

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

© Copyright KU Leuven

Zonder voorafgaande schriftelijke toestemming van zowel de promotor als de auteur is overnemen, kopiëren, gebruiken of realiseren van deze uitgave of gedeelten ervan verboden. Voor aanvragen tot of informatie i.v.m. het overnemen en/of gebruik en/of realisatie van gedeelten uit deze publicatie, wend u tot het Departement Computerwetenschappen, Celestijnenlaan 200A bus 2402, B-3001 Leuven, +32-16-327700 of via e-mail info@cs.kuleuven.be.

Voorafgaande schriftelijke toestemming van de promotor is eveneens vereist voor het aanwenden van de in deze masterproef beschreven (originele) methoden, producten, schakelingen en programma's voor industrieel of commercieel nut en voor de inzending van deze publicatie ter deelname aan wetenschappelijke prijzen of wedstrijden.

Todo list

voorwoord toevoegen met bedanking	ii
stuk toevoegen over het gebruik van chatGTP in deze thesis, bijvoorbeeld om termen te vertalen	v
in de inleiding van elk stuk vertellen over wat elke subsectie gaat	v
betere vtree derin kijken	5
figuur aanpassen zodat alpha niet meer gebruikt wordt	9
tabellen fiksen met mooie ratio's	11
bibtex toevoegen	17
γ_6 big-O moet eigenlijk exponentieel zijn	27
figuur aanpassne zodat alpha niet meer gebruikt wordt	28
appendix maken	30
voeg appendix toe	33

Inhoudsopgave

Lijst van figuren en tabellen	iii
Lijst van afkortingen en symbolen	iv
Samenvatting	vi
1 Inleiding	1
2 SDD als nieuwe voorstelling van een kennisbasis	3
2.1 SDD als deelverzameling van d-DNNF en superverzameling van OBDD	3
2.2 Vtree	5
2.3 Apply-operatie	6
3 Probleemstelling	8
3.1 Bottom-up compilatie	8
3.2 Formeel	9
4 Eerder onderzoek	13
4.1 Vtree heuristieken	14
4.2 Compilers	15
4.3 Fase-transitie	16
4.4 Conjunctieve normaalvorm	17
4.5 SDD package en pySdd	18
5 Heuristieken	19
5.1 Bovengrens na apply	19
5.2 Heuristieken op basis van vtree	23
5.3 Optimale heuristieken	26
6 Experimenten	28
6.1 Resultaten: geheugengebruik	30
7 Combinatie van heuristieken	35
8 Toekomstvisie	36
Bibliografie	37
A Engelse termen	40

voorwoord
toevoegen
met bedan-
king

Lijst van figuren en tabellen

Lijst van figuren

2.1	OBDD, rechtslineaire SDD en gebalanceerde SDD voor de formule $f = (A \wedge B) \vee (C \wedge D)$	5
2.2	Gebalanceerde vtree voor 15 variabelen	6
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	10
5.1	Deel van SDD Ψ , met vtreenknoten v^p , v , v^l en v^r	21
6.1	experiment pijplijn	28

Lijst van tabellen

1.1	Bewerkingen en queries voor een kennisbasis	1
3.1	Variatie van compilatietijd ct voor $r = 1/3$ (a), $r = 4/3$ (b), $r = 7/3$ clausules (c) en $r = 10/3$ clausules (d).	12
6.1	Heuristieken gebruikt in de experimenten	30
6.2	Groottes tijdens compilatiesequentie voor 4 (a), 6 (b), 8 (c) en 10 (d) clausules.	31
6.3	Groottes tijdens compilatiesequentie voor 24 (a), 26 (b), 28 (c), 30 (d), 32 (e) en 34 (f) clausules.	32
6.4	Groottes tijdens compilatiesequentie voor 72 (a), 74 (b), 76 (c) en 78 (d) clausules.	33

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	Conjunctive Normal Form
SAT	Booleaans Satisfiability Probleem
ct	Totale compilatietijd
g	Maximale geheugengebruik
VP	Heuristiek: Vtree partitionering
KE	Heuristiek: Kleinste eerst
EL	Heuristiek: Element bovengrens
VO	Heuristiek: Vtree variabelenvolgorde
IVO-LR	Heuristiek: Inverse vtree variabelenvolgorde links naar rechts
IVO-RL	Heuristiek: Inverse vtree variabelenvolgorde rechts naar links

Symbolen

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

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

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 [Bov16] en de meer algemene BDD [SDD].

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 compilatie waarbij de tussenresultaten zo klein mogelijk zijn en zo snel mogelijk berekend kunnen worden, dit minimaliseert het geheugengebruik en de totale compilatietijd. 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 en het geheugengebruik 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 \perp voor *Onwaar*.

Introduceer misschien kort eerst propositionele logica:
variable, literal, negatie, conjunctie, disjunctie, \top , \bot , clause, CNF, conditionering $f|x, \dots$

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

definities decomposability, determinism en NNF kan dan ineens ook in die subsectie ?

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 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].

2.1.1 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.

[Zet SDD paper citation bij de definitie](#)

Definitie 1 ((X, Y)-decompositie) Gegeven een *booleaanse* functie f over variabelen \mathbf{X}, \mathbf{Y} , met $\mathbf{X} \cap \mathbf{Y} = \emptyset$. Als $f = (p_1(\mathbf{X}) \wedge s_1(\mathbf{Y})) \vee \dots \vee (p_n(\mathbf{X}) \wedge s_n(\mathbf{Y}))$, dan is $\{(p_1, s_1), \dots, (p_n, s_n)\}$ een (\mathbf{X}, \mathbf{Y}) -decompositie van f .

Als ook nog geldt dat $p_i \wedge p_j = \perp$ 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.

[en hier ook](#)

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

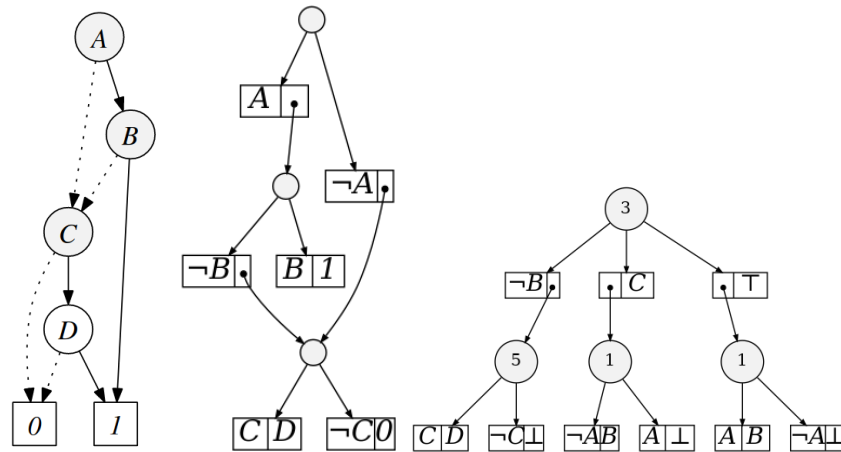
Een partitie heeft als eigenschappen:

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 \mathbf{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 \mathbf{X} -partitie heeft [Dar11]. Omwille van deze eigenschap is een gecomprimeerde SDD canoniek.

Eigenschap 1 (X-partitie van $f \circ g$) Neem \circ een booleaanse operator, en $\{(p_1, s_1), \dots, (p_n, s_n)\}$ en $\{(q_1, r_1), \dots, (q_m, r_m)\}$ \mathbf{X} -partities van respectievelijk $f(\mathbf{X}, \mathbf{Y})$ en $g(\mathbf{X}, \mathbf{Y})$, dan is $\{(p_i \wedge q_j, s_i \circ r_j) \mid p_i \wedge q_j \neq \perp\}$ een \mathbf{X} -partitie van $f \circ g$.

