Föreläsning 9

Grafalgoritmer

Grafalgoritmer

- Traversering
  - Bredden-först och djupet-först
- Konstruera ett (minsta) uppspännande träd
- Finna vägarna från en nod till alla andra noder
  - Kortaste vägen mellan två noder
- Finna maximala flödet
  - Finna det maximala flödet mellan två noder

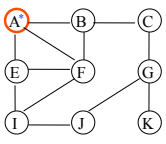
Djupet-först-traversering

- Man besöker utgångsnoden och sedan dess grannar djupet-först rekursivt.
  - Undersöka en labyrint genom att markera de vägar man gått med färg.
- Om grafen innehåller cykler finns det risk för oändlig traversering
  - Löses genom att hålla reda på om noden är besökt eller ej. Om noden redan besökt görs ingen rekursivt anrop.
  - Endast de noder man kan nå från utgångsnoden kommer att besökas.

Djupet-först-algoritm:

```
Algoritm depthFirst(Node n, Graph g)
     input: A node n in a graph g to be traversed
visited(n, g) // Marks the node as visited
neighbourSet ← neighbours(n, g);
for each neighbour in neighbourSet do
   if not isVisited(neighbour)
         depthFirst(neighbour, g)
```

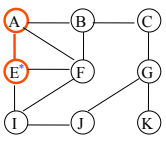

depthFirst(A)



Markera noden som besökt.
Grannar = {E, F, B}
E ej besökt, rekursivt anrop.

depthFirst(A)

depthFirst(E)

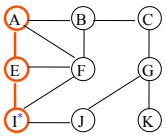


Markera noden som besökt.
Grannar = {I, F, A}
I ej besökt, rekursivt anrop.

depthFirst(A)

depthFirst(E)

depthFirst(I)



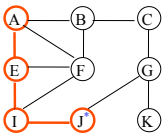
Markera noden som besökt.
Grannar = {J, F, E}
J ej besökt, rekursivt anrop.

depthFirst(A)

depthFirst(E)

depthFirst(I)

depthFirst(J)



Markera noden som besökt.
Grannar = {G, I}
G ej besökt, rekursivt anrop.

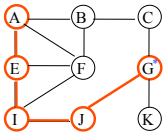
depthFirst(A)

depthFirst(E)

depthFirst(I)

depthFirst(J)

depthFirst(G)



Markera noden som besökt.
Grannar = {C, K,J}
C ej besökt, rekursivt anrop.

depthFirst(A)

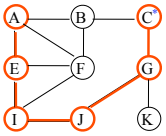
depthFirst(E)

depthFirst(I)

depthFirst(J)

depthFirst(G)

depthFirst(C)



Markera noden som besökt.
Grannar = {B, G}
B ej besökt, rekursivt anrop.

depthFirst(A)

depthFirst(E)

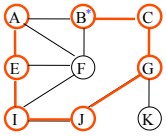
depthFirst(I)

depthFirst(J)

depthFirst(G)

depthFirst(C)

depthFirst(B)



Markera noden som besökt.
Grannar = {A, F, C}
A redan besökt
F ej besökt, rekursivt anrop.

depthFirst(A)

depthFirst(E)

depthFirst(I)

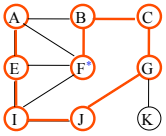
depthFirst(J)

depthFirst(G)

depthFirst(C)

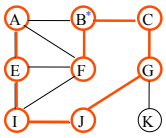
depthFirst(B)

depthFirst(F)



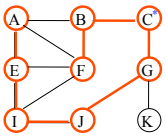
Markera noden som besökt.
Grannar = {B, A, E, I}
Alla redan besökta.

depthFirst(A)
depthFirst(E)
depthFirst(I)
depthFirst(J)
depthFirst(G)
depthFirst(C)
depthFirst(B)



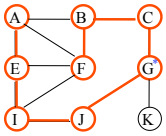
Nod B:
Grannar = {A, F, C}
Alla redan besökta

depthFirst(A)
depthFirst(E)
depthFirst(I)
depthFirst(J)
depthFirst(G)
depthFirst(C)



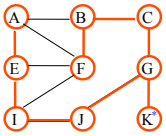
Nod C:
Grannar = {B, G}
Alla redan besökta

depthFirst(A)
depthFirst(E)
depthFirst(I)
depthFirst(J)
depthFirst(G)



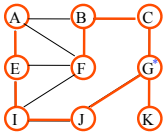
Nod G:
Grannar = {C, K, J}
K ej besökt, rekursivt anrop.

depthFirst(A)
depthFirst(E)
depthFirst(I)
depthFirst(J)
depthFirst(G)
depthFirst(K)

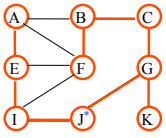


Markera noden som besökt.
Grannar = {G}
G redan besökt.

depthFirst(A)
depthFirst(E)
depthFirst(I)
depthFirst(J)
depthFirst(G)



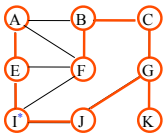
depthFirst(A)
depthFirst(E)
depthFirst(I)
depthFirst(J)



depthFirst(A)

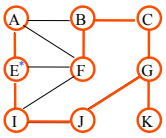
depthFirst(E)

depthFirst(I)

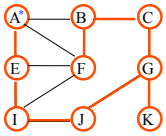


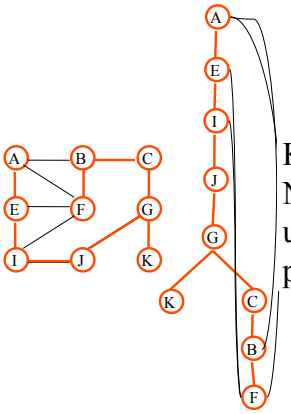
depthFirst(A)

depthFirst(E)

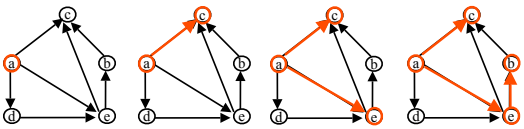


depthFirst(A)

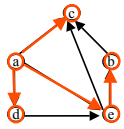




Klart!
Notera att vi fick ett uppspännande träd på samma gång.



