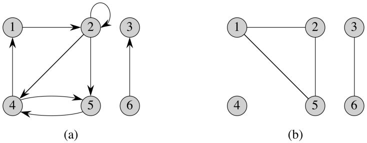
Grafer og graf-gennemløb

Grafer

En mængde V af *knuder* (vertices).

En mængde $E \in V \times V$ af *kanter* (edges). Dvs. par af knuder.



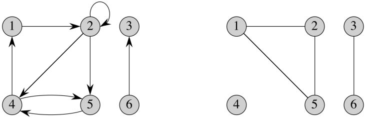
- Orienterede grafer: kanter er ordnede par.
- Uorienterede grafer: kanter er uordnede par.
- Vægtede grafer: hver kant har et tal tilknyttet.
- Notation: $n = |V|$, $m = |E|$.
- Bemærk at $0 \leq m \leq n^2$ for orienterede grafer og $0 \leq m \leq (n^2 + n)/2$ for uorienterede grafer.

Læs yderligere om graf-terminologi i de to første sider af appendix B i lærebogen.

Grafer

Modeller for mange ting:

- Ledningsnet (telefon, strøm, olie, vand,...).

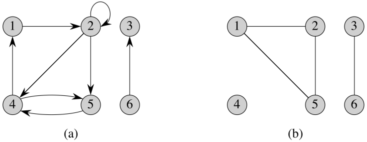


(a) (b)

Grafer

Modeller for mange ting:

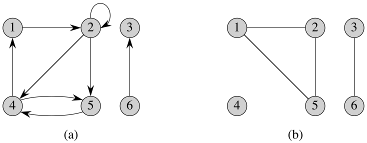
- Ledningsnet (telefon, strøm, olie, vand,...).
- Vejnet (vejkryds er knuder, veje mellem kryds er kanter)



Grafer

Modeller for mange ting:

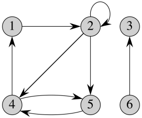
- Ledningsnet (telefon, strøm, olie, vand,. . .).
- Vejnet (vejkryds er knuder, veje mellem kryds er kanter)
- Venner på SoMe.



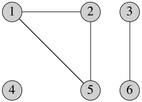
Grafer

Modeller for mange ting:

- Ledningsnet (telefon, strøm, olie, vand,. . .).
- Vejnet (vejkryds er knuder, veje mellem kryds er kanter)
- Venner på SoMe.
- Følgere på SoMe.



(a)

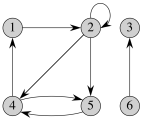


(b)

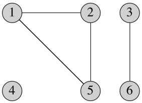
Grafer

Modeller for mange ting:

- Ledningsnet (telefon, strøm, olie, vand,...).
- Vejnet (vejkryds er knuder, veje mellem kryds er kanter)
- Venner på SoMe.
- Følgere på SoMe.
- WWW-sider.



(a)

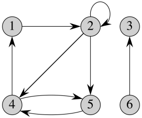


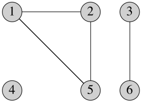
(b)

Grafer

Modeller for mange ting:

- Ledningsnet (telefon, strøm, olie, vand,...).
- Vejnet (vejkryds er knuder, veje mellem kryds er kanter)
- Venner på SoMe.
- Følgere på SoMe.
- WWW-sider.
- Medforfatterskaber.



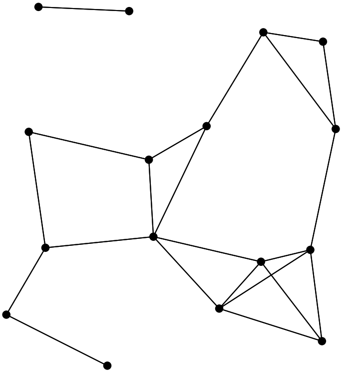


Masser af algoritmiske spørgsmål på grafer

- Hvordan repræsentere grafer på en computer (datastruktur)?
- Findes der en sti mellem to angivne knuder?
- Hvad er en korteste sti mellem to angivne knuder?
- Hvad er en mindste delmængde af kanter, som stadig holder alle knuder forbundet?
- Hvad er en største samling kanter, hvor ingen kanter har fælles knuder?
- $\vdots$

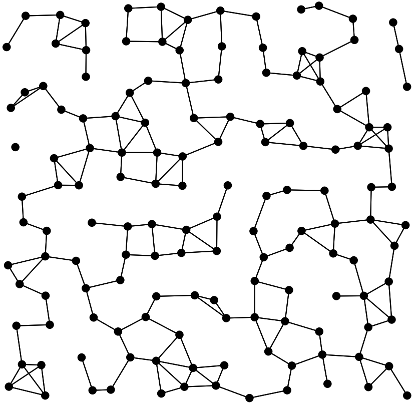
Eksempel på algoritmisk spørgsmål

Afgør om der findes en sti mellem to givne knuder.



