Propositional DAGs: a New Graph-Based Language for Representing Boolean Functions

Michael Wachter and **Rolf Haenni**
University of Bern
Institute of Computer Science and Applied Mathematics
CH-3012 Bern, Switzerland
{wachter,haenni}@iam.unibe.ch

Abstract

This paper continues the line of research on knowledge compilation in the context of Negation Normal Forms (NNF) and Binary Decision Diagrams (BDD). The idea is to analyze different target languages according to their succinctness and the classes of queries and transformations supported in polytime. We identify a new property called simple-negation, which is an implicit restriction of all NNFs and BDDs. The removal of this restriction leads to Propositional Directed Acyclic Graphs (PDAG), a more general family of graph-based languages for representing Boolean functions or propositional theories. With respect to certain NNF-based languages, we will show that corresponding PDAG-based languages are at least as succinct and support the same transformations. The most interesting language even supports the same queries and an additional transformation, making it more flexible.

Introduction

Boolean functions are important in many areas of computer science and mathematics, most notably in artificial intelligence, digital system design, formal verification, mathematical logic, and combinatorial optimization. In most cases, they are used to represent knowledge as sets of possible states with respect to some propositional variables describing the world. In this sense, Boolean functions are fundamental knowledge representation tools.

In practice, working with Boolean functions presupposes efficient ways to represent them. Classical approaches such as truth tables, Karnaugh maps, or canonical sum-of-products (e.g. prime implicates or prime implicants) are known to be impractical, as they impose representations of size $O(2^n)$ for *most* or *all* possible n-ary functions (Hill & Peterson 1974). More sophisticated representations are not of exponential size at least for *many* functions. Among them, the most prominent ones are *Binary Decision Diagrams* (BDD) (Akers 1978; Bryant 1986). They are rooted *directed acyclic graphs* (DAG), in which non-leaf nodes are labeled with the names of propositional variables, and the leaf nodes are the truth values 0 or 1. The discovery of BBDs has tremendously accelerated the progress of applications which rely on efficient methods to represent and manipulate Boolean functions. This is mainly due to the fact that the class of so-called *Ordered* BDDs (OBDD) permits a Boolean function to be compactly represented in a canonical form (Bryant 1992).

A more general view offers Darwiche in a remarkable series of papers on *Negation Normal Forms* (NNF) and their derivates (Darwiche 1999; 2001a; 2002; Darwiche & Marquis 2002). Like BDDs, NNFs are rooted directed acyclic graphs, but the non-leaf nodes are now labeled with $\wedge$ (logical and) or $\vee$ (logical or), and the leaf nodes are literals (propositional symbols or their negations). BDDs can be regarded as special cases of NNFs. In Darwiche's terminology, the sets of all possible NNFs or all possible BDDs are *languages*, i.e. `BDD` is a sub-language of the `NNF` language. Other `NNF` sub-languages are obtained from a number of different properties which may or may not hold. The most important ones are *flatness* (`f`), *determinism* (`d`), and *decomposability* (`D`). Every combination of these properties defines another sub-language, e.g. `d-DNNF` denotes the sub-language for which determinism and decomposability hold. A complete picture of the most relevant `NNF` sub-languages is given in (Darwiche & Marquis 2002).

The crucial point about Darwiche's approach is the observation that some `NNF` sub-languages allow certain classes of queries to be answered in polytime. With respect to this, the most interesting languages are `DNNF` (allows consistency and clause entailment checking as well as model enumeration in polytime), `d-DNNF` (additionally allows validity and term implication checking as well as model counting in polytime), and `OBDD` (additionally allows equivalence testing in polytime). Among these, `DNNF` is the *most succinct* and `OBDD` the *least succinct* language. In general, more succinct languages answer less queries in polytime and vice versa. The picture given in (Darwiche & Marquis 2002) thus serves as a *knowledge compilation map*, from which the most succinct *target compilation language* is chosen, but such that the necessary queries are answerable in polytime. Darwiche's idea thus is to *compile* the knowledge off-line into the chosen target language, which is then used on-line to answer queries in polytime. An overview of such *knowledge compilation* techniques is given in (Cadoli & Donini 1997).

This paper continues this line of research and proposes an extension of the knowledge compilation map given in (Darwiche & Marquis 2002). It starts from the observation, that a leaf node of a NNF with a negative literal attached to it is

like the *negation* of a leaf node with the corresponding positive literal attached to it. The idea thus is to extend NNFs with negations, i.e. additional nodes labeled with $\neg$ (logical not), and to restrict the leaf nodes to propositional symbols only. In general, such *Propositional Directed Acyclic Graphs* (PDAG) do not impose the negations to be "at the bottom" of the DAG. This *simple-negation* property is implicitly included in all NNFs, but with respect to polytime queries, it turns out that this is not always a necessary requirement.

In the following section, we start by formally defining propositional DAGs and the corresponding PDAG language. We will then discuss different properties and identify a variety of sub-languages. Thereafter, the succinctness relations between these sub-languages are analyzed, leading to a reduced list of the interesting languages. Then, we are going to take a look at queries and transformations, and we will show for each sub-language which queries and transformations can be handled in polytime. Finally, two open questions will be discussed with respect to possible polytime queries and transformations and the selection of an appropriate target compilation language. The appendix gives the necessary proofs.

Propositional DAGs

Consider a set V of r propositional variables and a Boolean function (BF) $f : \{0,1\}^r \to \{0,1\}$. Such a function f can also be viewed as the set of n-dimensional vectors $\mathbf{x} \in \{0,1\}^r$ for which f evaluates to 1. This is the so-called *satisfying set* or *set of models* $S_f = \{\mathbf{x} \in \{0,1\}^r : f(\mathbf{x}) = 1\}$ of f, for which an efficient representation has to be found (Clote & Kranakis 1998). As in the case of BDDs and NNFs, the representation we propose here is based on directed acyclic graphs, but now we impose the following particularities.

Definition 1. A *Propositional DAG* (PDAG[1]) is a rooted directed acyclic graph of the following form:

1. Leaves are represented by $\bigcirc$ and labeled with $\top$ (true), $\bot$ (false), or $x \in V$;
2. Non-leaves are represented by $\triangle$ (logical and), $\triangledown$ (logical or), or $\diamond$ (logical not);
3. $\triangle$- and $\triangledown$-nodes have at least one child;
4. $\diamond$-nodes have exactly one child.

