Föreläsning 9

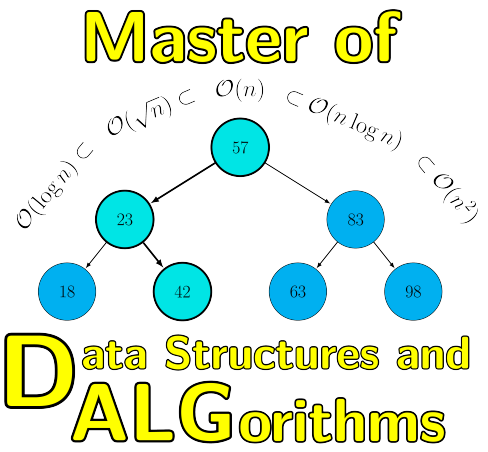
Grafer och grafsökning

TDDC91,TDDE22,725G97: DALG

Utskriftsversion av föreläsning i *Datastrukturer och algoritmer* 2 oktober 2018

Magnus Nielsen, IDA, Linköpings universitet

#WINNING

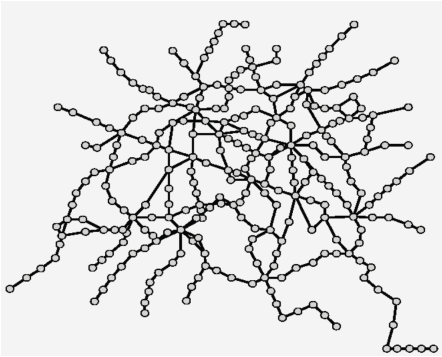


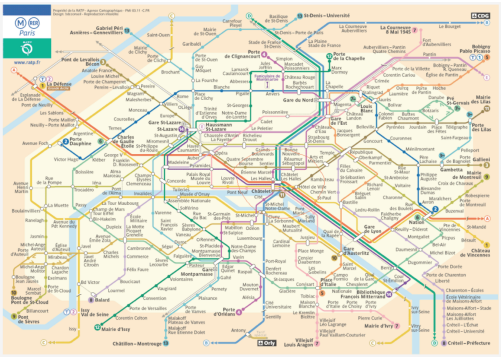
1 Grafer

1.1 Introduktion

Definition

- En graf är ett par (V, E), där
  - V är en mängd noder (eller hörn)
  - E är en mängd av par av noder kallade bågar (eller kanter)
  - Noder och bågar är positioner och kan lagra element
  - Eng. *vertices, edges*

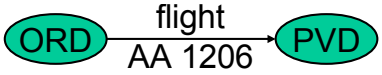


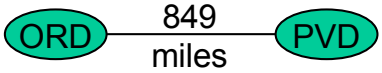


9.3

Bågtyper

- Riktad båge
  - ordnat par av noder (u, v)
  - u är startnoden, v är slutnoden
- Oriktad båge
  - oordnat par av noder $\{u, v\}$
- I en riktad graf är alla bågar riktade
- I en oriktad graf är alla bågar oriktade





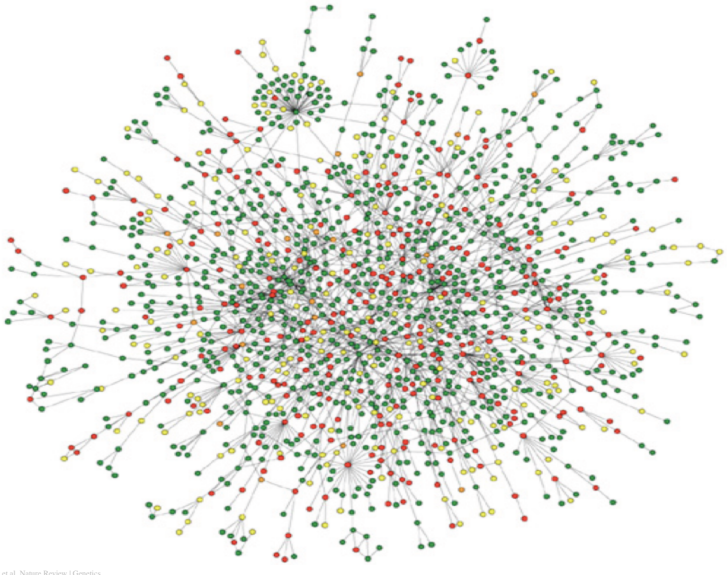
9.4

Varför skall man studera grafalgoritmer?

- Tusentals praktiska tillämpningar
- Hundratals kända grafalgoritmer
- Intressant abstraktion med stor tillämpbarhet
- Gren av datalogi och diskret matematik med många utmaningar

9.5

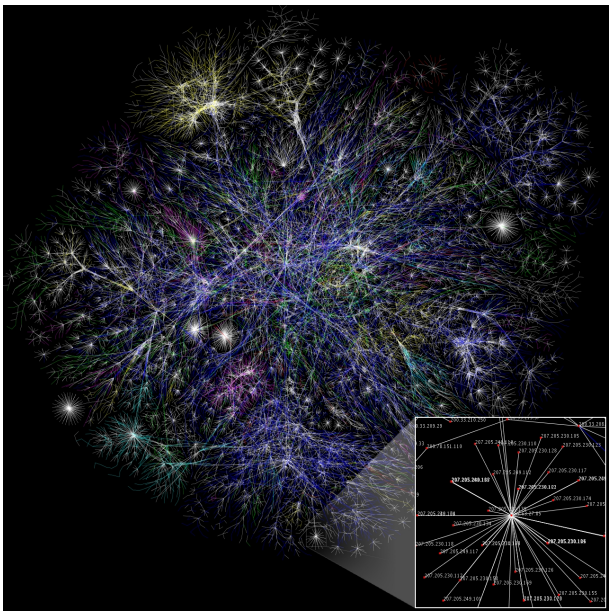
Protein till protein-interaktionsnätverk



