Algoritmer og Datastrukuter

Johan Vik Mathisen

2021

Kompendium
Alle feil er mine egne.

Contents

1 Algoritmer

1.1 INSERTION-SORT

Beskrivelse: Input er en liste $A = \langle a_1, \ldots, a_n \rangle$. Algoritmen "deler" listen i to, en sortert del og en usortert del. Starter med $\langle a_1 | \, a_2, \ldots, a_n \rangle$. For hver iterasjon flyttes det første elementet i den usorterte delen inn på riktig plass i den sorterte listen.

Formål: Sortering

Datastruktur: Liste

Kjøretid: Best: $O(n)$, average: $O(n^2)$, worst: $O(n^2)$.

Krav til input: Ingen

1.2 MERGE-SORT

Beskrivelse: Rekursiv sorteringsalgoritme, designet med designmetoden split og hersk. Sorteringen skjer i "combine" fasen.

Tar inn et array, og kaller seg selv på hver av halvdelene. Rekusjonen bunner ut når lengden på listene er 1, og lister av lenge en er trivielt sorterte. Kombinerer de sorterte listene steg for steg til hele arrayet er sortert.

$$
\begin{array}{cccccccc}
 & & & [2,1,4,1] & & & & \\
 & & [2,1] & & & & [4,1] & \\
[2] & & [1] & & [4] & & [1] & \\
 & & [1,2] & & & & [1,4] & \\
 & & & [1,1,2,3] & & & &
\end{array}
$$

Formål: Sortering

Datastruktur: Array

Kjøretid: Worst: $O(n \lg(n))$, average: $O(n \lg(n))$, best: $O(n \lg(n))$

Krav til input: Ingen

1.3 (RANDOMIZED)-QUICKSORT

Beskrivelse: Rekrusiv sorteringsalgoritme, designet med split og hersk. Sorteringen skjer i "divide" fasen. Sorterer in place.

Nøkkelen bak effektiviteten til QUICKSORT ligger i subrutinen PARTITION. PARTITION permuterer (stokker om) elementene i inputarrayet A rundt et *pivotelement* med indeks p slik at elementene $A[i]$ hvor $i \leq p$, har mindre eller lik verdi enn $A[p]$. Elementene med indeks større enn p har høyere eller lik verdi som $A[p]$. Det ideelle er om pivotelementet $A[p]$ er medianen av tallene i arrayet.
Ideen i algoritmen er at QUICKSORT kjører PARTITION rekursivt på hver halvdel av arrayet, og med små nok arrays vil PARTITION sortere riktig. "Combine" steget er trivielt fordi arrayet er sortert idet rekusjonen bunner ut.
Forskjellen på RANDOMIZED og vanlig QUICKSORT er valget at pivotelement. I vanlig er det det siste elementet i arrayet, i den randomiserte varianten er det en tilfeldig annen indeks (uniformt). Siden QUICKSORT er $O(n^2)$ i værste tilfelle og det er kjent at det intreffer når pivotelementet er det største elementet i arrayet, kan det utnyttes at pivotelementet alltid velges likt.

Formål: Sortering

Datastruktur: Array

Kjøretid: Worst: $\Theta(n^2)$, average: $\Theta(n \lg(n))$, best: $\Theta(n \lg(n)$

Krav til input: Ingen

1.4 BINARY SEARCH (BISECT)

Beskrivelse: Søker etter en bestemt verdi x i en sortert liste ved å sammenlikne det med det midterste elementet m. Hvis $x = m$ er vi ferdig, er $x < m$ kjøres BISECT rekursivt på nedre halvdel, hvis $x > m$ kjøres BISECT rekursivt på øvre halvdel. Lynrask!

Formål: Søk

Datastruktur: Array

Kjøretid: $O(\lg(n))$

Krav til input: Arrayet må være sortert.

1.5 COUNTING-SORT

Beskrivelse: Input er et array A og et heltall k. Lager et tellearray med k elementer og itererer gjenom A og lagrer frekvensen til tallene $0, ..., k$ i C. Gjør om C fra frekvensen av elementene til den komulative frekvensen til elementene ($C[i]$ er antall tall mindre eller lik i etter dette). For i fra n ned til 1 puttes $A[i]$ inn som det $C[A[i]]$'te elementet i outputarrayet, og $C[A[i]]$ dekrementeres. Det vil si at $A[i]$ puttes sist av alle elemntene med den verdien, og neste gang

et element med lik verdi dukker opp vil det bli plasser forran (dette steget gjør at algoritmen er stabil).

Formål: Sortering

Datastruktur: Array

Kjøretid: Hvis $k = O(n)$ kjører den i $\Theta(n)$.

Krav til input: $A[i] \leq k$ for alle $i \leq n$. Elementene i arrayet er begrenset ovenfra av et heltall k som vi kjenner.

1.6 RADIX-SORT

Beskrivelse: Input er et array og antall siffer d. Fra minst til mest signifikatnt sorterer vi A etter siffer i med en stabil sorteringsalgoritme.

Sorterer fra minst til mest signifikant fordi noe annet blir veldig rart. Subrutinen må være stabil, fordi mindre signifikante siffer utgjør en forskjell når noen av de mer signifikante er like.

Formål: Sortering

Datastruktur: Array

Kjøretid: $\Theta(d(n+k)) = \Theta(n)$ når $k = O(n)$ og d er konstant.

Krav til input: Elementene i arrayet har d siffer.

1.7 BUCKET-SORT

Beskrivelse: Input er et array A med n elemnter. Deler intervallet $[0, 1)$ i n like store deler. Lager en liste B av n lenkede lister. Itererer gjennom A og legger hvert element i sin lenkede liste i B (som bestemt av oppdelingen av $[0, 1)$).

Avslutningsvis sorteres alle listene $B[i]$ med INSERTION-SORT før de konkatineres.

Formål: Sortering

Datastruktur: Array

Kjøretid: $O(n)$,

Krav til input: Elementene er trukket uniformt fra $[0, 1)$.

1.8 RANDOMIZED-SELECT

Beskrivelse: Input er et array A, start- og slutt-indeks og i (som i det i'te minste elementet). Bruker RANDOMIZED-PARTITION som RANDOMIZED-QUICKSORT, og bruker pivotelementet til å bestemme hvilken av partisjonene det ønskede elementet ligger i. Trenger bare å undersøke den ene grenen i rekursjonstreet, derfor er den kjappere enn (RANDOMIZED-)QUICKSORT.

Formål: Utvalg, finne elementet i en liste som er større enn nøyaktig $i - 1$ andre.

Datastruktur: Array

Kjøretid: Forventet kjøretid er $\Theta(n)$ når elementene er distinkte. Worst: $\Theta(n^2)$

Krav til input: Distinkte elementer (for kjøretidsanalysen).

1.9 SELECT

Beskrivelse: Input er et array A, start og slutt indeks og i (som i det i'te minste elementet). Bruker PARTITION som QUICKSORT, og garanterer en god oppdeling ved å velge pivotelement på en god måte. (Ikke pensum å forstå hvordan)

Formål: Utvalg, finne elementet i en liste som er større enn nøyaktig $i - 1$ andre.

Datastruktur: Array

Kjøretid: Forventet kjøretid er $\Theta(n)$ når elementene er distinkte. Worst: $\Theta(n)$

Krav til input: Distinkte elementer (for kjøretidsanalysen).

1.10 HEAP-SORT

Beskrivelse: Utnytter maks-haug til å sortere data. Inputarrayet gjøres til en maks-haug med BUILD-MAX-HEAP. Utnytter at arrayets største element er rota i haugen. Bytter om rota med det siste elementet i arrayet, dekrementerer heap-size og kaller MAX-HEAPIFY for $i = A$.length ned til 2. Man sitter igjen med et array sortert i stigende rekkefølge, konstruert "baklengs".

Formål: Sortering

Datastruktur: Heap (Haug)

Kjøretid: $O(n \lg(n))$.

Krav til input: Ingen

1.11 HUFFMANN CODE

Beskrivelse: Variabel-lengde prefiks kode som identifiserer korte kodeord med symboler med høy frekvens (opptrer hyppig).

Tar inn et alfabet C hvor elemntene har en frekvens og lager en min-prioritetskø Q som initialiseres til å inneholde elementene i C. Slår så iterativt sammen to og to av de minst hyppige symbolene og erstatter de (i Q) med en foreldrenode med den samlede frekvensen. Returnerer roten til det binære treet som er lagd.

