`pccompile` tool

Petr Kučera

Charles University, Czech Republic

KOCOON Workshop, Arras
December 16–19, 2019

Contents

- Short tool description
- Propagation complete formula
- Checking propagation completeness
- Algorithms
- Invoking `pccompile`
- Conclusion

`pccompile`

- `pccompile` aims to solve the following problems:
  - Checking if a CNF formula is propagation complete (PC).
  - Compile a CNF formula into an equivalent PC formula.
- Two obstacles:
  - Checking propagation completeness is hard and
  - an equivalent PC formula might be exponentially bigger than the input CNF.
- Often works on formulas with 40–50 variables and a few hundreds of clauses.
- Solves some bigger formulas as well.
- The input CNF must be easy for a SAT solver (`glucose` is used internally).
- The tool is EXPERIMENTAL.
- Available at

  `http://ktiml.mff.cuni.cz/~kucerap/pccompile`

Other approaches

- (Brain et al., 2016) (GenPCE) — also tries to add auxiliary variables.
- (Ehlers and Palau Romero, 2018) — also consider approximations of propagation complete formulas
- Both approaches are based on a systematic way of checking partial assignments, usable only for a small number of variables

Propagation Complete Formulas

$\mathrm{lit}(\mathbf{x})$	literals over variables $\mathbf{x}$.
$\varphi \wedge \alpha \vdash_1 l$	Literal l can be derived by unit propagation from $\varphi \wedge \alpha$.
$\perp$	the contradiction (empty clause).

Definition (Bordeaux and Marques-Silva, 2012)

A CNF formula $\varphi(\mathbf{x})$ on variables $\mathbf{x} = (x_1, \ldots, x_n)$ is propagation complete (PC) if for every partial assignment $\alpha \subseteq \mathrm{lit}(\mathbf{x})$ we have

$$\varphi(\mathbf{x}) \wedge \alpha \models l \Leftrightarrow \varphi(\mathbf{x}) \wedge \alpha \vdash_1 \perp \text{ or } \varphi(\mathbf{x}) \wedge \alpha \vdash_1 l$$

- Allows checking consistency and propagating using unit propagation.

Checking Propagation Completeness

- Checking if a CNF is PC is co-NP complete (Babka et al., 2013).
- $\varphi(\mathbf{x})$ is not PC if and only if to asking if there is a partial assignment α and a literal l such that
  1. $\varphi \wedge \alpha \nvdash_1 l$ ($C = \neg\alpha \vee l$ is empowering) and
  2. $\varphi \wedge \alpha \wedge \neg l \vdash_1 \perp$ (C is 1-provable).
- We can check this using a SAT solver.

Encoding 1-provability

- `pccompile` offers two encodings of 1-provability:

  quadratic size $\Theta(\|\varphi\| \cdot n)$ (n times dual rail encoding)

  logarithmic size $\Theta(\|\varphi\| \cdot \log n)$ (smaller, but sometimes harder to solve)
- Allows to pick the smaller of these for each check
- Bounding the depth of the unit resolution proof of $\varphi \wedge \alpha \wedge \neg l \vdash_1 \perp$ during compilation.

Algorithms

- Incremental algorithm
  - Idea: While the formula is not PC, find an empowering implicate and add it to the formula
- Learning approach
  - Dual rail encoding of a PC formula represents a specific Horn function (K. and Savický, 2020)
  - Learn the Horn function using equivalence and closure queries
  - A modification of the algorithm described by Arias, Balcázar, and Tîrnăucă (2015).

    equivalence: try to find an empowering implicate (by SAT, randomly)

    closure: find all literals implied by an assumption
  - Smaller number of PC checks, but bigger overhead
- Use learned clauses as empowering
- Regularly remove the clauses which are not empowering anymore (are absorbed) during the compilation.

Invoking `pccompile`

- Checking if a formula is PC

  ```
  pccompile input.cnf
  ```

- Compiling a formula with the incremental algorithm

  ```
  pccompile -mca incremental input.cnf output.cnf
  ```

- Compiling a formula with the learning algorithm

  ```
  pccompile -mca learning input.cnf output.cnf
  ```

- For other parameters (preprocessing, inprocessing, timeouts, encoding parameters, ...) see the help screen

  ```
  pccompile --help
  ```

Example output (PC check)

Simplified end of the output of an unsuccessful check if a formula is PC

```
c [..FindEmpoweringWithLevel] level=1, input cnf 40 77
c ... Calling SAT with encoding (p cnf 539 2266)
c ... timeout: -1s
c ... Found empowering implicate, time=0.005042s
c Inprocess -2 3 -4 5 8 10 11 12 ..
c Found empowering implicate with empowering variable 10:
5 10 12 0
c Total time: 0.0908475s
c Total processor time: 0.089818s
c Found empowering implicate
c 5 10 12 0
c with empowering variable 10
c No output written
```

Example output (incremental)

A simplified end of the output of an incremental compilation of a randomly generated formula on 40 variables and 80 clauses.

```
c Minimizing hypothesis (p cnf 40 346)
c Finished minimization of hypothesis (p cnf 40 346), time=0.050125s
c Compilation finished successfully, formula is propagation complete
c Total time: 68.8542s
c Total processor time: 68.0544s
c Processor time until the last SAT based EQ check: 24.7915
c Processor time of the last SAT based EQ check: 43.2125
c Total number of empowering clauses: 485
c Total number of added clauses: 485
c Total number of empowering clauses found by SAT: 350
c Total number of learned clauses used: 139
c Total number of learned clauses added as empowering: 135
c Total time of SAT based equivalence queries: 66.4629s
c SAT based equivalence with result SAT (time/count): 21.7473s / 350
c SAT based equivalence with result UNSAT (time/count): 44.7159s / 4
c Maximum UP level: 4
```

