Hypergraph Partitioning for Compiling Pseudo-Boolean Formulae

Romain Wallon
ROADEF’21, Session Partitionnement des Graphes – April 29, 2021
Laboratoire d’Informatique de l’X (LIX), École Polytechnique, X-Uber Chair
We consider Boolean (i.e., \{0, 1\}) variables to represent knowledge.
We consider Boolean (i.e., \{0, 1\}) variables to represent knowledge.

Most of the time, knowledge is encoded using *conjunctions of clauses*, a.k.a. CNF formulae.

\[(a \lor \bar{b}) \land (\bar{a} \lor c) \land (b \lor \bar{d} \lor \bar{e}) \land (\bar{b} \lor e \lor f)\]
Symbolic AI and Boolean Reasoning

We consider Boolean (i.e., \{0, 1\}) variables to represent knowledge. Most of the time, knowledge is encoded using *conjunctions of clauses*, a.k.a. CNF formulae.

\[(a \lor \bar{b}) \land (\bar{a} \lor c) \land (b \lor \bar{d} \lor \bar{e}) \land (\bar{b} \lor e \lor f)\]

The problem is often to check whether such a formula is *satisfiable*, i.e., has a solution.
It is often convenient to use *(hyper)*graph representations of CNF Formulae to get some information about the *structure* of the formula.
It is often convenient to use (hyper)graph representations of CNF Formulae to get some information about the structure of the formula.

\[
\begin{align*}
\gamma_1 & \equiv a \lor \bar{b} & \gamma_2 & \equiv \bar{a} \lor c & \gamma_3 & \equiv b \lor \bar{d} \lor \bar{e} & \gamma_4 & \equiv \bar{b} \lor e \lor f
\end{align*}
\]
Dual Hypergraph of CNF Formulae

It is often convenient to use (hyper)graph representations of CNF Formulae to get some information about the structure of the formula

\[
\gamma_1 \equiv a \lor \bar{b} \quad \gamma_2 \equiv \bar{a} \lor c \quad \gamma_3 \equiv b \lor \bar{d} \lor \bar{e} \quad \gamma_4 \equiv \bar{b} \lor e \lor f
\]

Its dual hypergraph has as hypervertices the clauses of the formula and as hyperedges the variables of this formula
Dual Hypergraph of CNF Formulae

It is often convenient to use (hyper)graph representations of CNF Formulae to get some information about the structure of the formula

\[ \gamma_1 \equiv a \lor \overline{b} \quad \gamma_2 \equiv \overline{a} \lor c \quad \gamma_3 \equiv b \lor \overline{d} \lor \overline{e} \quad \gamma_4 \equiv \overline{b} \lor e \lor f \]

Its dual hypergraph has as hypervertices the clauses of the formula and as hyperedges the variables of this formula.

A hyperedge covers the clauses containing the corresponding variable.
Dual Hypergraph of CNF Formulae

It is often convenient to use (hyper)graph representations of CNF Formulae to get some information about the structure of the formula

\[ \gamma_1 \equiv a \lor \overline{b} \quad \gamma_2 \equiv \overline{a} \lor c \quad \gamma_3 \equiv b \lor \overline{d} \lor \overline{e} \quad \gamma_4 \equiv \overline{b} \lor e \lor f \]

Its dual hypergraph has as hypervertices the clauses of the formula and as hyperedges the variables of this formula.

A hyperedge covers the clauses containing the corresponding variable.
Reasoning on CNF Formulae and Limitations

The main reason for using CNF formulae to represent knowledge is that we can use SAT solvers to reason with such formulae.
Reasoning on CNF Formulae and Limitations

The main reason for using CNF formulae to represent knowledge is that we can use SAT solvers to reason with such formulae.

Modern SAT solvers (Silva and Sakallah, 1996; Moskewicz et al., 2001; Eén and Sörensson, 2004) are very efficient in practice.
The main reason for using CNF formulae to represent knowledge is that we can use SAT solvers to reason with such formulae. Modern SAT solvers (Silva and Sakallah, 1996; Moskewicz et al., 2001; Eén and Sörensson, 2004) are very efficient in practice. However, they do not offer time guarantees when given an input to solve.
The main reason for using CNF formulae to represent knowledge is that we can use SAT solvers to reason with such formulae.

