Øving 6 Algoritme og datastrukturer IDATT2101

# *Deloppgave 1: Bredde-først søk*

# Implementasjon

For å implementere en bredde-først søk algoritme, ble følgende kode implementert.

Et bilde som inneholder tekst, skjermbilde, Font, nummer

Automatisk generert beskrivelse

Figur : Utsnitt av metoden bfs som gjør et bredde-først søk.

Denne algoritmen tar hensyn til at grafen er implementert som naboliste. Ved start initialiserer den forgjengere (predecessors), kø (cue) og putter oppgitt start node i køen. While-løkka sjekker om at køen ikke er tom, og for-løkka traverserer gjennom alle kantene til som den har tatt ut av køen. Herfra sjekkes alle nodene fra kantene, og hvis avstanden til kantens sluttnodes forgjengeren er uendelig, settes distansen til den aktive nodens distanse + 1. Deretter puttes sluttnoden inn i køen, og neste traversering begynner osv. Mye av denne koden er tatt fra boka (noe som jeg føler skjedde litt for ofte i denne øvingen).

# Resultat

Siden resultatet skulle vises på et spesifikt format har jeg implementert metoden nedenfor:  
Et bilde som inneholder tekst, skjermbilde, Font, line

Automatisk generert beskrivelse

Figur : Utsnittet av metoden bfsResultat som returnerer resultatet i en streng på et definert format.

Her bruker jeg en StringBuilder for å lage resultatet. Denne metoden fungerer på alle filen, men er best tilpasset for mindre grafer. Her brukes ternary operasjoner for å tilpasse mellom tallsifre og mellomrom for å matche plasseringen som strengen «Node Forgj Dist» gir.  
  
Resultatet fra BFS på grafen ø6g1 med startnode 5, gir følgende resultat:  
Et bilde som inneholder tekst, skjermbilde, Font, nummer

Automatisk generert beskrivelse

Figur : Utskrift fra bfs på øg61 med startnode 5.

Dette er da også identisk til eksempelet i oppgave fila. I kildekoden gjør main metoden flere av disse bredde-først søkene med tilfeldige startnoder.

# *Deloppgave 2: Topologisk sortering*

For å implementere topologisk sortering ble metodene *dfTopology* og *topologySort* implementert slik figuren under viser:

Et bilde som inneholder tekst, skjermbilde, Font

Automatisk generert beskrivelse

Figur : Utsnitt av både dfTopology og topologySort metodene.

I metoden dfTopology som ligner mye på et dybde-først søk har to parametere: startNode og firstNode. Noe som kan virke litt visvisende. Grunnen til at jeg har kalt dem for dette er fordi startNodeer den noden man starter dybde-først søket med, og firstNode som er den noden som vanligvis ligger først i den lenkede resultatlisten (TopologyList). Metoden traverserer gjennom nodene den ikke har vært i før og lenker den ferdige noden (firstNode) fremst i listen, før den til slutt returnerer start noden startNode*.* Metoden topologySort utfører dfTopology på alle nodene i grafen, og returnerer til slutt hodet til listen som er sortert i topologisk orden. Her har det litt å si for hvordan rekkefølgen på nodene kjører dfTropology. Det kan endre litt på rekkefølgen i den sorterte topologiske listen, selv om listen fortsatt er riktig sortert. Disse metodene er også mer eller mindre hentet fra boka.

# Resultat

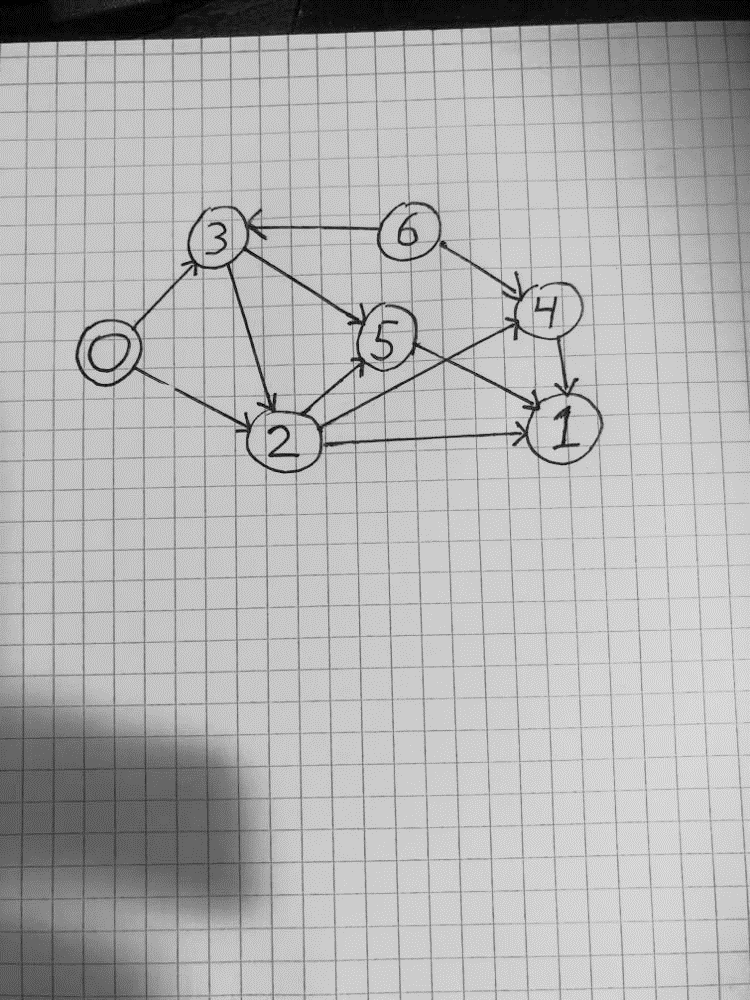
Resultatet av å kjøre metoden topologySort på ø6g og ø6g7 gir følgende rekkefølger:

Et bilde som inneholder tekst, skjermbilde, Font, line

Automatisk generert beskrivelse

Figur : Utskrift fra topologisk sortering.

Fra oppgavefila ser vi derimot at dette ikke er helt identisk med det topologiske resultatet fra ø6g5. Dette kan forklares visuelt med en liten illustrasjon:



Figur : Illustrasjon av grafen ø6g5.

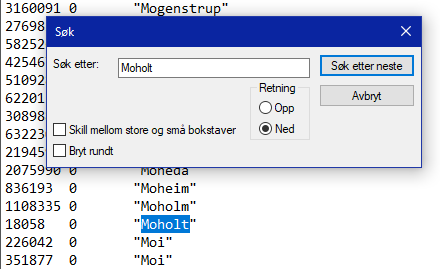
Fra min løsning fikk jeg rekkefølgen: 6 0 3 2 5 4 1, mens oppgavens versjon er 6 0 3 2 4 5 1. Ved å analysere grafen ser man 6 og 0 ikke er avhengig av noen andre noder. Rekkefølgen kan altså begynne på 6 eller 0. Begge er vilkårlig, men den ene må komme etter den andre. Deretter kan vi slå fast at neste node må være 3, siden 3 er den neste noden som er tilgjengelig når 6 og 0 er åpnet. Deretter må neste node være 2 av samme grunn. Herfra åpner det igjen to nye muligheter. Både 4 og 5 er tilgjengelige. I min versjon fikk jeg 5 4, mens oppgavens versjon er 4 5. Begge to er derfor gyldige. Til slutt åpner det seg for 1, og rekkefølgen ferdig.

Som oppsummering ser vi at det er to steder hvor resultatet kan endre på rekkefølgen mellom to noder. Det må derfor finnes ulike kombinasjoner for hvordan denne grafen kan bli topologisk sortert.

# *Kryss mellom Kalvskinnet og Moholt*

(Kildekode er ikke med i vedlegg)

For å finne ut hvor mange kryss det er mellom Kalvskinnet og Moholt kan man bruke filene *ø6Skandinavia* og *ø6Skandinavia-navn*. Men først må disse monstrene lastes ned.   
  
I filen *ø6Skandinavia-navn* finner vi ved å søke at noden til Kalvskinnet er 37774, og at noden til Moholt er 18058. Vi kan dermed bruke et BFS for å finne antall veikryss mellom dem.  
  
Et bilde som inneholder tekst, skjermbilde, display, programvare

Automatisk generert beskrivelse 

Men å lese resultatet fra BFS-algoritmen på terminalen er noe som ikke lar seg gjøres effektivt. Et alternativ til dette er heller å skrive ut resultatet til filer slik som kodeutsnittet under viser.  
Et bilde som inneholder tekst, skjermbilde, Font

Automatisk generert beskrivelse

Ved å kjøre denne koden opprettes to filer: BFS\_Result\_Kalvskinnet.txt og BFS\_Result\_Moholt.txt.

Et bilde som inneholder tekst, skjermbilde, Font, line

Automatisk generert beskrivelse

Herfra kan vi nå finne avstanden fra Kalvskinnet og Moholt ved å søke etter hverandres node-nummer i hver fil. Altså finner vi at antall veikryss fra Kalvskinnet og Moholt er 49, og antall veikryss fra Moholt til Kalvskinnet er også 49.

Et bilde som inneholder tekst, skjermbilde, display, programvare

Automatisk generert beskrivelseEt bilde som inneholder tekst, skjermbilde, display, nummer

Automatisk generert beskrivelse