Apply order of a sdd-compiler

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

19 februari 2024

Inhoudsopgave

Samenvatting

1 inleiding

In allerlij vakgebieden van computerwetenschappen worden kennisbases gebruikt. Bijvoorbeeld binnen Probabilistic Reasoning voor het tellen van mogelijkheden. [https://dtai.cs.kuleuven.be/stories/post/vincent-derkinderen/role-of-counting-in-probabilistic-reasoning/] Een kennisbasis bestaat uit verschillende variabelen. elke variabele kan True of False zijn, en afhankelijk van de waarheidswaarde van de variabelen is de volledige kennisbasis True of False.

Er kunnen verschillende soorten bewerkingen of queries worden uitgevoerd op een kennisbasis, namelijk: [knowledge compilation map]

Misschien beter over logische formules praten ipv "kennisbases"
een logische combinatie van variabelen

bewerkingen	queries
conditioning	consistency check
forgetting	validity check
singleton forgetting	clausal entailment check
conjunction	implicant check
bounded conjunction	equivalence check
disjunction	sentential entailment check
bounded disjunction	model counting
negation	model enumeration

Dit zijn enkele voorbeelden van simpele bewerkingen, door een combinatie van simpele bewerkingen kunnen geavanceerde toepassingen zoals de eerder genoemde Probabilistic Reasoning (=probabilistische inferentie?), foutdetectie en diagnose van systemen etc. gesimuleerd worden. [veel papers als je gewoon zoekt naar sdd of andere voorstellingen, ook iets van [Graph-based algorithms for Boolean function manipulation.]?]

Het probleem van logische formules is dat ze op verschillende manieren kunnen voorgesteld worden. Een bepaalde voorstelling kan efficiënt zijn voor bepaalde queries of bewerkingen. Maar als gevolg zijn er nadelen zoals andere queries of bewerkingen die minder efficiënt uitgevoerd kunnen worden, meer geheugengebruik, ... noem maar op. Het bestuderen van dergelijke voorstellingen is het vakgebied van kenniscompilatie. Elke toepassing vereist andere queries of bewerkingen met als gevolg dat er heel wat kennisvoorstellingen zijn bedacht en bestudeerd[bron knowledge compilation map]. Er bestaan 2 algemene termen om voorstellingen te labelen, succintness voor de grootte van de representatie en tractability voor de set van bewerkingen/queries die in polytime kunnen worden uitgevoerd. In het algemeen geldt: hoe meer tractable een voorstelling is, hoe minder succint [A knowledge compilation map]. 2 soorten voorstellingen die tractable zijn, en dus voor veel toepassingen gebruikt kunnen worden, maar toch ook relatief succint zijn, zijn de Deterministic Decomposable Negation Normal Form (d-dnnf) en de Ordered Binary Decision Diagram (OBDD). (of is ddnnf enkel vanuit theoretisch standpunt nuttig?)

Recent is er een nieuwe veelbelovende representatie voorgesteld, de SDD. citeer SDD paper De SDD is een subset van de D-DNNF, maar een superset van de OBDD, met interessante eigenschappen van beide voorstellingen. De paper onderzoekt het probleem van grote tussenresultaten tijdens de bottom-up compilatie van een SDD.

* Keuze van Engelse woorden ipv NL is soms vreemd.
* Let op bij afkortingen: d-DNNF, OBDD en SDD.
* Schrijf "dmv." etc voluit.

Zeg iets meer over het probleem van grote tussenresultaten.
Schets bv kort het probleem van (A & B) & C of A & (B & C).

2 SDD als nieuwe voorstelling van een kennisbasis

De meest belangrijke eigenschappen van de d-dnnf zijn decomposability and determinism [On probabilistic inference by weighted model counting], deze eigenschappen maken dat de voorstelling succint is.

De meest belangrijke eigenschappen van de obdd zijn canonicity en polytime apply (conjoin, disjoin, negatie). Canonicity houdt in dat een bepaalde kennisbasis altijd op dezelfde manier zal worden voorgesteld afhankelijk van de variable ordering, waardoor equality checking in polytime kan, maar ook zoeken naar een optimale representatie (adhv variable ordering) is hierdoor mogelijk. Polytime apply maakt mogelijk dat een voorstelling bottom-up kan gecompiled worden. De OBDD is minder succint dan de D-DNNF, maar door de canonicity en polytime apply is ze in de praktijk wel het meest populair.

