IN2010 - Algoritmer og datastrukturer

HØSTEN 2018

Institutt for informatikk, Universitetet i Oslo

Forelesning 6:

Grafer III

Dagens plan:

Dybde-først søk

- Biconnectivity
- Strongly connected components

All-pairs shortest paths

- Floy-Warshall

Biconnectivity

Biconnectivity

Definition

En sammenhengende urettet graf er bi-connected hvis det ikke er noen noder som ved fjerning gjør at grafen blir ikke sammenhengende. Slik node heter *cut-vertices* eller *separation vertices*.

Egenskapen ved biconnectede grafer:
Det er to disjunkte stier mellom hvilke to noder som helst.

F.eks. Nettverk-feiltoleranse: dersom en ruter går ned, finnes det alltid en alternativ vei å rute datapakkene på.

Viktig å identifisere disse nodene.
Single point to failure!

Biconnectivity

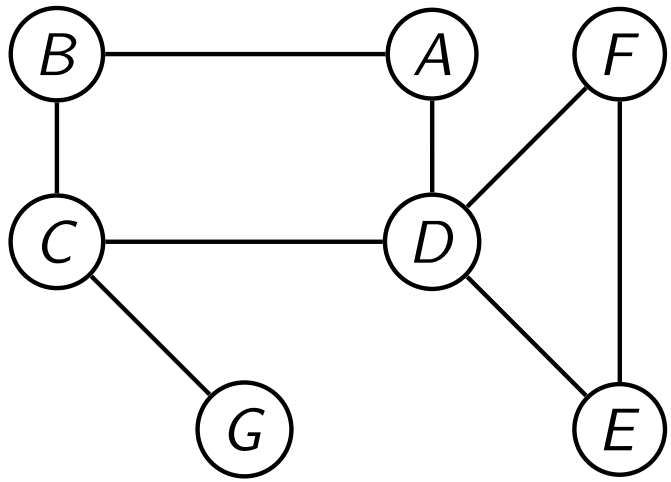
Definition

En sammenhengende urettet graf er bi-connected hvis det ikke er noen noder som ved fjerning gjør at grafen blir ikke sammenhengende. Slik node heter *cut-vertices* eller *separation vertices*.

Egenskapen ved biconnectede grafer:
Det er to disjunkte stier mellom hvilke to noder som helst.

F.eks. Nettverk-feiltoleranse: dersom en ruter går ned, finnes det alltid en alternativ vei å rute datapakkene på.

Viktig å identifisere disse nodene.
Single point to failure!



Biconnectivity

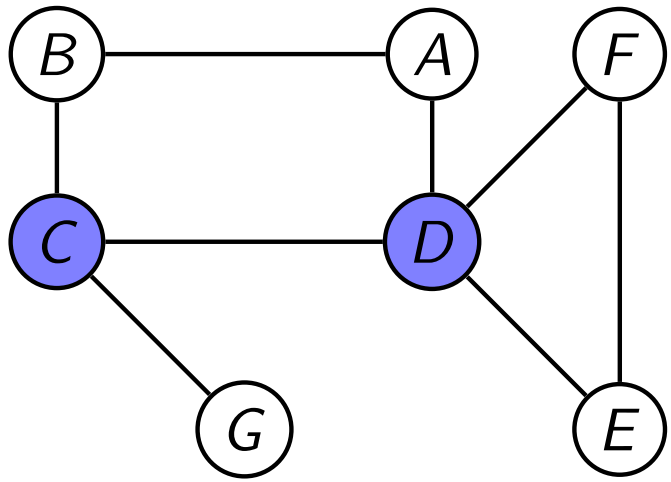
Definition

En sammenhengende urettet graf er bi-connected hvis det ikke er noen noder som ved fjerning gjør at grafen blir ikke sammenhengende. Slik node heter *cut-vertices* eller *separation vertices*.

Egenskapen ved biconnectede grafer:
Det er to disjunkte stier mellom hvilke to noder som helst.

F.eks. Nettverk-feiltoleranse: dersom en ruter går ned, finnes det alltid en alternativ vei å rute datapakkene på.

Viktig å identifisere disse nodene.
Single point to failure!

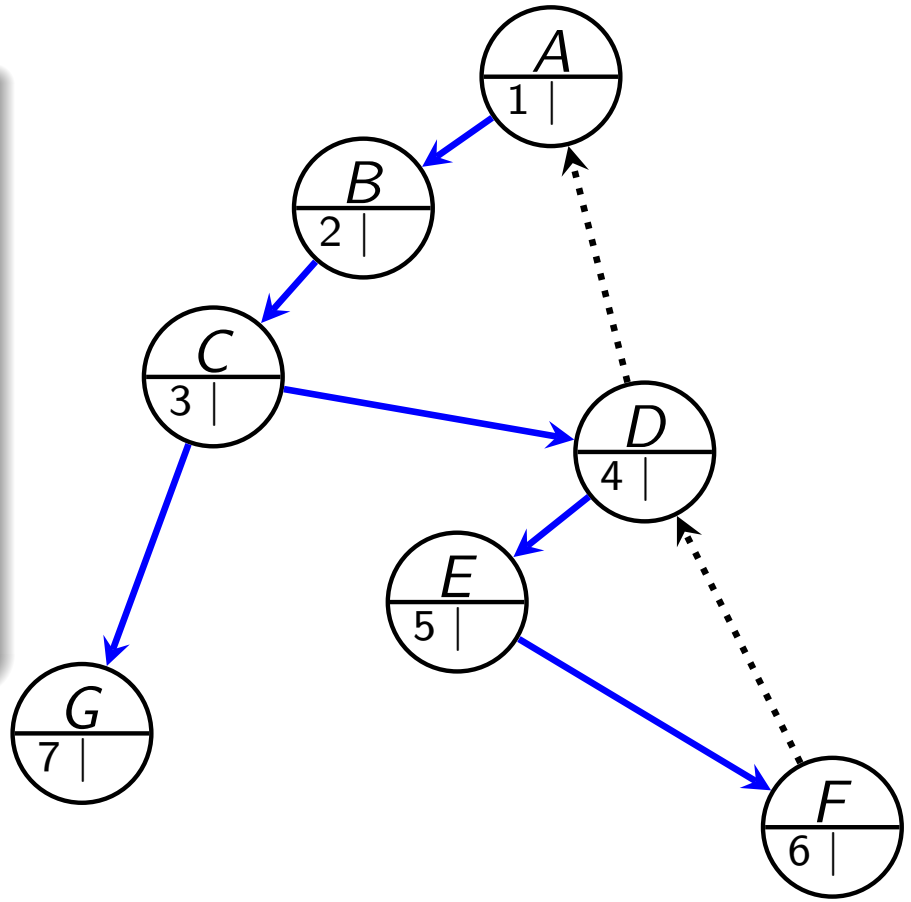


Er grafen bi-connected?

1. DFS-spenntre

Konstruer et dfs spenntre fra en node **v** av **G**

- Alle kantene (**v**, **w**) i **G** er representert i treet som enten en tree edge eller som en back edge
- Nummerer nodene når vi besøker dem

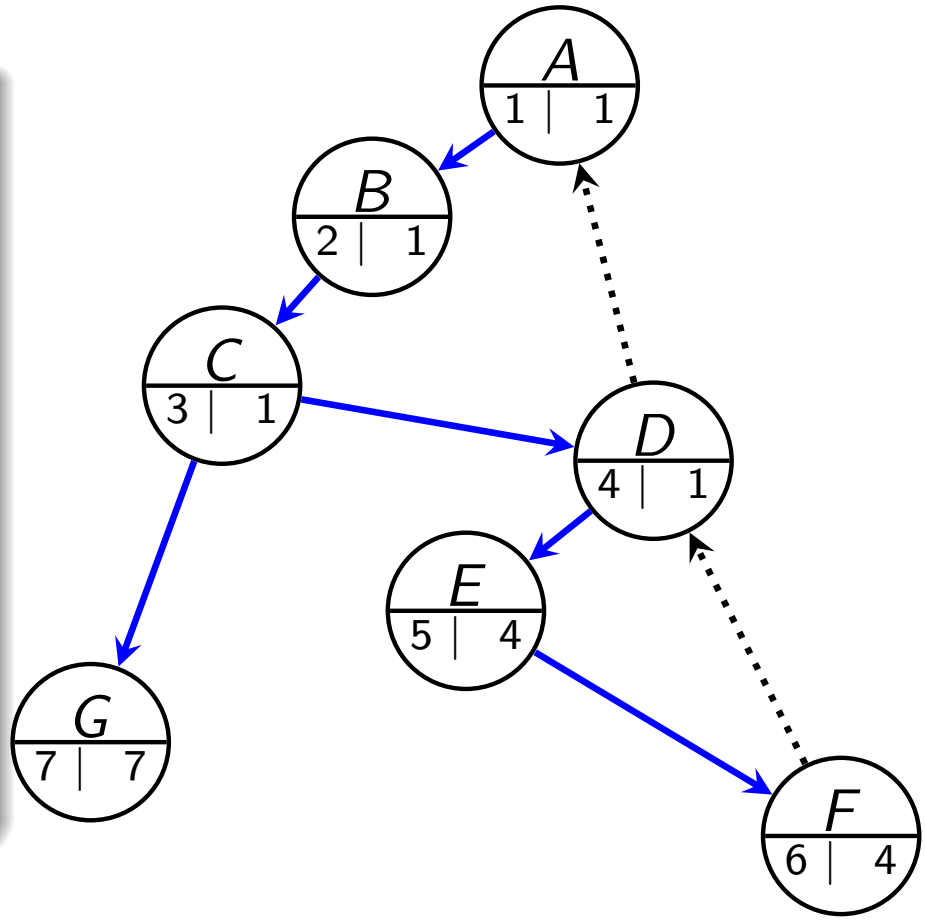


Er grafen bi-connected?

2. Low-number

for hver node i treet: beregn low-nummeret

- laveste noden som kan nås ved å ta 0 eller flere tree edge etterfulgt av 0 eller 1 back edge
- Dette gjør vi like før vi trekker oss tilbake (idet vi er ferdig med kallet)
- note: på dette tidspunktet har funnet low for alle barna til noden

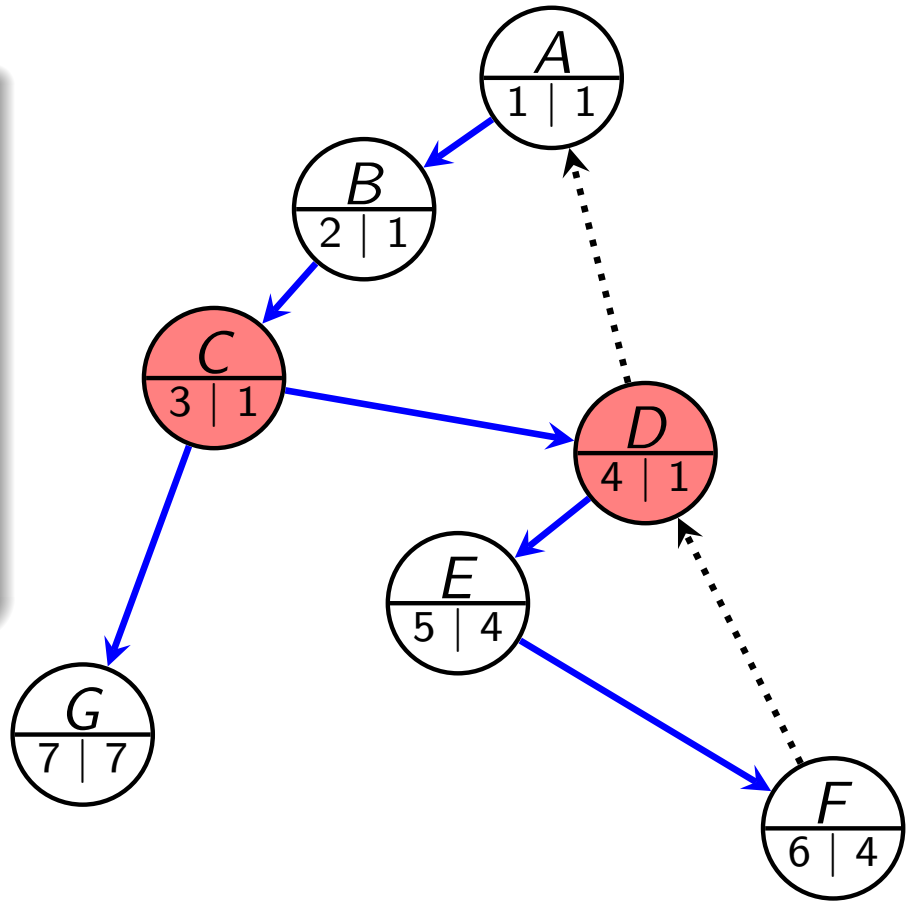


Er grafen bi-connected?

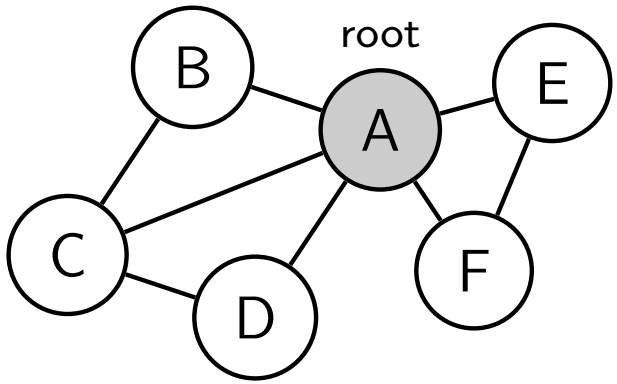
3. Cut-vertex

En node **v** er en cutvertex i **G** hvis:

1. **v** er rotnoden av $DFS_{\mathbf{G}}$-treet og har to eller flere tree edges
2. **v** *ikke* er rotnoden av $DFS_{\mathbf{G}}$ og det finnes en tree edge $(\mathbf{v}, \mathbf{w})$ slik at $\text{low}(w) \geq \text{num}(v)$



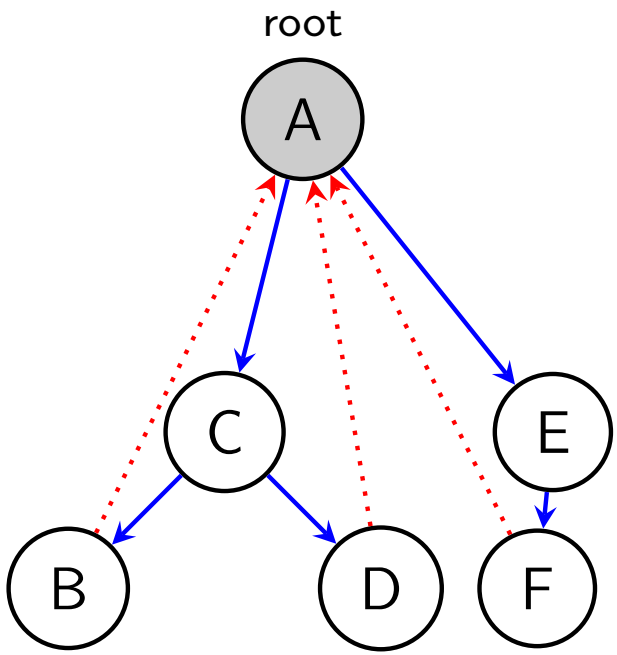
Rot



Roten av DFS-treet er en cutvertex hvis den har to eller flere utgående tree edges.

- tilbaketrekking må gå gjennom rooten for å komme fra den ene til den andre sub-treet.

