Oversikt over algoritmer og metoder i TDT4120

Morten Fyhn Amundsen

NTNU

31. august 2016

1 Sortering (av lister)

Insertion sort Anse første element som sortert. Ta første usorterte element og SWAP til venstre til det er på riktig sted i den sorterte delmengden. Gjør for alle elementer. Worst og average: $O(n^2)$. Rask på små mengder (lite overhead), og mengder som allerede er delvis sorterte.

Heapsort Bygg en min-/max-heap av data. Popp elementer fra toppen. Elementene havner da i rekkefølge. Worst og average: $O(n \log n)$. Å bygge heapen er $O(n \log n)$ (kan gjøres lurere). Å poppe alle elementene er også $O(n \log n)$. In place.

Quicksort Velg en pivot. Kjør PARTITION: Legg alle høyere verdier over pivoten, alle lavere under. Nå er pivoten på riktig plass. Sorter så delmengdene rekursivt. Worst: $O(n^2)$. Average: $O(n \log n)$.

Counting sort Veldig rask på relativt lave heltall. Lag tabell over antall elementer under eller lik hvert tall. Bruk tabellen til å generere en sortert liste. $O(n + k)$ der k er maksverdi.

Bucket sort God på jevnt fordelte data. Del intervallet $[0, 1)$ i n like store subintervaller (bøtter), fordel elementene i bøttene. Sorter hver bøtte for seg og sett sammen. Average: $O(n)$. Worst: $O(n^2)$. Alternativ: Kjør insertion sort på hele greia etter å fordele i bøtter.

Radix sort Sorter *stabilt* etter minst (evt. mest) signifikante siffer. Gjenta for så mange siffer maksverdien har. Worst: $O(dn)$, der d er antall siffer. Radix sort benytter vanligis bucket eller counting sort for hver «runde».

2 Graftraversering/-sortering

Binærtrær Kan traverseres pre-, in- eller postorder:

Preorder Besøk rot $\rightarrow$ traverser venstre subtre $\rightarrow$ traverser høyre subtre.

Inorder Traverser venstre subtre → besøk rot → traverser høyre subtre.

Postorder Traverser venstre subtre → traverser høyre subtre → besøk rot.

BFS Legg rotnoden i en kø (FIFO). Dequeue en node og se på den. Legg til dens (ubesøkte) naboer i køen. Dequeue neste og gjenta til riktig node funnet eller kø tom. Worst: $O(|E|)$.

DFS Start med rotnoden. Legg alle naboer i en stakk (LIFO). Gå til neste node i stakken. Legg til dens (ubesøkte) naboer i køen. Gjenta til ferdig. Worst: $O(|E|)$.

Topologisk sortering Kun mulig på en DAG. Kjør DFS, og sett noder inn i en liste etter hvert som de er ferdigbehandlede (les: ikke besøkte). $O(|V| + |E|)$.

3 Minimale spenntrær

Prims algoritme (Grådig.) Velg vilkårlig startnode, anse den som et tre. Utvid treet med den minste kanten som leder fra treet til en ny node. Gjenta til alle noder er med i treet.

Kruskals algoritme (Grådig.) Lag en skog der hver node er et eget tre. Lag en mengde av alle kanter. Så lenge skogen ikke er sammenhengende og det fremdeles fins kanter: Ta den minste kanten fra mengden. Legg den til i skogen om den sammenkopler to trær. $O(|E| \log |V|)$

4 Korteste vei (én til alle)

Relax Forutsetter en liste over lengden på hittil beste vei til hver node ($v.d$) og hvilken forrige node man i så fall må komme fra ($v.\pi$). Relax sjekker om kanten fra u til v gir en forbedring, og oppdaterer i så fall estimatene: $v.d = u.d + w(u, v)$ og $v.\pi = u$.

Bellmann–Ford Takler negative kanter. Kan også avsløre om det fins negative sykler (i så fall ingen løsning). Gjenta $|V| - 1$ ganger: Kjør Relax på hver kant i grafen. Hvis nå $v.d > u.d + w(u, v)$ stemmer for minst én kant fins det negative sykler. Worst: $O(|V||E|)$.

DAG shortest path For hver node i topologisk sortert rekkefølge: Kjør Relax på hver kant til en nabo. (Kan også løse longest path.)

Dijkstras algoritme Takler ikke negative kanter. Worst: $O(|E| + |V| \log |V|)$.

1. Gi hver node et foreløpig avstandsestimat $v.d$ fra startnoden. (0 for startnoden, ∞ for resten.)
2. Legg alle noder i en menge for ubesøkte. Sett startnode til aktiv node.
3. Kjør Relax på alle kanter fra den aktive noden for å oppdatere estimater.
4. Fjern aktiv node fra ubesøkt-mengden.
5. Velg den ubesøkte noden med lavest estimat, og sett den til ny aktiv node. Gå til steg 3.

