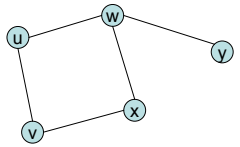
Sterke og 2-sammenhengende komponeneter, DFS

- Grafer (urettede og rettede)
- Dybde Først-Søk (DFS)
- Sammenhengende komponenter v.hj.a DFS
- Topologisk sortering / Løkker v.hj.a DFS
- Sterkt sammenhengende komponenter v.hj.a. DFS
- 2-sammenhengende komponenter v.hj.a. DFS

DFS er en **veldig** nyttig teknikk, andre anvendelser er bla. testing av bipartithet og planaritet.

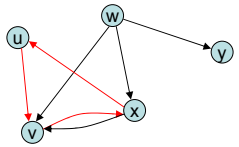
(urettede) Grafer

- En *graf* G=(V,E) består en mengde noder V=V(G), og en mengde kanter E=E(G).
- Hver *kant* er et par av noder {u,v}, slik at u,v ∈ V(G).
  (Vi skriver noen ganger bare uv).
- En kant {u,v} modellerer at nodene u og v er relatert til hverandre.



Rettede grafer

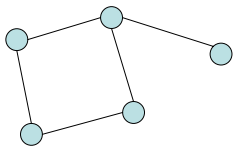
- En rettet graf D=(V,E) består en mengde noder V=V(G), og en mengde rettede kanter E=E(G).
- Hver kant er et ordnet par av noder (u,v), slik at u,v □ V(G).
  (Vi skriver noen ganger bare uv).
- En kant (u,v) modellerer at noden u står i relasjon til v.

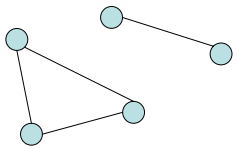


- En rettet graf uten (rettede) løkker kalles ofte en DAG (directed, acyclic graph).

Sammenhengende grafer

- Grafer kan være *sammenhengende*, eller *usammenhengende* (bestående av flere komponenter).
- En urettet graf er *sammenhengende* dersom det for ethvert par av noder u og v, finnes en sti mellom u og v.

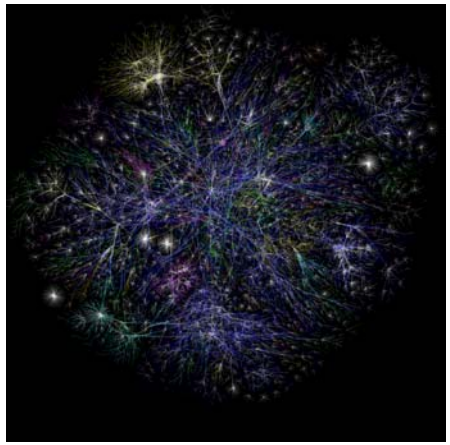




- Rettede grafer kan selvsagt også være usammenhengende, vi ser da på stier i den underliggende, urettede grafen.

Grafer

- Grafer er den mest fleksible datastrukturen vi kjenner (”alt” kan modelleres med grafer).
- Grafer finnes i mange problemer i dagliglivet
  - Datanettverk
  - Ruteplanlegging
  - VLSI (dagliglivet?)
  - …



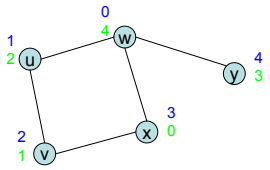
DFS

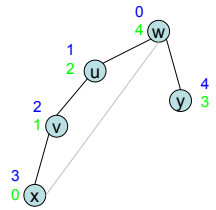
Malen for et dybde først-søk ser slik ut:

```
proc DFS(v)
{
        v.visited=TRUE
        for <hver nabo w av v> do
                if not w.visited then DFS(w)
        od
}
```

prenummer

postnummer





Sammenhengende komponenter

DFS kan brukes til å sjekke om en graf G er sammenhengende eller ei.

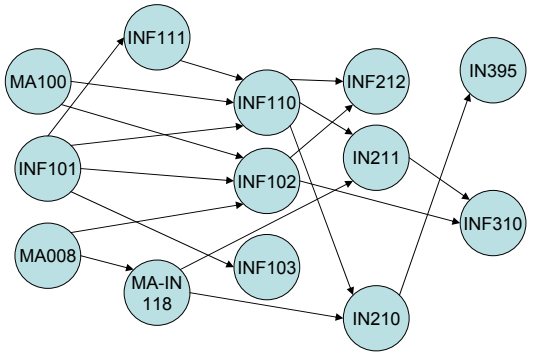
1. Start et dybde først-søk i en node.
2. Merk nodene når de aksesseres.
3. Er alle nodene merket når søket er over, er grafen sammenhengende.
4. Hvis ikke, finn en umerket node og fortsett med et nytt merke.

