IN2010

Uvekta grafer

Terminologi

Parallelle kanter

Rettet

Enkel graf

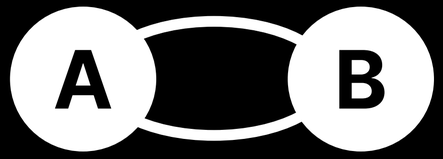
(Enkle) Løkker

Vektet

Terminologi

Parallelle kanter

Flere enn 1 kant mellom 2 noder



(Enkle) Løkker

Rettet

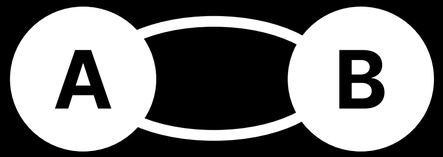
Vektet

Enkel graf

Terminologi

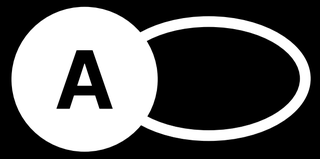
Parallelle kanter

Flere enn 1 kant mellom 2 noder



(Enkle) Løkker

En kant fra en node til seg selv



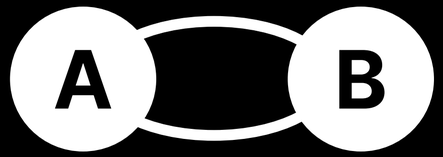
Rettet

Vektet

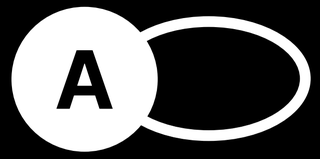
Enkel graf

Terminologi

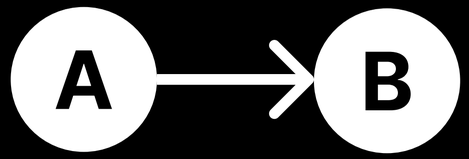
Parallelle kanter
Flere enn 1 kant mellom 2 noder



(Enkle) Løkker
En kant fra en node til seg selv



Rettet
Kantene i grafen har en retning

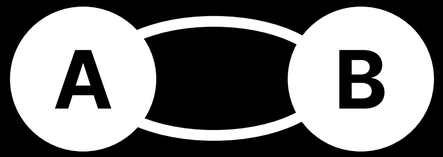


Vektet

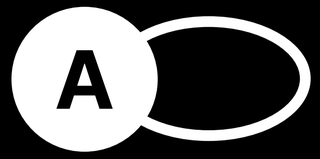
Enkel graf

Terminologi

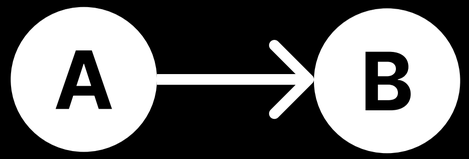
Parallelle kanter
Flere enn 1 kant mellom 2 noder



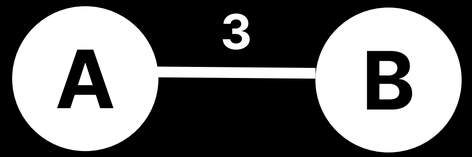
(Enkle) Løkker
En kant fra en node til seg selv



Rettet
Kantene i grafen har en retning



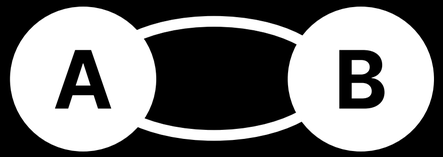
Vektet
Kantene i grafen har en verdi



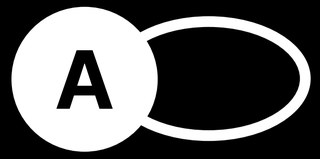
Enkel graf

Terminologi

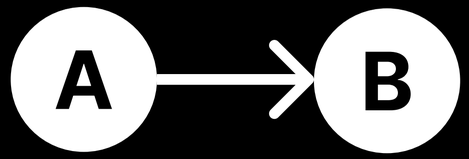
Parallelle kanter
Flere enn 1 kant mellom 2 noder



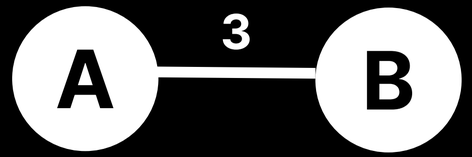
(Enkle) Løkker
En kant fra en node til seg selv



Rettet
Kantene i grafen har en retning



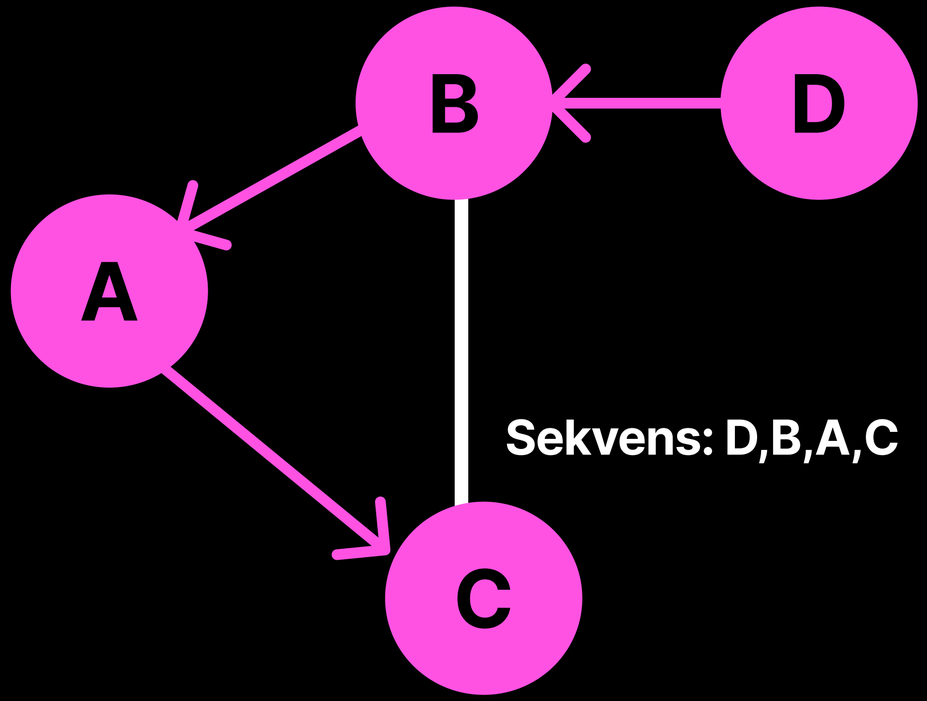
Vektet
Kantene i grafen har en verdi



Enkel graf
En graf uten løkker, parallelle kanter, retning eller vekt

Sti vs. vei

Sti
Sekvens av noder, forbundet av kanter, der ingen **noder** gjentas

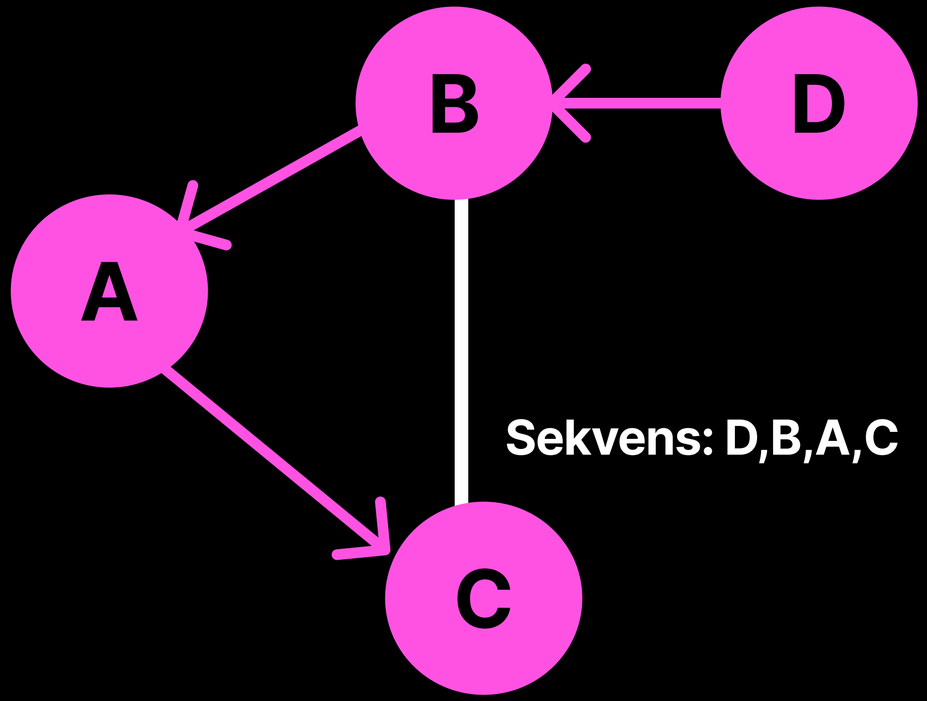


Hva er en vei?

Sti vs. vei

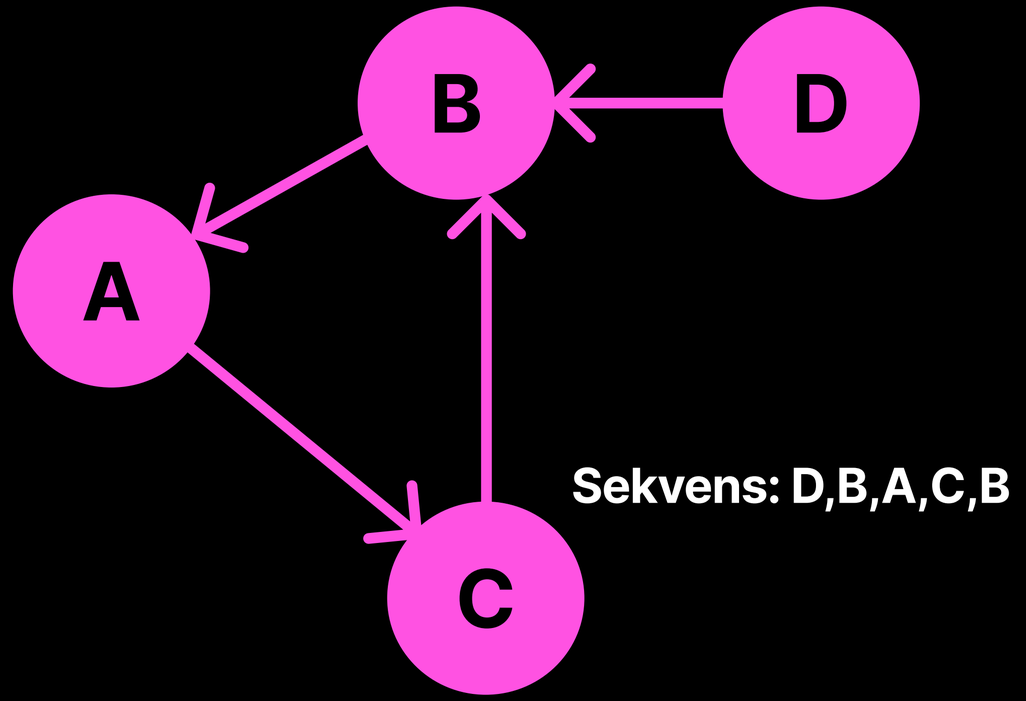
Sti

Sekvens av noder, forbundet av kanter, der ingen **noder** gjentas



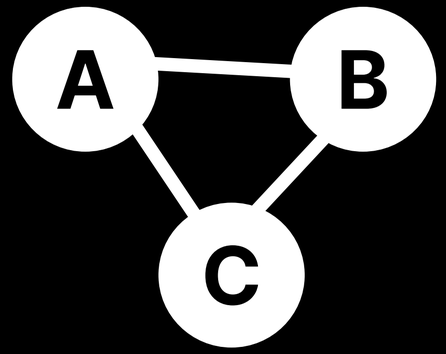
Vei

Sekvens av noder, forbundet av kanter, der ingen **kanter** gjentas



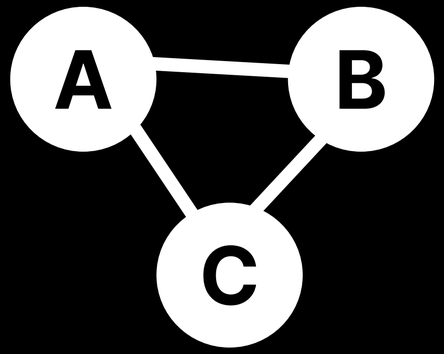
Sykler

En sti med minst tra noder som forbinder første og siste node kalles en sykel

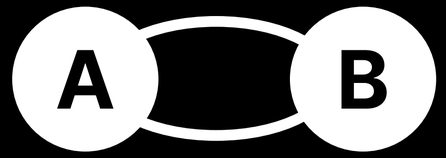


Sykler

En sti med minst tra noder som forbinder første og siste node kalles en sykel



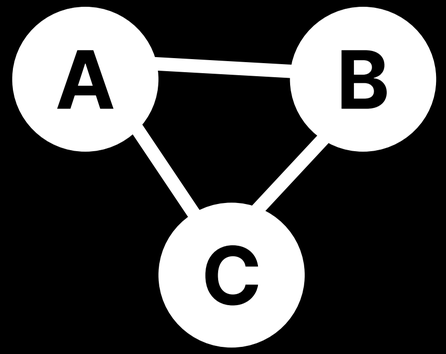
Hvorfor er ikke dette en sykel??



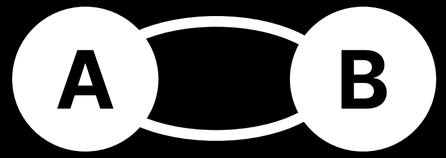
Teoretisk sett er det en sykel, men mange definisjoner av en sykel krever minst tre noder for å kvalifisere som en "ekte" sykel.
I IN2010 er denne "sykelen" kjent som parallelle kanter

Sykler

En sti med minst tra noder som forbinder første og siste node kalles en sykel

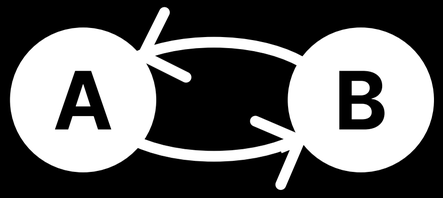


Hvorfor er ikke dette en sykel??



Teoretisk sett er det en sykel, men mange definisjoner av en sykel krever minst tre noder for å kvalifisere som en "ekte" sykel.
I IN2010 er denne "sykelen" kjent som parallelle kanter

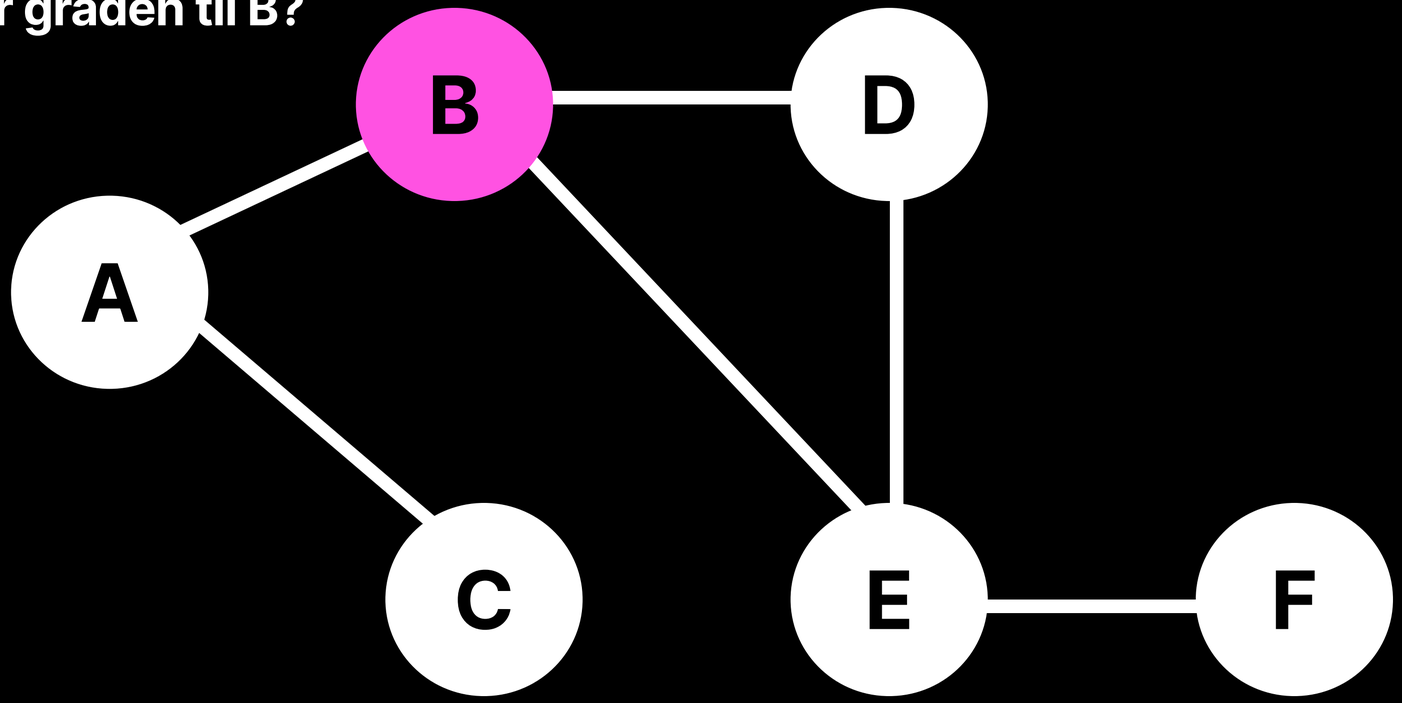
Sykler i en rettet graf



Kantene som utgjør en sykel må peke i riktig retning.
Kravet om minst tre noder gjelder ikke for rettede grafer.

Graden til en node

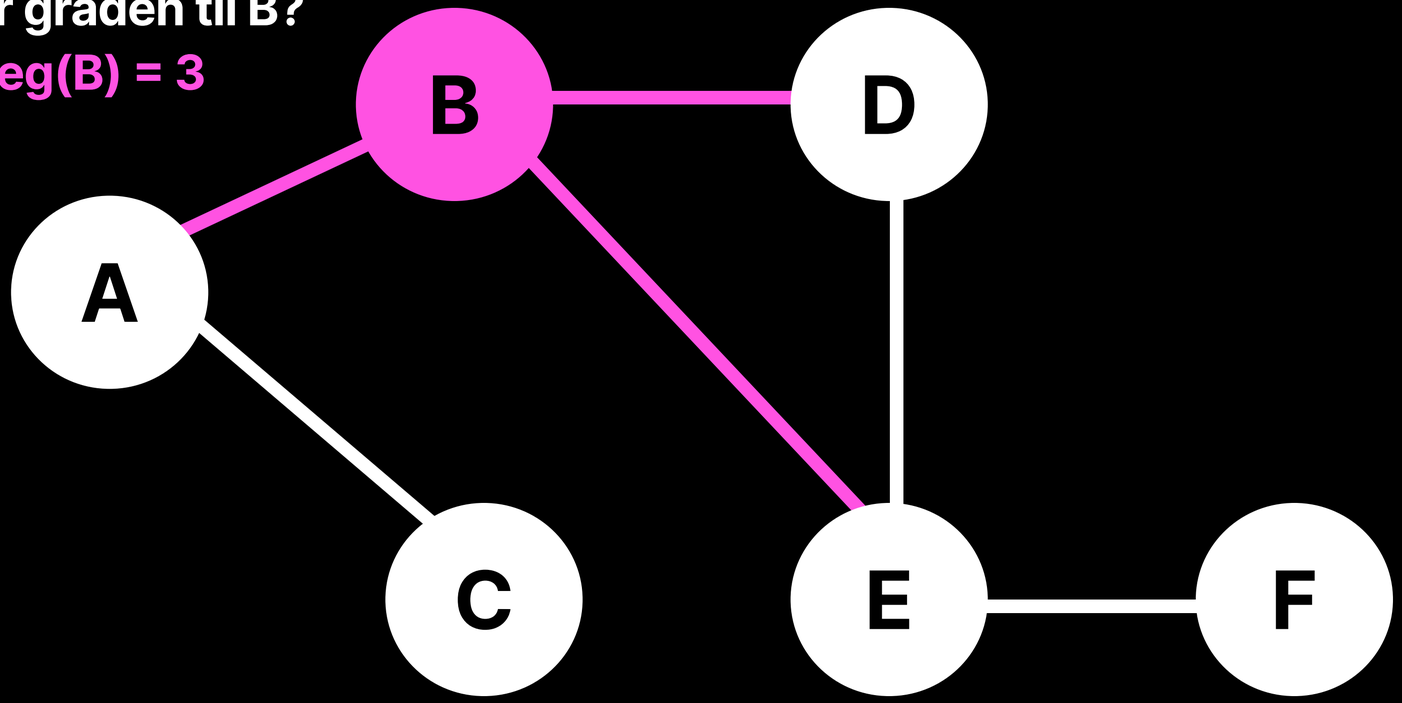
Hva er graden til B?



Graden til en node

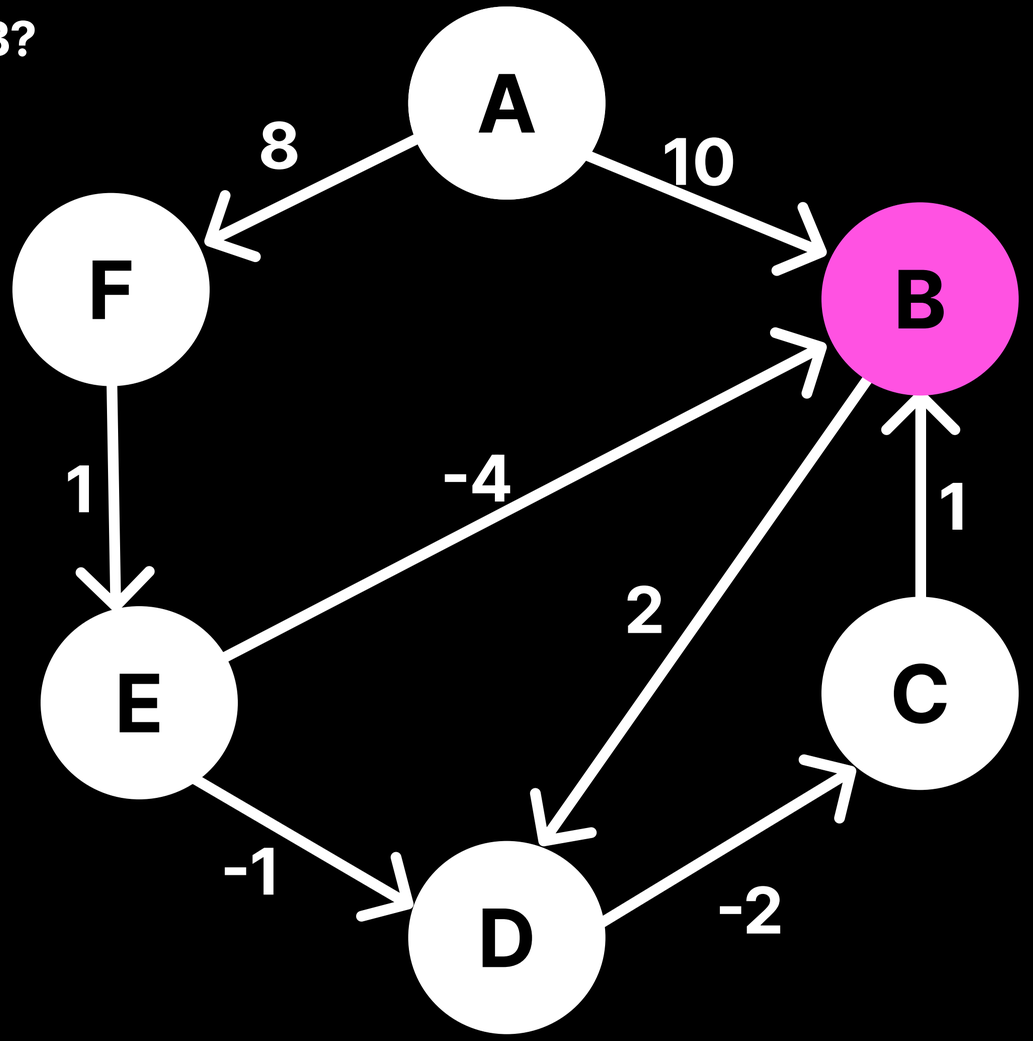
Hva er graden til B?

deg(B) = 3



Graden til en node

Hva er graden til B?

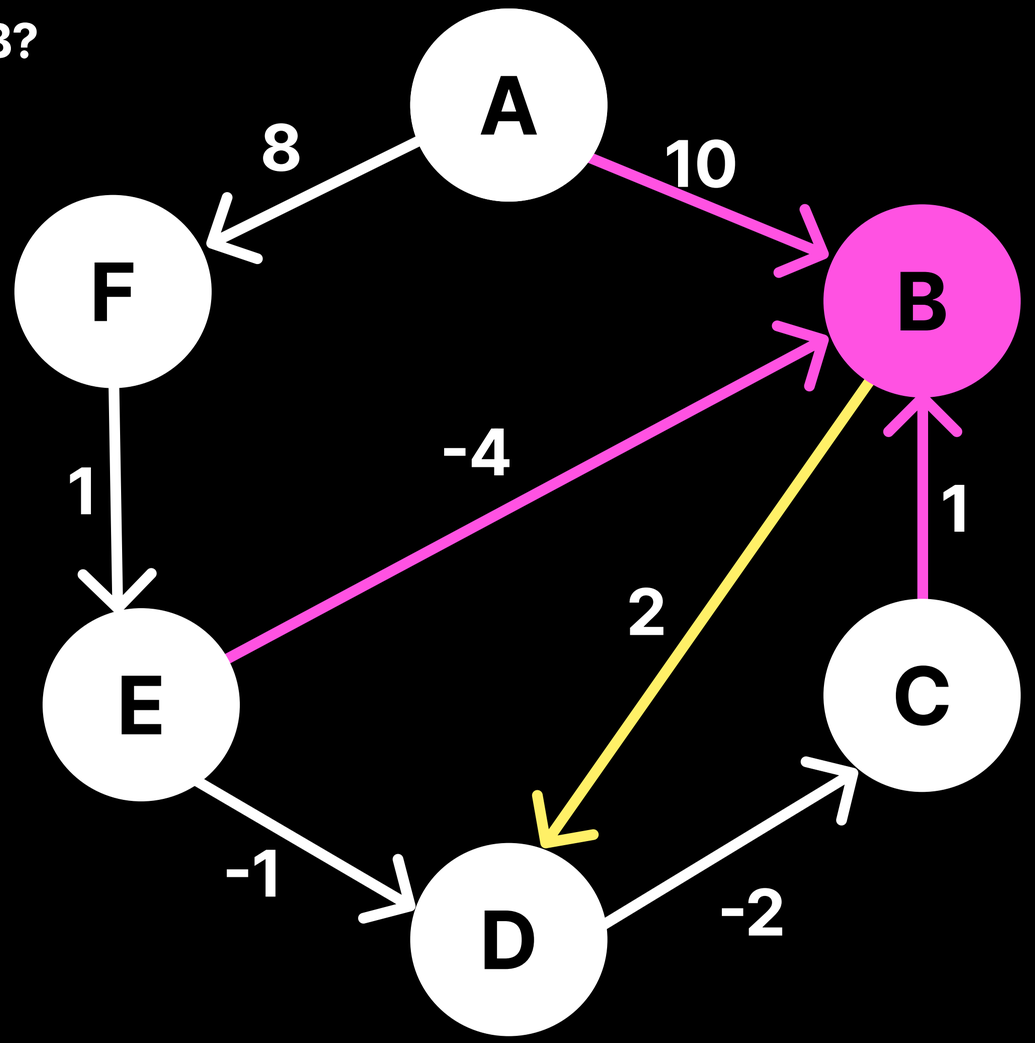


Graden til en node

Hva er graden til B?

inngrad: 3

utgrad: 1

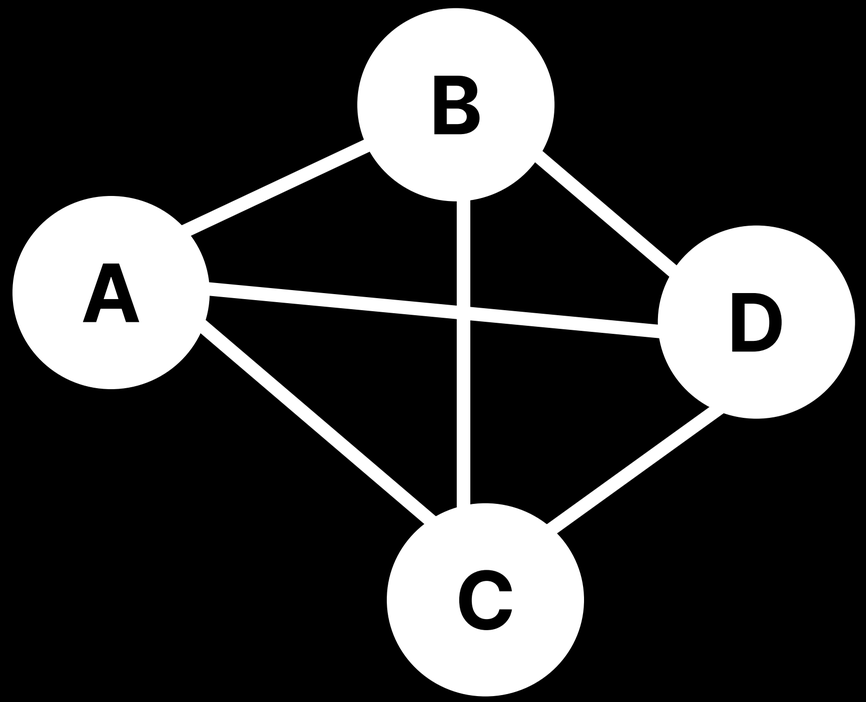


Størrelsen på en enkel komplett graf

En graf der alle noder er koblet til hverandre.

Antall noder: $|V|$

Antall kanter: $|E|$

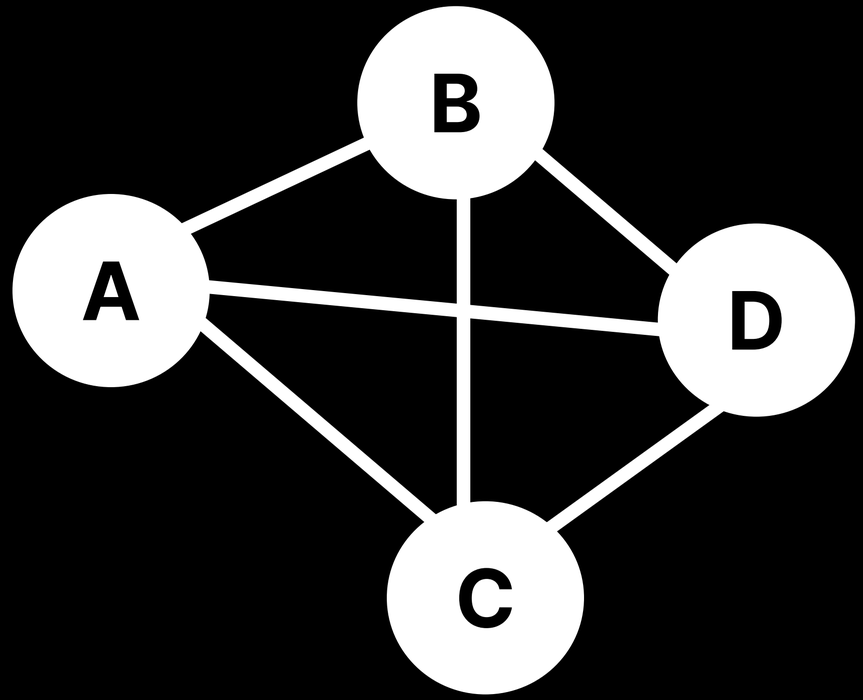


Størrelsen på en enkel komplett graf

En graf der alle noder er koblet til hverandre.

Antall noder: $|V| = 4$

Antall kanter: $|E|$

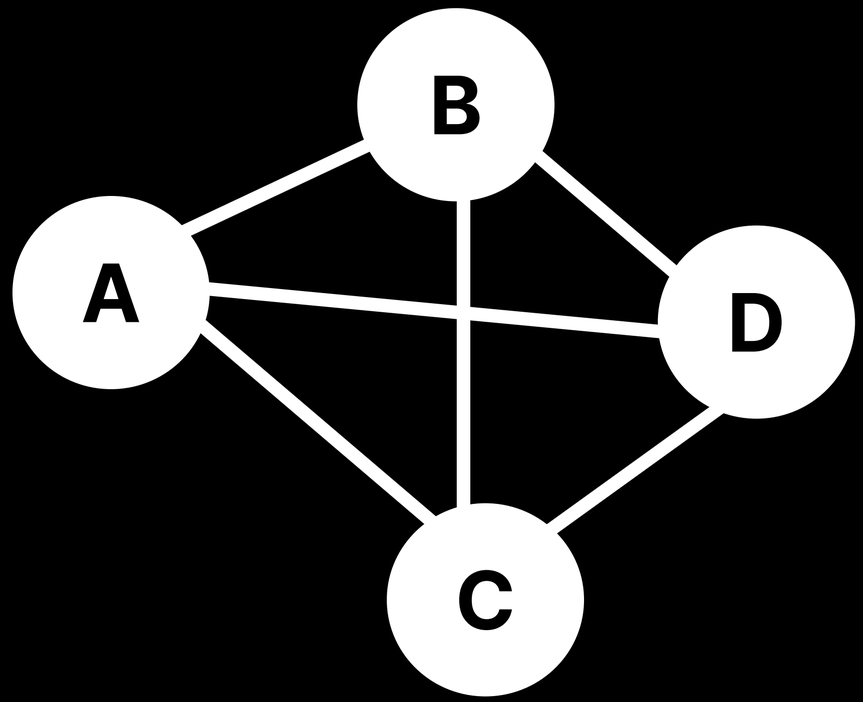


Størrelsen på en enkel komplett graf

En graf der alle noder er koblet til hverandre.

Antall noder: $|V| = 4$

Antall kanter: $|E| = |V|*(|V|-1) / 2$

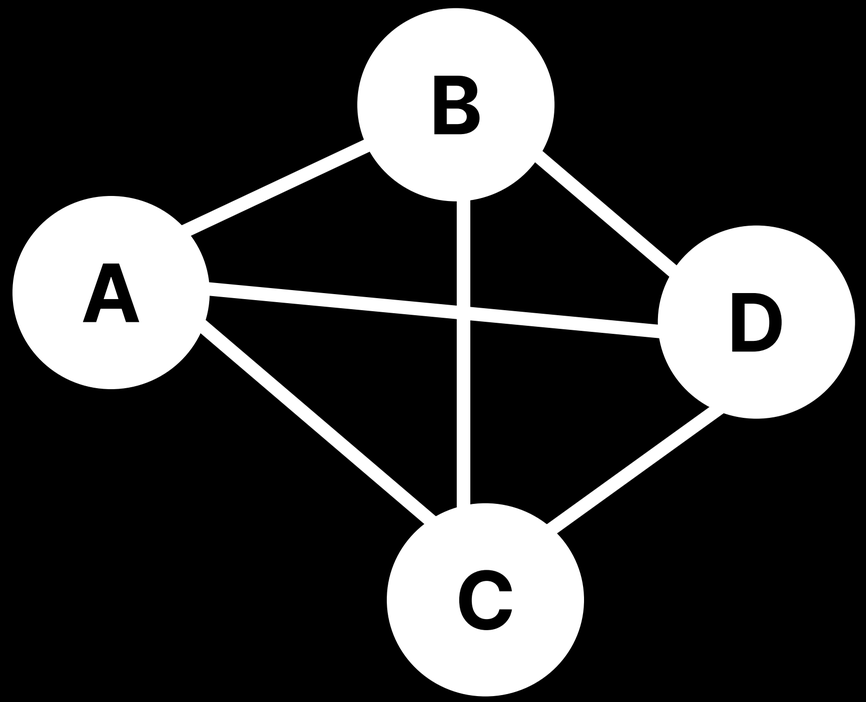


Størrelsen på en enkel komplett graf

En graf der alle noder er koblet til hverandre.

Antall noder: $|V| = 4$

Antall kanter: $|E| = |V|*(|V|-1) / 2 = 4*3 / 2$

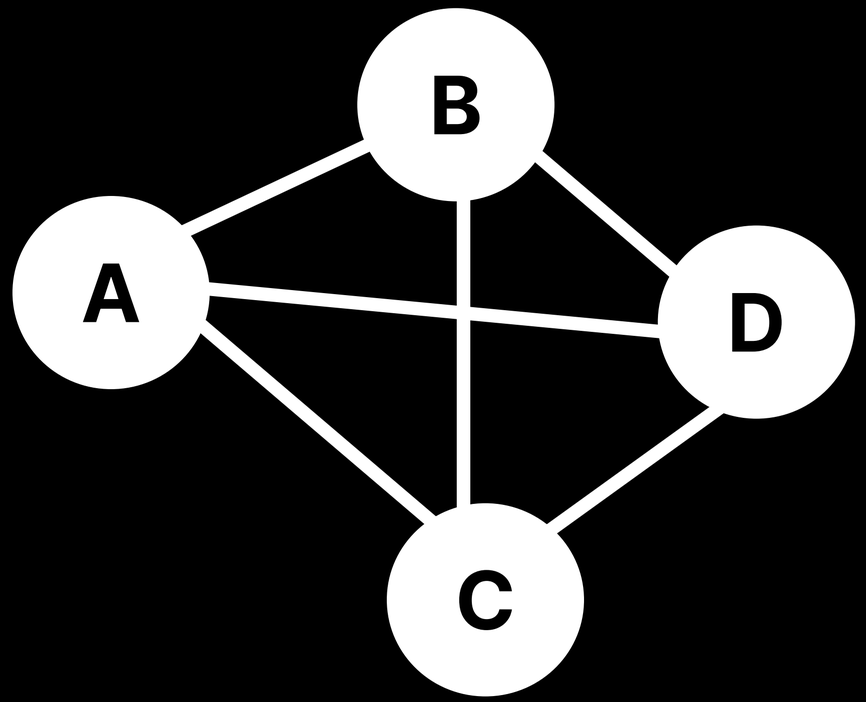


Størrelsen på en enkel komplett graf

En graf der alle noder er koblet til hverandre.

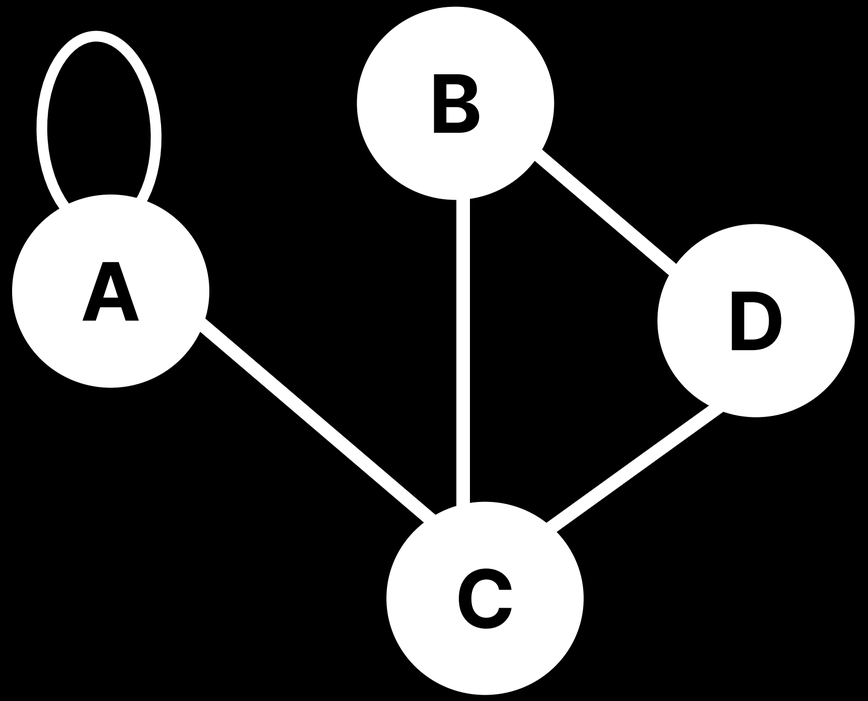
Antall noder: $|V| = 4$

Antall kanter: $|E| = |V|*(|V|-1) / 2 = 4*3 / 2 = 6$



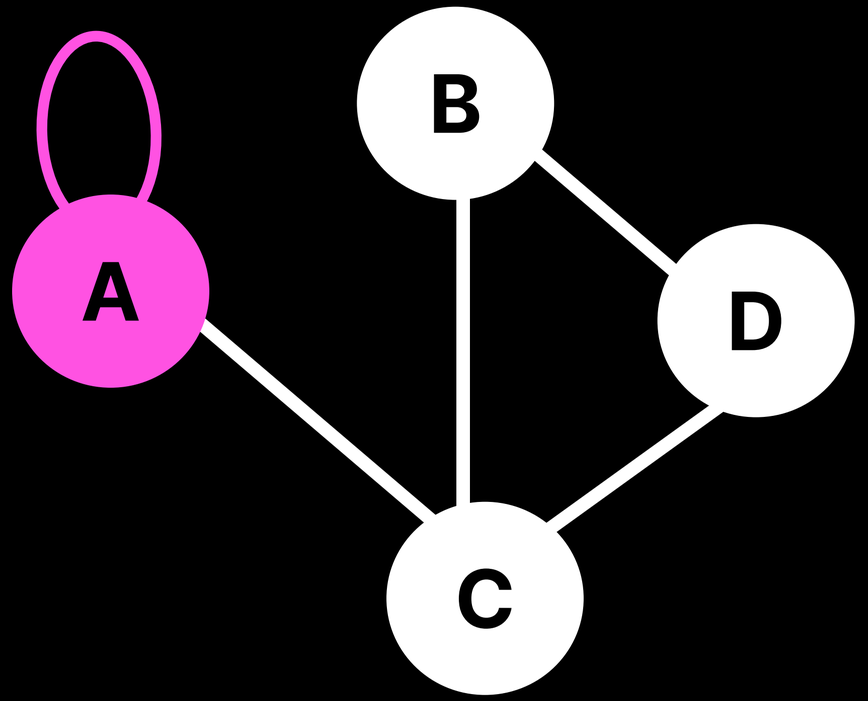
Nabomatrise

	A	B	C	D
A				
B				
C				
D				



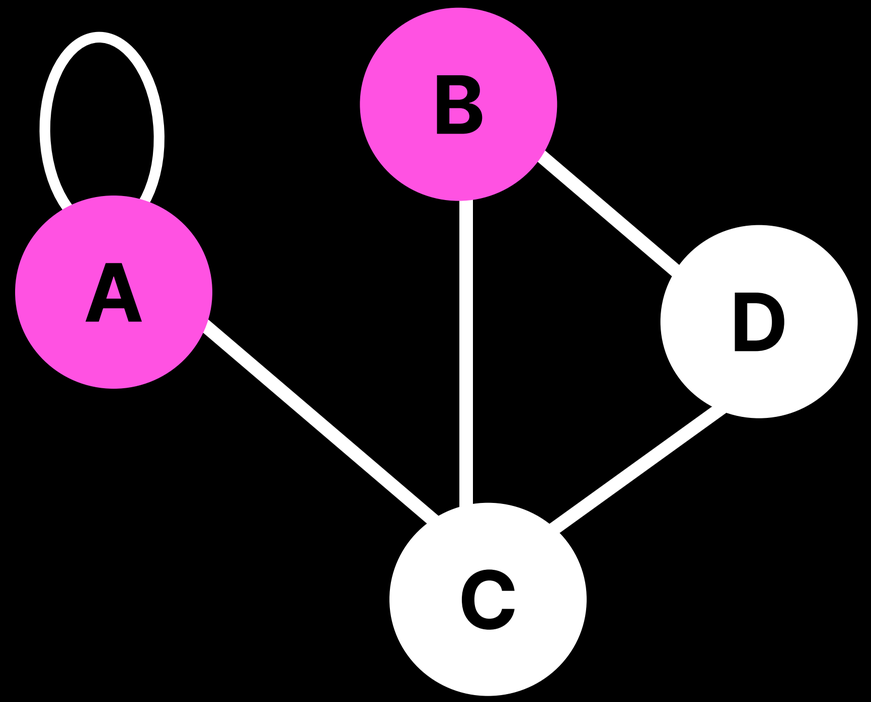
Nabomatrise

	A	B	C	D
A	1			
B				
C				
D				



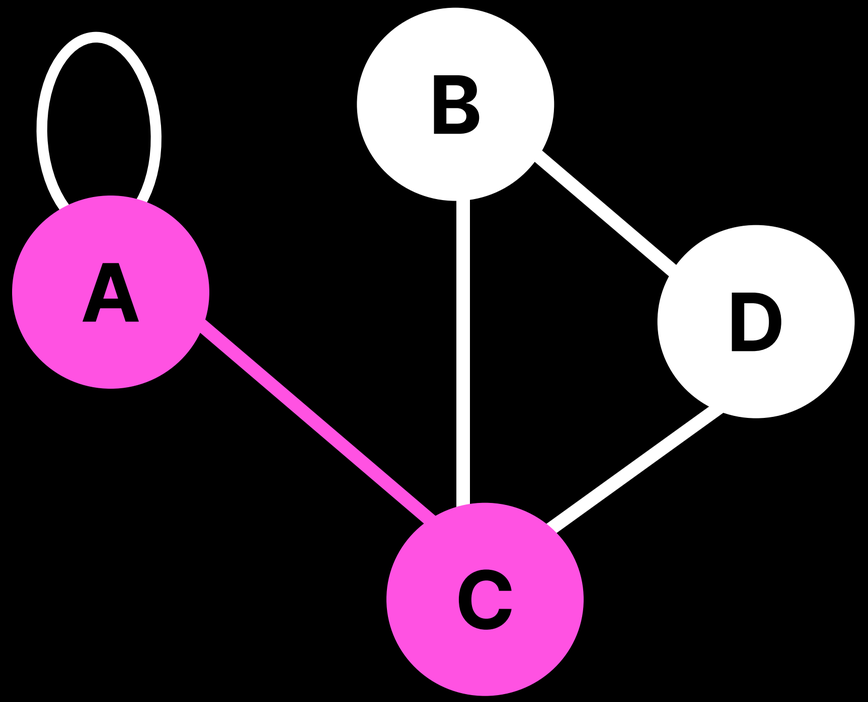
Nabomatrise

	A	B	C	D
A	1	0		
B	0			
C				
D				



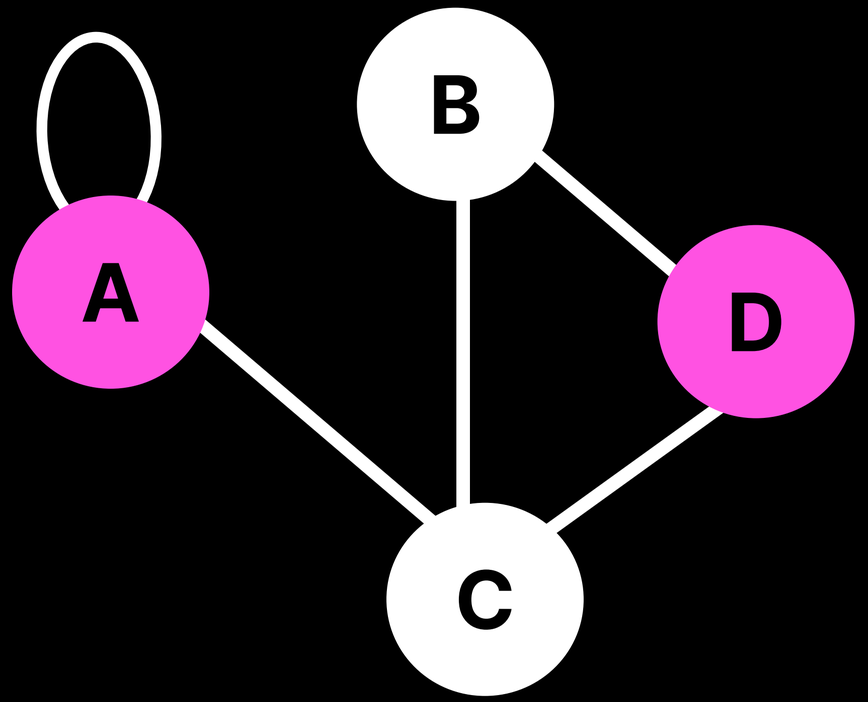
Nabomatrise

	A	B	C	D
A	1	0	1	
B	0			
C	1			
D				



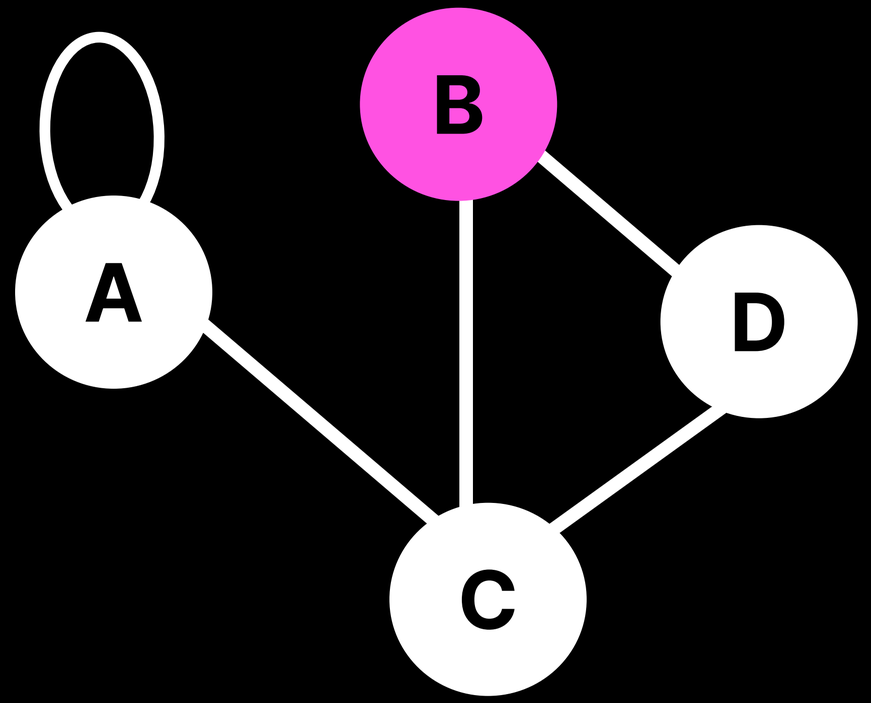
Nabomatrise

	A	B	C	D
A	1	0	1	0
B	0			
C	1			
D	0			



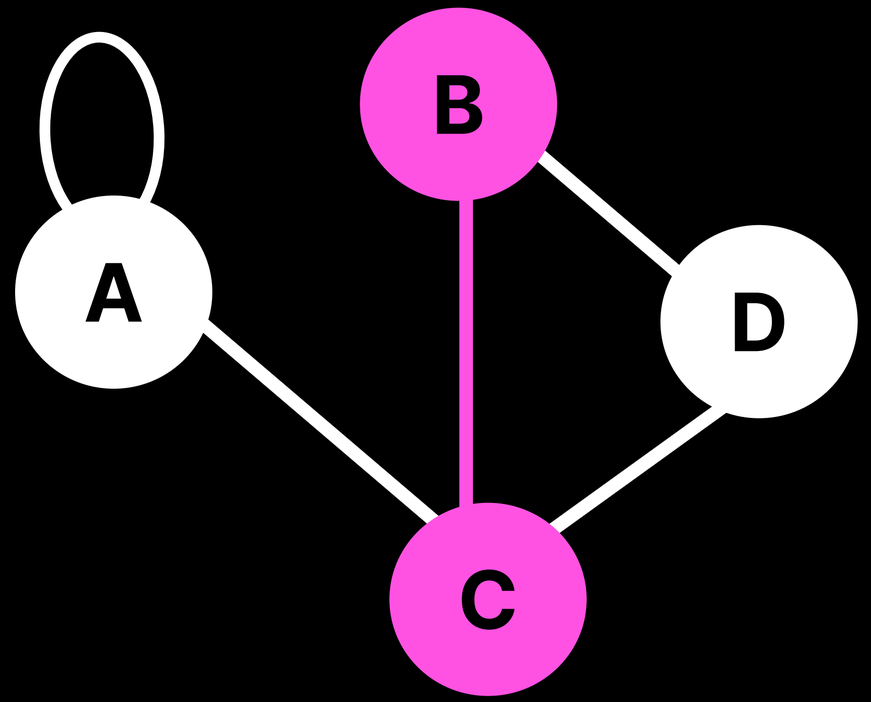
Nabomatrise

	A	B	C	D
A	1	0	1	0
B	0	0		
C	1			
D	0			



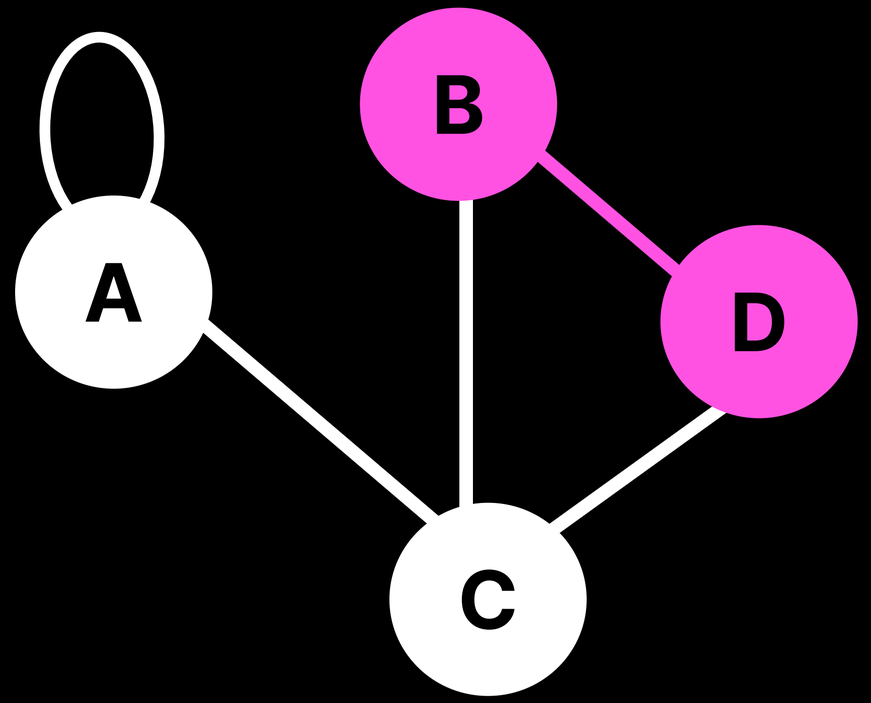
Nabomatrise

	A	B	C	D
A	1	0	1	0
B	0	0	1	
C	1	1		
D	0			



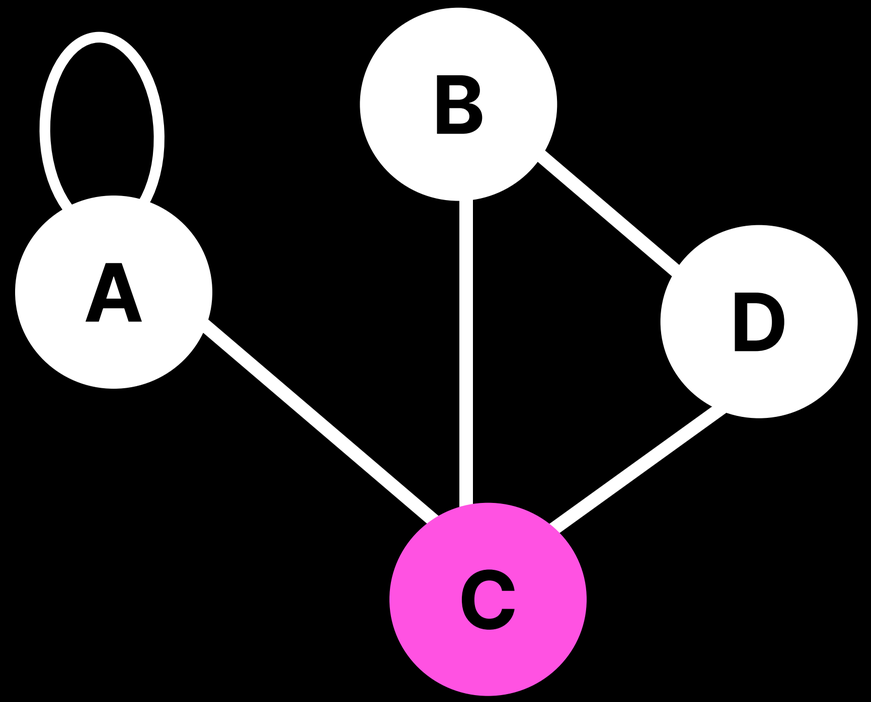
Nabomatrise

	A	B	C	D
A	1	0	1	0
B	0	0	1	1
C	1	1		
D	0	1		



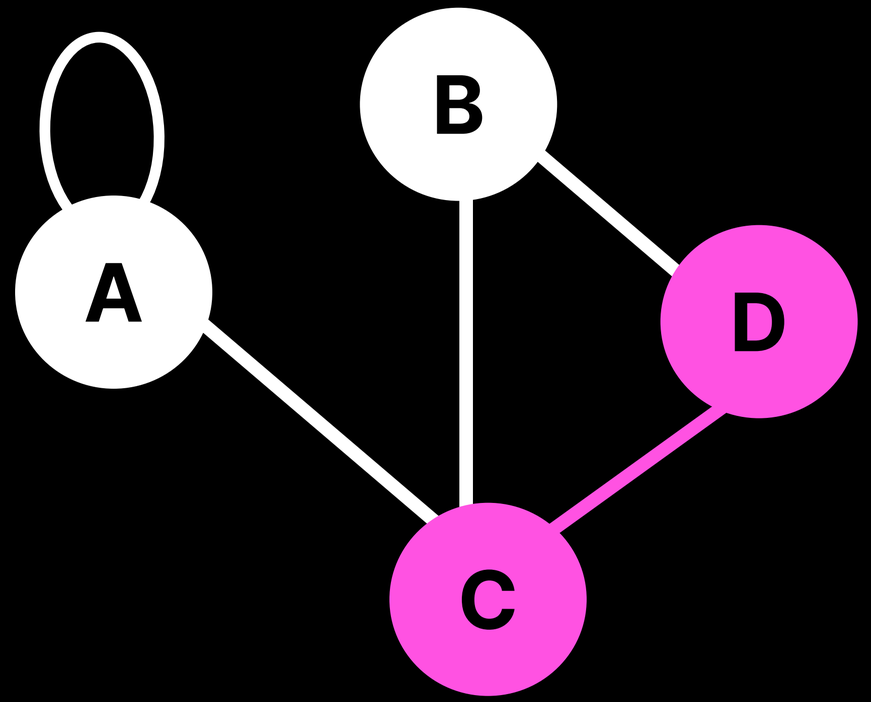
Nabomatrise

	A	B	C	D
A	1	0	1	0
B	0	0	1	1
C	1	1	0	
D	0	1		



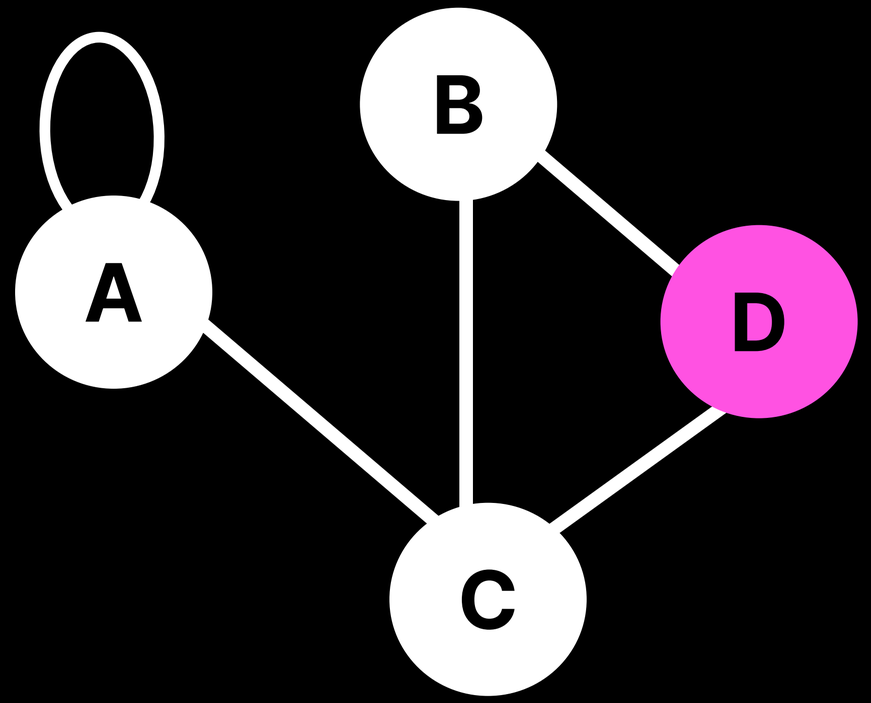
Nabomatrise

	A	B	C	D
A	1	0	1	0
B	0	0	1	1
C	1	1	0	1
D	0	1	1	



Nabomatrise

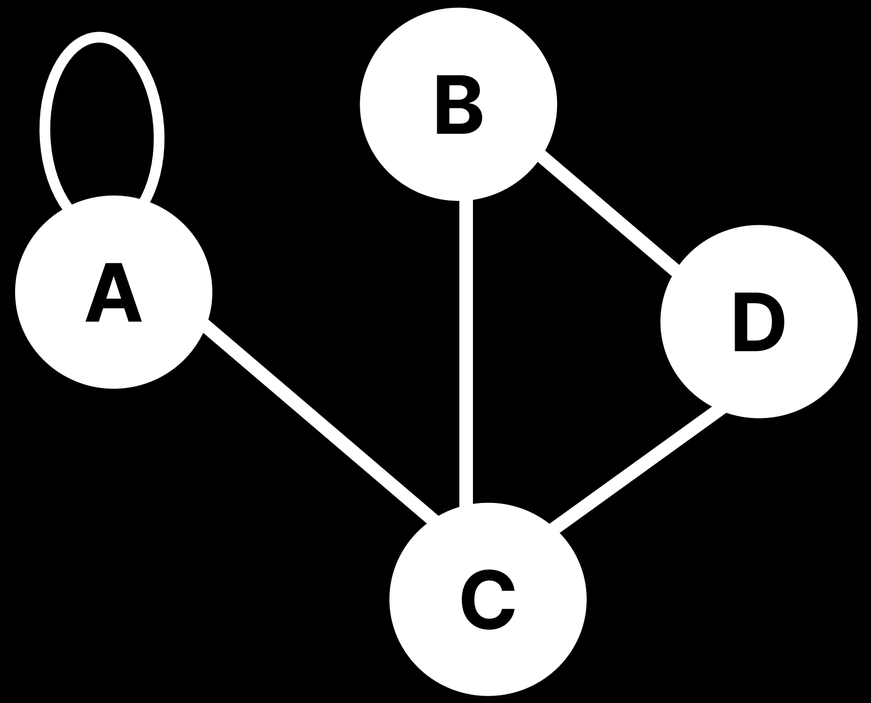
	A	B	C	D
A	1	0	1	0
B	0	0	1	1
C	1	1	0	1
D	0	1	1	0



