KU LEUVEN

FACULTY OF
ENGINEERING SCIENCE

Apply volgorde van een SDD compilatie

Gijs Mon M Vliegen - r0810400 - gijs.vliegen@student.kuleuven.be

Thesis voorgedragen tot het behalen
van de graad van Master of Science
in de ingenieurswetenschappen:
computerwetenschappen, hoofdoptie
Artificiële intelligentie

Promotor:
Prof. Dr. Luc De Raedt

Evaluator:

Begeleider:
Dr. Ir. Jessa Bekker
Dr. Ir. Vincent Derkinderen assessor

Academiejaar 2023 – 2024

Todo list

Inhoudsopgave

Bedanking

Bij het voltooien van deze thesis wil ik mijn oprechte dank uitspreken aan iedereen die mij gedurende dit proces heeft ondersteund en geïnspireerd.

In het bijzonder wil ik mijn promotor, Prof. Dr. Luc De Raedt, bedanken voor zijn onmisbare begeleiding, waardevolle inzichten en voortdurende steun.

Daarnaast ben ik bijzonder dankbaar voor de assistenten, Dr. Ir. Jessa Bekker en Dr. Ir. Vincent Derkinderen, voor hun constructieve feedback, aanmoediging en technische ondersteuning. Hun deskundige adviezen en bereidheid om te helpen hebben een grote bijdrage geleverd aan de totstandkoming van deze thesis.

Tenslotte wil ik mijn familie en vrienden bedanken voor hun onvoorwaardelijke steun en begrip tijdens deze intensieve periode. Hun aanmoediging heeft mij gemotiveerd om door te zetten en dit werk tot een goed einde te brengen.

Aan iedereen die op welke manier dan ook heeft bijgedragen aan dit werk, mijn oprechte dank.

Gijs Vliegen

Gebruik van generatieve AI

stuk toevoegen over het gebruik van chatGTP in deze thesis, bijvoorbeeld om termen te vertalen

vsc vermelding toevoegen: De infrastructuur en dienstverlening gebruikt in dit werk werd voorzien door het VSC (Vlaams Supercomputer Centrum), gefinancierd door het FWO en de Vlaamse overheid.

Lijst van figuren en tabellen

Lijst van figuren

Lijst van tabellen

Lijst van afkortingen en symbolen

Afkortingen

d-DNNF	Deterministic Decomposable Negation Normal Form
OBDD	Ordered Binary Decision Diagram
SDD	Sentential Decision Diagram
vtree	Variabelenboom
rooted-DAG	Gewortelde Gerichte Acyclische Graaf
CNF	Conjunctieve Normaalvorm
DNF	Disjunctieve Normaalvorm SAT
Booleaans Geldigheidsprobleem	
ct	Totale compilatietijd
VP	Heuristiek: Vtree partitionering
KE	Heuristiek: Kleinste eerst
EL	Heuristiek: Elementgewijze bovengrens
DP	Heuristiek: Top-down variabelenvolgorde
BO-LR	Heuristiek: Bottom-up variabelenvolgorde LR
BO-RL	Heuristiek: Bottom-up variabelenvolgorde RL

Symbolen

γ_i	SDD in de context van bottom-up compilatie
$\alpha, \beta, \Omega, \Psi$	SDD elders
v	Vtreeknoop
v^l	Linkerkind van vtreeknoop v
v^r	Rechterkind van vtreeknoop v
v^p	Ouder van vtreeknoop v
p, r	Primes
s, t	Subs
$\neg$	Negatie
$\wedge$	Conjunctie
$\vee$	Disjunctie
$\triangle$	Symmetrisch verschil
$\top$	Waar
$\bot$	Onwaar

in de inleiding van elk stuk vertellen over wat elke subsectie gaat

Samenvatting

Tijdens de bottom-up compilatie van logische circuits naar behandelbare voorstellingen zoals SDD's, OBDD's en d-DNNF's worden veel tussenresultaten berekend. In deze thesis worden verschillende heuristieken voorgesteld om de tussenresultaten zo klein mogelijk te houden en de bottom-up compilatietijd van de SDD te versnellen.

Hoofdstuk 1

Inleiding

In allerlei vakgebieden van computerwetenschappen worden logische circuits gebruikt, bijvoorbeeld binnen Probabilistische Inferentie voor het tellen van mogelijkheden [Der23]. Een logisch circuit bestaat uit verschillende variabelen. Elke variabele kan waarheidswaarde *Waar* of *Onwaar* hebben, de waarheidswaarde van het volledige circuit is een functie van de waarden van zijn variabelen. Een logisch circuit wordt ook wel een kennisbasis genoemd.

Er kunnen verschillende soorten bewerkingen of queries worden uitgevoerd op een kennisbasis [DM11], zie tabel 1.1 voor enkele voorbeelden van simpele bewerkingen en queries. Door een combinatie van simpele bewerkingen kunnen geavanceerde toepassingen zoals de eerder genoemde probabilistische inferentie, foutendetectie en diagnose van systemen, stochastische beperking-optimalisatie-problemen et cetera gesimuleerd worden [Bry86, LBD$^+$17]. Kennisbasissen kunnen op verschillende manieren voorgesteld worden. De verschillende voorstellingen worden ook wel compilatietalen genoemd. Een bepaalde voorstelling kan efficiënt zijn voor bepaalde queries of bewerkingen (poly tijdscomplexiteit). Maar als gevolg zijn er nadelen zoals andere queries of bewerkingen die minder efficiënt uitgevoerd kunnen worden, meer geheugen gebruiken, ... Het bestuderen van dergelijke voorstellingen is het vakgebied van kenniscompilatie. Elke toepassing vereist andere queries of bewerkingen met als gevolg dat er heel wat voorstellingen zijn bedacht en bestudeerd, zie [DM11] voor

Bewerkingen	Queries
Conditionering	Consistentiecontrole
Vergeten	Geldigheidscontrole
Vergeten van singletons	Clausale-implicatie-controle
Conjunctie	Implicatiecontrole
Begrensde conjunctie	Equivalentiecontrole
Disjunctie	Zin-implicatie-controle
Begrensde disjunctie	Modeltelling
Negatie	Modelenumeratie

Tabel 1.1: Bewerkingen en queries voor een kennisbasis

een overzicht.

Er bestaan twee algemene termen om voorstellingen te labelen, bondigheid als maat voor de grootte van de representatie, en behandelbaarheid voor de set van bewerkingen/queries die in polytijd kunnen worden uitgevoerd. In het algemeen geldt: hoe meer behandelbaar een voorstelling is, hoe minder bondig [DM11]. Twee type voorstellingen die een hoge behandelbaarheid hebben, en dus voor veel toepassingen gebruikt kunnen worden, maar toch ook relatief bondig zijn, zijn de Deterministic Decomposable Negation Normal Form (d-DNNF) [Dar00] en de Ordered Binary Decision Diagram (OBDD) [Bry86]. De OBDD is een deelverzameling van de d-DNNF. De OBDD heeft dus alle eigenschappen van de D-DNNF en nog enkele andere nuttige eigenschappen, maar is als gevolg minder bondig. Recent is er een nieuwe veelbelovende representatie voorgesteld, de Sentential Decision Diagram (SDD). De SDD is een deelverzameling van de d-DNNF, maar een superverzameling van de OBDD, met interessante eigenschappen van beide voorstellingen. Er is al aangetoond dat de SDD bondiger is dan de OBDD [Bov16a] en de meer algemene BDD [Bov16b].

Om een kennisbasis om te zetten naar een bepaalde voorstelling, moet deze gecompileerd worden. Voor SDD's, OBDD's en d-DNNF's kan dit zowel bottom-up als top-down. Tijdens de bottom-up compilatie van een kennisbasis naar een bepaalde representatie zoals de SDD worden er veel tussenresultaten bekomen. De tussenresultaten hangen af van de volgorde waarin een kennisbasis bottom-up wordt gecompileerd. We willen een een lage compilatietijd. Dit kunnen we bereiken door een compilatie waarbij de tussenresultaten en daarmee ook het eindresultaat zo snel mogelijk berekend kunnen worden. Voorlopig is er nog niet een duidelijke manier om een goede volgorde te zoeken. Zowel [NW07] als [HD04] kaartten dit probleem aan. In deze thesis gaan we opzoek naar heuristieken om een volgorde te bepalen, met als doel de totale compilatietijd te minimaliseren.

Hoofdstuk 2

SDD als nieuwe voorstelling van een kennisbasis

Deze sectie geeft belangrijke theorie om de SDD te begrijpen. Deze theorie komt voornamelijk uit de originele paper waarin de SDD werd geïntroduceerd [Dar11].

Eerst en vooral een paar opmerkingen over notatie. In context van een bottom-up compilatie zal γ_i gebruikt worden om SDD's te noteren, in andere gevallen worden de letters α, β, Ω en Ψ gebruikt. Hoofdletters zoals X zijn voor variabelen; Vetgedrukte hoofdletters $\mathbf{X}$ zijn voor sets van variabelen; Kleine letters zoals x zijn voor een instantie van een variabele; $\top$ wordt gebruikt voor de waarheidswaarde $Waar$ en $\bot$ voor $Onwaar$.

2.1 Propisitionele logica

Sentential Decision Diagrams zijn een bepaalde voorstelling van een logische formule. We leggen kort uit wat een logische formule is. De basis van een logische formule is een propositie, een propositie kan waar zijn, aangeduid met $\top$, of onwaar, aangeduid met $\bot$. Enkele voorbeelden van proposities zijn "Het regent", "Jan is 10 jaar oud" en "De deur is gesloten". Een variabele P is een propositie die waar of onwaar kan zijn. Als we willen zeggen dat P waar is, duiden we dat aan met de literal p. Als we willen zeggen dat P onwaar is, doen we dat met de literal $\neg p$. Variabelen kunnen gecombineerd worden met logische operatoren. We hebben al een operator gebruikt in de vorige zin, namelijk de negatie ($\neg$). Naast de negatie zijn er nog twee belangrijke operatoren, de conjunctie ($\wedge$) en disjunctie ($\vee$). De conjunctie van twee variabelen is enkel waar als beide variabelen waar zijn, de disjunctie is waar als minstens één van de twee variabelen waar is. In tabel 2.1 geven we twee waarheidstabellen, deze bieden een overzicht van alle mogelijke waarheidswaarden van de variabelen samen met de resulterende waarheidswaarde na een bepaalde operatie.

Met behulp van variabelen en logische operatoren kunnen we logische formules construeren. Een formule over de variabelen $\mathbf{X}$ kan geschreven worden als $f(\mathbf{X})$. Een logische formule $f(\mathbf{X})$ kan op verschillende manieren voorgesteld worden; de wetten van De Morgan zijn daar een voorbeeld van, zie figuur 2.2.

P	$\neg P$
$\bot$	$\top$
$\top$	$\bot$

P	Q	$P \wedge Q$	$P \vee Q$
$\bot$	$\bot$	$\bot$	$\bot$
$\top$	$\bot$	$\bot$	$\top$
$\bot$	$\top$	$\bot$	$\top$
$\top$	$\top$	$\top$	$\top$

Tabel 2.1: Waarheidstabellen met operaties negatie, conjunctie en disjunctie

$\neg(P \wedge Q) = \neg P \vee \neg Q$	$P \wedge \top = P \mid P \wedge \bot = \bot$
$\neg(P \vee Q) = \neg P \wedge \neg Q$	$P \vee \top = \top \mid P \vee \bot = P$

Tabel 2.2: De wetten van De Morgan en operaties met elementaire waarden

Een veelgebruikt type formule is de Conjunctieve Normaalvorm (CNF). Een CNF bestaat uit de conjunctie van disjunctieclausules. Een disjunctieclausule is een logische formule die enkel uit de disjunctie van (genegateerde) variabelen bestaat, bijvoorbeeld $(P \vee \neg Q \vee R)$. Zijn tegenpool is de conjunctieclausule. Een formule in K-CNF is een CNF waarbij elke clausule uit exact K variabelen bestaat. Zo bestaat er voor elke mogelijke formule een voorstelling in 3-CNF.

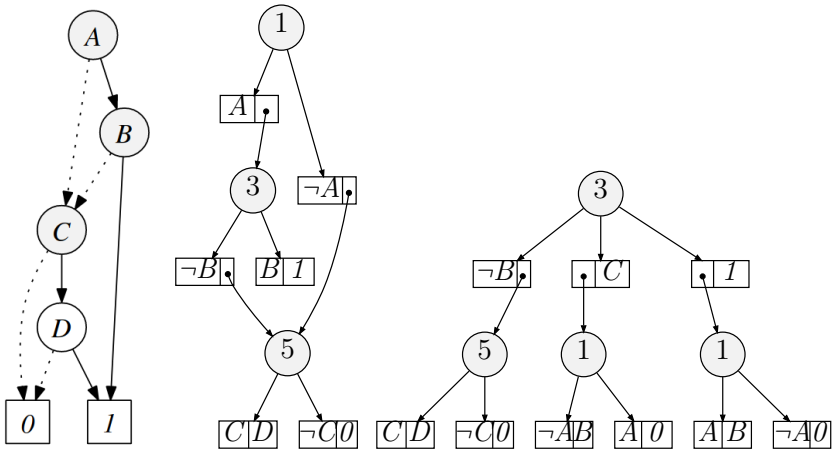
Als tegenpool van de Conjunctieve Normaalvorm bestaat de Disjunctieve Normaalvorm (DNF) . Een DNF is een logische formule die bestaat uit de disjuncties van conjunctieclausules. Een CNF kan getransformeerd worden naar een DNF door de negatie toe te passen en door toepassing van de De Morgan's wetten en distributiviteit, en vice versa.

Als we veronderstellen dat een variabele P waar is in de formule f, schrijven we dat als $f \mid P$, lees "formule f gegeven P".

2.2 SDD als deelverzameling van d-DNNF en superverzameling van OBDD

De meest belangrijke eigenschappen van de d-DNNF zijn splitsbaarheid en determinisme, deze eigenschappen maken dat de voorstelling een grote behandelbaarheid heeft [CD08]. Twee belangrijke eigenschappen van de OBDD die de d-DNNF niet heeft zijn canoniciteit en een polytijd apply-operatie. Canoniciteit houdt in dat een bepaalde kennisbasis altijd op dezelfde manier zal worden voorgesteld. Hierdoor kan equivalentiecontrole in polytijd, maar ook zoeken naar een optimale representatie (door de variabelenvolgorde te veranderen) is hierdoor mogelijk. Een polytijd apply-operatie houdt in dat twee OBDD's samengevoegd kunnen worden (de exclusieve disjunctie, disjunctie of conjunctie van twee OBDD's) in polytijd. Deze eigenschap maakt mogelijk dat een OBDD bottom-up kan gecompileerd worden. De OBDD is minder bondig dan de d-DNNF, maar door de canoniciteit en een polytijd apply-operatie is ze in de praktijk wel het meest populair.

De Sentential Decision Diagram (SDD) heeft als eigenschappen gestructureerde splitsbaarheid [PD08] en sterk determinisme [PD10]. Deze eigenschappen zijn strikter



FIGUUR 2.1: OBDD, rechtslineaire SDD en gebalanceerde SDD voor de formule $f = (A \wedge B) \vee (C \wedge D)$

dan splitsbaarheid en determinisme an sich. De SDD-taal is dus een strikte deelverzameling van de d-DNNF-taal. Verder is de SDD-taal een strikte superverzameling van de OBDD-taal. De SDD behoudt bepaalde belangrijke eigenschappen van de OBDD, namelijk canoniciteit en een polytijd apply-operatie. Het is al gebleken dat de SDD bondiger is dan de OBDD in termen van de padbreedte [Dar11] en voor bepaalde kennisbasissen is de SDD exponentieel meer compact [XCD12]. Voorbeelden van een OBDD en SDD zijn terug te vinden in figuur 2.1.

2.2.1 Sterk determinisme en partities

OBDD's zijn gebaseerd op Shannon-decomposities, hierbij wordt een formule f opgesplitst in twee elementen $(X, f \mid X), (\neg X, f \mid \neg X)$ op basis van de variabele X. De SDD breidt dit idee uit door niet enkel te splitsen op de waarde van één variabele X, maar door te splitsen op de waarde van meerdere variabelen, een zogenaamde zin of sentence.

