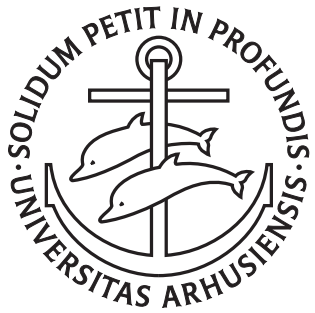
Approksimative afstande i planare grafer

Louise Skouboe Bjerg og Lone Asferg Laursen

Speciale



Datalogisk Institut
Aarhus Universitet
Danmark

Abstract

This thesis consists of two parts, each describing the construction of a data structure on a planar graph. Both data structures are described in the article *Compact Oracles for Reachability and Approximate Distances in Planar Digraphs* by Mikkel Thorup.

The first part of the thesis deals with the construction of a reachability oracle, which is a compact data structure, used to answer questions of reachability in planar oriented graphs. We give a thorough theoretical walk-through of the construction of a reachability oracle, and how queries are executed in the oracle. Furthermore we have analysed the resource usage of the oracle. The thesis contains an implementation of a reachability oracle, and the first part includes a description and practical evaluation of this implementation.

The second part of the thesis deals with the construction of a distance oracle, which given a planar oriented graph with non-negative integer valued weights on the edges can return an approximation of the shortest distance between two nodes in the graph. In this part we give a detailed theoretical view on how to construct a distance oracle, and how to use it to find approximate distances. We have analysed the resource usage for a distance oracle. In this part is furthermore a description of our implementation of a distance oracle and an evaluation of its practical use.

The main contributions of this thesis is a clarification of the theory behind the algorithms and data structures and filling in nontrivial details which in the article above has been left to the reader. Furthermore we demonstrate that the above mentioned data structures are implementable.

Resume

Dette speciale af sammensat af to dele, der hver omhandler konstruktionen af en datastruktur, der er konstrueret med udgangspunkt i en planar graf. Begge datastrukturer er beskrevet i artiklen *Compact Oracles for Reachability and Approximate Distances in Planar Digraphs* af Mikkel Thorup.

Den første del af specialet omhandler konstruktionen af et reachability-orakel, som er en kompakt datastruktur, der kan anvendes til at svare på spørgsmål om reachability i planare, orienterede grafer. Vi giver en grundig teoretisk gennemgang af konstruktionen af et reachability-orakel og af, hvordan forespørgsler udføres i oraklet. Vi har ligeledes analyseret ressourceforbruget for oraklet. Specialet indeholder en implementation af et reachability-orakel og første del omfatter en beskrivelse af implementationen samt en praktisk evaluering af denne.

Den anden del af specialet omhandler konstruktionen af et afstandsorakel, der for planare, orienterede grafer med ikke-negative heltallige vægte på kanterne kan give en approksimering af den korteste afstand mellem to knuder i grafen. Vi giver i denne del en detaljeret teoretisk gennemgang af, hvorledes et afstandsorakel konstrueres og anvendes til at finde approksimative afstande. Vi har analyseret ressourceforbruget for et afstandsorakel. Vi har ligeledes implementeret afstandoraklet, og denne implementation samt en praktisk evaluering beskrives i denne del.

Hovedbidraget i dette speciale er en tydeliggørelse af teorien bag algoritmer og datastrukturer, udfyldelse af ikke-trivielle detaljer, der i ovennævnte artikel er overladt til læseren samt en eftervisning af, at de nævnte datastrukturer kan implementeres.

Tak til

Færdiggørelsen af dette speciale havde ikke været mulig uden hjælp og støtte fra en del mennesker. Disse mennesker vil vi gerne takke.

Tak til vores vejleder Gerth Stølting Brodal for hjælp, en stor mængde tålmodighed og for altid at tage sig tid til at diskutere både stort og småt.

Vi vil gerne takke vores respektive mand/kæreste Mikkel Bjerg og Jan Grinderslev Nielsen for uanede mængder af positiv energi, psykisk opbakning og konstruktiv hjælp – både fagligt og praktisk.

Tak til Bjarke Skjernaa, Alex Rune Berg og Kåre Fiedler Christiansen for utallige udbytterige diskussioner, gode råd og ideer.

Vi vil gerne takke Kåre Fiedler Christiansen og Niels Ole Jensen for at være LaTeX-guruer og for altid at have tid til at svare på tekniske spørgsmål.

Tak til Jonas Martin Thomsen og Sabrina Vestergaard Nielsen for korrekturlæsning og gode ideer. Vi vedkender os det fulde ansvar for tilbageværende fejl.

Tak til frokostklubben bestående af: Niels 'Hotdog' Jensen, Kåre Fiedler Christiansen, Thomas Brochmann Pedersen, Kristoffer Arnsfelt Hansen, Bolette Ammitzbøll Madsen, Bjarke Skjernaa, Jesper Makholm Byskov, Mads Johan Jurik, Aske Simon Christensen, Trine Kornum Christiansen og Alex Rune Berg. De har alle bidraget med gode ideer og gjort de sidste år på universitetet hyggeligere.

Sidst, men ikke mindst, vil vi gerne takke vores familier og venner for støtte, opbakning, opmuntring og praktisk hjælp.

Lone Asferg Laursen og Louise Skouboe Bjerg,
Århus, 2. januar 2004.

Indhold

Figurer

Kapitel 1

Introduktion

Formålet med dette speciale har været at give en tilgængelig beskrivelse af de to datastrukturer, der er beskrevet i artiklen *Compact Oracles for Reachability and Approximate Distances in Planar Digraphs* [Tho01] af Mikkel Thorup. Begge datastrukturer tager udgangspunkt i planare, orienterede grafer og beskæftiger sig med hhv. reachability og approksimative afstande mellem knuder i graferne. Vi har gennem tilføjelse af illustrationer, motivationer og detaljer ønsket at give overblik over og forståelse for de forskellige algoritmer og datastrukturer, der er anvendt. Vi har ligeledes ønsket at vise, at datastrukturerne kan implementeres samt at vise, hvornår de i praksis giver en bedre udførelse end gængse algoritmer som bredde først-søgning og Dijkstras algoritme.

I artiklen [Tho01] beskrives som nævnt to datastrukturer. Den første datastruktur er et *reachability oracle* (herefter benævnt reachability-orakel). Dette kan anvendes til at afgøre, om der eksisterer en vej fra en knude til en anden i en orienteret, planar graf. Den anden datastruktur er et *distance oracle* (herefter benævnt afstandsorakel). Afstandsoraklet anvendes til at give et estimat af længden af den korteste vej fra en knude til en anden. Vores speciale er delt op i to dele hvor hvert orakel behandles for sig. Inden de to dele introducerer vi kort C++-biblioteket LEDA, som vi har anvendt under vores implementation.

Givet en planar, orienteret graf G beskriver Mikkel Thorup i artiklen [Tho01], hvorledes et reachability-orakel kan konstrueres i tid $\mathcal{O}(n \log(n))$, således at det fylder $\mathcal{O}(n \log(n))$. Reachability-oraklet kan i tid $\mathcal{O}(\log(n))$ afgøre, om der eksisterer en vej mellem to knuder i G. I artiklen gives en kompakt beskrivelse af de algoritmer og datastrukturer, der skal bruges for at opbygge et reachability-orakel. Der gives først en beskrivelse af hvorledes et reachability-orakel, der kan besvare forespørgsler i $\mathcal{O}(\log(n))$ konstrueres. Herefter følger en forklaring på hvorledes der kan ændres i konstruktionen således at forespørgsler kan besvares i konstant tid.

Vi gennemgår i første del af dette speciale hvordan et reachability-orakel konstrueres. I denne gennemgang har vi tilføjet en del detaljer, der i den omtalte artikel er overladt til læseren. Vi har ligeledes tilføjet en række illustrationer samt motivationer for indførelsen af de anvendte konstruktioner og algoritmer. Vi har implementeret et reachability-orakel og derved opnået en større forståelse af de principper der beskrives i teorien. Efter den teoretiske gennemgang af hver del af konstruktionen af reachability-oraklet, beskriver vi hvorledes vi har valgt

at implementere de beskrevne strukturer og algoritmer.

Vi sandsynliggør gennem en omfattende mængde af test, at vores implementerede reachability-orakel overholder de beviste asymptotiske grænser for forbruget af tid og plads. Vi har endvidere undersøgt, hvorledes ressourceforbruget ændres når reachability-oraklet konstrueres på baggrund af forskellige typer af grafer og vi har givet en teoretisk forklaring på disse ændringer. Allerede ved vores mindste testgrafer med 100 knuder kan vi se, at udførelsestiden for forespørgsler er bedre end en bredde først-søgning.

Vi giver en teoretisk gennemgang af, hvorledes der kan laves ændringer i konstruktionen af reachability-oraklet, således at forespørgsler kan besvares i konstant tid. Vi beskriver ligeledes ændringer i konstruktionen af oraklet, der gør, at oraklet kan anvendes til at finde en vej mellem to knuder. Vi giver en gennemgang af, hvorledes det er muligt at distribuere den reachability-information, der er beregnet, på grafens knuder.

Mikkel Thorup giver i anden del af artiklen [Tho01] en beskrivelse af, hvorledes et afstandsorakel konstrueres. Givet en planar, orienteret graf med ikkenegative, heltallige vægte på kanterne forklares, hvorledes et afstandsorakel kan konstrueres. Et afstandsorakel kan for to knuder i grafen beregne en approksimativ afstand, der er konservativ og ligger inden for en faktor $(1 + \varepsilon)$ fra den korteste afstand mellem knuderne. I artiklen gives en beskrivelse af de basale konstruktioner, der er nødvendige for at lave et afstandsorakel. Derefter gives en kompakt beskrivelse af hvorledes pladsforbruget og tiden for at besvare en forespørgsel kan optimeres.

Vi giver i anden del af specialet en detaljeret og illustreret gennemgang af teorien bag konstruktionen af oraklet og vi redegør for, at et afstandsorakel kan konstrueres i tid $\mathcal{O}(n \log^3(n) \log(nW)/\varepsilon)$, hvor W er vægten af den tungeste kant. Oraklet fylder $\mathcal{O}(n \log(n) \log(nW)/\varepsilon)$ og kan besvare forespørgsler i tid $\mathcal{O}(\log(\log(nW)) \log(n) + \log(n)/\varepsilon)$. Vi har tilføjet motivationer for de mange konstruktioner, der indføres, for at skabe et overblik over sammenhængen af oraklet. Vi har igen implementeret det beskrevne orakel. Efter en teoretisk gennemgang af hver del beskriver vi, hvorledes vi har valgt at implementere de omtalte konstruktioner. Vi giver en kortfattet opsummering af hvilke forbedringer, der efterfølgende kan laves.

Vi har lavet en stor mængde test af vores implementation og vi konkluderer bl.a., at de approksimative afstande, der beregnes er langt bedre end det lovede og at en stor del af forespørgslerne besvares med den korteste afstand. Vi konkluderer endvidere, at for at oraklet skal være praktisk anvendeligt kræves, at der optimeres væsentligt på forbruget af plads.

Der introduceres i dette speciale en lang række begreber i forbindelse med gennemgangen af teorien om oraklerne og vi har motiveret nødvendigheden af disse. Vi har lagt stor vægt på at give en bedre forståelse for konstruktioner og definitioner ved intensiv brug af illustrationer. Endvidere har vi udfyldt mange detaljer i beviser, hvor der i artiklen har været uklarheder. Vi har ligeledes lavet bidrag i form af mindre ændringer i konstruktionen og tilføjelse af beviser. Artiklen er skrevet særdeles kompakt og mange detaljer er overladt til læseren. Vores hovedbidrag har været at tydeliggøre teorien, at udfylde de til læseren overladte ikke-trivielle detaljer samt at vise, at implementation af et

reachability-orakel og et afstandsorakel er mulig.

Vores implementation er vedlagt på cd. På cd'en findes kildekode, mindre testprogrammer, make-filer samt en kompileret udgave af programmerne. En vejledning kan findes i bilag C. På cd'en findes ligeledes dette speciale som hhv. postscript og pdf.

Kapitel 2

LEDA

LEDA (Library of Efficient Darastructures and Algorithms) er et C++-bibliotek, som indeholder datastrukturer og algoritmer, der blandt andet er velegnede at bruge, når man arbejder med grafer. Vi vil i det følgende beskrive den del af LEDA vi har anvendt. Først gives en beskrivelse af de datastrukturer vi har anvendt. Vi vil kort omtale køer, lister og arrays. Derefter vil vi i detaljer beskrive LEDAs parametriserede grafer som vi intensivt har benyttet. Endelig vil vi beskrive de af LEDAs algoritmer vi har brugt og hvor effektive de er. LEDA har indbygget hukommelseshåndtering og vi vil sidst i dette kapitel beskrive hvordan den virker.

2.1 Anvendte datastrukturer fra LEDA

I LEDA er implementeret lister, arrays og køer. Alle datastrukturerne er parametriserede og effektivt implementeret. De indeholder alle metoder (og lidt til) som man kan forestille sig at have brug for. Deriblandt er tilgangs- og opdateringsmetoder og forskellige metoder til at søge og sortere i elementerne. Vi vil kort beskrive de datastrukturer vi har brugt i implementationen af reachability- og distance-oraklet.

2.1.1 Lister, arrays og køer

En liste er i LEDA implementeret ved brug af en dobbeltkædet liste [MN99], side 70. Dette giver de forventede udførelsestider på alle tilgangs- og opdateringsmetoder. Det tager således lineær tid at finde det i'te element i listen, mens vi i konstant tid kan fjerne et element på et givet sted i listen. Givet en liste `l` med elementer af af type `value_type` kan elementerne i listen gennemløbes ved at benytte funktionen `forall_items(value_type x, list l)` der successivt tildeler `x` værdien af indholdet af elementerne i `l`.

Et array er realiseret ved brug af C++ vektorer [MN99], side 76. Dette giver mulighed for i konstant tid at tilgå et element ved brug af dets indeks.

Prioritetskøer er implementeret ved brug af en Fibonacci-heap [MN99] side 149. I en Fibonacci-heap kan man indsætte et element og fjerne det mindste element i logaritmisk tid i størrelsen af heapen.

2.1.2 Parametriserede grafer

Alle vores grafer er implementeret ved hjælp af LEDAs parametriserede grafer. Denne datatype giver os mulighed for at knytte vilkårlig ekstra information til knuder, kanter og flader. Grafen er repræsenteret med kantlister, dvs. for hver knude har vi en liste med udkanter og en med indkanter. Det er det muligt at tilknytte yderligere information til knuder og kanter. Givet en parametriseret graf kan man få et `node_array`, der kan indeholde information, der kan tilknyttes knuder. Givet en knude i grafen kan man slå op `node_array`et og få den information, der er tilknyttet knuden. Det er på samme måde muligt at få et `edge_array`. Disse arrays er meget nyttige, da de giver os mulighed for nemt at tilknytte temporær information til knuder hhv. kanter i en graf.

LEDAs graftype giver mulighed for en bred vifte af forespørgsler, som alle tager konstant tid. For en kant kan man bl.a. få dens endepunkter og fladen til venstre for kanten i forhold til kantens orientering. Man kan få iteratorer over næsten enhver ønskelig mængde f.eks. alle kanter, knuder og flader eller ud-/indkanter i forhold til en knude.

I LEDA er der metoder til at generere maksimale planare grafer og planare grafer med et ønsket antal knuder og kanter. Vi vil i forbindelse med gennemgangen af vores test i kapitel 10 beskrive hvorledes vi har benyttet LEDAs genererede grafer.

2.2 Anvendte LEDA-algoritmer

Der er i LEDA implementeret mange geometriske algoritmer. Vil vil i dette afsnit kort beskrive de algoritmer vi har anvendt.

Vi bruger algoritmen `make_bidirected()`, der laver en graf tovejsorienteret, dvs. der tilføjes ekstra kanter i grafen således, at der for enhver kant eksisterer en anden kant med samme endepunkter, men med modsat orientering. Algoritmens udførelsestid er lineær i størrelsen af grafen [MN99], side 277.

En planar graf bliver indlejret i planen ved at kalde af LEDA-funktionen `make_planar_map()`. Den omarrangerer kantlisterne i knuderne således at rækkefølgen repræsenterer en plan indlejring. Kanternes rækkefølge er i knudernes lister af kanter angivet mod urets orientering. Udførelsestid er lineær i størrelsen af grafen [Alg], side 198 og 202d.

Vi kalder funktionen `triangulate_planar_map()` for at triangulere vores grafer. Funktionen bevirker, at der indsættes kanter i grafen således at enhver flade omkranses af tre kanter. Denne algoritme udføres ligeledes i lineær tid i størrelsen af grafen [MN99], side 566.

Det er i LEDA muligt at teste om en graf har bestemte egenskaber. Vi tester om en graf er sammenhængende og om den er plan, disse tjek tager lineær tid.

2.3 LEDAs Hukommelseshåndtering

Normalt allokeres plads i C++ ved brug af `new`-operatoren, der typisk kalder c-funktionen `malloc` og plads deallokeres ved brug af `delete`, der oftest kalder

c-funktionen `free`. Kald til `malloc` og `free` er dyre kald, hvilket kan give et program et stort overhead hvis der ofte allokeres og deallokeres små stykker af hukommelsen. LEDAs indbyggede hukommelseshåndtering forsøger at amortisere de dyre systemkald til `malloc` over en lang sekvens af kald til `new`, hvor der i hvert kald allokeres en mindre mængde hukommelse (mindre end 256 bytes). LEDA allokerer altid større blokke af hukommelse (3-255 kB), der skæres i mindre lige store stykker og håndteres i en kædet liste. Disse lister indsættes i et array af størrelse 256 – en indgang for hver størrelse. For hver liste eksisterer der en friliste, som holder styr på hvilke blokke, der er fri i en liste. Ved kald af `new`, hvor der skal allokeres n bytes, undersøges først om der eksisterer en liste på den n'te plads i arrayet. Hvis ikke, allokeres en blok, der skæres i mindre stykker. Disse stykker er ikke nødvendigvis n store. Eksisterer der en liste, gennemses den tilknyttede friliste og kun hvis der ikke i forvejen er en fri blok, udføres et kald til `malloc`. Kald til `delete`, hvor det allokerede bliver håndteret af LEDAs hukommelseshåndtering udføres ved, at den pågældende blok markeres som fri og derfor senere kan genbruges [Thi98]

Kald til `new` hhv. `delete` hvor der skal allokeres hhv. deallokeres mere end 256 bytes håndteres på normal vis.

En bemærkelsesværdig ting ved LEDAs hukommelseshåndtering er, at den aldrig deallokerer automatisk [led]. Det er dog muligt at tvinge LEDA til at frigive en del af den allokerede hukommelse. Ved at kalde `clear()` gennemløbes alle lister, men kun hvis **alle** blokkene i en liste er frie deallokeres hukommelsen [Thi98].

Det er muligt at vælge hvorvidt man vil have LEDA til at håndtere hukommelse, der allokeres i forbindelse med egne klasser. Vi har valgt, at få LEDA til at håndtere hukommelsen for hele vores program og det er der to grunde til. For det første viste nogle simple test at det tidsmæssigt var mere effektivt og for det andet øgede det ikke, som man måske kunne frygte, forbruget af hukommelse.

Del I

Reachability

Kapitel 3

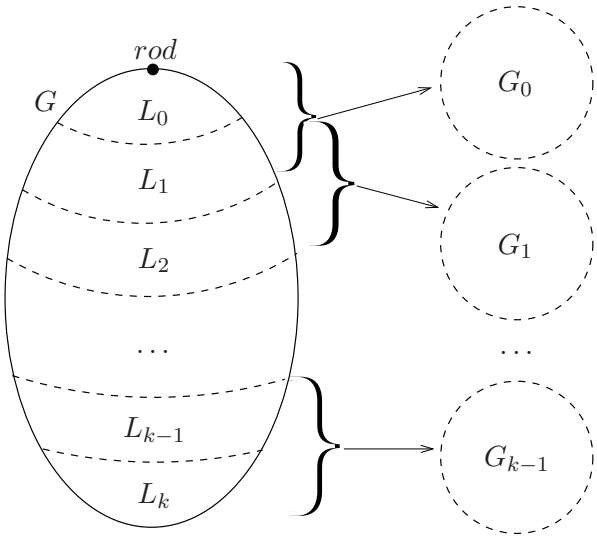
Et kompakt rechability-orakel

> *Givet en orienteret, planar og sammenhængende graf G med n knuder og $\mathcal{O}(n)$ kanter ønsker vi at konstruere en datastruktur, der kan anvendes til at besvare forespørgsler om reachability i G. Datastrukturen ønsker vi at konstruere i tid $\mathcal{O}(n \log(n))$ samt at pladsforbruget ligeledes er $\mathcal{O}(n \log(n))$. Forespørgsler ønskes foretaget i tid $\mathcal{O}(\log(n))$. Vi kalder datastrukturen et reachability-orakel.*

Vi ser på orienterede, planare, sammenhængende grafer. For en sådan graf G ønsker vi at konstruere en datastruktur, som kan anvendes til at besvare forespørgsler om reachability, dvs. svare på om der eksisterer en vej fra en knude v til en knude w i G. Datastrukturen skal konstrueres så den er kompakt – pladsforbruget skal være mindre end $\mathcal{O}(n^2)$, som er pladsforbruget for det trivielle reachability-orakel: en tabel, hvori resultatet for alle par af knuder findes. Den ønskede konstruktionstid for datastrukturen er $\mathcal{O}(n \log(n))$. Vores datastruktur skal i tid $\mathcal{O}(\log(n))$ kunne svare på, om en knude v kan nå en anden knude w. Vi beskriver i denne del, hvorledes et orakel, der opfylder disse krav kan konstrueres. Fra et teoretisk synspunkt beskriver vi de forskellige faser i konstruktionen – hvilke algoritmer og datastrukturer der anvendes undervejs, samt hvorledes den færdige konstruktion anvendes til at besvare reachability-spørgsmål. Vi beskriver desuden, hvorledes vi har implementeret et reachability-orakel – hvordan vi har udfyldt de manglende detaljer i de beskrevne algoritmer.

I dette kapitel introducerer vi kort de forskellige dele af reachability-oraklet samt de anvendte teknikker. I de følgende afsnit vil vi kort introducere hver de forskellige faser i algoritmen, der opbygger et reachability-orakel. Vi vil skitsere hvorledes datastrukturen opbygges og hvordan den kan bruges til at besvare forespørgsler angående reachability. Endvidere vil vi kortfattet beskrive hvorledes vi eksperimentelt har evalueret den implementerede datastruktur. Vi afslutter dette kapitel med kort at introducere et antal mulige udvidelser i forhold til hvad vi har implementeret.

De begreber og konstruktioner, som vi introducerer i dette kapitel, vil vi i de følgende kapitler (4-11) beskrive i detaljer. Vi vil i disse kapitler også beskrive vores implementation. Vi vil desuden komme ind på hvilke detaljer, ændringer og tilføjelser vi har bidraget med.



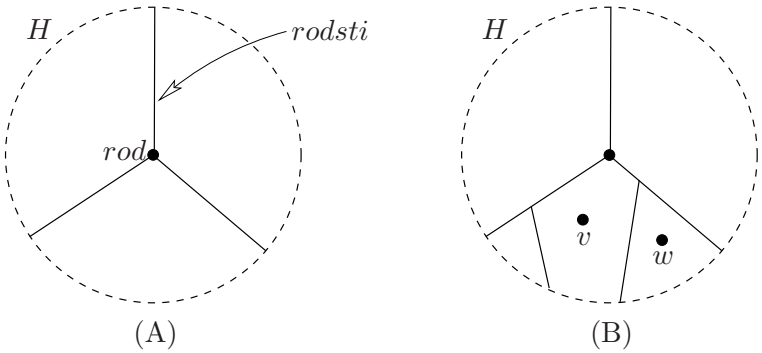
Figur 3.1: Grafen G opdeles i lag og to på hinanden følgende lag bliver til en tolagsgraf.

3.1 Reducering af problemet

Antag, at vi er givet en planar, orienteret og sammenhængende graf G. Den første fase i konstruktionen af vores reachability-orakel er at konstruere et antal mindre og simplere grafer med udgangspunkt i G og dermed at reducere problemet. Den oprindelige graf G er en vilkårlig planar, orienteret graf, hvor de nye grafer er simplere idet de indeholder et udspændende træ, hvor alle rodstier (dvs. sti mellem rod og knude) højst er sammensat af to orienterede dele. Disse grafer kalder vi tolagsgrafer, da der set fra roden er to orienterede lag i grafen. Vi konstruerer tolagsgraferne ved at opdele G i lag med udgangspunkt i en vilkårligt valgt rod.

Hver gang vi i G har to på hinanden følgende lag, konstruerer vi en tolagsgraf, der indeholder knuderne i disse to lag, se figur 3.1. Vi kalder lagene $L_0, L_1, \ldots, L_k$ og tolagsgraferne kaldes $G_0, G_1, \ldots, G_{k-1}$.

Hver knude i den oprindelige graf vil eksistere i op til to af de nye grafer, hvorfor en query kan besvares ved blot at se på op til to af disse simplere grafer. Vores orakel skal derfor sammensættes af et del-orakel for hver af disse mindre grafer. Hvorledes den oprindelige graf opdeles i tolagsgrafer er beskrevet i kapitel 4. Her beskrives endvidere de ændringer vi har lavet i forhold til det oprindelige konstruktionsforslag [Tho01]. Som et eksempel på en ændring kan det nævnes, at vi indsætter en ekstra knude som rod i den første graf G_0, hvilket betyder, at vi undgår specialtilfælde. Denne ændring beskriver vi i afsnit 4.2.1. Vi beviser at størrelsen af tolagsgraferne er lineær i størrelsen af den oprindelige graf og at de kan konstrueres i tid $\mathcal{O}(n \log(n)$. Vi beviser ligeledes, at et reachability-spørgsmål til den oprindelige graf kan oversættes til reachability-spørgsmål til op til to af disse tolagsgrafer.



Figur 3.2: (A) En tolagsgraf H opdeles i dele vha. rodstier, der udgør en separator. (B) I H adskilles v fra w af et antal rodstier – dermed må en vej fra v til w krydse en af disse.

3.2 Dekomposition af tolagsgrafer

Vi har nu reduceret problemet til tolagsgrafer og skal kunne lave orakler, der kan besvare reachability-spørgsmål for sådanne grafer. Hver tolagsgraf behandles for sig.

Hovedideen i konstruktionen af et orakel for en tolagsgraf er at lave en hierarkisk dekomposition tolagsgrafen. Givet en tolagsgraf G_i deler vi denne op i et antal mindre tolagsgrafer vha. en *separator*. En separator er en samling af stier, i det udspændende træ, der har roden som det ene endepunkt. Disse stier kaldes *rodstier*. Bemærk at rodstier kan være sammensat af to orienterede stier. En delgraf H af G_i er ligeledes en tolagsgraf. Vi deler igen de mindre tolagsgrafer op vha. af separatorer og fortsætter således rekursivt indtil der ikke er flere knuder tilbage i delgraferne. På figur 3.2 (A) ser vi hvordan en tolagsgraf H opdeles i tre dele vha. rodstier.

Formålet med opdelingen vha. af separatorer er at identificere de mulige måder en knude v kan nå en anden knude w på. Når vi dekomponerer grafen, vil vi på et tidspunkt dele v fra w. Hvis der eksisterer en vej fra v til w må denne derfor krydse en af de rodstier, der deler grafen op således at v og w ikke findes i samme del, se figur 3.2 (B). I en separator identificerer vi de orienterede stier. For hver orienteret sti Q af de stier, der opdeler grafen i vores rekursive dekomposition gemmer vi derfor information, der kan anvendes til at afgøre, om der findes en sti fra en given knude v til en given knude w, der krydser Q. Hvilken information vi gemmer og hvorledes denne beregnes er beskrevet i kapitel 8.

Et skridt i dekompositionen foregår ved, at vi triangulerer grafen og finder en passende trekantet flade med tre knuder. Stierne fra disse tre knuder til roden udgør da den separator, der deler grafen op i tre dele, som vist på figur 3.2 (A). Algoritmen for hvorledes vi finder en flade, der opdeler grafen er skitseret i [LT79] og i kapitel 6 beskriver vi denne algoritme samt hvorledes vi har udfyldt de manglende detaljer. I [LT79] er algoritmen for hvorledes en trekant findes overfladisk beskrevet. Det er illustreret, hvordan et enkelt problem, der opstår i forbindelse med udførelsen af algoritmen, skal løses, men en gene-

rel løsning er overladt til læseren. Vi har derfor bidraget med en forklaring af, hvorledes algoritmen kan implementeres, så en passende flade altid kan findes.

I kapitel 5 beskriver vi hvorledes vi finder separatoren og de orienterede stier, som denne indeholder. Selve opdelingen af grafen i delgrafer foregår ved at separatoren sammentrækkes til én knude, hvilket deler grafen i tre dele. Opdelingen beskrives i kapitel 7. Når et skridt i dekompositionen er lavet, kaldes rekursivt på de dele, som vi har delt grafen op i.

Et skridt i dekompositionen kan foretages i lineær tid i størrelsen af grafen, der opdeles. Vi argumenterer for, at hver af de netop nævnte faser kan foretages i lineær tid.

Efter en teoretisk gennemgang af hver fase i dekompositionen, vil vi beskrive hvorledes hver fase er implementeret, og vi vil her komme ind på hvilke udfordringer vi har mødt undervejs og hvilke detaljer vi selv har tilføjet.

3.3 Det samlede orakel

Når vi foretager den rekursive dekomposition af en tolagsgraf G_i opbygger vi undervejs et rekursionstræ. En knude i dette svarer til et kald i den rekursive opdeling og knuden indeholder den information, der er beregnet i dette kald. Dette rekursionstræ er det bidrag til den samlede datastruktur, der svarer til G_i. Vi introducerer rekursionstræer og deres indhold i kapitel 5.

For at lave det samlede orakel, skal vi have et rekursionstræ for hver tolagsgraf. Det samlede orakel kommer dermed til at bestå af et array af rekursionstræer. Hvorledes datastrukturen samles er beskrevet i kapitel 9, hvor en sammenfatning af dekompositionen også findes.

Vi argumenterer i samme kapitel for, at det samlede orakel kan konstrueres i tid $\mathcal{O}(n \log(n))$ og at pladsforbruget for oraklet ligeledes er $\mathcal{O}(n \log(n))$.

3.4 Forespørgsler

Når datastrukturen er konstrueret, kan der laves forespørgsler på, om hvorvidt en knude v kan nå en knude w i grafen G. Første fase i en forespørgsel er at afgøre hvilke tolagsgrafer v og w findes i – for at der kan eksistere en vej fra v til w, må knuderne findes i samme tolagsgraf. Der kan være to sådanne grafer, da knuderne v og w hver højst befinder sig i to tolagsgrafer. Hvis en sådan graf G_i findes, undersøger vi for alle de separatorstier, der tilsammen adskiller v fra w i G_i, om der findes en sti fra v til w, der krydser denne sti. Der vil være et logaritmisk antal stier, der skal undersøges. Hvis der er to fælles grafer for v og w udføres søgningen gennem separatorstierne for begge disse grafer. Forespørgsler er beskrevet i kapitel 9. Vi argumenterer her for, at en forespørgsel kan foretages i tid $\mathcal{O}(\log(n))$.

3.5 Eksperimentel evaluering

Den datastruktur som vi har beskrevet i kapitlerne 4 til 9 har vi implementeret. I kapitel 10 beskriver vi de tests vi har lavet. Vi har lavet eksperimenter,

som underbygger de asymptotiske grænser på tidsforbrug for konstruktionen, pladsforbrug for konstruktionen samt tidsforbruget for forespørgsler. Vores testresultater underbygger vores forventninger om de asymptotiske grænser på forbruget af ressourcer. Det har ikke været et mål for os optimere på forbruget af hukommelse og vi diskuterer og eksemplificerer hvorledes dette har medført, at der er et stort overhead på hukommelsesforbruget.

3.6 Mulige udvidelser

Sidst i denne del diskuterer vi hvilke udvidelser, man kunne lave til vores implementation. Den første udvidelse består i at gøre tiden for en forespørgsel konstant uden at forøge den asymptotiske konstruktionstid og det asymptotiske pladsforbrug. Den konstante forespørgselstid opnås ved at reducere antallet af stier, der skal gennemses i en forespørgsel til et konstant antal. Den anden udvidelse beskriver hvorledes man kan tilføje ekstra information, der gør det muligt at finde en vej mellem to knuder. Denne ekstra information ændrer ikke på det asymptotiske pladsforbrug og det øger heller ikke konstruktionstiden. Den sidste udvidelse, vi beskriver, har til formål at distribuere den information vi beregner på knuderne, således at det bliver overflødigt at opbevare de rekursionstræer, der er opbygget. Igen bliver der ikke ændret på den asymptotiske konstruktionstid eller pladsforbrug. Vi vil komme kort ind på at der overhead, der er på vores implementation, kunne gøres mindre, hvis vi anvendte labels. Udvidelserne er alle beskrevet i kapitel 11.

Kapitel 4

Reduktion til tolagsgrafer

I dette kapitel beskrives, hvorledes vi reducerer reachability-problemet i en planar, orienteret, sammenhængende graf G til reachability i tolagsgrafer. Med udgangspunkt i G konstruerer vi tolagsgraferne, $G_0, G_1, \ldots, G_{k-1}$, hvor k er en variabel afhængig af strukturen af G. En forespørgsel i grafen G vil blive oversat til en forespørgsel i nul, en eller to af tolagsgraferne. Konstruktionen fordrer ikke planaritet, men bevarer den, så vi efter reduktionen ved, at de resulterende tolagsgrafer alle er planare.

Vi vil definere tolagsgrafer og beskrive algoritmen for konstruktion af disse. I artiklen [Tho01] er beskrevet en fremgangsmåde for hvorledes tolagsgraferne kan konstrueres. Det vil fremgå tydeligt når vi afviger fra dette forslag og vi vil beskrive hvorfor vi i nogle tilfælde har valgt en anden fremgangsmåde end den beskrevne. På baggrund af konstruktionen af graferne vil vi bevise en række egenskaber for disse. Endelig vil vi bevise at tolagsgraferne kan konstrueres i lineær tid. Kapitlet bliver afsluttet med en beskrivelse af hvorledes vi har implementeret algoritmen for konstruktion af tolagsgraferne.

4.1 Definition af tolagsgrafer

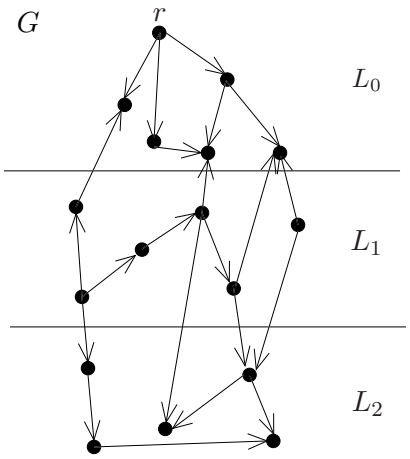
Vi vil i det følgende definere tolagsgrafer, som er meget centrale for konstruktionen af et reachability-orakel.

Definition 4.1 *Lad en planar, orienteret graf G være givet. Lad T være et udspændende træ i den underliggende uorienterede graf for G og kald roden i træet for r. Træet T er et* tolags-udspændende træ *hvis enhver sti fra roden i T højst er sammensat af to orienterede stier i G.*

En tolagsgraf *er en orienteret graf, der indeholder et tolags-udspændende træ.*

4.2 Konstruktion af tolagsgrafer

Lad G være en planar og orienteret graf. Mængden af knuderne i G er $\mathcal{V}(G)$, mængden af kanter er $\mathcal{E}(G)$ og vi kalder antallet af knuder i grafen for n. Da grafen er planar og ikke indeholder dobbeltkanter har vi ifølge Eulers formel, at



Figur 4.1: En rod vælges og grafen deles op i lag vha. kanternes orientering.

antallet af kanter er lineært i antallet af knuder. For at konstruere tolagsgraferne skal knuderne i G deles op i knudedisjunkte *lag* $L_0, L_1, \ldots, L_k$.

Definition 4.2 *Et* lag L_i *i en orienteret graf* G *defineres som*

$$L_0 = \{v \in \mathcal{V}(G) \mid r \rightsquigarrow v\}$$

$$L_i = \begin{cases} \{v \in \mathcal{V}(G) \setminus \bigcup_{j<i} L_j \mid v \rightsquigarrow \bigcup_{j<i} L_j\} & \textit{hvis } i \textit{ er ulige} \\ \{v \in \mathcal{V}(G) \setminus \bigcup_{j<i} L_j \mid \bigcup_{j<i} L_j \rightsquigarrow v\} & \textit{hvis } i \textit{ er lige} \end{cases}$$

hvor $v \rightsquigarrow w$ *betyder, at der er en vej fra* v *til* w.

Inddelingen af grafen G i lag foregår i følgende trin:

1. En vilkårlig rod r vælges blandt knuderne i G.

2. Første lag L_0 defineres til at bestå af r og alle knuder, der kan nås fra den valgte rod (enten direkte eller indirekte).

3. De følgende lag defineres skiftevis som de knuder, der hhv. kan nå det foregående lag og de knuder, der kan nås fra det foregående lag.

Processen slutter iterationen før et tomt lag konstrueres. En knude v indekseres med det lag $\iota(v)$ den tilhører, dvs. tilhører v lag L_i er $\iota(v) = i$. Da grafen er sammenhængende vil alle knuder blive indekseret. Lagdelingen er illustreret på figur 4.1

Lagene definerer graferne $G_0, G_1, \ldots, G_{k-1}$. Knuderne i hver graf er defineret af to på hinanden følgende lag og en ekstra knude r_i der er fremkommet ved en

kontraktion af de foregående lag.

$$\begin{aligned}\mathcal{V}(G_0) &= L_0 \cup L_1 \\ \mathcal{V}(G_1) &= L_1 \cup L_2 \cup \{r_1\} \\ &\vdots \\ \mathcal{V}(G_{k-1}) &= L_{k-1} \cup L_k \cup \{r_{k-1}\}\end{aligned}$$

Denne konstruktion medfører, at en knude er med i netop én graf, hvis den tilhører første eller sidste lag, og i to grafer ellers. En kant $(v, w) \in \mathcal{E}(G)$ er inkluderet i G_i hvis både v og w er i G_i. Kanter hvor det ene endepunkt tilhører den kontraherede rod og det andet er i grafen G_i bliver ligeledes inkluderet. For grafen G_i bliver dette i praksis kanter mellem knuder i L_{i-1} og knuder i L_i. Mere formelt:

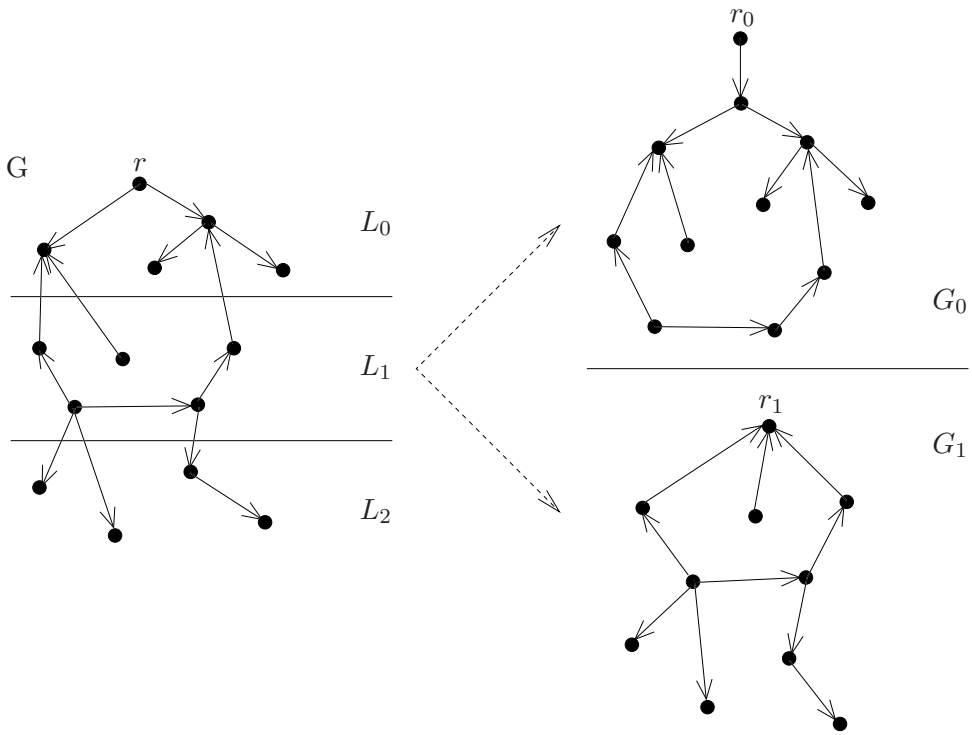
Definition 4.3 *Lad en en planar orienteret graf G være givet og definer lagene $L_0, L_1, \ldots, L_k$ som i definition 4.2. Tolagsgraferne $G_0, G_1, \ldots, G_{k-1}$ defineres som*

$$\begin{aligned}
G_i =& (V_i, E_i) && \text{for } 0 \leq i < k \\
V_0 =& L_0 \cup L_1 && \\
V_i =& L_i \cup L_{i+1} \cup \{r_i\} && \text{for } 0 < i < k \\
E_i =& E_i' \cup \{(v, w) \mid (v, w) \in \mathcal{E}(G) \wedge \\
& \qquad v \in L_i \cup L_{i+1} \wedge \\
& \qquad w \in L_i \cup L_{i+1}\} && \text{for } 0 \leq i < k \\
E_i' =& \{(r_i, w) \mid \exists v \in L_{i-1} \wedge w \in L_i \wedge \\
& \qquad (v, w) \in \mathcal{E}(G)\} \\
& && \text{for } 0 \leq i < k \wedge \ i \text{ lige} \\
E_i' =& \{(v, r_i) \mid v \in L_i \wedge \exists w \in L_{i-1} \wedge \\
& \qquad (v, w) \in \mathcal{E}(G)\} \\
& && \text{for } 0 \leq i < k \wedge \ i \text{ ulige}
\end{aligned}$$

4.2.1 Ændringer i konstruktionen af tolagsgrafer

Tolagsgraferne $G_1, G_2, \ldots, G_{k-1}$ adskiller sig fra G_0 idet de har en rod, der ikke er fra den oprindelige graf G. I afsnit 8.2 beskrives, at vi ikke ønsker at se på stier over en rod, der ikke er en del af den oprindelige graf. Da vi ønsker at behandle alle tolagsgrafer ens, har vi valgt også at indsætte en ny rod r_0 i grafen G_0, og en kant fra denne til den gamle rod. Der vil nu ikke opstå specialtilfælde i forbindelse med roden. Et eksempel på, hvordan en graf laves om til tolagsgrafer kan ses på figur 4.2.

Ved at tilføje en ekstra knude og en kant fra denne til grafen G_0 har vi ikke fjernet eksisterende veje i forhold til den oprindelige konstruktion af G_0.



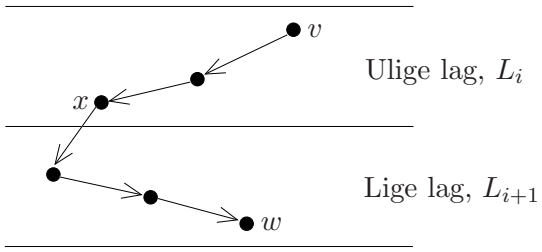
Figur 4.2: Grafen deles op i mindre grafer hver med to lag.

Det er ikke muligt at lave en forespørgsel, som inkluderer en knude, der ikke er en del af grafen G og vi har ikke tilføjet nye kanter mellem knuder fra G. Det betyder, at vi ikke har tilføjet nye veje i forbindelse med vores ændring af konstruktionen.

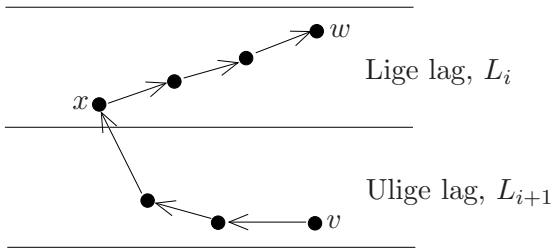
4.2.2 Tolags-udspændende træ

Grafen G_i skal indeholde et tolags-udspændende træ T for at være en tolagsgraf. Antag at i er lige. Da kan man fra roden r_i nå enhver knude i G_i, der kommer fra lag L_i. Alle knuder i L_{i+1} kan nå knuder fra L_i, det betyder, at der fra enhver knude i G_i fra lag L_{i+1} vil være en sti op til L_i. Ved et bredde først-gennemløb startende i r_i kan der laves et udspændende træ, der dækker knuder i G_i, der stammer fra L_i. Herefter kan man fra knuder, der er blevet dækket af en udspændende kant lave et bredde først-gennemløb mod kanternes orientering og undervejs opbygges et udspændende træ for den resterende del af grafen. En sti i træet fra roden til en knude er højst sammensat af to orienterede stier; en orienteret sti konstrueret i det første bredde først-gennemløb og en sti med modsat orientering, der er konstrueret i det andet bredde først-gennemløb.

Tilfældet hvor i er ulige er symmetrisk, her starter man med at lave et bredde først-gennemløb mod orienteringen af kanterne efterfulgt af et gennemløb, der følger kanternes orientering.



Figur 4.3: Stien P skærer to lag, det med mindst indeks er et ulige lag.



Figur 4.4: P skærer to lag, det med mindst indeks er et lige lag.

4.3 Egenskaber for tolagsgrafer

Vi vil i dette kapitel bevise egenskaber for tolagsgraferne $G_0, G_1, \ldots, G_{k-1}$ og lagene $L_0, L_1, \ldots, L_k$, der er konstrueret som beskrevet i afsnit 4.2.

Sætning 4.4 *Der eksisterer en sti fra knuden v til w i G, hvis og kun hvis der i $G_{\iota(v)-1}$ eller $G_{\iota(v)}$ eksisterer en sti fra v til w.*

Bevis. Antag at w kan nås fra v i G og lad P være en sti fra v til w. Lad i være det mindste indeks på et lag, der skæres af stien P.

Antag at L_i er et ulige lag, og lad x være den sidste knude på stien P i L_i, se figur 4.3. Da x befinder sig i et ulige lag og intet af P befinder sig i et foregående lag må knuder, der kan nå x, dvs. knuderne fra v til x, ligeledes befinde sig i lag L_i. Hvis knuderne i P ikke er fuldstændig indeholdt i lag L_i må de resterende knuder pr. konstruktion af lagene være i lag L_{i+1}. Knuder på stien P er altså indeholdt i $L_i \cup L_{i+1}$ og dermed i tolagsgrafen G_i. Knuden v befinder sig i L_i det betyder at $\iota(v) = i$ og dermed er stien P indeholdt i grafen $G_{\iota(v)}$.

Situationen er tilsvarende, hvis vi antager at L_i er et lige lag, og x er den første knude på stien P i L_i, se figur 4.4. Pr. konstruktion af lagene og da ingen af knuderne i P befinder sig i foregående lag må knuder på P som kan nås af x, dvs. knuderne fra x til w være i lag L_i. Resten af knuderne i P vil (medmindre alle knuderne i P er indeholdt i L_i) befinde sig i L_{i+1}, da alt som kan nå x og ikke er i L_i, befinder sig i L_{i+1}. I dette tilfælde er v enten indeholdt i L_i eller L_{i+1}, dvs. $\iota(v) = i$ eller $\iota(v) = i + 1$. Stien P er indeholdt G_i som er lig $G_{\iota(v)}$ eller $G_{\iota(v)-1}$.

Dermed er det vist, at hvis v kan nå w i G, kan v også nå w i enten $G_{\iota(v)-1}$ eller $G_{\iota(v)}$.

Beviset for den modsatte vej er oplagt. □

Sætning 4.5 *Givet en planar graf G er tolagsgraferne $G_0, G_1, \ldots G_{k-1}$ planare.*

Bevis. Tolagsgraferne $G_0, G_1, \ldots, G_{k-1}$ er konstrueret ved en sammentrækning af incidente knuder i G og her bevares planaritet. I tolagsgrafen G_0 er tilføjet en ekstra knude og en kant, men dette kan ikke ændre grafens planare egenskaber. □

Sætning 4.6 *Givet en planar, orienteret graf G, lad tolagsgraferne G_i være defineret som i definition 4.3, da gælder at størrelsen af tolagsgraferne er lineær i størrelsen af G.*

$$\sum_{i=0}^{k-1}(|\mathcal{V}(G_i)| + |\mathcal{E}(G_i)|) \in \mathcal{O}(n)$$

Bevis. Da tolagsgraferne ifølge sætning 4.5 er planare er det tilstrækkeligt at vise, at antallet af knuder eller kanter i tolagsgraferne er lineært i antallet af knuder i G. Vi har valgt at at vise begge ting, da det øger forståelsen for opbygningen af tolagsgraferne.

Vi observerer, at laginddelingen af G er en partition af knuderne og at hver knude højst er med i to af de konstruerede tolagsgrafer. Derudover er der en ekstra rod for hver af tolagsgraferne G_i. Dermed har vi at $\sum_{i=0}^{k-1}|\mathcal{V}(G_i)| \leq 2n + n \in \mathcal{O}(n)$.

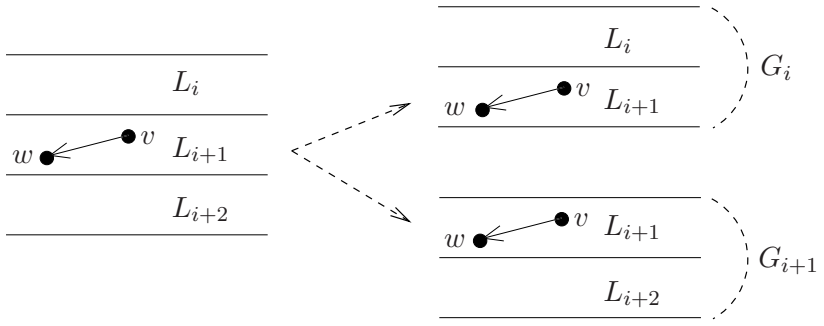
Hver kant $(v, w) \in G$ kan højst være anledning til to kanter i de nye grafer. Der er to mulige situationer:

1. Antag v og w er i samme lag, da vil kanten optræde i én tolagsgraf hvis $(v, w) \in L_0 \cup L_k$ og i to tolagsgrafer ellers, se figur 4.5

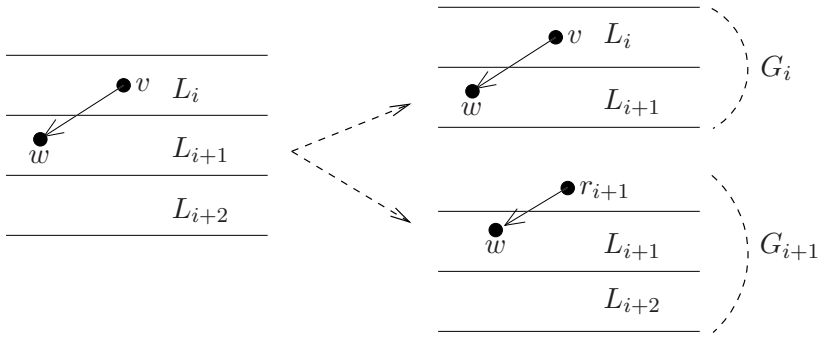
2. Antag er v og w i to lag, der er umiddelbart efter hinanden. Vi kan uden tab af generalitet antage, at v er i det mindste af de to lag. Kanten vil da optræde i den graf, som både v og w findes i, samt som en kant mellem w og den rod, som v's lag er blevet sammentrukket til, se figur 4.6.

Konstruktionen af lagene medfører, at hvis knuderne v og w ikke befinder sig i samme lag må de befinde sig i to på hinanden følgende lag. Vi har i konstruktionen af G_0 tilføjet en ekstra kant fra den nye rod, dvs. samlet får vi at $\sum_{i=0}^{k-1}|\mathcal{E}(G_i)| \leq 2|\mathcal{E}(G)| + 1 \in \mathcal{O}(n)$.

□



Figur 4.5: En kant, der har endepunkter i samme lag, bliver til to kanter i tolagsgraferne.



Figur 4.6: En kant, der har endepunkter i to forskellige lag, bliver til to kanter i tolagsgraferne, hvoraf den ene er en kant til roden.

4.4 Udførelsestid

Vi har beskrevet hvorledes vi med udgangspunkt i en orienteret, planar graf G kan konstruere tolagsgrafer og vi har bevist en række egenskaber for tolagsgraferne $G_0, G_1, \ldots, G_{k-1}$. På baggrund af dette vil vi bevise, at G_i'erne kan konstrueres i lineær tid.

Sætning 4.7 *Givet en planar orienteret graf G kan vi i lineær tid konstruere en mængde af orienterede planare tolagsgrafer $G_0, G_1, \ldots, G_{k-1}$.*

Bevis. Knuderne i grafen G deles op i lagene $L_0, L_1, \ldots, L_k$ ved brug af et bredde først-gennemløb. Dette kan gøres i lineær tid og i afsnit 4.6 beskrives vores implementation af algoritmen.

En tolagsgraf G_i konstrueres ved at kopiere knuderne fra $L_i \cup L_{i+1}$ og konstruere en rod r_i. Kanter mellem de valgte knuder tilføjes og kanter, der er incidente til $v \in L_i$ og $w \in L_{i-1}$ tilføjes som kanter mellem kopien af v og r_i. Dette kan gøres i tid lineær i størrelsen af G_i. Dvs. konstruktionstiden for alle tolagsgraferne bliver lineær i størrelsen af disse. Fra sætning 4.6 har vi, at den samlede størrelse af G_i'erne er lineær i størrelsen af G. Vi mangler nu at bevise at tolagsgraferne er planare, hvilket vi har fra sætning 4.5. □

4.5 Det reducerede problem

Det oprindelige problem, reachability i en planar, orienteret graf, er nu reduceret til reachability i $k-1$ planare, orienterede tolagsgrafer. For at kunne svare på, om to knuder i en planar, orienteret graf G kan nå hinanden, skal vi kunne svare på, om disse kan nå hinanden i relevante tolagsgrafer. For en knude v er de relevante tolagsgrafer jf. sætning 4.4 $G_{\iota(v)-1}$ og $G_{\iota(v)}$. Målet er, at kunne lave et reachability-orakel for G_i, da dette vil betyde, at vi kan lave et reachability-orakel for den oprindelige graf G. I kapitel 5-9 beskriver vi hvorledes et reachability-orakel for en tolagsgraf konstrueres.

4.6 Implementation

Vi vil i dette afsnit give en kort beskrivelse af hvordan vi ud fra en planar, orienteret graf G konstruerer tolagsgraferne $G_0, G_1, \ldots, G_{k-1}$. Først beskrives hvordan lagene $L_0, L_1, \ldots, L_k$ findes og derefter beskrives konstruktionen af tolagsgraferne. Til sidst argumenterer vi for, at vores implementation overholder den ønskede udførelsestid.

Vi bruger to køer k_1 og k_2 i forbindelse med indeksering af knuderne. Den ene kø k_1 indeholder knuder vi arbejder med i lag L_i, mens den anden indeholder knuder, der skal indekseres med $i+1$. Vi starter i grafen G med at udvælge en vilkårlig knude og denne kaldes for r. Vi putter r ind i k_1 og denne kø er udgangspunktet for et bredde først-gennemløb, der følger kanternes orientering.

Antag at i er lige og at vi er ved at konstruere laget L_i. Når en knude v tages ud af køen k_1 gennemløber vi v's udkanter og ser på w, der er kantens andet endepunkt. Hvis w ikke er besøgt sættes den ind i k_1. Vi gennemløber også v's indkanter (u, v). Hvis u ikke er besøgt sættes den ind i køen k_2. Denne kø er udgangspunktet for et bredde først-gennemløb mod kanternes orientering under hvilket knuder, der skal tilhøre lag L_{i+1} indekseres.

Tilfældet hvor i er ulige er symmetrisk.

Processen fortsættes indtil der ikke er flere knuder, der kan besøges. Da grafen G er sammenhængende ved vi, at alle knuder er indekseret. Vi ser højst på alle kanterne to gange; én gang fra hver endeknude, dvs. vi bruger lineær tid på at indeksere lagene.

Tolagsgraferne konstrueres en efter en fra G_0 til G_{k-1}. For hver graf laves først en ny knude, der bliver grafens rod. For grafen G_i ser vi på knuderne i lag L_i og L_{i+1}. Når en knude v indsættes i G_i, gennemløbes alle kanter, der er incidente med v. Hvis det andet endepunkt, w allerede findes i den nye graf indsættes kanten i G_i, hvis w tilhører et tidligere lag indsættes en kant fra v til r_i. Konstruktionen af G_i'erne tager lineær tid i antallet af kanter i G_i, da vi ser på hver kant en gang for hver af dens endepunkter. Vi har fra sætning 4.6, at det samlede antal af kanter i G_i-graferne er lineært i n. Vi får da at udførelsestiden for at konstruere G_i-graferne er lineært størrelsen af G

Til sidst laves et udspændende træ i tolagsgrafen G_i. Det udspændende træ konstrueres således, at det er et tolags-udspændende træ (se definition 4.1), idet der først konstrueres et træ, der dækker det første af grafens to lag og

derefter laves resten af træet. Beskrivelsen i afsnit 4.2.2 af hvordan et tolags-udspændende træ konstrueres svarer til vores implementation og udførelsestiden er ligeledes lineær i n.

4.6.1 Udførelsestid for konstruktion af tolagsgrafer

Vi har netop forklaret hvordan vi ud fra en planar orienteret graf G konstruerer tolagsgrafer. Konstruktionen består af tre faser og vi har efter beskrivelsen af hver fase argumenteret for at udførelsestiden er lineær i størrelsen af G. Dette giver anledning til følgende korollar.

Korollar 4.8 *Givet en planar orienteret graf G kan vi konstruere tolagsgraferne $G_0, G_1, \ldots, G_k$ i tid $\mathcal{O}(n)$.*

4.6.2 Placering i programmet

Den del af vores implementation, der indholder opdeling af grafen i lag findes i `reachability/layer.cpp` og den del der konstruerer $G_0, G_1, \ldots, G_{k-1}$ med tilhørende udspændende træer findes i `reachability/gi.cpp`. Selve tolagsgraferne er implementeret i klassen `reachability/LayeredGraph`.

Kapitel 5

Separation af tolagsgrafer

I dette og de følgende kapitler (6–9) vil vi beskrive hvordan vi arbejder videre med de tolagsgrafer, der er konstrueret i kapitel 4 med udgangspunkt i den oprindelige graf G.

Vi laver en hierarkisk dekomposition af en tolagsgraf G_i, hvor vi rekursivt deler grafen op i mindre dele. Dekompositionen foretages rekursivt vha. *separatorer*. En separator er en samling af stier, som opdeler en graf i mindre dele. Rekursionsdybden ønskes at være logaritmisk, hvorfor delgraferne højst må være halvt så store, som den graf, der opdeles. Vi ønsker derfor at kunne opdele en tolagsgraf H i tre dele, der hver højst indeholder halvt så mange knuder som H. Denne opdeling vil vi fortsætte rekursivt med, indtil alle tre dele er tomme (dvs. at de ikke indeholder knuder). Hver del (inkl. H selv) bidrager med information til det endelige orakel. Oraklet gemmes som et rekursionstræ, der svarer til den rekursion, der anvendes i opdelingen af grafen. Således er der en knude i rekursionstræet for hvert rekursivt kald vi foretager i vores dekomposition. Hver knude i rekursionstræet indeholder information, som vi har beregnet i det tilsvarende kald i rekursionen.

I dette kapitel beskrives hvilke skridt, der er i dekompositionen. Vi fokuserer på, hvorledes vi kan finde en separator, som kan opdele en tolagsgraf H i tre mindre dele, der alle opfylder kravet om højst at indeholde $|\mathcal{V}(H)|/2$ knuder. Vi ønsker at finde denne separator i lineær tid, ligesom vi ønsker at foretage selve opdelingen af grafen i lineær tid. Dette vil betyde, at vi for en tolagsgraf G_i kan lave hele den rekursive opdeling i tid $\mathcal{O}(|\mathcal{V}(G_i)| \log(|\mathcal{V}(G_i)|))$, da rekursionsdybden er logaritmisk.

Når vi har fundet separatoren, finder vi de orienterede stier, som denne består af. Vi ønsker, at der for en separator findes et konstant antal orienterede stier, da dette vil betyde at der på en sti i rekursionstræet for grafen G_i højst er $\mathcal{O}(\log(|\mathcal{V}(G_i)|))$ stier. Udførelsestiden for forespørgsler afhænger af dette antal stier.

Vi starter med en teoretisk gennemgang af strategien for opdeling af grafen i et kald i den rekursive dekomposition. Nogle af faserne nævner vi kort, da disse faser uddybes i de efterfølgende kapitler. Herefter gennemgår vi vores implementation i detaljer – igen vil nogle faser udelukkende være nævnt kort, da de beskrives i et efterfølgende kapitel.

5.1 Strategi for opdeling af grafen

Givet en planar tolagsgraf H med et tolags-udspændende træ T, skal vi finde en opdeling af knuderne og dele grafen op i tre dele mht. denne opdeling. Vi erindrer at T er et udspændende træ for den underliggende uorienterede graf, der svarer til H.