Nabomatrise

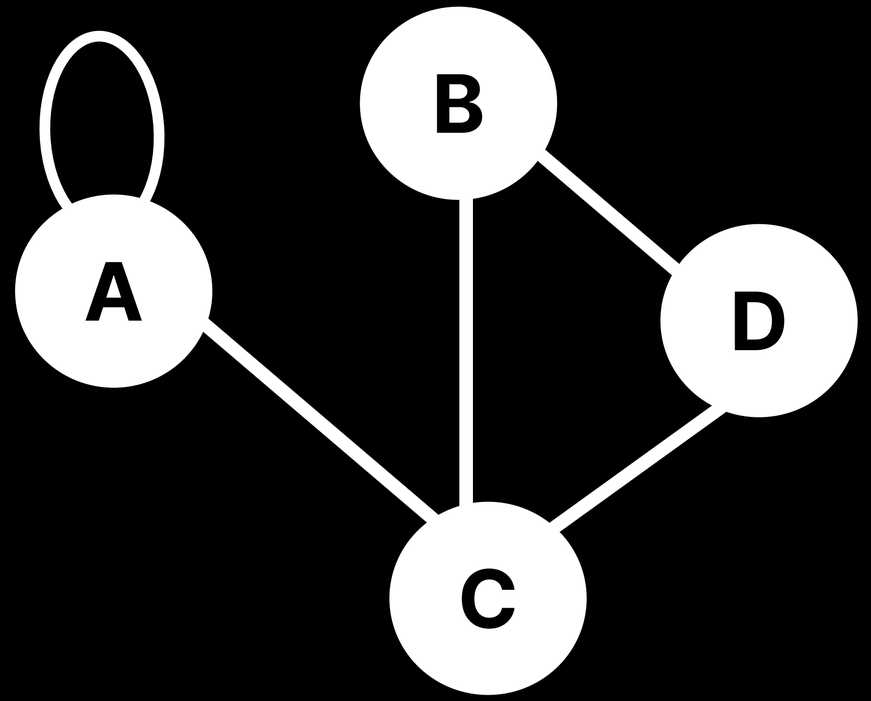
	A	B	C	D
A	1	0	1	0
B	0	0	1	1
C	1	1	0	1
D	0	1	1	0

løkker



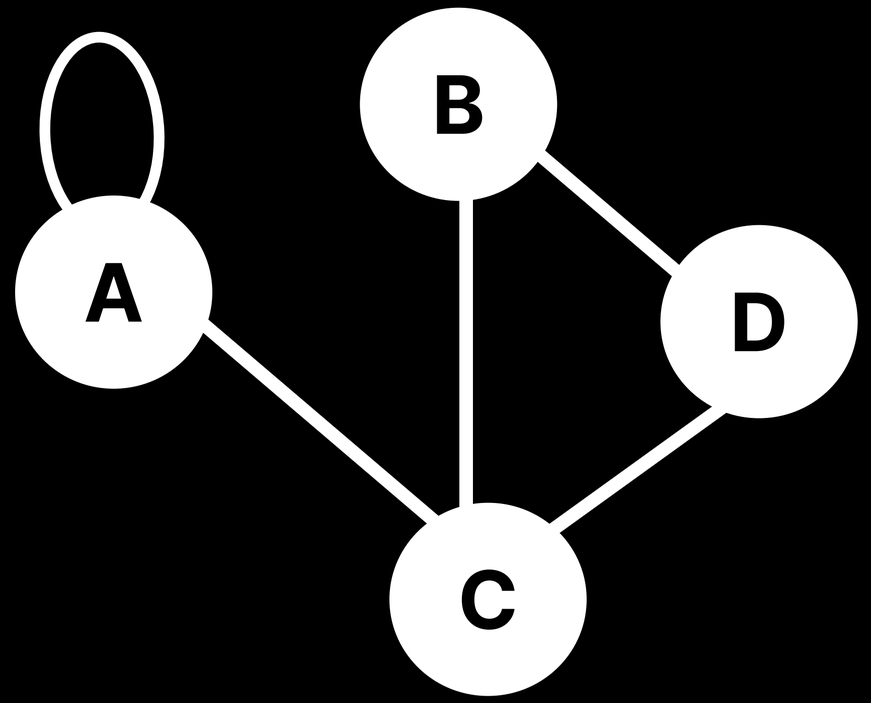
Nabomatrise

	A	B	C	D
A	1	0	1	0
B	0	0	1	1
C	1	1	0	1
D	0	1	1	0



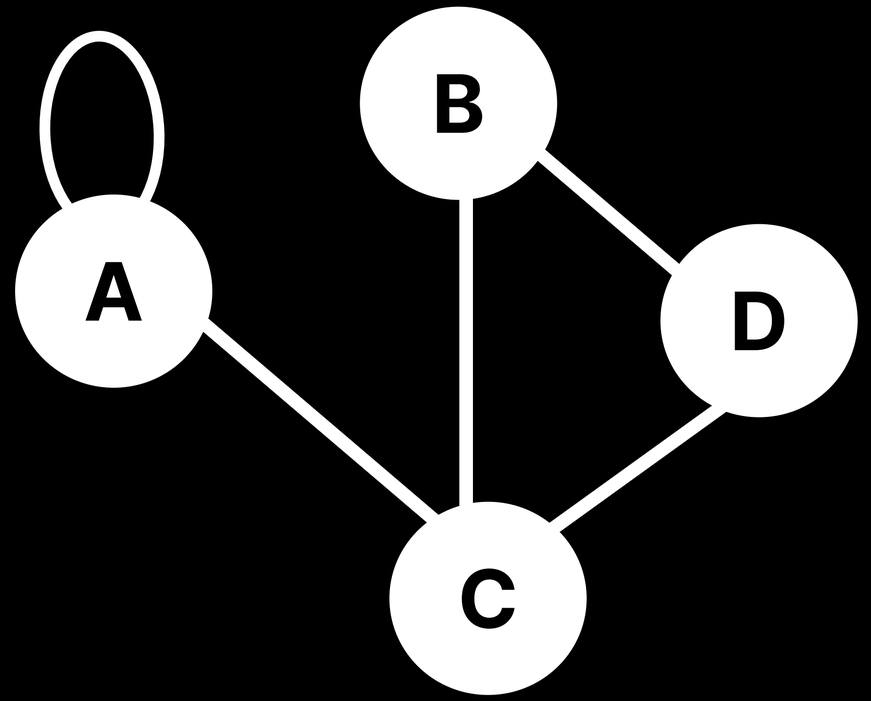
Nabomatrise

	A	B	C	D
A	1	0	1	0
B	0	0	1	1
C	1	1	0	1
D	0	1	1	0



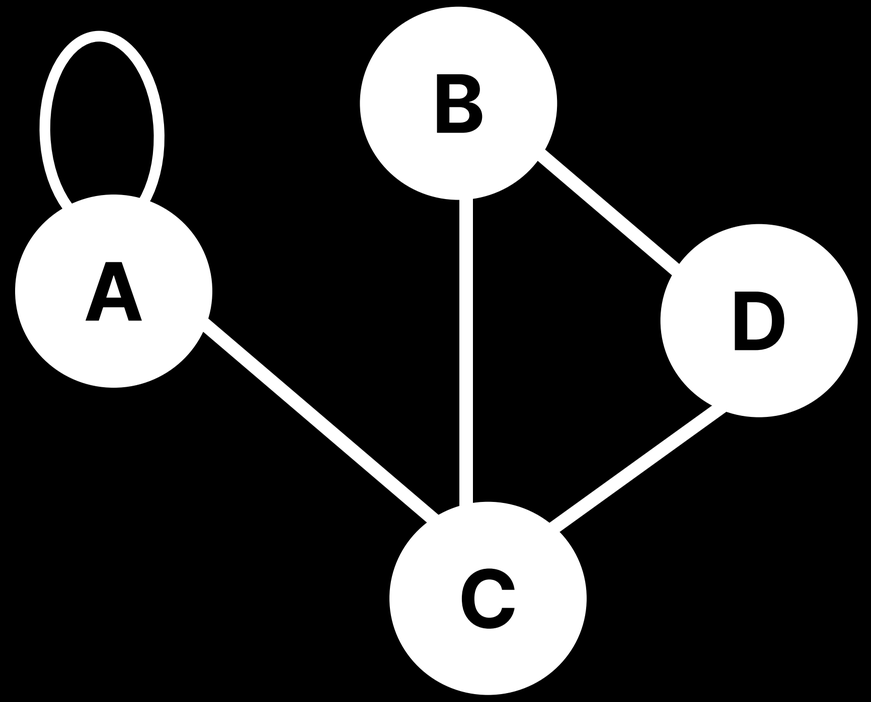
Nabomatrise

	A	B	C	D
A	1	0	1	0
B	0	0	1	1
C	1	1	0	1
D	0	1	1	0



Nabomatrise

	A	B	C	D
A	1	0	1	0
B	0	0	1	1
C	1	1	0	1
D	0	1	1	0



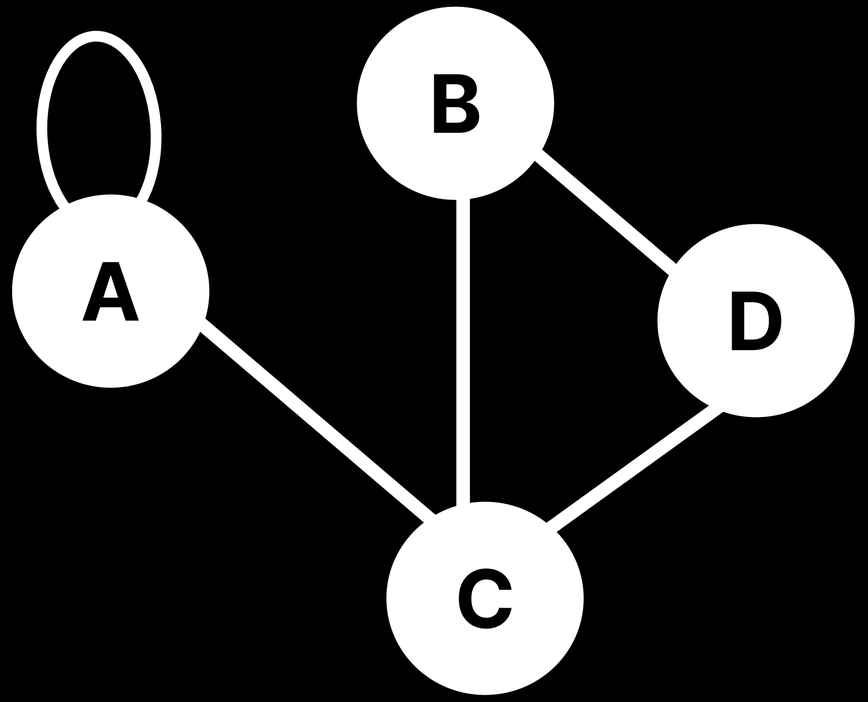
Naboliste

A:

B:

C:

D:



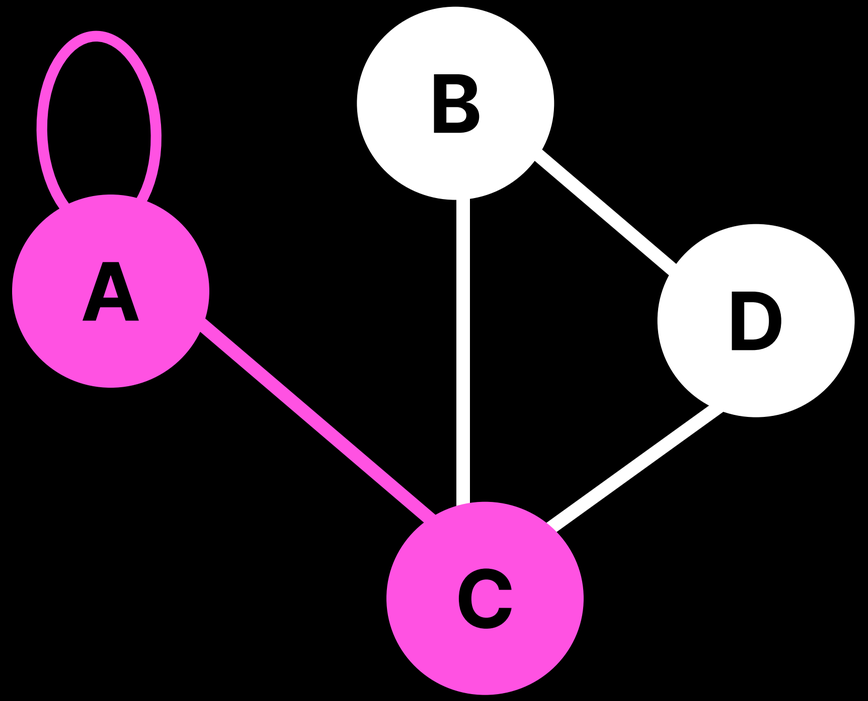
Naboliste

A: [A, C]

B:

C:

D:



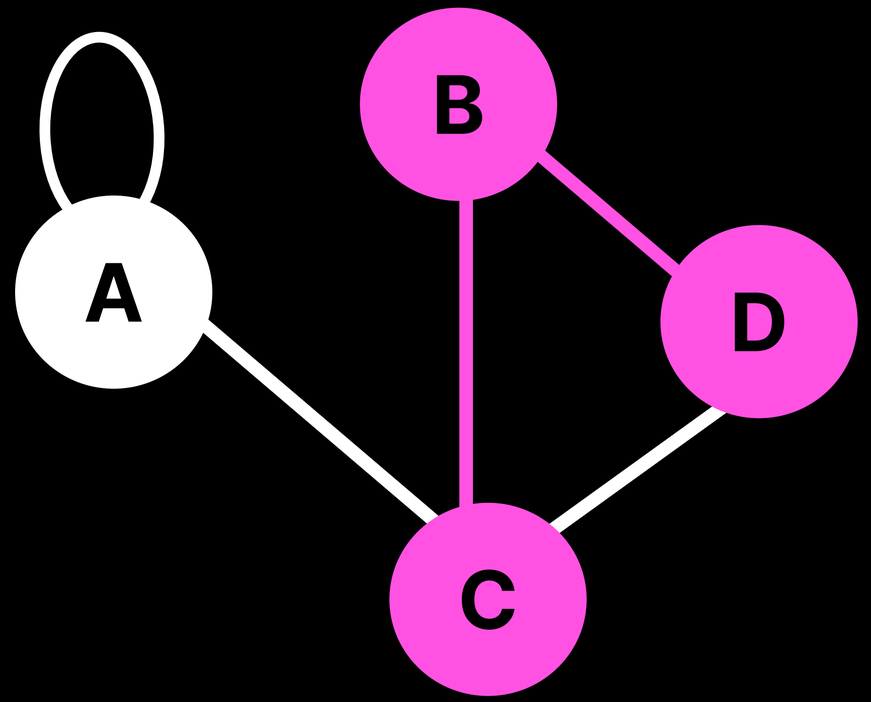
Naboliste

A: [A, C]

B: [C, D]

C:

D:



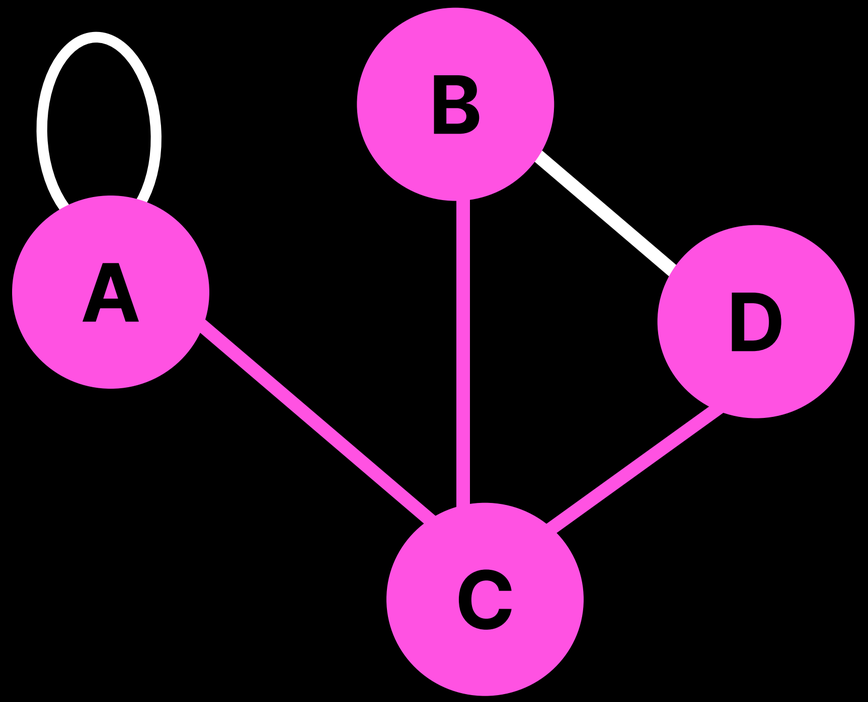
Naboliste

A: [A, C]

B: [C, D]

C: [A, B, D]

D:



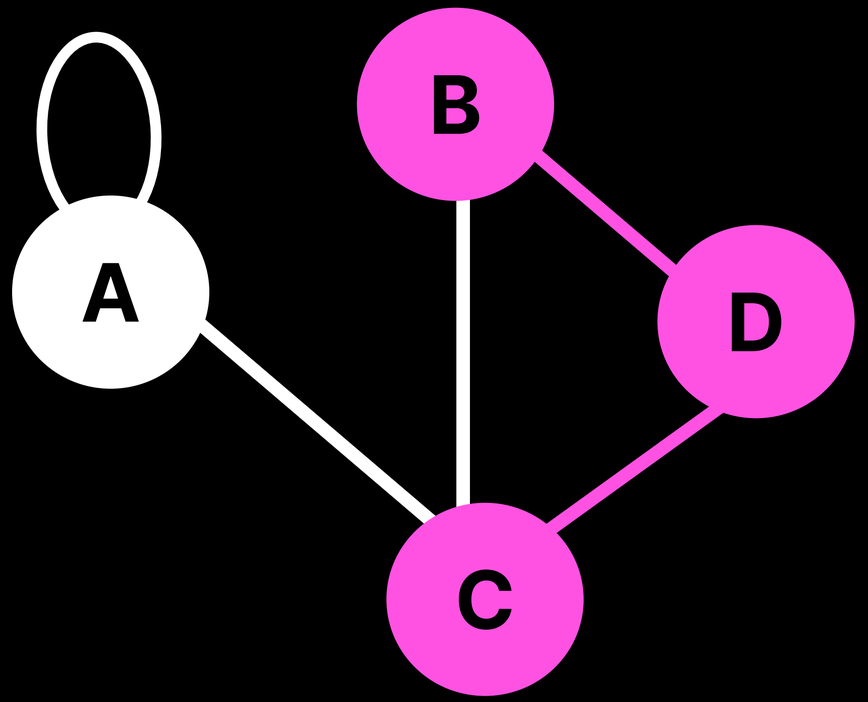
Naboliste

A: [A, C]

B: [C, D]

C: [A, B, D]

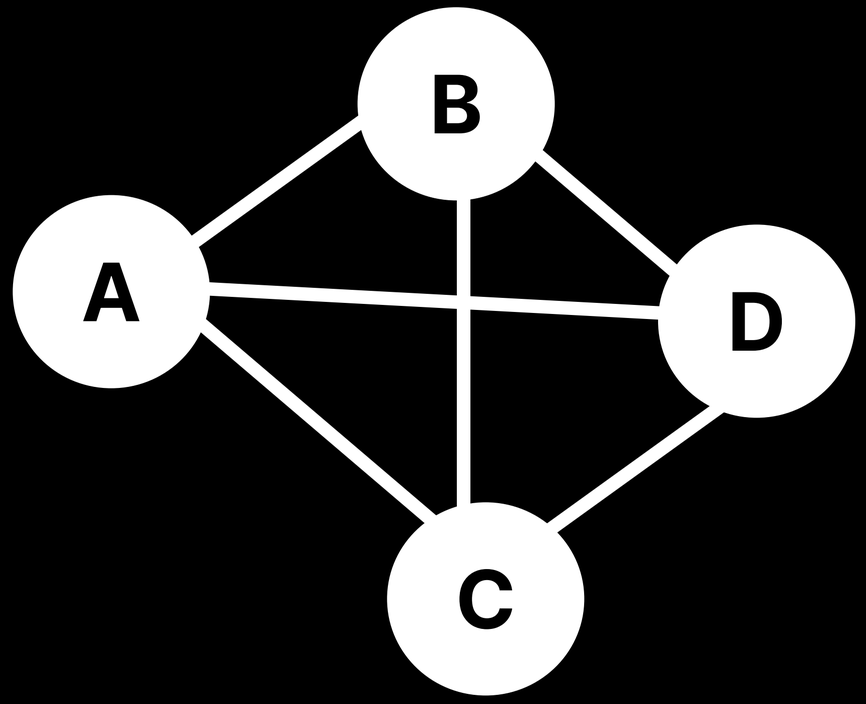
D: [B, C]



Nabomatrise vs. Naboliste

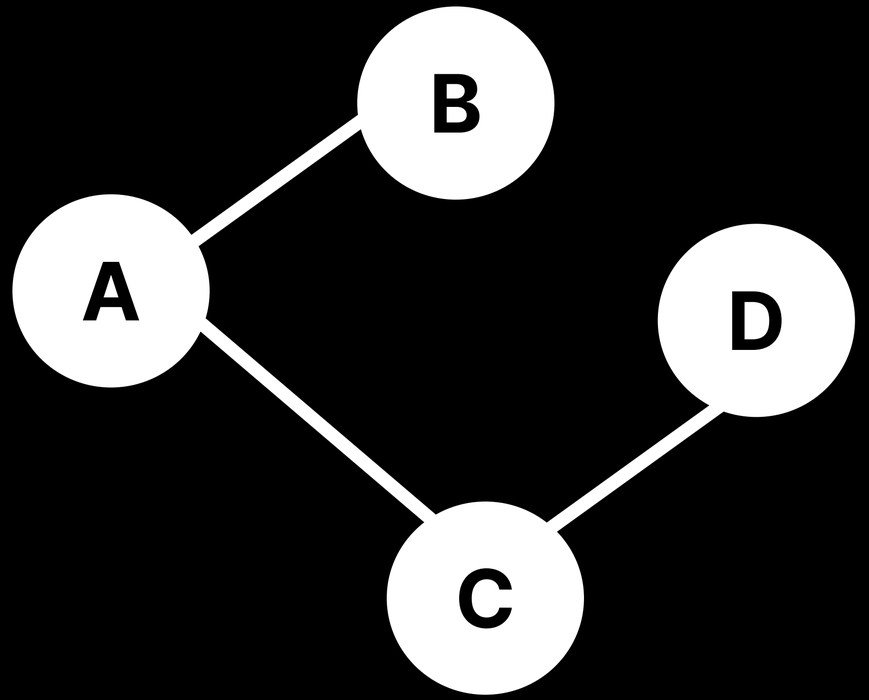
Nabomatrise

Egner seg for tette grafer



Naboliste

Egner seg for tynne grafer



Dybde-først søk (DFS)

Følger en vilkårlig sti bort fra en gitt startnode.

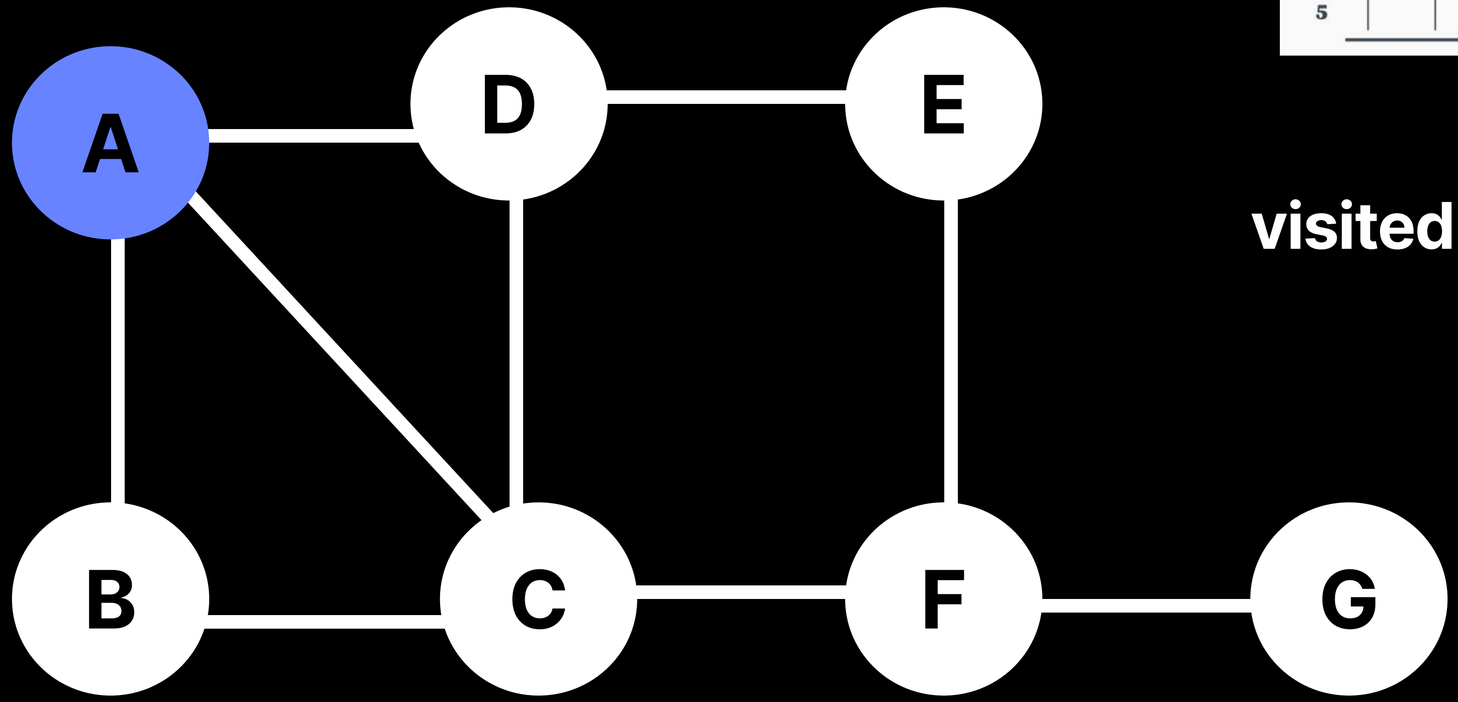
Når den ikke finner en ubesøkt node sporer den stien tilbake steg for steg og prøver å finne ubesøkte noder.

Beveger seg altså i dybden, og prøver å gå så langt bort fra startnoden som mulig via ubesøkte noder.

Kan blant annet brukes til å finne ut om en rettet graf er sterkt sammenhengende.

Dybde-først søk (DFS)

Gitt startnoden A.



ALGORITHM: REKURSIVT DYBDE-FØRST SØK

Input: En graf $G = (V, E)$, en startnode u og en mengde `visited` med besøkte noder

Output: Besøker alle noder i G som kan nås fra u nøyaktig én gang

```
1 Procedure DFSVisit(G, u, visited)
2     add u to visited
3     for (u, v) ∈ E do
4         if v ∉ visited then
5             DFSVisit(G, v, visited)
```

u=A

visited

De rekursive kallene hittil:

Dybde-først søk (DFS)

Gitt startnoden A.

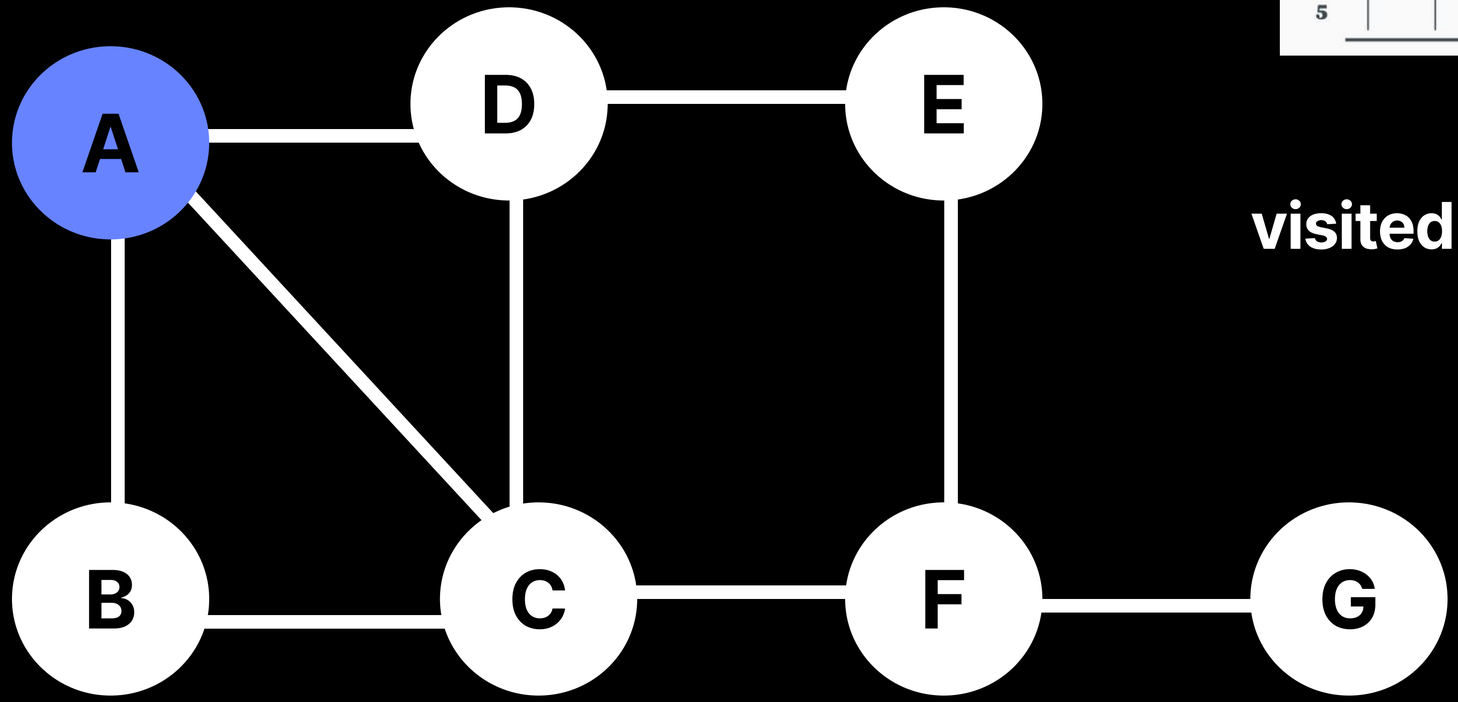
ALGORITHM: REKURSIVT DYBDE-FØRST SØK

Input: En graf $G = (V, E)$, en startnode u og en mengde `visited` med besøkte noder

Output: Besøker alle noder i G som kan nås fra u nøyaktig én gang

```
1 Procedure DFSVisit(G, u, visited)
2     add u to visited
3     for (u, v) ∈ E do
4         if v ∉ visited then
5             DFSVisit(G, v, visited)
```

u=A



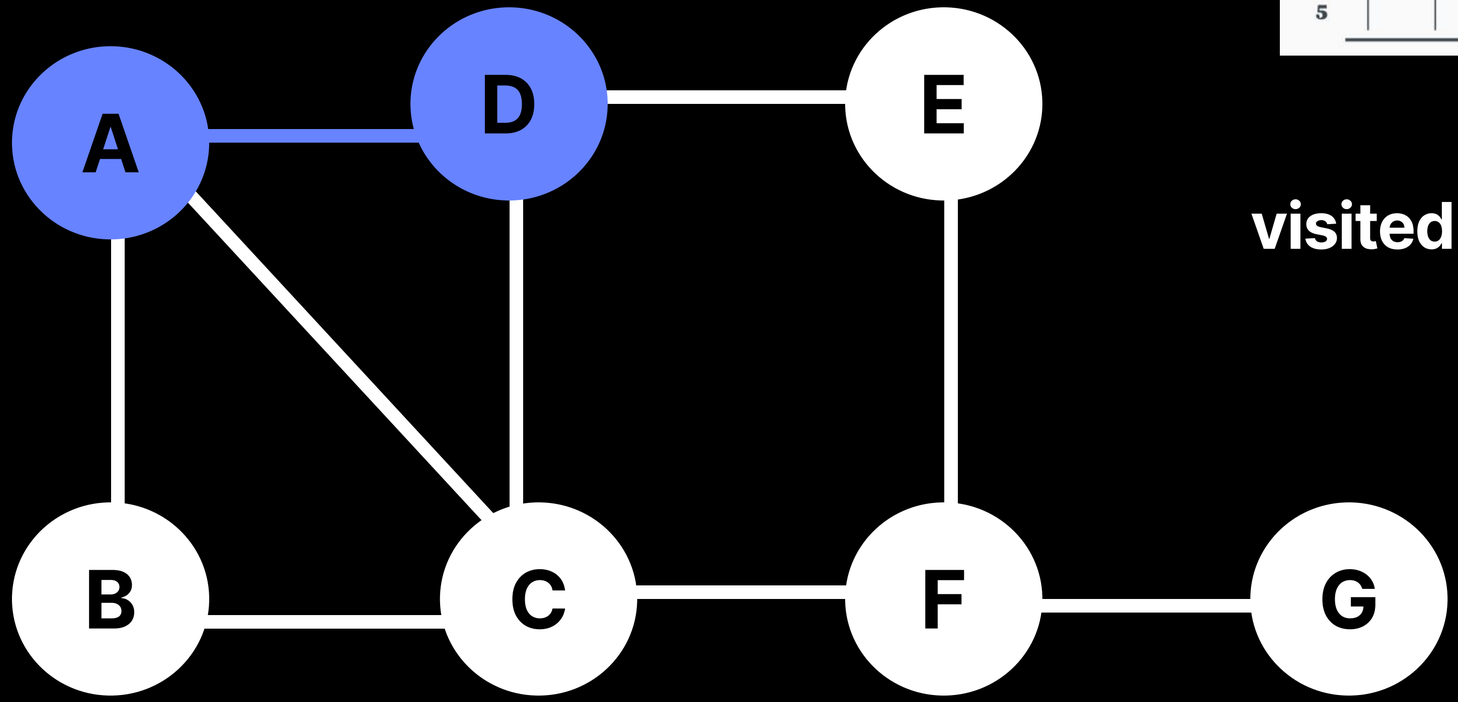
visited A

De rekursive kallene hittil:

DFSVisit(G, A, visited)

Dybde-først søk (DFS)

Følger vi så en vilkårlig sti bort fra A, så langt vi klarer med ubesøkte noder.



ALGORITHM: REKURSIVT DYBDE-FØRST SØK

Input: En graf $G = (V, E)$, en startnode u og en mengde `visited` med besøkte noder

Output: Besøker alle noder i G som kan nås fra u nøyaktig én gang

```
1 Procedure DFSVisit(G, u, visited)
2     add u to visited
3     for (u, v) ∈ E do
4         if v ∉ visited then
5             DFSVisit(G, v, visited)
```

u=D

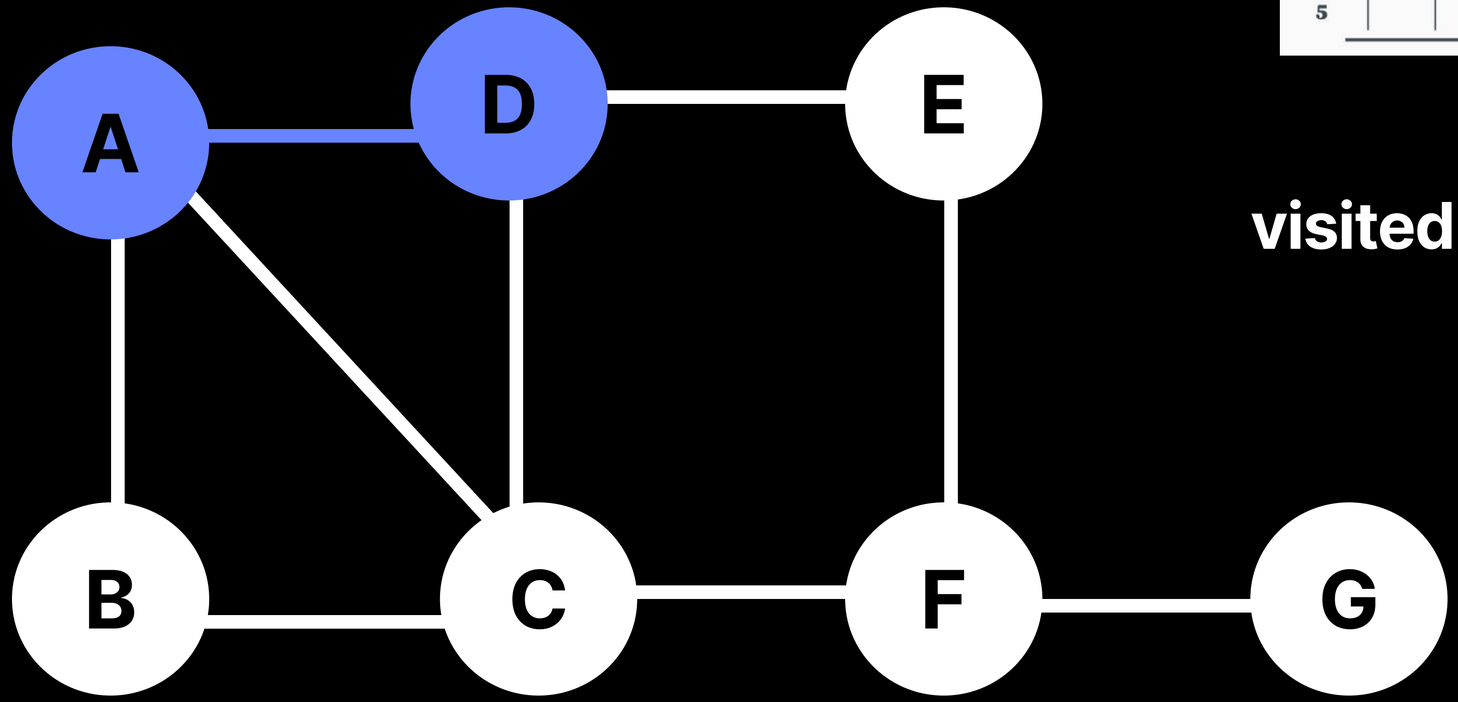
visited A

De rekursive kallene hittil:

DFSVisit(G, A, visited)
  DFSVisit(G, D, visited)

Dybde-først søk (DFS)

Følger vi så en vilkårlig sti bort fra A, så langt vi klarer med ubesøkte noder.



ALGORITHM: REKURSIVT DYBDE-FØRST SØK

Input: En graf $G = (V, E)$, en startnode u og en mengde `visited` med besøkte noder

Output: Besøker alle noder i G som kan nås fra u nøyaktig én gang

```
1 Procedure DFSVisit(G, u, visited)
2     add u to visited
3     for (u, v) ∈ E do
4         if v ∉ visited then
5             DFSVisit(G, v, visited)
```

u=D

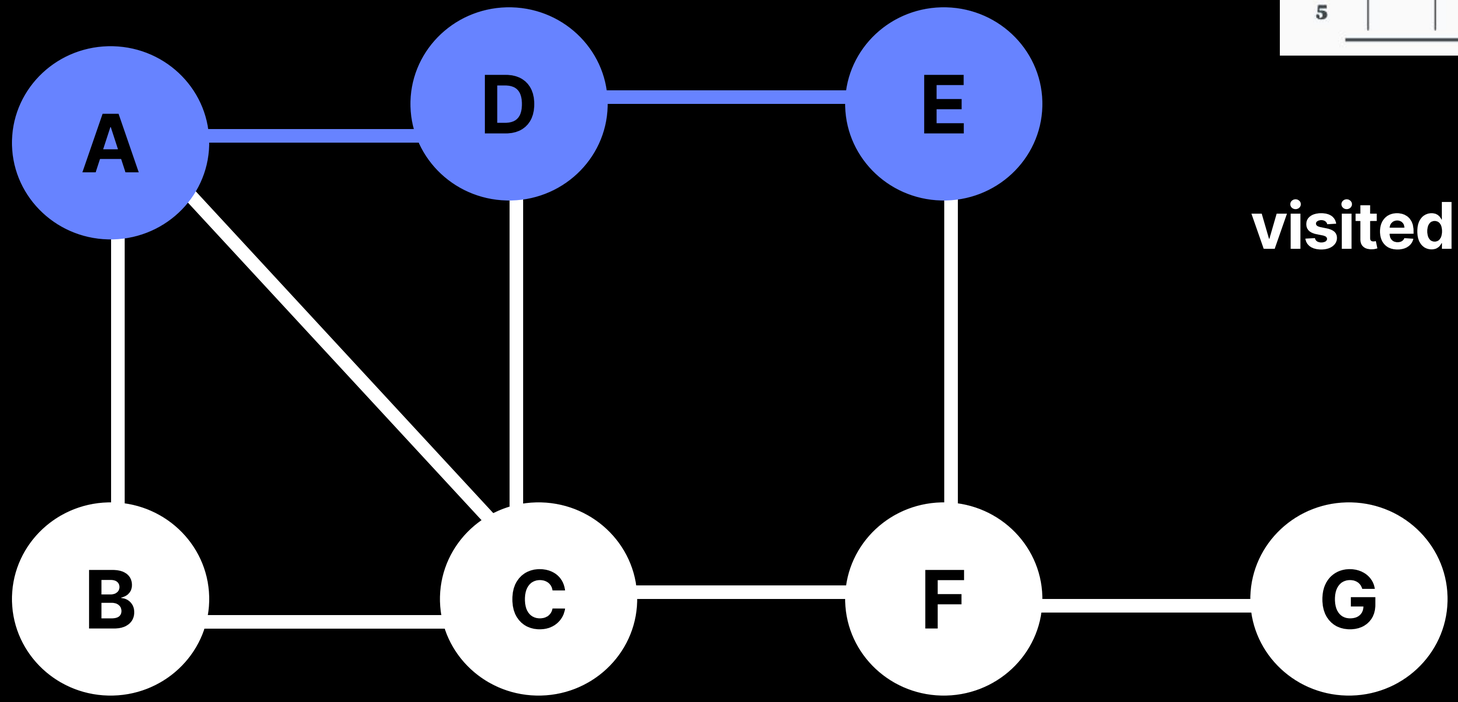
visited A D

De rekursive kallene hittil:

DFSVisit(G, A, visited)
  DFSVisit(G, D, visited)

Dybde-først søk (DFS)

Følger vi så en vilkårlig sti bort fra A, så langt vi klarer med ubesøkte noder.



ALGORITHM: REKURSIVT DYBDE-FØRST SØK

Input: En graf $G = (V, E)$, en startnode u og en mengde `visited` med besøkte noder

Output: Besøker alle noder i G som kan nås fra u nøyaktig én gang

```
1 Procedure DFSVisit(G, u, visited)
2     add u to visited
3     for (u, v) ∈ E do
4         if v ∉ visited then
5             DFSVisit(G, v, visited)
```

u=E

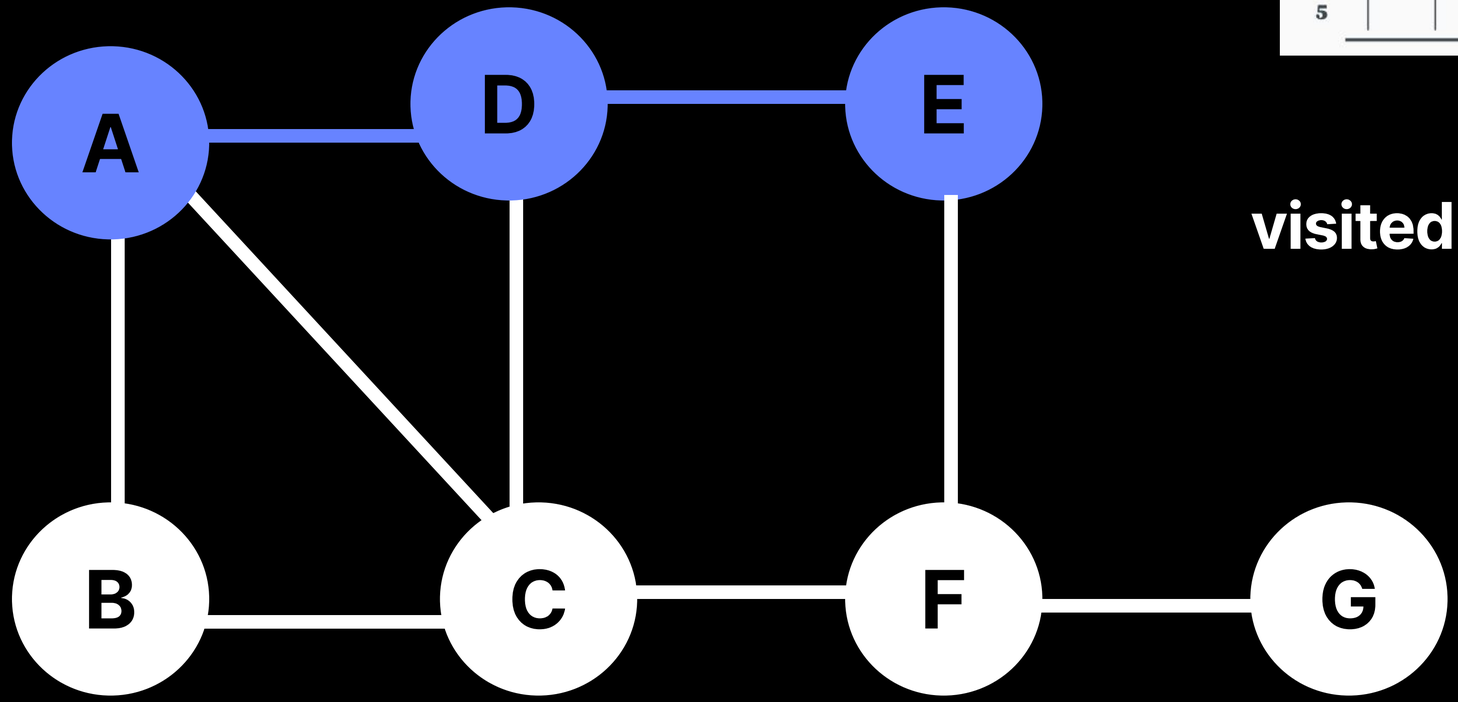
visited A D

De rekursive kallene hittil:

DFSVisit(G, A, visited)
DFSVisit(G, D, visited)
DFSVisit(G, E, visited)

Dybde-først søk (DFS)

Følger vi så en vilkårlig sti bort fra A, så langt vi klarer med ubesøkte noder.



ALGORITHM: REKURSIVT DYBDE-FØRST SØK

Input: En graf $G = (V, E)$, en startnode u og en mengde `visited` med besøkte noder

Output: Besøker alle noder i G som kan nås fra u nøyaktig én gang

```
1 Procedure DFSVisit(G, u, visited)
2     add u to visited
3     for (u, v) ∈ E do
4         if v ∉ visited then
5             DFSVisit(G, v, visited)
```

u=E

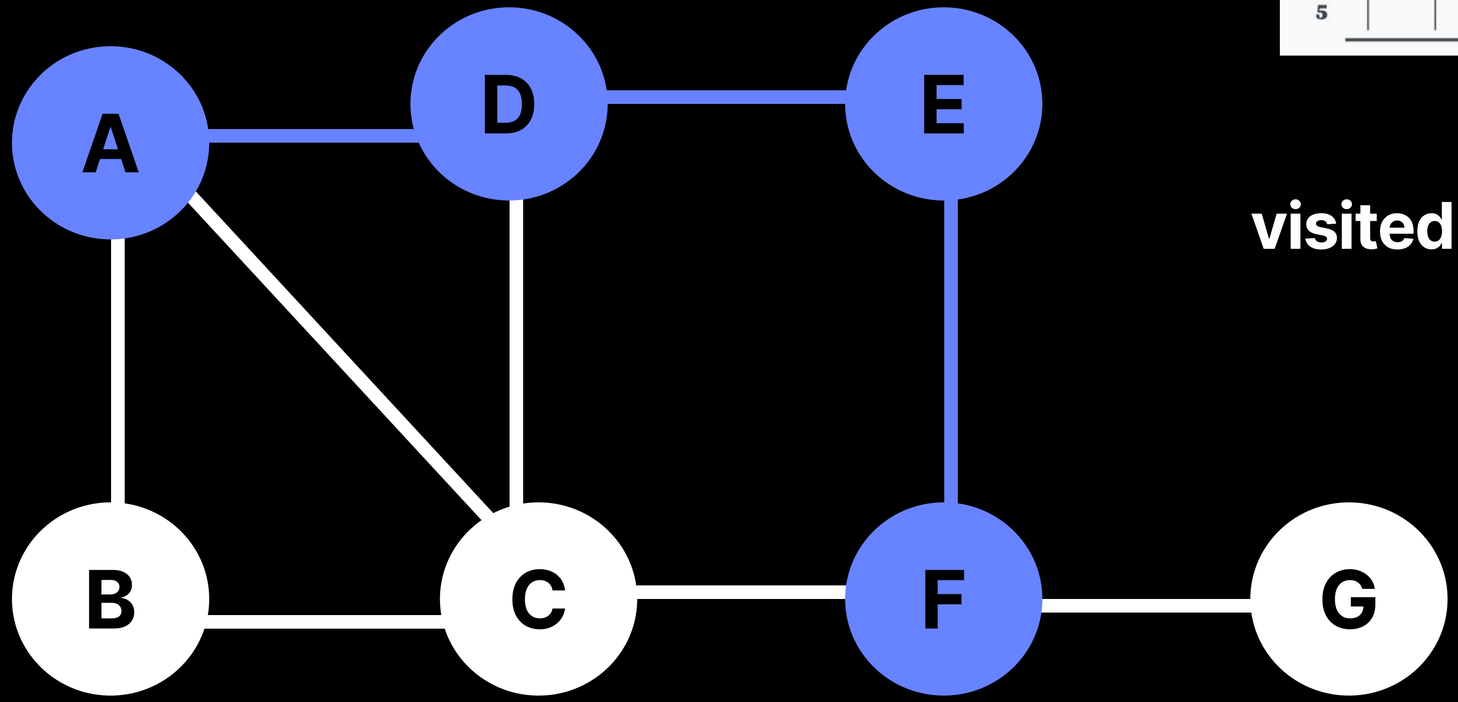
visited A D E

De rekursive kallene hittil:

DFSVisit(G, A, visited)
DFSVisit(G, D, visited)
DFSVisit(G, E, visited)

Dybde-først søk (DFS)

Følger vi så en vilkårlig sti bort fra A, så langt vi klarer med ubesøkte noder.



ALGORITHM: REKURSIVT DYBDE-FØRST SØK

Input: En graf $G = (V, E)$, en startnode u og en mengde `visited` med besøkte noder

Output: Besøker alle noder i G som kan nås fra u nøyaktig én gang

```
1 Procedure DFSVisit(G, u, visited)
2     add u to visited
3     for (u, v) ∈ E do
4         if v ∉ visited then
5             DFSVisit(G, v, visited)
```

u=F

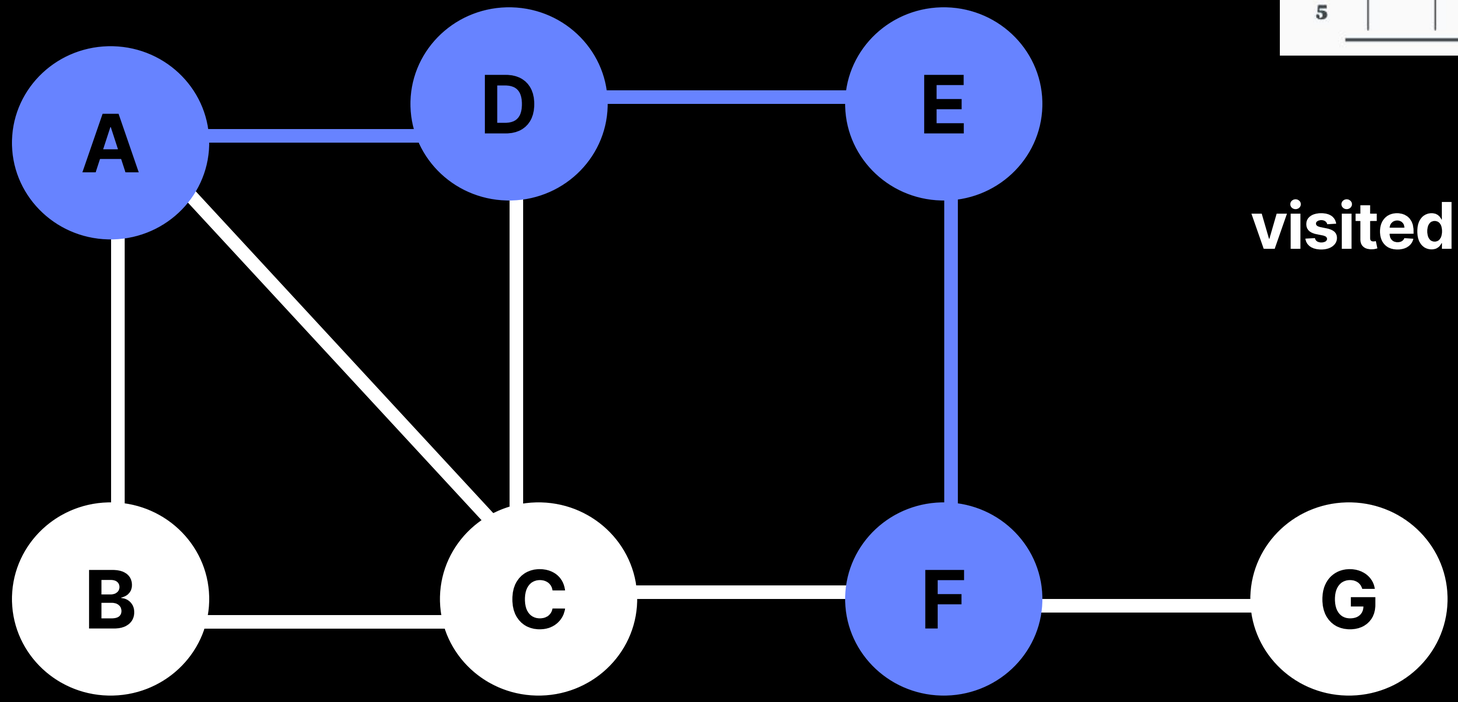
visited A D E

De rekursive kallene hittil:

DFSVisit(G, A, visited)
DFSVisit(G, D, visited)
DFSVisit(G, E, visited)
DFSVisit(G, F, visited)

Dybde-først søk (DFS)

Følger vi så en vilkårlig sti bort fra A, så langt vi klarer med ubesøkte noder.



ALGORITHM: REKURSIVT DYBDE-FØRST SØK

Input: En graf $G = (V, E)$, en startnode u og en mengde `visited` med besøkte noder

Output: Besøker alle noder i G som kan nås fra u nøyaktig én gang

```
1 Procedure DFSVisit(G, u, visited)
2     add u to visited
3     for (u, v) ∈ E do
4         if v ∉ visited then
5             DFSVisit(G, v, visited)
```

u=F

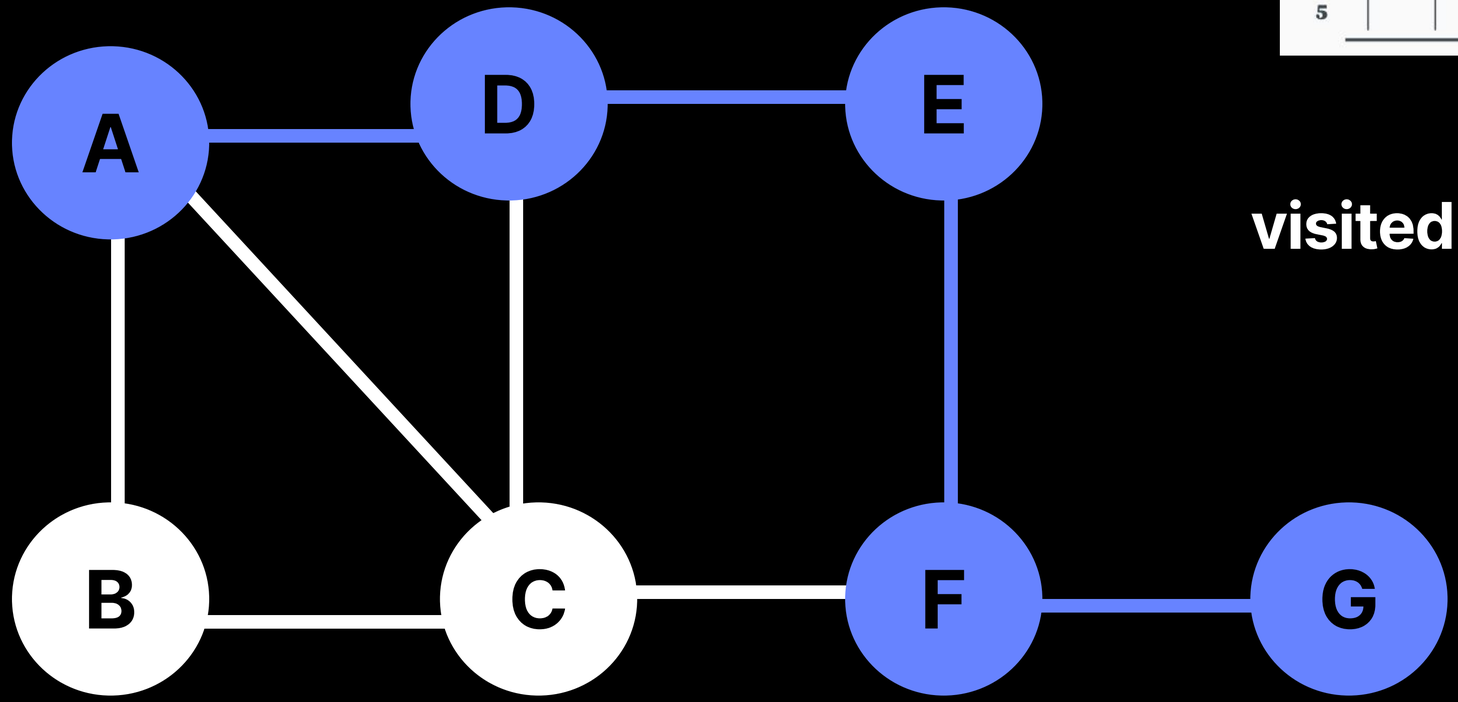
visited A D E F

De rekursive kallene hittil:

DFSVisit(G, A, visited)
DFSVisit(G, D, visited)
DFSVisit(G, E, visited)
DFSVisit(G, F, visited)

Dybde-først søk (DFS)

Følger vi så en vilkårlig sti bort fra A, så langt vi klarer med ubesøkte noder.



ALGORITHM: REKURSIVT DYBDE-FØRST SØK

Input: En graf $G = (V, E)$, en startnode u og en mengde `visited` med besøkte noder

Output: Besøker alle noder i G som kan nås fra u nøyaktig én gang

```
1 Procedure DFSVisit(G, u, visited)
2     add u to visited
3     for (u, v) ∈ E do
4         if v ∉ visited then
5             DFSVisit(G, v, visited)
```

u=G

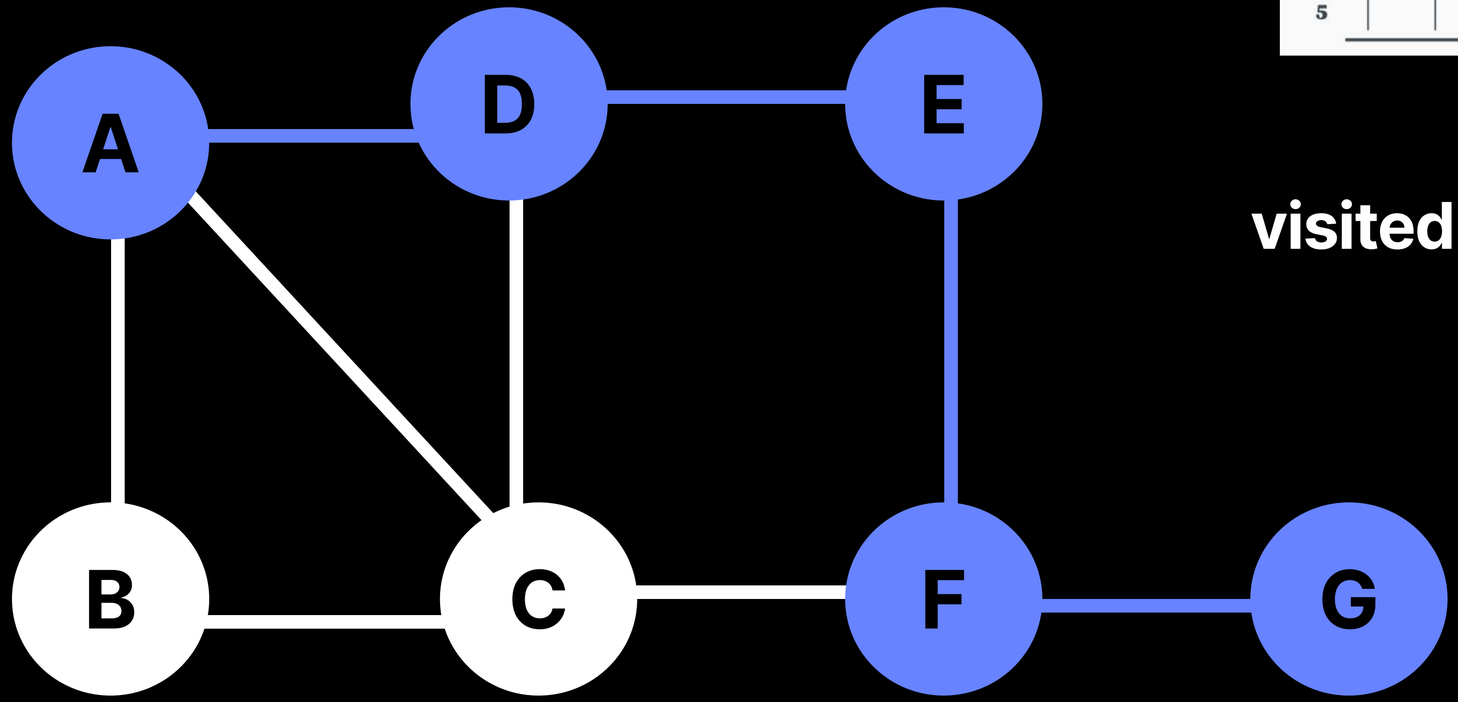
visited A D E F

De rekursive kallene hittil:

DFSVisit(G, A, visited)
DFSVisit(G, D, visited)
DFSVisit(G, E, visited)
DFSVisit(G, F, visited)
DFSVisit(G, G, visited)

Dybde-først søk (DFS)

Følger vi så en vilkårlig sti bort fra A, så langt vi klarer med ubesøkte noder.



