Minimale eliminasjonstrær for parallell Cholesky-faktorisering

Hovedfagsoppgave i informatikk, retning databehandling
av

Fredrik Manne

September 1989
Institutt for Informatikk
Universitetet i Bergen

Forord

Arbeidet med denne hovedfagsoppgaven ble utført ved Institutt for Informatikk, Universitetet i Bergen, i tidsrommet september 1987 til september 1989.

Jeg vil gjerne takke de som jeg har hatt gleden av å samarbeide med under gjennomføringen av denne hovedfagsoppgaven.

Min veileder professor Bengt Aspvall har inspirert meg og kommet med konstruktive komentarer under hele arbeidet. Han har dessuten holt motet oppe når jeg selv har tvilt på om det ville bli noe av denne oppgaven.
Professor John Gilbert introduserte meg til parallell Cholesky-faktorisering og eliminasjonstrær, og var med å stake ut retningen for mitt arbeid.
Stipendiat Trond-Henning Olesen har bidratt med nyttige samtaler og nøyaktig korrekturlesing.

Tilslutt vil jeg rette en takk til alle studentene som var med å planlegge og å gjennomføre hovedfagstudentenes ekskursjon til USA sommeren 1988.

Bergen 16.09.89.

Fredrik Manne

Innhold

1 Innledning **1**

1.1 Graf-notasjon 2

1.2 Cholesky-faktorisering og grafer 4

1.3 NP-komplette problem 6

2 Eliminasjonstrær **7**

2.1 Parallell Cholesky-faktorisering og eliminasjonstrær 9

3 Minimale eliminasjonstrær **11**

3.1 Minimale eliminasjonstrær til delgrafer 11

3.2 En nedre grense på høyden til et minimalt eliminasjonstre . . . 13

3.3 Nøstet oppdeling og minimale eliminasjonstrær 17

4 En klasse av algoritmer for å redusere høyden til et eliminasjonstre **27**

4.1 Minimale-kuttsett algoritmen 27

4.2 Algoritmene til Liu og Hafsteinsson 30

4.3 Sammenligning av algoritmene 33

4.4 Et resultat om permuteringer av eliminasjonstrær 35

4.5 Nederste-først algoritmen 38

5 Minimale eliminasjonstrær til trær **43**

5.1 Balansering av eliminasjonstrær 43

5.2 Algoritme 50

5.3 En algoritme som finner et eliminasjonstre av høyde $\lfloor logn \rfloor$ til et tre 58

5.4 Analyse 62

5.5 Eksperimentelle resultater 65

6 Konklusjon **69**
6.1 Sammendrag 69
6.2 Åpne problem 70
Appendiks A Komplette *k*-nære trær **71**
Appendiks B Stirlings approksimasjon **73**
Bibliografi **75**

1 Innledning

Vi skal i denne hovedfagsoppgaven se på hvordan vi permuterer en symmetrisk matrise A slik at vi kan løse ligningsystemet $Ax = b$ raskest mulig ved hjelp av parallell Cholesky-faktorisering. Vi gir her en kort innføring til problemet samt en beskrivelse av hvordan denne hovedfagsoppgaven er disponert.

Etterhvert som man har fått kraftigere og hurtigere datamaskiner har størrelsen og kompleksiteten på beregningene man ønsker å utføre vokst. Man har nå kommet så langt i utvikling av datamaskiner at lyshastigheten begynner å bli en begrensende faktor. Dette har ført til en stadig økende interesse for parallelle datamaskiner og distribuerte systemer. De har den fordelen at man har flere regne-enheter som samtidig arbeider med hver sin del av problemet man ønsker å løse.

Et problem som ofte dukker opp i en rekke sammenhenger, både industrielt og forskningsmessig, er løsning av linjære ligningsystemer av typen $Ax = b$. Dette er et problem som egner seg godt for løsning ved hjelp av parallelle metoder. Dersom A er glissen symmetrisk, positiv definit er den raskeste måten å løse $Ax = b$ på en en-prosessor maskin ved hjelp av Cholesky-faktorisering. Cholesky-faktorisering foregår vanligvis i fire steg: (1) Dersom vi ønsker å endre på strukturen til A finner vi først en symmetrisk permutasjon av A. (2) Vi finner så den symbolske faktoriseringen av A i to triangulære matriser L og L^T slik at $A = L * L^T$. (3) Når vi kjenner strukturen til L regner vi ut de numeriske verdiene til ikke-null elementene i L. (4) Til slutt løser vi ligningsystemene $Ly = b$ og $L^T x = y$.

Det har vært utført mye arbeid med å finne gode parallelle algoritmer for Cholesky-faktorisering, se f.eks. Liu [11] [9]. Det har vist seg at en faktor som begrenser hvor lang tid slike algoritmer tar å utføre er høyden til *eliminasjonstreet* til A. Eliminasjonstreet er en rettet graf som gjenspeiler de data-avhengighetene som oppstår når vi utfører Cholesky-faktorisering. Vi kan imidlertid forandre strukturen og dermed også høyden til eliminasjonstreet gjennom å foreta symmetriske permutasjoner av A.

I denne hovedfagsoppgaven skal vi se på hvordan vi kan permutere A slik at vi får et så lavt eliminasjonstre som mulig, et såkalt *minimalt eliminasjonstre*.

Vi skal i dette kapittelet presentere den notasjonen som vil bli brukt i denne hovedfagsoppgaven. Vi skal også vise hvordan vi kan representere plasseringen av ikke-null elementene i A ved hjelp av en graf. Dette gjør at vi kan se på

problemet å finne permuteringer av A som gir lave eliminasjonstrær, som et grafteoretisk problem.

Vi skal i kapittel 2 forklare hvordan vi finner eliminasjonstreet til A, og hvordan høyden til det virker inn på kjøretiden til algoritmer for parallell Cholesky-faktorisering.

I kapittel 3 viser vi hvordan vi kan finne en nedre grense for høyden til et minimalt eliminasjonstre. Vi viser også at det alltid eksisterer en permutasjon av A gitt ved en fremgangsmåte kalt *nøstet oppdeling* som gir et minimalt eliminasjonstre til A. Utifra dette resultatet utvikler vi en ny algoritme i kapittel 4 for å redusere høyden til et eliminasjonstre. Denne algoritmen sammenligner vi med to andre algoritmer med samme foremål fra henholdsvis Liu [10] [11] og Hafsteinsson [5]. Vi viser at alle tre algoritmene tilhører en felles klasse av algoritmer som baserer seg på et resultat om hvordan man kan omordne eliminasjonstrær. Ut fra dette resultatet utvikler vi så enda en ny algoritme for å redusere høyden til et eliminasjonstre.

I kapittel 5 viser vi en algoritme som finner et minimalt eliminasjonstre til alle matriser som har grafer som er trær. Analysen av denne algoritmen gir at den kan ha kjøretid $n^{log(logn)}$. Da dette trolig er et overestimat har vi tatt med en del eksperimentelle resultater som viser at vi aldri kommer i nærheten av en slik kjøretid.

Kapittel 6 inneholder en oppsummering av oppgaven samt en del problemer som fremdeles står igjen å løse.

1.1 Graf-notasjon

Her presenteres graf-notasjonen som vil bli brukt i denne hovedfagsoppgaven. Notasjonen er stort sett den samme som i Ph.D. avhandlingene til Hafsteinsson [5] og Zmijewski [19]. Der det ikke finnes naturlige norske oversettelser av den engelske notasjonen er det laget egne norske termer. I disse tilfellene står den engelske notasjonen i parantes etter den norske definisjonen.

En *graf* $G = (V, E)$ består av en *nodemengde* V og en *kantmengde* E. Vi skriver $V(G)$ og $E(G)$ når vi vil gjøre det klart hvilken graf det refereres til. Antall noder i en graf betegnes med n og antall kanter med m.

En *kant* er et uordnet par av noder (v, w). Nodene v og w er *naboer* dersom $(v, w) \in E$. *Gradtallet* til en node v er summen av antall naboer til v.

En graf $G' = (V', E')$ er en *delgraf* til G dersom $V' \subseteq V$ og $E' \subseteq E$. Delgrafen G' er en *node-delgraf* dersom alle kanter $(x, y) \in E$ der $x, y \in V'$ så er også kanten $(x, y) \in E'$ (eng. G' is *induced* by V'). Dersom $S \subseteq V$, betegner vi med

$G - S$ node-delgrafen til G bestående av alle noder i $V - S$. Dersom $S = \{v\}$ vil vi skrive $G - v$ istedenfor $G - \{v\}$.

G er en *komplett graf* dersom alle nodene i G er naboer. Dersom en delgraf G' inneholder k noder og alle nodene er naboer, er G' en *k-klikk* (eng. *k-clique*), eller bare en *klikk*.

Vi vil i enkelte tilfeller forenkle notasjonen ved å identifisere en node v_j med dens indeks j.

En sti i en graf G mellom to noder v_1 og v_k er en sekvens av noder $v_1, v_2, ..., v_k$ for $k \geq 1$, slik at $(v_i, v_{i+1}) \in E$ for $1 \leq i < k$, og der alle noder er distinkte, med mulig unntak av v_1 og v_k. $Lengden$ av en sti er antall kanter i den.

En graf er *sammenhengende* dersom det er en sti fra enhver node til enhver annen node. En sammenhengende node-delgraf til G er *maksimal* dersom den ikke kan utvides med noen node fra G slik at node-delgrafen fremdeles er sammenhengende. De maksimale sammenhengende node-delgrafene til G kalles *sammenhengende komponenter* (eller bare *komponenter*).

Dersom G' er en delgraf til G slik at $V(G') = V(G)$, er G' en *utspennende graf* til G.

En $sykel$ (eng. *cycle*) er en sti av lengde ≥ 1 der $v_1 = v_k$. Et tre er en sammenhengende graf som ikke inneholder noen sykler. Nodene som har eksakt en nabo i et tre er *løv*. En graf med en eller flere sammenhengende komponenter, der hver sammenhengende komponent er et tre, utgjør en *skog*.

Et *kuttsett* til G er en delmengde av V slik at hvis mengden blir fjernet fra G, deles G i to eller flere sammenhengende komponenter.

I en *rettet graf* $G = (V, E)$ er hver kant et ordnet par $<v, w>$. Vi sier at kanten går $fra\ v\ til\ w$. *Ut-gradtallet* til en node v er antall kanter som går ut fra den. *Inn-gradtallet* til en node v er antall kanter som kommer inn til den.

Et *rettet tre* T er en rettet graf som ikke inneholder noen sykler, der eksakt en node har ut-gradtall 0 og alle andre noder har ut-gradtall 1. Noden som har ut-gradtall 0 er *roten* til T og betegnes med $rot(T)$.

Dersom kanten $<v, w>$ er i T, så er w *foreldre-noden* til v, og v er et *barn* til w. Vi betegner foreldre-noden til en node v med $p(v)$. Foreldre-noden til $rot(T)$ er definert som $rot(T)$. Nodene uten barn kalles *løv*.

Høyden til en node v i T er lengden av den lengste stien fra et løv til v og betegnes ved $h(v)$. Høyden til et tre er høyden til $rot(T)$ og betegnes med $h(T)$. Dersom det finnes en sti fra v til w, så er w en *forfader* til v, og v er en *etterkommer* av w.

Dybden til en node v er lengden av stien $v, ..., rot(T)$ i T.

En sti i T der alle noder med mulig unntak av den første noden har eksakt ett barn, er en *kjede* i T. Den første noden i en kjede vil bli betegnet som den *nederste* noden, og tilsvarende vil den siste noden i kjeden bli betegnet som den *øverste*. En *maksimal kjede* er en kjede av maksimal lengde i T. En *nederste kjede* er en kjede der den nederste noden i kjeden er et løv.

Et *deltre* T' er en node-delgraf til T bestående av en node v i T og alle dens etterkommere i T. Noden v er roten til T'. Vi skriver $T(v)$ for å betegne deltreet med v som rot.

En rettet graf bestående av et eller flere rettete trær utgjør en *rettet skog*.

Vi tar tilslutt i denne seksjonen med at $log_k n$ er logaritmen med base k. Dersom k er sløyfet angir $log n$ logaritmen med base 2. Den naturlige logaritmen betegnes med lnn.

1.2 Cholesky-faktorisering og grafer

I denne oppgaven skal vi se på de data-avhengigheter som oppstår når vi løser et ligningsystem $Ax = b$ v.h.a. Cholesky-faktorisering, der A er en $n \times n$ glissen, symmetrisk, positiv definit matrise.

Vi skal i denne seksjonen vise hvordan deler av Cholesky-faktorisering kan sees på som et graf-teoretisk problem. Cholesky-faktorisering foregår vanligvis i fire faser:

(1) Dersom vi først ønsker å endre på strukturen til A kan vi foreta en symmetrisk permutasjon av A slik at vi får en matrise A' som vi bruker videre. Matrisen A' er produktet av PAP^T, der P er konstruert v.h.a. rekkepermutasjoner av identitetsmatrisen. (2) Vi beregner så den symbolske faktoriseringen av A i $L * L^T$, der L er nedre triangulær. Matrisen L kalles for *Cholesky-faktoren* til A. (3) Når vi kjenner plasseringen av ikke-null elementene i L beregner vi de numeriske verdiene i L. (4) Til slutt kan vi nå løse de to ligningsystemene $Ly = b$ og $L^T x = y$.

George og Liu gir i [2] en mer utførlig presentasjon av Cholesky-faktorisering for glisne matriser.

At A er positiv definit, garanterer at A ikke har noen null-elementer på diagonalen. Siden A også er symmetrisk, kan vi representere strukturen til A ved å bruke en graf $G = (V, E)$, hvor V=$\{v_1, ..., v_n\}$ og $(v_i, v_j) \in E$ hvis og bare hvis $a_{ij} = a_{ji} \neq 0$ og $i \neq j$. Et eksempel på en matrise A og representasjonen av den som en graf G kan sees i figur 1.1. Vi ser at kantene i G representerer plasseringen av ikke-null elementene i A.

Figur 1.1 En symmetrisk matrise og representasjonen av den som en graf.

Vi finner som nevnt plasseringen til ikke-null elementene i den triangulære faktoren L før vi regner ut de numeriske verdiene. Matrisen $L + L^T$ er symmetrisk, og vi kan derfor representere plasseringene av ikke-null elementene i den v.h.a. en graf G^*. Vi ser at når vi kjenner G^*, så vet vi også plasseringen av ikke-null elementene i L. Følgende algoritme fra Parter [13] viser hvordan vi kan lage G^* fra G:

Start med $G^* = G$.

Sett alle noder umerkede.

For i = 1 til n

Legg til kanter til $E(G^*)$ slik at alle umerkede naboer til v_i i G^* danner en klikk.

Merk node v_i.

Vi betegner operasjonen å merke v_i i algoritmen ovenfor med å *eliminere* v_i. Kantene som blir lagt til G for å skape G^* kalles *fyllkanter*, og G^* kalles den *fylte grafen* til G.

Følgende resultat fra Rose, Tarjan og Lueker [16] beskriver hva som må være tilfredstilt for at det skal oppstå en fyllkant mellom nodene u og w.

Teorem 1.1

Det oppstår en fyllkant mellom u og w, hvis og bare hvis det finnes en sti $u = v_{r_1}, v_{r_2}, ..., v_{r_k} = w$ i G slik at $r_i < min(r_1, r_k)$ for $1 < i < k$. □

En *eliminasjonsordning* til G er en ordning av nodene i G som vi skriver som en en-til-en funksjon α: $V \rightarrow$ {1,2,...,n}. Grafen vi får gjennom å renummerere nodene i G med α betegnes ved G_α. Dersom en node har indeks j i G, så vil den ha indeks $\alpha(v_j)$ i G_α.

Når vi bruker identitetsordningen $\alpha(v_i) = i$ skriver vi bare G. I algoritmen ovenfor for å finne G^* har vi brukt identitetsordningen.

Siden G er grafen til A, kan vi se på α som en symmetrisk permutasjon av A, d.v.s. at hvis hver kolonne j i permutasjonsmatrisen P har sitt eneste ikke-null element i rekke $\alpha(v_j)$, så er G_α grafen til PAP^T. Det følger at det er ekvivalent å finne en eliminasjonsordning til G og å finne en symmetrisk permutasjon til A. Dette gjør at vi kan arbeide med strukturelle problemer på en matrise A v.h.a. graf-algoritmer på grafen til A.

1.3 NP-komplette problem

Da en del problem vi støter på i forbindelse med Cholesky-faktorisering, er med i klassen av NP-komplette problem, gir vi her en kort innføring i de vanligste begrepene vedrørende NP-komplette problem. Garey og Johnson gir i [1] en grundig gjennomgang av teorien for NP-komplette problem.

For at en funksjon f skal være en *polynomisk transformasjon* mellom to ja/nei-problem Γ og Π må den tilfredstille følgende to krav:

1. a er et ja-tilfelle av Γ hvis og bare hvis $f(a)$ er et ja-tilfelle av Π.
2. Funksjonen f er beregnbar i polynomisk tid.

Problem-klassen NP består av alle ja/nei-problem som en ikke-deterministisk Turing maskin kan løse i polynomisk tid. Dersom et problem Γ er med i NP, og det for ethvert annet problem i NP finnes en polynomisk transformasjon til Γ, er Γ med i klassen av NP-*komplette problem*. For å vise at et problem $\Gamma \in$ NP er NP-komplett, kan vi vise at det eksisterer en polynomisk transformasjon fra et kjent NP-komplett problem til Γ.

Et problem som er minst like vanskelig som et NP-komplett problem er NP-*hardt*.

2 Eliminasjonstrær

Vi skal her beskrive en data-modell kalt *eliminasjonstrær* som gjenspeiler en rekke av de data-avhengigheter som oppstår når vi vil løse et ligningsystem $Ax = b$ v.h.a. parallell Cholesky-faktorisering, der A er en symmetrisk, positiv definit matrise.

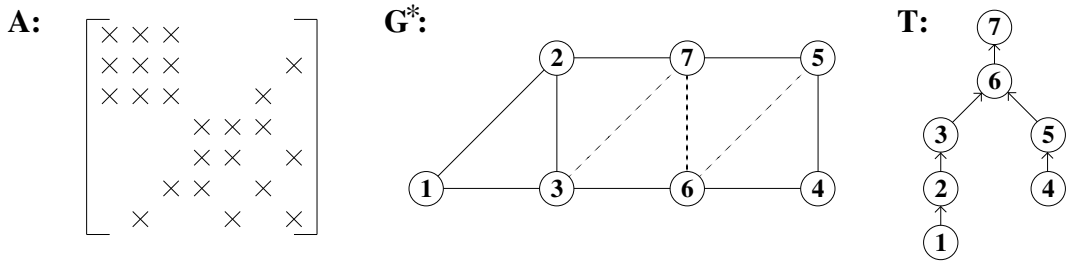
Eliminasjonstrær ble først introdusert av Schreiber [17] og har vist seg å være en viktig data-modell innenfor mange områder av faktorisering av glisne matriser. I denne hovedfagsoppgaven skal vi bruke eliminasjonstrær for å se på egenskaper ved symmetriske permutasjoner av A som gir kortest mulig parallell løsningstid av Cholesky-faktoriseringen.

Som beskrevet i seksjon 1.2 er det å foreta en symmetrisk permutasjon av A ekvivalent med å finne en eliminasjonsordning til G der G er grafen til A. Vi vil derfor bruke grafer når vi viser resultat og algoritmer.

Definisjon 2.1 La $A = (a_{ij})$ være en $n \times n$ symmetrisk, positiv definit matrise og la $L = (l_{ij})$ være dens Cholesky-faktor. *Eliminasjonstreet* til A er et rettet tre T, med noder 1,2,...,n og en kant <i, j> hvis og bare hvis $j = min\{k \mid l_{ki} \neq 0$ og $k > i\}$.

Vi ser at noden j er foreldre-noden til i i T dersom det første ikke-null elementet under diagonalen i kolonne i i L er i rekke j. Vi kan også bestemme T fra G^*: Noden j er foreldre-noden til i dersom $(j, i) \in V(G^*)$, $j > i$ og $j < k$ for hver nabo k til i i G^* der $k > i$.

Dersom T er en rettet skog med mer enn en komponent (og ikke et rettet tre), så er A reduserbar. Vi kan da permutere A slik at den er blokk-diagonal og hver blokk kan faktoriseres for seg. Vi antar derfor i resten av oppgaven at A er ikke-reduserbar, slik at grafen til A er sammenhengende, og T er et rettet tre. Figur 2.1 viser en matrise A, den fylte grafen G^* til A og tilhørende eliminasjonstre T. De kantene i G^* som ikke er med i G, er tegnet med stiplet linje.



Figur 2.1 En matrise A, den fylte grafen G^* til A og tilhørende eliminasjonstre T.

Vi vil i senere figurer ikke markere eksplisitt at kantene i T er rettete. Dersom $<i, j> \in E(T)$, vil node j være tegnet ovenfor node i.

Vi ser fra Definisjon 2.1 at alle kanter i T går fra lavere nummererte noder til høyere nummererte noder. Av dette følger at dersom j er en forfader til i i T, så er $j > i$.

Teorem 2.1, 2.2 og 2.3 er hentet fra Liu [12] og gjengis her uten bevis. De beskriver viktige egenskaper ved eliminasjonstrær som vi vil gjøre bruk av senere.

Teorem 2.1

Gitt en graf G med eliminasjonstre T. For enhver $x \in V(G)$ utgjør nodene i $T(x)$ en sammenhengende node-delgraf til G. □

Teorem 2.2

La $L = (l_{ij})$ være Cholesky-faktoren til en symmetrisk positiv definit matrise, og la T være eliminasjonstreet til den samme matrisen. Dersom $l_{ij} \neq 0$, $j < i$, så er node j en etterkommer av node i i T. □

Fra Teorem 2.2 kan vi utlede følgende korollar:

Korollar 2.2.1

Gitt en graf G med n noder. Dersom G inneholder en delgraf som er en k-klikk, vil enhver eliminasjonsordning gi et eliminasjonstre av høyde minst $k - 1$. □

Vi ser også fra Teorem 2.2 at for en gitt eliminasjonsordning α til en graf G vil eliminasjonstreet til G_α minst ha høyde $k - 1$, der k er antall noder i den største klikken i G^*_α.

Teorem 2.3

La x og y være to noder i G slik at $x < y$. Da eksisterer det en fyllkant $(x, y) \in E(G^*)$, hvis og bare hvis $(x, y) \notin E(G)$ og det eksisterer en node w der w er en etterkommer til x i T, slik at $(w, y) \in E(G)$. $\square$

Fra Teorem 2.3 ser vi at dersom w er en etterkommer til v i T og $(w, v) \in E(G)$, så vil alle nodene på stien $w, ..., v$ i T være naboer til v i G^*. Det følger derfor at en node $x \in V(G)$ er nabo i G^* til eksakt de av sine forfedre i T som nodene i $T(x)$ er nabo til i G.

2.1 Parallell Cholesky-faktorisering og eliminasjonstrær

Liu har i [9] diskutert bruken av eliminasjonstrær i parallell Cholesky-faktorisering. Han viser at høyden til eliminasjonstreet gir en begrensning på kjøretiden til en parallell algoritme for Cholesky-faktorisering. For å sitere Liu: "The height of the elimination tree represents an effective but crude measure of the amount of work in parallel elimination".