Rot

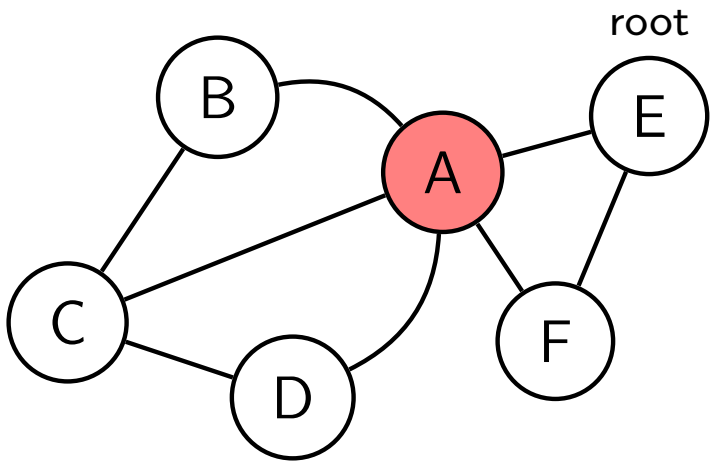


Roten av DFS-treet er en cutvertex hvis den har to eller flere utgående tree edges.

- tilbaketrekking må gå gjennom rooten for å komme fra den ene til den andre sub-treet.

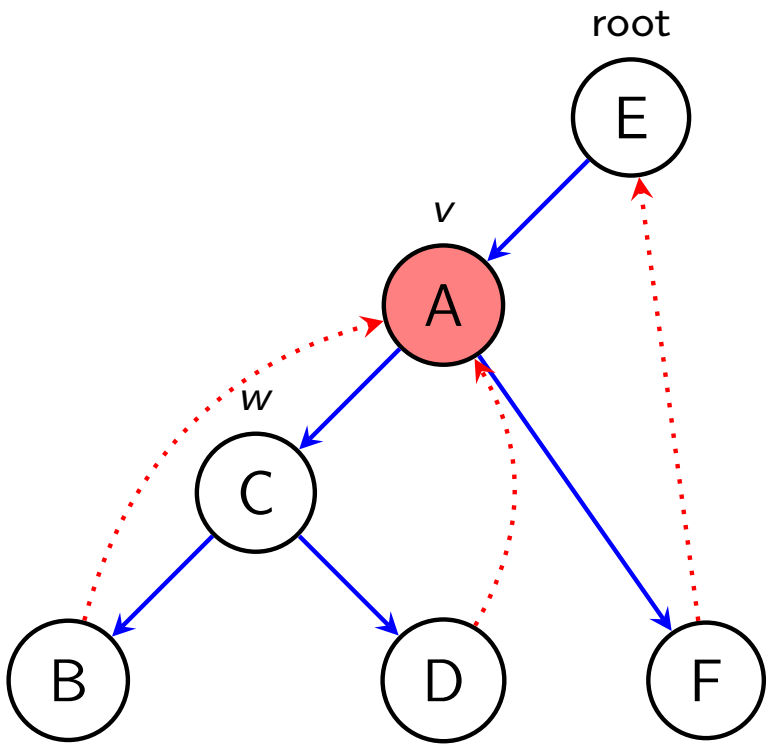
Ikke-rot node

Et ikke-rot node **v** er en cutvertex hvis den har et barn **w** slik at *ingen back edge* som starter i subtreet av **w** når en *predesessornode* til **v**.



Ikke-rot node

Et ikke-rot node **v** er en cutvertex hvis den har et barn **w** slik at *ingen back edge* som starter i subtreet av **w** når en *predesessornode* til **v**.



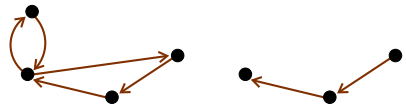
```
void dfs(v)
  {
    v.num    = counter++;          //
    v.status = visited;            // I am visiting now
    for each w adjacent to v
      if w.status ≠ visited
        w.parent  = v;             // remember parent
        dfs (w)                    // recur
  }
```

```
void assignlow(Node v) {
      v.low = v.num;                          // at least num
      for each w adjacent to v  {             // neighbors in G!
        if (w.num > v.num)                    //forward edge
        {
            assignlow(w);
            if (w.low ≥ v.num)
                system.out.println(v +
                        ''an articulation point!'');
            v.low = min(v.low, w.low);
        }
        else                                  // w.num ≤ v.num
            if (v.parent ≠ w) //back edge
                v.low = min (v.low, w.num);
      }
   }
```

- test for root not shown
- the two traversals (dfs + `assignlow`) can be combined into 1 pass.

Strongly Connected Components

rettet graf & DFS



En rettet graf er sterkt sammenhengende hvis og bare hvis vi fra hver eneste node **v** klarer å besøke alle de andre nodene i grafen ved et dybde-først søk fra **v**

Strongly Connected Components (SCC)

partisjonere G i såkalt strongly connected components

Definition (SCC)

Gitt en rettet graf $G = (V, E)$. En *strongly connected component* av G er en maksimal sett av noder $U \subseteq V$ s.t.: for alle $u_1, u_2 \in U$ vi har at $u_1 \rightarrow^* u_2$ and $u_2 \rightarrow^* u_1$.

Hvordan sjekker om en graf er SC?
Tanken er:
Hvis v kan nå alle noder og alle noder kan nå v, så kan alle nå alle ved å gå gjennom v.

Strongly Connected Components (SCC)

partisjonere G i såkalt strongly connected components

Definition (SCC)

Gitt en rettet graf $G = (V, E)$. En *strongly connected component* av G er en maksimal sett av noder $U \subseteq V$ s.t.: for alle $u_1, u_2 \in U$ vi har at $u_1 \rightarrow^* u_2$ and $u_2 \rightarrow^* u_1$.

Hvordan sjekker om en graf er SC?
Tanken er:
Hvis v kan nå alle noder og alle noder kan nå v, så kan alle nå alle ved å gå gjennom v.

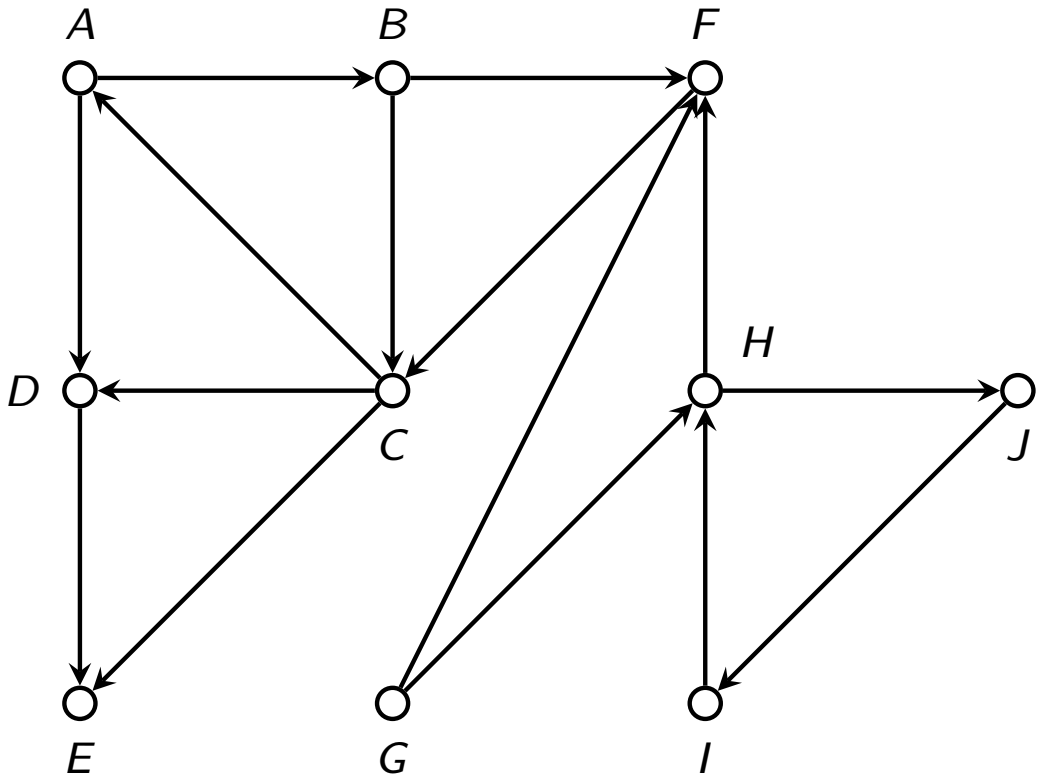
- Ide: G og G^t har de samme SCCs $\implies$ bruk dfs 2 ganger, en gang på G og en gang på den reverserte grafen[1] G^t
- kompleksitet: lineær tid $\mathcal{O}(E + V)$

[1] Gitt $G = (V, E)$ da er $G^t = (V, E^t)$ hvor $(v, u) \in E^t$ iff $(u, v) \in E$

SCC

1. DFS på G

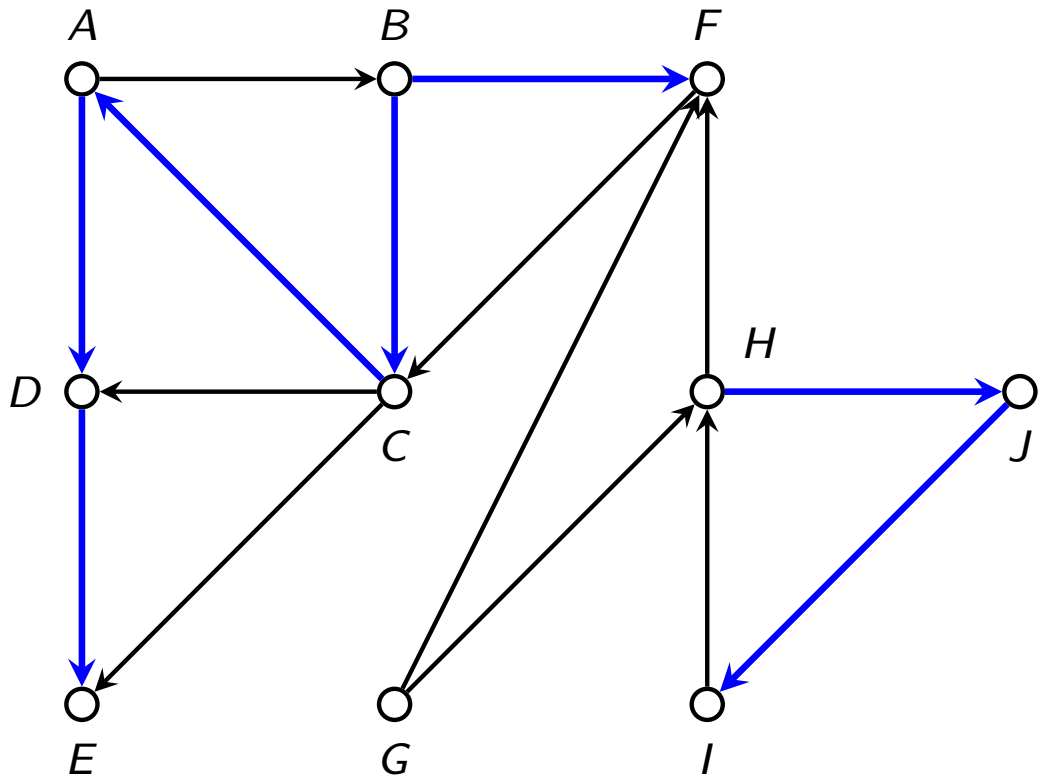
1. gjør DFS traversering
2. husker post-order (`finished` time stamp)



SCC

1. DFS på G

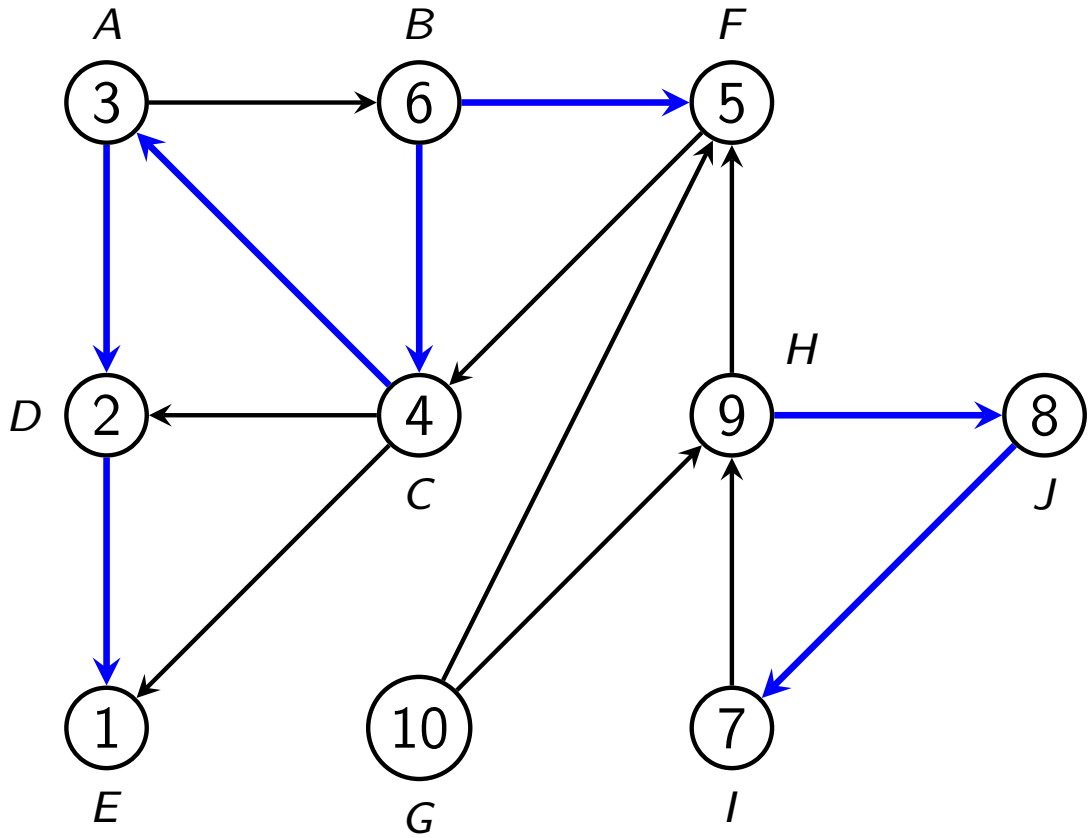
1. gjør DFS traversering
2. husker post-order (`finished` time stamp)



SCC

1. DFS på G

1. gjør DFS traversering
2. husker post-order (`finished` time stamp)



SCC

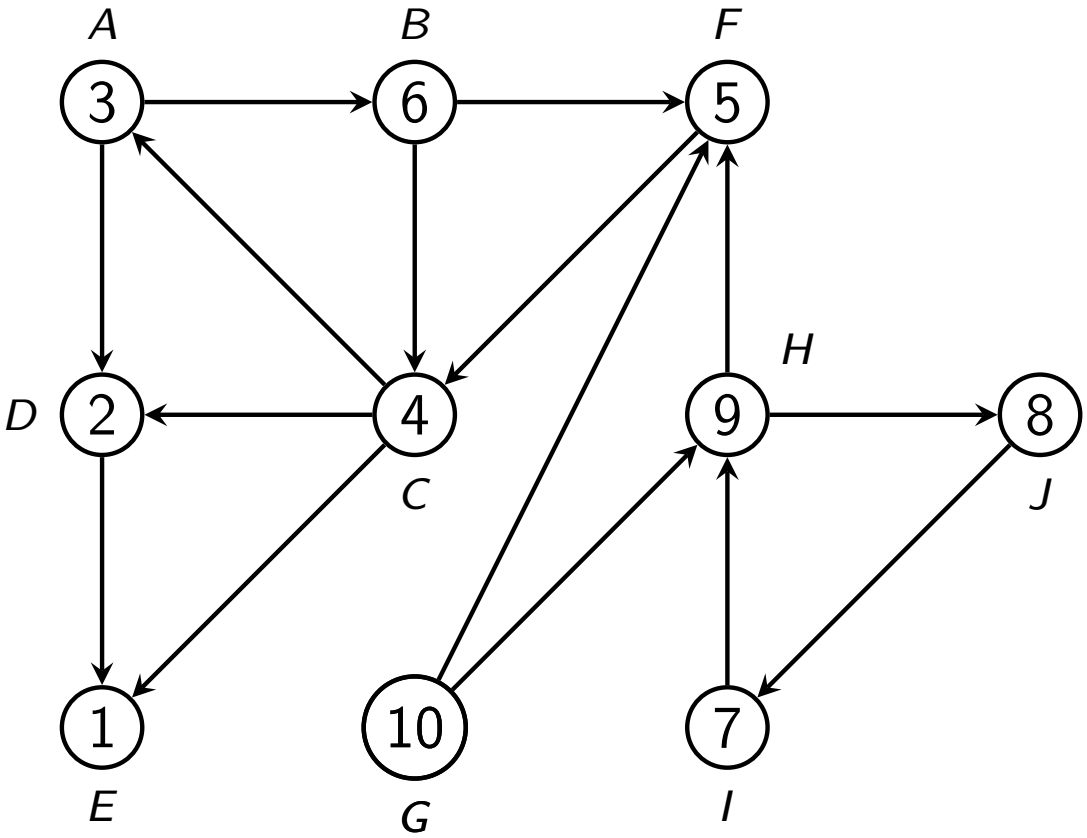
2. Reversere

reversere kantene til G og får G^t

3. DFS på G^t

iterere DFS på G^t i avtagende rekkefølger!