ALGORITHM: REKURSIVT DYBDE-FØRST SØK

Input: En graf $G = (V, E)$, en startnode u og en mengde `visited` med besøkte noder

Output: Besøker alle noder i G som kan nås fra u nøyaktig én gang

```
1 Procedure DFSVisit(G, u, visited)
2     add u to visited
3     for (u, v) ∈ E do
4         if v ∉ visited then
5             DFSVisit(G, v, visited)
```

u=G

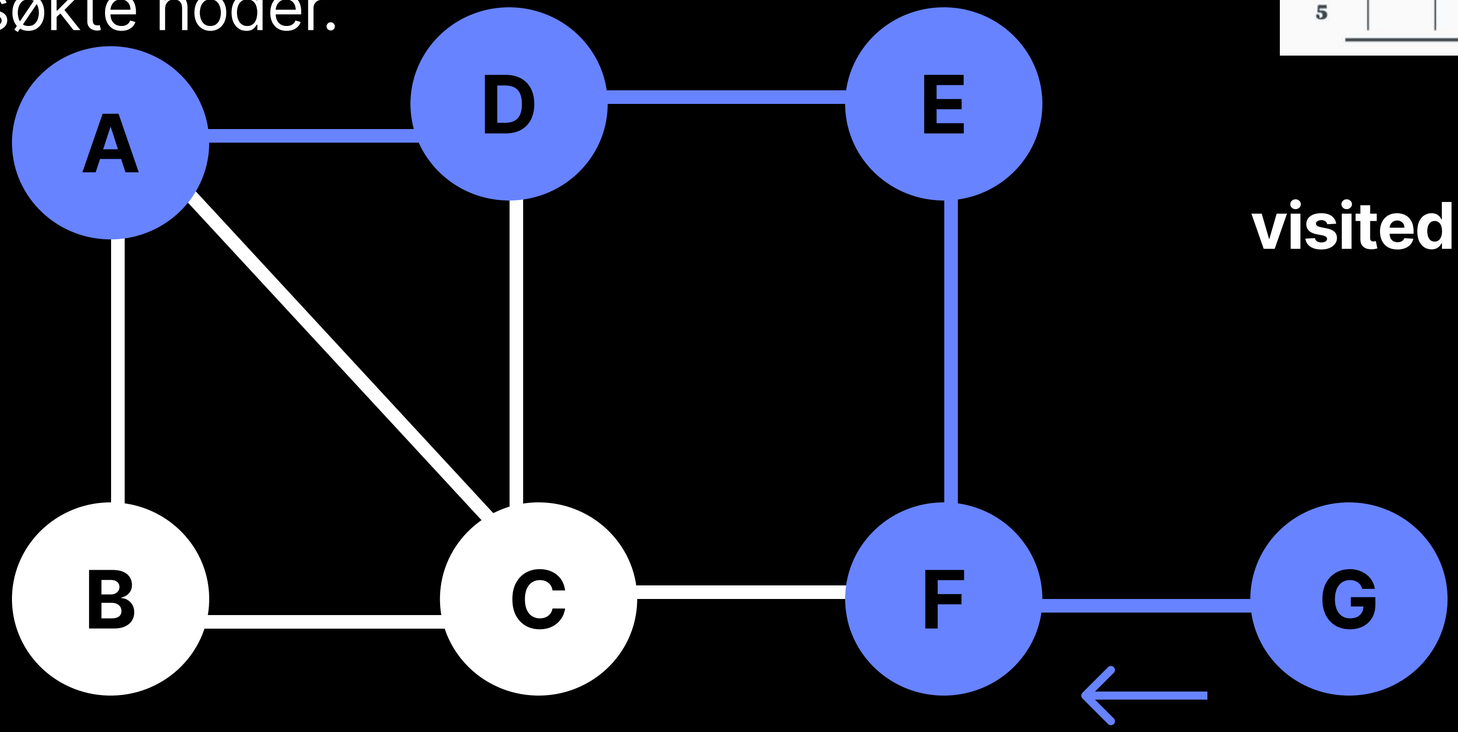
visited A D E F G

De rekursive kallene hittil:

DFSVisit(G, A, visited)
DFSVisit(G, D, visited)
DFSVisit(G, E, visited)
DFSVisit(G, F, visited)
DFSVisit(G, G, visited)

Dybde-først søk (DFS)

Når vi ikke finner en ubesøkt node sporer vi stien tilbake steg for steg og prøver å finne ubesøkte noder.



ALGORITHM: REKURSIVT DYBDE-FØRST SØK

Input: En graf $G = (V, E)$, en startnode u og en mengde `visited` med besøkte noder

Output: Besøker alle noder i G som kan nås fra u nøyaktig én gang

```
1 Procedure DFSVisit(G, u, visited)
2     add u to visited
3     for (u, v) ∈ E do
4         if v ∉ visited then
5             DFSVisit(G, v, visited)
```

u=F

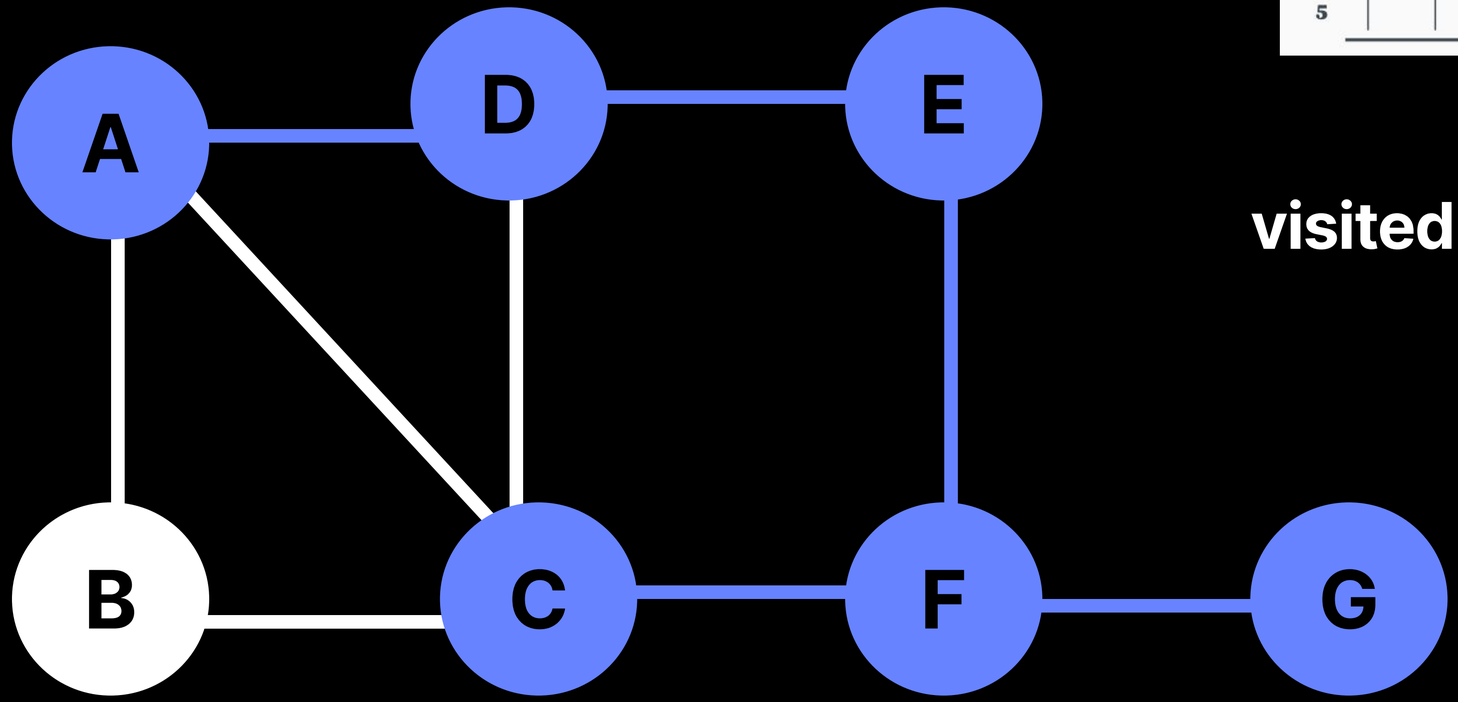
visited A D E F G

De rekursive kallene hittil:

DFSVisit(G, A, visited)
DFSVisit(G, D, visited)
DFSVisit(G, E, visited)
DFSVisit(G, F, visited)

Dybde-først søk (DFS)

Og fortsetter å bevege oss sålangt vi klarer fra A via ubesøkte noder



ALGORITHM: REKURSIVT DYBDE-FØRST SØK

Input: En graf $G = (V, E)$, en startnode u og en mengde `visited` med besøkte noder

Output: Besøker alle noder i G som kan nås fra u nøyaktig én gang

```
1 Procedure DFSVisit(G, u, visited)
2     add u to visited
3     for (u, v) ∈ E do
4         if v ∉ visited then
5             DFSVisit(G, v, visited)
```

u=C

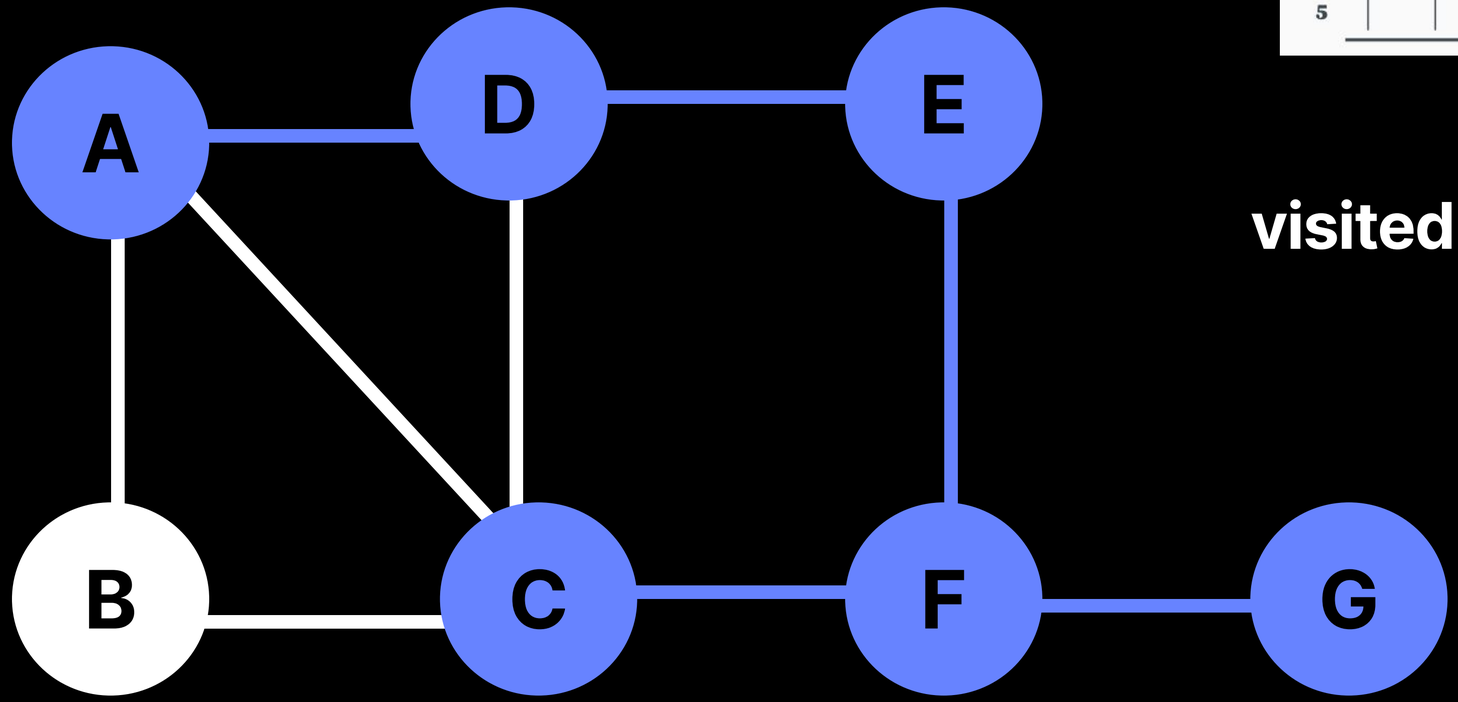
visited A D E F G

De rekursive kallene hittil:

DFSVisit(G, A, visited)
DFSVisit(G, D, visited)
DFSVisit(G, E, visited)
DFSVisit(G, F, visited)
DFSVisit(G, C, visited)

Dybde-først søk (DFS)

Og fortsetter å bevege oss sålangt vi klarer fra A via ubesøkte noder



ALGORITHM: REKURSIVT DYBDE-FØRST SØK

Input: En graf $G = (V, E)$, en startnode u og en mengde `visited` med besøkte noder

Output: Besøker alle noder i G som kan nås fra u nøyaktig én gang

```
1 Procedure DFSVisit(G, u, visited)
2     add u to visited
3     for (u, v) ∈ E do
4         if v ∉ visited then
5             DFSVisit(G, v, visited)
```

u=C

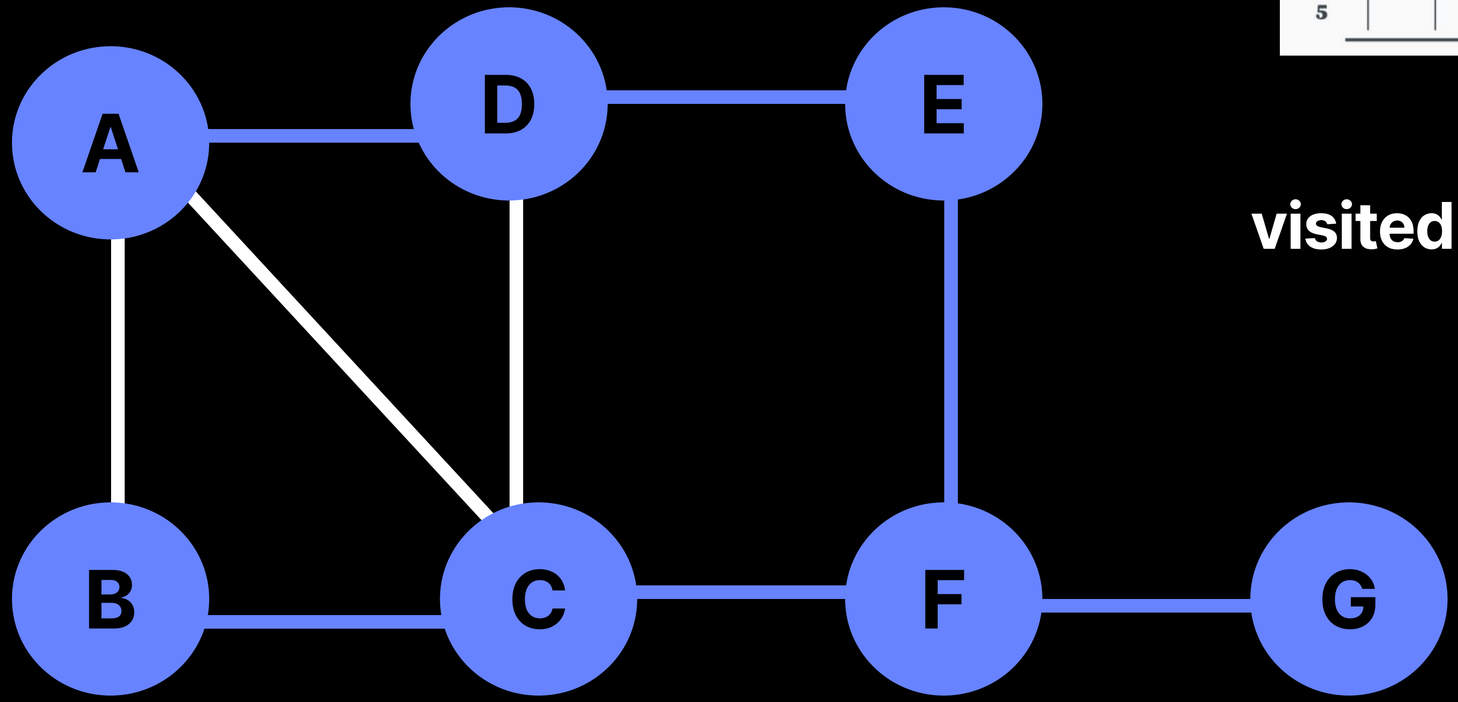
visited A D E F G C

De rekursive kallene hittil:

DFSVisit(G, A, visited)
DFSVisit(G, D, visited)
DFSVisit(G, E, visited)
DFSVisit(G, F, visited)
DFSVisit(G, C, visited)

Dybde-først søk (DFS)

Og fortsetter å bevege oss sålangt vi klarer fra A via ubesøkte noder



ALGORITHM: REKURSIVT DYBDE-FØRST SØK

Input: En graf $G = (V, E)$, en startnode u og en mengde `visited` med besøkte noder

Output: Besøker alle noder i G som kan nås fra u nøyaktig én gang

```
1 Procedure DFSVisit(G, u, visited)
2     add u to visited
3     for (u, v) ∈ E do
4         if v ∉ visited then
5             DFSVisit(G, v, visited)
```

u=B

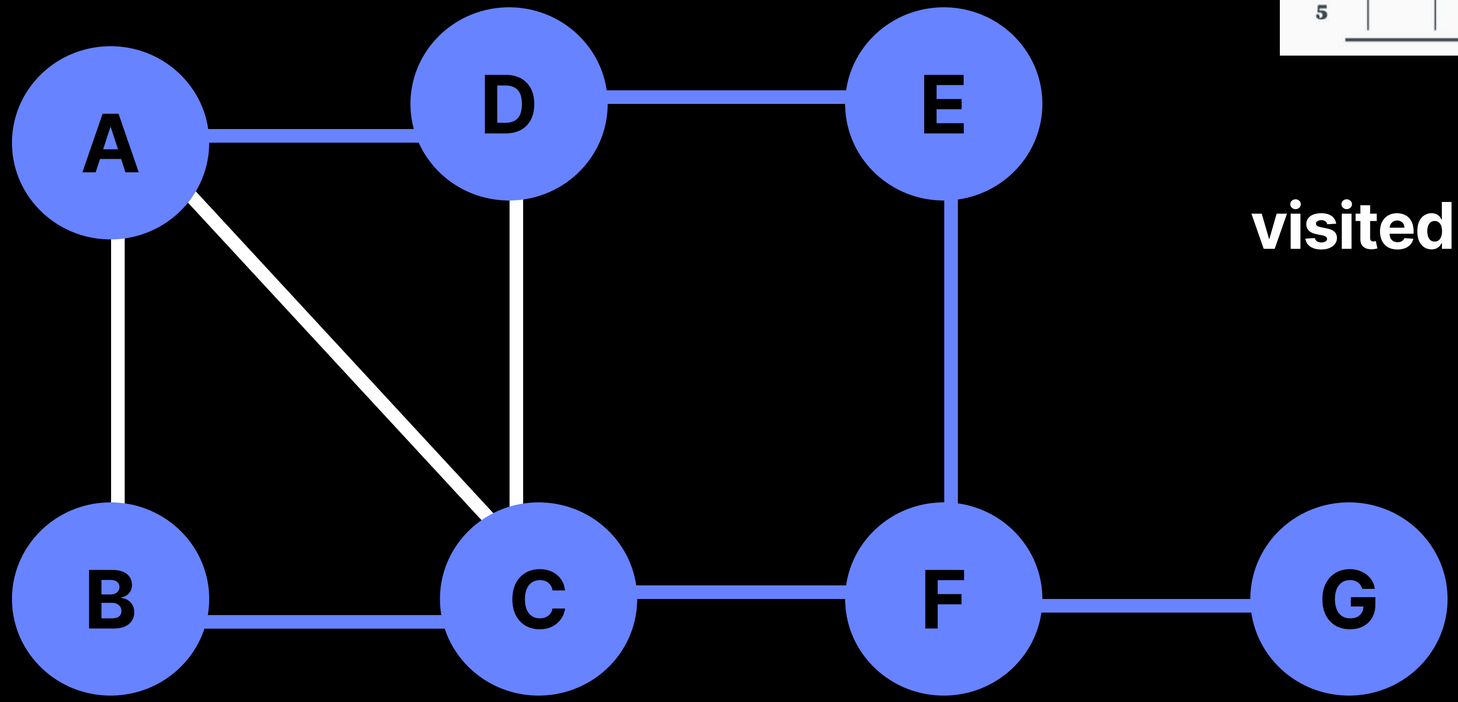
visited A D E F G C

De rekursive kallene hittil:

DFSVisit(G, A, visited)
DFSVisit(G, D, visited)
DFSVisit(G, E, visited)
DFSVisit(G, F, visited)
DFSVisit(G, C, visited)
DFSVisit(G, B, visited)

Dybde-først søk (DFS)

Og fortsetter å bevege oss sålangt vi klarer fra A via ubesøkte noder



ALGORITHM: REKURSIVT DYBDE-FØRST SØK

Input: En graf $G = (V, E)$, en startnode u og en mengde `visited` med besøkte noder

Output: Besøker alle noder i G som kan nås fra u nøyaktig én gang

```
1 Procedure DFSVisit(G, u, visited)
2     add u to visited
3     for (u, v) ∈ E do
4         if v ∉ visited then
5             DFSVisit(G, v, visited)
```

u=B

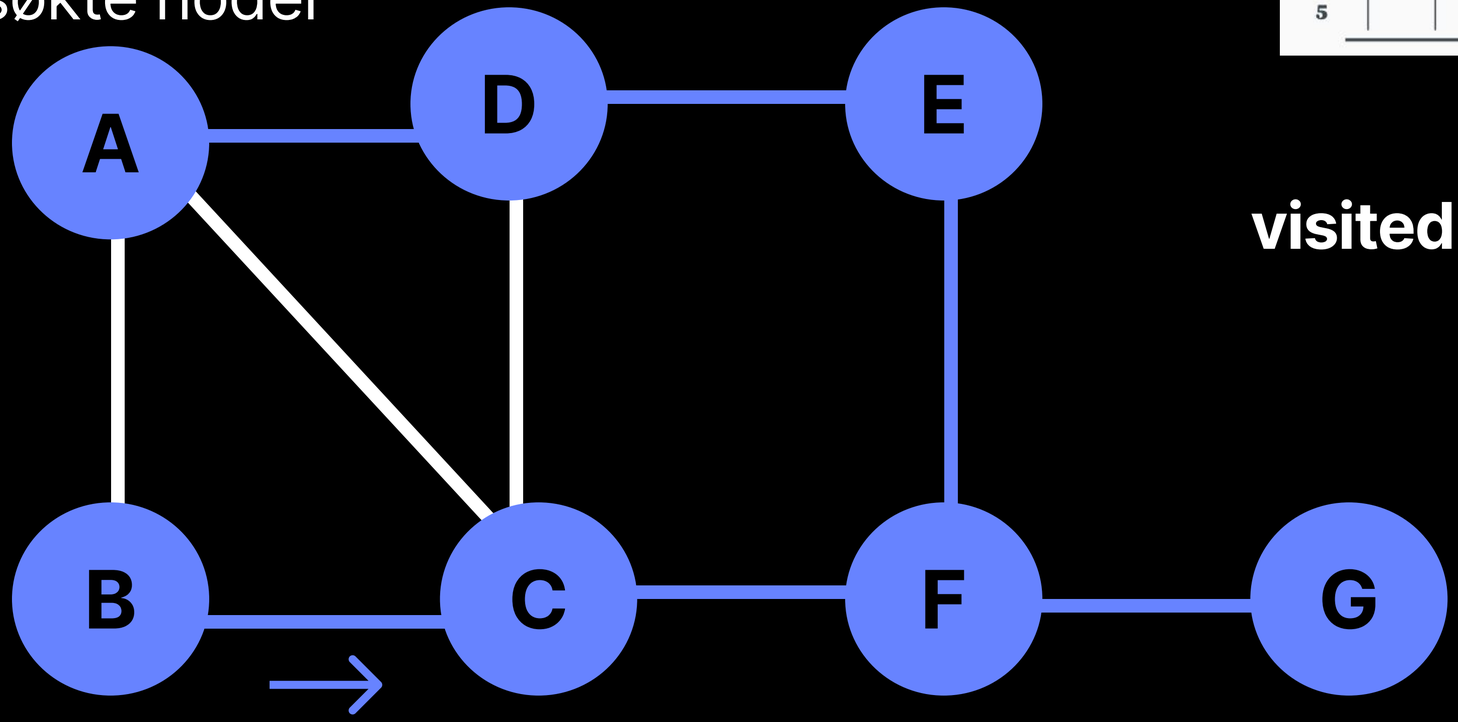
visited A D E F G C B

De rekursive kallene hittil:

DFSVisit(G, A, visited)
DFSVisit(G, D, visited)
DFSVisit(G, E, visited)
DFSVisit(G, F, visited)
DFSVisit(G, C, visited)
DFSVisit(G, B, visited)

Dybde-først søk (DFS)

Til slutt vil algoritmen rekursivt gå tilbake til startnoden steg for steg og sjekke etter ubesøkte noder



ALGORITHM: REKURSIVT DYBDE-FØRST SØK

Input: En graf $G = (V, E)$, en startnode u og en mengde `visited` med besøkte noder

Output: Besøker alle noder i G som kan nås fra u nøyaktig én gang

```
1 Procedure DFSVisit(G, u, visited)
2     add u to visited
3     for (u, v) ∈ E do
4         if v ∉ visited then
5             DFSVisit(G, v, visited)
```

u=C

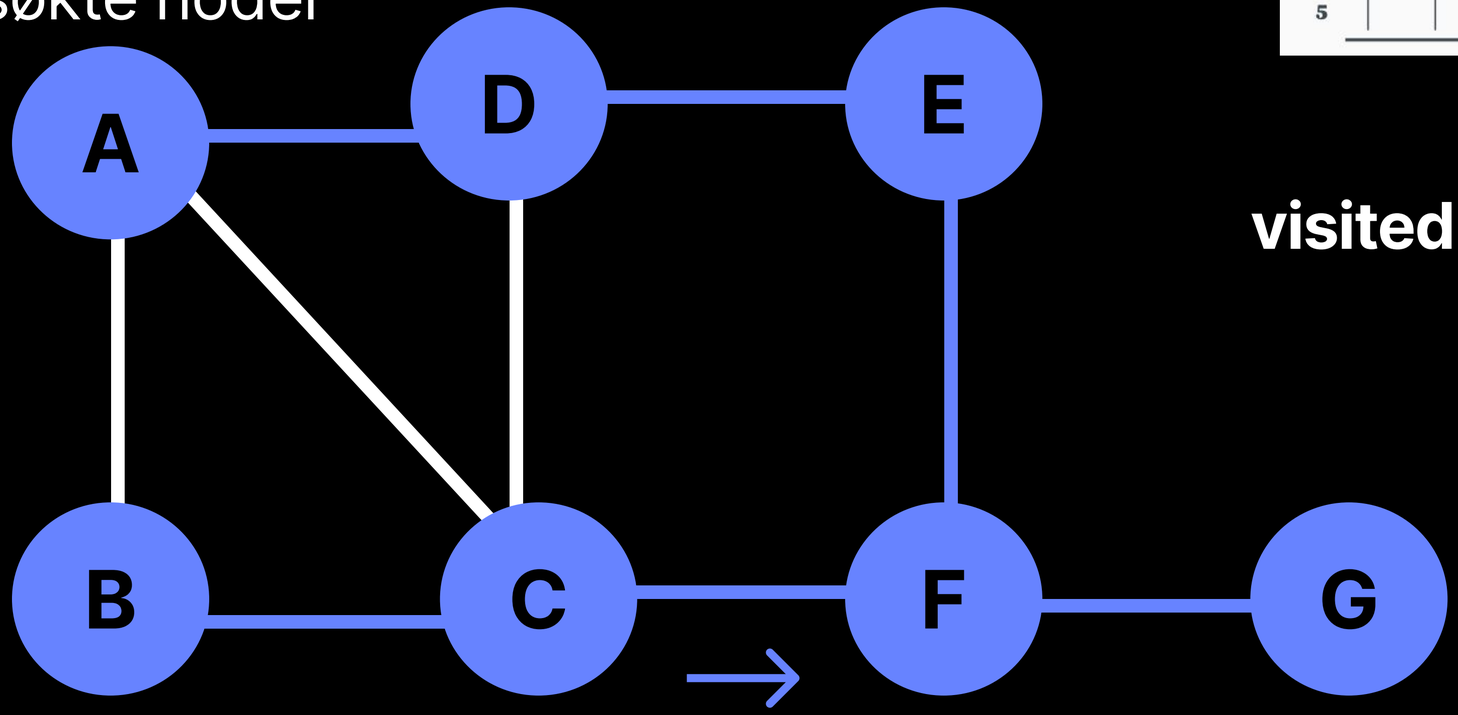
visited A D E F G C B

De rekursive kallene hittil:

DFSVisit(G, A, visited)
DFSVisit(G, D, visited)
DFSVisit(G, E, visited)
DFSVisit(G, F, visited)
DFSVisit(G, C, visited)

Dybde-først søk (DFS)

Til slutt vil algoritmen rekursivt gå tilbake til startnoden steg for steg og sjekke etter ubesøkte noder



ALGORITHM: REKURSIVT DYBDE-FØRST SØK

Input: En graf $G = (V, E)$, en startnode u og en mengde `visited` med besøkte noder

Output: Besøker alle noder i G som kan nås fra u nøyaktig én gang

```
1 Procedure DFSVisit(G, u, visited)
2     add u to visited
3     for (u, v) ∈ E do
4         if v ∉ visited then
5             DFSVisit(G, v, visited)
```

u=F

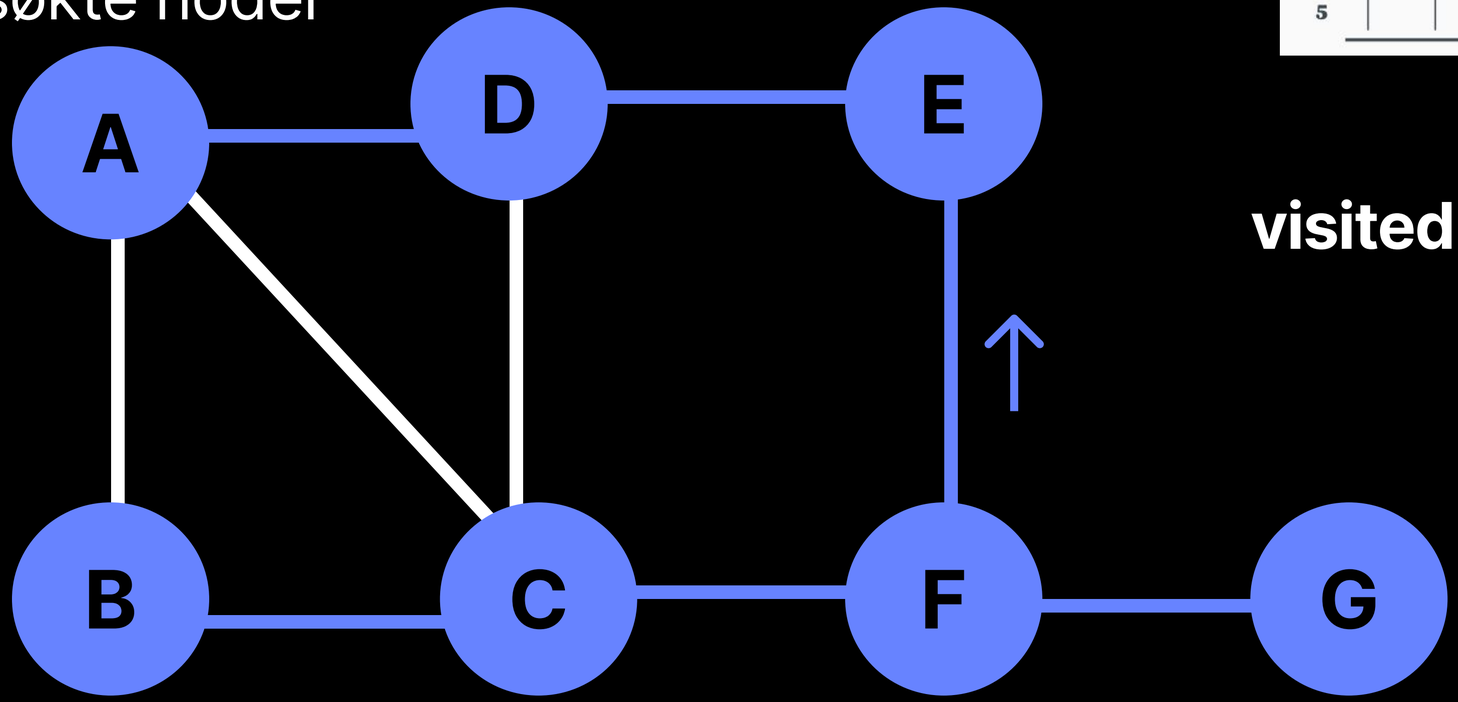
visited A D E F G C B

De rekursive kallene hittil:

DFSVisit(G, A, visited)
DFSVisit(G, D, visited)
DFSVisit(G, E, visited)
DFSVisit(G, F, visited)

Dybde-først søk (DFS)

Til slutt vil algoritmen rekursivt gå tilbake til startnoden steg for steg og sjekke etter ubesøkte noder



ALGORITHM: REKURSIVT DYBDE-FØRST SØK

Input: En graf $G = (V, E)$, en startnode u og en mengde `visited` med besøkte noder

Output: Besøker alle noder i G som kan nås fra u nøyaktig én gang

```
1 Procedure DFSVisit(G, u, visited)
2     add u to visited
3     for (u, v) ∈ E do
4         if v ∉ visited then
5             DFSVisit(G, v, visited)
```

u=E

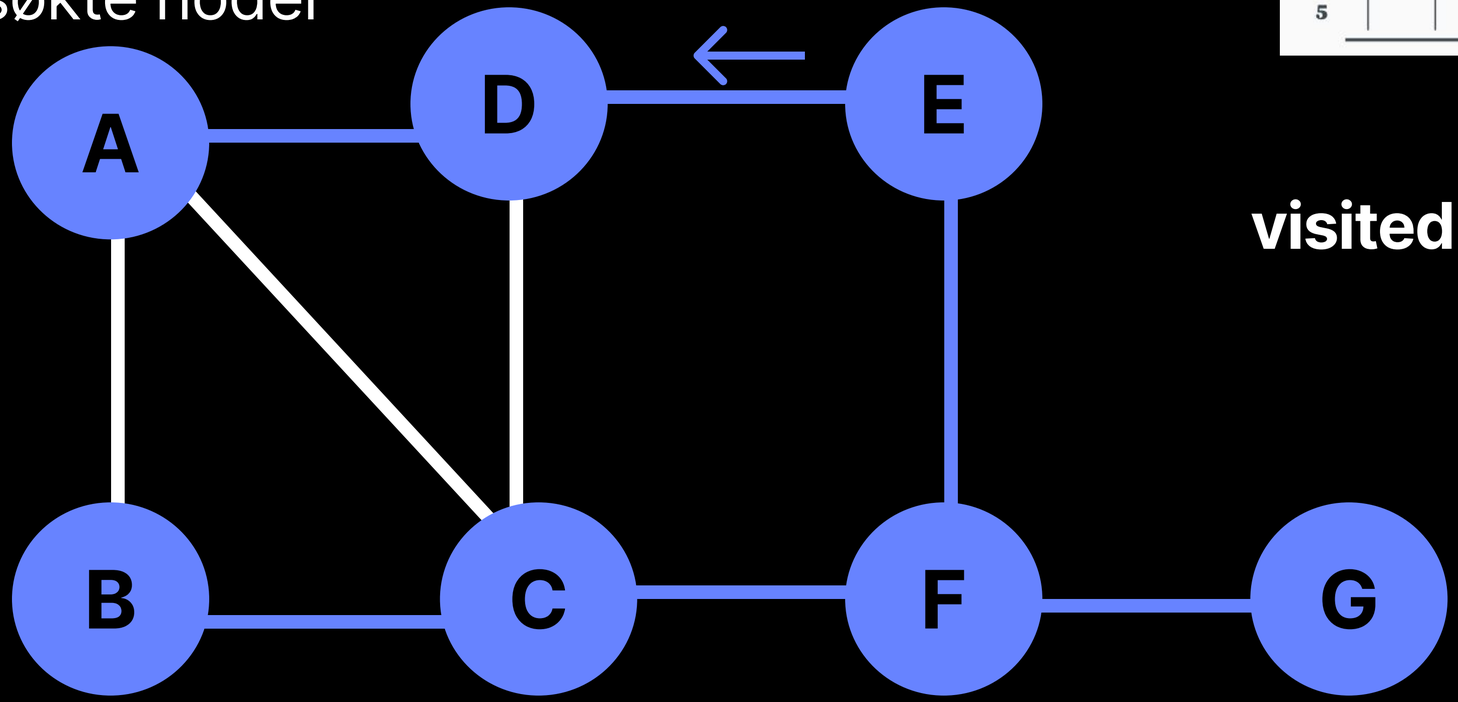
visited A D E F G C B

De rekursive kallene hittil:

DFSVisit(G, A, visited)
DFSVisit(G, D, visited)
DFSVisit(G, E, visited)

Dybde-først søk (DFS)

Til slutt vil algoritmen rekursivt gå tilbake til startnoden steg for steg og sjekke etter ubesøkte noder



ALGORITHM: REKURSIVT DYBDE-FØRST SØK

Input: En graf $G = (V, E)$, en startnode u og en mengde `visited` med besøkte noder

Output: Besøker alle noder i G som kan nås fra u nøyaktig én gang

```
1 Procedure DFSVisit(G, u, visited)
2     add u to visited
3     for (u, v) ∈ E do
4         if v ∉ visited then
5             DFSVisit(G, v, visited)
```

u=D

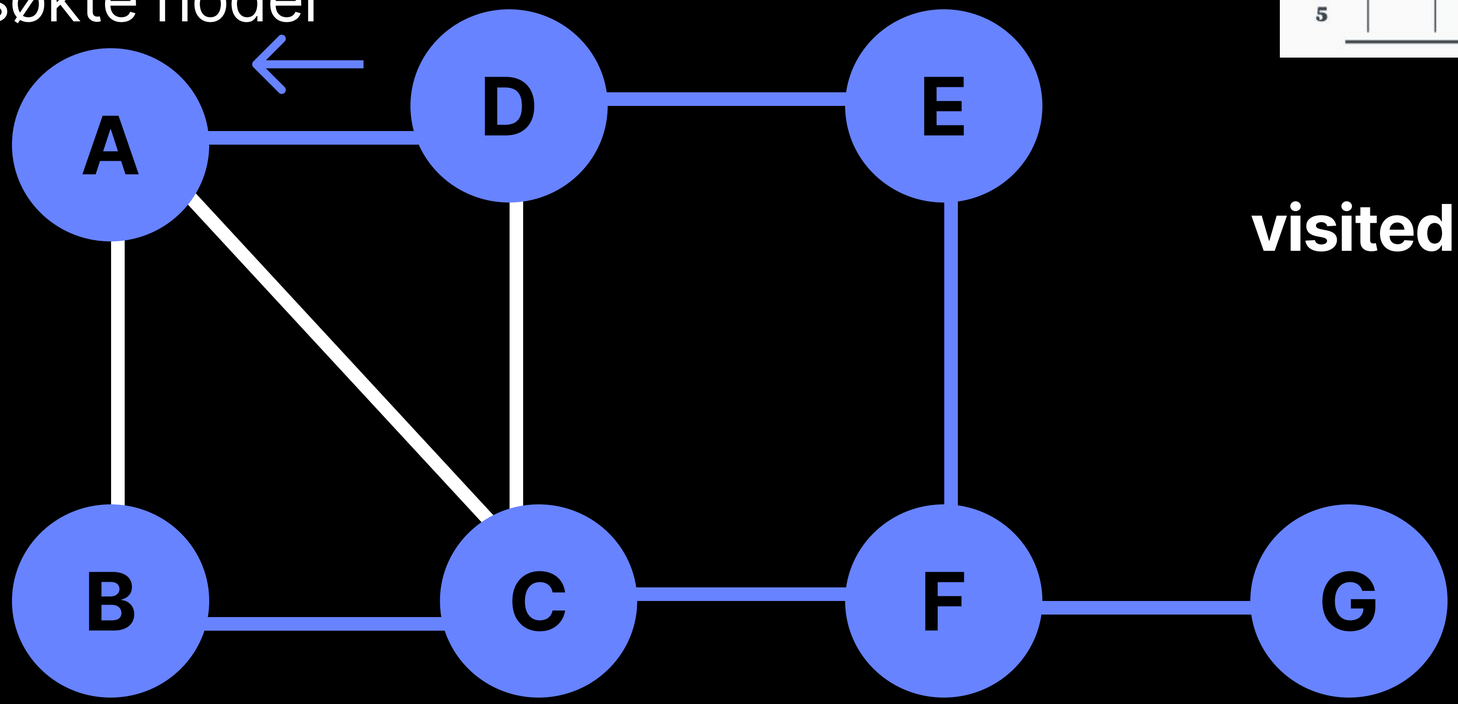
visited A D E F G C B

De rekursive kallene hittil:

DFSVisit(G, A, visited)
  DFSVisit(G, D, visited)

Dybde-først søk (DFS)

Til slutt vil algoritmen rekursivt gå tilbake til startnoden steg for steg og sjekke etter ubesøkte noder



ALGORITHM: REKURSIVT DYBDE-FØRST SØK

Input: En graf $G = (V, E)$, en startnode u og en mengde `visited` med besøkte noder

Output: Besøker alle noder i G som kan nås fra u nøyaktig én gang

```
1 Procedure DFSVisit(G, u, visited)
2     add u to visited
3     for (u, v) ∈ E do
4         if v ∉ visited then
5             DFSVisit(G, v, visited)
```

u=A

visited A D E F G C B

De rekursive kallene hittil:

DFSVisit(G, A, visited)

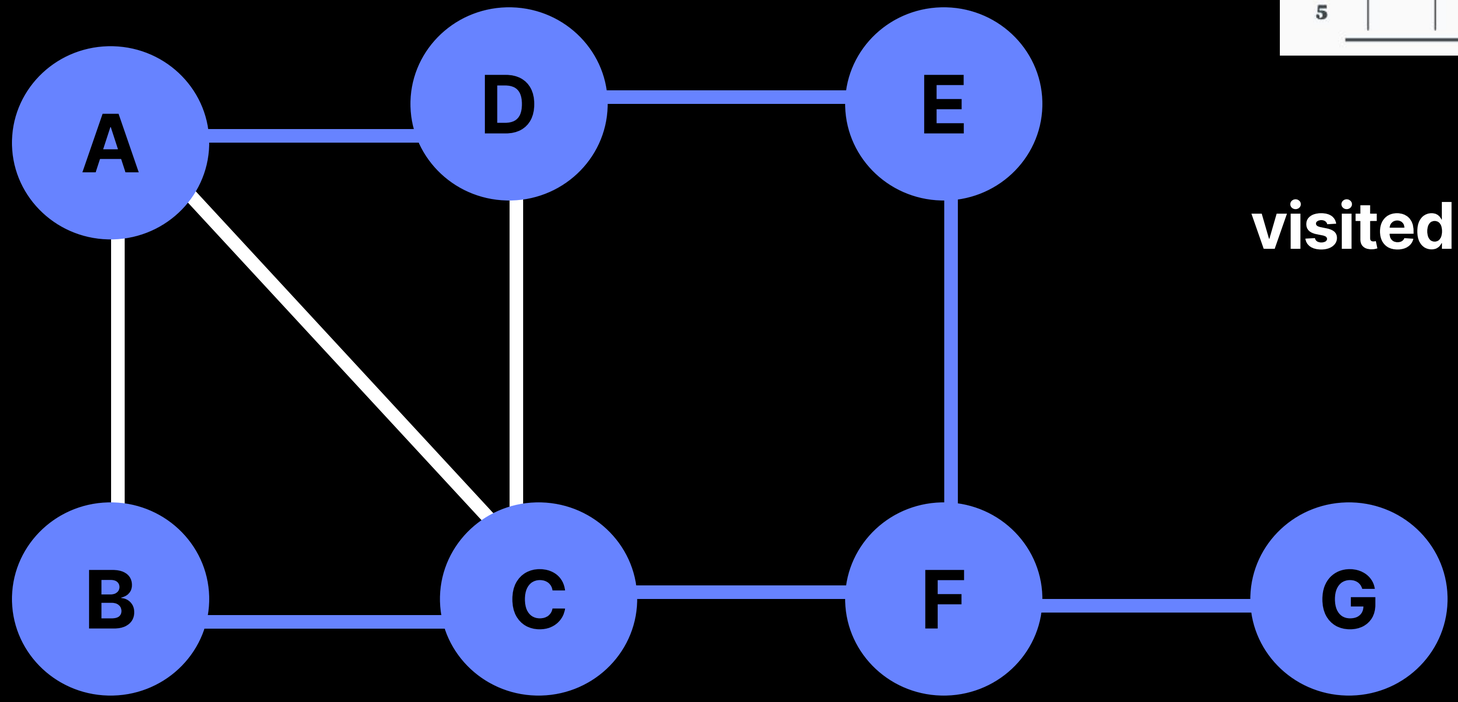
Dybde-først søk (DFS)

ALGORITHM: REKURSIVT DYBDE-FØRST SØK

Input: En graf $G = (V, E)$, en startnode u og en mengde `visited` med besøkte noder

Output: Besøker alle noder i G som kan nås fra u nøyaktig én gang

```
1 Procedure DFSVisit(G, u, visited)
2     add u to visited
3     for (u, v) ∈ E do
4         if v ∉ visited then
5             DFSVisit(G, v, visited)
```



visited A D E F G C B

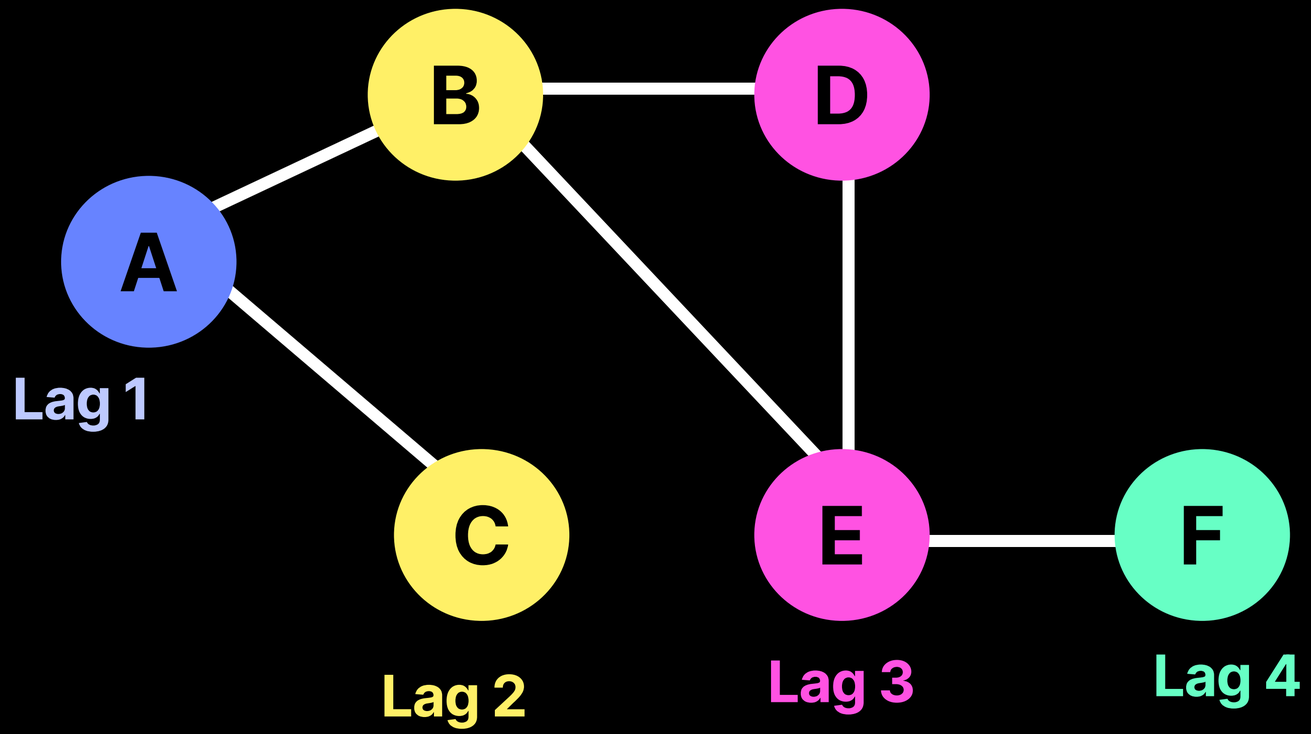
De rekursive kallene hittil:

Bredde-først søk (BFS)

Gitt en startnode, beveger vi oss lag for lag til alle ubesøkte nabonoder.

Beveger seg altså i bredden.

Gitt at A er start noden er dette lagenen:



Kan blant annet brukes til å finne korsteste sti i en uvekta graf

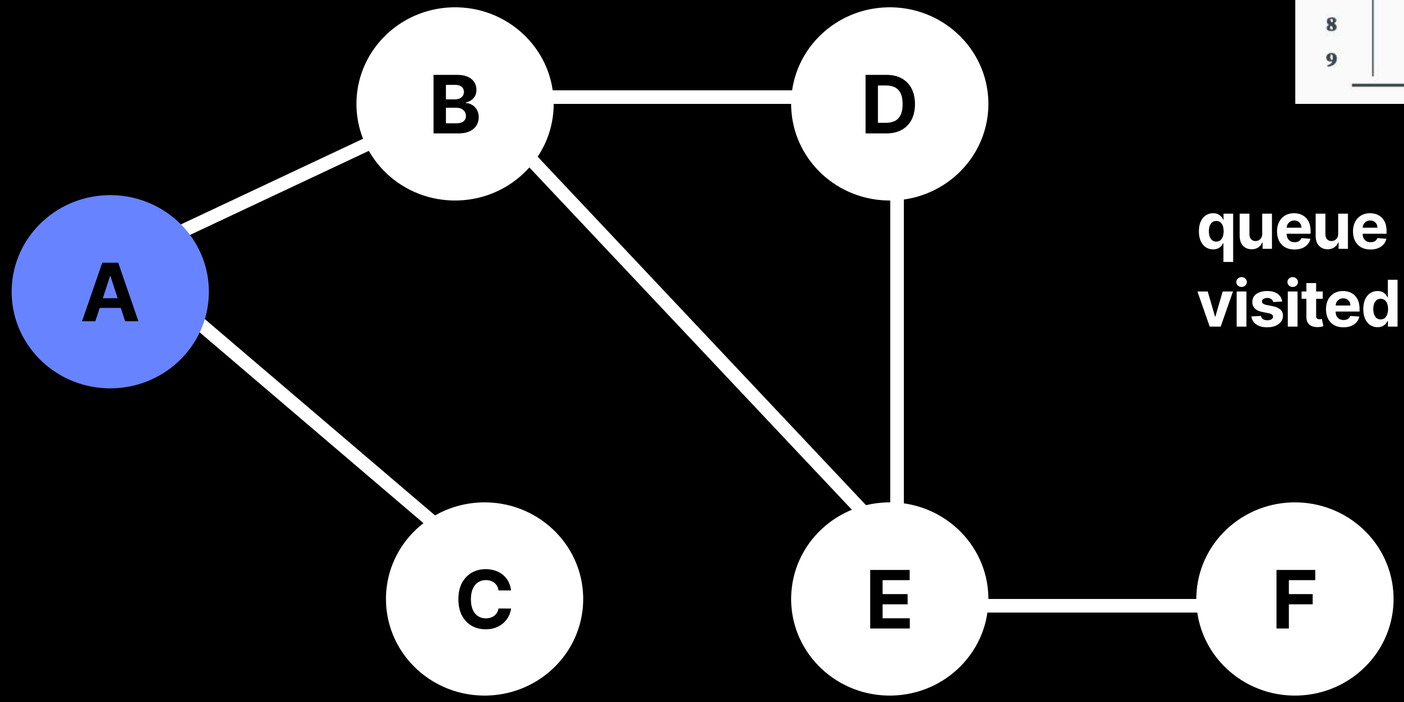
Bredde først søk (BFS)

ALGORITHM: BREDDE-FØRST SØK

Input: En graf $G = (V, E)$ og en startnode s, og en mengde `visited` med besøkte noder

Output: Besøker alle noder i G som kan nås fra s nøyaktig én gang

```
1 Procedure BFSVisit(G, s, visited)
2     add s to visited
3     queue ← queue containing s
4     while queue is not empty do
5         u ← queue.dequeue()
6         for (u, v) ∈ E do
7             if v ∉ visited then
8                 add v to visited
9                 queue.enqueue(v)
```



queue A
visited A

dequeue: fjerner første fra køen
enqueue: legger til bakerst i køen

Bredde først søk (BFS)

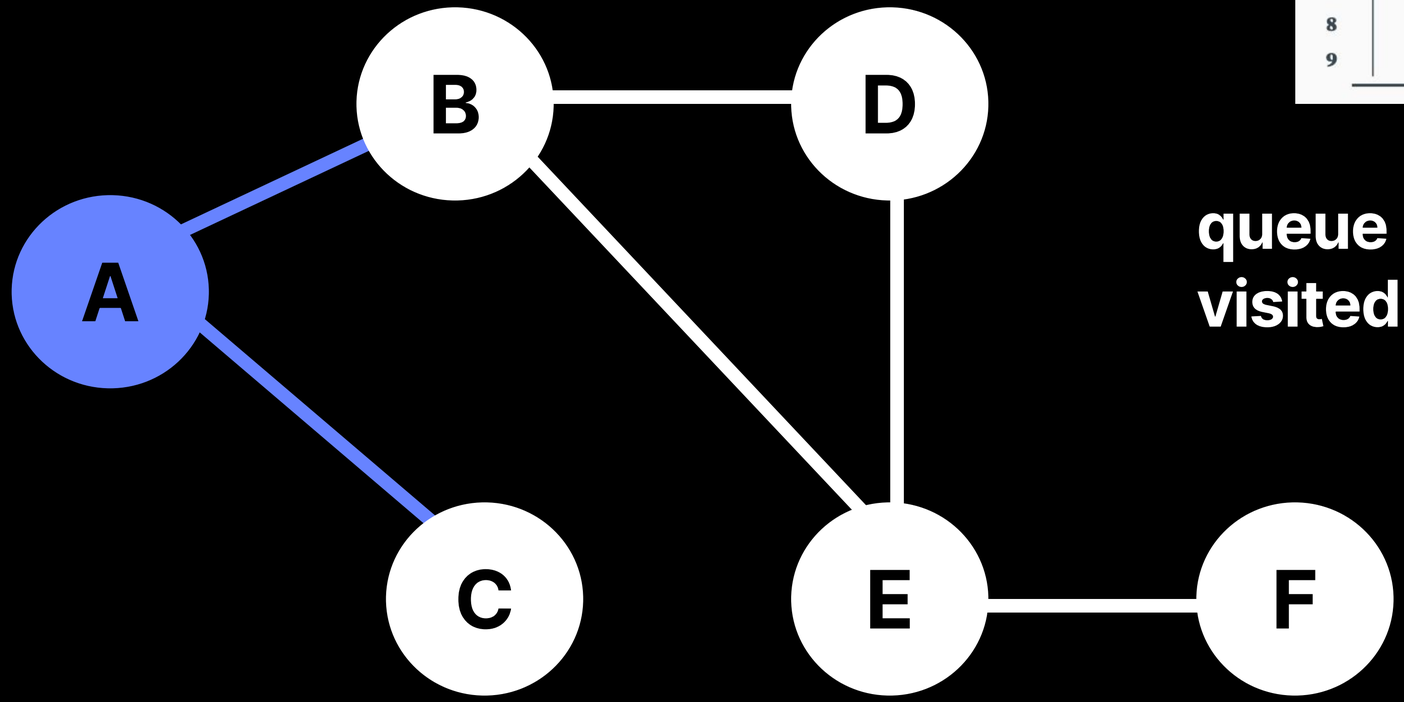
ALGORITHM: BREDDE-FØRST SØK

Input: En graf $G = (V, E)$ og en startnode s, og en mengde `visited` med besøkte noder

Output: Besøker alle noder i G som kan nås fra s nøyaktig én gang

```
1 Procedure BFSVisit(G, s, visited)
2     add s to visited
3     queue ← queue containing s
4     while queue is not empty do
5         u ← queue.dequeue()
6         for (u, v) ∈ E do
7             if v ∉ visited then
8                 add v to visited
9                 queue.enqueue(v)
```

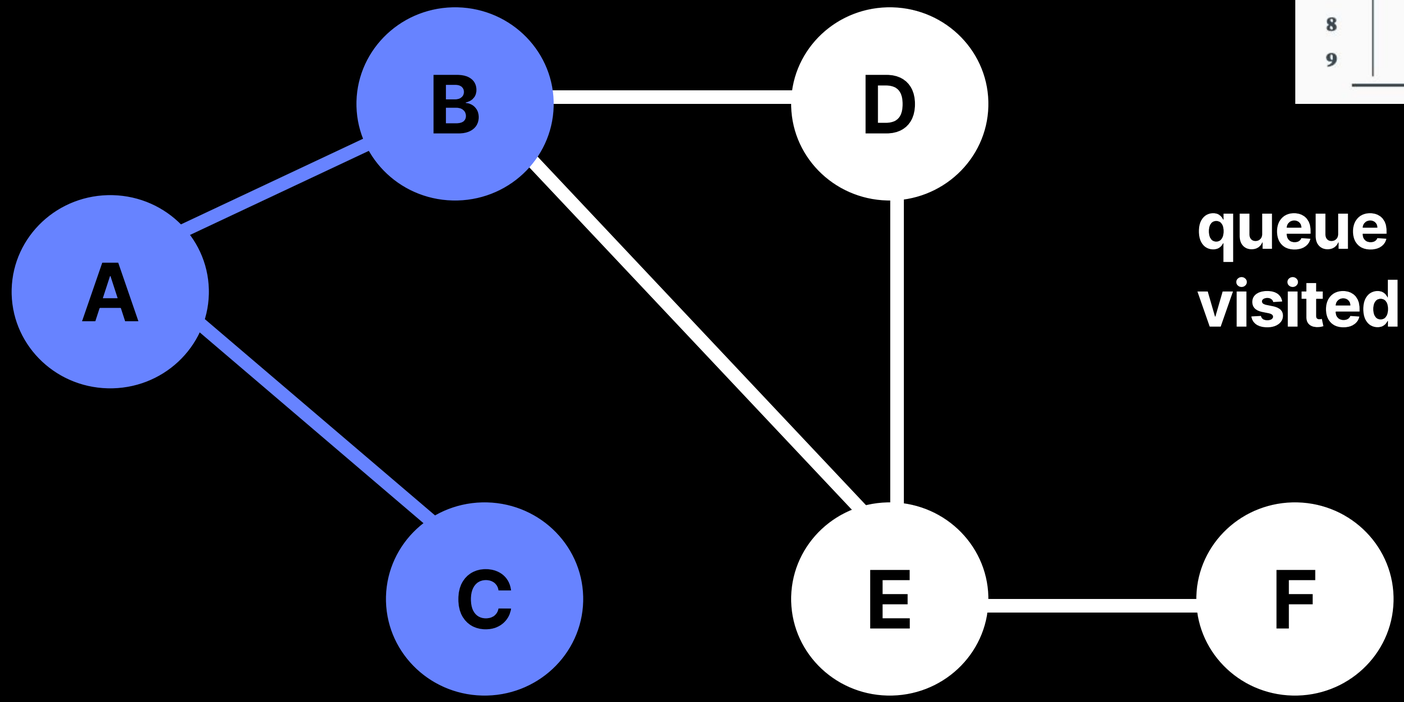
u=A



queue
visited A

dequeue: fjerner første fra køen
enqueue: legger til bakerst i køen

Bredde først søk (BFS)



ALGORITHM: BREDDE-FØRST SØK

Input: En graf $G = (V, E)$ og en startnode s, og en mengde `visited` med besøkte noder

Output: Besøker alle noder i G som kan nås fra s nøyaktig én gang

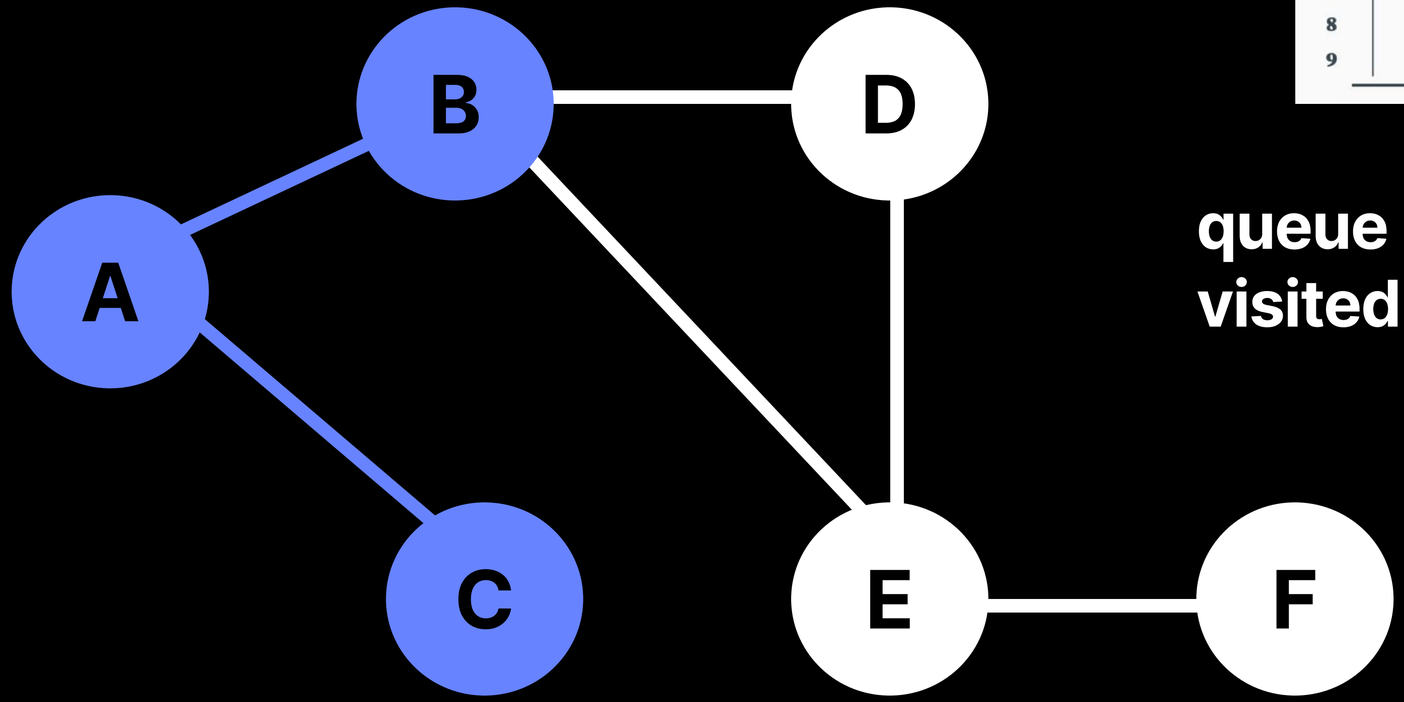
```
1 Procedure BFSVisit(G, s, visited)
2     add s to visited
3     queue ← queue containing s
4     while queue is not empty do
5         u ← queue.dequeue()
6         for (u, v) ∈ E do
7             if v ∉ visited then
8                 add v to visited
9                 queue.enqueue(v)
```

u=A

queue B C
visited A B C

dequeue: fjerner første fra køen
enqueue: legger til bakerst i køen

Bredde først søk (BFS)



