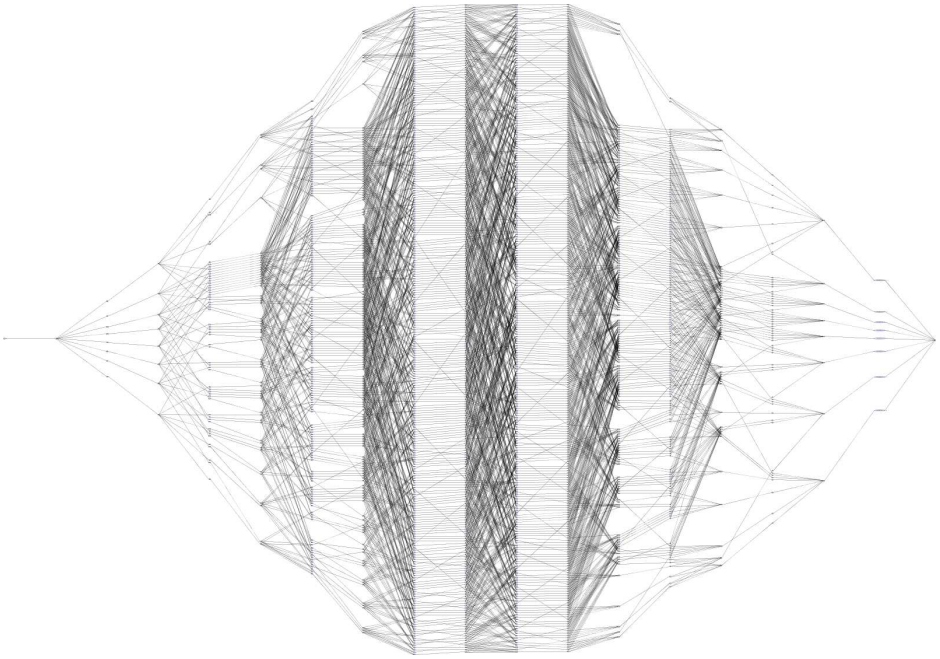
Algoritmer til løsning af ubegrænsede influensdiagrammer på begrænset plads



Speciale af
Kristian Ahlmann-Ohlsen og Ole Pedersen
12. juni 2006, Institut for datalogi, Aalborg Universitet

AALBORG UNIVERSITET ▪ DATALOGISK INSTITUT

Fredrik Bajers Vej 7 ▪ DK-9220 Aalborg Øst ▪ Telefon 96 35 80 80

Titel:	Algoritmer til løsning af ubegrænsede influensdiagrammer på begrænset plads
Tema:	Maskinintelligens
Projekt periode:	Dat6, 2006
Projekt gruppe:	E1-117
Projekt afsluttet:	12. juni 2006
Antal kopier:	7
Rapport sider:	146
Sider i alt:	178

Gruppemedlemmer:

Kristian Ahlmann-Ohlsen

Ole Pedersen

Vejleder:

Thomas D. Nielsen

Sammendrag:

Ubegrænsede influensdiagrammer (UIDer) er blevet introduceret som en generalisering af influensdiagrammer, hvor kravet fra influensdiagrammer, om at alle beslutninger skal være ordnet i forhold til hinanden, er blevet ophævet. Således er UIDer velegnede til at repræsentere problemer, hvor ordningen af beslutningerne ikke eller kun delvist er fastlagt, dvs. ordningsasymmetriske beslutningsproblemer.
En algoritme, der kan tilbyde en afvejning mellem tid og plads ved løsning af UIDer, opstilles og aftestes empirisk. Den opstillede algoritme er baseret på konditionering og opnår afvejningen mellem tid og plads ved indførelse af cache. Sammen med algoritmen opstilles øvre grænser for algoritmens tids- og pladskompleksitet i grænserne, hvor der er plads til fuld cache, og hvor der ikke er plads til cache. I grænsen, hvor der ikke er plads til cache, vokser algoritmens pladsforbrug (ud over en given basisallokering) lineært med antallet af knuder i UIDet. Der opstilles ligeledes en formel til estimering af kørselstiden ved en given mængde plads.

Forord

Denne rapport er skrevet af DAT6 projektgruppen "E1-117" på Datalogisk institut på Aalborg universitet i foråret 2006. Rapporten behandler udviklingen af anyspacealgoritmer til løsning af ubegrænsede influensdiagrammer.

Vi vil gerne takke Zwi Altman, Finn Verner Jensen, Bente Jørgensen, Rana Khanafer, Manuel Luque, Lars Moltsen Nielsen og Marta Vomlelová for deres hjælp i forbindelse med arbejdet bag denne rapport.

Forsideillustrationen viser en løsningsgraf, der kan bruges til løsning af ubegrænsede influensdiagrammer. Løsningsgrafen repræsenterer et beslutningsproblem med 8 beslutninger, der er opstillet således, at løsningsgrafen bliver størst mulig. Den viste løsningsgraf har 2.051 knuder og 4.624 kanter.

I denne rapport udvikles et system til løsning af ubegrænsede influensdiagrammer. Dette system kan hentes fra `www.cs.aau.dk/~ole/dat6/`.

Kristian Ahlmann-Ohlsen

Ole Pedersen

Indhold

Del I

Analyse

Introduktion

Indenfor beslutningsstøttesystemer for beslutningstagning under usikkerhed med én beslutningstager er der for nyligt introduceret et sprog, der tilbyder en intuitiv modellering af beslutningsproblemer af en type kaldet ordningsasymmetriske. Ved ordningsasymmetriske beslutningsproblemer forstås beslutningsproblemer, hvor en mængde beslutninger skal træffes i en rækkefølge, der ikke nødvendigvis er fastlagt. Det nye sprog til repræsentation af disse beslutningsproblemer kaldes ubegrænsede influensdiagrammer [Jensen and Vomlelová, 2002, Jensen et al., 2006b].

Ved løsning af modeller udtrykt i dette sprog forstås bestemmelse af en strategi for, hvilke valg en bruger skal træffe, for at handle optimalt, når han står overfor et beslutningsscenarie, der er modelleret i sproget.

Der er tidligere opstillet to algoritmer til bestemmelse af disse strategier. Den første algoritme kunne bestemme den optimale strategi, hvis der var tid og plads nok til rådighed på det system, der skulle afvikle algoritmen. Den anden algoritme udviklede vi i [Ahlmann-Ohlsen and Pedersen, 2005], som en anytimealgoritme, hvor vi muliggjorde bestemmelse af en suboptimal strategi hurtigt, med en efterfølgende forfinelse af den fundne strategi indtil den optimale strategi var fastlagt. Med denne algoritme blev det muligt at handle ud fra modellen, selvom tidsressourcerne ikke tillod en bestemmelse af den optimale strategi De eksisterende algoritmer lider dog under nogle høje pladskrav, hvorfor den tilgængelige pladsressource ofte vil være en begrænsende faktor for hvilke modeller, der kan løses.

Vi vil i dette arbejde udvikle en algoritme, der kan løse beslutningsproblemer, repræsenteret med ubegrænsede influensdiagrammer under mindre pladskrav end dem, der gives for de eksisterende løsningsalgoritmer. Efter en

analyse af forskellige tilgangsvinkler til udvikling af en løsningsalgoritme til løsning af ubegrænsede influensdiagrammer på begrænset plads, beskriver vi udviklingen af en anyspace-løsningsalgoritme. Denne algoritme tilbyder en trinløs afvejning mellem tid og plads uden at gå på kompromis med kvaliteten af den opnåede løsning. Sammen med algoritmen udleder vi et udtryk til estimering af kørselstiden for algoritmen under en given mængde plads, og vi opstiller teoretiske øvre grænser for algoritmens kørselstid i ekstremerne med vilkårligt meget hukommelse til rådighed og meget lidt hukommelse til rådighed.

I denne rapport starter vi således med at analysere problemområdet i kapitel 1 ved at analysere beslutningsproblemer generelt, samt nogle gængse sprog til repræsentation af beslutningsproblemer. I kapitel 2 præsenteres sproget ubegrænsede influensdiagrammer som en mulig løsning til nogle repræsentationsproblemer ved de første sprog.

I rapportens anden del, designet, beskriver vi (i kapitel 3) en række overvejelser, som vi har gjort os i forbindelse med valget af løsningsmetode. I kapitel 4 gennemgår vi algoritmen Recursive Conditioning, der danner grundlaget for den løsningsmetode, som vi anvender i vores løsningsalgoritme, der præsenteres i kapitel 5.

I kapitel 6 beskriver vi en række strategier for, hvordan vores løsningsalgoritmes kørselstid kan forbedres ved udnyttelse af cache, og i kapitel 7 sammenligner vi teoretisk vores løsningsalgoritme med den eksisterende algoritme mht. algoritmernes tids- og pladskompleksiteter.

Tredje del af rapporten indeholder en empirisk aftestning af vores løsningsalgoritme (kapitel 8). Slutteligt afrundes rapporten med en gennemgang af hvordan den præsenterede algoritme kan forbedres (kapitel 9) efterfulgt af en konklusion i kapitel 10.

I de to første kapitler indgår dele af [Ahlmann-Ohlsen and Pedersen, 2005].

Kapitel 1

Beslutningsproblemer

Dette kapitel introducerer baggrunden for ubegrænsede influensdiagrammer i form af influensdiagrammer samt beslutningsproblemer generelt.

Beslutningsproblemer omhandler det, at foretage valg under usikkerhed [Neumann and Morgenstein, 1953] og er traditionelt blevet repræsenteret ved beslutningstræer [Raiffa and Schlaifer, 1961]; et udtryksfuldt, men hurtigt voksende repræsentationssprog.

Senere introduceredes influensdiagrammer [Howard and Matheson, 1984] som en, i nogle situationer, mere kompakt repræsentation af beslutningsproblemer. Disse har vundet indpas og anses i vid udstrækning for at være en udvidelse af bayesianske net [Pearl, 1986, Verma, 1987] fremfor et særtilfælde af beslutningstræer [Madsen and Jensen, 1999a].

I dette kapitel vil vi præsentere beslutningsproblemer såvel som beslutningstræer, som referenceramme til introduktionen af influensdiagrammer. Sidstnævnte vil blive formelt defineret, og en algoritme til løsning af influensdiagrammer vil blive præsenteret. Afrundingsvis gives en kort diskussion af nogle ulemper ved influensdiagrammer, hvilket skal fungere som motivation for introduktionen af ubegrænsede influensdiagrammer.

Primær kilde til dette kapitel er [Jensen, 2001]. Dette kapitel er lettere revideret udgave af et tilsvarende kapitel fra [Ahlmann-Ohlsen and Pedersen, 2005].

1.1 Baggrund

Beslutningsproblemer er en klasse af problemer, der beskæftiger sig med den proces, det er at træffe beslutninger under usikkerhed, dvs. der skal træffes beslutninger, hvor det ikke er muligt at være sikker på udfaldet af disse, og hvor det samtidigt kan være nødvendigt at træffe de valg, der vil være mest gavnlige for beslutningstageren.

Beslutningsproblemer med én beslutningstager klassificeres indenfor tre typer af problemstillinger: Én problemstilling er dét at træffe beslutninger i en fast sekvens; fx vælger man først om man vil have cornflakes eller havregryn til morgenmad, hvorefter man bestemmer, om man vil have sukker på morgenmaden. Det sidste valg kan afhænge af, hvad man valgte til morgenmad og af, om der er sukker i huset. Usikkerheden kommer til udtryk idet, beslutningstageren ikke kender til, om der er sukker i huset, førend han har observeret om dette er tilfældet. Omkostningerne ved at tage sukker på morgenmaden afhænger af, om man har sukker, eller om man først skal ud og købe ind. Det karakteristiske ved denne problemstilling er, at ligegyldigt hvilke beslutninger der bliver truffet, skal de altid træffes i *samme rækkefølge* (først vælges typen af morgenmad, og dernæst vælges sukkeret til eller fra). Når beslutningerne og rækkefølgen af beslutningerne på denne måde er den samme for alle de forskellige beslutningsforløb, dette beslutningsproblem tillader, kalder vi beslutningsproblemet symmetrisk. Vi kommer med en formel definition af symmetri i definition 1.2.

En anden problemstilling handler ikke blot om at træffe de bedste beslutninger, men også dét at finde den bedste rækkefølge af beslutningerne. Eksempelvis skal en person en morgen beslutte sig for hvilket overtøj, han skal have på, og om han skal køre på cykel eller tage bussen. Det første han gør er at kigge ud ad vinduet for at se, hvordan vejret er. Hvis solen skinner og vejret er godt, kan han tage tøj på uden at overveje, om han skal køre på cykel eller ej. Hvis det derimod regner og vejret er koldt, vil det være en god ide først at vælge, om han skal med bus eller afsted på cykel, da en cykeltur i denne type vejr stiller specielle krav til påklædningen. Da rækkefølgen af beslutningerne varierer alt afhængigt af hvordan vejret er, vil de forskellige kombinationer over beslutningerne i dette tilfælde ikke længere udvikle sig symmetrisk, til trods for at alle beslutninger stadig skal træffes.

Den tredje og sidste type af problemer, der er inkluderet i klassen af beslutningsproblemer med én beslutningstager, er dét at finde optimale sekvenser af beslutninger; hvis man vælger at køre på cykel i stedet for at tage med bussen, skal man, efter at have observeret om det er mørkt, beslutte, om man vil tage cykellygter med – et valg der ikke er relevant og dermed ikke nødvendigt at træffe, hvis man vælger bussen. I dette tilfælde kan valget for

de første beslutninger få indflydelse på, *hvilke* beslutninger der efterfølgende skal tages stilling til, og samtidigt kan de optimale valg for de efterfølgende beslutninger afhænge af de forudgående beslutninger og observationer. Det er derfor klart at dette problem ikke har de symmetriske egenskaber, som vi så i den første problemstilling.

En klassisk måde at repræsentere beslutningsproblemer af ovenstående typer på er som beslutningstræer. Et beslutningstræ, BT, er en træstruktur, der vha. chanceknuder, beslutningsknuder og nytteknuder danner en graf over beslutningsproblemets struktur. Dette gør det muligt at anvende den som en prædiktiv model, der, som navnet antyder, kan anvendes ved beslutningstagning. BTer er defineret i definition 1.1.

Definition 1.1 (**Beslutningstræ**). Et beslutningstræ er et orienteret og rodfæstet træ bestående af beslutningsknuder (*rektangler*), chanceknuder (*cirkler*), og nytteknuder (*diamanter*). Nytteknuder er altid, og som de eneste, bladknuder.

En kant fra en chanceknude skal være mærket med en tilstand for chanceknuden. Tilsvarende gælder det for kanter fra en beslutningsknude, at de skal være mærket med en beslutning fra knuden.

Det er et krav at beslutningstræer er *komplette*, dvs. at der for enhver chanceknude skal være en kant for hver mulig tilstand, ligesom der for enhver beslutningsknude skal være en kant for hver mulig beslutning.

□

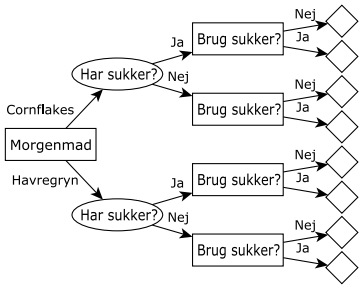
Eksempler på BTer findes i figur 1.1. BTer læses fra rodknuden ned imod en af bladknuderne, og den sekvens af knuder, som besøges udgør et *beslutningsscenarie*.

Kravet i definition 1.1, om at et beslutningstræ skal være komplet, har den implikation, at der for beslutningsproblemer altid vil være mere end én sti, der går fra roden til et blad (under antagelse af at *ikke* alle knuder er unære), som det også kendetegner figurerne i figur 1.1.

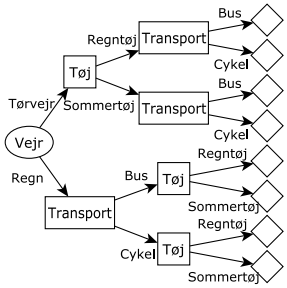
I figur 1.1 er der afbildet tre beslutningstræer, der hver især illustrerer én type af de beslutningsproblemer, der blev introduceret indledningsvis.

Et BT kan anvendes til at modellere et beslutningproblem. Dette er muligt selv for beslutningsproblemer, hvor nogle observationer eller beslutninger kan forekomme på arbitrære tidspunkter, dvs. at beslutningsproblemet ikke i sig selv specificerer rækkefølgen af disse variable i forhold til resten. Dette resulterer i flere mulige BTer for samme beslutningsproblem. En mængde bestående af flere mulige BT-repræsentationer for samme problem, kaldes en *familie* af BTer.

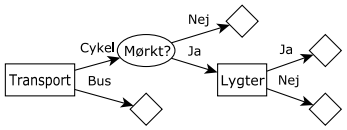
Beslutningstræet er en meget udtryksfuld repræsentationsform for beslutningsproblemer. Dvs. at det er muligt at repræsentere mange forskellige typer



(a) Symmetrisk beslutningstræ, hvor beslutningstageren først skal beslutte sig for hvad han vil have til morgenmad. Herefter kan han konstatere om han har sukker for til sidst at bestemme sig for om han vil bruge sukker.



(b) Beslutningstræ med ordningsasymmetri, hvor beslutningstageren starter med at konstatere hvordan vejret er. Denne observation kan bruges til først enten at bestemme sig for et transportmiddel, eller hvilket tøj han skal have på, men alle beslutningerne skal træffes.



(c) Beslutningstræ med strukturel asymmetri, hvor beslutningstageren skal bestemme sig for et transportmiddel. Hvis han vælger cyklen, kan han konstatere om det er lyst eller mørkt udenfor, afhængigt af hvilket, det er nødvendigt at beslutte om han vil have sine lygter med.

Figur 1.1: Beslutningstræer der illustrerer symmetri og forskellige former for asymmetri.

problemer, uden at sløre forståelsen af modellen. En fordel ved BTer er at netop førnævnte egenskab kombineret med den relativt simple model, der ligger til grund for disse, gør BTer simple at forstå. Desværre vokser størrelsen af BTer eksponentielt med størrelsen af beslutnings- og chancevariable, hvilket hurtigt gør dem svært overskuelige, hvorfor BTer ikke er egnede til repræsentation af store beslutningsproblemer.

Figur 1.1(a) viser et beslutningsproblem, hvor rækkefølgen såvel som mængden af beslutninger er bestemt, og ens, for alle scenarier. Dermed illustrerer figuren en problemstilling, hvor det udelukkende drejer det sig om at finde de bedst mulige *instanser* for beslutningerne i en fast sekvens af beslutninger. Et sådan problem kaldes *symmetrisk* og er formelt defineret i definition 1.2 [Bielza and Shenoy, 1999] [Jensen et al., 2006a].

Definition 1.2 (**Symmetri**). Lad **BT** være mængden af alle beslutningstræsrepræsentationer over et beslutningsproblem, BP. BP siges at være symmetrisk hvis

1. for alle $BT \in \mathbf{BT}$ gælder at antallet af mulige scenarier i BT er lig med størrelsen af det kartesiske produkt over alle beslutnings- og chancevariable i BP.

2. i mindst et $BT \in \mathbf{BT}$ er rækkefølgen af alle beslutnings- og chancevariable den samme for alle scenarier i BT.

□

Problemstillingerne illustreret i figurerne 1.1(b) og 1.1(c) overholder derimod ikke definition 1.2, idet de begge afviger fra det andet krav i definitionen; rækkefølgen af alle beslutnings- og chancevariable i beslutningsproblemet er *ikke* den samme for alle scenarier. Ligeledes overholder BTet i figur 1.1(c) heller ikke det første krav, da antallet af beslutningsscenarier gennem BTet ikke er lig med størrelsen af det kartesiske produkt over variablene i BTet. Problemer, der afviger fra definition 1.2, kaldes *asymmetriske*.

I figur 1.1(b) ses det at mængden af beslutninger og observationer er den samme for alle scenarier. Asymmetriske problemer med denne egenskab siges at være *ordningsasymmetriske*. Modsat opfylder problemet illustreret i figur 1.1(c) ikke denne egenskab, da det klart fremgår at der findes scenarier, hvor mængden af mulige beslutninger er forskellige. Sådanne problemer kaldes *strukturelt* asymmetriske.

Når man arbejder med beslutningsproblemer, er formålet at kunne afgøre hvilke beslutninger, der er mest gavnlige for beslutningstageren i de konkrete situationer. For at kunne afgøre, hvilke beslutninger der forventes at være bedst, har vi brug for et enhedssystem til at angive og sammenligne udgifter, omkostninger, priser, gevinster med videre. Alle disse størrelser kalder vi tilsammen for *nytte*. Negativ nytte er således fx udgifter og omkostninger, mens positiv nytte fx er gavn, belønninger og gevinster.

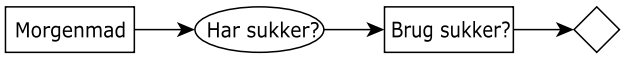
Vi kan anvende nytte-begrebet, til at afgøre hvilke beslutninger, der skal tages under givne forudsætninger. Den forventede bedste beslutning under en given forudsætning kalder vi således den *optimale beslutning*. Når flere beslutninger følger hinanden i et beslutningsscenarie, snakker vi om et *optimalt* beslutningsscenarie. I et optimalt beslutningsscenarie vil alle beslutninger være optimale.

Med BTer søger man at kunne støtte valg under usikkerhed med det mål at opnå den største forventede nytte. Dvs. at man søger at afgøre hvilke beslutninger, der indgår i det optimale beslutningsscenarie. Netop fordi det optimale beslutningsscenarie kan være påvirket af ukontrollerbare faktorer, er det ikke altid muligt at bestemme én fast sekvens af valg, der altid vil give det optimale resultat. Det betyder at det ikke er nok for en løsning til

et BT at foreskrive en enkelt serie af beslutninger, men snarere foreskrive den forventede mest optimale beslutning for ethvert skridt i ethvert muligt scenarie. Sådan en løsning kaldes en *strategi* (se afsnit 1.4).

Som tidligere nævnt er det et problem for BTer, at de hurtigt bliver svære at overskue og ikke mindst repræsentere i en computer, idet deres størrelse vokser eksponentielt med antallet af beslutnings- og chancevariable. Denne egenskab har også den betydning, at det bliver sværere at finde en strategi for BTet idet kompleksiteten stiger.

Der er blevet foreslået en række metoder til at reducere kompleksiteten af BTer, fx sammenflettede BTer, der genbruger identiske undertræer i strukturen. Dette reducerer ganske vist størrelsen, men gør strukturen mere kompleks i kraft af nye forbindelser mellem grenene. En anden tilgangsvinkel, der bevarer overskueligheden ved symmetriske beslutningsproblemer, er at folde BTet sammen til en kædestruktur. Denne ide ligger til grund for *influensdiagrammer*, et repræsentationssprog til repræsentation af symmetriske beslutningsproblemer. Figur 1.2 er et eksempel på en sammenfoldning af BTet fra figur 1.1(a).



Figur 1.2: Kæderepræsentation af beslutningstræet i figur 1.1(a), hvor rækkefølgen af beslutninger såvel som observationer er totalt ordnet.

1.2 Influensdiagrammer

Når symmetriske beslutningsproblemers BTer kan foldes sammen til en kæde, som vist i figur 1.2, bliver repræsentationen af beslutningsproblemerne ofte mere overskuelige. Dette princip udnyttes ved influensdiagrammer, IDer, der er introduceret som en udvidelse af BTer med henblik på at kunne repræsentere symmetriske beslutningsproblemer mere kompakt. Dog er det som tidligere nævnt også muligt at anskue IDer som en udvidelse af bayesianske net til bayesianske net med beslutninger.

Et eksempel på et symmetrisk beslutningsproblem, og dermed et beslutningsproblem der med fordel kan specificeres som et ID, er givet i det følgende:

> 100 teenagepiger er mødt op til casting til et nyt talentshow på tv. En person, casteren, har fået til opgave at finde de piger, der skal være med i showet. Der er tre kriterier, som pigerne bliver bedømt ud fra: udseende og talent for at synge og danseevner.

Statistikken viser, at 40% af pigerne er kønne, og at 20% af pigerne har talent. Derudover ved casteren af erfaring, at hvis en pige har talent, så vil hun med 60% chance klare sangopgaven, og med 70% chance klare danseopgaven. Hvis en pige derimod ikke har talent, vil hun kun have 30% chance for at klare hver opgave. Der er ingen sammenhæng mellem en piges udseende og hendes talent.

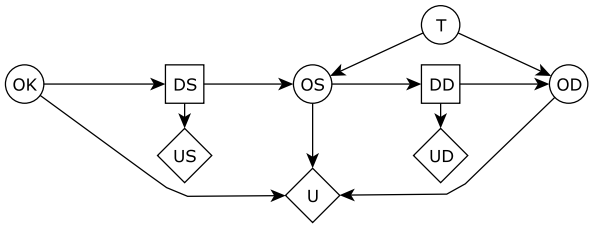
I en sangtest går der typisk fem minutter (nytte -25), før casteren kan afgøre om den håbefulde teenagepige klarede testen eller ej. Til gengæld afgøres dansetesten noget hurtigere med kun tre minutter (nytte -15). Pigens udseende kan bedømmes umiddelbart.

På trods af at TV-showet er et talentshow med fokus på sang, er det ikke muligt at komme med i showet ved blot at være god til at synge; det skal være i kombination med enten det at være køn (nytte 180), at være god til at danse (nytte 160), eller begge dele (nytte 200). Kombinationen af at være køn og kunne synge er at foretrække fremfor at kunne synge og kunne danse, da man mener at alle kan lære at danse. Kombination af ikke at kunne synge, men dog være køn og kunne danse er mindre attraktiv (nytte 100), men dog stadig attraktiv for tv-selskabet, da kønne mennesker måske kan fastholde flere seere.

Når casteren har observeret om en pige er køn, skal han beslutte, om han vil stille hende overfor sangtesten. Når han kender resultatet af sangtesten (såfremt han stillede denne test), skal han beslutte om han vil stille pigen over for danseopgaven. Casteren ønsker at vide, hvilke tests han skal stille pigerne, når han har observeret om de er kønne eller ej med henblik på at få de bedste piger men samtidigt også at få castet alle pigerne hurtigt.

Ovenstående problem modelleres som et ID ved at identificere beslutninger, chancevariable og nytter, samt disses interne sammenhænge. Dette giver IDet som er illustreret på figur 1.3 på den følgende side. Mængden af beslutninger, der skal træffes består af (1) om sangtesten skal tages, DS, og (2) om dansetesten skal tages, DD; begge afbildet som rektangler i figuren. Chanceknuderne (afbildet som cirkler) består af om pigen er køn, OK, om hun kunne danse, OD, og om hun kunne synge, OS. Om pigen har talent, T, er også inkluderet som en chanceknude, men til forskel fra de andre chanceknuder observeres denne aldrig direkte. Derudover er de forskellige nytter også inkluderet i modellen i form af diamanter, hvor US er nytten af at udføre sangtesten, UD er nytten af at udføre dansetesten og U er nytten af pigen,

som resultat af hvordan hun klarer sig i de forskellige test.



Figur 1.3: Castingproblemet som ID: Afhængigt af en piges udsende (OK) ønsker casteren først at vide, om han skal stille sangopgaven (DS) og derefter ønsker casteren at vide, om han skal stille danseopgaven (DD). U er nytten af at finde en pige med kompetencerne OK, OS og OD, mens US og UD er prisen for at få udført henholdsvis en sangtest og en dansetest.

Med udgangspunkt i figur 1.3 virker det klart, med kendskab til variablene i IDet, at IDet giver et præcist og let forståeligt overblik over beslutningsproblemet. At indse at denne repræsentation tilmed er væsentligt mindre end den tilsvarende BT repræsentation synes trivielt med udgangspunkt i kravet om, at et BT skal være komplet (se definition 1.1 på side 7).

1.3 Definition af influensdiagrammer

I det foregående afsnit blev IDer introduceret som en kompakt måde at repræsentere symmetriske beslutningsproblemer og i eksemplet bestod IDet af beslutninger og chancevariable, såvel som nytter. Mere formelt defineres IDer i definition 1.3.

Definition 1.3 (**Influensdiagram**). Et influensdiagram, ID, er en orienteret acyklisk graf over chancevariable (*cirkler*), beslutningsvariable (*rektangler*) og nytteknuder (*diamanter*).

Det gælder for et ID at:

- Der er en orienteret sti, gennem alle beslutningsvariable.
- En knude har ingen børn hvis og kun hvis det er en nytteknude.

□

Af kvantitative krav gælder følgende:

- Beslutnings- og chancevariablene har hver en endelig mængde gensidigt udelukkende og fyldestgørende tilstande.
- Til hver chancevariabel, X, er der en betinget sandsynlighedstabel, der er en funktion af typen $P : st(\{X\} \cup \pi(X)) \rightarrow [0;1]$. Her angiver

st-operatoren en konfiguration over en eller flere variable, mens $\pi(X)$ betegner forældrene til variablen X. P er da en funktion af tilstandene over X og Xs forældre til en sandsynlighed – denne funktion kaldes en sandsynlighedsfunktion eller blot et potentiale.

- For hver nytteknude, U, er der en funktion af typen $U : st(\pi(U)) \rightarrow \mathbb{R}$, der afbilder fra tilstandene over Us forældre til et reelt tal.

Hvis man for et ID tillader ikke-nytteknuder at have børn, så kaldes disse for *barren*-knuder. Men fordi de ikke vil kunne påvirke nogen anden knude i IDet, er det rimeligt at fjerne dem [Shachter, 1986]. Hvorfor det er rimeligt at definere nytteknuder som værende de eneste knuder i et ID uden børn.

En knude i et ID er navngivet efter den variabel den repræsenterer, og der er som sådan en semantisk forskel på knuder og variable. Dog vil denne spidsfindighed i flere tilfælde ikke have nogen betydning for arbejdet i denne rapport, hvorfor betegnelserne variable og knuder ofte vil kunne betragtes som synonymer.

I forbindelse med definitionen af et ID knytter der sig en semantik, der stiller krav til IDets struktur og udtrykker sig om betydningen af denne. Semantikken for et ID er som følger:

Kanter fra chancevariable ind i beslutningsvariable kaldes *informationskanter* og angiver at chancevariablene observeres inden beslutningen kan tages. Kanter mellem beslutningsvariable kaldes *præcedenskanter* og angiver temporale ordninger af beslutningerne. Kanter ind i nyttefunktioner angiver funktionelle afhængigheder. Kanter ind i chancevariable kaldes *kausale kanter*, og angiver kausal afhængighed.

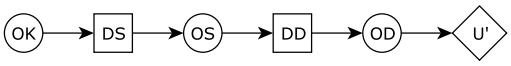
Der er antaget *no-forgetting*, hvilket vil sige, at beslutningstageren til enhver tid kender til alle de forudgående beslutninger og observationer.

Det er et krav til et ID, at der findes en sti af kanter, der går i gennem alle beslutningsvariable. Denne sti angiver den temporale ordning af beslutningsvariablene. Givet denne ordning, kan der opstilles en partiel temporal ordning af alle chancevariablene, der angiver hvordan disse er ordnet i forhold til beslutningsknuder, men ikke nødvendigvis i forhold til hinanden.

For at opstille den partielle temporale ordning lader vi mængden I_0 angive mængden af chancevariable observeret inden den første beslutning. Ligeledes angiver mængden I_i chancevariable observeret efter beslutning D_i men før beslutningen D_{i+1}. Til sidst angiver I_n mængden af chancevariable der ikke nødvendigvis observeres – n er her antallet af beslutningsvariable.

Med udgangspunkt i førnævnte mængder er den temporale ordning for et ID givet ved $I_0 \prec D_1 \prec I_1 \prec \ldots \prec D_n \prec I_n$. For IDet på figur 1.3 på forrige side gælder det at $I_0 = \{OK\}$, $D_1 = \{DS\}$, $I_1 = \{OS\}$, $D_2 = \{DD\}$ og $I_2 = \{OD, T\}$. Denne ordning anvendes til at lave en kæde for IDet, der kan

foldes ud til ét BT (jf. afsnit 1.1). Alle stier igennem et BT slutter med en nytteknude (se definition 1.1) og i et BT for et ID er denne nytteknude lig med summen af alle nytteknuderne i IDet. Figur 1.4 viser et eksempel på en kæde for IDet på figur 1.3.



Figur 1.4: En kædepræsentation af IDet for castingproblemet på figur 1.3, hvor $U' = U + US + UD$.

1.4 Løsning af influensdiagrammer

Som ved et BT søger man med IDer at konstruere modeller, der kan anvendes til at støtte en beslutningstager gennem et scenarie. Atter forstås støtten som rådgivning om hvilket næste skridt, der vil give højest nytte. Løsningen for et ID er givet i form af en strategi, der søger at maksimere nytten i forbindelse med gennemførelse af et scenarie.

Som følge af usikkerhederne i et beslutningsproblem er det ikke muligt at bestemme den eksakte nytte, som en beslutningstager vil opnå; det er som oftest ikke muligt at bestemme fremtiden helt sikkert. Dette giver anledning til at tale om *forventet* nytte af en strategi, når en løsning til et ID såvel som førnævnte BTer skal findes. Det har den konsekvens at en løsning til et ID, vil søge at støtte en beslutningstager ved at angive det næste skridt, der giver størst forventet nytte.

For en variabel, N, er mængden af alle forudgående variable (eksklusiv N selv) *fortiden* for N og betegnes $Past(N)$. Fortiden for en variabel, N, kaldes også Ns historie.

Mere formelt gives vejledningen til beslutningstageren om hvilken beslutning han skal træffe ved de forskellige beslutningsknuder i form af *politikker*. En politik for en beslutningsknude, D, er således en funktion af typen $\delta : st(Past(D)) \rightarrow st(D)$, der fortæller beslutningstageren, hvilken beslutning han skal tage givet en specifik fortid for D. En strategi for et ID består af en mængde bestående af en politik for hver beslutning i IDet. En strategi angiver altså hvilke beslutninger, beslutningstageren skal tage i det beslutningsproblem, som IDet repræsenterer.

Der er intet krav om at en strategi skal maksimere den forventede nytte. En strategi i sig selv kan altså misvejlede beslutningstageren, men hvis strategien er *optimal*, er den garanteret at maksimere den forventede nytte. En sådan strategi udgør løsningen af et ID.

Et ID kan i princippet løses ved at folde IDet ud til et BT, og løse det fremkomne BT. Denne metode er dog unødig kompleks, idet et ID kan løses mere effektivt ved at udnytte dets struktur.

Hvis man ser en politik for en beslutning som en deterministisk chancevariabel, der altid giver det udfald, som politikken specificerer, så kan man beregne den forventede nytte, FN, af en strategi, Δ, bestående af politikkerne, δ_D, for alle beslutninger, D, i mængden af beslutninger i IDet, W_D, som

$$FN_\Delta = \sum_W \left(\prod_{D \in W_D} P(\delta_D | Rel(D)) \prod_{C \in W_C} P(C|\pi(C)) \cdot U \right), \tag{1.1}$$

hvor W er alle variable i IDet, W_C er mængden af chancevariable i IDet, U er summen af alle nyttefunktionerne til IDets nytteknuder og $Rel(D)$ angiver den relevante fortid for D – det vil sige de variable i Ds fortid, der kan have betydning for det optimale valg for beslutningen D [Shachter, 1999, Nielsen and Jensen, 1999].

Den optimale strategi, Δ^*, er således den samling af politikker, der maksimerer den forventede nytte,

$$\Delta^* = argmax_\Delta FN_\Delta, \tag{1.2}$$

hvor $argmax_\Delta$ giver den strategi, Δ, der maksimerer den forventede nytte, FN_Δ.

Det er dog praktisk umuligt at prøve sig frem med alle mulige strategier, for at finde den strategi, der maksimerer den forventede nytte. I stedet kan man lave en rekursiv bestemmelse af de politikker der indgår i den optimale strategi, således at man starter med at finde den maksimale forventede nytte for den sidste beslutning, ρ_n, som

$$\begin{aligned} &\rho_n(I_0, D_1, \ldots, I_{n-1}) \\ &= \frac{1}{P(I_0, \ldots, I_{n-1} | D_1, \ldots, D_{n-1})} max_{D_n} \sum_{I_n} P(W_C|W_D) \cdot U, \end{aligned} \tag{1.3}$$

hvor $max_\mathcal{X}\phi$ og $\sum_\mathcal{X}\phi$ er hhv. max- og sum-marginaliseringsoperatoren, der marginaliserer variablene i mængden $\mathcal{X}$ ud af potentialet ϕ. Når $\mathcal{X}$ er marginaliseret ud af ϕ gælder det, at variablene, som det nye potentiale, ϕ', er defineret over, domænet af ϕ' (skrevet $dom(\phi')$), er $dom(\phi') = dom(\phi) \backslash \mathcal{X}$.

Den optimale politik for den sidste beslutning kan samtidigt findes ved at udskifte max_{D_n} med $argmax_{D_n}$, hvilket giver os den instans af D_n, der maksimerer den forventede nytte. Vi får altså

$$\begin{aligned}
&\delta_n(I_0, D_1, \ldots, I_{n-1}) \\
&= \frac{1}{P(I_0, \ldots, I_{n-1}|D_1, \ldots, D_{n-1})} argmax_{D_n} \sum_{I_n} P(W_C|W_D) \cdot U \\
&= argmax_{D_n} \sum_{I_n} P(W_C|W_D) \cdot U. \qquad (1.4)
\end{aligned}$$

Når den optimale politik er fastlagt for den sidste beslutning, kan man finde den optimale politik for den *næstsidste* beslutning, $\delta_{D_{n-1}}$, under antagelse af at man i fremtiden træffer de optimale valg. Den optimale politik for den næstsidste beslutning kan findes, når politikken for den sidste beslutning er fastlagt. Når politikken for den sidste beslutning er fastlagt, kan man betragte den sidste beslutning som en deterministisk chanceknude, hvis udfald er defineret ved den fastlagte optimale politik. Da den sidste beslutning således kan opfattes som en deterministisk chanceknude, kan politikken for den næstsidste beslutning også bestemmes med udtryk (1.4). Således kan man fortsætte baglæns gennem ordningen af beslutninger indtil alle de optimale politikker er fundet. Mængden af disse politikker udgør den optimale strategi.

Når man først har de optimale politikker for de efterfølgende beslutninger $(D_{i+1} \ldots D_n)$, bliver den maksimale forventede nytte for D_i således

$$\begin{aligned}
&\rho_i(I_0, D_1, \ldots, I_{i-1}) \\
&= \frac{1}{P(I_0, \ldots, I_{i-1}|D_1, \ldots, D_{i-1})} \\
&\quad max_{D_i} \sum_{I_i} max_{D_{i+1}} \sum_{I_{i+1}} \ldots max_{D_n} \sum_{I_n} P(W_C|W_D) \cdot U, \qquad (1.5)
\end{aligned}$$

mens den optimale politik for D_i bliver

$$\begin{aligned}
&\delta_i(I_0, D_1, \ldots, I_{i-1}) \\
&= argmax_{D_i} \sum_{I_i} max_{D_{i+1}} \sum_{I_{i+1}} \ldots max_{D_n} \sum_{I_n} P(W_C|W_D) \cdot U. \qquad (1.6)
\end{aligned}$$

Grunden til, at det er en beregningsmæssig fordel at bestemme politikkerne baglæns gennem ordningen af beslutningerne skyldes, at den forventede nytte af politikken δ_i for beslutning D_i afhænger af, hvilke valg man vælger at træffe i fremtiden. Hvis disse valg er fast defineret i form af en optimal

politik for alle beslutninger i fremtiden, så forelægger der et deterministisk grundlag for bestemmelsen af den optimale politik for D_i. Dette gør ovenstående metode til bestemmelse af den optimale strategi mere operationel end udtryk (1.2).

Denne rekursive bestemmelse af de optimale politikker og den forventede nytte er også ideen bag variabelelimineringsalgoritmen, der er en algoritme til løsning af IDer.

1.4.1 Variabelelimineringsalgoritmen

Variabelelimineringsalgoritmen [Madsen and Jensen, 1999a] forløber som følger: Først deles alle potentialer for IDet op i to mængder. Én mængde til sandsynlighedspotentialerne og en mængde til nyttepotentialerne. Med udgangspunkt i de to mængder af potentialer elimineres én variabel ad gangen. Chancevariable sum-marginaliseres, mens beslutningsvariable max-marginaliseres. Da sum- og max-marginalisering ikke kommuterer, er elimineringsrækkefølgen underlagt den temporale ordning for IDet og variablene elimineres modsat den rækkefølge, hvori de bliver observeret eller besluttet. Dvs. at variablene skal elimineres i en rækkefølge, hvor først variablene fra I_n elimineres, derefter $D_n, I_{n-1}, D_{n-1} \ldots I_0$. Dette kaldes en *stærk* elimineringsrækkefølge.

Lad Φ betegne mængden af sandsynlighedspotentialer for et ID og lad Ψ betegne mængden af nyttepotentialer. Når variablen N skal elimineres, identificeres først mængden af sandsynlighedspotentialer med N i domænet, Φ_N, og mængden af nyttepotentialer med N i domænet, Ψ_N

$$\Phi_N \leftarrow \{\phi \in \Phi | N \in dom(\phi)\} \tag{1.7}$$
$$\Psi_N \leftarrow \{\psi \in \Psi | N \in dom(\psi)\}. \tag{1.8}$$

Herefter elimineres N ud af de mængder af potentialer, der indeholder den i deres domæner, hvorved vi får mængderne ϕ_N og ψ_N. Hvis N er en chancevariabel er ϕ_N og ψ_N givet ved følgende udtryk:

$$\phi_N \leftarrow \sum_N \prod \Phi_N \tag{1.9}$$
$$\psi_N \leftarrow \sum_N \prod \Phi_N \left(\sum \Psi_N \right), \tag{1.10}$$

og hvis N er en beslutningsvariabel gives de ved:

$$\phi_N \leftarrow max_N \prod \Phi_N \tag{1.11}$$

$$\psi_N \leftarrow max_N \prod \Phi_N \left(\sum \Psi_N \right). \tag{1.12}$$

Her har vi brugt notationen $\prod \Phi$ til at betegne produktet over alle sandsynlighedspotentialerne i mængden Φ og $\sum \Psi$ til at betegne summen af alle nyttepotentialerne i mængden Ψ.

Tilsidst forenes resultatet af elimineringen med de potentialer, der ikke indeholdt N i deres domæne:

$$\Phi \leftarrow (\Phi \backslash \Phi_N) \cup \{\phi_N\} \tag{1.13}$$

$$\Psi \leftarrow (\Psi \backslash \Psi_N) \cup \left\{ \frac{\psi_N}{\phi_N} \right\}. \tag{1.14}$$

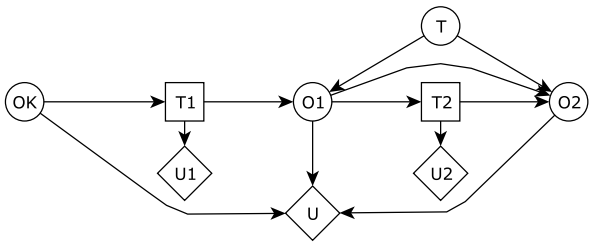
Udtryk (1.13) forener det resulterende sandsynlighedspotentiale efter marginaliseringen af N med de sandsynlighedspotentialer som ikke indeholdte N. En tilsvarende operation sker i udtryk (1.14), men her er vi nødt til at dividere med ϕ_N for at skabe en konsistent operation. Hvis vi ikke dividerede med ϕ_N ville en nærmere matematisk analyse af algoritmens rekursive forløb afsløre, at denne faktor kan blive multipliceret med nyttepotentialet flere gange.

For de matematiske detaljer bag variabelelimineringsalgoritmen henviser vi til [Madsen and Jensen, 1999a] og [Jensen, 2001].

1.5 Ulemper ved influensdiagrammer

Vi har nu set, hvordan vi kunne modellere castingproblemet med et ID. Med IDer som modelleringssprog blev det muligt at opstille en intuitiv model for beslutningsproblemet.

Det var naturligt at vælge IDer som repræsentationssprog til castingproblemet, da der i castingproblemet var dikteret en total ordning af beslutninger og castingproblemet således var et symmetrisk beslutningsproblem. Lige netop denne totale ordning giver dog anledning til en kunstig begrænsning af casteren – Det virker ulogisk, at casteren skal være begrænset til først at beslutte, om han vil stille sangopgaven og derefter tage stilling til, om han vil stille danseopgaven. Fjernes denne begrænsning fra eksemplet, findes den totale ordning over beslutningerne ikke længere, hvorved problemet vil være blevet ordningsasymmetrisk. Dette betyder, at modellen skal revideres til at indeholde begge mulige rækkefølger af beslutningerne.



Figur 1.5: ID repræsentation for det modificerede castingproblem uden begrænsninger på rækkefølger af test.

Da definitionen af et ID foreskriver, at der skal findes én sti gennem alle beslutninger i IDet, og det ikke giver mening at inkludere den samme variabel mere end en gang, kræver det en eksplicit modellering af de mulige rækkefølger i den grafiske model. Dette kan gøres ved at introducere kunstige variable eller tilføje kunstige tilstande til chancevariablene.

Uden en givet rækkefølge på beslutningerne i casting problemet, fjernes de to testvariable, DS og DD, fra det tidligere ID og erstattes af to nye generelle testvariable, $T1$ og $T2$, med tilstandene *sangtest*, *dansetest* og *ingen test*. Desuden får observationen $O1$ direkte indflydelse på resultatet af $T2$, observation $O2$, for at kunne modellere, at en gentagelse af en test altid vil give det samme testresultat – en spidsfindighed der ikke var nødvendig at tage højde for, dengang sang- og dansetestene havde hver deres knude i grafstrukturen. Resultatet af denne modificering af modellen for castingproblemet kan ses i figur 1.5.

Modificeringen har betydet, at de nye chancevariable, $O1$ og $O2$, hver har fået fem tilstande – to for hver mulig test (positiv og negativ), samt tilstanden *ikke observeret*. Dermed får en tabel over sandsynlighedspotentialet for $O2$ 150 indgange ($|st(O2)| \cdot |st(T2)| \cdot |st(O1)| \cdot |st(T)| = 5 \cdot 3 \cdot 5 \cdot 2 = 150$). Som reference havde den tilsvarende chancevariabel, OD, en potentialetabel med blot 8 indgange.

Ikke blot har ophævelsen af ordningen af beslutninger resulteret i flere tilstande i IDet, men modellen er også blevet gjort mere kompleks og sværere at overskue. Dette afspejler sig dog kun ganske moderat i IDet, grundet størrelsen af beslutningsproblemet.

Den nye model, der repræsenterer det modificerede casting problem, er blevet mindre intuitiv end tidligere; der findes ikke længere en klar afspejling af beslutningsproblemet i modellen og i stedet for klare unikke beslutningsvariable, bliver flere beslutninger rodet sammen over flere knuder.

På trods af væksten i størrelsen af potentialerne såvel som sløringen af

modellen og det faktum at IDer opnår den største fordel over beslutningstræer ved symmetriske beslutningsproblemer, er IDer stadig i stand til at repræsentere asymmetriske beslutningsproblemer, som vi netop har vist.

Det har dog den konsekvens, at det er nødvendigt at tilføje ekstra variable eller kanter til IDet eller ekstra tilstande til IDets variable, der ikke umiddelbart reflekterer det beslutningsproblem, der søges repræsenteret.

Denne uheldige egenskab leder frem til et behov for et andet repræsentationssprog, da en mere direkte repræsentation af ordningsasymmetri ville være at foretrække. En sådan repræsentation er mulig med ubegrænsede influensdiagrammer, som vi vil behandle i næste kapitel.

Kapitel 2

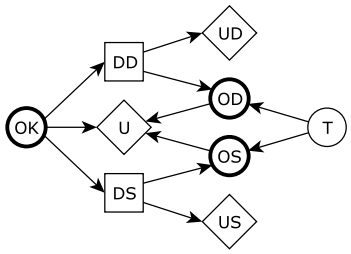
Ubegrænsede influensdiagrammer

Vi har nu set på BTer og et alternativ til disse i form af IDer. Begge har hver deres styrker og svagheder. I dette kapitel introduceres endnu et alternativ, der kompenserer for en del af disse svagheder. Dette kapitel er baseret på arbejde indenfor ubegrænsede influensdiagrammer, UIDer, præsenteret i [Jensen et al., 2006b], [Jensen and Vomlelová, 2002] og [Vomlelová, 2003], samt [Jensen, 2001]. Kapitlet er en sammen- og videreskrivning af flere kapitler i [Ahlmann-Ohlsen and Pedersen, 2005]. Specielt er afsnit 2.2.2 helt nyt, mens afsnit 2.4 og 2.5 er helt omskrevet.

Det nye alternativ er UIDer, der søger at gøre repræsentationen af ordningsasymmetriske beslutningsproblemer mere kompakt og intuitiv. Dette opnås ved at udskyde repræsentation af mulige rækkefølger af beslutninger til løsningsfasen, modsat IDer der modellerer de forskellige rækkefølger eksplicit i sin modellering af problemet.

Årsagen til den, i mange tilfælde, uheldige modellering af ordningsasymmetriske beslutningsproblemer som optræder ved IDer, er kravet om at der skal være én sti gennem alle beslutninger i IDet.

UIDer indeholder i deres struktur ikke specifikke krav til den temporale ordning af det problem, de skal repræsentere. Derfor kan UIDer betragtes som en generalisering af IDer, hvor der er slækket på kravet om, at alle beslutningsvariable skal være ordnet ift. hinanden. Dette kommer til udtryk i, at der for et UID ikke nødvendigvis skal være en sti gennem alle beslutninger. Samtidigt introducerer UIDer også syntaktiske forskelle i form af grafisk adskillelse af observerbare og ikke observerbare chanceknuder, nu givet ved



Figur 2.1: Castingproblemet modeleret som et UID: Afhængigt af en piges udsende (OK) ønsker vi at vide i hvilken rækkefølge vi skal stille sang- og danseopgaverne (DS og DD) for hurtigst at finde ud af, om en teenagepige med et givet talent (T) besidder tilstrækkeligt mange af de ønskede egenskaber: (OK), at være køn, (OS), at kunne synge og (OD), at kunne danse. U er nytten af at finde en pige med kompetencerne OK, OS og OD, mens US og UD er priserne for at få udført henholdsvis en sangtest og en dansetest.

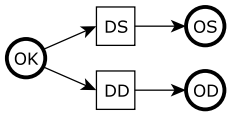
henholdsvis fede cirkler og (tynde) cirkler. Ligeledes er der med de ændrede krav ændringer i både opfattelse af, hvad der udgør en løsning, og hvordan en sådan findes – set i forhold til et ID. En UID modellering af casting problemet er vist i figur 2.1.

Som det blev nævnt i det foregående afsnit, er bevæggrunden for indførelsen af UIDer ikke nogen beregningsmæssig fordel, men snarere en repræsentationsmæssig fordel, idet flere problemkonstruktioner nu kan beskrives uden behov for tidligere nævnte *kunstige* tilstande. I mange situationer giver det en mere virkelighedstro modellering af problemerne og gør det lettere at formidle problemerne til andre, ligesåvel som de bliver lettere at repræsentere og håndtere i computere. Samtidig optræder stigninger i størrelsen af potentialer, grundet kunstige tilstande, ikke i lige så høj grad. Men de kan stadig forekomme, hvis man fx ønsker at modellere strukturel asymmetri med UIDer.

Der er, som nævnt, et strukturelt krav til IDer, at der altid er en totalt ordnet rækkefølge af de beslutninger, der optræder i modellen, mens dette krav ikke eksisterer for UIDer. Dette betyder dog ikke, at et UID ikke kan indeholde temporale krav til ordningen af beslutninger i det problem den modellerer. I et UID er det både muligt at have problemer med en totalt ordnet rækkefølge, som ved IDer, problemer med en partielt ordnet rækkefølge og problemer helt uden nogen ordning på rækkefølgen af beslutninger. Sidstnævnte bringer den største udfordring i forbindelse med løsningen af UIDerne, da det her er nødvendigt at undersøge alle mulige rækkefølger af beslutningerne for at finde en løsning i stil med dén givet for IDer (se afsnit 1.4 på side 14). Udfordringen ved totalt ordnede rækkefølger giver sig selv; det svarer til et ID, mens man ved partielt ordnede problemer stadig skal

have undersøgt alle *tilladelige rækkefølger* (også kaldet tilladelige ordninger) af variablene. Med tilladelige rækkefølger menes de rækkefølger af variable i UIDet, der overholder den givne partielle temporale ordning af variablene. Sidstnævnte mængde af rækkefølger, kan findes ved at analysere strukturen af et UID, der indeholder de temporale bånd, der kræves for at opstille den partielle ordning af variablene. Med andre ord kan den partielle temporale ordning opstilles givet et UID. Denne ordning udtrykker netop de temporale bånd, som UIDet foreskriver.

