UiT Norges arktiske universitet

Institutt for Informatikk

Sudoku

Oblig 1

Erling Heimstad Willassen

INF-1400, Vår 2025

Innholdsfortegnelse

1 Innledning

I denne obligatoriske innleveringen skal en programmere en algoritme for å løse sudoku problemer i dataspråket Python. Sudoku er et spill der hver kolonne, rad og boks på brettet skal inneholde alle tall fra 1 til 9 uten å inneholde replikanter. Når alle rad, kolonner og bokser har unike tall fra 1 til 9, er da sudokuen løst.

2 Design

I oppgaven fikk man en pre-kode som inneholdt funksjoner og klasser som en skal fortsette å skrive algoritmer i. I pre-koden eksisterte det en hovedklasse («Board») som man skal arve fra til en sudoku brett («SudokuBoard)», og funksjoner man skal skrive algoritmer for å løse et eller fler sudoku problemer. Videre skal man skrive inn klasser for elementer og «Square» som skal fylle en sudoku brett som videre blir brukt til å finne en løsning til en sudoku brett. I tillegg til dette inneholdt pre-koden også tekst-filer for sudoku problemer og en python kode for å lese filene og returnere en liste over alle tallene/firkantene til en sudoku problem.

«Square» klassen skal være en firkant, eller nummer, på en sudoku brett. Siden det er 81 firkanter i et sudokubrett blir det 81 «Squares» totalt som blir produsert i algoritmen. Hver «Square» vil ha variabler som sitt eget nummer, referanse til elementer den tilhører (rad, kolonne og boks). En «Square» inneholder også flere funksjoner som hjelper den å vite hvilken tall som er gyldig å plassere på sudokubrettet.

«Element» klassen representer en rad, kolonne eller boks på en sudoku brett. Siden en sudoku brett er 9x9 firkanter vil det være ni elementer av rad, kolonne og boks hver, totalt 27 elementer. Funksjonen med «Element» klassen er at den skal inneholde alle «Squares» den inneholder, og dermed holde kontroll på hvilke tall som er lov eller ikke er lov en «Square» kan ha. «Square» kan dermed sjekke sine elementer den er del av og finne ut hvilken tall den kan sette i Sudokubrettet.

Hovedklassen «Board» skal i teorien være en klasse med funksjoner som kan bli brukt til å produsere hvilket som helst brettspill. Derfor skal «Sudoku» klassen som arver fra «Board» inneholde alle funksjonene og variablene for å produsere en sudoku brett. «SudokuBoard» har ingen funksjoner i seg selv og arver alle funksjonene til å utføre løsningen fra «Board». «SudokoBoard» har derimot en liste av alle squares, elementer og brukte squares i sin utregning.

3 Implementasjon

Algoritmen starter ved at et sudokubrett blir lest av tekstfil algoritmen som fulgte med pre-koden og returnert som en liste. Deretter blir «SudokuBoard» klassen initiert ved å bruke denne returnerte listen som et argument til sin algoritme.

I «SudokuBoard» klassen blir «Board» klassen arvet i __init__ funksjonen og den returnerte sudokubrett listen lagret til senere bruk. En «solved» variabel blir laget med verdi 0; når et sudokubrett blir løst vil denne bli til 1 som avslutter en while-loop til selve problemløsningen. Deretter blir det laget lister for å inneholde alle «Squares», elementer på sudokubrettet og «Squares» som har blitt løst. Listene blir brukt for å ha kontroll og tilgang til alle klassene i problemløsningen algoritmen. Listen for «Squares» blir initiert ved å arve fra «Board» klassen der alle «Squares» blir laget og lagt til i selve listen. Hver «Square» blir også lagt tilbake i listen for selve sudokubrettet for enklere printing i terminal. Deretter blir dannelsen av elementene initiert ved å arve fra «Board» klassen. Hver «Square» blir deretter lagt til sitt respektive element, og hver «Square» får også en referanse til sitt element. En liste i hvert element blir laget som inneholder alle tall av sine «Squares», og er derfor forbudt for andre «Squares» å bruke.

Etter oppsett av elementer og «Squares», blir problemløsning algoritmen initiert i en while-loop under «SudokuBoard» klassen arvet fra «Board». Her itererer algoritmen alle «Squares» som finner seg i listen for «Squares». «Squares» velger da første instans av et nummer som er lovlig etter å ha gått gjennom listene over ulovlige tall i fra sine tre elementer den er del av. Nummeret vil da bli lagt til i ulovlige tall listen av sine elementer slik at de nestkommende «Square» ikke kan velge samme tall. «Squares» som allerede inneholder et tall (et fastsatt tall fra sudoku brettet) blir ignorert siden disse kan ikke endres på etter reglene av sudoku. Hvis en «Square» har ingen gyldige tall å velge etter å ha gått gjennom listen over ulovlige tall fra sine elementer vil «backtrack» funksjon bli initiert.

