

Eksamensforelesning algdat 2020

Del 1

Dagens program

- Øystein (Del 1)

- Algoritmer, analyse og Asymptotisk notasjon
- Grafer og Graftraversering
- Grådighet
- Minimale spenntrær
- Korteste vei

- Theodor (Del 2)

- Sortering
- Datastrukturer
- Dynamisk programmering (DP)
- Maks flyt
- NP

Formål og tilnærming

- Hjemmeeksamen: ny form
- Fokus på pensum og forståelse heller enn mekanisk løsning av gamle eksamensoppgaver
- Eksamenseksemppler der det tydeliggjør prinsipp
- Spør!

Algoritmer, analyse og asymptotisk notasjon

- Formell definisjon av et problem
- Datamaskinmodellen brukt i faget
- Dekomponering av et problem
- Asymptotisk notasjon
- Å analysere algoritmer

Formell definisjon av et problem

Et **beregningsproblem** består av en **input** og en **output**, regler for hva som er gyldig input og output, samt en generell beskrivelse av hvilken **output** vi ønsker å få gitt en spesifikk **input**.

En **algoritme** er en samling med klart definerte steg som tar oss fra en input til en gyldig output

En konkret input til et beregningsproblem kalles en **instans** av dette problemet.

Datamaskinmodellen brukt i faget

Faget bruker datamaskinmodellen som er definert i læreboka. Denne modellen er en maskin med **kun en prosessor** og et arbeidsminne, som kan utføre enkle aritmetiske operasjoner og minneaksess i konstant tid.

Maskinen modellerer **ikke** minnehierarki og caches og den kan naturligvis heller ikke modellere parallell utførelse.

Dekomponering av et problem

Iterasjon

Splitt og hersk

Dynamisk programmering

Grådighet

Asymptotisk notasjon

Asymptotisk notasjon beskriver hvordan en funksjon oppfører seg når inputstørrelsen blir veldig stor. I algoritmesammenheng er funksjonen ofte **tidsbruk gitt en inputstørrelse**.

Asymptotisk notasjon gir oss ikke en presis beskrivelse av veksten til en funksjon, men den gir oss øvre og nedre grenser.

Asymptotisk notasjon gjør det enklere å beskrive og sammenligne ulike algoritmer

Asymptotisk notasjon - matematisk definisjon

$$\Theta(g(n)) = \{f(n) : \exists c_1, c_2, n_0, 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n), \forall n \geq n_0\}$$

$$O(g(n)) = \{f(n) : \exists c, n_0, 0 \leq f(n) \leq c g(n), \forall n \geq n_0\}$$

$$\Omega(g(n)) = \{f(n) : \exists c, n_0, 0 \leq c g(n) \leq f(n), \forall n \geq n_0\}$$

$$o(g(n)) = \{f(n) : \forall c \exists n_0, 0 \leq f(n) < c g(n), \forall n \geq n_0\}$$

$$\omega(g(n)) = \{f(n) : \forall c \exists n_0, 0 \leq c g(n) < f(n), \forall n \geq n_0\}$$

Asymptotisk notasjon - eksempel

- $f(n) = \Theta(g(n))$

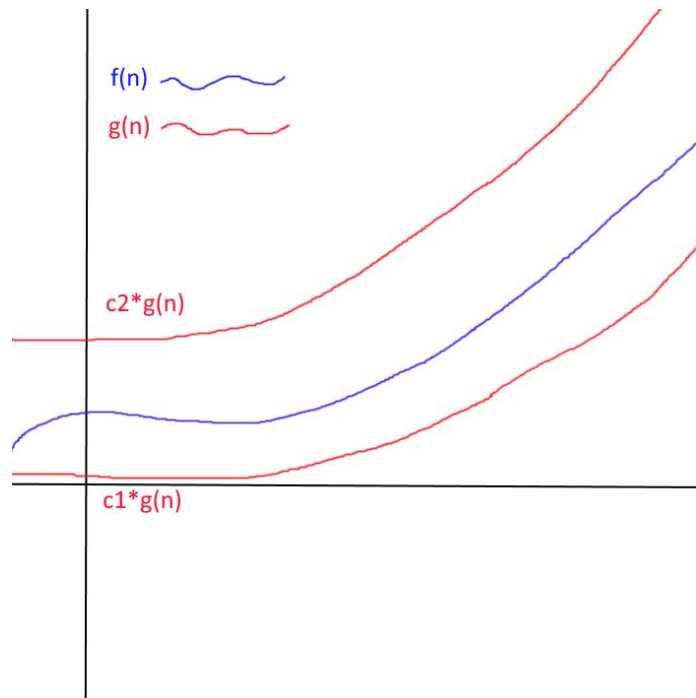
$$\Theta(g(n)) = f(n) : \quad =$$

$$O(g(n)) = f(n) : \quad \geq$$

$$\Omega(g(n)) = f(n) : \quad \leq$$

$$o(g(n)) = f(n) : \quad >$$

$$\omega(g(n)) = f(n) : \quad <$$



Asymptotisk notasjon - oppgave 1

Basert på Hetlands FAQ:

<https://algsat.idi.ntnu.no/faq/2018/11/05/asymptotisk-notasjon-og-beste-verste-tilfellet.html>

Vi vet fra pensum at insertion sort har kjøretid på $O(n^2)$, altså at vi alltid kan finne et tall c å gange med n^2 slik at $c \cdot n^2$ til slutt alltid vil være høyere enn $T(n)$ for insertion sort. Hvorfor er ikke insertion sort også $\theta(n^2)$?

Asymptotisk notasjon - oppgave 1

Basert på Hetlands FAQ:

<https://algsdat.idi.ntnu.no/faq/2018/11/05/asymptotisk-notasjon-og-beste-verste-tilfelle.html>

Vi vet fra pensum at insertion sort har kjøretid på $O(n^2)$, altså at vi alltid kan finne et tall c å gange med n^2 slik at $c \cdot n^2$ til slutt alltid vil være høyere enn $T(n)$ for insertion sort. Hvorfor er ikke insertion sort også $\Theta(O(n^2))$?

Svar: fordi insertion sort sin kjøretid avhenger både av inputlengde, men også egenskaper til input (hvor sortert den allerede er). Best case er $\Omega(n)$, når input allerede er sortert. $\Theta(n^2)$ gir oss mengden av alle funksjoner som er begrensa av n^2 over og under, og da faller de tilfellene der $T(n)$ er lineære utenfor

Asymptotisk notasjon - obs 1

Husk at de forskjellige typene asymptotisk notasjon ikke sier noe i seg selv om det er snakk om best-case, worst-case eller average case

Igjen, les Hetlands oppklaring:

<https://algsdat.idi.ntnu.no/faq/2018/11/05/asymptotisk-notasjon-og-beste-veste-tilfelle.html>

Asymptotisk notasjon - obs 2

Når vi sier at $f(n) = O(g(n))$ og ikke $f(n) \in O(g(n))$ er dette **misbruk av notasjon**. Stort sett kan man bruke dette likhetstegnet som vanlig, men det er en del spesialregler. En av de viktigste er: **I ligninger med asymptotisk notasjon kan ikke høyre side være mer presis enn venstre**

Hetlands oppklaring:

<https://algdatt.idi.ntnu.no/faq/2020/05/27/likhet-og-asymptotisk-notasjon.html>

Å analysere algoritmer

Når vi analyserer algoritmer er vi gjerne på jakt etter 2 ting. **Korrekthet** og **Ressursbruk**

Korrekthet viser vi gjerne gjennom induksjon

Ressursbruk ønsker vi gjerne å kunne beskrive ved hjelp av asymptotisk notasjon

Å analysere algoritmer - Løkkeinvarianter og induksjon

En **løkkeinvariant** er noe som bestandig er sant om en løkke i en algoritme. For å sjekke om en påstand er en løkkeinvariant så må vi sjekke at den gjelder **før** løkka starter, **etter hver iterasjon** og **når løkka er ferdig**.

Dette er analogt til induksjon der man først sjekker at påstanden holder for grunntilfellet, og deretter for alle steg som følger.

Tilsvarende induktive teknikker lar oss vise at rekursive dekomponeringer er korrekte.

Å analysere algoritmer - input og kjøretid

Vi ønsker ofte å kunne si noe om kjøretiden til en algoritme gitt en viss input.

Oftest er det **kjøretid som funksjon av inputstørrelsen** vi er interesserte i. Det kan være enten **antall elementer** i input eller **antall bits** inputen trenger for å representeres.

Av og til er det ikke bare input-størrelsen men også egenskapene til input som styrer hvor lang tid algoritmen trenger. I dette tilfellet så er det ofte nyttig å snakke om **kjøretid i beste og i verste fall**

Å analysere algoritmer - input og kjøretid: eksempel

Insertion sort består av to løkker. Den ytterste går $(n-1)$ ganger, den innerste går maksimalt $(n-2)$ ganger.

Kjøretiden til alle mulige kjøring av insertion sort vil derfor være i $O(n^2)$

```
def insertionsort(A, len):  
    for j in range(1, len):  
        key = A[j]  
        i = j - 1  
        while(i >= 0 and A[i] > key):  
            A[i+1] = A[i]  
            i = i - 1  
        A[i+1] = key
```

Å analysere algoritmer - rekursiv kjøretid

I enkelte tilfeller, så er det praktisk å uttrykke kjøretiden som **et rekursivt uttrykk**

For å kunne for å kunne bruke disse i praksis ønsker vi å oversette den til et uttrykk kun avhengig av n

$$T(n) = T(n/2) + n$$

Dette kan gjøres ved hjelp av **iterasjonsmetoden**, **rekurrenstrær**, eller, **substitusjonsmetoden**

Å analysere algoritmer - iterasjonsmetoden

Ta et rekursivt uttrykk, for eksempel $T(n) = T(n/2) + 1$, $T(1) = 1$

Utvid opp til i iterasjoner: $T(n) = T(n/2) + 1 = T(n/4) + 2 = T(n/2^i) + i$

Vi ønsker nå å få argumentet til T lik 1, for å få grunntilfellet. $n/2^i = 1 \Rightarrow 2^i = n \Rightarrow i = \lg(n)$

Dette gir $T(n) = T(1) + \lg(n) = 1 + \lg(n)$

Å analysere algoritmer - et substitusjonsbevis

Steg 1: gjett en løsning

$$T(n) = T(n/2) + n \qquad T(n) \leq c * n \log(n)$$

Steg 2 anta at det holder for alle $m < n$:

$$n/2 = m < n$$

$$T(n/2) \leq c(n/2) * \log(n/2)$$

Steg 3: sett inn og substituer

$$T(n) \leq c(n/2) * \log(n/2) + n$$

$$\leq (cn/2) * \log(n) - (cn/2) * \log(2) + n$$

$$\leq (cn/2) * \log(n) - (cn/2) + n$$

$$\leq (cn/2) * \log(n)$$

$$\leq cn * \log(n)$$

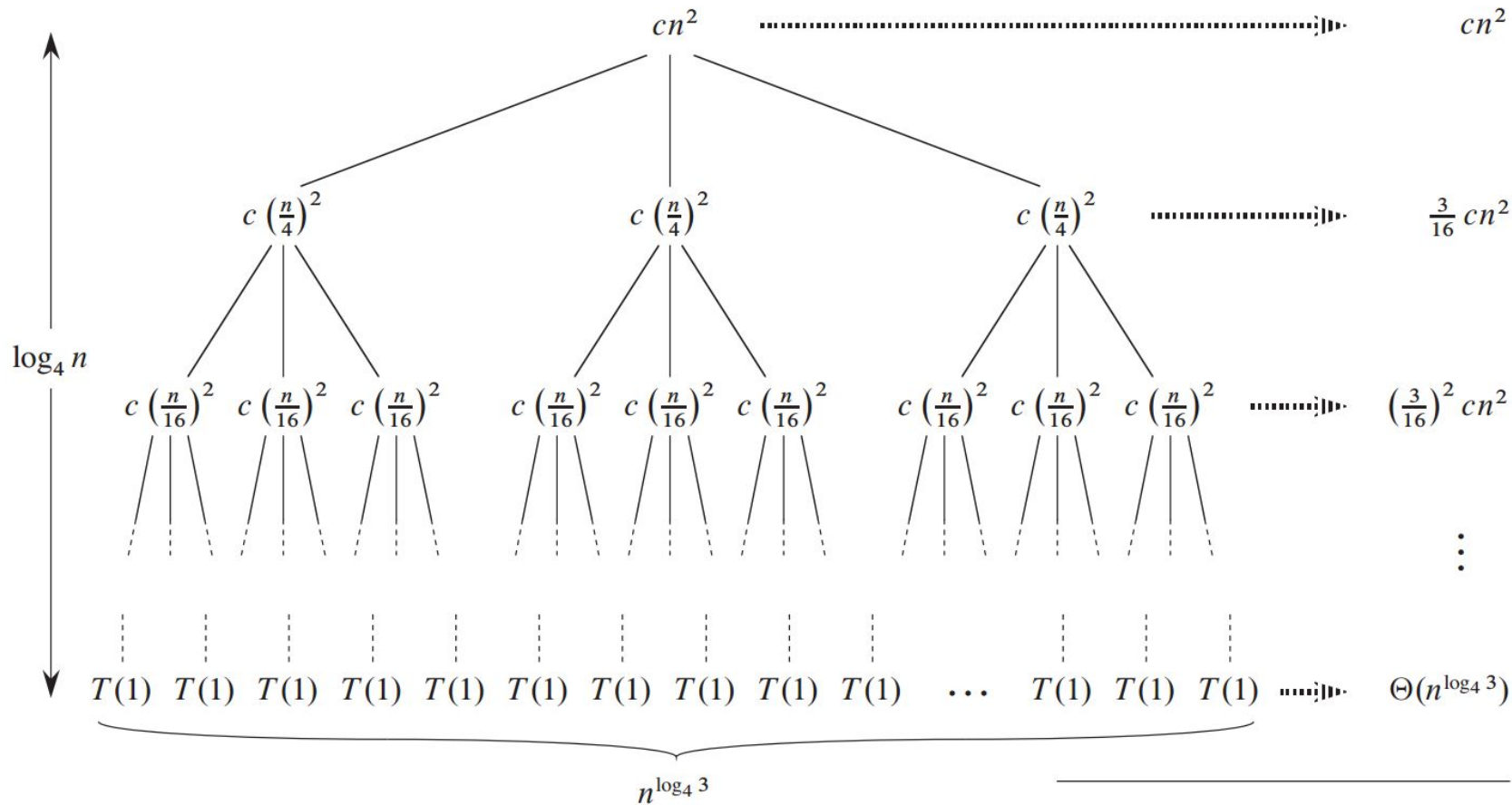
Steg 4: vis at det holder for ett grunntilfelle (vilkårlig valgt n)

Å analysere algoritmer - rekurenstrær

Tegn ut hvert rekursive kall som noder

Kan brukes som gjett til substitusjonsmetoden, eller som løsning i seg selv om man er nøye

Vi ser på treet til $T(n) = 3T(n/4) + cn^2$



Å analysere algoritmer - mastermetoden

Fungerer om vi har et uttrykk på formen $T(n) = a \cdot T(n/b) + f(n)$, der $a \geq 1$ og $b > 1$

Dekker mange tilfeller av rekursiv dekomponering.

Krever at betingelsene følges nøyaktig, kan være fort gjort å tro at en funksjon passer inn i masterteoremet uten at den faktisk gjør det.

Ä analysere algoritmer - masterteoremet

Theorem 4.1 (Master theorem)

Let $a \geq 1$ and $b > 1$ be constants, let $f(n)$ be a function, and let $T(n)$ be defined on the nonnegative integers by the recurrence

$$T(n) = aT(n/b) + f(n) ,$$

where we interpret n/b to mean either $\lfloor n/b \rfloor$ or $\lceil n/b \rceil$. Then $T(n)$ has the following asymptotic bounds:

1. If $f(n) = O(n^{\log_b a - \epsilon})$ for some constant $\epsilon > 0$, then $T(n) = \Theta(n^{\log_b a})$.
2. If $f(n) = \Theta(n^{\log_b a})$, then $T(n) = \Theta(n^{\log_b a} \lg n)$.
3. If $f(n) = \Omega(n^{\log_b a + \epsilon})$ for some constant $\epsilon > 0$, and if $af(n/b) \leq cf(n)$ for some constant $c < 1$ and all sufficiently large n , then $T(n) = \Theta(f(n))$. ■

Å analysere algoritmer - masterteoremet: eksempel

$$T(n) = 27T(n/3) + n^3$$

$$a = 27, b = 3$$

$$f(n) = n^3, n^{\log_b a} = n^{\log_3 27} = n^3$$

$$f(n) = n^3 = \Theta(n^3)$$

$$t(n) = n^{\log_b a} * \log(n) = n^3 * \log(n)$$

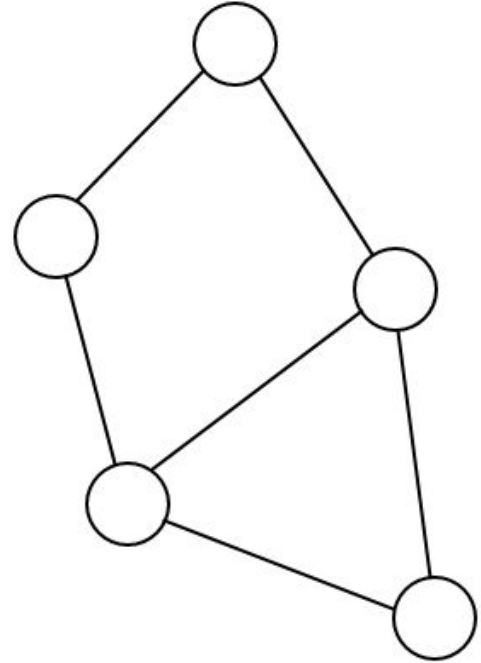
Å analysere algoritmer - masterteoremet: fallgruver

- $f(n)$ er ikke asymptotisk større
- $f(n)$ er ikke asymptotisk mindre
- regulartietsbetingelsen holder ikke

Grafteraversering

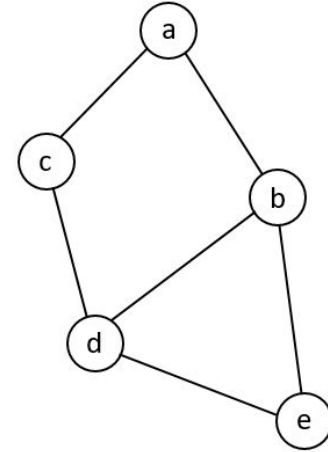
Hva er en graf?

- Samling av noder og kanter
- Retta eller uretta kanter
- Kan representeres ved nabomatriser eller nabolister



Nabomatrise

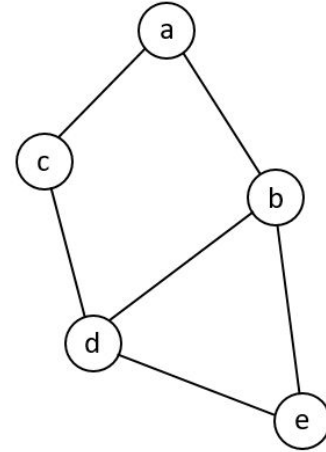
- Leses fra rad til kolonne
- Rask å slå opp en spesifikk kant
- Mindre praktisk for traversering



	a	b	c	d	e
a	0	1	1	0	0
b	1	0	0	1	1
c	1	0	0	1	0
d	0	1	1	0	1
e	0	1	0	1	0

Naboliste

- Hver node har en liste med naboer
- Ikke mer informasjon enn nødvendig, god for traversering
- Å sjekke om en kant finnes krever et lineærsøk



$a \rightarrow [b, c]$

$b \rightarrow [a, d, e]$

$c \rightarrow [a, d]$

$d \rightarrow [c, b, e]$

$e \rightarrow [d, b]$

Traversering av en graf

- Vi ønsker å søke gjennom nodene i en graf
- Felles: Vi starter i en node, og velger nye noder å traversere blant naboene
- Hvilke noder velger vi?
- Hvordan holder vi styr på hvilke noder som allerede er traversert?