Topologisk sortering / løkkesøk

En topologisk sortering er en ordning av nodene i en graf, slik at hvis det finnes en sti fra u til v, så kommer u før v i sorteringen. (Ikke unik, flere mulig ordninger kan være lovlige.)

Har vi en rettet graf D, kan vi bruke DFS til å finne en topologisk sortering av nodene i grafen, evt. fremvise en løkke i D.

(Grafer med løkker kan ikke sorteres topologisk. Hvilken node i sykelen skulle i så fall komme først?)



MA100
INF101
MA008
INF111
MA-IN118
INF110
INF102
INF103
INF212
IN211
IN210
IN394
INF310

Topologisk sortering / løkkesøk

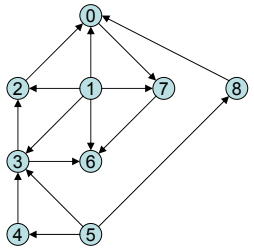
```
proc TopSort(D, Sortering[0, n-1])
    merke[0, n-1] = 0                          // Tabell med nodenes merker  (-1,0,1)
    teller = n-1
    for v = 0 to n-1 do
        if merke[v] = 0 then DFSOutTopLabel(D,v)
    endfor

    proc DFSOutTopLabel(D,v) recursive
        merke = 1
        for < hver nabo w av v > do
            if merke[w] = 0 then DFSOutTopLabel(D,w)      // w uutforsket, fortsett
            elseif merke[w] = 1 then Output("Løkke funnet")   // utforsker w, løkke
        endfor
        merke[v] = -1                                     // v ferdig utforsket
        Sortering[teller] = v
        teller = teller - 1
    end proc
end proc
```

$O(n+m)$

Topologisk sortering / løkkesøk

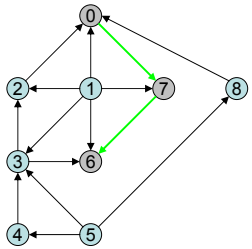
Eksempel



v	0	1	2	3	4	5	6	7	8
Sortering									

Topologisk sortering / løkkesøk

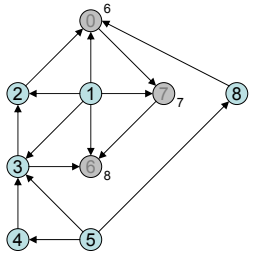
Eksempel



v	0	1	2	3	4	5	6	7	8
Sortering									

Topologisk sortering / løkkesøk

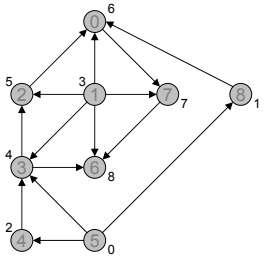
Eksempel



v	0	1	2	3	4	5	6	7	8
Sortering							0	7	6

Topologisk sortering / løkkesøk

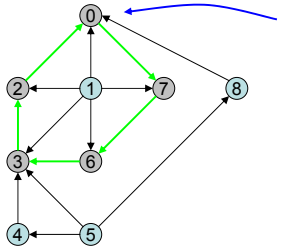
Eksempel



v	0	1	2	3	4	5	6	7	8
Sortering	1	8	4	1	3	2	0	7	6

Topologisk sortering / løkkesøk

Eksempel 2

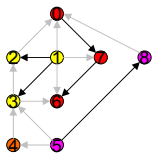


Node 0 er ”under utforskning”, merke[0] = 1.

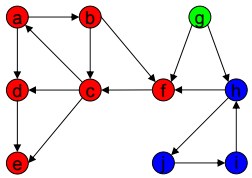
- LØKKE !

v	0	1	2	3	4	5	6	7	8
Sortering									

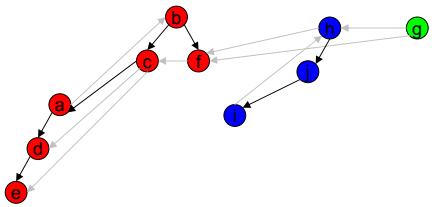
Sterkt sammenhengende komponenter



- Måtte gjøre flere DFS-søk
- Fikk skog av DFS-trær



(rettet) graf

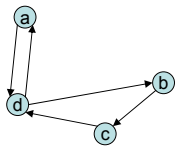


DFS-traversering

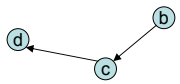
Klarer ikke traversere hele grafen i en jafs.