Jeong et al, Nature Review | Genetics

9.6

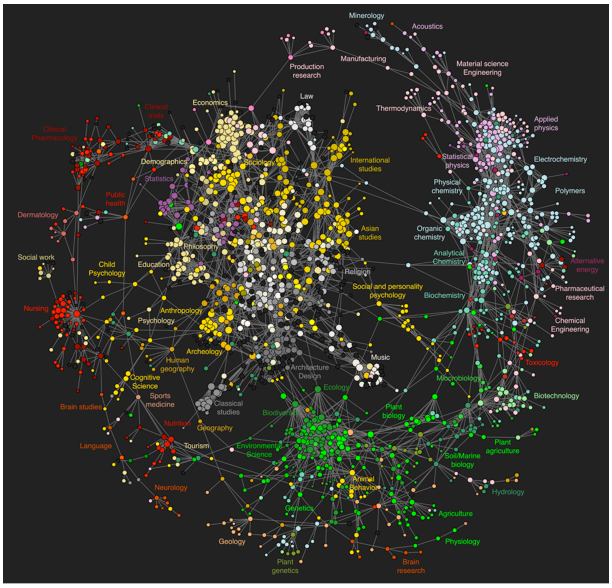
Internet kartlagt av Opte-projektet



Opte Project

9.7

Vetenskapliga klickströmmar



http://www.plosone.org/article/info:doi/10.1371/journal.pone.0004803

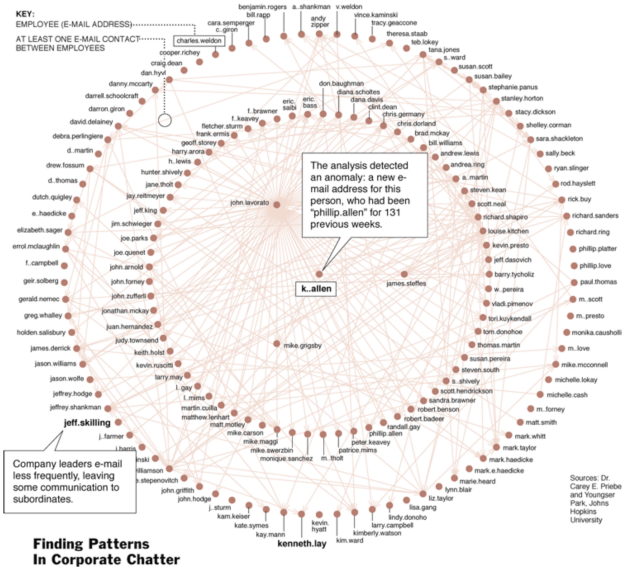
9.8

10 miljoner Facebook-vänner

"Visualizing Friendship" av Paul Butler

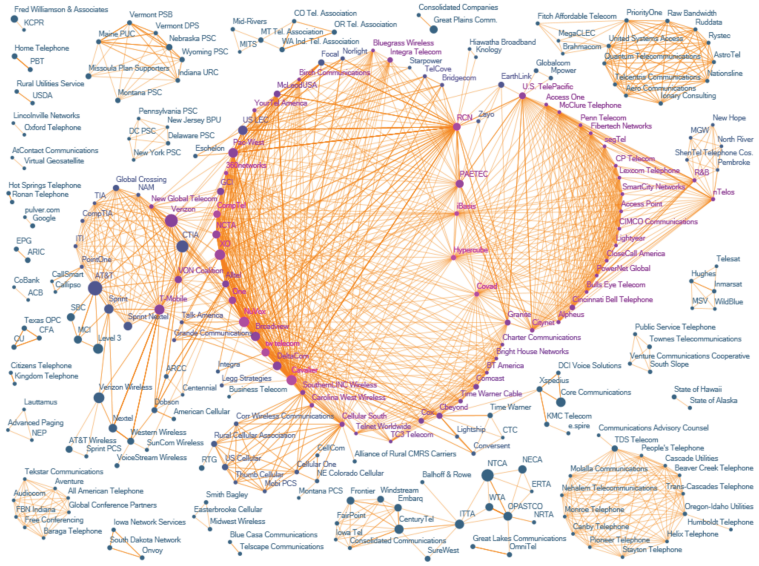
9.9

En veckas mejl inom Enron



9.10

Utvecklingen hos lobbyingsamarbeten inom FCC



9.11

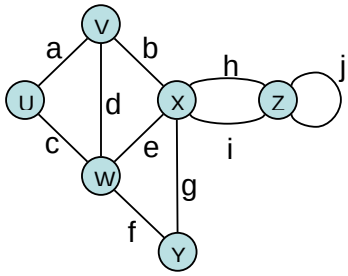
Tillämpningar

graf	nod	båge
kommunikation	telefon, dator	fiberoptisk kabel
krets	grind, register, processor	koppling
mekanisk	led	stag, bjälke, fjäder
finansiell	aktie, valuta	transaktion
transport	gatukorsning, flygplats	väg, flygrutt
internet	klass C-nät	förbindelse
brädspel	pjäsernas positioner	giltigt drag
socialt nätverk	person, aktör	vänskap, relation
neuralt nätverk	neuron	synaps
proteinnätverk	priotein	protein till protein-interaktion
kemisk sammansättning	molekyl	bindning