Definitie 1 ((X, Y)-decompositie [Dar11]) *Gegeven een Booleaanse functie f over variabelen $\boldsymbol{X}$, $\boldsymbol{Y}$, met $\boldsymbol{X} \cap \boldsymbol{Y} = \emptyset$. Als $f = (p_1(\boldsymbol{X}) \wedge s_1(\boldsymbol{Y})) \vee \cdots \vee (p_n(\boldsymbol{X}) \wedge s_n(\boldsymbol{Y}))$, dan is $\{(p_1, s_1), \cdots, (p_n, s_n)\}$ een $(\boldsymbol{X}, \boldsymbol{Y})$-decompositie van f.*

Als ook nog geldt dat $p_i \wedge p_j = \bot$ voor elke $i \neq j$ dan wordt de decompositie sterk deterministisch op X genoemd. Elk paar (p_i, s_i) wordt een element genoemd, met p_i de prime en s_i de sub. De grootte van de decompositie is het aantal elementen erin.

Definitie 2 (X-partitie [Dar11]) *Gegeven een $(\boldsymbol{X}, \boldsymbol{Y})$-decompositie α van een functie f die sterk deterministisch is. α wordt een $\boldsymbol{X}$-partitie van f genoemd a.s.a. de primes een partitie vormen.*

Een partitie heeft als eigenschappen:

1. Elke prime is niet logisch onwaar.
2. Elke paar van twee verschillende primes kunnen niet door dezelfde instantie van variabelen waar zijn.
3. De disjunctie van alle primes is logisch waar.

Als geldt dat elke sub in de **X**-partitie verschillend is ($s_i \neq s_j$ als $i \neq j$), dan is α **gecomprimeerd**. Het is bewezen dat een Booleaanse functie $f(\mathbf{X}, \mathbf{Y})$ exact één gecomprimeerde **X**-partitie heeft [Dar11]. Omwille van deze eigenschap is een gecomprimeerde SDD canoniek.

Eigenschap 1 (X-partitie van $f \circ g$ [Dar11]) *Neem $\circ$ een Booleaanse operator, en $\{(p_1, s_1), \ldots, (p_n, s_n)\}$ en $\{(q_1, r_1), \ldots, (q_m, r_m)\}$ $\boldsymbol{X}$-partities van respectievelijk $f(\boldsymbol{X}, \boldsymbol{Y})$ en $g(\boldsymbol{X}, \boldsymbol{Y})$, dan is $\{(p_i \wedge q_j, s_i \circ r_j) \mid p_i \wedge q_j \neq \bot\}$ een $\boldsymbol{X}$-partitie van $f \circ g$.*

Deze eigenschap is zeer essentieel voor de bruikbaarheid van de SDD. Het maakt een polytijd apply-operatie mogelijk.

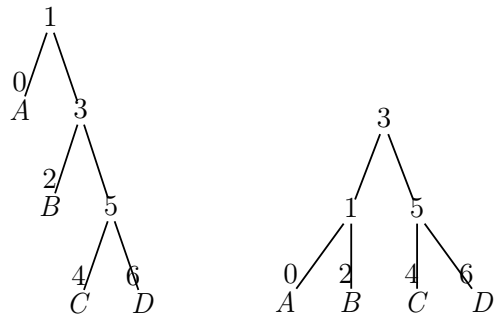
2.3 Vtree

In een OBDD wordt gesplist op variabelen. De volgorde waarop wordt gesplitst, is gespecifieerd in de variabelenvolgorde. Voor een SDD moet ook een volgorde worden vastgelegd, maar er wordt gesplitst op meerdere variabelen in één keer. Hiervoor is de vtree verantwoordelijk. Een vtree voor variabelen **X** is een binaire boom, vanwelke de bladeren een één-op-één correspondentie hebben met de variabelen in **X**. In figuur 2.2 kunnen we twee voorbeelden van een vtree zien. We refereren naar een knoop in de vtree of de deelboom geworteld in die knoop dmv. v. Het linker- en rechterkind of de linker- en rechterdeelboom van v duiden we aan met v^l en v^r. De ouder van v, indien die bestaat, duiden we aan met v^p. Het verschil tussen linkse en rechtse kindknopen in de vtree is belangrijk omwille van het verschil tussen primes en subs. In de SDD library wordt functionaliteit voorzien voor vier types vtrees:

- Rechtslineaire vtree: elk linkerkind is een bladknoop.
- Linkslineaire vtree: elk rechterkind is een bladknoop.
- Gebalanceerde vtree: elke bladknoop zit op dezelfde diepte.
- Willekeurige vtree.

Een SDD genormaliseerd voor een rechtslineaire vtree is een OBDD.

Nu we de vtree geïntroduceerd hebben, kunnen we de SDD formeel definiëren. De originele SDD paper [Dar11] introduceert een afbeelding $\langle \cdot \rangle$ van SDD's naar Booleaanse formules. Deze herhalen we hier omdat ze essentieel is om SDD's te begrijpen.



FIGUUR 2.2: Rechtslineaire en gebalanceerde vtree gebruikt voor figuur 2.1

Definitie 3 (Sentential Decision Diagram [Dar11]) *α is een SDD die vtree v respecteert asa*

1. *$\alpha = \bot$ of $\alpha = \top$, $\langle \bot \rangle = Onwaar$, $\langle \top \rangle = Waar$.*
2. *$\alpha = X$ of $\alpha = \neg X$ en v is een blad met variabele X. $\langle X \rangle = X$, $\langle \neg X \rangle = \neg X$.*
3. *$\alpha = \{(p_1, s_1), \cdots, (p_n, s_n)\}$, elke p_i is een SDD die een deelboom van v^l respecteert en elke s_i is een SDD die een deelboom van v^r respecteert. Bovendien moet $\langle p_1 \rangle, \cdots, \langle p_n \rangle$ een partitie vormen. $\langle \alpha \rangle = \bigvee_{i=1}^{n} \langle p_i \rangle \wedge \langle s_i \rangle$.*

Een belangrijke type SDD's zijn gecomprimeerde SDD's. Een SDD is gecomprimeerd a.s.a. voor elke decompositie $\{(p_1, s_1), \cdots, (p_n, s_n)\}$ in de SDD geldt dat $s_i \not\equiv s_j$ voor $i \neq j$. De grootte van een SDD α wordt gedefinieerd als de som van de groottes van al zijn decomposities en wordt geschreven als $|\alpha|$.

2.4 Apply-operatie

Een apply-operatie heeft als input twee kennisbasissen en een Booleanse operatie, en als output de combinatie van die twee kennisbassissen volgens die operatie. Hoe de apply-operatie werkt hangt af van het type kennisbasis. Zoals eerder vermeld is er voor OBDD's een polytijd apply-operatie. Het is relatief makkelijk om een algoritme te verzinnen dat een Booleaanse operatie op twee SDD's uitvoert in polytijd m.b.v. de eerder geziene eigenschap van **X**-partities: **X**-partitie van $f \circ g$ [Dar11]. Die eigenschap vereist wel dat beide SDD's genormaliseerd zijn voor dezelfde vtree. In algoritme 1 wordt pseudocode gegeven voor zo een apply-operatie. Het resultaat van die apply-operatie is een SDD, maar deze SDD is niet per se gecomprimeerd. Op lijn 12 kan het gebeuren dat de berekende sub s al in γ zit en opnieuw toegevoegd wordt, dit is in strijd met de voorwaarden voor een gecomprimeerde SDD. Een niet-gecomprimeerde SDD is niet canoniek. Zoals eerder vermeld is canoniciteit in de praktijk zeer interessant, dus kunnen we niet de standaard apply-operatie gebruiken. De aanpassingen aan de pseudo-code om canoniciteit te behouden zijn minimaal. We moeten enkel bij het toevoegen van (p, s) detecteren of er een element (q, s) al in

γ zit. Als dat zo is moeten we $(Apply(p, q, \vee), s)$ toevoegen aan γ. Dit aangepaste algoritme zullen we de compressie-apply-operatie noemen. Het is gebleken dat de compressie-apply-operatie niet in polytijd uitvoerbaar is [dBD15]. Het bewijs hiervan is gebaseerd op bepaalde logische formules waarvan de SDD's na de compressie-apply-operatie exponentieel groter zijn. Wel is uit datzelfde werk gebleken dat in de praktijk een compressie-apply-operatie wel degelijk verkiesbaar is. Aangezien in de praktijk canoniciteit een belangrijke eigenschap is zullen we in deze paper veronderstellen dat elke SDD een gecomprimeerde SDD is en met de apply-operatie bedoelen we de compressie-behoudende versie. Als nadeel moeten we nu veronderstellen dat elke apply-operatie van twee SDD's in theorie exponentieel veel tijd en geheugen nodig kan hebben.

Algorithm 1 Apply$(\alpha, \beta, \circ)$: α en β zijn SDD's genormaliseerd voor dezelfde vtreeknoop, $\circ$ is een Booleaanse operator [Dar11]

Cache(., ., .) = **nil** bij start.
Expand(γ) geeft $\{(\top, \top)\}$ terug als $\gamma = \top$; $\{(\top, \bot)\}$ als $\gamma = \bot$; anders γ.
UniqueD(γ) geeft $\top$ terug als $\gamma = \{(\top, \top)\}$; $\bot$ als $\gamma = \{(\top, \bot)\}$; anders de unieke SDD met elementen γ.

```
1: if α en β zijn constanten of variabelen then
2:     return α ∘ β
3: else if Cache(α, β, ∘) ≠ nil then
4:     return Cache(α, β, ∘)
5: else
6:     γ ← {}
7:     for all elementen (p_i, s_i) in Expand(α) do
8:         for all elementen (q_j, r_j) in Expand(α) do
9:             p ← Apply(p_i, q_j, ∧)
10:            if p is consistent then
11:                s ← Apply(s_i, r_j, ∘)
12:                voeg element (p, s) toe aan γ
13:            end if
14:        end for
15:    end for
16: end if
17: return Cache(α, β, ∘) ← UniqueD(γ)
```

Met behulp van de Apply-operatie kunnen we de negatie van een SDD meer in detail bekijken. Neem SDD α, dan is $\neg\alpha = \alpha \oplus \top =$**Apply**$(\alpha, \top, \oplus)$. Met behulp van eigenschap 1 kunnen we de operatie doorvoeren naar de subs van α.

Eigenschap 2 (Negatie van een SDD) *Neem* $\alpha = \{(p_1, s_1), \cdots, (p_n, s_n)\}$ *een SDD, dan is* $\neg\alpha = \{(p_1, \neg s_1), \cdots, (p_n, \neg s_n)\}$ *de inverse SDD.*

De negatie van α kan recursief verder gedaan worden op de subs s_i van α.

Hoofdstuk 3

Probleemstelling

Er is in het verleden veel onderzoek gedaan om compacte OBDD's efficiënt te bekomen. De SDD is in theorie bondiger dan de OBDD, maar er moeten verschillende zaken onderzocht worden om dat in de praktijk waar te maken. Onderzoeksonderwerpen zijn bijvoorbeeld het effect van de vtree op de grootte van de SDD, omvangslimieten, gespecialiseerde compilers, etc. In deze thesis onderzoeken we het probleem van grote tussenresultaten dat optreedt tijdens de bottom-up compilatie van een kennisbasis naar een SDD.

3.1 Bottom-up compilatie

Een kennisbasis/logische formule/logisch circuit kan in het algemeen als een Gewortelde Gerichte Acyclische Graaf (rooted-DAG) beschouwd worden. De rooted-DAG bestaat uit knopen die een conjunctie, disjunctie of negatie voorstellen van de kinderen van die knoop. De bladeren van de rooted-DAG stellen variabelen voor. Elke knoop stelt dus een deelformule voor van de kennisbasis. Een bottom-up compilatie zet eerst alle bladeren van de rooted-DAG om naar SDD's, die SDD's worden door middel van apply-operaties samengevoegd volgens de rooted-DAG. Op die manier wordt de SDD-representatie van elke knoop van de rooted-DAG berekend, en dit van onder naar boven. Een algoritme voor bottom-up compilatie van een rooted-DAG is beschreven in algoritme 2, en een diagram van een bottom-up compilatie is te vinden in figuur 3.1.

In de praktijk is gemerkt dat SDD's die tijdens de compilatie worden berekend, zogenaamde tussenresultaten, erg kunnen variëren in grootte. De SDD's die een knoop in de rooted-DAG voorstellen moeten verplicht berekend worden in de bottom-up compilatie (in figuur 3.1 de SDD's $\alpha, \alpha_1, \alpha_2, \alpha_3, \alpha_4$ en α_5). Die SDD's zijn onvermijdelijk nodig om andere SDD's hogerop in de rooted-DAG te berekenen (tenzij de structuur van de rooted-DAG wordt veranderd). In het algoritme komen deze onvermijdelijke SDD's overeen met de SDD's die worden gegeven als resultaat in regels 2, 5 en 13.

Waar we de bottom-up compilatie kunnen optimaliseren is bij de berekening van die onvermijdelijke SDD's. We kunnen telkens maar twee SDD's samennemen m.b.v.

Algorithm 2 Bottom-up compilatie(knoop): knoop is in het begin de rootknoop van de rooted-DAG

Heuristiek(SDD-lijst) kiest 2 SDD's uit SDD-lijst volgens de heuristiek. **SDD** stelt de functionaliteit van de SDD-library voor

```
1: if knoop.operatie = Atoom then
2:     return SDD.Literal(knoop.variabele)
3: end if
4: if knoop.operatie = Negatie then
5:     return SDD.Negatie(Bottom-up compilatie(knoop.kinderen[0]))
6: end if
7: kindCompilaties ← map(Bottom-up compilatie, knoop.kinderen)
8: while kindCompilaties bevat meer dan één element do
9:     γ_i, γ_j ← kindCompilaties.pop(), kindCompilaties.pop()
10:    γ_k ← Apply(γ_i, γ_j, knoop.operatie)
11:    kindCompilaties ← (kindCompilaties ∪γ_k)/{γ_i, γ_j}
12: end while
13: return kindCompilaties[0]
```

de apply-operatie. Als een knoop n van de rooted-DAG meer dan twee kinderen k_i heeft, moet er een keuze gemaakt worden welke kinderen met welke worden applied (Deze keuze gebeurt op regel 9 in algoritme 2). Niet elke apply-operatie duurt even lang en het bekomen tussenresultaat kan verschillen in grootte. Het eindresultaat, de SDD die overeenkomt met knoop n, zal altijd dezelfde zijn wegens de canoniciteit van SDD's.
De volgorde waarop de kinderen van knoop n worden samengevoegd, is te kiezen. We verkiezen een volgorde die zo snel mogelijk berekend kan worden.

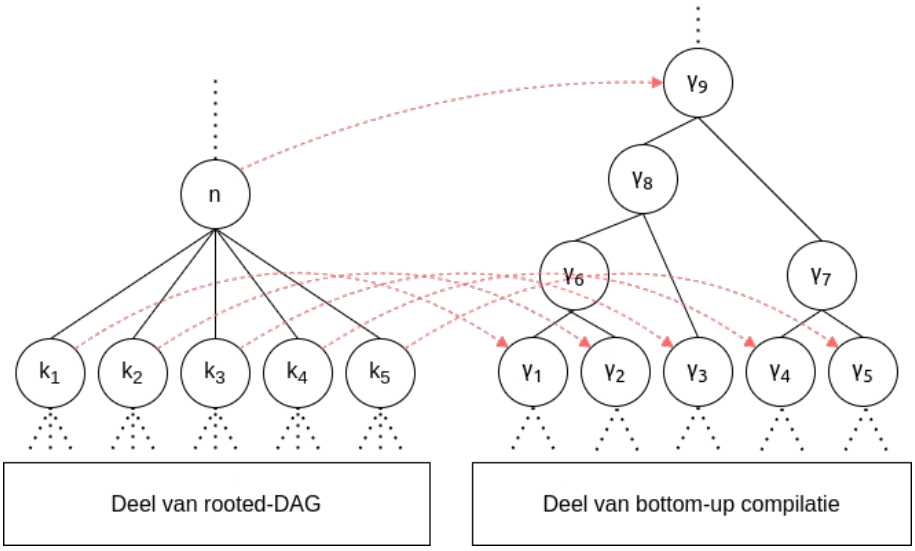
3.2 Formele probleemstelling

Het probleem kunnen we formeel als volgt stellen.