1. Markera noden som besökt. Grannarna = {c, e, d} Rekursivt anrop c.
2. Markera noden som besökt. Inga grannar. Återgå till 1, nytt anrop e.
3. Markera noden som besökt. Grannarna = {b, c} Rekursivt anrop b.
4. Markera noden som besökt. Grannarna = {c} c redan besökt, åter till 3 c redan besökt. Åter till 1 nytt anrop d
5. Markera noden som besökt. Grannarna ={e}. Redan besökt. Åter 1.

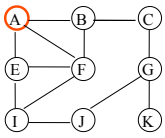


Bredden-först-algoritm

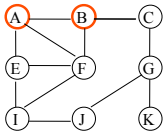
- Man undersöker först noden, sedan dess grannar, grannarnas grannar osv.
- Finns risk för oändlig körning här med om man inte använder en markör för att noden besökts.
- Endast noder till vilka det finns en väg från utgångsnoden kommer att besökas.
- En kö hjälper oss hålla reda på grannarna.

Bredden-först-algoritm:

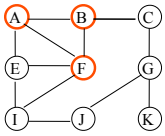
```
Algoritm breadthFirst(Node n, Graph g)
   input: A node n in a graph g to be traversed
Queue q ← empty();
visited(n, g) // Marks the node as visited
q ← enqueue(n, q);
while not isempty(q) do
   newNode ← front(q)
   q ← dequeue(q);
   neighbourSet ← neighbours(newNode, g);
   for each neighbour in neighbourSet do
      if not isVisited(neighbour)
        visited(neighbour, g);
        q ← enqueue(neighbour, q);
```

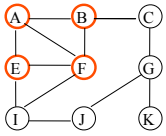
Markera noden som besökt och lägg in den i kön.
q = (A)
Ta fram första elementet (A),
q = ( )
Ta sedan fram grannmängden till A
S = {B, F, E}



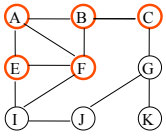
För var och en av grannarna:
B är inte besökt, besök B och lägg in B i kön
q = (B)



F är inte besökt, besök F och lägg in F i kön
q = (B, F)



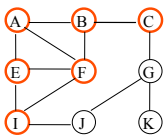
E är inte besökt, besök E och lägg in E i kön
q = (B, F, E)



q = (B, F, E), ta fram första elementet (B)
q = (F, E)
Ta sedan fram grannmängden till B
S = {A, F, C}

För var och en av grannarna:
A och F är besökta

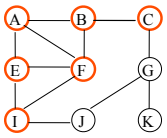
C är inte besökt, besök C och lägg in C i kön
q = (F, E, C)



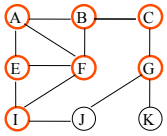
q = (F, E, C), ta fram första elementet (F)
q = (E, C)
Ta sedan fram grannmängden till F
S = {B, A, E, I}

B, A, och E är besökta

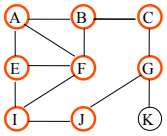
I är inte besökt, besök I och lägg in I i kön
q = (E, C, I)



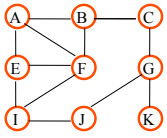
q = (E, C, I), ta fram första elementet (E)
q = (C, I)
Ta sedan fram grannmängden till E
S = {A, F, I}
För var och en av grannarna:
Alla är besökta



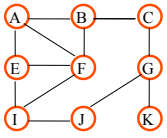
q = (C, I), ta fram första elementet (C)
q = (I)
Ta sedan fram grannmängden till C
S = {B, G}
För var och en av grannarna:
B är besökt
G är är inte besökt, besök G och lägg in G i kön
q = (I, G)



q = (I, G), ta fram första elementet (I)
q = (G)
Ta sedan fram grannmängden till I
S = {E, F, J}
E och F är besökta
J är inte besökt, besök J och lägg in J i kön
q = (G, J)

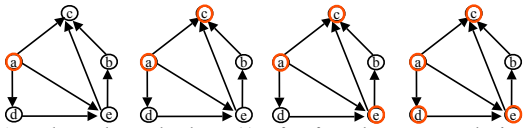


q = (G, J), ta fram första elementet (G)
q = (J)
Ta sedan fram grannmängden till G
S = {C, J, K}
C och J är besökta
K är inte besökt, besök K och lägg in K i kön
q = (J, K)

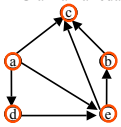


q =(J, K), ta fram första elementet (J)
q = (K)
Ta sedan fram grannmängden till J
S = {I, G}
Båda är besökta

q = (K), ta fram första elementet (K)
q = ()
Ta sedan fram grannmängden till K
S = {G}
Den är besökt.
Nu är kön tom och algoritmen klar.