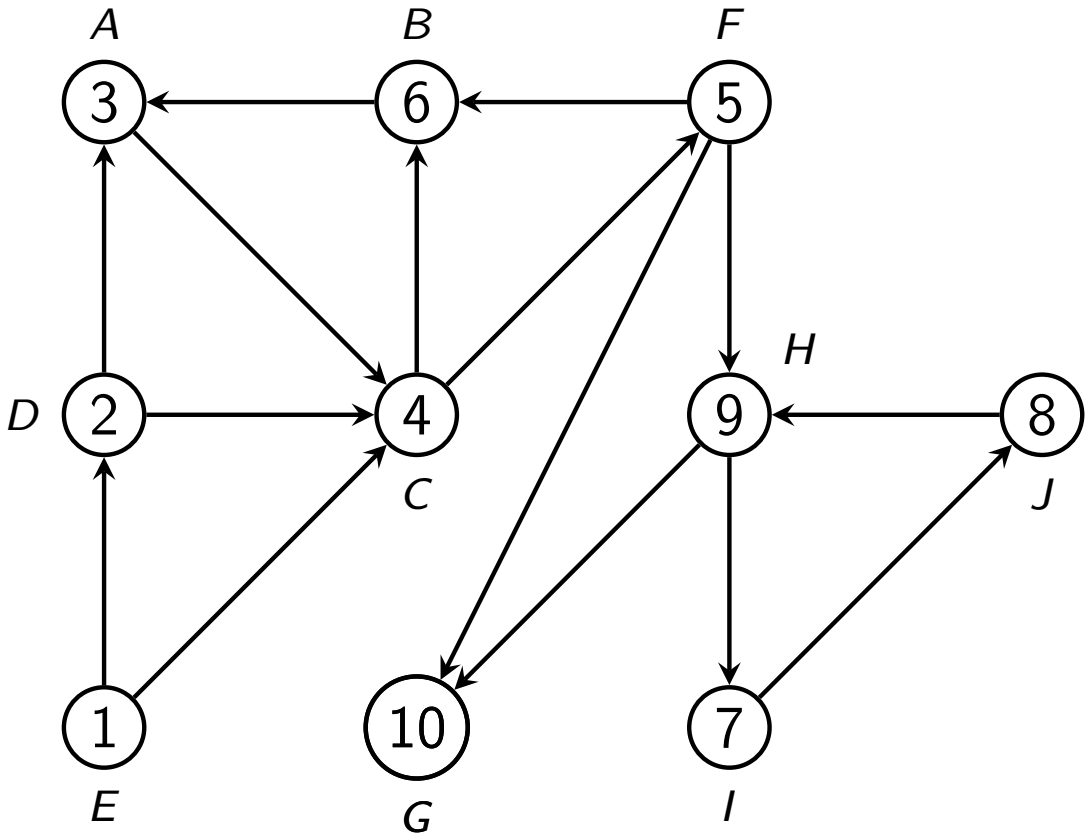
Andre fase (1)



strongly connected components:

{}

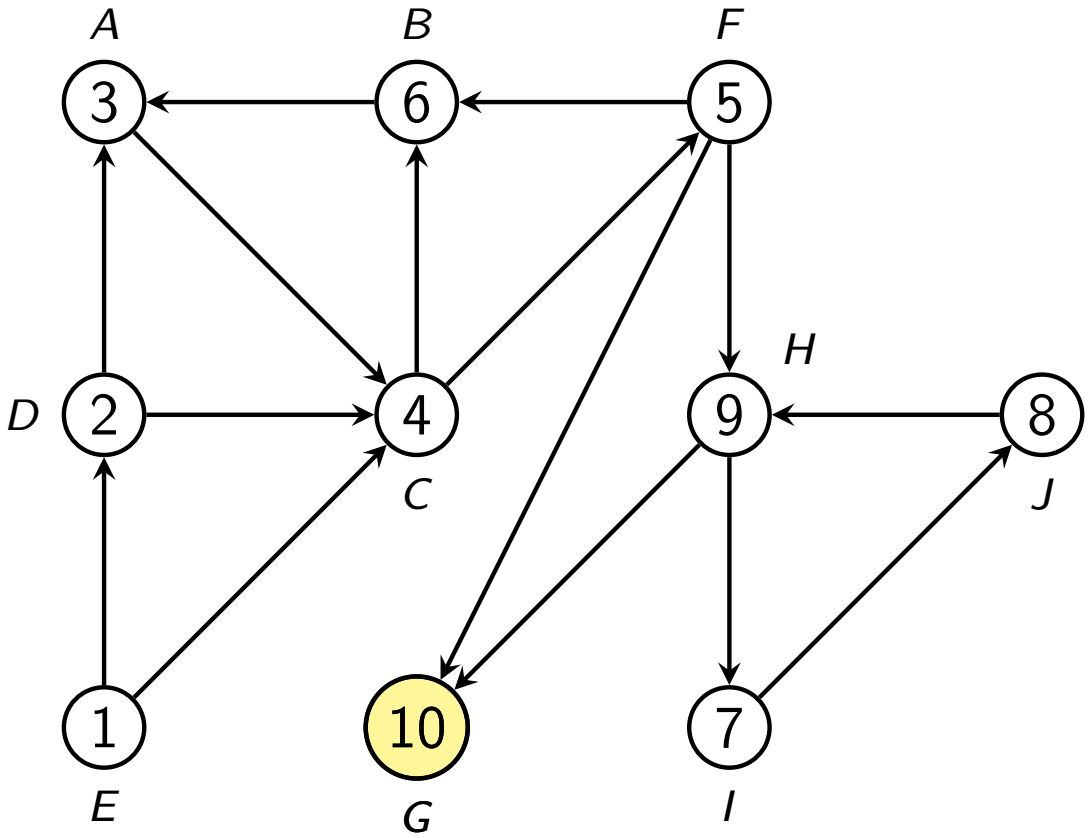
Andre fase (2)



strongly connected components:

{}

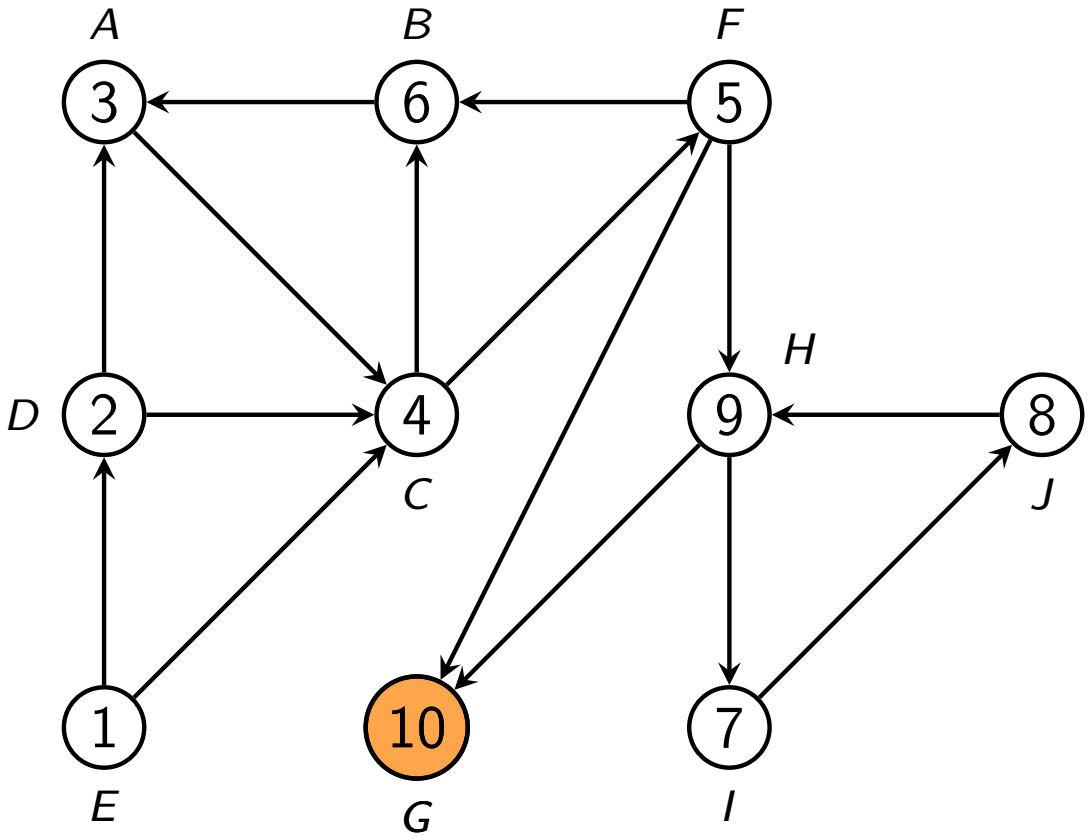
Andre fase (3)



strongly connected components:

{}

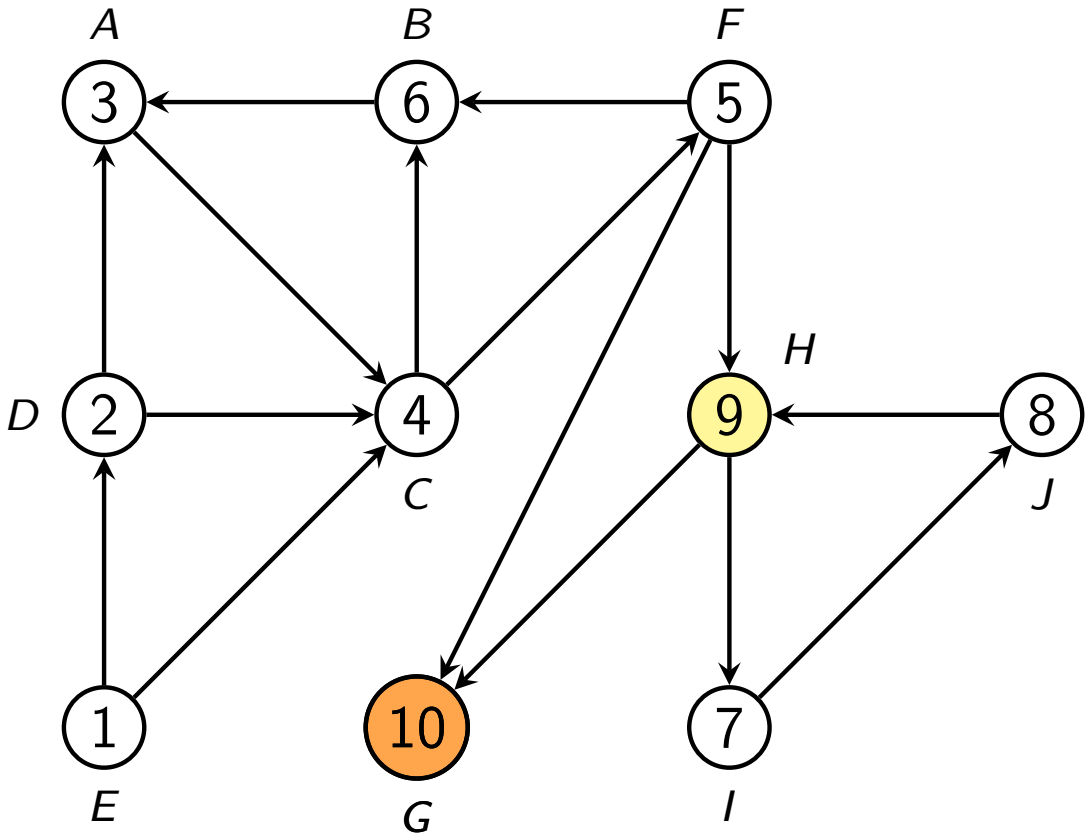
Andre fase (4)



strongly connected components:

$$\{\{G\}\}$$

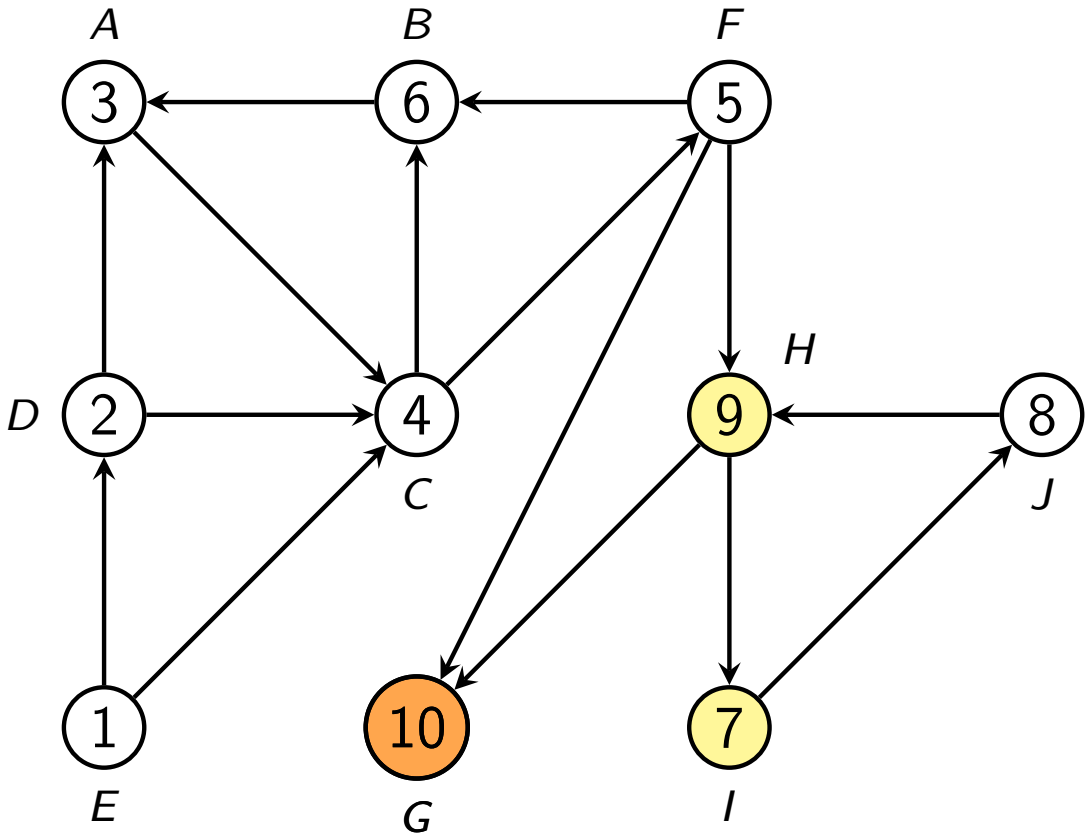
Andre fase (5)



strongly connected components:

