Grafer, traversering og topologisk sortering

IN2010 – Algoritmer og Datastrukturer

Lars Tveito

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

Høsten 2023

Oversikt

Oversikt

- I de neste tre forelesningene skal vi lære om grafer!

Oversikt

- I de neste tre forelesningene skal vi lære om grafer!
- Grafer lar oss representere «hvordan ting henger sammen»

Oversikt

- I de neste tre forelesningene skal vi lære om grafer!
- Grafer lar oss representere «hvordan ting henger sammen»
  - De er utrolig generelle og anvendelige

Oversikt

- I de neste tre forelesningene skal vi lære om grafer!
- Grafer lar oss representere «hvordan ting henger sammen»
  - De er utrolig generelle og anvendelige
- Denne uken dekker vi

Oversikt

- I de neste tre forelesningene skal vi lære om grafer!
- Grafer lar oss representere «hvordan ting henger sammen»
  - De er utrolig generelle og anvendelige
- Denne uken dekker vi
  - Hva grafer *er*

Oversikt

- I de neste tre forelesningene skal vi lære om grafer!
- Grafer lar oss representere «hvordan ting henger sammen»
  - De er utrolig generelle og anvendelige
- Denne uken dekker vi
  - Hva grafer *er*
  - Hvordan de *representeres*

Oversikt

- I de neste tre forelesningene skal vi lære om grafer!
- Grafer lar oss representere «hvordan ting henger sammen»
  - De er utrolig generelle og anvendelige
- Denne uken dekker vi
  - Hva grafer *er*
  - Hvordan de *representeres*
  - Hvordan de *traverseres*

Oversikt

- I de neste tre forelesningene skal vi lære om grafer!
- Grafer lar oss representere «hvordan ting henger sammen»
  - De er utrolig generelle og anvendelige
- Denne uken dekker vi
  - Hva grafer *er*
  - Hvordan de *representeres*
  - Hvordan de *traverseres*
  - Hvordan de kan ordnes

Oversikt

- I de neste tre forelesningene skal vi lære om grafer!
- Grafer lar oss representere «hvordan ting henger sammen»
  - De er utrolig generelle og anvendelige
- Denne uken dekker vi
  - Hva grafer *er*
  - Hvordan de *representeres*
  - Hvordan de *traverseres*
  - Hvordan de kan ordnes
- I neste forelesning handler det om grafer *med vekter*

Oversikt

- I de neste tre forelesningene skal vi lære om grafer!
- Grafer lar oss representere «hvordan ting henger sammen»
  - De er utrolig generelle og anvendelige
- Denne uken dekker vi
  - Hva grafer *er*
  - Hvordan de *representeres*
  - Hvordan de *traverseres*
  - Hvordan de kan ordnes
- I neste forelesning handler det om grafer *med vekter*
  - Hvordan finne korteste stier i en graf

Oversikt

- I de neste tre forelesningene skal vi lære om grafer!
- Grafer lar oss representere «hvordan ting henger sammen»
  - De er utrolig generelle og anvendelige
- Denne uken dekker vi
  - Hva grafer *er*
  - Hvordan de *representeres*
  - Hvordan de *traverseres*
  - Hvordan de kan ordnes
- I neste forelesning handler det om grafer *med vekter*
  - Hvordan finne korteste stier i en graf
  - Hvordan finne minimale spenntrær

Oversikt

- I de neste tre forelesningene skal vi lære om grafer!
- Grafer lar oss representere «hvordan ting henger sammen»
  - De er utrolig generelle og anvendelige
- Denne uken dekker vi
  - Hva grafer *er*
  - Hvordan de *representeres*
  - Hvordan de *traverseres*
  - Hvordan de kan ordnes
- I neste forelesning handler det om grafer *med vekter*
  - Hvordan finne korteste stier i en graf
  - Hvordan finne minimale spenntrær
- I forelesningen etter dekker vi

Oversikt

- I de neste tre forelesningene skal vi lære om grafer!
- Grafer lar oss representere «hvordan ting henger sammen»
  - De er utrolig generelle og anvendelige
- Denne uken dekker vi
  - Hva grafer *er*
  - Hvordan de *representeres*
  - Hvordan de *traverseres*
  - Hvordan de kan ordnes
- I neste forelesning handler det om grafer *med vekter*
  - Hvordan finne korteste stier i en graf
  - Hvordan finne minimale spenntrær
- I forelesningen etter dekker vi
  - Bikonnektivitet og separasjonsnoder

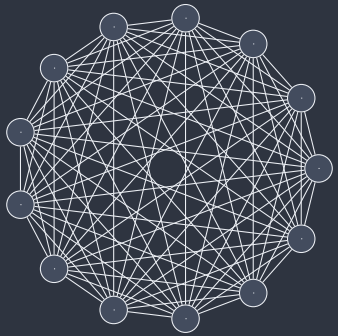
Oversikt

- I de neste tre forelesningene skal vi lære om grafer!
- Grafer lar oss representere «hvordan ting henger sammen»
  - De er utrolig generelle og anvendelige
- Denne uken dekker vi
  - Hva grafer *er*
  - Hvordan de *representeres*
  - Hvordan de *traverseres*
  - Hvordan de kan ordnes
- I neste forelesning handler det om grafer *med vekter*
  - Hvordan finne korteste stier i en graf
  - Hvordan finne minimale spenntrær
- I forelesningen etter dekker vi
  - Bikonnektivitet og separasjonsnoder
  - Sterkt sammenhengende komponenter

Definisjoner og terminologi

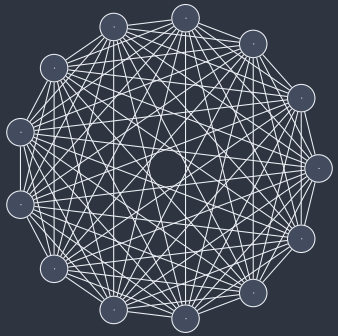
Eksempler på grafer

- Kart



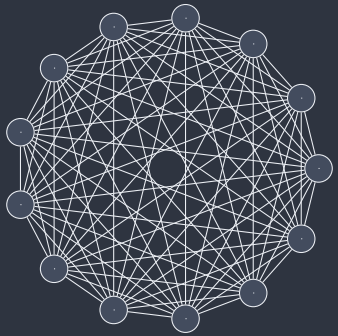
Eksempler på grafer

- Kart
- Bekjentskapsgrafer



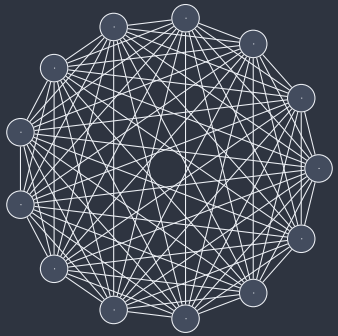
Eksempler på grafer

- Kart
- Bekjentskapsgrafer
- Nettverk



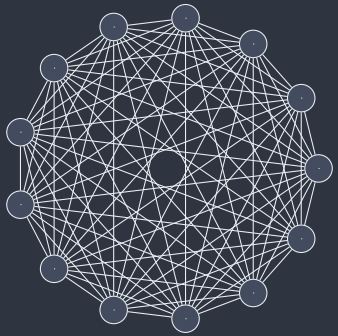
Eksempler på grafer

- Kart
- Bekjentskapsgrafer
- Nettverk
- Tilstander

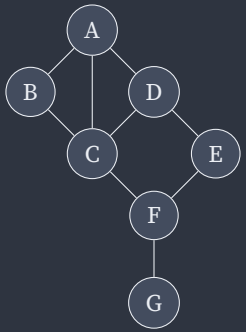


Eksempler på grafer

- Kart
- Bekjentskapsgrafer
- Nettverk
- Tilstander
- Trær

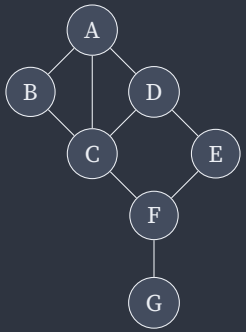


Grafer



- En mengde med noder: $\{A, B, C, D, E, F, G\}$

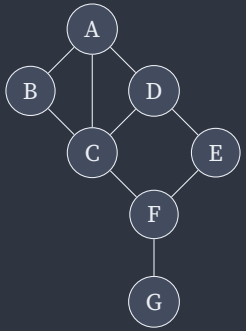
Grafer



- En mengde med noder: $\{A, B, C, D, E, F, G\}$
- En mengde med kanter:

$$\{\{A, B\}, \{A, C\}, \{A, D\},$$
$$\{B, C\}, \{C, D\}, \{C, F\},$$
$$\{D, E\}, \{E, F\}, \{F, G\}\}$$

Grafer

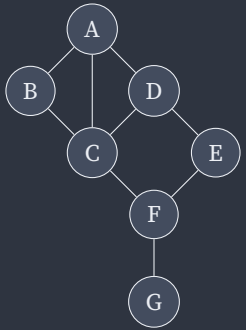


- En mengde med noder: $\{A, B, C, D, E, F, G\}$
- En mengde med kanter:

$$\{\{A, B\}, \{A, C\}, \{A, D\},$$
$$\{B, C\}, \{C, D\}, \{C, F\},$$
$$\{D, E\}, \{E, F\}, \{F, G\}\}$$

- En graf G er definert som to mengder $G = (V, E)$

Grafer

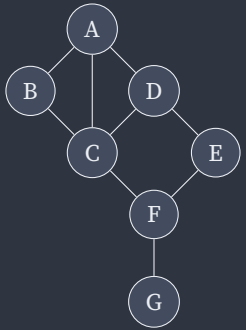


- En mengde med noder: $\{A, B, C, D, E, F, G\}$
- En mengde med kanter:

$$\{\{A, B\}, \{A, C\}, \{A, D\},$$
$$\{B, C\}, \{C, D\}, \{C, F\},$$
$$\{D, E\}, \{E, F\}, \{F, G\}\}$$

- En graf G er definert som to mengder $G = (V, E)$
  - der V er en mengde av noder (eng: vertices) og

Grafer

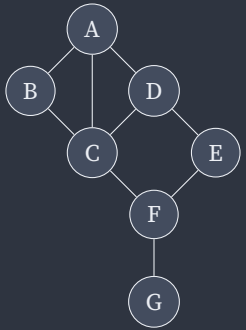


- En mengde med noder: $\{A, B, C, D, E, F, G\}$
- En mengde med kanter:

$$\{\{A, B\}, \{A, C\}, \{A, D\},$$
$$\{B, C\}, \{C, D\}, \{C, F\},$$
$$\{D, E\}, \{E, F\}, \{F, G\}\}$$

- En graf G er definert som to mengder $G = (V, E)$
  - der V er en mengde av noder (eng: vertices) og
  - E er en mengde av kanter (eng: edges) mellom noder fra V

