About Caching in D4 2.0

Jean-Marie Lagniez[1] **Pierre Marquis**[1,2]

6th July 2021

[1]CRIL, Université d'Artois & CNRS, France
[2]Institut Universitaire de France, France

{lagniez, marquis}@cril.fr

SAT and #SAT

SAT

- Variables: w, x, y, z, . . . , a, b, c, . . .
- Literals: w, y, a, . . . , but also $\neg a$, $\neg c$, $\neg y$, . . .
- Clauses: disjunction of literals (or set of literals)
- CNF formula: conjunction of clauses (or set of clauses)
- Model: mapping from variables to $\{0, 1\}$ that satisfies the CNF formula
- a formula can be **SAT** or **UNSAT**
- The SAT problem is NP-complete

#SAT

- Counting the number of models is the #SAT problem
- It generalizes SAT and is the canonical #P-complete problem

Basic Model Counter

Algorithm 1: `dpll-counter`(Σ)

input : a CNF formula Σ
output: its model count

if solve(Σ) $= \bot$ then return 0;
if cache[Σ] $\neq$ nil then return cache[Σ];
if cache.mustBeCleaned() then return cache.clean();
$\langle \Sigma', \tau \rangle \leftarrow BCP(\Sigma)$;
$comps \leftarrow$ connectedComponents(Σ');
$ret \leftarrow 1$;
if $|comps| > 1$ then
  foreach $c \in comps$ do
    $ret \leftarrow ret \times$ `dpll-counter`(c);
else if $|comps| = 1$ then
  $v \leftarrow$ selectVar($var(\Sigma')$);
  $ret \leftarrow$ `dpll-counter`($\Sigma'|_v$) + `dpll-counter`($\Sigma|_{negv}$);
  cache[Σ'] $= ret$;
return $ret \times 2^{|var(\Sigma) \setminus (var(\Sigma' \cup var(\tau)))|}$;

Basic Model Counter

Algorithm 2: `dpll-counter`(Σ)

input : a CNF formula Σ
output: its model count

if solve$(\Sigma) = \bot$ then return 0;
▷ if cache$[\Sigma] \neq$ nil then return cache$[\Sigma]$;
▷ if cache.mustBeCleaned() then return cache.clean();
$\langle \Sigma', \tau \rangle \leftarrow BCP(\Sigma)$;
$comps \leftarrow$ connectedComponents(Σ');
$ret \leftarrow 1$;
if $|comps| > 1$ then
  foreach $c \in comps$ do
    $ret \leftarrow ret \times$ `dpll-counter`(c);
else if $|comps| = 1$ then
  $v \leftarrow$ selectVar$(var(\Sigma'))$;
  $ret \leftarrow$ `dpll-counter`$(\Sigma'|_v) +$ `dpll-counter`$(\Sigma|_{negv})$;
▷ cache$[\Sigma'] = ret$;
return $ret \times 2^{|var(\Sigma) \setminus (var(\Sigma' \cup var(\tau)))|}$;

Caching Ingredients

- **Caching scheme:**
  - a *caching scheme* c is a mapping associating with any $\varphi \in S(\Sigma)$ for some CNF formula Σ a representation $r_c(\varphi)$ of φ
  - a *cache* for Σ given a caching scheme c is a mapping associating with representations $r_c(\varphi)$ of CNF formulae $\varphi \in S(\Sigma)$ their numbers of models
  - A *correct caching scheme* c is a caching scheme such that for any CNF formula Σ, $\varphi_1, \varphi_2 \in S(\Sigma)$, if $r_c(\varphi_1) = r_c(\varphi_2)$ then $\varphi_1 \equiv \varphi_2$
  - in practice only a syntactic equivalence relation is considered

- **Cache cleaning strategy:**
  - makes precise the entries that have to be removed from the cache
  - the cleaning operations can be achieved periodically, or be triggered by some events (number of entries, quantity of memory, ...)

