Institut for Matematik og Datalogi
Syddansk Universitet, Odense

10. maj 2020

DM507 — Algoritmer og Datastrukturer

Eksaminatorie-timer uge 19, Forår 2020

Instruktorerne for DM507

Indhold

1 Opgave 1 - Cormen et al. øvelse 22.3-2 (side 610)

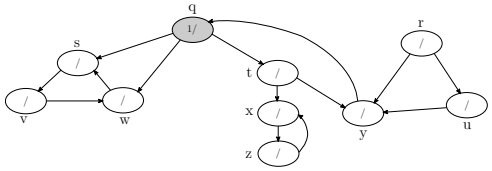
Vi følger DFS og DFS-VISIT algoritmerne som beskrevet på Rolfs slides [3]. Hver gang der vælges en knude bruger vi den alfabetiske rækkefølge (sker i linje 1 og 5 i DFS og linje 4 i DFS-VISIT). Den alfabetiske rækkefølge af knuderne er: q, r, s, t, u, v, w, x, y, z.

Vi klassificerer kanterne mens algoritmen kører ved at følge slide 24 [3]. Kort sagt, når en kant (u,v) undersøges fra u, så har vi:

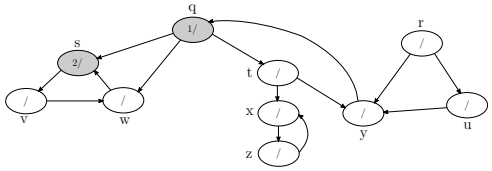
- tree-kant hvis v er hvid
- back-kant hvis v er grå
- forward-kant hvis v er sort og $u.d < v.d$
- cross-kant hvis v er sort og $u.d > v.d$

Klassifikationen af en kant er skrevet på kanten, hvor ingenting indikerer en tree-kant, B indikerer en back-kant, F indikerer en forward-kant og C indikerer en cross-kant.

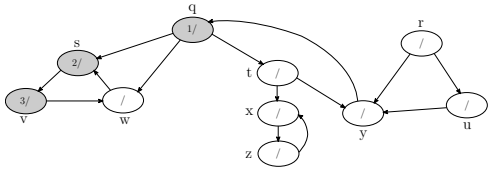
En kørsel af DFS er vist på figurerne på de næste to sider (man er nok nødt til at zoome ind for at se detaljer).



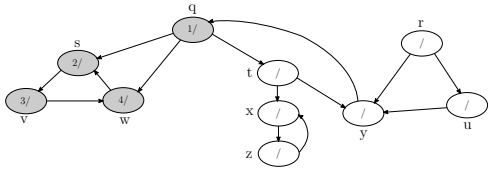
(a)



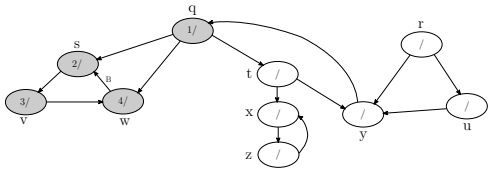
(b)



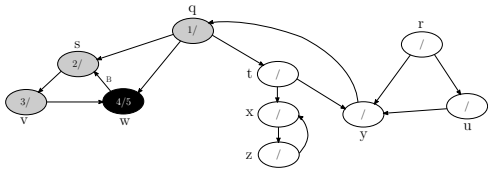
(c)



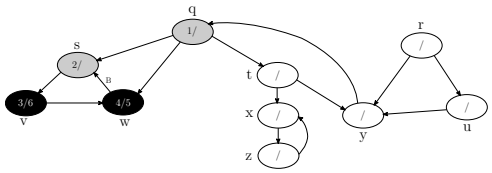
(d)



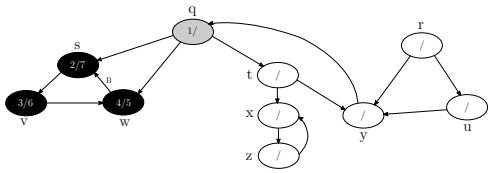
(e)



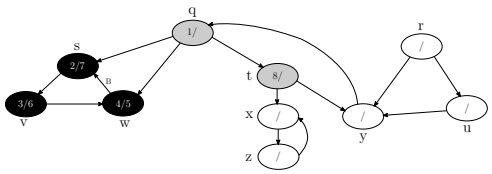
(f)



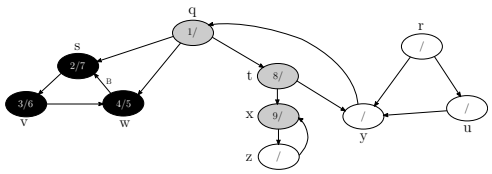
(g)



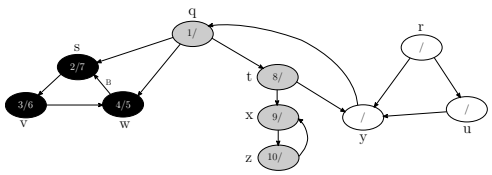
(h)



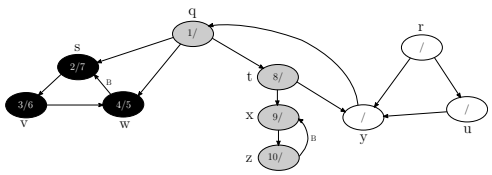
(i)



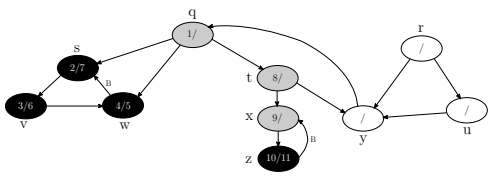
(j)



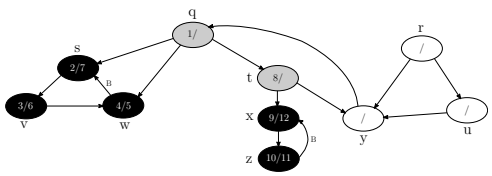
(k)



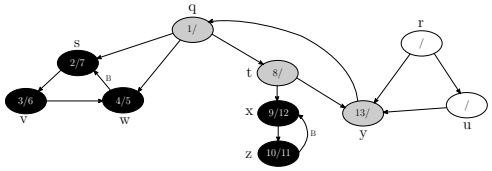
(l)



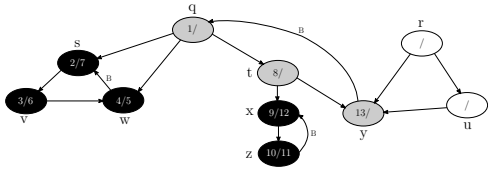
(m)



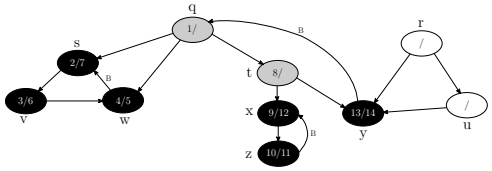
(n)



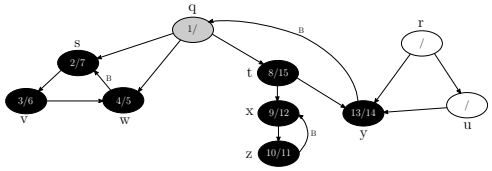
(o)



