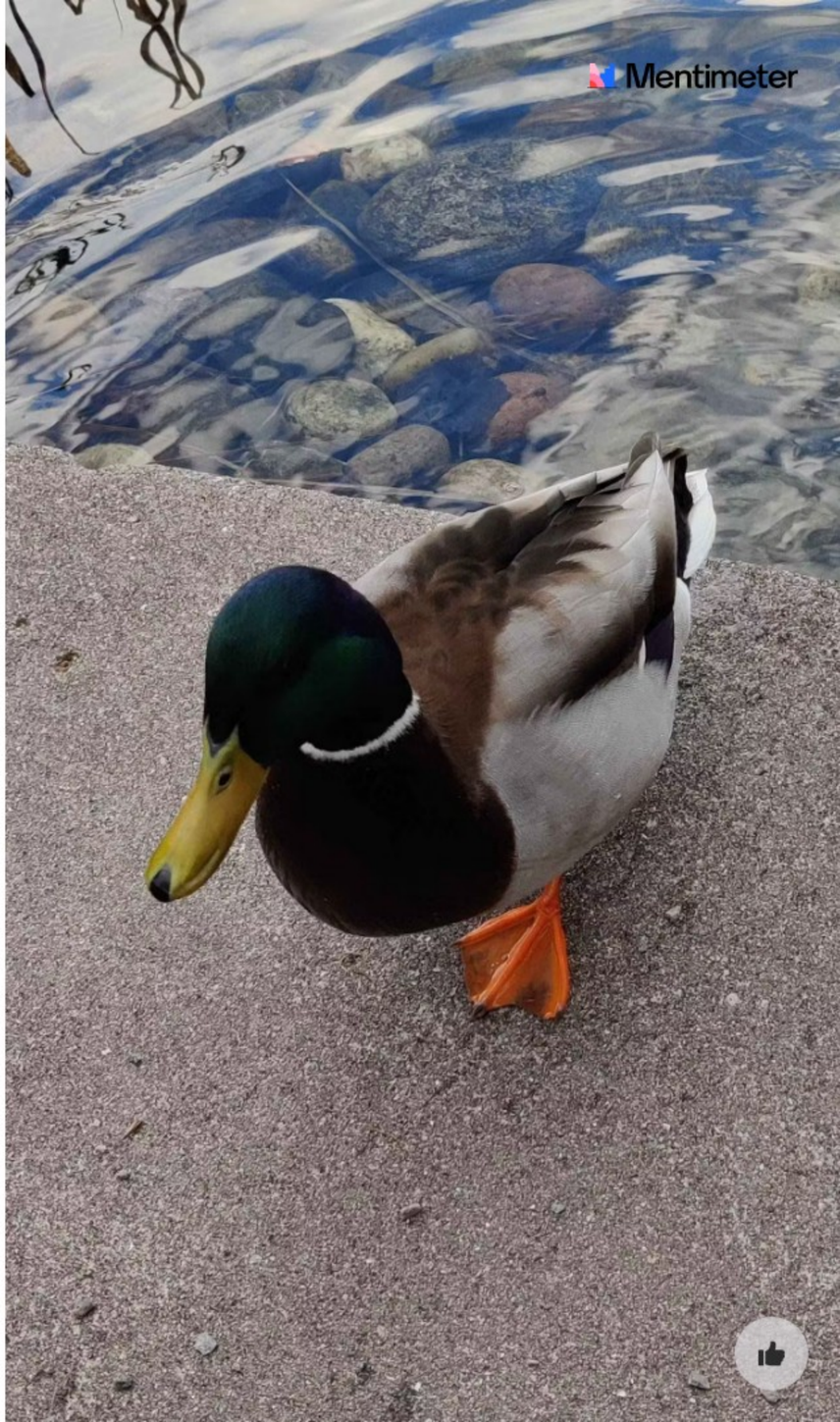
Velkommen til IN2010 gruppe 6



Mentimeter

Siste innspurt på grafer...

