Grafer

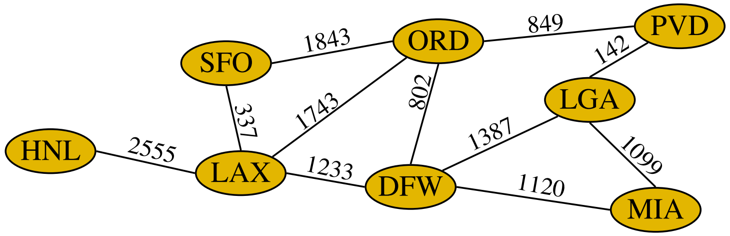
Graf

En **graf** er et par (***V*, *E***), hvor

- *V* er en mængde af **knuder** (vertices)
- ***E*** er en samling af par af knuder, kaldet **kanter** (edges)

Eksempel:

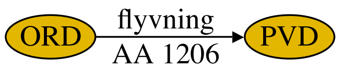
- En knude repræsenterer en lufthavn og har tilknyttet en lufthavnskode
- En kant repræsenterer en flyrute imellem to lufthavne og har tilknyttet en afstand



Kanttyper

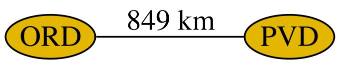
Orienteret kant

- ordnet par af knuder (***u***,***v***)
- første knude kaldes **startknuden**
- anden knude kaldes **slutknuden**
- eksempel: en flyvning



Ikke-orienteret kant

- ikke-ordnet par af knuder (***u***,***v***)
- eksempel: en flyrute



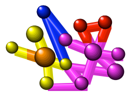
Orienteret graf

- alle kanter er orienteret
- eksempel: et netværk af flyvninger

Ikke-orienteret graf

- alle kanter er ikke-orienteret
- eksempel: et netværk af flyruter

Anvendelser



Alt hvad der involverer relationer imellem objekter kan modelleres ved hjælp af en graf

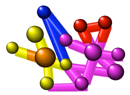
Trafiknetværk:
Knuder: byer, vejkryds
Kanter: veje

Organiske molekyler:
Knuder: atomer
Kanter: bindinger

Elektriske kredsløb:
Knuder: komponenter
Kanter: ledninger

Datanetværk:
Knuder: computere, routere
Kanter: forbindelser

Anvendelser (fortsat)



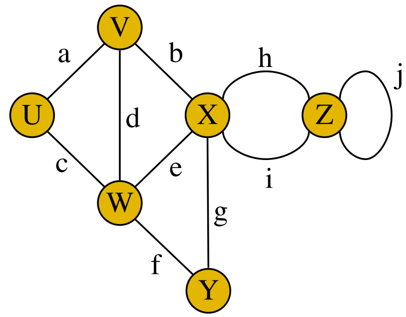
Programsystemer:
Knuder: metoder
Kanter: metode *A* kalder metode *B*

Objektorienteret design (UML-diagrammering):
Knuder: klasser/objekter
Kanter: nedarvning, aggregering eller associering

Projektplanlægning:
Knuder: delopgaver
Kanter: præcedenser (delopgave *A* skal udføres før delopgave *B*)

Terminologi

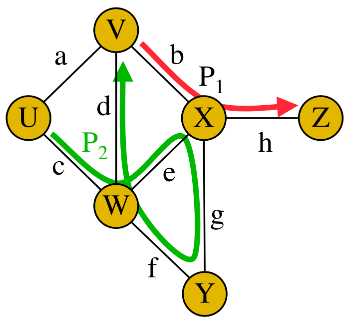
- **Endeknuder** for en kant
  U og V er endeknuder for a
- **Nabokanter** til en knude
  a, b og d er nabokanter til V
- **Graden** af en knude
  V har graden 3
- **Parallelle** (multiple) **kanter**
  h og i er parallelle kanter
- **Sløjfe**
  j er en sløjfe



Terminologi (fortsat)

- **Vej**
  følge af skiftevis knuder og kanter, som begynder og ender med en knude, og hvor hver kants endeknuder kommer umiddelbart før og efter knuden
- **Simpel vej**
  vej, hvor alle knuder og kanter er forskellige

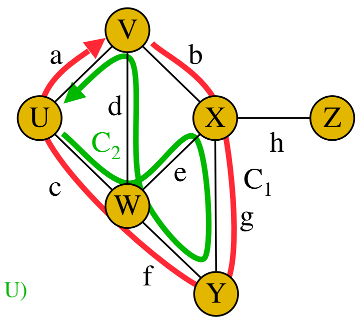
Eksempler:
P_1=(V, b, X, h, Z) er en simpel vej
P_2=(U, c, W, e, X, g, Y, f, W, d, V) er en vej, der ikke er simpel



Terminologi (fortsat)

- **Cykel** (eller **kreds**)
  vej, som begynder og ender i samme knude
- **Simpel cykel**
  cykel hvor alle kanter og alle knuder, med undtagelse af den første og sidste knude, er forskellige

Eksempler:
C_1=(V, b, X, g, Y, f, W, c, U, a, V) er en simpel cykel
C_2=(U, c, W, e, X, g, Y, f, W, d, V, a, U) er en cykel, der ikke er simpel



Egenskaber

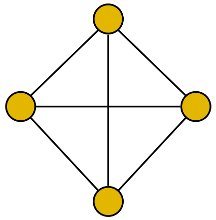
Notation

n	antal knuder
m	antal kanter
deg(v)	graden af knude v

Egenskab 1
$\Sigma_v \deg(v) = 2m$
Bevis: Hver kant tælles med to gange

Egenskab 2
I en ikke-orienteret graf uden sløjer og parallelle kanter
$m \leq n(n-1)/2$
Bevis: Hver knude har højst graden $n - 1$. Hver kant tælles med to gange

Hvad er den øvre grænse for en orienteret graf?



Eksempel:
$n = 4$
$m = 6$
$\forall v : \deg(v) = 3$

Basale grafproblemer

Veje:
Er der en vej fra knude *A* til knude *B*?

Cykler:
Indeholder grafen en cykel?

Sammenhæng (udspændende træ)**:**
Er der for enhver knude en vej til enhver anden knude?

2-sammenhæng:
Vil en sammenhængende graf altid forblive sammenhængende, hvis en knude og dennes nabokanter fjernes fra grafen?

Planaritet:
Kan grafen tegnes, uden at to kanter krydser hinanden?

Basale grafproblemer (fortsat)

Korteste vej:
Hvilken vej er den korteste fra knude *A* til knude *B*?

Længste vej:
Hvilken vej er den længste fra knude *A* til knude *B*?

Minimalt udspændende træ:
Hvad er den billigste måde at forbinde alle knuder?

Hamilton-cykel:
Er der en cykel, som indeholder samtlige knuder?

Den rejsende sælgers problem:
Hvilken Hamilton-cykel er den billigste?

Centrale metoder for en graf ADT

- **Knuder og kanter**
  er positioner
  lagrer elementer

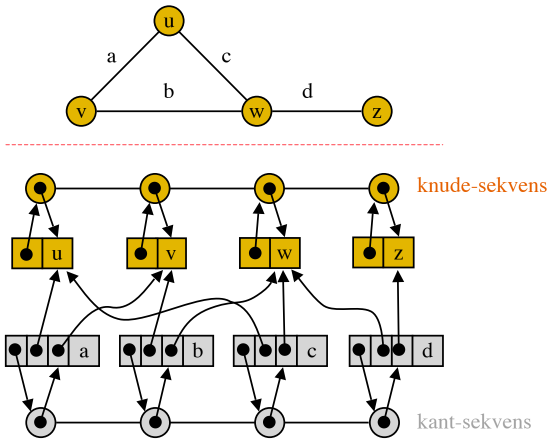
- **Tilgangsmetoder**
  incidentEdges(v)
  endVertices(e)
  isDirected(e)
  origin(e)
  destination(e)
  opposite(v, e)
  areAdjacent(v, w)

- **Opdateringsmetoder**
  insertVertex(o)
  insertEdge(v, w, o)
  insertDirectedEdge(v, w, o)
  removeVertex(v)
  removeEdge(e)

- **Generiske metoder**
  numVertices()
  numEdges()
  vertices()
  edges()
  (iteratorer)

Kantlistestruktur

- **Knude-objekt**
  - reference til position i knude-sekvensen
  - element
- **Kant-objekt**
  - reference til position i kant-sekvensen
  - startknude-objektet
  - slutknude-objektet
  - element

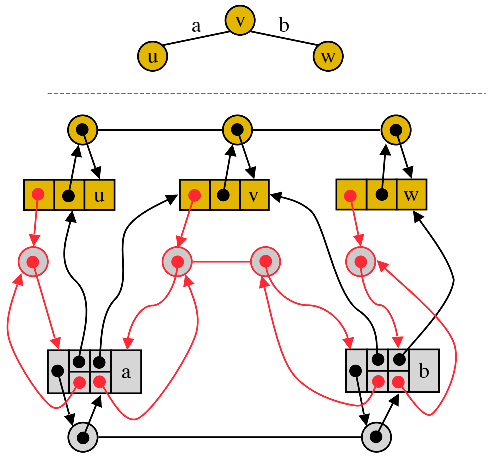


Asymptotisk effektivitet

• n knuder, m kanter • ingen parallelle kanter • ingen sløjfer • grænser udtrykkes ved O-notation	Kant-liste
Plads	$n + m$
incidentEdges(v)	m
areAdjacent(v, w)	m
insertVertex(o)	1
insertEdge(v, w, o)	1
removeVertex(v)	m
removeEdge(e)	1