For et fast antall prosessorer vil også antall kanter i den fylte grafen G^* virke inn på kjøretiden til en parallell algoritme. Det er derfor ønskelig å finne en eliminasjonsordning som introduserer få fyllkanter samtidig som den gir et lavt eliminasjonstre. Dersom vi har minst n^2 prosessorer, der n er antall noder i grafen, så trenger ikke antall fyllkanter å virke inn på kjøretiden. Hafsteinsson har i [5] vist en algoritme med m prosessorer, der m er antall kanter i G^*, hvor m ikke innvirker på kjøretiden til algoritmen.

Det arbeidet som har vært gjort for å finne eliminasjonsordninger med lave eliminasjonstrær, har stort sett foregått langs to linjer. Den ene linjen baserer seg på først å finne en eliminasjonsordning α som gir få fyllkanter. Deretter finner man en ny eliminasjonsordning β som gir et lavest mulig eliminasjonstre til G^* over alle eliminasjonsordninger som gir samme mengde fyllkanter som α. Jess og Kees viser i [6] en algoritme som med utgangspunkt i G^* finner en eliminasjonsordning, med det laveste eliminasjonstreet over alle eliminasjons-ordninger, som bevarer mengden av fyllkanter.

Den andre linjen av arbeid har bestått i, direkte fra G, å finne eliminasjonsordninger som gir lave eliminasjonstrær. Leiserson og Lewis viser i [8] en slik metode.

Vi kommer i denne oppgaven til å arbeide videre på denne siste linjen, og se på eliminasjonsordninger som gir lave eliminasjonstrær, uavhengig av hvor mange fyllkanter som oppstår.

Vi skal nå som et eksempel vise hvordan høyden til eliminasjonstreet virker inn på et av stadiene av parallell Cholesky-faktorisering. Vi skal vise at tiden for å løse det triangulære ligningsystemet $Ly = b$ v.h.a. en parallell algoritme for foroversubstitusjon er begrenset av høyden til eliminasjonstreet. Vi trenger først følgende resultat:

Lemma 2.1
Node i er et løv i T hvis og bare hvis $l_{ij} = 0$ for alle $j < i$.

Bevis:
$\Rightarrow$ Anta $l_{ij} \neq 0$ for en j, $j < i$. Vi ser fra Teorem 2.2 at i må være en forfader til j, og i kan derfor ikke være et løv i T.

$\Leftarrow$ Anta at i ikke er et løv i T. Noden i må ha minst et barn j slik at $j < i$. Fra Definisjon 2.1 ser vi at dersom j er et barn til i må $l_{ij} \neq 0$ gjelde. $\square$

Vi ønsker å løse ut så mange ukjente y_i som mulig parallelt i ligning-systemet $Ly = x$ v.h.a. foroversubstitusjon. Fra L ser vi at dersom $l_{ij} \neq 0$, $j < i$ så må y_j løses før y_i. Verdien av y_j må deretter settes inn i rekke i av L før vi kan løse y_i. I første steg kan vi derfor bare finne eksakt de y_i der $l_{ij} = 0$ for alle $j < i$. Fra Lemma 2.1 ser vi at dette er eksakt alle løvene i T. Vi finner derfor først de ukjente hvis noder er løv i T, og substituerer verdien for disse inn i de ligningene de inngår i. Når dette er gjort, fjerner vi alle løvene fra T og løser for de nodene som nå er løv i T. Vi ser av dette at høyden til eliminasjonstreet gir en begrensning på hvor raskt vi kan løse ligningen $Ly = b$ v.h.a foroversubstitusjon.

3 Minimale eliminasjonstrær

Siden høyden på eliminasjonstreet er en begrensende faktor for kjøretiden til parallelle algoritmer for Cholesky-faktorisering, er det interessant å se på egenskaper og begrensninger til et minimalt eliminasjonstre. Vi definerer først hva vi mener med et minimalt eliminasjonstre.

Definisjon 3.1
Et *minimalt eliminasjonstre*[1] T_{min} til en graf G er et eliminasjonstre gitt ved en eliminasjonsordning α, slik at for enhver annen eliminasjonsordning β til G med eliminasjonstre T', så gjelder $h(T_{min}) \leq h(T')$.

I dette kapittelet skal vi bl.a. vise en ny nedre grense for høyden til et minimalt eliminasjonstre. Vi skal også se på en klasse av eliminasjonsordninger kalt nøstede oppdelingsordninger, som kan brukes for å finne minimale eliminasjonstrær. Først skal vi imidlertid se på et nytt resultat om minimale eliminasjonstrær til delgrafer.

3.1 Minimale eliminasjonstrær til delgrafer

Vi skal i denne seksjonen vise at en delgraf G' til en vilkårlig graf G alltid har et eliminasjonstre av høyde mindre eller lik høyden til et minimalt eliminasjonstre til G. Dette resultatet vil bli presentert som Teorem 3.2. Vi vet allerede fra Korollar 2.2.1 at dersom G inneholder en delgraf G' som er en k-klikk, så må et eliminasjonstre til G ha høyde *minst* $k-1$, mens et eliminasjonstre til G' vil ha høyde *eksakt* $k-1$.
For å vise at teoremet gjelder generelt, trenger vi først følgende definisjon:

Definisjon 3.2
En sti $w_{r_1}, w_{r_2}, ..., w_{r_l}$ er *monoton* dersom $r_i < r_{i+1}$ for $1 \leq i < l$.

Liu har i [11] vist følgende resultat om sammenhengen mellom en lengste monoton sti i den fylte grafen G^* og høyden til eliminasjonstreet til G:

[1] Den mest korrekte termen rent matematisk sett, ville vært et *minste eliminasjonstre* (Eng. *elimination tree of minimum height*.). Av lingvistiske årsaker vil vi likevel bruke betegnelsen *minimalt eliminasjonstre*.

Teorem 3.1

Høyden til et eliminasjonstre T til en graf G er lik lengden til den lengste monotone stien i G^*. $\square$

Fra Teorem 3.1 kan vi utlede det ønskede resultatet:

Teorem 3.2

Gitt to grafer G og H slik at H er en sammenhengende delgraf av G. La α være en eliminasjonsordning til G som gir et minimalt eliminasjonstre T_{min}. Da eksisterer det en eliminasjonsordning β til H som gir et eliminasjonstre T' slik at $h(T') \leq h(T_{min})$.

Bevis:

Velg β slik at den relative ordningen mellom nodene i H blir den samme som i G_α.

Fra Teorem 3.1 vet vi at $h(T')$ er lik lengden av den lengste monotone stien i H^*_β. Vi skal nå vise at denne stien har lengde $\leq h(T_{min})$.

Først viser vi at $H^*_\beta \subseteq G^*_\alpha$:

Vi vet fra Teorem 1.1 at det oppstår en fyllkant (u, w) i H^*_β hvis og bare hvis det finnes en sti i H_β mellom u og w slik at alle nodene på denne stien er nummerert lavere enn u og w. Siden vi har valgt β slik at den relative ordningen mellom nodene i H_β er den samme som i G_α, vil det også finnes en slik sti mellom u og w i G_α. Vi ser av dette at alle fyllkanter i H^*_β også forekommer i G^*_α, og det følger at $H^*_\beta \subseteq G^*_\alpha$

Vi betrakter nå en monoton sti v_{r_0},v_{r_1},v_{r_2},...,v_{r_l} i H^*_β. Kanten $(v_{r_j}, v_{r_{j+1}})$ finnes også i G^*_α. Siden v_{r_j} er nummerert før $v_{r_{j+1}}$ i β, vil v_{r_j} også være nummerert før $v_{r_{j+1}}$ i α. Vi ser av dette at dersom det finnes en monoton sti av lengde l i H^*_β, så finnes det også en monoton sti av lengde l i G^*_α. Dette medfører at den lengste monotone stien i H^*_β har lengde $\leq h(T_{min})$. Dermed følger det at $h(T') \leq h(T_{min})$. $\square$

Dersom H i Teorem 3.2 ikke er sammenhengende, så vil resultatet gjelde for hver sammenhengende komponent til H.

Vi ser fra Teorem 3.2 at dersom vi kan gi en nedre grense for høyden til et minimalt eliminasjonstre til en delgraf, så vil den samme grensen også gjelde for den opprinnelige grafen. Dette skal vi se videre på i seksjon 3.2.

Vi tar til slutt i denne seksjonen med følgende korollar til Teorem 3.2 som vi vil få bruk for i kapittel 5:

Korollar 3.2.1
Gitt en graf G med $h(T_{min}) = k$. Anta at en vilkårlig node $v \in V(G)$ blir fjernet fra G, slik at $G - v$ ikke nødvendigvis er sammenhengende. Dersom T' er et høyeste minimalt eliminasjonstre til $G - v$ så er $h(T') = k - 1$ eller k.

Bevis:
La α være en eliminasjonsordning som gir et minimalt eliminasjonstre til hver komponent av $G - v$, og la T' være et høyeste minimalt eliminasjonstre til $G - v$. Fra Teorem 2.1 vet vi at nodene i T' utgjør en sammenhengende node-delgraf til G. Det følger derfor fra Teorem 3.2 at $h(T') \leq k$.
Dersom $h(T') < k - 1$, så kunne vi ha funnet et eliminasjonstre til G av høyde $< k$ gjennom å eliminere v sist og nodene i $G - v$ som angitt av α. Dette er umulig da et minimalt eliminasjonstre til G har høyde k. Vi ser av dette at $k - 1 \leq h(T') \leq k$ må gjelde. □

3.2 En nedre grense på høyden til et minimalt eliminasjonstre

Det er interessant å søke etter teoretiske nedre grenser for høyden til et minimalt eliminasjonstre. Dersom vi kan gi en nedre grense som ligger tett opp til den virkelige verdien, får vi en god målestokk for om en gitt løsning er god eller ikke. Vi har allerede sett at dersom G inneholder en klikk på k noder, så gjelder $h(T_{min}) \geq k - 1$.
I denne seksjonen skal vi bruke Teorem 3.2 til å vise en ny nedre grense for $h(T_{min})$ (Teorem 3.3). Vi trenger først følgende resultat:

Lemma 3.1

Gitt en graf G med eliminasjonstre T. La G' være en sammenhengende delgraf til G med $n' > 1$ noder og la $v \in V(G')$ være den høyeste nummererte noden i G'. Da gjelder for alle $w \in V(G')$ at $w \in T(v)$.

Bevis:

Anta at ikke alle noder i $V(G')$ er med i $T(v)$. Siden G' er sammenhengende, må det eksistere to noder $x, y \in V(G')$ slik at x og y er naboer i G' og $x \in T(v)$ mens $y \notin T(v)$.

Siden v elimineres sist av nodene i G', kan y ikke ligge på stien $v, ..., rot(T)$ i T. Dette medfører at hverken x er en etterkommer til y eller y er en etterkommer til x. Samtidig vet vi at siden x og y er naboer i G så er $l_{xy} \neq 0$. Fra Teorem 2.2 ser vi at dette er umulig, og det følger at alle nodene i G' må ligge i $T(v)$. □

Ved hjelp av Lemma 3.1 kan vi nå utlede:

Lemma 3.2

Gitt en graf G. For enhver eliminasjonsordning α til G gjelder: Ingen node har flere barn i T enn den er nabo til i G.

Bevis:

Se på en node v som har minst et barn x i T som den ikke er nabo til i G. For at x skal ha v som foreldre-node i T, må det eksistere en fyllkant mellom x og v i G^*. For at denne fyllkanten skal oppstå, ser vi fra Teorem 1.1 at nodene på en sti fra x til v i G må elimineres før x. Siden stien går mellom x og v, må den inneholde minst en node som er nabo til v i G. Fra Lemma 3.1 følger det at alle nodene på denne stien må være etterkommere til x i T. Vi ser derfor at fordi x er barn til v i T, så er det en node blant naboene til v i G som ikke kan være barn til v. Siden hver nabo til v i G høyst kan gi opphav til et barn til v i T følger det at noden v ikke kan ha flere barn i T enn antall noder den er nabo til i G. □

Vi ser av Lemma 3.2 at vi aldri kan gjøre eliminasjonstreet til en graf lavere enn et komplett k-nært tre, der k er det maksimale gradtallet til noen node i grafen. Dersom en graf inneholder $n > 2$ noder gir dette oss:

$$\lfloor log_k(n(k-1)) \rfloor \leq h(T_{min})$$

Appendiks A gir en nærmere beskrivelse av komplette k-nære trær.

Fra Teorem 3.2 vet vi at en delgraf ikke kan ha et høyere minimalt eliminasjonstre enn et minimalt eliminasjonstre til den opprinnelige grafen. Dette sammen med Lemma 3.2 gir oss følgende nedre grense på høyden til et minimalt eliminasjonstre:

Teorem 3.3

Dersom G' er en sammenhengende delgraf til G der $n' = |V(G')|$ for $n' > 2$, og k' er det maksimale gradtallet til noen node i G', så gjelder følgende nedre grense på høyden til et minimalt eliminasjonstre til G:

$$\lfloor log_{k'}(n'(k'-1)) \rfloor \leq h(T_{min})$$

□

For fiksert k, vil verdien til $\lfloor log_k(n(k-1)) \rfloor$ i Teorem 3.3 øke med stigende n og med fiksert n vil verdien til $\lfloor log_k(n(k-1)) \rfloor$ avta med stigende k.

Vi kan i Teorem 3.3 begrense oss til å finne delgrafer til G som er trær uten at det vil gi oss en mindre tett nedre grense: Anta at G' ikke er et tre, og la H være et utspennende tre til G'. Da vil H inneholde like mange noder som G', og det maksimale gradtallet til noen node i H vil være mindre eller lik det maksimale gradtallet til noen node i G'.

La H være et utspennende tre til G' som har minimalt gradtall over alle utspennende trær til G'. Da vil H gi en tetteste nedre grense over alle utspennende grafer til G', på høyden til et minimalt eliminasjonstre til G. Å avgjøre om en graf har et utspennende tre der ingen node har gradtall $> k$ for en gitt $k > 2$, er imidlertid et NP-komplett problem (Garey og Johnson [1]).

Den tetteste grensen vi kan gi fra Teorem 3.3 på høyden til et minimalt eliminasjonstre til en graf G, får vi hvis det eksisterer en sti i G som inkluderer alle nodene i $V(G)$ (en Hamiltonsk sti). Dette gir oss følgende korollar:

Korollar 3.3.1

Dersom den lengste stien som ikke er sykel i en graf G har lengde l, så gjelder:

$$\lfloor log_2(l+1) \rfloor \leq h(T_{min})$$

□

Vi har fra Korollar 2.2.1 og Teorem 3.3 to ulike nedre grenser på høyden til et minimalt eliminasjonstre. Vi skal nå vise at ingen av grensene er strengt bedre

enn den andre. Dette gjør vi ved å vise at for ulike grafer vil hver av grensene gi den tetteste nedre grensen.

Den tetteste grensen på høyden til et minimalt eliminasjonstre til en komplett graf på n noder som vi kan gi v.h.a. Teorem 3.3 er $\lfloor log_2 n \rfloor$. Samtidig vet vi fra Korollar 2.2.1 at ethvert eliminasjonstre til en komplett graf vil ha høyde $n-1$. Dersom vi har en graf der den største klikken er forholdsvis stor, er det derfor trolig at Korollar 2.2.1 vil gi den tetteste nedre grensen på høyden til et minimalt eliminasjonstre.

For et tre G med minst 3 noder, vil Korollar 2.2.1 gi en nedre grense på 1 på høyden til et minimalt eliminasjonstre. Teorem 3.3 gir en nedre grensen på $\lfloor log_2(l+1) \rfloor$, der l er lengden av den lengste stien i G. Dersom $l \geq 3$, ser vi at Teorem 3.3 gir den tetteste nedre grensen på høyden til et minimalt eliminasjonstre til G. For en $k \times k$ rutenett-graf (se figur 3.3) vil Korollar 2.2.1 gi en nedre grense på 1 på høyden til et minimalt eliminasjonstre. Fra Teorem 3.3 får vi en nedre grense på $\lfloor 2log_2 k \rfloor$. Vi ser av dette at dersom den største klikken i G er liten, kan Teorem 3.3 gi en tettere nedre grense enn Korollar 2.2.1 på høyden til et minimalt eliminasjonstre. George og Liu gir i [2] den tetteste nedre grensen på $h(T_{min})$ til en rutenett-graf. De viser at $h(T_{min})$ er $\Theta(k)$.

For en gitt eliminasjonsordning til en graf G, kan vi gi en annen nedre grense på høyden til eliminasjonstreet til G enn den grensen Teorem 3.3 gir oss. Denne nye grensen vil være minst like bra, og kan være bedre enn den fra Teorem 3.3.

Korollar 3.3.2

Gitt en graf G med eliminasjonstre T. Dersom G' er en delgraf til den fylte grafen G^* der $n' = |V(G')|$ for $n' > 2$, og k' er det maksimale gradtallet til noen node i G', så gjelder følgende nedre grense på høyden til T:

$$\lfloor log_k\big(n'\big(k'-1\big)\big) \rfloor \leq h(T)$$

Bevis:

La T'_{min} være et minimalt eliminasjonstre til G^*. Da gjelder fra Teorem 3.3:

$$\lfloor log_k\big(n'\big(k'-1\big)\big) \rfloor \leq h\big(T'_{min}\big) \leq h(T)$$

□

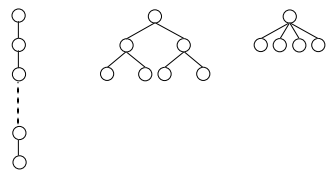
I Korollar 3.3.2 finner vi en delgraf til G^* (og ikke til G). Når vi skal bestemme G har vi derfor flere kanter å velge mellom enn i Teorem 3.3. Det gjør det mulig at Korollar 3.3.2 kan gi en tettere nedre grense på høyden til T.

Til slutt tar vi med at det ikke er gitt fra Lemma 3.2 at fordi en node v har k naboer i G, så eksisterer det en eliminasjonsordning slik at v har k barn i T. F.eks. vil ingen node i eliminasjonstreet til en komplett graf ha mer enn et barn. Vi skal i seksjon 3.3 se nærmere på hva som er betingelsen for at det skal oppstå forgrening i eliminasjonstreet.

3.3 Nøstet oppdeling og minimale eliminasjonstrær

Det eksisterer en rekke heuristikker for å finne eliminasjonsordninger som gir få fyllkanter i G^* (George og Liu [2]). Vi skal i denne seksjonen se nærmere på en slik heuristikk kalt *nøstet oppdeling* (eng. *nested dissection*) som også gir et lavt eliminasjonstre. Vi skal vise i Teorem 3.5 at det for en vilkårlig graf alltid eksisterer en eliminasjonsordning gitt ved nøstet oppdeling som gir et minimalt eliminasjonstre. Før vi gjør dette skal vi se nærmere på hvilke betingelser som må gjelde for at vi skal få forgrening i eliminasjonstreet.

For å få lavest mulig høyde på et eliminasjonstre T, må T forgrene seg utover i størst mulig grad. Dersom vi ikke får noen forgrening i T og alle nodene henger under hverandre, får vi høyde $n-1$. Klarer vi derimot å få til et eliminasjonstre der hver node v som ikke er et løv, har 2 barn og like mange noder i hvert deltre som henger fra v, får vi et tre med høyde $\lfloor log_2\, n \rfloor$. Henger alle nodene fra roten, får vi høyde 1.



Figur 3.1 Eliminasjonstrær av høyde $n-1$, $\lfloor log_2 n \rfloor$ og 1.

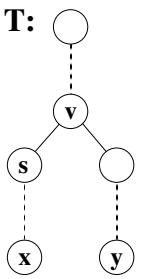
Vi har sett at ingen node kan ha flere barn i eliminasjonstreet enn den er nabo til i G. Vi skal nå se hva som er betingelsen for at vi skal få forgrening i eliminasjonstreet, eller mer formelt hva som må til for at $T(x) \cap T(y) = \emptyset$ for to gitte noder x og y.

Teorem 3.4

Gitt en graf G med eliminasjonstre T. La v være laveste felles forfader til to noder x og y i T. Da er $T(x)$ og $T(y)$ node-disjunkte hvis og bare hvis nodene på stien $v, ..., rot(T)$ i T utgjør et kuttsett som deler G slik at x og y kommer i to forskjellige komponenter.

Bevis:[1]

La s være et barn til v slik at $x \in T(s)$. Vi vet fra Teorem 2.1 at nodene i $T(s)$ utgjør en sammenhengende node-delgraf G' til G.



Figur 3.2 Noden s er et barn til v slik at $x \in T(s)$.

$\Rightarrow$ La K være mengden av alle noder $\in V(G) - V(G')$ som i G er nabo til en node i $V(G')$. Nodene i K utgjør et kuttsett mellom x og y i G. Fra Teorem 2.2 vet vi at alle nodene i K må ligge på stien $v, ..., rot(T)$ i T. Det følger at nodene på stien $v, ..., rot(T)$ utgjør et kuttsett mellom x og y i G.

$\Leftarrow$ Vi vet fra Teorem 2.1 at nodene i $T(s)$ utgjør en sammenhengende node-delgraf til G. I og med at nodene på stien $v, ..., rot(T)$ i T utgjør et kuttsett mellom x og y i G kan y derfor ikke ligge i $T(s)$. Det følger at $T(x) \cap T(y) = \emptyset$. $\square$

Ved hjelp av Teorem 3.4 kan vi avgjøre hvor stor forgrening det vil bli i eliminasjonstreet.

Korollar 3.4.1

Gitt en graf G med eliminasjonstre T, der en nodemengde K elimineres sist. Anta at K deler G i l sammenhengende komponenter, $l > 0$. Da gjelder for alle $v \in K$ at $p(v) \in K$, og det vil være eksakt l noder $\in G - K$ som har en

[1] Liu gir i [12] et bevis for "$\Rightarrow$ delen" av Teorem 3.4. For å bevare helheten og fordi Liu formulerer sitt resultat og bevis på en annen måte enn det som blir gjort her, er hele beviset likevel tatt med.

foreldre-node i T fra K.

Bevis:

Vi viser først at $p(v) \in K$ gjelder for alle $v \in K$. For alle $v \in K$ og alle $w \in V(G) - K$ gjelder $w < v$. Siden en node må være nummerert høyere enn alle sine barn, følger det at hver node i K må ha en foreldre-node fra K.

Vi viser så at eksakt l noder fra $G - K$ vil ha foreldre-noder fra K. Dersom $l > 1$, så utgjør nodene i K et kuttsett mellom hvert par av de l komponentene. Vi ser fra Teorem 3.4 at da gjelder for hvert par av noder x, y, der x og y er med i ulike komponenter, at $T(x) \cap T(y) = \emptyset$.