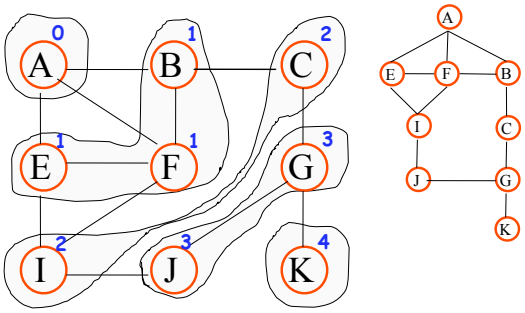
1. Markera noden som besökt. q = (a), ta fram första elem ur q. Leta reda på grannarna = {c, e, d}
2. Markera c som besökt. Stoppa in i kön q = (c)
3. Markera e som besökt. Stoppa in i kön q = (c, e)
4. Markera d som besökt. Stoppa in i kön q = (c, e, d)
5. Ta första ur kön (= c) q = (e, d), c har inga grannar. Ta första ur kön (=e) q = (d). e har grannarna = {b} Markera b som besökt. Stoppa in i kön q = (d, b)
6. Ta första ur kön (=d) q = (b). Grannarna redan besökta. Ta första ur kön (=b) q = (). Grannarna redan besökta. Kön tom. Klart!



Kortaste-vägen-algoritm vid lika vikt

- Om vi har en graf med lika vikter på alla bågar kan man använda en variant av bredden-först traversering för att beräkna kortaste vägen från en nod till de andra.

```
Algoritm breadthFirst(Node n, Graph g)
    input: A node n in a graph g to be
traversed
Queue q ← empty();
visited(n, g) // Marks the node as visited
setDist(n, 0)
q ← enqueue(n, q);
while not isempty(q) do
    newNode ← front(q)
    q ← dequeue(q);
    neighbourSet ← neighbours(newNode, g);
    for each neighbour in neighbourSet do
        if not isVisited(neighbour)
            visited(neighbour, g);
            setDist(neighbour, getDist(newNode)+1)
            q ← enqueue(neighbour, q);
```

Om alla bågar har vikt 1

Tidskomplexitet

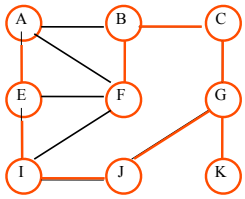
- För både bredden-först och djupet-först gäller:
  - Varje nod besöks exakt en gång O(n)
  - För varje nod följer man bågarna ut från noden för att hitta grannarna. Detta bör vara effektivt O(grad(v)), värsta fallet är O(n).
    - I bokens navigeringsorienterade spec. har vi O(grad(v))
    - Mängdorienterad spec. ger O(m) där m=antalet bågar i grafen
    - Eftersom grannmängden behöver evalueras en gång för varje nod blir komplexiteten O(Σ grad(v)) =O(m)
  - Totalt O(n) + O(m)

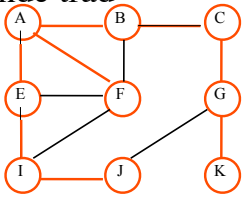
Uppspännande träd

- Både bredden-först och djupet-först-traverseringarna gav oss uppspännande träd.
  - Om vi sparat undan informationen vill säga…
  - Måste utöka grafspecifikationen med operationer som stöder detta.
- Är det minimalt?
  - Den totala längden i trädet ska vara minimalt.
  - Om varje kant har samma vikt är trädet minimalt uppspännande för bredden-först traversering.

Uppspännande träd

Skapat med djupet-först
Ej minimalt

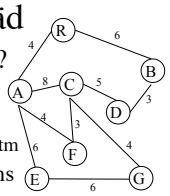




Skapat med bredden-först
Minimalt

Uppspännande träd

- Hur hanterar man grafer med vikter?
  - 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

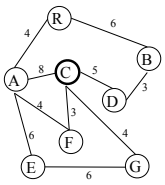


Prims algoritm

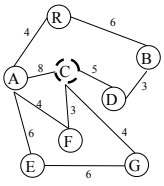
- Går ut på att bygga upp ett allt större träd som till slut spänner upp grafen eller en sammanhängande komponent av den.
- Man väljer i varje steg en båge med minimal vikt.
  - Lika vikter måste behandlas konsekvent. Regeln styr hur det färdiga trädet ser ut.

Prims algoritm:

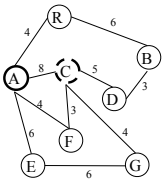
1. Välj en nod vilken som helst och markera den som öppen. Låt den bli rot.
2. Markera den som stängd.
3. För var och en av (de icke-stängda) grannarna:
   1. Markera den som öppen (om den inte är det).
   2. Stoppa in den aktuella noden, grannen och vikten i en prioritetskö. Är vikterna lika ska det nya elementet läggas in först i kön. (Dvs relationen är $\leq$)
3. Ta fram ett element ur prioritetskön och bilda ett nytt delträd genom att lägga in den båge som finns i elementet i trädet. OBS! Lägg endast in bågen om slutnoden inte är stängd! Låt ändnoden bli den nya aktuella noden, stäng den och gå till 3.



1. Välj en nod vilken som helst och markera den som öppen. Låt den bli rot.

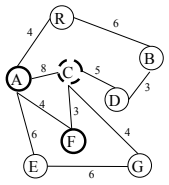


2. Markera den som stängd.



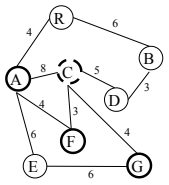
P = (**(C, A, 8)**)

3. För var och en av (de icke-stängda) grannarna:
   1. Markera den som öppen (om den inte är det).
   2. Stoppa in den aktuella noden, grannen och vikten i en prioritetskö. Är vikterna lika ska det nya elementet läggas in först i kön. (Dvs relationen är $\leq$)



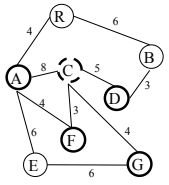
P = (**(C, F, 3)**, (C, A, 8))

