Grafer, traversering og topologisk sortering

IN2010 – Algoritmer og Datastrukturer

Lars Tveito

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

Høsten 2022

Oversikt uke 39 – 42

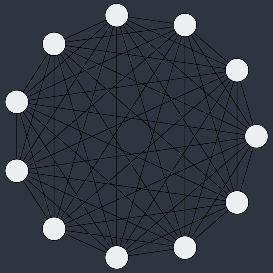
OVERSIKT UKE 39 – 42

- 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

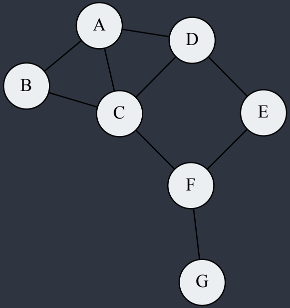
Definisjoner og terminologi

EKSEMPLER PÅ GRAFER

- Kart
- Bekjentskapsgrafer
- Nettverk
- Tilstander
- Trær



Grafer

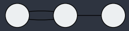
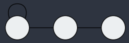
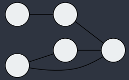


- En mengde med noder: $\{A, B, C, D, E, F, G\}$
- En mengde med kanter:

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

- 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	
(Enkle) løkker	En kant fra en node til seg selv	
Urettet/rettet	Kantene i grafen har retning	 vs 
Vektet/uvektet	Kantene har en verdi	 vs 
Enkel graf	Uten løkker, parallelle kanter, retning og vekt	

GRAFER MED OG UTEN RETNING

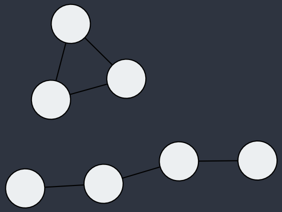
- En graf kan ha rettede kanter:  vs 
- For *urettede* grafer skriver vi $\{u, v\}$ for en kant mellom u og v
  - En slik kant kan «krysses i begge retninger»
- For *rettede* grafer skriver vi (u, v) for en kant fra u til v
  - En slik kant kan kun følges *fra u til v*, men *ikke fra v til u*
- Urettede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

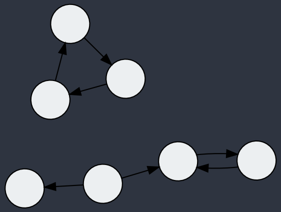
Veier og stier

- Sti: En sekvens av noder i grafen, forbundet av kanter, der ingen *noder* gjentas: A, B, C, D
- 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, som er tema for neste uke

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

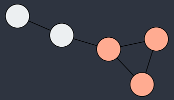




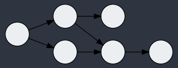
- For *rettede* snakker vi om *sterkt sammenhengende komponenter*
  - Dette dekkes nærmere i en senere forelesning

SYKLER

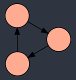
- Sykel: sti i en graf med minst tre noder, som forbinder første og siste node



- Grafer uten sykler kalles *asykliske*

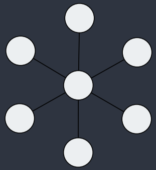


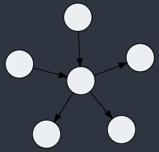
- I rettede grafer må kantene som utgjør en sykel «peke i riktig retning»



Graden til en node

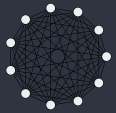
- Graden til en node, $\deg(v)$, beskriver hvor mange kanter v er koblet sammen med
  - I den øverste grafen til høyere har den midterste noden grad 5
  - De resterende nodene har grad 1
- I en rettet graf skiller man mellom *inn*- og *ut*-grad
  - I den nederste grafen til høyere har den midterste noden 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
  - 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 $\mathcal{O}(\frac{|V|(|V|-1)}{2}) = \mathcal{O}(\frac{|V|\cdot|V|-|V|}{2}) = \mathcal{O}(\frac{|V|^2-|V|}{2}) = \mathcal{O}(\frac{|V|^2}{2}) = \mathcal{O}(|V|^2)$
- Kjøretidskompleksiteten av 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)