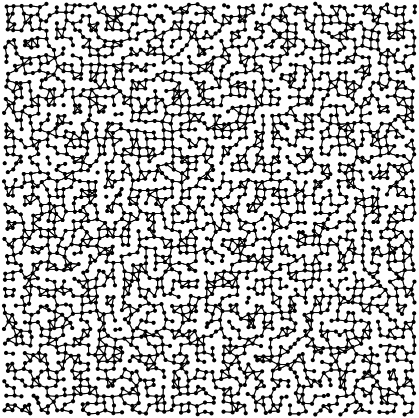
Eksempel på algoritmisk spørgsmål

Afgør om der findes en sti mellem to givne knuder.



Eksempel på algoritmisk spørgsmål

Afgør om der findes en sti mellem to givne knuder.

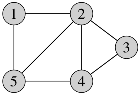


Behov for hjælp fra computer. Behov for algoritme til at løse problemet.

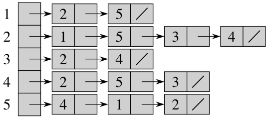
Datastrukturer for grafer

Datastrukturer for grafer

Adjacency lists og adjacency matrix



(a)



(b)

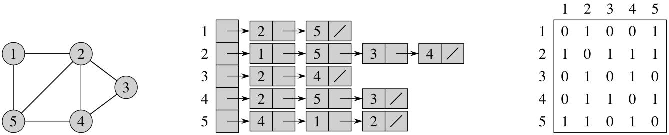
	1	2	3	4	5
1	0	1	0	0	1
2	1	0	1	1	1
3	0	1	0	1	0
4	0	1	1	0	1
5	1	1	0	1	0

(c)

Adjacency lists: listen for u indeholder v for alle kanter $(u, v) \in E$.
Knuder er repræsenteret som heltal mellem 1 og n (eller 0 og $n - 1$).

Datastrukturer for grafer

Adjacency lists og adjacency matrix



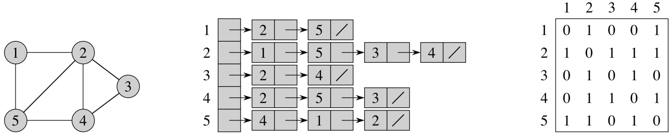
(a) (b) (c)

Adjacency lists: listen for u indeholder v for alle kanter $(u, v) \in E$.
Knuder er repræsenteret som heltal mellem 1 og n (eller 0 og $n - 1$).

Plads: $O(n + m)$ for adjacency lists, $O(n^2)$ for adjacency matrix.

Datastrukturer for grafer

Adjacency lists og adjacency matrix



(a) (b) (c)

Adjacency lists: listen for u indeholder v for alle kanter $(u, v) \in E$.
Knuder er repræsenteret som heltal mellem 1 og n (eller 0 og $n - 1$).

Plads: $O(n + m)$ for adjacency lists, $O(n^2)$ for adjacency matrix.

Hvis ikke andet oplyses, bruges adjacency lists repræsentationen i algoritmer i dette kursus.

En kant i en uorienterede graf repræsenteres som to orienterede kanter (så mht. implementation er uorienterede grafer bare et specialtilfælde af orienterede grafer).

Grafgennemløb

Opgave: givet en graf i adjacency lists repræsentation, besøg alle knuder og kanter. Målet er, at lære om forskellige egenskaber for grafen.

Generel idé: Besøg en startknude s. Brug kanter i nabolisterne for besøgte knuder til at besøge flere knuder.

Marker knuder undervejs for at holde styr på processen:

- Hvide knuder: endnu ikke besøgt
- Grå knuder: besøgt, men ikke alle kanter i naboliste brugt
- Sorte knuder: besøgt, alle kanter i naboliste brugt

Grafgennemløb

Generisk algoritme til grafgennemløb:

```
GenericGraphTraversal1(s)
    Gør s grå og resten af knuderne hvide
    while der findes grå knuder:
        vælg en grå knude v
        if v's naboliste er brugt op
            gør v sort
        else
            vælg en ubrugt kant (v, u) fra v's naboliste
            if u hvid:
                gør u grå
```

En knudes livs-cyklus: hvid $\rightarrow$ grå $\rightarrow$ sort. Når algoritmen stopper, er alle knuder enten hvide eller sorte.