Grafer



- En mengde med noder: $\{A, B, C, D, E, F, G\}$
- En mengde med kanter:

$$\{\{A, B\}, \{A, C\}, \{A, D\},$$
$$\{B, C\}, \{C, D\}, \{C, F\},$$
$$\{D, E\}, \{E, F\}, \{F, G\}\}$$

- En graf G er definert som to mengder $G = (V, E)$
  - der V er en mengde av noder (eng: vertices) og
  - E er en mengde av kanter (eng: edges) mellom noder fra V
- En kant er representert som en mengde som består av to noder $\{u, v\}$

Terminologi

Betegnelse	Forklaring	Eksempel
Parellelle kanter	Flere enn én kant mellom to noder	

Terminologi

Betegnelse	Forklaring	Eksempel
Parellelle kanter	Flere enn én kant mellom to noder	
(Enkle) løkker	En kant fra en node til seg selv	

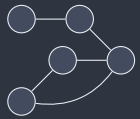
Terminologi

Betegnelse	Forklaring	Eksempel
Parellelle kanter	Flere enn én kant mellom to noder	
(Enkle) løkker	En kant fra en node til seg selv	
Urettet/rettet	Kantene i grafen har retning	

Terminologi

Betegnelse	Forklaring	Eksempel
Parellelle kanter	Flere enn én kant mellom to noder	
(Enkle) løkker	En kant fra en node til seg selv	
Urettet/rettet	Kantene i grafen har retning	
Vektet/uvektet	Kantene har en verdi	

Terminologi

Betegnelse	Forklaring	Eksempel
Parellelle kanter	Flere enn én kant mellom to noder	
(Enkle) løkker	En kant fra en node til seg selv	
Urettet/rettet	Kantene i grafen har retning	
Vektet/uvektet	Kantene har en verdi	
Enkel graf	Uten løkker, parallelle kanter, retning og vekt	

Grafer med og uten retning

- En graf kan ha rettede kanter:  vs

Grafer med og uten retning

- En graf kan ha rettede kanter:  vs
- For *urettede* grafer skriver vi $\{u, v\}$ for en kant mellom u og v

Grafer med og uten retning

- En graf kan ha rettede kanter:  vs
- For *urettede* grafer skriver vi $\{u, v\}$ for en kant mellom u og v
  - En slik kant kan «krysses i begge retninger»

Grafer med og uten retning

- En graf kan ha rettede kanter:  vs
- For *urettede* grafer skriver vi $\{u, v\}$ for en kant mellom u og v
  - En slik kant kan «krysses i begge retninger»
- For *rettede* grafer skriver vi (u, v) for en kant fra u til v

Grafer med og uten retning

- En graf kan ha rettede kanter:  vs
- For *urettede* grafer skriver vi $\{u, v\}$ for en kant mellom u og v
  - En slik kant kan «krysses i begge retninger»
- For *rettede* grafer skriver vi (u, v) for en kant fra u til v
  - En slik kant kan kun følges *fra u til v*, men *ikke fra v til u*

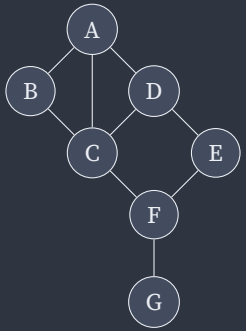
Grafer med og uten retning

- En graf kan ha rettede kanter:  vs
- For *urettede* grafer skriver vi $\{u, v\}$ for en kant mellom u og v
  - En slik kant kan «krysses i begge retninger»
- For *rettede* grafer skriver vi (u, v) for en kant fra u til v
  - En slik kant kan kun følges *fra u til v*, men *ikke fra v til u*
- Urettede grafer kan representeres som rettede grafer

Grafer med og uten retning

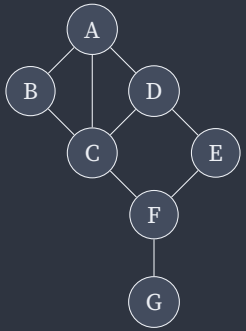
- En graf kan ha rettede kanter:  vs
- For *urettede* grafer skriver vi $\{u, v\}$ for en kant mellom u og v
  - En slik kant kan «krysses i begge retninger»
- For *rettede* grafer skriver vi (u, v) for en kant fra u til v
  - En slik kant kan kun følges *fra u til v*, men *ikke fra v til u*
- Urettede grafer kan representeres som rettede grafer
  - hvor for hver kant $(u, v) \in E$, så finnes det en kant $(v, u) \in E$

Veier og stier



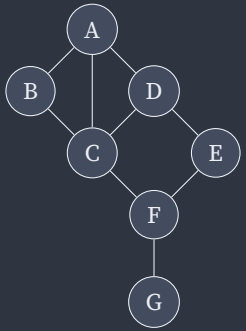
- Sti: En sekvens av noder i grafen, forbundet av kanter, der ingen *noder* gjentas: A, B, C, D

Veier og stier



- Sti: En sekvens av noder i grafen, forbundet av kanter, der ingen *noder* gjentas: A, B, C, D
- Vei: En sekvens av noder i grafen, forbundet av kanter, der ingen *kanter* gjentas: A, B, C, A, D

Veier og stier



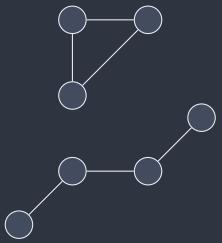
- Sti: En sekvens av noder i grafen, forbundet av kanter, der ingen *noder* gjentas: A, B, C, D
- Vei: En sekvens av noder i grafen, forbundet av kanter, der ingen *kanter* gjentas: A, B, C, A, D
- Ofte ønsker vi å finne den korteste stier mellom noder (tema for neste uke)

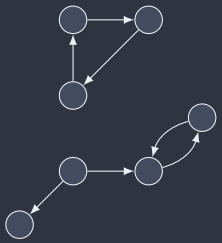
Sammenhengende grafer og komponenter

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

Sammenhengende grafer og komponenter

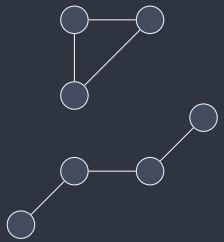
- En graf kalles *sammenhengende* hvis det finnes en sti mellom alle par av noder
- En graf som ikke er sammenhengende kan deles inn i komponenter

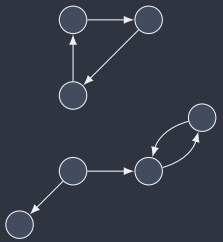




Sammenhengende grafer og komponenter

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

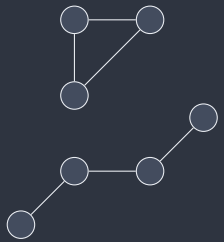


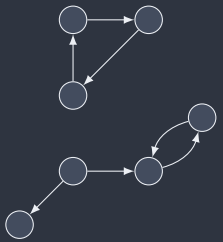


- For *rettede* snakker vi om *sterkt sammenhengende komponenter*

Sammenhengende grafer og komponenter

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

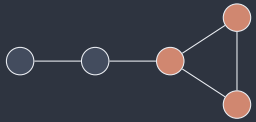




- For *rettede* snakker vi om *sterkt sammenhengende komponenter*
  - Dette dekkes nærmere i en senere forelesning

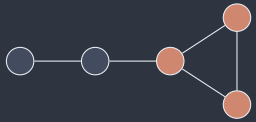
Sykler

- En sti i med minst tre noder som forbinder første og siste node kalles en *sykel*

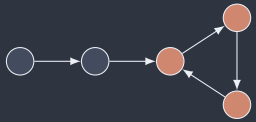


Sykler

- En sti i med minst tre noder som forbinder første og siste node kalles en *sykel*

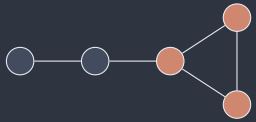


- I rettede grafer må kantene som utgjør en sykel «peke i riktig retning»

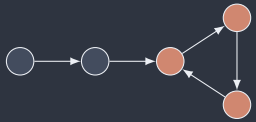


Sykler

- En sti i med minst tre noder som forbinder første og siste node kalles en *sykel*



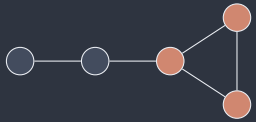
- I rettede grafer må kantene som utgjør en sykel «peke i riktig retning»



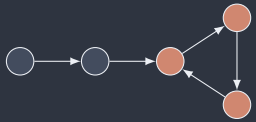
- En graf som ikke inneholder en sykel kalles *asykliske*

Sykler

- En sti i med minst tre noder som forbinder første og siste node kalles en *sykel*



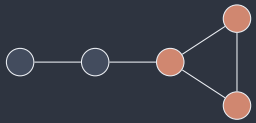
- I rettede grafer må kantene som utgjør en sykel «peke i riktig retning»



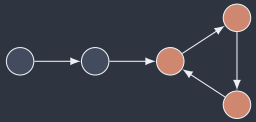
- En graf som ikke inneholder en sykel kalles *asykliske*
- En *urettet* og *asyklisk* graf er et *tre*

Sykler

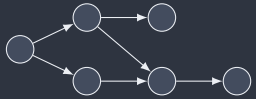
- En sti i med minst tre noder som forbinder første og siste node kalles en *sykel*



- I rettede grafer må kantene som utgjør en sykel «peke i riktig retning»

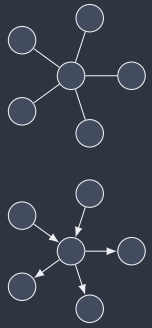


- En graf som ikke inneholder en sykel kalles *asykliske*
- En *urettet* og *asyklisk* graf er et *tre*
- En *rettet* og *asyklisk* graf er en veldig anvendelig type graf, og kalles en *DAG* (eng: directed asyclic graph)



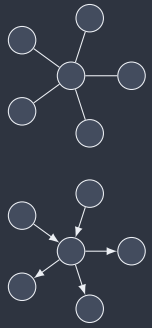
Graden til en node

- Graden til en node, $\deg(v)$, beskriver hvor mange kanter v er koblet sammen med



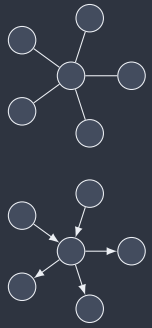
Graden til en node

- Graden til en node, $\deg(v)$, beskriver hvor mange kanter v er koblet sammen med
  - I den øverste grafen til høyre har den midterste noden grad 5



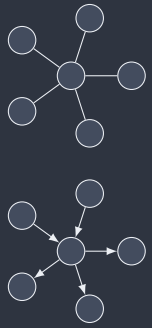
Graden til en node

- Graden til en node, $\deg(v)$, beskriver hvor mange kanter v er koblet sammen med
  - I den øverste grafen til høyre har den midterste noden grad 5
  - De resterende nodene har grad 1



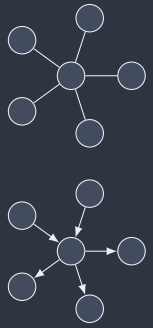
Graden til en node

- Graden til en node, $\deg(v)$, beskriver hvor mange kanter v er koblet sammen med
  - I den øverste grafen til høyre har den midterste noden grad 5
  - De resterende nodene har grad 1
- I en rettet graf skiller man mellom *inn-* og *ut*-grad