De Sentential Decision Diagram (SDD) heeft als eigenschappen structured decomposability [New compilation languages based on structured decomposability] en strong determinism [A lower bound on the size of decomposable negation normal form]. Deze eigenschappen zijn stricter dan decomposability en determinism, de set van SDD's is dus een stricte subset van de set van D-DNNF's. Verder is de set van sdds een stricte superset van de set van obdds. Als gevolg bevat de SDD ook bepaalde belangrijke eigenschappen van de OBDD, namelijk canonicity en polytime apply Het is al gebleken dat SDD meer succint is dan de OBDD tov pathwidth [SDD: A New Canonical Representation of Propositional Knowledge Bases] en voor bepaalde kennisbases is de SDD exponentieel meer compact [Basing Decisions on Sentences in Decision Diagrams]

2.1 strong determenism en partities

Eerst en vooral een paar opmerkingen over notatie. Hoofdletters zoals X zijn voor variabelen, vetgedrukte hoofdletters $\mathbf{X}$ zijn voor sets van variabelen, lowercase letters zoals zijn voor een instantiatie van een variabele, $\bot$ wordt gebruikt voor False en $\top$ voor True.

OBDD's zijn gebasseerd op Shannon-decomposities, hierbij wordt een formule f opgesplitst in 2 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 sentence.

Gegeven een booleanse functie f over variabelen $\mathbf{X}$, $\mathbf{Y}$, met $\mathbf{X} \cup \mathbf{Y} = \emptyset$. Als $f = (p_1(\mathbf{X}) \wedge s_1(\mathbf{Y})) \vee \cdots \vee (p_n(\mathbf{X} \wedge s_n(\mathbf{Y}))$, dan is $\{(p_1, s_1), \cdots, (p_n, s_n)\}$

een $(\mathbf{X}, \mathbf{Y})$-decompositie van f. Als ook nog geldt dat $p_i \wedge p_j = \bot$ voor elke $i \neq j$ dan wordt de decompositie strongly deterministic 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.

Gegeven een $(\mathbf{X}, \mathbf{Y})$-decompositie α van een functie f die strongly determenistic is. α wordt een $\mathbf{X}$-partitie van f genoemd asa. de primes een partitie vormen [sdd paper]. 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 waar.

Als geldt dat elke sub in de $\mathbf{X}$-partitie distinct is ($s_i \neq s_j$ als $i \neq j$), dan is α compressed.

Een apply-operatie bestaat uit de conjunctie of disjunctie van twee SDD's. 2 formules zou je simpelweg kunnen samennemen, maar de apply-operatie moet als resultaat opnieuw een SDD outputten. We moeten dus als het ware de twee SDD's samenvoegen.

Neem $\circ$ als een booleanse operator en $\{(p_1, s_1), \ldots, (p_n, s_n)\}$ en $\{(q_1, r_1), \ldots, (q_m, r_m)\}$ $\mathbf{X}$-partities van respectievelijk $f(\mathbf{X}, \mathbf{Y})$ en $g(\mathbf{X}, \mathbf{Y})$, dan is $\{(p_i \wedge q_j, s_i \circ r_j) \mid p_i \wedge q_j \neq \bot\}$ een $\mathbf{X}$-partitie van $f \circ g$. [sdd paper]