$$\{\{G\}\}$$

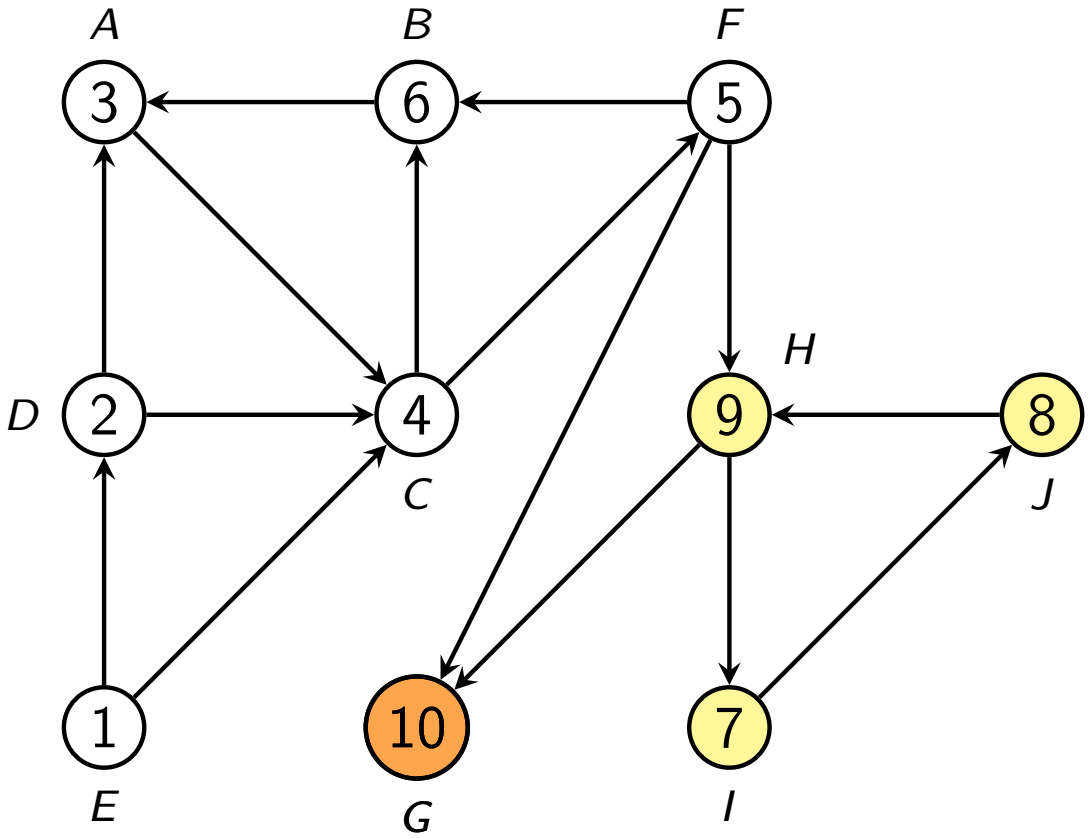
Andre fase (6)



strongly connected components:

$$\{\{G\}\}$$

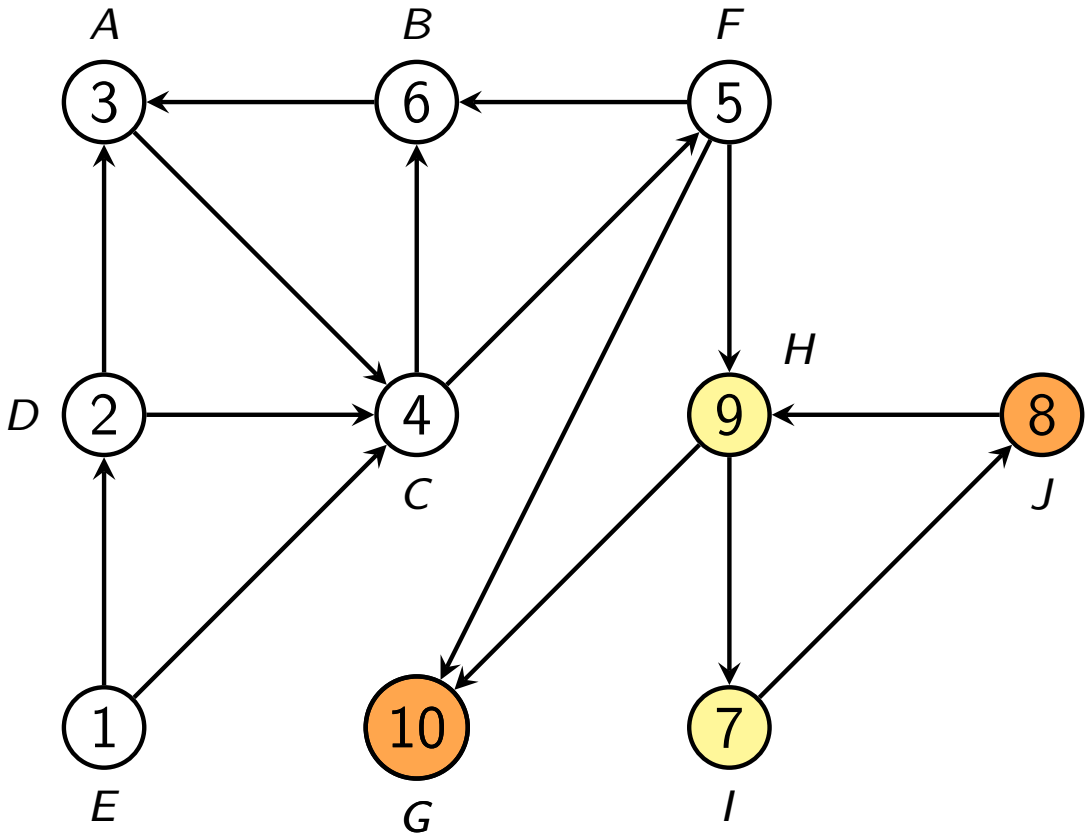
Andre fase (7)



strongly connected components:

$$\{\{G\}\}$$

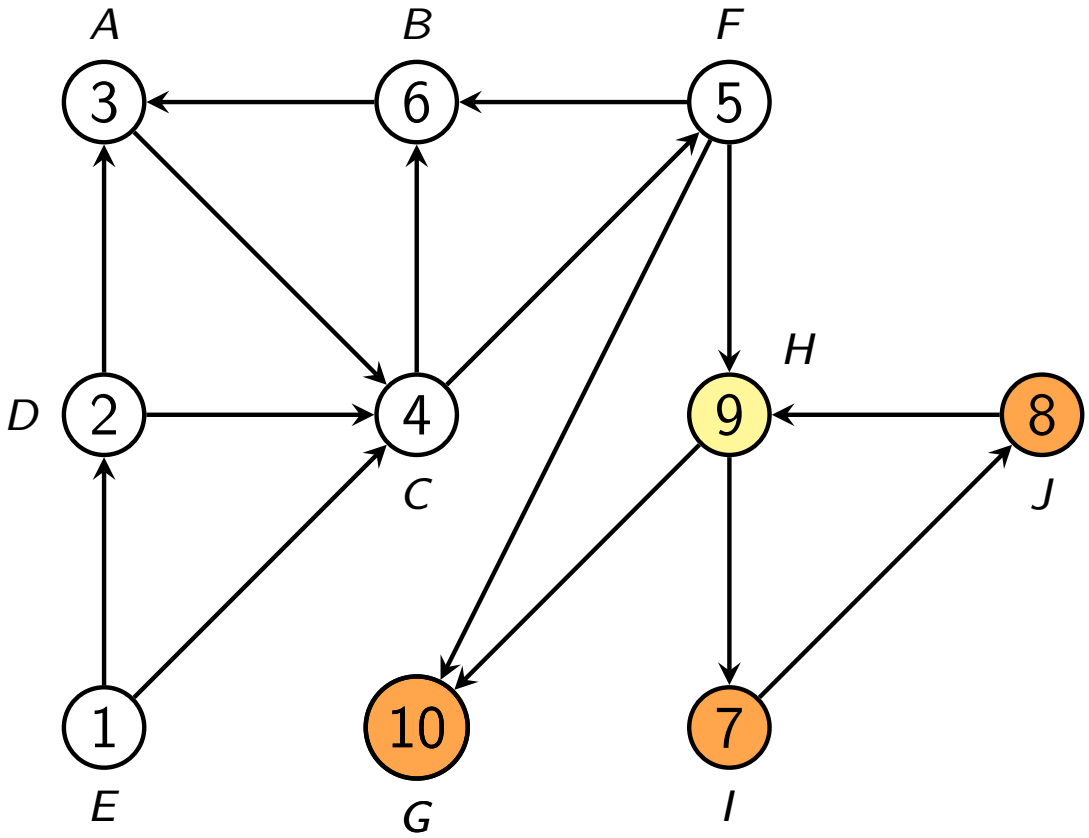
Andre fase (8)



strongly connected components:

$$\{\{G\}\}$$

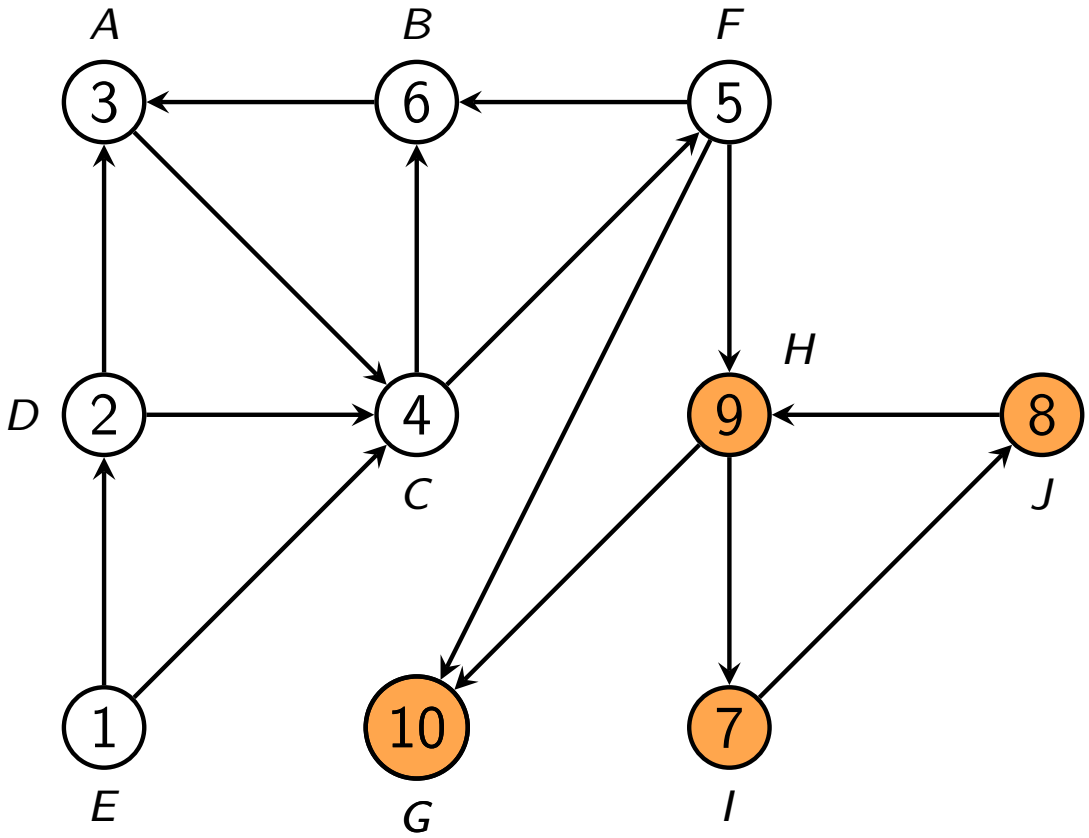
Andre fase (9)



strongly connected components:

$$\{\{G\}\}$$

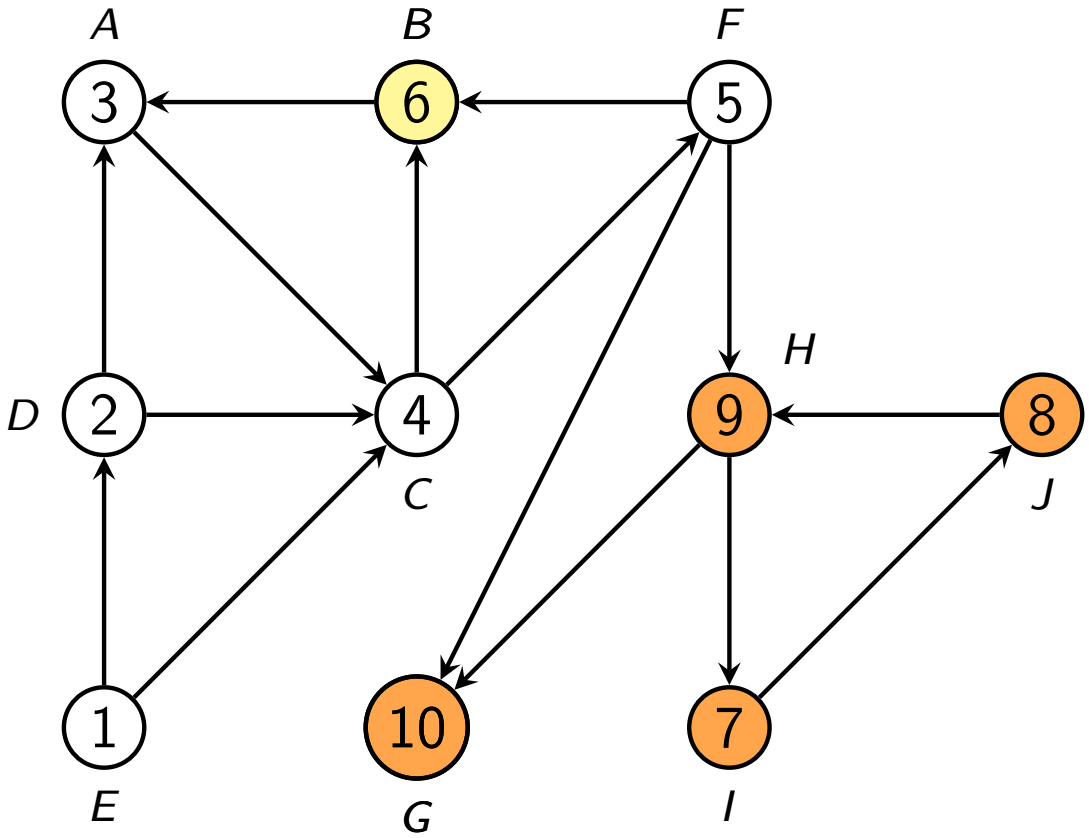
Andre fase (10)



strongly connected components:

$$\{\{G\}, \{H, I, J\}\}$$

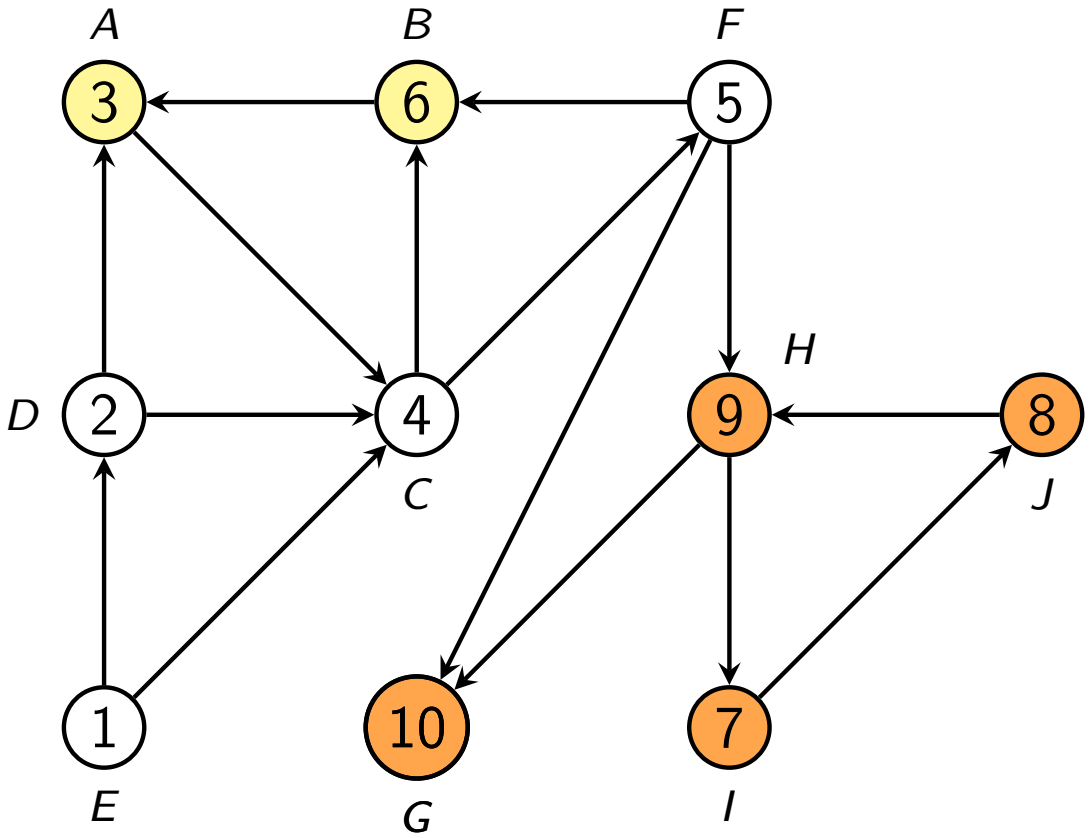
Andre fase (11)