Koden produsert av HUFFMANN er en optimal prefiks kode.

Formål:Datakomprimering

Datastruktur: Min-prioritetskø

Kjøretid: $O(n \lg(n))$

Krav til input: En mengde med symboler og tilhørende frekvenser.

1.12 BFS

Beskrivelse: BFS starter i en node s og utforsker alle sine nabonoder, så sine nabonoders nabonoder, osv. Den beregner minste antall kanter fra s til alle andre noder. Samtidig produseres et bredde først tre, med rot s, som inneholder alle noder som kan nås fra s.

Hvis vi tenker på avstanden mellom to noder som antall kanter imellom dem vil bredde-først søk først finne alle noder avstand 1, så 2, osv.

Formål: Søk, finne korteste vei i en uvektet graf.

Datastruktur: Graf, bruker en FIFO kø.

Kjøretid: $O(V + E)$

Krav til input: Ingen

1.13 DFS

Beskrivelse: DFS starter i en tilfeldig node og besøker en nabonode, så besøker den en av nabonodenes nabonoder, osv. Med denne strategien forsøker DFS å komme seg så dypt i grafen som mulig. Når det ikke er mulig å oppdage fler

startes DFS fra en tilfeldig annen node, helt til hele grafen er oppdaget. Noder blir kun besøkt en gang. Ved rekursiv implementering vil kallstakken fungere som en LIFO kø.

Denne rekursive og "diskontinuerlige" naturen i algoritmen resulterer i en bredde-først skog (istedenfor et tre) av besøkte noder og kanter. Skogen kan blandt annet hjelpe til med å klassifisere kanter i grafen.

Formål: Søk, klassifisere kanter

Datastruktur: Graf, kan bruke LIFO kø.

Kjøretid: $\Theta(V + E)$

Krav til input: Ingen

1.14 TOPOLOGICAL-SORT

Beskrivelse: En topologisk sortering gir en lineær ordning (tenk på $\leq$) av nodene i en rettet asyklisk graf. Den bruker DFS som subrutine og legger $v.f$ inn i en lenket liste fortløpende. Den lenkede listen er en lineær ordning av nodene.

Ordningen kan blandt annet følges av en DP algoritme hvis det er en delinstansgraf. Mer typisk beskriver grafen ting som skal gjøres og kanter sier noe om hvilke oppgaver som må fullføres før andre. Den topologiske sorteringen gir informasjon om hvilke oppgaver som kan gjøres uavhengig av hverandre og ikke.

Formål: Hente ut "hendelsesforløpet" i en graf.

Datastruktur: Graf

Kjøretid: $\Theta(V + E)$

Krav til input: Rettet asyklisk graf (DAG)

1.15 GENERIC-MST

Beskrivelse: Starter med en tom mengde A og legger til trygg kant etter trygg kant til den danner et minimalt spenntre. Med andre ord vedlikeholder algoritmen følgende løkkeinvariant: Før enhver iterasjon er A en delmengde av et minimalt spenntre. Utfordringen er å finne ut hva trygge kanter er. Det viser seg at lette kanter er trygge, og problmet kan løses grådig.

Formål: Finne minimalt spenntre

Datastruktur: Graf

Kjøretid: NaN

Krav til input: Urettet vektet graf

1.16 MST-KRUSKAL

Beskrivelse: Lager ettpunktsmengder for hver node i input grafen, og så sorteres kantene i stigende rekkefølge etter vekt. Itererer gjennom kantene i stigende rekkefølge og sjekker om nodene i kanten ligger i samme tre, hvis ikke er dette en lett kant og legges til i det minimale spenntreet. Mengdene representert ved nodene kombineres (vi tar unionen).

Subrutiner:

- MAKE-SET(V)
- FIND-SET(U)
- UNION(v,u)

Formål: Finne minimalt spenntre

Datastruktur: Graf

Kjøretid: Avhenger av subrutinene, hvis vi antar disjoint-set-forest implementasjonen er den $O(E \lg(E))$.

Krav til input: Urettet vektet graf

1.17 MST-PRIM

Beskrivelse: Alle nodene har en key-attributt som er vekten til den letteste kanten som kobler noden til treet som bygges. Det holdes også styr på foreldrerelasjoner i treet ved attributten π. Nodene initialiserer til å ha key $= -\infty$ og $\pi =$ NIL. Det lages en min-prioritetskø av nodene og så lenge den ikke er tom hentes minimum ut og naboenes key og π attributter oppdateres. Når aloritmen terminerer er $A = \{(v, v.\pi) \mid v \in V - \{r\}\}$, hvor r er startnoden, et minimalt spenntre.

Formål: Finne minimalt spenntre

Datastruktur: Graf, min-prioritetskø.

Kjøretid: $O(E \lg(V)$ i bokas variant, men kan kan forbedres til $O(E + V \lg(V)$ ved å bruke en fibonacci haug i min-køen.

Krav til input: Urettet vektet graf

1.18 RELAX

Beskrivelse: Input er en kant (u, v). RELAX oppdaterer den øvre skranken på korteste vei fra kilden til noden v om estimatet til u pluss vekten til kanten (u, v) er (strengt) bedre enn estimatet til v. Kritisk subrutine i algoritmer som løser kortese vei fra en til alle problemet.

Formål: Hjelpe å løse korteste vei.

Datastruktur: Graf (naboliste)

Kjøretid: $O(1)$

Krav til input: INITIALIZE-SINGLE-SOURCE må ha blitt kjørt

1.19 BELLMANN-FORD

Beskrivelse: Kjører RELAX på alle nodene $|V| - 1$ ganger, som fører til at at $v.d = \delta(s, v)$ er for alle nodene, fra Path-relaxation property. Korteste simple vei kan ikke være innom fler enn $|V| - 1$ kanter, fordi den da ville hatt en sykel og positive sykler er aldri en del av en korteste vei. Algoritmen oppdager om grafen har negative sykler ved å sjekke om noen av $v.d$ verdiene forbedres etter de $|V| - 1$ første iterasjonene.

Formål: Single-source shortest path

Datastruktur: Graf (naboliste)

Kjøretid: $O(VE)$

Krav til input: Vektet og rettet graf.

1.20 DAG-SHORTEST-PATH

Beskrivelse: Kjører en topologisk sortering på nodene. I denne lineære ordningen kjøres RELAX node for node, og ved Path-relaxation property vil alle nodene ha $v.d = \delta(s, v)$. Noder som ikke kan nås fra kilden ligger forran kilden i den lineære ordningen.

Formål: Single-source shortest path

Datastruktur: Graf (naboliste)

Kjøretid: $\Theta(V + E)$

Krav til input: Vektet og rettet asyklisk graf (DAG)

1.21 DJIKSTRA

Beskrivelse: Holder styr på nodene som har $v.d = \delta(s, v)$ i en mengde S. Gjør kantmengden til en min-prioritetskø. Med EXTRACT-MIN hentes noden med minst $v.d$ verdi ut og legge i S. Så kjøres RELAX på alle naboene til denne noden. Fortsetter til køen er tom. Grådig algoritme.

Formål: Single-source shortest path

Datastruktur: Graf (naboliste), min-prioritetskø

Kjøretid: $O(V^2)$, kan forbedres under visse omstendigheter ved å endre haugimplementasjonen.

Krav til input: Vektet og rettet graf med ikke-negative kanter.

1.22 FLOYD-WARSHALL

Beskrivelse: DP algoritme. Dekomponeringen sier i utgangspunktet "skal vi innom k eller ikke".

Formål: All pairs shortest path

Datastruktur: Graf (nabomatrise)

Kjøretid: $\Theta(n^3)$

Krav til input: Ingen negative sykler

1.23 TRANSITIVE-CLOSURE

Beskrivelse: Modifiserer FLOYD-WARSHALL til å finne den transitive tillukningen av en graf ved å bytte ut min() med $\vee$ og + med $\wedge$. Elementet t_{ij} i matrisen initialiseres til å være 0 om $(i, j) \notin E$ og 1 hvis $(i, j) \in E$.

Formål: Finne den transitive tillukningen til en graf.

Datastruktur: Graf (nabomatrise)

Kjøretid: $\Theta(n^3)$

Krav til input: Rettet graf

1.24 JOHNSON

Beskrivelse: Løsning på alle til alle problemet for glisne grafer (Sparse graphs). *Veier om vektene* i grafen slik at de er ikke-negative, samtidig korteste vei mellom alle nodene holder seg uendret. Bruker BELLMANN-FORD en gang for å undersøke om grafen har negative sykler og for å initilisere omveiingen. Avsluttningsvis kjøres DJIKSTRA fra alle noder i grafen.

