Notater til eksamen i Algoritmer og datastrukturer 2015

Sortering

Det finnes mange algoritmer for å sortere data. Noen er svært krevende for en datamaskin når mengden blir stor nok, mens andre slike algoritmer er svært effektive for tilsvarende store datamengder, forutsatt at implementasjonen foregår på rett måte. Et eksempel på en slik algoritme er ***innsettingssortering***, som sorterer elementer samtidig som elementene settes inn. Denne algoritmen er effektiv for små datamengder. En annen sorteringsalgoritme heter ***flettesortering*** (Merge Sort) og foregår ved at problemene deles inn i delproblemer av samme problem, i dette tilfelle å dele listen inn i 2 like store lister, for så å rekursivt sortere delproblemene og kombinere de ulike delproblemene. De to sorterte dellistene kombineres for å produsere det ferdige sorterte resultatet. **Heapsort** bygger en max-heap ut fra en usortert tabell med elementer, for så å fjerne det største elementet fra roten og sette det på korrekt plass ved å bytte det med det siste elementet. Deretter fjerner man det siste elementet fra heapen og ser til at det nye elementet i roten ligger på rett plass (ved hjelp av en funksjon, maxHeapify). Dette gjentas til det bare er ett element igjen.

Quicksort er en kraftig og populær sorteringsalgoritme. Den har verstefall kjøretid på $\Theta(n^2)$, men gjennomsnittlig kjøretid er $\Theta(n \log n)$. Quicksort fungerer slik at den plukker ut det siste elementet i listen som en pivotelement. Deretter flytter den de andre elementene slik at de som er mindre enn eller lik pivotelementet kommer til venstre, og de som er høyrere, kommer til høyre. Denne prosedyren kalles *partition*. Når denne partition-metoden er over returneres det et heltall. Dette heltallet brukes så til å kalle Quicksort rekursivt, slik: quicksort(A, p, q-1) og quicksort(A,q+1, r).q er det returnerte heltallet, A er en tabell med heltall og, p og q er startelementet og lengden på tabellen. Verstefalls kjøretid for denne algoritmen er når elementene enten er sorterte eller omvendt sorterte. Bestefalls kjøretid er når rekursjonstreet er balansert. En annen variant av Quicksort heter ***randomizedQuicksort***. Den fungerer akkurat som «vanlig» quicksort, men partisionsfunksjonen bruker nå et tilfeldig valgt pivotelement istedenfor å alltid bruke siste element. **Tellesortering (*countingSort)*** er en algoritme som fungerer på det premisset at tallene som skal sorteres, er heltall mellom 0 og n, id est 0, 1, 2, …, n. Denne algoritmen fungerer slik at den teller seg frem til hvilken posisjon hvert element må ha i en *utenhetstabell*. Dette er en såkalt stabil sorteringsalgoritme fordi den ivaretar relativ posisjonering mellom like elementer. Hvor stor effektiviteten til tellesorteringa er, kommer an på størrelsen til n, id est hvor store tallene kan være. Størrelsen på tabellen er det som *kan* skape problemer: Hvis vi snakker om 32-biters verdier, ville C vært $4*2^{32}$ som er ca. 17 TB stor dersom vi bruker 4 bytes per verdi. Mao, ikke veldig effektivt. Bruker vi 16-bits verdier, blir ting vesentlig bedre, da størrelsen på tabellen bare blir $4*2^{16}$, som er ca. 262MB. 8-biters verdier ville bare krevd en størrelse på $4*2^8$ = 1 MB, men her må vi heller passe på størrelsen til n. Bruker vi 4-bits, blir n såpass liten at man heller kan spørre seg om innsettingssortering er et bedre alternativ.

Radikssortering fungerer slik at man sorterer tallene ut fra de ulike sifrene i hvert tall, hvor man begynner med den med lavest verdi og jobber oss om den største. Det vil si at vi jobber oss fra høyre mot venstre. (Her har vi grunnlaget for de første millionene IBM tjente.) Radikssortering krever en stabil sorteringsalgoritme for å sortere på hvert siffer. Her er tellesortering et vanlig valg. I så fall blir den totale kjøretiden til radikssortering $\Theta(d(n+k))$. Hvis $k = O(n)$, så er radikssortering $\Theta(d*n)$. Effektiv kjøring av radikssortering avhenger av å finne et effektivt antall siffer. n elementer som skal sorteres, b bits per element. Man deler opp tallene/elementene i r-bit sifre. Da får vi d = [b/r9] siffer og 2^r-1 mulige verdier. E.g.: 32-bit tall, 8-bit sifre. Da får vi b=32, r=8, d=32/8=4 og k=2^8-1=255. Kjøretiden blir da $\Theta(b/r(n+2^r))$ og fornuftig r må velges, f.eks. r=log n. **Bøttesortering** er en lineær sorteringsalgoritme for flyttall. Den fungerer ved å anta at

input er uniformt fordelt over intervallet (0,1): Den deler opp 0,1 i n like deler, fordeler de ulike inndatatallene i de tilhørende delene, sorterer hver del med innsettingssortering og går til slutt gjennom hver del i rekkefølge og setter sammen resultatet.

