INF2220 - Algoritmer og datastrukturer

HØSTEN 2015

Institutt for informatikk, Universitetet i Oslo

Forelesning 6:

Grafer II

Dagens plan:

Minimale spenntrær

- Prim
- Kruskal

Dybde-først søk

- Løkkeleting
- DFS Spenntre
- Biconnectivity

Minimale spenntrær

Minimale spenntrær

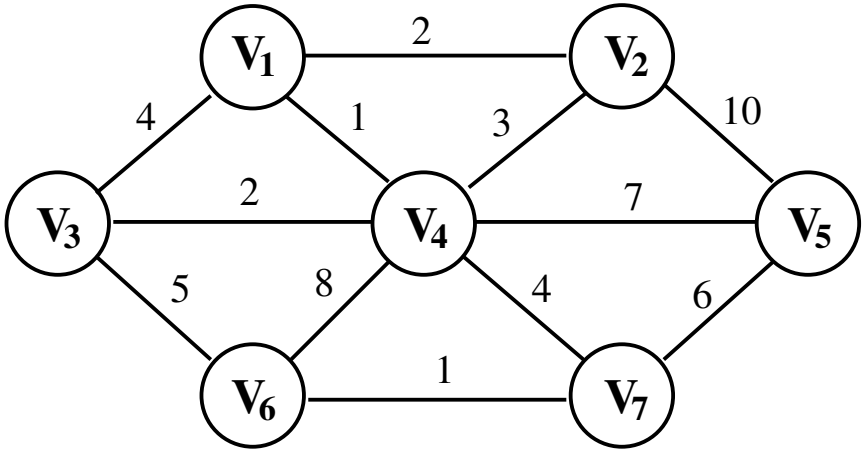
Definition

Et minimalt spenntre for en **urettet** graf **G** er et tre med kanter fra grafen, slik at alle nodene i **G** er forbundet til lavest mulig kostnad

- eksisterer bare for sammenhengende grafer
- Generelt: flere minimale spenntrær for samme graf

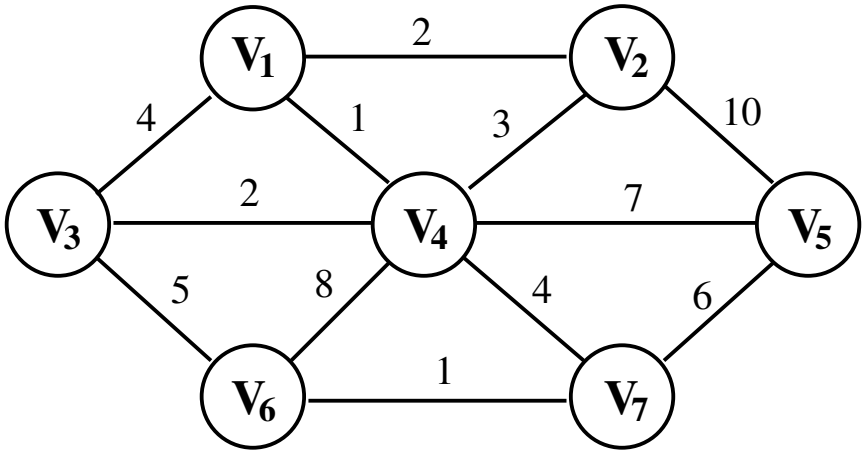
Hvor mange kanter får spenntreet?

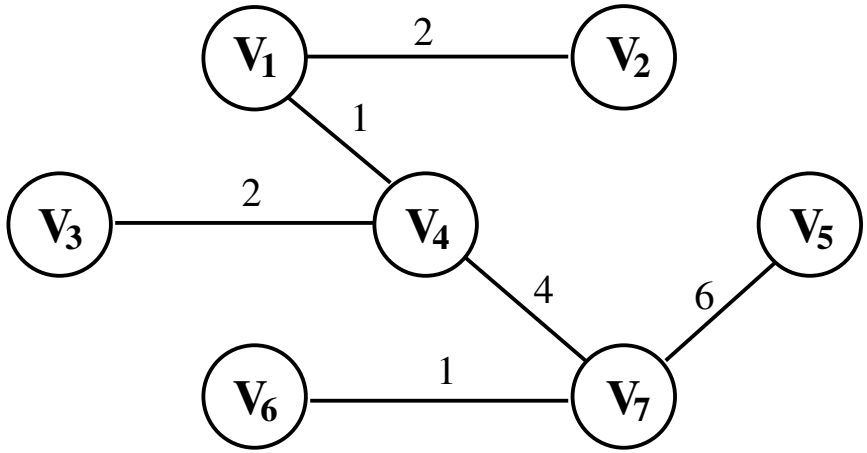
Minimale spenntrær



Hvor mange kanter får spenntreet?

Minimale spenntrær

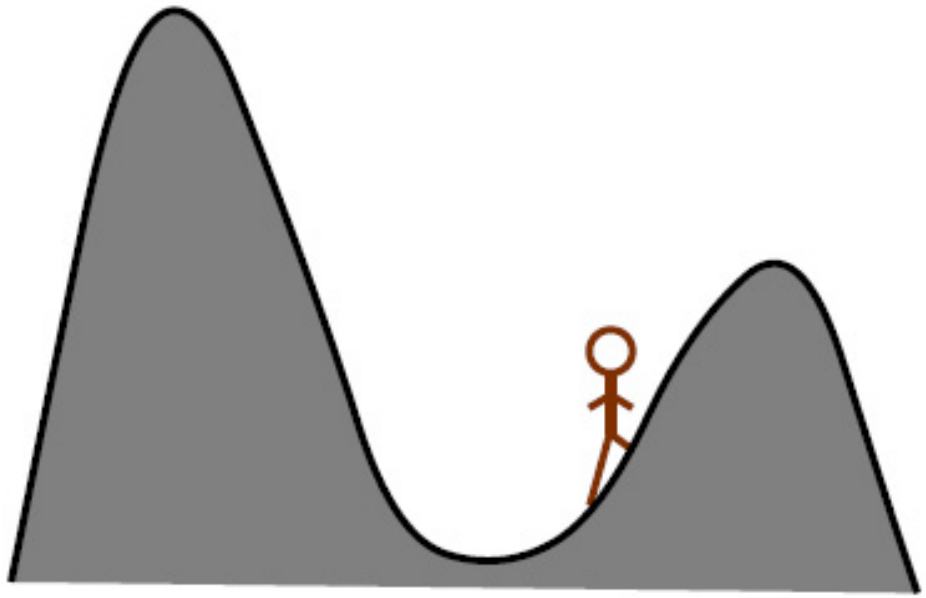




Hvor mange kanter får spenntreet?

Grådige algoritmer

- Prøver i hvert trinn å gjøre det som ser best ut der og da
- Typisk eksempel: Gi vekslepenger
- Raske algoritmer, men kan ikke løse alle problemer:



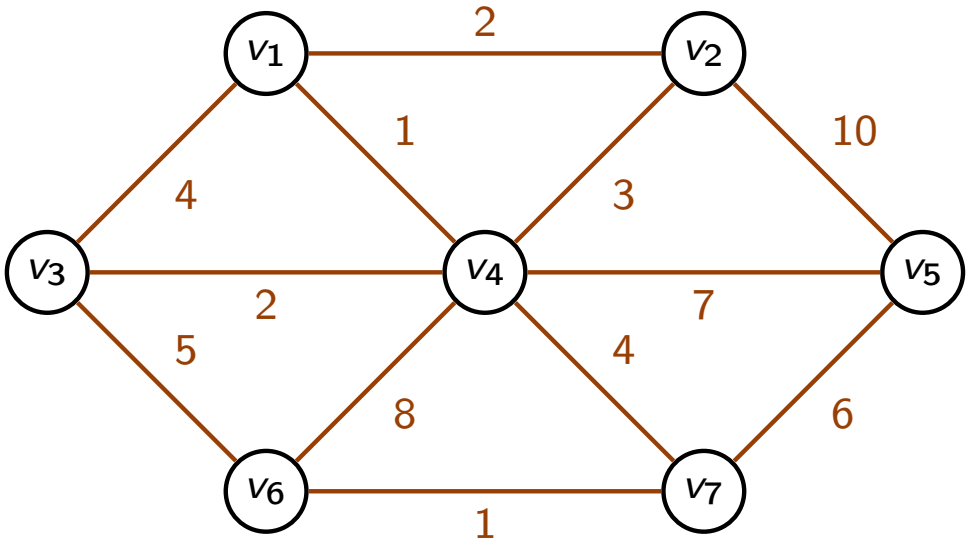
Finn det høyeste punktet!

Vi skal se på to ulike grådige algoritmer for å finne minimale spenntrær

Prims algoritme

- Treet bygges opp trinnvis

  I hvert trinn: pluss en kant (og dermed en tilhørende node) til treet
- 2 typer noder:
  - Noder som er med i treet
  - Noder som ikke er med i treet
- Nye noder: velge en kant $(\mathbf{u}, \mathbf{v})$ med minst vekt slik at $\mathbf{u}$ er med i treet, og $\mathbf{v}$ ikke er det.



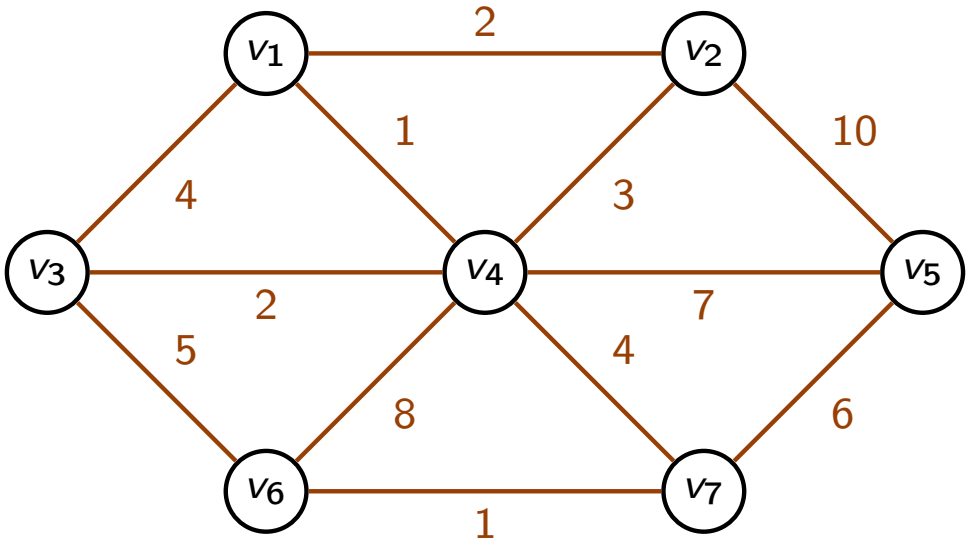
- Algoritmen begynner med å velge en vilkårlig node

  Eksempel: La oss velge $\mathbf{v_1}$

Prims algoritme

- Treet bygges opp trinnvis

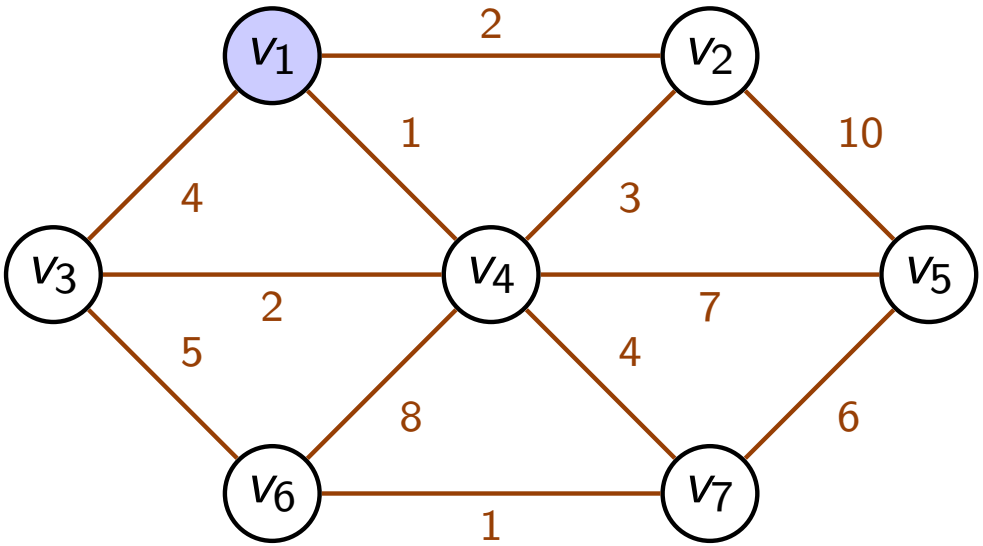
  I hvert trinn: pluss en kant (og dermed en tilhørende node) til treet
- 2 typer noder:
  - Noder som er med i treet
  - Noder som ikke er med i treet
- Nye noder: velge en kant $(\mathbf{u}, \mathbf{v})$ med minst vekt slik at $\mathbf{u}$ er med i treet, og $\mathbf{v}$ ikke er det.



- Algoritmen begynner med å velge en vilkårlig node

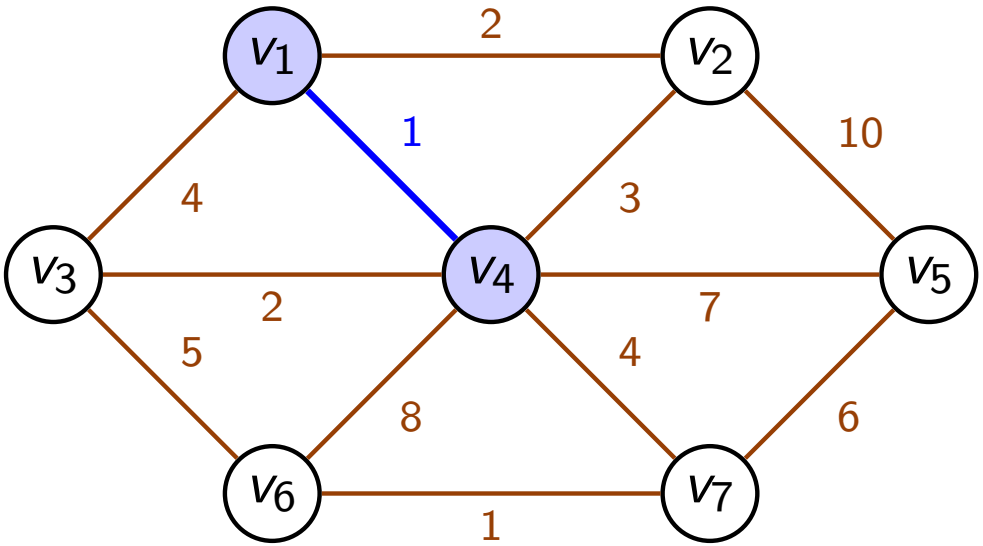
  Eksempel: La oss velge $\mathbf{v_1}$

Prim: growing a tree (1)



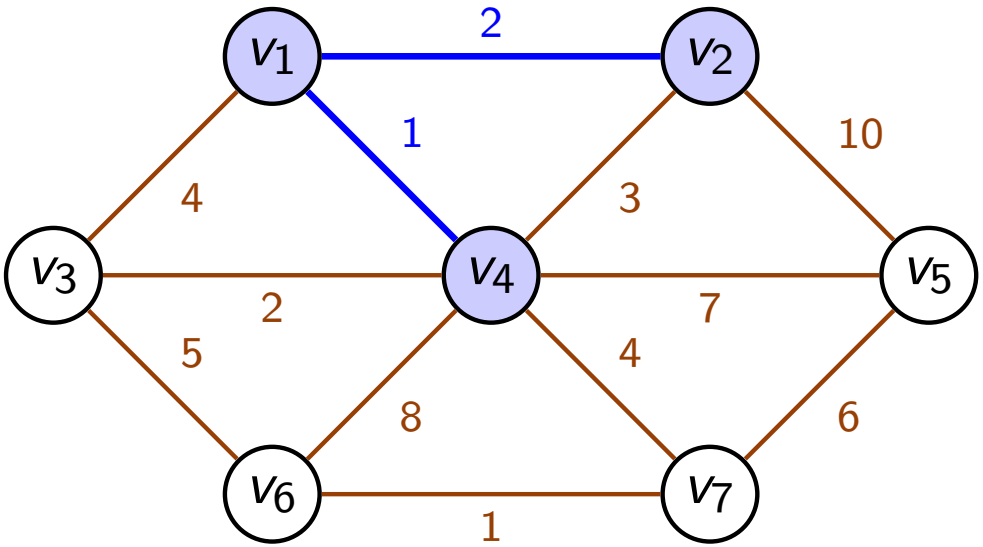
minste kant ut fra $\mathbf{v_1}$ går til $\mathbf{v_4}$, så vi legger den inn i spenntreet

Prim: growing a tree (2)

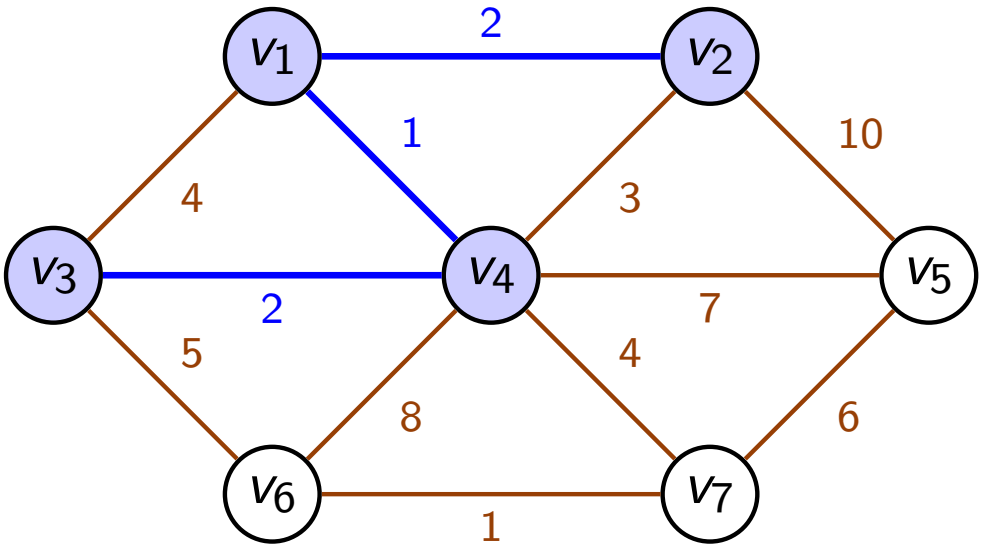


- Vi har nå to mulige fortsettelser:
  - Kanten fra $\mathbf{v_1}$ til $\mathbf{v_2}$
  - Kanten fra $\mathbf{v_4}$ til $\mathbf{v_3}$
- Samme hvilken vi velger, vil den andre av dem bli neste kant, så vi legger dem begge inn i spenntreet

Prim: growing a tree (3)