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



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

2.2 Vtree

In een OBDD wordt gesplitst op variabelen. De volgorde waarop wordt gesplitst, is gespecificeerd 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 \mathbf{X} is een binaire boom, van welke de bladeren een één-op-één correspondentie hebben met de variabelen in \mathbf{X} . In figuur 2.2 kunnen we een voorbeeld van een vtree zien. We refereren naar een knoop in de vtree of de deelboom geworteld in die knoop dm.v. 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 vtrees: elk linkerkind is een bladknoop.
- Linkslineaire vtrees: elk rechterkind is een bladknoop.
- Gebalanceerde vtrees: elke bladknoop zit op dezelfde diepte.
- Willekeurige vtrees.

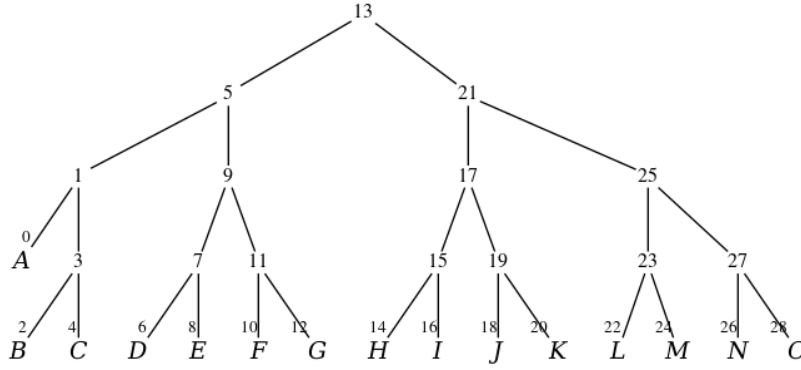
"is een OBDD", drop de "gelijk aan"

Een SDD genormaliseerd voor een rechtslineaire vtree is gelijk aan 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.

Boolean is altijd met hoofdletter B geschreven

betere vtree
derin kijken



FIGUUR 2.2: Gebalanceerde vtree voor 15 variabelen

[Ref naar SDD paper](#)

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

1. $\alpha = \perp$ of $\alpha = \top$, $\langle \perp \rangle = \text{Onwaar}$, $\langle \top \rangle = \text{Waar}$.
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. $\alpha = \{(p_1, s_1), \dots, (p_n, s_n)\}$, 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 $\langle p_1 \rangle, \dots, \langle p_n \rangle$ een partitie vormen. $\langle \alpha \rangle = \bigvee_{i=1}^n \langle p_i \rangle \wedge \langle s_i \rangle$.

Een belangrijke type SDD's gegeven in de originele paper over SDD's zijn gecomprimeerde SDD's. Een SDD is gecomprimeerd a.s.a. voor elke decompositie $\{(p_1, s_1), \dots, (p_n, s_n)\}$ in de SDD geldt dat $s_i \neq s_j$ voor $i \neq j$.

2.3 Apply-operatie

Een apply-operatie heeft als input twee kennisbasissen en een booleanse operatie, en als output de combinatie van die twee kennisbasissen 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 booleanse operatie op twee SDD's uitvoert in polytijd m.b.v. de eerder geziene eigenschap van **X**-partities. Die eigenschap vereist wel dat beide SDD's genormaliseerd zijn voor dezelfde vtree. In algoritme 1 wordt pseudocode gegeven voor zo een apply-operatie, deze is overgenomen uit de [originele SDD paper](#). 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 diezelfde [paper](#) 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 vtreesknoop, \circ is een booleaanse operator

Cache(., ., .) = **nil** bij start.

Expand(γ) geeft $\{(\top, \top)\}$ terug als $\gamma = \top$; $\{(\top, \perp)\}$ als $\gamma = \perp$; anders γ .

UniqueD(γ) geeft \top terug als $\gamma = \{(\top, \top)\}$; \perp als $\gamma = \{(\top, \perp)\}$; anders de unieke SDD met elementen γ .

```

1: if  $\alpha$  en  $\beta$  zijn constanten of variabelen then
2:   return  $\alpha \circ \beta$ 
3: else if Cache( $\alpha, \beta, \circ$ )  $\neq$  nil then
4:   return Cache( $\alpha, \beta, \circ$ )
5: else
6:    $\gamma \leftarrow \{\}$ 
7:   for all elementen  $(p_i, s_i)$  in Expand( $\alpha$ ) do
8:     for all elementen  $(q_j, r_j)$  in Expand( $\alpha$ ) do
9:        $p \leftarrow \mathbf{Apply}(p_i, q_j, \wedge)$ 
10:      if  $p$  is consistent then
11:         $s \leftarrow \mathbf{Apply}(s_i, r_j, \circ)$ 
12:        voeg element  $(p, s)$  toe aan  $\gamma$ 
13:      end if
14:    end for
15:  end for
16: end if
17: return Cache( $\alpha, \beta, \circ$ )  $\leftarrow$  UniqueD( $\gamma$ )

```

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

Eigenschap 2 (Negatie van een SDD) *Neem $\alpha = \{(p_1, s_1), \dots, (p_n, s_n)\}$ een SDD, dan is $\neg\alpha = \{(p_1, \neg s_1), \dots, (p_n, \neg s_n)\}$ 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, omvangslijmieten, gespecialiseerde compilers, etc. In deze paper 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 Gerichtte 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 deel formule 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  $\leftarrow$  map(Bottom-up compilatie, knoop.kinderen)
8: while kindCompilaties bevat meer dan één element do
9:    $\gamma_i, \gamma_j \leftarrow$  kindCompilaties.pop(), kindCompilaties.pop()
10:   $\gamma_k \leftarrow$  Apply( $\gamma_i, \gamma_j$ , knoop.operatie)
11:  kindCompilaties  $\leftarrow$  (kindCompilaties  $\cup \gamma_k$ ) /  $\{\gamma_i, \gamma_j\}$ 
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 niet lang duurt om te berekenen en zo weinig mogelijk extra geheugen gebruikt.

figuur aan-
passen zo-
dat alpha
niet meer
gebruikt
wordt

3.2 Formeel

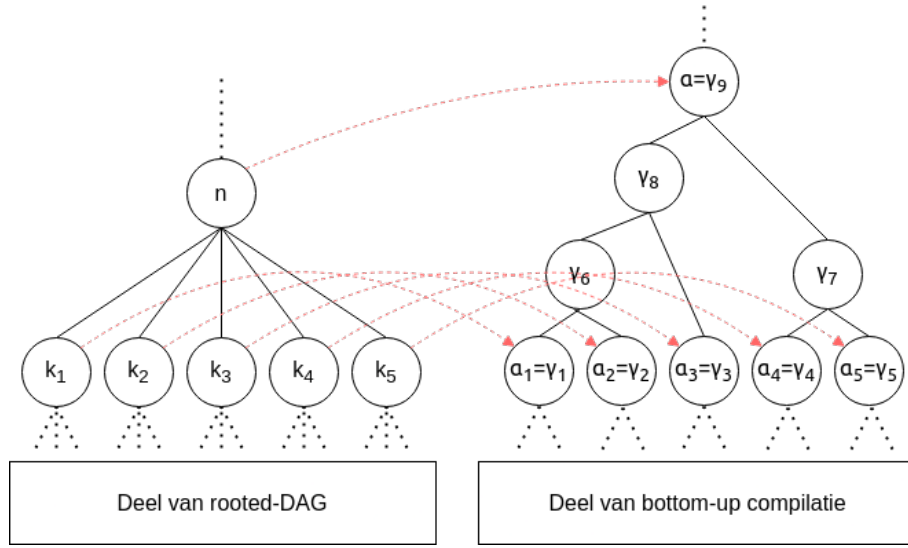
Het probleem kunnen we formeel als volgt stellen.