Definition 5.1 *Lad r være roden i T og lad v være en knude i H. Stien, der går mellem v og r i T kaldes en* rodsti *og betegnes $T(v)$. Stien består af op til to orienterede stier.*

For at kunne finde opdelingen er det nødvendigt at forberede H, hvilket er det første skridt i opdelingen (vi beskriver dette skridt i detaljer i afsnit 5.1.1). Næste skridt er at finde en flade, der defineres af tre knuder, x, y og z – denne flade definerer opdelingen. Der er stier i det udspændende træ, som går fra disse tre knuder til roden; $T(x)$, $T(y)$ og $T(z)$. Ved fjernelse af disse stier deles grafen op i tre mindre dele. Vi kalder de tre rodstier for en *separator*, da den opdeler grafen. Afsnit 5.1.2 beskriver kort hvorledes opdelingsfladen findes. Dette vender vi tilbage til i kapitel 6.

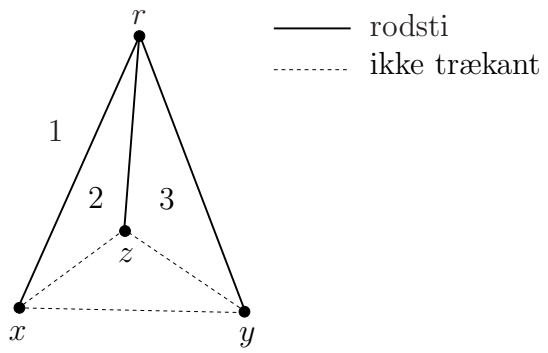
Når vi har fundet de tre rodstier, skal vi identificere alle orienterede stier i separatoren, da disse skal anvendes til at finde information, der skal gemmes i oraklet og senere anvendes til at svare på spørgsmål om reachability. Disse stier identificeres inden opdelingen af grafen foretages. Identificeringen beskrives i afsnit 5.1.3. Det sidste skridt, der er selve opdelingen, beskrives kort i afsnit 5.1.4.

5.1.1 Forberedelse af H

Tolagsgrafen H forberedes, idet H indlejres på en sfære og trianguleres. Således vil de kanter, der er incidente med en knude v, nu forekomme i en fastlagt rækkefølge. Indlejringen på en sfære betyder, at alle flader har tre naboflader. Vi kalder den triangulerede graf H'. Grafen har et antal flader, der alle er defineret af tre knuder og af de tre kanter, der går mellem disse knuder. Alle disse flader grænser op mod tre andre flader, da grafen befinder sig på en sfære. Det udspændende træ T for H er ligeledes et udspændende træ for H', da vi ikke har ændret på hvilke knuder, grafen indeholder, vi har blot indsat yderligere kanter, da vi triangulerede grafen.

5.1.2 Lokalisering af knuder

Når H er forberedt, identificeres en flade, der defineres af tre knuder, som grafen skal deles op efter. Den flade, som de tre knuder udspænder, er en af de flader, der blev konstrueret, da vi triangulerede grafen. Den flade vi skal finde, skal definere en opdeling af H' i tre dele, hvor ingen af de tre dele indeholder mere end $|\mathcal{V}(H)|/2$ knuder. Denne flade kan findes i lineær tid, hvilket beskrives i kapitel 6. Hvorledes en flade deler grafen i tre dele kan ses på figur 5.1.



Figur 5.1: Opdeling af grafen vha. rodstier fra x, y og z.

5.1.3 Identificering af separator

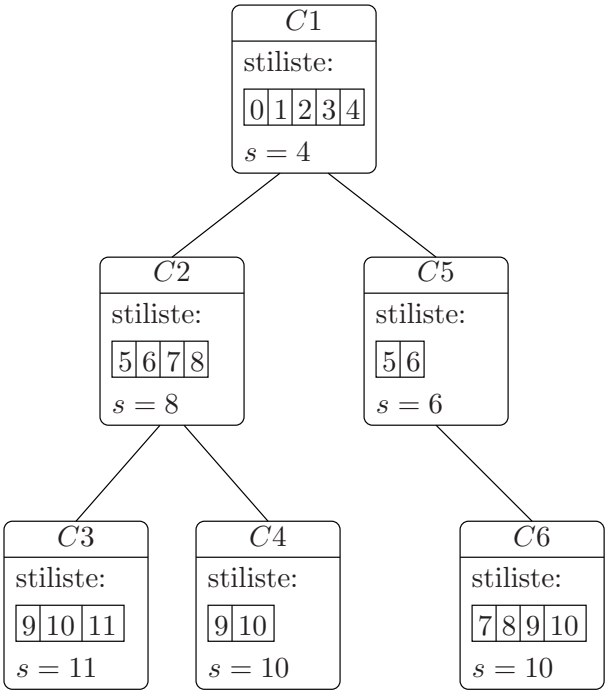
Antag, at vi står med en planar tolagsgraf H', der er indlejret på en sfære og trianguleret og har et udspændende træ T med en rod r. Vi antager ligeledes, at der er fundet en flade, som defineres af tre knuder, x, y og z, som definerer en opdeling i tre dele, der hver højst indeholder $|\mathcal{V}(H)|/2$ knuder.

Definition 5.2 *Antag at en trianguleret tolagsgraf H med udspændende træ T samt en flade (x, y, z) er givet. Vi definerer* separatoren *S, til at være rodstierne $T(x)$, $T(y)$ og $T(z)$. Disse opdeler grafen i tre dele, se figur 5.1.*

Givet fladen (x, y, z) finder vi nu separatoren S, som består af rodstierne i tolagstræet T fra hhv. x, y og z. Den findes ved at starte ved hhv. x, y og z, og gå opad i træet, til vi når r. Separatoren kommer til at bestå af op til seks orienterede stier, da hver rodsti består af op til to orienterede dele. Vi kalder en orienteret sti i en separator for en *separatorsti*. Separatorstierne skal senere anvendes til at finde information, der skal bruges til at svare på spørgsmål om reachability i grafen.

For at kunne beregne den information, der er nødvendig for at kunne repræsentere orienteret reachability, må alle separatorstierne i S nummereres. I en knude i rekursionstræet, der repræsenterer et kald C, gemmes et *separatornummer* s, der er nummeret på den sidste separatorsti, der findes i kaldet C. I de rekursive kald der laves fra C, starter nummereringen ved $s+1$. Nummereringen er dermed fortløbende fra roden ned gennem en vilkårlig sti i rekursionstræet. Et eksempel kan ses på figur 5.2. Her ses et rekursionstræ, der bl.a. indeholder kaldet $C3$, hvor der er lavet tre separatorstier. Disse får numrene 9, 10 og 11, da numrene $0-8$ er brugt i de foregående kald. Separatornummeret s i kaldet $C3$ er 11, da dette er det sidste nummer, der er anvendt i dette kald.

Hvorledes vi gemmer information i oraklet, der repræsenterer orienteret reachability og hvorledes forespørgsler besvares, er beskrevet i detaljer i kapitel 8 og 9.



Figur 5.2: Eksempel på rekursiontræ med lister af stier og separatornumre. Nummereringen starter ved roden i rekursionstræet og er fortløbende i alle stier ned gennem træet.

5.1.4 Opdeling ved hjælp af en separator

Den trekantede flade og de tre rodstier definerer tre dele af grafen, som det kan ses på figur 5.1. I kapitel 7 beskrives hvorledes grafen i lineær tid kan opdeles i op til tre delgrafer vha. separatoren.

5.1.5 Udførelsestid

Vi skal argumentere for, at de ovennævnte fire skridt i opdelingen alle kan udføres i lineær tid, da det vil give en samlet lineær udførelsestid.

Forberedelse af H. Da der er tale om en planar graf, kan den trianguleres i lineær tid (jf. [HU]).

Lokalisering af knuderne x, y og z. I kapitel 6 gives beviser for, at fladen med de tre knuder x, y og z, der opdeler grafen i tre dele, der hver højst indeholder $\mathcal{V}(H)/2$ knuder, eksisterer (sætning 6.1) og kan findes i lineær tid (sætning 6.10).

Identificering af separator. Givet, at hver knude i træet T har en reference til sin forælder, kan vi finde separatoren i lineær tid, da vi blot skal følge disse forælder-referencer op gennem træet fra tre forskellige knuder. Separatoren skal gennemløbes for at identificere de op til seks orienterede stier, som separatoren indeholder. Denne fase foregår derfor i lineær tid i størrelsen af separatoren – denne er $\mathcal{O}(\mathcal{V}(H))$.

Opdeling vha. separator. I kapitel 7 gennemgår vi, hvorledes vi kan lave opdelingen i delgrafer i lineær tid.

5.2 Implementation

Vores implementation af opdelingen involverer en del mere arbejde, end den oprindelige strategi fra artiklen [Tho01] antyder. Det er nødvendigt at forberede grafen på en omfattende måde, da de forskellige faser af opdelingen kræver, at der er adgang til en lang række informationer om knuder og kanter i konstant tid. Specielt fasen, der finder den flade, der definerer opdelingen, anvender denne information.

I det følgende beskrives hvorledes de forskellige skridt er implementeret, og hvilke ændringer, der er lavet i forhold til den oprindelige strategi, som den er beskrevet i afsnit 5.1. Hvordan fladen med de tre opdelingsknuder findes, er beskrevet i kapitel 6. Hvordan selve opdelingen foretages, når separatoren er identificeret, beskrives i kapitel 7.

Alle de anvendte eksempler i beskrivelsen af implementationen anvender tolagsgrafer, hvor det første lag er udgående fra roden og det andet lag vender mod roden. Vores implementation for grafer, hvor lagene er omvendt sat sammen, er helt analog.

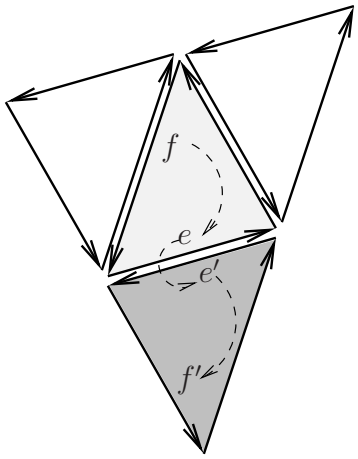
5.2.1 Forberedelse af H

Indlejring og triangulering

Vi anvender nogle algoritmer fra LEDA, der kræver at grafen er en *tovejsgraf* (på engelsk: *bidirected graph*). Grafen laves derfor til en tovejsgraf vha. LEDA's `make_bidirected()`, som indsætter ekstra kanter og skaber relationer mellem hvert par af kanter i lineær tid ([Alg], side 195 og 202). Dette betyder, at man for enhver kant har adgang til en modsatrettet kant i konstant tid. For en kant $e = (v, w)$ kaldes den modsatrettede kant $e' = (w, v)$ også for e's *tvillingekant*.

Når grafen er en tovejsgraf, kan LEDAs funktion `make_planar_map()` omarrangere kanterne i grafen, således at deres rækkefølge repræsenterer en indlejring i planen. Hver knude har dermed et array af udkanter, og de er ordnet som de forekommer mod uret i den valgte indlejring. Ligeledes har hver knude et array af indkanter. Funktionen `make_planar_map()` har lineær udførelsestid ([Alg], side 198 og 202).

Nu kan grafen trianguleres og igen anvendes en lineærtids-funktion fra LEDA: `triangulate_planar_map()` (se [MN99], side 566). Denne funktion indsætter kanter i grafen, således at den er trianguleret og bevarer, at grafen er en tovejsgraf. Funktionen beregner en liste over grafens flader og sørger for at knytte en flade til de kanter, der omkredser fladen og omvendt. Således kan man, givet en flade, få fat i en af kanterne på randen i konstant tid og fra denne kant få de efterfølgende kanter, der ligger på randen af fladen. Man kan ligeledes i konstant tid, givet en kant, få fat i den flade, der befinder sig til venstre for kanten ift. dens orientering.



Figur 5.3: Adgang fra fladen f til nabofladen f'.

Når vi laver en graf til en tovejsgraf og triangulerer den, tilføjes ekstra kanter. Disse kanter markeres, således at vi kan kende disse kanter fra grafens originale kanter.

Vi har nu en graf, som er trianguleret og kan opfattes som indlejret på en sfære, idet vi for en flade f har adgang til tre naboflader. Dette har vi da f har adgang til de kanter, der ligger på randen af den. Fra en sådan kant e har vi igen adgang til dens tvillingekant e' og sidst har vi fra tvillingekanten adgang til en af de tre flader, der grænser op til f. På figur 5.3 ses, hvordan man fra fladen f via kanten e og dens tvillingekant e' har adgang til nabofladen f'.

Information på knuder og kanter

Vi tilføjer en række informationer til hver knude og kant. Disse informationer skal bl.a. anvendes til at finde opdelingsfladen (kapitel 6), samt til at opdele grafen (afsnit 7.2). Informationerne beregnes og tilføjes til grafen vha. et konstant antal gennemløb af grafens knuder og kanter. Dette trin i opdelingen foregår derfor i lineær tid.

Vi sætter labels på hhv. knuder og kanter, således at den ønskede information kan tilgås i konstant tid fra en given knude eller kant. De fleste af informationerne skal anvendes til at finde opdelingsfladen. Hvorledes disse anvendes er illustreret i kapitel 6. De følgende informationer beregnes og gemmes i en label for hver knude:

Knude. En knudes label ved hvilken knude, den er tilknyttet.

Dybde. Knudens dybde i det udspændende træ gemmes.

Forælder. Knudens forælder i det udspændende træ.

Forælder-kant. Den kant, der går fra knuden til dens forælder. Denne kant er ikke nødvendigvis med i det udspændende træ, da kanten i træet, som går

mellem knuden og dens forælder evt. vender den modsatte vej. Vi vælger den kant, der vender fra knuden mod forældre-knuden.

Indeks ift. søskende. Knudens forælder indekserer sine børn. Et barn ved selv hvilket indeks det har hos dets forælder.

Kanter til børn. En liste over de kanter fra knuden, der peger på knudens børn i retning mod uret.

Prefixsum over størrelsen af undertræerne. Prefixsum over antallet af knuder, der er i undertræerne. Når prefixsummen beregnes bruges den samme rækkefølge af kanterne, som i listen over kanter til børn.

Følgende informationer beregnes og gemmes i et label for hver kant:

Kantnummer ift. startknude. For hver knude nummereres de udgående kanter mod urets retning. Disse numre gemmes i kanterne.

Næste udspændende kant. Den næste kant mod urets retning, der er udspændende. Er kanten selv udspændende, peger den på sig selv.

Foregående udspændende kant. Som ovenstående, men her er det kanten i modsatte retning, der vælges.

Kantens type. Kanten har en af følgende typer: Kanten er en *original kant*, hvis kanten findes i den oprindelige graf. Kanten er en *trækant*, hvis den er med i det udspændende træ (disse kanter er også med i den oprindelige graf, men har sin egen type). Kanten er en *tovejskant*, hvis den er indsat, da vi lavede grafen til en tovejsgraf. Kanten er en *triangulering skant*, hvis den blev indsat, da vi triangulerede grafen.

5.2.2 Identificering af separator

Vi skal nu finde den separator S, som grafen H' skal deles op efter samt de orienterede stier, kaldet separatorstier, som denne indeholder.

Vi starter med at lokalisere den opdelingsflade (x, y, x), der definerer opdelingen. Separatoren S består da af rodstierne $T(x)$, $T(y)$ og $t(z)$.

Vi har som et eksperiment valgt, at når vi finder de orienterede stier, der er i separatoren, vil vi finde de længst mulige stier. Dette gør vi for at afgøre om vi på denne måde kan få færre separatorstierstier, uden at skulle gennemgå alle stierne bagefter for evt. at luge ud i antallet. Vi gennemgår senere i dette afsnit de observationer vi har gjort om maksimale stier og vender i afsnit 10.6 tilbage til resultatet af vores eksperimenter.

Når separatorstierstierne er fundet deler vi graferne op. Hvordan selve opdelingen foregår beskriver vi i kapitel 7.

Observationer om maksimale stier

I et forsøg på at gøre antallet af separatorstierstier mindre har vi valgt at eksperimentere med en anden fremgangsmåde end blot at finde de to dele fra

hver af x, y og x mod roden. Vi har set på, hvorledes stierne har fælles dele og på anden vis hænger sammen, når rodstierne $T(x)$, $T(y)$ og $T(z)$ ikke er disjunkte. Fælles for mange af de scenarier er, at der er ofte er stier der er ens eller overlappende – der kan altså være overflødige stier, hvilket vi ikke er interesserede i. For at undgå at skulle gennemløbe alle par af stier for at finde ens stier og stier, der er helt indeholdt i andre stier, har vi valgt at søge S igennem og vælge orienterede stier, der har maksimal længde (se eksempler på figur 5.4).

Vi bemærker, at når separatorer trækkes sammen kan der opstå orienterede stier, der går hen over roden. Hvordan dette kan opstå vender vi tilbage til i afsnit 8.2. Vi ønsker ikke at have stier, der går hen over roden da disse stier ikke er stier, er stammer fra den oprindelige graf G. I dette tilfælde afviger vi derfor fra at finde maksimale stier.

Vi kan observere at en af delene af en rodsti kan være *tom*. Dette sker hvis f.eks. $T(x)$ kun består af en orienteret del. I dette tilfælde vil der under alle omstændigheder være en separatorsti mindre. En rodsti kan ligeledes være helt tom, hvis f.eks. x er roden.

Vores valg betyder ikke at det maksimale antal af stier er anderledes – der er stadig tilfælde, hvor der er seks stier (f.eks. tilfældet hvor $T(x)$, $T(y)$ og $T(z)$ er disjunkte). Der er dog flere tilfælde hvor der bliver færre stier, når vores strategi anvendes.

Vi vil i det følgende afsnit kort argumentere for hvor mange maksimale orienterede stier, der findes.

Antal orienterede stier

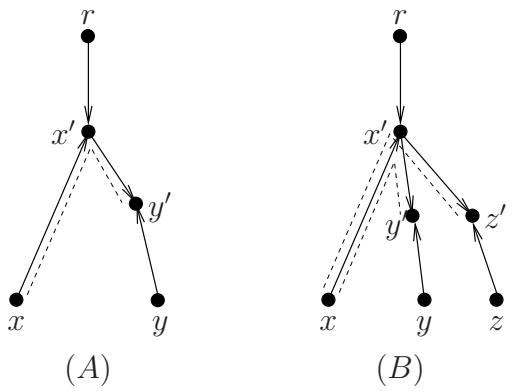
Vi vil nu argumentere for, at S, der består af $T(x)$, $T(y)$ og $T(z)$, er sammensat af højst seks maksimale orienterede stier. Vi antager at første lag i grafen er orienteret væk fra roden. Tilfældet med omvendte lag er symmetrisk. Separatoren S består af rodstier fra knuderne i fladen (x, y, z). Hver af disse rodstier består af op til to dele. På rodstien fra x kalder vi punktet hvor de to dele mødes for x'. De to dele er orienteret hhv. fra r mod x' og fra x mod x'. Ligeledes har vi punkterne y' og z'.

Vi observerer først, at da grafen er en tolagsgraf og da S består af stier, der går mellem knuder i fladen (x, y, z) og roden, vil alle orienterede stier i S enten starte i roden eller i en af knuderne i fladen (da vi ikke ønsker stier, der går over roden).

Vi betragter først stier fra roden r. Der kan maksimalt være tre stier, der leder hhv. mod x', y' og z'. Er en af disse indeholdt i en af de andre, vil der være en sti mindre, men der kan altid højst være tre stier, der starter i roden.

Vi betragter nu stier, der starter i en af knuderne i fladen (x, y, z). Hvis stierne er disjunkte, er der en sti fra x til x', fra y til y' samt fra fra z til z'. I dette tilfælde er der i alt tre stier fra disse knuder. Sammen med stierne fra roden er der altså i alt højst seks stier.

Vi ser nu på, hvornår der kan være mere end en sti, der starter i en af fladens knuder, f.eks. x. Dette kan lade sig gøre, hvis både y' og z' befinder sig i undertræet under x', se figur 5.4 (B). Hvis kun den ene, f.eks. y' er i



Figur 5.4: Hvorledes stien fra x kan fortsætte efter x'. (A) Sti fra x over x' til y' (B) Sti fra x over x' til y' samt en sti fra x over x' til z'

undertræet, er der stadig kun en sti fra x, se figur 5.4 (A). Vi bemærker at der ikke er andre muligheder for flere stier fra x, da det kun er $T(y)$ og $T(z)$, der kan give anledning til fortsættelser af stien fra x og at y' og z' skal være i undertræet under x' for at vi får stier, der vender væk fra x'.

Vi observerer nu, at hvis der er to stier fra x, er der kun én sti fra hver af y og z – der er altså maksimalt fire stier, der starter i x, y og z. I dette tilfælde kan der højst være to stier fra roden, da stien til x' fra roden nu er indeholdt i stierne til y' og z'. Der er altså stadig højst seks stier.

Vi samler konklusionerne om antallet af maksimale stier i følgende korollar:

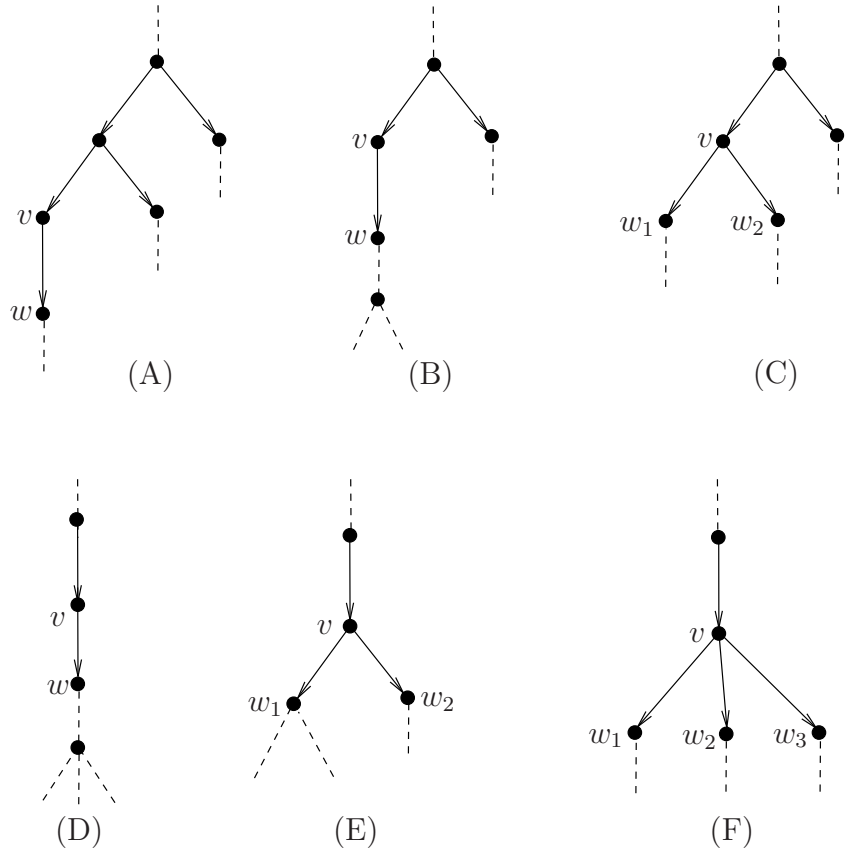
Korollar 5.3 *En separator S, der er defineret af fladen (x, y, z) indeholder højst seks maksimale orienterede stier. Der er maksimalt tre af disse stier, der starter i roden. Der er maksimalt to af stierne der starter i en af fladens knuder.*

Lokalisering af orienterede stier

Vi ser nu på hvorledes vi finder de forskellige stier. Vi ved, at de orienterede stier enten starter i roden r eller i en af knuderne i opdelingsfladen, x, y og z. Vi starter med at gennemgå hvorledes stierne fra roden findes. Herefter beskriver vi hvorledes stierne fra de resterende knuder findes.

Farvning

Før vi identificerer de orienterede stier farves de knuder, der er med i S, så de senere kan genkendes. Farvningen starter ved x og vha. forælder-referencer farves alle knuder på stien fra x til roden. På samme måde farves fra hhv. y og z til roden. Under farvningen gemmes referencer til alle kanter i S. For en given kant e, gemmes en reference til e i begge de incidente knuder. Referencerne gemmes i hhv. en udliste og en indliste i alle knuder. Disse referencer til kanter skal anvendes til at finde de orienterede stier i S. Vi gennemløber alle knuder i S under farvningen og denne tager derfor lineær tid.



Figur 5.5: Eksempler på forskellige stier, der starter i roden.

Stier fra roden

Vi ved, at der maksimalt er tre stier, der starter i roden. Vi har derfor tre gensidigt rekursive metoder, der med udgangspunkt i en knude v behandler situationer, hvor der maksimalt er hhv. en, to eller tre stier fra roden, der fortsætter under v. Først betragtes de kanter fra roden, som er med i separatoren og som blev gemt i en liste, da vi farvede separatoren. Hvis kun en kant udgår fra roden, kan stien senere dele sig, og blive til højst tre stier. Er der to kanter, kan en af disse dele sig i to stier, så der i alt bliver tre stier – hvilken der evt. deler sig, er det ikke muligt at afgøre, når man står i roden. Er der tre kanter, har vi allerede tre stier og ingen af disse kan dele sig yderligere.

Da vi maksimalt kan konstruere tre stier fra roden, konstruerer vi tre sti-objekter og roden sættes ind i dem alle. Argumenterne til et rekursivt kald på en knude i træet v er nu bl.a. et antal sti-objekter, der svarer til hvor mange stier, der maksimalt er under v. Når vi står i knuden v, indeholder sti-objekterne de knuder over v, som er med i separatoren – disse er allerede besøgt i rekursionen. Vi beskriver nu hvorledes vi fortsætter rekursivt, når vi står i en knude, der maksimalt kan have enten en, to eller tre stier under sig. Der er følgende tre procedurer, som alle eksemplificeres på figur 5.5. Tegningerne viser de forskellige

muligheder, når vi er nået til en knude v.

Maksimalt en sti (`max1`)**.** Som argument fås et sti-objekt, der indeholder de knuder, der er over v i den sti, vi er ved at konstruere. Vi sætter v ind i stien. Hvis der en kant i S, der udgår fra v, kalder vi rekursivt med samme procedure, `max1`, på knuden w, der ligger under under v (figur 5.5 (A)). Er der ingen kant slutter rekursionen her, og hele stien er fundet.

Maksimalt to stier (`max2`)**.** Her indsættes v i begge de sti-objekter, der gives som argument. Hvis der er en enkelt kant i S, der udgår fra v fortættes rekursionen med `max2`-proceduren fra knuden, w, som kanten peger på – stien kan senere deles i to (figur 5.5 (B)). En anden mulighed er, at der er to kanter, med to endeknuder, w_1 og w_2 (figur 5.5 (C)). Er der to kanter, laver vi to rekursive kald på hhv. w_1 og w_2 til `max1`-proceduren – da stien nu deles, kan der ikke ske flere opdelinger. Hvert af disse kald får som argument et af de to sti-objekter. Er der ingen kanter slutter rekursionen her. I dette tilfælde sørger vi for, at kun det ene sti-objekt gemmes, da der kun blev en sti.

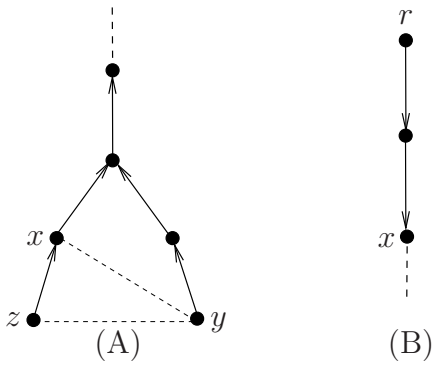
Maksimalt tre stier. Vi indsætter v i de tre sti-objekter, som er givet. Er der kun en kant fra v i S kaldes samme procedure rekursivt på knuden, w, der ligger for enden af denne kant (figur 5.5 (D)). Bemærk, at stien senere kan deles i to eller tre stier. Hvis der er to kanter, hhv. med endepunkt w_1 og w_2, kan vi som nævnt ikke vide hvilken af dem, om nogen, der fører til to stier længere nede i grafen. Derfor laver vi først et rekursivt kald på w_1 med `max2`-proceduren, da denne gren højst kan deles i to. Kaldet gives to af de tre sti-objekter, som vi er givet. Når kaldet returnerer, testes om kaldet resulterede i en eller to stier. Hvis resultatet var én sti, anvender vi `max2`-proceduren igen på w_2 og sender to sti-objekter med. Resulterede første kald i to stier, kan den anden kant kun give anledning til en sti, og derfor anvender vi da `max1`-proceduren på w_2 (figur 5.5 (E)). Er der i stedet tre kanter i S, der udgår fra v, laver vi tre kald til max1-proceduren, der hver får et sti-objekt som argument (figur 5.5 (F)). Er der ingen kanter fra v i S, slutter rekursionen her. Vi sørger også her for kun at gemme ét sti-objekt, da stien ikke blev delt.

Alle knuder, der besøges under rekursionen farves med en anden farve end den hele S er farvet med, for at knuderne senere kan kendes fra resten af S. Når rekursionen slutter, er alle stier fra roden fundet. Herefter mangler vi de stier, der starter i hhv. x, y og z.

Stierne fra roden kan findes i lineær tid, da vi laver et gennemløb af en del af separatoren, hvor vi bruger konstant tid i hver knude.

Stier fra x, y og z

Vi konstruerer nu de stier i S, der udgår fra hhv. x, y og z. Vi bemærker at i en graf, hvor separatorstierne er udgående fra roden, er stierne også udgående fra x, y og z. Igen er tilfældet hvor grafens lag er modsat helt analogt.



Figur 5.6: Situationer, hvor der *ikke* laves stier fra en knude (i begge tilfælde knuden x).

Før vi konstruerer stierne fra x, y og z, afgør vi, om der er en eller flere knuder, der *ikke* skal laves stier fra. Hvis der ikke laves stier fra en knude, har det en af følgende årsager, som eksemplificeres på figur 5.6.

Knuden ligger på en anden rodsti. Det kan ske, at en af knuderne, f.eks. x ligger på en af stierne mod roden fra en af de andre knuder, f.eks. z, se figur 5.6 (A). Alle stier, der har udgangspunkt x, er delstier af de stier, der har udgangspunkt i z. Derfor konstruerer vi i dette tilfælde ikke stier, der har udgangspunkt i x.

Roden når knuden. Når vi laver stierne fra roden, farver vi som nævnt de besøgte knuder. Hvis en af knuderne, f.eks. x, er blevet farvet, betyder det, at den kan nås fra roden, og at den sti, der er mellem x og roden kun er orienteret i én retning, se figur 5.6 (B). Der kan derfor ikke være en sti, der er rettet fra x mod roden.

Knuden er roden. Hvis x, y eller z er roden, er der allerede lavet stier fra den, da vi starter med at lave stier fra roden.

Når det er afgjort hvilke af x, y og z, der skal laves stier fra, laves stierne fra en knude ad gangen. Vi så tidligere, at der højst kan være to stier fra hver af disse knuder. Stierne findes ved at lave to sti-objekter og bruger to gensidigt rekursive procedurer, der virker som `max1` og `max2`, som vi anvendte til at finde stier fra roden.

Efterarbejde med stierne

Når alle stier er fundet, skal vi gennemføre følgende skridt for hver sti:

1. Nogle af stierne vender omvendt i forhold til deres orientering. Dette sker fordi graferne ikke er opbygget ens mht. lag – hvis første lag vender mod roden, finder vi stierne baglæns i forhold til deres orientering, da vi jo starter i roden (tilsvarende for stier vi finder med udgangspunkt i x, y og z). Vi vender derfor alle stier, så de følger deres orientering.

2. Vi laver en indeksering (nummerering: 1, 2, 3, ...) af knuderne på hver sti. Indekseringen gælder kun for den sti. En knude, der er med i flere stier, har et indeks tilknyttet for hver af disse stier.

3. Når en knude er med i separatoren S, betyder det, at knuden ikke er med i flere rekursive kald af opdelingsalgoritmen, da den ikke optræder i en af de delgrafer, der konstrueres. Dette er derfor knudens *final call* og vi sætter knuden til at referere til den aktuelle knude i rekursionstræet.

Hver sti har et nummer, og i den aktuelle knude i rekursionstræet gemmer vi nummeret på den sidste separatorsti, der kan findes på dette sted i rekursionstræet, hvilket er beskrevet i afsnit 5.1.3 og illustreret på figur 5.2. I kapitel 8 redegøres for, hvorledes de tilknyttede informationer anvendes til at afgøre, om en knude kan nå en anden.

5.2.3 Udførelsestid for vores implementation

Vi opsummerer tidsforbruget for de forskellige skridt. For at forberede H anvender vi flere algoritmer fra LEDA, der alle tager tid $\mathcal{O}(|\mathcal{V}(H)|)$ samt et konstant antal lineære gennemløb af grafen for at tilføje information til knuder og kanter. Denne del af implementationen klares derfor i lineær tid. Vi beskriver kapitel 6 hvorledes opdelingsfladen kan findes i lineær tid. For at finde separatoren og de orienterede stier, den indeholder, laver vi igen et konstant antal lineære gennemløb af grafen hvorfor også denne del af implementationen har udførelsestid $\mathcal{O}(|\mathcal{V}(H)|)$. I kapitel 7 beskriver vi, hvorledes selve opdelingen i delgrafer kan foretages i lineær tid. Vi kan dermed konkludere, at denne del af vores implementation foregår i tid $\mathcal{O}(|\mathcal{V}(H)|)$.

5.2.4 Placering i programmet

Den del af programmet, der finder separatoren og de orienterede stier i denne findes i `reachability/separate.cpp`. Vi sætter labels på knuder og kanter i `reachability/label.cpp`.

Kapitel 6

Lokalisering af trekant til separation

I dette kapitel redegøres for, at der givet en orienteret, trianguleret, tolags-graf H, der er indlejret på en sfære, eksisterer en flade, hvor de tre knuder, der beskriver fladen giver anledning til en opdeling af grafen. I kapitel 5.1.4 er beskrevet hvorledes en graf opdeles i tre dele af en flade og rodstierne fra knuderne der afgrænser fladen. I kapitel 5.1.3 beskrives, at vi ønsker, at hver del af grafen højst indeholder halvdelen af grafens knuder. Vi beskriver endvidere i dette kapitel, hvorledes en passende flade kan findes i lineær tid og hvordan vi i vores implementation har løst forskellige problemer og specialtilfælde, mens vi har bevaret den lineære udførselstid.

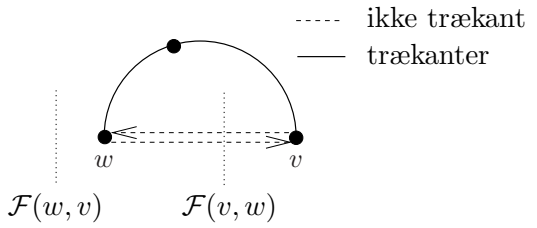
Vores implementering af, hvorledes en trekant findes tager udgangspunkt i en algoritme skitseret i [LT79]. Vi beskriver i afsnit 6.2.1 denne algoritme og i afsnit 6.2.3 forklarer vi, hvorfor algoritmen ikke umiddelbart kan implementeres. I dette afsnit beskrives endvidere, hvorledes vi har udfyldt de manglende dele af algoritmen således at den kan realiseres.

6.1 Eksistens af trekant

Grafen H har et tolags-udspændenede træ T med en rod r. Kanterne i H har, som beskrevet i afsnit 5.2.1, en type og dermed kan vi afgøre, om en given kant er en del af det udspændende træ. En kant, der er en del af det udspændende træ kaldes som bekendt en *trækant*. Enhver trækant har en reference til dens forælderkant og det er således muligt for enhver knude i H at finde vejen i træet op til roden.

Givet en ikke-trækant $(v, w) \in H$ vil en delmængde af kanterne i træet og (v, w) danne en cykel, denne kaldes den *fundamentale cykel* i forhold til kanten (v, w). En fundamental cykel deler en graf i to dele, se figur 6.1. Kald den mængde, der er til højre for en kant (v, w) for $\mathcal{F}(v, w)$. Den nærmeste fælles forfader for v og w ($nca(v, w)$) vil være en del af den fundamentale cykel for kanten (v, w).

Lad v være en knude i træet. Vi erindrer, at stien mellem roden og v kaldes en *rodsti* og betegnes $T(v)$. Mængden af knuder på stien er $\mathcal{V}(T(v))$. Vi vil først give et konstruktivt bevis for, at der eksisterer en trekant, der har de ønskede egenskaber og i sætning 6.10 beviser vi, at en trekant kan findes i lineær tid.



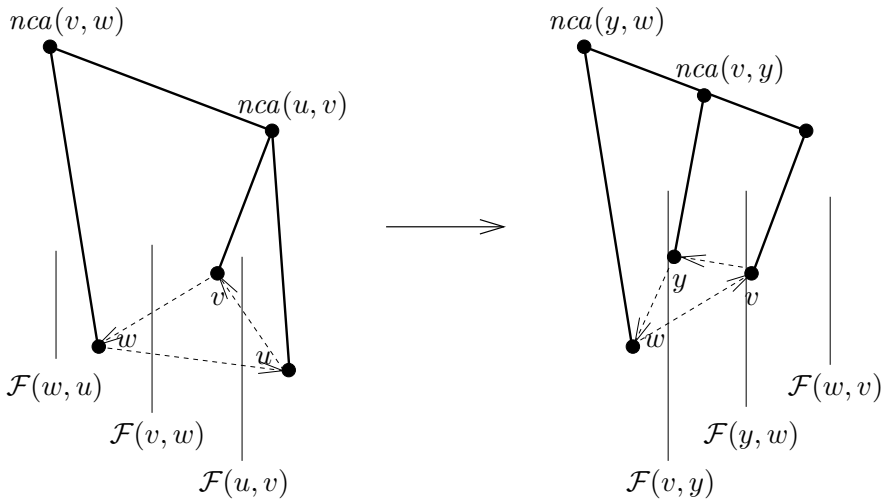
Figur 6.1: En kant definerer sammen med det udspændende træ to mængder af grafen.

Sætning 6.1 *Givet en trianguleret graf H, med et udspændende træ T og en rod, er det muligt at finde en flade (u,v,w) således at antallet af knuder i hver af komponenterne i $H \setminus \mathcal{V}(T(u) \cup T(v) \cup T(w))$ ikke er større end halvdelen af knuderne i grafen H.*

Bevis. Opfat grafen som om den er indlejret på en sfære. Da H er orienteret, triangueleret og indlejret på en sfære er en flade omkranset af tre kanter, endvidere eksisterer der for enhver kant i H en kant med samme endepunkter, men med modsat orientering. Givet en trekant (u,v,w) vil hver af de tre komponenter i $H \setminus \mathcal{V}(T(u) \cup T(v) \cup T(w))$ svare til en del, der er afgrænset af en fundamental cykel i forhold til en af kanterne (u,v), (v,w) og (w,u). På figur 6.2 kan de navngivne komponenter ses.

Vælg en vilkårlig trekant defineret ved (u,v,w) og beregn størrelsen af de tre komponenter. Hvis alle disse er små nok er vi færdige, ellers fortsættes. Kun en af komponenterne kan indeholde mere end halvdelen af knuderne. Antag, at kanten (v,w) definerer den mængde, der indeholder for mange knuder og lokaliser trekanten (v,w,y) inden i cyklen, se figur 6.2. Dvs. find knuden y, der er nabo til v og w og befinder sig i mængden, der er for stor. Beregn størrelsen af de tre nye komponenter, der defineres af den nye trekant. Gentag indtil en passende trekant er fundet, dvs. en trekant som definerer tre komponenter, der hver højst indeholder halvdelen af grafens knuder.

Vi mangler nu at bevise at processen terminerer. Se på trekanten (u,v,w) og antag at mængden $\mathcal{F}(v,w)$ indeholder for mange knuder; de to resterende mængder indeholder da tilsammen mindre end halvdelen af knuderne i H, se figur 6.2. Ved at lokalisere trekanten (v,y,w) i den største mængde og kigge på de nye komponenter ses, at to af disse dannes ved at $\mathcal{F}(v,w)$ deles op i to. Den sidste mængde er de to små mængder, $\mathcal{F}(u,v)$ og $\mathcal{F}(w,u)$, slået sammen og knuden u tilføjet. Hvis de tre nye mængder alle er mindre end $\mathcal{F}(v,w)$, betyder det, at størrelsen af den største mængde aftager hver gang en ny trekant lokaliseres og processen vil terminere. Mængderne $\mathcal{F}(v,y)$ og $\mathcal{F}(y,w)$ er oplagt mindre end $\mathcal{F}(v,w)$, da de er fremkommet ved en disjunkt opdeling at denne mængde. Nedenstående beregning giver os at $\mathcal{F}(w,v)$ ligeledes er mindre end



Figur 6.2: Givet trekanten (u, v, w) lokaliseres trekanten (w, v, y), der er indeholdt i den største delgraf.

$\mathcal{F}(v, w)$:

$$
\begin{aligned}
&|\mathcal{F}(v, w) > |\mathcal{V}(H)|/2 \\
&\Downarrow \\
&|\mathcal{F}(w, u) \cup \mathcal{F}(u, v)| \leq |\mathcal{V}(H)|/2 \\
&\Downarrow \\
&|\mathcal{F}(w, v)| \leq |\mathcal{V}(H)|/2
\end{aligned}
$$

Vi har nu at størrelsen af den største komponent aftager hver gang vi lokaliserer en ny trekant og processen vil terminere. □

6.2 Metode til lokalisering af en trekant

I de følgende afsnit vil vi beskrive hvordan vi finder en trekant i lineær tid. Først vil vi overordnet beskrive vores algoritme og derefter vil vi i detalje beskrive de enkelte dele og hvilke specialtilfælde der eksisterer. Til sidst vil vi argumentere for tidsforbruget.

Vores algoritme tager udgangspunkt i en algoritme beskrevet i [LT79] kapitel 3, skridt 8 og 9 i hvilken forfatterne opfordrer læseren til selv at udfylde de manglende detaljer. Vi erfarede, at det ikke blot var detaljer der manglede, men derimod nogle væsentlige dele for at muliggøre implementation af algoritmen. De ting vi ændrer i algoritmen gør, at der skal laves nogle modifikationer i beviset for, at trekanten kan findes i lineær tid.

Vi vil starte med at beskrive den overordnede struktur af den foreslåede algoritme og derefter vil vi i gå i detaljer med de enkelte dele, herunder kommer vi ind på hvilke dele af algoritmen vi selv har konstrueret. Vi vil beskrive hvorfor det har været nødvendigt at ændre på dele af algoritmen. Vi vil afslutte kapitlet med at bevise at trekanten kan findes i lineær tid.

6.2.1 Algoritmen

I artiklen [Tho01] henvises udelukkende til [LT79] for hvordan en trekant findes i lineær tid. Algoritmen for at finde en trekant, der opfylder kravene i sætning 6.1 er overordnet beskrevet i [LT79] kapitel 3, skridt 8 og 9 og er som følger

1. Vælg en vilkårlig kant (v, w), der ikke er en del af det udspændende træ.
2. Beregn størrelsen af de to mængder afgrænset af den fundamentale cykel i forhold til (v, w).
3. Lokaliser knuden y, der er nabo til v og w og ligger i den største af de to mængder.
4. Følg forældrereferencerne fra v, y, og w. Når stien fra y møder en sti fra v eller w kan størrelsen af de tre komponenter beregnes.
5. Stop hvis størrelsen af de tre komponenter alle er mindre end halvdelen af knuderne i grafen. Ellers fortsættes ved at lokalisere en trekant, der indeholder en tvillingekant til den gamle trekant, og er inde i den mængde, der er for stor.

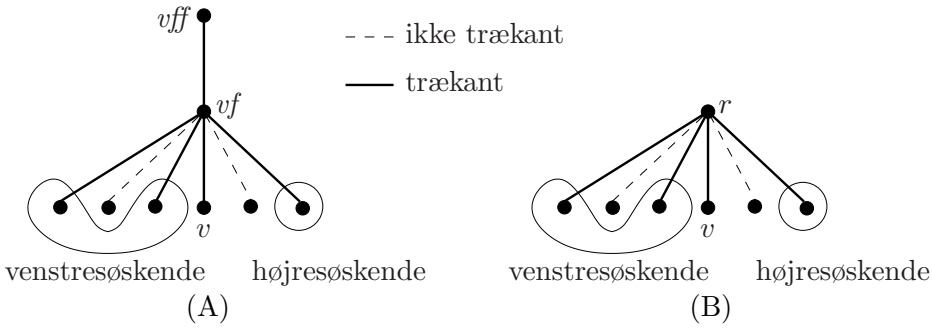
Algoritmen er umiddelbart en simpel rekursiv algoritme, der starter med en tilfældig kant og lokaliserer flader, der er indeholder en given kant, indtil en passende trekant findes. Fra beviset for sætning 6.1 følger, at ovenstående algoritme terminerer og en trekant findes. Da vi skulle implementere algoritmen erfarede vi, at der er nogle væsentlige tilfælde, der ikke er beskrevet i artiklen. I de følgende afsnit vil vi i detalje gennemgå de enkelte dele af algoritmen og beskrive, hvor der er afvigelser fra hvad der er beskrevet i [LT79]. Hvert af de følgende afsnit svarer til et punkt i den netop skitserede algoritme.

6.2.2 Udvælg en vilkårlig kant

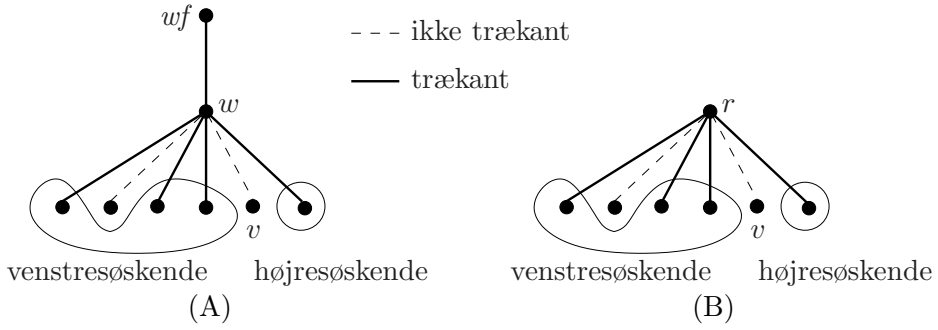
I LEDAs grafklasse eksisterer en funktion der i konstant tid kan returnere en vilkårlig kant og denne benytter vi. Hvis den valgte kant er en trækant skal vi vælge en anden. Da grafen er trianguleret er en flade defineret af tre kanter. Mindst én af disse er ikke en trækant, vi kan derfor ved at kigge på den forrige og den efterfølgende kant i forhold til fladen og vælge en der ikke er en trækant. Det er muligt i konstant tid at få information om den næste/forrige kant i forhold til en flade.

6.2.3 Størrelsen af mængder defineret ved en fundamental cykel

Vi har fundet en kant (v, w) og ønsker at beregne størrelsen af de to mængder $\mathcal{F}(v, w)$ og $\mathcal{F}(w, v)$, der afgrænses af den fundamentale cykel i forhold til kanten (v, w). I [Tho01] henvises til [LT79] kapitel 3, skridt 8 og 9 for hvordan en trekant findes i lineær tid og herunder beskrives hvordan størrelsen af de to mængder findes. Først vil vi definere nogle begreber. Vi erindrer, at knuder i grafen har to kantlister; én med udkanter og én med indkanter. Rækkefølgen af kanterne i listerne er angivet mod urets retning.



Figur 6.3: Venstre- og højresøskende for en knude v.



Figur 6.4: Venstre- og højresøskende for en kant (v, w).

Definition 6.2 *Lad v være en knude i grafen således at vf er forælder til v i forhold til det udspændende træ og vff er forælder til vf, se figur 6.3 (A).*

En venstresøskende *for en knude v, er en knude k som er barn af vf. Kanten (vf, k), er i vf's liste af udkanter placeret efter kanten (vf, vff) og før kanten (vf, v). En* højresøskende *for en knude er defineret tilsvarende.*

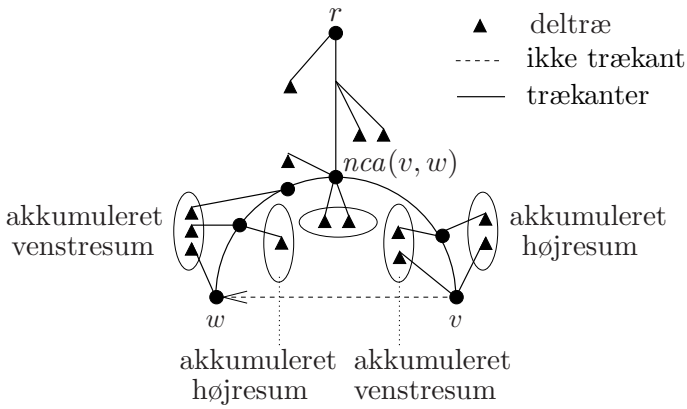
En venstre- *og* højresøskende *for en knude hvis forælder er en rod er defineret analogt, se figur 6.3 (B). Den første kant, er den kant der er først i rodens liste af udkanter.*

Definition 6.3 *Lad (v, w) være en kant i grafen således at wf er forælder til w i forhold til det udspændende træ, se figur 6.4 (A).*

En venstresøskende *for en kant (v, w), er en knude k som er barn af w. Kanten fra w til k, er i w's liste af udkanter placeret efter kanten (w, wf) og før kanten (w, v). En* højresøskende *for en kant er defineret tilsvarende.*

En venstre- *og* højresøskende *for en kant hvis ene endepunkt er en rod er defineret analogt, se figur 6.4 (B).*

Definition 6.4 *En* venstresum *for en knude v, defineres som antallet af knuder i de deltræer, der som rod har en en venstresøskende af v. Højresummen for en knude og summerne for en kant defineres tilsvarende.*



Figur 6.5: Kanten (v, w) og dens fundamentale cykel. De akkumulerede højre- og venstresummer for stierne fra v hhv. w til $nca(v, w)$ er indtegnet.

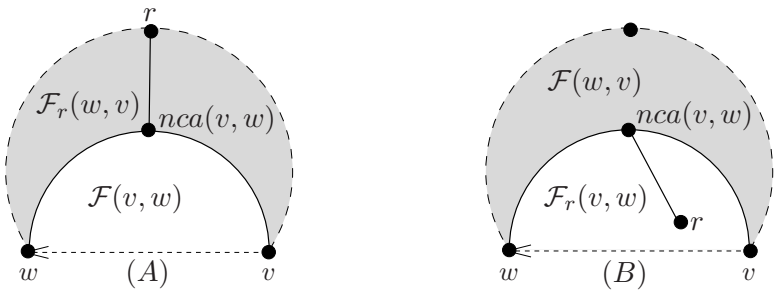
Definition 6.5 *Den* akkumulerede venstresum *for en sti er summen af venstresummer for knuder på stien.*

På figur 6.5 ses de akkumulerede summer for stierne, der begynder med kanten (v, w) hhv. (w, v) og slutter umiddelbart før $nca(v, w)$.

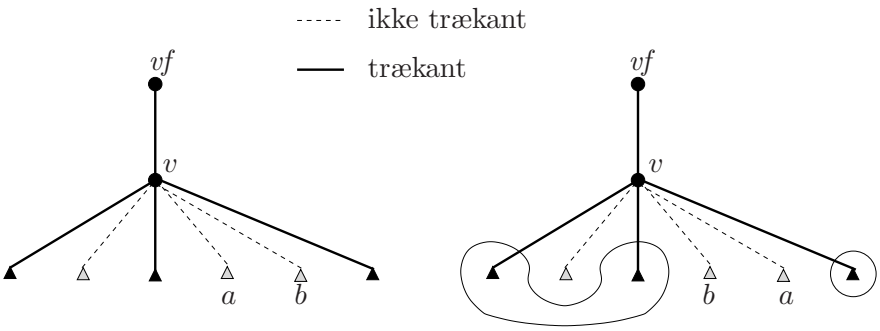
For at beregne størrelsen af mængderne, $\mathcal{F}(v, w)$ og $\mathcal{F}(w, v)$, afgrænset af den fundamentale cykel mht. kanten (v, w), følger vi forældrereferencer fra knuderne v og w indtil nærmeste fælles forfader $nca(v, w)$ er fundet. Knuden $nca(v, w)$ findes ved, at vi skiftevis går et skridt fra v og fra w, farver de knuder vi ser og husker på hvilken kant en knude er blevet farvet fra. Undervejs beregnes de akkumulerede venstre- og højresummer og værdierne gemmes i de besøgte knuder. Når vi finder $nca(v, w)$ kan vi beregne størrelsen af de to mængder. Først skal det undersøges i hvilken mængde knuderne fra $nca(v, w)$ og op til roden befinder sig i, se figur 6.6. Dette afgøres i konstant tid ved at se på rækkefølgen af de tre kanter, der går fra $nca(v, w)$ mod w, mod v og mod roden. Antag at mængden $\mathcal{F}(v, w)$ indeholder roden og kald den for $\mathcal{F}_r(v, w)$, se figur 6.6 (B). Vi beregner størrelsen af mængden der ikke indeholder roden, $\mathcal{F}(w, v)$, ved at addere to akkumulerede summer og huske at medregne relevante undertræer af $nca(v, w)$, se figur 6.5. Vi kan beregne størrelsen af $\mathcal{F}_r(v, w)$ idet vi kender antallet af knuder i grafen og længden af stierne fra v hhv. w til $nca(v, w)$ i det udspændende træ T. Vi betegner antallet af kanter mellem to knuder v og w i T med $\lambda(v, w)$ og størrelsen af mængden, der indeholder roden beregnes som følger:

$$|\mathcal{F}_r(v, w)| = |\mathcal{V}(G)| - |\mathcal{F}(w, v)| - \lambda(w, nca(v, w)) - \lambda(v, nca(v, w)) - 1$$

Vi ønsker, at tiden for at beregne størrelsen af mængderne $\mathcal{F}_r(v, w)$ og $\mathcal{F}(w, v)$ er proportional med summen af længden af vejene fra v hhv. w til $nca(v, w)$. Når vi beregner størrelsen af $\mathcal{F}(v, w)$ skal vi huske at medregne undertræer af $nca(v, w)$, der er i mængden, se figur 6.5. Vi vil gerne kunne beregne størrelsen af disse undertræer i konstant tid. Det kræver, at vi for en given knude



Figur 6.6: Roden af det udspændende træ befinder sig i $\mathcal{F}(v,w)$ eller $\mathcal{F}(w,v)$.



Figur 6.7: Mængden $\mathcal{H}_v(a,b)$ er markeret.

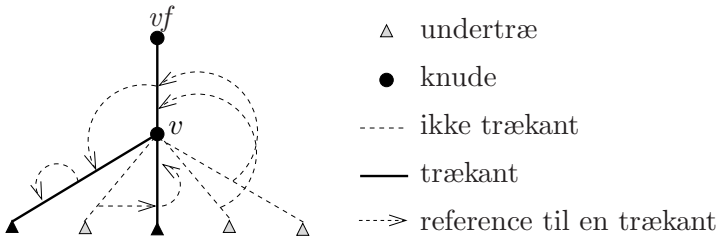
v og kanter (v,a), (v,b) i konstant tid kan beregne antallet af knuder i undertræer af v der ligger mellem kanterne (v,a) og (v,b), se figur 6.7. En kant (v,c) ligger mellem kanterne (v,a) og (v,b) hvis den i v's liste af udkanter er placeret efter (v,a) og før (v,b). For en graf H, en knude v og kanter (v,a) og (v,b) kalder vi mængden af knuder i deltræer mellem (v,a) og (v,b) for $\mathcal{H}_v(a,b)$. Når vi beregner akkumulerede summer ønsker vi ligeledes at kunne beregne mængden $\mathcal{H}_v(a,b)$ i konstant tid.

I afsnit 5.2.1 beskrives hvilken information, der tilknyttes knuder og kanter i grafen H. Vi vil kort opremse den information vi bruger i de følgende beregninger. For en knude v bruger vi, at den indeholder:

- en liste med dens udkanter – rækkefølgen af kanter er i overensstemmelse med en planar repræsentation af grafen.
- en liste med dens børn – rækkefølgen er identisk med knudens liste af udkanter.
- en liste med prefixsummerne af antallet af knuder i de undertræer som hvert af dens børn er rod i.
- et indeks, der svarer til hvor v står i sin forælders liste af børn.

Vi bruger ligeledes en del af den infomation som en kant $e = (v,w)$ indeholder:

- et indeks, der svarer til hvor e står i v's liste af udkanter.



Figur 6.8: Referencer til trækanter.

- hvilken type kant den er

Vi bruger endvidere, at en ikke-trækant har en reference til den efterfølgende trækant i kildeknudens udkantliste og en trækant har en referencer til sig selv, se figur 6.8. Kanten fra knuden og til dens forælder har en reference til den næste trækant. Denne konstruktion giver anledning til følgende faktum og vi skitserer hvorledes i det følgende.

Faktum 6.6 *Givet en graf H med et udspændende træ, en knude v og to kanter (v, a) og (v, b) er det muligt i konstant tid at beregne størrelsen af mængden $\mathcal{H}_v(a, b)$ der indeholder knuderne i undertræerne af v mellem kanterne (v, a) og (v, b).*

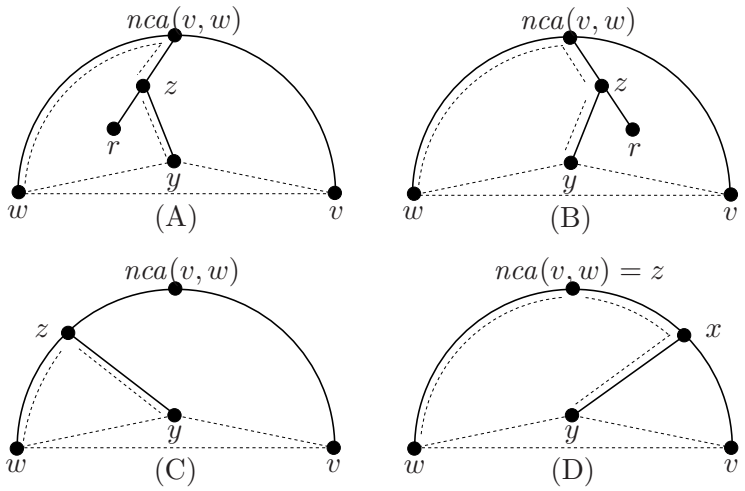
For kanterne (v, a) og (v, b) finder vi de trækanter, evt. kanterne selv og ikke forælderkanten, der ligger umiddelbart efter de to kanter. Knuderne, som disse kanter peger på, kender deres nummer i v's liste af børn. Da vi har prefixsummerne af antallet af knuder i undertræerne kan vi i konstant tid beregne antallet af knuder mellem de fundne kanter. Algoritmen besværliggøres dog af, at a og b ikke nødvendigvis kommer i den angivne rækkefølge og at listen skal opfattes som en cyklisk liste, se figur 6.7 for et eksempel på dette. Dette ændrer dog ikke på, at summen kan beregnes i konstant tid; vi skal blot undersøge hvordan kanterne (v, a) og (v, b) ligger i forhold til hinanden og om de er trækanter.

Vi har argumenteret for, at beregninger der skal foretages på en knude tager konstant tid. Den samlede tid for beregne størrelsen af mængderne $\mathcal{F}_r(v, w)$ og $\mathcal{F}(w, v)$ må da være lineær i længden af vejene fra v hhv. w til $nca(v, w)$. Dvs. vi har følgende:

$|\mathcal{F}_r(v, w)|$ og $|\mathcal{F}(w, v)|$ kan beregnes i tid $\mathcal{O}(\lambda(v, nca(v, w)) + \lambda(w, nca(v, w)))$

6.2.4 Find en trekant i den største mængde

Vi står med en kant (v, w) og har beregnet på hvilken side af denne, den største mængde er. Vi ønsker nu at finde fladen (v, y, w) i denne mængde. Dette er simpelt, da vi for en kant kan bede om fladen til venstre for kanten og fra en kant kan vi få dens tvilling. Dette kan, som beskrevet i kapitel 2.1.2 ske i konstant tid.



Figur 6.9: Sammenhæng mellem en flade i grafen H og det udspændende træ.

6.2.5 Størrelsen af mængder defineret ved brug af en trekant

Når vi når punkt 4 i algoritmen står vi med en trekant (v, y, w). Vi kender størrelsen af mængden $\mathcal{F}(w, v)$, defineret af den fundamentale cykel i forhold til kanten (w, v), idet denne er beregnet i punkt 2. Vi ønsker at beregne størrelsen af mængderne $\mathcal{F}(v, y)$ og $\mathcal{F}(y, w)$, se igen figur 6.2.