Grafgennemløb

Vi skal senere i kurset møde tre varianter, som har forskellige strategier for at vælge næste kant (v, u) at bruge, dvs. for valgene (*):

```
GenericGraphTraversal1(s)
    Gør s grå og resten af knuderne hvide
    while der findes grå knuder:
        vælg en grå knude v (*)
        if v's naboliste er brugt op
            gør v sort
        else
            vælg en ubrugt kant (v, u) fra v's naboliste (*)
            if u hvid:
                gør u grå
```

- Breadth-First-Search (BFS)
- Depth-First-Search (DFS)
- Priority-Search (Dijkstras algoritme, A*)

Hvor langt når vi rundt i grafen?

Vi når alt, som kan nås fra s:

Sætning: Hvis der er en sti fra s til v, vil v være sort (og dermed besøgt) når $\textsc{GenericGraphTraversal1}(s)$ stopper.

Bevis: Når algoritmen stopper, er alle knuder enten hvide eller sorte. Da s startede grå, må den nu være sort. Antages at v er hvid, må der være mindst én kant (u, w) på stien med u sort og w hvid. Men u kan kun være sort hvis (u, w) er blevet brugt, hvorved w blev grå og nu må være sort. Så antagelsen kan ikke gælde og v må være sort.

For at nå rundt i *hele* grafen:

```
GenericGraphTraversalGlobal()
    Gør alle knuder hvide
    for alle knuder s:
        if s hvid:
                GenericGraphTraversal2(s)
```

```
GenericGraphTraversal2(s)
    Gør s grå
    while der findes grå knuder:
        vælg en grå knude v (*)
        if v's naboliste er brugt op
            gør v sort
        else
            vælg en ubrugt kant (v, u) fra v's naboliste (*)
            if u hvid:
                gør u grå
```

Hvis (*) tager tid $O(1)$, er samlet køretid $O(n+m)$. [En kant kan kun vælges én gang, så alt arbejde udført i **else**-del tager $O(m)$ tid i alt. Resten tager $O(n)$ tid i alt.]

Hvor langt når vi rundt i grafen per kald?

Sætning: Hvis der ved starten af et kald til $\textsc{GenericGraphTraversal2}(s)$ er en sti fra s til v bestående af hvide knuder (inkl. v), vil v være sort (og dermed besøgt) når $\textsc{GenericGraphTraversal2}(s)$ stopper.

Bevis: Det samme som før for $\textsc{GenericGraphTraversal1}(s)$.

Husk hvem der opdagede hvem:

Når en knude u ($\neq s$) besøges første gang, gemmer den i variablen $u.\pi$ knuden, som opdagede u (predecessor). Bemærk, at $u.\pi$ højst bliver sat én gang (efter initialisering til NIL), da u gøres grå samtidig.

```
GenericGraphTraversalGlobalWithParents()
    Gør alle knuder hvide og sæt deres π til NIL
    for alle knuder s:
        if s hvid:
            GenericGraphTraversal3(s)
```

```
GenericGraphTraversal3(s)
    Gør s grå
    while der findes grå knuder:
        vælg en grå knude v (*)
        if v's naboliste er brugt op
            gør v sort
        else
            vælg en ubrugt kant (v, u) fra v's naboliste (*)
            if u hvid:
                gør u grå
                sæt u.π lig v
```

Husk hvem der opdagede hvem:

Sætning: De knuder, som er opdaget (gjort ikke-hvide) i et kald $\textsc{GenericGraphTraversal3}(s)$, udgør et træ med s som rod og π i opdagede knuder som parent pointers. For hver sti fra en knude v til roden i træet findes den samme sti i grafen, men i modsat retning (fra s til v).

Bevis: Det er nemt at se, at dette udsagn er en invariant som vedligeholdes under kørslen af $\textsc{GenericGraphTraversal3}(s)$.

Husk hvem der opdagede hvem:

Sætning: De knuder, som er opdaget (gjort ikke-hvide) i et kald $\textsc{GenericGraphTraversal3}(s)$, udgør et træ med s som rod og π i opdagede knuder som parent pointers. For hver sti fra en knude v til roden i træet findes den samme sti i grafen, men i modsat retning (fra s til v).

Bevis: Det er nemt at se, at dette udsagn er en invariant som vedligeholdes under kørslen af $\textsc{GenericGraphTraversal3}(s)$.

Bemærk, at i $\textsc{GenericGraphTraversalGlobalWithParents}()$ kaldes $\textsc{GenericGraphTraversal3}(s)$ gentagne gange. Hvert kald giver ét træ. Træerne fra forskellige kald deler ikke knuder, og tilsammen indeholder de alle knuder i grafen.