Definitie 4 (Bottom-up knoopcompilatie) *De bottom-up compilatie van één knoop $n = \bigcirc_{i=1}^{N} k_i$, met $\bigcirc$ de functionele compositie van een logische operatie, wordt geschreven als een sequentie van SDD's $\gamma_1, \cdots \gamma_M$, met $\gamma_M \equiv n$ en voor elke $i \in [1, M]$:*

1. *$\gamma_i \equiv k_i$ voor $i \in [1, N]$, of*
2. *$\gamma_i = Apply(\gamma_j, \gamma_k, \circ)$ met $j, k < i$ en γ_j en γ_k zijn genormaliseerd voor dezelfde vtree, of*
3. *$\gamma_i \equiv \gamma_j$ met $j < i$ en γ_i en γ_j zijn genormaliseerd voor verschillende vtrees.*

De derde mogelijkheid in deze definitie komt overeen met het dynamisch aanpassen van de vtree tijdens de compilatie. Dit zullen we niet gebruiken in deze thesis, zodat de keuze van de heuristiek het enige is dat effect heeft op de compilatietijd (meer



FIGUUR 3.1: Een mogelijke bottom-up compilatie van een rooted-DAG: γ is de SDD die de formule onder n voorstelt. Elk kind k_i van knoop n is gecompileerd naar een SDD γ_i

uitleg hierover is te vinden in sectie 4.1). We zullen enkel de eerste twee mogelijkheden gebruiken en elke γ_i slechts één keer gebruiken als parameter voor de Apply-operatie. Met die aannames zullen er $N-2$ tussenresultaten $\{\gamma_{N+1} \cdots \gamma_{2\times N-2}\}$ berekend worden voordat γ_M berekend kan worden, $M = 2 \times N - 1$. We willen de totale compilatietijd ct van knoop n minimaliseren. Deze wordt als volgt gemeten:

$$ct = \sum_{i=N+1}^{M} t(\gamma_i) \tag{3.1}$$

met $t(\gamma_i)$ de tijd nodig om γ_i te berekenen.

We verwachten dat de groottes van de tussenresultaten een grote invloed zullen spelen op de totale compilatietijd, met de voorkeur voor kleinere SDD's. Naast de compilatietijd zullen we dus ook deze groottes meten.

3.3 Voorbeeld

Neem de logische formule $f = (A \vee B) \wedge (B) \wedge (\neg B \vee C)$. We willen de SDD α overeenkomstig met deze formule berekenen. Eerst compileren we de drie deelformules van f naar een SDD.

$$\gamma_1 = (A \wedge \top) \vee (\neg A \wedge B) \equiv (A \vee B) \tag{3.2}$$

$$\gamma_2 = B \tag{3.3}$$

$$\gamma_3 = (B \wedge C) \vee (\neg B \wedge \top) \equiv (\neg B \vee C) \tag{3.4}$$

Nadat deze drie SDD's berekend zijn moet er een keuze gemaakt worden, berekenen we eerst $\gamma_1 \wedge \gamma_2$, $\gamma_1 \wedge \gamma_3$ of $\gamma_2 \wedge \gamma_3$.

$$\gamma_4^{'} = \gamma_1 \wedge \gamma_2 = B \tag{3.5}$$

$$\gamma_4^{''} = \gamma_1 \wedge \gamma_3 = (\neg A \wedge ((B \wedge C) \vee (\neg B \wedge \bot))) \vee (A \wedge ((B \wedge C) \vee (\neg B \wedge \top))) \tag{3.6}$$

$$\gamma_4^{'''} = \gamma_2 \wedge \gamma_3 = (B \wedge C) \vee (\neg B \wedge \bot) \tag{3.7}$$

In de laatste stap moeten we de SDD van de overgebleven deelformule toevoegen aan γ_4. Dit resultaat zal altijd hetzelfde zijn omwille van de canoniciteit.

$$\gamma_5 = (B \wedge C) \vee (\neg B \wedge \bot) \equiv f \tag{3.8}$$

In dit voorbeeld zien we dat $\gamma_4^{''}$ duidelijk groter is dan $\gamma_4^{'}, \gamma_4^{'''}$ en zelfs γ_5. We verwachten dat een Apply-operatie met $\gamma_4^{''}$ langer duurt dan een Apply-operatie met $\gamma_4^{'}$ of $\gamma_4^{'''}$ vanwege de grootte.

3.4 In de praktijk

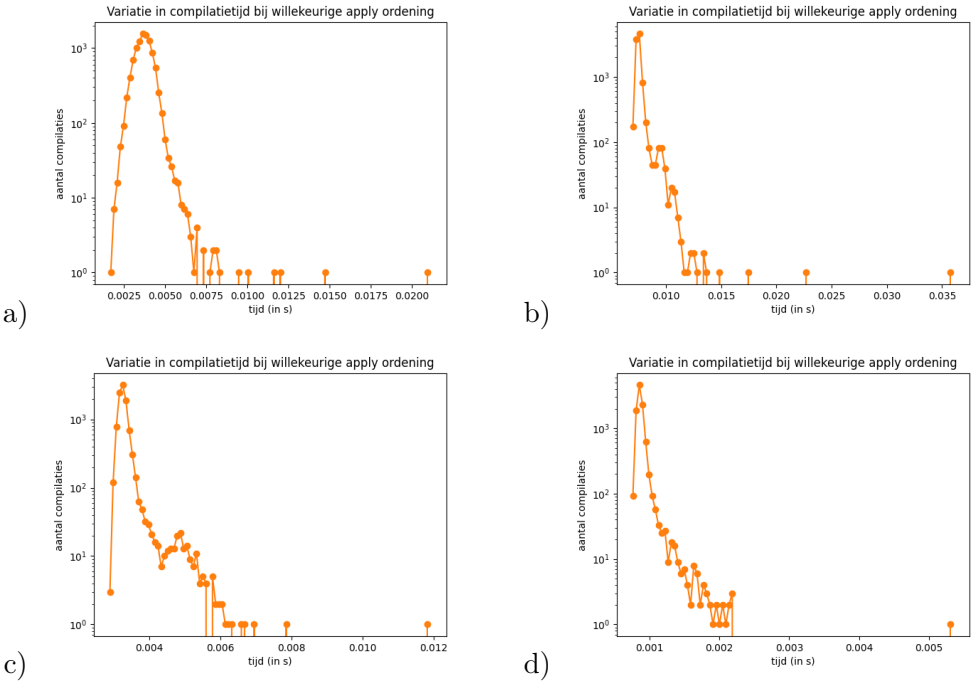
herschrijven

Aan één voorbeeld hebben we niet zoveel, om uit te testen hoe groot dit probleem is in de praktijk hebben we een klein experiment uitgevoerd. We genereren de kinderen k_i van een logische formule $n = \bigcirc_{i=1}^{N} k_i$ en zetten deze om naar de SDD's γ_i, daarna proberen we verschillende willekeurige sequentie's van $\gamma_{N+1}, \cdots \gamma_M$ en meten we de compilatietijd ct.

Hoe we k_i berekenen en andere details zijn terug te vinden in de sectie Experimenten. Wat voor nu belangrijk is, is dat we één probleeminstantie $n = \bigcirc_{i=1}^{N} k_i$ hebben. De verschillende k_i worden gegenereerd als 3-CNF's, met een bepaald aantal variabelen ν en clausules y. We compileren deze telkens met een willekeurige volgorde, we krijgen dus willekeurige sequentie's van $\gamma_1, \cdots \gamma_M$. Van elke compilatie meten we de tijd ct. Zo krijgen we een Gauss-curve van snelle tot trage sequenties.

We geven in tabel 3.1 de variatie weer, en we geven ook de ratio $r = y/\nu$ mee. Waarom dit nuttig is wordt ook pas later in sectie Experimenten verklaard. In het algemeen is de ratio een goede maat voor de grootte van de gecompileerde SDD. Wat we zien is dat er altijd wel wat variatie is in de compilatietijd. Dit is slechts voor één probleeminstantie, voor andere probleeminstanties is er soms meer en soms minder variatie.

tabellen fiksen met mooie ratio's



TABEL 3.1: Variatie van compilatietijd ct voor $r = 1/3$ (a), $r = 4/3$ (b), $r = 7/3$ (c) en $r = 10/3$ (d).

Hoofdstuk 4

Gerelateerd onderzoek

In de vorige secties hebben we de theorie van de SDD besproken, dit is belangrijk om de heuristieken te begrijpen die we later zullen introduceren. Nu bekijken we relevant onderzoek specifiek voor het probleem van de grote tussenresultaten. We bespreken enkele compilers, heuristieken voor een optimale vtree en een fase-transititie bij de grootte van kenniscompilaties. Een aanzienlijk deel van de hier aangehaalde bronnen beperken zich tot formules in Conjuntieve Normaalvorm. In deze thesis limiteren we ons juist niet tot enkel CNF's, maar tot algemene rooted-DAG's, toch zullen we er enige aandacht aan besteden omdat we CNF's wel gebruiken om SDD's te genereren.

In principe zijn de hieropvolgende technieken/heuristieken terug te brengen tot algemene heuristieken voor Beperking-Tevredenstellingsproblemen (CSP) zoals beschreven in [RN20]:

- **Opdelen in deelproblemen** door onafhankelijke componenten te vinden: Als twee SDD's α en β geen variabelen met elkaar delen, zijn ze onafhankelijk. Volledig onafhankelijke γ_i vinden zal meestal niet mogelijk zijn voor grote N, maar wellicht zijn er paren (γ_i, γ_j) met weinig overeenkomstige variabelen en paren met veel overeenkomstige variabelen.

- **Componenten met minst aantal mogelijke waarden** eerst bepalen: De grootte van een SDD α is het aantal elementen per vtreeknoop. Hoe het aantal elementen verandert na Apply met SDD β hangt af van de elementen die β bevat.

- **Componenten die in veel beperkingen een rol spelen** eerst bepalen: Een variabele aan de linkerzijde van de vtree (prime) is onderhevig aan meer restricties dan een variabele aan de rechterzijde (sub). De diepte van een variabele in de vtree speelt ook een rol.

Een resultaat om in het achterhoofd te houden werd geleverd door De Colnet en Mengel [dCM21]. Zij tonen aan dat de bottom-up compilatie naar structured-DNNF's (str-DNNF) van bepaalde formules exponentiële tijdscomplexiteit heeft, ondanks het bestaan van een constante str-DNNF representatie. De exponentiële tijdscomplexiteit

volgt uit tussenresultaten van exponentiële grootte, ongeacht de volgorde van Apply-operaties. str-DNNF is een superverzameling van SDD, dit resultaat geldt dus ook voor SDD's.

4.1 Vtree heuristieken

De OBDD heeft een variabelenvolgorde. Deze volgorde heeft een grote invloed op de totale grootte van de voorstelling. Daarom is er al veel onderzoek gedaan naar het zoeken van volgordes die resulteren in een relatief kleine OBDD. Zo is gebleken dat een optimale variabelenvolgorde vinden een NP-Compleet probleem is [BW96]. Omwille van die reden zijn er heuristieken ontwikkeld. Statische heuristieken zoals MINCE [AMS04] en FORCE [AMS03] bepalen voor de compilatie een goede volgorde, gebaseerd op de partitionering van variabelen volgens een Min-Cut. Dynamische heuristieken zoals het zeefalgoritme [Rud93] passen de variabelenvolgorde aan tijdens de compilatie, gebaseerd op wat er al reeds gecompileerd is. De ene techniek sluit de andere niet uit, er kan eerst statisch naar een goede startvolgorde gezocht worden, die volgorde kan dynamisch dan nog geoptimaliseerd worden. Een variabelenvolgorde kan ook beschouwd worden als een rechtslineaire vtree, waarin elk linkerkind een bladknoop is.

Voor de SDD geldt dat de vtree een grote invloed heeft op de grootte van voorstelling [CD13]. Er kan, net zoals bij de OBDD, op voorhand statisch een vtree bepaald worden aan de hand van structuur in de rooted-DAG of de vtree kan dynamisch aangepast worden tijdens de compilatie van de rooted-DAG naar SDD.

Er zijn al verschillende statische minimalisatieheuristieken onderzocht voor vtrees. Meestal in context van een specifiek type probleem met al dan niet een uitbreiding van de SDD als compilatierepresentatie. Een vtree-heuristiek voor symbolische model-verificatie wordt gegeven in [VL20]. Twee gelijkaardige heuristieken worden gegeven in [LBdB17] en [HKZDM19]. De vtrees worden in deze twee laatste gevallen gebruikt voor de structuur van de PSDD en P-XSDD, beiden uitbreidingen op de SDD. De drie heuristieken gebruiken gelijkaardige technieken als de statische variabelenvolgorde-heuristieken bij de OBDD, gebaseerd op een min-cut.

Een techniek voor dynamische minimalisatie is voorgesteld in [CD13]. Er wordt een manier aangehaald om de ruimte van vtrees te doorzoeken. Zo kan er tijdens de compilatie naar een betere vtree gezocht worden. Een compilatie-algoritme voor CNF's dat deze dynamische minimalisatie gebruikt werd ook voorgesteld.

Een betere vtree zoeken kan best op speciale momenten gebeuren, waarbij vermoedelijk de SDD genormaliseerd voor die vtree niet meer (veel) verandert. Stel de vtree v van een SDD γ_i wordt dynamisch geoptimaliseerd. Stel ook dat er nog een andere SDD is γ_j die variabelen met γ_i deelt. De vtree waarvoor γ_j is genormaliseerd, moet dezelfde deelvtree onder v hebben als de vtree waarvoor γ_i genormaliseerd is. γ_j zal aangepast moeten worden voordat Apply op γ_i en γ_j kan worden uitgevoerd.

Niet alleen kost de aanpassing aan γ_j extra tijd, na Apply moet de vtree onder v opnieuw dynamisch geoptimaliseerd worden.

De SDD γ_i genormaliseerd voor vtreeknoop v zal niet meer veranderen als elke SDD γ_j met variabelen in de deelboom v opgenomen is in γ_i, zelfs als γ_i met andere SDD's wordt applied. Zodra γ_i niet of bijna niet meer zal veranderen, kunnen we de deelboom v optimaliseren met de garantie dat er geen of weinig aanpassingen moeten gemaakt worden aan andere SDD's en dat de deelboom onder v optimaal blijft. De optimalisatie is "nevenkostenvrij".

De situatie waarin we de vtree nevenkostenvrij of bijna nevenkostenvrij kunnen optimaliseren komt normaal gezien niet veel voor in een rooted-DAG. Voor de compilatie van CNF's is dit wel bruikbaar. In een CNF-compiler worden enkel clausules bij elkaar gevoegd om de compilatie te verkrijgen. De kans dat we (bijna alle) clausules kunnen partitioneren over de vtree is groot, dit is exact wat gedaan wordt in heuristiek 1. In combinatie met de heuristiek kan de vtree-optimalisatie vaak toegepast worden én zijn de nevenkosten klein. Het resultaat is dat tijdens de compilatie de vtree deelboom per deelboom van onder naar boven dynamisch geoptimaliseerd wordt.

Het effect van de dynamische optimalisatie van de vtree tijdens de compilatie van een algemene rooted-DAG is nog niet bestudeerd, ondanks dat de techniek wel bestaat. Los daarvan heeft dynamische minimalisatie een invloed op de compilatietijd, terwijl we enkel het effect van de heuristieken op de compilatietijd willen meten. Daarom zullen we in de thesis veelal gebruik maken van een voorafbepaalde vtree (gebalanceerd, linkslineair,...) met willekeurige variabelenvolgorde, en passen we geen dynamische minimalisatie toe.

4.2 Compilers

De huidige state-of-the-art SDD-compiler voor CNF's is een top-down compiler. deze werd voorgesteld door Oztok en Darwiche [OD18], en maakt gebruik van DPLL. De compilatietaal van deze compiler is Decision-SDD, die minder bondig is dan SDD. Een top-down compiler heeft niet het probleem van grote tussenresultaten.

Een bottom-up compiler voor CNF's werd voorgesteld door Choi en Darwiche in [CD13]. Deze compiler doet aan dynamische minimalisatie door tijdens de compilatie aanpassingen aan de vtree te maken. Verder gebruikt de compiler twee heuristieken om een apply-volgorde op te leggen aan alle clausules.

Heuristiek 1 (Vtree partitionering) *Wijs elke clausule c toe aan de laagste vtreeknoop v die de variabelen van c bevat. Gebruik de vtree als recursieve partitionering van alle clausules, met elke vtreeknoop v verantwoordelijk voor een set van clausules. Doorloop de partitionering bottom-up van links naar rechts. De heuristiek compileert recursief de clausules gelinkt aan de kinderen van v.*

Zelfs als de clausules gepartitioneerd worden, kan het zijn dat er meer dan 2 clausules gelinkt worden aan dezelfde vtreeknoop v. In dat geval moet er nog altijd een keuze gemaakt worden welke clausules eerst applied worden. De compiler zal in

dit geval een tweede, interne, heuristiek raadplegen. In de paper wordt een tweede heuristiek gegeven, maar deze werkt enkel op CNF's. Verder in de thesis zullen we deze vervangen met onze eigen heuristieken.