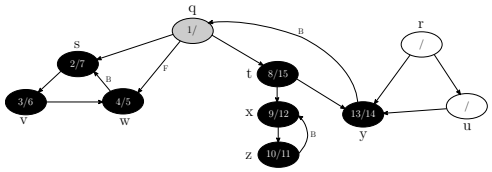
(p)



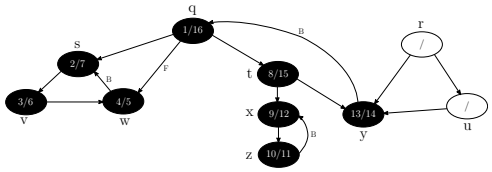
(q)



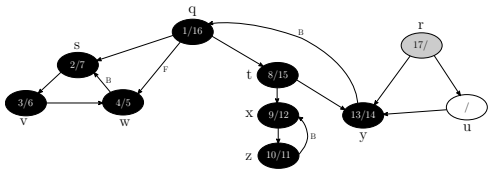
(r)



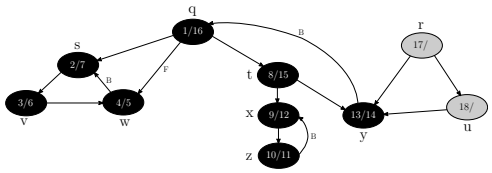
(s)



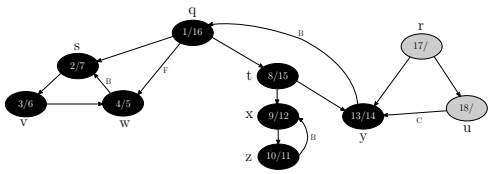
(t)



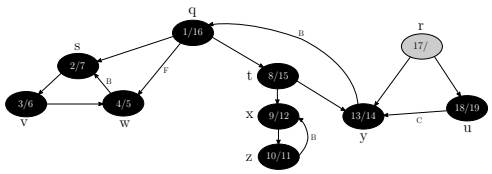
(u)



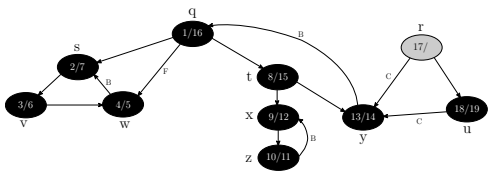
(v)



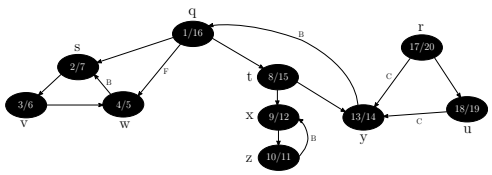
(w)



(x)



(y)

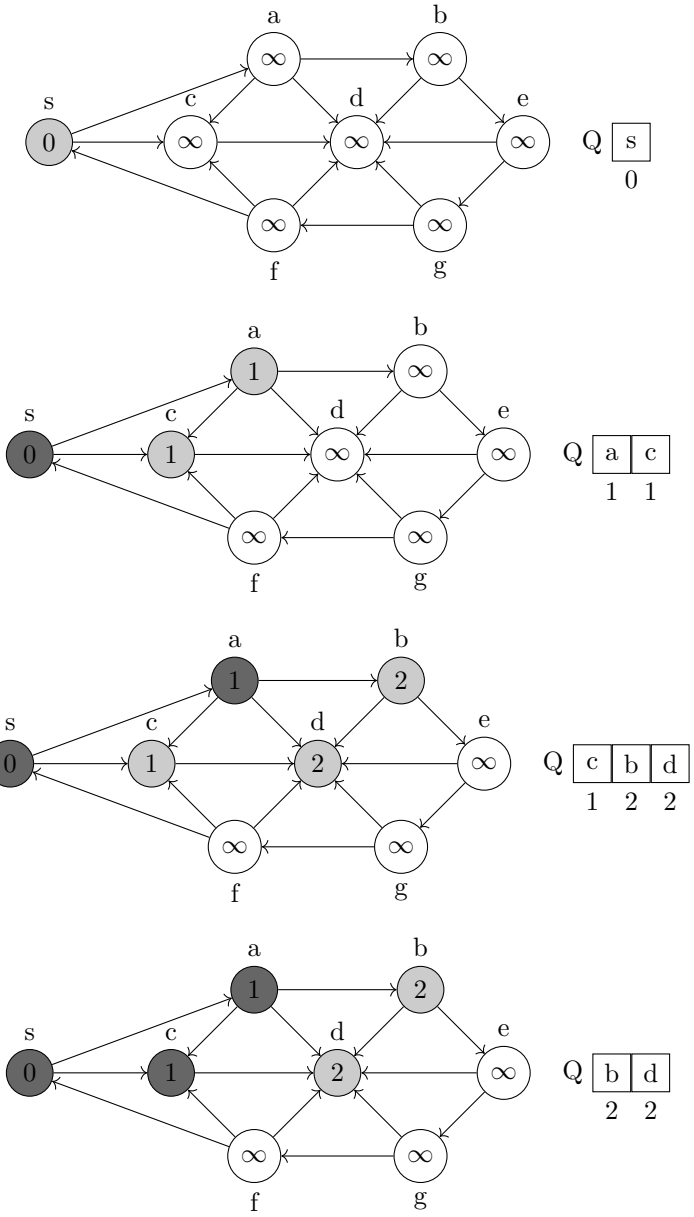


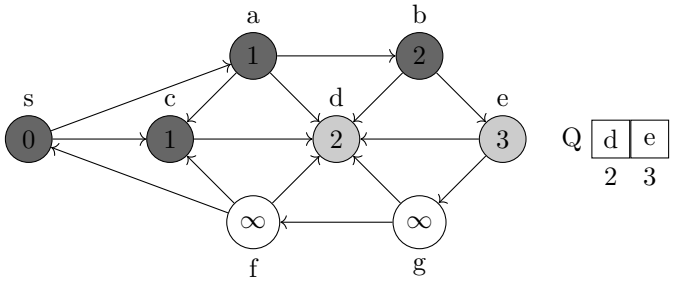
(z)

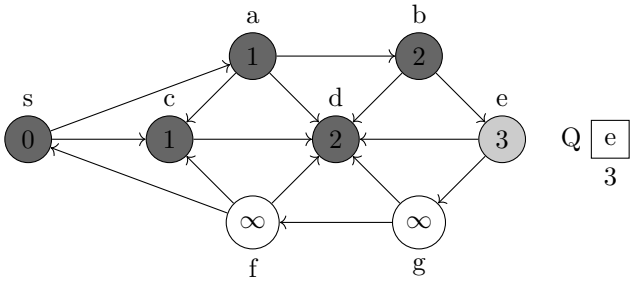
2 Opgave 2 - Eksamen juni 2010 opgave 2, spørgsmål a og b