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 $\gamma_1, \dots, \gamma_M$, met $\gamma_M \equiv n$ en voor elke $i \in [1, M]$:

1. $\gamma_i \equiv k_j$, of
2. $\gamma_i = \text{Apply}(\gamma_j, \gamma_k, \circ)$ met γ_j en γ_k hebben dezelfde vtree, of met $j, k < i$?
3. $\gamma_i \equiv \gamma_j$ met $j < i$ en de vtrees van γ_i en γ_j verschillen.

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 compilatie (meer

Misschien beter om de eerste $N - y_i$ gelijk te stellen aan k_i ?
Dan zijn alle tussenresultaten $[N+1 \dots M-1]$



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). Als we enkel de eerste twee mogelijkheden gebruiken, is $M = 2 \times N - 1$, er worden $N - 2$ tussenresultaten berekend voordat γ_M berekend kan worden. Er zijn twee zaken die we willen minimaliseren tijdens de compilatie van n . Ten eerste het geheugengebruik g :

$$g = \max_{1 \leq i \leq M} \{|\gamma_i|\} \quad (3.1)$$

en ten tweede de totale compilatietijd ct :

$$ct = \sum_{i=1}^M t(\gamma_i) \quad (3.2)$$

Dan kan je de input k's excluden: $\text{sum}_{i=1}^M \{i = N+1\} \wedge M$

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

Wat is het formele probleem exact? I.e., kies geheugengebruik of compilatietijd als primaire objective, of leg uit dat compilatietijd ook leidt naar lager geheugengebruik gezien grotere tussenrepresentaties ook tot grotere ct leiden.

3.2.1 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.

$$\gamma_1 = (A \wedge \top) \vee (\neg A \wedge B) \equiv (A \vee B) \quad (3.3)$$

$$\gamma_2 = B \quad (3.4)$$

$$\gamma_3 = (B \wedge C) \vee (\neg B \wedge \top) \equiv (\neg B \vee C) \quad (3.5)$$

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 $\gamma_2 \wedge \gamma_3$.

$$\gamma_4' = \gamma_1 \wedge \gamma_2 = B \quad (3.6)$$

$$\gamma_4'' = \gamma_1 \wedge \gamma_3 = (\neg A \wedge ((B \wedge C) \vee (\neg B \wedge \perp))) \vee (A \wedge ((B \wedge C) \vee (\neg B \wedge \top))) \quad (3.7)$$

$$\gamma_4''' = \gamma_2 \wedge \gamma_3 = (B \wedge C) \vee (\neg B \wedge \perp) \quad (3.8)$$

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.

$$\gamma_5 = (B \wedge C) \vee (\neg B \wedge \perp) \equiv f \quad (3.9)$$

In dit voorbeeld zien we dat γ_4'' duidelijk groter is dan γ_4' , γ_4''' en zelfs γ_5 . In deze thesis zullen we heuristieken aanbieden die grote tussenresultaten zoals γ_4'' proberen te vermijden.

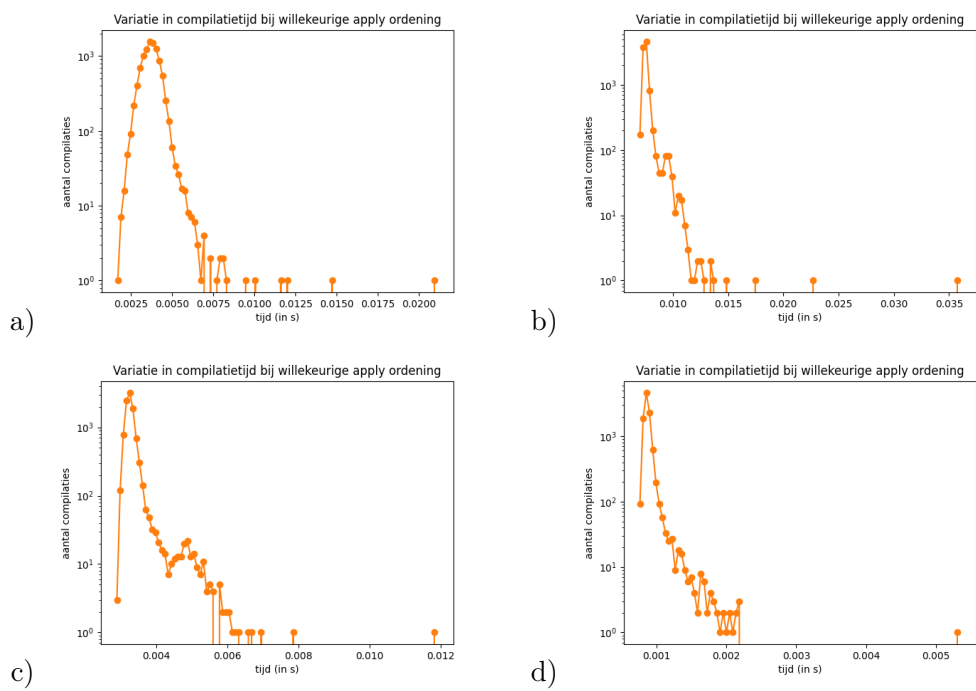
3.2.2 In de praktijk

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 het probleem $n = \bigcirc_{i=1}^N k_i$, daarna proberen we verschillende willekeurige sequentie's van $\gamma_1, \dots, \gamma_M$ en meten we de compilatietijd ct en geheugengebruik g .

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 x en clauses y . We compileren deze telkens met een willekeurige volgorde, we krijgen dus willekeurige sequentie's van $\gamma_1, \dots, \gamma_M$. Van elke compilatie meten we de tijd ct . Zo krijgen we een gauss-curve van goede tot slechte sequenties.

We geven in tabel 3.1 de variatie weer, en we geven ook de ratio $r = y/x$ 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 ra-
tio's



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

Hoofdstuk 4

Eerder 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-transitie bij de grootte van kenniscompilaties. Een aanzienlijk deel van de hier aangehaalde bronnen beperken zich tot formules in **Conjunctieve Normaalvorm (CNF)**. Een CNF is een logische formule die bestaat uit de conjunctie van clauses, en elke clause is een disjunctie van variabelen. Dit is een veelgebruikte voorstelling voor logische formules. 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: Volledig onafhankelijke componenten vinden zal meestal niet mogelijk zijn, maar het probleem opsplitsen in verschillende componenten die **intra meer afhankelijk van elkaar zijn dan inter**, kan ook gunstig zijn.
- **Variabelen met minst aantal mogelijke waarden** eerst bepalen: **Welke variabelen in een vtree hebben de minst mogelijke waarden**, welk effect kunnen ze hebben op de grootte van een SDD genormaliseerd voor die vtree?
- **Variabelen die in veel beperkingen een rol spelen** eerst bepalen: Welke variabelen spelen een belangrijke rol in een kennisbasis, hoe vertaalt die rol zich in een SDD?

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 modelverificatie wordt gegeven in [VL20]. Twee gelijkaardige heuristieken worden gegeven in [LBdB17] en [HKZDM19]. De vrees worden in deze twee laatste gevallen gebruikt voor de structuur van de PSSD 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 compilatie-algoritme met dynamische minimalisatie is voorgesteld in [CD13]. Er wordt een techniek aangehaald waarmee de ruimte van vrees kan doorzocht worden. Met die techniek wordt tijdens de compilatie, onder bepaalde voorwaarden, naar een betere vtree gezocht. 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 aangepast. Maar er is nog een andere SDD γ_j die variabelen met γ_i deelt. De vtree waarvoor γ_j is genormaliseerd, moet dezelfde deeltree 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.

