

Innholdsfortegnelse

Grafer	2
<i>Rettete/urettede grafer.....</i>	<i>2</i>
<i>Veier/stier</i>	<i>2</i>
<i>Sykler.....</i>	<i>2</i>
<i>Grad og størrelse</i>	<i>2</i>
<i>Representasjon</i>	<i>3</i>
<i>Eulerkrets/Hamiltonsykler</i>	<i>3</i>
Topologisk sortering	3
Graftraversering	3
<i>Dybde/Bredde først søk</i>	<i>3</i>
O-notasjon	3
Binærsøk, O-notasjon, trær og binære søketrær.....	3
<i>Binærsøk</i>	<i>4</i>
<i>Algoritmeanalyse (hvilken algoritme skal vi velge).....</i>	<i>4</i>
<i>Trær</i>	<i>4</i>
<i>Binære søketrær.....</i>	<i>4</i>
Balanserte søketrær.....	4
<i>AVL-trær.....</i>	<i>4</i>
<i>Rød-svarte trær.....</i>	<i>5</i>
Prioritetskø, heaps og huffman-koding	5
<i>Prioritetskøer</i>	<i>5</i>
<i>Total ordning.....</i>	<i>5</i>
<i>Binære heaps</i>	<i>5</i>
<i>Huffman</i>	<i>6</i>
Sortering: Bubble, Selection, Insert, Heap	7
Sortering: Merge, Quick, Bucket, Radix	7
<i>Stabilitet.....</i>	<i>7</i>
<i>In-place</i>	<i>7</i>
Hashing.....	7
<i>Hashfunksjoner</i>	<i>8</i>
Korteste stier (Dijkstra, Bellman Ford)	8
Spenntre (Prim, Kruskals, Bruvkas).....	8

Bikonektivitet (sammenhengende grafer)	9
<i>K-sammenhengende grafer</i>	9
Avgjørelsesproblemer, P og NP	9
<i>P</i>	10
<i>NP</i>	10
NP-komplettethet, Polynomtidsreduksjoner	10
Hopcroft-Tarjan eksempel	10

Grafer

<https://www.uio.no/studier/emner/matnat/ifi/IN2010/h20/slides/grafer-definisjon.pdf>





Består av to mengder (V, E)

V – Mengde noder

E – Mengde kanter mellom nodene

$\{u, v\}$ betyr at det finnes en kant mellom node u og v

Vektet/uvektet, Parallelle kanter, (Enkle) løkker, Enkel graf

Vektet/uvektet	Kantene har en verdi/kostnad knyttet til seg	
Parallelle kanter	Flere enn én kant mellom to noder	
(Enkle) løkker	En kant fra en node til seg selv	
Enkel graf	Ingen løkker, ingen parallelle kanter, urettet, uvektet	

Rettete/urettede grafer

Rettete grafer skrives (u, v)

Urettete grafer skrives $\{u, v\}$ evt. (u, v) OG (v, u)

Veier/stier

Sti: En sekvens av noder i grafen forbundet av kanter, der ingen *noder* gjentas.

Vei: Ev sekvens av noder i grafen forbundet av kanter, der ingen *kanter* gjentas.

Sykler

En sykel er en sti i en graf med minst tre noder som forbinder første og siste node.

Grafer uten sykler er asykliske.

I rettede grafer må kantene peke «riktig» vei for at vi skal ha en sykel

Grad og størrelse

Graden til en node $\deg(v)$ forteller hvor mange kanter som er koblet til en node. For rettede grafer regner vi både inngrad og utgrad.

En komplett graf er en graf med kanter mellom hver node. Blir tilsammen $V*(V-1)/2$. Dermed er $|E| = O(V^2)$. Antall kanter er begrenset av antall noder, men ikke motsatt.

Representasjon

Grafer kan representeres på mange måter, f.eks ved objektorientering, nabomatrise eller nabolister. Se forelesningslides for info.

Eulerkrets/Hamiltonsykler

Topologisk sortering

<https://www.uio.no/studier/emner/matnat/ifi/IN2010/h20/slides/grafer-topologisksort.pdf>

En graf kan beskrive avhengigheten mellom ting, f.eks. i et tidsperspektiv. Topologisk sortering kan sortere noder etter avhengigheter. Bruker DAG (directed acyclic graphs).

For pseudokode, se forelesning.