strongly connected components:

$$\{\{G\}, \{H, I, J\}\}$$

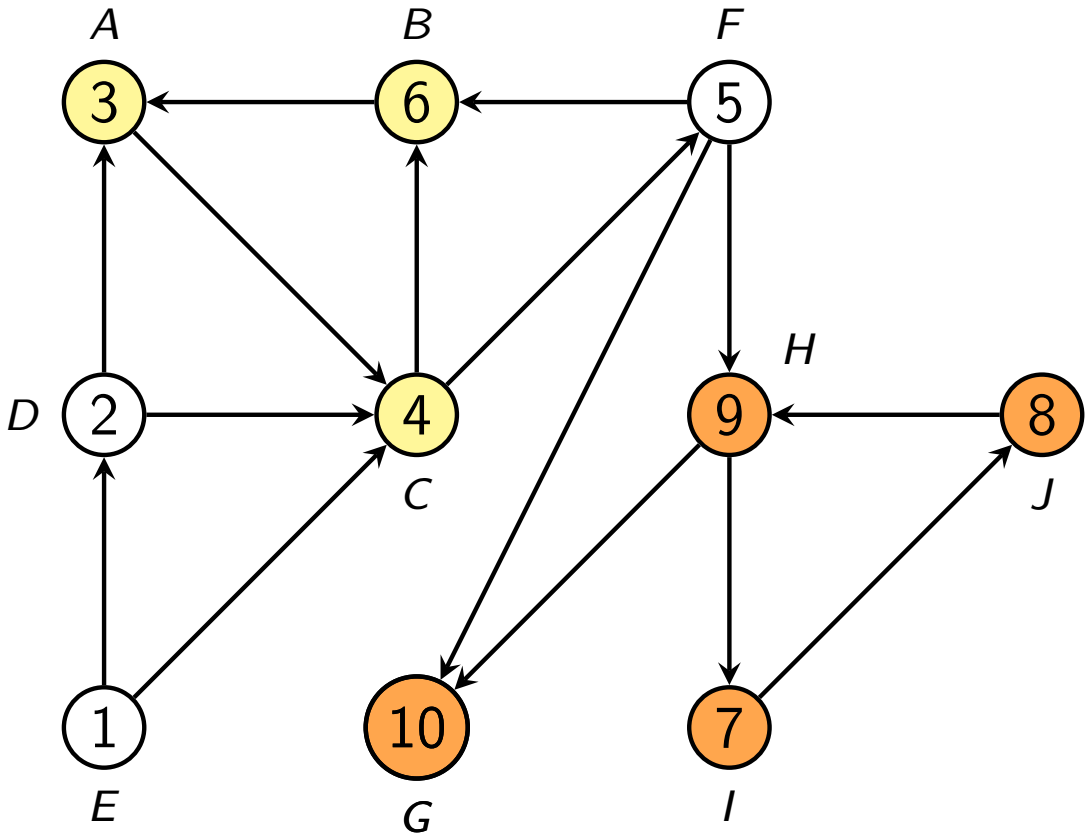
Andre fase (12)



strongly connected components:

$$\{\{G\}, \{H, I, J\}\}$$

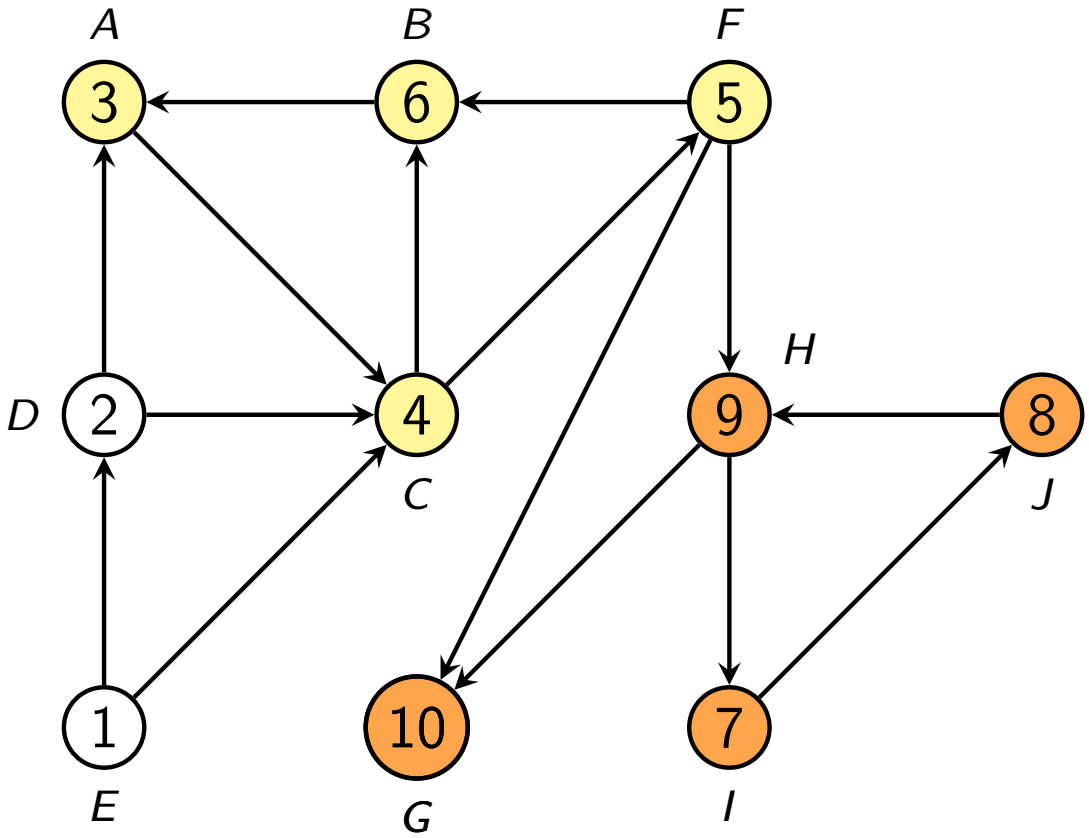
Andre fase (13)



strongly connected components:

$$\{\{G\}, \{H, I, J\}\}$$

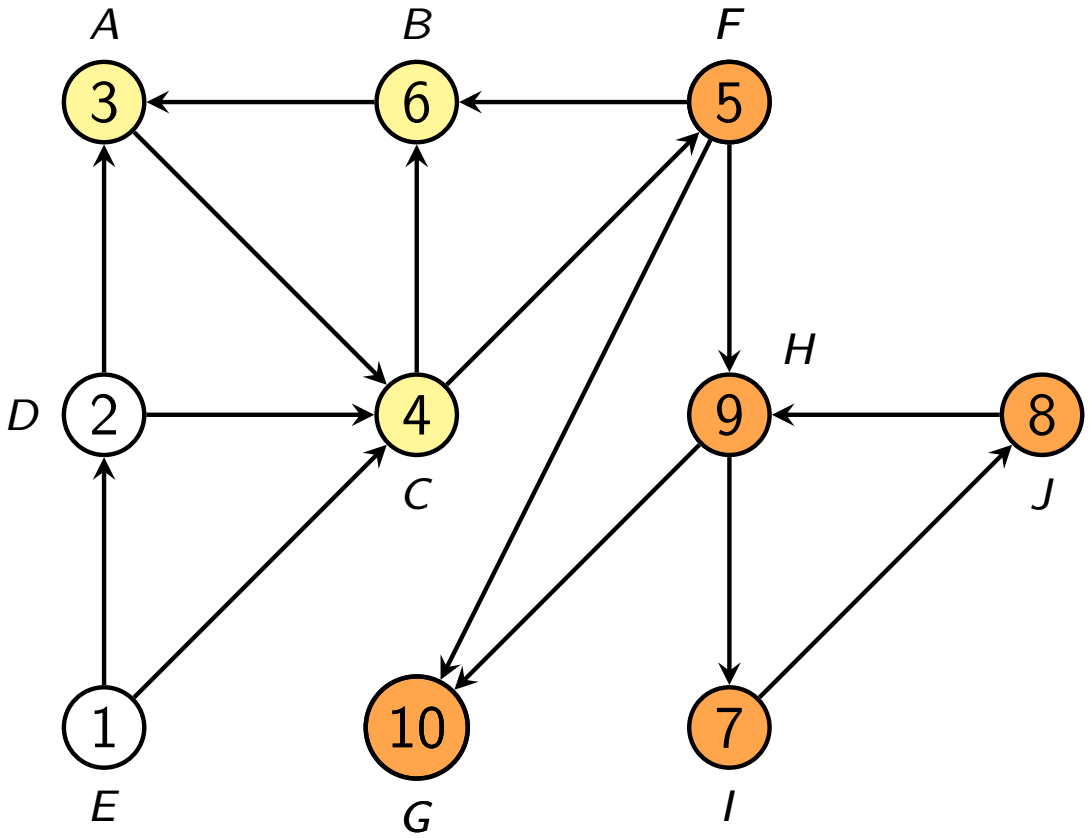
Andre fase (14)



strongly connected components:

$$\{\{G\}, \{H, I, J\}\}$$

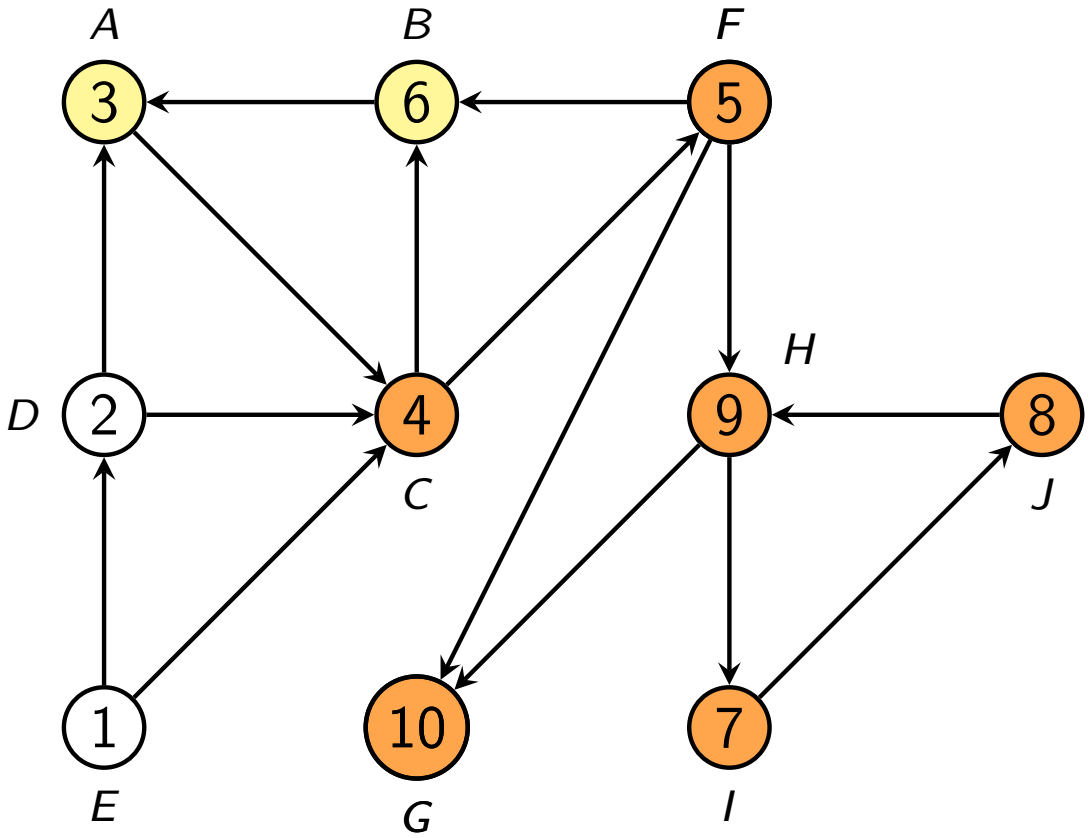
Andre fase (15)



strongly connected components:

$$\{\{G\}, \{H, I, J\}\}$$

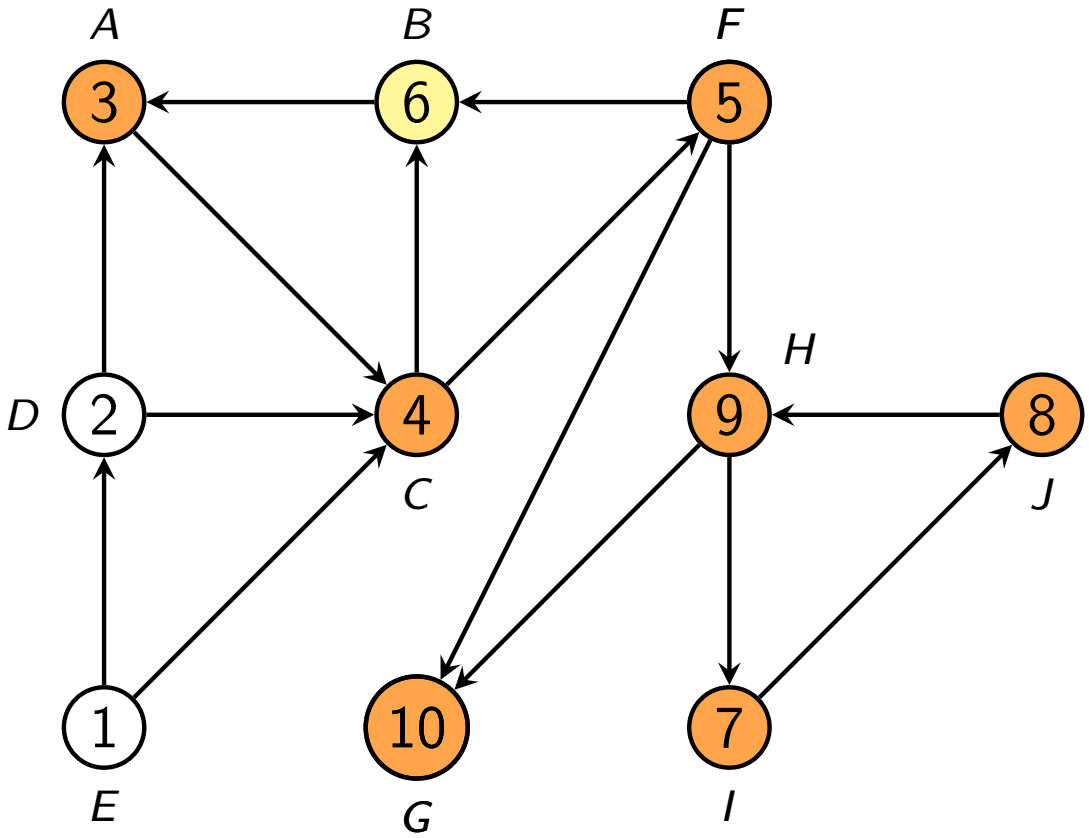
Andre fase (16)



