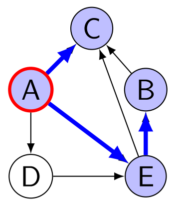


(n=E, {$\cancel{B}$,$\cancel{C}$})

(n=A, {$\cancel{C}$,E,D})

$g \leftarrow$Traverse-depth-first(A,g) för riktad graf

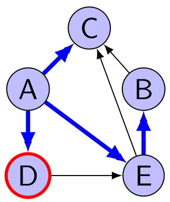
- $n \leftarrow$ A, markera som besökt
- Grannar: {C,E,D}
- C ej besökt $\rightarrow$ anropa Traverse-depth-first(C,g).
- C färdig $\rightarrow$ Grannar: {$\cancel{C}$,E,D}
- E ej besökt $\rightarrow$ anropa Traverse-depth-first(E,g).
- E färdig $\rightarrow$ Grannar: {$\cancel{C}$,$\cancel{E}$,D}
- D ej besökt $\rightarrow$ anropa Traverse-depth-first(D,g).



(n=A, {$\cancel{C}$,$\cancel{E}$,D})

$g \leftarrow$Traverse-depth-first(D,g) för riktad graf

- $n \leftarrow$ D, markera som besökt
- Grannar: {E}
- E besökt $\rightarrow$ Grannar: {$\cancel{\text{E}}$}
- Färdig med D, återvänd

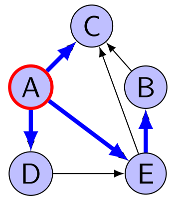


(n=D, {$\cancel{\text{E}}$})

(n=A, {$\cancel{\text{C}}$,$\cancel{\text{E}}$,D})

$g \leftarrow$Traverse-depth-first(A,g) för riktad graf

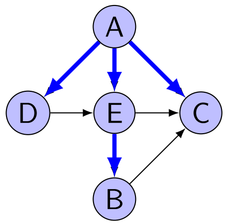
- $n \leftarrow$ A, markera som besökt
- Grannar: {C,E,D}
- C ej besökt $\rightarrow$ anropa Traverse-depth-first(C,g).
- C färdig $\rightarrow$ Grannar: {$\cancel{\text{C}}$,E,D}
- E ej besökt $\rightarrow$ anropa Traverse-depth-first(E,g).
- E färdig $\rightarrow$ Grannar: {$\cancel{\text{C}}$,$\cancel{\text{E}}$,D}
- D ej besökt $\rightarrow$ anropa Traverse-depth-first(D,g).
- D färdig $\rightarrow$ Grannar: {$\cancel{\text{C}}$,$\cancel{\text{E}}$,$\cancel{\text{D}}$}
- Färdig med A, återvänd

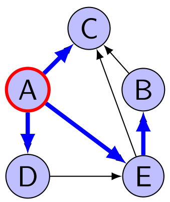


(n=A, {$\cancel{\text{C}}$,$\cancel{\text{E}}$,$\cancel{\text{D}}$})

Klar

- Även här fick vi ett *uppspännande träd*





Tidskomplexitet för Bredden-först, djupet-först-traversering

- Låt grafen ha n noder och m bågar
- Varje nod besöks exakt en gång
  - Den nodrelaterade kostnaden: $O(n)$
- För varje nod undersöker man alla bågar till grannarna
  - Kostnaden att hitta grannarna varierar:
    - Mängdorienterad specifikation:
      - $O(m)$ per nod
      - Totalt: $O(mn)$ för alla bågar
    - Navigeringsorienterade specifikation:
      - $O(grad(v))$ per nod
      - Totalt: $O(\sum_v grad(v))) = O(m)$ för alla bågar
- Total komplexitet:
  - Mängdorienterad: $O(n) + O(mn) = O(mn)$
  - Navigeringsorienterad: $O(n) + O(m) = O(m + n)$

2. Kortaste-vägen-algoritmer

Kortaste vägen

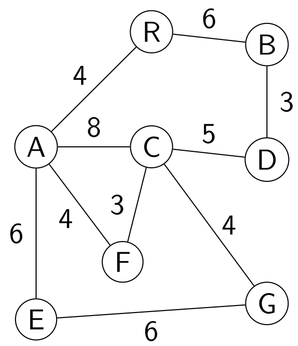
- Om grafen har lika vikt på alla bågar kan bredden-först-traversering användas för att beräkna kortaste vägen från en nod till alla andra noder
  - Krävs minimal modifiering av algoritmen:
    - Lägg till ett attribut avstånd (*distance*) till varje nod
    - Avståndet från startnoden till sig själv är 0
    - Kostnaden att gå från en nod till sin granne är 1
- För olika vikter ska vi titta på två algoritmer:
  - Floyd
    - Matrisorienterad
    - Alla-till-alla-avstånd
  - Dijkstra
    - Graforienterad, använder prioritetskö
    - En-till-alla-avstånd

Floyd's shortest path

Floyd's shortest path

- Bygger på matrisrepresentation M av grafen.
- Vid starten innehåller M de direkta avstånden mellan noderna
  - Avståndet till sig själv är 0
  - Saknas båge används ∞

	A	B	C	D	E	F	G	R
A	0	∞	8	∞	6	4	∞	4
B	∞	0	∞	3	∞	∞	∞	6
C	8	∞	0	5	∞	3	4	∞
D	∞	3	5	0	∞	∞	∞	∞
E	6	∞	∞	∞	0	∞	6	∞
F	4	∞	3	∞	∞	0	∞	∞
G	∞	∞	4	∞	6	∞	0	∞
R	4	6	∞	∞	∞	∞	∞	0



