IN2010 - Oblig 2

Stian Østgaard
Thuan Tran
Embrik Thoresen

19.10.2021

Kjøring av program

python3 oblig2.py → kjøring av kode.

time python3 oblig2.py → kjøring av kode inkludert tidsbruk.

Oppgave 1

Fremgangsmåte for å løse oppgaven:
1. Vi starter med å lese inn linjene fra de to ulike filene som inneholdt alt av informasjon vi trengte.
2. Vi lagde så en dictionary som inneholdt skuespillere som nøkkel og alle filmene skuespilleren er med i som verdi, en dictionary som inneholdt filmene som nøkkel og alle skuespillere som er med i filmen som verdi, og en dictionary som kombinerte disse. Dette gir oss den endelige 'ordboka'. Denne har id-navn som nøkkel og en liste av tupler som er filmnavn og rating, etterfulgt av id-navn på medskuespillleren.
3. Vi bruker så denne 'ordboka' til å regne ut antall noder og kanter til grafen.

Slik vi tolket oppgaven med hensyn på senere bruk av 'hovedordboka' vår tenkte vi at dette er den mest effektive og relativt enkle måte å løse problemet på. Vi kan ikke se noen feil eller mangler med vår implementasjon, selv om vi ikke er sikre på om dette er det mest effektive måten å løse oppgaven.

Når vi skal bygge grafen så lagrer vi alt av informasjon i dictionary-en som starter på imdb. Kjøretidskompleksitet på å lage denne dictionary-en er $O(|V|^2 \cdot |E|)$.

Oppgave 2

Fremgangsmåte for å løse oppgaven:
1. Vi definerer først en liste, en queue og en dictionary som vi senere skal bruke.
2. Nå kjører vi 'breadth first search' for å lage et tre. Når vi kommer til sluttnoden som har en navn-id så itererer vi oss bakover ved å bruke dictionary hvor verdien til nøkkel er foreldrenoden. Vi slutter å iterere når vi har kommet til

startnoden. For hver slik iterasjon appender vi dette til en liste 'path' som vi printer ut for å gi oss riktig svar.

På samme måte som forrige oppgaver mener vi at dette er en effektiv måte å løse oppgaven på. Ved kjøreeksemplet tar denne koden svært kort til å kjøre, og vi ser ingen andre effektive måter å implementere dette på. Vi mener det ikke er noen mangler eller feil med koden vår.

Når vi skal finne korteste sti mellom to skuespillere gjør vi først noen konstante operasjoner før vi kjører BFS. Denne algoritmen gir oss $O(|V| + |E|)$. Til slutt gjør vi noen konstante operasjoner før vi går igjennom en for-loop i en for-loop hvor vi printer ut svaret. Dette gir oss $O(|V|^2)$. Totalt får vi en kjøretidskompleksitet $O(|V|^2)$. Det skal sies at dette er grunnet print-funksjonen vår. I praksis vil denne funksjonen ha en kjøretid betraktelig mindre enn hovedfunksjonen vår.

Oppgave 3

Fremgangsmåte for å løse oppgaven:
1. Vi implementerer Dijkstras algoritme ved å først definere en heap 'unvisited', en mengde 'visited', en dictionary for å lage et tre og en liste som vi skal bruke for å printe resultatet.
2. 'Unvisited' henter nabonodene til en gitt node, og 'visited' sjekker om vi har besøkt en node tidligere. Ellers er resten implementert som forventet. Etter å ha regne ut vektene til kantene og blitt ferdig med algoritmen så bruker vi funksjonen vår til å finne veien fra start til slutt.

Vi har noen utfordringer på kjøretiden i denne oppgaven. Vi vet at det må finnes en bedre og mer effektiv måte å løse oppgaven på, hvor det tar for lang til å finne den chilleste veiene og printe ut slik som oppgaven sier vi skal gjøre. Vi har derimot valgt å ikke bruke alt for lang tid på å prøve å finne denne måten. Dette er den delen av programmet som tar lengst til å kjøre.

Når vi skal finne chilleste vei mellom to skuespillere gjør vi først noen konstante operasjoner før vi kjører Dijkstras algoritme. Denne algoritmen gir oss $O(|V|^2)$. Til slutt gjør vi noen konstante operasjoner før vi går igjennom en while-loop for å finne totalt vekt, og deretter går igjennom en for-loop som printer ut svaret. Totalt får vi en kjøretidskompleksitet på $O(|V|^2)$.

Oppgave 4

Fremgangsmåte for å løse oppgaven:
1. Vi definerte en mengde 'unvisited' og en dictionary 'components'. 'Unvisited' sørger for at vi har besøkt alle komponentene, mens 'components' lagrer antall noder i en komponent og antall komponenter av den størrelsen.
2. Vi kjører BFS igjen, men denne gangen bruker vi en counter for å telle hver gang vi er på en ny node. Til slutt oppdaterer vi 'components' og returnerer components når while-løkken er ferdig.

Vi mener vi har løst oppgaven på en effektiv måte, hvor det tar relativt kort tid å kjøre denne delen av programmet. Vi ser ingen mangler eller feil med denne koden, gitt hva oppgaven spør etter.

Når vi skal finne størrelsen på komponentene i grafen og antall komponenter som har denne størrelsen starter vi med å gjøre noen konstante operasjoner, før vi kjører en modifisert versjon av BFS som også teller antall komponenter. Til slutt går vi inn i en for-loop som printer ut antall komponenter som har en gitt størrelse. Totalt får vi en kjøretidskompleksitet $O(|V| \cdot |E|)$.

Kjøretidskompleksitet

Vi vet fra alle tidligere hva kjøretiden for hver oppgave er. Ved å se på dette får vi at kjøretidskompleksiteten på hele programmet er $O(|V|^2 \cdot |E|)$. Ved å kjøre 'time python3 oblig2.py' får vi at den raskeste tiden er 42 sekunder. Vi syns dette er relativt kjapt, men skjønner at utfordringene vi hadde var å optimalisere oppgave 3. Det er altså denne oppgaven som gjør at programmet går tregere enn det potensielt kunne gjort dersom vi jobbet hardere for å forbedre kjøretiden til hele programmet.