Den partielle temporale ordning for castingproblemet kan ses på figur 2.2, og det er tydeligt at den afviger fra den temporale ordning for et ID, der ville være givet ved en kæde (se afsnit 1.3 på side 12). Ud fra den partielle temporale ordning kan man opstille de tilladelige ordninger for variablene i UIDet. I det følgende bruges notationen $\prec$ til at betegne den partielle temporale ordning, der er dikteret af et UID.



Figur 2.2: Den partielle temporale ordning der er for castingproblemet på figur 2.1 på forrige side. Ud fra figuren kan det ses, at $OK \prec DS \prec DD \prec OS \prec OD$ er en tilladelig ordning, hvilket og er tilfældet for ordningen $OK \prec DS \prec OS \prec DD \prec OD$.

2.1 Definition af ubegrænsede influensdiagrammer

I det foregående afsnit, blev UIDer introduceret som et repræsentationssprog for beslutningsproblemer. Sproget har implicit mulighed for at repræsentere flere mulige rækkefølger af beslutninger og er dermed egnet til at give en kompakt og præcis repræsentation af ordningsasymmetriske beslutningsproblemer.

Ligesom IDer består UIDer af beslutninger, chancevariable og nytteknuder. Ydermere differentierer UIDer mellem observerbare og ikke observerbare chancevariable. En formel definition af et UID følger i definition 2.1.

Definition 2.1 (**Ubegrænset influensdiagram**). Et ubegrænset influensdiagram, UID, er en orienteret acyklisk graf over observerbare chancevariable (*fede cirkler*), uobserverbare chancevariable* (*cirkler*), beslutningsvariable (*rektangler*) og nytteknuder (*diamanter*).

*Også kaldet skjulte chancevariable.

Det gælder for et UID at:

- Uobserverbare chancevariable ikke må have beslutningsvariable som børn.
- En nytteknude har ingen børn.

□

For et UID gælder det, at en knude, K, er barren, hvis det er en uobserverbar chancevariabel, samt at alle Ks børn er barren. Barrenknuder kan udelades af modellen.

De kvantitative krav for et UID er de samme som for et ID (beskrevet i afsnit 1.3 på side 12), dog med den udvidelse at enhver beslutning, D, har en omkostning (evt. 0), repræsenteret ved at D indgår i domænet for en nyttefunktion. Grafisk vises dette i UIDet ved at alle beslutninger er forbundet til mindst én nytteknude. Hvis D er den eneste variabel i domænet for en nyttefunktion, så kan den grafiske repræsentation af nyttefunktionen undlades af hensyn til overskueligheden af modellen.

Ligesom de kvantitative krav, er semantikken for UIDer den samme som for IDer dog udvidet med to nye regler:

Børn af beslutningsvariable skal opfattes som ikke eksisterende indtil beslutningen er taget. Og en observationsvariabel, O, kan først observeres når alle beslutningsvariable, der er forfædre til O, er blevet besluttet. En observationsvariabel, der kan observeres, kaldes *fri*, og når den går fra bundet til fri, siger man at observationsvariablen bliver *frigivet*.

Et UID siges at være *fuldkomment*, når den grafiske struktur er udvidet med den kvantitative del.

2.2 Løsning af ubegrænset influensdiagrammer

En løsning til et UID minder om løsningen til et ID. Begge indeholder politikker for beslutningsknuderne, der afgør hvilken beslutning, der bør træffes givet en bestemt konfiguration over fortiden. Den afgørende forskel er at løsningen for et UID ydermere skal afgøre hvilken variabel, der skal være den næste i scenariet, givet en konfiguration over fortiden.

Med andre ord består løsningen af et UID af to problemer; ét, der handler om at træffe beslutninger og et andet, der består i at vælge rækkefølgen af disse.

Vi har tidligere diskuteret begrebet "tilladelige rækkefølger", der udtrykker mulige ordninger over variablene i UIDerne. De tilladelige ordninger kan repræsenteres i en såkaldt S-DAG, *Solution-DAG*, der er en grafisk struktur,

som bruges til at repræsentere mulige stier igennem et UID. S-DAGen kan således indeholde alle mulige rækkefølger af beslutninger dikteret af en vilkårlig strategi for et beslutningsproblem repræsenteret ved UIDet. En S-DAG er defineret i definition 2.2.

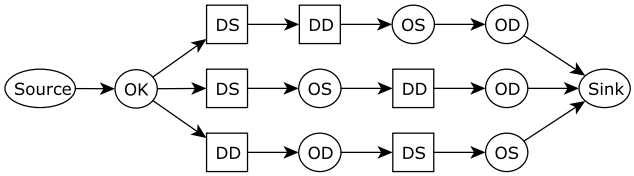
Definition 2.2 (**S-DAG**). Lad Γ være et UID med beslutningsvariable $\mathcal{D}_\Gamma$ og observerbare chanceknuder, $\mathcal{O}_\Gamma$. En *S-DAG* er en orienteret acyklisk graf Ξ. Knuderne i Ξ er navngivet med variable fra $\mathcal{D}_\Gamma \cup \mathcal{O}_\Gamma$ sådan, at enhver maksimal orienteret sti i Ξ repræsenterer en tilladelig ordning over alle variable fra $\mathcal{D}_\Gamma \cup \mathcal{O}_\Gamma$.

S-DAGen indeholder desuden to unære knuder *Source* og *Sink*. *Source* starter beslutningsproblemet og er den eneste knude uden forældre, mens *Sink* afslutter beslutningsproblemet og er den eneste knude uden børn.

□

Knuderne i en S-DAG indeholder mængder af UID-variable. Vi vil således betegne knuderne ved de variable, som de indeholder. Hvis mængden blot beslår af et element, kan mængdenotationen dog være implicit.

Figur 2.3 viser et eksempel på en S-DAG, der indeholder tre forskellige rækkefølger for variablene i det tidligere nævnte castingproblem (se figur 2.1).



Figur 2.3: En mulig S-DAG for castingproblemet. S-DAGen repræsenterer 3 forskellige tilladelige ordninger med beslutnings- og observationsvariablene fra UIDet for castingproblemet på figur 2.1.

Med udgangspunkt i S-DAG strukturen defineres strategien for et UID i kraft af en udvidelse af politikker, som de var givet for IDer (jf. afsnit 1.4 på side 14). Fortiden for en S-DAG-knude N er givet i lighed med den tilsvarende for IDer; den betegnes $Past(N)$ og betegner mængden af alle forudgående UID-variable for N (eksklusiv N selv).

Lad N være en knude i en S-DAG. En *skridtpolitik* for en knude N er en funktion med typen $\sigma : st(Past(N) \cup \{N\}) \rightarrow ch(N)$, der således angiver hvilken sti i S-DAGen, vi skal følge givet en specifik fortid samt tilstanden for N. Skridtpolitikken afhænger af fortiden for N, hvorfor flere forskellige stier i S-DAGen kan følge én knude afhængigt af den specifikke fortid. Hvis N kun har ét barn, er skridtpolitikken triviel.

En *politik* for en chanceknude er blot lig med σ, mens politikken for en beslutningsknude, D, er et par (σ, δ), hvor σ er skridtpolitikken for D

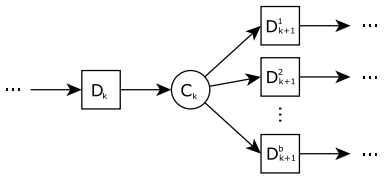
og δ er en *valgfunktion* med typen $\delta : st(Past(D)) \rightarrow st(D)$ der, givet konfigurationer over fortiden for D, angiver hvilken beslutning i D, der skal træffes.

En *strategi* for et UID er givet ved en S-DAG, Ξ, sammen med en politik for hver knude i Ξ. Hvis strategien maksimerer den forventede nytte kaldes den for en *optimal strategi* og udgør en løsning til UIDet.

For den optimale strategi for et UID gælder det, at valgfunktionen for beslutningsvariablen D_k i et UID med S-DAGen, Ξ, er givet ved

$$\begin{aligned}\delta_{D_k}(Past(D_k)) = \text{argmax}_{D_k} \sum_{C_k} P(C_k|Past(D_k), D_k)\cdot \\ \max\left\{\rho_{D^1_{k+1}}(Past(D^1_{k+1})), \ldots, \rho_{D^b_{k+1}}(Past(D^b_{k+1}))\right\},\end{aligned} \tag{2.1}$$

hvor $\rho_{D^i_{k+1}}$ er den maksimale forventede nytte for beslutning D^i_{k+1}. argmax_{D_k} giver os den beslutning i beslutningsknuden D_k, der maksimerer den forventede nytte. Således er dén del af udtryk (2.1), der følger argmax_{D_k} et udtryk for den forventede nytte. $P(C_k|Past(D_k), D_k)$ er en betinget sandsynlighedsfordeling over de chancevariable C_k, som måtte følge D_k i S-DAGen. Udtrykket gælder for situationer med forgreningspunkter i S-DAGen som vist på figur 2.4. Max-kombineringen i udtrykket er en max-kombinering over potentialer, der hver især udtrykker den forventede nytte, der opnås ved at følge den tilhørende gren i S-DAGen. Hvert element, der max-kombineres over, repræsenterer således en gren i S-DAGen. I udtryk (2.1) svarer b således til antallet af forgreninger efter knuden med variablen C_k.



Figur 2.4: Uddrag af en S-DAG der, efter beslutning D_k og dennes frigivne chanceknuder C_k, forgrener sig til b mulige efterfølgende stier.

Den maksimale forventede nytte for beslutning D_k i Ξ er givet ved

$$\begin{aligned}\rho_{D_k}(Past(D_k)) = \max_{D_k} \sum_{C_k} P(C_k|Past(D_k), D_k)\cdot \\ \max\left\{\rho_{D^1_{k+1}}(Past(D^1_{k+1})), \ldots, \rho_{D^b_{k+1}}(Past(D^b_{k+1}))\right\}.\end{aligned} \tag{2.2}$$

Udtrykket er parallelt til udtryk (2.1), blot har vi udskiftet argmax og max, således at vi får nytten af den enkelte beslutning fremfor argumentet for beslutningen.

Skridtpolitikkerne for S-DAG-knuden C_k med børnene $D^1_{k+1} \dots D^b_{k+1}$ kan opstilles ved

$$\sigma_{C_k}(Past(C_k) \cup C_k) = \text{argmax}_{\mathcal{D}} \left\{ \rho_{D^1_{k+1}}(Past(D^1_{k+1})), \dots, \rho_{D^b_{k+1}}(Past(D^b_{k+1})) \right\}, \tag{2.3}$$

hvor $\mathcal{D}$ her er mængden af børn til C_k, og $\text{argmax}_{\mathcal{D}}$ giver dét barn, der maksimerer den forventede nytte.

2.2.1 En løsningsmetode – normalforms S-DAGe

Den løsningsmetode, der er givet i [Jensen and Vomlelová, 2002, Jensen et al., 2006b], foreskriver at et UID løses ved at finde den partielle temporale ordning over variablene i UIDet og repræsentere denne ordning i en S-DAG. Når S-DAGen er opstillet, løses den, hvorved man kan finde den optimale politik for hver knude, og dermed løse UIDet. I løsningsmetoden er det et krav til den opstillede S-DAG, at den er garanteret at indeholde den ordning på variablene, der vil føre til den største forventede nytte for alle mulige udfald af de observerbare chancevariable.

Den garanti kan opnås ved en S-DAG, der inkluderer samtlige rækkefølger af variable gennem UIDet, men en sådan S-DAG vil have $O\left((|\mathcal{O}_\Gamma| + |\mathcal{D}_\Gamma|)!\right)$ forskellige stier, hvorfor det er attraktivt at finde en mindre struktur. En art S-DAG, der kan være mindre end den nævnte S-DAG, og som samtidigt garanterer at indeholde den eller de optimale ordninger af variablene kaldes en *normalforms S-DAG*[†] (NFS-DAG).

Med udgangspunkt i to observationer omkring nytte og beregning af denne findes og defineres en række egenskaber i en S-DAG. Disse egenskaber danner baggrund for definitionen af en NFS-DAG, der netop overholder tidligere nævnte ønskværdige egenskaber. De indledende observationer er som følger:

1. Den forventede nytte kan aldrig stige ved at udsætte en observation. Altså bør alle observationer observeres så snart at de er blevet frigivet.

2. Rækkefølgen af variable af samme type har ingen betydning for den forventede nytte. Rækkefølgen af sådanne variable kan ombyttes, med

[†]Også kendt som en GS-DAG eller en "General Solution DAG" i [Jensen et al., 2006b].

mindre en præcedenskant eksplicit specificerer andet, da summations-operatoren for chancevariable, og maksimations-operatoren for beslutningsvariable, begge er kommutative på endelige variable – dog ikke indbyrdes.

Konsekvensen af disse observationer forklares ud fra S-DAG eksemplet i figur 2.3. Ifølge observation 1 burde observationen OS ikke blive udsat til efter DD i det øverste scenarie, da OS bliver frigivet allerede efter at beslutningen DS er blevet besluttet.

På grund af observation 2 kan variable af samme type, der besluttes eller observeres umiddelbart efter hinanden, betragtes som en *mængde* af variable, der kan løses i arbitrær rækkefølge, fremfor at være bundet af en stærk elimineringsrækkefølge.

I figur 2.3 kan variablene DS og DD samt variablene OS og OD i det øverste scenarie således slås sammen til mængde-knuderne $\{DS, DD\}$ og $\{OS, OD\}$, uden at dette scenaries forventede nytte vil ændres. Dette er dog ikke aktuelt, idet det øverste scenarie bliver fjernet pga. den rækkefølge i hvilken observationerne bliver frigivet; det illustrerer dog stadig en vigtig pointe.

En S-DAG bruges, som nævnt, til at repræsentere tilladelige rækkefølger af beslutninger i et UID. Alt afhængig af UIDet kan dette være en stor mængde af rækkefølger, hvorfor S-DAGen kan blive stor. Det kan derfor være ønskværdigt at minimere antallet af tilladelige rækkefølger uden at fjerne en potentiel optimal sti. En S-DAG indeholder ikke nødvendigvis alle tilladelige stier for et UID, men snarere en *mængde* af tilladelige rækkefølger, som defineret i definition 2.3.

Definition 2.3 (**Tilladelige rækkefølger i en S-DAG**). Mængden af *tilladelige rækkefølger* indeholdt i en S-DAG, Ξ, er mængden af sekvenser af variable der kan opnås ved at følge orienterede stier fra *Source* til *Sink* i Ξ. □

Figur 2.3 viser således et eksempel på en S-DAG med 3 forskellige tilladelige rækkefølger for UIDet på figur 2.1.

To tilladelige rækkefølger giver den samme forventede nytte, hvis de kun adskiller sig fra hinanden mht. rækkefølgen af variable af samme type. Sådanne tilladelige rækkefølger kaldes ækvivalente, og vi definerer mængden indeholdende alle ækvivalente rækkefølger som en ækvivalensklasse.

Slås variable i S-DAGen sammen i overensstemmelse med de tidligere nævnte regler, kan antallet af stier i S-DAGen minimeres. En sådan S-DAG vil repræsentere ækvivalensklasser frem for egentlige tilladelige ordninger.

Hvis alle variable i en S-DAG er slået sammen, så er det ikke længere muligt at minimere S-DAGen ud fra overvejelser omkring observation 2. En

sådan S-DAG kaldes en minimal S-DAG og er formelt defineret i definition 2.4.

Definition 2.4 (**Minimal S-DAG**). En S-DAG er *minimal*, hvis ingen på hinanden følgende knuder kan blive lagt sammen uden at mængden af tilladelige rækkefølger bliver ændret.

□

Eksempelvis er S-DAGen på figur 2.3 ikke minimal, da variablene DD og DS i det øverste scenarie kan lægges sammen til én knude i S-DAGen (jf. observation 2), hvorved der opstår en ny tilladelig rækkefølge i S-DAGen, hvor DD kommer før DS, og mængden af tilladelige rækkefølger bliver ændret.

Første observation siger, at en beslutningstager intet har at vinde ved at udsætte observationer. Det betyder, at stier gennem S-DAGen, hvor der forekommer observationer, der kunne have været inkluderet tidligere på stien, kan ændres, så disse observationer observeres tidligere, uden at den forventede nytte for det nye scenarie vil være lavere end for det originale scenarie. Det leder frem til et koncept af en *fejlplaceret* observationsvariabel, som defineret i definition 2.5

Definition 2.5 (**Fejlplaceret observationsvariabel**). Lad κ være en ækvivalensklasse af tilladelige ordninger for et UID, Γ. En observationsvariabel O siges at være *fejlplaceret*, hvis den kommer umiddelbart efter en beslutning D i κ uden at den temporale ordning for Γ, $\prec$, foreskriver at O skal komme efter D - dvs. $D \not\prec O$.

□

Som eksempel er variablen OS i figur 2.3 fejlplaceret ift. DD, da den temporale ordning på figur 2.2 ikke forskriver, at OS *skal* komme efter DD.

På baggrund af disse overvejelser er det nu muligt at opstille en S-DAG, der er garanteret at være tilpas stor til at repræsentere alle de temporale ordninger, der kan være aktuelle for den optimale strategi. Dette er samtidigt vores NFS-DAG (definition 2.6).

Definition 2.6 (**NFS-DAG**). Lad Γ være et UID. En *normalforms S-DAG* for Γ er en minimal S-DAG med samtlige tilladelige ordninger på variablenes interne rækkefølger og ingen fejlplacerede observationsvariable.

□

Det gælder således, at den forventede nytte for en NFS-DAG for et UID, Γ, aldrig kan være lavere end den forventede nytte for en arbitrær S-DAG for Γ. Løsningen til et UID kan altså være givet ved en NFS-DAG sammen med en optimal politik for hver knude i NFS-DAGen.

2.2.2 Relevant fortid

Ovenstående gennemgang definerer NFS-DAGen, som den beskrives i [Jensen et al., 2006b]. I [Jensen and Vomlelová, 2002] beskrives en udvidelse til kravene for NFS-DAGen, der kan reducere antallet af stier i NFS-DAGen yderligere ift. gennemgangen i afsnit 2.2.1.

Udtryk (2.2) kan reduceres ved at udskifte den samlede fortid for en beslutning D, $Past(D)$, med den relevante fortid for D, $Rel(D)$ [Shachter, 1999, Nielsen and Jensen, 1999].

Med den relevante fortid, $Rel(D)$, for beslutningen D, forstås de variable, der kan have indflydelse på politikken for D. For enhver variabel i $Past(D)\backslash Rel(D)$ gælder det, at de ikke kan have indflydelse på politikken for D. Mængden $Rel(D)$ udgør derfor den *maksimale* mængde af variable, der kan have indflydelse på politikken for D. Anvender vi $Rel(D)$ i stedet for $Past(D)$ i udtryk (2.2), får vi således

$$\begin{aligned} \rho_{D_k}(Rel(D_k)) &= \max_{D_k} \sum_{C_k} P(C_k|Rel(D_k), D_k) \cdot \\ &\quad \max\left\{\rho_{D^1_{k+1}}(Rel(D^1_{k+1})), \ldots, \rho_{D^b_{k+1}}(Rel(D^b_{k+1}))\right\}, \end{aligned} \tag{2.4}$$

I det følgende undersøger vi, hvordan den relevante fortid kan identificeres for alle beslutningerne i et UID med en given S-DAG. Til dette formål skal vi bruge begreberne d-separeret og d-forbundet, der er defineret i definition 2.7 fra [Pearl, 1986, Jensen, 2001]

Definition 2.7 (**d-separation**). To forskellige variable, A og B, i et kausalt net er d-separerede hvis, for alle stier mellem A og B, der er en mellemliggende variabel V (forskellig fra A og B) sådan at forbindelsen enten er

- seriel eller divergerende og V er instantieret, eller
- konvergerende og hverken V eller en af Vs efterfølgere har modtaget evidens.

Hvis A og B ikke er d-separerede, kalder vi dem d-forbundede. □

I vores gennemgang vil vi kun beskrive hvordan den relevante fortid kan identificeres ud fra strukturelle overvejelser ud fra UIDet og S-DAGen og ikke ud fra uafhængighed mellem d-forbundede variable.

Vi starter med at se, hvordan vi kan identificere den relevante fortid for den sidste beslutning i et ID. Til dette formål kan vi bruge følgende definition fra [Nielsen and Jensen, 1999]

Definition 2.8 (**Relevant fortid**). Lad D være den sidste beslutning i et influensdiagram, γ. Og lad $de(D)$ være efterfølgerne for D i γ. En variabel $A \in Past(D)$ er *relevant* for D, hvis der er mindst en nytteknude $U \in de(D)$, der er d-forbundet med A givet $Past(D) \backslash \{A\}$. □

For et ID, γ, er det muligt at finde den relevante fortid for alle beslutninger i γ ved at starte med den sidste beslutning, D, i den temporale ordning for γ, identificere $Rel(D)$ og udskifte beslutningsknuden D med en chanceknude med forældrene $Rel(D)$. Nu vil en ny beslutningsknude være den sidste beslutning. Dennes relevante fortid kan nu undersøges ud fra samme princip. Sådan kan man iterativt fortsætte baglæns gennem den temporale ordning indtil den relevante fortid er identificeret for alle beslutningerne i γ.

I IDet på figur 1.3 på side 12 er DD den sidste beslutning. Nytteknuderne, der er efterfølgere til DD, er $\{U, UD\}$. Fortiden for DD er $Past(DD) = \{OK, DS, OS\}$. Og den relevante fortid for DD er $Rel(DD) = \{OK, OS\}$. OK og OS er relevante, fordi U er direkte afhængig af disse variable, og DS er irrelevant for DD, da DS er d-separeret fra $\{U, UD\}$ givet evidens på $Past(DD) \backslash \{DS\} = \{OK, OS\}$. Hvis DD nu udskiftes til en chanceknude med $\{OK, OS\}$ som forældre, så kan vi identificere den relevante fortid for DS til $Rel(DS) = \{OK\}$.

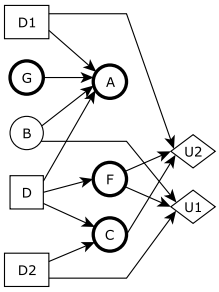
Bemærk at den relevante fortid for en beslutningsknude, D, kan komme til at afhænge af den relevante fortid for de beslutningsknuder, der kommer efter D i den temporale ordning. Dette grunder i at udledningen af den relevante fortid for D tager udgangspunkt i strukturen af den graf, der er dannet ved at udskifte beslutningsknuderne, der følger D, med chanceknuder og disse kan have andre forældre end de oprindelige beslutningsknuder.

For et UID er opgaven med at finde den relevante fortid for beslutningerne i S-DAGen ikke lige så simpel som for et ID. Vi kan stadig iterativt finde den relevante fortid for beslutningerne baglæns gennem S-DAGen – der repræsenterer den del af den temporale ordning, vi ønsker at undersøge. Men da der i en S-DAG kan være *flere forskellige stier* fra en beslutning D til $Sink$, kan vi få flere forskellige bud på den relevante fortid for D. Hvis der fra D er n forskellige stier til $Sink$, kan vi få n bud. Lad disse bud være $Rel_1(D), Rel_2(D), \ldots, Rel_n(D)$. Den relevante fortid for D er da $Rel_1(D) \cup Rel_2(D) \cup \ldots \cup Rel_n(D)$.

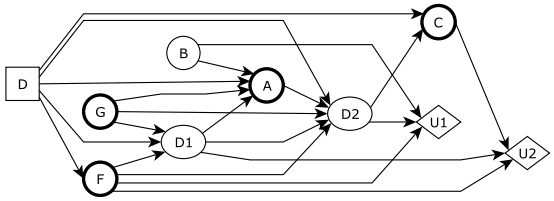
Figur 2.5(a) viser et eksempel på et UID, og figur 2.5(c) viser skelettet for en S-DAG for dette UID. I skelettet for en S-DAG er alle chanceknuder fjernet, og alle beslutninger er i stedet direkte forbundet, hvis de i S-DAGen er forbundet direkte eller via en eller flere chanceknuder. Dette skelet indholder ordningerne $D \prec D1 \prec D2$ og $D \prec D2 \prec D1$. I dette skelet er den relevante fortid for beslutningen D forskellig afhængigt af, om D efterfølges af først

$D1$ og så $D2$, eller om D efterfølges af $D2$ og så $D1$.

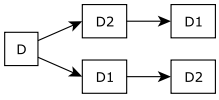
Når D efterfølges af først $D1$ og så $D2$, så er den relevante fortid for D $Rel(D) = \{G\}$. Dette kan ses i figur 2.5(b), hvor vi først har konverteret $D2$ og så $D1$ til chanceknuder, og givet disse chanceknuder knuderne i beslutningernes relevante fortid som forældre. Hvis D først efterfølges af $D2$ og så $D1$, bliver den relevante fortid for D lig $Rel(D) = \emptyset$. Dette kan ses i figur 2.5(d). Samlet set er den relevante fortid for D altså $Rel(D) = \{G\} \cup \emptyset = \{G\}$.



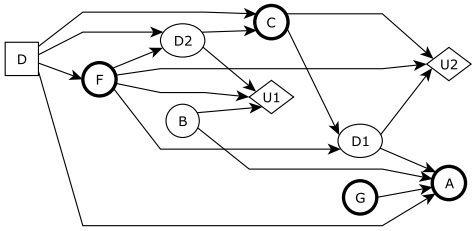
(a) Et UID.



(b) UIDet fra 2.5(a) hvor først $D2$ og så $D1$ er blevet konverteret til chanceknuder og har fået de variable som forældre, der var relevante for beslutningerne. I UIDet er den relevante fortid for D lig $Rel(D) = \{G\}$.



(c) Skelettet for en S-DAG for UIDet på figur 2.5(a).



(d) UIDet fra 2.5(a), hvor først $D1$ og så $D2$ er blevet konverteret til chanceknuder. I UIDet er den relevante fortid for D lig $Rel(D) = \emptyset$.

Figur 2.5: Et eksempel på, hvordan den relevante fortid for D kan afhænge af ordningen af beslutningerne i fremtiden for D.

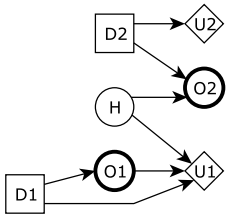
Reducering af NFS-DAGen Hvis vi har en NFS-DAG med stierne $D1 \prec O1 \prec D2 \prec O2$ og $D2 \prec O2 \prec D1 \prec O1$, og vi når frem til, at $\{D1, O1\}$ er irrelevant for $D2$, så kan vi undlade den første sti i NFS-DAGen, da den aldrig kan biddrage med højere forventede nytte end den anden sti. Vi siger at beslutningsvariablen, $D1$, er fejlplaceret ift. $D2$.

Definition 2.9 (**Fejlplacerede beslutningsvariable**). Lad D og D' være to beslutninger i et UID, Γ, og lad $\mathcal{O}'$ være de observationsvariable, der frigives af D'. Lad κ være en ækvivalensklasse af tilladelige ordninger for Γ. D' siges

at være *fejlplaceret*, hvis den optræder før beslutning D i κ og det gælder at $\{D'\} \cup \mathcal{O}'$ er irrelevante for D. □

Således gælder det, at hvis $\{D'\} \cup \mathcal{O}'$ er irrelevant for D, så vil der ikke tilvejebringes ny information for beslutningen D ved at træffe beslutningen D' før D. Derfor kan en sti i en S-DAG, hvor D' optræder før D undlades til fordel for en sti, hvor D optræder før D'.

Hvis D heller ikke frigiver nogle observationsvariable, der er relevante for D', så kan en vilkårlig ordning på D og D' vælges, da de forventede nytter for ordningerne vil være identiske. Det gælder dog ikke altid symmetrisk, at hvis D og de observationsvariable, som D frigiver er irrelevante for D' så er D' og de observationsvariable, som D' frigiver, irrelevante for D. Et eksempel på dette kan ses i figur 2.6, hvor $D2$ og $O2$ er relevante for $D1$, mens $D1$ og $O1$ er irrelevante for $D2$.



Figur 2.6: Et UID, hvor $D2$ og $O2$ er relevante for $D1$ mens $D1$ og $O1$ er irrelevante for $D2$. En S-DAG, der kun indeholder stien $D2 \prec O2 \prec D1 \prec O1$, vil være garanteret at kunne repræsentere den optimale løsning til UIDet.

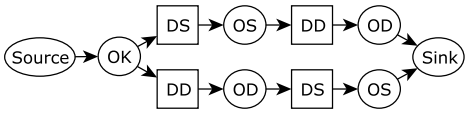
Dette leder os frem til definitionen af en S-DAG på anden normalform (2NFS-DAG)

Definition 2.10 (**2NFS-DAG**). Lad Γ være et UID. En *anden normalforms S-DAG for* Γ er en S-DAG på (første) normalform, som defineret i definition 2.6, uden fejlplacerede beslutningsvariable. Hvis en mængde beslutninger vil være gensidigt fejlplacerede, er en vilkårlig ordning af disse valgt. □

I [Ahlmann-Ohlsen and Pedersen, 2005] præsenterer vi en algoritme til generering af S-DAGe på første normalform. Denne algoritme kan forholdsvis simpelt udvides til også at opstille S-DAGe på anden normalform.

Figur 2.7 på næste side viser en 2NFS-DAG for UIDet for castingproblemet fra figur 2.1 på side 22.

I resten af denne rapport vil vi ikke arbejde med første normalform af S-DAGe men derimod anden normalform, da den giver mindre S-DAGe, der er hurtigere at evaluere. Derfor vil vi fremover i denne rapport lade to-tallet foran forkortelsen "NFS" være implicit.



Figur 2.7: En 2NFS-DAG for castingproblemet på figur 2.1.

2.3 Løsning af en S-DAG

En løsning til et UID består af en optimal strategi, givet som en S-DAG med en politik for hver knude. At løse en S-DAG (herunder en NFS-DAG) består således i at finde politikkerne for knuderne i S-DAGen.

En S-DAG kan løses på en måde, der minder om variabeleliminationsmetoden fra IDer. Der arbejdes igen med mængderne af sandsynlighedspotentialer, Φ, og nyttepotentialer, Ψ, og der elimineres igen modsat den temporale ordning – dvs. baglæns op gennem stierne i S-DAGen. Da de uobserverbare variable altid er placeret sidst i den temporale ordning, kan man opfatte dem som implicit repræsenteret på knuden *Sink* i enhver S-DAG. Når vi eliminerer baglæns gennem stierne i en S-DAG, elimineres således først de uobserverbare variable. Derefter elimineres de øvrige variable op til *Source*. Ved forgreningspunkterne laves der en kopi af Φ og Ψ, og der regnes uafhængigt op gennem hver forgrening. Ved samlingspunkterne vil sandsynlighedspotentialerne være de samme, mens nyttepotentialerne forenes med maksimationsoperatoren.

Under elimineringen af variablene i S-DAGen kan politikkerne for knuderne i S-DAGen opstilles, som angivet i udtryk (2.1) og udtryk (2.3).

Når en variabel N skal elimineres, skal potentialemængderne modificeres. Måden hvorpå mængderne skal modificeres er i princippet den samme, som ved variabelelimineringsalgoritmen for IDer. Dvs. at operationerne angivet i udtryk (1.7) - (1.14) på side 17 i princippet blot kan genbruges.

Det gælder dog, at når beslutningsvariable skal elimineres, så kan ingen sandsynlighedstabeller afhænge af beslutningen [Vomlelova and Jensen, 2002]. Derfor kan vi optimere algoritmen ved at udskifte udtryk (1.11) med

$$\Phi'_N \leftarrow \{max_N \phi \mid \phi \in \Phi_N\}. \tag{2.5}$$

Operationen i udtryk (2.5) fjerner N fra de sandsynlighedspotentialer, der skulle indeholde N. Da disse potentialer i forvejen udtrykker uafhængighed af N, kan N fjernes ved fx at max-marginaliserer N ud af potentialerne.

Produktet (og summen) af to potentialer er ét potentiale med et domæne der er lig foreningsmængden af domænerne for udgangspotentialerne. Således kan udtryk (2.5) gøre, at algoritmen kommer til at arbejde med mindre

potentialer, end hvis den havde taget produktet af alle sandsynlighedspotentialer som i udtryk (1.11).

Da N er uafhængig af alle sandsynlighedspotentialer, kan vi ligeledes udskifte udtryk (1.12) med

$$\psi'_N \leftarrow max_N \left(\sum \Psi_N \right). \tag{2.6}$$

Når N er en beslutningsvariabel, kan vi altså i stedet for at opdatere potentialemængderne med udtryk (1.13) - (1.14) opdatere potentialemængderne med

$$\Phi \leftarrow (\Phi \backslash \Phi_N) \cup \Phi'_N \tag{2.7}$$
$$\Psi \leftarrow (\Psi \backslash \Psi_N) \cup \{\psi'_N\}. \tag{2.8}$$

Vi kan nu eliminere variable op gennem grenene i en S-DAG. Ved forgreningspunkter laves kopier af Φ og Ψ, og ved samlingspunkter forenes potentialerne i Ψ ved max-kombination som i

$$\Psi \leftarrow \left\{ \max \left(\sum \Psi^1, \ldots, \sum \Psi^n \right) \right\}, \tag{2.9}$$

hvor n er antallet af grene der samles ved en given knude i S-DAGen, og Ψ^i er Ψ for gren i.

For alle potentialer $\Psi' \subseteq \Psi^i$, der er ens for alle grenene, kan man anvende den distributive lov [Jensen, 2001] og trække Ψ' udenfor max-kombineringen, så vi får

$$\Psi \leftarrow \left\{ \max \left(\sum (\Psi^1 \backslash \Psi'), \ldots, \sum (\Psi^n \backslash \Psi') \right) \right\} \cup \Psi'. \tag{2.10}$$

Sandsynlighedspotentialemængderne, Φ^i, i de forskellige grene ved samlingspunkterne kan ligeledes forenes med max-kombineringen, men dette er ikke nødvendigt, da sandsynlighedspotentialerne i de forskellige grene altid vil være de samme, når de skal kombineres. Dette er et resultat af, at det gælder at knuderne ved ethvert samlingspunkt (og forgreningspunkt) i S-DAGen har de samme variable i deres fremtid, og da alle variable i fremtiden ved samlingsstederne er fjernet ved sum-marginaliseringen i udtryk (1.9) eller minimeringen i udtryk (2.5), og disse to operationer kommuterer i disse situationer, betyder rækkefølgen af operationernes udførelse intet, og Φ^i for de forskellige stier vil således være ens.

2.3.1 Eksempel

Lad os anvende disse formler til at løse NFS-DAGen på figur 2.7 på side 34. Vi starter med mængderne

$$\begin{aligned}\Phi &= \{P(OK), P(T), P(OS|T, DS), P(OD|T, DD)\}\\ \Psi &= \{U(OK, OS, OD), US(DS), UD(DD)\},\end{aligned}$$

og eliminerer først den uobserverbare variabel T ud, og får

$$\begin{aligned}\Phi^{-T} &= \{P(OK), P(OS, OD|DS, DD)\}\\ \Psi^{-T} &= \Psi,\end{aligned}$$

hvor

$$P(OS, OD|DS, DD) = \sum_T P(T)P(OS|T, DS)P(OD|T, DD).$$

Nu arbejdes der videre med Φ^{-T} og Ψ^{-T} gennem de to grene. Den øverste gren, der slutter med OD, kalder vi OD-grenen, og den nederste gren, der slutter med OS, kalder vi OS-grenen.

I OD-grenen elimineres nu OD, og vi får

$$\begin{aligned}\Phi_{OD}^{-OD} &= \{P(OK), P(OS|DS, DD)\}\\ \Psi_{OD}^{-OD} &= \{U'_{OD}(OK, OS, DS, DD), US(DS), UD(DD)\},\end{aligned}$$

hvor

$$\begin{aligned}P(OS|DS, DD) &= \sum_{OD} P(OS, OD|DS, DD)\\ U'_{OD}(OK, OS, DS, DD) &= \frac{\sum_{OD} P(OS, OD|DS, DD)U(OK, OS, OD)}{P(OS|DS, DD)}.\end{aligned}$$

Nu elimineres DD, hvilket giver

$$\begin{aligned}\Phi_{OD}^{-DD} &= \{P(OK), P(OS|DS)\}\\ \Psi_{OD}^{-DD} &= \{U''_{OD}(OK, OS, DS), US(DS)\},\end{aligned}$$

hvor

$$\begin{aligned}P(OS|DS) &= max_{DD} P(OS|DS, DD)\\ U''_{OD}(OK, OS, DS) &= max_{DD}\left[U'_{OD}(OK, OS, DS, DD) + UD(DD)\right].\end{aligned}$$

Her kan vi finde valgfunktionen for knuden DD i OD-grenen, som

$$\delta(OK, OS, DS) = \quad argmax_{DD}\left[U'_{OD}(OK, OS, DS, DD) + UD(DD)\right]. \tag{2.11}$$

Sådan fortsættes elimineringen af alle variablene i de to grene. Når DS er blevet elimineret ud af OD-grenen, og DD er blevet elimineret ud af OS-grenen, har vi mængderne

$$\begin{aligned}
\Phi_{OD}^{-DS} &= \{P(OK)\} \\
\Psi_{OD}^{-DS} &= \{U''''_{OD}(OK)\} \\
\Phi_{OS}^{-DD} &= \{P(OK)\} \\
\Psi_{OS}^{-DD} &= \{U''''_{OS}(OK)\}.
\end{aligned}$$

Som omtalt sidst i afsnit 2.3 er sandsynlighedspotentialerne de samme, og når OK skal elimineres, regnes der med mængderne

$$\begin{aligned}
\Phi' &= \Phi_{OD}^{-DS} = \Phi_{OS}^{-DD} = \{P(OK)\} \\
\Psi' &= \{max[U''''_{OS}(OK), U''''_{OD}(OK)]\}.
\end{aligned}$$

Samtidigt kan vi bestemme skridtpolitikken for OK som

$$\sigma(OK) = \begin{cases} DS & \text{hvis } U''''_{OD}(OK) \geq U''''_{OS}(OK) \\ DD & \text{ellers.} \end{cases} \tag{2.12}$$

Altså vælger vi grenen, der starter med DS (OD-grenen), hvis vi kan forvente den største nytte via denne gren givet den observerede tilstand for OK, ellers vælges den anden gren.

Sådan kan man fortsætte og finde optimale politikker for alle knuder i NFS-DAGen.

Skridt-politikken for alle knuder N med netop ét barn er triviel – det er blot en funktion, der, ligemeget hvilken fortid der er for N, altid går over i barnet til N. Skridt-politikken for knuder med flere børn kan findes som eksemplificeret i udtryk (2.12). Valgfunktionen for knuder med en beslutning kan findes som eksemplificeret i udtryk (2.11).

En løsning til UIDet på figur 2.1 er altså givet ved NFS-DAGen på figur 2.7 sammen med en optimal politik for hver knude, fundet som eksemplificeret i ovenstående eksempel.

2.4 Tids- og pladskompleksitet

Ved løsning af UIDer med variabelelimineringsalgoritmen, VE algoritmen, kan det blive et pladsproblem, at repræsentere potentialerne, der optræder i potentialemængderne Φ og Ψ samt politikkerne for knuderne i S-DAGen.

Problemet optræder, når vi tager summen eller produktet over en mængde af potentialer; her bliver størrelsen af det resulterende potentiale lig med størrelsen af det kartesiske produkt over foreningsmængden af variablene i de oprindelige potentialers domæner. Hvis potentialet skal repræsenteres som en tabel, vil tabellen indeholde én indgang per indgang i potentialet.

Det betyder, at hvis én variabel X indgår i domænet for mange potentialer, og X skal elimineres, kan det blive et problem at udføre denne operation, da repræsentationen af det mellemliggende potentiale kan blive så stor, at der ikke er plads til repræsentationen i den hukommelse, der er tilgænglig.

Herudover vokser politikkernes størrelse eksponentielt med den relevante fortid for beslutningerne. Da løsningen til et UID består af en politik for hver knude i S-DAGen, kan det blive et problem at repræsentere denne løsning.

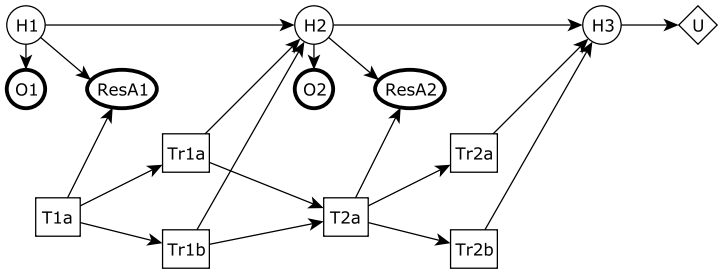
En øvre grænse for VE algoritmens pladskompleksitet er $O(n \cdot exp(w^*))$, hvor n er antallet af knuder i S-DAGen og w^* er antallet af variable i det største potentiale, der arbejdes med under løsningen af S-DAGen med VE algoritmen. w^* er ligeledes kendt som *bredden af en elimineringsrækkefølge* [Dechter, 1996].

En øvre grænse for Tidskompleksiteten for VE algoritmen er $O(n \cdot |nChildren|^* \cdot exp(w^*))$, der udtrykker, tiden det tager, at udføre max-kombinering over $O(|nChildren|^*)$ forskellige potentialer, der hver har en størrelse givet ved $O(exp(w^*))$. Her er $|nChildren|^*$ det maksimale antal børn, som en knude i S-DAGen har. Denne operation skal udføres for alle n knuder i S-DAGen, hvorfor tidskompleksiteten bliver som netop angivet.

Resultaterne for tids- og pladskompleksiteterne er afhængige af antallet af knuder i S-DAG strukturen. I løsningsmetoden beskrevet i afsnit 2.2.1 på side 27 svarer dette til antallet af knuder i NFS-DAGen. Der er to ekstremer for størrelsen af NFS-DAGen; (1) den indeholder kun én sti og antallet af knuder bliver derfor ikke større end i det oprindelige UID, og (2) den indeholder stier for samtlige mulige ordninger af beslutningsvariablene. Det første ekstrem kaldes en *best case NFS-DAG*, mens det sidstnævnte ekstrem kaldes en *worst case NFS-DAG*. I [Ahlmann-Ohlsen and Pedersen, 2005] viste vi, at antallet af knuder i en worst case NFS-DAG vokser eksponentielt med antallet af beslutninger i det tilhørende UID.

Figur 2.8 viser et eksempel på et UID, der giver anledning til en best case NFS-DAG, da NFS-DAGen for dette UID kun indeholder én sti. Denne NFS-DAG kan ses i bilag A i figur A.2 på side 150.

I figur 2.9 er der afbilledet et eksempel på et UID, der giver anledning til en worst case NFS-DAG. Dette er tilfældet, da der ikke er angivet nogen ordning af beslutningerne og alle beslutningerne frigiver hver deres variabel, hvorfor alle rækkefølger af beslutningerne skal optræde i S-DAGen på første



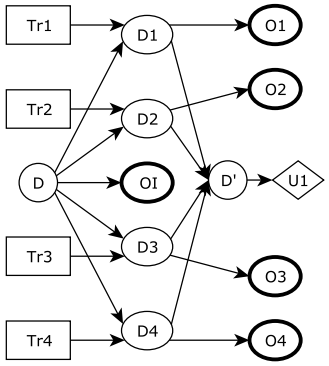
Figur 2.8: Et UID, der modellerer et beslutningsproblem, der indeholder periodiske inspektioner, test og behandlinger af fx et dyr. $H1$, $H2$ og $H3$ er dyrets faktiske helbredstilstand, og $O1$ og $O2$ er den umiddelbart observerbare helbredstilstand. Testene $T1a$ og $T2a$ kan i kraft af testresultaterne $ResA1$ og $ResA2$ yderligere være med til at afdække den faktiske helbredstilstand, mens behandlingerne $Tr1a \ldots Tr2b$ kan påvirke helbredstilstanden ved den efterfølgende inspektion. U er nytten af at have et dyr med helbredstilstanden $H3$ efter de sidste behandlinger. I bilag B beskriver vi et eksempel på en situation, hvor dette UID kunne anvendes.

normalform. Yderligere er de frigivne variable relevante for de andre beslutninger, hvorfor ingen stier i NFS-DAGen bliver reduceret væk ved anden normalform. NFS-DAGen for UIDet på figur 2.9 kan ses i bilag A i figur A.1 på side 149.

En anden egenskab ved de to UIDer er, at den relevante fortid for hver beslutningsknude i begge NFS-DAGer svarer til hele fortiden.

I kraft af det eksponentielt store antal knuder i en worst case NFS-DAG, kan man ved løsning af UIDer komme ud for at skulle repræsentere eksponentielt mange eksponentielt store politikker – hvilket kan give pladsproblemer selv på moderne computere.

I det ovenstående afsnit er det blevet klargjort at den eksisterende løsningmetode til UIDer, VE algoritmen, lider under en pladskompleksitet, med eksponentielt mange knuder i løsningsgrafen, samt eksponentielt store politikker for hver af disse knuder. Selvom størrelsen af en NFS-DAG udgør et kompleksitetsmæssigt problem, opstår de største kompleksitetsmæssige udfordringer dog ved beregningen og repræsentationen af de mange politikker såvel som de mellemliggende potentialer, der fremkommer ved løsning af et UID med VE algoritmen.



Figur 2.9: Et UID for en sygdomsbehandlingssituation hvor der er mulighed for at foretage fire forskellige behandlinger af en given sygdom, D. Alle behandlinger $Tr1 \ldots 4$ kan udføres i forsøg på at ændre sygdommens tilstand. De ændrede tilstande er repræsenteret ved $D1 \ldots 4$, mens D' er sygdommens tilstand, efter at der er besluttet, hvorvidt hver enkelt behandling skulle udføres, og de pågældende behandlinger er gennemført. Nytten af denne sygdomstilstand er givet ved $U1$. OI er en observation, som kan laves over sygdommens symptomer. $O1 \ldots 4$ er de symptomer, der kan observeres, efter eventuelle behandlinger er gennemført. Bemærk at der ikke er nogle temporale bindinger på beslutningsvariablene. Eksemplet er baseret på en lignende situation beskrevet i [Jensen et al., 2006b].

2.5 Opsummering og delkonklusion

Vi har indtil nu introduceret sprog til beskrivelse af beslutningsproblemer. Beskrivelsen af de forskellige sprog tog udgangspunkt i beslutningstræer og fortsatte med IDer (kapitel 1), der er et sprog til kompakt repræsentation af symmetriske beslutningsproblemer, men som alligevel ikke var begrænset til denne anvendelse. IDerne havde nogen repræsentationsmæssige begrænsninger, når man ville modellere asymmetriske beslutningsproblemer. Her blev UIDer indført som en mulig løsning til repræsentationen af ordningsasymmetriske beslutningsproblemer. Problemet ved IDer var, at der skulle specificeres en fuldstændig ordning af alle beslutninger. Dette krav blev ophævet ved UIDerne.

UIDer er introduceret med udgangspunkt i IDer og i kraft af dette har begge sprog lignende løsningsmetoder, omend metoden for UIDer er mere kompleks. Ligesom IDer løses ved eliminering af variablene i en stærk elimineringsrækkefølge dvs. modsat den temporale ordning af variablene, løses et UID ved eliminering af variable i fx en NFS-DAG også modsat den temporale ordning. NFS-DAGen kunne ses som en kompakt repræsentation af en række kandidat-IDer, hvorudfra den optimale ordning af beslutninger skulle findes.

I afsnit 2.4 beskrev vi flere kompleksitetsmæssige problemer ved løsning

af UIDer, der bl.a. inkluderede at en NFS-DAG for et givet UID kan blive eksponentielt stor i antallet af beslutningsvariable, ligesom politikkerne for knuderne i NFS-DAGen kan blive eksponentielt store i antallet af variable i deres relevante fortid.

Blandt disse problemer vurderer vi, at størrelsen af NFS-DAGen er underordnet i forhold til repræsentation af politikker samt de mellemliggende potentialer, der opstår som følge af eliminering af variable i NFS-DAGen med VE algoritmen. Det kan som følge heraf hurtigt blive et problem at løse UIDer eksakt med den gængse VE algoritme. En aftestning i praksis, som vi har udført, viste at vi på en testmaskine (der er beskrevet i bilag C på side 153) ikke var istand til at løse et worst case UID med mere end 7 beslutninger, idet vi løb tør for hukommelse.

Uheldigvis er de domæner, der modelleres som beslutningsproblemer, sjældent begrænset til 7 beslutninger, og det er let at forestille sig beslutningsproblemer af worst case typen med endnu flere beslutninger. Et eksempel er worst case UIDet i figur 2.9, der let skaleres til at inkludere et vilkårligt antal beslutninger.

Ét problem med UIDer er, at de i kraft af tidskompleksiteten af løsningsalgoritmen kan tage lang tid at løse. Dette problem behandlede vi i [Ahlmann-Ohlsen and Pedersen, 2005], hvor vi opstillede en række anytimeløsningsalgoritmer. Et andet problem er, at dét at løse UIDer kan kræve mere plads end der er til rådighed pga. pladskompleksiteten af de eksisterende løsningsalgoritmer. I modsætning til tid, er den tilgængelige plads i et computersystem altid *endelig*, hvilket vil sige, at den tilgængelige plads vil resultere i en skarp afgrænsning af hvor store UIDer, der kan løses og dermed også for anvendeligheden af UID sproget.

I denne rapport vil vi derfor søge at opstille en algoritme, der kan løse UIDer på begrænset plads.

Del II

Design

Kapitel 3

Design Overvejelser

Som det blev præciseret i afsnit 2.5, søges der med dette projekt at udvikle algoritmer til løsninger af UIDer, der kan eksekveres på begrænset plads. I dette kapitel vil vi præsentere vores generelle overvejelser i forbindelse med udvikling af sådanne løsningsalgoritmer. Disse overvejelser involverer krav til en algoritme såvel som mulige tilgangsvinkler til designet af en sådan.

I første afsnit introduceres således krav til algoritmer med en diskussion om hvorvidt det er nødvendigt at repræsentere alle politikker ved en løsning af et UID. I forlængelse heraf indføres begrebet *trinvis kompilering*, der danner rammen om algoritmer, der netop ikke genererer alle politikker.

Derudover vil vi, med udgangspunkt i forskellige teknikker til udførelse af sandsynlighedsinferens på begrænset plads i bayesianske net, afveje fordele og ulemper ved disse teknikker – ikke mindst i forhold til teknikkernes anvendelsesmuligheder indenfor løsning af UIDer. Dette gøres i forventning om, at eksisterende teknikker vil rumme ideer og antagelser, der med fordel kan fungere som udgangspunkt for udviklingen af en løsningsalgoritme til UIDer – en løsningsalgoritme, der kan eksekveres med begrænsede hukommelsesressourcer.

Slutteligt vælges den designstrategi, der skal fungere som fundament for udviklingen af vores løsningsalgoritme.

3.1 Trinvis kompilering

Af pladsproblemer forbundet med UIDer, beskrevet i afsnit 2.4, havde vi bl.a., at NFS-DAGen kunne blive eksponentielt stor i antallet af beslutninger i et

UID, og at politikkerne til knuderne i NFS-DAGen kunne blive ekspotentielt store i den relevante fortid for knuderne i NFS-DAGen. Da løsningen af et UID med den eksisterende løsningsmetode består af en NFS-DAG med en politik for hver knude, betyder det, som tidligere nævnt, at der kan forekomme eksponentielt mange eksponentielt store politikker i løsningen af et UID. Det kan derfor blive et problem at repræsentere samtlige politikker, der indgår i løsningen af et UID, i hukommelsen på selv en moderne computer.

For at en beslutningstager skal kunne handle optimalt i et beslutningsscenarie, kræver det i første omgang, at han ved hvilken beslutning han *først* skal tage stilling til og hvilken *instans* af denne han skal vælge. Hvis der ikke er plads i hukommelsen til at repræsentere den optimale strategi for et UID, forventer vi at en bruger, der skal handle ud fra UIDet, vil kunne have gavn af et system, der blot kan vejlede ham om den *næste* beslutning i det aktuelle scenarie, som brugeren befinder sig i.