ALGORITHM: BREDDE-FØRST SØK

Input: En graf $G = (V, E)$ og en startnode s, og en mengde `visited` med besøkte noder

Output: Besøker alle noder i G som kan nås fra s nøyaktig én gang

```
1 Procedure BFSVisit(G, s, visited)
2     add s to visited
3     queue ← queue containing s
4     while queue is not empty do
5         u ← queue.dequeue()
6         for (u, v) ∈ E do
7             if v ∉ visited then
8                 add v to visited
9                 queue.enqueue(v)
```

u=B

queue C
visited A B C

dequeue: fjerner første fra køen
enqueue: legger til bakerst i køen

Bredde først søk (BFS)

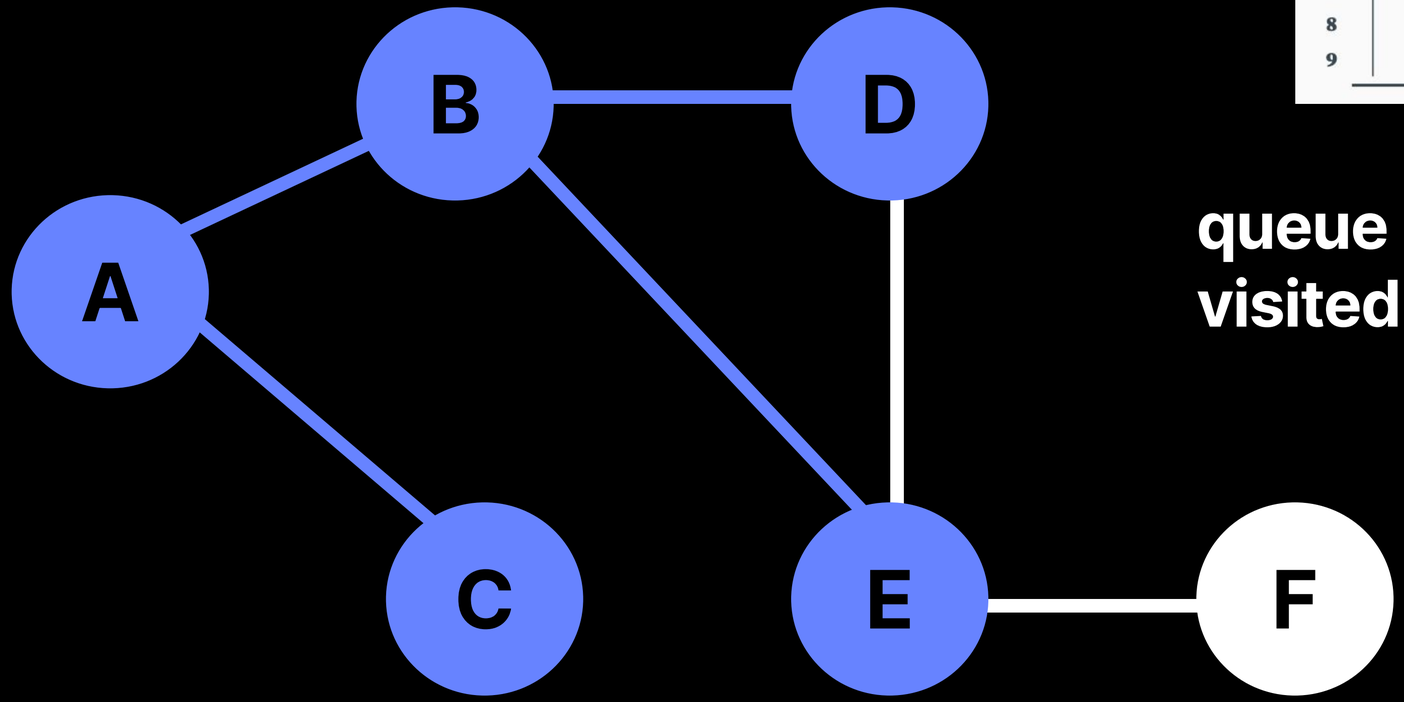
ALGORITHM: BREDDE-FØRST SØK

Input: En graf $G = (V, E)$ og en startnode s, og en mengde `visited` med besøkte noder

Output: Besøker alle noder i G som kan nås fra s nøyaktig én gang

```
1 Procedure BFSVisit(G, s, visited)
2     add s to visited
3     queue ← queue containing s
4     while queue is not empty do
5         u ← queue.dequeue()
6         for (u, v) ∈ E do
7             if v ∉ visited then
8                 add v to visited
9                 queue.enqueue(v)
```

u=B



queue C D E
visited A B C D E

dequeue: fjerner første fra køen
enqueue: legger til bakerst i køen

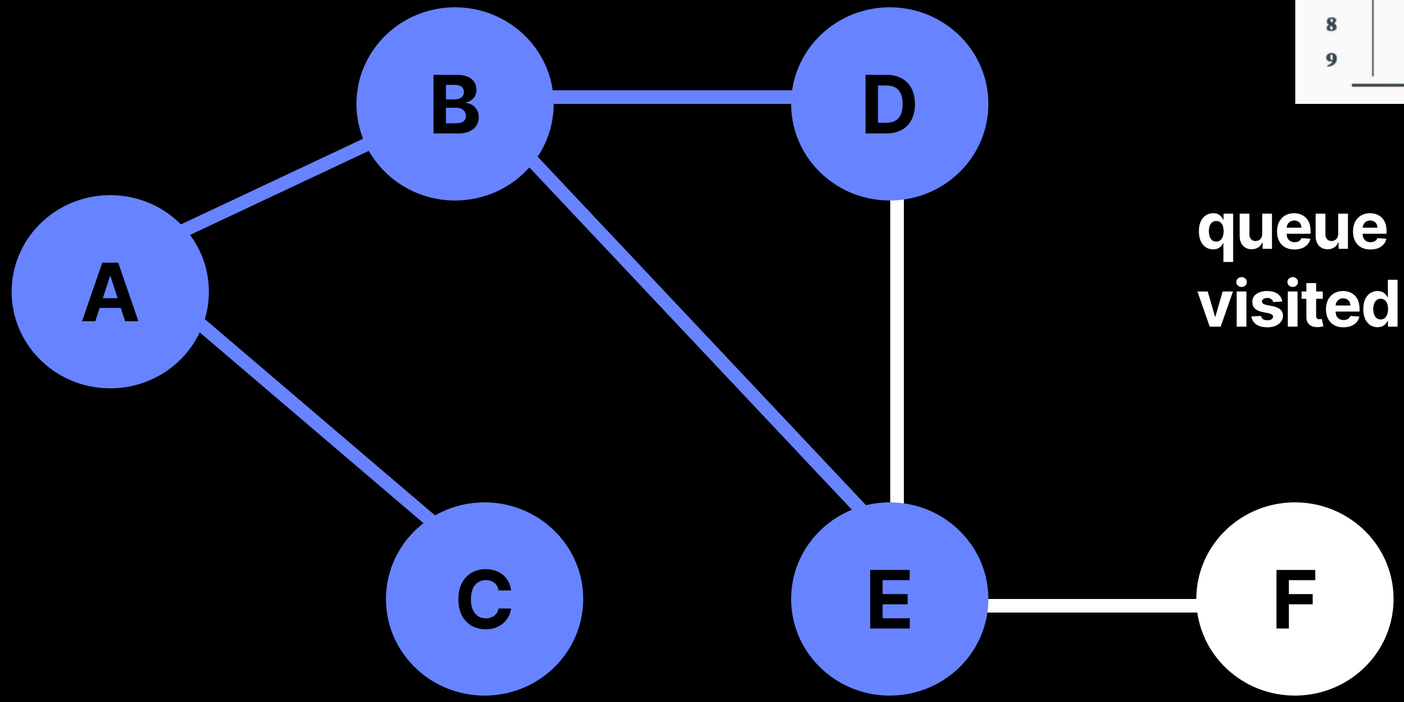
Bredde først søk (BFS)

ALGORITHM: BREDDE-FØRST SØK

Input: En graf $G = (V, E)$ og en startnode s, og en mengde `visited` med besøkte noder

Output: Besøker alle noder i G som kan nås fra s nøyaktig én gang

```
1 Procedure BFSVisit(G, s, visited)
2     add s to visited
3     queue ← queue containing s
4     while queue is not empty do
5         u ← queue.dequeue()
6         for (u, v) ∈ E do
7             if v ∉ visited then
8                 add v to visited
9                 queue.enqueue(v)
```



u=C

queue D E
visited A B C D E

dequeue: fjerner første fra køen
enqueue: legger til bakerst i køen

Bredde først søk (BFS)

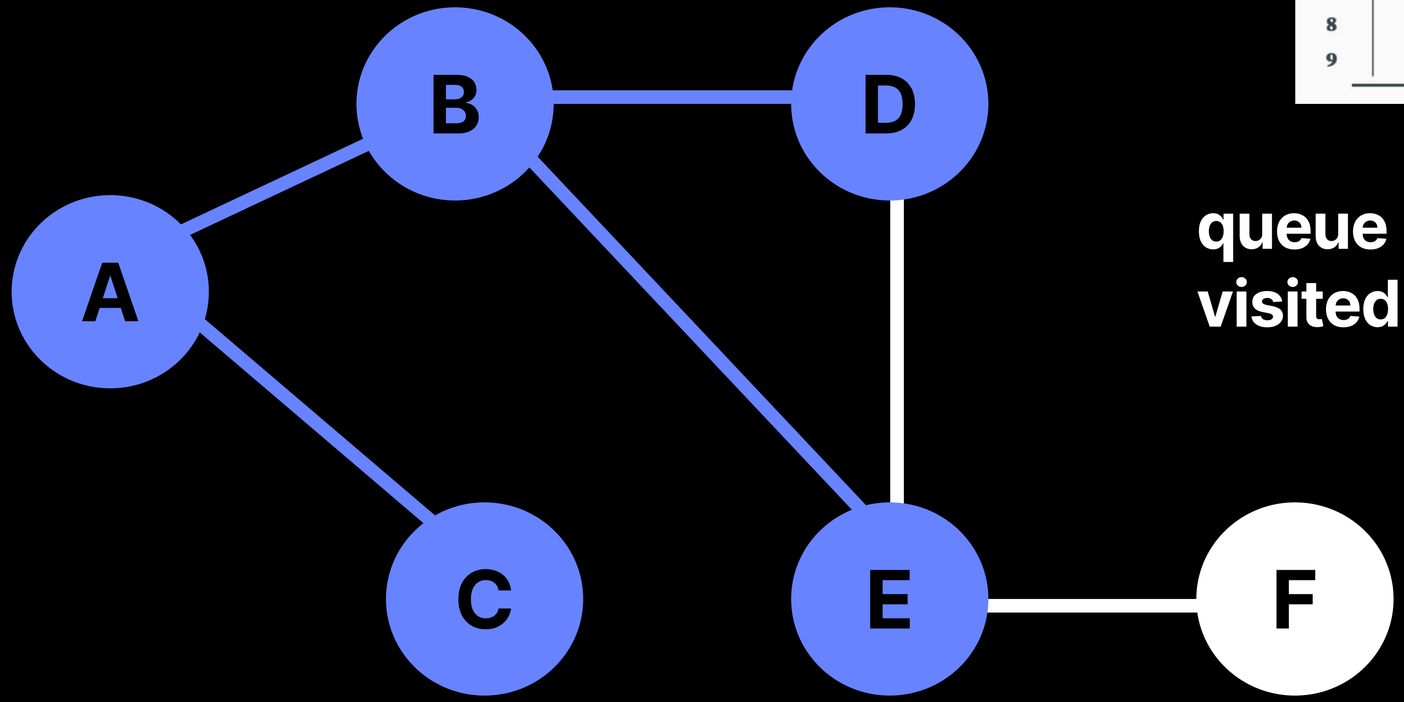
ALGORITHM: BREDDE-FØRST SØK

Input: En graf $G = (V, E)$ og en startnode s, og en mengde `visited` med besøkte noder

Output: Besøker alle noder i G som kan nås fra s nøyaktig én gang

```
1 Procedure BFSVisit(G, s, visited)
2     add s to visited
3     queue ← queue containing s
4     while queue is not empty do
5         u ← queue.dequeue()
6         for (u, v) ∈ E do
7             if v ∉ visited then
8                 add v to visited
9                 queue.enqueue(v)
```

u=C



queue D E
visited A B C D E

dequeue: fjerner første fra køen
enqueue: legger til bakerst i køen

Bredde først søk (BFS)

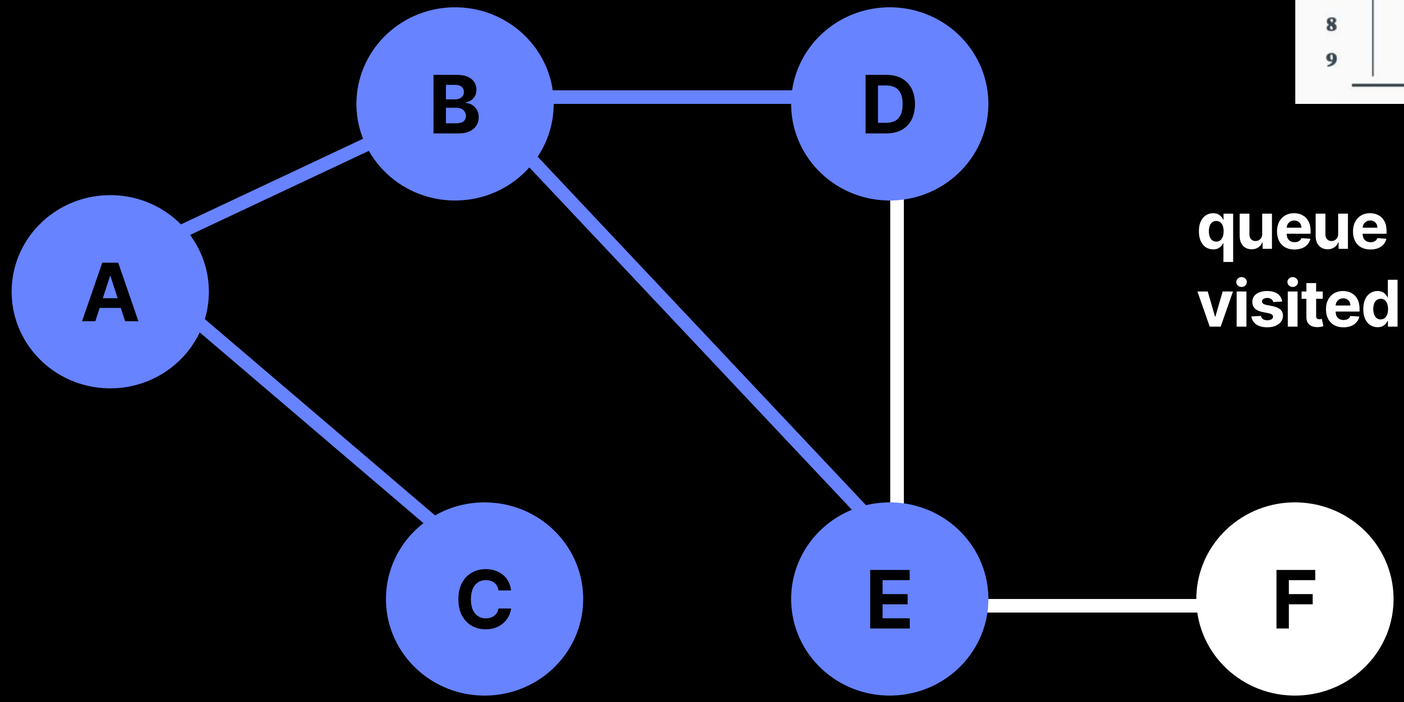
ALGORITHM: BREDDE-FØRST SØK

Input: En graf $G = (V, E)$ og en startnode s, og en mengde `visited` med besøkte noder

Output: Besøker alle noder i G som kan nås fra s nøyaktig én gang

```
1 Procedure BFSVisit(G, s, visited)
2     add s to visited
3     queue ← queue containing s
4     while queue is not empty do
5         u ← queue.dequeue()
6         for (u, v) ∈ E do
7             if v ∉ visited then
8                 add v to visited
9                 queue.enqueue(v)
```

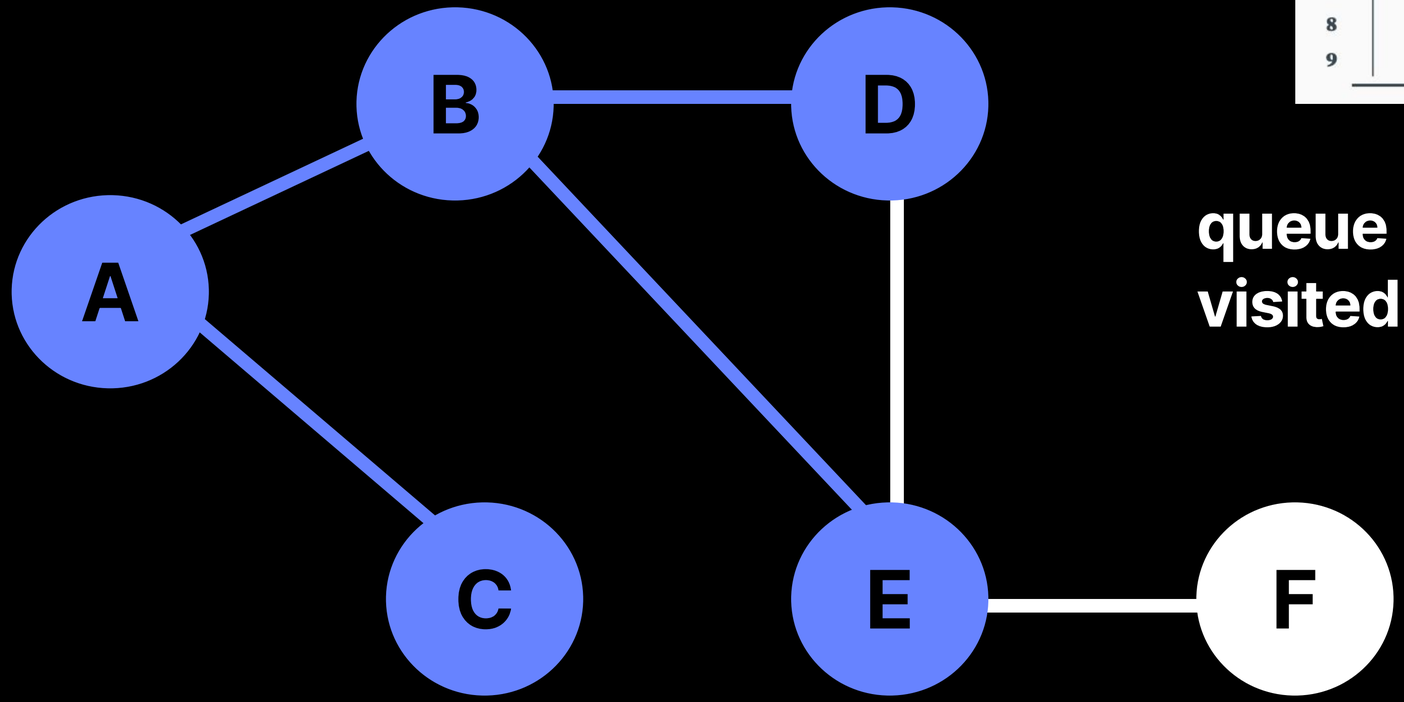
u=D



queue E
visited A B C D E

dequeue: fjerner første fra køen
enqueue: legger til bakerst i køen

Bredde først søk (BFS)



ALGORITHM: BREDDE-FØRST SØK

Input: En graf $G = (V, E)$ og en startnode s, og en mengde `visited` med besøkte noder

Output: Besøker alle noder i G som kan nås fra s nøyaktig én gang

```
1 Procedure BFSVisit(G, s, visited)
2     add s to visited
3     queue ← queue containing s
4     while queue is not empty do
5         u ← queue.dequeue()
6         for (u, v) ∈ E do
7             if v ∉ visited then
8                 add v to visited
9                 queue.enqueue(v)
```

u=D

queue E
visited A B C D E

dequeue: fjerner første fra køen
enqueue: legger til bakerst i køen

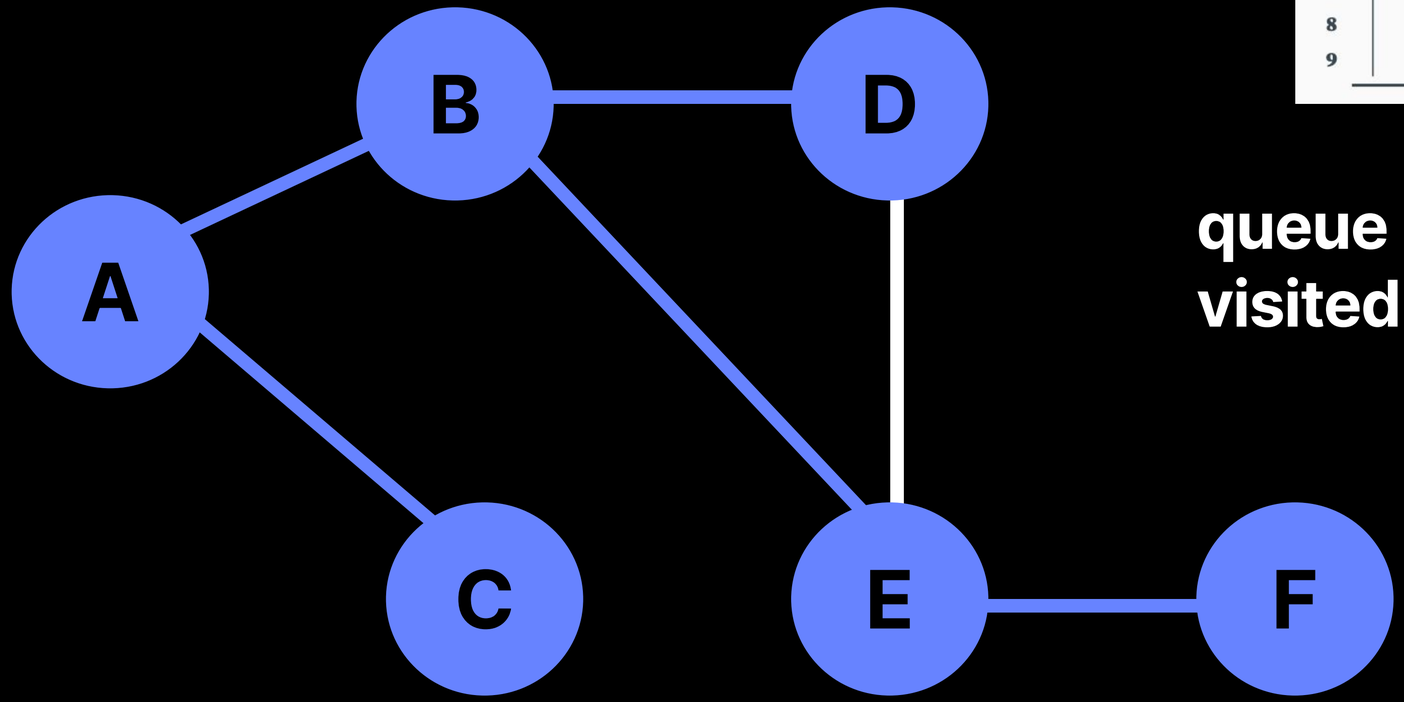
Bredde først søk (BFS)

ALGORITHM: BREDDE-FØRST SØK

Input: En graf $G = (V, E)$ og en startnode s, og en mengde `visited` med besøkte noder

Output: Besøker alle noder i G som kan nås fra s nøyaktig én gang

```
1 Procedure BFSVisit(G, s, visited)
2     add s to visited
3     queue ← queue containing s
4     while queue is not empty do
5         u ← queue.dequeue()
6         for (u, v) ∈ E do
7             if v ∉ visited then
8                 add v to visited
9                 queue.enqueue(v)
```



u=E

queue
visited A B C D E

dequeue: fjerner første fra køen
enqueue: legger til bakerst i køen

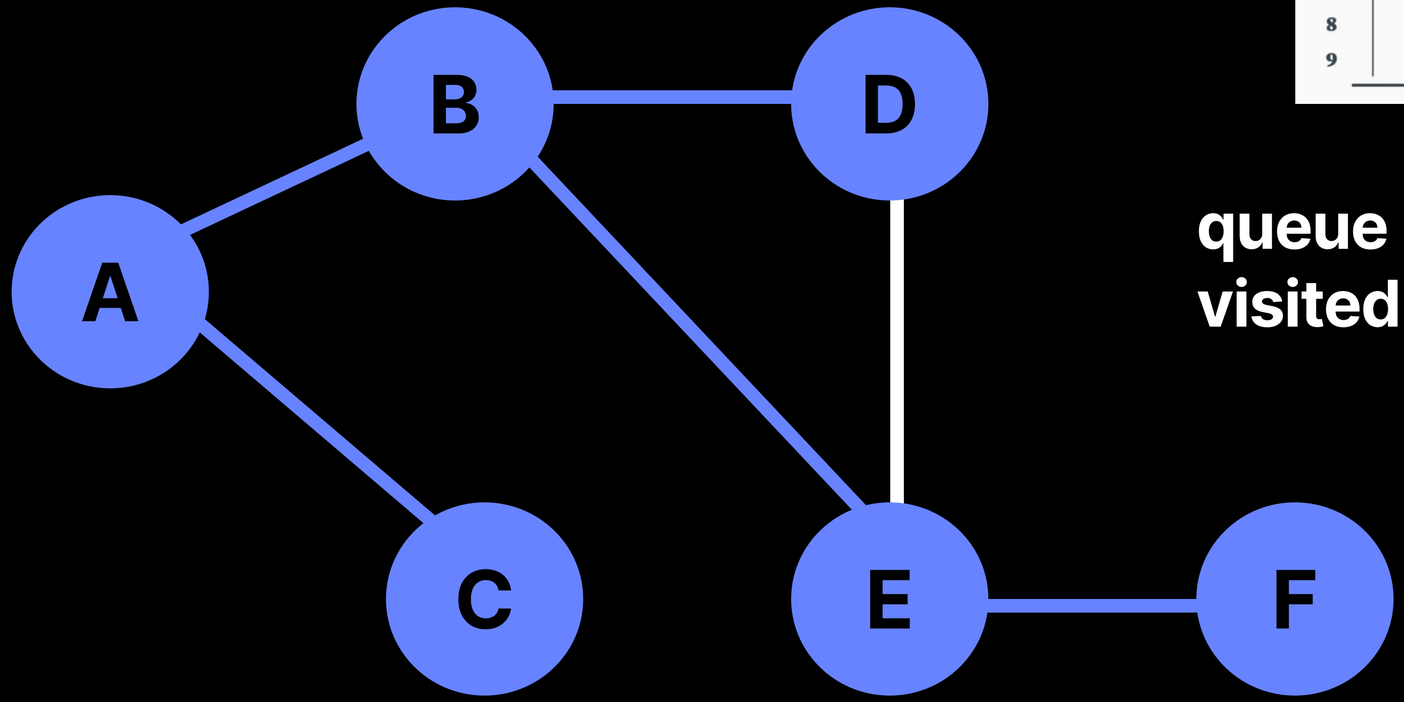
Bredde først søk (BFS)

ALGORITHM: BREDDE-FØRST SØK

Input: En graf $G = (V, E)$ og en startnode s, og en mengde `visited` med besøkte noder

Output: Besøker alle noder i G som kan nås fra s nøyaktig én gang

```
1 Procedure BFSVisit(G, s, visited)
2     add s to visited
3     queue ← queue containing s
4     while queue is not empty do
5         u ← queue.dequeue()
6         for (u, v) ∈ E do
7             if v ∉ visited then
8                 add v to visited
9                 queue.enqueue(v)
```



u=E

queue F
visited A B C D E F

dequeue: fjerner første fra køen
enqueue: legger til bakerst i køen

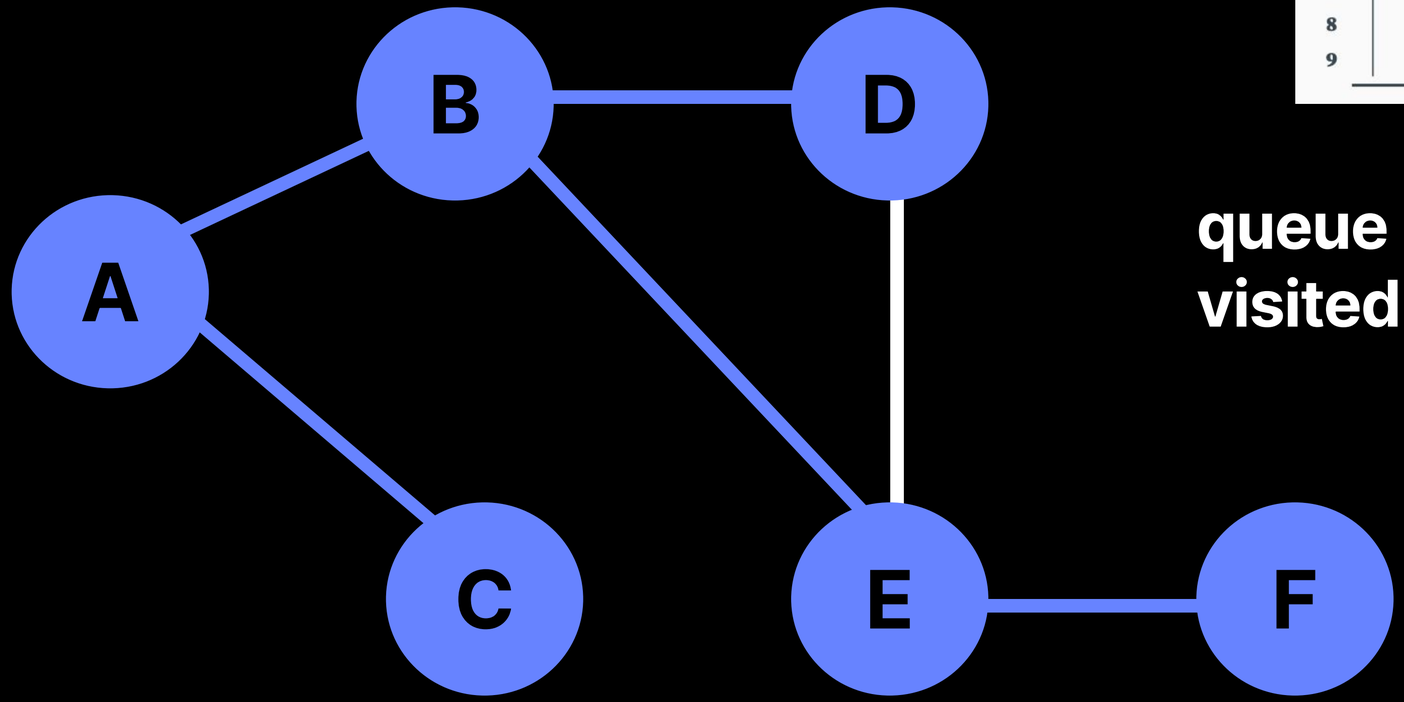
Bredde først søk (BFS)

ALGORITHM: BREDDE-FØRST SØK

Input: En graf $G = (V, E)$ og en startnode s, og en mengde `visited` med besøkte noder

Output: Besøker alle noder i G som kan nås fra s nøyaktig én gang

```
1 Procedure BFSVisit(G, s, visited)
2     add s to visited
3     queue ← queue containing s
4     while queue is not empty do
5         u ← queue.dequeue()
6         for (u, v) ∈ E do
7             if v ∉ visited then
8                 add v to visited
9                 queue.enqueue(v)
```



u=F

queue
visited A B C D E F

dequeue: fjerner første fra køen
enqueue: legger til bakerst i køen

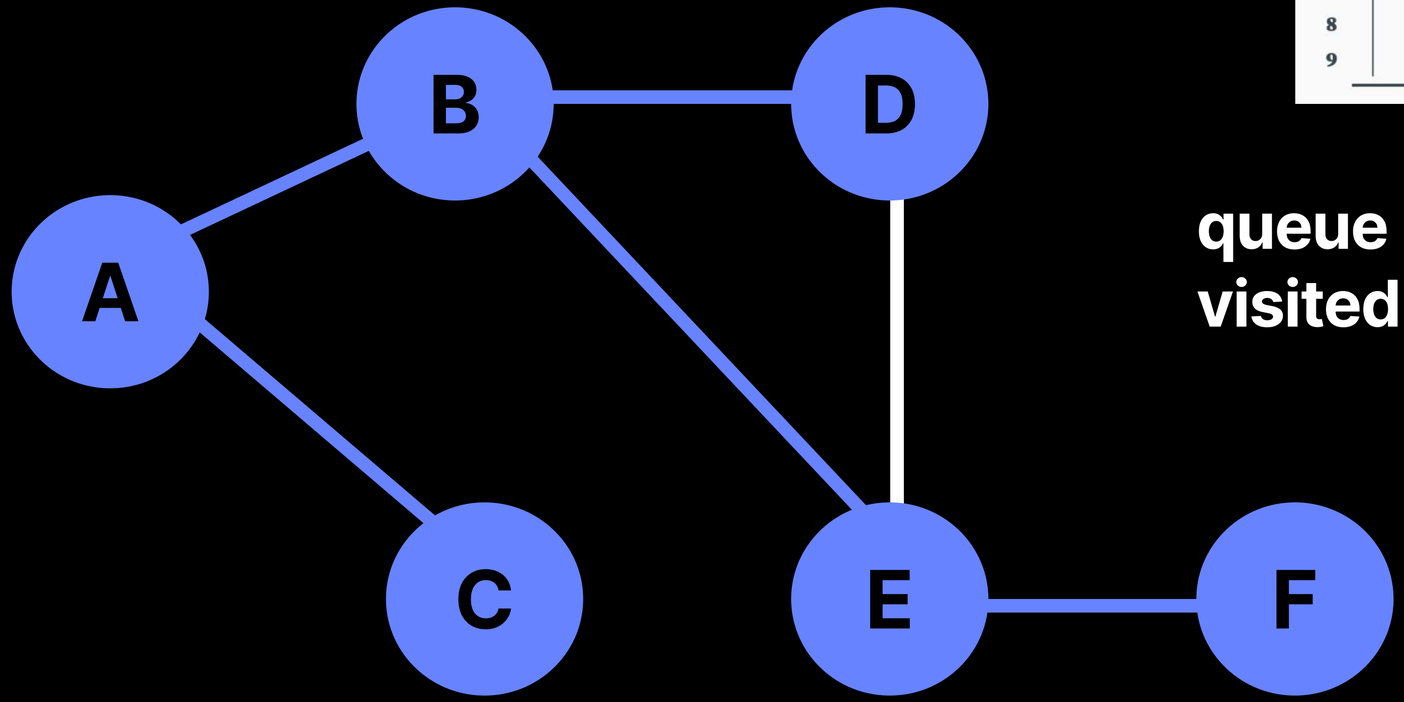
Bredde først søk (BFS)

ALGORITHM: BREDDE-FØRST SØK

Input: En graf $G = (V, E)$ og en startnode s, og en mengde `visited` med besøkte noder

Output: Besøker alle noder i G som kan nås fra s nøyaktig én gang

```
1 Procedure BFSVisit(G, s, visited)
2     add s to visited
3     queue ← queue containing s
4     while queue is not empty do
5         u ← queue.dequeue()
6         for (u, v) ∈ E do
7             if v ∉ visited then
8                 add v to visited
9                 queue.enqueue(v)
```



u=F

queue
visited A B C D E F

dequeue: fjerner første fra køen
enqueue: legger til bakerst i køen

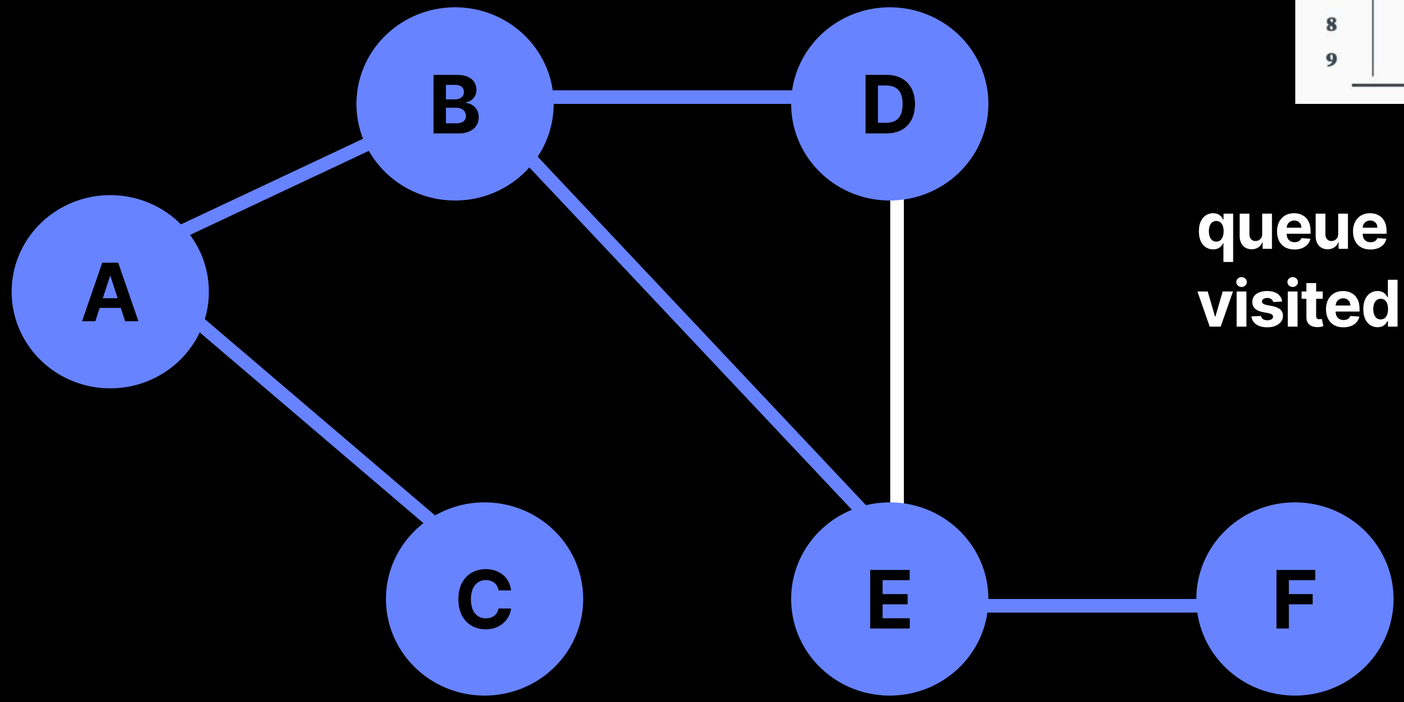
Bredde først søk (BFS)

ALGORITHM: BREDDE-FØRST SØK

Input: En graf $G = (V, E)$ og en startnode s, og en mengde `visited` med besøkte noder

Output: Besøker alle noder i G som kan nås fra s nøyaktig én gang

```
1 Procedure BFSVisit(G, s, visited)
2     add s to visited
3     queue ← queue containing s
4     while queue is not empty do
5         u ← queue.dequeue()
6         for (u, v) ∈ E do
7             if v ∉ visited then
8                 add v to visited
9                 queue.enqueue(v)
```



u=F

queue
visited A B C D E F

dequeue: fjerner første fra køen
enqueue: legger til bakerst i køen

Topologisk sortering (TopSort)

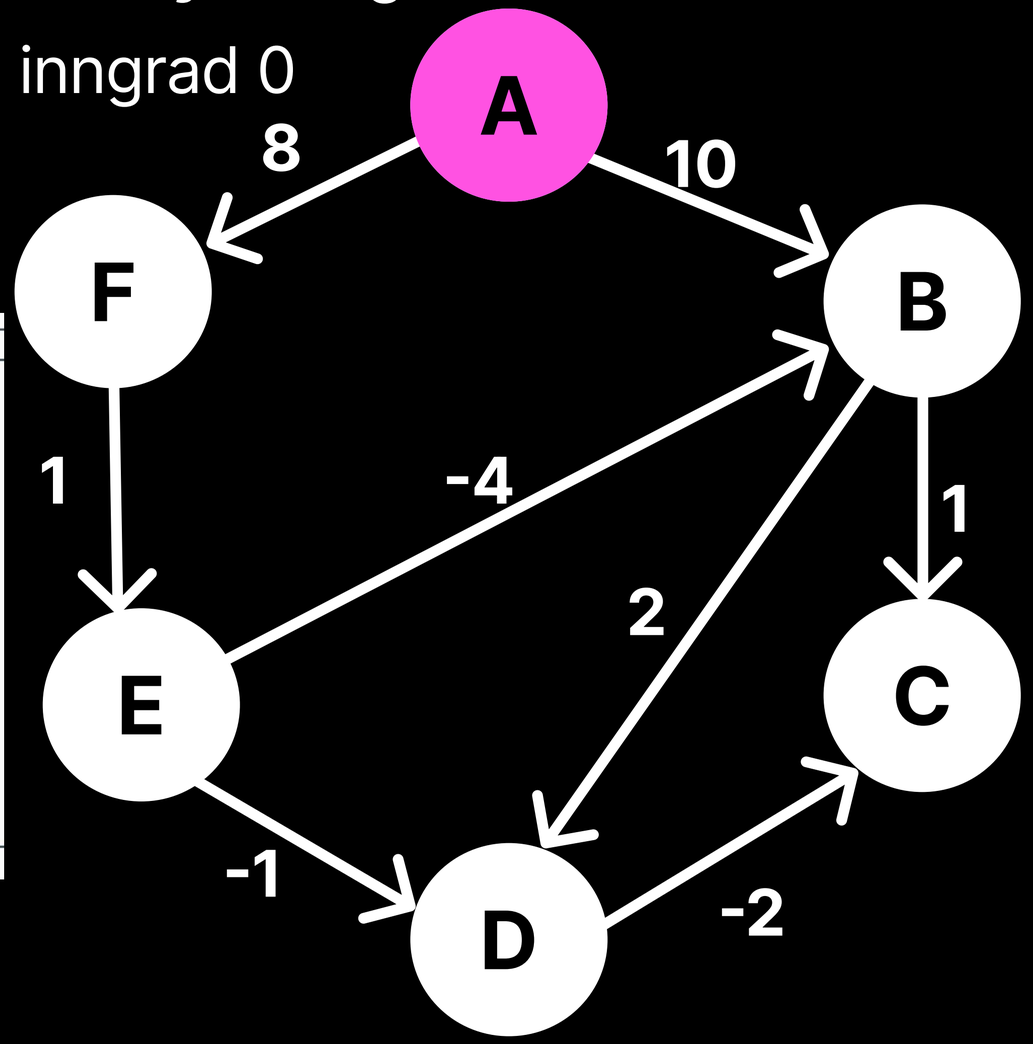
Ordner nodene i en **rettet asyklisk** graf (DAG).

Starter i noden(e) med inngrad 0

ALGORITHM: TOPOLOGISK SORTERING

Input: En rettet graf $G = (V, E)$
Output: En topologisk ordning av nodene i G

```
1  Procedure TopSort(G)
2      stack ← empty stack
3      output ← empty list
4      for v ∈ V do
5          if indegree of v is 0 then
6              stack.push(v)
7      while stack is not empty do
8          u ← stack.pop()
9          append u to output
10         for (u, v) ∈ E do
11             remove u from the incoming edges of v
12             if indegree of v is 0 then
13                 add v to stack
14     if |output| < |V| then
15         error G contains a cycle and cannot be topologically ordered
16     return output
```



Output:

A

Stack

Topologisk sortering (TopSort)

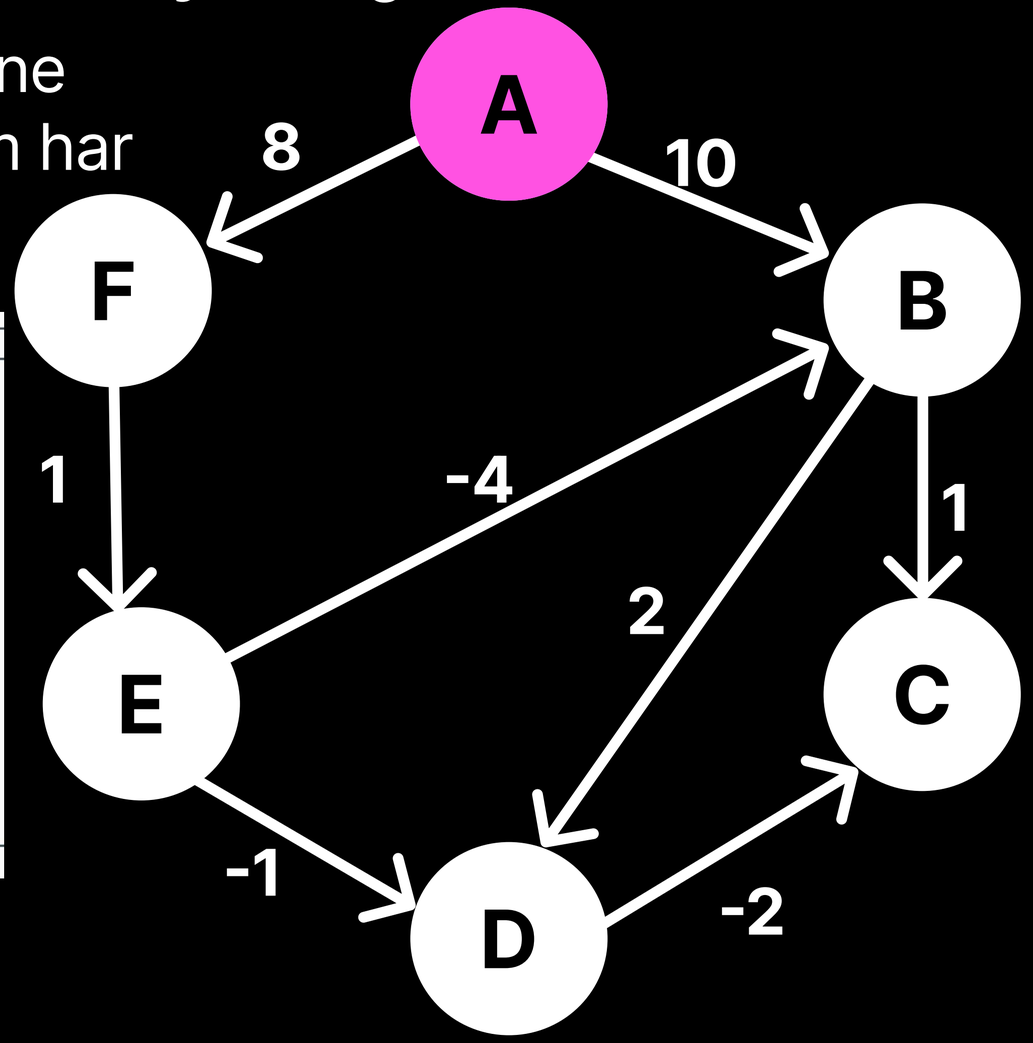
Ordner nodene i en **rettet asyklisk** graf (DAG).

Fjerner kantene til denne sjekker igjen hvem som har inngrad 0 nå

ALGORITHM: TOPOLOGISK SORTERING

Input: En rettet graf $G = (V, E)$
Output: En topologisk ordning av nodene i G

```
1  Procedure TopSort(G)
2      stack ← empty stack
3      output ← empty list
4      for v ∈ V do
5          if indegree of v is 0 then
6              stack.push(v)
7      while stack is not empty do
8          u ← stack.pop()
9          append u to output
10         for (u, v) ∈ E do
11             remove u from the incoming edges of v
12             if indegree of v is 0 then
13                 add v to stack
14     if |output| < |V| then
15         error G contains a cycle and cannot be topologically ordered
16     return output
```



Output: A

Stack

Topologisk sortering (TopSort)

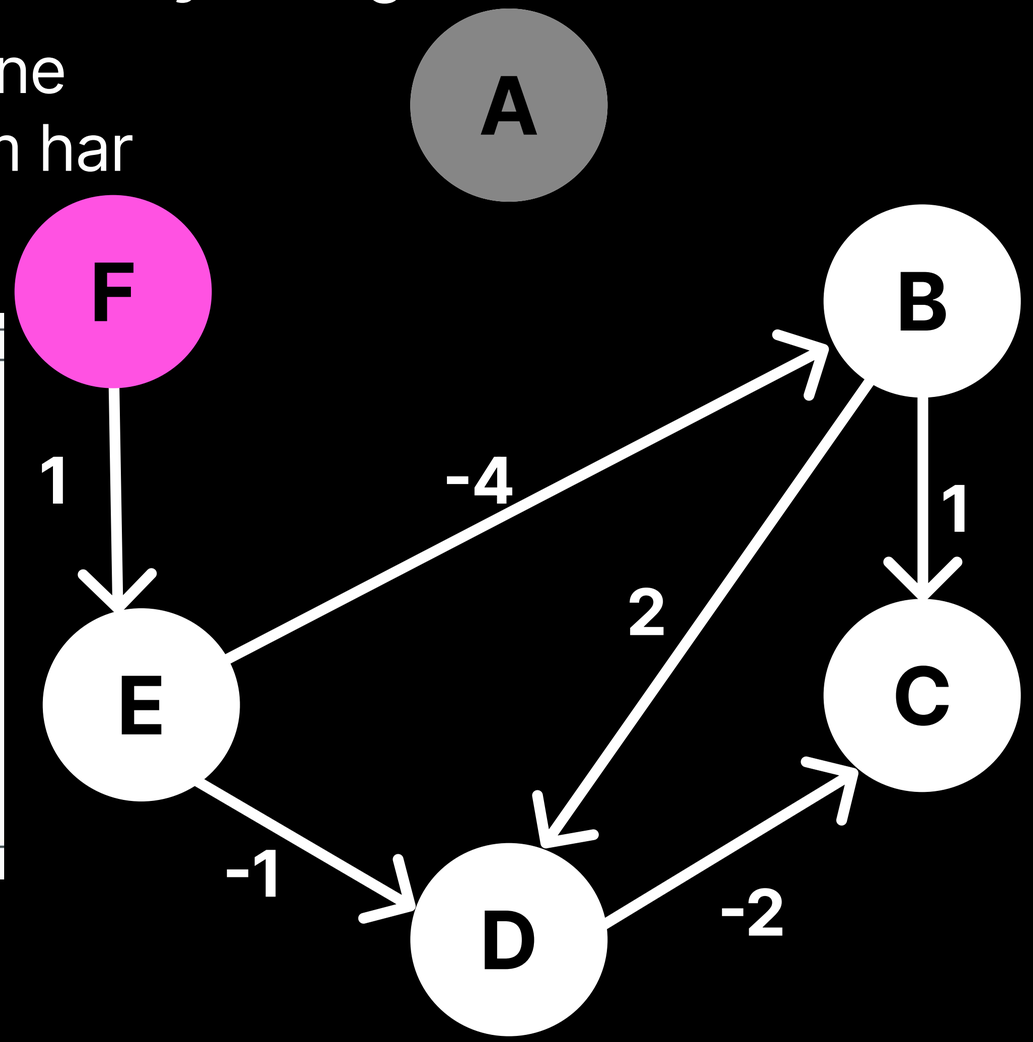
Ordner nodene i en **rettet asyklisk** graf (DAG).

Fjerner kantene til denne sjekker igjen hvem som har inngrad 0 nå

ALGORITHM: TOPOLOGISK SORTERING

Input: En rettet graf $G = (V, E)$
Output: En topologisk ordning av nodene i G

```
1  Procedure TopSort(G)
2      stack ← empty stack
3      output ← empty list
4      for v ∈ V do
5          if indegree of v is 0 then
6              stack.push(v)
7      while stack is not empty do
8          u ← stack.pop()
9          append u to output
10         for (u, v) ∈ E do
11             remove u from the incoming edges of v
12             if indegree of v is 0 then
13                 add v to stack
14     if |output| < |V| then
15         error G contains a cycle and cannot be topologically ordered
16     return output
```



Output: A

F

Stack

Topologisk sortering (TopSort)

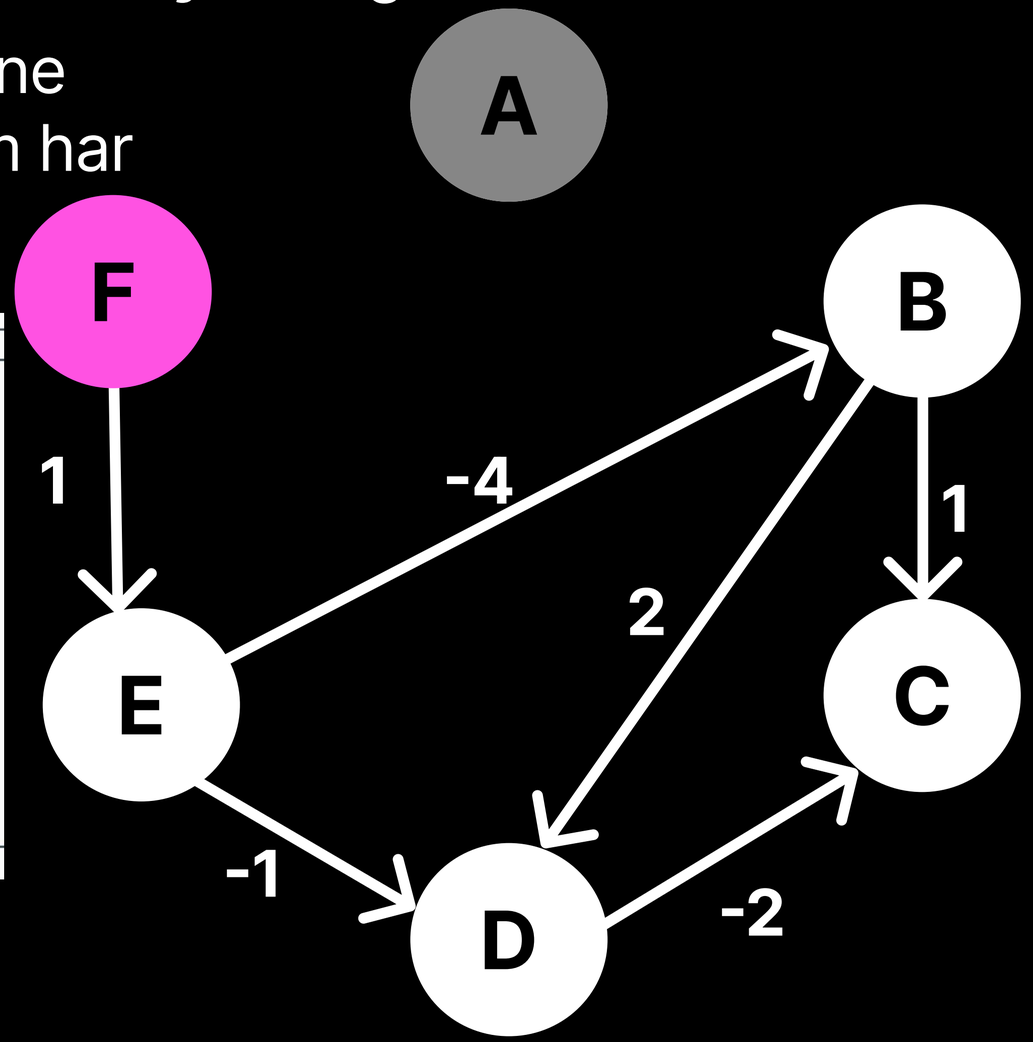
Ordner nodene i en **rettet asyklisk** graf (DAG).

Fjerner kantene til denne sjekker igjen hvem som har inngrad 0 nå

ALGORITHM: TOPOLOGISK SORTERING

Input: En rettet graf $G = (V, E)$
Output: En topologisk ordning av nodene i G

```
1  Procedure TopSort(G)
2      stack ← empty stack
3      output ← empty list
4      for v ∈ V do
5          if indegree of v is 0 then
6              stack.push(v)
7      while stack is not empty do
8          u ← stack.pop()
9          append u to output
10         for (u, v) ∈ E do
11             remove u from the incoming edges of v
12             if indegree of v is 0 then
13                 add v to stack
14     if |output| < |V| then
15         error G contains a cycle and cannot be topologically ordered
16     return output
```



Output: A F

Stack

Topologisk sortering (TopSort)

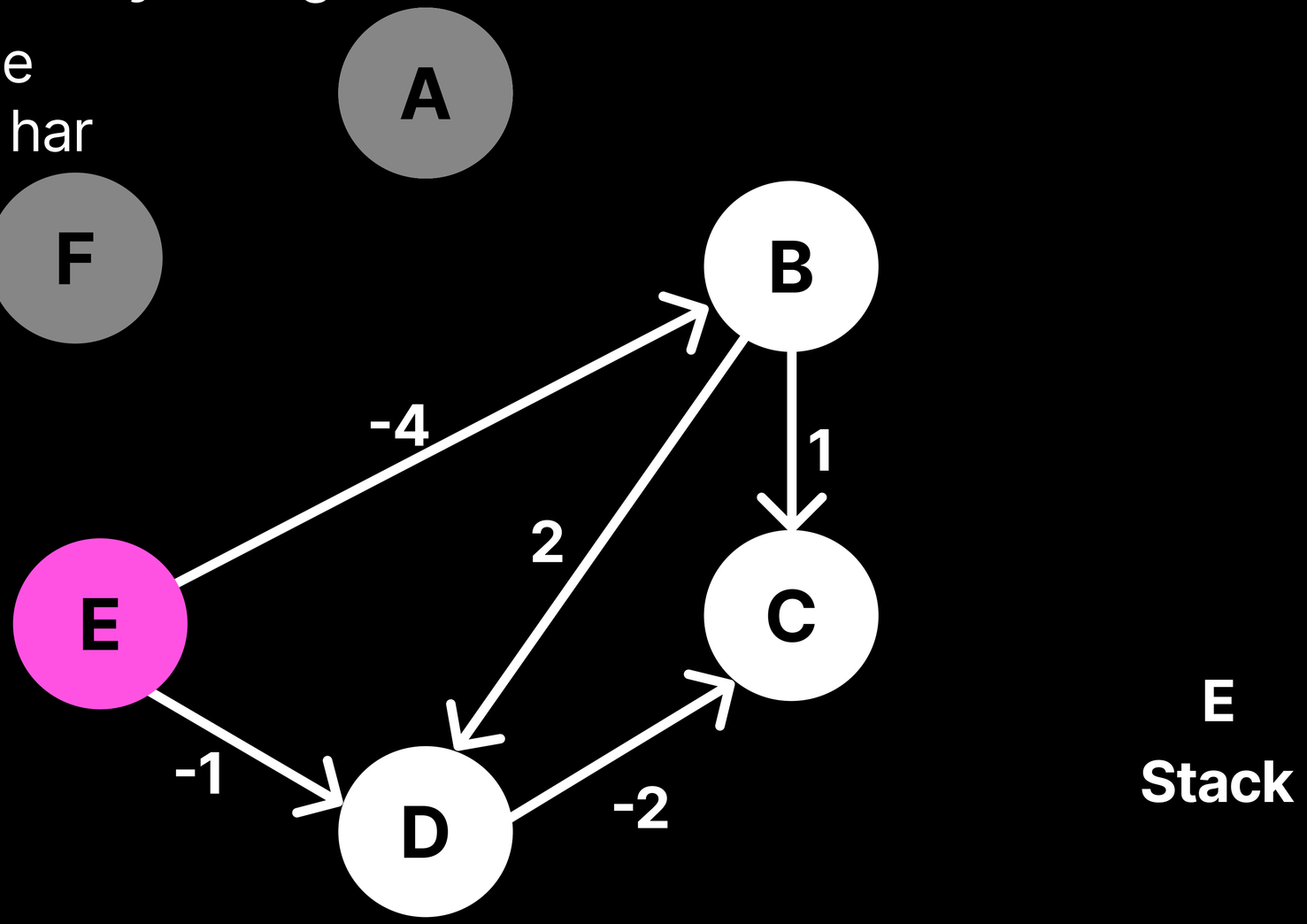
Ordner nodene i en **rettet asyklisk** graf (DAG).

Fjerner kantene til denne sjekker igjen hvem som har inngrad 0 nå

ALGORITHM: TOPOLOGISK SORTERING

Input: En rettet graf $G = (V, E)$
Output: En topologisk ordning av nodene i G

```
1  Procedure TopSort(G)
2      stack ← empty stack
3      output ← empty list
4      for v ∈ V do
5          if indegree of v is 0 then
6              stack.push(v)
7      while stack is not empty do
8          u ← stack.pop()
9          append u to output
10         for (u, v) ∈ E do
11             remove u from the incoming edges of v
12             if indegree of v is 0 then
13                 add v to stack
14     if |output| < |V| then
15         error G contains a cycle and cannot be topologically ordered
16     return output
```



Output: A F

E

Stack

Topologisk sortering (TopSort)

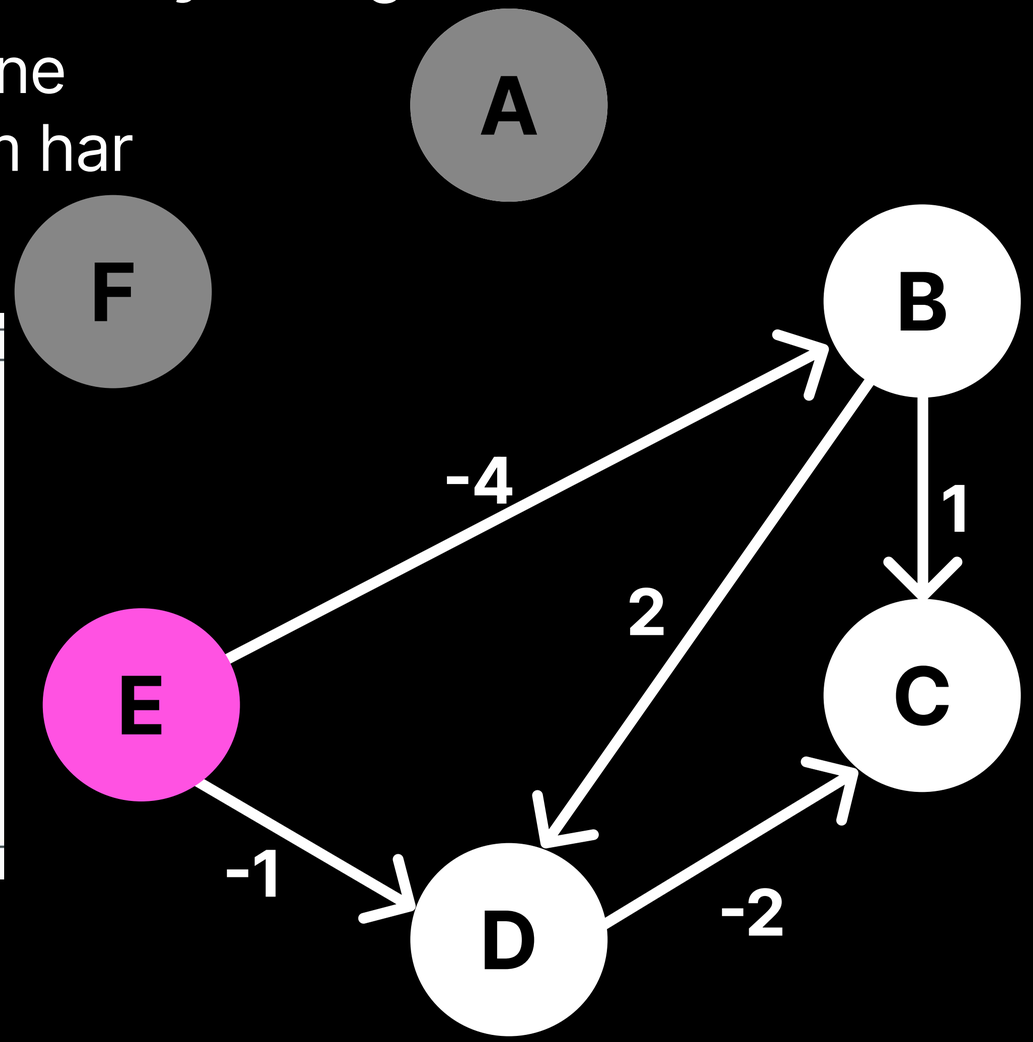
Ordner nodene i en **rettet asyklisk** graf (DAG).