Modern SAT solvers (Silva and Sakallah, 1996; Moskewicz et al., 2001; Eén and Sörensson, 2004) are very efficient in practice. However, they do not offer time guarantees when given an input to solve. For some applications, especially those involving interactions with users, this is not acceptable.
The main reason for using CNF formulae to represent knowledge is that we can use SAT solvers to reason with such formulae. Modern SAT solvers (Silva and Sakallah, 1996; Moskewicz et al., 2001; Eén and Sörensson, 2004) are very efficient in practice. However, they do not offer time guarantees when given an input to solve. For some applications, especially those involving interactions with users, this is not acceptable.

*In such cases, it may be interesting to rely on knowledge compilation*
Given a formula written in a specific language (e.g., CNF), some operations may be too expensive in practice to be performed online.
Given a formula written in a specific language (e.g., CNF), some operations may be too expensive in practice to be performed online.

*Compiling a formula is translating it (offline) into another language to obtain an equivalent formula on which performing the wanted (online) operations is easier.*
The language of d-DNNF is the language of deterministic and decomposable NNF.
The language of d-DNNF is the language of deterministic and decomposable NNF.

NNF is the language of Boolean circuits in Negation Normal Form, in which negations are only applied on variables.
The language of d-DNNF is the language of deterministic and decomposable NNF.

NNF is the language of Boolean circuits in Negation Normal Form, in which negations are only applied on variables.

Deterministic means that, for any disjunction \( \varphi \lor \psi \), there is no common model between \( \varphi \) and \( \psi \) (i.e., \( \varphi \land \psi \models \bot \)).
The language of d-DNNF is the language of deterministic and decomposable NNF.

NNF is the language of Boolean circuits in Negation Normal Form, in which negations are only applied on variables.

Deterministic means that, for any disjunction $\varphi \lor \psi$, there is no common model between $\varphi$ and $\psi$ (i.e., $\varphi \land \psi \models \bot$).

Decomposable means that, for each conjunction $\varphi \land \psi$, there is no common variable between $\varphi$ and $\psi$ (i.e., $\text{var}(\varphi) \cap \text{var}(\psi) = \emptyset$).
The language of d-DNNF is the language of deterministic and decomposable NNF.

NNF is the language of Boolean circuits in Negation Normal Form, in which negations are only applied on variables.

Deterministic means that, for any disjunction $\varphi \lor \psi$, there is no common model between $\varphi$ and $\psi$ (i.e., $\varphi \land \psi \models \bot$).

Decomposable means that, for each conjunction $\varphi \land \psi$, there is no common variable between $\varphi$ and $\psi$ (i.e., $\text{var}(\varphi) \cap \text{var}(\psi) = \emptyset$).

These two properties allow the efficient computation of different queries.
Ensuring Determinism

To ensure determinism, each disjunction node in the circuit will be a decision node.
To ensure **determinism**, each disjunction node in the circuit will be a **decision** node.

For more readability, we will represent the decision node above as

\[
\begin{align*}
\text{\textcircled{x} } & \quad A \\
\text{\textcircled{x} } & \quad B
\end{align*}
\]
Ensuring Determinism

To ensure determinism, each disjunction node in the circuit will be a decision node.

For more readability, we will represent the decision node above as

The d-DNNFs we obtain in this case are called Decision-DNNF.
To ensure decomposability, a partition of the dual hypergraph of the CNF to compile is computed, to extract independent connected components.
To ensure decomposability, a partition of the dual hypergraph of the CNF to compile is computed, to extract independent connected components. This operation yields a cutset, which is a set of hyperedges (i.e., variables) that must be removed (i.e., assigned) to get disjoint components.
To ensure decomposability, a partition of the dual hypergraph of the CNF to compile is computed, to extract independent connected components. This operation yields a cutset, which is a set of hyperedges (i.e., variables) that must be removed (i.e., assigned) to get disjoint components. By construction, each connected component do not share variables.
Ensuring Decomposability

To ensure decomposability, a partition of the dual hypergraph of the CNF to compile is computed, to extract independent connected components. This operation yields a cutset, which is a set of hyperedges (i.e., variables) that must be removed (i.e., assigned) to get disjoint components.