Leaves labeled with $\top$ ($\bot$) represent the constant BF which evaluates to 1 (0) for all $\mathbf{x} \in \{0,1\}^r$. A leaf labeled with $x \in V$ is interpreted as the assignment $x = 1$, i.e. it represents the BF which evaluates to 1 iff $x = 1$. The BF represented by a $\triangle$-node is the one that evaluates to 1, iff the BFs of all its children evaluate to 1. Similarly, a $\triangledown$-node represents the BF that evaluates to 1, iff the BF of at least one child evaluates to 1. Finally, a $\diamond$-node represents the complementary BF of its child, i.e. the one that evaluates to 1, iff the BF of its child evaluates to 0. The BF of an arbitrary PDAG φ will be denoted by f_φ.

Formally, we will write $\texttt{PDAG}_V$ for the set of all possible PDAGs with respect to V. We follow the view from (Darwiche 2001a; Darwiche & Marquis 2002) and call $\texttt{PDAG}_V$ a *language*. When no confusion is anticipated, we omit the reference to the set V, i.e. we simply write PDAG instead of $\texttt{PDAG}_V$. Our convention is to denote PDAGs by lower-case Greek letters such as φ, ψ, or the like. Note that any node α included in a PDAG φ defines its own (sub-) PDAG, and is thus another element of PDAG.

The number of edges of a PDAG $\varphi \in \texttt{PDAG}$ is called its *size* and denoted by $|\varphi|$. A *literal* is either a leaf or a $\diamond$-node whose child is a leaf. PDAGs are called *binary*, if no $\triangle$- or $\triangledown$-node has more than two children. The set of variables included in a sub-PDAG α of φ is denoted by $vars(\alpha)$. The *path-length* of a path from the root to a leave is the number of edges minus the number of $\diamond$-nodes along the path. The *height* of φ, denoted by $h(\varphi)$, is its maximal path-length.

Any Boolean function can be represented by a PDAG, so the PDAG language is *complete*. On the other hand, PDAGs are not *canonical*, i.e. we may have several equivalent PDAGs representing the same Boolean function. Two PDAGs $\varphi, \psi \in \texttt{PDAG}$ are *equivalent*, iff $f_\varphi = f_\psi$. This is denoted by $\varphi \equiv \psi$. Furthermore, φ *entails* ψ, denoted by $\varphi \models \psi$, iff $f_\varphi(\mathbf{x}) \leq f_\psi(\mathbf{x})$ for all $\mathbf{x} \in \{0,1\}^r$. In terms of their satisfying sets, $S_{f_\varphi} = S_{f_\psi}$ means equivalence and $S_{f_\varphi} \subseteq S_{f_\psi}$ entailment.

Figure 1 depicts two equivalent PDAGs φ_1 and φ_2 representing the so-called *odd parity function* with respect to $V = \{a, b, c, d\}$. They are both binary and have the same height $h(\varphi_1) = h(\varphi_2) = 4$, but φ_1 with $|\varphi_1| = 24$ is substantially smaller than φ_2 with $|\varphi_2| = 32$.

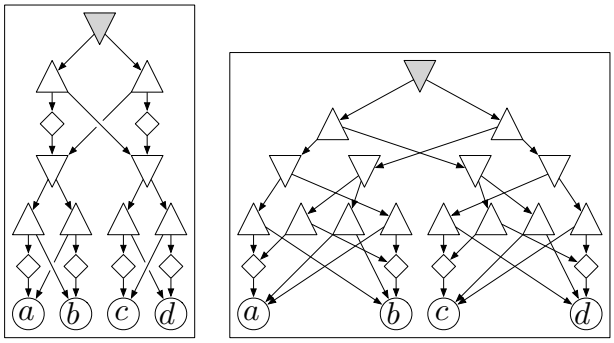

Figure 1: Two PDAGs representing the odd parity function.

We will now turn our attention to certain sub-languages of PDAG. According to (Darwiche 2001a; Darwiche & Marquis 2002), a sub-language qualifies as a target compilation language if it permits at least polytime consistency and clause entailment tests. PDAG does not qualify as a target compilation language unless P = NP (Papadimitriou 1994; Darwiche & Marquis 2002), but many of its sub-languages do. The classification of sub-languages is done according to the following basic properties.

[1] Some authors use the same acronym PDAG for *Partially Directed Acyclic Graphs* (Verma & Pearl 1992). In graph theory, these structures are more commonly known as *mixed (acyclic) graphs* (Gross & Yellen 2003), i.e. the ambiguity of using PDAG for *Propositional* DAGs is not severe.

Definition 2. A propositional DAG $\varphi \in$ PDAG may possess the following properties:

1. *Flatness*, if $h(\varphi) \leq 2$;
2. *Decomposability*, if the sets of variables of the children of each $\wedge$-node α in φ are pairwise disjoint (i.e. if $\beta_1, \ldots, \beta_n$ are the children of α, then $vars(\beta_i) \cap vars(\beta_j) = \emptyset$ for all $i \neq j$);
3. *Determinism*, if the children of each $\vee$-node α in φ are pairwise logically contradictory (i.e. if $\beta_1, \ldots, \beta_n$ are the children of α, then $\beta_i \wedge \beta_j \equiv \bot$ for all $i \neq j$);
4. *Simple-negation*, if the child of each $\diamond$-node in φ is a leaf.

These basic properties, i.e. flatness, decomposability, determinism, and simple-negation, will be abbreviated by f, c, d, and n, respectively.[2] This allows us to identify the following 15 sub-languages of PDAG:

- One property: f-PDAG, c-PDAG, d-PDAG, n-PDAG;
- Two properties: fc-PDAG, fd-PDAG, fn-PDAG, cd-PDAG, cn-PDAG, dn-PDAG;
- Three properties: fcd-PDAG, fcn-PDAG, fdn-PDAG, cdn-PDAG;
- All four properties: fcdn-PDAG.

Note that φ_1 on the left hand side of Fig. 1 is an element of cd-PDAG, whereas φ_2 on the right hand side is an element of cdn-PDAG. Some sub-languages are of minor relevance and will not be further discussed. The upper part of Fig. 2 shows the PDAG language together with the nine most relevant of these sub-languages. They are emphasized by bold borders.

