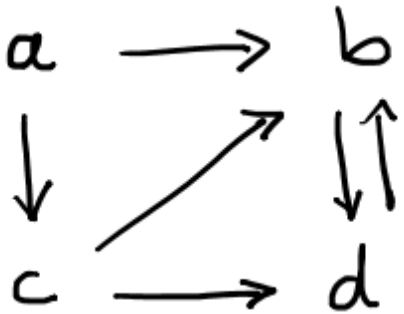
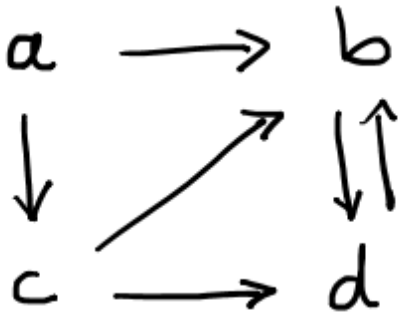
Informatik

Graphen: Adjazenlisten, Tiefensuche, Zusammenhangskomponenten

Implementation eines ungewichteten Graphen durch Adjazenzlisten, hier: Adjazenzmengen. Jedem Knoten wird in einem dictionary die Menge der Zielknoten der von ihm ausgehenden Kanten zugeordnet.

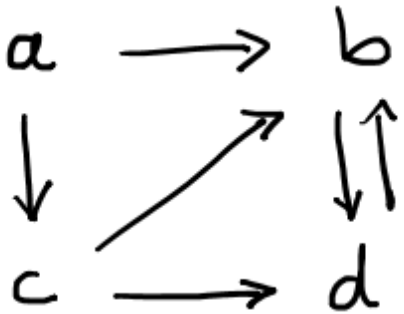


Implementation eines ungewichteten Graphen durch Adjazenzlisten, hier: Adjazenzmengen. Jedem Knoten wird in einem dictionary die Menge der Zielknoten der von ihm ausgehenden Kanten zugeordnet.




```
G = {'a':set('bc'),
     'b':set('d'),
     'c':set('bd'),
     'd':set('b')}
```

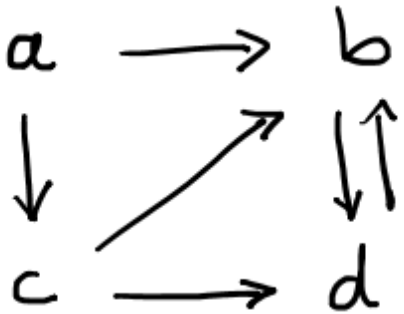
Implementation eines ungewichteten Graphen durch Adjazenzlisten, hier: Adjazenzmengen. Jedem Knoten wird in einem dictionary die Menge der Zielknoten der von ihm ausgehenden Kanten zugeordnet.




```
G = {'a':set('bc'),
     'b':set('d'),
     'c':set('bd'),
     'd':set('b')}
```

```
# gibt es Kante von a nach b
```

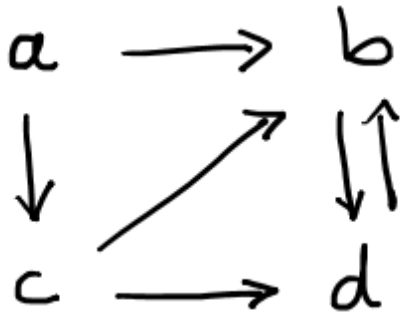
Implementation eines ungewichteten Graphen durch Adjazenzlisten, hier: Adjazenzmengen. Jedem Knoten wird in einem dictionary die Menge der Zielknoten der von ihm ausgehenden Kanten zugeordnet.




```
G = {'a':set('bc'),
     'b':set('d'),
     'c':set('bd'),
     'd':set('b')}
```

```
# gibt es Kante von a nach b
>>> 'b' in G['a']
True
# alle Nachbarn von a:
```

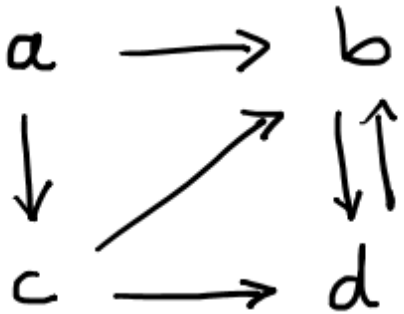
Implementation eines ungewichteten Graphen durch Adjazenzlisten, hier: Adjazenzmengen. Jedem Knoten wird in einem dictionary die Menge der Zielknoten der von ihm ausgehenden Kanten zugeordnet.




```
G = {'a':set('bc'),
     'b':set('d'),
     'c':set('bd'),
     'd':set('b')}
```

```
# gibt es Kante von a nach b
>>> 'b' in G['a']
True
# alle Nachbarn von a:
>>> for v in G['a']:
        print(v)
c
b
# alle Knoten von G durchlaufen:
```

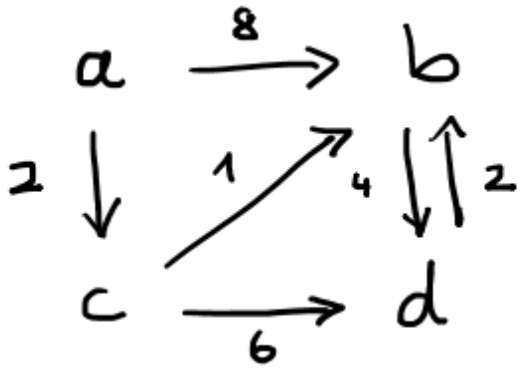
Implementation eines ungewichteten Graphen durch Adjazenzlisten, hier: Adjazenzmengen. Jedem Knoten wird in einem dictionary die Menge der Zielknoten der von ihm ausgehenden Kanten zugeordnet.



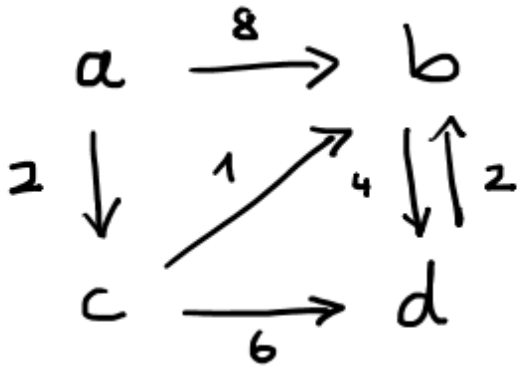

```
G = {'a':set('bc'),
     'b':set('d'),
     'c':set('bd'),
     'd':set('b')}
```

```
# gibt es Kante von a nach b
>>> 'b' in G['a']
True
# alle Nachbarn von a:
>>> for v in G['a']:
        print(v)
c
b
# alle Knoten von G durchlaufen:
>>> for v in G:
        ....
```

Implementation eines gewichteten Graphen durch ein dictionary von dictionaries. Jedem Knoten wird ein dictionary zugeordnet, die jeder ausgehenden Kante ihre Kosten zuordnet.

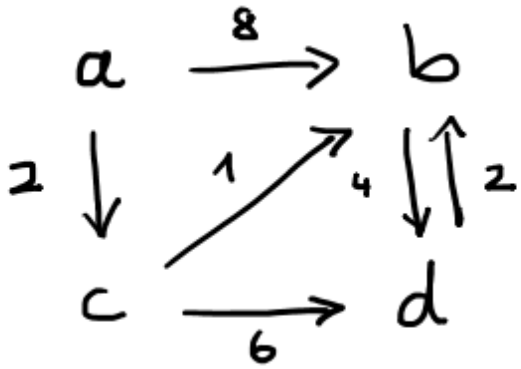


Implementation eines gewichteten Graphen durch ein dictionary von dictionaries. Jedem Knoten wird ein dictionary zugeordnet, die jeder ausgehenden Kante ihre Kosten zuordnet.




```
G = {'a': {'b':8, 'c':2},
     'b': {'d':4},
     'c': {'b':1, 'd':6},
     'd': {'b':2}}
```