«Backtrack» funksjonen fungerer ved å at algoritmen går gjennom alle «Squares» som har tidligere blitt gått gjennom i while-loopen i revers rekkefølge («Square» som hadde ingen muligheter for tall blir først i listen). «Square» som ikke kunne legge tall initiere en «iterator» funksjon ved at den går innom forrige «Square» og endre den slik at den velge neste tall i sin lovlige tall liste. Hvis den listen har ingen andre muligheter, vil den også initiere «iterator» funksjonen slik at den går innom forrige «Square» for den igjen. Dette vil gjøres helt til

«backtrack» funksjonen har en «Square» som kan velge et annet tall fra sin liste over lovlige tall, og «backtrack» funksjonen avsluttes tilbake til start. «Solved» variabelen blir satt til 0, og while loopen initieres på nytt ved å iterere gjennom alle «Square» enda en gang. Når alle «Squares» har blitt gått gjennom vil «Solved» bli satt til 1, og while-loopen avsluttes. Brettet er da fyllt med «Squares» med et tall som oppfyller sudokureglene, og algoritmen avsluttes.

4 Resultat

Algoritmen fungerer som tiltenkt med å løse et sudokubrett etter reglene for sudoku. I listen nedenfor (Tabell 1) er det oppført behandlingstid for antall sudokubrett, og man kan se at det er tilnærmet en lineær regresjon hvor mange problemer algoritmen kan løse per sekund.

Tabell 1: Oversikt av tidsforbruk over antall sudokubrett løst av algoritmen.

Antall sudokubrett	Antall sekunder
10 (ti)	0,01252 sekunder
100 (hundre)	0,10794 sekunder
1000 (tusen)	1,15886 sekunder
10000 (titusen)	11,65706 sekunder
100000 (hundre tusen)	124,08961 sekunder
1000000 (en million)	~1200 sekunder (ikke gjennomført)

5 Diskusjon

Algoritmen fungerer bra til formålet den er gitt og har klart å løse alle problemer den har blitt utsatt for. Den er rask til formålet med å kunne løse tusen sudoku brett på litt over ett sekund, men ved en million brett vil den ta omtrent 20 minutter. Den innehar flere lister og algoritmer som er nok unødvendig og kunne blitt optimalisert bedre, og grunnen til at dette var implementert er oversikt og enklere å kode. Som for eksempel listen over alle «Squares» er lagt i en egen liste i «SudokuBoard» klassen, men alle «Squares» ble også plassert i listen fra «reader()» funksjonen og dermed har man to lister som innehar alle «Squares». Grunnen til at dette ble gjort ligger i forskjellen mellom listene. Forskjellen mellom listene er at listen i «SudokuBoard» er en 1-dimensjonal-liste som gjør det lettere å feilsøke og iterere igjennom, mens den andre må man ha en for-loop for å iterere gjennom alle «Squares» og litt mer avansert for å se gjennom hvor det gikk galt under kodingen når brettene ikke ble løst som

tiltenkt. Ved å inneha flere lister og andre variabler vil algoritmen bruke lengre tid og bruke mer minne som også vil gjøre tidsbruken større.

Problemløsningsalgoritmen er også gjort gjennom «bruteforce» metoden, at man tvinger fram en løsning ved å prøve flere forskjellige kombinasjoner helt til det funker. Dette er heller ikke en optimal måte å kode fram en algoritme for å bruke kortest tid mulig ved en løsning. Andre metoder som krever litt mer matematisk gjennomgang er raskere, men mer avansert å kode og tenke seg igjennom. «Bruteforce» er enklere å kode, men svakheten er at den vil prøve mange mulige kombinasjoner helt til den finner en løsning som kan ta veldig lang tid.

Dataspråket Python er heller ikke kjent for å være rask eller god i minnebruk, men er mer brukervennlig. Python har flere bakgrunns algoritmer som gjør at koden fungerer slik at man har skrevet, men på den andre siden gjør dette at Python må gå gjennom flere algoritmer for å kjøre en kode. Et annet språk som C eller C++ er mer «rå» og har mindre «bakgrunnsstøy» enn Python, og vil mest sannsynlig kjøre samme problemløsning mye raskere enn med Python.

Ved tanke om pre-koden og hva som kunne gjort annerledes, er det ikke så mye som jeg ville endret på. Oppsettet i pre-koden er logisk og gir mening, og man klarer å se en rød tråd i tanken om koden man ble gitt. Eneste som jeg personlig ikke syntes ga mening var at «Board» klassen innehadde funksjonene å lage elementene rad, kolonne, og boks, som ikke nødvendigvis alle brettspill er viktig å ha. Derfor kan det være vanskelig å iverksette andre brettspill i samme kode eller arving fra «Board» når den innehar slike funksjoner, spesielt når det gjelder boks elementet som er veldig en sudoku egenskap. Utenom det, er det lite jeg ville endret med pre-koden som ble gitt.

6 Konklusjon

I oppgaven ble det gitt at man skal kode en algoritme for å løse sudokubrett ved å arve fra en «Board» klasse i Python. Ved å gjøre dette, kunne man innhente funksjoner fra klassen til å løse en eller flere sudokubrett. Algoritmen som ble skrevet klarte å løse flere brett under sekundet, og har klart å løse alle brett den har blitt matet med.

7 Source

“oblig1_precode_datafiles.zip”, til faget INF-1400, publisert 03 Januar 2025