Nabolistestruktur

- Udvidet kantlistestruktur
- Nabosekvens for hver knude: sekvens af referencer til nabokanter
- Udvidede kant-objekter: referencer til positioner i nabosekvensen

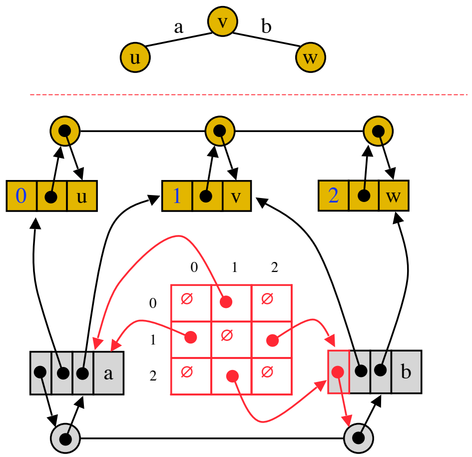


Asymptotisk effektivitet

• n knuder, m kanter • ingen parallelle kanter • ingen sløjfer • grænser udtrykkes ved O-notation	Nabo-liste	Kant-liste
Plads	$n + m$	$n + m$
incidentEdges(v)	deg(v)	m
areAdjacent(v, w)	min(deg(v), deg(w))	m
insertVertex(o)	1	1
insertEdge(v, w, o)	1	1
removeVertex(v)	deg(v)	m
removeEdge(e)	1	1

Nabomatrixstruktur

- Udvidet kantlistestruktur
- Udvidede knude-objekter
  - heltalsnøgle (index)
- Todimensionalt array
  - referencer til kanter, der forbinder knuder
- Simpel udgave har 0 for ingen kant, og 1 for en kant

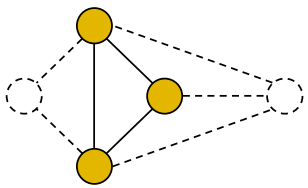


Asymptotisk effektivitet

• n knuder, m kanter • ingen parallelle kanter • ingen sløjfer • grænser udtrykkes ved O-notation	Nabo-matrix	Nabo-liste	Kant-liste
Plads	n^2	$n + m$	$n + m$
incidentEdges(v)	n	deg(v)	m
areAdjacent(v, w)	1	min(deg(v), deg(w))	m
insertVertex(o)	n^2	1	1
insertEdge(v, w, o)	1	1	1
removeVertex(v)	n^2	deg(v)	m
removeEdge(e)	1	1	1

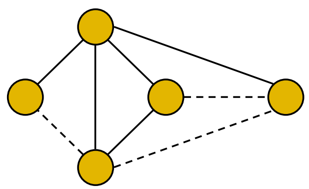
Delgrafer

En **delgraf** S af en graf G er en graf, hvor
- knuderne i S er delmængde af knuderne i G, og
- kanterne i S er en delmængde af kanterne i G



Delgraf

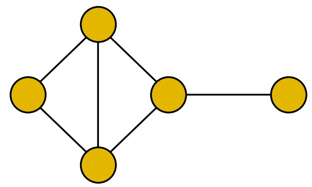
En **udspændende delgraf** af G er en delgraf, der indeholder alle G's knuder



Udspændende delgraf

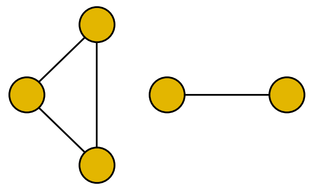
Sammenhæng

En graf er **sammenhængende**, hvis der findes en vej imellem ethvert par af knuder



Sammenhængende graf

En **sammenhængende komponent** i en graf er en maksimal sammenhængende delgraf af G



Ikke-sammenhængende graf med to sammenhængende komponenter

Træer og skove

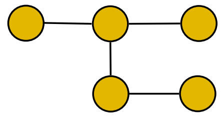
- Et (frit) **træ** er en ikke-orienteret graf ***T***, hvor

  T er sammenhængende, og
  T ikke har cykler

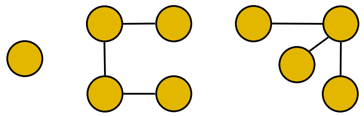
  Denne definition af et træ er forskellig fra definitionen af et træ med rod

- En **skov** er en ikke-orienteret graf uden cykler

  De sammenhængende komponenter af en skov er træer



Træ



Skov

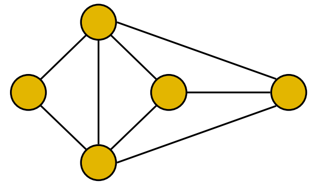
Udspændende træer og skove

- Et **udspændende træ** af en sammenhængende graf er en udspændende delgraf, som er et træ

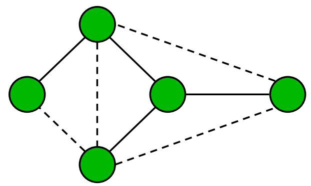
  Et udspændende træ er ikke unikt, medmindre grafen er et træ

  Udspændende træer finder anvendelse ved design af kommunikationsnetværk

- En **udspændende skov** af en graf er et udspændende delgraf, som er en skov



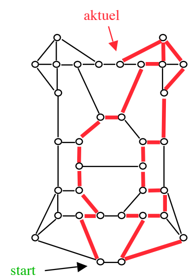
Graf



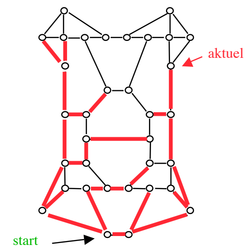
Udspændende træ

Dybde-først-søgning

Animering af DFS og BFS

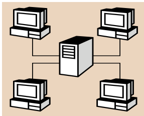


Dybde-først



Bredde-først

Dybde-først-søgning



Dybde-først-søgning (DFS) er en generel teknik til traversering af en graf

En dybde-først traversering af grafen G

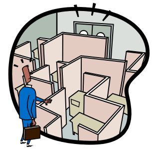
- besøger alle knuder og kanter i G
- afgør om G er sammenhængende
- bestemmer de sammenhængende komponenter i G
- bestemmer en udspændende skov i G

DFS i en graf med n knuder og m kanter tager $O(n+m)$ tid

DFS kan udvides til at løse andre grafproblemer:

- Find en vej imellem to givne knuder
- Find en cykel i grafen

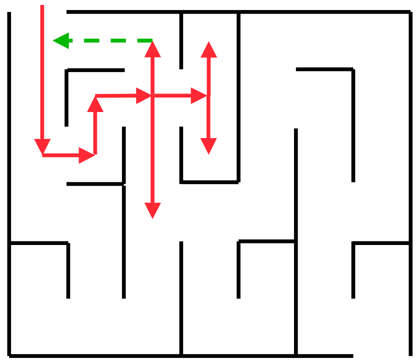
DFS til søgning i labyrint



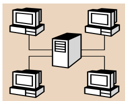
DFS-algoritmen ligner den klassiske strategi til søgning i en labyrint:

- Markér hvert kryds, hjørne og endevæg (knude), der besøges
- Markér hver korridor (kant), der besøges
- Hold rede på vejen tilbage til indgangen (startknuden) ved hjælp af et reb (rekursionsstak)

*** Backtracking ***



DFS-algoritme

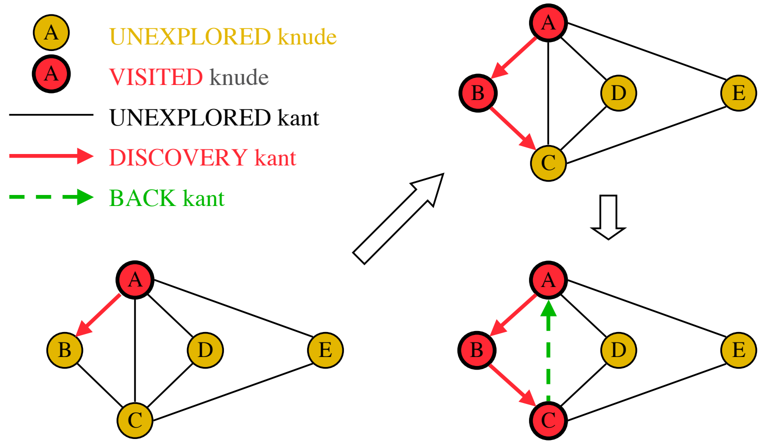


Algoritmen bruger en mekanisme til tildele og aflæse "etiketter" på knuder og kanter

