Sammenhengende grafer

IN2010 – Algoritmer og datastrukturer

Lars Tveito

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

Oversikt

Oversikt

- Siste uke med grafer

Oversikt

- Siste uke med grafer
- Vi skal lære hva det vil si at en urettet graf er 2-sammenhengende

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

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

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

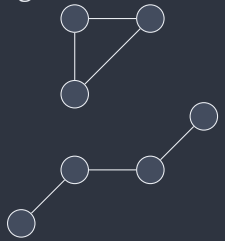
2-sammenhengende grafer

Sammenhengende

- En graf kalles *sammenhengende* hvis det finnes en sti mellom alle par av noder

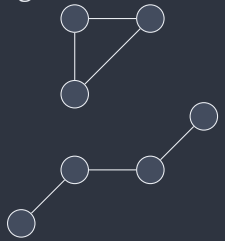
Sammenhengende

- 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



Sammenhengende

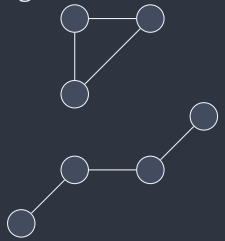
- En graf kalles *sammenhengende* hvis det finnes en sti mellom alle par av noder
- En graf som ikke er sammenhengende kan deles inn i komponenter



- Vi kan sjekke om en urettet graf er sammenhengende med et dybde først søk

Sammenhengende

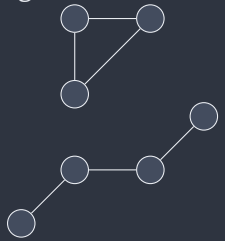
- En graf kalles *sammenhengende* hvis det finnes en sti mellom alle par av noder
- En graf som ikke er sammenhengende kan deles inn i komponenter



- Vi kan sjekke om en urettet graf er sammenhengende med et dybde først søk
  - Grafen er sammenhengende hvis `DFSVisit` besøker alle noder i grafen

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

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
- Følgende er ekvivalente for sammenhengende grafer med mer enn to noder:

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
- Følgende er ekvivalente for sammenhengende grafer med mer enn to noder:
  - Grafen er 2-sammenhengende

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
- Følgende er ekvivalente for sammenhengende grafer med mer enn to noder:
  - Grafen er 2-sammenhengende
  - Det finnes to *distinkte* stier mellom hvert par av ulike noder

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
- Følgende er ekvivalente for sammenhengende grafer med mer enn to noder:
  - Grafen er 2-sammenhengende
  - Det finnes to *distinkte* stier mellom hvert par av ulike noder
    - To stier mellom u og v er *distinkte* dersom de ikke deler noen noder utenom u og v

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
- Følgende er ekvivalente for sammenhengende grafer med mer enn to noder:
  - Grafen er 2-sammenhengende
  - Det finnes to *distinkte* stier mellom hvert par av ulike noder
    - To stier mellom u og v er *distinkte* dersom de ikke deler noen noder utenom u og v
  - Mellom hvert par av noder finnes det en sykel som forbinder dem

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
- Følgende er ekvivalente for sammenhengende grafer med mer enn to noder:
  - Grafen er 2-sammenhengende
  - Det finnes to *distinkte* stier mellom hvert par av ulike noder
    - To stier mellom u og v er *distinkte* dersom de ikke deler noen noder utenom u og v
  - Mellom hvert par av noder finnes det en sykel som forbinder dem
- Mer generelt sier vi at en graf er k-sammenhengende dersom grafen forblir sammenhengende hvis man fjerner færre enn k noder

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
- Følgende er ekvivalente for sammenhengende grafer med mer enn to noder:
  - Grafen er 2-sammenhengende
  - Det finnes to *distinkte* stier mellom hvert par av ulike noder
    - To stier mellom u og v er *distinkte* dersom de ikke deler noen noder utenom u og v
  - Mellom hvert par av noder finnes det en sykel som forbinder dem
- 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

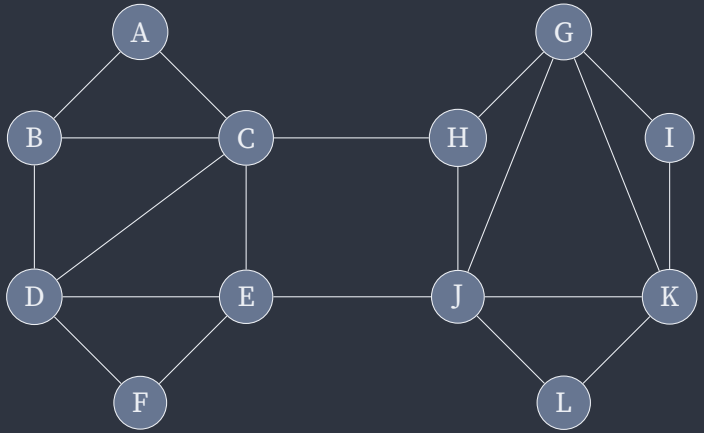
2-sammenhengende grafer

- 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
- Følgende er ekvivalente for sammenhengende grafer med mer enn to noder:
  - Grafen er 2-sammenhengende
  - Det finnes to *distinkte* stier mellom hvert par av ulike noder
    - To stier mellom u og v er *distinkte* dersom de ikke deler noen noder utenom u og v
  - Mellom hvert par av noder finnes det en sykel som forbinder dem
- 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

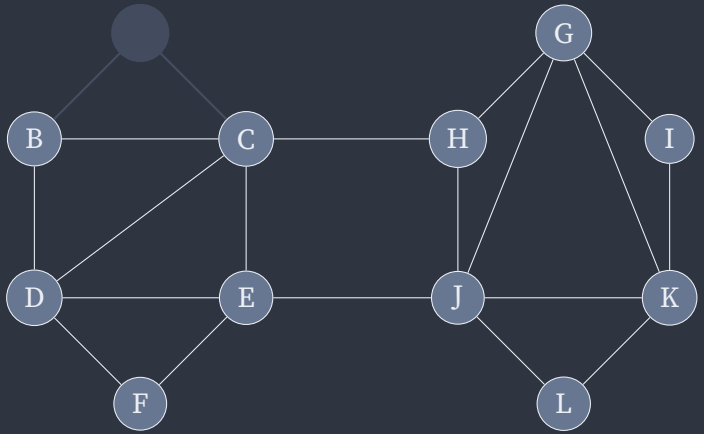
2-sammenhengende grafer

- 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
- Følgende er ekvivalente for sammenhengende grafer med mer enn to noder:
  - Grafen er 2-sammenhengende
  - Det finnes to *distinkte* stier mellom hvert par av ulike noder
    - To stier mellom u og v er *distinkte* dersom de ikke deler noen noder utenom u og v
  - Mellom hvert par av noder finnes det en sykel som forbinder dem
- 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

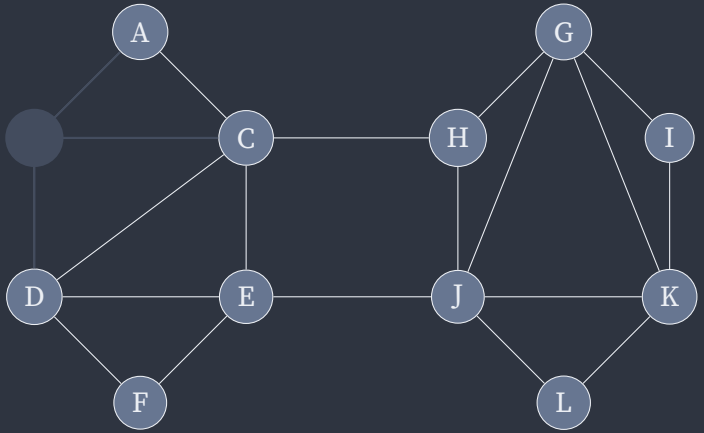
En 2-sammenhengende graf (eksempel)



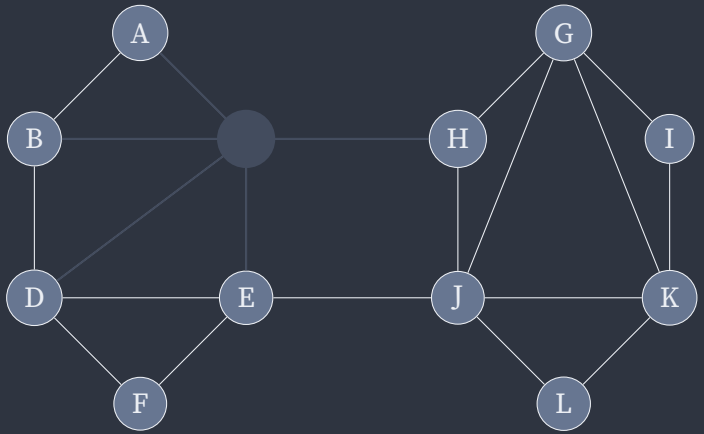
En 2-sammenhengende graf (eksempel)



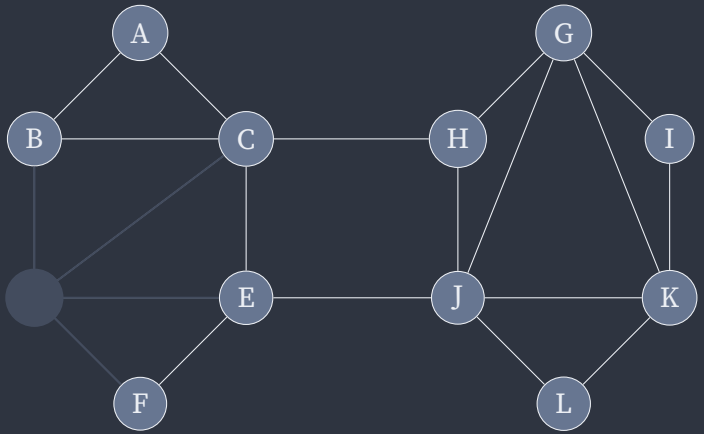
En 2-sammenhengende graf (eksempel)



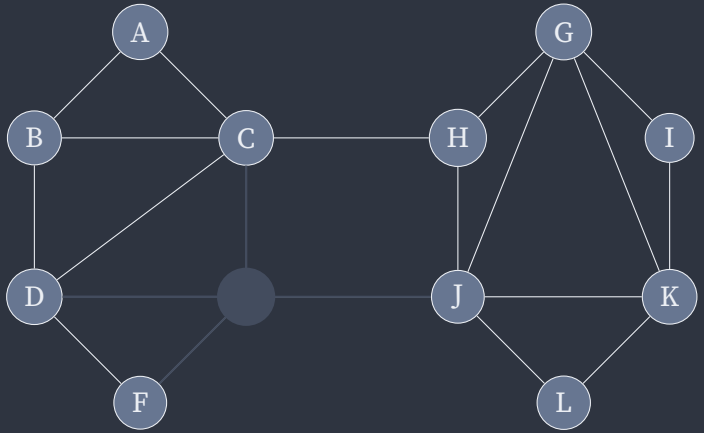
En 2-sammenhengende graf (eksempel)



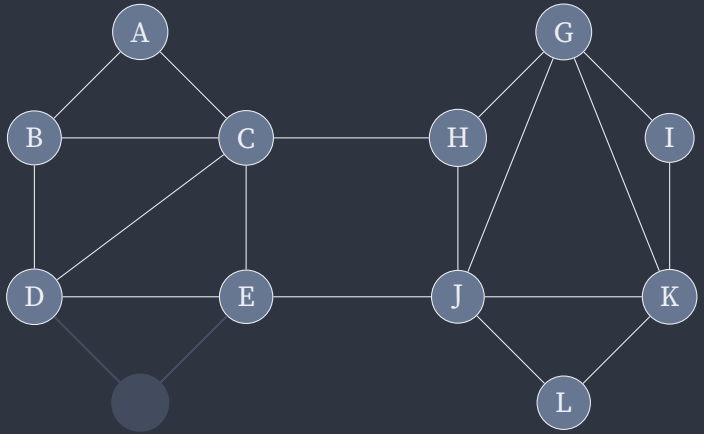
En 2-sammenhengende graf (eksempel)



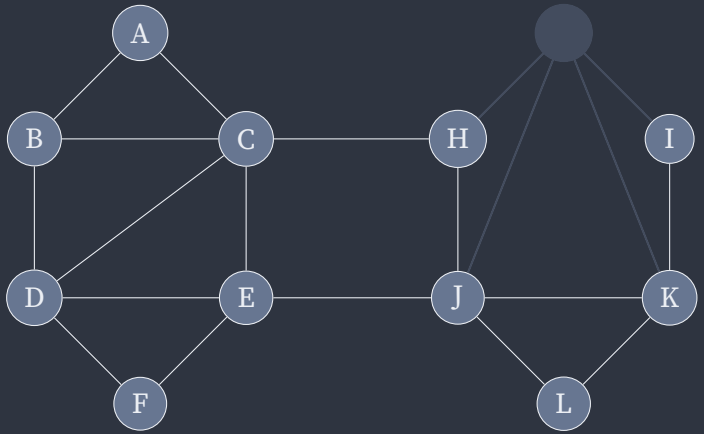
En 2-sammenhengende graf (eksempel)