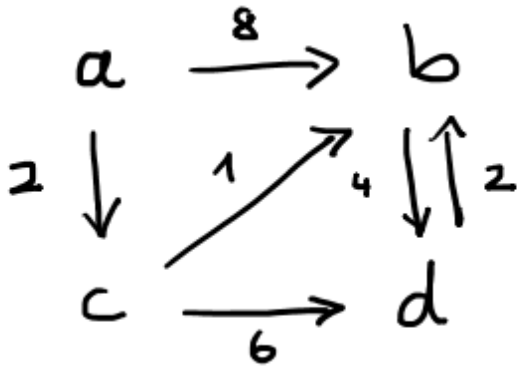
Implementation eines gewichteten Graphen durch ein dictionary von dictionaries. Jedem Knoten wird ein dictionary zugeordnet, die jeder ausgehenden Kante ihre Kosten zuordnet.




```
G = {'a': {'b':8, 'c':2},
     'b': {'d':4},
     'c': {'b':1, 'd':6},
     'd': {'b':2}}
```

```
# gibt es Kante von a nach b
```

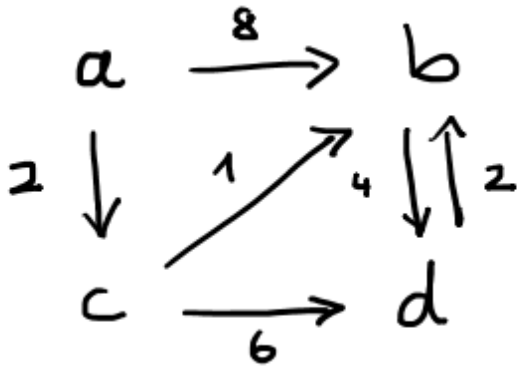
Implementation eines gewichteten Graphen durch ein dictionary von dictionaries. Jedem Knoten wird ein dictionary zugeordnet, die jeder ausgehenden Kante ihre Kosten zuordnet.




```
G = {'a': {'b':8, 'c':2},
     'b': {'d':4},
     'c': {'b':1, 'd':6},
     'd': {'b':2}}
```

```
 # gibt es Kante von a nach b
>>> 'b' in G['a']
# die Kosten der Kante von a nach b:
```

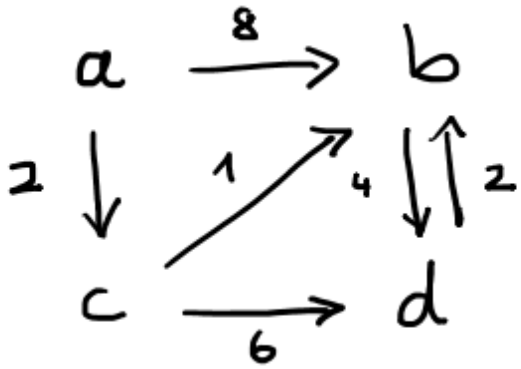
Implementation eines gewichteten Graphen durch ein dictionary von dictionaries. Jedem Knoten wird ein dictionary zugeordnet, die jeder ausgehenden Kante ihre Kosten zuordnet.




```
G = {'a': {'b':8, 'c':2},
     'b': {'d':4},
     'c': {'b':1, 'd':6},
     'd': {'b':2}}
```

```
 # gibt es Kante von a nach b
>>> 'b' in G['a']
# die Kosten der Kante von a nach b:
>>> G['a']['b']
# alle Nachbarn von a
```

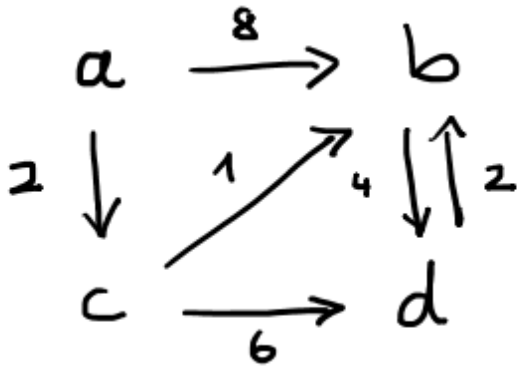
Implementation eines gewichteten Graphen durch ein dictionary von dictionaries. Jedem Knoten wird ein dictionary zugeordnet, die jeder ausgehenden Kante ihre Kosten zuordnet.




```
G = {'a': {'b':8, 'c':2},
     'b': {'d':4},
     'c': {'b':1, 'd':6},
     'd': {'b':2}}
```

```
 # gibt es Kante von a nach b
>>> 'b' in G['a']
# die Kosten der Kante von a nach b:
>>> G['a']['b']
# alle Nachbarn von a
>>> for v in G['a']:    ....
# alle Knoten von G durchlaufen:
```

Implementation eines gewichteten Graphen durch ein dictionary von dictionaries. Jedem Knoten wird ein dictionary zugeordnet, die jeder ausgehenden Kante ihre Kosten zuordnet.

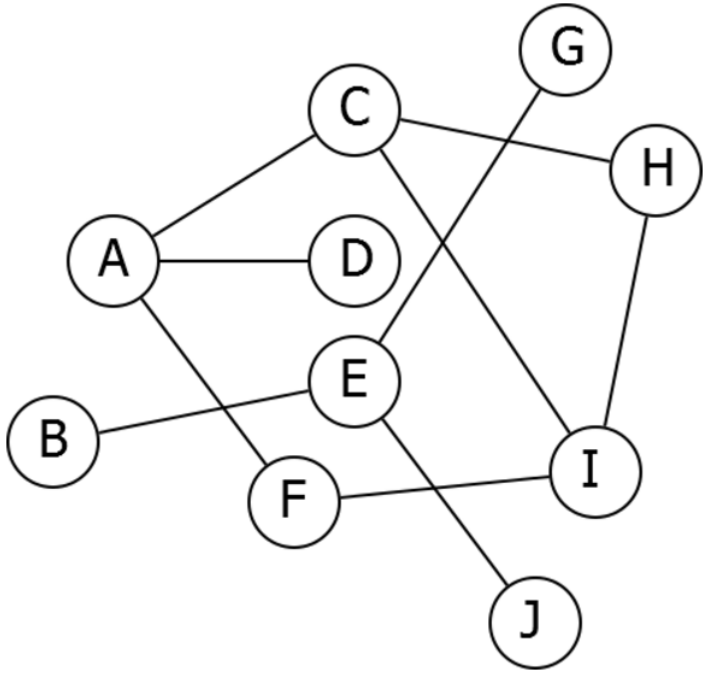



```
G = {'a': {'b':8, 'c':2},
     'b': {'d':4},
     'c': {'b':1, 'd':6},
     'd': {'b':2}}
```

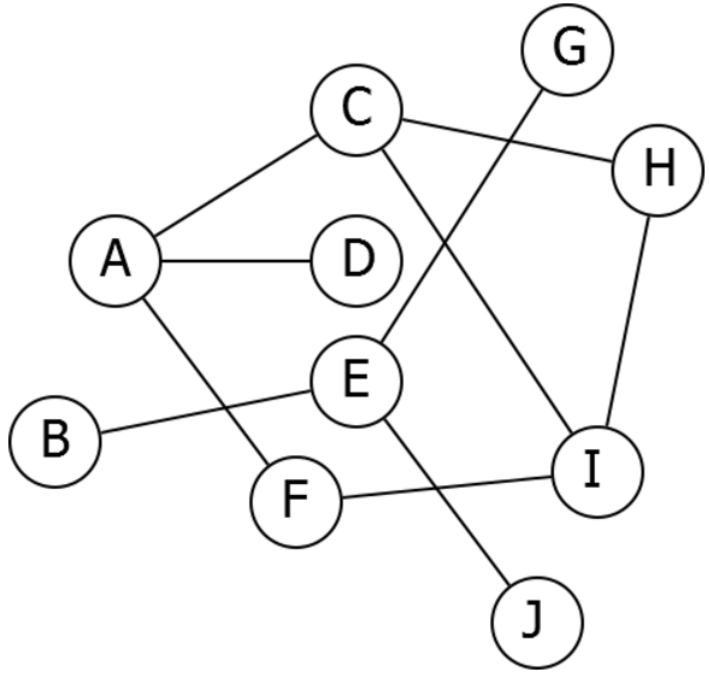
```
 # gibt es Kante von a nach b
>>> 'b' in G['a']
# die Kosten der Kante von a nach b:
>>> G['a']['b']
# alle Nachbarn von a
>>> for v in G['a']:    ....
# alle Knoten von G durchlaufen:
>>> for v in G:    ....
```

Erreichbarkeit

Gegeben ein Graph G und ein Startknoten s. Welche Knoten sind von s erreichbar?