Graftraversering

<https://www.uio.no/studier/emner/matnat/ifi/IN2010/h20/slides/graftroversering.pdf>

Det finnes mange forskjellige måter å gå gjennom elementene i en graf.

Dybde/Bredde først søk

Både dybde og bredde først søk besøker alle nabonodene av en startnode til alle nodene er besøkt en gang. Det tas ikke hensyn til vekt.

Dybde først traverserer rekursivt gjennom en sti av noder til den kommer til en slutt. Da traverserer den baklengs til den kommer til et kryss og fortsetter på en ny sti.

Bredde først lager en kø der alle ubesøkte nabonoder til nodene vi besøker legges til.

O-notasjon

<https://www.uio.no/studier/emner/matnat/ifi/IN2010/h20/slides/onotasjon-eksempler.pdf>

Binærsøk, O-notasjon, trær og binære søketrær

<https://www.uio.no/studier/emner/matnat/ifi/IN2010/h20/slides/uke-35.pdf>

Binærsøk

Avgjør om et element er i et array eller ikke. Er raskere enn rett frem søk, men arrayet må være ordnet. Slår opp på midten og ser om verdien vi finner er større eller mindre enn den verdien vi leter etter. Deretter søker vi igjen på høyere eller lavere halvdel på samme måte. $O(\log(n))$

Algoritmeanalyse (hvilken algoritme skal vi velge)

Vi vil velge over algoritme som tar kortest tid og bruker minst minne. Worst case vs. Average case. For å beregne average case må vi finne sannsynlighetsregning for mulige instanser. Noen algoritmer vil ha en veldig dårlig worst case, men som i de aller fleste tilfeller kjører på en mye kortere tid. Da kan average case være nyttig.

Trær

Bra for hierarkis struktur. Alle lister er trær, men ikke alle trær er lister. Et tre kan ha flere nestepekere. Et tre tillater ikke sykler. En node har kun en foreldrenode.

Øverste noden (startnoden) er roten. Vi har søskennoder (samme plan), foreldrenoder og barnenoder. De nederste nodene (uten barn) er løvnoder eller eksterne noder. Nodene mellom rotnoden og løvnoden(e) er interne noder.

Alle trær har dybde, dybden til rotnoden er 0. Et tomt tre har dybde -1.

Høyden til et tre er lik dybden til den dypeste løvnoden.

Når man skal traversere gjennom et tre kan vi bruke:

Preorder: Utfører operasjon på seg selv først, så barnet.

Postorder: utfører operasjonen på barnet så seg selv.

Man kan kopiere et tre ved preordre, og slette et tre med postorder

Binære søketrær

Gjør binærsøk enkelt, og støtter effektiv innsettning og sletting.

I et binærtre har hver node maksimalt to barn (høyre og venstre).

Hver node i et binært søketre har større verdi i sitt høyre barn, og mindre verdi i sitt venstre barn. (evnt \leq og \geq med duplikater). Med et balansert binært søketre kan man bruke teorien bak binærsøk til å finne elementer i treet, og vil få en worst case på $O(\log(n))$

For forskjellige funksjoner til binærtre (sletting, innsettning, finn minste osv.) se forelesing.

Balanserte søketrær

<https://www.uio.no/studier/emner/matnat/ifi/IN2010/h20/slides/uke-36.pdf>

AVL-trær

Et AVL-tre er et binært søketre der høydeforskjellen mellom høyre og venstre subtre er mindre eller lik 1. Altså et balansert tre. Det må gjelde etter innsettning og sletting.

Vi ønsker at alle noder har sin egen høyde.

Etter hver innsettning eller sletting vil vi utføre en balanseringsalgoritme for å holde treet konstant balansert. For algoritme for innsettning, balansering og sletting se forelesning.

Rød-svarte trær

Også et balansert søketre:

Likheter med AVL:

- Selvb balanserende søketær
- $O(\log(n))$ på innsettning, sletting og oppslag.
- Begge bruker rotasjon for å beholde kravet om balanse

Forskjeller mellom rød-svarte- og AVL-trær:

- Rød-svarte trær har mindre krav om balanse
- Rød-svarte trær bruker mindre minne siden vi ikke trenger å lagre høyde
- Rød-svarte trær bruker færre rotasjoner

Hvem er best?

Rød-svarte trær er raskere enn AVL-trær når innsettning og sletting

forekommer ofte, til sammenligning med oppslag

- Dette er fordi rød-svarte trær trenger færre rotasjoner
- AVL-trær er raskere enn rød-svarte trær når oppslag forekommer ofte, til sammenligning med innsettning og sletting
- Dette er fordi AVL-trær er mer balanserte

For logikk bak rød-svarte trær, se forelesning.

