2-sammenhengende grafer og sterkt sammenhengende komponenter

IN2010 – Algoritmer og Datastrukturer

Lars Tveito

Institutt for informatikk, Universitetet i Oslo
larstvei@ifi.uio.no

Høsten 2024

Oversikt

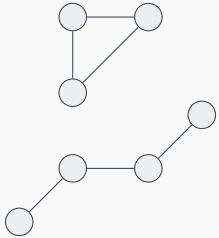
Oversikt

- Siste uke med grafer
- Vi skal lære hva det vil si at en urettet graf er 2-sammenhengende
  - og hvordan vi effektivt sjekker om den er 2-sammenhengende
- Vi skal lære hva de *sterkt sammenhengende komponentene* av en rettet graf er
  - og hvordan vi effektivt finner dem

2-sammenhengende grafer

Sammenhengende

- En graf kalles *sammenhengende* hvis det finnes en sti mellom alle par av noder
- En graf som ikke er sammenhengende kan deles inn i komponenter



- Vi kan sjekke om en urettet graf er sammenhengende med et dybde først søk
  - Grafen er sammenhengende hvis `DFSVisit` besøker alle noder i grafen
  - Dette er i $\mathcal{O}(|E|)$

2-sammenhengende grafer

- En graf $G = (V, E)$ er 2-sammenhengende hvis G forblir sammenhengende selv hvis en hvilken som helst node $v \in V$ fjernes fra G
  - Sagt annerledes, det finnes to *distinkte* stier mellom hver par av noder
  - To stier mellom u og v er *distinkte* dersom de ikke deler noen kanter og ikke deler noen noder utenom u og v
- Mer generelt sier vi at en graf er k-sammenhengende dersom grafen forblir sammenhengende hvis man fjerner færre enn k noder
- Dette er et nyttig begrep i anvendelser der det er et ønske om redundans
  - I et nettverk betyr det at en hvilken som helst maskin kan gå ned, og pakker vil fremdeles nå frem
  - Hos Ruter kan det bety at det kan være full stans rundt en holdeplass, men at det finnes en alternativ rute for alle reisende

En 2-sammenhengende graf (eksempel)

Er G 2-sammenhengende? (naiv)

- Definisjonen gir opphav til en enkel (men ineffektiv) løsning

ALGORITHM: NAIV ALGORITME FOR Å SJEKKE OM EN GRAF ER 2-SAMMENHENGENDE

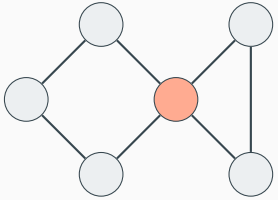
Input: En sammenhengende graf $G = (V, E)$
Output: Gir **true** hvis grafen er 2-sammenhengende, **false** ellers

```
1 Procedure IsBiconnectedNaive(G)
2     for v ∈ V do
3         G' = (V', E') ← G with v removed
4         visited ← empty set
5         u ← any vertex u ∈ V'
6         DFSVisit(G', u, visited)
7         if visited ≠ V' then
8             return false
9     return true
```

- Algoritmen er i $\mathcal{O}(|V| \cdot (|V| + |E|))$
  - som kan forenkles til $\mathcal{O}(|V|^3)$

Separasjonsnoder

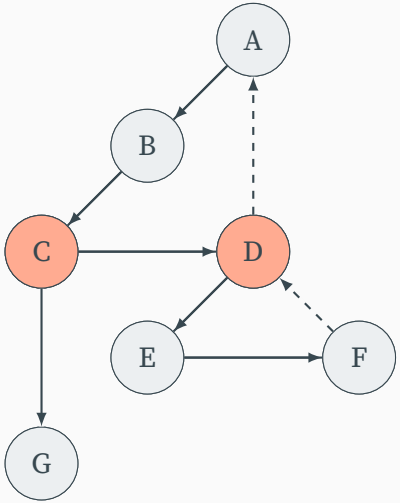
- Intuitivt er en separasjonsnode en node som holder grafen sammenhengende