Bredde-Først-Søgning (BFS)

Strategi: Hold de grå knuder i en KØ, brug nabolister op med det samme.

Tilføj også en variabel $v.d$ til alle knuder v (d for distance.)

Bredde-Først-Søgning (BFS)

Strategi: Hold de grå knuder i en KØ, brug nabolister op med det samme.

Tilføj også en variabel $v.d$ til alle knuder v (d for distance.)

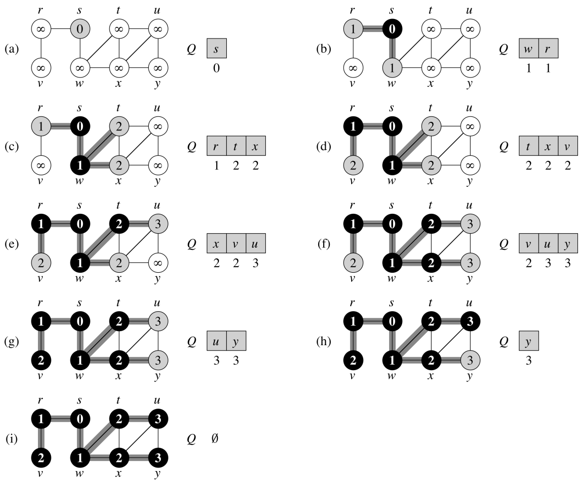
Mest brugt er versionen uden GLOBAL-del (for BFS er vi ofte mere interesserede i ét bestemt s fremfor at komme hele grafen rundt):

```
BFS(G, s)
for each vertex u ∈ G.V − {s}
    u.color = WHITE
    u.d = ∞
    u.π = NIL
s.color = GRAY
s.d = 0
s.π = NIL
Q = ∅
ENQUEUE(Q, s)
while Q ≠ ∅
    u = DEQUEUE(Q)
    for each v ∈ G.Adj[u]
        if v.color == WHITE
            v.color = GRAY
            v.d = u.d + 1
            v.π = u
            ENQUEUE(Q, v)
    u.color = BLACK
```

Invariant:
kø = alle grå knuder.

Bredde-Først-Søgning (BFS)

Eksempel:



Bredde-Først-Søgning (BFS)

For BFS kan sætningen om $\textsc{GenericGraphTraversal3}(s)$ udvides til også at sige noget om værdierne af $v.d$:

Sætning: De knuder, som er opdaget (gjort ikke-hvide) i et kald $\textsc{GenericGraphTraversal3}(s)$, udgør et træ med s som rod og π i opdagede knuder som parent pointers. For hver sti fra en knude v til roden i træet findes den samme sti i grafen, men i modsat retning (fra s til v) og $v.d$ er lig antal kanter på denne sti.

Bevis: Det er nemt at se, at dette udsagn er en invariant som vedligeholdes under kørslen af $\mathrm{BFS}(G, s)$.

Bredde-Først-Søgning (BFS)

For BFS kan sætningen om $\textsc{GenericGraphTraversal3}(s)$ udvides til også at sige noget om værdierne af $v.d$:

Sætning: De knuder, som er opdaget (gjort ikke-hvide) i et kald $\textsc{GenericGraphTraversal3}(s)$, udgør et træ med s som rod og π i opdagede knuder som parent pointers. For hver sti fra en knude v til roden i træet findes den samme sti i grafen, men i modsat retning (fra s til v) og $v.d$ er lig antal kanter på denne sti.

Bevis: Det er nemt at se, at dette udsagn er en invariant som vedligeholdes under kørslen af $\mathrm{BFS}(G, s)$.

Bemærk, at $v.d$ højst bliver sat én gang (efter initialisering til $-\infty$): $v.d$ sættes kun, når v er hvid, og v gøres ikke-hvid samtidig med at $v.d$ sættes.

Egenskaber for BFS

Køretid: $O(n+m)$.

Beviset er det samme som under GenericGraphTraversalGlobal, da valgene (*) i BFS tager $O(1)$ tid. I BFS bruger man som sagt ofte kun at kalde på én startknude s, dvs. uden at bruge Global-delen. Men køretiden kan kun falde ved dette.

Egenskaber for BFS

Køretid: $O(n + m)$.

Beviset er det samme som under $\textsc{GenericGraphTraversalGlobal}$, da valgene (*) i BFS tager $O(1)$ tid. I BFS bruger man som sagt ofte kun at kalde på én startknude s, dvs. uden at bruge $\textsc{Global}$-delen. Men køretiden kan kun falde ved dette.