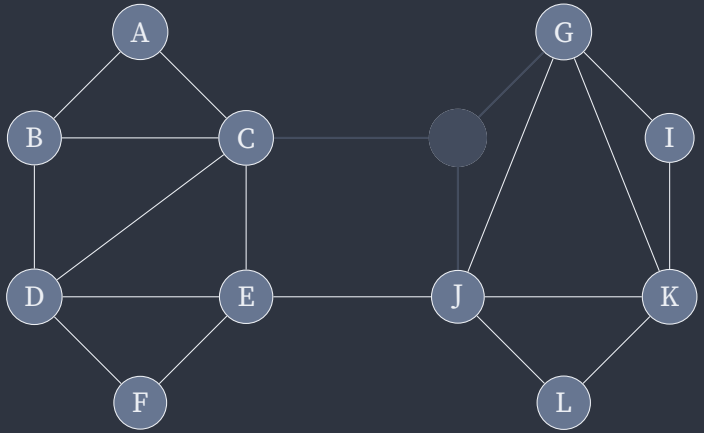
En 2-sammenhengende graf (eksempel)



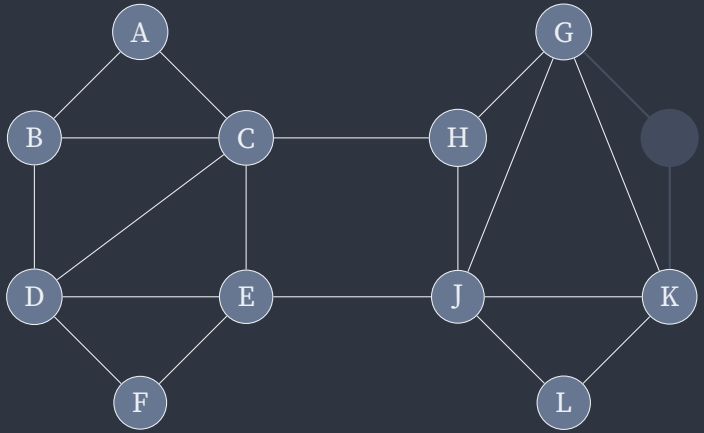
En 2-sammenhengende graf (eksempel)



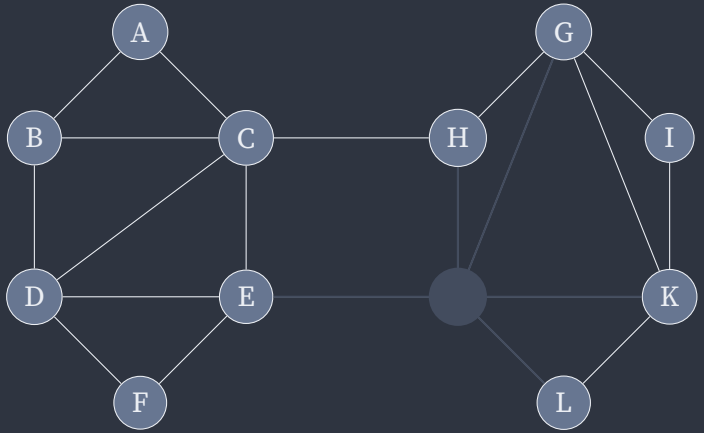
En 2-sammenhengende graf (eksempel)



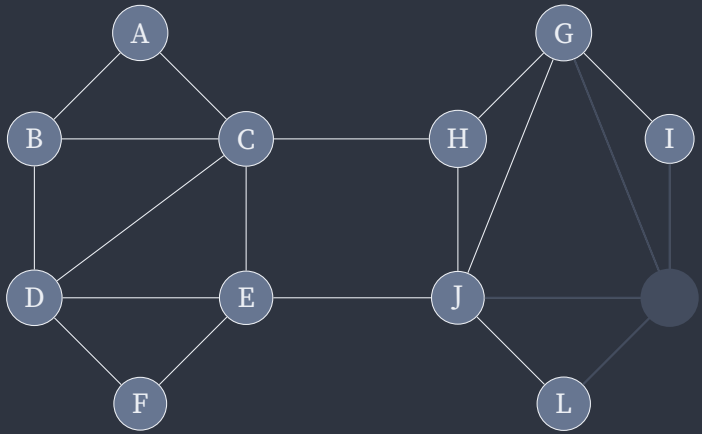
En 2-sammenhengende graf (eksempel)



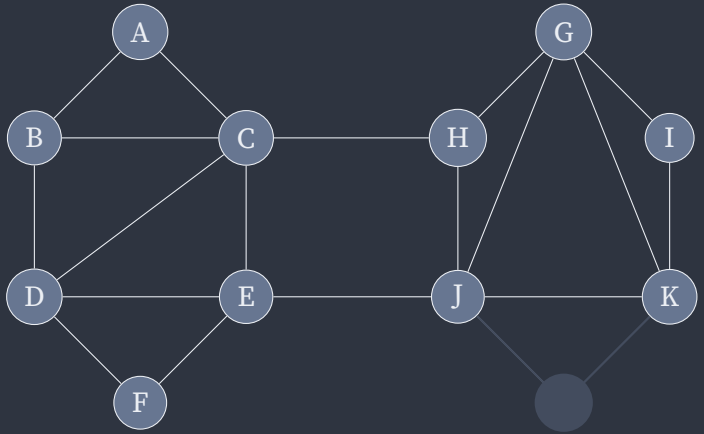
En 2-sammenhengende graf (eksempel)



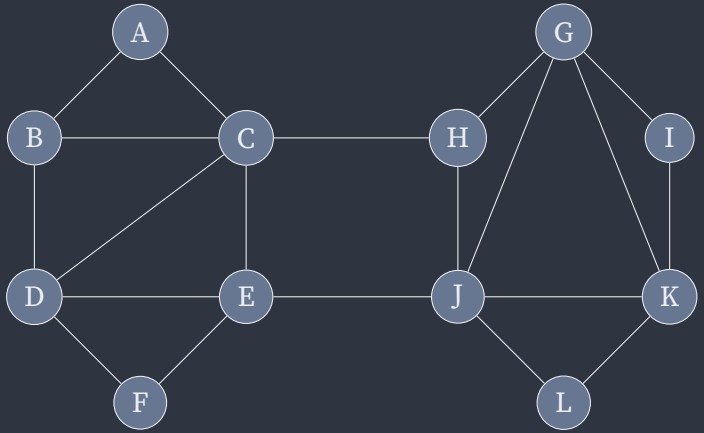
En 2-sammenhengende graf (eksempel)



En 2-sammenhengende graf (eksempel)



En 2-sammenhengende graf (eksempel)



Er G 2-sammenhengende? (naiv)

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

Er G 2-sammenhengende? (naiv)

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

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
```

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)
```

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
```

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 $\mathcal{O}(|V| \cdot (|V| + |E|))$

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

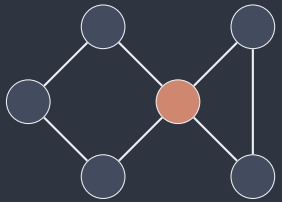
Input: En sammenhengende graf $G = (V, E)$
Output: Gir **true** hvis grafen er 2-sammenhengende, **false** ellers

```
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 $\mathcal{O}(|V|^3)$

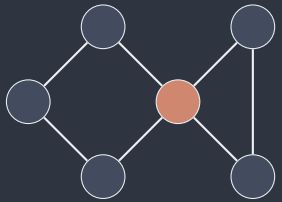
Separasjonsnoder

- Intuitivt er en separasjonsnode en node som holder grafen sammenhengende



Separasjonsnoder

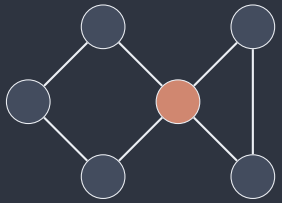
- Intuitivt er en separasjonsnode en node som holder grafen sammenhengende



- Hvis en separasjonsnode fjernes, så får grafen flere komponenter

Separasjonsnoder

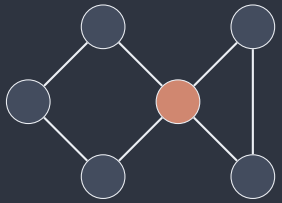
- Intuitivt er en separasjonsnode en node som holder grafen sammenhengende



- Hvis en separasjonsnode fjernes, så får grafen flere komponenter
- Dersom alle stier mellom to noder u og v går gjennom den samme noden $x \in V$, så er x en separasjonsnode (så lenge $x \neq u$ og $x \neq v$)

Separasjonsnoder

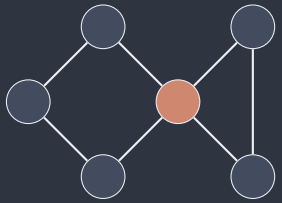
- Intuitivt er en separasjonsnode en node som holder grafen sammenhengende



- Hvis en separasjonsnode fjernes, så får grafen flere komponenter
- Dersom alle stier mellom to noder u og v går gjennom den samme noden $x \in V$, så er x en separasjonsnode (så lenge $x \neq u$ og $x \neq v$)
- Vi skal se en algoritme som finner alle separasjonsnoder i $\mathcal{O}(|V| + |E|)$

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 u og v går gjennom den samme noden $x \in V$, så er x en separasjonsnode (så lenge $x \neq u$ og $x \neq v$)
- 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 over en graf G, som gir opphav til et spenntre T

Separasjonsnoder fra DFS

- Vi gjør et dybde-først søk over en graf G, som gir opphav til et spenntre T
- Dersom vi gikk fra u til v i søket legger vi til en kant fra u til v i T

Separasjonsnoder fra DFS

- Vi gjør et dybde-først søk over en graf G, som gir opphav til et spenntre T
- Dersom vi gikk fra u til v i søket legger vi til en kant fra u til v i T
- Dersom det finnes en kant fra u til v i G og v allerede er lagt til i T kaller vi det en *tilbakekant*

Separasjonsnoder fra DFS

- Vi gjør et dybde-først søk over en graf G, som gir opphav til et spenntre T
- Dersom vi gikk fra u til v i søket legger vi til en kant fra u til v i T
- Dersom det finnes en kant fra u til v i G og v allerede er lagt til i T kaller vi det en *tilbakekant*
- En node u er en separasjonsnode hvis:

Separasjonsnoder fra DFS

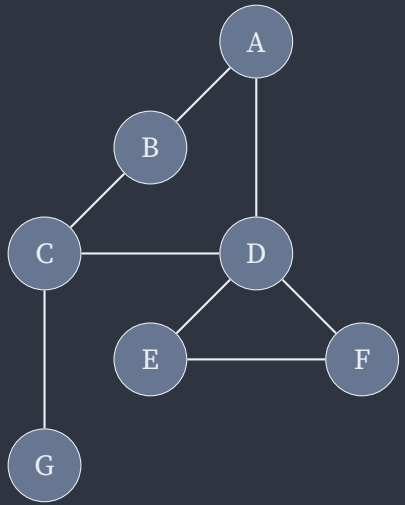
- Vi gjør et dybde-først søk over en graf G, som gir opphav til et spenntre T
- Dersom vi gikk fra u til v i søket legger vi til en kant fra u til v i T
- Dersom det finnes en kant fra u til v i G og v allerede er lagt til i T kaller vi det en *tilbakekant*
- En node u er en separasjonsnode hvis:
  - u er rot i T og har mer enn én utgående kant i T, eller

Separasjonsnoder fra DFS

- Vi gjør et dybde-først søk over en graf G, som gir opphav til et spenntre T
- Dersom vi gikk fra u til v i søket legger vi til en kant fra u til v i T
- Dersom det finnes en kant fra u til v i G og v allerede er lagt til i T kaller vi det en *tilbakekant*
- En node u er en separasjonsnode hvis:
  - u er rot i T og har mer enn én utgående kant i T, eller
  - u ikke er roten, og har et barn v, slik at ingen etterkommere av v har en tilbakekant til en forgjenger av u (ikke inkludert u selv)

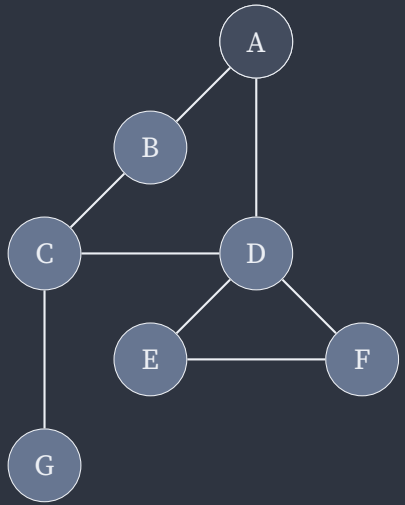
Separasjonsnoder fra DFS

- Vi gjør et dybde-først søk over en graf G, som gir opphav til et spenntre T
- Dersom vi gikk fra u til v i søket legger vi til en kant fra u til v i T
- Dersom det finnes en kant fra u til v i G og v allerede er lagt til i T kaller vi det en *tilbakekant*
- En node u er en separasjonsnode hvis:
  - u er rot i T og har mer enn én utgående kant i T, eller
  - u ikke er roten, og har et barn v, slik at ingen etterkommere av v har en tilbakekant til en forgjenger av u (ikke inkludert u selv)