Algoritme	Gjennomsnittlig kjøretid	Verstefall kjøretid
Innsettingssortering	$\Theta(n^2)$	$\Theta(n^2)$
Flettesortering	$\Theta(n \log n)$	$\Theta(n \log n)$
Heapsort	$O(n \log n)$	---
Quicksort	$\Theta(n^2)$	$\Theta(n \log n)$
Tellesortering	$\Theta(n + k)$	$\Theta(n + k)$
Radikssortering	$\Theta(d(n+k))$	$\Theta(d(n+k))$
Bøttesortering	$\Theta(n^2)$	$\Theta(n)$

Datastrukturer

Dynamiske sett er datastrukturer som ofte kan utføre flere av disse operasjonene:
Search(S,k): returnerer en peker med nøkkel k i sett S eller NIL hvis den ikke finnes.
Insert(S,x): setter inn element x i settet S.
Delete(S,x): sletter element x i settet S.
Minimum(S) returnerer elementet med minste nøkkel i settet S.
Maximum(S) returnerer elementet med største nøkkel i settet S.
Successor(S,x) returnerer det elementet som kommer etter x, sortert i henhold til nøkkelen til x.
Predecessor(S,x) returnerer det elementet som kommer før x, sortert i henhold til nøkkelen til x.
Direkteadresserte tabeller kan enkelt brukes til å implementere dictionary-funksjonene. Hvert element x har en nøkkel i universet U = {0,1,2, …, n.1} hvor n er relativt liten, id est vi har ganske få nøkler. Hver nøkkel må være unik, dvs. ingen elementer har samme nøkkel. Direkteadresserte tabeller representeres av en tabell T[0, 1, 2, …, n-1]. Hver plass (indeks) fungerer som en nøkkel i U. Finnes det et element x med nøkkel k i settet, vil, T[k] inneholde en peker til elementet. Hvis ikke, vil T[k] være tom. Direkteadresserte tabeller bruker 3 dictionary-funksjoner: Insert(x): T[key[x]] = x, search(k): return k og delete(k) T[key[x]] = nil. Alle operasjoner tar konstant tid.
Problemet med direkteadresserte tabeller er at universet U kan bli ganske stort, altså at det er mange nøkler. I mange tilfeller blir |U| så stor at en tabell med plass til hver nøkkel blir upraktisk eller helt umulig. Det som ofte er tilfellet, er at antall nøkler som faktisk lagres er liten, hvilket kaster bort unødvendig mye plass dersom man bruker direkteadresserte tabeller.
Da kan **hashtabeller** være et godt alternativ: *) Når settet med faktiske nøkler er mye mindre enn universet U med alle mulige nøkler. *) Kan redusere lagringsbehovet til $\Theta(|K|)$. *) Kan få O(1) i gjennomsnitt på operasjoner, men ikke i verstefall. Ideen er at man tar i bruk en funksjon h og lagrer elementet på plass h(k). Vi må fortsatt bruke en tabell T[0, 1, 2, …, n-1], men nå er ikke n antall mulige nøkler, men heller antall ledige plasser.
Vi kaller h for en **hash-funksjon**. h: U → {0, 1, 2, …, n-1}, dvs. h er en funksjon som mapper fra elementene i U til en gyldig plass i T. Da sier vi at k *hashes* til h(k). Man kan komme borti tilfeller der 2 nøkler får samme plass i hashtabellen (kollisjon). Dette kan forekomme hvis h er en dårlig hashfunksjon, eller hvis |U| > m (hvilket er vanlig når man bruker hashtabeller istedenfor direkte adressering). En kollisjon må forekomme dersom |K| > n, men ikke nødvendigvis hvi |K| <= n.
For å løse problemet med kollisjoner er det to alternativer: Såkalt chaining (med lenkede lister) eller åpen adressering. *Chaining* er en metode som tar i bruk lenkede lister for å løse problemet med hashkollisjoner. T |j| inneholder en peker til en lenket liste med alle elementer med h(k) = j. Hvis plassen er tom, er T|j| nil. Av de to alternativene er chaining det som tar størst plass, men det er også det som fungerer best. En god hashfunksjon er en funksjon som i prinsippet er uniform, dvs. den skal se til at elementene fordeles jevnt utover tabellen. Vanligvis er ikke dette mulig, da vi ikke vet noe om distribusjonsnøklene og hvorvidt de er uavhengige trukket eller ikke.