Planen for i dag

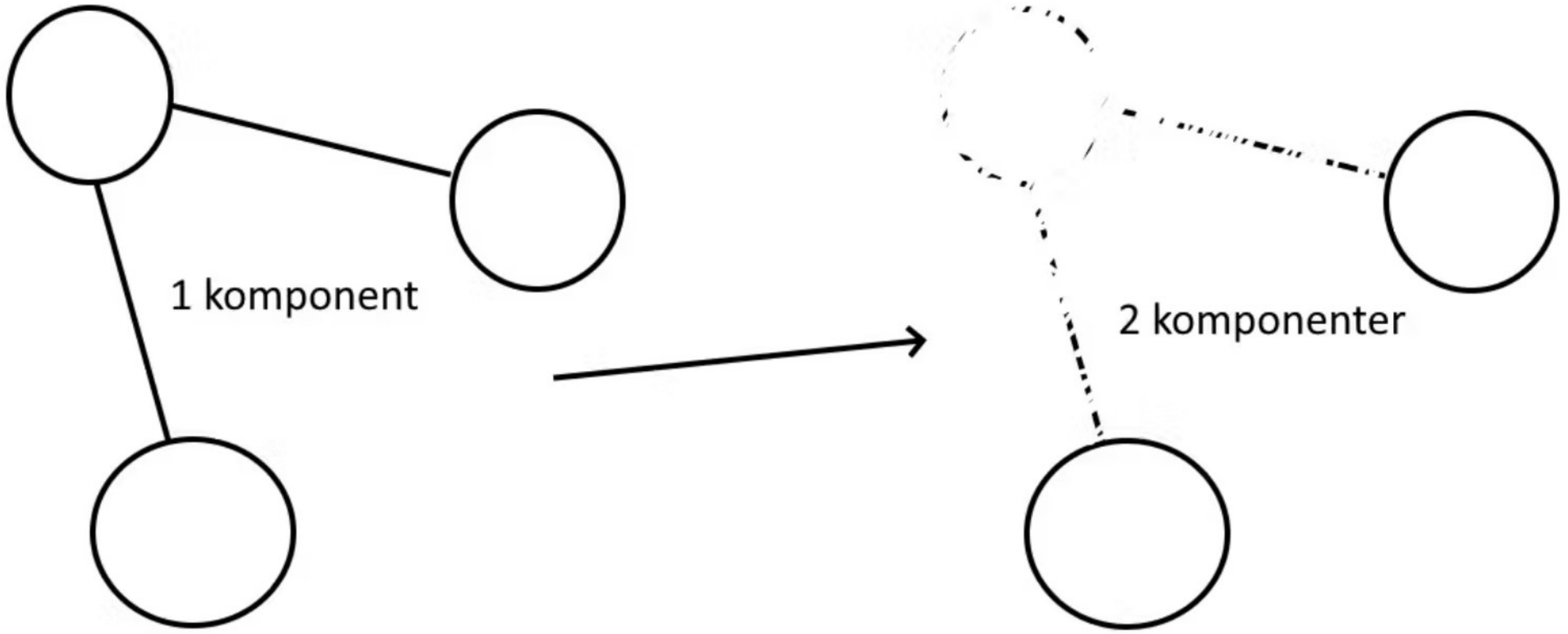
- 2-sammenhengende naiv
- 2-sammenhengende vanskelig
- Sterk sammenhengende grafer
- Evt noen oppgaver / Lab

N-Sammenhengende Grafer

- Fra tidligere vet vi at en graf består av en mengde V av noder v og en mengde E av kanter (v, u)

- N-Sammenhengende grafer sier noe om hvor sammenhengende den er

- Mer presist -> En graf er n-sammenhengende dersom man kan fjerne n-1 vilkårlige noder
  - Enda mer presist på neste slide



Hvordan å finne ut "hvor" sammenhengende en graf er

- Vi kan starte med å sjekke om en graf er 2-sammenhengende
  - Det vil si at vi trenger å bare fjerne en node, men en vilkårlig node
  - Vilkårlig vil si at vi kan fjerne en hvilken som helst node, og grafen vil fortsatt være sammenhengende