Erreichbarkeit
Gegeben ein Graph G und ein Startknoten s. Welche Knoten sind von s erreichbar?




```
Setze alle Knoten auf nicht besucht
explore(s)
Gib alle Knoten aus, die besucht wurden

def explore(v):
    merke v als besucht
    für alle Nachbarn w von v:
        wenn w nicht besucht:
            explore(w)
```

```
visited = {v : False for v in G}
def explore(v):
    visited[v] = True
    for w in G[v]:
        if not visited[w]:
            explore(w)

explore(s)
result = [v for v in G if visited[v]]
print(*result)
```

```
visited = {v : False for v in G}
def explore(v):
    visited[v] = True
    for w in G[v]:
        if not visited[w]:
            explore(w)

explore(s)
result = [v for v in G if visited[v]]
print(*result)
```

Die Funktion explore realisiert eine rekursive Tiefensuche (dfs, depth first search).

Laufzeit:

```
visited = {v : False for v in G}
def explore(v):
    visited[v] = True
    for w in G[v]:
        if not visited[w]:
            explore(w)

explore(s)
result = [v for v in G if visited[v]]
print(*result)
```

Die Funktion explore realisiert eine rekursive Tiefensuche (dfs, depth first search).

Laufzeit: jeder Knoten wird höchstens einmal explored: $O(|V|)$, jeder Knoten prüft seine Nachbarn, die Zahl der Nachbarn liegt in $O(|E|)$, insgesamt also $O(|V| + |E|)$.

Es gilt: Die Knoten eines ungerichteten Graphen G können in **Zusammenhangskomponenten** (Connected Components) aufgeteilt werden, so dass v von w genau dann erreichbar ist, wenn beide in derselben Zusammenhangskomponente liegen.

Bestimmung der Zusammenhangskomponenten:

```
Setze alle Knoten auf nicht besucht
Setze die Komponentennummer aller Knoten auf 0
cc = 1  # aktuelle Komponentennummer

Für alle Knoten u in G:
    falls u noch nicht besucht:
        explore(u)
        cc += 1
Gib Resultat aus

def explore(v):
    merke v als besucht
    merke cc als Komponentennummer von v
    für alle Nachbarn w von v:
        wenn w nicht besucht:
            explore(w)
```

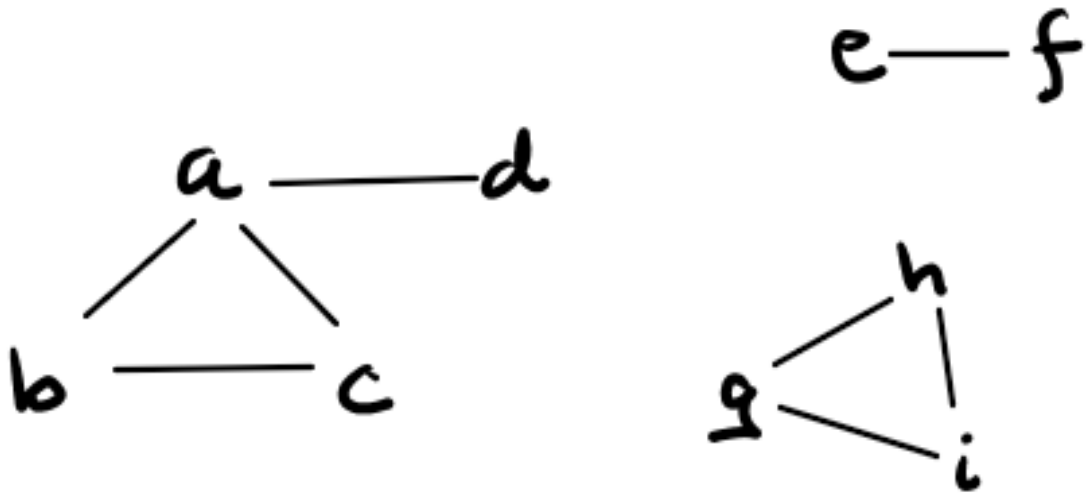
```
visited = {v : False for v in G}
ccnum = {v : 0 for v in G}      # Komponentennummer von v
cc = 1                          # aktuelle Komponentennummer

def explore(v):
    visited[v] = True
    ccnum[v] = cc
    for w in G[v]:
        if not visited[w]:
            explore(w)

for v in G:
    if not visited[v] :
        explore(v)
        cc+=1

for i in range(1,cc):
    result = [v for v in G if ccnum[v] == i]
    print(i,'-',*result)
```

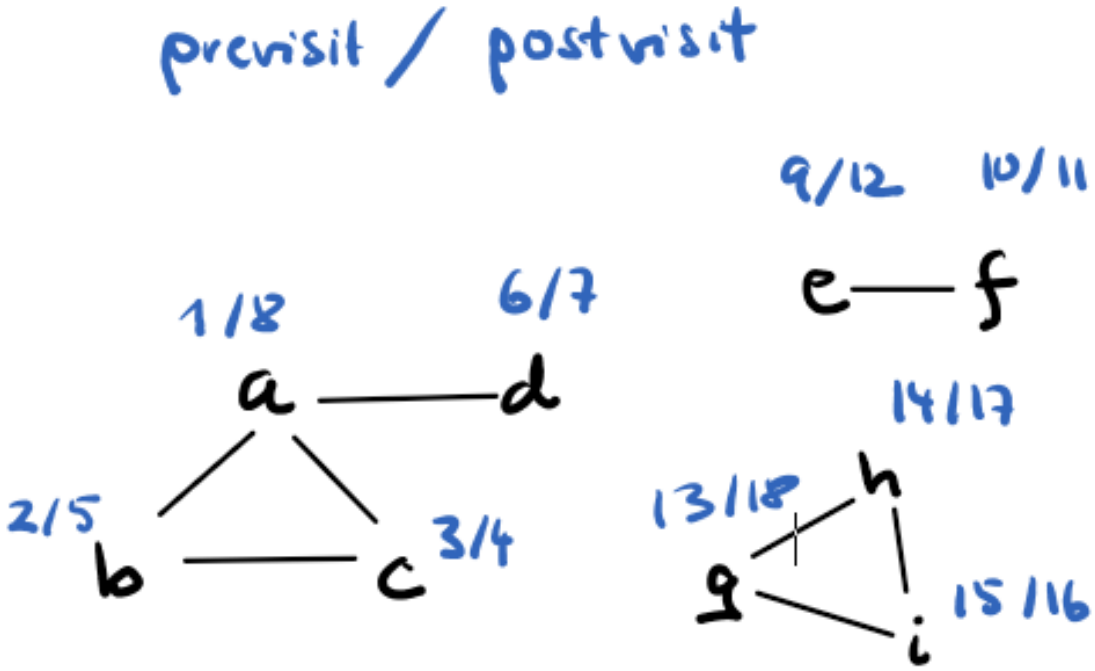
Laufzeit:

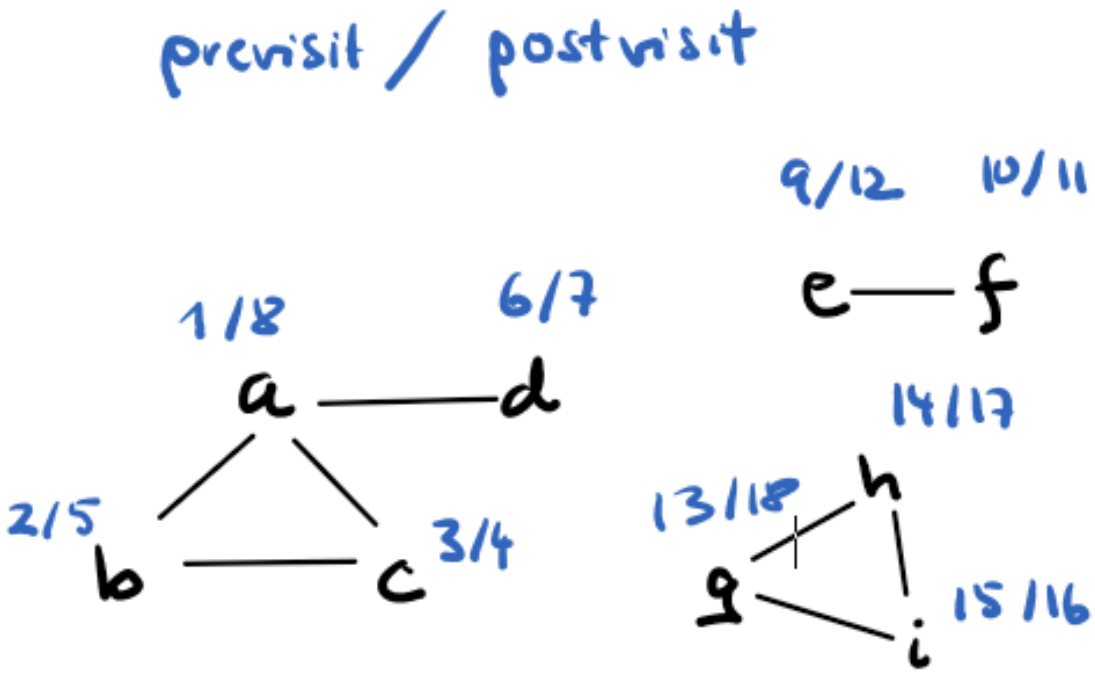
```
visited = {v : False for v in G}
ccnum = {v : 0 for v in G}      # Komponentennummer von v
cc = 1                          # aktuelle Komponentennummer

def explore(v):
    visited[v] = True
    ccnum[v] = cc
    for w in G[v]:
        if not visited[w]:
            explore(w)

for v in G:
    if not visited[v] :
        explore(v)
        cc+=1

for i in range(1,cc):
    result = [v for v in G if ccnum[v] == i]
    print(i,'-',*result)
```