Kjøretiden for innsetting er Θ(1), for søking (gjennomsnitt) O(1), og sletting Θ(1). Et alternativ til «chaining» er åpen adressering. Ideen går ut på å lagre nøklene i tabellen direkte istedenfor i lenkede lister. Dette gjør man ved å bruke såkalte probes for å sjekke om plassen er ledig eller ikke. For å gjøre dette, må man utvide hash-funksjonen slik at den kan returnere en sekvens av plasser å utføre «probes» på. Da får vi en funksjon h(k, i) hvor k er nøkkelen som skal hashes og i ∈ {0, 1, 2, …, n-1}. Sekvensen av posisjonene som sjekkes må være en permutasjon av {0, 1, 2, …, n-1} slik at vi er innom alle posisjoner. Dictioary-funksjonenes kjøretid avhenger av hvor mange probes som må utføres før vi finner elementet det søkes etter, eller før vi kommer til ledig plass (hvis innsetting)

Stakker er dynamiske sett som følger en LIFO-politikk, hvilket vil si at siste element som settes inn, er det første som tas ut. En stakk kan implementeres som en vanlig tabell, men den har også et attributt som peker på det siste elementet som ble satt inn. Innsetting og sletting fra stakker kalles for push og pop.

Køer er dynamiske sett som følger en FIFO-politikk, der første element som setes inn, er det første elementet som skal tas ut. En kø har to attributter som ivaretas, kø.hode som peker på det første elementet i køen, og kø.hale som peker på den første ledige plassen etter det siste elementet som er satt inn. Innsetting og sletting fra køer kalles for enqueue og dequeue.

Lenkede lister er dynamiske sett som er ordnet i en lineær rekkefølge som tabeller. Men i motsetning til vanlige tabeller, hvor rekkefølgen er styrt av indeksen i tabellen, så er rekkefølgen i lenkede lister styrt av en peker i hvert element. Enkeltlenkede lister ivaretar ett attributt, x.next på hvert element x. Dobbeltlenkede lister ivaretar *to* attributter per element x: x.next og x.prev. En lenket liste kan være både sorterte og usorterte. En listes head-attributt peker på første element i listen, dersom denne finnes, og på NIL hvis den er tom.

Binære søketrær er datastrukturer som støtter mange dynamiske sett med operasjoner. Disse kan brukes både som såkalt dictionary og prioritetskø. Kjøretiden til operasjonene er proporsjonal med høyden til treet: beste fall Θ(log n), verste fall Θ(n). Det finnes flere typer søketrær: Binære søketrær, rødsvarte trær og B-trær for å nevne 3.

Binære søketrær er en viktig datastruktur. De kan implementeres med O(h) kjøretid, der h er treets høyde. Et binært søketre representeres med et sett av noder som er koblet sammen med pekere. Hver node inneholder disse egenskapene: En nøkkel/verdi, peker til venstre barn, peker til høyre barn og en peker til foreldrenoden. I et slikt tre peker tre.rot til rotnoden og tre.root.forelder = NIL. Det skiller seg fra et vanlig binærtre ved at et binært søketre er at hvis y er i x sitt venstre subtre, så må y.verdi <= x.verdi. Hvis y er i x sitt høyretre, må y.verdi være > x.verdi. Dette gjør at man kan skrive ut elementer i sortert rekkefølge med en funksjon som skriver ut nodene inOrder (venstre-rot-høyre). Verstefall for høyden i et binært søketre er O(n).

Rød-svarte trær er akkurat som et vanlig binært søketre, men med noen vesentlige forskjeller: et rød-svart tre består av to farger: rød og svart. I tillegg er dette et selvbalanserende tre, hvilket betyr at det skjer en balansering (rotasjon) når noder settes inn eller slettes fra treet. Dette fører til at høyden på treet alltid er O(log n) og tiden for operasjonene også tar O(log n). Rød-svarte trær har følgende egenskaper: Hver node er enten rød eller svart, rotnoden er svart, hver eneste bladnode (T.nil) er svart (og tom), hvis en node er rød, er begge bladnodene svarte (aldri to røde på rad i en enkel sti fra roten til en bladnode), for alle noder inneholder hver eneste sti til dens bladnoder samme antall svarte noder. Ved innsetting eller sletting av noder, må treet roteres for å opprettholde nevnte egenskaper. Kjøretiden for en operasjon er O(1).

