Bikonnektivitet

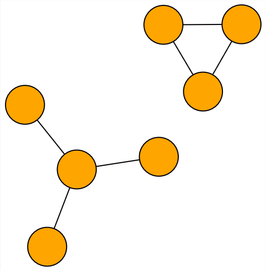
IN2010 – Algoritmer og Datastrukturer

Uke 41, 2020

Institutt for Informatikk

Sammenhengende grafer

En graf $G = (V, E)$ kalles for *sammenhengende* hvis det finnes en sti mellom hvert par av noder

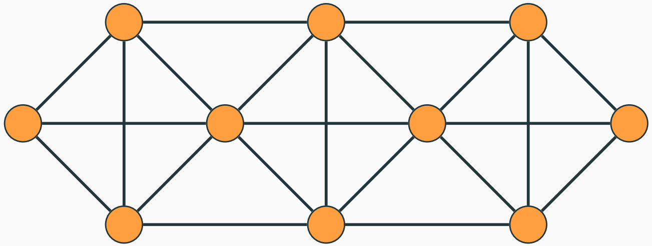


Men hvordan sjekke om en graf er sammenhengende?

- begynn i en vilkårlig node og traverser grafen (BFS/DFS). Hvis det finnes ubesøkte noder, er grafen ikke sammenhengende. $O(|V| + |E|)$.

k-sammenhengende grafer

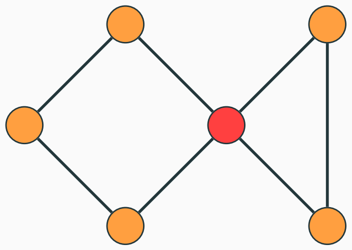
En sammenhengende graf G er k-sammenhengende, eller k-node-sammenhengende, hvis G forblir sammenhengende om færre enn k noder blir fjernet.



Grafen over er 3-sammenhengende. Vi må fjerne 3 noder for at grafen skal slutte å være sammenhengende.

2-sammenhengende grafer (eng. biconnected graphs)

- en graf G er 2-sammenhengende/biconnected, hvis G forblir sammenhengende ved fjerning av én vilkårlig node
- formulert på en annen måte: hvis alle par av noder u og v har to distinkte stier (deler ingen kanter eller noder) mellom dem



- Noder som ved fjerning fører til at grafen blir ikke sammenhengende heter *separasjonsnoder*.
- tilsvarer kritiske punkter i nettverk

2-sammenhengende grafer

Hvordan sjekke om en graf inneholder separasjonsnoder?

- Fjern en node, sjekk om grafen er sammenhengende. Gjenta dette for alle noder

Algorithm: Naiv Biconnectivity

Output: Alle separasjonsnoder i inputgrafen G

```
1 Procedure Biconnectivity(G)
2     initialize sep_vertices empty list
3     for each vertex v in G do
4         construct graph G′ as G without v and its edges
5         if connected(G′) = false then sep_vertices.add(v)
6     return sep_vertices
```

Vi så tidligere: å sjekke om en graf er sammenhengende er $O(|V| + |E|)$.