Laufzeit: $O(|V| + |E|)$.

DFS endet damit, dass alle Knoten markiert sind. Das ist als Ergebnis noch nicht sonderlich interessant. Interessant wird DFS daduch, dass man bei der Ausführung noch Daten sammelt, z.B. die previsit und postvisit Nummern.




previsit / postvisit
1/18
6/7
a
d
2/5
b
c
3/4
9/12
10/11
e
f
14/17
13/18
h
g
i
15/16



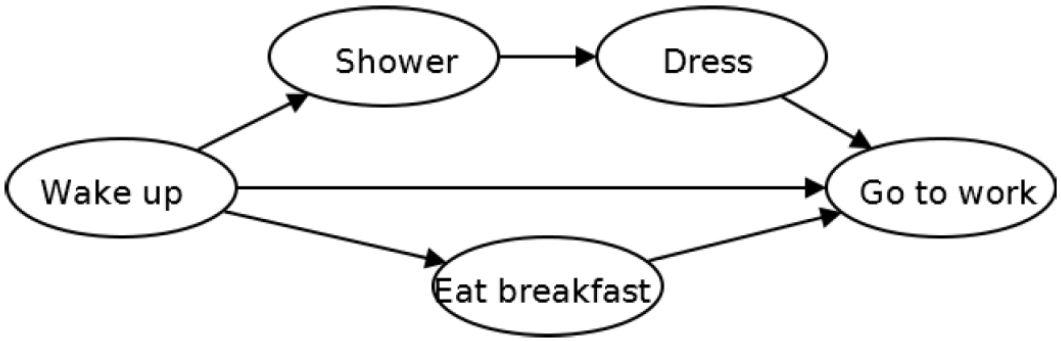
Für alle Knoten u und v gilt: die Intervalle [pre(u),post(u)] und [pre(v),post(v)] sind entweder ineinander verschachtelt oder disjunkt.

Bestimmung der previsit und postvisit-Nummern der Tiefensuche

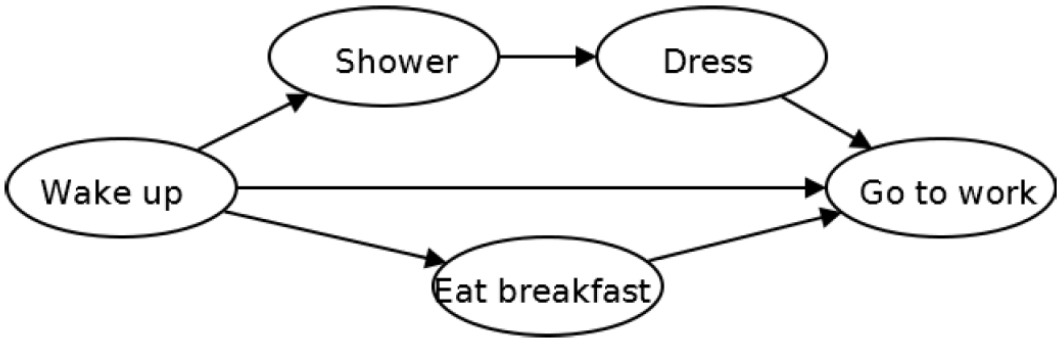
```
Setze alle Knoten auf nicht besucht
Setze previsit und postvisit Nummer aller Knoten auf 0
counter = 1  # Zähler für die visit-Nummer

Für alle Knoten u in G:
    falls u noch nicht besucht:
        explore(u)
Gib Resultat aus

def explore(v):
    merke v als besucht
    setze previsit-Nummer von v auf counter
    erhöhe counter um 1
    für alle Nachbarn w von v:
        wenn w nicht besucht:
            explore(w)
    setze postvisit-Nummer von v auf counter
    erhöhe counter um 1
```

```
visited = {v : False for v in G}
previsit = {v : 0 for v in G}
postvisit = {v : 0 for v in G}
counter = 1

def explore(v):
    global counter
    visited[v] = True
    previsit[v] = counter
    counter += 1
    for w in G[v]:
        if not visited[w]:
            explore(w)
    postvisit[v] = counter
    counter += 1

for v in G:
    if not visited[v] :
        explore(v)

for v in G:
    print("{:2} {:2} {:2}".format(v,previsit[v],postvisit[v]))
```

Topologische Sortierung eines gerichteten Graphen: wir wollen eine Nummerierung der Knoten finden, die mit den Kantenrichtungen kompatibel ist.



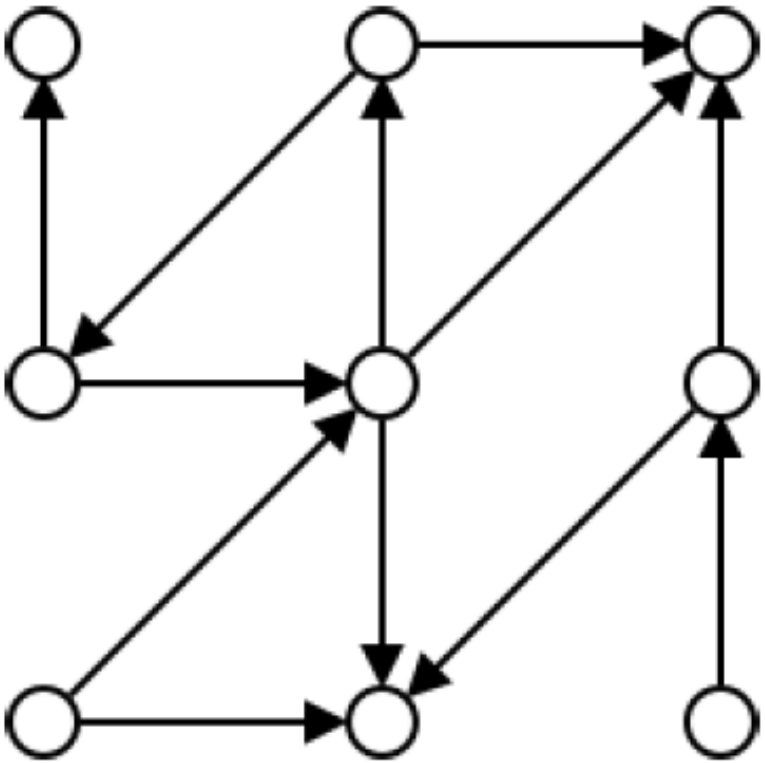
Topologische Sortierung eines gerichteten Graphen: wir wollen eine Nummerierung der Knoten finden, die mit den Kantenrichtungen kompatibel ist.



Es gilt: ein Graph kann genau dann topologisch sortiert werden, wenn er keinen Kreis enthält. Anders formuliert: Genau die gerichteten azyklischen Graphen (directed acyclic graph, DAG) können topologisch sortiert werden.