Subrutiner: DJIKSTRA og BELLMANN-FORD

Formål: All pairs shortest path

Datastruktur: Graf, fibonacci haug min-prioritetskø.

Kjøretid: $O(V^2 \lg(v) + VE)$

Krav til input: Rettet graf.

1.25 FORD-FULKERSON(-METHOD)/EDMONDS-KARP

Beskrivelse: Tar inn et flytnett og initialiserer flyten til å være 0. Så lenge det finnes en forøkende sti i restnettet få forøkes flyten langs denne stien. Utnytter maks-flyt/min-snitt teoremet som forteller at denne prosedyren må returnere en maksimal flyt.

Hvis BSF brukes til å finne forøkende stier kalles den EDMONDS-KARP.

Formål: Max-flow

Datastruktur: Graf

Kjøretid: $O(E|f^*|)$ hvor f^* er den maksimale flyten i nettet etter kapasiteten er skalert til heltallsverdier (FF). For EDMONDS-KARP er den $O(VE^2)$.

Krav til input: Flytnett

2 Datastrukturer

2.1 Stack (FIFO)

Beskrivelse: Dynamisk mengde hvor siste element lagt til er første element som forsvinner (FIFO).

Implementasjonsmetoder: Liste

Metoder:

- PUSH(x), flytter S.top en opp og setter S[S.top] = x
- POP, sjekker for underflow, eller dekrementerer S.top med 1 og returnerer elementet som var S.top.
- STACK-EMPTY(), sjekker om S.top = 0.

Attributter:

- S.top, indeksen til det øverste elementet i stakken.

2.2 Queue (LIFO)

Beskrivelse: Dynamisk mengde hvor første element lagt til er første element som forsvinner (LIFO). I en kø er den indeliggende listen en sirkel, Q[n+1] = Q[1].

Implementasjonsmetoder: Liste med n elementer: Q. Elementene ligger på posisjonene Q.head, Q.head-1,..., Q.tail-1.

Metoder:

- ENQUEUE(x), setter Q[Q.tail] = x og flytter halen en plass fremover ($\rightarrow$), tar høyde for wraparound.
- DEQUEUE(), returnerer verdien til hodet og flytter det fremover ($\rightarrow$), tar høyde for wraparound.

Attributter:

- Q.tail, indeksen bak det siste elementet i køen (der neste element som legges til blir plassert)
- Q.head, indeksen til det første elementet i køen.

2.3 Linked lists

Beskrivelse: En datastruktur hvor objektene har en lineær ordning, ordningen bestemmes av pekere fra et objekt til neste. Enkelt eller dobbelt lenka. Veldig fleksibel, ikke veldig effektiv.

Implementasjonsmetoder: Kan implementeres i en vanlig liste med satelittdata.

Metoder:

- LIST-SEARCH(L,k), finner det første elementet i L med key = k, lineært søk, returnerer peker til elementet eller NIL.
- LIST-INSERT(L,x), legger et listeelement x (x = [key—prev—next]) inn forran i lista og gjør dette til hodet.

- LIST-DELETE(L,x), x er en peker til et listeelement. Fjerner x fra listen ved å oppdatere pekerne til x.next og x.prev.

Attributter:

- key, verdien/objektet
- next, en peker til neste objekt i listen
- prev, en peker til forrige objekt i listen (hvis dobbelt lenka)
- head, første elementet i lista.

2.4 Hash table (chaining)

Beskrivelse: En minneeffektiv liste. Nyttig når en liten andel av de mulige nøklene blir brukt eller når universet av nøkler er uhåndterbart stort. Hvis vi antar simple uniform hashing har alle metodene $O(1)$ gjennomsnittlig kjøretid.

Implementasjonsmetoder: Liste med lenkede lister. Trenger en god hash-funksjon h.

Metoder:

- CHAINED-HASH-INSERT(T,x), insert x at the head of list T[h(x.key)]
- CHAINED-HASH-SEARCH(T,k), search for an element with key k in list T[h(k)]
- CHAINED-HASH-DELETE(T,x), delete x from the list T[h(x.key)]

Attributter:

- key
- T[k].next (og prev hvis delete skal gå fortere enn search).

2.5 Dynamic array

Beskrivelse: Fleksibelt array som endrer hvor mye minne som er allokert dynamisk etterhvert som elementer legges til og fjernes. Stjerneeksempel på amortisert analyse, alle metodene er $O(1)$ amortisert.

Implementasjonsmetoder: Array
Metoder:

- TABLE-INSERT, hvis T.num < T.size kjøres vanlig insert, ellers kopieres T over i en dobbelt så stor tabell og så kjøres insert. Alle verdier oppdateres

- TABLE-DELETE, hvis T.num > LoadFactor*T.size kjøres vanlig delete, ellers kopiers T over til en halparten så stor tabell og delete kjøres. Alle verdier oppdateres

Attributter:

- T.table, peker til tabellen
- T.num, antall elementer in tabellen
- T.size, totalt antall plasser i tabellen.

2.6 (Max-)Heap

Beskrivelse: En rotfast trestruktur hvor alle noder er mindre eller lik enn rotnoden, denne egenskapen gjelder for alle deltrær (heap-property). Tilnærmet komplett binært tre, kun siste lag som evt. ikke er fullt.

Implementasjonsmetoder: Array

Metoder:

- PARENT(i), returnerer indeksen til foreldrenoden som er $\lfloor$ i/2$\rfloor$.
- LEFT(i), returnerer indeksen til venstre barnenode som er 2i.
- RIGHT(i), returnerer indeksen til høyre barnenode som er 2i+1.
- MAX-HEAPIFY(A,i), antar at deltrærne med røtter i LEFT(i) og RIGHT(i) er max-hauger. Hvis node i bryter haugegenskapen fikser denne metoden det ved. Kjøretid $O(\lg(n))$.
- BUILD-HEAP(A), bruker MAX-HEAPIFY til å gjøre et array om til en maks-haug. Kjøretid $O(n)$.

Attributter:

- length, antall elementer i arrayet;
- heap-size, antall elementer i haugen som er lagret i arrayet, A.heap-size $\leq$ A.length.

2.7 Max-priority queue (Heap)

Beskrivelse: Utnytter at roten i en haug er største element til å lage en effektiv kø hvor key attributtet bestemmer hvilkent element som forlater køen først.

Implementasjonsmetoder:

Metoder: Array

- Arver fra max-heap
- INSERT(S,x), legger til x i mengden S.
- MAXIMUM(S), returnerer elementet i S med størst nøkkel.
- EXTRACT-MAX(S), fjerner og returnerer elementet i S med størst nøkkel.
- INCREASE-KEY(S,x,k), øker verdien til x sin nøkkel til en ny verdi k, som antas å være minst like stor som den nåværende nøkkelen.

Attributter:

- Arver fra max-heap

2.8 Rooted tree

Beskrivelse: Ikke-lineær dobbelt lenket liste med en utvalgt node vi kaller rotnoden.

Implementasjonsmetoder: Dobbelt lenkede lister med ekstra satellittdata. Alternativt kan de implementeres som vi implementerer hauger.

Metoder:

- Kommer frem i de ulike trestrukturene som introduseres nedover.

Attributter:

- root, peker til rotnoden.
- p (parent), peker til foreldrenoden. NIL om noden er T.root.
- child_1,..., child_k, pekere til hver av barnenodene. child_i er NIL om noden ikke har barn der.

2.9 Binary search tree

Beskrivelse: Et binært rotfast tre som tilfredsstiller binary-search-tree property (BSTP): La x være en node i et binært søketre. Hvis y er en node i det venstre deltreet til x, da er y.key $\leq$ x.key. Hvis y er en node i det høyre deltreet deltreet til x, da er y.key $\geq$ x.key.

Implementasjonsmetoder: Dobbelt lenkede lister med ekstra satellittdata.

Metoder:

- SEARCH(x,k), x er en peker til rota, k er en nøkkel. Returnerer en peker til en node med nøkkel lik k. Kjøretid er $O(h)$ hvor h er høyden på treet, og h er typisk $\ln(n)$.