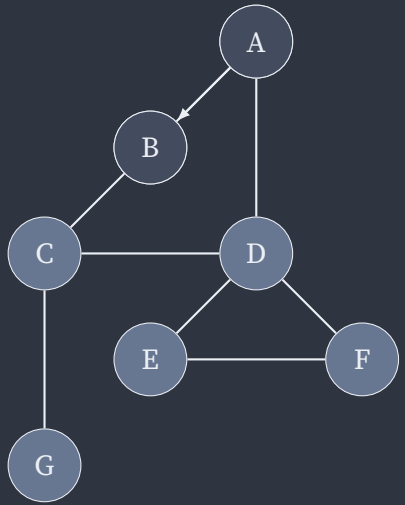
Separasjonsnoder fra DFS

- Vi gjør et dybde-først søk over en graf G, som gir opphav til et spenntre T
- Dersom vi gikk fra u til v i søket legger vi til en kant fra u til v i T
- Dersom det finnes en kant fra u til v i G og v allerede er lagt til i T kaller vi det en *tilbakekant*
- En node u er en separasjonsnode hvis:
  - u er rot i T og har mer enn én utgående kant i T, eller
  - u ikke er roten, og har et barn v, slik at ingen etterkommere av v har en tilbakekant til en forgjenger av u (ikke inkludert u selv)



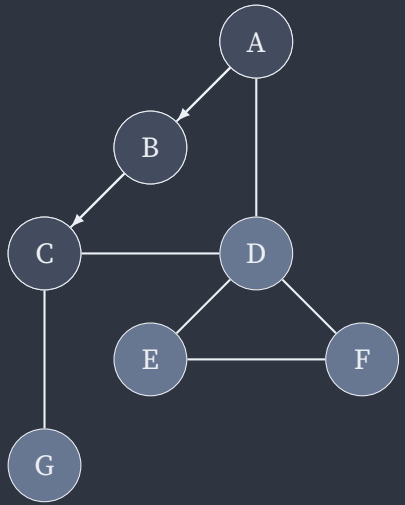
Separasjonsnoder fra DFS

- Vi gjør et dybde-først søk over en graf G, som gir opphav til et spenntre T
- Dersom vi gikk fra u til v i søket legger vi til en kant fra u til v i T
- Dersom det finnes en kant fra u til v i G og v allerede er lagt til i T kaller vi det en *tilbakekant*
- En node u er en separasjonsnode hvis:
  - u er rot i T og har mer enn én utgående kant i T, eller
  - u ikke er roten, og har et barn v, slik at ingen etterkommere av v har en tilbakekant til en forgjenger av u (ikke inkludert u selv)



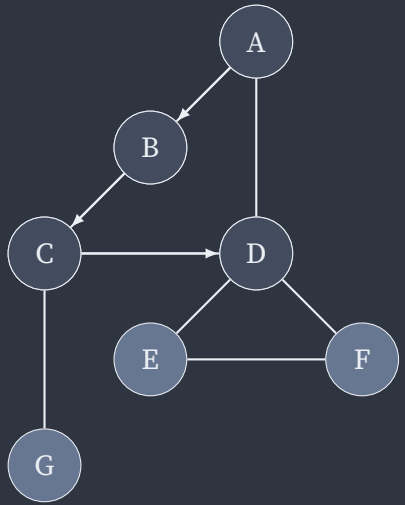
Separasjonsnoder fra DFS

- Vi gjør et dybde-først søk over en graf G, som gir opphav til et spenntre T
- Dersom vi gikk fra u til v i søket legger vi til en kant fra u til v i T
- Dersom det finnes en kant fra u til v i G og v allerede er lagt til i T kaller vi det en *tilbakekant*
- En node u er en separasjonsnode hvis:
  - u er rot i T og har mer enn én utgående kant i T, eller
  - u ikke er roten, og har et barn v, slik at ingen etterkommere av v har en tilbakekant til en forgjenger av u (ikke inkludert u selv)



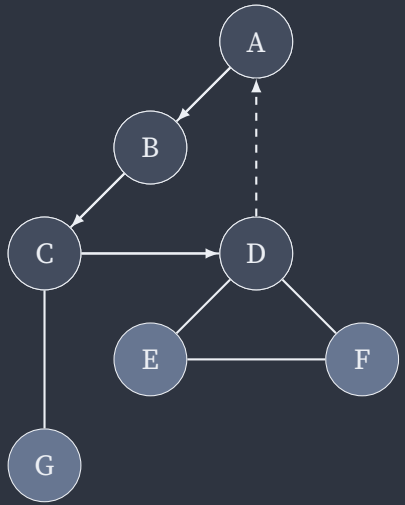
Separasjonsnoder fra DFS

- Vi gjør et dybde-først søk over en graf G, som gir opphav til et spenntre T
- Dersom vi gikk fra u til v i søket legger vi til en kant fra u til v i T
- Dersom det finnes en kant fra u til v i G og v allerede er lagt til i T kaller vi det en *tilbakekant*
- En node u er en separasjonsnode hvis:
  - u er rot i T og har mer enn én utgående kant i T, eller
  - u ikke er roten, og har et barn v, slik at ingen etterkommere av v har en tilbakekant til en forgjenger av u (ikke inkludert u selv)



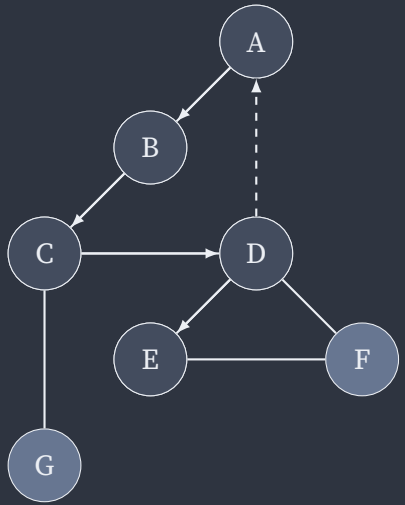
Separasjonsnoder fra DFS

- Vi gjør et dybde-først søk over en graf G, som gir opphav til et spenntre T
- Dersom vi gikk fra u til v i søket legger vi til en kant fra u til v i T
- Dersom det finnes en kant fra u til v i G og v allerede er lagt til i T kaller vi det en *tilbakekant*
- En node u er en separasjonsnode hvis:
  - u er rot i T og har mer enn én utgående kant i T, eller
  - u ikke er roten, og har et barn v, slik at ingen etterkommere av v har en tilbakekant til en forgjenger av u (ikke inkludert u selv)



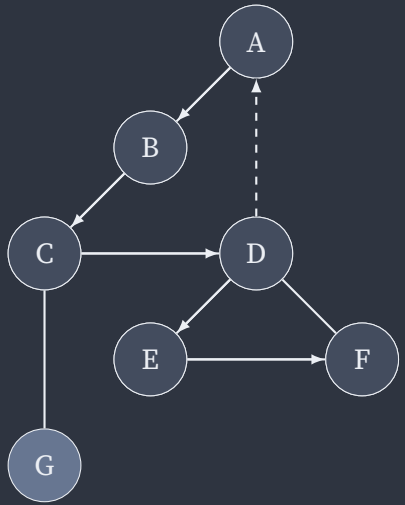
Separasjonsnoder fra DFS

- Vi gjør et dybde-først søk over en graf G, som gir opphav til et spenntre T
- Dersom vi gikk fra u til v i søket legger vi til en kant fra u til v i T
- Dersom det finnes en kant fra u til v i G og v allerede er lagt til i T kaller vi det en *tilbakekant*
- En node u er en separasjonsnode hvis:
  - u er rot i T og har mer enn én utgående kant i T, eller
  - u ikke er roten, og har et barn v, slik at ingen etterkommere av v har en tilbakekant til en forgjenger av u (ikke inkludert u selv)



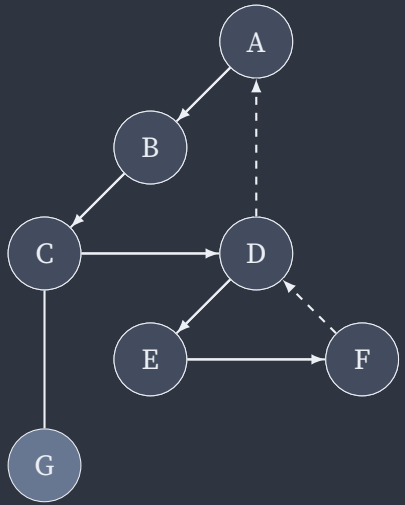
Separasjonsnoder fra DFS

- Vi gjør et dybde-først søk over en graf G, som gir opphav til et spenntre T
- Dersom vi gikk fra u til v i søket legger vi til en kant fra u til v i T
- Dersom det finnes en kant fra u til v i G og v allerede er lagt til i T kaller vi det en *tilbakekant*
- En node u er en separasjonsnode hvis:
  - u er rot i T og har mer enn én utgående kant i T, eller
  - u ikke er roten, og har et barn v, slik at ingen etterkommere av v har en tilbakekant til en forgjenger av u (ikke inkludert u selv)



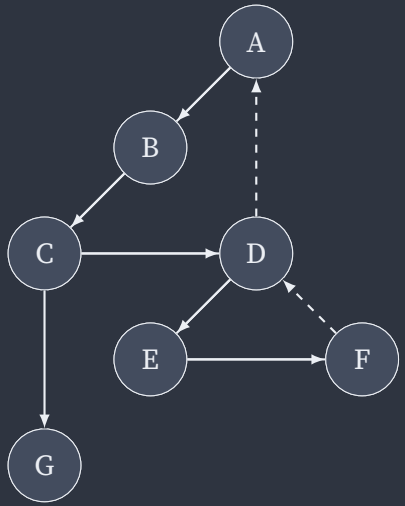
Separasjonsnoder fra DFS

- Vi gjør et dybde-først søk over en graf G, som gir opphav til et spenntre T
- Dersom vi gikk fra u til v i søket legger vi til en kant fra u til v i T
- Dersom det finnes en kant fra u til v i G og v allerede er lagt til i T kaller vi det en *tilbakekant*
- En node u er en separasjonsnode hvis:
  - u er rot i T og har mer enn én utgående kant i T, eller
  - u ikke er roten, og har et barn v, slik at ingen etterkommere av v har en tilbakekant til en forgjenger av u (ikke inkludert u selv)



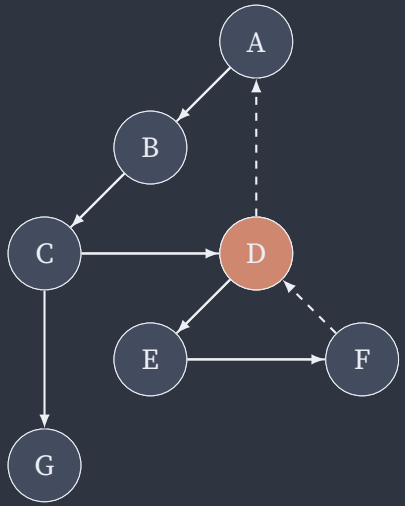
Separasjonsnoder fra DFS

- Vi gjør et dybde-først søk over en graf G, som gir opphav til et spenntre T
- Dersom vi gikk fra u til v i søket legger vi til en kant fra u til v i T
- Dersom det finnes en kant fra u til v i G og v allerede er lagt til i T kaller vi det en *tilbakekant*
- En node u er en separasjonsnode hvis:
  - u er rot i T og har mer enn én utgående kant i T, eller
  - u ikke er roten, og har et barn v, slik at ingen etterkommere av v har en tilbakekant til en forgjenger av u (ikke inkludert u selv)



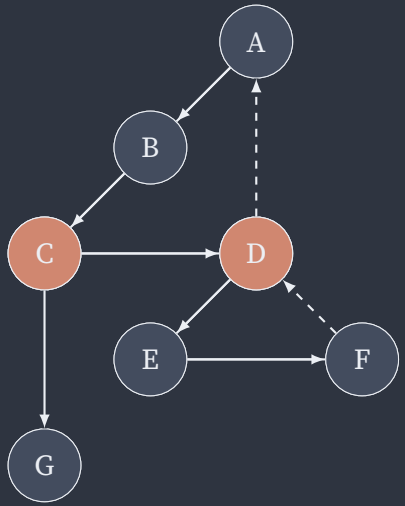
Separasjonsnoder fra DFS

- Vi gjør et dybde-først søk over en graf G, som gir opphav til et spenntre T
- Dersom vi gikk fra u til v i søket legger vi til en kant fra u til v i T
- Dersom det finnes en kant fra u til v i G og v allerede er lagt til i T kaller vi det en *tilbakekant*
- En node u er en separasjonsnode hvis:
  - u er rot i T og har mer enn én utgående kant i T, eller
  - u ikke er roten, og har et barn v, slik at ingen etterkommere av v har en tilbakekant til en forgjenger av u (ikke inkludert u selv)