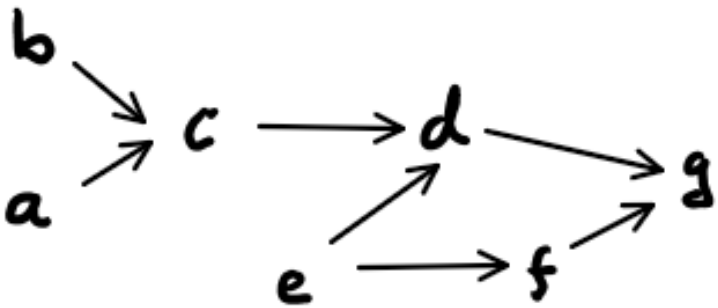
Eine Quelle (source) ist ein Knoten mit dem Eingangsgrad 0. Eine Senke (sink) ist ein Knoten mit dem Ausgangsgrad 0.

Bestimme die Senken des folgenden Graphen:



Idee für die topologische Sortierung (1. Versuch)

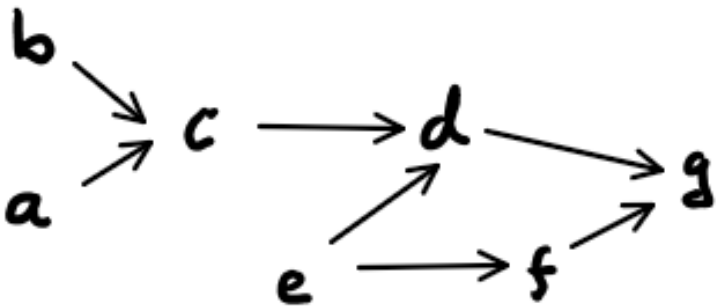
Solange der Graph nicht leer:
    Folge einem Pfad bis es nicht mehr weitergeht.
    Finde dort eine Senke
    Füge die Senke **in** die Sortierung wie **in** einen Stapel ein
    Entferne die Senke aus dem Graphen.



Laufzeit:

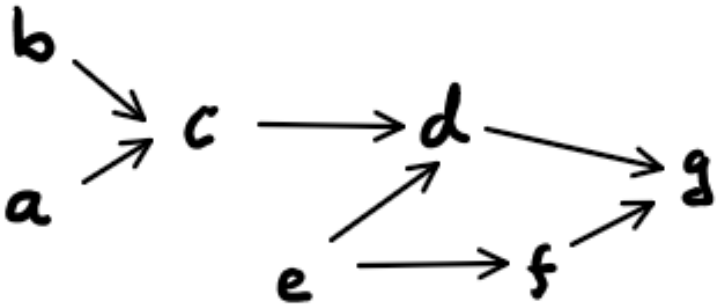
Idee für die topologische Sortierung (1. Versuch)

Solange der Graph nicht leer:
Folge einem Pfad bis es nicht mehr weitergeht.
Finde dort eine Senke
Füge die Senke **in** die Sortierung wie **in** einen Stapel ein
Entferne die Senke aus dem Graphen.

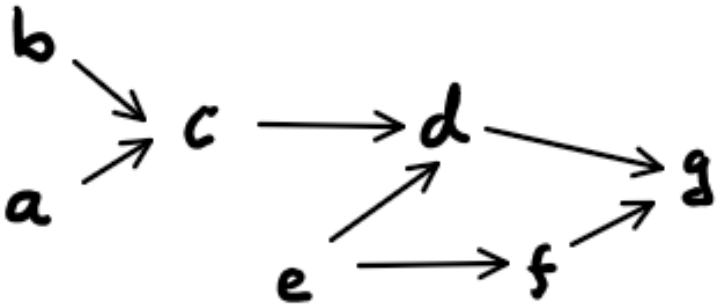


Laufzeit: $O(|V|^2)$ - nicht gut

Verbesserungsmöglichkeit: statt den Pfad immer wieder von vorne zu beginnen, wird nach der Entfernung der Senke der Weg so weit wie nötig zurückgegangen

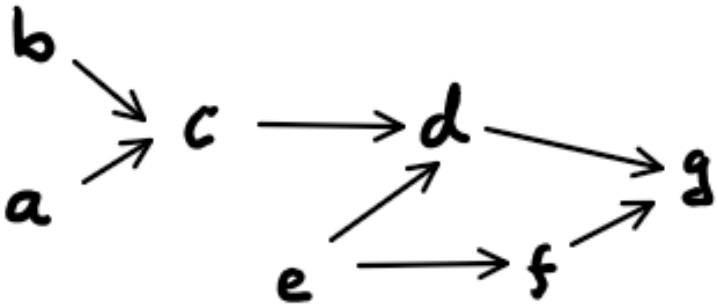


Verbesserungsmöglichkeit: statt den Pfad immer wieder von vorne zu beginnen, wird nach der Entfernung der Senke der Weg so weit wie nötig zurückgegangen



Das ist Tiefensuche! Welche Nummerierung gibt uns eine topologische Sortierung? Zur Auswahl stehen: previsit, postvisit, aufwärts, abwärts.

Verbesserungsmöglichkeit: statt den Pfad immer wieder von vorne zu beginnen, wird nach der Entfernung der Senke der Weg so weit wie nötig zurückgegangen



Das ist Tiefensuche! Welche Nummerierung gibt uns eine topologische Sortierung? Zur Auswahl stehen: previsit, postvisit, aufwärts, abwärts. Die abwärts sortierte Reihe der postvisit-Nummern ergibt eine topologische Sortierung.

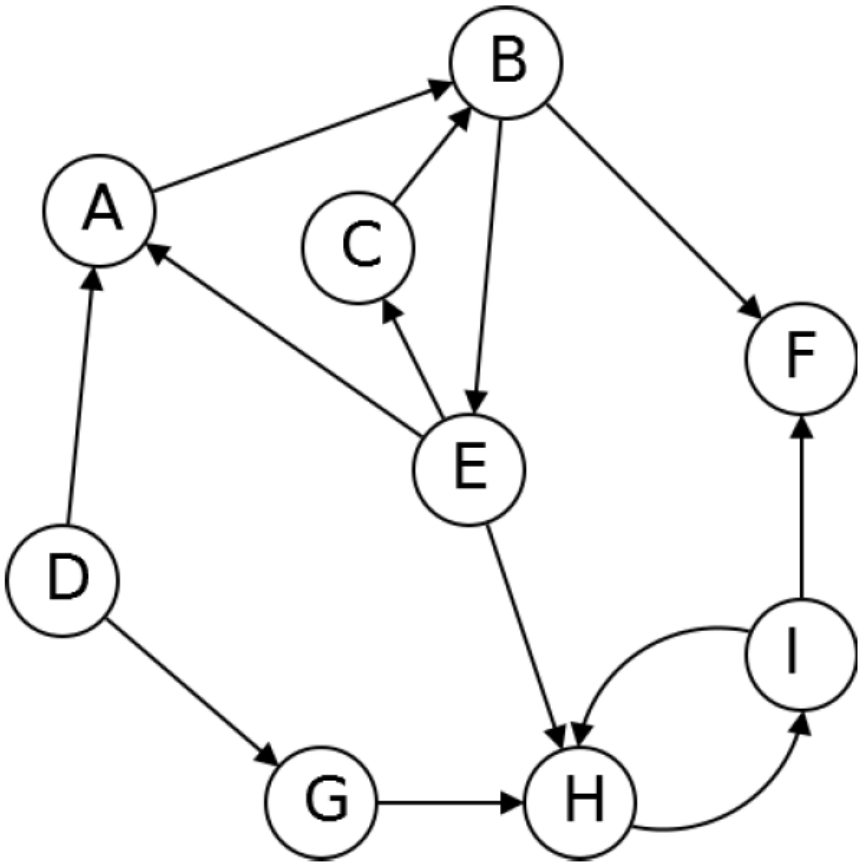
```
visited = {v : False for v in G}
postvisit = {v : 0 for v in G}
counter = 1

def explore(v):
    global counter
    visited[v] = True
    for w in G[v]:
        if not visited[w]:
            explore(w)
    postvisit[v] = counter
    counter += 1

for v in G:
    if not visited[v] :
        explore(v)

result = sorted(G, key=lambda v: postvisit[v],reverse = True)
for i in range(len(result)):
    print(i+1,result[i])
```