Sterkt sammenhengende komponenter

- En rettet graf er *sterkt sammenhengende* dersom det går en (rettet) sti fra enhver node til enhver annen node. (Både $u \rightarrow v$ og $v \rightarrow u$.)
- En rettet graf er *svakt sammenhengende* dersom den underliggende grafen er sammenhengende.



Sterkt sammenhengende



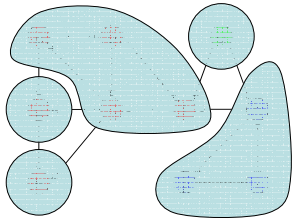
Svakt sammenhengende

- DFS kan brukes til å finne ut om en rettet graf er sterkt sammenhengende. Evt. til å finne de sterkt sammenhengende komponentene i grafen.

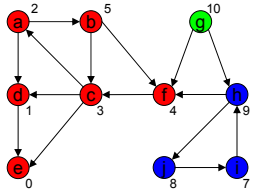
(Alternativ formulering: For ethvert par av noder, finnes en rettet sykel som inneholder nodene.)

Sterkt sammenhengende komponenter

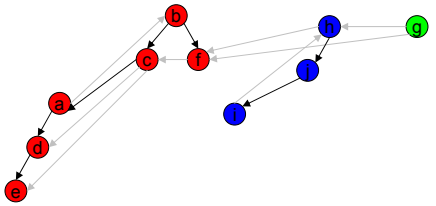
Bruker vi definisjonen av strekt sammenhengende komponenter, ser vi at vi har følgende strekt sammenhengende komponenter i grafen (se etter syklene).



Sterkt sammenhengende komponenter

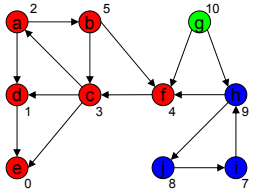


graf *D*

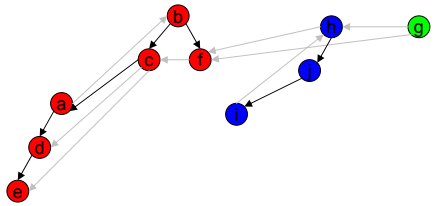


DFS-traversering av *D*

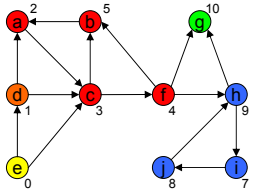
Sterkt sammenhengende komponenter



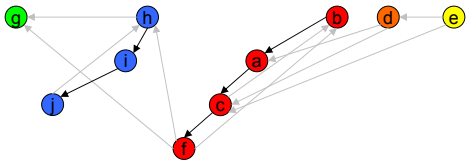
graf *D*



DFS-traversering av *D*

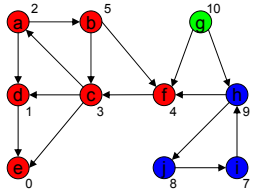


snudd graf D_r

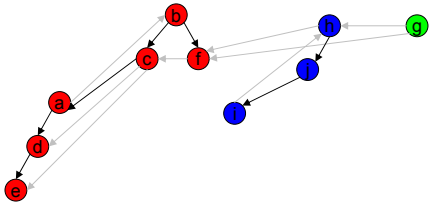


DFS-traversering av D_r,
noder med høyest postnummer tatt først.

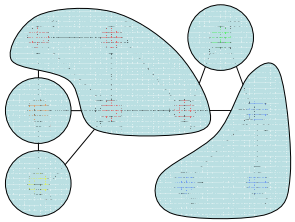
Sterkt sammenhengende komponenter



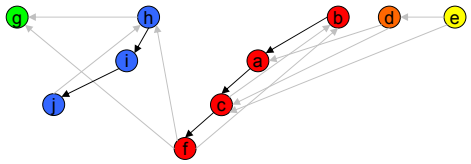
graf *D*



DFS-traversering av *D*

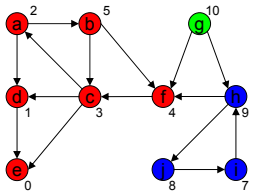


snudd graf D_r

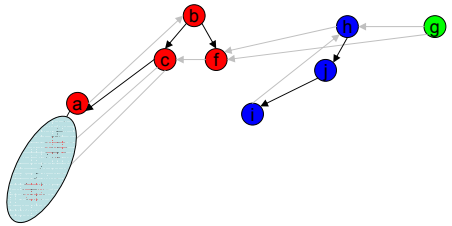


DFS-traversering av D_r,
noder med høyest postnummer tatt først.