Floyds shortest path, algoritm

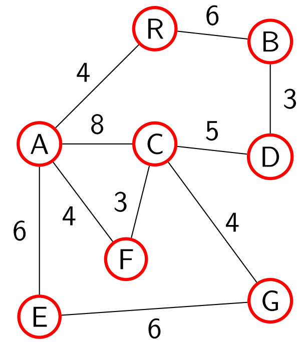
```
Algorithm Floyd-shortest-distance(g: Graph)
// Input: A graph g to find shortest paths in

// Get matrix representation of the graph
M ← Get-matrix-representation(g)
n ← Get-number-of-nodes(g)
for k=1 to n do
  for i=1 to n do
    for j=1 to n do
      if M(i,j) > M(i,k) + M(k,j) then
        // We found a shorter path from i to j
        M(i,j) = M(i,k) + M(k,j)
```

- $M(i,j)$ innehåller kortaste avståndet hittills mellan i och j
- $M(i,k) + M(k,j)$ är avståndet mellan i och j via k
- Vid slut innehåller $M(i,j)$ kortaste avståndet mellan i och j via alla noder

Floyds shortest path, exempel 1

- Vid starten
- Efter k=1 (vägar via A)
- Efter k=2 (vägar via B)
- Efter k=3 (vägar via C)
- Efter k=4 (vägar via D)
- Efter k=5 (vägar via E)
- Efter k=6 (vägar via F)
- Efter k=7 (vägar via G)
- Efter k=8 (vägar via R)

	A	B	C	D	E	F	G	R
A	0	16	7	12	6	4	11	4
B	16	0	8	3	22	11	12	6
C	7	8	0	5	10	3	4	11
D	12	3	5	0	15	8	9	9
E	6	22	10	15	0	10	6	10
F	4	11	3	8	10	0	7	8
G	11	12	4	9	6	7	0	15
R	4	6	11	9	10	8	15	0



Floyd, komplexitet

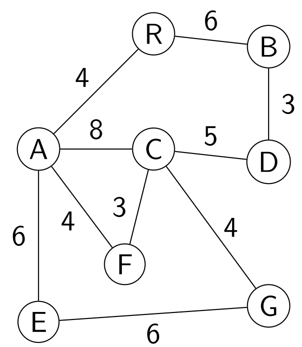
```
Algorithm Floyd-shortest-distance(g: Graph)
// Input: A graph g to find shortest paths in

// Get matrix representation of the graph
M ← Get-matrix-representation(g)
n ← Get-number-of-nodes(g)
for k=1 to n do
  for i=1 to n do
    for j=1 to n do
      if M(i,j) > M(i,k) + M(k,j) then
        // We found a shorter path from i to j
        M(i,j) = M(i,k) + M(k,j)
```

- Komplexitet?
- Trippel-loop: $O(n^3)$

Floyds shortest path, hitta kortaste vägen

- M innehåller kortaste avstånden men hur få tag på vägen?
- Modifiera algoritmen till att spara en föregångarmatris.

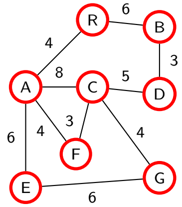
	A	B	C	D	E	F	G	R
A	0	10	7	12	6	4	11	4
B	10	0	8	3	16	11	12	6
C	7	8	0	5	10	3	4	11
D	12	3	5	0	15	8	9	9
E	6	16	10	15	0	10	6	10
F	4	11	3	8	10	0	7	8
G	11	12	4	9	6	7	0	15
R	4	6	11	9	10	8	15	0



Floyds algoritm, modifierad

```
Algorithm Floyd-shortest-path(g: Graph)
// Input: A graph g to find shortest paths in
M ← Get-matrix-representation(g)
n ← Get-number-of-nodes(g)
// Set up the initial path matrix
for i=1 to n do
  for j=1 to n do
    if i==j or M(i,j)==inf then
      // No direct path from i to j
      Path(i,j) = -1
    else
      // We came to node j from node i
      Path(i,j) = i
for k=1 to n do
  for i=1 to n do
    for j=1 to n do
      if M(i,j) > M(i,k) + M(k,j) then
        // Remember the new distance...
        M(i,j) = M(i,k) + M(k,j)
        // ...and how we came to j
        Path(i,j) = Path(k,j)
```


Floyds shortest path, exempel 2

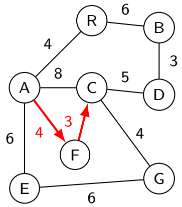


- Efter initiering
- Efter k=1 (vägar via A)
- Efter k=2 (vägar via B)
- Efter k=3 (vägar via C)
- Efter k=4 (vägar via D)
- Efter k=5 (vägar via E)
- Efter k=6 (vägar via F)
- Efter k=7 (vägar via G)
- Efter k=8 (vägar via R)

	A	B	C	D	E	F	G	R
A	0	16	7	12	6	4	11	4
B	16	0	8	3	22	11	12	6
C	7	8	0	5	10	3	4	11
D	12	3	5	0	15	8	9	9
E	6	22	10	15	0	10	6	10
F	4	11	3	8	10	0	7	8
G	11	12	4	9	6	7	0	15
R	4	6	11	9	10	8	15	0

	A	B	C	D	E	F	G	R
A	–	D	A	C	A	A	C	A
B	C	–	D	B	A	C	C	B
C	C	D	–	C	G	C	C	A
D	C	D	D	–	G	C	C	B
E	E	D	G	C	–	A	E	A
F	F	D	F	C	A	–	C	A
G	C	D	G	C	G	C	–	A
R	R	R	A	B	A	A	C	–

Floyds shortest path, hitta kortaste vägen

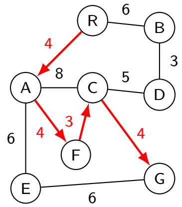


- Vad är kortaste vägen mellan A och C?

	A	B	C	D	E	F	G	R
A	0	10	7	12	6	4	11	4
B	10	0	8	3	16	11	12	6
C	7	8	0	5	10	3	4	11
D	12	3	5	0	15	8	9	9
E	6	16	10	15	0	10	6	10
F	4	11	3	8	10	0	7	8
G	11	12	4	9	6	7	0	15
R	4	6	11	9	10	8	15	0

	A	B	C	D	E	F	G	R
A	-	R	F	C	A	A	C	A
B	R	-	D	B	A	C	C	B
C	F	D	-	C	G	C	C	A
D	F	D	D	-	G	C	C	B
E	E	R	G	C	-	A	E	A
F	F	D	F	C	A	-	C	A
G	F	D	G	C	G	C	-	A
R	R	R	F	B	A	A	C	-

Floyds shortest path, hitta kortaste vägen



- Vad är kortaste vägen mellan R och G?