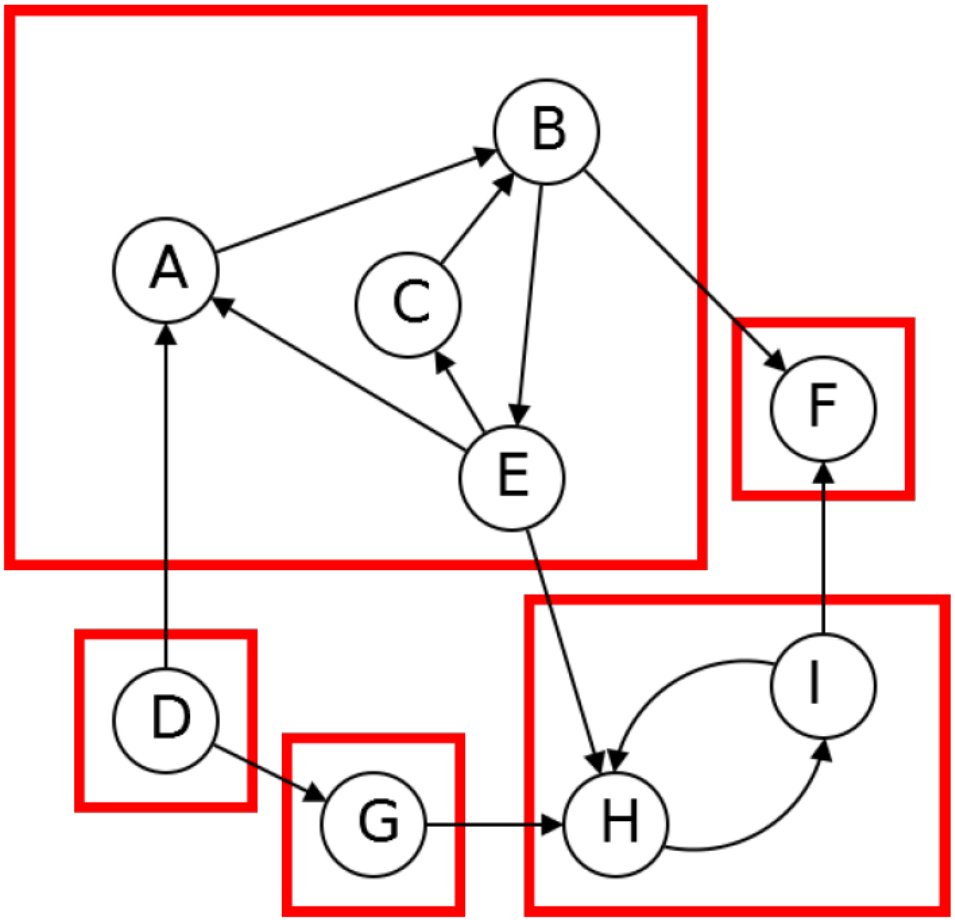
In einem gerichteten Graphen heißen zwei Knoten u und v **zusammenhängend**, wenn u von v und v von u erreichbar ist.

In einem gerichteten Graphen heißen zwei Knoten u und v **zusammenhängend**, wenn u von v und v von u erreichbar ist.

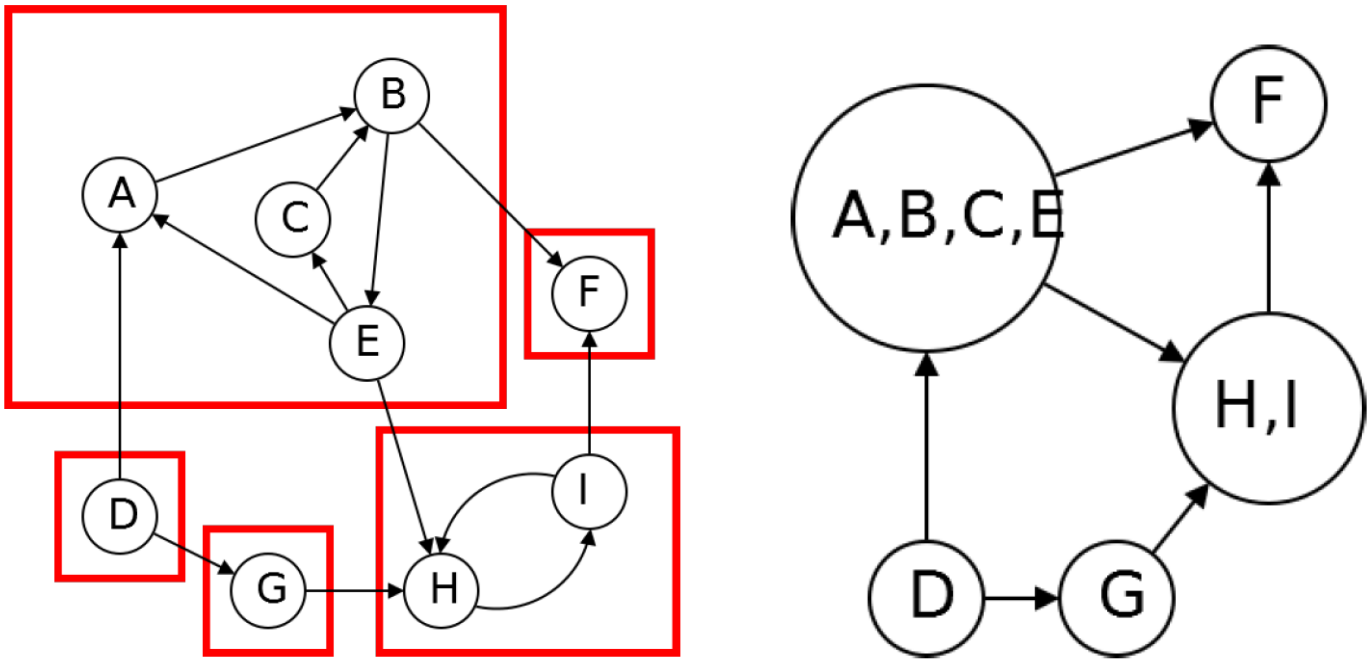
Es gilt: Ein gerichteter Graph kann partitioniert werden in **starke Zusammenhangskomponenten** (strongly connected components (SCC)), wobei zwei Knoten genau dann zusammenhängen, wenn sie in der selben Komponente sind.

Finde die starken Zusammenhangskomponenten in dem Graphen.



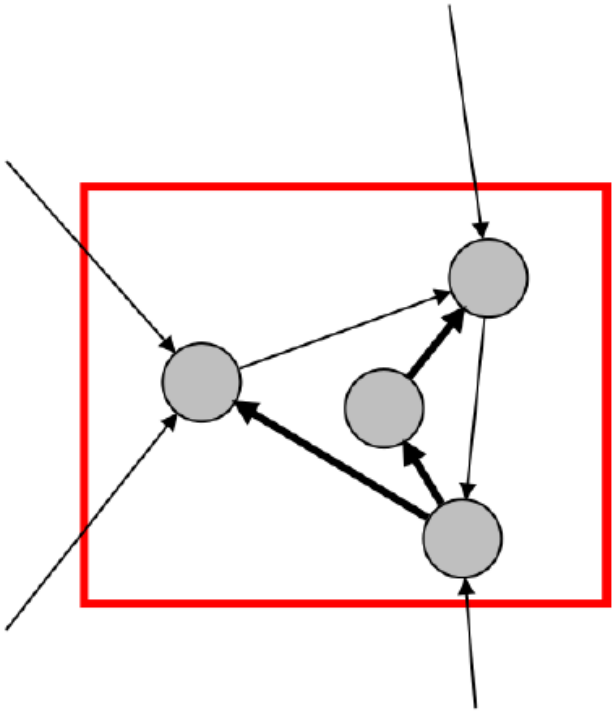

A
B
C
D
E
F
G
H
I

Der **Metagraph** zeigt, wie die starken Zusammenhangskomponenten untereinander verbunden sind.



Der Metagraph ist immer ein DAG.

Wenn v in einer SCC-Senke liegt, dann findet explore(v) genau die Elemente des SCC.



Es gilt: Wenn C und C' zwei starke Zusammenhangskomponenten sind und es eine Kante von einem Knoten aus C nach einem Knoten aus C' gibt, dann ist die größte postvisit-Nummer in C größer als die größte postvisit-Nummer in C'.

Es gilt: Wenn C und C' zwei starke Zusammenhangskomponenten sind und es eine Kante von einem Knoten aus C nach einem Knoten aus C' gibt, dann ist die größte postvisit-Nummer in C größer als die größte postvisit-Nummer in C'.

Folgerung: der Knoten mit der größten postvisit-Nummer ist in einer SCC-Quelle. Wir suchen aber eine SCC-Senke.

Es gilt: Wenn C und C' zwei starke Zusammenhangskomponenten sind und es eine Kante von einem Knoten aus C nach einem Knoten aus C' gibt, dann ist die größte postvisit-Nummer in C größer als die größte postvisit-Nummer in C'.