9.12

Terminologi

- En båge har ändpunkter (a har ändpunkterna U och V)
- Bågar som slutar i en nod n sägs vara incidenta (a, d och b är incidenta till V)
- Noder kan vara grannar (U och V är grannar)
- Noder har grad (X har grad 5)
- Parallella bågar (h och i är parallella bågar)
- Öglor (j är en ögla)

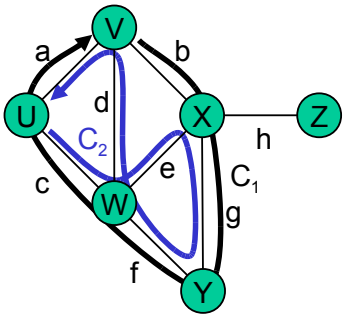


Eng. *endpoints, incident, adjacent, degree, parallel, loops*

9.13

Mer terminologi

- En cykel är en cirkulär sekvens av alternerande noder och bågar. Varje båge föregås och efterföljs av sina ändpunkter.
- En enkel cykel är en cykel sådan att alla dess noder och bågar är distinkta.
- $C_1 = (V, b, X, g, Y, f, W, c, U, a, V)$ är en enkel cykel.
- $C_2 = (U, c, W, e, X, g, Y, f, W, d, V, a, U)$ är *inte* en enkel cykel.



Eng. *cycle, simple cycle* 9.14

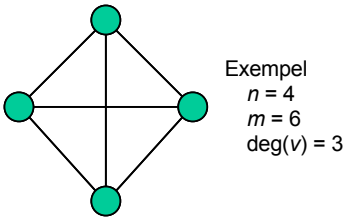
Egenskaper

Egenskap 1
$\sum_v \deg(v) = 2m$ Bevis: Varje båge räknas två gånger

Egenskap 2
I en oriktad graf utan öglor och parallella bågar gäller $m \leq n(n-1)/2$ Bevis: Varje nod har max grad $(n-1)$

Notation

- n antalet noder
- m antalet bågar
- $\deg(v)$ är nod v:s grad



9.15

Några algoritmiska grafproblem

- Stig. Finns det en stig mellan s och t?
- Kortaste väg. Vilken är den kortaste stigen mellan s och t?

- Cykel. Finns det en cykel i grafen?
- Eulertur. Finns det en cykel som använder varje båge exakt en gång?
- Hamiltoncykel. Finns det en cykel som använder varje nod exakt en gång?

- Konnektivitet. Finns det en förbindelse mellan alla noder?
- Bikonnektivitet. Finns det en nod som gör att grafen inte hänger samman om man tar bort den?

- Planaritet. Går det att rita grafen utan att några bågar korsar varandra?
- Grafisomorfi. Är två grafer identiska bortsett från namnen på noderna?

Utmaning. Vilka av problemen ovan är enkla? Svåra? Omöjliga att lösa effektivt? 9.16

1.2 ADT graf

De viktigaste metoderna för oriktade grafer

- Noder och bågar
  - är positioner
  - lagrar element
- Åtkomstmetoder
  - endVertices(e): en array med e:s två ändpunkter
  - opposite(v,e): noden motsatt v längs e
  - areAdjacent(v,w): **true** omm v och w är grannar
  - replace(v,x): ersätt elementet i nod v med x
  - replace(e,x): ersätt elementet i båge e med x

9.17

De viktigaste metoderna för oriktade grafer

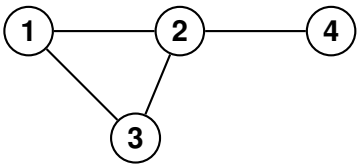
- Uppdateringsmetoder
  - insertVertex(o): sätt in en nod som lagrar elementet o
  - insertEdge(v, w, o): sätt in en kant (v, w) som lagrar elementet o
  - removeVertex(v): ta bort nod v (och dess incidenta bågar)
  - removeEdge(e): ta bort båge e
- Iteratormetoder
  - incidentEdges(v): bågarna incidenta till v
  - vertices(): alla noder i grafen
  - edges(): alla bågar i grafen

9.18

1.3 Datastrukturer

Båglista (Edge list)

- Båglistan representerar alla kanter i grafen
- Nodernas ordning är inte relevant i *oriktade* grafer
- *Riktade* grafer: nodernas ordning dikterar kantens riktning
- Kan innehålla ytterligare information