3. För var och en av (de icke-stängda) grannarna:
   1. Markera den som öppen (om den inte är det).
   2. Stoppa in den aktuella noden, grannen och vikten i en prioritetskö. Är vikterna lika ska det nya elementet läggas in först i kön. (Dvs relationen är ≤)



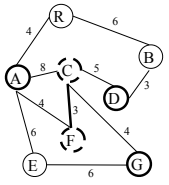
P = ((C, F, 3), **(C, G, 4)**, (C, A, 8))

3. För var och en av (de icke-stängda) grannarna:
   1. Markera den som öppen (om den inte är det).
   2. Stoppa in den aktuella noden, grannen och vikten i en prioritetskö. Är vikterna lika ska det nya elementet läggas in först i kön. (Dvs relationen är ≤)



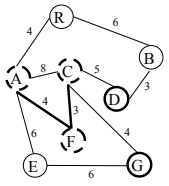
P = ((C, F, 3), (C, G, 4), **(C, D, 5)**, (C, A, 8))

3. För var och en av (de icke-stängda) grannarna:
   1. Markera den som öppen (om den inte är det).
   2. Stoppa in den aktuella noden, grannen och vikten i en prioritetskö. Är vikterna lika ska det nya elementet läggas in först i kön. (Dvs relationen är ≤)



P = ((C, G, 4), (C, D, 5), (C, A, 8))

4. Ta fram ett element ur prioritetskön och bilda ett nytt delträd genom att lägga in den båge som finns i elementet i trädet. OBS! Lägg endast in bågen om slutnoden inte är stängd! Låt ändnoden bli den nya aktuella noden, stäng den och gå till 3.

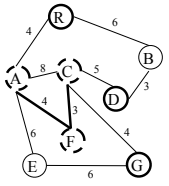


P = (**(F, A, 4)**, (C, G, 4), (C, D, 5), (C, A, 8))

P = ((C, G, 4), (C, D, 5), (C, A, 8))

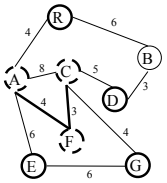
3. För var och en av (de icke-stängda) grannarna:
   1. Markera den som öppen (om den inte är det).
   2. Stoppa in den aktuella noden, grannen och vikten i en prioritetskö. Är vikterna lika ska det nya elementet läggas in först i kön. (Dvs relationen är ≤)

4. Ta fram ett element ur prioritetskön och bilda ett nytt delträd genom att lägga in den båge som finns i elementet i trädet. OBS! Lägg endast in bågen om slutnoden inte är stängd! Låt ändnoden bli den nya aktuella noden, stäng den och gå till 3.



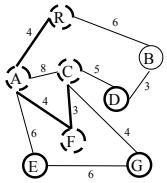
P = (**(A, R, 4)**, (C, G, 4), (C, D, 5), (C, A, 8))

3. För var och en av (de icke-stängda) grannarna:
   1. Markera den som öppen (om den inte är det).
   2. Stoppa in den aktuella noden, grannen och vikten i en prioritetskö. Är vikterna lika ska det nya elementet läggas in först i kön. (Dvs relationen är ≤)



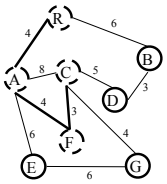
P = ((A, R, 4), (C, G, 4), (C, D, 5), **(A, E, 6)**, (C, A, 8))

3. För var och en av (de icke-stängda) grannarna:
   1. Markera den som öppen (om den inte är det).
   2. Stoppa in den aktuella noden, grannen och vikten i en prioritetskö. Är vikterna lika ska det nya elementet läggas in först i kön. (Dvs relationen är ≤)



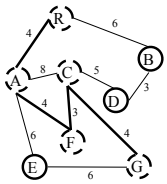
P = ((C, G, 4), (C, D, 5),(A, E, 6), (C, A, 8))

4. Ta fram ett element ur prioritetskön och bilda ett nytt delträd genom att lägga in den båge som finns i elementet i trädet. OBS! Lägg endast in bågen om slutnoden inte är stängd! Låt ändnoden bli den nya aktuella noden, stäng den och gå till 3.



P = ((C, G, 4), (C, D, 5), **(R, B, 6)**, (A, E, 6), (C, A, 8))

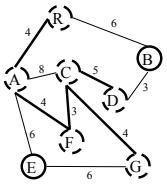
3. För var och en av (de icke-stängda) grannarna:
   1. Markera den som öppen (om den inte är det).
   2. Stoppa in den aktuella noden, grannen och vikten i en prioritetskö. Är vikterna lika ska det nya elementet läggas in först i kön. (Dvs relationen är ≤)



P = ((C, D, 5), (R, B, 6), (A, E, 6), (C, A, 8))

P = ((C, D, 5), **(G, E, 6)**, (R, B, 6), (A, E, 6), (C, A, 8))

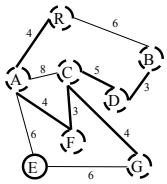
4. Ta fram ett element ur prioritetskön och bilda ett nytt delträd genom att lägga in den båge som finns i elementet i trädet. OBS! Lägg endast in bågen om slutnoden inte är stängd! Låt ändnoden bli den nya aktuella noden, stäng den och gå till 3.
3. För var och en av (de icke-stängda) grannarna:
   1. Markera den som öppen (om den inte är det).
   2. Stoppa in den aktuella noden, grannen och vikten i en prioritetskö. Är vikterna lika ska det nya elementet läggas in först i kön. (Dvs relationen är ≤)



P = ((G, E, 6), (R, B, 6), (A, E, 6), (C, A, 8))

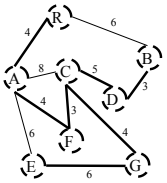
P = (**(D, B, 3)**, (G, E, 6), (R, B, 6), (A, E, 6), (C, A, 8))