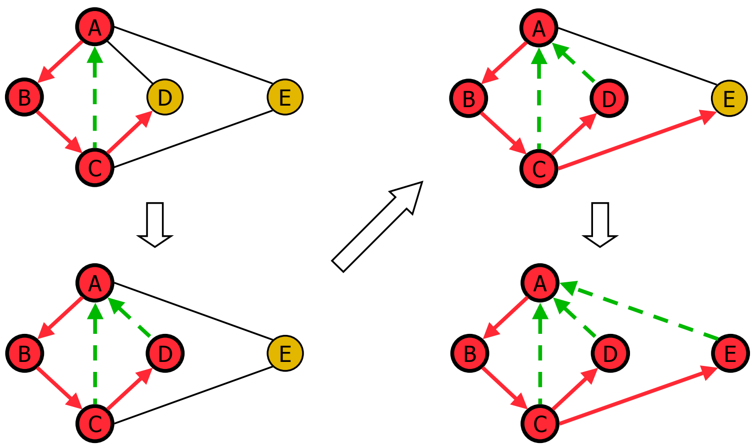
Algorithm *DFS*(G)
  Input graph G
  Output labeling of the edges of G as discovery edges and back edges
  for all $u \in G.vertices()$
    setLabel(u, *UNEXPLORED*)
  for all $e \in G.edges()$
    setLabel(e, *UNEXPLORED*)
  for all $v \in G.vertices()$
    if *getLabel*(v) = *UNEXPLORED*
      DFS(G, v)

Algorithm *DFS*(G, v)
  Input graph G and a start vertex v of G
  Output labeling of the edges of G in the connected component of v as discovery edges and back edges
  setLabel(v, *VISITED*)
  for all $e \in G.incidentEdges(v)$
    if *getLabel*(e) = *UNEXPLORED*
      $w \leftarrow opposite(v,e)$
      if *getLabel*(w) = *UNEXPLORED*
        setLabel(e, *DISCOVERY*)
        DFS(G, w)
      else
        setLabel(e, *BACK*)

Eksempel



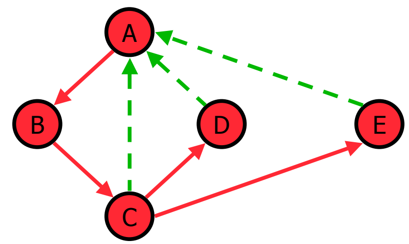
Eksempel (fortsat)



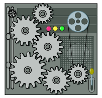
Egenskaber ved DFS

Egenskab 1
DFS*(*G, v*)** besøger alle knuder og kanter i den sammenhængende komponent, der indeholder ***v

Egenskab 2
De kanter, der besøges af ***DFS*(*G, v*)**, udgør et udspændende træ af den sammenhængende komponent, der indeholder ***v***



Analyse af DFS



- Tildeling og aflæsning af en knude- eller kant-etikette tager $O(1)$ tid
- Hvert knude får tildelt en etikette to gange
  første gang som UNEXPLORED
  anden gang som VISITED
- Hver kant får tildelt en etikette to gange
  første gang som UNEXPLORED
  anden gang som DISCOVERY eller BACK
- Metoden **incidentEdges** kaldes en gang for hver knude
- DFS kører i $O(n + m)$ tid, forudsat at grafen er repræsenteret med en nabolistestruktur $(\Sigma_v \deg(v) = 2m)$

Bestemmelse af vej

Vi kan specialisere DFS-algoritmen til at finde en vej imellem to givne knuder ***u*** and ***z***

Vi kalder ***DFS*(*G, u*)** med ***u*** som startknuden

Vi bruger en stak, ***S***, til at holde rede på vejen imellem startknuden og den aktuelle knude

Så snart målknuden ***z*** mødes, returneres vejen som indholdet af stakken

```
Algorithm pathDFS(G, v, z)
  setLabel(v, VISITED)
  S.push(v)
  if v = z
    return S.elements()
  for all e ∈ G.incidentEdges(v)
    if getLabel(e) = UNEXPLORED
      w ← opposite(v,e)
      if getLabel(w) = UNEXPLORED
        setLabel(e, DISCOVERY)
        S.push(e)
        pathDFS(G, w, z)
        S.pop(e)
      else
        setLabel(e, BACK)
  S.pop(v)
```

Bestemmelse af en cykel

Vi kan specialisere DFS-algoritmen til at finde en cykel

Vi bruger en stak, ***S***, til at holde rede på vejen imellem startknuden og den aktuelle knude

Så snart en **tilbage**-kant **(*v, w*)** mødes, returneres cyklen som den del af stakken, der findes fra toppen og ned til knuden ***w***

```
Algorithm cycleDFS(G, v)
  setLabel(v, VISITED)
  S.push(v)
  for all e ∈ G.incidentEdges(v)
    if getLabel(e) = UNEXPLORED
      w ← opposite(v,e)
      S.push(e)
      if getLabel(w) = UNEXPLORED
        setLabel(e, DISCOVERY)
        cycleDFS(G, w)
        S.pop(e)
      else
        T ← new empty stack
        repeat
          o ← S.pop()
          T.push(o)
        until o = w
        return T.elements()
  S.pop(v)
```


2-sammenhæng

Adskillelseskanter og -knuder

Definitioner

- Lad ***G*** være en sammenhængende graf
- En **adskillelseskant** i ***G*** er en kant, hvis fjernelse gør grafen usammenhængende
- En **adskillelsesknude** i ***G*** er en knude, hvis fjernelse gør grafen usammenhængende

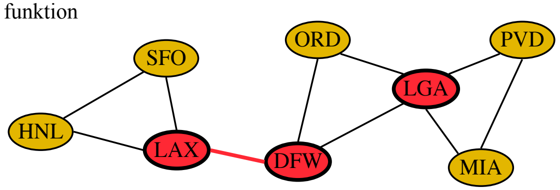
Eksempel

DFW, LGA og LAX er adskillelsesknuder

(LAX, DFW) er en adskillelseskant

Anvendelser:

Adskillelseskanter og -knuder repræsenterer enkeltpunkter for fejl i et netværk og er kritiske for netværkets funktion



2-sammenhængende graf

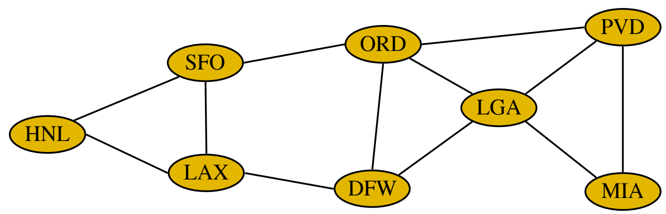
Ækvivalente definitioner af 2-sammenhængende graf ***G***:

(1) Grafen ***G*** har ingen adskillelseskanter og ingen adskillelsesknuder

(2) For ethvert par af knuder ***u*** og ***v*** i ***G*** er der to disjunkte simple veje imellem ***u*** og ***v*** (d.v.s. to simple veje, der ikke har nogen knuder og kanter til fælles)

(3) For ethvert par af knuder ***u*** og ***v*** i ***G*** er der en simpel cykel, der indeholder ***u*** og ***v***

Eksempel:

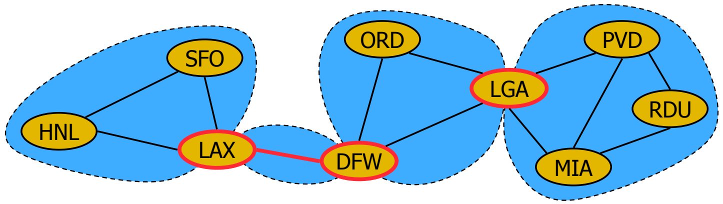


2-sammenhængende komponenter

2-sammenhængende komponent i en graf ***G*** er
- en maksimal 2-sammenhængende delgraf af ***G***, eller
- en delgraf bestående af en adskillelseskant i ***G*** og dennes endeknuder

Samspil imellem 2-sammenhængende komponenter:
- En kant tilhører præcis én 2-sammenhængende komponent
- En adskillelseskant tilhører en 2-sammenhængende komponent med præcis én kant
- En adskillelsesknude tilhører to eller flere 2-sammenhængende komponenter

Eksempel på graf med fire 2-sammenhængende komponenter:



Ækvivalensklasser

Lad der være givet en mængde ***S***

En **relation** ***R*** på ***S*** er en mængde af ordnede par af elementer fra ***S***, d.v.s. ***R*** er delmængde af ***S***×***S***

En **ækvivalensrelation** ***R*** på ***S*** tilfredsstiller følgende betingelser:

Refleksivitet: $(x,x) \in R$
Symmetri: $(x,y) \in R \Rightarrow (y,x) \in R$
Transitivitet: $(x,y) \in R \wedge (y,z) \in R \Rightarrow (x,z) \in R$

En ækvivalensrelation ***R*** på ***S*** inducerer en opdeling af ***S*** i disjunkte delmængder, kaldet **ækvivalensklasser**

Eksempel på ækvivalensrelation

Forbindelsesrelation imellem knuderne i en graf:

Lad ***V*** være mængden af knuder i en graf ***G***

Definer relationen

$C = \{(v,w) \in V \times V :$ der findes en vej fra v til w i $G\}$

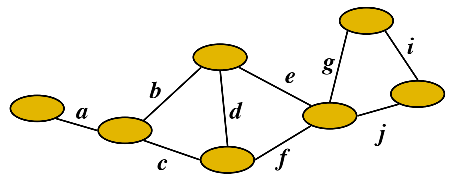
Relationen ***C*** er en ækvivalensrelation

En ækvivalensklasse udgøres af knuderne i en sammenhængende komponent i ***G***

Refleksivitet: $(x,x) \in C$
Symmetri: $(x,y) \in C \Rightarrow (y,x) \in C$
Transitivitet: $(x,y) \in C \wedge (y,z) \in C \Rightarrow (x,z) \in C$

Sammenkædet-relation

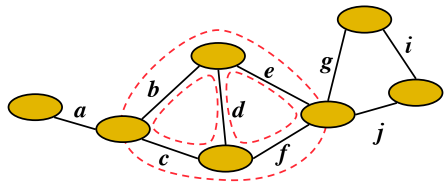
To kanter ***e*** og ***f*** i en sammenhængende graf ***G*** er **sammenkædet**, hvis
- $e = f$, eller
- ***G*** har en simpel cykel, der indeholder både ***e*** og ***f***



Sætning:
Sammenkædet-relationen på kanter i en graf er en ækvivalensrelation

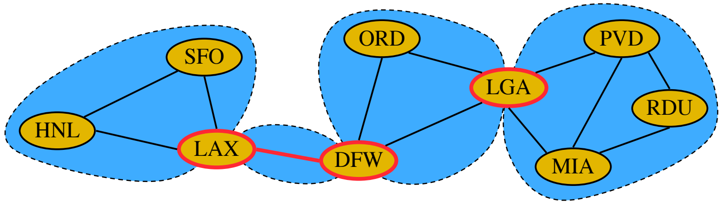
Ækvivalensklasser af sammenkædede kanter:
$\{a\}$ $\{b, c, d, e, f\}$ $\{g, i, j\}$

Skitse af bevis:
De refleksive og symmetriske egenskaber følger af definitionen. For at påvise den transitive egenskab betragtes to simple cykler, der har en kant til fælles



Sammenkædet-komponenter

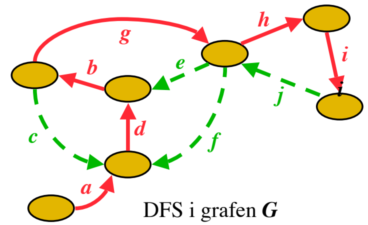
- Sammenkædet-komponenterne i en sammenhængende graf ***G*** er ækvivalensklasserne af kanter med hensyn til sammenkædet-relationen
- En 2-sammenhængende komponent i ***G*** er den delgraf af ***G***, der induceres af en ækvivalensklasse af sammenkædede kanter
- En adskillelseskant er en ækvivalensklasse med 1 element
- En adskillelsesknude har nabokanter i mindst to forskellige ækvivalensklasser



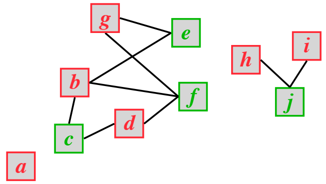
Hjælpegraf

Hjælpegraf ***B*** for en sammenhængende graf ***G***:

- associeret med en DFS-traversering af ***G***
- knuderne i ***B*** er kanterne i ***G***
- for enhver **tilbage**-kant ***e*** i ***G*** har ***B*** kanterne $(e,f_1), (e,f_2), \ldots, (e,f_k)$, hvor $f_1, f_2, \ldots, f_k$ er de besøgte kanter i ***G***, som sammen med ***e*** udgør en simpel cykel
- dens sammenhængende komponenter svarer til sammenkædet-komponenterne i ***G***



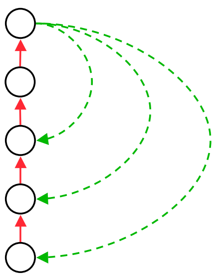
DFS i grafen ***G***



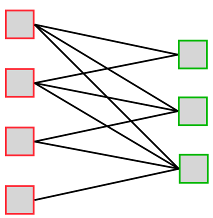
Hjælpegraf ***B***

Hjælpegraf (fortsat)

I værste tilfælde er antallet af kanter i hjælpegrafen $O(nm)$



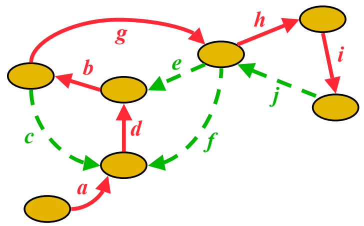
DFS i en graf



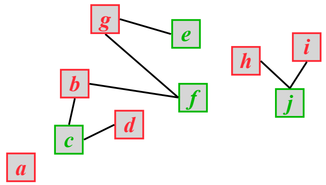
Hjælpegraf

Proxygraf

```
Algorithm proxyGraph(G)
  Input connected graph G
  Output proxy graph F for G
 F ← empty graph
 DFS(G, s) { s is any vertex of G }
 for all discovery edges e of G
   F.insertVertex(e)
   setLabel(e, UNLINKED)
 for all vertices v of G in DFS visit order
   for all back edges e = (u,v)
     F.insertVertex(e)
     repeat
       f ← discovery edge with dest. u
       F.insertEdge(e,f,∅)
       if getLabel(f) = UNLINKED then
         setLabel(f, LINKED)
         u ← origin of edge f
       else
         u ← v { ends the loop }
     until u = v
 return F
```



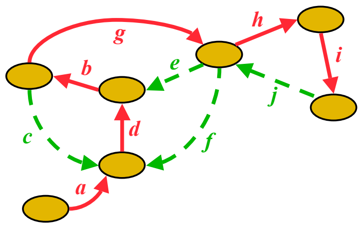
DFS i grafen ***G***



Proxygraf ***F***

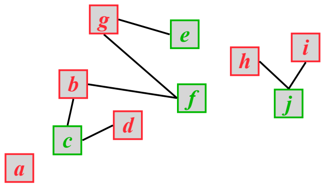
Proxygraf (fortsat)

Proxygraf F for en sammenhængende graf G
- er en udspændende skov i hjælpegrafen B
- har m knuder og $O(m)$ kanter
- kan konstrueres i $O(n+m)$ tid
- dens sammenhængende komponenter (træer) svarer til sammenkædet-komponenterne i G



DFS i grafen G

Givet en graf G med n knuder og m kanter kan vi i $O(n+m)$ tid konstruere
- de 2-sammenhængende komponenter i G
- adskillelsesknuderne i G
- adskillelseskanterne i G



Proxygraf F

Bredde-først-søgning

Bredde-først-søgning

Bredde-først-søgning (BFS) er en generel teknik til traversering af en graf

En bredde-først traversering af grafen G
- besøger alle knuder og kanter i G
- afgør om G er sammenhængende
- bestemmer de sammenhængende komponenter i G
- bestemmer en udspændende skov i G

BFS i en graf med n knuder og m kanter tager $O(n+m)$ tid

BFS kan udvides til at løse andre grafproblemer:
- Find en vej med et *minimalt* antal kanter imellem to knuder
- Find en cykel i grafen

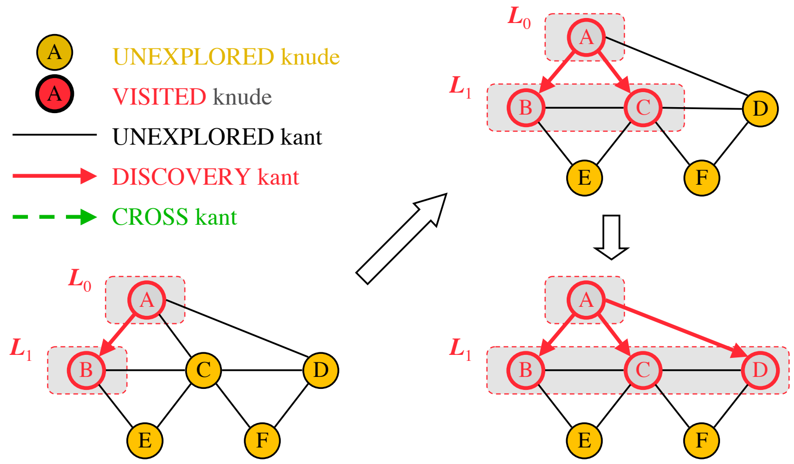
BFS-algoritme

Algoritmen bruger en mekanisme til tildele og aflæse "etiketter" på knuder og kanter