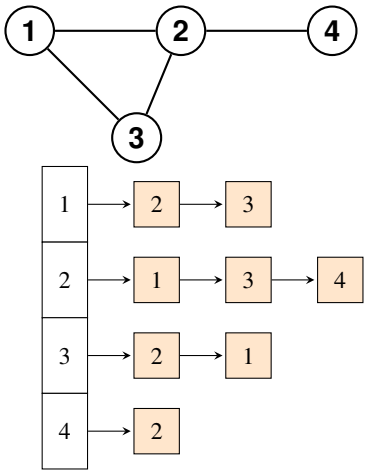


Exempel på arrayrepresentation:
edgeList=$[[1,2],[1,3],[2,3],[2,4]]$

9.19

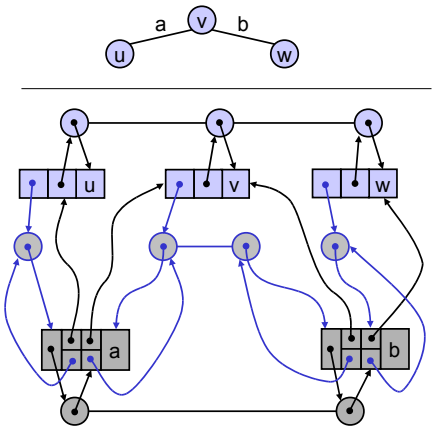
Grannlista (Adjacency list)

- Lägger till extra struktur till båglistan
- Utöka nodobjekten med heltalsnycklar som mappas mot index i listan
- Varje nod har en lista med grann-noder



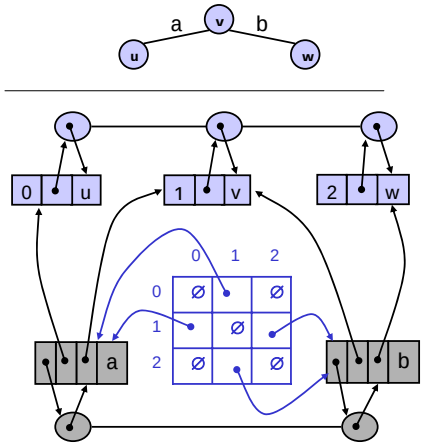
9.20

Grannlista (Adjacency list)



Grannmatris (Adjacency matrix)

- Lägger till extra struktur till båglistan
- Utöka nodobjekten med heltalsnycklar som mappas mot index i matrisen (ett heltal räcker)
- Tvådimensionell grannarray
  - Referens till bågobjekt för noder som är grannar
  - **null** för noder som inte är grannar



Asymptotisk prestanda

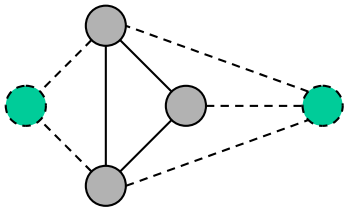
n noder, ***m*** bågar inga parallella kanter inga öglor	Båglista	Grannlista	Grann-matris
minne	$O(n + m)$	$O(n + m)$	$O(n^2)$
incidentEdges(***v***)	$O(m)$	$O(\deg(v))$	$O(n)$
areAdjacent (***v, w***)	$O(m)$	$O(\min(\deg(v),\deg(w))$	$O(1)$
insertVertex(***o***)	$O(1)$	$O(1)$	$O(n^2)$
insertEdge(***v, w, o***)	$O(1)$	$O(1)$	$O(1)$
removeVertex(***v***)	$O(m)$	$O(\deg(v))$	$O(n^2)$
removeEdge(***e***)	$O(1)$	$O(1)$	$O(1)$

2 Sökning i oriktade grafer

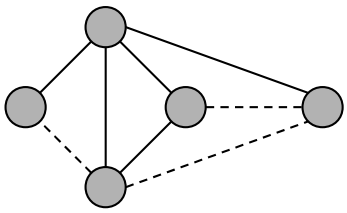
2.1 DFS

Delgrafer

- En delgraf S av en graf G är en graf sådan att
  - Noderna i S är en delmängd av noderna i G
  - Bågarna i S är en delmängd av bågarna i G
- En spännande delgraf av G är en delgraf som innehåller alla noder i G



Delgraf

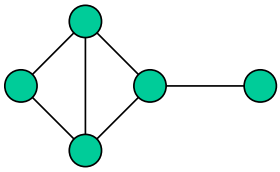


Spännande delgraf

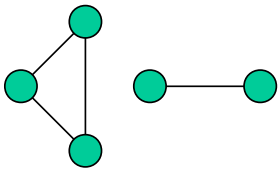
Eng. *subgraph, spanning subgraph*

Konnektivitet

- En graf är sammanhängande om det finns en stig mellan varje par av noder
- En sammanhängande komponent i en graf G är en maximal sammanhängande delgraf av G



Sammanhängande graf



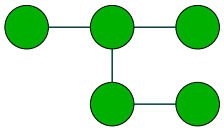
Ej sammanhängande graf med två sammanhängande komponenter

Eng. *connected, connected component*

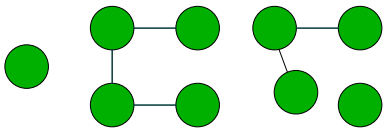
9.25

Träd och skogar

- Ett (fritt) träd är en oriktad graf T sådan att
  - T är sammanhängande
  - T inte har några cykler
  - Den här definitionen av träd skiljer sig från den för rotade träd
- En skog är en oriktad graf utan cykler
- De sammanhängande komponenterna i en skog är träd