4. Ta fram ett element ur prioritetskön och bilda ett nytt delträd genom att lägga in den båge som finns i elementet i trädet. OBS! Lägg endast in bågen om slutnoden inte är stängd! Låt ändnoden bli den nya aktuella noden, stäng den och gå till 3.
3. För var och en av (de icke-stängda) grannarna:
   1. Markera den som öppen (om den inte är det).
   2. Stoppa in den aktuella noden, grannen och vikten i en prioritetskö. Är vikterna lika ska det nya elementet läggas in först i kön. (Dvs relationen är ≤)



P = ((G, E, 6), (R, B, 6), (A, E, 6), (C, A, 8))

4. Ta fram ett element ur prioritetskön och bilda ett nytt delträd genom att lägga in den båge som finns i elementet i trädet. OBS! Lägg endast in bågen om slutnoden inte är stängd! Låt ändnoden bli den nya aktuella noden, stäng den och gå till 3.



P = ((R, B, 6), (A, E, 6), (C, A, 8))

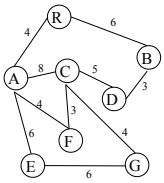
P = ((A, E, 6), (C, A, 8))

P = ((C, A, 8))

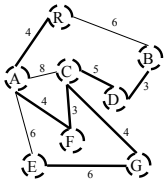
P = () Klart!

4. Ta fram ett element ur prioritetskön och bilda ett nytt delträd genom att lägga in den båge som finns i elementet i trädet. OBS! Lägg endast in bågen om slutnoden inte är stängd! Låt ändnoden bli den nya aktuella noden, stäng den och gå till 3.

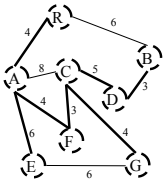
Resultat av Prims algoritm:



Start



Slut med ≤



Slut med <

Prims algoritm - komplexitet

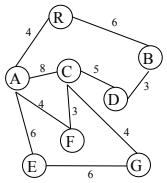
- Man gör en traversering av grafen, dvs O(m) + O(n).
- Sen tillkommer köoperationer
  - För varje båge sätter man in ett element i kön, inspekterar det och tar ut det. Detta blir O(m)
  - Om man använder Heap (partiellt sorterat binärt träd) får vi O(log m).
- Totalt: $O(n) + O(m^2)$ eller O(n) + O(m log m) beroende på implementation av prioritetskön.

Kruskals algoritm

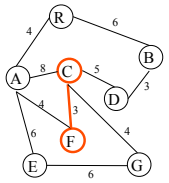
- Här väljer man också bågar allt eftersom men man bryr sig inte om att forma delträd under konstruktionen.
- Man gör ingen traversering utan arbetar på ett annat sätt med bågarna.
- Man färglägger bågarna för att hålla reda på vilken delgraf de tillhör.

Kruskals algoritm

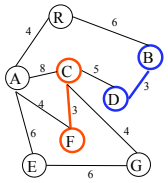
1. Skapa en prioritetskö av alla bågarna utifrån vikterna på dessa.
2. Den första bågen plockas fram och bildar den första delgrafen. Noderna färgläggs.
3. Upprepa tills kön är tom:
   1. Ta fram en ny båge.
   2. Om ingen av noderna är färgade
      1. Färglägg med ny färg och bilda ny delgraf.
   2. Om endast en nod är färgad
      1. Ingen risk för cykel utöka grafen och färglägg.
   2. Om båda noderna är färgade med *olika* färg
      1. Välj en av färgerna och färga om den nya gemensamma grafen.
   2. Om båda noderna har *samma* färg
      1. Ignorera bågen, den skapar en cykel



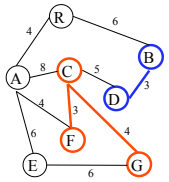
P = ((C,F,3), (B,D,3), (C,G,4),(A,F,4), (A,R,4), (C,D,5), (E,G,6), (B,R,6), (A,E,6), (A,C,8)



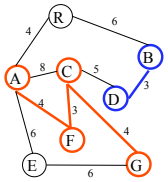
P = ((B,D,3), (C,G,4),(A,F,4), (A,R,4),
(C,D,5), (E,G,6), (B,R,6), (A,E,6), (A,C,8)



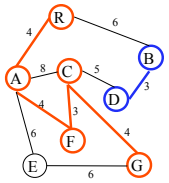
P = ((C,G,4),(A,F,4), (A,R,4), (C,D,5),
(E,G,6), (B,R,6), (A,E,6), (A,C,8)



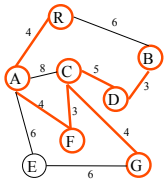
P = ((A,F,4), (A,R,4), (C,D,5),
(E,G,6), (B,R,6), (A,E,6), (A,C,8)



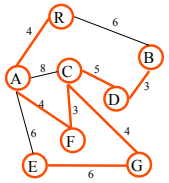
P = ((A,R,4), (C,D,5),(E,G,6),
(B,R,6), (A,E,6), (A,C,8)



P = ((C,D,5),(E,G,6), (B,R,6),(A,E,6), (A,C,8)



P = ((E,G,6),(B,R,6),(A,E,6), (A,C,8))



P = ((B,R,6),(A,E,6), (A,C,8))

P = ((A,E,6), (A,C, 8)) P = ((A,C,8))

P = ()

Kruskals algoritm - komplexitet

- Första steget i algoritmen bygger en prioritetskö utifrån en bågmängd.
  - Komplexitet beror på implementationen av bågmängden och prioritetskön…
- Varje båge traverseras en gång.
- Resten kan delas in i fyra fall:
  - Tre fall med komplexitet O(1) där bågen kan läggas till utan problem.
  - Ett fall där en delgraf måste färgas om. Komplexitet O(n).