Heuristiek 2 (Clausule-lengte) *De clausules van vtreeknoop v worden van kort naar lang (aantal variabelen) doorlopen.*

Heuristiek 1 wordt ook gebruikt in JUICE, een Julia package voor logische en probalitische circuits [DKL+21]. Hiervoor wordt de heurisitiek uitgebreidt om te werken op rooted-DAG's, niet enkel op CNF's. Er wordt geen interne heuristiek gebruikt.

Verder zijn er twee compilers voor specifieke probleeminstanties ontwikkelt. Een compiler voor probabilistische grafische modellen [CKD13] en een compiler voor probabilistische logische programma's [VRdBR14]. Beide compilers maken gebruik van structuur die voortkomt uit het type input om efficiënte compilatie mogelijk te maken. Deze structuur wordt ook gebruikt om een volgorde te geven aan de apply-operaties. De compilers verwachten een specifiek soort input, deze veronderstelling kan niet worden gemaakt voor de compilatie van een algemene rooted-DAG.

Het probleem van grote tussenresulaten komt ook voor bij de constructie van OBDD's. Daarom kijken we ook naar OBDD-compilers. Neem bijvoorbeeld MINCE [AMS04], naast een heuristiek voor de variabelenvolgorde worden ook heuristieken voorgesteld voor de volgorde waarin de clausules worden samengevoegd. Één voorstel werkt met de hypergraaf van de formule, elke knoop van de rooted-DAG is een knoop in deze hypergraaf. Voor elke variabele X wordt een verbinding gelegd tussen de knopen die X bevatten. Op die hypergraaf wordt gebalanceerde min-cut uitgevoerd om de clausules op te splitsen. Een tweede voorstel, voor CNF's, legt een volgorde op aan de clausules gebaseerd op de variabele van elke clausule, die het laatste in de variabelenvolgorde voorkomt. De clausules met variabelen die laat in de variabelenvolgorde voorkomen, zullen het eerst samengevoegd worden.

4.3 Fase-transitie

In het domein van SAT-problemen is er een concept genaamd fase-transitie. De moeilijkheid van het vinden van een bevredigende oplossing verandert abrupt voor bepaalde waarden van een parameter [CA96], [MSL92]. In K-SAT draait het om het vinden van een toewijzing van waarheidswaarden aan variabelen in een formule f in K-CNF zodat de formule geldig is.

De fase-transitie conjunctuur is gelinkt aan de verhouding r van het aantal clausules y ten opzichte van het aantal variabelen ν, $r = y/\nu$. De conjunctuur stelt dat voor K-SAT er een kritieke waarde r^* is. Als $r < r^*$ en $n \leftarrow \infty$, is er met kans $P = 1$ een combinatie van waarheidswaarden zodat f geldig is. Als $r > r^*$ en $n \leftarrow \infty$ is er met kans $P = 1$ geen combinatie van waarheidswaarden zodat f geldig is. In de praktijk is gemerkt dat oplossingsstrategieën voor K-SAT vooral moeite hebben met formules waarvoor $r \approx r^*$, en voor andere K-SAT is een oplossing vinden relatief makkelijk. We krijgen een makkelijk-moeilijk-makkelijk patroon.

In veel andere problemen is er een soortgelijk kritiek punt voor een bepaalde parameter, waarbij instanties met die parameter op het kritieke punt, moeilijk oplosbaar zijn. Als de parameter onder het kritieke punt ligt is het probleem onderbepaald, en bijgevolg makkelijk oplosbaar. Boven het kritieke punt is het probleem overbepaald en zijn er bijgevolg weinig oplossingen, een goed algoritme kan dan ook makkelijk een oplossing vinden [CKT91].

Er is ook een fase-transitie waar te nemen bij de groottes van kenniscompilaties en de compilatietijd, met als parameter de verhouding r van aantal clausules tot aantal variabelen van een CNF. Empirische studies naar de grootte van compilaties werden gedaan in [GWY15], en naar de grootte en compilatietijd in [GRM20]. De kritieke waarde voor r wordt geschat op $1,8$.

Verder in deze thesis zullen we SDD's genereren vanuit formules in CNF. Het is belangrijk dat we rekening houden met de verhouding r van deze CNF's omwille van die twee fasetransities. Wanneer we van twee CNF's f en g de conjunctie $h = f \wedge g$ nemen, is de ratio de som van de ratio van de CNF's $r_h = r_f + r_g$. Als we de conjunctie van n CNF's met ratio r doen, krijgen we een CNF met ratio $n \times r$. Als de ratio over de K-SAT fase-transitie gaat is de CNF met hoge kans triviaal onwaar, dit willen we vermijden omdat de Apply-operaties dan ook triviaal zijn, zie 2.2.

nakijken of dit ergens anders al uitgelegd is

Als we niet willen dat de ratio te groot wordt na toepassen van Apply, moeten we werken met DNF's. Als we de conjunctie van DNF's nemen hebben we niet het probleem dat de formule triviaal onwaar wordt. We weten dat een CNF kan omgezet worden naar een DNF door middel van negatie, de wetten van De Morgan en distributiviteit. De Morgan en distributiviteit behouden gelijkheid tussen formules. Omwille van de canoniciteit van SDD's, is de negatie van CNF's voldoende om SDD's te bekomen waarop de conjunctie kan uitgevoerd worden.

Gao *et al.* stellen ook een fase-transitie vast in termen van de gemiddelde lengte van de Primaire Implicaties (IP) van de formule. De lengte van de Primaire Implicaties is dan weer gelinkt aan de oplossingsuitwisselbaarheid, een term voor hoe hard verschillende oplossingen op elkaar lijken. In theorie kan men dus bepalen of een gegeven SDD onder- of overbepaald is op basis van de IP lengte.

iets dooen met die laatste twee zinnen over IP lengte

4.4 Conjunctieve normaalvorm

Zoals eerder vermeld zullen we bij onze experimenten gebruikmaken van willekeurig gegenereerde CNF's. Dit doen we omdat CNF's relatief makkelijk zonder bias genereerbaar zijn. De generatie gebeurt zoals beschreven in [MSL92]. We kunnen voor elke CNF drie parameters kiezen. k voor de lengte van de clausules, y voor het aantal clausules en ν voor het aantal variabelen. Elke CNF wordt gegenereerd door y clausules uniform en onafhankelijk te genereren. Elke clausule wordt gegenereerd door willekeurig k unieke variabelen te selecteren uit ν variabelen, en elke variabele met kans 0.5 negatief te maken. De reden dat we met willekeurige probleeminstanties werken is als volgt.

- We kunnen zoveel willekeurige instanties genereren als we nodig hebben.

- De instanties zijn aanpasbaar aan de hand van de drie parameters k, ν en y.
- Willekeurige instanties hebben geen structuur die gebruikt kan worden.

Tenslotte geven we nog een citaat uit [?] . "there is an argument that randomly generated problems represent a “core” of hard satisfiability problems. Certainly real problems have structure, regularity, symmetries, etc., and algorithms will have to make use of this structure to simplify the problems. However, once all the structure is “squeezed out”, the remainder will be a problem that requires search, and if all the structure is used up then the remainder will presumably be a random problem."

bibtex toevoegen

Hoofdstuk 5

Heuristieken

De heuristieken die we in deze sectie beschrijven dienen om telkens twee SDD's γ_i en γ_j te verwijderen uit een set van SDD's, die applied moeten worden. Het resultaat $\gamma_i \circ \gamma_j$ wordt toegevoegd aan de set van SDD's, zodat de set een element minder bevat. Dit komt overeen met de tweede optie in definitie 4: Bottom-up knoopcompilatie. Het zoeken wordt herhaald tot de set slechts uit één SDD bestaat, alle SDD's zijn dan samengevoegd tot één SDD. In Algoritme 2 van de bottom-up compilatie moet de heuristiek als extra parameter worden meegegeven, en moet regel 9 worden aangepast naar $\gamma_i, \gamma_j \leftarrow$ Heuristiek(kindCompilaties) om de heuristiek te gebruiken. Heuristiek 1: Vtree partitionering is moeilijker te beschrijven op deze manier. Voor deze heuristiek wordt een gespecialiseerd algoritme gegeven in sectie 5.2.1. Van elke heuristiek geven we de tijdscomplexiteit, die bovenop de tijdscomplexiteit van de compilatie komt. De tijdscomplexiteit zal onder andere in functie zijn van N, het aantal SDD's dat applied moet worden, en ν, het aantal variabelen aanwezig in de probleeminstantie.

We gebruiken heuristieken om een compilatiesequentie te genereren waarvan we de compilatietijd *ct* kunnen meten. Deze meting zegt echter niks als we geen compilatiesequentie hebben waarmee we kunnen vergelijken. Als basisgeval implementeren we een willekeurige heuristiek.

Heuristiek 3 (Willekeurig) *kies willekeurig twee SDD's*

5.1 Bovengrens na apply

Een eerste heuristiek die we voorstellen is gebaseerd op de maximale grootte na Apply $|\gamma_i \circ \gamma_j|$. We vermoeden dat kleinere SDD's gunstiger zijn voor de compilatietijd, daarom willen we eerst de Apply-operatie doen met de kleinste maximale grootte. Uit eigenschap 1 volgt dat voor de polytijd Apply-operatie geldt:

$$|\gamma_i \circ \gamma_j| \leq |\gamma_i| * |\gamma_j| \tag{5.1}$$

Deze bovengrens geldt niet voor de compressie-apply operatie (niet polytijd), desondanks is deze bovengrens wel een goede schatting. In de praktijk kunnen we deze heuristiek implementeren door alle SDD's te sorteren op hun grootte en telkens de twee kleinsten te kiezen.

Heuristiek 4 (Kleinste eerst) *Kies de twee SDD's met de kleinste grootte.*

Het berekenen van de grootte van een SDD α heeft een tijdscomplexiteit van $O(|\alpha|)$. De complexiteit van deze heuristiek is $O(max_{\gamma_i}(|\gamma_i|) \times N \times log(N))$.

In de praktijk is het mogelijk om de grootte van een SDD tijdens runtime bij te houden, zodat het opvragen hiervan in constante tijd kan gebeuren.

Heuristiek 4 is niet optimaal, er zijn betere, strengere bovengrenzen en er zijn andere zaken waarmee rekening kan gehouden worden, zoals de variabelen die in de SDD aanwezig zijn. We doen een tweede poging door middel van een strengere bovengrens.

5.1.1 Bovengrens door elementen

De grootte van een SDD α is de som van de groottes van de decomposities. We kunnen dit ook anders schrijven:

$$|\alpha| = \sum_{v \in V} \#el(\alpha, v) \tag{5.2}$$

met $\#el(\alpha, v)$ het aantal elementen onder vtreeknoop v in SDD α en V de set van alle vtreeknopen. Als we algoritme 1 bestuderen, zien we dat het aantal recursieve oproepen afhangt van het aantal elementen per vtreeknoop v.

$$\#el(\alpha \circ \beta, v) \leq \#el(\alpha, v) \times \#el(\beta, v) \tag{5.3}$$

We vullen vergelijking 5.3 in in vergelijking 5.2 om de volgende bovengrens te bekomen.

$$|\alpha \circ \beta| = \sum_{v \in V} \#el(\alpha, v) \times \#el(\beta, v) \tag{5.4}$$

In de meeste gevallen zal deze functie gedomineerd worden door $(max_{\gamma_i}(|\gamma_i|)$. Het kan daarom nuttig zijn om de grootte van SDD's tijdens de runtime bij te houden zodat dit niet apart berekend moet worden.

5.1.2 Extra bovengrens dmv. primes

Er is een tweede, meer theoretische, bovengrens voor het aantal elementen in een vtreeknoop v, gebaseerd op de variabelenverzameling van de primes.

Definitie 5 (Maximum grootte van een partitie) *Een partitie over ν variabelen bestaat maximimaal uit 2^ν primes.*

Bewijs door inductie: De grootste partitie over 1 variabele A bestaat uit de primes $\{A, \neg A\}$. Neem aan dat een partitie over $\nu - 1$ variabelen maximaal uit $2^{\nu-1}$ primes bestaat. Neem de partitie $PAR_{\nu-1}$ een partitie van maximale grootte

$$PAR_{\nu-1} = \{P_1, P_2, \cdots, P_{2^{\nu-1}}\} \tag{5.5}$$

We weten door de maximale grootte van $PAR_{\nu-1}$ dat voor eenderwelke prime $Q_{\nu-1}$ over $\nu-1$ variabelen moet gelden dat er een prime $P_i \in PAR_{\nu-1}$ bestaat zodat $P_i \wedge Q_{\nu-1} = \bot$ of P_i en $Q_{\nu-1}$ zijn voldaanbaar door dezelfde instantie I van variabelen.

We breiden $PAR_{\nu-1}$ uit naar een partitie over ν variabalen door de nieuwe variabele O toe te voegen.

$$PAR_\nu = \{P_1 \wedge O, P_1 \wedge \neg O, P_2 \wedge O, P_2 \wedge \neg O, \cdots, P_{2^{\nu-1}} \wedge O, P_{2^{\nu-1}} \wedge \neg O\} \quad (5.6)$$

. De grootte van PAR_ν is $2^{\nu-1} \times 2 = 2^\nu$. Eenderwelke prime Q_ν over ν variabelen kunnen we schrijven als de conjunctie van een prime Q_O over de variabele O en een prime $Q_{\nu-1} = Q_\nu \mid Q_O$:

$$\begin{aligned} Q_\nu &= Q_{\nu-1} \wedge O \text{ of,} \\ Q_\nu &= Q_{\nu-1} \wedge \neg O \end{aligned} \quad (5.7)$$

Er zijn twee gevallen: Ofwel is er een $P_i \in VAR_{\nu-1}$ zodat $Q_{\nu-1}$ en P_i voldaanbaar zijn door dezelfde instantie I van variabelen. Dan geldt dat voor $Q_\nu = Q_{\nu-1} \wedge O$ de instantie $I \wedge O$ zowel Q_ν als $P_i \wedge O \in VAR_\nu$ voldaanbaar maakt. Voor $Q_\nu = Q_{\nu-1} \wedge \neg O$ maakt de instantie $I \wedge \neg O$ zowel Q_ν als $P_i \wedge \neg O \in VAR_\nu$ voldaanbaar.

Het andere geval is wanneer er een prime $P_i \in VAR_{\nu-1}$ is zodat $P_i \wedge Q_{\nu-1} = \bot$. Voor elke mogelijke Q_ν bestaat er een prime $(P_i \wedge P_O) \in PAR_\nu$ zodat $(P_i \wedge P_O) \wedge Q_\nu = \bot$, met $P_O = O$ of $\neg O$.

$$\begin{aligned} (Q_{\nu-1} \wedge O) \wedge (P_i \wedge \neg O) &= O \wedge \neg O = \bot \\ (Q_{\nu-1} \wedge \neg O) \wedge (P_i \wedge O) &= O \wedge \neg O = \bot \end{aligned} \quad (5.8)$$

Er kan geen prime Q_ν toegevoegd worden aan PAR_ν zonder de voorwaarden van een partitie te breken, dit bewijst dat PAR_ν een partitie van maximale grootte is. □

Neem $\{(p_1(\mathbf{X}), s_1(\mathbf{Y})), \ldots (p_n(\mathbf{X}), s_n(\mathbf{Y}))\}$ en $\{(q_1(\mathbf{X}), r_1(\mathbf{Y})), \ldots (q_m(\mathbf{X}), r_m(\mathbf{Y}))\}$ de elementen van SDD's $\alpha(\mathbf{X}, \mathbf{Y})$ en $\beta(\mathbf{X}, \mathbf{Y})$. α en β zijn genormaliseerd voor vtreeknoop v. Elke p_i en q_j is een SDD over de variabelen $\mathbf{X}$. Het kan zijn dat een variabele $X \in \mathbf{X}$ in geen enkele p_i of q_j voorkomt. We schrijven daarom dat elke p_i een SDD is over de variabelen $\mathbf{X}_\alpha$ en elke q_j een SDD over de variabelen $\mathbf{X}_\beta$. De primes van een SDD moeten een partitie vormen. Volgens definitie 5 kan een partitie over de variabelen $\mathbf{X}$ kan maximaal uit $2^{|\mathbf{X}|}$ primes bestaan.