Definition: Vi definerer $\delta(s, v)$ som længden af en korteste sti, *målt i antal kanter*, fra startknuden s til knuden v. Findes ingen sti, defineres $\delta(s, v) = \infty$.

> Sætning: Når BFS stopper, gælder $v.d = \delta(s, v)$ for alle knuder.

Dvs. BFS kan finde korteste veje (målt i antal kanter) fra s til alle v.

Bevis for sætning

De mulige værdier for $\delta(s, v)$ er $0, 1, 2, 3, \ldots$ samt ∞.

Bevis for sætning

De mulige værdier for $\delta(s, v)$ er $0, 1, 2, 3, \dots$ samt ∞.

For knuder v med $\delta(s, v) = \infty$ findes der ikke en sti fra s til v. Så kan v ikke være opdaget (som vist tidligere er der en sti i grafen fra s til alle opdagede knuder). Derfor kan værdien $v.d = \infty$ sat under initialisering ikke ændres, så når BFS stopper, gælder $v.d = \delta(s, v)$ for disse knuder.

Bevis for sætning

De mulige værdier for $\delta(s, v)$ er $0, 1, 2, 3, \ldots$ samt ∞.

For knuder v med $\delta(s, v) = \infty$ findes der ikke en sti fra s til v. Så kan v ikke være opdaget (som vist tidligere er der en sti i grafen fra s til alle opdagede knuder). Derfor kan værdien $v.d = \infty$ sat under initialisering ikke ændres, så når BFS stopper, gælder $v.d = \delta(s, v)$ for disse knuder.

For resten af knuderne er $\delta(s, v) = i < \infty$. For dem viser vi, via induktion på i, at

$$\delta(s, v) = i$$

$$\Downarrow$$

$$v.d = i \text{ når BFS stopper}$$

Tilsammen giver dette sætningen.

Observationer

1. Pga. virkemåden for en kø vil BFS-algoritmen for $i = 0, 1, 2, 3, \ldots$ udtage alle knuder med d-værdi lig i mens den indsætter alle knuder med d-værdi lig $i + 1$ (og derefter fortsætter den med næste værdi for i).

   Heraf ses, at d-værdierne for de udtagne knuder stiger monotont.
2. Vi ved allerede at $\delta(s, v) \leq v.d$, eftersom vi tidligere har vist, at der er en sti af længde $v.d$ i grafen.

Induktionsbevis

Basis ($i = 0$): Hvis $\delta(s, v) = 0$ er $v = s$. BSF sætter $s.d = 0$.

Induktionsbevis

Basis ($i = 0$): Hvis $\delta(s, v) = 0$ er $v = s$. BSF sætter $s.d = 0$.

Induktionssskridt ($i > 0$): Vi antager, at $\delta(s, v) = j \Rightarrow v.d = j$ er sandt for $j = i - 1$. Vi skal vise at det er sandt for $j = i$.

Induktionsbevis

Basis ($i = 0$): Hvis $\delta(s, v) = 0$ er $v = s$. BSF sætter $s.d = 0$.

Induktionssskridt ($i > 0$): Vi antager, at $\delta(s, v) = j \Rightarrow v.d = j$ er sandt for $j = i - 1$. Vi skal vise at det er sandt for $j = i$.

Hvis $\delta(s, v) = i$, eksisterer en sti fra s til v af længde i. For næstsidste knude u på denne sti gælder $\delta(s, u) = i - 1$ (hvis u havde en kortere vej, ville v også have det).

Fra induktionsantagelsen har vi $u.d = \delta(s, u)$ når BFS stopper. Da u blev taget ud af køen, var v (en nabo til u) enten uopdaget (hvid) og bliver nu opdaget af u, eller v var allerede opdaget fra en knude t, som derfor allerede var taget ud af køen og derfor (via observation 1) har $t.d \leq u.d$.

I BFS tildeles v en d-værdi, som er én større end d-værdien af knuden, som opdager den.

Så hvad enten v bliver optaget af u eller v, bliver $v.d$ derfor sat til *højst* $u.d + 1 = \delta(s, u) + 1 = (i - 1) + 1 = i = \delta(s, v)$. Vi ved (via observation 2) at $v.d$ er *mindst* $\delta(s, v)$. I alt har vi $v.d = \delta(s, v)$.

Dybde-Først-Søgning (DFS)

Strategi: Hold de grå knuder i en STAK, avancér minimalt i deres nabolister per gang.

Dybde-Først-Søgning (DFS)

Strategi: Hold de grå knuder i en STAK, avancér minimalt i deres nabolister per gang.