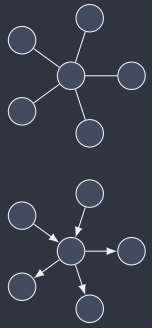
Graden til en node

- Graden til en node, $\deg(v)$, beskriver hvor mange kanter v er koblet sammen med
  - I den øverste grafen til høyre har den midterste noden grad 5
  - De resterende nodene har grad 1
- I en rettet graf skiller man mellom *inn-* og *ut*-grad
  - I den nederste grafen til høyre har den midterste noden inngrad 2 og utgrad 3



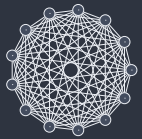
Graden til en node

- Graden til en node, $\deg(v)$, beskriver hvor mange kanter v er koblet sammen med
  - I den øverste grafen til høyre har den midterste noden grad 5
  - De resterende nodene har grad 1
- I en rettet graf skiller man mellom *inn-* og *ut*-grad
  - I den nederste grafen til høyre har den midterste noden inngrad 2 og utgrad 3
  - De resterende nodene har utgrad 1 *eller* inngrad 1



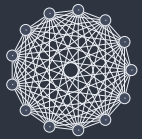
Størrelse av grafer

- I en enkel *komplett* graf er alle noder koblet til hverandre



Størrelse av grafer

- I en enkel *komplett* graf er alle noder koblet til hverandre
  - En slik graf har $\frac{|V|(|V|-1)}{2}$ kanter



Størrelse av grafer

- I en enkel *komplett* graf er alle noder koblet til hverandre
  - En slik graf har $\frac{|V|(|V|-1)}{2}$ kanter
- Dermed er antall kanter $|E|$ i en enkel graf i $\mathcal{O}(|V|^2)$



Størrelse av grafer

- I en enkel *komplett* graf er alle noder koblet til hverandre
  - En slik graf har $\frac{|V|(|V|-1)}{2}$ kanter
- Dermed er antall kanter $|E|$ i en enkel graf i $\mathcal{O}(|V|^2)$
  - Fordi $\mathcal{O}(\frac{|V|(|V|-1)}{2}) = \mathcal{O}(\frac{|V|\cdot|V|-|V|}{2}) = \mathcal{O}(\frac{|V|^2-|V|}{2}) = \mathcal{O}(\frac{|V|^2}{2}) = \mathcal{O}(|V|^2)$



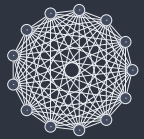
Størrelse av grafer

- I en enkel *komplett* graf er alle noder koblet til hverandre
  - En slik graf har $\frac{|V|(|V|-1)}{2}$ kanter
- Dermed er antall kanter $|E|$ i en enkel graf i $\mathcal{O}(|V|^2)$
  - Fordi $\mathcal{O}(\frac{|V|(|V|-1)}{2}) = \mathcal{O}(\frac{|V|\cdot|V|-|V|}{2}) = \mathcal{O}(\frac{|V|^2-|V|}{2}) = \mathcal{O}(\frac{|V|^2}{2}) = \mathcal{O}(|V|^2)$
- Kjøretidskompleksitet for grafalgoritmer er både avhengig av $|V|$ og $|E|$



Størrelse av grafer

- I en enkel *komplett* graf er alle noder koblet til hverandre
  - En slik graf har $\frac{|V|(|V|-1)}{2}$ kanter
- Dermed er antall kanter $|E|$ i en enkel graf i $\mathcal{O}(|V|^2)$
  - Fordi $\mathcal{O}(\frac{|V|(|V|-1)}{2}) = \mathcal{O}(\frac{|V|\cdot|V|-|V|}{2}) = \mathcal{O}(\frac{|V|^2-|V|}{2}) = \mathcal{O}(\frac{|V|^2}{2}) = \mathcal{O}(|V|^2)$
- Kjøretidskompleksitet for grafalgoritmer er både avhengig av $|V|$ og $|E|$
  - Det er verdt å huske at antall kanter er begrenset av antall noder



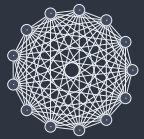
Størrelse av grafer

- I en enkel *komplett* graf er alle noder koblet til hverandre
  - En slik graf har $\frac{|V|(|V|-1)}{2}$ kanter
- Dermed er antall kanter $|E|$ i en enkel graf i $\mathcal{O}(|V|^2)$
  - Fordi $\mathcal{O}(\frac{|V|(|V|-1)}{2}) = \mathcal{O}(\frac{|V|\cdot|V|-|V|}{2}) = \mathcal{O}(\frac{|V|^2-|V|}{2}) = \mathcal{O}(\frac{|V|^2}{2}) = \mathcal{O}(|V|^2)$
- Kjøretidskompleksitet for grafalgoritmer er både avhengig av $|V|$ og $|E|$
  - Det er verdt å huske at antall kanter er begrenset av antall noder
  - ...men ikke vice versa!



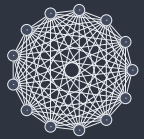
Størrelse av grafer

- I en enkel *komplett* graf er alle noder koblet til hverandre
  - En slik graf har $\frac{|V|(|V|-1)}{2}$ kanter
- Dermed er antall kanter $|E|$ i en enkel graf i $\mathcal{O}(|V|^2)$
  - Fordi $\mathcal{O}(\frac{|V|(|V|-1)}{2}) = \mathcal{O}(\frac{|V|\cdot|V|-|V|}{2}) = \mathcal{O}(\frac{|V|^2-|V|}{2}) = \mathcal{O}(\frac{|V|^2}{2}) = \mathcal{O}(|V|^2)$
- Kjøretidskompleksitet for grafalgoritmer er både avhengig av $|V|$ og $|E|$
  - Det er verdt å huske at antall kanter er begrenset av antall noder
  - …men ikke vice versa!
- Grafer med *mange* kanter relativt til noder kalles ofte *tette* (eng: dense)



Størrelse av grafer

- I en enkel *komplett* graf er alle noder koblet til hverandre
  - En slik graf har $\frac{|V|(|V|-1)}{2}$ kanter
- Dermed er antall kanter $|E|$ i en enkel graf i $\mathcal{O}(|V|^2)$
  - Fordi $\mathcal{O}(\frac{|V|(|V|-1)}{2}) = \mathcal{O}(\frac{|V|\cdot|V|-|V|}{2}) = \mathcal{O}(\frac{|V|^2-|V|}{2}) = \mathcal{O}(\frac{|V|^2}{2}) = \mathcal{O}(|V|^2)$
- Kjøretidskompleksitet for grafalgoritmer er både avhengig av $|V|$ og $|E|$
  - Det er verdt å huske at antall kanter er begrenset av antall noder
  - …men ikke vice versa!
- Grafer med *mange* kanter relativt til noder kalles ofte *tette* (eng: dense)
- Grafer med *få* kanter relativt til noder kalles ofte *tynne* (eng: sparse)



Representasjon

Representasjon av grafer

- En graf er vanligvis representert ved *nabomatrise*, eller en *naboliste*

Representasjon av grafer

- En graf er vanligvis representert ved *nabomatrise*, eller en *naboliste*
- Vi ønsker å enkelt kunne avgjøre om det er en kant mellom to noder u og v

Representasjon av grafer

- En graf er vanligvis representert ved *nabomatrise*, eller en *naboliste*
- Vi ønsker å enkelt kunne avgjøre om det er en kant mellom to noder u og v
- Vi vil anta at vi har tilgang på raske mappinger og mengder

Representasjon av grafer

- En graf er vanligvis representert ved *nabomatrise*, eller en *naboliste*
- Vi ønsker å enkelt kunne avgjøre om det er en kant mellom to noder u og v
- Vi vil anta at vi har tilgang på raske mappinger og mengder
  - Vi skal lære mer om datastrukturer implementert med hashing senere

Representasjon av grafer

- En graf er vanligvis representert ved *nabomatrise*, eller en *naboliste*
- Vi ønsker å enkelt kunne avgjøre om det er en kant mellom to noder u og v
- Vi vil anta at vi har tilgang på raske mappinger og mengder
  - Vi skal lære mer om datastrukturer implementert med hashing senere
    - Tenk `HashMap` og `HashSet` fra Java

Representasjon av grafer

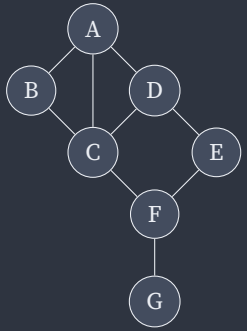
- En graf er vanligvis representert ved *nabomatrise*, eller en *naboliste*
- Vi ønsker å enkelt kunne avgjøre om det er en kant mellom to noder u og v
- Vi vil anta at vi har tilgang på raske mappinger og mengder
  - Vi skal lære mer om datastrukturer implementert med hashing senere
    - Tenk `HashMap` og `HashSet` fra Java
    - Tenk `dictionary` og `set` fra Python

Representasjon av grafer

- En graf er vanligvis representert ved *nabomatrise*, eller en *naboliste*
- Vi ønsker å enkelt kunne avgjøre om det er en kant mellom to noder u og v
- Vi vil anta at vi har tilgang på raske mappinger og mengder
  - Vi skal lære mer om datastrukturer implementert med hashing senere
    - Tenk `HashMap` og `HashSet` fra Java
    - Tenk `dictionary` og `set` fra Python
  - Vi *antar* at dette gir oss *konstant tid* på oppslag

Nabomatrise

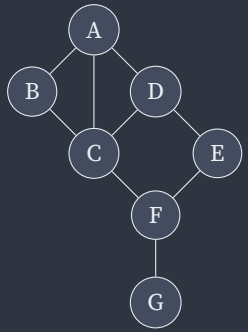
	A	B	C	D	E	F	G
A	0	1	1	1	0	0	0
B	1	0	1	0	0	0	0
C	1	1	0	1	0	1	0
D	1	0	1	0	1	0	0
E	0	0	0	1	0	1	0
F	0	0	1	0	1	0	1
G	0	0	0	0	0	1	0



- Nabomatriser egner seg for tette grafer

Nabomatrise

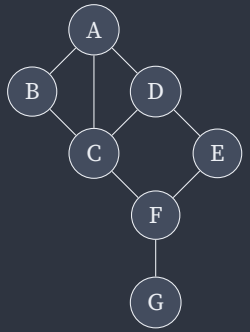
	A	B	C	D	E	F	G
A	0	1	1	1	0	0	0
B	1	0	1	0	0	0	0
C	1	1	0	1	0	1	0
D	1	0	1	0	1	0	0
E	0	0	0	1	0	1	0
F	0	0	1	0	1	0	1
G	0	0	0	0	0	1	0



- Nabomatriser egner seg for tette grafer
  - Minnebruken er i $\mathcal{O}(|V|^2)$