By construction, each connected component do not share variables.

*The connected components can then be compiled independently, before adding their conjunction to the build d-DNNF.*
Compiling our CNF Formula

\[(a \lor \overline{b}) \land (\overline{a} \lor c) \land (b \lor \overline{d} \lor \overline{e}) \land (\overline{b} \lor e \lor f)\]
Compiling our CNF Formula

\[(a \lor \overline{b}) \land (\overline{a} \lor c) \land (b \lor \overline{d} \lor \overline{e}) \land (\overline{b} \lor e \lor f)\]
Compiling our CNF Formula

\[(a \lor \neg b) \land (\neg a \lor c) \land (b \lor \neg d \lor \neg e) \land (\neg b \lor e \lor f)\]
Compiling our CNF Formula

\[(a \lor \bar{b}) \land (\bar{a} \lor c) \land (b \lor \bar{d} \lor \bar{e}) \land (\bar{b} \lor e \lor f)\]
Compiling our CNF Formula

$$(a \lor \overline{b}) \land (\overline{a} \lor c) \land (b \lor \overline{d} \lor \overline{e}) \land (\overline{b} \lor e \lor f)$$
Compiling our CNF Formula

\[(a \lor \overline{b}) \land (\overline{a} \lor c) \land (b \lor \overline{d} \lor \overline{e}) \land (\overline{b} \lor e \lor f)\]
Compiling our CNF Formula

\[(a \lor \overline{b}) \land (\overline{a} \lor c) \land (b \lor \overline{d} \lor \overline{e}) \land (\overline{b} \lor e \lor f)\]
Compiling our CNF Formula

\[(a \lor \bar{b}) \land (\bar{a} \lor c) \land (b \lor \bar{d} \lor \bar{e}) \land (\bar{b} \lor e \lor f)\]
Compiling our CNF Formula

\[(a \lor \bar{b}) \land (\bar{a} \lor c) \land (b \lor \bar{d} \lor \bar{e}) \land (\bar{b} \lor e \lor f)\]
Compiling our CNF Formula

\((a \lor \overline{b}) \land (\overline{a} \lor c) \land (b \lor \overline{d} \lor \overline{e}) \land (\overline{b} \lor e \lor f)\)
Compiling our CNF Formula

\[(a \lor \overline{b}) \land (\overline{a} \lor c) \land (b \lor \overline{d} \lor \overline{e}) \land (\overline{b} \lor e \lor f)\]
Compiling our CNF Formula

\[(a \lor \overline{b}) \land (\overline{a} \lor c) \land (b \lor \overline{d} \lor \overline{e}) \land (\overline{b} \lor e \lor f)\]
Compiling our CNF Formula

\((a \lor \overline{b}) \land (\overline{a} \lor c) \land (b \lor \overline{d} \lor \overline{e}) \land (\overline{b} \lor e \lor f)\)
Compiling our CNF Formula

\[(a \lor \bar{b}) \land (\bar{a} \lor c) \land (b \lor \bar{d} \lor \bar{e}) \land (\bar{b} \lor e \lor f)\]
Compiling our CNF Formula

\[(a \lor \bar{b}) \land (\bar{a} \lor c) \land (b \lor \bar{d} \lor \bar{e}) \land (\bar{b} \lor e \lor f)\]
Impact of the Quality of the Partition

Finding a good partition is crucial for compiling the input into a small Decision-DNNF
Impact of the Quality of the Partition

Finding a **good** partition is crucial for compiling the input into a **small** Decision-DNNF
Finding a **good** partition is crucial for compiling the input into a **small** Decision-DNNF.