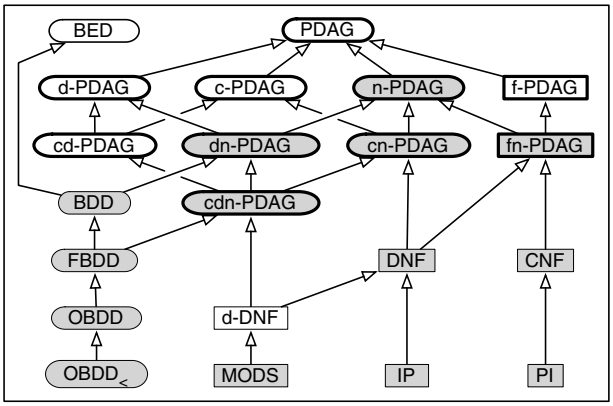

Figure 2: The family of PDAG-based languages considered in this paper. An edge $L_1 \rightarrow\!\!\!\!\!\cdot\; L_2$ means that L_1 is a sub-language of L_2. Flat languages are represented by rectangles. The gray colored shapes are the NNF languages discussed in (Darwiche & Marquis 2002).

The n-PDAG language together with its descendants correspond to Darwiche's NNF languages (Darwiche & Marquis 2002). In order to make our notation self-consistent and to strengthen that n-PDAG is the subset of PDAG for which the simple-negation property holds, n-PDAG is used instead of NNF. Accordingly, we prefer cn-PDAG to DNNF, dn-PDAG to d-NNF, cdn-PDAG to d-DNNF, and so on.

Other sub-languages are obtained from further properties. The BDD language, for example, is the subset of dn-PDAG for which the *decision* property holds. This is the connection to the world of *Binary Decision Diagrams* (Akers 1978; Bryant 1986) mentioned before. Other relevant properties for BDDs are *read-once* and *ordering*, from which we obtain the languages FBDD of *free* and OBDD of *ordered* BDDs. The BDD language with its descendants are depicted on the left hand side of Fig. 2. For further details about BDDs and the corresponding properties, we refer to (Darwiche & Marquis 2002).

Classical propositional languages such as disjunctive normal forms (DNF), conjunctive normal forms (CNF), prime implicates (PI), and prime implicants (IP) are all sub-languages of fn-PDAG for which either the *simple-conjunction* or *simple-disjunction* property holds, as shown at the bottom of Fig. 2. Again consider (Darwiche & Marquis 2002) for further details. To round off the picture, we also include the language d-DNF of *deterministic* DNFs, also called *disjoint Sum-of-Products* (SOP), which plays an important role in reliability theory (Rai, Veeraraghavan, & Trivedi 1995; Rauzy *et al.* 2003). However, we will not further consider d-DNF in the rest of this paper.

By replacing the non-terminal nodes of $\varphi \in$ PDAG by corresponding logical gates, one can look at it as a *digital circuit* implementing f_φ. We prefer to make a distinction between PDAGs and digital circuits to emphasize their respective purposes. Digital circuits are mainly used to *implement* BFs, whereas PDAGs are useful to *represent*, *manipulate*, and answer *queries* about BFs. In this respect, PDAGs are similar to *Boolean Expression Diagrams* (BED), a generalization of BDDs (Andersen & Hulgaard 1997), but the PDAG perspective offers a much finer graduation of sub-languages.

Succinctness

In the previous section, PDAG and a variety of sub-languages have been discussed. The languages we are mainly interested in are shown in Fig. 2. Most of them, namely those for which the simple-negation property holds, have been extensively studied in the literature (Darwiche 2001a; Darwiche & Marquis 2002; Darwiche 2002; Bryant 1986). The new languages proposed here are PDAG, d-PDAG, c-PDAG, f-PDAG, and cd-PDAG. In the same way as in (Darwiche & Marquis 2002), the *succinctness* relations of these languages will be analyzed now. With respect to two languages L1 and L2, the intuitive idea is to figure out whether a Boolean function may be represented more compactly by an element $\varphi_1 \in$ L1 or by an element $\varphi_2 \in$ L2. The following definition corresponds to the one given in (Darwiche & Marquis 2002).

Definition 3. Let L1 and L2 be two languages. L1 is *more succinct* than L2, denoted L1 $\preceq$ L2, iff for every $\varphi_2 \in$ L2, there is a $\varphi_1 \in$ L1 such that $\varphi_1 \equiv \varphi_2$ and the size $|\varphi_1|$ is

[2] Note that Darwiche uses D rather than c for decomposability. We prefer the latter to make our notation case insensitive. Furthermore, it is easier to think of c and d as properties referring to conjunctions and disjunction, respectively.

polynomial in the size $|\varphi_2|$.

The relation $\preceq$ is clearly *reflexive*, *anti-symmetric*, and *transitive*, i.e. it defines a *partial order* over all possible subsets of PDAG. Two languages L1 and L2 are called *equally succinct*, denoted by L1 $\equiv$ L2, iff L1 $\preceq$ L2 and L2 $\preceq$ L1. The language L1 is called *strictly more succinct* than L2, denoted by L1 $\prec$ L2, iff L1 $\preceq$ L2 and L2 $\not\preceq$ L1. They are *incomparable*, iff L1 $\not\preceq$ L2 and L2 $\not\preceq$ L1. Note that L1 $\supseteq$ L2 implies L1 $\preceq$ L2, i.e. the edges in Fig. 2 represent both subset and succinctness relationships where L1$\multimap$L2 means L1 $\preceq$ L2.

Proposition P1. PDAG, d-PDAG, c-PDAG, n-PDAG, and BED are equally succinct.

Proposition P2. f-PDAG and fn-PDAG are equally succinct.

Figure 3 repeats the upper part of Fig. 2 and illustrates the results of P1 and P2. To simplify the picture, the idea is to group equally succinct languages, i.e. we will use PDAG as a general term for PDAG, d-PDAG, c-PDAG, n-PDAG, and BED. Similarly, f-PDAG becomes a general term for both f-PDAG and fn-PDAG.