- MINIMUM(x), x er en peker til rota. Traverser ned venstresiden på treet og returnerer det nederste (som er det minste) elementet. Kjøretid er $O(h)$
- MAXIMUM(x), samme som MINIMUM bare til høyre. Kjøretid er $O(h)$
- SUCCESSOR(), x er en peker til en node. Returnerer en peker til noden med nøkkel som kommer etter x.key i rekkerfølgen bestemt av INORDER-TREE-WALK. Kjøretid er $O(h)$.
- PREDECESSOR(x), helt symmetrisk til SUCCESSOR. Kjøretid er $O(h)$.
- INSERT(T,z), setter inn noden z i treet T og bevarer BSTP. Kjøretid er $O(h)$.
- DELETE(T,z), sletter noden z og bevare BSTP. Ganske omfattende metode å implementere. Kjøretid er $O(h)$.
- INORDER-TREE-WALK(x), printer ut nodene i stigende rekkefølge. Kjøretid er $\Theta(n)$!

Attributter:

- key, verdien til noden.
- root, peker til rotnoden.
- p (parent), peker til foreldrenoden. NIL om noden er T.root.
- left, pekere til venstre barnenode. NIL om noden er en løvnode.
- right, pekere til høyre barnenode. NIL om noden kun har en barnenode.

2.10 Graph

Beskrivelse: Retta og uretta grafer.

Implementasjonsmetoder: $G = (V, E)$ kan implementeres som en nabomatrise eller en samling av nabolister. Fler alernativer finnes, men disse brukes mest i faget.

Naboliste representasjonen består av et array Adj med $|V|$ lister. For enhver $u \in V$ inneholder Adj$[u]$ alle nodene v slik at $(u, v) \in E$.

Nabomatrise representasjonen antar at nodene er nummerert (arbitrært) $1, 2, ..., |V|$. Representasjonen er da en $|V| \times |V|$ matrise slik at

$$a_{ij} = \begin{cases} 1 & \text{hvis } (i,j) \in E \\ 0 & \text{ellers} \end{cases}$$

Attributter:

- Implementasjon av attributter har ingen god standard på tvers av språk.

2.11 (Minimum) Spanning trees

Beskrivelse: Gitt en graf $G = (V, E)$ er ett spenntre til G en asyklisk delgraf $S = (V, E')$. Det vil si at S er et tre som inneholder alle nodene til G. Hvis G er vektet, sier vi at S er et minamalt spenntre om den totale vekten av kantene er minst av alle mulige spenntrær av G.

3 Teoremer

Master-teoremet

La $a > 0$ og $b > 1$ være konstanter, la $f(n)$ være en funksjon, og la $T(n)$ vere definert på ikkenegative heltall ved rekurensen

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

hvor $\frac{n}{b}$ tolkes til å være enten floor$\left(\frac{n}{b}\right)$ eller ceil$\left(\frac{n}{b}\right)$. Da har $T(n)$ følgende asymptotisk grense:

1. Hvis $f(n) = O(n^{\log_b(a-\epsilon)})$ for $\epsilon > 0$, da er $T(n) = \Theta(n^{\log_b(a)})$
2. Hvis $f(n) = \Theta(n^{\log_b(a)})$,da er $T(n) = \Theta(n^{\log_b(a)} \lg(n))$
3. Hvis $f(n) = \Omega(n^{\log_b(a+\epsilon)})$ for $\epsilon > 0$, og hvis $af(n/b) \leq cf(n)$ for en konstant $c < 1$ og alle store nok n, da er $T(n) = \Theta(f(n))$.

Parentesteoremet

I ethvert dybde-først søk i en graf $G = (V, E)$, for ethvert par med noder u og v holder nøyaktig ett av følgende betingelser:

- Intervallene $[u.d, u.f]$ og $[v.d, v.f]$ er disjunkte, og hverken u eller v er etterkommere av hverandre i dybde-først skogen.
- Intervallet $[u.d, u.f]$ er strengt inneholdt i $[v.d, v.f]$, og u er en etterkommer av v i et dybde-først tre.
- Intervallet $[v.d, v.f]$ er strengt inneholdt i $[u.d, u.f]$, og v er en etterkommer av u i et dybde-først tre.

Hvit sti teoremet

I en dybde først skog til en graf $G = (V, E)$ er noden v er etterkommer av noden u hvis og bare hvis ved tid $u.d$, når søket oppdager u, er det en sti fra u til v bestående av kun hvite noder.

4 Designmetoder

4.1 Splitt og Hersk

1. Del opp et problem i mindre instanser av det samme problemet (Divide).
2. Løs delproblemene rekursivt, til man når grunntilfelle (n=1 eller n=0) (Conquer).
3. Kombinere løsningene på delproblemene til en løsning på det orginale problemet (Combine).

4.2 Dynamisk Programmering

For å kunne bruke dynamisk programmering trenger vi to ting:

Optimal delstruktur: En optimal løsning til et problem er en kombinasjon av optimale løsninger på delproblemer.

Overlappende delproblemer: Når en rekursiv algoritme møter på nøyaktig det samme problemet flere ganger sies det at optimaliseringsproblemet har overlappende delstruktur. Alternativt at delinstansgrafen hadde hatt sykler om den var urettet.

1. Finn formen til en optimal løsning.
2. Definer rekursivt verdien til en optimal løsning.
3. Regn ut verdien til en optimal løsning, typisk "bottoms-up".
4. Konstruer en optimal løsning fra den utregnede informasjonen. (Typisk dropper vi dette)

4.3 Grådighet

For at grådig metode skal fungere trenger problemet optimal delstruktur, grådighetsegenskapen og at grådige valg kun etterlater ett delproblem.

Prosessen vi følger når vi skal utvikle grådige algoritmer er som følger:

1. Formuler optimaliseringsproblemet slik at når vi tar et valg sitter vi igjen med ett delproblem å løse.
2. Bevis at det alltid finnes en optimal løsning til det orginale problemet som tar det grådige valget, slik at det grådige valget alltid er trygt.
3. Demonstrer optimal delstruktur. Det vil si at vi skal vise at delproblemet vi sitter igjen med etter det grådige valget har følgende egenskap:
   Hvis vi kombinerer en optimal løsning på delproblemet med det grådige valget har vi en optimal løsning på det orginale problemet.

5 Læringsmål

5.1 Forelesning 1

A3 Kunne definere problem, instans og problemstørrelse.

Problem: Et abstrakt problem Q er en binær relasjon på $I \times S$, hvor I er mengden instanser og S er mengden løsninger. En koding $e : I \to \{0,1\}^*$ gir oss et konkret problem $e(Q)$ (som er en binær relasjon på $\{0,1\}^* \times S$). Kodingen endrer ikke kjøretiden drastisk om man kan dekode og enkode i samme størrelsesorden som algoritmen.

Instans: Et element $i \in I$.

Problemstørrelse: Lengden på instansen $|i| = n$ er problemstørrelsen.

!A4 Kunne definere asymptotisk notasjon, O, Ω, Θ, o og ω.

O: Hvis $f(n)$ er $O(T(n))$ finnes det en c og n_0 slik at for alle $n > n_0$ er

$$0 \leq f(n) \leq cT(n)$$

Det vil si at f er asymptotisk begrenset ovenfra av T. $O(T(n))$ er mengden av funksjoner som f.

Θ :Hvis $f(n)$ er $\Theta(T(n))$ finnes det en c_0, c_1 og n_0 slik at for alle $n > n_0$ er

$$c_0T(n) \leq f(n) \leq c_1T(n)$$

Det vil si at f vokser asymptotisk som T. $\Theta(T(n))$ er mengden av funksjoner som f.

Ω: Hvis $f(n)$ er $\Omega(T(n))$ finnes det en c og n_0 slik at for alle $n > n_0$ er

$$cT(n) \leq f(n)$$

Det vil si at f er asymptotisk begrenset nedenfra av T. $\Omega(T(n))$ er mengden av funksjoner som f.

o: Hvis $f(n)$ er $o((T(n))$ vil det for enhver c finner en n_0 slik at

$$0 \leq f(n) < cT(n)$$

Asymptotisk streng ulikhet, $T(n)$ er ikke $o(T(n))$. Vi har en streng inklusjon av mengder $o(T(n)) \subset O(T(n))$.

ω er samme som o, bare snu ulikhetene (på same måte som Ω og O).

A5 Kunne definere best-case, average-case og worst-case.

- Best-case: Asymptotisk kjøretid ved optimale forhold. Eks: Lineært søk, første elementet er elementet som søkes etter.