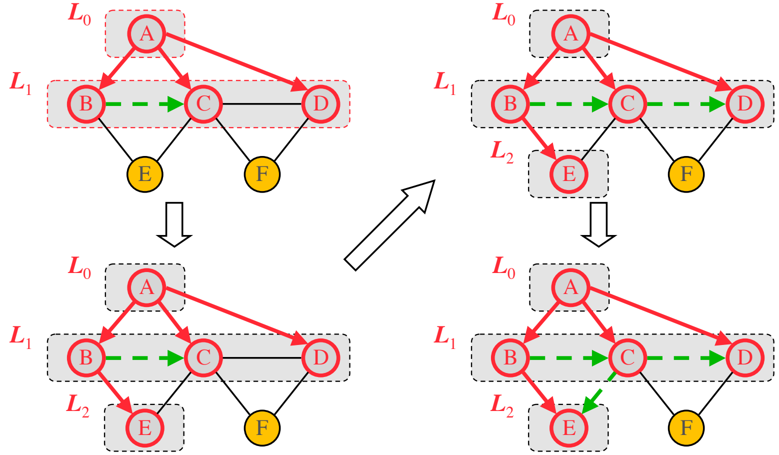
```
Algorithm BFS(G)
    Input graph G
    Output labeling of the edges
        and partition of the
        vertices of G
  for all u ∈ G.vertices()
    setLabel(u, UNEXPLORED)
  for all e ∈ G.edges()
    setLabel(e, UNEXPLORED)
  for all v ∈ G.vertices()
    if getLabel(v) = UNEXPLORED
        BFS(G, v)
```

```
Algorithm BFS(G, s)
  L_0 ← new empty sequence
  L_0.insertLast(s)
  setLabel(s, VISITED)
  i ← 0
  while ¬L_i.isEmpty()
    L_{i+1} ← new empty sequence
    for all v ∈ L_i.elements()
      for all e ∈ G.incidentEdges(v)
        if getLabel(e) = UNEXPLORED
          w ← opposite(v,e)
          if getLabel(w) = UNEXPLORED
            setLabel(e, DISCOVERY)
            setLabel(w, VISITED)
            L_{i+1}.insertLast(w)
          else
            setLabel(e, CROSS)
    i ← i +1
```

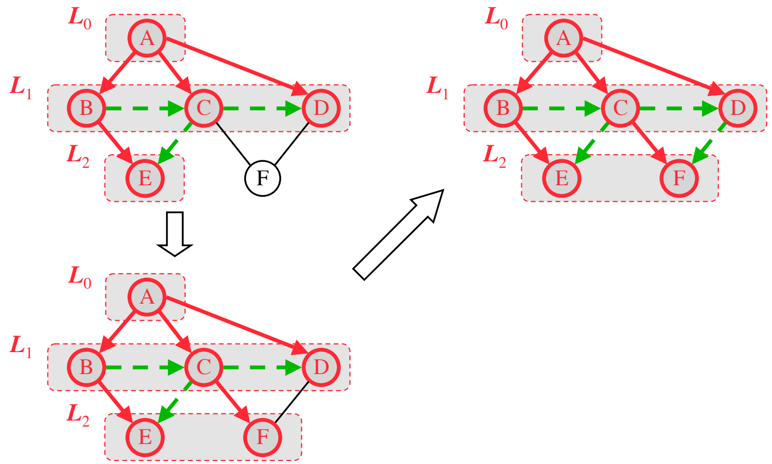
Eksempel



Eksempel (fortsat)



Eksempel (fortsat)



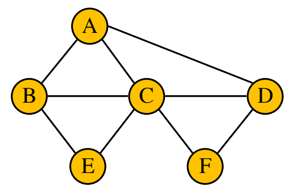
Egenskaber ved BFS

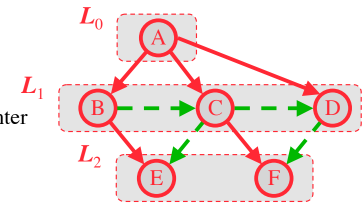
Notation
G_s: sammenhængende komponent, som knuden s tilhører

Egenskab 1
$BFS(G, s)$ besøger alle knuder og kanter i G_s

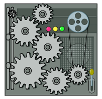
Egenskab 2
De besøgte kanter af $BFS(G, s)$ udgør et udspændende træ T_s af G_s

Egenskab 3
For enhver knude v i L_i gælder, at
- vejen i T_s fra s til v har i kanter
- enhver vej fra s til v i G_s har mindst i kanter





Analyse af BFS



- Tildeling og aflæsning af en knude- eller kant-etikette tager $O(1)$ tid
- Hvert knude får tildelt en etikette to gange
  første gang som UNEXPLORED
  anden gang som VISITED
- Hver kant får tildelt en etikette to gange
  første gang som UNEXPLORED
  anden gang som DISCOVERY eller CROSS
- Hver knude indsættes i en sekvens L_i
- Metoden **incidentEdges** kaldes en gang for hver knude
- BFS kører i $O(n + m)$ tid, forudsat at grafen er repræsenteret med en nabolistestruktur ($\Sigma_v \deg(v) = 2m$)

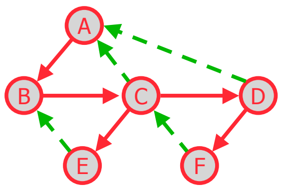
Anvendelser

Vi kan specialisere BFS-traversering af en graf G til at løse følgende problemer i $O(n + m)$ tid:

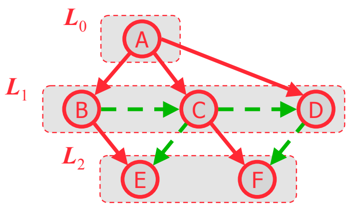
- Find de sammenhængende komponenter i G
- Find den udspændende skov i G
- Find en simple cykel i G - eller rapporter, at G er en skov
- Givet to knuder i G. Find en vej imellem dem med et minimalt antal kanter - eller rapporter, at en sådan vej ikke eksisterer

DFS versus BFS

Anvendelser	DFS	BFS
Udspændende skov, sammenhængende skov, veje, cykler	√	√
Korteste veje		√
2-sammenhængende komponenter	√	



DFS

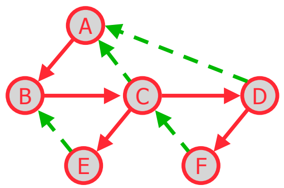


BFS

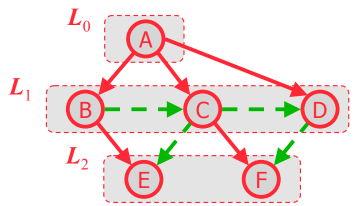
DFS versus BFS (fortsat)

Tilbage-kant (v,w)
w er en forfader til v i træet af besøgte kanter

Tvær-kant (v,w)
w er på samme niveau som v, eller på det næste niveau i træet af besøgte kanter



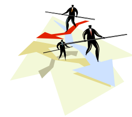
DFS



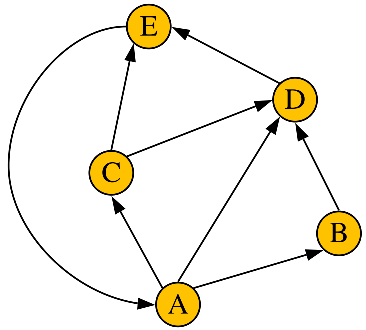
BFS

Digrafer

Digraf



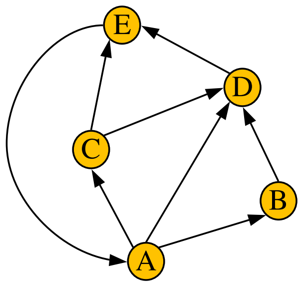
En **digraf** er en graf, hvor alle kanter er orienteret
(forkortelse for "directed graph")



Anvendelser:

- ensrettede gader
- flyvninger
- projektplanlægning

Egenskaber ved digrafer



En graf $\boldsymbol{G}$=($\boldsymbol{V}$,$\boldsymbol{E}$), hvor hver kant går i en retning:
En kant (a,b) går fra a til b, men **ikke** fra b til a

Hvis $\boldsymbol{G}$ er simpel, $m \leq n(n\text{-}1)$

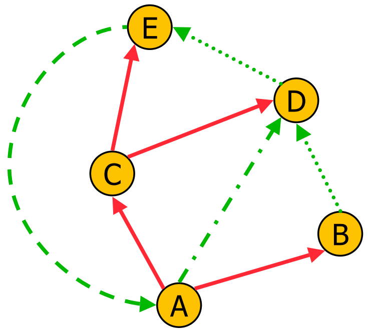
Hvis vi holder ind-kanter og ud-kanter i særskilte nabolister, kan vi opremse ind- og ud-kanter i tid proportional med deres antal

Orienteret DFS

Vi kan specialisere traverserings-algoritmerne (DFS og BFS) til digrafer ved kun at traversere kanterne i deres retning

Ved orienteret DFS har vi fire type af kanter:

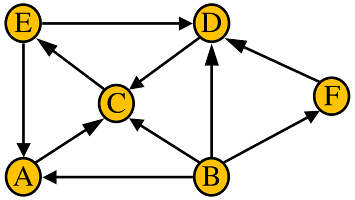
- besøgte kanter
- tilbage-kanter
- fremad-kanter
- tvær-kanter