strongly connected components:

$$\{\{G\}, \{H, I, J\}\}$$

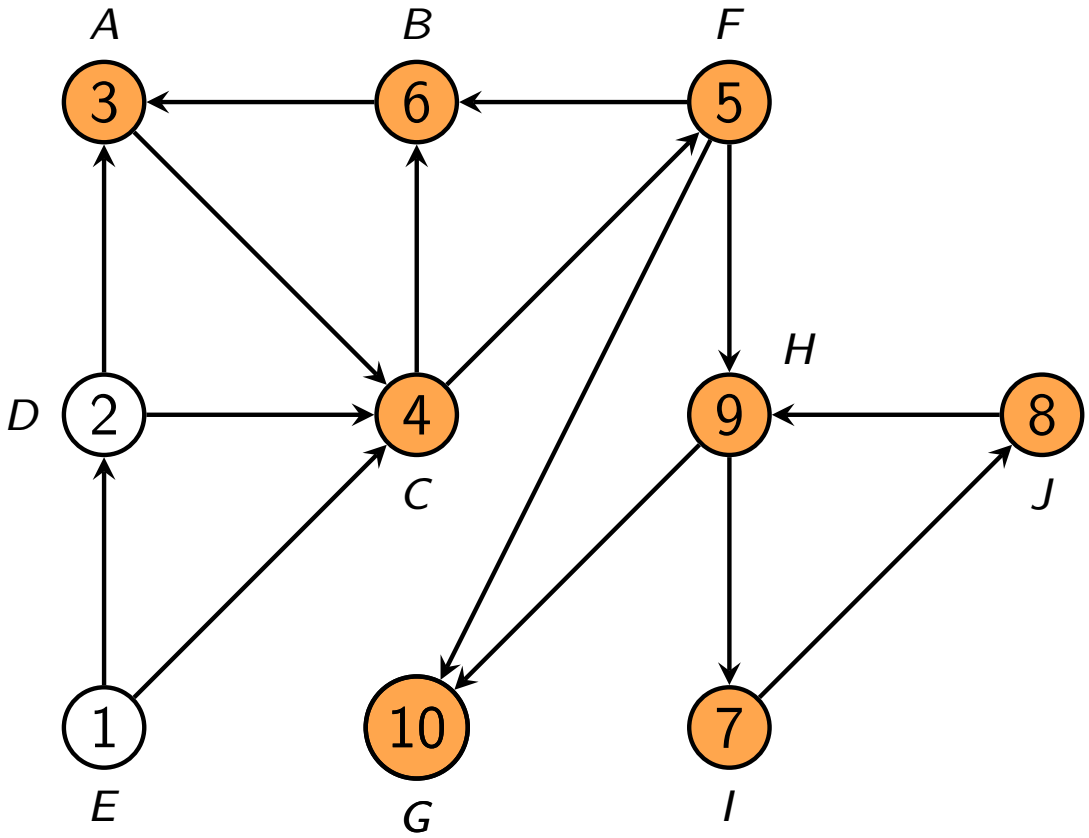
Andre fase (17)



strongly connected components:

$$\{\{G\}, \{H, I, J\}\}$$

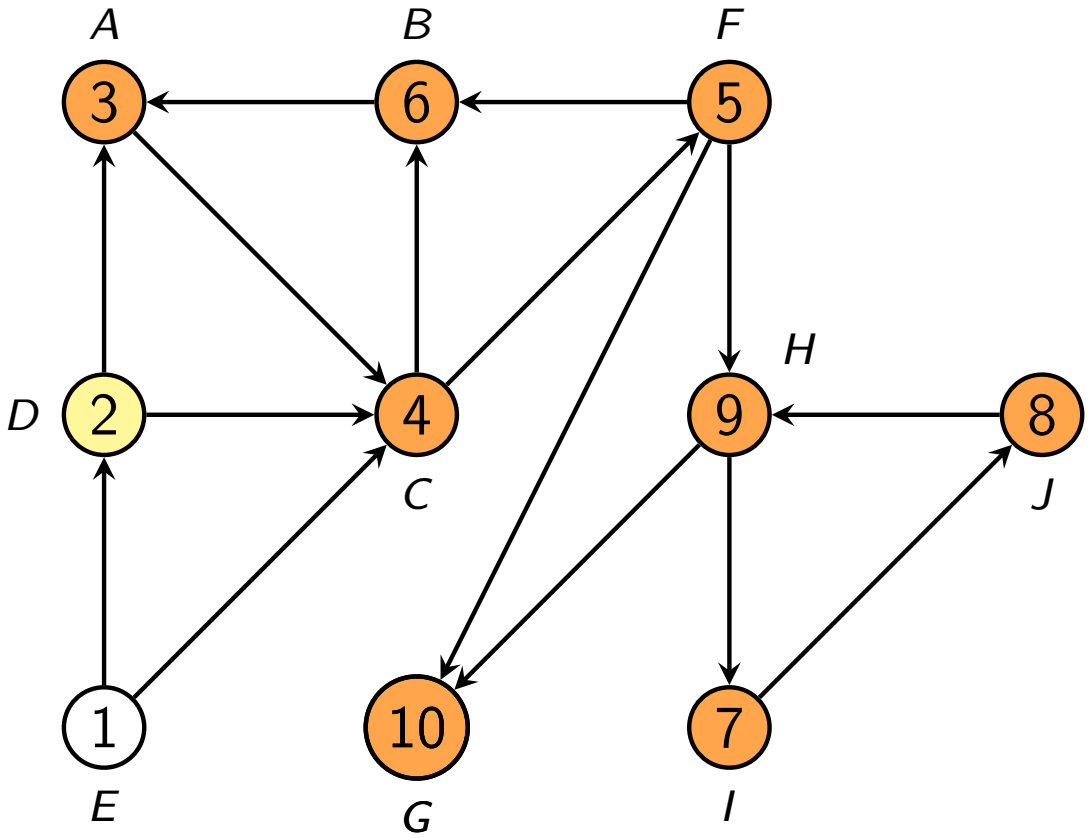
Andre fase (18)



strongly connected components:

$$\{\{G\},\{H,I,J\},\{B,A,C,F\}\}$$

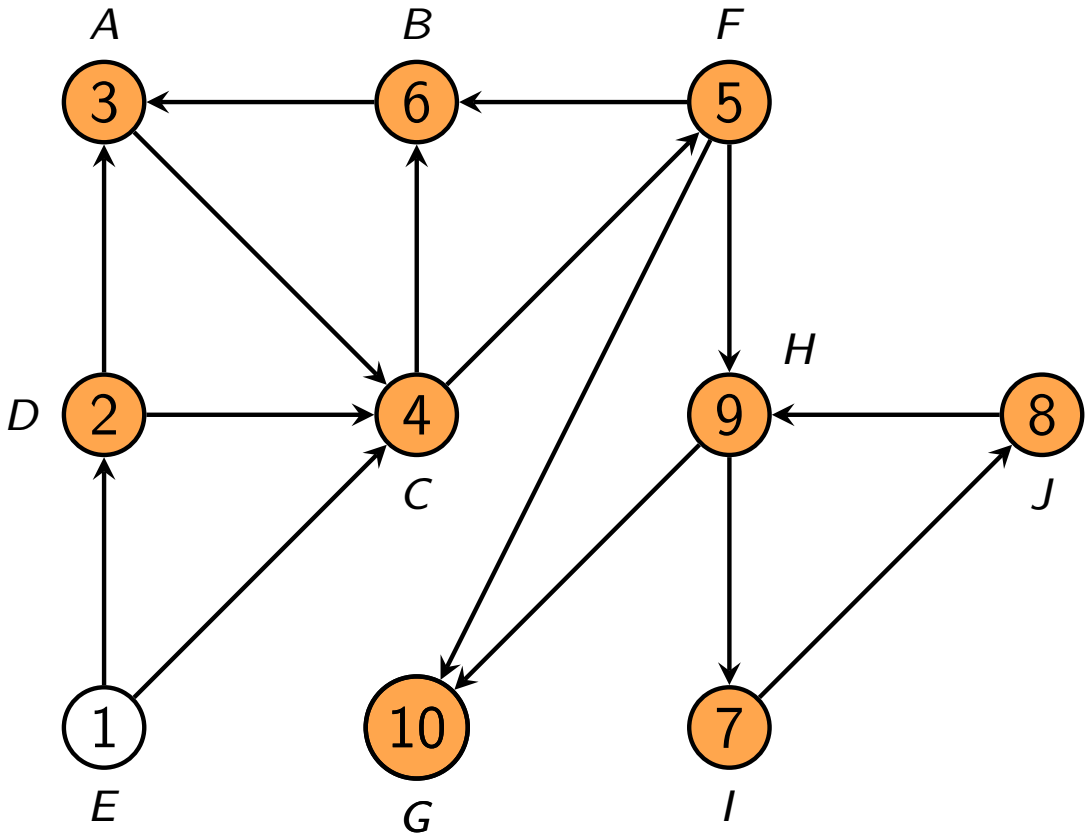
Andre fase (19)



strongly connected components:

$$\{\{G\},\{H,I,J\},\{B,A,C,F\}\}$$

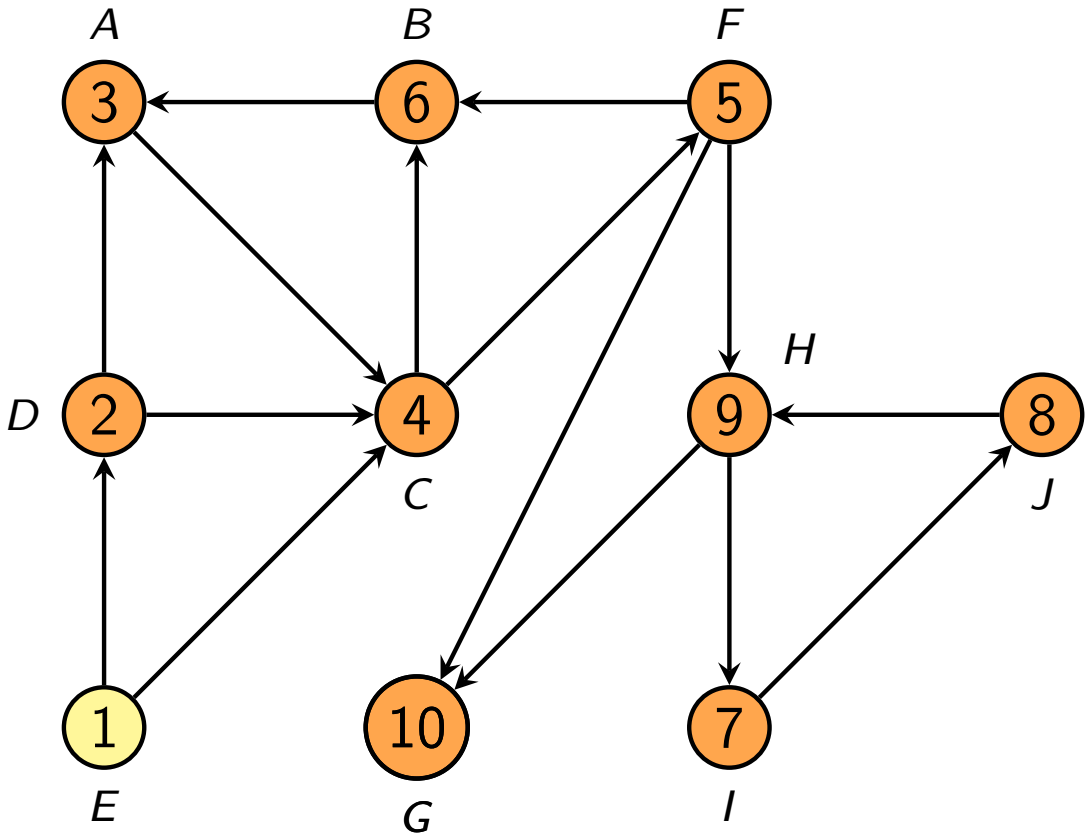
Andre fase (20)



strongly connected components:

$$\{\{G\},\{H,I,J\},\{B,A,C,F\},\{D\}\}$$

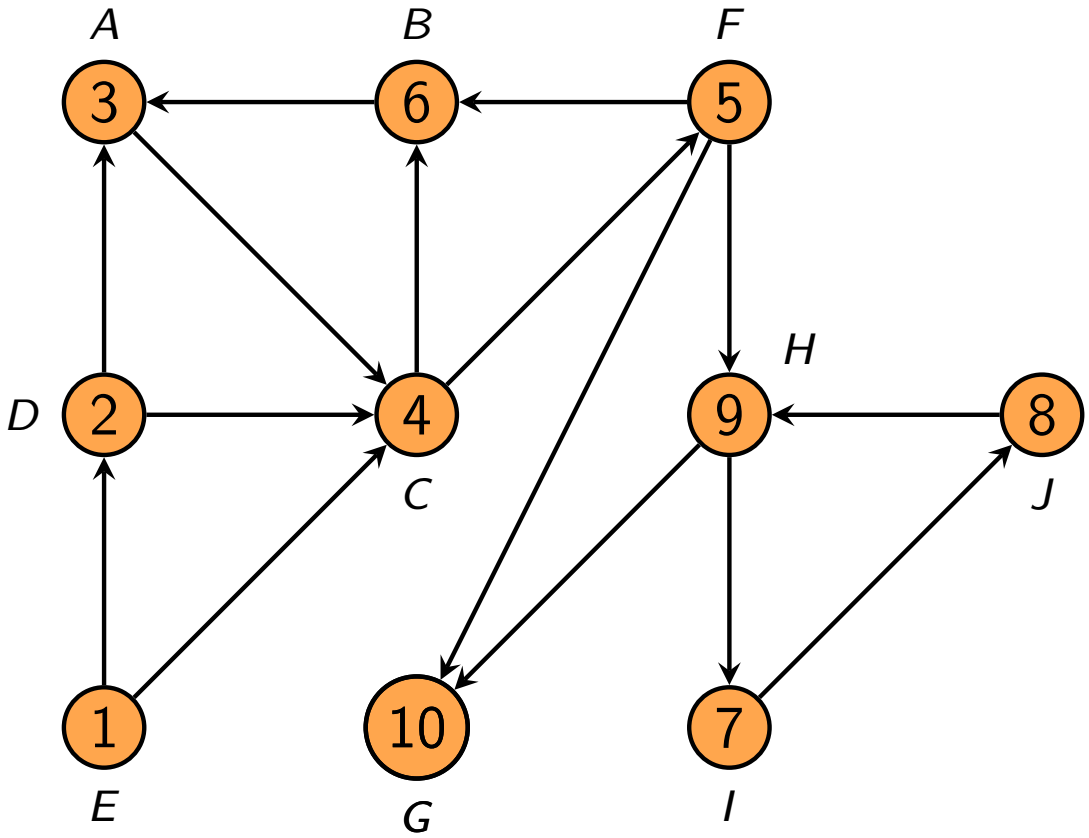
Andre fase (21)



strongly connected components:

$$\{\{G\},\{H,I,J\},\{B,A,C,F\},\{D\}\}$$

Andre fase (22)



strongly connected components:

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

Argument

Trenger å vise

v og w er i den samme DFS treet av G^t. Da har vi $v \longrightarrow^* w \longrightarrow^* v$

Argument

Trenger å vise

La x være roten av en dfs tre av G^t og 2 noder v og w i det samme treet av G^t. Da har vi at

$$x \longrightarrow^* v \longrightarrow^* x \quad \text{and} \quad x \longrightarrow^* w \longrightarrow^* x$$

- x er en rot
  - $\Rightarrow$ $x \longrightarrow^* v$ in G^t
  - $\Rightarrow$ $v \longrightarrow^* x$ in G
- x har høyere post-order nummer enn v
  - $\Rightarrow$ x ble ferdig senere enn v i den første dybde-først traversering av G
  - $\Rightarrow$ v må være en etterkommer av x ellers vil v bli ferdig etter x.
  - $\Rightarrow$

$$x \longrightarrow^* v \in G$$

Floyds algoritme

Dynamisk programmering

- Brukes først og fremst når vi ønsker optimale løsninger
- Må kunne dele det globale problemet i *delproblemer*
  - Disse løses typisk ikke-rekursivt ved å lagre del-løsningene i en tabell
- En optimal løsning på det globale problemet må være en sammensetning av optimale løsninger på (noen av) delproblemene