Prioritetskø, heaps og huffman-koding

<https://www.uio.no/studier/emner/matnat/ifi/IN2010/h20/slides/uke-37.pdf>

Prioritetskøer

Prioritetskø er et eksempel på en abstrakt datatype (ADT). Interface i Java. Eksempler på abstrakte datatyper er list, set og map.

Prioritetskøer er en betegnelse for samlede elementer som støtter:

Insert(e) (push)

Removemin/max (pop)

Heap er en slik datastruktur.

Total ordning – se slides

Binære heaps

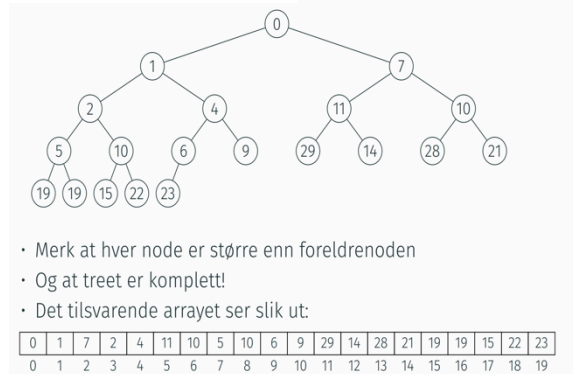
(min heap) En binær heap er et binærtrep som oppfyller at alle nodene som ikke er rotnoden er større enn foreldrenoden. Binærtreet må være komplett. Dvs det fylles opp fra venstre mot høyre(balansert). Den dypeste noden vil dermed alltid være lengst mot venstre.



$O(\log(n))$ på innsettning og sletting av minste, det er samme som balansert søketre.

Fordeler/forskjeller med heaps:

- Heaps støtter færre operasjoner og har en svakere invariant
- Heaps er komplette, så de er alltid balanserte
- De er mer balanserte enn både AVL- og rød-svarte trær
- Vi trenger ingen rotasjoner
- Kan implementeres effektivt med arrayer



- Merk at hver node er større enn foreldrenoden
- Og at treet er komplett!
- Det tilsvarende arrayet ser slik ut:

Idé bak innsetting: Ny node plasseres på neste ledige plass. Er verdien lavere enn foreldrenoden må de bytte plass. Dette skjer rekursivt.

Idé bak sletting: Fjern rotnoden, flytt den «siste» noden til rotnodenplassen. Er den større enn et barn, flyttes den ned rekursivt.

Binære heaps kan implementeres som både tre og array, men array er det raskeste, og krever mindre minne.

For implementasjon av heap som array, se forelesningen fra slide 15

Huffman

Brukes for å komprimere data. Hvert symbol blir representert med en bitstreng (tar mindre plass). En slik mapping mellom symboler og bitstrenger kalles enkoding. Bitstrenger kalles kodeord.

Med fast lengde (n) kan vi representere 2^n forskjellige symboler. Har du m antall symboler, må $\log_2(m) \leq n$ for å kunne representere alle symbolene.

Noe av ideen bak huffman er at noen symboler forekommer oftere enn andre. Da kan vi gi disse symbolene kortere bitstrenger, og de sjeldne symbolene får lengre bitstrenger.

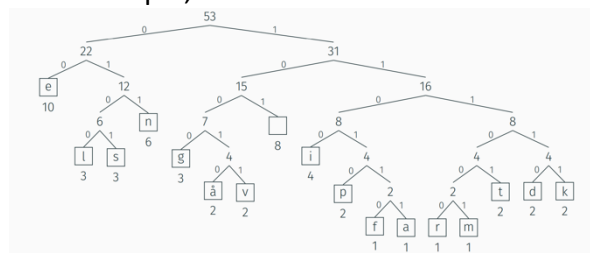
Den mest optimale huffman-komprimeringen får du hvis du bruker frekvensen av symboler i den gitte strengen du jobber med for å velge lengden på bitstrengen.

Huffman har ingen loss.

Variabel lengde gir altså bedre komprimering. Men det er viktig å vite når et symbol starter og når det slutter. Det skjes ved at ingen symboler kan representeres med et prefiks av en annen representasjon.