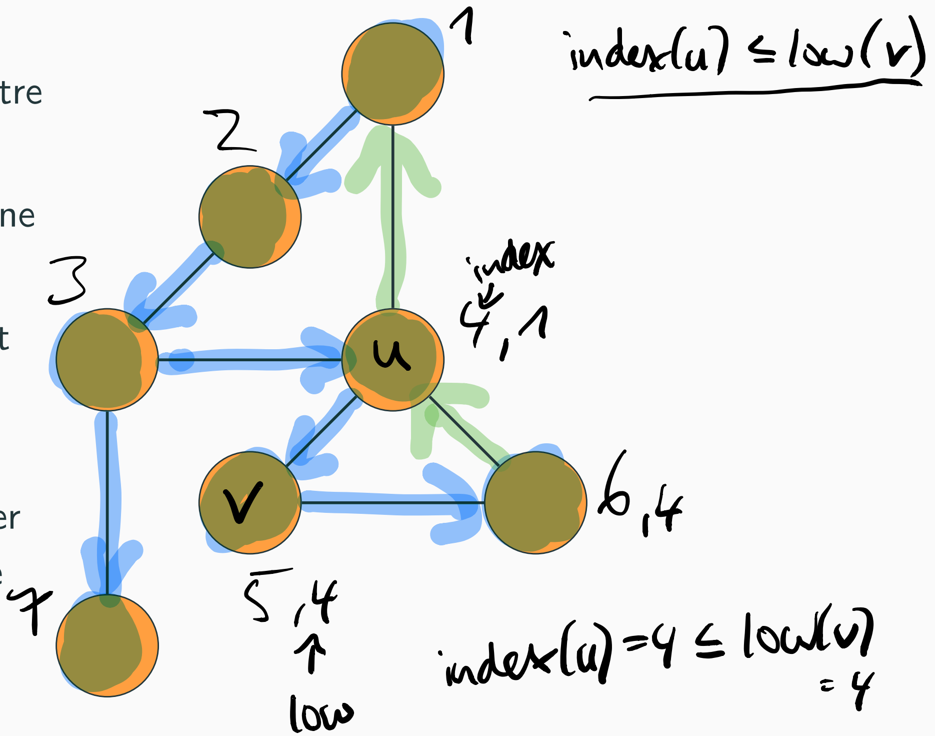
Dette må vi gjøre for hver node (linje 3), dermed i $O(|V| \cdot (|V| + |E|))$.

Dette er $O(|V|^3)$.

2-sammenhengende grafer i lineær tid

Vi bruker DFS til å lage et spenntre

- indekserer rekkefølgen nodene blir besøkt
- *Discovery edges* (blå) i treet er kanter som fører til nye noder
- *back edges* (grønn) er kanter som fører tilbake til allerede besøkte noder.



2-sammenhengende grafer i lineær tid III

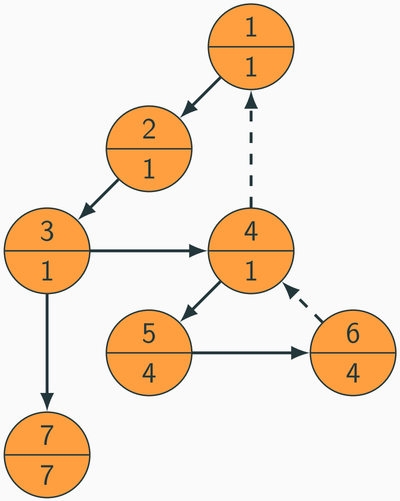
En node u er en separasjonsnode hvis:

- u er rot i T og har mer enn ett barn, eller
- u ikke er roten, og har et barn v, slik at ingen etterkommere av v (inkludert v selv) har en back-edge til en forgjenger av u.

Det andre punktet kan vi sjekke ved hjelp av *low*-nummer. *Low*-nummeret til en node er definert som indeksen til den noden med lavest indeks som kan nås ved å følge:

1. 0 eller flere kanter i T (discovery edges) etterfulgt av
2. 0 eller 1 back-edge.

Nå kan vi uttrykke punkt to som: u er en separasjonsnode hvis det finnes en kant $\langle u, v \rangle$, i T, slik at $index(u) \leq low(v)$.



2-sammenhengende grafer i lineær tid IV

Algorithm: Hopcroft-Tarjan Algorithm

Output: Alle separasjonsnoder i inputgrafen G

```
Procedure HopcroftTarjan(G, u, depth)
    visited[u] = true
    low[u] = index[u] = depth
    childCount = 0
    for each edge {u, v} in G do
        if visited[v] = false then
            childCount = childCount + 1
            HopcroftTarjan(G, v, depth + 1)
            low[u] = min(low[u], low[v])
            if index[u] != 1 then
                if index[u] ≤ low[v] then
                    sep_vertices.add(u)
        else
            low[u] = min(low[u], index[v])
    if index[u] = 1 then
        if childCount > 1 then sep_vertices.add(u)
    return sep_vertices
```

Vi kjører en DFS og oppdaterer indeks og low-nummeret til hver node underveis

Vi får at *low*[u] er det minste av (1) *index*[u], (2) low-nummeret til alle sine barn, og alle nodene u kan nå via en back-edge.