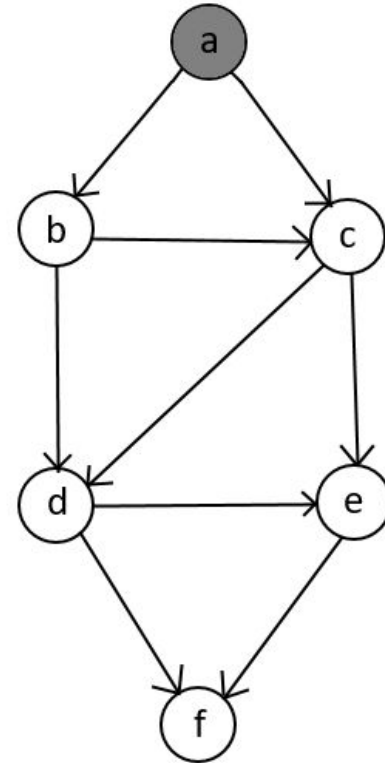
Bredde først (BFS)

- Vi gir alle nodene en farge, og starter med å farge alle hvite.
 - Nodene organiseres i en **kø**.
 - Noder i køen er grå
 - Noder som er tatt ut av køen er svarte
-
- Når alle naboene er lagt til i køen tar vi den første ut, og inspiserer naboene til den neste i køen

BFS steg 1

Kø = [a] (vi legger inn til høyre, og tar ut til venstre)

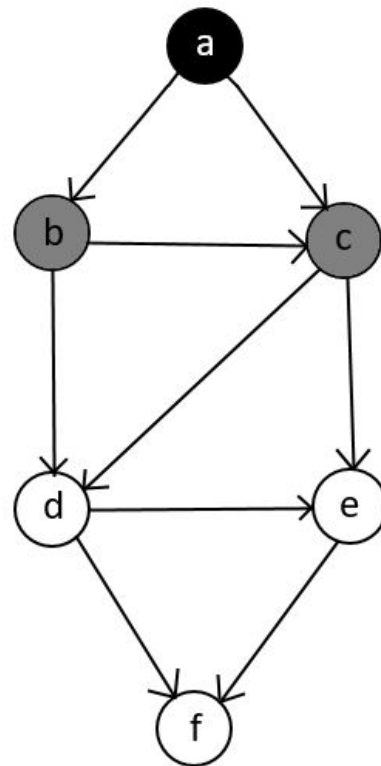
Ferdig = Ø



BFS steg 2

Kø = [b, c]

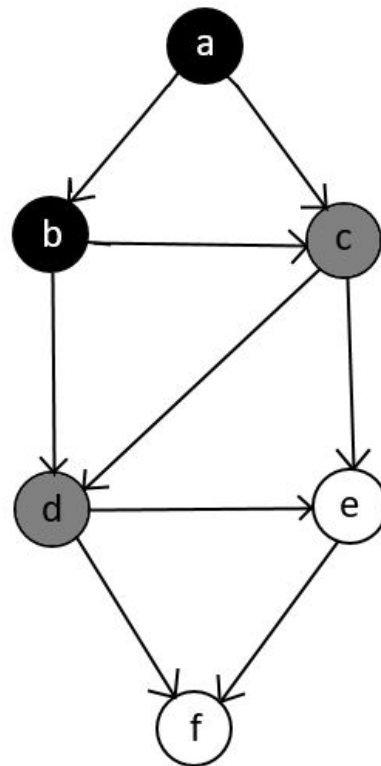
Ferdig = {a}



BFS steg 3

Kø = [c, d]

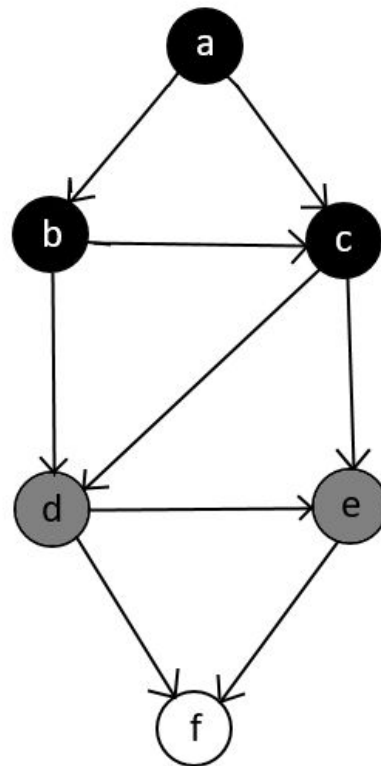
Ferdig = {a, b}



BFS steg 4

Kø = [d, e]

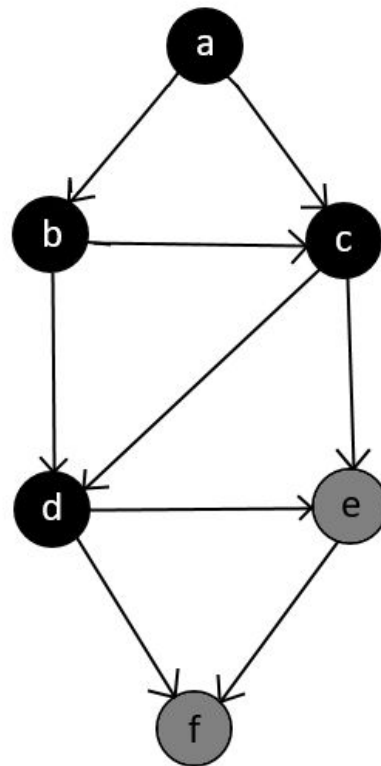
Ferdig = {a, b, c}



BFS steg 5

Kø = [e, f]

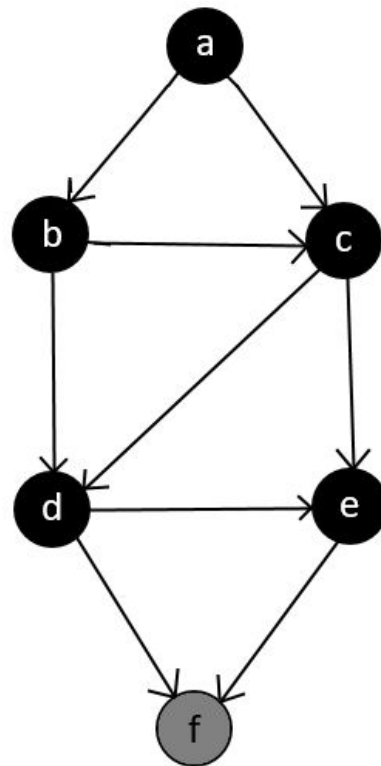
Ferdig = {a, b, c, d}



BFS steg 6

Kø = [f]

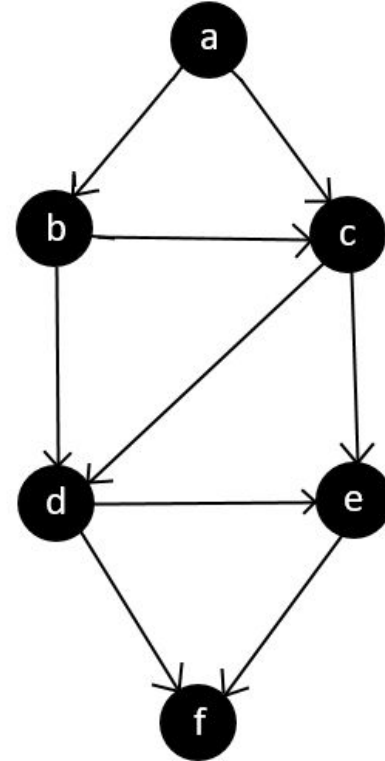
Ferdig = {a, b, c, d, e}



BFS ferdig

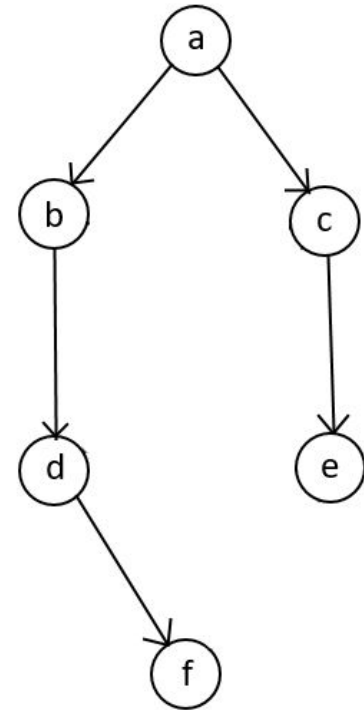
Kø = []

Ferdig = {a, b, c, d, e, f}



Kjøretid og anvendelser BFS

- $O(V + E)$
- Mulig å lagre forgjengere for å beholde traverseringstreet
- Gir korteste vei fra startnode til alle andre nåbare noder
- Delalgoritme i andre algoritmer
 - Edmonds-Karp
 - Dijkstra



Dybde først (DFS)

- Vi gir alle nodene en farge, og starter med å farge alle hvite.
- Nodene organiseres i en **stakk**.
- Noder i stakken er grå
- Noder som er tatt ut av stakken er svarte
- I tillegg tidsstempler vi en node når den legges til i stakken, og når den tas ut
- Cormen velger å utforske eventuelle manglende noder når et DFS-tre er ferdig, dette er et implementasjonsvalg (men standard i følge pensum!)

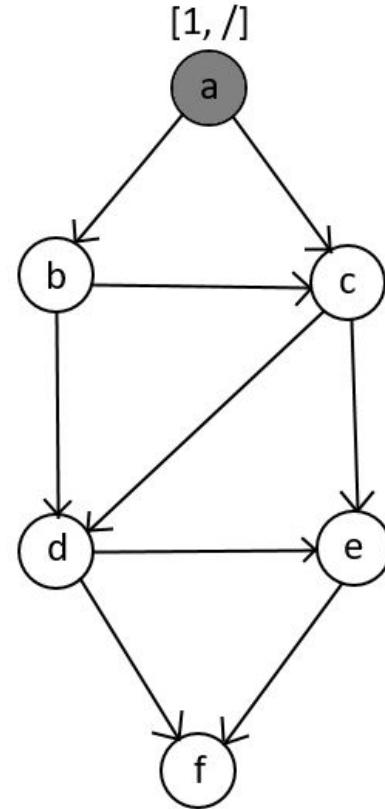
DFS

Stakk = [a]

(Stakken vokser mot venstre)

Ferdig = []

(Vi plasserer en node bakerst
når den er ferdig)



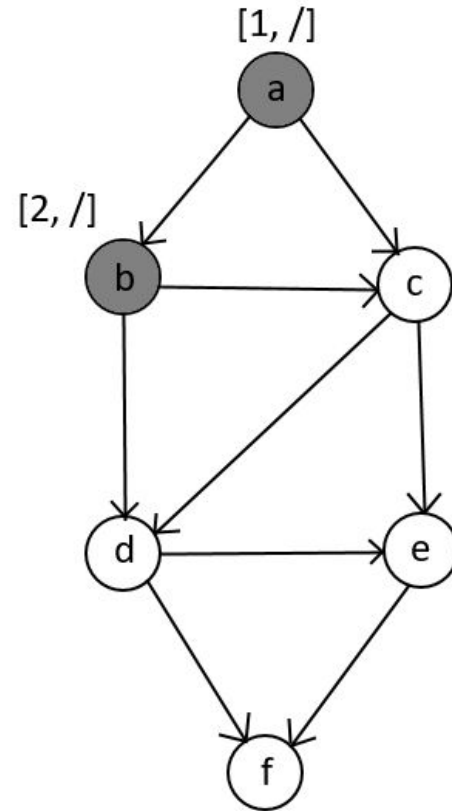
DFS

Stakk = [a, b]

(Stakken vokser mot venstre)

Ferdig = []

(Vi plasserer en node bakerst
når den er ferdig)



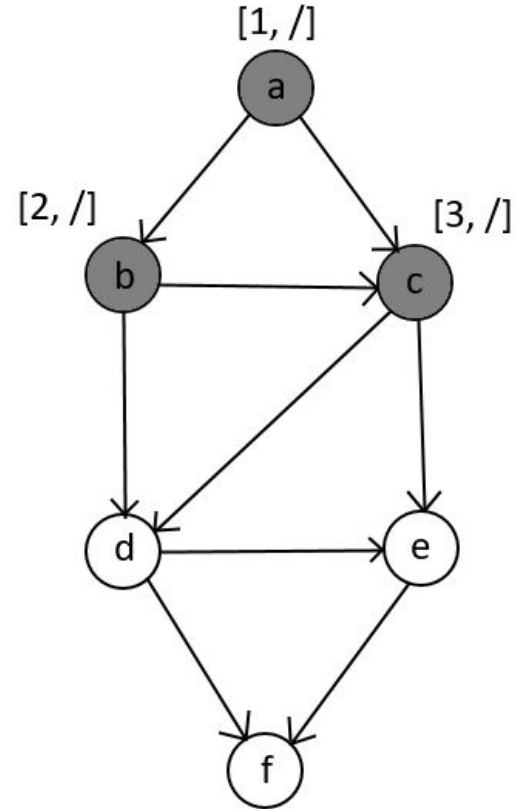
DFS

Stakk = [a, b, c]

(Stakken vokser mot venstre)

Ferdig = []

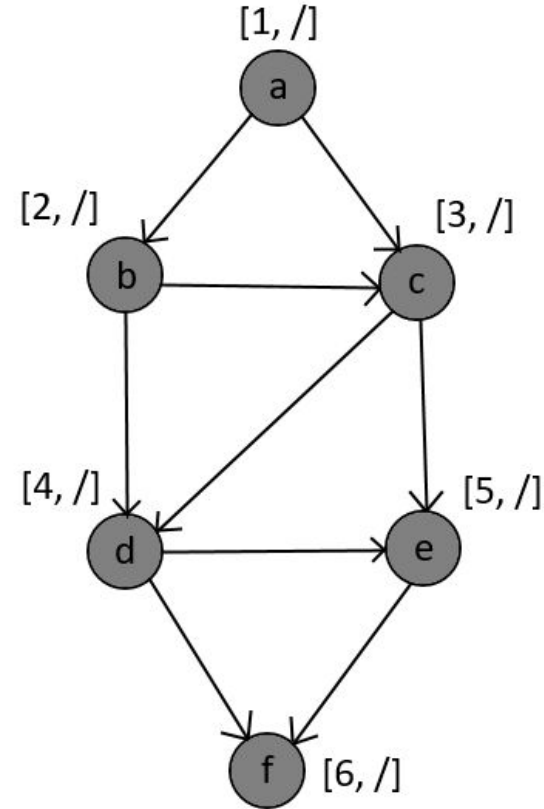
(Vi plasserer en node bakerst
når den er ferdig)



DFS

Stakk = [a, b, c, d, e, f]
(Stakken vokser mot venstre)

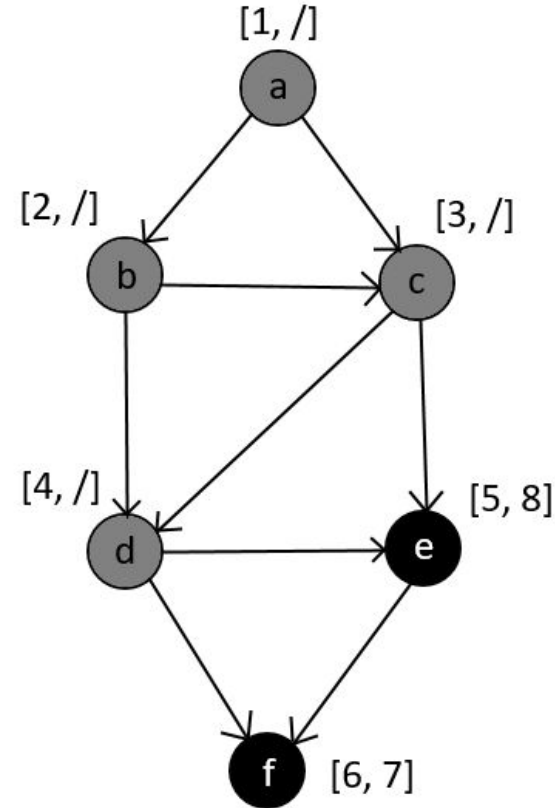
Ferdig = []
(Vi plasserer en node bakerst når den er ferdig)



DFS

Stakk = [a,b ,c ,d]
(Stakken vokser mot venstre)

Ferdig = [e, f]
(Vi plasserer en node bakerst når den er ferdig)



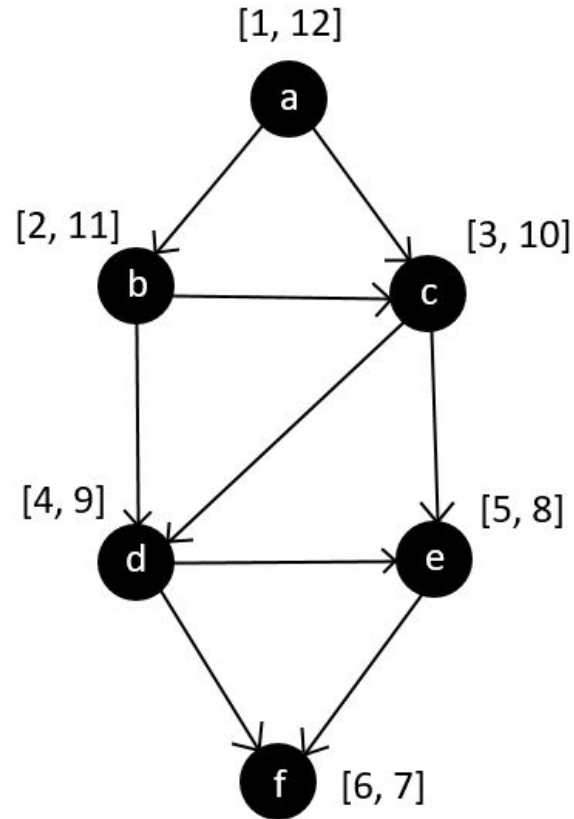
DFS ferdig

Stakk = []

(Stakken vokser mot venstre)

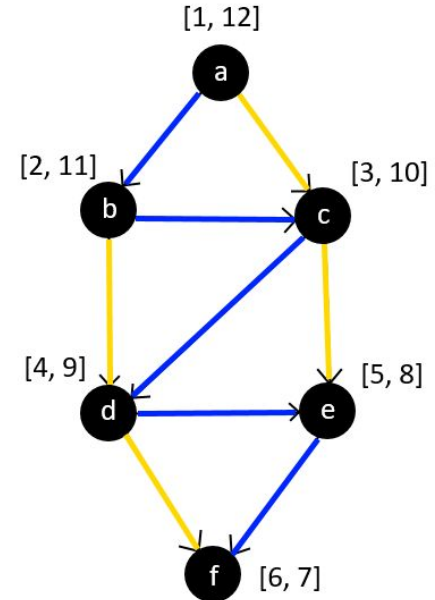
Ferdig = [a, b, c, d, e, f]

(Vi plasserer en node bakerst når den er ferdig)



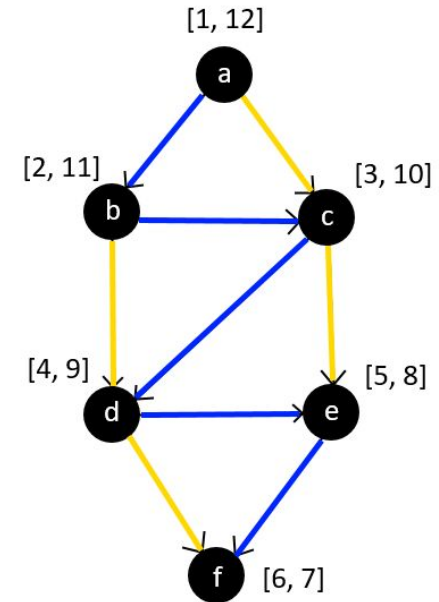
Kjøretid og anvendelser DFS

- $O(V + E)$
- Mulig å lagre forgjengere for å beholde traverseringstreet
- Om grafen vår er en DAG får vi en topologisk sortering(etter sluttid)
- Kantklassifisering:
 - Forover (markert gule)
 - Bakover (ingen i eksempel)
 - Tre (markert blå)
 - Kryss (ingen i eksempel)



Topologisk sortering

- Vi vet at grafen vår er en rettet, asyklisk graf (DAG), ettersom kantklassifiseringen ikke returnerte noen bakoverkanter
- Om vi sorterer nodene etter synkende sluttid får vi en topologisk sortering
 - $[a, b, c, d, e, f]$
- Denne sorteringen av en DAG åpner for en effektiv korteste-vei-algoritme
 - Vi kommer tilbake til denne senere



Grådighet

Grådighet

Mange algoritmer består av å gjøre en serie med valg. I enkelte tilfeller vil det å gjøre det valget som ser best ut her og nå gi en optimal løsning. De problemene dette fungerer for har **grådighetsegenskapen**

Grådige algoritmer er ofte enkle å implementere, men det kan være utfordrende å vite for hvilke problemer de gir et optimalt resultat. Mange problemer som kan løses med dynamisk programmering kan også løses med en grådig algoritme, men ikke alle.

Viktige eksempler fra pensum er **aktivitet-utvelgelse**, **kontinuerlig ryggsekk-problemet** og **huffmankoding**

Hva skal til for at et problem skal kunne løses grådig?

Om et problem skal kunne løses optimalt av en grådig algoritme må det ha **optimal substruktur**. Dette betyr at en hver optimal løsning består av optimale delløsninger.

Vi må være garantert at det grådige valget gir oss et element som hører til i en optimal løsning; Det grådige valget må velge et element som en annen optimal algoritme også ville ha valgt. Det er dette som er selve **grådighetsegenskapen**.

Hvert valg må kun gi ett nytt delproblem.

Eksempel: huffman-koder

Huffmankoder er en måte å kode data som består av tegn på slik at den tar minst mulig plass.

Selve kodingen defineres av et **Huffman-tre**, som gir informasjonen som trengs for å kode en streng med data, og for å dekode den igjen.

Selve treet konstrueres grådig basert på frekvensen til hvert tegn i inputdataen.