- Hvis en separasjonsnode fjernes, så får grafen flere komponenter
- Dersom alle stier mellom to noder går gjennom den samme noden $v \in V$, så er v en separasjonsnode
- Vi skal se en algoritme som finner alle separasjonsnoder i $\mathcal{O}(|V| + |E|)$
  - Da får vi også en effektiv algoritme for å sjekke om en graf er 2-sammenhengende

Separasjonsnoder fra DFS

- Vi gjør et dybde-først søk, som gir opphav til et spenntre T
  - Dersom vi gikk fra u til v i søket lager vi en kant fra u til v i treet (discovery-edge)
- I tillegg holder vi styr på hvilke noder som kan nås som allerede er oppdaget (back-edge)
- En node u er en separasjonsnode hvis:
  - u er rot i T og har mer enn ett barn, eller
  - u ikke er roten, og har et barn v, slik at ingen etterkommere av v (inkludert v selv) har en back-edge til en forgjenger av u

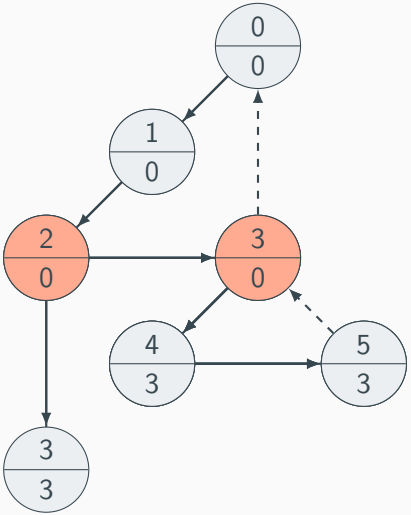


Finn separasjonsnoder fra DFS med `depth` og `low`

- Vi gjør et dybde-først søk
- For hver node u noterer vi:

  `depth`[u] *dybden* til u i T

  `low`[u] den laveste dybden som kan nås ved å følge én eller flere etterkommere av u og maksimalt én tilbakekant
- Dersom `depth`[u] $\leq$ `low`[v], der v er et barn av u, så er u en separasjonsnode



Finn separasjonsnoder

Algorithm: Finn alle separasjonsnoder i en sammenhengende graf

Input: En graf sammenhengende $G = (V, E)$
Output: Returnerer alle separasjonsnoder i G

```
1  depth ← empty map
2  low ← empty map
3  seps ← empty set
4  Procedure SeparationVertices(G)
5      s ← choose arbitrary vertex from V
6      depth[s] ← 0
7      low[s] ← 0
8      children ← 0
9
10     for (s, u) ∈ E do
11         if u ∉ depth then
12             SepRec(G, s, u, 1)
13             children ← children + 1
14
15     if children > 1 then
16         add s to seps
17     return seps
18
19 Procedure SepRec(G, p, u, d)
20     depth[u] ← d
21     low[u] ← d
22
23     for (u, v) ∈ E do
24         if v = p then
25             continue
26         if v ∈ depth then
27             low[u] ← min(low[u], depth[v])
28             continue
29
30         SepRec(G, u, v, d + 1)
31         low[u] ← min(low[u], low[v])
32         if d ≤ low[v] then
33             add u to seps
```

Er G 2-sammenhengende?

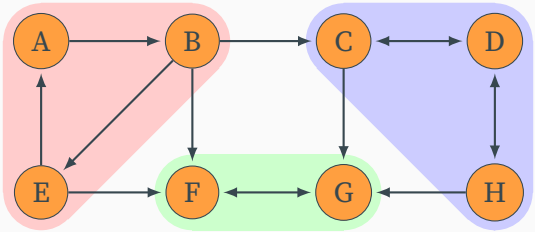
Algorithm: Er grafen 2-sammenhengende?

