Kompleksitet

Oversikt • Så langt i IN2010 har vi fokusert på ressursbruk for konkrete algoritmer: ○ Hvor mye tid og minne en algoritme krever. ○ Vi har analysert O-notasjon for ulike algoritmer og datastrukturer. • Denne uken skifter fokuset fra algoritmer til selve problemene: ○ Vi spør oss: Hvor vanskelig er det å løse problemet, uavhengig av algoritmen? ○ Dette handler om problemets kompleksitet, ikke bare algoritmens. • Et viktig poeng: ○ Alle algoritmene vi har sett til nå, har vært polynomielle: § Tiden for å løse problemet vokser som et polynom i inputstørrelsen n (f.eks. O(n), O(n^2), O(n^3), O(n log n)). ○ Polynomielle algoritmer regnes generelt som praktisk effektive. • Neste steg blir å forstå: ○ At noen problemer ikke har kjent polynomalgoritme, og hvordan dette knyttes til klasser som P og NP. ○ Hvorfor noen problemer kan være teoretisk mulig å løse, men praktisk umulige for store n.

Kompleksiteten til algoritmer • Kompleksitet handler om hvor mye ressurser en algoritme bruker: ○ Tid: antall steg algoritmen utfører. ○ Minne: hvor mye lagringsplass som kreves. • I IN2010 fokuserer vi hovedsakelig på tidskompleksitet. • Tidsmåling gjøres vanligvis i forhold til inputstørrelse n, og vi ser ofte på verste tilfelle (worst-case).

Et eksempel på en ineffektiv løsning • Problem: Summer tallene fra 1 til n. • Ineffektiv algoritme: ALGORITHM: SumTo(n) Input: Et naturlig tall n Output: Summen av alle tallene fra 1 til n 1 Procedure SumTo(n) 2 sum ← 0 3 for i ← 1 to n do 4 for j ← 1 to i do 5 sum ← sum + 1 6 return sum • Analyse av løkker: ○ Ytre løkke: n iterasjoner. ○ Indre løkke: varierer fra 1 til i. • Totalt antall steg: summen av 1 + 2 + … + n.

Hvordan regne antall iterasjoner • Summen av tallene 1 til n: $1+2+3+\quad+n$ • Triks for å summere: skriv summen fremover og bakover: $(1+2+\quad+n)+(n+(n-1)+\quad+1)=n\cdot(n+1)$ • Del på 2 for å få summen: $1+2+\quad+n=n\cdot(n+1)2$ • Tidskompleksitet: ○ Selv om algoritmen utfører n(n+1)/2 steg, er dette O(n^2), altså kvadratisk tid. ○ Det finnes mer effektive løsninger (f.eks. sum=n(n+1)/2) som kjører i O(1).

Litt om kjøretidskompleksitet • Hensikten med kjøretidsanalyse: Forstå hvordan kjøretiden til en algoritme vokser når inputstørrelsen n øker. • Eksempel på vekstkurver: ○ $10\cdot n$ – vokser lineært. ○ n2 og n(n+1)/2 – vokser kvadratisk. • Observasjoner: ○ Kurvene beholder formen selv om konstantfaktorer endres. ○ Store-O-notasjon (O(⋅)) fanger den generelle veksttrenden, men ignorerer detaljer og konstante faktorer. ○ Vekst av algoritmen domineres av det tregeste leddet, ikke av små konstante forskjeller. • Perspektiv fra O-notasjon: 1. Fokus på hvordan funksjonen vokser med n. 2. Konstantfaktorer ignoreres. 3. Algoritmens “tregeste ledd” bestemmer asymptotisk vekst. 4. Gjør det enklere å resonnere om effektivitet og skalering.

Tilbake til summen av tallene 1 til n Ineffektiv løsning (kvadratisk tid) 1 Procedure SumTo(n) 2 sum ← 0 3 for i ← 1 to n do 4 for j ← 1 to i do 5 sum ← sum + 1 6 return sum • Analyse: ○ Ytre løkke: n iterasjoner ○ Indre løkke: 1 til i iterasjoner ○ Totalt: n(n+1)/2=O(n2) • Ineffektiv fordi det finnes mye raskere løsninger.