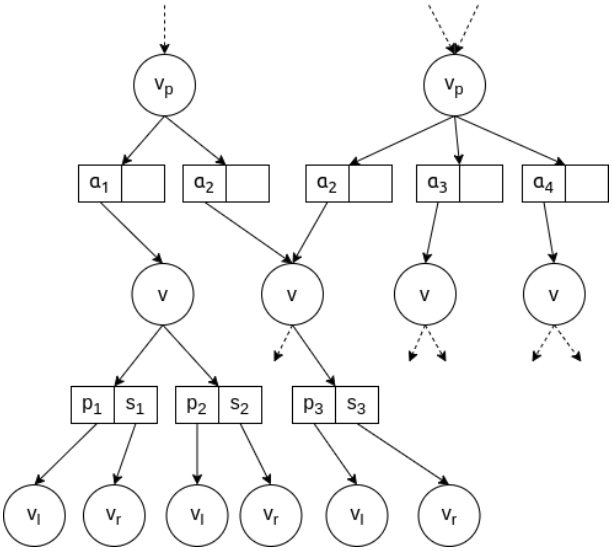
$$\#el(\alpha, v) \leq 2^{|X_\alpha|} \quad \#el(\beta, v) \leq 2^{|X_\beta|} \quad (5.9)$$

We zijn geïnteresseerd in het aantal elementen van $\alpha \circ \beta$ onder vtreeknoop v.

$$\alpha \circ \beta = \{(p_i \wedge q_j, s_i \circ r_j) | p_i \wedge q_j \neq \bot\} \quad (5.10)$$

Na de apply-operatie moeten de primes $p_i \wedge q_j$ een partitie moeten vormen over de variabelen $\mathbf{X}_\alpha \cup \mathbf{X}_\beta$. Het aantal elementen onder vtreeknoop v voor $\alpha \circ \beta$ is

$$\#el(\alpha \circ \beta, v) \leq 2^{|X_\alpha \cup X_\beta|} \quad (5.11)$$



FIGUUR 5.1: Deel van SDD Ψ, met vtreeknopen v^p, v, v^l en v^r

Vergelijking 5.11 geldt enkel omdat v de wortelknoop van α en β is. Het kan zijn dat α en β deel is van de grotere SDD's Ψ en Ω zoals in figuur 5.1. Neem v^p de ouderknoop van vtreeknoop v. Elk element in Ψ onder vtreeknoop v^p bevat een α_i als prime of als sub als v het linker- dan niet het rechterkind van v^p is. In totaal zijn er maximum $\#el(\Psi, v^p)$ verschillende α_i. Elke α_i kan bestaan uit andere elementen. Hetzelfde geldt voor het aantal β_i in Ω.

$$\#el(\Psi, v) \leq \#el(\Psi, v^p) \times 2^{|X_\alpha|} \quad \#el(\Omega, v) \leq \#el(\Omega, v^p) \times 2^{|X_\beta|} \tag{5.12}$$

In deze vergelijking zijn $\mathbf{X}_p$ en $\mathbf{X}_q$ de sets van variabelen die een voorkomen hebben in een prime van respectievelijk α_i en β_i. In de apply-operatie $\Psi \circ \Omega$ zal elk element onder vtreeknoop v^p van Ψ met elk element onder v^p van Ω applied worden. Voor elk van die apply-operaties geldt vergelijking 5.11. We kunnen een vergelijking opschrijven voor het aantal elementen in $\Psi \circ \Omega$ onder v:

$$\#el(\Psi \circ \Omega, v) \leq \#el(\Psi \circ \Omega, v^p) \times 2^{|X_\alpha \cup X_\beta|} \tag{5.13}$$

We hebben nu wel een bovengrens voor $\#el(\Psi \circ \Omega, v)$, maar de vraag is of deze ook nuttig is, is deze bovengrens strenger dan die in vergelijking 5.3? Als we de vergelijkingen van 5.12 invoegen in vergelijking 5.3 krijgen we

$$\#el(\Psi \circ \Omega, v) \leq \#el(\Psi, v^p) \times \#el(\Omega, v^p) \times 2^{|X_\alpha| + |X_\beta|}. \tag{5.14}$$

De bovengrens in 5.13 kan kleiner zijn dan bovengrens in 5.14 als $\mathbf{X}_\alpha \cap \mathbf{X}_\beta \neq \emptyset$. Hoe kleiner het symmetrisch verschil $\mathbf{X}_\alpha \triangle \mathbf{X}_\beta$ is, hoe beter vergelijking 5.13 de bovengrens gaat beïnvloeden.

5.1.3 Extra bovengrens dmv. subs

Er kan een tweede soortgelijke bovengrens gevonden worden in termen van de subs.

Definitie 6 (Aantal verschillende SDD's over ν variabelen) *Er bestaan 2^{2^ν} verschillende gecomprimeerde SDD's over ν variabelen*

Bewijs: Een kennisbasis over ν variabelen bestaat uit een combinatie van werelden, elke wereld is een instantie van de ν variabelen. In een wereld is elke variabele ofwel waar ofwel onwaar, we hebben 2^ν mogelijke werelden. Ofwel zit een wereld in de kennisbasis, ofwel niet, we kunnen op die manier 2^{2^ν} verschillende kennisbasissen creëren. Wegens de canoniciteit van gecomprimeerde SDD's kunnen we een één-op-ééń mapping maken tussen gecomprimeerde SDD's en kennisbasissen. Het aantal verschillende SDD's is bijgevolg gelijk aan het aantal verschillende kennisbassisen $\square$.

Neem opnieuw $\{(p_1(\mathbf{X}), s_1(\mathbf{Y})), \ldots (p_n(\mathbf{X}), s_n(\mathbf{Y}))\}$ en $\{(q_1(\mathbf{X}), r_1(\mathbf{Y})), \ldots (q_m(\mathbf{X}), r_m(\mathbf{Y}))\}$ de elementen van SDD's $\alpha(\mathbf{X}, \mathbf{Y})$ en $\beta(\mathbf{X}, \mathbf{Y})$. elke s_i en r_j is een SDD over de variabelen $\mathbf{Y}$. Zoals bij de primes schrijven we specifieker dat elke s_i een SDD over $\mathbf{Y}_\alpha$ en elke r_j een SDD over $\mathbf{Y}_\beta$ is. De subs van een SDD moeten geen partitie vormen, dus elke s_i kan eenderwelke SDD zijn. Als de subs s_i en s_j gelijk zijn worden de elementen (p_i, s_i) en (p_j, s_j) samengenomen naar $(p_i \vee p_j, s_i)$. Het aantal elementen onder v is bijgevolg gelimiteerd door het aantal mogelijke verschillende subs. Met definitie 6 kunnen we volgende ongelijkheden opstellen.

$$\#el(\alpha, v) \leq 2^{(2^{|Y_\alpha|})} \quad \#el(\beta, v) \leq 2^{(2^{|Y_\beta|})} \tag{5.15}$$

Op dezelfde wijze als bij de primes kunnen we een bovengrens bekomen voor het aantal elementen van $\alpha \circ \beta$, zoals in 5.11, maar dan voor $\mathbf{Y}_\alpha$ en $\mathbf{Y}_\beta$ en met een extra macht.

$$\#el(\alpha \circ \beta, v) \leq 2^{(2^{|Y_\alpha \cup Y_\beta|})} \tag{5.16}$$

Dit kunnen we dan weer uitbreiden naar het meer algemene geval $\Psi \circ \Omega$.

$$\#el(\Psi \circ \Omega, v) \leq \#el(\Psi, v^p) \times \#el(\Omega, v^p) \times 2^{(2^{|Y_\alpha \cup Y_\beta|})} \tag{5.17}$$

Deze bovengrens is in de praktijk minder nuttig dan 5.13, omdat de bovengrens een macht extra heeft in de vergelijking.

De drie bovengrenzen in 5.3, 5.13 en 5.17 geven ons de vergelijking

$$\#el(\alpha \circ \beta, v) \leq Min \left\{ \begin{array}{l} \#el(\alpha, v) \times \#el(\beta, v) \\ \#el(\alpha \circ \beta, v^p) \times 2^{|X_\alpha \cup X_\beta|} \\ \#el(\alpha \circ \beta, v^p) \times 2^{(2^{|Y_\alpha \cup Y_\beta|})} \end{array} \right\} \tag{5.18}$$

Met v^p de ouderknoop van v, $\mathbf{X}_\alpha$ en $\mathbf{X}_\beta$ de set van variabelen in α en β onder vtreeknoop v^l en $\mathbf{Y}_\alpha$ en $\mathbf{Y}_\beta$ de set van variabelen in α en β onder vtreeknoop v^r.

We merken op dat de bovengrens op $\#el(\alpha \circ \beta, v)$ afhangt van de bovengrens op $\#el(\alpha \circ \beta, v^p)$. In de praktijk is het dus belangrijk dat de vtree van boven naar onder wordt doorlopen tijdens het berekenen van de totale bovengrens.

Heuristiek 5 (Elementgewijze bovengrens) *Kies de twee SDD's γ_i, γ_j waarvoor $\sum_{v \in V} \#el(\gamma_i \circ \gamma_j, v)$ de kleinste bovengrens heeft.*

Net zoals bij heuristiek 4 geldt de gegeven bovengrens enkel voor de polytijd apply-operatie. In de gecomprimeerde versie is de bovengrens in vergelijking 5.18 niet strikt (vergelijking 5.3 geldt dan niet). Toch vermoeden we dat deze heuristiek goed kan werken.

5.1.4 Praktische implementatie

In de SDD library kunnen we een SDD α volledig doorlopen en in een lijst opslaan hoeveel elementen de SDD bevat per vtreeknoop, de tijdscomplexiteit hiervan is $O(|\alpha|)$. In theorie zou deze telling ook bijgehouden kunnen worden tijdens runtime, zodat het niet apart berekend moet worden. De lijsten van twee SDD's kunnen we paarsgewijs vermenigvuldigen om een eerste schatting te krijgen van het aantal elementen na de apply-operatie. De lijsten bevatten een getal voor elke vtreeknoop, de lengte van de lijsten is $O(\nu)$, met ν het aantal variabelen in de kennisbasis.

Een belangrijk gegeven bij de telling is als volgt. Stel een SDD γ_i is niet genormaliseerd voor vtreeknoop v, maar het linkerkind v^l. Dan is $\{(\gamma_i, \top), (\neg\gamma_i, \bot)\}$ een SDD genormaliseerd voor vtreeknoop v. Ook al lijkt het initieel alsof γ_i maar één element heeft onder v, omdat γ_i een prime is, en de primes een partitie moeten vormen, heeft γ_i twee elementen onder v. Verder weten we door eigenschap 2 dat de negatie doorgevoerd kan worden in de subs. Om een correcte telling te bekomen moet het aantal elementen worden verdubbeld voor elke vtreeknoop op het meest rechtse pad van de vtree v^l ($\neg\gamma_i$ is namelijk genormaliseerd voor v^l).

Naast het tellen van de elementen per vtreeknoop kunnen we ook een lijst opvragen van alle variabelen die aanwezig zijn in een SDD. Deze lijst moeten we dan omzetten naar een lijst van aantal variabelen per vtreeknoop. Met die informatie kunnen we vergelijking 5.18 gebruiken om de eerste schatting bij te stellen, de tijdscomplexiteit hiervan is $O(\nu)$.

Van elk paar SDD's berekenen we de bovengrens van de grootte na apply, deze bovengrenzen sorteren we, en we kiezen het paar SDD's met de kleinste bovengrens. Het sorteren kost $O(N^2 \times log(N^2))$ tijd.

De totale tijdscomplexiteit van deze heuristiek is $O((max_{\gamma_i}(|\gamma_i|) + \nu + log(N^2)) \times N^2)$. De complexiteit van deze heuristiek wordt net zoals heuristiek Kleinste eerst gedomineerd worden door $(max_{\gamma_i}(|\gamma_i|))$. Voor deze heuristiek kan het nuttig zijn om het aantal elementen per vtreeknoop tijdens de runtime bij te houden zodat dit in constante tijd kan worden opgevraagd.

5.2 Heuristieken op basis van vtree

5.2.1 Partitionering volgens de vtree

Deze sectie beschrijft de praktische implementatie van heuristiek 1. Zoals eerder vermeld wordt deze gebruikt in JUICE en de CNF-compiler. In tegenstelling tot andere heuristieken legt deze heuristiek geen directe volgorde op aan de SDD's α_i, maar worden de SDD's recursief gepartitioneerd volgens de vtree. Daarom geven we een specifiek algoritme voor de bottom-upcompilatie met deze heuristiek in Algoritme 3. In de laatste lijn, waarbij de SDD's voor dezelfde vtree-node zijn genormaliseerd, wordt algoritme 2 opgeroepen, met een secundaire heuristiek. De complexiteit van de heuristiek hangt af van de complexiteit van **Partitioneer**(SDDs, vtreeknoop v). De SDD library voorziet rechtsreekse functionaliteit om te testen of een vtree een deelboom is van een andere vtree, in constante tijd. De opsplitsing van de kinderen gebeurt door te kijken of het kind genormaliseerd is voor een vtreeknoop in de linkerdeelboom of rechterdeelboom van v. Een oproep van **Partitioneer** kost $O(N)$ tijd. Door de recursiviteit van **PartitieCompilatieKnoop** kan **Partitioneer** meerdere keren opgeroepen worden op een lijst van SDD's. Het aantal recursieve oproepen is gelimiteerd door de diepte van de vtree, die op zijn beurt gelimiteerd is door het aantal variabelen ν. De tijdscomplexiteit van deze heuristiek is bijgevolg $O(N * \nu)$, hierbij is niet de tijdscomplexiteit van de secundaire heuristiek gerekend.

5.2.2 Variabelenvolgorde op basis van de vtree

We hebben al heuristieken bepaald op basis van een bovengrens en op basis van de vtree. In deze subsectie proberen we heuristieken te bepalen op basis van de variabelen in de SDD's, en waar die variabelen zich bevinden in de vtree.

We bekijken de apply-operatie van SDD α met variabele x.

$$\alpha = \{(p_1, s_1), \ldots, (p_n, s_n)\} = \{(p_i, s_i) | i \in [1, n]\} \tag{5.19}$$

α is genormaliseerd voor vtreeknoop v. Als x een variabele onder v^l is dan geldt:

$$\alpha \circ x = \alpha \circ \{(x, \top), (\neg x, \bot)\} \tag{5.20}$$

$$\alpha \circ x = \{(p_i \wedge x, s_i \circ \top), (p_i \wedge \neg x, s_i \circ \bot) | i \in [1, n]\} \tag{5.21}$$

De operatie $\circ$ kan ofwel een conjunctie of een disjunctie voorstellen. We kunnen de elementaire gelijkheden toepassen op basis van de operatie, zie tabel 2.2. In geval van een conjunctie wordt vergelijking 5.21

$$\alpha \wedge x = \{(p_i \wedge x, s_i), (p_i \wedge \neg x, \bot) | i \in [1, n]\} \tag{5.22}$$

De primes $p_i \wedge \neg x$ hebben dezelfde sub, om compressie te behouden zullen ze samengenomen worden door middel van een disjunctie. Omdat we weten dat alle primes p_i een partitie moeten vormen kunnen we dit verder versimpelen.

$$\bigvee_{i=1}^{n} (p_i \wedge \neg x) = \bigvee_{i=1}^{n} (p_i) \wedge \neg x = \top \wedge \neg x = \neg x \tag{5.23}$$

Algorithm 3 PartitieCompilatie(knoop): knoop is in het begin de rootknoop van de rooted-DAG

Partitioneer(kinderen, v) geeft drie lijsten terug, één met alle kinderen die onder v^l zitten, één met alle kinderen onder v^r, en de overige kinderen. **interneHeuristiek** is de secundaire heuristiek die wordt gebruikt. **SDD** stelt de functionaliteit van de SDD-library voor.

```
1: if knoop.operatie = Atoom then
2:     return SDD.Literal(knoop.variabele)
3: end if
4: if knoop.operatie = Negatie then
5:     return SDD.Negatie(PartitieCompilatie(knoop.kinderen[0]))
6: end if
7: kindSDDs = map(PartitieCompilatie, knoop.kinderen)
8: return PartitieCompilatieKnoop(kindSDDs, knoop.vtreeknoop)
```

PartitieCompilatieKnoop(kindSDDs, heuristiek, v)

```
1: links, rechts, midden = Partitioneer(kindSDDs, v)
2: if links bevat mistens één element then
3:     midden ← midden ∪ PartitieCompilatieKnoop(links, v^l)
4: end if
5: if rechts bevat mistens één element then
6:     midden ← midden ∪ PartitieCompilatieKnoop(rechts, v^r)
7: end if
8: return Bottom-upCompilatie(midden, interneHeuristiek)
```

We kunnen dus vergelijking 5.22 nog verder versimpelen naar

$$\alpha \wedge x = \{(p_i \wedge x, s_i) | i \in [1, n]\} \cup \{(\neg x, \bot)\} \tag{5.24}$$

Voor de disjunctie wordt vergelijking 5.21

$$\alpha \vee x = \{(p_i \wedge x, \top), (p_i \wedge \neg x, s_i) | i \in [1, n]\} \tag{5.25}$$

Op gelijke manier als voor vergelijking 5.22 kunnen we vergelijking 5.26 verder versimpelen naar

$$\alpha \vee x = \{(p_i \wedge \neg x, s_i) | i \in [1, n]\} \cup \{(x, \top)\} \tag{5.26}$$

Het kan ook zijn dat de variabele x onder v^r valt i.p.v. v^l, dan krijgen we een andere situatie.

$$\alpha \circ x = \alpha \circ (\top \wedge x) \tag{5.27}$$

$$\alpha \circ x = \{(p_i, s_i \circ x) | i \in [1, n]\} \tag{5.28}$$

In dit geval kan het zijn dat de sub $s_i \circ x$ gelijk is aan sub $s_j \circ x$ en zullen primes p_i en p_j samengenomen worden. Deze compressie-behoudende stap is exact wat kan leiden tot exponentiële groei van de grootte van $\alpha \circ x$ [dBD15].