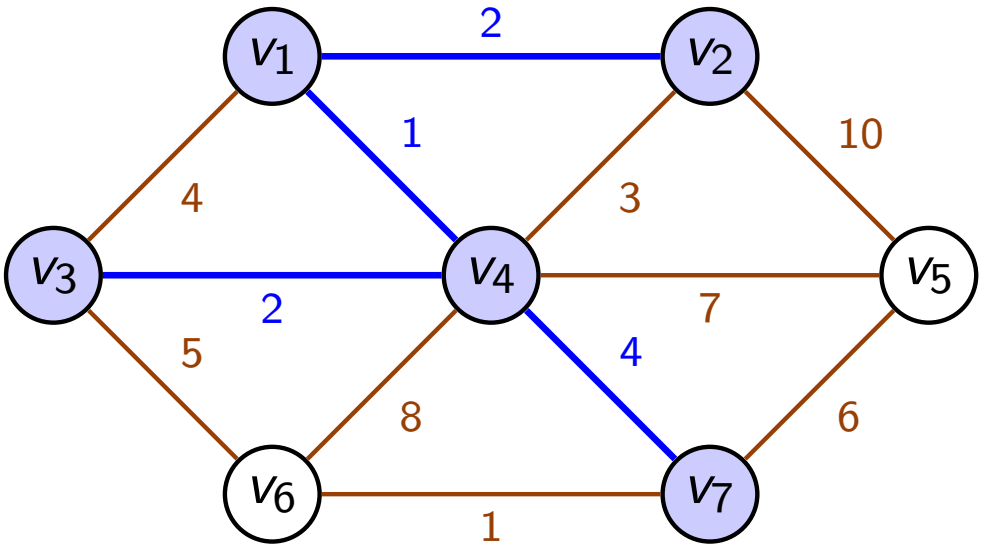


Prim: growing a tree (4)

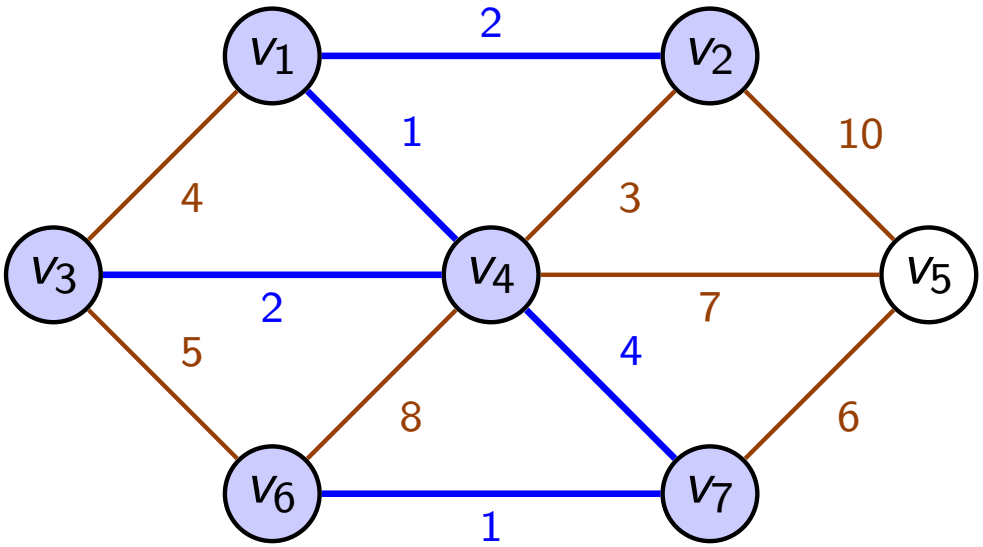


- Minste kant ut fra spenntreet går nå fra $\mathbf{v_4}$ til $\mathbf{v_7}$
- Så får vi kanten fra $\mathbf{v_7}$ til $\mathbf{v_6}$
- Til slutt får vi kanten fra $\mathbf{v_7}$ til $\mathbf{v_5}$

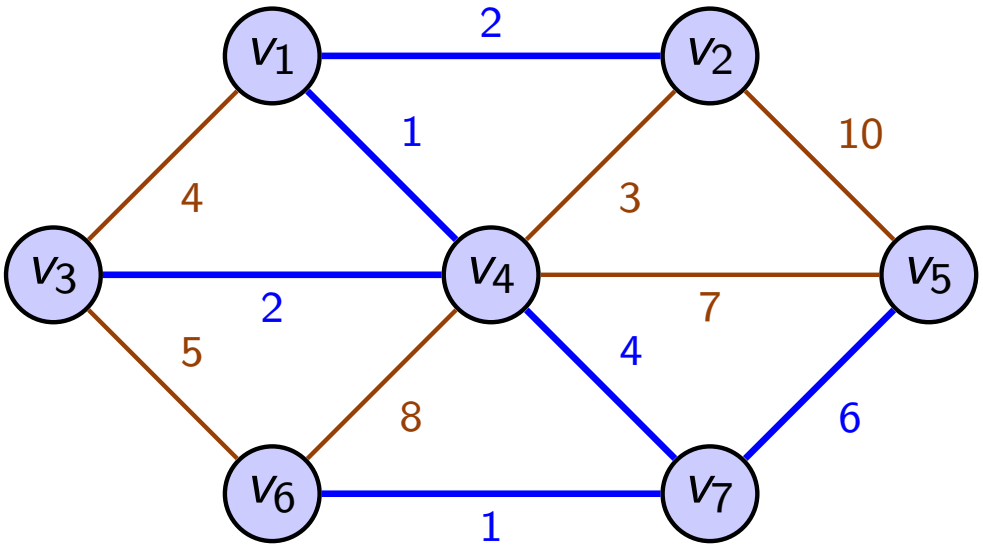
Prim: growing a tree (5)



Prim: growing a tree (6)



Prim: growing a tree (7)



- Prims algoritme: essensielt lik Dijkstras algo for korteste vei!

  I Prims algoritme er avstanden til en ukjent node **v** den minste vekten til en kant som forbinder **v** med en **kjent** node
- Husk: vi har urettede grafer, så hver kant befinner seg i **2** nabolister
- Kjøretidsanalysen er den samme som for Dijkstras algoritme

Hvorfor virker Prim?

Lemma (Løkke-lemma for spenntrær)

Anta at **U** *er et spenntre for en graf, og at kanten* **e** *ikke er med i treet* **U**
Hvis vi legger kanten **e** *til treet* **U**, *dannes en entydig bestemt enkel løkke.*
Hvis vi fjerner en vilkårlig kant i denne løkken, vil vi igjen ha et spenntre for grafen

Invariant

Det treet **T** som dannes av de kantene (og deres endenoder) vi til nå har plukket ut, er slik at det finnes et minimalt spenntre **U** for grafen som inneholder (alle kantene i) **T**

Kruskals algoritme:

Se på kantene en etter en, sortert etter minst vekt

Kanten aksepteres hvis, og bare hvis, den ikke fører til noen løkke

Algoritmen implementeres vha. en prioritetskø og disjunkte mengder:

- Initielt plasseres kantene i en prioritetskø og nodene i hver sin disjunkte mengde (slik at **find**(**v**) gir mengden til (**v**)).

  Invariant:

  De disjunkte mengdene er subtrær av det endelige spenntreet

- **deleteMin** gir neste kant $(\mathbf{u}, \mathbf{v})$ som skal testes
  - Hvis **find**(**u**) ! = **find**(**v**), har vi en ny kant i treet og gjør **union**(**u**, **v**)
  - Hvis ikke, ville $(\mathbf{u}, \mathbf{v})$ ha dannet en løkke, så kanten forkastes
- Algoritmen terminerer når prioritetskøen er tom, eventuelt når vi har lagt inn $|\mathbf{V}| - 1$ kanter

Kruskals algoritme:

Se på kantene en etter en, sortert etter minst vekt

Kanten aksepteres hvis, og bare hvis, den ikke fører til noen løkke

Algoritmen implementeres vha. en prioritetskø og disjunkte mengder:

- Initielt plasseres kantene i en prioritetskø og nodene i hver sin disjunkte mengde (slik at **find**(**v**) gir mengden til (**v**)).

  Invariant:

  De disjunkte mengdene er subtrær av det endelige spenntreet

- **deleteMin** gir neste kant $(\mathbf{u}, \mathbf{v})$ som skal testes
  - Hvis **find**(**u**) ! = **find**(**v**), har vi en ny kant i treet og gjør **union**$(\mathbf{u}, \mathbf{v})$
  - Hvis ikke, ville $(\mathbf{u}, \mathbf{v})$ ha dannet en løkke, så kanten forkastes
- Algoritmen terminerer når prioritetskøen er tom, eventuelt når vi har lagt inn $|\mathbf{V}| - 1$ kanter

Kruskals algoritme:

Se på kantene en etter en, sortert etter minst vekt

Kanten aksepteres hvis, og bare hvis, den ikke fører til noen løkke

Algoritmen implementeres vha. en prioritetskø og disjunkte mengder:

- Initielt plasseres kantene i en prioritetskø og nodene i hver sin disjunkte mengde (slik at **find**(**v**) gir mengden til (**v**)).

  Invariant:

  De disjunkte mengdene er subtrær av det endelige spenntreet

- **deleteMin** gir neste kant $(\mathbf{u}, \mathbf{v})$ som skal testes
  - Hvis **find**(**u**) ! = **find**(**v**), har vi en ny kant i treet og gjør **union**$(\mathbf{u}, \mathbf{v})$
  - Hvis ikke, ville $(\mathbf{u}, \mathbf{v})$ ha dannet en løkke, så kanten forkastes
- Algoritmen terminerer når prioritetskøen er tom, eventuelt når vi har lagt inn $|\mathbf{V}| - 1$ kanter

Kruskals algoritme:

Se på kantene en etter en, sortert etter minst vekt

Kanten aksepteres hvis, og bare hvis, den ikke fører til noen løkke

Algoritmen implementeres vha. en prioritetskø og disjunkte mengder:

- Initielt plasseres kantene i en prioritetskø og nodene i hver sin disjunkte mengde (slik at **find**(**v**) gir mengden til (**v**)).

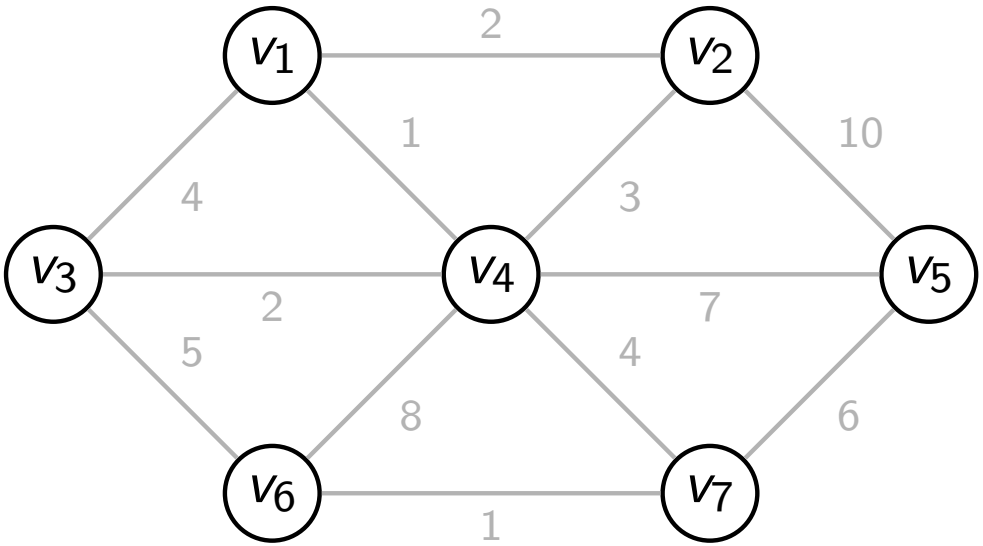
  Invariant:

  De disjunkte mengdene er subtrær av det endelige spenntreet

- **deleteMin** gir neste kant (**u**, **v**) som skal testes
  - Hvis **find**(**u**) ! = **find**(**v**), har vi en ny kant i treet og gjør **union**(**u**, **v**)
  - Hvis ikke, ville (**u**, **v**) ha dannet en løkke, så kanten forkastes
- Algoritmen terminerer når prioritetskøen er tom, eventuelt når vi har lagt inn $|\mathbf{V}| - \mathbf{1}$ kanter

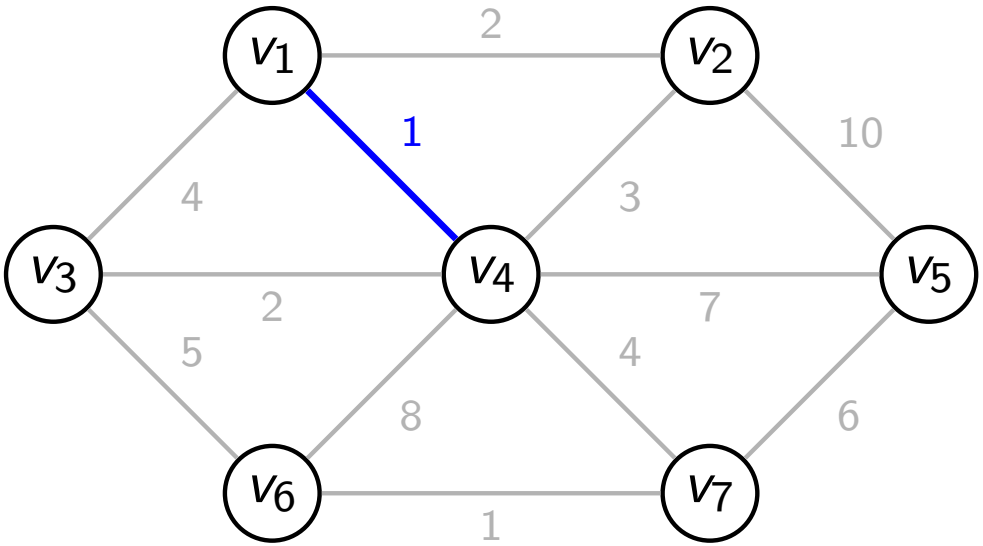
Growing a forest (1)

Utgangspunkt for Kruskals algoritme:



Growing a forest (2)

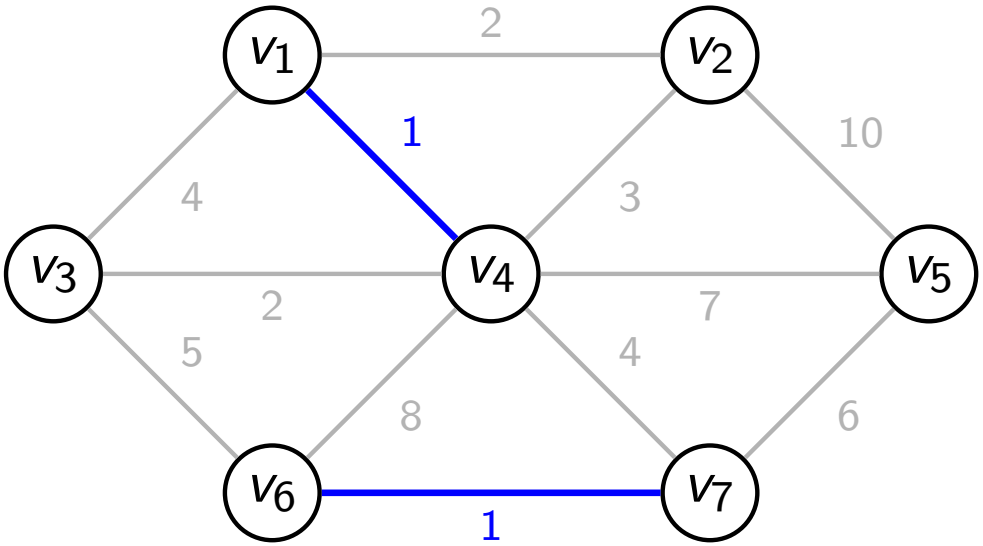
Utgangspunkt for Kruskals algoritme:



Vi har to kanter med vekt 1 — De blir til to subtrær i spenn-treet

Growing a forest (3)

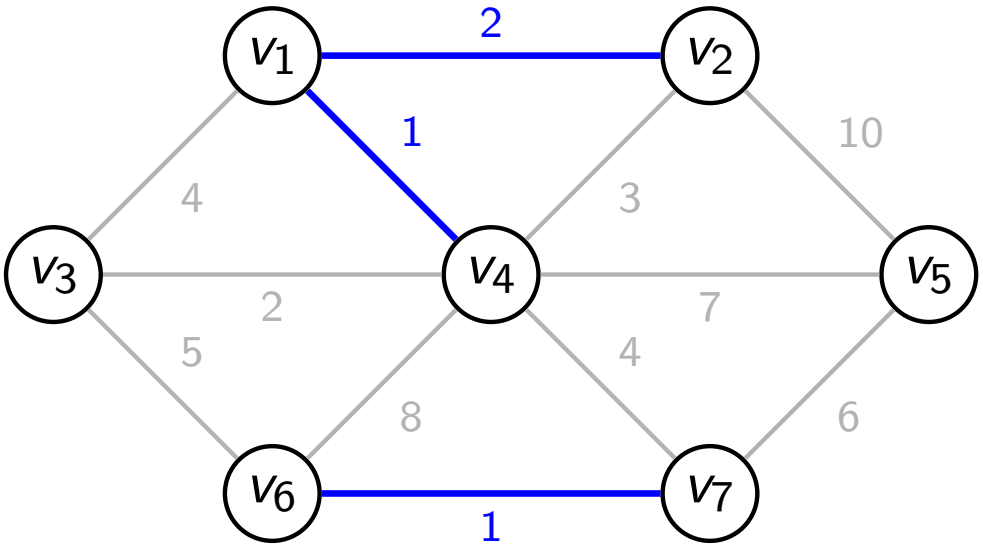
Utgangspunkt for Kruskals algoritme:



Vi har to kanter med vekt 1 — De blir til to subtrær i spenn-treet

Growing a forest (4)

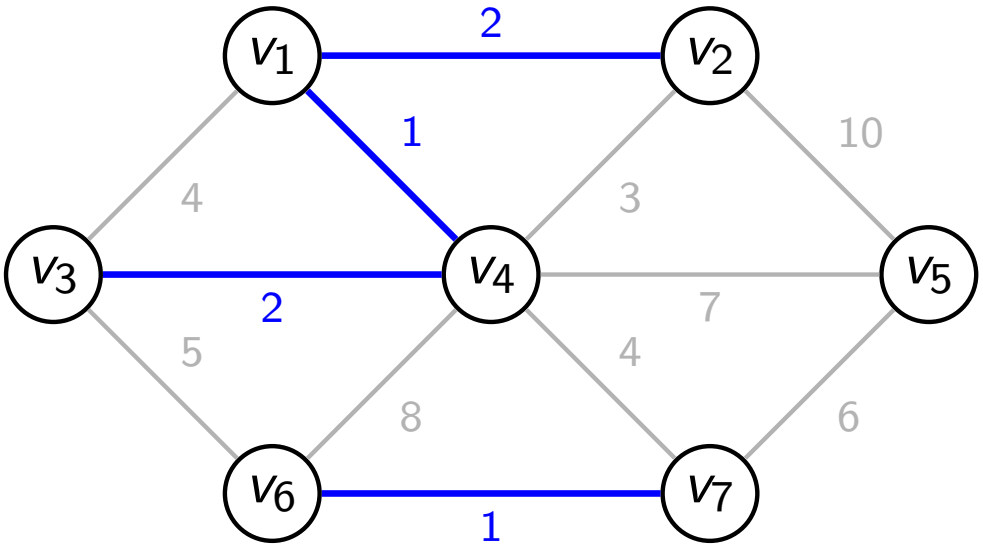
Utgangspunkt for Kruskals algoritme:



Vi har to kanter med vekt 2 — De blir del av det øverste subtreet

Growing a forest (5)