	A	B	C	D	E	F	G	R
A	0	10	7	12	6	4	11	4
B	10	0	8	3	16	11	12	6
C	7	8	0	5	10	3	4	11
D	12	3	5	0	15	8	9	9
E	6	16	10	15	0	10	6	10
F	4	11	3	8	10	0	7	8
G	11	12	4	9	6	7	0	15
R	4	6	11	9	10	8	15	0

	A	B	C	D	E	F	G	R
A	-	R	F	C	A	A	C	A
B	R	-	D	B	A	C	C	B
C	F	D	-	C	G	C	C	A
D	F	D	D	-	G	C	C	B
E	E	R	G	C	-	A	E	A
F	F	D	F	C	A	-	C	A
G	F	D	G	C	G	C	-	A
R	R	R	F	B	A	A	C	-

Dijkstra's shortest path

Dijkstras shortest path

- Söker kortaste vägen från en nod *n* till alla andra noder
  - Använder en prioritetskö av obesökta noder
- Fungerar enbart på grafer med positiva vikter
- Låt varje nod ha följande attribut:
  - *seen*: Sann när vi hittat en väg till noden ("sett" noden)
  - *distance*: Värdet på den hittills kortaste vägen fram till noden
  - *parent*: Referens till föregångaren längs vägen

Dijkstras shortest path, algoritm

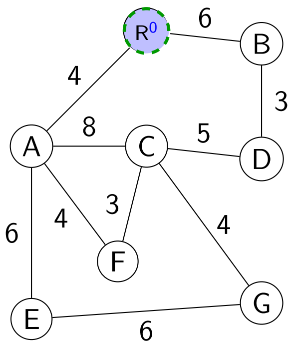
```
Algorithm Dijkstra-shortest-path(n: Node, g: Graph)
// Input: A graph g to find shortest path from node n

// Distance to start node is zero
n.distance ← 0; n.seen ← True; n.parent ← NULL
// Initialize pqueue with start node
q ← Insert(n, Pqueue-empty())
while not Isempty(q)
  // Get node with shortest distance from queue
  n ← Inspect-first(q); q ← Delete-first(q)
  nd ← n.distance
  // ...and its neighbours
  neighbour-set ← Neighbours(n, g)
  for each neighbour b in neighbour-set do
    // Compute distance to b VIA n
    d ← nd + Get-weight(n, b, g)
    if not Is-seen(b, g) then
      // We've never seen b; this is the first path to arrive at b
      b.distance ← d
      b.seen ← True
      b.parent ← n
      // Add new node to pqueue
      q ← Insert(b, q)
    else if d < b.distance then
      // We've seen b before, but path via n is shorter
      b.distance ← d
      // Update how we came to b
      b.parent ← n
      // Update the pqueue based on the new distance
      q ← Update(b, q)
```


Dijkstras shortest path, visualisering

- Symboler:
  - Aktuell nod har röd ring
  - Sedda noder är ljusblåa
  - Nodens etikett har aktuellt avstånd som exponent
  - Noder i prioritetskön har grönstreckad ring
- Prioritetskön presenteras sorterad

Dijkstras shortest path för oriktad graf

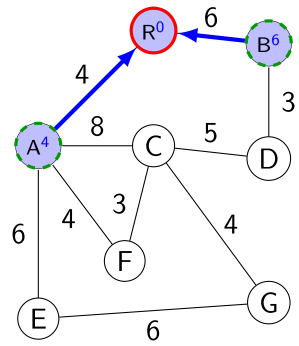
- R.seen = True
- R.distance = 0
- R.parent = NULL
- q = Insert(R(T,0,-), Pqueue-empty())
- while not Isempty(q)...



q = { R(T,0,-) }

Dijkstras shortest path för oriktad graf

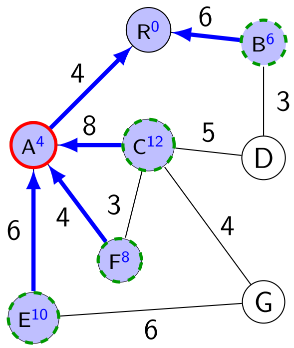
- while not Isempty(q)...
  - n = R(T,0,-); q = Delete-first(q)
  - n_d = n.distance = 0
  - neighbour-set = {A,B}
  - neighbour-set = {$\cancel{A}$,B}
  - neighbour-set = {$\cancel{A}$,$\cancel{B}$}
  - A not seen
    - d = n_d + Get-weight(n,A,g) = 4
    - A.seen = True
    - A.distance = d
    - A.parent = R
    - q = Insert(A(T,4,R),q)
  - B not seen
    - d = nd + Get-weight(n,B,g) = 6
    - B.seen = True
    - B.distance = d
    - B.parent = R
    - q = Insert(B(T,6,R),q)



q = { A(T,4,R), B(T,6,R) }

Dijkstras shortest path för oriktad graf

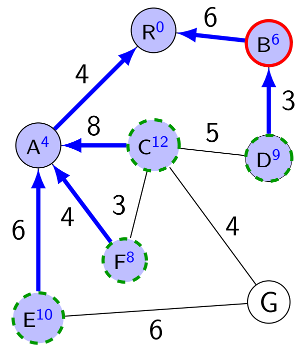
- while not Isempty(q)...
  - n = A(T,4,R); q = Delete-first(q);
  - n_d = n.distance = 4;
  - neighbour-set = {E,R,F,C};
  - neighbour-set = {$\cancel{E}$,R,F,C};
  - neighbour-set = {$\cancel{E}$,$\cancel{R}$,F,C};
  - neighbour-set = {$\cancel{E}$,$\cancel{R}$,$\cancel{F}$,C};
  - neighbour-set = {$\cancel{E}$,$\cancel{R}$,$\cancel{F}$,$\cancel{C}$};
  - E not seen
    - d = nd + Get-weight(n,E,g) = 10;
    - E.seen = True;
    - E.distance = d;
    - E.parent = A;
    - q = Insert(E(T,10,A),q);
  - R seen
    - d = nd + Get-weight(n,R,g) = 8;
    - d **not** < R.distance
  - F not seen
    - d = nd + Get-weight(n,F,g) = 8;
    - F.seen = True;
    - F.distance = d;
    - F.parent = A;