Stakken er implicit i den rekursive formulering nedenfor (dvs. er lig rekursionsstakken), men kan også kodes eksplicit. Mere præcist: elementerne på stakken er de grå knuder, hver med en delvist gennemløbet naboliste, nemlig gennemløbet i for-løkken i DFS-VISIT. [Bemærk: koden til venstre svarer til GLOBAL-delen i terminologien fra tidligere.]

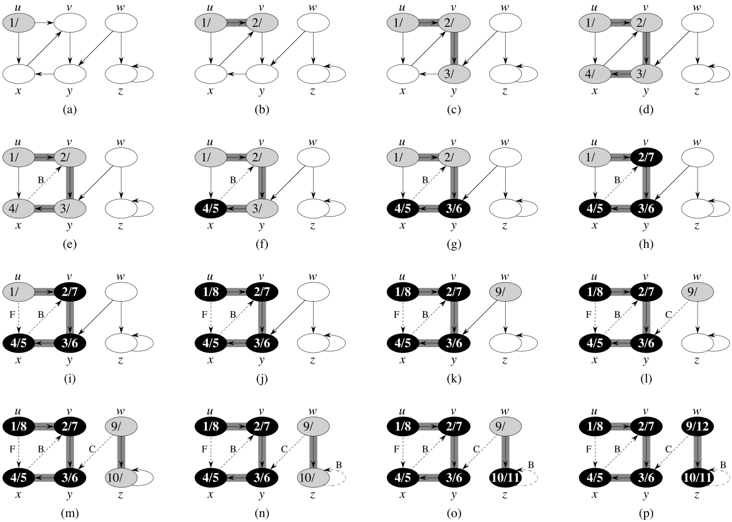
DFS tilføjer også timestamps $u.d$ for "discovery" (hvid $\rightarrow$ grå) og $u.f$ for "finish" (grå $\rightarrow$ sort) til alle knuder u. [$u.d$ er *ikke* "distance" i DFS.]

```
DFS(G)
for each vertex u ∈ G.V
    u.color = WHITE
    u.π = NIL
time = 0
for each vertex u ∈ G.V
    if u.color == WHITE
        DFS-VISIT(G, u)
```

```
DFS-VISIT(G, u)
time = time + 1              // white vertex u has just been discovered
u.d = time
u.color = GRAY
for each v ∈ G.Adj[u]        // explore edge (u, v)
    if v.color == WHITE
        v.π = u
        DFS-VISIT(G, v)
u.color = BLACK              // blacken u; it is finished
time = time + 1
u.f = time
```

Dybde-Først-Søgning (DFS)

Eksempel:



Egenskaber

Køretid: $O(n+m)$.

Beviset er det samme som under $\textsc{GenericGraphTraversalGlobal}$, da valgene (*) i DFS tager $O(1)$ tid.

Egenskaber

Køretid: $O(n + m)$.

Beviset er det samme som under $\textsc{GenericGraphTraversalGlobal}$, da valgene (*) i DFS tager $O(1)$ tid.

Observér:

- Discovery (hvid $\rightarrow$ grå) af v = sæt $v.d$ = kald af $\textsc{DFS-Visit}$ på v = PUSH af v på stakken.
- Finish (grå $\rightarrow$ sort) af v = sæt $v.f$ = retur fra kald af $\textsc{DFS-Visit}$ på v = POP af v fra stakken.

Egenskaber

Køretid: $O(n+m)$.

Beviset er det samme som under $\textsc{GenericGraphTraversalGlobal}$, da valgene (*) i DFS tager $O(1)$ tid.

Observér:

- Discovery (hvid $\rightarrow$ grå) af v = sæt $v.d$ = kald af $\textsc{DFS-Visit}$ på v = PUSH af v på stakken.
- Finish (grå $\rightarrow$ sort) af v = sæt $v.f$ = retur fra kald af $\textsc{DFS-Visit}$ på v = POP af v fra stakken.

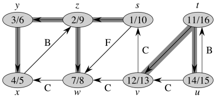
Værdien $v.\pi$ sættes ved kald af $\textsc{DFS-Visit}$ på v. Af dette, samt ovenstående, følger at:

- Kanterne $(v.\pi, v)$ udgør præcis rekursionstræerne for $\textsc{DFS-Visit}$ (ét træ for hvert kald fra $\textsc{DFS}$).
- Intervallet $[v.d, v.f]$ er den periode v er på stakken.
- Knuden v er grå hvis og kun hvis den er på stakken.

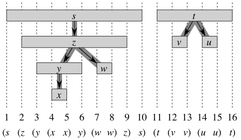
Egenskaber

Af måden en stak virker: Hvis to knuder u og v på et tidpunkt er på stakken samtidig, og v er øverst, må v poppes før u kan poppes.

