TDT4120 Algoritmer og datastrukturer

Eksamen, 7. august 2020, 09:00–13:00

Faglig kontakt	Magnus Lie Hetland
Hjelpemiddelkode	A

Løsningsforslag

Løsningsforslagene i rødt nedenfor er *eksempler* på svar som vil gi full uttelling. Det vil ofte være helt akseptabelt med mange andre, beslektede svar, spesielt der det bes om en forklaring eller lignende. Om du svarte noe litt annet, betyr ikke det nødvendigvis at du svarte feil!

På grunn av koronapandemien er dette en hjemmeeksamen med alle hjelpemidler tillatt og med karakteruttrykk *bestått/ikke bestått*. Derfor er det ikke lagt vekt på å skille mellom prestasjoner i det øvre sjiktet av karakterskalaen, og utvalget av oppgavetyper er noe utenom det vanlige. Følgelig bør ikke eksamenssettet ses som representativt for ordinære eksamener.

1 Du har en tabell (*array*) med n heltall og skal finne de k største tallene. Noen løsninger vil ha lavere asymptotisk kjøretid mens andre vil være lettere å implementere, og ha lavere overhead på grunn av enklere datastrukturer, for eksempel $O(kn)$, $O(n \lg n)$, $O(n \lg k)$ og $O(n)$. Si litt om hvordan de fungerer, og hvilke fordeler og ulemper de har. Hvilke av løsningene dine vil fungere om du vil unngå ekstra minnebruk, og utføre operasjonene *in-place*, ved å bytte om på posisjonene til elementene? Hvilken av disse *in-place*-løsningene ville du ha brukt i en faktisk implementasjon?

Forklar og utdyp. Knytt til relevant teori, gjerne i ulike deler av pensum.

Her kan man f.eks. diskutere følgende løsninger:

1. Gå gjennom tabellen og ha en annen tabell med k kandidater. Sammenlign hver verdi x med de k kandidatene, og bytt ut den minste, om den er mindre enn x: $O(kn)$
2. Sorter tabellen med HEAPSORT eller MERGE-SORT (ev. QUICKSORT, selv om man da ikke har samme garanti i verste tilfelle) og velg de k siste elementene: $O(n \lg n)$
3. Samme som 1, men bruk en min-haug (*min-heap*) til å ta vare på de k kandidatene: $O(n \lg k)$

4. Bruk SELECT (ev. RANDOMIZED-SELECT, selv om man da ikke har samme garanti i verste tilfelle) med parameter $n-k+1$ og velg de k siste elementene: $O(n)$

Man kan også bruke et binært søketre i stedet for en haug, men bør da diskutere kjøretid i verste tilfelle (at man kan få lineær kjøretid, eller at det er mulig å balansere slike trær så man kan garantere logaritmisk tid), og at de har en del overhead sammenlignet med hauger.

Diskusjonen rundt fordeler og ulemper kan inneholde mye forskjellig, og og mange varianter vil være fullgode om de viser en forståelse for hvordan metodene man har valgt fungerer. Et naturlig punkt kan være at SELECT er ganske omstendelig og mest av teoretisk interesse, og at ikke-deterministiske eller randomiserte utgaver som QUICKSORT (eller RANDOMIZED-QUICKSORT) og RANDOMIZED-SELECT kan være bedre enn deterministiske alternativer i praksis.

Av metodene over vil f.eks. HEAPSORT, QUICKSORT og SELECT (ev. randomiserte utgaver) fungere *in-place*. (Man kan også lage en *in-place*-versjon av MERGE-SORT, men man må i så fall forklare noe om hvordan.) Av disse er SELECT urealistisk i praksis, men sortering og RANDOMIZED-SELECT kan fungere.

Man kan også innføre en annen løsning, som vil kunne være en svært god løsning i praksis, der man utfører de k første trinnene av HEAPSORT, noe som vil ha kjøretid $O(n + k \lg n)$ og vil plassere de k største elementene sist, *in-place*.

Relevante læringsmål: Kunne konstruere nye effektive algoritmer; kunne definere *asymptotisk notasjon*; forstå MERGE-SORT; forstå QUICKSORT og RANDOMIZED-QUICKSORT; forstå RANDOMIZED-SELECT; kjenne til SELECT; forstå hvordan *heaps* fungerer, og hvordan de kan brukes som *prioritetskøer*; forstå HEAPSORT; forstå hvordan *binære søketrær* fungerer; vite at forventet høyde for et tilfeldig binært søketre er $\Theta(\lg n)$; vite at det finnes søketrær med garantert høyde på $\Theta(\lg n)$

2 En artikkel fra femtitallet beskriver to problemer med tilhørende algoritmer: Å koble sammen alle nodene i en graf ved hjelp av kanter med minst mulig totalvekt, og å koble sammen to gitte noder ved hjelp av kanter med minst mulig totalvekt. Diskuter forholdet mellom problemene, og hvordan de løses. Hvilke likheter og forskjeller ser du? Er det forskjeller i hvilke grafer de fungerer på? Argumenter kort for svaret ditt.