Siden hver av komponentene i $G - K$ er sammenhengende, ser vi fra Lemma 3.1 at høyst en node fra hver komponent kan ha en foreldre-node fra K. Vi vet også at siden K elimineres sist, kan ingen node fra $G - K$ være rot i T. Den høyeste nummererte noden z fra hver komponent vil derfor ha en foreldre-node $\neq z$ som ikke er fra $G - K$. Det følger at foreldre-noden til z må være fra K. □

Korollar 3.4.1 sier ikke bare hvor stor forgrening det blir fra det kuttsettet som elimineres sist. Vi vet fra Teorem 2.1 at nodene i $T(x)$ for vilkårlig $x \in V(G)$ utgjør en sammenhengende node-delgraf G' til G. Dersom et kuttsett K' til G' elimineres sist av nodene i G', så sier Korollar 3.4.1 hvor mange deltrær som vil henge fra K' i T.

Det følger også fra Korollar 3.4.1 at dersom det er mange kanter i en graf slik at det er vanskelig å finne et lite kuttsett som deler G i mange komponenter, så vil det også være vanskelig å finne en eliminasjonsordning som gir et lavt eliminasjonstre.

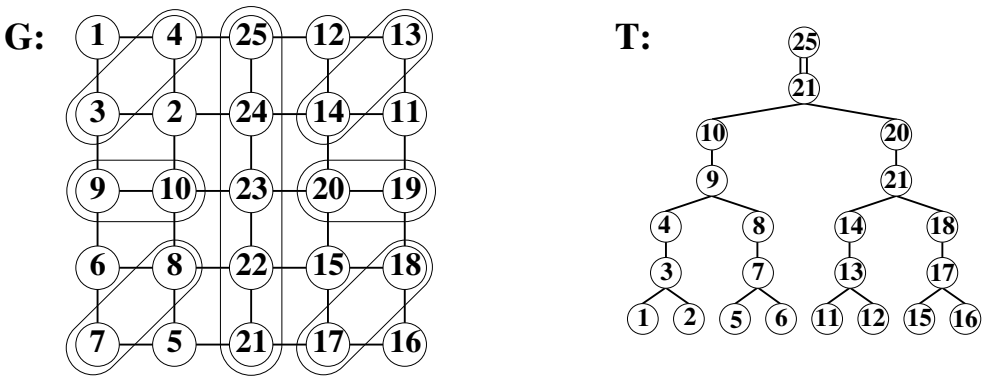
Korollar 3.4.1 gir oss en heuristikk for å finne en eliminasjonsordning som gir forgrening i eliminasjonstreet, og derfor forhåpentligvis også et lavt eliminasjonstre:

For en graf G finner vi et kuttsett som, hvis det fjernes, deler G i to eller flere komponenter. Vi nummererer nodene i kuttsettet sist i eliminasjonsordningen til G. Vi gjentar så prosessen (rekursivt) for hver komponent inntil komponentene vi står igjen med utgjør klikker i G.

En eliminasjonsordning gitt ved en slik fremgangsmåte kalles for en *nøstet oppdelingsordning* (eng. *nested dissection*). George og Liu gir i [2] en mer utførlig presentasjon av nøstet oppdeling.

Fra Teorem 1.1 ser vi at det heller ikke kan oppstå fyllkanter mellom de ulike komponentene. Dersom vi for hver komponent finner et lite kuttsett som deler komponenten i mange delkomponenter, vil vi få et lavt eliminasjonstre, samtidig som vi får relativt få fyllkanter i G^*.

Figur 3.3 viser et eksempel på nøstet oppdeling utført på en 5×5 rutenett-graf og det eliminasjonstreet det gir. Det er en strek rundt nodene i hvert kuttsett i G. En dobbel kant i eliminasjonstreet står for en sti av direkte etterpåfølgende nummererte noder.

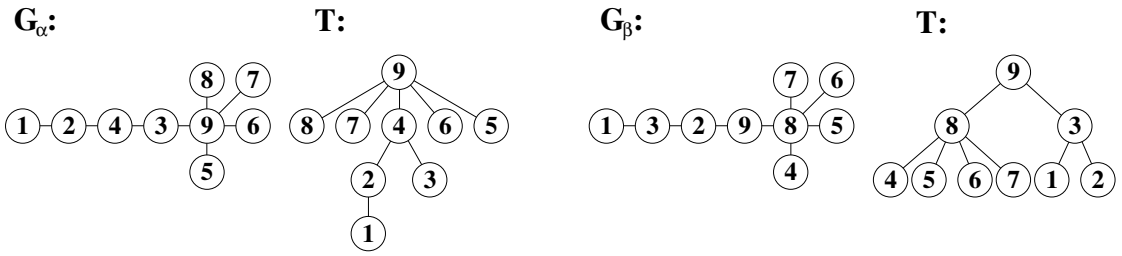


Figur 3.3 Nøstet oppdeling utført på en 5×5 rutenett-graf.

George og Liu forlanger i [2] at i en nøstet oppdelingsordning skal fjerning av et kuttsett gi komponenter av "omtrent" samme størrelse. Gilbert [4] setter som krav til hvert kuttsett at det må være slik at ingen ny komponent inneholder mer enn $n/2$ noder, der n er antall noder i den opprinnelige komponenten. Begge disse kravene settes for å garantere at det ikke oppstår fyllkanter mellom flest mulig noder.

Vi skal nå vise at det ikke eksisterer en nøstet oppdelingsordning som oppfyller kravet om at ingen komponent får inneholde mer $\lfloor c * n \rfloor$ noder, der n er antall noder i den opprinnelige komponenten og c er en gitt konstant der $0 < c < 1$, som gir et minimalt eliminasjonstre for en vilkårlig graf.

I figur 3.4 ser vi to nøstede oppdelingsordninger α og β brukt på en og samme graf samt de eliminasjonstrærne disse gir. Bare α tilfredstiller kravet om at ingen komponent skal ha mer enn $n/2$ noder. Vi ser at β gir det laveste eliminasjonstreet.



Figur 3.4 To nøstede oppdelingsordninger utført på samme graf.

I figur 3.4 er det 4 noder som bare er nabo til node 9 i G_α. Anta at vi forandrer G_α slik at det er k noder, der k er vilkårlig, som bare er nabo til node 9. La δ være en eliminasjonsordning som gir et minimalt eliminasjonstre til G_α. Node 3 eller node 4 må elimineres sist i δ, og vi ser at vi kan bestemme k slik at komponenten som inneholder node 9 vil inneholde mer enn $\lfloor c * n \rfloor$ noder for en gitt verdi av c der $0 < c < 1$. Det følger at det ikke gjelder generelt at en nøstet oppdelingsordning med en begrensning på størrelsen av komponentene alltid vil gi et minimalt eliminasjonstre.

Vi ser også fra grafen i figur 3.4 at en fremgangsmåte hvor vi eliminerer det kuttsettet sist som deler G i flest komponenter, heller ikke vil gi et minimalt eliminasjonstre.

Leiserson og Lewis viser i [8] en variant av nøstet oppdeling utviklet for å finne lave eliminasjonstrær. Vi skal her vise at det alltid finnes en nøstet oppdelingsordning som for en vilkårlig graf gir et eliminasjonstre av minimal høyde. Denne setter ingen krav på størrelsen til komponentene.

Vi trenger først følgende definisjon:

Definisjon 3.3 Et kuttsett til en graf G er *minimalt*[1] dersom ingen undermengde av det også er et kuttsett til G.

Vi ser at ingen overmengde til et minimalt kuttsett kan være et minimalt kuttsett.

Dersom vi eliminerer nodene i en graf G etter en nøstet oppdelingsordning med minimale kuttsett kan vi v.h.a. Korollar 3.4.1 avgjøre hvor mange barn de ulike nodene vil ha i det tilhørende eliminasjonstreet T. Anta at vi eliminerer et minimalt kuttsett sist av nodene i en komponent, og at kuttsettet deler komponenten i $l \geq 2$ delkomponenter. Da følger det av Korollar 3.4.1 og definisjonen av minimale kuttsett at den nederste noden i kuttsettet vil ha eksakt l barn i T, og at alle andre noder i kuttsettet vil ha et barn hver.

[1] Rose definerer i [15] et minimalt kuttsett til en graf som det kuttsettet som inneholder færrest noder.

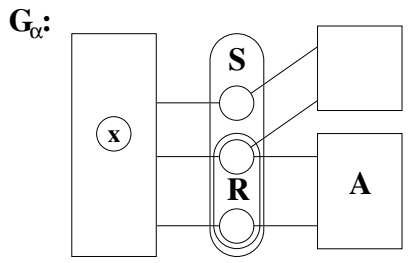
Vi kan nå vise at det for en vilkårlig graf G alltid finnes en nøstet oppdelingsordning med minimale kuttsett som gir et minimalt eliminasjonstre. Måten vi gjør dette på er å vise at for enhver eliminasjonsordning α til G, med tilhørende eliminasjonstre T, kan vi alltid permutere α slik at vi får et nytt eliminasjonstre T' der $h(T') \leq h(T)$, og det eksisterer en nøstet oppdelingsordning med minimale kuttsett til G som ville gitt T'.

Lemma 3.3

Gitt en graf G_α med tilhørende eliminasjonstre T der minst en node har 2 eller flere barn. Vi kan da permutere α slik at vi får et nytt eliminasjonstre T' der følgende er oppfylt: (1) Nodene på stien fra og med den øverste noden i T' som har minst 2 barn til og med $rot(T')$ utgjør et minimalt kuttsett til G_α. (2) $h(T') \leq h(T)$.

Bevis:

La v være den øverste noden i T som har minst 2 barn x og y. Vi ser fra Teorem 2.3 at x vil være nabo i G^*_α til eksakt de av nodene på stien $v, ..., rot(T)$ som nodene i $T(x)$ er nabo til i G_α. Kall denne mengden S. Nodene i S utgjør et kuttsett til G_α mellom nodene i $T(x)$ og resten av nodene i $G_\alpha - S$. La A være en komponent i $G_\alpha - S$ som er nabo til færrest mulig noder i S. Kall mengden av noder i S som A er nabo til for R.



Figur 3.5 Nodene i R utgjør et minimalt kuttsett til G_α. (Merk at x også kan ligge i A.)

Nodene i R utgjør et kuttsett til G_α slik at alle komponentene til $G - R$ er nabo til samtlige noder i R. Det følger at ingen undermengde til R kan være et kuttsett til G_α. Derfor utgjør R et minimalt kuttsett til G_α.

Vi skal nå vise at en innbyrdes permutasjon i α av nodene på stien $v, ..., rot(T)$ i T ikke fører til at høyden til eliminasjonstreet til G øker. La U være mengden av noder som opprinnelig ligger på stien $v, ..., rot(T)$ i T. Vi konstaterer først at en innbyrdes permutasjon av nodene i U ikke vil føre til at det oppstår eller forsvinner fyllkanter mellom noen av etterkommerene til v i T. Det gjør at

strukturen til deltrærne som henger fra v i T, vil forbli uforandret. Siden nodene i U, etter en omordning, fremdeles blir eliminert sist av nodene i G_α, vil en node $w \in U$ bare ha forfedre fra U. Det følger at w ikke kan ha kommet lenger ned i eliminasjonstreet enn v var opprinnelig, og at hvert deltre som hengte av v før omordningen, fremdeles bare vil ha forfedre fra U. Vi ser av dette at høyden til eliminasjonstreet til G_α ikke kan øke etter en innbyrdes permutasjon av nodene i U.

Dersom vi eliminerer nodene i R sist, og resten av nodene på stien $v, ..., rot(T)$ i T rett før R i samme relative ordning som tidligere, ser vi fra Teorem 3.4 at den nederste noden i R vil ha minst to barn i det nye eliminasjonstreet T'. Ingen forfader til denne noden kan ha mer enn ett barn da R er et minimalt kuttsett til G'. Vi har nå fått et eliminasjonstre T' der nodene på stien fra og med den øverste noden som har minst to barn til og med $rot(T')$, utgjør et minimalt kuttsett til G og der $h(T') \leq h(T)$. $\square$

Dersom vi utfører permutasjonen slik den er beskrevet i beviset av Lemma 3.3, ser vi at nodene fra og med den første noden v med minst to barn til og med $rot(T)$, nå utgjør et minimalt kuttsett til G. Vi kan for hvert deltre som henger fra v i T, gjenta prosessen rekursivt inntil deltrærene som står igjen, utgjør kjeder i eliminasjonstreet. Vi har nå fått et eliminasjonstre T' der hver ikke-nederste maksimale kjede K utgjør et minimalt kuttsett til node-delgrafen til G bestående av nodene i K og alle etterkommere til K i T'. Vi skal nå se at vi kan permutere eliminasjonsordningen som gav T', uten at høyden til T' øker, slik at hver nederste kjede i T' utgjør en klikk i G.

Lemma 3.4

Gitt en graf G med n noder og eliminasjonstre T der $h(T) = n - 1$. Dersom nodene i G ikke utgjør en klikk, kan vi finne en eliminasjonsordning α til G slik at vi får et nytt eliminasjonstre T' der følgende er oppfylt: (1) Nodene på stien fra den øverste noden som minst 2 barn i T' til $rot(T')$ utgjør et minimalt kuttsett til G_α. (2) $h(T') < n - 1$.

Bevis:
La v være en node som ikke er en nabo til samtlige andre noder i G. La S være mengden av naboer til v i G. Vi ser at S utgjør et kuttsett til G. La A være en komponent i $G - S$ som i G er nabo til færrest mulig noder i S. Kall den mengden av noder i S som A er nabo til for R. Eliminer nodene i R sist, og resten av nodene i G i samme relative ordning som tidligere. Nodene i R utgjør et minimalt kuttsett til G, og den nederste noden i R vil derfor ha minst to barn i T. Siden en node har minst to barn i T, er $h(T) < n - 1$. □

Fremgangsmåten i beviset av Lemma 3.4 kan brukes rekursivt på hver nederste kjede i T inntil hver nederste kjede utgjør en klikk i G.
Vi ser nå fra bevisene av Lemma 3.3 og 3.4 at for ethvert eliminasjonstre T, tilhørende en graf G, kan vi finne et nytt eliminasjonstre T' til G slik at følgende er oppfylt: (1) Det eksisterer en nøstet oppdelingsordning til G, der alle kuttsettene er minimale, som ville gitt T'. (2) $h(T') \leq h(T)$.
Siden vi kan gjøre dette for alle eliminasjonstrær til G får vi følgende resultat:

Teorem 3.5
For en vilkårlig graf G eksisterer det en nøstet oppdelingsordning, der alle kuttsett er minimale, som gir et eliminasjonstre av minimal høyde. □

Pothen har i [14] vist at det er et NP-komplett problem å avgjøre om det eksisterer en eliminasjonsordning til en vilkårlig graf som gir et eliminasjonstre av høyde mindre eller lik k for et gitt heltall k. Vi skal nå vise at problemet fremdeles er NP-komplett om vi innfører et ekstra krav om at ordningen som gir eliminasjonstreet skal være gitt ved en nøstet oppdelingsordning med minimale kuttsett. Gitt en vilkårlig graf G og et heltall k. De to problemene kan da defineres som:

Γ. Eksisterer det en eliminasjonsordning som gir et eliminasjonstre til G av høyde mindre eller lik k?

Π. Eksisterer det en nøstet oppdelingsordning med minimale kuttsett til G som gir et eliminasjonstre av høyde mindre eller lik k?

Vi ser direkte fra Teorem 3.5 at følgende gjelder:

Korollar 3.5.1

For en gitt graf G og et heltall k gjelder: Γ har en løsning hvis og bare hvis Π har en løsning. $\square$

Vi kan gjette en løsning α til Π, og teste i polynomisk tid om α er gitt ved en nøstet oppdeling med minimale kuttsett, lage eliminasjonstreet T til G_α og avgjøre om $h(T) \leq k$. Det følger at Π er med i NP. Sammen med Korollar 3.5.1 gir dette oss:

Korollar 3.5.2

Π er NP-komplett. $\square$

Vi vet at høyden til et eliminasjonstre til en graf med n noder kan ligge mellom 0 og $n-1$. Dersom vi kan løse Π i tid $O(f(n))$ så kan vi finne den minste k slik at Π har en ja-løsning i tid $O(f(n)logn)$. Dette gir oss følgende korollar:

Korollar 3.5.3

Det er NP-hardt å finne den minste k slik at Π har en løsning. $\square$

Vi skal i kapittel 4 bruke Teorem 3.5 for å utvikle en algoritme for å redusere høyden til et eliminasjonstre. I den forbindelse trenger vi følgende resultat:

Teorem 3.6

Gitt en graf G med eliminasjonstre T. La K være en ikke-nederste maksimal kjede i T og la G' være node-delgrafen til G bestående av nodene i K og alle etterkommere til K i T. Da er K et minimalt kuttsett til G' hvis og bare hvis hvert barn til den nederste noden i K er nabo i G^* til samtlige noder i K.

Bevis:

$\Rightarrow$ Anta at K er et minimalt kuttsett til G'. Da vil hver komponent i $G' - K$ være nabo i G' til samtlige noder i K. Hver komponent utgjør et deltre av T som henger fra den nederste noden i K. Fra Teorem 2.3 vet vi at en node x er nabo i G^* til de nodene som nodene i $T(x)$ er nabo til i G. Det følger derfor at hvert barn i T til den nederste noden i K vil være nabo i G^* til samtlige noder i K.

$\Leftarrow$ Vi vet fra Teorem 3.4 at K er et kuttsett til G' som deler G' i et antall komponenter. Hver komponent utgjør et deltre av T som henger fra den nederste noden v i K. For hvert barn x til v i T gjelder det at den er nabo i G^* til de av nodene i K som nodene i $T(x)$ er nabo til i G. Siden hvert barn til v er nabo til samtlige noder i K, følger det at hver sammenhengende komponent er nabo til samtlige noder i K. Vi ser derfor at ingen undermengde av K kan være et kuttsett til G', og det følger at K er et minimalt kuttsett. $\square$

Ved hjelp av Teorem 3.6 kan vi lett avgjøre om nodene i en maksimal kjede $x, ..., y$ i T utgjør et minimalt kuttsett til node-delgrafen til G bestående av nodene i $T(y)$.
Fra Teorem 3.6 kan vi videre vise følgende resultat som beskriver en egenskap ved eliminasjonstrær som er gitt ved nøstede oppdelingsordninger med minimale kuttsett:

Korollar 3.6.1
Gitt en graf G med eliminasjonstre T. Dersom T er gitt ved en nøstet oppdelingsordning med minimale kuttsett så utgjør nodene i enhver kjede i T en klikk i G^*.

Bevis:
Siden enhver kjede er en delgraf av en maksimal kjede er det nok å vise resultatet for maksimale kjeder.
La K være en maksimal kjede i T. Hvis K er en nederste kjede i T, er K en klikk i G og derfor også i G^*. Anta derfor at K ikke er en nederste kjede. Fra Teorem 3.6 vet vi at hvert barn til den nederste noden i K er nabo i G^* til samtlige noder i K. Det følger derfor fra Teorem 2.3 at K utgjør en klikk i G^*. $\square$

Dette resultatet vil vi nyttegjøre oss i kapittel 4.

4 En klasse av algoritmer for å redusere høyden til et eliminasjonstre

Liu [10] [11] og Hafsteinsson [5] har utviklet hver sin algoritme for å redusere høyden til et gitt eliminasjonstre. Vi skal her utvikle en ny algoritme med samme foremål. Vi skal sammenligne denne med algoritmene til Liu og Hafsteinsson, og vise at alle tre er varianter av algoritmer som baserer seg på et felles resultat. På bakgrunn av dette resultatet skal vi så utvikle en fjerde algoritme for å redusere høyden til et eliminasjonstre.

4.1 Minimale-kuttsett algoritmen

I denne seksjonen skal vi utvikle en ny algoritme, for å redusere høyden til et eliminasjonstre. Denne algoritmen baserer seg på bevisene av Lemma 3.3 og 3.4.

Fremgangsmåten som bevisene for Lemma 3.3 og 3.4 gir for å finne en nøstet oppdelingsordning med minimale kuttsett, kan med en liten modifikasjon brukes til en algoritme for å redusere høyden til et eliminasjonstre.

Vi ser først hvordan vi kan bruke resultatet fra Lemma 3.3 for å omordne en ikke-nederste maksimal kjede.

Når vi har en ikke-nederste maksimal kjede $x, ..., y$ i T, og skal bestemme S slik som beskrevet i beviset av Lemma 3.3, tar vi utgangspunkt i et barn v til x, og setter S lik alle naboer til v i G^* som ligger på stien $x, ..., y$ i T. Istedenfor å først bestemme det minimale kuttsettet $R \subseteq S$, foretar vi en lokal permutasjon i eliminasjonsordningen av nodene på stien $x, ..., y$ i T. Denne permutasjonen består i at vi eliminerer nodene i S sist av nodene på stien $x, ..., y$ i T. Dette gjør vi slik at nodene i S blir eliminert i samme relative ordning som tidligere. Vi eliminerer nodene på stien $x, ..., y$ i T som ikke er med i S, rett før nodene i S, også disse i samme relative ordning som tidligere. Det vil føre til at nodene i S henger under hverandre i det nye eliminasjonstreet. Nodene som opprinnelig lå på stien $x, ..., y$ i T som ikke er med i S, vil fordele seg øverst i et eller flere deltrær som henger fra noder i S. Deltrærne som opprinnelig hengte fra x vil nå henge fra noder som opprinnelig lå på stien $x, ..., y$ i T. Som en følge av denne omordningen vil x i det nye eliminasjonstreet ha minst to barn.

Vi kan nå lett bestemme R. La s være den øverste noden i S som har minst to barn, og la t være en node blant barna til s som i G^* er nabo til færrest noder i S. Det minimale kuttsettet R blir da alle noder i S som er nabo til t i G^*. Vi

permuterer nå rekkefølgen i eliminasjonsordningen av nodene i S slik at R blir eliminert sist. Dette vil ha tilsvarende konsekvenser som når vi eliminerte nodene i S sist av nodene som opprinnelig lå på stien $x, ..., y$ i T.

Anta nå at når vi skal bestemme S så velger vi v som en node av maksimal høyde blant barna til x. Det medfører at etter omordningen har nodene i $T(v)$ kommet $|x, ..., y| - |S|$ plasser nærmere $rot(T)$.

Dersom vi likevel har flere kandidater å velge mellom når vi skal bestemme S, velger vi en node som vil bli flyttet lengst opp etter en omordning. Det vil si at vi velger en node som er nabo i G^* til færrest noder på stien $x, ..., y$ i T. Grunnen for å gjøre et slikt valg, er at hvis et deltre kommer høyt nok opp, så kan vi håpe at det ikke vil virke inn på den totale høyden til eliminasjonstreet når algoritmen er ferdig.

Dersom vi har flere muligheter for t når vi skal bestemme R, velger vi også t som en høyeste node blant de aktuelle kandidatene.

Vi modifiserer på samme måte den omordningen av en nederste kjede, som er gitt i beviset til Lemma 3.4. Når vi skal bestemme S for en nederste kjede, tar vi utgangspunkt i den øverste noden i kjeden, som ikke er nabo i G til samtlige av sine etterkommere. For å gjøre det lettere å bestemme det minimale kuttsettet R, eliminerer vi S sist på samme måte som vi gjorde med S for en ikke-nederste kjede. Deretter bestemmer vi også R på samme måte som for en ikke-nederste kjede, og eliminerer R sist av nodene i S.