Bedre løsning (lineær tid) 1 Procedure SumTo(n) 2 sum ← 0 3 for i ← 1 to n do 4 sum ← sum + i 5 return sum • Analyse: ○ Én løkke: n iterasjoner ○ Tidskompleksitet: O(n) – lineær tid

Optimal løsning (konstant tid) 1 Procedure SumTo(n) 2 return n * (n+1) / 2 • Analyse: ○ Ingen løkker, kun en formel ○ Tidskompleksitet: O(1) – konstant tid ○ Dette viser viktigheten av å se etter direkte matematiske løsninger når mulig.

Kompleksiteten til problemer I kompleksitetsteori er hovedfokuset å forstå hvor vanskelig et problem er i sin natur, altså problemets iboende vanskelighet, uavhengig av konkrete implementasjoner. Dette gjør vi ved å studere den mest effektive algoritmen som kan løse problemet. • Når vi snakker om vanskelighetsgrad, handler det om den teoretisk raskeste måten å løse problemet på, ikke nødvendigvis algoritmer vi allerede kjenner. • Et sentralt mål er å kunne argumentere for at det ikke finnes noen algoritme som kan løse problemet raskere enn en viss tid. • Å vise at et problem ikke kan løses mer effektivt er ofte svært utfordrende, og mange slike spørsmål er fortsatt uløste i datavitenskapen (for eksempel P vs NP-problemet).

Avgjørelsesproblemer I kompleksitetsteori arbeider vi som regel med avgjørelsesproblemer. Dette er problemer der vi får en instans og et spørsmål som vi må svare på med JA eller NEI. • Tidligere i algoritmekurset har vi sett på problemer som kan gi mer komplekse eller rike output, som lister, tall eller strukturer. • Selv slike "rikere" problemer kan ofte formuleres som et relatert avgjørelsesproblem, hvor spørsmålet er om et bestemt kriterium er oppfylt. • Eksempel: I stedet for å finne den korteste veien mellom to byer, kan et avgjørelsesproblem være: "Finnes det en vei som er kortere enn 100 km?" • Hvis vi har en algoritme som løser det originale problemet, kan vi enkelt bruke den til å besvare det tilhørende avgjørelsesproblemet. • Omvendt er de rikere variantene minst like vanskelige som deres avgjørelsesproblem; det vil si at hvis vi kan løse avgjørelsesproblemet, kan vi potensielt bygge løsninger for den rikere versjonen. Kort sagt: Å forstå og analysere avgjørelsesproblemer gir oss et kraftig verktøy for å karakterisere kompleksiteten til problemer, uten å måtte håndtere all ekstra kompleksitet fra mer detaljerte output.

Kompleksitetsklassen P Definisjon: Et avgjørelsesproblem hører til klassen P hvis det finnes en algoritme som løser det i polynomiell tid i forhold til størrelsen på inputen. • Med andre ord: Det finnes en algoritme som har kjøretid $O(n^k)$ for en konstant k, hvor n er størrelsen på probleminstansen. • Intuisjon: Polynomielle algoritmer kan ofte tenkes som programmer med flere nøstede løkker, hvor dybden av nestingen bestemmer graden k. Eksempel: Procedure Polynomial(n) for i1 ← 0 to n–1 do for i2 ← 0 to n–1 do ... for ik ← 0 to n–1 do Constant(i) • Her viser k-nivåer med løkker en kompleksitet på $O(n^k)$. • Alle algoritmer vi har sett i kurset så langt har vært polynomielle, altså tilhørt P. • Viktig: O-notasjon gir en øvre grense, så en algoritme i O(log n) er også i $O(n^2)$ – den er fortsatt polynomiell.