- Average case: Asymptotisk kjøretid for en gjennomnittelig instans av en viss størrelse. Kan være vanskelig å matematisk beskrive hva en gjennomsnittlig instans er.
- Worst-case: En øvre grense for algoritmes asymptotiske kjøretid. Ingen instans vil gjøre at algoritmen overskrider worst-case (asymptotisk).

A6 Forstå løkkeinvarianter og induksjon.

Vi bruker løkkeinvarianter til å induktiv vise korrekthet av algoritmer. En løkkeinvariant er en egenskap ved algoritmen som er sann før enhver iterasjon av løkken.

Vi må vise at tre ting om en løkkeinvariant for at den skal vise korrekthet:

- Initialisering: Invarianten er sann før første iterasjon av løkken. (Grunnsteg)
- Vedlikehold: Hvis invarianten er sann før en iterasjon av løkken, er den sann før den neste. (Induktivt steg)
- Terminering: Når løkken avsluttes, gir invarianten en nyttig egenskap som hjelper å vise at algoritmen er korrekt. (Tillegsgreie fordi vi gjør "endelig induksjon").

!A7 Forstå rekursiv dekomponering og induksjon over delinstanser.

Hvis vi kan løse grunntilfellet, og viser at vi kan løse en instans av størrelse n gitt av vi har en løsning på delinstansen av størrelse n-1, har vi induktivt en løsning på det orginale problemet.

5.2 Forelesning 2

H3 Forstå hvordan pekere og objekter kan implementeres

Se i boka, tegningene hjelper.

!B4 Forstå hvordan direkte adressering og hashtabeller fungerer

En dynamisk mengde hvor alle elementer har en nøkkel fra universet $\{0, ..., m-1\}$ (Tenk at dette er så mye minne som må holdes av), representeres av et array (direkte addresserings tabell) av lengde m.

Direkte addressering refererer til måten vi gir elementene en andresse, elementet med nøkkel k har indeks k i lista (L[k] har nøkkel k). Hensiktsmessig når andelen nøkkler brukt er stor, og når antall elementer i universet ikke er voldsomt stort.

I en hashtabell ($T[0, 1, ..., m-1]$) lagres elementet med nøkkel k på plassen $h(k)$. $h : U \to \{0, 1, ..., m-1\}$ er en hashfunksjon. Istedenfor at $|T| = |U|$, som vi får ved direkte adressering, får vi nå at $|T| = m$.

B5 Forstå konfliktløsing ved kjeding (chaining) (chaining) (Chained-Hash-Insert, Chained-Hash-Search, Chained-Hash-Delete)

Fordi hashfunksjoner ikke er injektive vil minst to elementer med ulike nøkler hashes til samme plass i tabellen (kollisjon). Kjeding er å plassere elementer som hasher til samme plass i en lenket liste.

Vi finner elementer med nøkkel k ved å se på listen T[h(k)].

B6 Kjenne til grunnleggende hashfunksjoner

- For k som trekkes uniformt fra $[0, 1]$ er $h(k) = \lfloor km \rfloor$ en simpel uniform hashfunksjon.
- Division method: $h(k) = k \mod m$. Hvor $m \neq 2^p$, hvis ikke er $h(k)$ kun de p minste bitsa i k. Et primall ikke for nære en eksakt potens av 2 er ofte et godt valg.
- Multiplication method, IKKE FORSTÅTT

B7 Vite at man for statiske datasett kan ha worst-case $O(1)$ for søk

Et statisk datasett har et konstant antall c innholdselementer. Worst case for søk er dermed c og $O(c) = O(1)$.

B8 Kunne definere amortisert analyse

Amortisert analyse fordeler tiden brukt på en sekvens med datastruktur-operasjoner likt på alle operasjonene utført. Har vi en sekvens operasjoner $\langle s_1, s_2, s_3, s_4, s_5 \rangle$ med tider $\langle 1, 1, 1, 1, 6 \rangle$ sier vi at enhver operasjon bruker 2 sek amortisert.

B9 Forstå hvordan dynamiske tabeller fungerer

Stjerneeksempel på amortisert analyse. Når lastfatoren i tabellen blir høy/lav nok kopieres elementene over i en større/mindre tabell. Amortisert er INSERT og DELETE $O(1)$.

5.3 Forelesning 3

C2 Forstå maximum-subarray-problemet med løsninger

Problemet: I et array A[0,1,...,n], finn det sammenhengende delarrayet med høyest sum.

Løsningen: Deler listen i to omtrent like store deler, og får to delinstanser av det samme problemet. Det maksimale delarrayet ligger nå i en av de to delene, eller så krysser det delingspunktet. Lager en subrutine som håndterer casen hvor arrayet kryssert delingspunktet. Det kan igjen løses ved finne det maksimale delarrayet som starter i delingspunktet og går nedover ($\leftarrow$), konkatinert det maksimale delarrayet som starter i delingspunktet+1 og går oppover ($\rightarrow$).

Den ferdige løsningen tar høyde for basecase (low=high), og kaller rekursivt på FIND-MAXIMUM-SUBARRAY i øvre og nedre halvdel og så FIND-MAXIMUM-CROSSING-SUBARRAY og returnerer maximum av disse tre.

!C6 Kunne løse rekurrenser med substitusjon, rekursjonstrær og masterteoremet

Regn noen oppgaver.

!C7 Kunne løse rekurrenser med iterasjonsmetoden (se appendiks B)

Regn noen oppgaver.

C8 Forstå hvordan variabelskifte fungerer

Regn noen oppgaver.

5.4 Forelesning 4

D1 Forstå hvorfor sammenligningsbasert sortering har en worst-case på $\Omega(n \lg(n))$

En korrekt sammenlikningsbasert sorteringsalgoritme er nødt til å kunne produsere enhver permutasjon av inputarrayet.

For et array med n elementer er det $n!$ permutasjoner. Valgtreet til en sammenlikningsbasert sorteringsalgoritme er et fullt binært tre, og enhver løvnode tilsvarer en permutasjon av input. Antall sammenlikninger gjort i værste tilfelle tilsvarer høyden på treet som er $\ln(n!) = \Omega(n \lg(n))$ (Stirling's formel).

D2 Vite hva en stabil sorteringsalgoritme er

En sorteringsalgoritme er stabil om elementer med samme verdi opptrer i samme rekkefølge i input og output arrayet. COUNTING-SORT er stabil.

5.5 Forelesning 5

E5 Vite at forventet høyde for et tilfeldig binært søketre er $\Theta(\lg(n))$

Vit dette, beviset er hårete og lite intuitivt.

E6 Vite at det finnes søketrær med garantert høyde på $\Theta(\lg(n))$

Vit dette. Tegn opp ett hvis du tviler.

5.6 Forelesning 6

!F1 Forstå ideen om en delinstansgraf

Hvis vi har en instans av et problem av størrelse n (f.eks en stav av lengde n) kan vi lage en *delinstansgraf* ved å lage noder for hver delinstans og kanter mellom delinstanser som er avhengige av hverandre. Med det menes det at det er en kant fra u til v hvis en optimal løsning på v brukes direkte i en optimal løsning på u.

!F2 Forstå designmetoden dynamisk programmering

Dynamisk programmering (DP), på samme måte som splitt og hersk (S&H), løser problemer ved å kombinerer løsninger på delprpblemer. Hovedforskjellen er at de samme delproblemene i problemer vi vil løse med dynamisk programmering dukker opp flere ganger. Vi vet at binære trær representerer rekursiv dekomponering i splitt og hersk, det er delinstansgrafen til problemer vi løser med S&H. I DP vil delinstansgrafen ikke lenger være et tre, men en rettet asyklisk graf. For å hindre at vi løser samme delinstans mange ganger lagrer vi resultatene av utregninger og slår opp fortløpende.

For at DP skal kunne anvendes trenger vi to ting: optimal substruktur og overlappende delproblemer. Når vi utvikler DP algoritmer følger vi disse fire (tre) stegene:

1. Finn formen til en optimal løsning.
2. Definer rekursivt verdien til en optimal løsning.
3. Regn ut verdien til en optimal løsning, typisk "bottoms-up".
4. Konstruer en optimal løsning fra den utregnede informasjonen. (Typisk dropper vi dette)

!F3 Forstå løsning ved memoisering (top-down)

Top-down løsningen følger ordningen til delinstansgrafen, og må derfor være en rekursiv algoritme. Memoisering refererer til lagringen av løsninger på delproblemer, og fører til at vi bare må løse de ulike delproblemene en gang hver (på bekostning av minne).