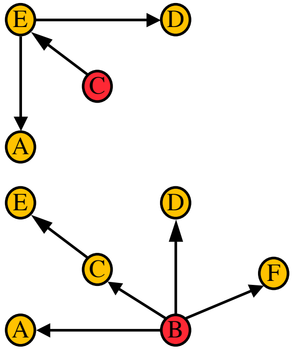


En orienteret DFS, der starter i knuden $\boldsymbol{s}$ bestemmer de kanter, der kan nås fra $\boldsymbol{s}$

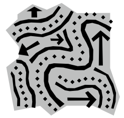
Forbundethed

DFS-træ med rod i ***v***: de knuder, der kan nås fra ***v*** ved hjælp af orienterede veje

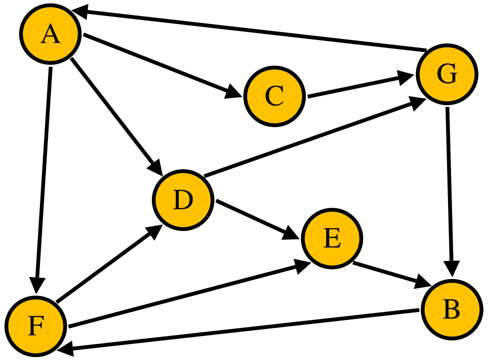




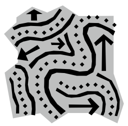
Stærk sammenhæng



Fra enhver knude kan alle andre knuder nås



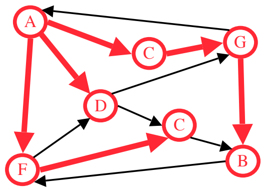
Algoritme til afgørelse af stærk sammenhæng



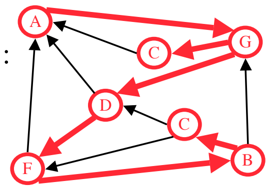
- Vælg en knude ***v*** i ***G***
- Udfør DFS from ***v*** i ***G***
  Hvis der er en knude ***w***, der ikke er besøgt, så udskriv "nej"
- Lad ***G'*** være ***G*** med kanterne vendt om
- Udfør DFS fra ***v*** i ***G'***
  Hvis der er en knude **w**, der ikke er besøgt, så udskriv "nej"
  Ellers, udskriv "ja"

Køretid: $O(n + m)$

G:



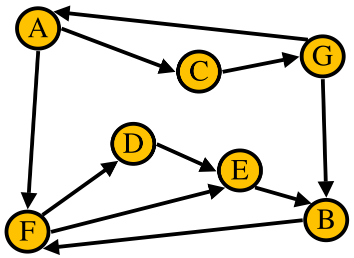
G':



Stærkt sammenhængende komponenter

Maksimale delgrafer, hvor der fra enhver knude kan nås fra alle andre knuder i delgrafen

Kan også udføres i $O(n + m)$ tid ved brug af DFS, men er mere kompliceret (i lighed med til 2-sammenhæng)



{ A , C , G }

{ F , D , E , B }

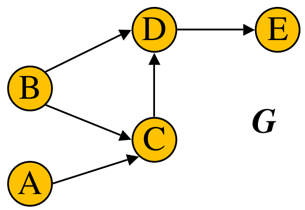
Transitiv afslutning

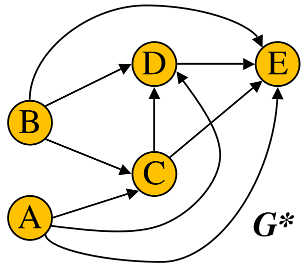
Lad der være givet en digraf G

Den **transitive afslutning** af G er den digraf G^*, hvor

- G^* har de samme knuder som G
- Hvis G har en orienteret vej fra u til v ($u \neq v$), så har G^* en orienteret kant fra u til v

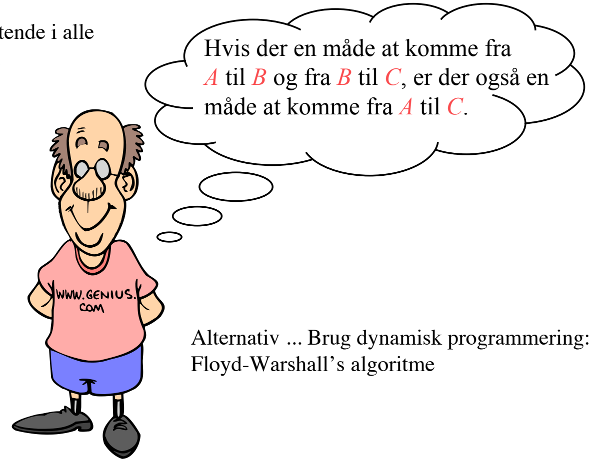
Den transitive afslutning giver hurtig oplysning om forbundethed i en digraf





Bestemmelse af den transitive afslutning

Vi kan udføre DFS startende i alle knuder i $O(n(n+m))$ tid



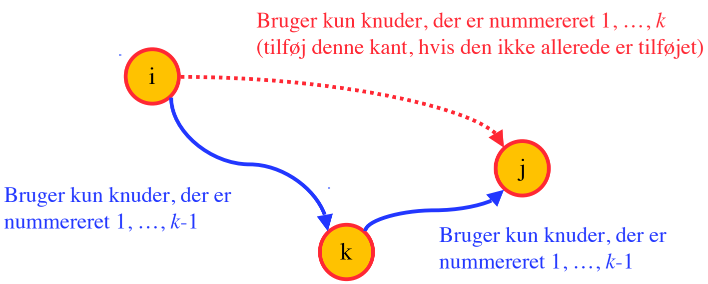
Alternativ ... Brug dynamisk programmering: Floyd-Warshall's algoritme

Floyd-Warshall's algoritme

Ide 1: Nummerer knuderne $1, 2, \ldots, n$

Ide 2: Betragt veje, der kun bruger de knuder, der er nummereret $1, 2, \ldots, k$ som *mellemliggende* knuder:

(Dynamisk programmering)



Floyd-Warshall's algoritme

Floyd-Warshall's algoritme nummererer knuderne i G som $v_1, \ldots, v_n$ og bestemmer en række af digrafer $G_0, \ldots, G_n$

- $G_0 = G$
- G_k har en orienteret kant (v_i, v_j), hvis G har en orienteret vej v_i til v_j med mellemliggende knuder fra mængden $\{v_1, \ldots, v_k\}$

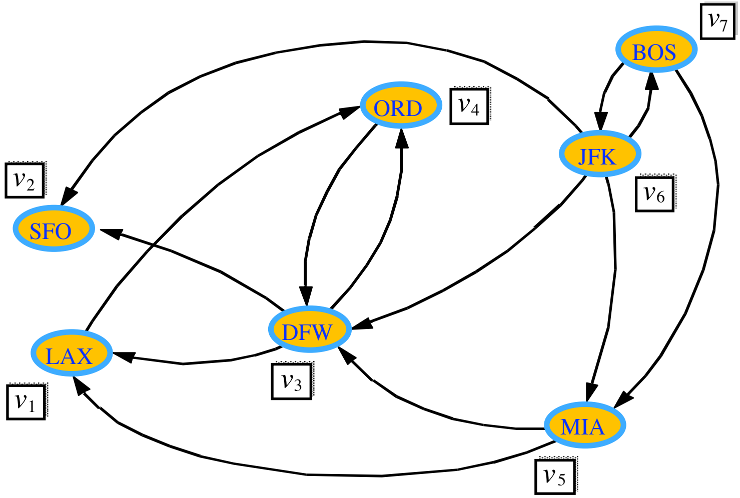
Vi har, at $G_n = G^*$

I iteration k, bestemmes G_k ud fra G_{k-1}

Køretid: $O(n^3)$, hvis operationen **areAdjacent** er $O(1)$ (f.eks., hvis der benyttes en nabomatrix)

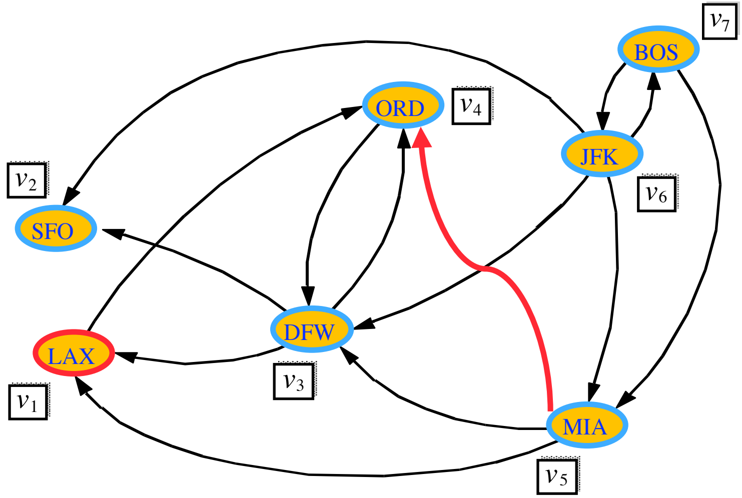
Algorithm *FloydWarshall*(G)
  Input digraph G
  Output transitive closure G^* of G
  $i \leftarrow 1$
  for all $v \in G.vertices()$
    denote v as v_i
    $i \leftarrow i + 1$
  $G_0 \leftarrow G$
  for $k \leftarrow 1$ **to** n **do**
    $G_k \leftarrow G_{k-1}$
    for $i \leftarrow 1$ **to** n $(i \neq k)$ **do**
      for $j \leftarrow 1$ **to** n $(j \neq i, k)$ **do**
        if $G_{k-1}.areAdjacent(v_i, v_k) \wedge$
          $G_{k-1}.areAdjacent(v_k, v_j)$
          if $\neg G_k.areAdjacent(v_i, v_j)$
            $G_k.insertDirectedEdge(v_i, v_j, k)$
  return G_n