Eksempel: Sudoku n × n Problemet: • I klassisk Sudoku (9×9) får du et delvis utfylt brett. • En løsning er et komplett utfylt brett der hver rad, kolonne og boks inneholder tallene 1 til 9 nøyaktig én gang. Avgjørelsesproblemet – SUDOKU n × n: • Instans: Et delvis utfylt n×n Sudoku-brett • Spørsmål: Finnes det en gyldig løsning som fullfører brettet korrekt? Vanskelighetsgrad: • Ingen kjent polynomiell algoritme for å løse dette problemet generelt. • Det er heller ikke kjent om en slik løsning kan eksistere. • Problemet vokser ekstremt raskt med n: ○ n = 4 → 288 gyldige brett ○ n = 9 → 6 670 903 752 021 072 936 960 gyldige brett Sjekking vs. løsning: • Selv om det er vanskelig å finne en løsning, er det enkelt å sjekke om en gitt løsning er gyldig. • Dette skiller løsningsproblemet fra verifikasjonsproblemet – verifikasjon kan ofte gjøres i polynomiell tid, mens det å finne løsningen kan være mye vanskeligere.

Verifisering av løsninger med sertifikater Avgjørelsesproblemer er problemer hvor man får en instans og må svare JA eller NEI. • Selv om det kan være vanskelig å finne en løsning, kan vi ofte verifisere at en gitt løsning er korrekt. • Dette gjøres ved hjelp av et sertifikat, som er et ekstra bevismateriale som gjør det mulig å kontrollere løsningen raskt. Eksempel: Sudoku • Spørsmålet: "Har dette brettet en gyldig løsning?" • Dersom svaret er JA, kan vi bruke en verifikasjonsalgoritme: ○ Input: Det delvis utfylte brettet + et løst brett (sertifikatet) ○ Algoritmen sjekker at alle regler er fulgt: § Hver rad inneholder tallene 1–9 nøyaktig én gang § Hver kolonne inneholder tallene 1–9 nøyaktig én gang § Hver boks inneholder tallene

1–9 nøyaktig én gang • Det løste brettet fungerer som sertifikat. Hvis verifikasjonen lykkes, kan vi konkludere med at løsningen er korrekt. Kort sagt: Sertifikater gjør det mulig å raskt verifisere løsninger, selv om det å finne løsningen kan være vanskelig.

Kompleksitetsklassen NP Definisjon: Et avgjørelsesproblem tilhører NP dersom en gitt løsning kan verifiseres i polynomiell tid. • Alle problemer i P er også i NP, siden dersom man kan løse et problem raskt, kan man også verifisere løsningen raskt. • Forkortelsen NP står for Nondeterministic Polynomial time. Viktige punkter: • NP-problemer kan være mye vanskeligere å løse enn P-problemer, men løsningen kan verifiseres raskt. • Alle problemer i NP er minst like vanskelige som de i P. • Det er fortsatt ukjent om P = NP eller P ≠ NP – dette er et av de største uløste problemene i matematikk og informatikk. • Løsning av P vs NP-problemet gir en premie på én million dollar fra Clay Mathematics Institute. Intuisjon: • P-problemer: Kan løses raskt • NP-problemer: Løsning kan verifiseres raskt, men kan være vanskelig å finne

Oversettelse mellom problemer I problemløsning, både praktisk og teoretisk, er det ofte nyttig å oversette et problem til et annet problem som vi vet hvordan vi kan løse. • Dette gjelder spesielt i kompleksitetsteori, hvor vi ønsker å forstå hvor vanskelige problemer egentlig er. • Vi gjør dette systematisk gjennom reduksjoner, som er en metode for å transformere ett problem til et annet. • Reduksjoner gir oss en måte å sammenligne vanskelighetsgraden av ulike problemer: ○ Hvis vi kan redusere problem A til problem B, og B er lett å løse, kan vi dermed også løse A. ○ Hvis B er kjent vanskelig, indikerer det at A også er vanskelig.

Polynomtidsreduksjoner Definisjon: En polynomtidsreduksjon fra problem A til problem B er en algoritme som:

1. Tar en instans av A som input
2. Transformerer den til en instans av B i polynomiell tid
3. Bevarer svarene: ○ Hvis instansen av A har JA-svar, må den tilsvarende instansen av B også ha JA-svar ○ Hvis instansen av A har NEI-svar, må den tilsvarende instansen av B også ha NEI-svar Viktige poenger: • Reduksjoner gjør det mulig å overføre vanskelighetsgrad fra ett problem til et annet. • Alle avgjørelsesproblemer i P kan reduseres til hverandre i polynomiell tid, siden alle kan løses effektivt. • Polynomtidsreduksjoner er også kjernen i å definere NP-komplette problemer: ○ Et problem er NP-komplett hvis det tilhører NP og alle andre NP-problemer kan reduseres til det i polynomiell tid. Kort sagt: Reduksjoner er et verktøy for å måle og sammenligne hvor vanskelig problemer er, og de gir et formelt grunnlag for å vise at visse problemer er spesielt vanskelige å løse.