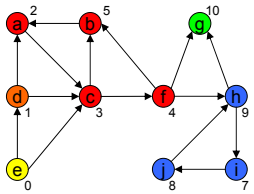
Sterkt sammenhengende komponenter



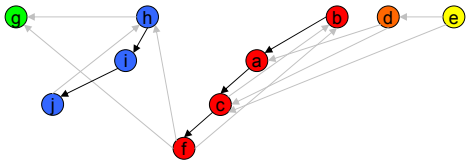
graf *D*



En DFS-traversering av *D tar med for mye*.

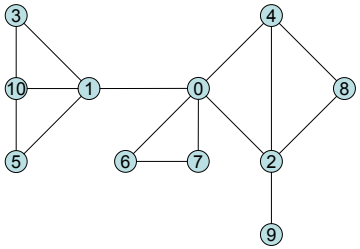


snudd graf D_r



En DFS-traversering av D_r i tillegg (rekkefølgen er viktig) kniper av det som for mye.

2-sammenhengende komponenter

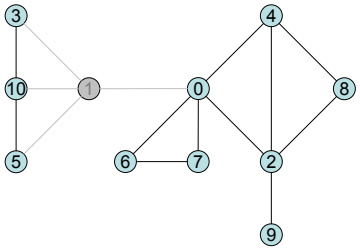


En node *v* i en graf *G* er et *artikulasjonspunkt* dersom grafen *G* – *v* (grafen som fremkommer ved å slette *v* fra *V*(*G*) og alle kanter som har *v* som et av sine endepunkter fra *E*(*G*)) er usammenhengende.

(Vi antar at *G* er sammenhengende til å begynne med)

En *2-sammenhengende komponent* av en graf *G* er en maksimal subgraf *B* som ikke inneholder noen artikulasjonspunkter.

2-sammenhengende komponenter

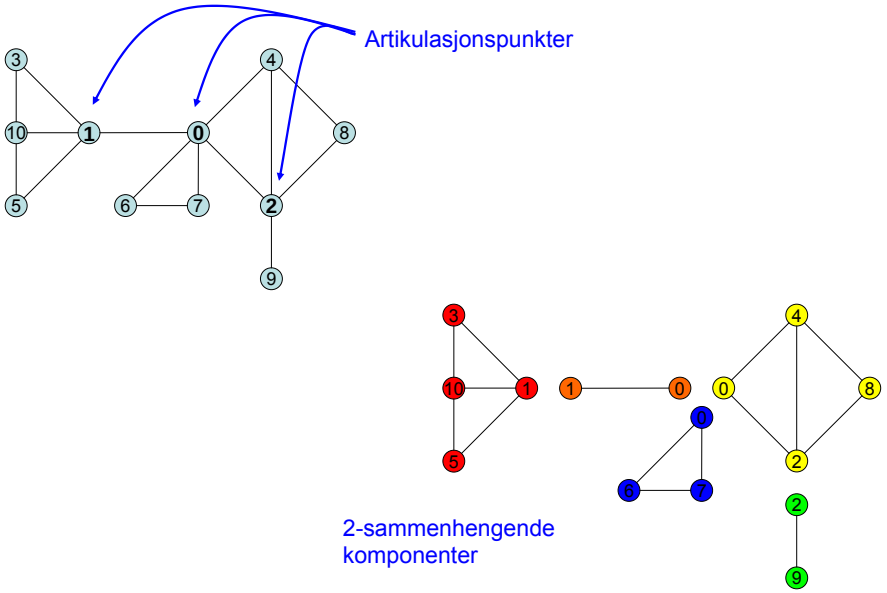


En node *v* i en graf *G* er et *artikulasjonspunkt* dersom grafen *G* – *v* (grafen som fremkommer ved å slette *v* fra *V*(*G*) og alle kanter som har *v* som et av sine endepunkter fra *E*(*G*)) er usammenhengende.