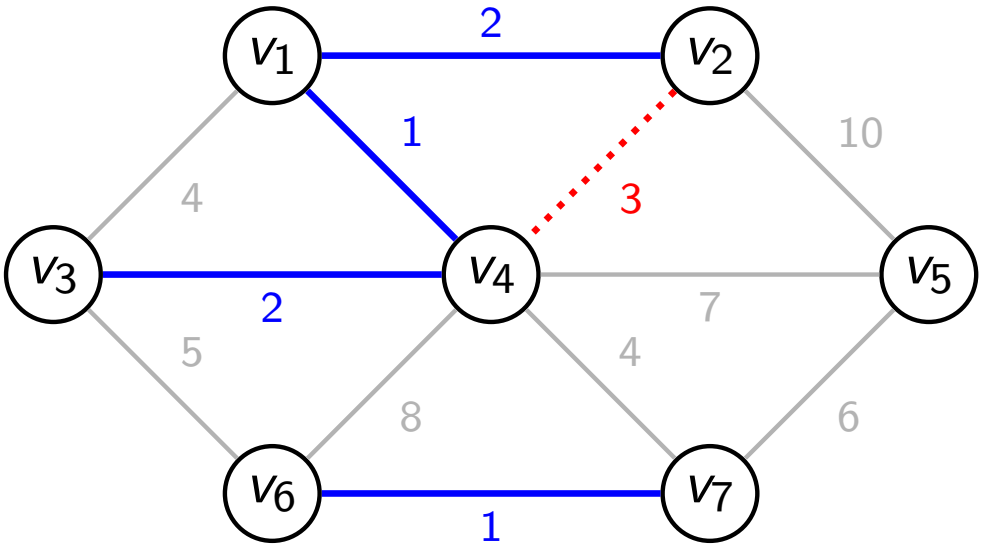
Utgangspunkt for Kruskals algoritme:



Vi har to kanter med vekt 2 — De blir del av det øverste subtreet

Growing a forest (6)

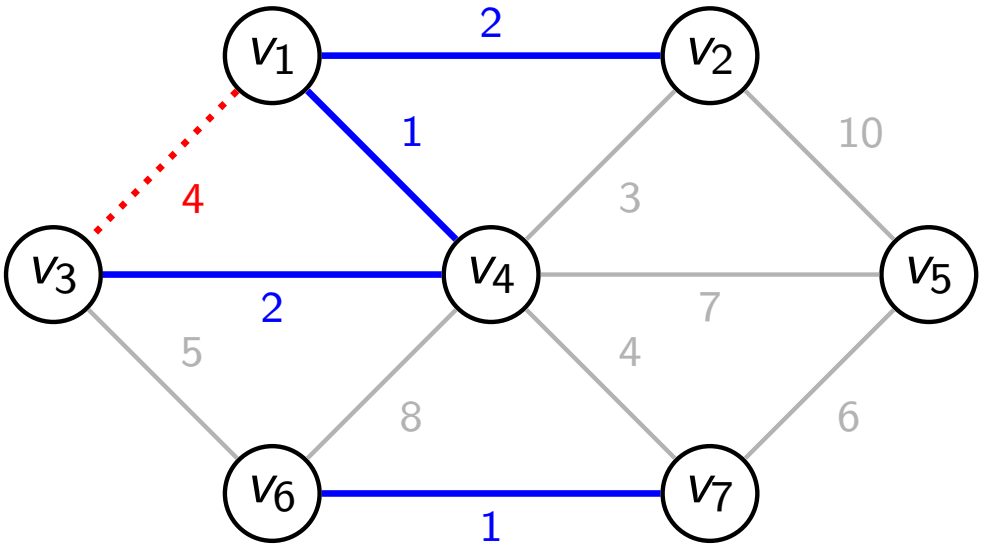
Utgangspunkt for Kruskals algoritme:



kant med vekt 3: — Den vil lage en løkke og må forkastes

Growing a forest (7)

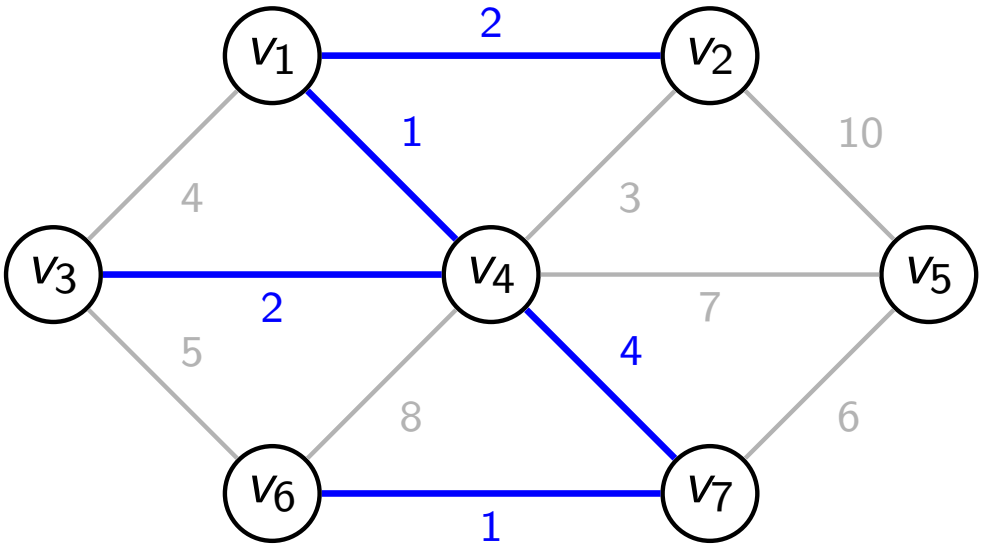
Utgangspunkt for Kruskals algoritme:



2 kanter med vekt 4 hvor den mellom $\mathbf{v_1}$ og $\mathbf{v_3}$ danner en løkke, mens den andre binder de to subtrærne sammen

Growing a forest (8)

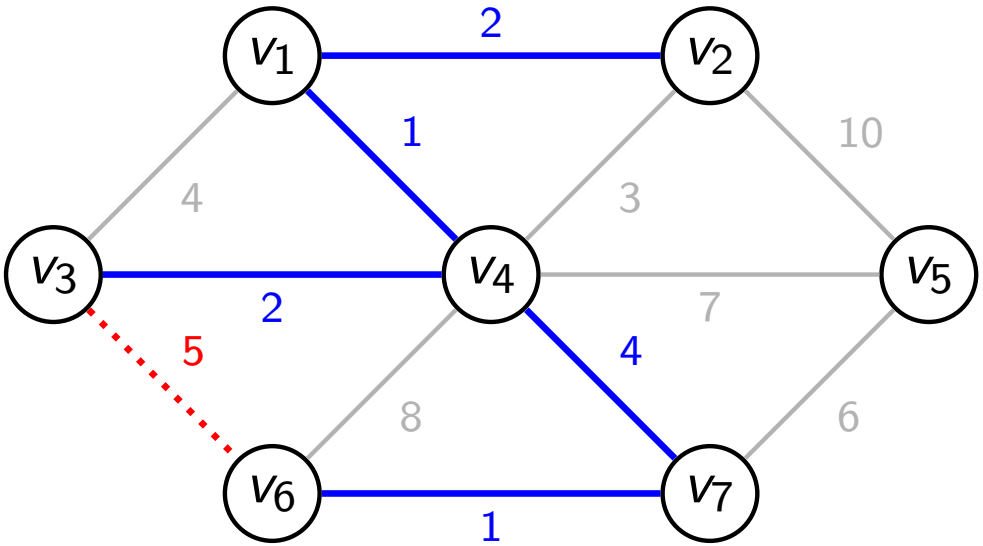
Utgangspunkt for Kruskals algoritme:



2 kanter med vekt 4 hvor den mellom $\mathbf{v_1}$ og $\mathbf{v_3}$ danner en løkke, mens den andre binder de to subtrærne sammen

Growing a forest (9)

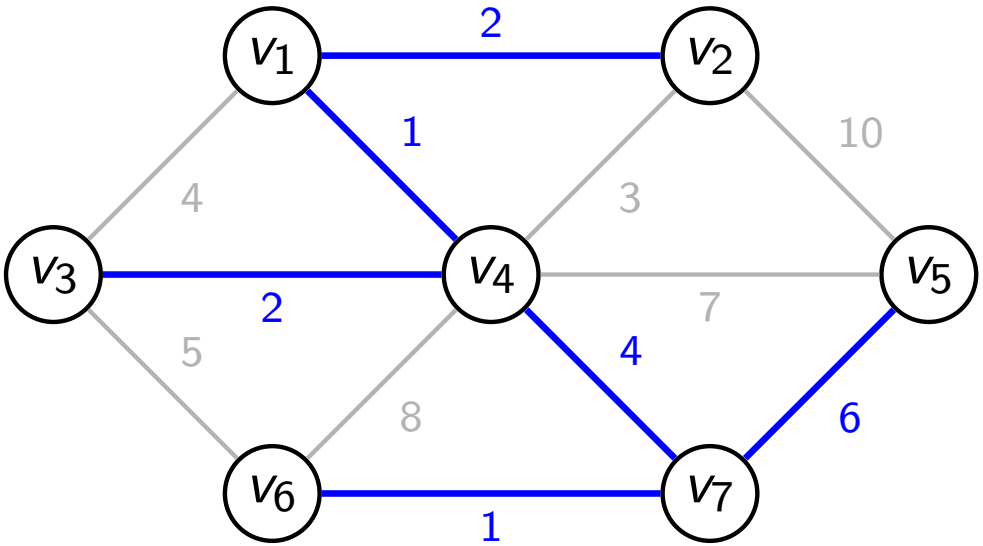
Utgangspunkt for Kruskals algoritme:



kant med vekt 5: løkke

Growing a forest (10)

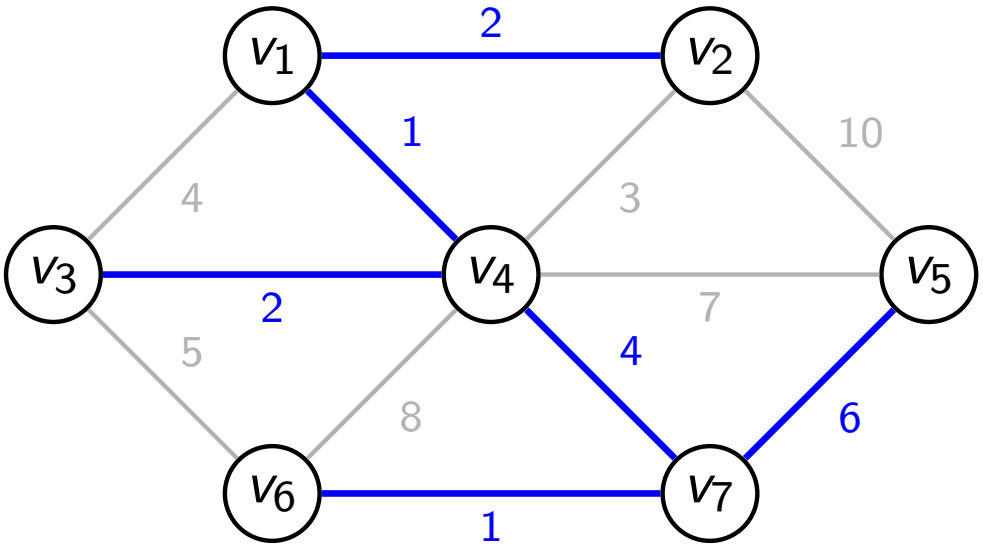
Utgangspunkt for Kruskals algoritme:



kant med vekt 6: knytter den siste noden til treet

Growing a forest (11)

Utgangspunkt for Kruskals algoritme:



- Nå har vi lagt inn $|V| - 1$ kanter i spenn-treet og kan slutte.
- Hvis vi fortsetter med resten av kantene, vil alle danne løkker og bli forkastet

Tidsanalyse:

- Hovedløkken går $|\mathbf{E}|$ ganger
- I hver iterasjon gjøres en **deleteMin**, to **find** og en **union**, med samlet tidsforbruk

$$\mathcal{O}(\log|\mathbf{E}|) + \mathbf{2}\cdot\mathcal{O}(\log|\mathbf{V}|)+\mathcal{O}(\mathbf{1}) = \mathcal{O}(\log|\mathbf{V}|)$$

  (fordi $\log|\mathbf{E}| < \mathbf{2}\cdot\log|\mathbf{V}|$)
- Totalt tidsforbruk er $\mathcal{O}(|\mathbf{E}|\cdot\log|\mathbf{V}|)$

Prim vs. Kruskal

- Prims algoritme er noe mer effektiv enn Kruskals, spesielt for tette grafer
- Prims algoritme virker bare i sammenhengende grafer
- Kruskas algoritme gir et minimalt spenn-tre i hver sammenhengskomponent i grafen

Tidsanalyse:

- Hovedløkken går $|\mathbf{E}|$ ganger
- I hver iterasjon gjøres en **deleteMin**, to **find** og en **union**, med samlet tidsforbruk

$$\mathcal{O}(\log|\mathbf{E}|) + \mathbf{2}\cdot\mathcal{O}(\log|\mathbf{V}|)+\mathcal{O}(\mathbf{1}) = \mathcal{O}(\log|\mathbf{V}|)$$

(fordi $\log|\mathbf{E}| < \mathbf{2}\cdot\log|\mathbf{V}|$)

- Totalt tidsforbruk er $\mathcal{O}(|\mathbf{E}|\cdot\log|\mathbf{V}|)$

Prim vs. Kruskal

- Prims algoritme er noe mer effektiv enn Kruskals, spesielt for tette grafer
- Prims algoritme virker bare i sammenhengende grafer
- Kruskas algoritme gir et minimalt spenn-tre i hver sammenhengskomponent i grafen

Dybde-først søk

Dybde-først søk

- klassisk graf traversering
- generalisering av prefiks traversering for trær
- gitt: start node v: rekursivt traverserer alle nabonodene
- rekursjon $\Rightarrow$ vi undersøker alle noder som kan nåes fra første etterfølger til **v**, før vi undersøker *neste* etterfølger til **v**
- for vilkårlige grafer: pass på å terminerer rekursjon!

$\Rightarrow$ unvisited $\leftrightarrow$ visited nodes.

DFS

Initialize: all nodes "unvisited".

Recur:
- when visited: return immidiately
- when unvisited
  - set to "visited"
  - recur on all neighbors

Dybde-først søk

- klassisk graf traversering
- generalisering av prefiks traversering for trær
- gitt: start node v: rekursivt traverserer alle nabonodene
- rekursjon $\Rightarrow$ vi undersøker alle noder som kan nåes fra første etterfølger til **v**, før vi undersøker *neste* etterfølger til **v**
- for vilkårlige grafer: pass på å terminerer rekursjon!

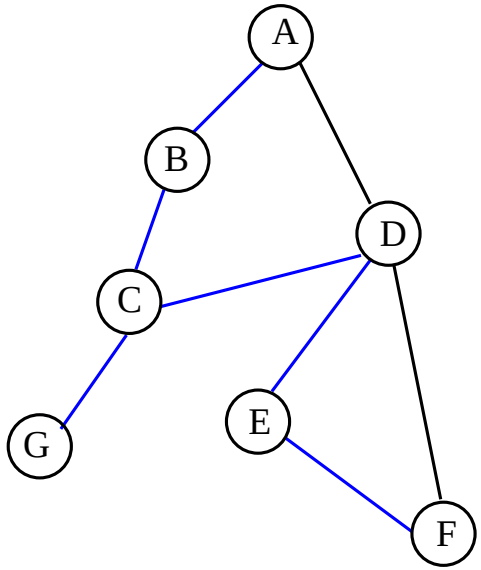
$\Rightarrow$ unvisited $\leftrightarrow$ visited nodes.

DFS

Initialize: all nodes "unvisited".

Recur:
- when visited: return immidiately
- when unvisited
  - set to "visited"
  - recur on all neighbors

```
void dybdeFørstSøk(Node v) {
    v.merke = true;
    for < hver nabo w til v > {
        if (!w.merke) {
            dybdeFørstSøk(w);
        }
    }
}
```

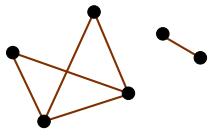


Løkkeleting

- Vi kan bruke dybde-først søk til å sjekke om en graf har løkker
- 3 verdier til tilstandsvariablen: usett, igang og ferdig (besøkt)

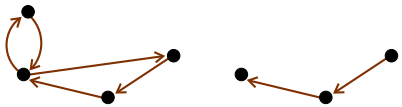
```
void løkkeLet(Node v) {
    if (v.tilstand == igang) {
        < Løkke er funnet >
    } else if (v.tilstand == usett) {
        v.tilstand = igang;
        for < hver nabo w til v > {
            løkkeLet(w);
        }
        v.tilstand = ferdig;
    }
}
```

Er grafen sammenhengende?



urettet graf & DFS

En urettet graf er sammenhengende hvis og bare hvis et dybde-først søk som starter i en tilfeldig node, besøker alle nodene i grafen

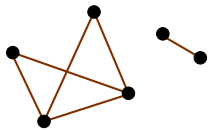


rettet graf & DFS

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

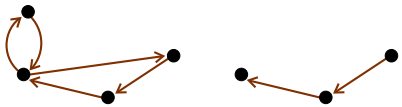
Hvis grafen ikke er sammenhengende, kan vi foreta nye dybde-først søk fra noder som ikke er besøkte, inntil alle nodene er behandlet

Er grafen sammenhengende?



urettet graf & DFS

En urettet graf er sammenhengende hvis og bare hvis et dybde-først søk som starter i en tilfeldig node, besøker alle nodene i grafen



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**

Hvis grafen ikke er sammenhengende, kan vi foreta nye dybde-først søk fra noder som ikke er besøkte, inntil alle nodene er behandlet

Urettet grafer

Dybde-først-spenntre

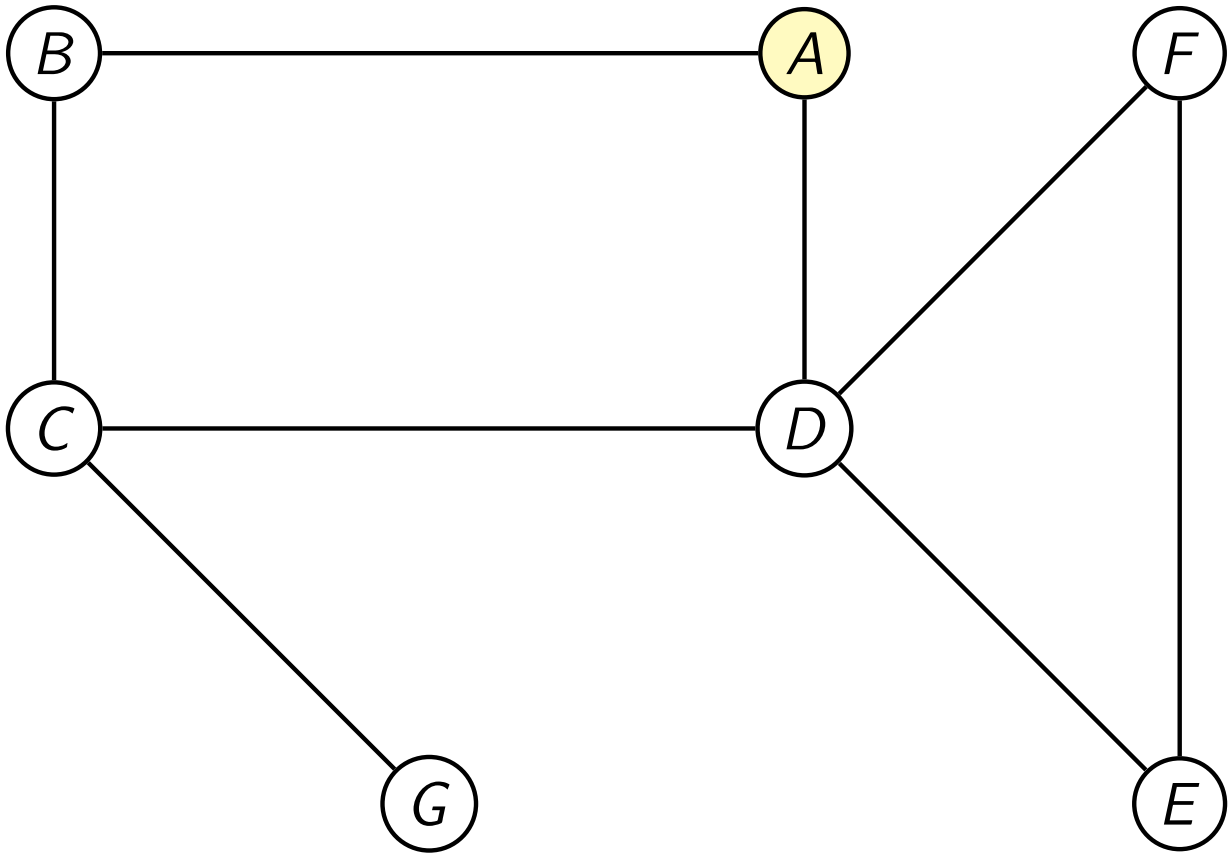
- for urettet sammenhengende grafer
- *ikke sammenhengende* grafer: dfs spanning forest
- huske "back-pointers"