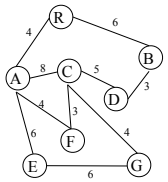
Finna vägen till en nod

- Bredden-först traversering ger oss vägarna från en nod till alla andra.
  - Om vi sparar undan vägen…
- Är det den kortaste?
  - Ja, om alla vikter lika!
  - Annars då? Vi kommer titta på två algoritmer:
    - Floyds shortest path $O(N^3)$
    - Dijkstras shortest path

Floyds shortest path

- Bygger på att man representerar grafen med hjälp av en matris. (Eller skapar en matrisrepresentation)

	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 floyd(Graph g)
   input: A graph g to find shortest path in
// Get matrix representation
A ← getMatrix(g)
N ← getNoOfNodes(g)
for k=0 to N-1
   for i=0 to N-1
      for j=0 to N-1
        A(i,j) = min(A(i,j), A(i,k)+ A(k,j))
```

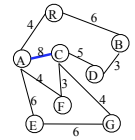
A innehåller kortaste avstånden men hur få tag på vägen?
Spara på samma gång en föregångar-matris. Det kommer också kosta $O(N^3)$ så den ökar inte komplexiteten.

Uppdaterad Floyd

```
Algoritm floyd(Graph g)
   input: A graph g to find shortest path in
// Get matrix representation
A ← getMatrix(g)
N ← getNoOfNodes(g)
for i = 0 to N-1
   for j = 0 to N-1
     if (i==j or A(i,j)==inf) Path(i,j) = -1
     else Path(i,j) = i
for k=0 to N-1
   for i=0 to N-1
     for j=0 to N-1
       if (A(i,j)> A(i,k)+A(k,j))
            Path(i,j) = Path(k,j)
       A(i,j) = min(A(i,j), A(i,k)+A(k,j))
```

k0

	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



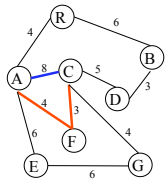
k8

	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

Vi har hittat en kortare väg mellan A och C. Vilken är den?
Vilken är vägen mellan R och G?

	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	-

Låt oss leta i vår föregångarmatris. (För enkelhetens skull har jag kodat om siffrorna till motsvarande noder på OH-bilden.)



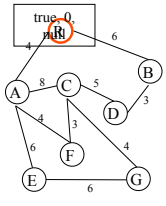
Om vi vill hitta vägen mellan A och C gör man så här: Titta på kolumnen för A. Leta reda på raden för C. Där ser vi F. Sedan tittar vi i kolumnen för F och raden C där ser vi C. Vägen är alltså A-F-C. På samma sätt ser vi att kortaste vägen mellan R och G är R-A-F-C-G (med kostnad 15).

Dijkstras algoritm

- Söker kortaste vägen från en nod n till alla andra noder.
- Fungerar enbart på grafer med positiva vikter.
- Låt varje nod ha följande attribut
  - Visited – som blir sann när vi hittat en väg till den
  - Distance – värdet på den kortaste vägen fram till noden
  - Parent – Referens till föregångaren på vägen

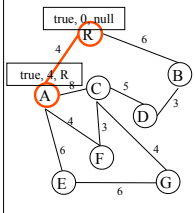