Ifølge [LT79] kapitel 3, skridt 9 skal vi fra y følge forældrereferencer til vi rammer (w, v)-cyklen i knuden kaldet z, se figur 6.9 (C). Herefter kan vi bruge samme princip som beskrevet i 6.2.3 til at finde størrelsen af den ene mængde og vi kan beregne størrelsen af den anden mængde hvis vi kender afstanden fra y til z. Dette er dog ikke altid muligt; stien fra $nca(v, w)$ til roden kan befinde sig inden i mængden vi ønsker at beregne størrelsen af og når vi følger forældrereferencer fra y kan vi møde en knude på denne sti og vi vil derfor aldrig ramme en knude på (w, v)-cyklen, se figur 6.9 (A) og (B). Det er i artiklen [LT79] ikke beskrevet hvordan dette tilfælde skal håndteres. For at løse problemet har vi redefineret z på følgende måde: Knuden z defineres til at være punktet hvor stien fra y først møder en sti fra v eller w. På figur 6.9 ses fire forskellige illustrationer hvor knuden z er fundet. Vi benytter følgende fremgangsmåde for at beregne størrelsen af de ønskede mængder::

1. Gå et skridt fra skiftevis v, w og y. For knuderne v hhv. w beregnes den akkumulerede venstre- hhv. højresum og for y beregnes både højre- og venstresum. Disse summer gemmes i knuderne på stierne.

2. Hvis stien fra v eller w går ud af cyklen (ved $nca(v, w)$, se figur 6.9 (A) og (B)) skal dybden af knuden $nca(v, w)$ gemmes, således at afstanden mellem $nca(v, w)$ og z kan beregnes. Hvis stien fra v hhv. w når $nca(v, w)$ markeres dette, hvis den anden sti efterfølgende når til $nca(v, w)$ stopper vi med at følge forældrereferencer fra denne knude.

3. Stop når stierne fra v og y eller w og y mødes. Det er nu muligt at

beregne størrelsen af mængderne $\mathcal{F}(v,y)$ og $\mathcal{F}(y,w)$, hvilket beskrives i nedenstående afsnit.

Vi går skiftevis et skridt fra hver af knuderne v, w og y. For at kunne afgøre hvilke stier der mødes farves hver sti med en farve, og samtidig markeres hvilken kant en knude er blevet farvet fra. For hvert skridt afgøres om andre før har været ved denne knude, hvis ikke farves knuden, de relevante summer beregnes og resultatet gemmes i knuden. Hvis stierne fra v og y eller w og y mødes kan vi beregne størrelsen af mængderne.

Vi bruger tre forskellige beregningsmetoder alt efter hvilken situation vi befinder os i. De tre metoder vi nu vil gennemgå, løser problemerne, der opstår i forbindelse med de fire situationer der er illustreret på figur 6.9. Situationen skitseret på billede (D) løses som i (A) eller (B). Situationerne er dækkende for, hvad der kan ske, når man følger forældrereferencer fra knuderne v, w og y; enten møder stien fra y cyklen eller også gør den ikke. Hvis stien ikke når cyklen, se figur 6.9 (A) og (B), kan roden være til højre eller til venstre for stien fra y. Hvis stien fra y rammer cyklen, vil dette enten ske mellem knuderne v hhv w og $nca(v,w)$ eller i $nca(v,w)$, se billede (C) og (D).

Vi antager i den følgende gennemgang, at det er stierne fra w og y der mødes. I tilfældene (A) og (B) betyder det, at $\lambda(w, nca(v,w)) \leq \lambda(v, nca(v,w))$. I tilfælde (C), at $\lambda(w,z)+\lambda(y,z) \leq \lambda(y, nca(v,w))+\lambda(v, nca(v,w))$ og i tilfælde (D) at $\lambda(w, nca(v,w)) + \lambda(y, nca(v,w)) \leq \lambda(y,x) + \lambda(v,x)$. Det vil sige, at vi altid går den korteste mulige vej rundt om en af de mængder vi ønsker at beregne størrelsen af. Tilfældet hvor det er stierne fra v og y der mødes er analogt.

Se figur 6.9 (A). Antag at stierne fra w og y mødes i knuden z og at stien fra w har passeret $nca(v,w)$. Først skal vi afgøre om roden befinder sig i mængden $\mathcal{F}(y,w)$. Dette afgøres i konstant tid, ved fra z at se på rækkefølgen af kanterne mod y, mod v og mod roden. Antag at roden befinder sig i $\mathcal{F}(y,w)$. Vi har beregnet den akkumulerede højresum for stien fra w til z og den akkumulerede venstresum fra y til z. Faktum 6.6 giver os, at tiden for at beregne de akkumulerede summer er lineær i længden af stierne som beregningerne foretages på. Vi mangler at medregne mængden af knuder mellem z og roden og antallet af knuder i undertræer af z, der befinder sig i $\mathcal{F}(y,w)$. Enhver knude kender antallet af knuder i det træ det selv er rod i. Vi har referencer til roden og til z og kan derfor beregne antallet af knuder mellem z og roden i konstant tid. Fra faktum 6.6 har vi, at antallet knuder der befinder sig i undertræer af z mellem stien mod w og mod y ligeledes kan beregnes i konstant tid. Samlet har vi nu, at udførselstiden for at beregne størrelsen af mængden $\mathcal{F}(y,w)$ bliver lineær i antallet af knuder vi besøger. Vi kan beregne $|\mathcal{F}(v,y)|$ ved at bruge følgende beregningsmetode:

Formel 6.7 $|\mathcal{F}(v,y)| = |\mathcal{F}(v,w)| - |\mathcal{F}(y,w)| - \lambda(y,z) - \lambda(nca(v,w),z)$

Det tager oplagt konstant tid at beregne størrelsen af $\mathcal{F}(v,y)$ blot vi har beregnet $|\mathcal{F}(y,w)|$ og vi får da:

$$|\mathcal{F}(v,y)| \text{ og } |\mathcal{F}(y,w)| \text{ kan beregnes i tid } \mathcal{O}(\lambda(w,z) + \lambda(y,z))$$

Antag at stierne fra w og y mødes i knuden z, at stien fra w har passeret $nca(v, w)$ og at roden ikke befinder sig i $\mathcal{F}(y, w)$, se figur 6.9 (B). Som i det ovenstående, kan størrelsen af $\mathcal{F}(y, w)$ beregnes ud fra de akkumulerede summer for stierne fra w hhv. y til z og størrelsen af undertræerne af z, der befinder sig i $\mathcal{F}(y, w)$. Vi ved dette tager tid lineært i antallet af besøgte knuder. Vi bruger formel 6.7 til at beregne størrelsen af $\mathcal{F}(v, y)$ og får igen:

$$|\mathcal{F}(v, y)| \text{ og } |\mathcal{F}(y, w)| \text{ kan beregnes i tid } \mathcal{O}(\lambda(w, z) + \lambda(y, z))$$

Se figur 6.9 (C). Antag at stierne fra w og y mødes og at stien fra w ikke har passeret $nca(v, w)$. Stien fra $nca(v, w)$ til roden kan ikke befinde sig i mængden $\mathcal{F}(y, w)$, da $nca(v, w)$ ikke er på randen af mængden. Størrelsen af $\mathcal{F}(y, w)$ beregnes ud fra den akkumulerede højresum på stien fra w til z, venstresummen på stien fra y til z og antallet af knuder i undertræer af z, der befinder sig i mængden $\mathcal{F}(y, w)$. Disse beregninger kan foretages i lineær tid i antallet af knuder vi besøger. Når vi har beregnet størrelsen af $\mathcal{F}(y, w)$ og kender størrelsen af mængden $\mathcal{F}(v, w)$ og afstanden fra y til z kan vi bruge følgende simple formel til at beregne størrelsen af $\mathcal{F}(v, y)$.

Formel 6.8 $|\mathcal{F}(v, y)| = |\mathcal{F}(v, w)| - |\mathcal{F}(y, w)| - \lambda(y, z)$

Disse beregninger tager konstant tid og samlet får vi endnu engang at:

$$|\mathcal{F}(v, y)| \text{ og } |\mathcal{F}(y, w)| \text{ kan beregnes i tid } \mathcal{O}(\lambda(w, z) + \lambda(y, z))$$

Antag at stien fra w mødes med stien fra y i $nca(v, w)$, se figur 6.9 (D). Først skal vi afgøre om roden r befinder sig i $\mathcal{F}(y, w)$. Hvis r befinder sig i $\mathcal{F}(y, w)$, beregnes størrelsen af mængden som i situation (A) ellers bruger vi samme beregningsmetode som i situation (B). Udførelsestiden bliver som i de ovenstående tilfælde.

Vi har nu argumenteret for, at tiden for at beregne størrelsen af mængderne $\mathcal{F}(v, y)$ og $\mathcal{F}(y, w)$ er proportional med antallet af knuder der besøges. Vi erindrer, at dette er det samme som den korteste vej rundt om en af mængderne, fraregnet kanten (v, w). Vi nedskriver dette som et korollar:

Korollar 6.9 $|\mathcal{F}(v, y)|$ *og* $|\mathcal{F}(y, w)|$ *kan beregnes i tid* $\mathcal{O}(min(\lambda(w, y), \lambda(v, y)))$

6.2.6 Iteration

Vi har i punkt 4 beregnet størrelsen af tre komponenter defineret ved brug af en trekant. Hvis de alle indeholder mindre end halvdelen af grafens knuder er vi færdige. Hvis en komponent, f.eks. $\mathcal{F}(v, y)$, indeholder for mange knuder lokaliserer vi i konstant tid en knude i komponenten, der er nabo til v og y. Vi har derved fundet en ny trekant og kan fortsætte algoritmen fra punkt 4. Vi har i sætning 6.1 bevist, at algoritmen terminerer.

6.2.7 Tidsforbrug for at finde en trekant

Vi vil i dette afsnit bevise, at en trekant, der opfylder kravene beskrevet i sætning 6.1 og er konstrueret som beskrevet i afsnit 6.2.1 kan findes i lineær tid. Vi har skrevet vores egen udgave af dette bevis, da der i beviset i [LT79] på side 188 er antaget, at vi er i tilfældet illustreret på figur 6.9 (C). Dvs. at vi ved at følge forældrereferencerne fra y vil ramme (w, v)-cyklen.

Kald antallet af flader i grafen H for $\mathcal{F}(H)$. Da H er sammenhængende og planar har vi fra Eulers sætning antallet af flader er lineært i antallet af knuder.

Sætning 6.10 *Givet en orienteret, trianguleret, planar graf H, med et udspændende træ T og en rod, er det muligt i tid $\mathcal{O}(\mathcal{V}(H))$ at finde en flade (u, v, w) således at antallet af knuder i hver af komponenterne i $H \setminus \mathcal{V}(T(u) \cup T(v) \cup T(w))$ ikke er større end halvdelen af knuderne i grafen H.*

Bevis. Vi har i afsnit 6.2.1 beskrevet algoritmen, der finder en trekant. I det følgende vil vi gennemgå algoritmen punkt for punkt og bevise, at den kan udføres i lineær tid i antallet af flader.

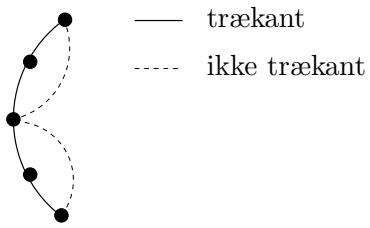
I punkt 1 skal der udvælges en vilkårlig kant i H. Vi har i afsnit 6.2.2 beskrevet hvorledes dette kan udføres i konstant tid.

I punkt 2 skal vi givet en kant (v, w) beregne størrelsen de to mængder, der er afgrænset af den fundamentale cykel i forhold til (v, w). Vi har i afsnit 6.2.3 beskrevet hvorledes dette kan gøres i tid $\mathcal{O}(\lambda(v, nca(v, w)) + \lambda(w, nca(v, w)))$, hvilket oplagt tilhører $\mathcal{O}(\mathcal{F}(H))$.

Punkt 3, hvor der skal findes en knude, der ligger på randen af en flade, kan som beskrevet i afsnit 6.2.4 udføres i konstant tid.

Punkterne 4 og 5 gentages indtil en passende trekant findes. Vi koncentrerer os om punkt 4, da punkt 5 kan udføres i konstant tid. Under punkt 4 skal vi beregne størrelsen af to mængder, se figur 6.9. Vi har i afsnit 6.2.5 beskrevet, hvorledes størrelsen af mængderne findes ved at besøge alle kanter på den korteste vej rundt om mængden med den mindste omkreds. Udførelsestiden er lineær i antallet af kanter der besøges. Når vi har beregnet størrelsen af mængderne laver vi efterfølgende kun beregninger på den største af disse og den mindste mængde ser vi bort fra.

Antag at antallet af kanter, der besøges rundt om mængden med den mindste omkreds er k. Da en flade beskrives af tre kanter, indeholder den mindste mængde mindst $k/2$ flader, se figur 6.10. Dvs. antallet af flader vi efterfølgende ser bort fra er lineært i antallet af kanter, der besøges. Algoritmen terminerer før vi har valgt at se bort fra alle grafens flader og antallet af flader er som tidligere nævnt lineært i antallet af knuder. Vi har nu argumenteret for, at algoritmen kan udføres i lineær tid. □



Figur 6.10: Flader incidente med trækanter

6.2.8 Placering i programmet

Den del af vores implementation, der givet en forberedt tolagsgraf finder en passende flade findes i `reachability/findTriangle.cpp`.

Kapitel 7

Opdeling af tolagsgraf

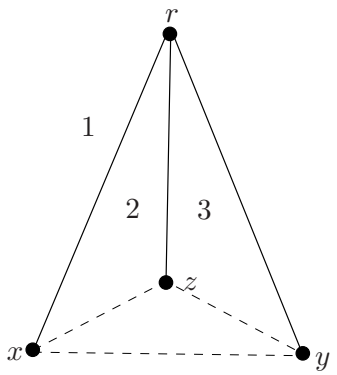
I dette kapitel beskriver vi, hvorledes vi givet en trianguleret tolagsgraf H med et udspændende træ T, samt en flade (x, y, z) opdeler grafen i op til tre delgrafer, der ligeledes er tolagsgrafer. Vi argumenterer for, at denne opdeling kan udføres i lineær tid. Vi har i kapitel 6 vist, hvorledes denne flade findes, således at den definerer en opdeling af H. I denne opdeling deles H i tre delgrafer, hvoraf ingen indeholder mere end $|\mathcal{V}(H)|/2$ knuder.

Vi starter med en kort teoretisk gennemgang af, hvorledes opdelingen af H foretages. Herefter giver vi en detaljeret beskrivelse af vores implementation, hvor vi også kommer ind på specialtilfælde.

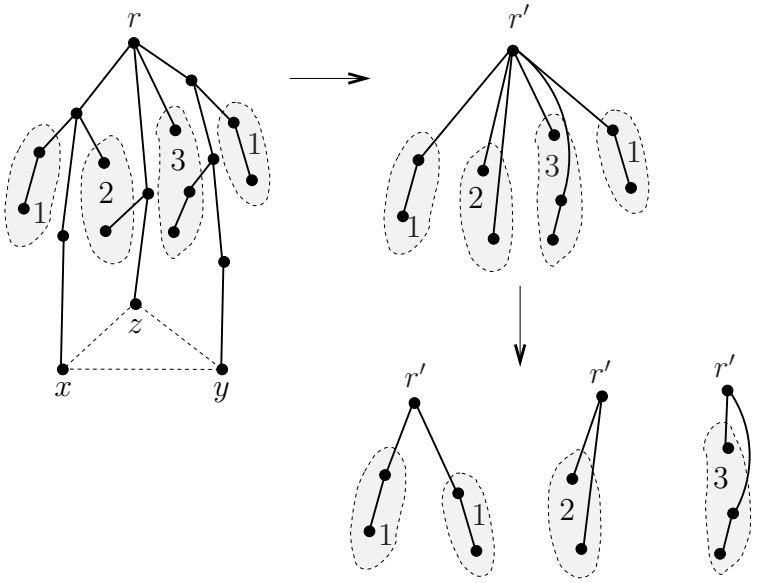
7.1 Opdeling i delgrafer

Vi er givet en trianguleret tolagsgraf H med et udspændende træ T med rod r og trekantet opdelingsflade (x, y, z) der definerer separatoren S, som er en samling rodstier, der definerer en opdeling af H. Rodstierne $T(x)$, $T(y)$, $T(z)$ som udgør S, går mellem roden r og hhv. x, y og z. Rodstierne angiver de dele, vi deler grafen op i. Vi beskriver nu hvorledes grafen deles op.

Fladen og de tre rodstier definerer som nævnt tre dele af grafen, som det kan ses på figur 7.1. Del 1 omkranses af kanten (x, y) og af rodstierne $T(x)$ og



Figur 7.1: Delene, som grafen deles i. Del 1 defineres af (x, y), del 2 af (y, z) og del 3 af (z, x).



Figur 7.2: Eksempel på, at en graf opdeles og bliver til tre nye grafer.

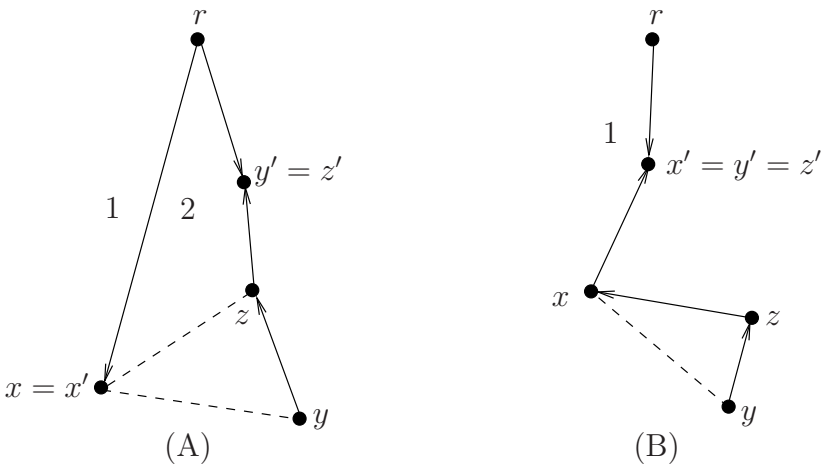
$T(y)$, der enten som her mødes i roden, eller mødes i en knude før roden. De andre dele omkranses ligeledes af en kant i trekanten og to rodstier.

Da separatoren S består af rodstier, er S sammenhængende og vil altid indeholde r. Vi kan nu trække S sammen til én knude, r', og grafen vil stadig være sammenhængende. Derfor kan vi lave tre nye grafer, hvor hver graf består af r', knuderne fra én af de tre dele samt kanter, der har disse knuder som endepunkter. Kanterne, der går til og fra r', er de kanter, som i den oprindelige graf gik til og fra de knuder, som nu er blevet sammentrukket til r'.

Et eksempel på, hvordan en graf opdeles i tre mindre grafer ses på figur 7.2. Her ses først grafen, hvor de tre knuder x, y og z er fundet. Herefter er rodstierne og roden, r, trukket sammen til den nye rod r'. Sidst er grafen delt op i tre mindre grafer.

Hvis en af kanterne i den flade, som grafen opdeles efter, er en trækant, vil der blive en delgraf mindre, da den del af H', der er mellem det udspændende træ og denne kant vil være tom. Der vil således kun være to delgrafer. Hvis to af kanterne er trækanter, vil to dele være tomme og der vil kun være en enkelt delgraf. Dette vender vi tilbage til i afsnit 7.2.1.

Under opdelingen ser vi først på alle knuder i separatoren og de incidente kanter, når separatoren sammentrækkes. Herefter ser vi på alle de tilbageværende knuder og kanter, som hver skal med i en graf. Dog skal r' indsættes i alle tre grafer. Da alle knuder og kanter besøges et konstant antal gange, er udførelsestiden for denne fase $\mathcal{O}(|\mathcal{V}(H)|)$.



Figur 7.3: Hvis en eller to af kanterne i opdelingsfladen er trækanter, vil opdelingen resultere i færre delgrafer. (A) En trækant, to delgrafer. (B) To trækanter, en delgraf.

7.2 Implementation

Vi beskriver i dette afsnit, hvorledes vi har implementeret opdelingen af en tolagsgraf H i delgrafer vha. separatoren S.

Hver delgraf er defineret af en kant i den valgte flade, f.eks. (x, y) og delen er afgrænset af kanten samt de to rodstier fra endepunkterne på kanten, f.eks. rodstien $T(x)$ mellem x og r samt rodstien $T(y)$ mellem y og r. Der er nu tre dele, hhv. defineret af kanterne (x, y), (y, z) og (z, x), se figur 7.1. Dog kan en del være tom, hvis den definerende kant er med i det udspændende træ. Dette beskrives og illustreres i det følgende afsnit. Vi beskriver herefter hvorledes vi for en ikke-tom del, konstruerer en delgraf med de rette knuder og kanter. Vi afslutter med en opsummering af udførelsestiden for opdelingen.

7.2.1 Muligheder for færre dele af H

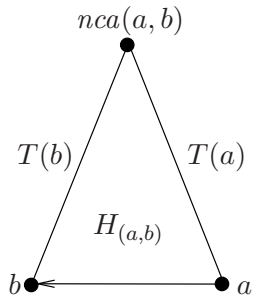
Vi ser nu på hvordan der bliver færre end tre delgrafer af H, når en eller to af kanterne i opdelingsfladen (x, y, z) er med i det udspændende træ.

På figur 7.3 betragtes tilfælde, hvor en eller to af de kanter, der er i opdelingstrekanten, er med i det udspændende træ. Her kan vi observere, hvordan der bliver en delgraf mindre for hver af kanterne, der tilhører det udspændende træ. Dette skyldes, at det der ligger mellem en kant og det udspændende træ er tomt, hvis kanten selv er en del at det udspændende træ.

Når vi finder delgrafer undersøger vi om en kant er en trækant, før vi ser på den del af H, som defineres af kanten. Hvis den *ikke* er en trækant, konstruerer vi delgrafen, der er defineret af kanten.

7.2.2 Konstruktion af en delgraf

Vi ser nu på en kant, som vi kalder (a, b), og skal konstruere den delgraf, der hører til den del af H, som (a, b) definerer. Vi kalder denne del af grafen for



Figur 7.4: $H_{(a,b)}$: Delen af H', som defineres af (a,b).

$H_{(a,b)}$, se figur 7.4. En kant (a,b) definerer altså netop én del, som ligger til venstre for $T(a)$ og til højre for $T(b)$. Alle knuder, der ligger i det indre af $H_{(a,b)}$ skal med i delgrafen. Kanter, som svarer til kanter i den originale graf, som har mindst et endepunkt i $H_{(a,b)}$ skal med i delgrafen. En sådan kant kalder vi en *original kant*. Vi bemærker dog, at hvis en original kant er med i det udspændende træ, kaldes den også en *trækant*. Vi bemærker, at vi med den originale graf mener grafen som dekomponeres, dvs. den graf, der optræder i det øverste kald i rekursionen – denne er en af G_i-graferne. De kanter vi ikke tager med, er f.eks. kanter, som er indsat under trianguleringen.

Vi starter med at finde den nærmeste fælles forfader for a og b i T, $nca(a,b)$. Vi observerer, at denne forfader kan være roden, men at den ikke nødvendigvis er det. Hvis $nca(a,b)$ er roden, vil roden ligge på stien, der afgrænser $H_{(a,b)}$. Hvis $nca(a,b)$ *ikke* er roden, vil roden enten ligge udenfor eller inden i den aktuelle del $H_{(a,b)}$, jf. figur 7.5.

Forskellige områder skal afsøges for knuder, der skal med i vores delgraf, alt efter hvilken af de nævnte tilfælde, vi befinder os i. Tilfældet, hvor roden ligger på stien, behandles på samme måde som tilfældet, hvor roden ligger udenfor delen. Vi har derfor to tilfælde; tilfældet, hvor roden ikke er i $H_{(a,b)}$ og tilfældet hvor roden er i $H_{(a,b)}$.

Hvis roden *ikke* ligger i $H_{(a,b)}$, afsøger vi følgende områder, for at finde knuder og kanter, der skal inkluderes i den nye delgraf, se figur 7.5 (A). Numrene i nedenstående liste svarer til numrene på figuren.

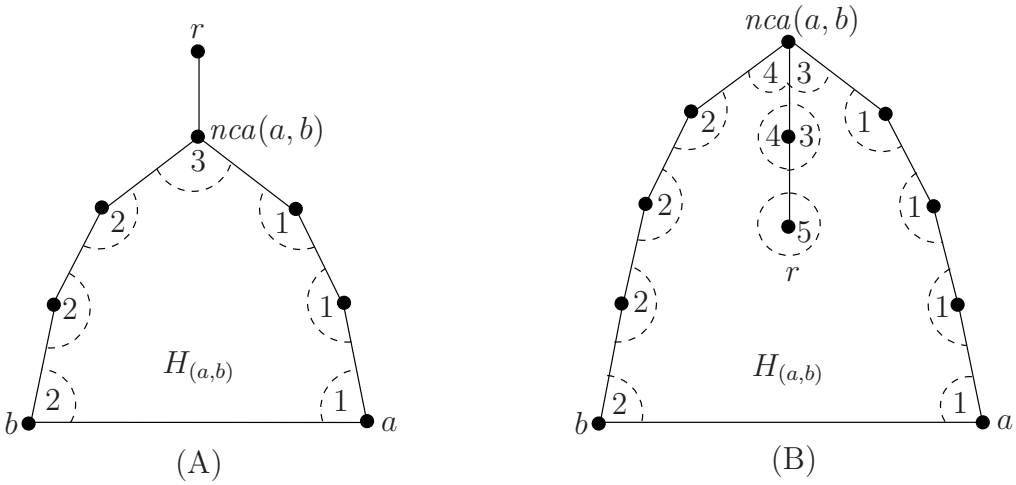
1. Området til venstre for stien mellem a og $nca(a,b)$ (Inklusiv a, eksklusiv $nca(a,b)$).

2. Området til højre for stien mellem b og $nca(a,b)$ (Inklusiv b, eksklusiv $nca(a,b)$).

3. Området rundt om $nca(a,b)$, der ligger mellem de to stier.

Hvis roden befinder sig inden i delen, afsøger vi følgende områder, se figur 7.5 (B). Igen svarer numrene på figuren til numrene i nedenstående liste.

1. Området til venstre for stien mellem a og $nca(a,b)$ (Inklusiv a, eksklusiv $nca(a,b)$).



Figur 7.5: Rodens placering ift. $H_{(a,b)}$. (A) Roden er ikke i det indre af $H_{(a,b)}$. (B) Roden er i det indre af $H_{(a,b)}$.

2. Området til højre for stien mellem b og $nca(a, b)$ (Inklusiv b, eksklusiv $nca(a, b)$).

3. Området til venstre for stien mellem $nca(a, b)$ og r (Inklusiv $nca(a, b)$, eksklusiv r).

4. Området til højre for stien mellem $nca(a, b)$ og r (Inklusiv $nca(a, b)$, eksklusiv r).

5. Området rundt om r.

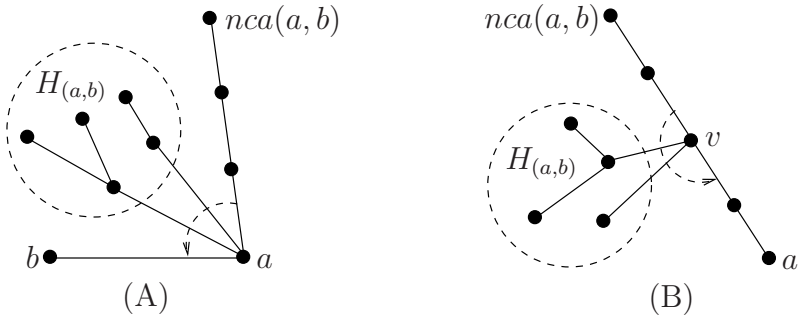
Når vi afsøger et område, finder vi de trækanter, som har et endepunkt på stien, der afgrænser $H_{(a,b)}$ og det andet endepunkt i det indre af $H_{(a,b)}$ (eksempler kan ses på figur 7.6). Når en sådan kant er fundet, følger vi den i retning mod det indre af $H_{(a,b)}$ – alle knuder og originale kanter vi møder, ligger i det indre af $H_{(a,b)}$ og skal med i den nye delgraf. Vi erindrer, at originale kanter, er kanter, der svarer til en kant i den originale graf (inklusiv trækanter) – det er disse kanter, vi vil have med i vores delgraf. Vi fortsætter med at følge trækanter indtil alle knuder og kanter vi kan nå er fundet.

I de følgende afsnit beskrives, hvorledes de rette trækanter findes for de forskellige områder samt hvordan det afgøres, hvilket af de to nævnte tilfælde vi befinder os i mht. rodens placering. Opgaverne gennemgås i den rækkefølge de udføres i.

Området til venstre for stien mellem a og $nca(a, b)$

Vi ser på hver knude på stien for sig og finder de trækanter, der leder fra denne knude ind i $H_{(a,b)}$.

For den første knude a vælges de kanter, der ligger mellem kanten (a, b) og den første kant i stien fra a mod $nca(a, b)$. Vi starter i sidstnævnte kant, går



Figur 7.6: Området til venstre for stien mellem a og $nca(a, b)$. (A) viser den første knude. (B) viser en af de øvrige knuder.

mod uret (da vores kantlister er ordnede mod uret). Vi vælger alle kanter, vi møder, indtil vi møder (a, b), hvor vi stopper. Se figur 7.6 (A).

De næste knuder behandles tilsvarende. For en knude v findes de kanter, der ligger i retning mod uret mellem de to kanter, på stien mellem a og $nca(a, b)$, der er incidente med v. Vi starter ved den kant på stien, der findes i retning mod $nca(a, b)$ og går mod uret til vi finder kanten, der findes på stien i retning mod a. Se figur 7.6 (B).

Vi bemærker, at vi slutter i den sidste knude på $T(a)$ inden $nca(a, b)$ og dermed står med den sidste kant på $T(a)$ før $nca(a, b)$ – denne kant skal anvendes når vi finder rodens placering, samt når vi afsøger de øvrige områder, hvorfor vi gemmer en reference til denne kant, som vi kalder e_a.

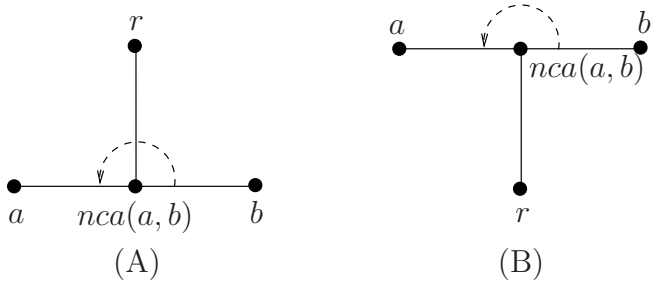
Området til højre for stien mellem b og $nca(a, b)$

Dette område findes på en tilsvarende måde som ovenstående område. Denne gang skal vi dog have de kanter, der ligger til højre for den sti, vi ser på. Vi bytter derfor om på, i hvilken kant vi starter og slutter, men går stadig mod uret. Dermed får vi de kanter, der ligger på modsatte side end før. Her står vi til sidst med den sidste kant på $T(b)$ før $nca(a, b)$, som vi gemmer en reference til. Vi kalder denne kant e_b.

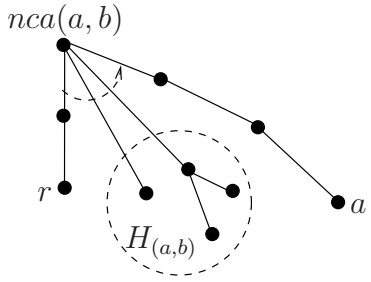
Afgørelse af rodens placering

Vi har nu fundet de områder (1 og 2), der er fælles for begge placeringer af roden. Derfor afgør vi nu hvor roden er placeret, inden vi fortsætter med at afsøge de øvrige områder.

Hvis roden ligger i det indre af $H_{(a,b)}$, vil vi møde kanten mod roden, hvis vi i knuden $nca(a, b)$ går mod uret fra e_b mod e_a, se figur 7.7 (A). Hvis vi ikke møder kanten mod roden, er roden udenfor $H_{(a,b)}$, som vi kan se på figur 7.7 (B). Vi gennemløber derfor alle de kanter, der er incidente med $nca(a, b)$ i retning mod uret, og ser på i hvilken rækkefølge vi møder kanterne mod hhv. roden, a og b. Vi har let adgang til disse tre kanter, som er hhv. forælderkanten samt e_a og e_b. Førstnævnte findes i labelen for $nca(a, b)$ og de sidstnævnte har vi gemt referencer til, da vi afsøgte de to første områder.



Figur 7.7: Placering af roden. (A) Roden er i $H_{(a,b)}$. (B) Roden er ikke i $H_{(a,b)}$.



Figur 7.8: Første knude i området til venstre for stien mellem $nca(a, b)$ og roden.

Området rundt om $nca(a, b)$ mellem de to stier

Dette udføres kun hvis vi er i tilfældet, hvor roden er udenfor $H_{(a,b)}$. Vi starter i e_b og vælger de kanter vi møder, når vi går i retning mod uret, indtil vi møder e_a. Igen følger vi de trækanter vi møder mod det indre af $H_{(a,b)}$, for at finde knuder, der skal med i delgrafen. De ønskede kanter, e_a og e_b, er igen til rådighed, da vi har sørget for at gemme referencer til dem.

Områderne til venstre/højre for stien mellem $nca(a, b)$ og roden

Dette udføres kun hvis vi er i tilfældet, hvor roden er i det indre af $H_{(a,b)}$. Metoderne til at finde de rette kanter i disse områder er analoge til metoderne vi anvender for område 1 og område 2. Den første kant i disse tilfælde er dog hhv. e_a og e_b, hvor de i de første tilfælde var kanten mellem a og b. På figur 7.8 viser vi hvordan de rette kanter i området til venstre for stien mellem $nca(a, b)$ og r findes ud fra den første knude, $nca(a, b)$.

Området rundt om roden

Dette udføres kun hvis vi er tilfældet, hvor roden er i det indre af $H_{(a,b)}$. Vi starter i den sidste kant på separatorstien – denne er incident med roden. Vi går mod uret og vælger alle de kanter vi møder, til vi igen er tilbage ved startkanten. På denne måde bliver alle kanter omkring roden valgt, bortset fra den, der er med i separatoren.

7.2.3 Udførelsestid for vores implementation

Vi ser nu på udførelsestiden for vores implementation af opdelingen af en graf H vha. en separator S. Vi gennemløber to af separatorens rodstier for hver del vi finder, dvs. hver rodsti gennemløbes to gange. Dette giver derfor et bidrag til udførelsestiden, der er lineær i størrelsen af S, der igen er lineær i størrelsen af H. Hver kant, der ikke er med i S besøges op til to gange – en gang fra hvert endepunkt, og der anvendes konstant tid for hvert besøg til at se på kanten og dens incidente knuder. Dette giver da ligeledes et bidrag til udførelsestiden, der er lineært i størrelsen af H. Den samlede udførelsestid af opdelingen er da lineær i størrelsen af H.

7.2.4 Placering i programmet

I `reachability/partition.cpp` findes den del af programmet, der konstruerer delgraferne.

Kapitel 8

Forbindelser over separatorer

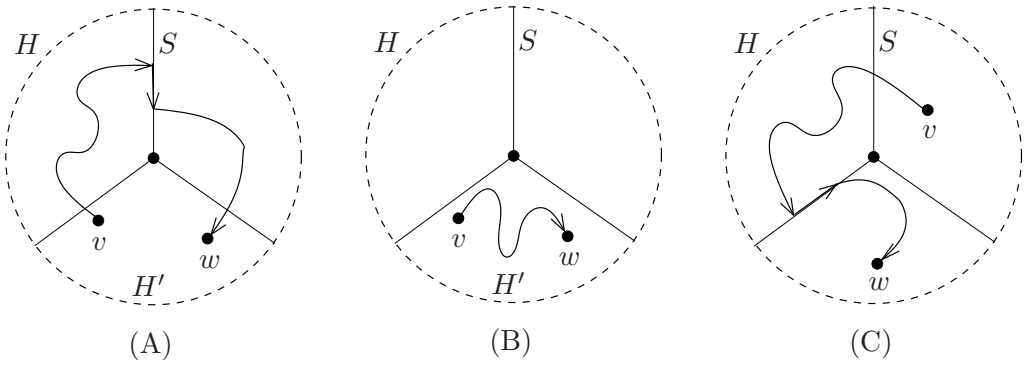
Vi har nu en tolagsgraf, H, som vi kan dele op i tre delgrafer vha. en separator S, som beskrevet i kapitel 5-7. Separatoren består af op til seks orienterede stier, hvilket vi redegør for i afsnit 5.2.2. Næste skridt i algoritmen er at repræsentere orienteret reachability over disse separatorstier på en effektiv måde.

Vi ser først på, hvordan en knude v kan nå en knude w i en af delgraferne under dekompositionen. I forbindelse med dette vil vi se på, hvilke informationer der er nødvendige for at kunne afgøre spørgsmål om reachability.

Vi laver *forbindelser* mellem knuder og separatorstierne, som indikerer, at en knude kan nå en sti hhv. nås fra en sti. I dette kapitel definerer vi forbindelser over separatorstier og vi beskriver sammenhængen mellem forbindelser og reachability samt hvorledes forbindelser anvendes til at afgøre reachability.

8.1 Reachability i en graf med separatorer

Hvis vi ser på dekompositionen af en af vores tolagsgrafer G_i, kan vi se på, hvorledes en knude v kan nå en knude w. Når vi på et sted i løbet af dekompositionen står med en tolagsgraf H, som opdeles af en separator S kan vi være i to situationer ift. knuderne v og w i H (situationerne illustreres på figur 8.1):



Figur 8.1: Separatoren S deler H i tre dele. Enten kommer v og w i samme del (billede A og B), ellers adskilles de af S (billede C).

1. Separatoren S deler H i tre dele, hvor v og w er i samme del, dvs. de to knuder adskilles ikke fra hinanden. Dermed vil v og w optræde i en delgraf H', som der kaldes rekursivt på i dekompositionen. Der er to måder, hvorpå der kan være en sti fra v til w i H. Enten forlader stien fra v til w H' og krydser en sti i S (billede (A)) ellers findes stien indenfor H' (billede (B)).

2. Separatoren S deler H i tre dele, hvor v og w er i hver sin del. Dermed er H den sidste graf i dekompositionen, der indeholder både v og w. I denne situation har v udelukkende mulighed for at nå w vha. en sti, der krydser en sti i S (billede (C)).

For at kunne svare på reachability-spørgsmål skal vi sørge for at have information om alle stier mellem knuder i en delgraf H, som krydser en af stierne i separatoren S. Vi bemærker, at en sti P fra v til w som er indeholdt i en delgraf H' af H, som det ses på figur 8.1 (B), senere i dekompositionen vil komme til at krydse en separator – dette vil senest ske, når v og w adskilles. Den information, vi har behov for at gemme, er da netop informationen om stier, der krydser de separatorer, der laves i vores dekomposition.

Faktum 8.1 *En knude v når en knude w i G_i hvis og kun hvis, v når w via en sti der krydser en separatorsti i en af de grafer i dekompositionen, der indeholder v og w (dvs. de separatorstier der adskiller v fra w i G_i).*

Vi vil i resten af kapitlet beskæftige os med hvorledes vi i vores datastruktur opbevarer information, der repræsenterer stier, der krydser en separatorsti.

8.2 Forbindelser over separatorstier

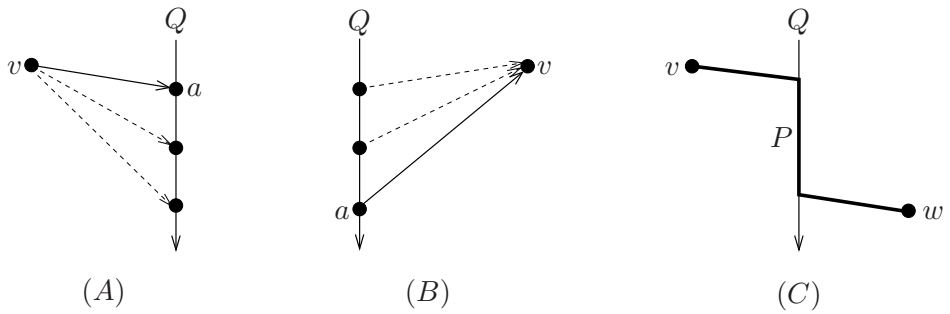
I det følgende vil vi definere begreberne *forbindelse* og *forbindelse over separatorsti*. Separatorstierne vi anvender, er de op til seks orienterede stier, som separatoren S består af – identifikationen af disse er beskrevet i afsnit 5.1.3. Hvorledes separatorstierne er nummereret er beskrevet i afsnit 5.1.3.

Definition 8.2 *Vi siger at der er en forbindelse fra en knude v til en knude a på en separatorsti Q hvis v kan nå a, og hvis a er den første knude på Q, som v kan nå, se figur 8.2 (A). Vi siger at der er en forbindelse til v fra a, hvis a kan nå v, og hvis a er den sidste knude på Q, der når v, se figur 8.2 (B).*

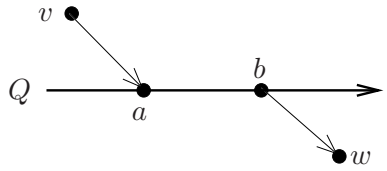
Hvis der er en forbindelse fra v til a på Q siger vi også, at v *forbinder* til a på Q.

Definition 8.3 *Vi siger at en knude v når en knude w over en separatorsti Q, hvis der er en sti P fra v til w, der skærer Q. Hvis v når w over Q, siger vi ligeledes, at der en forbindelse fra v til w over Q, se figur 8.2 (C).*

Vi har nu, at forbindelser mellem knuder kan sammensættes til forbindelser over stier. En sådan forbindelse indikerer, at der findes en vej mellem to knuder. Forbindelser over separatorstier er netop den konstruktion, som er nødvendig, for at vi kan repræsentere orienteret reachability, jf. nedenstående faktum.



Figur 8.2: (A) Forbindelse fra v til a på $Q - a$ er den første knude på Q, som kan nås fra v. (B) Forbindelse til v fra a på $Q - a$ er den sidste knude på Q, der kan nå v. (C) Forbindelse fra v til w over Q.



Figur 8.3: Forbindelse fra v til w over Q.

Faktum 8.4 *Der er en sti fra v til w over Q hvis og kun hvis v når en knude a på Q og w kan nås fra en knude b på Q, hvor a og b er samme knude eller hvor a kommer før b på Q, se figur 8.3.*

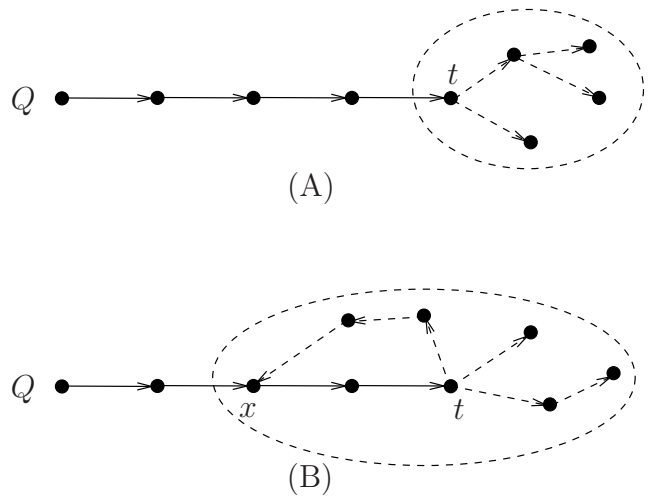
På figur 8.3 ses et eksempel på en forbindelse fra v til w over Q, hvor v når a på Q og w når b på Q. Ovenstående faktum giver anledning til, at vi ønsker at være i stand til at undersøge i konstant tid, om a kommer før b på Q. Dette kan lade sig gøre, hvis knuderne på hver separatorsti indekseres i retning med orienteringen af stien.

Definition 8.5 *Lad $i(a, Q)$ være indekset for a på Q. Hvis $i(a, Q) \leq i(b, Q)$, så er a samme knude eller en tidligere knude på Q ift. b.*

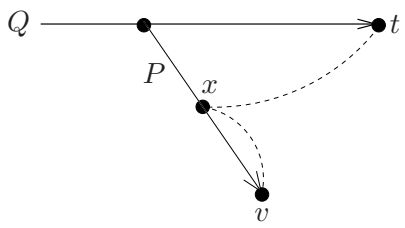
For at kunne afgøre, om to givne knuder kan nå hinanden over en given separatorsti, skal alle forbindelser, der går fra knuder i H til knuder på en sti Q i separatoren, identificeres. Ligeledes skal forbindelser, der går fra knuder på en separatorsti Q til knuder i H, identificeres. Hvorledes forbindelserne findes, beskrives i det konstruktive bevis for følgende lemma.

Lemma 8.6 *Givet en orienteret graf H og en orienteret sti Q, kan alle forbindelser mellem knuder i H og knuder på Q identificeres i lineær tid.*

Bevis. Først findes forbindelser fra Q. Vi laver et gennemløb af Q, der starter i Q's sidste knude, som vi kalder t. Vha. et bredde først-gennemløb, finder vi alle de knuder i H, der kan nås fra t, se figur 8.4 (A). Der skal pr. definition



Figur 8.4: Knuderne, der kan nås fra den sidste knude t på Q.



Figur 8.5: Hvis en knude, x på en sti P, kan nås fra t, kan de resterende knuder på P også nås fra t.

være forbindelser til alle disse knuder fra t på Q. Indekset for t på Q ($i(t, Q)$) gemmes i de knuder t når, da t er den sidste knude på Q der kan nå disse – dette gælder også t selv. Vi fjerner nu alle knuder (inkl. t selv), der kan nås fra t fra H og Q. Derefter kaldes rekursivt på resten af H og Q.

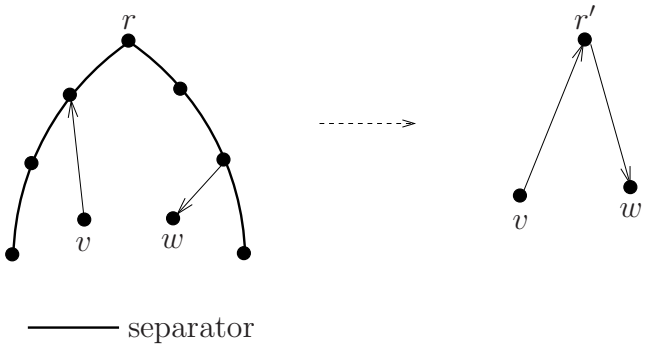
Udførelsestiden er lineær, da hver kant kun besøges en gang og for hver gang en kant besøges, anvendes konstant tid.

Korrektheden følger af disse to punkter:

1. Q bliver ved med at være en orienteret sti, fordi hvis t kan nå en knude, x, i Q, så kan t også nå alle knuder efter x. Derfor vil det altid være et suffiks af Q, der bliver fjernet, og det resterende af Q er dermed stadig en orienteret sti. Se figur 8.4 (B).

2. De eneste forbindelser, der ødelægges, når vi fjerner knuder, er de forbindelser fra t, som vi netop har identificeret. Dette ses ved at betragte en sti P, fra en knude på Q til en knude v i H. Hvis en knude, x, fra P fjernes, må t kunne nå denne knude og dermed også nå v, hvilket betyder, at v forbinder fra t på Q. Se figur 8.5.

Forbindelser til Q findes analogt. Her startes i Q's første knude, s, og vi ser på knuder, der kan nå s (vha. bredde først-gennemløb mod kanternes retning)



Figur 8.6: En separator sammentrækkes til en ny rod r', hvorved der skabes stier, der går over roden.

og fortsætter derefter med resten af Q. Vi får på denne måde den første knude på Q, som en knude v kan nå. □

Vi observerer her, at en knude v får tildelt to indekser for hver separatorsti Q, der opdeler en graf, som v er med i. De to indekser er et indeks for den sidste knude på Q, som kan nå v, samt et for den første knude på Q, som v kan nå. Da rekursionsdybden af dekompositionen er logaritmisk, er v med i et logaritmisk antal kald (og dermed i et logaritmisk antal delgrafer). I hvert kald er der et konstant antal separatorstier. Antallet af indekser, der skal gemmes for en knude v, bliver derfor $\mathcal{O}(\log(|\mathcal{V}(G_i)|))$, hvor G_i er den første graf i den rekursive opdeling.

Vi bemærker, at når der laves forbindelser til eller fra en sti, der indeholder roden, laver vi ikke forbindelser til og fra roden. Dette skyldes, at der kan skabes stier over roden ikke er stier i den oprindelige graf. Hvorledes stier over en rod kan opstå, er illustreret på figur 8.6. Dog vil der i den første tolagsgraf G_0 være en rod, der svarer til en knude i den oprindelige graf. For ikke at skulle behandle dette som et specialtilfælde, har vi tilføjet en ekstra knude til den første tolagsgraf G_0, som er rod i denne graf. Dette er beskrevet i afsnit 4.2.1. Hvis vi ikke havde tilføjet den ekstra knude, ville vi have et specialtilfælde i G_0, hvor der godt måtte være forbindelser til og fra roden. Den ekstra rod er derfor indsat for at afskaffe dette specialtilfælde.

8.2.1 Tidsforbrug og pladsforbrug

Vi kan konkludere, at det tager lineær tid at finde forbindelserne til og fra en separatorsti Q, ifølge lemma 8.6. For stien Q gemmes et indeks i alle knuder, der kan nås fra Q samt et indeks i alle knuder, der kan nå Q. Pladsforbruget for indekser til og fra Q er da også lineært.

Da der højst er $\mathcal{O}(\log(|\mathcal{V}(G_i)|))$ stier i de grafer en knude v er med i når vi rekursivt opdeler G_i, vil v højst gemme $\mathcal{O}(\log(|\mathcal{V}(G_i)|))$ indekser. Det samlede pladsforbrug for forbindelser der gemmes i knuder i en tolagsgraf er da $\mathcal{O}(n \log(|\mathcal{V}(G_i)|))$.

8.3 Implementation

I vores implementation anvender vi konstruktionen fra beviset for lemma 8.6 til at beregne indekser for hvilke knuder på en sti Q, der kan nå hhv. nås fra en knude v. For hver knude v beregnes altså et indeks på den første knude på Q, som v kan nå, samt et indeks for den sidste knude på Q, som kan nå v. Indekserne gemmes i lister i knuderne. For en sti Q anvendes klart lineær tid.

En forskel i vores konstruktion i forhold til beviset for lemma 8.6 er, at vi ikke sletter knuder og kanter i grafen. Hver knude v i grafen får en indliste (ind_v) og en udliste (ud_v), som de beregnede indekser gemmes i. For hver sti Q sætter vi de beregnede indekser ind i hver liste i en knude v såfremt der er en forbindelse hhv. til stien fra v og fra stien til v. Vi kan derfor anvende længderne af ind_v og ud_v til at afgøre, om knuden v er besøgt for den aktuelle sti. Længderne af listerne anvendes på denne måde som en farvning, der gør, at det er unødvendigt at slette knuder og kanter fra grafen. Har en knude ikke en forbindelse hhv. til eller fra en tidligere sti (eller flere), fyldes listen op med indgange der er -1, til den har den rigtige længde, inden et nyt indeks indsættes. Da der i de kald, hvor knuden v optræder i opdelingen af grafen G_i i alt er $\mathcal{O}(\log(|\mathcal{V}(G_i)|))$ separatorstier, vil ind_v og ud_v få en længde, der ligeledes er $\mathcal{O}(\log(|\mathcal{V}(G_i)|))$.

Vi bemærker igen, at vi har indsat en ekstra knude som rod i G_0. Når vi står med en tolagsgraf H, vil roden i H da altid være en knude, der ikke findes i den oprindelige graf G. Roden er enten denne ekstra knude eller en knude, der er fremkommet ved kontraktion af lag eller separatorer. Vi skal derfor ikke finde veje, der går hen over roden i en tolagsgraf. Når vi laver forbindelser fra hhv. til en knude på en separatorsti Q, der indeholder roden, laver vi da ikke forbindelser fra hhv. til roden.

Vi finder altså i lineær tid forbindelser hhv. fra og til en sti Q. Når vi rekursivt opdeler grafen G_i og finder forbindelser over alle separatorstier, er pladsforbruget for alle forbindelser $\mathcal{O}(|\mathcal{V}(G_i)|\log(|\mathcal{V}(G_i)|))$, da pladsforbruget i hver knude er $\mathcal{O}(\log(|\mathcal{V}(G_i)|))$.

8.3.1 Placering i programmet

Funktionerne, som konstruerer forbindelserne til og fra en given sti findes i `reachability/connection.cpp`.

Kapitel 9

Samlet datastruktur og forespørgsler

I dette kapitel gennemgås, hvorledes den samlede datastruktur er komponeret. Vi vil opsummere vores beviser angående forbrug af tid og plads og bevise at begge dele tilhører $\mathcal{O}(n \log(n))$. Herefter følger en beskrivelse af hvorledes datastrukturen kan anvendes til at svare på, om en given knude v kan nå en anden knude w og vi vil argumentere for, at en forespørgsel kan besvares i tid $\mathcal{O}(\log(n))$.

Strukturen af programmet, der udgør vores reachability-orakel, er illustreret i bilag A.

9.1 Samling af datastrukturen

I dette afsnit giver vi en kort gennemgang af, hvorledes de forskellige konstruktioner, der er beskrevet i kapitel 4-8 sammensættes til en datastruktur, der kan anvendes til at afgøre reachability. Vi afslutter med at vise at datastrukturen fylder $\mathcal{O}(n \log(n))$ og at konstruktionstiden ligeledes er $\mathcal{O}(n \log(n))$.

9.1.1 Basisrekursion

Givet en orienteret, planar graf G anvendes konstruktionen i kapitel 4 til at reducere problemet til at omhandle tolagsgraferne G_0-G_{k-1}. Vi arbejder nu med en tolagsgraf H med tilhørende udspændende træ T. I første skridt i rekursionen, er H en af graferne G_0-G_{k-1}. For tolagsgrafen H udfører vi følgende skridt:

- Vi forbereder H ved at triangulere den og tilføje labels, som beskrevet i afsnit 5.1.1. Denne del af algoritmen udføres i lineær tid.
- Vi finder en trekantet flade, der definerer en opdeling af grafen i tre dele, som hver højst indeholder halvt så mange knuder som H. Eksistensen af en sådan flade er bevist i sætning 6.1. Fladen kan findes i lineær tid – dette vises i sætning 6.10.
- Vi finder separatoren S, som er defineret af den netop lokaliserede flade. De orienterede stier identificeres. Disse skridt beskrives i kapitel 5, hvor

vi også viser, at der maksimalt er seks sådanne stier. Denne del udføres ligeledes i lineær tid.

- For hver af de orienterede separatorstier laver vi nu forbindelser. For en sti Q laves forbindelser fra Q til alle knuder i H, som kan nås fra Q. Tilsvarende laves der forbindelser fra knuderne i H til Q. Forbindelserne gemmes i hver knude i to lister – listerne findes i knuderne i den graf G_i, som H er en delgraf af. I listerne gemmes indekserne for hvilken knude i hver sti, der kan nås fra/nå til knuden. Længden af listerne er altså det samlede antal separatorstier, der findes i grafer, hvor knuden forekommer. For hver sti tager denne procedure lineær tid, og da der er et konstant antal stier, er den samlede udførelsestid for denne fase lineær. Hvorledes vi finder forbindelser er beskrevet i kapitel 8.

- Grafen H deles op vha. separatoren S. Vi konstruerer i lineær tid op til tre nye delgrafer. Denne fase beskrives i kapitel 7.

- Vi udfører rekursivt disse skridt på hver af de konstruerede delgrafer. Rekursionsdybden er logaritmisk, da grafernes størrelse som minimum halveres for hvert kald. Da der er et konstant antal separatorstier i hvert kald, vil der være $\mathcal{O}(\log(|\mathcal{V}(G_i)|))$ separatorstier på en sti fra roden til et blad i rekursionstræet for G_i.

Da alle skridt, der foretages i et kald tager lineær tid, og da rekursionsdybden er logaritmisk, er udførelsestiden for den basale rekursion for en graf G_i $\mathcal{O}(|\mathcal{V}(G_i)|\log(|\mathcal{V}(G_i)|))$. På hvert niveau i rekursionen findes højst seks orienterede separatorstier, hvorfor listerne af forbindelser i knuderne højst forlænges med seks. Derfor får listerne højst længde $\mathcal{O}(\log(|\mathcal{V}(G_i)|))$. Det samlede pladsforbrug for lister ved alle knuder er da $\mathcal{O}(|\mathcal{V}(G_i)|\log(|\mathcal{V}(G_i)|))$.

9.1.2 Rekursionstræer

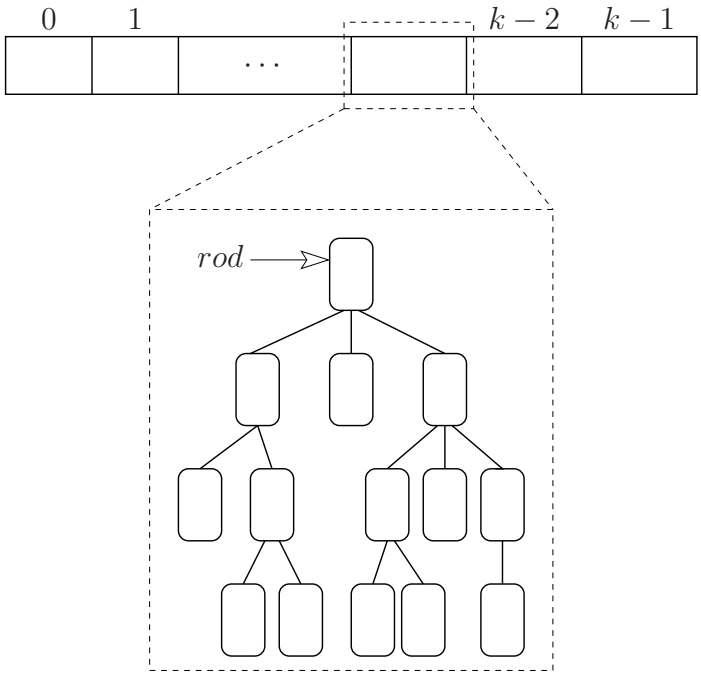
Undervejs i rekursionen skaber vi et rekursionstræ – der skabes altså et rekursionstræ for hver af graferne G_0-G_{k-1}. Hvert rekursivt kald C i opdelingen af G_i repræsenteres ved en knude i det tilsvarende rekursionstræ. Børnene til knuden er de knuder, der svarer til rekursive kald der foretages i kaldet C. Når vi gemmer et rekursionstræ gemmer vi ligeledes en reference til roden i træet.

Nummerering af separatorstier

Vi erindrer nu, at vi i afsnit 5.1.3 beskrev, hvorledes separatorstierne blev nummereret således at den første separatorsti i et kald fik et nummer højere end den sidste sti i forælderkaldet. Nummeret på den sidste sti i et kald kaldes separatornummeret for dette kald.

Knuder i et rekursionstræ

Vi kan nu forklare indholdet af rekursionstræet. En knude i rekursionstræet består af følgende dele:



Figur 9.1: Den samlede datastruktur, som er et array af rekursionstræer med rod. Der er en indgang i arrayet for hver af tolagsgraferne G_0-G_{k-1}

- Den graf H, der blev behandlet på dette sted i rekursionen.
- Roden i det udspændende træ i H
- Retning på H's første lag (indgående til roden eller udgående fra roden)
- En liste over de orienterede separatorstier, der findes i den separator, som anvendes til at dele H op med.
- Separatornummeret, dvs. nummeret på den sidste separatorsti, der findes i denne knude i rekursionstræet.

Samlet datastruktur

Et rekursionstræ (med rod) som det vi her har beskrevet, konstrueres for hver af tolagsgraferne G_0-G_{k-1} og gemmes i et array. Dette array af rekursionstræer udgør den endelige reachability-datastruktur, som nu kan danne grund for svar på reachability-forespørgsler. Den samlede datastruktur er illustreret på figur 9.1.

9.1.3 Ressourceforbrug for oraklet

Den samlede størrelse af tolagsgraferne G_0-G_{k-1} er lineær i størrelsen af den oprindelig graf G ifølge. sætning 4.6.

Dekompositionen for G_i, som er en af graferne G_0-G_{k-1}, kan udføres i tid $\mathcal{O}(|\mathcal{V}(G_i)| \log(|\mathcal{V}(G_i)|))$. Under denne skabes rekursionstræet, der er bidraget

til oraklet, som svarer til G_i. Den samlede udførelsestid for konstruktionen af oraklet for grafen G er da $\mathcal{O}(n \log(n))$, da den samlede størrelse af G_0-G_{k-1} er lineær i størrelsen af G.

Med et tilsvarende argument er pladsforbruget for oraklet $\mathcal{O}(n \log(n))$.

9.1.4 Placering i programmet

Basisrekursionen er implementeret i `reachability/reachability.cpp` samt `reachability/separate.cpp`. Implementationen af et rekursionstræ findes i klassen `reachability/RecursionTree`. Den samlede datastruktur er implementeret i klassen `reachability/ReachabilityOracle`. Alle generelle typedefinitioner, der anvendes i programmet, har vi samlet i `reachability/types.cpp`.

Hvorledes de forskellige klasser og filer indeholdende funktioner arbejder sammen, har vi illustreret i et diagram, der findes i bilag A.

9.2 Forespørgsler

Givet den samlede datastruktur, der er komponeret af rekursionstræer for hver af graferne G_0-G_{k-1}, vil vi nu beskrive, hvorledes vi givet to knuder v og w afgør, om v kan nå w. Vi argumenterer for, at udførelsestiden for en sådan forespørgsel er $\mathcal{O}(\log(n))$.