Fjerner kantene til denne sjekker igjen hvem som har inngrad 0 nå

ALGORITHM: TOPOLOGISK SORTERING

Input: En rettet graf $G = (V, E)$
Output: En topologisk ordning av nodene i G

```
1  Procedure TopSort(G)
2      stack ← empty stack
3      output ← empty list
4      for v ∈ V do
5          if indegree of v is 0 then
6              stack.push(v)
7      while stack is not empty do
8          u ← stack.pop()
9          append u to output
10         for (u, v) ∈ E do
11             remove u from the incoming edges of v
12             if indegree of v is 0 then
13                 add v to stack
14     if |output| < |V| then
15         error G contains a cycle and cannot be topologically ordered
16     return output
```



Output: A F E

Stack

Topologisk sortering (TopSort)

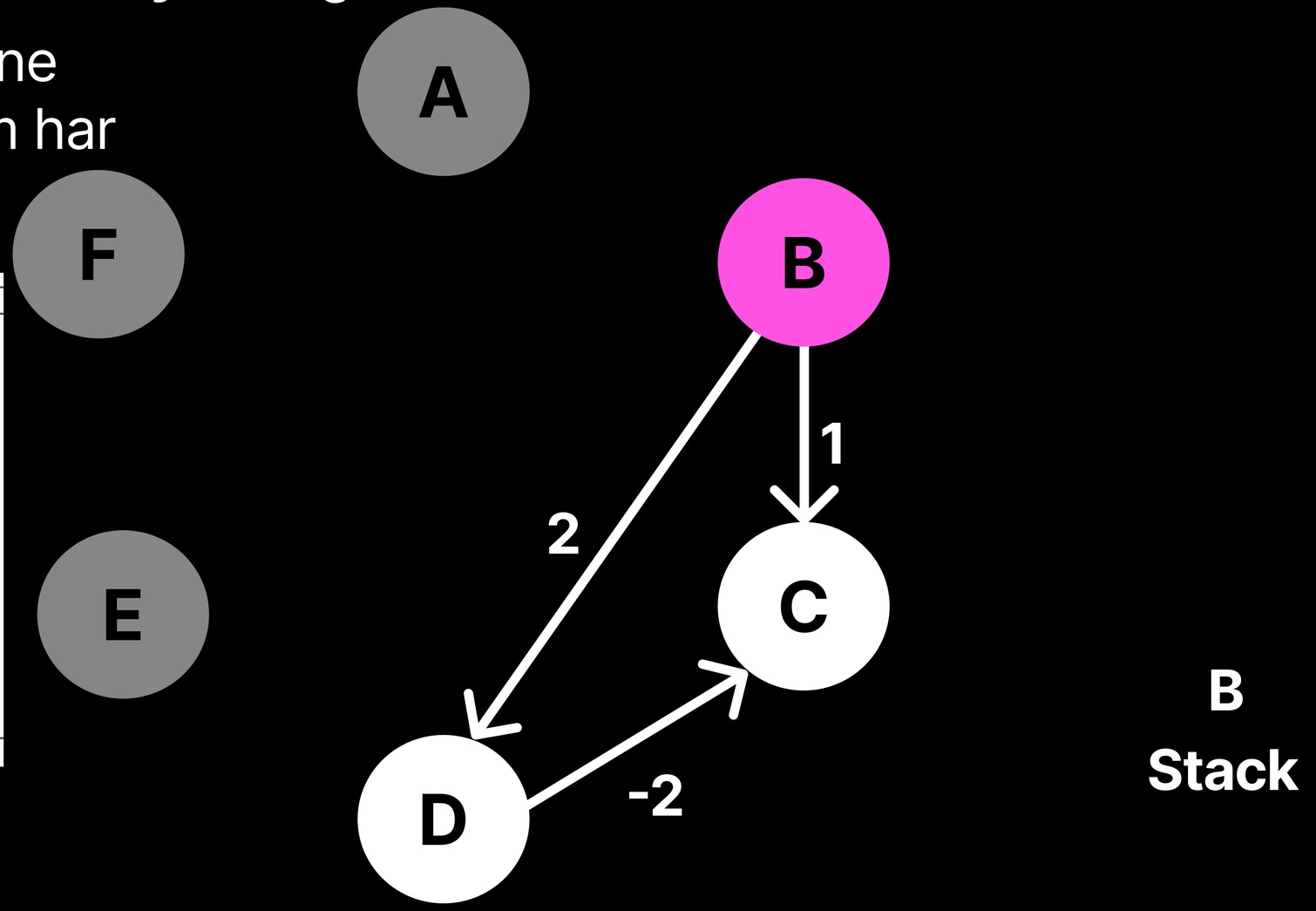
Ordner nodene i en **rettet asyklisk** graf (DAG).

Fjerner kantene til denne sjekker igjen hvem som har inngrad 0 nå

ALGORITHM: TOPOLOGISK SORTERING

Input: En rettet graf $G = (V, E)$
Output: En topologisk ordning av nodene i G

```
1  Procedure TopSort(G)
2      stack ← empty stack
3      output ← empty list
4      for v ∈ V do
5          if indegree of v is 0 then
6              stack.push(v)
7      while stack is not empty do
8          u ← stack.pop()
9          append u to output
10         for (u, v) ∈ E do
11             remove u from the incoming edges of v
12             if indegree of v is 0 then
13                 add v to stack
14     if |output| < |V| then
15         error G contains a cycle and cannot be topologically ordered
16     return output
```



Output: A F E

B

Stack

Topologisk sortering (TopSort)

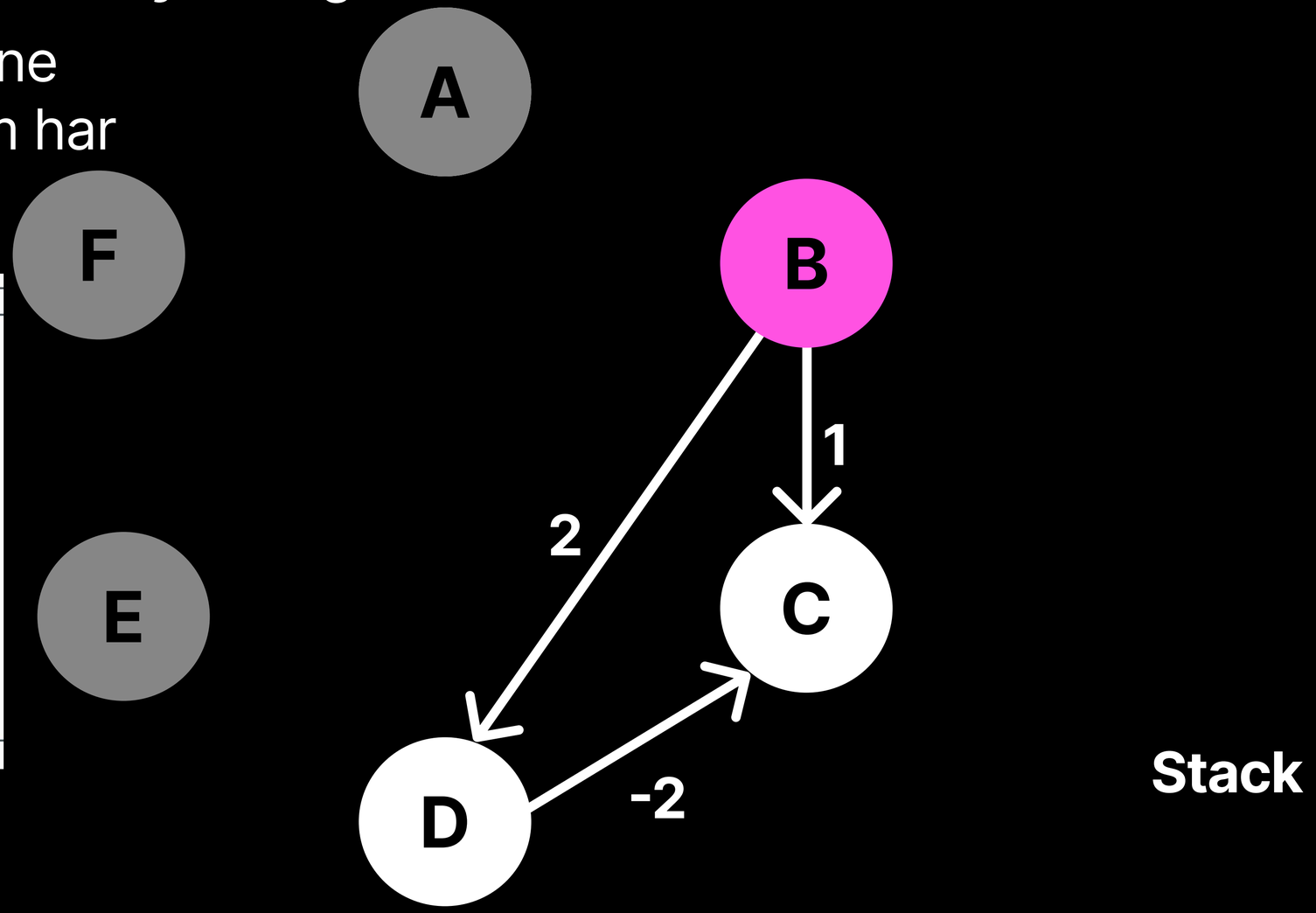
Ordner nodene i en **rettet asyklisk** graf (DAG).

Fjerner kantene til denne sjekker igjen hvem som har inngrad 0 nå

ALGORITHM: TOPOLOGISK SORTERING

Input: En rettet graf $G = (V, E)$
Output: En topologisk ordning av nodene i G

```
1  Procedure TopSort(G)
2      stack ← empty stack
3      output ← empty list
4      for v ∈ V do
5          if indegree of v is 0 then
6              stack.push(v)
7      while stack is not empty do
8          u ← stack.pop()
9          append u to output
10         for (u, v) ∈ E do
11             remove u from the incoming edges of v
12             if indegree of v is 0 then
13                 add v to stack
14     if |output| < |V| then
15         error G contains a cycle and cannot be topologically ordered
16     return output
```



Output: A F E B

Stack

Topologisk sortering (TopSort)

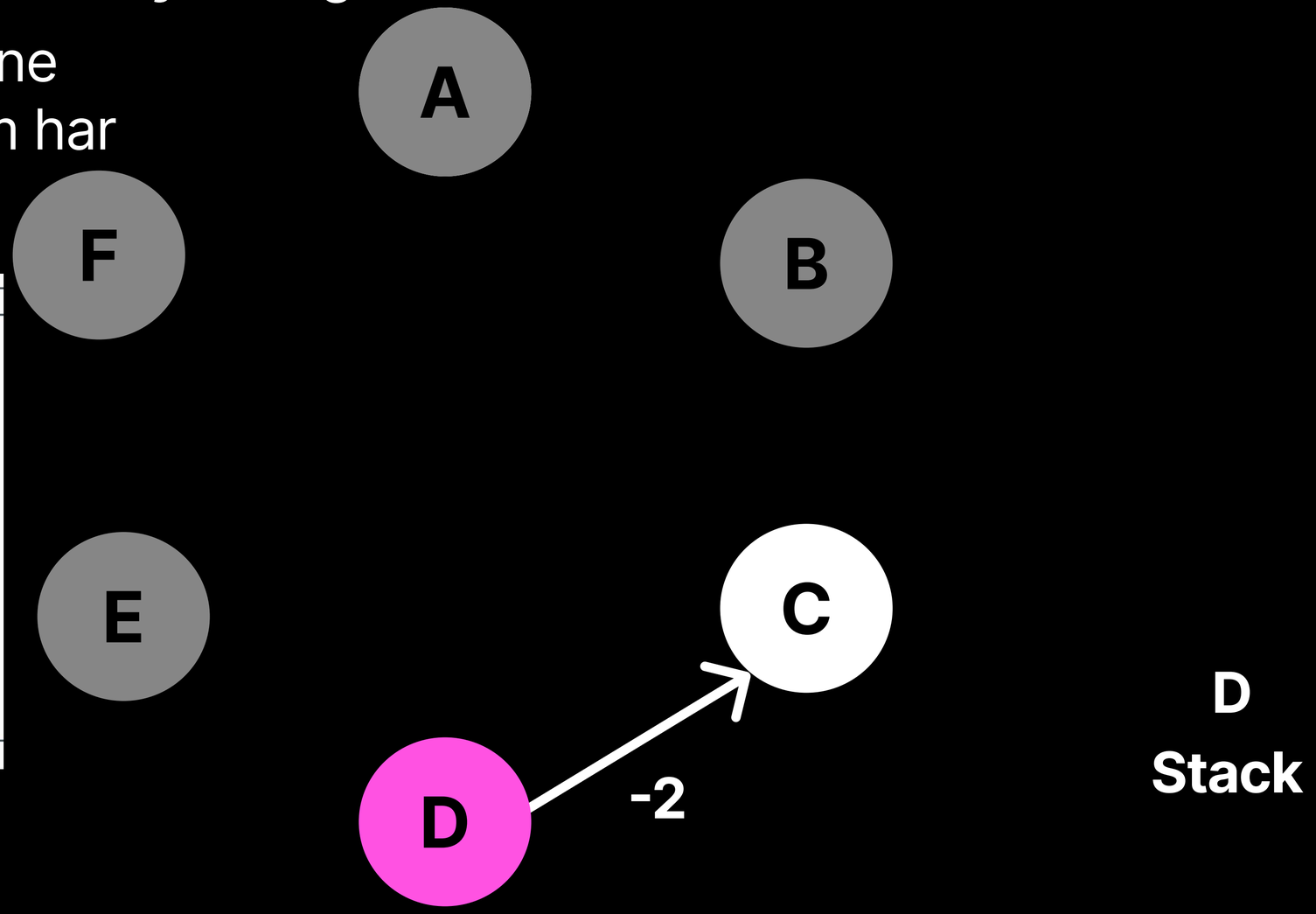
Ordner nodene i en **rettet asyklisk** graf (DAG).

Fjerner kantene til denne sjekker igjen hvem som har inngrad 0 nå

ALGORITHM: TOPOLOGISK SORTERING

Input: En rettet graf $G = (V, E)$
Output: En topologisk ordning av nodene i G

```
1  Procedure TopSort(G)
2      stack ← empty stack
3      output ← empty list
4      for v ∈ V do
5          if indegree of v is 0 then
6              stack.push(v)
7      while stack is not empty do
8          u ← stack.pop()
9          append u to output
10         for (u, v) ∈ E do
11             remove u from the incoming edges of v
12             if indegree of v is 0 then
13                 add v to stack
14     if |output| < |V| then
15         error G contains a cycle and cannot be topologically ordered
16     return output
```



Output: A F E B

Topologisk sortering (TopSort)

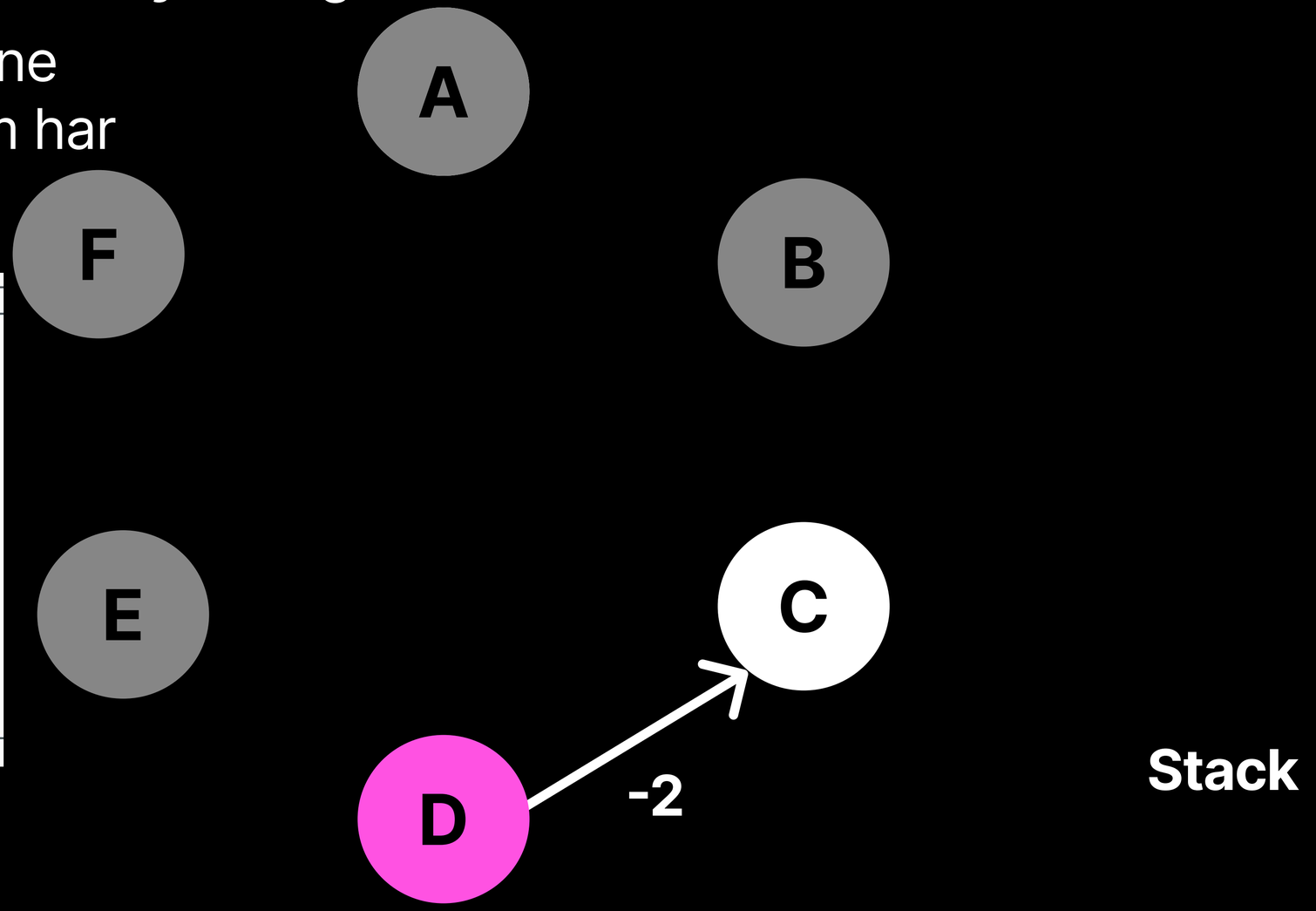
Ordner nodene i en **rettet asyklisk** graf (DAG).

Fjerner kantene til denne sjekker igjen hvem som har inngrad 0 nå

ALGORITHM: TOPOLOGISK SORTERING

Input: En rettet graf $G = (V, E)$
Output: En topologisk ordning av nodene i G

```
1  Procedure TopSort(G)
2      stack ← empty stack
3      output ← empty list
4      for v ∈ V do
5          if indegree of v is 0 then
6              stack.push(v)
7      while stack is not empty do
8          u ← stack.pop()
9          append u to output
10         for (u, v) ∈ E do
11             remove u from the incoming edges of v
12             if indegree of v is 0 then
13                 add v to stack
14     if |output| < |V| then
15         error G contains a cycle and cannot be topologically ordered
16     return output
```



Output: A F E B D

Topologisk sortering (TopSort)

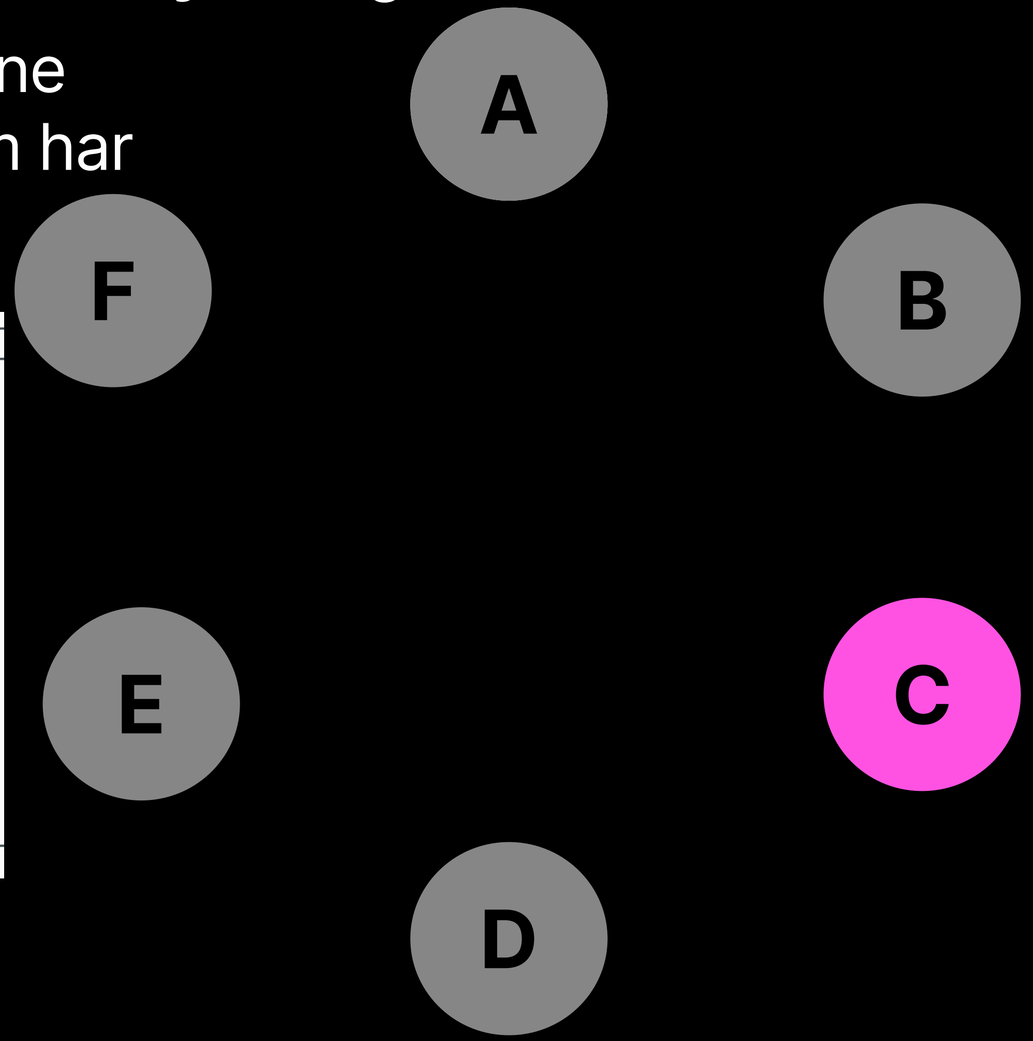
Ordner nodene i en **rettet asyklisk** graf (DAG).

Fjerner kantene til denne sjekker igjen hvem som har inngrad 0 nå

ALGORITHM: TOPOLOGISK SORTERING

Input: En rettet graf $G = (V, E)$
Output: En topologisk ordning av nodene i G

```
1  Procedure TopSort(G)
2      stack ← empty stack
3      output ← empty list
4      for v ∈ V do
5          if indegree of v is 0 then
6              stack.push(v)
7      while stack is not empty do
8          u ← stack.pop()
9          append u to output
10         for (u, v) ∈ E do
11             remove u from the incoming edges of v
12             if indegree of v is 0 then
13                 add v to stack
14     if |output| < |V| then
15         error G contains a cycle and cannot be topologically ordered
16     return output
```



C
Stack

Output: A F E B D

Topologisk sortering (TopSort)

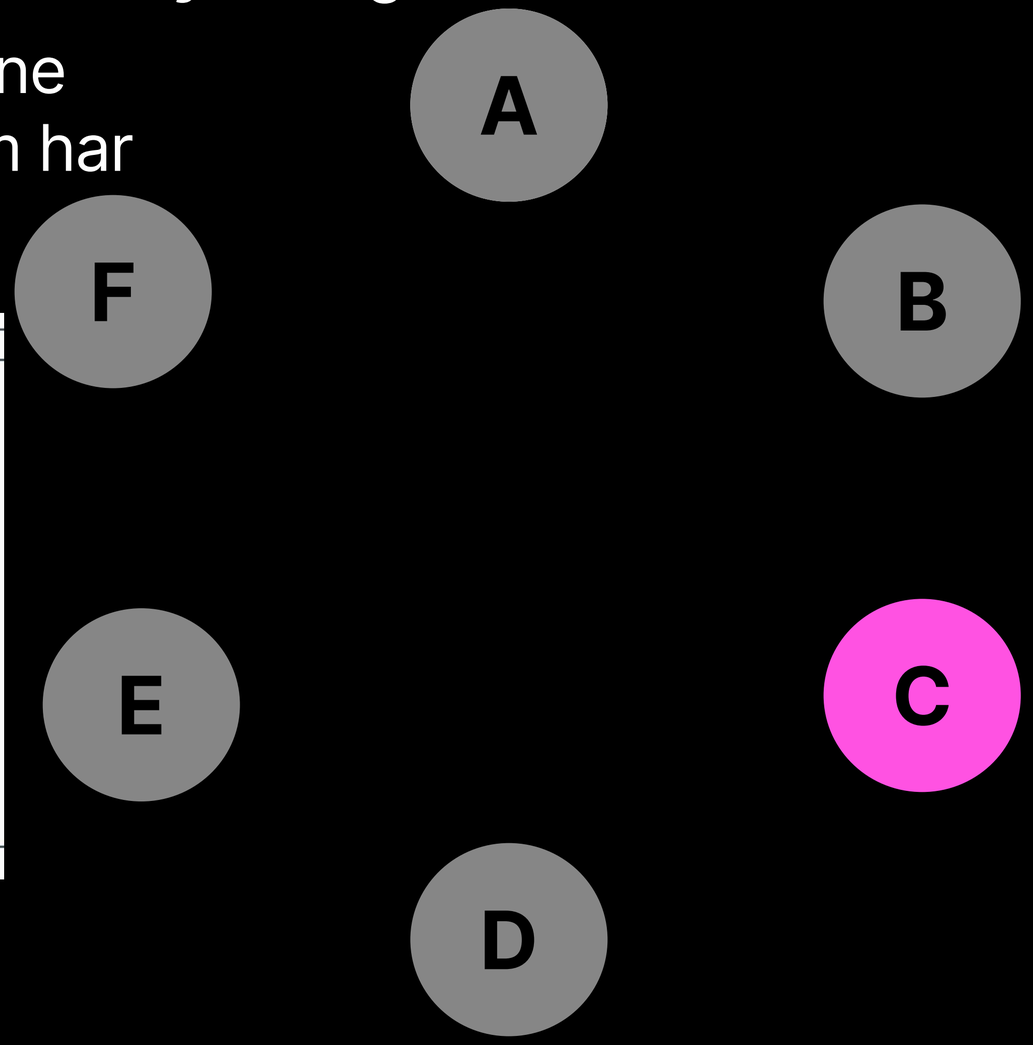
Ordner nodene i en **rettet asyklisk** graf (DAG).

Fjerner kantene til denne sjekker igjen hvem som har inngrad 0 nå

ALGORITHM: TOPOLOGISK SORTERING

Input: En rettet graf $G = (V, E)$
Output: En topologisk ordning av nodene i G

```
1  Procedure TopSort(G)
2      stack ← empty stack
3      output ← empty list
4      for v ∈ V do
5          if indegree of v is 0 then
6              stack.push(v)
7      while stack is not empty do
8          u ← stack.pop()
9          append u to output
10         for (u, v) ∈ E do
11             remove u from the incoming edges of v
12             if indegree of v is 0 then
13                 add v to stack
14     if |output| < |V| then
15         error G contains a cycle and cannot be topologically ordered
16     return output
```



Output: A F E B D C

Stack

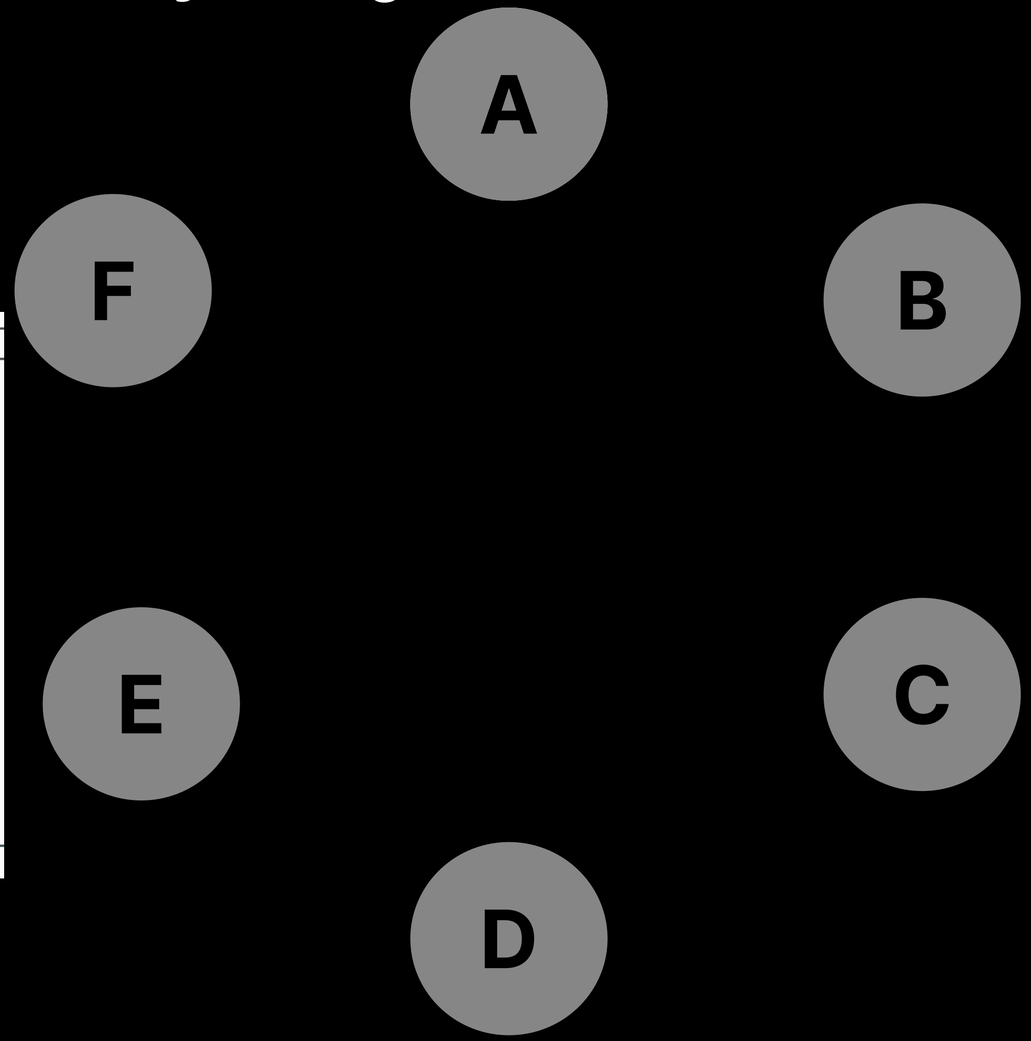
Topologisk sortering (TopSort)

Ordner nodene i en **rettet asyklisk** graf (DAG).

ALGORITHM: TOPOLOGISK SORTERING

Input: En rettet graf $G = (V, E)$
Output: En topologisk ordning av nodene i G

```
1  Procedure TopSort(G)
2      stack ← empty stack
3      output ← empty list
4      for v ∈ V do
5          if indegree of v is 0 then
6              stack.push(v)
7      while stack is not empty do
8          u ← stack.pop()
9          append u to output
10         for (u, v) ∈ E do
11             remove u from the incoming edges of v
12             if indegree of v is 0 then
13                 add v to stack
14     if |output| < |V| then
15         error G contains a cycle and cannot be topologically ordered
16     return output
```