AAABCDACCB

A	B	C	D
4	2	3	1

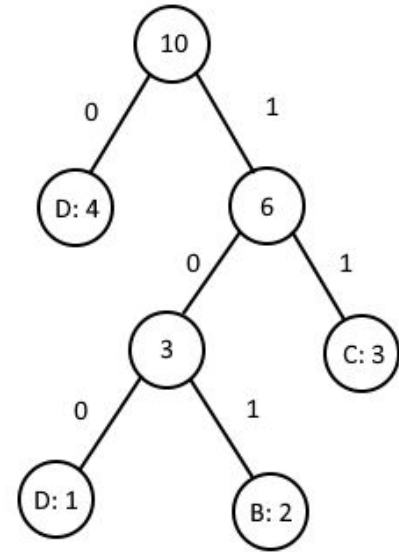
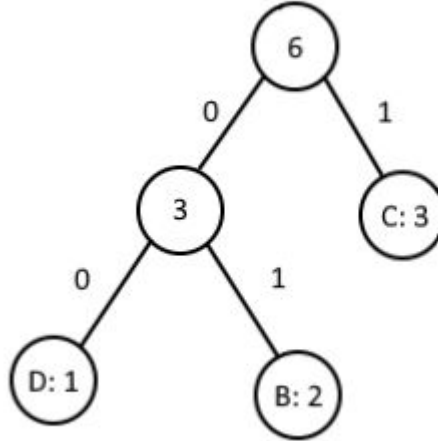
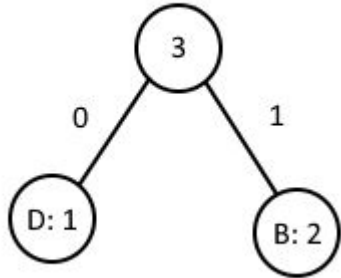
Eksempel: huffman-koder

$O(n \lg n)$ med minheap

AAABCDACCB ->

0 0 0 101 11 100 0 11 11 1 0 1

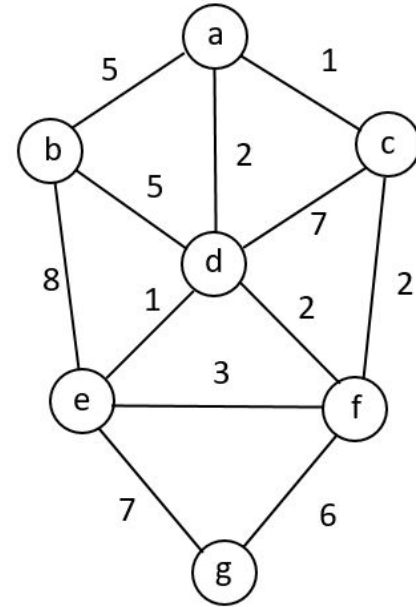
5% reduksjon vs 2-bit



Minimale spenntrær

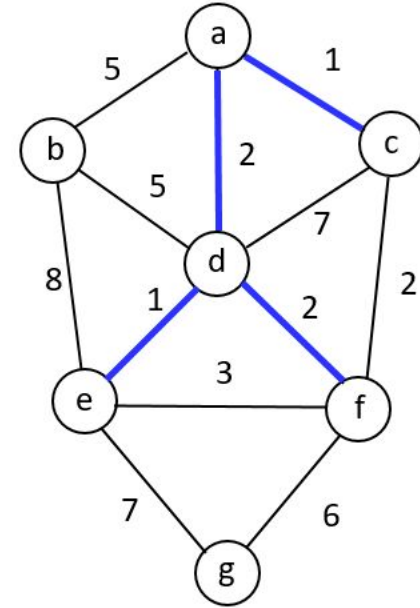
Problemet

- Vi har en urettet graf med n noder, der hver kant har en kostnad forbundet ved seg
- Denne kostnaden er gitt ved en vektfunksjon $w(u, v)$, som gir alle kanter (u, v) en tallverdi
- Mål: Koble alle nodene sammen med billigst mulig tre



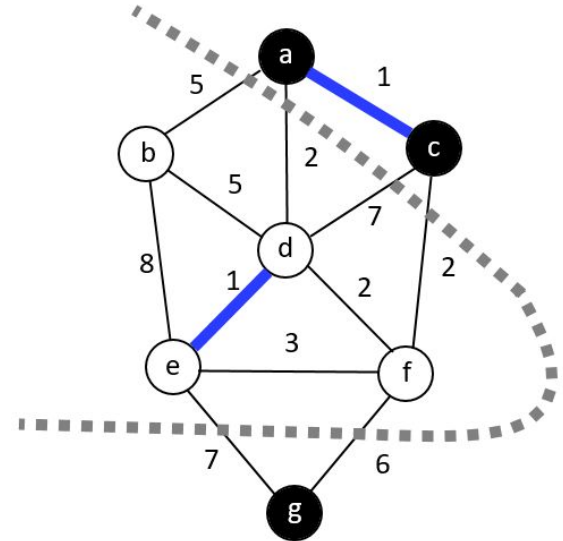
Generisk MST

- Grådig løsning: plukk med den billigste trygge kanten
- En kant er trygg hvis den kan legges til løsningsmengden slik at:
 - Vi fortsatt har et tre
 - Resultatet er en delmengde av et minimalt spennetre
- I grafen til høyre er f.eks (b,d) trygg, mens (f, c) ikke er det



Lette kanter

- Om vi deler grafen i to slik at snittet ikke krysser noen kanter som allerede er med i løsningen, så får vi en enkel definisjon på hvilken kant vi nå skal ta med oss
- Denne kanten kalles for en “lett kant”, og er den kanten med lavest vekt som krysser snittet
- Begge de følgende algoritmene finner denne lette kanten i hver iterasjon

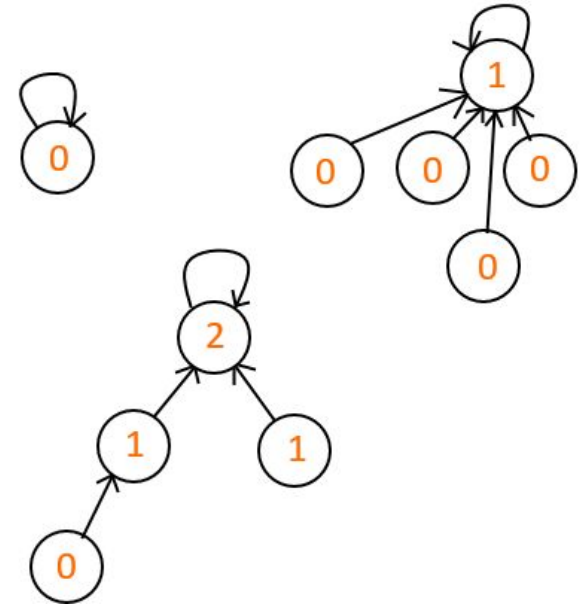


Nyttig hjelpestruktur: disjoint set-strukturen

- Datastruktur med operasjonene:
 - Make-set(x): Lag en ny mengde som inneholder x
 - Union(x, y): Slå sammen de to mengdene til x og y
 - Find-set(x): Finn representanten til x
- Formål:
 - Å kunne fordele elementer i forskjellige mengder
 - Å kunne teste om to elementer er i samme mengde
 - Å kunne slå sammen to mengder

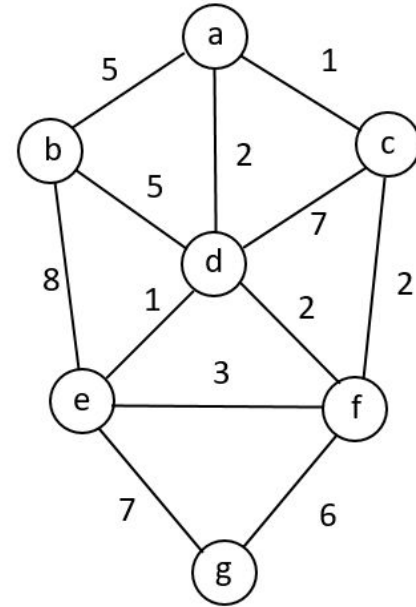
Disjoint set-skog

- Representerer mengdene som trær, der rotnoden representerer mengden
- Hver gang find-set(x) kjøres, så komprimeres stien fra noden x til rotnoden
- Hvert node har et estimat på hvor dypt det er. Når to mengder slås sammen legges det grunneste treet til som barn av det dypeste
- Veldig god kjøretid (invers Ackermann). Praktisk talt konstant

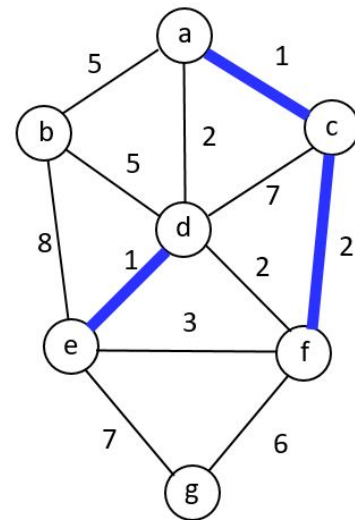
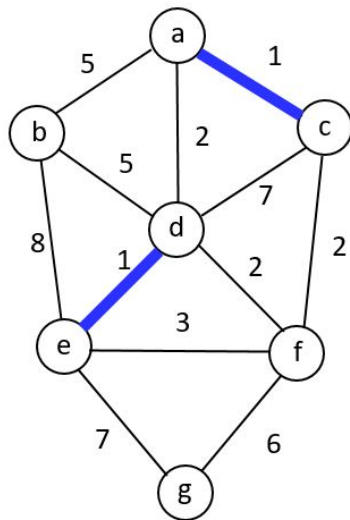
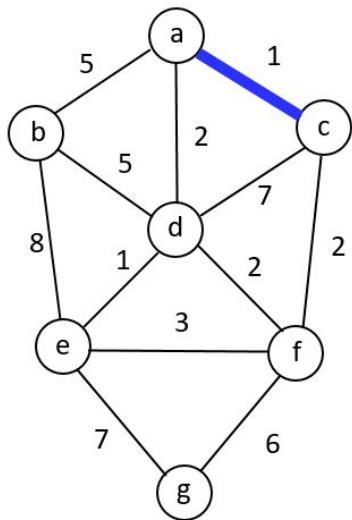


Kruskals algoritme

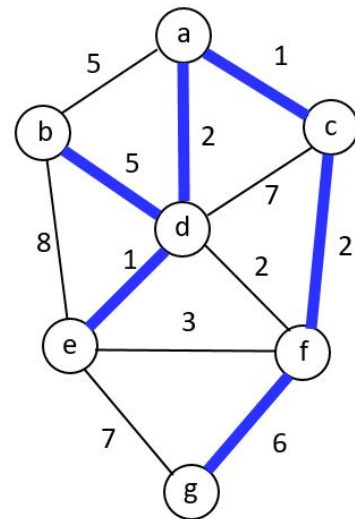
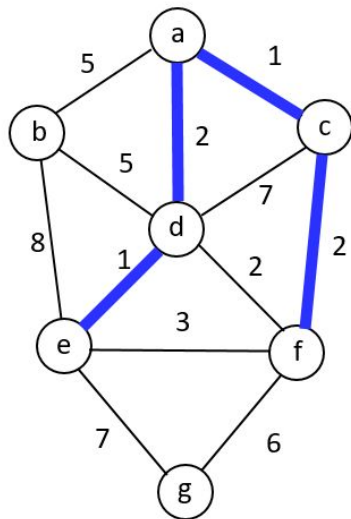
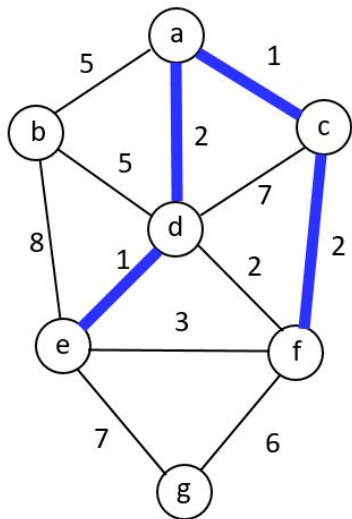
- Finner en trygg kant ved å ta den letteste kanten som kobler sammen to trær i skogen av allerede valgte kanter
- Holder styr på skogen vha disjoint-set
- Når det ikke lenger er flere trær å koble sammen betyr det at det kun er ett igjen som spenner grafen, og dette er det minimale spenntreet
- $O(E \lg V)$ (med min-heap)



Kruskals algoritme

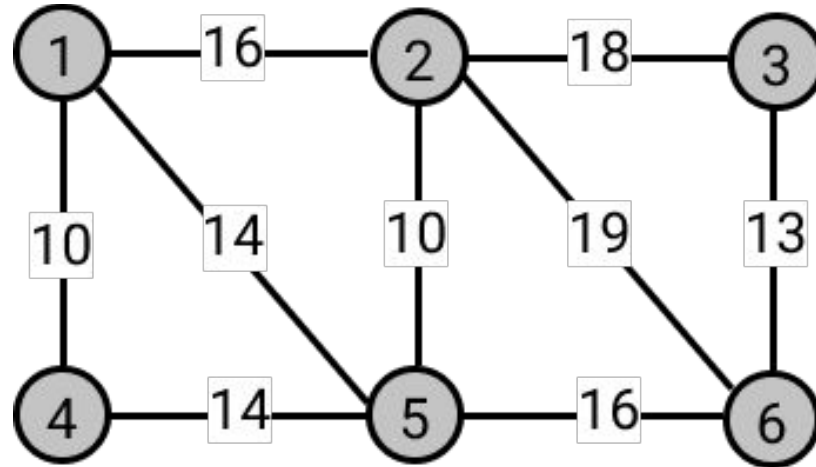


Kruskals algoritme



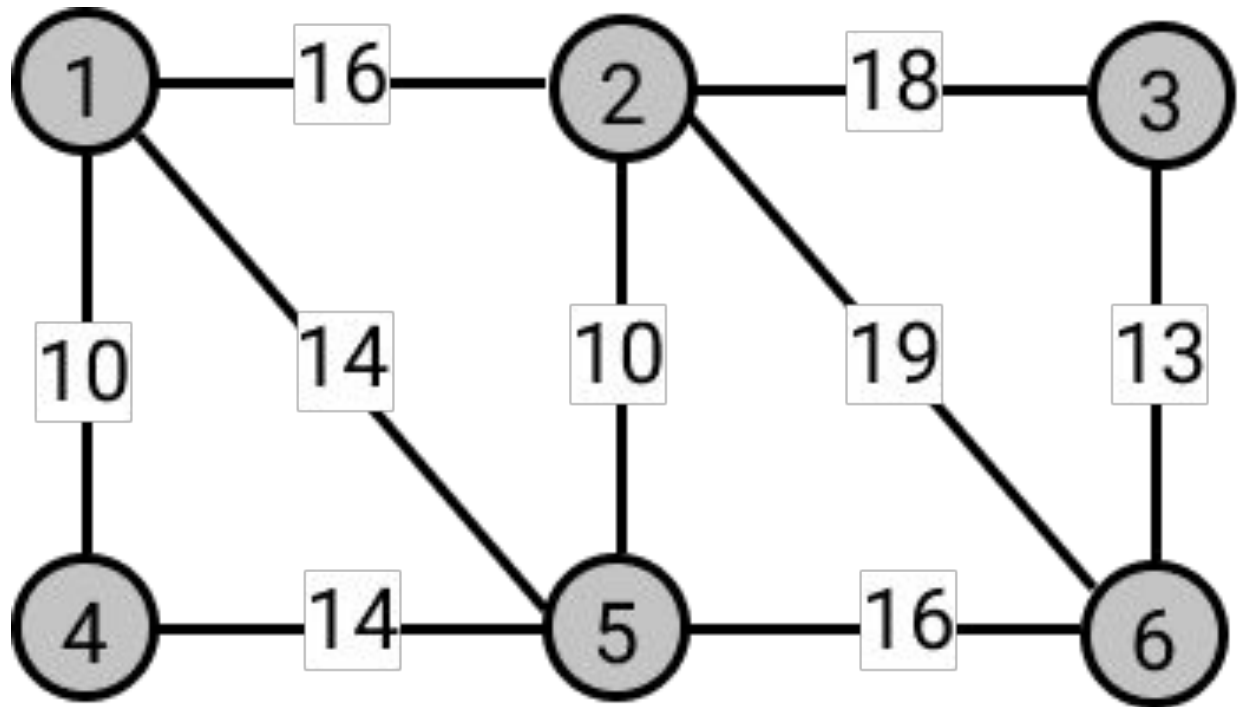
2016H - Oppgave 14

- Hvis du utfører MST-Kruskal på grafen i figur 1, hvilken kant vil velges som den femte i rekken? Det vil si, hvilken kant vil være den femte som legges til i løsningen? Oppgi kanten på formen (i, j) , der $i < j$.