q = { B(T,6,R), F(T,8,A), E(T,10,A), C(T,12,A) }

Dijkstras shortest path för oriktad graf

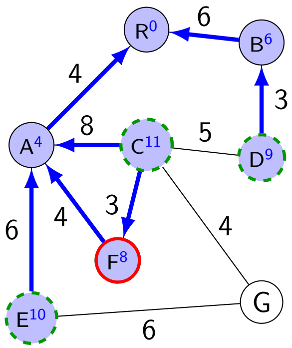
- while not Isempty(q)...
  - n = B(T,6,R); q = Delete-first(q);
  - n_d = n.distance = 6;
  - neighbour-set = {D,R};
  - neighbour-set = {$\cancel{D}$,R};
  - neighbour-set = {$\cancel{D}$,$\cancel{R}$};
  - D not seen
    - d = nd + Get-weight(n,D,g) = 9;
    - D.seen = True;
    - D.distance = d;
    - D.parent = B;
    - q = Insert(D(T,9,B),q);
  - R seen
    - d = nd + Get-weight(n,R,g) = 12;
    - d **not** < R.distance



q = { F(T,8,A), D(T,9,B), E(T,10,A), C(T,12,A) }

Dijkstras shortest path för oriktad graf

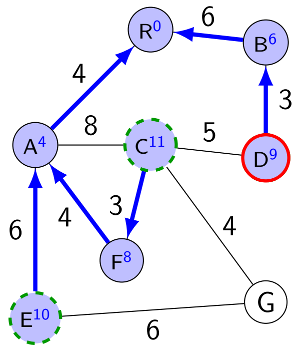
- while not Isempty(q)...
  - n = F(T,8,A); q = Delete-first(q);
  - n_d = n.distance = 8;
  - neighbour-set = {A,C};
  - neighbour-set = {$\cancel{A}$,C};
  - neighbour-set = {$\cancel{A}$,$\cancel{C}$};
  - A seen
    - d = nd + Get-weight(n,A,g) = 12;
    - d **not** < A.distance
  - C seen
    - d = nd + Get-weight(n,C,g) = 11;
    - d **is** < C.distance
      - C.distance = d;
      - C.parent = F;
      - q = update(C,q);



q = { D(T,9,B), E(T,10,A), C(T,12,A) }

Dijkstras shortest path för oriktad graf

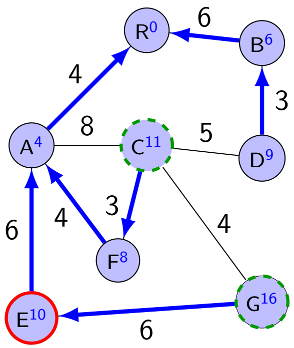
- while not Isempty(q)...
  - n = D(T,9,B); q = Delete-first(q);
  - n_d = n.distance = 9;
  - neighbour-set = {B,C};
  - neighbour-set = {$\cancel{B}$,C};
  - neighbour-set = {$\cancel{B}$,$\cancel{C}$};
  - B seen
    - d = nd + Get-weight(n,B,g) = 12;
    - d **not** < B.distance
  - C seen
    - d = nd + Get-weight(n,C,g) = 14;
    - d **not** < C.distance



q = { D(T,9,B), E(T,10,A), C(T,11,F) }

Dijkstras shortest path för oriktad graf

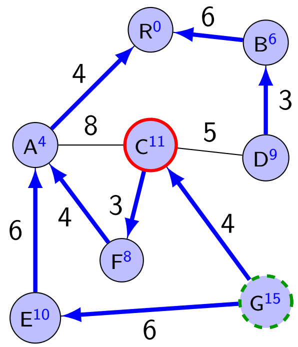
- while not Isempty(q)...
  - n = E(T,10,A); q = Delete-first(q);
  - n_d = n.distance = 10;
  - neighbour-set = {A,G};
  - neighbour-set = {$\cancel{A}$,G};
  - neighbour-set = {$\cancel{A}$,$\cancel{G}$};
  - A seen
    - d = nd + Get-weight(n,A,g) = 16;
    - d **not** < A.distance
  - G not seen
    - d = nd + Get-weight(n,G,g) = 16;
    - G.seen = True;
    - G.distance = d;
    - G.parent = E;
    - q = Insert(G(T,16,E),q);



q = { E(T,10,A), G(T,16,E) }

Dijkstras shortest path för oriktad graf

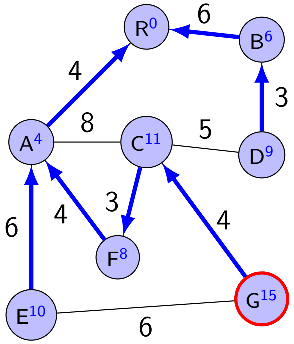
- while not Isempty(q)...
  - n = C(T,11,F); q = Delete-first(q);
  - n_d = n.distance = 11;
  - neighbour-set = {A,F,G,D};
  - neighbour-set = {$\cancel{A}$,F,G,D};
  - neighbour-set = {$\cancel{A}$,$\cancel{F}$,G,D};
  - neighbour-set = {$\cancel{A}$,$\cancel{F}$,$\cancel{G}$,D};
  - neighbour-set = {$\cancel{A}$,$\cancel{F}$,$\cancel{G}$,$\cancel{D}$};
  - A seen
    - d = nd + Get-weight(n,A,g) = 19;
    - d **not** < A.distance
  - F seen
    - d = nd + Get-weight(n,F,g) = 14;
    - d **not** < F.distance
  - G seen
    - d = nd + Get-weight(n,G,g) = 15;
    - d **is** < G.distance
      - G.distance = d;
      - G.parent = C;
      - q = update(G,q);
  - D seen
    - d = nd + Get-weight(n,D,g) = 16;