Polynomtidsreduksjoner – Enkelt eksempel: EVEN → ODD Vi kan illustrere polynomtidsreduksjoner med et veldig enkelt eksempel: Problem EVEN: • Instans: Et heltall n • Spørsmål: Er n et partall? Problem ODD: • Instans: Et heltall n • Spørsmål: Er n et oddetall? Reduksjon: • Vi kan transformere en instans av EVEN til en instans av ODD ved å gjøre: $n \mapsto n+1$ • Hvorfor er dette korrekt? ○ Hvis n er partall (JA-instans i EVEN), blir n+1 oddetall (JA-instans i ODD) ○ Hvis n er ikke-partall (NEI i EVEN), blir n+1 ikke-oddetall (NEI i ODD) Konklusjon: • Dette viser hvordan et problem kan oversettes til et annet problem på en systematisk og korrekt måte i polynomiell tid.

Polynomtidsreduksjoner – Kompleks eksempel: SORT ↔ SCC-k Problem SORT: • Instans: To arrayer A1 og A2 • Spørsmål: Består A2 av de samme elementene som A1 i sortert rekkefølge? Problem SCC-k (Sterkt Sammenhengende Komponenter): • Instans: En graf G og et tall k • Spørsmål: Har G minst k sterkt sammenhengende komponenter? Reduksjon: • Alle avgjørelsesproblemer i P kan polynomtidsreduseres til hverandre, fordi de kan løses i polynomiell tid. • For eksempel kan SCC-k reduseres til SORT slik: 1. Løs

SCC-k(G, k) med en polynomiell algoritme 2. Oversett resultatet til instanser av SORT: § Hvis SCC-k(G, k) = JA → sett A1=[1,2,3],A2=[1,2,3] § Hvis SCC-k(G, k) = NEI → sett A1=[1,2,3],A2=[2,1,3] Poeng: • Denne prosessen illustrerer hvordan polynomtidsreduksjoner bevarer svaret (JA/NEI) og kan brukes til å sammenligne vanskelighetsgrad mellom problemer. • Selv om dette eksemplet virker kunstig, er prinsippet det samme for mye mer komplekse problemer som NP-komplette problemer.

De vanskeligste problemene i NP NP-harde problemer er de problemene som er minst like vanskelige som alle andre problemer i NP. • Hvis du kan løse et NP-hardt problem i polynomiell tid, kan du i prinsippet løse alle NP-problemer i polynomiell tid. • Det første problemet som ble bevist å være NP-hardt var SAT (Satisfiability-problemet). • Hvordan vise at et problem A er NP-hardt: 1. Ta et kjent NP-hardt problem, f.eks. SAT 2. Vis at dette problemet kan reduseres i polynomiell tid til A 3. Hvis reduksjonen er korrekt, er A minst like vanskelig som SAT, og dermed NP-hardt Kort sagt: NP-harde problemer representerer den øvre grensen av vanskelige problemer i NP, selv om det ikke nødvendigvis er sikkert at de tilhører NP.

NP-komplette problemer Et problem A er NP-komplett hvis det oppfyller to kriterier:

1. A er i NP: ○ Løsningen kan verifiseres i polynomiell tid
2. A er NP-hardt: ○ Alle problemer i NP kan polynomtidsreduseres til A Viktige punkter om NP-komplette problemer: • De er antatt vanskelige å løse, men løsninger er lette å verifisere. • NP-komplette problemer fungerer som referanseproblemer for vanskelighet: ○ Hvis du kan løse ett NP-komplett problem raskt, kan du løse alle NP-problemer raskt • De har også en viktig rolle i kryptografi, fordi sikkerheten til mange kryptosystemer bygger på antakelsen om at visse NP-komplette eller NP-harde problemer ikke kan løses effektivt. Eksempel på NP-komplette problemer: • SAT (Boolean Satisfiability) • 3-SAT • Hamiltonian Path • Traveling Salesman Problem (TSP, beslutningsversjon) Kort sagt: NP-komplette problemer er både verifiserbare og minst like vanskelige som alle NP-problemer, og de utgjør kjernen i studiet av problemers kompleksitet.