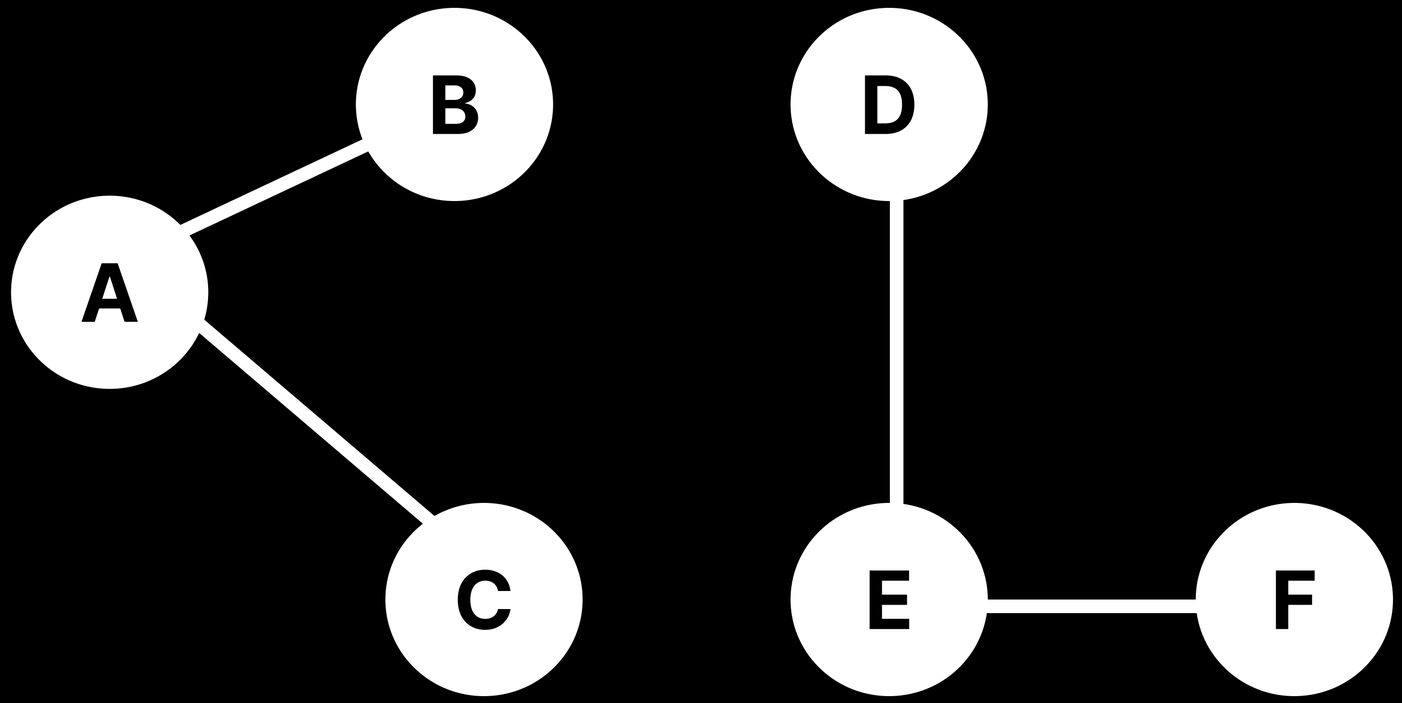


Output: A F E B D C

Stack

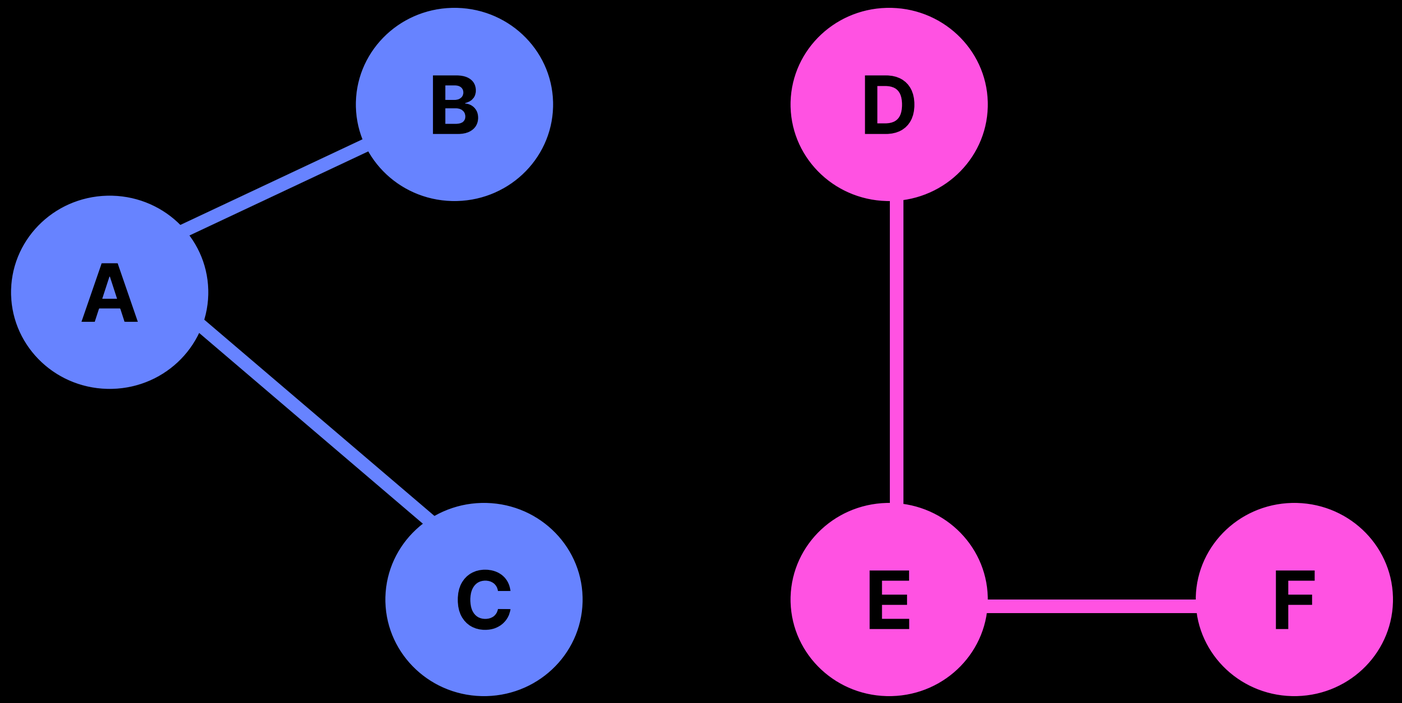
Ikke-sammenhengende grafer

har komponenter



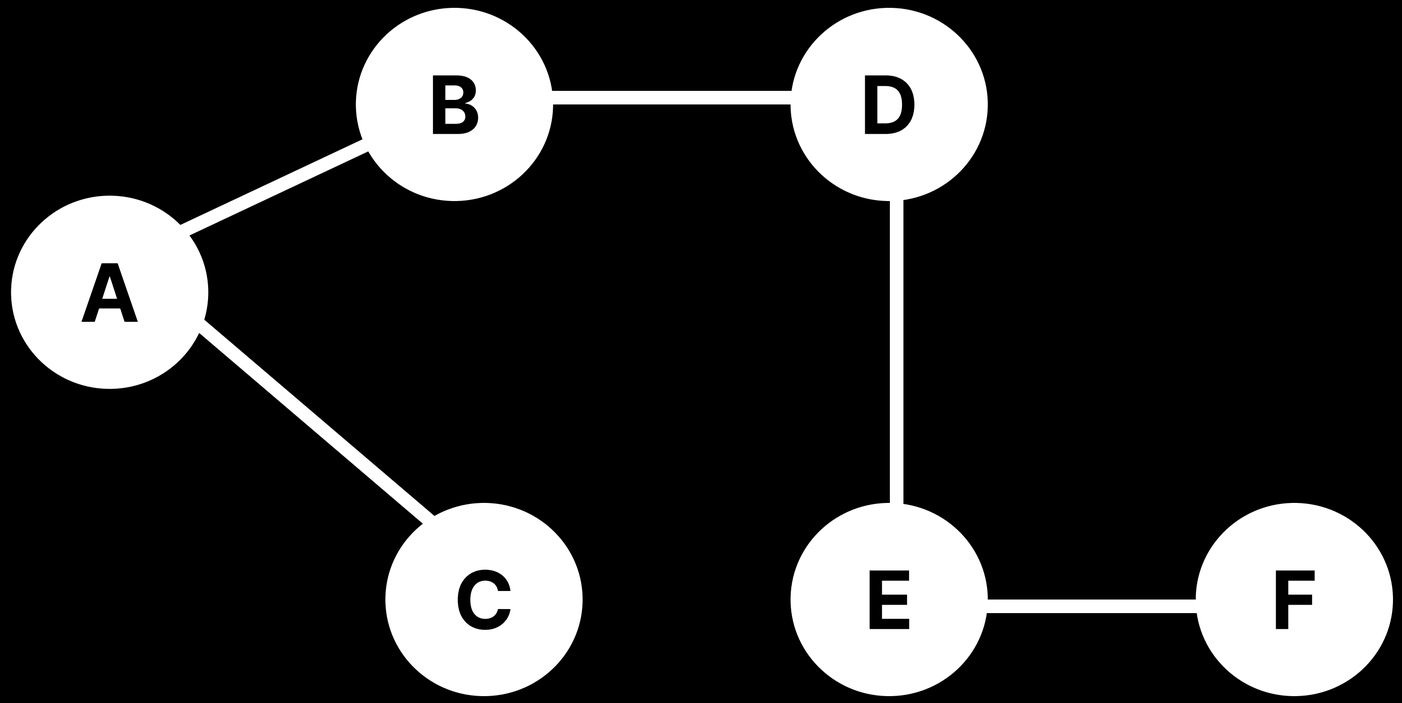
Ikke-sammenhengende grafer

har komponenter



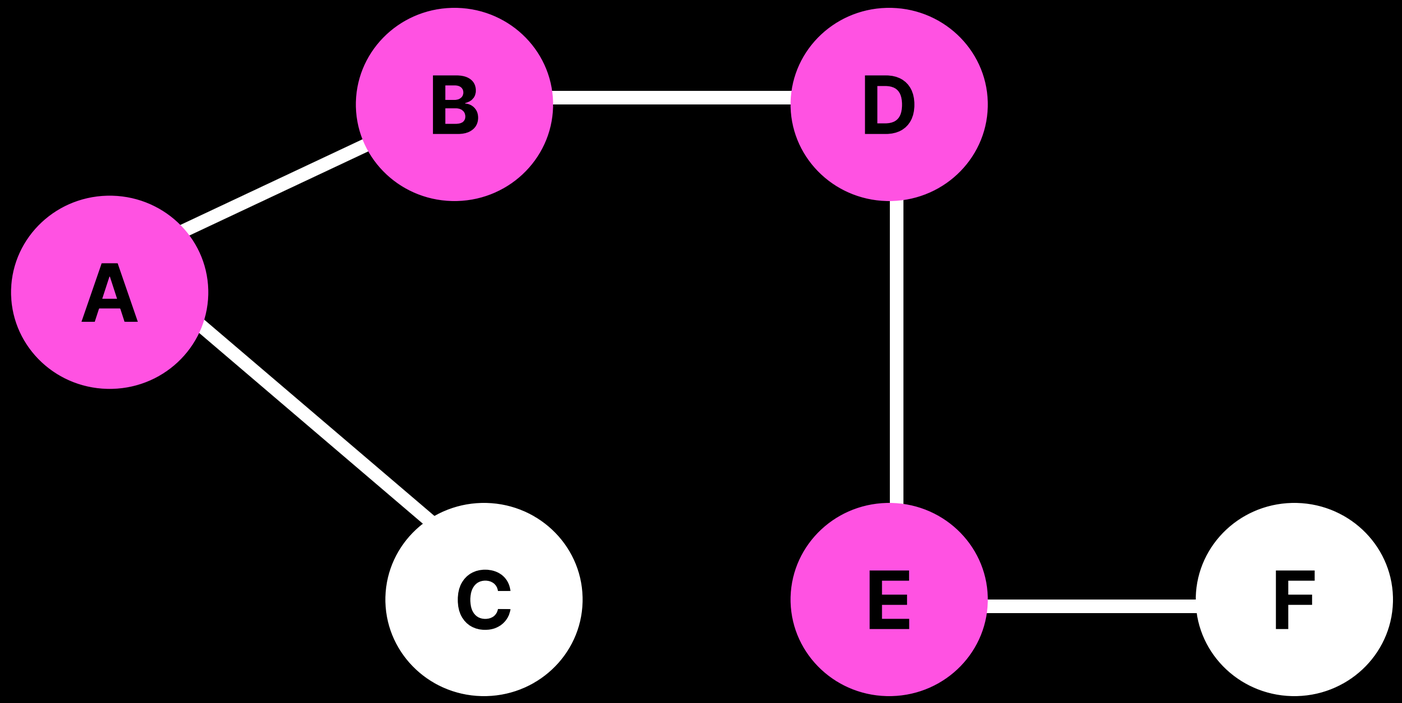
Sammenhengende grafer

det finnes en sti mellom alle par av noder



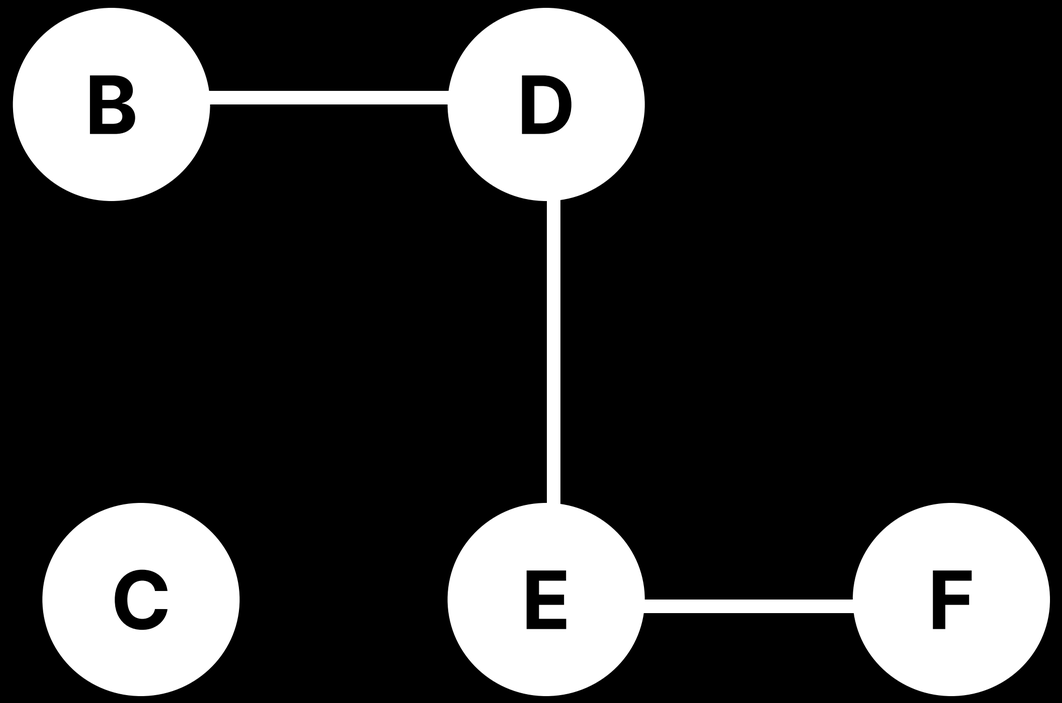
Separasjonsnoder

En node er en seperasjonsnode om grafen blir ikke-sammenhende om noden fjernes



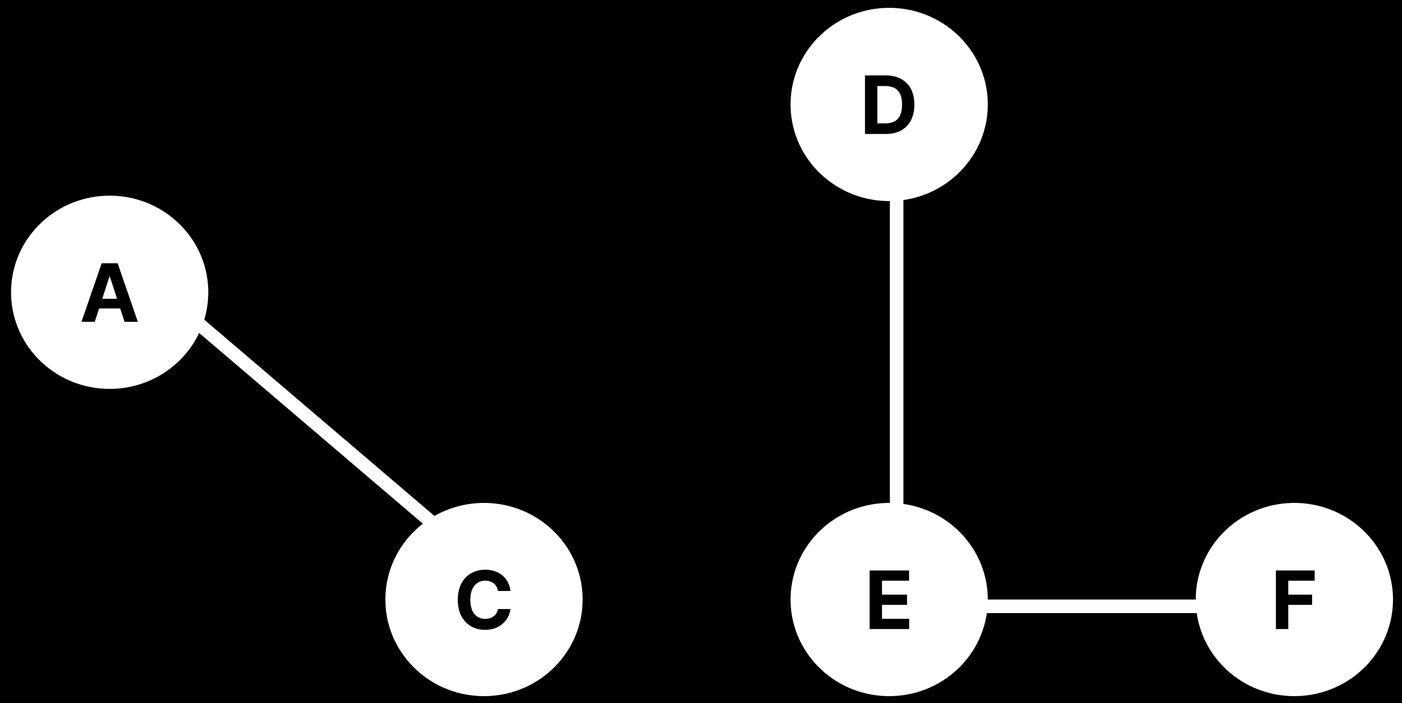
Separasjonsnoder

En node er en seperasjonsnode om grafen blir ikke-sammenhende om noden fjernes



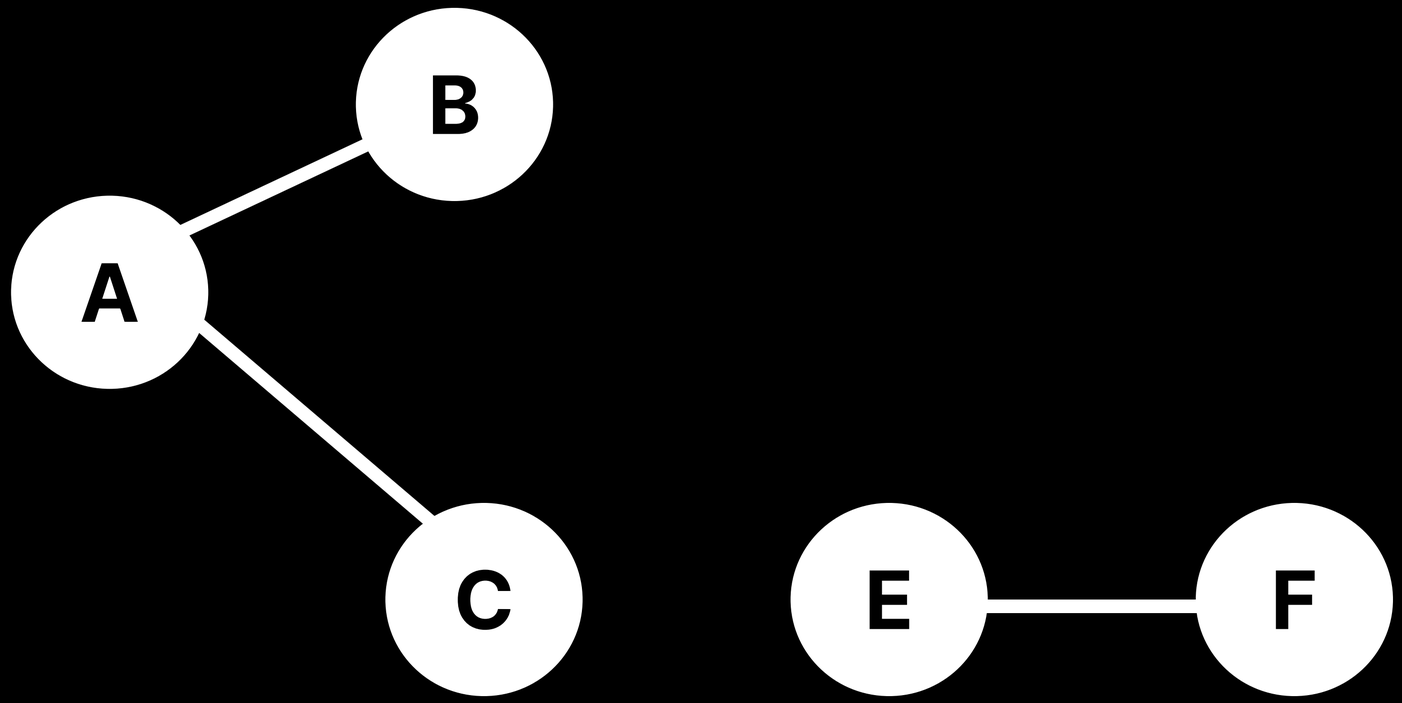
Separasjonsnoder

En node er en seperasjonsnode om grafen blir ikke-sammenhende om noden fjernes



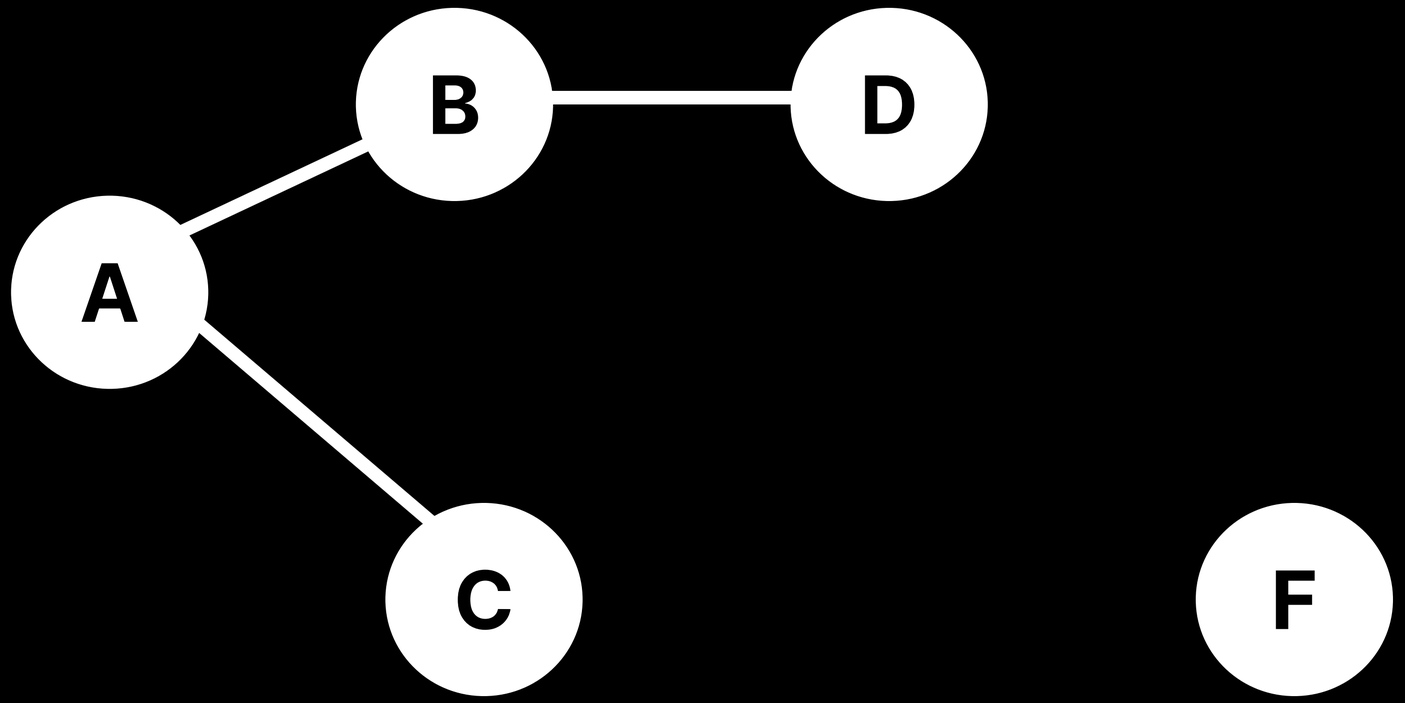
Separasjonsnoder

En node er en seperasjonsnode om grafen blir ikke-sammenhende om noden fjernes



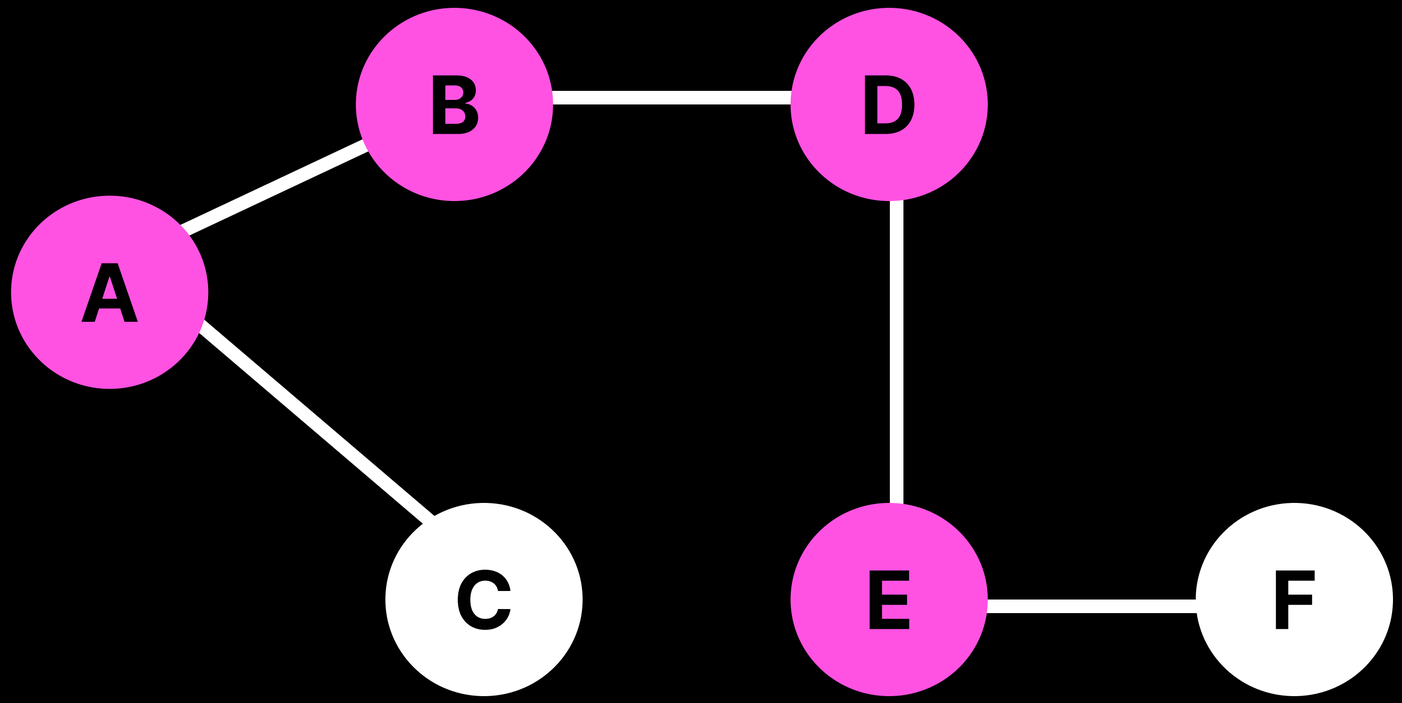
Separasjonsnoder

En node er en seperasjonsnode om grafen blir ikke-sammenhende om noden fjernes



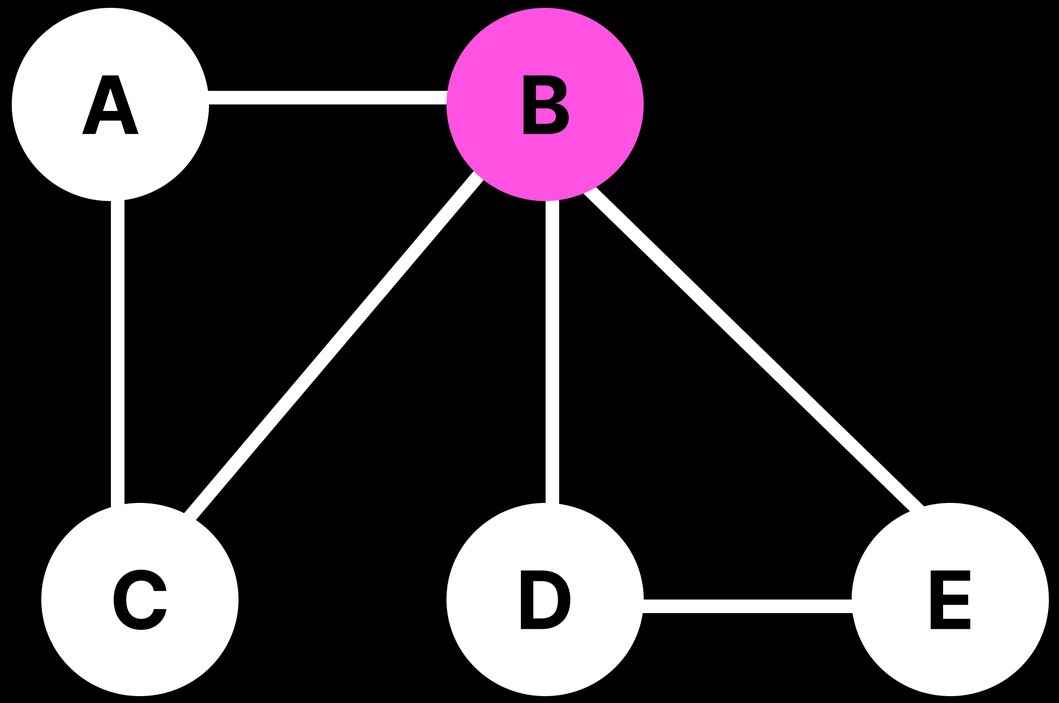
Separasjonsnoder

En node er en seperasjonsnode om grafen blir ikke-sammenhende om noden fjernes



Finne seperasjonsnoder med DFS

Dersom alle stier går gjennom en node så er denne noden en seperasjonsnode

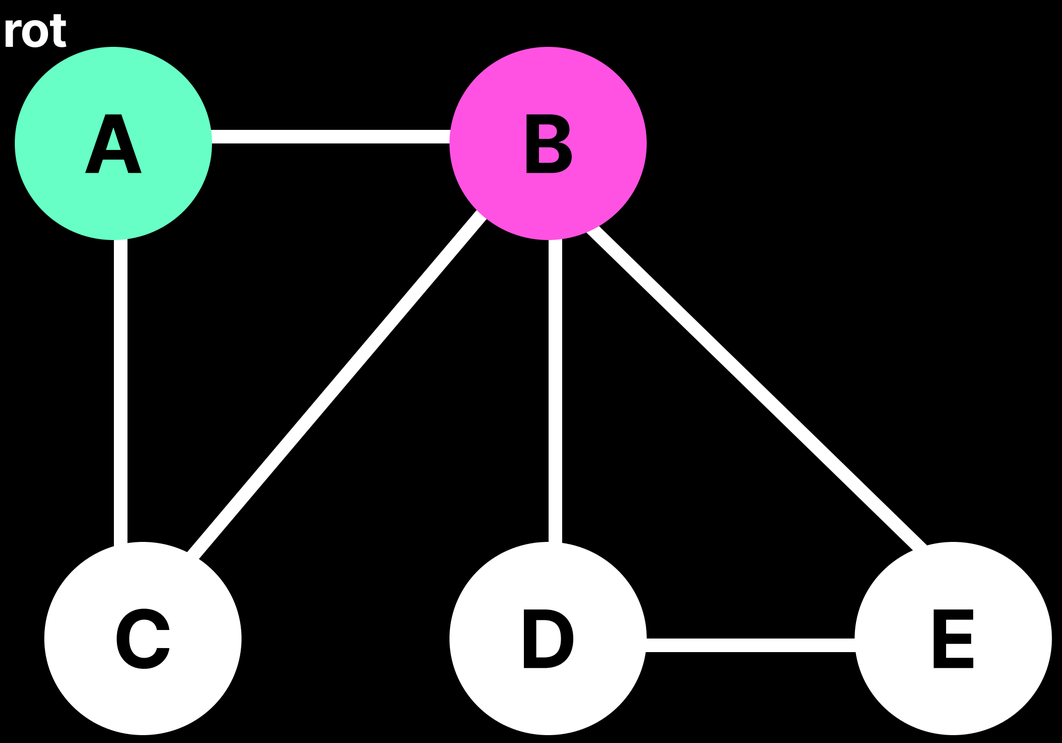


Finne seperasjonsnoder med DFS

Dersom alle stier går gjennom en node så er denne noden en seperasjonsnode

Gjør DFS fra A

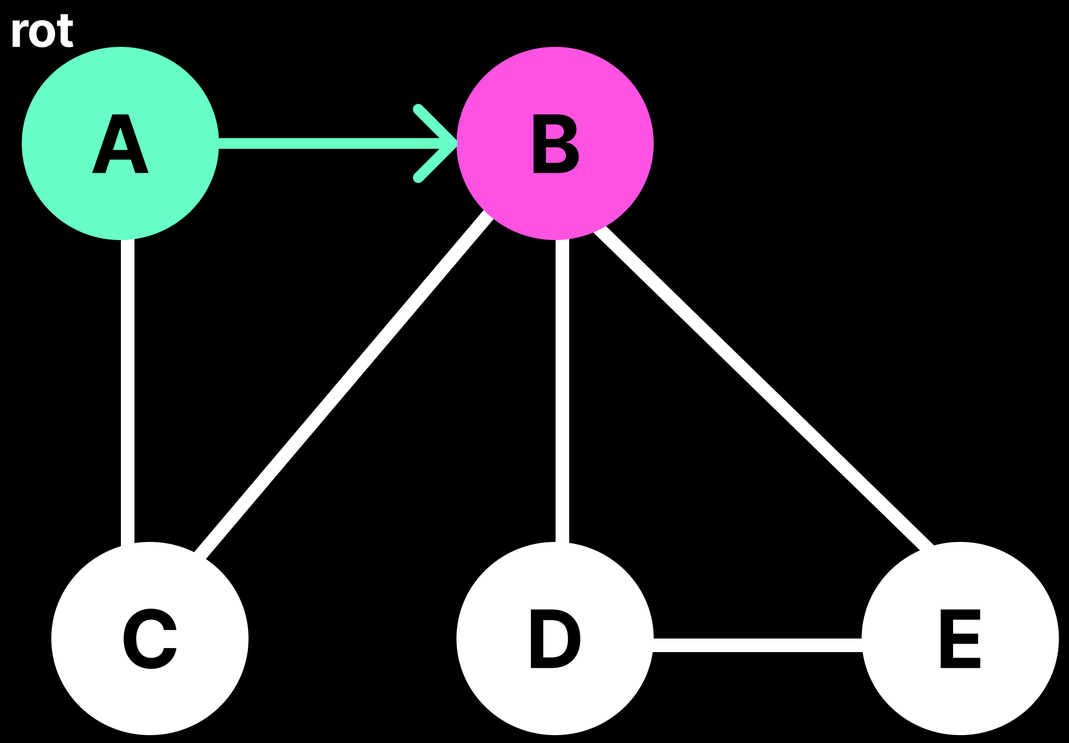
A blir rot i spenntreet T



Finne seperasjonsnoder med DFS

Dersom alle stier går gjennom en node så er denne noden en seperasjonsnode

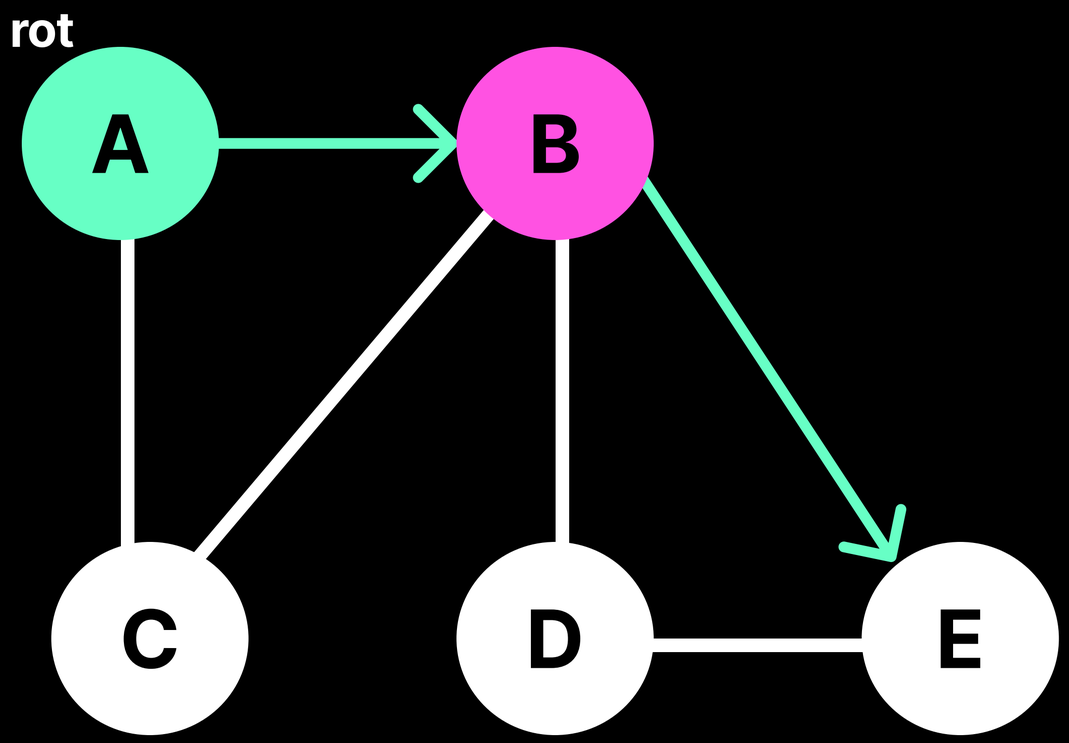
Gjør DFS fra A



Finne seperasjonsnoder med DFS

Dersom alle stier går gjennom en node så er denne noden en seperasjonsnode

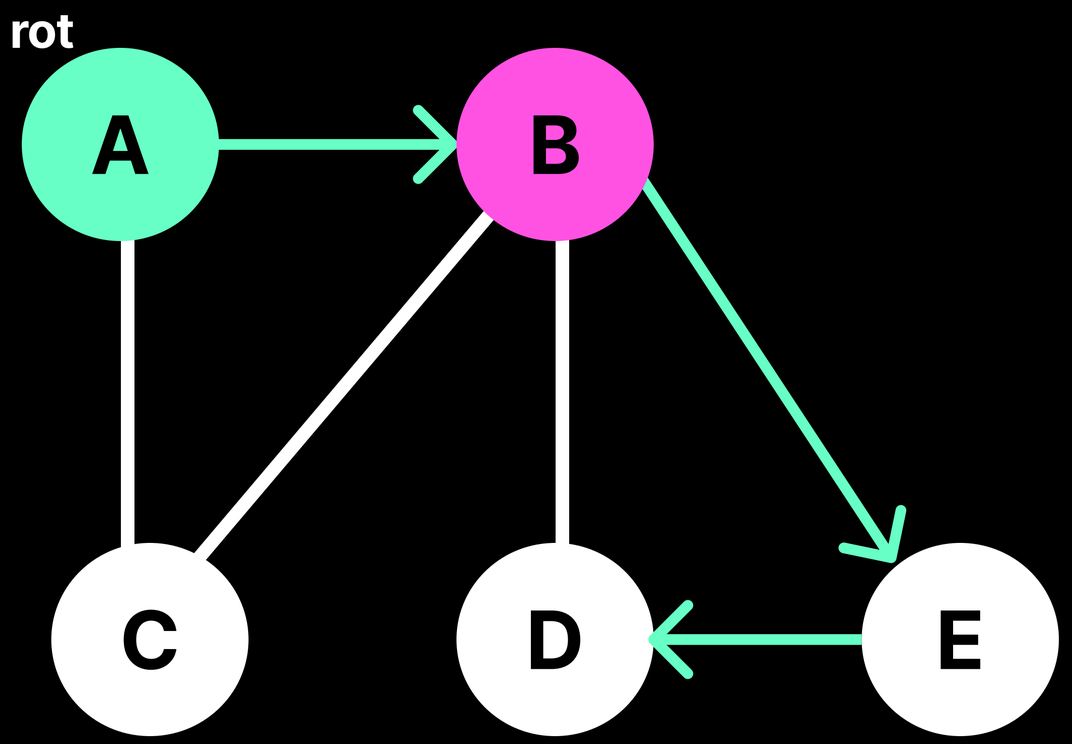
Gjør DFS fra A



Finne seperasjonsnoder med DFS

Dersom alle stier går gjennom en node så er denne noden en seperasjonsnode

Gjør DFS fra A

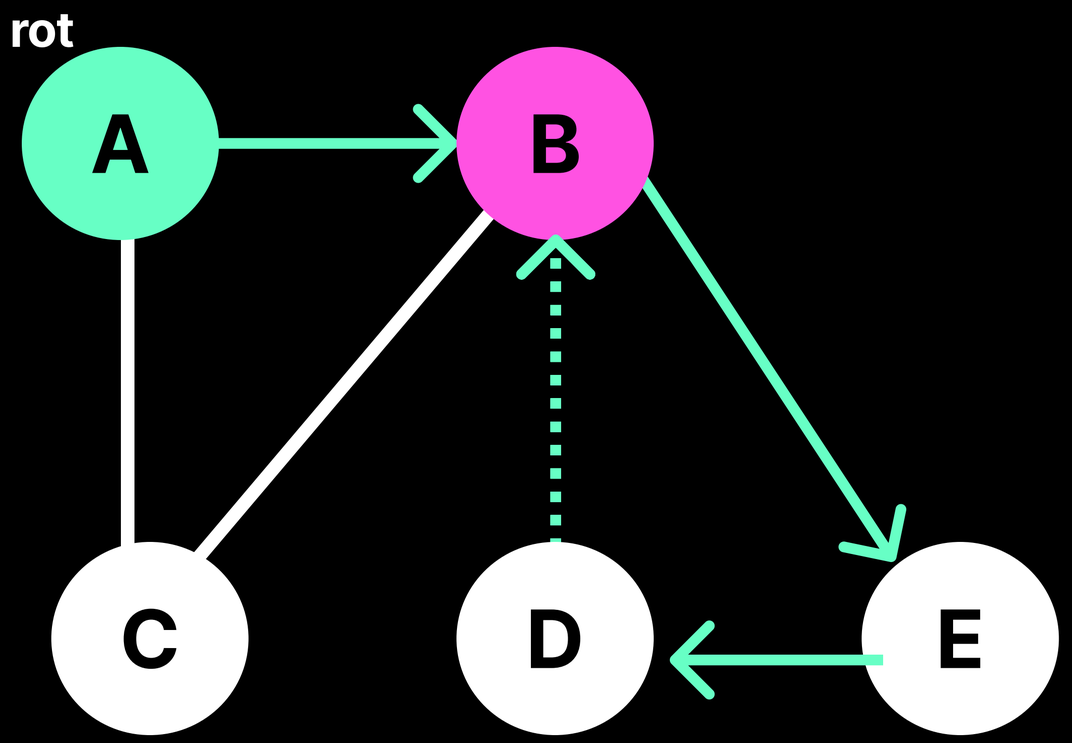


Finne seperasjonsnoder med DFS

Dersom alle stier går gjennom en node så er denne noden en seperasjonsnode

Gjør DFS fra A

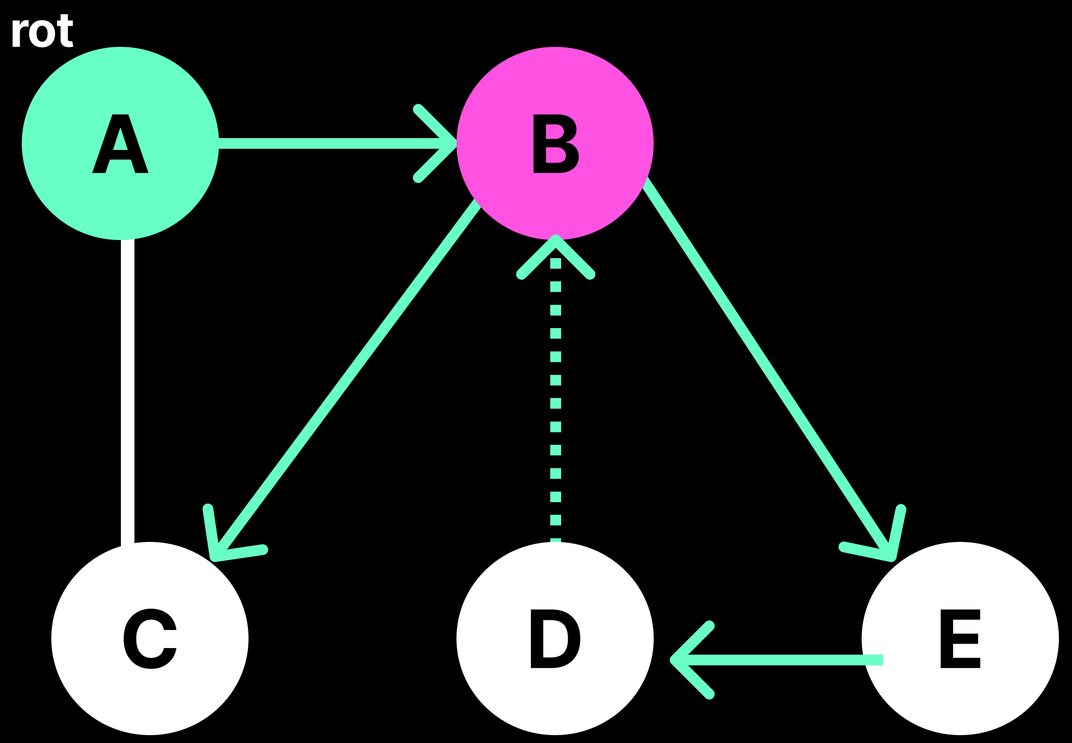
Vi har allerede besøkt B, så her får vi en back-edge



Finne seperasjonsnoder med DFS

Dersom alle stier går gjennom en node så er denne noden en seperasjonsnode

Gjør DFS fra A

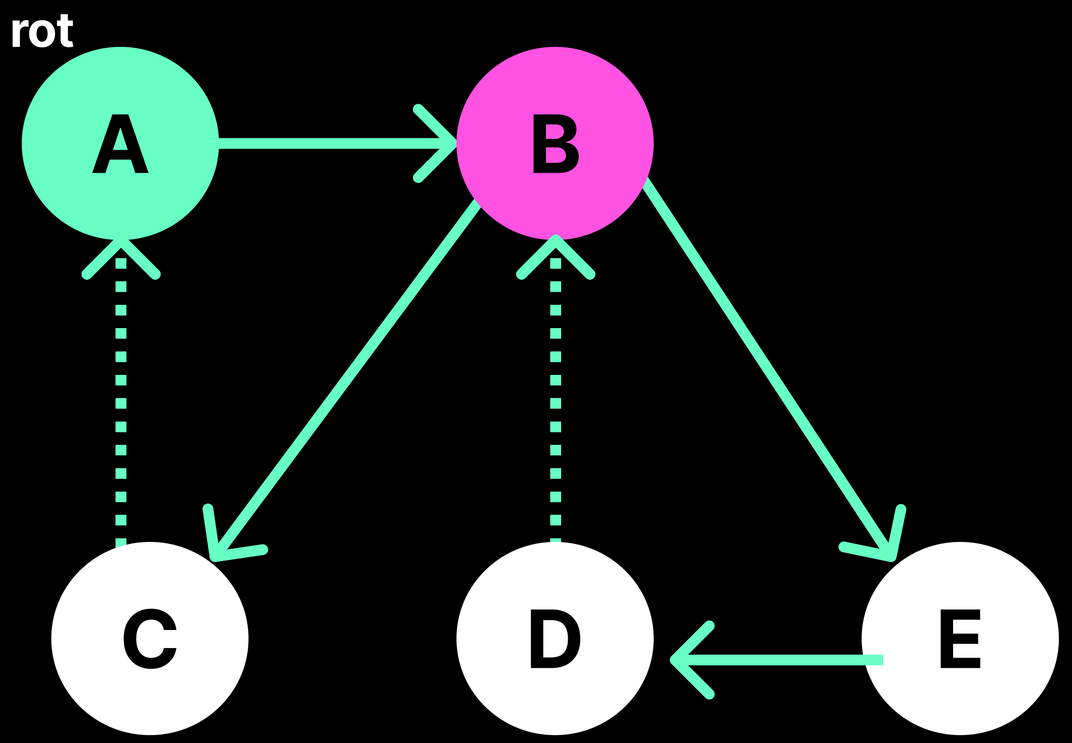


Finne seperasjonsnoder med DFS

Dersom alle stier går gjennom en node så er denne noden en seperasjonsnode

Gjør DFS fra A

Vi har allerede besøkt A, så her får vi også en back-edge

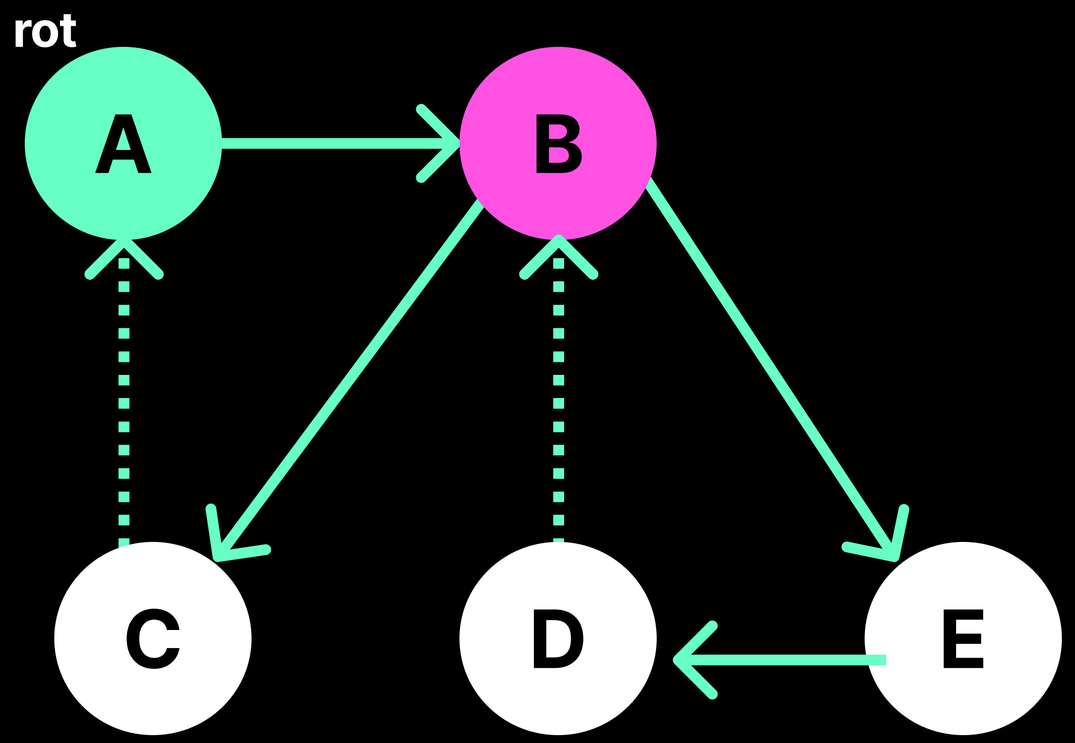


Finne seperasjonsnoder med DFS

Dersom alle stier går gjennom en node så er denne noden en seperasjonsnode

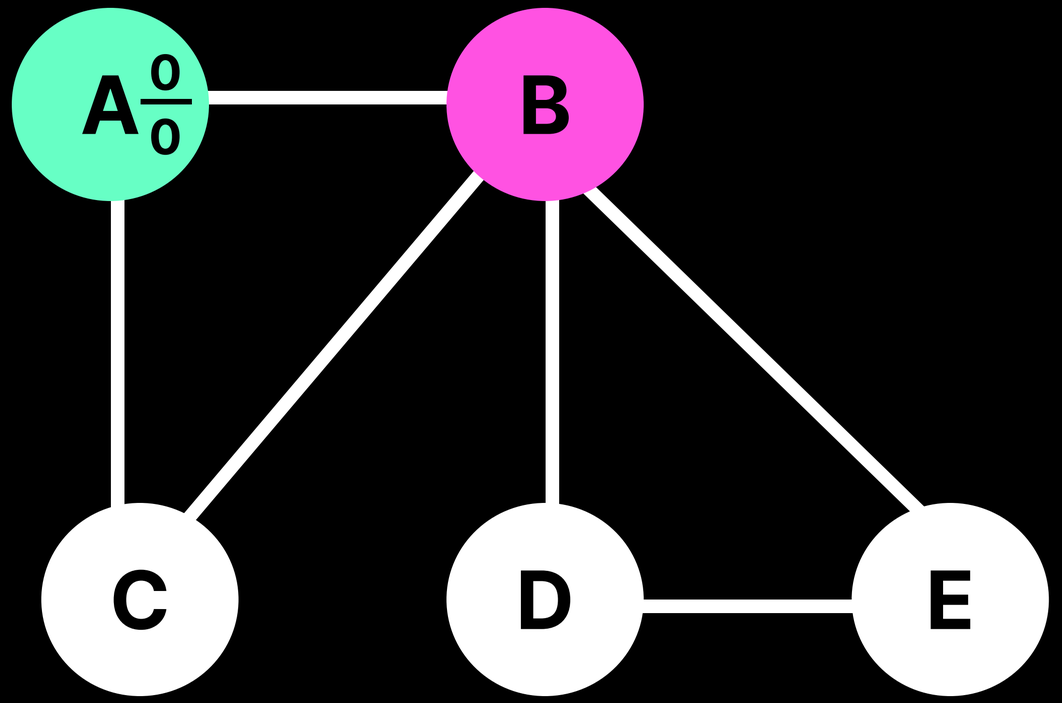
En node er en seperasjonsnode hvis:

- **noden er rot i T, og har mer enn ett barn**
- **noden er *ikke* rot, og har et barn der hverken barnet selv eller barnets etterkommere har en back-edge til seperasjonsnodens forgjengere**



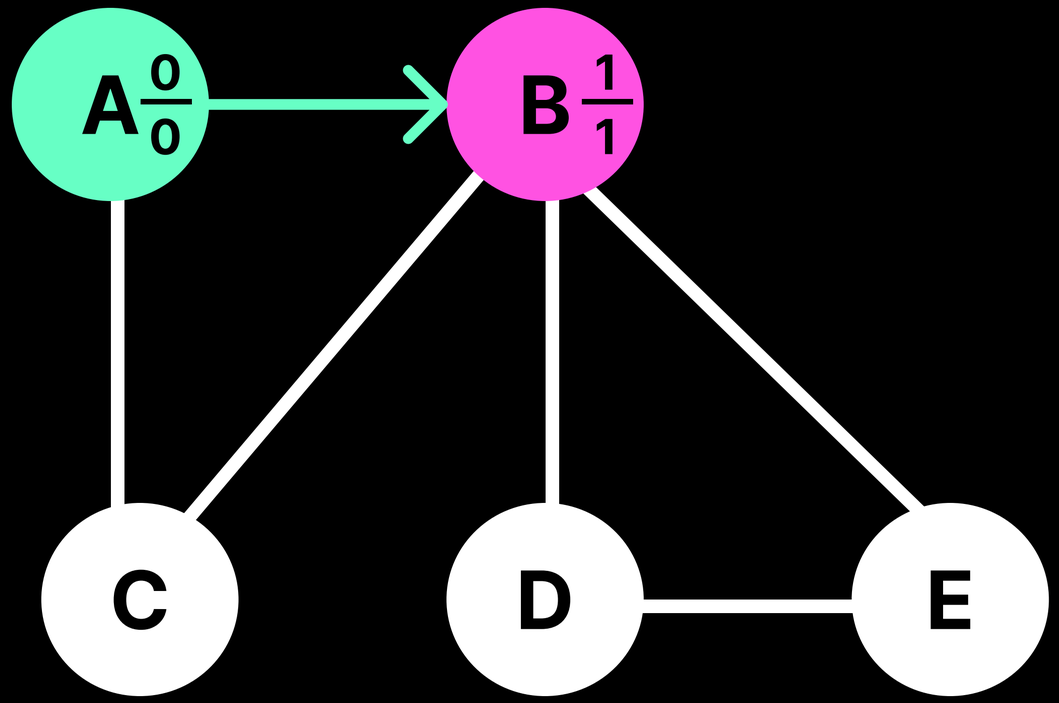
Finne seperasjonsnoder - DFS med depth og low

Gjør DFS fra A
For hver node noterer vi depth og low,
der low er den laveste dybden som kan nås ved å følge en eller flere etterkomere (max en back-edge)



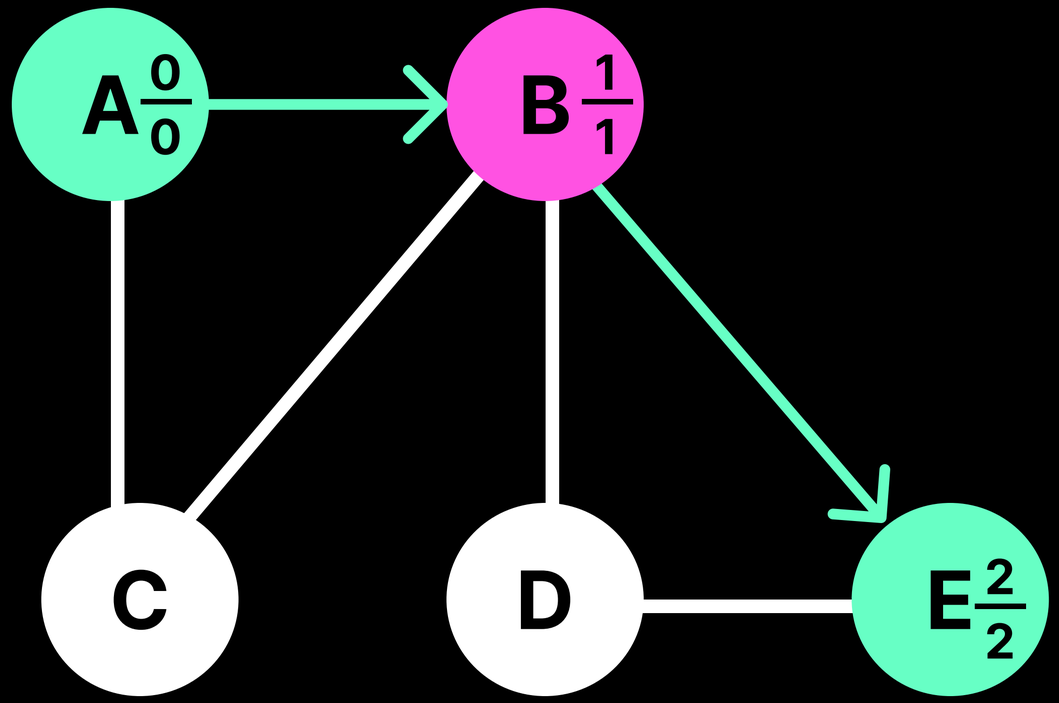
Finne seperasjonsnoder - DFS med depth og low

Gjør DFS fra A
For hver node noterer vi depth og low,
der low er den laveste dybden som kan nås ved å følge en eller flere etterkomere (max en back-edge)



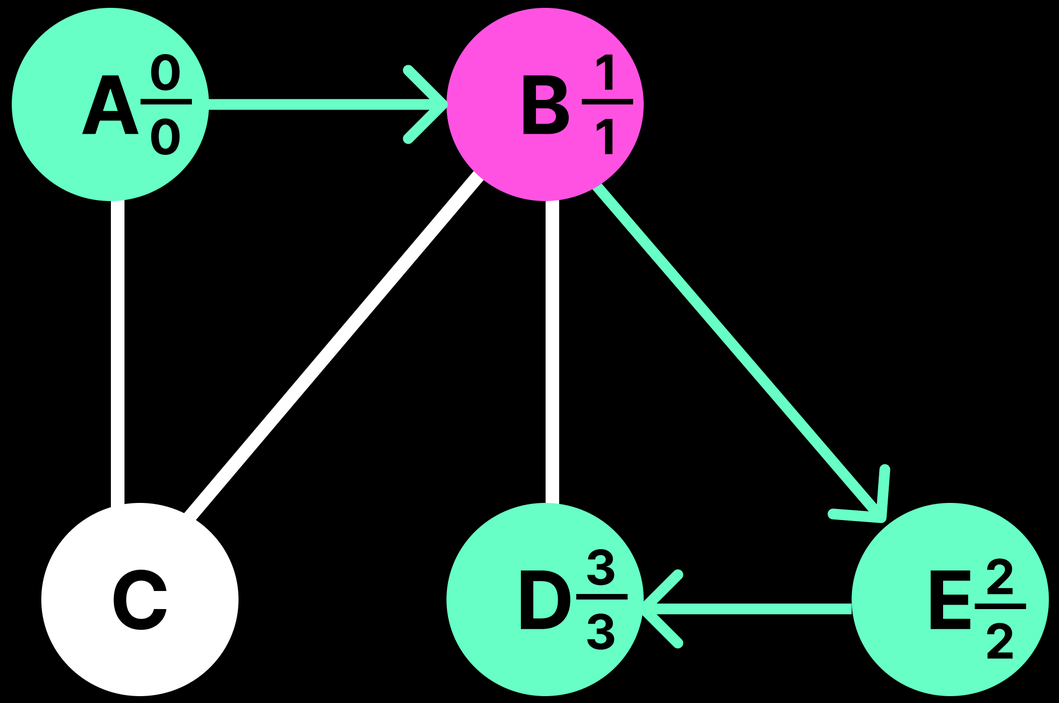
Finne seperasjonsnoder - DFS med depth og low

Gjør DFS fra A
For hver node noterer vi depth og low,
der low er den laveste dybden som kan nås ved å følge en eller flere etterkomere (max en back-edge)



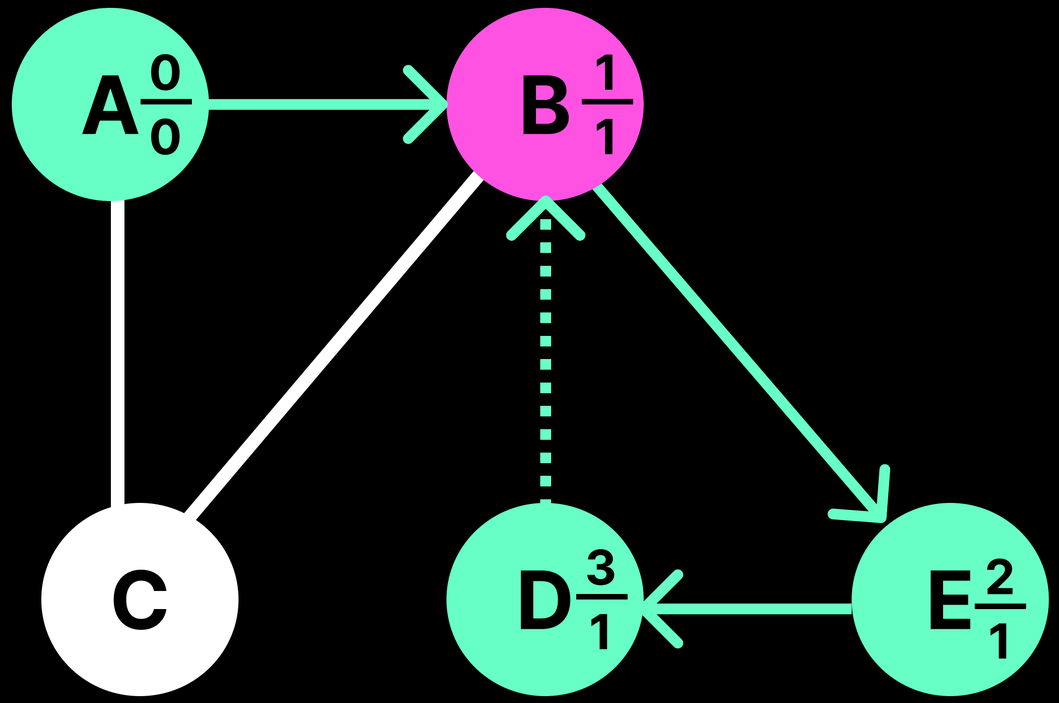
Finne seperasjonsnoder - DFS med depth og low

Gjør DFS fra A
For hver node noterer vi depth og low,
der low er den laveste dybden som kan nås ved å følge en eller flere etterkomere (max en back-edge)



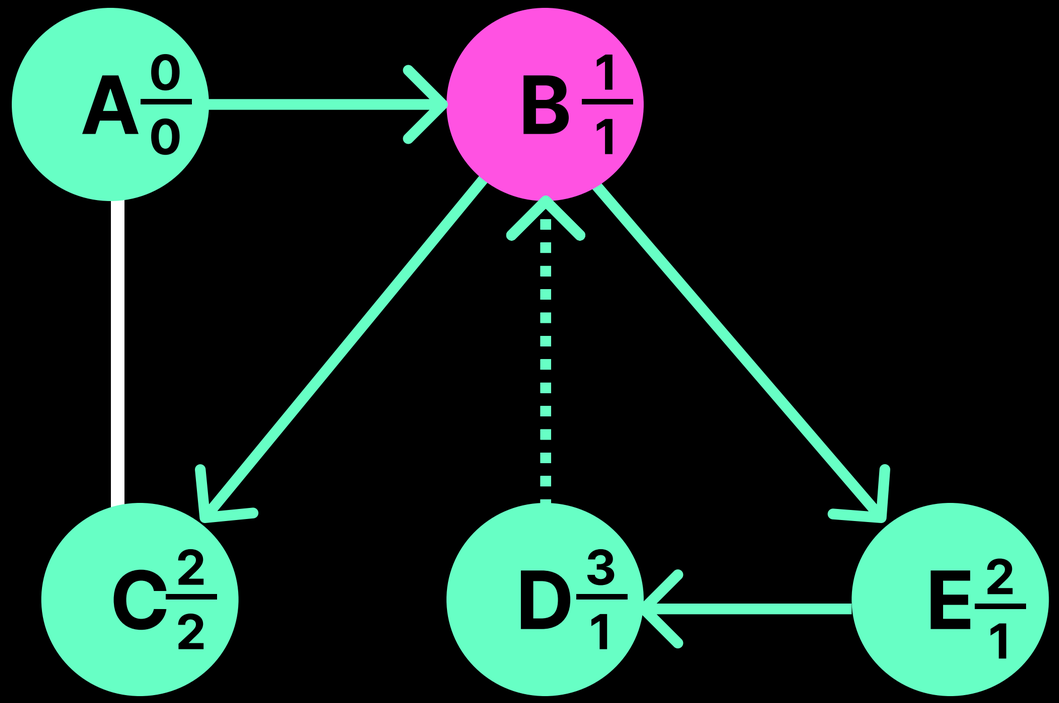
Finne seperasjonsnoder - DFS med depth og low

Gjør DFS fra A
For hver node noterer vi depth og low,
der low er den laveste dybden som kan nås ved å følge en eller flere etterkomere (max en back-edge)



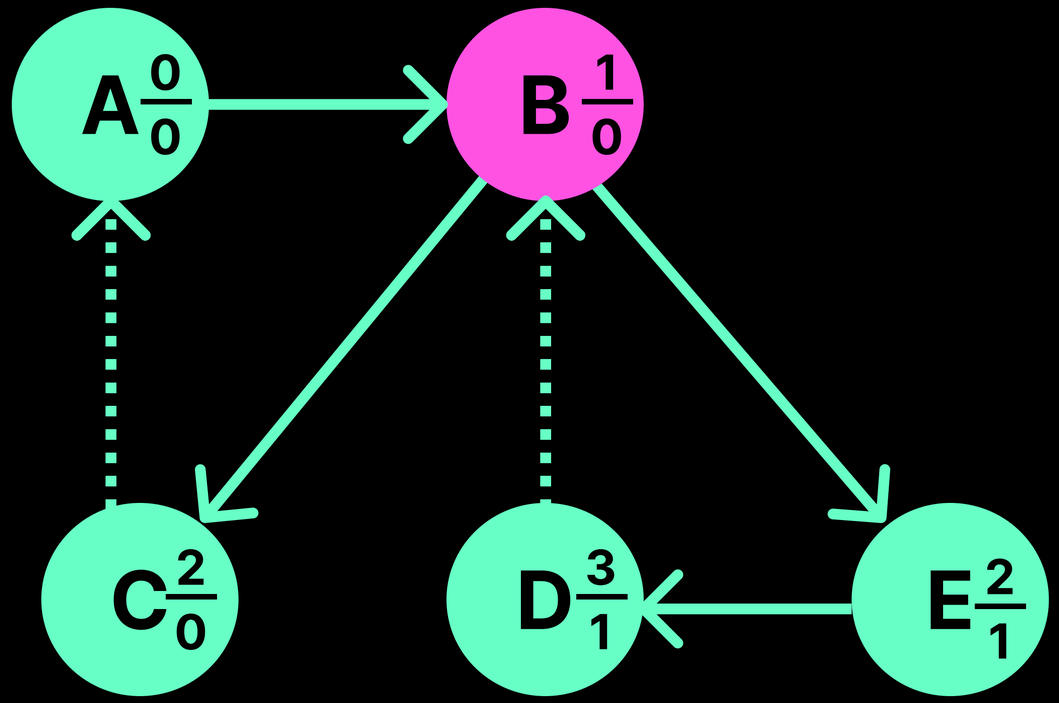
Finne seperasjonsnoder - DFS med depth og low

Gjør DFS fra A
For hver node noterer vi depth og low,
der low er den laveste dybden som kan nås ved å følge en eller flere etterkomere (max en back-edge)



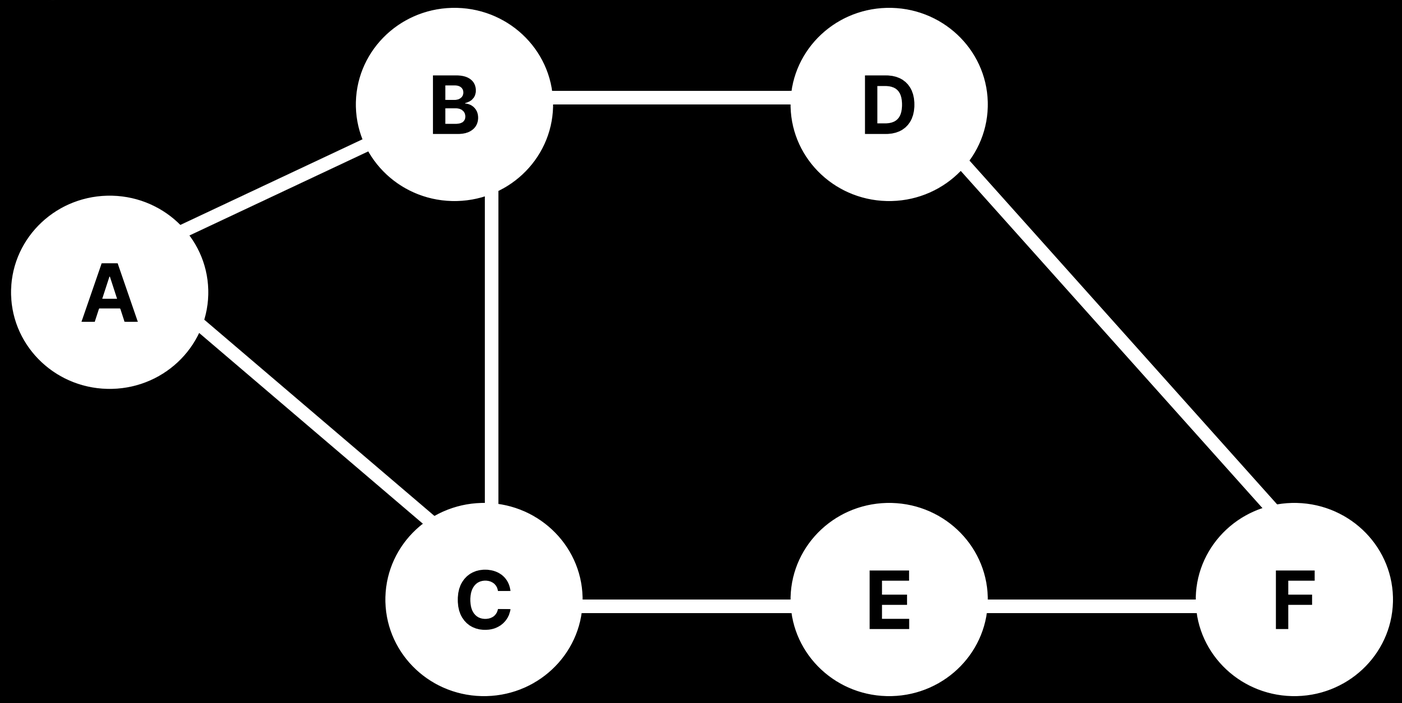
Finne seperasjonsnoder - DFS med depth og low

Gjør DFS fra A
For hver node noterer vi depth og low,
der low er den laveste dybden som kan nås ved å følge en eller flere etterkomere (max en back-edge)



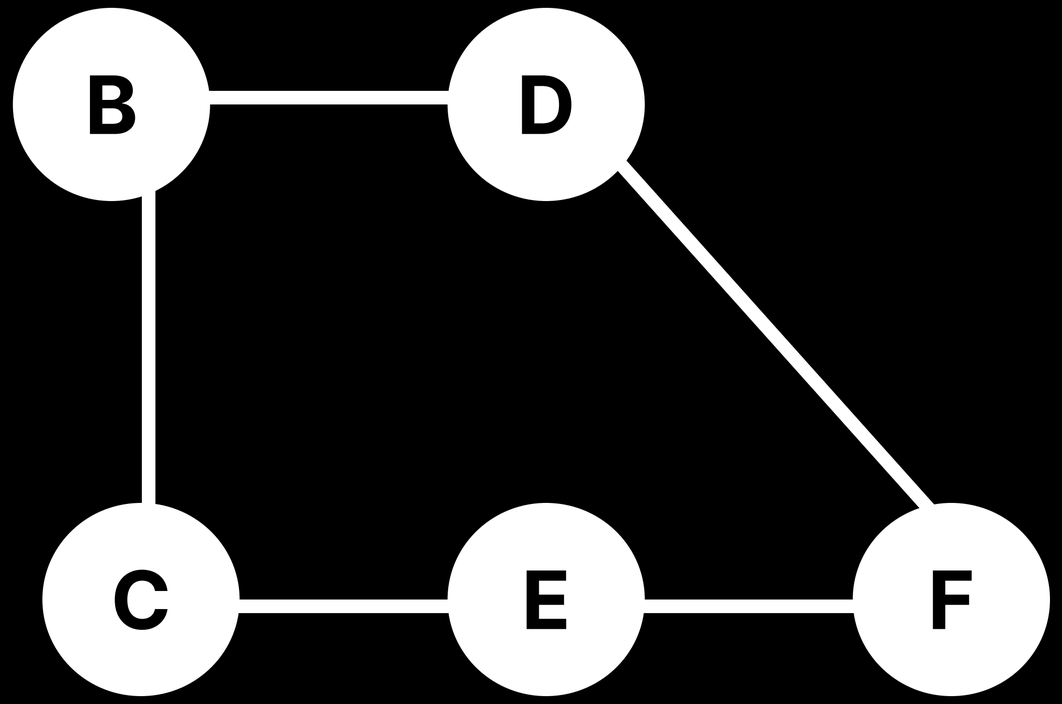
2-sammenhenende grafer

Må fjerne 2 noder for at grafen ikke skal være sammenhende lenger
Har ingen seperasjonsnoder!



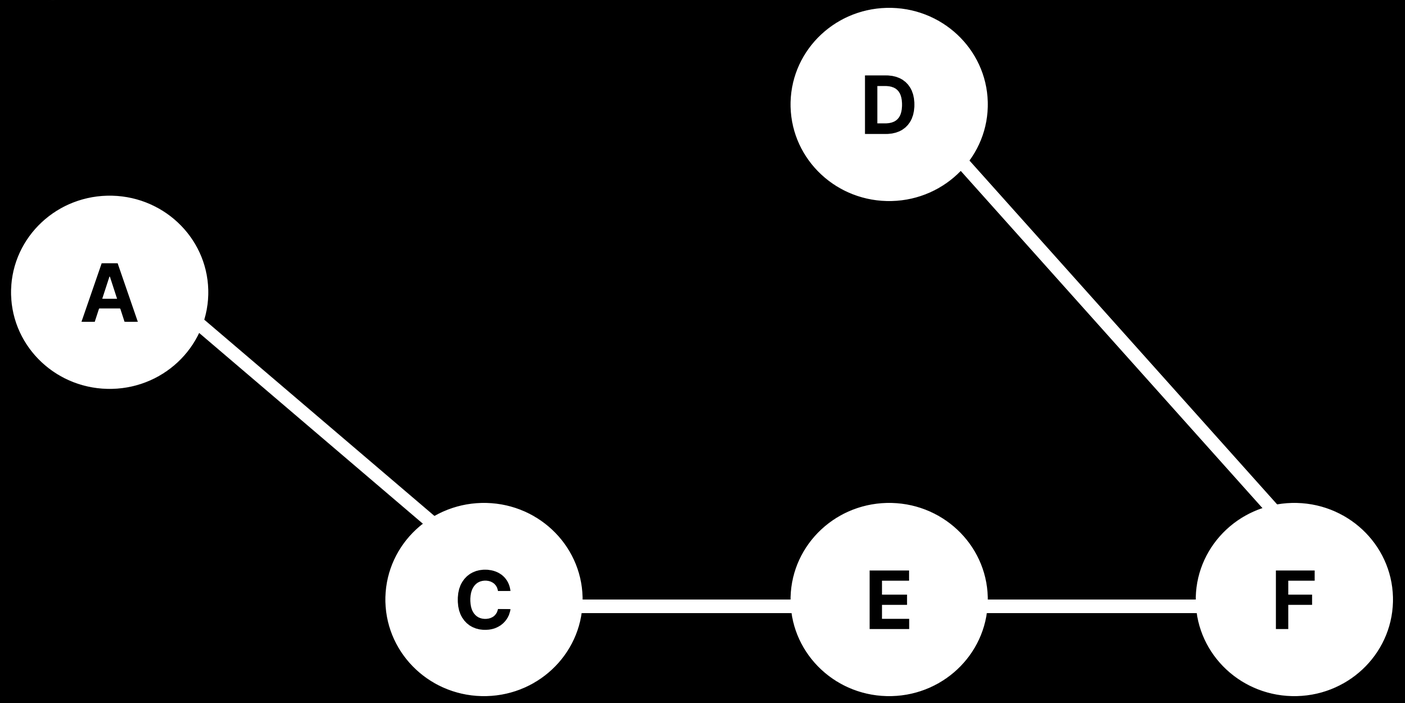
2-sammenhenende grafer

Må fjerne 2 noder for at grafen ikke skal være sammenhende lenger
Har ingen seperasjonsnoder!



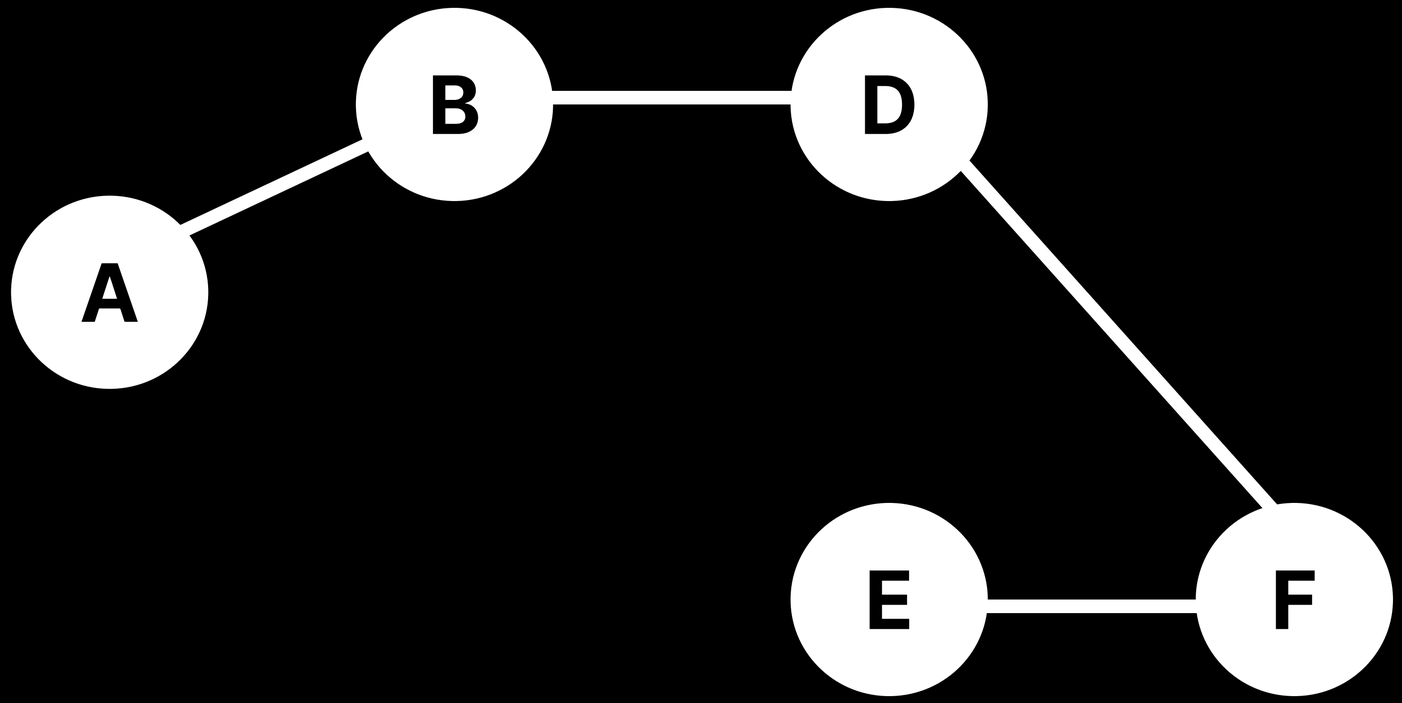
2-sammenhenende grafer

Må fjerne 2 noder for at grafen ikke skal være sammenhende lenger
Har ingen seperasjonsnoder!



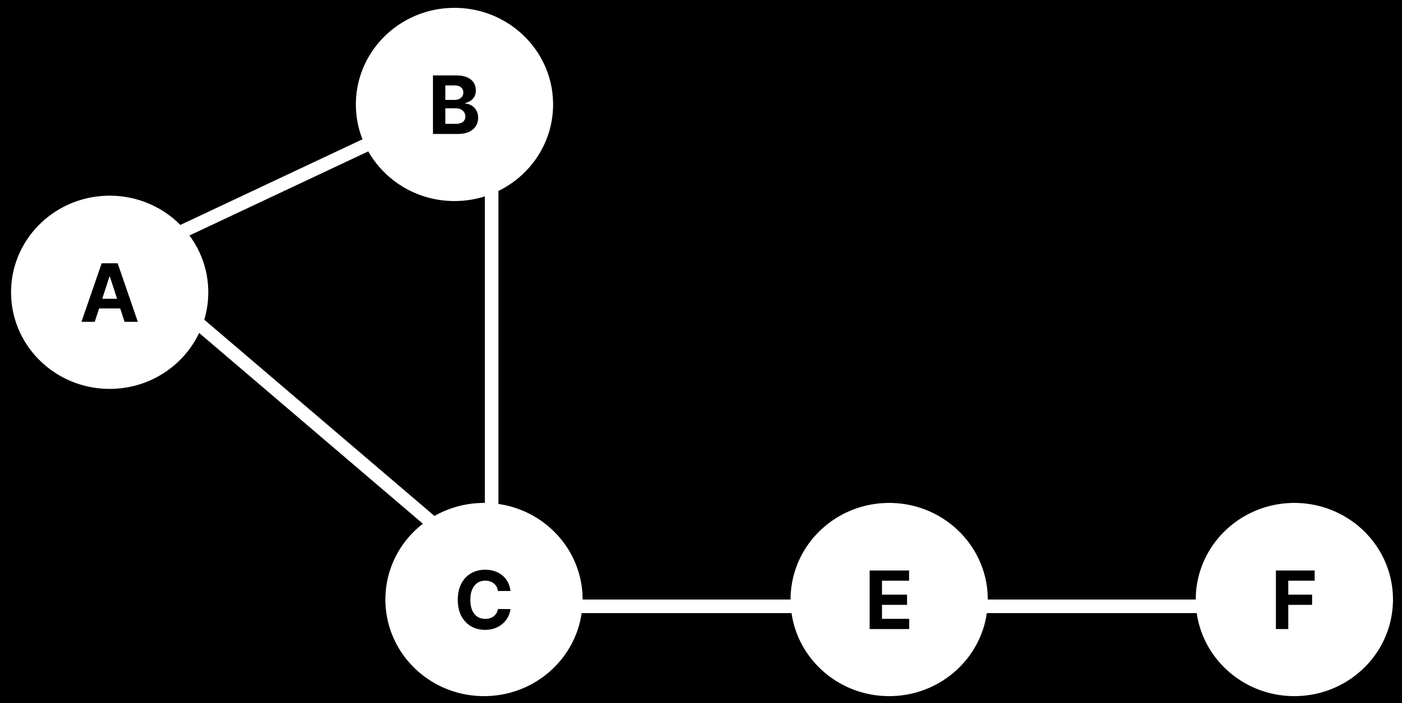
2-sammenhenende grafer

Må fjerne 2 noder for at grafen ikke skal være sammenhende lenger
Har ingen seperasjonsnoder!



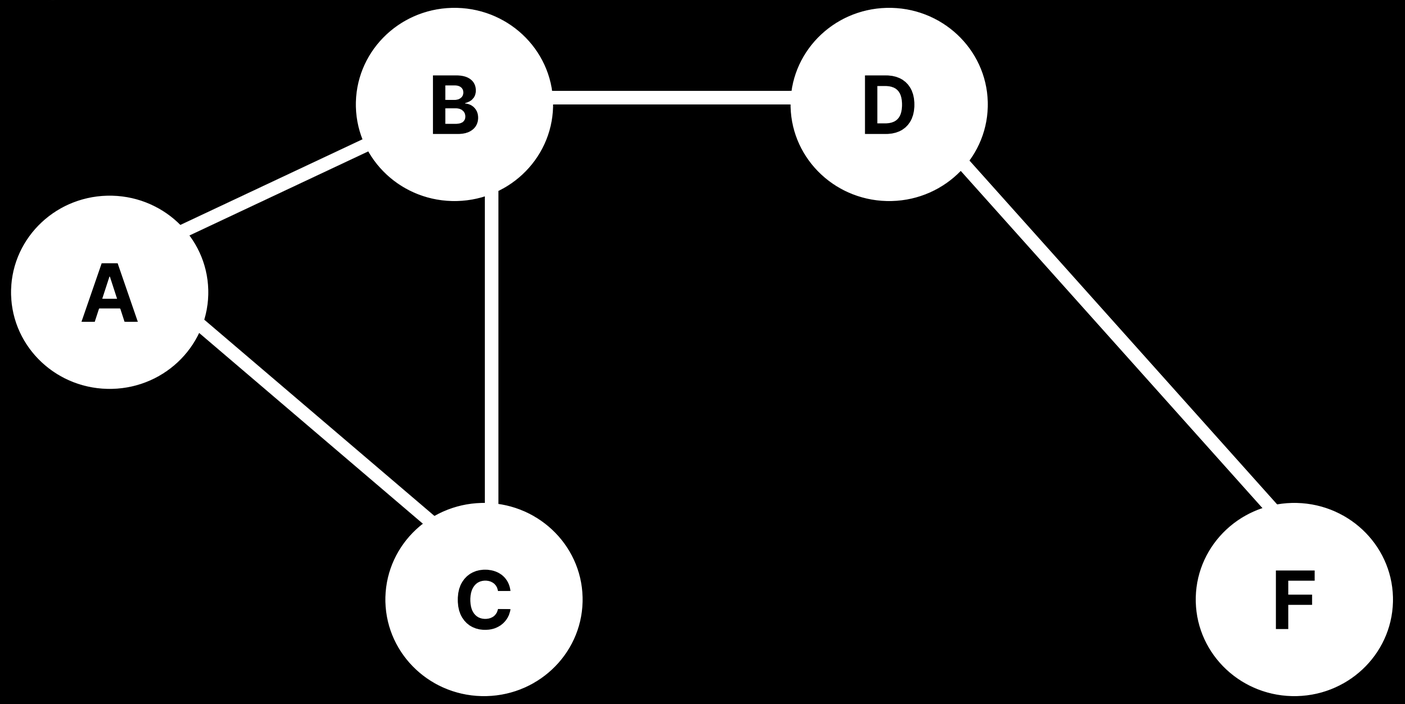
2-sammenhenende grafer

Må fjerne 2 noder for at grafen ikke skal være sammenhende lenger
Har ingen seperasjonsnoder!



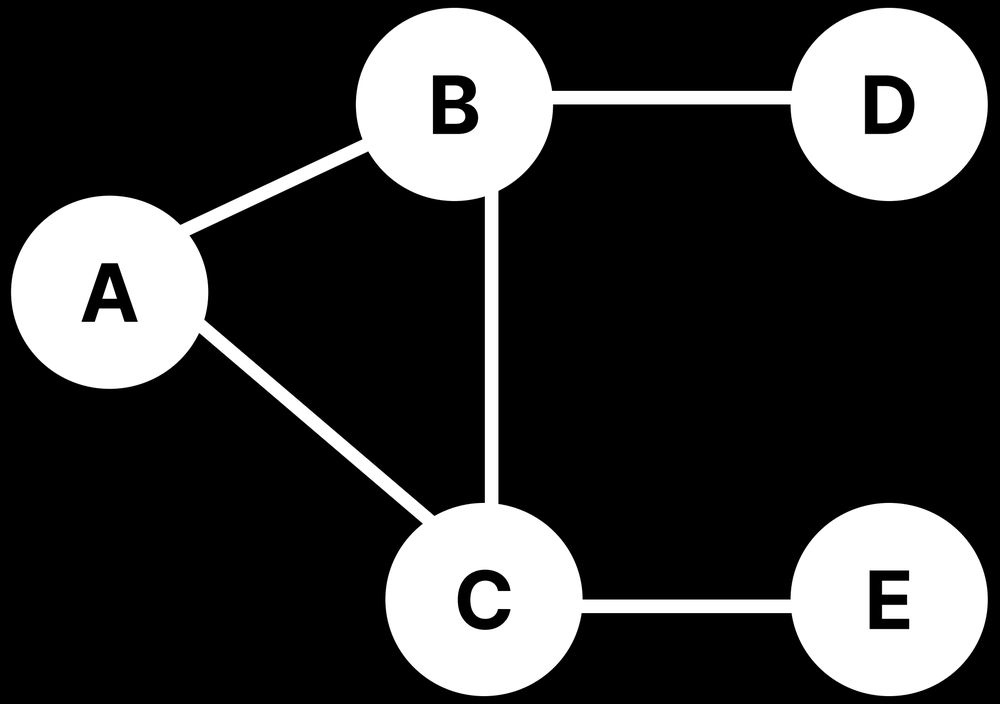
2-sammenhenende grafer

Må fjerne 2 noder for at grafen ikke skal være sammenhende lenger
Har ingen seperasjonsnoder!