Separasjonsnoder fra DFS

- Vi gjør et dybde-først søk over en graf G, som gir opphav til et spenntre T
- Dersom vi gikk fra u til v i søket legger vi til en kant fra u til v i T
- Dersom det finnes en kant fra u til v i G og v allerede er lagt til i T kaller vi det en *tilbakekant*
- En node u er en separasjonsnode hvis:
  - u er rot i T og har mer enn én utgående kant i T, eller
  - u ikke er roten, og har et barn v, slik at ingen etterkommere av v har en tilbakekant til en forgjenger av u (ikke inkludert u selv)



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

- Vi gjør et dybde-først søk

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

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

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

- Vi gjør et dybde-først søk
- For hver node u noterer vi:
  - `depth`$[u]$ gir *dybden* til u i T

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

- Vi gjør et dybde-først søk
- For hver node u noterer vi:
  - `depth`$[u]$ gir *dybden* til u i T
  - `low`$[u]$ gir den laveste dybden som kan nås ved å følge kanter i T og maksimalt én tilbakekant

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

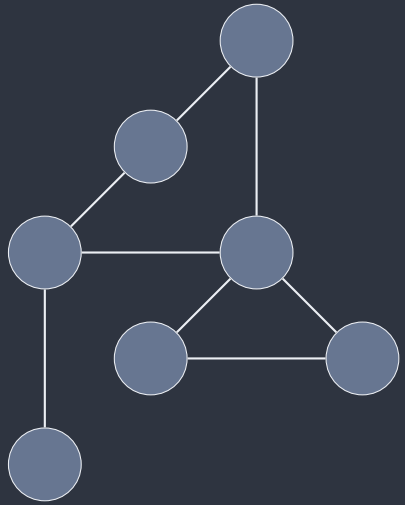
- Vi gjør et dybde-først søk
- For hver node u noterer vi:
  - `depth`$[u]$ gir *dybden* til u i T
  - `low`$[u]$ gir den laveste dybden som kan nås ved å følge kanter i T og maksimalt én tilbakekant
- Dersom `depth`$[u] \leq$ `low`$[v]$, der v er et barn av u, så er u en separasjonsnode

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

- Vi gjør et dybde-først søk
- For hver node u noterer vi:
  - `depth`$[u]$ gir *dybden* til u i T
  - `low`$[u]$ gir den laveste dybden som kan nås ved å følge kanter i T og maksimalt én tilbakekant
- Dersom `depth`$[u] \leq$ `low`$[v]$, der v er et barn av u, så er u en separasjonsnode
  - Roten vil alltid ha `depth`$[u] =$ `low`$[v] = 0$, så der sjekker vi i stedet om den har to barn i T

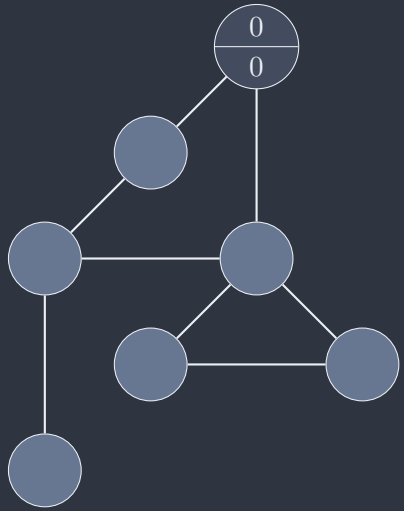
Finn separasjonsnoder fra DFS med `depth` og `low`

- Vi gjør et dybde-først søk
- For hver node u noterer vi:
  - `depth`$[u]$ gir *dybden* til u i T
  - `low`$[u]$ gir den laveste dybden som kan nås ved å følge kanter i T og maksimalt én tilbakekant
- Dersom `depth`$[u] \leq$ `low`$[v]$, der v er et barn av u, så er u en separasjonsnode
  - Roten vil alltid ha `depth`$[u] =$ `low`$[v] = 0$, så der sjekker vi i stedet om den har to barn i T



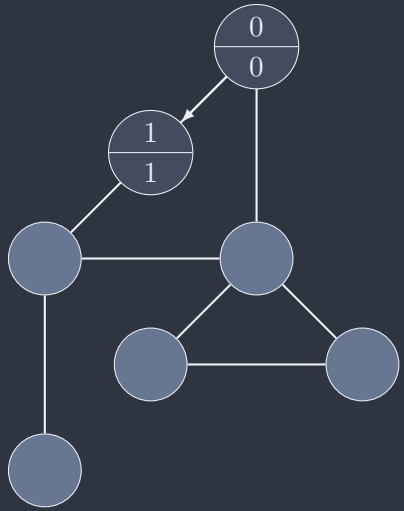
Finn separasjonsnoder fra DFS med `depth` og `low`

- Vi gjør et dybde-først søk
- For hver node u noterer vi:
  - `depth`$[u]$ gir *dybden* til u i T
  - `low`$[u]$ gir den laveste dybden som kan nås ved å følge kanter i T og maksimalt én tilbakekant
- Dersom `depth`$[u] \leq$ `low`$[v]$, der v er et barn av u, så er u en separasjonsnode
  - Roten vil alltid ha `depth`$[u] =$ `low`$[v] = 0$, så der sjekker vi i stedet om den har to barn i T



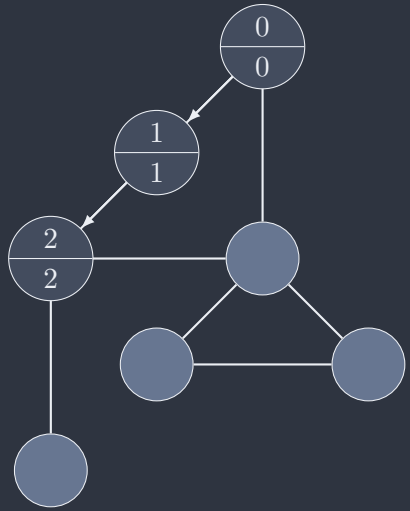
Finn separasjonsnoder fra DFS med `depth` og `low`

- Vi gjør et dybde-først søk
- For hver node u noterer vi:
  - `depth`$[u]$ gir *dybden* til u i T
  - `low`$[u]$ gir den laveste dybden som kan nås ved å følge kanter i T og maksimalt én tilbakekant
- Dersom `depth`$[u] \leq$ `low`$[v]$, der v er et barn av u, så er u en separasjonsnode
  - Roten vil alltid ha `depth`$[u] =$ `low`$[v] = 0$, så der sjekker vi i stedet om den har to barn i T



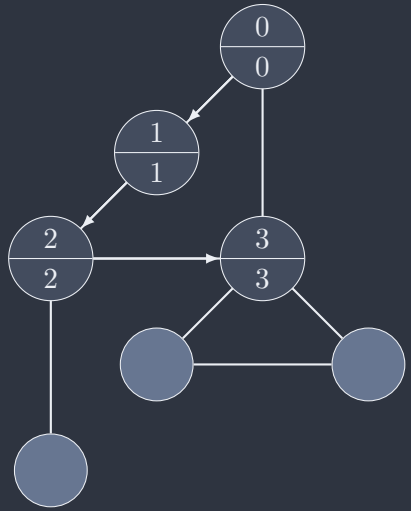
Finn separasjonsnoder fra DFS med `depth` og `low`

- Vi gjør et dybde-først søk
- For hver node u noterer vi:
  - `depth`$[u]$ gir *dybden* til u i T
  - `low`$[u]$ gir den laveste dybden som kan nås ved å følge kanter i T og maksimalt én tilbakekant
- Dersom `depth`$[u] \leq$ `low`$[v]$, der v er et barn av u, så er u en separasjonsnode
  - Roten vil alltid ha `depth`$[u] =$ `low`$[v] = 0$, så der sjekker vi i stedet om den har to barn i T



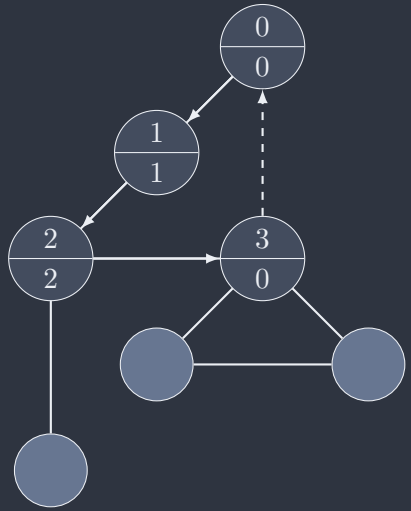
Finn separasjonsnoder fra DFS med `depth` og `low`

- Vi gjør et dybde-først søk
- For hver node u noterer vi:
  - `depth`$[u]$ gir *dybden* til u i T
  - `low`$[u]$ gir den laveste dybden som kan nås ved å følge kanter i T og maksimalt én tilbakekant
- Dersom `depth`$[u] \leq$ `low`$[v]$, der v er et barn av u, så er u en separasjonsnode
  - Roten vil alltid ha `depth`$[u] =$ `low`$[v] = 0$, så der sjekker vi i stedet om den har to barn i T



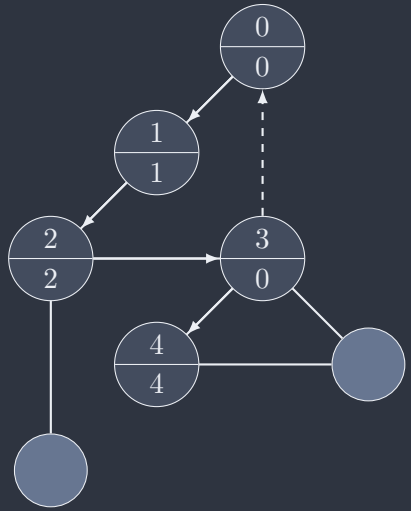
Finn separasjonsnoder fra DFS med `depth` og `low`

- Vi gjør et dybde-først søk
- For hver node u noterer vi:
  - `depth`$[u]$ gir *dybden* til u i T
  - `low`$[u]$ gir den laveste dybden som kan nås ved å følge kanter i T og maksimalt én tilbakekant
- Dersom `depth`$[u] \leq$ `low`$[v]$, der v er et barn av u, så er u en separasjonsnode
  - Roten vil alltid ha `depth`$[u] =$ `low`$[v] = 0$, så der sjekker vi i stedet om den har to barn i T



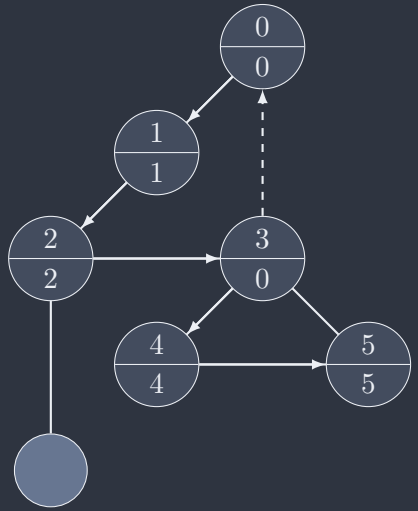
Finn separasjonsnoder fra DFS med `depth` og `low`

- Vi gjør et dybde-først søk
- For hver node u noterer vi:
  - `depth`$[u]$ gir *dybden* til u i T
  - `low`$[u]$ gir den laveste dybden som kan nås ved å følge kanter i T og maksimalt én tilbakekant
- Dersom `depth`$[u] \leq$ `low`$[v]$, der v er et barn av u, så er u en separasjonsnode
  - Roten vil alltid ha `depth`$[u] =$ `low`$[v] = 0$, så der sjekker vi i stedet om den har to barn i T



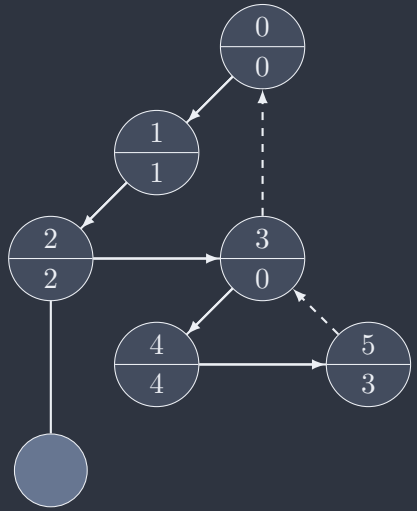
Finn separasjonsnoder fra DFS med `depth` og `low`

- Vi gjør et dybde-først søk
- For hver node u noterer vi:
  - `depth`$[u]$ gir *dybden* til u i T
  - `low`$[u]$ gir den laveste dybden som kan nås ved å følge kanter i T og maksimalt én tilbakekant
- Dersom `depth`$[u] \leq$ `low`$[v]$, der v er et barn av u, så er u en separasjonsnode
  - Roten vil alltid ha `depth`$[u] =$ `low`$[v] = 0$, så der sjekker vi i stedet om den har to barn i T



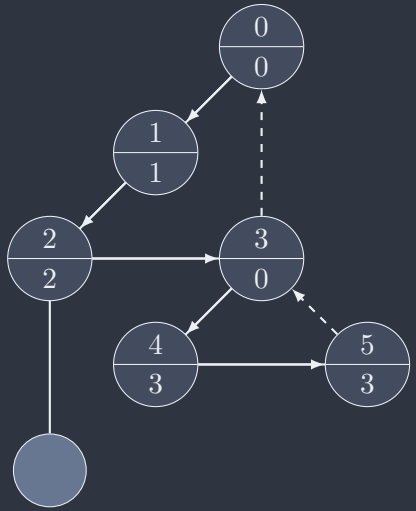
Finn separasjonsnoder fra DFS med `depth` og `low`