Clause Representations

- Clauses are simplified and satisfied clauses are not represented
- **Literal representation:**
  - a clause is represented as a string gathering the identifiers (integers) of its literals, terminating by zero
  - the identifier of a positive literal x_i is its index i in the enumeration $x_1 < x_2 < \ldots < x_n$, and the identifier of a negative literal $\neg x_i$ is $-i$
- **Index representation:**
  - the clauses of the input CNF formula are indexed in an arbitrary way (the index of the first clause of Σ is 1, the index of the second clause of Σ is 2, etc.)
  - a clause is represented by its index
  - being able to recover the variables of the CNF formula is required

Formulae Representations

- **All clauses:**
  - in the standard caching scheme all clauses are used to represent the CNF formula at hand
- **Not binary clauses:**
  - in the caching scheme implemented in `sharpSAT` all the clauses except those of size at most two are used to represent the CNF formula
  - to retrieve the CNF formula, variables occurring in the formula must be part of the CNF representation
- **Not touched clauses:**
  - in the caching scheme implemented in `D4` all the clauses except those that have not been touched are used to represent the CNF formula
  - as for `sharpSAT`, to retrieve the CNF formula, variables occurring in the formula must be part of the CNF representation

Example - "All Clauses" Scheme

$\Sigma = (x_1 \vee x_2 \vee x_4) \wedge (x_2 \vee x_3 \vee x_4) \wedge (x_3 \vee x_4 \vee x_5) \wedge (x_1 \vee \neg x_6)$ a CNF, and $\varphi = (x_3 \vee x_4 \vee x_5) \wedge (x_1 \vee \neg x_6)$ the formula obtained after conditioning Σ by x_2 and simplifying it using BCP

- **Literal representation (scheme b):**

$$r_s(\varphi) = [3, 4, 5, 0, 1, -6, 0]$$

- **Index representation (scheme h'):**

$$r_h(\varphi) = ([1, 3, 4, 5, 6], [3, 4])$$

Example - "Not Binary Clauses" Scheme

$\Sigma = (x_1 \vee x_2 \vee x_4) \wedge (x_2 \vee x_3 \vee x_4) \wedge (x_3 \vee x_4 \vee x_5) \wedge (x_1 \vee \neg x_6)$ a CNF, and $\varphi = (x_3 \vee x_4 \vee x_5) \wedge (x_1 \vee \neg x_6)$ the formula obtained after conditioning Σ by x_2 and simplifying it using BCP

- **Literal representation (scheme 2):**

$$r_2(\varphi) = ([1, 3, 4, 5, 6], [3, 4, 5, 0])$$

- **Index representation (scheme p):**

$$r_p(\varphi) = ([1, 3, 4, 5, 6], [3])$$

Example - "Not Touched Clauses" Scheme

$\Sigma = (x_1 \vee x_2 \vee x_4) \wedge (x_2 \vee x_3 \vee x_4) \wedge (x_3 \vee x_4 \vee x_5) \wedge (x_1 \vee \neg x_6)$ a CNF, and $\varphi = (x_3 \vee x_4 \vee x_5) \wedge (x_1 \vee \neg x_6)$ the formula obtained after conditioning Σ by x_2 and simplifying it using BCP

- **Literal representation (scheme i):**

$$r_i(\varphi) = ([1, 3, 4, 5, 6], [])$$

- **Index representation (scheme i'):**

$$r_{i'}(\varphi) = ([1, 3, 4, 5, 6], [])$$

Previous Cache Cleaning Strategies

- `Cachet` **cleaning strategy:**
  - **observation:** the utility of the cached components typically declines dramatically with age
  - each cached component is given a sequence number and those components that are too old are removed from the cache (the age limit is considered as an input parameter)
  - the cache is cleared whenever the number of entries exceeds $2^{21} \times 10$ and 25% of the entries are kept

Previous Cache Cleaning Strategies

- **`sharpSAT` cleaning strategy:**
  - **observation:** some entries are more used than others
  - the entries to be cleaned up do not depend only on their ages, but also on their activity levels and on their sizes
  - a score $score(e)$ is associated with each entry e. At start, $score(e)$ is given by the age of the entry but it is reset during the search each time a positive hit corresponding to the entry is obtained
  - all scores are divided periodically to penalize the oldest entries
  - the cache is cleared only if its size exceeds a fixed fraction of the maximal allowed size
  - entries are removed from the cache until a sufficient amount of space has been recovered