Example output (learning)

A simplified end of the output of an learning compilation of a randomly generated formula on 40 variables and 80 clauses.

```
c Finished minimization of hypothesis (p cnf 40 346), time=0.07498s
c Hypothesis minimized (p cnf 40 346), time=0.075456s
c Compilation finished successfully, formula is propagation complete
c Total time: 74.6476s
c Total processor time: 71.4704s
c Processor time until the last SAT based EQ check: 14.9207
c Processor time of the last SAT based EQ check: 56.4741
c Negatives added to the hypothesis: 281
c Clauses added to the hypothesis: 385
c Number of successful refinements: 17
c Total number of candidates with closure: 228
c Total number of candidates without closure: 253
c Total number of learned clauses considered: 95
c Total number of random empowering implicates: 0
c Total number of random bodies: 0
c Total number of learned clauses from random queries: 0
c Total number of random queries: 0
c Total number of empowering implicates found by SAT: 205
... (statistical information continue for a few lines)
```

Experiments on random instances

Randomly generated formulas with modularity-based generator (Giráldez-Cru and Levy, 2015).

- Two sets of 50 instances — 40 variables, 80 clauses and 50 variables, 100 clauses.
- Other parameters $k = 3$, $Q = 0.8$, $c = 3$

	Avg. time	until the last check	emp. found by SAT cnt	emp. found by SAT time
n=40 (I)	35.99	11.89	206.35	10.47
n=40 (L)	40.44	12.27	126.60	8.23
n=50 (I)	3115.02	296.05	638.69	275.51
n=50 (L)	3459.58	343.37	426.10	303.38

(I)=incremental, (L)=learning algorithm
CPU Intel Xeon 2.00 GHz (2007)

Configuration problems

- We were able to solve some instances from the configuration problem set, here are some of them.
- Sizes after propagating backbones

	n	m	Total time	until the last check	emp. cnt	found by SAT time
C169_FV (I)	50	93	0.32	0.21	2	0.08
C169_FV (L)	50	93	0.59	0.48	2	0.08
C171_FR (I)	451	1793	484.47	448.35	271	412.14
C171_FR (L)	451	1793	692.86	629.93	88	162.16
C211_FS (I)	247	906	4191.64	2349.89	1536	2228.13
C211_FS (L)	247	906	2632.64	956.29	305	720.43
C250_FV (I)	129	327	5.46	4.86	46	3.96
C250_FV (L)	129	327	5.49	4.98	8	0.70

(I)=incremental, (L)=learning algorithm
CPU Intel Xeon 2.00 GHz (2007)

Conclusion

- `pccompile` can be also used to check unit refutation completeness (URC) and compile into a URC formula
  - Bigger encoding
  - Only incremental algorithm
- Future directions
  - Different solvers for checking if a formula is PC (other SAT solvers, QBF, SMT)
  - Other approaches to checking if a formula is PC
  - Testing on some interesting formulas
  - Adding auxiliary variables

References I

Arias, Marta, José L. Balcázar, and Cristina Tîrnăucă (2015). "Learning definite Horn formulas from closure queries". In: *Theoretical Computer Science*, pp. -. ISSN: 0304-3975. DOI: http://dx.doi.org/10.1016/j.tcs.2015.12.019.

Babka, Martin et al. (2013). "Complexity issues related to propagation completeness". In: *Artificial Intelligence* 203.0, pp. 19–34. ISSN: 0004-3702. DOI: http://dx.doi.org/10.1016/j.artint.2013.07.006.

Bordeaux, Lucas and Joao Marques-Silva (2012). "Knowledge Compilation with Empowerment". In: *SOFSEM 2012: Theory and Practice of Computer Science*. Ed. by Mária Bieliková et al. Vol. 7147. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, pp. 612–624. ISBN: 978-3-642-27659-0.

References II

- Brain, Martin et al. (2016). "Automatic Generation of Propagation Complete SAT Encodings". In: *Verification, Model Checking, and Abstract Interpretation: 17th International Conference, VMCAI 2016, St. Petersburg, FL, USA, January 17-19, 2016. Proceedings*. Ed. by Barbara Jobstmann and K. Rustan M. Leino. Springer Berlin Heidelberg, pp. 536–556. ISBN: 978-3-662-49122-5. DOI: `10.1007/978-3-662-49122-5_26`.
- Ehlers, Rüdiger and Francisco Palau Romero (2018). "Approximately Propagation Complete and Conflict Propagating Constraint Encodings". In: *Theory and Applications of Satisfiability Testing – SAT 2018*. Ed. by Olaf Beyersdorff and Christoph M. Wintersteiger. Cham: Springer International Publishing, pp. 19–36. ISBN: 978-3-319-94144-8.

References III

Giráldez-Cru, Jesús and Jordi Levy (2015). "A modularity-based random SAT instances generator". In: *Twenty-Fourth International Joint Conference on Artificial Intelligence*.

K. and Petr Savický (2020). "On the size of CNF formulas with high propagation strength". To appear at ISAIM 2020.