Dette gir oss en algoritme for å minske høyden til et eliminasjonstre. Vi begynner med den øverste kjeden i T, og gjennomfører omordningen som beskrevet over. Deretter fjerner vi det øverste kuttsettet og gjentar prosessen rekursivt på hvert av deltrærne vi står igjen med. Algoritmen stopper opp når deltrærne utgjør klikker i G. Vi kaller denne algoritmen for *Minimale-kuttsett algoritmen*. Her følger en pseudo-koden til algoritmen. Den tar som inn-parameter et eliminasjonstre T.
I algoritmen er S og R nodemengder, og s, t, v og w er noder.

Minimale-kuttsett algoritmen(T)

Dersom nodene i T ikke er en klikk i G

$\rightarrow$ Dersom T er en kjede

- $\rightarrow$ v = Den høyeste noden i T som ikke er nabo i G til alle sine etterkommere.
  S = Alle naboer til v i G.
- $\square$ v = En høyeste node blant barna til den øverste noden i T, som har minst to barn. Dersom det er flere muligheter, velges en node som i G^* er nabo til færrest av sine forfedre.

S = Alle naboer til v i G^*, som er forfedre til v i T.

Foreta en lokal omordning slik at nodene i S blir eliminert sist.

s = Den øverste noden i S som har minst to barn.

t = En barn til s, som i G^* er nabo til færrest noder i S. Dersom det er flere muligheter, velges en node som er høyest i T.

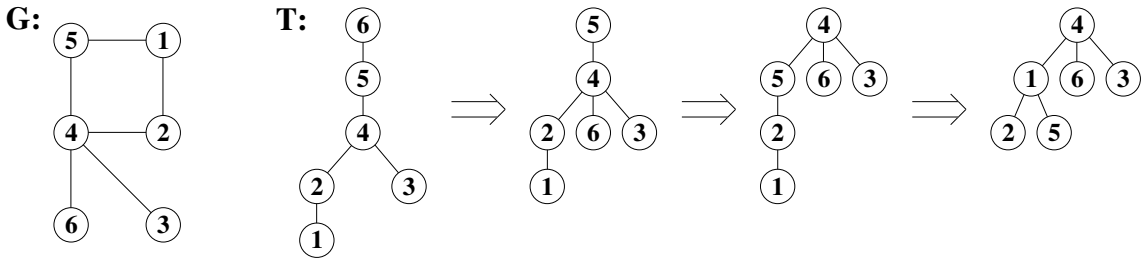
R = Alle naboer til t i G^*, som er forfedre til t i T.

Foreta en lokal omordning slik at nodene i R blir eliminert sist.

For hvert barn w til den nederste noden i R

Minimale-kuttsett algoritmen($T(w)$)

Figur 4.1 viser stegvis hvordan et eliminasjonstre blir omordnet ved bruk av Minimale-kuttsett algoritmen.



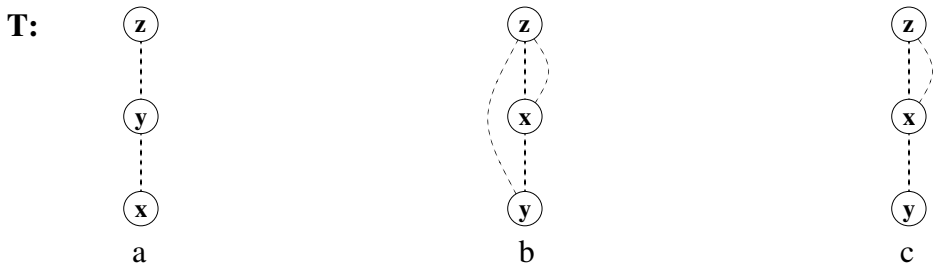
Figur 4.1 Eksempel på bruk av Minimale-kuttsett algoritmen.

Under utførelsen av algoritmen setter vi først $v = 2$ og $S = \{5,4\}$. Når vi har eliminert nodene i S, sist setter vi $s = 4$ og $t = 6$ eller 3. Uansett hva vi setter t til, så får vi $R = \{4\}$. Vi eliminerer 4 sist og utfører algoritmen på alle deltrær som henger fra node 4. Det fører til at deltreet med 5 som rot blir brettet sammen. Mengden av fyllkanter i G^* forandrer seg også. Når vi starter har vi kantene (2,5) og (5,6). Når algoritmen er ferdig, har vi bare kanten (1,4).

Den lengste stien i grafen i figur 4.1 er av lengde 4. Siden $\lfloor log_2(5) \rfloor = 2$ og eliminasjonstreet vi står igjen med har høyde 2, ser vi fra Korollar 3.3.1 at eliminasjonstreet er minimalt etter omordningen.

Det er trolig at vi kan forbedre behandlingen av nederste kjeder i forhold til den fremgangsmåten som algoritmen gir. Dersom vi bruker en nøstet oppdelingsordning direkte på node-delgrafen til G som består av nodene i en nederste kjede, får vi sannsynligvis bedre balanse i eliminasjonstreet og derfor også et lavere tre.

Når vi foretar en omordning av en kjede slik at et kuttsett blir minimalt, kan det både oppstå og forsvinne fyllkanter i G^*. Anta at vi har tre noder x, y og z, valgt på følgende måte: Før omordningen er y forfader til x, og z er forfader til y (figur 4.2.a). Etter omordningen er x en forfader til y, og z en forfader til x. Dersom $(y, z) \in V(G)$ og $(z, x) \notin V(G)$ (figur 4.2.b), vil x etter omordningen ha en fyllkant til z. Dersom $(x, z) \in V(G)$ og hverken y eller noen etterkommere til y etter omordningen er nabo til z (figur 4.2.c), så vil fyllkanten $(y, z) \in V(G^*)$ forsvinne.



Figur 4.2 a. Før omordning. b. Kanten (x, z) oppstår. c. Kanten (y, z) forsvinner.

I en analyse av antall fyllkanter i en graf, kan vi ha som kriterium for at det ikke skal eksistere en fyllkant mellom to noder, at de skal være i node-disjunkte deltrær av T. Etter en omordning kan vi da garantere at det er færre fyllkanter.

Tidskompleksiteten ved en sekvensiell utføring av Minimale-kuttsett algoritmen vil være begrenset nedad av høyden til eliminasjonstreet. Vi ser at resultatet av en omordning kan påvirke resultatet av omordninger lenger nede i eliminasjonstreet. Dersom vi skal parallellisere algoritmen, må vi begynne med det øverste kuttsettet og arbeide oss nedover i eliminasjonstreet. Hver gang vi kommer til en forgrening i eliminasjonstreet, kan vi assosiere hvert gjenværende deltre med en prosessor. Også på en parallell maskin vil kjøretiden derfor være begrenset nedad av eliminasjonstreets høyde.

4.2 Algoritmene til Liu og Hafsteinsson

Vi skal her beskrive algoritmene som Liu [10] [11] og Hafsteinsson [5] har utviklet for å redusere høyden til et eliminasjonstre.
Algoritmen til Liu bygger på følgende omordningsteg:

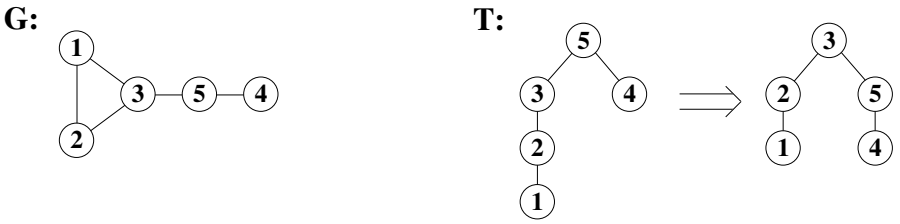
Gitt en graf G med eliminasjonstre T. Ta utgangspunkt i en node x, og foreta en permutasjon av forfedrene til x i T. Permutasjonen består i å eliminere de

av forfedrene til x, som x er nabo til i G^*, sist i samme relative ordning som tidligere. Resten av forfedrene til x elimineres rett før, også de i samme relative ordning som tidligere.

Selve algoritmen består i en gjennomgang av en sti fra $rot(T)$ til et dypeste løv[1] og foreta omordningsteget så lenge det gir et lavere eliminasjonstre. Her følger pseudo-koden til algoritmen. I algoritmen er v og x noder.

Algoritmen til Liu(T)
v = et dypeste løv i T.
For hver node x på stien $rot(T), ..., v$ i T:
  Dersom x ikke er nabo i G^* til samtlige av sine forfedre i T
  $\rightarrow$ Dersom omflytting gjør T lavere
    $\rightarrow$ Foreta omflytting
    $\square$ Avslutt algoritmen

Figur 4.3 viser effekten av algoritmen til Liu på et eliminasjonstre tilhørende en graf G.



Figur 4.3 Eksempel på bruk av algoritmen til Liu.

Før en omordning vil forfedrene til x utgjøre en klikk i G^*. Det medfører at de av forfedrene til x som ikke er nabo til x i G^* vil etter omordningen henge under hverandre i et deltre som henger fra $p(x)$. Noden x vil nå bare ha forfedre som den er nabo til i G^*. Det følger at $T(x)$ nå har kommet nærmere $rot(T)$. Dersom x opprinnelig har k forfedre og er nabo til $l \leq k$ av disse i G^*, har nodene i deltreet $T(x)$ etter ett steg av algoritmen blitt flyttet $k - l$ plasser nærmere roten til T.

Vi kan se på et omflyttingsteg i algoritmen som bestående av flere enkeltoperasjoner. En enkeltoperasjon består i å ta den laveste noden w blant forfedrene til x som x ikke er nabo til i G^*, og eliminere w rett etter x. Noden w vil da få samme foreldre-node som x.

[1] Stien går egentlig fra det dypeste løvet v til $rot(T)$ slik at det vi gjør er en baklengs gjennomgang av stien $v, ..., rot(T)$.

I hvert steg av algoritmen foretar vi en lokal permutasjon av noder som utgjør en klikk i G^*. Det medfører at det ikke vil oppstå nye fyllkanter etter bruk av algoritmen.

Algoritmen til Hafsteinsson bygger på følgende resultat som han viser i [5]:

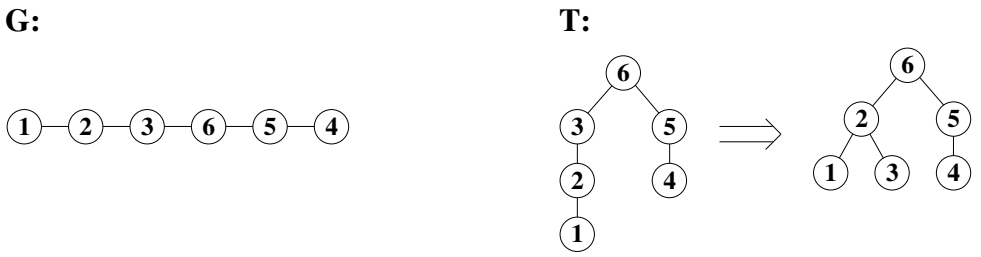
Teorem 4.1
La K være en kjede i T slik at K ikke er en klikk i G^*. La v være den øverste noden i K som ikke er nabo i G^* til samtlige andre node i K, og la w være den øverste noden i K som ikke er nabo til v i G^*. Vi kan da permutere nodene i K slik at v får samme foreldre-node som w.

Bevis:
La x være barnet til v i T. Da utgjør nodene på stien $p(w), ..., x$ i T et kuttsett S mellom w og v i G. Fra Teorem 3.4 vet vi at dersom nodene i S elimineres etter v og w så gjelder $T(v) \cap T(w) = \emptyset$.
Vi vet at v er nabo i G^* til alle noder i S. Det følger derfor at dersom vi eliminerer alle nodene i samme ordning som tidligere bortsett fra at v nå elimineres rett etter w, så vil v og w ha samme foreldre-node i T. $\square$

Etter at vi har utført en omordning som beskrevet i beviset av Teorem 4.1 så vil noden v ha blitt et løv i T, samtidig som nodene i $T(w)$ har blitt flyttet en plass nærmere roten til T.
Fra Teorem 4.1 får vi følgende Korollar som beskriver en egenskap ved minimale eliminasjonstrær:

Korollar 4.1.1
Et minimalt eliminasjonstre må inneholde en lengste sti der hver kjede er en klikk i G^*. $\square$

Algoritmen til Hafsteinsson består i å ta for seg hver kjede som ligger på en lengste sti i T og som ikke utgjør en klikk i G^*, og utføre omordningen som beskrevet over. Dette gjentar vi inntil alle kjeder i T, er klikker i G^*. Dersom vi bare velger kjeder i T av maksimal lengde, vil rekkefølgen vi velger dem i, ikke innvirke på det endelige resultatet.
Figur 4.4 viser algoritmen til Hafsteinsson utført på eliminasjonstreet til en graf G.



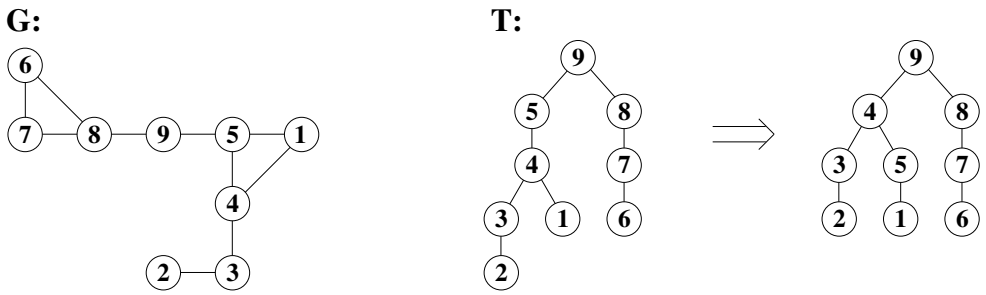
Figur 4.4 Eksempel på bruk av algoritmen til Hafsteinsson.

På en en-prosessor datamaskin vil kjøretiden til begge algoritmene være avhengig av høyden til eliminasjonstreet. Dersom vi ønsker å parallellisere algoritmene, ser vi at algoritmen til Hafsteinsson egner seg best, da vi kan sette en prosessor til å arbeide med hver maksimal kjede parallelt, slik at kjøretiden blir så lang som det tar å ordne den mest arbeidskrevende kjeden.
Algoritmen til Liu er vanskeligere å parallellisere, da hvert steg i algoritmen påvirker hva vi gjør i det neste steget.

4.3 Sammenligning av algoritmene

Vi skal nå vise at ingen av de to algoritmene vi så på i seksjon 4.2 og Minimale-kuttsett algoritmen fra seksjon 4.1, er strengt bedre enn hverandre. Dette gjør vi ved å vise at det for hvert ordnet par av algoritmer $<A, B>$ eksisterer en graf med tilhørende eliminasjonstre slik at A gir er lavere eliminasjonstre enn B.
I figur 4.3 ser vi at hverken algoritmen til Hafsteinsson eller Minimale-kuttsett algoritmen gir et lavere eliminasjonstre enn det vi opprinnelig hadde. Grunnen til at algoritmen til Liu i dette eksempelet gir bedre resultat enn de to andre algoritmene, er at den kan flytte ut noder som har mer enn et barn i T, fra en lengste sti i T. Ingen av de to andre algoritmene klarer dette.

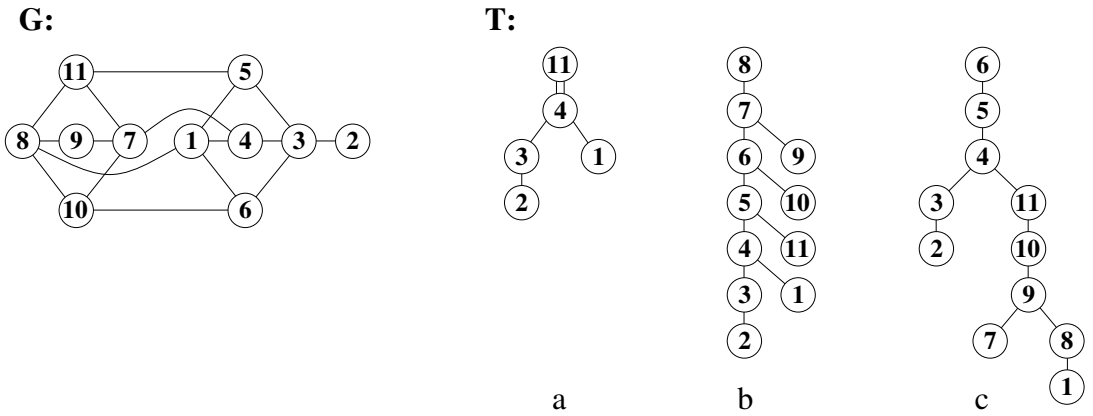
Figur 4.5 viser en graf hvor bare Minimale-kuttsett algoritmen gir et lavere eliminasjonstre.



Figur 4.5 Eksempel på bruk av Minimale-kuttsett algoritmen.

En av årsakene til at Minimale-kuttsett algoritmen kan gi bedre resultat enn algoritmen til Hafsteinsson, er at Minimale-kuttsett algoritmen kan løse opp stier i T hvor nodene danner klikker i G^* men ikke i G. I algoritmen til Hafsteinsson blir den noden som blir flyttet ut, alltid et løv i det endelige eliminasjonstreet. En slik begrensning gjelder ikke for de to andre algoritmene.

Fra Korollar 3.6.1 vet vi at alle kjeder vil være klikker i G^* etter at vi har utført Minimale-kuttsett algoritmen. Det følger derfor at etter bruk av Minimale-kuttsett algoritmen vil algoritmen til Hafsteinsson ikke kunne forbedre resultatet videre. Dersom algoritmen til Hafsteinsson klarer å redusere høyden til en kjede, kan kjeden ikke være et minimalt kuttsett. Dette gjør at Minimale-kuttsett algoritmen alltid vil gi positive resultat dersom algoritmen til Hafsteinsson gjør det. Det eksisterer likevel grafer hvor algoritmen til Hafsteinsson gir bedre resultat enn Minimale-kuttsett algoritmen. Et eksempel på dette er vist i figur 4.6.



Figur 4.6 Eksempel på bruk av algoritmen til Hafsteinsson og Minimale-kuttsett algoritmen på samme graf. a. Opprinnelig eliminasjonstre. b. Algoritmen til Hafsteinsson. c. Minimale-kuttsett algoritmen.

Det er verdt å merke seg at dersom algoritmen til Hafsteinsson ikke plukker ut kjeder av maksimal lengde, vil den likevel ikke være strengt bedre enn noen av de andre algoritmene. Vi ser at uansett hvordan vi velger kjedene i de opprinnelige eliminasjonstrærne i figur 4.3 og figur 4.5, så vil algoritmen til Hafsteinsson ikke gi lavere eliminasjonstrær enn det vi hadde opprinnelig.

Til slutt ser vi at algoritmen til Liu brukt på det opprinnelige eliminasjonstreet til grafen i figur 4.4 ikke ville gi et lavere eliminasjonstre. En av årsakene til at algoritmen til Hafsteinsson og Minimale-kuttsett algoritmen kan gi bedre resultat en algoritmen til Liu, er at algoritmen til Liu ikke arbeider videre nedover i eliminasjonstreet, når den ikke lenger kan gjøre treet lavere uten å risikere å introdusere nye fyllkanter.

4.4 Et resultat om permuteringer av eliminasjonstrær

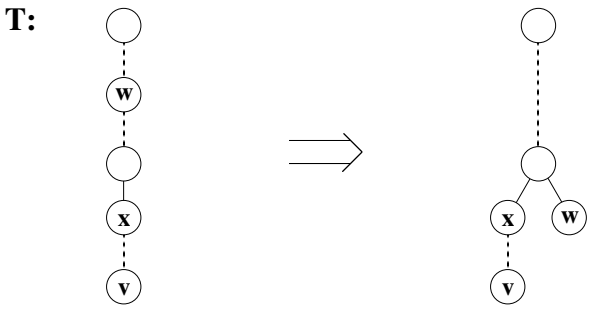
Vi har nå sett at ingen av de tre algoritmene fra seksjon 4.1 og 4.2 er strengt bedre enn noen av de andre. Vi skal nå vise at alle tre algoritmene har et felles trekk som gjør at vi kan si at de tilhører samme klasse av algoritmer. Vi skal vise at samtlige algoritmer baserer seg på følgende resultat:

Teorem 4.2

Gitt en graf G med eliminasjonstre T. La v være et dypeste løv i T og w en forfader til v. Dersom $(v, w) \notin E(G)$ kan vi permutere nodene på stien $v, ..., w$ i T, slik at eksakt w blir fjernet fra stien $v, ..., rot(T)$ i T.

Bevis:

Siden v ikke er nabo til w i G, og v er løv i T, kan v heller ikke være nabo til w i G^*. Det finnes derfor en øverste node x på stien $v, ..., w$ i T, som ikke er nabo til w i G^*, og derfor heller ikke i G. La z være det barnet til w som også er en forfader til x i T. Da utgjør nodene på stien $p(x), ..., z$ i T et kuttsett mellom x og w i G. Det følger derfor at dersom nodene elimineres i samme rekkefølge som tidligere, bortsett fra at w elimineres rett etter x, så vil x og w nå ha samme foreldre-node i T.



Figur 4.7 Noden w blir flyttet ned slik at den får samme foreldre-node som x.

Vi kan se på en omordning som beskrevet ovenfor, som bestående av flere enkeltsteg. Hvert steg består i at w bytter plass i eliminasjonsordningen med et av sine barn y i T. Vi vet at w er nabo til $p(w)$ i G^* før en slik lokal omordning. Det følger derfor at y vil være nabo i G^* til $p(w)$ etter at y og w har byttet plass. Noden y vil få $p(w)$ som foreldre-node, mens w får y som foreldre-node. Vi ser av dette at ingen andre noder enn w vil bli fjernet fra stien $v, ..., rot(T)$ i T etter at hele omordningen er ferdig. □

Vi kaller permuteringen av en lengste sti i T, slik at en node w blir fjernet fra denne stien, som beskrevet i beviset av Teorem 4.2, for at vi *flytter ned* w.

Merk at bruk av resultatet fra Teorem 4.2 ikke nødvendigvis vil føre til at vi får et lavere eliminasjonstre. Dersom en node y har en foreldre-node fra stien $p(x), ..., w$ i T, og y er nabo til w i G^*, så vil y etter omflyttingen ha w som foreldre-node. Dersom $h(y) \geq h(x)$ så vil ikke høyden til T avta.

Det er ikke nødvendig i Teorem 4.2 at v er et *dypeste* løv. Resultatet gjelder så lenge v er et løv i T. Grunnen til at resultatet er formulert slik, er at den totale høyden til T ikke kan avta dersom v er nabo til et dypeste løv.