q = { G(T,16,E) }

Dijkstras shortest path för oriktad graf

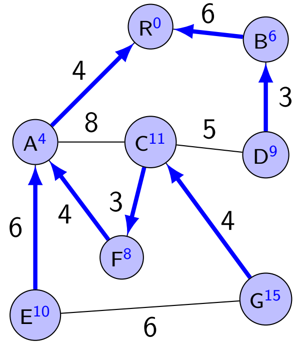
- while not Isempty(q)...
  - n = G(T,15,C); q = Delete-first(q);
  - $n_d = n$.distance = 15;
  - neighbour-set = {E,C};
  - neighbour-set = $\{\cancel{E},C\}$;
  - neighbour-set = $\{\cancel{E},\cancel{C}\}$;
  - E seen
    - d = nd + Get-weight(n,E,g) = 21;
    - d **not** < E.distance
  - C seen
    - d = nd + Get-weight(n,C,g) = 19;
    - d **not** < C.distance



$q = \{\ \}$G(T,15,C) $\}$

Dijkstras shortest path för oriktad graf

- while not Isempty(q)...
- Klar!
- Varje nod innehåller nu
  - *avståndet* till startnoden
  - *bågen* som leder tillbaka till startnoden



$q = \{\ \}$

Komplexitet?

```
Algorithm Dijkstra-shortest-path(n: Node, g: Graph)
// Input: A graph g to find shortest path from node n

// Distance to start node is zero
n.distance ← 0; n.seen ← True; n.parent ← NULL
// Initialize pqueue with start node
q ← Insert(n, Pqueue-empty())
while not Isempty(q)
  // Get node with shortest distance from queue
  n ← Inspect-first(q); q ← Delete-first(q)
  nd ← n.distance
  // ...and its neighbours
  neighbour-set ← Neighbours(n, g)
  for each neighbour b in neighbour-set do
    // Compute distance to b VIA n
    d ← nd + Get-weight(n, b, g)
    if not Is-seen(b, g) then
      // We've never seen b; this is the first path to arrive at b
      b.distance ← d
      b.seen ← True
      b.parent ← n
      // Add new node to pqueue
      q ← Insert(b, q)
    else if d < b.distance then
      // We've seen b before, but path via n is shorter
      b.distance ← d
      // Update how we came to b
      b.parent ← n
      // Update the pqueue based on the new distance
      q ← Update(b, q)
```


Dijkstras shortest path, komplexitet

- Vi sätter in varje nod i prioritetskön en gång:
  - Totalt $n \cdot O$(Insert)
- Vi läser av varje nod i prioritetskön en gång
  - Totalt $n \cdot O$(Inspect-first)
- Vi tar ut varje nod ur prioritetskön en gång
  - Totalt $n \cdot O$(Delete-first)
- Vi kan behöva uppdatera element i prioritetskön
  - Maximalt m gånger: $m \cdot O$(update)
- Totalt för olika konstruktioner av prioritetskön:
  - Osorterad lista (av referenser till noderna):
    - $nO(1) + nO(n) + nO(n) + mO(1) = O(n^2 + m)$
  - Sorterad lista:
    - $nO(n) + nO(1) + nO(1) + mO(n) = O(n^2 + mn)$
  - Heap:
    - $nO(\log n) + nO(1) + nO(\log n) + mO(\log n) = O((n + m)\log n)$
- Heap är snabbast!

Komplexitet, kortaste vägen

- En-till-alla:
  - Floyd: $O(n^3)$ (finns ej i en-till-alla-version)
  - Dijkstra: $O((n+m)\log n)$
- Alla-till-alla:
  - Floyd: $O(n^3)$
  - Dijkstra: $O((n+m)\log n)$ för en-till-alla
    - Måste köras n gånger för att få alla-till-alla:
      - $nO((n+m)\log n) = O(n^2 \log n + mn \log n)$
    - För gles graf $m \approx n$: $O(n^2 \log n)$
    - För tät graf $m \approx n^2$: $O(n^3 \log n)$
  - Djikstra snabbare på stora, glesa grafer

Blank

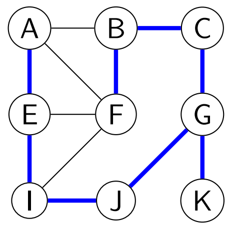
Blank

Blank

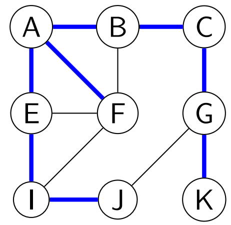
3. Minsta uppspännande träd

Uppspännande träd, oviktad graf

- Både bredden-först och djupet-först-traverseringarna gav oss uppspännande träd:
  - Djupet-först:



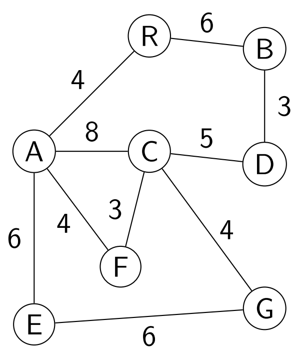
  - Bredden-först:



- Har träden minimal längd?
  - För oviktade grafer — ja!
    - Längd = $n - 1$
  - Om varje kant har samma vikt är alla uppspännande träd minimala

Uppspännande träd, viktad graf

- Hur hanterar man grafer med vikter?
  - Exempel: Bygga fibernät mellan byar
    - Vikten på bågen motsvarar kostnaden att dra fiber mellan grannbyarna
  - Man söker ett uppspännande träd med minsta möjliga totala längd
    - Det är alltså inte en kortaste-vägen-algoritm
  - För mängdorienterad specifikation finns Kruskals algoritm
  - För navigeringsorienterad specifikation finns Prims algoritm