F4 Forstå løsning ved iterasjon (bottom-up)

Løsninger ved iterasjon utnytter den naturlig ordningen, basert på problemstørrese, av delproblemer (delinstansgrafen er en DAG, så vi får en (reversert) topologisk sortering). Istedenfor å kalle på seg selv rekursivt løses problemet iterativt, og det trengs dermed ingen hjelpeprosedyre for å få inn arrayet hvor løsninger på delproblemer lagres.

F5 Forstå hvordan man rekonstruerer en løsning fra lagrede beslutninger

Ved å lagre valgene tatt i de optimale løsningene av delproblemene i et separat array. Med delløsningene og valgene er det ikke vanskelig å rekonstruere løsningen.

F6 Forstå hva optimal delstruktur er

Et problem har optimal delstruktur dersom optimale løsninger på problemet er kombinasjoner av optimale løsninger av delproblemer som vi kan løse uavhengig av hverandre. (At delinstansgrafen er en DAG, med andre ord)

F7 Forstå hva overlappende delinstanser er

Når en rekursiv algoritme møter på nøyaktig det samme problemet flere ganger sies det at optimaliseringsproblemet har overlappende delstruktur. Alternativt at delinstansgrafen ikke er et tre.

F8 Forstå eksemplene stavkutting og LCS

Gjør dette.

F9 Forstå løsningen på det binære ryggsekkproblemet (se appendiks D i dette heftet) (Knapsack, Knapsack')

Gjør det.

5.7 Forelesning 7

!G1 Forstå designmetoden grådighet

Lokalt optimale valg. Problemet ligger i å vise at den endelige løsningen er optimal, altså at problemet har grådighetsegenskapen. Trenger optimal delstruktur, grådighetsegenskapen og at grådig valg etterlater ett delproblem.

!G2 Forstå grådighetsegenskapen (the greedy-choice property)

Grådighetsegenskapen er å kunne bygge en globalt optimal løsning ved å ta lokalt optimale valg (grådige valg). Det er grådighetsegenskapen som avgjør om vi lager en grådig eller DP løsning.

G3 Forstå eksemplene aktivitet-utvelgelse og det kontinuerlige ryggsekkproblemet

Det kontinuerlige ryggsekkproblemet er greit. Regn ut verdi/vekt for alle objektene, sorter fra høyest til lavest og ta fra starten av lista (det grådige valget) til sekken er full. Kjøretid $O(n \lg(n))$-

Her er et bevis (veldig inspirert av boka) for at aktivitet-utvelgelse har grådighets-egenskapen:

La S_k være et ikke-tomt delproblem og a_m være aktiviteten i S_k med tidligst sluttid. La A_k være en optimal løsning på aktivitet-utvelgelse og a_j være aktiviteten i A_k med tidligst sluttid. Hvis $a_k = a_j$ er vi ferdig, så anta at de ikke er like. Vi vet at $f_j \leq f_k$ og at $f_k < s_i$ for enhver aktivitet $a_i \in A_k$ $(i \neq k)$ fra antagelsene våre, som impliserer at å bytte ut a_k med a_j i A_k fortsatt bevarer egenskapen at aktivitene i A_k er disjunkte. Spessielt vet vi nå at det grådige valget er den del av en optimal løsning, uavhengig av delproblemstørrelse. Det betyr at aktivitet-utvelgelse har grådighetsegenskapen. □ ;)

5.8 Forelesning 8

H1 Forstå hvordan grafer kan implementeres

Ved blant annet nabolister og nabomatriser. Står forklart under **Datastukturer** (2.10)

H2 Forstå BFS, også for å finne korteste vei uten vekter

BFS utforsker alle noder med avstand i fra s før den utforsker noen med avstand $i + 1$. Ved første iterasjon utforsker vi alle $v \in Adj.s$, vi har at $d(s, v) = \delta(s, v) = 1$ for alle disse. Anta at BFS har funnet korteste vei for alle noder v med $\delta(s, v) \leq k$, vi vil vise at ved neste iterasjon av BFS er alle noder med $\delta(s, v) \leq k + 1$ utforsket. La v være en vilkårlig node med $\delta(s, v) = k$, og $u \in Adj.v$ være en uutforsket node. Vi vet at $\delta(s, u) \leq \delta(s, v)+1$ fra Lemma 22.1 (s.598), fra antagelsen vår er alle noder med minste avstand til s mindre eller lik k allerede oppdaget, dermed må

$$\delta(s, u) = \delta(s, u) + 1 = k + 1$$

Derfor finner BFS korteste vei til alle noder s kan nå.

H3 Forstå DFS, parentesteoremet og hvit-sti-teoremet

Parentesteoremet er ganske klart, ettersom $v.d$ settes på vei ned i rekusjonen og $v.f$ setter på vei opp (etter at den har bunnet ut). Det fører til at hvis $v.d$ bil satt før $u.d$ vil $v.f$ bli satt før $u.f$, gitt at de er i samme DFS tre. Hvis ikke de er i samme tre vil $v.f < u.d$ eller motsatt, og intervallene er disjunkte.

Hvit sti teoremet sier at at v er en etterkommer av u hvis og bare hvis det ved tidspunkt $u.d$, når søket oppdager u, finnes en sti fra u til v med kun hvite noder. Siden $u.d \leq v.d$, med likhet kun hvis $u = v$, vil stien i deres dybde-først tre vil være hvit på dette tidspunktet. Den andre veien følger av parentesteoremet.

H4 Forstå hvordan DFS klassifiserer kanter

Når vi først oppdager kanten (u, v) sier fargen til noden v oss noe om kanten.

Er noden *hvit* er det en tre kant. Ligger i ett av trærne i DFS-skogen.

Er noden *grå* er det en bakoverkant. Kobler u med en forgjenger v i samme DFS-tre.

Er noden *svart* er det enten en en foroverkant, som kobler u med en etterkommer v, eller en krysskant, som er alle andre kanter.

H6 Forstå hvordan DFS kan implementeres med en stakk

Gjør som kallstakken.

H7 Forstå hva traverseringstrær (som bredde-først- og dybde-først-trær) er

!H8 Forstå traversering med vilkårlig prioritetskø

5.9 Forelesning 9

I1 Forstå skog-implementasjonen av disjunkte mengder (Connected-Components, Same-Component, Make-Set, Union, Link, Find-Set)

I2 Vite hva spenntrær og minimale spenntrær er

Gitt en graf $G = (V, E)$ er ett spenntre til G en asyklisk delgraf $S = (V, E')$. Det vil si at S er et tre som inneholder alle nodene til G. Hvis G er vektet, sier vi at S er et minamalt spenntre om den totale vekten av kantene er minst av alle mulige spenntrær av G.

I4 Forstå hvorfor lette kanter er trygge kanter

Et rett frem "cut and paste" argument. Anta vi har en mengde A som er inneholdt i et minimalt spenntre T av grafen G. La $(S, V-S)$ være et snitt som respekterer A, og la (u, v) være en lett kant som krysser $(S, V - S)$. Hvis (u, v) er en kant i T er vi ferdige, så anta at en annen kant (x, y) krysser $(S, V - S)$ og er i T. Ved å lage et nytt tre T' hvor alle kanter er like som i T, med untak av at (x, y) er byttet ut med (u, v) har vi at

$$w(T') = w(T) - w(x, y) + w(u, v) \leq w(T)$$

Minimaliteten til T tvinger ulikheten til å være likhet, og T' er også et minimalt spenntre. Det grådige valget er en del av den optimale løsnigen, med andre ord er lette kanter trygge kanter.

5.10 Forelesning 10

J2 Forstå strukturen til korteste-vei-problemet

Optimal delstruktur: Gitt en korteste sti $p = \langle v_o, \ldots, v_k \rangle$ mellom v_0 og v_k vil alle delstier mellom v_i og v_j $(0 \leq i < j \leq k)$ være korteste stier. Hvis vi antar at en av delstiene ikke er minimale kan vi bare bytte ut med den minmale og få en kortere sti mellom v_0 og v_k, men det er en selvmotsligelse.

J3 Forstå at negative sykler gir mening for korteste enkle vei (simple path)

Enkle veier inneholder ikke sykler, per def, dermed går det helt fint at det finnes negative sykler. De vil ikke kunne være en del av en løsning. Hvis vi ikke spesifiserer at veien skal være enkel blir det brått andre boller.