B-trær er balanserte trær som er designet for å kunne fungere bra på sekundære lagringsmedium, som f.eks. en harddisk. Et slikt tre kan minne om et rød-svart tre, men er spesialtilpasset for å minimere antall I/O-operasjoner. Høyden i et b-tre vil alltid være O(log n). De kan ha flere enn to barn (i motsetning til rød-svarte trær og binære søketrær). De har derfor større branchfaktor. Dette gjør at den nøyaktige høyden til en node vil være vesentlig mindre enn i et rød-svart tre, hvilket også gjør at basen i logaritmen som beskriver høyden i b-treet vil være større enn 2. En intern node x i et B-tre inneholder x.n nøkler som fungerer som skiller

mellom x.n + 1 barn. Nøklene i x definerer intervallet som blir håndtert av x og de deler opp intervallet i delintervallet som blir håndtert av barna til x. Dette gjør at når vi søker i et B-tre, gjør vi et (x.n+1)-veis valg i hver node, noe som tar oss nærmere og nærmere nøkkelen vi leter etter.

Grafalgoritmer

Regnes som en viktig del av informatikken, da problemer som har med grafer å gjøre, ofte er ting som går igjen. En graf kan enten være urettet eller rettet, og den kan representeres på to måter: Nabolister eller nabomatriser. **Nabolister** bruker en lenket liste for å holde på alle naboene til node V i grafen. Vi bruker en tabell med |V| lister slik at hver node får hver sin egen liste. Node u sin liste skal da inneholde alle nodene v, slik at (u, v) ∈ E (leses som «u og v» er elementer i E»), id est den inneholder alle nodene som er forbundet til u med en kant. Dette fungerer både med rettede og urettede grafer. En grafrepresentasjon som bruker nabolister, vil trenge Θ(V+E) plass i minnet. Dette er å foretrekke så lenge |E| er liten i forhold til $|V|^2$. Dette gir oss en effektiv representasjon for en graf med «få kanter». Ettersom at kantene er lagret i en lenket liste, må man gjøre lineære søk for å kunne finne en kant. Det å finne alle naboene til u vil at Θ(grad(u)) tid, og det vil ta O(grad(u)) tid å finne ut om en kant (u, v) ∈ E. «grader» brukes her til å angi antall kanter fra u. En **nabomatrise** bruker en todimensjonal tabell som kobler nodene sammen. Vi bruker da en V*V matrise. En nabomatrise krever mer plass enn nabolister, men i noen situasjoner kan de være raskere. I minnet trenger vi $\Theta(V^2)$ plass. I tillegg må vi kjøre en løkke gjennom alle noder for å finne nodene som er naboer til u, dvs. en kjøretid på Θ(V). Vi trenger ikke å søke for å bedømme om en kant (u, v) eksisterer eller ikke.

Bredde-først-søk: Input er en graf G = (V, E) og en node s ∈ V som angir startnoden for søket. Output er v.d som angir minste antall kanter mellom s og v for alle v ∈ V. Et bredde-først-søk kan ses på som en bølge som sendes ut i alle retninger fra startnoden. Først finner vi alle noder som kan nås ved å gå over 1 kant fra s, dvs. de er direkte noder. Deretter finner vi alle de nodene som kan nås ved å gå over 2 kanter fra, etc. Algoritmen benytter seg av en først-inn-først-ut-kø Q for å ivareta «bølgefronten» (noder som er oppdaget). v ∈ Q bare hvis noden er blitt truffet av bølgen, men ikke forlatt den ennå. Vi kan vise at Q aldri vil inneholde noder med mer enn 2 forskjellige avstander i v.d. Skulle det være 2 ulike verdier, vil alltid de med lavest verdi ligge først. Dette gjør at disse nodene alltid blir behandlet først. Hver node v får satt v.d bare en gang, og v.d er monotonisk stigende over tid. Når vi skal analysere et bredde-først-søk, kan vi med en gang merke oss at BFS ikke alltid vil nå samtlige noder i G. Algoritmen legger til og tar ut hver node fra Q bare én gang, og *enqueue* og *dequeue* tar begge konstant tid O(1). Deretter undersøker vi kantene til hver node bare én gang (når noden tas ut av køen), hvilket gir oss en kjøretid på O(V+E) for BFS-algoritmen.

Et **dybde-først-søk (DFS)** vil gå så dypt som mulig (i motsetning til BFS) før den returnerer. Input til DFS er en graf G = (V, E) som enten kan være rettet eller urettet. Utenhet er to «tidsstempler», v.d og v.f som angir når en node er oppdaget og når den er ferdig utforsket.
I et DFS starter vi med å undersøke en node så fort den oppdages. Et DFS benytter seg av følgende fargekoder under kjøring av algoritmen: hvit (noder som hittil er uoppdagede), grå (noder som er oppdagede, men ikke ferdig utforskede), og svart (noder som er ferdig utforskede). Tidsstemplene må følge disse reglene: 1) Unike heltall mellom 1 og 2|V|, og 2) Av 2 V : 1 _ v:d < v: f _ 2jVj (A-en skal egentlig være opp-ned). I et dybde-først-søk vil alle noder utforskes én gang, og alle kanter besøkes én gang. Dette gir oss en kjøretid på Θ(V+E), hvilket er nesten det samme som BFS (forskjellen er at DFS besøker alle nodene og undersøker alle kantene. Derfor Θ og ikke O).