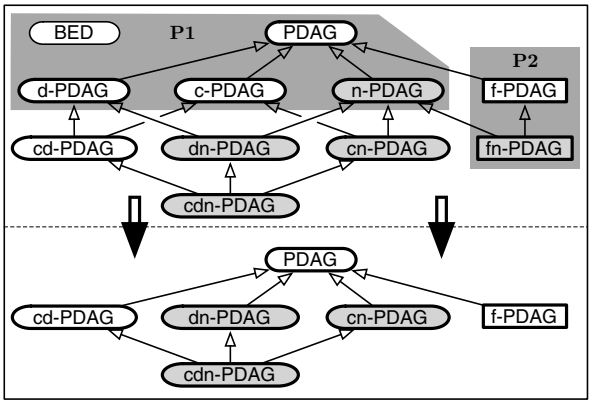

Figure 3: PDAG, d-PDAG, c-PDAG, n-PDAG, and BED are equally succinct and thus grouped into PDAG. Similarly, f-PDAG and fn-PDAG are equally succinct and thus grouped into f-PDAG.

The six remaining languages are depicted in the lower part of Fig. 3. Of these, dn-PDAG and f-PDAG will be omitted in the following discussion of succinctness, since they do not allow any polytime queries, as we will see later. The same remark holds for PDAG, but it is kept as the common roof of all considered languages. The remaining analysis will thus focus on the succinctness relations between PDAG, cd-PDAG, cn-PDAG, and cdn-PDAG as shown on the left hand side of Fig. 4.

Proposition P3. PDAG is strictly more succinct than cn-PDAG and cd-PDAG

Proposition P4. cn-PDAG is strictly more succinct than cdn-PDAG.

Proposition P5. cd-PDAG is not more succinct than cn-PDAG.

The results of P3, P4, and P5 are illustrated on the right hand side of Fig. 4. With respect to the succinctness relations between PDAG, cd-PDAG, cn-PDAG, and cdn-PDAG, two open questions remain. The first one concerns the relationship between cd-PDAG and cdn-PDAG. It follows from cd-PDAG $\supseteq$ cdn-PDAG that cd-PDAG is more succinct than cdn-PDAG, but whether cd-PDAG is strictly more succinct than cdn-PDAG is unknown. The second question concerns the relationship between cd-PDAG and cn-PDAG. P5 guarantees cd-PDAG $\not\preceq$ cn-PDAG to hold, but nothing is known about the converse question of cn-PDAG $\preceq$ cd-PDAG. This paper will not answer these questions, but they will be further discussed later. The discussion depends on the queries and the transformations that can be done in polytime with respect to the size. In the next section, we will take a closer look at the queries.

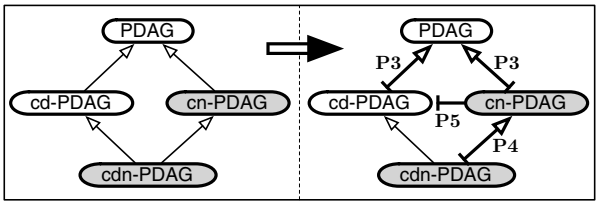

Figure 4: Succinctness relationships between PDAG, cd-PDAG, cn-PDAG, and cdn-PDAG. We use L1$\multimap$L2 for L1 $\preceq$ L2, L1$\mapsto$L2 for L1 $\not\preceq$ L2, L1$\leftrightarrow$L2 for L1 $\equiv$ L2, L1$\leftarrow\!\!\shortmid$L2 for L1 $\prec$ L2, and L1$\mapsto\!\!\shortmid$L2 for L1 $\not\preceq$ L2 and L2 $\not\preceq$ L1.

Queries

The succinctness of a language is not the only factor to consider when choosing an appropriate target compilation language for a particular application. The second crucial factor is the set of *queries* supported in polytime, and the third factor is the set of *transformations* supported in polytime, which we will discuss in the next section.

As pointed out in (Darwiche & Marquis 2002), there is a trade-off between the succinctness of the target language and the set of queries that is supported in polytime, i.e. more succinct languages support less queries in polytime, and vice versa. In other words, if language L supports a query in polytime, all sub-languages of L will also support this query in polytime. Queries can be classified into *simple* and *complex* ones. The former are *predicates* returning simply *yes/no* or 1/0, whereas the latter are general functions with more complex ranges.

In the following, $\varphi, \psi \in$ PDAG will be general PDAGs, and $\lambda_1, \ldots, \lambda_t$ will be literals. The considered predicates are the following:

- *Consistency* or *satisfiability* (SAT): $\varphi \not\equiv \bot$;
- *Validity*: $\varphi \equiv \top$;
- *Clause entailment*:, $\varphi \models \lambda_1 \vee \cdots \vee \lambda_t$;
- *Term implication*: $\lambda_1 \wedge \cdots \wedge \lambda_t \models \varphi$;
- *Sentential entailment*: $\varphi \models \psi$;
- *Equivalence*: $\varphi \equiv \psi$.

The following discussion also includes a *probabilistic equivalence check*, which operates like a normal equivalence check, but with a (arbitrarily small) failure probability

(Blum, Chandra, & Wegman 1980; Jain *et al.* 1991; Darwiche & Huang 2002; Wachter & Haenni 2006).

Clearly, the tests for consistency and validity are special cases of clause entailment and term implication, respectively. Similarly, the tests for clause entailment and term implication are special cases of sentential entailment. Finally, the equivalence test $\varphi \equiv \psi$ is the combination of two sentential entailment tests $\varphi \models \psi$ and $\psi \models \varphi$.

In addition to these predicates, we consider the following complex queries:

- *Model counting*: compute $|S_{f_\varphi}|$;
- *Probability computation*: compute the probability $p(\varphi)$ for independent marginal probabilities $p(x)$, $x \in V$;
- *Model enumeration*: list the elements of S_{f_φ};
- *Counter-model enumeration*: list the elements of S_{1-f_φ}.

Note that the tests for consistency and validity are special cases of model counting. Model counting in turn can be regarded as special case of probability computation.

If a sub-language $\mathtt{L} \in \mathtt{PDAG}$ supports a query in polytime with respect to the size $|\varphi|$ for all $\varphi \in \mathtt{L}$ (in the case of model or counter-model enumeration, the reference size is both $|\varphi|$ and $|S_{f_\varphi}|$ or $|S_{1-f_\varphi}|$), we simply say that L *supports* this query.[3] This is denoted by CO for consistency, VA for validity, CE for clause entailment, IM for term implication, SE for sentential entailment, EQ for equivalence, PEQ for probabilistic equivalence, CT for model counting, PR for probability computation, ME for model enumeration, and $\mathtt{ME}^{\mathsf{c}}$ for counter-model enumeration.