Nabomatrise

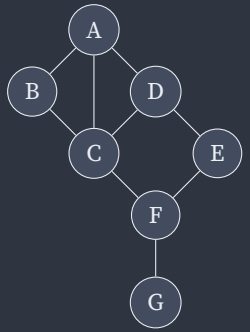
	A	B	C	D	E	F	G
A	0	1	1	1	0	0	0
B	1	0	1	0	0	0	0
C	1	1	0	1	0	1	0
D	1	0	1	0	1	0	0
E	0	0	0	1	0	1	0
F	0	0	1	0	1	0	1
G	0	0	0	0	0	1	0



- Nabomatriser egner seg for tette grafer
  - Minnebruken er i $\mathcal{O}(|V|^2)$
- De er enkle å implementere ved å numerere nodene i grafen fra 0 til $|V| - 1$

Nabomatrise

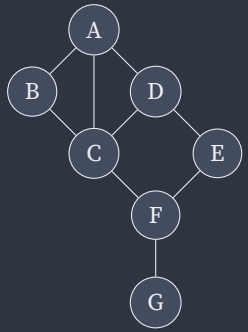
	A	B	C	D	E	F	G
A	0	1	1	1	0	0	0
B	1	0	1	0	0	0	0
C	1	1	0	1	0	1	0
D	1	0	1	0	1	0	0
E	0	0	0	1	0	1	0
F	0	0	1	0	1	0	1
G	0	0	0	0	0	1	0



- Nabomatriser egner seg for tette grafer
  - Minnebruken er i $\mathcal{O}(|V|^2)$
- De er enkle å implementere ved å numerere nodene i grafen fra 0 til $|V| - 1$
- Konstant tid for å sjekke om to noder er naboer

Nabomatrise

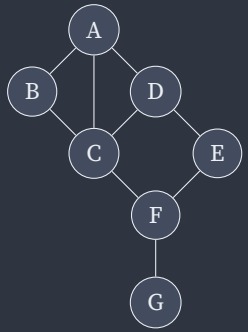
	A	B	C	D	E	F	G
A	0	1	1	1	0	0	0
B	1	0	1	0	0	0	0
C	1	1	0	1	0	1	0
D	1	0	1	0	1	0	0
E	0	0	0	1	0	1	0
F	0	0	1	0	1	0	1
G	0	0	0	0	0	1	0



- Nabomatriser egner seg for tette grafer
  - Minnebruken er i $\mathcal{O}(|V|^2)$
- De er enkle å implementere ved å numerere nodene i grafen fra 0 til $|V| - 1$
- Konstant tid for å sjekke om to noder er naboer
- De egner seg ikke godt når man trenger å finne *alle* naboer av en node (krever $\mathcal{O}(|V|)$ tid)

Naboliste

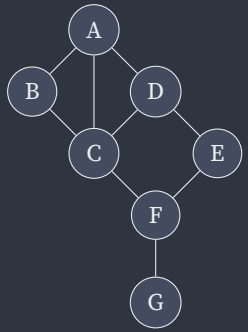
$A : [B, C, D]$
$B : [A, C]$
$C : [A, B, D, F]$
$D : [A, C, E]$
$E : [D, F]$
$F : [C, E, G]$
$G : [F]$



- Egner seg for tynne grafer

Naboliste

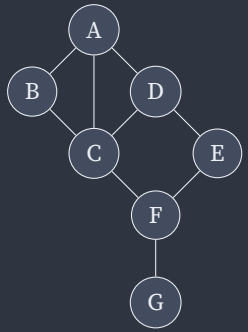
$A : [B, C, D]$
$B : [A, C]$
$C : [A, B, D, F]$
$D : [A, C, E]$
$E : [D, F]$
$F : [C, E, G]$
$G : [F]$



- Egner seg for tynne grafer
  - Minnebruken er i $\mathcal{O}(|V| + |E|)$

Naboliste

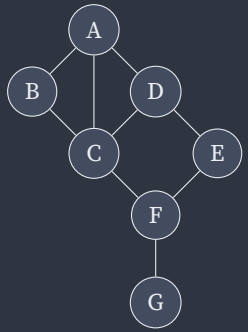
$A : [B, C, D]$
$B : [A, C]$
$C : [A, B, D, F]$
$D : [A, C, E]$
$E : [D, F]$
$F : [C, E, G]$
$G : [F]$



- Egner seg for tynne grafer
  - Minnebruken er i $\mathcal{O}(|V| + |E|)$
- Kan implementeres med en ordbok fra noder til lister av naboer

Naboliste

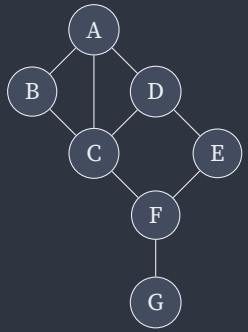
$A : [B, C, D]$
$B : [A, C]$
$C : [A, B, D, F]$
$D : [A, C, E]$
$E : [D, F]$
$F : [C, E, G]$
$G : [F]$



- Egner seg for tynne grafer
  - Minnebruken er i $\mathcal{O}(|V| + |E|)$
- Kan implementeres med en ordbok fra noder til lister av naboer
  - Listene kan erstattes med mengder dersom man trenger å kunne sjekke om to noder er naboer i konstant tid

Naboliste

$A : [B, C, D]$
$B : [A, C]$
$C : [A, B, D, F]$
$D : [A, C, E]$
$E : [D, F]$
$F : [C, E, G]$
$G : [F]$



- Egner seg for tynne grafer
  - Minnebruken er i $\mathcal{O}(|V| + |E|)$
- Kan implementeres med en ordbok fra noder til lister av naboer
  - Listene kan erstattes med mengder dersom man trenger å kunne sjekke om to noder er naboer i konstant tid
- Egner seg når man trenger å finne *alle* naboer av en node (krever $\mathcal{O}(deg(v))$ tid for en node v)

Graftraversering

Graftraversering

- Å traversere en graf betyr år «gå gjennom» hele grafen

Graftraversering

- Å traversere en graf betyr år «gå gjennom» hele grafen
  - Gjerne med en spesifikk strategi

Graftraversering

- Å traversere en graf betyr år «gå gjennom» hele grafen
  - Gjerne med en spesifikk strategi
- Vi kan enkelt besvare spørsmål som

Graftraversering

- Å traversere en graf betyr år «gå gjennom» hele grafen
  - Gjerne med en spesifikk strategi
- Vi kan enkelt besvare spørsmål som
  - Hvilke noder kan nås fra en gitt startnode s?

Graftraversering

- Å traversere en graf betyr år «gå gjennom» hele grafen
  - Gjerne med en spesifikk strategi
- Vi kan enkelt besvare spørsmål som
  - Hvilke noder kan nås fra en gitt startnode s?
  - Hvor mange komponenter har grafen?

Graftraversering

- Å traversere en graf betyr år «gå gjennom» hele grafen
  - Gjerne med en spesifikk strategi
- Vi kan enkelt besvare spørsmål som
  - Hvilke noder kan nås fra en gitt startnode s?
  - Hvor mange komponenter har grafen?
- Vi skal se på to traverseringer:

Graftraversering

- Å traversere en graf betyr år «gå gjennom» hele grafen
  - Gjerne med en spesifikk strategi
- Vi kan enkelt besvare spørsmål som
  - Hvilke noder kan nås fra en gitt startnode s?
  - Hvor mange komponenter har grafen?
- Vi skal se på to traverseringer:
  - Dybde-først søk, traversering med en *stack*

Graftraversering

- Å traversere en graf betyr år «gå gjennom» hele grafen
  - Gjerne med en spesifikk strategi
- Vi kan enkelt besvare spørsmål som
  - Hvilke noder kan nås fra en gitt startnode *s*?
  - Hvor mange komponenter har grafen?
- Vi skal se på to traverseringer:
  - Dybde-først søk, traversering med en *stack*
  - Bredde-først søk, traversering med en *kø*

Dybde-først søk

- Dybde-først søk følger en vilkårlig sti *vekk fra* en gitt startnode

Dybde-først søk

- Dybde-først søk følger en vilkårlig sti *vekk fra* en gitt startnode
  - Når den ikke kan finne en *ubesøkt* node sporer den stien tilbake

Dybde-først søk

- Dybde-først søk følger en vilkårlig sti *vekk fra* en gitt startnode
  - Når den ikke kan finne en *ubesøkt* node sporer den stien tilbake
  - I hvert ledd forsøker den å følge ubesøkte noder som leder vekk fra noden den er i

Dybde-først søk

- Dybde-først søk følger en vilkårlig sti *vekk fra* en gitt startnode
  - Når den ikke kan finne en *ubesøkt* node sporer den stien tilbake
  - I hvert ledd forsøker den å følge ubesøkte noder som leder vekk fra noden den er i
- Dybde-først søk er som regel den enkleste og raskeste traverseringen

Dybde-først søk

- Dybde-først søk følger en vilkårlig sti *vekk fra* en gitt startnode
  - Når den ikke kan finne en *ubesøkt* node sporer den stien tilbake
  - I hvert ledd forsøker den å følge ubesøkte noder som leder vekk fra noden den er i
- Dybde-først søk er som regel den enkleste og raskeste traverseringen
  - Mest på grunn av lavere minnebruk

Dybde-først søk

- Dybde-først søk følger en vilkårlig sti *vekk fra* en gitt startnode
  - Når den ikke kan finne en *ubesøkt* node sporer den stien tilbake
  - I hvert ledd forsøker den å følge ubesøkte noder som leder vekk fra noden den er i
- Dybde-først søk er som regel den enkleste og raskeste traverseringen
  - Mest på grunn av lavere minnebruk
- Implementeres som regel med rekursjon

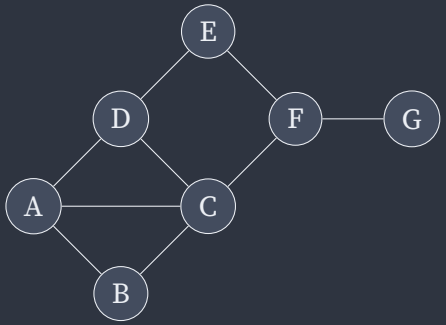
Dybde-først søk

- Dybde-først søk følger en vilkårlig sti *vekk fra* en gitt startnode
  - Når den ikke kan finne en *ubesøkt* node sporer den stien tilbake
  - I hvert ledd forsøker den å følge ubesøkte noder som leder vekk fra noden den er i
- Dybde-først søk er som regel den enkleste og raskeste traverseringen
  - Mest på grunn av lavere minnebruk
- Implementeres som regel med rekursjon
  - Men kan også implementeres med en eksplisitt stack