- Vi gjør et dybde-først søk
- For hver node u noterer vi:
  - `depth`$[u]$ gir *dybden* til u i T
  - `low`$[u]$ gir den laveste dybden som kan nås ved å følge kanter i T og maksimalt én tilbakekant
- Dersom `depth`$[u] \leq$ `low`$[v]$, der v er et barn av u, så er u en separasjonsnode
  - Roten vil alltid ha `depth`$[u] =$ `low`$[v] = 0$, så der sjekker vi i stedet om den har to barn i T



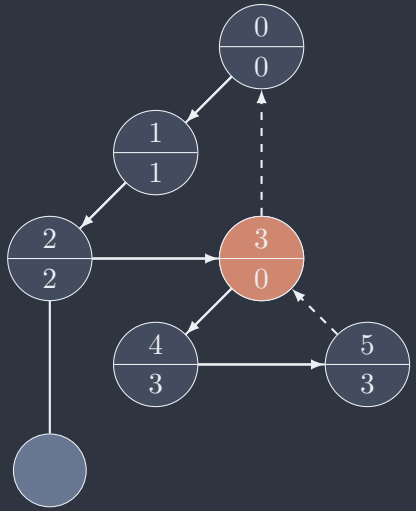
Finn separasjonsnoder fra DFS med `depth` og `low`

- Vi gjør et dybde-først søk
- For hver node u noterer vi:
  - `depth`$[u]$ gir *dybden* til u i T
  - `low`$[u]$ gir den laveste dybden som kan nås ved å følge kanter i T og maksimalt én tilbakekant
- Dersom `depth`$[u] \leq$ `low`$[v]$, der v er et barn av u, så er u en separasjonsnode
  - Roten vil alltid ha `depth`$[u] =$ `low`$[v] = 0$, så der sjekker vi i stedet om den har to barn i T



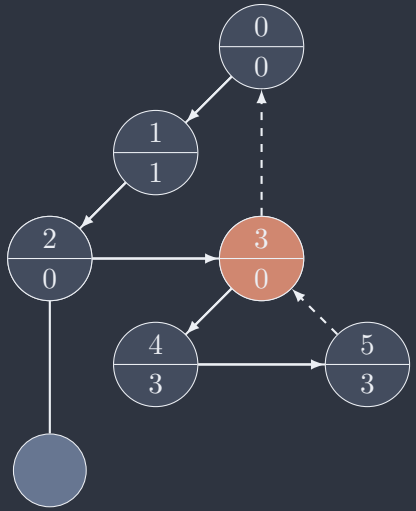
Finn separasjonsnoder fra DFS med `depth` og `low`

- Vi gjør et dybde-først søk
- For hver node u noterer vi:
  - `depth`$[u]$ gir *dybden* til u i T
  - `low`$[u]$ gir den laveste dybden som kan nås ved å følge kanter i T og maksimalt én tilbakekant
- Dersom `depth`$[u] \leq$ `low`$[v]$, der v er et barn av u, så er u en separasjonsnode
  - Roten vil alltid ha `depth`$[u] =$ `low`$[v] = 0$, så der sjekker vi i stedet om den har to barn i T



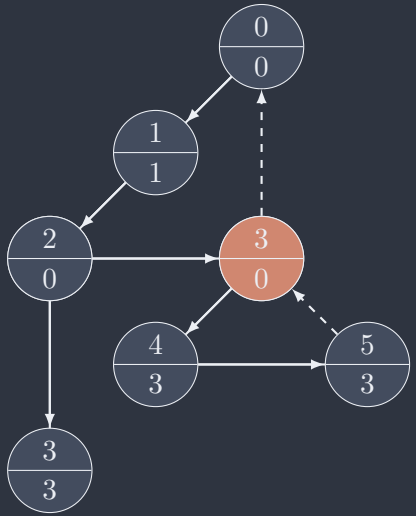
Finn separasjonsnoder fra DFS med `depth` og `low`

- Vi gjør et dybde-først søk
- For hver node u noterer vi:
  - `depth`$[u]$ gir *dybden* til u i T
  - `low`$[u]$ gir den laveste dybden som kan nås ved å følge kanter i T og maksimalt én tilbakekant
- Dersom `depth`$[u] \leq$ `low`$[v]$, der v er et barn av u, så er u en separasjonsnode
  - Roten vil alltid ha `depth`$[u] =$ `low`$[v] = 0$, så der sjekker vi i stedet om den har to barn i T



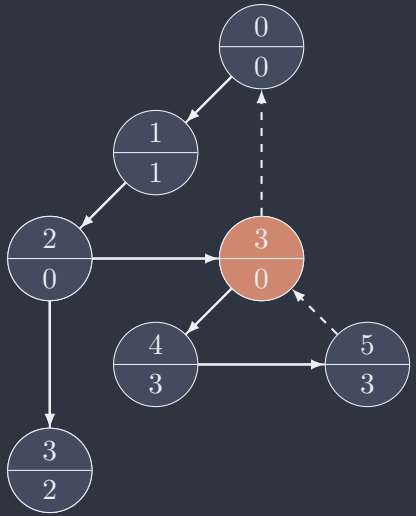
Finn separasjonsnoder fra DFS med `depth` og `low`

- Vi gjør et dybde-først søk
- For hver node u noterer vi:
  - `depth`$[u]$ gir *dybden* til u i T
  - `low`$[u]$ gir den laveste dybden som kan nås ved å følge kanter i T og maksimalt én tilbakekant
- Dersom `depth`$[u] \leq$ `low`$[v]$, der v er et barn av u, så er u en separasjonsnode
  - Roten vil alltid ha `depth`$[u] =$ `low`$[v] = 0$, så der sjekker vi i stedet om den har to barn i T



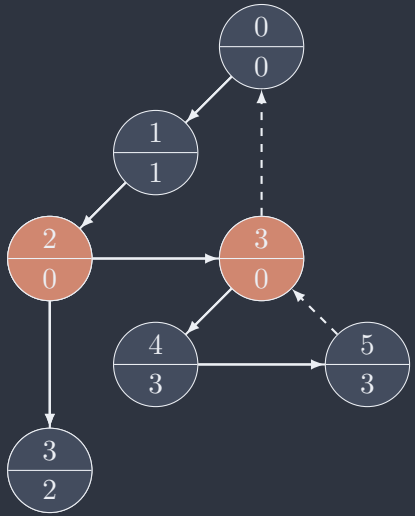
Finn separasjonsnoder fra DFS med `depth` og `low`

- Vi gjør et dybde-først søk
- For hver node u noterer vi:
  - `depth`$[u]$ gir *dybden* til u i T
  - `low`$[u]$ gir den laveste dybden som kan nås ved å følge kanter i T og maksimalt én tilbakekant
- Dersom `depth`$[u] \leq$ `low`$[v]$, der v er et barn av u, så er u en separasjonsnode
  - Roten vil alltid ha `depth`$[u] =$ `low`$[v] = 0$, så der sjekker vi i stedet om den har to barn i T



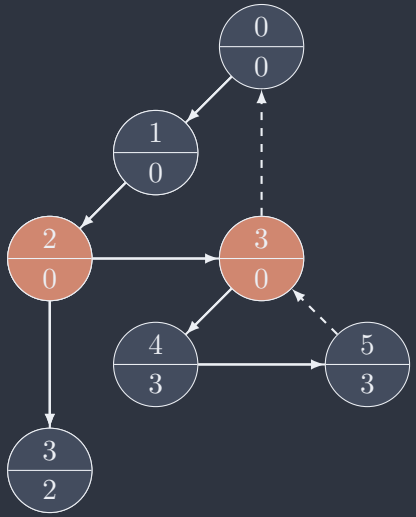
Finn separasjonsnoder fra DFS med `depth` og `low`

- Vi gjør et dybde-først søk
- For hver node u noterer vi:
  - `depth`$[u]$ gir *dybden* til u i T
  - `low`$[u]$ gir den laveste dybden som kan nås ved å følge kanter i T og maksimalt én tilbakekant
- Dersom `depth`$[u] \leq$ `low`$[v]$, der v er et barn av u, så er u en separasjonsnode
  - Roten vil alltid ha `depth`$[u] =$ `low`$[v] = 0$, så der sjekker vi i stedet om den har to barn i T



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

- Vi gjør et dybde-først søk
- For hver node u noterer vi:
  - `depth`$[u]$ gir *dybden* til u i T
  - `low`$[u]$ gir den laveste dybden som kan nås ved å følge kanter i T og maksimalt én tilbakekant
- Dersom `depth`$[u] \leq$ `low`$[v]$, der v er et barn av u, så er u en separasjonsnode
  - Roten vil alltid ha `depth`$[u] =$ `low`$[v] = 0$, så der sjekker vi i stedet om den har to barn i T



Finn separasjonsnoder

Algorithm: Finn alle separasjonsnoder i en sammenhengende graf

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

```
1 depth ← empty map
2 low ← empty map
3 seps ← empty set
4 Procedure SeparationVertices(G)
5     r ← choose arbitrary vertex from V
6     depth[r] ← 0
7     Visit(G, r)
```

Finn separasjonsnoder

Algorithm: Finn alle separasjonsnoder i en sammenhengende graf

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

```
1  depth ← empty map
2  low ← empty map
3  seps ← empty set
4  Procedure SeparationVertices(G)
5      r ← choose arbitrary vertex from V
6      depth[r] ← 0
7      Visit(G, r)
8
9      children ← 0
10     for u ∈ V do
11         if depth[u] = 1 then
12             children ← children + 1
13     if children > 1 then
14         add r to seps
15
16     return seps
```

Finn separasjonsnoder

Algorithm: Finn alle separasjonsnoder i en sammenhengende graf

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

```
1  depth ← empty map
2  low ← empty map
3  seps ← empty set
4  Procedure SeparationVertices(G)
5      r ← choose arbitrary vertex from V
6      depth[r] ← 0
7      Visit(G, r)
8
9      children ← 0
10     for u ∈ V do
11         if depth[u] = 1 then
12             children ← children + 1
13     if children > 1 then
14         add r to seps
15
16     return seps
17 Procedure Visit(G, u)
18     low[u] ← depth[u]
```

Finn separasjonsnoder

ALGORITHM: FINN ALLE SEPARASJONSNODER I EN SAMMENHENGENDE GRAF

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

```
1  depth ← empty map
2  low ← empty map
3  seps ← empty set
4  Procedure SeparationVertices(G)
5      r ← choose arbitrary vertex from V
6      depth[r] ← 0
7      Visit(G, r)
8
9      children ← 0
10     for u ∈ V do
11         if depth[u] = 1 then
12             children ← children + 1
13     if children > 1 then
14         add r to seps
15
16     return seps
17 Procedure Visit(G, u)
18     low[u] ← depth[u]
19
20     for (u, v) ∈ E do
21         if v ∈ depth then
22             low[u] ← Min(low[u], depth[v])
```

Finn separasjonsnoder

Algorithm: Finn alle separasjonsnoder i en sammenhengende graf

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

```
1  depth ← empty map
2  low ← empty map
3  seps ← empty set
4  Procedure SeparationVertices(G)
5      r ← choose arbitrary vertex from V
6      depth[r] ← 0
7      Visit(G, r)
8
9      children ← 0
10     for u ∈ V do
11         if depth[u] = 1 then
12             children ← children + 1
13     if children > 1 then
14         add r to seps
15
16     return seps
17 Procedure Visit(G, u)
18     low[u] ← depth[u]
19
20     for (u, v) ∈ E do
21         if v ∈ depth then
22             low[u] ← Min(low[u], depth[v])
23         else
24             depth[v] ← depth[u] + 1
25             Visit(G, v)
26             low[u] ← Min(low[u], low[v])
```

Finn separasjonsnoder

Algorithm: Finn alle separasjonsnoder i en sammenhengende graf

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

```
1  depth ← empty map
2  low ← empty map
3  seps ← empty set
4  Procedure SeparationVertices(G)
5      r ← choose arbitrary vertex from V
6      depth[r] ← 0
7      Visit(G, r)
8
9      children ← 0
10     for u ∈ V do
11         if depth[u] = 1 then
12             children ← children + 1
13     if children > 1 then
14         add r to seps
15
16     return seps
17 Procedure Visit(G, u)
18     low[u] ← depth[u]
19
20     for (u, v) ∈ E do
21         if v ∈ depth then
22             low[u] ← Min(low[u], depth[v])
23         else
24             depth[v] ← depth[u] + 1
25             Visit(G, v)
26             low[u] ← Min(low[u], low[v])
27             if depth[u] ≠ 0 and depth[u] ≤ low[v] then
28                 add u to seps
```

Er G 2-sammenhengende?