Obviously, if query Q1 is a special case of query Q2, it means that any language L supporting Q2 also supports Q1. The general case of these correlations is shown on the left hand side of Fig. 5. Since CT, PR, and PEQ are very similar problems, they are all supported as long as at least one of them is supported. This is indicated by CT/PR/PEQ.

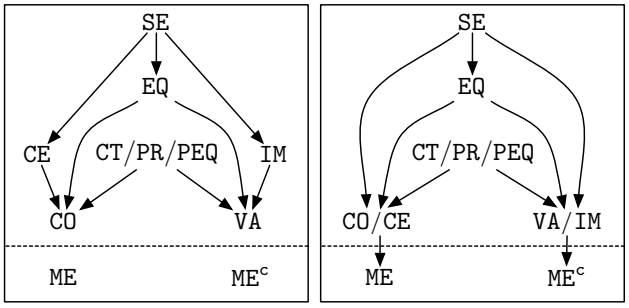

Figure 5: The correlations between supported queries: general case on the left, for languages supporting term conditioning on the right.

Further correlations are obtained if the transformation of *term conditioning* is supported (Darwiche & Marquis 2002). The definition and discussion of term conditioning follows in the next section. We will write TC to denote that a language supports term conditioning.

[3]The authors of (Darwiche & Marquis 2002) prefer to say that L *satisfies* the corresponding query.

Proposition P6. If a language supports TC and CO, then it also supports CE and ME.

Proposition P7. If a language supports TC and VA, then it also supports IM and $\mathtt{ME}^{\mathsf{c}}$.

As we will see in the next section, every single PDAG sublanguage supports TC. In (Darwiche & Marquis 2002), this has been demonstrated for all NNF (= n-PDAG) sublanguages, but it is easy to see that this also holds for all PDAG sub-languages. With this, we are now ready for the main statements of this section.

Proposition P8. PDAG, f-PDAG, and dn-PDAG do not support any query.

Proposition P9. cn-PDAG supports CO and thus CE and ME.

Proposition P10. cd-PDAG and cdn-PDAG support CT and thus all queries except EQ and SE.

With respect to possible target compilation languages, P8, P9, and P10 lead us to the conclusion that PDAG, f-PDAG, and dn-PDAG are not qualified (no queries are supported). The eligibility of the three remaining target languages cn-PDAG, cd-PDAG, and cdn-PDAG is further discussed after the next section on transformations. The following table summarizes the supported queries of the entire family of PDAG sublanguages. Again, equally succinct languages are grouped as suggested in Fig. 3. The lower part of the table corresponds to he results presented in (Darwiche & Marquis 2002).

	CO/CE/ME	VA/IM/$\mathtt{ME}^{\mathsf{c}}$	CT/PR/PEQ	EQ	SE
PDAG	○	○	○	○	○
f-PDAG	○	○	○	○	○
cd-PDAG	√	√	√	?	○
dn-PDAG / d-NNF	○	○	○	○	○
cn-PDAG / DNNF	√	○	○	○	○
cdn-PDAG / d-DNNF	√	√	√	?	○
BDD	○	○	○	○	○
FBDD	√	√	√	?	○
OBDD	√	√	√	√	○
$\mathtt{OBDD}_<$	√	√	√	√	√
DNF	√	○	○	○	○
CNF	○	√	○	○	○
PI	√	√	○	√	√
IP	√	√	○	√	√
MODS	√	√	√	√	√

Table 1: Subsets of the PDAG language and their supported queries. The symbol √ means "supports", ○ means "does not support unless P = NP", and ? means "unknown".

Transformations

The set of *transformations* supported in polytime is the third crucial factor to consider when choosing an appropriate target compilation language. As pointed out in (Darwiche & Marquis 2002), sub-languages do in general not inherit the tractability of transformations, in contrast to the queries which are inherited.

In the following, let $W \subseteq V$ be a subset of variables and $x \in V$ a single variable. Furthermore, we will use

$\varphi, \psi, \varphi_1, \ldots, \varphi_s \in$ PDAG for general PDAGs, $\lambda_1, \ldots, \lambda_t$ for literals, and τ for the term $\tau = \lambda_1 \wedge \cdots \wedge \lambda_t$. The considered transformations are:

- *Term conditioning*: $\varphi|\tau$;
- *Forgetting*: φ^{-W};
- *Singleton forgetting*: $\varphi^{-\{x\}}$;
- *Conjunction*: $\varphi_1 \wedge \cdots \wedge \varphi_s$;
- *Binary conjunction*: $\varphi \wedge \psi$;
- *Disjunction*: $\varphi_1 \vee \cdots \vee \varphi_s$;
- *Binary disjunction*: $\varphi \vee \psi$;
- *Negation*: $\neg\varphi$.

The idea of term conditioning is to transform φ and τ into a PDAG $\varphi|\tau$, such that $\tau \wedge \varphi|\tau \equiv \tau \wedge \varphi$ and $vars(\tau) \cap vars(\varphi|\tau) = \emptyset$ hold. Essentially, this means to replace each occurrence of a variable x in φ by $\top$ (resp. $\bot$), if x occurs as a positive (resp. negative) literal in τ. As pointed out in (Darwiche & Marquis 2002), term conditioning preserves all the relevant properties like decomposability, determinism, or flatness, and this is also true for simple-negation. Consequently, term conditioning is supported by the entire family of PDAG sub-languages. Note that term conditioning can be extended to a more general form of conditioning with respect to sub-PDAGs of φ instead of literals.

Forgetting W from φ generates a new PDAG φ^{-W}, in which the variables from W are no longer included. Its satisfying set $S_{\varphi^{-W}}$ is supposed be the projection of S_φ to the restricted set of variables $V \setminus W$. Singleton forgetting is forgetting with $W = \{x\}$. One way to realize singleton forgetting is by $\varphi|x \vee \varphi|\neg x$, whereas forgetting in general is realized by a sequence of singleton forgetting. In the literature, forgetting was originally called *elimination of middle terms* (Boole 1854), but it is also common to call it *projection*, *variable elimination*, or *marginalization* (Kohlas 2003).