Dybde-først søk

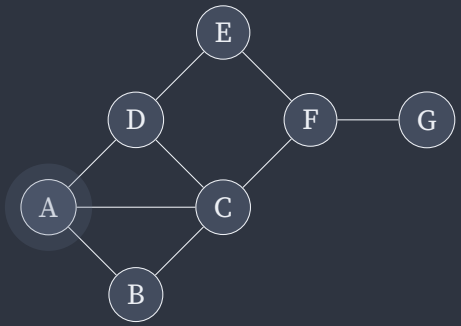
- Dybde-først søk følger en vilkårlig sti *vekk fra* en gitt startnode
  - Når den ikke kan finne en *ubesøkt* node sporer den stien tilbake
  - I hvert ledd forsøker den å følge ubesøkte noder som leder vekk fra noden den er i
- Dybde-først søk er som regel den enkleste og raskeste traverseringen
  - Mest på grunn av lavere minnebruk
- Implementeres som regel med rekursjon
  - Men kan også implementeres med en eksplisitt stack
  - (Rekursive kall legges på en stack, så dette er å erstatte en stack med en annen)

Dybde-først søk (illustrasjon)



`stack`

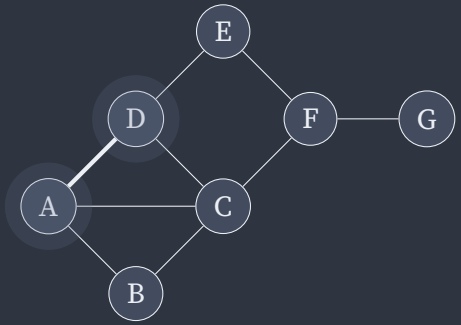
Dybde-først søk (illustrasjon)



A

`stack`

Dybde-først søk (illustrasjon)

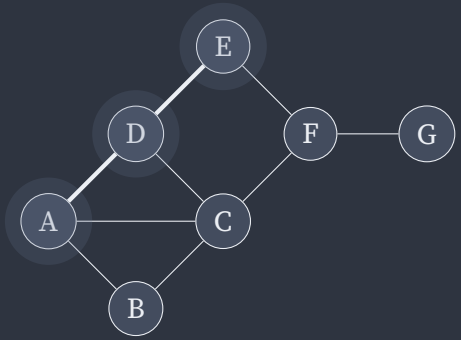


D

A

stack

Dybde-først søk (illustrasjon)



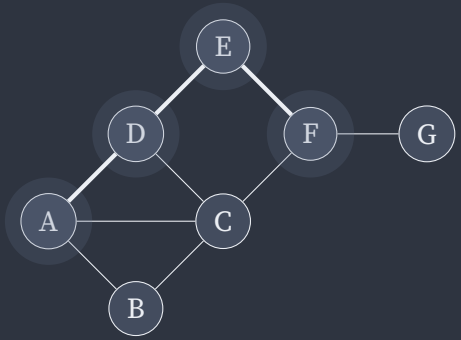
E

D

A

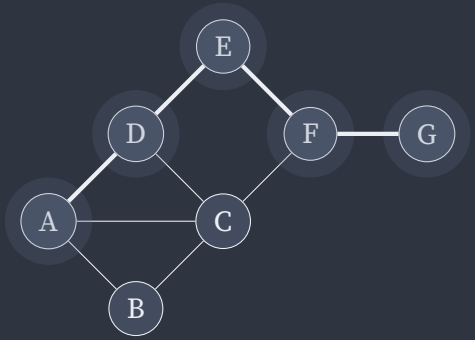
stack

Dybde-først søk (illustrasjon)



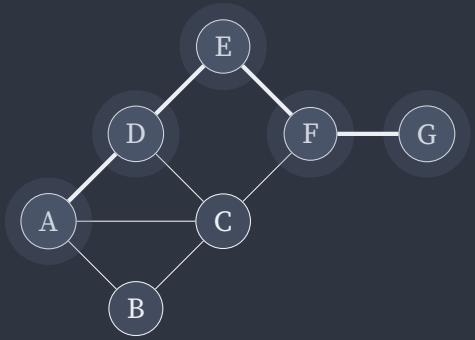
F
E
D
A
stack

Dybde-først søk (illustrasjon)



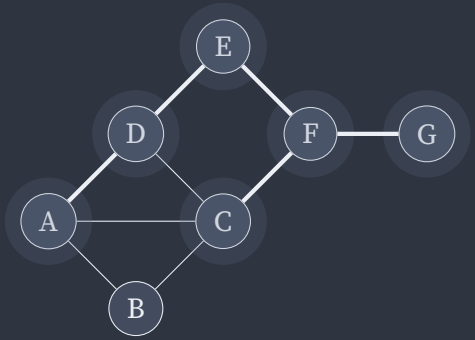
G
F
E
D
A
stack

Dybde-først søk (illustrasjon)



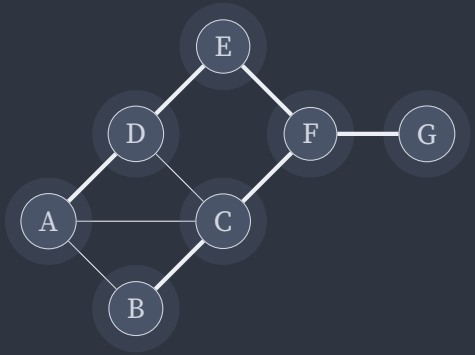
F
E
D
A
stack

Dybde-først søk (illustrasjon)



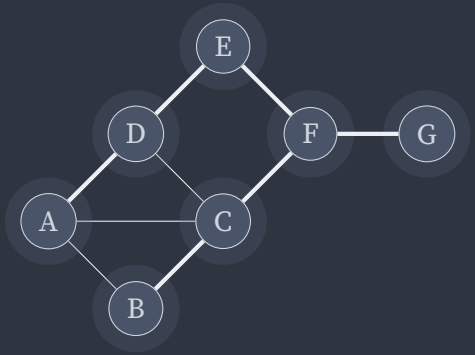
C
F
E
D
A
stack

Dybde-først søk (illustrasjon)



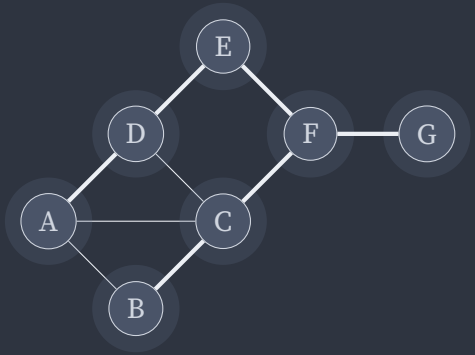
B
C
F
E
D
A
stack

Dybde-først søk (illustrasjon)



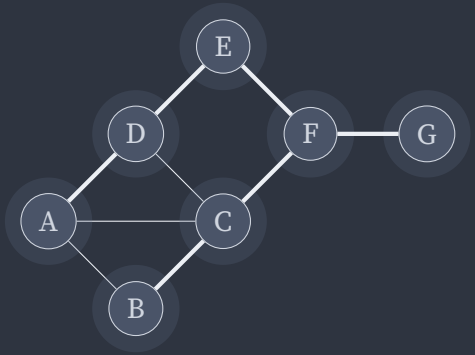
C
F
E
D
A
stack

Dybde-først søk (illustrasjon)



F
E
D
A
stack

Dybde-først søk (illustrasjon)



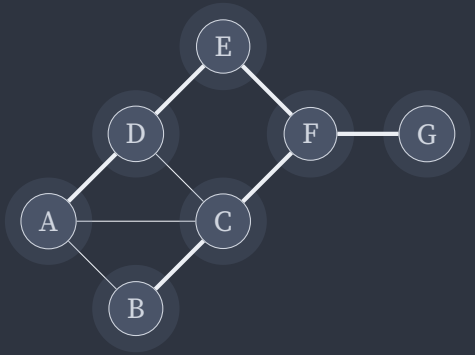
E

D

A

stack

Dybde-først søk (illustrasjon)

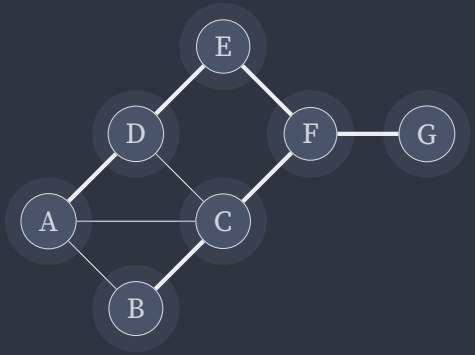


D

A

stack

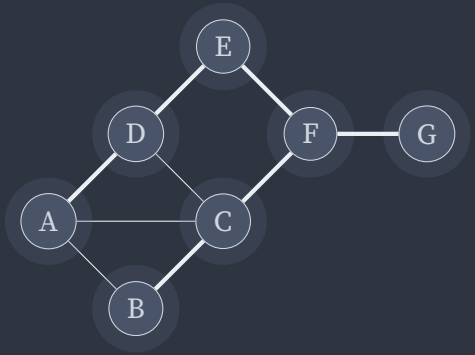
Dybde-først søk (illustrasjon)



A

`stack`

Dybde-først søk (illustrasjon)



`stack`

Dybde-først søk (rekursiv)

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
```

Dybde-først søk (rekursiv)

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
  |   |
  |   |
```

Dybde-først søk (rekursiv)

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)
```

Algorithm: Fullt rekursivt dybde-først søk

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

Output: Besøker alle noder i G nøyaktig én gang dybde først

```
1 Procedure DFSFull(G)
2     visited ← empty set
```

Dybde-først søk (rekursiv)

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)
```

Algorithm: Fullt rekursivt dybde-først søk

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

Output: Besøker alle noder i G nøyaktig én gang dybde først

```
1 Procedure DFSFull(G)
2 |   visited ← empty set
3 |   for v ∈ V do
```

Dybde-først søk (rekursiv)

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)
```

Algorithm: Fullt rekursivt dybde-først søk

Input: En graf $G = (V, E)$
Output: Besøker alle noder i G nøyaktig én gang dybde først

```
1 Procedure DFSFull(G)
2 |   visited ← empty set
3 |   for v ∈ V do
4 |   |   if v ∉ visited then
5 |   |   |   DFSVisit(G, v, visited)
```

- `DFSVisit` fra en node u besøker alle nodene i komponenten til u

Dybde-først søk (rekursiv)

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)
```

Algorithm: Fullt rekursivt dybde-først søk

Input: En graf $G = (V, E)$
Output: Besøker alle noder i G nøyaktig én gang dybde først

```
1 Procedure DFSFull(G)
2 |   visited ← empty set
3 |   for v ∈ V do
4 |   |   if v ∉ visited then
5 |   |   |   DFSVisit(G, v, visited)
```

- `DFSVisit` fra en node u besøker alle nodene i komponenten til u
- `DFSFull` sørger for at vi går gjennom alle komponenter i G

Dybde-først søk (rekursiv)

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)
```

Algorithm: Fullt rekursivt dybde-først søk

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

Output: Besøker alle noder i G nøyaktig én gang dybde først

```
1 Procedure DFSFull(G)
2 |   visited ← empty set
3 |   for v ∈ V do
4 |   |   if v ∉ visited then
5 |   |   |   DFSVisit(G, v, visited)
```

- `DFSVisit` fra en node u besøker alle nodene i komponenten til u
- `DFSFull` sørger for at vi går gjennom alle komponenter i G
- Med nabolister tar det $\mathcal{O}(deg(u))$ tid å gå gjennom alle naboene til u

Dybde-først søk (rekursiv)

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)
```