Usammenhengende sett er ikke en grafalgoritme i seg selv, men de brukes i flere andre algoritmer. Usammenhengende sett er en datastruktur for å ivareta et sett med komponenter uten felles medlemmer. Den ivaretar en samling $T = \{S_1, S_2, \ldots, S_k\}$ med k usammenhengende dynamiske sett. Hvert sett er identifisert av et representativt element som kan være hvilket som helst av elementene i settet. Det er 3 hovedoperasjoner på usammenhengende sett:
makeSet(x): Lager et nytt usammenhengende sett $S_i = \{x\}$ og legger S_i til T.

union(x, y): Hvis $x \in S_x$ og $y \in S_y$, så er $T = T\text{-}S_x\text{-}S_y \cup \{S_x \cup S_y\}$. *findSet(x)*: Returnerer det representative elementet i settet som inneholder x. For å analysere usammenhengende sett, må vi ta hensyn til 2 ulike størrelser. N er det totale antallet makeSet-operasjoner. M er det totale antallet operasjoner på strukturen (dvs n pluss alle andre operasjoner). Siden m inneholder n, vil m >= n. Vi kan ikke ha flere enn n-1 union-operasjoner, siden etter n-1 kall til union, er alle elementer medlem av samme sett. I minnet er det flere måter å representere usammenhengende sett. Den mest effektive er å bruke en skog med usammenhengende sett. Vi har da et tre per sett. Roten i treet er det representative elementet. Hver node peker baret il sin forelder, dvs. ingen referanser til eventuelle barn. Vi kan da sammenfatte operasjoner på usammenhengende sett slik: makeSet: lager et tre med bare rotnode, union: seter den ene roten til å være barn av den andre, findSet: følg foreldrereferansen til vi når rotnoden. Kjøretiden til findSet kan i verste fall bli lineær. Derfor har man funnet opp et par optimaliseringer: 1) union rank: la roten i det minste treet bli barn til det største, og istedenfor størrelse bruker vi rank som en øvre grense på høyden til en node. 2) path compression: La alle nodene på veien til roten under kjøring av findPath få rotnoden som forelder. Hver node vil da ha to attributter: x.p og x.rank.

Et **minimum spenntre** er det spenntreet med lavest vekt på kantene. Et spenntre er et tre som spenner over alle nodene i en graf G = (V, E). Vi har en urettet graf G = (V, E) med en vekt w(u, v) satt på hver kant $(u, v) \in E$. Vi kan finne et minimum spenntre $T \subseteq E$ slik at T kobler sammen alle nodene i grafen, og $w(T) = \sum_{(u, v) \in T} w(u,v)$ er minst mulig. En graf kan ha flere enn ett minimum spenntre. For å vokse et minimum spenntre kan vi bygge oss et sett A med kanter, sette $A = \emptyset$, A vil nå alltid være et subset av et MST. Så legger vi til noen trygge kanter: Vi deler opp grafen i to deler, S og V – S slik at h er med i S og g er med i V – S. For _a kunne bevise at vi velger trygge kanter trenger vi noen definisjoner.

Vi lar $S \subset V$ og $A \subseteq E$.

Et kutt (cut) (S, V - S) er partisjonering av nodene i to disjunkte sett S og V - S.

En kant (u; v) krysser (crosses) et kutt (S, V - S) hvis en av sluttpunktene er i S og en i V - S.

Et kutt respekterer (respect) A hvis ingen av kantene i A krysser kuttet.

En kant er en lett kant (light edge) hvis vekten til kanten er den minste av alle kantene som krysser kuttet. For ethvert kutt er det > 1 antall lette kanter.

Kruskal-algoritmen kan beskrives slik: Vi har en sammenkoblet urettet graf G = (V, E) og en vektorfunksjon $w: E \rightarrow R$. *) Vi starter med at alle noder er sitt eget komponent. *) Vi slår sammen hver node ved å velge en lett kant som kobler de sammen *) Vi søker gjennom kantene i monotonisk stigende rekkefølge i forhold til vekten til kanten. *) Vi bruker datastrukturen disjoint-sett for å bedømme om en kant knytter sammen to ulike komponenter.