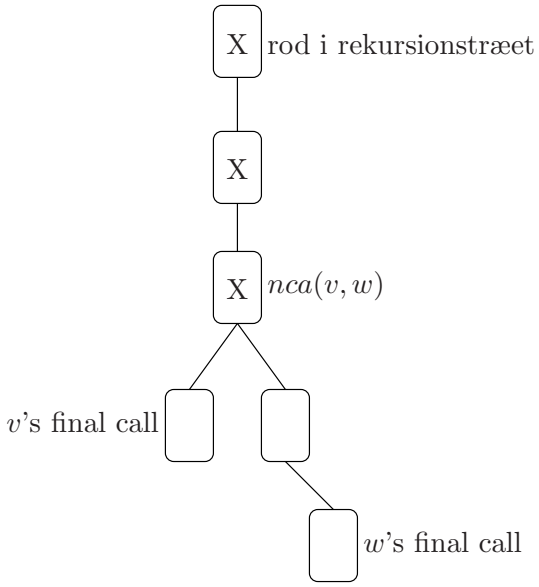
9.2.1 Fælles grafer

Første skridt i en forespørgsel er at finde ud af, i hvilke af tolagsgraferne G_0-G_{k-1} vi kan finde v og w. Vi ved hvilke lag v og w tilhører (vi har gemt $\iota(v)$ hhv. $\iota(w)$ i knuderne), og kan derfor beregne hvilke grafer v og w tilhører. Hvilke lag v og w er i kan slås op i konstant tid i v og w. Hvis ingen af graferne indeholder både v og w ved vi, at ingen vej kan eksistere fra v til w, hvilket er bevist i sætning 4.4. Er dette tilfældet returneres `falsk` på forespørgslen. Er det ikke tilfældet, vil der være en eller to grafer, der indeholder både v og w. Vi laver en forespørgsel i hvert at de rekursionstræer vi har konstrueret for grafen/graferne, der indeholder v og w. Returnerer en sådan `sandt` returneres ligeledes `sandt`, ellers returneres `falsk`. I det følgende afsnit gennemgår vi, hvorledes vi kan lave en forespørgsel i rekursionstræet for en tolagsgraf G_i. Vi argumenterer for, at en sådan forespørgsel kan foretages i tid $\mathcal{O}(\log(|\mathcal{V}(G_i)|))$.

9.2.2 Forespørgsel i rekursionstræet for G_i

Vi antager, at vi er givet knuderne v og w, som begge er indeholdt i grafen G_i, samt at vi er givet det konstruerede rekursionstræ for grafen G_i. Vi ønsker at afgøre, om v kan nå w i G_i.

Vi skal se på de rekursive kald, hvor både v og w er med i den tilhørende delgraf. Da vi har gemt det sidste kald en knude er med i (*final call*) for alle knuder ved vi hvilke kald, knuderne er med i. En knude er med i netop de kald, der findes i rekursionstræet på stien, der går fra knudens final call til roden i rekursionstræet. De kald, som v og w begge er med i, er altså de kald, der ligger



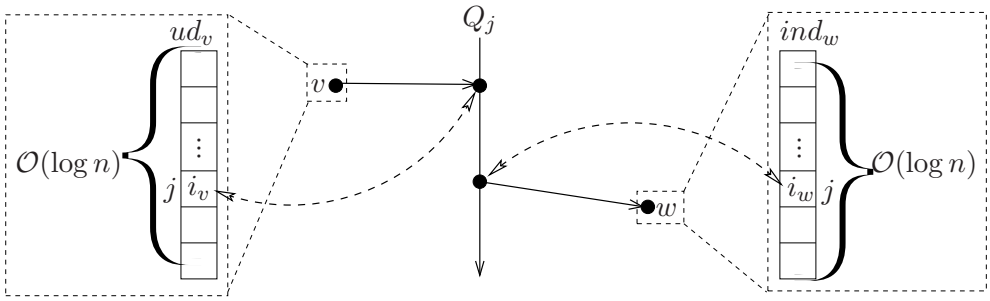
Figur 9.2: Rekursionstræet for en graf G_i. Kaldene markeret med X er de kald, vi betragter, når vi laver en forespørgsel på knuderne v og w.

på den fælles del af disse stier. Det sidste kald i træet, der indeholder både v og w, er den nærmeste fælles forfader for v's final call og w's final call. Denne knude kalder vi $nca(v, w)$. De kald, der indeholder v og w er nu de kald der ligger på stien, der går fra roden i rekursionstræet til $nca(v, w)$. Hvilke knuder i rekursionstræet vi betragter, kan ses illustreret på figur 9.2. Vi starter altså med at finde $nca(v, w)$, hvilket tager tid $\mathcal{O}(\log(|\mathcal{V}(G_i)|))$ pga. rekursionstræets højde.

Knuden $nca(v, w)$ indeholder et separatornummer s, som er nummeret på den sidste orienterede separatorsti, der findes på stien fra roden til denne knude i rekursionstræet. Vi kan nu gennemgå separatorstierne fra 0-s for at afgøre, om v når w over en af disse stier. For hver separatorsti Q, som har nummer højst s, finder vi indekset i_v på den første knude på Q, som v kan nå, hvilket slås op i ud_v. Ligledes finder vi indekset i_w på den sidste knude på Q, som kan nå w, hvilket slås op i ind_w. Hvis $i_v \leq i_w$, kan v nå w over den orienterede separatorsti Q. Er dette tilfældet, returnerer vi `sandt`. Er dette ikke tilfældet for en af stierne, returneres `falsk`.

En illustration af knuden v, der når knuden w over stien Q_j ses på figur 9.3. Indekserne i_v og i_w er slået op på den j'te plads i hhv. ud_v og ind_w, og disse svarer til knuder på Q_j, og hvis $i_v \leq i_w$ kan v nå w over Q, som det ses på tegningen. Størrelsen af ud_v og ind_w er $\mathcal{O}(\log(|\mathcal{V}(G_i)|))$, svarende til antallet af separatorstier i kald, der indeholder hhv. v og w.

Vi skal højst undersøge $\mathcal{O}(\log(|\mathcal{V}(G_i)|))$ stier, da der er et konstant antal stier på hvert niveau i rekursionstræet, og dette har logaritmisk højde. Det tager konstant tid at afgøre, om v når w over en givet sti Q. Derfor kan det afgøres i tid $\mathcal{O}(\log(|\mathcal{V}(G_i)|))$ om v når w i G_i.



Figur 9.3: Knuden v når knuden w over Q_j. Indekserne i_v og i_w, der er slået op i ud_v og ind_w, svarer til knuder på stien Q_j. Da $i_v \leq i_w$ kan v nå w over Q.

9.2.3 Udførelsestid for en forespørgsel

For hver graf G_i, der indeholder v og w anvendes tid $\mathcal{O}(\log(|\mathcal{V}(G_i)|))$ for at besvare en forespørgsel. Da den samlede størrelse af tolagsgraferne G_0-G_{k-1} er lineær i størrelsen af G (jf. sætning 4.6), vil denne udførelsestid være $\mathcal{O}(\log(n))$.

Der er højst to af tolagsgraferne G_0-G_{k-1}, der indeholder v og w. Derfor er den samlede udførelsestid for en forespørgsel ligeledes $\mathcal{O}(\log(n))$.

9.2.4 Placering i programmet

Klassen `reachability/ReachabilityOracle`, der implementerer den samlede datastruktur, indeholder metoden `query`, der foretager en forespørgsel.

Kapitel 10

Eksperimentel evaluering

Vi har efter endt implementation foretaget eksperimenter, der har forskellige formål. Disse formål er:

- At sandsynliggøre korrektheden af de resultater, som forespørgsler i reachability-oraklet returnerer. Her tester vi op mod en kendt korrekt algoritme.
- At lave statistik over vores forbrug af ressourcer, dvs. at lave statistik over hvor lang tid det tager at konstruere oraklet, hvor lang tid en forespørgsel tager samt hvor meget oraklet fylder.
- At teste om højden af dekompositionen af tolagsgraferne er logaritmisk i antallet af knuder i graferne.
- At lave statistik over, hvorvidt vores eksperiment, hvor vi valgte at finde maksimale orienterede stier i separatoren giver mærkbart færre stier, end hvis vi blot fandt alle sti-dele.

Vi har i høj grad lavet test på forskellige typer af automatisk genererede grafer. I afsnit 10.1 vil vi beskrive, hvordan vi genererer grafer til brug i forbindelse med test.

I afsnit 10.2 beskriver vi, hvorledes vi har testet korrektheden af vores reachability-orakel og vi konkluderer, at vi ikke har kunnet konstruere grafer hvor en forespørgsel giver et forkert svar.

Vi beskriver i afsnit 10.3 hvorledes vi har undersøgt pladsforbruget af vores orakel. Ud fra de test vi laver ser det ud til, at den forventede grænse på $\mathcal{O}(n \log(n))$ overholdes. På trods af, at det ser ud til, at oraklet fylder $\mathcal{O}(n \log(n))$ er det faktiske hukommelseforbrug temmeligt højt. Det skyldes, at det ikke har været et mål for os at optimere på pladsforbruget. Vi vil i afsnit 10.3.2 skitsere hvordan vi for en konkret graf med 10.000 knuder får brugt godt 200 MB.

I afsnit 10.4 beskriver vi de test vi har foretaget for at undersøge forbruget af tid. Vi har testet på, hvor lang tid det tager at opbygge mindre dele af oraklet og vi har set på den samlede konstruktionstid. Endvidere har vi undersøgt

hvor lang tid det tager at lave forespørgsler til vores orakel og vi har sammenlignet vores svartider med en lineærtids-algoritme, der kan svare på de samme forespørgsler.

I det sidste afsnit kigger vi på antallet af maksimale orienterede stier vi finder i separatorer og vi sammenligner dette antal med, hvor mange orienterede stier en mere simpel algoritme ville finde.

Vores test er kørt på en 1 GHz Pentium III processor med en 256 kB cache og 1 GB hukommelse.

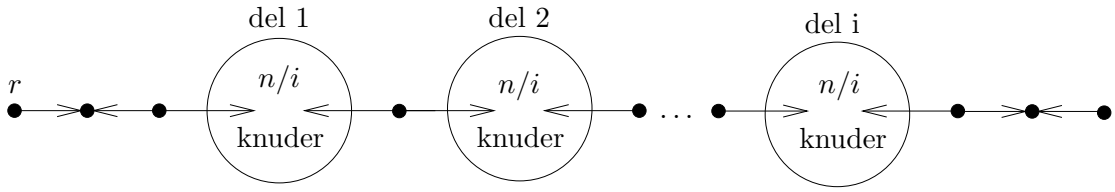
10.1 Testgrafer

Vi bruger metoder fra LEDA-biblioteket til at genere planare grafer. En tilfældig planar graf med n knuder og m kanter konstrueres ved først at generere en maksimal planar graf med n knuder og efterfølgende fjerne tilfældige kanter til det ønskede antal er opnået. En maksimal planar graf genereres ved først at konstruere tre knuder og forbinde dem med kanter. De resterende knuder tilføjes en ad gangen, ved at placere dem i en tilfældig flade og forbinde knuden med kanter til de knuder, der beskriver fladen. Når knuder tilføjes i forbindelse med generering af tilfældige planare grafer nummereres de i den rækkefølge de indsættes i grafen. Vi har observeret at der er en overvejende tendens til at kanter går fra en knude med et lavere nummer til en knude med et højere nummer. Måden hvorpå graferne genereres medfører, at graden af knuder med et lavt nummer er markant større end graden af knuder med et højt nummer.

Vi bruger LEDAs funktioner til at generere tilfældige grafer, men vi ændrer efterfølgende på graferne for opnå, at vores program tvinges ud i forskellige situationer.

For at undgå at kanter ofte går fra en knude med et lavt indeks til en knude med et højere indeks løber vi altid kanterne i en genereret graf igennem og bruger LEDAs tilfældighedsgenerator til at afgøre hvilken vej en kant skal vende. Dette øger sandsynligheden for, at der eksisterer en sti fra en knude med et højt nummer til en knude med et lavt nummer.

Vi ønsker at generere testgrafer hvor antallet af lag (definition 4.2) varierer, da dette giver os mulighed for at teste hvorledes forbruget af plads og tid varierer når antallet af lag varierer. Hvis vi bruger LEDA-funktioner til at generere tilfældige grafer vil der i en graf med 10.000 knuder og tre gange så mange kanter typisk være ca. 3 lag og det første lag vil typisk indeholde omkring 90% af grafens knuder. Vi har prøvet at variere antallet af kanter i forhold knuder, men dette har ikke givet nogen mærkbar forskel i antallet af lag. I stedet har vi valg at generere graferne ved at sætte dem sammen af delgrafer, se figur 10.1. Vi beslutter hvor mange knuder n og delgrafer i der skal være i grafen. Derefter genererer vi i af LEDAs tilfældige grafer hver med n/i knuder og $3*n/i$ kanter. Vi sætter graferne sammen i en kæde med to modsatrettede kanter mellem hver. Tilsidst laver vi en sti af alternerende kanter fra den første og den sidste graf. Vi lader roden være den første knude i en alternerende sti, på denne måde sikrer vi os, at der i de første grafer kun vil være få knuder. Den anden alternerede sti medfører, at de sidste grafer ligeledes har få knuder. Vi kalder graferne, der



Figur 10.1: Eksempel på hvorledes vi genererer grafer.

er opbygget som skitseret på figur 10.1 for *grafer bygget op af i delgrafer*.

I de følgende testafsnit vil vi arbejde med forskellige typer af genererede grafer og vi vil før hver test beskrive hvilken type af grafer vi har arbejdet med.

10.2 Test af korrektheden af resultaterne

Vi har naturligvis testet de enkelte dele af programmet efterhånden som vi har implementeret dem. Når vi laver en rekursiv dekomposition af en tolagsgraf har vi efterfølgende kunnet undersøge om opdelingen er foretaget korrekt i det vi gemmer de enkelte delgrafer ved hvert rekursivt kald. Efter at alle delene er sat sammen har vi har implementeret en simpel bredde-først algoritme til at teste, om der er en sti fra en knude til en anden. På en graf med 1000 knuder og tre gange så mange kanter tager det godt et par minutter for os at teste om vores resultatet er korrekt for et hvert par af knuder. Vi har derfor ikke lavet systematiske test med grafer med mere end 1000 knuder. På grund at størrelsen af vores program har vi ikke på nogen måde sikret os, at alle delene af koden bliver udført i forbindelse med de test vi har lavet. Vi har i stedet testet korrektheden af programmet med grafer, der indeholder $3 - 1000$ knuder og er opbygget på forskellig vis:

1. Tynde grafer (grafer med få kanter i forhold til knuder)
2. Tætte grafer (bl.a. maksimale planare grafer)
3. Grafer bygget op af mange delgrafer
4. Grafer bygget op af få delgrafer

De forskellige typer af grafer er opbygget med henblik på to ting; at variere antallet af tolagsgrafer, der konstrueres og variere antallet af forbindelser mellem knuder i grafen. Derudover varierer vi antallet af knuder i testgraferne.

De kørte test har alle givet korrekte resultater. Dette er naturligvis ikke et bevis for at vores orakel altid giver korrekte resultater, men blot en indikation af, at den måde vi genererer grafer sandsynligvis ikke får vores program til at give ukorrekte resultater.

10.3 Pladsforbrug for reachability-oraklet

Vi har testet om vores implementation af datastrukturen overholder de viste grænser for plads. Dette gøres ved at måle pladsforbruget umiddelbart før og

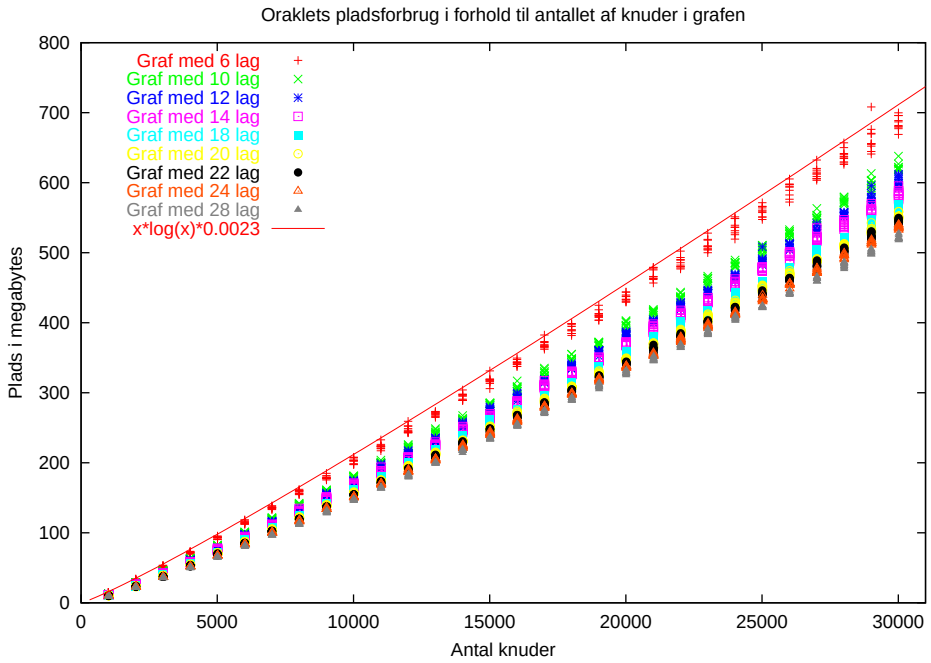
efter oraklet opbygges. Hvis vi under opbygningen af vores orakel leaker hukommelse vil det med de test vi laver se ud som om oraklet fylder flere bytes end det reelt gør. Vi har naturligvis været omhyggelige med ikke at leake hukommelse. Vi har forsøgt at bruge et profileringsværktøj (purify [pur]) til at hjælpe os med at undersøge om vi under opbygningen af oraklet får frigivet den nødvendige hukommelse, men det har desværre ikke været muligt for os at få det til at samarbejde med LEDA.

Vi har testet størrelsen af oraklet for grafer med op til 30.000 knuder og tre gange så mange kanter, se figur 10.2. Givet en graf med 30.000 knuder fylder oraklet godt 600 MB, hvilket lige knap er størrelsen af den tilgængelige hukommelsen i den maskine vi arbejder på. Vi har ikke arbejdet med større grafer, da det for disse tager for lang tid at opbygge oraklet. Vi har lavet en testsuite hvor vi 10 gange beregner pladsforbruget af grafer af størrelse $1000, 2000, \ldots, 30.000$. For hver testsuite har vi varieret antallet delgrafer, som den færdige testgraf er opbygget af, fra en til ti. Dette giver os grafer med 6 til 28 lag, dvs. 5 til 27 tolagsgrafer. Typisk består den første og den sidste graf af meget få knuder. Det betyder, at en stor procentdel af knuderne er med i to tolagsgrafer. Vi vil sidst i afsnittet diskutere testresultater hvor procentdelen af knuder, der er i to grafer er mindre.

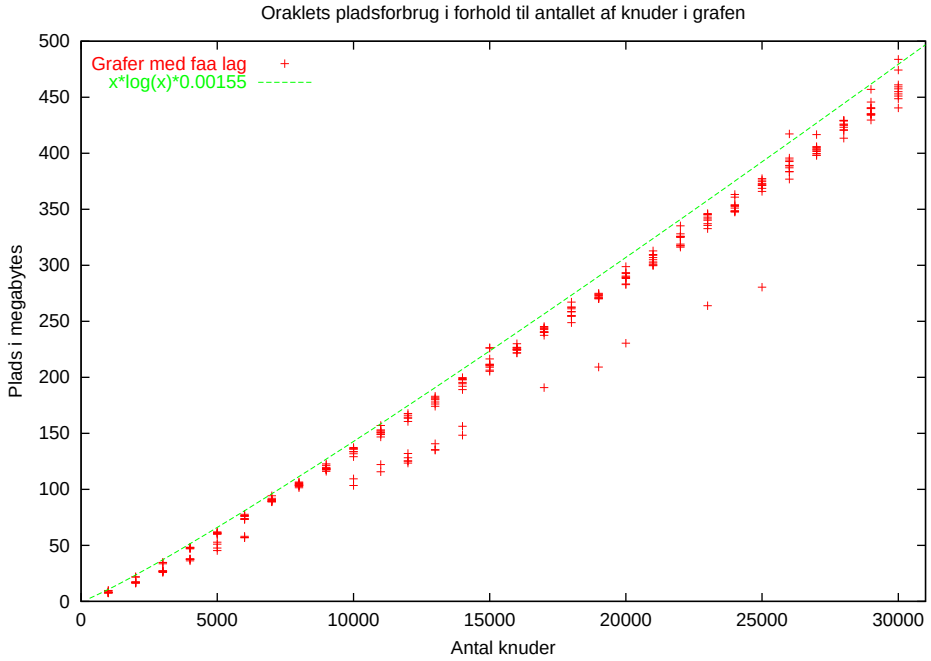
Vi har i afsnit 9.1.3 argumenteret for at pladsforbruget af vores orakel er $\mathcal{O}(n \log(n))$. Worst-case er, når alle knuder er i to tolagsgrafer og antallet af lag er minimalt. Hvis vi holder antallet af knuder fast og reducerer antallet af delgrafer, reduceres antallet af lag i den endelige graf. Når antallet af lag reduceres, øges størrelsen af de enkelte lag, dvs. størrelsen af tolagsgraferne øges. For hver tolagsgraf G_i opbygger vi et rekursionstræ, hvis højde er logaritmisk i antallet af knuder i G_i. En knude i rekursionstræet svarer til et i rekursivt kald i algoritmen, der deler grafen op ved brug af separatorer. I hvert rekursivt kald lave vi en kopi af de dele af grafen, vi arbejder videre med. Det betyder, at hvis højden af rekursionstræet øges laver vi flere kopier af knuderne og kanterne. En knude v i G_i er med i maksimalt $\mathcal{O}(\log(\mathcal{V}(G_i)))$ knuder i rekursionstræet; den er med indtil den bliver en del af en separatorsti. For hver gang v er med i et rekursivt kald skal den indeholde information om forbindelser til og fra separatorer i delgrafen. Hvis rekursionstræet er højt, er der til hver knude tilknyttet mere information end hvis træet er lavt. På figur 10.2 ser vi, at når antallet af lag reduceres, øges pladsforbruget, dvs. vores testresultater stemmer fint overens med teorien. Det bør bemærkes, at vores orakel fylder temmeligt meget. Vi vil i afsnit 10.3.2 give et eksempel på hvorledes vi får vores orakel til at fylde så meget.

På figur 10.2 kan vi tydeligt adskille de forskellige lag. En forklaring på dette er, at vi i vores test ikke har varieret antallet af knuder i de delgrafer som den endelige testgraf er opbygget af. Det betyder, at størrelsen af lagene for grafer, der er bygget op af samme antal delgrafer og med det samme antal knuder ikke vil variere meget. Når størrelsen af lagene er rimelig konstant vil højden af rekursionstræerne ikke variere meget. Det er hovedsaglige højden af rekursionstræerne, der er afgørende for hvor meget oraklet fylder.

Vi har lavet test hvor vi bygger testgrafen op af en enkelt af LEDAs tilfældige grafer og hvor vi ikke tilføjer to altenerende stier. Vi har dog vendt orienteringen



Figur 10.2: Forholdet mellem antallet af knuder i grafer med varierende antal lag og størrelsen af det konstruerede orakel.



Figur 10.3: Forholdet mellem antallet af knuder i grafer med få lag og størrelsen af det konstruerede orakel.

på et tilfældigt udsnit af kanterne. På figur 10.3 ses denne test. Når vi genererer grafer på denne måde, bliver der kun få lag. Det første lag bliver større end de sidste lag hvilket betyder, at mange af knuderne kun vil være i én tolagsgraf. Vi kan på figur 10.3 se at vi får to niveauer. Det nederste niveau er fra grafer, der har et eller to lag og alle knuderne er i én tolagsgraf. Uanset hvordan vi bygger testgraferne op, ser det ud til at pladsforbruget er $\mathcal{O}(n \log(n))$.

10.3.1 Højden af dekompositionen

Under opbygningen af oraklet laver vi en hierarkisk dekomposition af tolagsgrafer G_i. Dekompositionen laves ved, at G_i deles op i delgrafer ved brug af separatorer. Hver delgraf må ikke indeholde mere end halvdelen af knuderne i G_i. Hvis dette er opfyldt vil højden af dekompositionen blive $\mathcal{O}(\log(\mathcal{V}(G_i)))$. Vi har lavet en række test, der undersøger forholdet mellem antallet af knuder i en delgraf og højden af dekompositionen. Dataene ses afbilledet på figur 10.4, bemærk at x-aksen er logaritmisk. Vi har indtegnet en logaritmisk funktion sammen med vores data og det kan ses at vores beregnede resultater stemmer overens med teorien. For hver højde af dekompositionen, er der en del forskel på hvor mange knuder der er indeholdt i graferne, det skyldes at delene som graferne deles op i ikke er lige store.

10.3.2 Størrelsen af pladsforbruget

Når vi opbygger vores orakel for en graf G starter vi med at konstruere tolagsgraferne G_i. Hermed kopieres kanter og knuder i G op til 2 gange. Når vi opbygger rekursionstræet for en tolagsgraf G_i tager vi i hvert rekursivt kald en kopi af den del af grafen vi skal arbejde videre med. Derudover indeholder en knude i en tolagsgraf en stor mængde information, der bruges når diverse beregninger skal foretages. Endeligt indeholder oraklet informationen om forbindelser og separatorstier.

Vi har kigget på en graf med ca. 10.000 knuder og 30.000 kanter og undersøgt hvorledes pladsforbruget stiger under opbygningen af det endelige orakel. Vi har specielt fokuseret på hvad der sker når vi opbygger rekursionstræet for en tolagsgraf med mange knuder. Når vi står med en tolagsgraf H behandler vi den som følger:

- Trianguler grafen.
- Sæt labels på knuder og kanter.
- Find en trekant, find separatorstier og lav forbindelser over separatorstierne.
- Konstruer delgraferne og kald rekursivt.

Vi har genereret en graf med få lag, hvor størrelsen af den første og den sidste tolagsgraf er minimal. Antallet af knuder i tolagsgraferne er fordelt som følger:

Tolagsgraf	Antal knuder	Antal kanter
G_0	4	3
G_1	7.141	21.155
G_2	10.000	29.990
G_3	2.864	4.444
G_4	5	4

En del af rekursionstræet fra G_1 kan ses på figur 10.5. Tallet ved siden af knuderne i rekursionstræet er antallet af knuder i den tolagsgraf, der arbejdes med på det givne sted i rekursionen. Vi kalder de tre dele som G_i opdeles i på øverste niveau for H_1, H_2 og H_3. Vi vil nu gennemgå hukommelsesforbruget på udvalgte steder i programmet. De nævnte tal er i kilobytes.

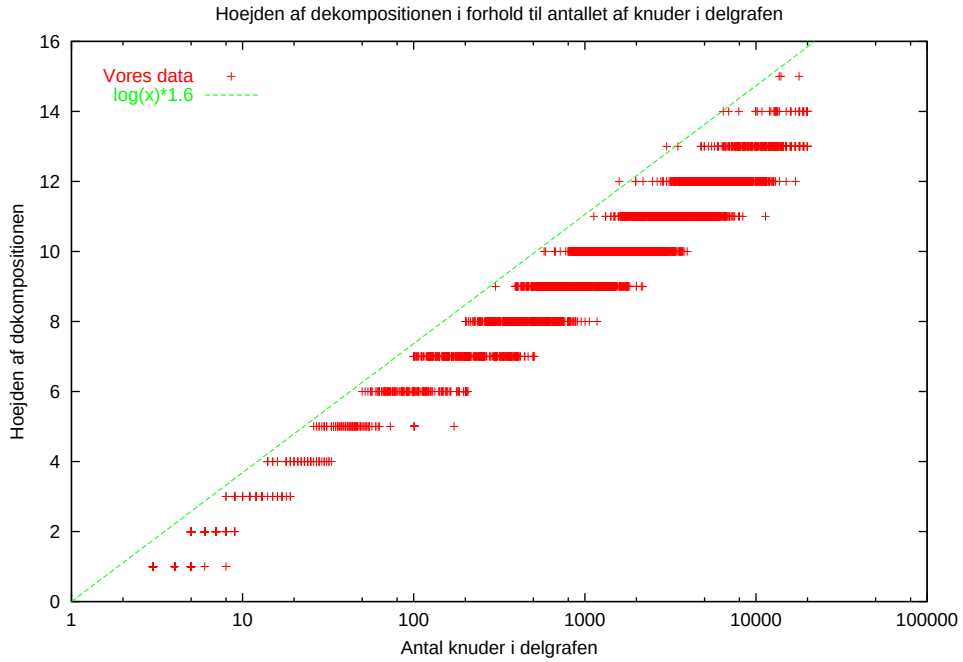
LEDA har konstrueret grafen G	11.760
G_i'erne er konstrueret	28.624
G_1 er trianguleret og er en tovejsgraf	32.672
Knuder i G_1 har fået labels	32.752
En trekant i G_1 er fundet og forbindelser er beregnet	33.416
Delgraferne af G_1 er konstrueret	37.344
Rekursionstræet for H_1 er konstrueret	62.108
Rekursionstræet for H_2 er konstrueret	85.276
Rekursionstræet for H_3 og dermed for G_1 er konstrueret	89.192
Rekursionstræet for G_2 er konstrueret	179.704
Rekursionstræet for G_3 er konstrueret	208.132
Rekursionstræet for G_4 er konstrueret og oraklet er opbygget	208.136

Vi kan ud fra ovenstående tabel se hvordan vi får brugt over 200.000 kB. Som ventet stiger pladsforbruget kraftigt når vi opbygger rekursionstræerne for G_i-graferne. Det første man skal gøre, hvis man vil nedbringe pladsforbruget, må derfor være at slette tolagsgraferne efterhånden som den relevante mængde information beregnes. I vores implementation indeholder separatorerne knuder fra den tolagsgraf i hvilken de blev konstrueret. Det er derfor ikke muligt for os blot at slette de overflødige tolagsgrafer. I kapitel 11 beskriver vi forskellige metoder til hvorledes pladsforbruget kan nedbringes. Det har som tidligere nævnt ikke været et mål for os at reducere pladsforbruget og det kan tydeligt ses.

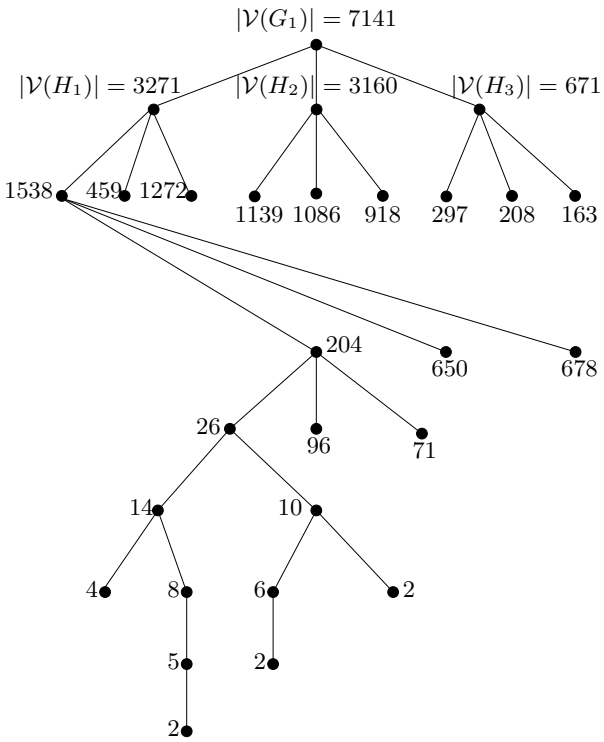
I afsnit 2.3 har vi beskrevet, at LEDA ikke automatisk frigiver hukommelse som den selv har allokeret. Som nævnt går hukommelsesallokeringen kun gennem LEDA, hvis det er blokke mindre end 255 bytes, der skal allokeres. Vi tror ikke, at LEDAs hukommelseshåndtering er en væsentlig medvirkende faktor til det store pladsforbrug, da vi ikke allokerer en stor mængde af små hukommelsesblokke som vi senere vil deallokere.

10.4 Tidsforbrug for opbygning af reachability-oraklet

Vi har lavet tests, hvor vi har beregnet hvor lang tid, det tager at opbygge datastrukturen. Dette er gjort ved at vi har målt hvor lang tid det tager at opdele grafen i lag, konstruere tolagsgraferne og lave en hierarkisk dekomposition



Figur 10.4: Forholdet mellem antallet af knuder i en delgraf og højden af dekompositionen.



Figur 10.5: Antallet af knuder i delgraferne for en delmængde af et rekursionstræ for en tolagsgraf G_i med 7141 knuder

af disse. Vi har ligeledes lavet undersøgelser på enkelte dele af opbygningen, idet vi har taget tid på hvor lang tid det tager at udføre to forskellige dele af dekompositionen af en tolagsgraf. Vi forsøgte at udføre test på mindre dele af både opbygningen af tolagsgrafer og dekompositionen, men det tog kortere tid, end vi kunne måle.

10.4.1 Tidsforbrug for første del af dekompositionen

Givet en graf G opdeler vi den i lag og konstruerer tolagsgraferne G_i. For hver tolagsgraf G_i, opbygger vi et rekursionstræ. I hvert af de rekursive kald, der udføres står vi med en tolagsgraf H, der skal forberedes. Vi har målt hvor lang tid nedenstående del af, hvad der udføres i et rekursivt kald tager.

- Lav H til en tovejsgraf.
- Trianguler H.
- Sæt labels på grafen.
- Find en trekant.
- Lokaliser separatoren.
- Find forbindelser mellem separatoren og resten af grafen.

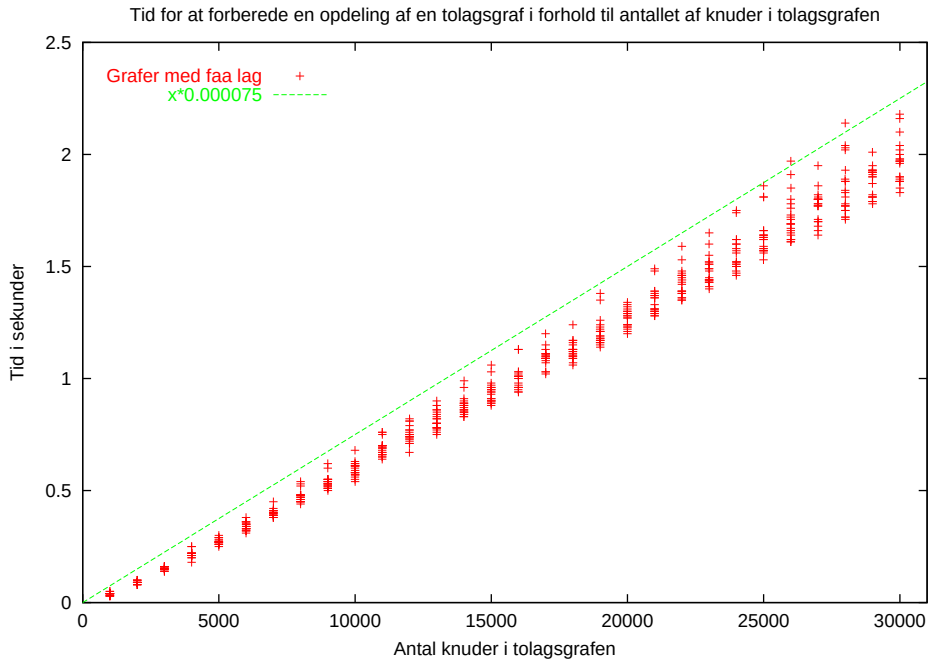
I afsnit 5.2.1 beskrives, at vi kan sætte labels på H, lave den til en tovejsgraf og triangulere den i lineær tid. Vi har i sætning 6.10 bevist, at vi kan finde en trekant i lineær tid og i lemma 8.6 beviser vi, at forbindelser mellem en separator og H findes i lineær tid. Samlet giver det os at udførelsestid bør være lineær.

Vi har lavet test på grafer med 1.000 til 35.000 knuder. Vores testgrafer er opbygget ved hjælp af LEDAs grafgenerator. Vi har gennemløbet grafens kanter og vendt en tilfældig mængde af dem. De genererede grafer har få lag og hovedparten af knuderne er i den første tolagsgraf. Vi har testet hvor lang tid den første del af opbygningen tager for den første tolagsgraf G_0. På figur 10.6 ses vores testresultater afbilledet. Vi har indtegnet en ret linje på grafen og de beregnede data følger den nogenlunde.

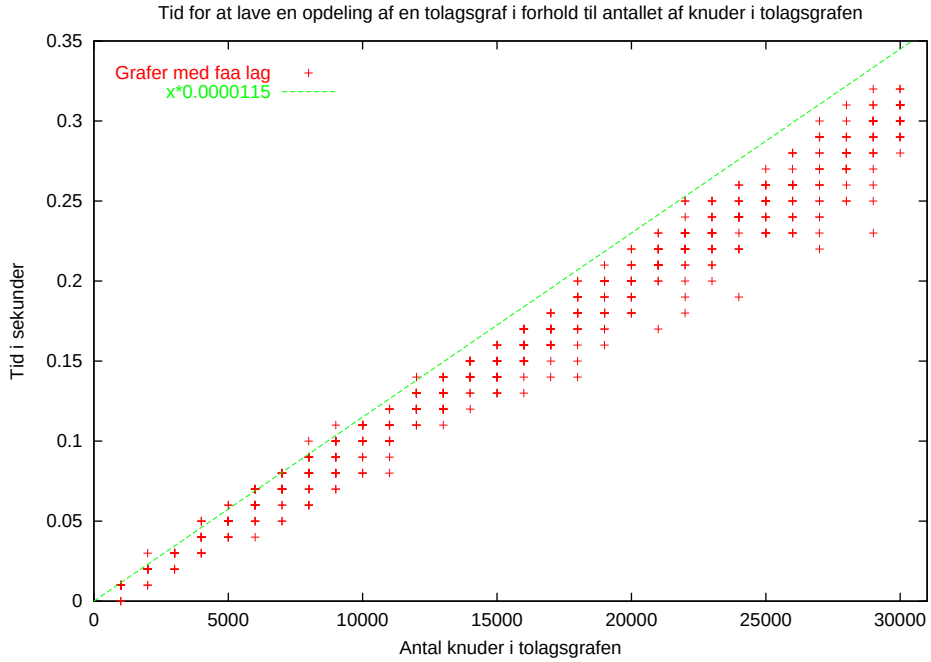
Vi har blot været interesserede i at undersøge om udførelsestiden så ud til at være lineær og har derfor ikke foretaget yderligere test med andre typer af genererede grafer.

10.4.2 Tidsforbrug for at lave opdelingen af tolagsgraferne

Givet en tolagsgraf H, finder vi en separator S og beregner forbindelserne fra knuder i H til stierne i S. Derefter deles tolagsgrafen op i højst tre delgrafer. I afsnit 7.2 argumenterer vi for, at vi foretager opdelingen i lineær tid i antallet af knuder i H. Vi har set på, hvor lang tid det givet en tolagsgraf og en separator tager at konstruere de op til tre delgrafer som S deler H op i.



Figur 10.6: Forholdet mellem antallet af knuder i en tolagsgraf og hvor lang tid det tager at lave første del af dekompositionen.



Figur 10.7: Forholdet mellem antallet af knuder i en tolagsgraf G_i og hvor lang tid det tager at udføre partitionen af G_i.

Vi har lavet testene på den samme typer grafer som i afsnit 10.4.1 og resultatet kan ses på figur 10.7. Opdelingen ser på vores testdata ud til at blive udført i lineær tid i størrelsen af H, hvilket stemmer overens med teorien.

Vi har igen ikke testet med andre typer grafer. Vi har blot været interesserede i at undersøge om udførelsestiden så ud til at være lineær, ikke at undersøge hvordan udførelsestiden ændrer sig når testgraferne er anderledes bygget op.

10.4.3 Tidsforbrug for den samlede konstruktion

I afsnit 9.2.3 argumenterer vi for, at vores orakel kan konstrueres i tid $\mathcal{O}(n \log(n))$. Udførelsestiden for at dele grafen G op i lag og konstruere tolagsgraferne er ifølge korollar 4.8 $\mathcal{O}(n)$. For hver tolagsgraf G_i konstruerer vi et rekursionstræ og dette tager tid $\mathcal{O}(|\mathcal{V}(G_i)| \log(|\mathcal{V}(G_i)|))$. Vi argumenterede i afsnit 10.3 for, at oraklet fylder mere des færre lag grafen G dels op i. Med samme argumenter tager det naturligvis også længere tid at opbygge vores datastruktur des færre lag testgraferne deles op i.

Vi har målt på, hvor lang tid det tager at opbygge oraklet. Vores testgrafer er de samme som i afsnit 10.3. På figur 10.8 ses vores resultater. Som forventet tager det længere tid at konstruere orakler ud fra testgrafer med få lag end ud fra grafer med mange lag. Vi har indtegnet en $n \log(n)$-funktion i grafen og vores resultater ser ud til at stemme overens med vores forventinger.

10.5 Tidsforbrug for forespørgsler

Når vores orakel skal besvare en forespørgsel for to knuder v og w, undersøges først hvilke tolagsgrafer, der indeholder knuderne. Som beskrevet i afsnit 9.2.1 kan dette afgøres i konstant tid ved et tabelopslag. Begge knuder kan være indeholdt i op til to tolagsgrafer. For hver tolagsgraf G_i, som indeholder begge knuder, undersøges det om der eksisterer en separator, der forbinder dem. I afsnit 9.2.2 beskriver vi hvorledes dette gøres i tid proportional med højden af rekursionstræet tilhørende G_i. Denne højde er som bekendt $\mathcal{O}(\log(|\mathcal{V}(G_i)|))$. Højden af rekursiontræet er som tidligere nævnt afhængig af antallet af lag i grafen G. Vi forventer igen at se et større forbrug af tid, des færre delgrafer vores testgrafer er opbygget af.

Vi har ikke kunnet udføre målinger på enkelte forespørgsler, da dette tager kortere tid end vi kan måle. I stedet har vi for en graf med n knuder foretaget samtlige n^2 forespørgsler. Vi har efterfølgende beregnet den gennemsnitlige svartid for en query og dette har vi afbilledet, se figur 10.9. Bemærk at x-aksen er logaritmisk og at tiden på y-aksen er i mikrosekunder. Som i testene beskrevet i de forrige kapitler, ser vi igen på grafer af forskellig størrelse og med forskellige antal lag. Som forventet er svartiden længere des færre lag vores testgrafer har. Dette skyldes som tidligere nævnt at rekursionstræet for tolagsgraferne bliver højere.

Vi forventede, at svartiden i forhold til antallet af knuder i testgraferne skulle kunne approksimeres med en logaritmisk funktion. Vi har testet intensivt på grafer med knuder i intervallet $100, \ldots, 1000$, men erfarede, at dette interval ikke var stort nok til vi kunne se en klar tendens. Vi har derfor testet med

grafer med flere knuder, for at undersøge om vores resultater så ud til efter et vist punkt at kunne approksimeres med en logaritmisk funktion. Det tager naturligvis væsentligt længere tid at udføre test med større grafer og vi har derfor ikke udført så mange. Vi mener, at de indtegnede logaritmiske funktioner er rimelige approksimationer. Begge funktioner er på formen $a\log(x) - b$, dette skyldes, at mange queries tager væsentligt kortere end logaritmisk tid, da vi ofte ikke løber helt ned i bunden af rekursionstræet.

Vi har lavet tidskørsler på forespørgsler til vores datastruktur, som vi har sammelignet med tidskørsler for et simpelt bredde først-gennemløb. Vi har kun kørt test med grafer med få lag, da vi blot er interesserede i at undersøge om oraklet besvarer spørgsmål hurtigere end et bredde først-gennemløb. Som det ses på figur 10.10 er vores orakelstruktur heldigvis væsentligt hurtigere til at besvare forespørgsler end et lineært gennemløb af grafen.

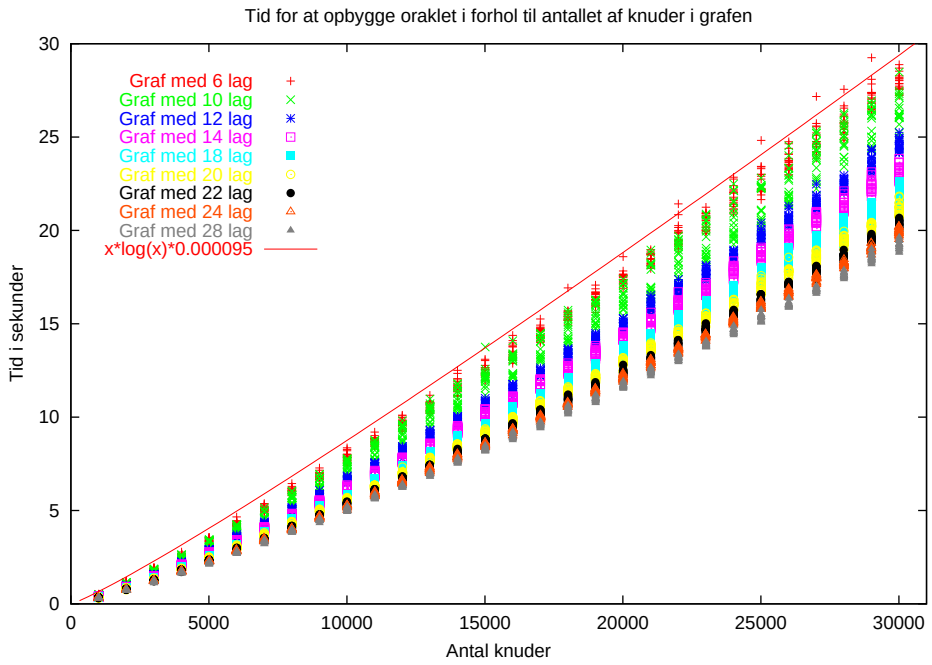
Vi har undersøgt, om der et så stort overhead når en query besvares, at det for mindre mængder af data bedst vil kunne betale sig at lave et bredde først-gennemløb af grafen. På figur 10.11 ses resultatet af disse test. Vi kan konkludere, at når oraklet først er bygget, er det hurtigst at bruge dette til at besvare queries angående reachability.

10.6 Antal stier i en separator

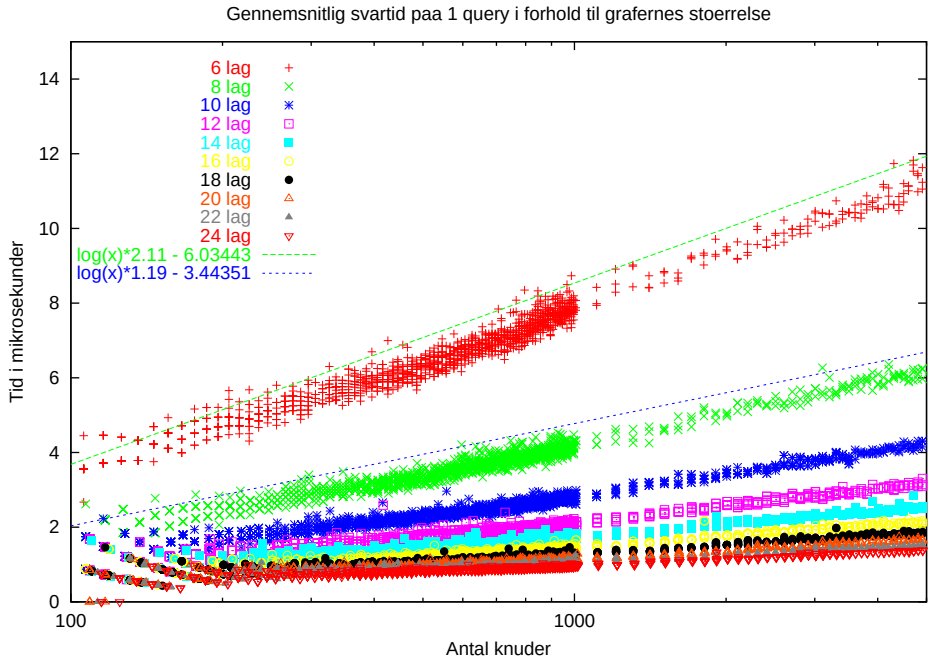
Givet en tolagsgraf H og en flade F, defineres en separator S. Vi beskriver i afsnit 5.2.2, hvordan vi finder de maksimale orienterede stier i S. I dette kapitel tester vi, hvor mange stier vi finder. Vi sammenligner antallet af maksimale stier, der identificeres i vores program, med antallet af stier, der ville blive lavet, hvis vi brugte den simple strategi, som er antydet i artiklen [Tho01]. Vi erindrer, at den simple strategi indebærer, at man for hver af knuderne der definerer F følger stien til roden og vælger de orienterede dele man møder. Vi kalder de stier som vi finder for *maksimale stier* og dem der findes ved brug af den simple strategi for *simple sier*.

Vi har set på, hvor mange stier, hhv. maksimale og simple, der bliver fundet for en given graf. Vi har kørt få tests grafer med op til 500 knuder, for at kunne se en tendens. Resultaterne kan ses på figur 10.12.

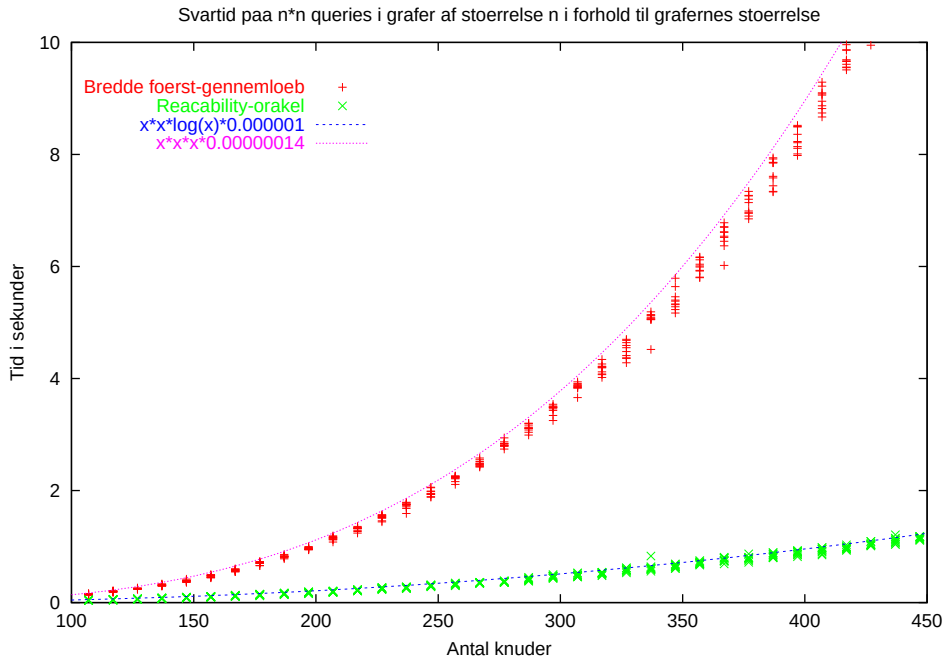
Vi kan se, at antallet af simple stier er lidt større end antallet af maksimale stier. Det betyder, at vi i nogle rekursive kald finder et mindre antal orienterede stier. Vi konkluderer, at vi sparer et antal stier i forhold til metoden, hvor vi blot vælger alle simple stier, der findes på rodstierne i separatoren.



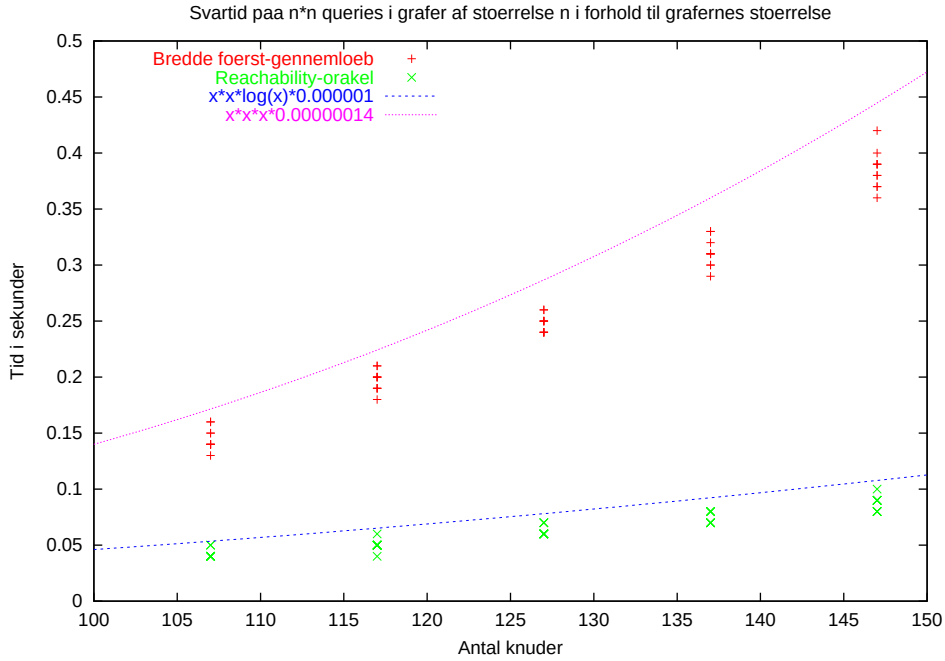
Figur 10.8: Forholdet mellem antallet af knuder i grafer med varierende antal lag og tiden for at konstruere oraklet.



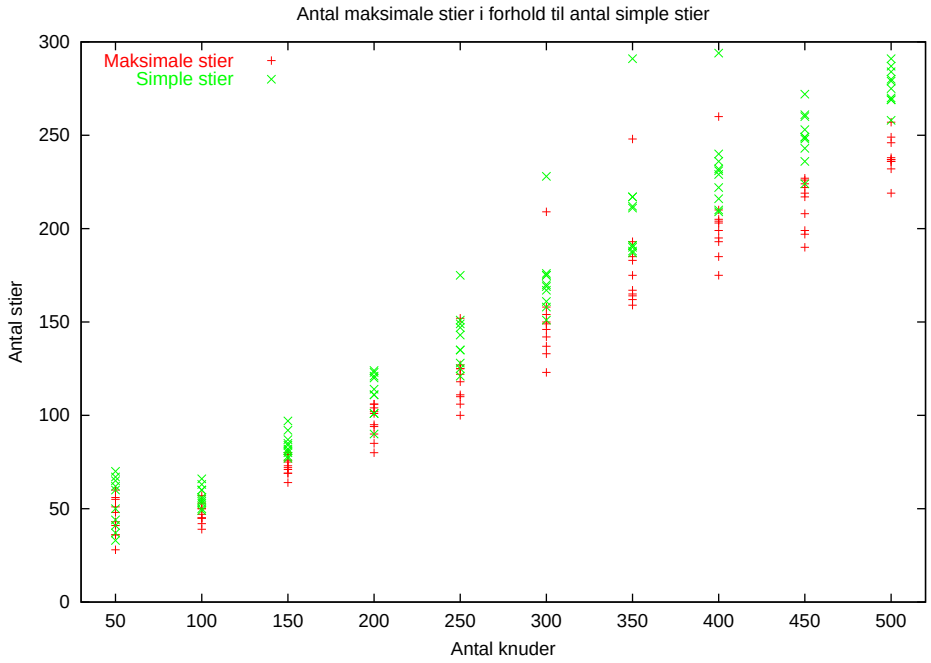
Figur 10.9: Forholdet mellem antallet af knuder i grafer med varierende antal lag og gennemsnitstiden for at lave en forespørgsel



Figur 10.10: Forholdet mellem antallet af knuder i grafer med få lag og tiden for at lave n^2 forespørgsler



Figur 10.11: Forholdet mellem antallet af knuder i grafer med få lag og tiden for at lave n^2 forespørgsler



Figur 10.12: Forholdet mellem antallet af simple stier og antallet af maksimale stier.

Kapitel 11

Muligheder for forbedringer og udvidelser

I dette kapitel vil vi beskrive hvorledes opbygningen af datastrukturen kan ændres således at forespørgsler kan besvares i konstant tid. Dette kræver, at man i stedet for at skulle gennemse $\mathcal{O}(\log(n))$ separatorstier kun skal se et konstant antal stier igennem, for at se om en knude v kan nå en knude w. Det kræver ligeledes, at den nærmeste fælles forfader for to rekursionstræsknuder kan findes i konstant tid. Vi vil give en teoretisk gennemgang af, hvorledes disse forbedringer af datastrukturen opnås uden at konstruktionstiden og pladsforbruget øges.

Vi vil i dette kapitel også se på, hvilke udvidelser, der skal foretages, for at en vej fra en knude v til en knude w kan angives, hvis v kan nå w.

Sidst vil vi komme ind på hvorledes reachability-information kan distribueres som labels på grafens knuder.

11.1 Reducering af udførelsestid for forespørgsler

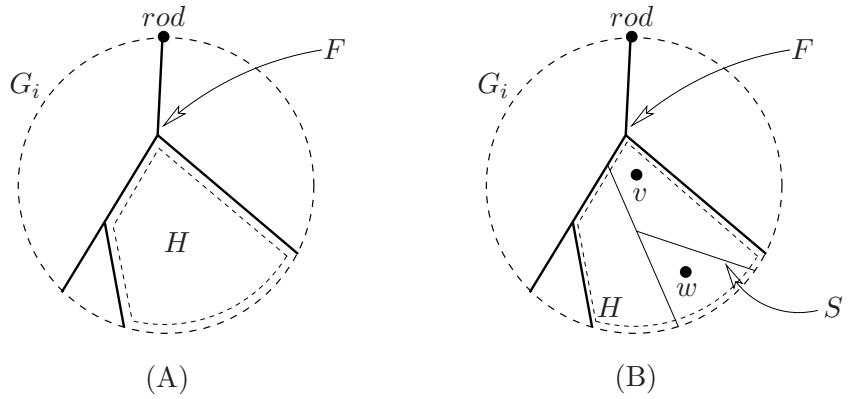
For at gøre forespørgselstiden konstant, indfører vi begrebet *frame*, der ligesom en separator er en samling af rodstier. Vi indfører ændringer i vores basisrekursion, således at vi får en alternerende rekursion, der i hvert andet skridt har til formål at gøre delgraferne mindre og i de modsatte skridt har til formål at mindske antallet af rodstier, der er relevante for knuderne i en delgraf.

Vi bemærker at vi i dette afsnit ikke betragter maksimale stier, som tidligere, men blot ser på hele rodstier, som hver er sammensat af op til to orienterede stier.

11.1.1 Frame og separator for en delgraf

Vi ser først på, hvad en frame er og derefter på, hvorledes forbindelser over framen samt separatoren for tolagsgrafen H udgør den reachabilityinformation, vi har brug for.

Definition 11.1 *Lad en tolagsgraf H, som er en delgraf af tolagsgrafen G_i, være givet. En* frame *F for H er en samling af rodstier, der adskiller H fra resten af G_i. En illustration kan ses på figur 11.1 (A).*



Figur 11.1: (A) Framen F adskiller H fra resten af grafen G_i. (B) $F \cup S$ adskiller v fra w i G_i.

Hvis vi har en frame F, der adskiller H fra resten af grafen G_i og en separator S for H, som adskiller knuderne v og w fra hinanden i H, da vil $F \cup S$ adskille v fra w i G_i, se figur 11.1 (B). For hver af stierne $Q \in F \cup S$ vil vi derfor finde forbindelser til og fra alle knuderne i H. Forbindelserne findes over G_i, dvs. at for $v \in H$ finder vi den første knude på Q, som v kan nå i G_i samt den sidste knude på Q, som kan nå v i G_i. Ideen er nu, at hvis F og S holdes på en konstant størrelse, vil det kræve et gennemsyn af et konstant antal stier at afgøre, om knuden v kan nå knuden w i G_i.

11.1.2 Alternerende rekursion

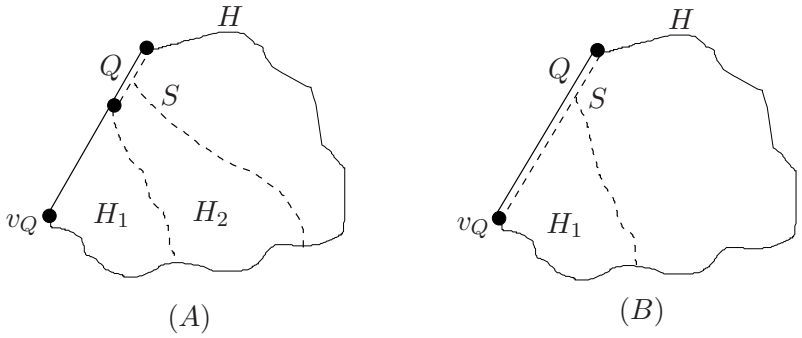
Når vi arbejder med frames, laver vi nogle ændringer i den basale rekursion, som den er beskrevet i afsnit 9.1.1. For det første arbejder vi med en graf, der indeholder framen. For det andet laver vi en alternerende rekursion, hvor vi skifter mellem to forskellige handlinger i de rekursive kald.