If a sub-language L $\in$ PDAG supports a transformation in polytime with respect to the size $|\varphi|$ for all $\varphi \in$ L, we simply say that L *supports* this transformation.[4] This is denoted by TC for term conditioning, FO for forgetting, SFO for singleton forgetting, AND for conjunction, AND_2 for binary conjunction, OR for disjunction, OR_2 for binary disjunction, and NOT for negation. In the following, we will restrict the discussion of supported transformations to PDAG, f-PDAG, and cd-PDAG. All other languages are extensively discussed in (Darwiche & Marquis 2002).

Proposition P11. PDAG supports all transformations except FO (unless P = NP).

Proposition P12. f-PDAG supports SFO and NOT, but it does not support FO (unless P = NP), AND, AND_2, OR, and OR_2.

Proposition P13. cd-PDAG supports NOT, but unless P = NP, it does not support FO, SFO, AND, AND_2, OR, and OR_2.

Table 2 summarizes the supported transformations of all considered PDAG sub-languages. Again, equally succinct languages are grouped according to the suggestion in Fig. 3.

[4]The authors of (Darwiche & Marquis 2002) prefer to say that L it is *closed under* the corresponding operator.

	TC	FO	SFO	AND	AND_2	OR	OR_2	NOT
PDAG	√	○	√	√	√	√	√	√
f-PDAG	√	○	√	•	•	•	•	√
cd-PDAG	√	○	○	○	○	○	○	√
dn-PDAG / d-NNF	√	○	√	√	√	√	√	√
cn-PDAG / DNNF	√	√	√	○	○	√	√	○
cdn-PDAG / d-DNNF	√	○	○	○	○	○	○	?
BDD	√	○	√	√	√	√	√	√
FBDD	√	•	○	•	○	•	○	√
OBDD	√	•	√	•	○	•	○	√
$\text{OBDD}_<$	√	•	√	•	√	•	√	√
DNF	√	√	√	•	√	√	√	•
CNF	√	○	√	√	√	•	√	•
PI	√	√	√	•	•	•	√	•
IP	√	•	•	•	√	•	•	•
MODS	√	√	√	•	√	•	•	•

Table 2: Subsets of the PDAG language and their supported transformations. √ means "supports", • means "does not support", and ○ means "does not support unless P = NP."

Discussion

To make further statements on how to choose an appropriate target compilation language, we need to consider the succinctness relationships shown on the right hand side of Fig. 4, the supported queries shown in Tab. 1, and the supported transformations shown in Tab. 2. Recall that there are two open questions related to succinctness. We are unable to answer them here, but the following discussion of all possible scenarios will finally allow us to favor one language over the other. These scenarios are shown in Figure 6. Note that the scenario of Fig. 6d is self-inconsistent[5] and therefore impossible.

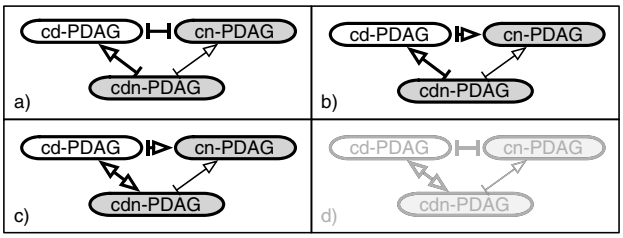

Figure 6: Possible succinctness relationships between PDAG, cd-PDAG, cn-PDAG, and cdn-PDAG.

With the exception that EQ is still an open question for both cd-PDAG and cdn-PDAG (= d-DNNF), we conclude from Tab. 1 that they support exactly the same set of queries. According to Tab. 2, they also support the same transformations, except that NOT is unknown in the case of cdn-PDAG. This is a potential advantage of cd-PDAG over cdn-PDAG. The negation of $\varphi \in$ cd-PDAG is always a trivial operation: If φ is a $\diamond$-node with child ψ, the negation of φ is simply ψ, otherwise the it is a $\diamond$-node with child φ. Independently of whether NOT is supported by cdn-PDAG or not, it is clear

[5]From cd-PDAG $\equiv$ cdn-PDAG and cn-PDAG $\prec$ cdn-PDAG we conclude that cn-PDAG $\prec$ cd-PDAG.

that no such trivial negation exists. This is a definite plus of cd-PDAG, making it a much more flexible language.

There is some evidence that neither cd-PDAG nor cdn-PDAG supports EQ, but a definite proof is still missing. Let us first exclude this problem from the discussion about the choice of the most appropriate target compilation language. This allows us to argue as follows:

- If only queries from the set $\{\mathtt{CO}, \mathtt{CE}, \mathtt{ME}\}$ are needed, then cn-PDAG (= d-DNNF) is the most appropriate target language. In Fig. 6b and Fig. 6c, cn-PDAG is strictly more succinct than cd-PDAG. In Fig. 6a, cn-PDAG and cd-PDAG are incomparable, but cn-PDAG is still preferred, since it supports more transformations. As long as cn-PDAG $\preceq$ PI is unknown (Darwiche & Marquis 2002), another candidate for these queries is PI.
- If not only queries from $\{\mathtt{CO}, \mathtt{CE}, \mathtt{ME}\}$ but also from $\{\mathtt{VA}, \mathtt{IM}, \mathtt{CT}, \mathtt{PR}, \mathtt{PEQ}, \mathtt{MEC}\}$ are needed, then cn-PDAG (and PI) is no longer appropriate. But then cd-PDAG is the most appropriate target language. In Fig. 6a and Fig. 6b, cd-PDAG is strictly more succinct than cdn-PDAG. In Fig. 6c, cd-PDAG and cdn-PDAG are equally succinct, but cd-PDAG is still preferred, since it supports NOT.

Now let the equivalence check be an additionally required query. If we accept PEQ instead of EQ, the discussion could stop here, because PEQ is supported by both cd-PDAG and cdn-PDAG. For the same reasons as above, cd-PDAG would then be the preferred language. But what happens if EQ is indispensable?