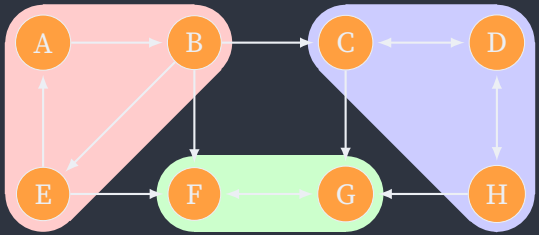
Algorithm: Er grafen 2-sammenhengende?

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
```

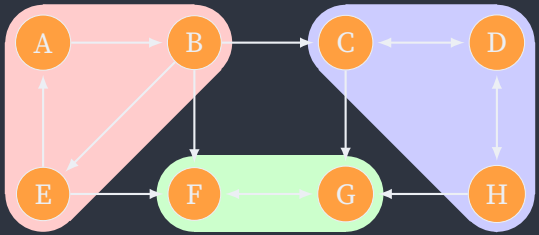
Sterkt sammenhengende komponenter

Sterkt sammenhengende komponenter



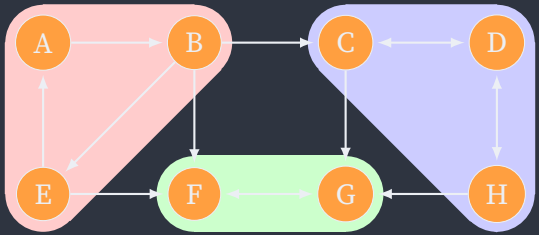
- En *rettet* graf er *sterkt* sammenhengende dersom det finnes en sti mellom alle par av noder

Sterkt sammenhengende komponenter



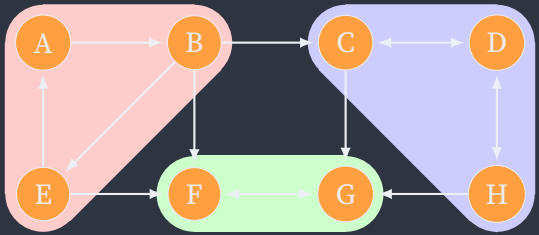
- En *rettet* graf er *sterkt* sammenhengende dersom det finnes en sti mellom alle par av noder
- En sterkt sammenhengende komponent er en delgraf slik at

Sterkt sammenhengende komponenter



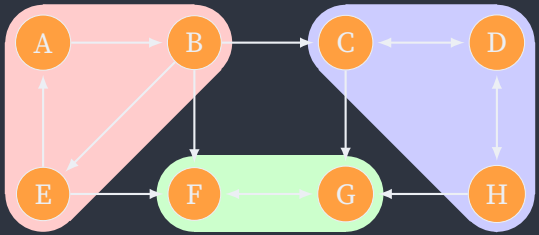
- En *rettet* graf er *sterkt* sammenhengende dersom det finnes en sti mellom alle par av noder
- En sterkt sammenhengende komponent er en delgraf slik at
  - Det finnes en sti mellom alle par av noder i komponenten

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*

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

Den reverserte grafen

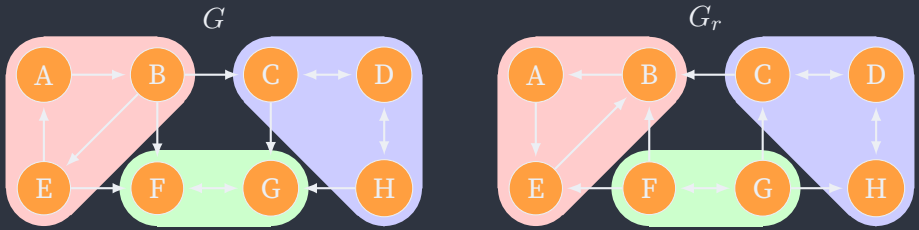
- Vi sier at den *reverserte* grafen til G er den grafen hvor alle kanter er snudd
  - Hvis $G = (V, E)$ sier vi at $G_r = (V, E_r)$ er den reverserte grafen

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 $(v, u) \in E_r$

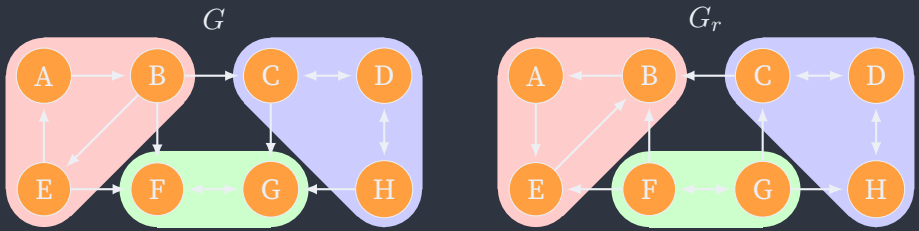
Den reverserte grafen

- 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 $(v, u) \in E_r$



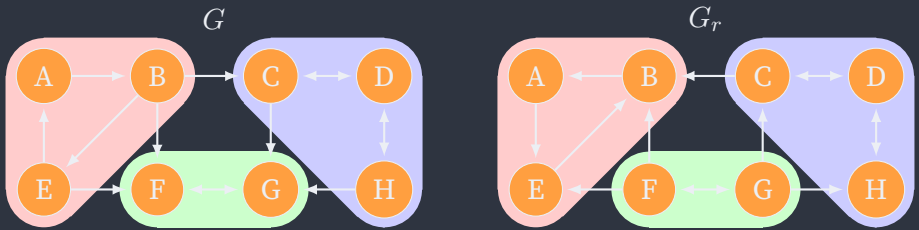
Den reverserte grafen

- 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

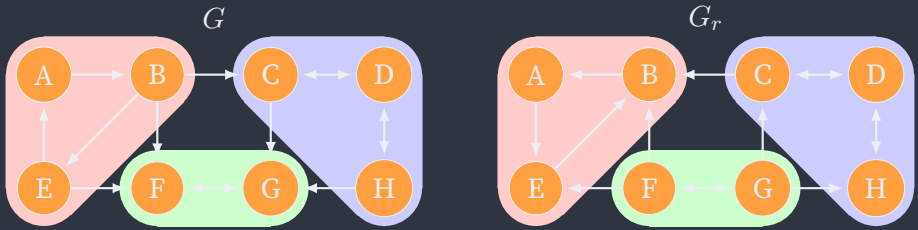


ALGORITHM: REVERSER EN GITT GRAF

Input: En graf $G = (V, E)$
Output: Returner den reverserte grafen $G_r = (V, E_r)$

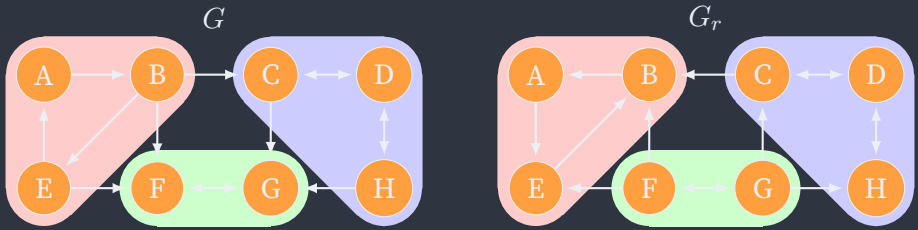
```
1 Procedure ReverseGraph(G)
2     E_r ← empty set
3     for (u, v) ∈ E do
4         add (v, u) to E_r
5     return (V, E_r)
```

Finne sterkt sammenhengende komponenter



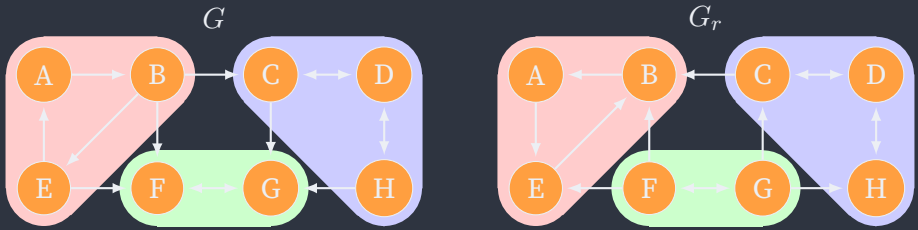
- Vi kan finne den sterkt sammenhengende komponenten til en node v ved å

Finne sterkt sammenhengende komponenter



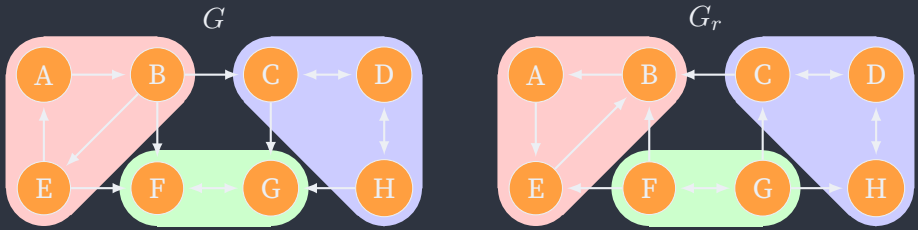
- Vi kan finne den sterkt sammenhengende komponenten til en node v ved å
  - Finne alle noder som kan nås fra v i G

Finne sterkt sammenhengende komponenter



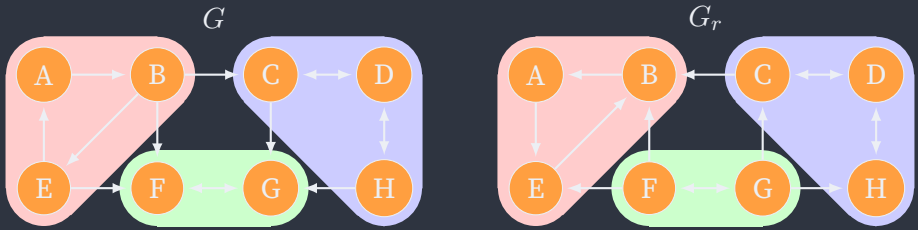
- Vi kan finne den sterkt sammenhengende komponenten til en node v ved å
  - Finne alle noder som kan nås fra v i G
  - Finne alle noder som kan nås fra v i G_r

Finne sterkt sammenhengende komponenter



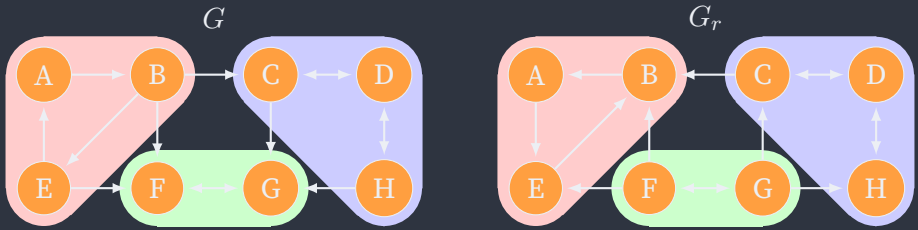
- Vi kan finne den sterkt sammenhengende komponenten til en node v ved å
  - Finne alle noder som kan nås fra v i G
  - Finne alle noder som kan nås fra v i G_r
  - Nodene som kunne nås i både G og G_r utgjør den sterkt sammenhengende komponenten til v

Finne sterkt sammenhengende komponenter



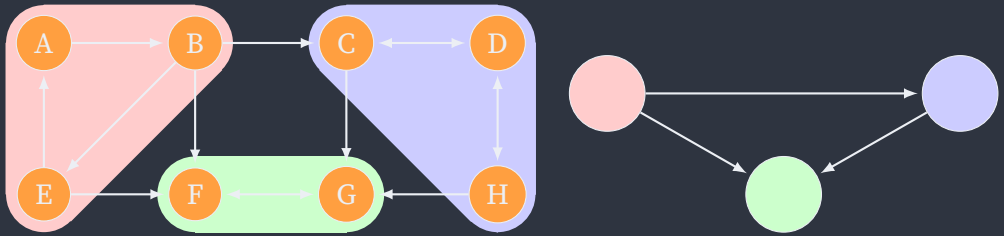
- Vi kan finne den sterkt sammenhengende komponenten til en node v ved å
  - Finne alle noder som kan nås fra v i G
  - Finne alle noder som kan nås fra v i G_r
  - Nodene som kunne nås i både G og G_r utgjør den sterkt sammenhengende komponenten til v