Floyd-Warshall eksempel



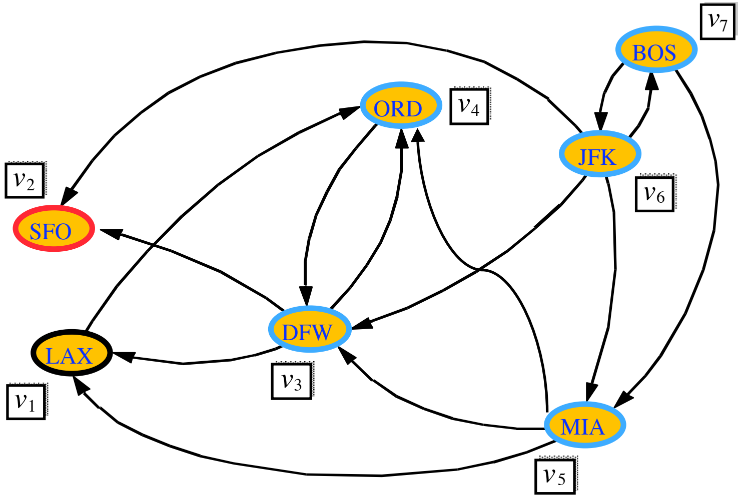
69

Floyd-Warshall (iteration 1)



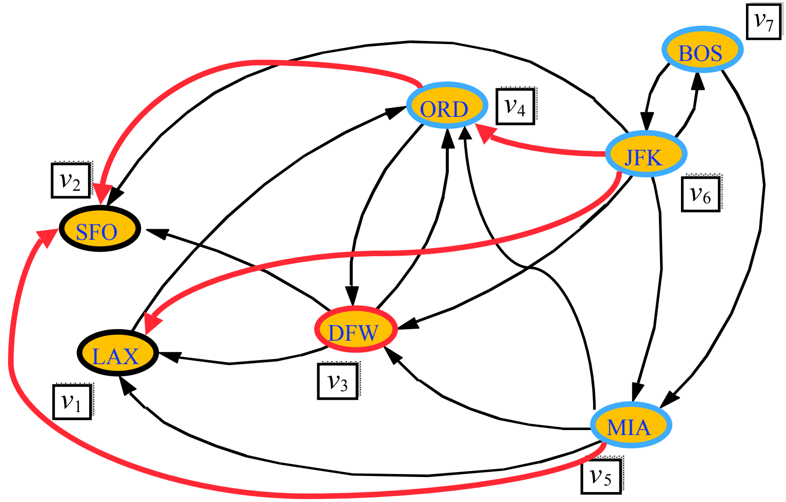
70

Floyd-Warshall (iteration 2)



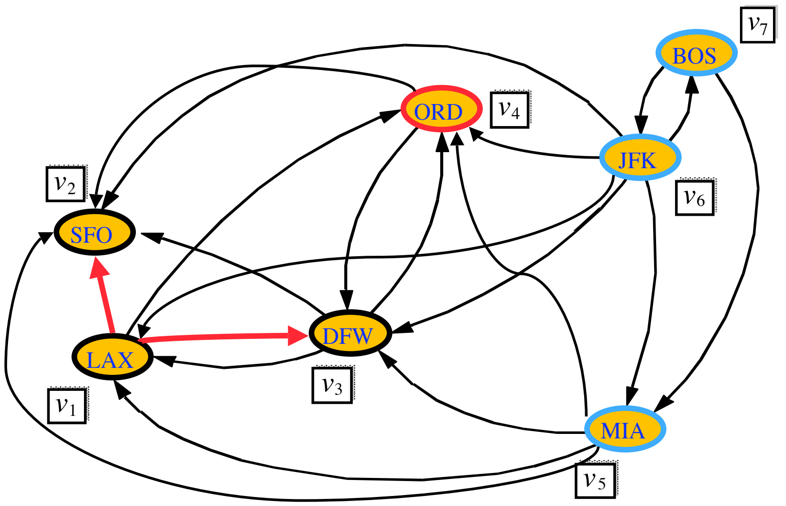
71

Floyd-Warshall (iteration 3)

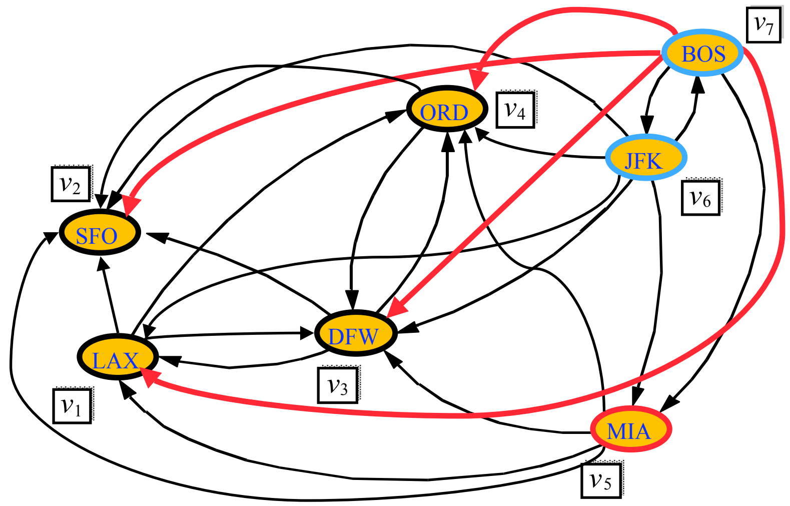


72

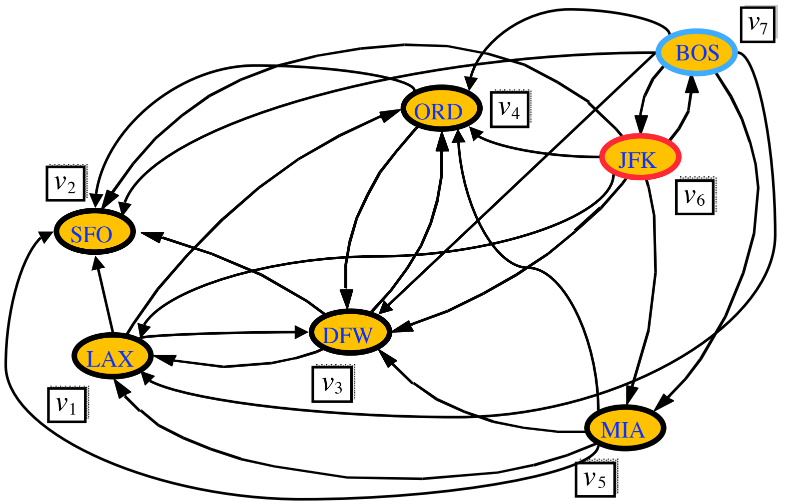
Floyd-Warshall (iteration 4)



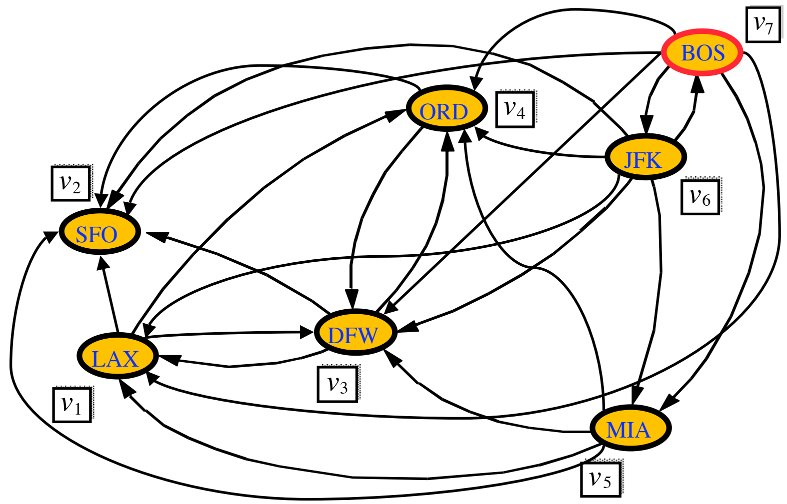
Floyd-Warshall (iteration 5)



Floyd-Warshall (iteration 6)



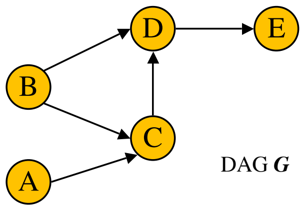
Floyd-Warshall (slut)