Ulike type kanter

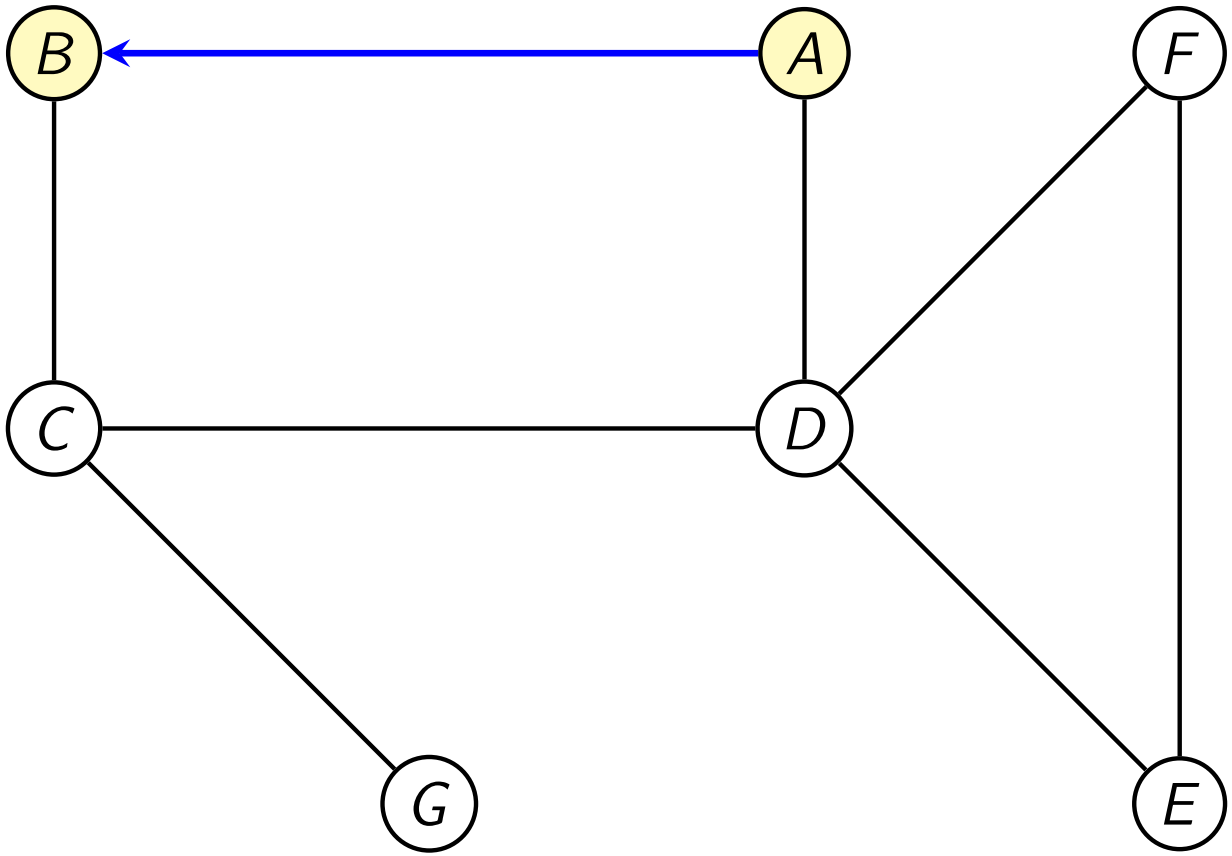
gitt en bestemt kjøring av dfs

1. tree edges
2. back edges

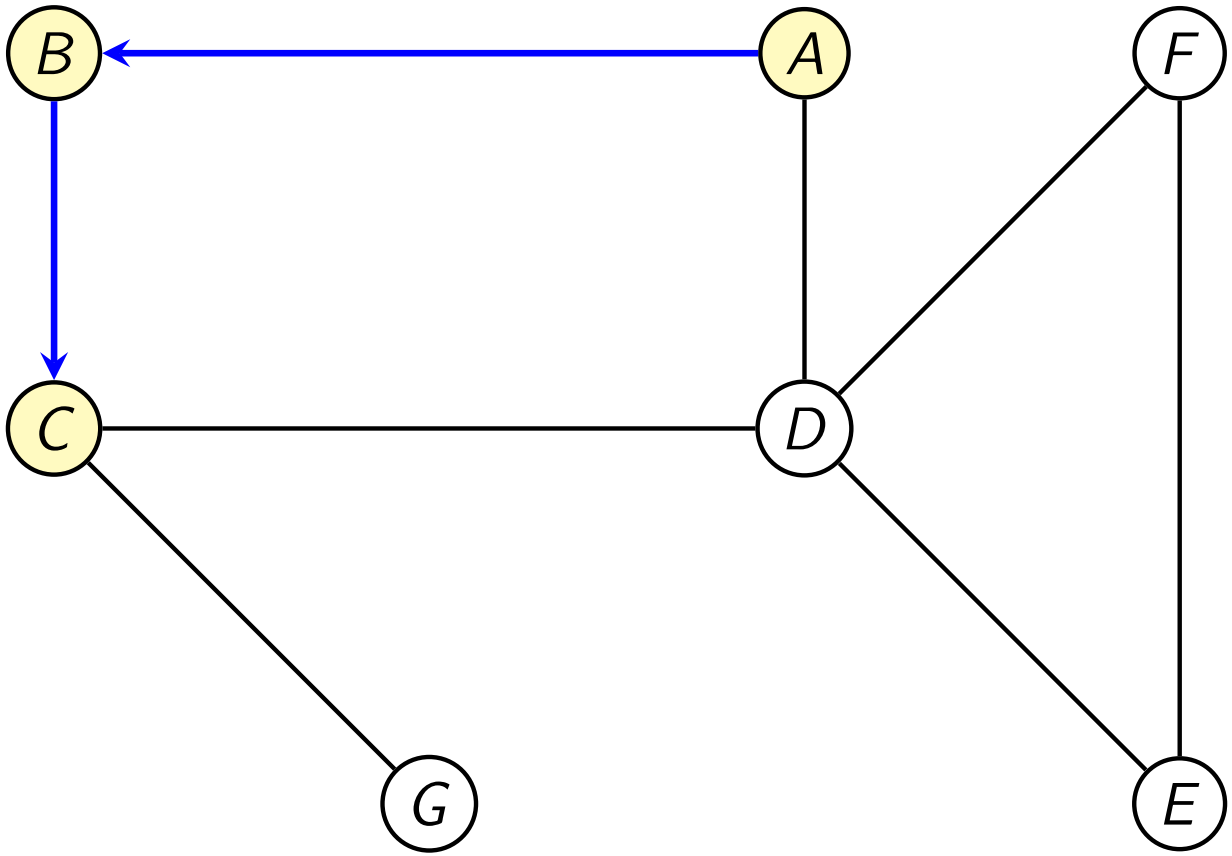
DFS, urettet graf (1)



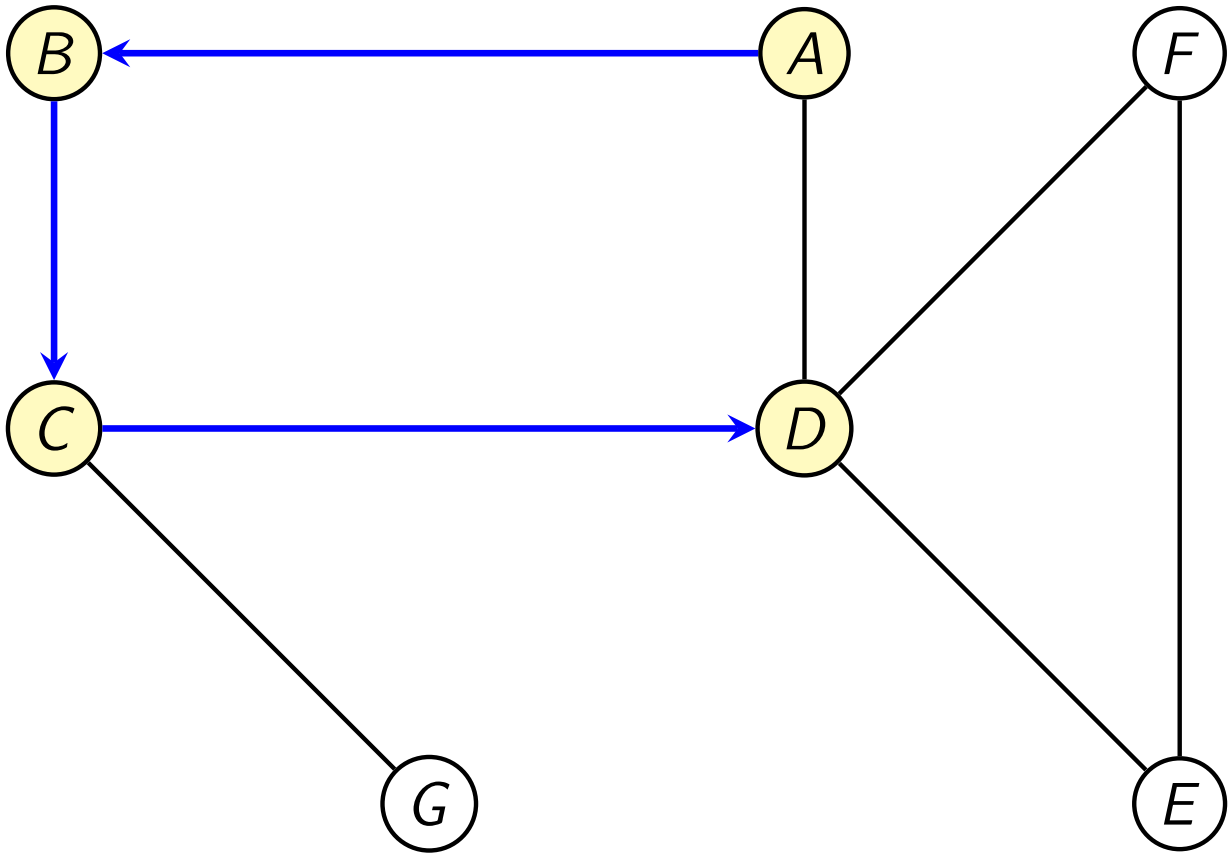
DFS, urettet graf (2)



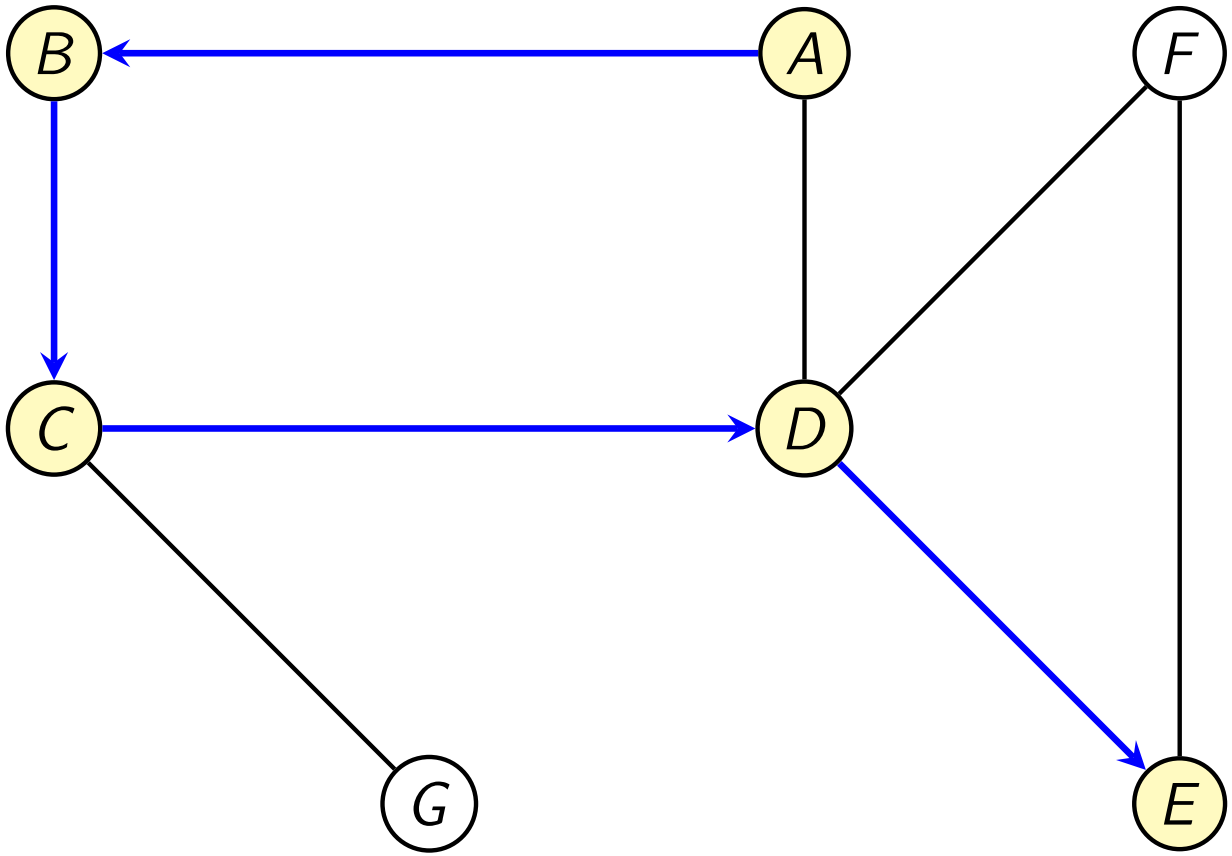
DFS, urettet graf (3)



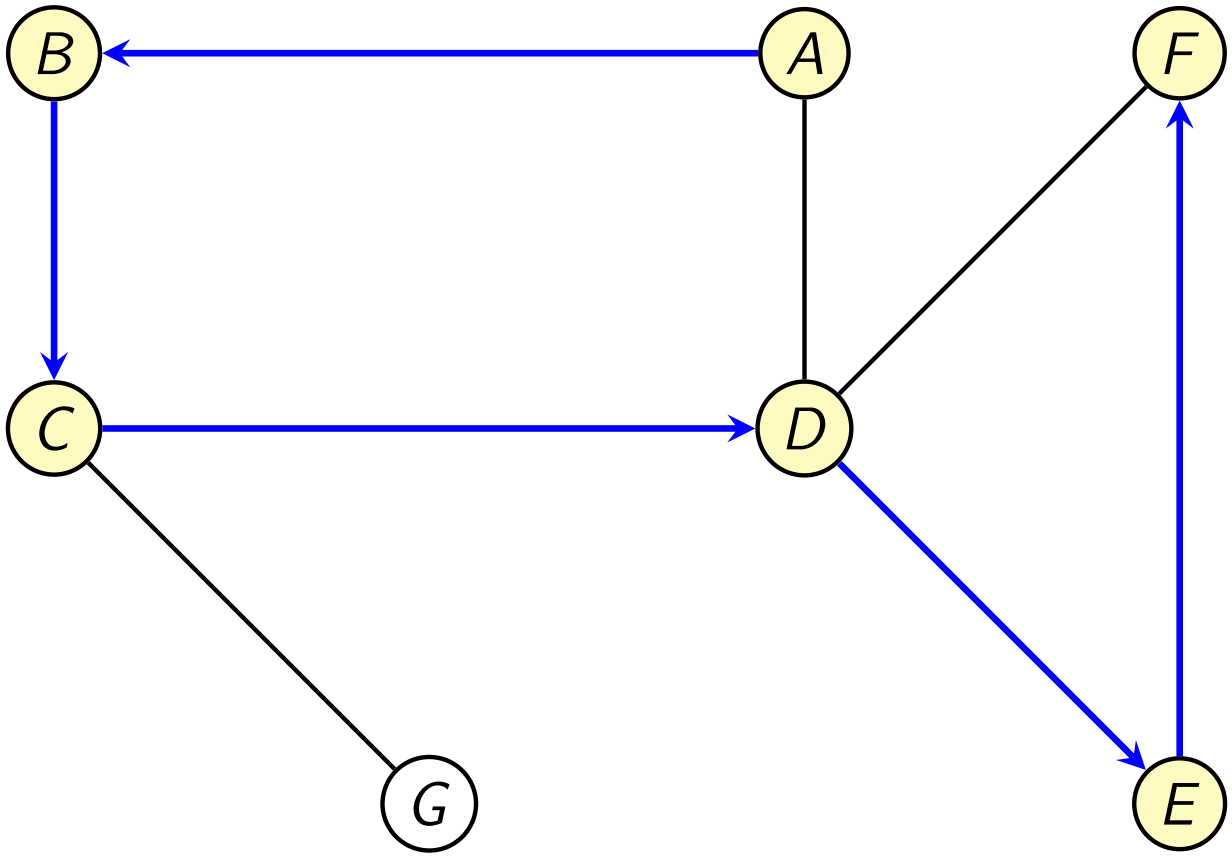
DFS, urettet graf (4)



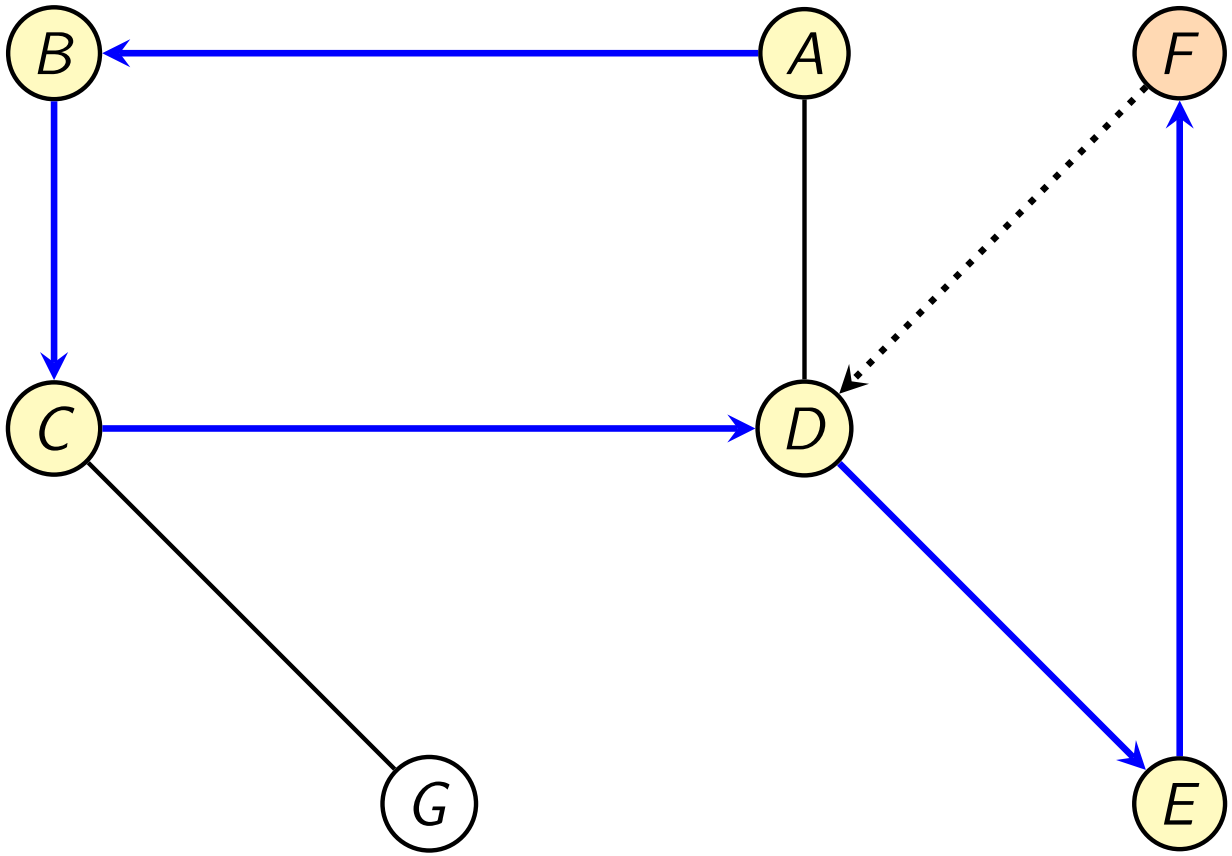
DFS, urettet graf (5)