Ideally, we need **small** cutsets and **balanced** partitions.
Outline of *D4* (Lagniez and Marquis, 2017)

1. Invoke a **SAT Solver** on the input
2. If the formula is **UNSAT**, then the compiled form is \(\bot\)
3. If all variables are assigned, then the compiled form is \(\top\)
4. For each **connected component** \(\varphi\) of the formula:
   a. Choose a variable \(v\) based on a **cutset** of \(\varphi\) computed with **PaToH**
      (Çatalyürek and Aykanat, 2011)
   b. Compile \(\varphi|v\) as \(\varphi_v\)
   c. Compile \(\varphi|\overline{v}\) as \(\varphi_{\overline{v}}\)
   d. The compiled form of \(\varphi\) is \(\text{ite}(v, \varphi_v, \varphi_{\overline{v}})\)
5. The compiled form is the conjunction of the compiled forms obtained above
Outline of \textit{D4} (Lagniez and Marquis, 2017)

1. Invoke a \textbf{SAT Solver} on the input
2. If the formula is \textbf{UNSAT}, then the compiled form is $\bot$
3. If all variables are assigned, then the compiled form is $\top$
4. For each \textbf{connected component} $\varphi$ of the formula:
   a. Choose a variable $v$ based on a \textbf{cutset} of $\varphi$ computed with \textbf{PaToH}
      (Çatalyürek and Aykanat, 2011)
   b. Compile $\varphi|_v$ as $\varphi_v$
   c. Compile $\varphi|_{\bar{v}}$ as $\varphi_{\bar{v}}$
   d. The compiled form of $\varphi$ is $\text{ite}(v, \varphi_v, \varphi_{\bar{v}})$
5. The compiled form is the conjunction of the compiled forms obtained above

\textit{D4} is available at \url{https://github.com/crillab/d4}
SAT Solver Limitations

The algorithm we presented uses SAT solvers as oracles to benefit from their practical efficiency.
The algorithm we presented uses SAT solvers as oracles to benefit from their practical efficiency.

However, some instances remain completely out of reach for modern SAT solvers, especially when counting capabilities are required.
The algorithm we presented uses SAT solvers as oracles to benefit from their practical efficiency.

However, some instances remain completely out of reach for modern SAT solvers, especially when counting capabilities are required.

For instance, SAT solvers cannot prove efficiently that “n pigeons do not fit in n − 1 holes” (Haken, 1985).
The algorithm we presented uses SAT solvers as oracles to benefit from their practical efficiency.

However, some instances remain completely out of reach for modern SAT solvers, especially when counting capabilities are required.

For instance, SAT solvers cannot prove efficiently that “$n$ pigeons do not fit in $n - 1$ holes” (Haken, 1985).

On such instances, pseudo-Boolean reasoning can offer better performance.
Pseudo-Boolean (PB) Constraints

PB solvers are generalizations of SAT solvers that allow to consider

- normalized PB constraints \( \sum_{i=1}^{n} \alpha_i \ell_i \geq \delta \)
- cardinality constraints \( \sum_{i=1}^{n} \ell_i \geq \delta \)
- clauses \( \sum_{i=1}^{n} \ell_i \geq 1 \)

in which

- the coefficients \( \alpha_i \) are non-negative integers
- \( \ell_i \) are literals, i.e., a variable \( v \) or its negation \( \bar{v} = 1 - v \)
- the degree \( \delta \) is a non-negative integer
Pseudo-Boolean (PB) Constraints

PB solvers are generalizations of SAT solvers that allow to consider

- normalized PB constraints \( \sum_{i=1}^{n} \alpha_i \ell_i \geq \delta \)
- cardinality constraints \( \sum_{i=1}^{n} \ell_i \geq \delta \)
- clauses \( \sum_{i=1}^{n} \ell_i \geq 1 \)

in which

- the coefficients \( \alpha_i \) are non-negative integers
- \( \ell_i \) are literals, i.e., a variable \( v \) or its negation \( \bar{v} = 1 - v \)
- the degree \( \delta \) is a non-negative integer