- Dette er en algoritme i $\mathcal{O}(|V| \cdot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

Finne sterkt sammenhengende komponenter



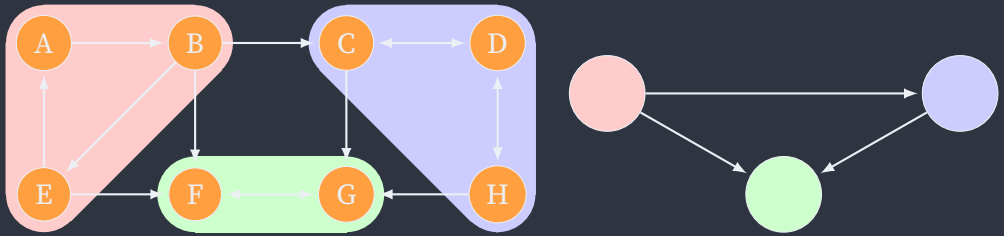
- Vi kan finne den sterkt sammenhengende komponenten til en node v ved å
  - Finne alle noder som kan nås fra v i G
  - Finne alle noder som kan nås fra v i G_r
  - Nodene som kunne nås i både G og G_r utgjør den sterkt sammenhengende komponenten til v
- Dette er en algoritme i $\mathcal{O}(|V| \cdot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

Komponentgrafen



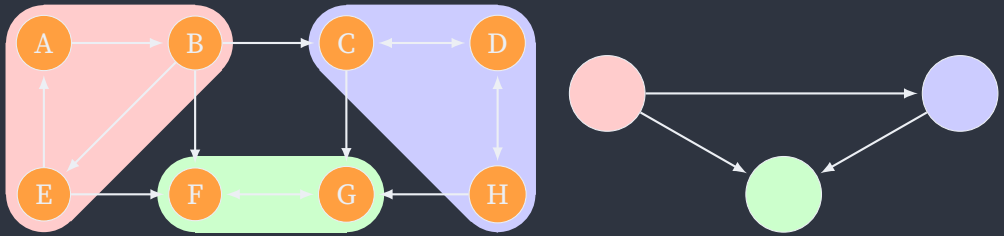
- Ved å anse hver sterkt sammenhengende komponent som en enkelt node, så får vi det vi kaller for *komponentgrafen*

Komponentgrafen



- 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*)

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

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*

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

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

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:

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`)

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

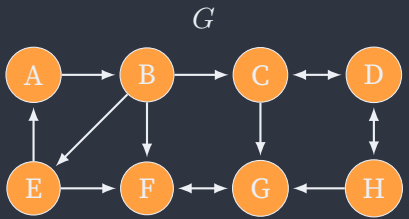
Kosaraju

- 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 t av stacken fra det første søket

Kosaraju

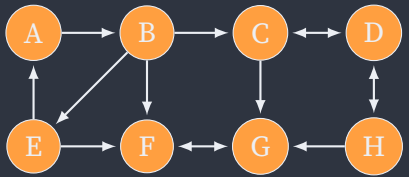
- Algoritmen kan i korte trekk beskrives slik:
  1. Gjør et (fullt) dybde-først søk i en graf G, der hver node legges på en stack etter alle naboer er besøkt (`DFSTopSort`)
  2. Konstruer den reverserte grafen G_r
  3. Gjør et nytt (fullt) dybde-først søk på G_r, der det rekkefølgen i det fulle dybde-først søket er t av stacken fra det første søket
- To dybde-først søk gir kjøretidskompleksitet $\mathcal{O}(|V| + |E|)$

Kosaraju (eksempel)

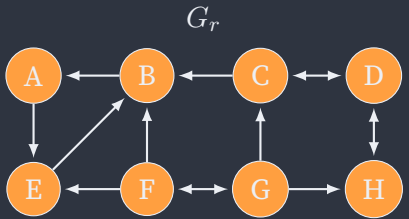


Kosaraju (eksempel)

G

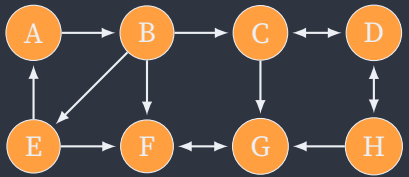


G_r

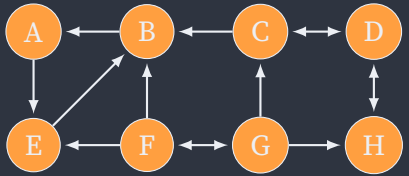


Kosaraju (eksempel)

G



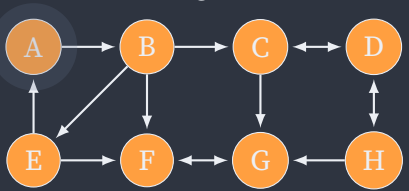
G_r



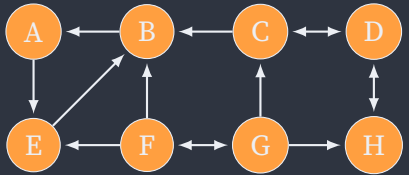
`stack`

Kosaraju (eksempel)

G

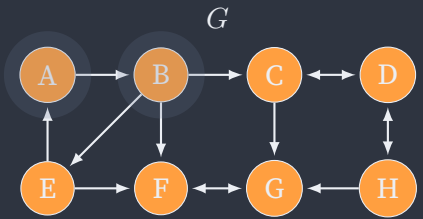


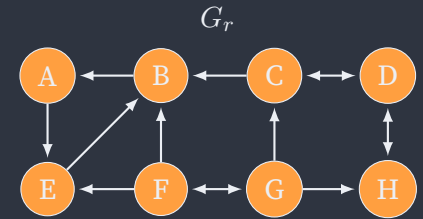
G_r



stack

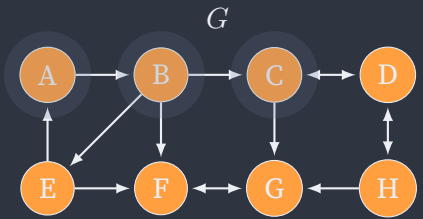
Kosaraju (eksempel)

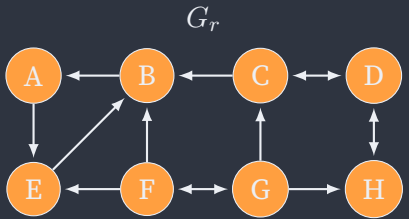




stack

Kosaraju (eksempel)

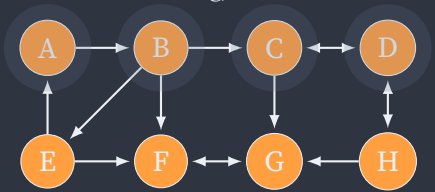




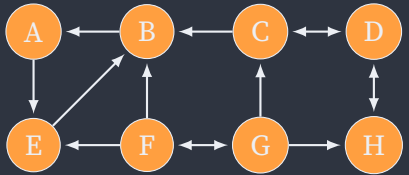
`stack`

Kosaraju (eksempel)

G



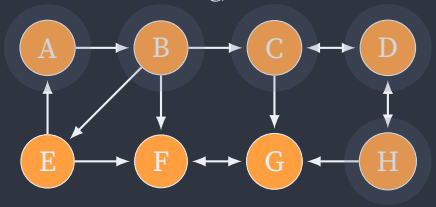
G_r



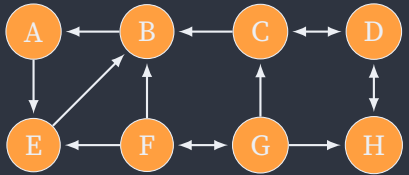
`stack`

Kosaraju (eksempel)

G



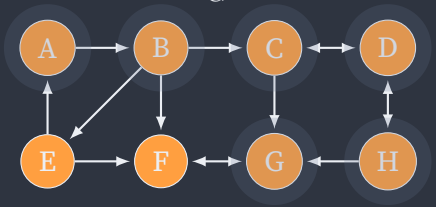
G_r



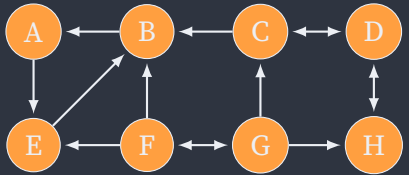
stack

Kosaraju (eksempel)

G



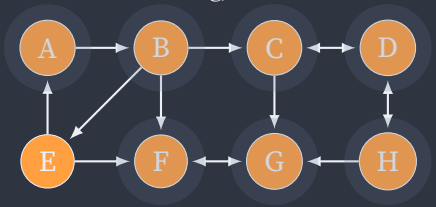
G_r



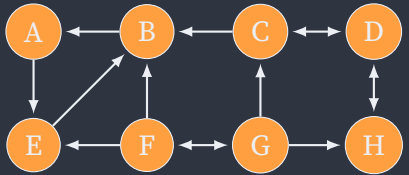
`stack`

Kosaraju (eksempel)

G



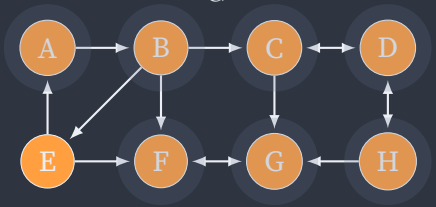
G_r



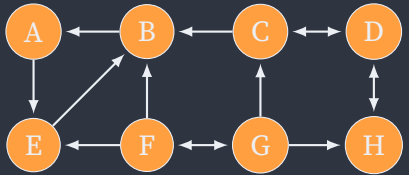
`stack`

Kosaraju (eksempel)

G



G_r

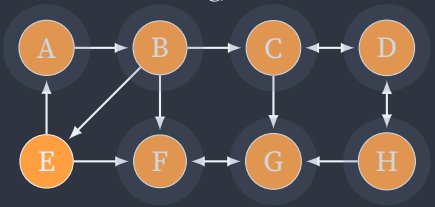


F

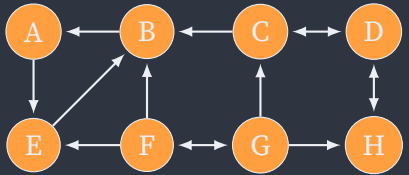
stack

Kosaraju (eksempel)

G



G_r



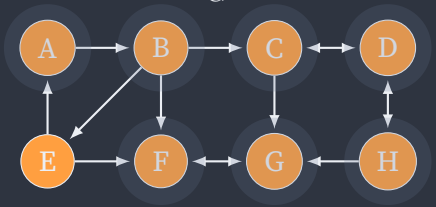
G