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. De optimalisatie is "nevenkostenvrij".

De situatie waarin we de vtree nevenkostenvrij of bijna nevenkostenvrij kunnen optimaliseren komt normaalgezien niet veel voor in een rooted-DAG. We kunnen dan slechts beperkte delen van de vtree optimaliseren tijdens de compilatie. Waar deze dynamische techniek wel effectief is, is voor de bottom-up compilatie van CNF's. In een CNF-compiler worden enkel clauses bij elkaar gevoegd om de compilatie te verkrijgen. De kans dat we (bijna alle) clauses 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.

Ondanks de in deze sectie aangehaalde technieken om de vtree te optimaliseren, is er voor zover wij weten (nog) geen algemene techniek om statisch/dynamisch een vtree te bepalen. De dynamische minimalisatie gebruikt voor CNF's is gemakkelijk te implementeren voor algemene rooted-DAG's, maar het effect hiervan is nog niet bestudeerd. 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 gebalanceerde vtree 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 clauses.

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

Zelfs als de clauses gepartitioneerd worden, kan het zijn dat er meer dan 2 clauses gelinkt worden aan dezelfde vtreenoep v . In dat geval moet er nog altijd een keuze gemaakt worden welke clauses eerst applied worden. De compiler zal in dit geval een tweede, interne, heuristiek raadplegen. In de paper wordt een tweede

heuristisch gegeven, maar deze werkt enkel op CNF's. Verder in de thesis zullen we deze vervangen met onze eigen heuristieken.

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

Heuristiek 1 wordt ook gebruikt in JUICE, een Julia package voor logische en probabilistische circuits [DKL⁺21]. Hiervoor wordt de heuristiek uitgebreid 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 ontwikkeld. 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. Voor een algemene heuristiek zijn hier **helaas niet veel nuttige zaken uit te leren**.

Het probleem van grote tussenresultaten 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 clauses 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 clauses op te splitsen. Een tweede voorstel, voor CNF's, legt een volgorde op aan de clauses gebaseerd op de variabele van elke clause, die het laatste in de variabelenvolgorde voorkomt. De clauses 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. K -CNF is een CNF waarbij elke clause uit K variabelen bestaat.

De fase-transitie conjunctuur is gelinkt aan de verhouding r van het aantal clauses y ten opzichte van het aantal variabelen x , $r = y/x$. 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 dus belangrijk dat we rekening houden met de verhouding r van deze CNF's. 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.

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 x voor het aantal variabelen. Elke CNF wordt gegenereerd door y clausules uniform en onafhankelijk te genereren. Elke clause wordt gegenereerd door willekeurig k unieke variabelen te selecteren uit x 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 , x 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

4.5 SDD package en pySdd

Alle experimenten in deze thesis zijn uitgevoerd met behulp van de PySDD library, een wrapper voor de SDD library geschreven door A. Darwiche in C [Dar11]. Complexiteit van de heuristieken is afhankelijk van deze library.

Hoofdstuk 5

Heuristieken

De heuristieken die we in deze sectie beschrijven dienen om telkens twee SDD's γ_i en γ_j te zoeken in een lijst van SDD's, die applied moeten worden. Dit komt overeen met de tweede optie in definitie 4. Het zoeken wordt herhaald tot alle SDD's zijn samengevoegd tot één SDD. In Algoritme 2 moet de heuristiek als extra parameter worden meegegeven, en moet regel 9 worden aangepast naar $\gamma_i, \gamma_j \leftarrow \text{Heuristiek}(\text{kindCompilaties})$ om de heuristiek te gebruiken. Heuristiek 1 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 x , het aantal variabelen aanwezig in de probleeminstantie.

5.1 Bovengrens na apply

Is grootte van SDD reeds gedefinieerd?
aantal decision knopen?

Een eerste heuristiek die we proberen is gebaseerd op de maximale grootte van $|\gamma_i \circ \gamma_j|$, en die apply-operatie te doen, die de kleinste maximale grootte heeft. uit eigenschap 1 volgt dat voor de polytijd apply-operatie geldt:

$$|\gamma_i \circ \gamma_j| \leq |\gamma_i| * |\gamma_j| \quad (5.1)$$

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. De complexiteit van deze heuristiek is $O(N \times \log(N))$.

Heuristiek 3 (Kleinste eerst) *Kies de twee SDD's met de kleinste grootte.*

Heuristiek 3 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:

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

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

$$\#el(\alpha \circ \beta, v) \leq \#el(\alpha, v) \times \#el(\beta, v) \quad (5.3)$$

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

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

5.1.2 Extra bovengrens dmv. primes

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

Neem $\{(p_1(\mathbf{X}), s_1(\mathbf{Y})), \dots, (p_n(\mathbf{X}), s_n(\mathbf{Y}))\}$ en $\{(q_1(\mathbf{X}), r_1(\mathbf{Y})), \dots, (q_m(\mathbf{X}), r_m(\mathbf{Y}))\}$ de elementen van SDD's $\alpha(\mathbf{X}, \mathbf{Y})$ en $\beta(\mathbf{X}, \mathbf{Y})$. α en β zijn genormaliseerd voor vtreesknoop 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}_α en elke q_j een SDD over de variabelen \mathbf{X}_β . De primes van een SDD moeten een partitie vormen. Een partitie over de variabelen \mathbf{X} kan maximaal uit $2^{|\mathbf{X}|}$ primes bestaan.

$$\#el(\alpha, v) \leq 2^{|\mathbf{X}_\alpha|} \quad \#el(\beta, v) \leq 2^{|\mathbf{X}_\beta|} \quad (5.5)$$

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

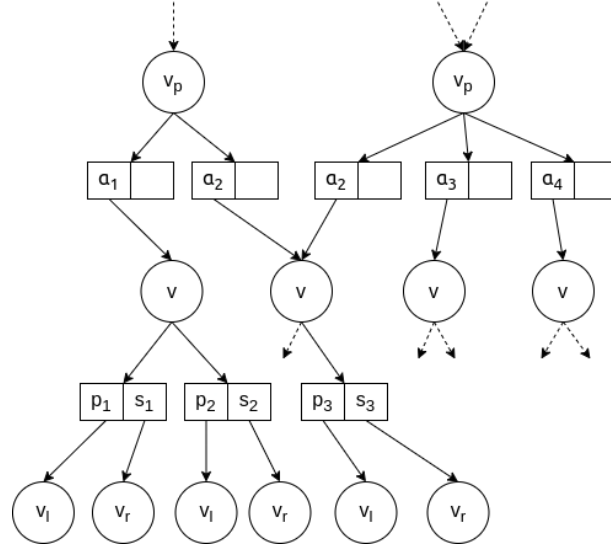
$$\alpha \circ \beta = \{(p_i \wedge q_j, s_i \circ r_j) | p_i \wedge q_j \neq \text{Onwaar}\} \quad (5.6)$$

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 vtreesknoop v voor $\alpha \circ \beta$ is

$$\#el(\alpha \circ \beta, v) \leq 2^{|\mathbf{X}_\alpha \cup \mathbf{X}_\beta|} \quad (5.7)$$

Vergelijking 5.7 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 vtreesknoop v . Elk element in Ψ onder vtreesknoop 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 .

$$\#el(\Psi, v) \leq \#el(\Psi, v^p) \times 2^{|\mathbf{X}_\alpha|} \quad \#el(\Omega, v) \leq \#el(\Omega, v^p) \times 2^{|\mathbf{X}_\beta|} \quad (5.8)$$