Har vi 010 som en kode, kan ikke 0101 være en annen kode (vet ikke når vi starter/slutter)

For eksempel, se slides s.40



Et huffman-tre bruker frekvenstabell til å plasere bokstaver/symbol som løvnoder i en heap. Alle pekerene er vektet med enten 0 eller 1. For å finne bitstrengen til et symbol følger du pekerene ned til symbolet. Tallene på veien blir da bitstrengen. Se mer slides s.42

Sortering: Bubble, Selection, Insert, Heap

<https://www.uio.no/studier/emner/matnat/ifi/IN2010/h20/slides/uke-42.pdf>

For psedukode, se slides.

Sortering er hensiktsmessig for å samle ting som hører sammen, finne like ting, søking i sorterte array er mye raskere.

Ingen sammenligningsalgoritme kan være raskere enn $O(n \cdot \log(n))$

Kjørtid til algoritmene:

Bubble sort, selection sort, Insert sort – $O(n^2)$

Heap sort – $O(n \cdot \log(n))$

Sortering: Merge, Quick, Bucket, Radix

<https://www.uio.no/studier/emner/matnat/ifi/IN2010/h20/slides/uke-43.pdf>

Stabilitet

Vi sorterer ofte på nøkler. F.eks. i personobjekter kan navn være nøkkelen

En sorteringsalgoritme er stabil om:

- For alle elementer x, y med samme nøkkel k .

Om x kommer før y før sorteringen, skal x komme før y etter sorteringen.

Hvorvidt en algoritme er stabil er noen ganger implementasjonsavhengig.

In-place

En algoritme er in-place dersom den ikke bruker ekstra datatyper. Å mellomlagre data i andre datatyper er ikke in-place

Det betyr i kontekst av sorteringsalgoritmen at den bruker $O(1)$ minne.

Både bubble, selection, insert og heap sort er alle in-place (heap-sort kan være ikke in-place om den mellomlagrer i en heap)

For detaljer rundt algoritmene, se slides

Kjøretidsanalyse for algoritmene (worst case):

Quicksort – $O(n^2)$

Mergesort – $O(n \cdot \log(n))$

Bucket sort – $O(N + n)$ (denne algoritmen krever at vi har informasjon til å sortere data i bønner, f.eks. kort som har 4 kortsorter.)

Radix sort – $O(n)$

Hashing

<https://www.uio.no/studier/emner/matnat/ifi/IN2010/h20/slides/uke-44.pdf>

Hashing brukes for å oppnå effektivt oppslag, innseting og sletting.
Map assosierer en nøkkel med en verdi.

Hashfunksjoner

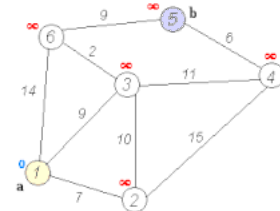
Må gi samme output for likt input hver gang.
Den må ikke gi lik output for ulike inputs.

Rehashing er å lage et nytt og større array og sette inn alle elementene fra forrige array.
Worst case for hash maps er $O(n)$ for alle operasjoner.

Korteste stier (Dijkstra, Bellman Ford)

<https://www.uio.no/studier/emner/matnat/ifi/IN2010/h20/slides/uke39-kortestestier.pdf>

Er grafen uvektet kan vi bruke BFS for å finne korteste sti
Dijkstra brukes med kanter med positiv vekt.
Dijkstra vil starte med en node, setter verdiene til alle andre noder til uendelig.



Bellman Ford brukes med kanter med både positiv og negativ vekt.

Med en negativ sykel vil det ikke finnes en korteste sti, men kan oppdage negative sykler med noen «smarte triks»

Har vi en sti med V antall noder, vil den ha $V-1$ antall kanter, ellers er det en sykel.

En negativ sykel mellom x antall noder tar minst x ganger for å oppdage

Dijkstra har worst case på $O(|E| \cdot \log(|V|))$

Bellman Ford har worst case på $O(|E| + |V|)$

Bellman Ford er altså tregere, så om man vet at man ikke har negative vekter er Dijkstra bedre.

Spennetre (Prim, Kruskals, Bruvkas)

<https://www.uio.no/studier/emner/matnat/ifi/IN2010/h20/slides/uke39-spenntreer.pdf>

En graf $G = (V, E)$ er sammenhengende om det finnes en sti mellom hvert par av noder. Altså alle noder kan nå hverandre.

Spennetrer er sammenhengende grafer uten sykler. Inneholder en graf flere kanter en noder har den en sykel og er ikke et tre. Et spennetre har $V-1$ kanter.