De verschillende besproken gevallen lijsten we hier op in tabelvorm:

$\alpha \circ x$	$x \in v^l$	$x \in v^r$
Conjunctie	$\{(p_i \wedge x, s_i) \mid i \in [1,n]\} \cup \{(\neg x, \bot)\}$	$\{(p_i, s_i \wedge x) \mid i \in [1,n]\}$
Disjunctie	$\{(p_i \wedge \neg x, s_i) \mid i \in [1,n]\} \cup \{(x, \top)\}$	$\{(p_i, s_i \vee x) \mid i \in [1,n]\}$

(5.29)

Zolang x voorkomt in v^l, dus als prime, groeit de grootte van de SDD maar met 1. Als x in v^r voorkomt kan het zijn dat subs gelijk worden en primes samengenomen moeten worden.

Al de vergelijkingen en de tabel zijn veralgemeenbaar naar $\alpha \circ \beta$, met β een SDD genormaliseerd voor v^l of v^r. We concluderen dat in algoritme 1, zolang β onder v^l valt, de SDD slechts met 1 knoop groeit. Zodra β onder v^r valt, kan de compressiestap optreden. Als β niet genormaliseerd is voor v^l of v^r, maar v kunnen we niks speciaals zeggen over de apply-operatie.

Gebaseerd op tabel 5.29 kunnen we een heuristiek bedenken: apply eerst die SDD's, die voorkomens hebben van variabelen die links in de vtree voorkomen. In onze experimentatie werkte dit minder goed voor niet-gebalanceerde vtrees, We moeten ook rekening houden met hoe hoog elke variabele in de vtree voorkomt. We willen prioriteit geven aan variabelen die hoog voorkomen.

Heuristiek 6 (Top-down variabelenvolgorde) *Leg een volgorde op aan alle variabelen door de vtree breedte-eerst van links naar rechts te doorlopen. Kies de twee SDD's die een variabele bevatten die als eerste voorkomt in de volgorde.*

De implementatie van deze heuristiek vereist het sorteren van de SDD's, en het sorteren vereist de vergelijking van de variabelenvoorkomens. De tijdscomplexiteit is $O(N * log(N) * \nu)$.

De heuristiek geeft een volgorde aan de apply-operaties dmv. de meest belangrijke variabele in de SDD (Bij gelijke voorkomens wordt naar de tweede belangrijkste variabele gekeken enzv.). Met veel andere zaken wordt niet rekening gehouden in deze heuristiek, toch zijn de resultaten interessant zullen we verder zien.

5.2.3 Inverse variabelenvolgorde

We herbekijken de tweede techniek voorgesteld bij MINCE (zie sectie 4.2). De techniek legt een volgorde op aan de clausules gebaseerd op de variabele met het laatste voorkomen in de variabelenvolgorde. De uitbreiding van deze techniek naar rooted-DAGs en vtrees gaat als volgt: Doorloop de vtree breedte-eerst om een variabelenvolgorde te bekomen. Apply eerst die SDD's, die een voorkomen hebben van een variabele die het laatste voorkomt in de volgorde. Dit is het omgekeerde van wat we doen in de vorige subsectie 5.2.2. Nu is nog de vraag of we best de vtree van links naar rechts doorlopen of van rechts naar links. De vorige sectie geeft voorkeur aan primes, dus dan zouden we van rechts naar links moeten doorlopen (aangezien het laatste voorkomen telt). De heuristiek voorgesteld in MINCE geeft automatisch prioriteit aan variabelen in de rechtertakken, omdat deze bij de OBDD altijd lager liggen in de volgorde (elke linkertak in de vtree bevat slechts één variabele).

Heuristiek 6.1 (Bottom-up variabelenvolgorde LR) *Leg een volgorde op aan alle variabelen door de vtree breedte-eerst van links naar rechts te doorlopen. Kies de twee SDD's die een variabele bevatten die als laatste voorkomt in de volgorde.*

Heuristiek 6.2 (Bottom-up variabelenvolgorde RL) *Leg een volgorde op aan alle variabelen door de vtree breedte-eerst van rechts naar links te doorlopen. Kies de twee SDD's die een variabele bevatten die als laatste voorkomt in de volgorde.*

De heuristieken Top-down variabelenvolgorde, Bottom-up variabelenvolgorde LR en Bottom-up variabelenvolgorde RL resulteren in een incrementele compilatie, zie definitie 7. Dit komt omdat de SDD $\gamma_i \circ \gamma_j$ met hoge kans de variabelen bevat die in γ_i of in γ_j voorkomen. Als γ_i en γ_j geprioriteerd worden door de heuristiek, zal $\gamma_i \circ \gamma_j$ ook geprioriteerd worden.

Definitie 7 (Incrementele compilatie) *Een compilatiesequentie $\gamma_1, \cdots, \gamma_M$ noemen we incrementeel, als*

1. $\gamma_{N+1} = Apply(\gamma_i, \gamma_j, \circ)$, *met* $i, j \leq N$
2. $\gamma_i = Apply(\gamma_{i-1}, \gamma_j)$, *met* $j < N + 1 < i$
3. *elke γ_i wordt maar één keer gebruikt als parameter van de Apply-operatie.*

5.3 Probabiliteit variabelenvoorkomens

De heuristieken 1: Vtree partitionering, 6: Top-down variabelenvolgorde, 6.1: Bottom-up variabelenvolgorde LR en 6.2: Bottom-up variabelenvolgorde RL hangen sterk af van de aanwezig variabelen in de SDD's. Als de SDD's alle variabelen bevatten, zullen heuristieken 6, 6.1 en 6.2 resulteren in dezelfde compilatiesequentie. Ook het effect van heuristiek 1 wordt kleiner naarmate er meer variabelen in de SDD's γ_i aanwezig zijn.

We berekenen de kans dat alle variabelen aanwezig zijn in een gegenereerde K-CNF over ν variabelen en met ratio r. De kans dat één clausule van K variabelen de variabele P niet bevat kan als volgt berekend worden:

$$Prob(P \text{ niet in clausule}) = \frac{\nu - 1}{\nu} \times \frac{\nu - 2}{\nu - 1} \times \cdots \times \frac{\nu - K}{\nu - K + 1} = \frac{\nu - K}{\nu} \tag{5.30}$$

De kans dat een K-CNF met $r \times \nu$ clausules de variabele P niet bevat berekenen we door de verschillende kansen te vermenigvuldigen:

$$Prob(\text{CNF bevat variabele } P \text{ niet}) = \left(\frac{\nu - K}{\nu}\right)^{r \times \nu} \tag{5.31}$$

De kans dat een K-CNF minstens één voorkomen heeft van de variabele P is de inverse van 5.31:

$$Prob(\text{CNF bevat variabele } P) = 1 - \left(\frac{\nu - K}{\nu}\right)^{r \times \nu} \tag{5.32}$$

ratio r	$Prob$(CNF bevat alle ν variabelen)
0.25	0.5476507846631062
0.5	0.7953801873840964
0.75	0.9074403883208138
1.0	0.9581307322850325
1.25	0.9810604696024041
1.5	0.9914327182857983
1.75	0.9961245968390107
2.0	0.9982469644210124

TABEL 5.1: Kans op voorkomen van elke variabele in een 3-CNF voor $\nu = 28$

De kans dat een K-CNF alle variabelen bevat is moeilijk exact te berekenen omdat het voorkomen van de ene variabele de kans beïnvloedt van het voorkomen van een andere variabele. We kunnen een schatting maken van de kans door onafhankelijkheid van de voorkomens van variabelen te veronderstellen:

$$Prob(\text{CNF bevat alle } \nu \text{ variabelen}) \approx \left(1 - \left(\frac{\nu - K}{\nu}\right)^{r \times \nu}\right)^{\nu} \tag{5.33}$$

In tabel 5.1 geven we de waarden voor $\nu = 28, K = 3$ en een aantal ratio's. We merken dat de kans al voor kleine r naar 1 convergeert. We kunnen dus verwachten dat heuristiek Vtree partitionering bijna geen effect gaat hebben voor $r > 1$. De heuristieken gebaseerd op een variabelenvolgorde zullen samenvallen voor $r > 1$.

5.4 Optimale heuristieken

De heuristieken die we hebben omschreven in deze sectie proberen telkens de twee SDD's γ_i en γ_j te applyen waarvoor $|\gamma_i \circ \gamma_j|$ zo klein mogelijk is. De heuristieken lossen het probleem op door in elke stap gulzig te zijn. Omdat de SDD zo een complexe voorstelling is, kan de grootte gaan van 1 (Waar of Onwaar) tot exponentieel groter dan γ_i. Hierdoor is het moeilijk om een heuristiek te vinden die de optimale sequentie van γ_i vindt. We kunnen wel een tegenvoorbeeld geven van wanneer onze heuristieken niet optimaal zijn. Voornamelijk geven we een tegenvoorbeeld voor heuristieken 4 en 5.

Neem probleeminstantie $\alpha = \alpha_1 \wedge \alpha_2 \wedge \alpha_3$ over de variabelen $\mathbf{X}$, met

$$\begin{aligned}
&\alpha_1 = (B \wedge \alpha_{11}) \vee (\neg B \wedge \alpha_{12}) = \gamma_1 \quad && |\alpha_{11}| \ll |\alpha_{12}| \\
&\alpha_2 = (B \wedge \alpha_{21}) \vee (\neg B \wedge \alpha_{22}) = \gamma_1 \quad && |\alpha_{21}| \ll |\alpha_{22}| \\
&\alpha_3 = B = \gamma_3
\end{aligned}$$

De optimale sequentie vereist tweemaal het gebruikt van γ_3 is

$$\begin{aligned}
&\gamma_4 = \gamma_1 \wedge \gamma_3 = B \wedge \alpha_{11} && O(|\alpha_1|) \\
&\gamma_5 = \gamma_2 \wedge \gamma_3 = B \wedge \alpha_{21} && O(|\alpha_2|) \\
&\gamma_6 = \gamma_4 \wedge \gamma_5 && O(|\alpha_{11}| * |\alpha_{21}|)
\end{aligned}$$

γ_6 big-O moet eigenlijk exponentieel zijn

Deze volledige sequentie heeft tijdscomplexiteit $O(|\alpha_{11}| * |\alpha_{21}| + |\alpha_1| + |\alpha_2|)$.

Uiteraard hangt de optimale sequentie af van de gekozen vtree. Als de vtree als volgt is: rootknoop $= v$, over variabelen $\mathbf{X}$, linkerkind v^l bevat de variabele B, rechterkind v^r bevat de variabelen $\mathbf{X} \cap B$; dan is de tijdscomplexiteit van $\gamma_4 \wedge \gamma_3 = O(|\alpha_{11}| * |\alpha_{21}|)$. In dit geval moet α_3 niet tweemaal gebruikt worden voor een optimale sequentie, dit toont nog maar eens de impact van een goede vtree.

Hoofdstuk 6

Experimenten

Om onze heuristieken te testen gebruiken we ze telkens tijdens de bottom-up compilatie van een probleeminstantie. Even ter herhaling, een probleeminstantie ziet er als volgt uit: $n = \bigcirc_{i=1}^{N} k_i$. Voor onze experimenten hebben we probleeminstanties nodig, deze genereren we willekeurig. We hebben verschillende parameters voor een probleeminstantie: het aantal $k_i = N$, het aantal variabelen in $n = \nu$ en de operatie (conjunctie of disjunctie). Voor elke probleeminstantie doen we het volgende:

nalezen tekst over dnf

1.1 We genereren eerst N verschillende CNF's, met y clausules en 3 variabelen per clausule, deze stellen de verschillende k_i voor.

1.2 In geval van conjunctie: We nemen de negatie van de CNF.

2 We compileren elke formule k_i naar een SDD γ_i met een vooraf bepaalde vtree v.

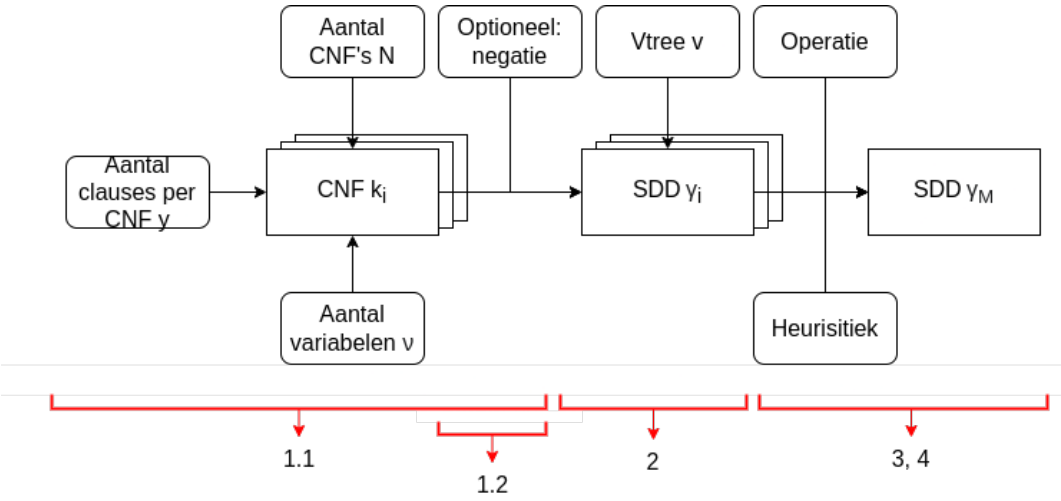
3 De bewerking $\gamma_M = \bigcirc_{i=1}^{N} \gamma_i$ wordt uitgevoerd met een bepaalde heuristiek.

4 De totale compilatietijd ct en de groottes van de tussenresultaten worden gemeten.

Een diagram van dit stappenplan is te zien in figuur 6.1 Voor ons experiment zijn er nog 2 extra (hyper)parameters, y en v. We bespreken eerst alle parameters: waarvoor ze dienen, welk effect ze hebben, en hoe ze van elkaar afhankelijk zijn.

Het aantal variabelen ν heeft een invloed op de grootte van de SDD's. Een elementaire bovengrens voor een SDD α met ν variabelen is $|\alpha| \leq 2^{\nu}$. We willen ν niet te groot kiezen zodat de SDD's nog steeds in het computergeheugen passen, maar ook niet te klein omdat de compilatietijden dan klein zijn en meer afhankelijk van externe factoren.

De keuze van het aantal clausules per CNF y is sterk gelinkt aan onze keuze van ν. Zoals gezien in sectie 4.3 moeten we opletten op de ratio $r = y/\nu$. Als de ratio over de drempelwaarde r^* gaat, is er een grote kans dat de CNF niet voldaan kan worden. Dit willen we niet want een CNF die niet voldaan kan worden komt overeen met de



FIGUUR 6.1: Stappenplan voor experimenten

logische formule $f = \bot$. Voor 3-CNF wordt de drempelwaarde geschat op $r^* = 4,2$ [CA96]. Naast de drempelwaarde voor onvoldaanbaarheid, is de ratio ook gelinkt aan de grootte van de compilatie. In zekere mate kunnen we de ratio beschouwen als parameter voor de complexiteit van een SDD: hoe complexer een SDD, hoe groter het is.