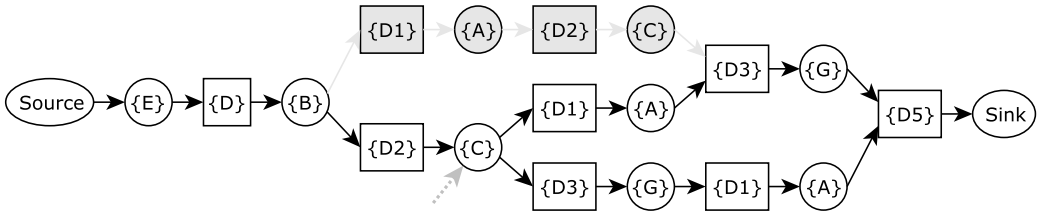
Således kan brugeren løbende interagere med systemet og indtaste, hvad der besluttes, og hvad der observeres. Disse oplysninger kan bruges til at begrænse problemet og reducere beregningen af en løsning til det resterende beslutningsproblem. Dette nye problem indeholder udelukkende de scenarier, der begynder med den sekvens af beslutninger og observationer, som allerede er gennemgået.

En fordel ved at anvende denne trinvise tilgang til løsning af UIDer er, at det ved løsningen af det efterfølgende trin bliver muligt at begrænse de krævede beregninger til kun at omfatte de tilstande, der er aktuelle for det pågældende scenarie. Det betyder, at nogle stier i NFS-DAGen bliver afskåret og dermed irrelevante for den løsningsstrategi, der skal beregnes, og som beslutningstageren følger. Det vil medføre, at størrelsen af de politikker, der skal opstilles, kan reduceres til kun at omhandle de næste valg i det pågældende scenarie.

Efterhånden som beslutningsscenariet udspiller sig, bliver flere stier i S-DAGen skåret fra. I figur 3.1 har en beslutningstager gennemgået en række beslutninger, og han har videregivet information om sine beslutninger og observationer til sit system. Brugeren er i diagrammet på vej til at observere chanceknude C, indikeret ved den stiplede pil. I kraft heraf er det ikke længere relevant at kunne følge det scenarie, der indeholder sekvensen $D1 \prec A \prec D2 \prec C$, og hele den sti kan således ignoreres, når den næste politik skal beregnes.

Ideen med at opstille politikker, efterhånden som beslutningsscenariet udspiller sig, kalder vi *trinvis kompilering*, idet de forskellige politikker bliver opstillet i flere trin.

Ved trinvis kompilering ønsker vi altså ikke at opstille politikker for alle



Figur 3.1: NFS-DAG hvor beslutningsscenariet er nået til knuden C, indikeret ved den stiplede pil, derved gøres scenariet der indeholder sekvensen $D1 \prec A \prec D2 \prec C$ overflødig.

knuderne i en S-DAG, men i stedet for starte med kun at opstille politikkerne for beslutningsknuderne i det første beslutningsniveau i S-DAGen – de beslutninger, som kan starte et beslutningsscenarie. De efterfølgende politikker defineres herefter kun for de variable, der endnu hverken er observerede eller besluttede. Med trinvis kompilering bliver det således muligt, at brugeren kan handle optimalt gennem et helt beslutningscenarie, selvom der ikke er plads til at repræsentere de optimale politikker for alle beslutningscenarier på brugerens system.

Ud over at trinvis kompilering stiller lavere pladskrav til repræsentationen af politikker, kan trinvis kompilering også give anledning til en reduktion af den S-DAG, der skal løses for at opstille politikkerne, når brugeren er undervejs i beslutningsscenariet.

Hvis en bruger skal kunne handle optimalt i et beslutningsscenarie, og hans system ikke tillader repræsentation af hele den optimale strategi, ser vi trinvis kompilering som et alternativ til at opstille den optimale strategi, da det stadig tillader brugeren at kunne handle optimalt.

3.2 Sandsynlighedsinferens på begrænset plads

Ved sandsynlighedsinferens forstås beregningsmetoder til at udlede svar på forespørgsler i et bayesiansk net. Et eksempel på en forespørgsel er at finde sandsynlighedsfordelingen til $P(A|\boldsymbol{e})$, hvor A er en variabel i det bayesianske net, og $\boldsymbol{e}$ er evidens. Sandsynlighedsinferens af denne type er NP-hårdt [Cooper, 1987].

Der er udviklet flere metoder til udførelse af sandsynlighedsinferens bl.a. algoritmer baseret på *Junction Trees* [Huand and Darwiche, 1996, Jensen, 2001], som fx Shenoy-Shafer [Shafer and Shenoy, 1990] og Lazy Propagation [Madsen and Jensen, 1999b].

Der findes også forskellige teknikker til at udføre sandsynlighedsinferens

på begrænset plads. Disse teknikker laver alle afvejninger mellem plads og tid eller præcision, hvilket resulterer i forskellige algoritmer, med hver deres fordele og ulemper. Vi kalder algoritmer, der kan løse et problem på en vilkårlig mængde plads, over et vist minimum, for anyspacealgoritmer.

I dette afsnit undersøger vi tre tilgangsvinkler til at udføre sandsynlighedsinferens på begrænset plads for at afdække tilgangsvinklernes respektive fordele og ulemper. Dette gøres med en forventning om at elementer fra de forskellige tilgange vil kunne overføres til UID paradigmet og anvendes ved løsning af UIDer på begrænset plads.

De tre tilgange til sandsynlighedsinferens, som vi beskæftiger os med i dette afsnit, er:

1. Konditionering.
2. Approksimerede potentialetræer.
3. Simulering.

3.2.1 Konditionering

Inden for eksakt løsning af bayesianske net på begrænset plads er der udviklet flere forskellige algoritmer, der udnytter *konditionering* til at komme med en eksakt løsning på bekostning af den tid, det tager at komme med løsningen [Pearl, 1986, Shachter et al., 1994, Darwiche, 2001, Darwiche, 2001, Dechter and Fattah, 2001, Bacchus et al., 2003].

Konditioneringsalgoritmerne kan betragtes som en klasse af del og hersk algoritmer, der reducerer et komplekst problem til mindre delproblemer. Disse delproblemer løses hver for sig, og deres løsninger kombineres til en løsning på det samlede problem.

Darwiche beskriver intuitionen bag konditionering i følgende citat, taget fra en generel introduktion til emnet i [Darwiche, 2000b]:

> "A very common form of human reasoning – which is dominant in mathematical proofs – is that of reasoning by cases or assumptions. To solve a complicated problem, we try to simplify it by considering a number of cases which correspond to a set of mutually exclusive and exhaustive assumptions. We then solve each of the cases under its corresponding assumption, and combine the results to obtain a solution to the original problem."

Herefter forklares, hvordan sandsynligheden $P(\mathbf{x})$ kan findes ved udtrykket

$$P(\mathbf{x}) = \sum_{\mathbf{c} \in \mathcal{C}} P(\mathbf{x}, \mathbf{c}), \tag{3.1}$$

hvor $\mathbf{x}$ og $\mathbf{c}$ er instantieringer over variablene $\mathcal{X}$ og $\mathcal{C}$. Her svarer $\mathbf{c}$ til vores antagelse, og ved at tage summen af $P(\mathbf{x}, \mathbf{c})$ for alle antagelser, $\mathbf{c} \in \mathcal{C}$, får vi sandsynligheden for $P(\mathbf{x})$. Denne tilgangsvinkel benyttes i Recursive Conditioning, der er et eksempel på en konditioneringsalgoritme [Darwiche, 2001, Allen and Darwiche, 2003a].

Fordelen ved konditioneringsalgoritmerne er, at de kan løse inferensproblemer eksakt med et pladsforbrug, der kun vokser lineært med størrelsen af det bayesianske net, hvilket står i stor kontrast til algoritmer som fx Junction Tree algoritmer. Disse algoritmer har en pladskompleksitet, der er eksponentiel i størrelsen af det bayesianske net.

Specielt for Recursive Conditioning gælder det, at der kan opnås en trinløs afvejning mellem tid og pladsforbrug – forstået på den måde, at algoritmen kan udnytte et vilkårligt lille pladslager (dog større end et vist minimum, der vokser linært med størrelsen af det bayesianske net), og jo mere plads, der er til rådighed for algoritmen, jo hurtigere vil algoritmen terminere. I grænsen hvor der er tilstrækkeligt med plads, vil algoritmens tidskompleksitet tilsvare dem for Junction Tree algoritmerne [Darwiche, 2001].

Således kan konditionering bruges til at løse problemer, der ikke var mulige at løse med fx Junction Tree algoritmerne, pga. Junction Tree algoritmernes eksponentielle pladskompleksitet.

De eksisterende konditioneringsalgoritmer kan ikke direkte overføres til UIDer, da de er udviklet til løsning af bayesianske net. Det vil kræve, at UIDerne først konverteres til bayesianske net, hvorefter algoritmerne kan anvendes. Konvertering af IDer til bayesianske net er behandlet i [Cooper, 1988, Zhang, 1998]. Det må forventes, at en lignende konvertering af UIDer vil være mulig. Denne tilgangsvinkel vil vi dog ikke behandle nærmere i denne rapport.

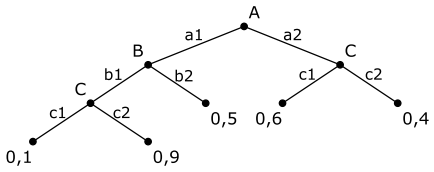
Selvom de eksisterende konditioneringsalgoritmer ikke umiddelbart kan anvendes til løsning af UIDer eller S-DAGe, kan ideen bag konditionering alligevel overføres til løsning af UIDer. Fx ved at tage udgangspunkt i det udtryk, der ligger til grund for politikken for første beslutning i S-DAGen, udtryk (2.1) på side 26. Dette udtryk kan reduceres med konditionering ved at betinge på variablene i udtrykket. De reducerede dele vil efterfølgende kunne løses hver for sig for sidenhen at blive kombineret til en politik for den første beslutning.

3.2.2 Approksimerede Potentialetræer

Ligesom med konditionering findes der indenfor eksakt og approksimativ inferens i bayesianske net flere eksempler på anvendelsen af approksimerede potentialetræer [Cano et al., 2000b, Cano et al., 2000a, Salmerón et al., 2000].

$P(C\|A,B)$		c_1	c_2
a_1	b_1	$0,1$	$0,9$
a_1	b_2	$0,5$	$0,5$
a_2	b_1	$0,6$	$0,4$
a_2	b_2	$0,6$	$0,4$

(a)



(b)

Figur 3.2: En betinget sandsynlighedstabel (a), der repræsenterer sandsynligheden $P(C|A,B)$, og dens tilsvarende potentialetræ (b). På potentialetræet er hver kant annoteret med den tilstand, den repræsenterer. Bemærk hvordan tilstanden for B er uden betydning når $A = a_2$, hvorfor B's grene kan slås sammen og B helt udelades fra det pågældende deltræ. Det samme er tilfældet for C når $A = a_1$ og $B = b_2$.

Lideledes er der eksempler på anvendelse af approksimerede potentialetræer indenfor løsning af IDer [Gómez and Cano, 2003].

Ideen med approksimerede potentialetræer er, som det fremgår af navnet, at repræsentere et potentiale som et *træ*. I dette træ repræsenterer interne knuder variable, kanter ud af en knude repræsenterer en tilstand for variablen i knuden, og blade i træet repræsenterer en værdi (fx en sandsynlighed eller en nytte), der knytter sig til den instantiering af variable, der fås ved at følge stien gennem træet fra træets rod knude ned til det aktuelle blad. Figur 3.2 illustrerer et potentiale både som tabel og som potentialetræ.

Ved at slå flere grene, i det eksakte potentialetræ, sammen til ét blad, der fx indeholder gennemsnitsværdien af de værdier, der lå i de fjernede grene, kan man opnå et approksimeret træ, der udtrykker et approksimeret potentiale på mindre plads end det tilsvarende eksakte potentialetræ. Fx kunne værdierne $0,6$ og $0,4$ i potentialet i figur 3.2(b) approksimeres til $0,5$, hvorefter kanten, hvor $A = a_2$, vil lede direkte til denne værdi.

I [Salmerón et al., 2000] beskrives, hvordan de basale operationer som fx kombinering og marginalisering kan udføres på potentialetræer – eksakte såvel som approksimerede.

Indenfor anvendelse på UIDer vil man kunne bruge eksakte potentialetræer til at repræsentere sandsynligheds- og nyttepotentialerne, og løse S-DAGen eksakt med disse potentialer, fx med VE. Hvis der ikke er plads til at gemme de eksakte træer, kan man i stedet for gemme approksimerede og dermed mindre potentialetræer. Hermed vil den udregnede løsning af UIDet blive en approksimation af den eksakte løsning.

Det vil i dette design være relevant bl.a. at overveje, hvor der bør anvendes eksakte potentialetræer, og hvor der bør bruges approksimerede potentialetræer, for at komme med den mest præcise løsning på den tilgængelige plads.

Man kan fx starte med at beregne hvor meget plads, der vil blive brug for ved en eksakt løsning, og hvis der ikke er plads nok, kan der lægges en plan for, hvor store potentialer man vil tillade på de forskellige knuder i S-DAGen.

Et anytime og anyspace aspekt kan opnås med approksimerede potentialetræer ved først at løse hele S-DAGen med meget små potentialetræer, hvilket hurtigt giver en (meget upræcis) approksimeret løsning til UIDet. Disse små approksimative potentialetræer kan udvides iterativt, hvilket vil give bedre og bedre løsninger. Sådan kan man fortsætte, indtil hele hukommelsen er brugt af potentialetræerne. Dette vil give en anytimealgoritme til løsning af UIDer, der vil kunne arbejde på begrænset plads og forbedre løsningen, hvis den afsatte plads tillader det.

Der vil i dette design kunne overvejes, hvor det er bedst at udvide potentialetræerne for at give gode løsninger hurtigt.

3.2.3 Simulering

En tredje metode til at udføre sandsynlighedsinferens med begrænset plads er stokastisk simulation – også kendt som sampling. Denne metode anvendes til at approksimere de ønskede sandsynligheder.

Ideen i stokastisk simulation er, at generere en mængde eksempler, samples, i overensstemmelse med de sandsynlighedspotentialer, der eksisterer i det bayesianske net. Dvs. at sammensætningen af samples, der genereres, ideelt set kommer til at afspejle fællessandsynlighedspotentialet for de variable, der samples over.

De ønskede sandsynligheder findes ved at tælle antallet af samples, der overholder den ønskede sandsynlighed. Dette kan gøres mens de forskellige samples genereres og det er derfor muligt at udføre inferens i et bayesiansk net, mens man blot gemmer tal, der repræsenterer antallet af de samples, der overholder den ønskede sandsynlighed, såvel som det samlede antal samples.

Kvaliteten af den fundne approksimerede sandsynlighed, dvs. hvor præcist de ønskede sandsynligheder er bestemt i forhold til den eksakte sandsynlighed, er afhængig af antallet af samples, der undersøges, og hvordan de udtages. Hvis problemområdet indeholder mange variable, eller hvis vi ønsker sandsynligheden for evidens, der forekommer med lille sandsynlighed, kan anvendelige samples forekomme sjældent, hvorfor der kan blive behov for store mængder samples [Jensen, 2001].

For at afhjælpe ovenstående problem, er der udviklet en række forskellige metoder til at udvælge samples, med det formål at få mængden af samples til hurtigt at afspejle fællessandsynlighedspotentialet over variablene i det bayesianske net. Dette har resulteret i en mængde forskellige algoritmer til

stokastisk simulation, der adskiller sig i kraft af måden, hvorpå de genererer samples. Af eksisterende algoritmer nævnes Gibbs sampling og Metropolis Hastings algoritmen, der begge er eksempler på Markov chain Monte Carlo algoritmer [Chib and Greenberg, 1995]. Disse algoritmer beskrives bl.a. i [York, 1992, Neal, 1993, Green and Murdoch, 1998, Jensen, 2001].

Arbejdet med approksimative algoritmer til inferens, baseret på sampling, er også forsøgt udvidet til IDer i bl.a. [Charnes and Shenoy, 2004, Ortiz and Kaelbling, 2000]. I modsætning til bayesianske net er målet her ikke længere at bestemme sandsynligheder, men at bestemme en suboptimal strategi for IDet.

Metoderne for simulering over IDer, der skitseres i førnævnte kilder, simulerer hver beslutning i IDerne modsat den temporale ordning; dvs. at politikken for den sidste beslutning først approksimeres vha. sampling, hvorefter beslutningen konverteres til en chanceknude, hvis sandsynlighedsfordeling afspejler den approximerede politik for beslutningen. Herefter bestemmes en politik for den næstsidste beslutning på samme måde. Således fortsættes iterativt baglæns gennem den temporale ordning, indtil politikker for alle beslutninger er bestemt.

Politikken for en beslutningsknude i et ID kan bestemmes ved, for hver instans af beslutningen og konfigurationerne over beslutningens relevante fortid, at sample over chancevariablene, der følger beslutningen. For hvert af disse samples bestemmes nytten. Den gennemsnitlige nytte vil således udgøre en approksimation af den forventede nytte for instansen af beslutningen og den valgte konfiguration over den relevante fortid for beslutningen. Ved at approksimere nytten for alle instanser af beslutningen og konfigurationerne over den relevante fortid, kan en approksimeret politik for beslutningen opstilles.

Anvendelse af sampling til løsning af UIDer, ville kunne opnås ved en forholdsvis naiv overførsel af sampling fra IDer til UIDer: Ved at opfatte de forskellige stier i S-DAGen som værende IDer, vil det være muligt at løse disse IDer hver for sig og ved at lave max-kombinering, hvor flere stier i S-DAGen mødes, vil det være muligt at få en approksimeret løsning til S-DAGen.

3.3 Valg af løsningsstrategi

Vi har i det foregående afsnit analyseret tre forskellige tilgange til design af en algoritme til løsning af UIDer på begrænset plads – tilgange, der primært har fundet deres anvendelse ved inferens i bayesianske net.

Hver af de tre tilgange til problemstillingen var forskellig fra de andre, og

de har som sådan repræsenteret hvert sit paradigme. Inden for hvert paradigme har vi forsøgt at give en kort beskrivelse af, hvorledes de pågældende teknikker ville kunne anvendes til at løse UIDer. Dette har hjulpet til at afdække fordele og ulemper ved de forskellige tilgangsvinkler. Fx er sampling begrænset af den kendsgerning, at det aldrig bliver muligt at opnå en eksakt løsning på et problem. Vi finder denne egenskab uheldig, idet vi mener, at et beslutningsstøttesystem, om muligt, bør give den bedst mulige rådgivning, hvilket indtræffer, når den eksakte løsning er fundet. I den forbindelse virker det som en unødvendig begrænsning i de situationer, hvor de nødvendige ressourcer er tilgængelige. En fordel ved denne tilgangsvinkel er muligheden for at kombinere ideen om anyspace med anytimeegenskaber, idet præcisionen vokser efterhånden som flere samples bliver undersøgt.

Algoritmer, der både vil kunne fungere i anyspace såvel som anytime, er ligeledes opnåelige ved brug af approksimerede potentialetræer, og modsat simulering vil det her være muligt at løse UIDer eksakt. Dog kræver sidstnævnte, at der er plads nok til at repræsentere sandsynligheds- og nyttepotentialer eksakt som træer og i værste fald, vil det betyde samme pladsforbrug som den traditionelle tabelrepræsentation.

Det sidste paradigme, konditionering, giver mulighed for at kunne løse UIDer eksakt uden at være bundet af noget betydeligt krav til den tilgængelige plads. Særligt Recursive Conditioning giver anledning til en trinvis afvejning af plads og tid. Ved at kombinere konditionering med vores overvejelser om trinvis kompilering, forventer vi at det vil blive muligt at videreføre en trinvis afvejning af plads og tid til en løsningsalgoritme for UIDer, idet det således ikke bliver nødvendigt at repræsentere samtlige politikker på en gang.

Med det på plads vil vi udvikle en algoritme til løsning af UIDer ved hjælp af konditionering. Vi vil yderligere tage udgangspunkt i Recursive Conditioning, idet dette er en algoritme, der indholder den trinvise afvejning af plads og tid, samt muligheden for at løse problemer eksakt. Denne algoritme vil kunne kombineres med trinvis kompilering og på denne måde opnås et løsningsværktøj til UIDer, der vil kunne anvendes i miljøer med begrænset plads. Et eksempel kunne være i autonome agenter.

I resten af denne rapport vil vi ikke fokusere op at opstille en løsning til et UID i form af en S-DAG med politikker for hver knude. I stedet for vil vi koncentrere os om at bestemme den maksimale forventede nytte for en S-DAG.

Under beregningerne af den maksimale forventede nytte vil det være muligt, parallelt med disse beregninger, at opstille valgfunktionerne, som diskuteret i afsnit 3.1, ved at udskifte max-operationen med en argmax-operation

ved max-marginaliseringen af beslutningsvariable. Ligeledes vil det være muligt, at opstille skridt-politikkerne, når der laves max-kombinering over nyttepotentialerne for børnene af knuderne i S-DAGen. Disse valgfunktioner og skridtpolitikker kan bestemmes og gemmes, for så vidt den tilgængelige plads tillader dette. Hvis der ikke er plads til at opstille alle politikker, kan trinvis kompilering i stedet benyttes.

Kapitel 4

Recursive Conditioning

I kapitel 3 tog vi udgangspunkt i forskellige tilgange til anyspacesandsynlighedsinferens i bayesianske net og afvejede fordele og ulemper ved disse tilgange – ikke mindst i forhold til løsningen af UIDer. Vi brugte denne afvejning til at bestemme en strategi for udviklingen af en anyspaceløsningsalgoritme til UIDer. Vi vil med udgangspunkt i konditionering, særligt Recursive Conditioning, RC, designe en sådan algoritme.

Før vi præsenterer en konditioneringsalgoritme til løsning af UIDer, vil vi i dette kapitel overordnet beskrive ideen bag RC, som beskrives i [Darwiche, 2000a, Darwiche, 2001]. I næste kapitel introduceres *S-DAG konditionering*, som den ønskede anyspaceløsningalgoritme for UIDer.

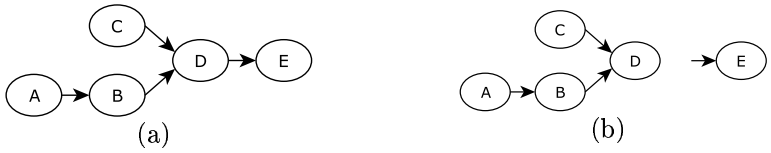
RC algoritmen er en algoritme til beregning af fællessandsynligheder på formen $P(\mathbf{x})$. Ønskes betingede sandsynligheder, må den fundamentale regel benyttes.

4.1 Anyspacesandsynlighedsinferens med konditionering

Anyspacesandsynlighedsinferens over et bayesiansk net kan gøres simpelt som eksemplificeret i følgende eksempel.

Et bayesiansk net kan ses som en kompakt repræsentation af fællessandsynlighedsfordelingen over alle variablene i nettet. Hvis vi har et bayesiansk net, som i figur 4.1(a), med variablene A, B, C, D og E, kan vi finde en vilkårlig fællessandsynlighed, fx $P(a_1, b_1, c_1, d_1, e_1)$, ved at instantiere alle variable i nettet til deres respektive tilstande. Når alle variable er instantierede,

vil de sandsynlighedspotentialer, der er specificeret for alle knuderne i nettet, hver især være reduceret ned til én værdi. Ved at tage produktet af disse værdier, fås sandsynligheden for den givne instantiering. Ønsker vi sandsynligheden $P(d_1)$, findes den ved at gennemløbe alle mulige konfigurationer over alle variablene på nær D, og holde $D = d_1$. Ved at addere alle de fundne fællessandsynligheder findes sandsynligheden $P(d_1)$.



Figur 4.1: Det bayesianske net i (a) bliver d-separeret til de to net i (b), når D bliver instantieret.

Hvis alle variable er binære, er denne udregning givet ved

$$P(d_1) = P(a_1, b_1, c_1, d_1, e_1) + P(a_2, b_1, c_1, d_1, e_1) + \ldots + P(a_2, b_2, c_2, d_1, e_2) \tag{4.1}$$

Ved denne løsningsmetode behøver man reelt set ikke mere plads end dén, der skal til for at gemme det originale bayesianske net og et kommatal; det foreløbige resultat, der gradvist opdateres til den ønskede sandsynlighed.

Ved at benytte ovenstående metode til sandsynlighedsinferens, tages der ikke højde for kausale sammenhænge og dermed heller ikke for variable, der kan være irrelevante i forhold til sandsynligheden for en anden variabel. Fx. gælder det, at E i ovenstående eksempel ikke har indflydelse på sandsynlighedsfordelingen for D, hvorfor E giver anledning til unødigt mange beregninger i metoden, der anvendes i udtryk (4.1). Med den beskrevne naive metode vil antallet af fællessandsynligheder, der skal udregnes, være eksponentielt i antallet af variable i det baysianske net, der ikke er instantieret i den ønskede sandsynlighed.

Det er imidlertid ikke altid nødvendigt at beregne alle disse fællessandsynligheder. Konditionering er en generel teknik til at reducere disse beregninger samtidigt med, at det stadig er muligt at anvende teknikken til at løse problemer på begrænset plads. Ideen med RC er at betinge på variable i et bayesiansk net og dermed dele nettet op i mindre net, der hver især kan løses på mindre plads. Løsningen til delnettene vil efterfølgende kunne kombineres til den ønskede sandsynlighed.

Selve RC algoritmen udnytter d-separationsegenskaber i et bayesiansk net til at opdele sandsynlighedsinferensproblemet. Ved at lægge evidens på en

mængde knuder, $\mathcal{C}$, i et bayesiansk net, kan man opnå at nettet kommer til at bestå af to d-separerede delnet, der kun har knuderne i $\mathcal{C}$ til fælles. I figur 4.1 er dette illustreret ved at lade knude D i det baysianske net i 4.1(a) få evidens. Dette bevirker at knuden E bliver d-separet fra resten af nettet, og denne knude kommer til at udgøre sit eget (trivielle) net 4.1(b).

Under den givne instantiering, $\mathbf{c}$, over variablene i $\mathcal{C}$ er det muligt at løse hvert delnet separat. Lad $\mathbf{x}$ være den instantiering, som vi ønsker sandsynligheden for. Hvis et net, N, deles i delnettene N^l og N^r, så kan sandsynligheden for instantieringen $\mathbf{x}$ for nettet N findes som

$$P^N(\mathbf{x}) = \sum_{\mathbf{c}} P^N(\mathbf{c}, \mathbf{x}) = \sum_{\mathbf{c}} P^{N^l}(\mathbf{c}, \mathbf{x}^l) P^{N^r}(\mathbf{c}, \mathbf{x}^r), \tag{4.2}$$

hvor $\mathbf{x}^l$ er instantieringen fra $\mathbf{x}$ på de variable, der indgår i delnettet N^l (tilsvarende for delnettet N^r). Sandsynligheden $P^{N^l}(\mathbf{c}, \mathbf{x}^l)$ kalder vi for løsningen til delnettet N^l og er altså sandsynligheden for $\mathbf{x}^l$ under konditionen $\mathbf{c}$.

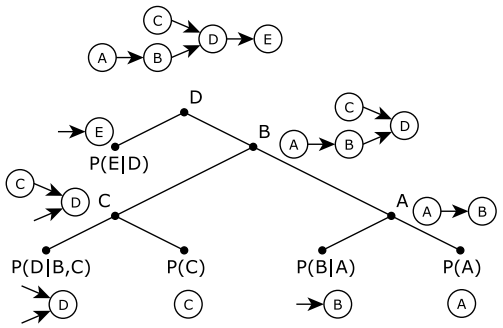
Hvert delnet kan iterativt splittes op. Således er det muligt at anvende udtryk (4.2) rekursivt på de opdelte delnet.

Opdelingen af det bayesianske net kan gentages iterativt indtil hvert delnet blot består af én knude. Dette giver anledning til lige så mange delnet, som der er knuder i det bayesianske net. Sammenhængene mellem de forskellige delnet kan repræsenteres ved et træ, hvor roden udgør det originale bayesianske net og de interne knuder udgør de opdelte net. Bladknuderne udgøres af de delnet, der kun består af én knude og til hver bladknude knyttes det sandsynlighedspotentiale fra det originale bayesianske net, der hører til denne enlige knude. Denne struktur kaldes et d-træ, idet det repræsenterer sammenhænge mellem d-separerede dele af et bayesiansk net vha. en træstruktur.

Da hvert net deles i to, betyder det, at der i et d-træ vil være to børn for hver intern knude, og at der vil være én bladknude for hver knude i det oprindelige bayesianske net. Et eksempel på et d-træ er givet i figur 4.2.

Børnene af en intern knude i et d-træ kan udgøre deltræer, som det også fremgår af figur 4.2. Børnene for en knude T benævnes T^l og T^r og refererer til henholdsvis det venstre og det højre deltræ. Knuder i d-træet bliver benævnt $T(\mathcal{C})$, hvor $\mathcal{C}$ er den mængde af variable, der d-separerer nettet, der knytter sig til knuden. Fx har vi i figur 4.2 en knude $T(\{B\})$, hvis venstre og højre deltræ henholdsvis er $T^l(\{B\}) = T(\{C\})$ og $T^r(\{B\}) = T(\{A\})$.

Potentialet, der knytter sig til bladknuden T, kalder vi ϕ_T. De variable, der findes i domænet for ϕ_T, benævnes $Vars\,(T)$. For en intern knude, T, er



Figur 4.2: En d-træs repræsentation af det bayesianske net i figur 4.1(a). På figuren har vi også angivet hvilke net, hver knude repræsenterer.

der per definition ikke associeret et potentiale og $Vars\,(T)$ udgør i stedet foreningsmængden af højre og venstre deltræs variable: $Vars\left(T^l\right) \cup Vars\,(T^r)$. I ovenstående eksempel svarer det til at $Vars\,(T(\{A\})) = \{A, B\}$.

Opfatter vi hvert deltræ T som værende isoleret, vil $Vars\left(T^l\right) \cap Vars\,(T^r)$ betegne de variable, som de to børn til T har tilfælles. Disse variable, vil være dem, der d-forbinder de to net, der er repræsenteret ved de to børn. Denne mængde af variable kaldes $Cutset\,(T)$ og er uafhængig af hvilke variable, der tidligere er blevet betinget på for at opnå deltræ T. Dette er inkluderet i definition 4.1 af et *cutset*.

Definition 4.1 (**Cutset**). Cutsettet for en intern knude, T, er givet ved:

$$Cutset\,(T) = Vars\left(T^l\right) \cap Vars\,(T^r) \setminus ACutset\,(T)\,,$$

hvor $ACutset\,(T)$ er foreningsmængden af cutsettene for alle Ts forfædre i d-træet. □

I et d-træ med flere knuder vælger vi at identificere de enkelte knuder ved deres cutset og refererer til dem som $T(cutset)$. I sammenhænge hvor der ikke er tvivl om meningen, kan T undlades.

Hvor mængden $Cutset\,(T)$ betegner de variable, der skal instantieres for at dele T^l og T^r fra hinanden, kan vi definere en anden mængde, der repræsenterer de variable, der skal instantieres for at d-separere delnettet T fra resten af nettet. Denne mængde består af de variable, der bliver instantieret for at opnå deltræ T og som også optræder i de potentialer, der findes ved bladknuderne i deltræet T. For knuden T betegnes denne mængde $Context\,(T)$ og den er formelt defineret i definition 4.2.

Definition 4.2 (**Kontekst**). For knuden T gives konteksten ved:

$$Context\,(T) = Vars\,(T) \cap ACutset\,(T)$$

□

I figur 4.2 er konteksten for knuden $T(\{A\})$ således defineret som $Context\,(T(\{A\})) = Vars(T(\{A\})) \cap ACutset\,(T(\{A\})) = \{A, B\} \cap \{B, D\} = \{B\}$.

Med udgangspunkt i ovenstående kendskab til d-træer, der udgør det beregningsmæssige fundament for Recursive Conditioning, bliver RC-algoritmen, nu beskrevet i det følgende afsnit.

4.2 RC-algoritmen

RC finder, som tidligere nævnt, fællessandsynligheden for givet evidens, $\mathbf{e}$, i et bayesiansk net. Med udgangspunkt i udtryk (4.2) beregnes sandsynligheden for evidensen, som angivet i følgende rekursive udtryk – den matematisk baggrund for RC-algoritmen:

$$\mathrm{RC}(T, \mathbf{e}) = \begin{cases} \phi_T(\mathbf{e}), & \text{hvis } T \text{ er en bladknude;} \\ \sum_{\mathbf{c}} \mathrm{RC}(T^l, \mathbf{e}^l\mathbf{c})\mathrm{RC}(T^r, \mathbf{e}^r\mathbf{c}), & \text{ellers,} \end{cases} \tag{4.3}$$

hvor T er en knude i et d-træ, $\mathbf{c}$ er en instantiering af variable i $Cutset\,(T)$ og $\mathbf{e}^l$ og $\mathbf{e}^r$ er instantieringen over variablene i hhv. $Vars(T^l)$ og $Vars(T^r)$.

RC algoritmen traverserer rekursivt ned gennem d-træet fra roden T, og betinger i hver knude på variablene i dens cutset. Herefter fortsættes rekursionen først af venstre deltræ, $T^l(T)$ og dernæst af det højre, $T^r(T)$. Dette gøres for alle konfigurationer over $\mathbf{c}$. Ved en bladknude initieres der ikke flere rekursive kald og i stedet for returneres ét tal; nemlig det der findes i potentialet, for den indgang, der er angivet for de registrerede instantieringer.

Giver de registrerede instantieringer ikke anledning til en entydig indgang i bladknudens potentiale, har dette potentiale ingen indflydelse på fællessandsynligheden for evidensen og der returneres blot tallet 1. I disse situationer vil variablen i bladknuden ikke have nogen indflydelse på den ønskede sandsynlighed. En sådan variabel kaldes barren. For et bayesiansk net gælder det, at en knude er barren, hvis hverken knuden selv eller dens efterfølgere har modtaget evidens.

I algoritme 1 er der angivet en pseudokode for RC-algoritmen. Linierne 1-3 bruges, når algoritmen når til en bladknude, mens linie 8 starter de rekursive kald. Den instantiering, der ønskes sandsynligheden for, skal registreres inden algoritmen kaldes første gang. De registrerede instantieringer gemmes globalt.

Når der rekursivt betinges ned gennem d-træet i RC-algoritmen, betinges der på de forskellige cutset fra rodknuden mod bladene. Med udgangspunkt

Algoritme 1 Recursive Conditioning. Denne algoritme finder sandsynligheden $P(\mathbf{e})$, hvor $\mathbf{e}$ er de instantieringer, der er registrerede fra starten.

RC(T)
1: **if** T *er en bladknude* **then**
2: $\mathbf{x} \leftarrow$ *den registrerede instantiering over* $Vars\,(T)$
3: **return** *Den indgang i* ϕ_T*, der matcher* $\mathbf{x}$
4: **else**
5: $p \leftarrow 0$
6: **for** *enhver tilladelig instantiering* $\mathbf{c}$ *over* $Cutset\,(T)$ **do**
7: *Registrér instantiering* $\mathbf{c}$
8: $p \leftarrow p +$ RC(T^l)RC(T^r)
9: *Fjern registreringen af* $\mathbf{c}$
10: **return** p

i d-træet på figur 4.2 betinges der således først på variabel D, så B og til sidst dennes børn C og A.

Med det in mente, er det muligt at finde eksempler på konfigurationer, hvor nogle rekursive kald bliver redundante; kald til T^r for $T(\{B\})$ hvor kun instantieringen af D er ændret, vil altid give det samme, da D ikke optræder i potentialerne, der knytter sig til bladknuderne under $T(\{B\})$ – dvs. $Vars\,(T^r(\{B\}))$.

Ved et kald til en given knude, T, er de instantierede variable, dem som optræder i $ACutset\,(T)$. Derfor forekommer redundante rekursive kald, når der findes variable i $ACutset\,(T)$, der ikke optræder i $Vars(T)$. Eller sagt med andre ord; så er rekursive kald til et deltræ, hvor instantieringen af dets kontekst ikke er ændret, redundante.

Vi har fundet, for d-træet på figur 4.2, at $Vars\,(T^r(\{B\})) = \{A, B\}$ og $ACutset\,(T^r(\{B\})) = \{B, D\}$ hvilket giver at $Context\,(T^r(\{B\})) = \{A, B\} \cap \{B, D\} = \{B\}$. Altså er instantieringen af D irrelevant for udfaldet af et rekursivt kald til $T^r(\{B\})$ og hvis D er binær, giver det dobbelt så mange kald som nødvendigt af RC algoritmen til $T^r(\{B\})$.

Ved at udnytte det faktum at rekursive kald til en knude er redundante, når instantieringen af dens kontekst tidligere har optrådt, kan antallet af kald minimeres ved at gemme resultatet af tidligere kald i en cache.

Hver mulig instantiering af en knudes kontekst kan give anledning til et unikt resultat for kaldene til knuden, hvorfor det er nødvendigt at gemme resultatet for hver instantiering over konteksten for knuden for at undgå redundante kald. Det er nødvendigt, at kunne undersøge om en given instantiering tidligere har optrådt, hvilket kan opnås ved at indeksere de gemte

resultater med deres respektive instantieringer over variablene i konteksten. Når der påny laves et kald til en knude, kan man således starte med at undersøge, om der findes et gemt resultat for den pågældende instantiering over variablene i konteksten – er det tilfældet, returneres blot det gemte resultat.

Gemmes alle resultater, undgås redundante kald og dermed fås en hurtigere algoritme. Denne forbedring af kørselstid sker dog på bekostning af algoritmens pladsforbrug. Algoritmen, der ikke bruger cache, kræver kun plads til, at repræsentere d-træet, de aktive rekursive kald, instantieringerne over variablene og resultatet, p. Algoritmen der bruger cache kræver yderligere plads til denne cache. Ved fuld cache, skal der være plads til en cache defineret over konteksten for alle knuderne i d-træet.

For at gemme resultatet for én konfiguration over konteksten for en knude, kræves lige så meget plads, som det kræver at repræsentere et kommatal (typisk en "double"). Det er muligt, kun at gemme en del af de opnåede resultater, og det bliver således muligt at forbedre algoritmens kørselstid, ved blot at øge dens tilladte pladsforbrug med størrelsen af ét kommatal. Dette giver anledning til en jævn afvejning af tid og plads; jo mere plads der er tilgængeligt, jo hurtigere kan algoritmen køre og omvendt. (Dog kan antallet af rekursive kald ikke reduceres yderligere, når der er tildelt cache til konteksten for alle knuderne.)

I de tilfælde, hvor der ikke er plads nok til at gemme alle resultater, er det nødvendigt at definere et regelsæt, der angiver om et givet resultat skal gemmes eller ej. Antallet af rekursive kald, der skal udføres før algoritmen terminerer, vil afhænge af, hvilke resultater der er gemt i cachen. Det er derfor ønskværdigt med et regelsæt, der søger at minimere antallet af rekursive kald ved at regulere, hvorvidt et resultat skal gemmes givet den tilgængelige plads. Et sådan regelsæt kaldes en *cachingstrategi*.

Tilføjelse af et resultat til en evt. cache, skal således ske i overenstemmelse med en valgt cachingstrategi; dvs. mellem ekstremerne alt eller intet, gemmes de indgange, der er plads til i henhold til en overordnet strategi – strategien er som sådan kun bundet af den plads, den har til rådighed.

Det er forholdsvis simpelt at introducere cache i RC-algoritmen, som den blev præsenteret i algoritme 1. Som tidligere nævnt, kan man i starten af hvert kald til algoritmen undersøge, om der er gemt et resultat i cachen og returnere dette resultat, hvis det findes. Dette gøres ved at indsætte nedenstående linier mellem linie 4 og 5 i algoritmen.

$\mathbf{y} \leftarrow$ *den registrerede instantiering over* $Context\,(T)$
if $cache_T\,[\mathbf{y}] \neq \text{nil}$ **then return** $cache_T\,[\mathbf{y}]$

Her gemmes den aktuelle instantiering af konteksten for knuden T i variablen

$\mathbf{y}$, hvorefter $cache_T[\mathbf{y}]$ laver et opslag i cachen for knuden T på indeks $\mathbf{y}$. Hvis dette opslag giver et resultat returnerer algoritmen med dette, ellers fortsættes algoritmen som normalt.

Umiddelbart inden de enkelte kald i algoritmen terminerer, ligger resultatet af kaldet gemt i p. Dette resultat skal gemmes i cachen til senere brug, hvis cachingstrategien dikterer dette. Resultatet kan gemmes i cachen ved at tilføje nedenstående linie umiddelbart før, de enkelte kald returnerer i linie 10.

if $cache?(T, \mathbf{y})$ **then** $cache_T[\mathbf{y}] \leftarrow p$

Her anvendes igen konfigurationen over konteksten, der blev gemt i $\mathbf{y}$, og kaldet til $cache?(T, \mathbf{y})$ undersøger om p skal gemmes i knuden T for konfigurationen $\mathbf{y}$. Skal p gemmes, gøres dette ved at indsætte p i cachen for T på indeks $\mathbf{y}$.

Lad os anvende RC-algoritmen, udvidet med cache, på d-træet i figur 4.2 til at finde $P(E{=}e_1)$. Instantieringen $E{=}e_1$ skal være registreret inden kaldet til rodknuden af d-træet. Algoritmen starter med at instantiere $D = d_1$ da cutsettet for rodknuden består af variablen D. Herefter foretages rekursive kald videre ned i træet; først det venstre deltræ, $T^l(\{D\})$, dernæst det højre, $T^r(\{D\})$.

$T^l(\{D\})$ er en bladknude, så dens sandsynlighedsbidrag findes umiddelbart med udgangspunkt i de registrerede instantieringer over D og E og potentialet, $P(E|D)$, der knytter sig til bladknuden. Svaret for kaldet til $T^l(\{D\})$ er således første gang $T^l(\{D\})$ bliver kaldt $P(e_1|d_1)$.

$T^r(\{D\})$ udgør et nyt ikke trivielt træ, $T(\{B\})$, hvor processen gentager sig; først køres rekursivt ned ad venstre deltræ, nu $T^l(\{B\})$, og derefter det højre; første gang med instantieringen $B = b_1$ og igen med $B = b_2$. Når de enkelte kald terminerer, gemmes resultaterne for kaldene i cachen ud for de aktuelle instantieringer over knudernes kontekst.

Når de første kald til henholdsvis $T^l(\{D\})$ og $T^r(\{D\})$ returnerer, er RC-algoritmen halvt færdig; den mangler stadig et gennemløb for $D = d_2$. Dette gennemløb forløber i høj grad som det første gennemløb. Først registreres $D = d_2$ og der udføres kald til først $T^l(\{D\})$ og dernæst til $T^r(\{D\})$. Ved algoritmens kald til $T(\{A\})$ returneres den gemte indgang for først $B = b_1$ og sidst $B = b_2$ til trods for at tilstanden for D har ændret sig ift. de to første kald til $T(\{A\})$. Resultatet af de første kald til $T(\{A\})$ kan genbruges, selvom instantieringen over D har ændret sig, da D ikke optræder i konteksten for $T(\{A\})$.

Kapitel 5

S-DAG konditionering

Vi beskriver i dette kapitel et design af en anyspaceløsningsalgoritme, der benytter konditionering. Designet i dette kapitel og arbejdet dokumenteret i resten af denne rapport er nyudviklet i forbindelse med dette speciale-projekt, med mindre andet er angivet.

I RC algoritmen, som vi beskrev i kapitel 4, udførte vi sandsynlighedsinferens ved hjælp af en del og hersk strategi, der opdelte et bayesiansk net i mindre net og løste disse under en række gensidigt udelukkende og fyldestgørende betingelser. Efterfølgende kombinerede vi løsningerne under de forskellige betingelser til en løsning på det samlede problem.

RC algoritmens resultater bygger på det faktum, at et bayesiansk net kan ses som en faktoriseret repræsentation af en fællessandsynlighedsfordeling, og disse faktorer kan opdeles i to grupper, der kan evalueres individuelt. Uheldigvis kan et UID ikke direkte betragtes som en faktorisering af en fællessandsynlighedsfordeling, da vi i et UID både har sandsynlighedsfordelinger og nyttepotentialer. Det giver således ikke umiddelbart mening at opdele et UID i to mindre UIDer, der kan løses individuelt.

RC algoritmens strategi kan derfor ikke anvendes direkte til løsning af UIDer. I dette kapitel vil vi i stedet beskrive en ide, der bygger videre på RC algoritmens strategi med at løse det aktuelle problem under en række forskellige gensidigt udelukkende og fyldestgørende betingelser, og kombinere disse delresultater til en samlet løsning.

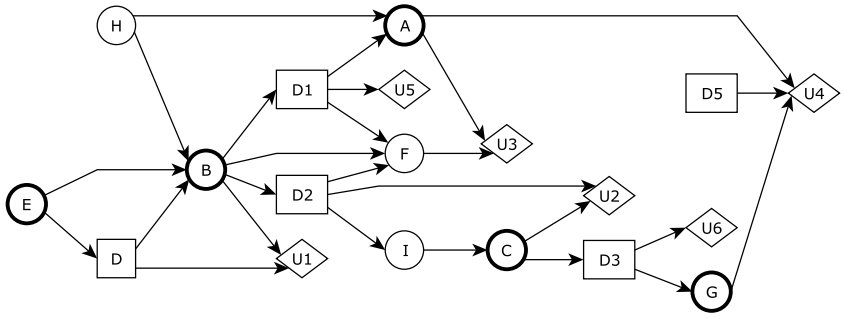
I dette kapitel starter vi med at illustrere, hvordan vores algoritme virker i hovedtræk med udgangspunkt i et eksempel, og i afsnit 5.2 følger vi op med en mere generel beskrivelse, samt en pseudokode for algoritmen. Den algo-

ritme vi præsenterer arbejder på lineært plads, hvilket sker på bekostning af algoritmens kørselstid. I afsnit 5.3 skal vi se på, hvordan vi kan forbedre algoritmens kørselstid ved at gemme mellemregninger, med henblik på senere genbrug. Gennemgangen i de første afsnit behandler ikke, hvordan vi kan finde betingede sandsynligheder – sandsynligheder der skal bruges under løsningen af UIDer. Dette behandles i afsnit 5.4, hvor vi ser på to forskellige anyspace måder at finde disse betingede sandsynligheder på.

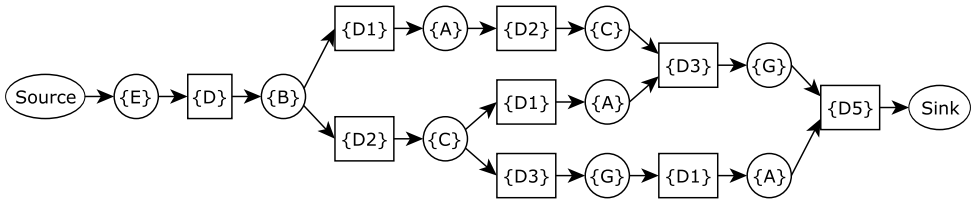
5.1 Eksempel

Vi vil nu beskrive hovedideen med vores anyspacealgoritme med udgangspunkt i et eksempel.

Lad os betragte UIDet på figur 5.1 med NFS-DAGen på figur 5.2.



Figur 5.1: Et UID.



Figur 5.2: En NFS-DAG for UIDet på figur 5.1.

Den maksimale forventede nytte, MFN, for D, jf. udtryk (2.4) på side 30, er givet ved

$$\rho_D(E) = \max_D \sum_B P(B|D,E) \max\{\rho_{D1}(E,D,B), \rho_{D2}(E,D,B)\}, \quad (5.1)$$

hvor $\rho_{D1}(E, D, B)$ og $\rho_{D2}(E, D, B)$ er den MFN for de to beslutningsknuder, der følger D i NFS-DAGen i figur 5.2 (disse beslutninger kaldes fremover *beslutningsbørn* af D).

For at bestemme $\rho_D(E)$ skal vi altså bestemme $\rho_{D1}(E, D, B)$ og $\rho_{D2}(E, D, B)$. Når vi skal bestemme $\rho_{D1}(E, D, B)$ skal vi arbejde med en funktion over E, D og B. Dvs. en funktion over 3 variable. Disse 3 variable er ikke instantierede.

I udtryk (5.1) skal vi også arbejde med sandsynlighedsfordelingen $P(B|D, E)$. Dette potentiale er et af de potentialer, der er specificeret sammen med UIDet, hvorfor fordelingen i denne situation ikke behøver at blive beregnet. I afsnit 5.4 skal vi se på det mere generelle tilfælde, hvor sandsynlighedsfordelingen skal beregnes ud fra de andre sandsynlighedsfordelinger for UIDet.

Ved udregning af den MFN gælder det generelt, at vi skal arbejde med sandsynlighedsfordelinger indeholdende $|\mathcal{C}_k \cup D_k \cup Rel(D_k)|$ variable, forventet-nytte-funktioner over den relevante fortid for D_k og den relevante fortid for hver af de i S-DAGen umiddelbart efterfølgende beslutningsknuder, D^i_{k+1}. Hvis de relevante fortider for D_k og D^i_{k+1} er store, kan de nævnte fordelinger og funktioner være så store, at det kan være et problem, at repræsentere dem i hukommelsen på selv moderne computere – som omtalt i afsnit 2.4 på side 37.

Dette problem kan vi omgå, ved at betinge på alle variablene i domænerne. I vores eksempel vil det betyde, at vi betinger på E, D og B. Hvis vi antager, at der allerede er betinget på variablene i den relevante fortid for D, så har vi

$$\rho_D(e) = \max_D \sum_B P(B|D, e) \max\{\rho_{D1}(e, D, B), \rho_{D2}(e, D, B)\}. \tag{5.2}$$

Ud af dette udtryk kan vi max-marginalisere den binære beslutning D, som vist i følgende udtryk

$$\begin{aligned} & \max_D \sum_B P(B|D, e) \max\{\rho_{D1}(e, D, B), \rho_{D2}(e, D, B)\} \\ = \; & \max \Big\{ \sum_B P(B|d_1, e) \max\{\rho_{D1}(e, d_1, B), \rho_{D2}(e, d_1, B)\} \\ , \; & \sum_B P(B|d_2, e) \max\{\rho_{D1}(e, d_2, B), \rho_{D2}(e, d_2, B)\} \Big\}, \end{aligned} \tag{5.3}$$

og vi kan sum-marginalisere den binære chancevariabel B ud af fx første led

i udtryk (5.3) ved

$$\begin{aligned} & \sum_{B} P(B|d_1,e)\max\{\rho_{D1}(e,d_1,B),\rho_{D2}(e,d_1,B)\} \\ = \; & \big(P(b_1|d_1,e)\max\{\rho_{D1}(e,d_1,b_1),\rho_{D2}(e,d_1,b_1)\} \\ + \; & P(b_2|d_1,e)\max\{\rho_{D1}(e,d_1,b_2),\rho_{D2}(e,d_1,b_2)\}\big) \end{aligned} \tag{5.4}$$

Ved at opstille et tilsvarende udtryk for det andet led i udtryk (5.3) og vælge resultatet af det udtryk med den største løsningsværdi, har vi i princippet fundet $\rho_D(e)$ fra udtryk (5.2).

Hver gang vi betinger på en variabel, opdeler vi det originale problem i mindre problemer, der kan løses hver for sig. Fx er løsningen til $\rho_D(e)$ blot én værdi. Som vist i udtryk (5.2) skal vi arbejde med potentialer over 2 uinstantierede variable for at finde den værdi, der er løsningen til $\rho_D(e)$. Ved først af betinge D til d_1 og løse første led af udtryk (5.3) og derefter at betinge D til d_2 og løse andet led af udtryk (5.3), reducerer vi problemet fra udtryk (5.2), så vi nu kun skal arbejde med potentialer over én uinstantieret variabel (variabel B). Dette problem simplificerer vi yderligere ved at betinge på B, så vi til sidst har betinget på alle variable i potentialerne. På denne måde kan vi reducere alle udregningerne, så de hver især kan udføres på lineært plads.

Ved S-DAG konditionering, som vi har valgt at kalde den beskrevne fremgangsmåde, "bevæger" vi os således rekursivt frem gennem den temporale ordning, der er specificeret af S-DAGen. Vi betinger på variablene gennem den temporale ordning, og reducerer således størrelsen af de potentialer vi skal løse. Reduktionen af potentialerne fortsætter, indtil potentialerne blot består af enkelte værdier, der beregnes individuelt. Til forskel fra RC for bayesianske net, er målet ved at betinge på variablene gennem den temporale ordning ikke at *opdele* det originale net (UIDet) i mindre net, men i stedet for at reducere størrelsen af de potentialer, vi skal finde.