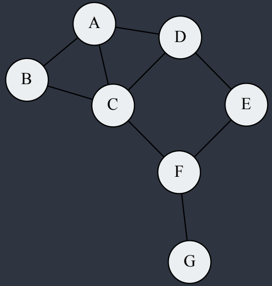
Representasjon

Representasjon av grafer

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

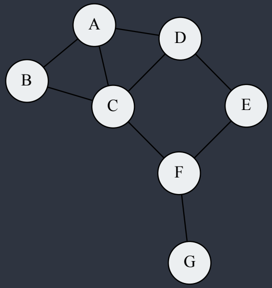
$$
\begin{array}{c}
 \\ A \\ B \\ C \\ D \\ E \\ F \\ G
\end{array}
\begin{array}{c}
\begin{array}{ccccccc} A & B & C & D & E & F & G \end{array} \\
\begin{pmatrix}
0 & 1 & 1 & 1 & 0 & 0 & 0 \\
1 & 0 & 1 & 0 & 0 & 0 & 0 \\
1 & 1 & 0 & 1 & 0 & 0 & 0 \\
1 & 0 & 1 & 0 & 1 & 0 & 0 \\
0 & 0 & 0 & 1 & 0 & 1 & 0 \\
0 & 0 & 0 & 0 & 1 & 0 & 1 \\
0 & 0 & 0 & 0 & 0 & 1 & 0
\end{pmatrix}
\end{array}
$$



- Nabomatriser egner seg for tette grafer
  - Minnebruken er i $\mathcal{O}(|V|^2)$
- De er enkle å implementere
  - … hvis du kan 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

$$
\begin{aligned}
A &: [B, C, D] \\
B &: [A, C] \\
C &: [A, B, D] \\
D &: [A, C, E] \\
E &: [D, F] \\
F &: [C, E, G] \\
G &: [F]
\end{aligned}
$$



- Egner seg for tynne grafer
  - Minnebruken er i $\mathcal{O}(|V| + |E|)$
- Er *ganske* enkle å implementere
  - …hvis du har tilgang på raske hashmaps og hashsets
- Egner seg når man trenger å finne *alle* naboer av en node (krever $\mathcal{O}(deg(v))$ tid for en node v)
- Kan sjekke om to noder er naboer i konstant tid

GRAFTRAVERSERING

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
  - Når den ikke kan finne en *ubesøkt* node sporer den stien tilbake
  - I hvert ledd forsøker den å følge nye noder som leder vekk fra noden den er i
- Dybde-først søk er som regel den 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 (rekursiv)

Algorithm: Rekursivt dybde-først søk

Input: En graf $G = (V, E)$ og en startnode u

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

```
Procedure DFSVisit(G, u, visited)
    add u to visited
    for (u, v) ∈ E do
        if v ∉ visited then
            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

```
Procedure DFSFull(G)
    visited ← empty set
    for v ∈ V do
        if v ∉ visited then
            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 $G = (V, E)$ og en startnode s

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

```
Procedure DFSVisit(G, s, visited)
    stack ← singleton stack containing s
    while stack is not empty do
        u ← stack.pop()
        if u ∉ visited then
            add u to visited
            for (u, v) ∈ E do
                stack.push(v)
```

Algorithm: Fullt iterativt dybde-først søk

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

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

```
Procedure DFSFull(G)
    visited ← empty set
    for v ∈ V do
        if v ∉ visited then
            DFSVisit(G, v, visited)
```

- 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
  - 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 (IMPLEMENTASJON)

ALGORITHM: BREDDE-FØRST SØK

Input: En graf $G = (V, E)$ og en startnode s
Output: Besøker alle noder i G som kan nås fra s nøyaktig én gang

```
Procedure BFSVisit(G, s, visited)
    queue ← singleton queue containing s
    while queue is not empty do
        u ← queue.dequeue()
        for (u, v) ∈ E do
            if v ∉ visited then
                add v to visited
                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

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

- I `BFSVisit` vil hver node vil legge alle sine noder på på køen
    - Merk at hver node kan legges på køen flere ganger
    - Men *ikke* flere ganger enn den har naboer
    - Det gir $\mathcal{O}(|E|)$
- I tillegg går `BFSFull` gjennom alle nodene
- Den totale kjøretiden blir $\mathcal{O}(|V| + |E|)$

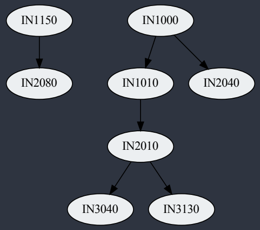
Topologisk sortering

TOPOLOGISK SORTERING

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

```
Procedure TopSort(G)
    stack ← empty stack
    output ← empty list
    for v ∈ V do
        if indegree of v is 0 then
            stack.push(v)
    while stack is not empty do
        u ← stack.pop()
        append u to output
        for (u, v) ∈ E do
            remove u from the incoming edges of v
            if indegree of v is 0 then
                add v to stack
    if |output| < |V| then
        error G contains a cycle and cannot be topologically ordered
    return output
```

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

```
Procedure DFSTopSort(G)
    stack ← empty stack
    visited ← empty set
    for u ∈ V do
        if u ∉ visited then
            DFSVisit(G, u, visited,
             stack)
    return stack

Procedure DFSVisit(G, u, visited, stack)
    add u to visited
    for (u, v) ∈ E do
        if v ∉ visited then
            DFSVisit(G, v, visited, stack)
    stack.push(u)
```