Er grafen uvektet kan vi bruke BFS eller DFS til å få spennetre

De neste algoritmene er grådige og vil ta inn en graf og returnere det billigste spennetreet.

For pseudokode se slides.

Sammenligning

Alle har samme verste tilfellet analyse...så hvilken algoritme skal man velge?

- på tynne grafer er Kruskal i praksis ofte raskere
- hvis man har tilgang til kantene sortert etter vekt: Kruskal raskere
- det er mulig å implementere Prim slik at den blir raskere (med datastrukturer ikke dekket av pensumet), med verste tilfellet $O(|E| + |V| \log(|V|))$. Da er den mye bedre for tette grafer. Dette er ikke pensum
- Borůvka er lett å parallelisere

Bikonektivitet (sammenhengende grafer)

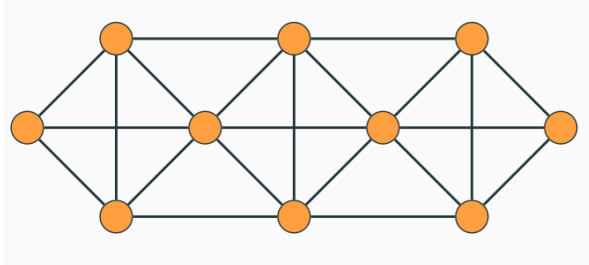
<https://www.uio.no/studier/emner/matnat/ifi/IN2010/h20/slides/uke41-bikonnektivitet-handout.pdf>

En graf $G = (V, E)$ er sammenhengende om det finnes en sti mellom hvert par av noder. Altså alle noder kan nå hverandre.

For å sjekke om en graf er sammenhengende kan man kjøre BFS/DFS og se om vi har ubesøkte noder. $O(V + E)$

K-sammenhengende grafer

En graf er k-sammenhengende om man må fjerne minst k noder for at grafen ikke skal være sammenhengende lenger. Denne grafen er 3-sammenhengende.



En graf er biconnected (2-sammenhengende) om det finnes to noder som er festet bare med en node mellom.

Noder som kan fjernes for at grafen ikke skal være sammenhengende kalles separasjonsnoder.

Se mer om sammenhengende grafer i slides.

Avgjørelsesproblemer, P og NP

<https://www.uio.no/studier/emner/matnat/ifi/IN2010/h20/slides/uke45-pnp.pdf>

Avgjørelsesproblemer er ja/nei-spørsmål. Inputs kalles instanser.

Ja-instanser gir ja som output, nei-instanser gir nei.

P – løsninger kan bli effektivt beregnet

NP – løsninger kan bli effektivt verifisert.

«effektivt» = polynomisk tid.

Sorted	
Instans:	En liste L
Spørsmål:	Er L sortert?

P

Alle problemer der det finnes en algoritme som kan løse det på $O(n^k)$ tid.

Eks:

- sorteringsalgoritmene løser

Sort	
Instans:	To lister L_1 og L_2
Spørsmål:	Består L_2 av elementene fra L_1 i sortert rekkefølge?

- minimale spenntreer:

MST-k	
Instans:	En graf G , et tall k
Spørsmål:	Finnes det et spenntre i G som koster mindre enn k ?

NP

NP er alle problemer som kan verifiseres med et gitt sertifikat på $O(n^k)$ tid. Returnerer ja eller nei.

P er en delmengde av NP. Alle problemer som kan løses i polynomisk tid kan verifiseres i det. Man kan løse NP-problemer med å kjøre samme sjekk som vi ville med P-problemer.

NP-kompletthet, Polynomtidsreduksjoner

<https://www.uio.no/studier/emner/matnat/ifi/IN2010/h20/slides/uke46-reduksjon.pdf>

Reduksjon er å ta et problem og oversette det til et annet som tar samme instanser og gir samme svar. F.eks. mobiltårnoppgraden. Startes som et kostnad/byggeproblem og ble løst ved å gjøre det til et grafproblem.

Om B er minst like vanskelig som A er $B \geq A$ (B og A er før og etter reduksjon)

Alle polynomtidsproblemer kan reduseres til hverandre. Alle er dermed like vanskelige. SAT er vanskeligere enn alle problemer i NP, men er selv i NP. Det er altså komplett

Hopcroft-Tarjan eksempel

<https://www.uio.no/studier/emner/matnat/ifi/IN2010/h20/slides/uke41-bik-eksempel.pdf>