5.2 S-DAG konditionering

Lad os se på, hvordan denne konditioneringsløsningsmetode virker generelt. Og lad os for forenklingens skyld antage, at alle variable er binære.

Fremover bruger vi $rel(X)$ til at betegne en *konkret* relevant fortid for en variabel X – en relevant fortid, hvor alle variablene er instantieret – og $Rel(X)$ til at betegne den relevante fortid for X, hvor alle variablene ikke nødvendigvis er instantieret. På samme måde kan vi skrive $past(X)$ og $Past(X)$ for at beskrive en hhv. konkret fortid for X, og blot variablene i

Xs fortid. Denne notation indeholder ikke information om, hvilken konkret fortid der er tale om, hvorfor vi kun vil bruge notationen i situationer, hvor denne information er irrelevant for vores argumentation.

For at finde den MFN for en beslutning D_k i en S-DAG for en instantieret konfiguration over den relevante fortid – en *konkret* relevant fortid, $rel(D_k)$ – opstilles udtrykket:

$$\rho_{D_k}(rel(D_k)) = \max_{D_k} \sum_{\mathcal{C}_k} P(\mathcal{C}_k|rel(D_k), D_k) \cdot \max\left\{\rho_{D^1_{k+1}}(Rel(D^1_{k+1})), \ldots, \rho_{D^b_{k+1}}(Rel(D^b_{k+1}))\right\}, \tag{5.5}$$

hvor D_k er en beslutning i en S-DAG umiddelbart inden en forgrening ud til børnene $D^1_{k+1}, \ldots, D^b_{k+1}$, og $\mathcal{C}_k$ er de frigivne chancevariable for D_k (se evt. figur 2.4 på side 26).

Fordi der netop tages udgangspunkt i en konkret fortid, er alle variable i $Past(D_k)$ instantierede. I udtryk (5.5) er D_k, $\mathcal{C}_k$ og de øvrige knuder i fremtiden for D_k ikke instantieret.

Da $\max_{D_k}$ operationen her er den yderste operation – dvs. den operation, der opererer på resultatet af de øvrige operationer – starter vi med at instantiere D_k til den ene af D_ks værdier, d_1, og løser det resulterende udtryk, hvor $D_k = d_1$. Værdien af resulterende udtryk med $D_k = d_1$ sammenlignes nu med værdien af det resulterende udtryk med $D_k = d_2$. Og den største af de to værdier er da løsningen til $\rho_{D_k}(rel(D_k))$. Det resulterende element, der skal løses, når D_k er instantieret, er

$$\sum_{\mathcal{C}_k} P(\mathcal{C}_k|rel(D_k), d_k) \cdot \max\left\{\rho_{D^1_{k+1}}(rel(D^1_{k+1})), \ldots, \rho_{D^b_{k+1}}(rel(D^b_{k+1}))\right\}, \tag{5.6}$$

for hver instans d_k af D_k.

Den yderste operation er her $\sum_{\mathcal{C}_k}$. Derfor instantieres nu $\mathcal{C}_k$ til dens tilstande. De resulterende led løses og resultaterne adderes, hvorved løsningen til elementet i (5.6) opnås.

Det led der fås ved at instantiere $\mathcal{C}_k$ til $\mathbf{c}_k$ i (5.6) er givet ved:

$$P(\mathbf{c}_k|rel(D_k), d_k) \cdot \max\left\{\rho_{D^1_{k+1}}(rel(D^1_{k+1})), \ldots, \rho_{D^b_{k+1}}(rel(D^b_{k+1}))\right\}. \tag{5.7}$$

Den yderste operation er nu en multiplikation af $P(\mathbf{c}_k|rel(D_k), d_k)$ med resultatet af max-kombineringen. Vi vil i afsnit 5.4 se på, hvordan vi kan beregne $P(\mathbf{c}_k|rel(D_k), d_k)$. Max-kombineringen udføres ved at løse de enkelte led for beslutningsbørnene til D_k. Disse led kan opstilles og løses rekursivt, som vi netop har beskrevet ved løsning af udtryk (5.5).

Ved rekursivt at reducere det oprindelige udtryk til mindre udtryk ved at betinge på de uinstantierede variable er det således muligt at finde den MFN for enhver beslutning i en S-DAG.

Den MFN for en S-DAG, Ξ, findes ved at løse følgende udtryk

$$\rho_\Xi = \sum_{\mathcal{C}_0} P(\mathcal{C}_0) \cdot \max\left\{\rho_{D_1^1}(rel(D_1^1)), \ldots, \rho_{D_1^b}(rel(D_1^b))\right\}, \tag{5.8}$$

hvor C_0 er de observationer, der er frie inden første beslutning tages (kaldet gratisobservationerne), og $D_1^1, \ldots, D_1^b$ er de mulige første beslutninger.

Fremgangsmåden er at vi rekursivt betinger på alle variable der skal max- og sum-marginaliseres, hvorved vi hele tiden arbejder med potentialer, der altid er fuldt instantieret.

På denne måde kan vi beregne den MFN af en S-DAG på lineært plads i antallet af knuder i UIDet, ud over den plads, vi skal bruge til at repræsentere UIDet og S-DAGen. Det eneste vi skal have plads til, er resultatet af tidligere kald i form af ét kommatal, en bogføring over hvilke instantieringer vi har over variablene i UIDet, og en stack til at holde styr på de rekursive kald vi har lavet ned gennem S-DAGen. Denne stack bliver aldrig højere end antallet af knuder fra *Source* til *Sink* i S-DAGen, hvorfor denne stack aldrig bliver højere end antallet af beslutninger og observerbare chancevariable i UIDet.

5.2.1 Algoritme – uden cache

Vi giver her en algoritme til at løse en S-DAG, som netop beskrevet. Vi starter med at beskrive algoritmen, som den virker uden caching. I afsnit 5.3 vil vi udvide denne algoritme til også at udnytte caching.

Algoritmen kan ses som en dybde først vandring gennem S-DAGen. Under denne vandring beregnes den MFN af S-DAGen som angivet i udtryk (5.8), hvor ρ_{D_k} findes som

$$\begin{aligned}&\rho_{D_k}(rel(D_k))\\ &= \begin{cases} \max_{D_k} \sum_{\mathcal{C}_k} P(\mathcal{C}_k|rel(D_k), D_k) \cdot \rho_{D_{k+1}} & \text{hvis} \quad k < nDec \\ \max_{D_k} \sum_{\mathcal{C}_k} P(\mathcal{C}_k|rel(D_k), D_k) \cdot V & \text{hvis} \quad k = nDec \end{cases}\end{aligned} \tag{5.9}$$

hvor $\rho_{D_{k+1}} = \max\left\{\rho_{D^1_{k+1}}(rel(D^1_{k+1})), \ldots, \rho_{D^b_{k+1}}(rel(D^b_{k+1}))\right\}$, $V = \sum \Psi$ og $nDec$ er antallet af beslutninger i UIDet.

Algoritmen består af en række hjælpefunktioner, der udfører max-marginalisering, sum-marginalisering, max-kombinering mv. over variablene i S-DAGens knuder. Ud fra S-DAGens struktur bestemmes hvilke hjælpefunktioner, der skal kaldes, og hvornår de skal kaldes.

Algoritmen RcSdag(Ξ) (algoritme 2) finder den MFN for S-DAGen, Ξ. Algoritmen starter med at finde barnet, $\mathcal{C}_0$, til *Source* i Ξ, der indeholder gratisobservationerne, og sum-marginaliserer disse observationer ud ved et kald til RcSumMar($\mathcal{C}_0$). Denne funktion betinger rekursivt over gratisobservationerne og starter for hver instantiering 2 nye kald, der beregner hhv. sandsynligheden for den aktuelle instantiering (RcCondProb($\mathcal{C}_0$) – algoritme 4) og den MFN for fremtiden (RcMaxComb($\mathcal{C}_0$)).

RcCondProb($\mathcal{C}_0$) beskriver vi i afsnit 5.4, mens RcMaxComb($\mathcal{C}_0$) for hvert af beslutningsbørnene, D_i, til $\mathcal{C}_0$ starter et kald af RcMaxMar(D_i), der beregner den MFN af D_i ved at max-marginalisere D_i. Dette sker ved at betinge på alle D_i og sum-marginalisere de frigivne chancevariable i den efterfølgende chanceknude, $\mathcal{C}$, med et kald til RcSumMar($\mathcal{C}$). Sådan fortsættes rekursivt.

Hele processen kan ses som en dybde først vandring gennem S-DAGen, hvor der rekursivt betinges på alle variable, der optræder i S-DAG knuderne. Hver gang de rekursive kald når til *Sink*, betinges der på de uobserverbare variable, og der laves et kald til RcUtil(), der finder nytten af de instantieringer, der er blevet lavet.

Bemærk at vi kan håndtere de uobserverbare chancevariable på lige fod med de observerbare chancevariable ved at placere dem i *Sink* og således behandle *Sink* på linie med de øvrige chanceknuder. Eneste forskel på *Sink* og de andre chanceknuder er, at RcSumMar(*Sink*) kører RcUtil() i linie 5, i stedet for at køre RcMaxComb($\mathcal{C}$) i linie 7, der starter flere kald ned gennem S-DAGen.

For at simplificere forklaringen af S-DAG konditionering og pseudokoden har vi antaget, at S-DAGen, Ξ, har en speciel opbygning. Vi har antaget, (1) at beslutningsknuder kun indeholder *en* beslutningsvariabel. Generelt gælder det, at beslutningsknuder kan være mængdeknuder på samme måde som chanceknuder. Hvis man skal håndtere en mængdeknude med beslutningsvariable, kan man blot lægge en vilkårlig ordning over disse beslutningsvariable, og betragte dem som flere efterfølgende knuder. Dette leder os over til vores anden antagelse: Vi har antaget (2) at en beslutningsknude altid efterfølges af en chanceknude med de chancevariable, der frigives af beslutningsvariab-

Algoritme 2 S-DAG konditionering til bestemmelse af den MFN for en S-DAG.

RcSdag(Ξ)
Input: Ξ - *En S-DAG.*
1: $\mathcal{C}_0 \leftarrow$*Barnet til Source i* Ξ.
2: **return** RcSummar($\mathcal{C}_0$)

RcSummar($\mathcal{C}$)
Input: $\mathcal{C}$ - *En S-DAG-knude med chancevariable.*
1: $v \leftarrow 0$
2: **for all** *konfigurationer,* **c**, *over* $\mathcal{C}$ **do**
3: *Registrér instantieringen* **c**
4: **if** $\mathcal{C}$ *er Sink* **then**
5: $v \leftarrow v +$ RcCondProb($\mathcal{C}$) $\cdot$ RcUtil()
6: **else**
7: $v \leftarrow v +$ RcCondProb($\mathcal{C}$) $\cdot$ RcMaxComb($\mathcal{C}$)
8: *Afregistrér instantieringen* **c**
9: **return** v

RcUtil()
1: **x** $\leftarrow$ *De registrerede instantieringer over potentialerne i* Ψ.
2: **return** $\sum \Psi(\mathbf{x})$

RcMaxComb($\mathcal{C}$)
Input: $\mathcal{C}$ - *En S-DAG-knude med chancevariable.*
1: $v \leftarrow -\infty$
2: **for all** *børn,* D_i, *af* $\mathcal{C}$ **do**
3: $v \leftarrow \max\{v,$ RcMaxMar(D_i)$\}$
4: **return** v

RcMaxMar(D)
Input: D - *En S-DAG-knude med en beslutning.*
1: $\mathcal{C} \leftarrow$ *S-DAG-knuden med de chancevariable som* D *frigiver.*
2: $v \leftarrow -\infty$
3: **for all** *konfigurationer,* **d**, *over* D **do**
4: *Registrér instantieringen* **d**.
5: $v \leftarrow \max\{v,$ RcSummar($\mathcal{C}$)$\}$
6: *Afregistrér instantieringen* **d**.
7: **return** v

lene i beslutningsknuden i Ξ. Denne antagelse gælder ikke generelt, som det kan ses i fx S-DAGen på figur 2.3 på side 25. Hvis denne situation skal fanges i pseudokoden, så skal der indføres et tjek for, om den efterfølgende knude er en chanceknude eller en beslutningsknude, og den funktion, der startes, skal så hhv. sum- eller max-marginalisere variablene i den efterfølgende knude ud. Vi har også antaget (3) at der kun er forgreningspunkter i S-DAGen efter chanceknuder. Dette gælder heller ikke generelt, da fx en beslutningsknude i S-DAGen kan efterfølges af flere beslutningsknuder. Dette kan ligeledes håndteres ved at udvide pseudokoden med et ekstra tjek. Endelig har vi antaget, (4) at der altid vil være mindst én gratis observation, således at barnet til *Source* er en chanceknude. Pseudokoden kan forholdsvis direkte generaliseres til at kunne løse alle tilladelige S-DAGe. Dette vil vi dog ikke behandle nærmere i denne rapport.

5.2.2 Tidskompleksitet

I afsnit 5.4 beskriver vi en metode til at bestemme sandsynlighederne, der skal bruges under S-DAG konditioneringen. Med denne metode til at bestemme sandsynligheder, bliver tidskompleksiteten af S-DAG konditionering uden cache $O(nNodes \cdot nPath \cdot exp(nDec + nObs + nHid) \cdot exp(cond^*))$, hvor $nNodes$ er antallet af knuder i S-DAGen, $nPath$ er antallet af forskellige maksimalt orienterede stier i S-DAGen og $nDec$, $nObs$ og $nHid$ er antallet af beslutninger, observerbare og uobserverbare chancevariable i UIDet. Endelig er $cond^*$ en størrelse, der aldrig er større end antallet af uobserverbare chancevariable i UIDet.

Denne kompleksitet udleder vi i kapitel 7.

5.3 Caching

S-DAG konditioneringsalgoritmens tidskompleksitet er væsentligt højere end VE algoritmens tidskompleksitet, der blev beskrevet i afsnit 2.4. Til gengæld har den en langt bedre pladskompleksitet, da den kun bruger lineært plads i antallet af beslutninger og chancevariable i UIDet, mens VE algoritmen derimod har et pladsforbrug, der i worst case vokser eksponentielt med antallet af knuder i UIDet. I dette afsnit søger vi at forbedre algoritmens kørselstid ved at indføre en cache, der giver mulighed for at vi kan genbruge beregninger og dermed reducere antallet af rekursive kald – i lighed med RC. Ved at bruge plads til cache, vil algoritmen ikke længere have en lineær pladskompleksitet. Dog kan det kontrolleres, hvor meget plads, der bliver brugt.

Indførelsen af cache betyder dermed at der opstår en afvejning af plads

og tid; jo mere plads der gives til cachen, jo mindre tid skal der ideelt set gå, inden algoritmen terminerer.

I kapitel 4 introducerede vi begrebet $Context\,(T)$. Det gjaldt, at udregningerne der blev udført i d-træet under T var afhængige af instantieringerne over variablene i $Context\,(T)$. Da vi indførte caching i d-træet, definerede vi den over tilstandsrummet for variablene i $Context\,(T)$, og det gjaldt således, at flere kald til T, der var enige om instantieringerne over $Context\,(T)$ ville give det samme resultat, hvorfor resultatet af første kald ville kunne genbruges ved de efterfølgende kald.

Det virker naturligt, at undersøge mulighederne for at overføre begrebet kontekst til S-DAG-knuderne, så vi tilsvarende kan identificere redundante kald under S-DAG konditionering, og dermed åbne mulighed for genbrug af beregninger.

Vi skal nu se, hvordan vi bl.a. kan reducere det samlede antal kald, som vores algoritme udfører, ved at indføre en cache på knuden $D3$ i den øverste sti i S-DAGen på figur 5.2. For hver maksimalt orienteret sti mellem $Source$ og $D3$ vil $D3$ blive kaldt én gang per mulig instantiering over variablene i fortiden for $D3$. For hver af kaldene til $D3$ vil konditioneringsalgoritmen beregne ρ_{D3} én gang, hvis der ikke bruges cache. Når vi har gennemgået, hvordan vi kan indføre cache på S-DAGen, vil det stå klart, at beregningen af ρ_{D3} kun behøver at blive udført to gange. De resterende kald til $D3$ kan i stedet for blive erstattet med opslag i en cache for $D3$.

5.3.1 Konteksten for en S-DAG-knude

Vi starter nu med at fastlægge hvilke variable, der skal indgå i konteksten for en S-DAG-knude, og i afsnit 5.3.4 skal vi se på, hvordan vi kan bruge en cache defineret over denne kontekst til at reducere antallet af rekursive kald, der udføres under løsningen af en S-DAG. Vi vil kalde konteksten for en S-DAG-knude $\mathcal{K}$ for $SContext\,(\mathcal{K})$.

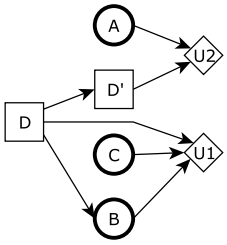
Da $Rel(D)$ netop er de variable, der kan have indflydelse på politikken for en given beslutning, D, kan man foranlediges til at tro, at $SContext\,(D) = Rel(D)$. Men en variabel, A, i fortiden for D, der er irrelevant for D kan godt være relevant for en beslutning, D', i fremtiden for D. Og hvis instantieringen af A ændres, vil det medføre, at udregningerne, der udføres ved S-DAG konditionering for D', vil kunne give et nyt resultat. Figur 5.3 viser et eksempel, hvor en variabel i fortiden for en beslutning, D, er irrelevant for D men relevant for en beslutning i fremtiden for D. Da sådanne variable kan forekomme i en S-DAG, gælder det, at konteksten for en S-DAG beslutningsknude, D, ikke altid vil være lig med $Rel(D)$ – konteksten kan

altså være større.

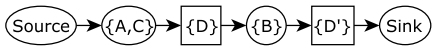
En NFS-DAG for UIDet på figur 5.3 består af stien $Source \prec \{A, C\} \prec D \prec B \prec D' \prec Sink$ – som vist på figur 5.4. $SContext(D)$ skal her være $\{A, C\}$, da den relevante fortid for D er C og den relevante fortid for D' er A. Således er UIDet på figur 5.3 et eksempel på, at $SContext(D)$ kan indeholde variable, der ikke er i $Rel(D)$.

Hvis en cache for D kun var defineret over den relevante fortid for D, så ville værdier gemt i denne cache ikke afspejle, at en ændring af instantieringen af A kan ændre den forventede nytte for det delscenarie, der følger beslutningen D. En sådan cache-værdi ville være forkert at genbruge, da den forventede nytte for et scenarie startende med D afhænger af den forventede nytte af beslutning D', og denne forventede nytte afhænger af instantieringen af A.

Der kan dog være flere situationer, hvor det vil være tilstrækkeligt at definere en cache over den relevante fortid for en beslutning, D. Dette vil være de situationer, hvor alle de variable fra fortiden af D, der bliver relevante i delscenarier følgende D, alle er variable, der også er relevante for D selv. Hvis vi lader $FtrRel(D)$ betegne foreningsmængden over $Rel(\mathcal{K})$ for alle variable, $\mathcal{K}$, i fremtiden for D (inkl. D selv), så vil det altså være tilstrækkeligt med en cache defineret over $Rel(D)$ hvis $FtrRel(D) \cap Past(D) = Rel(D)$.



Figur 5.3: Et UID. A er i fortiden for D, da A kan observeres inden første beslutning træffes, men A er kun relevant for D'.



Figur 5.4: En NFS-DAG for UIDet på figur 5.3.

Vi lader konteksten for en S-DAG-knude $\mathcal{K}$ være defineret i det følgende

Definition 5.1 (**Kontekst for en S-DAG-knude**). Konteksten for en knude, $\mathcal{K}$, i en S-DAG, Ξ, er de variable, i fortiden for $\mathcal{K}$, hvis instantieringer ikke kan ændres, uden at det har betydning for de udregninger algoritme 2 udfører for $\mathcal{K}$ og for knuderne, der efterfølger $\mathcal{K}$ i Ξ. □

Konteksten for en S-DAG-knude er identificeret ved

$$SContext\,(\mathcal{K}) = Vars\,(\mathcal{K}) \cap Past(\mathcal{K}), \tag{5.10}$$

hvor $Vars\,(\mathcal{K})$ er de variable, der optræder i nyttepotentialer, der knytter sig til S-DAG-knuder, der ligger i fremtiden for $\mathcal{K}$ (inkl. $\mathcal{K}$ selv) – fremover betegnet $Ftr(\mathcal{K})$ – samt de variable, der optræder i den relevante fortid for knuderne i $Ftr(\mathcal{K})$.

Den relevante fortid for en beslutningsknude er bestemt som beskrevet i afsnit 2.2.2 på side 30. Den "relevante fortid", $Rel(\mathcal{C})$, for en chanceknude (observerbar eller uobserverbar), $\mathcal{C}$, i en S-DAG, er de variable, hvis instantiering kan have indflydelse på sandsynlighedsfordelingen $P(\mathcal{C}|Past(\mathcal{C}))$. Det gælder således, at

$$P(\mathcal{C}|Past(\mathcal{C})) = P(\mathcal{C}|Rel(\mathcal{C})). \tag{5.11}$$

Ideelt set, skal $Rel(\mathcal{C})$ altså være de variable, som $\mathcal{C}$ er betinget afhængig af. Men da en analyse af betinget afhængighed vil kræve, at sandsynlighedstabellerne skal analyseres, kan det blive en dyr operation at tjekke for betinget afhængighed. I stedet for vil vi lade $Rel(\mathcal{C})$ betegne de variable, som $\mathcal{C}$ er d-forbundet med (se definition 2.7 på side 30). Dette tjek kan laves ud fra en strukturel analyse af UIDet og vil derfor ofte være billigere at udføre. Da det gælder, at alle variable, som $\mathcal{C}$ er betinget afhængig af, også vil være d-forbundede med $\mathcal{C}$, er vi altså sikre på, at $Rel(\mathcal{C})$ på denne måde kommer til at indeholde alle variable, som $\mathcal{C}$ er betinget afhængig af.

For at kunne omtale Rel konsistent for chancevariable og beslutningsvariable, vil vi kalde Rel for chanceknudens relevante fortid.

5.3.2 Reduktion af konteksten for en S-DAG-knude

Som algoritmen i afsnit 5.2.1 virker, laves alle opslag til nyttepotentialer i $Sink$. Vi siger, at nytteknuderne er knyttet til $Sink$. Dette er en konsekvens af, at algoritmen er en direkte implementering af udtryk (5.9). Algoritmen er lavet således for at forklare grundideen i S-DAG konditionering på en simpel måde. Som vi skal se i det følgende, er der dog flere ulemper ved at implementere dette udtryk direkte.

Da alle nyttepotentialer er tilknyttet $Sink$, bliver $Vars\,(D5)$ for NFS-DAGen på figur 5.2 alle variable, der optræder i domænet for alle nyttepotentialerne ($\{D, B, C, D2, A, F, D5, G, D1, D3\}$) og dem som optræder i $FtrRel(D5)$. $FtrRel(D5)$ er her $Rel(D5) \cup Rel(Sink) = Rel(D5) \cup Rel(F) \cup Rel(H) \cup Rel(I) = \{A, G, B, E, D, D1, D2, C\}$. Hvilket giver $Vars(D5) =$

$\{A, B, C, D, D1, D2, D3, D5, E, F, G\}$ og $SContext(D5) = Vars(D5) \cap Past(D5) = Past(D5)$.

$SContext\,(D5)$ bliver således lig med $Past(D5)$. Dette er uheldigt, da vi definerer vores cache for knuden $D5$ over variablene i $SContext\,(D5)$, og da sådan en cache skal være defineret over hele tilstandsrummet over variablene i $SContext\,(D5)$, vokser pladsforbruget af en sådan cache eksponentielt med antallet af variable i $SContext\,(D5)$. Vi skal nu se på, hvordan vi kan ændre vores algoritme, så konteksten for en knude i mange tilfælde bliver reduceret.

Fordeling af nyttepotentialer Den eneste grund til, at alle nyttepotentialer ligger på $Sink$ er som sagt, at algoritme 2 på denne måde kommer til at modsvare udtryk (5.9). Dette udtryk kan vi imidlertid omskrive v.h.a. den distributive lov til

$$\rho_{D_k}(rel(D_k)) = \begin{cases} \max_{D_k}\left(V_{D_k} + \sum_{\mathcal{C}_k} P(\mathcal{C}_k|rel(D_k), D_k) \cdot (V_{\mathcal{C}_k} + \rho_{D_{k+1}})\right) & \text{hvis } k < nDec \\ \max_{D_k}\left(V_{D_k} + \sum_{\mathcal{C}_k} P(\mathcal{C}_k|rel(D_k), D_k) \cdot V_{\mathcal{C}_k}\right) & \text{hvis } k = nDec \end{cases} \tag{5.12}$$

hvor $\rho_{D_{k+1}} = \max\left\{\rho_{D^1_{k+1}}(rel(D^1_{k+1})), \ldots, \rho_{D^b_{k+1}}(rel(D^b_{k+1}))\right\}$ og V_{D_k} og $V_{\mathcal{C}_k}$ er summen af de nytteknuder, der omhandler hhv. D_k og $\mathcal{C}_k$ og ikke nogle variable, der følger hhv. D_k og $\mathcal{C}_k$ i S-DAGen. Vi betegner disse summer med hhv. $V_{D_k} = \sum \Psi_{D_k}$ og $V_{\mathcal{C}_k} = \sum \Psi_{\mathcal{C}_k}$. Vi skal nu se på, hvordan vi kan identificere mængderne Ψ_{D_k} og $\Psi_{\mathcal{C}_k}$.

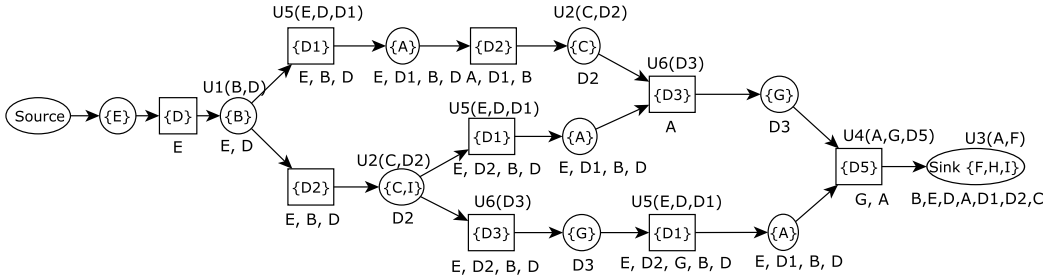
Hvis en nytteknude, U, kun afhænger af instantieringen over variable, der ligger før en knude $\mathcal{K}$ i S-DAGen, vil ændringerne i instantieringen over variablene i knuderne efter $\mathcal{K}$ ikke ændre nyttebidraget fra U.

Hvis vi fx ser på nytteknuden $U1$ i UIDet på figur 5.1, afhænger den kun af instantieringen over D og B. Nyttebidraget fra nytteknuden $U1$ er fastlagt, når D og B er instantieret. Vi siger, at $U1$ frigives, når D og B er instantieret og vi indfører definitionen

Definition 5.2 (**Frigivelse af nytteknuder**). En nytteknude *frigives*, når alle knuder i dens domæne er instantieret. □

Vi kan således flytte $U1$ op gennem NFS-DAGen og udregne nyttebidraget fra $U1$ allerede når D og B er blevet instantieret.

Figur 5.5 viser NFS-DAGen fra figur 5.2, hvor vi har angivet de frigivne nytteknuder (og Rel) for alle knuderne i NFS-DAGen. Derudover har vi også eksplicit vist de uobserverbare chancevariable i $Sink$.



Figur 5.5: En NFS-DAG for UIDet på figur 5.1. På alle knuder er *Rel* og de frigivne nyttepotentialer angivet.

Når potentialerne på denne måde medtages i beregningen af den MFN, når de bliver relevante, modsvarer det variabelelimineringsalgoritmens måde at medtage nyttepotentialerne, når potentialernes domæne – første gang modsat gennem elimineringsrækkefølgen – indeholder den variabel, der skal elimineres. Dette blev tidligere identificeret ved tildelingen i udtryk (1.8) på side 17.

Ved at introducere nogle af nytteknuderne, $\mathcal{U}$, tidligere i den temporale ordning opnås, at beregninger, der baserer sig på disse nytteknuder, ikke skal gentages, hvis vi senere ændrer på instantieringen over variable, der ligger placeret senere i den temporale ordning. Eller sagt på en anden måde: Når vi ændrer instantieringen over variable, der er placeret senere i den temporale ordning, behøver vi ikke hele tiden ajourføre bidraget til den samlede forventede nytte, der komme fra $\mathcal{U}$, da dette bidrag ikke bliver ændret.

$Vars\,(\mathcal{K}')$ kan ligeledes blive reduceret, for alle de knuder, $\mathcal{K}'$, som vi flytter nyttepotentialerne op forbi, når vi flytter nyttepotentialerne fra $Sink$ og op til en knude $\mathcal{K}$. Hvis de variable, vi reducerer $Vars\,(\mathcal{K}')$ med også optræder i $Past(\mathcal{K}')$, så reducerer vi samtidigt $SContext\,(\mathcal{K}')$ ved denne operation. Når vi reducerer konteksten, reducerer vi dermed også størrelsen af de cache, vi skal indføre for at udnytte genbrug af beregninger under S-DAG konditionering. Når vi reducerer cachen, reducerer vi også algoritmens samlede pladskrav ved fuld cache.

5.3.3 Medtag uobserverbare chancevariable tidligere

Vi har nu set, hvordan vi kan reducere konteksten for S-DAG-knuderne ved at medtage nyttepotentialerne tidligere i S-DAGen. Dette kunne vi gøre, hvis nyttebidraget fra nytteknuderne var uafhængigt af de sidste variable i S-DAGen. Vi skal nu se på, hvordan vi ligeledes kan flytte de uobserverbare

chancevariable op gennem S-DAGen, hvilket også kan reducere konteksten for S-DAG-knuderne.

Som vi har illustreret på figur 5.5, er $Rel(Sink) = Rel(F) \cup Rel(H) \cup Rel(I) = \{B, D1, D2\} \cup \{B, E, D, A, D1\} \cup \{C, D2\} = \{B, E, D, A, D1, D2, C\}$. Da $Sink$ er den sidste knude for alle stierne i en S-DAG og da alle de uobserverbare chancevariable ligger på $Sink$, kommer $FtrRel$ for alle knuder, $\mathcal{K}$, i S-DAGen til at indeholde alle variable i $Rel(Sink)$. De af disse variable, der også indgår i $Past(\mathcal{K})$, vil ud fra udtryk (5.10) også indgå i konteksten for $\mathcal{K}$. Der er altså mulighed for, at reducere konteksten for $\mathcal{K}$ ved at reducere $Rel(Sink)$, hvilket vi kan gøre, ved at flytte nogle af de uobserverbare chancevariable i $Sink$ op gennem S-DAGen.

Den temporale ordning for UIDer dikterer, at alle de uobserverbare chancevariable, $\mathcal{U}$, skal komme sidst. Argumentet for dette er, at ved at diktere, at de uobserverbare chancevariable skal være sidst i den temporale ordning, er man sikker på, at alle variable fra $Past(\mathcal{U})$, der kan være relevante for sandsynligheden $P(\mathcal{U}|Rel(\mathcal{U}))$, har modtaget evidens.

Denne sandsynlighed er essentiel at få bestemt, når vi arbejder med uobserverbare chancevariable, da denne sandsynlighed således udtrykker vores bedste bud på tilstanden for de uobserverbare chancevariable. Det er altså en betingelse for at bestemme den MFN, at sandsynligheden for de uobserverbare chancevariables tilstand (givet det konkrete beslutningsscenarie) fastlægges bedst muligt.

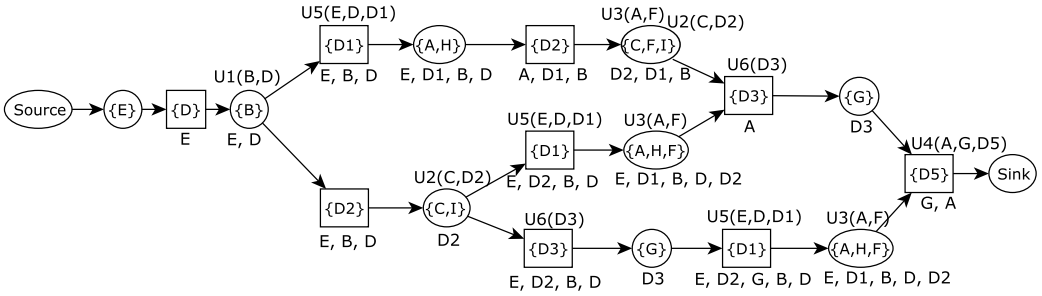
Denne betingelse vil imidlertid også være opfyldt for flere af de uobserverbare variable, selvom de placeres længere oppe i S-DAGen end i $Sink$. Hvis vi betragter den øverste sti på figur 5.2, bidrager hverken C, $D3$, G eller $D5$ med viden til at bestemme $P(F|Past(F))$. Således vil F kunne placeres før C, og udregningerne for løsningen af S-DAGen vil stadig give det samme resultat.

Generelt gælder det, at tilstanden for en uobserverbar variabel, $U \in \mathcal{U}$, ikke vil kunne bestemmes mere præcist, når først der er evidens på alle variablene i $Rel(U)$. Således vil vi kunne flytte en uobserverbar chancevariabel op gennem S-DAGen, og placere den umiddelbart efter den sidste variabel fra $Rel(U)$. Dog må vi aldrig flytte U op før en beslutningsvariabel, D, hvis U ville blive relevant for D. Da vi ikke kan observere U, vil det altså ikke give mening, at have U før D i S-DAGen, hvis U ville være relevant for D.

S-DAGen repræsenterer vores elimineringsrækkefølge af variablene i UIDet. Når vi flytter en uobserverbar chancevariabel, U, op gennem S-DAGen, vil det ikke ændre beregningerne af de forventede nytter for beslutningerne i S-DAGen, så længe sum-marginaliseringsoperatoren over U kommuterer med de øvrige marginaliseringsoperationer, der skal udføres for at løse S-

DAGen. Dette kan være en anden måde at anskue flytningen af uobserbare chanchvariable på.

Resultatet af at flytte de uobserverbare variable op gennem NFS-DAGen i figur 5.2 kan ses på NFS-DAGen på figur 5.6. (På denne figur har vi samtidigt udnyttet, at ordningen af chancevariable er underordnet, hvorfor de uobserverbare chancevariable, som vi har flyttet op gennem S-DAGen kan repræsenteres i samme S-DAG chanceknude, som den ellers ville være nabo til.)



Figur 5.6: En NFS-DAG for UIDet på figur 5.1. På NFS-DAGen har vi medtaget de uobserverbare variable tidligere. På alle knuder er *Rel* og de frigivne nytteknuder angivet.

Som en S-DAG var defineret i definition 2.2 på side 25, indeholdt alle maksimalt orienterede stier i en S-DAG for et UID, Γ, netop alle observerbare chancevariable og beslutningsvariable i Γ. Definitionen lod implicit de uobserverbare variable i Γ være placeret på $Sink$. Når vi flytter de uobserverbare variable op gennem S-DAGen, gør vi samtidigt deres placering i S-DAGen *eksplicit*.

Fordelen ved at flytte en uobserverbar chancevariable, U, op gennem S-DAGen på denne måde er, at vi således kan få reduceret $FtrRel(\mathcal{K})$ for alle de knuder, $\mathcal{K}$, som vi flytter U op forbi. Dette vil ske i de situationer, hvor en variabel $V \in FtrRel(\mathcal{K})$ kun er i $FtrRel(\mathcal{K})$ fordi, den var i $Rel(U)$. Og når vi får reduceret $FtrRel(\mathcal{K})$, kan vi samtidigt få reduceret $Vars(\mathcal{K})$ samt $SContext(\mathcal{K})$ og dermed pladskravene for den cache vi kan tilføje til knuderne i S-DAGen.

Nu hvor vi har styr på, hvad der bestemmer konteksten for en S-DAG-knude, kan vi se på, hvordan vi kan bruge en cache defineret over konteksten til at genbruge beregninger under S-DAG konditionering.

5.3.4 Genbrug v.h.a. cache

Der er flere af de rekursive kald, der udføres under løsningen af en S-DAG, der resulterer i de samme værdier. Vi har identificeret to forskellige situationer, der begge giver anledning til at de samme MFN værdier bliver beregnet. Disse to situationer vil vi nu beskrive.

Redundans p.g.a. irrelevant fortid Vi betegner den irrelevante fortid for et (del)scenarie, der starter med D, ved $Irrel(D)$, der er givet ved

$$Irrel(D) = Past(D) \backslash Vars(D). \tag{5.13}$$

En situation, hvor to udregninger af den MFN giver samme værdi, er, hvis to forskellige instantieringer over fortiden for en beslutningsvariabel, D, kun adskiller sig mht. instantieringen over variable i den irrelevante fortid, $Irrel(D)$.

Lad os finde $SContext\,(D3)$ for $D3$ (i den øverste sti) i NFS-DAGen på figur 5.6. Konteksten er defineret ved udtryk (5.10). Til at bestemme denne skal vi identificere $Past(D3)$, der direkte kan aflæses af NFS-DAGen til $Past(D3) = \{E, D, B, D2, C, D1, A, F, I, H\}$, og vi skal identificere $Vars(D3)$. $Vars(D3)$ er foreningsmængden af den relevante fortid for variablene i fremtiden for $D3$ (kaldet $FtrRel\,(D3)$) og domænet af de nytteknuder, der er knyttet til de samme knuder.

En analyse af den relevante fortid for variablene i fremtiden for $D3$, dvs. $D3$, G, $D5$ og $Sink$ giver $Rel(D3) = \{A\}$, $Rel(G) = \{D3\}$, $Rel(D5) = \{A, G\}$ og $Rel(Sink) = \emptyset$ (se evt. figur 5.6 på modstående side). Dette giver $FtrRel\,(D3) = \{A, D3, G\}$. Dette forener vi med domænet af nyttepotentialet $U4$, der knytter sig til $D5$, og får $Vars(D3) = \{A, D3, G\} \cup \{A, G, D5\} = \{A, D3, G, D5\}$. Vi får nu $SContext\,(D3) = Vars(D3) \cap Past(D3) = \{A, D3, G, D5\} \cap \{E, D, B, D2, C, D1, A, F, I, H\} = \{A\}$.

Cachen for $D3$ er således defineret over A. Alle udregningerne efter $D3$ giver således den samme værdi for $D3$, så længe A er betinget til den samme værdi. Da fortiden for $D3$ indeholder 9 irrelevante variable ($Irrel(D3) = \{E, D, B, D1, D2, C, F, I, H\}$), og da alle variable i vores eksempel er binære, vil $\rho_{D3}(A = a_1)$ blive beregnet $2^9 = 512$ gange, hvis der ikke bruges cache. Disse udregninger vil alle give det samme resultat. Hvis man gemte resultatet af den første beregning, kunne dette resultat blive genbrugt de efterfølgende 511 gange. Dermed kan alle de rekursive kald, der ellers ville have beregnet $\rho_{D3}(A = a_1)$ 511 ekstra gange, blive erstattet af 511 opslag i cachen.

På figur A.3 på side 150 i bilag A har vi angivet konteksten for alle knuderne i NFS-DAGen på figur 5.6.

Stier med fælles knuder Den forventede nytte for politikken delscenarie afhænger ikke af *ordningen* over variablene i konteksten, men kun af *konfigurationen* over variablene i denne. Hvis vi ser på $D3$ fra før, så er der to stier ned til denne beslutning. Udregningen, der ligger til grund for værdierne i cachen for $D3$, er uafhængig af hvilken sti, der er blevet betinget ned igennem, så længe betingelserne på variablene i $SContext\,(D3)$ er de samme.

I situationen fra før, hvor vi gemte $\rho_{D3}(A = a_1)$, bliver det nu endnu mere fordelagtigt at gemme dette resultat, da der går to stier ned til $D3$, hvorfor værdien for $cache_{D3}(A = a_1)$ skal beregnes dobbelt så mange gange, hvis den ikke bliver gemt i cachen.

Inden vi giver den nye algoritme for S-DAG konditionering, skal vi se på endnu en måde, hvorpå vi ofte kan reducere antallet af variable i $SContext\,(\mathcal{K})$ for en S-DAG-knude $\mathcal{K}$.

5.3.5 Fremskyd observationer til de bliver relevante

I afsnit 2.2.1 på side 27 beskrev vi en række krav til en S-DAG, der garanterede at S-DAGen ville indeholde alle de temporale ordninger, der kunne indgå i en optimal strategi. En S-DAG, der overholdte disse krav, kaldte vi en NFS-DAG. Her brugte vi bl.a. en observation, der foreslog, at alle observationer skulle observeres så snart, de var blevet frigivet, da den forventede nytte aldrig kunne stige ved at udsætte en observation.

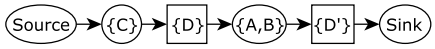
Dette resulterer bl.a. i, at A skal være den første chanceknude i NFS-DAGen for UIDet på figur 5.3, da A er en gratisobservation.

A er imidlertid ikke relevant for D, hvorfor det ikke er nødvendigt at beregne den MFN for D for alle værdier af A. Dette er imidlertid hvad algoritmen i afsnit 5.2.1 gør, da den betinger dybde først gennem NFS-DAGen.

Vi kan derfor flytte A frem, så den observeres efter D. Da A er relevant for D', placerer vi A før D' – som vist på figur 5.7. På denne måde fremskyder vi sum-marginaliseringen af A, indtil denne er relevant for resultatet. Dette medfører samtidigt en reduktion af konteksten for D, da den nu ikke længere vil indeholde A.

Generelt gælder det, at vi kan fremskyde alle observerbare variable i en S-DAG, til de bliver relevante for udregningerne. Denne operation kan illustreres ved, at man på en S-DAG "løsner" alle observerbare chancevariable, og lader dem "falde" imod $Sink$. De observerbare chancevariable skal sætte sig fast i chanceknuden umiddelbart inden den første beslutningsknude, hvori de indgår i den relevante fortid. Hvis en chancevariabel ikke indgår i den relevante fortid for nogle af de efterfølgende beslutninger (som fx B i figur

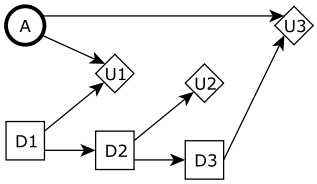
5.3), så lader vi den bare blive siddende.



Figur 5.7: En NFS-DAG for UIDet på figur 5.3, hvor den observerbare chancevariabel A er fremskudt til den bliver relevant. Da B ikke bliver relevant senere i den temporale ordning, er den placeret, dér hvor den frigives, da vi således får frigivet $U1(B, C, D)$ så tidligt som muligt.

Det vil ikke medføre nogle ændringer på NFS-DAGen på figur 5.2, at fremskyde observationer. Men vi forventer at denne operation i andre situationer kan forbedre algoritmens kørselstid. Som fx i eksemplet i figur 5.3. I dette eksempel medfører operationen også, at $SContext\,(D)$ bliver reduceret fra $\{A, C\}$ til $\{C\}$.

Ved denne operation kan $SContext\,(\mathcal{K})$ blive reduceret. $SContext\,(\mathcal{K})$ bliver dog aldrig mindre end $Rel(\mathcal{K})$, ligesom vi heller ikke er garanteret at nå ned til denne mængde. Et eksempel hvor $Rel(\mathcal{K})$ er en ægte delmængde af $SContext\,(\mathcal{K})$ kan ses på figur 5.8.



Figur 5.8: På dette UID er A i den relevante fortid for $D1$ og $D3$ men ikke for $D2$. A skal dog stadig være i $SContext\,(D2)$, da udregningerne for $D3$ afhænger af instantieringen over A, og A er i fortiden for $D2$. Da $Rel(D2) = \emptyset$ og $SContext\,(D2) = \{A\}$, er $Rel(D2)$ således en ægte delmængde af $SContext\,(D2)$.

Operationerne med at fremskyde observationer, til de bliver relevante, og flytning af uobserverbare variable op gennem S-DAGen kan udføres på en S-DAG, inden den løses med fx S-DAG konditionering.

Vi har valgt først at flytte de observerbare variable frem, så de placeres umiddelbart før første beslutningen, for hvilken de indgår i den relevante fortid. Under denne operation er de uobserverbare variable placerede på $Sink$. Herefter flytter vi de uobserverbare variable tilbage gennem S-DAGen som beskrevet i afsnit 5.3.3. Endelig bestemmes placeringen af nytteknuderne.

Yderligere undersøgelser vil formentligt kunne afgøre, om rækkefølgen af disse operationer er betydende for den resulterende S-DAG, og hvilken rækkefølge der i så fald er optimal.

5.3.6 Algoritme – med cache

Vi er nu klar til at udbygge vores pseudokode for løsningsalgoritmen, så den nu gør brug af cache. Pseudokoden for RcSummMar($\mathcal{C}$) er vist i algoritme 3. Den implementerer udtryk (5.12), hvilket kan ses i linie 9, der nu, ved et kald til RcUtil($\mathcal{C}$)), finder summen af nytten fra de netop frigivne nytteknuder og adderer denne sum med resultatet af at udføre maxkombinering over alle beslutningsbørn efter den aktuelle chanceknude (hvilket sker i RcMaxComb($\mathcal{C}$)). På samme måde laver RcMaxMar(D) også et nytteopslag i linie 7.

Funktionen RcUtil($\mathcal{K}$) er blevet ændret, så den nu ikke længere finder nytten over alle nyttepotentialer, men kun over de nyttepotentialer, der netop er blevet frigivet.

Endelig er RcSummMar($\mathcal{C}$) og RcMaxMar(D) udvidet med en cache. Cachen er implementeret på samme måde som vi udvidede RC algoritmen, ved først at tjekke om resultatet af den ønskede udregning ligger i cachen (linie 2), og i så fald returnere denne værdi. Hvis resultatet ikke ligger i cachen, så beregnes det, og til sidst tjekkes, om resultatet skal gemmes i cachen (linie 11 i RcSummMar og linie 9 i RcMaxMar). For at denne fremgangsmåde skal virke, skal alle cacheindgange initialiseres til "nil".

Vi har valgt ikke at gemme værdierne, som RcUtil($\mathcal{C}$) returnerer, da disse værdier blot er en række opslag i nyttepotentialer, og disse opslag vil i worst case kunne laves linært i antallet af knuder i UIDet. Vi gemmer heller ikke resultaterne af RcMaxComb($\mathcal{C}$), da denne funktion blot er en hjælpefunktion til RcSummMar($\mathcal{C}$), og RcSummMar($\mathcal{C}$) har sin egen cache.

5.4 Beregning af sandsynligheder

I udtryk (5.12) skal vi bruge en betinget sandsynlighed af typen $P(c_k|d_k, rel(D_k))$. Da problemet med at finde denne fordeling kan ses som et rent sandsynlighedsinferensproblem, kan vi anvende RC til dette problem. Dette vil dog kræve en modifikation af UIDet, så det bliver et bayesiansk net. I afsnit 5.4.1 skal vi se på, hvordan vi kan lave denne modifikation. I afsnit 5.4.2 skal vi se på, hvordan sandsynligheden kan findes ud fra det nye bayesianske net.

5.4.1 Konvertering af UIDer til bayesianske net

Vi skal nu se på, hvordan vi kan opstille et bayesiansk net ud fra et UID, som vi kan bruge til at finde vores betingede sandsynligheder ud fra.

Hvis vi har et bayesiansk net, der består af to variable, A og B, hvor A

Algoritme 3 S-DAG konditionering med cache. Funktionerne erstatter de tilsvarende funktioner i algoritme 2.

RcSumMar($\mathcal{C}$)
Input: $\mathcal{C}$ - *En S-DAG-knude med chancevariable.*
1: $\mathbf{y} \leftarrow$ *de registrerede instantieringer over* $SContext\,(\mathcal{C})$
2: **if** $cache_{\mathcal{C}}\,[\mathbf{y}] \neq \text{nil}$ **then return** $cache_{\mathcal{C}}\,[\mathbf{y}]$
3: $v \leftarrow 0$
4: **for all** *konfigurationer,* $\mathbf{c}$*, over* $\mathcal{C}$ **do**
5: *Registrér instantieringen* $\mathbf{c}$
6: **if** $\mathcal{C}$ *er* $Sink$ **then**
7: $v \leftarrow v + \text{RcCondProb}(\mathcal{C}) \cdot \text{RcUtil}(\mathcal{C})$
8: **else**
9: $v \leftarrow v + \text{RcCondProb}(\mathcal{C}) \cdot (\text{RcUtil}(\mathcal{C}) + \text{RcMaxComb}(\mathcal{C}))$
10: *Afregistrér instantieringen* $\mathbf{c}$
11: **if** $cache?\,(\mathcal{C}, \mathbf{y})$ **then** $cache_{\mathcal{C}}\,[\mathbf{y}] \leftarrow v$
12: **return** v

RcUtil($\mathcal{K}$)
Input: $\mathcal{K}$ - *En S-DAG-knude.*
1: $\Psi_{\mathcal{K}} \leftarrow$ *De nytteknuder der frigives af* $\mathcal{K}$.
2: $\mathbf{x} \leftarrow$ *De registrerede instantieringer over potentialerne i* $\Psi_{\mathcal{K}}$.
3: **return** $\sum \Psi_{\mathcal{K}}(\mathbf{x})$.

RcMaxComb($\mathcal{C}$)
Input: $\mathcal{C}$ - *En S-DAG-knude med chancevariable.*
1: $v \leftarrow -\infty$
2: **for all** *børn,* D_i*, af* $\mathcal{C}$ **do**
3: $v \leftarrow \max\{v, \text{RcMaxMar}(D_i)\}$
4: **return** v

RcMaxMar(D)
Input: D - *En S-DAG-knude med en beslutning.*
1: $\mathbf{y} \leftarrow$ *de registrerede instantieringer over* $SContext\,(D)$
2: **if** $cache_D\,[\mathbf{y}] \neq \text{nil}$ **then return** $cache_D\,[\mathbf{y}]$
3: $\mathcal{C} \leftarrow$ *S-DAG-knuden med de chancevariable som* D *frigiver.*
4: $v \leftarrow -\infty$
5: **for all** *konfigurationer,* $\mathbf{c}$*, over* D **do**
6: *Registrér instantieringen* $\mathbf{c}$.
7: $v \leftarrow \max\{v, \text{RcUtil}(D) + \text{RcSumMar}(\mathcal{C})\}$
8: *Afregistrér instantieringen* $\mathbf{c}$.
9: **if** $cache?\,(D, \mathbf{y})$ **then** $cache_D\,[\mathbf{y}] \leftarrow v$
10: **return** v

er forælder til B, vil den specificerede sandsynlighedsfordeling over A indgå i beregningen af marginalsandsynlighedsfordelingen over B, $P(B)$, men den specificerede sandsynlighedsfordeling over B vil ikke indgå i beregningen af marginalsandsynlighedsfordelingen over A, da B er barren. Hvis B har børn i det bayesianske net, vil den specificerede sandsynlighedsfordeling over B kun indgå i beregningen af marginalsandsynlighedsfordelingen over A, hvis der er evidens på mindst et af Bs børn eller B selv, således at B ikke er barren.

Da vi skal bruge betingede sandsynligheder på formen $P(\mathbf{c}_k|d_k, rel(D_k))$, hvor $\{D_k\} \cup Rel(D_k)$ altid er knuder, der er *forfædre* til knuderne i $\mathcal{C}_k$, gælder det, at sandsynlighedsfordelingen over knuderne der er *efterfølgere* af $\mathcal{C}_k$ aldrig vil have indflydelse på sandsynligheden $P(\mathbf{c}_k|d_k, rel(D_k))$, da alle disse efterfølgere vil være barren.

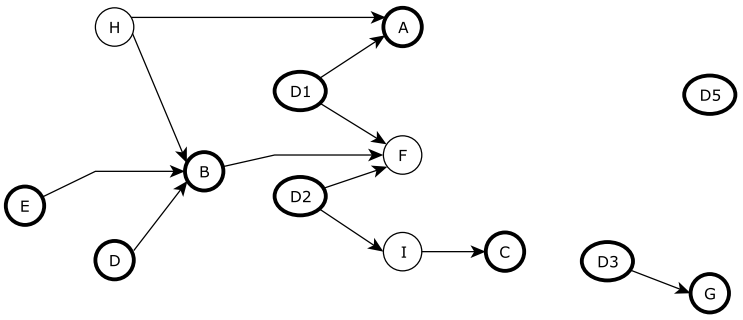
Det betyder, at hverken domænet for eller sandsynlighedsfordelingen over en knude, der er efterfølger til $\mathcal{C}_k$, vil kunne have indflydelse på sandsynligheden $P(\mathbf{c}_k|d_k, rel(D_k))$.