Forklar og utdyp. Knytt til relevant teori, gjerne i ulike deler av pensum.

Artikkelen det er snakk om er E. W. Dijkstras «A Note on Two Problems in Connexion with Graphs» (1959), der han beskriver det som vanligvis kalles Prims og Dijkstras algoritmer. Her er det altså meningen man skal si noe om minimale spenntrær og korteste-vei-problemet, og identifisere nær likhet mellom PRIM og DIJKSTRA, der forskjellen i hovedsak er prioriteten som brukes for å plukke ut neste node. En viktig forskjell i hvilke grafer de fungerer på er at DIJKSTRA ikke fungerer om man har negative kantvekter, i motsetning til PRIM. Man bruker også gjerne DIJKSTRA på rettede grafer, mens retning ikke gir mening for PRIM.

Relevante læringsmål: Vite hva *spenntrær* og *minimale spenntrær* er; forstå MST-KRUSKAL; forstå MST-PRIM; forstå ulike varianter av *korteste-vei-* eller *korteste-sti*-problemet; forstå DIJKSTRA

3 Når en algoritme utføres, gjentas gjerne et «trinn» mange ganger, til vi har funnet resultatet for en gitt instans. Hvordan relaterer dette seg til dekomponering av instansen i delinstanser (*subproblems)*? Hvordan arter dette seg forskjellig for ulike designmetoder? Hvilken rolle spiller matematisk induksjon oppi det hele? Hvilket slektskap har dekomponering til reduksjoner og hardhetsbevis, og hvordan stemmer forklaringen din overens med dette?

Forklar og utdyp. Knytt til relevant teori, gjerne i ulike deler av pensum.

Her er det stor frihet i hvordan man forklarer, men et kjernepoeng er at ett enkelt trinn (én iterasjon eller ett kall av en rekursiv funksjon) bygger en løsning basert på én eller flere del-løsninger, som er konstruert av tidligere trinn. Hvert trinn løser en delinstans.

Enkle, iterative algoritmer har gjerne en veldig enkel dekomponering, og hvert trinn baserer seg bare på delinstansen som ble løst av forrige trinn (jf. INSERTION-SORT), men i de viktigste designmetodene i pensum (splitt og hersk, dynamisk programmering og grådighet) har hvert trinn flere delinstanser de kan bygge på. For splitt og hersk er disse delløsningene helt uavhengige, og dekomponeringen utgjør en trestruktur. For dynamisk programmering overlapper de, og man har en rettet asyklisk graf, der man mellomlagrer delløsninger for å unngå eksponentiell kjøretid. For grådighet har man også flere delinstanser, men man gjør et lokalt valg og løser kun én av dem, som man så bygger videre på.

Matematisk induksjon er en måte å bevise korrekthet på, eller å vise at designmetodene fungerer: Man viser at grunntilfellene stemmer, og så antar man (induktiv hypotese) at delløsningene er korrekte, og viser at man så kan bygge en løsning ut fra disse. Alle løsninger som så kan bygges på denne måten vil da være korrekte.

Man kan se på dekomponering som en reduksjon til flere problemer, der

vanlig reduksjon er spesialtilfellet der man reduserer til kun ett (og man krever at instansen man konstruerer f.eks. er mindre). I begge tilfeller er det man reduserer eller dekomponerer til noe man bruker til å løse det opprinnelige problemet.

Hardhetsbevis kan man så se på som å dekomponere fra et opprinnelig problem til et delproblem som utgjør nøkkelen til å løse det opprinnelige problemet. Om man ikke kan løse det opprinnelige problemet, så kan man umulig løse dette delproblemet. (Her er det viktig at man ikke bytter om på rollene eller retningen.)