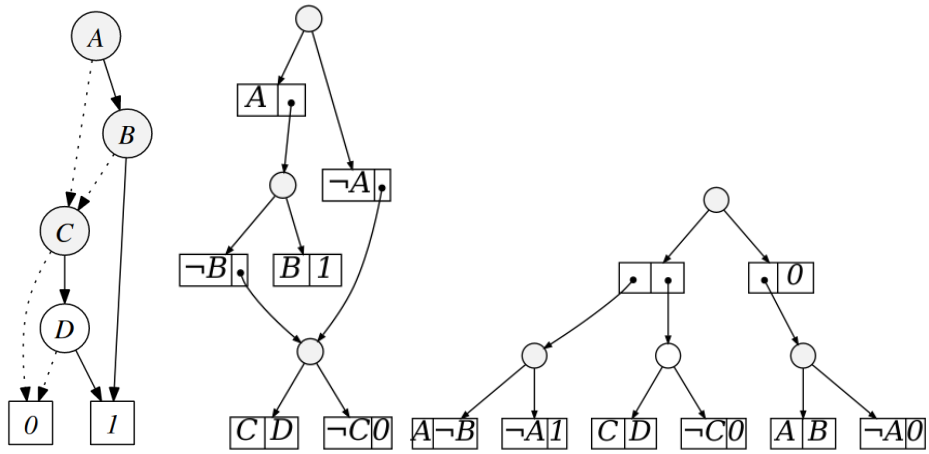
Deze eigenschap is zeer essentieel voor de bruikbaarheid van de SDD. Het maakt een polytime apply-operatie mogelijk.

Verder is het bewezen dat een booleanse functie $f(\mathbf{X}, \mathbf{Y})$ exact één compressed $\mathbf{X}$-partition heeft [sdd paper]. Omwille van deze eigenschap is een compressed sdd canonical [sdd paper].

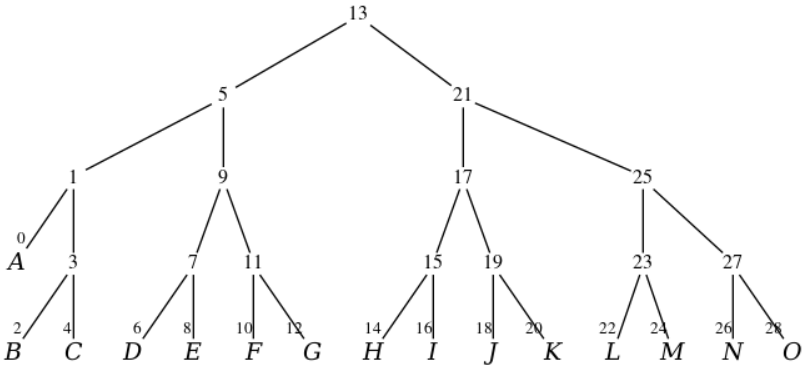
2.1.1 vtree

In een OBDD wordt gesplist op variabelen, de volgorde van deze variabelen wordt vastgelegd door de variable ordering. De volgorde heeft een grote invloed op de totale grootte van de voorstelling. Daarom is er al veel onderzoek naar gedaan, zo is gebleken dat een optimale variable ordering vinden een NP-Complete probleem is [bron?]. Omwille van die reden zijn er heuristieken ontwikkeld zoals Mince [bron?].

Voor een SDD moet ook een volgorde worden vastgelegd, maar er wordt gesplitst op meerdere variabelen in één keer. Hiervoor is de vtree verantwoordelijk [New compilation languages based on structured decomposability]. Een vtree voor variabelen $\mathbf{X}$ is een binaire boom, wiens leaves een



Figuur 1: obdd, right-sdd en balanced-sdd voor de formule $(A \wedge B) \vee (C \wedge D)$



Figuur 2: balanced Vtree voor een set van 15 variabelen

één-op-één correspondentie hebben met de variabelen in $\mathbf{X}$ [sdd paper]. De vtree wordt gebruikt om recursief een booleanse functie f op te splitsen. In figuur 2 kunnen we een voorbeeld van een vtree zien. We refereren naar een node in de vtree of de subtree rooted in die node dmv. v, en de linker en rechter childs of subtrees dmv v^l en v^r. Het verschil tussen linker en rechternodes in de vtree is belangrijk omwille van het verschil tussen primes en subs

2.2 sdd definition

De originele paper introduceert een mapping $\langle\dot{}\rangle$ van SDD's naar booleanse formules, deze herhalen we hier omdat ze essentieel is om SDD's te begrijpen.

α is een sdd die vtree v respecteert asa.

1. $\alpha = \bot$ of $\alpha = \top$, $\langle \bot \rangle = False$, $\langle \top \rangle = True$.