Algorithm: Fullt rekursivt dybde-først søk

Input: En graf $G = (V, E)$
Output: Besøker alle noder i G nøyaktig én gang dybde først

```
1 Procedure DFSFull(G)
2 |   visited ← empty set
3 |   for v ∈ V do
4 |   |   if v ∉ visited then
5 |   |   |   DFSVisit(G, v, visited)
```

- `DFSVisit` fra en node u besøker alle nodene i komponenten til u
- `DFSFull` sørger for at vi går gjennom alle komponenter i G
- Med nabolister tar det $\mathcal{O}(deg(u))$ tid å gå gjennom alle naboene til u
- Dette gjøres for alle noder, så vi får summen av gradene til alle nodene i G

Dybde-først søk (rekursiv)

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)
```

Algorithm: Fullt rekursivt dybde-først søk

Input: En graf $G = (V, E)$
Output: Besøker alle noder i G nøyaktig én gang dybde først

```
1 Procedure DFSFull(G)
2 |   visited ← empty set
3 |   for v ∈ V do
4 |   |   if v ∉ visited then
5 |   |   |   DFSVisit(G, v, visited)
```

- `DFSVisit` fra en node u besøker alle nodene i komponenten til u
- `DFSFull` sørger for at vi går gjennom alle komponenter i G
- Med nabolister tar det $\mathcal{O}(deg(u))$ tid å gå gjennom alle naboene til u
- Dette gjøres for alle noder, så vi får summen av gradene til alle nodene i G
  - Med andre ord, vi sjekker hver kant i grafen, som er $\mathcal{O}(|E|)$

Dybde-først søk (rekursiv)

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)
```

Algorithm: Fullt rekursivt dybde-først søk

Input: En graf $G = (V, E)$
Output: Besøker alle noder i G nøyaktig én gang dybde først

```
1 Procedure DFSFull(G)
2 |   visited ← empty set
3 |   for v ∈ V do
4 |   |   if v ∉ visited then
5 |   |   |   DFSVisit(G, v, visited)
```

- `DFSVisit` fra en node u besøker alle nodene i komponenten til u
- `DFSFull` sørger for at vi går gjennom alle komponenter i G
- Med nabolister tar det $\mathcal{O}(deg(u))$ tid å gå gjennom alle naboene til u
- Dette gjøres for alle noder, så vi får summen av gradene til alle nodene i G
  - Med andre ord, vi sjekker hver kant i grafen, som er $\mathcal{O}(|E|)$
- I tillegg går `DFSFull` gjennom alle nodene

Dybde-først søk (rekursiv)

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)
```

ALGORITHM: FULLT REKURSIVT DYBDE-FØRST SØK

Input: En graf $G = (V, E)$
Output: Besøker alle noder i G nøyaktig én gang dybde først

```
1 Procedure DFSFull(G)
2 |   visited ← empty set
3 |   for v ∈ V do
4 |   |   if v ∉ visited then
5 |   |   |   DFSVisit(G, v, visited)
```

- `DFSVisit` fra en node u besøker alle nodene i komponenten til u
- `DFSFull` sørger for at vi går gjennom alle komponenter i G
- Med nabolister tar det $\mathcal{O}(deg(u))$ tid å gå gjennom alle naboene til u
- Dette gjøres for alle noder, så vi får summen av gradene til alle nodene i G
  - Med andre ord, vi sjekker hver kant i grafen, som er $\mathcal{O}(|E|)$
- I tillegg går `DFSFull` gjennom alle nodene
- Den totale kjøretiden blir $\mathcal{O}(|V| + |E|)$

Dybde-først søk (iterativt)

Algorithm: Iterativt dybde-først søk

Input: En graf $G = (V, E)$, 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 DFSVisit(G, s, visited)
2     stack ← singleton stack containing s
```

Dybde-først søk (iterativt)

ALGORITHM: ITERATIVT DYBDE-FØRST SØK

Input: En graf $G = (V, E)$, 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 DFSVisit(G, s, visited)
2     stack ← singleton stack containing s
3     while stack is not empty do
4         u ← stack.pop()
```

Dybde-først søk (iterativt)

Algorithm: Iterativt dybde-først søk

Input: En graf $G = (V, E)$, 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 DFSVisit(G, s, visited)
2     stack ← singleton stack containing s
3     while stack is not empty do
4         u ← stack.pop()
5         if u ∉ visited then
6             add u to visited
```

Dybde-først søk (iterativt)

Algorithm: Iterativt dybde-først søk

Input: En graf $G = (V, E)$, 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 DFSVisit(G, s, visited)
2     stack ← singleton stack containing s
3     while stack is not empty do
4         u ← stack.pop()
5         if u ∉ visited then
6             add u to visited
7             for (u, v) ∈ E do
8                 stack.push(v)
```

- Dette er en alternativ implementasjon som bruker en eksplisitt stack i stedet for rekursjon

Bredde-først søk

- Bredde-først søk besøker hele tiden den nærmeste ubesøkte noden fra startnoden

Bredde-først søk

- Bredde-først søk besøker hele tiden den nærmeste ubesøkte noden fra startnoden
  - Den jobber seg lagvis gjennom grafen

Bredde-først søk

- Bredde-først søk besøker hele tiden den nærmeste ubesøkte noden fra startnoden
  - Den jobber seg lagvis gjennom grafen
  - Den besøker først alle direkte naboer

Bredde-først søk

- Bredde-først søk besøker hele tiden den nærmeste ubesøkte noden fra startnoden
  - Den jobber seg lagvis gjennom grafen
  - Den besøker først alle direkte naboer
  - Så alle naboene sine naboer

Bredde-først søk

- Bredde-først søk besøker hele tiden den nærmeste ubesøkte noden fra startnoden
  - Den jobber seg lagvis gjennom grafen
  - Den besøker først alle direkte naboer
  - Så alle naboene sine naboer
  - Så alle naboene sine naboer sine naboer

Bredde-først søk

- Bredde-først søk besøker hele tiden den nærmeste ubesøkte noden fra startnoden
  - Den jobber seg lagvis gjennom grafen
  - Den besøker først alle direkte naboer
  - Så alle naboene sine naboer
  - Så alle naboene sine naboer sine naboer
  - …

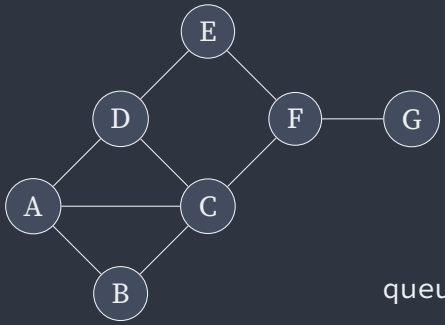
Bredde-først søk

- Bredde-først søk besøker hele tiden den nærmeste ubesøkte noden fra startnoden
  - Den jobber seg lagvis gjennom grafen
  - Den besøker først alle direkte naboer
  - Så alle naboene sine naboer
  - Så alle naboene sine naboer sine naboer
  - ...
- Brukes for å finne korteste stier fra en startnode til andre noder

Bredde-først søk

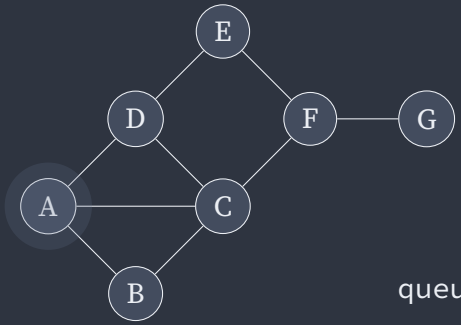
- Bredde-først søk besøker hele tiden den nærmeste ubesøkte noden fra startnoden
  - Den jobber seg lagvis gjennom grafen
  - Den besøker først alle direkte naboer
  - Så alle naboene sine naboer
  - Så alle naboene sine naboer sine naboer
  - ...
- Brukes for å finne korteste stier fra en startnode til andre noder
- Implementeres iterativt med en *kø*

Bredde-først søk (illustrasjon)



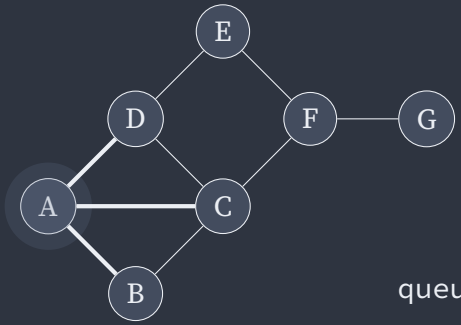
queue

Bredde-først søk (illustrasjon)



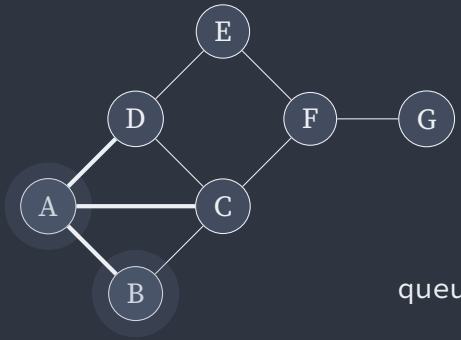
queue **A**

Bredde-først søk (illustrasjon)



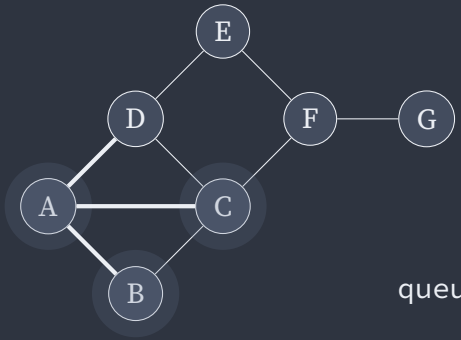
queue **A** B C D

Bredde-først søk (illustrasjon)



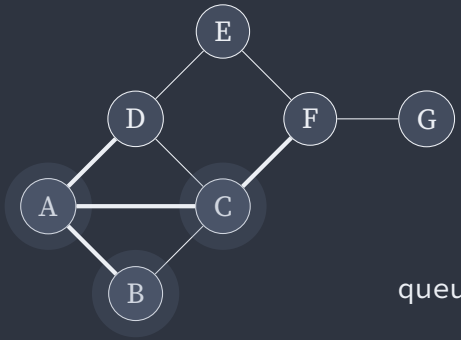
queue A **B** C D

Bredde-først søk (illustrasjon)



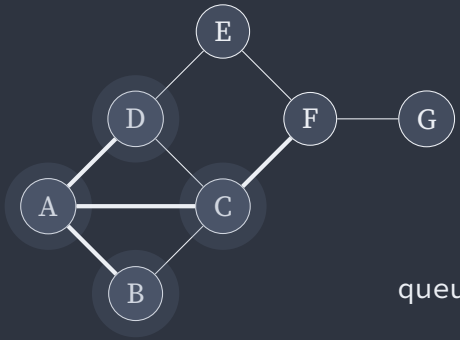
queue A B **C** D

Bredde-først søk (illustrasjon)



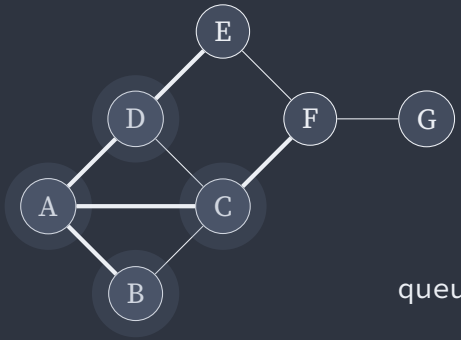
queue A B **C** D F

Bredde-først søk (illustrasjon)



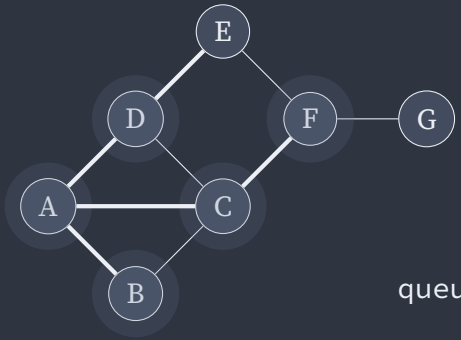
queue A B C **D** F

Bredde-først søk (illustrasjon)



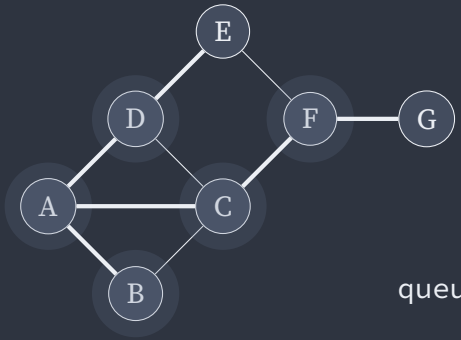
queue A B C **D** F E

Bredde-først søk (illustrasjon)



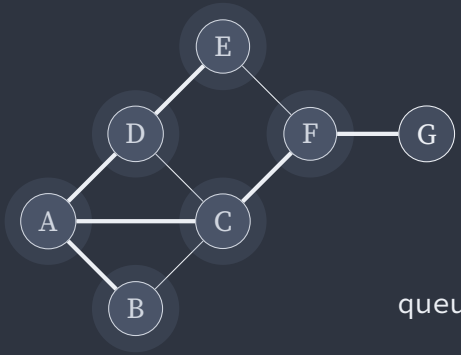
queue A B C D **F** E

Bredde-først søk (illustrasjon)



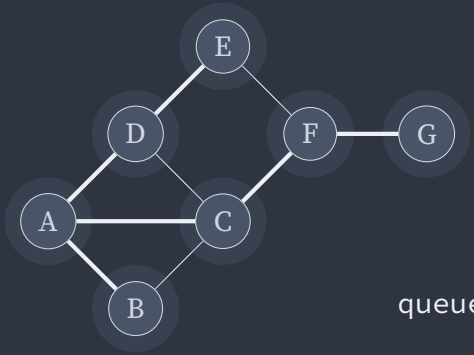
queue A B C D **F** E G

Bredde-først søk (illustrasjon)



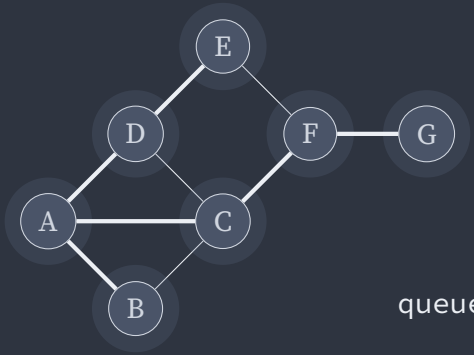
queue A B C D F **E** G

Bredde-først søk (illustrasjon)



queue A B C D F E **G**

Bredde-først søk (illustrasjon)



queue A B C D F E G

Bredde-først søk (implementasjon)

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 ← singleton queue containing s
```