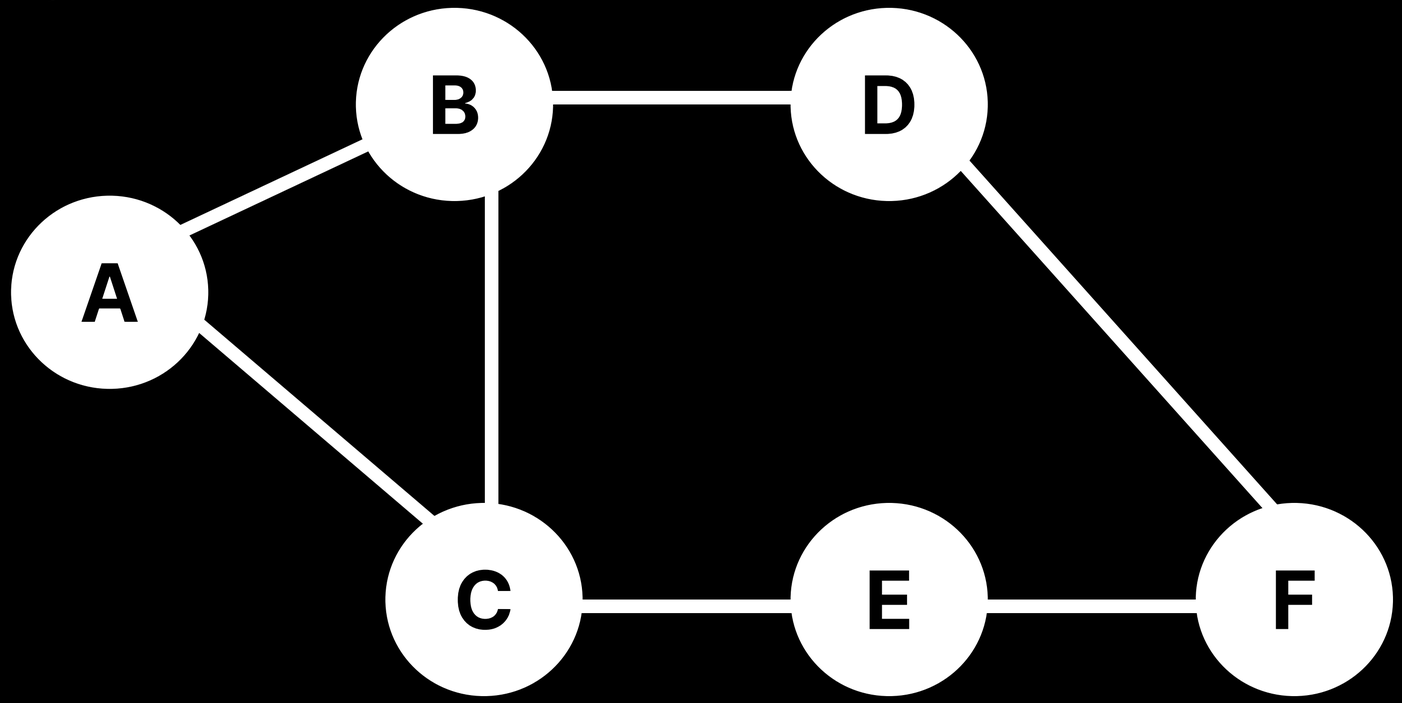
2-sammenhenende grafer

Må fjerne 2 noder for at grafen ikke skal være sammenhende lenger
Har ingen seperasjonsnoder!



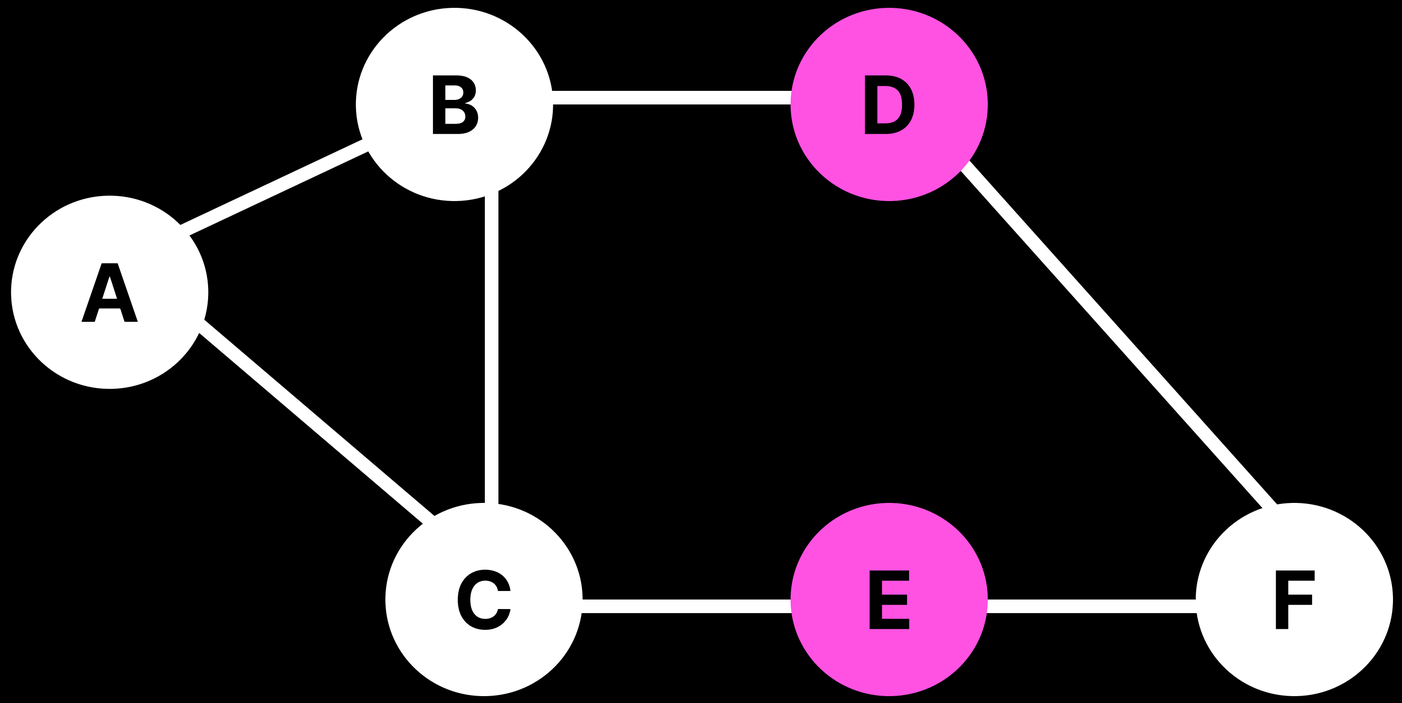
2-sammenhenende grafer

Må fjerne 2 noder for at grafen ikke skal være sammenhende lenger
Har ingen seperasjonsnoder!



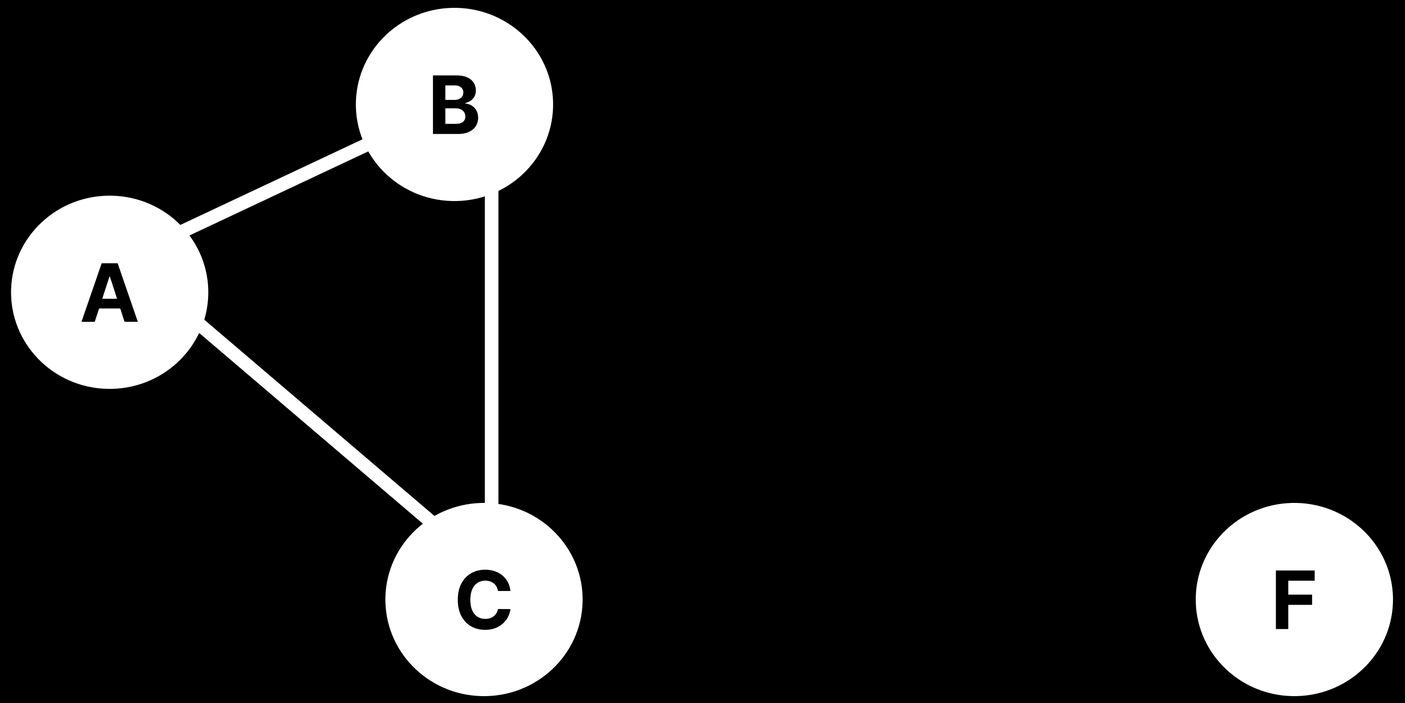
2-sammenhenende grafer

Må fjerne 2 noder for at grafen ikke skal være sammenhende lenger
Har ingen seperasjonsnoder!



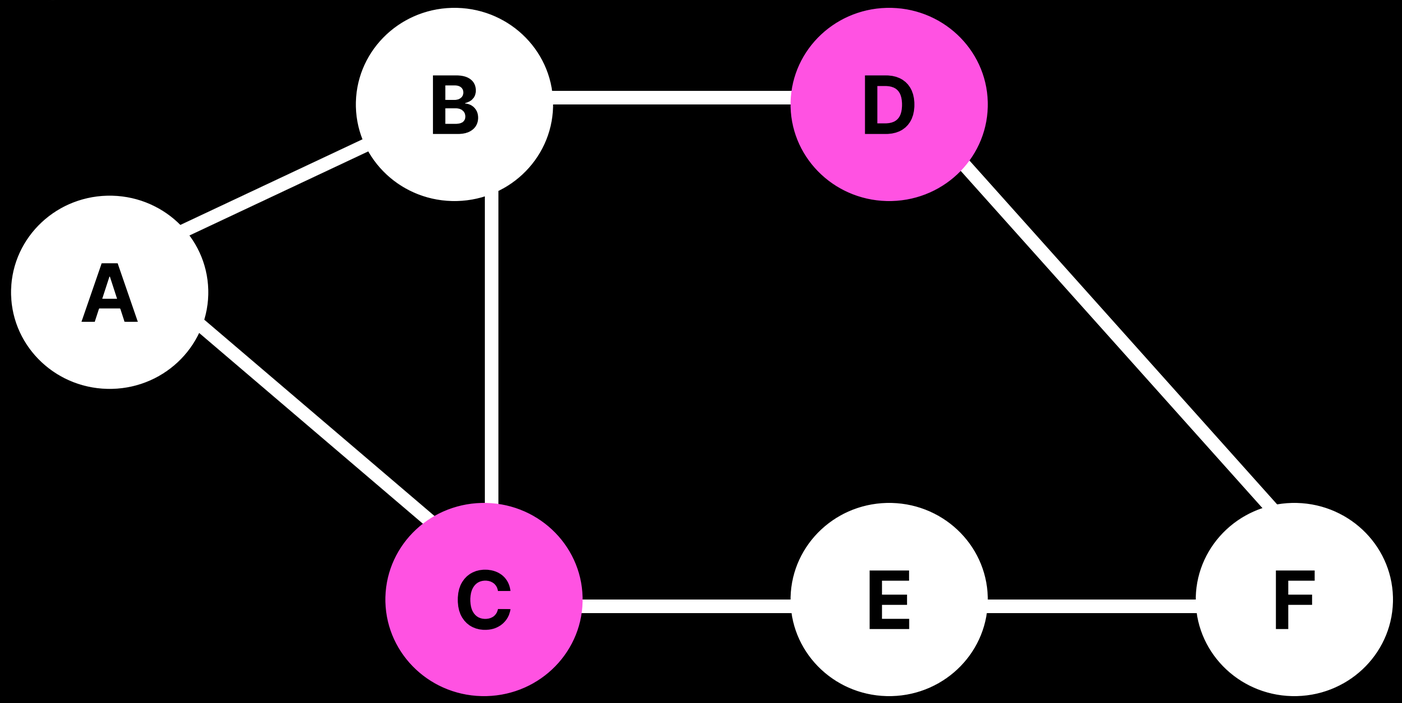
2-sammenhenende grafer

Må fjerne 2 noder for at grafen ikke skal være sammenhende lenger
Har ingen seperasjonsnoder!



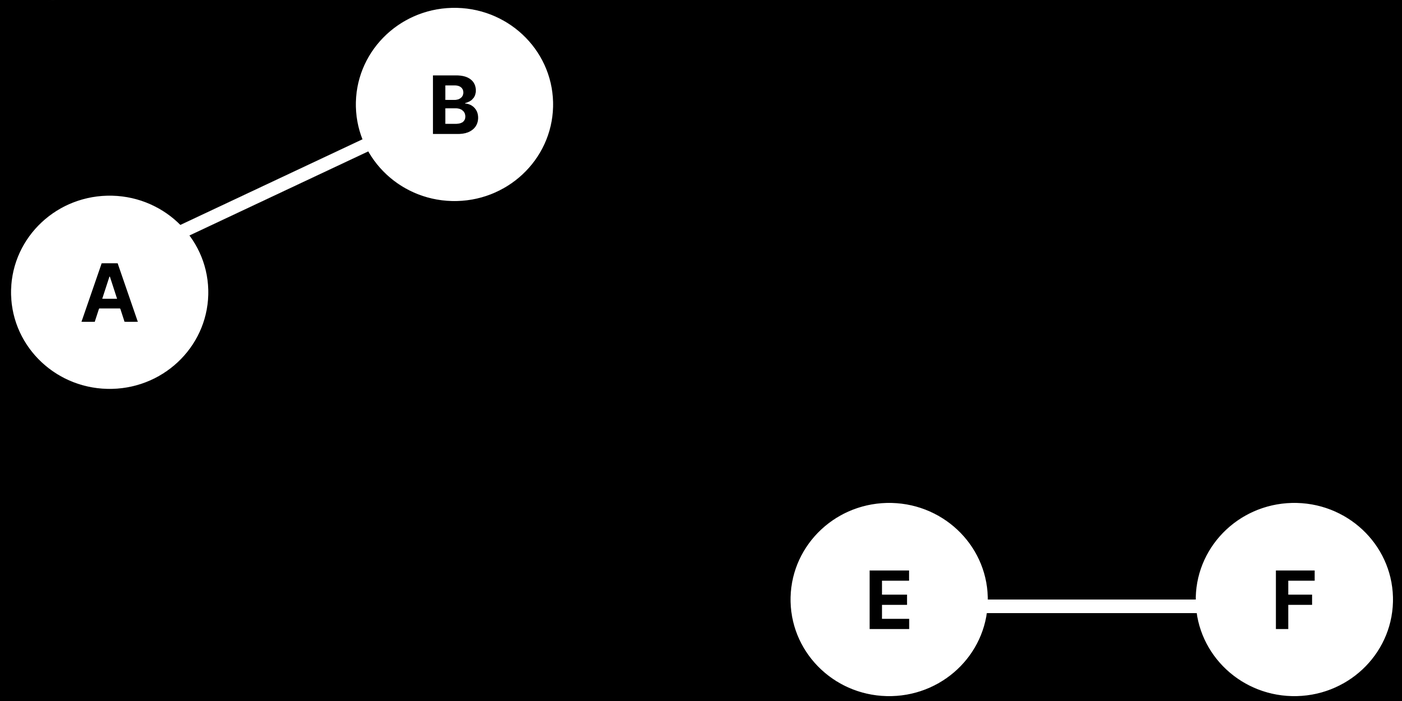
2-sammenhenende grafer

Må fjerne 2 noder for at grafen ikke skal være sammenhende lenger
Har ingen seperasjonsnoder!



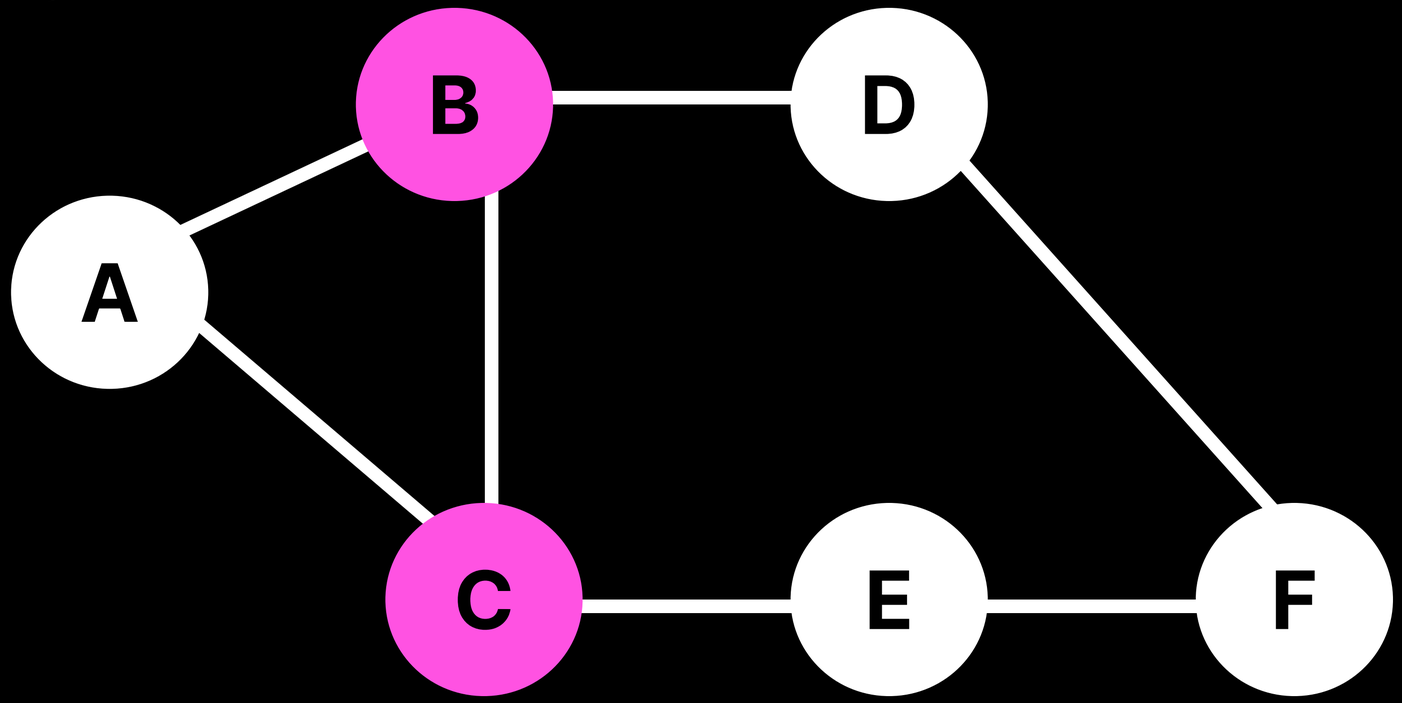
2-sammenhenende grafer

Må fjerne 2 noder for at grafen ikke skal være sammenhende lenger
Har ingen seperasjonsnoder!



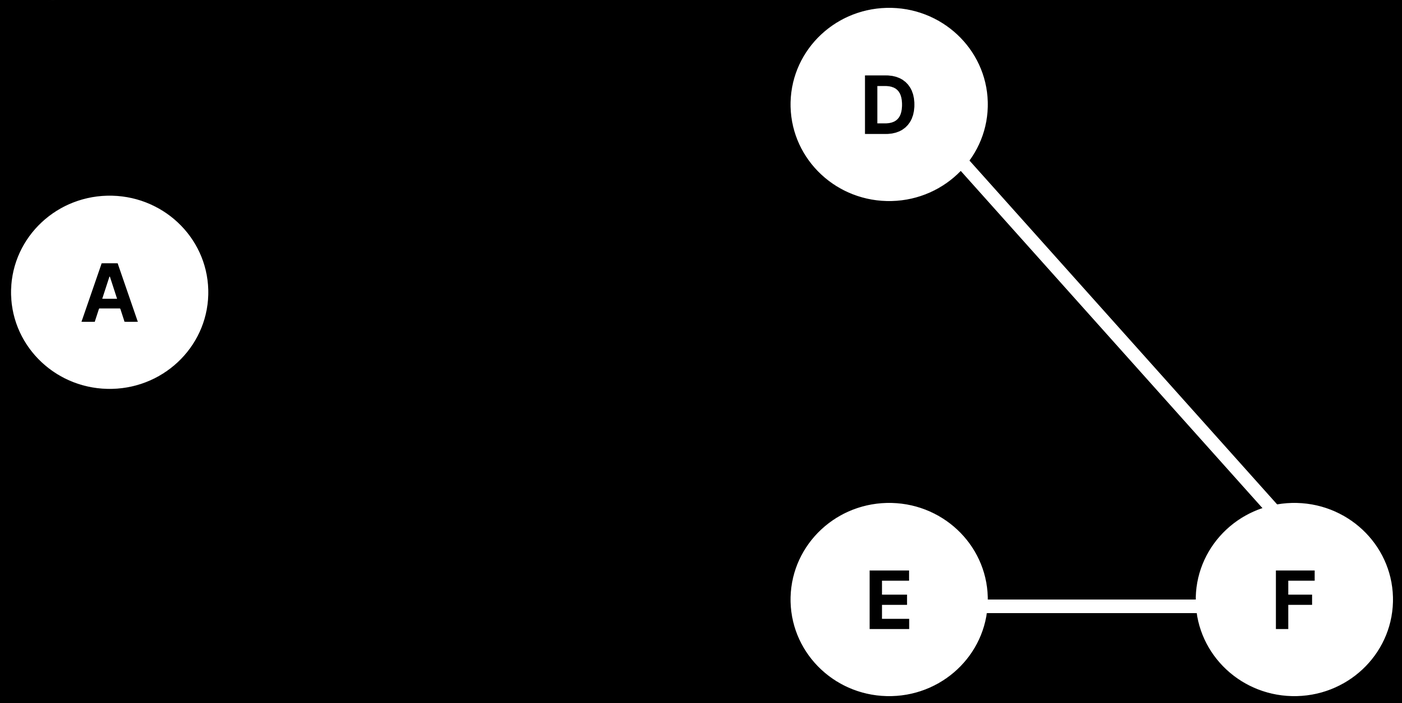
2-sammenhenende grafer

Må fjerne 2 noder for at grafen ikke skal være sammenhende lenger
Har ingen seperasjonsnoder!



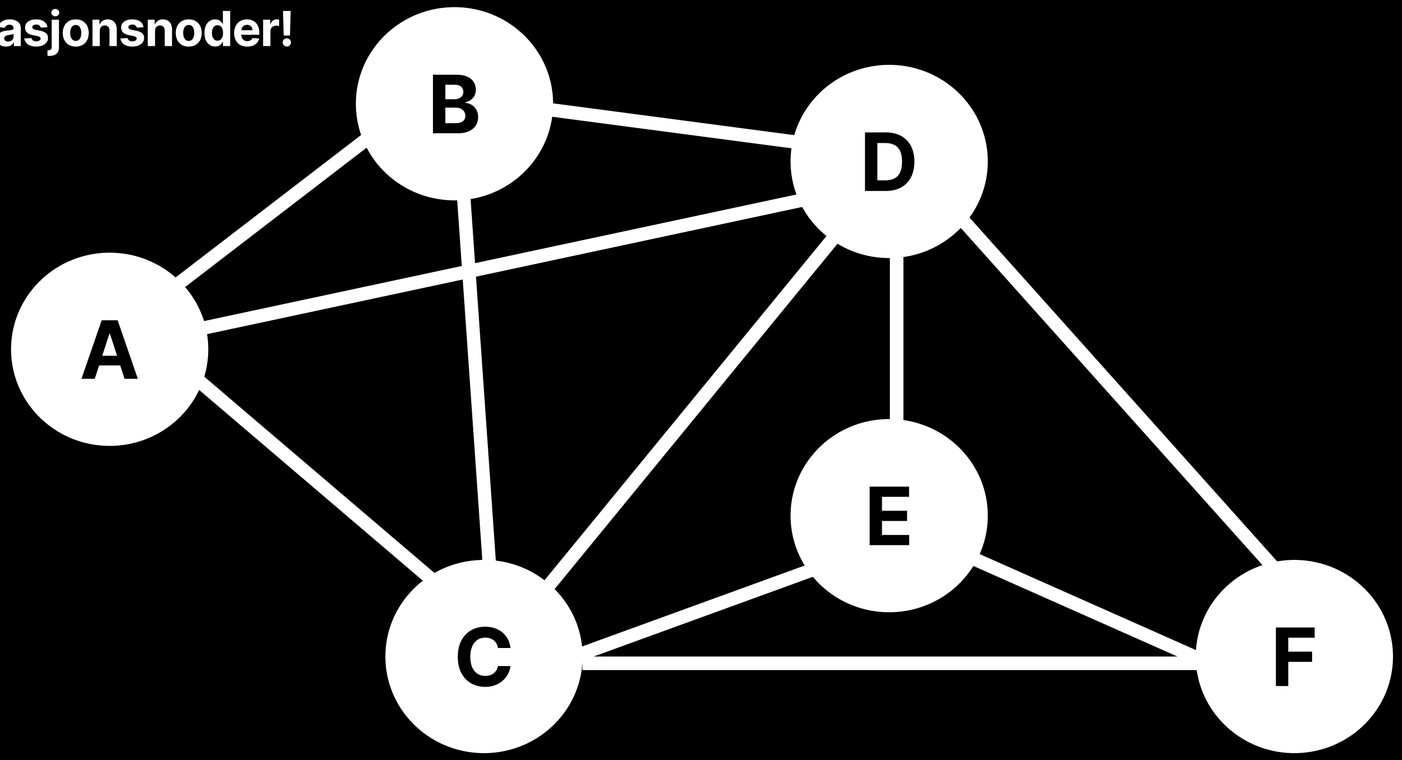
2-sammenhenende grafer

Må fjerne 2 noder for at grafen ikke skal være sammenhende lenger
Har ingen seperasjonsnoder!



k-sammenhenende grafer

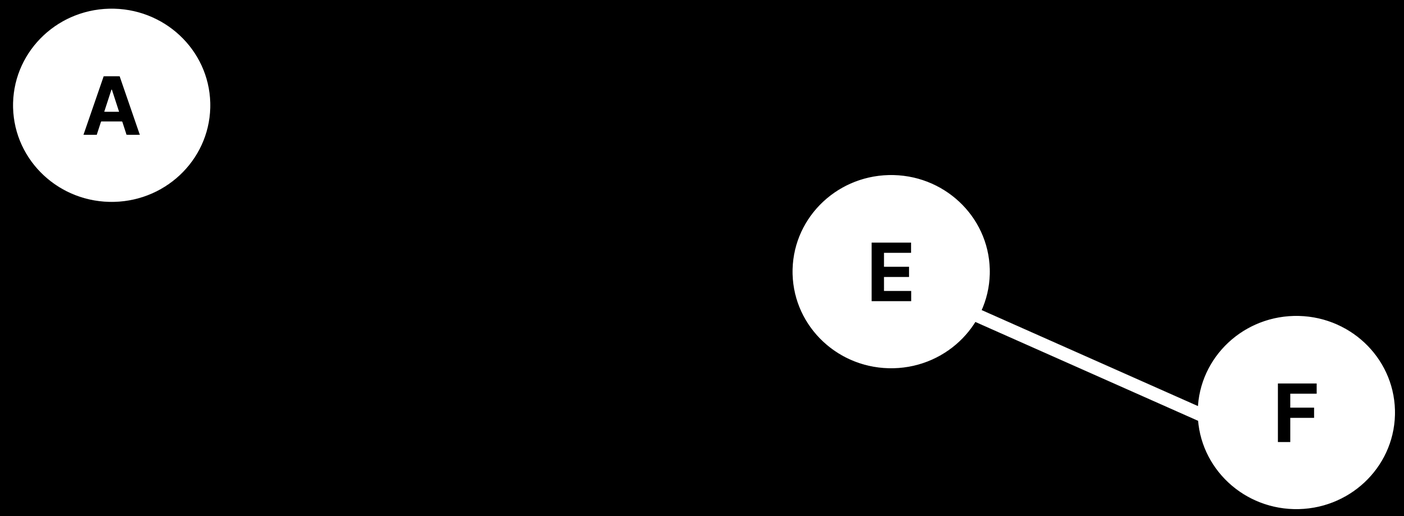
Må fjerne k noder for at grafen ikke skal være sammenhende lenger
Har ingen seperasjonsnoder!



Denne f.eks er 3-sammenhengende

k-sammenhenende grafer

Må fjerne k noder for at grafen ikke skal være sammenhende lenger
Har ingen seperasjonsnoder!

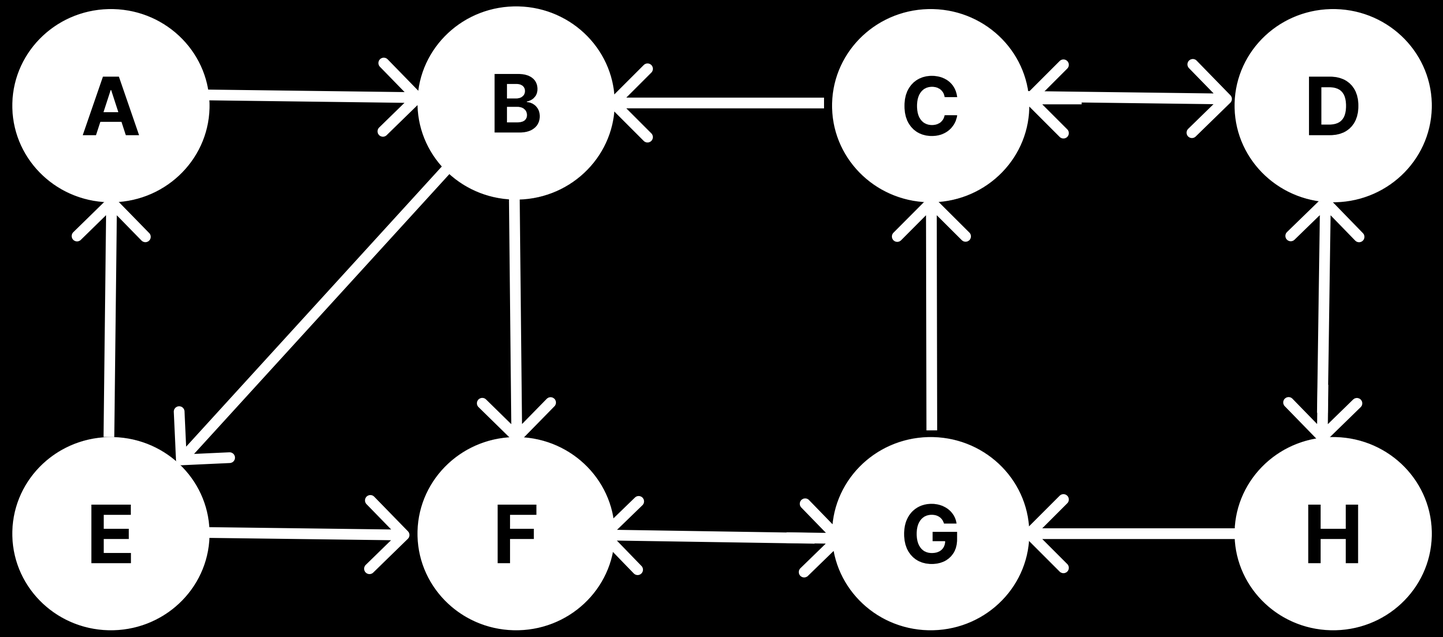


Denne f.eks er 3-sammenhengende
Fjerner vi f.eks B,D og C er den ikke sammenhengende lenger

Sterkt sammenhende grafer

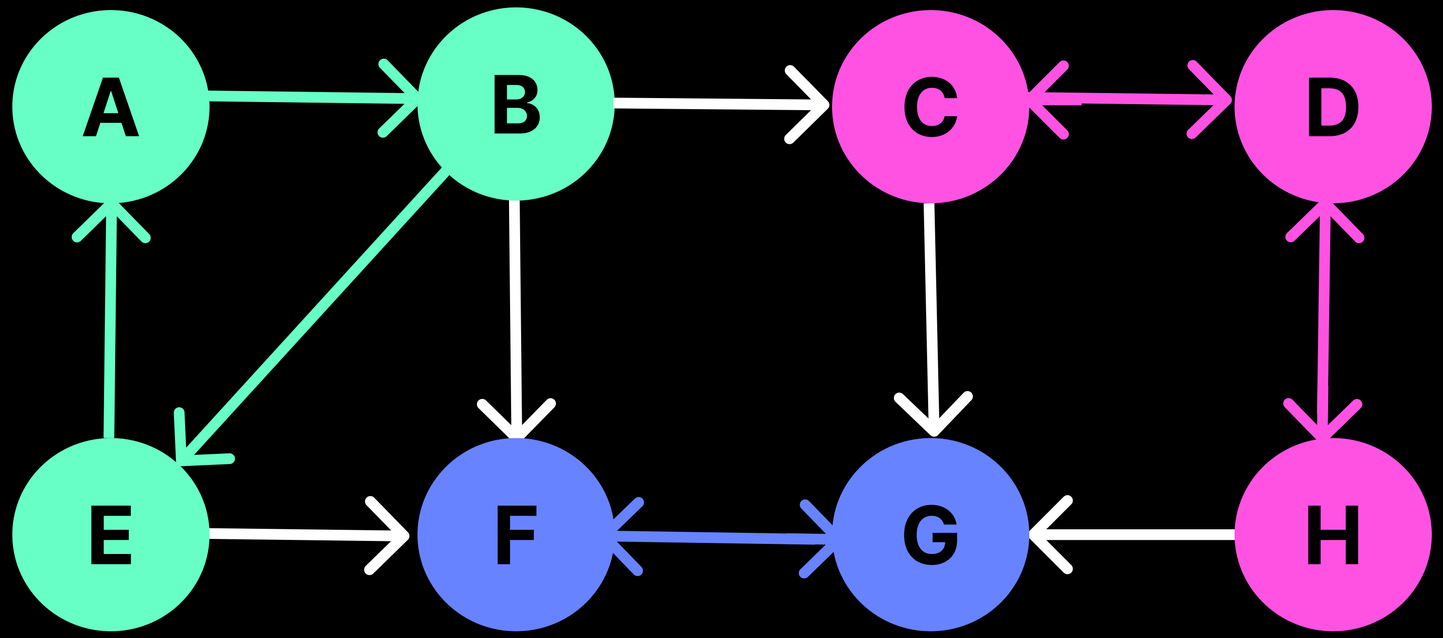
En rettet graf er sterkt sammenhengende om det går en rettet sti fra enhver node til en hver annen node

Kan bruke DFS til å finne ut om en rettet graf er sterkt sammenhengende.



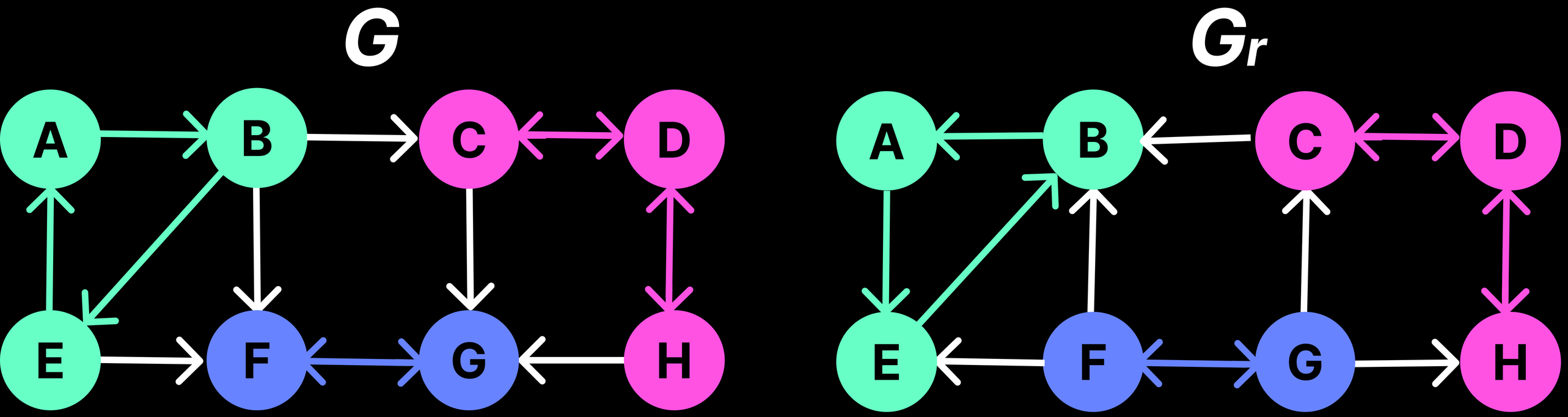
Sterkt sammenhende komponenter

Tenk på de som sykler



Den reverserte grafen

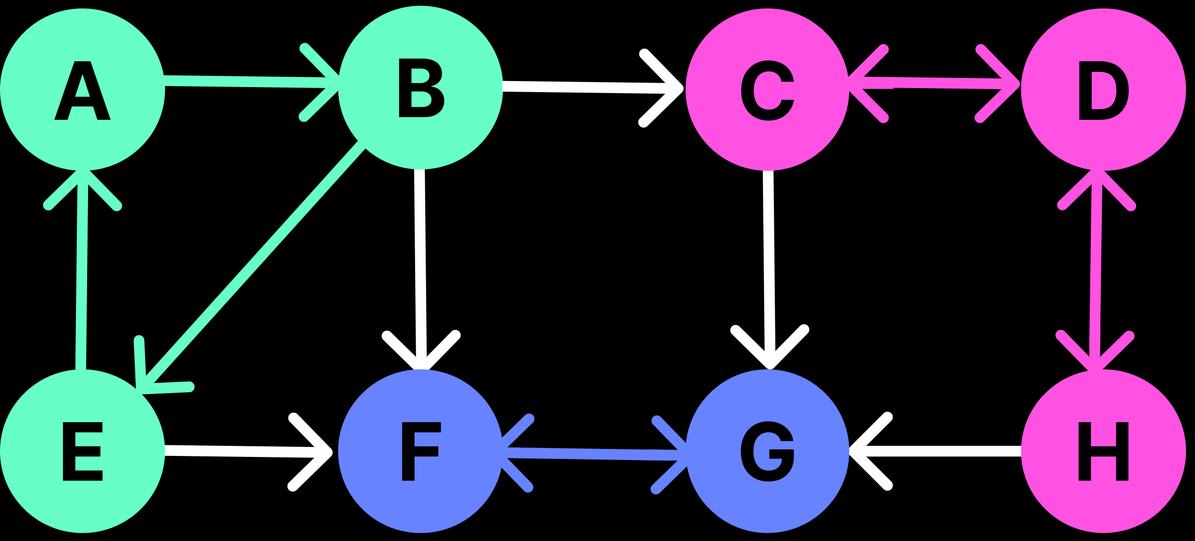
Alle kanter er snudd
De sterkt sammenhengende komponentene er de samme!

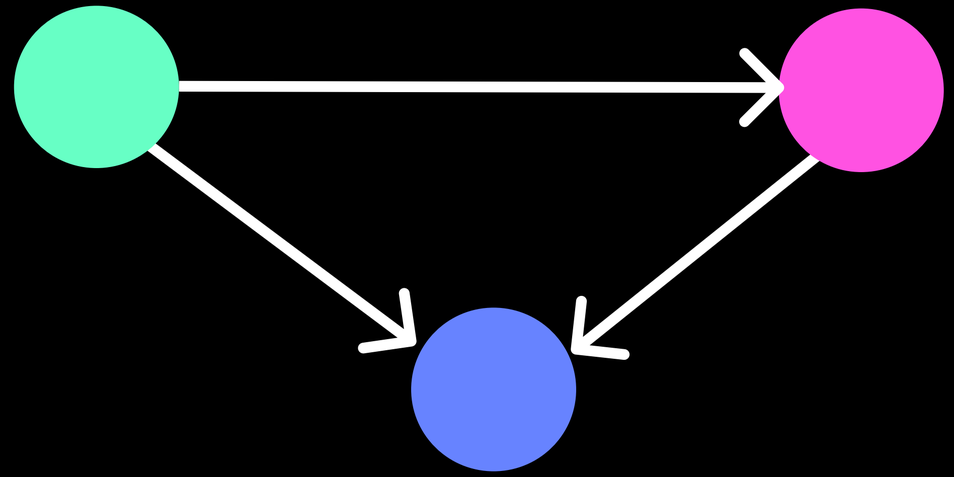


Kan finne sterkt sammenhengende komponenter med den reverserte grafen.
Alle noder som kan nås fra en node v i både G og i Gr er den sterkt sammenhengende komponenten til v.

Komponent grafen

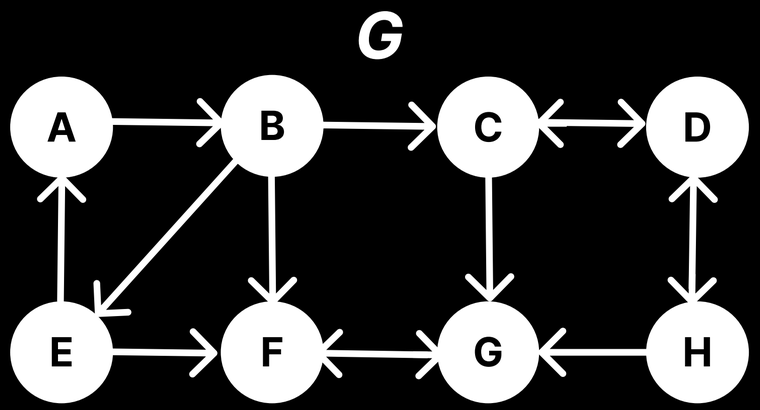
Alltid asyklisk





Kosaraju

1. DFSTopSort: DFS, der hver node legges på en stack etter alle noder er besøkt
1. Lager den reverserte grafen
2. DFS på den reverserte grafen. Rekkefølgen er gitt i stacken fra DFSTopSort



DFSTopSort: Gjør DFS først

ALGORITHM: TOPOLOGISK SORTERING VED DFS

Input: En rettet asyklisk graf $G = (V, E)$
Output: En topologisk ordning av nodene i G

```
1  Procedure DFSTopSort(G)
2      stack ← empty stack
3      visited ← empty set
4      for u ∈ V do
5          if u ∉ visited then
6              DFSVisit(G, u, visited, stack)
7      return stack
8
9  Procedure DFSVisit(G, u, visited, stack)
10     add u to visited
11     for (u, v) ∈ E do
12         if v ∉ visited then
13             DFSVisit(G, v, visited, stack)
14     stack.push(u)
```

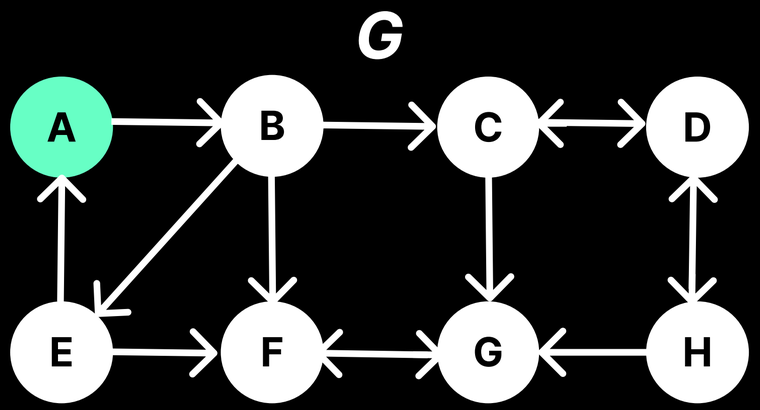
ALGORITHM: FINN DE STERKT SAMMENHENGENDE KOMPONENETENE AV EN GRAF

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
```

Kosaraju

1. DFSTopSort: DFS, der hver node legges på en stack etter alle noder er besøkt
1. Lager den reverserte grafen
2. DFS på den reverserte grafen. Rekkefølgen er gitt i stacken fra DFSTopSort



DFSTopSort: Gjør DFS først

ALGORITHM: TOPOLOGISK SORTERING VED DFS

Input: En rettet asyklisk graf $G = (V, E)$
Output: En topologisk ordning av nodene i G

```
1  Procedure DFSTopSort(G)
2      stack ← empty stack
3      visited ← empty set
4      for u ∈ V do
5          if u ∉ visited then
6              DFSVisit(G, u, visited, stack)
7      return stack
8
9  Procedure DFSVisit(G, u, visited, stack)
10     add u to visited
11     for (u, v) ∈ E do
12         if v ∉ visited then
13             DFSVisit(G, v, visited, stack)
14     stack.push(u)
```

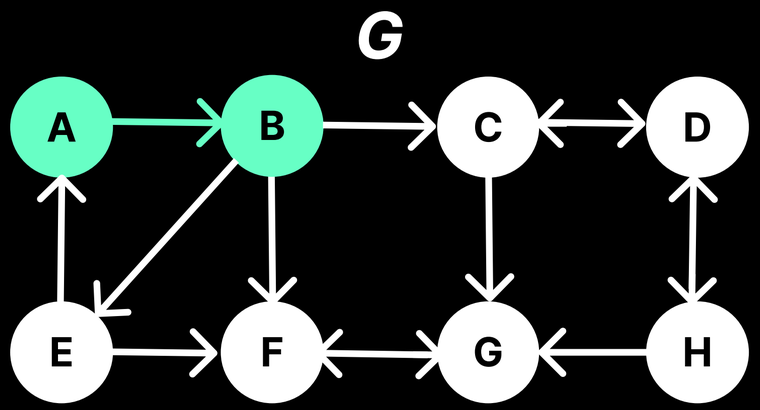
ALGORITHM: FINN DE STERKT SAMMENHENGENDE KOMPONENETENE AV EN GRAF

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
```

Kosaraju

1. DFSTopSort: DFS, der hver node legges på en stack etter alle noder er besøkt
1. Lager den reverserte grafen
2. DFS på den reverserte grafen. Rekkefølgen er gitt i stacken fra DFSTopSort



DFSTopSort: Gjør DFS først

ALGORITHM: TOPOLOGISK SORTERING VED DFS

Input: En rettet asyklisk graf $G = (V, E)$
Output: En topologisk ordning av nodene i G

```
1  Procedure DFSTopSort(G)
2      stack ← empty stack
3      visited ← empty set
4      for u ∈ V do
5          if u ∉ visited then
6              DFSVisit(G, u, visited, stack)
7      return stack
8
9  Procedure DFSVisit(G, u, visited, stack)
10     add u to visited
11     for (u, v) ∈ E do
12         if v ∉ visited then
13             DFSVisit(G, v, visited, stack)
14     stack.push(u)
```

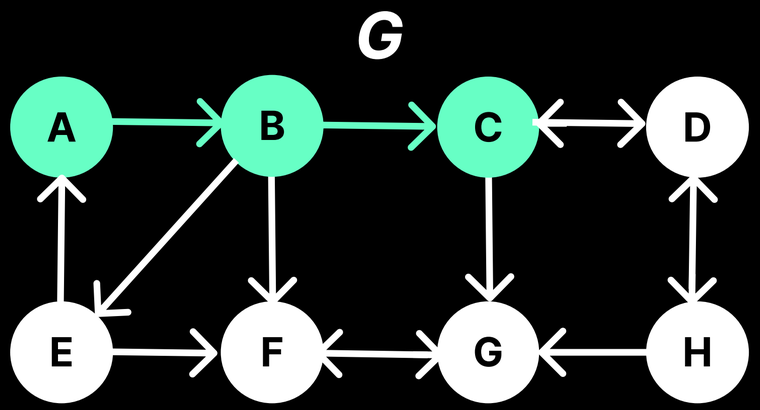
ALGORITHM: FINN DE STERKT SAMMENHENGENDE KOMPONENETENE AV EN GRAF

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
```

Kosaraju

1. DFSTopSort: DFS, der hver node legges på en stack etter alle noder er besøkt
1. Lager den reverserte grafen
2. DFS på den reverserte grafen. Rekkefølgen er gitt i stacken fra DFSTopSort



DFSTopSort: Gjør DFS først

ALGORITHM: TOPOLOGISK SORTERING VED DFS

Input: En rettet asyklisk graf $G = (V, E)$
Output: En topologisk ordning av nodene i G

```
1  Procedure DFSTopSort(G)
2      stack ← empty stack
3      visited ← empty set
4      for u ∈ V do
5          if u ∉ visited then
6              DFSVisit(G, u, visited, stack)
7      return stack
8
9  Procedure DFSVisit(G, u, visited, stack)
10     add u to visited
11     for (u, v) ∈ E do
12         if v ∉ visited then
13             DFSVisit(G, v, visited, stack)
14     stack.push(u)
```

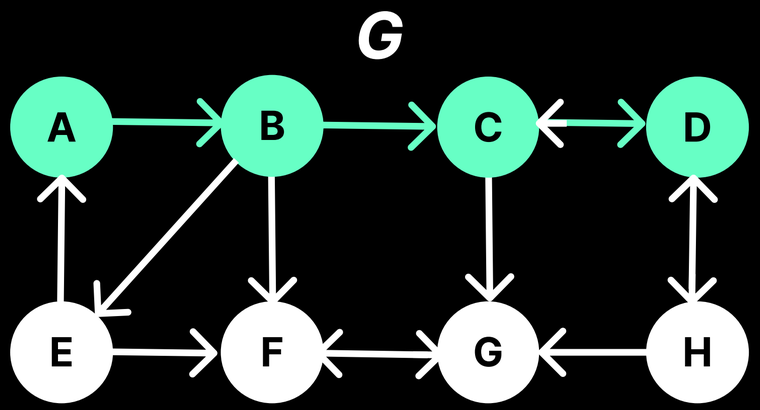
ALGORITHM: FINN DE STERKT SAMMENHENGENDE KOMPONENETENE AV EN GRAF

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
```

Kosaraju

1. DFSTopSort: DFS, der hver node legges på en stack etter alle noder er besøkt
1. Lager den reverserte grafen
2. DFS på den reverserte grafen. Rekkefølgen er gitt i stacken fra DFSTopSort



DFSTopSort: Gjør DFS først

ALGORITHM: TOPOLOGISK SORTERING VED DFS

Input: En rettet asyklisk graf $G = (V, E)$
Output: En topologisk ordning av nodene i G

```
1  Procedure DFSTopSort(G)
2      stack ← empty stack
3      visited ← empty set
4      for u ∈ V do
5          if u ∉ visited then
6              DFSVisit(G, u, visited, stack)
7      return stack
8
9  Procedure DFSVisit(G, u, visited, stack)
10     add u to visited
11     for (u, v) ∈ E do
12         if v ∉ visited then
13             DFSVisit(G, v, visited, stack)
14     stack.push(u)
```

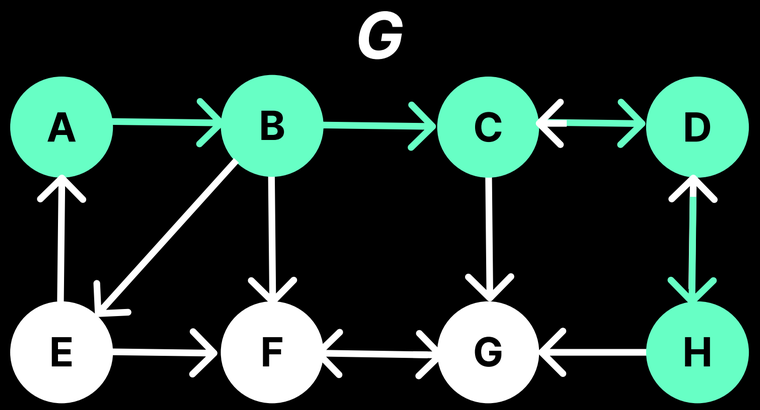
ALGORITHM: FINN DE STERKT SAMMENHENGENDE KOMPONENETENE AV EN GRAF

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
```

Kosaraju

1. DFSTopSort: DFS, der hver node legges på en stack etter alle noder er besøkt
1. Lager den reverserte grafen
2. DFS på den reverserte grafen. Rekkefølgen er gitt i stacken fra DFSTopSort



DFSTopSort: Gjør DFS først

ALGORITHM: TOPOLOGISK SORTERING VED DFS

Input: En rettet asyklisk graf $G = (V, E)$
Output: En topologisk ordning av nodene i G

```
1  Procedure DFSTopSort(G)
2      stack ← empty stack
3      visited ← empty set
4      for u ∈ V do
5          if u ∉ visited then
6              DFSVisit(G, u, visited, stack)
7      return stack
8
9  Procedure DFSVisit(G, u, visited, stack)
10     add u to visited
11     for (u, v) ∈ E do
12         if v ∉ visited then
13             DFSVisit(G, v, visited, stack)
14     stack.push(u)
```

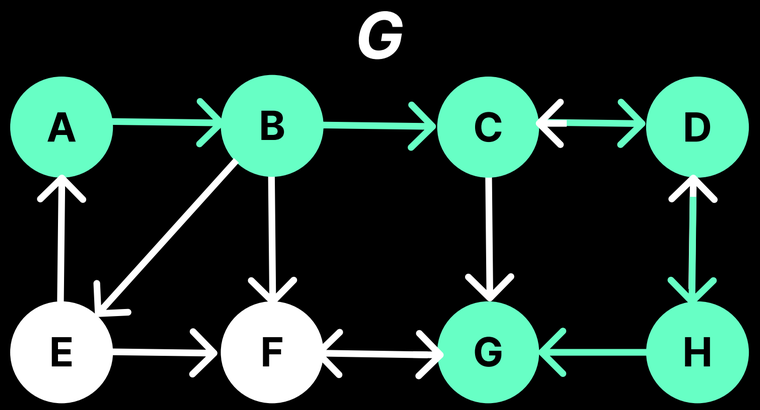
ALGORITHM: FINN DE STERKT SAMMENHENGENDE KOMPONENETENE AV EN GRAF

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
```

Kosaraju

1. DFSTopSort: DFS, der hver node legges på en stack etter alle noder er besøkt
1. Lager den reverserte grafen
2. DFS på den reverserte grafen. Rekkefølgen er gitt i stacken fra DFSTopSort



DFSTopSort: Gjør DFS først

ALGORITHM: TOPOLOGISK SORTERING VED DFS

Input: En rettet asyklisk graf $G = (V, E)$
Output: En topologisk ordning av nodene i G

```
1  Procedure DFSTopSort(G)
2      stack ← empty stack
3      visited ← empty set
4      for u ∈ V do
5          if u ∉ visited then
6              DFSVisit(G, u, visited, stack)
7      return stack
8
9  Procedure DFSVisit(G, u, visited, stack)
10     add u to visited
11     for (u, v) ∈ E do
12         if v ∉ visited then
13             DFSVisit(G, v, visited, stack)
14     stack.push(u)
```

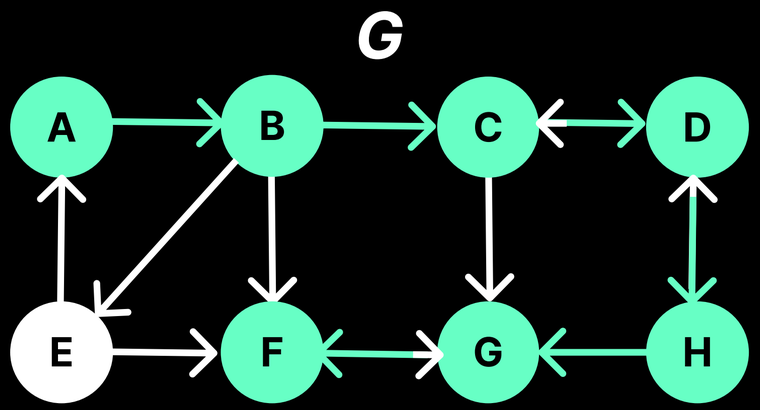
ALGORITHM: FINN DE STERKT SAMMENHENGENDE KOMPONENETENE AV EN GRAF

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
```

Kosaraju

1. DFSTopSort: DFS, der hver node legges på en stack etter alle noder er besøkt
1. Lager den reverserte grafen
2. DFS på den reverserte grafen. Rekkefølgen er gitt i stacken fra DFSTopSort



DFSTopSort: Gjør DFS først
For hvert steg tilbake legges noden til på stacken

ALGORITHM: TOPOLOGISK SORTERING VED DFS

Input: En rettet asyklisk graf $G = (V, E)$
Output: En topologisk ordning av nodene i G

```
1  Procedure DFSTopSort(G)
2      stack ← empty stack
3      visited ← empty set
4      for u ∈ V do
5          if u ∉ visited then
6              DFSVisit(G, u, visited, stack)
7      return stack
8
9  Procedure DFSVisit(G, u, visited, stack)
10     add u to visited
11     for (u, v) ∈ E do
12         if v ∉ visited then
13             DFSVisit(G, v, visited, stack)
14     stack.push(u)
```

ALGORITHM: FINN DE STERKT SAMMENHENGENDE KOMPONENETENE AV EN GRAF

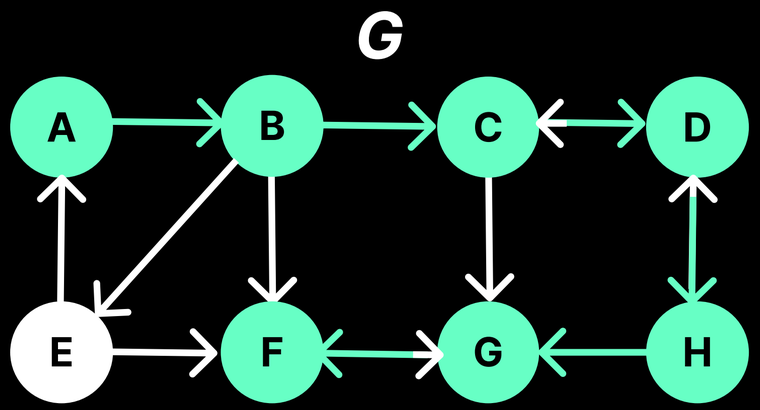
Input: En rettet graf $G = (V, E)$
Output: Returner de sterkt sammenhengende komponenetene til G

```
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
```

stack

Kosaraju

1. DFSTopSort: DFS, der hver node legges på en stack etter alle noder er besøkt
1. Lager den reverserte grafen
2. DFS på den reverserte grafen. Rekkefølgen er gitt i stacken fra DFSTopSort



DFSTopSort: Gjør DFS først
For hvert steg tilbake legges noden til på stacken

ALGORITHM: TOPOLOGISK SORTERING VED DFS

Input: En rettet asyklisk graf $G = (V, E)$
Output: En topologisk ordning av nodene i G

```
1  Procedure DFSTopSort(G)
2      stack ← empty stack
3      visited ← empty set
4      for u ∈ V do
5          if u ∉ visited then
6              DFSVisit(G, u, visited, stack)
7      return stack
8
9  Procedure DFSVisit(G, u, visited, stack)
10     add u to visited
11     for (u, v) ∈ E do
12         if v ∉ visited then
13             DFSVisit(G, v, visited, stack)
14     stack.push(u)
```

ALGORITHM: FINN DE STERKT SAMMENHENGENDE KOMPONENETENE AV EN GRAF

Input: En rettet graf $G = (V, E)$
Output: Returner de sterkt sammenhengende komponenetene til G

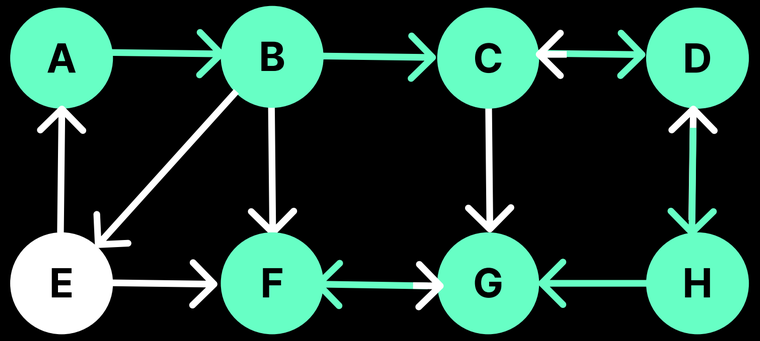
```
1  Procedure StronglyConnectedComponents(G)
2      stack ← DFSTopSort(G)
3      G_r ← ReverseGraph(G)
4      visited ← empty set
5      components ← empty set
6      while stack is not empty do
7          u ← stack.pop()
8          if u ∉ visited then
9              component ← empty set
10     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

F
stack

Kosaraju

1. DFSTopSort: DFS, der hver node legges på en stack etter alle noder er besøkt
1. Lager den reverserte grafen
2. DFS på den reverserte grafen. Rekkefølgen er gitt i stacken fra DFSTopSort

G



DFSTopSort: Gjør DFS først
For hvert steg tilbake legges noden til på stacken

ALGORITHM: TOPOLOGISK SORTERING VED DFS

Input: En rettet asyklisk graf $G = (V, E)$
Output: En topologisk ordning av nodene i G

```
1  Procedure DFSTopSort(G)
2      stack ← empty stack
3      visited ← empty set
4      for u ∈ V do
5          if u ∉ visited then
6              DFSVisit(G, u, visited, stack)
7      return stack
8
9  Procedure DFSVisit(G, u, visited, stack)
10     add u to visited
11     for (u, v) ∈ E do
12         if v ∉ visited then
13             DFSVisit(G, v, visited, stack)
14     stack.push(u)
```