Input: En sammenhengende graf $G = (V, E)$
Output: Gir **true** hvis grafen er 2-sammenhengende, **false** ellers

```
1 Procedure IsBiconnected(G)
2     return SeparationVertices(G) is empty
```

Sterkt sammenhengende komponenter

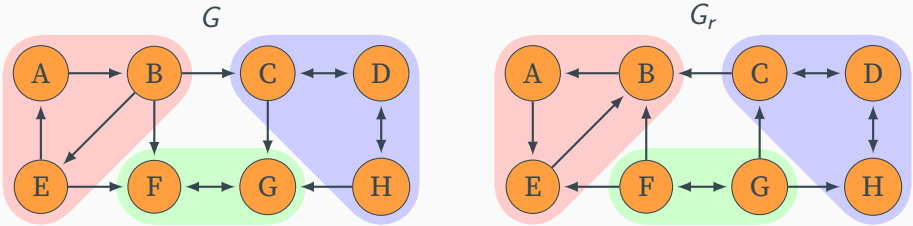
Sterkt sammenhengende komponenter



- En *rettet* graf er *sterkt* sammenhengende dersom det finnes en sti mellom alle par av noder
- En sterkt sammenhengende komponent er en delgraf slik at
  - Det finnes en sti mellom alle par av noder i komponenten
  - Komponenten er *maksimal*
- Intuitivt kan vi tenke på en sterkt sammenhengende komponent som en sykel

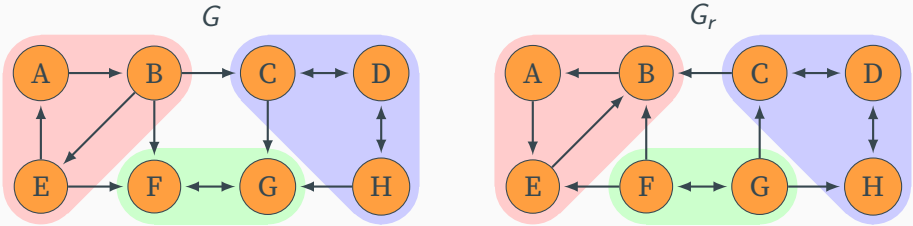
Den reverserte grafen

- Vi sier at den *reverserte* grafen til G er den grafen hvor alle kanter er snudd
  - Hvis $G = (V, E)$ sier vi at $G_r = (V, E_r)$ er den reverserte grafen
  - Hvis $(u, v) \in E$ så er $(v, u) \in E_r$



- Grafen G og dens reverserte graf G_r har alltid de samme sterkt sammenhengende komponentene!

Reverser graf



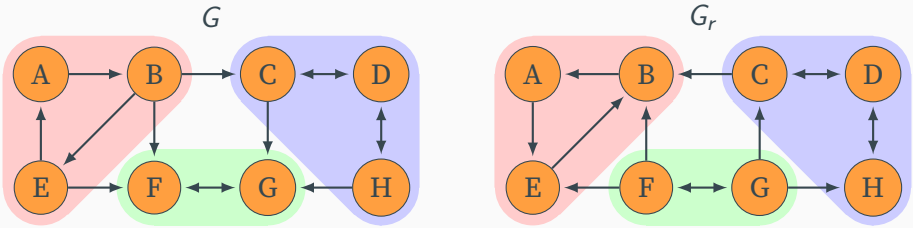
ALGORITHM: REVERSER EN GITT GRAF

Input: En graf $G = (V, E)$

Output: Returner den reverserte grafen $G_r = (V, E_r)$

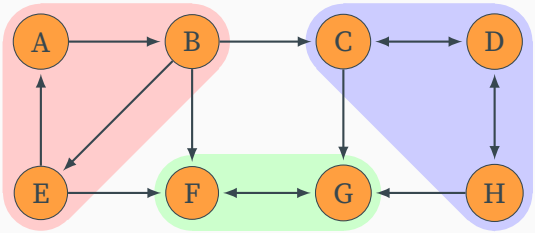
```
1 Procedure ReverseGraph(G)
2     E_r ← empty set
3     for (u, v) ∈ E do
4         add (v, u) to E_r
5     return (V, E_r)
```