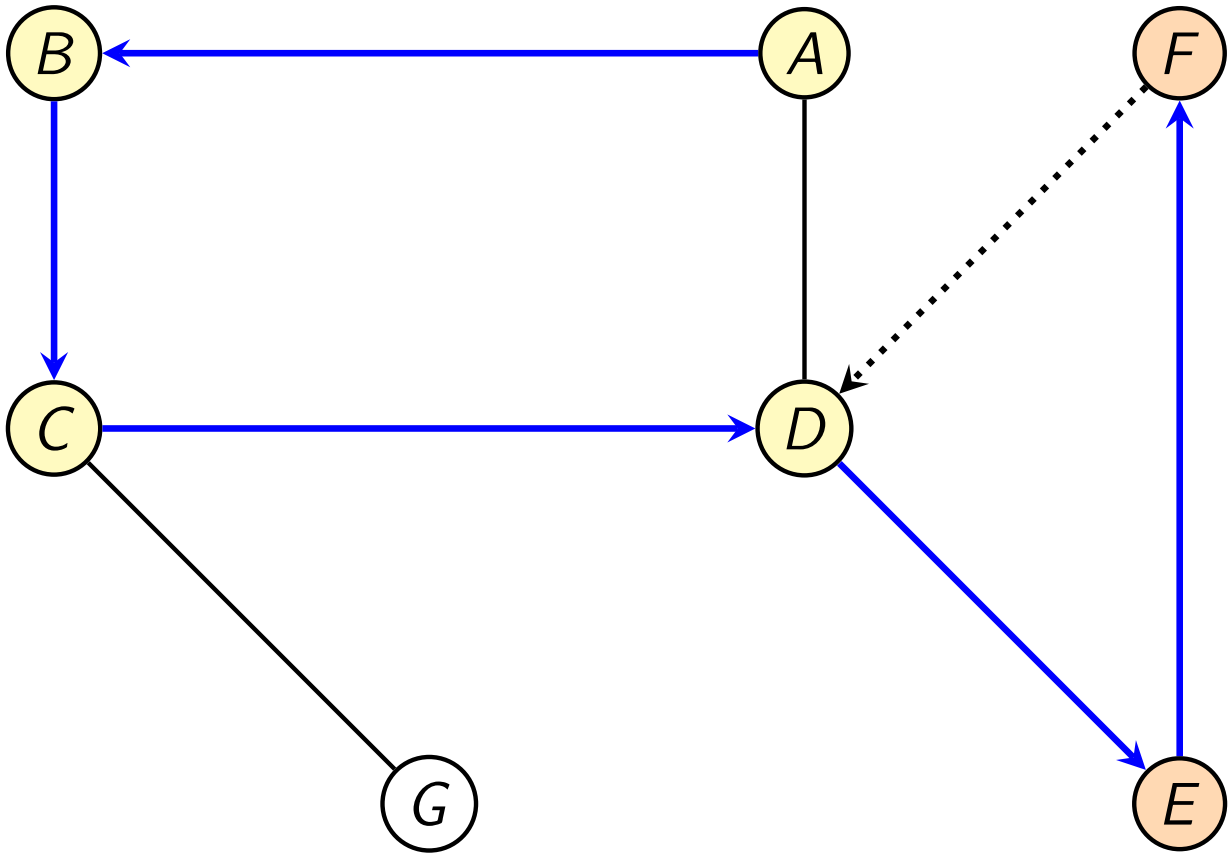
DFS, urettet graf (6)



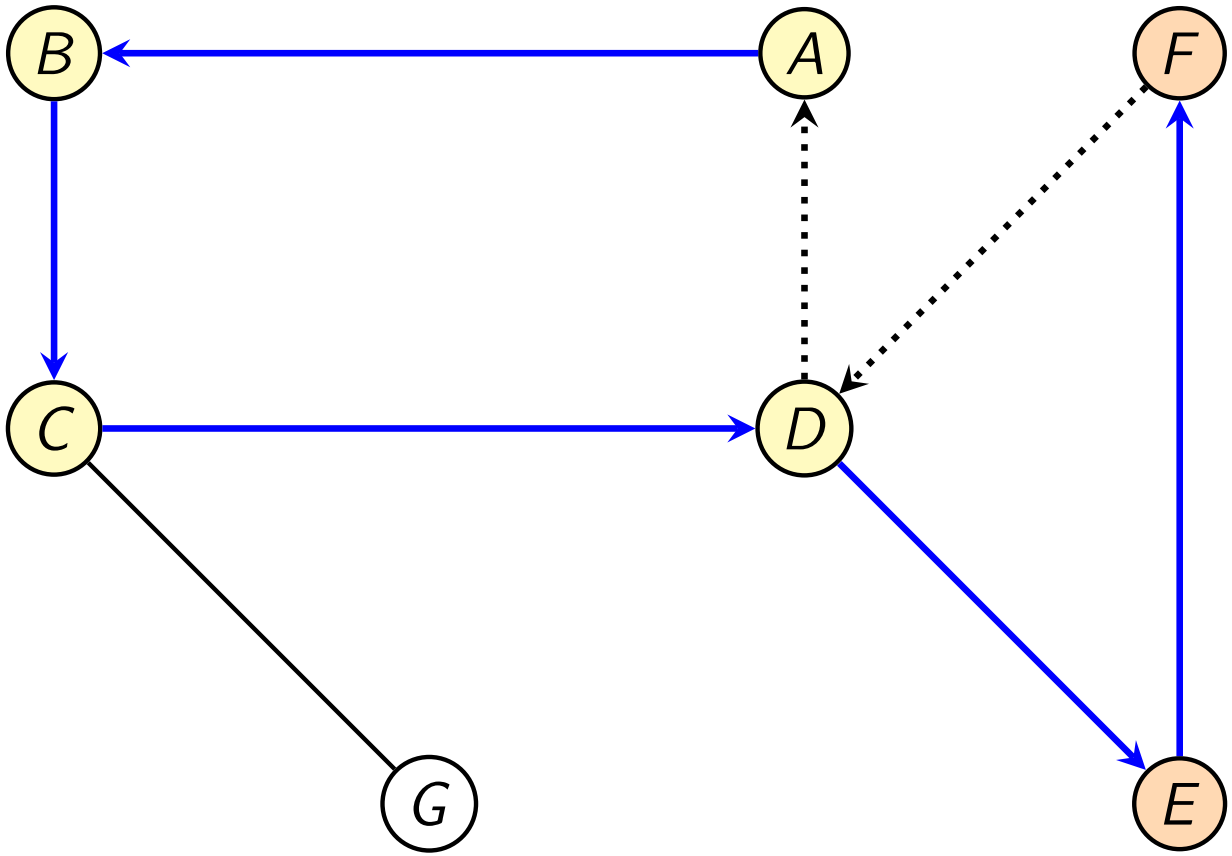
DFS, urettet graf (7)



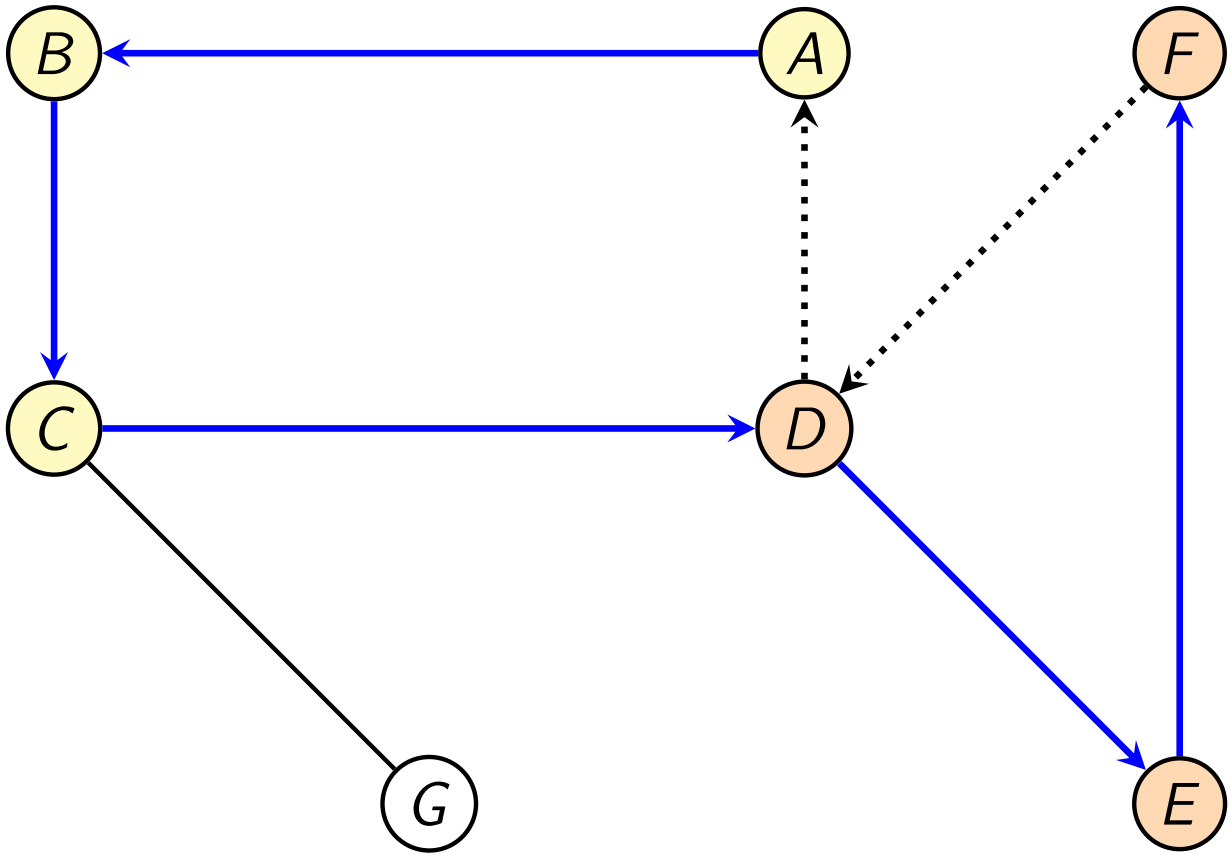
DFS, urettet graf (8)



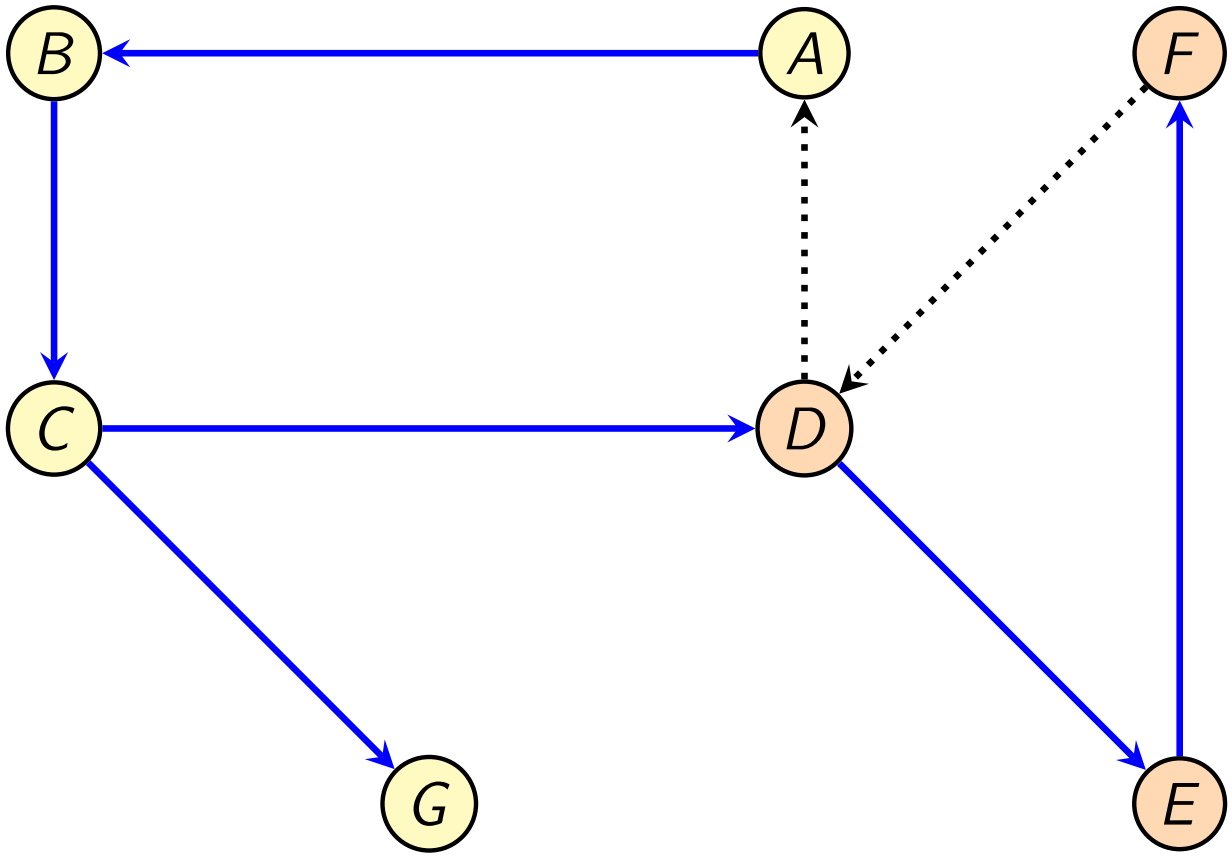
DFS, urettet graf (9)



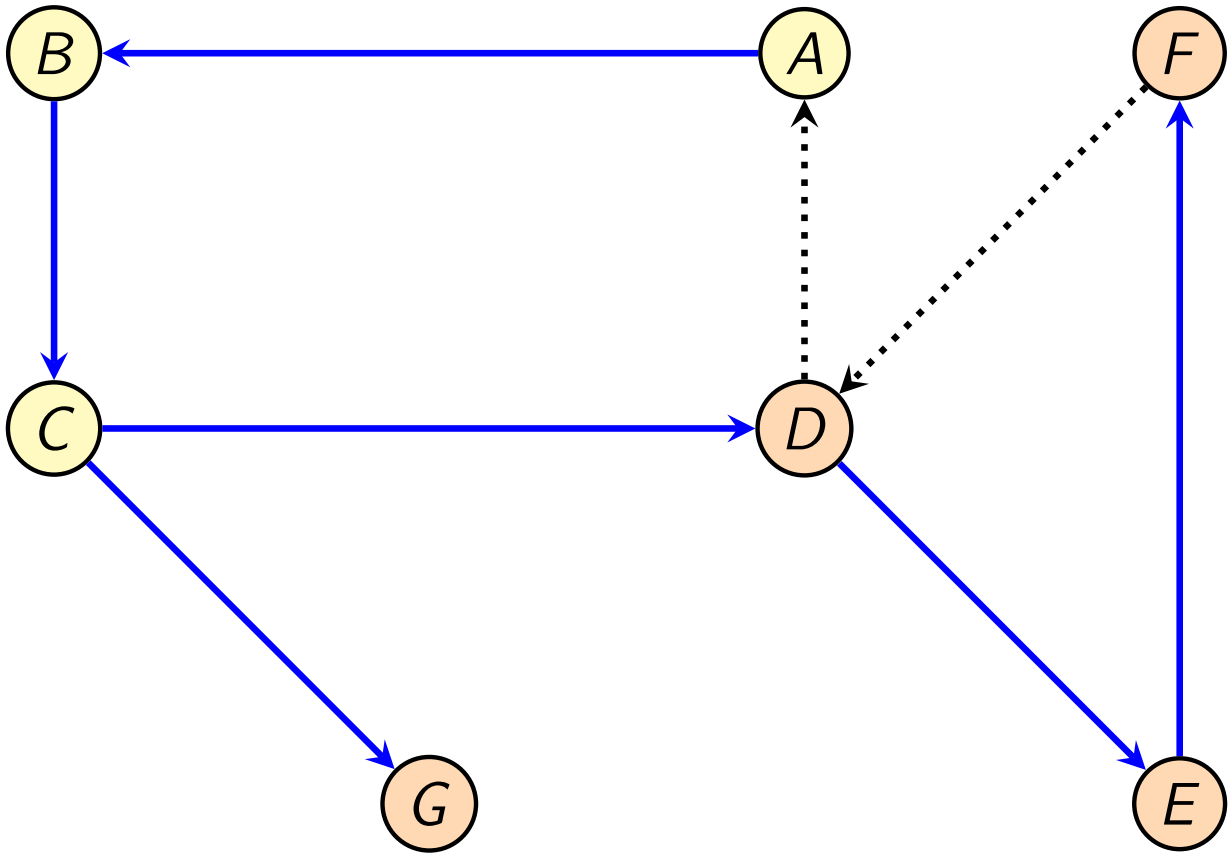
DFS, urettet graf (10)



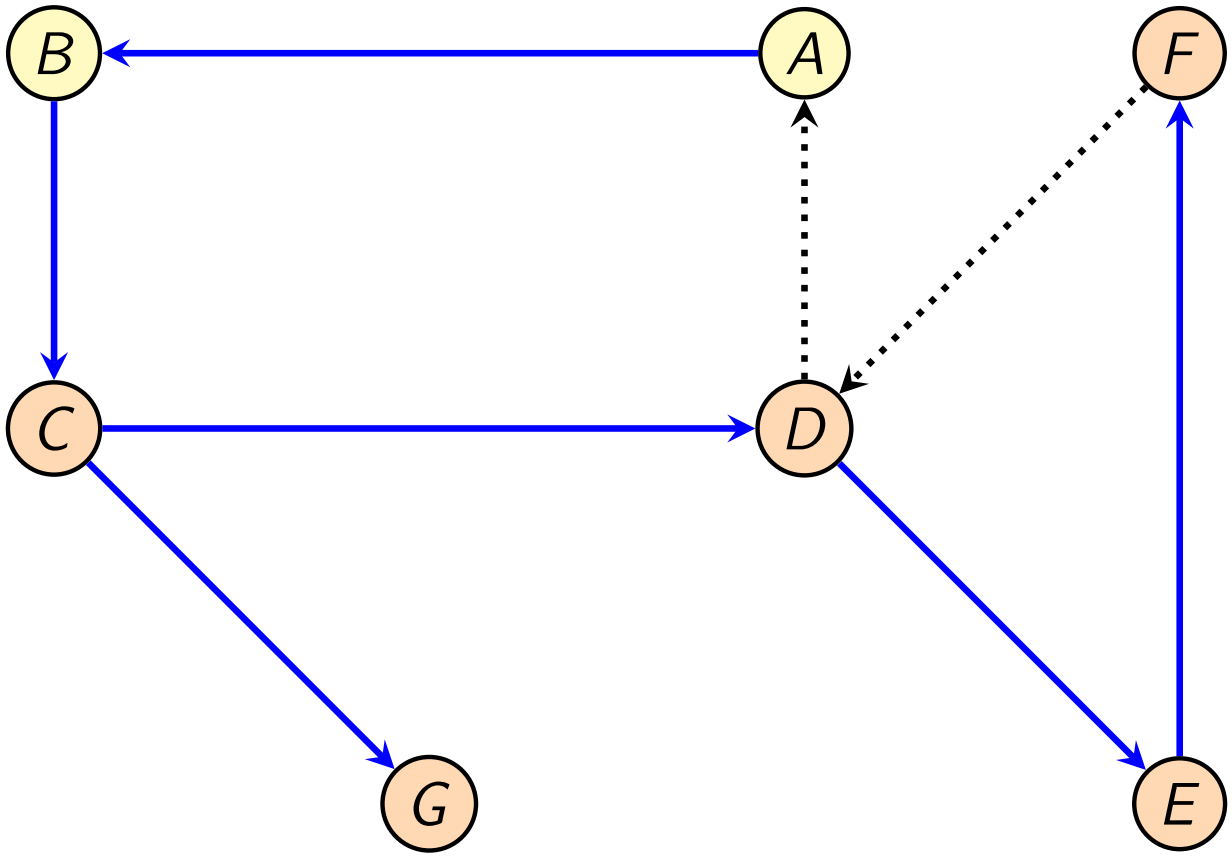
DFS, urettet graf (11)