Relevante læringsmål: Forstå *løkkeinvarianter* og *induksjon*; forstå *rekursiv dekomponering* og *induksjon over delinstanser*; forstå designmetoden *divide-and-conquer* (*splitt og hersk*); forstå ideen om en *delproblemgraf*; forstå designmetoden *dynamisk programmering*; forstå løsning ved *memoisering* (*top-down*); forstå løsning ved *iterasjon* (*bottom-up*); forstå hva *optimal delstruktur* er; forstå hva *overlappende delinstanser* er; forstå designmetoden *grådighet*

4 Din venn Lurvik mener han har funnet på en ny algoritme for å finne korteste vei i vektede, rettede grafer, der vektene er positive heltall. Ideen hans er å transformere kanter (u, v) med vekt $w(u, v) = k > 1$ til stier $\langle u, x_1, x_2, \ldots, x_{k-1}, v \rangle$ med lengde k, og så bruke BFS til å finne korteste vei. Hvilke fordeler eller ulemper har denne metoden? Diskuter slektskap med algoritmer i pensum. Kunne du ha gjort noe lignende for å finne maksimal flyt med heltallskapasiteter? Hva slags algoritme måtte du i så fall ha hatt for å ta BFS sin plass?

Forklar og utdyp. Knytt til relevant teori, gjerne i ulike deler av pensum.

Noen fordeler og ulemper kan være subjektive, f.eks. at det kan være en enklere algoritme (enn f.eks. Dijkstra), eller at det er mer knot å holde styr på en slik oppsplitting. Den viktigste ulempen er likevel at algoritmen generelt vil ha *eksponentiell kjøretid*, som funksjon av den opprinnelige inputstørrelsen, siden vektene kan være eksponentielt store, og man dermed kan få eksponentielt mange noder i den nye grafen.

Et naturlig slektskap å påpeke er det til Dijkstra, siden algoritmene vil oppføre seg «nesten likt», på mange vis. Man kan si at Dijkstra simulerer atferden til Lurviks algoritme, uten å faktisk lage den nye grafen; dvs., nodene (fra den opprinnelige grafen) vil besøkes i samme rekkefølge, dvs., i avstandsrekkefølge. Der Dijkstra bruker en prioritetskø og Relax-operasjonen til å avgjøre hvordan beregningen skal spre seg i grafen, så utfører Lurviks algoritme denne spredningen direkte i grafen, der lange kanter automatisk tar lengre tid.

For å gjøre noe lignende med flyt, måtte man bytte ut en kant med kapasi-

tet k med k nye kanter (i parallell). For å tilfredstille lærebokas krav, måtte også $k-1$ av disse splittes med en ny node (men dette er ikke essensielt). Det man måtte ha for å ta BFS sin plass er en algoritme som kan finne det største antall disjunkte stier fra kilde s til sluk t.

Relevante læringsmål: Forstå hvorfor løsningen vår på 0-1-ryggsekk-problemet *ikke er polynomisk*; forstå BFS, også for å finne *korteste vei uten vekter*; forstå DIJKSTRA; kunne håndtere *antiparallelle kanter* (her anvendt på parallelle)

5 I grunnleggende kompleksitetsteori bruker vi gjerne NP som et slags «univers» for problemene vi ser på. Diskuter fordeler og ulemper med dette. Gi noen argumenter for at det er en stor klasse. Gi også eksempler på typer problemer som *ikke* er i NP, og som likevel har stor praktisk betydning. Gitt hva vi gjerne bruker klassen NP til, hvorfor er ikke dette så problematisk?

Forklar og utdyp. Knytt til relevant teori, gjerne i ulike deler av pensum.

Her er det åpent for mange diskusjoner av fordeler og ulemper, som f.eks. at det gjør det mulig å studere problemene som formelle språk, men at det er begrensende å kun se på binærstrenger. Argumenter for at det er en stor klasse bør inneholde en skisse av definisjonen, dvs., at det er mengden av språk (eller beslutningsproblemer) som har polynomiske verifikasjons-algoritmer, og at dette gjelder alle beslutningsproblemer der vi i praksis kan vise at en løsning er korrekt, o.l. Et viktig poeng er at klassen ikke inneholder f.eks. optimeringsproblemer, men siden vi stort sett bruker NP som et univers for hardhetsbevis, og vi fortsatt kan vise NP-hardhet for optimeringsproblemer (ev. at vi kan bruke optimeringsutgaver til å løse NP-komplette beslutningsproblemer), så kan hardhetsbevisene våre likevel dekke mange problemer som ikke er i NP.

Relevante læringsmål: Forstå sammenhengen mellom *optimerings-* og *beslutnings-problemer*; forstå definisjonen av klassene P, NP og co-NP; forstå definisjonen av NP-*hardhet* og NP-*kompletthet*