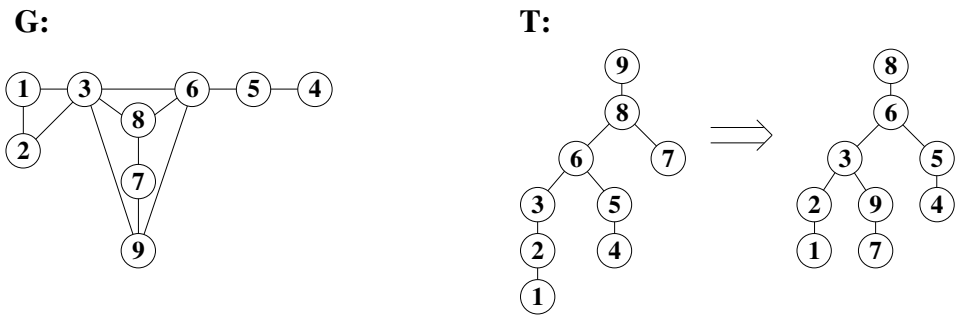
Vi skal nå vise hvordan de ulike algoritmene utnytter resultatet fra Teorem 4.2. Vi har allerede sett at algoritmen til Liu består i, med utgangspunkt i en node x, å flytte ned en node w som ikke er nabo til x i G^*, slik at w får samme foreldre-node som x. Noden x velges i slik at den ligger på en lengste sti $v, ..., rot(T)$ i T. Siden x ikke er nabo til w i G^* kan heller ikke v være det i G. Vi ser av dette at algoritmen utfører det samme omflyttingsteget som i beviset til Teorem 4.2.

I et omflyttingsteg i algoritmen til Hafsteinsson omordner vi er kjede på en lengste sti i T slik at en node v blir et løv. Dette omordningsteget er helt ekvivalent med at vi flytter ned v. Det følger derfor at algoritmen til Hafsteinsson baserer seg på resultatet fra Teorem 4.2

I Minimale-kuttsett algoritmen kan vi også se på omordningen av et allerede eksisterende kuttsett K slik at det blir minimalt, som at vi utnytter resultatet fra Teorem 4.2. Vi bestemmer S ut fra en node w som ligger på en lengste sti i T. Dersom vi tar den til enhver tid laveste noden blant forfedrene til w, som er med i $K - S$, og flytter den ned slik at den får samme foreldre-node som w, så vil S bli eliminert sist. Vi kan gjennomføre en tilsvarende operasjon når vi skal eliminere R sist av nodene i S. Den eneste forskjellen er at noden vi tar utgangspunkt i, ikke nødvendigvis ligger på en lengste sti i T.

Omordning av en nederste kjede i T som ikke er en klikk i G, kan derimot ikke alltid sees på som at vi bruker Teorem 4.2. Dersom vi har en nederste kjede i T, som er en klikk i G^* men ikke i G, kan vi permutere kjeden slik at den får lavere høyde. Teorem 4.2 kan derimot ikke hjelpe oss, da hver node i kjeden er nabo til et dypeste løv. Vi ser av dette at omflyttingsteget fra Teorem 4.2 ikke er kraftig nok til at man alltid kan permutere et vilkårlig eliminasjonstre slik at det blir minimalt.

Selv om alle tre algoritmene utnytter Teorem 4.2, er det ingen av algoritmene som utnytter det fullt ut. Vi ser i figur 4.8 et eliminasjonstre til en graf som ingen av de tre algoritmene kan gjøre lavere. Node 9 er ikke nabo til node 1 i G^*. Vi kan derfor flytte ned node 9 slik at den ikke lenger ligger på stien fra node 1 til roten i T. Det fører til at vi får redusert høyden til eliminasjonstreet med en.



Figur 4.8 Eksempel på bruk av resultatet fra Teorem 4.2.

Merk at etter omordningen av grafen i figur 4.8 kan Minimale-kuttsett algoritmen brukes, og vil føre til at vi får et eliminasjonstre av høyde 3, som også er minimalt (Korollar 3.3.1).

Vi har nå vist at alle tre algoritmene vi har sett på tilhørere en klasse av algoritmer som bruker resultatet fra Teorem 4.2. Forskjellen mellom algoritmene ligger i hvordan man velger ut den noden som skal flyttes ned. Det er mulig å lage andre algoritmer som også utnytter resultatet fra Teorem 4.2. Det som er avgjørende

for algoritmen, er etter hvilket kriterium vi velger ut den noden som skal flyttes ned. Andre kriterier man kunne lagt til grunn for en algoritme er f.eks:

1. Flytt ned den øverste noden som er mulig.
2. Flytt ned den nederste noden som er mulig.
3. Flytt ned den noden som må flyttes kortest.
4. Flytt ned den noden som etter ned-flyttingen vil være nærmest $rot(T)$.

Dersom vi etter å ha flyttet ned en node w, får et eliminasjonstre som er høyere eller like høyt som det vi opprinnelig hadde, kan vi fortsette å arbeide med $T(w)$ ut fra et håp om at vi kan redusere $h(p(w))$ så mye at den totale høyden likevel går ned. I seksjon 4.5 skal vi se nærmere på en slik algoritme.

4.5 Nederste-først algoritmen

Vi skal i denne seksjonen se på en algoritme kalt *Nederste-først algoritmen*, som også har som mål å redusere høyden til et gitt eliminasjonstre. Grunnen til at vi ser på denne algoritmen er at den gir et minimalt eliminasjonstre til en klasse av grafer kalt *linje-grafer*. Nederste-først algoritmen baserer seg også på resultatet fra Teorem 4.2.

Algoritmen består i å velge en vilkårlig lengste sti $v, ..., rot(T)$ i T, for så å flytte ned den nederste noden som ikke er nabo i G^* til et dypeste løv i T. Vi fortsetter å arbeide rekursivt med deltreet til noden som blir flyttet ned, så langt det er mulig.

Vi beskriver nå den klassen av grafer som Nederste-først algoritmen finner et minimalt eliminasjonstre til:

En graf G som har nodemengde $V(G) = \{v_1, ..., v_n\}$ og kant $(v_i, v_j) \in E(G)$ hvis og bare hvis $|i - j| = 1$, kaller vi en *linje-graf*.

Vi skal vise at Nederste-først algoritmen gir et minimalt eliminasjonstre til en linje-graf under forutsetning av at eliminasjonstreet vi starter med, er gitt ved identitetsordningen. Det er verdt å merke at algoritmene til Hafsteinsson og Liu og Minimale kuttsett algoritmen alle vil gi et eliminasjonstre av høyde $\lfloor n/2 \rfloor$ til et slikt eliminasjonstre.

Dersom G er en linje-graf kan vi finne et minimalt eliminasjonstre direkte fra G. En nøstet oppdelingsordning til G hvor vi krever at ingen delkomponent får inneholde mer enn $\lfloor k/2 \rfloor$ noder, der k er antall noder i den opprinnelige

komponenten, vil gi et eliminasjonstre av høyde $\lfloor log_2 n \rfloor$. Vi ser fra Korollar 3.3.1 at dette også er høyden til et minimalt eliminasjonstre til G. Forskjellen mellom en nøstet oppdelingsordning og Nederste-først algoritmen er at en nøstet oppdelings-ordning arbeider utifra G, mens Nederste-først algoritmen forbedrer et allerede eksisterende eliminasjonstre.

Dersom vi har et minimalt eliminasjonstre til en linje-graf G med $2^k - 1$ noder for $k \geq 0$, sier vi at eliminasjonstreet er *fullt*. Vi ser at hvis vi legger til en node til et fullt eliminasjonstre, så vil høyden til eliminasjonstreet øke med en.

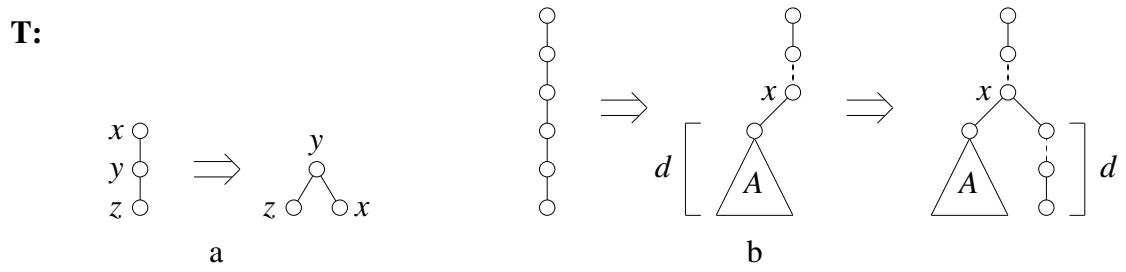
I et fullt eliminasjonstre av høyde k, vil 2^k av nodene være løv. Fra Teorem 2.2 vet vi at to noder som er nabo i G, ikke begge kan være løv. Det følger derfor at odde noder i G, vil bli løv i det minimale eliminasjonstreet. Vi ser at i et fullt eliminasjonstre vil samtlige noder være nabo i G til et nederste løv i T.

Vi skal nå gi et induksjonsbevis for at Nederste-først algoritmen gir et minimalt eliminasjonstre til en linje-graf, når vi har eliminert grafen etter identitetsordningen. I eliminasjonstreet vi starter med, ligger alle nodene på en nederste kjede, slik som i eliminasjonstreet til venstre i figur 4.9b. Når vi videre i denne seksjonen omtaler en kjede, er det underforstått at den utgjør eliminasjonstreet til en linje-graf, som er eliminert etter identitetsordningen.

Dersom kjeden vi starter med inneholder 1 eller 2 noder, utgjør den i utgangspunktet et minimalt eliminasjonstre. Fra figur 4.9a ser vi at dersom kjeden inneholder 3 noder, så vil algoritmen også klare å brette den til et eliminasjonstre som er minimalt.

Vår induksjonshypotese er: Vi kan brette en kjede av lengde n der $2^k \leq n < 2^{k+1}$ for alle $k \leq d$ der $d \geq 0$, til et eliminasjonstre av høyde k. Vi skal nå vise at hvis hypotesen stemmer, så vil algoritmen også brette en kjede av lengde n' der $2^{d+1} \leq n' < 2^{d+2}$, til et eliminasjonstre av høyde $d + 1$.

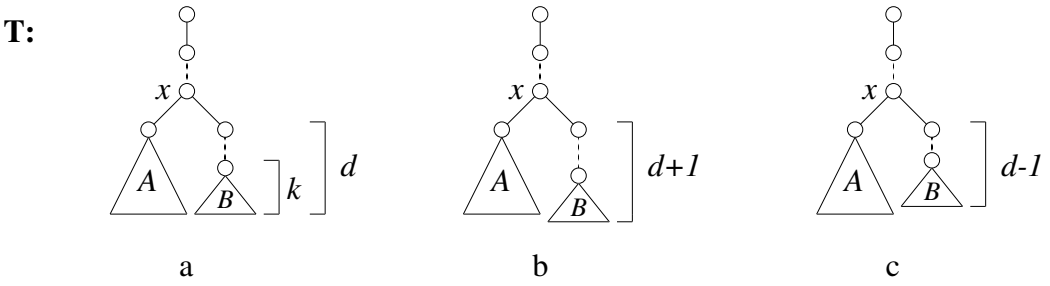
Anta at vi har en kjede med n' noder der $2^{d+1} \leq n' < 2^{d+2}$. Fra induksjonshypotesen vet vi at vi kan brette de $2^{d+1}-1$ første nodene til et fullt eliminasjonstre A av høyde d. Vi har nå fått et eliminasjonstre som det i midten i figur 4.9b. Vi vet at node $2^{d+1} - 1$ er et løv i A. Det følger derfor at node 2^{d+1}, som er markert med x i figur 4.9b, nå er nabo i G til et dypeste løv, og kan derfor ikke flyttes ned. Vi vil de $d + 1$ neste gangene, når vi skal avgjøre hvilken node som skal flyttes ned, ta utgangspunkt i et dypeste løv i A, og hver gang flytte ned $p(x)$, slik at den får x som foreldre-node. Det vil gi et eliminasjonstre som det vi ser til høyre i figur 4.9b. Merk at nå er x nabo i G til et dypeste løv fra hvert av deltrærne som henger av x.



Figur 4.9 a. Eliminasjonstrær til en linje-graf med 3 noder. b. Algoritmen gir først et tre A av høyde d. De neste $d+1$ stegene flytter vi hver gang ned $p(x)$.

La B være det høyeste korrekt sammenbrettede treet i det høyre deltreet til x. Vi ser at vi nå har $h(B) = 1$. Når begge barna til x er like høye, slik som i figur 4.10a, sier vi at eliminasjonstreet er i en *stabil tilstand*.

Vi skal vise at hvis $T(x)$ ikke er fullt, og vi starter med eliminasjonstreet i en stabil tilstand, så vil algoritmen alltid komme tilbake til en stabil tilstand etter høyst 2 iterasjoner, uten at høyden til eliminasjonstreet har økt.



Figur 4.10 a. Stabil tilstand. b. Etter bruk av et nederste løv i A. c. Etter bruk av et nederste løv i B, når $k < d$.

Anta at et dypeste løv i B er nabo i G til en av sine forfedre $y \neq x$, som ikke er med i B. I så tilfelle kan y ikke flyttes ned på sin rette plass i B. Dersom nodene i A og noden x ble fjernet fra G, vil y fremdeles være nabo til et dypeste løv i B. Siden vi da har høyst 2^{d+1} noder, ser vi fra induksjonshypotesen at inget dypeste løv i B er nabo til noen av sine forfedre som ikke er med i B.

Når eliminasjonstreet er i en stabil tilstand, er det dypeste løv i både A og B. Vi ser først på hva som skjer når vi tar utgangspunkt i et dypeste løv i A, for å bestemme hvilken node som skal flyttes ned.

Alle nodene i A og noden x er nabo i G til et dypeste løv, mens inget dypeste løv i A er nabo i G^* til $p(x)$. Vi vil derfor flytte ned $p(x)$, som vil få x som foreldrenode. Vi får da et eliminasjonstre som i figur 4.10b. Nodene i A har nå kommet

en plass nærmere roten i eliminasjonstreet, og vi må neste gang ta utgangspunkt i et dypeste løv i B, og flytte ned $p(rot(B))$. Fra induksjonshypotesen vet vi at denne noden vil bli plassert på rett plass i B. Vi vil nå være tilbake i en stabil tilstand. Det eneste som kan ha forandret seg er at $h(B)$ øker til $k+1$ dersom noden som ble flyttet ned, gjorde at B ble fullt.

Ta nå utgangspunkt i et dypeste løv i B. Dersom $k < d$, vil vi flytte ned $p(rot(B))$ og få et eliminasjonstre som i figur 4.10c. Nodene i B har nå kommet en plass nærmere roten i eliminasjonstreet. Vi må derfor i neste steg ta utgangspunkt i et dypeste løv i A, og flytte ned $p(x)$. Igjen gjelder det at dersom B ble fullt når vi flyttet ned $p(rot(B))$, så øker $h(B)$ til $k+1$. Det eneste unntaket er hvis B ble fullt og $h(B) = d-1$. Da vil ikke $h(B)$ øke før etter at vi har flyttet ned $p(x)$. Uansett så er vi i en stabil tilstand etter at $p(x)$ er flyttet ned. Dersom $k = d$ vil vi flytte ned $p(x)$ til rett plass i B uten at høyden til eliminasjonstreet øker.

Vi ser at vi kan utføre algoritmen inntil x er rot i T eller $T(x)$ er fullt, uten at høyden til eliminasjonstreet øker. Hvis $T(x)$ blir fullt er $h(x) = d+1$ og $T(x)$ vil inneholde $2^{d+2}-1$ noder. Der følger derfor at x uansett vil være rot i eliminasjonstreet. Vi ser av dette at dersom induksjonshypotesen stemmer, så klarer vi å brette en kjede av lengde n' der $2^{d+1} \leq n' < 2^{d+2}$ til et eliminasjonstre av høyde $d+1$, som også vil være minimalt.

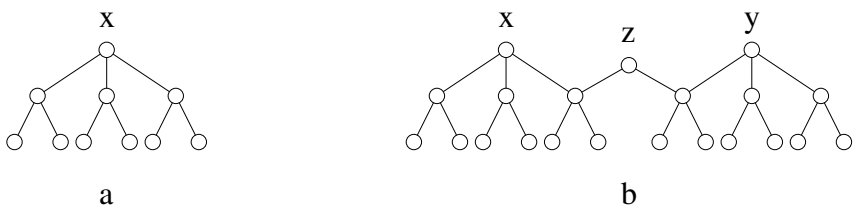
Et naturlig spørsmål å stille seg er om Nederste-først algoritmen gir minimale eliminasjonstrær til andre klasser av grafer enn linje-grafer. Linje-grafer er en undermengde av klassen av grafer som utgjør trær. Vi skal nå se at vi ikke kan bruke Nederste-først algoritmen for å få et minimalt eliminasjonstre til et tre G, der vi har eliminert G slik at det ikke oppstår fyllkanter i G^*. Skal vi få en eliminasjonsordning til et tre som ikke fører til at det oppstår fyllkanter i G^* må vi eliminere G etter følgende fremgangsmåte: Eliminer først et løv v i G. Fjern v, og gjenta prosessen rekursivt for $G - v$.

I en eliminasjonsordning som ikke gir fyllkanter i G^* vil alle løv i G også være løv i T.

Nederste-først algoritmen har den egenskapen at dersom en node i løpet av utførelsen av algoritmen blir et løv i eliminasjonstreet, så vil den også være et løv når algoritmen er ferdig.

Treet i figur 4.11a har et minimalt eliminasjonstre av høyde 2. I et slik eliminasjonstre må x elimineres sist. Dersom vi eliminerer en annen node enn x sist i en eliminasjonsordning som ikke gir noen fyllkanter, vil Nederste-først algoritmen gjøre x til et løv i T, og dermed ikke gi et minimalt eliminasjonstre.

G:



Figur 4.11 Et tre som Nederste-først algoritmen ikke gir et minimalt eliminasjonstre til.

Treet i figur 4.11b har et minimalt eliminasjonstre av høyde 3. For at vi skal få et minimalt eliminasjonstre T_{min} må z være rot i T_{min}, og ha x og y som barn. Uansett hvilken node vi eliminerer sist i en eliminasjonsordning som ikke gir noen fyllkanter, så vil bruk av Nederste-først algoritmen føre til at enten x eller y blir et løv i T. Vi får derfor inget minimalt eliminasjonstre til treet i figur 4.11b ved bruk av Nederste-først algoritmen. Dette viser at Nederste-først algoritmen ikke gir et minimalt eliminasjonstre til et vilkårlig tre som er eliminert slik at det ikke oppstår fyllkanter i G^*. Vi skal imidlertid i kapittel 5 se på en algoritme som gir et minimalt eliminasjonstre til et tre.

5 Minimale eliminasjonstrær til trær

Vi så i seksjon 4.5 at Nederste-først algoritmen gav et minimalt eliminasjonstre til en linje-graf, men ikke til et tre. Vi skal i dette kapittelet utvikle og analysere en mer avansert algoritme som gir et minimalt eliminasjonstre til et tre. I tillegg til å gi en teoretisk analyse av kjøretiden vil vi også presentere en del eksperimentelle resultater ved bruk av algoritmen. Vi skal videre vise en enklere algoritme som gir et eliminasjonstre som ikke er høyere enn $\lfloor logn \rfloor$ til et tre.

Først trenger vi en del resultater for å kunne avgjøre når et eliminasjonstre til et tre er minimalt.

5.1 Balansering av eliminasjonstrær

Vi skal i denne seksjonen v.h.a. et induksjonsbevis vise hvordan man med utgangspunkt i et vilkårlig eliminasjonstre til et tre G, kan omordne dette slik at man får et minimalt eliminasjonstre.

Vi repeterer kort at et tre er en graf som ikke inneholder noen sykler, og at nodene som bare har en nabo kalles løv. I et tre er hver node som ikke er et løv, et minimalt kuttsett. Det gjør at ethvert minimalt kuttsett til et tre alltid består av en node. En sammenhengende node-delgraf til et tre vil også være et tre.

Gitt et tre G med eliminasjonstre T. Vi skal nå gjennom induksjon på høyden til T vise at dersom alle deltrær som henger fra $rot(T)$ er minimale, så kan vi i høyst $h(T)$ "steg" omordne T slik at vi får et minimalt eliminasjonstre. Et steg består i at vi bestemmer oss for en node x og flytter denne oppover i eliminasjonstreet inntil x er rot. Mens vi flytter x oppover sørger vi hele tiden for at alle deltrær som henger fra x er minimale.

Dersom $h(T) = 0$, kan G bare inneholde en node, og T er i utgangspunktet minimalt. Dersom $h(T) = 1$ inneholder G en sti på minst 2 noder. Fra Korollar 3.3.1 vet vi at $\lfloor log2 \rfloor = 1 \leq h(T_{min})$, og det følger at T er minimalt.

Vår induksjonshypotese er: Dersom $h(T) \leq k-1$, $k > 1$, og alle deltrær som henger fra $rot(T)$ er minimale så kan vi v.h.a. samme type omordningsteg som beskrevet over, forandre T slik at vi får et eliminasjonstre som vi vet er minimalt.

Vi antar nå at $h(T) = k$ og at alle deltrær som henger fra $v = rot(T)$ er minimale. Vi skal i denne seksjonen gjennom en sekvens av lemma vise at dersom induksjonshypotesen er sann, så kan vi i høyst k steg finne et minimalt eliminasjonstre til T.

Siden alle deltrær som henger fra $v = rot(T)$ er minimale, følger det fra Korollar 3.2.1 at $h(T_{min}) = k$ eller $k-1$. Det medfører at dersom vi finner en eliminasjonsordning slik at vi får et eliminasjonstre av høyde $k-1$, så vet vi at $h(T_{min}) = k-1$.

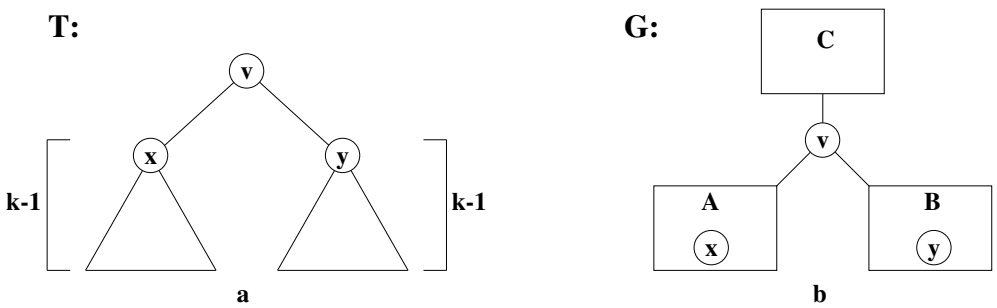
Før vi begynner å omordne T, skal vi gjennom å se på to høyeste barn til v i T forsøke å bestemme om vi har et minimalt eliminasjonstre. Til det trenger vi følgende lemma:

Lemma 5.1

Gitt et tre G med eliminasjonstre T, der alle deltrærne som henger fra $v = rot(T)$ i T er minimale. La x og y være to høyeste barn til v i T. Dersom $h(x) = h(y)$ så er T minimalt.

Bevis:

Vi setter $h(T) = k$. Da er $h(x) = h(y) = k-1$, og vi har et eliminasjonstre slik som i figur 5.1a. Grafen $G - v$ inneholder komponenter A og B slik at $x \in A$ og $y \in B$. Både A og B har minimale eliminasjonstrær av høyde $k-1$. Vi ser i figur 5.1b hvordan v deler G. Grafen $G - v$ kan inneholde flere komponenter enn A og B. For å markere eventuelle andre komponenter har vi tatt med komponenten C i figur 5.1b.



Figur 5.1 a. De to høyeste deltrærne til v har samme høyde.
b. Noden v deler G slik at x og y kommer i ulike komponenter.