Träd



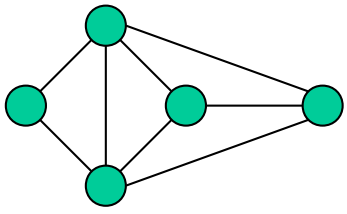
Skog

Eng. *tree, forest*

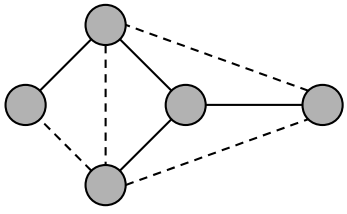
9.26

Spännande träd och skogar

- Ett spännande träd till en sammanhängande graf är en spännande delgraf som är ett träd
- Ett spännande träd är inte unikt om inte ursprungsgrafen är ett träd
- En spännande skog till en graf är en spännande delgraf som är en skog



Graf



Spännande träd

Eng. *spanning tree, spanning forest*

9.27

Djupetförstsökning

- Djupetförstsökning (DFS) är en allmän teknik för att traversera en graf
- DFS i en graf G
  - Besöker alla noder och bågar i G
  - Avgör om G är sammanhängande
  - Beräknar de sammanhängande komponenterna i G
  - Beräknar en spännande skog till G
- DFS på en graf med n noder och m bågar tar $O(n+m)$ tid
- DFS kan utökas för att lösa andra grafproblem
  - Hitta och beskriv en stig mellan två givna noder i en graf
  - Hitta en cykel i en graf

9.28

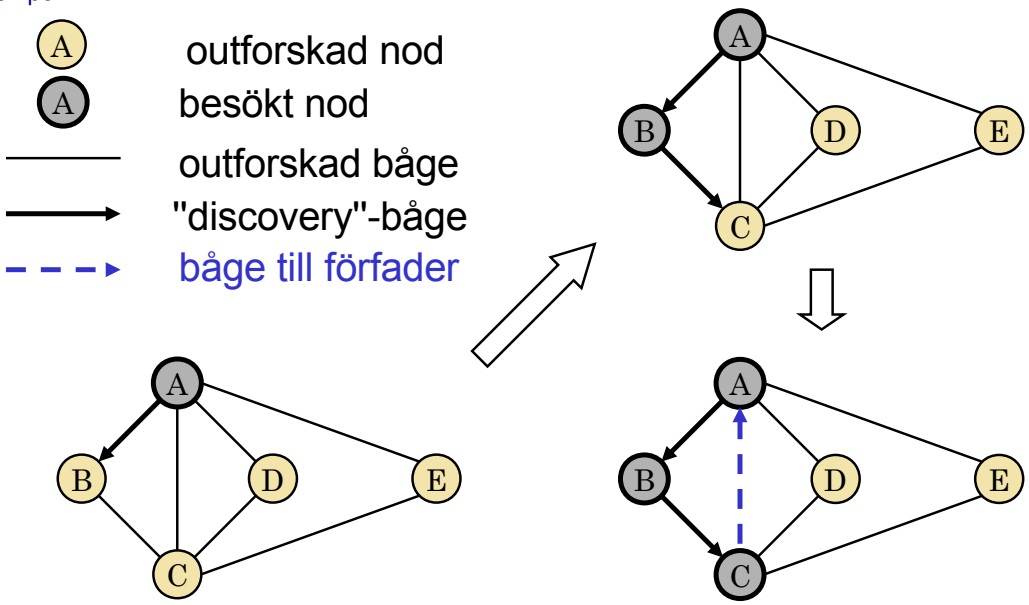
Algoritm för DFS

procedure DFS(G)
  for all $u \in G$.VERTICES() **do**
    SETLABEL(u, $UNEXPLORED$)
  for all $e \in G$.EDGES() **do**
    SETLABEL(e, $UNEXPLORED$)
  for all $v \in G$.VERTICES() **do**
    if GETLABEL(v) = $UNEXPLORED$ **then**
      DFS(G, v)

procedure DFS(G, v)
  SETLABEL(v, $VISITED$)
  for all $e \in G$.INCIDENTEDGES(v) **do**
    if GETLABEL(e) = $UNEXPLORED$ **then**
      $w \leftarrow$OPPOSITE(v, e)
      if GETLABEL(w) = $UNEXPLORED$ **then**
        SETLABEL(e, $DISCOVERY$)
        DFS(G, w)
      else
        SETLABEL(e, $BACK$)