J4 Forstå at korteste enkle vei kan løses vha. lengste enkle vei og omvendt

J5 Forstå hvordan man kan representere et korteste-vei-tre

!J6 Forstå kant-slakking (edge relaxation) og Relax

Kant-slakking brukes til å forbedre det korteste vei estimatet $v.d$. Gitt en kant (u, v), hvis $v.d > u.d + w(u, v)$ oppdateres $v.d$ til å være lik $u.d + w(u, v)$. Det vil si at om estimatet på korteste vei til v er høyere enn estimatet på korteste vei til u + vekten til kanten fra u til v oppdateres estimatet. Dette funker fordi estimatet $v.d$ er en øvre skranke for alle noder v. Kjøretid for RELAX er $O(1)$.

J7 Forstå ulike egenskaper ved korteste veier og slakking

Forstår du disse bevisene er det veldig greit å forstå korrekthet av algoritmene i dette temaet.

Triangle inequality: For enhver kant $(u, v) \in E$ har vi $\delta(s, v) \leq \delta(s, u) + w(u, v)$.

Bevis: La p_v være en korteste sti fra s til v. Hvis u ligger på stien vet vi, siden korteste sti problemet har optimal delstruktur, at hvis vi fjerner v fra stien p_v sitter vi igjen med en korteste sti fra s til v. Skriver vi ut i symboler har vi da $\delta(s, v) = \delta(s, u) + w(u, v)$. Hvis u ikke ligger på en korteste vei fra s til v må $\delta(s, v) < \delta(s, u) + w(u, v)$, ellers ville en korteste sti fra s til u hvor vi legger til kanten (u, v) vært en korteste sti fra s til v som u ligger på. □

Upper-bound property: Vi har alltid at $v.d \geq \delta(s, v)$ for alle noder $v \in V$, og når $v.d = \delta(s, v)$ endre den aldri.

Bevis: Verdien $v.d$ starter ut som ∞ (med unntak av $s.d = 0$) og oppdateres med regelen: hvis $v.d > u.d + w(u, v)$ så settes $v.d = u.d + w(u, v)$, for en kant (u, v). Den eneste måten $v.d$ får en annen verdi enn ∞ er om en sti har blitt oppdatert, kant for kant fra $s = v_0$ til $v = v_k$, ved å pakke ut hvordan $v_k.d$ regnes ut får vi

$$v_k.d = v_{k-1}.d + w(v_{k-1}, v_k) = \cdots = \sum_{i=1}^{k} w(v_{i-i}, v_i)$$

som per def må være større eller lik $\delta(s, v_k)$. Når først $v.d = \delta(s, v)$ har vi fra trekantulikheten at $\delta(s, v) \leq \delta(s, u) + w(u, v)$ for enhver kant (u, v), og hvis betingelsen kan ikke være sann. □

No-path property: Hvis det ikke finnes en sti fra s til v vil $v.d = \infty$.

Bevis: Ikke-uendelige verdier flyter ut fra s langs kantene, hvis ikke v kan nås fra s vil heller ikke verdien oppdateres. □

Convergence property: Hvis $p \rightsquigarrow u \rightarrow v$ en en korteste vei i G for $u, v \in V$, og hvis $u.d = \delta(s, u)$ ved et tidspunkt før vi avslapper kanten (u, v), da er $v.d = \delta(s, v)$ i all tid fremover.

Bevis: Dette er jo bare Upper-bound property.

Path-relaxation property: Hvis $p\langle v_0, \ldots, v_k\rangle$ er en korteste vei fra $s = v_0$ til v_k, og vi avslapper kantene i p i rekkefølgen $(v_0, v_1), \ldots, (v_{k-1}, v_k)$, er $v_k.d = \delta(s, v_k)$. Denne egenskapen holder uavhengig av andre avslappninger, selv om de blandes med avslapningene til kantent i p.

Bevis: Ta en titt på omgjøringen av $v_k.d$ til en sum over, og sammenlikn det med definisjonen av korteste vei. Kombiner Upper-bound og Convergence property for siste del. □

Predecessor-subgraph property: Når først $v.d = \delta(s, v)$ for alle $v \in V$ er forgjenger delgrafen et tre med rot s.

Bevis: Når første korteste vei p mellom s og u fullføres vil ingen av de andre korteste veiene mellom s og u lagres i forgjengergrafen, fordi "hvis" betingelsen er en streng ulikhet. Dermed vil enhver vei fra s til en annen node i forgjengergrafen være unik, og det er ekvivalent med at forgjengergrafen er et tre. □

!J10 Forstå kobling mellom Dag-Shortest-Paths og dynamisk programmering

Her er jeg usikker, holla at me hvis du vet!

5.11 Forelesning 11

K1 Forstå forgjengerstrukturen for alle-til-alle-varianten av korteste vei-problemet (Print-All-Pairs-Shortest-Path)

5.12 Forelesning 12

L1 Kunne definere flytnett, flyt og maks-flyt-problemet

Flytnett: Et flytnett er en rettet graf $G = (U, V)$ hvor enhver kant $(u, v) \in E$ har en ikke-negativ *kapasitet* $c(u, v) \geq 0$ (kan defineres som en funksjon fra $V \times V$). Det kreves at G ikke har noen *antiparalelle* kanter, som vil si at høyst en av kantene (u, v) og (v, u) er i E (Den eneste grunnen er at vi vil definere en funksjon c_f senere, og antiparalelle linjer gjør ting vanskelig, hadde det ikke vært for denne kunne vi droppet kravet). To utvalgte noder s og t som refereres til som kilde og sluk respektivt. Vi antar at for alle $v \in V$ har vi $s \rightsquigarrow v \rightsquigarrow t$.

Flyt: En flyt i (flyutnettet) G er en funksjon $f : V \times V \to \mathbb{R}$ som oppfyller to krav:

Kapasitet begrensning: For alle $u, v \in V$ krever vi $0 \leq f(u, v) \leq c(u, v)$.

Flytbevaring: For alle $u, v \in V - \{s, t\}$ krever vi

$$\sum_{v \in V} f(v, u) = \sum_{v \in V} f(u, v)$$

Hvis $(u, v) \notin V$ definerer vi at $f(u, v) = 0$.

L2 Kunne håndtere antiparallelle kanter og flere kilder og sluk

Hvis en graf G ellers tilfrdsstiller kravene for å være et flytnett, men har antiparalelle kanter (u, v) og (v, u) introduserer vi en ny node v' og bytter ut (u, v) med (u, v') og (v', v). Vi setter kapasiteten til de to nye kantene lik kapasiteten til kanten de erstatter.

Hvis et flytnett har fler kilder eller fler sluk introduserer vi en ny *superkilde* og et nytt *supersluk*, bege med uendelig kapasitet. Superkilden har kanter til alle kildene og alle slukene har kanter til supersluket. En flyt i det nye flytnettet er ekvivalent med en flyt i det orginale flytnettet.

!L3 Kunne definere restnettet til et flytnett med en gitt flyt

Gitt et flytnett G og en flyt f defineres *restkapasiteten* slik

$$c_f(u, v) = \begin{cases} c(u, v) - f(u, v) & \text{hvis } (u, v) \in E \\ f(v, u) & \text{hvis } (v, u) \in E \\ 0 & \text{ellers} \end{cases}$$

Restnettet indusert av en flyt f er $G_f = (V, E_f)$ hvor

$$E_f = \{(u, v) \in V \times V | c_f(u, v) > 0\}$$

L4 Forstå hvordan man kan oppheve (cancel) flyt

Gitt en kant (u, v) i et flytnett kan man oppheve flyten langs kanten ved å øke flyten langs (v, u) i restnettet.

L5 Forstå hva en forøkende sti (augmenting path) er

En *forøkende sti* p er en enkel sti fra s til t i restnettet G_f. Vi kan øke flyten langs hver kant i en forøkende sti med maksimalt *restkapasiteten til* p, som er gitt ved

$$c_f(p) = \min\{c_f(u, v) | (u, v) \text{ er på } p\}$$

uten å bryte kapasitetsbegrensningen.

L6 Forstå hva snitt, snitt-kapasitet og minimalt snitt er

Et *snitt* (S, T) av et flytnett $G = (V, E)$ en en partisjon av V i S og $T = V - S$ slik at $s \in S$ og $t \in T$.

Kapasiteten til et snitt (S, T) er

$$c(S, T) = \sum_{u \in S} \sum_{v \in T} c(u, v)$$

Et *minimalt snitt* i et flytnett er et snitt hvor kapasiteten er minimal over alle snitt i nettet.