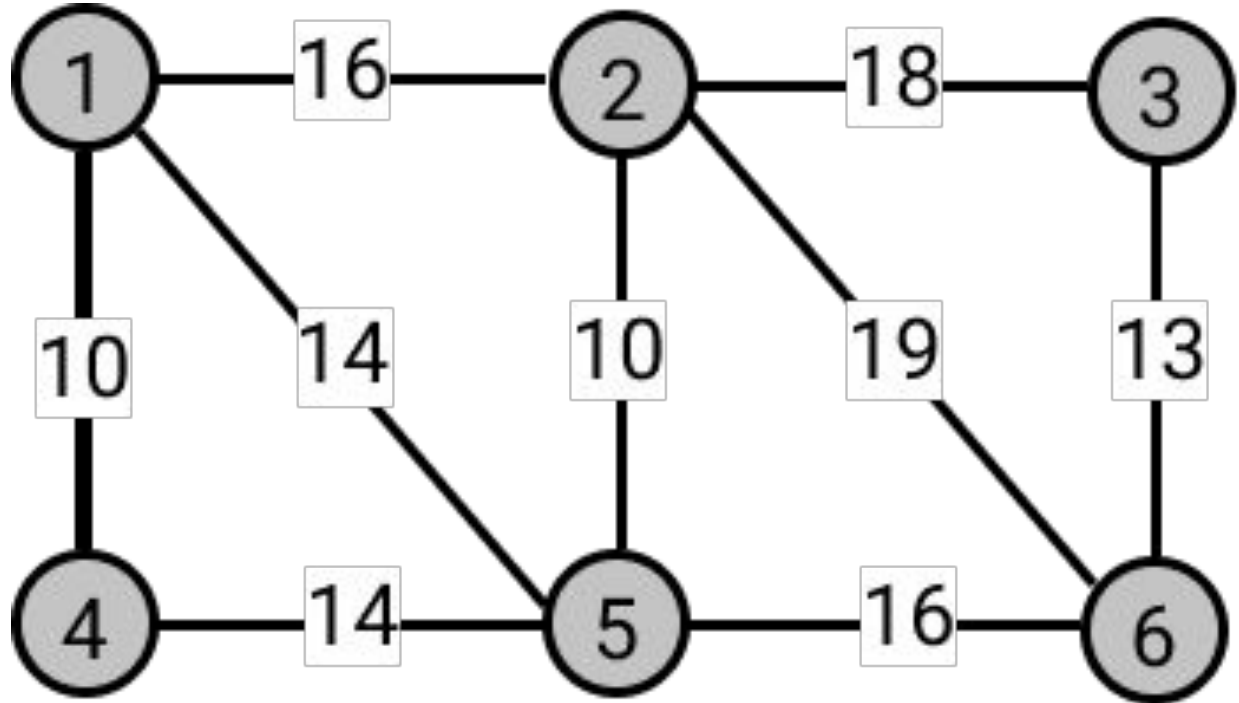
2016H - Oppgave 14

$lf = \{\}$



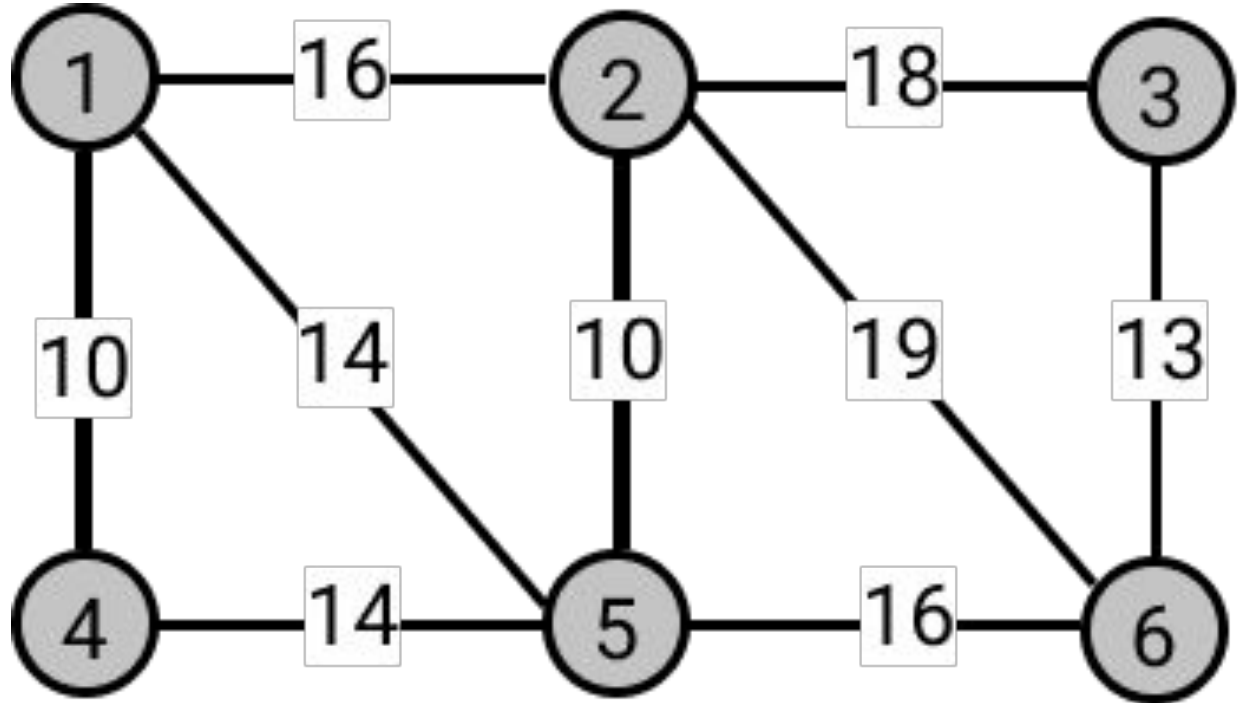
2016H - Oppgave 14

$lf = \{$
 $(1, 4)$
 $\}$



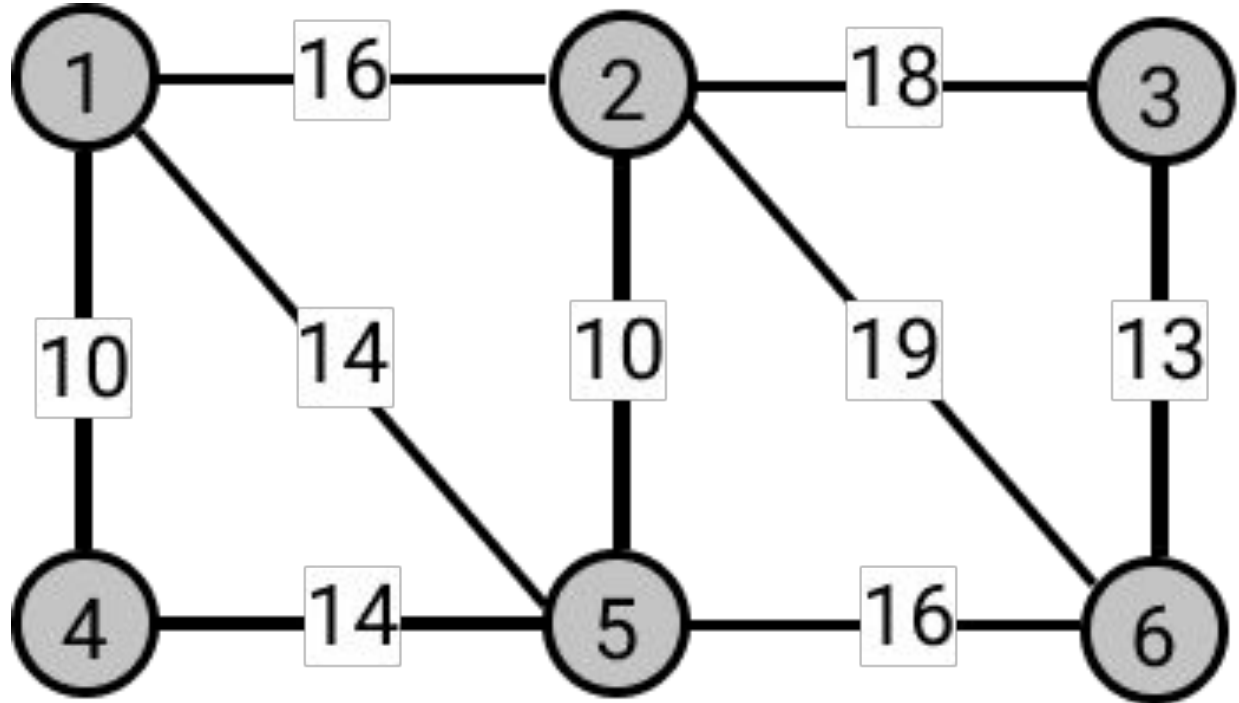
2016H - Oppgave 14

$lf = \{$
 $(1, 4),$
 $(2, 5)$
 $\}$



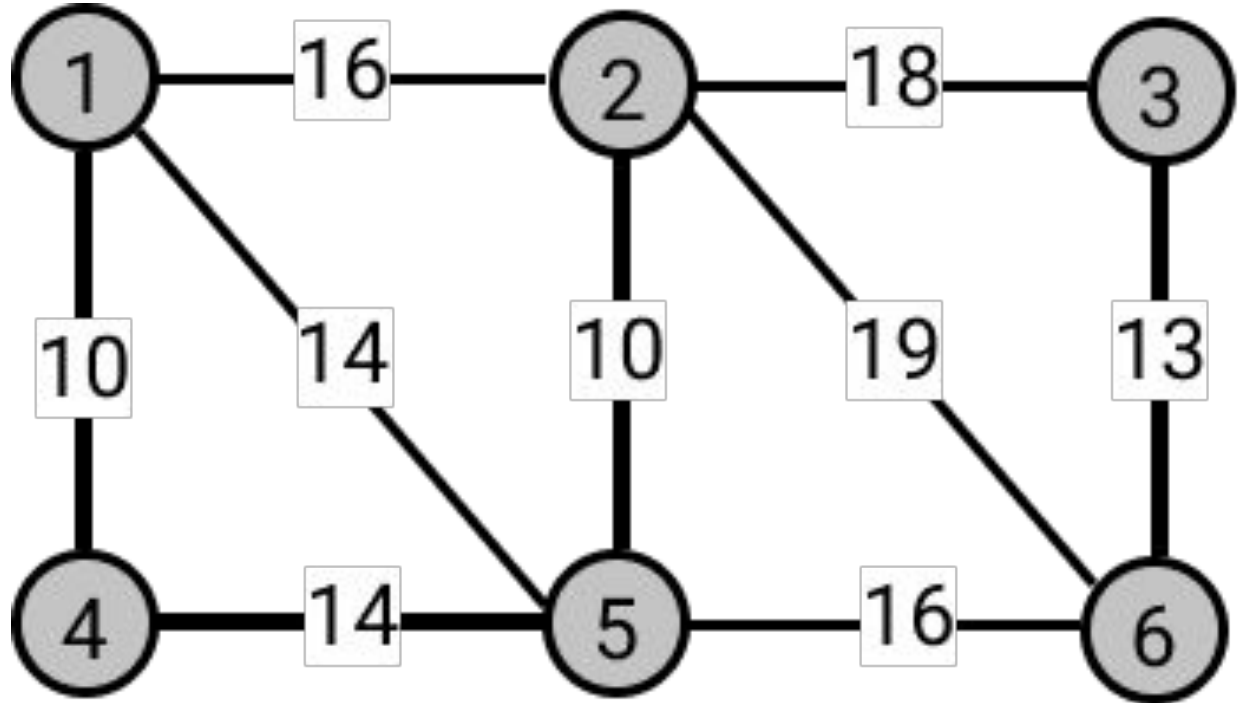
2016H - Oppgave 14

$lf = \{$
 $(1, 4),$
 $(2, 5),$
 $(3, 6)$
 $\}$



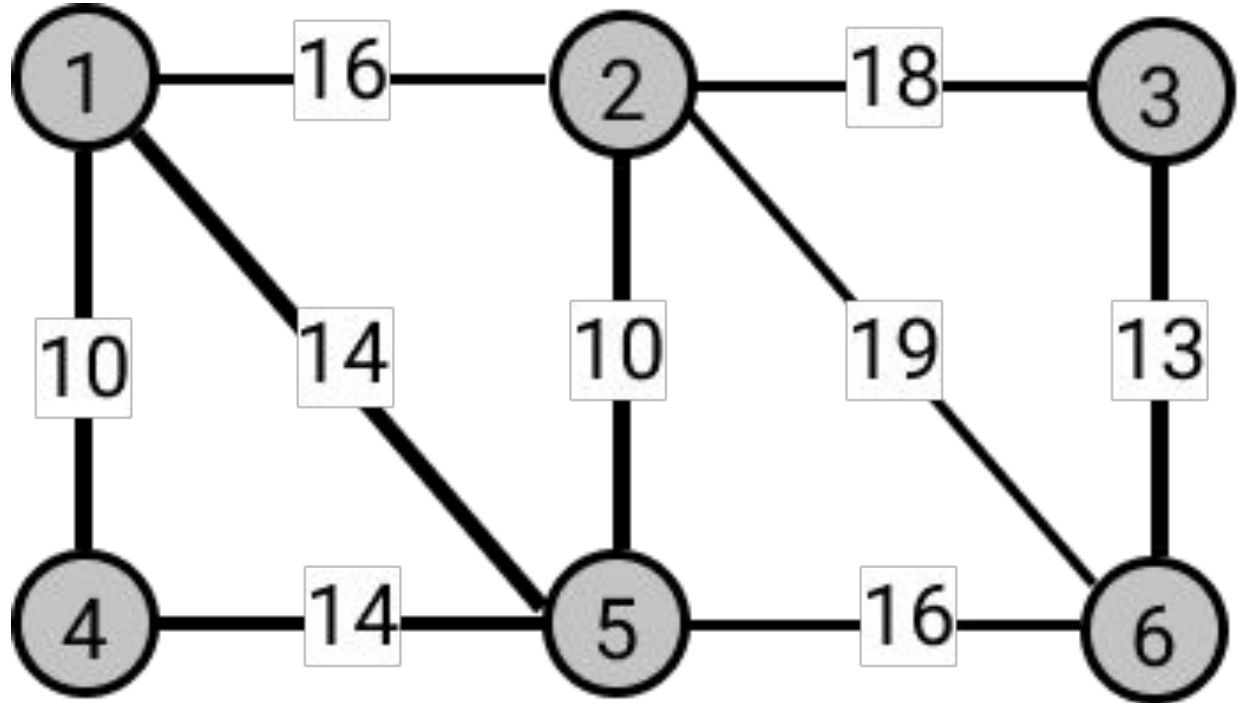
2016H - Oppgave 14

$lf = \{$
 $(1, 4),$
 $(2, 5),$
 $(3, 6),$
 $(4, 5)$
 $\}$



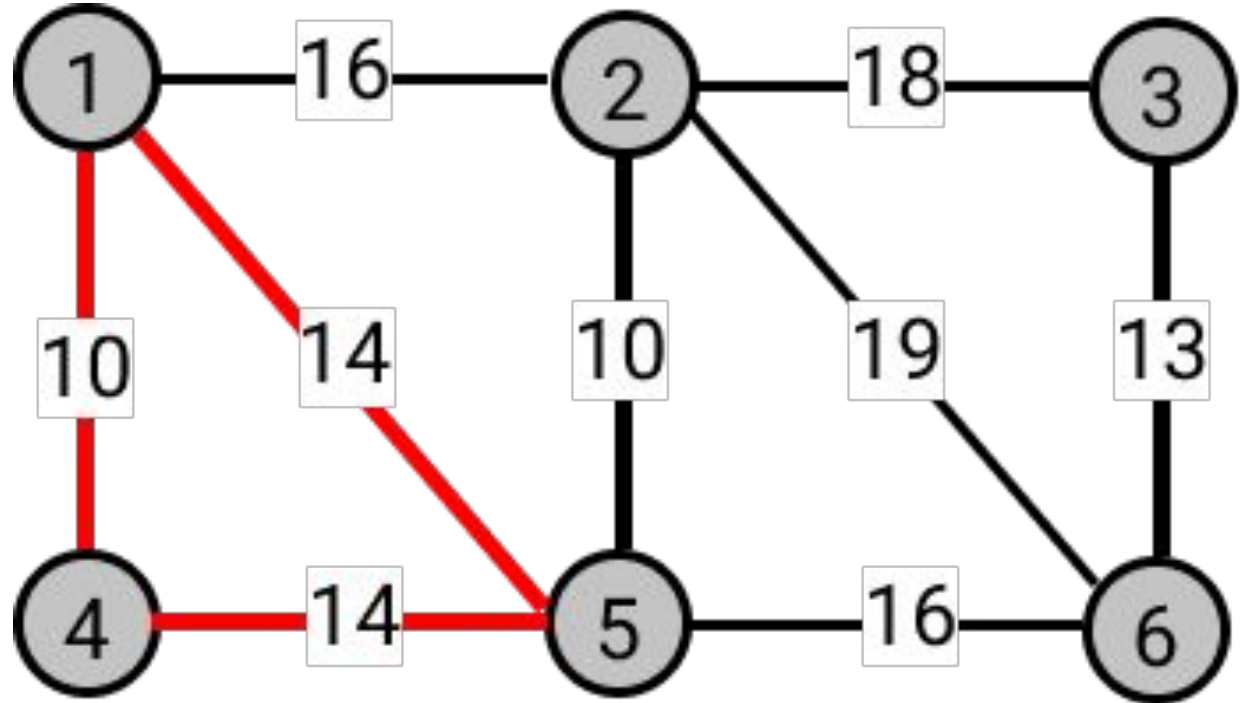
2016H - Oppgave 14

$lf = \{$
 $(1, 4),$
 $(2, 5),$
 $(3, 6),$
 $(1, 5)$
 $\}$



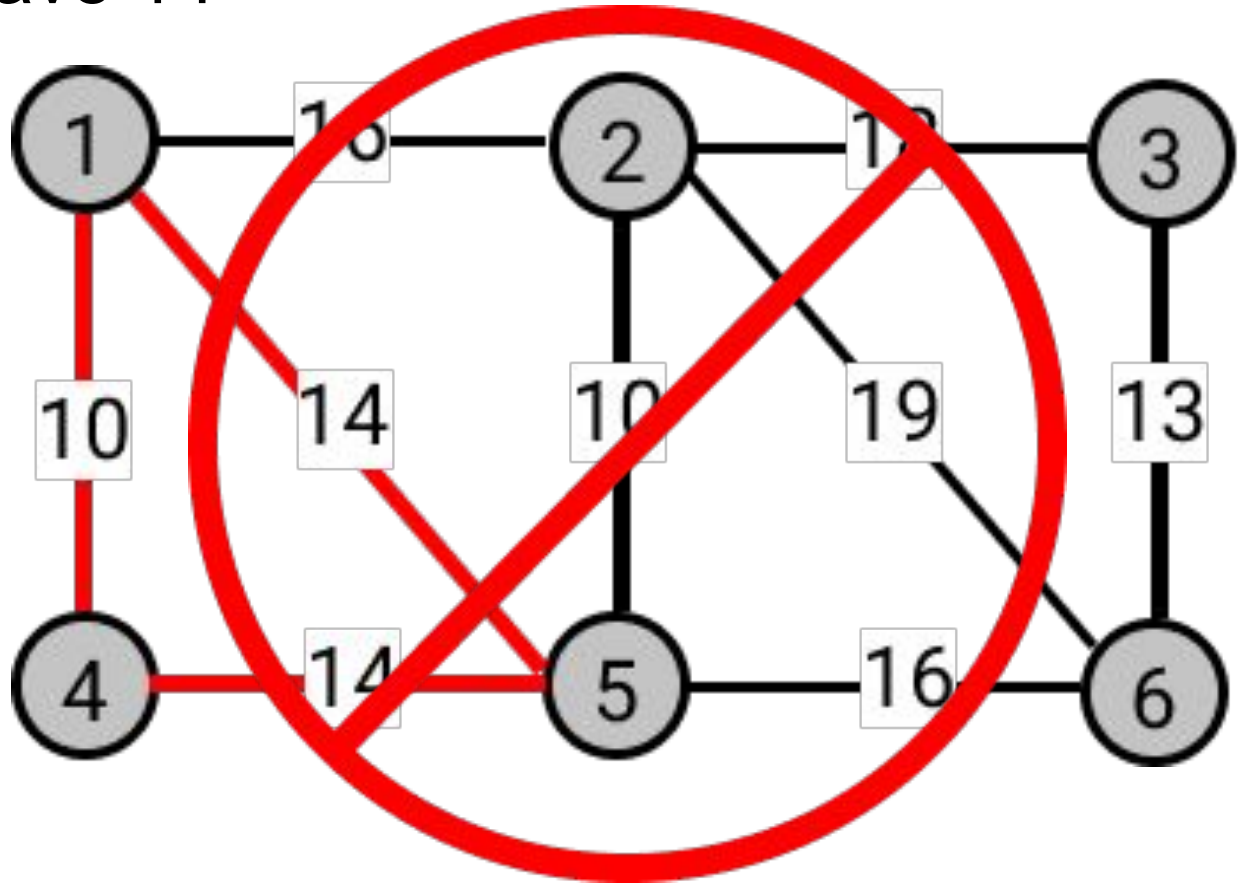
2016H - Oppgave 14

$lf = \{$
 $(1, 4),$
 $(2, 5),$
 $(3, 6),$
 $(4, 5),$
 $(1, 5)$
 $\}$



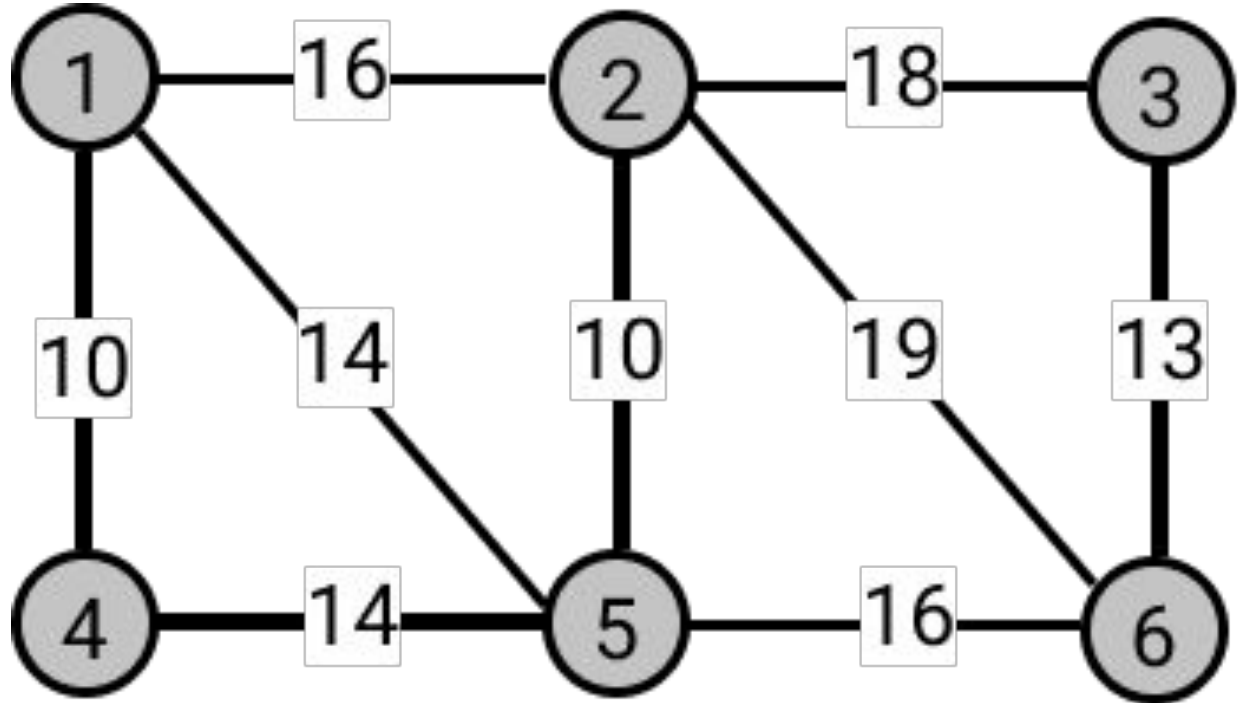
2016H - Oppgave 14

$lf = \{$
 $(1, 4),$
 $(2, 5),$
 $(3, 6),$
 $(4, 5)$
 $\}$



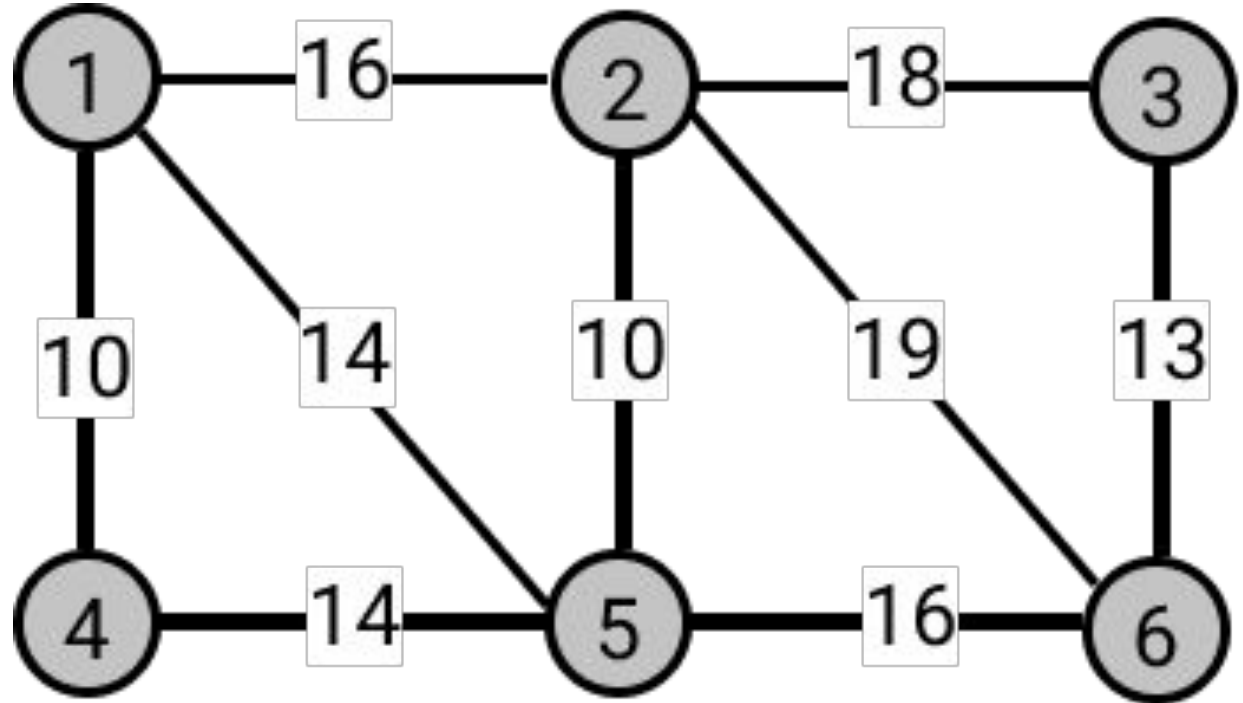
2016H - Oppgave 14

$lf = \{$
 $(1, 4),$
 $(2, 5),$
 $(3, 6),$
 $(4, 5)$
 $\}$



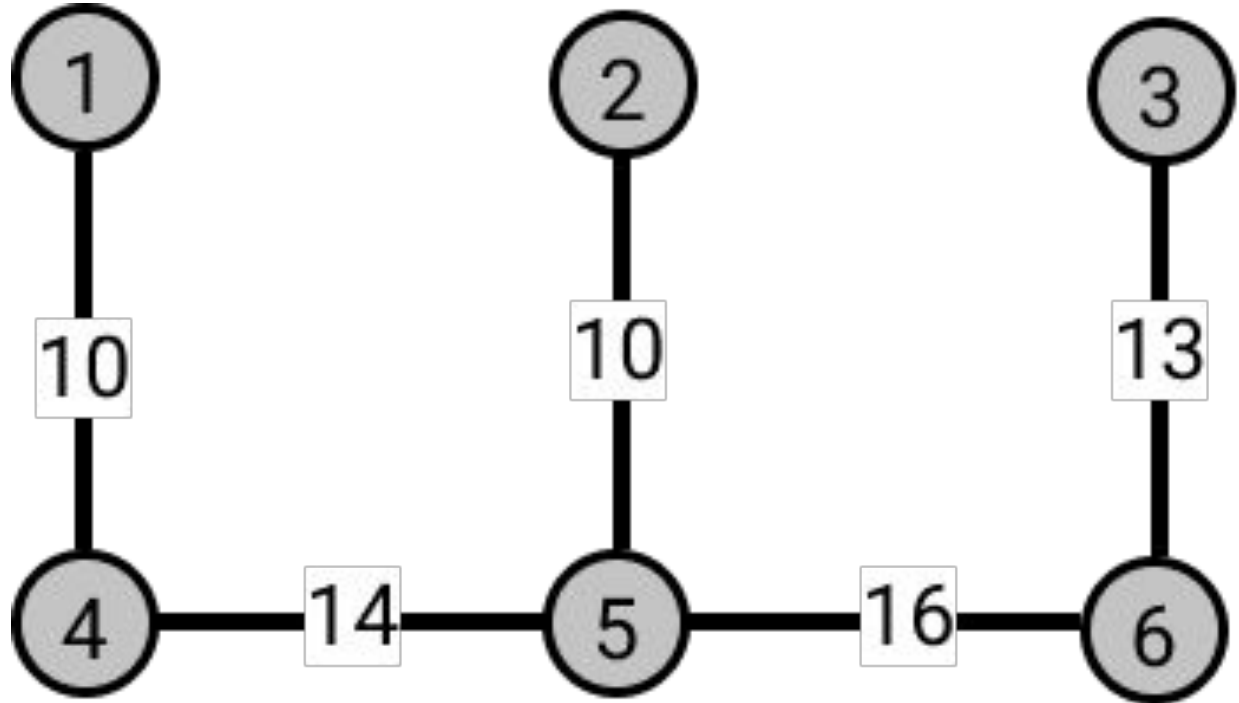
2016H - Oppgave 14

$lf = \{$
 $(1, 4),$
 $(2, 5),$
 $(3, 6),$
 $(4, 5),$
 $(5, 6)$
 $\}$



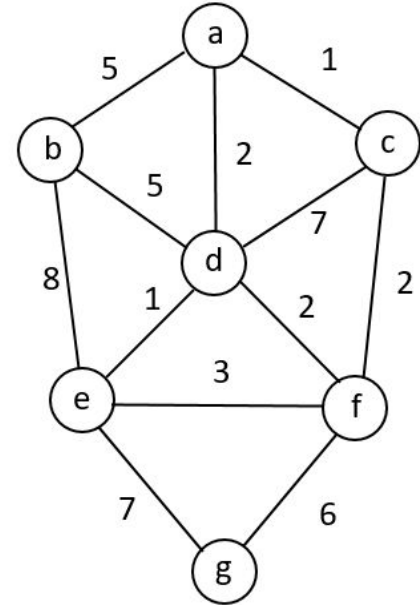
2016H - Oppgave 14

$lf = \{$
 $(1, 4),$
 $(2, 5),$