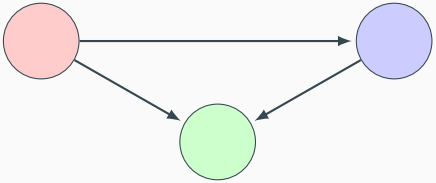
Finne sterkt sammenhengende komponenter



- Vi kan finne den sterkt sammenhengende komponenten til en node v ved å
  - Finne alle noder som kan nås fra v i G
  - Finne alle noder som kan nås fra v i G_r
  - Nodene som kunne nås i både G og G_r utgjør den sterkt sammenhengende komponenten til v
- Dette er en algoritme i $\mathcal{O}(|V| \cdot (|V| + |E|))$
  - Vi er en liten innsikt unna å få dette ned til $\mathcal{O}(|V| + |E|)$

Komponentgrafen





- Ved å anse hver sterkt sammenhengende komponent som en enkelt node, så får vi det vi kaller for *komponentgrafen*
- Komponentgrafen er garantert å ikke inneholde noen sykel (er *asyklisk*)
- Observer at noder i den topologisk siste komponenten kan ikke nå noen andre komponenter!

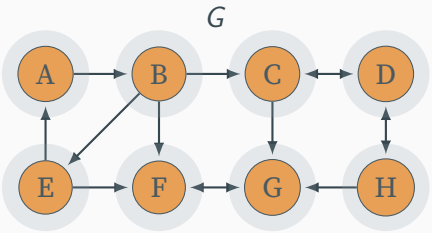
Topologisk ordning av den underliggende komponentgrafen

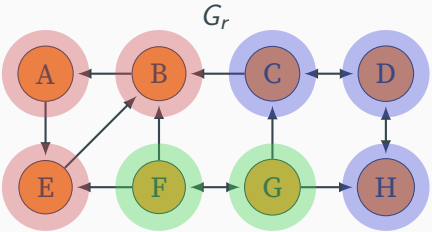
- En graf som inneholder sykler har ingen topologisk ordning
- Hvis vi bruker `DFSTopSort`, får vi nodene i en rekkefølge som respekterer ordningen av den *underliggende komponentgrafen*
- Den siste noden som blir prossesert i et dybde-først søk er topologisk først
- Den topologisk første noden er den topologisk *siste* noden i den *reverserte* grafen
- Ved å utforske noder i den motsatt topologisk rekkefølge, så er vi garantert å ikke krysse komponentgrenser

Kosaraju

- Algoritmen kan i korte trekk beskrives slik:
  1. Gjør et (fullt) dybde-først søk i en graf G, der hver node legges på en stack etter alle naboer er besøkt (`DFSTopSort`)
  2. Konstruer den reverserte grafen G_r
  3. Gjør et nytt (fullt) dybde-først søk på G_r, der det rekkefølgen i det fulle dybde-først søket er diktert av stacken fra det første søket.

Kosaraju (eksempel)





A
B
E
C
D
H
G
F

stack

Sterkt sammenhengende komponenter

ALGORITHM: FINN DE STERKT SAMMENHENGENDE KOMPONENETENE AV EN GRAF

Input: En rettet graf $G = (V, E)$

Output: Returner de sterkt sammenhengende komponenetene til G

```
1  Procedure StronglyConnectedComponents(G)
2      stack ← DFSTopSort(G)
3      G_r ← ReverseGraph(G)
4      visited ← empty set
5      components ← empty set
6      while stack is not empty do
7          u ← stack.pop()
8          if u ∉ visited then
9              component ← empty set
10             DFSVisit(G_r, u, visited, component)
11             add component to components
12     return components
```