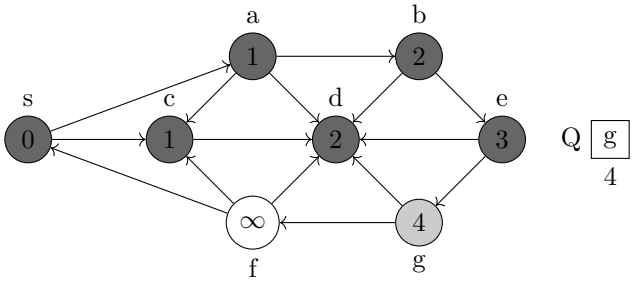
Spørgsmål a

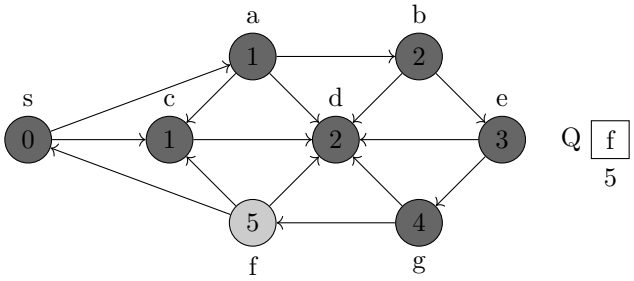
Vi følger pseudokoden som beskrevet på Rolfs slides[3, p.14]. Knuderne uden navn navngives, så der kan refereres til dem. Hver figur er vist inden en iteration (den sidste inden løkken stopper):

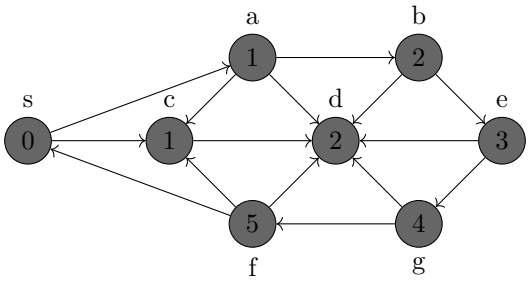



a
b
1
2
s
c
d
e
0
1
2
3
∞
∞
f
g
Q
d
e
2
3


a
b
1
2
s
c
d
e
0
1
2
3
∞
∞
f
g
Q
e
3

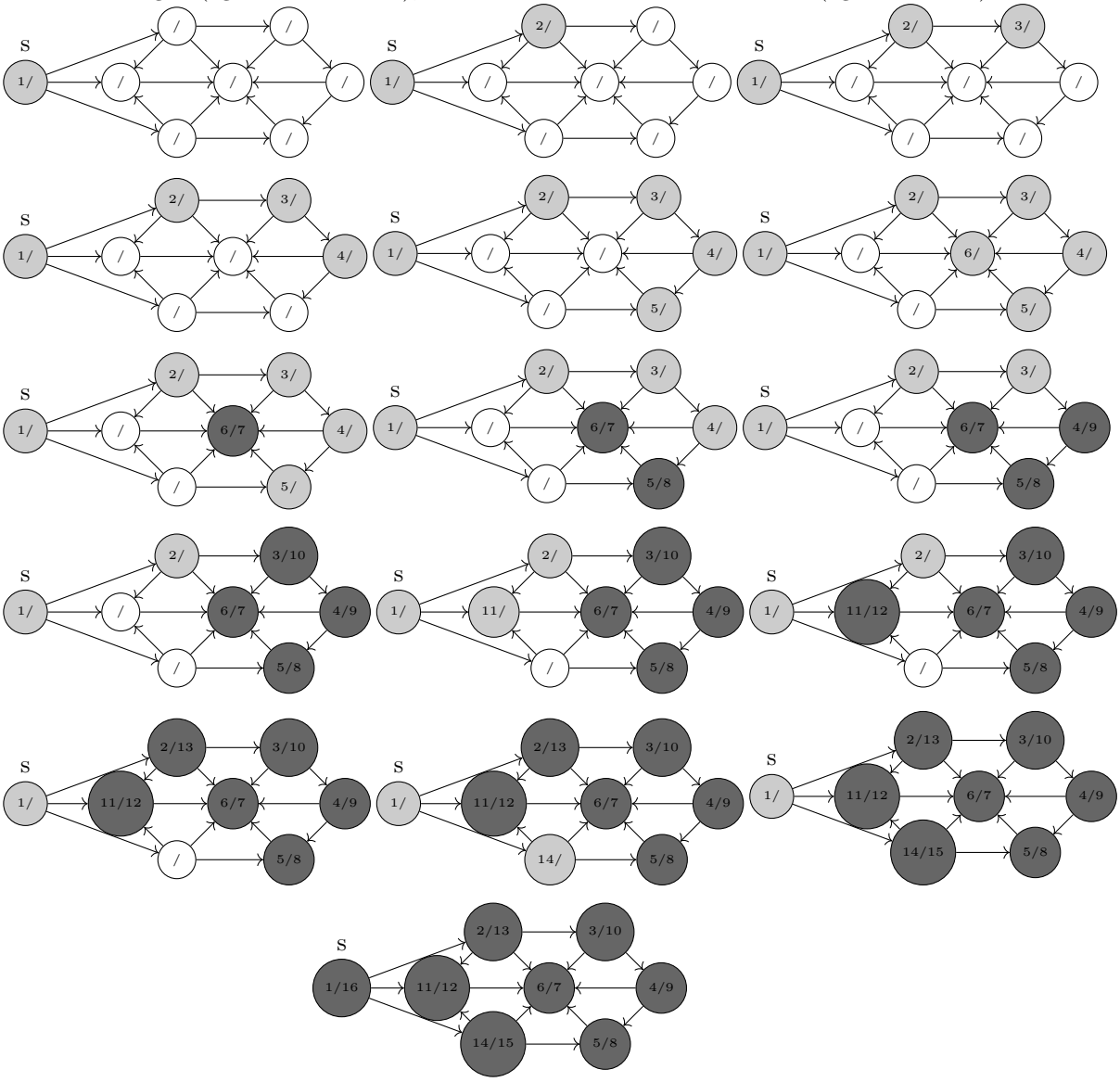

a
b
1
2
s
c
d
e
0
1
2
3
∞
4
f
g
Q
g
4


a
b
1
2
s
c
d
e
0
1
2
3
5
4
f
g
Q
f
5


a
b
1
2
s
c
d
e
0
1
2
3
5
4
f
g

Spørgsmål b

Vi følger pseudokoden som beskrevet på Rolfs slides[3, p.20]. Hver figur er vist efter en knude farves grå (og får en starttid), eller efter en knude får en sluttid (og farves sort):



3 Opgave 3 - Cormen et al. øvelse 22.3-4 (side 611)

I denne opgave skal vi argumentere for, at når DFS anvendes på en given graf, så er det nok at bruge en enkelt bit til at gemme farven på en knude i grafen. Hver bit for hver knude kan antage værdien 0 eller 1, alt efter om knuden er hvid (ubesøgt) eller ikke-hvid (besøgt), henholdsvis.