 $(3, 6),$
 $(4, 5),$
 $(5, 6)$
 $\}$

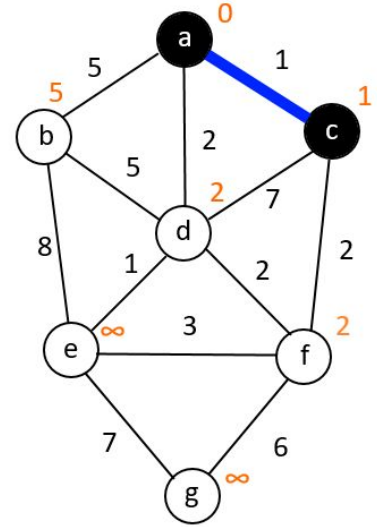
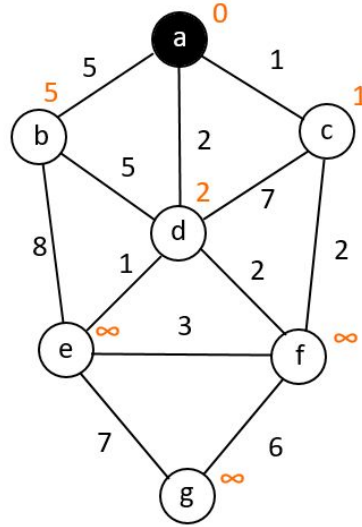
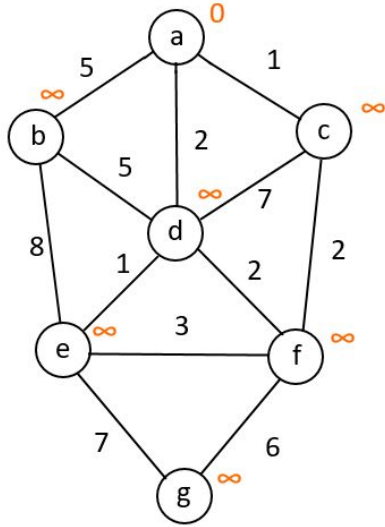


Prims algoritme

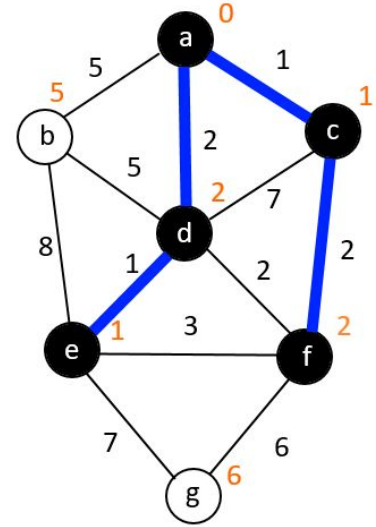
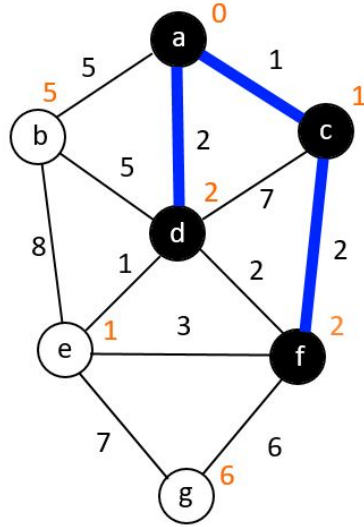
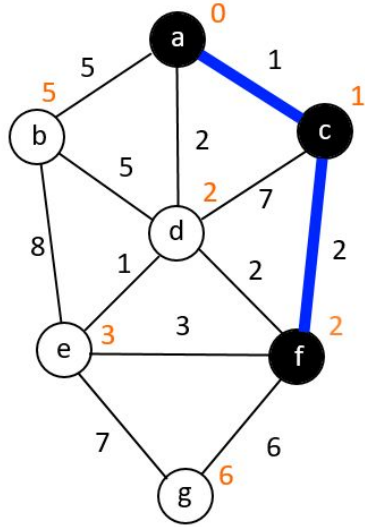
- Vedlikeholder en løsningsmengde med noder
- Legger til en node som deler kant med løsningsmengden hver iterasjon
- Så lenge kanten denne noden legger til ikke gir en sykel og er den billigst mulige, så er den trygg
- $O(E \lg V)$ (med min-heap)



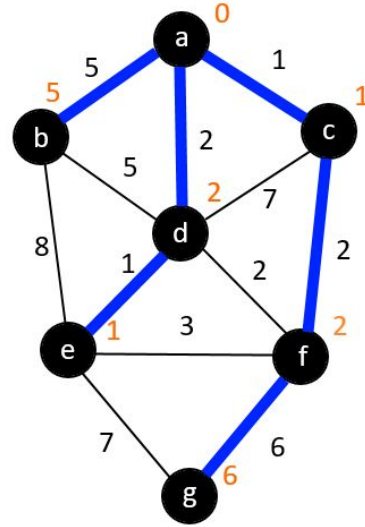
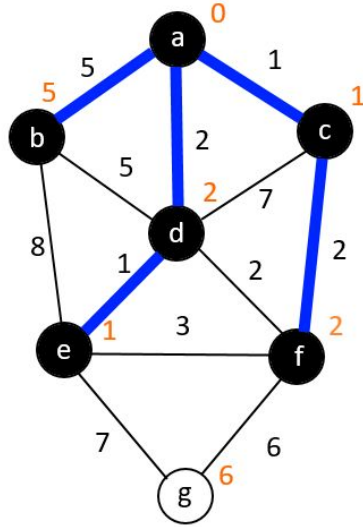
Prims algoritme



Prims algoritme

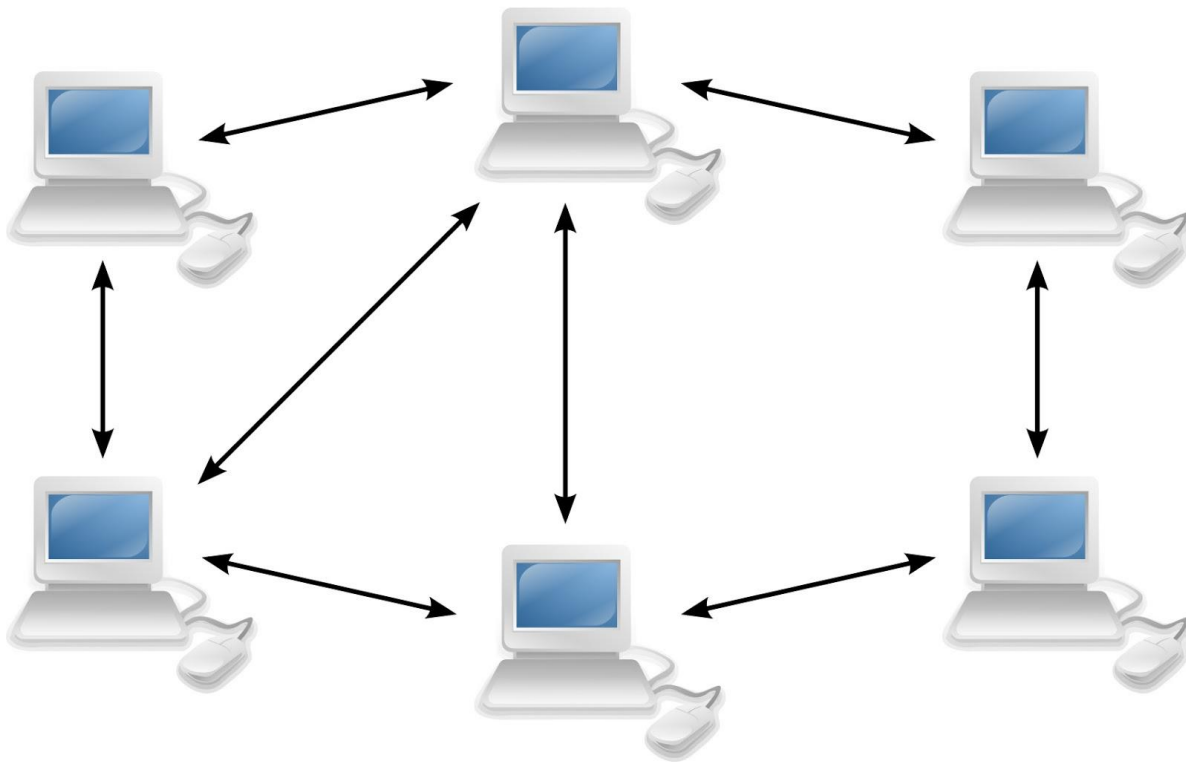


Prims algoritme

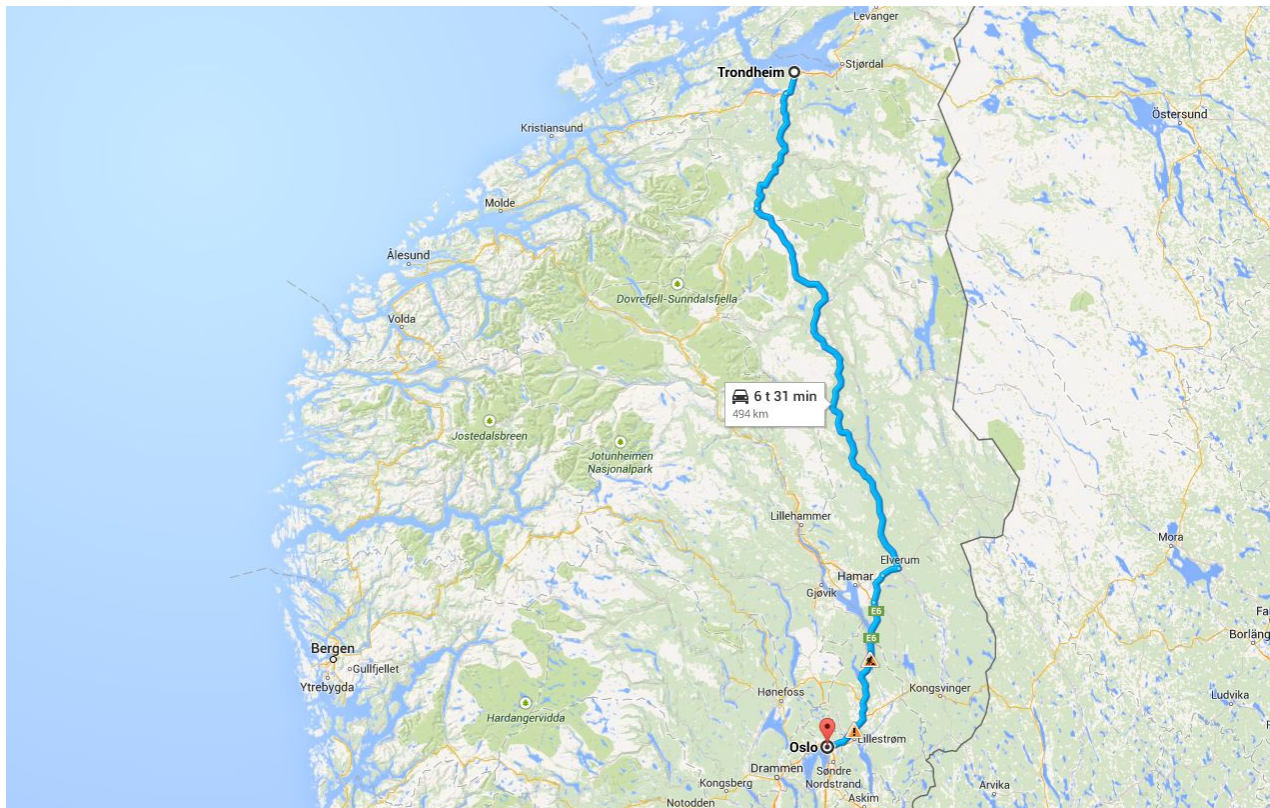


Korteste vei

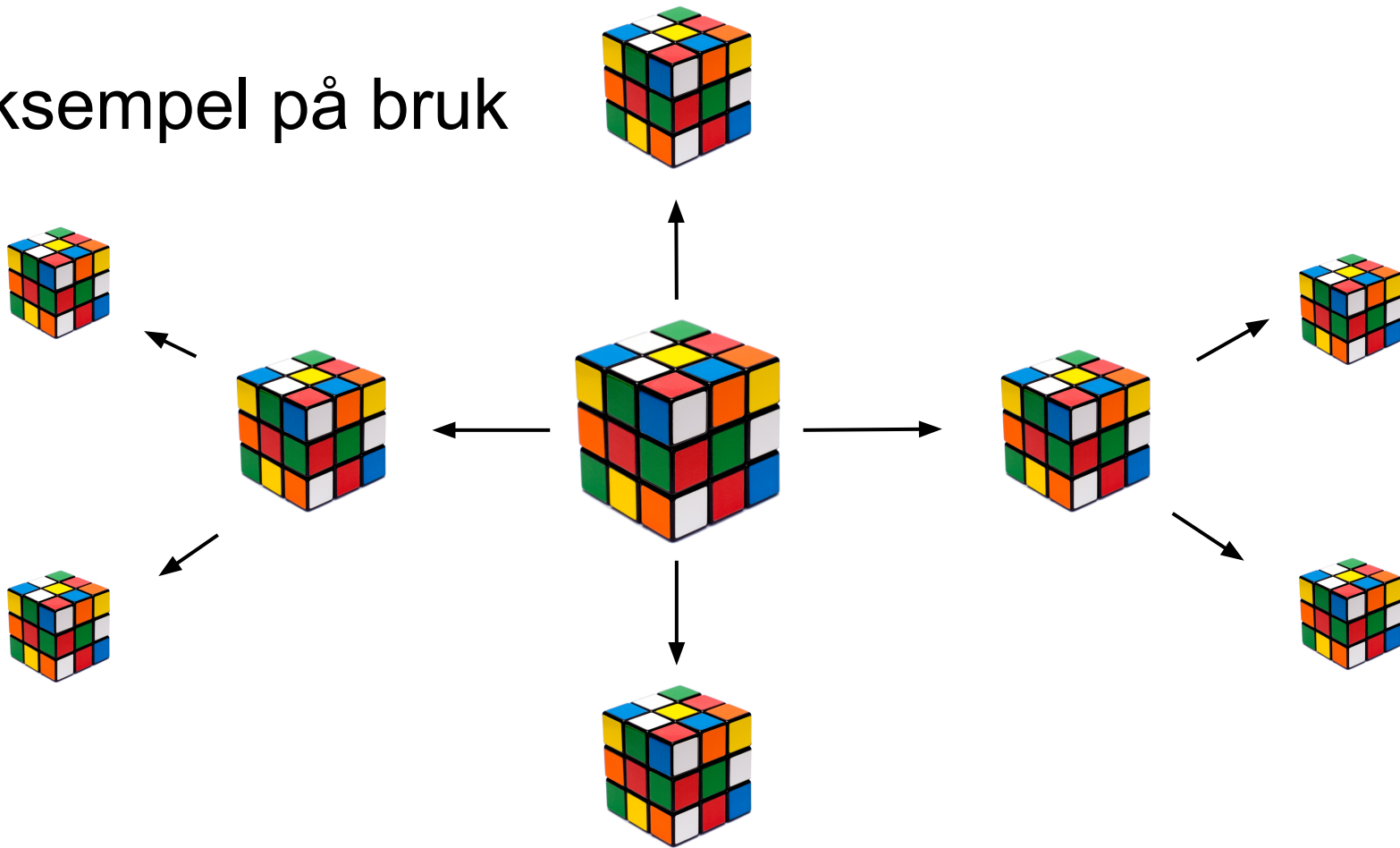
Eksempel på bruk



Eksempel på bruk

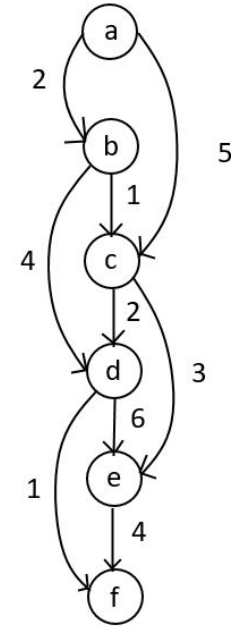


Eksempel på bruk



Generelt om problemet

- En korteste vei mellom to noder i en graf er en sti mellom to noder med minst mulig vekt
- Optimal substruktur: en korteste vei består av korteste veier
- Om grafen har negative sykler er ikke problemet definert.
- Korteste veier er enkle, de inneholder ingen sykler