!L7 Forstå maks-flyt/min-snitt-teoremet

Teoremet forteller oss at garantert at f er en maksimal flyt dersom det ikke finnes fler forøkende stier i G_f. I tillegg sier det at den maksimale flyten vil ha verdi lik kapasiteten til et eller annet snitt (kombinert med et tidligere resultat vet vi at verdien er lik kapasiteten til **ethvert** snitt).

(1) $\implies$ (2): Dette gir mening, fordi vi vet at $|f \uparrow f_p| = |f| + |f_p| > |f|$.

(2) $\implies$ (3): Hvis det ikke finnes en forøkende sti fra s til t i G_f finnes det et snitt (S, T) slik at for $u \in S$ og $v \in T$ vil $c_f(u, v) = 0$. Fra definisjonen av $c_f(u, v)$ vet vi da at en av føldende tre ting er sant, for ethvert par u, v med $u \in S$ og $v \in T$.

- Hvis $(u, v) \in E$ så er $c(u, v) - f(u, v) = 0 \iff c(u, v) = f(u, v)$. **(1)**
- Hvis $(v, u) \in E$ så er $f(v, u) = 0$. **(2)**
- Eller så er bare $c_f(u, v) = 0$.

Hvis vi regner ut netto flyt (net flow) over snittet (S, T) får vi

$$f(S, T) \overset{\text{def}}{=} \sum_{u \in S} \sum_{v \in T} f(u, v) - \sum_{u \in S} \sum_{v \in T} f(v, u) = \overbrace{\sum_{u \in S} \sum_{v \in T} c(u, v)}^{\text{Fra (1)}} - \overbrace{\sum_{u \in S} \sum_{v \in T} 0}^{\text{Fra (2)}} \overset{\text{def}}{=} c(S, T)$$

Vi har et lemma som forteller oss at $|f| = f(S,T)$ for alle mulige snitt, derfor også for dette. Kan konkludere med at $|f| = c(S,T)$

(3) $\implies$ (1): Vi vet at $c(S,T)$ er en øvre skranke for verdien til flyten, vi antar likhet, derfor er f en maksimal flyt.

L9 Vite at Ford-Fulkerson med BFS kalles Edmonds-Karp-algoritmen

Vit dette.

L10 Forstå hvordan maks-flyt kan finne en maksimum bipartitt matching

En bipartitt graf G kan sees på som et mange kilder/mange sluk flytnett hvor alle kapasiteten er 1 og hvis vi velger oss et orientering på kantene. Legger til superkilde og supersluk og løser som et max-flyt problem. Vi har et resultat som sier at enhver flyt i dette flytnettet korresponderer til en matching, med $|f| = |M|$.

L11 Forstå heltallsteoremet (integrality theorem)

Forstå det.

5.13 Forelesning 13

M1 Forstå sammenhengen mellom optimerings- og beslutnings-problemer

Vi kan løse et optimeringsproblem med det tilhørende beslutningsproblemet ved å finne en begrensning på parameteren som skal optimeres. I tillegg vil beslutningsproblemet aldri være "vanskeligere" enn optimeringsproblem, fordi en løsning på optimeringsproblem gir en løsning på beslutningsproblemet (Vi har altså en reduksjon fra OP til BP). Det betyr at hvis BP er vanskelig er OP også vanskelig (Kontraposisjon av forrige setning).

M2 Forstå koding (encoding) av en instans

En koding er en funksjon fra en menge S til mengden av binære strenger $\{0,1\}^*$. Altså $e : S \to \{0,1\}^*$ er en koding. En koding av en instans er bare funksjonsverdien $e(s)$ for en $s \in S$. Et eksempel er $S = \{A, B, ..., Z\}$ og e er ASCII-kodingen, da er $e(A) = 1000001$.

M3 Forstå hvorfor løsningen på det binære ryggsekkproblemet ikke er polynomisk

Det er polynomisk som en funkjson av n og W (kapasiteten til sekken) men i uttrykket *polynomisk kjøretid* er det implisitt som en funksjon av input-størrelsen (i antall bits i forbindelse med NP − kompletthet). Inputstørrelsen er $\Theta(n \lg(n) + \lg(W))$. Hvis vi kaller antall bits i W for m har vi at kjøretiden er $\Theta(n2^m)$ som er eksponentiell som en funksjon av m. Har det vi kaller *Pseudopolynomisk* kjøretid.

M4 Forstå forskjellen på konkrete og abstrakte problemer

I et abstrakt problem er instansmengden I en (vilkårlig) mengde, og et konkret problem er instansmengden $I = \{0,1\}^*$ (en spesifik mengde).

M5 Forstå representasjonen av beslutningsproblemer som formelle språk

Instansmengden til ethvert konkret problem Q er $\{0,1\}^*$. Et problem er unikt bestemt av delmengden av instanser som resulter i løsninger, altså kan et problem betraktes om et formelt språk L over $\{0,1\}^*$, som betyr at $L = \{x \in \{0,1\}^* | Q(x) = 1\}$.

M6 Forstå definisjonen av klassene P, NP og co-NP

$\mathrm{P} = \{L \subseteq \{0,1\}^* |$ det finnes en algoritme A som bestemmer L i polynomisk tid$\}$.

Det betyr P er mengden av alle besluttningsproblemer L, hvor det for ethvert ord $x \in \{0,1\}^*$ av lengde n finnes en algoritme A som avgjør om x ligger i L eller ikke, i polynomisk tid ($O(n^k)$).

En algoritme *bekrefter* (verifies) en streng x om det finnes et *sertifikat* y slik at $A(x,y) = 1$. En algortime bekrefter et språk om den bekrefter alle strengene i språket. Klassen NP er samlingen av språk som kan bekreftes i polynomisk tid. Mer presist er et språk L en del av NP hvis og bare hvis det finnes en to-input polynomisk-tid algoritme A og en konstant c slik at

$L = \{x \in \{0,1\}^* |$ det finnes et sertifikat y hvor $|y| = O(|x|^c)$ slik at $A(x,y) = 1\}$

Klassen $\mathrm{co-NP}$ er $\{L \,|\, \bar{L} \in \mathrm{NP}\}$, hvor strek over betyr komplement (i universet $\{0,1\}^*$). I ord er det problemene hvor det er enkelt å sjekke om en løsning er feil (og algoritmen gir et 0 svar).

M7 Forstå redusibilitets-relasjonen $\leq_p$

For besluttningsproblemer L_1 og L_2 har vi at $L_1 \leq_p L_2$ hvis L_2 er minst like "vanskelig" som L_1. Mer utfyllende har vi en polynomisk-tid reduksjon fra L_1 til L_2 som gir at:

L_2 kan løses i polynomisk tid $\implies$ L_1 kan løses i polynomisk tid.

Via kontraposisjon får vi:
L_1 kan ikke løses i polynomisk tid $\implies$ L_2 kan ikke løses i polynomisk tid.

!M8 Forstå definisjonen av NP-hardhet og NP-kompletthet

Et problem L er *NP-komplett* dersom $L \in \mathrm{NP}$ og $L' \leq_p L$ for alle $L' \in \mathrm{NP}$. Altså i NP og minst like hardt som alle andre problemer i NP.

Et problem er *NP-hardt* dersom det er minst like hardt som ethvert problem i NP.

M9 Forstå den konvensjonelle hypotesen om forholdet mellom P, NP og NPC

Spørsmålet er P $\stackrel{?}{=}$ NP. De aller fleste tror ikke det.

!M10 Forstå hvordan NP-kompletthet kan bevises ved én reduksjon

Har vi et problem Q vi vet er NP − komplett og reduserer det til et annet problem Q' i NP vet vi at $L \leq_p Q \leq_p Q'$ for alle $L \in$ NP.

!M11 Kjenne de NP-komplette problemene CIRCUIT-SAT, SAT, 3-CNF-SAT, CLIQUE, VERTEX-COVER, HAM-CYCLE, TSP og SUBSET-SUM

Kjenn til disse.

M12 Forstå at det binære ryggsekkproblemet er NP-hardt

Har med størrelsen til input å gjøre.

M13 Forstå at lengste enkle-vei-problemet er NP-hardt

Ekvivalent til HAM-CYCLE.

M14 Være i stand til å konstruere enkle NP-kompletthetsbevis

Vær det.

6 Oppgaver

Gjør noen!