Kruskalalgoritmen er en sammensatt algoritme. Vi har |V| kall til makeSet, dvs. $\Theta(V)$. Sortering av kantene i monotonisk stigende rekkefølge tar O(E log E). Den andre for-løkken inneholder O(E) kall til findSet og union. Går vi ut ifra at vi bruker både union by rank og path compression, og om vi ser bort ifra den minste av disse, så har vi $O((V+E)\alpha(V))+O(E \log E)$. Dette kan igjen forenkles ytterligere: Siden G er sammenkoblet er |E| >= |V| -1, så da kan vi forenkle til $O(E\alpha(V)) + O(E \log E)$. $\alpha(V) \in O(\log V) \in E\ O(\log E)$. Dette gir oss en kjøretid på O(E log E). $|E| <= |V|^2$ som gir oss $\log |E| = O(2 \log V) = O(\log V)$.

Den endelige kjøretiden blir derfor typisk oppgitt som O(E log V). Dersom alle kantene er sorterte, blir kjøretiden riktignok $O(E\alpha(V))$. **Prim-algoritmen** bygger alltid bare på et tre. Dette i motsetning til Kruskal-algoritmen, som bygger på en skog. I Prim-algoritmen er A alltid et tre. I tillegg starter den fra en tilfeldig «rot» r. For hvert steg så finner vi en lett kant mellom kuttet (V_A, $V – V_A$), hvor V_A er de nodene som er koblet sammen av A. Prim-algoritmen bruker en prioritetskø for å finne en passende kant effektivt. Køen inneholder noder som er med i $V – V_A$. Prioritetskøen er ordnet i forhold til den minste vekten til en kant (u, v) slik at $u \in V_A$. Sorteringsnøkkelen til en node v er evig hvis v ikke er en nabo til en av nodene i V_A. Når vi kaller extractMin, får vi en node v slik at det eksisterer en node $u \in V_A$ og (u, v) er en lett kant som

krysser (V_A, $V-V_A$). Kantene i A er et tre med rot i r. r blir gitt som input, men kan være hvilken som helst av nodene. Hver node v er klar over forelderen sin via v.pi (r.pi = nil). Under kjøring er A = {(v, v.pi): v ∈ V – {r} -Q}. Når algoritmen er ferdig, er V_A = V som impliserer at Q = Ø, og da har vi et MST i A ={(v, v.pi) : v ∈ V – {r]}. Prim-algoritmen er også en sammensatt algoritme. Kjøretiden til algoritmen er helt avhengig av hvordan man implementerer prioritetskøen. Hvis vi antar at prioritetskøen er en såkalt binærheap, får vi følgende: Vi har V kall til insert som gir oss O(V log V), det å minke nøkkelen til rotnoden er O(log V), vi har V kall til extractMin som gir oss O(V log V), samt E kall til decreaseKey som gir oss O(E log V). Siden |V| <= |E|, får vi en total kjøretid på O(E log V).

Bellman-Ford-algoritmen er en såkalt single-source-algoritme for å finne korteste vei. Dette er en algoritme som tillater negative vekter. Den returnerer true hvis ingen negative sykler er tilgjengelig fra startnoden. Den finner v.d og v.pi for alle v ∈ V. Ettersom at man bare trenger å telle løkker for å analysere denne algoritmen, er dette en svært enkel algoritme å analysere. Vi har en løkke som kjører |V|-1 ganger med en nestet løkke som kaller relax på alle kantene. Den andre løkken er av lavere orden siden den bare går gjennom alle kantene. Dette resulterer i en kjøretid på Θ(VE).

Dijkstras algoritme er også en single-source-algoritme for korteste vei. I motsetning til Belman-Ford-algoritmen, så støtter ikke Dijkstras algoritme negative vekter. Denne algoritmen kan beskrives som en «vektet versjon av bredde-først-søk». Istedenfor en FIFO-kø, bruker Dijkstra en prioritetskø og sorteringsnøkkelen er v.d som er den antatte korteste veien mellom A og B. Vi har 2 sett med noder: S er settet med noder hvor v.d = δ(s, v). Q er settet med noder som ligger i prioritetskøen, dvs. V – S. Dijkstras algoritme ligner veldig på Prim-algoritmen for minimum spenntre. Den beregner v.d som benyttes som sorteringsnøkkel. Dijkstras algoritme kan sees som en grådig algoritme, siden den alltid velger den nærmeste noden i V – S for å legge til S. Akkurat som Prim-algoritmen, er Dijkstras algoritme for korteste vei avhengig av hva slags prioritetskø som brukes. Hvis vi, i likhet med eksempelet for Prim-algoritmen, tar i bruk en prioritetskø som er implementert på en binærheap, ville vi fått: *) At hver operasjon gar O(log V). *) Vi har |V| extractMin og |E| decreaseKey-operasjoner. *) Da får vi totalt en kjøretid på O((V+E) * log V). Dette kan forenkles til O(E log V) hvis alle nodene er sammenkoblet.