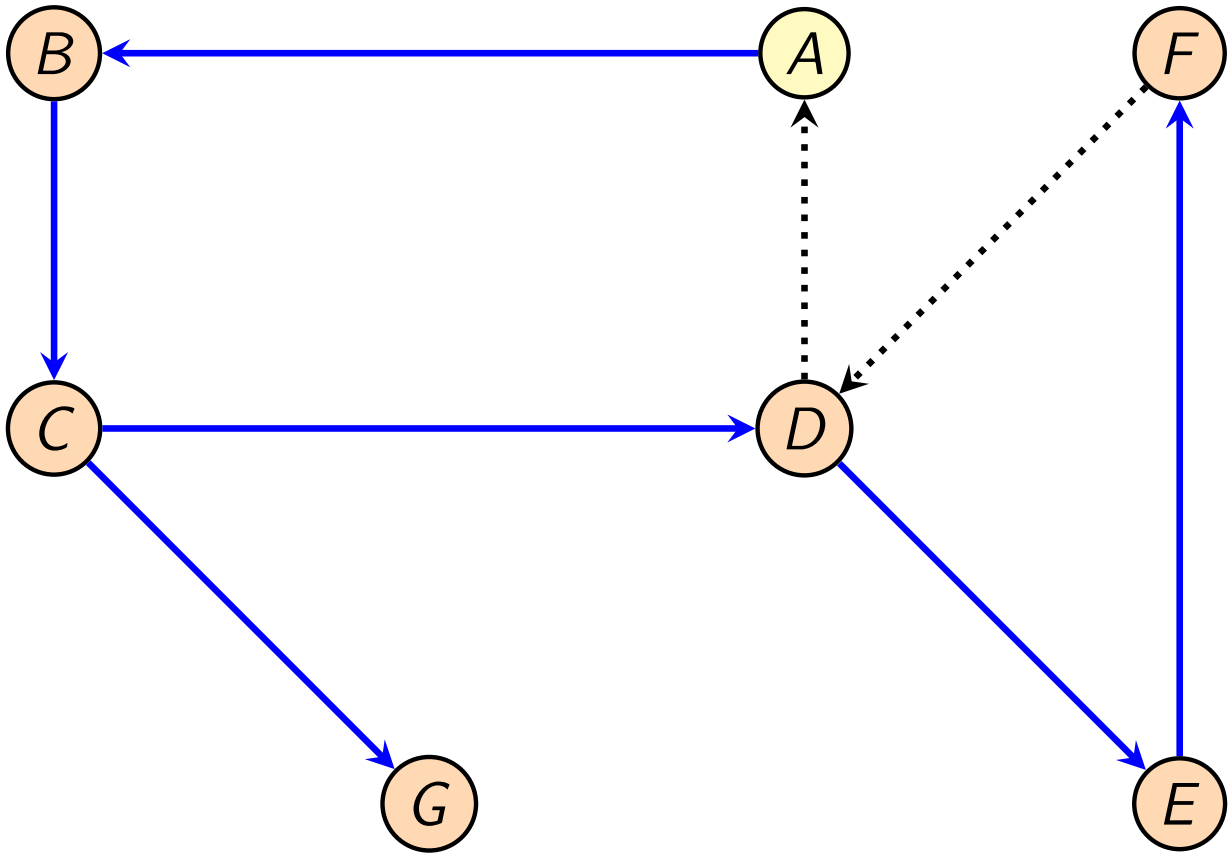
DFS, urettet graf (12)



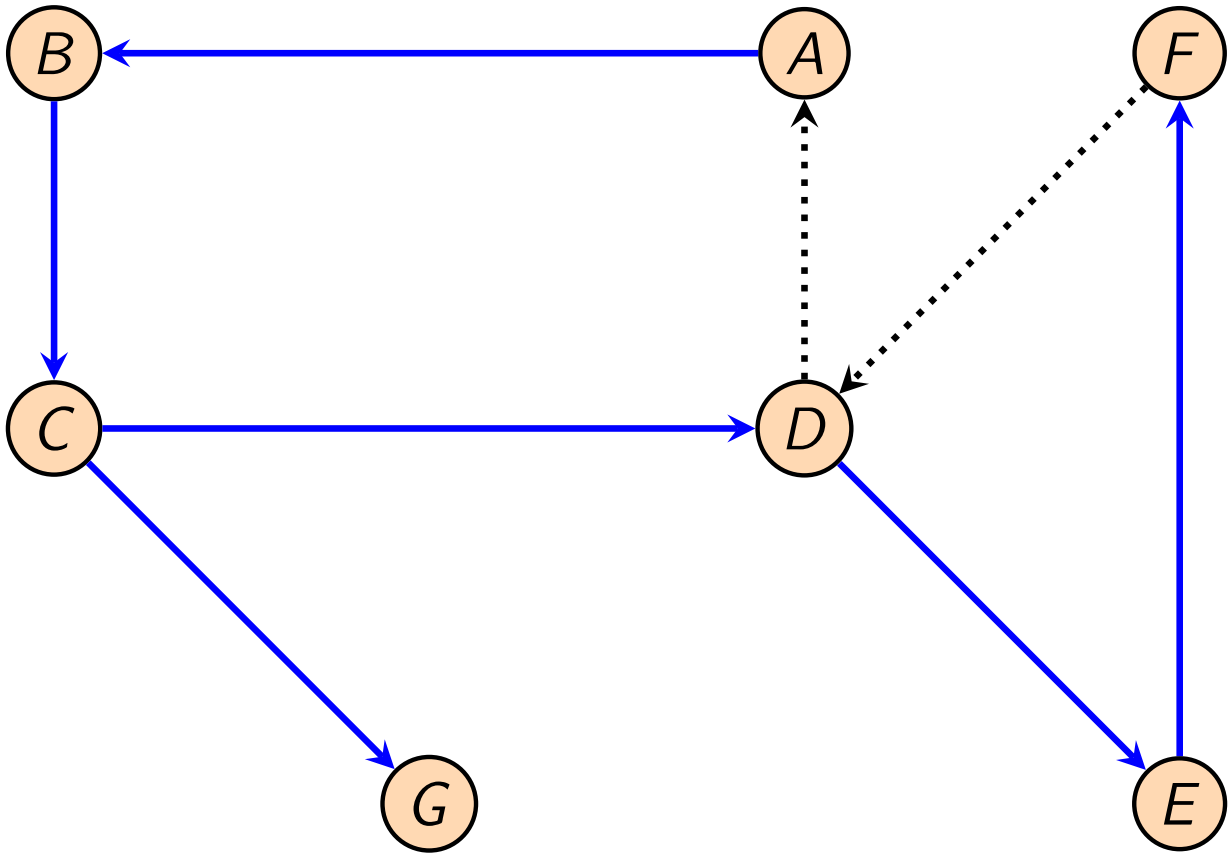
DFS, urettet graf (13)



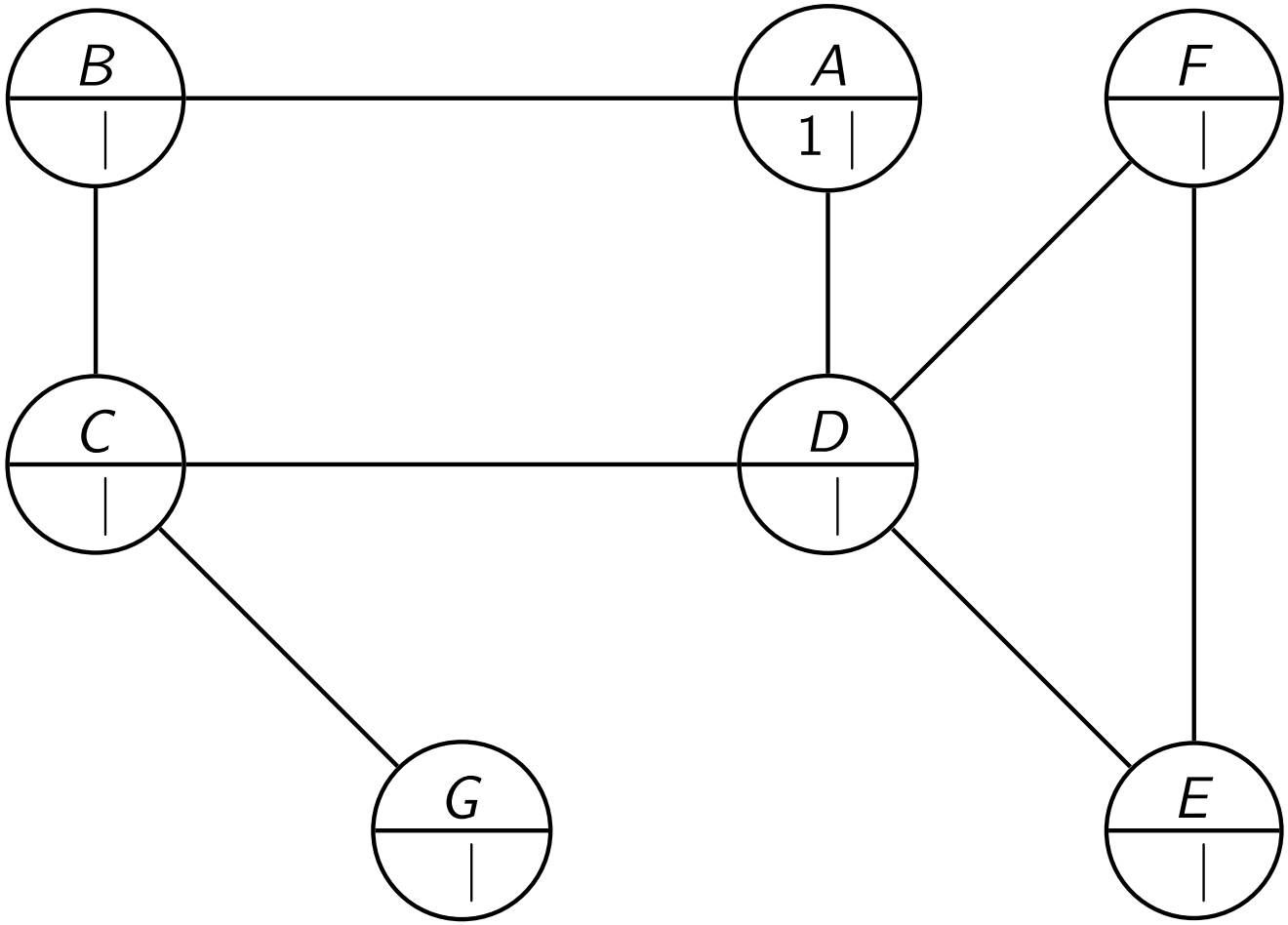
DFS, urettet graf (14)



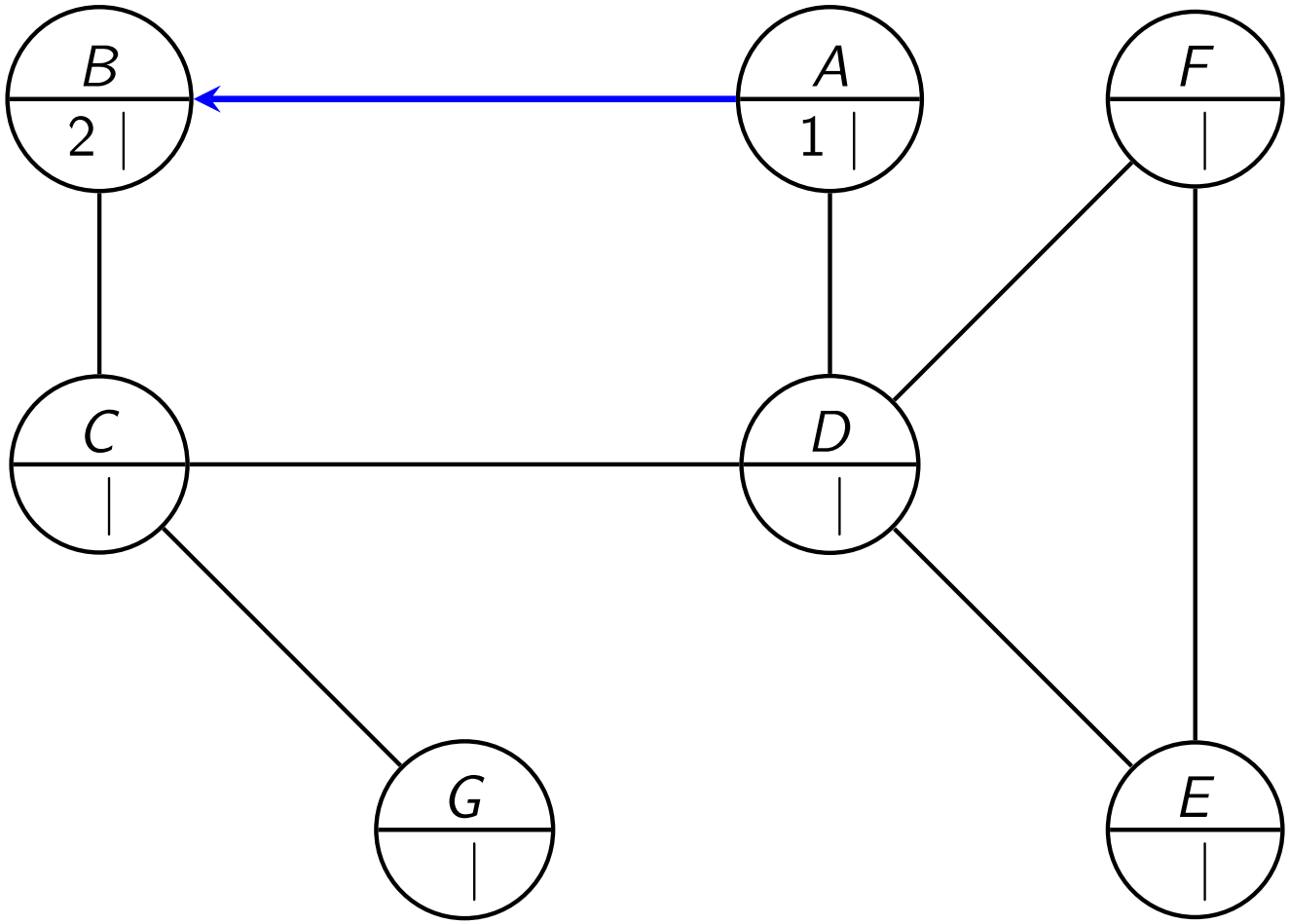
DFS, urettet graf (15)



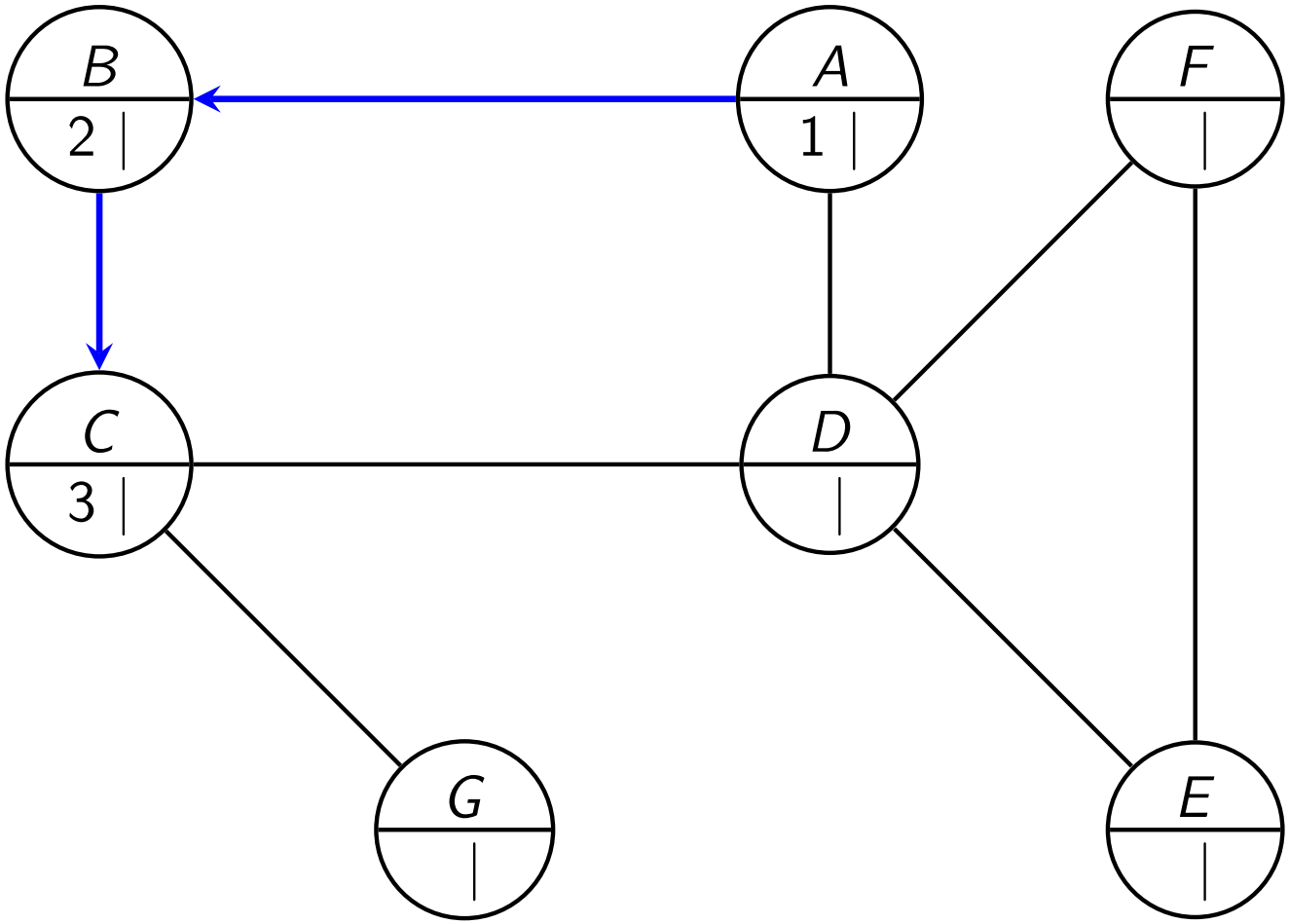
DFS: adding visiting time (1)