I Figur 3 ses pseudokoden for DFS og delproceduren DFS-Visit. Hvis vi kigger på linje 8 i DFS-Visit, så ser vi at denne linje kan fjernes, da vi faktisk kun behøver at kunne differentiere mellem hvide og ikke-hvide knuder, dvs. i dette tilfælde grå knuder. Efter vi har farvet en knude grå besøger vi den nemlig aldrig igen (vi er kun interesserede i at kigge på endnu ubesøgte knuder i grafen og checker derfor kun om en knude er hvid eller ej. Dette gøres i linje 6 i DFS og i linje 5 i DFS-Visit delproceduren. Med andre ord, efter en knude er blevet farvet grå og vi har løbet alle knudens naboer igennem, så er vi ikke længere interesserede i knuden. Det er derfor unødvendigt efterfølgende at farve knuden sort.

```
DFS(G)
1  for each vertex u ∈ G.V
2      u.color = WHITE
3      u.π = NIL
4  time = 0
5  for each vertex u ∈ G.V
6      if u.color == WHITE
7          DFS-VISIT(G, u)

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

Figur 3: Pseudokode for DFS taget fra Cormen et al. side 604 (3rd edition).

4 Opgave 4 - Cormen et al. øvelse 22.3-10 (side 612)

I denne opgave skal vi modificere pseudokoden for dybde-først-søgning så alle kanter i grafen bliver printet sammen med deres type. Først skal vi lave dette for orienterede grafer og hvis det kræver modifikationer for ikke-orienterede grafer skal vi også vise disse.

I denne opgave tager vi udgangspunkt i den originale pseudokode for `DFS` som kan findes i [1, side 604]. Definitionen af typer for kanter kan findes i [1, side 609] eller [3, slide 25].