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

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 vtreenknoop v^p van Ψ met elk element onder v^p van Ω geapplied worden. Voor elk van die apply-operaties geldt vergelijking 5.7. We kunnen een vergelijking opschrijven voor het aantal elementen in $\Psi \circ \Omega$ onder v :

$$\#el(\Psi \circ \Omega, v) \leq \#el(\Psi \circ \Omega, v^p) \times 2^{|X_\alpha \cup X_\beta|} \quad (5.9)$$

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 5.8 invoegen in vergelijking 5.3 krijgen we

$$\#el(\Psi \circ \Omega, v) \leq \#el(\Psi, v^p) \times \#el(\Omega, v^p) \times 2^{|X_\alpha| + |X_\beta|}. \quad (5.10)$$

De bovengrens in 5.9 kan kleiner zijn dan bovengrens in 5.10 als $\mathbf{X}_\alpha \cap \mathbf{X}_\beta \neq \emptyset$. Hoe kleiner het symmetrisch verschil $\mathbf{X}_\alpha \Delta \mathbf{X}_\beta$ is, hoe beter vergelijking 5.9 de bovengrens gaat beïnvloeden.

5.1.3 Extra bovengrens dmv. subs

Er kan een tweede soortgelijke bovengrens gevonden worden in termen van de subs. Neem opnieuw $\{(p_1(\mathbf{X}), s_1(\mathbf{Y})), \dots (p_n(\mathbf{X}), s_n(\mathbf{Y}))\}$ en $\{(q_1(\mathbf{X}), r_1(\mathbf{Y})), \dots (q_m(\mathbf{X}), r_m(\mathbf{Y}))\}$ 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}_α en elke r_j een SDD over \mathbf{Y}_β 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.

$$\#el(\alpha, v) \leq 2^{(2^{|\mathbf{Y}_\alpha|})} \quad \#el(\beta, v) \leq 2^{(2^{|\mathbf{Y}_\beta|})} \quad (5.11)$$

Op dezelfde wijze als bij de primes kunnen we een bovengrens bekomen voor het aantal elementen van $\alpha \circ \beta$, zoals in 5.7, maar dan voor \mathbf{Y}_α en \mathbf{Y}_β en met een extra macht.

$$\#el(\alpha \circ \beta, v) \leq 2^{(2^{|\mathbf{Y}_\alpha \cup \mathbf{Y}_\beta|})} \quad (5.12)$$

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

$$\#el(\Psi \circ \Omega, v) \leq \#el(\Psi, v^p) \times \#el(\Omega, v^p) \times 2^{(2^{|\mathbf{Y}_\alpha \cup \mathbf{Y}_\beta|})} \quad (5.13)$$

Deze bovengrens is in de praktijk minder nuttig dan 5.9, omdat de bovengrens een macht extra heeft in de vergelijking.

De drie bovengrenzen in 5.3, 5.9 en 5.13 geven ons de vergelijking

$$\#el(\alpha \circ \beta, v) \leq \text{Min} \left\{ \begin{array}{l} \#el(\alpha, v) \times \#el(\beta, v) \\ \#el(\alpha \circ \beta, v^p) \times 2^{|\mathbf{X}_\alpha \cup \mathbf{X}_\beta|} \\ \#el(\alpha \circ \beta, v^p) \times 2^{(2^{|\mathbf{Y}_\alpha \cup \mathbf{Y}_\beta|})} \end{array} \right\} \quad (5.14)$$

Met v^p de ouderknoop van v , \mathbf{X}_α en \mathbf{X}_β de set van variabelen in α en β onder vtreeknoop v^l en \mathbf{Y}_α en \mathbf{Y}_β 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 4 (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 3 geldt de gegeven bovengrens enkel voor de polytijd apply-operatie. In de gecomprimeerde versie is de bovengrens in vergelijking 5.14 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 $O(x)$, met x 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 $\{(\gamma_i, \top), (\neg\gamma_i, \perp)\}$

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 5.14 gebruiken om de eerste schatting bij te stellen, de tijdscomplexiteit hiervan is $O(x)$.

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 $O((\max_{\gamma_i}(|\gamma_i|) + x + \log(N^2)) \times N^2)$. In de meeste gevallen zal deze functie gedomineerd worden door $(\max_{\gamma_i}(|\gamma_i|))$. Het kan daarom nuttig zijn om het aantal elementen per vtreeknoop tijdens de runtime bij te houden. Helaas is deze functionaliteit niet ingebouwd in de SDD library en was er geen tijd om de library uit te breiden.

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 x . De tijdscomplexiteit van deze heuristiek is bijgevolg $O(N * x)$, hierbij is niet de tijdscomplexiteit van de secundaire heuristiek gerekend.