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 subtrees van v^l respecteert en elke s_i is een sdd die subtrees 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 gegeven in de originele paper over SDD's [sdd paper] zijn compressed SDD's. Een SDD is compressed asa. voor elke decompositie $\{(p_1, s_1), \cdots, (p_n, s_n)\}$ in de SDD geldt dat $s_i \not\equiv s_j$ voor $i \neq j$.

2.3 apply operatie

Het is relatief makkelijk om een algoritme te verzinnen dat twee compressed SDD's in polytime applied mbv. de eerder geziene eigenschap van **X**-partities. In de originele paper wordt pseudo-code gegeven [sdd bron] voor de polytime apply-operatie. Het resultaat van die apply-operatie is een SDD, maar deze SDD is niet persé compressed. Een uncompressed SDD is niet canonical. Zoals eerder gezegd is canonicity in de praktijk zeer interessant, dus kunnen we niet de standaard apply-operatie gebruiken. Het is gebleken dat een apply-operatie waarvan de resulterende SDD compressed is, niet in polytime uitvoerbaar is [bron]. Het bewijs hiervan is gebaseerd op bepaalde logische formules waarvan de SDD's na een apply-operatie exponentieel groeien. Wel is uit diezelfde paper gebleken dat in de praktijk een compressed-apply-operatie wel degelijk verkiesbaar is [2de sdd paper]. Aangezien in de praktijk canonicity een belangrijke eigenschap is zullen we in deze paper veronderstellen dat elke SDD ook een compressed SDD is en met de apply-operatie bedoelen we de compressed versie. Als nadeel moeten we nu rekening houden dat elke apply-operatie van twee SDD's in theorie exponentieel veel tijd en geheugen nodig kan hebben.

3 Probleemstelling

Zoals in de originele paper al werd verteld, is er veel onderzoek gedaan naar de OBDD. Als de SDD de OBDD wil outperformen zal er ook onderzoek gedaan moeten worden, bijvoorbeeld naar het effect van de vtree op de grootte van de sdd, size bounds, compilers...

In deze paper onderzoeken we het probleem van grote tussenresultaten. In de praktijk zijn er twee manieren om een logische formule om te zetten naar

een voorstelling om er daarna bewerkingen op uit te voeren. Er zijn top-down compilers[top down compiler paper] en bottom-up compilers[verschillende papers]. Een logische formule kan in het algemeen als een Directed Acyclic Graph beschouwd worden, elke node n is dan een conjunctie, disjunctie of negatie van verschillende subformules, de childnodes van n. Een bottom-up compilatie zet eerst alle leaves van de DAG om naar SDD's, die SDD's worden dmv. een apply-operatie samengevoegd volgens de DAG. Als een node n meer dan twee childnodes heeft, moet er een keuze gemaakt worden welke childnodes eerst geapplied worden. Niet elke apply-operatie duurt even lang, maar het eindresultaat, de SDD van die node

Een voorbeeld: neem de logische formule $f = (A \vee B) \wedge (A) \wedge (\neg A \vee C)$. We willen de SDD representatie α van deze formule berekenen. Tijdens bottom-up compilatie zal eerst SDD α_1, $\langle \alpha_1 \rangle = (A \vee B)$ berekend worden, dan α_2, $\langle \alpha_2 \rangle = A$ en tenslotte α_3, $\langle \alpha_3 \rangle = (\neg A \vee C)$. Nadat deze drie SDD's berekend zijn moet er een keuze gemaakt worden, berekenen we eerst $\alpha_1 \wedge \alpha_2$, $\alpha_2 \wedge \alpha_3$ of $\alpha_1 \wedge \alpha_3$. We weten dat de volgorde niet uitmaakt voor het finale resultaat omwille van de canonicity, maar de tussenresultaten verschillen wel. Dit kan leiden tot verschillend tijdsgebruik en geheugengebruik van de compilatie van α.

Concreet kunnen we het probleem als volgt stellen: We willen $\alpha = \bigvee_{i=1}^{N} \alpha_i$, met α_i een SDD, berekenen. Dit vergt (N-1) apply operaties. Er zijn $2^(N-1)$ manieren om dit te doen omdat de apply-operatie binair is. Zijn er heuristieken die het geheugen en tijdsgebruik van de apply-operaties minimaliseren?