In the most likely scenario, i.e. when neither cd-PDAG nor cdn-PDAG supports EQ, one should select OBDD, the most succinct language supporting all required queries (Darwiche & Marquis 2002). If we assume that EQ is supported by both cd-PDAG and cdn-PDAG, then cd-PDAG should be selected instead of OBDD. Finally, if we suppose that EQ is supported by cdn-PDAG but not by cd-PDAG, then cdn-PDAG should be selected, since it is strictly more succinct than OBDD. This scenario is rather unlikely, but it is the only one where cdn-PDAG (= d-DNNF) does not drop out as target language.

Finally, if we require all queries to be supported, including a polytime test for sentential entailment, then $\mathtt{OBDD}_<$ is the most succinct appropriate language.

Conclusion

By removing the implicit simple-negation property of NNFs, this paper extends the family of graph-based languages for representing Boolean functions proposed in (Darwiche & Marquis 2002). This leads to an extended knowledge compilation map, which helps a knowledge engineer to decide by means of two opposing criteria which kind of language he should use. The two criteria are the succinctness of the language and the sets of supported queries and transformations.

It turns out that most queries are either supported by cn-PDAG or cd-PDAG. Concerning the problem of choosing an appropriate target language, these are thus the most interesting languages among all. Other interesting languages are OBDD for equivalence checking and $\mathtt{OBDD}_<$ for sentence entailment. All other languages should not be further considered as possible target compilation languages. While cn-PDAG (= DNNF) was first proposed in (Darwiche 2001a), cd-PDAG is a new language. This is the main contribution of this paper, and we suggest that cdn-PDAG (= d-DNNF) should be entirely replaced by cd-PDAG.

Besides the potential of being strictly more succinct, the main advantage of cd-PDAG over cdn-PDAG is the simplicity of the negation and the resulting flexibility, which allows the complement of a Boolean function to be represented with one additional node only, i.e. with almost no computational overhead. This can be useful in many ways, but in particular for building more general and more efficient compilers.

As an example, consider Darwiche's CNF to d-DNNF compiler (Darwiche 2002). Of course, we can use it one-to-one as a CNF to cd-PDAG compiler, but it can also be used as a DNF to cd-PDAG compiler. The idea is to first transform the negation $\neg\varphi$ of $\varphi \in$ DNF into $\overline{\varphi} \in$ CNF (using De Morgan's laws), then apply Darwiche's CNF to d-DNNF compiler on $\overline{\varphi}$, and finally add a $\diamond$-node on top of the result. Building efficient cd-PDAG compilers will an important topic of our future research activity.

Another future project is to resolve the open questions related to succinctness, EQ, and NOT. These questions are mainly of theoretical interest, but conclusive responses would still be useful to further underline our suggestions concerning the choice of the most appropriate target compilation language.

Finally, we will have to carefully explore the possible applications and consequences of using theses languages (in particular cd-PDAG) in areas like formal verification, Bayesian networks, reliability theory, diagnostics, planning, and many more. Some work is currently in progress and preliminary results are promising.

Proofs

Most proofs for NNFs given in (Darwiche & Marquis 2002) can be easily adapted to prove the propositions P1 to P13.

Proof of Proposition P1. Every BED can be easily transformed into a digital circuit and vice versa. Since every digital circuit can be regarded as a PDAG and vice versa, BED and PDAG are equally succinct. Since d-PDAG, c-PDAG, and n-PDAG are sub-languages of PDAG, we only have to show that the elements of PDAG have all a polynomial representation in d-PDAG, c-PDAG and n-PDAG.

A deterministic PDAG is obtained from $\varphi \in$ PDAG by replacing each $\triangledown$-node ψ with children $\psi_1, \ldots, \psi_s$ by a $\diamond$-node, whose child is a $\triangle$-node with children $\neg\psi_1, \ldots, \neg\psi_s$.

Similarly, a decomposable PDAG is obtained from $\varphi \in$ PDAG by replacing each $\triangle$-node ψ with children $\psi_1, \ldots, \psi_s$ by a $\diamond$-node whose child is a $\triangledown$-node with children $\neg\psi_1, \ldots, \neg\psi_s$.

Finally, a NNF is obtained from $\varphi \in$ PDAG by De Morgan's law, i.e. the following recursive replacement of each $\diamond$-node ψ with child π. If π is a $\diamond$-node with child σ, ψ is replaced by σ. If π is a $\triangle$-node with children $\sigma_1, \ldots, \sigma_s$, we replace it by a $\triangledown$-node with children $\neg\sigma_1, \ldots, \neg\sigma_s$. Finally,

if π is a $\bigtriangledown$-node with children $\sigma_1, \ldots, \sigma_s$, we replace it by a $\bigtriangleup$-node with children $\neg\sigma_1, \ldots, \neg\sigma_s$. Note that the time for this replacement is polynomial in the size of φ.

In all cases, the size of the involved PDAG changes only by a constant factor. □

Proof of Proposition P2. According to the proof of P1, obtaining simple-negation changes the size only by a constant factor and a close look reveals that it does not increase the height. □

Proof of Proposition P3. The proof that PDAG is strictly more succinct than cn-PDAG (= DNNF) is given in the proof of Proposition 3.1, Table 12, in (Darwiche & Marquis 2002). The same proof also holds for cd-PDAG. □

Proof of Proposition P4. The proof that cn-PDAG (= DNNF) is strictly more succinct than cdn-PDAG (= d-DNNF) is given in the proof of Proposition 3.1, Table 14, in (Darwiche & Marquis 2002). □

Proof of Proposition P5. Since the Lemmas A.6 and Lemma A.11 from (Darwiche & Marquis 2002) also hold for cd-PDAG, the proof of P4 also holds for cd-PDAG. □

Proof of Proposition P6. This follows from the fact that Lemma A.3 and Lemma A.4 from (Darwiche & Marquis 2002) also hold for PDAG. □

Proof of Proposition P7. Since Lemma A.7 from (Darwiche & Marquis 2002) also holds for PDAG, we only have to show that $\mathtt{ME}^{\mathtt{C}}$ is supported, when both TC and VA are supported. This follows from the proof of Lemma A.3 by replacing "$\Sigma|\alpha\wedge x_i$ is consistent" by "$\Sigma|\alpha\wedge x_i$ is not valid" and "$\Sigma|\alpha \wedge \neg x_i$ is consistent" by "$\Sigma|\alpha \wedge \neg x_i$ is not valid". □