5 Korteste vei (alle til alle)

Floyd–Warshall Takler negative kanter. $D^{(k)}$ er en matrise over alle korteste veiestimater $v.d$ etter k iterasjoner. Sett $D^{(0)} = W$. For $k = 1 \dots n$: For $i = 1 \dots n$: For $j = 1 \dots n$: $D^{(k)}_{i,j} = \min\left(D^{(k-1)}_{i,j}, D^{(k-1)}_{i,k} + D^{(k-1)}_{k,j}\right)$. Gir kjøretid $\Theta(|V|^3)$. W er definert slik:

$$W_{i,j} = \begin{cases} 0 & \text{hvis } i = j \\ w(i,j) & \text{hvis } i \neq j \text{ og } (i,j) \in E \\ \infty & \text{hvis } i \neq j \text{ og } (i,j) \notin E \end{cases}$$

Når algoritmen er ferdig, kan man avsløre negative sykler ved å se om diagonalen av $D^{(n)}$ har negative verdier. Har den det, betyr det at en node kan gå til seg selv med negativ vektsum.

6 Maksflyt

Ford–Fulkerson Så lenge det fins en flytforøkende sti p: Øk flyten langs p.

Edmonds–Karp Variant av Ford–Faulkerson der BFS brukes for å finne flytforøkende sti p. Finner alltid den korteste flytforøkende stien (målt i antall kanter), og øker flyten langs den. $O(|V||E|^2)$.

7 Hashing

Direkteadressert tabell Bruker nøkkelen som tabellindeks. Funker bare med små nøkkelmengder. Garantert kollisjonsfri.

Hashtabell Bruker $h(k)$ som tabellindeks, der k er nøkkelen og h er en hash-funksjon. Minker størrelsen på tabellen, men kan gi kollisjoner. Viktig å velge en lur h.

Chaining Lagrer flere verdier som en lenket liste når det oppstår kollisjoner. (Dobbeltlenket er mye raskere enn enkeltlenket, og er de facto standard for chaining.)

Hash-funksjon 1: Divisjon $h(k) = k \mod n$. Tallet n kan godt være et primtall som er et stykke unna nærmeste potens av to.

Hash-funksjon 2: Multiplikasjon $h(k) = \lfloor m(kA \mod 1) \rfloor$: Gang nøkkelen med et tall $A \in (0,1)$, fjern alt foran kommaet, gang med m, og rund ned. Visse verdier A fungerer bedre enn andre.

Universell hashing Velger en hash-funksjon tilfeldig. Garanterer mot muligheten for konsekvent worst case-oppførsel.

Åpen adressering Maks ett element per tabellindeks (ingen chaining e.l.). Lagrer ingen pekere $\rightarrow$ kan lagre større tabell med like mye minne. Hash-funksjonen blir $h(k,i)$, hvor i er *probetallet* (starter som 0). Hvis $h(k,0)$ er opptatt, prøv $h(k,1)$ osv.

Lineær probing Har en vanlig hash-funksjon $h'(k)$ (her: *auxilliary hash function*). Lineær probing bruker da $h(k,i) = (h'(k) + i) \mod m$. Gir *primary clustering*, dvs. en tendens til at nøkler hoper seg opp, som gir tregere søking.

Kvadratisk probing $h(k,i) = (h'(k) + c_1 i + c_2 i^2) \mod m$. Gir *secondary clustering*, som ikke er like fælt.

Double hashing $h(k,i) = (h_1(k) + i h_2(k)) \mod m$. Kan komme nær «ideell» hashing.

8 Grådige algoritmer

Algoritmer som velger det lokalt «beste» for hvert delproblem. Raske, og funker på endel problemer.

Huffman-koding Gitt en mengde symboler med tilhørende forekomster: Lag en skog av symbolene. Finn de to med lavest forekomst, gjør de til barn av en ny node med summen av barna lagret som sin forekomst. Fortsett å finne symboler eller noder og sett sammen til det blir et sammenhengende tre. Stien til løvnoden for hvert symbol gir symbolets kode: Venstre = 0, høyre = 1 (f.eks.).

Activity selection Gitt en mengde aktiviteter som delvis overlapper i tid, finn en delmengde med maksimalt antall aktiviteter der ingen overlapper. Kan løses ved å alltid velge den neste aktiviteten som er først ferdig og som ikke overlapper.

9 Dynamisk programmering

Underproblemer er avhengige av hverandre. Løser og lagrer resultatet av alle nødvendige underproblemer for å løse hovedproblemet.