```
1  Modified-DFS(G)
2      for each vertex u ∈ G.V
3          u.color = WHITE
4          u.π = NIL
5      time = 0
6      for each egde e ∈ G.E
7          e.type = NIL
8      for each vertex u ∈ G.V
9          if u.color == WHITE
10             Modified-DFS-Visit(G, u)
11
12 Modified-DFS-Visit(G, u)
13     time = time + 1
14     u.d = time
15     u.color = GRAY
16     for each v ∈ G.Adj(u)
17         if v.color == WHITE
18             v.π = u
19             e = G.E(u,v)
20             e.type = Tree
21             print("edge: " + e + " type: " + e.type)
22             Modified-DFS-Visit(G, v)
23         else if v.color == GRAY
24             e = G.E(u,v)
25             if e.type == NIL
26                 e.type = Back
27                 print("edge: " + e + " type: " + e.type)
28         else if v.color == BLACK
29             e = G.E(u,v)
30             if e.type == NIL
31                 if u.d < v.d
32                     e.type = Forward
33                 else       //u.d > v.d
34                     e.type = Cross
35                 print("edge: " + e + " type: " + e.type)
36     u.color = BLACK
37     time = time + 1
38     u.f = time
```

Vi introducere at for en graf $G = (V, E)$ har en kant $e \in E$ et felt der bestemmer kantens type. En kant kan findes med funktionen $G.E(u, v)$. I `Modified-DFS` sætter vi typen for alle kanter til at være `NIL` for ikke at blive forstyrret af mulige tidligere gennemløb.

I `Modified-DFS-Visit` tilføjer vi for en knude v, i knuden u's naboliste:

1. Hvis v har farven `WHITE` så er kanten imellem de to knuder af typen `Tree-edge`.
2. Hvis v har farven `GRAY` og kanten $e = E(u, v)$ ikke er blevet givet en type endnu så er kanten af typen `Back`.
3. Hvis v har farven `BLACK` og kanten $e = E(u, v)$ ikke er blevet givet en type endnu er der to muligheder:
   (a) Hvis u blev opdaget før v er kanten af typen `Forward`
   (b) Hvis u blev opdaget efter v er kanten af typen `Cross`

   I dette case ved vi at begge knuder, u og v har fået sat deres opdagelses-tid, da u er

grå og v er sort.

Denne kode virker både for orienterede og ikke-orienterede grafer. I ikke-orienterede grafer returnere $G.E(u, v)$ og $G.E(v, u)$ den samme kant. Når vi gennemløber træet giver vi kun en kant en type hvis den ikke har en i forvejen, med mindre v har farven `WHITE` i hvilket tilfælde vi ikke kan have givet kanten en type i forvejen, da det er første gang kanten er tilgængelig. I orienterede grafer kan vi kun få fat i hver kant en gang så der er ingen fare for at ændre en kants type når den er blevet sat i første omgang.

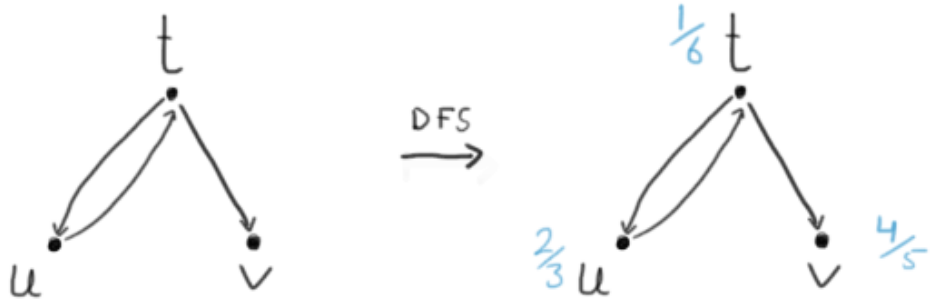
Som et alternativ kan man kører den originale DFS algoritme fra bogen. Når DFS algoritmen er færdig kan man gennemløbe alle kanter (u, v) i kantlistestrukturen. Her kan man så tildele kanter typer ud fra deres givne d- og f-værdier som på [3, slide 25-26]. Det kan dog ikke ses ud fra d- og f-værdier om en kant (u, v) er en tree- eller forward-kant. Her skal π-værdien bruges. Hvis $v.\pi = u$ er kanten en tree-kant, ellers er det en forward-kant.

Det skal også nævnes at da DFS kan give forskelligt resultat alt efter hvilke kanter der undersøges først gælder dette selvfølgelig også for pseudokoden vist her.

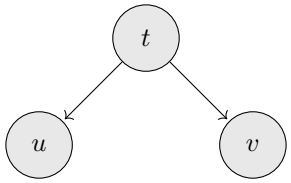
5 Opgave 5 - Cormen et al. øvelse 22.3-8 (side 611)

I denne opgave skal vi give et modeksempel til påstanden, om at hvis en graf G indeholder en sti fra u til v og $u.d < v.d$, så vil v være en efterkommer af u i skoven af træer som en Dybde-Først-Søgning producerede. Påstanden siger med andre ord at hvis en Dybde-Først-Søgning opdager u før v, så vil v være fundet ved at følge en sti fra u til v.

Så en måde at lave et modeksempel på kunne være at afskære u fra v inden u bliver opdaget. Det vil sige, først opdage en knude i stien mellem u og v, sådan at algoritmen ikke kan bruge stien til at gå fra u til v. Da det ikke er specificeret i Dybde-Først-Søgning algoritmen, hvor man skal starte i grafen så lad os antage at algoritmen starter i en knude t i stien fra u til v. Så vil vi gerne konstruere grafen G sådan at algoritmen først opdager u, og derefter går tilbage til t for at finde v. En sådan graf kan ses i figur 4, hvor man kan verificere at påstanden ikke holder for G, da v kun vil være en efterkommer af t (skoven kan ses i figur 5).



Figur 4: Dybde-Først-Søgning foretages på venstre graf, hvor knuderne vælges efter den alfabetiske rækkefølge. *Discovery* og *Finish* tiderne for knuderne efter Dybde-Først-Søgning på grafen er illustreret i højre graf.



Figur 5: Skoven for G, der er kun et træ med start i t

6 Opgave 6 - Cormen et al. øvelse 22.3-9 (side 612)

Giv et modeksempel til påstanden om, at hvis en graf G indeholder en vej fra u til v, så vil enhver Dybde-Først-Søgning resultere i $v.d \leq u.f$.

Påstanden siger altså, at en vej fra u til v er en tilstrækkelig betingelse for, at knuden v opdages (tilføjes til stakken) før u slutter (fjernes fra stakken). I Figur 4 er givet en graf, hvor der er en vej fra u til v gennem knuden t. Her har vi efter Dybde-Først-Søgning, at $v.d = 4$ og $u.f = 3$, men $v.d > u.f$; altså, vi har fundet et modeksempel til påstanden.

7 Opgave 7 - Cormen et al. øvelse 22.4-3 (side 615)

Vi er givet et hint, som siger at for en uorienteret graf gælder, at hvis $|E| \geq |V|$, så har grafen en kreds.[1]

I algoritmen kan vi derfor starte med at tælle kanterne, og hvis vi når at tælle mindst $|V|$ kanter, så ved vi med sikkerhed at der er en kreds, og dette kan algoritmen rapportere med det samme (der er ingen grund til at tælle flere kanter). Dette har vi højst brugt $O(|V|)$ tid på. Hvis vi ikke når at tælle mindst $|V|$ kanter, så ved vi ikke om der er en kreds eller ej, men vi har højst brugt $O(|V|)$ tid på at tælle.

Vi har nu brugt højst $O(|V|)$ tid, og vi ved ikke om der er en kreds eller ej. Men vi har fundet ud af, at antallet af kanter er mindre end $|V|$. Vi ved, at DFS kan afgøre om der er en kreds i en graf: hvis der optræder en back-kant under gennemkørslen af DFS, så er der en kreds, og hvis der ikke optræder en back-kant, så er der ikke en kreds. Vi kan derfor køre DFS på grafen, og hvis der optræder en back-kant rapporterer vi, at der er en kreds i grafen, og hvis der ikke optræder en back-kant rapporterer vi, at der ikke er en kreds i grafen. DFS har køretid $O(|V| + |E|)$, men siden vi ved at $|E| < |V|$, så bliver køretiden for DFS på grafen $O(|V|)$.

Som konklusion kan vores algoritme afgøre om der er en kreds i en graf, og den bruger højst $O(|V|)$ tid.

(Alternativt kunne man også køre DFS direkte, og så snart man ser en back-kant, rapportere at der er en kreds. Dette ville også være korrekt, men algoritmen vi har præsenteret her, er nem at se ikke overskrider køretidsbegrænsningen på $O(|V|)$).

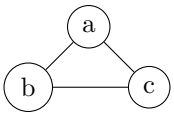
[1] Hvis man repræsenterer den uorienterede graf som en orienteret graf med to orienterede kanter for hver kant i den uorienterede graf (f.eks. hvis grafen er givet i en nabolistestruktur), vil dette svare til, at hvis $|E| \geq 2|V|$, så har grafen en kreds.

8 (*) Opgave 8 - Cormen et al. øvelse 22.2-7 (side 602)

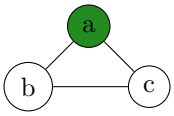
I denne opgave skal vi opstille en algoritme, der givet en graf med n knuder og m kanter, kan opdele knuderne i to mængder, således at ingen kant forbinder to knuder fra samme mængde. Hvis sådan en opdeling er umulig, så skal algoritmen returnere "False". Algoritmens køretid må maximal være $O(n + m)$.

Først skal vi tænke på hvordan opdelingen skal beskrives. Her bruges farverne rød og grøn som de to mængder.

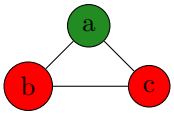
For at lave en god algoritme skal man først forstå problemet. Generelt er et konkret eksempel et godt sted at starte. Hvis vi kigger på en sammenhængende graf:



Knude a kan enten farves rød eller grøn. Hvilken en der vælges betyder intet, da a er vores begyndelsespunkt. Her farves a grøn.



Det kan nu ses, at der kun er en mulig farvning for b og c; den farve som a ikke har.



Vi kan nu se, at både b og c er røde, og der er en kant imellem dem. Altså kan b og c ikke begge to farves røde. Men deres farve kan kun være rød pga. ovenstående. Dette efterlader en ændring hos a's farvning, da der var et valg her. Dog kan det ses, at hvis a blev farvet rødt, så vil samme problem opstå, da farverne for hele grafen bare bliver byttet. I dette eksempel kan knuderne ikke opdeles.

Fra dette eksempel kan vi lære 3 ting:

1. Hvilken farve, den første knude får, har ingen betydning.
2. Farverne af en farvet knudes naboer kan kun være det modsatte af den farvede knudes farve.
3. Hvis to nabo-knuder har samme farve, så kan grafens knuder ikke opdeles.

Hvis vi kigger på BFS, som der bliver forslået af Rolf, så kan der ses, at BFS kigger på knuderne i samme rækkefølge, som vi gjorde i det tidligere eksempel. Dvs. vælg en startknude s, og kig på alle s's naboer. Derefter kig på alle s's naboers naboer osv.

Følgende algoritme skabes:

```
1  BFS-Split(G, s)
2  s.color = green
3  Q = ∅
4  ENQUEUE(Q,s)
5  while Q ≠ ∅
6      u = DEQUEUE(Q)
7      for each v ∈ G.Adj[u]
8          if v.color == u.color
9              return FALSE
10         if v.color == white
11             if u.color == green
12                 v.color == red
13             else
14                 v.color == green
15             ENQUEUE(Q, v)
16
17 return TRUE
```

Algoritmens start (linje 2-7) foregår som BFS med den ændring, at startpunktet s farves grøn. I while-løkken kigges der på en knude, u, fra køen Q. For hver af u's naboer, kaldet v her, laves følgende:

- Hvis v har samme farve som u, så kan der ikke laves en opdeling.
- Hvis v ikke er farvet endnu (hvid), så skal v farves det modsatte af u og indsættes i Q.

Dette fortsætter indtil Q er tom.

Da en knude kun bliver indsat når den bliver farvet, og en knude bliver kun farvet når den ikke er farvet, så kan vi være sikker på, at hver knude maksimalt er i Q én gang.

Vi kan også være sikker på at algoritmen er korrekt. For enhver sammehængende graf er der to cases:

1. Grafen kan ikke opdeles, da enhver farve-opdeling af knuderne vil have mindst en kant mellem to knuder af samme farve.
2. Grafen kan opdeles, så der kan skabes en farve-opdeling af knuderne, uden at to knuder af samme farve deler en kant.

I begge tilfælde gør algoritmen det korrekte.

Case 1: Algoritmen laver en rød/grøn opdeling af knuderne og opdager derfor sådan en kant, hvorved den returnerer "FALSE".

Case 2: Antag startpunktet s er grøn (ellers byt om på rød og grøn i opdelingen). Det følger nu pr. induktion over antal farvetildelinger i algoritmen at de tildelte farver svarer til denne opdeling: hvis u har en korrekt tildelt farve og kanten (u,v) findes, må v have den modsatte farve i opdelingen, hvilket er, hvad algoritmen tildeler den.

Dog er dette ikke den fulde løsning. Opdelingen gælder kun for en sammenhængende graf[2], men hvad med en ikke-sammenhængende graf? Color skal også initialiseres til hvid for alle knuder, og algoritmen skal kunne returnere en opdelingen af knuderne, hvis en opdeling kan skabes.

[2]En graf hvor der er en sti mellem alle knuder i grafen

Derfor skabes følgende algoritme, ved brug af DFS stukturen[1, p.604]:

```
1  DFS-Split(G)
2  for each vertex u ∈ G.V
3      u.color = white
4
5  for each vertex u ∈ G.V
6      if u.color == white
7          if BFS-Split(G, u) == FALSE
8              return FALSE
9
10 Greens = {u | u.color == green, u ∈ G.V}
11 Reds = {u | u.color == red, u ∈ G.V}
12
13 return (Greens, Reds)
```

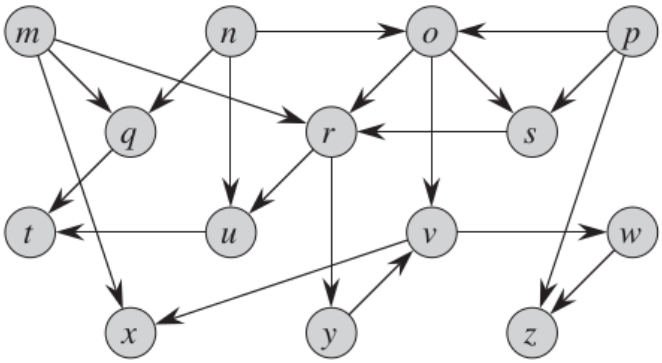
Først "farves"hver knude hvid (Linje 2-3). Derefter kaldes *BFS-Split* på hver sammenhængende del af grafen (Linje 5-8). Hvis en del ikke kan opdeles, så returnerer *BFS-Split* False, og *DFS-Split* returnerer False. Til sidst samles alle de grønne knuder i *Greens* (Linje 10), og alle de røde knuder i *Reds* (Linje 11). Disse mængder returneres.

Algoritmens køretid må være $O(n + m)$, da hver knude kun bliver indsat én gang i Q, og hver kant kun bliver undersøgt 2 gange, en gang fra hver af de to knuder som kanten forbinder:

$$n + 2m = \Theta(n + m)$$

9 Opgave 9 - Cormen et al. øvelse 22.4-1 (side 614)

I følgende opgave anvender vi Topological-Sort (Cormen et al. Section 22.4) på den givne orienterede acykliske graf i Figur 6, under antaglese af at grafens knuder betragtes i alfabetisk rækkefølge og knudernes nabolister er sorteret i alfabetisk rækkefølge.



Figur 6: Den givne graf (Cormen et al. Figure 22.8) som Topological-Sort anvendes på.

Topological-Sort består af to komponenter, nemlig (i) kørsel af DFS på grafen og (ii) indsættelse i en linked list som sluttider fremkommer. Ved kørsel af DFS på ovenstående graf, hvor forrige antagelser holder, der får vi følgende start- og sluttider (asterisk/stjerne indikerer nyt udgangspunkt for delproceduren DFS-Visit i DFS algoritmen linje 7, vist i Figur 3):

knude	m^*	q	t	r	u	y	v	w	z	x	n^*	o	s	p^*
starttid	1	2	3	6	7	9	10	11	12	15	21	22	23	27
sluttid	20	5	4	19	8	18	17	14	13	16	26	25	24	28

Tabel 1: De resulterende start- og sluttider der fremkommer ved kørsel af DFS på grafen i Firgur 6.

Ved at aflæse knuderne og deres korresponderende sluttider, i faldende orden, får vi følgende ordning af knuderne, som vil de vil fremkomme i en linked list:

sluttid	28	26	25	24	20	19	18	17	16	14	13	8	5	4
knude	p	n	o	s	m	r	y	v	x	w	z	u	q	t

Tabel 2: Den resulterende ordning af knuderne, som endeligt fås som output af Topological-Sort.

10 Opgave 10 - Cormen et al. øvelse 22.4-5 (side 615)

I denne opgave får vi at vide at vi kan udskrive den topologiske sortering af en orienteret, acyklisk graf $G = (V, E)$ ved at finde en knude i grafen med ind-grad 0, udskrive den og derefter fjerne alle knudens ud-kanter. Vi skal nu beskrive en algoritme til at udføre denne udskrivning i $O(|V| + |E|)$. Vi skal også beskrive hvad der sker hvis algoritmen får en graf der indeholder kredse (en ikke acyklisk graf).

Vi observere først at hvis en graf ikke har en kreds (grafen er acyklisk) så vil der være minimum en knude med ind-grad 0. Vælg en tilfældig knude i grafen, hvis denne har ind-grad 0 er vi færdige. Hvis knuden ikke har ind-grad 0 kan vi følge en af dens ind-kanter baglæns til kantens oprindelse. Vi kan blive ved med at følge ind-kanter baglæns indtil vi finder en knude med ind-grad 0. Da en graf $G = (V, E)$ har et begrænset antal knuder vil vi ikke kunne gøre dette uendeligt. Hvis vi på et tidspunkt finder en knude vi har besøgt før er der en kreds i grafen, den er altså ikke acyklisk.

Herunder er givet pseudokode til en algoritme der løser ovenstående problem. Når der i koden bliver refereret til en knude v er variablen lig knudens id som i dette tilfælde bliver set som heltal. De kan derfor bruges som indeks i lister mm.