- Dersom vi vil sjekke for noe vilkårlig må vi sjekke alle tilfeller
- Det betyr at vi må gjøre et søk |V| ganger for å se om det ble laget flere komponenter etter vi fjernet noden

- Hvis vi sjekker 3-sammenhengenhet så må vi sjekke alle kombinasjoner av noder, så fjerne A B, neste blir A C ... A D, B C så B D, osv

```python
def removenode(G, v):
    V, E, w = G

    newV = V.copy()
    newE = E.copy()

    newV.discard(v)
    del newE[v]

    for u in newV:
        neighbours = newE[u].copy()
        neighbours.discard(v)
        newE[u] = neighbours

    return newV, newE, w
```

```python
def isbiconnected_naive(G):
    V, E, w = G

    for v in V:
        newV, newE, newW = newG = removenode(G, v)
        nodelist = depth_first_search(newG, next(iter(newV)))

        if set(nodelist) != newV:
            return False
    return True
```

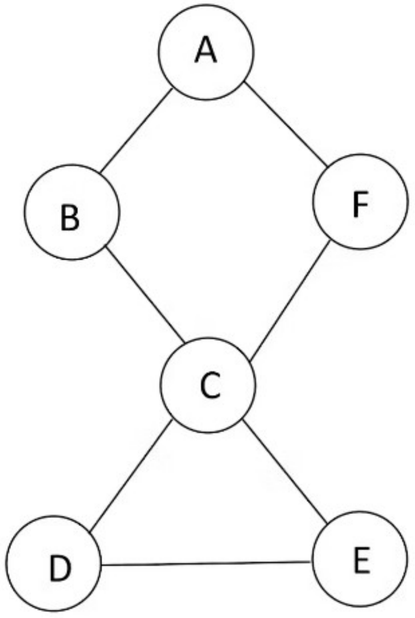
2-Sammenhengende vanskelig

- Forrige løsning er for naiv
- Vi kan heller vri litt på perspektivet for å få det raskere

  - Vi gir hver node en **dybde** verdi og en **low** verdi

  - **Dybde** er hvor langt inn i søket vi er
  - **Low** er hva den minste **dybde** verdien noden kan nå ved å gå bakover med maks en tilbakekant

- **Dybde** verdien vil ikke endre seg, men **low** verdien vil endre seg under algoritmen
- Algoritmen har et dybde først søk struktur



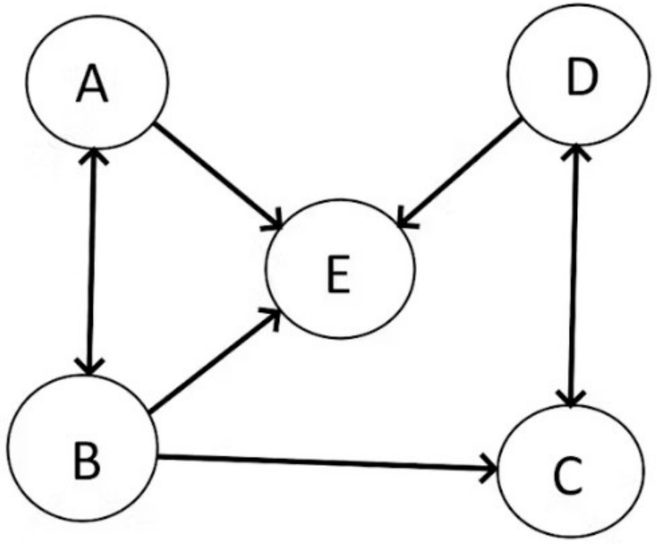

```python
def seperationnodes(G):
    V, E = G
    s = next(iter(V))
    depth = {s: 0}
    low = {s: 0}
    parents = {s: None}
    seps = set()

    for v in E[s]:
        if v not in depth:
            seperationnodes_rec(E, v, 1, depth, low, parents, seps)

    if len([u for u in E[s] if depth[u] == 1]) > 1:
        seps.add(s)

    return seps
```

```python
def seperationnodes_rec(E, v, d, depth, low, parents, seps):
    depth[v] = low[v] = d
    for u in E[v]:
        if u in parents and parents[u] == v:
            continue
        if u in depth:
            low[v] = min(low[v], depth[u])
            continue

        parents[u] = v
        seperationnodes_rec(E, u, d + 1, depth, low, parents, seps)
        low[v] = min(low[u], low[v])
        if d <= low[u]:
            seps.add(v)
```

