2021

IN2010

OBLIG 2

SVEIN-GISLE SÆTRE

Oblig 2

Programmet kjøres ved å kompilere Oblig2.java og kjøre kommandoen java Oblig2. Programmet kjører så gjennom alle oppgavene etter hverandre og printer svarene til terminalen. På min pc brukte programmet først rundt 1 minutt og 10 sekund på å fullføre alle oppgavene, der oppgave 3 desidert var den største synderen. Jeg brukte så løsningen fra oppgave 2 og den korteste stien der til å returnere en vekt som oppgave 3 iallfall måtte finne en mindre eller lik løsning til. Det eliminerte mange iterasjoner og programmet kom da ned i 9 sekunder. (Jeg oppdaget senere at oppgaven var så treig til å starte med fordi jeg hadde implementert comparable-metoden til Nodene feil vei. Innsparingen jeg nå får ved å bruke algoritme 2 først er minimal, men jeg lar det stå som en kuriositet.) Dette gjør nok ikke kompleksiteten til algoritmen bedre siden man fra algoritmen i oppgave 2 kan finne den lengste stien som er mulig om grafen er en lenket liste og målnoden er på slutten. Algoritmen blir nok i motsetning heller mer kompleks siden den nå baserer seg på å kjøre algoritmen fra oppgave 2 først, men i praksis vil det derimot i majoriteten av tilfellene minske kjøretiden til algoritmen i oppgave 3 ganske drastisk.

Oppgave 1: Bygg grafen

Oppgave 1 kjører direkte i main-metoden, mens de øvrige oppgavene kjøres i egne metoder.

Algoritme

For å representere grafen har jeg valgt å kombinere objektorientering og nabo-liste.

Skuespillere er lagret som noder som inneholder:

- ID
- Navn på skuespiller
- Nabo-liste, en liste med kanter
- Film-liste, en liste med filmer de har spilt i
- Vekt

Filmer blir opprettet som egne filmobjekter i innlesing av fil for å holde på informasjonen, men informasjonen om filmer blir så inneholdt i kant-objekter. Etter at film-filen er lest inn og alle filmene er lagret som film-objekter leses skuespiller-filen inn. Når skuespiller-filen leses inn blir alle skuespillere opprettet som nodeobjekter. Når alle filmer og skuespillere er lagret kan en forløkke kjøre gjennom hver film for å knytte sammen skuespillere ved å opprette kantobjekter som inneholder:

- Vekt
- Node1 og Node2
- Filmnavn

Kjøretid

Om vi regner filene som input kan vi si at film-fil er X og skuespiller-fil er Y, som til sammen blir N. Algoritmen kjører først gjennom X og så gjennom Y. Det itereres så gjennom alle filmene som tilsvarer å kjøre gjennom X igjen, så vi har at algoritmen må gjøre 2X+Y operasjoner som kan forenkles til at algoritmen kjører i O(N).

Kjøretid: Rundt et sekund

Oppgave 2: Six Degrees of IMDB

Algoritme

Oppgave 2 bruker et bredde-først-søk for å finne korteste sti til skuespilleren vi søker etter. Til å begynne med implementerte jeg en metode der jeg lagret alle noder og kanter for hvert lag i dybdesøket, sammen med en index for å kunne finne veien tilbake til startnoden etter at korrekt node ble funnet. Det ble derimot noe rot med indekseringen som gjorde at feil svar ble printet, så jeg la heller til en ledende-kant attributt i hver node som gjorde at jeg kunne nøste tilbake til start via disse ledende kantene som ble opprettet når søket kjørte. Listen over besøkte er et HashSet som ifølge dokumentasjonen har $O(1)$ kjøretid på contains-metoden, i motsetning til ArrayList som har $O(n)$ kjøretid på samme metode.

Kjøretid

Programmet sjekker alle naboer til startnoden, og så alle naboer til naboene etc. I verste fall ville grafen vært en lang lenket liste hvor noden vi søker etter er på slutten, og kjøretiden ville da ha vært lik antall kanter, O(|E|).

Kjøretid: Under et sekund

Oppgave 3: Chilleste vei

Algoritme

I forklaringen ser jeg først bort fra at algoritmen baserer seg på å kjøre algoritmen til oppgave 2 først.

Oppgave 3 bruker en modifisert versjon av Dijkstras algoritme. Algoritmen holder løpende telling på korteste vei til mål-noden så langt, og hopper over videre sjekking når vekten til noden og kanten vi sjekker er høyere enn korteste vei.

Kjøretid

Kjøretiden til Dijkstra er $O(|E|_{log}|v|)$. Min implementasjon er derimot litt annerledes, så vi ser på den:

Min while-løkke vil ha samme kompleksitet som vanlig, altså $O(|V|)$

Heap-remove er også lik og vil være O(log|V|), slik at vi har $O(|V|_{log}|V|)$

For-løkken er lik, med de endringer at løkken vil stoppe om vektingen har overgått nåværende korteste distanse, eller om vi finner målnoden. Dette vil derimot ikke ha noe å si i et verste tilfelle og vi burde dermed ha samme kompleksitet som i Dijkstras algoritme - $O(|E|_{log}|v|)$.

Kjøretid: Under et sekund

Oppgave 4: Komponenter

Algoritme

Oppgave 4 bruker en slags versjon av Kruskals algoritme for å bygge komponenter. Siden jeg ikke bare vil finne et spenntre fra en node må jeg derimot ikke returnere et tre, men komponenter. Igjen bruker jeg HashSet for å lagre componenter og kanter, da jeg trenger å hente disse via navn og ikke indeks. Dette vil ifølge dokumentasjonen være O(1) operasjoner, i motsetning til O(n) operasjoner om jeg måtte ha iterert gjennom en ArrayList.

Kjøretid

For å legge til alle kanter må jeg hente alle kanter ut fra alle noder, som blir $O(|E|)$. Datastrukturen HashSet hjelper også med å unngå å legge til duplikater, altså kanter som ligger i en node og nabonoden.

While-løkken min vil gjøre |E| iterasjoner med HashSet-remove som er $O(1)$.

Jeg har en $for(alle\ kanter)$ inne i while-løkken, men det er bare en slags måte å ta ut en vilkårlig kant på. Den kjører bare en gang og er dermed $O(1)$.

Implementeringen min av løkken over naboene er litt annerledes implementert, men den vil fortsatt gjøre O(|E|) iterasjoner med oppdatering i kø som er $O(_{log}(|V|))$.

Til sammen er kompleksiteten da på $O(|E|_{log}(|V|))$

Kjøretid: Rundt 7 sekunder