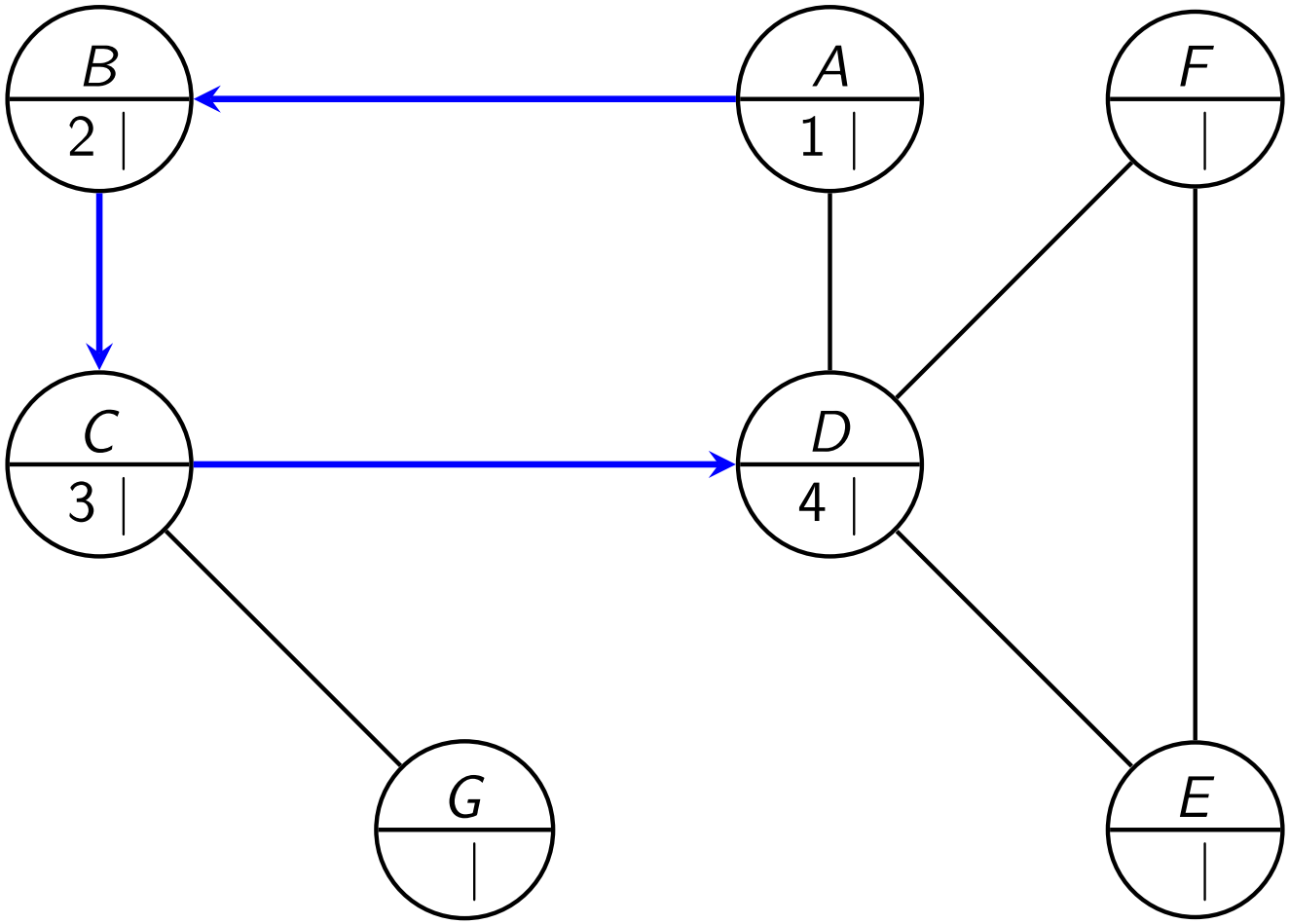
DFS: adding visiting time (2)



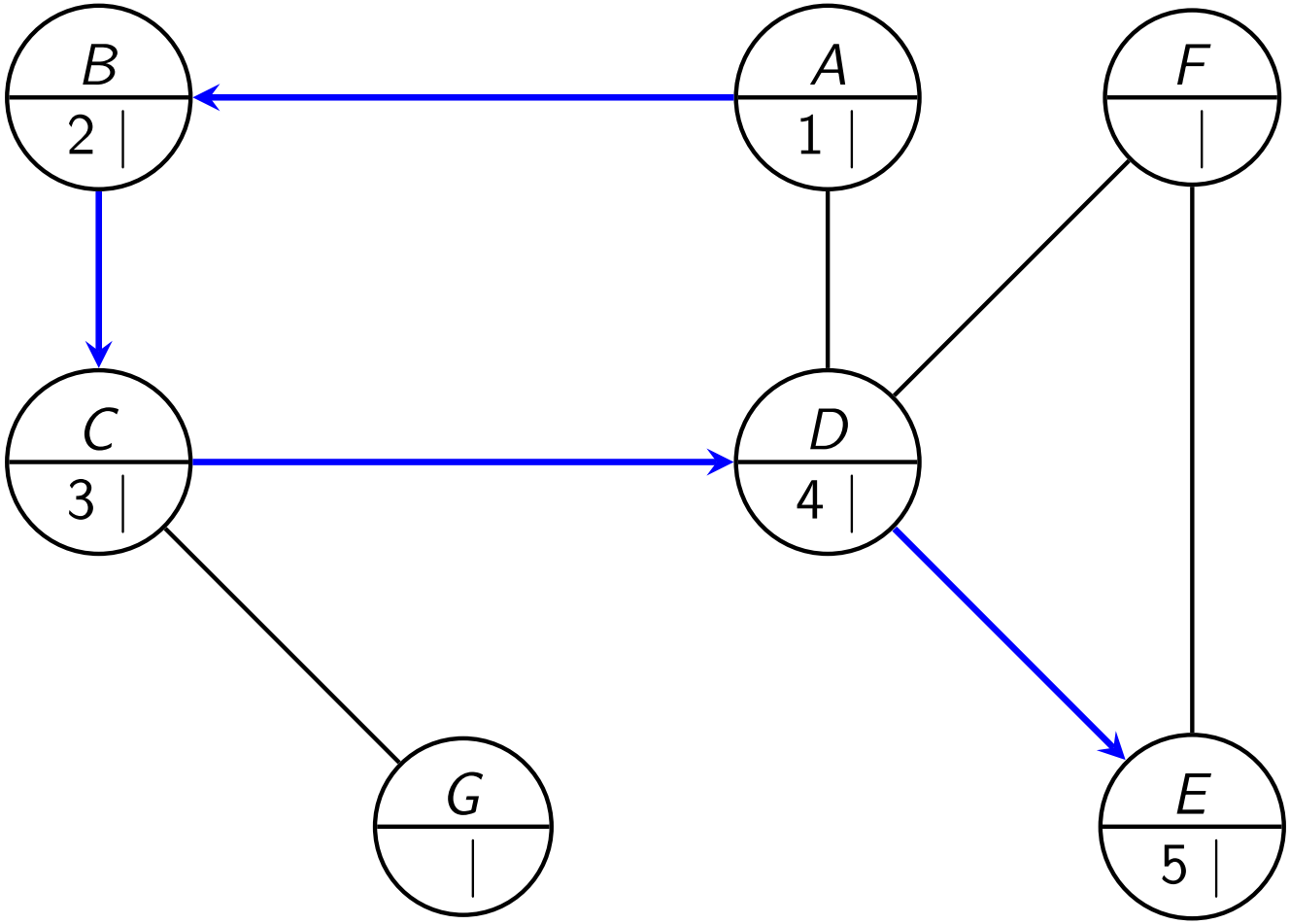
DFS: adding visiting time (3)



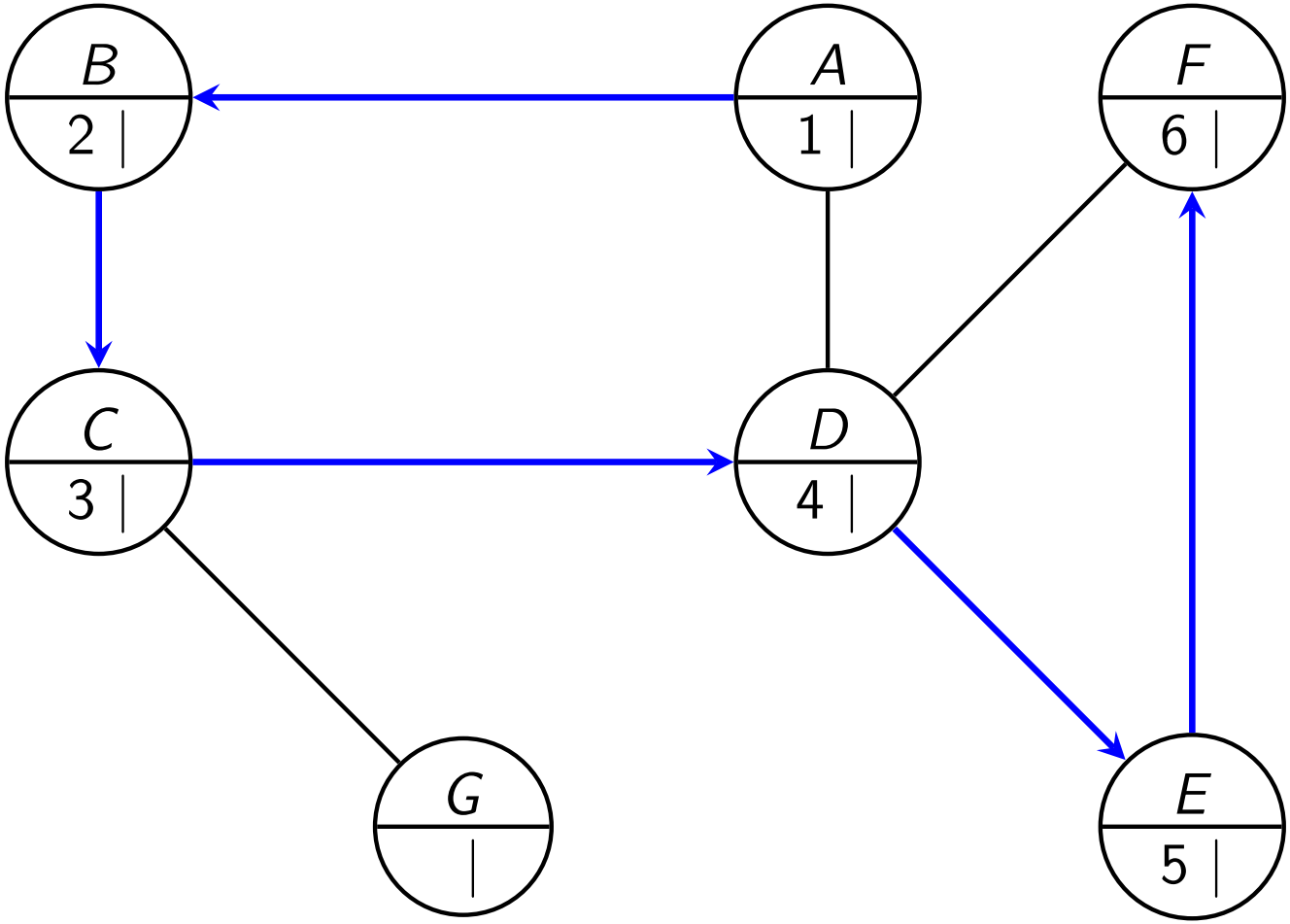
DFS: adding visiting time (4)



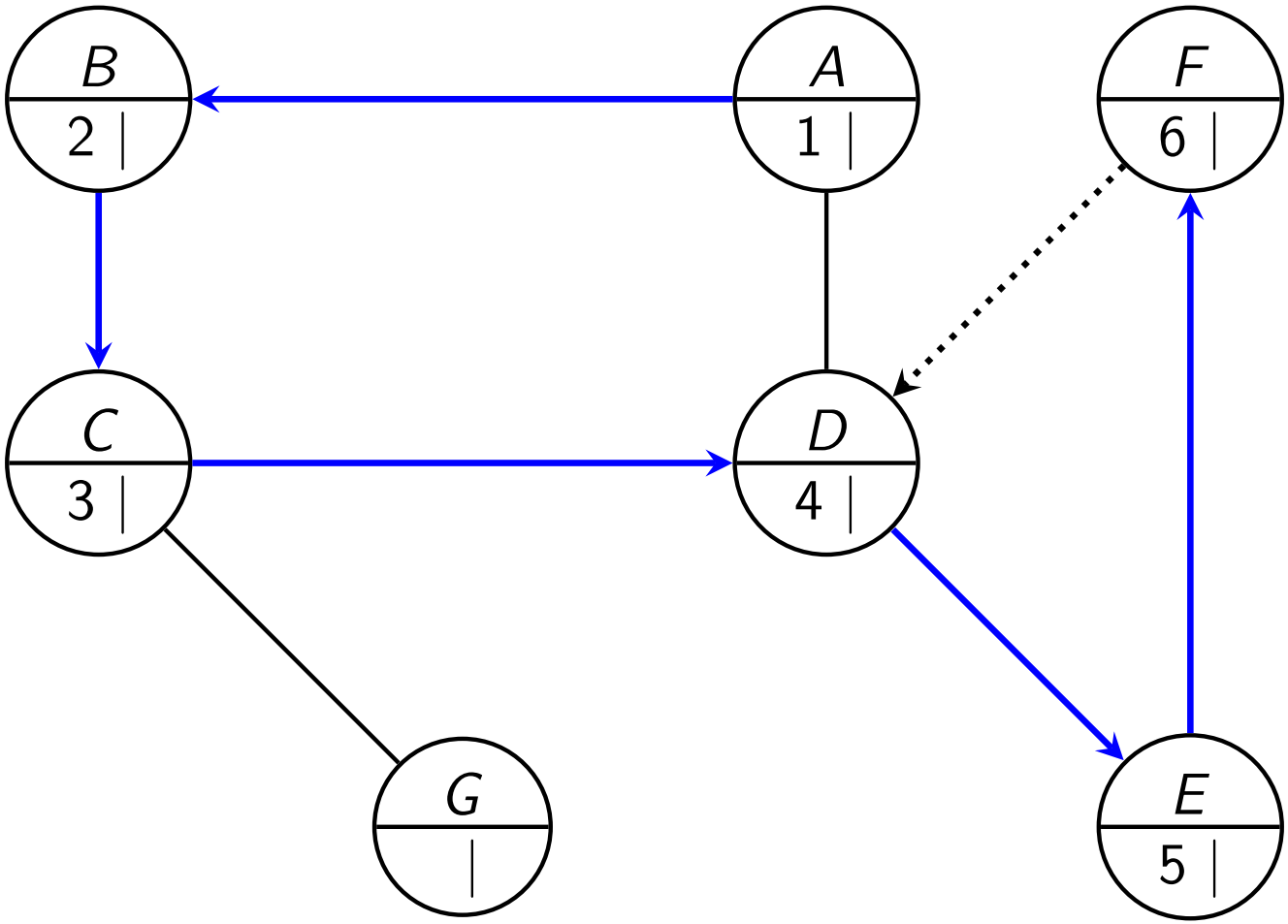
DFS: adding visiting time (5)



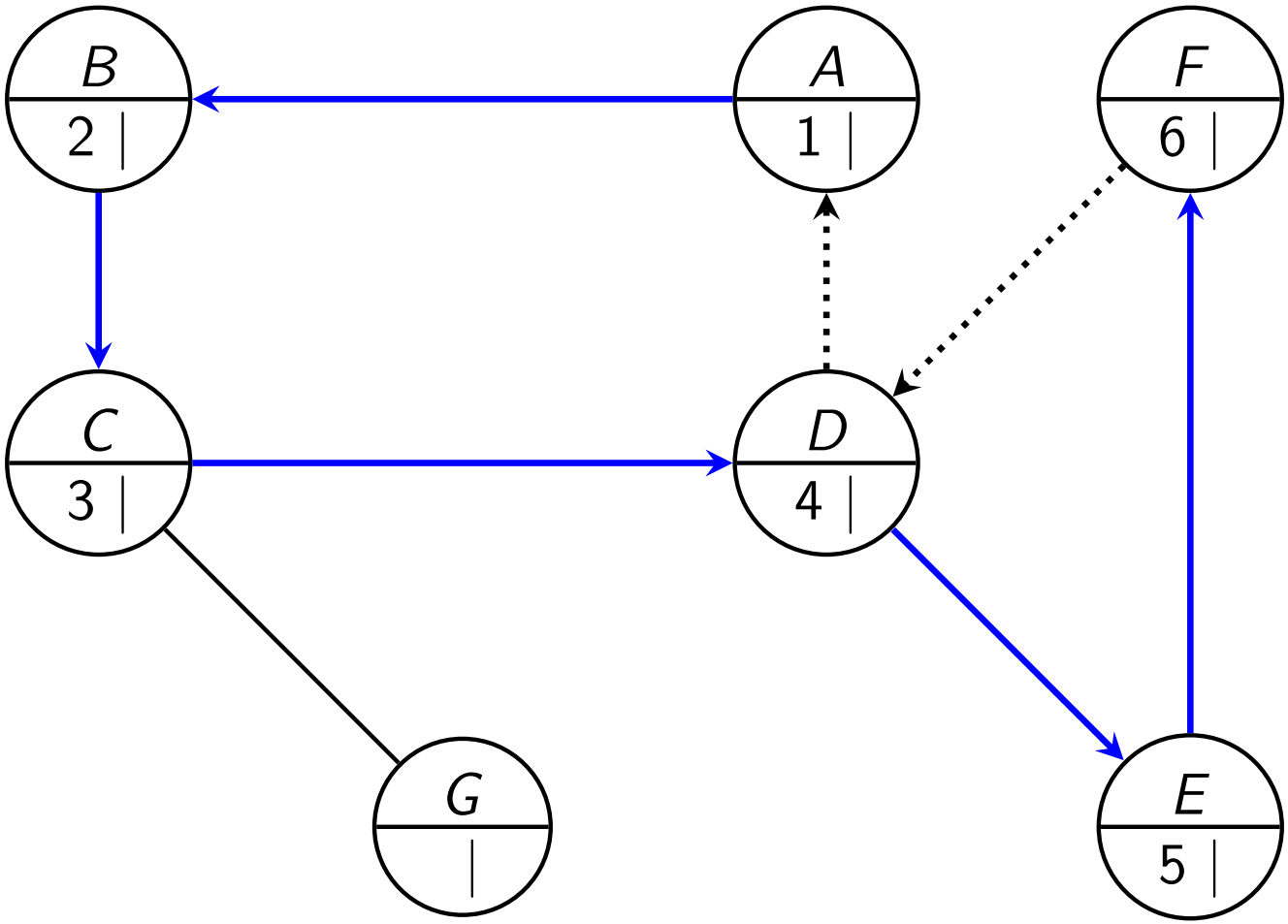
DFS: adding visiting time (6)