Blank

Kruskals algoritm

Kruskals algoritm för minsta uppspännande träd

- Utgå från en prioritetskö av alla bågar
- I varje steg, plocka kortaste bågen från kön
  - Fyra alternativ:
    1. Bilda nytt träd
    2. Bygg ut ett träd
    3. Ignorera bågen
    4. Slå ihop två träd
- Under algoritmens gång kan vi ha en skog
- Till slut har vi bara ett träd (för sammanhängande gra)fc
- Vår beskrivning använder färger för att hålla i sär träden

Kruskals algoritm för minsta uppspännande träd, algoritm

- Låt alla noder sakna färg
- Stoppa in alla bågarna i en prioritetskö q, sorterade efter vikt
- Upprepa tills q är tom:
  0. Ta första bågen ur q
  1. Om ingen av noderna är färgade:
     - Färglägg med ny färg (bilda nytt träd)
  2. Om endast en nod är färgad:
     - Färglägg den ofärgade noden (utöka trädet)
  3. Om bägge noderna har samma färg:
     - Ignorera bågen (den skulle skapa en cykel)
  4. Om noderna har olika färg
     - Välj en av färgerna och färga om det nya gemensamma trädet (slå ihop träden)

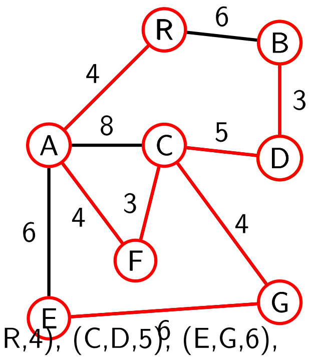
Kruskals algoritm för minsta uppspännande träd, exempel

- Upprepa tills kön är tom:
- Klar!
  - Ta första bågen (C,F,3) ur kön
  - Ingen av (C,F) är färgade:
    - Färglägg med ny färg (fall 1)
  - Ta första bågen (B,D,3) ur kön
  - Ingen av (B,D) är färgade:
    - Färglägg med ny färg (fall 1)
  - Ta första bågen (C,G,4) ur kön
  - C är färgad
    - Färglägg med C:s färg (fall 2)
  - Ta första bågen (A,F,4) ur kön
  - F är färgad
    - Färglägg med F:s färg (fall 2)
  - Ta första bågen (A,R,4) ur kön
  - A är färgad
    - Färglägg med A:s färg (fall 2)
  - Ta första bågen (C,D,5) ur kön
  - C och D färgade med olika färg
    - Färglägg bägge graferna med C:s färg

q={ (C,F,3), (B,D,3), (C,G,4), (A,F,4), (A,R,4), (C,D,5), (E,G,6), (B,R,6), (A,E,6), (A,C,8) } q={ (B,D,3), (C,G,4), (A,F,4), (A,R,4), (C,D

(A,E,6), (A,C,8) } q={ (C,G,4), (A,F,4), (A,R,4), (C,D,5), (E,G,6), (B,R

(A,C,8) } q={ (A,F,4), (A,R,4), (C,D,5), (E,G,6), (B,R,6), (A,E,6), (A,C



Kruskals algoritm, komplexitet

- Bygg upp en prioritetskö utifrån en bågmängd
  - $O(m \log m)$ om heap
- Varje båge traverseras en gång: $O(m)$:
  - Hanteringen av bågen kan delas in i fyra fall:
    - Ingen nod färgad: $O(1)$
    - En nod färgad: $O(1)$
    - Noderna samma färg: $O(1)$
    - Noderna olika färg:
      - Naiv lösning: $O(n)$
      - Effektiv lösning $O(1)$
- Total komplexitet:
  - $O(m \log m) + O(m) = O(m \log m) = O(m \log n)$

Kruskals algoritm för minsta uppspännande träd, naiv

```
Algorithm Kruskal(g: Graph)
next-color ← 1; q = Pqueue-empty()
for each node n in g do
  n.color ← 0
for each edge e in g do
  q ← Insert(q,e)
while not Isempty(q) do
  e = (a,b) ← Inspect-first(q); q ← Delete-first(q)
  if a.color = b.color then // same color
    if a.color = 0 then // uncolored
      a.color ← next-color
      b.color ← next-color
      next-color ← next-color + 1
    else
      // same but color!=0, do nothing
  else // different colors
    if a.color = 0 then // b colored, not a
      a.color ← b.color
    else if b.color = 0 then // a colored, not b
      b.color ← a.color
    else // both colored with different colors
      for each node n in g do
        if n.color = b.color then
          n.color ← a.color
```


”Omfärgning” av delgraf

- En naiv algoritm för omfärgning av ett träd/delgraf måste traversera alla noderna i delgrafen: $O(n)$
- Effektivare att definiera om likhet för färger
- Använd ett fält E med ekvivalenta färger

Kruskals algoritm för minsta uppspännande träd, effektiv

```
Algorithm Kruskal(g: Graph)
next-color ← 1; q = Pqueue-empty(); E(0) = 0
for each node n in g do
  n.color ← 0
for each edge e in g do
  q ← Insert(q,e)
while not Isempty(q) do
  e = (a,b) ← Inspect-first(q); q ← Delete-first(q)
  if E(a.color) = E(b.color) then // same color
    if a.color = 0 then // uncolored
      a.color ← next-color
      b.color ← next-color
      E(next-color) ← next-color
      next-color ← nextColor + 1
    else
      // same but color!=0, do nothing
  else // different colors
    if a.color = 0 then // b colored, not a
      a.color ← b.color
    else if b.color = 0 then // a colored, not b
      b.color ← a.color
    else // both colored with different colors
      E(a.color) ← min(E(a.color), E(b.color))
      E(b.color) ← min(E(a.color), E(b.color))
```

Fråga

- Hur fungerar Kruskals algoritm på en riktad graf?
  - Samma som oriktad!
- Hur fungerar Kruskals algoritm på en icke sammanhängade graf?
  - Resultatet blir en skog!