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  $\leftarrow$  midden  $\cup$  PartitieCompilatieKnoop(links,  $v^l$ )
4: end if
5: if rechts bevat mistens één element then
6:   midden  $\leftarrow$  midden  $\cup$  PartitieCompilatieKnoop(rechts,  $v^r$ )
7: end if
8: return Bottom-upCompilatie(midden, interneHeuristiek)

```

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 .

$$\alpha = \{(p_1, s_1), \dots, (p_n, s_n)\} = \{(p_i, s_i) | i \in [1, n]\} \quad (5.15)$$

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

$$\alpha \circ x = \alpha \circ \{(x, \top), (\neg x, \perp)\} \quad (5.16)$$

$$\alpha \circ x = \{(p_i \wedge x, s_i \circ \top), (p_i \wedge \neg x, s_i \circ \perp) | i \in [1, n]\} \quad (5.17)$$

De operatie \circ kan ofwel een conjunctie of een disjunctie voorstellen. We hebben volgende elementaire gelijkheden:

$$s_i \wedge \top = s_i, s_i \wedge \perp = \perp, s_i \vee \top = \top, s_i \vee \perp = s_i \quad (5.18)$$

In geval van een conjunctie wordt vergelijking 5.17

$$\alpha \wedge x = \{(p_i \wedge x, s_i), (p_i \wedge \neg x, \perp) | i \in [1, n]\} \quad (5.19)$$

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

$$\bigvee_{i=1}^n (p_i \wedge \neg x) = \bigvee_{i=1}^n (p_i) \wedge \neg x = \top \wedge \neg x = \neg x \quad (5.20)$$

We kunnen dus vergelijking 5.19 nog verder versimpelen naar

$$\alpha \wedge x = \{(p_i \wedge x, s_i) | i \in [1, n]\} \cup \{(\neg x, \perp)\} \quad (5.21)$$

Voor de disjunctie wordt vergelijking 5.17

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

Op gelijke manier als voor vergelijking 5.19 kunnen we vergelijking 5.23 verder versimpelen naar

$$\alpha \vee x = \{(p_i \wedge \neg x, s_i) | i \in [1, n]\} \cup \{(x, \top)\} \quad (5.23)$$

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

$$\alpha \circ x = \alpha \circ (\top \wedge x) \quad (5.24)$$

$$\alpha \circ x = \{(p_i, s_i \circ x) | i \in [1, n]\} \quad (5.25)$$

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	$\{(p_i \wedge x, s_i) i \in [1, n]\} \cup \{(\neg x, \perp)\}$	$\{(p_i, s_i \wedge x) i \in [1, n]\}$
Disjunctie	$\{(p_i \wedge \neg x, s_i) i \in [1, n]\} \cup \{(x, \top)\}$	$\{(p_i, s_i \vee x) i \in [1, n]\}$

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 5.26 kunnen we een heuristiek bedenken: apply eerst die SDD's, die voorkomen 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 5 (Vtree 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 $O(N * \log(N) * x)$.

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 rechttertakken, omdat deze bij de OBDD altijd lager liggen in de volgorde (elke linkertak in de vtree bevat slechts één variabele).

Heuristiek 5.1 (Inverse vtree 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 5.2 (Inverse vtree 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.*

5.3 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 werken. Voornamelijk geven we een tegenvoorbeeld voor heuristieken 3 en 4.

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

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

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 eigen-
lijk expo-
nentieel 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

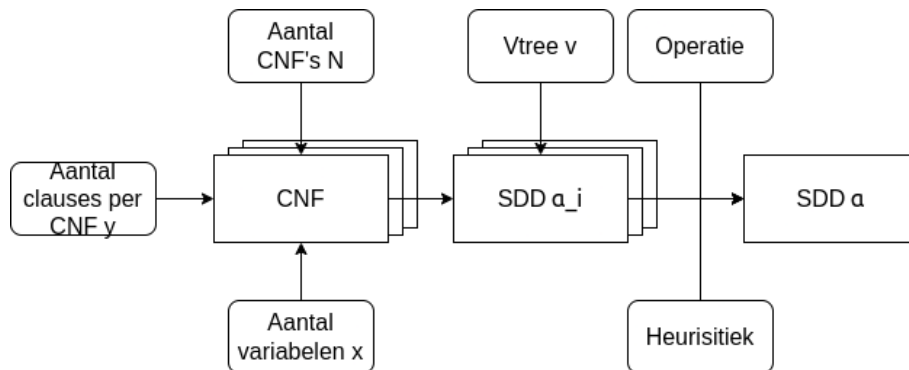
Experimenten

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 $n = x$ en de operatie (conjunctie of disjunctie). Voor elke probleeminstantie doen we het volgende:

1. We genereren eerst N verschillende CNF's, met y clauses en 3 variabelen per clause, deze stellen de verschillende k_i voor.
2. We compileren elke CNF 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 tijd ct en het geheugengebruik g nodig voor de bewerking wordt 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 elke parameter: waarvoor

figuur aanpassne zodat alpha niet meer gebruikt wordt



FIGUUR 6.1: experiment pijplijn

hij dient, welk effect hij heeft, en hoe hij van andere parameters afhankelijk is.

Het aantal variabelen x heeft een invloed op de grootte van de SDD's. Een elementaire bovengrens voor een SDD α met x variabelen is $|\alpha| \leq 2^x$.