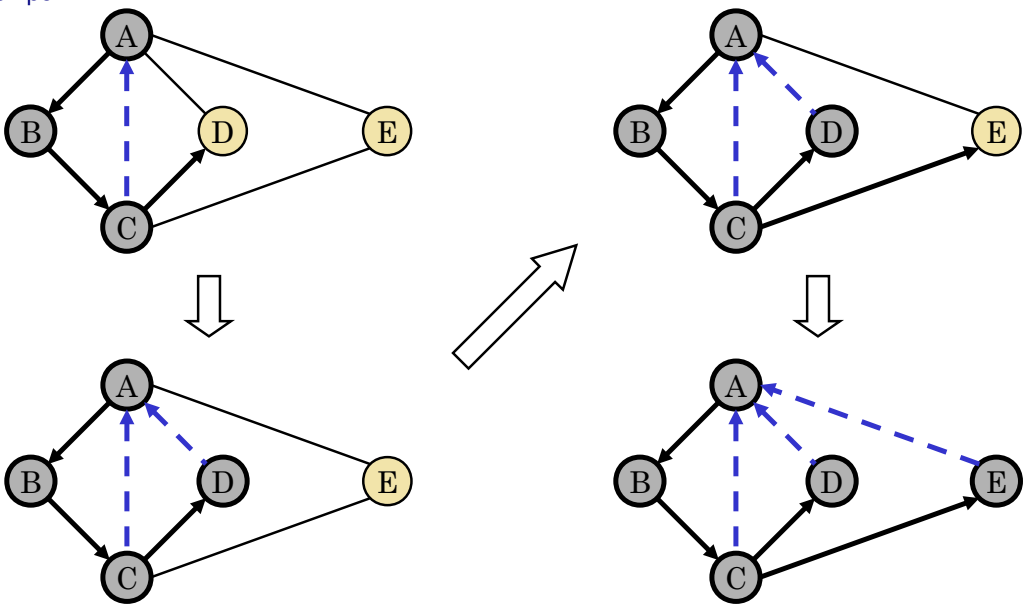
9.29

Exempel



9.30

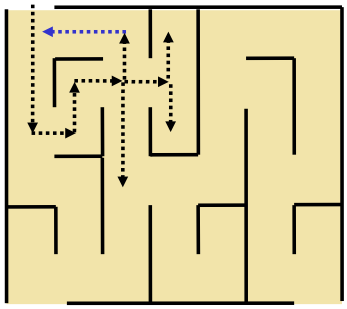
Exempel



9.31

DFS och labyrintutforskning

- Algoritmen för DFS liknar en klassisk strategi för att utforska labyrinter
  - Vi märker varje korsning, hörn och återvändsgränd (nod) vi besöker
  - Vi märker varje korridor (båge) vi går genom
  - Vi håller reda på vägen tillbaka till ingången (startnoden) m.h.a. ett snöre (rekursionsstacken)

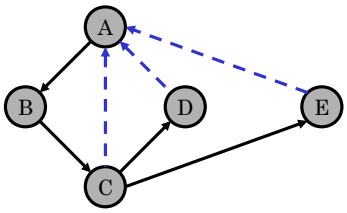


9.32

Egenskaper

Egenskap 1
DFS(G,v) besöker alla noder och bågar i den sammanhängande delen av G som v ingår i

Egenskap 2
”discovery”-bågarna DFS(G,v) märker upp utgör ett spännande träd till den sammanhängande komponenten av G som v ingår i



9.33

Analys av DFS

- Märka/hämta märkning av nod/båge tar $O(1)$ tid
- Varje nod märks två ggr
  - en gång som *UNEXPLORED*
  - en gång som *VISITED*

- Varje båge märks två ggr
  - en gång som *UNEXPLORED*
  - en gång som *DISCOVERY* eller *BACK*
- Metoden incidentEdges anropas en gång för varje nod
- DFS körs i tid $O(n+m)$ givet att grafen är representerad med en grannlista
  - kom ihåg att $\sum_v deg(v) = 2m$

9.34

Hitta stigar

- Vi kan specialisera DFS-algoritmen till att hitta en stig mellan två givna noder v och z
- Vi anropar DFS(G,v) med v som startnod
- Vi använder en stack S för att hålla reda på vägen från startnoden till aktuell nod
- Så snart vi stöter på målnoden z returnerar vi innehållet på stacken som den sökta stigen

```
procedure PATHDFS(G, v, z)
    SETLABEL(v, VISITED)
    S.PUSH(v)
    if v = z then
        skriv ut elementen i S
        return
    for all e ∈ G.INCIDENTEDGES(v) do
        if GETLABEL(e) = UNEXPLORED then
            w ← OPPOSITE(v, e)
            if GETLABEL(w) = UNEXPLORED then
                SETLABEL(e, DISCOVERY)
                S.PUSH(e)
                PATHDFS(G, w, z)
                S.POP() // e
            else
                SETLABEL(e, BACK)
    S.POP() // v
```

9.35

Hitta cykler

- Vi kan specialisera DFS-algoritmen till att hitta en enkel cykel
- Vi använder en stack S för att hålla reda på vägen från startnoden till aktuell nod
- Så snart vi stöter på en kant (v,w) som leder till en förfader returnerar vi cykeln som innehållet på stacken från toppen till noden w

```
procedure CYCLEDFS(G, v, z)
    SETLABEL(v, VISITED)
    S.PUSH(v)
    for all e ∈ G.INCIDENTEDGES(v) do
        if GETLABEL(e) = UNEXPLORED then
            w ← OPPOSITE(v, e)
            S.PUSH(e)
            if GETLABEL(w) = UNEXPLORED then
                SETLABEL(e, DISCOVERY)
                CYCLEDFS(G, w)
                S.POP() // e
            else // hittat cykel
                repeat
                    o ← S.POP()
                    skriv ut o
                until o = w
                return
    S.POP() // v
```

9.36

2.2 BFS

Breddenförstsökning

- Breddenförstsökning (BFS) är en allmän teknik för att traversera en graf
- BFS i en graf G