PB constraints allow in general more succinct encodings than CNF, and are often more natural to use.
To illustrate the succinctness of PB constraints compared to CNF, consider the cardinality constraint

\[ a + b + c + d + e \geq 3 \]
To illustrate the succinctness of PB constraints compared to CNF, consider the cardinality constraint

\[ a + b + c + d + e \geq 3 \]

Its CNF encoding is given by

\[ (a \lor b \lor c) \land (a \lor b \lor d) \land (a \lor b \lor e) \land (a \lor c \lor d) \land (a \lor c \lor e) \land (a \lor d \lor e) \land (b \lor c \lor d) \land (b \lor c \lor e) \land (b \lor d \lor e) \land (c \lor d \lor e) \]
To illustrate the succinctness of PB constraints compared to CNF, consider the cardinality constraint

\[ a + b + c + d + e \geq 3 \]

Its CNF encoding is given by

\[
(a \lor b \lor c) \land (a \lor b \lor d) \land (a \lor b \lor e) \land (a \lor c \lor d) \land (a \lor c \lor e) \\
\land (a \lor d \lor e) \land (b \lor c \lor d) \land (b \lor c \lor e) \land (b \lor d \lor e) \land (c \lor d \lor e)
\]

In general, PB representations may be exponentially smaller than CNF representations.
A first advantage of the native support of PB constraints for knowledge compilation is that PB representations may be more succinct and more natural than CNF representations.
A first advantage of the native support of PB constraints for knowledge compilation is that PB representations may be more succinct and more natural than CNF representations.

They allow to consider instances that are too big when represented in CNF to be compiled.
A first advantage of the native support of PB constraints for knowledge compilation is that PB representations may be more succinct and more natural than CNF representations. They allow to consider instances that are too big when represented in CNF to be compiled.

PB solvers may find exponentially shorter proofs while inheriting many of the efficient techniques developed in SAT solvers.
A first advantage of the native support of PB constraints for knowledge compilation is that PB representations may be more succinct and more natural than CNF representations.

They allow to consider instances that are too big when represented in CNF to be compiled.

PB solvers may find exponentially shorter proofs while inheriting many of the efficient techniques developped in SAT solvers.

Using such solvers as oracles may allow to speed up the compilation time.
A first advantage of the native support of PB constraints for knowledge compilation is that PB representations may be more succinct and more natural than CNF representations. They allow to consider instances that are too big when represented in CNF to be compiled.

PB solvers may find exponentially shorter proofs while inheriting many of the efficient techniques developed in SAT solvers. Using such solvers as oracles may allow to speed up the compilation time.

Supporting PB constraints only requires to extend the existing algorithm, without forcing to redesign a completely new approach.
A first advantage of the native support of PB constraints for knowledge compilation is that PB representations may be more succinct and more natural than CNF representations.

They allow to consider instances that are too big when represented in CNF to be compiled.

PB solvers may find exponentially shorter proofs while inheriting many of the efficient techniques developed in SAT solvers.

Using such solvers as oracles may allow to speed up the compilation time.

Supporting PB constraints only requires to extend the existing algorithm, without forcing to redesign a completely new approach.

To support PB compilation, one basically needs to replace by a PB solver the SAT solver used in the compilation procedure.
The dual hypergraph of a PB formula is defined as for CNF formulae.
The dual hypergraph of a PB formula is defined as for CNF formulae

\[ \chi_1 \equiv a + \bar{b} \geq 1 \quad \chi_2 \equiv \bar{a} + c \geq 1 \quad \chi_3 \equiv b + d + \bar{e} \geq 2 \quad \chi_4 \equiv 2\bar{b} + e + f \geq 3 \]
The dual hypergraph of a PB formula is defined as for CNF formulae

\[ \chi_1 \equiv a + \overline{b} \geq 1 \quad \chi_2 \equiv \overline{a} + c \geq 1 \quad \chi_3 \equiv b + \overline{d} + \overline{e} \geq 2 \quad \chi_4 \equiv 2\overline{b} + e + f \geq 3 \]

Its dual hypergraph has as hypervertices the constraints of the formula and as hyperedges the variables of this formula.