(Vi antar at *G* er sammenhengende til å begynne med)

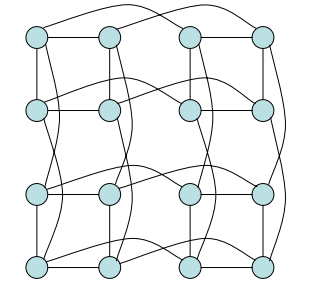
En *2-sammenhengende komponent* av en graf *G* er en maksimal subgraf *B* som ikke inneholder noen artikulasjonspunkter.

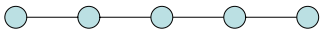
2-sammenhengende komponenter

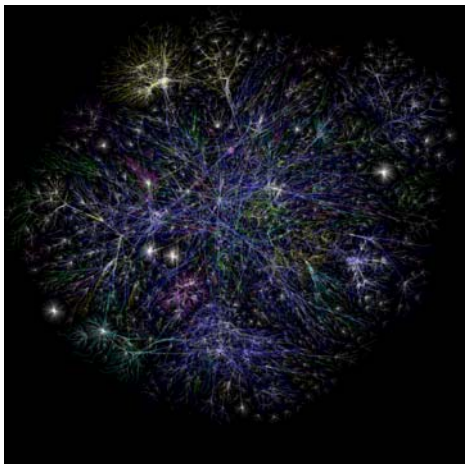


2-sammenhengende komponenter

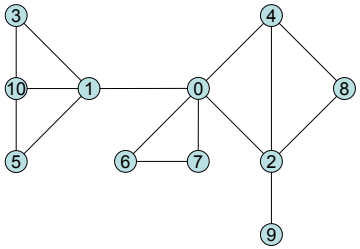
Artikulasjonspunkter





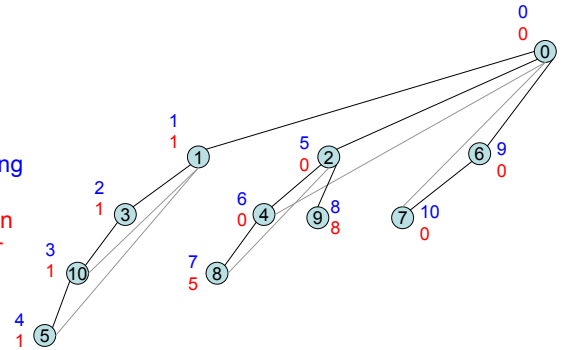


2-sammenhengende komponenter



DFSNum - preorder nummerering

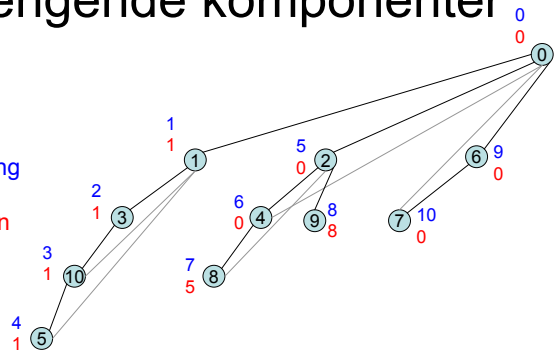
Lowest - laveste DFSNum vi kan komme til ved å følge tre-kanter og *en* tilbake-kant.



2-sammenhengende komponenter

DFSNum - preorder nummerering

Lowest - laveste DFSNum vi kan komme til ved å følge tre-kanter og *en* tilbake-kant.



En node *v* i et DFS-tre, *v* forskjellig fra roten i treet, er et artikulasjonspunkt i den underliggende grafen hvis og bare hvis *v* har minst et barn *c* med

Lowest[*c*] ≥ DFSNum(*v*) .

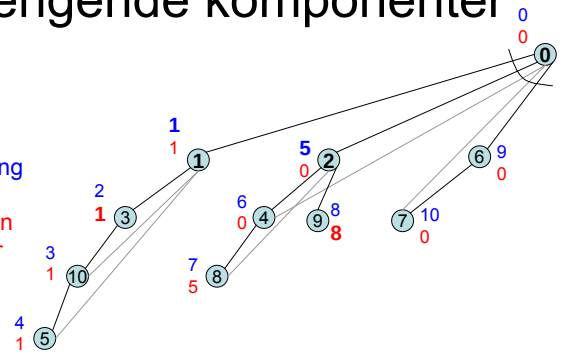
(Dette uttrykker at vi ikke ”kommer oss opp igjen” i treet.)