De keuze van het aantal clauses per CNF y is sterk gelinkt aan onze keuze van x . Zoals gezien in sectie 4.3 moeten we opletten op de ratio $r = y/x$. 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 logische formule $f = \perp$. 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 dezelfde ratio, dus na een conjunctie zal die ratio verdubbelen (tenzij 2 CNF's toevallig eenzelfde clause hebben). Daar komt nog eens bij dat de conjunctie $f \wedge \perp = \perp$. 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 operatie kiezen we voornamelijk de disjunctie, als $r \leq 0.5$ kunnen we ook de conjunctie als operatie kiezen.

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 metriecken ct en g kleinere verschillen noteren. De tijdscomplexiteit van de heuristieken is ook in functie van N . Zo is de tijdscomplexiteit van heuristiek 4 kwadratisch in N , terwijl dit lineair is voor de andere heuristieken.

Voor v kiezen we een gebalanceerde vtree. De keuze voor deze vtree is voor de hand liggend. 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. In een gebalanceerde vtree ligt elke variabele, voor zover mogelijk, op dezelfde diepte, op die manier wordt er dus geen onderscheid gemaakt tussen de variabelen. De volgorde van de variabelen is ook belangrijk. Wij kiezen ervoor om tussen alle experimenten telkens dezelfde 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 x constant te houden en $y = r * x$ te kiezen.

We testen onze heuristieken ook in mindere mate voor de conjunctie-operatie als $r \leq 0.5$. Ook voor andere types vtrees, rechtslineair, linkslineair en willekeurig,

Heuristiek	Afkorting
Vtree partitionering + Kleinste eerst	VP + KE
Vtree partitionering + Elementgewijze bovengrens	VP + EL
Vtree variabelenvolgorde	VO
Inverse vtree variabelenvolgorde LR	IVO-LR
Inverse vtree variabelenvolgorde RL	IVO-RL

TABEL 6.1: Heuristieken gebruikt in de experimenten

testen we of onze heuristieken dezelfde verschijnselen vertonen. De resultaten van deze experimenten zijn te vinden in de appendix .

De heuristieken die we besproken hebben **zijn niet altijd logisch om te testen**, heuristiek 2: Clause-lengte werkt enkel op CNF's bijvoorbeeld. Heuristiek Vtree partitionering heeft een interne heuristiek nodig die de compilatietijd sterk kan beïnvloeden. Voor we de analyse van de experimenten beginnen, geven we voor elke (combinatie van) heuristiek(en) die we testen een logische afkorting.

appendix
maken

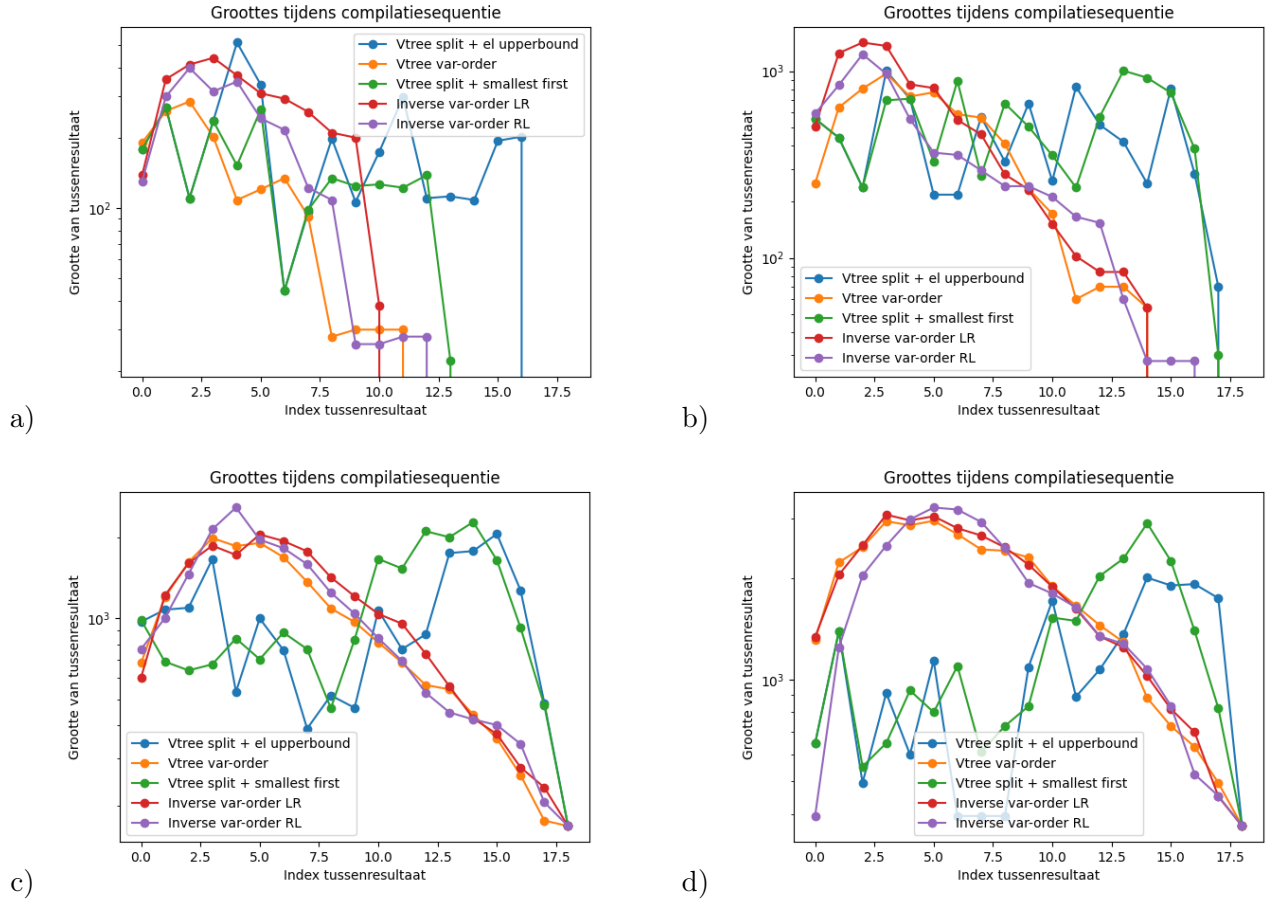
6.0.1 Generatie van SDD's

6.1 Resultaten: geheugengebruik

Eerst geven de resultaten van het geheugengebruik van elke heuristiek. Deze resultaten zullen ons inzicht geven van wat het globale effect is van elke heuristiek op de compilatiesequentie. Van elke heuristiek meten we de groottes γ_i van de compilatiesequentie van één probleeminstantie. De groottes van γ_i waarbij $\langle \gamma_i \rangle = k_i$ hebben we weggelaten, want deze zijn dezelfde ongeacht de heuristiek. Voor dit experiment kiezen we volgende parameters: $N = 20, x = 16, y \in \{4, 6, \dots, 5 \times x\}$, operatie = disjunctie, vtree = gebalanceerd.

Eerst geven de resultaten voor $y = 4, 6, 8, 10$. Deze zijn terug te vinden in tabel 6.2 In deze experimenten zijn de SDD's al onderbeperkt door het lage aantal clauses, en met de disjunctie als operatie zal de finale SDD met hoge kans triviaal waar zijn. We merken dat VO en IVO-RL ongeveer hetzelfde traject volgen, dit is niet verwonderlijk. We werken met een gebalanceerde vtree, hierdoor is er weinig verschil tussen de hoogtes van de variabelen. Het traject van deze twee is ook gelijkaardig aan IVO-LR. We concluderen dat er weinig verschil is tussen primes of subs de voorkeur geven bij een kleine ratio $r = y/x$. De variabelenvolgorde heuristieken doen het beter dan VP+EL en VP+KE. De tussenresultaten worden veel gelijkmatiger kleiner, en een triviaal resultaat (\top of \perp) wordt sneller gevonden. De trajecten van VP+EL en VP+KE lopen ook gelijkmatig. Dit is niet verwonderlijk, EL en KE werken volgens hetzelfde principe: een limiet leggen op de grootte na apply.

Als volgende bekijken we de groottes voor een ratio $r = y/x$ rond de kritieke waarde voor de grootte van een SDD (zie sectie 4.3) r ligt rond de kritieke waarde