F

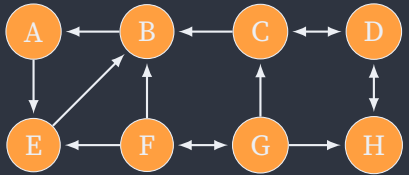
stack

Kosaraju (eksempel)

G



G_r

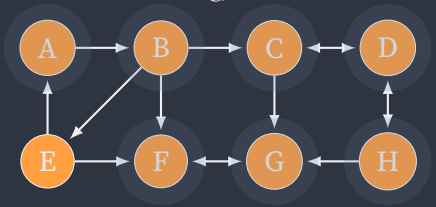


H
G
F

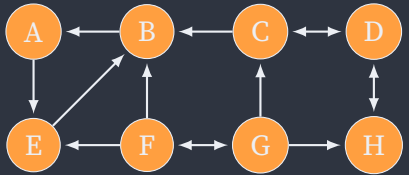
stack

Kosaraju (eksempel)

G



G_r

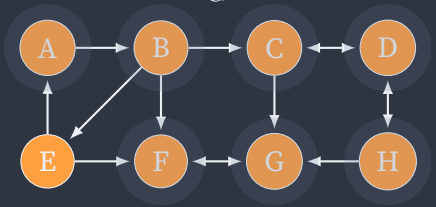


D
H
G
F

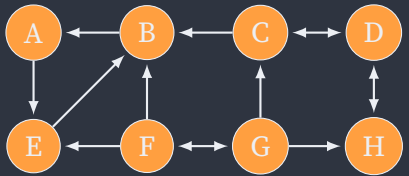
stack

Kosaraju (eksempel)

G



G_r

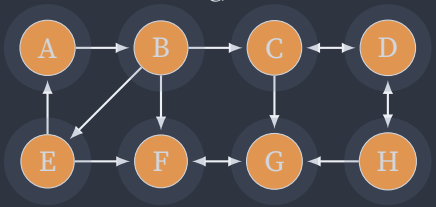


C
D
H
G
F

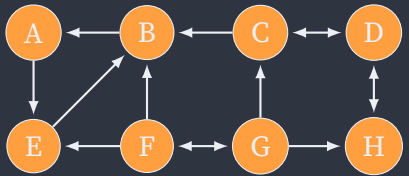
stack

Kosaraju (eksempel)

G



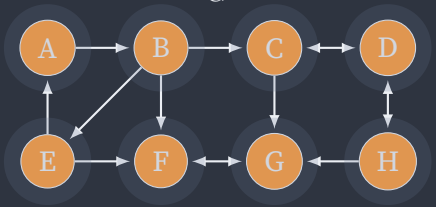
G_r



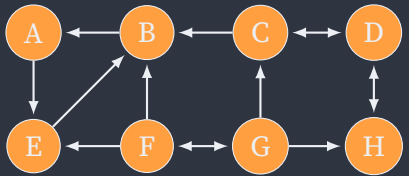
C
D
H
G
F
stack

Kosaraju (eksempel)

G



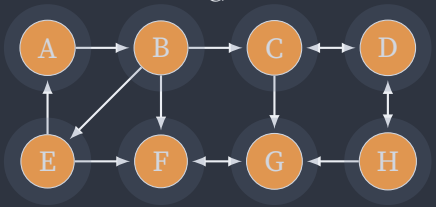
G_r



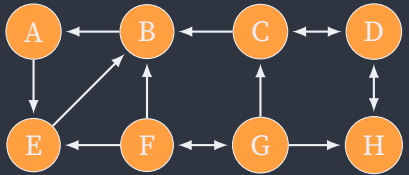
E
C
D
H
G
F
stack

Kosaraju (eksempel)

G



G_r

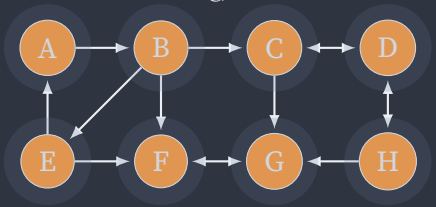


B
E
C
D
H
G
F

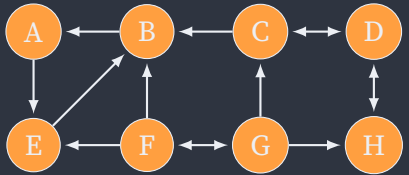
stack

Kosaraju (eksempel)

G



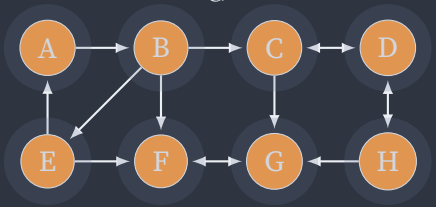
G_r



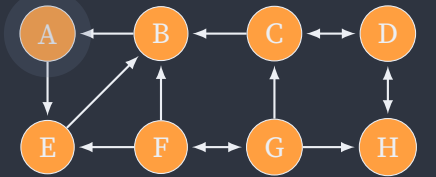
A
B
E
C
D
H
G
F
stack

Kosaraju (eksempel)

G

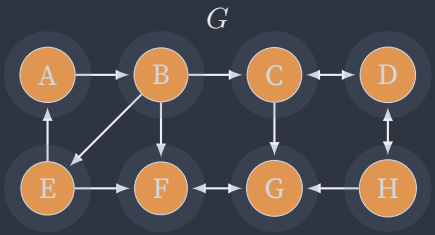


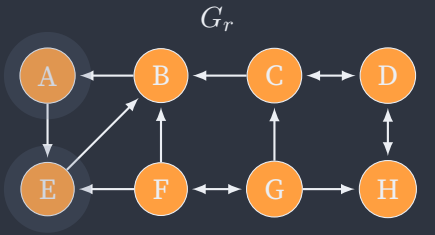
G_r



A
B
E
C
D
H
G
F
stack

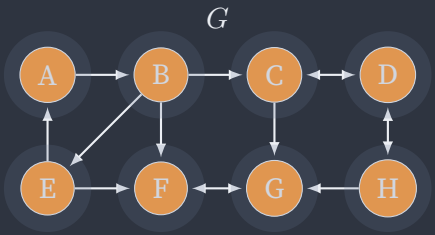
Kosaraju (eksempel)

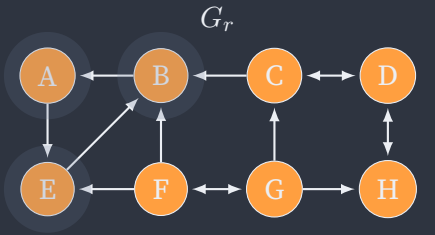




A
B
E
C
D
H
G
F
stack

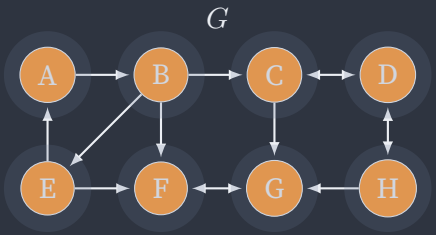
Kosaraju (eksempel)

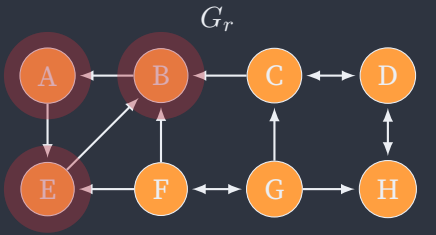




A
B
E
C
D
H
G
F
stack

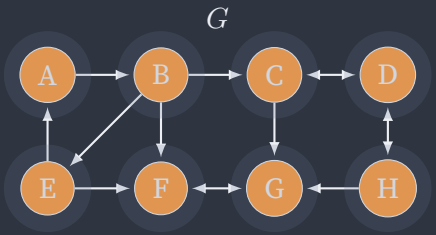
Kosaraju (eksempel)

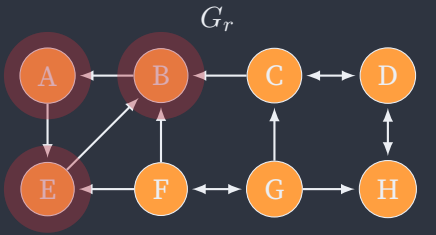




A
B
E
C
D
H
G
F
stack

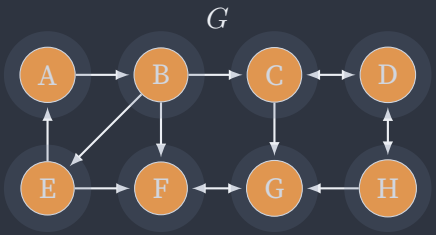
Kosaraju (eksempel)

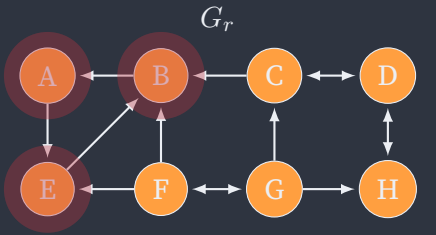




A
B
E
C
D
H
G
F
stack

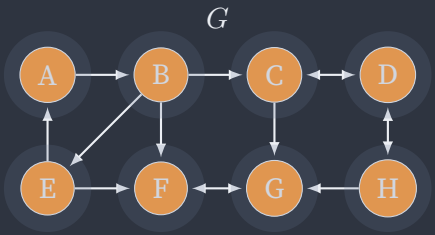
Kosaraju (eksempel)

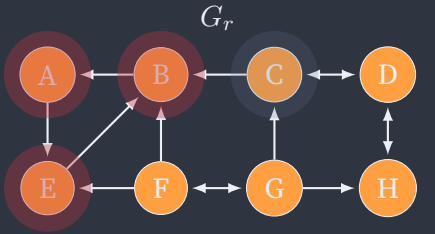




A
B
E
C
D
H
G
F
stack

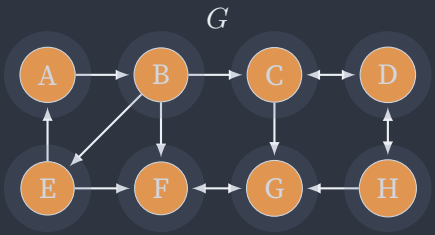
Kosaraju (eksempel)

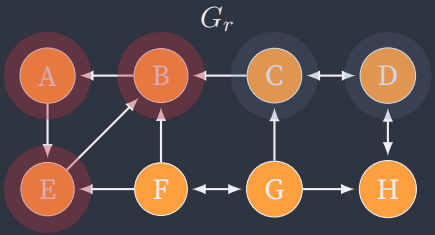




A
B
E
C
D
H
G
F
stack

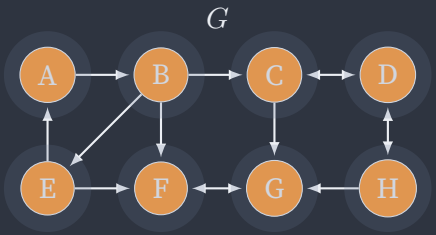
Kosaraju (eksempel)

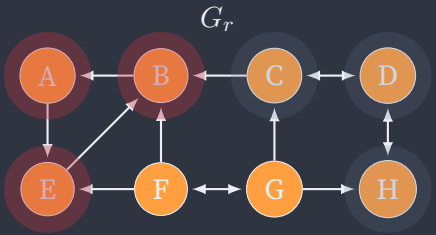




A
B
E
C
D
H
G
F
stack

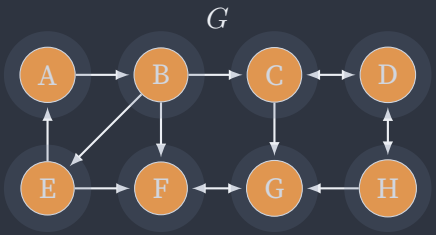
Kosaraju (eksempel)

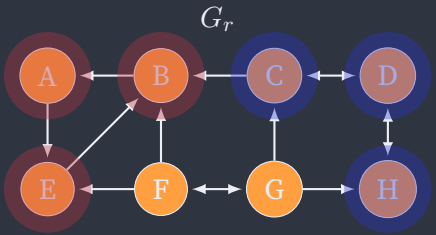




A
B
E
C
D
H
G
F
stack

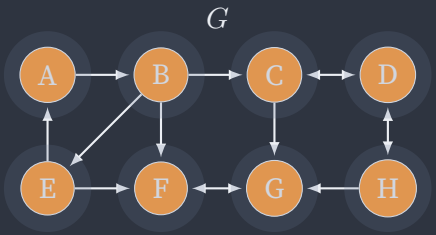
Kosaraju (eksempel)

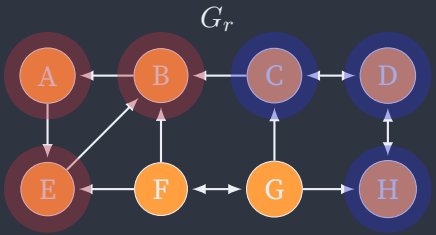




A
B
E
C
D
H
G
F
stack

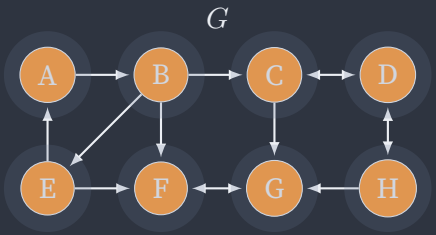
Kosaraju (eksempel)

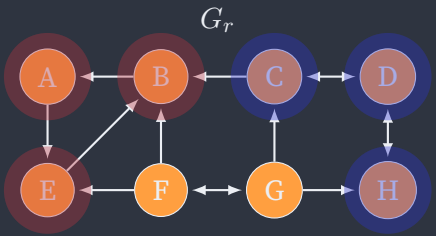




A
B
E
C
D
H
G
F
stack

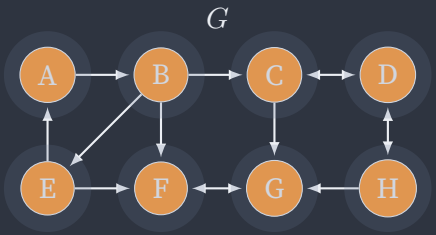
Kosaraju (eksempel)

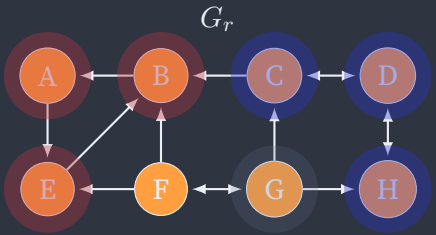




A
B
E
C
D
H
G
F
stack

Kosaraju (eksempel)

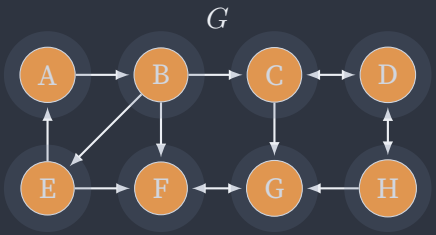


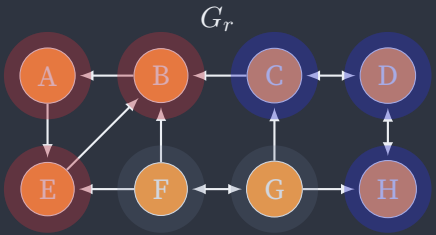


A
B
E
C
D
H
G
F

stack

Kosaraju (eksempel)

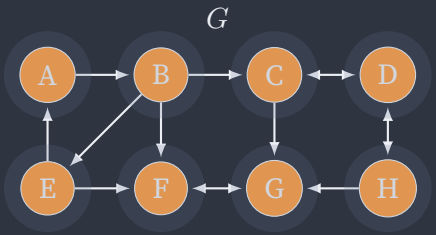


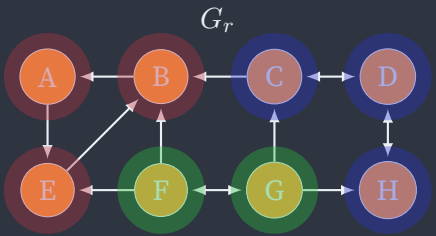


A
B
E
C
D
H
G
F

stack

Kosaraju (eksempel)

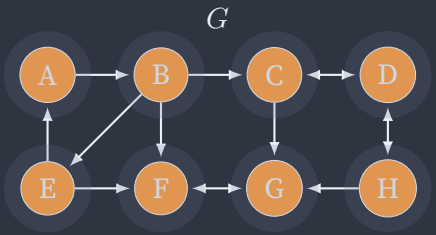


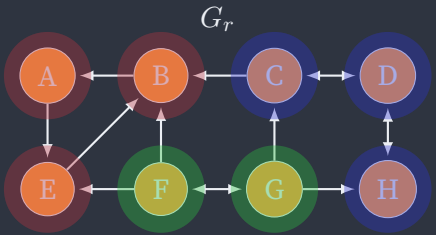


A
B
E
C
D
H
G
F

stack

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)
```

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
```

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
```

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
```