Intervallet $[v.d, v.f]$ er den periode v er på stakken. Det følger derfor, at for alle par af knuder u og v må intervallerne $[u.d, u.f]$ og $[v.d, v.f]$ *enten* være disjunkte (u og v var aldrig på stakken samtidig) *eller* det ene interval må være helt indeholdt i den andet (u og v var på stakken samtidig, knuden med det største interval kom på først).



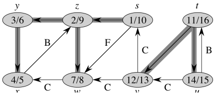
Discovery- og finish-tider er derfor nestede som parenteser er det.

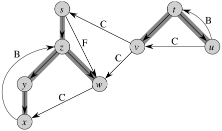


Egenskaber

Når en kant (u, v) undersøges fra u haves flg. tilfælde:

1. *tree-kanter:* v hvid.
2. *back-kanter:* v er grå (er på stak).
3. *forward-kanter:* v er sort (den er ikke længere på stak, men har været det sammen med u).
4. *cross-kanter:* v er sort (den er ikke længere på stak, og har ikke været det sammen med u).





Egenskaber

I lidt større detalje:

Når en kant (u, v) undersøges fra u haves flg. tilfælde:

1. *tree-kanter:* v hvid. Her er $u.d < v.d = \text{nu} < v.f < u.f$.
2. *back-kanter:* v er grå (er på stak – det må være under u, som er toppen af stakken (evt. $u = v$ hvis self-loop)). Her er $v.d \leq u.d < \text{nu} < u.f \leq v.f$.
3. *forward-kanter:* v er sort (den er ikke længere på stak, men har været det sammen med u). Her er $u.d < v.d < v.f < \text{nu} < u.f$.
4. *cross-kanter:* v er sort (den er ikke længere på stak, og har ikke været det sammen med u). Her er $v.d < v.f < u.d < \text{nu} < u.f$.

Bemærk at disse cases kan genkendes under DFS via hvid/grå/sort-farvningen og d-værdierne i knuder.

Egenskaber

For *uorienterede grafer* er der kun tree-kanter og back-kanter (såfremt en kant kategoriseres første gang den undersøges fra én af dens ender).

Dette følger af at u allerede må være blevet undersøgt fra v hvis v er sort (hele nabolisten er gennemløbet) og kanten (v, u) må derfor allerede være kategoriseret. Derved kan 3 og 4 ikke opstå.

1. *tree-kanter:* v hvid.
2. *back-kanter:* v er grå (er på stak).
3. *forward-kanter:* v er sort (den er ikke længere på stak, men har været det sammen med u).
4. *cross-kanter:* v er sort (den er ikke længere på stak, og har ikke været det sammen med u).

Hvid-sti lemma

Hvid-sti lemma:

Hvis findes en sti af hvide knuder (inkl. w) fra u til w til tid $u.d$, da gælder $u.d < w.d < w.f < u.f$.

Hvid-sti lemma

Hvid-sti lemma:

> Hvis findes en sti af hvide knuder (inkl. w) fra u til w til tid $u.d$, da gælder $u.d < w.d < w.f < u.f$.

Bevis (lemma):

Da stien er hvid til tid $u.d$, gælder $u.d \leq v.d$ for alle knuder v på stien. Af parentesstrukturen for d- og f-tider gælder så enten 1) $u.d \leq v.d < v.f \leq u.f$ eller 2) $u.d < u.f < v.d < v.f$.

Antag, at 2) forekommer og lad y være den første sådanne knude på stien. Da har y en forgænger x som opfylder 1) [evt. er x lig u, som jo opfylder 1)]. Men pga. kanten (x, y) må y opdages inden tid $x.f$, hvilket er i modstrid med at y opfylder 2). $\square$

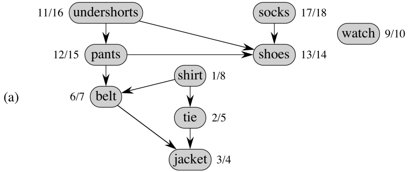
DAGs og topologisk sortering

DAG = Directed Acyclic Graph. En orienteret graf uden kredse (cycles).

DAGs og topologisk sortering

DAG = Directed Acyclic Graph. En orienteret graf uden kredse (cycles).

Bruges ofte til at modellere afhængigheder. Eksempel:

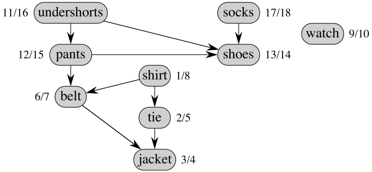


DAGs og topologisk sortering

DAG = Directed Acyclic Graph. En orienteret graf uden kredse (cycles).