  - Besöker alla noder och bågar i G
  - Avgör om G är sammanhängande
  - Beräknar de sammanhängande komponenterna i G
  - Beräknar en spännande skog till G
- BFS på en graf med n noder och m bågar tar $O(n+m)$ tid
- BFS kan utökas för att lösa andra grafproblem
  - Hitta och beskriv en kortaste stig mellan två givna noder i en graf
  - Hitta en enkel cykel i en graf, om det finns någon

9.37

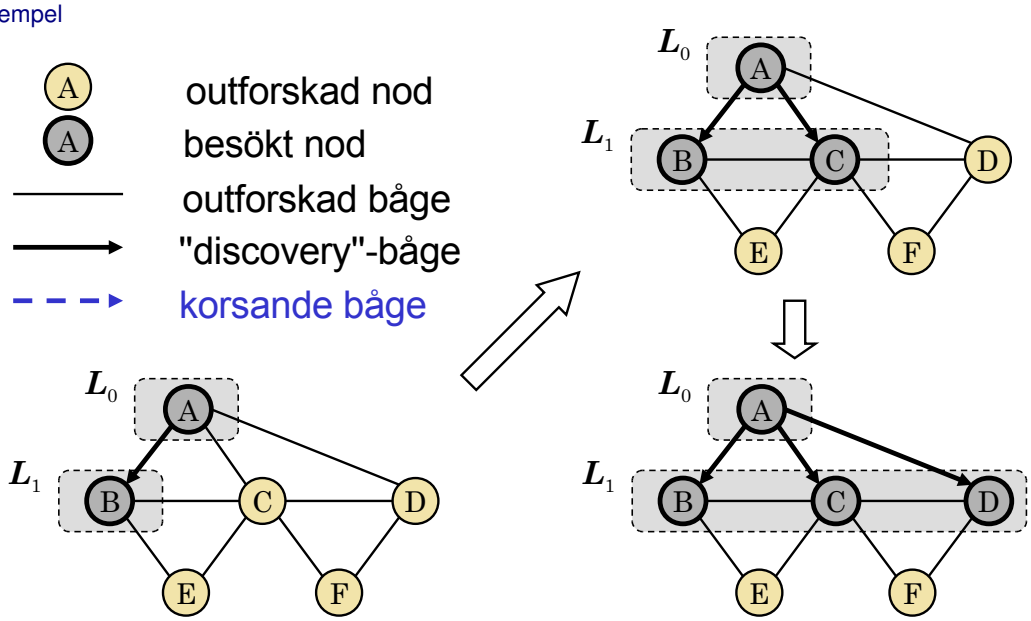
Algoritm för BFS

procedure BFS(G)
  märk alla noder/bågar med *UNEXPLORED* som i DFS
  for all $v \in G$.VERTICES() **do**
    if GETLABEL(v) = *UNEXPLORED* **then** BFS(G, v)

procedure BFS(G, s)
  $L_0 \leftarrow$ ny tom sekvens; L_0.INSERTLAST(s); SETLABEL(s, *VISITED*); $i \leftarrow 0$
  while $\neg L_i$.ISEMPTY() **do**
    $L_{i+1} \leftarrow$ ny tom sekvens
    for all $v \in L_i$.ELEMENTS() **do**
      for all $e \in G$.INCIDENTEDGES(v) **do**
        if GETLABEL(e) = *UNEXPLORED* **then**
          $w \leftarrow$OPPOSITE(v, e)
          if GETLABEL(w) = *UNEXPLORED* **then**
            SETLABEL(e, *DISCOVERY*)
            SETLABEL(w, *VISITED*)
            L_{i+1}.INSERTLAST(w)
          else
            SETLABEL(e, *CROSS*)
    $i \leftarrow i+1$