Folgerung: der Knoten mit der größten postvisit-Nummer ist in einer SCC-Quelle. Wir suchen aber eine SCC-Senke.

Dazu betrachten wir den reversen Graphen. Den reversen Graphen G^R eines gerichteten Graphen G erhält man, wenn man die Kantenrichtungen umdreht.

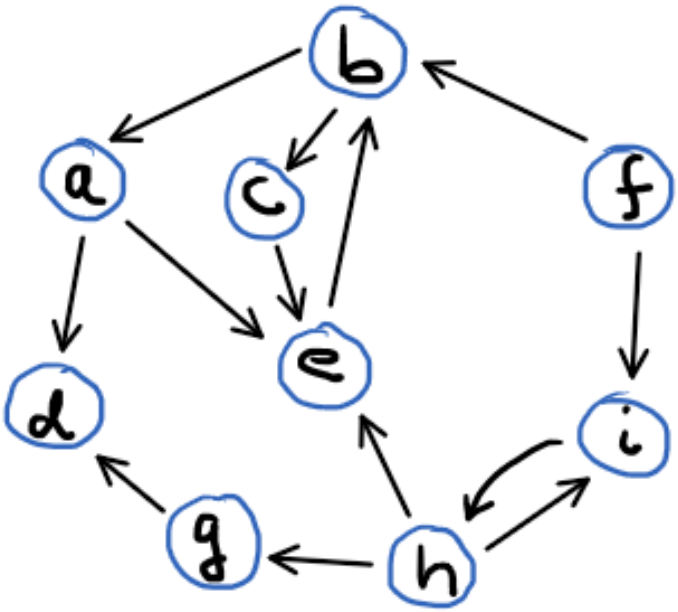
Es gilt: Wenn C und C' zwei starke Zusammenhangskomponenten sind und es eine Kante von einem Knoten aus C nach einem Knoten aus C' gibt, dann ist die größte postvisit-Nummer in C größer als die größte postvisit-Nummer in C'.

Folgerung: der Knoten mit der größten postvisit-Nummer ist in einer SCC-Quelle. Wir suchen aber eine SCC-Senke.

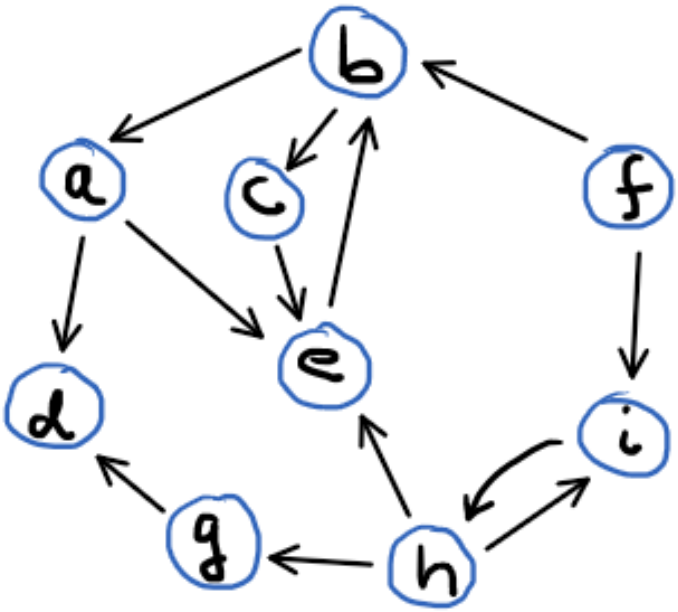
Dazu betrachten wir den reversen Graphen. Den reversen Graphen G^R eines gerichteten Graphen G erhält man, wenn man die Kantenrichtungen umdreht.

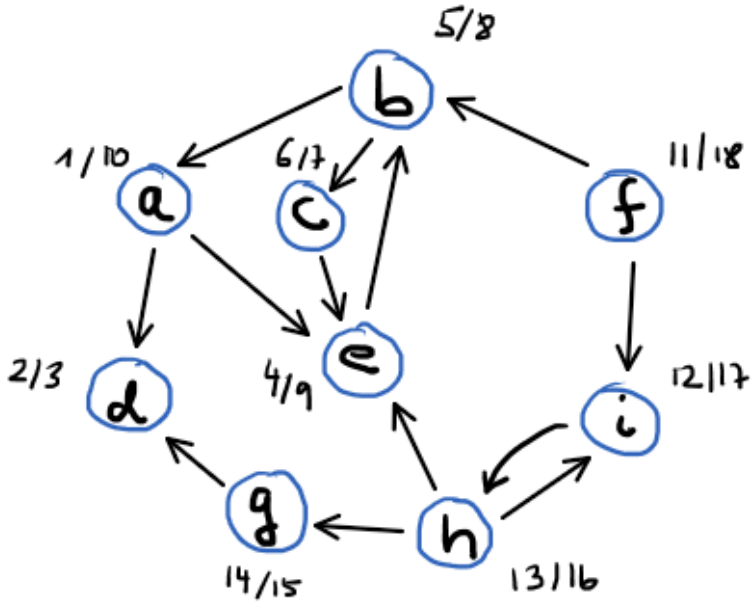
G und G^R haben dieselben SCCs.
SCC-Quellen in G^R sind SCC-Senken in G.