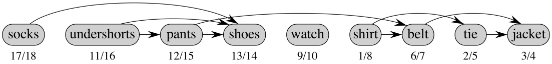
Bruges ofte til at modellere afhængigheder. Eksempel:

(a)



Topologisk sortering af en DAG: en lineær ordning af knuderne så alle kanter går fra venstre til højre.

(b)



DAGs og topologisk sortering

Lemma: En orienteret graf har en kreds (cycle) $\Leftrightarrow$ der findes back-edges under et DFS-gennemløb.

DAGs og topologisk sortering

Lemma: En orienteret graf har en kreds (cycle) $\Leftrightarrow$ der findes back-edges under et DFS-gennemløb.

Bevis:

$\Rightarrow$: DFS (med GLOBAL ydre loop) opdager alle knuder. Se på første knude v i kredsen som bliver grå. Dvs. at til tid $v.d$ er alle andre knuder hvide.

Af hvid-sti lemmaet fås så $v.d < u.d < u.f < v.f$ for den sidste knude u i kredsen (som peger på v), hvorved kanten (u, v) erklæres en backedge (v er grå, når denne kant undersøges).

DAGs og topologisk sortering

Lemma: En orienteret graf har en kreds (cycle) $\Leftrightarrow$ der findes back-edges under et DFS-gennemløb.

Bevis:

$\Rightarrow$: DFS (med GLOBAL ydre loop) opdager alle knuder. Se på første knude v i kredsen som bliver grå. Dvs. at til tid $v.d$ er alle andre knuder hvide.

Af hvid-sti lemmaet fås så $v.d < u.d < u.f < v.f$ for den sidste knude u i kredsen (som peger på v), hvorved kanten (u, v) erklæres en backedge (v er grå, når denne kant undersøges).

$\Leftarrow$: Når en back-edge findes: Der er en kreds af trækanter (mellem knuderne som lige nu er på stakken) og én back-kant.

DAGs og topologisk sortering

Lemma: For en kant (u, v) gælder $u.f \leq v.f \Leftrightarrow$ kanten er en back-edge.

DAGs og topologisk sortering

Lemma: For en kant (u, v) gælder $u.f \leq v.f \Leftrightarrow$ kanten er en back-edge.

Bevis: Check de fire cases for kanter (tree, back, forward, cross) og deres ordning af $u.f$ og $v.f$, se tidligere slide.

DAGs og topologisk sortering

Lemma: For en kant (u, v) gælder $u.f \leq v.f \Leftrightarrow$ kanten er en back-edge.

Bevis: Check de fire cases for kanter (tree, back, forward, cross) og deres ordning af $u.f$ og $v.f$, se tidligere slide.

Korollar til to foregående lemmaer: Graf er en DAG $\Leftrightarrow$ DFS finder ingen back-edges $\Leftrightarrow$ ordning af knuder efter faldende finish-tider giver en topologisk sortering.

DAGs og topologisk sortering

Lemma: For en kant (u, v) gælder $u.f \leq v.f \Leftrightarrow$ kanten er en back-edge.

Bevis: Check de fire cases for kanter (tree, back, forward, cross) og deres ordning af $u.f$ og $v.f$, se tidligere slide.

Korollar til to foregående lemmaer: Graf er en DAG $\Leftrightarrow$ DFS finder ingen back-edges $\Leftrightarrow$ ordning af knuder efter faldende finish-tider giver en topologisk sortering.

Så følgende algoritme finder en topologisk sortering i en DAG:

```
TOPOLOGICAL-SORT(G)
call DFS(G) to compute finishing times v.f for each vertex v
as each vertex is finished, insert it onto the front of a linked list
return the linked list of vertices
```

DAGs og topologisk sortering

Lemma: For en kant (u, v) gælder $u.f \leq v.f \Leftrightarrow$ kanten er en back-edge.

Bevis: Check de fire cases for kanter (tree, back, forward, cross) og deres ordning af $u.f$ og $v.f$, se tidligere slide.

Korollar til to foregående lemmaer: Graf er en DAG $\Leftrightarrow$ DFS finder ingen back-edges $\Leftrightarrow$ ordning af knuder efter faldende finish-tider giver en topologisk sortering.

Så følgende algoritme finder en topologisk sortering i en DAG:

```
TOPOLOGICAL-SORT(G)
    call DFS(G) to compute finishing times v.f for each vertex v
    as each vertex is finished, insert it onto the front of a linked list
    return the linked list of vertices
```

Tid: $O(n + m)$.