Bredde-først søk (implementasjon)

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 ← singleton queue containing s
4     while queue is not empty do
5         u ← queue.dequeue()
```

Bredde-først søk (implementasjon)

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 ← singleton 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)
```

ALGORITHM: FULLT BREDDE-FØRST SØK

Input: En graf $G = (V, E)$
Output: Besøker alle noder i G nøyaktig én gang bredde først

```
1 Procedure BFSFull(G)
2     visited ← empty set
```

Bredde-først søk (implementasjon)

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 ← singleton 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)
```

ALGORITHM: FULLT BREDDE-FØRST SØK

Input: En graf $G = (V, E)$
Output: Besøker alle noder i G nøyaktig én gang bredde først

```
1 Procedure BFSFull(G)
2     visited ← empty set
3     for v ∈ V do
```

Bredde-først søk (implementasjon)

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 ← singleton 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)
```

ALGORITHM: FULLT BREDDE-FØRST SØK

Input: En graf $G = (V, E)$
Output: Besøker alle noder i G nøyaktig én gang bredde først

```
1 Procedure BFSFull(G)
2     visited ← empty set
3     for v ∈ V do
4         if v ∉ visited then
5             BFSVisit(G, v, visited)
```

- I `BFSVisit` vil hver node vil legge alle sine naboer på på køen

Bredde-først søk (implementasjon)

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 ← singleton 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)
```

ALGORITHM: FULLT BREDDE-FØRST SØK

Input: En graf $G = (V, E)$
Output: Besøker alle noder i G nøyaktig én gang bredde først

```
1 Procedure BFSFull(G)
2     visited ← empty set
3     for v ∈ V do
4         if v ∉ visited then
5             BFSVisit(G, v, visited)
```

- I `BFSVisit` vil hver node vil legge alle sine naboer på på køen
- I tillegg går `BFSFull` gjennom alle nodene

Bredde-først søk (implementasjon)

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 ← singleton 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)
```

ALGORITHM: FULLT BREDDE-FØRST SØK

Input: En graf $G = (V, E)$
Output: Besøker alle noder i G nøyaktig én gang bredde først

```
1 Procedure BFSFull(G)
2     visited ← empty set
3     for v ∈ V do
4         if v ∉ visited then
5             BFSVisit(G, v, visited)
```

- I `BFSVisit` vil hver node vil legge alle sine naboer på på køen
- I tillegg går `BFSFull` gjennom alle nodene
- Den totale kjøretiden blir $\mathcal{O}(|V| + |E|)$

Topologisk sortering

Topologisk sortering

- Topologisk sortering ordner nodene i en rettet asyklisk graf (DAG)

Topologisk sortering

- Topologisk sortering ordner nodene i en rettet asyklisk graf (DAG)
- En avhengighetsgraf er et typisk eksempel på en DAG

Topologisk sortering

- Topologisk sortering ordner nodene i en rettet asyklisk graf (DAG)
- En avhengighetsgraf er et typisk eksempel på en DAG
- En topologisk sortering av en avhengighetsgraf gir en mulig «gjennomføringsplan»

Topologisk sortering

- Topologisk sortering ordner nodene i en rettet asyklisk graf (DAG)
- En avhengighetsgraf er et typisk eksempel på en DAG
- En topologisk sortering av en avhengighetsgraf gir en mulig «gjennomføringsplan»
- Grafer som representerer noe med tid er ofte DAGs

Topologisk sortering

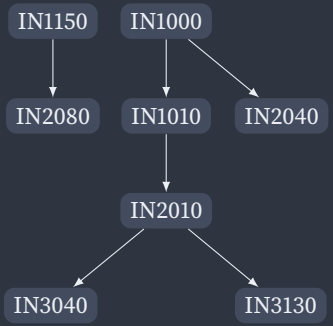
- Topologisk sortering ordner nodene i en rettet asyklisk graf (DAG)
- En avhengighetsgraf er et typisk eksempel på en DAG
- En topologisk sortering av en avhengighetsgraf gir en mulig «gjennomføringsplan»
- Grafer som representerer noe med tid er ofte DAGs
- For eksempel kan nodene representere hendelser og de rettede kantene at noe skjedde før noe annet

Topologisk sortering

- Topologisk sortering ordner nodene i en rettet asyklisk graf (DAG)
- En avhengighetsgraf er et typisk eksempel på en DAG
- En topologisk sortering av en avhengighetsgraf gir en mulig «gjennomføringsplan»
- Grafer som representerer noe med tid er ofte DAGs
- For eksempel kan nodene representere hendelser og de rettede kantene at noe skjedde før noe annet
- Hvis vi sorterer en slik graf ved hjelp av topologisk sortering får vi et mulig hendelsesforløp

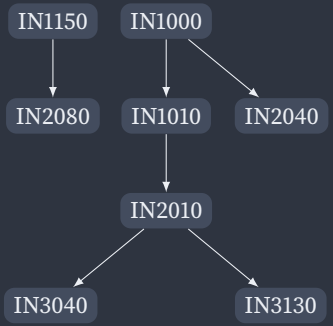
Topologisk ordning av emner

- I figuren ser vi noen emner fra IFI



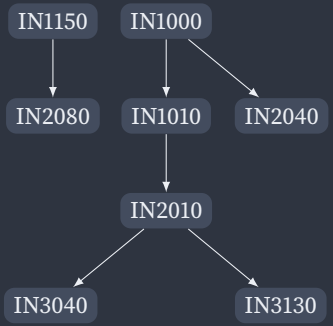
Topologisk ordning av emner

- I figuren ser vi noen emner fra IFI
- En kant tolkes som at et emne inneholder antatte forkunnskaper for et annet



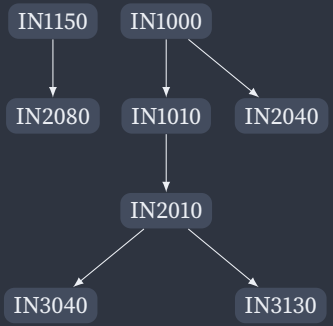
Topologisk ordning av emner

- I figuren ser vi noen emner fra IFI
- En kant tolkes som at et emne inneholder antatte forkunnskaper for et annet
- En topologisk sortering gir deg en rekkefølge på nodene



Topologisk ordning av emner

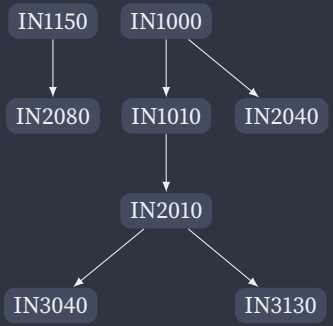
- I figuren ser vi noen emner fra IFI
- En kant tolkes som at et emne inneholder antatte forkunnskaper for et annet
- En topologisk sortering gir deg en rekkefølge på nodene
  - slik at du oppfyller alle forkunnskaper når du tar kurset