Eksempler og annet

Quicksort: I eksamensoppgaven fra 2014 skulle man «quicksorte» følgende tabell med heltall: 4, 3, 6, 9, 5, 11, 8, 7, 1, 2.

1) 4, 3, 2, 9, 5, 11, 8, 7, 1, 6
2) 4, 3, 2, 1, 5, 11, 8, 7, 9, 6
3) 1, 3, 2, 4, 5, 11, 8, 7, 9, 6 (etter dette steget blir 1 byttet med 1, så det tar jeg ikke med. Deretter blir tallene 3 og 2 sortert, slik at tallene til venstre for piovtverdien (som her er 4) blir sorterte.)
4) 1, 2, 3, 4, 5, 11, 8, 7, 9, 6 (herifra og inn konsentrerer vi oss om verdiene som er større enn 4. Det første som skjer, er at 5 byttes ut med 5, slik at den nye startverdien er 11. Så bytter vi 11 og 6, og 6 blir nå den nye starverdien.).
5) 1, 2, 3, 4, 5, 6, 8, 7, 9, 11 (Her må 6 byttes med 6 igjen, slik at den nye startverdien er 8. Så bytter vi 8 med 7).
6) 1, 2, 3, 4, 5, 6, 7, 8, 9, 11 (Her er den endelig sorterte tabellen, og programmet går ut av rekursjonen for å avsluttes).

Finner gjetning ved hjelp av et rekursjonstre. Vi begynner med ˚
a sette inn cn for Θ(n) som intern kostnad i hver node i treet og ekspanderer til vi ser et mønster.

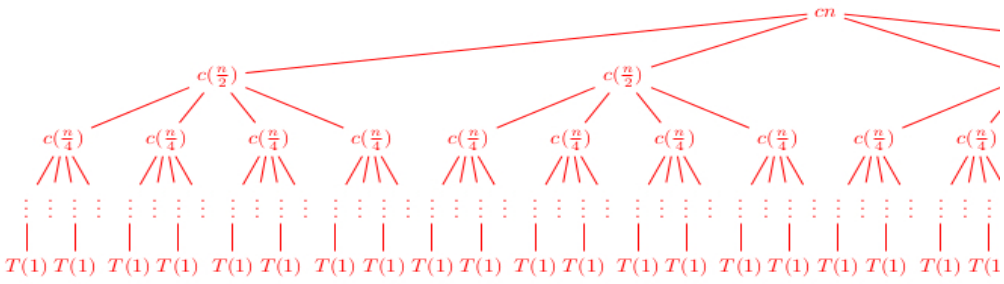
Neste side: Detaljert analyse og avansert analyse

Sum = 0	1
I = 1	1
While I <=n	n+2
Sum = sum + i	n
I = I + 2	n
Return n	n

Total: 1+1+(n+2)+n+n+n+1 = T(3n+5)=Θ(n)

Sum = 0	1
I=1	1
While I <= n	Log_2 n+2
Sum = sum + i	Log_2 n+1
I = I *2	Log_2 n+1
Return sum	1

Total: 1+1+((log_2 n)+2)+((log_2 n) +1) + ((log_2 n) +1)+1=T(n)3(log_2 n)+7 = Θ(log n)



Høyden h på treet blir:

$$\frac{n}{2^h} = 1$$
$$2^h = n$$
$$h = \log_2 n$$

Da har vi $4^h = 4^{\log_2 n} = n^{\log_2 4} = n^2$ antall barn med kostnad $\Theta(1)$ hver, som gir en total kostnad for barna på $\Theta(n^2)$.

Vi har kan nå sette sammen en ligning for kostnaden til recurrence-ligningen ved å summere kostnadene på alle nivåene i treet.

$$\begin{aligned}
T(n) &= cn + 2cn + 4cn + \cdots + \Theta(n^2) \\
&= \sum_{i=0}^{h-1} 2^i cn + \Theta(n^2) \\
&< \sum_{i=0}^{\log_2 n - 1} 2^i cn + \Theta(n^2) \quad \text{geometric series} \\
&= \frac{2^{\log_2 n} - 1}{2 - 1} cn + \Theta(n^2) \\
&= (2^{\log_2 n} - 1)cn + \Theta(n^2) \\
&= (n-1)cn + \Theta(n^2) \\
&= \Theta(n^2)
\end{aligned}$$

Vår gjetning blir dermed $O(n^2)$ siden det bare blir spurt om en øvre grense. Dette gjør også beviset enklere. Beviset er gitt under med substitusjonsmetoden.