Definition 11.2 *Grafen $H + F$ er den graf, der består af H og F samt de kanter i G_i, der ligger mellem H og F.*

Framen F er som nævnt en samling rodstier, der adskiller H fra G_i. Når vi i delgrafen H har fundet en separator S skal vi kunne finde forbindelser mellem knuder i H og S over G_i. I forbindelse med dette bruger vi kanterne mellem H og F. Størrelsen af $H + F$ er ikke begrænset af størrelsen H. Dette skyldes at antallet af knuder i F ikke er afhængig af størrelsen af H. Vi vil i afsnit 11.1.4 beskrive hvorledes vi reducerer antallet af knuder i F, således at den kun indeholder knuder, der er relevante for H.

I vores alternerende rekursion arbejder vi med $H+F$ og vi alternerer mellem de følgende to typer af rekursion – vi beskriver her, hvad der sker i kald af hver type rekursion, hvor vi skal opdele en graf H med en frame F:

Delgraf-reducering. Den første type er *delgraf-reducering*, hvor vi udfører de samme skridt som i basisrekursionen, som vi har beskrevet den i afsnit 9.1.1. Der findes altså en separator S, som opdeler H i op til tre



Figur 11.2: Illustration af hvornår stien Q, som tilhører framen for H, er relevant for de delgrafer, som H opdeles i.

dele. Vi kalder denne for delgraf-reducering, da de resulterende delgrafer højst indeholder halvt så mange knuder som H. I dette tilfælde får alle delgraferne $F \cup S$ som frame. Vi øger altså F med tre rodstier.

Frame-reducering. Den anden form for rekursion kalder vi *frame-reducering*, da den har til formål at reducere antallet af rodstier i de frames, som sendes videre til delgraferne. Således ønsker vi at de frames, der knyttes til de konstruerede delgrafer, højst indeholder halvdelen af rodstierne i F samt rodstierne i S. For at opnå dette laves der om i den del af basisrekursionen, som finder den trekant, der definerer opdelingen. Vi ønsker, at den trekant vi finder, definerer en opdeling i op til tre dele, der hver højst indeholder halvdelen af bladene fra rodstierne i F. Til dette anvender vi algoritmen, som er beskrevet i kapitel 6.2, med den ændring at bladene i F får vægt 1 og alle andre knuder får vægt 0, når vi beregner størrelsen af delene. Denne metode giver os klart en separator, der deler grafen i delgrafer, der hver højst indeholder halvdelen af bladene i F. Hvilke stier, der bliver frames for delgraferne ser vi på i beviset for lemma 11.3.

Vi mangler nu at argumentere for, at den frame-reducering vi netop har beskrevet giver os mulighed for at reducere antallet af rodstier i framen.

Lemma 11.3 *Hvis vi bruger frame-reducering på en graf H med en frame F og denne frame-reducering giver os en separator S, vil hver af de konstruerede delgrafer kunne adskilles fra G_i af en frame, der højst indeholder halvdelen af rodstierne i F samt de op til tre rodstier fra S.*

Bevis. Rodstierne fra S inkluderes i framen til de delgrafer af H, som vi konstruerer. Vi ser nu på hvilke stier, der kan undværes i framen for en delgraf. Vi illustrerer undervejs på figur 11.2, hvor vi ser på et udsnit af grafen H og stien $Q \in F$.

Betragt stien $Q \in F$, der har knuden v_Q som blad. Hvis vi går fra v_Q mod roden på Q og finder et sted hvor Q møder S, da vil resten af Q være en del af S (billede (A)). Specielt vil hele Q være en del af S hvis $v_Q \in S$ (billede (B)). Stien Q er kun nødvendig for delgrafer, der indeholder bladet v_Q.

Dette skyldes, at det kun er i delgrafer, der indeholder v_Q, at det kun er Q, der adskiller dem fra andre dele. For en delgraf er det højst halvdelen af stierne i F, der er nødvendige, da der i en delgraf højst findes halvdelen af bladene fra stier i F.

På billede (A) ses et eksempel, hvor stien Q skal være med i framen for delen H_1, da Q's blad v_Q er i H_1. Stien Q er ikke relevant for H_2 idet den ikke adskiller H_2 fra dele som S ikke adskiller H_2 fra. På billede (B) er Q's blad ikke i H_1 og vi ser at Q heller ikke er nødvendig for H_1.

Vi kan konkludere, at vi kan lave en frame for hver delgraf H' af H, som adskiller H' fra G_i og at framen højst indeholder halvdelen af stierne fra F samt stierne fra S. □

En sti, som ikke kan undværes, og derfor *skal* med i framen for en delgraf H_1, kalder vi en relevant sti for H_1.

11.1.3 Antallet af rodstier i en frame

Vi kan nu vise, at en graf H kan adskilles fra G_i vha. en frame F, der indeholder et konstant antal rodstier. Vi viser, at den alternerende rekursion vil resultere i frames, der indeholder et konstant antal rodstier.

Lemma 11.4 *Hvis vi anvender alternerende rekursion, vil den frame F, der er knyttet til grafen H maksimalt indeholde 12 rodstier.*

Bevis. Vi viser ved induktion, at vi efter en frame-reducering højst har 9 rodstier i vores frame. Dermed kan vi efter en delgraf-reducering højst have 12 rodstier, da der tilføjes tre rodstier i en delgraf-reducering.

Vi viser følgende udsagn ved induktion: Efter den i'te iteration af en delgraf-reducering og en frame-reducering vil framen for delgraferne højst indeholde 9 stier.

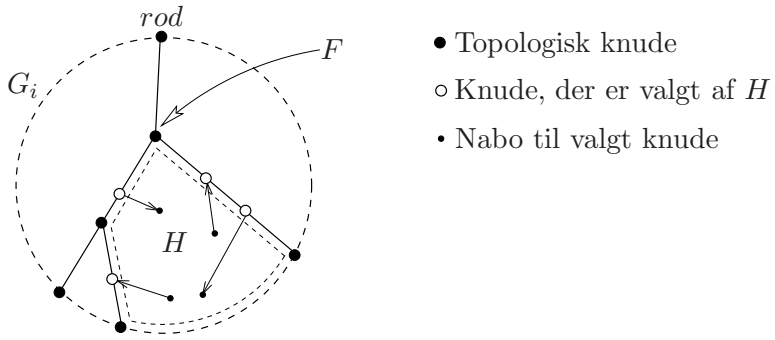
Basis: I den første graf er der ingen stier i framen, og dermed er udsagnet opfyldt.

Antag nu at en graf H har en frame F af størrelse $f \leq 9$. Efter en delgraf-reducering har vi $f' \leq f+3$ rodstier og efter den efterfølgende frame-reducering har vi $f'' \leq f'/2 + 3 = (f+3)/2 + 3 \leq 9$.

Vi kan dermed konkludere, at der i framen F for en graf H højst er 12 rodstier. □

11.1.4 Reachability over frames og separatorer

Vi vil nu se på, hvordan vi finder de nødvendige forbindelser for en delgraf H, som adskilles fra G_i af en frame F, og som opdeles af en separator S. De nødvendige forbindelser er, som vi konstaterede i afsnit 11.1.1, alle forbindelser over G_i mellem knuder i H og stierne i $F \cup S$. En sti i en frame eksisterer altid i forælderkaldet enten som separatorsti eller som framesti. Disse stier sendes med ned i rekursionen, således at man kun beregner forbindelser en gang for hver sti. Dette sker i det første kald, hvor stien er med, dvs. i det kald, hvor stien



Figur 11.3: Illustration af knuder på F, der er topologiske eller valgte af H.

er med i separatoren. Forbindelserne mellem H og F over G_i antages altså at være beregnet tidligere og de kan derfor sendes med ned i rekursionen. Disse forbindelser indsætter vi som kanter i $H + F$ og vi kalder den resulterende graf $H \star F$. Bemærk, at $H \star F$ ikke nødvendigvis er planar.

Faktum 11.5 *Lad knuderne v og w i H være givet. Da kan v nå w i G_i hvis og kun hvis v kan nå w i $H \star F$, da alle forbindelser mellem F og H over G_i findes i $H \star F$.*

Vi vil nu vise, at antallet af ekstra indsatte kanter i $H \star F$ er lineært i størrelsen af H.

Lemma 11.6 *Antallet af de ekstra kanter i $H \star F$ (ift. $H + F$) er lineært i antallet af knuder i H.*

Bevis. Hver knude i H har en forbindelse til og fra hver orienteret sti i F. Da der er et konstant antal stier i F, vil der være et konstant antal ekstra kanter for hver knude i H. □

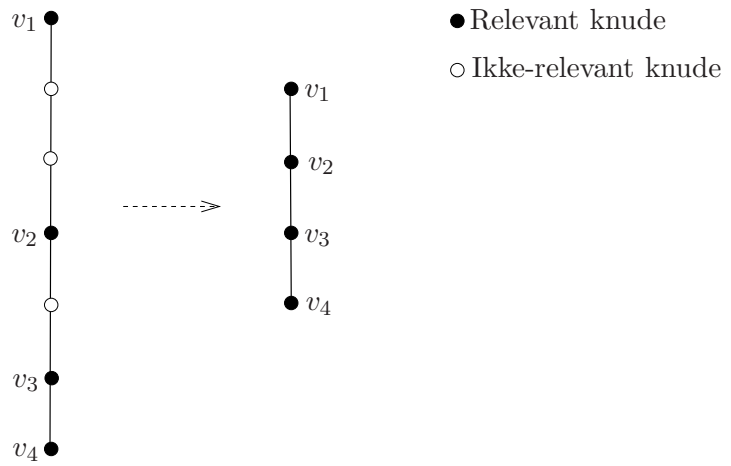
Grafen $H \star F$ tilføjer et antal ekstra kanter til $H + F$, der er lineært ift. H. Problemet er, som tidligere konstateret, størrelsen af $H + F$, der gør, at $H \star F$ ikke har den ønskede størrelse. Vi vil derfor modificere $H \star F$ således at størrelsen reduceres.

Vi starter med at definere et antal begreber, som vi anvender i vores modificering af $H \star F$.

Definition 11.7 *En knude i en frame F er* topologisk*, hvis den er et endepunkt på en af stierne i F eller hvis den er et forgreningspunkt for stier i F. Se figur 11.3 for en illustration.*

Vi bemærker, at der er et konstant antal topologiske knuder, da der er et konstant antal rodstier i F.

Definition 11.8 *En knude i F er* valgt af H *hvis den i $H \star F$ er nabo til en knude i H. Se figur 11.3 for en illustration.*



Figur 11.4: En sti i framen F for grafen H sammentrækkes således at det kun er knuder, der er relevante for H, der beholdes.

For hver knude, der er valgt af H eksisterer der en kant mellem H og F. Antallet af valgte knuder er derfor lineært i antallet af kanter i G_i, der er incidente til knuder i H.

Definition 11.9 *Hvis en knude i F enten er topologisk eller valgt af H, siger vi at knuden er* relevant for H.

Når vi skal finde stier mellem knuder i H og S over G_i er det netop stier over de valgte knuder vi benytter. De topologiske knuder er interessante idet de definerer strukturen af framen.

Vi vil nu vise, at vi kan modificere $H \star F$ således at størrelsen af denne reduceres. Vi modificerer $H \star F$ ved at fjerne de knuder, som det ikke er nødvendigt at beholde, dvs. knuder, der *ikke* er relevante.

Lemma 11.10 *Vi kan reducere størrelsen af $H \star F$, så denne er lineær i antallet af kanter i G_i, der er incidente til knuder i H.*

Bevis. Vi fravælger de knuder på F, der hverken er topologiske eller valgte af H, dvs. vi beholder kun de knuder i F, der er relevante for H. Hvis vi har en sti fra F, trækkes alle segmenter, der ikke indeholder en relevant knude, sammen til en enkelt kant. Vi ser et eksempel på dette på figur 11.4, hvor en sti sammentrækkes så den kun indeholder relevante knuder samt kanter mellem disse.

Vi ved, at der er et konstant antal topologiske knuder samt at antallet af knuder, der er valgt af H er lineært i antallet af kanter i G_i, der er incidente med knuder i H, og dermed har $H \star F$ den ønskede størrelse. □

Vi kan nu finde alle forbindelser i G_i mellem knuder i H og separatoren S, ved at bruge konstruktionen fra lemma 8.6 på $H \star F$. Vi erindrer at vi i denne konstruktion for hver sti Q gør følgende. Vi gennemløber Q bagfra og

vha. bredde først-gennemløb findes de knuder, der kan nås fra knuder på Q. På tilsvarende vis findes knuder, der kan nå Q.

Den reducerede frame (dvs. den frame, der kun indeholder topologiske og valgte knuder) konstrueres på vej ned gennem rekursionen. Derfor skal vi, før vi kalder rekursivt på delgrafer af grafen H med framen F, konstruere den reducerede frame for hver af disse delgrafer.

Vi starter med at finde ud af hvilke stier i $F \cup S$, der skal med i framen for en delgraf og vi markerer hvilke stier, der er relevante for hver delgraf. Vi markerer ligeledes hvilke knuder i stierne, der er relevante for hver delgraf. Herefter løbes hver sti i $F \cup S$ igennem en ad gangen. For en sti $Q \in F \cup S$ starter vi med at registrere Q's startpunkt ved de delgrafer, som Q er relevant for. Herefter gennemløbes knuderne i Q og for hver knude $v \in Q$ ser vi på de delgrafer, som v er relevant for. Ved hver delgraf registreres den sidste knude på Q, som er indsat i den reducerede version Q', denne knude kalder vi u. Når v nu skal indsættes i Q' tilføjer vi blot v samt en kant (u, v) til Q'. Vi husker nu på v som den sidste knude, der er indsat i Q'.

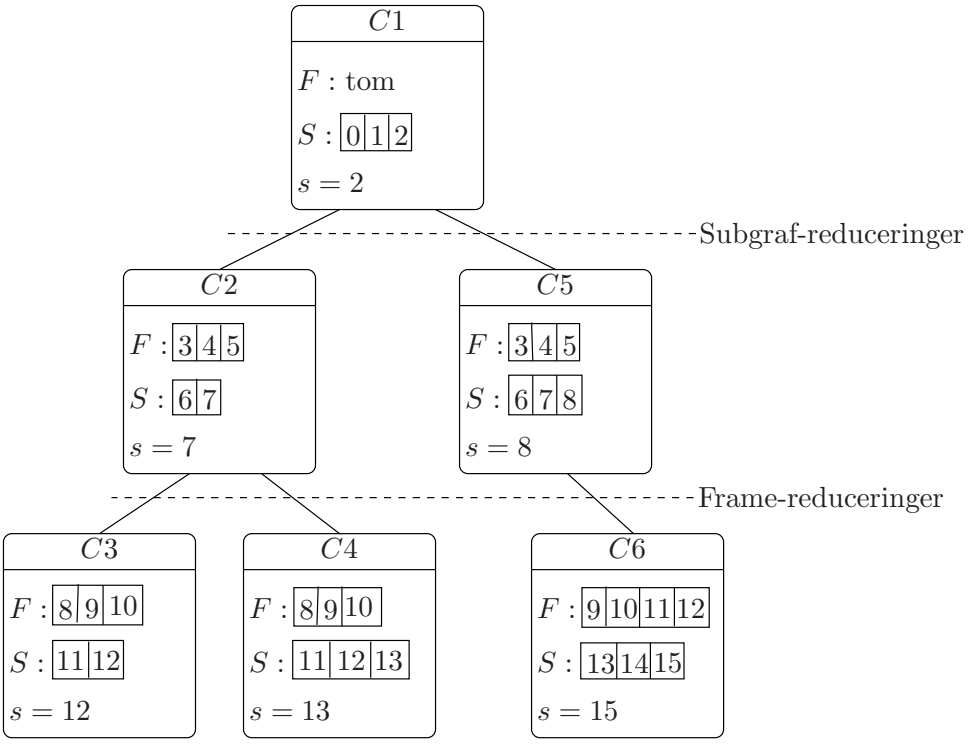
Konstruktionstiden for de reducerede frames for delgraferne er lineær i størrelsen er $H \star F$ da vi gennemløber $F \cup S$ og anvender konstant tid i hver knude. Udførelsestiden for et skridt i rekursionen er dermed stadig lineær. Den samlede udførelsestid forbliver $\mathcal{O}(n \log(n))$, da rekursionsdybden er logaritmisk idet vi halverer antallet af knuder i grafen i hvert andet rekursive kald.

11.1.5 Indeksering af stier og forespørgsler

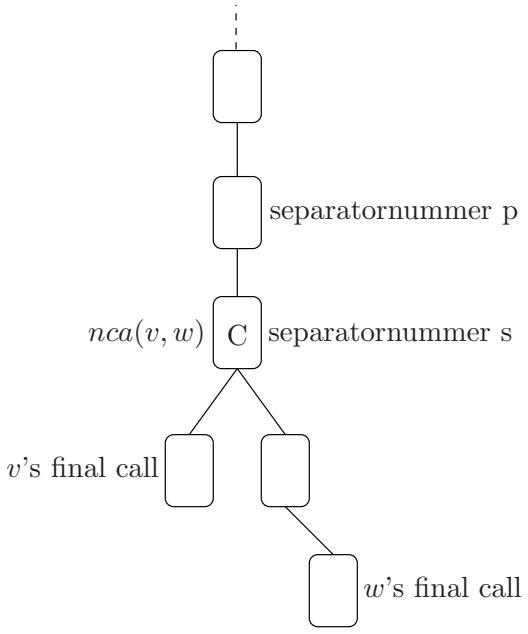
Stierne, som deler en delgraf H fra resten af G_i skal indekseres således at vi kan finde ud af, om en knude v kan nå en anden knude w, ved at se på de stier, der deler v fra w i G_i. Da knuder kan nå hinanden over enten en sti i en frame eller i en separator, skal begge slags stier indekseres. Dette betyder, at en sti, som har fået et indeks som separatorsti i et kald, kan blive nummereret som frame i de næste kald og en sti kan dermed blive nummereret flere gange. Vi erindrer, at for hver rodsti i S og F er der op til to orienterede stier, der skal nummereres, da en rodsti består af to orienterede dele.

Hvert rekursionskald C får igen et *separatornummer* s, som svarer til det sidste nummer, der er anvendt til at nummerere stier i C. På figur 11.5 ses et rekursionstræ, hvor hvert kald har et antal framestier, et antal separatorstier, samt et separatornummer. Her ses tre stier, der er separatorstier i kaldet $C1$, som i de to kald $C2$ og $C3$ er blevet til framestier, og derfor er blevet nummereret igen.

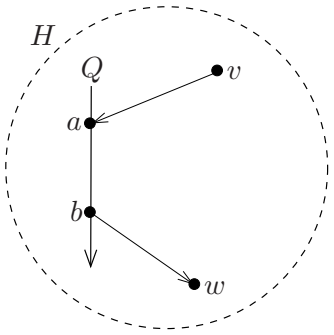
De stier, som deler v fra w i G_i er netop de stier, der findes i $F \cup S$ i den sidste delgraf H, der både indeholder v og w. Denne graf H stammer fra kaldet C, hvis tilsvarende knude i rekursiontræet er den nærmeste fælles forfader $nca(v, w)$ for v's final call og w's final call (vi erindrer, at final call er det kald, hvor en knude er med i separatoren og dermed det sidste kald, en knude er med i). I $nca(v, w)$ findes et separatornummer s, som er det sidste nummer, som en sti i $F \cup S$ har fået. For at finde det første nummer på en sådan sti, ser vi i forælderknuden til $nca(v, w)$, hvor vi ligeledes finder et separatornummer p. Da p er det sidste nummer, der er anvendt tidligere, må $p+1$ være det første nummer, der er givet



Figur 11.5: Rekursionstræ, hvor der er anvendt frames og alternerende rekursion.



Figur 11.6: Vi skal se på de stier, der findes i kaldet C, for at finde ud af om v kan nå w. Vi skal altså se på stier med numrene $p+1, \ldots, s$.



Figur 11.7: Når v kan nå w finder vi en separatorsti Q i grafen H over hvilken v når w vha. to forbindelser (a, v) og (b, w).

til en sti i $F \cup S$. For at finde ud af, om v når w i G_i, skal vi altså se på, om v når w over en af stierne med nummer $p+1, \ldots, s$. Se en illustration på figur 11.6. Vi bemærker, at metoden til at finde ud af om v når w over en sti Q foretages ved at kigge i ud_v og ind_w, som det sker i den oprindelige konstruktion, se afsnit 9.2. I specialtilfældet hvor $nca(v, w)$ er roden i rekursionstræet, skal vi se på stierne $0, \ldots, s$.

Vi skal altså se på et konstant antal stier under en forespørgsel, hvilket tager konstant tid. Det er muligt for hele forespørgslen at tage konstant tid, hvis man kan finde den nærmeste fælles forfader $nca(v, w)$ i rekursionstræet i konstant tid. En sådan konstruktion er mulig ifølge [HT84].

11.2 Stier mellem knuder

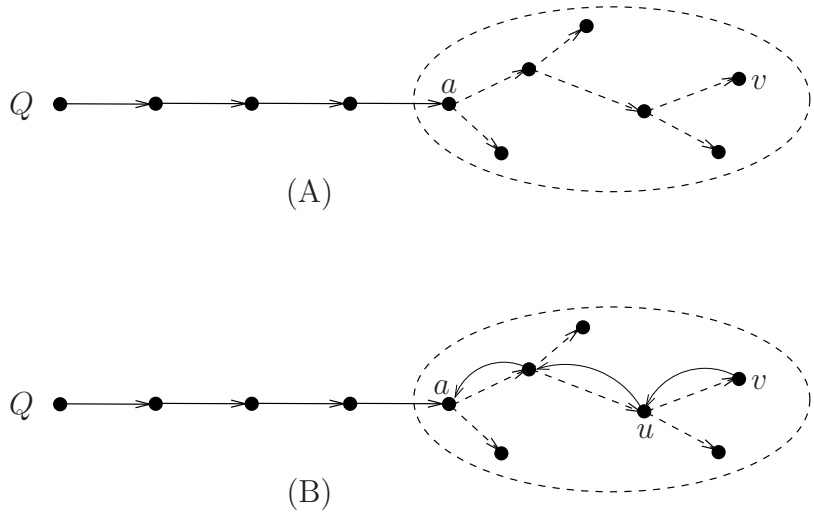
En udvidelse, der kunne laves til vores program, var at sørge for, at vi kunne angive en sti fra v til w, hvis en sådan eksisterer. Vi vil nu beskrive hvorledes en sti kan findes, givet at v kan nå w.

Antag at vi har fundet ud af, at v kan nå w i en af vores tolagsgrafer G_i. Da har vi i vores forespørgsel fundet en sti Q med indeks q i en delgraf H, over hvilken v når w vha. en forbindelse (v, a) og en forbindelse (b, w). Om a og b gælder, at de er knuder på Q og $\iota(a, Q) \leq \iota(b, Q)$ (vi erindrer at dette betyder, at a er den samme knude som b eller at a kommer før b på Q, se definition 8.5). Dette er illustreret på figur 11.7.

Vi ved, at alle de stier, der er i H også findes i G_i, derfor kan vi finde en sti fra v til w i H og den vil dermed være en sti i G_i. Vi vil nu vise, hvordan vi udvider datastrukturen således, at vi kan finde en sti, der svarer til en forbindelse mellem Q og H. I det følgende ser vi kun på forbindelser fra H til Q – forbindelser fra Q til H er symmetriske.

Vi ser på konstruktionen af forbindelserne i beviset for lemma 8.6. Hvis vi i denne konstruktion finder en sti fra a på Q til v så laver vi altid en forbindelse (a, v), og dette bliver da forbindelsen fra Q til v. Vi ser på figur 11.8 (A) hvordan vi har fundet en sti fra a til v vha. en søgning fra a.

Vi gemmer en pointer til den knude u, der er forgænger til v på stien fra



Figur 11.8: (A) Der er fundet en sti fra a på Q til v. Derfor har vi en forbindelse (a, v). (B) Vi har indsat pointere til forgængere i vores søgning således at stien fra v til a kan findes.

a i den søgning vi har lavet. Vi anvender q som indeks i en ny liste $\overrightarrow{ind}_v$ som gemmes ved v således at $\overrightarrow{ind}_v[q]$ indeholder en pointer til u. Når vi skal finde stien fra v til a kan vi følge pointerne fra v til u, fra u til dennes forgænger og så fremdeles indtil vi når a. På figur 11.8 (B) ser vi hvorledes pointere til forgængere fører fra v til a. Da vi bruger konstant tid ved hver knude på en fundne sti, kan denne findes i lineær tid i dens længde. I det symmetriske tilfælde, hvor vi laver forbindelser fra H til Q konstruerer vi listen $\overrightarrow{ud}_v$.

Vi har nu at hvis v når w over stien Q i en delgraf H via forbindelserne (v, a) og (b, w) kan vi sammensætte en sti fra v til w ved at finde tre delstier (se figur 11.9):

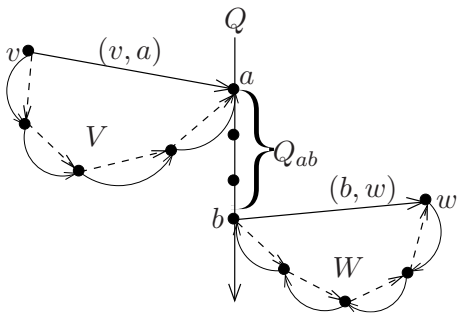
1. En sti V, der svarer til forbindelsen (v, a).
2. Stykket af Q, der ligger mellem a og b. Vi kalder dette Q_{ab}.
3. En sti W, der svarer til forbindelsen (b, w)

Stien $VQ_{ab}W$ er nu en sti fra v til w i grafen H og dermed også en sti fra v til w i G_i. Da hver del kan findes i lineær tid i delens længde, kan hele stien findes i lineær tid i dennes længde.

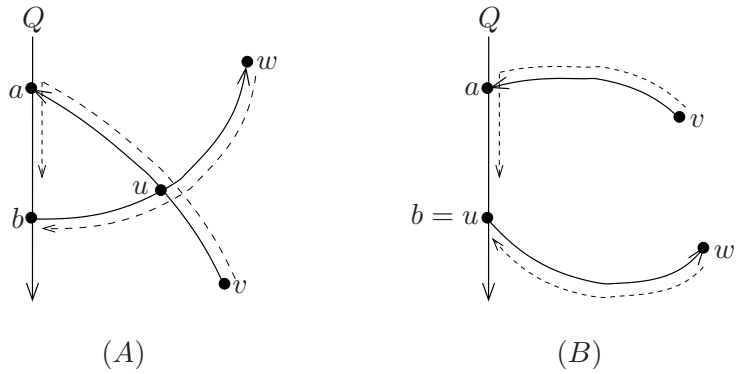
De nye lister, vi tilføjer, giver ikke et højere asymptotisk pladsforbrug, da vi i forvejen gemmer lister af størrelse $\mathcal{O}(\log(n))$ i hver knude og de lister vi nu tilføjer har ligeledes størrelse $\mathcal{O}(\log(n))$.

11.2.1 Simple stier mellem knuder

Det eneste problem ved ovenstående konstruktion er, at den sti vi finder ikke nødvendigvis er simpel – den skærer evt. sig selv (se figur 11.10 (A), hvor stien skærer sig selv i knuden u). Vi bemærker, at Q_{ab} ikke skærer sig selv, da den



Figur 11.9: Skitsering af hvorledes vi finder stykkerne V, Q_{ab} og W, som sættes sammen til en sti fra v til w.



Figur 11.10: (A) Stien fra v til w over Q krydser sig selv. (B) Stien fra v til w over Q krydser ikke sig selv.

stammer fra det udspændende træ for den graf H hvori Q findes. Det er altså stien fra v til a, der kan skære stien fra b til w.

For at finde en simpel sti, følger vi disse to delstier, hvilket er illustreret på figur 11.10:

1. Stien der går fra v til a og videre ned langs Q.

2. Baglæns fra w til b.

Hvis u er det første skæringspunkt vi møder mellem de to delstier, vil stien fra v til u til w være en simpel sti. Bemærk at hvis den oprindelige sti ikke skærer sig selv, så vil knuden u være den samme knude som b.

For at kunne finde den simple sti i tid, der er lineær i stiens længde, følger vi de to nævnte delstier parallelt og stopper, når vi møder skæringspunktet u.

Korollar 11.11 *Givet at knuden v når knuden w, kan vi finde en simpel sti fra v til w. Vi kan finde stien i lineær tid i dennes længde.*

Det er også muligt at finde stier mellem knuder, når man anvender frame-reducering. Denne konstruktion vil vi dog ikke komme ind på.

11.3 Distribution af information i labels

I dette afsnit vil vi skitsere, hvorledes vi kan samle den beregnede information i labels, som tilknyttes hver knude. For hver knude vil vi gemme et label af størrelse $\mathcal{O}(\log(n))$. Dette vil gøre det overflødigt at opbevare de rekursionstræer, der beregnes undervejs. Det asymptotiske pladsforbrug vil forblive $\mathcal{O}(n \log(n))$, men den konstante faktor vil blive mindre, da vi ikke gemmer de forskellige delgrafer, rekursionstræer osv.

Vi ser nu på, hvilke dele et label for knuden v skal indeholde for hver G_i af tolagsgraferne $G_0, \ldots, G_{k-1}$ knuden er med i:

1. Listerne ind_v og ud_v, som angiver de forbindelser, der er mellem v og separatorstierne i de delgrafer v er med i. Disse lister fylder $\mathcal{O}(\log(|\mathcal{V}(G_i)|))$

2. I den oprindelige konstruktion opbevares separatornumrene i rekursionstræet. I v laver vi nu en liste sep_v, der indeholder separatornumrene for knuderne i rekursionstræet på stien fra roden til v's final call. Indgangen $sep_v[d]$ indeholder separatornummeret for den knude på stien, som findes i dybde d, se figur 11.11. Da højden af træet er logaritmisk fylder denne liste ligeledes $\mathcal{O}(\log(|\mathcal{V}(G_i)|))$.

3. Et label, der indeholder information, der skal anvendes til at finde dybden af nærmeste fælles forfader for to knuder. Vi ønsker derfor at sætte et label på hver rekursionstræsknude, der har størrelse $\mathcal{O}(\log(|\mathcal{V}(G_i)|))$ og som giver os mulighed for at finde dybden af nærmeste fælles forfader i tid $\mathcal{O}(\log(|\mathcal{V}(G_i)|))$. Hvorledes dette opnås forklares kort i afsnit 11.3.1.

   Hvis man anvender frame-reducering, skal man finde nærmeste fælles forfader i konstant tid. I [AGKR02] findes en algoritme, der vha. af labels på hver knude af størrelse $\mathcal{O}(\log(|\mathcal{V}(G_i)|))$, finder nærmeste fælles forfader i konstant tid.

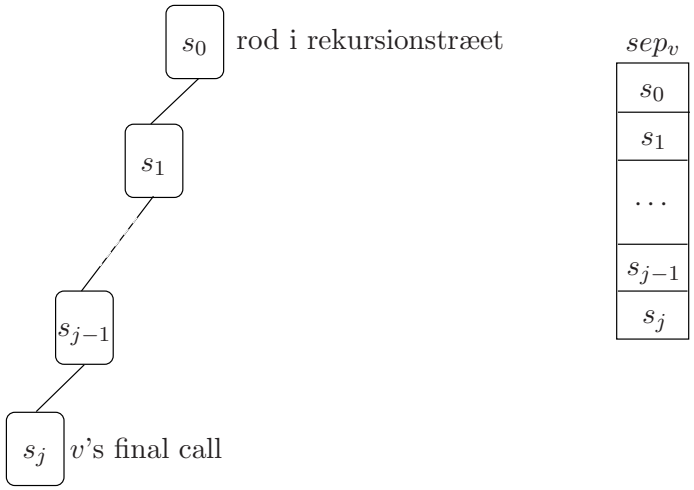
Vi kan konkludere at størrelsen af et label for en knude v der er i en tolagsgraf G_i er $\mathcal{O}(\log(|\mathcal{V}(G_i)|))$.

En knude skal nu have et label tilføjet for hver af tolagsgraferne $G_0, \ldots, G_{k-1}$ den er med i. En knude skal ligeledes vide hvilket lag den tilhører og dermed hvilke af $G_0, \ldots, G_{k-1}$ den er med i. Da en knude v højst er med i to af graferne $G_0, \ldots, G_{k-1}$, vil det samlede label indeholde et lagnummer samt to af de ovenfor beskrevne labels. Det samlede label for v har derfor størrelse $\mathcal{O}(\log(n))$, da den samlede størrelse af $G_0, \ldots, G_{k-1}$ er $\mathcal{O}(n)$ ifølge sætning 4.6.

11.3.1 Nærmeste fælles forfader

Vi beskriver her kort, hvorledes vi kan konstruere labels, der kan anvendes til at finde dybden af nærmeste fælles forfader. Vi skal lave labels for det rekursionstræ, der svarer til en tolagsgraf G_i. Hver knude i G_i får tildelt det label, der svarer til knudens final call i rekursionstræet.

Vi laver en nummerering af knuderne i rekursionstræet, hvor vi anvender to bits til hver knude. Vi starter i roden, som nummereres med nummer `01`. For



Figur 11.11: Rekursionstræ, hvor separatornumrene er gemt i knuderne og den tilsvarende liste sep_v, der tilhører v's label.

hver knude i træet nummererer vi nu knudens børn. Grunden til at vi anvender to bits er, at hver knude i rekursionstræet har op til tre børn. Hvis der er tre børn, får de nu bitsene `01`, `10` og `11`. På figur 11.12 ses i nummereringen – øverst i hver knude ses de bits, rekursionstræsknuden er blevet tildelt.

Hver rekursionstræsknude får nu et label, hvor alle dens forfædres bits er gemt (inkl. dens egne bits). Dette label fylder $\mathcal{O}(\log(n))$ bits, da der er $\mathcal{O}(\log(n))$ forfædre, der hver bidrager med to bits.

Vi kan nu givet to knuder v og w finde dybden af nærmeste fælles forfader for de to knuders final call $nca(v, w)$, ved at se på de to knuders labels (der indeholder det label, der svarer til knudernes final call i rekursionstræet). Dybden af $nca(v, w)$ findes på følgende måde:

- Start ved den 0'te indgang i listerne og løb dem parallelt igennem. Den første indgang vil være ens (svarer til roden i træet).
- Stop når der findes to indgange, der ikke er ens. Dybden af $nca(v, w)$ svarer til nummeret på den sidste indgang, der er ens for de to lister.

Et eksempel kan ses på figur 11.12. Da listerne, der skal løbes igennem, har længde $\mathcal{O}(\log(n))$, tager det tid $\mathcal{O}(\log(n))$ at finde dybden af den nærmeste fælles forfader.

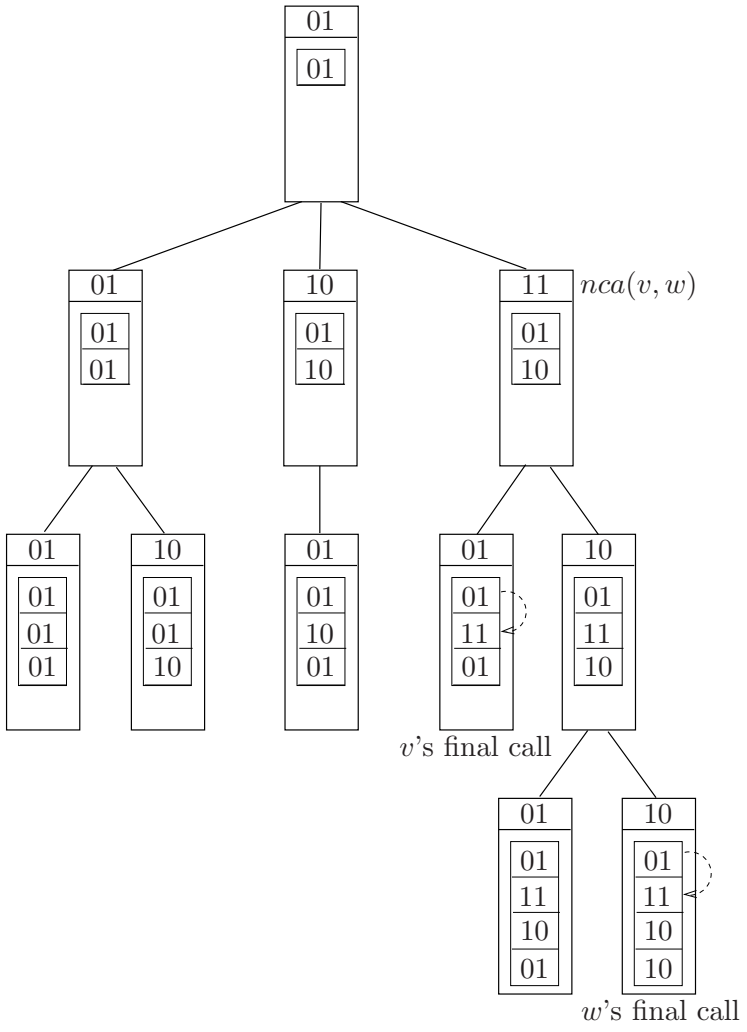
11.3.2 Forespørgsel ved hjælp af labels

Når en forespørgsel på, om en knude v kan nå en knude w skal besvares vha. labels, starter vi med at slå op i v og w, i hvilke lag knuderne findes. Dermed ved vi hvilke af graferne $G_0, \ldots, G_{k-1}$ knuderne findes i. Har de en eller to sådanne grafer til fælles gør vi følgende for hver sådan graf, G_i (vi bemærker at grafen G_i ikke gemmes, men labelen er lavet med udgangspunkt i G_i):

- Find de labels i hhv. v og w, der hører til G_i.

- Slå dybden af den nærmeste fælles forfader op.
- Find separatornummeret for den nærmeste fælles forfader. Dette kan slås op i enten sep_v eller sep_w.
- Gennemløb de fælles separatorstier, som beskrevet i afsnit 9.2.2 for at se om v når w over en af disse.

Finder vi i en af de fælles grafer en sti Q over hvilken v når w, returneres `true` ellers returneres `false` som i den originale forespørgsel i afsnit 9.2.



Figur 11.12: Opbygningen af labels, der benyttes når nærmeste fælles forfader skal findes.

Kapitel 12

Konklusion

Vi har i denne del af specialet givet en grundig teoretisk gennemgang af de algoritmer og datastrukturer, der indgår i konstruktionen af et reachability-orakel, som er beskrevet i artiklen *Compact Oracles for Reachability and Approximate Distances in Planar Digraphs* [Tho01] af Mikkel Thorup.

Vi har gennem detaljeret beskrivelse og illustrationer gjort teorien mere tilgængelig og vi har tilført de detaljer og opklaringer vi mener, der skal til for at øge forståelsen. For at skabe en større intuition om de forskellige konstruktioner, har vi motiveret de forskellige konstruktioner og algoritmer vi har indført, så læseren har en fornemmelse af, hvorfor en sådan indføres samt hvornår og hvordan denne skal anvendes. Vi har vha. illustrationer, motivationer og opsummeringer givet et overblik over konstruktionen af et reachability-orakel.

Vi har tilført detaljer til beviser for de forskellige konstruktioner og algoritmers effektivitet og korrekthed for at præcisere beviserne. En del af algoritmerne er forholdsvist simple, hvorimod korrektheden er ikke er indlysende, hvorfor vi har tilføjet detaljer og illustrationer for at tydeliggøre korrektheden.

Vi mener vi har opnået vores mål om at tydeliggøre teorierne bag konstruktionen af et reachabilityorakel og har gjort stoffet mere tilgængeligt. Vi har som ønsket tilført de mange ikke-trivielle detaljer, der i artiklen [Tho01] er overladt til læseren.

Vi har selv bidraget med en mindre ændring, idet vi har tilføjet en rod til en tolagsgraf, for at undgå et specialtilfælde, som er beskrevet i artiklen [Tho01]. Vi har ligeledes bidraget med en udredning af de specialtilfælde, der er i den algoritme, der anvendes til at finde en opdelingstrekant og har tilføjet et bevis, der tager højde for disse specialtilfælde.

Vi har lavet en implementation af reachability-oraklet og har på denne måde vist, at implementation er mulig. Vi har beskrevet vores implementation og på den måde tydeliggjort hvilke overvejelser og detaljer, der skal føjes til det teoretiske grundlag for at kunne implementere de algoritmer og datastrukturer vi har gennemgået. Implementationen er omfattende, men en del af algoritmerne er forholdsvist simple at implementere.

Vi har i løbet af implementationsfasen inspiceret de mellemresultater og de datastrukturer, der konstrueres undervejs, for at få overblik over implementationen og for at sandsynliggøre at de forskellige dele af vores implementation er korrekte.

Vi har testet den færdige implementation og har derigennem sandsynliggjort at de beregnede resultater er korrekte ved at teste op mod en kendt korrekt algoritme. Vi har ligeledes lavet test, der sandsynliggør de asymptotiske grænser på ressourceforbruget, vi har vist i den teoretiske gennemgang af reachability-oraklet. Vores test viser tendenser, der følger de asymptotiske grænser.

Vi har som nævnt inspiceret mellemresultater og datastrukturer, der er opbygget undervejs. Dette betyder at alle strukturer, der er opbygget undervejs gemmes i vores reachability-orakel. Vores pladsforbrug har derfor et stort overhead. Hvis oraklet skal være praktisk anvendeligt for store grafer skal pladsforbruget reduceres. Reduktionen kan f.eks. foretages ved at indføre distribution af information som labels på knuderne, hvilket vil betyde en begrænsning på pladsforbruget.

Vi har givet en teoretisk gennemgang af tre forbedringsmuligheder, der kan danne baggrund for fremtidigt arbejde på reachability-oraklet. I gennemgangen af forbedringsmulighederne har vi igen givet motivationer, illustrationer og detaljer, der har tydeliggjort teorien og skabt større forståelse og overblik.

Vi kan konkludere, at vi har nået vores mål, idet vi har fået præsenteret teorien på en tilgængelig måde. Vi har implementeret et reachability-orakel, og har dermed vist, at dette er muligt og beskrevet hvad denne implementation involverer. Vi har beskrevet hvilke muligheder, der er for fremtidige udvidelser, samt hvorledes programmet kan gøres mere anvendeligt i praksis. Vi har selv lavet små bidrag, men vores hovedbidrag er tydeliggørelsen af teorien, udfyldelse af de til læseren overladte detaljer samt at vi har vist, at implementation af et reachability-orakel er mulig.

Del II

Approksimerede afstande

Kapitel 13

Approksimerede afstande i planare grafer

Givet en orienteret, planar og sammenhængende graf G, med n knuder og ikke-negative heltallige vægte på kanterne, ønsker vi at konstruere en datastruktur, der for to knuder i G kan beregne en approksimativ afstand mellem knuderne. Den approksimative afstand skal være konservativ og ligge inden for en faktor $(1+\varepsilon)$ fra den korteste afstand mellem knuderne. Datastrukturen ønsker vi at konstruere i tid $\mathcal{O}(n\log^3(n)\log(nW)/\varepsilon)$ med et pladsforbrug, der er $\mathcal{O}(n\log(n)\log(nW)/\varepsilon)$ og forespørgsler skal kunne besvares i tid $\mathcal{O}(\log(\log(nW))\log(n)+\log(n)/\varepsilon)$, hvor W er vægten på den tungeste kant. Vi kalder datastrukturen et afstandsorakel.

I denne del ser vi igen på orienterede planare grafer, men der er nu ikke-negative heltallige vægte på kanterne. Vi vil konstruere et afstandsorakel for en graf G, der givet to knuder v og w i G, kan beregne en approksimation på afstanden mellem knuderne. Hvis vi kalder den korteste afstand mellem v og w for $\delta(v,w)$ og den approksimative afstand for $\delta_r(v,w)$, ønsker vi at $\delta(v,w) \leq \delta_r(v,w) \leq (1+\varepsilon)\delta(v,w)$.

I de næste afsnit vil vi give en kort oversigt over de forskellige faser i konstruktionen af et afstandsorakel for en graf G. Et afstandsorakel konstrueres vha. et antal mindre orakler, som vi kalder α-orakler. Et α-orakel konstrueres med udgangspunkt i G, hvor man kun betragter kanter med vægt op til α. Givet to knuder v og w i G, kan et α-orakel give en approksimativ afstand mellem v og w, der højst har en additiv fejl på $\varepsilon\alpha$, givet at $\delta(v,w) \leq \alpha$.

Vi vil i de næste kapitler introducere faserne i opbygningen af et α-orakel. Her antager vi, at vægten på kanterne i grafen vi arbejder med, højst er α. Hovedideen er at dekomponere grafen vha. separatorer som i del I, hvor vi konstruerede reachability-orakler. Konstruktionen af et α-orakel minder en del om konstruktionen af et reachability-orakel og vi kan derfor anvende nogle af konstruktionerne fra reachability-oraklet, når vi konstruerer α-orakler, med den forskel, at vi nu skal tage kanternes vægt i betragtning. Vi starter igen med at dele grafen op i lag og konstruere et antal mindre grafer. Denne fase introduceres i afsnit 13.1. Herefter dekomponeres grafen vha. separatorer – denne fase introduceres i afsnit 13.2. Vi finder igen forbindelser over separatorstier, men der skal nu tages højde for vægten på kanterne. Vi introducerer konstruktionen

af forbindelser i afsnit 13.3. I afsnit 13.4 og afsnit 13.5 introducerer vi hhv. samlingen af et α-orakel samt forespørgsler i et α-orakel.

Givet at vi kan konstruere α-orakler, kan vi konstruere et samlet afstandsorakel, der kan give den approksimative afstand vi ønsker. Det samlede afstandsorakel består af to reachability-orakler samt et antal α-orakler. I afsnit 13.6 beskrives kort for hvilke værdier af α vi konstruerer α-oraklerne og hvordan G forberedes således at et α-orakel kan konstrueres. Vi introducerer det samlede orakel samt forespørgsler heri i afsnit 13.7.

I afsnit 13.8 vil vi kort introducere vores eksperimentelle evaluering af vores implementation af et afstandsorakel.

13.1 Reduktion til $(3, \alpha)$-lagsgrafer

Antag, at vi er givet en planar, orienteret graf G med ikke-negative heltallige vægte på kanterne, der alle højst er α. Som i del I opdeler vi grafen i lag, men vi begrænser den længste sti i en lag til højst at være af længde α. For hver tre på hinanden følgende lag, konstruerer vi en $(3, \alpha)$-lagsgraf, der indeholder knuderne i disse tre lag. En $(3, \alpha)$-lagsgraf indeholder et $(3, \alpha)$-lags-udspændende træ T. En rodsti i T er sat sammen af 3 minimale orienterede stier af længde højst α. Konstruktionen beskrives i detaljer i kapitel 14, hvor vi endvidere vil bevise egenskaber for $(3, \alpha)$-lagsgraferne.

13.2 Opdeling og rekursion

Givet en $(3, \alpha)$-lagsgraf H laver vi en hierarkisk dekomposition af denne stort set som beskrevet i kapitel 5-7. Vi finder en trekant, hvor rodstierne fra knuderne definerer en opdeling af grafen i tre dele, der hver højst indeholder halvdelen af knuderne i H. Efterfølgende kalder vi rekursivt på hver af de tre dele. Konstruktion og de få ændringer i forhold til del I er beskrevet i kapitel 15.

13.3 Forbindelser over separatorstier

Vi anvender som tidligere forbindelser over separatorstier, når vi nu skal repræsentere approksimerede afstande. Da der nu er vægte på kanterne, skal der ligeledes være vægte på forbindelserne. Hvor det før udelukkende var nødvendigt at gemme én forbindelse hver vej for hvert par bestående af en knude og en separatorsti, må vi nu gemme flere. Grunden til dette er, at det ikke længere er nok at vide, at der er en vej, vi vil nu have den korteste. Da vi ikke ønsker at gemme alle de forbindelser, som det er muligt at konstruere, da det vil koste både i pladsforbrug og tidsforbrug for forespørgsler, ønsker vi at reducere antallet af forbindelser, der er mellem en sti og en knude. Vi går derfor på kompromis med præcisionen til gengæld for lavere forbrug på de nævnte områder.

I kapitel 16 introduceres konstruktioner, der gør os i stand til at gemme et antal forbindelser, der kan anvendes til at finde en approksimeret afstand, der er konservativ og højst $\varepsilon\alpha$ for stor i forhold til den korteste afstand. Vi gennemgår hvorledes vi konstruerer mængder af forbindelser, hvorledes disse gøres mindre

samt hvorledes de kan anvendes til at finde en approksimeret afstand. Størrelsen af en mængde af forbindelser mellem en knude og en separatorsti vil være $\mathcal{O}(1/\varepsilon)$.

For at konstruere alle mængder af forbindelser i et rekursivt kald, anvender vi tid $\mathcal{O}(|\mathcal{V}(H)|\log^2(|\mathcal{V}(H)|)/\varepsilon)$. Beviset for, hvor lang tid vi anvender på at konstruere forbindelser, har vi selv konstrueret. I artiklen [Tho01] antages, at man anvender en lineær-tids *single source shortests paths*-algoritme, hvilket vi ikke antager.

13.4 Samling af et α-orakel

Når alle dele af konstruktionen af et α-orakel er gennemgået i kapitel 14-16, opsummerer vi i kapitel 17, hvorledes vi samlet set konstruerer α-orakler. Vi afrunder dette med at argumentere for tidsforbruget samt pladsforbruget for et α-orakel. Tidsforbruget for konstruktionen af et α-orakel er $\mathcal{O}(n\log^3(n)/\varepsilon)$ og pladsforbruget er $\mathcal{O}(n\log(n)/\varepsilon)$.

13.5 Forespørgsler i et α-orakel

I kapitel 17 vil vi beskrive, hvorledes vi foretager forespørgsler i et α-orakel. Vi skal igen se på forbindelser over separatorstier, men denne gang skal vi gennemløbe mængder af forbindelser over en separatorsti og finde den korteste. Vi konkluderer i kapitel 17, at en sådan forespørgsel kan foretages i tid $\mathcal{O}(\log(n)/\varepsilon)$. Vi afrunder kapitel 17 med at beskrive begrænsningerne for de resultater vi kan få, når vi anvender et α-orakel. Begrænsningerne er, at vi kun kan forvente at få en approksimeret afstand mellem knuder v og w, hvis den reelle korteste afstand mellem knuderne højst er α. Desuden er den approksimerede afstand, der returneres, op til $\varepsilon\alpha$ for stor i forhold til den korteste afstand.

13.6 Valg af værdier for α

Når vi kan konstruere α-orakler erindrer vi, at disse har en begrænsning mht. til stilængde og præcision. Vi skal derfor konstruere et antal α-orakler, som tilsammen dækker alle stilængder. I disse kan man vha. en række velvalgte forespørgsler komme frem til et resultat, der er så præcist som vi ønsker.

I kapitel 18 ser vi på, hvorledes vi kan sammensætte et antal α-orakler, således at vi bliver i stand til at give en approksimeret afstand for ethvert par af knuder. Vi beskriver for hvilke værdier af α hhv. ε, vi konstruerer α-orakler. Vi konstruerer to grupper af α-orakler med forskellig præcision. Vi anvender hhv. $\varepsilon' = 1/2$ og $\varepsilon' = \varepsilon/4$ i de to typer orakler. De valgte α-værdier er for begge typer orakler potenser af to op til $2^{\lceil\log(nW)\rceil}$, hvilket betyder, at alle stilængder er dækket. Vi konstruerer da i alt $2 * \lceil\log(nW)\rceil$ α-orakler, hvor W er den højeste vægt på en kant i G.

Vi viser, hvorledes vi modificerer den graf G vi er givet, således at den kun indeholder kanter med vægt op til α, men stadig er sammenhængende og planar. Modificeringen består i indsættelse af en knude og et antal kanter.

Vi anvender ligeledes vores konstruktion af et reachability-orakel i vores afstandsorakel. Vi anvender reachability-orakler til at afgøre om en vej eksisterer og til at finde afstande med vægt 0.

Vi viser, hvorledes vi kan lave en samlet datastruktur, der kan sikre os en approksimeret afstand, der er konservativ og ligger inden for en faktor $(1+\varepsilon)$ fra den rigtige afstand. Vi argumenterer for tidsforbruget samt pladsforbruget for denne samlede datastruktur. Det samlede tidsforbrug for at konstruere et afstandsorakel er $\mathcal{O}(n\log^3(n)\log(nW)/\varepsilon)$ og pladsforbruget for det samlede afstandsorakel er $\mathcal{O}(n\log(n)\log(nW)/\varepsilon)$.

13.7 Samlet orakel og forespørgsler

Vi afslutter kapitel 18 med at opsummere indholdet af det samlede afstandsorakel samt at beskrive, hvorledes vi foretager forespørgsler i dette samlede orakel. Disse foretages ved at lave et antal forespørgsler i de α-orakler og reachability-orakler, det samlede orakel er sammensat af. Vi har som nævnt to grupper af α-orakler og vi anvender en binær søgning i den ene mindst præcise gruppe til at finde det orakel i den anden mere præcise gruppe, der kan give os det resultat, vi søger. Vi argumenterer for, at den samlede forespørgselstid er $\mathcal{O}(\log(\log(nW))\log(n)+\log(n)/\varepsilon)$. Vi kommer sidst i kapitlet kort ind på muligheder for forbedringer af afstandsoraklet.

13.8 Eksperimentel evaluering

Vi beskriver i afsnit 19, hvorledes vi har testet den endelige implementation af afstandsoraklet. Vi starter med at beskrive, hvorledes vi har testet korrektheden af de svar, der returneres ved at teste op mod en bevist korrekt algoritme. Vi beskriver, hvorledes vi har teste afstandsoraklet og dele deraf. Vi forsøger at underbygge det forventede asymptotiske ressourceforbrug, men må konstatere, at vi ikke kan lave afstandsorakler ud fra grafer, der er store nok til, at vi kan udlede nogle klare tendenser. Vi kigger i stedet på, hvorledes forbruget af ressourcer ændrer sig når vægten af den tungeste kant og ε varieres. På baggrund af disse test konkluderer vi, at betydningen af værdien af ε er minimal i forhold til forbruget af ressourcer. Endvidere ser vi, som forventet, at forbruget af både tid og plads øges når vægten på den tungeste kant øges. Derefter kigger vi på kvaliteten af de approksimative afstande vi beregner og konkluderer, at en stor procentdel af de beregnede svar er korrekte og for de resterende svar gælder at afvigelserne er en del mindre end det lovede.

Kapitel 14

Introduktion til (3,α)-lagsgrafer

I dette kapitel beskrives, hvordan vi reducerer problemet med at beregne approksimative afstande i en planar, orienteret, sammenhængende graf G, hvor vægtene på kanterne er ikke-negative og højst α til et problem i $(3, \alpha)$-lagsgrafer. Med udgangspunkt i G konstruerer vi $(3, \alpha)$-lagsgraferne, $G_0, G_1, \ldots, G_{k-2}$, hvor k er afhængig af strukturen af G. For at approksimere en vilkårlig afstand af længde højst α i et α-orakel laver vi forespørgsler i nul, en, to eller tre af $(3, \alpha)$-lagsgraferne.

Vi vil starte med at definere hvad en $(3, \alpha)$-lagsgraf er og hvordan den er opbygget. Definitionen af $(3, \alpha)$-lagsgrafer har mange ligheder med definitionen af tolagsgrafer i afsnit 4.1, og der vil ligeledes være en del af de egenskaber vi beviser for $(3, \alpha)$-lagsgrafer, der vil minde om de sætninger vi beviste i afsnit 4.2.2. På trods af disse ligheder er der store implementationsmæssige forskelle. Vi vil beskrive vores implementation i afsnit 14.5.

14.1 Definition af $(3, \alpha)$-lagsgrafer

Definition 14.1 *Lad en planar, orienteret graf G med ikke-negative vægte ikke større end α på kanterne være givet. Lad T være et udspændende træ i den underliggende uorienterede graf for G. Træet T har en rod r og en sti mellem r og en knude v er højst er sammensat af tre minimale orienterede stier i G. Træet T er et* $(3, \alpha)$-lags-udspændende træ *hvis enhver af de minimale stier højst er α lange.*

En $(3, \alpha)$-lagsgraf *er en orienteret graf, der indeholder et* $(3, \alpha)$*-lags-udspændende træ.*

Det bør bemærkes, at der er to vigtige forskelle mellem $(3, \alpha)$-lagsgrafer og tolagsgrafer. For det første er der i $(3, \alpha)$-lagsgraferne en begrænsning på længden af vejene i de enkelte lag. For det andet er stierne i det $(3, \alpha)$-lags-udspændende træ sammensat af minimale stier. Vi vil i de følgende afsnit beskrive, hvad disse ændringer betyder for strukturen af $(3, \alpha)$-lagsgraferne.

14.2 Opbygning af $(3, \alpha)$-lagsgrafer

Lad G være en orienteret, planar graf hvor vægtene på kanterne er ikke-negative og højst α. Som tidligere nævnt beskrives mængden af knuder i G med $\mathcal{V}(G)$, mængden af kanter er $\mathcal{E}(G)$, antallet af knuder er n og antallet af kanter tilhører $\mathcal{O}(n)$. For at konstruere $(3, \alpha)$-lagsgraferne skal knuderne i G deles op i knudedisjunkte *lag* $L_0, L_1, \ldots, L_k$. Vi vil i dette afsnit ikke beskrive hvorledes konstruktionen af lagene kan realiseres, dette findes i afsnit 14.5, hvor vores implementation af lagindelingen og konstruktionen af $(3, \alpha)$-lagsgraferne beskrives.

Definition 14.2 *Et* lag L_i *i en orienteret, vægtet graf G defineres som*

$$L_0 = \{v \in \mathcal{V}(G) \mid \delta(r, v) \leq \alpha\}$$

$$L_i = \begin{cases} \{v \in \mathcal{V}(G) \setminus \bigcup_{j<i} L_j \mid \delta(v, \bigcup_{j<i} L_j) \leq \alpha\} & \textit{hvis } i \textit{ er ulige og } i > 0 \\ \{v \in \mathcal{V}(G) \setminus \bigcup_{j<i} L_j \mid \delta(\bigcup_{j<i} L_j, v) \leq \alpha\} & \textit{hvis } i \textit{ er lige og } i > 0 \end{cases}$$

hvor $\delta(v, \bigcup_{j<i} L_j) \leq \alpha$ er den korteste afstand i G fra v til en knude i mængden $\bigcup_{j<i} L_j$. Betydningen af $\delta(\bigcup_{j<i} L_j, v)$ er symmetrisk.

Inddelingen af grafen G i lag er som følger:

1. En vilkårlig rod r vælges blandt knuderne i G.

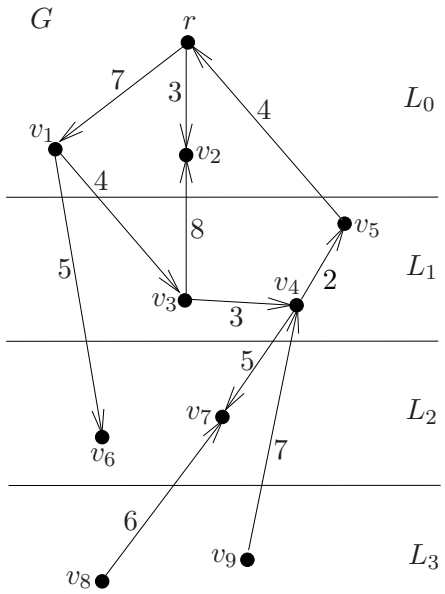
2. Første lag L_0 defineres til at bestå af r og alle knuder v, der kan nås fra den valgte rod (enten direkte eller indirekte) ad en sti, der er kortere end α.

3. Lad i være lige. Laget L_i defineres som de knuder, der kan nås fra de foregående lag ad en sti, der er kortere end α.

4. Lad i være ulige. Laget L_i defineres som de knuder, der kan nå de foregående lag ad en sti, der er kortere end α.

Processen slutter, når der ikke længere kan defineres lag indeholdende knuder. Ingen kanter i grafen G er tungere end α og G er pr. antagelse sammenhængende. Alle knuder vil derfor være indekseret når processen slutter. En knude v indekseres med det lag $\iota(v)$ den tilhører, dvs. at hvis v tilhører lag L_i er $\iota(v) = i$. En lagdeling med $\alpha = 10$ er illustreret på figur 14.1.