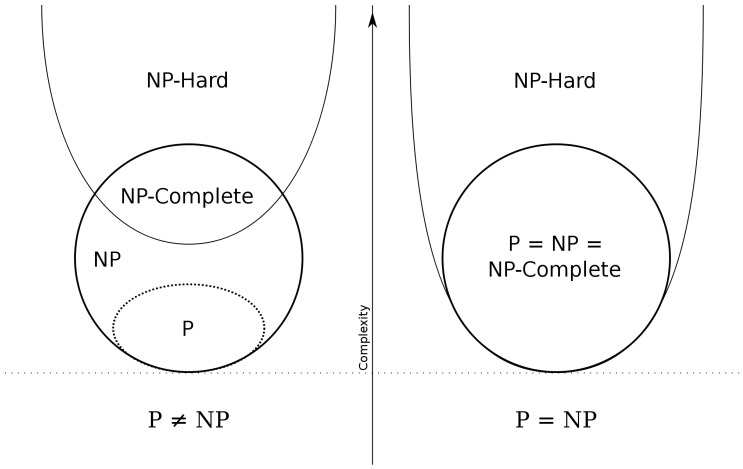
10 NP-kompletthet

P Problemer som kan løses i polynomisk tid $O(n^k)$.

NP Problemer der man kan sjekke om en løsning er korrekt i polynomisk tid. P $\in$ NP. Uvisst om P $=$ NP, men sannsynligvis ikke.

NP-hard Alt som er like vanskelig eller vanskeligere enn NP: (Alt i NP) $\leqslant$ (Alt i NP-hard)

NPC Problemer både i NP og NP-hard, eller: Et problem i NP som er like «vanskelig» som et hvilket som helst annet problem i NP. Om *ett* NPC-problem kan løses i polynomisk tid, så kan *alle* NP-problemer løses i polynomisk tid. (Alt i NP) $\leqslant$ (Alt i NPC). Formell definisjon: Et problem C er NP-komplett dersom: 1) C $\in$ NP, og 2) alt i NP kan reduseres til C i polynomisk tid.



Figur 1: Sammenhenger mellom kompleksitetskategorier

11 NPC-problemer

Knapsack Gitt en mengde gjenstander med tilordnet verdi og vekt, hvilken kombinasjon gir størst total verdi gitt en øvre vektgrense?

1-0 Knapsack Man kan kun ta med én eller ingen av hver gjenstand.

Subset-sum Gitt en mengde tall, finn en delmengde som summeres til 0. Spesialtilfelle av knapsack, kan mao. reduseres til knapsack: Subset-sum $\leqslant$ knapsack.

Vertex cover Finn et mengde noder slik at alle kanter i en graf grenser til minst én node i mengden.

Hamiltonian path En sti som besøker hver node presist én gang.

Travelling salesman Minimal Ham-cycle i en komplett, vektet graf. NP-hard, ikke NP-komplett.

CIRCUIT-SAT Finn en boolsk krets har et fast sett innganger som alltid gjør utgangen sann. Bevist å være NP-komplett.

SAT Som CIRCUIT-SAT, men med et matematisk boolsk uttrykk. CIRCUIT-SAT kan reduseres til SAT og vice versa.

Max clique En clique er en delgraf som er komplett. En max clique er den største cliquen i en graf. Merk at SAT $\leqslant$ CLIQUE. Eksempel: Finn grupper i et sosialt nettverk der alle kjenner hverandre.

12 Parallellprogrammering

Spawn Nøkkelord som gjør et prosedyrekall som *kan* kjøres parallelt.

Sync Nøkkelord som pauser kjøring til alle barn kalt med **spawn** er ferdige.

T_P Tiden en algoritme bruker når den kjøres på P prosessorer.

Work Work $= T_1$ er totalt arbeid, dvs. tiden algoritmen ville brukt på én prosessor.

Span Span $= T_\infty$ er den lengste serielle utregningen, dvs. tiden algoritmen ville brukt gitt ubegrenset mange prosessorer.

Speedup Speedup $= T_1/T_P$ er hvor mye raskere beregningen går på P kontra én prosessor.

Parallellitet Parallellitet $= T_1/T_\infty$ er et mål på i hvilken grad en beregning gjøres parallelt.

13 Masterteoremet

$$T(n) = aT\left(\frac{n}{b}\right) + n^c$$

1. $\log_b a < c \implies T(n) = \Theta(n^c)$
2. $\log_b a = c \implies T(n) = \Theta(n^c \log n)$
3. $\log_b a > c \implies T(n) = \Theta(n^{\log_b a})$