Topologisk ordning av emner

- I figuren ser vi noen emner fra IFI
- En kant tolkes som at et emne inneholder antatte forkunnskaper for et annet
- En topologisk sortering gir deg en rekkefølge på nodene
  - slik at du oppfyller alle forkunnskaper når du tar kurset
- Et eksempel på en topologisk ordning av emnene er:

  IN1150, IN2080, IN1000, IN2040, IN1010, IN2010, IN3130, IN3040

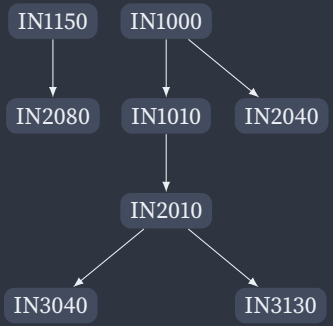


Topologisk ordning av emner

- I figuren ser vi noen emner fra IFI
- En kant tolkes som at et emne inneholder antatte forkunnskaper for et annet
- En topologisk sortering gir deg en rekkefølge på nodene
  - slik at du oppfyller alle forkunnskaper når du tar kurset
- Et eksempel på en topologisk ordning av emnene er:

  IN1150, IN2080, IN1000, IN2040, IN1010, IN2010, IN3130, IN3040
- En *annen* topologisk ordning av emnene er:

  IN1000,IN1150, IN1010, IN2040, IN2010, IN2080, IN3130, IN3040

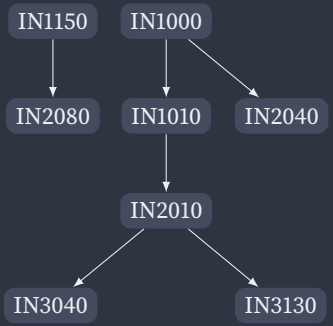


Topologisk ordning av emner

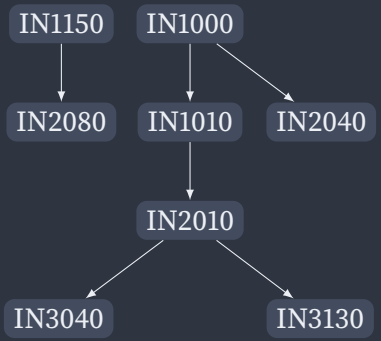
- I figuren ser vi noen emner fra IFI
- En kant tolkes som at et emne inneholder antatte forkunnskaper for et annet
- En topologisk sortering gir deg en rekkefølge på nodene
  - slik at du oppfyller alle forkunnskaper når du tar kurset
- Et eksempel på en topologisk ordning av emnene er:

  IN1150, IN2080, IN1000, IN2040, IN1010, IN2010, IN3130, IN3040
- En *annen* topologisk ordning av emnene er:

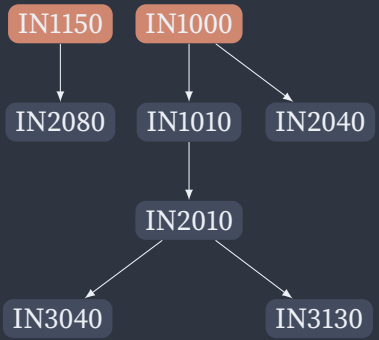
  IN1000,IN1150, IN1010, IN2040, IN2010, IN2080, IN3130, IN3040
- Merk at det kan finnes flere topologiske ordninger!



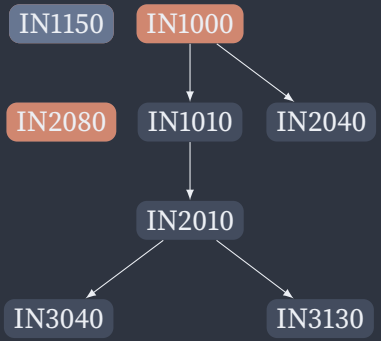
Topologisk sortering (illustrasjon)



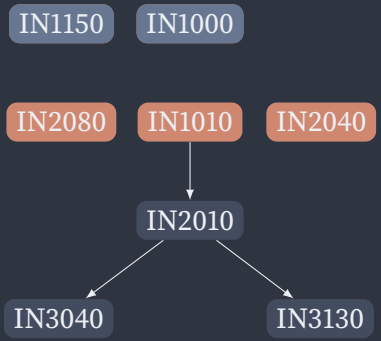
Topologisk sortering (illustrasjon)



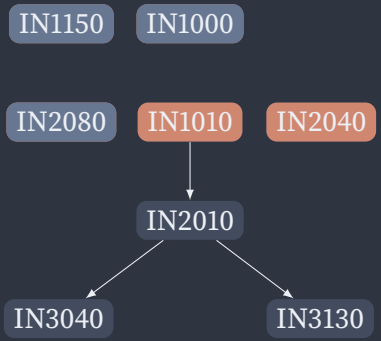
Topologisk sortering (illustrasjon)



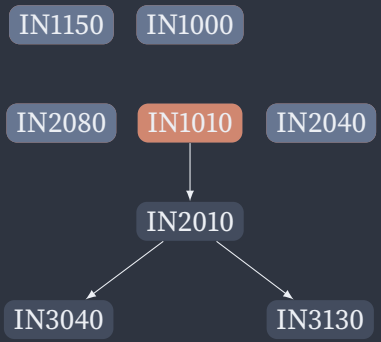
Topologisk sortering (illustrasjon)



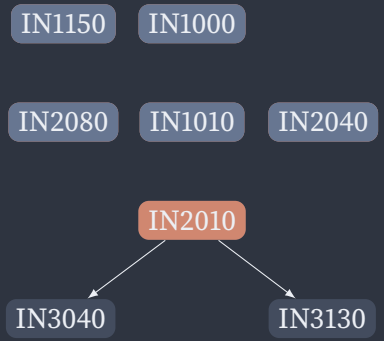
Topologisk sortering (illustrasjon)



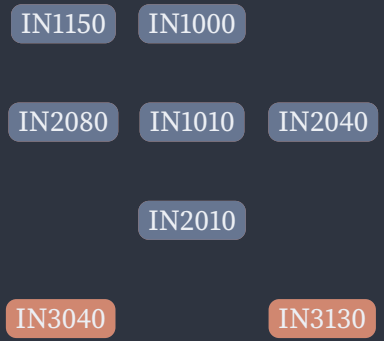
Topologisk sortering (illustrasjon)



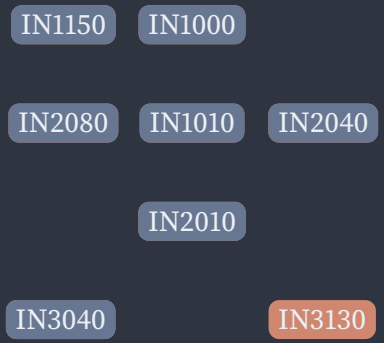
Topologisk sortering (illustrasjon)



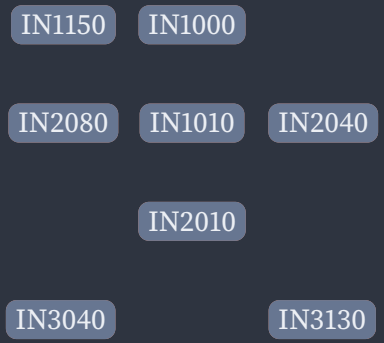
Topologisk sortering (illustrasjon)



Topologisk sortering (illustrasjon)



Topologisk sortering (illustrasjon)



Topologisk sortering (algoritme)

- Den (konseptuelt) enkleste algoritmen for topologisk sortering

Topologisk sortering (algoritme)

- Den (konseptuelt) enkleste algoritmen for topologisk sortering
- Ideen er å finne en node med inngrad null

Topologisk sortering (algoritme)

- Den (konseptuelt) enkleste algoritmen for topologisk sortering
- Ideen er å finne en node med inngrad null
  - Så fjerne den og alle utgående kanter

Topologisk sortering (algoritme)

- Den (konseptuelt) enkleste algoritmen for topologisk sortering
- Ideen er å finne en node med inngrad null
  - Så fjerne den og alle utgående kanter
  - Fortsett med neste node som nå har inngrad null

Topologisk sortering (algoritme)

- Den (konseptuelt) enkleste algoritmen for topologisk sortering
- Ideen er å finne en node med inngrad null
  - Så fjerne den og alle utgående kanter
  - Fortsett med neste node som nå har inngrad null
- Dersom det ikke går å topologisk sortere nodene på denne måten

Topologisk sortering (algoritme)

- Den (konseptuelt) enkleste algoritmen for topologisk sortering
- Ideen er å finne en node med inngrad null
  - Så fjerne den og alle utgående kanter
  - Fortsett med neste node som nå har inngrad null
- Dersom det ikke går å topologisk sortere nodene på denne måten
  - så kan vi konkludere med at grafen inneholder en sykel!

Topologisk sortering (algoritme)

- Den (konseptuelt) enkleste algoritmen for topologisk sortering
- Ideen er å finne en node med inngrad null
  - Så fjerne den og alle utgående kanter
  - Fortsett med neste node som nå har inngrad null
- Dersom det ikke går å topologisk sortere nodene på denne måten
  - så kan vi konkludere med at grafen inneholder en sykel!
- Algoritmen er i $\mathcal{O}(|V| + |E|)$

Topologisk sortering (implementasjon)

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
```

Topologisk sortering (implementasjon)

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)
```

Topologisk sortering (implementasjon)

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
```

Topologisk sortering (implementasjon)

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
```

Topologisk sortering (implementasjon)

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
```

Topologisk sortering (implementasjon)

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
```

Topologisk sortering med DFS

- En graf kan også topologisk sorteres ved hjelp av et dybde-først søk

Topologisk sortering med DFS

- En graf kan også topologisk sorteres ved hjelp av et dybde-først søk
- Modifiser dybde-først søket slik at noden legges på en stack *etter alle dens naboer er prossesert*

Topologisk sortering med DFS

- En graf kan også topologisk sorteres ved hjelp av et dybde-først søk
- Modifiser dybde-først søket slik at noden legges på en stack *etter alle dens naboer er prossesert*
- Til slutt kan man poppe alle nodene av stacken i topologisk ordnet

Topologisk sortering med DFS

- En graf kan også topologisk sorteres ved hjelp av et dybde-først søk
- Modifiser dybde-først søket slik at noden legges på en stack *etter alle dens naboer er prossesert*
- Til slutt kan man poppe alle nodene av stacken i topologisk ordnet
- Intuisjonen er at nodene med utgrad null legges først på stacken

Topologisk sortering med DFS

- En graf kan også topologisk sorteres ved hjelp av et dybde-først søk
- Modifiser dybde-først søket slik at noden legges på en stack *etter alle dens naboer er prossesert*
- Til slutt kan man poppe alle nodene av stacken i topologisk ordnet
- Intuisjonen er at nodene med utgrad null legges først på stacken
  - Det kan være litt vanskelig å overbevise seg om at det ikke spiller noen rolle hvilken node søket starter i

Topologisk sortering med DFS (implementasjon)

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)
```