Blank

Blank

Blank

Prims algoritm

Prims algoritm för minsta uppspännande träd (1)

- Utgå från godtycklig startnod
- I varje steg, bygg på trädet med en båge med minimal vikt
- Använd en prioritetskö för att hålla reda på vilka bågar som kan vara aktuella
- Till slut spänner trädet upp grafen (eller en sammanhängande komponent av den)

Prims algoritm för minsta uppspännande träd (2)

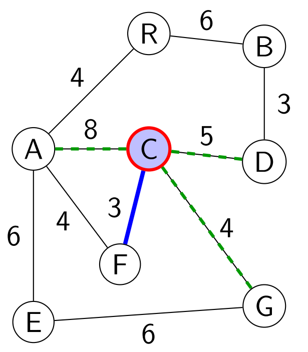
- Välj godtycklig startnod n ur grafen och låt n bli rot i trädet
- Skapa en tom prioritetskö q
- Upprepa:
  - Fas 0:
    - Markera n som stängd
  - Fas 1: Lägg till nya bågar till prioritetskön:
    - För var och en av de öppna (icke-stängda) grannarna w till n:
      - Lägg bågen (n, w, d) i prioritetskön q
  - Fas 2: Hitta bästa bågen att lägga till trädet:
    - Upprepa:
      - Ta första bågen (n, w, d) ur q
      - Om destinationsnoden w är öppen:
        - Lägg till bågen (n, w, d) till trädet

      tills w öppen (lagt till en båge) eller q tom (klara)
  - Fas 3: Gå till den nya noden
    - Låt $n = w$

  tills q är tom

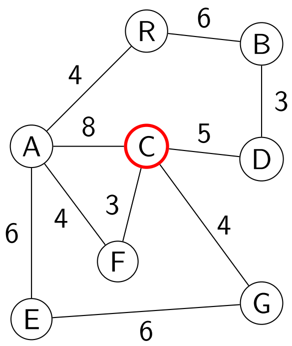
Symboler

- Stängda noder färgas ljusblått
- Aktuell nod ritas med röd cirkel
- Bågar i prioritetskön ritas grönstreckade
- Prioritetskön presenteras sorterad
- Bågar i den nuvarande trädet ritas i mörkblått



Prims minsta uppspännande träd, exempel

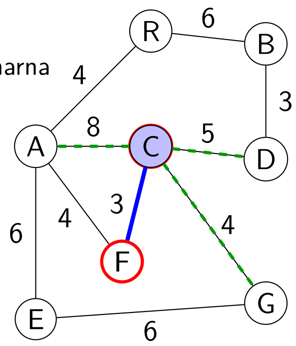
- $n \leftarrow$ C.
- Låt n blir rot i trädet.
- Skapa en tom prioritetskö q.
- Upprepa:



q={ }

Prims minsta uppspännande träd, exempel

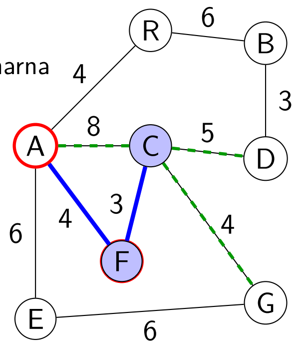
- Upprepa:
  - Fas 0: Markera C som stängd.
  - Fas 1: För var och en av de öppna grannarna {A,F,G,D} till C:
    - Lägg (C,A,8) till q.
    - Lägg (C,F,3) till q.
    - Lägg (C,G,4) till q.
    - Lägg (C,D,5) till q.
  - Fas 2: Upprepa
    - Ta (n, w, d)=(C,F,3) från q.
    - F ej stängd.
      - Lägg (C,F,3) till trädet.
  - tills F ej stängd eller q är tom.
  - Fas 3: $n \leftarrow$ F.
- tills q är tom.



q={ (C,F,3), (C,G,4), (C,D,5), (C,A,8) }

Prims minsta uppspännande träd, exempel

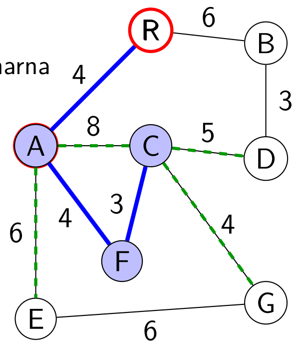
- Upprepa:
  - Fas 0: Markera F som stängd.
  - Fas 1: För var och en av de öppna grannarna {A} till F:
    - Lägg (F,A,4) till q.
  - Fas 2: Upprepa
    - Ta (n, w, d)=(F,A,4) från q.
    - A ej stängd.
      - Lägg (F,A,4) till trädet.
  - tills A ej stängd eller q är tom.
  - Fas 3: $n \leftarrow$ A.
- tills q är tom.



q={ (F,A,4), (C,G,4), (C,D,5), (C,A,8) }

Prims minsta uppspännande träd, exempel

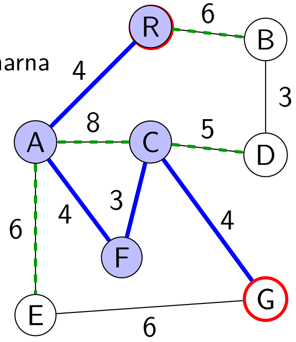
- Upprepa:
  - Fas 0: Markera A som stängd.
  - Fas 1: För var och en av de öppna grannarna {R,E} till A:
    - Lägg (A,R,4) till q.
    - Lägg (A,E,6) till q.
  - Fas 2: Upprepa
    - Ta (n, w, d)=(A,R,4) från q.
    - R ej stängd.
      - Lägg (A,R,4) till trädet.
  - tills R ej stängd eller q är tom.
  - Fas 3: $n \leftarrow$ R.
- tills q är tom.