Gelinkt aan de ratio r is de operatie die we kiezen. Een conjunctie van 2 CNF's is nog steeds een CNF. In onze experimenten genereren we elke CNF met dezelde ratio, dus na een conjunctie zal die ratio verdubbelen. Daar komt nog eens bij dat de conjunctie $f \wedge \bot = \bot$. Deze gevallen meenemen in onze resultaten zou een vertekend beeld geven, omdat we niet altijd apply-operaties doen tussen twee niet-elementaire SDD's. Als we de conjunctie kiezen voor ons experiment, moeten we daarom als extra stap de negatie nemen van de CNF k_i voordat we deze omzetten naar de SDD γ_i.

Nog een belangrijke parameter is N. Als N klein is bijvoorbeeld zijn er weinig verschillende mogelijke sequenties voor een bottom-up compilatie en zullen de metrieken ct en g kleinere verschillen noteren tussen de heuristieken. De tijdscomplexiteit van de heuristieken is ook in functie van N. Zo is de tijdscomplexiteit van heuristiek 5 kwadratisch in N, terwijl dit lineair is voor de andere heuristieken.

Als laatste parameter hebben we de vtree v. De grootte van een SDD is, naast het aantal variabelen ν, sterk afhankelijk van de vtree. De manier waarop we CNF's genereren maakt dat elke variabele dezelfde kans op voorkomen heeft. We kunnen stellen dat elke variabele geacht wordt dezelfde invloed te hebben op de formule. Omwille daarvan zullen we voornamelijk met gebalanceerde vtrees werken, zodat we voor ν een hoge waarde kunnen kiezen.

In een gebalanceerde vtree ligt elke variabele, voor zover mogelijk, op dezelfde diepte. Hierdoor zullen heuristieken 6: Top-down variabelenvolgorde en 6.2: Bottom-

Heuristiek	Afkorting
Willekeurig	RAN
Vtree partitionering + Kleinste eerst	VP + KE
Vtree partitionering + Elementgewijze bovengrens	VP + EL
Top-down variabelenvolgorde	TD
Bottom-up variabelenvolgorde LR	BU-LR
Bottom-up variabelenvolgorde RL	BU-RL

TABEL 6.1: Heuristieken gebruikt in de experimenten

up variabelenvolgorde RL zo goed als samenvallen. Omwille hiervan maar ook omdat in de praktijk niet altijd gebalanceerde vtrees (zullen) worden gebruikt, testen we onze heuristieken ook in mindere mate uit op andere types vtrees.

De volgorde van de variabelen in de vtree speelt ook een rol. Wij kiezen ervoor om tussen alle experimenten telkens dezelfde willekeurige volgorde aan te houden.

Met al deze overwegingen in de gedachten, kiezen we $N = 20$, de ratio r laten we verschillende waarden in het interval $[0, 5]$ aannemen. Concreet doen we dit door ν constant te houden en $y = r * \nu$ te kiezen. Voor gebalanceerde vtrees kiezen we $\nu = 28$

andere ν keuzes bepalen/toelichten

De heuristieken die we besproken hebben zijn niet altijd logisch om te testen. Heuristiek 2: Clausule-lengte werkt enkel op CNF's, terwijl we heuristieken onderzoeken voor algemene rooted-DAG's. Heuristiek Vtree partitionering willen we enkel testen in combinatie met een interne heuristiek. Voor we de analyse van de experimenten beginnen, geven we voor elke (combinatie van) heuristiek(en) die we testen een logische afkorting.

6.1 SDD package en PySDD

Alle experimenten in deze thesis zijn uitgevoerd met behulp van de PySDD library [Mee23], een Pythonwrapper voor de SDD library geschreven door A. Darwiche in C [Dar11]. De library biedt een interface voor het creëren, manipuleren en evalueren van SDDs. Meer informatie en documentatie is beschikbaar op de officiële GitHub pagina [Mee23]. Voor deze implementatie werd een specifieke fork gebruikt gemaakt door V. Derkinderen. Deze fork bevat een extra functie $local\_vtree\_element\_count()$, die gebruikt wordt om het aantal elementen per vtreeknoop in een SDD te berekeken.

De tijdscomplexiteit van de heurisitieken is afhankelijk van deze library. De complexiteit van heuristiek Kleinste eerst is eigenlijk $O(max_{\gamma_i}(|\gamma_i|) \times N \times log(N))$, maar in de library wordt de grootte van SDD's tijdens runtime bijgehouden. Het opvragen van de grootte kan in constante tijd, waardoor de complexiteit $O(N \times log(N))$ wordt.

6.2 Resultaten

We voeren ons experiment telkens 2160 keer uit met als parameters $\nu = 28, r \in [0.5, 1, 1.5, \cdots, 4.5]$, en vtree v = gebalanceerd. Van elk experiment meten we de compilatietijd *ct* per heuristiek. We visualiseren deze resultaten met behulp van boxplots. Voor heuristiek 5: Elementgewijze bovengrens (VP + EL) geven we ook de compilatietijd gemeten zonder overhead (VP + EL 2). Dit doen we omdat we de overhead van deze heuristiek dominant is, maar we verwachten dat die nog te verkleinen is met de juiste programmeertechnieken. Door ook de compilatietijden zonder overhead te plotten tonen we het mogelijke potentieel van deze heuristiek. Van elk experiment meten we ook de groottes van de tussenresultaten per heuristiek. Om de groottes te visualiseren nemen we het gemiddelde van elk experiment per tussenresultaat per heuristiek. De resultaten zijn te vinden in de tabellen 6.2, 6.3, 6.4, 6.5 en 6.6.

Wat ons meteen opvalt is dat de drie heuristieken gebaseerd op een variabelenvolgorde gemiddeld resulteren in een tweemaal kleinere compilatietijd voor $r = 0.5$, en voor grotere r Bij de grotere ratio's $r \geq 3$ zien we dat de overhead door Bovengrens door elementen een grote invloed heeft op de totale compilatietijd. Als we de compilatietijd berekenen zonder deze overhead zien we dat het potentieel van deze heuristiek groot is. Het kan daarom nuttig zijn om in de toekomst de library aan te passen zodat het aantal elementen per vtreeknoop tijdens runtime worden bijgehouden, waardoor de overheadkost van $O((max_{\gamma_i}(|\gamma_i|) + \nu + log(N^2)) \times N^2)$ naar $O((\nu + log(N^2)) \times N^2)$ kan gereduceerd worden.

6.2.1 algemeen

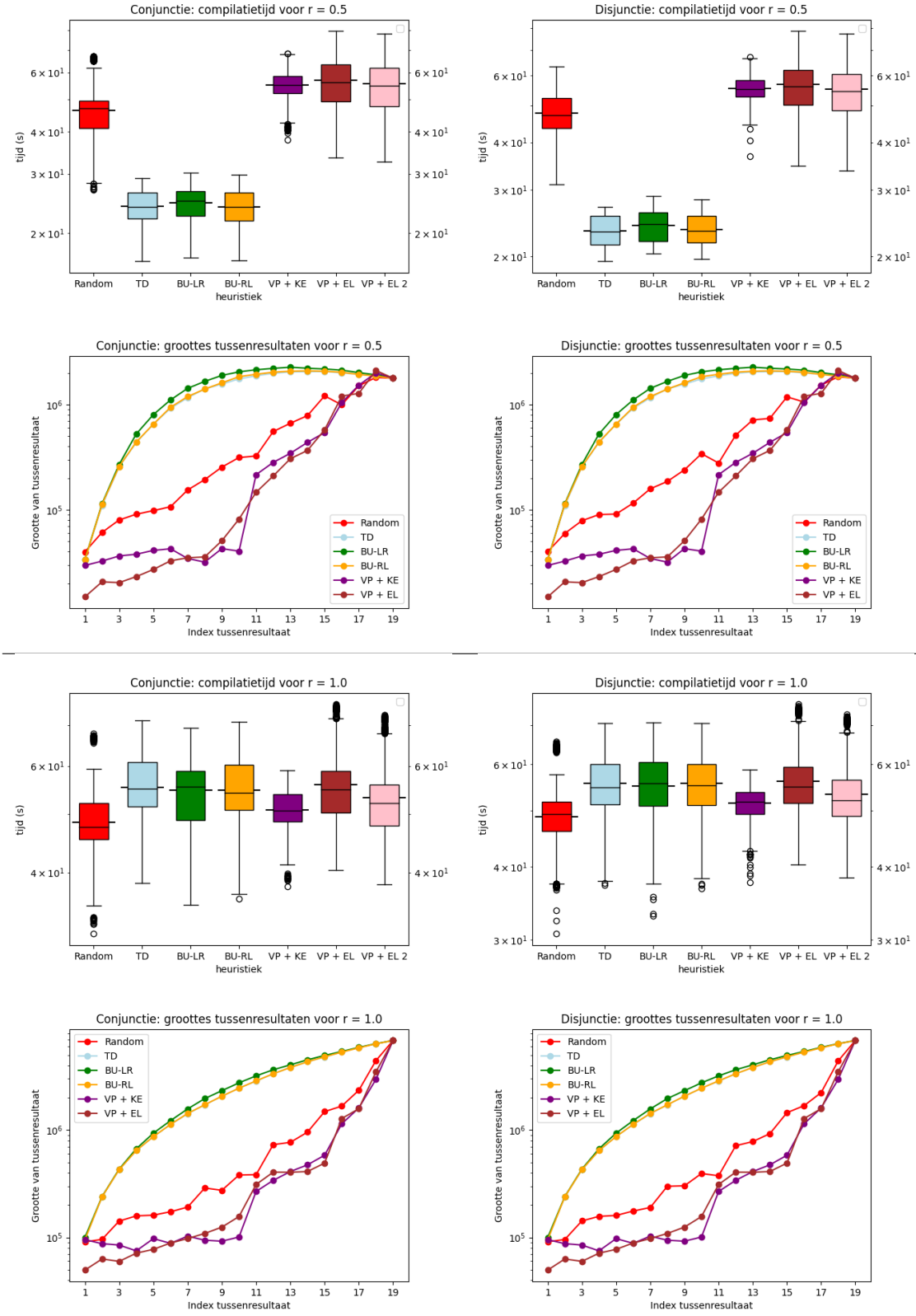
-(gemiddeld) grotes van finale sdd in termen van ratio -finale sdd is (gemiddeld) groter dan alle tussenresultaten voor veel ratios

6.2.2 Resultaten: variabelenvolgorde

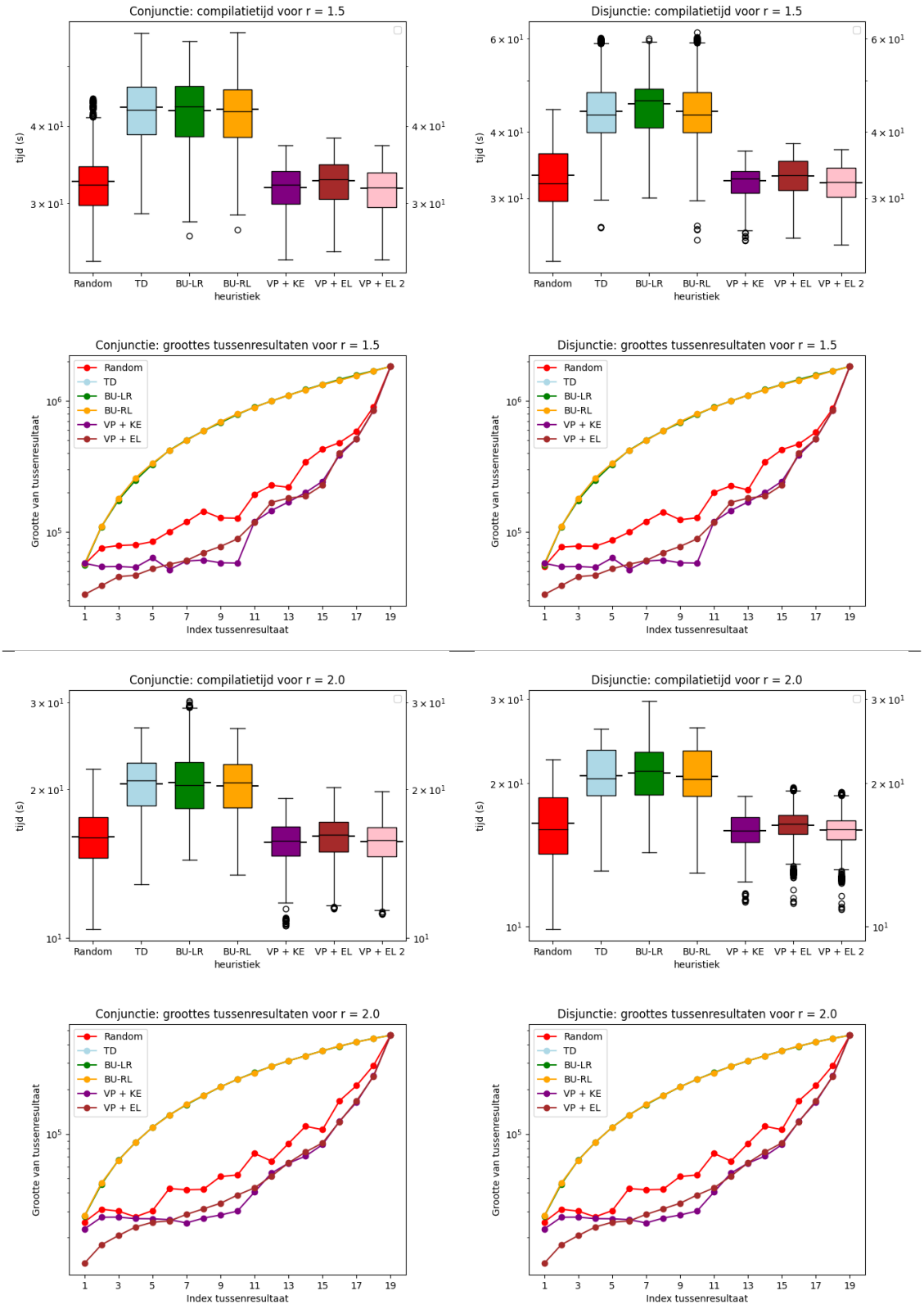
tijdsmetingen, -> bijna gelijk voor ivo$_r$$lenivo_lr- > incrementeleopbouw(depthplots)- >$
$sizeexperimententonen(voorlageratioenhogeratiouitlichten)- > eventueelvarCountstonenvoorlageen$

6.2.3 Resultaten: size upperbounds

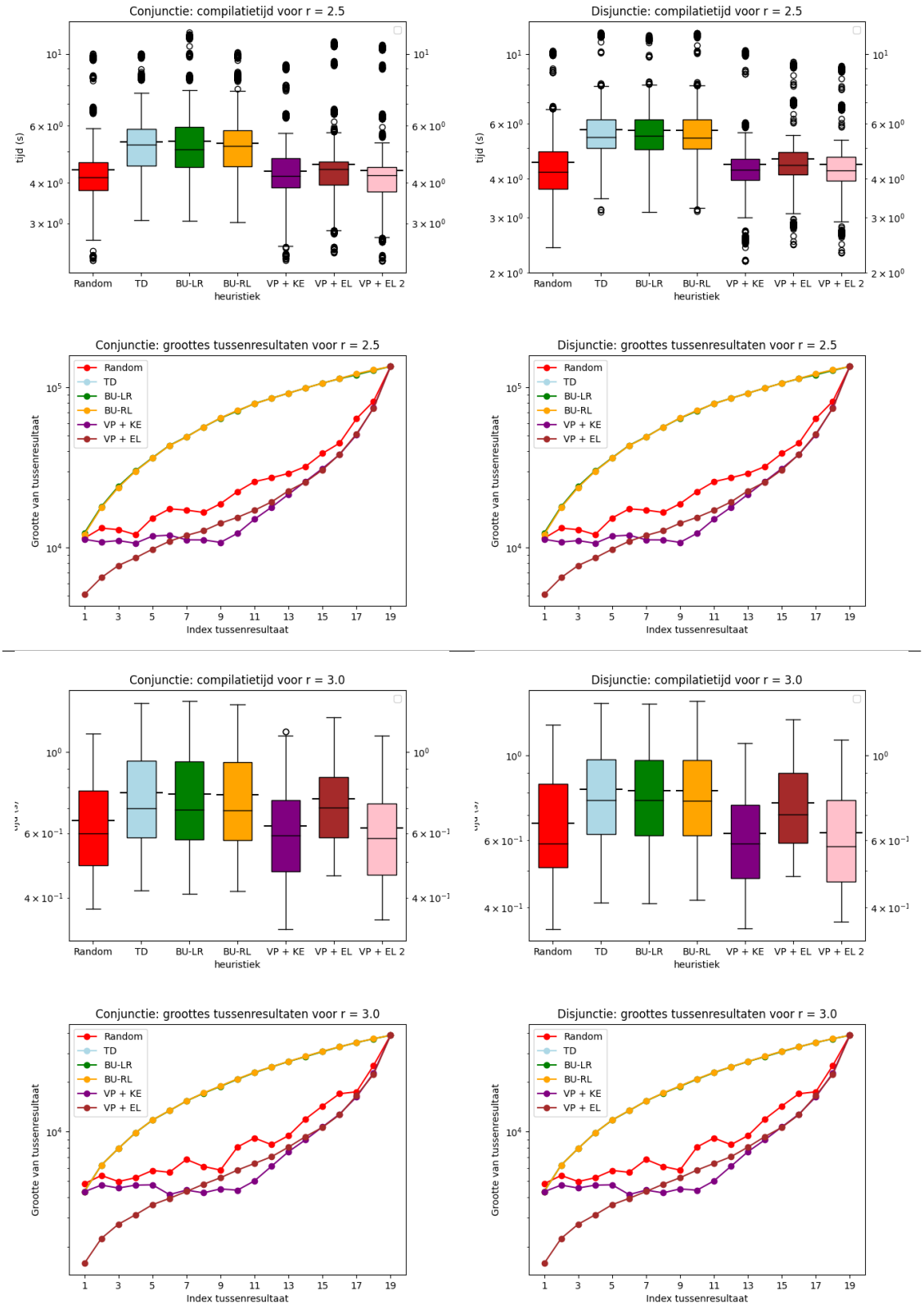
tijdsmetingen, -> heel gelijke opbouw (depth plots) -> size experimenten tonen -> el upperbound is goed in kleine sdds vinden -> eventueel varCounts tonen voor lage en hoge ratio