Lagene definerer graferne $G_0, G_1, \ldots, G_{k-2}$. Knuderne i hver graf er defineret af tre på hinanden følgende lag og en ekstra knude r_i der er fremkommet ved en kontraktion af de foregående lag. Vi diskuterede i afsnit 4.2.1, at vi tilføjer en ekstra knude r_0 til tolagsgrafen G_0 for at undgå specialtilfælde. Vi tilføjer igen en ekstra rod r_0 til G_0, således at alle $(3, \alpha)$-lagsgraferne har en rod, der *ikke* er en kopi af en knude i G. Knuderne i $(3, \alpha)$-lagsgraferne er som følger:

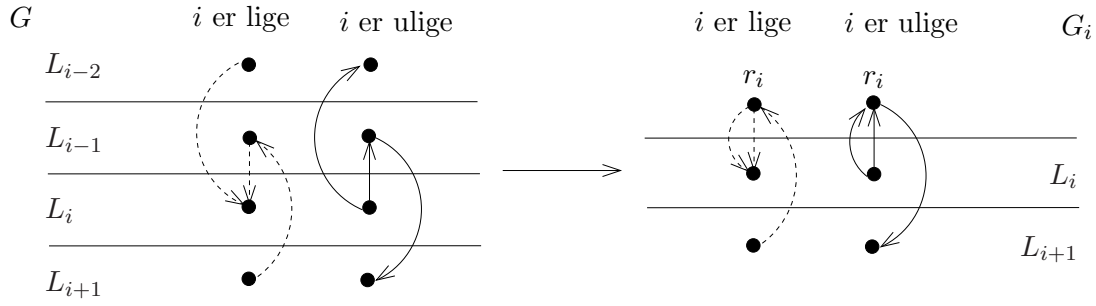


Figur 14.1: En rod vælges og grafen deles op i lag ud fra vægten på kanterne og deres orientering. Vi har valgt $\alpha = 10$.

$$\begin{aligned}
\mathcal{V}(G_0) &= L_0 \cup L_1 \cup L_2 \cup \{r_0\} \\
\mathcal{V}(G_1) &= L_1 \cup L_2 \cup L_3 \cup \{r_1\} \\
&\vdots \\
\mathcal{V}(G_{k-2}) &= L_{k-2} \cup L_{k-1} \cup L_k \cup \{r_{k-1}\}
\end{aligned}$$

Konstruktionen af $(3, \alpha)$-lagsgraferne medfører, at knuder i første og sidste lag er med i én graf, knuder i andet og næstsidste lag er med i to grafer og alle andre knuder er med i tre grafer. En kant $(v, w) \in \mathcal{E}(G)$ er inkluderet i G_i hvis både v og w er i G_i. Kanter hvor det ene endepunkt tilhører den kontraherede rod og det andet er i grafen G_i bliver ligeledes inkluderet. Mere formelt:

Definition 14.3 *Lad en planar, orienteret graf G og et α være givet og definer lagene $L_0, L_1, \ldots, L_k$ som i definition 14.2. $(3, \alpha)$-lagsgraferne $G_0, G_1, \ldots, G_{k-2}$*



Figur 14.2: Hvorledes kanter incidente med en rod opstår.

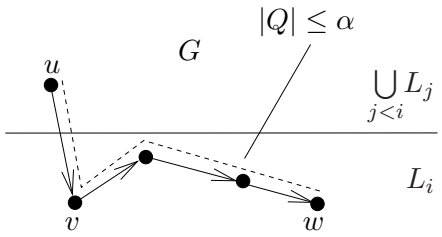
defineres som

$$
\begin{aligned}
G_i =& (V_i, E_i) && \text{for } 0 \leq i < k-1 \\
V_i =& L_i \cup L_{i+1} \cup L_{i+2} \cup \{r_i\} && \text{for } 0 \leq i < k-1 \\
E_i =& E_i' \cup \{(v,w) \mid (v,w) \in \mathcal{E}(G) \wedge \\
& \qquad v \in L_i \cup L_{i+1} \cup L_{i+2} \wedge \\
& \qquad w \in L_i \cup L_{i+1} \cup L_{i+2}\} && \text{for } 0 \leq i < k-1 \\
E_i' =& \{(r_i, w) \mid \exists v \in L_{i-2} \cup L_{i-1} \wedge w \in L_i \wedge \\
& \qquad (v,w) \in \mathcal{E}(G)\} \cup \\
& \{(v, r_i) \mid v \in L_{i+1} \ \wedge \exists w \in L_{i-1} \wedge \\
& \qquad (v,w) \in \mathcal{E}(G)\} && \text{for } 0 \leq i < k-1 \ \wedge i \text{ lige} \\
E_i' =& \{(v, r_i) \mid v \in L_i \ \wedge \exists w \in L_{i-2} \cup L_{i-1} \wedge \\
& \qquad (v,w) \in \mathcal{E}(G)\} \cup \\
& \{(r_i, w) \mid \exists v \in L_{i-1} \ \wedge w \in L_{i+1} \wedge \\
& \qquad (v,w) \in \mathcal{E}(G)\} && \text{for } 0 \leq i < k-1 \ \wedge i \text{ ulige}
\end{aligned}
$$

Bemærk, at en kant $(v, w) \in \mathcal{E}(G)$ pr. konstruktion højst kan spænde over tre lag. På figur 14.2 ses hvorledes kanter til/fra roden opstår.

14.2.1 $(3, \alpha)$-lags-udspændende træ

Vi har defineret hvorledes grafen G er blevet delt op i lagene $L_0, L_1, \ldots, L_k$ og hvilke knuder og kanter, der er indeholdt i $(3, \alpha)$-lagsgraferne. Vi mangler blot at konstruere et $(3, \alpha)$-lags-udspændende træ i hver af G_i'erne. Vi udfører en modificeret Dijkstra-algoritme for hvert lag; når en knude opdateres i prioritetskøen huskes på hvilken kant, der er årsag til opdateringen og når knuden tages ud defineres kanten til at være en del af det udspændende træ. For en $(3, \alpha)$-lagsgraf G_i er roden r_i udgangspunkt for Dijkstra-algoritmen for første lag. Når en knude v tages ud af prioritetskøen, bliver den sat ind i en ny prioritetskø, der er udgangspunktet for en udførelse af Dijkstras algoritme for det



Figur 14.3: En minimal sti.

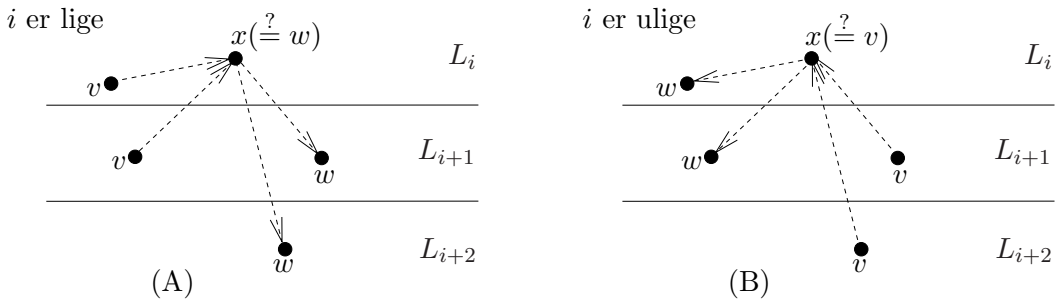
næste lag. Prioriteten af v forbliver uændret. Når en knude w tages ud af en prioritetskø for anden gang er den blevet indekseret med et lag. Vi skal derfor kun opdatere prioriteten på w's naboer. Knuden w skal ikke indsættes i en ny prioritetskø og der skal ikke tilføjes en kant til træet. Dette sikrer os, at der ingen cykler opstår og at alle knuder dækkes af træet. I afsnit 14.5 beskrives i detaljer, hvorledes vi i vores implementation finder $(3, \alpha)$-lags-udspændende træer.

Se på grafen G_i og antag, at i er lige. Da vil kanter mellem roden og knuder i L_i være orienteret væk fra r_i. Den første udførsel af Dijkstra-algoritmen skal derfor følge kanternes orientering, den næste skal være mod kanternes orientering og den sidste skal igen følge kanternes orientering. Tilfældet hvor i er ulige er symmetrisk, her starter man med at udføre en Dijkstra-algoritme, der går modsat kanternes orientering.

Vi vil argumentere for, at det netop konstruerede træ T_i i en vilkårlig $(3, \alpha)$-lagsgraf G_i er et $(3, \alpha)$-lags-udspændende træ. En sti i træet vil højst bestå af tre orienterede stier; en orienteret sti for hver af de tre udførelser af Dijkstra-algoritmen, der hver dækker et lag i grafen.

Se på en sti P fra roden r_i til en vilkårlig knude v, denne sti er opbygget af maksimalt tre orienterede stier. De tre stier er opbygget i forbindelse med en udførelse af Dijkstra-algoritmen og er derfor minimale i G.

Vi skal argumentere for, at en sti, der er fundet i forbindelse med en udførelse af Dijkstras algoritme ikke kan være længere end α. Antag uden tab af generalitet, at i er lige og antag, at der eksisterer en sti P i T_i, fra roden r_i til en knude $w \in L_i$, der er længere end α. Pr. konstruktion af træet beskriver P en minimal sti fra r_i til w. Knuden w er blevet inkluderet i L_i, fordi der eksisterer en sti Q i G fra et tidligere lag til w og denne sti er højst α lang. Kald den sidste knude på Q, der tilhører et tidligere lag for u og den første knude i L_i for v, se figur 14.3. Alle knuderne i L_i eksisterer som en kopi i G_i og kanterne i Q fra v til w er ligeledes kopieret i G_i. Pr. konstruktion af kanterne incidente med roden, er kanten på Q fra u til v, repræsenteret i G_i som en kant fra r_i til v. Da en kopi af en kant har samme vægt som den originale kant, har vi fundet en sti fra r_i til w, der er kortere end P. Dette strider mod, at P skulle være minimal, dvs. der eksisterer *ikke* en sti fra roden r_i til en knude $w \in L_i$, der er længere end α. Vi kan på samme måde argumentere for, at der i de to øvrige lag ikke kan eksistere stier i T_i, der er længere end α. Hermed har vi argumenteret for, at træet er et $(3, \alpha)$-lags-udspændende træ.



Figur 14.4: En sti, der er kortere end α.

14.3 Egenskaber for $(3, \alpha)$-lagsgrafer

På baggrund af definition 14.1 af $(3, \alpha)$-lagsgrafer, vil vi bevise egenskaber for graferne.

Definition 14.4 *Den korteste afstand mellem to knuder v og w i grafen G betegnes $\delta_G(v, w)$.*

Sætning 14.5 *Der eksisterer en minimal sti i G af længde $\delta_G(v, w) \leq \alpha$ mellem knuderne v og w hvis og kun hvis den korteste afstand mellem v og w i graferne $G_{\iota(v)-2}$, $G_{\iota(v)-1}$ og $G_{\iota(v)}$ er $\delta_G(v, w)$ eller:*

$$\begin{aligned} &\delta_G(v, w) \leq \alpha \\ &\Updownarrow \\ &min(\delta_{G_{\iota(v)-2}}(v, w), \delta_{G_{\iota(v)-1}}(v, w), \delta_{G_{\iota(v)}}(v, w)) = \delta_G(v, w) \end{aligned}$$

Bevis. Antag, at der eksisterer en sti $P_{[v,w]}$ af længde højst α i grafen G mellem knuderne v og w. Lad i være det mindste indeks på et lag, der indeholder knuder i $P_{[v,w]}$ og lad x være en knude i dette lag. Vi kalder vejen fra v til x for $P_{[v,x[}$ og vejen fra x til w kaldes $P_{]x,w]}$, knuden x er ikke inkluderet i stierne.

Se figur 14.4 (A), antag at i er lige og at x er den sidste knude i lag L_i på $P_{[v,w]}$. Hvis den korteste afstand fra et tidligere lag til w er kortere end α vil knuderne i $P_{]x,w]}$ befinde sig i lag L_i, dvs. $x = w$. Hvis vejen er længere end α vil $P_{]x,w]}$ befinde sig i $L_{i+1} \cup L_{i+2}$. Lag L_{i+1} er et ulige lag og vil indeholde de knuder y på stien, hvorom der gælder, at afstanden fra y til et tidligere lag er kortere end α. Knuder på $P_{]x,w]}$, der ikke er inkluderet i L_{i+1} vil blive inkluderet i lag L_{i+2}, da de alle kan nås fra et tidligere lag (L_i) ad en sti, der er kortere end α. Stien $P_{[v,w]}$ befinder sig ikke i et tidligere lag end L_i, derfor vil $P_{[v,x[}$ befinde sig i $L_i \cup L_{i+1}$. Den del af stien, der kan nås af knuder fra et tidligere lag via en vej, der er kortere end α vil være inkluderet i L_i. Den resterende del vil være i lag L_{i+1}, da alle knuder i $P_{[v,x[}$ kan nå en knude (x) i et tidligere lag ad en sti, der er kortere end α. Vi har argumenteret for, at hvis i er lige, vil knuderne i $P_{[v,w]}$ være i lagene $L_i \cup L_{i+1} \cup L_{i+2}$. Pr. konstruktion af grafen G_i vil knuderne og kanterne mellem dem være i G_i. Da kanterne i G_i er repræsenteret

med samme vægt som i G vil længden af stien $P_{[v,w]}$ i G_i være den samme som i G. Knuden v er i lag L_i eller i lag L_{i+1}, hvilket betyder at $P_{[v,w]}$ er i $G_{\iota(v)}$ eller i $G_{\iota(v)-1}$.

Antag at i er ulige, se figur 14.4 (B), og lad x være den første knude i lag L_i på $P_{[v,w]}$. Vi kan med et lignende argument argumentere for, at $P_{[v,w]}$ er indeholdt $G_{\iota(v)-1}$ eller i $G_{\iota(v)-2}$ og at længden af vejen er den samme som i G.

Beviset for den modsatte vej er oplagt. □

Sætning 14.6 *Givet en planar graf G er $(3,\alpha)$-lagsgraferne $G_0, G_1, \ldots G_{k-2}$ planare.*

Bevis. Beviset er identisk med beviset for sætning 4.5 i afsnit 4.3. □

Sætning 14.7 *Givet en planar, orienteret graf G, hvor vægtene på kanterne er ikke-negative og højst α. Lad $(3,\alpha)$-lagsgraferne $G_0, G_1, \ldots, G_{k-2}$ være defineret som i definition 14.1. Da gælder, at størrelsen af $(3,\alpha)$-lagsgraferne er lineær i størrelsen af G:*

$$\sum_{i=0}^{k-2} |\mathcal{V}(G_i)| + |\mathcal{E}(G_i)| \in \mathcal{O}(n)$$

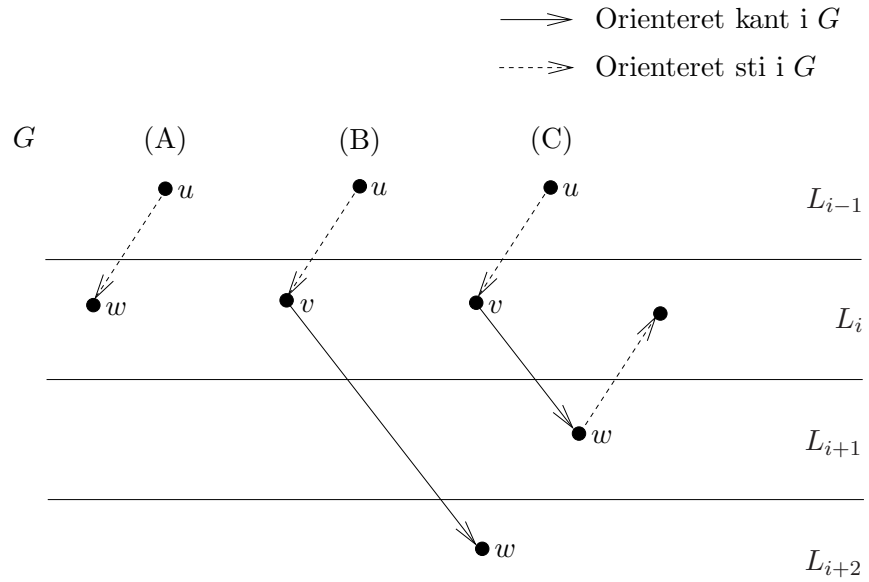
Bevis. Beviset for denne sætning er næsten identisk med beviset for sætning 4.6 i afsnit 4.3, blot har vi, at en knude og en kant i G vil være repræsenteret i højst tre af G_i-graferne. □

14.4 Udførelsestid

Vi vil i dette kapitel skitsere hvorledes $(3,\alpha)$-lagsgraferne $G_0, G_1, \ldots G_{k-2}$ kan konstrueres i tid $\mathcal{O}(n \log(n))$. For en mere udførlig argumentation for udførelsestiden henvises til afsnit 14.5.1, hvor vi diskuterer udførelsestiden af vores implementerede algoritme.

Sætning 14.8 *Givet en planar orienteret graf G med ikke-negative vægte mindre end α på kanterne kan vi i tid $\mathcal{O}(n \log(n))$ konstruere en mængde af orienterede planare $(3,\alpha)$-lagsgrafer $G_0, G_1, \ldots, G_{k-2}$.*

Bevis. Konstruktionen af $(3,\alpha)$-lagsgraferne foregår i to faser; først opdeles knuderne i grafen G i lag og efterfølgende konstrueres $(3,\alpha)$-lagsgraferne. Opdelingen af knuder i lag kan foretages ved at bruge en modificeret udgave af Dijkstras algoritme, hvilket tager tid $\mathcal{O}(n \log(n))$. Herefter kan $(3,\alpha)$-lagsgraferne konstrueres på samme måde som tolagsgraferne blev konstrueret. Dette er beskrevet i afsnit 4.4 og kan gøres i lineær tid, da man ser på hver knude højst tre gange. Den samlede tid for at konstruere $(3,\alpha)$-lagsgraferne bliver da $\mathcal{O}(n \log(n))$. □



Figur 14.5: Eksempler på hvorledes kanter og stier i grafen G kan være placeret i forhold til $(3, \alpha)$-lagsindelingen

14.5 Implementation

Vi vil i dette kapitel beskrive, hvorledes vi har implementeret opbygningen $(3, \alpha)$-lagsgraferne givet en planar, orienteret graf G med ikke-negative vægte mindre end α på kanterne. Vi vil afslutte med at argumentere for, at vi kan konstruere graferne i $\mathcal{O}(n \log(n))$ tid.

Vi har i vores implementation valgt at finde det udspændende træ samtidig med at lagene konstrueres. Der bruges en modificeret udgave af Dijkstras algoritme til at finde lagene og kanterne i det udspændende træ. Vi laver tre prioritetskøer p_0, p_1 og p_2. Køen p_0 indeholder knuder, der er interessante for det lag L_i, der arbejdes med, p_1 hhv. p_2 indeholder knuder, der er interessante for L_{i+1} hhv. L_{i+2}.

Vi starter med at udvælge en tilfældig knude r og lader den være rod i træet. For at konstruere L_0 og det tilhørende træ sætter vi roden ind i køen p_0 med prioritet 0. Herefter fortsætter vi som beskrevet nedenfor.

Antag, at i er lige og at vi er ved at konstruere L_i. Vi tager en knude w ud af prioritetskøen p_0. Knudens prioritet er den korteste afstand, vi har set op til et tidligere lag eller til r, hvis vi er ved at konstruere L_0. Knuden w husker på den kant, der er årsag til, at den har den givne prioritet. Da L_i er et lige lag, skal det indeholde knuder, der kan nås fra et tidligere lag ad en vej, der er kortere end α. På figur 14.5 ses tre forskellige eksempler på, hvad der kan ske når en knude w tages ud af køen p_0. Knudens prioritet er afstanden fra u til w.

Antag at w's prioritet er mindre end eller lig α, se figur 14.5 (A). Kanten, som w husker på, mærker vi som en trækant og vi kalder kanten w's forælderkant. Knuden w sættes ind i lag L_i og den husker selv på, hvilket lag den er blevet sat ind i. Vi kigger på alle w's udkanter og opdaterer prioriteten på

knuderne som kanterne peger på. Laget L_{i+1} består af knuder, der kan nå et tidligere lag ad en vej, der er kortere end α. Knuden w er derfor interessant for L_{i+1} og vi sætter den ind i køen p_1 med prioritet 0. Prioritetskøerne kan altså indeholde knuder, der allerede er inkluderet i et lag. Vi vil senere forklare, hvordan dette håndteres.

Antag nu, at w's prioritet er større end α, se figur 14.5 (B) og (C). Det betyder, at knuden ikke skal inkluderes i lag L_i. Vi erindrer, at ingen kanter har vægt større end α, dvs. w kan blive inkluderet i lag L_{i+2} og vi sætter knuden ind i p_2. Knudens prioritet skal være vægten af kanten (v, w). Knuden w kan naturligvis også komme i lag L_{i+1}, se figur 14.5 (C). Vi vil da have en knude (w) i en kø (p_2) selvom knuden allerede er inkluderet i et lag (L_{i+1}).

Når en knude tages ud af en prioritetskø, undersøger vi først om knuden er inkluderet i et andet lag. Hvis dette er opfyldt skal ingen kanter mærkes som trækant og knuden skal ikke inkluderes i dette lag. Vi skal dog stadig opdatere prioriteten på knudens naboer.

En knude kan være med i fire udførelser af Dijkstras algoritme. Antag v inkluderes i forbindelse med indekseringen af lag i, men ikke bliver inkluderet i dette lag. Da vil v blive inkluderet i L_{i+1} eller L_{i+2} og den kan være med i begge udførelser af Dijkstra algoritme. I opbygningen af L_{i+1} kan vi se på knuden og konstatere den ikke skal i dette lag, fordi stien fra v til L_i er længere end α. Vi vil da sætte den ind i prioritetskøen for L_{i+3}. Knuden v vil blive indsat i lag L_{i+2} og når vi tager den ud af prioritetskøen i forbindelse med opbygningen af L_{i+3} vil vi derfor ikke sætte den ind i flere prioritetskøer.

Tilfældet hvor i er ulige er symmetrisk. Når en knude tages ud af en prioritetskø kigge vi på knudens indkanter og opdaterer prioriteten på kantens andet endepunkt.

Vi har nu fået markeret en mængde af kanter som værende trækanter og vi skal argumentere for, at kanterne danner et udspændende træ. Da grafen er sammenhængende og vægten på alle kanterne højst er α bliver alle knuderne indekseret med et lag. Roden tilhører lag L_0 og den har ingen forælderkant. Når en knude, (med undtagelse af roden) indekseres med et lag, bliver en kant der er incident med knuden markeret som værende trækant. Vi laver ikke en cykel i grafen, da en knude ikke bliver indekseret to gange og den vil derfor få præcis en forælderkant. Det betyder, at vi konstruerer et udspændende træ.

Vi har nu fået delt grafen G op i lag $L_0, L_1, \ldots, L_k$ og fundet et udspændende træ i grafen. Det betyder, at vi har information nok til at kunne konstruere $(3, \alpha)$-lagsgraferne $G_0, G_1, \ldots, G_{k-2}$ hver med et $(3, \alpha)$-lags-udspændende træ. I definition 14.3 er beskrevet hvilke kanter og knuder $(3, \alpha)$-lagsgraferne skal indeholde. Vi laver et laver et array af knude-lister, med en knude-liste for hvert lag. En liste indeholder referencer til de knuder i grafen G, der tilhører det pågældende lag. En $(3, \alpha)$-lagsgraf G_i opbygges et lag ad gangen. Først laves en ny rod r_i og efterfølgende gennemløbes listen med knuderne $v \in L_i$. For hver knude $v \in G$ laves en ny knude $v_i \in G_i$. Incidente kanter til knuden v gennemløbes og vi ser på kantens andet endepunkt w. Der er nu tre muligheder:

$w \in L_{i-2} \cup L_{i-1}$: Lav en ny kant (v_i, r_i).

$w \in L_i$: Hvis en kopi af w er blevet lavet i G_i laves en ny kant (v_i, w_i), ellers

laves kanten når w indsættes.

$w \in L_{i+1} \cup L_{i+2}$**:** Der laves ikke en ny kant, kanten laves når w indsættes.

Gennemløb alle v's indkanter og lav på samme måde kanter fra v_i i grafen G_i. De resterende to lag i grafen G_i konstrueres på tilsvarende vis. En kant, der er en trækant i G skal ligeledes være en trækant i G_i. På denne måde vil en $(3, \alpha)$-lagsgraf G_i komme til at indeholde et udspændende træ T_i.

Se på en sti fra roden r_i til en knude $v_i \in G_i$. Stien består af tre orienterede stier, en for hvert lag i grafen. Pr. konstruktion af lagene vil hver sti højst være α lang. Hver del af stien er minimal, da den er fundet i forbindelse med en udførelse af Dijkstras algoritme. De konstruerede $(3, \alpha)$-lagsgrafer indeholder altså alle et $(3, \alpha)$-lags-udspændende træ.

14.5.1 Udførelsestid for konstruktion af $(3, \alpha)$-lagsgraferne

Vi har netop beskrevet hvorledes $(3, \alpha)$-lagsgraferne opbygges i to faser. I den første fase opdeles grafen G i lag og et udspændende træ findes. Vi udfører Dijkstras algoritme én gang for hvert lag der konstrueres. En knude kan som nævnt højst være med i en udførelse af Dijkstras algoritme fire gange. Det betyder, at udførelsestiden for at dele grafen op i lag er $\mathcal{O}(n \log(n))$. I anden fase opbygges $(3, \alpha)$-lagsgraferne. Tiden for at konstruere en $(3, \alpha)$-lagsgraf G_i er oplagt lineær i størrelsen af G_i. Vi har i sætning 14.7 bevist, at størrelsen af G_i-graferne er lineær i størrelsen af G. Samlet giver det, at vi konstruerer $(3, \alpha)$-lagsgraferne i tid $\mathcal{O}(n \log(n))$.

14.5.2 Placering i programmet

Den del af programmet, der opdeler grafen i lag og finder et udspændende træ findes i `distances/layer.cpp`. Opbygningen af $(3, \alpha)$-lagsgraferne foregår i `distances/gi.cpp` og de er implementeret i klassen `ThreeAlphaGraph`.

Kapitel 15

Opdeling og rekursion

Vi skal konstruere en datastruktur, som kan anvendes til effektivt at finde approksimerede afstande i planare orienterede grafer, hvor kanterne har vægt op til α. Vi kalder den resulterende datastruktur et α-orakel. I dette kapitel gennemgår vi de dele af konstruktionen, hvor der findes tilsvarende dele i konstruktionen af reachability-datastrukturen. Vi beskriver udelukkende disse dele fra et implementationssynspunkt, da det teoretiske grundlag er er beskrevet i del I.

Det første skridt er at reducere problemet til $(3, \alpha)$-lagsgrafer og derefter lave en del-datastruktur for hver af disse grafer. Delene udgør tilsammen den datastruktur vi ønsker. Konstruktionen af $(3, \alpha)$-lagsgraferne er beskrevet i kapitel 14.

Separations-teknikken, som blev anvendt til at konstruere reachability-oraklet kan tilpasses, således at den kan anvendes til konstruktion af en datastruktur, der kan beregne approksimerede afstande i planare grafer. I det følgende beskriver vi, hvordan vi har tilpasset teknikken til $(3, \alpha)$-lagsgrafer i vores implementation af datastrukturen.

Den information, der skal tilføjes datastrukturen for at kunne beregne approksimerede afstande effektivt, beregnes vha. nye konstruktioner, som vi ikke anvendte i forbindelse med reachability-oraklet. Disse konstruktioner beskrives i kapitel 16.

15.1 Lokalisering af trekant til separation

Vi antager, at vi er givet en $(3, \alpha)$-lagsgraf H med et $(3, \alpha)$-lags-udspændende træ T. Ligesom i del I, kapitel 6 ønsker vi at opdele H i tre dele, således at ingen del indeholder mere end halvdelen af knuderne i H. Opdelingen foretages ved at lokalisere en trekant og grafen separeres ved at bruge rodstierne fra knuderne i trekanten. Eksistensen af en trekant er bevist i sætning 6.1 og at den kan findes i lineær tid vises i sætning 6.10. Beviserne for de to sætninger bruger kun, at grafen er planar og indeholder et udspændende træ, og sætningerne er derfor også gældende, når vi arbejder med $(3, \alpha)$-lagsgrafer. Den algoritme vi implementerede i forbindelse med del I bruger ikke, at grafen er en tolagsgraf, blot at den indeholder et udspændende træ. Vi kan derfor bruge nøjagtig den samme implementation til at finde en trekant i en $(3, \alpha)$-lagsgraf som vi brugte

til at finde trekanten i en tolagsgraf.

15.2 Lokalisering af separatorstier

Givet en trekant i en tolagsgraf skal vi lokalisere de op til 9 orienterede separatorstier. Vi starter fra hver af de tre knuder, der definerer trekanten og følger forældrereferencer op til roden. Undervejs finder vi de orienterede stier. Til sidst gennemløber vi alle stierne parvist og fjerner dem, der er en delsti af en anden sti. Vi bemærker, at vi ikke som i reachability finder maksimale stier, da vi nu har brug for, at de orienterede separatorstier højst har længde α.

15.3 Opdeling af grafen vha. en separator

Vi antager, at vi har en $(3, \alpha)$-lagsgraf H samt en separator S. Separatoren beskriver en opdeling af grafen i tre dele, hvoraf ingen indeholder mere end halvt så mange knuder som H. Vi skal nu konstruere tre delgrafer, som svarer til de tre dele.

I kapitel 7 beskrev vi, hvorledes vi konstruerer delgraferne i tilfældet, hvor vi står med en tolagsgraf og en tilhørende separator. Denne konstruktion fordrer ikke, at grafen er en tolagsgraf, og kan derfor ligeledes anvendes til at opdele $(3, \alpha)$-lagsgrafer.

15.4 Basisrekursion

Givet en vægtet, planar, orienteret graf G, anvendes den konstruktion, der er beskrevet i kapitel 14 til at reducere afstandsproblemet til at omhandle $(3, \alpha)$-lagsgraferne $G_1, G_2, \ldots, G_{k-2}$.

Givet en $(3, \alpha)$-lagsgraf H med tilhørende udspændende træ T dekomponerer vi H rekursivt. Vi anvender i et rekursivt kald flg. skridt:

- Vi forbereder H ved at tilføje ekstra information på knuder og kanter. Dette skridt foregår som i basisrekursionen for reachability-oraklet og er beskrevet i afsnit 5.1.1. Den ekstra information gemmes i et `node_array` og et `edge_array`, som knyttes til grafen. Disse typer er introduceret i afsnit 2.1.2.

- Vi finder en trekantet flade, der definerer en opdeling af grafen i tre dele, som hver højst indeholder halvt så mange knuder som H. Denne del af basisrekursionen er beskrevet i afsnit 15.1.

- Vi finder separatoren S, som er defineret af den netop lokaliserede flade. De orienterede stier identificeres. Dette skridt er beskrevet i afsnit 15.2.

- For hver af de orienterede separatorstier laver vi forbindelser. I kapitel 16 beskrives i detaljer hvorledes vi finder forbindelser, der repræsenterer approksimative afstande i H.

- Grafen H deles op vha. separatoren S. Vi konstruerer i lineær tid op til tre nye delgrafer. Denne fase er beskrevet i afsnit 15.3.
- Der kaldes rekursivt på de dele, som opdelingen af H resulterer i.

Vi bemærker at denne dekomposition stort set svarer til den vi foretog for de uvægtede grafer i del I.

Rekursionsdybden er som i reachability-tilfældet $\mathcal{O}(\log(n))$. Alle skridt i basisrekursionen bortset fra det skridt, der finder forbindelser, foregår i tid, der er lineær i størrelsen af H. Skridtet, hvor der findes forbindelser bliver som nævnt beskrevet i kapitel 16, hvor vi konkluderer, at tiden for dette skridt er $\mathcal{O}(|\mathcal{V}(H)| \log^2(|\mathcal{V}(H)|)/\varepsilon)$. Det er altså dette skridt, der dominerer udførelsestiden for et rekursivt kald i vores dekomposition.

Vi vil i kapitel 17 beskrive den samlede konstruktionstid for et α-orakel. Her vil vi ligeledes beskrive, hvor meget plads et α-orakel kræver.

15.5 Rekursionstræ og separatornumre

Under basisrekursionen opbygger vi et rekursionstræ på samme måde som vi gjorde det i reachability-tilfældet i afsnit 5.1.3. Vi erindrer at dette er et træ, hvor der er en knude for hvert rekursivt kald i dekompositionen. Vi tildeler ligeledes numre til separatorstierne, således at numrene er fortløbende på alle stier fra rod til blad i rekursionstræet. En knude i rekursionstræet har tilknyttet et separatornummer. Dette er det sidste nummer, der er anvendt til at nummerere separatorstier i det kald, der svarer til knuden.

15.6 Placering i programmet

Basisrekursionen findes i `distance/recursion.cpp`. Vi starter med at tilføje labels med ekstra information til knuder og kanter i `distance/labels.cpp`. I `distance/ArrayInfo` findes klasserne som vi samler information til knuder og kanter i. Klasserne er hhv. `NodeArrayInfo` og `EdgeArrayInfo`. Trekanten, som grafen opdeles efter finder vi i `distance/findTriangle.cpp`. Separatoren og de orienterede stier i denne finder vi i `distance/separator.cpp`. Selve opdelingen af grafen i delgrafer findes i `distance/partition.cpp`.

Kapitel 16

Forbindelser over separatorstier

Antag vi er givet en $(3, \alpha)$-lagsgraf H med et udspændende træ T, samt en separator S, der består af et antal orienterede separatorstier. Vi ønsker at lave mængder af forbindelser fra knuder til separatorstier og fra separatorstier til knuder. Disse forbindelser skal gøre os i stand til at finde en approksimeret afstand mellem to givne knuder. Mængderne skal være tilpas små til at afstanden kan beregnes effektivt, men skal også repræsentere nok information til at give et resultat, der har den ønskede præcision. Vi ønsker, at resultatet skal være konservativt og højst $\varepsilon\alpha$ for stort. Vi ønsker at kunne give en approksimativ afstand for de par af knuder (v, w), hvorom det gælder, at $\delta(v, w) \leq \alpha$.

For hver separatorsti, der opstår under opdelingen af grafen H, laves forbindelser, som senere skal anvendes til at lave forespørgsler om afstanden mellem to knuder. For hver sti laves et antal forbindelser fra en knude til stien samt fra stien til knuden. Dette gøres for alle knuder, der findes i H på det tidspunkt, forbindelserne bliver lavet (givet, at der findes veje til og fra den pågældende knude). Forbindelserne gemmes som kanter i den graf G_i, der dekomponeres og disse gemmes i lister i knuderne i G_i. For hver knude v gemmes to lister, ind_v og ud_v, hvor hver indgang indeholder en mængde af forbindelser til v fra en separatorsti Q hhv. fra v til stien Q. Længden af disse lister er $\mathcal{O}\log(\mathcal{V}(G_i))$ svarende til reskursionsdybden af dekompositionen.

For at kunne repræsentere approksimerede afstande vha. en mængde forbindelser, har vi brug for flere nye konstruktioner, som vi først vil præsentere. Disse konstruktioner omfatter forbindelser, afstande, mængder af forbindelser samt ordning af disse mængder. Vi vil ligeledes vise nogle fakta om de forskellige konstruktioner. De nævnte konstruktioner præsenteres i afsnit 16.1.

Herefter ser vi på, hvorledes mængder af forbindelser kan konstrueres, så de indeholder tilstrækkelig information til at en approksimativ afstand kan beregnes, så den overholder vores ønsker om præcision. Disse mængder af forbindelser tilknyttes knuderne i den graf G_i, som rekursivt dekomponeres. Vi beviser korrektheden af konstruktionen, hvor store mængderne er, samt argumenterer for tidsforbruget for konstruktionen. Vi konkluderer, at tidsforbruget for konstruktionen af alle mængder på et niveau i den rekursive dekomposition er $\mathcal{O}(|\mathcal{V}(H)| \log^2(|\mathcal{V}(H)|)/\varepsilon)$. Konstruktionen af mængderne og de nævnte beviser findes i afsnit 16.2.

Når vi har konstrueret mængderne, skal vi sørge for, at de har form og

størrelse, der giver os mulighed for effektivt at beregne en approksimeret afstand med ønsket præcision. Vi beskriver, hvorledes vi mindsker antallet af forbindelser uden at give afkald på den ønskede præcision af forespørgsler. Denne fase er beskrevet i afsnit 16.3, hvor vi også argumenterer for, at de resulterende mængder af forbindelser mellem en knude og en separatorsti er af størrelse $\mathcal{O}(1/\varepsilon)$. Vi opsummerer den samlede konstruktion af mængder og vi argumenterer ligeledes for det samlede tidsforbrug ved konstruktionen. Vi konkluderer, at det samlede tidsforbrug er $\mathcal{O}(|\mathcal{V}(H)| \log^2(|\mathcal{V}(H)|)/\varepsilon)$.

Vi ser til sidst kort på hvorledes vi i vores implementation konstruerer de forbindelser, som skal anvendes i forespørgsler. Implementationen beskrives i afsnit 16.4.

Forespørgslerne behandles i kapitel 17, hvor vi også giver en kort opsummering af den samlede konstruktion af et α-orakel samt ser på, hvilke begrænsninger der er på de resultater, et α-orakel giver.

16.1 Forbindelser, afstande og mængder

Antag, at vi er givet en $(3, \alpha)$-lagsgraf H og den tilhørende separator S, som beskrevet i afsnit 15.4. Vi definerer først et antal konstruktioner, som anvendes, når vi konstruerer forbindelser over separatorstier i S. Vi definerer først formelt, hvad en forbindelse er. Herefter definerer vi mængder af forbindelser mellem knuder og separatorstier, der indeholder den information vi har brug for, når vi skal beregne en approksimeret afstand mellem to knuder.

16.1.1 Forbindelser og afstande

Givet en planar, orienteret, vægtet $(3, \alpha)$-lagsgraf H vil vi definere forbindelser mellem knuder og separatorstier samt afstande via forbindelser. Afstande via forbindelser over separatorstier vil udgøre de afstande, som vores datastruktur kan returnere på forespørgsler.

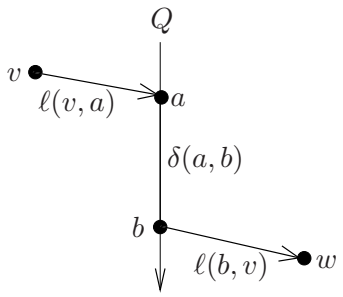
Definition 16.1 *Den korteste afstand fra en knude v til en knude w i H betegnes $\delta(v, w)$.*

Definition 16.2 *En* forbindelse *(på engelsk* connection*) fra en separatorsti Q til en knude v er en ny kant $(a, v) \in \mathcal{V}(Q) \times \{v\}$ med længde $\ell(a, v) \geq \delta(a, v)$.*

Ligeledes er en forbindelse *fra en knude v til en separatorsti Q en ny kant $(v, a) \in \{v\} \times \mathcal{V}(Q)$ med længde $\ell(v, a) \geq \delta(v, a)$.*

Eksempler på forbindelser (v, a) fra knuden v til a på stien Q samt (b, w) fra knuden b på Q til w kan ses på figur 16.1.

Vi kan nu definere de konstruktioner, som vi skal anvende mht. forbindelser. Vi definerer først den afstand, som vi kan finde mellem to knuder, hvis vi kombinerer to forbindelser via en separatorsti.



Figur 16.1: Forbindelser (v, a) og (b, w) mellem Q og knuder i H der sammen med $\delta(a, b)$ udgør $dist((v, a), (b, w))$.

Definition 16.3 *Antag at (v, a) er en forbindelse fra v til a på Q og at (b, w) er en forbindelse fra b på Q til w. Hvis a og b er den samme knude, eller hvis a kommer før b på Q, da definerer vi*

$$dist((v, a), (b, w)) = \ell(v, a) + \delta(a, b) + \ell(b, w)$$

Hvis a kommer efter b på Q, defineres $dist((v, a), (b, w))$ til ∞.

En illustration af $dist((v, a), (b, w))$ kan ses på figur 16.1. Vi vil gerne kunne beregne $dist((v, a), (b, w))$ effektivt og gemmer derfor ekstra information i knuderne. Vi har brug for at kunne beregne $\delta(a, b)$ samt afgøre, om en knude kommer før en anden på Q.

Definition 16.4 *Lad $i(a, Q)$ være indekset for a på Q. Hvis $i(a, Q) \leq i(b, Q)$, så er a samme knude eller en tidligere knude på Q ift. b.*

Lad $d(a, Q)$ være afstanden på Q fra den første knude på Q til a. Nu kan $\delta(a, b)$ beregnes som $d(b, Q) - d(a, Q)$, givet at $i(a, Q) \leq i(b, Q)$.

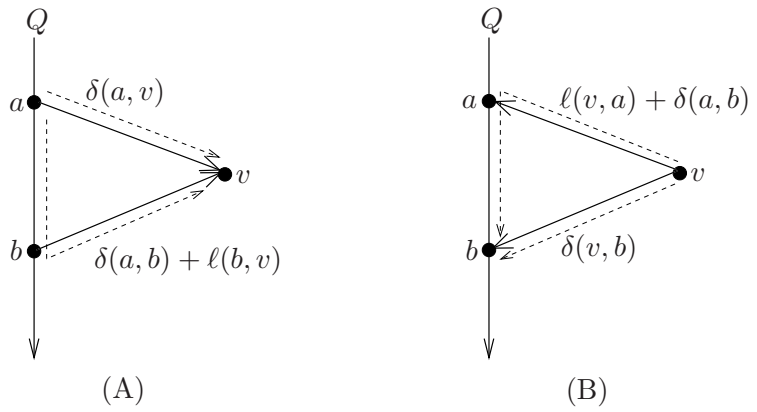
Vi bemærker at $d(a, Q)$ og $d(b, Q)$ ikke kan anvendes til at afgøre, hvilken af a og b, der er først på Q, da der kan være kanter med vægt 0.

16.1.2 Mængder af forbindelser

Hver knude skal opbevare en mængde af forbindelser for hver separatorsti, Q, som opdeler de grafer, som knuden findes i. Disse mængder skal repræsentere tilstrækkelig information til at man kan finde en afstand mellem to knuder v og w over Q, som er passende tæt på den korteste afstand, der findes mellem v og w over Q.

Vi definerer først et mål for, hvornår en forbindelse f_1 er unødvendig, hvilket vil sige, at en anden forbindelse f_2 kan anvendes i stedet for f_1 uden at vi indfører en afvigelse fra den korrekte korteste afstand, der er for stor til at vi kan overholde kravet til vores resultat om, at det højst er $\varepsilon\alpha$ for stort.

Definition 16.5 *Antag, at $i(a, Q) \leq i(b, Q)$ samt at (a, v) og (b, v) er forbindelser fra hhv. a og b på Q til knuden v. Vi siger nu, at (b, v) ε-dækker (a, v) hvis $\delta(a, b) + \ell(b, v) \leq \delta(a, v) + \varepsilon\alpha$.*



Figur 16.2: Illustration af ε-dækning vha. forbindelser, der går hhv. til og fra separatorstien Q.

Dette betyder, at hvis vi vælger vejen fra a til v, der går over b, fremfor den direkte (korteste) vej fra a til v, får vi højst en additiv fejl på $\varepsilon\alpha$. En illustration af de to veje ses på figur 16.2 (A).

Vi kan ligeledes definere ε-dækning for forbindelser, der går fra en knude v til Q.

Definition 16.6 *Antag, at $i(a, Q) \leq i(b, Q)$ samt at (v, a) og (v, b) er forbindelser fra knuden v til hhv. a og b på Q. Vi siger at (v, a)* ε-dækker *(v, b) hvis $\ell(v, a) + \delta(a, b) \leq \delta(v, b) + \varepsilon\alpha$.*

Vælger vi nu vejen fra v til b over a fremfor den direkte (korteste) vej fra v til b, kan vi indføre en additiv fejl på højst $\varepsilon\alpha$, se figur 16.2 (B).

Vi kan nu definere en mængde af forbindelser, der dækker vores behov for passende præcision mht. den korteste vej fra en knude v til en knude w.

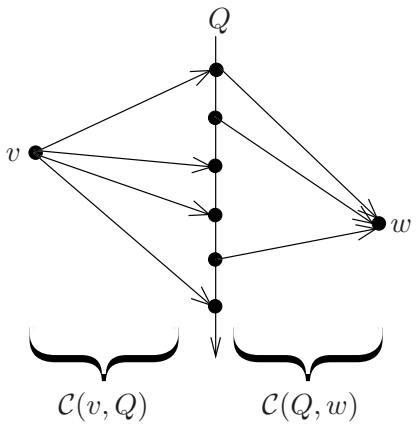
Definition 16.7 *En mængde $\mathcal{C}(v, Q)$ af forbindelser fra v til Q er* ε-dækkende*, hvis alle par (v, a) i $\{v\} \times \mathcal{V}(Q)$ hvor $\delta(v, a) \leq \alpha$ er ε-dækket af en forbindelse i $\mathcal{C}(v, Q)$.*

Ligeledes er en mængde $\mathcal{C}(Q, w)$ af forbindelser fra Q til w ε-dækkende*, hvis alle par (b, w) i $\mathcal{V}(Q) \times \{w\}$ hvor $\delta(b, w) \leq \alpha$ er ε-dækket af en forbindelse i $\mathcal{C}(Q, w)$.*

Givet to mængder af forbindelser, hhv. fra v til Q og fra Q til w, kan vi nu vha. to ε-dækkende mængder definere en afstand fra v til w over separatorstien Q. Denne er afstanden, vi anvender, når vi i forespørgsler finder en afstand mellem v og w over Q.

Definition 16.8 *Antag at $\mathcal{C}(v, Q)$ er en mængde af forbindelser fra v til Q samt at $\mathcal{C}(Q, w)$ er en mængde af forbindelser fra Q til w. Vi definerer*

$$dist(\mathcal{C}(v, Q), \mathcal{C}(Q, w)) = min_{(v,a)\in\mathcal{C}(v,Q),(b,w)\in\mathcal{C}(Q,w)}\, dist((v, a), (b, w))$$



Figur 16.3: Illustration af mængderne $\mathcal{C}(v,Q)$ og $\mathcal{C}(Q,w)$, som definerer afstanden fra v til w over Q.

En illustration af de to mængder $\mathcal{C}(v,Q)$ og $\mathcal{C}(Q,w)$ kan ses på figur 16.3. Disse mængder definerer altså afstanden fra v til w over Q.

Med konstruktionen af ε-dækkende mængder er vi i stand til at finde approksimerede afstande over separatorstier om hvilke vi kan vise, at de ikke afviger *for meget* fra den reelle korteste afstand. Dette vil vi vise i det flg. lemma.

Lemma 16.9 *Lad $v, w \in \mathcal{V}(H)$. Antag, at der findes en korteste vej, P, fra v til w, der skærer Q samt at $\delta(v,w) \leq \alpha$. Hvis $\mathcal{C}(v,Q)$ og $\mathcal{C}(Q,w)$ er ε-dækkende mængder, da er $dist(\mathcal{C}(v,Q),\mathcal{C}(Q,w)) \leq \delta(v,w) + 2\varepsilon\alpha$.*

Bevis. Lad x være en knude på Q, der ligger på vejen P, se figur 16.4. Dette betyder, at $\delta(v,w) = \delta(v,x) + \delta(x,w)$.

Da $\delta(v,w) \leq \alpha$ gælder det at $\delta(v,x) \leq \alpha$. Dermed er (v,x) ifølge definition 16.7 ε-dækket af en forbindelse $(v,a) \in \mathcal{C}(v,Q)$, dvs. at $\ell(v,a) + \delta(a,x) \leq \delta(v,x) + \varepsilon\alpha$.

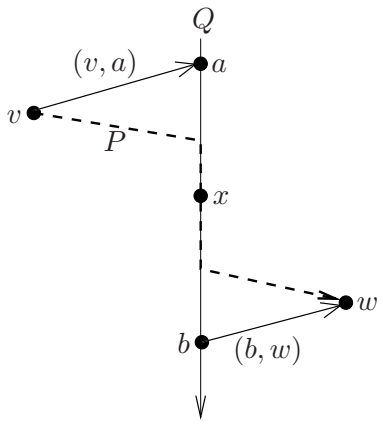
Ligeledes er $\delta(x,w) \leq \alpha$ da $\delta(v,w) \leq \alpha$. Dermed er (x,w) ε-dækket af en forbindelse $(b,w) \in \mathcal{C}(Q,w)$ og da er $\delta(x,b) + \ell(b,w) \leq \delta(x,w) + \varepsilon\alpha$.

Dermed kan vi konkludere at

$$\begin{aligned} dist(\mathcal{C}(v,Q),\mathcal{C}(Q,w)) &\leq dist((v,a),(b,v)) \\ &= \ell(v,a) + \delta(a,b) + \ell(b,w) \\ &= \ell(v,a) + \delta(a,x) + \delta(x,b) + \ell(b,w) \\ &\leq \delta(v,x) + \varepsilon\alpha + \delta(x,w) + \varepsilon\alpha \\ &= \delta(v,w) + 2\varepsilon\alpha \end{aligned}$$

□

Vi har nu den ønskede måde hvorpå vi kan repræsentere afstande af passende præcision over en separatorsti Q. Hvis vi har to ε-dækkende mængder $\mathcal{C}(v,Q)$ og $\mathcal{C}(Q,w)$, er det muligt at finde en afstand fra v til w over Q, som højst har en additiv fejl på $2\varepsilon\alpha$. Hvis vi, når vi laver vores mængder, bruger et ε',



Figur 16.4: En korteste vej P over separatorstien q fra v til w, en knude x, hvor $x \in Q$ og $x \in P$ samt to forbindelser (v, a) og (b, w), der ε-dækker hhv. (v, x) og (x, w).

der er halvt så stort som ε, får vi nu $\varepsilon/2$-dækkende mængder, og kan derfor få afstande, der højst har en additiv fejl på $\varepsilon\alpha$, hvilket er den ønskede præcision.

I det næste afsnit vi vil vise, hvorledes ε-dækkende mængder kan konstrueres for et givet ε.

16.2 Konstruktion af ε-dækkende mængder

Vi vil i dette afsnit beskrive, hvorledes ε-dækkende mængder kan konstrueres. For en separatorsti Q og en knude v i $(3, \alpha)$-lagsgrafen H ser vi på, hvordan en ε-dækkende mængde $\mathcal{C}(Q, v)$ af forbindelser fra Q til v konstrueres. Konstruktionen af mængden $\mathcal{C}(v, Q)$ af forbindelser fra v til Q er tilsvarende.

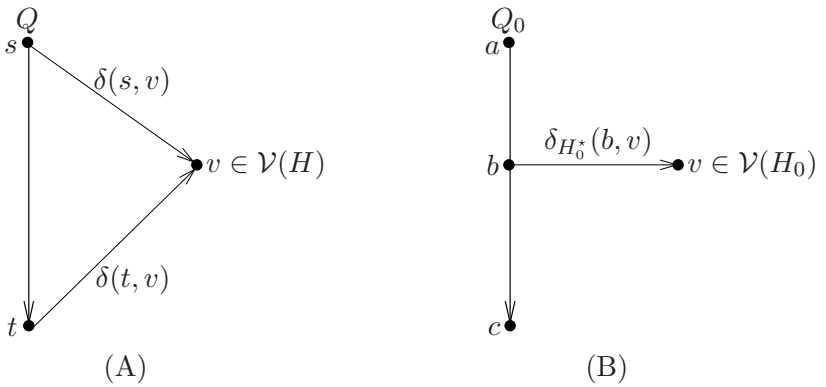
Det er en fordel for os, når vi skal anvende forbindelserne til finde afstande, at mængderne af forbindelser er ordnet på en passende måde. Derfor definerer vi nu, hvorledes vi ønsker vores mængder er ordnede.

Definition 16.10 *Mængden $\mathcal{C}(Q, v)$ er* ordnet*, hvis forbindelserne i mængden er ordnet mht. knudernes orden på Q.*

Vi vil nu vise, hvordan det er muligt at konstruere ordnede ε-dækkende mængder fra en separatorsti Q til knuderne i grafen H. Vi argumenterer for størrelsen af de konstruerede mængder samt for konstruktionstiden.

Lemma 16.11 *Givet en $(3, \alpha)$-lagsgraf H og en orienteret separatorsti Q i H kan vi for hver knude $v \in \mathcal{V}(H)$ konstruere en ordnet ε-dækkende mængde $\mathcal{C}(Q, v)$ af størrelse $\mathcal{O}(\log(|\mathcal{V}(Q)|)/\varepsilon)$. Konstruktionen af mængder for alle knuder i H foregår i samlet tid $\mathcal{O}(|\mathcal{V}(H)| \log(|\mathcal{V}(H)|) \log(|\mathcal{V}(Q)|)/\varepsilon)$.*

Vi deler beviset op i fire dele, som vi viser hver for sig. De fire dele er konstruktion, korrekthed, størrelse af mængderne og effektivitet.



Figur 16.5: Konstruktion af ε-dækkende mængder. (A) De første forbindelser fra s og t. (B) Rekursivt laves forbindelser fra Q_0's midtpunkt b.

16.2.1 Konstruktion

I konstruktionen laver vi forbindelserne med udgangspunkt i stien Q. Som nævnt beskriver vi kun hvorledes vi laver forbindelser fra Q til knuder i H. Hvorledes forbindelser til Q laves er tilsvarende.

Vi introducerer først *semi-ε-dækning*, som vi vil anvende frem for ε-dækning, da det kan beregnes i konstant tid, om en forbindelse semi-ε-dækker en anden forbindelse. Selvom vi anvender semi-ε-dækning er det muligt at konstruere ε-dækkende mængder.

Definition 16.12 *Antag, at $i(a, Q) \leq i(b, Q)$ samt at (a, v) og (b, v) er forbindelser fra hhv. a og b på Q til knuden v. Forbindelsen (b, v)* semi-ε-dækker *(a, v) hvis $\delta(a, b) + \ell(b, v) \leq \ell(a, v) + \varepsilon\alpha$.*

I vores konstruktion anvender vi *single source shortest paths*-beregninger, som vi forkorter til *sssp*.

Indledende konstruktion

Vi er givet en $(3, \alpha)$-lagsgraf H og en separatorsti Q med en startknude s og en slutknude t. Indledningsvis laver vi *sssp* fra s og fra t. For hver $v \in \mathcal{V}(H)$ forbinder vi s til v med $\ell(s, v) = \delta(s, v)$ og ligeledes forbinder vi t til v med $\ell(t, v) = \delta(t, v)$, se figur 16.5 (A). Hvis ingen vej findes til en knude fra hhv. s og t, sættes længden af forbindelsen til ∞.

Grafen kan nu overlades til en rekursiv procedure, der konstruerer de resterende forbindelser fra knuder på Q til knuder i H.

Rekursiv konstruktion

Vi antager, at vi har (Q_0, H_0), at Q_0 et segment af Q, at H_0 er en delgraf af H samt at der er forbindelser fra endeknuderne på Q_0 til alle knuder i H_0. Lad a være startknuden og c være slutknuden på Q_0. Vi har altså forbindelser fra a og c til alle knuder i H_0.

Definition 16.13 *Vi definerer $H_0^\star$ til at være grafen der indeholder H_0, Q_0 samt alle forbindelser fra a og c på Q_0 til knuder i H_0. Afstanden fra v til w i $H_0^\star$ betegner vi $\delta_{H_0^\star}(v, w)$.*

Lad b være den midterste knude på Q_0 (dvs. knude nummer $\lceil |Q_0|/2 \rceil$ på Q_0). Vi laver nu *sssp* fra b i $H_0^\star$ og for hver $v \in \mathcal{V}(H_0)$ forbinder vi b til v med $\ell(b, v) = \delta_{H_0^\star}(b, v)$, se figur 16.5 (B).

Vi deler nu Q_0 op i to dele med henblik på rekursive kald. Vi deler op ved b således at vi har en del fra a til b som vi kalder Q_1 og en del fra b til c, som vi kalder Q_2.

Vi laver to delgrafer af H_0 ligeledes med henblik på rekursive kald. Disse delgrafer er ikke nødvendigvis disjunkte. Vi definerer mængden U_1 til at være mængden af knuder v, hvor $\ell(a, v) > 2\alpha$ eller hvor (b, v) semi-ε-dækker (a, v). Vores første delgraf er nu $H_1 = H_0 \setminus U_1$. Ligeledes definerer vi mængden U_2 til at være mængden af knuder v, hvor $\ell(b, v) > 2\alpha$ eller hvor (c, v) semi-ε-dækker (b, v). Den anden delgraf er nu $H_2 = H_0 \setminus U_2$.

Om en knude v i U_1 kan vi bemærke følgende. Hvis $\ell(a, v) > 2\alpha$, kan der ikke eksistere en sti fra en knude på Q_1 til v af længde højst α. Hvis (b, v) semi-ε-dækker (a, v), viser vi i lemma 16.15 at v ikke er nødvendig for de forbindelser, der skal findes senere i rekursionen. Tilsvarende gælder om knuder i U_2.

Vi kalder rekursivt på (Q_1, H_1) samt på (Q_2, H_2). Rekursionen stopper, når der ikke er flere indre knuder i Q_0.

Når rekursionen er slut, har vi for hver knude $v \in \mathcal{V}(H)$ en mængde $\mathcal{C}(Q, v)$, som består af de forbindelser vi har lavet i ovenstående konstruktion. Vi har altså forbindelser fra s og t samt fra et b i hver af de rekursive kald, hvor $v \in H_0$, se igen figur 16.5. Korrektheden af konstruktionen vises i afsnit 16.2.2.

Ordning

Vi skal nu redegøre for, hvorledes vi sørger for at mængderne er ordnede. I artiklen [Tho01] beskrives ikke hvorledes dette opnås. Vi opnår ordnede mængder på flg. måde. For at $\mathcal{C}(Q, v)$ skal være ordnet, sørger vi for at sætte forbindelserne ind i mængden i den rette rækkefølge. I den indledende konstruktion sættes først forbindelser fra s ind, derefter kaldes den rekursive procedure og til sidst sættes forbindelser fra t ind. I den rekursive procedure kaldes først rekursivt på (Q_1, H_1), derefter indsættes forbindelser fra b og til sidst kaldes rekursivt på (Q_2, H_2). På denne måde bliver forbindelserne ordnet ift. knuderne på Q.

Egenskaber for de konstruerede forbindelser

Vi viser, at de afstande vi har tildelt vores forbindelser svarer til afstanden i $H_0^\star$.

Lemma 16.14 *For en knude $v \in \mathcal{V}(H_0)$ gælder det at $\ell(a, v) = \delta_{H_0^\star}(a, v)$, $\ell(b, v) = \delta_{H_0^\star}(b, v)$ samt at $\ell(c, v) = \delta_{H_0^\star}(c, v)$.*

Bevis. Vi vil bevise påstanden ved induktion i rekursionsdybden.

I basistilfældet har vi at, $(Q_0, H_0) = (Q, H)$. Vi laver sssp fra a, b og c og og laver forbindelser fra disse knuder til knuderne i H med $\ell(a, v) = \delta(a, v)$, $\ell(b, v) = \delta(b, v)$ og $\ell(c, v) = \delta(c, v)$. Da en forbindelserne dækker over en eksisterende vej, har vi at udsagnet er opfyldt.

Antag, at udsagnet gælder for (Q_0, H_0), vi skal vise, at det gælder for (Q_1, H_1) og (Q_2, H_2). Vi viser, at udsagnet gælder for (Q_1, H_1), beviset for (Q_2, H_2) er analogt.

Pr. induktionsantagelse gælder udsagnet for $a \in Q_0$ og $b \in Q_0$ i $H_0^\star$. Vi sender forbindelserne fra a og b med ned i rekursionen. Dvs. hvis $v \in H_1$ har vi at $\ell(a, v) = \delta_{H_1^\star}(a, v)$, da vi i $H_1^\star$ ikke tilføjer længere eller kortere veje fra a og c i forhold til $H_0^\star$. Dette skyldes at hvis en korteste vej går via knuder der er i U_1 og dermed ikke er med i H_1 er denne vej en del af de forbindelser vi har fra a og c. Knuden $b \in Q_0$ bliver til $c \in Q_1$ og udsagnet er da opfyldt med samme argument som for a. For $b \in Q_1$ er udsagnet opfyldt pr. konstruktion da vi i $H_1^\star$ laver en sssp fra b og laver forbindelserne med $\ell(b, v) = \delta_{H_1^\star}(b, v)$. □

Ændringer i konstruktionen

Vi har nu beskrevet hvilke forbindelser, der for en knude $v \in \mathcal{V}(H)$ sættes ind i $\mathcal{C}(Q, v)$ samt hvorledes disse kan konstrueres.

Vi beskriver nu nogle simple ændringer i konstruktionen. Forbindelserne, der findes er de samme, men vi laver om i konstruktionen således at vi ikke skal opbygge $H_0^\star$, men alligevel kan give forbindelserne en længde, der svarer til afstanden i $H_0^\star$.

I den oprindelige konstruktion anvender vi $H_0^\star$, når vi laver *sssp* fra b, hvor $H_0^\star$ består af H_0, Q_0 samt forbindelser fra a og c til knuder i H_0. I den ændrede konstruktion indsættes ikke forbindelser fra a og c, så vi arbejder udelukkende på $H_0 \cup Q_0$, hvorfor den afstand vi beregner i vores *sssp* er $\delta_{H_0 \cup Q_0}(b, v)$.