Med dette i mente kan vi nu konvertere et UID til et bayesiansk net ved at fjerne alle nytteknuder, samt ikke kausale kanter og konvertere alle beslutningsknuder til chanceknuder med en vilkårlig sandsynlighedsfordeling.

Når vi fjerner alle ikke kausale kanter, bliver alle beslutningsknuderne forældreløse, og deres sandsynlighedsfordelinger bliver kun over tilstandene i beslutningsknuden selv. Således bliver det bayesianske nets strukturelle opbygning uafhængig af, hvilke knuder der er i fortiden for beslutningsknuderne.

Sandsynlighedsfordelingen over de konverterede beslutningsknuder kan vi vælge vilkårligt, da de konverterede beslutninger enten er (1) i den relevante fortid for den betingede sandsynlighed, som vi vil finde eller de er (2) i efterfølgere til de variable, $\mathcal{C}_k$, som vi vil finde den betingede sandsynlighed for. Hvis de er (1) i den relevante fortid, så er de instantieret, og fordelingen er i så fald ligegyldig, og hvis de er (2) efterfølgere til $\mathcal{C}_k$, er fordelingen irrelevant, da der ikke er evidens på nogle variable efterfølgende $\mathcal{C}_k$. Og en sådan evidens er nødvendig, hvis sandsynlighedsfordelingen over de efterfølgende konverterede beslutningsknuder skulle have indflydelse på $P(\mathbf{c}_k|d_k, rel(D_k))$ (dvs. hvis de konverterede beslutningsknuder ikke skal være barren).

Figur 5.9 viser strukturen for det bayesianske net, der fremkommer ved at konvertere UIDet fra figur 5.1 til et bayesiansk net med den metode, som vi netop har beskrevet. I et bayesiansk net er alle knuder chanceknuder. Alligevel har vi her valgt grafisk at gøre forskel på de knuder, der var uobserverbare chanceknuder i UIDet og de øvrige knuder.



Figur 5.9: Det bayesiansk net, der fremkommer ved at fjerne alle ikke kausale kanter fra UIDet på figur 5.1, fjerne alle nytteknuder og konvertere alle beslutningsknuder til chanceknuder. Vi har her valgt grafisk at gøre forskel på de knuder, der var uobserverbare chanceknuder i UIDet (tynde cirkler) og de andre knuder (fede cirkler).

5.4.2 R-DAG konditionering

Nu hvor vi har et bayesiansk net for UIDet, er vi klar til at bestemme de betingede sandsynligheder, der skal bruges ved løsningen af S-DAGen.

I RC finder vi fællessandsynligheder på formen $P(\mathbf{b})$, men vi skal bruge en betinget sandsynlighed på formen $P(\mathbf{b}|past(\mathcal{B}))$. Hvis vi vil bruge RC til at finde denne sandsynlighed vha. de bayesianske net, som vi kan lave ved at konvertere vores UIDer, som beskrevet i afsnit 5.4.1, så kan vi både finde $P(\mathbf{b}, past(\mathcal{B}))$ og $P(past(\mathcal{B}))$ og herefter bestemme den betingede sandsynlighed ved den fundamentale regel:

$$P(\mathbf{b}|past(\mathcal{B})) = \frac{P(past(\mathcal{B}), \mathbf{b})}{P(past(\mathcal{B}))}. \tag{5.14}$$

Bemærk at vi vil beskrive, hvordan vi finder $P(\mathbf{b}|past(\mathcal{B}))$ og ikke, hvordan vi finder $P(\mathbf{b}|rel(\mathcal{B}))$. Dette kan vi tillade os, da $P(\mathbf{b}|past(\mathcal{B})) = P(\mathbf{b}|rel(\mathcal{B}))$. I flere situationer vil det kræve færre beregninger at bestemme $P(\mathbf{b}|rel(\mathcal{B}))$ end det kræver at bestemme $P(\mathbf{b}|past(\mathcal{B}))$ – fx hvis det bayesianske net er opdelt i øer, som det er tilfældet på figur 5.9. Disse situationer vil vi dog ikke behandle i denne rapport.

Formelt set er alle variablene i det bayesianske net nu chancevariable. Men vi vil beskrive variablene i det bayesianske net som observerbare-, uobserverbare og beslutningsvariable og med disse betegnelser referere tilbage til knudernes type i UIDet.

I dette afsnit vil vi beskrive, hvordan vi kan finde fællessandsynligheder på formen $P(past(\mathcal{B}), \mathbf{b})$.

En mulighed er at lave ét d-træ ud fra det bayesianske net og anvende RC algoritmen på dette. Der er imidlertid nogle barrenegenskaber, som vi i det følgende vil prøve at udnytte med henblik på at reducere antallet af variable, der skal konditioneres over for at bestemme fællessandsynlighederne.

Når vi vil finde sandsynligheden for $P(past(\mathcal{B}), \mathbf{b})$, gælder det, at alle de observerbare chancevariable og beslutningsvariable, der ikke er evidens på, vil være barren, da disse variable kommer efter $\mathcal{B}$ i den temporale ordning og derfor ikke vil være instantierede. Vi kan således se bort fra sandsynlighedsfordelingerne specifiseret for disse knuder. De eneste sandsynlighedsfordelinger, der kan have indflydelse på sandsynligheden $P(past(\mathcal{B}), \mathbf{b})$ er sandsynlighedsfordelingerne over de uobserverbare chancevariable, der ikke er barren eller instantierede, når der er evidens på $Past(\mathcal{B}) \cup \mathcal{B}$.

Vi forventer, at vi kan opnå en algoritme med bedre kørselstid end en algoritme, der laver konditionering på et d-træ over det originale bayesianske net, hvis vi forsøger at udnytte disse barrenegenskaber. Således kan vi nemlig afskære dele af det bayesianske net, og ikke medtage disse dele, når vi skal udregne vores sandsynligheder.

Der vil ikke nødvendigvis blive taget højde for disse barrenegenskaber, hvis vi blot anvendte RC algoritmen på et vilkårligt d-træ over det originale bayesianske net. Dette kan medføre, at der bliver konditioneret over barrenvariable. Et eksempel ville være, hvis $P(B = b_1)$ skulle beregnes ud fra d-træet på figur 4.2 på side 58. Når denne sandsynlighed skal findes, er D barren. RC algoritmen ville imidlertid konditionere over D, når $P(B = b_1)$ skal beregnes ud fra dette d-træ. Da konditionering over barrenvariable vil være redundante, er det unødvendigt at konditionere over D. I stedet for kunne man starte med at fjerne alle barren-variable, og lave et nyt d-træ over det resterende net. Men det vil betyde, at der skal laves ét d-træ for hver chanceknude i S-DAGen, da mængden af barren-variable er afhængig af hvilke variable, der er evidens på. Disse d-træer ville optage unødig hukommelse, hvis de alle skal gemmes samtidigt, og da vi søger at minimere pladsforbruget for den samlede løsningsalgoritme, mener vi ikke, at dette er en acceptabel løsning. Alternativt kunne d-træerne genereres efter behov, men det ville give et overhead med at generere d-træerne, hver gang der skal bestemmes én sandsynlighed, og mange af de generede d-træer vil være ens.

Vi vil i stedet for opbygge én grafstruktur, en *rekursions DAG*, R-DAG, der holder styr på, hvornår hvilke variable er barren, når vi lægger evidens på variablene gennem de temporale ordninger, der er repræsenteret ved stierne i en S-DAG. Ved således at have én struktur til at repræsentere alle disse oplysninger, forventer vi at opnå en hurtigere algoritme end én der genererer d-træerne efter behov, og en mindre pladskrævende algoritme end én, der

gemmer alle de forskellige d-træer.

Med udgangspunkt i en S-DAG, Ξ, over et UID, Γ, fjerner vi nu alle uobserverbare variable fra deres aktuelle placering i Ξ. Vi inkluderer dem nu i nye knuder, som vi tilføjer til Ξ, så vi får dannet en ny graf – vores R-DAG. Knuderne med uobserverbare chancevariable skal indskydes imellem eksisterende S-DAG-knuder. Det skal gælde at, de uobserverbare chancevariable, $\mathcal{A}$, der skal indskydes umiddelbart før S-DAG-knuden $\mathcal{B}$, alle skal være barren, hvis der er evidens på alle variable i $Past(\mathcal{B})$ i S-DAGen. Samtidigt skal det gælde, at variablene i $\mathcal{A}$ *ikke* er barren, hvis der er evidens på variablene i $Past(\mathcal{B}) \cup \mathcal{B}$. Således er $\mathcal{B}$ bestemmende for, om $\mathcal{A}$ er barren.

Sandsynlighedspotentialerne til de uobserverbare chancevariable $\mathcal{A}$ vil således ikke have indflydelse på sandsynligheden $P(past(\mathcal{B}))$ men vil have indflydelse på sandsynligheden $P(past(\mathcal{B}), \mathbf{b})$.

Vi kan identificere de uobserverbare chancevariable, $\mathcal{A}$, der skal inkluderes umiddelbart inden S-DAG-knuden $\mathcal{B}$ ved

$$\mathcal{A} = \{H \in \mathcal{H} | Past(\mathcal{B}) \cap Ek(H) = \emptyset \wedge \mathcal{B} \cap Ek(H) \neq \emptyset\}, \tag{5.15}$$

hvor $\mathcal{H}$ er alle de uobserverbare chancevariable i Γ, og $Ek(H)$ er de variable, der er efterkommere til H i Γ og $Past(\mathcal{B})$ er de variable der er før $\mathcal{B}$ i den temporale ordning som Ξ dikterer. Formlen dikterer, at $\mathcal{A}$ består af de variable, H, der har en efterkommer iblandt $\mathcal{B}$, og ingen efterkommere iblandt variablene i $Past(\mathcal{B})$.

De uobserverbare variable, der er barren uanset hvilken evidens, der lægges på observerbare chancevariable og beslutninger, placeres i de samme knuder, som de også er placeret i i S-DAGen. Mens de uobserverbare variable, der ikke er barren, givet evidens på gratisobservationerne, placeres i *Source*.

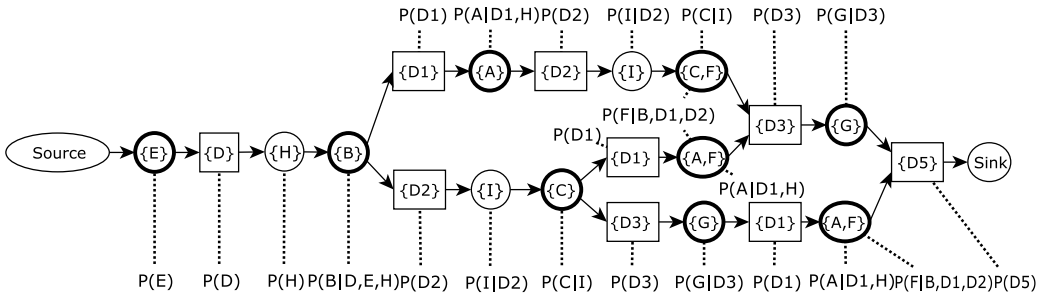
Vi definerer en R-DAG i det følgende:

Definition 5.3 (**R-DAG**). Lad Γ være et UID, og lad $\mathcal{N}$ være det bayesianske net, der fremkommer, ved at konvertere UIDet som beskrevet i afsnit 5.4.1. Lad Ξ være en S-DAG for Γ. En R-DAG, $\mathcal{R}$, for Ξ er da grafstrukturen fra Ξ, hvor de uobserverbare knuder fra Γ er indskudt mellem knuderne i Ξ, således at deres placering overholder udtryk (5.15). Til hver variabel, X, i $\mathcal{R}$ er der knyttet det samme sandsynlighedspotentiale, som der er knyttet til X i $\mathcal{N}$. □

R-DAGen for S-DAGen på figur 5.2 kan ses på figur 5.10.

Det gælder at

Sætning 5.1. Ingen uobserverbar variabel bliver placeret senere i den temporale ordning i R-DAGen end variablens tilsvarende placering i S-DAGen.



Figur 5.10: En R-DAG for S-DAGen på figur 5.6.

Bevis. Hvis en variabel, H, skulle placeres senere i den temporale ordning i R-DAGen end dens placering i S-DAGen, så skulle det gælde, at H er barren, når der er evidens på alle variablene ned til og med Hs placering i S-DAGen. Ingen efterkommere af H i det bayesianske net må således komme tidligere end H i S-DAGen. Dette vil aldrig gælde, da H i S-DAGen netop er placeret, så den kommer efter alle de knuder, som den er betinget afhængig af – også efter dens umiddelbare efterkommere (dens børn). Hvis H ingen børn havde, kom den på samme plads i R-DAGen, som den havde i S-DAGen. □

Med en R-DAG, $\mathcal{R}$, for S-DAGen Ξ kan vi nu finde sandsynligheder af typen $P(\mathbf{b}, past(\mathcal{B}))$, hvor $\mathcal{B}$ er de variable, der er i én knude i S-DAGen, og $past(\mathcal{B})$ er instantieringen over variablene mellem $Source$ og $\mathcal{B}$ i Ξ. Fremover vil mængden $Hst(\mathcal{X})$ betegne $\mathcal{X} \cup Past(\mathcal{X})$.

Vi bruger notationen $Hst_G(\mathcal{X})$ til at betegne alle de variable, der er i $\mathcal{X}$ og i $Past(\mathcal{X})$ i grafen G. G kan være en S-DAG eller en R-DAG. Ligeledes bruger vi notationen $hst_G(\mathcal{X})$ til at betegne en konfiguration over variablene i $Hst_G(\mathcal{X})$. Vi vil også bruge et indeks på $Past(\mathcal{X})$ så vi får $Past_G(\mathcal{X})$, hvis der kan være tvivl om for hvilken graf der henvises til for fortiden af $\mathcal{X}$ i.

Når variablene er ordnet efter en temporal ordning (angivet ved en sti i en Ξ), og der udelukkende er evidens på variablene i $Hst_\Xi(\mathcal{B})$, gælder det, at alle variable, der følger $\mathcal{B}$ i R-DAGen for Ξ er barren. Alle disse variables sandsynlighedsfordelinger behøver således ikke at blive medregnet, når vi vil finde sandsynligheden $P(hst_\Xi(\mathcal{B}))$.

Vi kan nu finde $P(hst_\Xi(\mathcal{B}))$ ved at lave konditionering på $\mathcal{R}$. Konditionering på en R-DAG, minder om RC. Vi kan betragte en orienteret sti, s, fra $Source$ til $\mathcal{B}$ som ét d-træ, hvor hver knude, $\mathcal{K}$, er et træ, hvor den ene gren er produktet af alle de potentialer, der er tilknyttet $\mathcal{K}$ og den anden gren er barnet til $\mathcal{K}$ på stien s. Knuden $\mathcal{B}$ angiver hvor langt ned gennem stien i R-DAGen vi skal konditionere. Vi kalder derfor $\mathcal{B}$ for *termineringsknuden*.

Vi kan vælge en vilkårlig maksimalt orienteret sti fra $Source$ til $\mathcal{K}$ som vores d-træ. Variablene i en knude $\mathcal{K}$ på stien s svarer til $Cutset(\mathcal{K})$. Når vi laver konditionering ned gennem s, bruger vi de instantieringer, der allerede er lavet i S-DAGen og konditionerer over de resterende uinstantierede variable. Dvs. at de variable, der skal konditioneres over i R-DAGen er de uobserverbare chancevariable, der er i fortiden for termineringsknuden i R-DAGen men ikke i fortiden for termineringsknuden i S-DAGen.

Når det rekursive forløb gennem $\mathcal{R}$ når til $\mathcal{B}$, ved vi, at konditionering videre ned gennem efterfølgerne af $\mathcal{B}$ vil være konditionering over barren variable, og vi kan således stoppe gennemløbet af stien i $\mathcal{R}$ ved $\mathcal{B}$.

Eksempel Hvis termineringsknuden er $\{G\}$ på den nederste sti i R-DAGen, $\mathcal{R}$, på figur 5.10, skal vi konditionere over stien, der er markeret på figur 5.11. Når vi konditionerer på denne sti, svarer det til at lave konditionering over d-træet på figur 5.12. På denne måde kan vi finde sandsynligheden $P(hst_{\Xi}(\{G\}))$, hvor $hst_{\Xi}(\{G\})$ er en instantiering over variablene i fortiden til $\{G\}$ i S-DAGen, Ξ, på figur 5.2.

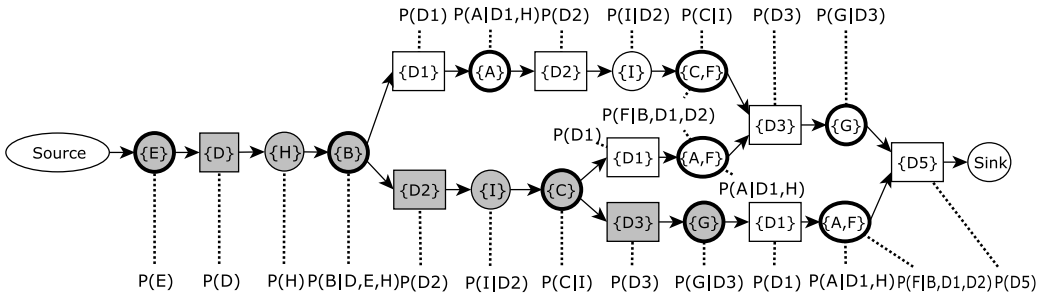
Når denne sandsynlighed skal bestemmes, er der evidens på alle variablene i $Hst_{\Xi}(\{G\})$. De eneste uinstantierede variable er således dem, der er blevet flyttet op forbi termineringsknuden $\{G\}$, da $\mathcal{R}$ blev lavet ud fra Ξ. Disse variable er givet ved:

$$\begin{aligned} &Hst_{\mathcal{R}}(\{G\}) \backslash Hst_{\Xi}(\{G\}) \\ &= \{E, D, H, B, D2, I, C, D3, G\} \backslash \{E, D, B, D2, C, I, D3, G\} \\ &= \{H\}. \end{aligned}$$

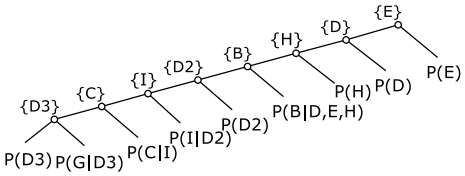
Vi skal altså kun konditionere over H for at bestemme vores fællessandsynlighed.

I en R-DAG er der en mængde potentialer på alle knuderne. Disse indeholder sandsynlighedsfordelingerne fra det bayesianske net for de variable, der er i knuderne. Det var et krav for et d-træ, at variablene i $Cutset(T)$ for et træ, T, var givet ved fællesmængden over variablene i potentialerne i bladene under T^l og T^r, hvor T^l og T^r er de to børn til T. Til forskel fra et d-træ, kan vi i en R-DAG ikke være sikre på, at alle variable i $\mathcal{K}$ også indgår i potentialerne til knuderne længere nede i R-DAGen. Et eksempel er knuden $\{B\}$ i d-træet på figur 5.12. Variablen B optræder ikke i potentialerne for $T^l(\{B\})$*. Ved at definere $Cutset(\mathcal{K})$ for en R-DAG-knude $\mathcal{K}$ som variablene

*Dette ville imidlertid være tilfældet, hvis termineringsknuden var $\{A, F\}$. Potentialet for F ville således være en af efterkommeren til $\{B\}$, og da B er forælder til F, ville B optræde i potentialet, der knyttede sig til F.



Figur 5.11: En R-DAG, $\mathcal{R}$, for S-DAGen, Ξ, på figur 5.6. På denne R-DAG har vi markeret de knuder, der udgør vores konditioneringsstruktur, der skal bruges til at bestemme sandsynligheden $P(hst_{\Xi}(\{G\}))$.



Figur 5.12: Et d-træ for stien i R-DAGen på figur 5.11, der er markeret med grå.

i $\mathcal{K}$, gælder egenskaben fra d-træet, om at variablene i cutsettet for en intern knude T også indgår i $Vars(T^l)$ og $Vars(T^r)$, ikke længere. Således er træet på figur 5.12 ikke et "rigtigt" d-træ. Vi kan dog stadig beregne sandsynlighederne ved konditionering over variablene i cutsettene. Ulempen ved at gøre det på denne måde er, at vi kan risikere at skulle lave flere rekursive kald, end hvis vi havde konditioneret på et d-træ. Ideen med at konditionere på en sti i R-DAGen er, at vi slipper for overheadet med at konditionere over et d-træ med barren-variable, eller alternativt lave (og evt. gemme) alle d-træerne, der tager højde for disse barren-egenskaber. Yderligere forskning vil kunne afgøre, hvornår det vil være en fordel med hhv. R-DAG konditionering og d-træer.

Genbrug af beregninger under udregning af en sandsynlighed i en R-DAG Givet en R-DAG, $\mathcal{R}$, for en S-DAG, Ξ, kan vi lave R-DAG konditionering som netop beskrevet, når vi skal finde en sandsynlighed $P(hst_{\Xi}(\mathcal{B}))$. Dette vil kræve, at vi konditionerer over de uinstantierede (uobserverbare) variable givet ved $Hst_{\mathcal{R}}(\mathcal{B}) \backslash Hst_{\Xi}(\mathcal{B})$. Disse variable betegner vi $Cond(\mathcal{B})$. For en mængde varible, $\mathcal{X}$, betegner $\mathcal{X}^{\#}$ antallet af mulige konfigurationer over variablene i $\mathcal{X}$. Størrelsen af det tilstandsrum der skal konditionere over bliver således $Cond(\mathcal{B})^{\#}$, hvilket er eksponentielt stort i antallet af knuder i

$Cond(\mathcal{B})$.

Vi vil nu se, om vi kan reducerer denne kompleksitet, ved at indføre caching på samme måde, som vi så cache anvendt i RC.

Ideen er ligesom ved RC og S-DAG konditionering at, når udregningerne i R-DAGen efter en variabel $H \in Cond(\mathcal{B})$ er uafhængige af instantieringen af H, behøver disse udregninger ikke at blive udført mere end én gang. Udregningen fra den første udregning kan således gemmes og genbruges.

Et eksempel fra figur 5.10 er, at resultatet af kaldet fra $\{B\}$ til $\{D2\}$ er uafhængigt af instantieringen over H, hvorfor resultatet af det første kald fra $\{B\}$ til $\{D2\}$ kan genbruges ved de efterfølgende kald fra $\{B\}$ til $\{D2\}$. Disse kald vil kun adskille sig mht. instantieringen over H, da H er den eneste variabel i fortiden for $\{B\}$, der ved første kald i R-DAGen er uinstantieret (når termineringsknuden er $\{G\}$).

Vi definerer mængden $RContext(\mathcal{K})_{\mathcal{B}}$ som de variable i fortiden for $\mathcal{K}$ i R-DAGen, $\mathcal{R}$, (skrevet $Past_{\mathcal{R}}(\mathcal{K})$) fra $Cond(\mathcal{B})$, der også optræder i domænet for potentialerne fra $\mathcal{K}$ og frem til termineringsknuden $\mathcal{B}$ (kaldet $Vars(\mathcal{K})_{\mathcal{B}}$). Konteksten for en knude $\mathcal{K}$ afhænger således af, hvor "langt ned gennem" R-DAGen $\mathcal{R}$ vi skal konditionerer, specificeret ved termineringsknuden, $\mathcal{B}$, og er givet ved

$$RContext(\mathcal{K})_{\mathcal{B}} = Vars(\mathcal{K})_{\mathcal{B}} \cap Cond(\mathcal{B}) \cap Past_{\mathcal{R}}(\mathcal{K}). \tag{5.16}$$

Bemærk ligheden med definition 4.2 på side 58. I en R-DAG svarer $ACutset\,(T)$ til $Past_{\mathcal{R}}(\mathcal{K})$ og $Vars(T)$ til $Vars(\mathcal{K})_{\mathcal{B}}$. Derudover har vi begrænset konteksten til kun at være defineret over de variable, der er uinstantierede i S-DAGen givet ved $Cond(\mathcal{B})$.

Eksempel Hvis vi skal finde sandsynligheden $P(hst(\{G\}))$ fra før, så er termineringsknuden $\{G\}$ og konteksten for $\{C\}$ bliver således

$$\begin{aligned} RContext(\{C\})_{\{G\}} &= Vars(\{C\})_{\{G\}} \cap Cond(\{G\}) \cap Past_{\mathcal{R}}(\{C\}) \\ &= \{C, I, D3, G\} \cap \{H\} \cap \{E, D, H, B, D2, I\} \\ &= \emptyset. \end{aligned}$$

Konteksten for $\{B\}$ bliver

$$\begin{aligned} RContext(\{B\})_{\{G\}} &= Vars(\{B\})_{\{G\}} \cap Cond(\{G\}) \cap Past_{\mathcal{R}}(\{B\}) \\ &= \{B, D, E, H, D2, I, C, D3, G\} \cap \{H\} \cap \{E, D, H\} \\ &= \{H\}. \end{aligned}$$

Pseudokode for R-DAG konditionering Pseudokoden for RcCondProb-algoritmen (algoritme 4) finder betingede sandsynligheder ved anvendelsen af udtryk (5.14). Når RcCondProb bliver kaldt er der evidens på alle variable fra *Source* til $\mathcal{B}$ i S-DAGen. Dette skyldes, at vi arbejder videre med de registrerede instantieringer, der blev lavet i algoritme 3. Algoritmen laver to kald til RcProb. Et kald giver sandsynligheden for den registrerede instantiering. Således giver det kaldet til RcProb (i linie 1 af RcCondProb) sandsynligheden for den registrerede instantiering, der er ved første kald til RcCondProb, dvs. sandsynligheden for $P(\mathbf{b}, past(\mathcal{B}))$. Herefter fjernes den registrerede instantiering over $\mathcal{B}$ (i linie 3 af RcCondProb) og det efterfølgende kald til RcProb finder nu sandsynligheden for $P(past(\mathcal{B}))$.

Ved igen at indføre instantieringen over $\mathcal{B}$ inden RcCondProb terminerer (linie 6 af RcCondProb), efterlader RcCondProb de samme instantieringer, som der var registreret inden kaldet til RcCondProb.

RcProb finder sandsynlighederne. Pseudokoden minder om pseudokoden for Rc fra algoritme 1, og burde således ikke kræve yderligere forklaring. Dog vil vi pointere, at når vi i linie 4 af hjælpefunktionen RcSum gennemløber alle tilladelige konfigurationer over variablene i en knude $\mathcal{C}$, betyder det, at vi gennemløber alle konfigurationerne over variablene i $Cond(\mathcal{B}) \cap \mathcal{C}$, hvor $\mathcal{B}$ er termineringsknuden - dvs. de uinstantierede uobserverbare variable, der er i $\mathcal{C}$. Der vil således kun være én tilladelig konfiguration over $\mathcal{C}$, hvis $\mathcal{C}$ ikke indeholder nogle variable fra $Cond(\mathcal{B})$.

Algoritmen konditionerer ikke længere ned gennem R-DAGen end til termineringsknuden $\mathcal{B}$, hvilket sikres med tjekket i linie 6 af RcSum. Ved at terminere ved $\mathcal{B}$ udnytter vi vores viden om, at alle de knuder der følger $\mathcal{B}$, de er barren og derfor ikke vil bidrage til den sandsynlighed, vi ønsker at bestemme.

Ignorering af beslutningsvariable under R-DAG konditionering Som psudokoden til udregning af sandsynligheder netop er beskrevet, laves der opslag i de sandsynlighedspotentialer, der er knyttet til knuderne i R-DAGen. Det vil for chancevariable betyde et opslag i de sandsynlighedspotentialer, der er specificeret sammen med UIDet. For beslutningsvariable vil det betyde et opslag i de vilkårlige sandsynlighedspotentialer, som vi genererede, da vi konverterede UIDet til et bayesianske net, som beskrevet i afsnit 5.4.1. I afsnit 5.4.1 argumenterede vi for, at værdierne i disse sandsynlighedspotentialer ikke ville have indflydelse på de udregnede betingede sandsynligheder. Denne konklusion kan vi bruge til at optimere RcProb.

Som vi indtil nu har beskrevet RcProb, laves der opslag i de sandsynlighedspotentialer, der blev lavet til de konverterede beslutningsvariable. Da

Algoritme 4 Finder en betinget sandsynlighed af typen $P(\mathbf{b}|past(\mathcal{B}))$ med rekursiv konditionering over en R-DAG. Inden kaldet til RcCondProb skal den aktuelle instantiering over $\mathcal{B} \cup Past(\mathcal{B})$ være registreret.

RcCondProb($\mathcal{B}$)
Input: $\mathcal{B}$ - *En R-DAG-knude med chancevariable.*
1: $t \leftarrow$ RcProb($\mathcal{B}$).
2: $\mathbf{b} \leftarrow$ *instantieringen over* $\mathcal{B}$.
3: *Afregistrér instantieringen* $\mathbf{b}$.
4: $\mathcal{P} \leftarrow$ *en forælder til* $\mathcal{B}$.
5: $n \leftarrow$ RcProb($\mathcal{P}$).
6: *Registrér instantieringen* $\mathbf{b}$.
7: **return** t/n.

RcProb($\mathcal{B}$)
Input: $\mathcal{B}$ - *R-DAG-knuden, der terminerer gennemløbet i R-DAGen.*
1: *Sæt* $\mathcal{B}$ *til termineringsknude.*
2: $\mathcal{C} \leftarrow$ *Topknuden i R-DAGen.*
3: **return** RcSum($\mathcal{C}$)

RcSum($\mathcal{C}$)
Input: $\mathcal{C}$ - *En R-DAG chanceknude.*
1: $\mathbf{y} \leftarrow$ *de registrerede instantieringer over* $RContext\,(\mathcal{C})$
2: **if** $cache_{\mathcal{C}}\,[\mathbf{y}] \neq \text{nil}$ **then return** $cache_{\mathcal{C}}\,[\mathbf{y}]$
3: $p \leftarrow 0$
4: **for all** *tilladelige konfigurationer,* $\mathbf{c}$*, over variablene i* $\mathcal{C}$ **do**
5: *Registrer instantieringen* $\mathbf{c}$
6: **if** $\mathcal{C}$ *er termineringsknuden* **then**
7: $p \leftarrow p +$ RcProdFac($\mathcal{C}$)
8: **else**
9: $\mathcal{B} \leftarrow$ *Et barn til* $\mathcal{C}$*, der fører ned til termineringsknuden.*
10: $p \leftarrow p +$ RcProdFac($\mathcal{C}$) $\cdot$ RcSum($\mathcal{B}$)
11: *Afregistrér instantieringen* $\mathbf{c}$
12: **if** $cache?\,(\mathcal{C}, \mathbf{y})$ **then** $cache_{\mathcal{C}}\,[\mathbf{y}] \leftarrow v$
13: **return** p

RcProdFac($\mathcal{C}$)
Input: $\mathcal{C}$ - *En R-DAG-knude.*
1: $\Phi_{\mathcal{C}} \leftarrow$ *De potentialer der knytter sig til* $\mathcal{C}$.
2: $\mathbf{x} \leftarrow$ *De registrerede instantieringer over potentialerne i* $\Phi_{\mathcal{C}}$.
3: **return** $\prod \Phi_{\mathcal{C}}(\mathbf{x})$

resultaterne af disse opslag ikke vil influere på den betingede sandsynlighed, der ønskes fundet, kan disse opslag for de konverterede beslutningsvariable helt undlades. Faktisk behøver vi ikke engang at generere sandsynlighedspotentialer til de konverterede beslutningsvariable.

Optimering af RcSumMar($\mathcal{C}$) Som RcSumMar($\mathcal{C}$) (algoritme 3 på side 83) er beskrevet indtil nu, finder den betingede sandsynlighed ved et kald til RcCondProb($\mathcal{C}$), der bestemmer den betingede sandsynlighed ud fra to fællessandsynligheder, der hver især bestemmes ved ét kald til RcProb.

RcSumMar($\mathcal{C}$) indeholder en løkke, der gennemløber alle tilladelige konfigurationer over chancevariablene i $\mathcal{C}$ og for hver af disse konfigurationer laves ét kald til RcCondProb($\mathcal{C}$). En nærmere analyse af algoritmerne vil vise, at sandsynligheden, der i RcCondProb($\mathcal{C}$) tildeles til variablen n (i linie 5), vil være den samme for alle kaldene fra RcSumMar($\mathcal{C}$) til RcCondProb($\mathcal{C}$). Det er således redundante beregninger, der foretages i linie 5.

Det gælder ligeledes, at n (fra linie 5 af RcCondProb) er lig summen af alle de sandsynligheder, der beregnes i linie 1 i RcCondProb($\mathcal{C}$) under gennemløbet af alle konfigurationerne, $\mathbf{c}$, over $\mathcal{C}$ i RcSumMar($\mathcal{C}$). Dette skyldes, at

$$n = P(past(\mathcal{C})) = \sum_{\mathbf{c} \in \mathcal{C}} P(\mathbf{c}, past(\mathcal{C})). \tag{5.17}$$

Vi kan således omskrive RcSumMar($\mathcal{C}$), så den kalder RcProb direkte i stedet for at kalde RcCondProb. Kaldene til RcProb skal bestemme sandsynlighederne $P(\mathbf{c}, past(\mathcal{C}))$, mens n aldrig udregnes med direkte kald til RcProb. I stedet bestemmes n som summen af de øvrige kald fra RcSumMar($\mathcal{C}$) til RcProb. Således kan vi halvere antallet af kald til RcProb, hvilket vi forventer kan mærkes på den samlede kørselstid for løsningen af en S-DAG.

Omskrivningen af RcSumMar($\mathcal{C}$) skulle være intuitiv jævnfør ovenstående forklaring, og den vil derfor ikke blive illustreret med en pseudokode.

Kapitel 6

Cachingstrategier

Vi har nu set på, hvordan vi kan lave S-DAG og R-DAG konditionering, og vi har muliggjort at mellemresultater kan gemmes i en cache, så et opslag i cachen efterfølgende kan erstatte konditionering gennem dele af S-DAG og R-DAG strukturene.

Vi har indtil nu fokuseret på, hvordan en S-DAG skal løses ved konditionering, og vi har i kraft af frigivning af nyttepotentialer, flytning af uobserverbare chancevariable og fremskydning af observerbare chancevariable, forsøgt at minimere S-DAG konditioneringsalgoritmens pladskrav ved caching.

Det står nu klart, at vi med S-DAG konditionering kan løse en S-DAG for et UID, Γ, på lineært plads i antallet af beslutninger og chancevariable i Γ. Det står ligeledes klart, at vi kan optimere S-DAG konditioneringsalgoritmens kørselstid ved at udnytte muligheden for at gemme mellemregninger.

Vi nævnte i afsnit 2.4 på side 37 at den kendte løsningsalgoritme til løsning af S-DAGe – VE algoritmen – skulle arbejde med potentialer, der vokser eksponentielt med antallet af variable i potentialets domæne. På samme måde kræver fuld caching i S-DAG konditionering også eksponentielt meget plads i antallet af variable i konteksten for knuderne i S-DAGen. Den afgørende forskel på VE og S-DAG konditionering, SC, er dog, at hvor VE algoritmen havde et eksponentielt *pladskrav*, har SC algoritmen kun et lineært pladskrav. Den plads cachen bruger er ikke et krav, der skal være opfyldt for at algoritmen kører, men i stedet en mulighed, der kan udnyttes for at optimere kørselstiden.

På flere systemer, der ikke kan opfylde VE algoritmens pladskrav, vil det med SC algoritmen således være muligt, at evaluere en S-DAG og bestemme de optimale politikker for flere af beslutningerne i S-DAGen.

Et endnu ubehandlet emne er imidlertid det centrale spørgsmål: Hvis der, når UIDet og S-DAGen ligger repræsenteret i hukommelsen på et system, stadig er hukommelse til rådighed, hvordan udnytter vi denne hukommelse til at reducere SC algoritmens kørselstid mest muligt?

Svaret på dette spørgsmål ligger i at vælge en cachingstrategi, der afgør hvilke konfigurationer over konteksten for knuderne i S-DAGen og R-DAGen vi skal gemme i cachen og hvilke, der ikke skal gemmes men i stedet for beregnes igen. I vores pseudokoder har vi repræsenteret cachingstrategierne ved funktionen $cache?(\mathcal{K}, \mathbf{y})$.

Vi forventer, at den strategi, der reducerer kørselstiden mest muligt, vil være hård at bestemme, da kørselstiden for den samlede algoritme vil afhænge af overheadet ved at bestemme cachingstrategien og det samlede antal kald, der udføres under strategien. Denne afvejning vil vi ikke se på, men i stedet vil vi i dette kapitel opstille to strategier, der forventes hurtigt at kunne fastlægge hvilke knuder der skal tildeles cache og stadig reducere det samlede antal kald. For os minimerer en god cachingstrategi således det samlede antal kald, der laves til S-DAG- og R-DAG-knuder.

Alle cachingstrategier kan opdeles i to grupper: statiske cachingstrategier og dynamiske cachingstrategier. De statiske strategier foreskriver en fast tildeling af hukommelsen til de enkelte konfigurationer – en tildeling, der ligger fast under hele SC algoritmens afviklingsforløb. De dynamiske strategier tillader derimod, at tildelingen af cache til de enkelte konfigurationer kan variere under afviklingen af SC algoritmen.

En statisk cachingstrategi tillader, at man én gang for alle tildeler cache til de konfigurationer, der skal gemmes. Modsætningsvist kan en dynamisk strategi løbende re-allokere hukommelsen under SC algoritmens afvikling. De statiske algoritmer vil formentligt være mere simple at forstå og implementere, mens de dynamiske algoritmer formentligt kan reducere antallet af kald bedre, end det er muligt med en statisk strategi, givet en begrænset mængde plads. Dog kan vi forvente et overhead ved en dynamisk strategi, da den skal aktiveres under SC algoritmens kørsel, mens en statisk strategi kan fastlægges inden SC algoritmen startes.

I denne rapport vil vi behandle statiske cachingstrategier. Det vil være et interessant emne for fremtidig forskning at fremkomme med en effektiv dynamisk cachingstrategi samt en effektiv algoritme til bestemmelse af den optimale cachingstrategi – fx ved en afsøgning af rummet af alle cachingstrategier. (En sådan søgealgoritme over statiske cachingstrategier opstilles i [Allen and Darwiche, 2003b] for RC.)

I et computersystem er der normalt flere niveauer af hukommelse af varie-

rende størrelse og med forskellige tilgangstider (fx L1 og L2 cache på CPU'en, RAM og virtuel hukommelse på harddisken). I denne rapport vil vi arbejde med cachingstrategier for ét hukommelsesniveau, og vi antager, at alle indgange i dette niveau kan tilgås lige hurtigt. Det vil være et emne for fremtidig forskning at bestemme, hvordan disse niveauer effektivt kan udnyttes.

I næste afsnit vil vi indføre begrebet *cachetildeling*, der – kort fortalt – specificerer hvor meget cache, vi skal allokere til de forskellige knuder i S-DAGen og R-DAGen. Med dette begreb på plads opstiller vi efterfølgende et udtryk for det samlede antal kald, der skal udføres for at løse en S-DAG med en given cachetildeling for de forskellige knuder i S-DAGen og R-DAGen. Dette udtryk kan bruges til at sammenligne forskellige cachingstrategier. Herefter opstiller vi to forskellige cachingstrategier, der giver forskellige bud på hvilken cachetildeling de enkelte knuder i S-DAGen og R-DAGen skal have.

Først starter vi dog med at repetere betydningen af den anvendte notation i dette kapitel. $SContext(\mathcal{K})$ betegner variablene som cachen for en S-DAG-knude $\mathcal{K}$ er defineret over. $RContext(\mathcal{K})_{\mathcal{B}}$ betegner de tilsvarende variable for en R-DAG-knude. For en R-DAG-knude er konteksten afhængig af termineringsknuden $\mathcal{B}$. Cachen for en knude, $\mathcal{K}$, betegnes $cache_{\mathcal{K}}$. Antallet af forskellige konfigurationer over variablene i $SContext(\mathcal{K})$ betegner vi $SContext(\mathcal{K})^{\#}$ (og tilsvarende for konteksten for en knude i R-DAGen). Størrelsen af $cache_{\mathcal{K}}$ betegnes $|cache_{\mathcal{K}}|$. Størrelsen behøver aldrig at være større end $SContext(\mathcal{K})^{\#}$.

6.1 Cachetildeling

Vi har tidligere beskrevet, hvordan vi kan optimere SC algoritmens kørselstid ved at indføre en cache defineret over konteksten for knuderne i S-DAGen og R-DAGen. Vi har altså én cache på hver knude i S-DAGen og R-DAGen.

Vi indfører begrebet cachetildeling, der skal bruges til at angive hvor meget plads, der skal afsættes til hver S-DAG- og R-DAG knude. Begrebet cachetildeling har vi videreført fra [Darwiche, 2000a, Darwiche, 2001].

Vi vil i beskrivelsen af cachetildelingen kun omtale forholdene ved caching i en S-DAG. Forholdene for caching i en R-DAG følger helt parallelt.

Definition 6.1 (**Cachetildeling**). En *cachetildeling* for en S-DAG, Ξ, er en funktion, cf_{Ξ}, der fra hver cache tilknyttet S-DAGens knuder, $\mathcal{K}$, afbilder over i et tal $0 \leq cf_{\Xi}(\mathcal{K}) \leq 1$. Vi siger, at cachetildelingen for en knude $\mathcal{K}$ er $cf_{\Xi}(\mathcal{K})$. Indekset, der refererer til S-DAGen, kan udelades hvis cachetildelingen således ikke blive tvetydig. □

Intuitionen med cachetildelingen er, at $cf(\mathcal{K})$ angiver den brøkdel af $SContext(\mathcal{K})^{\#}$, der vil blive afsat plads til i systemets hukommelse. Det gælder for den samlede størrelse af cachen, $|cache_{\mathcal{K}}|$, at

$$|cache_{\mathcal{K}}| = cf(\mathcal{K}) \cdot SContext(\mathcal{K})^{\#}. \tag{6.1}$$

Hvis tilstandsrummet for konteksten for en S-DAG-knude, $\mathcal{K}$, således indeholder fx 64 forskellige tilstande, ($SContext(\mathcal{K})^{\#} = 64$), og cachetildelingen for $\mathcal{K}$ er $0,25$, ($cf(\mathcal{K}) = 0,25$), så vil der blive allokeret $0,25 \cdot 64 = 16$ indgange i cachen for $\mathcal{K}$. Når først indholdet af denne cache er bestemt, vil 25% af kaldene til $\mathcal{K}$ medføre opslag i cachen, mens resten af kaldene til $\mathcal{K}$ vil medføre nye kald videre ned gennem S-DAGen.

Vi bruger således begrebet cachetildeling til at abstrahere over de enkelte konfigurationer over konteksten for S-DAG-knuderne, og vi behandler de enkelte konfigurationer for en knude ens.

Ved at specificere cachetildelingen for en S-DAG angiver vi således hvilke knuders kontekst, der skal afsættes plads til. Alternativt kunne man specificere cachetildelingen for de enkelte indgange i konteksten for de enkelte knuder. Men da de enkelte indgange for én knudes kontekst, i forbindelse med S-DAG konditionering, læses lige mange gange og erstatter lige mange kald videre ned gennem S-DAGen, finder vi begrebet "cachetildeling", som defineret i definition 6.1, som et passende abstraktionsniveau, da den sidestiller de enkelte konfigurationer over én knudes kontekst.

Hvis cachetildelingen for en knude, $\mathcal{K}$, ikke er 1, så er der ikke afsat plads til at gemme alle konfigurationerne over $SContext(\mathcal{K})$, men kun $cf(\mathcal{K}) \cdot SContext(\mathcal{K})^{\#}$ konfigurationer. (Vi ser bort fra de specielle situationer, der følger af afrunding). Hvordan afgør man, hvilke konfigurationer, der skal gemmes, og hvilke der ikke skal? Ja, som vi netop har diskuteret, erstatter en gemt indgang i cachen for $\mathcal{K}$ lige mange kald til efterkommerene til $\mathcal{K}$ uafhængigt af, hvilken konfiguration over $SContext(\mathcal{K})$, som indgangen repræsenterer. Således er det underordnet for kørselstiden, hvilke indgange man vælger at gemme, og hvilke man undlader, hvis der ikke er tildelt plads til at gemme alle indgange. Man kan altså frit vælge hvilke konfigurationer, man vil gemme.

6.2 Cachingstrategi for S-DAGen

Med begrebet cachetildeling på plads, vil vi nu opstille forskellige cachingstrategier. Med en cachingstrategi forstår vi et regelsæt, der afgør hvilke cache-

tildelinger, der skal være på de forskellige knuder i S-DAGen og R-DAGen. Da en god cachingstrategi minimerer det samlede antal kald, der laves til S-DAG- og R-DAG-knuder, starter vi derfor med at opstille et udtryk for hvor mange kald, der laves til S-DAG- og R-DAG-knuderne med en given cachetildeling. Med en antagelse om, at SC algoritmens samlede kørselstid er proportionalt med det samlede antal kald til S-DAG og R-DAG-knuder, kan vi sammenligne kørselstiden for SC algoritmen med to forskellige cachingstrategier ved at sammenligne antallet af kald til S-DAG- og R-DAG-knuder, der udføres under disse strategier.

Vi opstiller nu udtrykket, der bestemmer antallet af kald i forbindelse med at udføre S-DAG konditionering på en S-DAG, Ξ, med en given cachetildeling, cf_Ξ. Da vi kun behandler statiske cacheingstrategier, ændrer cf_Ξ sig ikke undervejs. Når dette gælder, kan antallet af kald, der laves under løsning af Ξ til en vilkårlig S-DAG-knude, $\mathcal{K}$, bestemmes ved udtryk (6.2). Intuitionen bag udtryk (6.2) er som følger: Der laves kun ét kald til $Source$, da $Source$ er den første knude i S-DAGen. Det er fra $Source$, at de efterfølgende kald laves. Antallet af kald til en knude $\mathcal{K}$ (forskellig fra $Source$) er summen af antallet af kald fra de forskellige forældre til $\mathcal{K}$ i S-DAGen. Antallet af kald fra én forælder, $\mathcal{P}$, afhænger af cachetildelingen for $\mathcal{P}$, $cf_\Xi(\mathcal{P})$. Hvis der ikke er tildelt noget cache til $\mathcal{P}$ (dvs. $cf_\Xi(\mathcal{P}) = 0$), så vil der for hvert kald til $\mathcal{P}$ blive lavet $\mathcal{P}^\#$ kald til $\mathcal{K}$ som et resultat af, at vi skal konditionere over alle variablene i $\mathcal{P}$. Hvis der er tildelt fuld cache til $\mathcal{P}$, så laves der kun kald fra $\mathcal{P}$ til $\mathcal{K}$ for at beregne indholdet af cachen for $\mathcal{P}$. Dette medfører $\mathcal{P}^\#$ kald fra $\mathcal{P}$ for hver indgang i $SContext(\mathcal{P})$. Det gennemsnitlige antal kald, udført ved en ikke diskret cachetildeling, vil være en lineær afvejning mellem de to ekstremer, som vi netop har beskrevet. Denne afvejning vil være bestemt af cachetildelingen for $\mathcal{P}$. Antallet af kald til $\mathcal{K}$ er således

$$SCallTo_{cf_\Xi}(\mathcal{K}) = \begin{cases} 1 & \text{hvis } \mathcal{K} = Source \\ \sum_{\mathcal{P}\in\pi(\mathcal{K})} \big([1 - cf_\Xi(\mathcal{P})]\ SCallTo_{cf_\Xi}(\mathcal{P})\ \mathcal{P}^\# & \\ \qquad + cf_\Xi(\mathcal{P})\ SContext(\mathcal{P})^\#\ \mathcal{P}^\#\big) & \text{ellers,} \end{cases} \tag{6.2}$$

hvor $\pi(\mathcal{K})$ er forældrene til $\mathcal{K}$ i S-DAGen.

Det samlede antal kald til S-DAG-knuderne, V, i en S-DAG, Ξ, ved en given cachetildeling, cf_Ξ, er nu

$$SCallTotal_{cf_\Xi}() = \sum_{\mathcal{K}\in V} SCallTo_{cf_\Xi}(\mathcal{K}). \tag{6.3}$$

$SCallTotal_{cf_\Xi}()$ giver dog ikke et eksakt billede af kørselstiden for at udføre S-DAG konditionering på Ξ. For at bestemme det samlede antal kald, der udføres under S-DAG konditionering, skal vi også medregne antallet af kald, der udføres i R-DAGen.

Antallet af kald til en knude, $\mathcal{K}$, i en R-DAG med cachetildelingen cf og termineringsknuden $TNode$ er givet ved

$$RCallTo_{cf}(\mathcal{K})_{TNode} = \begin{cases} 1 & \text{hvis } \mathcal{K} = Source \\ [1 - cf(\mathcal{P})]\ RCallTo_{cf}(\mathcal{P})_{TNode}\ \mathcal{U}^{\#} & \\ +\ cf(\mathcal{P})\ RContext(\mathcal{P})^{\#}_{TNode}\ \mathcal{U}^{\#} & \text{ellers,} \end{cases}$$

hvor $\mathcal{P}$ er en forælder til $\mathcal{K}$ og $\mathcal{U}$ er de uinstantierede uobserverbare chancevariable, der er i $\mathcal{P}$ ($\mathcal{U} = Cond(TNode) \cap \mathcal{P}$). Hvis alle variable i $\mathcal{P}$ er instantierede (dvs. $\mathcal{U} = \emptyset$), så er $\mathcal{U}^{\#} = 1$.

Denne funktion modsvarer (6.2) for S-DAG-knuder. Forskellene skyldes, at der kun laves kald til en knude $\mathcal{K}$ fra én af dens forældre, $\mathcal{P}$, i R-DAGen, og der i $\mathcal{P}$ kun konditioneres over de variable, $\mathcal{U}$, der er uinstantierede i $\mathcal{P}$.

Antallet af kald, der udføres i en R-DAG, $\mathcal{R}$, afhænger af hvilken sandsynlighed der skal bestemmes, dvs. hvilken termineringsknude, $TNode$, der er angivet inden kaldet til $\mathcal{R}$. Jo flere variable, der er i $Cond(TNode)$, jo flere kald, bliver der lavet i $\mathcal{R}$. Det samlede antal kald til bestemmelse af én fællessandsynlighed er bestemt ved summen af antallet af kald til de enkelte knuder på stien, $s(TNode)$, ned til termineringsknuden

$$RCallTotal_{cf_{\mathcal{R}}}(TNode) = \sum_{\mathcal{K} \in s(TNode)} RCallTo_{cf_{\mathcal{R}}}(\mathcal{K})_{TNode}. \tag{6.4}$$

Vi kan nu opstille et udtryk for det samlede antal R-DAG-kald, der udføres under løsningen af en S-DAG. Dette antal er givet ved

$$\begin{aligned} StoRCallTotal_{cf_\Xi, cf_{\mathcal{R}}}() \;=\; \sum_{\mathcal{C} \in V_C} \Big(& [1 - cf_\Xi(\mathcal{C})]\ SCallTo_{cf_\Xi}(\mathcal{C})\ \mathcal{C}^{\#} RCallTotal_{cf_{\mathcal{R}}}(\mathcal{C}) \\ & + cf_\Xi(\mathcal{C})\ SContext(\mathcal{C})\ \mathcal{C}^{\#} RCallTotal_{cf_{\mathcal{R}}}(\mathcal{C}) \Big), \end{aligned}$$

hvor V_C er alle knuderne med chancevariable i S-DAGen. For hvert kald til en chanceknude, $\mathcal{C}$, i S-DAGen, beregnes én sandsynlighed for hver konfiguration over chancevariablene i $\mathcal{C}$ vha. R-DAGen, (som omtalt sidst i afsnit 5.4.2 på side 94). Antallet af kald til R-DAG-knuder under udregningen af én af disse sandsynligheder er bestemt ved udtryk (6.4).

Det samlede antal S-DAG- og R-DAG-kald er nu givet ved summen af kald til S-DAG-knuder og kald til R-DAG-knuder.

$$CallTotal_{cf_\Xi,cf_\mathcal{R}}() = SCallTotal_{cf_\Xi}() + StoRCallTotal_{cf_\Xi,cf_\mathcal{R}}(). \quad (6.5)$$

I afsnit 8.3.2 på side 123 sammenholder vi antallet af rekursive kald estimeret af udtryk (6.5) med det faktiske antal kald, som SC algoritmen laver.