```
1  TopologicalWrite(G)
2      let in_degree[1 ... (size(G.V))] be a new array
3      let L[1 ... (size(G.V))] be a new array
4      let l = 1 be the next unfilled position in L
5      for i = 1 to size(G.V)
6          in_degree[i] = 0
7          L[i] = -1
8      for each vertex u ∈ G.V
9          for each vertex v ∈ G.adj(u)
10             in_degree[v] += 1
```

```
11      for i = 1 to size(G.V)
12          if in_degree[i] == 0
13              L[l] = i
14              l += 1
15      for j = 1 to size(G.V)
16          if L[j] == -1
17              print("graph contains cycles")
18              return
19          u = L[j]
20          print(u)
21          for each vertex v ∈ G.adj(u)
22              G.deleteEdge(u, v)
23              in_degree[v] -= 1
24              if in_degree[v] == 0
25                  L[l] = v
26                  l += 1
```

I koden ovenfor indeholder listen L id'er for de knuder i grafen der har ind-grad 0.

Køretid: For-løkken på linje 5 gennemløber alle knuderne en gang, $O(|V|)$. For-løkkerne på linje 8 og 9 gennemløber alle knuder og alle kanter en gang, $O(|V|+|E|)$. For-løkken på linje 11 gennemløber alle knuderne en gang, $O(|V|)$. for-løkken på linje 15 og for-løkken på linje 21 gennemløber tilsammen alle knuder og kanter højst en gang, $O(|V|+|E|)$.

Tilsammen har vi så $O(|V|+|V|+|V|+|V|+|E|+|E|) = O(4|V|+2|E|) = O(|V|+|E|)$.

Korrekthed: Da vi på linje 8 til 10 gennemløber alle kanter, ved vi, at vi i `in_degree` har taget højde for alle kanter i grafen. Når vi derefter indsætter knuder med ind-grad 0 i listen L ved vi at der ikke kan være kanter ind i nogen af disse. Derfor vil det være korrekt at udskrive enhver knude fra L som en af de første i den topologiske sortering.

Når vi har valgt en knude fra L fjerner vi alle de kanter der ligger i dens naboliste. Dette kan kun være ud-kanter da knuden ingen ind-kanter har. Vi tæller samtidig ind-graden ned for hver knude vi finder i nabolisten, hvis en af disse rammer ind-grad 0 bliver de tilføjet til L.

Hvis vi for en knude u i L finder at dens naboliste indeholder en knude v der tidligere er blevet udskrevet vil vi have en modstrid da knuden v ikke kunnet havde været i L til at starte på da dens ind-grad ville havde været større end 0 (i dette tilfælde er ind-graden for v større end 0 og vi fjerner kun kanter fra grafen så den kunne ikke være blevet placeret i L). Dette kan altså ikke ske.

På intet tidspunkt i algoritmen tilføjer vi kanter, vi kan derfor ikke skabe kredse eller introducere at en knude i L pludselig har en ind-grad over 0. Vi kan heller ikke gøre en acyklisk graf cyklisk ved at fjerne kanter, derfor vil der blive ved med at være en knude med ind-graden 0.

Tilfælde med kreds: Hvis grafen har en kreds (grafen er altså ikke acyklisk) vil knuder der er tilgængelige via en sti fra kredsen ikke blive udskrevet af den ovenstående pseudokode da deres ind-grad aldrig kan blive 0. Koden vil i dette tilfælde eksekvere indtil if-udtrykket på linje 16 bliver sandt og koden stopper. Koden vil altså udskrive alle knuder der ikke kan tilgås via en sti fra kredsen og derefter stoppe uden at havde udskrevet alle knuder. -

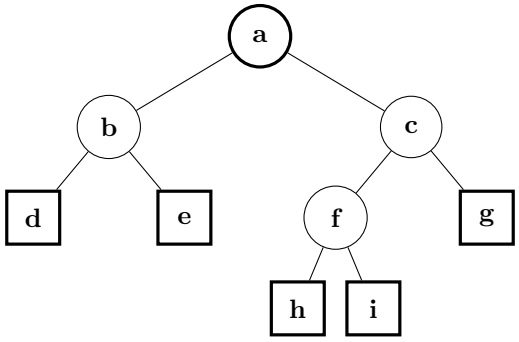
11 Opgave 11 - Eksamen juni 2012, opgave 1

I denne opgave skal vi kigge på forskellige måder at farve binære træer på sådan at de bliver til rødsorte træer. For at gøre det bliver vi nødt til at huske tilbage til kravene for rødsorte træer og for at opsummere så skal et rødsort træ overholde følgende:

1. Træet skal overholde reglerne for et binært søgetræ
2. Hver knude skal være enten rød eller sort
3. Roden og bladene skal være sorte
4. Der må ikke være 2 røde knuder i streg på en rod-blad sti (dvs. som er forbundet)
5. Der skal være samme antal sorte knuder på alle rod-blad stier

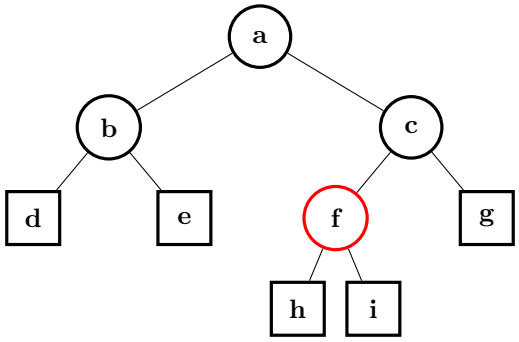
Spørgsmål a

For en farvning af knuder sådan at træet bliver til et rødsort træ kan vi starte med at farve bladene og roden sorte som de skal være ifølge kravene. Det resulterer i træet i figur 7



Figur 7: Træet med roden og bladene farvet sort

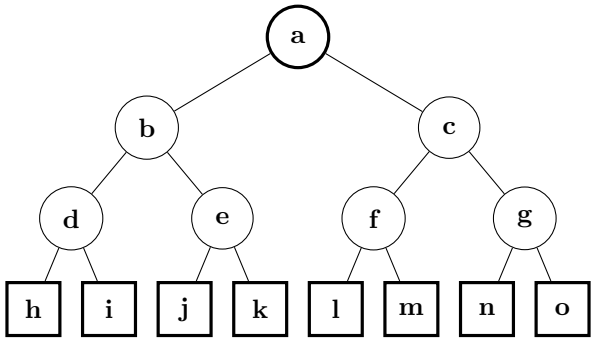
Så mangler vi bare at farve b, c og f. Vi kan udelukke at b skal være rød for så skal både c og f være røde for at træet kan leve op til krav 5, altså skal b farves sort. For de resterende knuder c og f kan vi udelukke at farve dem begge sorte, da det også ville bryde krav 5. Derfor skal en af knuderne være sort og den anden rød. Prøver man at farve f sort vil man kunne se at der ikke ville være lige mange sorte på alle rod-blad stier (der vil kun være a, g på stien a-c-g, og 3 sorte på alle andre stier). Da vi har udelukket alle andre muligheder skal knuderne være farvet som i figur 8 og man kan verifere at træet overholder alle kravene for et rødsort træ.



Figur 8: Træet farvet så det er et rødsort træ

Spørgsmål b

Her får vi givet et større, men fuldt binært træ (alle blade er i samme lag). Vi kan starte på samme måde som i opgaven ovenover, dvs. først gøre roden og bladene sorte. I figur 9 er disse farvet sorte.



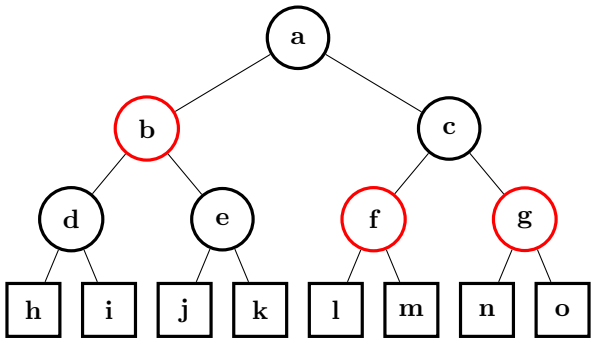