Relaxation

INITIALIZE-SINGLE-SOURCE (G, s)

for each vertex $v \in G.V$

$v.d = \infty$

$v.\pi = \text{NIL}$

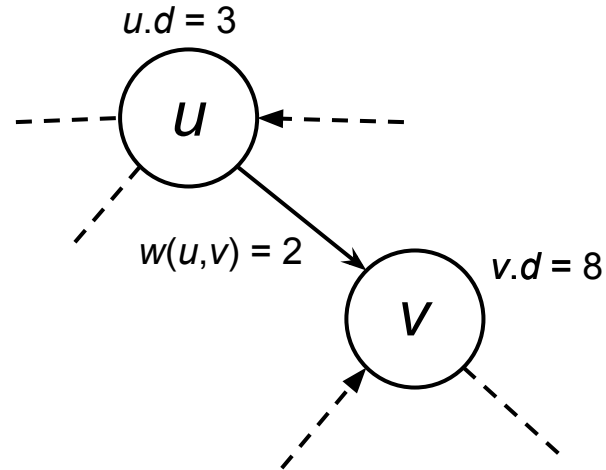
$s.d = 0$

RELAX (u, v, w)

if $v.d > u.d + w(u, v)$

$v.d = u.d + w(u, v)$

$v.\pi = u$



Relaxation

INITIALIZE-SINGLE-SOURCE (G, s)

for each vertex $v \in G.V$

$v.d = \infty$

$v.\pi = \text{NIL}$

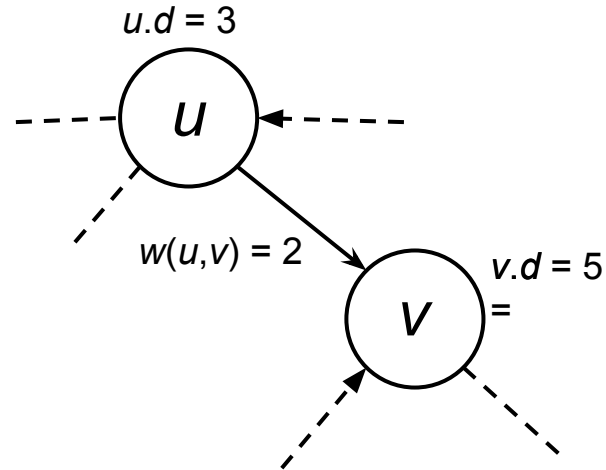
$s.d = 0$

RELAX (u, v, w)

if $v.d > u.d + w(u, v)$

$v.d = u.d + w(u, v)$

$v.\pi = u$



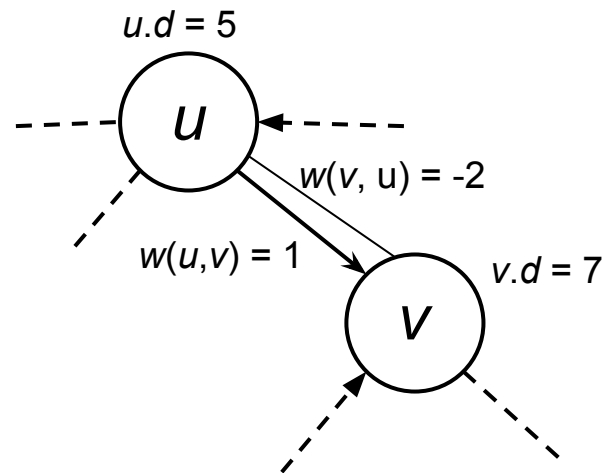
H2014, oppgave 2b

b) Under utførelsen av en eller annen korteste-vei-algoritme har vi $u.d = 5$, $v.d = 7$, $w(u, v) = 1$ og $w(v, u) = -2$. Vi utfører ett kall til $\text{RELAX}(u, v, w)$ og deretter ett kall til $\text{RELAX}(v, u, w)$. Hvilken verdi har $u.d$ nå?

Svar (5 p):

H2014, oppgave 2b

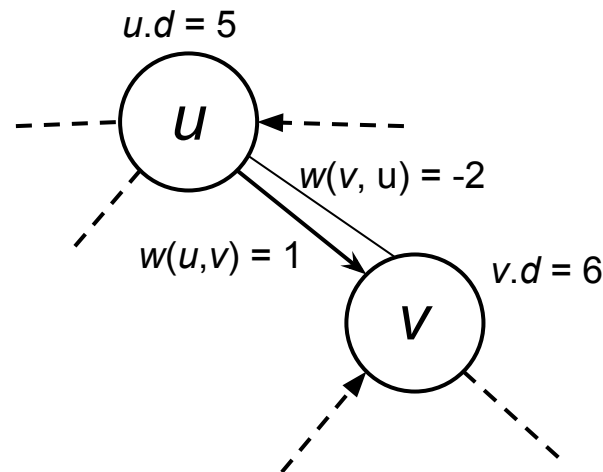
b) Under utførelsen av en eller annen korteste-vei-algoritme har vi $u.d = 5$, $v.d = 7$, $w(u, v) = 1$ og $w(v, u) = -2$. Vi utfører ett kall til $\text{RELAX}(u, v, w)$ og deretter ett kall til $\text{RELAX}(v, u, w)$. Hvilken verdi har $u.d$ nå?



H2014, oppgave 2b

b) Under utførelsen av en eller annen korteste-vei-algoritme har vi $u.d = 5$, $v.d = 7$, $w(u, v) = 1$ og $w(v, u) = -2$. Vi utfører ett kall til $\text{RELAX}(u, v, w)$ og deretter ett kall til $\text{RELAX}(v, u, w)$. Hvilken verdi har $u.d$ nå?

$\text{RELAX}(u, v, w)$

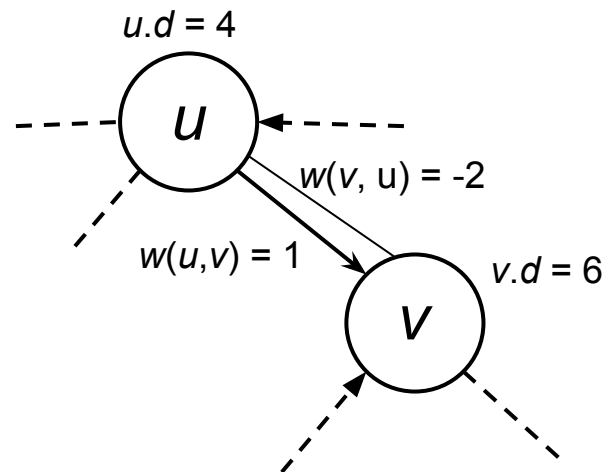


H2014, oppgave 2b

b) Under utførelsen av en eller annen korteste-vei-algoritme har vi $u.d = 5$, $v.d = 7$, $w(u, v) = 1$ og $w(v, u) = -2$. Vi utfører ett kall til $\text{RELAX}(u, v, w)$ og deretter ett kall til $\text{RELAX}(v, u, w)$. Hvilken verdi har $u.d$ nå?

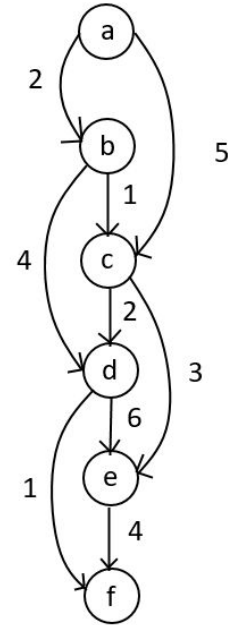
$\text{RELAX}(u, v, w)$

$\text{RELAX}(v, u, w)$



DAG shortest path

- Så lenge man traverserer i topologisk rekkefølge, så er man garantert at alle delproblemene man trenger løsning på er løst
- La den første noden være startnode, og oppdater de som kommer etter vha relax
- Den topologiske sorteringen lar oss finne korteste veg i $\Theta(E+V)$



Bellman-Ford

- Gjør $|V| - 1$ ganger:
 - **For** alle kanter (u,v) :
 - Relax(u,v)

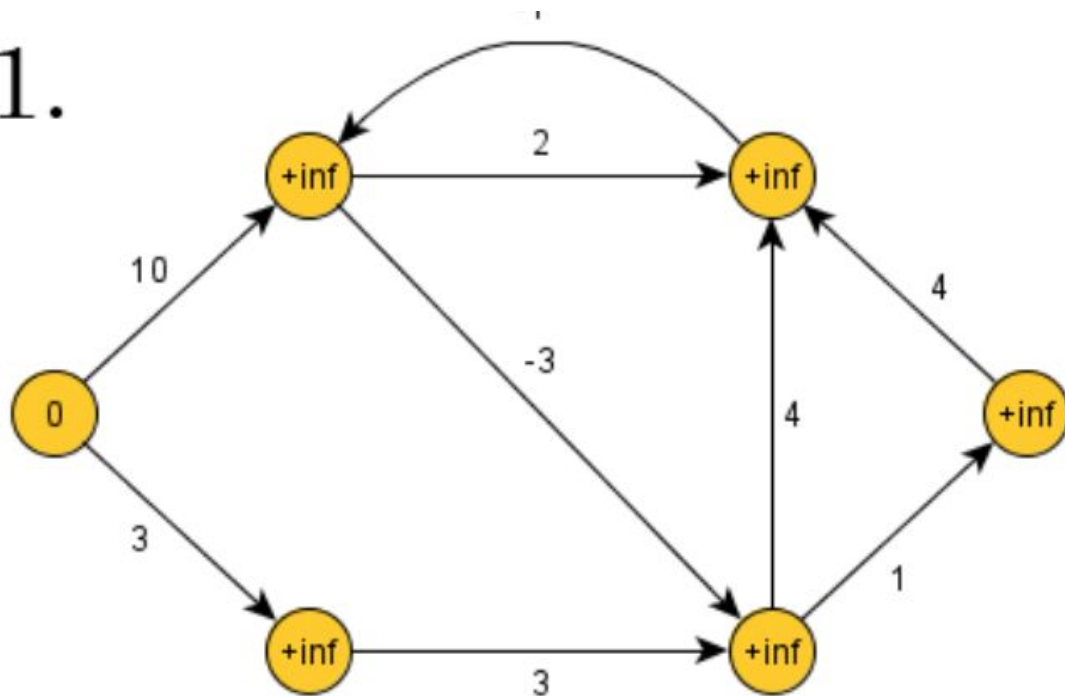
Kjøretid

$\Theta(VE)$

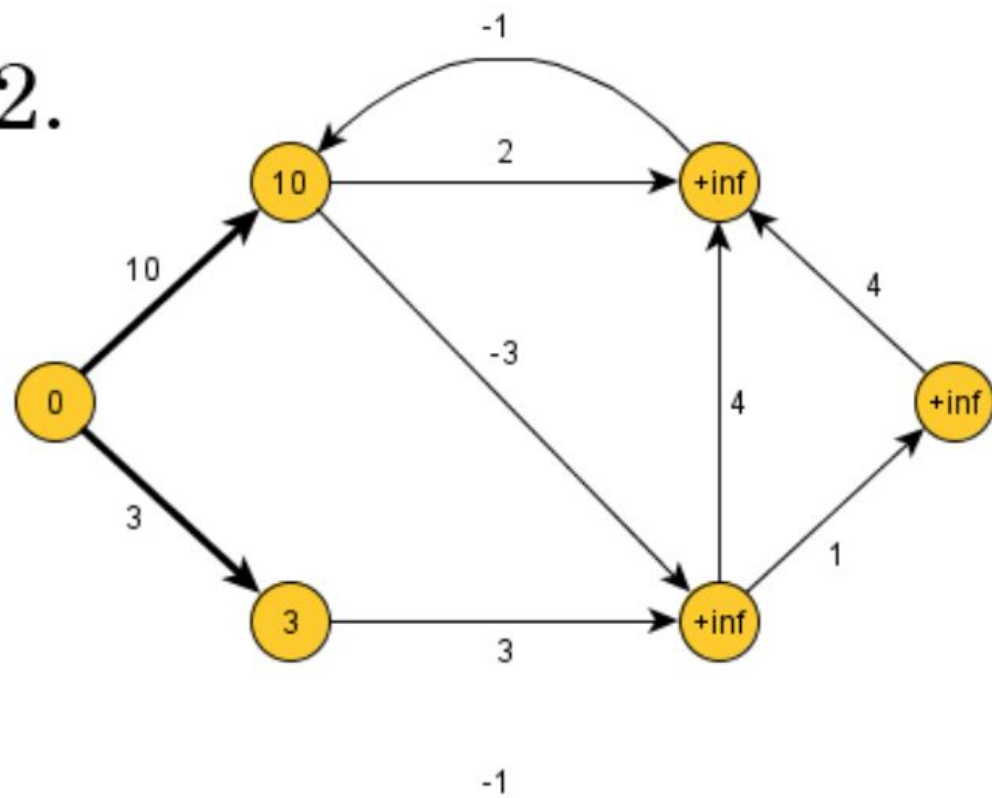
Kan oppdage negative sykler

Hvordan fungerer Bellman-Ford?

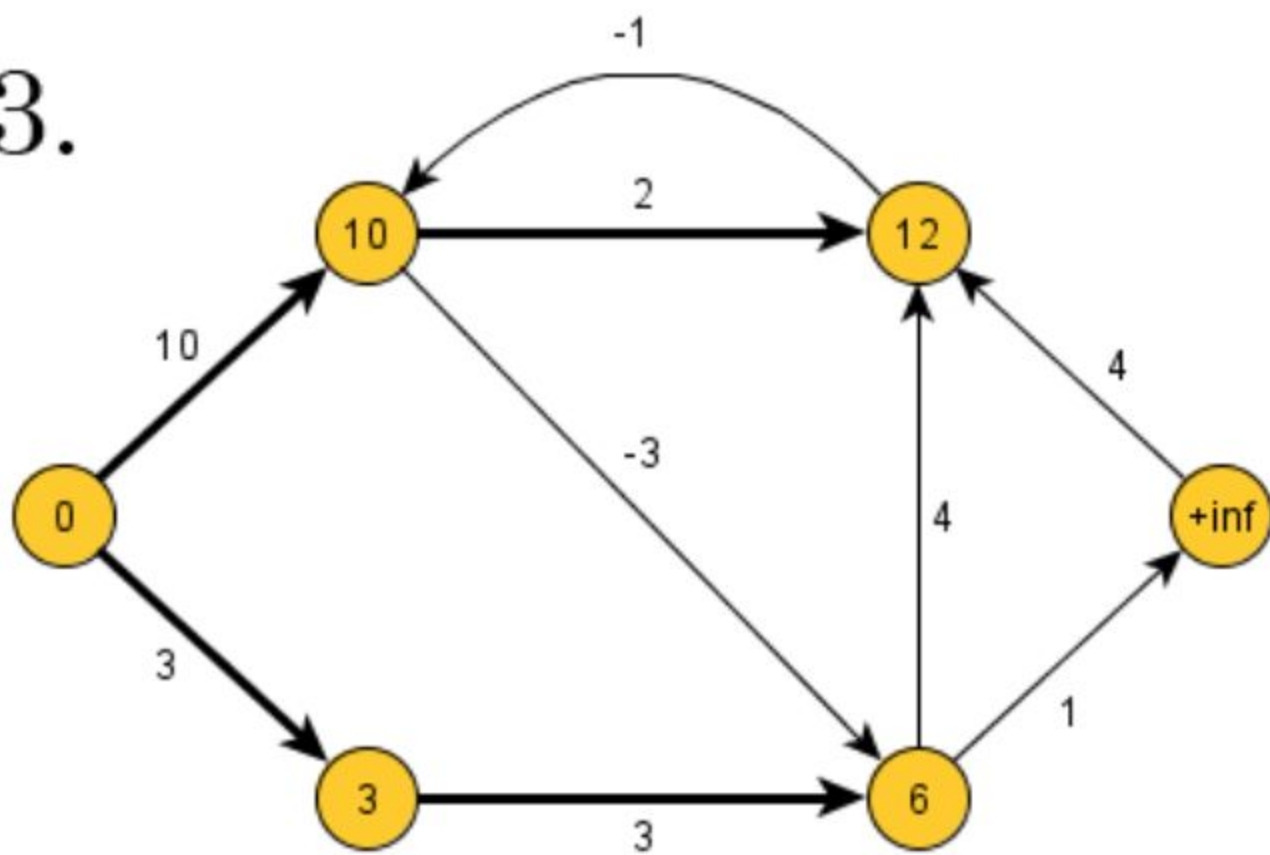
1.



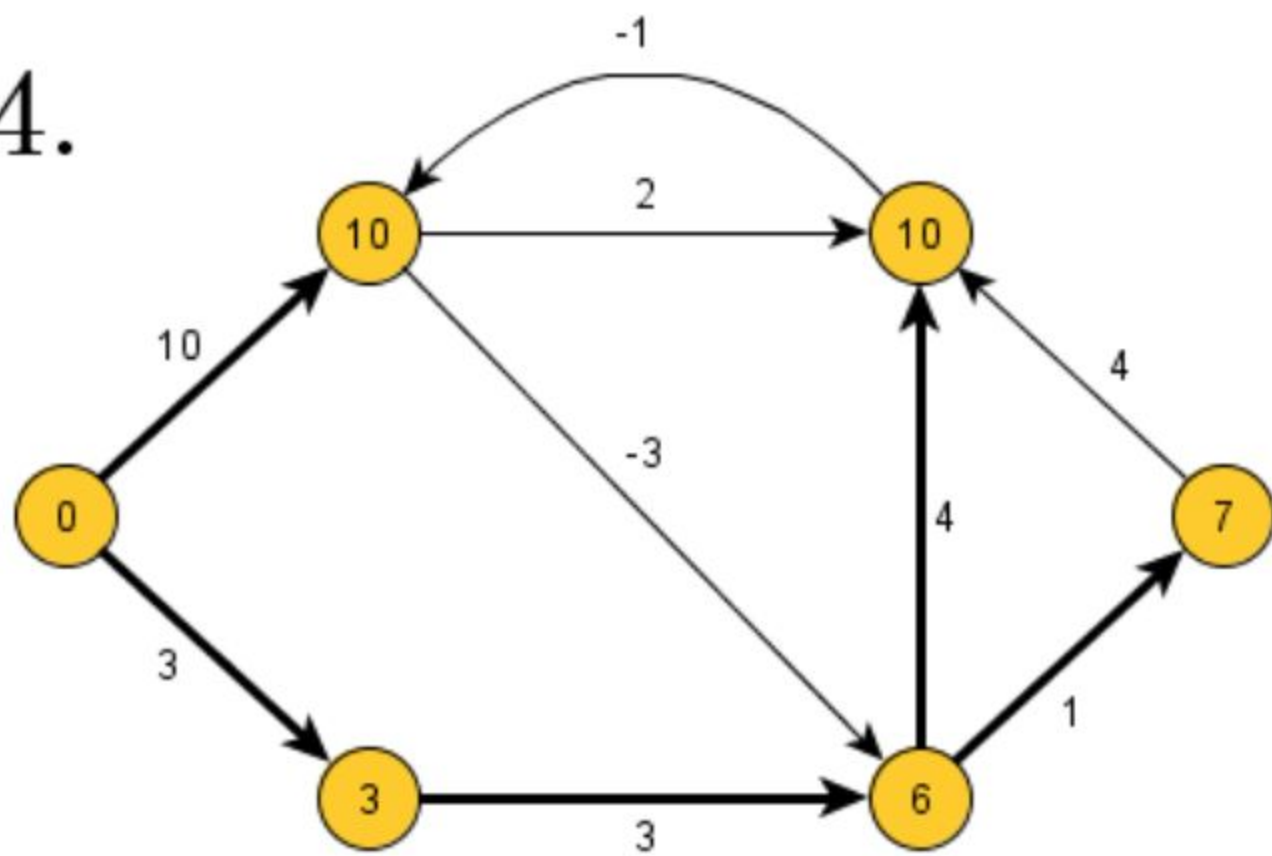
2.