Hvis vi antager, at antallet af kald til S-DAG- og R-DAG-knuder er proportionalt med den samlede kørselstid, så kan vi bruge udtryk (6.5) til at bestemme den samlede kørselstid, for at løse en given S-DAG med en given cachetildeling. I afsnit 8.3.3 på side 123 undersøger vi, om denne antagelse holder i praksis.

Målet for en cachingstrategien for S-DAGen og R-DAGen må således være at minimere udtryk (6.5). Vi skal i de næste to afsnit se på to forskellige strategier, der søger at minimere udtryk (6.5).

6.2.1 En naiv algoritme

Cachingstrategien beskrevet i dette afsnit bygger på en antagelse om, at alle instantieringer over konteksten for *alle* S-DAG-knuder er lige nyttige at gemme og at det er fordelagtigt, at have den tilgængelige hukommelse for cache fordelt jævnt ud over alle S-DAG-knuderne. Cachingstrategien der præsenteres i dette afsnit vil vi betegne "den naive cachingstrategi". Denne naive cachingstrategi er ikke nødvendigvis effektiv (mht. at reducere det samlede antal S-DAG- og R-DAG-kald), men den er forholdsvis simpel at forstå og implementere, og den giver et sammenligningsgrundlag for den næste cachingstrategi, vi vil opstille.

Denne cachingstrategi dikterer, at cachetildelingen er den samme for alle knuderne i S-DAGen. På denne måde bliver den hukommelse, der er til rådighed til caching, jævnt fordelt ud over alle knuderne i S-DAGen.

Hvis der således er x bytes hukommelse til rådighed, når UIDet og S-DAGen er repræsenteret i systemet, så vælger vi den højeste cachetildeling, der medfører, at det samlede cacheforbrug ikke overstiger x bytes. Vi skal nu se på, hvordan vi kan finde denne cachetildelingsværdi.

Hvis der er tilstrækkeligt meget hukommelse, kan alle konfigurationer over konteksterne for S-DAG-knuderne gemmes. Dette vil medføre et samlet hukommelseskrav, i bytes, der kan bestemmes som summen af antallet af konfigurationer over konteksten for alle S-DAG-knuderne, V_Ξ, i S-DAGen, Ξ, ganget med antallet af bytes, der kræves for at repræsentere værdien for én konfiguration i én kontekst, *Do*. Således får vi et maksimalt cacheforbrug

givet ved

$$MaxCache(\Xi) = Do \cdot \sum_{\mathcal{K} \in V_\Xi} SContext(\mathcal{K})^{\#}. \tag{6.6}$$

Cachetildelingen for alle knuderne i Ξ kan nu bestemmes som den tilgængelige plads, *available*, i forhold til den maksimalt krævede plads

$$cf(\mathcal{K}) = Min\left(1, \frac{available}{MaxCache(\Xi)}\right), \tag{6.7}$$

Bemærk at cachetildelingen for knuderne aldrig kan være større end 1.

Kørselstid Det er klart, at cachetildelingen for alle knuder med denne algoritme kan bestemmes lineært i antallet af knuder i S-DAGen.

6.2.2 En grådig algoritme

I dette afsnit skal vi se på en anden cachingstrategi, der modsat den naive cachingstrategi ikke antager, at alle knuders kontekst er lige fordelagtig at gemme, og at det derfor heller ikke er fordelagtigt at have den tilgængelige hukommelse for cache fordelt jævnt ud over alle S-DAG-knuderne.

Ideen bag denne strategi er at tildele pladsen i hukommelsen til cache til de S-DAG-knuder, hvor den umiddelbart forventes bedst at kunne reducere det samlede antal rekursive kald, der laves til S-DAG- *og R-DAG*-knuder. Vi vil identificere disse knuder ved at se på, hvor mange kald, der samlet set kan spares ved at tildele cache til de enkelte knuder og sætte dette mål i forhold til hvor meget plads, knudernes cache optager. Således tildeler vi iterativt og grådigt pladsen i hukommelsen efter et "størst reduktion i antallet af rekursive kald per plads i hukommelsen"-princip.

Uformelt beskrevet er tankegangen bag dette princip som følger: Det må forventes at være godt at tildele cache til en knude, hvis værdier ellers skulle beregnes mange gange, ligeledes må det forventes at være godt at tildele cache til en knude, hvis værdier kræver mange rekursive kald for at blive beregnet. Hvis begge disse egenskaber er til stede på en knude, der har et lille tilstandsrum for konteksten, så vil det samtidigt ikke optage meget hukommelse at tildele cache til denne knude.

Grundideen i denne cachingstrategi er som følger: Start med ikke at have tildelt cache til nogle knuder. Find så dén knude, der reducerer det samlede antal kald mest muligt og tildel fuld cache til denne knude. Gentag denne

procedure indtil alt den tilgængelige hukommelse er tildelt til knuder, eller indtil der er tildelt fuld cache til alle knuder.

Denne svæversynede strategi er ikke garanteret at finde den optimale cachetildeling, men vi forventer at få en mere effektiv tildeling end det var tilfældet for den naive strategi.

I princippet skal vi bestemme faldet i det samlede antal S-DAG- og R-DAG-kald ved at tildele cache til $\mathcal{K}$ for at reducere antallet af kald givet ved udtryk (6.5), men da tildelingen af cache til en enkelt knude, $\mathcal{K}$, kun påvirker antallet af kald der laves i forbindelse med at besvare kald til $\mathcal{K}$ selv, er det tilstrækkeligt, at bestemme ændringen i antallet af kald der laves for at besvare kald til $\mathcal{K}$.

Lad os opstille et udtryk, der identificerer den knude, der er mest fordelagtig at tildele cache til, når der ikke er tildelt cache til nogle andre knuder. Denne knude identificeres ved, for hver knude, $\mathcal{K}$, at finde antallet af kald, $Call_{cf_\Xi,cf_\mathcal{R}}(\mathcal{K})$, der laves for at besvare kald til $\mathcal{K}$. Fra dette subtraheres antallet af kald, $CallCache_{cf_\Xi,cf_\mathcal{R}}(\mathcal{K})$, der laves for at besvare kald til $\mathcal{K}$, hvis der er tildelt fuld cache til $\mathcal{K}$ (dvs. antallet af kald, der laves, for at bestemme indholdet af cachen for $\mathcal{K}$). Kaldene til at bestemme indholdet af cachen for $\mathcal{K}$ spares ikke ved at tildele cache til $\mathcal{K}$. Dette angiver faldet i det samlede antal kald, der opnås ved at tildele cache til $\mathcal{K}$.

Den knude, der medfører det største fald i det samlede antal kald *per* konfiguration over knudens kontekst, er således den knude, som med vores snæversynede strategi er mest fordelagtig at tildele cache til, og den får derfor tildelt fuld cache.

Vi vælger altså den knude, $\mathcal{K}$, med højest vægt, hvor vægten er ændringen i antallet af rekursive kald per indgang i konteksten for $\mathcal{K}$:

$$Weight_{cf_\Xi,cf_\mathcal{R}}(\mathcal{K}) = \frac{Call_{cf_\Xi,cf_\mathcal{R}}(\mathcal{K}) - CallCache_{cf_\Xi,cf_\mathcal{R}}(\mathcal{K})}{SContext(\mathcal{K})^\#}. \tag{6.8}$$

Det samlede antal af kald, der laves, for at besvare kald til en knude, $\mathcal{K}$, hvis der ikke er tildelt cache til $\mathcal{K}$, $Call_{cf_\Xi,cf_\mathcal{R}}(\mathcal{K})$, er givet ved antallet af kald *til* $\mathcal{K}$, $SCallTo_{cf_\Xi}(\mathcal{K})$, gange antallet af kald til efterfølgere af $\mathcal{K}$, $CallIn_{cf_\Xi,cf_\mathcal{R}}(\mathcal{K})$, for at besvare ét kald til $\mathcal{K}$. Dette er udtrykt ved

$$Call_{cf_\Xi,cf_\mathcal{R}}(\mathcal{K}) = SCallTo_{cf_\Xi}(\mathcal{K}) \cdot CallIn_{cf_\Xi,cf_\mathcal{R}}(\mathcal{K}). \tag{6.9}$$

$SCallTo_{cf_\Xi}(\mathcal{K})$ er givet ved

$$SCallTo_{cf_\Xi}(\mathcal{K}) = \begin{cases} 1 & \text{hvis } \mathcal{K} = Source \\ \sum_{\mathcal{P}\in\pi(\mathcal{K})} SCallTo_{cf_\Xi}(\mathcal{P})\ \mathcal{P}^\# & \text{ellers.} \end{cases} \tag{6.10}$$

Bemærk at dette udtryk er special-tilfældet af udtryk (6.2), når ingen knuder har fået tildelt cache.

$CallIn_{cf_\Xi, cf_\mathcal{R}}(\mathcal{K})$ er givet ved

$$CallIn_{cf_\Xi, cf_\mathcal{R}}(\mathcal{K}) = \begin{cases} \mathcal{K}^{\#} \cdot RCallTotal_{cf_\mathcal{R}}(\mathcal{K}) & \text{hvis } \mathcal{K} = Sink \\ \mathcal{K}^{\#} \cdot \sum_{\mathcal{C} \in ch(\mathcal{K})} CallIn_{cf_\Xi, cf_\mathcal{R}}(\mathcal{C}) & \\ & \text{hvis } \mathcal{K} \text{ er en beslutningsknude,} \\ \mathcal{K}^{\#} \cdot \Big(RCallTotal_{cf_\mathcal{R}}(\mathcal{K}) + \sum_{\mathcal{C} \in ch(\mathcal{K})} CallIn_{cf_\Xi, cf_\mathcal{R}}(\mathcal{C}) \Big) & \\ & \text{hvis } \mathcal{K} \text{ er en chanceknude,} \end{cases} \tag{6.11}$$

hvor $ch(\mathcal{K})$ er børnene til $\mathcal{K}$ i S-DAGen. Intuitionen bag udtryk (6.11) er som følger: Hvis $\mathcal{K}$ er $Sink$ så startes ikke yderligere S-DAG-kald, men for hver konfiguration over chancevariablene i $Sink$ laves der ét kald til R-DAGen. Hvert af disse kald medfører $RCallTotal_{cf_\mathcal{R}}(\mathcal{K})$ kald til R-DAG-knuder. Hvis $\mathcal{K}$ er en beslutningsknude, så startes der ét kald til hvert af $\mathcal{K}$s børn per konfiguration over $\mathcal{K}$. Endelig hvis $\mathcal{K}$ indeholder chancevariable, så startes der ligeledes ét kald til hvert af $\mathcal{K}$s børn per konfiguration over $\mathcal{K}$. Derudover startes der også $RCallTotal_{cf_\mathcal{R}}(\mathcal{K})$ kald i R-DAGen per konfiguration over $\mathcal{K}$ for at beregne sandsynligheder.

Hvis der er tildelt fuld cache til $\mathcal{K}$, er antallet af kald, der samlet set skal udføres for at besvare kald til $\mathcal{K}$, lig med antallet af kald, der skal udføres for at bestemme indholdet af $SContext\,(\mathcal{K})$. Dette antal er givet ved antallet af konfigurationer over $SContext\,(\mathcal{K})$ ganget med antallet af kald til efterfølgere af $\mathcal{K}$ for at beregne én indgang i $SContext\,(\mathcal{K})$. Dette antal er givet ved

$$CallCache_{cf_\Xi, cf_\mathcal{R}}(\mathcal{K}) = SContext(\mathcal{K})^{\#} \cdot CallIn_{cf_\Xi, cf_\mathcal{R}}(\mathcal{K}). \tag{6.12}$$

Vi kan nu identificere den knude, der vil reducere det samlede antal kald mest muligt ved at få tildelt cache. Det er knuden med den højeste vægt jf. udtryk (6.8).

Vi sætter $cf(\mathcal{K}) = 1$ for den knude $\mathcal{K}$, der har højest vægt. Cachetildelingen for alle andre knuder forbliver 0. Hvis der ikke er hukommelse nok til at tildele fuld cache til $\mathcal{K}$, så tildeles der så meget hukommelse til $\mathcal{K}$, som det er muligt indenfor den tilladelige hukommelsesmængde.

Hvis der er mere hukommelse til rådighed, når der er tildelt cache til den knude med højest vægt, så tildeler vi cache til den næste knude med højest vægt. Bemærk dog, at vægten for en knude afhænger af, hvilken cachetildeling

der er på de andre knuder, hvorfor vi er nødt til, at bestemme vægten igen for de øvrige knuder, hvor vi denne gang tager højde for den cache, der allerede er tildelt i S-DAGen.

Vi har allerede beskrevet hvordan $SCallTo_{cf_\Xi}(\mathcal{K})$ afhænger af cachetildelingen for de øvrige knuder. Dette er udtrykt i (6.2). På samme måde gælder det, at $CallIn_{cf_\Xi,cf_\mathcal{R}}(\mathcal{K})$ afhænger af cachetildelingen for de øvrige knuder som her angivet

$$CallIn_{cf_\Xi,cf_\mathcal{R}}(\mathcal{K}) = cf_\Xi(\mathcal{K}) + [1 - cf_\Xi(\mathcal{K})] \cdot \mathcal{K}^{\#} \cdot \begin{cases} RCallTotal_{cf_\mathcal{R}}(\mathcal{K}) & \text{hvis } \mathcal{K} = Sink \\ \sum_{\mathcal{C} \in ch(\mathcal{K})} CallIn_{cf_\Xi,cf_\mathcal{R}}(\mathcal{C}) & \text{hvis } \mathcal{K} \text{ er en beslutningsknude.} \\ \left(RCallTotal_{cf_\mathcal{R}}(\mathcal{K}) + \sum_{\mathcal{C} \in ch(\mathcal{K})} CallIn_{cf_\Xi,cf_\mathcal{R}}(\mathcal{C})\right) & \text{hvis } \mathcal{K} \text{ er en chanceknude.} \end{cases} \tag{6.13}$$

$CallIn_{cf_\Xi,cf_\mathcal{R}}(\mathcal{K})$ beskriver her det gennemsnitlige antal kald, der skal laves for at besvare ét kald til $\mathcal{K}$, når indholdet af cachen for $\mathcal{K}$ er bestemt. En brøkdel af kaldene til $\mathcal{K}$, givet ved $cf_\Xi(\mathcal{K})$, besvares med et opslag i cachen for $\mathcal{K}$, mens de øvrige $1-cf_\Xi(\mathcal{K})$ kald til $\mathcal{K}$ besvares som i udtryk (6.11).

Den cachingstrategi, som vi har præsenteret i dette afsnit, har vi selv udviklet. Vi er dog efterfølgende blevet opmærksomme på, at en cachingstrategi, der også arbejder med snævertsynet at reducere det samlede antal kald mest muligt, præsenteres i [Allen et al., 2004] for RC. Her er det dog kun antallet af kald i én graf (d-træ'et) der reduceres og ikke antallet af kald i to grafer (S-DAGen og R-DAGen) som her.

Kørselstid Lad os bestemme kørselstiden for at bestemme cachetildelingen for S-DAG knuderne med denne cachingstrategi. I den følgende udledning betegner $s\#$ antallet af S-DAG-knuder, og $r\#$ antallet af R-DAG-knuder.

Med en cachetildeling for S-DAGen kan $SCallTo_{cf_\Xi}(\mathcal{K})$ bestemmes for alle S-DAG-knuderne lineært i antallet af S-DAG-knuder. Denne kompleksitet kan opnås, ved først af bestemme $SCallTo_{cf_\Xi}(\mathcal{K})$ for $\mathcal{K} = Source$ og derefter for de øvrige S-DAG-knuder ved en bredde-først gennemgang ned gennem hele S-DAGen. På denne måde er $SCallTo_{cf_\Xi}(\mathcal{P})$ for forælderen af en knude, $\mathcal{K}$, fastlagt, når $SCallTo_{cf_\Xi}(\mathcal{K})$ for knuden selv skal fastlægges.

For én cachetildeling for S-DAGen kan $CallIn_{cf_\Xi,cf_\mathcal{R}}(\mathcal{K})$ også bestemmes lineært i antallet af S-DAG-knuder. Dog skal bredde-først gennemgangen

af hele S-DAGen nu tage udgangspunkt i $Sink$. Således er $CallIn_{cf_{\Xi},cf_{\mathcal{R}}}(\mathcal{C})$ fastlagt for alle børnene til $\mathcal{K}$, når $CallIn_{cf_{\Xi},cf_{\mathcal{R}}}(\mathcal{K})$ for $\mathcal{K}$ skal fastlægges.

For hver chanceknude, $\mathcal{K}$, i S-DAGen skal vi også bruge værdien $RCall\text{-}Total_{cf_{\mathcal{R}}}(\mathcal{K})$. Denne værdi kan udregnes én gang for hver S-DAG-knude og derefter genbruges. Denne værdi ændrer sig ikke efterhånden som vi tildeler cache til S-DAG-knuderne.

At bestemme $RCallTotal$ for én S-DAG-knude, kræver at alle knuderne på én sti mellem denne knude og $Source$ i R-DAGen besøges én gang. Det kan gøres lineært i højden af R-DAGen (højden er det maksimale antal knuder på én maksimalt orienteret sti mellem $Source$ og $Sink$ i R-DAGen). Lad n betegne højden. $RCallTotal$ for alle S-DAG-knuder kan bestemmes lineært i $s\# \cdot n$.

Når først $RCallTotal$ er bestemt for alle S-DAG-knuderne, kan $SCallTo$ og $CallIn$ ligeledes bestemmes lineært i sammenlagt $s\# + s\#$ for én cachetildeling for S-DAGen. For én cachetildeling kan vi altså finde vægten for alle S-DAG-knuderne lineært i $s\# + s\#$.

For at bestemme den samlede cachetildeling for alle S-DAG-knuderne med den grådige strategi skal vi igennem (maksimalt) $s\#$ runder, hvor vi bestemmer vægten for alle S-DAG-knuderne (der endnu ikke har fået tildelt cache). Hver runde kan laves lineært i $s\# + s\#$. Det giver samlet $s\# \cdot (s\# + s\#)$. Hertil skal vi lægge $s\# \cdot n$ for at bestemme værdierne for $RCallTotal$. Alt i alt får vi altså

$$s\#(s\# + s\#) + s\# \cdot n = s\#(s\# + s\# + n) = O(s\#^2). \tag{6.14}$$

I udtryk (6.14) har vi brugt, at $n \leq s\#$ (hvilket altid gælder).

Således kan cachetildelingen for denne strategi bestemmes kvadratisk i antallet af knuder i S-DAGen. Tidskompleksiteten for denne cachingstrategi kan formentligt forbedres yderligere ved at genbruge beregningerne af $SCallTo$ og $CallIn$ for de knuder, hvor de ikke ændrer sig ved tildelingen af cache til en anden knude.

Denne grådige cachingstrategi specificerer i hvilken rækkefølge, man skal tildele cache til S-DAG-knuderne. Når først denne rækkefølge er fastlagt, kan man tildele cache til S-DAGen i konstant tid. Det er blot at tildele cache til S-DAG-knuderne i den rækkefølge, der er fundet ved cachingstrategien.

Algoritmen til at bestemme cachetildelingen kan således køres inden vi evaluerer S-DAGen, hvorfor strategiens kørselstid er mindre vigtig for hvor hurtigt, man kan løse en opstillet S-DAG.

6.2.3 0-vægte

Når man tildeler cache iterativt til knuderne med højest vægt, kan man nå en tildeling, cf_Ξ, hvor $Weight_{cf_\Xi,cf_\mathcal{R}}(\mathcal{K}) = 0$ for alle knuder, $\mathcal{K}$, med $cf_\Xi(\mathcal{K}) = 0$. I denne situation gælder det, at man ikke yderligere kan reducere det samlede antal kald, der udføres for at løse S-DAGen, ved at tildele ekstra cache til nogle S-DAG-knuder.

Et eksempel på en situation, hvor vægten for en knude er 0, er, hvis en knude, $\mathcal{K}$, har én forælder, $\mathcal{P}$, med $cf_\Xi(\mathcal{P}) = 1$ og det gælder at $SContext(\mathcal{K}) = SContext(\mathcal{P}) \cup \mathcal{P}$. At denne situation medfører en vægt på 0 kan ses ud fra følgende udregning

$$
\begin{aligned}
SContext(\mathcal{K}) &= SContext(\mathcal{P}) \cup \mathcal{P} \Rightarrow \\
SContext(\mathcal{K})^\# &= SContext(\mathcal{P})^\# \mathcal{P}^\# \Rightarrow \\
&\qquad \text{(Udtryk (6.2) og } \mathcal{K} \text{ har én forælder, } \mathcal{P}\text{, med } cf_\Xi(\mathcal{P}) = 1) \\
SContext(\mathcal{K})^\# &= SCallTo_{cf_\Xi}(\mathcal{K}) \Rightarrow \\
0 &= \frac{(SCallTo_{cf_\Xi}(\mathcal{K}) - SContext(\mathcal{K})^\#) CallIn_{cf_\Xi,cf_\mathcal{R}}(\mathcal{K})}{SContext(\mathcal{K})^\#} \Rightarrow \\
&\qquad \text{(Udtryk (6.9) og (6.12))} \\
0 &= \frac{Call_{cf_\Xi,cf_\mathcal{R}}(\mathcal{K}) - CallCache_{cf_\Xi,cf_\mathcal{R}}(\mathcal{K})}{SContext(\mathcal{K})^\#} \Rightarrow \\
&\qquad \text{(Udtryk (6.8))} \\
0 &= Weight_{cf_\Xi,cf_\mathcal{R}}(\mathcal{K}).
\end{aligned}
$$

Der kan opstilles flere situationer, hvor vægten for en knude kan blive 0. Det gælder generelt, at vægten for en knude, $\mathcal{K}$, er 0, hvis antallet af kald til $\mathcal{K}$ er lig med antallet af konfigurationer over konteksten for $\mathcal{K}$. Når vægten for en knude, $\mathcal{K}$, er 0, vil værdierne i cachen for $\mathcal{K}$ aldrig blive læst.

Det gælder, at det aldrig vil medføre ekstra kald at have tildelt cache til en knude, $\mathcal{K}$, fremfor at undlade at tildele cache til $\mathcal{K}$. Samtidigt gælder det, at vægten for en knude, $\mathcal{K}$, udtrykker reduktionen i antallet af kald, ved at tildele cache til $\mathcal{K}$. Således må det gælde, at hvis vægten for alle knuder uden cache er 0 med en given cachetildeling, cf_Ξ, må det gælde, at det samlede antal kald, der skal udføres for at løse S-DAGen med cf_Ξ ikke er større end det antal kald, der skal bruges, hvis der er tildelt fuld cache til alle knuder i S-DAGen. Dette betyder, at vi med vores grådige strategi kan opnå et (globalt) minimum i antallet af rekursive kald, selvom vi ikke har tildelt fuld cache til alle knuder i S-DAGen. Dette vil netop være, når vægten for alle knuder uden tildelt cache er 0.

Dette er et vigtigt resultat, da det betyder, at SC algoritmen med denne strategi kan opnå et lavere pladsforbrug, end det vil være muligt med den

naive strategi, da den naive strategi skal have fuld cache tildelt til alle knuder, for at opnå det globale minimum i antallet af rekursive kald.

6.3 Cachingstrategi for R-DAGen

I afsnit 6.2 så vi 2 forskellige strategier til at bestemme cachetildelingen for en S-DAG. Disse to fremgangemåder kan også begge bruges til at bestemme cachetildelingen for R-DAGen.

Dette er imidlertid overkill, fordi hver gang vi bestemmer en sandsynlighed vha. R-DAGen, laver vi kun konditionering ned gennem én sti (fra *Source* til termineringsknuden). Hvis der lå cache på alle de R-DAG-knuder, der ikke ligger på denne sti, så vil de ikke forbedre kørselstiden, men kun optage plads. Således vil vi kun tildele cache til én sti ad gangen.

En mulighed er, at tildele cache til alle knuderne langs denne sti ud fra én af strategierne beskrevet i afsnit 6.2. Vi forventer dog, at overheadet ved at bestemme cachetildelingerne ud fra den grådige strategi vil være større, end den reduktion i tid, der kan opnås ved at anvende cache på R-DAGen. Husk da på, at hvis vi anvendte den grådige strategi til at tildele cache til R-DAGens knuder, så ville det medføre, at vi skulle fastlægge cachetildelingene én gang per sandsynlighed, der skal findes. Da der skal bestemmes et betydeligt antal sandsynligheder, forventer vi, at en meget primitiv caching strategi vil være en fordel for den samlede kørselstid.

Da vi i en R-DAG kan nøjes med kun at tildele cache til én sti af gangen, og da disse cache samtidigt kun vil være defineret over de uinstantierede uobserverbare chancevariable i UIDet (og altså ikke alle variablene i UIDet, som ved S-DAG-knudernes kontekst), forventer vi, at R-DAGens pladskrav ved fuld cache vil være moderate ift. pladskravene for fuld caching på S-DAGen.

Således vil vi foreslå en cachingstrategi, der blot tillader fuld caching til alle knuderne på den aktuelle sti i R-DAGen. Et alternativ kunne værre, helt at undlade at bruge caching på R-DAGen. Når sandsynlighederne skal bestemmes vha. R-DAG konditionering, er det kun de uinstantierede uobserverbare chancevariable, der skal konditioneres over, og hvis der er "tilpas få" uinstantierede uobserverbare chancevariable på den sti, der konditioneres over, vil overheaded ved at administrere cachen formentligt kunne overstige gevinsten ved at kunne genbruge beregninger.

Således ender vi ud med to forskellige cachingstrategier for R-DAGen: (1) fuld caching for alle knuderne på den enkelte sti i R-DAGen og (2) ingen caching i R-DAGen. Hvis der er fuld cache på R-DAGen, så er der afsat cache nok til at konteksten for R-DAG-knuderne langs én sti kan gemmes.

Kapitel 7

Teoretisk sammenligning med VE-algoritmen

I dette kapitel vil vi analysere og opstille tids- og pladskompleksiteten for SC algoritmen med og uden cache, og vi vil sammenligne disse kompleksiteter med tids- og pladskompleksiteten af VE algoritmen.

7.1 Pladskompleksitet

Uden cache Når der ikke benyttes cache i SC algoritmen, er algoritmens pladsforbrug lineært i antallet af S-DAG-knuder, h, på en maksimalt orienteret sti mellem $Source$ og $Sink$, i S-DAGen. Dette skyldes, at der maksimalt er ét aktivt rekursivt kald per niveau i S-DAGen, og der kun skal gemmes et konstant antal værdier per aktivt kald. Derudover er der også maksimalt h aktive kald i R-DAGen. Da h aldrig bliver større end antallet af beslutnings- og observationsvariable, $nDec$ og $nObs$, bliver pladskompleksiteten $2 \cdot (nDec + nObs) = O(nDec + nObs)$.

Med cache Når der benyttes cache i SC algoritmen, er algoritmens pladsforbrug afhængigt af størrelsen af konteksten for S-DAG-knuderne. Lad $|SContext|^*$ betegne antallet af variable i konteksten for den knude med flest variable i konteksten. Da pladsforbruget vokser eksponentielt med antallet af variable i konteksten for knuderne, og da ingen knuder har en kontekst med flere variable end $|SContext|^*$, bliver en øvre grænse for SC algoritmens pladskompleksitet ved anvendelse af cache $O(nNodes \cdot exp(|SContext|^*))$,

hvor $nNodes$ er antallet af knuder i S-DAGen. Her har vi antaget, at pladsforbruget til cache i R-DAGen er ubetydeligt ift. pladsforbruget til cache i S-DAGen (se afsnit 6.3 på side 108).

Sammenligning VE algoritmens pladskompleksitet er $O(nNodes \cdot exp(w^*))$, hvor w^* er den største bredde af de elimineringsrækkefølger, der er repræsenteret ved stierne i S-DAGen (se afsnit 2.4 på side 37).

Hvis vi antager, at w^* er større end $|SContext|^*$, så har SC algoritmen altså en bedre pladskompleksitet end VE algoritmen. Denne antagelse vil vi diskutere sidst i dette kapitel.

7.2 Tidskompleksitet

Vi vil udtrykke tidskompleksiteten af SC algoritmen ved antallet af rekursive kald, der udføres af SC algoritmen, og vi vil beskrive tidskompleksiteten for situationerne, hvor der er ingen og fuld cache på alle knuderne i S-DAGen og ingen og fuld cache på R-DAGen.

Uden cache Antallet af rekursive kald er netop givet ved udtryk (6.5) på side 101. Opstilles en øvre grænse for dette udtryk, når der ikke er cache, vil denne da være vores tidskompleksitet for SC algoritmen uden cache.

Udtryk (6.5) består af udtrykkene $SCallTotal$ og $StoRCallTotal$, så for at finde en øvre grænse for udtryk (6.5), skal vi finde en øvre grænse for $SCallTotal$ og for $StoRCallTotal$. En øvre grænse for $SCallTotal$ kan fås, ved at antage, at antallet af kald til hver knude er det samme som antallet af kald til den knude, der modtager flest kald. Da $Sink$ er den knude, der modtager flest kald, skal vi altså bestemme en øvre grænse for antallet af kald til $Sink$. Hvis S-DAGen kun indeholder én sti, så er antallet af kald til $Sink$ givet ved $exp(nDec + nObs + nHid)$, hvor $nDec$, $nObs$ og $nHid$ er antallet af beslutninger, observerbare og uobserverbare chancevariable i UIDet. Denne kompleksitet opnås, da alle variable i UIDet optræder på en vilkårlig maksimalt orienteret sti i en S-DAG, og $Sink$ bliver kaldt én gang per konfiguration over disse variable. Det samlede antal kald til $Sink$ i en S-DAG med $nPath$ forskellige stier ned til $Sink$ er nu $nPath \cdot exp(nDec + nObs + nHid)$.

En øvre grænse for $StoRCallTotal$ kan fås ved at antage, at der laves ét kald til R-DAGen per kald til en S-DAG-knude – i praksis laves der dog færre kald til R-DAGen, da vi ikke starter kald til R-DAGen fra beslutningsknuder i S-DAGen, hvorfor vi med denne antagelse får en øvre grænse. Ved hvert kald til R-DAGen skal der konditioneres over maksimalt $cond^*$ uinstantierede

chancevariable, hvor $cond^*$ er det maksimale antal uinstantierede uobserverbare chancevariable, der kan forekomme ved et kald til R-DAGen. Dvs. at der laves $O(exp(cond^*))$ kald i R-DAGen per kald til R-DAGen. En øvre grænse for $cond^*$ er $nHid$, idet der aldrig skal konditioneres over mere end $nHid$ uinstantierede variable, da de øvrige variable fra UIDet vil være instantierede eller barren (se afsnit 5.4.2 på side 90).

Det samlede antal kald til S-DAG og R-DAG-knuder, og dermed tidskompleksiteten for SC algoritmen uden cache, er således

$$\begin{aligned}
&O(SCallTotal_{\text{uden cache}}) + O(StoRCallTotal_{\text{uden cache}}) \\
&\quad = O(SCallTotal_{\text{uden cache}}) + O(SCallTotal_{\text{uden cache}}) \cdot O(exp(cond^*)) \\
&\quad = O(SCallTotal_{\text{uden cache}}) \cdot [1 + O(exp(cond^*))].
\end{aligned}$$

Ved at fjerne konstanten, 1, og udskifte $SCallTotal_{\text{uden cache}}$ med den fundne øvre grænse for $SCallTotal_{\text{uden cache}}$, får vi

$$O(nNodes \cdot nPath \cdot exp(nDec + nObs + nHid) \cdot exp(cond^*)). \tag{7.1}$$

Med cache En øvre grænse for antallet af kald ved fuld cache på S-DAGen og R-DAGen er ligeledes givet ved $O(SCallTotal \cdot (1 + exp(cond^*))$ idet vi stadig lader $exp(cond^*)$ være en øvre grænse for antallet af R-DAG-kald, når der er cache på R-DAGen. Dog skal $SCallTotal$ nu udtrykke en øvre grænse for det samlede antal kald til S-DAG-knuder med fuld cache på S-DAGen, $SCallTotal_{\text{med cache}}$.

Igen bestemmer vi en øvre grænse for antallet af kald til én knude, $SCallTo$, og ganger dette med antallet af S-DAG-knuder, $nNodes$. Udtryk (6.2) på side 99 beskrev, at antallet af kald til én knude, $\mathcal{K}$, var givet ved $\sum_{\mathcal{P} \in \pi(\mathcal{K})} SContext(\mathcal{P})^{\#}\ \mathcal{P}^{\#}$, når der er tildelt fuld cache til alle S-DAG knuder. En øvre grænse for antallet af kald til en knude, er således eksponentielt i antallet variable i mængden $SContext(\mathcal{P}) \cup \mathcal{P}$, for den knude $\mathcal{P}$, der maksimerer antallet af variable. Dette antal kalder vi $|SContext + Node|^*$. Dette skal ganges med det maksimale antal forældre, som en S-DAG-knude kan have, $|nParents|^*$. Samlet set bliver en øvre grænse for $SCallTo$ lig $|nParents|^* \cdot exp(|SContext + Node|^*)$.

Tidskompleksiteten for SC algoritmen med cache er således

$$O(SCallTotal_{\text{med cache}} \cdot exp(cond^*)),$$

og ved at indsætte den fundne øvre grænse for $SCallTotal_{\text{med cache}}$ får vi

$$O(nNodes \cdot |nParents|^* \cdot exp(|SContext + Node|^*) \cdot exp(cond^*)). \tag{7.2}$$

Sammenligning Pladskompleksiteten for SC algoritmen med fuld cache var $O(nNodes \cdot exp(|SContext|^*))$. Tidskompleksiteten for SC algoritmen er altså hårdere end algoritmens pladskompleksitet. Dette var også tilfældet med VE algoritmen, der havde en tidskompleksitet på $O(nNodes \cdot |nChildren|^* \cdot exp(w^*))$ (se afsnit 2.4), hvor $|nChildren|^*$ er det maksimale antal børn for en S-DAG-knude.

Hvis vi antager, at w^* er sammenlignelig med $|SContext + Node|^*$ og $|nParents|^*$ er sammenlignelig med $|nChildren|^*$, så er den eneste forskel på SC og VE algoritmernes tidskompleksitet, at SC algoritmen har leddet $exp(cond^*)$, hvor VE algoritmen har en konstant. Leddet $exp(cond^*)$ stammer fra beregningen af sandsynlighederne.

Da en knude aldrig behøver at have flere forældre eller børn i S-DAGen, end der er beslutningsvariable i UIDet, $nDec$, (en følge af kravene for en NFS-DAG) er det maksimale antal forældre og børn altså $nDec$, hvorfor $|nParents|^*$ og $|nChildren|^*$ er sammenlignelige i worst case.

SC og VE algoritmernes tidskompleksitet er forskellige i grænsen, hvor der er tilstrækkeligt meget plads, til VE algoritmens fordel. Forskellen skal findes i, at vi i SC algoritmen beregner alle sandsynligheder fra bunden ved ét kald til R-DAGen per sandsynlighed, der skal beregnes. Til forskel, beregner VE algoritmen alle sandsynligheder vha. dynamisk programmering, hvor alle variable langs én sti i S-DAGen kun bliver elimineret ud én gang under processen, der beregner alle sandsynligheder, der skal bruges for at løse denne sti. Således genbruger VE algoritmen det arbejde, der den tidligere har lagt i at beregne sandsynligheder, når den skal beregne nye sandsynligheder.

Hvis algoritmen, der beregner sandsynligheder vha. R-DAGen bliver erstattet af en anden algoritme, der i grænsen, hvor der er tilstrækkeligt meget hukommelse til rådighed, kan beregne alle sandsynligheder lineært i antallet af kald til S-DAG-knuder, så er der basis for, at kompleksiteten af SC og VE algoritmerne kunne blive identiske i denne grænse. Et sådan koncept kunne formentligt ligesom VE algoritmen gøre brug af dynamisk programmering.

Vi har indtil nu antaget, at w^* er større end $|SContext|^*$ og sammenlignelig med $|SContext + Node|^*$. Denne antagelse gælder i worst case situationen, hvor antallet af variable i det største potentiale, w^*, inden elimination af variablene i en knude, $\mathcal{K}$, i worst case er lig med alle variablene i fortiden for $\mathcal{K}$ og variablene i $\mathcal{K}$ selv. Dette er netop også antallet af variable i $|SContext + Node|^*$ i worst case, hvorfor w^* er sammenlignelig med $|SContext + Node|^*$ og mindre end $|SContext|^*$ i worst case. Og da sammenligningen i dette kapitel netop er en worst case sammenligning, er det altså rimeligt, at antage, at w^* er sammenlignelig med $|SContext + Node|^*$.

Del III

Empiriske resultater

Kapitel 8

Empirisk aftestning

I dette kapitel vil vi teste flere aspekter ved vores SC algoritmer. Først og fremmest vil vi sammenligne de forskellige cachingstrategier, for at bestemme hvilke strategier, der giver den bedste afvejning mellem tid og plads. I denne forbindelse vil vi sammenligne strategierne i grænsen, hvor der er vilkårligt meget plads for at undersøge hvor meget plads, der kan spares, som følge af 0-vægtene i de grådige strategier. Ligeledes vil vi undersøge, om det kan betale sig at have cache på R-DAGen, eller om overheadet ved at administrere denne cache overstiger besparelserne ved at anvende cachen.

Vi vil yderligere undersøge, om tiden per kald er stabil, hvilket i så fald vil muliggøre en estimering af kørselstiden vha. vores opstillede udtryk for det samlede antal rekursive kald (udtryk (6.5) på side 101).

Desuden vil vi undersøge, hvordan vores konditioneringsalgoritmer yder ift. VE algoritmen i grænsen, hvor der er vilkårligt meget plads til rådighed.

Tilsidst vil vi se på SC algoritmens basispladsforbrug og pladsforbruget som følge cachetildelingen.

Inden vi starter med at præsentere og diskutere de empiriske resultater, i afsnit 8.3, beskriver vi først, i afsnit 8.1, hvilke testmængder vi tester op imod, og efterfølgende, i afsnit 8.2, præsenteres hvilke algoritmer vi tester.

8.1 Testmængder

Vi ønsker at opstille testmængder bestående af UIDer, som vi kan teste vores løsningalgoritmer op imod. Det skal for vores testmængder gælde, at UIDerne

i én testmængde til en vis grad er lige krævende at løse.

I [Ahlmann-Ohlsen and Pedersen, 2005] argumenterede vi for, at man kan klassificere UIDer ud fra nedenstående parametre, og dermed opnå en klassificering af UIDer ift. hvor krævende de er at løse.

1. Antallet af beslutningsknuder ($nDec$)
2. Antallet af observerbare chanceknuder ($nObs$)
3. Antallet af uobserverbare chanceknuder ($nHid$)
4. Antallet af nytteknuder ($nUtil$)
5. Det maksimale antal forældre og børn til chanceknuder og det maksimale antal forældre til nytteknuder ($nDeg$)
6. Antallet af stier i NFS-DAGen ($nPath$)

I kapitel 7 argumenterede vi for, at en række S-DAG specifikke parametre kunne udgøre en øvre grænse for tid- og pladskompleksiteten af at løse en S-DAG med SC algoritmen. Da de ovenstående parametre er UID specifikke (på nær $nPath$), er disse parametre i sig selv ikke repræsentative for den eksakte kompleksitet af at løse et UID, da det vil kræve, at fx UIDets NFS-DAG først opstilles og analyseres. Dog vil vi stadig bruge de ovenstående parametre til overordnet at klassificere UIDer i grupper efter deres forventede kompleksitet.

Da vi ikke har kendskab til nogle praktisk anvendte UIDer, har vi testet vores algoritmer op imod en række autogenererede UIDer samt UIDerne fra figur 2.8 på side 39 (Monitoring4) og figur 2.9 på side 40 (Tests4). UIDet fra figur 2.8 er skaleret op til 4 inspektioner med test og behandlinger.

I bilag E har vi beskrevet en algoritme, der kan autogenerere UIDer ud fra de første 5 parametre i listen ovenfor. Ved at lave NFS-DAGe for disse autogenerede UIDer og tælle antallet af forskellige maksimalt orienterede stier i disse bestemmes den sidste parameter.

På denne måde har vi genereret testmængderne TS4, TS4h, TS3, TS7 og TS16. Disse testmængder er opsummeret i tabel 8.1 sammen med de 2 andre UIDer. I tabel 8.1 er $nUID$ antallet af UIDer i testmængden.

Navn	$nUID$	$nDec$	$nObs$	$nHid$	$nUtil$	$nDeg$	$nPath$
TS4	100	4	8	4	4	3	3
TS4h	20	4	8	12	4	3	3
TS3	20	3	16	2	16	2	3
TS7	20	7	14	7	7	3	12
TS16	20	16	4	2	3	6	1
Monitoring4	1	12	8	5	13	3	1
Tests4	1	4	5	6	5	5	24

Tabel 8.1: En oversigt over vores testmængder.

8.2 Testoversigt

Vi har implementeret SC algoritmen med 4 forskellige cachingstrategier, som vi har navngivet: Naive0, Naive1, Greedy0 og Greedy1. Første del af navnet angiver, om det er den naive eller den grådige strategi, og tallet tilsidst i navnet angiver, om vi har ingen (0) eller fuld cache (1) på R-DAGen. Derudover har vi implementeret VE algoritmen.

VE algoritmen som den er beskrevet i afsnit 2.3 på side 34 er en generel algoritme til løsning af S-DAGe, hvorfor vi har prøvet at teste VE algoritmen imod både optimerede og uoptimerede S-DAGe. Med en "optimeret S-DAG" mener vi en S-DAG, hvor de observerbare variable er flyttet frem til de bliver relevante og de uobserverbare variable er flyttet tilbage som beskrevet i afsnit 5.3. Når VE algoritmen køres på uoptimerede S-DAGe på anden normalform (se definition 2.10 på side 33) kalder vi den VE2, og når VE algoritmen køres på optimerede S-DAGe på anden normalform kalder vi den VE3. Navnet VE3 er valgt, da man i princippet kan se vores optimeringer til S-DAGen som en tredje normalform, når optimeringerne udføres på en S-DAG, der i forvejen var på anden normalform.

For en oversigt over vores testmaskine og de anvendte måleteknikker henviser vi til bilag C, og for en gennemgang af vores implementering henviser vi til bilag D.

8.3 Empiriske resultater

Vi starter nu præsentationen og diskussionen af vores empiriske resultater. Baseret på testopsætningen, som den er præsenteret i bilag C, er der udført en række test af de nævnte algoritmer på de opstillede testmængder. I dette afsnit vil vi præsentere resultaterne af disse test og i forlængelse heraf diskutere og vurdere resultaterne.

8.3.1 Sammenligning af cachingstrategierne

Vi starter med at sammenligne, hvordan vores forskellige cachingstrategier klarer sig ift. hinanden på TS4. Tabel 8.2 opsummerer SC og VE algoritmernes pladsforbrug ved løsning af testmængde TS4. Tilsvarende tabeller for de øvrige testmængder kan ses i bilag F. I tabel 8.4 opsummerer vi pladsforbruget for alle algoritmerne på alle testmængderne.

I tabel 8.2 er der i kolonnen "Indgange" angivet hvor mange indgange, der er i SC algoritmens cache samt i de potentialemængder, som VE algoritmen har genereret og gemt på alle knuderne i S-DAGen ved terminering.

TS4	Indgange	Heap size (KB)			
Alg.	Avg. (·1024)	Avg.	Std.	Min.	Max.
Naive0	1,057	11,51	6,330	4,398	42,80
Naive1	1,077	11,58	6,305	4,367	42,66
Greedy0	0,3054	5,510	2,043	3,086	13,52
Greedy1	0,3256	5,539	2,053	3,102	13,38
VE2	2,916	55,25	11,39	35,98	101,8
VE3	1,915	53,92	8,594	39,03	83,49

Tabel 8.2: Denne tabel viser pladsforbruget for SC algoritmens cache og VE algoritmens potentialemængder for TS4, når der er vilkårligt meget hukommelse til rådighed.

I kolonnerne for "Heap size" er der angivet hvor mange KB, der er blevet tilføjet til Javas heap, når S-DAGen er blevet evalueret ift. heapens størrelse umiddelbart inden S-DAGen blev evalueret. For at få algoritmernes samlede heap-forbrug skal der til de angivne værdier adderes en basisallokering til at repræsentere UIDet og S-DAGen samt selve løsningskoden. Størrelsen af denne basisallokering er ens for alle algoritmerne, hvorfor den behandles separat i afsnit 8.3.6.

Tabel 8.3 viser SC og VE algoritmernes tidsforbrug ved løsning af S-DAGene for UIDerne i TS4. De tilsvarende tabeller for de øvrige testmængder er medtaget i bilag F, og i tabel 8.5 har vi opsummeret tidsforbruget for alle algoritmerne på alle testmængderne.

TS4	Kald	Tid (sek.)			
Alg.	Avg. ($\cdot 10^6$)	Avg.	Std.	Min.	Max.
Naive0	0,1736	3,349	3,430	0,079	15,60
Naive1	0,1681	3,727	3,940	0,078	18,42
Greedy0	0,1736	3,330	3,441	0,078	15,64
Greedy1	0,1681	3,705	3,914	0,078	17,67
VE2		0,3277	0,4529	0,031	2,672
VE3		0,2893	0,2814	0,016	1,718

Tabel 8.3: Denne tabel viser tidsforbruget for SC og VE algoritmerne på TS4, når der er vilkårligt meget hukommelse til rådighed.

I tabel 8.3 angiver kolonnen "Kald", hvor mange rekursive kald, der er blevet lavet af SC algoritmerne. "Tid"-kolonnerne angiver, hvor mange sekunder, der gennemsnitligt bliver brugt på at evaluere de forskellige S-DAGe.

0-vægte Hvis vi sammenligner de naive strategier med de grådige strategier i tabel 8.2, så kan vi se, at de grådige strategier bruger færre indgange til cache end de naive strategier. Sammenligner vi det gennemsnitlige antal cache-indgange, brugt under løsningen af UIDerne i den pågældende testmængde,

T. m.	Plads (Heap-size i KB) Naive0	Naive1	Greedy0	Greedy1	VE2	VE3
TS4	11,51	11,58	5,510	5,539	55,25	53,92
TS4h	11,77	12,03	6,303	6,487	58,86	56,11
TS3	207,3	207,4	84,823	84,822	473,1	440,9
TS7	549,0	548,9	275,90	275,92	638,4	539,6
TS16	367,43	367,52	171,52	171,55	235,8	230,4
Mon.4	12159,2	12159,0	2,977	3,594	24354,91	24354,95
Tests4	35,85	37,21	19,26	20,46	154,9	151,4

Tabel 8.4: En opsummering over det gennemsnitlige pladsforbrug ved løsning af UIDerne i de forskellige testmængder.

T. m.	Tid (sek.) Naive0	Naive1	Greedy0	Greedy1	VE2	VE3
TS4	3,349	3,727	3,330	3,705	0,3277	0,2893
TS4h	84,46	26,76	82,15	26,51	0,9242	0,9203
TS3	355,1	375,5	353,9	374,6	34,06	35,31
TS7	469,3	439,1	465,9	442,4	24,24	23,63
TS16	38,47	41,70	38,60	40,57	7,601	7,451
Mon.4	5562	5177	5451	5390	1136	1131
Tests4	81,90	98,69	80,66	98,45	1,188	2,844

Tabel 8.5: En opsummering over det gennemsnitlige tidsforbrug til løsning af UIDerne i de forskellige testmængder.

for strategierne uden cache på R-DAGen, ser vi, at det gennemsnitlige antal indgange til cache for Greedy0 er $\frac{(1{,}057-0{,}3054)\cdot 100}{1{,}057} \approx 71\%$ mindre end det gennemsnitlige antal indgange for Naive0. Årsagen til at Greedy0 i gennemsnit bruger mindre plads end Naive0, er at Greedy0 ikke tildeler cache til knuder, hvis denne cache ikke medfører et fald i det samlede antal kald – jf. vores diskussion af 0-vægte i afsnit 6.2.3 på side 107. Disse færre cacheindgange medfører også et lavere pladsforbrug jf. værdien i Heap-size "Avg."-kolonnen i tabel 8.2.

Kigger vi i stedet på evalueringstiden, og sammenligner værdierne for denne for Naive0 og Greedy0 i tabel 8.3, kan vi se, at SC algoritmens kørselstid kan minimeres ved ikke at tildele cache til S-DAG-knuder med 0-vægte. Dette skyldes, at Greedy0 algoritmen slipper for overheadet med at tjekke for en værdi i cachen, hvis cachetildelingen for en knude er 0. Da det samlede antal

rekursive kald er det samme for Naive0 og Greedy0, medfører det en hurtigere evalueringstid, da vi kan undgå dette cache-tjek-overhead. Gennemsnittet for kørselstiden for Greedy0 på TS4 er $\frac{(3{,}349s-3{,}330s)\cdot 100}{3{,}349s} \approx 0,57\%$ mindre end den gennemsnitlige kørselstid Naive0.

Denne reduktion i plads- og tidsforbruget er generel for alle testmængderne både med og uden cache på R-DAGen (se tabel 8.4, tabel 8.5 og bilag F), hvorfor de grådige strategier må vurderes at være at foretrække fremfor de naive strategier i grænsen, hvor der er vilkårligt meget plads til rådighed. Dette resultat passer med vores forventninger, men kan dog pga. den lille forskel på kun $0,57\%$ også skyldes måleusikkerhed som følge af, at vi måler systemtid, og resultatet af en sådan måling vil afhænge af hvilke andre processer, der har benyttet CPUen samtidigt med, at testene blev afviklet. En gentagelse af testene vil kunne styrke eller svække vores tro på disse resultater.

Der er én undtagelse fra princippet om, at de grådige algoritmer er hurtigere end de tilsvarende naive algoritmer, og det er på TS16, hvor Greedy0s gennemsnitlige tidsforbrug er $\frac{(38{,}60s-38{,}47s)\cdot 100}{38{,}47s} \approx 0,34\%$ højere end Naive0s gennemsnitlige tidsforbrug. Dette resultat tilskriver vi måleusikkerhed.

Cache på R-DAGen Ved at sammenligne værdierne i kolonnen "Indgange" i tabel 8.2 for SC algoritmerne både med og uden cache på R-DAGen – dvs. Naive0 og Greedy0 sammenlignes med hhv. Naive1 og Greedy1 – kan vi se, at gennemsnittet for antallet af indgange som Naive1 benytter er $\frac{(1{,}077-1{,}057)\cdot 100}{1{,}057} \approx 1,9\%$ højere end gennemsnittet af antallet af indgange, som Naive0 benytter. Tilsvarende er Greedy1s gennemsnit $\frac{(0{,}3256-0{,}3054)\cdot 100}{0{,}3054} \approx 6,6\%$ højere end Greedy0s gennemsnit. Med cache på R-DAGen falder det gennemsnitlige antal kald, i tabel 8.3, der skal udføres for hvert par af cachingstrategier med $\frac{(0{,}1736-0{,}1681)\cdot 100}{0{,}1736} \approx 3,2\%$ ift. det gennemsnitlige antal kald, der skal udføres, hvis der ikke er cache på R-DAGen. Det er således begrænset hvor mange rekursive kald, der i gennemsnit kan spares, ved at tilføje cache til R-DAGen i TS4, selvom det ekstra plads, der genemsnitligt kræves, ligeledes er begrænset ift. cachen på S-DAGen. Dette passer med vores forventninger jf. vores diskussion i afsnit 6.3 på side 108.

Ved at sammenligne værdierne i "Avg." kolonnen under tid i tabel 8.3 kan vi desuden se, at algoritmerne, der har brugt R-DAG cache, sammenlagt har brugt mere tid end algoritmerne, der ikke har håndteret R-DAG cache. Dette skyldes, at disse algoritmer har et overhead med at administrere cachen på R-DAGen, der åbenbart ikke kan udlignes af besparelsen i antallet af rekursive kald.

I testmængde TS4h (tabel F.1 og F.2 på side 163) har vi prøvet at

fremtvinge en situation, hvor det kan betale sig at indføre cache på R-DAGen ved at teste SC algoritmen på UIDer med mange uobserverbare chancevariable ift. antallet af beslutningsvariable og ift. de øvrige testmængder. I en sådan testmængde er der mulighed for, at der vil være flere uinstantierede variable, der skal konditioneres over, hvorfor der kan blive en større reduktion i antallet af rekursive kald ved anvendelsen af cache (se afsnit 5.4.2 på side 85). Det er dog ikke sikkert, at der bliver en større reduktion i antallet af kald, da de uobserverbare chancevariable stadig kan blive instantierede i lighed med de observerbare chancevariable, da de kan være blevet flyttet fra *Sink* op til de øvrige chanceknuder i S-DAGen.