ALGORITHM: FINN DE STERKT SAMMENHENGENDE KOMPONENETENE AV EN GRAF

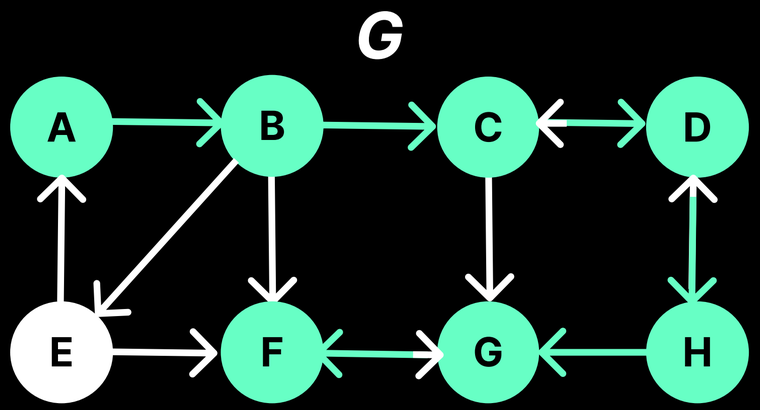
Input: En rettet graf $G = (V, E)$
Output: Returner de sterkt sammenhengende komponenetene til G

```
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
```

G
F
stack

Kosaraju

1. DFSTopSort: DFS, der hver node legges på en stack etter alle noder er besøkt
1. Lager den reverserte grafen
2. DFS på den reverserte grafen. Rekkefølgen er gitt i stacken fra DFSTopSort



DFSTopSort: Gjør DFS først
For hvert steg tilbake legges noden til på stacken

ALGORITHM: TOPOLOGISK SORTERING VED DFS

Input: En rettet asyklisk graf $G = (V, E)$
Output: En topologisk ordning av nodene i G

```
1  Procedure DFSTopSort(G)
2      stack ← empty stack
3      visited ← empty set
4      for u ∈ V do
5          if u ∉ visited then
6              DFSVisit(G, u, visited, stack)
7      return stack
8
9  Procedure DFSVisit(G, u, visited, stack)
10     add u to visited
11     for (u, v) ∈ E do
12         if v ∉ visited then
13             DFSVisit(G, v, visited, stack)
14     stack.push(u)
```

ALGORITHM: FINN DE STERKT SAMMENHENGENDE KOMPONENETENE AV EN GRAF

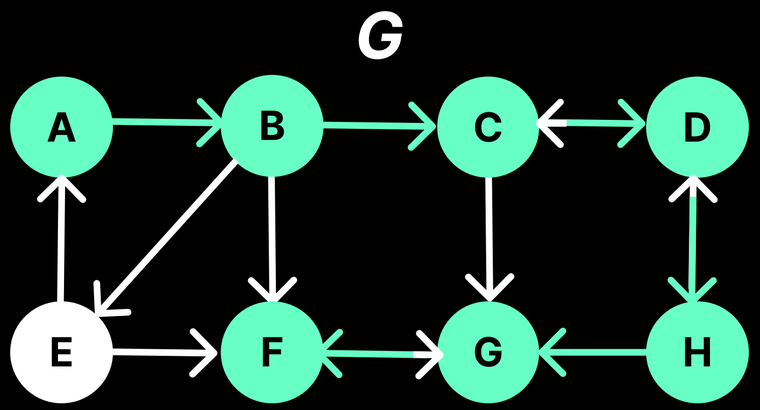
Input: En rettet graf $G = (V, E)$
Output: Returner de sterkt sammenhengende komponenetene til G

```
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
```

H
G
F
stack

Kosaraju

1. DFSTopSort: DFS, der hver node legges på en stack etter alle noder er besøkt
1. Lager den reverserte grafen
2. DFS på den reverserte grafen. Rekkefølgen er gitt i stacken fra DFSTopSort



DFSTopSort: Gjør DFS først
For hvert steg tilbake legges noden til på stacken

ALGORITHM: TOPOLOGISK SORTERING VED DFS

Input: En rettet asyklisk graf $G = (V, E)$
Output: En topologisk ordning av nodene i G

```
1  Procedure DFSTopSort(G)
2      stack ← empty stack
3      visited ← empty set
4      for u ∈ V do
5          if u ∉ visited then
6              DFSVisit(G, u, visited, stack)
7      return stack
8
9  Procedure DFSVisit(G, u, visited, stack)
10     add u to visited
11     for (u, v) ∈ E do
12         if v ∉ visited then
13             DFSVisit(G, v, visited, stack)
14     stack.push(u)
```

D
H
G
F
stack

ALGORITHM: FINN DE STERKT SAMMENHENGENDE KOMPONENETENE AV EN GRAF

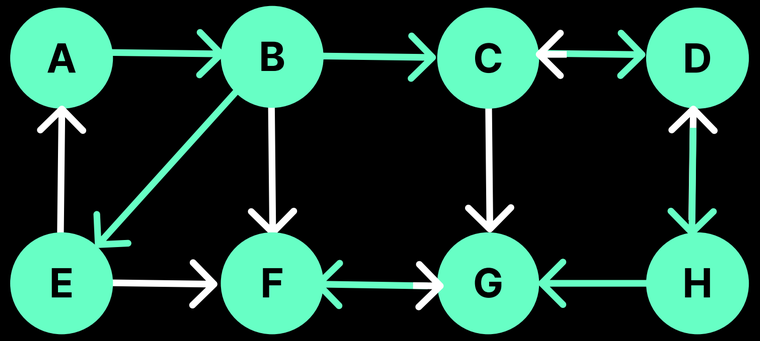
Input: En rettet graf $G = (V, E)$
Output: Returner de sterkt sammenhengende komponenetene til G

```
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
```

Kosaraju

1. DFSTopSort: DFS, der hver node legges på en stack etter alle noder er besøkt
1. Lager den reverserte grafen
2. DFS på den reverserte grafen. Rekkefølgen er gitt i stacken fra DFSTopSort

G



DFSTopSort: Gjør DFS først

ALGORITHM: TOPOLOGISK SORTERING VED DFS

Input: En rettet asyklisk graf $G = (V, E)$
Output: En topologisk ordning av nodene i G

```
1  Procedure DFSTopSort(G)
2      stack ← empty stack
3      visited ← empty set
4      for u ∈ V do
5          if u ∉ visited then
6              DFSVisit(G, u, visited, stack)
7      return stack
8
9  Procedure DFSVisit(G, u, visited, stack)
10     add u to visited
11     for (u, v) ∈ E do
12         if v ∉ visited then
13             DFSVisit(G, v, visited, stack)
14     stack.push(u)
```

C
D
H
G
F
stack

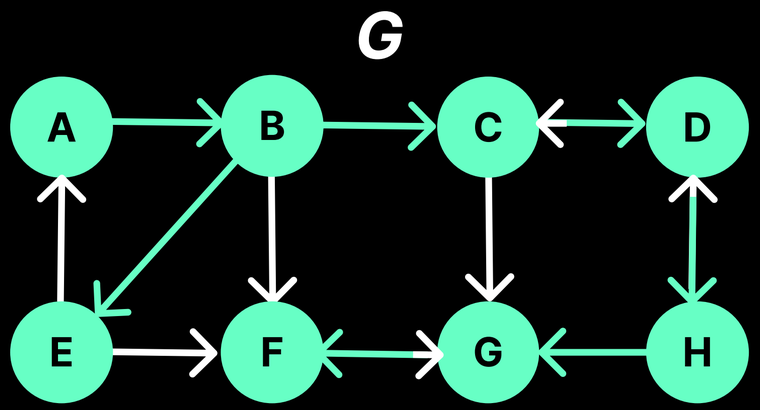
ALGORITHM: FINN DE STERKT SAMMENHENGENDE KOMPONENETENE AV EN GRAF

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
```

Kosaraju

1. DFSTopSort: DFS, der hver node legges på en stack etter alle noder er besøkt
1. Lager den reverserte grafen
2. DFS på den reverserte grafen. Rekkefølgen er gitt i stacken fra DFSTopSort



DFSTopSort: Gjør DFS først

ALGORITHM: TOPOLOGISK SORTERING VED DFS

Input: En rettet asyklisk graf $G = (V, E)$
Output: En topologisk ordning av nodene i G

```
1  Procedure DFSTopSort(G)
2      stack ← empty stack
3      visited ← empty set
4      for u ∈ V do
5          if u ∉ visited then
6              DFSVisit(G, u, visited, stack)
7      return stack
8
9  Procedure DFSVisit(G, u, visited, stack)
10     add u to visited
11     for (u, v) ∈ E do
12         if v ∉ visited then
13             DFSVisit(G, v, visited, stack)
14     stack.push(u)
```

E
C
D
H
G
F
stack

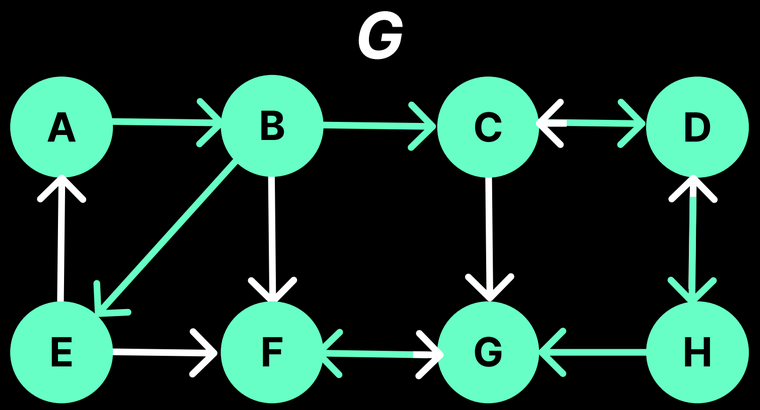
ALGORITHM: FINN DE STERKT SAMMENHENGENDE KOMPONENETENE AV EN GRAF

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
```

Kosaraju

1. DFSTopSort: DFS, der hver node legges på en stack etter alle noder er besøkt
1. Lager den reverserte grafen
2. DFS på den reverserte grafen. Rekkefølgen er gitt i stacken fra DFSTopSort



DFSTopSort: Gjør DFS først

B
E
C
D
H
G
F
stack

ALGORITHM: TOPOLOGISK SORTERING VED DFS

Input: En rettet asyklisk graf $G = (V, E)$
Output: En topologisk ordning av nodene i G

```
1  Procedure DFSTopSort(G)
2      stack ← empty stack
3      visited ← empty set
4      for u ∈ V do
5          if u ∉ visited then
6              DFSVisit(G, u, visited, stack)
7      return stack
8
9  Procedure DFSVisit(G, u, visited, stack)
10     add u to visited
11     for (u, v) ∈ E do
12         if v ∉ visited then
13             DFSVisit(G, v, visited, stack)
14     stack.push(u)
```

ALGORITHM: FINN DE STERKT SAMMENHENGENDE KOMPONENETENE AV EN GRAF

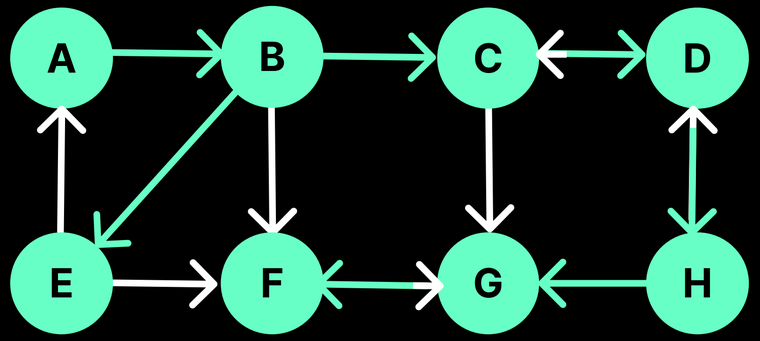
Input: En rettet graf $G = (V, E)$
Output: Returner de sterkt sammenhengende komponenetene til G

```
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
```

Kosaraju

1. DFSTopSort: DFS, der hver node legges på en stack etter alle noder er besøkt
1. Lager den reverserte grafen
2. DFS på den reverserte grafen. Rekkefølgen er gitt i stacken fra DFSTopSort

G



DFSTopSort: Gjør DFS først

A
B
E
C
D
H
G
F
stack

ALGORITHM: TOPOLOGISK SORTERING VED DFS

Input: En rettet asyklisk graf $G = (V, E)$
Output: En topologisk ordning av nodene i G

```
1  Procedure DFSTopSort(G)
2      stack ← empty stack
3      visited ← empty set
4      for u ∈ V do
5          if u ∉ visited then
6              DFSVisit(G, u, visited, stack)
7      return stack
8
9  Procedure DFSVisit(G, u, visited, stack)
10     add u to visited
11     for (u, v) ∈ E do
12         if v ∉ visited then
13             DFSVisit(G, v, visited, stack)
14     stack.push(u)
```

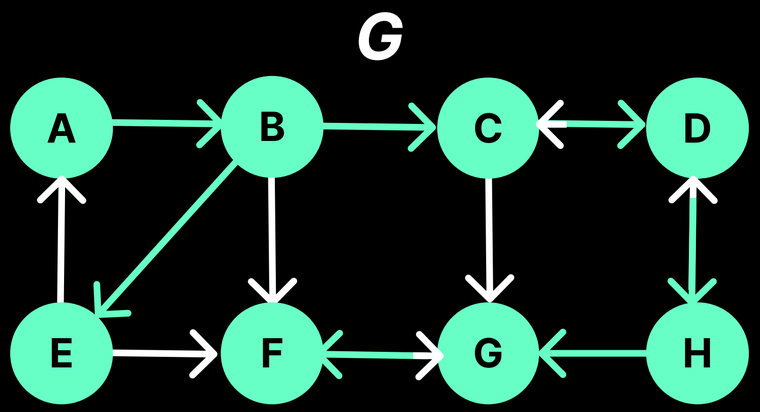
ALGORITHM: FINN DE STERKT SAMMENHENGENDE KOMPONENETENE AV EN GRAF

Input: En rettet graf $G = (V, E)$
Output: Returner de sterkt sammenhengende komponenetene til G

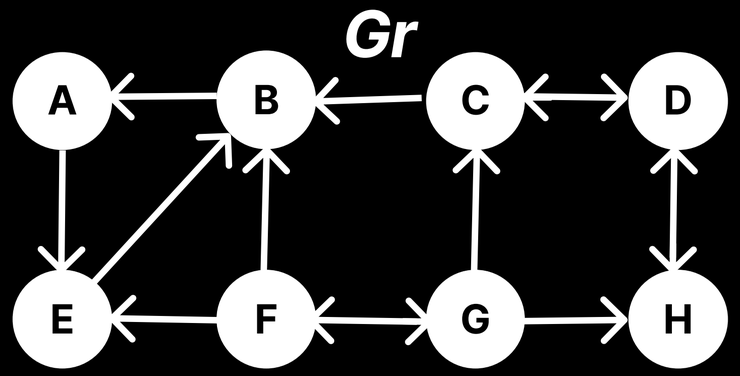
```
1  Procedure StronglyConnectedComponents(G)
2      stack ← DFSTopSort(G)
3      G_r ← ReverseGraph(G)
4      visited ← empty set
5      components ← empty set
6      while stack is not empty do
7          u ← stack.pop()
8          if u ∉ visited then
9              component ← empty set
10     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

Kosaraju

1. DFSTopSort: DFS, der hver node legges på en stack etter alle noder er besøkt
2. Lager den reverserte grafen
3. DFS på den reverserte grafen. Rekkefølgen er gitt i stacken fra DFSTopSort



Lager den reverserte grafen



A
B
E
C
D
H
G
F

stack

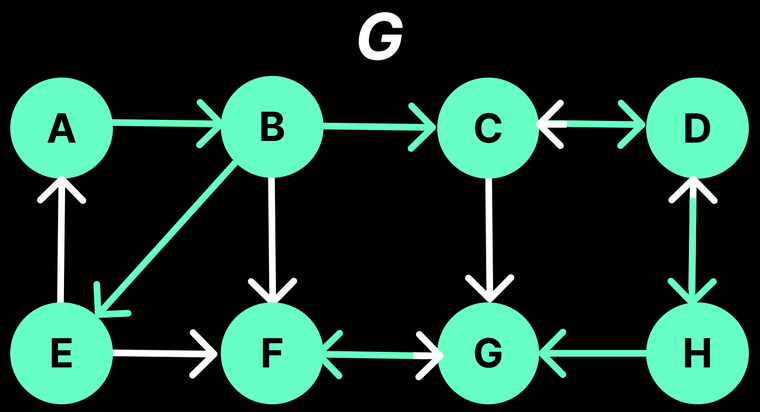
ALGORITHM: FINN DE STERKT SAMMENHENGENDE KOMPONENETENE AV EN GRAF

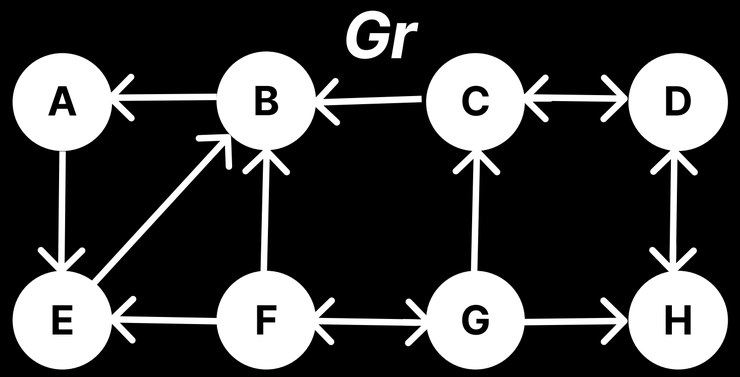
Input: En rettet graf $G = (V, E)$
Output: Returner de sterkt sammenhengende komponenetene til G

```
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
```

Kosaraju

1. DFSTopSort: DFS, der hver node legges på en stack etter alle noder er besøkt
2. Lager den reverserte grafen
3. DFS på den reverserte grafen. Rekkefølgen er gitt i stacken fra DFSTopSort





DFS på Gr, i rekkefølgen fra stacken
Så vi starter med DFS fra A, denne poppes fra stacken

A
B
E
C
D
H
G
F

stack

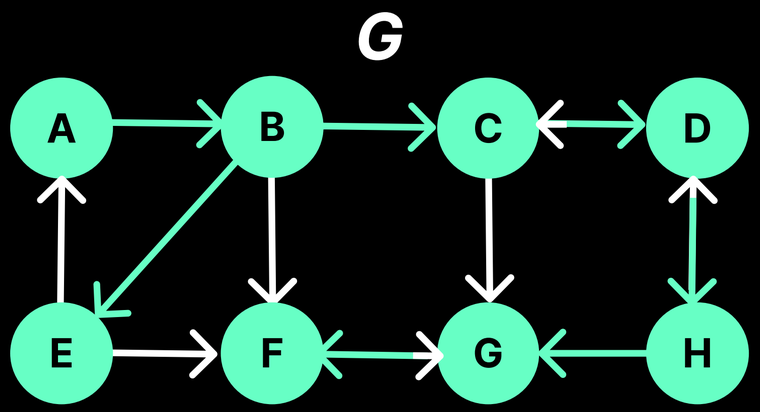
ALGORITHM: FINN DE STERKT SAMMENHENGENDE KOMPONENETENE AV EN GRAF

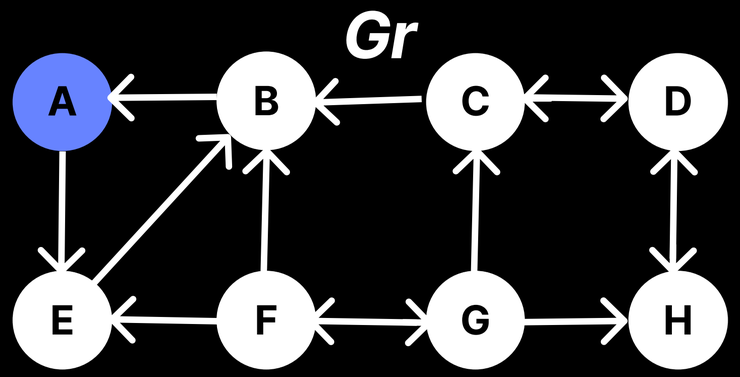
Input: En rettet graf $G = (V, E)$
Output: Returner de sterkt sammenhengende komponenetene til G

```
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
```

Kosaraju

1. DFSTopSort: DFS, der hver node legges på en stack etter alle noder er besøkt
2. Lager den reverserte grafen
3. DFS på den reverserte grafen. Rekkefølgen er gitt i stacken fra DFSTopSort





DFS på Gr, i rekkefølgen fra stacken
Så vi starter med DFS fra A, denne poppes fra stacken

B
E
C
D
H
G
F

stack

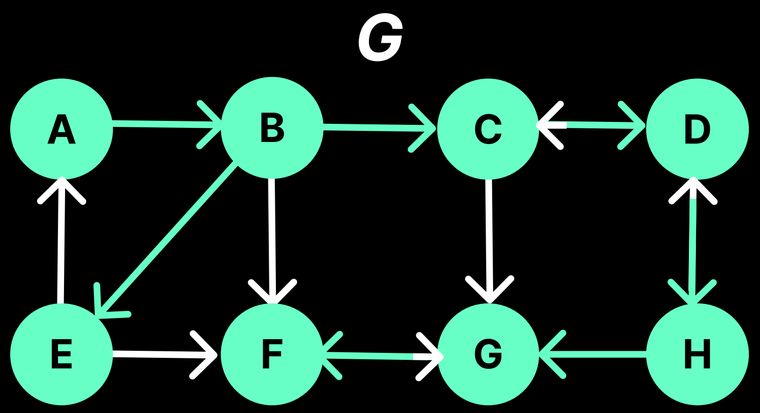
ALGORITHM: FINN DE STERKT SAMMENHENGENDE KOMPONENETENE AV EN GRAF

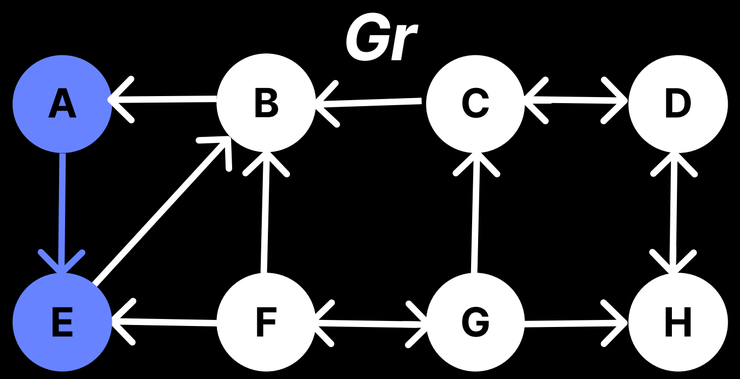
Input: En rettet graf $G = (V, E)$
Output: Returner de sterkt sammenhengende komponenetene til G

```
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
```

Kosaraju

1. DFSTopSort: DFS, der hver node legges på en stack etter alle noder er besøkt
2. Lager den reverserte grafen
3. DFS på den reverserte grafen. Rekkefølgen er gitt i stacken fra DFSTopSort





DFS på Gr, i rekkefølgen fra stacken
Så vi starter med DFS fra A, denne poppes fra stacken

B
E
C
D
H
G
F

stack

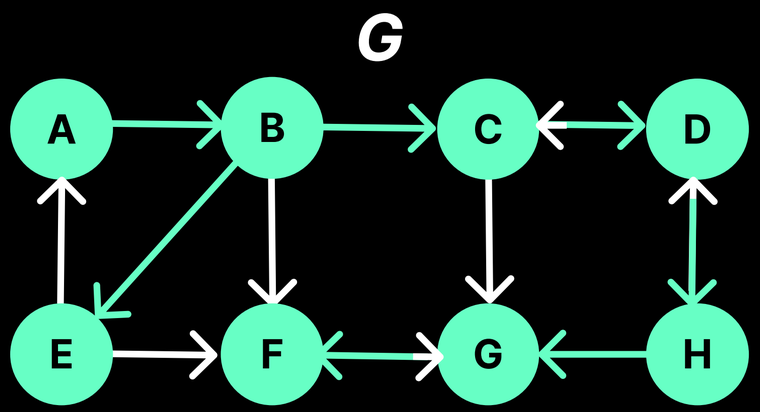
ALGORITHM: FINN DE STERKT SAMMENHENGENDE KOMPONENETENE AV EN GRAF

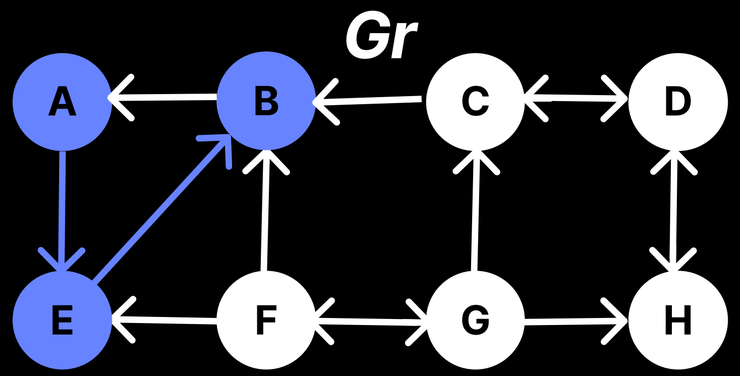
Input: En rettet graf $G = (V, E)$
Output: Returner de sterkt sammenhengende komponenetene til G

```
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
```

Kosaraju

1. DFSTopSort: DFS, der hver node legges på en stack etter alle noder er besøkt
2. Lager den reverserte grafen
3. DFS på den reverserte grafen. Rekkefølgen er gitt i stacken fra DFSTopSort





DFS på Gr, i rekkefølgen fra stacken
Så vi starter med DFS fra A, denne poppes fra stacken

B
E
C
D
H
G
F

stack

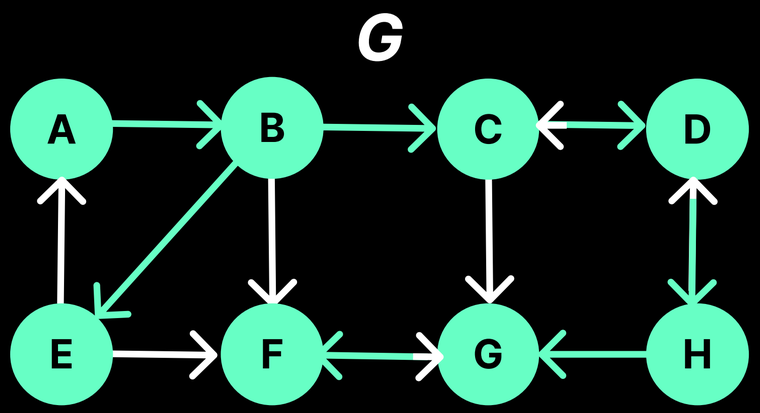
ALGORITHM: FINN DE STERKT SAMMENHENGENDE KOMPONENETENE AV EN GRAF

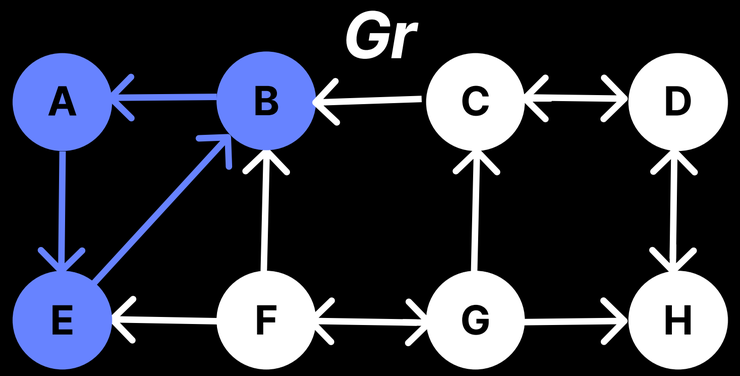
Input: En rettet graf $G = (V, E)$
Output: Returner de sterkt sammenhengende komponenetene til G

```
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
```

Kosaraju

1. DFSTopSort: DFS, der hver node legges på en stack etter alle noder er besøkt
2. Lager den reverserte grafen
3. DFS på den reverserte grafen. Rekkefølgen er gitt i stacken fra DFSTopSort





DFS på Gr, i rekkefølgen fra stacken
Vi har nå vår første sterkt sammenhengende komponent!

B
E
C
D
H
G
F
stack

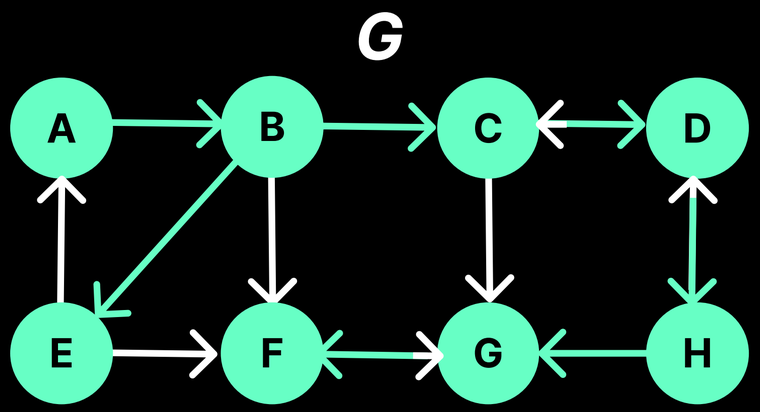
ALGORITHM: FINN DE STERKT SAMMENHENGENDE KOMPONENETENE AV EN GRAF

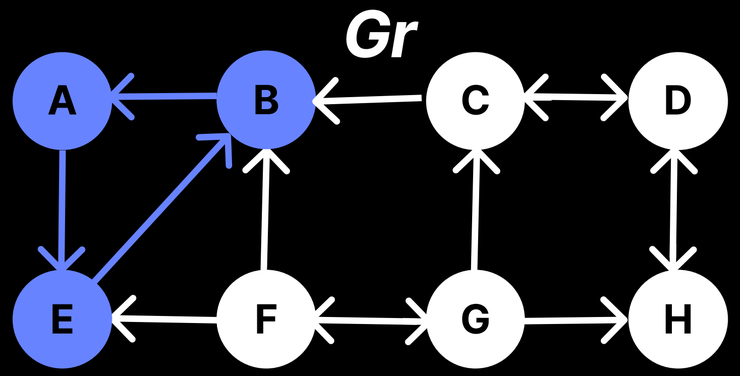
Input: En rettet graf $G = (V, E)$
Output: Returner de sterkt sammenhengende komponenetene til G

```
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
```

Kosaraju

1. DFSTopSort: DFS, der hver node legges på en stack etter alle noder er besøkt
2. Lager den reverserte grafen
3. DFS på den reverserte grafen. Rekkefølgen er gitt i stacken fra DFSTopSort





DFS på Gr, i rekkefølgen fra stacken

B
E
C
D
H
G
F

stack

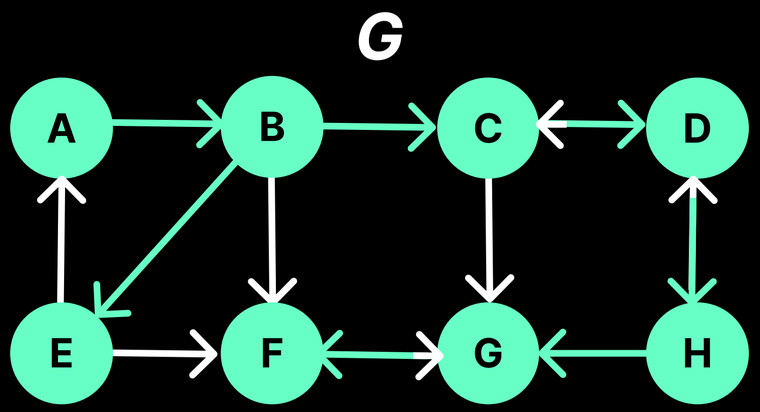
ALGORITHM: FINN DE STERKT SAMMENHENGENDE KOMPONENETENE AV EN GRAF

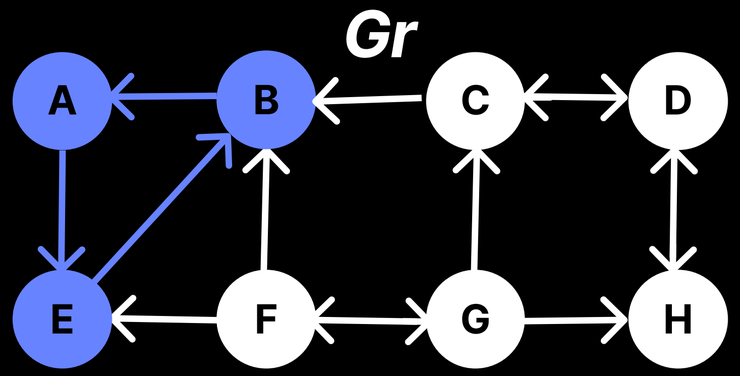
Input: En rettet graf $G = (V, E)$
Output: Returner de sterkt sammenhengende komponenetene til G

```
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
```

Kosaraju

1. DFSTopSort: DFS, der hver node legges på en stack etter alle noder er besøkt
2. Lager den reverserte grafen
3. DFS på den reverserte grafen. Rekkefølgen er gitt i stacken fra DFSTopSort





DFS på Gr, i rekkefølgen fra stacken

E
C
D
H
G
F

stack

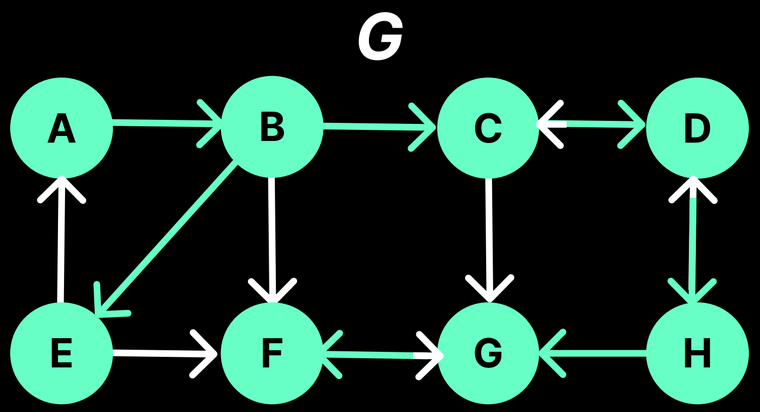
ALGORITHM: FINN DE STERKT SAMMENHENGENDE KOMPONENETENE AV EN GRAF

Input: En rettet graf $G = (V, E)$
Output: Returner de sterkt sammenhengende komponenetene til G

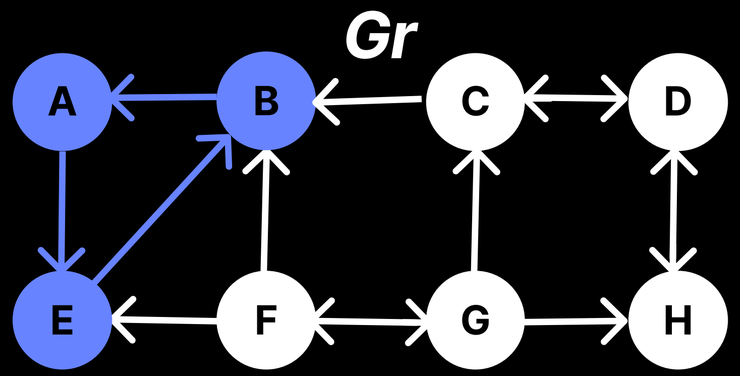
```
1  Procedure StronglyConnectedComponents(G)
2      stack ← DFSTopSort(G)
3      G_r ← ReverseGraph(G)
4      visited ← empty set
5      components ← empty set
6      while stack is not empty do
7          u ← stack.pop()
8          if u ∉ visited then
9              component ← empty set
10     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

Kosaraju

1. DFSTopSort: DFS, der hver node legges på en stack etter alle noder er besøkt
2. Lager den reverserte grafen
3. DFS på den reverserte grafen. Rekkefølgen er gitt i stacken fra DFSTopSort



DFS på Gr, i rekkefølgen fra stacken



C
D
H
G
F
stack

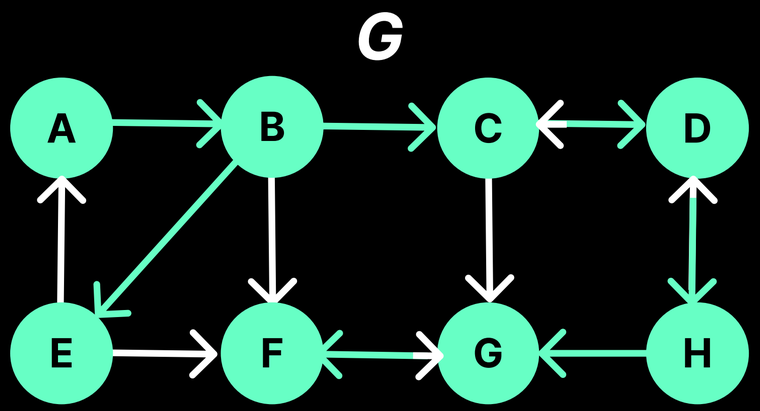
ALGORITHM: FINN DE STERKT SAMMENHENGENDE KOMPONENETENE AV EN GRAF

Input: En rettet graf $G = (V, E)$
Output: Returner de sterkt sammenhengende komponenetene til G

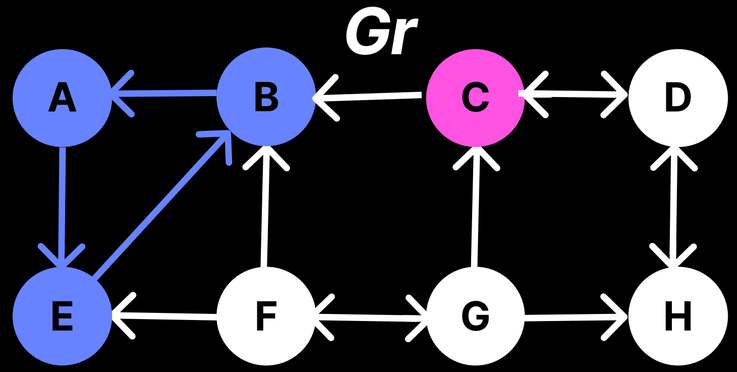
```
1  Procedure StronglyConnectedComponents(G)
2      stack ← DFSTopSort(G)
3      G_r ← ReverseGraph(G)
4      visited ← empty set
5      components ← empty set
6      while stack is not empty do
7          u ← stack.pop()
8          if u ∉ visited then
9              component ← empty set
10     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

Kosaraju

1. DFSTopSort: DFS, der hver node legges på en stack etter alle noder er besøkt
2. Lager den reverserte grafen
3. DFS på den reverserte grafen. Rekkefølgen er gitt i stacken fra DFSTopSort



DFS på Gr, i rekkefølgen fra stacken



D
H
G
F
stack

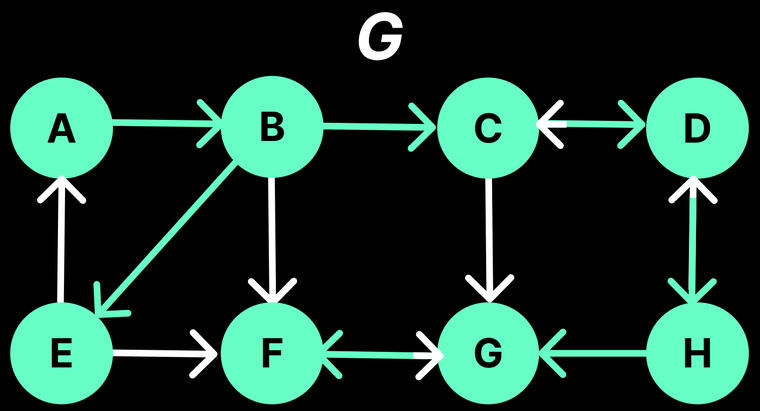
ALGORITHM: FINN DE STERKT SAMMENHENGENDE KOMPONENETENE AV EN GRAF

Input: En rettet graf $G = (V, E)$
Output: Returner de sterkt sammenhengende komponenetene til G

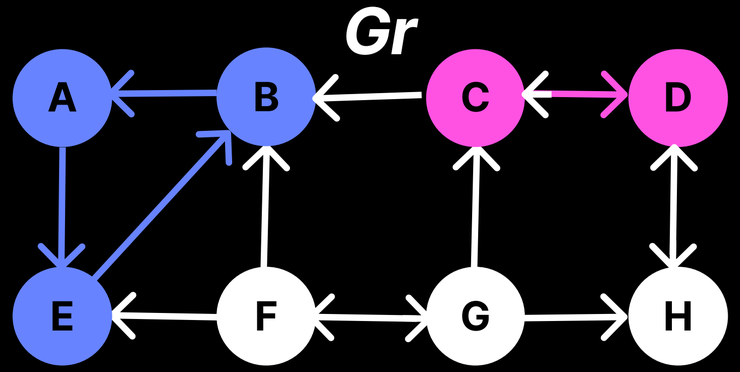
```
1  Procedure StronglyConnectedComponents(G)
2      stack ← DFSTopSort(G)
3      G_r ← ReverseGraph(G)
4      visited ← empty set
5      components ← empty set
6      while stack is not empty do
7          u ← stack.pop()
8          if u ∉ visited then
9              component ← empty set
10     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

Kosaraju

1. DFSTopSort: DFS, der hver node legges på en stack etter alle noder er besøkt
2. Lager den reverserte grafen
3. DFS på den reverserte grafen. Rekkefølgen er gitt i stacken fra DFSTopSort



DFS på Gr, i rekkefølgen fra stacken



D
H
G
F
stack

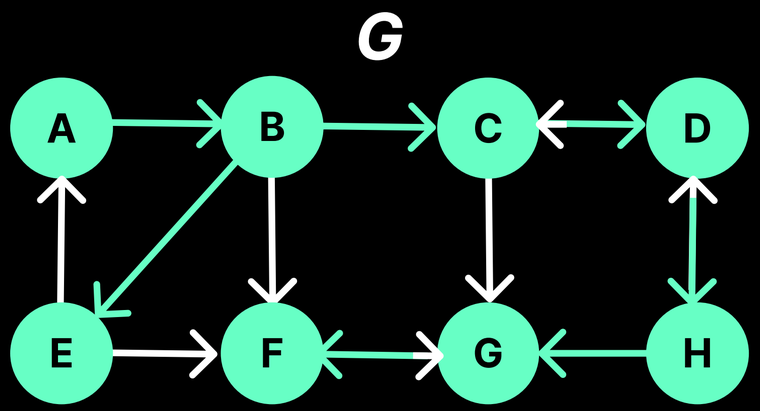
ALGORITHM: FINN DE STERKT SAMMENHENGENDE KOMPONENETENE AV EN GRAF

Input: En rettet graf $G = (V, E)$
Output: Returner de sterkt sammenhengende komponenetene til G

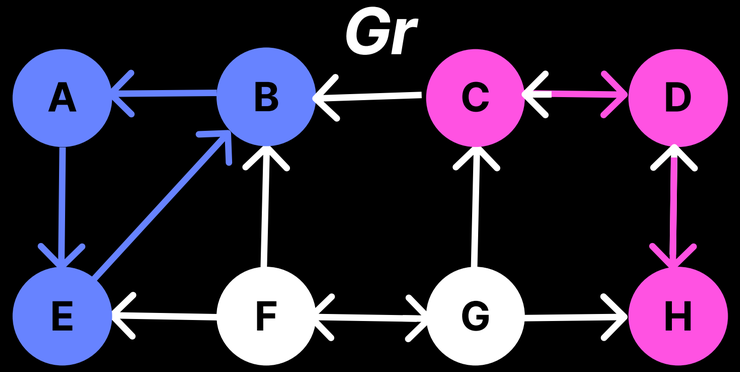
```
1  Procedure StronglyConnectedComponents(G)
2      stack ← DFSTopSort(G)
3      G_r ← ReverseGraph(G)
4      visited ← empty set
5      components ← empty set
6      while stack is not empty do
7          u ← stack.pop()
8          if u ∉ visited then
9              component ← empty set
10     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

Kosaraju

1. DFSTopSort: DFS, der hver node legges på en stack etter alle noder er besøkt
2. Lager den reverserte grafen
3. DFS på den reverserte grafen. Rekkefølgen er gitt i stacken fra DFSTopSort



DFS på Gr, i rekkefølgen fra stacken



D
H
G
F
stack

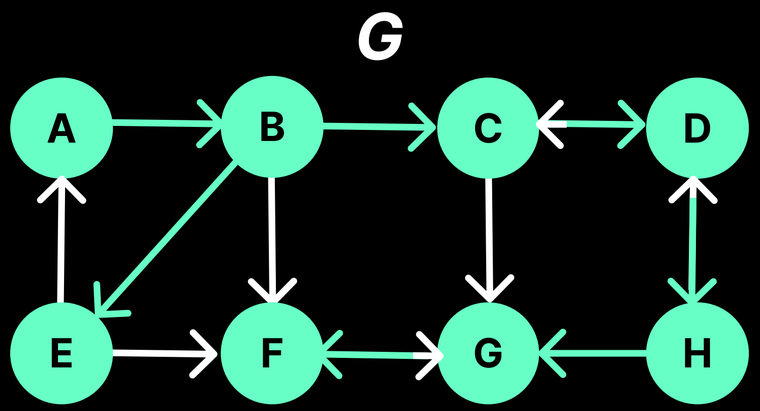
ALGORITHM: FINN DE STERKT SAMMENHENGENDE KOMPONENETENE AV EN GRAF

Input: En rettet graf $G = (V, E)$
Output: Returner de sterkt sammenhengende komponenetene til G

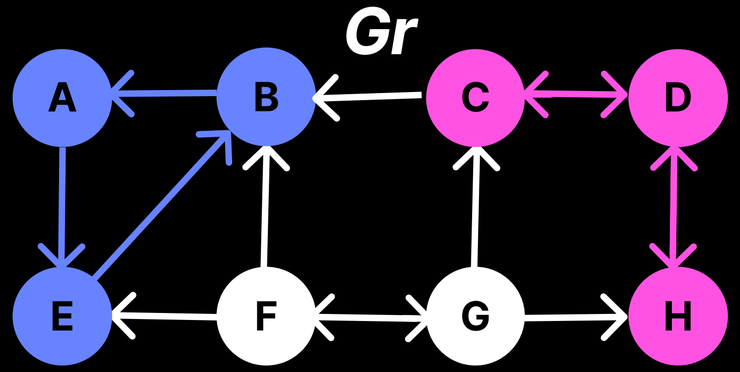
```
1  Procedure StronglyConnectedComponents(G)
2      stack ← DFSTopSort(G)
3      G_r ← ReverseGraph(G)
4      visited ← empty set
5      components ← empty set
6      while stack is not empty do
7          u ← stack.pop()
8          if u ∉ visited then
9              component ← empty set
10     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

Kosaraju

1. DFSTopSort: DFS, der hver node legges på en stack etter alle noder er besøkt
2. Lager den reverserte grafen
3. DFS på den reverserte grafen. Rekkefølgen er gitt i stacken fra DFSTopSort



DFS på Gr, i rekkefølgen fra stacken



D
H
G
F
stack

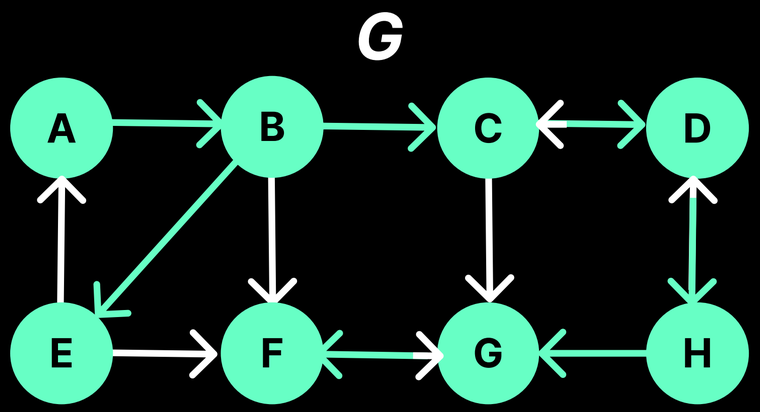
ALGORITHM: FINN DE STERKT SAMMENHENGENDE KOMPONENETENE AV EN GRAF

Input: En rettet graf $G = (V, E)$
Output: Returner de sterkt sammenhengende komponenetene til G

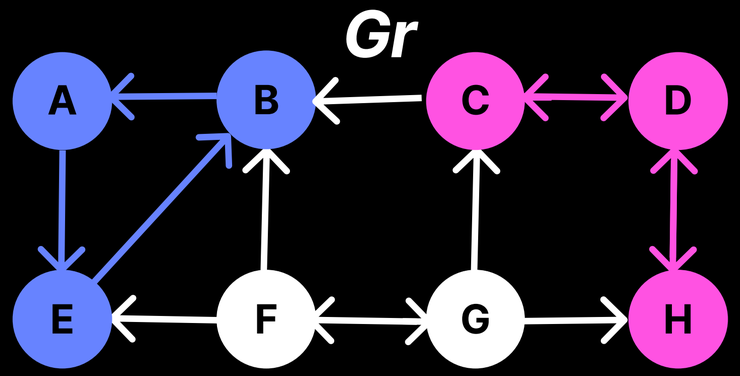
```
1  Procedure StronglyConnectedComponents(G)
2      stack ← DFSTopSort(G)
3      G_r ← ReverseGraph(G)
4      visited ← empty set
5      components ← empty set
6      while stack is not empty do
7          u ← stack.pop()
8          if u ∉ visited then
9              component ← empty set
10     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

Kosaraju

1. DFSTopSort: DFS, der hver node legges på en stack etter alle noder er besøkt
2. Lager den reverserte grafen
3. DFS på den reverserte grafen. Rekkefølgen er gitt i stacken fra DFSTopSort



DFS på Gr, i rekkefølgen fra stacken



H
G
F
stack

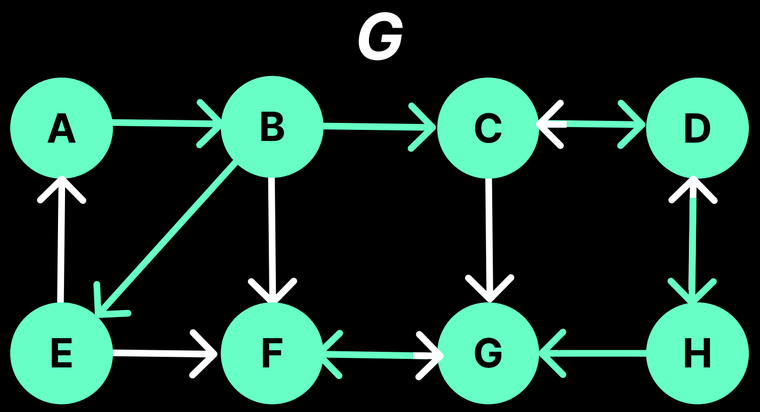
ALGORITHM: FINN DE STERKT SAMMENHENGENDE KOMPONENETENE AV EN GRAF

Input: En rettet graf $G = (V, E)$
Output: Returner de sterkt sammenhengende komponenetene til G

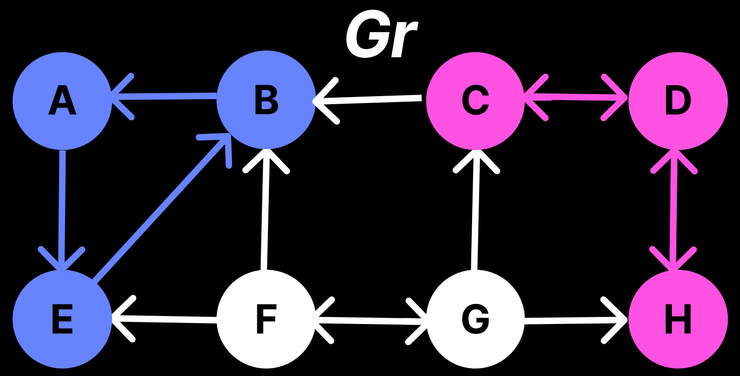
```
1  Procedure StronglyConnectedComponents(G)
2      stack ← DFSTopSort(G)
3      G_r ← ReverseGraph(G)
4      visited ← empty set
5      components ← empty set
6      while stack is not empty do
7          u ← stack.pop()
8          if u ∉ visited then
9              component ← empty set
10     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

Kosaraju

1. DFSTopSort: DFS, der hver node legges på en stack etter alle noder er besøkt
2. Lager den reverserte grafen
3. DFS på den reverserte grafen. Rekkefølgen er gitt i stacken fra DFSTopSort



DFS på Gr, i rekkefølgen fra stacken



G
F
stack

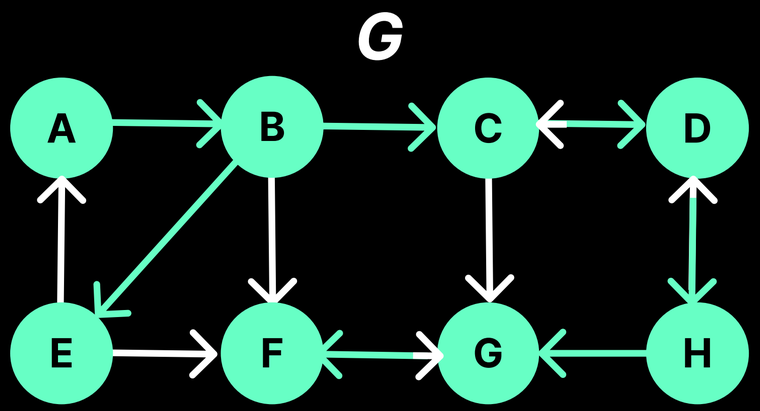
ALGORITHM: FINN DE STERKT SAMMENHENGENDE KOMPONENETENE AV EN GRAF

Input: En rettet graf $G = (V, E)$
Output: Returner de sterkt sammenhengende komponenetene til G

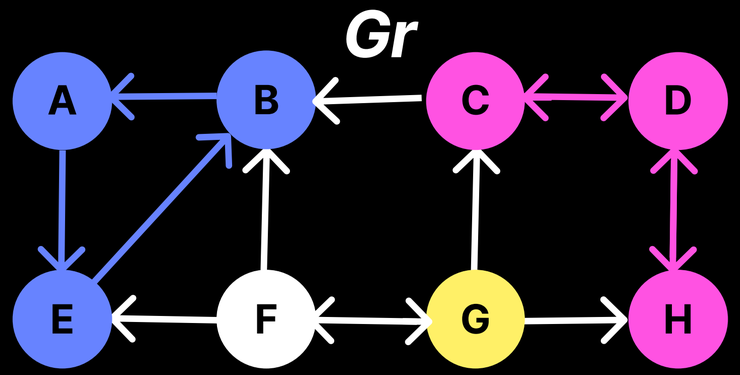
```
1  Procedure StronglyConnectedComponents(G)
2      stack ← DFSTopSort(G)
3      G_r ← ReverseGraph(G)
4      visited ← empty set
5      components ← empty set
6      while stack is not empty do
7          u ← stack.pop()
8          if u ∉ visited then
9              component ← empty set
10     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

Kosaraju

1. DFSTopSort: DFS, der hver node legges på en stack etter alle noder er besøkt
2. Lager den reverserte grafen
3. DFS på den reverserte grafen. Rekkefølgen er gitt i stacken fra DFSTopSort



DFS på Gr, i rekkefølgen fra stacken



F
stack

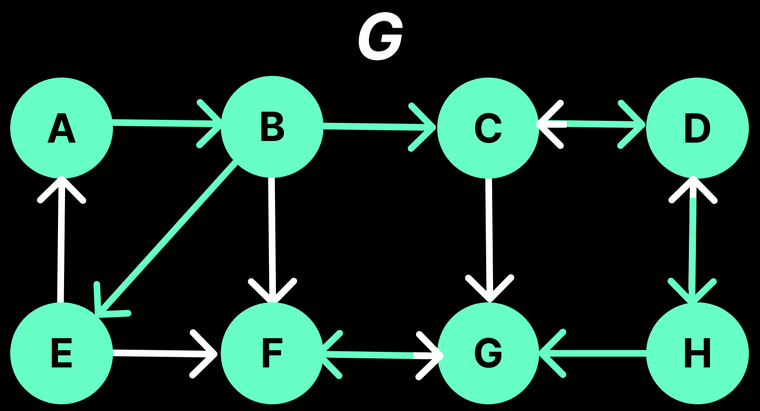
ALGORITHM: FINN DE STERKT SAMMENHENGENDE KOMPONENETENE AV EN GRAF

Input: En rettet graf $G = (V, E)$
Output: Returner de sterkt sammenhengende komponenetene til G

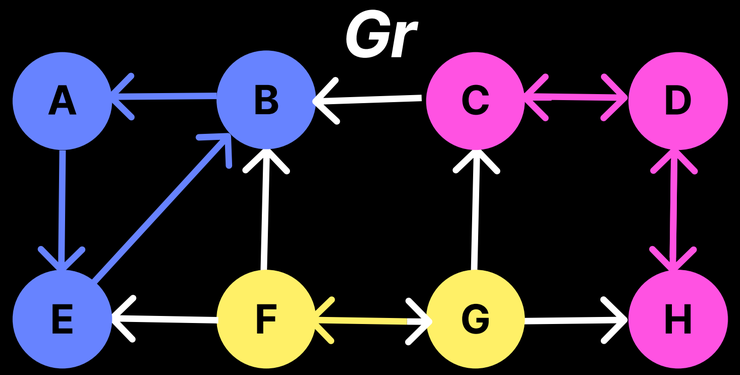
```
1  Procedure StronglyConnectedComponents(G)
2      stack ← DFSTopSort(G)
3      G_r ← ReverseGraph(G)
4      visited ← empty set
5      components ← empty set
6      while stack is not empty do
7          u ← stack.pop()
8          if u ∉ visited then
9              component ← empty set
10     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

Kosaraju

1. DFSTopSort: DFS, der hver node legges på en stack etter alle noder er besøkt
2. Lager den reverserte grafen
3. DFS på den reverserte grafen. Rekkefølgen er gitt i stacken fra DFSTopSort



DFS på Gr, i rekkefølgen fra stacken



F
stack

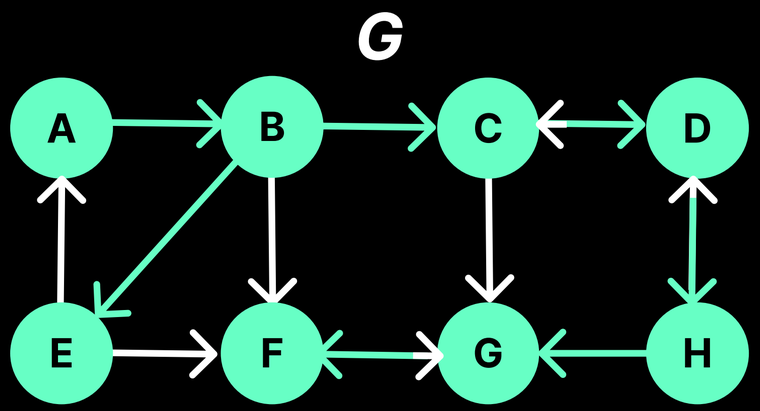
ALGORITHM: FINN DE STERKT SAMMENHENGENDE KOMPONENETENE AV EN GRAF

Input: En rettet graf $G = (V, E)$
Output: Returner de sterkt sammenhengende komponenetene til G

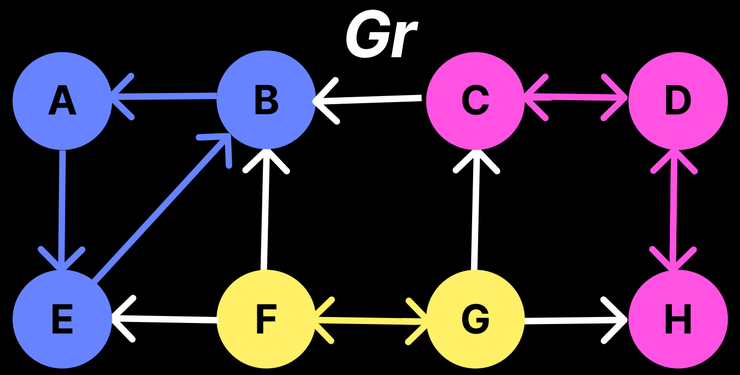
```
1  Procedure StronglyConnectedComponents(G)
2      stack ← DFSTopSort(G)
3      G_r ← ReverseGraph(G)
4      visited ← empty set
5      components ← empty set
6      while stack is not empty do
7          u ← stack.pop()
8          if u ∉ visited then
9              component ← empty set
10     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

Kosaraju

1. DFSTopSort: DFS, der hver node legges på en stack etter alle noder er besøkt
2. Lager den reverserte grafen
3. DFS på den reverserte grafen. Rekkefølgen er gitt i stacken fra DFSTopSort



DFS på Gr, i rekkefølgen fra stacken



F
stack

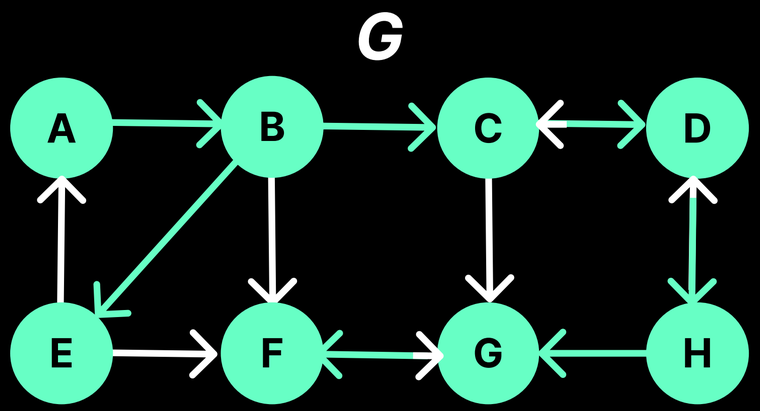
ALGORITHM: FINN DE STERKT SAMMENHENGENDE KOMPONENETENE AV EN GRAF

Input: En rettet graf $G = (V, E)$
Output: Returner de sterkt sammenhengende komponenetene til G

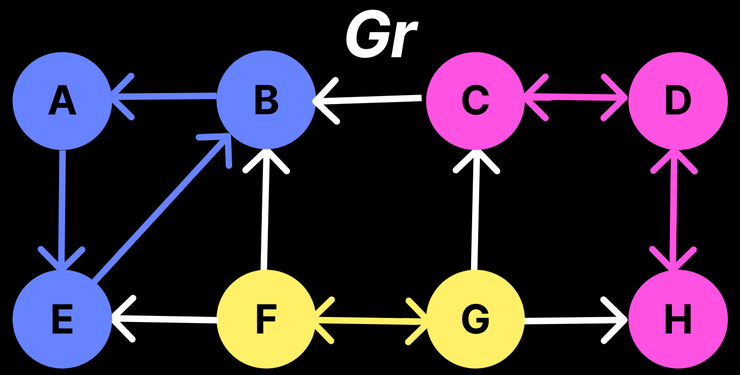
```
1  Procedure StronglyConnectedComponents(G)
2      stack ← DFSTopSort(G)
3      G_r ← ReverseGraph(G)
4      visited ← empty set
5      components ← empty set
6      while stack is not empty do
7          u ← stack.pop()
8          if u ∉ visited then
9              component ← empty set
10     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

Kosaraju

1. DFSTopSort: DFS, der hver node legges på en stack etter alle noder er besøkt
2. Lager den reverserte grafen
3. DFS på den reverserte grafen. Rekkefølgen er gitt i stacken fra DFSTopSort



DFS på Gr, i rekkefølgen fra stacken



stack

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
```