```
Algoritm dijkstra(Node n, Graph g)
  input: A graph g to find shortest path
         starting from node n
n.visited ← true; n.distance ← 0; n.parent ← null;
Pqueue q ← empty();
q ← insert(n,q);
while not isempty(q)
  v ← inspect-first(q); q ← delete-first(q);
  d ← v.distance;
  neighbourSet ← neighbours(v, g);
  for each w in neighbourSet do
    newDist ← d + getWeight(v,w);
    if not isVisited(w)
      w.visited←true;
      w.distance←newDist;
      w.parent←v;
      q ← insert(w,q);
    else if newDist < w.distance
      w.distance←newDist;
      w.parent←v; q←update(w,q)
```

Dijkstras algoritm

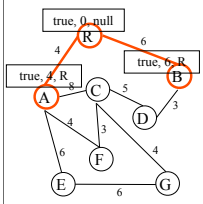


Vi startar i R. Sätter värden i noden. Skapar Kö och stoppar in R q = (R(true,0,Null)).

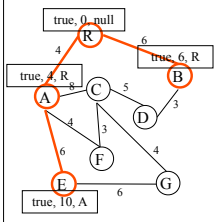
Tar fram v ur kön v = R(true,0,Null) och q = (). d = 0
Leta sedan fram grannarna = {A, B}
För granne A: newDist = 0+4 = 4. Ej besökt.
q = (A(true,4,R))



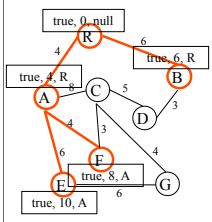
För granne B: newDist = 0+6 = 6. Ej besökt.
q = (A(true,4,R), B(true,6,R))



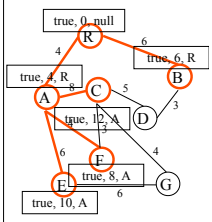
Tar fram v ur kön v = A(true,4,R) och
q = (B(true,6,R)). d = 4
Leta sedan fram grannarna = {E,F,C,R}
För granne E: newDist = 4+6 = 10. Ej besökt.
q = (B(true,6,R), E(true,10,A))



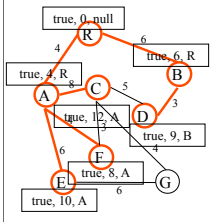
För granne F: newDist = 4+4 = 8. Ej besökt.
q = (B(true,6,R), F(true,8,A), E(true,10,A))



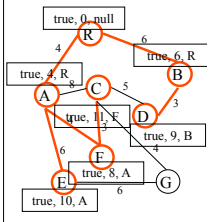
För granne C: newDist = 4+8 = 12. Ej besökt.
q = (B(true,6,R), F(true,8,A), E(true,10,A)
C(true,12,A))



För granne R: newDist = 4+4 = 8. Besökt.
8>0 gör inget.
Tar fram v ur kön v = B(true,6,R) och
q = (F(true,8,A), E(true,10,A), C(true,12,A))
d = 6
Leta sedan fram grannarna = {R, D}
För granne R: newDist = 6+6 = 12. Besökt.
12>0 gör inget.
För granne D: newDist = 6+3 = 9. Ej besökt.
q = (F(true,8,A), D(true,9,B), E(true,10,A),
C(true,12,A))

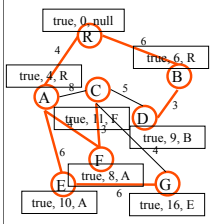


Tar fram v ur kön v = F(true,8,A) och
q = (D(true,9,B), E(true,10,A),C(true,12,A))
d = 8
Leta sedan fram grannarna = {A, C}
För granne A: newDist = 8+4 = 12. Besökt.
12>4 gör inget.
För granne C: newDist = 8+3 = 11. Besökt.
11<12 !!
C.distance=11 C.parent = F
q = (D(true,9,B), E(true,10,A),C(true,11,F))

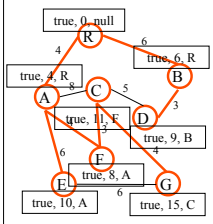


Tar fram v ur kön v = D(true,9,B) och
q = (E(true,10,A),C(true,11,F)) d =
9
Leta sedan fram grannarna = {B, C}
För granne B: newDist = 9+3 = 12. Besökt.
12>6 gör inget.
För granne C: newDist = 9+5 = 14. Besökt.
14>11 gör inget

Tar fram v ur kön v = E(true,10,A) och
q = (C(true,11,F)) d = 10
Leta sedan fram grannarna = {A, G}
För granne A: newDist = 10+6 = 16. Besökt.
16>4 gör inget.
För granne G: newDist = 10+6 = 16. Ej besökt.
q = (C(true,11,F), G(true, 16,E))



Tar fram v ur kön v = C(true,11,F) och
q = (G(true, 16,E)) d = 11
Leta sedan fram grannarna = {A, F, G, D}
För granne A: newDist = 11+8 = 19. Besökt.
19>4 gör inget.
För granne F: newDist = 11+3 = 14. Besökt.
14>8 gör inget
För granne G: newDist = 11+4 = 15. Besökt.
15<16!!
G.distance=15 G.parent = C
q = (G(true, 15,C))



För granne D: newDist = 11+5 = 16. Besökt.
16>9 gör inget.
Tar fram v ur kön v = G(true, 15,C) och
q = () d = 15
Leta sedan fram grannarna = {E, C}
För granne E: newDist = 15+6 = 21. Besökt.
21>10 gör inget.
För granne C: newDist = 15+4 = 19. Besökt.
19>11 gör inget

Stanna algoritmen klar.

Dijkstras algoritm - komplexitet

- Vi sätter in varje nod i kön en gång.
  - Totalt n*O(insert)
- Vi tar ut varje nod ur kön en gång.
  - Totalt n*O(delete-first)
- Vi kan behöva uppdatera element i kön.
  - Maximalt m gånger, m*O(update)
- Osorterad lista
  - n*O(1)+n*O(n) + m*O(1) = $O(n^2)$ +O(m)
- Heap
  - n*O(log n)+n*O(log n) + m*O(log n)
    = O((n+m)log n)

Om smart implementation…

Floyd vs. Dijkstra

- Floyd $O(n^3)$ hittar den kortaste vägen mellan alla noder.
- Dijkstra O((n+m) log n) med heap, hittar kortaste vägen mellan en nod och alla andra.
  - Måste köras N gånger för att få samma resultat som Floyd. Dvs O(n(n+m) log n).
  - Är bättre på stora glesa grafer.

Flödet i en graf

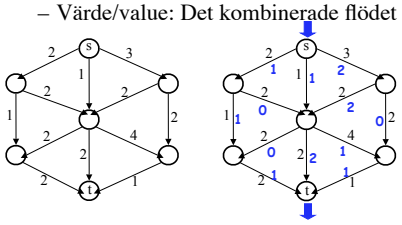
- Riktad graf med vikter $c_{v,w}$, som anger flödeskapacitet över bågen *(v,w)*.
- Kapaciteten kan t.ex. vara mängden vätska som kan flöda genom ett rör, maximala mängden trafik på en väg eller kommunikationskapaciteten i ett datornät.
- Grafen har två noder *s* (source) och *t* (sink) och uppgiften är att beräkna det maximala flödet mellan *s* och *t*.