- Floyds algoritme for å finne korteste vei alle-til-alle i en rettet graf

- Enkle delproblemer:
  Det må finnes en måte å dele problemet opp i delproblemer som er enkle å beskrive
- Delproblem-optimalisering:
  En optimal løsning på det globale problemet må kunne settes sammen av optimale løsninger på delproblemene
- Overlapp av delproblemer:
  Optimale løsninger av urelaterte problemer kan inneholde felles delproblemer

Korteste vei alle-til-alle

Vi ønsker å beregne den korteste veien mellom ethvert par av noder i en *rettet*, *vektet* graf

Grunnleggende idé

Hvis det går en vei fra node i til node k med lengde ik, og en vei fra node k til node j med lengde kj, så går det en vei fra node i til node j med lengde $ik + kj$

- Floyds algoritme:
  Denne betraktningen gjentas systematisk for alle tripler i, k og j:
  - Initielt: Avstanden fra node i til node j settes lik vekten på kanten fra i til j, uendelig hvis det ikke går noen kant fra i til j
  - Trinn 0: Se etter forbedringer ved å velge node 0 som mellomnode
  - Etter trinn k: Avstanden mellom to noder er den korteste veien som bare bruker nodene 0, 1, . . . , k som mellomnoder

Korteste vei alle-til-alle

Vi ønsker å beregne den korteste veien mellom ethvert par av noder i en *rettet*, *vektet* graf

Grunnleggende idé

Hvis det går en vei fra node ***i*** til node ***k*** med lengde ***ik***, og en vei fra node ***k*** til node ***j*** med lengde ***kj***, så går det en vei fra node ***i*** til node ***j*** med lengde ***ik*** + ***kj***

- Floyds algoritme:
  Denne betraktningen gjentas systematisk for alle tripler i, k og j:
  - Initielt: Avstanden fra node i til node j settes lik vekten på kanten fra i til j, uendelig hvis det ikke går noen kant fra i til j
  - Trinn 0: Se etter forbedringer ved å velge node 0 som mellomnode
  - Etter trinn k: Avstanden mellom to noder er den korteste veien som bare bruker nodene $0, 1, \ldots, k$ som mellomnoder

Korteste vei alle-til-alle

Vi ønsker å beregne den korteste veien mellom ethvert par av noder i en *rettet, vektet* graf

Grunnleggende idé

Hvis det går en vei fra node i til node k med lengde ik, og en vei fra node k til node j med lengde kj, så går det en vei fra node i til node j med lengde $ik + kj$

- Floyds algoritme:
  Denne betraktningen gjentas systematisk for alle tripler i, k og j:
  - Initielt: Avstanden fra node i til node j settes lik vekten på kanten fra i til j, uendelig hvis det ikke går noen kant fra i til j
  - Trinn 0: Se etter forbedringer ved å velge node 0 som mellomnode
  - Etter trinn k: Avstanden mellom to noder er den korteste veien som bare bruker nodene 0, 1, . . . , k som mellomnoder

Optimal substruktur

$V = \{1, 2, ..., n\}$
$V^{(k)} = \{1, 2, ..., k\}$ representerer de første k nodene

Fastsette en startnode (source) $i \in V$, en destinasjonsnode $j \in V$ og en $k \in \{1, 2, ..., n\}$.

P = korteste $i - j$ vei der alle indre nodene er i $V^{(k)}$

k begrenser hvilke noder vi kan velge for en gitt sub-problem.

Optimal substruktur

$V = \{1, 2, ..., n\}$
$V^{(k)} = \{1, 2, ..., k\}$ representerer de første k nodene

Fastsette en startnode (source) $i \in V$, en destinasjonsnode $j \in V$ og en $k \in \{1, 2, ..., n\}$.

P = korteste $i - j$ vei der alle indre nodene er i $V^{(k)}$

k begrenser hvilke noder vi kan velge for en gitt sub-problem.

Optimal substruktur

$V = \{1, 2, ..., n\}$
$V^{(k)} = \{1, 2, ..., k\}$ representerer de første k nodene

Fastsette en startnode (source) $i \in V$, en destinasjonsnode $j \in V$ og en $k \in \{1, 2, ..., n\}$.

P = korteste $i - j$ vei der alle indre nodene er i $V^{(k)}$

k begrenser hvilke noder vi kan velge for en gitt sub-problem.

Optimal substruktur

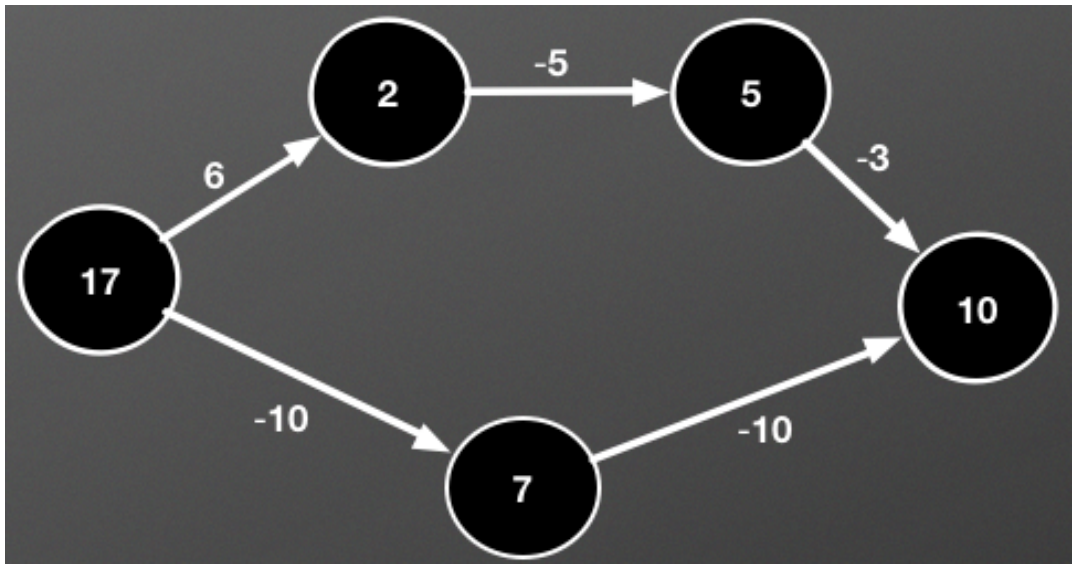
$V = \{1, 2, ..., n\}$
$V^{(k)} = \{1, 2, ..., k\}$ representerer de første k nodene

Fastsette en startnode (source) $i \in V$, en destinasjonsnode $j \in V$ og en $k \in \{1, 2, ..., n\}$.

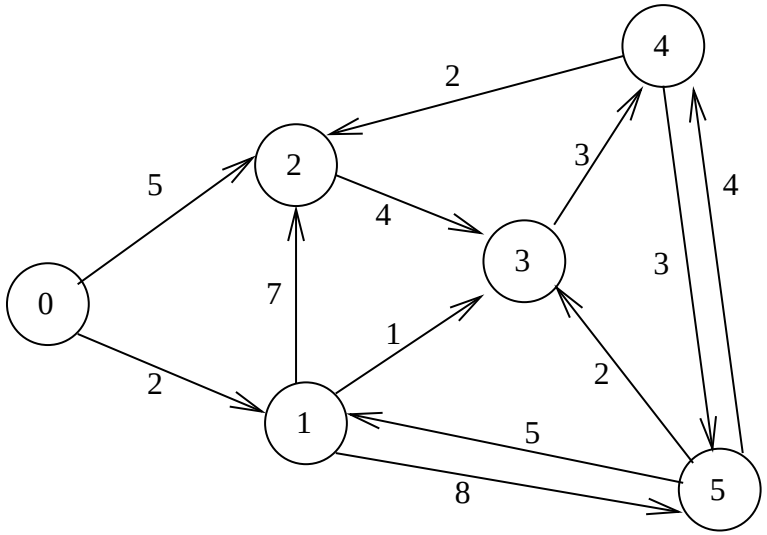
P = korteste $i - j$ vei der alle indre nodene er i $V^{(k)}$

k begrenser hvilke noder vi kan velge for en gitt sub-problem.

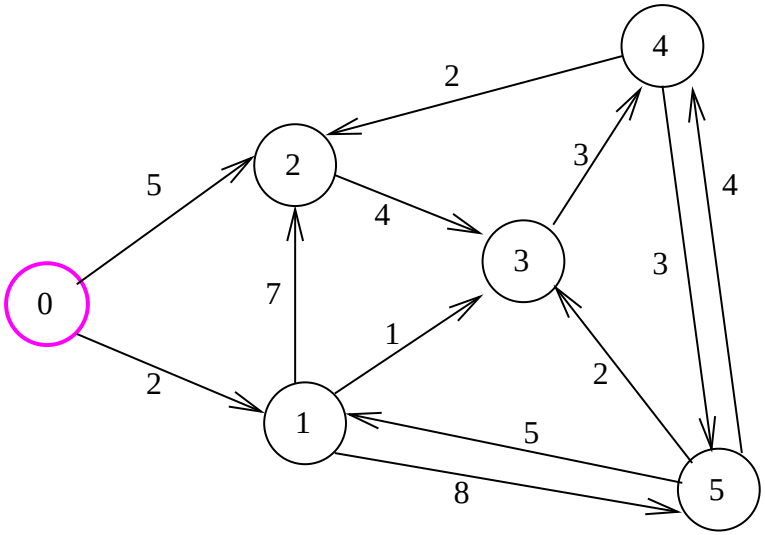
[i = 17, j=10, k=5]



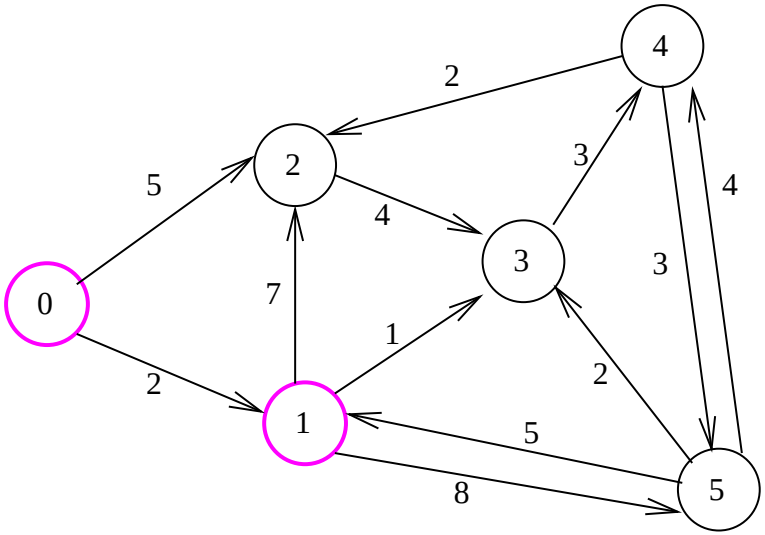
Hva er P?



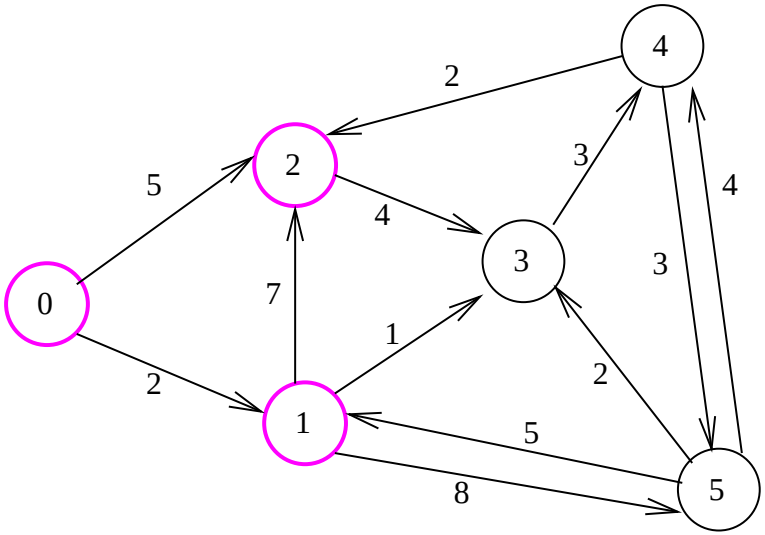
	0	1	2	3	4	5
0	0	2	5	∞	∞	∞
1	∞	0	7	1	∞	8
2	∞	∞	0	4	∞	∞
3	∞	∞	∞	0	3	∞
4	∞	∞	2	∞	0	3
5	∞	5	∞	2	4	0



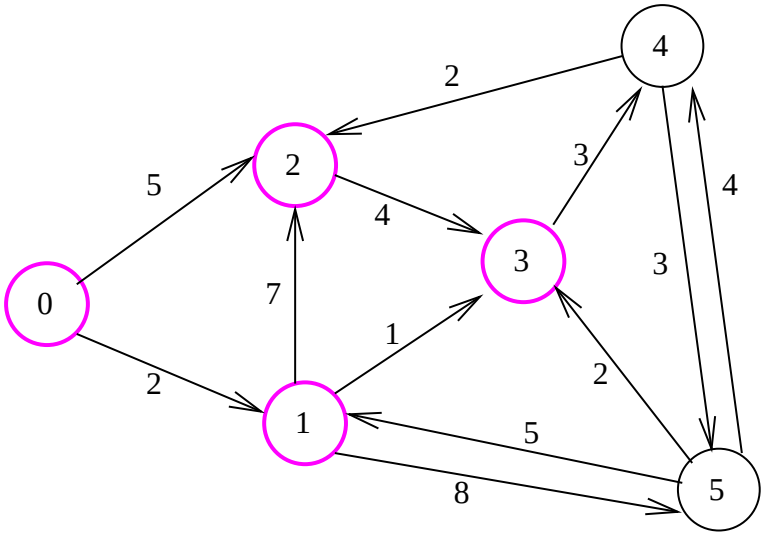
	0	1	2	3	4	5
0	0	2	5	∞	∞	∞
1	∞	0	7	1	∞	8
2	∞	∞	0	4	∞	∞
3	∞	∞	∞	0	3	∞
4	∞	∞	2	∞	0	3
5	∞	5	∞	2	4	0



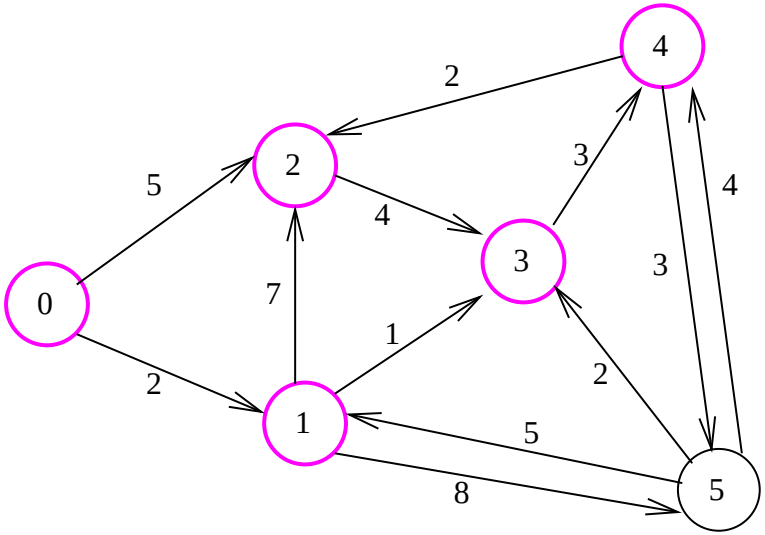
	0	1	2	3	4	5
0	0	2	5	3_1	∞	10_1
1	∞	0	7	1	∞	8
2	∞	∞	0	4	∞	∞
3	∞	∞	∞	0	3	∞
4	∞	∞	2	∞	0	3
5	∞	5	12_1	2	4	0