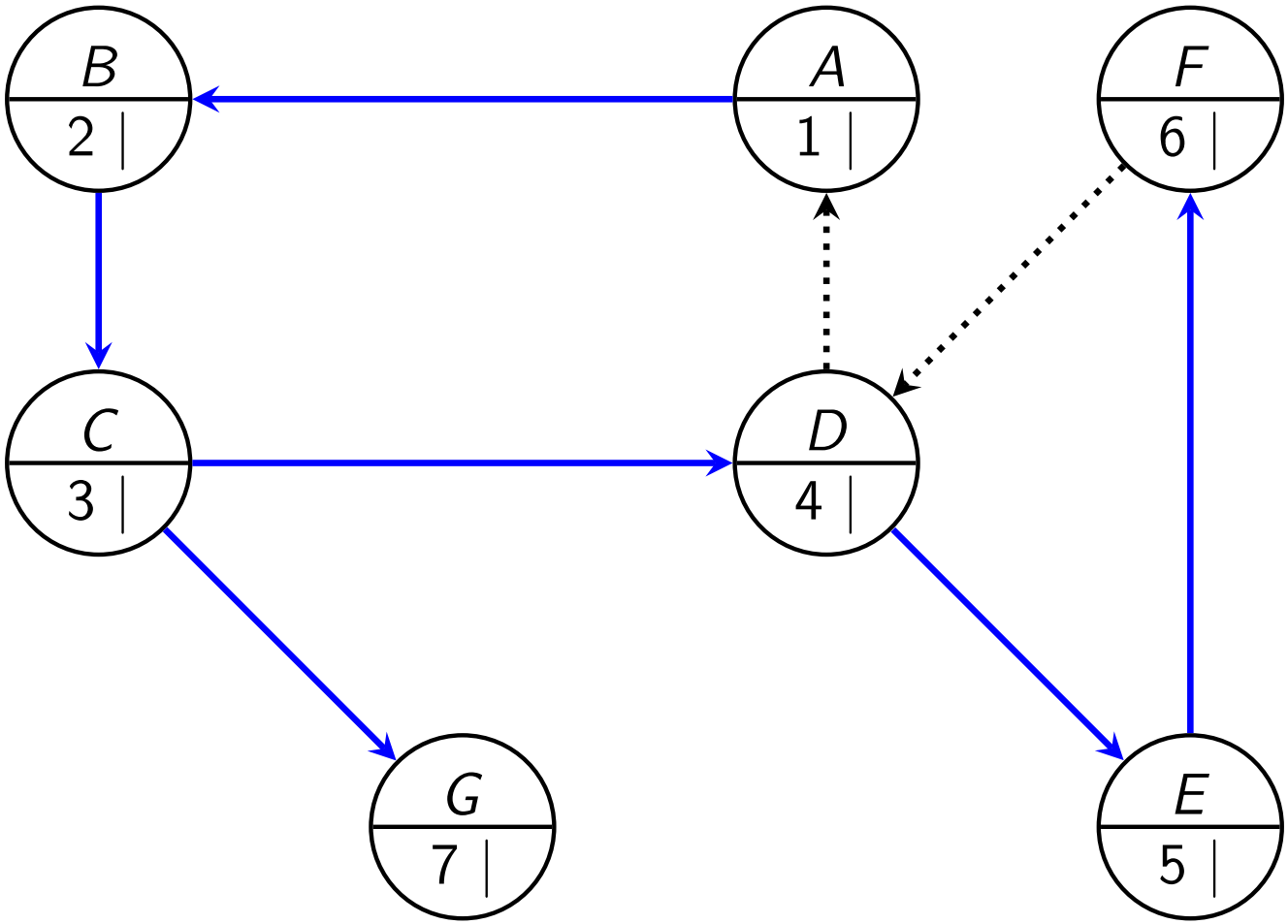
DFS: adding visiting time (7)



DFS: adding visiting time (8)



DFS: adding visiting time (9)



Biconnectivity

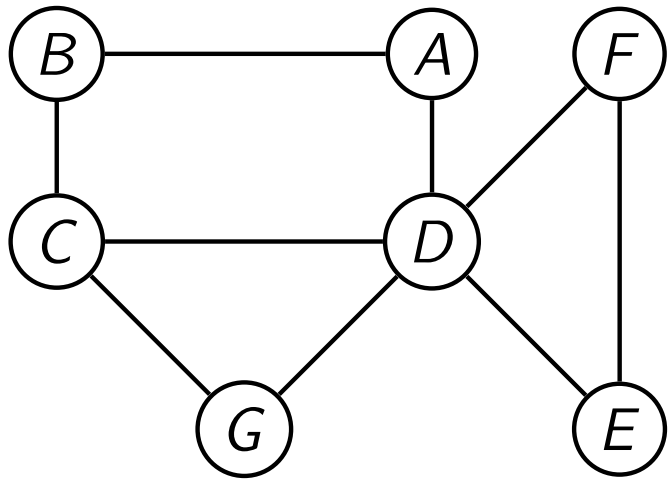
Definition

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 *articulation point*.

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

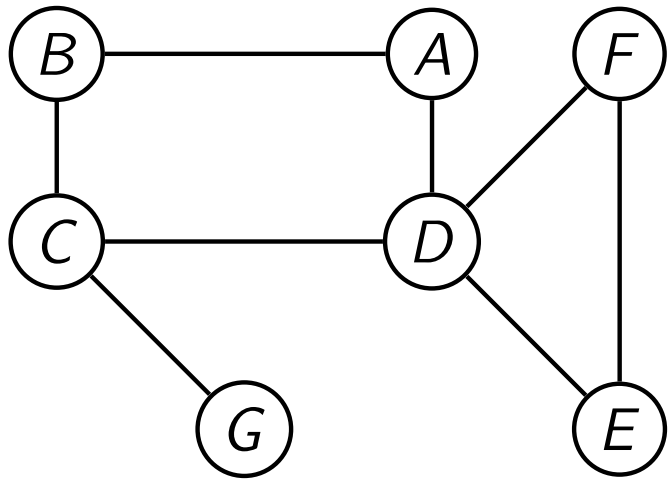
Definition

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 *articulation point*.

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

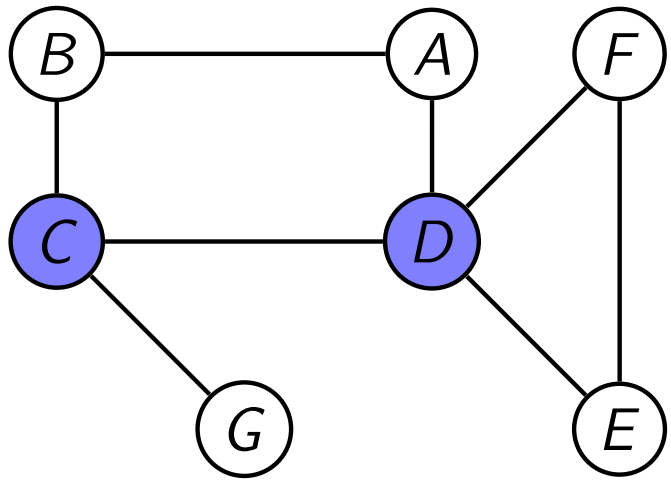
Definition

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 *articulation point*.

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!

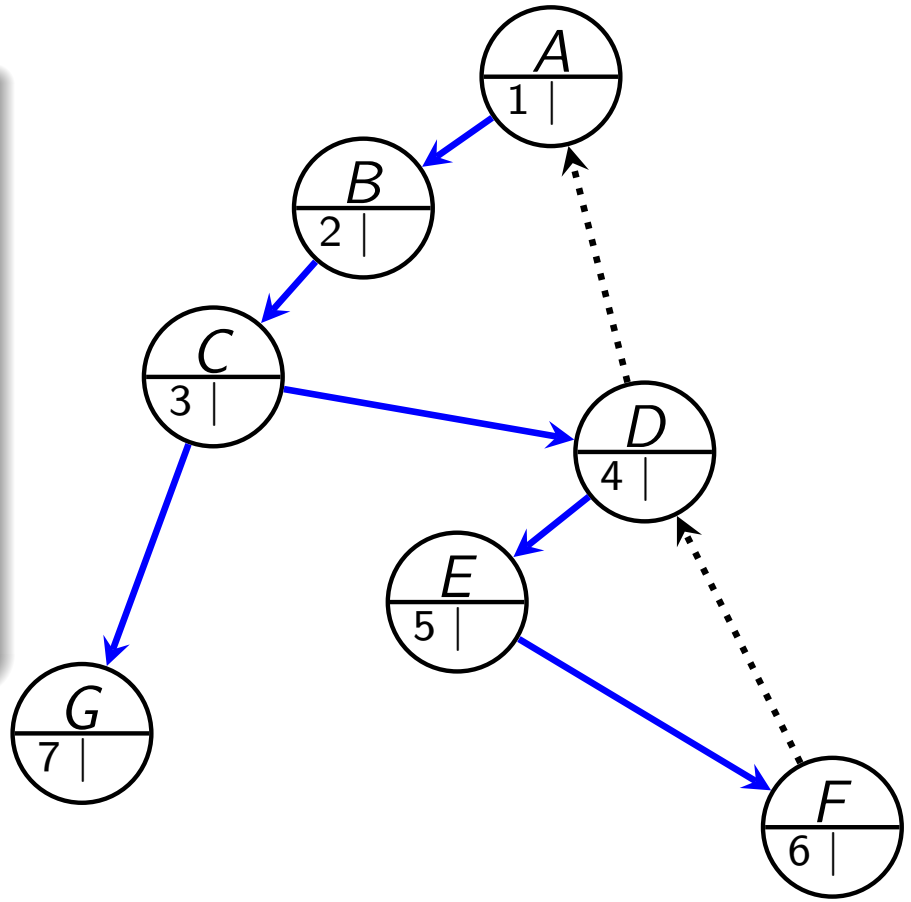


Er grafen bi-connected?

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

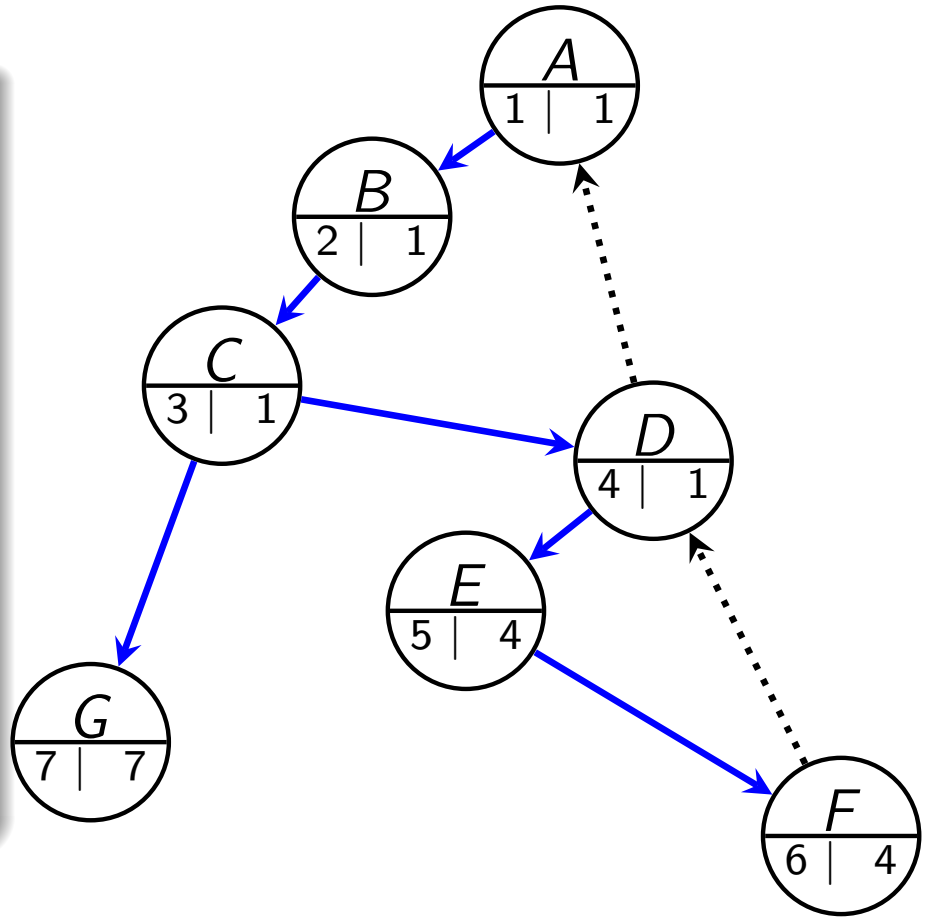


Er grafen bi-connected?

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

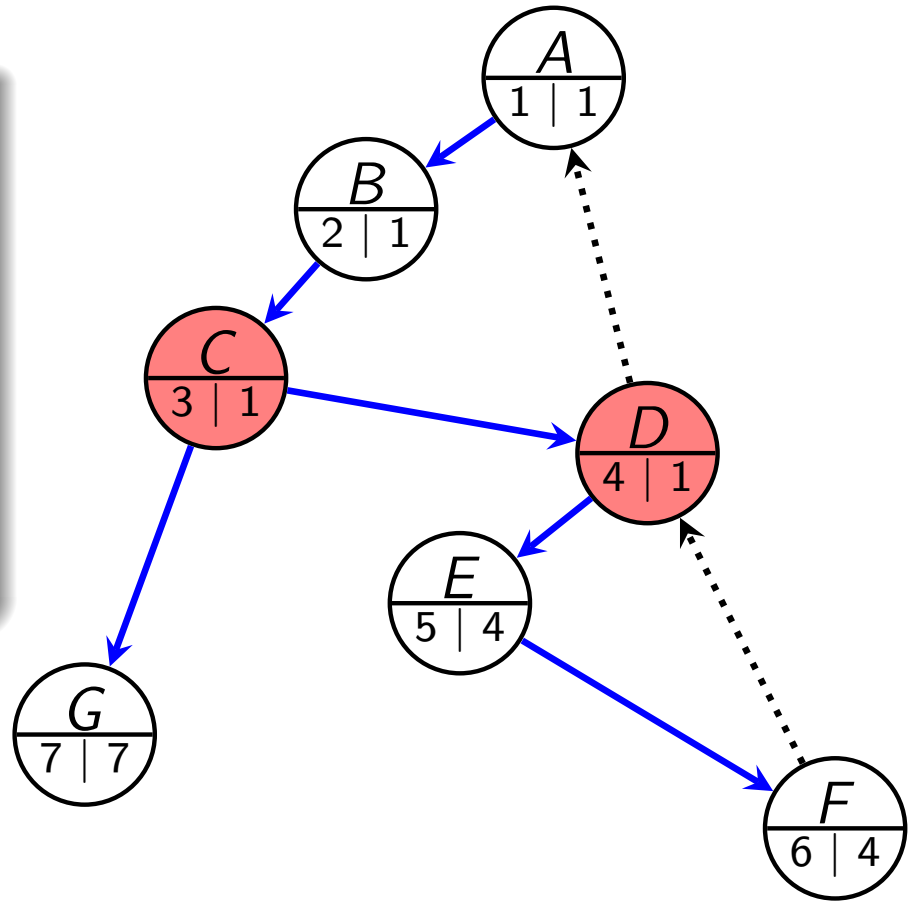


Er grafen bi-connected?

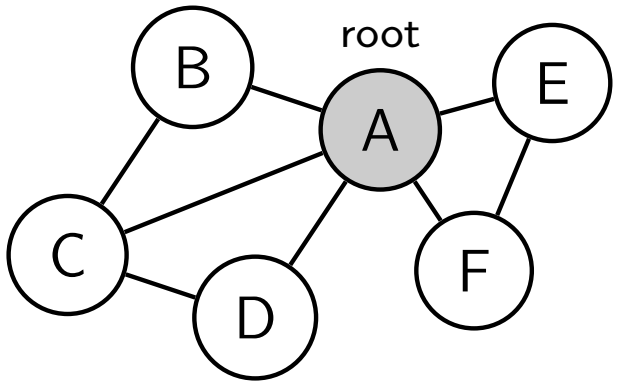
3. Cut-vertex

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 $\text{low}(w) \geq \text{num}(v)$



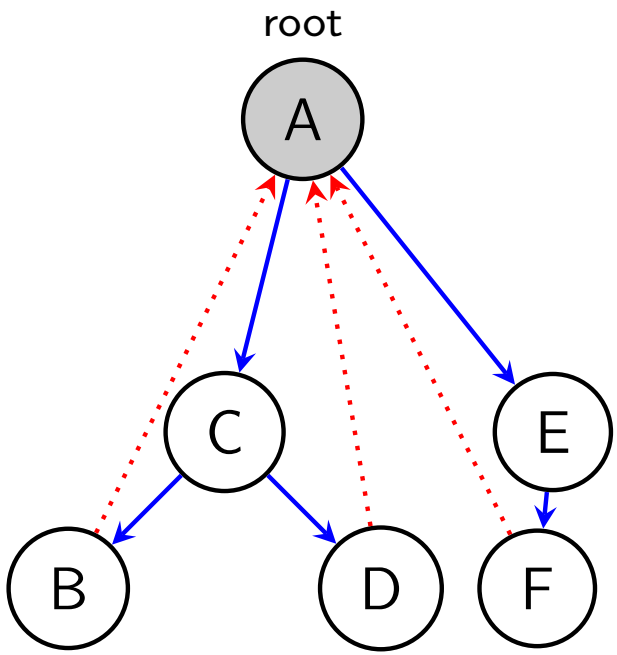
Rot



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

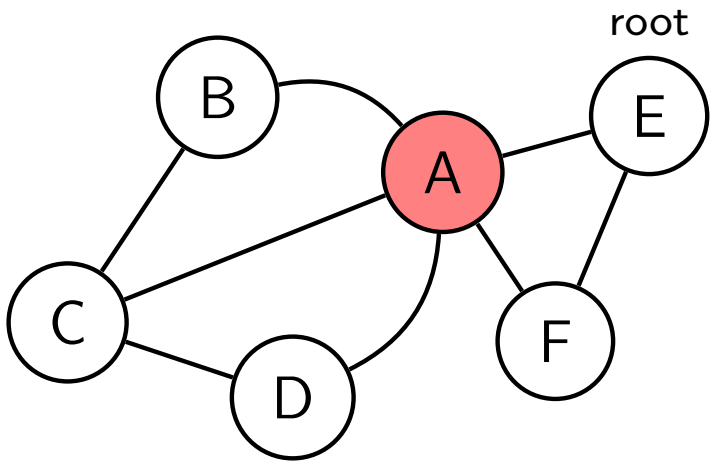


Roten av DFS-treet er en cutvertex hvis den har to eller flere utgående tree edges.

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

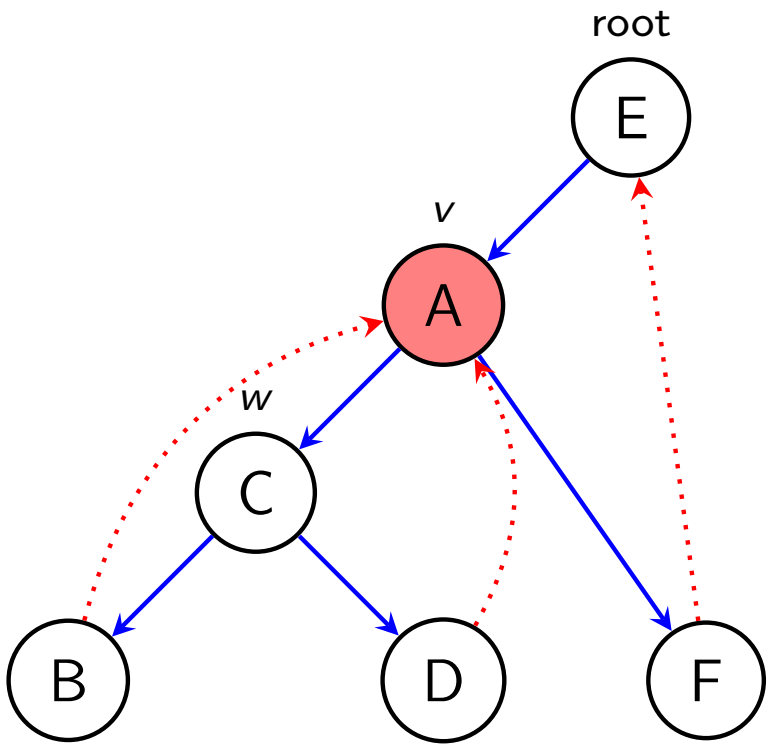
Ikke-rot node

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.

Neste forelesning: 1. oktober
Kombinatorisk søk og rekursjon

8. oktober:
Grafer III: Stongly connected components
Gjesteforelesning: Torbjørn Morland