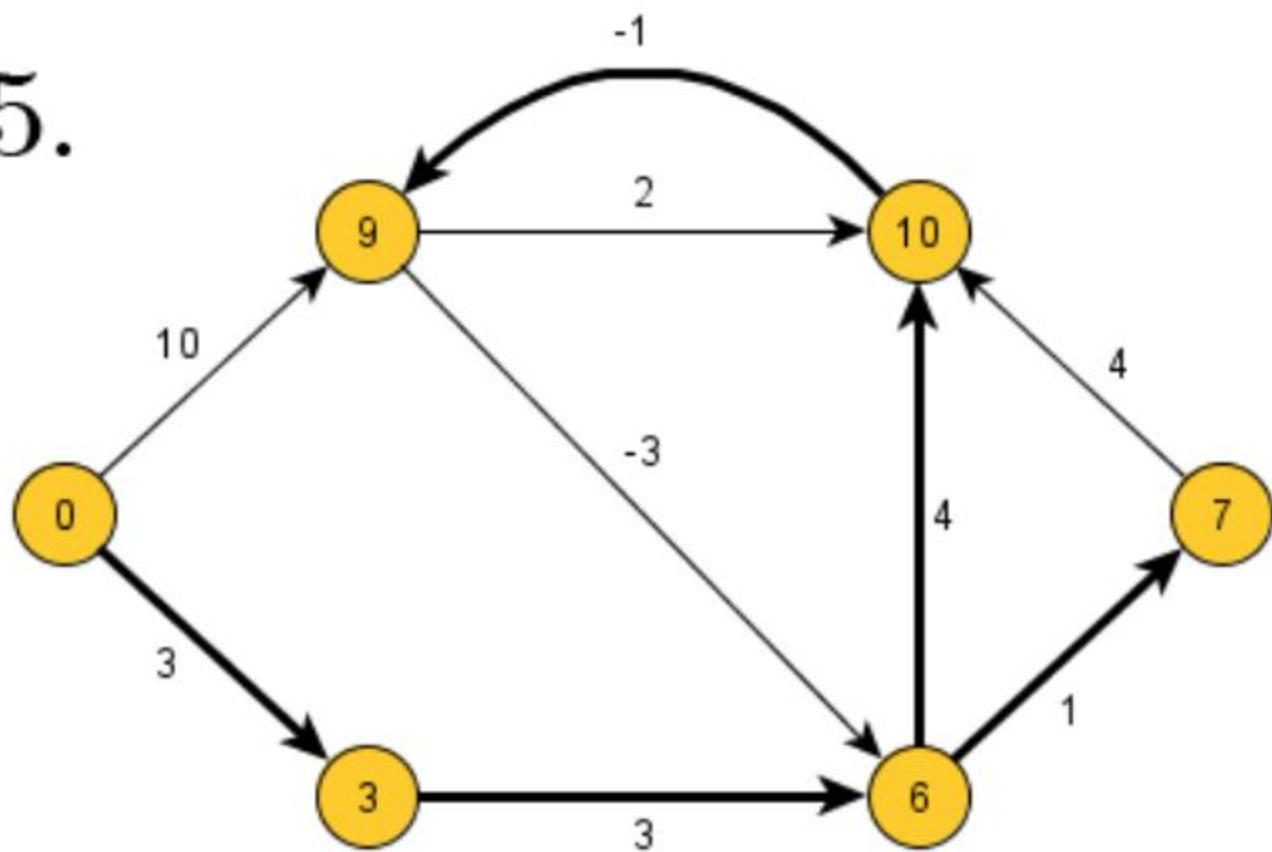
3.



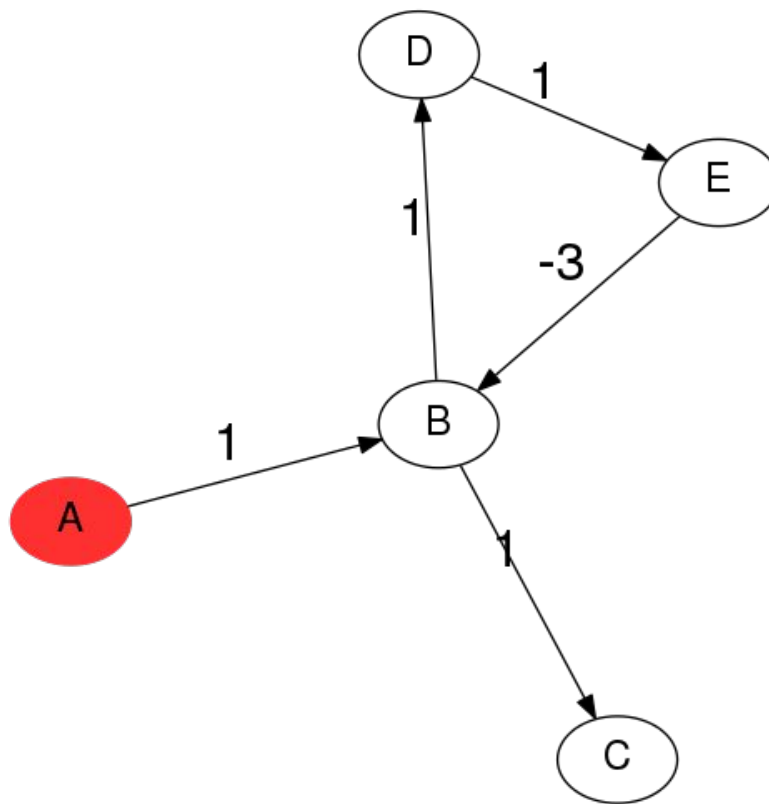
4.



5.



Negative sykler i Bellman-Ford



La oss kjøre Dijkstra på denne!

Dijkstra

- La Q være en min-prioritetskø
- Legg alle noder i Q
- Så lenge Q ikke er tom:
 - $u = Q.pop()$
 - for hver nabo v av u :
 - $Relax(u, v)$

Kjøretid:

$|V|$ Innsetninger og pop'inger

$|E|$ Relax

Array:

$O(V^2)$

Binary heap:

$O((V + E) \lg V)$

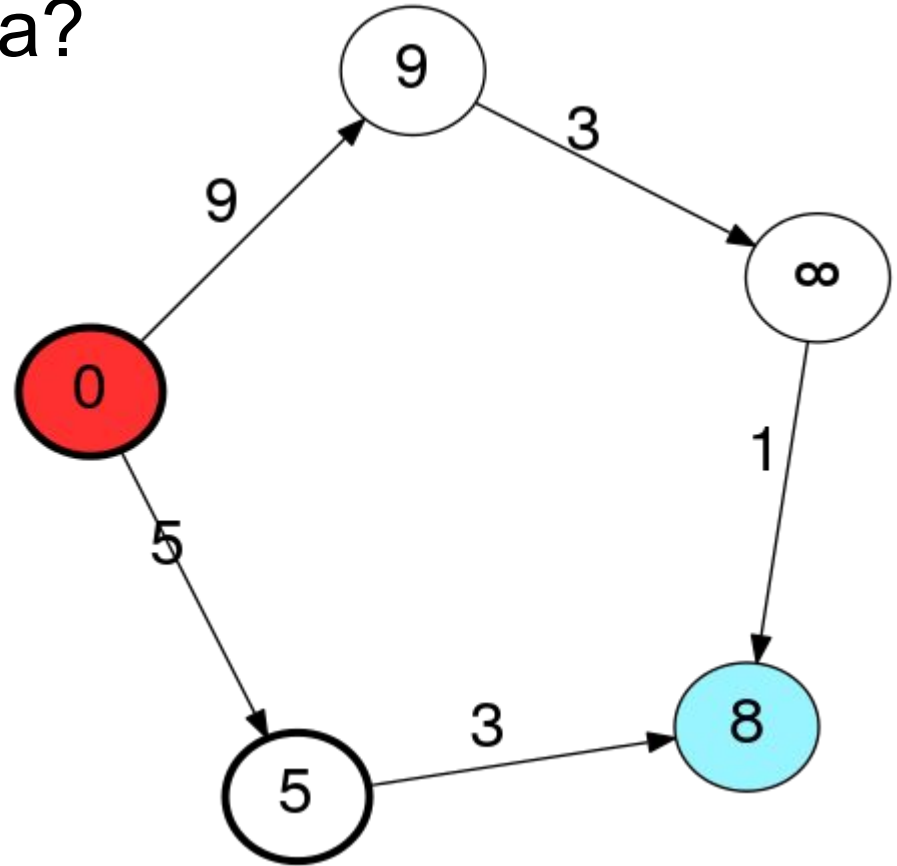
	Innsetting	Pop	Oppdater
Array	1	V	1
Binary Heap	$\log V$	$\log V$	$\log V$

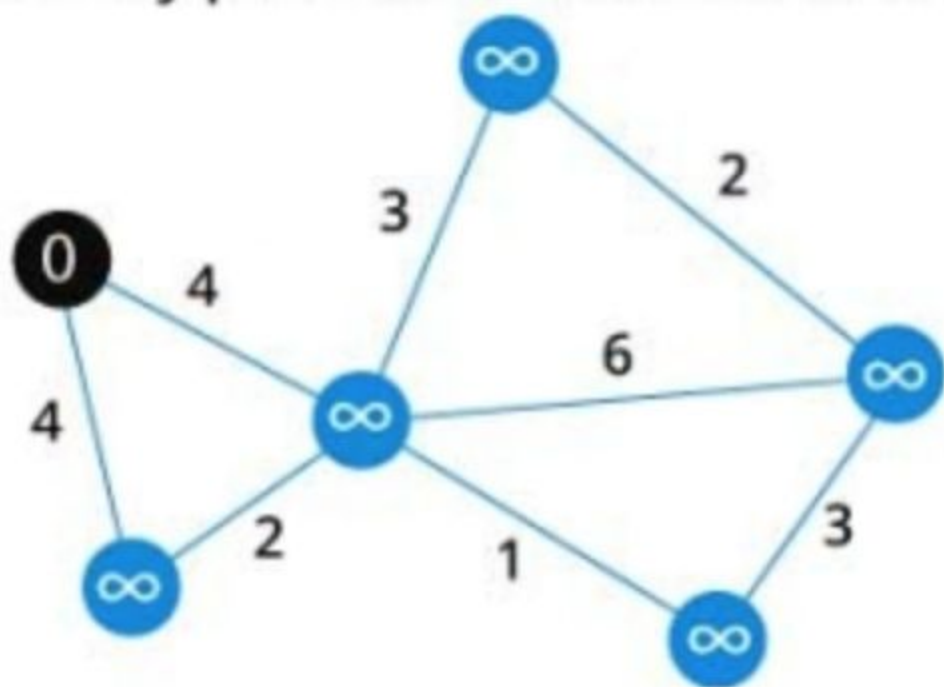
Hvorfor fungerer Dijkstra?

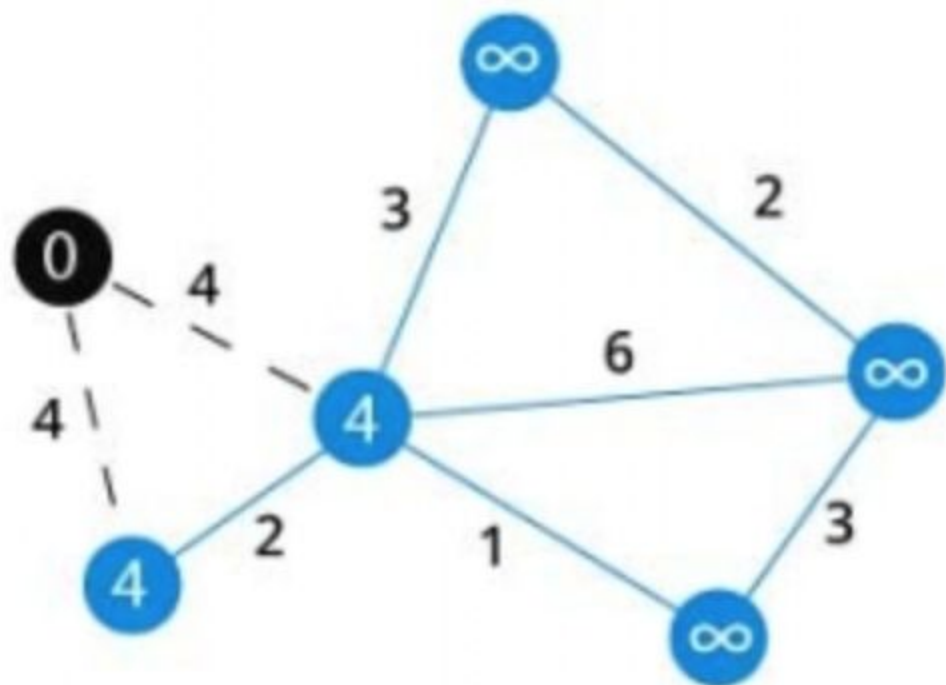
Når vi pop'er u vil $u.d$ være korteste vei.

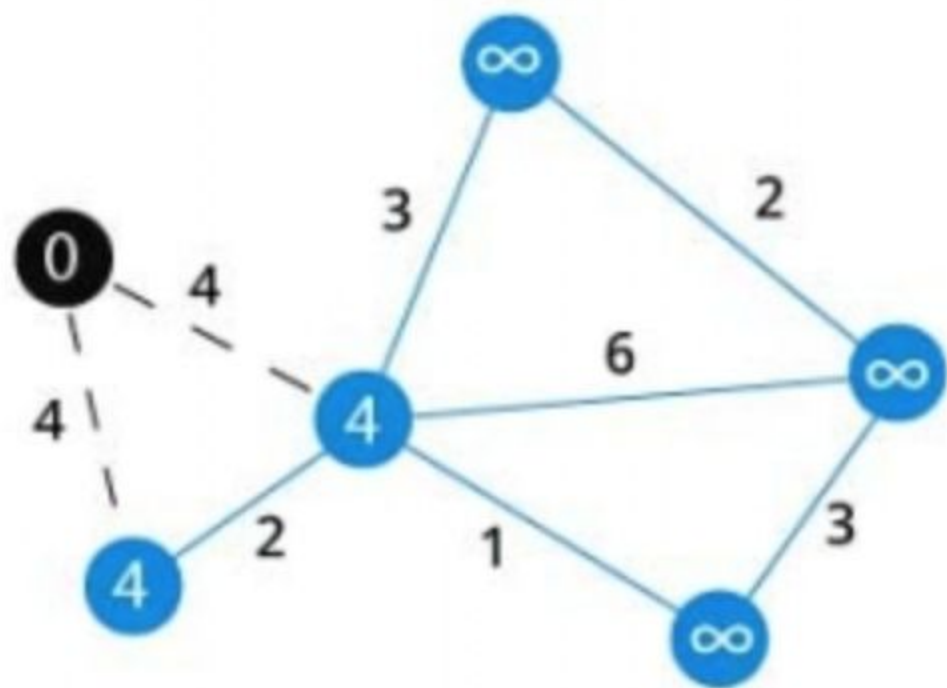
Altså: Når vi velger å besøke node u har vi allerede funnet korteste vei til u .

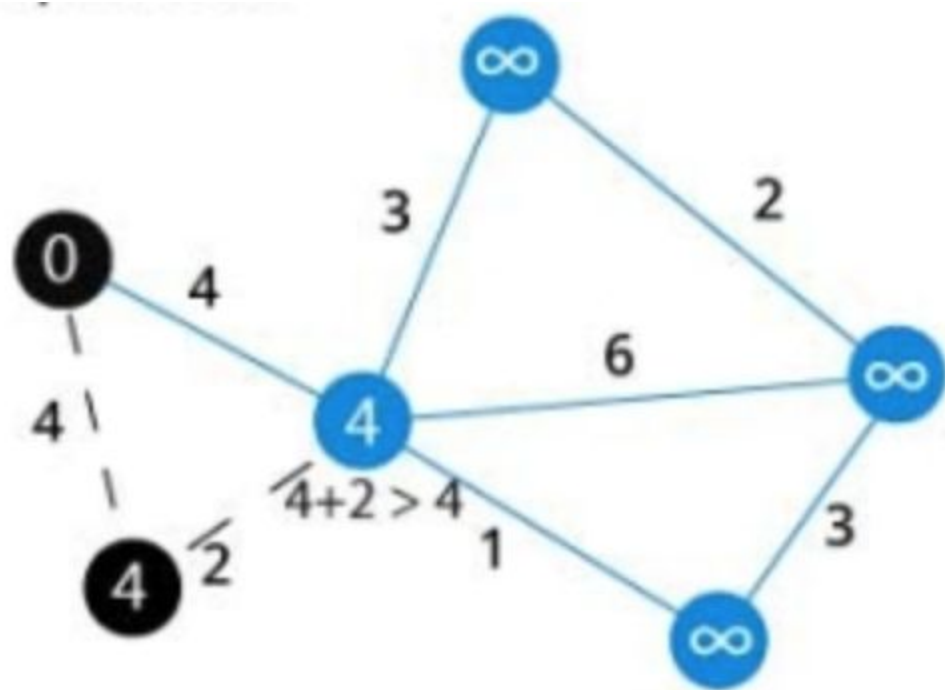
Hvorfor?