Fra Teorem 3.2 vet vi at en delgraf alltid har en eliminasjonsordning som gir et eliminasjonstre som er minst like lavt som et minimalt eliminasjonstre til den opprinnelige grafen.

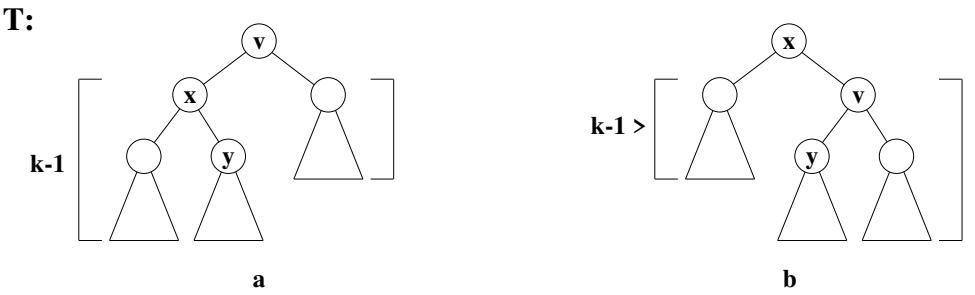
I enhver eliminasjonsordning som skal gi et eliminasjonstre til G som er lavere enn k, må en node $w \neq v$ elimineres sist. Uansett hvordan vi velger w, så vil det være en komponent i $G - w$ som inneholder alle nodene fra en av A eller B. Det

følger fra Teorem 3.2 at denne komponenten ikke kan ha et eliminasjonstre som er lavere enn $k-1$. Den totale høyden til eliminasjonstreet kan derfor ikke være lavere enn k, og vi ser av dette at T er minimalt. □

Vi ser fra Lemma 5.1 at dersom de to høyeste barna til v er minimale, så vet vi med en gang at T er minimalt. Vi antar derfor at v bare har ett barn x av høyde $k-1$.

Dersom vi velger en node $w \notin T(x)$ og eliminerer sist, så vil et deltre som henger fra w, inneholde alle nodene fra $T(x)$. Det følger fra Teorem 3.2 at dette deltreet ikke kan ha høyde mindre enn $k-1$ siden $h(x) = k-1$. Vi ser at dersom en eliminasjonsordning skal gi et eliminasjonstre av høyde $k-1$, må en node fra $T(x)$ elimineres sist.

Vi prøver først med å eliminere x sist. For at x skal bli rot i eliminasjonstreet foretar vi en *rotasjon*[1] med utgangspunkt i x. Rotasjonen består i at x elimineres rett etter v og alle andre noder elimineres i samme relative ordning som tidligere. Effekten på eliminasjonstreet er at x nå er rot. (Dersom x og v ikke er naboer i G, så vil det deltreet $T(y)$ som hengte av x, og som består av nodene i komponenten mellom x og v i G, nå henge fra v.) Vi ser i figur 5.2 effekten av å utføre rotasjonen.



Figur 5.2 Effekten av utføre en rotasjon med utgangspunkt i x. a. Før rotasjonen. b. Etter rotasjonen.

Siden $h(v)$ nå er mindre enn k, ser vi fra induksjonshypotesen at vi kan finne et minimalt eliminasjonstre til v og dens etterkommere. Vi kaller eliminasjonstreet til G, som vi får når v og dens etterkommere er gjort minimale for T'. La u være et barn til x i T' slik at $v \in T'(u)$. Noden u er nå det eneste barnet til x som kan ha høyde $k-1$. Dersom $h(u) \neq k-1$, vet vi at T' er minimalt med $h(T') = k-1$. Dersom $h(u) = k-1$, har vi fremdeles en mulighet til å avgjøre om T' er minimalt. Vi skal nå vise at dersom x og v er naboer i G,

[1] En rotasjonen er ekvivalent med en enkel rotasjon i søketrær slik Tarjan definerer det i [18].

så er T' minimalt.

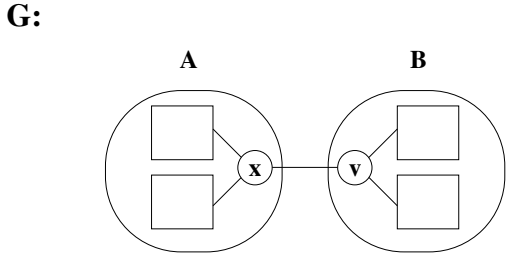
Lemma 5.2

Gitt et tre G. La x og v være to noder som er naboer i G. Anta at α er en eliminasjonsordning slik at x elimineres sist og at alle deltrær som henger fra x er minimale. La y være et barn til x slik at $v \in T(y)$. Vi setter $j = h(y)$. Tilsvarende er β en eliminasjonsordning slik at v elimineres sist, og alle deltrær som henger fra v er minimale, og w er et barn til v slik at $x \in T(w)$. Vi setter $l = h(w)$.

Dersom $j = l$, så gir både α og β et minimalt eliminasjonstre.

Bevis:

For å skille mellom eliminasjonstrærne til α og β, skriver vi T_α og T_β. Vi ser i figur 5.3 hvordan vi kan dele inn G i to komponenter A og B. En node er enten med i $T_\alpha(y)$ eller $T_\beta(w)$. Det følger ettersom komponenten A har samme nodemengde som $T_\beta(w)$ og komponenten B har samme nodemengde som $T_\alpha(y)$.



Figur 5.3 En node er enten med i A eller B.

Vi antar at $j = l$. Vi viser først at da er $h(T_\alpha) = j + 1$. Siden nodene fra alle deltrær som henger fra x i T_α bortsett fra nodene i $T_\alpha(y)$, er med i $T_\beta(w)$, følger det fra Teorem 3.2 at inget deltre som henger fra x i T_α kan være høyere enn j. Vi ser av dette at $h(T_\alpha) = j + 1$. Tilsvarende kan vi vise at $h(T_\beta) = j + 1$.

I en eliminasjonsordning som skal gi et eliminasjonstre av høyde $\leq j$, må noden z som elimineres sist være forskjellig fra x og v. Vi ser at z enten må være med i $T_\alpha(y)$ eller $T_\beta(w)$. Uansett hvordan vi velger z, så vil et deltre som henger fra z, inneholde minst nodene fra en av $T_\alpha(y)$ eller $T_\beta(w)$. Fra Teorem 3.2 ser vi at dette deltreet ikke kan ha minimal høyde mindre enn k. Det følger at et minimalt eliminasjonstre til G har høyde $j + 1$, og at både α og β gir et minimalt eliminasjonstre til G. $\square$

Dersom x og v ikke er naboer i G, kan vi fremdeles ikke si om vi har et minimalt eliminasjonstre eller ikke. Vi vet at dersom det eksisterer en eliminasjonsordning som gir et eliminasjonstre av høyde $k-1$, så må en node fra $T'(u)$ elimineres sist. Vi har tidligere vist at en slik node også må ligge i $T(x)$. Det følger at denne noden må ligge i komponenten mellom x og v i G. Vi skal nå vise at vi bare trenger å lete etter en slik node på stien $x, ..., v$ i G.

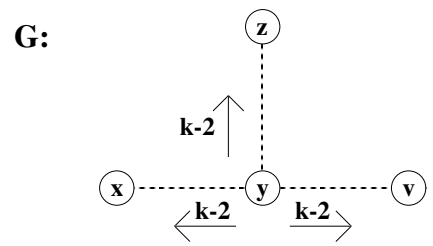
Lemma 5.3

Gitt et tre G med eliminasjonstre T der $h(T) = k$, og alle deltrær som henger fra $v = rot(T)$, er minimale. Anta at $h(T_{min}) = k-1$, og at x er det eneste barnet til v i T av høyde $k-1$. Da eksisterer det en eliminasjonsordning som gir et minimalt eliminasjonstre der en node fra stien $x, ..., v$ i G elimineres sist.

Bevis:

Vi vet at dersom det ikke finnes en eliminasjonsordning som gir et minimalt eliminasjonstre der x elimineres sist, så eksisterer det en eliminasjonsordning som gir et minimalt eliminasjonstre der en node z fra komponenten mellom x og v i G elimineres sist. Anta at z ikke ligger på stien $x, ..., v$ i G. La y være den første noden fra stien $z, ..., x$ i G som også ligger på stien $x, ..., v$ i G. Vi ser i figur 5.4 hvordan y er plassert i G. La videre α være en eliminasjonsordning der z elimineres sist og $h(T_\alpha) = k-1$. Vi skal nå vise at en eliminasjonsordning β der y elimineres sist, og alle deltrær som henger fra y er minimale, også gir et eliminasjonstre av høyde $k-1$.

La w være et barn til y i T_β slik at $z \in T_\beta(w)$. Siden det deltreet som henger fra z i T_α som inneholder y har høyde $\leq k-2$, følger det at ingen barn $\neq w$ til y i T_β kan ha høyde $> k-2$. Vi vet at nodene i $T_\beta(w)$ utgjør en delmengde av nodene mellom x og v i G. I det opprinnelige eliminasjonstreet T har disse nodene et minimalt eliminasjonstre av høyde $\leq k-2$. Det følger derfor at $h(T_\beta(w)) \leq k-2$. Figur 5.4 viser høyden til de ulike barna til y i T_β. (Verdien ved pilene angir en øvre grense på høyden til deltreet av T_β til den komponenten som det pekes på.)



Figur 5.4 Ingen av deltrærne som henger fra y i T_β har høyde $> k-2$.

Siden alle barn til y i T_β har høyde $\leq k-2$, følger det T_β er minimalt. $\square$

Vi vet nå at dersom $h(T_{min}) = k-1$, så eksisterer det en eliminasjonsordning som gir et minimalt eliminasjonstre der en node z fra stien $x, ..., v$ i G elimineres sist. Vi skal nå vise at dette også gjelder dersom $h(T_{min}) = k$.

Dersom det eksisterer en eliminasjonsordning som gir et minimalt eliminasjonstre av høyde k som tilfredstiller Lemma 5.1, så må en node fra komponenten mellom x og v i G elimineres sist. Anta at roten z i et slikt eliminasjonstre T ikke ligger på stien $x, ..., v$ i G. Det vil da henge et deltre $T(w)$ fra z som inneholder både x og v. Alle andre deltrær som henger fra z vil utelukkende bestå av noder fra komponenten mellom x og v i G. Vi vet at denne komponenten har et minimalt eliminasjonstre av høyde høyst $k-2$. Det følger derfor at ingen av deltrærne som henger fra z, bortsett fra $T(w)$, kan ha høyde $> k-2$. Siden z ikke kan ha to barn av høyde $k-1$, ser vi at en eventuell rot i et slikt minimalt eliminasjonstre må ligge på stien $x, ..., v$ i G.

Anta nå at $h(T_{min}) = k$ og at ingen eliminasjonsordning gir et eliminasjonstre som tilfredstiller Lemma 5.1. Det vil da for hver node z fra stien $x, ..., v$ i G som elimineres sist, være eksakt et minimalt deltre som henger fra z av høyde $k-1$. Dette deltreet inneholder enten x eller v. Vi vet at når vi eliminerer x sist, så er v i det eneste minimale deltreet av høyde $k-1$. Tilsvarende når vi eliminerer v sist, er x i det eneste minimale deltreet av høyde $k-1$. Det følger at det er to noder på stien $x, ..., v$ i G som er naboer i G slik at når hver av nodene elimineres sist, så er den andre noden i det minimale deltreet av høyde $k-1$. Fra Lemma 5.2 ser vi at vi har et minimalt eliminasjonstre.

Vi ser av dette at uansett om $h(T_{min}) = k$ eller $k-1$ så eksisterer det alltid et minimalt eliminasjonstre med en node fra stien $x, ..., v$ i G som rot.

Dersom vi ikke vet om eliminasjonstreet T' som vi har er minimalt, skal vi nå vise hvordan vi kan omordne T' slik at naboen z til x i G som ligger på stien $x, ..., v$ i G blir rot, og alle deltrær som henger fra z er minimale. Vi gjentar at dersom vi ikke kan avgjøre om T' er minimalt så har $x = rot(T')$ bare et barn u av høyde $k-1$. Deltreet $T'(u)$ inneholder alle noder på stien $x, ..., v$ i G bortsett fra x.

For å få et eliminasjonstre med z som rot foretar vi rotasjoner med utgangspunkt i z inntil z blir rot. Etter hver rotasjon sørger vi for at alle deltrær som henger fra z er minimale. Siden hver rotasjon flytter z en plass nærmere x, må vi høyst foreta k rotasjoner. Etter hver rotasjon kan vi risikere å måtte minimalisere et deltre med w som rot, som henger fra z, der alle deltrær som henger fra w er minimale.

Dersom z har dybde j etter en rotasjon, $1 \leq j < k$, så er $h(w) \leq k-j$. Siden det nesthøyeste barnet til x i T' har høyde $< k-1$ følger det at $h(x) \leq k-1$ etter rotasjonen som gjør at z blir rot. Vi kan derfor i følge induksjonshypotesen finne et minimalt eliminasjonstre til x og dens etterkommer etter den siste rotasjonen.

Dersom vi fremdeles ikke kan avgjøre om vi har et minimalt eliminasjonstre kan vi rotere opp en og en av nodene på stien $x, ..., v$ i G slik at den blir rot. Vi vil på denne måten tilslutt finne et minimalt eliminasjonstre.

Med dette er induksjonsbeviset ferdig. Vi ser at dersom vi har et eliminasjonstre av vilkårlig høyde der alle deltrær som henger fra roten er minimale, så kan vi v.h.a. rotasjoner omordne eliminasjonstreet slik at vi får et eliminasjonstre som vi vet er minimalt. Vi lovet imidlertid innledningsvis i denne seksjonen at dersom vi startet med et eliminasjonstre av høyde k så skulle det ikke være nødvendig å rotere opp mer enn k noder slik at de blir rot.. Vi skal nå se hvordan vi kan få dette til.

Vi går tilbake til tilstanden når vi hadde T' med x som rot, og x bare hadde et barn av høyde $k-1$. Istedenfor å rotere opp x sin nabo i G som ligger på stien $x, ..., v$ i G, tar vi noden y som ligger mitt på stien $x, ..., v$ i G og roterer opp slik at den blir rot. I den siste rotasjonen før y blir rot, vil det deltreet som henger fra y og som utgjøres av nodene mellom y og x i G, få x som foreldre-node. Siden nodene i dette deltreet er en delmengde av nodene i komponenten mellom x og v i G, følger det at dette deltreet ikke kan ha høyde $> k-2$. Vi vet også at det nesthøyeste barnet til x i T' er lavere enn $k-1$. Det følger at $h(x) \leq k-1$ like etter rotasjonen som gjør at y blir rot. Vi kan derfor gjøre deltreet med x som rot, minimalt etter den siste rotasjonen. Vi tester så om $h(y) = k-1$ eller om to av de høyeste barna til y er like høye. Hvis ikke ser vi om x er en etterkommer til det barnet q til y som har høyde $k-1$. I så fall trenger vi nå bare å prøve noder fra stien $x, ..., y$ i G som rot. Tilsvarende trenger vi bare å prøve noder fra stien $y, ..., v$ i G dersom v er en etterkommer til q. Slik kan vi hver gang halvere antall noder som står igjen å prøve som rot. Anta at vi aldri finner et eliminasjonstre av høyde $k-1$ eller et eliminasjonstre der to av de høyeste barna som henger fra roten er like høye. Vi vil da tilslutt finne to eliminasjonstrær som har eliminasjonsordninger som tilfredstiller betingelsene i Lemma 5.2.

Vi skal nå se hvor mange noder fra stien $x, ..., v$ som vi kan risikere å måtte rotere opp. Først prøver vi x. Vi kan deretter risikere å måtte gjøre et binært søk på stien $x, ..., v$. Dersom antall noder $\neq x, v$ på stien $x, ..., v$ i G er l, så må vi høyst prøve ytterligere $\lceil log\,(l+1) \rceil$ noder som rot før vi med sikkerhet kan si at vi har et minimalt eliminasjonstre. Siden $h(x)$ opprinnelig var $k-1$, følger det fra

Korollar 3.3.1 at $l \leq 2^{k-1} - 1$. Vi må derfor høyst prøve $\lceil log\left(2^{k-1}\right)\rceil + 1 = k$ noder som rot.
Sammen med induksjonsbeviset gir dette oss:

Teorem 5.1
Gitt et tre G med eliminasjonstre T der $h(T) = k$. Dersom alle deltrærne som henger fra $v = rot(T)$ er minimale, må vi høyst rotere opp k noder slik at de blir rot før vi har et minimalt eliminasjonstre. □

For et tre G med vilkårlig eliminasjonstre T danner Teorem 5.1 grunnlaget for en algoritme som finner et minimalt eliminasjonstre til G ut fra T. Algoritmen består i å finne minimale eliminasjonstrær til alle delgrafer til T av høyde j for $j = 2$ til $h(T)$. Kjøretiden til en slik algoritme vil være avhengig av høyden til det opprinnelige eliminasjonstreet. Vi skal derfor i seksjon 5.3 se hvor lavt vi kan gjøre eliminasjonstreet før vi starter algoritmen.

5.2 Algoritme

Vi skal i denne seksjonen gi pseudo-koden til en algoritme som finner et minimalt eliminasjonstre til et tre. Algoritmen består grovt sett i at vi først minimaliserer alle deltrær av høyde 1 for deretter å minimalisere alle av høyde 2, etc. For å avgjøre om et eliminasjonstre er minimalt, bruker vi resultatene fra seksjon 5.1.

Min_Tre
Funksjonen Min_Tre er den øverste funksjonen. Den tar et vilkårlig eliminasjonstre T til et tre G, og omordner det slik at T blir minimalt. Grafen G er en global variabel. Hver node i T har en d-haug der den til enhver tid har ordnet barna sine etter avtagende høyde. Det gjør at vi raskt kan finne to høyeste barn til en node. Min_Tre holder orden på hvilke deltrær som er klare for å gjøres minimale. Dette er de deltrærne der alle deltrær som henger fra roten er minimale. I utgangspunktet vet vi at alle deltrær av høyde 2 er klare for å gjøres minimale. Roten til hver av disse legges inn i en kø, og deltrærne mates ett og ett til funksjonen Lag_Min. Lag_Min foretar den virkelige omordningen slik at deltrærne blir minimale. For hvert deltre $T(v)$ som blir gjort minimalt returnerer Lag_Min noden y som er rot. La z være foreldre-noden til v. Kallet $y =$ Lag_Min($T(v)$) kan forandre hvilken node som er rot i dette deltreet og høyden kan også bli redusert med en. Dersom $y \neq rot(T)$, må vi derfor oppdatere haugen med barn til z. Dette steget foretas i

Bytt_Barn(T, z, v, y), som setter inn y for v i haugen av barn til z og omordner denne. Dersom alle deltrær som henger fra z dessuten er minimale, så legges z inn bakerst i køen. For at denne testen skal gå raskt, har hver node en teller for hvor mange av dens barn som er rot i et minimalt deltre.

Algoritmen stopper når alle deltrær til T er minimale. Her følger selve koden til Min_Tre:

```
Min_Tre(T)
{
  kø av noder k;
  node v, x, y, z;

  Sett_Tom(k);
  For hver node x der h(x) = 2
    k = k & x;

  Så lenge |k| > 0
  {
    v = k[1];
    k = k[2...];
    z = p(v);
    y = Lag_Min(T(v));
    Dersom y ≠ rot(T)
    → {
        p(y) = z;
        Bytt_Barn(T, z, v, y);
        Dersom alle deltrær som henger fra p(y) er minimale
        → k = k & p(y);
      }
  }
}
```

Lag_Min

Funksjonen Lag_Min får som nevnt inn et eliminasjonstre T der alle deltrær som henger fra $v = rot(T)$ er minimale. Eliminasjonstreet blir omordnet v.h.a. resultatene fra seksjon 5.1 slik at det blir minimalt.

Først tester vi om T allerede er minimalt. Hvis ikke roterer vi opp det høyeste barnet x til v. Det gjør vi v.h.a. funksjonen Roter som gitt x roterer x opp en plass og gjør alle deltrær som henger fra x minimale. Roter returnerer det barnet u til x som er forfader til v etter rotasjonen.

Dersom T fremdeles ikke er minimalt legger vi inn nodene på stien $v, ..., x$ i G i en tabell. Vi vet at det er minst 3 noder på denne stien. Som indekser i tabellen bruker vi i og j som henholdsvis nedre og øvre grense for noder som allerede er prøvd som rot. Vi tar den noden y som til enhver til ligger mitt imellom i og j i tabellen og roterer opp slik at den blir rot. Vi tester så om eliminasjonstreet vi har er minimalt. Hvis ikke flytter vi i eller j til å peke på y i tabellen og gjentar prosessen.

Vi kan avgjøre hvilken av i og j som skal flyttes ut fra høyden til de nodene de peker på og resultatet av det siste kallet til Roter. Før vi begynner å rotere opp y, peker enten i eller j på noden som er rot. Anta at det er i som peker på roten. Det siste kallet til Roter returnerer noden u som er barn til y, og som er forfader til noden som i peker på. Dersom u er det eneste barnet til y av høyde $h(y) - 1$, setter vi j til å peke på y. Hvis ikke setter vi i til å peke på y. Tilfellet når j opprinnelig peker på noden som er rot, er tilsvarende.

Når $j = i + 1$ er det ingen noder igjen å prøve, og vi vet fra Lemma 5.2 at vi har et minimalt eliminasjonstre.

node Lag_Min(T)
{
  node s[1..n] u, v, x, y;
  heltall i, j, hi, hj, hv;

  v = $rot(T)$;
  Dersom $h(v) < 2$ eller Balanserer(v, T)
  → Returner(v);

  hv = $h(v)$;
  x = Høyeste barn til v;
  u = Roter(x, T)

Dersom Balanserer(x, T) eller Krympet(x, hv, T) eller Naboer(v, x)
$\rightarrow$ Returner(x);

s = Legg_inn_sti(v, x);
$j = |s|$;
$i = 1$;

Så lenge $j \neq i + 1$
{
  $y = s[\lfloor (i + j)/2 \rfloor]$;
  $hi = h(s[i])$;
  $hj = h(s[j])$;

  Så lenge $y \neq rot(T)$
    u = Roter(y, T);

  Dersom Krympet(y, hv, T) eller Balanserer(y, T)
  $\rightarrow$ Returner(y);

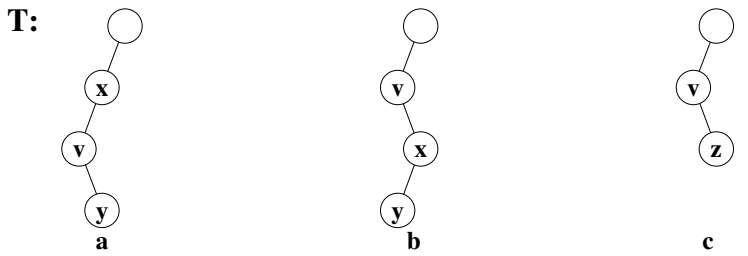
  Dersom $h(u) = h(y) - 1$ og $hi < hj$ eller
  dersom $h(u) \neq h(y) - 1$ og $hi > hj$
  $\rightarrow$ $i = \lfloor (i + j)/2 \rfloor$
  $\square$ $j = \lfloor (i + j)/2 \rfloor$;
}
Returner(y);
}

Roter

Funksjonen Roter flytter en gitt node y en plass opp i T samtidig som den gjør alle deltrær som henger fra y minimale. Vi forutsetter at $y \neq rot(T)$, og at alle deltrær som henger fra y og $z = p(y)$, er minimale.