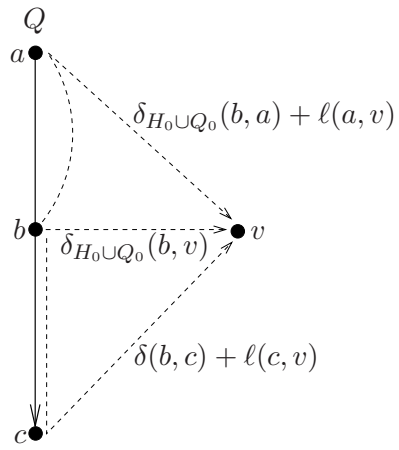
For at få de samme afstande som dem vi fandt i $H_0^\star$, skal vi nu se hvilke afstande vi kan få ved at anvende forbindelserne fra hhv. a og c til v, da vi mangler disse i $\delta_{H_0 \cup Q_0}$. Fra lemma 16.14 ved vi om disse forbindelser at $\ell(a, v) = \delta_{H_0^\star}(a, v)$ og $\ell(c, v) = \delta_{H_0^\star}(c, v)$. For en knude $v \in H_0$ finder vi derfor afstande fra b til v, der går over hhv. a og c på flg. måde:

- Hvis der findes en sti fra b til a i $H_0 \cup Q_0$ anvender vi afstanden af denne samt længden af forbindelsen (a, v), dvs. vi beregner afstanden over a som $\delta_{H_0 \cup Q_0}(b, a) + \ell(a, v)$. Hvis der ingen vej er fra b til a sætter vi $\delta_{H_0 \cup Q_0}(b, a) = \infty$.

- Afstanden over c beregnes vi vha. stykket på Q_0 mellem b og c, samt forbindelsen (c, v), dvs. vi beregner afstanden som $\delta(b, c) + \ell(c, v)$.

Vi kan nu finde afstanden fra b til v, som den er i $H_0^\star$ vha. følgende beregning, se figur 16.6:

$$\delta_{H_0^\star}(b, v) = min\{\delta_{H_0 \cup Q_0}(b, v),\ \delta_{H_0 \cup Q_0}(b, a) + \ell(a, v),\ \delta(b, c) + \ell(c, v)\}$$



Figur 16.6: Forbindelser over a og c på Q samt forbindelsen fra b.

16.2.2 Korrekthed

Vi vil i dette afsnit vise, at konstruktionen af $\mathcal{C}(Q, v)$ er korrekt, dvs. at vi får konstrueret en ε-dækkende mængde.

Den næste sætning viser, at hvis en sti fra en knude x på Q til en knude v ødelægges (dvs. der fjernes knuder på stien), når vi laver rekursive kald, vil (x, v) allerede være ε-dækket. Vi fjerner derfor ikke knuder, der er nødvendige for de forbindelser vi skal lave senere i rekursionen.

Lemma 16.15 *Lad x være en knude i det indre af Q_0 og lad v være en vilkårlig knude i H således at $\delta(x, v) \leq \alpha$. Hvis en korteste sti P fra x til v forlader H_0, da vil (x, v) være ε-dækket.*

Bevis. Vi viser udsagnet ved induktion i rekursionsdybden.

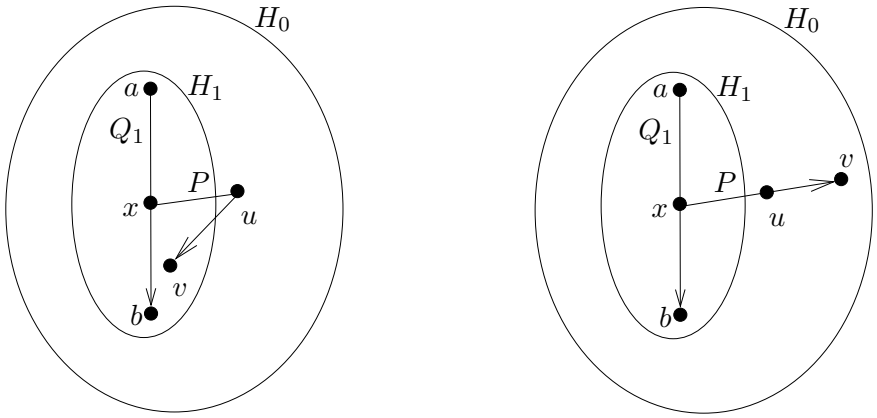
I basistilfældet har vi $(Q_0, H_0) = (Q, H)$. Her er udsagnet klart opfyldt, da ingen stier kan forlade H.

Vi antager nu, at udsagnet gælder for (Q_0, H_0), og vi skal vise, at udsagnet ligeledes gælder for (Q_1, H_1) – beviset for at udsagnet gælder for (Q_2, H_2) er analogt.

Vi ser nu på en knude x i det indre af Q_1 og betragter en korteste sti P fra x til v, som forlader H_1. Hvis P også forlader H_0 er (x, v) ε-dækket ifølge induktionsantagelsen. Vi ser derfor på tilfældet, hvor P forlader H_1, men ikke forlader H_0, se eksempler på figur 16.7.

Lad u være den første knude på P, der ikke befinder sig i H_1. Da $u \in H_0$ og $u \notin H_1$ må der gælde at $u \in U_1$. Vi ved da, at $\ell(a, u) > 2\alpha$ eller (b, u) semi-ε-dækker (a, u).

Vi viser først at $\ell(a, u) \leq 2\alpha$. Vi har, at længden af Q højst er α, dvs. $\delta(a, x) \leq \alpha$. Ligeledes er $\delta(x, v) \leq \alpha$ pr. antagelse. Desuden gælder følgende



Figur 16.7: En sti P, der forlader tolagsgrafen H_1, men er indeholdt i H_0.

formel.

$$\begin{aligned} &\delta(a,x) + \delta_{H_0}(x,u) \geq \delta_{H_0^\star}(a,u) && a \text{ og } x \text{ er knuder på } Q \\ &\Updownarrow \\ &\delta_{H_0}(x,u) \geq \delta_{H_0^\star}(a,u) - \delta(a,x) && (16.1) \end{aligned}$$

Vi kan nu regne os frem til at $\ell(a,u) \leq 2\alpha$.

$$\begin{aligned} \delta(x,v) &= \delta_{H_0}(x,u) + \delta_{H_0}(u,v) && P \text{ er kortest og forlader ikke } H_0 \\ &\geq \delta_{H_0^\star}(a,u) - \delta(a,x) + \delta_{H_0}(u,v) && \text{formel 16.1} \\ &= \ell(a,u) - \delta(a,x) + \delta_{H_0}(u,v) && \text{lemma 16.14} \end{aligned}$$

Hvilket betyder at $\ell(a,u) \leq 2\alpha$, da $\delta(a,x) \leq \alpha$ og $\delta(x,v) \leq \alpha$.

Da $u \in U_1$ ved vi nu, at (b,u) semi-ε-dækker (a,u) og vi ønsker at vise at $\delta(x,b) + \ell(b,v) \leq \delta(x,v) + \varepsilon\alpha$, da dette betyder at (x,v) er ε-dækket af (b,v). Dette kræver en del beregninger, vi vil starte med nogle små omskrivninger.

$$\begin{aligned} &\delta(a,b) + \ell(b,u) \leq \ell(a,u) + \varepsilon\alpha && (b,u) \text{ semi-}\varepsilon\text{-dækker } (a,u) \\ &\Updownarrow \\ &\ell(a,u) \geq \delta(a,b) + \ell(b,u) - \varepsilon\alpha && (16.2) \end{aligned}$$

$$\begin{aligned} \ell(b,u) + \delta_{H_0}(u,v) &= \delta_{H_0^\star}(b,u) + \delta_{H_0}(u,v) && \text{lemma 16.14} \\ &\geq \delta_{H_0^\star}(b,v) && (16.3) \end{aligned}$$

Vi er nu i stand til at regne os frem til det ønskede:

$$\begin{aligned}
\delta(x,v) &= \ell(a,u) - \delta(a,x) + \delta_{H_0}(u,v) \\
&\geq \delta(a,b) + \ell(b,u) - \varepsilon\alpha - \delta(a,x) + \delta_{H_0}(u,v) && \text{formel 16.2} \\
&= (\delta(a,b) - \delta(a,x)) + (\ell(b,u) + \delta_{H_0}(u,v)) - \varepsilon\alpha \\
&= \delta(x,b) + (\ell(b,u) + \delta_{H_0}(u,v)) - \varepsilon\alpha && a,\ x \text{ og } b \text{ er knuder på } Q \\
&\geq \delta(x,b) + \delta_{H_0^*}(b,v) - \varepsilon\alpha && \text{formel 16.3} \\
&= \delta(x,b) + \ell(b,v) - \varepsilon\alpha && \text{lemma 16.14}
\end{aligned}$$

Vi har nu vist, at (b,v) ε-dækker (x,v) og således opfylder (Q_1, H_1) udsagnet, der skulle vises. □

Vi kan nu i det følgende lemma konkludere at vi har konstrueret en mængde, der er ε-dækkende. Dermed er konstruktionen korrekt.

Lemma 16.16 *For en knude $v \in \mathcal{V}(H)$ har vi konstrueret en ε-dækkende $\mathcal{C}(Q,v)$.*

Bevis. Lad b være en knude på Q og lad $\delta(b,v) \leq \alpha$. Vi skal nu vise, at (b,v) er ε-dækket.

Hvis b er en af endeknuderne s eller t på Q er (b,v) klart ε-dækket, da vi i vores indledende konstruktion forbinder b til v med $\ell(b,v) = \delta(b,v)$.

Hvis b ikke er en af endeknuderne på Q, ser vi på det kald (Q_0, H_0), hvor b er Q_0's midterknude. Fra lemma 16.15 ved vi, at enten er (b,v) ε-dækket i et tidligere kald, ellers vil en korteste vej fra b til v være i H_0. Det betyder, at når vi laver sssp fra b i $H_0^\star$, forbinder vi b og v med $\ell(b,v) = \delta(b,v)$.

□

16.2.3 Antal forbindelser i mængderne

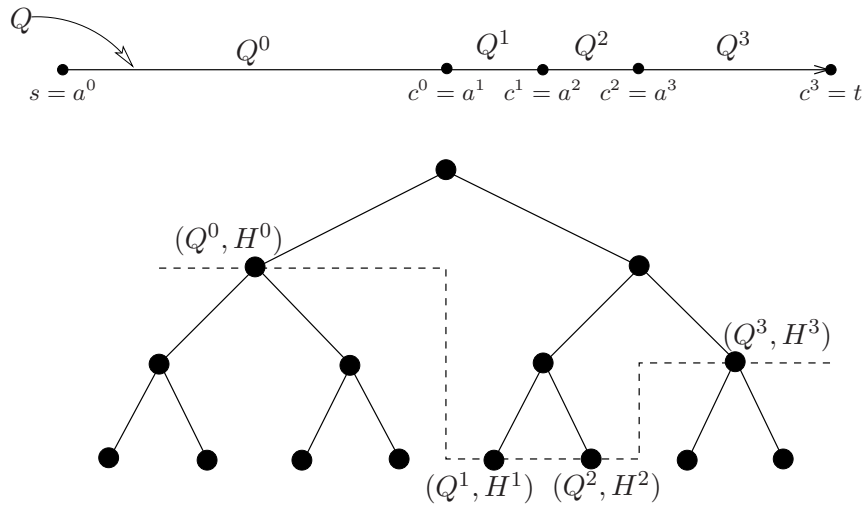
Vi vil nu se på, hvor store de konstruerede mængder er, dvs. hvor mange forbindelser en mængde $\mathcal{C}(Q,v)$ indeholder. Vi vil vise, at antallet af forbindelser til en knude $v \in H$ (dvs. størrelsen af $\mathcal{C}(Q,v)$) er $\mathcal{O}(\log(|\mathcal{V}(Q)|)/\varepsilon)$.

Vi har først to forbindelser til v fra hhv. s og t, der stammer fra den indledende konstruktion.

Fra den rekursive konstruktion har vi en forbindelse til v fra b i hvert rekursivt kald, hvor $v \in H_0$. Da vi i hvert rekursive kald deler Q op i to lige store dele, er rekursionsdybden $\mathcal{O}(\log(|\mathcal{V}(Q)|))$. Vi vil derfor vise, at der er $\mathcal{O}(1/\varepsilon)$ grene i rekursionen, der indeholder v, dvs. at det er $\mathcal{O}(1/\varepsilon)$ kald, hvor $v \in H_0$ og hvor v ikke er med i de rekursive kald, dvs. $v \notin H_1$ og $v \notin H_2$. Vi kalder disse kald for *slutkald for v*.

Vi bemærker, at hvis v udelukkende forekommer i rodkaldet, vil det ønskede klart være opfyldt fordi $\varepsilon \leq 1$ og vi i dette tilfælde kun har tre forbindelser til v. Vi antager derfor i det følgende, at v forekommer i flere kald end rodkaldet.

Definition 16.17 *En Q-kæde af rekursive kald defineres som en sekvens af kald $(Q^0, H^0), \ldots, (Q^k, H^k)$, der dækker hele Q på flg. måde: Hvis a^i og c^i er første hhv. sidste knude på Q^i så er $s = a^0$, $a^{i+1} = c^i$ samt $c^k = t$.*



Figur 16.8: Eksempel på en Q-kæde. Vi kan se hvordan kaldene i rekursionstræet svarer til dele af Q.

På figur 16.8 ser vi et eksempel på en Q-kæde. Kaldene $(Q^0, H^0), \ldots, (Q^3, H^3)$ er vist i rekursionstræet fra vores konstruktion og vi ser hvordan der i hvert kald indgår en del af Q.

Vi antager nu, at vi har en Q-kæde, hvor alle kald (Q^i, H^i) enten er kald, hvor $v \in H^i$ eller kald, hvor forælderkaldet indeholder v.

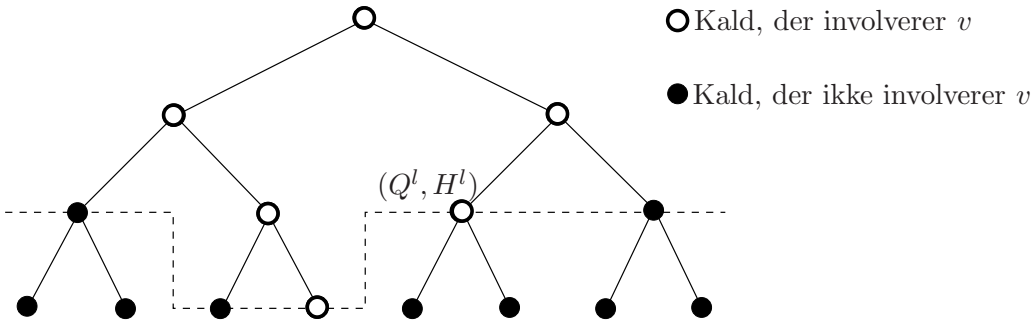
Dette betyder, at vi i hvert kald (Q^i, H^i) har forbindelser fra a^i til v samt fra c^i til v. Vi er nu interesserede at se på $d^i = \delta(a^i, c^i) + \ell(c^i, v) - \ell(a^i, v)$. Hvis $d^i \leq \varepsilon\alpha$ har vi at (c^i, v) semi-ε-dækker (a^i, v), hvilket vil sige, at hvis v er i H^i vil $d^i > \varepsilon\alpha$. Om d^i gælder flg. lemma:

Lemma 16.18 *For hver d^i gælder det at $d^i \geq 0$.*

Bevis. Fra lemma 16.14 har vi at grafen $H^{i\star}$ er således at $\ell(a^i, v) = \delta_{H^{i\star}}(a^i, v)$ og $\ell(c^i, v) = \delta_{H^{i\star}}(c^i, v)$. Vi ved ligeledes at $H^{i\star}$ indeholder Q^i og dermed gælder det, at $\delta_{H^{i\star}}(a^i, v) \leq \delta(a^i, c^i) + \delta_{H^{i\star}}(c^i, v)$. Dermed er $\ell(a^i, v) \leq \delta(a^i, c^i) + \ell(c^i, v)$, hvilket betyder, at $d^i \geq 0$. □

Lad nu $(Q^0, H^0), \ldots, (Q^k, H^k)$ være den Q-kæde, der består af alle slutkald for v samt kald, der er søskende til kald, der indeholder v, men som ikke selv indeholder v. Et eksempel kan ses på figur 16.9. For alle kaldene i kæden gælder, at de enten selv vil indeholde v eller at deres forælderkald indeholder v.

Lad (Q^l, H^l) være det sidste af kaldene på kæden, der involverer v. Da $v \in H^l$ ved vi at $\ell(a^l, v) \leq 2\alpha$. Vi ved ligeledes at længden af Q højst er α. Dermed har vi:



Figur 16.9: Eksempel på Q-kæde med alle slutkald for v. Kaldet (Q^l, H^l) er det sidste kald, der involverer v.

$$\begin{aligned}
\sum_{0 \le i < l} d^i &= \delta(a^0, c^0) + \ell(c^0, v) - \ell(a^0, v) \\
&\quad + \delta(a^1, c^1) + \ell(c^1, v) - \ell(a^1, v) \\
&\quad + \dots \\
&\quad + \delta(a^{l-1}, c^{l-1}) + \ell(c^{l-1}, v) - \ell(a^{l-1}, v) \\
&= \delta(a^0, c^{l-1}) + \ell(c^{l-1}, v) - \ell(a^0, v) \\
&= \delta(a^0, a^l) + \ell(a^l, v) - \ell(a^0, v) \\
&\le \alpha + 2\alpha - \ell(a^0, v) \\
&\le 3\alpha
\end{aligned}$$

Da $d^i \ge 0$ ifølge lemma 16.18 og da $d^i > \varepsilon\alpha$ hvis (Q^i, H^i) involverer v, kan vi konkludere at antallet af slutkald for v højst er $3\alpha/\varepsilon\alpha + 1 = 3/\varepsilon + 1 \in \mathcal{O}(1/\varepsilon)$, som ønsket.

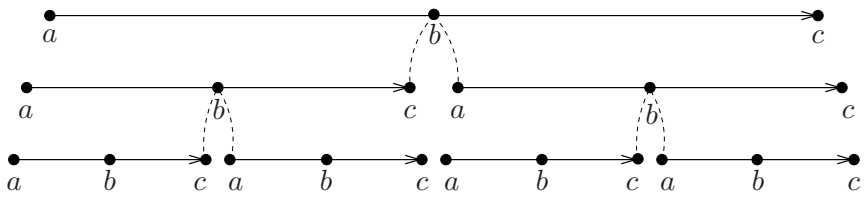
Vi konkluderer at mængden af forbindelser fra Q til v har den ønskede størrelse: $|\mathcal{C}(Q, v)| \in \mathcal{O}(\log(|\mathcal{V}(Q)|)/\varepsilon)$, da rekursionsdybden er $\mathcal{O}(\log(|\mathcal{V}(Q)|))$.

16.2.4 Effektivitet

Vi mangler nu at vise, hvor effektiv vores konstruktion er. Vi viser derfor, hvor lang tid vi anvender for at konstruere en ε-dækkende mængde. Vi bemærker at dette bevis ikke svarer til beviset i artiklen [Tho01], da der i denne antages, at der er implementeret *sssp* i lineær tid samt at der foretages ændringer på $H_0 \cup Q_0$. Vi har derfor selv konstrueret dette bevis, der stemmer overens med vores implementation, se afsnit 16.4.

Vi antager, at udførelsestiden for *sssp* (single source shortest paths) er $\mathcal{O}(m \log(n))$ for en graf med n knuder og m kanter. Dette kan man opnå ved at anvende Dijkstras algoritme.

I den indledende konstruktion laver vi *sssp* to gange i H, fra hhv. a og c, der er hhv. første og sidste knude på den sti Q, vi laver forbindelser fra. Da H er planar anvender vi tid $\mathcal{O}(|\mathcal{V}(H)| \log(|\mathcal{V}(H)|))$ på disse to gange *sssp*.



Figur 16.10: Q opdeles rekursivt. Når en knude er den midterste (b), kommer den med i begge rekursive kald, ellers er den kun med i et rekursivt kald.

Vi ser nu på den anvendte tid i den rekursive procedure. Vi viser i det følgende, hvor meget tid vi anvender pr. knude, der er med i $H_0 \cup Q_0$ i et rekursivt kald. Først viser vi, i hvor mange kald, en knude er med i H_0 samt hvor mange kald, en knude er med i Q_0.

Vi har fra afsnit 16.2.3, at en knude $v \in H$ højst er med i grafen H_0 i $\mathcal{O}(\log(|\mathcal{V}(Q)|)/\varepsilon)$ rekursive kald.

Vi ser nu på, i hvor mange kald en knude v kan være med i Q_0. I det første kald er alle knuder fra Q på Q_0. Når Q_0 deles op i to dele og der kaldes rekursivt, vil alle knuder bortset fra den midterste knude b komme med i et af de rekursive kald. Knuden b kommer med i begge rekursive kald. Se en illustration på figur 16.10. Hver knude vil på et tidspunkt være b-knude, men kun én gang. Derfor kan en knude i Q være med i to stier af rekursive kald, dvs. antallet af gange hver knude er med i Q_0 er $\mathcal{O}(\log(|\mathcal{V}(Q)|))$.

I hvert kald laver vi *sssp* fra b i $H_0 \cup Q_0$. Grafen $H_0 \cup Q_0$ er planar da den er en delgraf af H, derfor tager *sssp* i $H_0 \cup Q_0$ tid $\mathcal{O}(|\mathcal{V}(H_0 \cup Q_0)| \log(|\mathcal{V}(H_0 \cup Q_0)|))$. Vi anvender altså tid $\mathcal{O}(\log(|\mathcal{V}(H_0 \cup Q_0)|)) \in \mathcal{O}(\log(|\mathcal{V}(H)|))$ pr. knude, der er i $H_0 \cup Q_0$.

Vi kan nu konkludere, at vi for hver knude $v \in H$ samlet anvender tid

$$\begin{aligned}&\mathcal{O}(\log(|\mathcal{V}(H)|)\log(|\mathcal{V}(Q)|) + \log(|\mathcal{V}(H)|)\log(|\mathcal{V}(Q)|)/\varepsilon)\\ &\in \mathcal{O}(\log(|\mathcal{V}(H)|)\log(|\mathcal{V}(Q)|)/\varepsilon)\end{aligned}$$

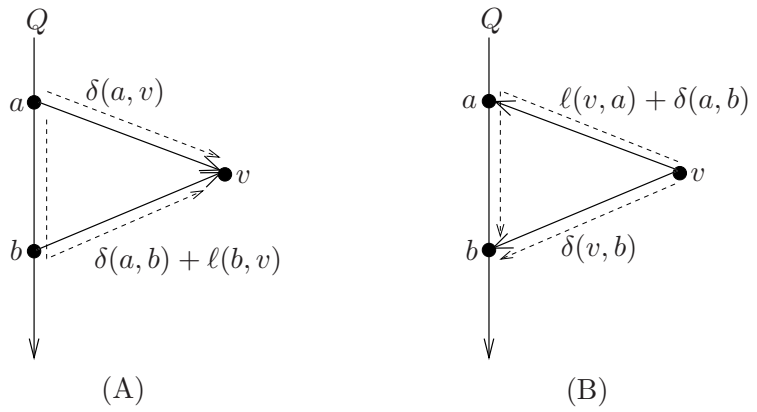
Det første led svarer til de kald, hvor v er med i Q_0 og det andet led svarer til de kald, hvor v er med i H_0. Den samlede tid for at konstruere ε-dækkende mængder for alle knuder i H, inkl. den indledende konstruktion, er da

$$\mathcal{O}(|\mathcal{V}(H)|\log(|\mathcal{V}(H)|)\log(|\mathcal{V}(Q)|)/\varepsilon)$$

16.2.5 Samlet tid for at finde forbindelser i H

Vi kan nu konkludere, hvor lang tid vi anvender i et rekursivt kald på en $(3, \alpha)$-lagsgraf H i vores dekomposition på at finde forbindelser over de $\mathcal{O}(1)$ separatorstier, der er i den separator S, som vi opdeler H efter. Da S højst kan bestå af alle knuder i H, er det samlede antal knuder i separatorstierne $\mathcal{O}(|\mathcal{V}(H)|)$. Den samlede tid vi anvender på at finde mængder af forbindelser i et rekursivt kald på H i dekompositionen er da

$$\mathcal{O}(|\mathcal{V}(H)|\log^2(|\mathcal{V}(H)|)/\varepsilon) \tag{16.4}$$



Figur 16.11: Illustration af forbindelser, der *ikke* eksisterer i rensede mængder.

16.3 Effektiv approksimering af afstande

Vi vil nu redegøre for hvorledes vi effektivt kan approksimere afstande over separatorstier.

For at gøre det mere effektivt at finde en afstand fra v til w over stien Q vil vi mindske størrelsen af de mængder af forbindelser, der er mellem knuder og separatorstier. Når vi mindsker størrelsen af mængderne, sørger vi for at de har en form, der gør, at vi effektivt kan beregne $dist(\mathcal{C}(v,Q),\mathcal{C}(Q,w))$.

Definition 16.19 *Mængden $\mathcal{C}(v,Q)$ er* renset *(på engelsk* clean*), hvis den ikke indeholder to forskellige forbindelser (v,a) og (v,b) hvorom det gælder, at $\ell(v,a)+\delta(a,b) \leq \ell(v,b)$.*

Ligeledes er mængden $\mathcal{C}(Q,v)$ renset*, hvis den ikke indeholder to forskellige forbindelser (a,v) og (b,v) hvorom det gælder, at $\delta(a,b)+\ell(b,v) \leq \ell(a,v)$.*

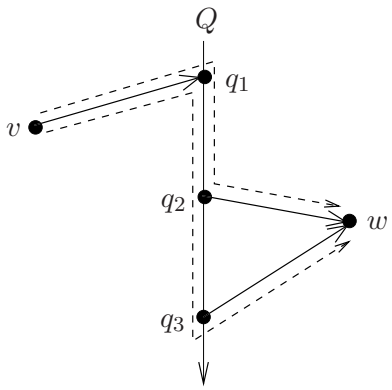
Der findes altså ikke (v,a) og (v,b) i en renset mængde $\mathcal{C}(v,Q)$ således at vejen fra v til b over forbindelsen (v,a) er kortere end forbindelsen (v,b) fra v til b. Se figur 16.11 (B).

Ligeledes gælder det for en renset mængde $\mathcal{C}(Q,v)$, at mængden ikke indeholder forbindelser (a,v) og (b,v) således at vejen fra a til v via forbindelsen (b,v) er kortere end den direkte forbindelse (a,v). Se figur 16.11 (A).

Vi kan nu vise, at hvis vi renser vores mængder, kan vi effektivt beregne $dist(\mathcal{C}(v,Q),\mathcal{C}(Q,w))$.

Lemma 16.20 *Hvis $\mathcal{C}(v,Q)$ og $\mathcal{C}(Q,w)$ er ordnede og rensede, kan vi finde $dist(\mathcal{C}(v,Q),\mathcal{C}(Q,w))$ i tid $\mathcal{O}(|\mathcal{C}(v,Q)|+|\mathcal{C}(Q,w)|)$.*

Bevis. De to mængder flettes vha. deres ordning mht. Q til en liste L. Hvis to elementer har samme orden, vælges elementet fra $\mathcal{C}(v,Q)$ først. Dette betyder, at hvis der er et element $(v,c) \in \mathcal{C}(v,Q)$ og et element $(c,w) \in \mathcal{C}(Q,w)$, vil (v,c) komme før (c,w) i den sammenflettede liste, L. Sammenfletningen tager lineær tid i den samlede størrelse af mængderne.



Figur 16.12: Vi har en forbindelse $(v, q_1) \in \mathcal{C}(v, Q)$ samt forbindelserne $(q_2, w), (q_3, w) \in \mathcal{C}(Q, w)$. Hvis de to mængder er rensede, vil vejen via (v, q_1) og (q_2, w) altid være kortere end vejen via (v, q_1) og (q_3, w).

Herefter gennemløbes L, for at finde forbindelserne $(v, a) \in \mathcal{C}(v, Q)$ og $(b, w) \in \mathcal{C}(Q, w)$, som minimerer $dist((v, a), (b, w))$. Da begge mængder af forbindelser er ordnede og rensede, vil (v, a) og (b, w) forekomme umiddelbart efter hinanden i L og kan derfor findes vha. et lineært gennemløb af L. □

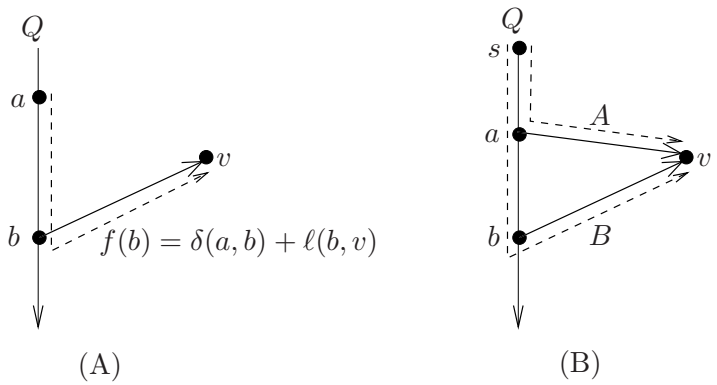
På figur 16.12 ses et eksempel på, at det kun er forbindelser, der følger umiddelbart efter hinanden, der er kandidater til at minimere $dist((v, a), (b, w))$. Her ses to mulige veje fra v til w, hhv. via forbindelsen (q_2, w) og via forbindelsen (q_3, w). Da $\mathcal{C}(Q, w)$ er renset, vil forbindelsen (q_2, w) være kortere end vejen fra q_2 til w via forbindelsen (q_3, w).

På baggrund af ovenstående lemma er vi interesserede i, at vi givet en ordnet ε-dækkende mængde $\mathcal{D}(Q, v)$ kan konstruere en ordnet og renset ε-dækkende $\mathcal{C}(Q, v)$. Det følgende lemma giver os en sådan konstruktion og giver os ligeledes en grænse på størrelsen af $\mathcal{C}(Q, v)$.

Lemma 16.21 *Givet en ordnet ε_0-dækkende mængde $\mathcal{D}(Q, v)$, som indeholder forbindelser fra Q til v, kan vi konstruere en renset og ordnet $(\varepsilon_0+\varepsilon_1)$-dækkende mængde $\mathcal{C}(Q, v) \subseteq \mathcal{D}(Q, v)$ af størrelse højst $1 + (2 + \varepsilon_0)/\varepsilon_1 \in \mathcal{O}(1/\varepsilon_1)$ i tid $\mathcal{O}(|\mathcal{D}(Q, v)|)$.*

Bevis. $\mathcal{C}(Q, v)$ konstrueres som følger. Alle forbindelser $(a, v) \in \mathcal{D}(Q, v)$, hvorom det gælder, at $\ell(a, v) > \alpha + \varepsilon_0\alpha$ fjernes fra $\mathcal{D}(Q, v)$. Herefter besøges de tilbageværende forbindelser i baglæns orden. Den første forbindelse (c, v) tilføjes altid til $\mathcal{C}(Q, v)$. Lad nu (b, v) være den sidste forbindelse, der er tilføjet til $\mathcal{C}(Q, v)$, når vi ser på en forbindelse (a, v). Vi tilføjer (a, v) til $\mathcal{C}(Q, v)$ hvis $\delta(a, b) + \ell(b, v) > \ell(a, v) + \varepsilon_1\alpha$ (dvs. hvis (b, v) *ikke* semi-ε_1-dækker (a, v)). Denne konstruktion tager klart lineær tid i størrelsen af $\mathcal{D}(Q, v)$.

Mængden $\mathcal{C}(Q, v)$ er renset pr. konstruktion. Vi vil nu argumentere for, at $\mathcal{C}(Q, v)$ er $(\varepsilon_0 + \varepsilon_1)$-dækkende. Se på en vilkårlig knude $a \in Q$ (dog bortset fra den sidste knude på Q, da vi ved at denne er dækket af $\mathcal{C}(Q, v)$). Vi skal vise, at hvis (a, v) ikke er i $\mathcal{C}(Q, v)$ så er (a, v) $(\varepsilon_0 + \varepsilon_1)$-dækket af en forbindelse i $\mathcal{C}(Q, v)$.



Figur 16.13: (A) Illustration af $f(b)$ (B) Vejen fra s til v bliver kortere, når vi tilføjer en ny forbindelse (a, v).

Vælg b således at $i(a, Q) \leq i(b, Q)$, $(b, v) \in \mathcal{D}(Q, v)$ og således at $f(b) = \delta(a, b) + \ell(b, v)$ er minimeret (se en illustration af $f(b)$ på figur 16.13 (A)). Da $\mathcal{D}(Q, v)$ er en ε_0-dækkende mængde ved vi, at $f(b) \leq \delta(a, v) + \varepsilon_0\alpha$, dvs. at (b, v) ε_0-dækker (a, v).

Hvis (b, v) ikke er blevet inkluderet i $\mathcal{C}(Q, v)$, er det fordi, der eksisterer $(c, v) \in \mathcal{C}(Q, v)$ således at $\delta(b, c) + \ell(c, v) \leq \ell(b, v) + \varepsilon_1\alpha$. Da har vi

$$\begin{aligned}\delta(a, c) + \ell(c, v) &= \delta(a, b) + \delta(b, c) + \ell(c, v) \\ &\leq \delta(a, b) + \ell(b, v) + \varepsilon_1\alpha \\ &= f(b) + \varepsilon_1\alpha \\ &\leq \delta(a, v) + (\varepsilon_0 + \varepsilon_1)\alpha\end{aligned}$$

Dermed er (a, v) $(\varepsilon_0 + \varepsilon_1)$-dækket af (c, v).

Vi skal nu argumentere for, at $\mathcal{C}(Q, v)$ er tilpas lille. Lad s være den første knude på stien Q. Vi observerer, at hver gang vi tilføjer en forbindelse (a, v) (efter at vi har tilføjet den første forbindelse (c, v)) til $\mathcal{C}(Q, v)$, mindsker vi afstanden fra s til v med mindst $\varepsilon_1\alpha$.

Lad (b, v) være den sidste forbindelse, der blev tilføjet før (a, v). Pr. konstruktion ved vi da, at $\delta(a, b) + \ell(b, v) > \ell(a, v) + \varepsilon_1\alpha$. På figur 16.13 (B) ser vi den gamle afstand B fra s til v og den nye afstand A. Vi kan nu vise at afstanden er blevet formindsket med mindst $\varepsilon_1\alpha$

$$\begin{aligned}B - A &= (\delta(s, b) + \ell(b, v)) - (\delta(s, a) + \ell(a, v)) \\ &= \delta(a, b) + \ell(b, v) - \ell(a, v) \\ &> \varepsilon_1\alpha\end{aligned}$$

Afstanden fra s til v, når vi har tilføjet den første forbindelse (c, v) er

$$\begin{aligned}\delta(s, c) + \ell(c, v) &\leq \alpha + (\alpha + \varepsilon_0\alpha) \\ &= 2\alpha + \varepsilon_0\alpha\end{aligned}$$

Vi kan derfor tilføje mindre end $(2\alpha + \varepsilon_0\alpha)/\varepsilon_1\alpha$ forbindelser udover (c, v). Der er altså i alt højst $1 + (2 + \varepsilon_0)/\varepsilon_1$ forbindelser. □

16.3.1 Sammenfatning af konstruktion og ressourceforbrug

Vi giver nu en sammenfatning af, hvorledes vi for en graf H med en separatorsti S konstruerer ordnede, rensede, $\varepsilon/2$-dækkende mængder af størrelse $\mathcal{O}(1/\varepsilon)$.

For en separatorsti $Q \in S$ anvender vi først konstruktionen fra afsnit 16.2.1 med $\varepsilon' = \varepsilon/2$ til at konstruere ordnede mængder af størrelse $\mathcal{O}(\log(|\mathcal{V}(Q)|/\varepsilon))$. Tiden for at konstruere mængder mellem alle stier $Q \in S$ og alle knuder $v \in H$ er $\mathcal{O}(|\mathcal{V}(H)| \log^2(|\mathcal{V}(H)|)/\varepsilon)$ ifølge. formel 16.4.

Herefter anvendes konstruktionen fra beviset for lemma 16.21 med $\varepsilon_0 = \varepsilon_1 = \varepsilon/2$, hvilket giver os ordnede, rensede, $\varepsilon/2$-dækkende mængder af størrelse $\mathcal{O}(1/\varepsilon)$. Til denne fase anvender vi tid, der er proportional med størrelsen af de oprindelige mængder. For alle stier $Q \in S$ og alle knuder $v \in H$ anvender vi da tid $\mathcal{O}(|\mathcal{V}(H)| \log(|\mathcal{V}(H)|/\varepsilon))$ på denne fase.

For en knude $v \in H$ og en sti $Q \in S$, konstruerer vi da mængderne $\mathcal{C}(Q, v)$ og $\mathcal{C}(v, Q)$. Disse gemmes i lister ind_v og ud_v i knuden svarende til v i den graf G_i, som dekomponeres. Listerne $\mathcal{C}(Q, v)$ og $\mathcal{C}(v, Q)$ har som nævnt størrelse $\mathcal{O}(1/\varepsilon)$.

Korollar 16.22 *For hver $v \in H$ og $Q \in S$ kan vi konstruere mængderne $\mathcal{C}(Q, v)$ og $\mathcal{C}(v, Q)$ af størrelse $\mathcal{O}(1/\varepsilon)$. Til dette anvender vi totalt tid*

$$\mathcal{O}(|\mathcal{V}(H)| \log(|\mathcal{V}(H)|) \log(|\mathcal{V}(Q)|)/\varepsilon)$$

Den samlede tid for konstruktionen for alle stier $Q \in S$ er

$$\mathcal{O}(|\mathcal{V}(H)| \log^2(|\mathcal{V}(H)|)/\varepsilon)$$

Da der er et konstant antal stier i S, tilføjer vi for en knude $v \in H$ mængder til listerne ind_v og ud_v, der samlet har en størrelse på $\mathcal{O}(1/\varepsilon)$.

16.4 Implementation

Vi anvender konstruktionen fra lemma 16.11 med $\varepsilon' = \varepsilon/2$, når vi konstruerer vores mængder. Givet en sti Q og en graf H, starter vi med at lave *sssp* fra startknuden s og slutknuden t på Q. Vi forbinder s til knuderne i H, hvorefter vi kalder den rekursive procedure og sidst forbinder vi t til knuderne i H.

Den rekursive procedure får en graf H_0 og en sti Q_0 samt arrays, hvor længden af forbindelserne fra endeknuderne a og c på Q_0 kan slås op. Vi finder midterknuden b og laver *sssp* fra denne. Vi ser herefter om vejen over a eller c er kortere – den korteste vej vi finder gemmes. Vi konstruerer Q_1 og H_1 og kalder rekursivt på disse (her sendes arrays med længder af forbindelser fra a og b med). Herefter forbinder vi b til knuderne i H_0. Sidst konstruerer vi Q_2 og H_2 og kalder rekursivt på disse (samt arrays for hhv. b og c).

Bemærk, at vi laver forbindelserne i en rækkefølge, der svarer til rækkefølgen af knuderne på Q, som vi forbinder fra.

Herefter anvender vi konstrutionen fra beviset for lemma 16.21 med $\varepsilon_0 = \varepsilon_1 = \varepsilon/2$ til at lave mængder, der er rensede, ordnede og af størrelse $\mathcal{O}(1/\varepsilon)$.

Når vi for en knude $v \in H$ har mængden $\mathcal{C}(Q, v)$ af forbindelser fra Q til v indsættes denne mængde i v's indliste ind_v, som findes i knuden svarende til v i den graf G_i, som dekomponeres. Hvis stien Q har indeks q, sættes mængden ind på plads q i ind_v. Vi konstruerer ligeledes $\mathcal{C}(v, Q)$, som indsættes i v's udliste ud_v.

Tiden og pladsforbrunget for vores konstruktion følger af korollar 16.22.

16.4.1 Placering i programmet

Forbindelser konstrueres i `distance/connections.cpp`. Funktionerne vi anvender tilhører klassen `ThreeAlphaGraph`.

Kapitel 17

Samlet α-orakel og forespørgsler

I dette kapitel vil vi først kort opsummere hvordan vi konstruerer et α-orakel og hvilke dele dette indeholder. Vi opsummerer hvorledes vi konstruerer mængder af forbindelser, der er ordnede, rensede og har en passende størrelse.

Vi beskriver herefter hvorledes vi i et α-orakel laver forespørgsler på afstanden mellem to knuder samt egenskaberne for den approksimerede afstand, vi finder i en sådan forespørgsel. Vi beskriver først hvordan en forespørgsel fortages i et α-orakel, og vi argumenterer for at udførelsestiden for en forespørgsel er $\mathcal{O}(\log(n)/\varepsilon)$. Derefter beskriver vi, hvilke afstande, der kan findes med et α-orakel samt relaterer den afstand, som returneres, til den reelle afstand. Den returnerede afstand er konservativ og højst $\varepsilon\alpha$ for stor.

17.1 Den samlede datastruktur for et α-orakel

Vi giver nu en kort gennemgang af, hvorledes de konstruktioner, der er beskrevet i kapitel 14 til 16 anvendes i den samlede konstruktion af et α-orakel.

Givet en orienteret, planar og sammenhængende graf G, med kantvægte op til α, opdeles G først i lag og vi konstruerer $(3,\alpha)$-lagsgraferne $G_0, \ldots, G_{k-2}$, som det er beskrevet i kapitel 14. På hver af disse grafer udfører vi basisrekursionen, som den er beskrevet i kapitel 15.4.

I hvert rekursionskald konstrueres forbindelser mellem den $(3,\alpha)$-lagsgraf H der er givet, samt de separatorstier, der anvendes til at opdele H. For hver orienteret sti Q i separatoren S, konstrueres en $\varepsilon/2$-dækkende mængde af størrelse $\mathcal{O}(1/\varepsilon)$. Hvorledes dette gøres er sammenfattet i afsnit 16.3.1, hvor vi konkluderer, at der anvendes tid $\mathcal{O}(|\mathcal{V}(H)|\log^2(|\mathcal{V}(H)|)/\varepsilon)$ for at finde forbindelser.

Vi konstruerer for hver $(3,\alpha)$-lagsgraf G_i et rekursionstræ vha. basisrekursionen, som vi gjorde det i del I. Rekursionsdybden er logaritmisk, da størrelsen af graferne, der arbejdes på, halveres for hvert kald. Konstruktionstiden i et rekursivt kald er domineret af tiden for at konstruere forbindelser, hvorfor den samlede konstruktionstid for del-oraklet for G_i er

$$\mathcal{O}(|\mathcal{V}(G_i)|\log^3(|\mathcal{V}(G_i)|)/\varepsilon) \tag{17.1}$$

Rekursionstræet, delgraferne og de øvrige ting, der konstrueres, fylder som i reachability-tilfældet $\mathcal{O}(|\mathcal{V}(G_i)|\log(|\mathcal{V}(G_i)|)$. I dette tilfælde har vi dog forbin-

delser, der anvender mere plads end de forbindelser, vi lavede i reachability-oraklet. Hver indgang i listerne ud_v og ind_v kræver nu $\mathcal{O}(1/\varepsilon)$, hvor de før krævede konstant plads. Det samlede pladsforbrug er da

$$\mathcal{O}(|\mathcal{V}(G_i)| \log(|\mathcal{V}(G_i)|)/\varepsilon) \tag{17.2}$$

Det samlede α-orakel består af et rekursionstræ for hver af $(3, \alpha)$-lagsgraferne $G_0, \ldots, G_{k-2}$. Da den samlede størrelse af disse grafer ifølge sætning 14.7 er lineær i størrelsen af G, er den samlede konstruktionstid for α-oraklet

$$\mathcal{O}(n \log^3(n)/\varepsilon) \tag{17.3}$$

Ligeledes vha. sætning 14.7 har vi at det samlede pladsforbrug for det samlede α-orakel er

$$\mathcal{O}(n \log(n)/\varepsilon) \tag{17.4}$$

17.1.1 Placering i programmet

Klassen `distance/DistanceOracle` indeholder implementationen af den samlede α-orakel.

17.2 Forespørgsel i et α-orakel

Givet to knuder v og w i en graf G samt et α-orakel O for grafen G antager vi, at der findes en vej fra v til w i G. Vi finder først de (3,α)-lagsgrafer, som indeholder både v og w. I hver af disse grafer G_i udfører vi flg. skridt:

- Vi finder final call for v og w i rekursionstræet som udgør del-oraklet for G_i. Vi finder den nærmeste fælles forfader i rekursionstræet for disse final calls – $nca(v, w)$.
- Vi finder separatornummeret s i $nca(v, w)$. Vi erindrer, at dette er det sidste nummer, der er anvendt til at nummerere separatorstier fra roden ned til $nca(v, w)$ i rekursionstræet.
- Vi gennemløber nu alle separatorstier fra nummer 0 til s. For en separatorsti Q finder vi $\mathcal{C}(v, Q)$ og $\mathcal{C}(Q, w)$ og disse flettes sammen, så vi finder $dist(\mathcal{C}(v, Q), \mathcal{C}(Q, w))$, som beskrevet i beviset for lemma 16.20. Vi finder dermed den korteste afstand oraklet kender mellem v og w over Q, hvis en vej over Q findes. Vi gemmer den korteste afstand vi finder mellem v og w over de stier vi gennemløber.

I hver G_i, som indeholder v og w finder vi en korteste afstand. Vi finder minimum over disse. Er dette minimum højst $\alpha + \varepsilon\alpha$, returneres det som resultatet af forespørgslen, ellers returneres ∞.

Antallet af separatorstier, der skal gennemløbes for hver (3,α)-lagsgraf G_i er $\mathcal{O}(\log(|\mathcal{V}(G_i)|))$, da dette er højden af rekursionstræet. Da antallet af (3,α)-lagsgrafer vi skal se på højst er tre, er det samlede antal separatorstier vi skal gennemløbe $\mathcal{O}(\log(n))$, da den samlede størrelse af $G_0, \ldots, G_{k-2}$ er lineær i

størrelsen af G, jf. sætning 14.7. Da $\mathcal{C}(v, Q)$ og $\mathcal{C}(Q, w)$ ifølge lemma 16.21 kan konstrueres, så de har størrelse $\mathcal{O}(1/\varepsilon)$ kan vi ifølge lemma 16.20 flette disse lister i tid $\mathcal{O}(1/\varepsilon)$. Den samlede udførelsestid for en forespørgsel er da

$$\mathcal{O}(\log(n)/\varepsilon) \tag{17.5}$$

17.2.1 Implementation af forespørgsel

Implementationen følger de skridt, der er beskrevet i afsnit 17.2. Givet to knuder v og w starter vi med at finde de G_i-grafer, som indeholder begge knuder. I en sådan graf finder vi final call for v og w og disses nærmeste fælles forfader $nca(v, w)$ i rekursionstræet. Alle separatorstier fra $nca(v, w)$ op til roden i rekursionstræet gennemløbes og den korteste fundne afstand huskes. Hvis der er flere G_i-grafer, der indeholder v og w vælges den korteste afstand, der er fundet i en af disse. Denne afstand returneres, hvis den er kortere end $\alpha + \varepsilon\alpha$. Hvis ingen vej findes mellem v og w eller hvis den fundne afstand er større end $\alpha+\varepsilon\alpha$ returneres `MAXINT`. Udførelsestiden for en forespørgsel er ifølge formel 17.5 $\mathcal{O}(\log(n)/\varepsilon)$.

Placering i programmet

Klassen `distance/DistanceOracle`, som implementerer et α-orakel, indeholder en funktion `query`, som foretager en forespørgsel i et α-orakel.

17.3 Begrænsning til afstande på α

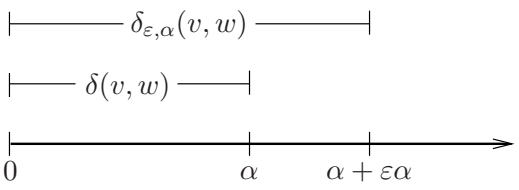
Når vi anvender et α-orakel, får vi en approksimeret afstand, der højst er $\alpha+\varepsilon\alpha$. Vi vil i dette afsnit se på hvilke resultater et α-orakel giver ift. den korteste vej.

Definition 17.1 *Lad et α-orakel konstrueret med et givet ε for en graf G være givet. For to knuder v og w i G, definerer vi $\delta_{\varepsilon,\alpha}(v, w)$ til at være den afstand fra v til w, som oraklet returnerer.*

Vi ved at det resultat vi returnerer altid er konservativt, da resultatet er beregnet på baggrund af forbindelser, hvis længde aldrig er kortere end den reelle korteste afstand. Fra lemma 16.9 (under anvendelse af et $\varepsilon' = \varepsilon/2$) har vi, at hvis der findes en korteste vej P fra v til w og $\delta(v, w) \leq \alpha$, får vi et resultat, der højst er $\varepsilon\alpha$ for stort (da P krydser en af de stier, der adskiller v fra w). Vi konkluderer at vores α-orakel finder et resultat der overholder flg. udsagn:

$$\delta(v, w) \leq \delta_{\varepsilon,\alpha}(v, w) \leq \delta(v, w) + \varepsilon\alpha \tag{17.6}$$

Vores α-orakel giver os altså en afstand, der er konservativ samt højst $\varepsilon\alpha$ for stor. På figur 17.1 ser vi hvilke afstande der kan returneres samt for hvilke reelle korteste afstande, man kan forvente at få et resultat, der overholder udsagn 17.6.



Figur 17.1: Hvis $\delta(v,w) \leq \alpha$, vil α-oraklet kunne svare med sikkerhed og returnerer da et konservativt resultat $\delta_{\varepsilon,\alpha}(v,w) \leq \alpha + \varepsilon\alpha$.

Problemet er, at vi ikke kender den reelle korteste afstand, samt at vi indtil nu udelukkende finder afstande af begrænset størrelse. Vi vil i næste kapitel se på, hvorledes et antal α-orakler kombineres, således at vi vha. af forespørgsler til de kombinerede orakler altid kan finde en approksimeret afstand mellem to knuder, der konservativ og højst er $\delta(v,w)(1+\varepsilon)$ givet at en vej eksisterer.

Kapitel 18

Samlet afstandsorakel og forespørgsler

Det skal være muligt at finde vilkårligt store afstande vha. vores datastruktur. Den struktur, α-oraklet, der er beskrevet i kapitel 14-17, kan som nævnt i introduktionen (kapitel 13) udelukkende anvendes til at rapportere afstande op til $\alpha + \varepsilon\alpha$ for et givet α. Samtidig er usikkerheden på afstandene, der findes vha. denne datastruktur en additiv fejl på op til $\varepsilon\alpha$. Vi ønsker at kunne give et resultat, som maksimalt er en faktor $(1 + \varepsilon)$ for stort. Givet en orienteret, planar graf G med heltallige vægte på kanterne konstruerer vi derfor en samlet datastruktur, der består af et antal α-orakler, som kan anvendes til at finde et resultat, der opfylder præcisionkravet. Vi beskriver i dette kapitel hvorledes vi vælger fornuftige værdier af α og beskriver hvilke værdier af ε, der skal anvendes for at få det ønskede resultat på forespørgsler. Vi redegør for, hvorledes vi med udgangspunkt i G konstruerer grafer, der kan anvendes til konstruktion af et α-orakel for et givet α. Vi sørger for, at disse grafer er sammenhængende og udelukkende indeholder kanter med vægt op til α uden at introducere nye veje mellem G's knuder. Derefter beskriver vi hvilke dele den resulterende datastruktur er sammensat af. Vi beskriver herefter, hvorledes man i den samlede datastruktur laver forespørgler og vi viser at udførelsestiden for en forespørgsel er $\mathcal{O}(\log(\log(nW))\log(n) + \log(n)/\varepsilon)$. Sidst i kapitlet vil vi kort komme ind på et par af de mulige forbedringer, der kan laves til afstandsoraklet.

18.1 Værdier af α

Vi definerer først afstanden mellem to knuder i en graf samt kravene til den afstand, vi ønsker at få som resultat af en forespørgsel på den endelige, samlede datastruktur.

Definition 18.1 *Lad to knuder v og w samt en graf G være givet. Vi definerer $\delta(v, w)$ til at være den korteste afstand fra v til w i grafen G.*

Definition 18.2 *Lad to knuder v og w være givet. På en forespørgsel på afstanden fra v til w skal resultatet $\delta_r(v, w)$ overholde flg.*

$$\delta(v, w) \leq \delta_r(v, w) \leq (1 + \varepsilon)\delta(v, w) \tag{18.1}$$

Vores resultat skal altså være konservativt samt højst være en faktor $(1 + \varepsilon)$ fra den korteste afstand.

Når vi laver den samlede datastruktur, gentager vi den konstruktion af et α-orakel, som er beskrevet i kapitel 14-17. Vi laver to grupper af α-orakler, som har forskellig grad af præcision. Den mindst præcise gruppe anvendes til at afgøre i hvilket af de mere præcise α-orakler vi skal lede efter det endelige resultat. Vi erindrer, at præcisionen af resultatet skal være afhængigt af det ε, der er givet.

Vi konstruerer de mindst præcise α-orakler, som vi kalder *søgeorakler*. Disse konstrueres med $\alpha = 2^i$ for $i = 1, 2, \ldots, \lceil \log(nW) \rceil$, hvor W er vægten af grafens tungeste kant (og vi erindrer her, at vægtene er heltal). Da vægten af den tungeste er W, er $\delta(v, w) \leq nW$, hvis der findes en vej fra v til w. For søgeoraklerne sættes $\varepsilon' = 1/2$ – dette er altså uafhængigt af den værdi for ε, som er givet. Søgeoraklet med et givet α kaldes S_α.

Definition 18.3 *Lad to knuder v og w være givet. Vi definerer $\delta_\alpha(v, w)$ til at være resultatet af en forespørgsel på afstanden fra v til w i S_α*

Vi konstruerer også de mest præcise af oraklerne, som vi kalder *præcisionsorakler*. Disse konstrueres ligeledes med $\alpha = 2^i$ for $i = 1, 2, \ldots, \lceil \log(nW) \rceil$. For præcisionsoraklerne vælger vi $\varepsilon' = \varepsilon/4$. Præcisionsoraklet med et givet α kaldes P_α.

Definition 18.4 *Lad to knuder v og w være givet. Vi definerer $\hat{\delta}_\alpha(v, w)$ til at være resultatet af en forespørgsel på afstanden fra v til w i P_α.*

Med følgende lemma vil vi redegøre for, hvilket af præcisionsoraklerne, vi søger i, når vi fortager en forespørgsel på afstanden fra en knude v til en knude w. Dette præcisionsorakel findes ved at lave en binær søgning i søgeoraklerne. Vi antager i lemmaet af $\delta(v, w) \geq 1$. Vi beskriver i afsnit 18.3 hvorledes vi behandler nul-afstande.

Lemma 18.5 *Lad to knuder v og w i grafen G være givet og antag at $\delta(v, w) \geq 1$. Vælg det største mulige $\alpha' = 2^i$ således at $S_{\alpha'}$ rapporterer, at der ingen vej findes fra v til w. Da vil det for $\alpha = 4\alpha' = 2^{i+2}$ gælde at*

$$\alpha/4 \leq \delta(v, w) \leq \alpha$$

Præcisionsoraklet P_α vil da rapportere en afstand fra v til w hvorom flg. gælder

$$\delta(v, w) \leq \hat{\delta}_\alpha(v, w) \leq (1 + \varepsilon)\delta(v, w)$$

Dermed er $\hat{\delta}_\alpha(v, w)$ det resultat vi leder efter jf. præcisionskravet i formel 18.1, der er givet i definition 18.2.

Findes der ikke et orakel P_α, dvs. hvis α' har en værdi, der er større end $\lceil \log(nW) \rceil/4$, kan det sidste præcisionsorakel, $P_{\lceil \log(nW) \rceil}$ give os den ønskede afstand.

Eksisterer intet $\alpha' = 2^i$ således at S_α rapporterer, at der ingen vej findes fra v til w, kan præcisionsoraklet P_4 anvendes til at finde den rette afstand.

Bevis. Vi betragter søgeoraklet $S_{\alpha'}$, der rapporterer, at ingen vej eksisterer fra v til w. Da $S_{\alpha'}$ altid finder en afstand mellem to knuder, hvis den korteste afstand mellem disse er højst α' må det gælde at

$$\delta(v, w) > \alpha' \tag{18.2}$$

Betragt nu det næste søgeorakel $S_{2\alpha'}$ og vi erindrer at søgeoraklerne har $\varepsilon = 1/2$. Dette orakel vil da returnere en afstand fra v til w, der er højst $2\alpha' + 1/2\alpha' = 3\alpha'$. Da $S_{2\alpha'}$ er et konservativt orakel ved vi nu at

$$\delta(v, w) \leq 3\alpha' \tag{18.3}$$

Vi kan dog ikke garantere, at $\delta(v, w) \leq 2\alpha'$ og derfor vil præcisionsoraklet $P_{2\alpha'}$ ikke nødvendigvis give os en afstand mellem v og w.

Vi betragter derfor søgeoraklet S_α hvor $\alpha = 4\alpha'$. Fra udsagnene 18.2 og 18.3 ved vi flg. om afstanden fra v til w

$$\alpha/4 = \alpha' < \delta(v, w) \leq 3\alpha' < \alpha \tag{18.4}$$

Vi ved at når $\delta(v, w) \leq \alpha$ vil præcisionsoraklet P_α rapportere en afstand $\hat{\delta}_\alpha(v, w)$ fra v til w, som overholder

$$\delta(v, w) \leq \hat{\delta}_\alpha(v, w) \leq \delta(v, w) + \varepsilon\alpha/4$$

Vi har da opfyldt det ene krav for vores resultat – at resultatet er konservativt. Fra udsagnet 18.4 har vi en nedre grænse for $\delta(v, w)$ på $\alpha/4$, som giver os at

$$\hat{\delta}_\alpha(v, w) \leq \delta(v, w) + \varepsilon\alpha/4 \leq (1 + \varepsilon)\delta(v, w)$$

Hermed opfylder $\hat{\delta}_\alpha(v, w)$ begge kravene til et resultat, som vi vil rapportere.

Vi vælger altså det orakel der har et α, der er fire gange så stort, som α'. Vi skal derfor tage højde for tilfælde, hvor et sådan α ikke eksisterer (dvs. hvis α' svarer til det næstsidste orakel). Vi observerer først, at da $\delta(v, w) \leq nW$, vil det sidste orakel altid give os en vej fra v til w. Derfor vil det være det næstsidste søgeorakel, som siger, at der ingen vej er fra v til w, hvilket betyder at $\delta(v, w) \geq \lceil \log(nW) \rceil / 2$. Det sidste præcisionsorakel $P_{\lceil \log(nW) \rceil}$ vil da give os et resultat, der overholder vores krav.

Vi mangler nu tilfældet, hvor der ikke eksisterer et søgeorakel, der rapporterer, at der ingen vej er fra v til w. Alle oraklerne vil derfor rapportere en afstand fra v til w, specielt vil det første søgeorakel S_2 rapportere en afstand $\hat{\delta}_2(v, w)$, der højst er 3. Dermed må $\delta(v, w) \leq 3$. Vi kan derfor finde en afstand med passende præcision i præcisionsoraklet P_4, da $\delta(v, w) \geq 1$. □

Vi mangler at redegøre for hvad vi gør, hvis den korteste afstand mellem v og w er nul. Vi konstruerer et reachability-orakel for grafen G^0, som er G hvor vi kun inkluderer kanter med vægt nul. I dette kan vi altid finde ud af, om der findes en vej med vægt nul mellem to knuder. Vi bemærker, at G^0 kan blive usammenhængende når vi kun beholder kanter med vægt nul. Vi beskriver i afsnit 18.2 en strategi til at sammensætte en usammenhængende graf (se også figur 18.1) – samme strategi anvendes til at sammensætte G^0.

Hvis afstanden ikke er nul, kan vi nu finde det ønskede resultat ved at foretage en binær søgning i søgeoraklerne, som giver os det rette α'. Dette kan lade sige gøre, da vi ved, at der findes et skillepunkt mellem søgeorakler der *kan* svare på forespørgslen på v og w og søgeorakler, der *ikke kan* svare. Vi ser på det sidste søgeorakel $S_{\alpha'}$, der ikke kan give et svar – det kan det ikke fordi $\delta(v, w) > \alpha'$. Da vil alle søgeorakler med en α-værdi, der er mindre end α' heller ikke kunne svare på forespørgslen, da $\delta(v, w)$ ligeledes vil være større end disse α-værdier. Vi kan derfor lave en binær søgning efter det sidste orakel, der ikke kan svare på forespørgslen. Denne søgning giver os α'.

Derefter anvendes præcisionsoraklet P_α til at finde det resultat $\hat{\delta}_\alpha(v, w)$, der skal returneres. Lemma 18.5 siger, at $\hat{\delta}_\alpha(v, w)$ overholder vores præcisionskrav. Bemærk, at hvis alle orakler kan svare, vælger vi P_4, som vi redegjorde for i beviset for lemma 18.5. Ligeledes vælger vi $P_{\lceil \log(nW) \rceil}$, hvis dette er det eneste orakel, der kan svare.

18.2 Kanter med vægt over α

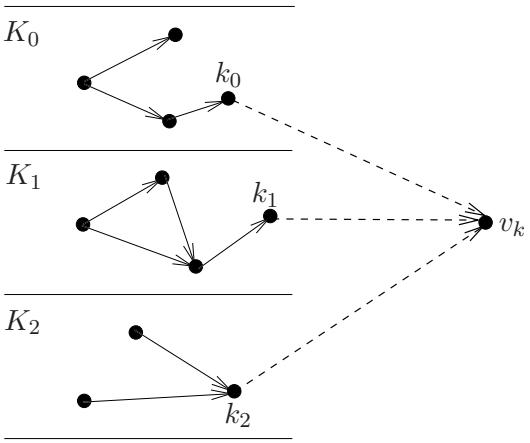
Når vi konstruerer et α-orakel antager vi, at den graf, der gives som argument er planar og sammenhængende samt at ingen kanter har vægt over α. I artiklen [Tho01] nævnes ingen løsning på dette problem, hvorfor vi selv har valgt en strategi, som vi vil beskrive i dette afsnit.