Roten i DFS-treet er et spesialtilfelle. Denne er et artikulasjonspunkt hvis og bare hvis den har to eller flere barn.

2-sammenhengende komponenter

DFSNum - preorder nummerering

Lowest - laveste DFSNum vi kan komme til ved å følge tre-kanter og *en* tilbake-kant.



En node *v* i et DFS-tre, *v* forskjellig fra roten i treet, er et artikulasjonspunkt i den underliggende grafen hvis og bare hvis *v* har minst et barn *c* med

Lowest[*c*] ≥ DFSNum(*v*) .

(Dette uttrykker at vi ikke ”kommer oss opp igjen” i treet.)

Roten i DFS-treet er et spesialtilfelle. Denne er et artikulasjonspunkt hvis og bare hvis den har to eller flere barn.

2-sammenhengende komponenter

DFSNum - preorder nummerering

Nummeret settes fortløpende med en gang vi kommer til en node.

Lowest - laveste DFSNum vi kan komme til ved å følge tre-kanter og *en* tilbakekant.

$$\text{Lowest}[v] = \min\begin{cases}\text{DFSNum}[v]. & & \text{Følger ingen kanter.}\\ \min\{\text{DFSNum}[w]\}, & vw \text{ en tilbakekant.} & \text{Følger beste tilbakekant.}\\ \min\{\text{Lowest}[c]\}, & c \text{ et barn av } v \text{ i treet.} & \text{Følger tre-kanter og muligens en tilbakekant.}\end{cases}$$

2-sammenhengende komponenter

```
proc DFSNumberings(G, v) recursive
    i = i +1                          // Global teller, initielt -1, s.a. vi får 0 første gang
    mark[v] = 1                       // Global array over nodenes merker, initielt 0
    DFSNum[v] = i                     // Global array med DFSNum (preorder nummerering)

    for < hver nabo u av v > do
        if mark[u] = 0 then           // u uutforsket, fortsett
            treeDFS[u] = v            // Gobalt forelder-array, v er u sin forelder i DFS-treet.
            DFSNumberings(G,u)        // Fortsett dybde først
        endif
    endfor

    min1 = {lowest[c] < for hvert barn c av v >}
    min2 = min {DFSNum[w] < vw en tilbakekant (w ikke forelder av v, sjekk treeDFS) >}
    lowest[v] = min {DFSNum[v], Min1, Min2}          // Globalt array med Lowest.
end proc
```



```
proc ArticulationPoints(G)
    for v = 0 to n-1 do               // Initier arrayer, tellere, liste over artikulasjonspunkter
        mark[v] = treeDFS = 0
    endfor
    i = -1
    art = ∅

    DFSNumberings(G, 0)               // Nummerer nodene, 0 blir rotnoden

    number = 0                        // Sjekker om rotnoden har flere enn to barn
    c = 1
    while number < 2 and c < n do
        if treeDFS[c] = 0 then number = number + 1
        c = c + 1
    endwhile
    if number = 2 then art = art ∪ {0}

    for c =1 to n-1 do                // Sjekker om de andre nodene er artikulasjonspkter
        v = treeDFS[c]                // finn forelder, v er c sin forelder i DFS-treet
        if lowest[c] ≥ DFSNum[v] and v > 0 then art = art ∪ {v}
    endfor

    return(art)
end proc
```

2-sammenhengende komponenter

Ønsker vi å finne de 2-sammenhengende komponentene, holder der ikke bare å finne artikulasjonspunktene. Noder kan tilhøre flere 2-sammenhengende komponenter.

Kanter kan derimot ikke tilhøre mer enn *en* 2-sammenhengende komponent.

For å finne de 2-sammenhengende komponenter bruker vi en ekstra stack til kantene, på følgende måte.

1. Initielt er kant-stacken tom.
2. Hver kan vi følger en kant *uv* i grafen i dybde først-søket pusher vi *uv*. (Alle kanter i grafen pushes (selv om noden *v* er utforsket). Det er altså ikke bare tre-kanter som havner på denne kant-stacken.
3. Når dybde først-søket backtracker langs en kant *uv*, fra *v* til *u*, og lowest[v] ≥ DFSNum[u], så popper vi alt som ligger på kant-stacken fram til og med kanten *uv.*
4. Kantene som poppes i en slik bolk induserer en 2-sammenhengende komponent.