Vi gjetter altså på $T(n) = O(n^2)$ som krever at vi beviser at $T(n) \leq cn^2$. Vi setter $cn2$ for T og kn for $\Theta(n)$ (vi setter inn konstanten k for å ikke blande med konstanten c) i recurrence ligningen får følgende.

$$\begin{aligned}
T(n) &\leq 4c\left(\frac{n}{2}\right)^2 + kn \\
&= cn^2 + kn
\end{aligned}$$

Dette fører ikke frem til den ønskede formen på $T(n) \leq cn^2$ siden $cn^2 + kn > cn^2$ så lenge $k > 0$. Vi reduserer derfor gjetningen vår med en lavereordens ledd for å prøve å få matematikken til å gå opp. Siden vi endte opp med et n-ledd til overs, er det naturlig å begynne med en gjetning som trekker fra et n-ledd i.e. $O(n^2 - n)$.

$$\begin{aligned} T(n) &\leq 4c\left(\left(\frac{n}{2}\right)^2 - \frac{n}{2}\right) + kn \\ &= cn^2 - 2cn + kn \\ &\leq c(n^2 - n), \text{ så lenge } c \geq k \end{aligned}$$

Begrensingen $c \geq k$ kommer fra å løse følgende ulikhet i henhold til c.

$$\begin{aligned} cn^2 - 2cn + kn &\leq cn^2 - cn \\ -cn &\leq -kn \\ c &\geq k \end{aligned}$$

Vi har da bevist at $T(n) = O(n^2)$ for det rekursive tilfellet, siden $O(n^2 - n) = O(n^2)$. Da gjenstår det bare å bevise det samme for grensetilfellet. Vi vet at $T(1) = \Theta(1) = O(1) = m$ for en positiv konstant m. Dette oppfyller kravet $m \leq c^2$ så lenge $c \geq m$.

Vi har da bevist at $T(n) = O(n^2)$ for en konstant $c \geq \max(k, m)$ og $n_0 = 1$.

B

Hvilke metoder for å løse recurrence-ligninger er brukbare på ligningen under. Begrunn svaret. Bruk en av de applikable metodene for å komme frem til en øvre grense.

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1 \\ T(n-1) + \Theta(n) & \text{if } n > 1 \end{cases}$$

Vi kan ikke bruke masterteoremet for å løse recurrence-ligningen siden den ikke er på formen $T(n) = aT(n/b) + f(n)$. Vi står da igjen med rekursjonstrær og substitusjonsmetoden.

Det er enkelt å se at siden n minkes med 1 på hvert nivå, så vil vi få rekursjonstreet få en høyde på n. Jobben internt i hver node er $\Theta(n) = cn$. Selvom n minker for hvert nivå, er kostnaden avhengig av størrelsen n. Vi kan derfor ganske enkelt multiplisere høyden med kostnaden og ende opp med en gjetning på $n \cdot n = n^2 = O(n^2)$.

Vi verifiserer så denne gjetningen med substitusjonsmetoden. Vi vil bevise at $T(n) = O(n^2)$. Dette krever at $T(n) = cn^2$. Vi bytter ut $T(n)$ med cn^2 og $\Theta(n)$ med kn (for den interne kostnaden).

$$\begin{aligned} T(n) &= c(n-1)^2 + kn \\ &= cn^2 - 2cn + c + kn \\ &= cn^2 - c(2n-1) + kn \\ &\leq cn^2, \text{ så lenge } n \geq 1 \text{ og } c \geq k \end{aligned}$$

Begrensingen kommer fra å løse følgende ulikhet.

$$\begin{aligned} cn^2 - c(2n-1) + kn &\leq cn^2 \\ c(2n-1) &\geq kn \\ c &\geq \frac{kn}{2n-1} \end{aligned}$$

For grensetilfellet så prøver vi $T(1) = c$ og $c \cdot 1^2 = c$. Vi kan derfor si at $n_0 = 1$ og $c = 1$ og at vi har bevist at $T(n) = O(n^2)$.

Masterteoremet

La a være >= og b > 1 være konstanter og la T(n) være definert på ikkenegative heltall av gjentakelsen T(n) = aT(n/b)+f(n). Da har T(n) de følgende asynptotiske grensene:

1) Hvis f(n) = $O(n^{\log_b a - e})$ for en konstant e > 0, så er T(n) = $\Theta(n^{\log_b a})$.

2) Hvis f(n) = $\Theta(n^{\log_b a})$, så er T(n)=$\Theta(n^{\log_b a} \log n)$

3) Hvis f(n) = $\Omega(n^{\log_b a + e})$ for en konstant e > 0, og hvis af(n/b) <= cf(n) for en konstant c < 1 og en tilsvarende stor n, så er T(n) = Θ(f(n))