Når vi skal konstruere søgeorakler og præcisionsorakler må vi konstruere grafer der overholder ovennævnte krav (dvs. graferne skal være planare og sammenhængende samt ikke indeholde kanter med vægt over α). Når vi skal konstruere et orakel for et givet α, slettes alle kanter, der har vægt over α. Vi risikerer nu, at grafen er blevet delt i flere komponenter, Vi identificerer disse komponenter og vælger en knude som repræsentant for hver komponent. Vi har nu komponenterne K_0 til K_i og de tilsvarende repræsentanter k_0 til k_i. Vi samler nu grafen til en sammenhængende graf ved at tilføje en knude og nogle kanter. Hvis der er mere end en komponent, indsættes en ny knude v_k, som forbindes med alle komponenter på følgende måde. For $j = 0, \ldots, i$ indsættes en ny kant fra k_j til v_k med vægt 0. Når dette er gjort, har alle komponenter en kant til v_k og dermed er grafen sammenhængende. Vi observerer, at dette ikke introducerer nye stier mellem knuderne i den originale graf. Komponenterne er planare hver for sig og efter indsættelse af en ekstra knude og ekstra kanter er grafen ligeledes planar – repræsentanterne for de forskellige komponenter samt den nye knude og de nye kanter vil befinde sig i samme flade. Grafen har stadig klart $\mathcal{O}(n)$ knuder og kanter. Grafen er nu planar, sammenhængende og har kantvægte, der alle er højst α.

På figur 18.1 ses et eksempel hvor tre planare komponenter samles til en sammenhængende planar graf.

18.3 Forespørgsler i det samlede afstandsorakel

Når en samlet datastruktur er konstrueret, er det muligt at foretage forespørgsler i den. I dette afsnit vil vi kort opremse de dele, datastrukturen består af og der-



Figur 18.1: Komponenterne K_1, K_2 og K_3 samles således at grafen er sammenhængende.

efter redegøre for, hvorledes en forespørgsel foretages.

18.3.1 Det samlede afstandsorakel

Det samlede orakel indeholder, som beskrevet i afsnit 18.1, et antal α-orakler. Foruden disse anvendes også reachability-orakler som dem der er beskrevet i del I.

Det samlede afstandsorakel for grafen G består af fire dele:

Reachability-orakel. Vi konstruerer et reachability-orakel R for grafen G hvor vi ser bort fra kantvægte. Dermed kan vi hurtigt kan teste om en knude v kan nå en knude w.

Nul-orakel. Vi konstruerer endnu et reachability-orakel R_0, for en graf, der indeholder de kanter, som har vægt nul i G. Vha. dette orakel finder vi nul-afstande.

Søgeorakler. Vi konstruerer de søgeorakler, som er beskrevet i afsnit 18.1.

Præcisionsorakler. Vi konstruerer de præcisionsorakler, som er beskrevet i afsnit 18.1.

Strukturen af programmet, der udgør vores samlede afstandsorakel, er illustreret i bilag B.

Vi ser nu på ressourceforbruget for det samlede afstandsorakel. Konstruktionstiden for de to reachability-orakler er $\mathcal{O}(n \log(n))$, jf. afsnit 9.1.3, hvor vi konkluderer ressourceforbruget for et reachability-orakel. Konstruktionstiden for søgeoraklerne er $\mathcal{O}(n \log^3(n) \log(nW))$, jf. formel 17.3. Konstruktionstiden for præcisionsoraklerne er ligeledes ifølge formel 17.3 $\mathcal{O}(n \log^3(n) \log(nW)/\varepsilon)$. Den samlede konstruktionstid for afstandsoraklet er da

$$\mathcal{O}(n \log^3(n) \log(nW)/\varepsilon) \tag{18.5}$$

Pladsforbruget for de to reachability-orakler er $\mathcal{O}(n \log(n))$, jf. afsnit 9.1.3. Pladsforbruget for søgeoraklerne er ifølge formel 17.4 $\mathcal{O}(n \log(n) \log(nW))$. Pladsforbruget for præcisionsoraklerne er $\mathcal{O}(n \log(n) \log(nW))/\varepsilon$ ligeledes jf. formel 17.4. Vi konkluderer, at det samlede pladsforbrug for afstandsoraklet er

$$\mathcal{O}(n \log(n) \log(nW)/\varepsilon) \tag{18.6}$$

18.3.2 Forespørgsel

Forespørgslen anvender efter tur de datastrukturer, der er til rådighed, som vi netop har beskrevet i afsnit 18.3.1.

Vi starter med at finde ud af om v overhovedet kan nå w, hvilket vi bruger vores reachabilityorakel R til. Hvis v ikke når w i G, returneres ∞.

Hvis v kan nå w, anvender vi reachability-oraklet R_0 til at afgøre, om afstanden mellem v og w er nul. Hvis der findes en vej fra v til w i R_0, vil der eksistere en vej af længde nul fra v til w i G. I dette tilfælde vil vi da returnere nul.

Ellers anvender vi den strategi, der er beskrevet i beviset for lemma 18.5 til at finde den afstand, som skal rapporteres. Vi erindrer at dette indebærer en binær søgning i søgeoraklerne, som vil fortælle os i hvilket af præcisionsoraklerne den rette afstand kan findes.

De første to skridt, der laver forespørgsler i reachability-orakler, kan udføres i tid $\mathcal{O}(\log(n))$. Den binære søgning kræver $\mathcal{O}(\log(\log(nW)))$ forespørgsler til α-orakler, hvor vi har anvendt $\varepsilon = 1/2$. En af disse forespørgsler kan derfor udføres i tid $\mathcal{O}(\log(n))$, jf. formel 17.5. I alt anvendes tid $\mathcal{O}(\log(\log(nW)) \log(n))$ på den binære søgning. Den sidste forespørgsel i et præcisionsorakel tager tid $\mathcal{O}(\log(n)/\varepsilon)$, igen jf. formel 17.5. Den samlede forespørgselstid er da

$$\mathcal{O}(\log(\log(nW)) \log(n) + \log(n)/\varepsilon) \tag{18.7}$$

18.4 Implementation

Implementationen følger den teoretiske gennemgang, der er givet i dette kapitel. Vi konstruerer de beskrevne reachability-orakler og α-orakler, når et samlet orakel konstrueres. Forespørgsler foregår som det er beskrevet i afsnit 18.3.2.

18.4.1 Placering i programmet

Det samlede orakel konstrueres i klassen `distance/Oracle`. I denne klasse findes ligeledes en `query`-metode.

18.5 Mulige forbedringer

Det er som i reachability-oraklet muligt at lave forskellige forbedringer og udvidelser. Vi vil her kort nævne to af disse, men vil ikke gå i detaljer med dem. Begge svarer til forbedringer, der er gennemgået i forbindelse med reachability-oraklet.

Man kan igen anvende framereducering til at gøre forespørgselstiden konstant. Man skal i dette tilfælde dog være opmærksom på, at når man anvender forbindelser til at konstruere forbindelser, vil den additive fejl blive tilføjet flere gange, hvorfor man skal anvende et ε, der tager højde for dette.

Det er i afstandsoraklet muligt at distribuere den information, der skal anvendes til at svare på forespørgsler, som labels på knuderne på grafen G. Størrelsen af disse labels er i dette tilfælde $\mathcal{O}(\log(n)/\varepsilon)$, da listerne af forbindelser, der skal gemmes i knuderne har denne størrelse. Hvis den nødvendige information distribueres som labels på knuderne i G, vil man undgå at skulle gemme rekursionstræer mm., hvilket vil kunne afhjælpe det store overhead på pladsforbruget.

Kapitel 19

Eksperimentel evaluering

Vi vil i dette afsnit beskrive de test vi har foretaget på vores afstandsorakel. Disse test har haft følgende formål:

- At sandsynliggøre at de resultater afstandsoraklet returnerer ligger inden for det tilladte interval. Her tester vi op mod en kendt korrekt algoritme.
- At lave statistik over vores forbrug af ressourcer, dvs. at lave statistik over hvor lang tid det tager at konstruere oraklet, hvor lang tid en forespørgsel tager samt hvor meget oraklet fylder.
- At undersøge kvaliteten af de approksimative afstande.

Da vi testede vores reachability-orakel, brugte vi meget energi på at generere forskellige typer af testgrafer og se på hvorledes dette påvirkede forbruget af ressourcer. For at begrænse mængden af testdata har vi i dette kapitel valgt kun at kigge på få typer af testgrafer. Vi vil i afsnit 19.1 beskrive de typer af grafer vi har brugt i forbindelse med test i dette kapitel.

I afsnit 19.2 vil vi beskrive hvorledes vi har undersøgt om svarene på en forespørgsel ligger inden for det tilladte interval. Vi konkluderer, at vi ikke har kunnet producere svar, der ligger uden for det tilladte interval.

Vi har lavet test vi hvor måler plads- og tidsforbrug under opbygningen af afstandsoraklet. Vi kigger i afsnit 19.3 og 19.4 på hvorledes forbruget af ressourcer, som forventet, stiger når vægten på den tungeste kant øges. Vi forklarer endvidere, hvorfor vi ikke kan se en ændring i forbruget af ressourcer når ε varieres.

Vi beskrev i afsnit 10.3, at vores reachability-orakel fylder temmeligt meget og at dette bl.a. skyldes, at vi gemmer delgrafer ned gennem rekursionen. Når vi konstruerer søge- og præcisionsoraklerne gemmer vi på samme måde delgraferne ned gennem rekursionen. Det medfører naturligvis, at pladsforbruget for et afstandsorakel er meget højt. Vi kan ikke konstruere orakler med mere end 500 knuder, når vi forlanger, at datastrukturen skal kunne være i hukommelsen. I afsnit 19.3.1 har vi eksemplificeret pladsforbruget af et orakel, der er konstrueret på baggrund af en graf med 300 knuder. Dette orakel fylder 700 MB og vi viser hvorledes dette er muligt.

Vi har undersøgt, hvor lang tid det tager at udføre samtlige n^2 forespørgsler i grafer hvor vi varierer antallet af knuder og vægten på den tungeste kant og hvor vi opbygget afstandsorakler med varierende værdier af ε. Vi har sammenlignet dette med, hvor lang tid det tager for en $\mathcal{O}(n \log(n))$-algoritme at besvare de samme forespørgsler. Vi beskriver i afsnit 19.5 de test vi har lavet og vi konkluderer, at på små grafer er der kun er en mindre tidsbesparelsen ved at benytte vores afstandsorakel i forhold til f.eks. at bruge Dijkstras algoritme. Det bør dog bemærkes, at der en klar tendens til at hvis størrelsen af grafen øges vil der være en væsentlig tidsbesparelse. Vi konstaterer endnu engang, at hvis afstandsoraklet skal være praktisk anvendeligt, skal der optimeres kraftigt på pladsforbruget. Dette kan f.eks. gøres ved brug af labels som beskrevet i afsnit 18.5.

I afsnit 19.6 beskriver vi, hvorledes vi har testet kvaliteten af de svar vores orakel giver. Vi har testet med varierende værdier af ε og α og vi konkluderer, at den måde vi genererer vores testgrafer på medfører, at en stor procentdel af svarene er korrekte. Vi har endvidere kigget på, hvordan de resterende svar ligger i forhold til det tilladte interval og konkluderer, at der er en tendens til, at afvigelsen er mindre end det tilladte.

Alle test, hvor vi har målt tid, er foretaget på en 2,4 GHz Pentium 4 med en 512 kB cache og 512 MB hukommelse. Alle andre test er kørt på en 1 GHz Pentium III processor med en 256 KB cache og 1 GB hukommelse. Vi har kørt vores test på to forskellige maskiner, da vi ikke har haft mulighed for at køre tidstest på en maskine med mere end 512 MB hukommelse og vi har ønsket at køre de resterende test på en maskine med størst mulig hukommelse.

19.1 Testgrafer

De grafer vi bruger, når vi udfører test på vores afstandsorakel er genereret som beskrevet i afsnit 10.1. Vi beskrev i dette afsnit, at testgrafer er sammensat af et antal delgrafer og med to modsatrettede kanter mellem hvert par af delgrafer. Endelig laver vi en alternerende sti fra den første og sidste graf. For at begrænse mængden af test har vi i dette afsnit kun arbejdet med to typer af grafer.

Vi konstruerer grafer, der er bygget op af kun én delgraf. Vi har valgt at tilføje den alternerende sti for at sikre, at hovedparten af de genererede grafer indeholder flere end tre lag og at der dermed bliver konstrueret mere end en $(3, \alpha)$-lagsgraf for hver søge- og præcisionsorakel.

Da et afstandsorakel fylder temmeligt meget, har vi, når vi undersøger tidsforbruget for at opbygge et afstandoraklet, valgt at arbejde med grafer, hvor vi ikke tilføjer alternerende stier. Dette medfører, at det første lag i grafen kommer til at indeholde hovedparten af knuderne. Dermed kopieres en stor del af knuderne ikke flere gange når $(3, \alpha)$-lagsgraferne konstrueres. Vi har kun valgt at bruge denne type af testgrafer, når vi måler tidsforbruget for at opbygge oraklet. Dette skyldes, at den maskine vi kører disse test på kun har 512 MB hukommelse.

19.2 Test af korrektheden af resultaterne

Vi har testet de enkelte strukturer og algoritmer efterhånden som vi har implementeret dem. Vi har testet, at vi får konstrueret det korrekte antal α-orakler og at vi laver forespørgsler til det korrekte præcisionsorakel. I forbindelse med implementationen af α-orakler har vi genbrugt mange dele fra implementationen af reachability-orakler. De nye dele vi har implementeret er testet efterhånden som de er blevet implementeret. På denne måde er vi blevet overbevist om, at vi givet en separator finder de korrekte forbindelser til og fra den og at den rekursive funktion, der producerer ordnede forbindelser er korrekt. Endvidere har vi undersøgt, at vi får luget passende ud i de forbindelser vi finder.

For at teste korrektheden af de resultater, vores afstands- og α-orakel returnerer, har vi implementeret Dijkstras algoritme. Når vi tester om et orakel returnerer et korrekt svar, laver vi den samme forespørgsel til Dijkstra-algoritmen og vi undersøger efterfølgende om oraklets svar ligger inden for det tilladte område.

19.2.1 Test korrektheden af resultater fra α-oraklerne

Vi betegner den korteste afstand mellem to knuder v og w med $\delta(v, w)$ og bruger $\delta_\alpha(v, w)$ til at betegne den approksimative afstand som et α-orakel beregner. For knuder v og w, om hvilke der gælder, at den korteste afstand mellem dem højst er α skal der gælde at $\delta(v, w) \leq \delta_\alpha(v, w) \leq \delta(v, w) + \varepsilon\alpha$.

Vi har testet α-oraklerne ved for varierende værdier af α at generere grafer hvor vægten af den tungeste kant er α. Vi har systematisk testet med fem α'er i intervallet 5-400 og på grafer med op til 5.000 knuder. Vi har testet med tre værdier af ε i intervallet 0,1-0,9. Det ikke har været muligt for os at producere ukorrekte resultater.

19.2.2 Test korrektheden af resultater fra afstandsoraklerne

Vi lader igen $\delta_\alpha(v, w)$ betegne den korteste afstand mellem to knuder og $\delta_r(v, w)$ betegner den approksimerede afstand som vores afstandsorakel betegner. Vores orakel returnerer et svar inden for det tilladte interval hvis $\delta(v, w) \leq \delta_r(v, w) \leq \delta(v, w)(\varepsilon + 1)$.

Vi har testet afstandsoraklet med de samme værdier af ε og α som de test, der er beskrevet i afsnit 19.2.1. Det maksimale antal af knuder i vores testgrafer er 400. Vi konkluderer endnu en gang at det ikke har været muligt for os at producere ukorrekte resultater.

19.3 Pladsforbrug for afstandsoraklet

Vi har undersøgt hvor meget et afstandsorakel fylder ved at måle, hvor meget vores proces fylder umiddelbart før og efter konstruktionen af oraklet.

Vi har lavet grafer hvor vægten på kanterne er et tilfældigt tal mellem 0 og W og vi har ladet W have tre forskellige værdier. På figur 19.1 ses resultaterne af de test vi har udført med disse grafer, hvor vi haft $\varepsilon = 0,5$. Som det ses på figuren, har vi forsøgt at approksimere de beregnede data med en funktion tilhørende

$\mathcal{O}(n \log(n) \log(nW)/\varepsilon)$. Vi konkluderer, at det ud fra vores testdata ikke er muligt at undersøge om pladsforbruget overholder de ønskede asymptotiske grænser, da tendensen ikke er klar for små grafer. Vi har ikke kunnet øge antallet af knuder i graferne, da processen i så fald vil fylde mere end hukommelsen på den maskine vi har kørt vores test på.

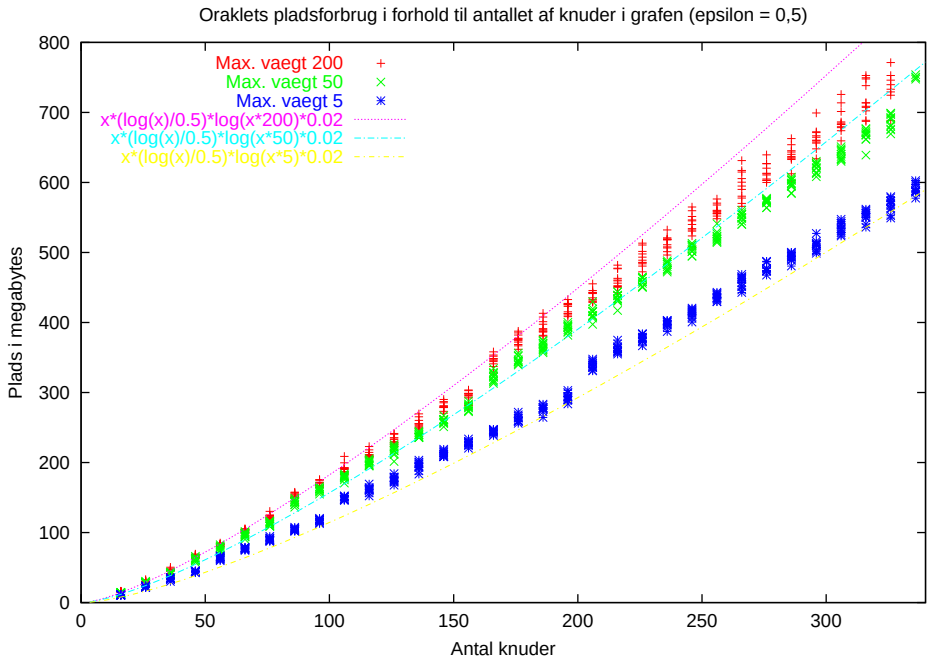
Vi kan på figur 19.1 se, at pladsforbruget som ventet stiger, når vi øger vægten af den tungeste kant. Det skyldes, at antallet af søge- og præcisionsorakler vi laver er $\lceil \log(nW) \rceil$.

Hvis vi på figur 19.1 ser på de resultater, hvor vægten af den højeste kant i grafen er 5 kan vi se, at der er et spring lige efter 100 og efter 200. Det passer med, at vi her laver et ekstra søge- og præcisionsorakel idet $2^9 = 5 * 102,4$ og $2^{10} = 5 * 204,8$. Vi kan ligeledes se, at der for de to andre typer af grafer, er et spring efter 150 knuder og dette passer med at hhv. $2^{13} = 50 * 163,84$ og $2^{15} = 200 * 163,84$.

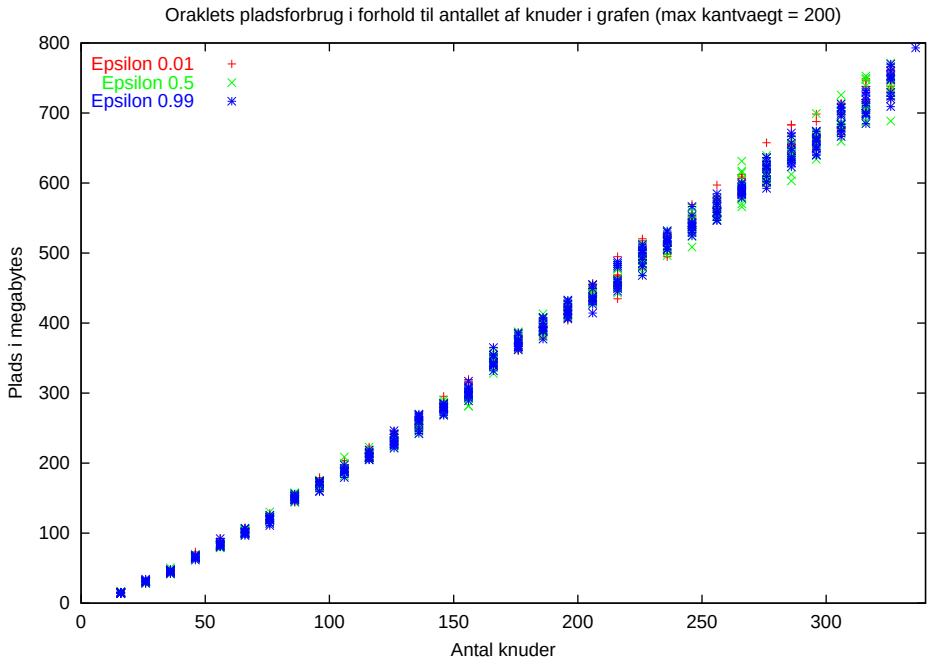
Vi har lavet test, hvor vi har varieret ε og hvor værdien af den tungeste kant i alle testgraferne er 200. Resultatet af disse test kan ses på figur 19.2. Når værdien af ε øges, forlanger vi en mindre præcision på de svar som oraklet skal returnere. Vi gemmer derfor et mindre antal forbindelser i afstandsoraklet. Det er ud fra de test vi har foretaget, ikke muligt at se dette. Vi mener, at årsagen til dette er, at forskellen i mængden af information, der gemmes, når værdien ε er lille, i forhold til når værdien er stor, er minimal i forhold til hvor meget et afstandsorakel fylder. Dette skyldes, at vi for hver α-orakel vi konstruerer, gemmer alle delgrafer i det rekursionstræ der opbygges. Hvis vi i stedet havde gemt informationen som labels på knuder, ville størrelsen af ε sandsynligvis have en meget større betydning for pladsforbruget.

19.3.1 Størrelsen af pladsforbruget

Vi har konstrueret en graf med godt 300 knuder og 900 kanter. Vægten på kanterne er et tilfældigt tal mellem 0 og 200 og vi har sikret os at mindst én kant har vægt 200. Vores testgrafer er bygget op af en enkelt delgraf. Vi har undersøgt hvorledes et afstandsorakel, der er konstrueret på baggrund af den beskrevne testgraf, kommer til at fylde knap 700.000 kB. Vi har målt hvor meget vi fylder efter konstruktionen af hvert α-orakel. Resultatet kan ses i den nedenstående tabel (alle tallene er i kB):



Figur 19.1: Forholdet mellem antallet af knuder i grafer, hvor vægten af den tungeste kant varierer og størrelsen af det konstruerede afstandsorakel med ε lig 0,5



Figur 19.2: Forholdet mellem antallet af knuder i grafer, og størrelsen af det konstruere afstandsorakel med varierende ε. Maksimal vægten på en kant er 200.

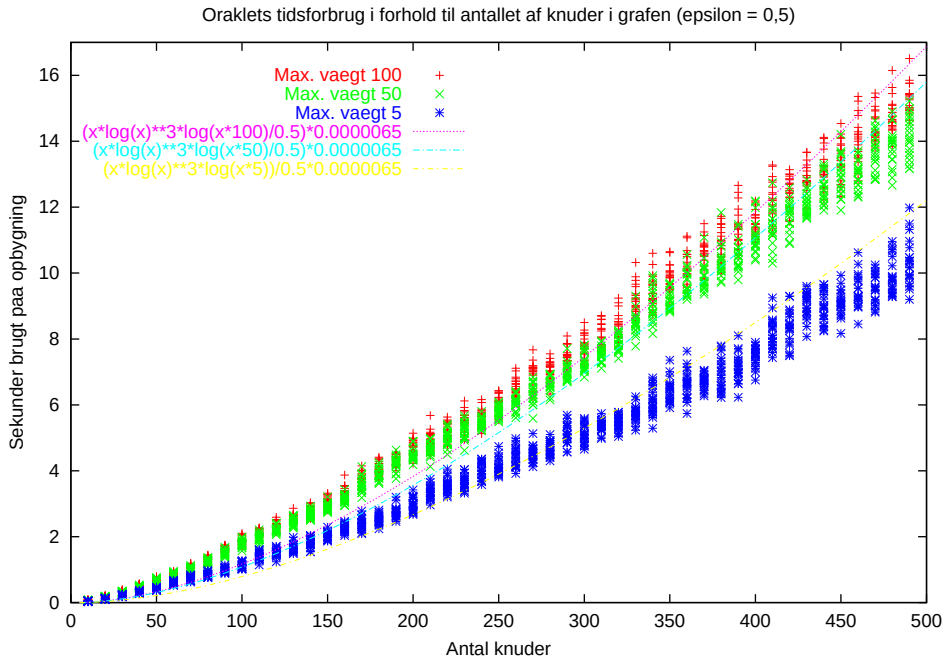
Opbygning af grafen	6776		
R og R_0-oraklet	15348		
0. søgeorakel	24824	0. præcisionsorakel	363392
1. søgeorakel	34112	1. præcisionsorakel	372644
2. søgeorakel	52188	2. præcisionsorakel	390720
3. søgeorakel	70384	3. præcisionsorakel	409148
4. søgeorakel	89280	4. præcisionsorakel	427984
5. søgeorakel	110632	5. præcisionsorakel	449492
6. søgeorakel	134752	6. præcisionsorakel	473940
7. søgeorakel	160552	7. præcisionsorakel	499844
8. søgeorakel	186048	8. præcisionsorakel	526084
9. søgeorakel	209860	9. præcisionsorakel	551580
10. søgeorakel	234008	10. præcisionsorakel	576892
11. søgeorakel	257572	11. præcisionsorakel	601520
12. søgeorakel	282000	12. præcisionsorakel	625580
13. søgeorakel	306180	13. præcisionsorakel	649844
14. søgeorakel	330120	14. præcisionsorakel	673468
15. søgeorakel	354024	15. præcisionsorakel	697996

Hver gang vi konstruerer et α-orakel tager vi en kopi af den oprindelige graf og når vi opbygger rekursionstræet for det pågældende orakel tager vi yderligere kopier af grafens knuder og kanter. Vi kan ud fra tabellen se, at hvert α-orakel fylder mellem 10.000 og 25.000 kB. Vi konkluderer, at der skal optimeres væsentligt på pladsforbruget hvis afstandsorakler skal være brugbare. En mulig måde at optimere dette på, er ved at distribuere informationen som labels på knuderne. Dette er kort beskrevet i afsnit 18.5.

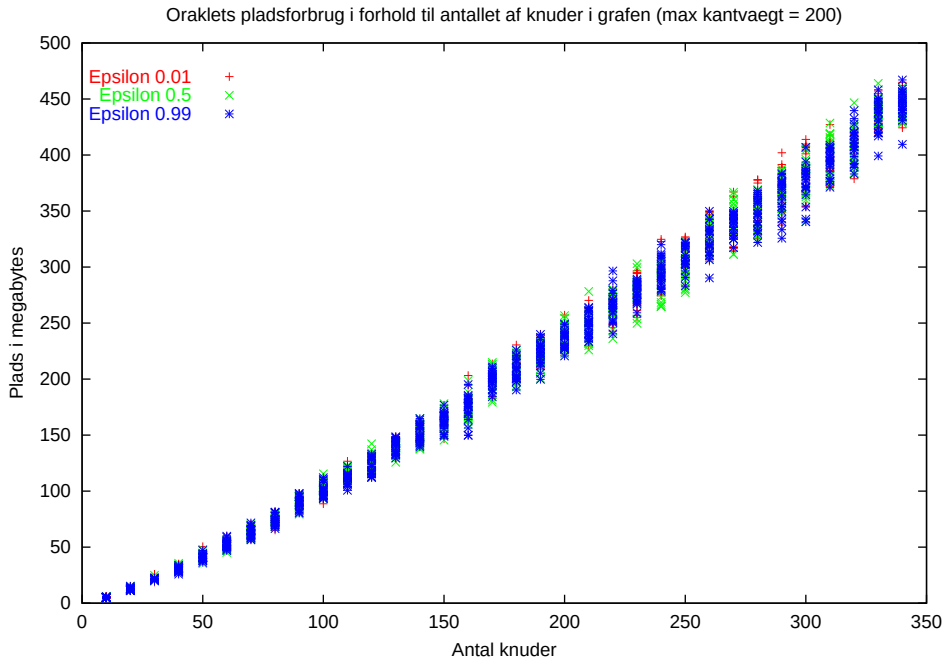
19.4 Tidsforbrug for opbygning af afstandsoraklet

Vi har undersøgt, hvor lang tid det tager at opbygge et afstandsorakel. Vi har konstrueret grafer hvor vi har varieret vægten af den tungeste kant og hvor antallet af knuder i graferne varierer. På figur 19.3 ses resultatet af vores test, hvor vægten af den tungeste kant i graferne varierer og hvor vi har $\varepsilon = 0,5$. Vi har forsøgt at approksimere vores data med en funktion i $\mathcal{O}(n \log^3(n) \log(nW)/\varepsilon)$, for at undersøge om vi overholder de forventede asymptotiske grænser for forbruget af tid. Som i afsnit 19.3 må vi konstatere, at vi ikke kan konkludere noget ud fra grafer med så få knuder og vi har som tidligere nævnt ikke kunne konstruere et afstandsorakel med udgangspunkt i en graf med flere knuder. Vi kan dog konkludere at forbruget af tid som forventet øges når vægten af den tungeste kant øges.

Vi har konstrueret grafer med varierende antal knuder, hvor vægten af den tungeste kant er 100. På baggrund af disse grafer har vi konstrueret afstandsorakler hvor vi har varieret værdien af ε. Resultatet af disse test ses på figur 19.4. Værdien af ε har indflydelse på udførselstiden for at konstruere et afstandsorakel. Når vi opbygger et α-orakel, bruger vi separatorer, når vi laver en dekomposition af grafen. I hvert rekursivt kald i dekompositionen finder vi forbindelser over separatorstier. I forbindelse med dette benytter vi en rekur-



Figur 19.3: Forholdet mellem antallet af knuder i grafer, hvor vægten af den tungeste kant varierer og tidsforbruget for at konstruere et afstandsorakel med ε lig 0,5.



Figur 19.4: Forholdet mellem antallet af knuder i grafer og tidsforbruget for at konstruere et afstandsorakel med varierende ε. Maksimal vægten på en kant er 100.

siv algoritme. Antallet af gange en knude er med i et rekursivt kald i denne algoritme, er afhængigt af ε, jf. afsnit 16.2.3. Det er ud fra de test vi har lavet, ikke muligt at se betydningen af værdien af ε. Vi konkluderer at dette skyldes, at tiden for at finde forbindelser har mindre indflydelse på den samlede udførelsestid.

19.5 Tidsforbrug for forespørgsler

Når man laver en forespørgsel i et afstandsorakel laver vi en binær søgning i de $\lceil \log(nW) \rceil$ søgeorakler efterfulgt af en forespørgsel i et præcisionsorakel. Det giver, som beskrevet i afsnit 18.3.2, at den samlede tid for at lave en forespørgsel bliver $\mathcal{O}(\log(\log(nW))\log(n) + \log(n)/\varepsilon)$.

Vi har undersøgt hvor lang tid det tager at lave forespørgsler i vores orakel sammenlignet med hvor lang tid det tager for en Dijkstra-algoritme. Vi har som i de tidligere test konstrueret grafer med varierende antal knuder og hvor vægten af den tungeste kant varierer. På figur 19.5 ses resultatet af vores test. Hvis vi kigger på resultaterne fra vores orakel, ser det ikke ud til, at værdien ε har betydning for svartiden. Det skyldes, at tiden for at lave den binære søgning i søgeoraklerne er dominerende. Dette udsagn understøttes af, at svartiden øges væsentligt når vægten på den tungeste kant øges.

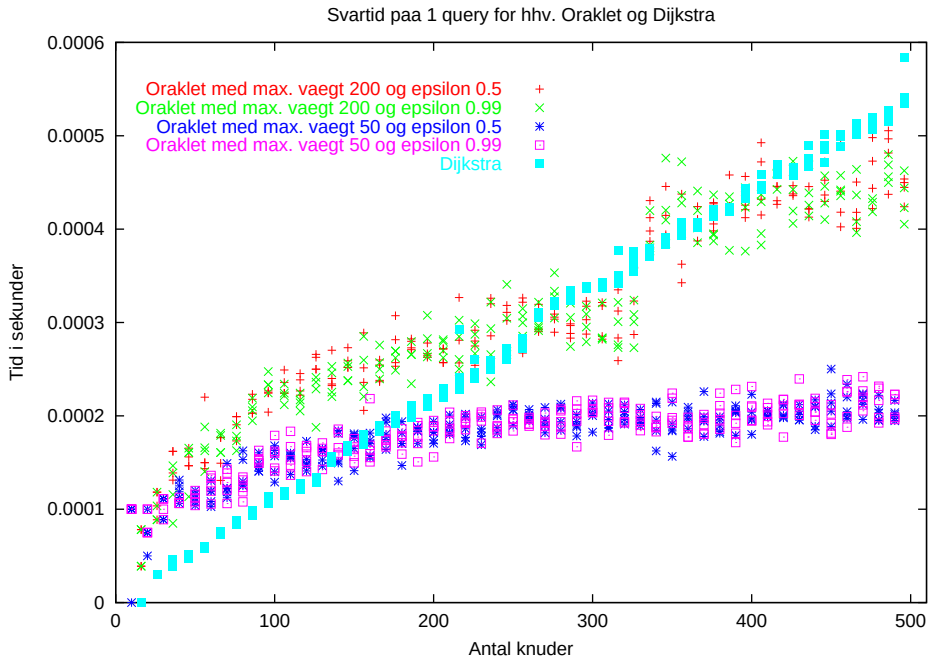
Hvis vi kigger på dataene fra grafer, hvor vægten af den tungeste kant er 200, kan vi se et spring mellem 300 og 400. Det passer med, at $2^{16} = 200 * 327,86$ hvilket betyder, at vi på dette tidspunkt laver et søge- og præcisionsorakel mere.

Da vi kiggede på tidsforbruget for forespørgsler til vores reachability-orakel, konstaterede vi, at vi skulle lave grafer med over 1000 knuder, for at kunne se en klar tendens. Vi har i denne del ikke haft mulighed for at kunne lave forespørgsler til grafer med så mange knuder og vi må konstatere, at vi ikke kan approksimere vores data med en funktion tilhørende $\mathcal{O}(\log(\log(nW))\log(n) + \log(n)/\varepsilon)$.

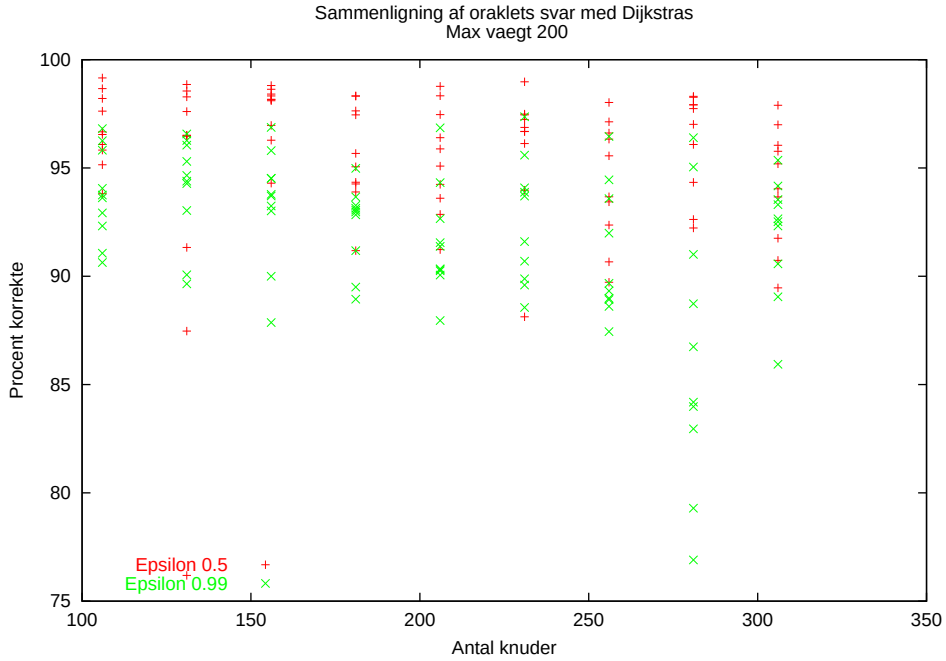
19.6 Kvaliteten af resultater af forespørgsler

Vi har kigget på kvaliteten af de svar som forespørgsler til vores afstandsorakel returnerer. Vi har lavet grafer hvor vi varierer vægten på den tungeste kant og antallet af knuder. Når vi konstruerer afstandsoraklet har vi varieret værdien af ε.

Vi har genereret grafer hvor den maksimale vægt på en kant er 200 og lavet afstandsorakler med ε lig 0,5 hhv. 0,99. For hver graf har vi foretaget samtlige n^2 forespørgsler og undersøgt hvor stor en procentdel af besvarelserne, der er lig den korteste afstand mellem de pågældende to knuder. På figur 19.6 ses resultatet af disse test. Vi konkluderer, at vi med den type grafer vi genererer, meget ofte returnerer den korteste afstand. Endvidere ser det ud til, at vi som forventet oftere returnerer den korteste afstand når værdien ε af reduceres. Hvis vi kigger på alle veje mellem to knuder, vil der være færre veje, der opfylder kravet om præcision når ε er lav sammenlignet med når ε er høj. Det medfører naturligvis, at den korteste vej oftere returneres når værdien af ε reduceres.



Figur 19.5: Gennemsnitstiden for en enkelt forespørgsel i afstandsorakler sammenlignet med Dijkstra. Varierende kantvægte og værdier af ε.



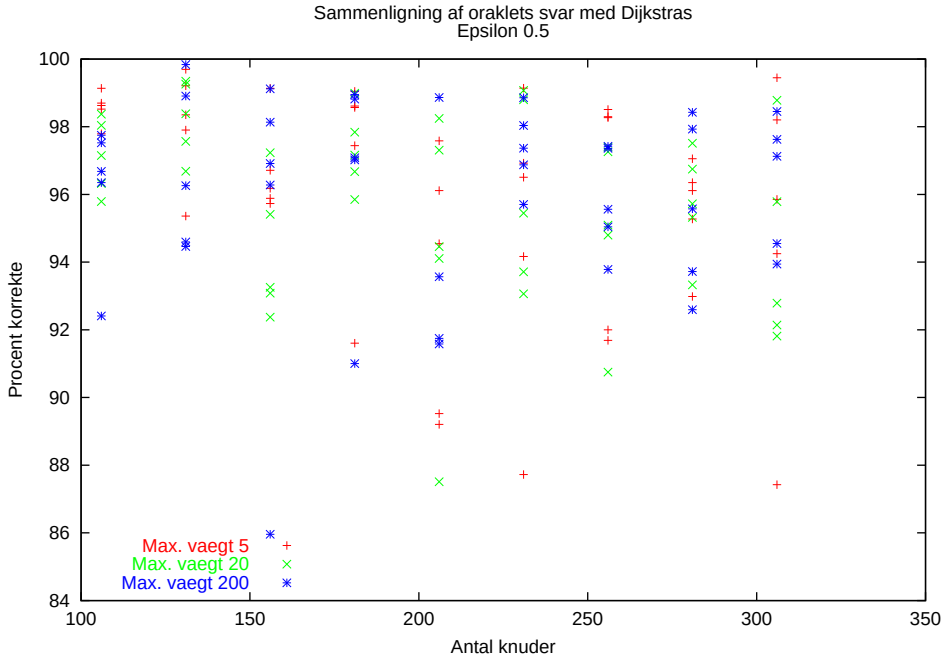
Figur 19.6: Forholdet mellem antallet af knuder i grafer og procentdelen af svar, produceret af afstandsorakler med varierende ε, der ikke afviger fra den korteste afstand.

Vi har gentaget de samme test som netop beskrevet, men hvor vi har holdt værdien af ε fast og varieret vægten på den tungeste kant, se figur 19.7. Der ses ikke nogen klar tendens i forholdet mellem hvor mange procent af svarene, der er lig den korteste afstand og vægten på den tungeste kant.

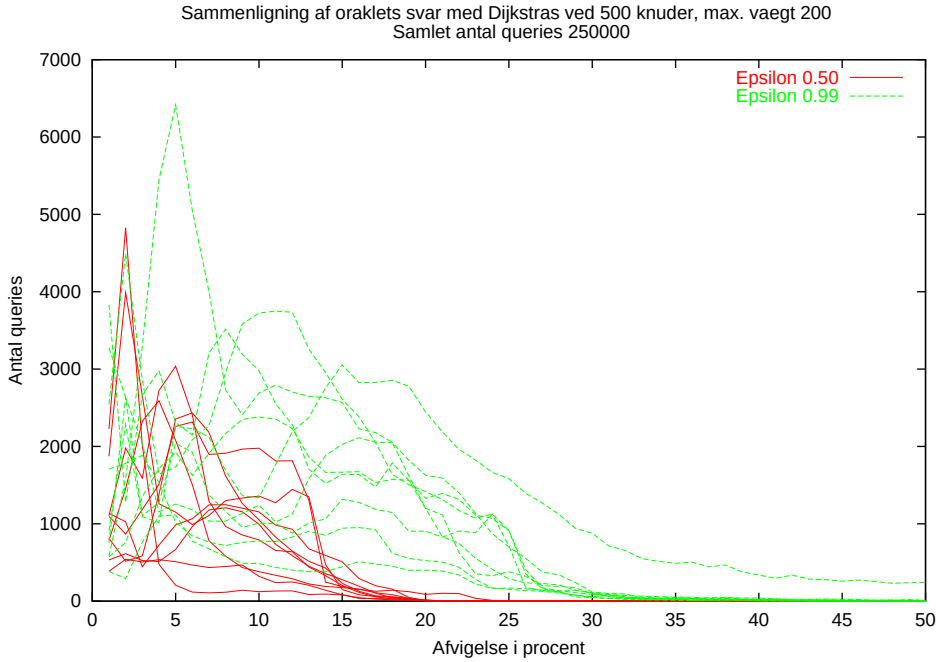
Efter at have kigget på hvor stor en procentdel af samtlige forespørgler, der returnerer den korteste afstand, er det naturligt at kigge på, hvor meget de approksimative afstande afviger fra den korteste afstand.

Vi har genereret grafer med 500 knuder, hvor vægten af den tungeste kant er 200 og konstrueret afstandsorakler hvor ε har været 0.5 hhv. $0,99$. Vi har lavet samtlige forespørgler og kigget på hvor stor den procentvise afvigelse af resultaterne har været. På figur 19.8 ses resultatet. Vi kan konstatere, at afvigelserne som ventet øges når værdien af ε øges. Dernæst kan vi konstatere at størsteparten af afvigelserne er noget mindre end det tilladte.

Vi konkluderer, at de resultater vores afstandsorakel har returneret alle opfylder kravet om præcision. Endvidere gælder, at en stor del af svarene er korrekte og om de resterende gælder, at afvigelsen er en del mindre end det lovede.



Figur 19.7: Forholdet mellem antallet af knuder i grafer hvor vægten af den tungeste kant varierer og procentdelen af svar, der ikke afviger fra den korteste afstand.



Figur 19.8: Den procentvise afvigelse af de approksimative queries i grafer med 500 knuder og den maksimale kantvægt er 200

Kapitel 20

Konklusion

Vi har i denne del af specialet givet en grundig teoretisk gennemgang af de algoritmer og datastrukturer, der indgår i konstruktionen af et afstandsorakel og dermed de α-orakler, der er hovedbestanddelen i et afstandsorakel. Afstandsoraklet er beskrevet i artiklen *Compact Oracles for Reachability and Approximate Distances in Planar Digraphs* [Tho01] af Mikkel Thorup.

Vi har i denne del i høj grad motiveret de mange konstruktioner, der indføres i gennemgangen af de forbindelser, der anvendes i et α-orakel. Vi mener at dette, sammen med en række illustrationer, er med til at øge forståelsen for indførelsen af konstruktionerne. Vi har ligeledes tilføjet detaljer til de beviser, der anvendes til at vise korrektheden af konstruktionerne for at øge forståelsen af disse beviser.

Vi har gennem motivationer, illustrationer og opsummeringer skabt et overblik over de konstruktioner, der tilsammen danner afstandsoraklet.

Vi har i denne del bidraget med overvejelser om ændringer af grafer, således at α-orakler kan konstrueres for alle værdier af α. Vi har ligeledes bidraget med et bevis for effektiviteten af konstruktionen af forbindelser, da denne afviger fra den oprindelige konstruktion i artiklen.

Vi har lavet en implementation af afstandsoraklet og vi har i denne del beskrevet denne implementation. I vores beskrivelse af implementionen kommer vi ind på de mange overvejelser og detaljer, der har været forbundet med den omfattende implementationen af afstandsoraklet.

Vi har igen i løbet af implementationsfasen set på mellemresultater og på de opbyggede datastrukturer for at sandsynliggøre at disse er fornuftige. Vi har ligeledes lavet en del test på det færdige afstandsorakel, hvor vi har set på resultater og ressourceforbrug. Vi kan konkludere at de resultater, vi har produceret i vores test alle overholder vores krav til præcision af resultater. Vi kan ligeledes konkludere, at selvom man tillader en fejl på de approksimerede resultater, er mange af de producerede resultater korekte. De resterende resultater holder sig alle indenfor den tilladte afvigelser fra den korteste afstand. Afvigelserne ser for de fleste resultaters vekommende desuden ud til at være noget mindre end det tilladte.

Det har været svært at se klare tendenser, da vi ikke har kunnet teste på grafer, der indeholder mange knuder. Grunden til dette er, at vi har et meget stort overhead på forbruget af plads. Dette betyder ligeledes, at tendenserne for

det asymptotiske tids- og pladsforbrug ikke er klare. For at afstandsoraklet vil kunne anvendes i praksis, er det derfor nødvendigt at reducere pladsforbruget væsentligt. Denne reducering kan f.eks. opnås ved brug af labels.

Vi kan konkludere, at vi har opnået at præsentere teorien omkring et afstandsorakel, så denne er tilgængelig for læseren. Vi har selv tilføjet mindre ting men hovedbidraget er illustrationer, motivationer og tydeliggørelser samt udfyldelse af ikke-trivielle detaljer, der i artiklen er overladt til læseren. Vi har ligeledes vist, at implementation af et afstandsorakel er mulig, omend ændringer er nødvendige for at sikre praktisk anvendelighed.

Litteratur

[AGKR02] Stephen Alstrup, Cyril Gavoille, Haim Kaplan, and Theis Rauhe. Nearest Common Ancestors: A Survey and a new Distributed Algorithm. In *14th ACM Symposium on Parallel Algorithms and Architectures*, 2002.

[Alg] Algorithmic Solutions. *The LEDA User Manual*.

[HT84] D. Harel and R. E. Tarjan. Fast algorithms for finding nearest common ancestor. *SIAM Journal on Computing*, 13:338–355, 1984.

[HU] T. Hagerup and C. Uhrig. Triangulating a Planar Map without Introducing multiple Ares. Upubliceret.

[led] `http://www.algorithmic-solutions.info/leda_guide/simple_data_types/memmgm%t.html`. LEDA-manualen på LEDAs egen hjemmeside.

[LT79] Richard J. Lipton and Robert Endre Tarjan. A separator theorem for planar graphs. *SIAM Journal on Applied Mathematics*, 36(2):177–189, April 1979.

[MN99] K. Mehlhorn and S. Näher. *LEDA*. Cambridge University Press, 1999.

[pur] `http://www-306.ibm.com/software/awdtools/purifyplus`.

[Thi98] Sven Thiel. The leda memory manager. Upubliceret, aug 1998.

[Tho01] Mikkel Thorup. Compact Oracles for Reachability and Approximate Distances in Planar Digraphs. In *42nd IEEE Annual Symposium on Foundations of Computer Science*, 2001.

Bilag A

Opbygning af reachability-programmet

I dette bilag illustrerer vi sammenhængen mellem de klasser og funktionsfiler, der findes i vores program, der realiserer et reachability-orakel.

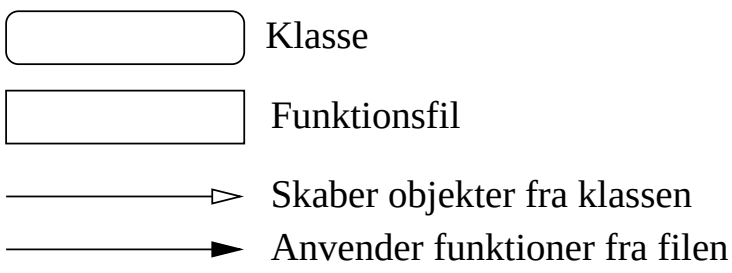
I programmet findes flg. klasser:

- ReachabilityOracle
- OriginalGraph
- LayeredGraph
- RecursionTree
- CycleElement

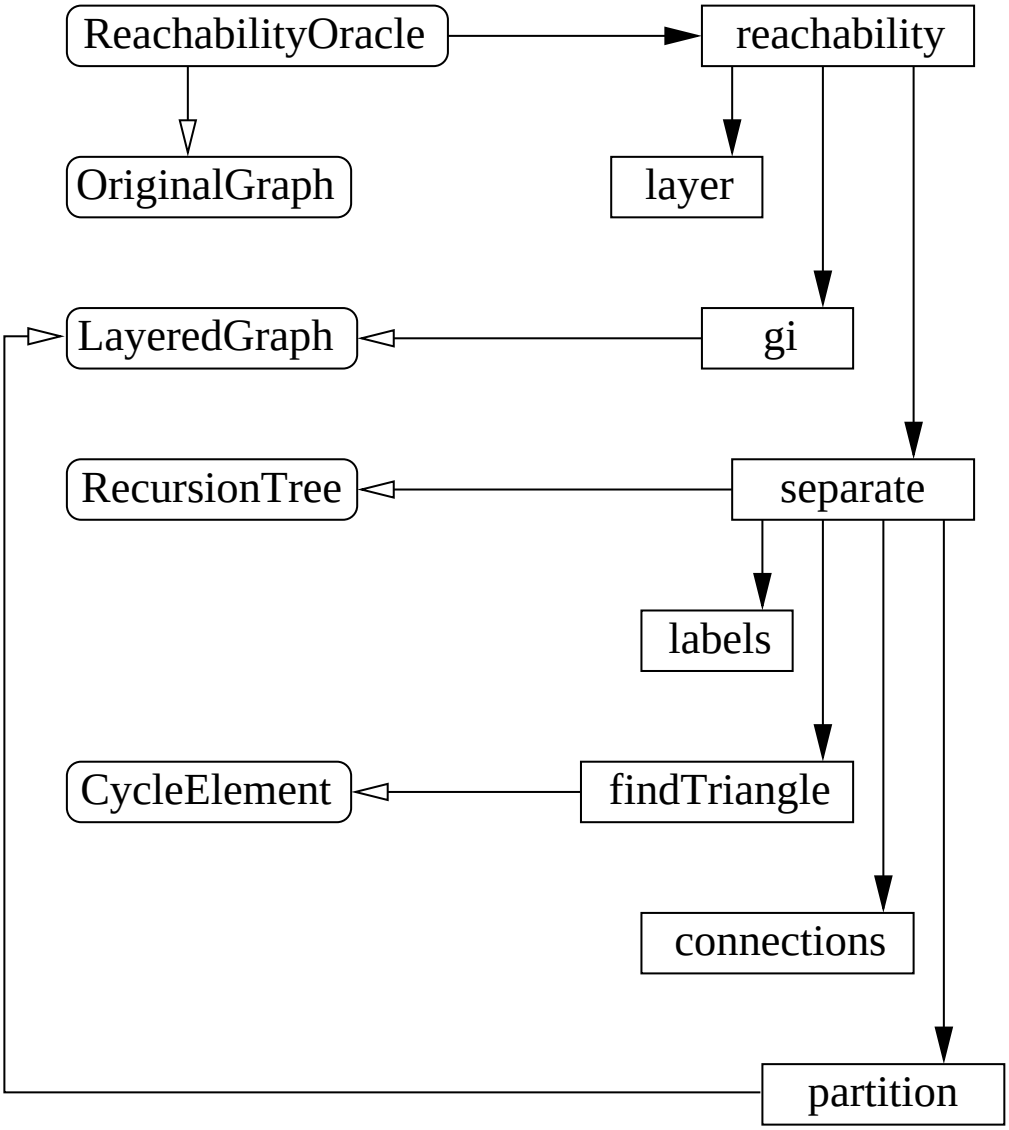
Vi har flg. filer, der indeholder funktioner:

- reachability
- layer
- gi
- separate
- labels
- findTriangle
- connections
- partition

Sammenhængen mellem klasser og filer er illustreret på figur A.2. En forklaring til de forskellige enheder på illustrationen kan ses på figur A.1.



Figur A.1: Tegnforklaring.



Figur A.2: Diagram over reachability-programmet.

Bilag B

Opbygning af distance-programmet

I dette bilag illustrerer vi sammenhængen mellem de klasser og funktionsfiler, der findes i vores program, der realiserer et afstandsorakel.

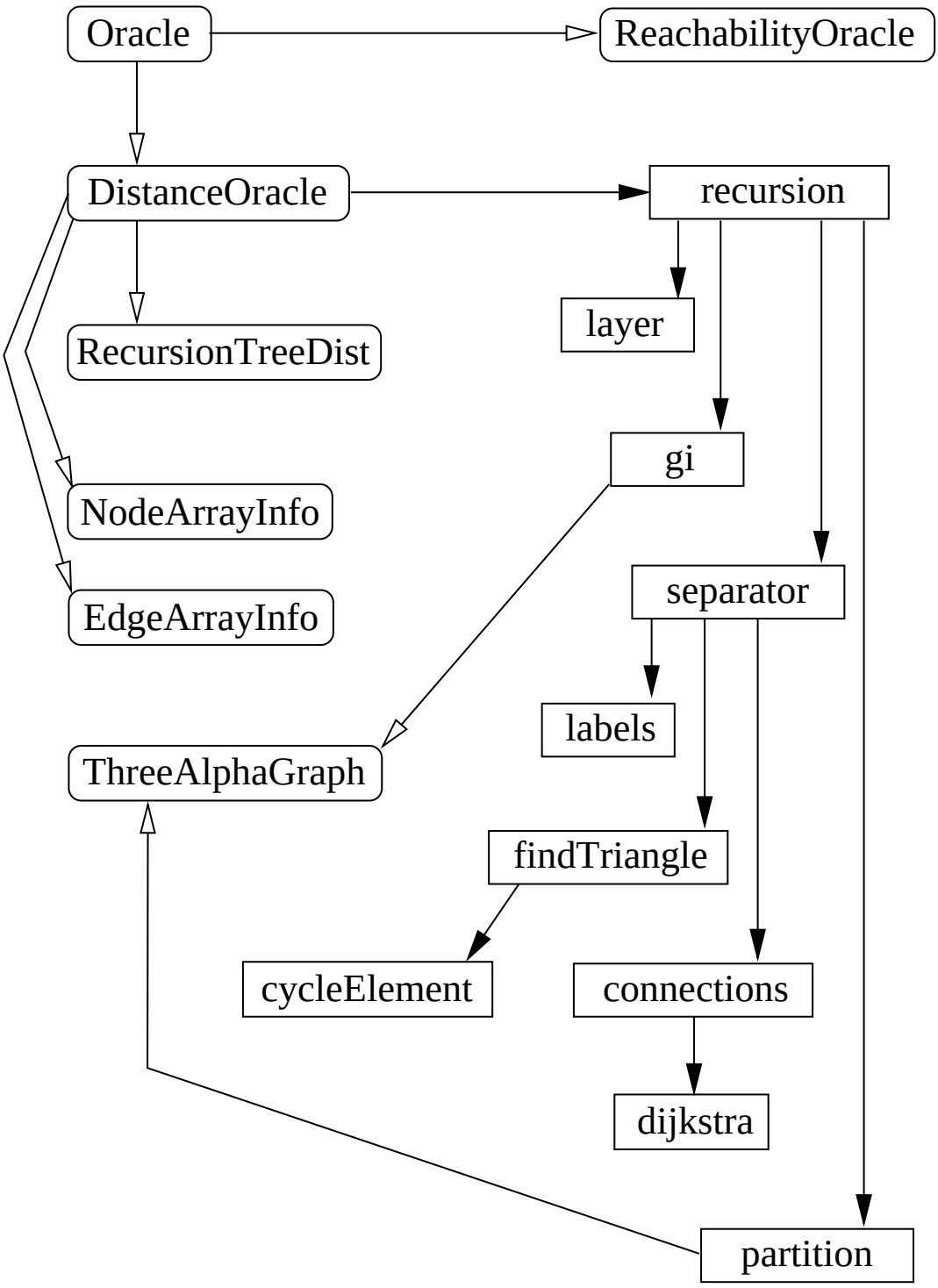
I programmet findes flg. klasser:

- Oracle
- DistanceOracle
- ThreeAlphaGraph
- RecursionTreeDist
- NodeArrayInfo
- EdgeArrayInfo

I afstandsoraklet har vi valgt at lave funktioner som medlemsfunktioner i klasserne. Vi har flg. filer, der indeholder funktioner – i parantes ses den/de klasser, som funktionerne er medlemsfunktioner i:

- layer (DistanceOracle)
- gi (DistanceOracle)
- recursion (DistanceOracle og ThreeAlphaGraph)
- separator (ThreeAlphaGraph)
- labels (ThreeAlphaGraph)
- findTriangle (ThreeAlphaGraph)
- cycleElement (ThreeAlphaGraph)
- connections (ThreeAlphaGraph)
- partition (ThreeAlphaGraph)
- dijkstra (ThreeAlphaGraph)

Sammenhængen mellem klasser og filer er illustreret på figur B.1. En forklaring til de forskellige enheder på illustrationen kan ses på figur A.1.



Figur B.1: Diagram over programmet, der finder approksimerede afstande.

Bilag C

Brugervejledning

Den medfølgende cd indeholder kildekoden af vores program. I biblioteket `reachabilty` findes den del, der implementerer reachability-oraklet og i `distance` findes delen, der implementerer afstandsoraklet. Filerne i bibliotekerne er struktureret som vist på bilag A og B.

Vi har forsøgt at lave en statisk kompileret udgave af programmet, men dette har ikke været muligt, da LEDA ikke er statisk kompileret. For at køre vores program kræves det derfor, at man har LEDA installeret. Vi har benyttet LEDA version 4.4, der er prekompileret til en i386 processor med redhat linux 7.0 og g++ 2.96.

På cd'en findes en kompileret version af programmet på en Pentium III med redhat linux 7.3 og g++ 2.96. På cd'en findes eksekverbare testfiler.

C.1 Reachability-oraklet

Vi har lavet et testprogram, der kan findes i `reachability/orakeltest.cpp`. Programmet kan generere en graf med et ønsket antal knuder og kanter samt delgrafer (jf. afsnit 10.1). På baggrund af grafen opbygges et reachability-orakel. Programmet tager argumenter fra kommandolinjen og køres ved at skrive følgende:

```
orakeltest [antal knuder] [antal kanter] [antal delgrafer]
```

Argumenter skal være heltallige og alle argumenter skal angives i det nævnte rækkefølge. Hvis antallet af kanter angives til at være nul, genererer vi en maksimal planar graf. Vi laver en testgraf uden de ekstra alternerende stier, hvis antallet af delgrafer angives til at være nul.

Testprogrammet genererer en graf, der dumpes i en fil ved navn `graf.txt`. Der konstrueres et reachability-orakel og for hvert par af knuder i grafen laver vi en forespørgsel og resultatet skrives i filen `result.txt`.

I biblioteket `reachability` findes en make-fil. Ved at udføre kommandoen `make` kompileres programmet og den eksekverbare fil `orakeltest` genereres.

C.2 Afstandsoraklet

I biblioteket `distance` findes to underbiblioteker; `reachability` og `distance`. I `distance` findes testfilen `orakeltest.cpp`, der kan generere testgrafer, konstruere et afstandsorakel og lave forespørgsler. Testprogrammet køres ved at skrive følgende:

```
orakeltest [antal knuder] [antal kanter] [antal delgrafer]
           [vægten på den tungeste kant] [epsilon]
```

Alle argumenter skal angives og med undtagelse af ε skal de alle være heltallige. Vægten på testgrafens kanter er et tilfældigt tal mellem 0 og den angivne tungeste kant.

Testgrafen dumpes i `graf.txt` og resultaterne i `result.txt`.

Begge underbibliotekerne i `distance` indeholder en make-fil. For at kompilere programmet skal kommandoen `make` udføres først i `distance/reachability` og efterfølgende i `distance/distance`. Herved genereres den eksekverbare testfil.