Bestimmung der postvisit-Nummern des reversen Graphen

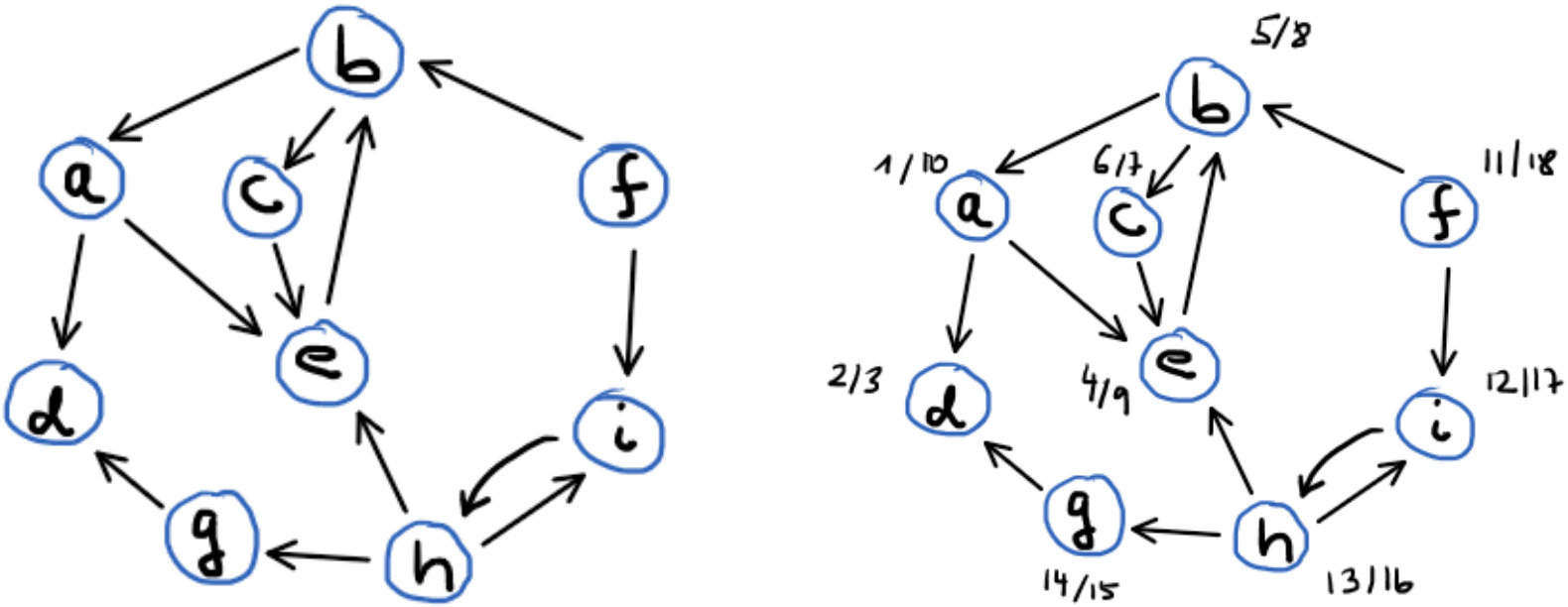


Bestimmung der postvisit-Nummern des reversen Graphen



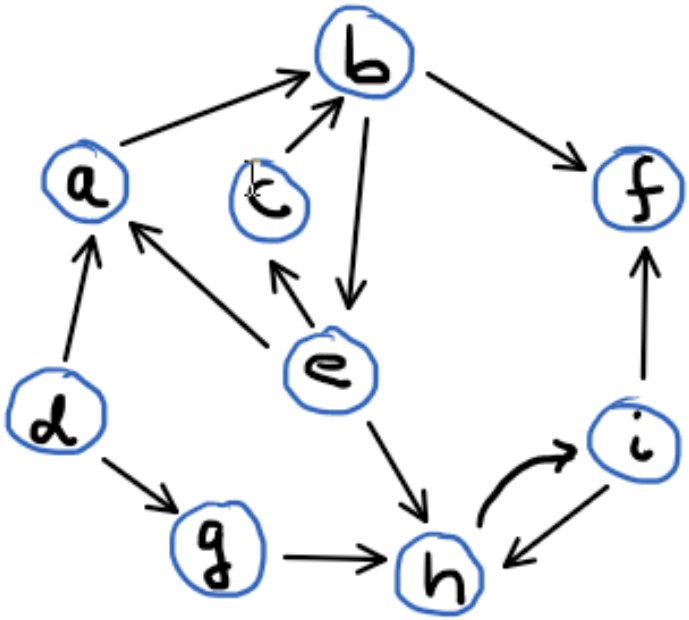


Bestimmung der postvisit-Nummern des reversen Graphen

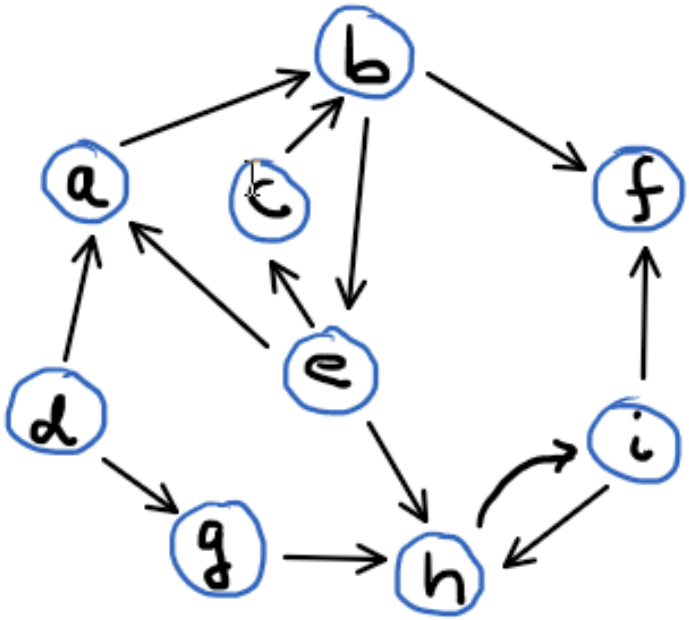


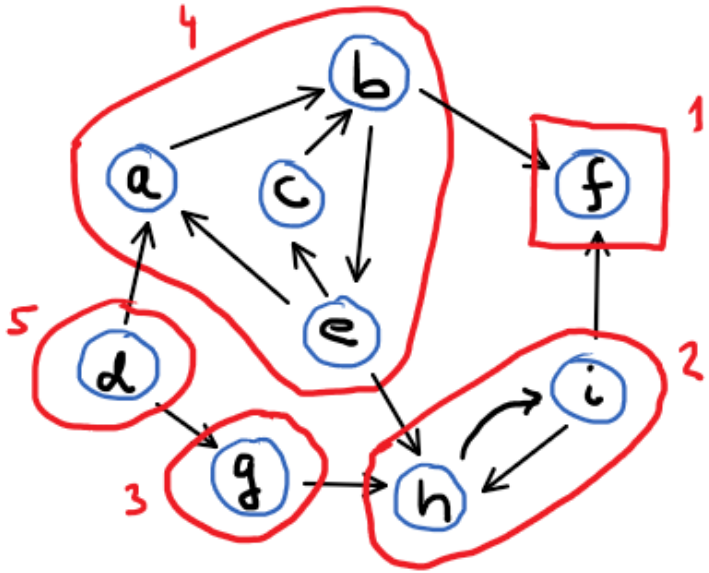
Umgekehrte Reihenfolge der postvisit-Nummern: f i h g a e b c d

Bestimmung der SCCs mit Tiefensuche in der Reihenfolge f i h g a e b c d.

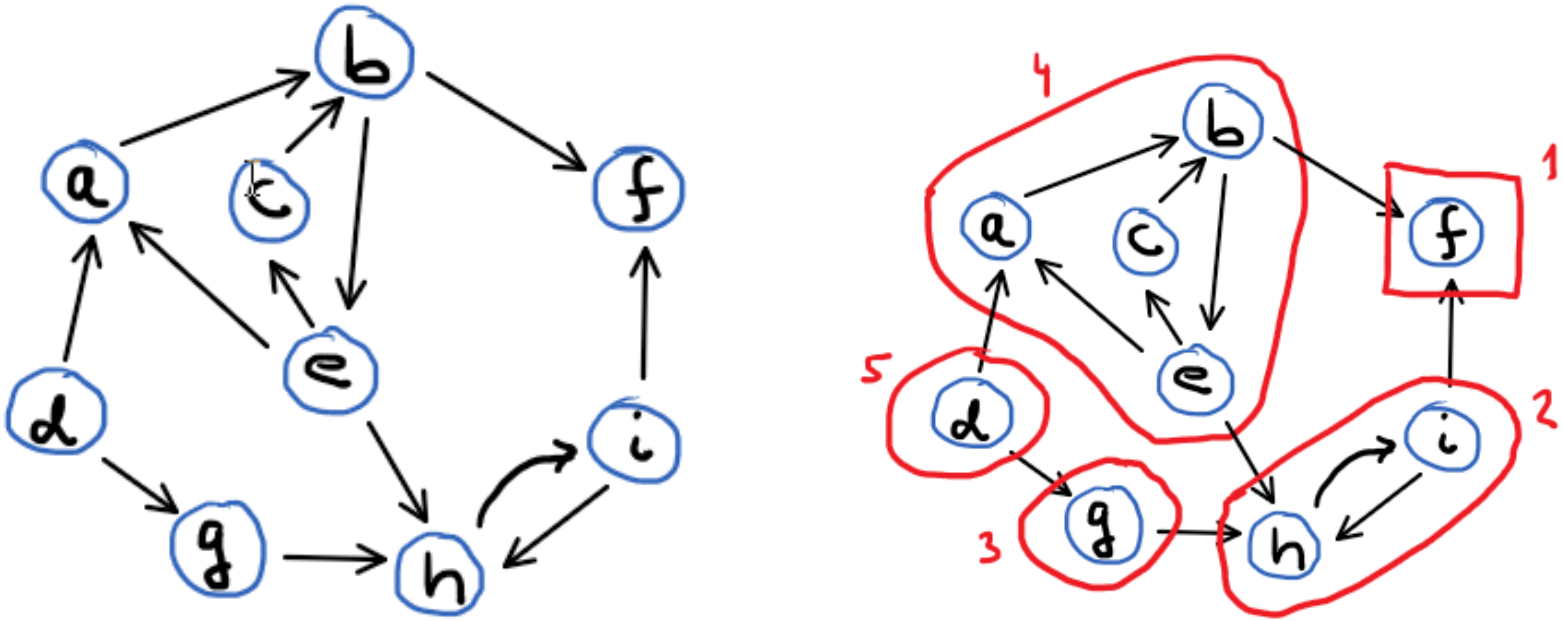


Bestimmung der SCCs mit Tiefensuche in der Reihenfolge f i h g a e b c d.





Bestimmung der SCCs mit Tiefensuche in der Reihenfolge f i h g a e b c d.



Laufzeit: im wesentlichen Tiefensuche auf G^R und dann auf G, also $O(|V| + |E|)$.

Bestimmung der starken Zusammenhangskomponenten in einem gerichteten Graphen G

```
Führe Tiefensuche auf G^R durch und merke postvisit-Nummern.
Für alle Knoten v in G in umgekehrter postvisit-Nummerierung:
    Falls v noch nicht besucht:
        explore(v) und markiere alle
            besuchten Knoten u als neuen SCC.
```