TABEL 6.2: Groottes tijdens compilatiesequentie voor 4 (a), 6 (b), 8 (c) en 10 (d) clausules.

voor $y = 24, 26, 28, 30, 32, 34$. De resultaten hiervan staan in tabel 6.3 Als de ratio r rond de kritieke waarde van 1, 8 ligt, zijn de SDD's α_i groot. Ze bevatten elk met hoge kans alle variabelen. Dit zien we terugkomen in de grafieken. De heuristieken VO, IVO-LR en IVO-RL vallen (bijna) volledig samen. Deze drie heuristieken doen niets anders dan telkens de eerste twee SDD's te nemen in de inputlijst en achteraan toe te

voegen, met als compilatiesequentie:

$$\begin{aligned} \gamma_{N+1} &= \text{apply}(\gamma_1, \gamma_2, \circ) \\ \gamma_{N+2} &= \text{apply}(\gamma_3, \gamma_4, \circ) \\ &\dots \end{aligned}$$

$$\gamma_{2N-1} = \text{apply}(\gamma_{2N-3}, \gamma_{2N-2}, \circ).$$

SDD zo groot is, zal VP ook geen effect hebben. Het effect van de interne heuristiek is er nog wel. We zien dat de tussenresultaten bij VP+EL gelijkmatiger groter worden dan bij VP+KE, dit is nog duidelijker als willekeurige vtrees worden gebruikt.

Tenslotte geven we de resultaten voor $y = 72, 74, 76, 78$ in tabel 6.4. Bij dit aantal clausules is de ratio $r \geq 4.5$, volgens de fase-transitie is een 3-CNF met

zet dat op eigen lijn.
bv. $\begin{equation*}$

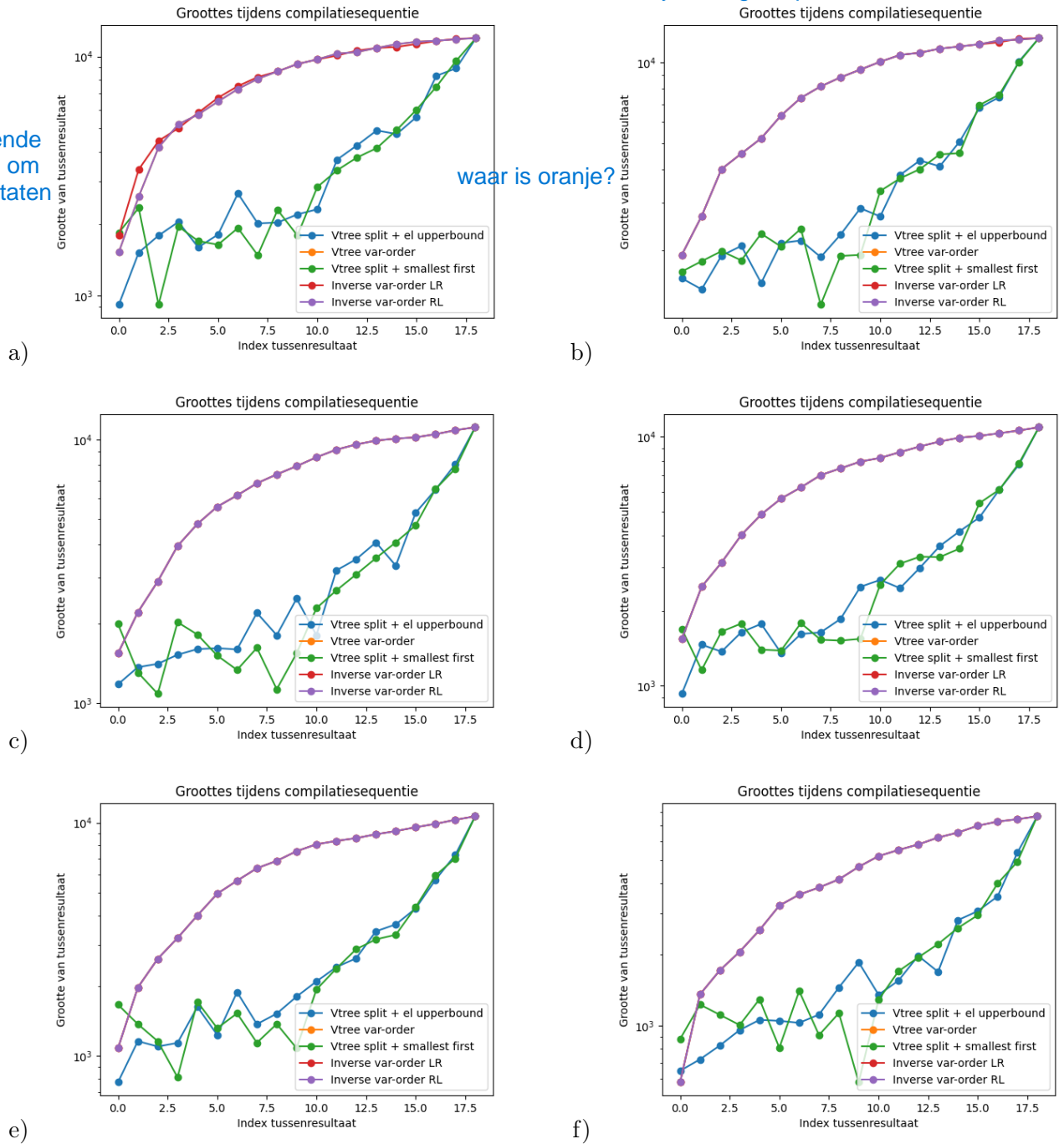
Omdat elke

Is dit gemiddelde grootte van meerdere SDDs?
...?

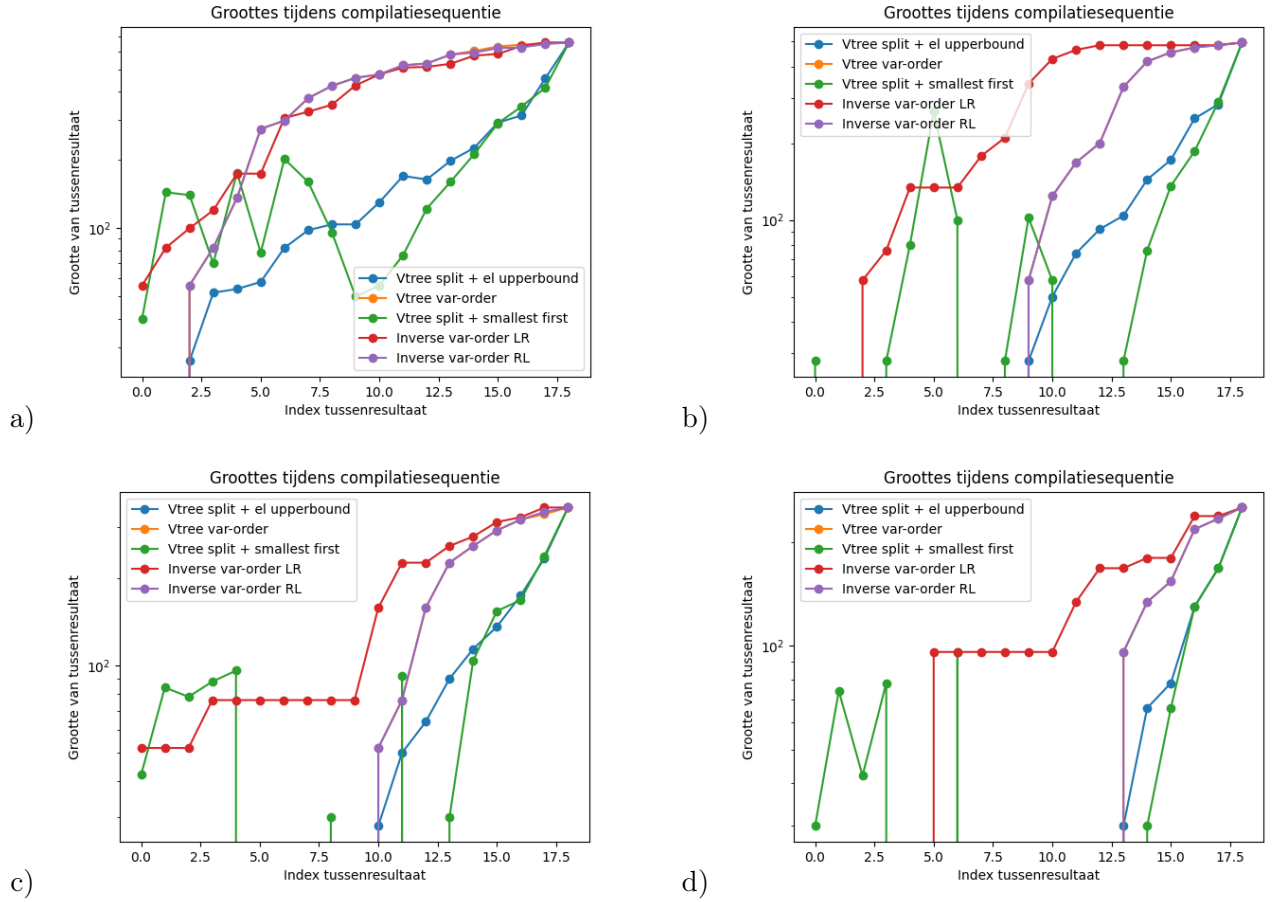
y-as begint op 0?

gebruik verschillende tekens (*, +, ., ...) om overlap van resultaten tegen te gaan.

waar is oranje?



TABEL 6.3: Groottes tijdens compilatiesequentie voor 24 (a), 26 (b), 28 (c), 30 (d), 32 (e) en 34 (f) clauses.



TABEL 6.4: Groottes tijdens compilatiesequentie voor 72 (a), 74 (b), 76 (c) en 78 (d) clausules.

deze ratio met hoge kans triviaal Onwaar, $\alpha_i = \perp$. Als de CNF niet triviaal is en $\alpha_i \neq \perp$, dan zal α_i veel of alle variabelen bevatten. Bij deze resultaten kunnen we voornamelijk twee zaken opmerken. Ten eerste zien we dat VP+EL meer triviale tussenresultaten vindt dan VP+KE, en een meer gelijkmatige toename van de grootte van de tussenresultaten op het einde. Voor de variabelenvolgorde heuristieken zien we dat IVO-RL het beter doet dan IVO-LR. (hier valt VO opnieuw samen met IVO-RL).

Deze resultaten tonen de groottes van de compilatiesequentie van één probleeminstantie. Andere probleeminstanties tonen hetzelfde macro-verloop. De resultaten voor andere ratio's r zijn te vinden in de appendix. Voor $0,5 < r < 1,8$ wordt het eindresultaat stapsgewijs groter, van triviaal tot de grootste in de sequentie. Voor $1,8 < r < 4,2$ zien de grafieken er gelijkaardig uit als voor $r \approx 1,8$. Voor $4,2 < r$ zien we dat er meer en meer SDD's triviaal zijn naarmate r groter wordt.

voeg appendix toe

De drie heuristieken gebaseerd op de vtree variabelenvolgorde werken incrementeel,

de compilatiegroottes nemen heel gelijkmatig toe of af. Dit is te verwachten. Voor de compilatie worden alle SDD's gesorteerd. Voor de apply-operatie geldt dat de SDD $|\alpha \circ \beta|$ met hoge kans alle variabelen in α en β bevat. Door de aard van de sorteervolgorde zal deze $|\alpha \circ \beta|$ hoogstwaarschijnlijk gebruikt worden in de volgende apply-operatie.

Gebaseerd op de compilatiegroottes verwachten we dat als de ratio r klein is $r < 1$ een heuristiek op basis van de vtreevariabelenvolgorde het goed zal doen. Voor hogere r wordt verwacht dat VP+EL het beste zal presteren.

6.1.1 Resultaten: compilatietijd

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.
- [Bov16] Simone Bova. Sdds are exponentially more succinct than obdds. *CoRR*, abs/1601.00501, 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 sdds are exponentially more succinct than obdds. 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.

- [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.
- [SDD]
- [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
(begrensd) conjunctie	(bounded) conjunction
(begrensd) 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) determinism
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