Først bytter y og z plass i T. Vi tester så om y og z er naboer i G. Hvis ikke må deltreet $T(w)$ som henger fra y, og som utgjøres av nodene i komponenten mellom z og y i G, flyttes over fra y til z. Alle deltrær som nå henger fra z, er minimale. Vi kan derfor utføre Lag_Min($T(z)$). Nå er alle deltrær som henger fra y minimale. Roter returnerer det barnet u til y som er forfader til z.

Når et deltre $T(w)$ flyttes over fra en node y til en annen node z, må haugen med barna til y og z oppdateres. Det gjøres med funksjonene Fjern_Barn(T, y, w) og Får_Barn(T, z, w) som fjerner w som barn til y og legger til w som barn til z. Vi bruker også funksjonen Bytt_Barn når en node både får og fjerner et barn. Figur 5.5 viser effekten av å utføre Roter.



Figur 5.5 a. Før rotasjonen. b. Etter at y og z har byttet plass. c. Etter at $T(z)$ er gjort minimalt.

node Roter(y, T)
{
  node z, w, u;

  $z = p(y)$;
  Dersom $z \neq rot(T)$
  $\rightarrow$ $p(y) = p(z)$
  $\square$ $p(y) = y$;
  $p(z) = y$;

  Dersom ikke Naboer(y, z)
  $\rightarrow$ {

w = Mellom_Barn(T, y, z);
$p(w)$ = z;
Fjern_Barn(T, y, w);
Bytt_Barn(T, z, y, w);
}
□ Fjern_Barn(T, z, y);

Dersom $y \neq rot(T)$
→ Bytt_Barn($T, p(y), z, y$);

u = Lag_Min($T(z)$);
Får_Barn(T, y, u);
Returner(u);
}

Mellom_Barn

Mellom_Barn tar to noder y og z som ikke er naboer i G, og returnerer det barnet w til y i T som ligger i komponenten mellom y og z i G. For at dette skal være mulig, forutsettes det at y er et barn til z i T.

Vi skal nå forklare hvordan vi finner denne noden. Dersom vi hadde en node som vi visste lå i $T(w)$, så kunne vi fulgt foreldrepekere inntil vi fant w.

For at vi raskt skal finne en slik node, går vi ut fra at hver node x vet hvor mange noder det er i hver komponent av $G - x$. For hver komponent A i $G - x$ har x lagret antall noder i A sammen med sin nabo $v \in A$. Vi benevner antall noder i A med $vekt_x(v)$. Vi skal se i seksjon 5.3 hvordan vi finner $vekt_x$ for hver node x.

La ty og tz være tyngste nabo til henholdsvis y og z. Dersom $vekt_y(ty) \geq vekt_z(tz)$ må ty ligge på stien $y, ..., z$ i G. Hvis ikke ville komponenten som inneholder ty i $G - y$ være en del av en komponent i $G - z$. Denne komponenten i $G - z$ vil også inneholde y slik at den er tyngre enn $vekt_y(ty)$. Tilsvarende gjelder dersom $vekt_z(tz) \geq vekt_y(ty)$.

Vi kan nå finne en node som ligger i $T(w)$. Dersom $vekt_z(tz) \geq vekt_y(ty)$, så velger vi tz, og ellers ty. Funksjonskallet Tyngste_Nabo(z) returnerer den naboen til z som har størst vekt.

Mellom_Barn(T, y, z)
{
  node w;

  Dersom $vekt_y$(Tyngste_Nabo(y)) $\geq$ $vekt_z$(Tyngste_Nabo(z))
  $\rightarrow$ w = Tyngste_Nabo(y)
  $\square$ w = Tyngste_Nabo(z);

  Så lenge $p(w) \neq y$
  $\rightarrow$ $w = p(w)$;
  Returner(w);
}

Legg_Inn_Sti

Denne funksjonen tar to noder v og x og returnerer en tabell som inneholder alle nodene på stien $v, ..., x$ i G.

Også denne funksjonen gjør bruk av at hver node v i G vet hvor mange noder det er i hver komponent av $G - v$. La tv og tx være tyngste nabo til henholdsvis v og x. Dersom $vekt_v(tv) \geq vekt_x(tx)$, så ligger tv på stien $v, ..., x$ i G. Hvis ikke ligger tx på stien.

Dette gjør at vi kan finne nodene på stien $v, ..., x$ i G med start både fra v og x. Vi har to tabeller $stiv$ og $stix$ hvor vi lagrer noder fra stien $v, ..., x$ i G startende fra henholdsvis v og x.

Anta at $vekt_v(tv) \geq vekt_x(tx)$. Vi legger da inn tv i $stiv$ og setter v til å peke på tv. Situasjonen når tx er tyngst er tilsvarende. Dette gjentar vi inntil $v = x$. Når dette skjer, har vi lagret hele stien. Den korteste av $stiv$ og $stix$ reverseres og konkateneres med den lengste. Når vi legger sammen tabellene, må vi ta hensyn til at den siste noden i begge tabellene vil være make.

Vi har nå en tabell som inneholder alle nodene på stien $v, ..., x$ i G for de opprinnelige verdiene av v og x.

Legg_Inn_Sti(v, x)
{
  heltall av, ax;
  node $tv, tx, stiv[1..n], stix[1..n], stivx[1..n]$;

$av = ax = 1$;
$stiv[1] = v$;
$stix[1] = x$;

Så lenge $v \neq x$
{
  $tv =$ Tyngste_Nabo(v);
  $tx =$ Tyngste_Nabo(x);
  Dersom $vekt_v(tv) \geq vekt_x(tx)$
  $\rightarrow$ {
      $av = av + 1$;
      $v = stiv[av] = tv$;
    }
  $\square$ {
      $ax = ax + 1$;
      $x = stix[ax] = tx$;
    }
}

Dersom $av \geq ax$
$\rightarrow$ $stivx =$ Konkat($stiv, av,$ Reverser($stix, ax$),ax)
$\square$ $stivx =$ Konkat($stix, ax,$ Reverser($stiv, av$),av)
Returner($stivx$);
}

Bytt_Barn, Får_Barn, Fjern_Barn

Disse funksjonene holder orden på haugene med barn til hver node. Haugene er ordnet etter høyden til barna. Bytt_Barn bytter ut et barn med et annet barn og omordner haugen. Får_Barn legger til en node til en haug og Fjern_Barn fjerner en. Dersom høyden til det høyeste barnet endrer seg, oppdateres høyden til foreldre-noden. Det er ikke gitt noen implementasjon av disse funksjonene.

Balanserer, Krympet og Naboer

Balanserer og Krympet er to boolske funksjoner som avgjører om vi har et minimalt eliminasjonstre etter henholdsvis Lemma 5.1 og Korollar 3.2.1.

Naboer(x, y) er en boolsk funksjon som returnerer sann dersom x og y er naboer i G, og falsk hvis de ikke er det. Det er ikke gitt noen implementasjon av Naboer.

boolsk Balanserer(x, T)
{
Dersom x har minst to barn i T og de to høyeste barna til x har samme høyde
$\rightarrow$ Returner(Sann)
$\square$ Returner(Falsk);
}

boolsk Krympet(x, hv, T)
{
Dersom $h(x) = hv - 1$
$\rightarrow$ Returner(Sann)
$\square$ Returner(Falsk);
}

5.3 En algoritme som finner et eliminasjonstre av høyde $\lfloor logn \rfloor$ til et tre

I denne seksjonen skal vi utvikle og analysere en algoritme som finner en nøstet oppdelingsordning som gir et eliminasjonstre av høyde $\leq \lfloor log\, n \rfloor$ til et tre. At et slikt eliminasjonstre alltid eksisterer, ser vi fra følgende resultat om separatorer i trær som Gilbert har vist i [3]:

Teorem 5.2
Et tre G har alltid en node x slik at ingen komponent i $G - x$ inneholder mer enn $\lfloor \frac{n}{2} \rfloor$ noder. $\square$

Vi ser fra Teorem 5.2 at et tre alltid har et eliminasjonstre av høyde $\leq \lfloor log\, n \rfloor$. Fra Korollar 3.3.1 vet vi at en linje-graf har et minimalt eliminasjonstre av høyde $\lfloor log\, n \rfloor$. Det følger derfor at dette er den beste øvre grensen vi kan gi på høyden til et minimalt eliminasjonstre til et vilkårlig tre.

Gitt et tre G. En algoritme som finner en eliminasjonsordning som gir et eliminasjonstre til G av høyde $\leq \lfloor log\, n \rfloor$ er som følger: Finn en node x som deler G

slik at ingen komponent i $G - x$ inneholder mer enn $\lfloor \frac{n}{2} \rfloor$ noder. Eliminer x sist, og gjenta prosessen rekursivt for hver komponent av $G - x$.

Vi kaller denne algoritmen for $\frac{n}{2}-$algoritmen. Vi skal nå gi en mer detaljert beskrivelse av denne. For å kunne avgjøre hvilken node som skal elimineres sist, må vi for hver node v vite hvor mange noder det er i hver komponent av $G - v$. Vi kaller antall noder i hver komponent av $G - v$ for dens *vekt*. For at v skal ha oversikt over vekten til de ulike komponentene i $G - v$ legger vi inn vekten til hver komponent A sammen med nabo-pekeren fra v til z der $z \in A$. Vi vil derfor videre omtale vekten til A som vekten til z. Vi skriver $vekt_v(z)$ for den vekten v har lagret for z.

Vi skal nå vise at gjennom å traversere grafen to ganger kan vi finne $vekt_v$ for hver node v.

Algoritmen begynner med at hvert løv v i G forteller sin nabo x at $vekt_x(v) = 1$. Dersom x nå kjenner vekten til alle sine naboer unntatt en node z, så forteller x nå z hvor mye de andre naboene til x veier. Dette gjentar seg så for z. Vi vil til slutt stå igjen med en node y som kjenner vekten til alle sine naboer. Noden y forteller nå hver nabo w hvor mye y veier. Nå kjenner w vekten av alle sine naboer og kan i sin tur fortelle sine naboer, bortsett fra y, hvor mye w veier. Dette fortsetter inntil hver node v i G vet hvor mye hver komponent i $G - v$ veier.

Senere i algoritmen ønsker vi at det skal være lett for hver node å finne sin tyngste nabo. Vi ordner derfor nabovektene til hver node i en d-haug etter avtagende verdi.

Når dette er gjort, kan vi nå finne rekkefølgen som nodene skal elimineres i. Vi begynner med en vilkårlig node x i G. Dersom den tyngste naboen z til x har $vekt_x(z) > \lfloor \frac{n}{2} \rfloor$ flytter vi til z. Slik fortsetter vi inntil vi finner en node y der ingen nabo til y veier mer enn $\lfloor \frac{n}{2} \rfloor$. Vi eliminerer y sist. Denne prosessen gjentar vi rekursivt for hver komponent av $G - y$. Vi begynner gjennomgangen av hver komponent $A \in G - y$ med den noden $v \in A$ som y er nabo til. Vi setter først $n = vekt_y(v)$ og fjerner y fra haugen av nabovekter til v. Når vi nå beveger oss fra en node s til en annen node t i A må vi hver gang oppdatere $vekt_t(s)$. Det gjør vi gjennom at hver node x har lagret hvor mange noder det var i komponenten sist vi var innom x. Vi kaller dette tallet for $tv(x)$. I utgangspunktet er $tv(x) = n$ for alle noder. Vi setter først $tv(t) = tv(s)$. Når t nå skal oppdatere vekten av s, setter den $vekt_t(s) = tv(t) - vekt_s(t) + 1$. Vi må nå foreta en siftdown på $vekt_t(s)$. Når dette er gjort, vil t ha en korrekt ordnet haug over vektene til sine naboer.

Vi kan også lage selve eliminasjonstreet samtidig som vi finner eliminasjonsordningen. Anta at vi bruker en rekursiv algoritme. Vi eliminerer en node i hvert kall, og kaller opp algoritmen for hver ny komponent. Hver gang vi har funnet

den noden som skal elimineres sist i en komponent, setter vi dens foreldrepeker til den noden som ble eliminert i kallet over. Vi må spesial-behandle noden som blir eliminert i det første kallet. Når algoritmen er ferdig vil foreldrepekerene fullstendig spesifisere eliminasjonstreet.

Vi skal nå gi en analyse av kjøretiden til algoritmen. Den første delen av algoritmen består i å først finne alle løv i G for deretter å traversere grafen 2 ganger. Vi skal vise at hver gjennomgang av grafen tar tid $O(n)$.

I den første gjennomgangen må vi sette $n+1$ vekter. For hver $vekt_x$ vi setter, må vi teste om x kjenner vekten til alle naboer unntatt en. Dersom x vet hvor mange naboer den har, og har en teller for hvor mange av disse den kjenner vekten til, kan vi gjøre dette i tid $O(1)$. Når x bare har en nabo den ikke kjenner vekten til, prøver vi alle naboene for å finne hvilken dette er. Hver kant gir opphav til 2 tester slik at vi totalt ikke må foreta mer enn $2n$ tester. Når x har lokalisert den aktuelle naboen y, må x fortelle y hvor mye x veier. Dersom x har akkumulert vektene for de naboene den kjenner, så blir $vekt_y(x)$ dette tallet pluss en. Vi ser av dette at den første gjennomgangen tar tid $O(n)$.

I den andre gjennomgangen gjenstår det for hver node x å sette vekten til en nabo y. Dersom summen av vektene til de naboene x kjenner er l, så blir $vekt_x(y) = n - l - 1$. Det følger at den andre gjennomgangen også tar tid $O(n)$.

For en gitt verdi av d tar det totalt $O(n)$ tid for at hver node skal lage en d-haug av vektene til sine naboer (Tarjan [18]).

Vi ser nå på den andre delen av algoritmen som finner selve eliminasjonsordningen. Vi antar først at alle haug-operasjoner tar tid $O(1)$. Når vi skal finne den noden som skal elimineres sist kan vi komme til å traversere høyst $\lfloor \frac{n}{2} \rfloor$ noder. Vi står nå igjen med minst 2 komponenter som tilsammens inneholder $n-1$ noder. I neste steg må vi høyst traversere $\lfloor \frac{n-1}{2} \rfloor$ noder og må høyst fjerne 2 noder fra grafen. Generelt må vi gang j høyst traversere $\frac{1}{2}\left(n - \sum_{i=0}^{j-1} 2^i\right) = \frac{1}{2}(n - 2^j + 1)$ noder. Totalt må vi høyst gjennomføre $\lfloor log\, n \rfloor + 1$ steg. Det gir en total tid på

$$\frac{1}{2} \sum_{j=1}^{\lfloor log\, n \rfloor + 1} (n - 2^j + 1) =$$

$$\frac{1}{2}(\lfloor log\, n \rfloor + 1)(n+1) - 2^{\lfloor log\, n \rfloor + 1} + 1.$$

Dette gir en øvre grense for å finne eliminasjonsordningen på $O(n log\, n)$ når alle haug-operasjoner er regnet som $O(1)$. Vi ser nå hva haug-operasjonene koster.

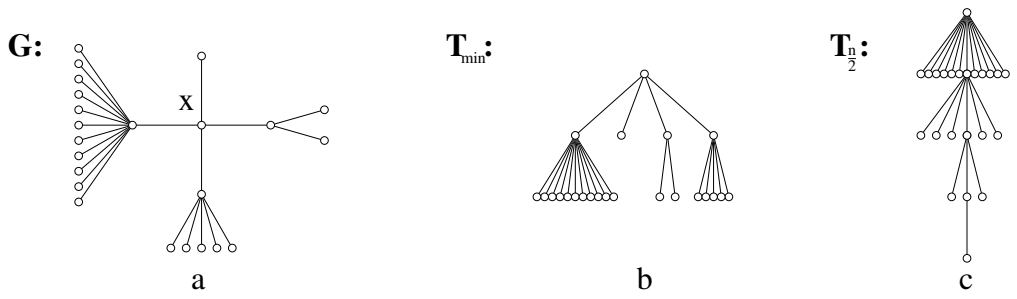
Vi må høyst slette n noder og foreta siftdown $O(nlogn)$ ganger. Ingen haug inneholder mer en n noder. For en gitt verdi av d vil hver haug-operasjon derfor ikke ta mer tid enn $O(logn)$. Det følger at den totale kjøretiden til den andre delen av algoritmen ikke er større enn $O(n(logn)^2)$. Siden den første delen av algoritmen ikke tok mer enn $O(n)$ tid, blir $O(n(logn)^2)$ også en øvre grense på kjøretiden til hele algoritmen.

Vi skal nå se på plassbehovet til $\frac{n}{2}-$algoritmen. Vi trenger ikke lagre mer enn konstant mengde informasjon per kant for å gjennomføre algoritmen. Siden G er et tre med n noder og $n-1$ kanter trenger vi ikke mer enn $O(n)$ plass.

Siden $\frac{n}{2}-$algoritmen gir oss et eliminasjonstre av høyde $\leq \lfloor log\, n \rfloor$ kan vi gi en øvre grense på antall fyllkanter vi får i den fylte grafen G^*. Vi vet at en node ikke kan ha fyllkanter til andre noder enn sine etterkommere og forfedre i eliminasjonstreet. Vi får derfor en øvre grense på antall fyllkanter gjennom å sumere antall forfedre hver node har. For å få dette tallet størst mulig må eliminasjonstreet være så høyt som mulig. For at en node skal ha høyde l, må den ha minst $2^l - 1$ noder under seg. Det gir oss at dersom $k = \lfloor log\, n \rfloor$, så vil vi ha mindre enn 2^{l+1} noder med dybde l. Disse nodene vil gi opphav til høyst l kanter hver. En øvre grense for antall fyllkanter blir da $\sum_{l=1}^{k} l * 2^{l+1} = (k-1)2^{k+2} + 4$. Vi ser av dette at antall fyllkanter i G^* er begrenset av $O(nlogn)$.

Denne grensen er sannsynligvis et overestimat. Vi har i analysen regnet med at hver node har fyllkanter til alle sine etterkommere. Dette er ikke tilfellet. En node x av høyde l kan høyst ha fyllkanter til l noder i hvert deltre som henger fra x i T. Det er et åpent spørsmål om et eliminasjonstre til et tre av høyde $\leq \lfloor log\, n \rfloor$ har mindre enn $\Theta(nlogn)$ fyllkanter.

Selv om $\frac{n}{2}-$algoritmen vil gi oss et lavt eliminasjonstre, så skal vi nå vise at det ikke gjelder generelt at $h(T)$ er innenfor en konstant av $h(T_{min})$. I figur 5.6 ser vi et tre G som har $h(T_{min}) = 2$. Dersom vi hadde brukt $\frac{n}{2}-$algoritmen ville vi fått et eliminasjonstre av høyde 4.



Figur 5.6 $\frac{n}{2}-$algoritmen gir et eliminasjonstre av høyde 4 til G, samtidig som $h(T_{min}) = 2$.

Legger vi nå til en node z til G som er nabo til x og som i sin tur har $3 \times 2^3 = 23$ naboer, så ville $\frac{n}{2}-$algoritmen gitt et eliminasjonstre av høyde 5. Generelt trenger grafen $3 \times 2^{n-1} - 1$ noder for at $\frac{n}{2}-$algoritmen skal gi et eliminasjonstre av høyde n for $n \geq 1$. Vi ser at gjennom å legge til $3 \times 2^{n-1}$ noder kan vi øke høyden $\frac{n}{2}-$algoritmen vil gi fra n til $n+1$. Samtidig vil høden til et minimalt eliminasjonstre forbli 2. Vi kan derfor finne en graf som $\frac{n}{2}-$algoritmen vil gi et eliminasjonstre til som er vilkårlig mye høyere enn $h(T_{min})$.

5.4 Analyse

Vi skal her gi en teoretisk øvre grense for kjøretiden til algoritmen Lag_Min fra seksjon 5.2.

Først skal vi se på kostnaden for å utføre Lag_Min. Vi setter $M(k)$ = kostnaden for å finne et minimalt eliminasjonstre til nodene i $T(v)$ dersom $h(v) = k$ og alle deltrær som henger fra v i T er minimale. Vi setter innledningsvis at det tar $O(1)$ tid hver gang en node må oppdatere sin haug med barn.

Dersom v bare har ett barn x av høyde $k-1$, foretar vi først en rotasjon med utgangspunkt i x. Det gir oss et eliminasjonstre av høyde høyst $k-1$ som vi må gjøre minimalt. I rotasjonen må vi lokalisere barnet til x som ligger mellom v og x i G. For å gjøre dette må vi høyst følge $k-1$ foreldrepekere, som hver tar konstant tid. Den totale kostnaden for en rotasjon blir derfor høyst $M(k-1) + k - 1$.

Har vi nå fremdeles ikke et minimalt eliminasjonstre, må vi begynne et binært søk på stien $v, ..., x$ i G etter den noden som skal elimineres sist. Vi må imidlertid først legge inn nodene på stien $v, ..., x$ i en tabell. I seksjon 5.1 så vi at det ikke kan være mer enn 2^{k-1} noder på denne stien. For å legge inn stien må vi først traversere stien en gang for deretter å reversere en sti som ikke er lenger enn 2^{k-2}. Den totale tiden dette tar, blir $O(3 * 2^{k-2})$.

Vi må nå høyst rotere opp $k-1$ noder fra tabellen slik at de blir rot i eliminasjonstreet. For at en node skal bli rot må vi høyst foreta k rotasjoner.

I rotasjon j, for $1 \leq j \leq k-1$ kan vi risikere å måtte minimalisere et eliminasjonstre av høyde høyst j til en kostnad av $M(j)$. Vi må også lokalisere og flytte over et deltre til en kostnad av høyst j. Vi vet at etter den siste rotasjonen som gjør z til rot, så er eliminasjonstreet vi må minimalisere ikke høyere enn $k-1$, og eliminasjonstreet vi må lokalisere og flytte over, er ikke høyere enn $k-2$. Det følger at kostnaden for denne rotasjonen ikke blir større enn $M(k-1)+k-1$. Dersom k er større enn 1 får vi følgende øvre grense på $M(k)$:

$$\begin{aligned} M(k) &\leq M(k-1)+k-1+3*2^{k-2}+(k-1)\left(M(k-1)+k-1+\sum_{i=1}^{k-1}(M(i)+i)\right) \\ &= (2k-1)M(k-1)+\frac{1}{2}k(k-1)^2+3*2^{k-2}+(k-1)\sum_{i=1}^{k-2}M(i) \end{aligned}$$

Samtidig gjelder $M(0) = M(1) = 1$.
Ved hjelp av et induksjonsbevis skal vi vise at $M(k) \leq (k+1)!2^{k+1}$.
Vi ser først at $M(1) = 1 \leq 8 = 2!2^2$. Vi antar nå at $M(j) \leq (j+1)!2^{j+1}$ for alle $j < k$. Vi skal vise at dette impliserer at $M(k) \leq (k+1)!2^{k+1}$. Vi ser først at

$$\begin{aligned} (k-1)\sum_{i=1}^{k-2}M(i) &\leq (k-1)!(k-1)2^{k-1}\sum_{i=1}^{k-2}\frac{(i+1)!}{(k-1)!2^{k-i-2}} \\ &\leq (k-1)!(k-1)2^k \end{aligned}$$

Det siste steget følger av

$$\sum_{i=1}^{k-2}\frac{(i+1)!}{(k-1)!2^{k-i-2}} \leq \sum_{i=0}^{\infty}\frac{1}{2^i} \leq 2$$

Dette gir oss

$$\begin{aligned} M(k) &\leq k!2^k(2k-1)+3*2^{k-2}+(k-1)!(k-1)2^k+\frac{1}{2}k(k-1)^2 \\ &\leq k!2^k(2k-1)+k!2^k+\frac{1}{2}k(k-1)^2 \\ &= k!2^{k+1}k+\frac{1}{2}k(k-1)^2 \\ &\leq (k+1)!2^{k+1} \end{aligned}$$

Induksjonsbeviset er ferdig, og vi ser at $M(k) \leq (k+1)!2^{k+1}$ gjelder for alle k.