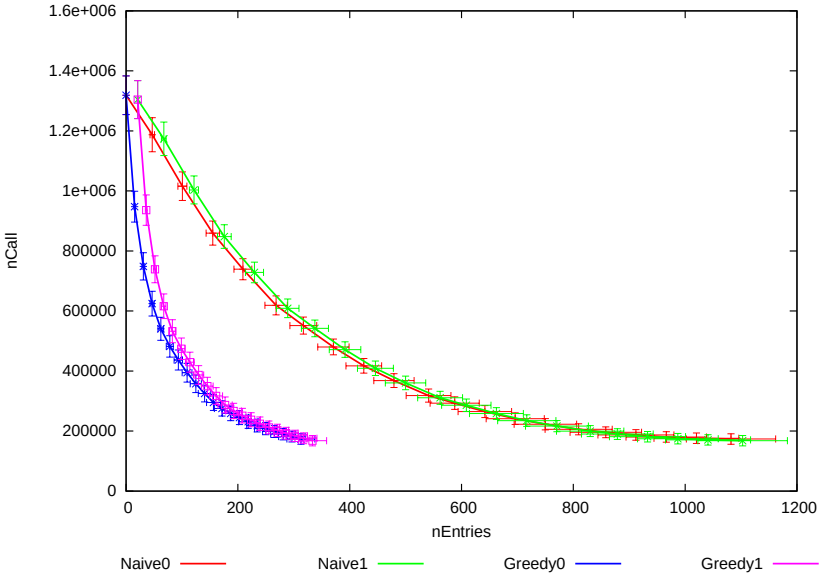
I tabel F.2 på side 163 kan vi se, at der i TS4h i gennemsnit spares $\frac{(1{,}354-0{,}2534)\cdot 100}{1{,}354} \approx 81\%$ rekursive kald ved at have cache på R-DAGen for både den grådige såvel som den naive caching strategi. Der er altså procentvis flere rekursive kald at spare ved at have cache på R-DAGen i TS4h ift. TS4. Målt i (gennemsnitlig) evalueringstid er Greedy1 $\frac{82{,}15s}{26{,}51s} \approx 3$ gange hurtigere end Greedy0, hvorfor vi kan konkludere, at det i TS4h ift. TS4 bedre kan betale sig at have cache på R-DAGen. For TS7 og Monitoring4 gælder det også, at tidsbesparelsen, der følger af reduktionen i antallet af kald ved cache på R-DAGen, mere end udligner overheadet med at håndtere cache på R-DAGen (se tabel 8.5 på side 119 eller tabel F.6 på side 164 og 8.7 på side 130).

Sammenfattet vurderer vi, at det ikke kan betale sig at håndtere cache på R-DAGen, hvis antallet af uobserverbare variable er "tilpas lavt", da antallet af kald, der spares ved denne cache ikke kan ophæve overheadet med at håndtere cachen. Men hvis der er "tilpas mange" uobserverbare variable, kan det godt betale sig, at håndtere cache på R-DAGen. Nærmere undersøgelser kan afgøre, hvor grænsen, for om R-DAG cache kan betale sig, ligger.

8.3.2 Afvejning af tid og plads - teoretisk

I dette afsnit vil vi se på, hvordan antallet af rekursive kald for SC algoritmerne afhænger af antallet af indgange, der afsættes cache til. Dette vil vi gøre ved at opstille grafer, der afbilder denne sammenhæng. Vi bruger termen "nEntries" til at betegne antallet af indgange, der afsættes cache til, og "nCalls" til at betegne antallet af rekursive kald, der udføres. Grafen for TS4 kan ses i figur 8.1. De tilsvarende grafer for de øvrige testmængder kan ses i bilag F.

På figur 8.1 kan man bl.a. se, at de grådige strategier medfører færre kald end de naive strategier for det samme antal anvendte cache-indgange. Dette gør sig også gældende for de øvrige testmængder. De grådige strategier må altså være at foretrække ift. de naive uanset hvor meget plads, der er til



Figur 8.1: $(nEntries, nCalls)$ for TS4. Figuren afbilder antallet af rekursive kald, der skal udføres, for at løse UIDerne i TS4, som funktion af den tilgængelige hukommelse til cache. Fejllinierne angiver standardafvigelsen af gennemsnittet.

rådighed. Dette passer med vores intuition.

Derudover kan man se, at de grådige strategier når deres minimum i antallet af rekursive kald ved et lavere antal cache-indgange end for de naive strategier, hvilket også kommer til udtryk i tabel 8.2. På figuren giver det sig til udtryk ved at kurverne for Greedy0 og Greedy1 ikke er defineret for lige så høje "nEntries"-værdier som Naive0 og Naive1. Dette skyldes, at der ikke tildeles cache til knuder med 0-vægte.

Generelt for alle graferne gælder det, at antallet af rekursive kald kun vokser "moderat", hvis man begynder at *fjerne* cacheindgange fra en *fuld cachetildeling*. Dette er en god egenskab, da det formentligt vil medføre, at vi kan opnå en kørselstid tæt på den optimale kørselstid, hvis den plads, vi har til rådighed, er sammenlignelig med mængden af cache, der kræves til fuld cache. Hvis vi modsat begynder at tilføje cache til en S-DAG, der ikke har cache i forvejen, så falder antallet af kald hurtigt med den første cache, der bliver tilføjet. Dette er også en god egenskab, da vi således kan reducere algoritmens eksekveringstid betragteligt – selv når der kun er lidt plads til rådighed.

Vores bedste bud på cachingstrategier ser ud til at være de grådige strategier, da de giver den bedste afvejning mellem plads og tid ift. de naive strategier. Yderligere har vores test vist, at det afhænger af UIDet, hvorvidt det kan betale sig at håndtere cache på R-DAGen.

Lad os se på, hvor godt det estimerede antal rekursive kald modsvarer det faktiske antal rekursive kald. Antallet af rekursive kald afbildet i figurene og præsenteret tabellerne er bestemt ved udtryk (6.5) på side 101. Vi har testet, hvor godt dette udtryk er til at estimere det faktiske antal rekursive kald, og resultatet for de grådige strategier blev, at estimatet svarede til det faktisk målte antal rekursive kald – plus/minus ét kald. For de naive strategier er det estimerede antal kald optimistisk som følge af, at vi ikke håndterer situationer, der følger af afrunding. Hvis cachetildelingen for alle knuderne fx er $0,73$ og konteksten for en knude, $\mathcal{K}$, indeholder 4 indgange, så bliver der kun afsat $\lfloor 0,73 \cdot 4 \rfloor = 2$ indgange til $\mathcal{K}$. Det medfører at den reelle cachetildeling for $\mathcal{K}$ er $0,5$, mens der i det teoretiske udtryk regnes med en cachetildeling på $0,73$, hvorfor det teoretiske udtryks estimat er optimistisk for den naive strategi. Alligevel har vi brugt det teoretiske udtryk til at bestemte antallet af rekursive kald, da det er hurtigere at beregne antallet af rekursive kald, end det er at kalde SC algoritmen på alle UIDernes NFS-DAGe for at bestemme det faktiske antal rekursive kald. Samtidigt mener vi, at formlen stadig giver en god estimering af det samlede antal kald.

I grænsen hvor alle cachetildelinger er 0 eller 1, er det estimerede antal kald for de naive strategier også lig med det faktiske antal kald. Dette gælder, da der ikke er nogen usikkerhed som følge af afrunding.

8.3.3 Estimering af kørselstid

Lad os diskutere, hvordan vi kan estimere kørselstiden for SC algoritmen med Greedy0 cachingstrategien – strategien, vi vurderer til at være vores bedste cachingstrategi, hvis antallet af uobserverbare variable er tilpas lavt. Dette vil vi gøre vha. udtryk (6.5) på side 101. For at estimere den samlede kørselstid ud fra det samlede antal kald, skal vi kende afviklingstiden per rekursivt kald. Tabel 8.6 opsummerer hvor mange kald og meget tid der i gennemsnit bliver brugt.

Af tabel 8.6 fremgår det, at tiden per kald i gennemsnit for hver testmængde ligger imellem $0,903 \cdot 10^{-5}$ og $10,1 \cdot 10^{-5}$ sekunder for alle testede UIDer. Sammenligner vi alle UIDer fra samtlige testmængder, kan vi bestemme variationskoefficienten til 52%. Tiden per kald varierer altså fra problem til problem. Hvis vi skelner mellem S-DAG og R-DAG-kald bliver tiden per kald ikke mere stabil, da variationskoefficienten for S-DAG-kaldene kan beregnes til 214% og for R-DAG-kaldene 55%.

Denne variation kan vi forklare ved at se nærmere på, hvilke operationer der udføres under S-DAG og R-DAG-kald. Kald til S-DAG og R-DAG-knuder

Test mgd.	Avg. Kald, $(\cdot 10^6)$	Avg. Tid, (s)	Avg. Tid/Kald, $\bar{t_S}$ $(\cdot 10^{-5} s)$	Std. Tid/Kald, $(\cdot 10^{-5} s)$	Min. Tid/Kald, $(\cdot 10^{-5} s)$	Max. Tid/Kald, $(\cdot 10^{-5} s)$	C.V. Tid/Kald, (%)
TS4	0,174	3,33	1,83	0,257	1,34	2,44	14
TS4h	1,35	82,1	3,42	2,08	1,42	10,1	61
TS3	7,78	354	4,02	0,57	3,18	4,92	14
TS7	18,0	466	2,43	0,36	1,95	3,30	15
TS16	1,60	38,6	3,19	1,98	1,36	7,24	62
Monitoring4	604	5456	0,903	0	0,903	0,903	0
Tests4	5,40	80,7	1,50	0	1,50	1,50	0
Samlet	6,60	135	2,45	1,27	0,903	10,1	52

Tabel 8.6: Tabellen viser gennemsnitsværdier for antallet af kald og tidsforbrug for at udføre disse kald, samt tiden *per* kald for UIDerne i de enkelte testmængder. Derudover vises standardafvigelsen, min., max. og variationskoefficienten over tiden per kald (for UIDerne i de forskellige testmængder). Testmængden "Samlet" indeholder alle de UIDer, der er testet imod. Tabellen er for Greedy0.

indebærer bl.a. at der itereres over alle tilladelige instantieringer over UID-variablene i knuderne. Disse instantieringer skal registreres og fjernes igen, og for hver af instantieringerne skal der laves opslag i sandsynlighedspotentialerne eller i de frigivne nyttepotentialer. For kald til en S-DAG-knude skal der yderligere startes kald til alle børnene af S-DAG-knuden. Samtidigt varierer antallet af børn fra S-DAG-knude til S-DAG-knude. Da antallet af UID variable per knude ikke er konstant for alle knuderne i en S-DAG og en R-DAG, og da antallet af børn, sandsynlighedspotentialer og frigivne nyttepotentialer heller ikke er konstant, forventer vi, at tiden per S-DAG og R-DAG-kald heller ikke blive stabil. Det er derfor tvivlsomt, om et mere præcist estimat for kørselstiden kan opnås direkte med udtryk (6.5).

Variationskoefficienten er her udregnet for en fiktiv testmængde indeholdende alle de testede UIDer. Rimeligheden i at udregne variationskoefficienten på denne måde kan diskuteres, da resultatet således vil bære mere præg af de 100 UIDer i TS4 end den ene UID i fx Tests4 (se tabel 8.1). Variationen i gennemsnittet vil dog næppe blive mindre, hvis den samlede testmængde var mere blandet sammensat.

Hvis antallet af UID variable, børn, sandsynlighedspotentialer og nyttepotentialer er stabilt mellem alle S- og R-DAG-knuderne i en S-DAG (og den

tilhørende R-DAG), så kan det estimerede antal kald stadig være proportionalt med den faktiske kørselstid. Dog må det forventes, at proportionalitetsfaktoren vil variere S-DAGene imellem.

Da vi opstillede vores cachingstrategier, antog vi, at tiden per kald til S-DAG-knuder og R-DAG-knuder var konstant og ens S- og R-DAG-knuder imellem. Vi har imidlertid bestemt den gennemsnitlige tid per S- og R-DAG-kald til henholdsvis $12 \cdot 10^{-5}$ og $1,8 \cdot 10^{-5}$ sekunder med standardafvigelser på $23 \cdot 10^{-5}$ og $0,98 \cdot 10^{-5}$ sekunder. Vores antagelse (fra afsnit 6.2) om, at disse 2 slags kald tager lige lang tid passer altså ikke. Den gennemsnitlige tid per S-DAG-kald er altså højere end den gennemsnitlige tid per R-DAG-kald. Når vi i vores grådige strategi snæversynet forsøger at reducere det samlede antal kald mest muligt, svarer dette derfor ikke nødvendigvis til at reducere den samlede kørselstid mest muligt. For at opnå dette, kan man vægte antallet af hhv. S-DAG og R-DAG-kald med den gennemsnitlige tid, disse kald tager. Ved at fjerne antagelsen om at S-DAG og R-DAG-kald tager lige lang tid og ved at identificere, hvad der er afgørende for, hvor lang tid kaldene tager, kan man muligvis optimere algoritmernes kørselstid i situationer, hvor der ikke er plads til fuld caching, i forhold til de eksisterende cachingstrategier.

Det vil kræve en nærmere analyse af vores aktuelle implementering at fastslå, hvad der præcist afgør tiden per S-DAG og R-DAG-kald.

I [Darwiche, 2001] postuleres, at kørselstiden er proportional med antallet af kald ved sandsynlighedsinferens over d-træer. Da tiden per R-DAG-kald ikke er stabil og da konditionering på R-DAGen har flere ligheder med konditionering over d-træer (bl.a. at antallet af variable per knude ikke er konstant), kan vi stille spørgsmålstegn ved dette postulat.

8.3.4 Afvejning af tid og plads - empirisk

I figur 8.2 har vi afbildet den faktiske kørselstid og det teoretisk bestemte antal rekursive kald som funktion af det antal cacheindgange, som vi har specificeret at algoritmerne må bruge til cache. Graferne er for henholdsvis Tests4 og UIDet fra TS4h vist i figur E.1 på side 161. Hver kurve i figuren er baseret på 20 forskellige cachetildelinger, fordelt således at det samlede pladsforbrug er jævnt fordelt imellem strategiernes minimale og maksimale pladsforbrug. 2NFS-DAGen for Tests4 kan ses i figur A.1 på side 149 og den optimerede 2NFS-DAG for UIDet fra TS4h kan ses i figur E.2. Antallet af børn per S-DAG-knude varierer i begge NFS-DAGe, og når 2NFS-DAGen for Tests4 bliver optimeret, kommer antallet af UID variable per S-DAG-knude

også til at variere for begge NFS-DAGe. Tiden per kald må således forventes at variere fra knude til knude i disse NFS-DAGe.

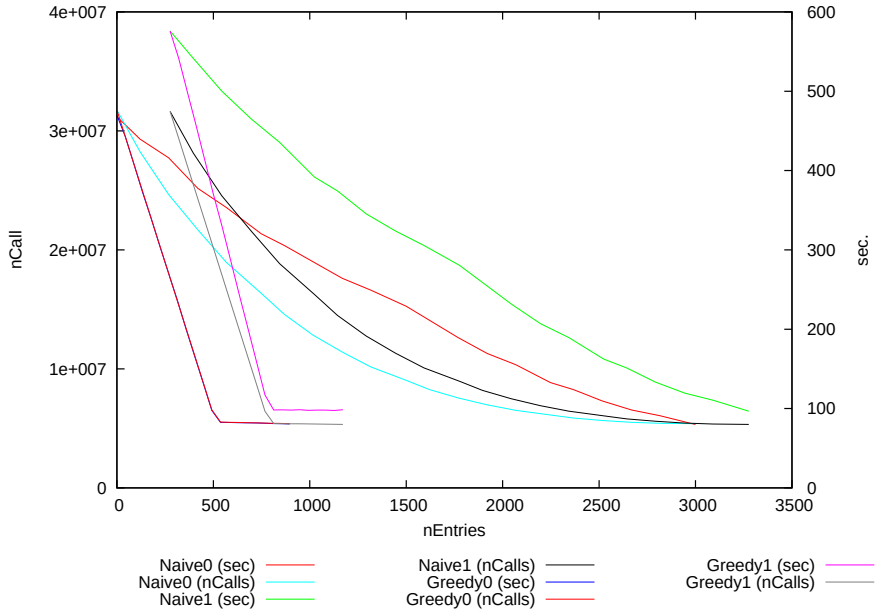
I figur 8.2 kan man se, at den teoretisk bestemte afvejning mellem tid og plads for de grådige algoritmer er proportional med den praktisk bestemte afvejning mellem tid og plads. Således har det været muligt at skalere de to andenakser så kurven for det teoretisk estimerede antal kald for Greedy0 stort set er sammenfaldende med kurven for den faktiske kørselstid. På figur 8.2(a) er skaleringsforholdet $1,50 \cdot 10^{-5}$ sekunder per kald, hvilket også passer med værdien for tid per kald i tabel 8.6, der bl.a. viser Greedy0s kørselstid og antallet af rekursive kald ved fuld cache på Tests4. Således bliver den teoretiske og den praktiske kurve for Greedy0 nærmest sammenfaldende.

Hvis skaleringsforholdet i stedet vælges til $\frac{98,45s}{5,336 \cdot 10^6} \approx 1,84 \cdot 10^{-5}$ sekunder per kald (se evt. tabel F.9), bliver kurverne for Greedy1 sammenfaldende (se figur F.1 på side 165 hvor dette skaleringsforhold er valgt). I figur 8.2(b) er skaleringsforholdet valgt til $2,00 \cdot 10^{-5}$ hvilket gør, at de teoretiske og praktiske kurver for både Greedy0 og Greedy1 nærmest er sammenfaldende. I Tests4 og UIDet fra TS4h er det estimerede antal kald således proportionalt med den faktiske kørselstid for de grådige strategier.

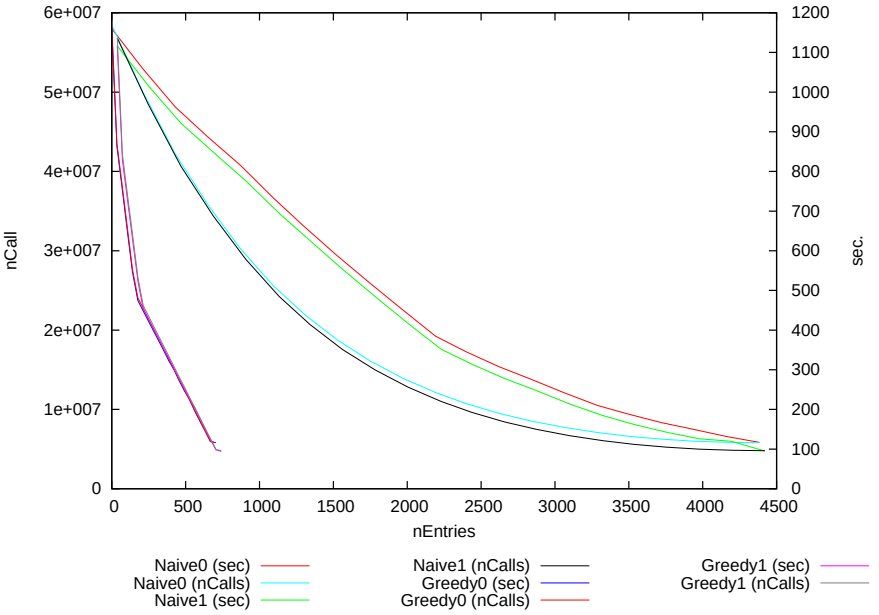
Ud fra de viste figurer synes proportionalitetsfaktoren uafhængig af fordelingen af UID variable og potentialer i de enkelte S-DAG-knuder, men dog afhængig af den enkelte S-DAG og cachingstrategi. Således tyder det på, at kørselstiden på denne måde kan estimeres, hvis man holder cachingstrategien og S-DAGen konstant. Da vi forventer, at kørselstiden per kald er forskellige fra S-DAG-knude til S-DAG-knude, forventer vi også, at proportionalitetsfaktoren ville være afhængig at den konkrete cachetildeling til S-DAG-knuderne. Dette synes imidlertid ikke at være tilfældet i de viste grafer, hvorfor mere forskning på området vil være nødvendig for at bestemme, hvad der er afgørende for tiden per kald. Denne yderligere forskning ville, som nævnt i afsnit 8.3.3, formentlig også kunne anvendes til at bestemme en bedre cachingstrategi.

Hvis det viser sig at gælde generelt, at der kan bestemmes én proportionalitetsfaktor givet én S-DAG og én cachingstrategi, så betyder det, at man med denne proportionalitetsfaktor og udtryk (6.5) kan bestemme kørselstiden for SC algoritmen forholdsvis præcist.

I afsnit 8.3.2 argumenterede vi for, at det teoretisk udregnede antal rekursive kald var optimistisk ift. det faktiske antal rekursive kald for de naive strategier (når vi ikke var i grænsesituationerne med ingen eller fuld cache). Dette kan også ses i figur 8.2, da kurverne for det teoretisk bestemte antal rekursive kald ligger under kurverne for den faktiske kørselstid imellem græn-



(a) Tests4.



(b) Et UID fra TS4h.

Figur 8.2: Figuren afbilder det teoretisk bestemte antal rekursive kald og den faktiske kørselstid som funktion af den tilgængelige hukommelse til cache. **Bemærk** at kurverne for "Greedy0 (sec)" og "Greedy0 (nCalls)" er sammenfaldene i figur 8.2(a) og kurverne for "Greedy0 (nCalls)" og "Greedy1 (nCalls)" er sammenfaldende med kurverne for "Greedy0 (sec)" og "Greedy1 (sec)" i figur 8.2(b).

sesituationerne, hvor der er ingen og fuld cache på S-DAGen. Dette passer med forventningerne, da det estimerede antal kald ($nCalls$-kurverne) forventes at være lavere end det faktiske antal kald (ikke vist), og da det faktiske antal kald ser ud til at være proportionalt med kørselstiden (sec-kurverne).

8.3.5 SC versus VE algoritmen

I dette afsnit sammenligner vi SC algoritmerne med VE algoritmerne i grænsesituationen, hvor der er vilkårligt meget hukommelse til rådighed.

Pladsforbrug Ideelt set ville vi sammenligne SC og VE algoritmernes maksimale pladsforbrug under afviklingen af disse, men i Java kan vi ikke måle det største heap-forbrug under algoritmens afvikling, hvorfor vi her har angivet ændringen i heapens størrelse som følge af, at S-DAGen er blevet evalueret. Vi antager, at denne ændring i heapens størrelse, for SC algoritmernes vedkommende, vil skyldes den cache, der er blevet allokeret til knuderne. For VE algoritmerne forventer vi at ændringen skyldes de potentialemængder, der er genereret og knyttet til knuderne. Dog kan ændringen i heapens størrelse også skyldes døde objekter i heapen, som Javas garbage collector ikke har fået fjernet. Der er mulighed for at algoritmerne har brugt mere heapplads undervejs, hvilket vi ikke får målt på denne måde. For SC algoritmerne vil dette ekstra forbrug formentligt være lavt, da SC algoritmen ikke bruger heap-plads til andet end cachen og et konstant antal lokale variable. For VE algoritmen derimod, kan det ekstra pladsforbrug være betydeligt, som det vil fremgå af følgende forklaring:

Husk på, at når VE algoritmen skal beregne et sandsynlighedspotentiale, ϕ_C, til potentialemængden, Φ, på en S-DAG-knude med chancevariablen C, beregnes dette potentiale som $\phi_C = \sum_C \prod \Phi_C$, hvor Φ_C er mængden af potentialer med C i domænet. Pladsforbruget som vi registrerer vil være pladsforbruget for ϕ_C. VE algoritmen har dog, inden den er nået frem til dette resultat, brugt heap-plads til at repræsentere et potentiale med variablene $\{C\} \cup dom(\phi_C)$. Dette potentiale vil være dobbelt så stort som ϕ_C, (da alle variable i vores test er binære). Da der arbejdes med knuder med mængder af variable, kan de mellemliggende potentialer have været mere end dobbelt så store. Der er altså risiko for, at VE algoritmen har brugt mere plads, end dét vi måler, når vi måler heapens størrelse, når S-DAGen er færdigevalueret.

Hvis vi sammenligner SC algoritmernes pladsforbrug med VE algoritmernes pladsforbrug i tabel 8.2, ser vi, at alle fire SC algoritmer har et lavere pladsforbrug end VE2 og VE3 algoritmerne – både i gennemsnit, min. og

max., samt i antallet af indgange i de gemte cache/potentialemængder. SC algoritmerne har altså et generelt lavere pladsforbrug end VE algoritmerne. Dette resultat går igen for Greedy algoritmerne i alle testmængderne (se tabel 8.4 og bilag F). For Naive-algoritmerne går det ligeledes igen i alle testmængderne på nær TS16, hvor VE2 og VE3 har et lavere pladsforbrug end de naive algoritmer og på TS7, hvor VE3 har et lavere pladsforbrug end de naive algoritmer.

For TS4 og TS4h gælder det, at de grådige strategier bruger mindre plads end VE2 og VE3, hvis vi sammenligner det maksimale pladsforbrug for de grådige strategier imod VE algoritmernes minimumspladsforbrug.

At SC algoritmerne har et lavere pladsforbrug end VE algoritmerne kan forklares ud fra algoritmernes pladskompleksitet og ud fra, at cachen i SC algoritmerne ikke vil indeholde irrelevante variable jf. overvejelser om relevant fortid og relevant fortid for knuderne i de efterfølgende del-scenarier. Ved VE algoritmerne kan der imidlertid godt forekomme irrelevante variable i domænerne af de potentialer, der arbejdes med i VE algoritmen [Nielsen and Jensen, 1999].

Ved at sammenligne pladsforbruget for VE2 og VE3 (fx i tabel 8.4) kan vi se, at VE algoritmens pladsforbrug også bliver forbedret når S-DAGene bliver optimeret. Vores optimeringer af S-DAGen har altså positiv indflydelse på VE algoritmens pladsforbrug, selvom de oprindeligt blev indført for at reducere konteksten på S-DAG knuderne i forbindelse med cache i SC algoritmerne. Det gælder uden undtagelser for alle testmængderne, at VE3 bruger mindre eller lige så meget plads som VE2. Dette resultat kan forklares med, at flytningen af observerbare variable, frem til de bliver relevante, sikrer, at VE algoritmen ikke arbejder med disse variable som irrelevante variable i potentialerne på de knuder, som de observerbare variable er blevet flyttet forbi. Ligeledes kan flytningen af de uobserverbare variable op gennem S-DAGen sikre, at de potentialer, som de uobserverbare variable indgår i, kan forblive faktoriserede i længere tid.

For Monitoring4 (tabel 8.7) er den relevante fortid for alle knuder i S-DAGen lig med den totale fortid, og da NFS-DAGen for Monitoring4 samtidigt kun indeholder én sti, medfører det en vægt på 0 for alle knuderne i NFS-DAGen. Dette viste vi i afsnit 6.2.3 på side 107. Det gælder generelt at SC algoritmens tidskompleksitet i grænsen uden cache er sammenfaldene med den for fuld cache for alle S-DAGe med én sti i S-DAGen og hele fortiden som relevant fortid for alle knuderne. Dette kan ses ved at observere, at tidskompleksiteten af SC algoritmen for disse to situationer (udtryk (7.1) og udtryk (7.2) på side 111) er identiske, når S-DAGen har én sti, og når den

Monitoring4	Plads		Tid	
Alg.	Indgange ($\cdot 1024$)	Eval. [KB]	Kald ($\cdot 10^6$)	Eval. [s]
Naive0	1519,48	12159,2	604,4	5562
Naive1	1519,51	12159,0	584,1	5177
Greedy0	0,000	2,977	604,4	5451
Greedy1	0,030	3,594	584,1	5390
VE2	5087,17	24354,91		1136
VE3	5087,17	24354,95		1131

Tabel 8.7: Denne tabel viser, hvordan SC og VE algoritmerne yder mht. tids- og pladsforbrug på Monitoring4 fra figur 2.8 på side 39 skaleret op til 4 inspektioner – når der er vilkårligt meget hukommelse til rådighed.

relevante fortid er lig med hele fortiden.

Hvis det ikke var for det eksponentielle led, der følger af SC algoritmens tidsforbrug ved beregning af sandsynligheder ($exp(cond^*)$-ledet), så møder SC algoritmerne altså VE algoritmens tidskompleksitet på denne generelle type problemer. For Greedy0 gælder dette samtidigt med, at pladsforbruget er konstant (her på $2,977$KB). Til sammenligning er VE algoritmens pladsforbrug eksponentielt i hele fortiden for knuderne i S-DAGen på denne type problemer.

Det pladsproblem, som Monitoring4 udgør for VE, er karakteristisk og går igen flere steder. Der er bl.a. et eksempel i [Jensen, 2001], hvor fiskekvoter skal bestemmes og i [Lauritzen and Nilsson, 2001], hvor grise skal være raske ved slagtning. Dette pladsproblem er altså ikke tilstede med SC algoritmen og den grådige strategi.

Tidsforbrug I tabel 8.5 kan man se, at tidsforbruget for VE algoritmerne generelt er lavere end for SC algoritmerne. Dette resultat følger forventningerne, da SC algoritmerne har en højere tidskompleksitet end VE algoritmerne, pga. det eksponentielle led for beregningen af sandsynligheder – jf. kapitel 7.

Hvis vi antager at SC algoritmen ikke skulle bruge tid til at beregne sandsynligheder, så ville SC og VE algoritmernes tidskompleksitet blive sammenlignelige, da faktoren $exp(cond^*)$ forsvinder fra udtrykket for SC algoritmens tidskompleksitet. Antagelsen er i sig selv urimelig, men med den ville Greedy0 algoritmens kørselstid svare til den tid, som Greedy0 algoritmen bruger på at lave konditionering i S-DAGen (eksklusiv tiden, der bruges på at lave konditionering i R-DAGen).

Bestemmes den tid, der blev brugt til konditionering i S-DAGen (eksklusiv tiden i R-DAGen) i forhold til tiden, der blev brugt af VE3 algoritmen for alle de testede UIDer, bliver gennemsnittet af disse forhold $7,18$ med en

standardafvigelse på 15 og min. og max. værdier på $0,0683$ og 112.

Dette forhold angiver en nedre grænse for, hvordan vores SC algoritme ville yde, hvis R-DAG tilgangsvinklen blev udskiftet med en anden tilgangsvinkel med en bedre tidskompleksitet. Det forholdsvist lave forhold (på i gennemsnit $7,18$) åbner et håb om, at SC algoritmens kørselstid kan blive i størrelsesorden af VE3 algoritmen, hvis den del af algoritmen, der beregner sandsynligheder bliver udskiftet. I gennemsnit vil SC algoritmen dog mindst være $7,18$ gange langsommere. Dette skyldes formentligt, at SC algoritmen arbejder med mange rekursive kald mens VE algoritmen arbejder på arrays i hukommelsen, og det (i Javas virtuelle maskine) formentlig tager længere tid at udføre disse rekursive kald end det tager at arbejde med arrays i hukommelsen. (Sammenligningen er dog ikke helt rimelig, da andre faktorer også spiller ind på dette forhold.)

Bemærk yderligere at der har været et UID, hvor forholdet mellem tiden brugt på konditionering i S-DAGen og VE3 algoritmens kørselstid har været $0,0683$. Dvs. at SC algoritmen her vil kunne blive op til $0,0683^{-1} \approx 15$ gange hurtigere end VE3 algoritmen. Dette skyldes formentligt, at VE3 algoritmen i dette UID har arbejdet med mange irrelevante variable.

8.3.6 Basisallokering

I dette afsnit vil vi undersøge, hvad SC algoritmernes basis-pladsforbrug er. Dvs. det minimums-pladsforbrug, der skal bruges til at repræsentere det originale UID, S-DAGen og R-DAGen.

Heap-forbruget til at repræsentere de testede UIDer har i gennemsnit været 33KB med standardafvigelse på $8,4$KB og min. og max værdier på 25KB og 49KB. Vi har også bestemt det gennemsnitlige pladsforbrug per UID knude for alle de UIDer som vores testmængder er sammensat af. Blandt alle UIDerne er gennemsnitspladsforbruget $1,25$KB per knude med en standardafvigelse på $0,025$KB og min. og max. værdier på $1,08$KB og $1,32$KB per knude og en variationskoefficient på $2,0\%$. Variationene skyldes formentlig at størrelsen af de aktuelle sandsynlighedspotentialer og nyttetabeller kan variere fra UID til UID.

I vores implementering gemmer vi ikke en separat grafstruktur til at repræsentere R-DAGen, men i stedet for tilføjer vi ekstra information til S-DAGen, der beskriver, hvordan R-DAGen afviger fra S-DAGen. På denne måde undgår vi, at skulle gemme to (i worst case) eksponentielt store grafstrukturer i hukommelsen.

For at optimere SC algoritmens kørselstid udregner vi konteksten for S-DAG-knuderne én gang for alle og gemmer en liste med variablene i kontek-

sten for en knude, $\mathcal{K}$, lokalt på $\mathcal{K}$. Disse optimeringer med flere er nærmere beskrevet i bilag D.

De testede UIDers S-DAGe (med R-DAG information og kontekstangivelse mv.) har optaget mellem 19KB og 155KB med et gennemsnit på 55KB og en standardafvigelse på 30KB. Variationskoefficienten er 54%, hvorfor størrelsen af S-DAGene varierer mere end størrelsen af UIDerne, hvilket formentlig skyldes, at antallet af knuder i S-DAGene er afhængigt af den temporale ordning over UIDets beslutninger.

For alle UIDer i alle testmængderne er den mindste plads per S-DAG-knude 1,70KB og den største plads per knude er 4,20 KB med et gennemsnit på 2,58KB, en standardafvigelse på 0,50KB og en variationskoefficient på 20%. Variationen kan skyldes variationer i størrelsen af R-DAG information der skal tilføjes, antallet af kanter i S-DAGen, størrelsen af konteksterne for S-DAG-knuderne, antallet af UID-variable i S-DAG-knuderne mv.

Da variationskoefficienterne over disse gennemsnit for pladsforbruget per knude er mindre end variationskoefficienterne for kørselstiden per kald (se afsnit 8.3.3), vil denne måde at estimere basis-allokeringen ud fra et samlet middel over alle de testede UIDer formentligt kunne bestemme basis-allokeringen mere sikkert, end vi kunne estimere kørselstiden ud fra den gennemsnitlige tid per kald over alle de testede UIDer.

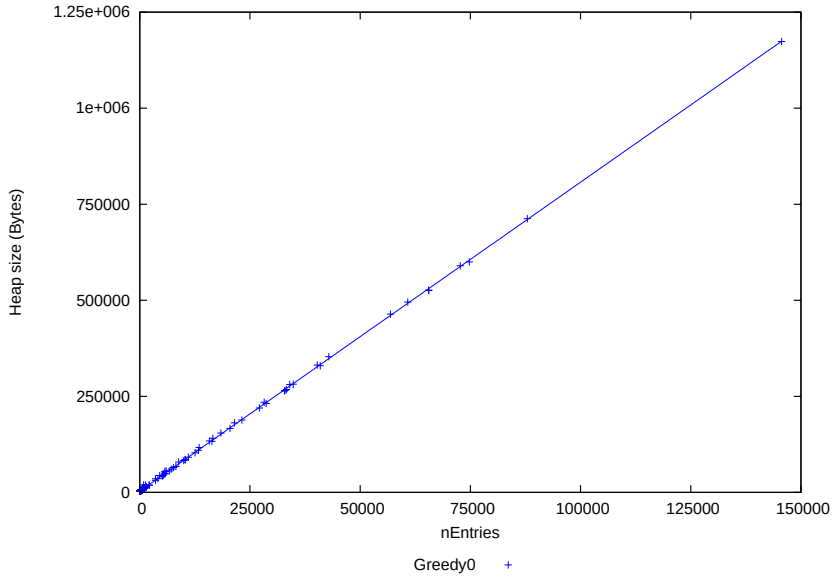
For at bestemme basisallokeringen skal vi til pladsen per UID og S-DAG knude lægge pladsen til selve koden af SC algoritmen. Da vores implementering er en udvidelse af Elvira rammeværket, kan vi ikke præcist fastslå dette forbrug, da det vil kræve, at vi laver vores implementering om til en selvstændig applikation, der kun består af vores kode og de anvendte Elvira-metoder. Et overslag over pladsforbruget kan fås, ved at tage Java-bytekodens størrelse for de klasser, som vores implementering udvider Elvira med. Størrelsen af disse klasser er 109 KB, og de indeholder koden, der genererer NFS-DAGen, fastlægger cacheingstrategierne og selve SC algoritmerne.

Herudover kan det nævnes, at selve kildekoden for SC algoritmen uden cache (som den beskrives i afsnit 5.2.1 på side 68) fylder omkring 300 linier kode, og er tilstrækkelig til at evaluere en allerede opstillet S-DAG med S-DAG konditionering uden cache. Desuden kan den implementeres på et enkelt døgn.

8.3.7 Estimering af pladsforbrug per cacheindgang

For at kunne estimere det samlede pladsforbrug skal vi kende det praktiske pladsforbrug per cache-indgang. I figur 8.3 har vi afbildet SC algoritmens heap-forbrug under evalueringen af alle de testede UIDers S-DAGe som funk-

tion af antallet af indgange, der bliver tildelt cache til under evalueringen. Afbildningen er for Greedy0.



Figur 8.3: Tilvæksten i heapen under evaluering af alle de testede UIDers S-DAGe som funktion af antallet af indgange i cachen for Greedy0.

På figuren har vi også tilføjet en tendenslinie med forskriften

$$Heap - size(nEntries) = 8,042\text{bytes/nEntries} \cdot nEntries + 3259. \quad (8.1)$$

På figuren kan vi se, at punkterne passer godt til tendenslinien og da determinationskoefficienten er $0,99985$, vurderer vi, at den givne forskrift giver et godt estimat for Greedy0s heap-forbrug under et givet antal cache-indgange.

Vores cache er repræsenteret som arrays af doubler, hvor hver indgang fylder 8 bytes. Det praktiske antal bytes per indgang vil på figuren blive afbildet som tendensliniens hældningskoefficient. Hældningskoefficienten er $8,042$ bytes/indgang, hvorfor det praktiske pladsforbrug per indgang svarer til vores forventninger. At forbruget er lidt større end 8 bytes per indgang kan forklares ved, at der sammen med de forskellige double-arrays også bliver afsat plads til pointere (af 8 bytes) fra S-DAG-knuderne til de forskellige arrays.

Ud over pladsen per indgang er der også et ekstra-forbrug på omkring 3 KB (som vi også så det ved Greedy0 i tabel 8.7 for Monitoring4). Denne plads er formentligt brugt til optællingsvariable, repræsentation af registrede

konfigurationer, variable til repræsentation af registrede instantieringer over konteksten mv. (se fx algoritme 3 på side 83). Desuden kan det tænkes, at heapen også indholder døde objekter, som Javas garbage collector ikke har fået fjernet. Dette kan også forklare ekstra-forbruget og at hældningskoefficienten er lidt større end 8 bytes/indgang.

Samlet vurderer vi, at pladsforbruget for et givet antal cacheindgange kan estimeres forholdsvist præcist.

8.4 Opsummering af de empiriske resultater

Samlet set har vores empiriske resultater vist, at vi kan opnå en jævn afvejning mellem plads og tid ved anvendelse af SC algoritmerne. Af alle SC algoritmerne gav de grådige strategier den bedste afvejning mellem tid og plads. Ligeledes har resultaterne vist, at det kan betale sig, at håndtere cache på R-DAGen, hvis der er tilpas mange uobserverbare variable. Dog er der behov for mere forskning på dette område, hvis den præcise grænse, for hvornår R-DAG cache kan betale sig, skal bestemmes. Resultaterne har også vist, at pladskravene for at indføre cache på R-DAGen er lave ift. pladskravene for cache på S-DAGen.

Derudover har resultaterne vist, at tiden per S-DAG og R-DAG-kald er forskellig og varierende fra S-DAG til S-DAG, hvorfor et præcist estimat for kørselstiden ikke umiddelbart kan opnås ud fra det opstillede udtryk for det samlede antal S-DAG og R-DAG-kald. I vores diskussion af dette resultat har vi anskueliggjort, hvilke parametre der forventes at have indflydelse på tiden per S-DAG og R-DAG-kald.

Tiden per kald varierer fra S-DAG til S-DAG og fra cachingstrategi til cachingstrategi, men bestemmes forholdet for den enkelte S-DAG og cachingstrategi tyder foreløbige resultater på, at kørselstiden kan estimeres rimeligt præcist. Yderligere vil det også være muligt at estimere algoritmernes basisallokering.

I vores sammenligning af SC og VE algoritmerne har resultaterne vist at SC algoritmerne generelt bruger mindre plads end VE algoritmerne i grænsen, hvor der tildeles fuld cache til alle S-DAG og R-DAG knuderne. Til gængæld er VE algoritmerne generelt hurtigere end SC algoritmerne, hvilket bl.a. kan forklares med at VE algoritmerne har en bedre tidskompleksitet.

Kapitel 9

Perspektivering og videre arbejde

I dette kapitel vil vi først kort se på, hvor SC algoritmen vil kunne anvendes, og efterfølgnede vil vi diskutere, hvordan SC algoritmen kan forbedres.

9.1 Perspektivering

I denne rapport har vi indtil nu beskrevet og testet vores algoritme til løsning af UIDer på begrænset plads. Den opstillede algoritme er blevet en anyspace-algoritme, der kan løse UIDer på lineær plads ud over den plads, der skal bruges til at repræsentere UIDet og S-DAGen. Algoritmen kan også tilbyde en trinløs afvejning af tid og plads. Dette muliggør, at vores algoritme kan anvendes i indlejrede systemer (fx i autonome agenter), hvor der kan være små pladsressourcer. Udtryk (6.5) på side 101 estimerer kørselstiden af SC algoritmen under en given hukommelsestildeling, hvilket muliggør, at man kan estimere hvor meget hukommelse, der skal tildeles til SC algoritmen for at opnå en given svartid for løsningsalgoritmen. Ligeledes kan man med estimatet afgøre, om man har tid til at beregne de optimale politikker, eller om man skal vælge en approksimativ løsningsalgoritme i stedet for.

SC algoritmen kan ligeledes anvendes på systemer, der bruges til flere formål samtidig. Hvis SC algoritmen skal afvikles på et system med flere aktive processer, vil man kunne drage fordel af, at SC algoritmen er fleksibel ift. dens pladsbehov. Således vil SC algoritmen kunne benytte den CPU-tid og den hukommelse, der ikke benyttes af andre processer. Med SC algoritmens

caching-princip er det således muligt, at implementere SC algoritmen, så den benytter den tilgængelige hukommelse til cache, og hvis andre processer senere ønsker at anvende mere hukommelse, kan det tages fra SC algoritmen, uden at SC algoritmen skal starte forfra på dens beregninger – hvis der deallokeres hukommelse fra SC algoritmen, så vil det blot medføre, at der vil gå længere tid, inden den terminerer. Denne fleksible hukommelsesegenskab gør, at SC algoritmen kan anvendes som baggrundsproces. Således vil store UIDer (fx for diagnose- eller fejlfindingsproblemer) kunne løses uden at en separat maskine skal dedikeres til formålet.

9.2 Videre arbejde

Under designet af vores anyspacealgoritmer har vi gjort nogle overvejelser om, hvordan man kan forbedre algoritmerne. Disse ideer er ikke blevet behandlet dybere end beskrevet her, men som fremtidigt arbejde indenfor løsning af UIDer kunne der fx tages udgangspunkt i disse ideer.

1. Man kunne udvikle en ny algoritme til beregning af sandsynligheder vha. dynamisk programmering. På denne måde vil SC algoritmen formentligt kunne matche VE algoritmens tidskompleksitet. En tilgangsvinkel kunne være, at adaptere VE algoritmens måde at beregne sandsynligheder. Denne tilgangsvinkel kunne laves til en anyspacealgoritme, ved at virke præcist som VE algoritmens sandsynlighedsberegningsdel (beregningen af Φ potentialemængderne) i grænsen, hvor der er plads nok. Hvis der ikke er plads til et bestemt potentiale på en S-DAG-knude, så erstattes dette med scripts, der angiver, hvordan potentialet kan beregnes ud fra potentialerne på S-DAG-knudens børn. I grænsen, hvor der kun er ganske lidt plads til rådighed, vil alle sandsynligheder bestemmes vha. scripts, der arbejder direkte på de originale sandsynlighedspotentialer. Denne tilgangsvinkel vil i grænsen, hvor der er tilstrækkeligt meget plads formentligt møde VE algoritmens tidskompleksitet.

2. Man kunne udvikle en dynamisk cachingstrategi, som vi lagde op til i kapitel 6. På denne måde kan man formentligt forbedre SC algoritmens kørselstid på systemer, hvor der ikke er plads til fuld caching.

3. Som beskrevet i afsnit 2.2.2 skal den relevante fortid for alle beslutningsknuder bestemmes under opbygningen af 2NFS-DAGen. Den relevante fortid for en beslutning, D, er afhængig af ordningen af beslutningerne i fremtiden for D, som blev bestemt ved d-forbundethedsegenskaben,

der var beskrevet i definition 2.8 på side 31. I et worst case UID er der $O(nDec!)$ forskellige ordninger på beslutningerne i fremtiden for D, og dermed tilsvarende mange UIDer at lave d-separationsanalyser på. Denne analyse skal desuden laves for hver beslutningsknude i hele 2NFS-DAGen. Vores implementering benytter en algoritme til d-separationsanalyse fra [Lauritzen et al., 1990], da denne algoritme i forvejen var implementeret i Elvira rammeværket. Med denne algoritme har det ikke være praktisk muligt at generere en 2NFS-DAG for fx UIDet på figur 2.9 på side 40 skaleret op til 8 behandlinger. Der er derfor behov for yderligere forskning på dette område. Ca. 98% af den tid, som vores algoritmer har brugt på at generere 2NFS-DAGe, er blevet brugt på at udføre d-separationsanalyser. Man kan altså arbejde videre med en effektiv algoritme til at opbygge 2NFS-DAGe.

4. Hvis man vil forbedre R-DAG konditioneringen, kunne man udnytte, at UIDet kan blive delt op i flere mindre net, når det bliver konverteret til et bayesiansk net, som det fx var tilfældet på figur 5.9 på side 85. Hvis der opstår flere mindre net, når UIDet bliver konverteret til et bayesiansk net, så vil flere af de betingede sandsynligheder kunne bestemmes ud fra et enkelt af de mindre bayesianske net. Dette kan man i flere tilfælde udnytte til at minimere antallet af rekursive kald, der skal laves, for at bestemme sandsynlighederne. I worst case vil det dog i sig selv formentligt ikke kunne bringe SC algoritmens samlede tidskompleksitet ned på niveau med VE algoritmens tidskompleksitet.

5. Det vil være muligt, at beregne de ønskede sandsynligheder ud fra fx et d-træ i stedet for ud fra en R-DAG. For et givet bayesiansk net gælder det, at der kan findes flere forskellige d-træer, og de forskellige d-træer kan medføre forskellige kørselstider under forskellige hukommelsesopsætninger [Darwiche, 2001]. Det vil således være muligt at undersøge mulighederne for at optimere SC algoritmens kørselstid under forskellige hukommelsesopsætninger ved at sammenligne R-DAG konditionering med konditionering over forskellige d-træer.

Kapitel 10

Konklusion

I denne rapport har vi først behandlet sprog til at repræsentere beslutningsproblemer i form af BTer og IDer. Under denne gennemgang så vi at BTer var et meget intuitivt sprog, der dog havde et problem ved repræsentationen af større beslutningsproblemer. Her blev IDer introduceret som et sprog til kompakt repræsentation af symmetriske beslutningsproblemer. Selvom IDer var designet til at repræsentere symmetriske beslutningsproblemer, var deres anvendelse ikke begrænset til dette formål, da IDerne stadig (om end mindre intuitivt) også kunne modellere asymmetriske beslutningsproblemer. UIDer blev efterfølgende indført som et sprog til en mere direkte repræsentation af ordningsasymmetriske beslutningsproblemer, dvs. beslutningsproblemer, hvor ordningen af beslutningerne kun delvist eller slet ikke var fastlagt.

Den gængse løsningsmetode til løsning af UIDer havde en høj pladskompleksitet, hvorfor vi valgte at fokusere på at udvikle en løsningsmetode, der kunne opstille politikker for knuderne i en løsningsgraf (en S-DAG) på begrænset plads. Denne løsningsgraf repræsenterede de tilladelige ordninger over beslutningerne i UIDet.

Efter en analyse af eksisterende løsningsmetoder til udførelse af sandsynlighedsinferens på begrænset plads, udviklede vi en anyspace-algoritme, der udnyttede konditionering og cache til at opnå en trinløs afvejning mellem tid og plads. Den opstillede algoritme, SC algoritmen, kunne opstille politikker for S-DAGen, og dermed muliggøre, at en beslutningstager kan handle optimalt ud fra et UID, selvom der på hans system ikke var plads til at repræsentere den eksakte løsning af UIDerne.

Der blev opstillet kompleksitetsudtryk for SC algoritmens tids- og pladskompleksitet i grænserne, hvor algoritmens pladsforbrug var størst og mindst

muligt. SC algoritmen kunne i det ene ekstrem, hvor mængden af plads var begrænset, beregne politikker for en S-DAG med et pladsforbrug, der (ud over en vis basisallokering) voksede lineært med antallet af variable i UIDet og med et tidsforbrug, der bl.a. voksede eksponentielt med antallet af variable i UIDet og med antallet af tilladelige ordninger over UIDets beslutninger.

Denne tidskompleksitet kunne trinløst reduceres på bekostning af den anvendte plads. Når mængden af plads ikke var den begrænsende faktor, voksede SC algoritmens pladsforbrug eksponentielt med konteksten for knuderne i S-DAGen. Dette var bedre end pladskompleksiteten for den eksisterende løsningsalgoritmes, VE algoritmen, der voksede eksponentielt med bredden af de elimineringsrækkefølger, som S-DAGen dikterede. Dog var SC algoritmens tidskompleksitet højere end VE algoritmens tidskompleksitet, da SC algoritmens tidskompleksitet voksede med en ekstra faktor, der var eksponentiel i (maksimalt) antallet af uobserverbare variable i UIDet.

Sammen med SC algoritmen udviklede vi også en række cachingstrategier og et udtryk til at beregne det samlede antal kald udført med SC algoritmen ved løsning af en given S-DAG på en given mængde plads. Tiden per kald synes konstant for en given S-DAG og cachingstrategi, hvorfor kørselstiden formentlig kan estimeres med dette udtryk. Yderligere forskning er dog nødvendig for at afgøre, om dette estimat altid vil være præcist.

For at forbedre SC algoritmens pladsforbrug ved udnyttelse af cache beskrev vi en række ideer til, hvordan S-DAGen kunne optimeres, således at pladskravene til cachen blev reducerede. Disse optimeringer af S-DAGen havde desuden en positiv effekt på VE algoritmens tids- og pladskrav.

Vores empiriske aftestning bekræftede, at det med SC algoritmerne var muligt at opnå en jævn afvejning mellem plads og tid. Desuden var kørselstiden for SC algoritmerne tæt på den optimale kørselstid for SC algoritmerne, hvis den mængde plads, der var til rådighed, var tæt på den plads, der skulle bruges for fuld cache.

Alt i alt har vi opnået et design af en anyspacealgoritme, der kan tilbyde en trinløs afvejning mellem tid og plads ved løsning af UIDer.

Litteratur

[Ahlmann-Ohlsen and Pedersen, 2005] Ahlmann-Ohlsen, K. and Pedersen, O. (2005). Anytimealgoritmer til løsning af ubegrænsede influensdiagrammer. Technical report, Aalborg University, Department of Computer Science.

[Allen and Darwiche, 2003a] Allen, D. and Darwiche, A. (2003a). New advances in inference by recursive conditioning. In *Proceedings of the 18th International Joint Conference on Artificial Intelligence (IJCAI-03)*, pages 2–10, San Francisco. Morgan Kaufmann Publishers.

[Allen and Darwiche, 2003b] Allen, D. and Darwiche, A. (2003b). Optimal time–space tradeoff in probabilistic inference. In *Proceedings of the 18th International Joint Conference on Artificial Intelligence (IJCAI-03)*, pages 969–975.