DAG og topologisk ordning

En orienteret ikke-cyklisk graf, **DAG** (Directed Acyclic Graph) er en digraf, der ikke har nogen orienterede cykler

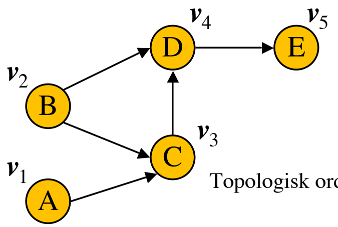
En **topologisk ordning** af en digraf er en nummerering, $v_1, \ldots, v_n$, af knuderne, således at der for enhver kant, (v_i, v_j), gælder, at $i < j$



DAG ***G***

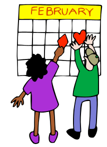
Eksempel:
I en opgaveplanlægningsgraf er en topologisk ordning en rækkefølge af opgaver, der for hver opgave tilfredsstiller startforudsætningerne (eng. the precedence constraints)

Sætning: En digraf tillader topologisk ordning, hvis og kun hvis den er en DAG



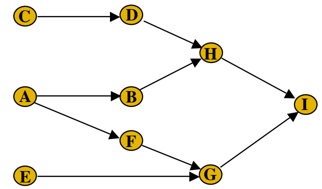
Topologisk ordning af ***G***

Netværksplanlægning



aktivitet	*forgænger(e)*	*varighed (uger)*
A konstruer lagerstyringsmodel	**-**	**4**
B implementer lagerstyringsprogram	**A**	**13**
C konstruer prognosemodel	**-**	**4**
D implementer prognoseprogram	**C**	**15**
E indsaml data	**-**	**12**
F design database	**A**	**4**
G implementer database	**E, F**	**2**
H oplær personale	**B, D**	**2**
I afprøv system	**G, H**	**2**

aktivitetsnetværk



Topologisk sortering ved hjælp af DFS

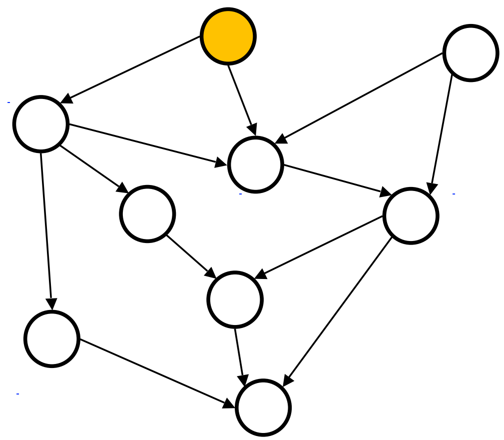
Simuler algoritmen ved brug af dybde-først-søgning

```
Algorithm topologicalDFS(G)
   Input dag G
   Output topological ordering of G
  n ← G.numVertices()
  for all u ∈ G.vertices()
     setLabel(u, UNEXPLORED)
  for all e ∈ G.edges()
     setLabel(e, UNEXPLORED)
  for all v ∈ G.vertices()
    if getLabel(v) = UNEXPLORED
       topologicalDFS(G, v)
```

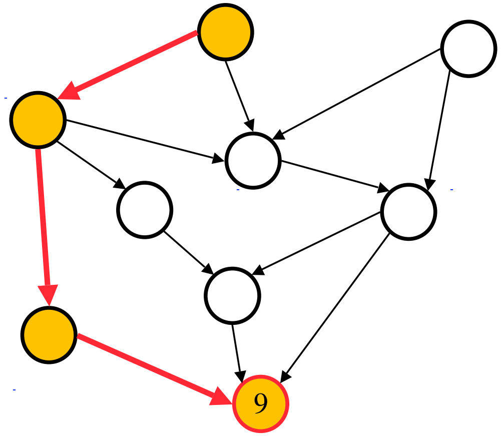
O(n+m) tid

```
Algorithm topologicalDFS(G, v)
  Input graph G and a start vertex v of G
  Output labeling of the vertices of G
    in the connected component of v
  setLabel(v, VISITED)
  for all e ∈ G.incidentEdges(v)
    if getLabel(e) = UNEXPLORED
      w ← opposite(v,e)
      if getLabel(w) = UNEXPLORED
        setLabel(e, DISCOVERY)
        topologicalDFS(G, w)
      else
        {e is a forward or cross edge}
  Label v with topological number n
  n ← n - 1
```

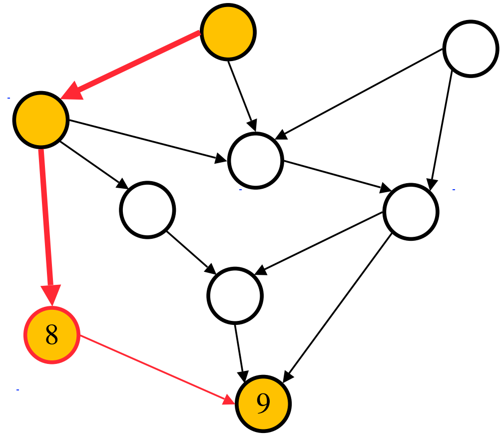

Eksempel på topologisk sortering



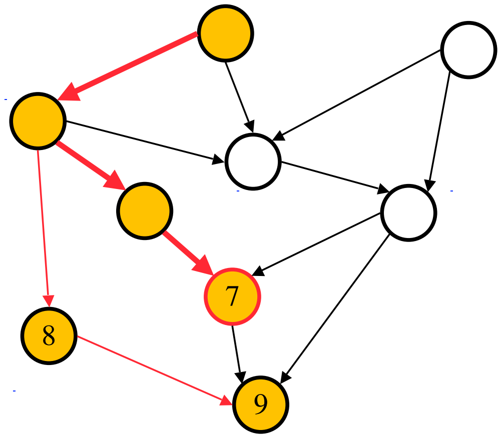
Eksempel på topologisk sortering



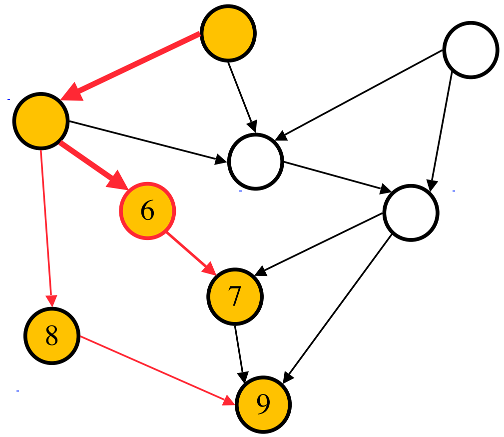
Eksempel på topologisk sortering



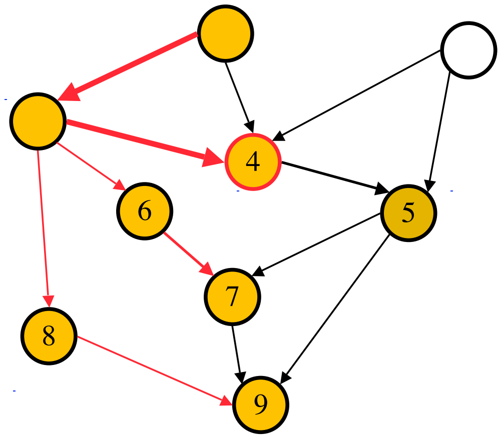
Eksempel på topologisk sortering



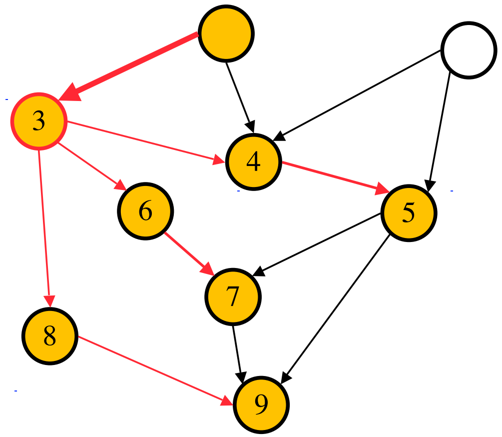
Eksempel på topologisk sortering



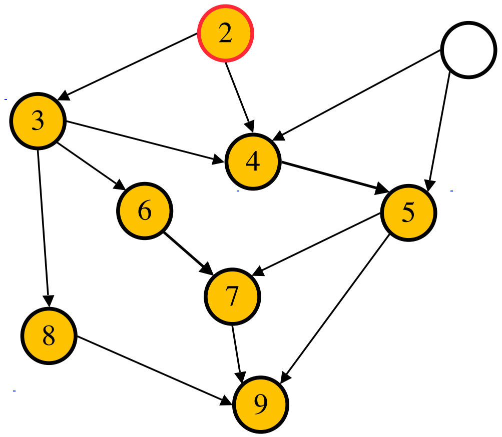
Eksempel på topologisk sortering



Eksempel på topologisk sortering



Eksempel på topologisk sortering



Eksempel på topologisk sortering