Sterkt sammenhengende komponenter

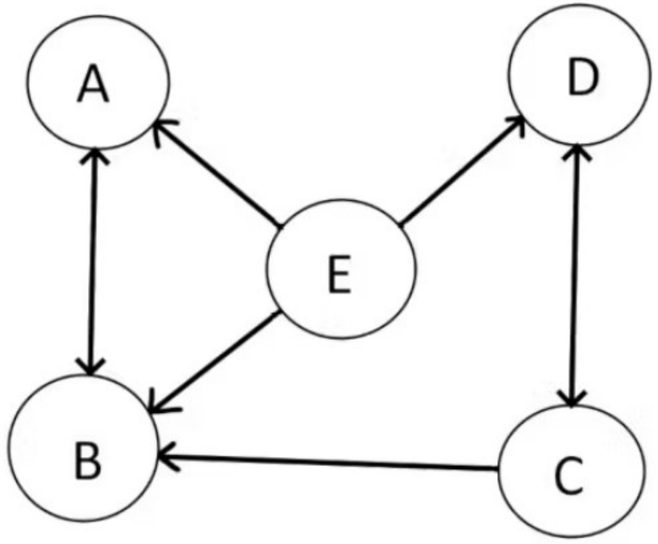
- En komponent i en urettet graf kan bli funnet ved bruk av et BFSøk på hver enkel node og en mengde med besøkte noder

- I rettede grafer kan man se etter sterkt sammenhengende komponenteter
- En sterk sammenhengende komponent vil si at:
  - Fra alle noder i komponenten så kan man besøke alle andre noder
- For å sjekke det så kan vi:
  - Gjøre det DFSøk og lagre resultatet
  - Reversere grafen, alle kanter peker motsatt vei
  - Gjøre et nytt DFSøk på den nye grafen basert på resultatet fra det første søket




```python
def strongly_connected_components(G):
    V, E = G

    stack = dfstopsort(G)

    Gr = reverse_graph(G)
    visited = set()
    components = []

    while stack:
        v = stack.pop()
        if v not in visited:
            component = []
            dfsvisit(Gr, v, visited, component)
            components.append(component)

    return components
```

```python
def dfsvisit(G, v, visited, stack):
    _, E = G
    visited.add(v)

    for u in E[v]:
        if u not in visited:
            dfsvisit(G, u, visited, stack)
    stack.append(v)

def dfstopsort(G):
    V, E = G
    visited = set()
    stack = []
    for v in V:
        if v not in visited:
            dfsvisit(G, v, visited, stack)
    return stack
```




```python
def strongly_connected_components(G):
    V, E = G

    stack = dfstopsort(G)

    Gr = reverse_graph(G)
    visited = set()
    components = []

    while stack:
        v = stack.pop()
        if v not in visited:
            component = []
            dfsvisit(Gr, v, visited, component)
            components.append(component)

    return components
```

```python
def dfsvisit(G, v, visited, stack):
    _, E = G
    visited.add(v)

    for u in E[v]:
        if u not in visited:
            dfsvisit(G, u, visited, stack)
    stack.append(v)

def dfstopsort(G):
    V, E = G
    visited = set()
    stack = []
    for v in V:
        if v not in visited:
            dfsvisit(G, v, visited, stack)
    return stack
```

Det var egentlig det...
Forslag til oppgaver:

- Lag en egen graf og implementer de nye algoritmene
- Ukes oppgaver

Eller:

- Ferdigstilling av Oblig 5
  - Veldig kult for retterne hvis dere leverer en retter-venlig oblig

mafredri@ifi.uio.no