Figur 9: Træet med roden og bladene farvet sort

Så mangler vi bare at finde de måder man kan farve de sidste knuder på. Da træet er fuldt er alle rod-blad stier lige lange. Ud fra den observation kan man udlede at hvis vi f.eks. farver et helt lag rødt (udover det første og sidste) og lader resten være sorte, så ville vi have fjernet lige mange sorte i alle rod-blad stier. Der vil heller ikke være to forbundet røde knuder, da knuder i samme lag ikke er forbundet. Det vil sige, at **en løsning** ville være at vi farver knuderne b,c røde, lader resten være sorte og træet vil være et rødsort træ. **En anden løsning** ville være at gøre det samme for laget med knuderne d,e,f,g, altså farve dem røde og lade b,c være sorte.

Ved at farve et helt lag rødt og resten sort fjerner vi en sort knude fra alle rod-blad stier (i forhold til hvis alle var sorte). Vi kan derfor finde **en tredje løsning** ved at farve d,e,c røde og lade b,f,g være sorte. **En fjerde løsning** kunne være at gøre det modsatte, det vil sige farve b,f,g røde og d,e,c sorte. Et eksempel på dette kan ses i figur 10

Til sidst kan vi bruge at der ikke er noget krav om at der skal være røde knuder i træet til at finde **en femte løsning**. For alle rod-blad stier har samme længde kan vi farve alle knuderne

sorte uden at overtræde noget krav. Vi kan derfor se at der er 5 forskellige måder at farve træet på sådan at det overholder kravene for et rødsort træ.



Figur 10: Den fjerde løsning

12 Opgave 12 - Eksamen juni 2012, opgave 3

Spørgsmål a: Angiv løsningen til følgende rekursionsligning:

$$T(n) = 8 \cdot T\left(\frac{n}{4}\right) + n^{1.5} \tag{1}$$

Da (1) er på formen

$$T(n) = a \cdot T\left(\frac{n}{b}\right) + f(n),$$

hvor $a = 8$, $b = 4$ og $f(n) = n^{1.5}$, kan Master Theorem benyttes. Da $\alpha = \log_4 8 = 1.5$ og

$$f(n) = n^{1.5} = \Theta(n^{\alpha}) = \Theta(n^{1.5})$$

så er vi i Case 2 af Master Theorem.

Dvs. løsningen er

$$T(n) = \Theta(n^{\alpha} \log n) = \Theta(n^{1.5} \log n)$$

Spørgsmål b: Angiv for hver af følgende rekursionsligninger om de kan løses vha. Master Theorem. For hver ligning hvor svaret er positivt, angiv hvilken af de tre cases i Master Theorem som løser den. (Du behøver ikke angive selve løsningen.)

i) $T(n) = 14 \cdot T\left(\frac{n}{13}\right) + n$: Rekursionsligningen er på den rigtige form. Her er $a = 14$, $b = 13$ og $f(n) = n$. Da $\alpha = \log_{13} 14 = 1.0288...$ og

$$f(n) = n = O(n^{\alpha - \varepsilon}) = O(n^{1.0288... - \varepsilon})$$

for $\varepsilon = 0.01$, så kan Case 1 af Master Theorem benyttes.

ii) $T(n) = 13 \cdot T\left(\frac{n}{13}\right) + n \log n$: Rekursionsligningen er på den rigtige form. Her er $a = 13$, $b = 13$, $f(n) = n \log n$ og dermed bliver $\alpha = \log_{13} 13 = 1$. I Case 1 skal vi finde et $\varepsilon > 0$ sådan at $f(n) = n \log n = O(n^{1-\varepsilon})$, men da

$$\frac{n^{1-\varepsilon}}{n \log n} = \frac{n \cdot n^{-\varepsilon}}{n \log n} = \frac{1}{n^{\varepsilon} \log n} \to 0 \text{ for } n \to \infty$$

er $n^{1-\varepsilon} = o(n \log n)$, hvilket gør Case 1 ikke kan bruges. I Case 2 skal vi vise at $f(n) = n \log n = \Theta(n)$, men da $n = o(n \log n)$ kan Case 2 ikke bruges. I Case 3 skal vi finde et $\varepsilon > 0$ sådan at $f(n) = n \log n = \Omega(n^{1+\epsilon})$, men da

$$\frac{n \log n}{n^{1+\epsilon}} = \frac{n \log n}{n \cdot n^{\epsilon}} = \frac{\log n}{n^{\epsilon}} \to 0 \text{ for } n \to \infty \quad \text{(jf. [2, s. 19])}$$

er $n \log n = o(n^{1+\epsilon})$, hvilket gør vi ikke kan benytte Case 3; altså, vi kan overhovedet ikke benytte Master Theorem for denne rekursionsligning.

iii) $T(n) = 14 \cdot T\left(\frac{n}{13}\right) + n \log n$: Rekursionsligningen er på den rigtige form. Her er $a = 14$, $b = 13$ og $f(n) = n \log n$. Da $\alpha = \log_{13} 14 = 1.0288...$ og

$$f(n) = n \log n = O(n^{\alpha-\varepsilon}) = O(n^{1.0288...-\varepsilon})$$

for $\varepsilon = 0.01$, så kan Case 1 af Master Theorem benyttes.

iv) $T(n) = 13 \cdot T\left(\frac{n}{14}\right) + n$: Rekursionsligningen er på den rigtige form. Her er $a = 13$, $b = 14$ og $f(n) = n$. Da $\alpha = \log_{14} 13 = 0.9719...$ og

$$f(n) = n = \Omega(n^{\alpha+\varepsilon}) = \Omega(n^{0.9719...+\varepsilon})$$

for $\epsilon = 0.01$, så kan Case 3 af Master Theorem benyttes hvis der findes et $c < 1$ og et n_0 sådan at $a \cdot f\left(\frac{n}{b}\right) \leq c \cdot f(n)$ når $n \geq n_0$. Da

$$\begin{aligned} a \cdot f\left(\frac{n}{b}\right) &\leq c \cdot f(n) \Leftrightarrow \\ 13 \cdot \frac{n}{14} &\leq c \cdot n \Leftrightarrow \\ \frac{13}{14} &\leq c \end{aligned}$$

er uligheden opfyldt for f.eks. $c = \frac{13}{14}$ og $n_0 = 1$. Altså, Case 3 af Master Theorem kan benyttes.

Litteratur

[1] T.H. Cormen, C.E. Leiserson, R.L. Rivest, and C. Stein. *Introduction to Algorithms*. MIT Press, third edition, 2009.

[2] Rolf Fagerberg. Asymptotisk analyse af algoritmers køretider. URL `https://imada.sdu.dk/~rolf/Edu/DM507/F20/asymptotiskAnalyseAfAlg.pdf`, 2020.

[3] Rolf Fagerberg. Grafer og graf-gennemløb. URL `https://imada.sdu.dk/~rolf/Edu/DM507/F20/graphTraversalSlides.pdf`, 2020.