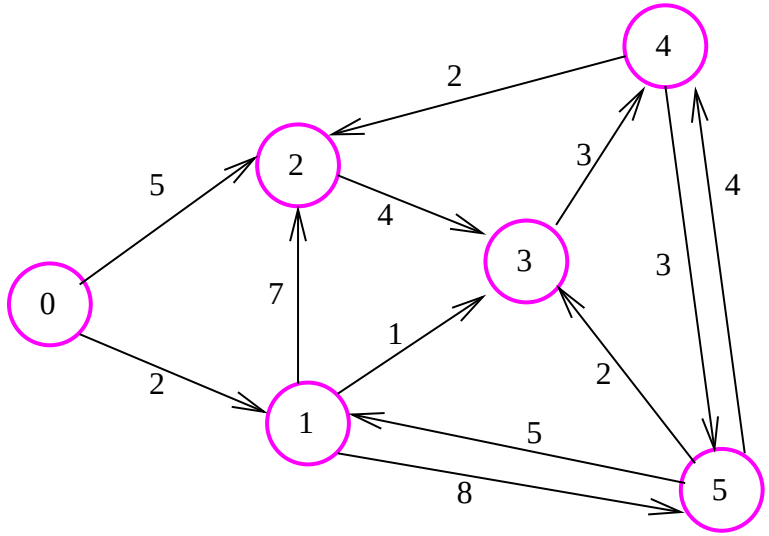
	0	1	2	3	4	5
0	0	2	5	3_1	∞	10_1
1	∞	0	7	1	∞	8
2	∞	∞	0	4	∞	∞
3	∞	∞	∞	0	3	∞
4	∞	∞	2	6_2	0	3
5	∞	5	12_1	2	4	0



	0	1	2	3	4	5
0	0	2	5	3_1	6_3	10_1
1	∞	0	7	1	4_3	8
2	∞	∞	0	4	7_3	∞
3	∞	∞	∞	0	3	∞
4	∞	∞	2	6_2	0	3
5	∞	5	12_1	2	4	0



	0	1	2	3	4	5
0	0	2	5	3_1	6_3	9_4
1	∞	0	6_4	1	4_3	7_4
2	∞	∞	0	4	7_3	10_4
3	∞	∞	5_4	0	3	6_4
4	∞	∞	2	6_2	0	3
5	∞	5	6_4	2	4	0

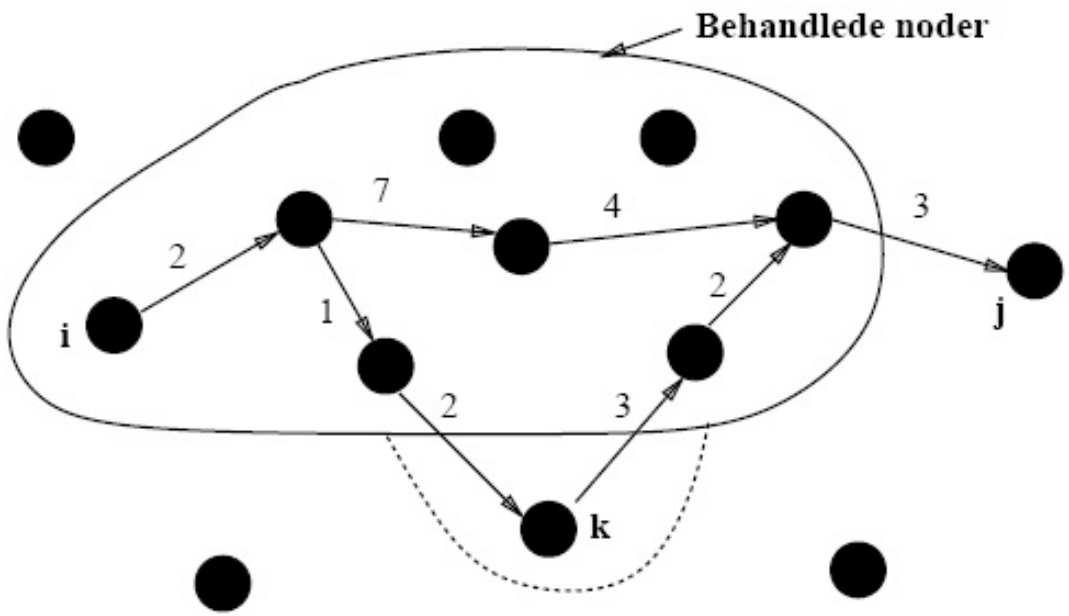


	0	1	2	3	4	5
0	0	2	5	3_1	6_3	9_4
1	∞	0	6_4	1	4_3	7_4
2	∞	15_5	0	4	7_3	10_4
3	∞	11_5	5_4	0	3	6_4
4	∞	8_5	2	5_5	0	3
5	∞	5	6_4	2	4	0

Hvorfor virker Floyd?

Floyd-invarianten:

avstand*[*i*][*j*]** vil være lik lengden av den korteste veien fra node ***i til node ***j*** som har alle sine indre noder behandlet



FØR:	ETTER:
A(i,j)=16	A(i,j)=13
A(i,k)=5	.....
A(k,j)=8	.....

Hva vet vi om P?

1. Hvis node k ikke er en indre node i P:
   P er korteste vei der alle interne noder er i $V^{(k-1)}$.
2. Hvis node k er en indre node i P:
   P_1= korteste $i-k$ vei der alle interne noder er i $V^{(k-1)}$.
   P_2= korteste $k-j$ vei der alle interne noder er i $V^{(k-1)}$.

Når vi gjør beregninger for k, vet vi at alle beregninger for $k-1$ er ferdige.
Trenger array av 3 dimensjoner (startnode, dest. node, prefix):

A[i,j,0] = 0 hvis i=j
A[i,j,0] = C_{ij} hvis (i,j) er en kant i grafen
A[i,j,0] = ∞ ellers

A[i,j,k] = min(A[i,j,k-1], A[i,k,k-1]+ A[k,j,k-1])

Hva vet vi om P?

1. Hvis node k ikke er en indre node i P:
   P er korteste vei der alle interne noder er i $V^{(k-1)}$.
2. Hvis node k er en indre node i P:
   P_1= korteste $i-k$ vei der alle interne noder er i $V^{(k-1)}$.
   P_2= korteste $k-j$ vei der alle interne noder er i $V^{(k-1)}$.

Når vi gjør beregninger for k, vet vi at alle beregninger for $k-1$ er ferdige.
Trenger array av 3 dimensjoner (startnode, dest. node, prefix):

A[i,j,0] = 0 hvis i=j
A[i,j,0] = C_{ij} hvis (i,j) er en kant i grafen
A[i,j,0] = ∞ ellers

A[i,j,k] = min(A[i,j,k-1], A[i,k,k-1]+ A[k,j,k-1])

Hva vet vi om P?

1. Hvis node k ikke er en indre node i P:
   P er korteste vei der alle interne noder er i $V^{(k-1)}$.
2. Hvis node k er en indre node i P:
   P_1= korteste $i-k$ vei der alle interne noder er i $V^{(k-1)}$.
   P_2= korteste $k-j$ vei der alle interne noder er i $V^{(k-1)}$.

Når vi gjør beregninger for k, vet vi at alle beregninger for $k-1$ er ferdige.
Trenger array av 3 dimensjoner (startnode, dest. node, prefix):

A[i,j,0] = 0 hvis i=j
A[i,j,0] = C_{ij} hvis (i,j) er en kant i grafen
A[i,j,0] = ∞ ellers

A[i,j,k] = min(A[i,j,k-1], A[i,k,k-1]+ A[k,j,k-1])

Hva vet vi om P?

1. Hvis node k ikke er en indre node i P:
   P er korteste vei der alle interne noder er i $V^{(k-1)}$.
2. Hvis node k er en indre node i P:
   P_1= korteste $i-k$ vei der alle interne noder er i $V^{(k-1)}$.
   P_2= korteste $k-j$ vei der alle interne noder er i $V^{(k-1)}$.

Når vi gjør beregninger for k, vet vi at alle beregninger for $k-1$ er ferdige.

Trenger array av 3 dimensjoner (startnode, dest. node, prefix):

A[i,j,0] = 0 hvis i=j

A[i,j,0] = C_{ij} hvis (i,j) er en kant i grafen

A[i,j,0] = ∞ ellers

A[i,j,k] = min(A[i,j,k-1], A[i,k,k-1]+ A[k,j,k-1])

Hva vet vi om P?

1. Hvis node k ikke er en indre node i P:
   P er korteste vei der alle interne noder er i $V^{(k-1)}$.
2. Hvis node k er en indre node i P:
   P_1= korteste $i-k$ vei der alle interne noder er i $V^{(k-1)}$.
   P_2= korteste $k-j$ vei der alle interne noder er i $V^{(k-1)}$.

Når vi gjør beregninger for k, vet vi at alle beregninger for $k-1$ er ferdige.
Trenger array av 3 dimensjoner (startnode, dest. node, prefix):

A[i,j,0] = 0 hvis i=j
A[i,j,0] = C_{ij} hvis (i,j) er en kant i grafen
A[i,j,0] = ∞ ellers

A[i,j,k] = min(A[i,j,k-1], A[i,k,k-1]+ A[k,j,k-1])

Hva vet vi om P?

1. Hvis node k ikke er en indre node i P:
   P er korteste vei der alle interne noder er i $V^{(k-1)}$.
2. Hvis node k er en indre node i P:
   P_1= korteste $i-k$ vei der alle interne noder er i $V^{(k-1)}$.
   P_2= korteste $k-j$ vei der alle interne noder er i $V^{(k-1)}$.

Når vi gjør beregninger for k, vet vi at alle beregninger for $k-1$ er ferdige.
Trenger array av 3 dimensjoner (startnode, dest. node, prefix):

A[i,j,0] = 0 hvis i=j
A[i,j,0] = C_{ij} hvis (i,j) er en kant i grafen
A[i,j,0] = ∞ ellers

A[i,j,k] = min(A[i,j,k-1], A[i,k,k-1]+ A[k,j,k-1])

```
public static void kortesteVeiAlleTilAlle(
  int[ ][ ] nabo, int[ ][ ] avstand, int[ ][ ] vei) {

  int n = avstand.length; // Forutsetning: arrayene er
                          // kvadratiske med samme dimensjon
  // Initialisering:
  for (int i = 0; i < n; i++) {
    for (int j = 0; j < n; j++) {
      avstand[i][j] = nabo[i][j];
      vei[i][j] = -1;                  // Ingen vei foreløpig
    }
  }

  for (int k = 0; k < n; k++) {
    for (int i = 0; i < n; i++) {
      for (int j = 0; j < n; j++) {
        if (avstand[i][k] + avstand[k][j] < avstand[i][j]) {
          // Kortere vei fra i til j funnet via k
          avstand[i][j] = avstand[i][k] + avstand[k][j];
          vei[i][j] = k;
}}}}}
```

Tidsforbruket er åpenbart $\mathcal{O}(n^3)$

```
public static void kortesteVeiAlleTilAlle(
  int[ ][ ] nabo, int[ ][ ] avstand, int[ ][ ] vei) {

  int n = avstand.length; // Forutsetning: arrayene er
                          // kvadratiske med samme dimensjon
  // Initialisering:
  for (int i = 0; i < n; i++) {
    for (int j = 0; j < n; j++) {
      avstand[i][j] = nabo[i][j];
      vei[i][j] = -1;                  // Ingen vei foreløpig
    }
  }

  for (int k = 0; k < n; k++) {
    for (int i = 0; i < n; i++) {
      for (int j = 0; j < n; j++) {
        if (avstand[i][k] + avstand[k][j] < avstand[i][j]) {
          // Kortere vei fra i til j funnet via k
          avstand[i][j] = avstand[i][k] + avstand[k][j];
          vei[i][j] = k;
}}}}}
```

Tidsforbruket er åpenbart $\mathcal{O}(n^3)$

Hvordan tolke resultatet av Floyd

- Ved start inneholder ***nabo*[*i*][*j*]** lengden av kanten fra ***i*** til ***j***, ∞ hvis det ikke er noen kant
- Floyd lar *nabo* være uendret og legger resultatet i *avstand* og *vei*
- $avstand[i][j]$ er lengden av korteste vei fra i til j, ∞ hvis det ikke er noen vei
- Når vi oppdager at den korteste veien fra i til j passerer gjennom en mellomnode k, setter vi $vei[i][j] = k$
- *vei* sier hva som er den korteste veien
  - $k_1 = vei[i][j]$ er den største verdi av k slik at k ligger på den korteste veien fra i til j
  - $k_2 = vei[i][k_1]$ er den største verdi av k slik at k ligger på den korteste veien fra i til k_1 osv.
  - Når $vei[i][k_m] = -1$, er (i, k_m) den første kanten i korteste vei fra i til j

Hvordan tolke resultatet av Floyd

- Ved start inneholder ***nabo*[*i*][*j*]** lengden av kanten fra ***i*** til ***j***, ∞ hvis det ikke er noen kant
- Floyd lar ***nabo*** være uendret og legger resultatet i ***avstand*** og ***vei***
- ***avstand*[*i*][*j*]** er lengden av korteste vei fra ***i*** til ***j***, ∞ hvis det ikke er noen vei
- Når vi oppdager at den korteste veien fra ***i*** til ***j*** passerer gjennom en mellomnode ***k***, setter vi ***vei*[*i*][*j*] = *k***
- *vei* sier hva som er den korteste veien
  - $k_1 =$ ***vei*[*i*][*j*]** er den største verdi av ***k*** slik at ***k*** ligger på den korteste veien fra ***i*** til ***j***
  - $k_2 =$ ***vei*[*i*][k_1]** er den største verdi av ***k*** slik at ***k*** ligger på den korteste veien fra ***i*** til k_1 osv.
  - Når ***vei*[*i*][k_m] $= -1$**, er (i, k_m) den første kanten i korteste vei fra ***i*** til ***j***

Hvordan tolke resultatet av Floyd

- Ved start inneholder ***nabo*[*i*][*j*]** lengden av kanten fra ***i*** til ***j***, ∞ hvis det ikke er noen kant
- Floyd lar ***nabo*** være uendret og legger resultatet i ***avstand*** og ***vei***
- ***avstand*[*i*][*j*]** er lengden av korteste vei fra ***i*** til ***j***, ∞ hvis det ikke er noen vei
- Når vi oppdager at den korteste veien fra ***i*** til ***j*** passerer gjennom en mellomnode ***k***, setter vi ***vei*[*i*][*j*] = *k***
- ***vei*** sier hva som er den korteste veien
  - $k_1 =$ ***vei*[*i*][*j*]** er den største verdi av ***k*** slik at ***k*** ligger på den korteste veien fra ***i*** til ***j***
  - $k_2 =$ ***vei*[*i*][k_1]** er den største verdi av ***k*** slik at ***k*** ligger på den korteste veien fra ***i*** til k_1 osv.
  - Når ***vei*[*i*][k_m] = -1**, er (i, k_m) den første kanten i korteste vei fra ***i*** til ***j***

Den korteste veien fra i til j

- Hvis $\boldsymbol{vei[i][j] = -1}$, passerer ikke den korteste veien gjennom noen mellomnoder og den korteste veien er $\boldsymbol{(i, j)}$
- Ellers, vi rekursivt beregne den korteste veien fra $\boldsymbol{i}$ til $\boldsymbol{vei[i][j]}$ og den korteste veien fra $\boldsymbol{vei[i][j]}$ til $\boldsymbol{j}$

```
Kortestevei(i,j){
  if (vei[i][j] = -1) //en kant
      output (i,j);
  else {
      kortestevei(i, vei[i][j]);
      kortestevei(vei[i][j], j);
  }
}
```

Oppsummering

Grafer: Oppsummering

- Implementasjon av grafer
- Graf traverseringer
- Topologisk sortering
- Korteste vei
- Minimale spenntrær
- biconnectivity, SCC