- Genom varje båge *(u,v)* kan vi maximalt ha ett flöde på *c(u,v)* enheter.
- För varje node *v* gäller att det totala inkommande flödet måste vara lika med det utgående flödet.

Flödet i en graf

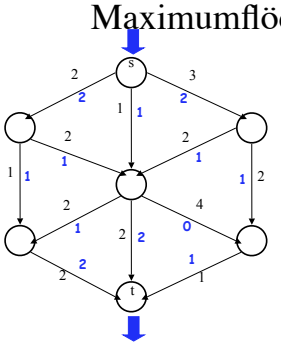
- Ett flödesnätverk består av
  - En riktad graf
  - Vikter c(u,v) som kallas kapaciteter på bågarna
    - Ickenegativa vikter
  - Två speciella noder
    - Källan (source), betecknas ”s”, en nod utan ingående bågar
    - Avloppet (sink) betecknas ”t”, en nod utan utgående bågar

Kapacitet och flöde

- Flödet är en funktion på kanterna:
  - $0 \leq$ flöde $\leq c(u, v)$
  - Flödet in till noden = flödet ut ur noden
  - Värde/value: Det kombinerade flödet in till avloppet.

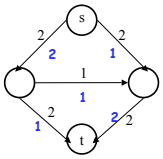


Maximumflödes problem

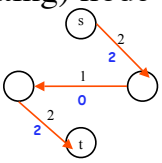


- Givet ett nätverk N, hitta ett flöde med maximalt värde
- Ett exempel på maximalt flöde
  Värde = 5

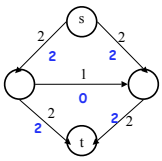
Förbättrande (augmenting) flöde



Nätverk med flödesvärde 3



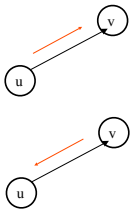
Förbättrande väg
(Augmenting path)



Nu har flödesvärdet ökat till 4!!

Ökande väg

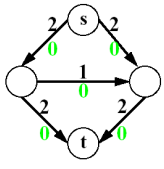
- Framåtriktade bågar
  - flödet(u, v) < c(u, v)
  - Flödet kan ökas!
- Bakåtriktade bågar
  - flödet(u,v) > 0
  - Flödet kan minskas!



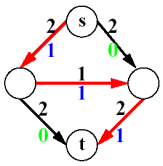
Maximala flödesteoremet + algoritm

- Ett flöde har maximum värde om och endast om nätverket inte har någon förbättrande väg.
- Ford & Fulkerson algoritmen:

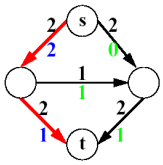
  Initialisera nätverket med noll flöde
  Anropa metoden findFlow() som definieras
    Om det finns förbättrande vägar:
      Hitta en av dem
      Öka flödet
      Anropa (rekursivt) findFlow()



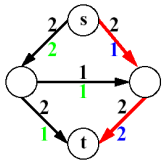
Initiera nätverket med nollflöde. Kapaciteterna i svart ovan bågarna och flödet i grönt nedan bågarna.



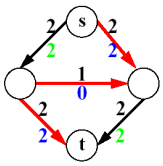
Skicka igenom ett enhetsflöde genom nätverket. Flödesvägen markerat med rött och de förbättrade flödesvärdena i blått.



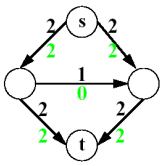
Skicka ytterligare ett enhetsflöde genom nätverket.



Skicka igenom ytterligare ett enhets-flöde genom nätverket. Notera att det finns ytterligare en förbättrande väg som går genom kanten i mitten.



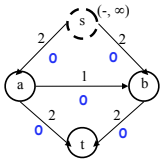
Skicka ett enhetsflöde genom den förbättrande vägen. Nu finns det inga fler förbättrande vägar. Alltså…



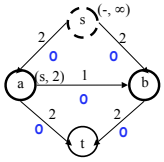
Vi har hittat detta nätverks maximala flöde!

Hur gör man mer specifikt?

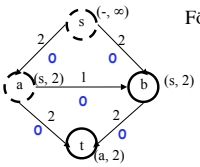
- Hur vet man att det finns en förbättrande väg?
- Hur vet man vilken av de förbättrade vägarna man ska ta först?
- Läs mer i boken.



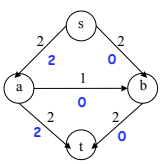
Första vägen från s till t:
s märks som stängd och (-, ∞) (dvs inte öppnats från någon nod flödet kan förändras oändligt)



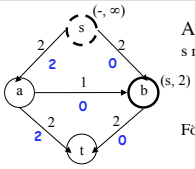
Bågarna från s traverseras, noderna i andra änden markeras öppna och märks med deras maximala kapacitet och sätts in i prio-kön.
q = (a, b)



Första noden tas från kön (a) och markeras stängd. Dess bågar undersöks i tur och ordning.
Bågen (a,s) leder till stängd nod och (a, b) leder till öppen nod - inget händer. Bågen (a,t) leder till en ny nod t som markeras (a, 2)

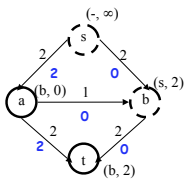


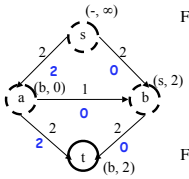
Nu har vi nått fram till t, traverseringen avbryts. Man följer stegen bakåt till s och markerar bågarna samtidigt som noderna avmarkeras.



Andra vägen från s till t:
s märks som stängd och (-, ∞)
Bågarna från s traverseras. Eftersom vägen till a inte kan ökas mer struntar vi i den. q=(b)

Första noden tas från kön (b) och markeras stängd. Dess bågar undersöks i tur och ordning. Bågen (b,s) leder till stängd nod - inget händer.
Bågen (b,a) leder till en ny nod a (baklänges) som kan markeras med maximalt det nuvarande flödet, dvs (b, 0).
Bågen (b,t) leder till en ny nod t som markeras (b, 2).
q=(a,t)





Första noden tas från kön (a) och markeras stängd. Dess bågar undersöks i tur och ordning. Bågen (a,s) och (a, b) leder till stängda noder och (a, t) till en öppen nod - inget händer.

Första noden tas från kön (t). Vi har nått t. Traversering avbryts och vi går bakåt och avmarkerar allt.

Om vi nu försöker börja om igen så hittar vi inget nytt eftersom både (s,a) och (s,b) utnyttjas maximalt.