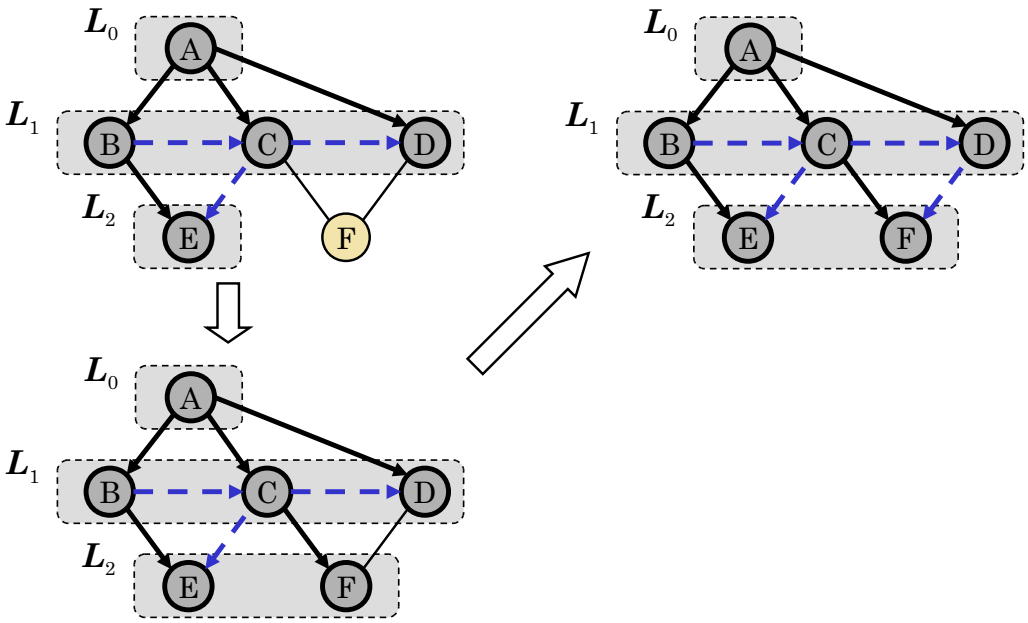
9.38

Exempel



9.39

Exempel



9.40

Egenskaper

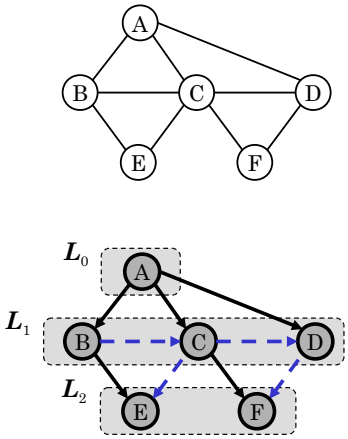
Låt G_s beteckna den samman-hängande delen av G som s ingår i

Egenskap 1
BFS(G,s) besöker alla noder och bågar i G_s

Egenskap 2
”discovery”-bågarna BFS(G,s) märker upp utgör ett spännande träd T_s till G_s

Egenskap 3
För varje nod v i L_i gäller

- Stigen i T_s från s till v har i bågar
- Varje stig från s till v i G_s har minst i bågar



9.41

Analys av BFS

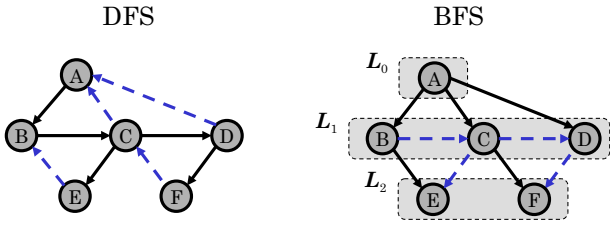
- Märka/hämta märkning av nod/båge tar $O(1)$ tid
- Varje nod märks två ggr
  - en gång som *UNEXPLORED*
  - en gång som *VISITED*
- Varje båge märks två ggr
  - en gång som *UNEXPLORED*
  - en gång som *DISCOVERY* eller *CROSS*
- Varje nod sätts in en gång i en sekvens L_i
- Metoden incidentEdges anropas en gång för varje nod
- BFS körs i tid $O(n+m)$ givet att grafen är representerad med en grannlista
  - kom ihåg att $\sum_v deg(v) = 2m$

9.42

2.3 DFS vs BFS

Tillämpningar

Tillämpningar	DFS	BFS
Spännande träd, sammanhängande komponenter, stigar, cykler	√	√
Kortaste stigar		√
2-sammanhängande komponenter	√	

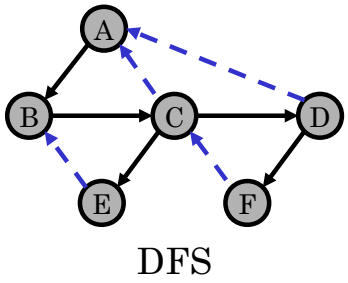


9.43

Kanter som leder till redan besökta noder

båge till förfader

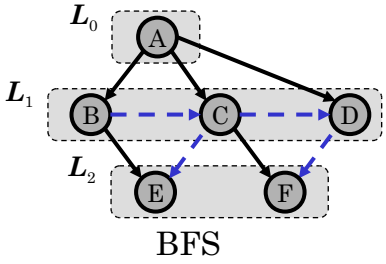
- *w* är en förfader till *v* i trädet av ”discovery”-bågar



DFS

korsande båge

- *w* finns i samma nivå som *v* eller i nästa nivå i trädet av ”discovery”-bågar

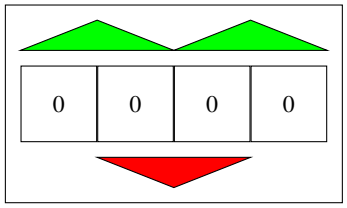


BFS

9.44

Problem

Du har hittat ett gammalt kodlås som du vill låsa upp. Du vet att koden är 3146, och just nu visar siffrorna på låset 0000. Låset har knappar för att öka eller minska de olika siffrorna, dock är många av knapparna trasiga. Hur skriver du in koden med så få knapptryck som möjligt? Låset ser ut som följer:



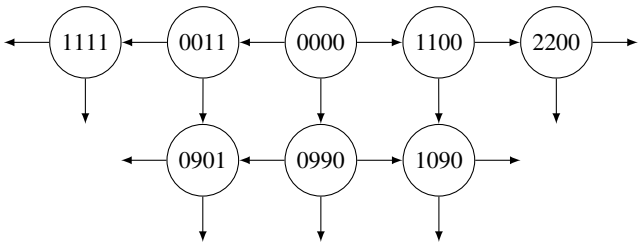
9.45

Lösningsidé

Representera problemet som en riktad graf:

- En nod för varje kombination
- Varje båge motsvarar ett knapptryck

- Oviktad graf ⇒ BFS
- Varje väg motsvarar en lösning!



9.46