Men er P = NP? Spørsmålet P vs NP er et av de mest kjente og uløste problemene i datavitenskap: • Hvis ett NP-komplett problem kan løses i polynomiell tid, betyr det at alle NP-problemer kan løses i polynomiell tid. ○ Dette ville bevise at P = NP. • Konsekvensene av en slik løsning ville vært enorme: ○ Kryptografi som baserer seg på vanskeligheten av visse NP-problemer ville bli utryddet. ○ Mange komplekse optimeringsproblemer som i dag er praktisk uløselige ville kunne løses raskt. • Selv om dette teoretisk er mulig, finnes det ingen kjent polynomiell algoritme for NP-komplette problemer, og de fleste forskere antar at P ≠ NP, selv om det ikke er bevist. Kort sagt: Å løse P vs NP ville revolusjonere hele datavitenskapen.

Beregnbarhet Et annet fundamentalt spørsmål i teoretisk informatikk er beregnbarhet: • Spørsmålet: Finnes det problemer som ikke kan løses, uansett hvor mye tid eller ressurser vi har? • Dette var et spørsmål som David Hilbert stilte tidlig på 1900-tallet som en del av sine berømte matematiske utfordringer. Løsningen kom fra Alonzo Church og Alan Turing i 1936: • De viste at det finnes uløselige problemer, altså problemer som ikke kan beregnes med noen algoritme. • Eksempel på et uløselig problem: Halting-problemet ○ Spørsmålet: “Vil et vilkårlig dataprogram stoppe, eller kjøre for alltid?” ○ Det finnes ingen generell algoritme som kan svare korrekt på alle mulige programmer og input. Konklusjon: • Selv i teorien finnes det grenser for hva som kan beregnes. • Kompleksitetsteori (P, NP, NP-komplett) studerer hvor raskt problemer kan løses, mens beregnbarhet handler om om de i det hele tatt kan løses.

Stoppeproblemet er uløselig Stoppeproblemet (Halting Problem) er det mest grunnleggende eksemplet på et uløselig problem i informatikk. • Formulert som et beslutningsproblem: ○ Instans: En algoritme A og en input x til A ○ Spørsmål: Terminerer A når den kjøres med input x? (JA/NEI) • Stoppeproblemet er uløselig, det finnes ingen algoritme som kan gi korrekt svar for alle mulige instanser. • Beviset gjøres vanligvis gjennom et motsigelsesbevis (proof by contradiction).

Motsigelsesbevis for stoppeproblemet La oss anta motsetningsvis at det finnes en algoritme H som løser stoppeproblemet:

1. H(A,x) returnerer JA hvis algoritme A terminerer på input x, og NEI ellers.
2. Konstruer en ny algoritme D som tar en algoritme A som input: ○ Sjekk H(A,A), altså om A terminerer når den kjører med seg selv som input. ○ Hvis H(A,A)=JA: gå inn i en evig løkke (terminerer aldri) ○ Hvis H(A,A)=NEI: terminer med JA
3. Kjør nå D(D), altså algoritmen D med seg selv som input: ○ Hvis H(D,D)=JA (D(D) terminerer), så går D(D) inn i en evig løkke → kontradiksjon ○ Hvis H(D,D)=NEI (D(D) terminerer ikke), så terminerer D(D) med JA → kontradiksjon Konklusjon: • Antagelsen om at H eksisterer fører til en motsigelse. • Dermed kan ingen algoritme løse stoppeproblemet. • Stoppeproblemet er et fundamentalt eksempel på uløselige problemer: det finnes problemer som ikke kan løses algoritmisk, uansett hvor mye tid eller ressurser man har.