Our New Cache Cleaning Strategy

- **Observation:** positive hits generally appear on small entries
- With each entry e in the cache we associate:
  - $score(e) \in \mathbb{N}$, initially set to $|var(e)|$ and incremented when e hits positively
  - $flag(e)$, a Boolean that is set to true once the entry hits positively
- Periodically, all the entries of the cache are visited in order to decide the ones that must be removed based on the following ratio

$$r(e) = \frac{nbHit[|Var(e)|]}{nbTotal[|Var(e)|]}$$

  with $nbTotal[s]$ the number of entries of size s in the cache, and $nbHit[s]$ the number entries of size s where $flag$ is true
- Every entry e that has a ratio $r(e)$ less that some fixed threshold rm, with a $score(e)$ equals to zero and a $flag(e)$ set to false is flushed. The other entries e are kept and their $score(e)$ are divided by two and whenever $score(e)$ falls to zero, $flag(e)$ is set to false

Experimental Protocol

- All the experiments have been conducted on a cluster equipped with quadcore bi-processors Intel XEON E5-5637 v4 (3.5 GHz) and 128 GiB of memory
- The kernel used was CentOS 7 (version 3.10.0-514.16.1.el7.x86_64) and the compiler used was gcc version 5.3.1
- Hyperthreading was disabled, and no cache share between cores was allowed
- A time-out (TO) of 1h and a memory-out (MO) of 7.6 GiB has been considered for each instance

- We have considered 400 CNF instances, which are precisely the benchmarks used for evaluating the performances of the (possibly weighted) model counters during the First International Competition on Model Counting

Caching Schemes Evaluated

- Seven caching schemes

Name	E/I	All	Not b	Not s
n	–	–	–	–
b	E	✓		
2	E		✓	
i	E			✓
h'	I	✓		
p	I		✓	
i'	I			✓

- `D4` has been used in the model counter mode
- tc was set to 2^{18} and rm to $\frac{1}{2}$
- The hash function used in the implementation of the cache is `MurMurHash2`

Empirical Results

E/I	Cleaning	Insert	All	Not b	Not s
E	none	all	244(155)	251(148)	276(119)
E	none	some	258(132)	264(125)	281(107)
E	Cachet	all	243(156)	261(133)	288(101)
E	Cachet	some	258(132)	274(110)	293(89)
E	sharpSAT	all	262(134)	266(127)	285(87)
E	sharpSAT	some	277(107)	275(108)	291(76)
E	ours	all	280(77)	283(69)	299(23)
E	ours	some	**294**(54)	**296**(47)	**305**(12)
I	none	all	254(145)	261(138)	271(122)
I	none	some	263(124)	269(118)	273(110)
I	Cachet	all	256(143)	271(109)	282(90)
I	Cachet	some	265(122)	279(89)	285(78)
I	sharpSAT	all	273(106)	274(102)	280(85)
I	sharpSAT	some	278(86)	282(85)	282(72)
I	ours	all	279(48)	283(35)	290(23)
I	ours	some	**288**(26)	**292**(16)	**292**(10)

Conclusion and Perspectives

- We have presented an improved caching scheme that can be exploited for model counting or the Decision-DNNF compilation
- We have also presented an improved cache cleaning strategy
- We have reported the results of an empirical evaluation showing the *i* scheme and the new cleaning strategy as useful

- As a perspective for further research, we plan to define and evaluate "mixed" caching schemes, i.e., schemes for which both explicit and implicit representations of clauses can be considered
- Set *tc* and *rm* dynamically by considering the structure of the input formula

About Caching in D4 2.0

Jean-Marie Lagniez[1] **Pierre Marquis**[1,2]

6th July 2021

[1]CRIL, Université d'Artois & CNRS, France
[2]Institut Universitaire de France, France

{lagniez, marquis}@cril.fr