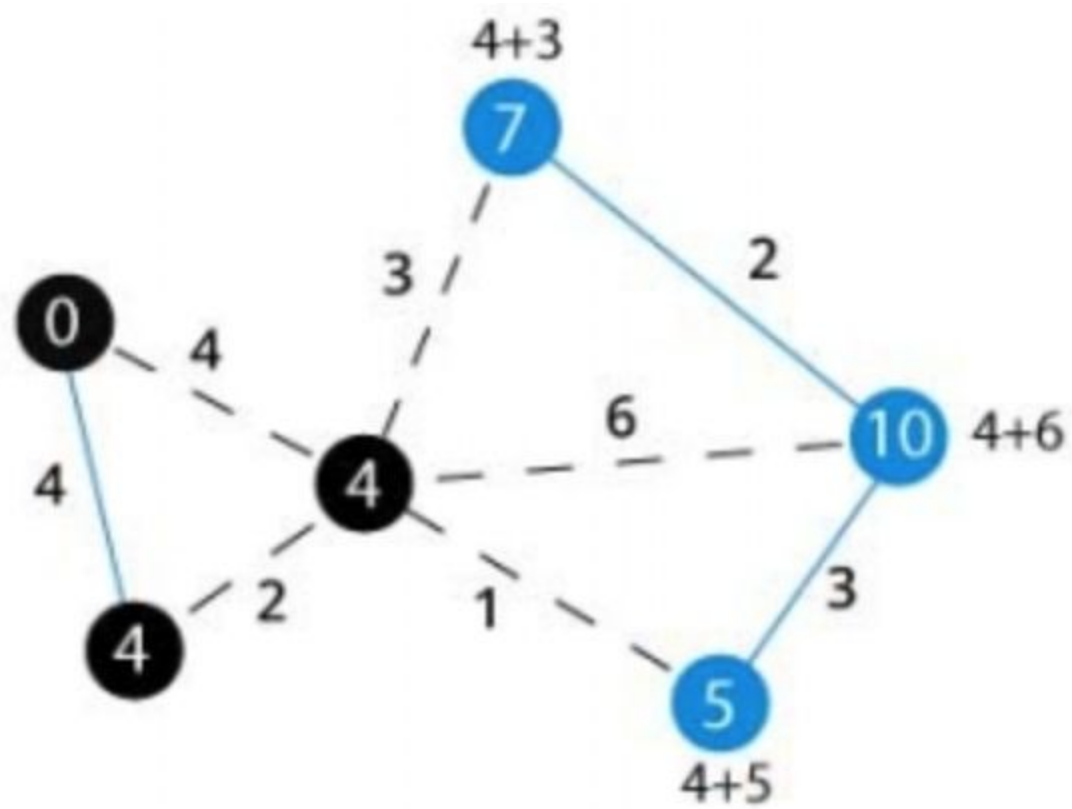


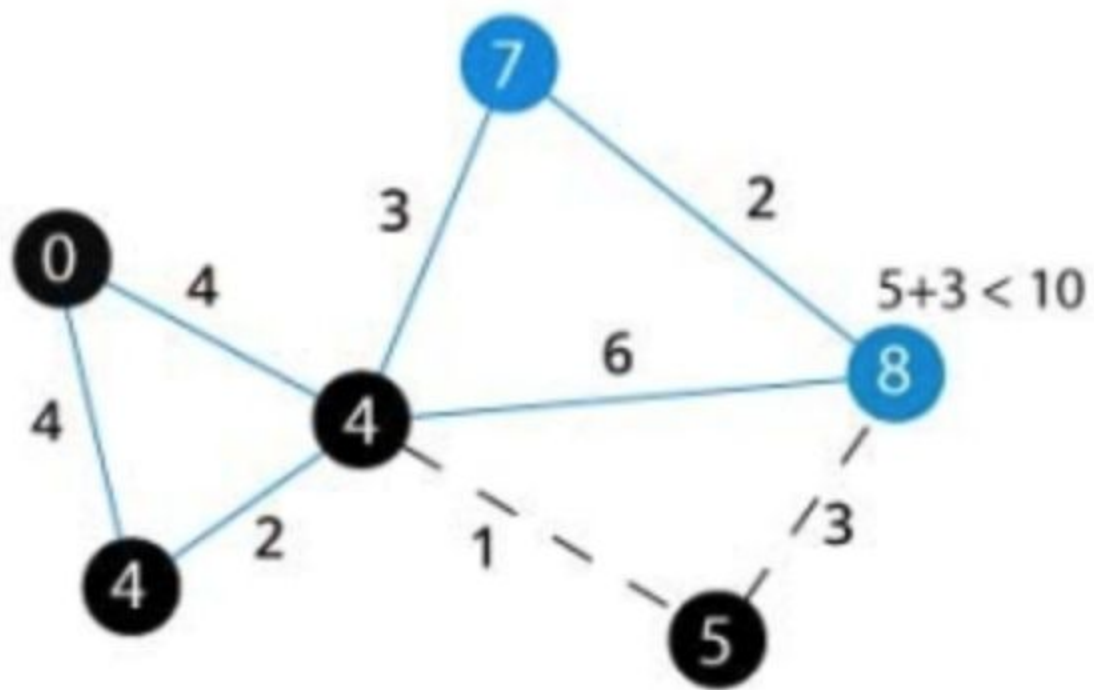


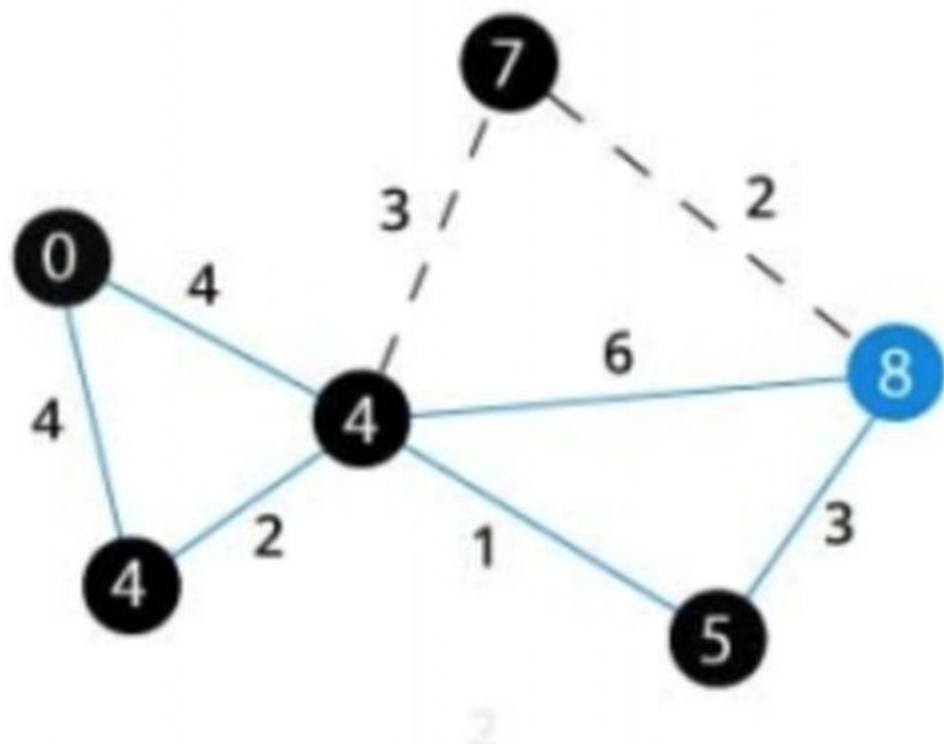


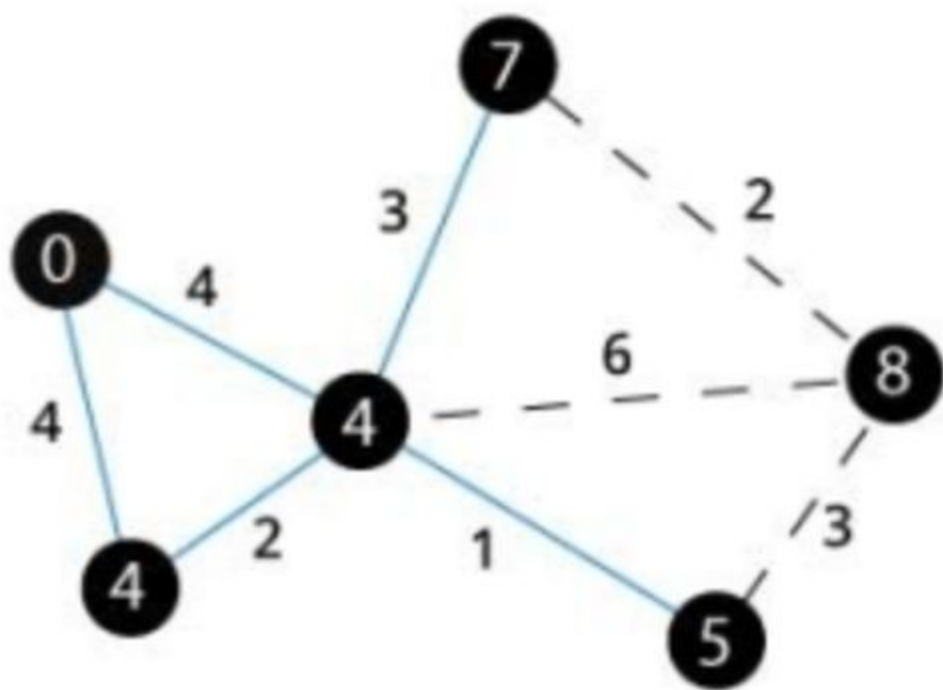












Dijkstra vs Prim

MST-PRIM(G, w, r)

```
1  for each  $u \in G.V$ 
2       $u.key = \infty$ 
3       $u.\pi = \text{NIL}$ 
4   $r.key = 0$ 
5   $Q = G.V$ 
6  while  $Q \neq \emptyset$ 
7       $u = \text{EXTRACT-MIN}(Q)$ 
8      for each  $v \in G.Adj[u]$ 
9          if  $v \in Q$  and  $w(u, v) < v.key$ 
10              $v.\pi = u$ 
11              $v.key = w(u, v)$ 
```

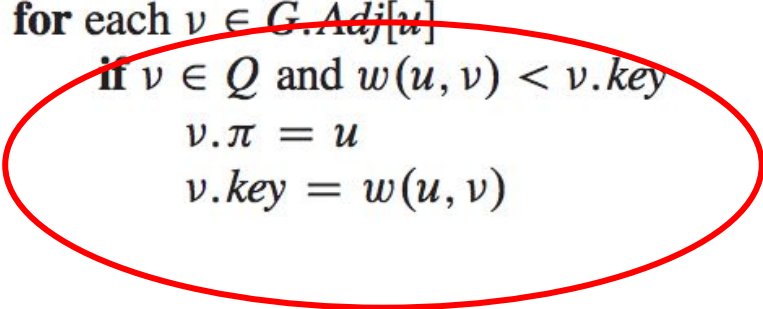
DIJKSTRA(G, w, s)

```
1  INITIALIZE-SINGLE-SOURCE( $G, s$ )
2   $S = \emptyset$ 
3   $Q = G.V$ 
4  while  $Q \neq \emptyset$ 
5       $u = \text{EXTRACT-MIN}(Q)$ 
6       $S = S \cup \{u\}$ 
7      for each vertex  $v \in G.Adj[u]$ 
8          RELAX( $u, v, w$ )
```

Dijkstra vs Prim

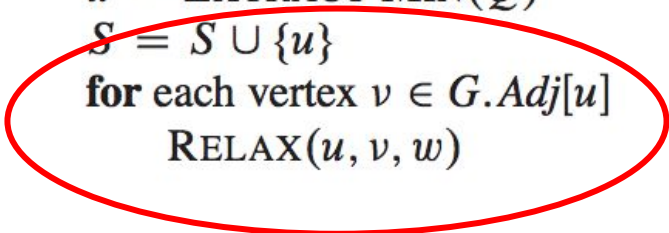
MST-PRIM(G, w, r)

```
1  for each  $u \in G.V$ 
2       $u.key = \infty$ 
3       $u.\pi = \text{NIL}$ 
4   $r.key = 0$ 
5   $Q = G.V$ 
6  while  $Q \neq \emptyset$ 
7       $u = \text{EXTRACT-MIN}(Q)$ 
8      for each  $v \in G.Adj[u]$ 
9          if  $v \in Q$  and  $w(u, v) < v.key$ 
10              $v.\pi = u$ 
11              $v.key = w(u, v)$ 
```



DIJKSTRA(G, w, s)

```
1  INITIALIZE-SINGLE-SOURCE( $G, s$ )
2   $S = \emptyset$ 
3   $Q = G.V$ 
4  while  $Q \neq \emptyset$ 
5       $u = \text{EXTRACT-MIN}(Q)$ 
6       $S = S \cup \{u\}$ 
7      for each vertex  $v \in G.Adj[u]$ 
8          RELAX( $u, v, w$ )
```



Dijkstra vs Prim

MST-PRIM(G, w, r)

```
1  for each  $u \in G.V$ 
2       $u.key = \infty$ 
3       $u.\pi = \text{NIL}$ 
4   $r.key = 0$ 
5   $Q = G.V$ 
6  while  $Q \neq \emptyset$ 
7       $u = \text{EXTRACT-MIN}(Q)$ 
8      for each  $v \in G.Adj[u]$ 
9          if  $v \in Q$  and  $w(u, v) < v.key$ 
10              $v.\pi = u$ 
11              $v.key = w(u, v)$ 
```

RELAX(u, v, w)

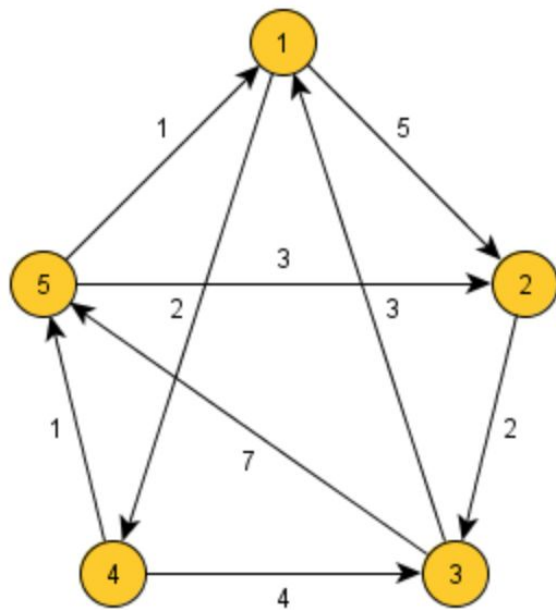
```
    if  $v.d > u.d + w(u, v)$ 
         $v.d = u.d + w(u, v)$ 
         $v.\pi = u$ 
```

DIJKSTRA(G, w, s)

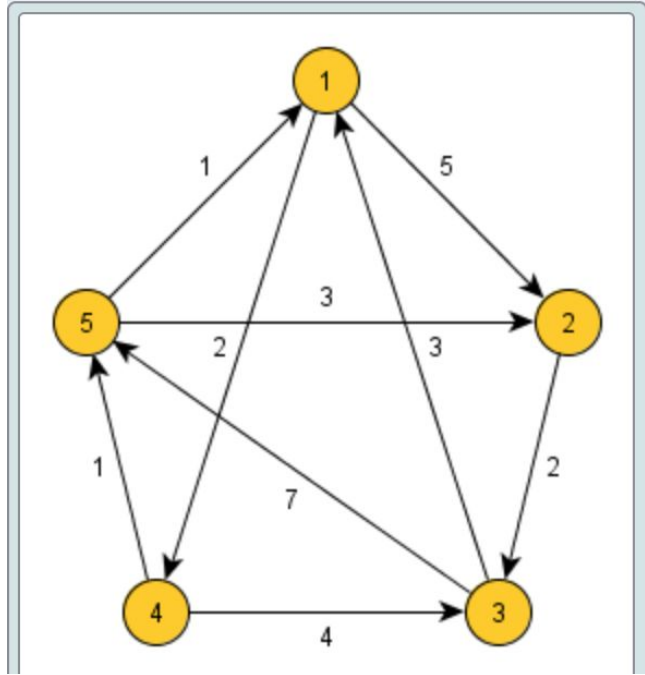
```
1  INITIALIZE-SINGLE-SOURCE( $G, s$ )
2   $S = \emptyset$ 
3   $Q = G.V$ 
4  while  $Q \neq \emptyset$ 
5       $u = \text{EXTRACT-MIN}(Q)$ 
6       $S = S \cup \{u\}$ 
7      for each vertex  $v \in G.Adj[u]$ 
8          RELAX( $u, v, w$ )
```


Floyd-Warshall

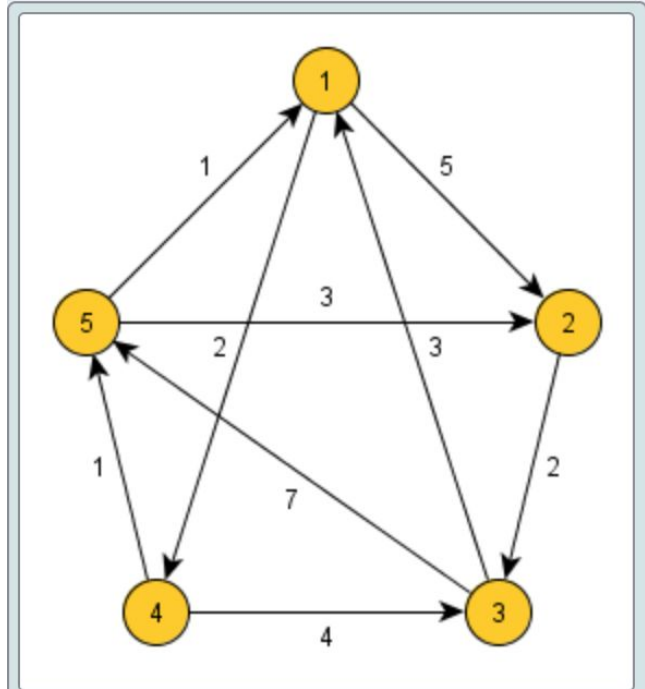
- Alle-Til-Alle
- Nabomatriser
- $O(n^3)$ for en $n \times n$ -matrise



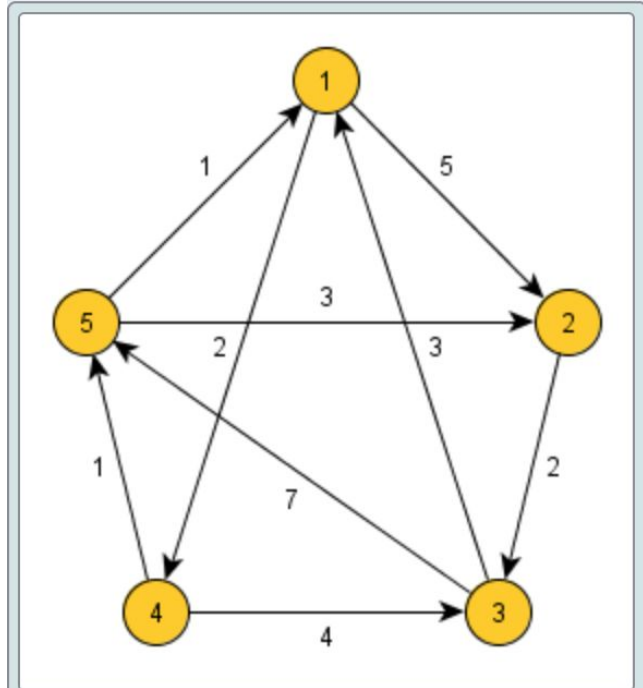
$$\begin{pmatrix} 0 & 5 & \infty & 2 & \infty \\ \infty & 0 & 2 & \infty & \infty \\ 3 & \infty & 0 & \infty & 7 \\ \infty & \infty & 4 & 0 & 1 \\ 1 & 3 & \infty & \infty & 0 \end{pmatrix}$$



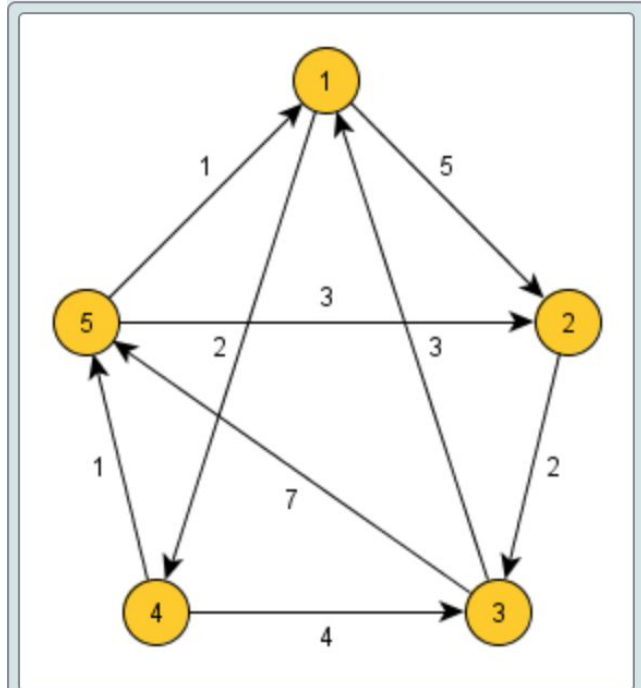
$$D_0 = \begin{pmatrix} 0 & 5 & \infty & 2 & \infty \\ \infty & 0 & 2 & \infty & \infty \\ 3 & \infty & 0 & \infty & 7 \\ \infty & \infty & 4 & 0 & 1 \\ 1 & 3 & \infty & \infty & 0 \end{pmatrix}$$



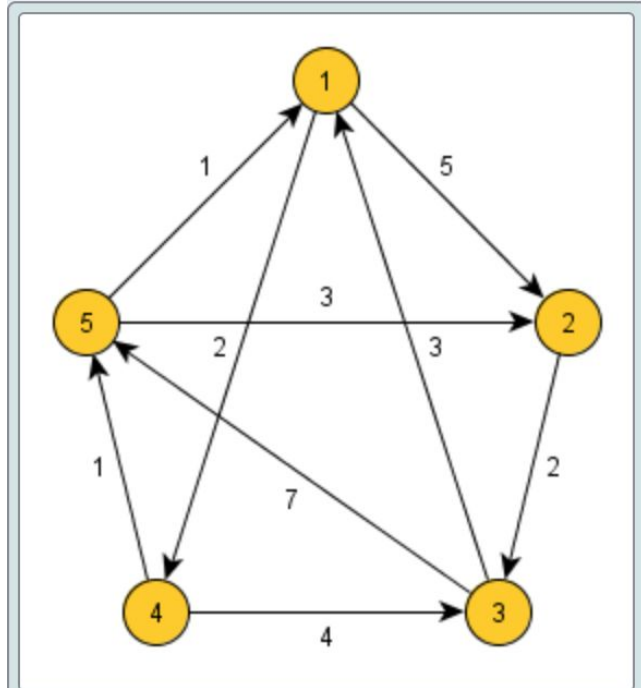
$$D_1 = \begin{pmatrix} 0 & 5 & \infty & 2 & \infty \\ \infty & 0 & 2 & \infty & \infty \\ 3 & 8 & 0 & 5 & 7 \\ \infty & \infty & 4 & 0 & 1 \\ 1 & 3 & \infty & 3 & 0 \end{pmatrix}$$



$$D_2 = \begin{pmatrix} 0 & 5 & 7 & 2 & \infty \\ \infty & 0 & 2 & \infty & \infty \\ 3 & 8 & 0 & 5 & 7 \\ \infty & \infty & 4 & 0 & 1 \\ 1 & 3 & 5 & 3 & 0 \end{pmatrix}$$



$$D_4 = \begin{pmatrix} 0 & 5 & 6 & 2 & 3 \\ 5 & 0 & 2 & 7 & 8 \\ 3 & 8 & 0 & 5 & 6 \\ 7 & 12 & 4 & 0 & 1 \\ 1 & 3 & 5 & 3 & 0 \end{pmatrix}$$



$$D_5 = \begin{pmatrix} 0 & 5 & 6 & 2 & 3 \\ 5 & 0 & 2 & 7 & 8 \\ 3 & 8 & 0 & 5 & 6 \\ 2 & 4 & 4 & 0 & 1 \\ 1 & 3 & 5 & 3 & 0 \end{pmatrix}$$

Johnsons Algorithm

- Kjører en kombinasjon av Bellman Ford og Dijkstra.
- Ingen negative kanter
 - Dijkstra med Fibonacci heap.
- Negative kanter
 - Gi nye vektorer til alle kantene - kjør Bellman-Ford og Dijkstra.

Kjøretid: $O(V^2 \lg V + VE)$

Bra for “sparse graphs”

SAMMENLIKNING

Algoritme	Kjøretid	OBS
Dag Shortest Path	$O(V+E)$	Kun på DAG
BFS	$O(V+E)$	Kun hvis alle kanter har samme vekt
Dijkstra	$O((V+E)\lg V)$	Ikke negative kanter
Bellman Ford	$O(VE)$	
Floyd-Warshall	$O(V^3)$	
Johnsons Algorithm	$O(V^2\lg V + VE)$	Bra for sparse graphs ($E <$