[Allen et al., 2004] Allen, D., Darwiche, A., and Park, J. D. (2004). A greedy algorithm for time-space tradeoff in probabilistic inference. In *Proceedings of the Second European Workshop on Probabilistic Graphical Models*, pages 1–8.

[Bacchus et al., 2003] Bacchus, F., Dalmao, S., and Pitassi, T. (2003). Value elimination: Bayesian inference via backtracking search. *In Uncertainty in Artificial Intelligence (UAI-03).*

[Bielza and Shenoy, 1999] Bielza, C. and Shenoy, P. P. (1999). A comparison of graphical techniques for asymmetric decision problems. *Management Science*, 45(11):1552–1569.

[Cano et al., 2000a] Cano, A., Moral, S., and Salmerón, A. (2000a). Lazy evaluation in penniless propagation over join trees. Technical report, Department of Computer Science and Artificial Intelligence.

[Cano et al., 2000b] Cano, A., Moral, S., and Salmerón, A. (2000b). Penniless propagation in join trees. *Internationl Journal of Intelligent Systems*, 15(11):1027–1059.

[Charnes and Shenoy, 2004] Charnes, J. M. and Shenoy, P. P. (2004). Multistage monte carlo method for solving influence diagrams using local computation. *Management Science*, 50(3):405–418.

[Chib and Greenberg, 1995] Chib, S. and Greenberg, E. (1995). Understanding the metropolis-hastings algorithm. *The American Statistician*, 49(4):327–335.

[Cooper, 1987] Cooper, G. F. (1987). Probabilistic inference using belief networks is NP-hard. *Knowledge Systems Laboratory.*

[Cooper, 1988] Cooper, G. F. (1988). A method for using belief networks as influence diagrams. In Schachter, R. D., Levitt, T. S., Kanal, L. N., and Lemmer, J. F., editors, *Preceedings of the Fourth Workshop on Uncertainty in Artificial Intelligence*, pages 55–63, Minneapolis, MN. Elsevier Science Publishers.

[Danish Meat Association, 2001] Danish Meat Association (2001). Produktion af grise. www.danishmeat.dk/view.asp?ID=24. Dateret 24. september 2001.

[Darwiche, 2000a] Darwiche, A. (2000a). Any-space probabilistic inference. In *UAI '00: Proceedings of the 16th Conference on Uncertainty in Artificial Intelligence*, pages 133–142, San Francisco, CA, USA. Morgan Kaufmann Publishers Inc.

[Darwiche, 2000b] Darwiche, A. (2000b). Recursive conditioning. Technical report, University of California. 3rd October.

[Darwiche, 2001] Darwiche, A. (2001). Recursive conditioning. *Artificial Intelligence*, 126(1-2):5–41.

[Dechter, 1996] Dechter, R. (1996). Bucket elimination: A unifying framework for probabilistic inference. In Horvits, E. and Jensen, F., editors, *Proceedings of the Twelthth Conference on Uncertainty in Artificial Intelligence*, pages 211–219, Portland, Oregon.

[Dechter and Fattah, 2001] Dechter, R. and Fattah, Y. E. (2001). Topological parameters for time-space tradeoff. *Artificial Intelligence*, 125(1-2):93–118.

[Green and Murdoch, 1998] Green, P. and Murdoch, D. (1998). Exact sampling for bayesian inference: towards general purpose algorithms. *Bayesian Statistics*, 6.

[Gómez and Cano, 2003] Gómez, M. and Cano, A. (2003). Applying numerical trees to evaluate asymmetric decision problems. In Nielsen, T. D. and Zhang, N. L., editors, *Lectures Notes in Artificial Intelligence: Symbolic and Quantitative Approaches to Reasoning with Uncertainty*, volume 2711, pages 196–207. Springer-Verlag. ISBN: 3-540-40494-5.

[Howard and Matheson, 1984] Howard, R. A. and Matheson, J. E., editors (1984). *Influence diagrams*, volume 2, Menlo Park, CA. Strategic Decisions Group. pages 719–762.

[Huand and Darwiche, 1996] Huand, C. and Darwiche, A. (1996). Inference in belief networks: A procedural guide. *International Journal of Approximate Reasoning*.

[Ide, 2003] Ide, J. S. (2003). Bngenerator version 0.3 - a generator for random bayesian network. www.pmr.poli.usp.br/ltd/Software/BNGenerator/. Senest opdateret: 29 september 2004.

[Ide and Cozman, 2002] Ide, J. S. and Cozman, F. G. (2002). Random generation of bayesian networks. *Brazilian Symposium on Artificial Intelligence*.

[Jensen, 2001] Jensen, F. V. (2001). *Bayesian Networks and Decision Graphs*. Springer-Verlag New York, Inc., Secaucus, NJ, USA.

[Jensen et al., 2006a] Jensen, F. V., Nielsen, T. D., and Shenoy, P. (2006a). Sequential influence diagrams: A unified asymmetry framework. *International Journal of Approximate Reasoning*, 42(1–2):101–118.

[Jensen and Vomlelová, 2002] Jensen, F. V. and Vomlelová, M. (2002). Unconstrained influence diagrams. *Eighteenth Conference on Uncertainty in Artificial Intelligence*, pages 234–241. Edmonton Canada, Morgan Kaufmann.

[Jensen et al., 2006b] Jensen, F. V., Vomlelová, M., Pedersen, O., Ahlmann-Ohlsen, K., and Nielsen, T. D. (2006b). UIDs - A representation language

for decision scenarios with partial temporal ordering of decisions. *Elsevier Science*. Preprint.

[Jørgensen, 2006] Jørgensen, B. (2006). Senior forsker og dyrlæge. Personlig meddelse.

[Lauritzen et al., 1990] Lauritzen, S. L., Dawid, A. P., Larsen, B. N., and Leimer, H.-G. (1990). Independence properties of directed markov fields. *Networks*, 20(5):491–505.

[Lauritzen and Nilsson, 2001] Lauritzen, S. L. and Nilsson, D. (2001). Representing and solving decision problems with limited information. *Management Science*, 47(9):1235–1251.

[Madsen and Jensen, 1999a] Madsen, A. L. and Jensen, F. V. (1999a). Lazy evaluation of symmetric bayesian decision problems. In *Proceedings of the Fifteenth Conference on Uncertainty in Artificial Intelligence*, pages 382–390. Morgan Kaufmann Publishers.

[Madsen and Jensen, 1999b] Madsen, A. L. and Jensen, F. V. (1999b). Lazy propagation: A junction tree inference algorithm based on lazy evaluation. In *Artificial Intelligence*, volume 113, pages 203–245. Elsevier Science Publishers Ltd., North-Holland.

[Neal, 1993] Neal, R. M. (1993). Probabilistic inference using markov chain monte carlo methods. Technical report, Department of Computer Science, University of Toronto.

[Neumann and Morgenstein, 1953] Neumann, J. v. and Morgenstein, O. (1953). *Theory of Games and Economic Behavior*. John Wiley and Sons, New York.

[Nielsen and Jensen, 1999] Nielsen, T. D. and Jensen, F. V. (1999). Well-defined decision scenarios. In Laskey, K. B. and Prade, H., editors, *Proceedings of the Fifteenth Conference on Uncertainty in Artificial Intelligence*, pages 502–511. Morgan Kaufmann Publishers.

[Ortiz and Kaelbling, 2000] Ortiz, L. E. and Kaelbling, L. P. (2000). Sampling methods for action selection in influence diagrams. In *AAAI/IAAI*, pages 378–385. AAAI Press / The MIT Press.

[Pearl, 1986] Pearl, J. (1986). Fusion, propagation, and structuring in belief networks. *Artificial Intelligence*, 29(3):241–288.

[Raiffa and Schlaifer, 1961] Raiffa, H. and Schlaifer, R. (1961). *Applied Statistical Decision Theory.* MIT Press, Cambridge, MA.

[Roubtsov, 2002] Roubtsov, V. (2002). Java tip 130: Do you know your data size? http://www.javaworld.com/javaworld/javatips/jw-javatip130.html. Dateret 16. august 2002.

[Salmerón et al., 2000] Salmerón, A., Cano, A., and Moral, S. (2000). Importance sampling in bayesian networks using probability trees. *Computational Statistics & Data Analysis*, 34:387–413.

[Salmerón, 2005] Salmerón, A. s. (2005). Elvira project. http://www.ual.es/personal/asalmero/elvira/. CVS Checkout per 5. oktober 2005.

[Shachter, 1999] Shachter, R. (1999). Efficient value of information computation. In *Proceedings of the 15th Annual Conference on Uncertainty in Artificial Intelligence (UAI-99)*, pages 594–60, San Francisco, CA. Morgan Kaufmann.

[Shachter et al., 1994] Shachter, R., Andersen, S., and Szolovits, P. (1994). Global conditioning for probabilistic inference in belief networks. In *Proceedings Tenth Conference on Uncertaincy in AI*, pages 514–522, Seattle WA.

[Shachter, 1986] Shachter, R. D. (1986). Evaluating influence diagrams. *Operations Research*, 34(6):871–882.

[Shafer and Shenoy, 1990] Shafer, G. and Shenoy, P. P. (1990). Probability propagation. *Ann. Math. Artif. Intell.*, 2:327–351.

[Verma, 1987] Verma, T. S. (1987). Causal networks: Semantics and expressiveness. In Kanal, L. N., Levitt, T. S., and Lemmer, J. F., editors, *Proceedings of the Third Workshop on Uncertainty in Artificial Intelligence*, pages 352–359, New York. Elsevier Science Publishers.

[Vomlelova and Jensen, 2002] Vomlelova, M. and Jensen, F. V. (2002). An extension of lazy evaluation for influence diagrams avoiding redundant variables in the potentials. In *Proceedings of the First European Workshop on Probabilistic Graphical Models*, pages 186–193, Cuenca, Spain.

[Vomlelová, 2003] Vomlelová, M. (2003). Unconstrained influence diagrams - experiments and heuristics. In *The Sixth Workshop on Uncertainty Processing WUPES'2003*, Hejnice, Czech Republic.

[York, 1992] York, J. (1992). Use of the gibbs sampler in expert systems. In *Artificial Intelligence*, volume vol. 56, pages pp. 115–130. Elsevier Science Publishers Ltd.

[Zhang, 1998] Zhang, N. L. W. (1998). Probabilistic inference in influence diagrams. In Cooper, G. F. and Moral, S., editors, *Proceedings of the fourteenth Conference on Uncertainty in Artificial Intelligence*, pages 514–522, Wisconsin. Morgan Kaufmann Publishers.

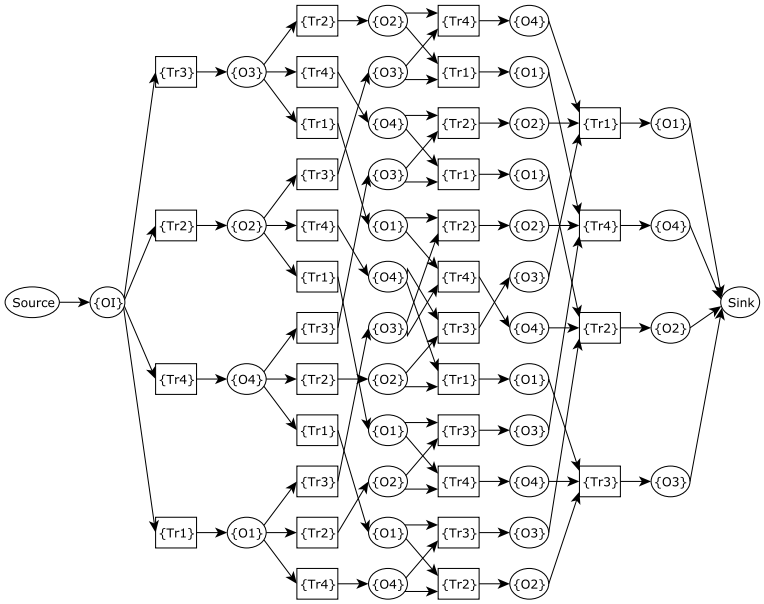
Del IV

Bilag

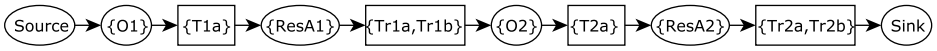
Bilag A

Yderligere diagrammer

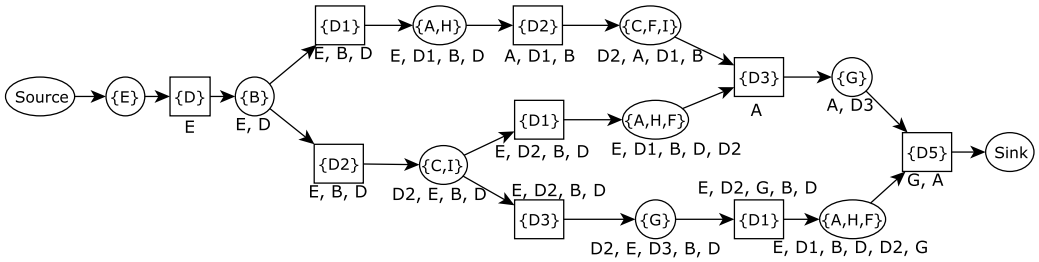
I dette bilag præsenterer vi en række grafer, der supplerer rapportens øvrige indhold.



Figur A.1: En NFS-DAG for UIDet i figur 2.9 på side 40. NFS-DAGen er en worst case NFS-DAG, da alle rækkefølger over beslutningerne er repræsenteret i NFS-DAGens stier.



Figur A.2: En NFS-DAG for UIDet i figur 2.8 på side 39. NFS-DAGen er en best case NFS-DAG, da den kun indeholder én sti.



Figur A.3: En NFS-DAG for UIDet på figur 5.1. Alle knuder er *konteksten* angivet – jf. udtryk (5.10) på side 74.

Bilag B

Benproblemer hos grise

I dette bilag beskriver vi en konkret problemstilling, hvor UIDet fra figur 2.8 på side 39 kunne anvendes. Dette eksempel er opstillet i samarbejde med [Jørgensen, 2006].

Griseproducenter leverer grise til slagterierne. Disse grise kaldes slagtesvin og er ved leveringstidspunktet typisk 5-6 måneder gamle [Danish Meat Association, 2001]. Inden da har de typisk været i en stald hos producenten. I flere stalde går grisene i stier, der er små båse med plads til ca. 15 grise. Gulvene er typisk hårde betongulve, som desværre ikke er egnede til grisenes ben og klove, hvorfor de kan risikere at få alvorlige benproblemer i løbet af den tid, de går i stalden. Sådanne benproblemer kan betyde at grisene får smerter og bevæger sig mindre. Det kan bl.a. medføre at de mindre hyppigt går over til foderautomaten for at spise, hvorfor de kan vokse langsommere, hvilket kan påvirke producentens økonomi.

Der findes metoder til at forebygge og i nogen grad afhjælpe de problemer, grisene kan få med deres ben; det er muligt at lægge særlige måtter ud, som giver grisene et bedre underlag at gå på. Ellers kan det hjælpe at dække gulvet med halm. Både løsningen med måtter og den med halm er forbundet med omkostninger; måtter er dyre i anskaffelse og bliver slidt, mens halm både har omkostninger forbundet med anskaffelse, såvel som ved rengøring af stierne.

I den periode grisene er hos producenten, foretager han periodiske inspektioner – fx én gang per dag – hvor han observerer, om det ser ud til, at grisene har det godt. I figur 2.8 er resultaterne af inspektionerne modeleret ved henholdsvis observation $O1$ og $O2$. Antallet af inspektioner kan dog være arbitrært mange, og UIDet i figur 2.8 kan således skaleres til flere.

For at forklare sammenhængen mellem UIDet og det konkrete eksempel med grisene, isoleres den del af UIDet, der har at gøre med den første inspektion; den del af grafen, der består af knuderne $H1, O1, ResA1, T1a, Tr1a$ og $Tr1b$.

Grisene i stien er i en samlet helbredsmæssig tilstand, $H1$, der påvirker hvad producenten oplever ved en inspektion, $O1$, ligesåvel som den helbredsmæssige tilstand påvirker resultatet af en given test, $ResA1$. I dette tilfælde består testen, $T1a$, af at producenten prøver at få grisene til at bevæge sig inde i stien. Ideen bag denne test er, at grise der ikke har benproblemer vil løbe glade og ivrige rundt i stien, mens grise med benproblemer vil halte eller forsøge at undgå bevægelse. Resultatet af en test vil bestå i en angivelse af hvor stor en del af grisene i stien, der ser ud til at have benproblemer.

Efter producenten har taget stilling til hvorvidt testen skal udføres eller ej, er det muligt at vælge at forbedre underlaget i stien. Der er to forbedringsmuligheder; en måtte eller halm. Disse er repræsenteret ved henholdsvis $Tr1a$ og $Tr1b$, hvorfor det også er muligt at kombinere de to behandlingsformer. Begge forbedringer er dog kun for en begrænset periode, således at producenten fx kan flytte måtten over til en anden sti på et senere tidspunkt.

Det er den ordningsmæssige asymmetri, der optræder i kraft af muligheden for to forskellige behandlinger, der gør det oplagt at modellere dette problem som et UID.

Bilag C

Opsætning og målemetoder

I dette bilag beskriver vi vores testmaskine samt de teknikker, som vi har brugt til at fastlægge de implementerede algoritmers tids- og pladsforbrug.

Alle test er afviklet på en PC med en Intel Pentium 4, 1800MHz CPU, 512 MB RAM og Windows XP Professional SP2 OS. Vores implementering er skrevet i Java 2 Standard Edition (J2SE), Development Kit 5.0 Update 6 og afviklet i J2SE Runtime Environment 5.0 Update 6.

I vores implementering starter vi først med at lave NFS-DAGen, optimere den og for den grådige cachingstrategi bestemme i hvilken rækkefølge S-DAG-knuderne skal have tildelt cache. Derefter starter vi evalueringen af NFS-DAGen.

I Java har man ikke direkte adgang til at få oplyst hvor meget CPU-tid en proces har brugt, hvorfor de tidsangivelser, der er angivet ved algoritmerne er *systemtid*. Vores tidsangivelser er således behæftet med en usikkerhed i kraft af, at der også har kørt andre processer i testmaskinens operativsystem samtidigt med vores Java proces. Vi har forsøgt at minimere denne usikkerhed ved at lukke så mange processer som muligt, inden vi startede vores målinger af systemtiden.

I Java har vi heller ikke hverken direkte adgang til at måle hvor meget plads et objekt optager i heapen, eller adgang til det største antal bytes en Java proces optager i heapen under sin afvikling. Derimod giver Java mulighed for, at man til et vilkårligt tidspunkt kan få oplyst den samlede størrelse af heapen, hvorfor vi kan måle, hvor meget plads vores datastrukturer fylder ved at måle hvor mange bytes, der samlet set er allokeret på heapen før og efter vi allokerer vores datastrukturer. På denne måde kan vi måle, hvor meget plads datastrukturen, der repræsenterer vores UID, S+R-DAGen og cachen,

fylder. Resultatet af at måle ændringen i heap-størrelsen kan dog afhænge af, hvor længe siden Javas garbage collector er blevet kørt, da mængden af objekter, der ikke længere refereres til, døde objekter, kan variere. For at minimere antallet af døde objekter, så kalder vi manuelt garbage collectoren 15 gange inden vi måler størrelsen af heapen. Garbage collectoren er dog ikke garanteret at fjerne alle døde objekter, (ikke engang efter at være kørt 15 gange,) hvorfor der er en vis usikkerhed i vores målinger af heap-størrelsen. Denne måde at bestemme størrelsen af objekter i Java er beskrevet i [Roubtsov, 2002].

Endeligt kan vi ikke måle det maksimale pladsforbrug på Javas stack, hvor alle de rekursive kald bliver administreret. Her ved vi dog, at Javas standard maksimum på 256KB ikke er blevet overskredet i vores test. Da det maksimale antal aktive kald, der er startet af SC algoritmen af gangen aldrig er mere end en konstant ganget antallet af knuder på en maksimalt orienteret sti mellem *Source* og *Sink* i vores S-DAG, og da pladsforbruget på stacken vokser lineært med antallet af aktive kald, så forventer vi ikke, at SC algoritmernes pladsforbrug på stacken kan udgøre et problem for afviklingen af SC algoritmerne.

Bilag D

Implementering

I dette bilag vil vi gennemgå, hvordan vi har implementeret vores algoritmer. Alle algoritmer er implementeret som udvidelser til Elvira rammeværket [Salmerón, 2005].

Inden vi starter opbygningen af NFS-DAGen, fjerner vi de barren knuder, der eventuelt måtte være i UIDet. Disse knuder kan fjernes, da de aldrig kan have indflydelse på løsningen til UIDet. Denne operation kan dog medføre afvigelser fra antallet af uobserverbare chanceknuder og beslutningsknuder, der er angivet i tabel 8.1 på side 116.

Den NFS-DAG vi opstiller overholder anden formalform, som beskrevet i definition 2.10. For SC samt VE3 algoritmen optimerer vi yderligere NFS-DAGen ved at flytte nyttepotentialer og uobserverbare chancevariable op og observerbare chancevariable ned gennem NFS-DAGen som beskrevet i afsnit 5.3.

NFS-DAGen er repræsenteret ved en `Vector` af S-DAG-knude-objekter. I hver S-DAG-knude objekt er der en `Vector` med referencer til de S-DAG knuder, der er forældre, og en `Vector` med referencer til de S-DAG knuder, der er børn. UID-knuderne, der knytter sig til en S-DAG knude er tilgængelige fra S-DAG-knuderne via et `HashSet`.

Potentialerne, der knytter sig til UID-knuderne, er repræsenteret som objekter af Elvira-typen `PotentialTable`, der repræsenterer domænet for potentialet ved en `Vector` af referencer til UID-knude objekter. Tabellen for potentialet er repræsenteret eksakt ved et array af doubler (den primitive type, der optager 8 bytes), der har én indgang per konfiguration over variablene i domænet for potentialet.

VE algoritmen Potentialemængerne (Φ og Ψ) der arbejdes med i VE algoritmen er hver især repræsenteret som en `Vector`, der indeholder referencer til `PotentialTable`-objekter. Hvis et potentiale, ϕ, optræder i potentialemængderne på flere på hinanden følgende S-DAG knuder, gemmes ϕ kun én gang, hvorefter der laves referencer til dette objekt, fra de efterfølgende potentialemængder, der indeholder ϕ. Dette er mere pladsbesparende, end at skulle gemme ϕ eksakt flere gange.

Når der laves variabeleliminering af flere variable fra samme S-DAG knude, så kan størrelsen af de resulterende potentialer afhænge af elimineringsrækkefølgen [Dechter, 1996]. Dette kan have betydning, når der skal elimineres flere variable ud af én S-DAG knude. Der laves imidlertid ingen undersøgelser, for at identificere den optimale (eller en suboptimal) elimineringsrækkefølge i vores implementering, hvorfor en vilkårlig elimineringsrækkefølge vælges.

SC algoritmen Cachen på S-DAG-knuder er implementeret som et array af `double`-primitiver. Disse arrays allokeres inden SC algoritmen køres på S-DAGen. Der allokeres ét array per S-DAG-knude, for hvilket størrelsen er givet ved udtryk (6.1) på side 98. Således kan vi opnå opslag i cachen for en S-DAG-knude i konstant tid.

Registreringen af instantieringer sker med Elvira-typen `Evidence`, der består af to `Vector`-objekter; ét med variabel-angivelser og ét med tilstandsangivelser for variablene.

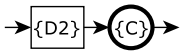
Cachen for en R-DAG-knude er ikke implementeret på samme måde som cachen på S-DAG-knuder, da det (som diskuteres i afsnit 6.3 på side 108) vil medføre et overhead i pladsforbruget, da vi kun bruger cachen på knuderne langs én sti i R-DAGen af gangen. Vores første løsning til dette var, at allokere cache til knuderne langs den aktuelle sti, umiddelbart inden der skulle laves R-DAG konditionering. Dette viste sig dog at medføre et stort overhead – der blev brugt mere tid på allokering af arrays til cache, end der blev brugt på at udføre den øvrige konditionering. Løsningen blev at allokere ét (globalt tilgængeligt) array én gang for alle, og lade dette array være stort nok til at indeholde alle cache-indgange for den sti, der kræver mest cache. Herefter bestemmer vi et *offset* for hver R-DAG-knude, der angiver, hvorfra (i det globale array) den enkelte R-DAG-knude kan bruge de efterfølgende pladser til at gemme sine resultater. Det globale array kan allokeres, og offsettet for R-DAG-knuderne kan findes, én gang for alle.

Optimeringer Hvis der ingen cache er på R-DAGen, undlader vi i vores implementering helt at lave opslag til cachen, allokering af array og bestem-

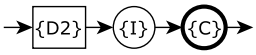
melse af offset, hvilket *kan* medføre en hurtigere afviklingstid på nogle problemer end algoritmen med fuld cache på R-DAGen – fx hvis der kun er få rekursive kald at spare ved at udnytte cache på R-DAGen.

I vores designkapitel beskriver vi R-DAGen som en selvstændig datastruktur, der eksisterer separat ift. S-DAGen. R-DAGen kan også implementeres på denne måde. Dette vil dog medføre, at der skal repræsenteres to forskellige datastrukturer i hukommelsen, hvilket kan være pladskrævende, da S-DAGen (og dermed også R-DAGen) kan vokse eksponentielt med antallet af beslutninger i UIDet. I stedet for, har vi valgt at repræsentere S-DAGen eksakt, og repræsentere R-DAGen i kraft af hvordan den adskiller sig fra S-DAGen.

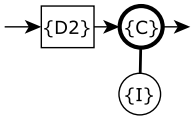
Hvis S-DAGen således har en sti $\mathcal{A} \prec \mathcal{B}$ og de uobserverbare chancevariable $\mathcal{U}$ skal indskydes umiddelbart før $\mathcal{B}$ i R-DAGen, så R-DAGen får stien $\mathcal{A} \prec \mathcal{U} \prec \mathcal{B}$, gemmer vi for blot information på $\mathcal{B}$ om, at variablene $\mathcal{U}$ i R-DAGen skal være umiddelbart før $\mathcal{B}$. Dette princip er illustreret på figur D.1. På denne måde kan vi næsten halvere den plads, der skal bruges til at repræsentere S-DAGen og R-DAGen.



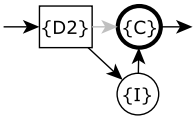
(a) Et udpluk af S-DAGen fra figur 5.2 på side 64.



(b) Et udpluk af R-DAGen fra figur 5.10 på side 88. Den adskiller sig fra S-DAG-udplukket i figur D.1(a) ved at $\{I\}$ er indskudt før $\{C\}$.



(c) I vores implementering repræsenterer vi kun én løsnings-graf; S-DAGen. På datastrukturen for S-DAGen har vi tilføjet information, der beskriver hvordan R-DAGen adskiller sig fra S-DAGen.



(d) Når der skal laves R-DAG konditionering, så benyttes denne information, som her skitseret.

Figur D.1: Illustrationer, der viser hvordan vi tilføjer de uobserverbare chanceknuder fra R-DAGen til S-DAGen så S-DAG-grafstrukturen også repræsenterer R-DAGen. På denne måde slipper vi for at repræsentere to selvstændige grafstrukturer, der i worst case er eksponentielt store.

Husk på, at vi ved R-DAG konditionering konditionerer fra *Source* og ned langs knuderne langs én sti ned til termineringsknuden. Det er lige meget hvilken sti, der konditioneres ad, så længe den går ned til termineringsknuden. Når vi skal vælge hvilket barn til en knude, $\mathcal{C}$, vi skal konditionere ned til, så er det eneste krav til dette barn, at det er termineringsknuden selv, eller en

forfader til denne. Det kan imidlertid være en dyr operation, at afgøre, om et givet barn er forfader til termineringsknuden. Et alternativ til at lave disse tjek er, inden man udfører R-DAG konditionering, at man laver en enkelt linket-liste af referencer mellem knuderne på stien fra *Source* ned til termineringsknuden, og når man så skal vælge et barn til $\mathcal{C}$, så tager man blot dét barn, som der er en reference til fra $\mathcal{C}$. Disse referencer kan laves, ved at starte med termineringsknuden, og så vælge en vilkårlig forældre, $\mathcal{P}$, til termineringsknuden, og gemme en reference fra $\mathcal{P}$ til termineringsknuden. Ved at fortsætte på samme måde rekursivt op gennem R-DAGen indtil *Source* er nået, vil man få genereret en enkelt linket-liste mellem *Source* og termineringsknuden. Således kan man slippe for at tjekke for, om et givet barn er forfader til termineringsknuden. Da vi implementerede denne optimering i vores kode fald tidsforbruget til R-DAG konditionering til en femtedel af, hvad den var før denne optimering.

Når vi skal tjekke om cachen indeholder en værdi for den aktuelle konfiguration over konteksten for en S-DAG-knude, skal vi i sagens natur kende konteksten for S-DAG-knuden. Denne kontekst kan bestemmes ud fra en række mængdeoperationer ud fra udtryk (5.10) på side 74. På denne måde kan konteksten bestemmes ved hvert cache-opslag. Men da konteksten for S-DAG-knuderne er den samme under hele SC algoritmens forløb, så kan man spare den tid der ville være brugt til disse mængdeoperationer ved at bestemme konteksten for de enkelte knuder én gang for alle og gemme resultatet for de enkelte S-DAG-knuder på knuderne selv. Dette har vi implementeret, hvilket også optimerede SC algoritmens tidsforbrug. Konteksten er i vores implementering tilgængelig via en reference til et `HashSet` af UID-knude-objekter.

På samme måde identificerer vi også én gang for alle, hvilke nyttepotentialer, der frigives på de enkelte S-DAG-knuder og gemmer referencer til disse nyttepotentialer på de enkelte S-DAG-knuder.

Vi gemmer også referencer fra UID chancevariablene til de sandsynlighedstabeller, der knytter sig til chancevariablene. Dette er en optimering ift. Elvira rammeværket, der normalt identificere den enkelte chancevariabels sandsynlighedstabel ved en gennemløbning af en `Vector` med alle sandsynlighedstabellerne, hver gang der skal bruges en sandsynlighedstabel.

Husk på, at konteksten for R-DAG-knuderne er afhængig af termineringsknuden. Derfor kan vi ikke på samme måde bestemme konteksten for R-DAG-knuderne én gang for alle. I stedet for bestemmer vi konteksten for knuderne på stien ned til termineringsknuden, samtidigt med, at vi bestemmer referencerne i den enkelt linket-liste fra *Source* til termineringsknuden.

Bilag E

Autogenerering af UIDer

I dette bilag beskriver vi den algoritme, vi har brugt til automatisk generering af UIDer. Algoritmen er vist i algoritme 5 på næste side. Den er inspireret af [Jensen et al., 2006b], [Vomlelová, 2003] og [Ide and Cozman, 2002]. Algoritmen blev første gang præsenteret i [Ahlmann-Ohlsen and Pedersen, 2005]. Vores implementering af UID generatoren er implementeret ovenpå `BNGenerator.java` fra [Ide, 2003].

Ideen bag denne algoritme er først at generere en orienteret acyklisk graf, og herefter tilføje nytteknuder til denne graf. Når grafstrukturen forelægger, så genereres tilsidst sandsynlighedspotentialer og nyttepotentialer tilfældigt.

Algoritmen har en række forskellige input parametre, der er beskrevet i de første linier af algoritmen. I linie 1 laves en simpel ordnet sti, som er vores start grafstruktur. Herefter itereres der nIt gange over grafstrukturen*. Under hver iteration vælges tilfældigt to knuder. Såfremt der findes en kant mellem disse knuder, så fjernes den og såfremt der ikke findes en kant mellem knuderne, så laves der en kant. Der tjekkes hele tiden for, om den resulterende graf forbliver forbundet og acyklisk, at antallet af forældre og børn til en knude ikke overstiger $nDeg$ og at der ikke laves en kant fra en uobserverbar chanceknude til en beslutningsknude (jf. definitionen af UIDer i afsnit 2.1 på side 23). Således opstår en vandring i rummet af orienterede acykliske grafer, og hvis der itereres nok gange, så vil de orienterede acykliske grafer ikke længere være præget af vores udgangsgraf. I stedet vil man have opnået en mere vilkårlig grafstruktur [Ide and Cozman, 2002].

Herefter genereres i linie 9 $nUtil$ forskellige nytteknuder. Da genererings-

*nIt kan fx udregnes ud fra de øvrige parametre. Vi har valgt $nIt = 6 \cdot (nObs + nHid + nDec)^2$ inspireret af [Ide, 2003].

Algoritme 5 Tilfældig generering af UIDer

UIDGENERATOR($nObs$, $nHid$, $nDec$, $nUtil$, $nDeg$)

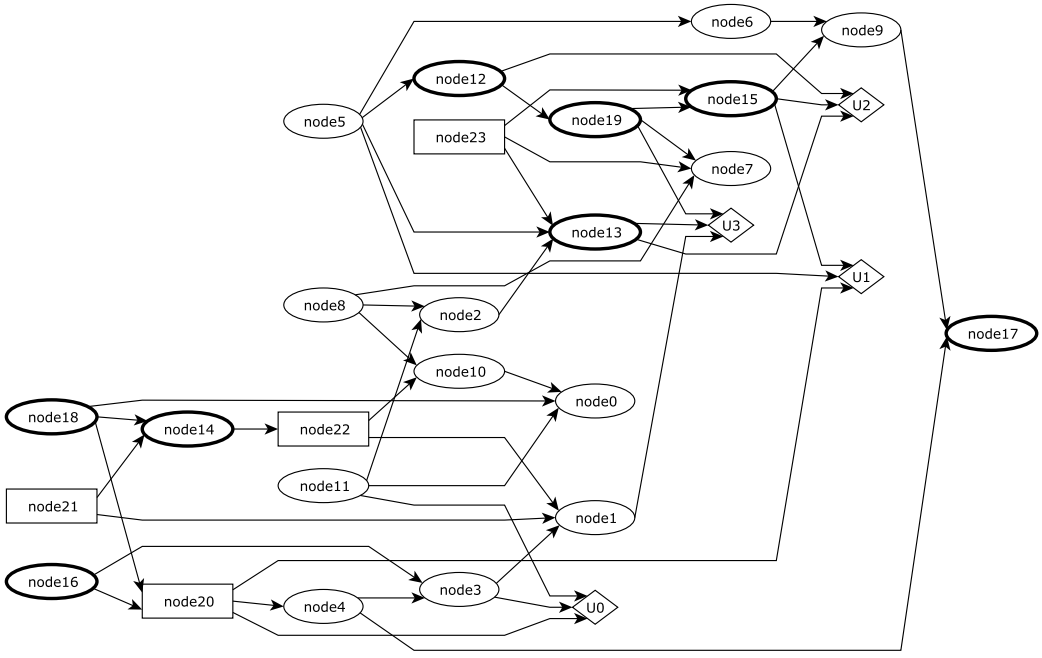
Input: $nObs$ - *Antallet af observerbare chanceknuder.*
Input: $nHid$ - *Antallet af uobserverbare chanceknuder.*
Input: $nDec$ - *Antallet af beslutningsknuder.*
Input: $nUtil$ - *Antallet af nytteknuder.*
Input: $nDeg$ - *Det maksimale antal forældre/børn til enhver knude.*
Output: *En tilfældig UID.*

1: *Lav en ordnet sti med $nObs$ observerbare chanceknuder, $nHid$ uobserverbare chanceknuder og $nDec$ beslutningsknuder. I denne sti må der ikke være nogle kanter fra en uobserverbar chanceknude til en beslutningsknude.*
2: **for** $i = 1$ **to** nIt **do**
3: *Vælg tilfældigt to forskellige knuder s og p.*
4: **if** *Kanten (s, p) findes* **then**
5: *Fjerne kanten, såfremt den resulterende graf forbliver forbundet.*
6: **else**
7: *Tilføj kanten, såfremt den resulterende graf forbliver acyklisk, antallet af forældre til p ikke overskrider $nDeg$, antallet af børn til s ikke overskrider $nDeg$ og kanten ikke kommer til at gå fra en uobserverbar chanceknude til en beslutningsknude.*
8: **for** $i = 1$ **to** $nUtil$ **do**
9: *Lav en nytteknude med $nDeg$ tilfældigt udvalgte forældre.*
10: **for all** *chanceknuder* **do**
11: *Lav en tilfældig sandsynlighedstabel.*
12: **for all** *nytteknuder* **do**
13: *Lav en tilfældig nyttetabel.*

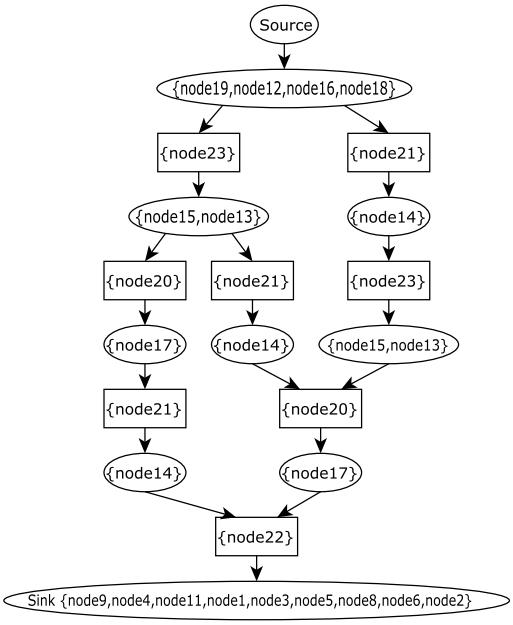
metoden hér ikke garanterer at alle beslutningsknuder indgår i en nytteknude (som det kræves i definition af UIDer), antager vi, at de beslutningsknuder der ikke indgår i en nytteknude har prisen 0 for alle deres beslutninger.

I linie 11 genereres en sandsynlighedstabel til alle chanceknuder, og i linie 13 genereres nyttetabeller til alle nytteknuderne. Alle variable er lavet binære. Vi har valgt at generere sandsynlighedspotentialerne ved at tildele tilfældige tal mellem 0 og 1 til alle tabellernes indgange, og efterfølgende normaliserer tabellerne, så sandsynlighederne summerer til 1 for alle de forskellige forældrekonfigurationer, der er repræsenteret i tabellerne. Nyttetabellerne genereres ved tilfældigt at udvælge værdier mellem -50 og $+150$ til alle indgange i tabellerne.

Figur E.1 er et eksempel på et UID, der er autogenereret med ovenstående algoritme. UIDet er tilfældigt udvalgt fra testmængden TS4h fra tabel 8.1 på side 116. På figuren kan man bl.a. se, at UIDet har 3 barrenknuder (node0, node7 og node10).



Figur E.1: Et autogenereret UID fra TS4h.



Figur E.2: (Optimeret) NFS-DAG for UIDet i figur E.1.

Bilag F

Flere testresultater

I dette bilag præsenterer vi flere empiriske resultater som supplement til dem i selve rapporten.

TS4h	Indgange	Heap size (KB)			
Alg.	Avg. ($\cdot 1024$)	Avg.	Std.	Min.	Max.
Naive0	1,146	11,77	6,927	4,320	28,31
Naive1	1,193	12,03	6,855	4,398	29,21
Greedy0	0,4523	6,303	2,467	2,477	11,40
Greedy1	0,4990	6,487	2,409	2,242	11,55
VE2	4,574	58,86	12,68	40,13	92,45
VE3	3,945	56,11	13,54	38,40	90,37

Tabel F.1: Denne tabel viser, hvordan SC og VE algoritmerne yder mht. pladsforbrug på TS4h, når der er vilkårligt meget hukommelse til rådighed.

TS4h	Kald	Tid (sek.)			
Alg.	Avg. ($\cdot 10^6$)	Avg.	Std.	Min.	Max.
Naive0	1,354	84,46	225,7	0,234	1005
Naive1	0,2534	26,76	79,34	0,218	360,2
Greedy0	1,354	82,15	223,3	0,250	995,7
Greedy1	0,2534	26,51	78,70	0,219	357,2
VE2		0,9242	0,8328	0,047	3,406
VE3		0,9203	0,8718	0,031	3,562

Tabel F.2: Denne tabel viser, hvordan SC og VE algoritmerne yder mht. tidsforbrug på TS4h, når der er vilkårligt meget hukommelse til rådighed.

TS3	Indgange	Heap size (KB)			
Alg.	Avg. (·1024)	Avg.	Std.	Min.	Max.
Naive0	25,55	207,3	152,8	30,68	483,09
Naive1	25,57	207,4	152,7	30,69	483,13
Greedy0	10,23	84,823	72,35	4,914	275,09
Greedy1	10,25	84,822	72,40	5,078	275,13
VE2	60,10	473,1	469,3	116,8	2144
VE3	47,33	440,9	436,3	100,8	2008

Tabel F.3: Sammenligning af pladsforbrug på TS3.

TS3	Kald	Tid (sek.)			
Alg.	Avg. ($\cdot 10^6$)	Avg.	Std.	Min.	Max.
Naive0	7,780	355,1	478,4	17,56	1251
Naive1	7,748	375,5	499,3	18,42	1294
Greedy0	7,780	353,9	476,5	17,30	1240
Greedy1	7,748	374,6	504,2	17,89	1302
VE2		34,06	25,00	3,625	95,06
VE3		35,31	24,50	4,094	82,31

Tabel F.4: Sammenligning af tidsforbrug på TS3.

TS7	Indgange	Heap size (KB)			
Alg.	Avg. (·1024)	Avg.	Std.	Min.	Max.
Naive0	67,58	549,0	470,992	28,49	1453,6
Naive1	67,64	548,9	470,990	28,74	1453,1
Greedy0	33,45	275,90	282,5	18,63	1146,1
Greedy1	33,50	275,92	282,4	18,88	1146,3
VE2	79,31	638,4	487,4	123,7	1738
VE3	60,34	539,6	342,7	125,1	1385

Tabel F.5: Sammenligning af pladsforbrug på TS7.

TS7	Kald	Tid (sek.)			
Alg.	Avg. ($\cdot 10^6$)	Avg.	Std.	Min.	Max.
Naive0	17,95	469,3	437,7	22,69	1470
Naive1	15,09	439,1	397,6	25,41	1273
Greedy0	17,95	465,9	432,8	22,80	1438
Greedy1	15,09	442,4	404,8	25,88	1338
VE2		24,24	23,80	1,172	85,31
VE3		23,63	27,76	1,157	107,0

Tabel F.6: Sammenligning af tidsforbrug på TS7.

TS16	Indgange	Heap size (KB)			
Alg.	Avg. (·1024)	Avg.	Std.	Min.	Max.
Naive0	45,721	367,43	540,79	10,16	2158,1
Naive1	45,729	367,52	540,81	10,24	2158,2
Greedy0	21,225	171,52	188,41	5,414	585,66
Greedy1	21,234	171,55	188,38	5,617	585,74
VE2	46,577	235,8	503,13	23,32	2205,33
VE3	33,311	230,4	499,03	23,29	2205,30

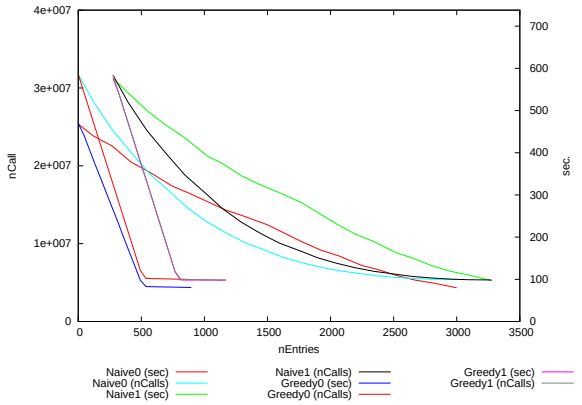
Tabel F.7: Sammenligning af pladsforbrug på TS16.

TS16	Kald	Tid (sek.)			
Alg.	Avg. ($\cdot 10^6$)	Avg.	Std.	Min.	Max.
Naive0	1,599	38,47	51,40	0,860	210,7
Naive1	1,564	41,70	56,98	0,969	238,0
Greedy0	1,599	38,60	53,03	0,844	221,7
Greedy1	1,564	40,57	55,90	0,953	235,5
VE2		7,601	15,70	0,047	63,30
VE3		7,451	15,64	0,047	63,44

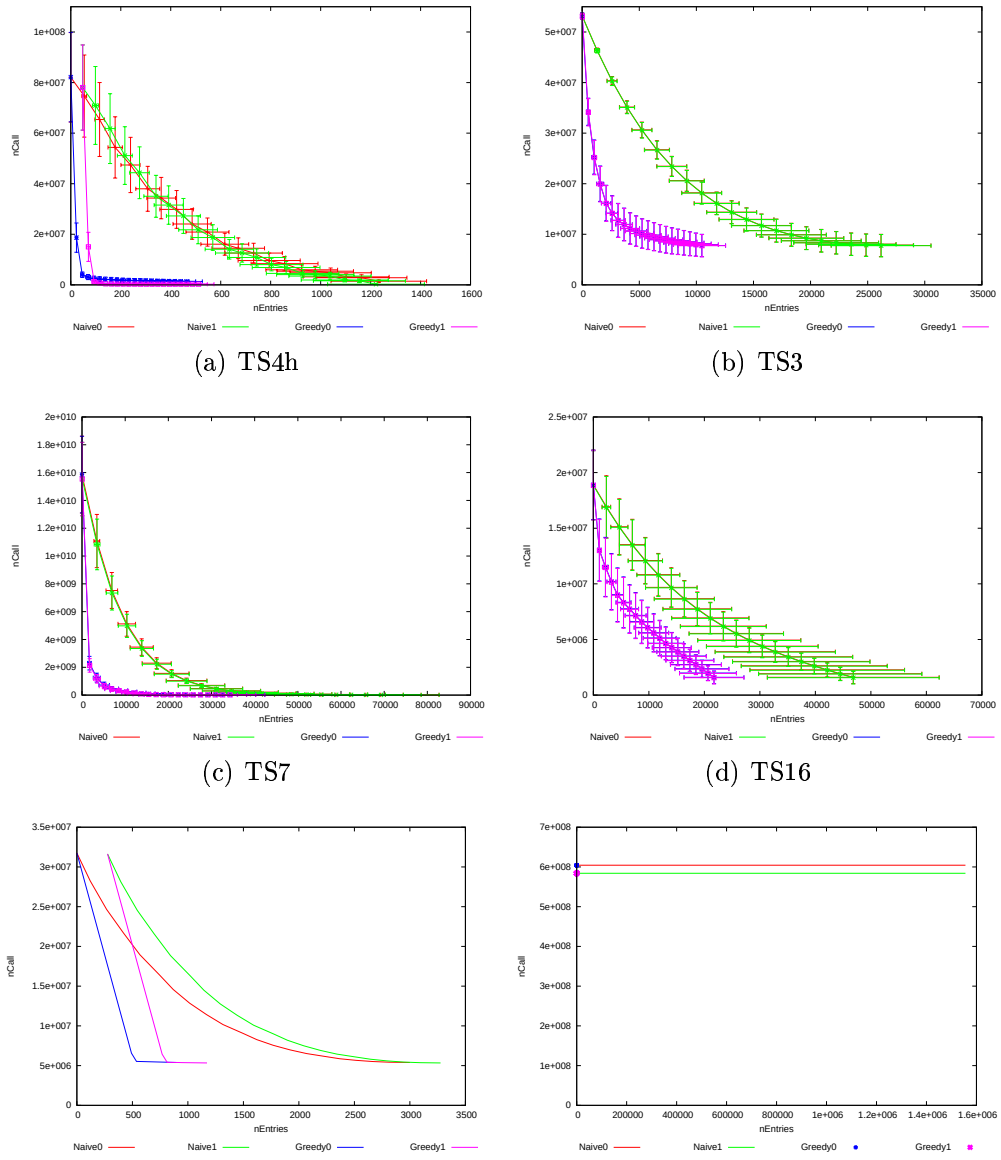
Tabel F.8: Sammenligning af tidsforbrug på TS16.

Tests4	Plads		Tid	
Alg.	Indgange (·1024)	Eval. [KB]	Kald ($\cdot 10^6$)	Eval. [s]
Naive0	2,930	35,85	5,398	81,90
Naive1	3,199	37,21	5,336	98,69
Greedy0	0,875	19,26	5,398	80,66
Greedy1	1,145	20,46	5,336	98,45
VE2	9,138	154,9		1,188
VE3	6,442	151,4		2,844

Tabel F.9: Sammenligning af tids- og pladsforbrug på Tests4.



Figur F.1: Teoretisk og praktisk afvejning af tid og plads for Tests4. Kurverne for "Greedy1 (sec)" og "Greedy1 (nCalls)" er sammenfaldene. Skaleringsforholdet for tiden per kald er her sat til $1,84 \cdot 10^{-5}$ sekunder per kald.



(a) TS4h (b) TS3

(c) TS7 (d) TS16

(e) Tests4 (f) Monitoring4

Figur F.2: $(nEntries, nCalls)$-grafer. Graferne illustrerer antallet af rekursive kald som funktion af antallet af anvendte cacheindgange. Sammenhængen er vist for vores fire cachingstrategier og er beregnet teoretisk ud fra udtryk (6.5) på side 101.

Summary

This thesis sets out arguing the need of an algorithm for solving unconstrained influence diagrams using limited space. It does so by presenting one of the existing algorithms, variable elimination, that represents the solution for an unconstrained influence diagram using a solution graph called an S-DAG. The algorithm solves the models exactly by eliminating variables in a strong elimination order, but suffers from time and space requirements that grow exponentially in two times the size of the unconstrained influence diagram.

With these computational challenges in mind we investigate different approaches towards overcoming these challenges, with emphasis on solving the possible memory problem that the space complexity constitutes. During this analysis the pros and cons of the different approaches is linked with the possibility of employing them for solving unconstrained influence diagrams efficiently using only limited space. The analysis yields conditioning to be the most interesting technique for us as it is capable of providing exact solutions in limited space.

Therefore a new algorithm called S-DAG Conditioning is introduced. This newly introduced algorithm utilizes conditioning, and recursively instantiates variables in an S-DAG to certain states in order to reduce the problem at hand, and it is as such a divide and conquer algorithm. By doing so, the algorithm may solve unconstrained influence diagrams in space that is linear in the number of nodes in the unconstrained influence diagrams. This approach, however, gives rise to a number of redundant recursive calls within the algorithm, that in turn prompts an increase in the consumption of time. Thus, reducing the use of space at the expense of time.

Allowing the algorithm to increase its space consumption and create a cache, the redundant calls may be eliminated by saving intermediate results for later reuse. For this reason the thesis carries on to investigate how to utilize additional space efficiently in the algorithm using a cache. As a result four different strategies for caching are introduced; two naive and two greedy. With the introduction of cache, the presented algorithm obtains a smooth

trade-off between space and time and it constitutes, as such, an anyspace algorithm. Furthermore, a number of optimizations were applied to the S-DAG structure in order to reduce caching requirements.

Caching all results yields different results for the complexities for space and time; the space complexity of the algorithm grows exponentially, whereas the time complexity of the algorithm compared to that of the variable elimination algorithm, grows with an additional exponential factor.

In addition to the results of complexity, a formula for calculating the number of calls, performed while executing the algorithm, was introduced. Though, in spite of being accurate to the degree of ± 1 calls, using it to estimate the execution time of the algorithm, empirical test results yielded a considerable coefficient of variation. This was obtained while comparing the estimated time for all the unconstrained influence diagrams that were included in our tests. Using the formula to estimate the running time of the algorithm for a single S-DAG, however, indicates that the estimation may be accurate to the degree of a factor of proportionality that varies from S-DAG to S-DAG as well as between the different caching strategies.

Performance of the proposed algorithm was tested empirically using the different caching strategies. The results that followed confirmed that it indeed was possible to obtain the aforementioned smooth time and space trade-off. Furthermore the results also showed that, using a greedy approach towards caching, it was possible to obtain an execution time *close to* the fastest time, as long as the space consumption was sufficiently close to that used to obtain the fastest execution time with the S-DAG Conditioning algorithm.

Additionally, the algorithm was tested against the existing variable elimination algorithm which yielded that S-DAG Conditioning, using the greedy caching strategies, consumed less space than variable elimination – even when it was possible to use as much space as needed to obtain the fastest execution times. One test even showed an reduction in space consumption from 24.354KB to 3KB.

Furthermore, S-DAG Conditioning proved to have slower execution times than variable elimination. This was to be expected, considering the time complexity of S-DAG Conditioning. As a spin-off, the tests yielded a noticeable improvement in variable elimination when it was executed using optimized S-DAGs.

Overall, this report documents the development of an anyspace algorithm for solving unconstrained influence diagrams. As such, it provides a viable solution for solving these diagrams in limited space, thus fulfilling the need for such an algorithm.