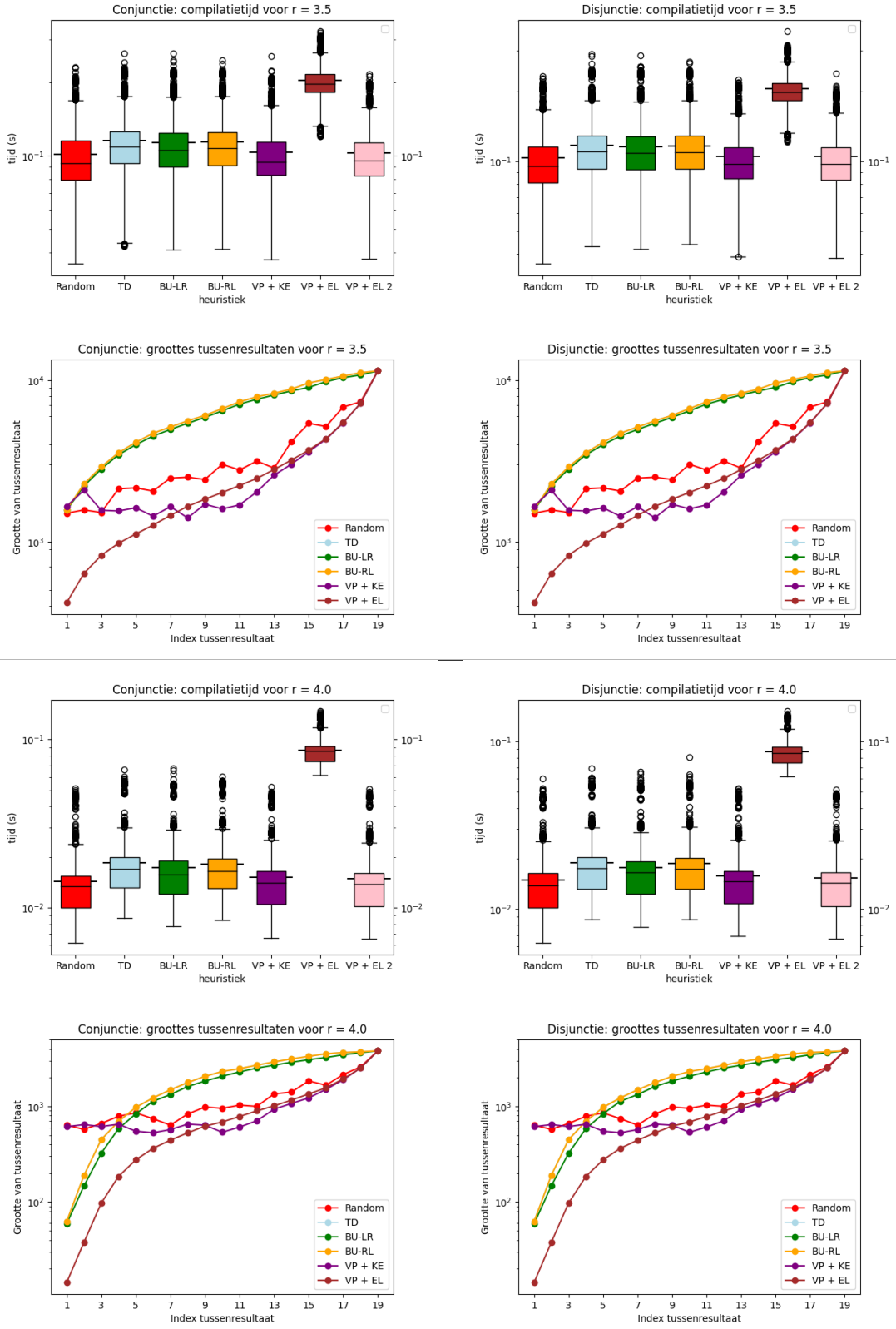
TABEL 6.2: Metrieken compilatiesequentie voor $r = 0.5, 1$



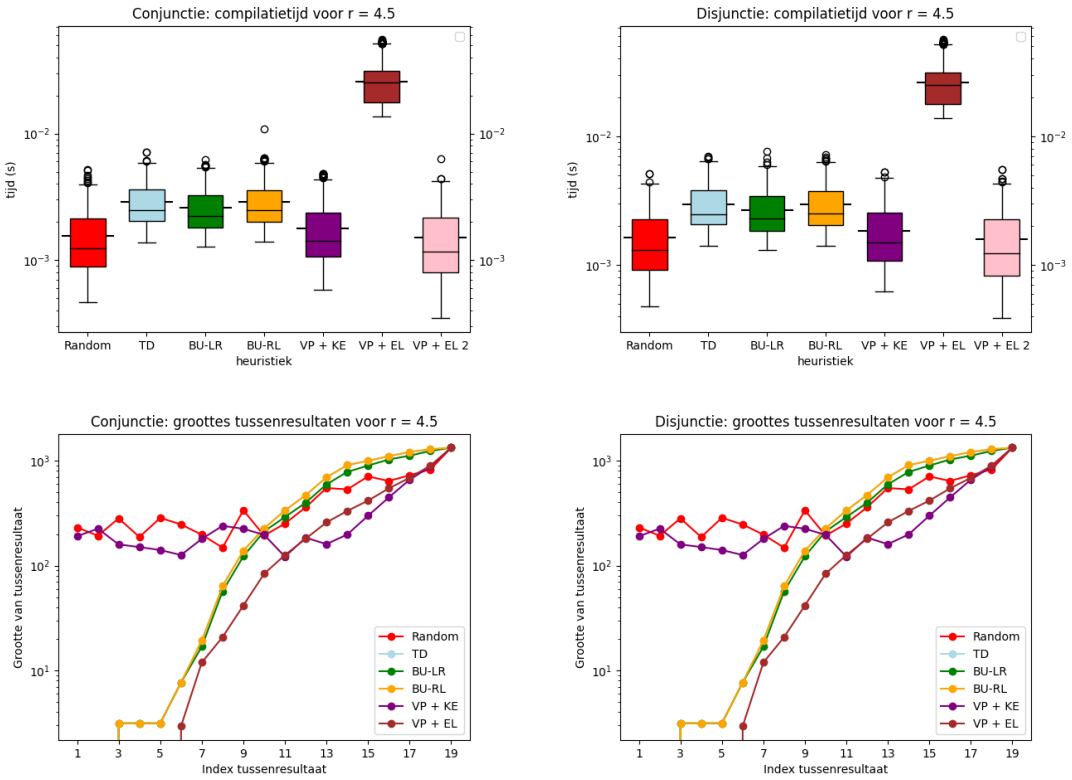
TABEL 6.3: Metrieken compilatiesequentie voor $r = 1.5, 2$



TABEL 6.4: Metrieken compilatiesequentie voor $r = 2.5, 3$



TABEL 6.5: Metrieken compilatiesequentie voor $r = 3.5, 4$



TABEL 6.6: Metrieken compilatiesequentie voor $r = 4.5$

Hoofdstuk 7

Combinatie van heuristieken

Hoofdstuk 8

Toekomstvisie

Bibliografie

[AMS03] Fadi A. Aloul, Igor L. Markov, and Karem A. Sakallah. FORCE: a fast and easy-to-implement variable-ordering heuristic. In *ACM Great Lakes Symposium on VLSI*, pages 116–119. ACM, 2003.

[AMS04] Fadi A. Aloul, Igor L. Markov, and Karem A. Sakallah. Mince: A static global variable-ordering heuristic for sat search and bdd manipulation. *J. Univers. Comput. Sci.*, 10:1562–1596, 2004.

[Bov16a] Simone Bova. Sdds are exponentially more succinct than obdds. *CoRR*, abs/1601.00501, 2016.

[Bov16b] Simone Bova. Sdds are exponentially more succinct than obdds. In *AAAI*, pages 929–935. AAAI Press, 2016.

[Bry86] Randal E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Trans. Computers*, 35(8):677–691, 1986.

[BW96] Beate Bollig and Ingo Wegener. Improving the variable ordering of obdds is np-complete. *IEEE Trans. Computers*, 45(9):993–1002, 1996.

[CA96] James M. Crawford and Larry D. Auton. Experimental results on the crossover point in random 3-satf. *Artificial Intelligence*, 81(1):31–57, 1996. Frontiers in Problem Solving: Phase Transitions and Complexity.

[CD08] Mark Chavira and Adnan Darwiche. On probabilistic inference by weighted model counting. *Artif. Intell.*, 172(6-7):772–799, 2008.

[CD13] Arthur Choi and Adnan Darwiche. Dynamic minimization of sentential decision diagrams. In *Proceedings of the Twenty-Seventh AAAI Conference on Artificial Intelligence*, AAAI'13, page 187–194. AAAI Press, 2013.

[CKD13] Arthur Choi, Doga Gizem Kisa, and Adnan Darwiche. Compiling probabilistic graphical models using sentential decision diagrams. In *European Conference on Symbolic and Quantitative Approaches to Reasoning and Uncertainty*, 2013.

[CKT91] Peter C. Cheeseman, Bob Kanefsky, and William M. Taylor. Where the really hard problems are. In *IJCAI*, pages 331–340. Morgan Kaufmann, 1991.

[Dar00] Adnan Darwiche. On the tractable counting of theory models and its application to belief revision and truth maintenance. *CoRR*, cs.AI/0003044, 2000.

[Dar11] Adnan Darwiche. Sdd: A new canonical representation of propositional knowledge bases. In *Proceedings of the Twenty-Second International Joint Conference on Artificial Intelligence (IJCAI)*, pages 819–826, 2011.

[dBD15] Guy Van den Broeck and Adnan Darwiche. On the role of canonicity in knowledge compilation. In *AAAI*, pages 1641–1648. AAAI Press, 2015.

[dCM21] Alexis de Colnet and Stefan Mengel. Lower bounds on intermediate results in bottom-up knowledge compilation. *ArXiv*, abs/2112.12430, 2021.

[Der23] Vincent Derkinderen. The role of counting in probabilistic reasoning, 2023.

[DKL+21] Meihua Dang, Pasha Khosravi, Yitao Liang, Antonio Vergari, and Guy Van den Broeck. Juice: A julia package for logic and probabilistic circuits. In *AAAI Conference on Artificial Intelligence*, 2021.

[DM11] Adnan Darwiche and Pierre Marquis. A knowledge compilation map. *CoRR*, abs/1106.1819, 2011.

[GRM20] Rahul Gupta, Subhajit Roy, and Kuldeep S. Meel. Phase transition behavior in knowledge compilation. In *CP*, volume 12333 of *Lecture Notes in Computer Science*, pages 358–374. Springer, 2020.

[GWY15] Jian Gao, Jianan Wang, and Minghao Yin. Experimental analyses on phase transitions in compiling satisfiability problems. *Sci. China Inf. Sci.*, 58(3):1–11, 2015.

[HD04] Jinbo Huang and Adnan Darwiche. Using DPLL for efficient OBDD construction. In *SAT (Selected Papers*, volume 3542 of *Lecture Notes in Computer Science*, pages 157–172. Springer, 2004.

[HKZDM19] Evert Heylen, Samuel Kolb, and Pedro Miguel Zuidberg Dos Martires. How to maximize exploitable structure in wmi, 2019.

[LBD+17] Anna L. D. Latour, Behrouz Babaki, Anton Dries, Angelika Kimmig, Guy Van den Broeck, and Siegfried Nijssen. Combining stochastic constraint optimization and probabilistic programming - from knowledge compilation to constraint solving. In *CP*, volume 10416 of *Lecture Notes in Computer Science*, pages 495–511. Springer, 2017.

[LBdB17] Yitao Liang, Jessa Bekker, and Guy Van den Broeck. Learning the structure of probabilistic sentential decision diagrams. In *UAI*. AUAI Press, 2017.

[Mee23] Wannes Meert. Pysdd. https://github.com/wannesm/PySDD, dec 2023. Accessed: 2024-05-21.

[MSL92] David G. Mitchell, Bart Selman, and Hector J. Levesque. Hard and easy distributions of SAT problems. In *AAAI*, pages 459–465. AAAI Press / The MIT Press, 1992.

[NW07] Nina Narodytska and Toby Walsh. Constraint and variable ordering heuristics for compiling configuration problems. In *IJCAI*, pages 149–154, 2007.

[OD18] Umut Oztok and Adnan Darwiche. An exhaustive DPLL algorithm for model counting. *J. Artif. Intell. Res.*, 62:1–32, 2018.

[PD08] Knot Pipatsrisawat and Adnan Darwiche. New compilation languages based on structured decomposability. In *AAAI*, pages 517–522. AAAI Press, 2008.

[PD10] Thammanit Pipatsrisawat and Adnan Darwiche. A lower bound on the size of decomposable negation normal form. In *AAAI*, pages 345–350. AAAI Press, 2010.

[RN20] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach (4th Edition)*. Pearson, 2020.

[Rud93] Richard Rudell. Dynamic variable ordering for ordered binary decision diagrams. In *ICCAD*, pages 42–47. IEEE Computer Society / ACM, 1993.

[VL20] Lieuwe Vinkhuijzen and Alfons Laarman. Symbolic model checking with sentential decision diagrams. In *SETTA*, volume 12153 of *Lecture Notes in Computer Science*, pages 124–142. Springer, 2020.

[VRdBR14] Jonas Vlasselaer, Joris Renkens, Guy Van den Broeck, and Luc De Raedt. Compiling probabilistic logic programs into sentential decision diagrams. 2014.

[XCD12] Yexiang Xue, Arthur Choi, and Adnan Darwiche. Basing decisions on sentences in decision diagrams. In *AAAI*, pages 842–849. AAAI Press, 2012.

Bijlage A

Engelse termen

kennisbasis	knowledge basis
conditionering	conditioning
vergeten (van singletons)	(singleton) forgetting
(begrensde) conjunctie	(bounded) conjunction
(begrensde) disjunctie	(bounded) disjunction
negatie	negation
consistentie	consistency
geldigheid	validity
clausale implicatie	clausal entailment
implicatie	implicant
equivalentie	equivalence
zin implicatie	sentential entailment
modeltellen	model counting
modelenumeratie	model enumeration
probabilistische inferentie	probabilistic reasoning
stochastische beperking-optimalisatie-problemen	stochastic constraint optimization problems
bondigheid	succintness
behandelbaarheid	tractability
gestructureerde opsplitsbaarheid	(structured) decomposability
(sterk) determinisme	(strong) determenism
deelverzameling	subset
superverzameling	superset
canoniciteit	canonicity
padbreedte	pathwidth
decompositie	decomposition
gecomprimeerd	compressed
gerichte acyclische grafe	Directed Acyclic Graph
gulzig	greedy
primaire implicaties	primary implicants
onder-/overbepaald	under-/overconstrained
symbolische modelverificatie	symbolic model checking