Proof of Proposition P8. This proof is analogue to the proofs for NNF, f-NNF, and d-NNF in (Darwiche & Marquis 2002). □

Proof of Proposition P9. Since cn-PDAG = DNNF, this is already shown in (Darwiche & Marquis 2002). □

Proof of Proposition P10. If cd-PDAG would support SE, d-DNNF would also support SE, but is does not. EQ remains unknown for cd-PDAG (as for d-DNNF). Model counting for $\varphi \in$ cd-PDAG, denoted by $count(\varphi)$, is done in the following way: $count(\varphi) =$

$$\begin{cases} \prod_{i=1}^{n} count(\psi_i), \text{ if } \varphi \text{ is a } \bigtriangleup\text{-node with children } \psi_i; \\ \sum_{i=1}^{n} count(\psi_i), \text{ if } \varphi \text{ is a } \bigtriangledown\text{-node with children } \psi_i, \\ 2^{|vars(\varphi)|} - count(\psi), \text{ if } \varphi \text{ is a } \diamond\text{-node with child } \psi; \\ 2, \text{ if } \varphi \text{ is a } \bigcirc\text{-node labeled with } \top; \\ 1, \text{ if } \varphi \text{ is a } \bigcirc\text{-node labeled with } x \in V; \\ 0, \text{ if } \varphi \text{ is a } \bigcirc\text{-node labeled with } \bot. \end{cases}$$

This definition is similar to the one for d-DNNF in (Darwiche 2001b), except for the treatment of negations. The other queries follow from the correlations between supported queries (right hand side of Fig. 5).

□

Proof of Proposition P11. Analogue to the proof of Proposition 5.1 for NNF in (Darwiche & Marquis 2002). □

Proof of Proposition P12. Except for the proof of SFO, this proof is analogue to the one of Proposition 5.1 for f-NNF in (Darwiche & Marquis 2002). The proof of SFO for f-NNF has ignored the fact that for $\varphi \in$ f-NNF the conjunction $\varphi|x \wedge \varphi|\neg x$ concerns either two DNFs or two CNFs. Since both DNF and CNF support $\mathtt{AND}_2$, SFO is supported by f-NNF. f-PDAG supports SFO because f-PDAG and f-NNF (= fn-PDAG) are equally succinct, and the time for the transformation from f-PDAG to f-NNF is polynomial in the size of the PDAG. □

Proof of Proposition P13. Except that cd-PDAG obviously supports NOT, this proof is analogue to the one of Proposition 5.1 for d-DNNF (Darwiche & Marquis 2002). □

Acknowledgments

This research is supported by the Swiss National Science Foundation, Project No. PP002102652/1. Special thanks to the anonymous reviewers, Jacek Jonczy, and Markus Gälli for careful proof-reading and comments.

References

Akers, S. B. 1978. Binary decision diagrams. *IEEE Transactions on Computers* 27(6):509—516.

Andersen, H. R., and Hulgaard, H. 1997. Boolean expression diagrams. In Winskel, G., ed., *LICS'97, 12th Annual IEEE Symposim on Logic in Computer Science*, 88–98. Warsaw, Poland: IEEE Computer Society.

Blum, M.; Chandra, A. K.; and Wegman, M. N. 1980. Equivalence of free boolean graphs can be decided probabilistically in polynomial time. *Information Processing Letters* 10(2):80–82.

Boole, G. 1854. *The Laws of Thought*. London: Walton and Maberley.

Bryant, R. E. 1986. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers* 35(8):677–691.

Bryant, R. E. 1992. Symbolic Boolean manipulation with ordered binary decision diagrams. *ACM Computing Surveys* 24(3):293–318.

Cadoli, M., and Donini, F. 1997. A survey on knowledge compilation. *AI Communications* 10(3–4):137–150.

Clote, P., and Kranakis, E. 1998. *Boolean Functions and Computation Models*. Springer.

Darwiche, A., and Huang, J. 2002. Testing equivalence probabilistically. Technical Report D–123, Computer Science Department, UCLA, Los Angeles, USA.

Darwiche, A., and Marquis, P. 2002. A knowlege compilation map. *Journal of Artificial Intelligence Research* 17:229–264.

Darwiche, A. 1999. Compiling knowledge into decomposable negation normal form. In *IJCAI'99, 16th International Joint Conference on Artificial Intelligence*, 284–289.

Darwiche, A. 2001a. Decomposable negation normal form. *Journal of ACM* 48(4):608–647.

Darwiche, A. 2001b. On the tractability of counting theory models and its application to belief revision and truth maintenance. *Journal of Applied Non-Classical Logics* 11(1-2):11–34.

Darwiche, A. 2002. A compiler for deterministic, decomposable negation normal form. In *AAAI'02, 18th National Conference on Artificial Intelligence*, 627–634. AAAI Press.

Gross, J. L., and Yellen, J. 2003. *Handbook of Graph Theory*. CRC Press.

Hill, F. J., and Peterson, G. R. 1974. *Introduction to Switching Theory and Logical Design*. New York, USA: John Wiley and Sons.

Jain, J.; Bitner, J.; Fussell, D. S.; and Abraham, J. A. 1991. Probabilistic design verification. In *ICCAD'91, International Conference on Computer-Aided Design*, 468–471.

Kohlas, J. 2003. *Information Algebras: Generic Stuctures for Inference*. London: Springer.

Papadimitriou, C. 1994. *Computational Complexity*. Addison-Wesley.

Rai, S.; Veeraraghavan, M.; and Trivedi, K. S. 1995. A survey on efficient computation of reliability using disjoint products approach. *Networks* 25(3):147–163.

Rauzy, A.; Châtelet, E.; Dutuit, Y.; and Brenguer, C. 2003. A practical comparison of methods to assess sum-of-products. *Reliability Engineering and System Safety* 79(1):33–42.

Verma, T. S., and Pearl, J. 1992. An algorithm for deciding if a set of observed independencies has a causal explanation. In Dubois, D., and Wellman, M. P., eds., *UAI'92, 8th Conference on Uncertainty in Artificial Intelligence*, 323–330.

Wachter, M., and Haenni, R. 2006. Probabilistic equivalence checking with propositional DAGs (submitted). In *CAV'06, 18th Conference on Computer-Aided Verification*.