A hyperedge covers the constraints containing the corresponding variable.
The dual hypergraph of a PB formula is defined as for CNF formulae

\[
\chi_1 \equiv a + \bar{b} \geq 1 \quad \chi_2 \equiv \bar{a} + c \geq 1 \quad \chi_3 \equiv b + \bar{d} + \bar{e} \geq 2 \quad \chi_4 \equiv 2\bar{b} + e + f \geq 3
\]

Its dual hypergraph has as hypervertices the constraints of the formula and as hyperedges the variables of this formula.

A hyperedge covers the constraints containing the corresponding variable.
1. Invoke a **PB Solver** on the input
2. If the formula is *UNSAT*, then the compiled form is $\bot$
3. If all variables are assigned, then the compiled form is $\top$
4. For each **connected component** $\varphi$ of the formula:
   a. Choose a variable $v$ based on a cutset of $\varphi$ computed with **KaHyPar** (Schlag, 2020)
   b. Compile $\varphi|_v$ as $\varphi_v$
   c. Compile $\varphi|_{\overline{v}}$ as $\varphi_{\overline{v}}$
   d. The compiled form of $\varphi$ is $\text{ite}(v, \varphi_v, \varphi_{\overline{v}})$
5. The compiled form is the conjunction of the compiled forms obtained above
Outline of *PBD4*

1. Invoke a **PB Solver** on the input
2. If the formula is **UNSAT**, then the compiled form is ⊥
3. If all variables are assigned, then the compiled form is ⊤
4. For each *connected component* \( \varphi \) of the formula:
   a. Choose a variable \( v \) based on a **cutset** of \( \varphi \) computed with *KaHyPar* (Schlag, 2020)
   b. Compile \( \varphi|v \) as \( \varphi_v \)
   c. Compile \( \varphi|\bar{v} \) as \( \varphi_{\bar{v}} \)
   d. The compiled form of \( \varphi \) is \( \text{ite}(v, \varphi_v, \varphi_{\bar{v}}) \)
5. The compiled form is the conjunction of the compiled forms obtained above

*PBD4* is available at [https://github.com/crillab/pbd4](https://github.com/crillab/pbd4)
Conclusion

- Knowledge compilation ensures runtime guarantees for online operations
Conclusion

- Knowledge compilation ensures **runtime guarantees** for online operations

- **Hypergraph partitioning** provides a heuristic to decide in which order to assign variables when building the compiled form
Knowledge compilation ensures runtime guarantees for online operations.

Hypergraph partitioning provides a heuristic to decide in which order to assign variables when building the compiled form.

Modern and efficient SAT solvers are used as oracles to determine whether it is worth compiling subformulae.
Knowledge compilation ensures runtime guarantees for online operations.

Hypergraph partitioning provides a heuristic to decide in which order to assign variables when building the compiled form.

Modern and efficient SAT solvers are used as oracles to determine whether it is worth compiling subformulae.

For compiling certain problems, using PB solvers instead may be more efficient.
Perspectives

- Take advantage of native PB compilation for considering new applications of knowledge compilation (e.g., for explaining (binarized) neural networks)
Perspectives

- Take advantage of native PB compilation for considering **new applications** of knowledge compilation (e.g., for explaining (binarized) neural networks)

- Use **speculation techniques** to speed up compilation:
Perspectives

- Take advantage of native PB compilation for considering new applications of knowledge compilation (e.g., for explaining (binarized) neural networks)

- Use speculation techniques to speed up compilation:
  - by predicting satisfiability before invoking the SAT/PB solver as an oracle
Perspectives

- Take advantage of native PB compilation for considering **new applications** of knowledge compilation (e.g., for explaining (binarized) neural networks)

- Use **speculation techniques** to speed up compilation:
  - by **predicting satisfiability** before invoking the SAT/PB solver as an oracle
  - by **predicting cutsets** before computing a partition of the hypergraph
Hypergraph Partitioning for Compiling Pseudo-Boolean Formulae

Romain Wallon
ROADEF’21, Session Partitonnement des Graphes – April 29, 2021

Laboratoire d’Informatique de l’X (LIX), École Polytechnique, X-Uber Chair