Dersom vi bare hadde hatt sett på antall rotasjoner kan vi vise at antall rotasjoner vi må utføre ikke overstiger $k!2^{k+1}$.

Vi skal nå se på den totale kostnaden for å utføre Min_Tre. Det vil si den totale kostnaden for å finne et minimalt eliminasjonstre til et tre G. Anta at G har n noder og at $k = \lfloor log\, n \rfloor$. Da gjelder $2^k \leq n < 2^{k+1}$. Vi setter $P(n)$ = total kostnad for å finne et minimalt eliminasjonstre til G gitt at G er eliminert slik at vi starter med et eliminasjonstre T av høyde $\leq k$. Vi vet at vi alltid kan finne en slik eliminasjonsordning i tid $O\Big(n(log\, n)^2\Big)$.

For at en node skal ha høyde l i T, må den ha minst $2^l - 1$ noder under seg. Det må derfor eksistere mindre enn 2^{k-l+1} noder i T av høyde l. Det koster $M(l)$ å minimalisere $T(v)$ dersom $h(v) = l$ og alle deltrær som henger fra v er minimale. Dette gir oss:

$$\begin{aligned} P(n) &< \sum_{i=0}^{k} 2^{k-i+1} M(i) \\ &\leq 2^{k+2} \sum_{i=0}^{k} (i+1)! \\ &= (k+1)!2^{k+2} \sum_{i=0}^{k} \frac{(i+1)!}{(k+1)!} \end{aligned}$$

For $k \geq 1$ gjelder:

$$\sum_{i=0}^{k} \frac{(i+1)!}{(k+1)!} \leq \sum_{i=0}^{k} \frac{1}{2^i} \leq 2$$

Dette gir oss $P(n) < (k+1)!2^{k+3}$. Vi ser av dette at kjøretiden til hele algoritmen er begrenset av $O(nlogn(logn)!)$. Dette er imidlertid under forutsetning av at hver haug-operasjon tar tid $O(1)$. Med et tilsvarende argument som det vi brukte i seksjon 5.3, ser vi at når vi regner med tiden for å utføre haug-operasjonene, så blir kjøretiden begrenset av $O(n(logn)^2(logn)!)$.

Vi kan skrive om $O(n(logn)^2(logn)!)$ v.h.a. Stirlings approksimasjon. Dette er utført i appendiks B og gir at kjøretiden er begrenset av $O(n^{log(logn)})$. Dersom $O(n^{log(logn)})$ virkelig er den tetteste øvre grensen vi kan gi, så tilhører algoritmen klassen av algoritmer som er både sub-eksponentielle og super-polynomiske.

Da det ikke har lykkes å finne noen god nedre grense på kjøretiden, skal vi i seksjon 5.5 se på en del eksperimentelle resultater ved bruk av algoritmen.

Vi skal til slutt i denne seksjonen se på plassbehovet for å utføre algoritmen. Anta at vi hver gang vi kaller opp en funksjon med et eliminasjonstre som parameter, bare gir en peker til den øverste noden. Da trenger vi ikke mer enn en datastruktur for å lagre det deltreet som det til enhver tid arbeides med.

Nøstingen av kall til Lag_Min vil aldri gå lenger enn høyden til T som er begrenset av $\lfloor log\, n \rfloor$. I hvert kall setter vi av plass til en konstant mengde noder og en tabell med plass til n noder. Dette skulle gi at vi ikke trengte mer enn $O(nlogn)$ plass. Vi vet imidlertid at lengden av den stien vi må lagre når vi arbeider på et deltre av høyde l, ikke er større enn 2^{l-1}. Det følger at summen av lengdene av alle stier vi til enhver tid har lagret, aldri vil overstige n. Dersom vi bruker en global tabell hvor vi lagrer stiene sekvensielt, kan vi derfor klare oss med $O(n)$ plass for å utføre hele algoritmen.

5.5 Eksperimentelle resultater

Vi skal i denne seksjonen se på en del eksperimentelle resultater av bruk av Min_Tre. Grunnen til at vi gjør dette er todelt. For det første har det vist seg å være vanskelig å finne en god nedre grense for kjøretiden. For det andre må et tre som skal generere et tenkt verste tilfelle med kjøretid $\Omega(n^{log(log n)})$, ha så spesielle egenskaper at det er tvilsomt om det eksisterer noe slikt tre.

Fra analysen i seksjon 5.4 vet vi at det er antall rotasjoner som driver opp kostnaden. Antall rotasjoner vi må utføre vokser ikke raskere enn $O(\lfloor log\, n \rfloor !n)$. For at vi skal få et verste tilfelle når vi bare ser på antall rotasjoner, må følgende krav være oppfylte for hvert deltre T', som vi gir til Lag_Min:

(1) $h(T'_{min}) = \lfloor log\, n' \rfloor$.
(2) Avstanden i G mellom $v = rot(T')$ og det høyeste barnet x til v må være minst $2^{\lfloor log\, n' \rfloor - 1}$.
(3) Vi må rotere opp noder fra stien $x, ..., v$ i G slik at de blir rot, inntil vi finner to eliminasjonstrær med eliminasjonsordninger som tilfredstiller betingelsene i Lemma 5.2.

I testkjøringene sammenligner vi vekstraten til hvor mange rotasjoner vi utfører som en funksjon av antall noder, og hvordan $\lfloor log\, n \rfloor !n$ vokser. Algoritmen har blitt implementert i C og prøvekjørt på et utvalg av tilfeldig genererte grafer.

For å generere tilfeldige trær har vi brukt følgende algoritme:

Start med n noder og ingen kanter.
Gjenta $n - 1$ ganger:
  Velg en tilfeldig node x.
  Velg en tilfeldig node y blant nodene som ikke ligger i samme komponent som x.
  Sett x og y som naboer.

Denne måten å generere trær på gir ikke like stor sannsynlighet for hvert ikke-isomorft tre. Vi ser i figur 5.7 de to ikke-isomorfe trærne med 4 noder.

a **b**

Figur 5.7 Alle ikke-isomorfe trær med 4 noder.

Med fremgangsmåten som er gitt ovenfor vil vi generere (a) med sannsynlighet $\frac{13}{18}$ og (b) med sannsynlighet $\frac{5}{18}$. Selv om dette er en ulempe ved metoden vil vi likevel bruke den da det ikke har lykkes å finne en enkel metode for å generere tilfeldige trær som er bedre enn denne.

Resultatet av kjøringene presenteres i tabell 5.1. Det er utført 1000 kjøringer for hver verdi av n. Tabellen er satt opp som følger:

Kolonne 1 angir hvor mange noder det er i grafen.
Kolonne 2 gir det største antall rotasjoner algoritmen trengte.
Kolonne 3 gir gjennomsnittlige antall rotasjoner.
Kolonne 4 gir standard-avviket målt etter følgende formel:

$$\sigma = \sqrt{\frac{1}{999}\sum_{i=1}^{1000}(r_i - \mu)^2}$$

Her er r_i er antall rotasjoner i testkjøring i, og μ er det gjennomsnittlige antall rotasjoner.
Kolonne 5 gir verdien av $\lfloor log\, n \rfloor !n$.
Kolonne 6 gir verdien av $nlogn$.

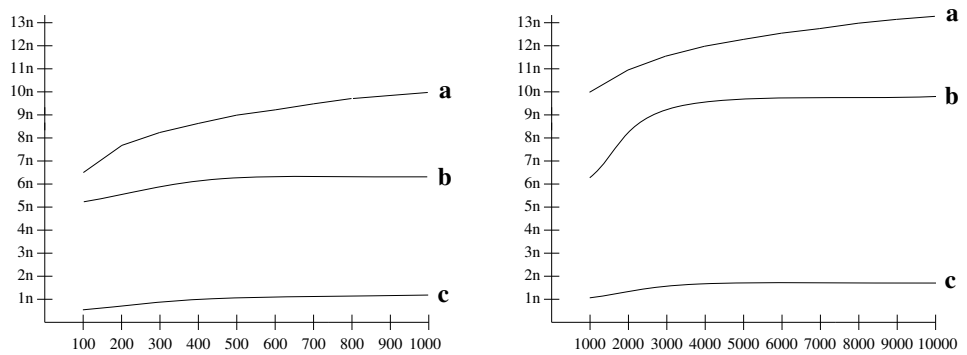
1	2	3	4	5	6
100	5.29n	0.54n	0.61n	720n	6.64n
200	4.35n	0.49n	0.40n	5040n	7.64n
300	3.69n	0.91n	0.47n	40320n	8.23n
400	3.19n	0.60n	0.30n	40320n	8.64n
500	6.24n	0.84n	0.81n	40320n	8.97n
600	4.37n	1.10n	0.73n	362880n	9.23n
700	3.32n	0.99n	0.46n	362880n	9.45n
800	2.68n	0.81n	0.32n	362880n	9.64n
900	2.56n	0.70n	0.27n	362880n	9.81n
1000	5.22n	0.73n	0.45n	362880n	9.97n
2000	3.60n	0.82n	0.34n	362880n	10.97n
3000	5.24n	1.55n	0.84n	3991800n	11.55n
4000	4.72n	1.05n	0.38n	3991800n	11.97n
5000	8.40n	1.05n	0.73n	479001600n	12.29n
6000	9.66n	1.69n	1.19n	479001600n	12.55n
7000	6.06n	1.70n	0.83n	479001600n	12.77n
8000	5.15n	1.43n	0.59n	479001600n	12.97n
9000	4.17n	1.15n	0.41n	6227020800n	13.14n
10000	3.81n	1.08n	0.34n	6227020800n	13.29n

Tabell 5.1 Eksperimentelle resultater fra prøvekjøring av Min_Tre algoritmen.

Som det fremgår fra tabellen trenger vi i verste tilfellet aldri utføre så mange som $nlogn$ rotasjoner. I gjennomsnitt trenger vi aldri utføre mer enn $2n$ rotasjoner. Dette indikerer at det burde være mulig å gi en lavere grense på kjøretiden enn $O\left(n^{log(log\, n)}\right)$.

Både det maksimale antallet rotasjoner og det gjennomsnittlige antall rotasjoner svinger. En av årsakene til dette kan være at ettersom n vokser så genererer vi en stadig mindre fraksjon av alle ikke-isomorfe trær med n noder.

I figur 5.8 har vi grafisk fremstilt hvordan det største antall og gjennomsnittlige antall rotasjoner forandrer seg med n. Vi har tegnet en glatt kurve som viser det høyeste antallet rotasjoner opp til hver verdi av n.



Figur 5.8 a. $nlogn$ b. Høyeste antall rotasjoner c. Gjennomsnittlig antall rotasjoner

Det fremgår av figuren at både gjennomsnittet og maksimal verdien vokser svakt med n, men likevel ikke så raskt som $nlogn$.

6 Konklusjon

I dette kapittelet skal vi gi en oversikt over resultatene fra denne hovedfagsoppgaven. Vi skal også se på en del problem som gjenstår å løse.

6.1 Sammendrag

Vi skal her gi et sammendrag av de mest sentrale resultatene fra kapittel 3 til 5. Vi så først på en del teoretiske egenskaper til minimale eliminasjonstrær. Disse brukte vi så for å utvikle nye algoritmer som reduserer høyden til et eliminasjonstre til en generell graf. Vi har også ut fra de teoretiske resultatene vist en algoritme som finner et minimalt eliminasjonstre til et tre.

Mer detaljert viste vi først i kapittel 3 at en delgraf alltid har et eliminasjonstre som er minst like lavt som et minimalt eliminasjonstre til den opprinnelige grafen. Ut fra dette resultatet kunne vi vise en ny nedre grense for $h(T_{min})$, som baserer seg på å finne delgrafer i G.

Vi så også nærmere på nøstet oppdelingsordninger, og kunne vise at det for enhver graf eksisterer en nøstet oppdelingsordning som gir et minimalt eliminasjonstre. Dette resultatet gjorde at vi i kapittel 4 kunne utvikle Minimale-kuttsett algoritmen for å redusere høyden til et eliminasjonstre.

Vi sammenlignet Minimale-kuttsett algoritmen og algoritmene til Liu og Hafsteinsson, og viste at ingen av algoritmene var strengt bedre enn noen av de andre. Vi kunne så vise at alle tre algoritmene tilhører en større klasse av algoritmer, som baserer seg på et felles resultat om hvordan man kan omordne eliminasjonstrær. Ut fra dette resultatet utviklet vi enda en ny algoritme som gir et minimalt eliminasjonstre til en linje-graf.

Til slutt presenterte vi i kapittel 5 algoritmen Lag_Min som finner et minimalt eliminasjonstre til et tre. Denne algoritmen bestod i å omordne eliminasjonstreet ved hjelp av lokale rotasjoner. For å styre algoritmen baserte vi oss både på resultatet om høyden til eliminasjonstrær til delgrafer og på resultater som bare gjelder for trær. I forbindelse med Lag_Min algoritmen utviklet vi også en algoritme som finner et eliminasjonstre av høyde $\leq \lfloor log\, n \rfloor$. Analysen av Lag_Min gav oss at vi ikke trengte å utføre mer enn $O(\lfloor log\, n \rfloor! n)$ rotasjoner. Da dette trolig er et overestimat gjennomførte vi en serie med testkjøringer og fant at antall rotasjoner vi måtte utføre aldri oversteg $nlogn$.

6.2 Åpne problem

Vi har i denne oppgaven sett på en del ulike sider ved minimale eliminasjonstrær. Det gjenstår mange interessante uløste problemer innenfor dette området. Vi skal her nevne noen.

Fra kapittel 5 gjenstår det å finne den virkelige kjøretiden til Lag_Min. Fra testkjøringene vet vi at denne trolig er polynomisk. Et interessant problem i tilknytting til Lag_Min er om det er mulig å utvikle effektive algoritmer, som finner minimale eliminasjonstrær til mer generelle grafer enn trær. For at dette skal være mulig må vi ha kriterier for ulike klasser av grafer, som gjør det mulig å avgjøre når et eliminasjonstre er minimalt.

Et relatert problem til å finne minimale eliminasjonstrær, er om man kan utvikle algoritmer som finner eliminasjonstrær som har høyde høyst en konstant fra minimum. For at det skal være mulig å se hvor høyt et eliminasjonstre er i forhold til $h(T_{min})$, må vi ut fra G kunne gi en tett nedre grense på $h(T_{min})$. Vi har allerede sett to måter for å finne slike grenser: Største klikk og utspennende trær. Det er trolig at ingen av disse virker særlig godt på generelle grafer, slik at det fremdeles er behov for andre måter å evaluere $h(T_{min})$.

Det finnes ulike metoder for å utvikle algoritmer som finner lave eliminasjonstrær. Vi har sett at en nøstet-oppdeling ordning finner et lavt eliminasjonstre direkte fra G. I kapittel 4 og 5 tok vi utgangspunkt i et eliminasjonstre og forsøker å gjøre det lavere. For å kunne bearbeide et allerede eksisterende eliminasjonstre trengs det omordningsregler slik som Teorem 4.2 og rotasjoner i eliminasjonstrær til trær. Det vil være interessant å se på flere slike regler for lokale omflyttinger i et eliminasjonstre, og å finne begrensninger for hvor langt de kan utnyttes.

Hensikten med å finne lave eliminasjonstrær er at de skal egne seg for parallell Cholesky-faktorisering. Det vil derfor også være naturlig å se på parallelle algoritmer for å finne lave eliminasjonstrær.

Det er ikke bare høyden til eliminasjonstreet som er avgjørende for gjennomføringstiden til parallell Cholesky-faktorisering. Antall fyllkanter virker også inn på kjøretiden. Det vil derfor være interessant å se om man kan finne algoritmer som gir lave (eventuelt minimale) eliminasjonstrær samtidig som de begrenser antall fyllkanter.

Et relatert problem er om man kunne vise hvor mange fyllkanter man får i et minimalt eliminasjonstre til ulike klasser av grafer. Dette tallet er sannsynligvis ikke så stort (i forhold til n), da det er vanskelig å finne små kuttsett i en graf dersom det er mange kanter i den.

Appendix A Komplette *k*-nære trær

Et komplett k-nært tre ($k \geq 2$), er et tre der hver node har høyst k barn og der noder blir lagt til treet i bredde-først ordning.
Vi kan maksimalt ha k^l noder av dybde l. Det følger derfor at vi maksimalt kan ha $\sum_{i=0}^{l-1} k^i = \frac{k^l-1}{k-1}$ noder i et komplett k-nært tre av høyde $l-1$.

Teorem A.1
Høyden til et komplett k-nært tre med n noder er gitt ved

$$\lfloor log_k(n(k-1)) \rfloor$$

Bevis:
Anta at vi har et komplett k-nært tre av høyde l. Vi ser på to tilfeller:

1. $l = 0$. Vi ser at vi maksimalt kan ha $\frac{k^1-1}{k-1} = 1$ node. Dette gir oss:

$$\lfloor log_k(n(k-1)) \rfloor = \lfloor log_k(k-1) \rfloor = 0$$

2. $l > 0$. Vi kan ha fra $\frac{k^l-1}{k-1} + 1$ til $\frac{k^{l+1}-1}{k-1}$ noder i et komplett k-nært tre av høyde l. Vi ser på grenseverdiene:

a) Dersom vi har $n = \frac{k^l-1}{k-1} + 1 = \frac{k^l+k-2}{k-1}$ noder, får vi:

$$\lfloor log_k(n(k-1)) \rfloor = \left\lfloor log_k\Big(k^l + k - 2\Big) \right\rfloor$$

Siden $k \geq 2$ og $l \geq 1$ så gjelder

$$k^l \leq k^l + k - 2 < k^{l+1}$$

Det gir oss:

$$\lfloor log_k(n(k-1)) \rfloor = l$$

b) Dersom vi har $n = \frac{k^{l+1}-1}{k-1}$ noder, får vi:

$$\lfloor log_k(n(k-1)) \rfloor = \left\lfloor log_k\left(k^{l+1}-1\right) \right\rfloor = l$$

Siden $log_k n$ er en strengt voksende funksjon av n, ser vi at $\lfloor log_k(n(k-1)) \rfloor = l$ gjelder for alle komplette k-nære trær av høyde l. $\square$

Vi skal nå se nærmere på uttrykket $\lfloor log_k(n(k-1)) \rfloor$. Vi vet at $0 \leq \lfloor log_k(k-1) \rfloor < 1$ gjelder for $k \geq 2$. Vi kan ut fra dette utlede følgende to ulikheter:

$$\lfloor log_k(n(k-1)) \rfloor = \lfloor log_k n + log_k(k-1) \rfloor \leq \lfloor log_k n \rfloor + 1 \tag{1}$$

$$\lfloor log_k n \rfloor \leq \lfloor log_k n + log_k(k-1) \rfloor = \lfloor log_k(n(k-1)) \rfloor \tag{2}$$

Fra ulikhetene (1) og (2) får vi følgende grenser på høyden til et komplett k-nært tre:

$$\lfloor log_k n \rfloor \leq \lfloor log_k(n(k-1)) \rfloor \leq \lfloor log_k n \rfloor + 1$$

Vi ser at høyden til et komplett k-nært tre er høyst en større enn $\lfloor log_k n \rfloor$.

Appendix B Stirlings approksimasjon

Vi skal i dette appendikset finne en øvre grense til uttrykket $n(logn)^2(logn)!$ ved hjelp av Stirlings approksimasjon.

Stirlings approksimasjon kan uttrykkes som:$n! \leq c\sqrt{n}\left(\frac{n}{e}\right)^n$, der c er en konstant. Knuth gir i [7] en utledning av Stirlings approksimasjon.

Vi skal nå se hvordan vi kan bruke dette til å finne en øvre grense for $(logn)!$. Vi får først:

$$(log\, n)! \leq c\sqrt{log\, n}\left(\frac{log\, n}{e}\right)^{log\, n}$$

Vi kan skrive $\left(\frac{log\, n}{e}\right)^{log\, n}$ videre om som følger.

$$\begin{aligned}\left(\frac{log\, n}{e}\right)^{log\, n} &= \left(e^{ln(log\, n)-1}\right)^{log\, n} \\ &= e^{(log\, n)(ln(log\, n)-1)} \\ &= n^{log(log\, n)-\frac{1}{ln2}}\end{aligned}$$

Dette gjør at vi nå kan gi følgende øvre grense for vårt uttrykk:

$$n(log\ n)^2(log\, n)! \leq \frac{c(log\, n)^{\frac{5}{2}} n^{log(log\, n)}}{n^{\frac{1-ln2}{ln2}}}$$

Vi skal se videre på brøken $(log\, n)^{\frac{5}{2}}/n^{\frac{1-ln2}{ln2}}$ og vise at for alle $n > 1$ så blir den aldri større enn 16. For å gjøre dette ser vi på den kontinuerlige funksjonen

$$f(x) = \frac{(log\, x)^{\frac{5}{2}}}{x^{\frac{1-ln2}{ln2}}}$$

Funksjonen $f(x)$ er definert for $x \geq 1$. Vi deriverer $f(x)$ og får

$$f'(x) = \frac{(ln\, x)^{\frac{3}{2}}\left(\frac{5}{2} - \left(\frac{1-ln2}{ln2}\right)ln\, x\right)}{(ln2)^{\frac{5}{2}} x^{\frac{1}{ln2}}}$$

Setter vi $f'(x) = 0$ får vi $x = 1$ og $x = e^{\frac{5ln2}{2-2ln2}} \approx 283.50$. Siden $f'(x) > 0$ for $x \in <1, e^{\frac{5ln2}{2-2ln2}}>$ og $f'(x) < 0$ for $x > e^{\frac{5ln2}{2-2ln2}}$ ser vi at $x = e^{\frac{5ln2}{2-2ln2}}$ gir et

absolutt maksimalpunkt for $f(x)$. Siden $f(283) \approx 15.55$ og $f(284) \approx 15.55$ ser vi at $\frac{(log\, n)^{\frac{5}{2}}}{n^{\frac{1-ln2}{ln2}}} < 16$ for alle heltalls verdier av $n \geq 1$. Det følger at

$$n(log\, n)^2(log\ n)! \leq c_1 n^{log(log\, n)}$$

der c_1 er en konstant.

Bibliografi