q={ (A,R,4), (C,G,4), (C,D,5), (A,E,6), (C,A,8) }

Prims minsta uppspännande träd, exempel

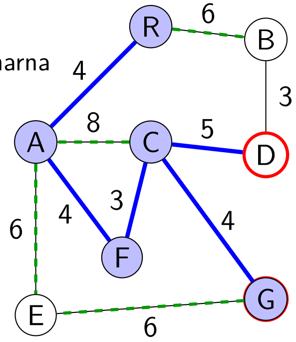
- Upprepa:
  - Fas 0: Markera R som stängd.
  - Fas 1: För var och en av de öppna grannarna {B} till R:
    - Lägg (R,B,6) till *q*.
  - Fas 2: Upprepa
    - Ta (n, w, d)=(C,G,4) från *q*.
    - G ej stängd.
      - Lägg (C,G,4) till trädet.
  - tills G ej stängd eller *q* är tom.
  - Fas 3: $n \leftarrow$ G.
- tills *q* är tom.



q={ (C,D,5), (R,B,6), (A,E,6), (C,A,8) }

Prims minsta uppspännande träd, exempel

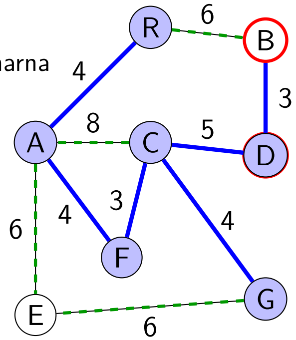
- Upprepa:
  - Fas 0: Markera G som stängd.
  - Fas 1: För var och en av de öppna grannarna {E} till G:
    - Lägg (G,E,6) till *q*.
  - Fas 2: Upprepa
    - Ta (n, w, d)=(C,D,5) från *q*.
    - D ej stängd.
      - Lägg (C,D,5) till trädet.
  - tills D ej stängd eller *q* är tom.
  - Fas 3: $n \leftarrow$ D.
- tills *q* är tom.



q={ (G,E,6), (R,B,6), (A,E,6), (C,A,8) }

Prims minsta uppspännande träd, exempel

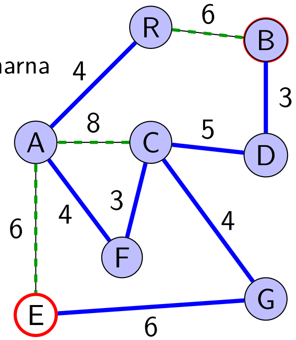
- Upprepa:
  - Fas 0: Markera D som stängd.
  - Fas 1: För var och en av de öppna grannarna {B} till D:
    - Lägg (D,B,3) till *q*.
  - Fas 2: Upprepa
    - Ta (n, w, d)=(D,B,3) från *q*.
    - B ej stängd.
      - Lägg (D,B,3) till trädet.
  - tills B ej stängd eller *q* är tom.
  - Fas 3: $n \leftarrow$ B.
- tills *q* är tom.



q={ (G,E,6), (R,B,6), (A,E,6), (C,A,8) }

Prims minsta uppspännande träd, exempel

- Upprepa:
  - Fas 0: Markera B som stängd.
  - Fas 1: För var och en av de öppna grannarna { } till B:
  - Fas 2: Upprepa
    - Ta (n, w, d)=(G,E,6) från *q*.
    - E ej stängd.
      - Lägg (G,E,6) till trädet.
  - tills E ej stängd eller *q* är tom.
  - Fas 3: $n \leftarrow$ E.
- tills *q* är tom.



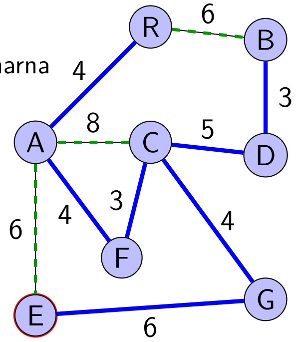
q={ (R,B,6), (A,E,6), (C,A,8) }

Prims minsta uppspännande träd, exempel

- Upprepa:
  - Fas 0: Markera E som stängd.
  - Fas 1: För var och en av de öppna grannarna { } till E:
  - Fas 2: Upprepa
    - Ta (n, w, d)=(R,B,6) från q.
    - B stängd.
  - tills B ej stängd eller q är tom.
    - Ta (n, w, d)=(A,E,6) från q.
    - E stängd.
  - tills E ej stängd eller q är tom.
    - Ta (n, w, d)=(C,A,8) från q.
    - A stängd.
  - tills A ej stängd eller q är tom.
  - Fas 3: $n \leftarrow A$

q={ (R,B,6), (A,E,6), (C,A,8) }

- tills q är tom.
- Klar!



Prims algoritm för minsta uppspännande träd (igen)

- Välj godtycklig startnod n ur grafen och låt n bli rot i trädet
- Skapa en tom prioritetskö q
- Upprepa:
  - Markera n som stängd
  - För var och en av de öppna grannarna w till n:
    - Lägg bågen (n, w, d) i prioritetskön q
  - Upprepa:
    - Ta första bågen (n, w, d) ur q
    - Om destinationsnoden w ej är stängd:
      - Lägg till bågen (n, w, d) till trädet

    tills w ej stängd eller q är tom
  - Låt $n = w$

  tills q är tom
- Vad blir komplexiteten?

Prims algoritm, komplexitet

- Man gör en traversering av grafen, dvs. $O(m) + O(n)$
- Sen tillkommer kööperationer:
  - För varje båge:
    - Sätt in ett element i prioritetskön
    - Inspektera elementet
    - Ta ut elementet
  - Komplexitet: $O(m)$ (lista) eller $O(\log m)$ (heap).
- Totalt: $O(n) + O(m^2)$ eller $O(n) + O(m \log m)$

Fråga

- Hur fungerar Prims algoritm på en riktad graf?
  - Vi får ett träd som spänner upp noderna "nedströms" startnoden
- Hur fungerar Prims algoritm på en icke sammanhängade graf?
  - Oriktad graf:
    - Vi får ett träd som spänner upp den sammanhängande komponent som startnoden ingick i
  - Riktad graf:
    - Vi får ett träd som spänner upp den sammanhängande komponent som är "nedströms" startnoden