EBERHARD KARLS
UNIVERSITÄT
TÜBINGEN

**Mathematisch-
Naturwissenschaftliche
Fakultät**

Computergrafik

Masters Thesis

Investigations on Backbone Computati-on

Eberhard Karls Universität Tübingen
Mathematisch-Naturwissenschaftliche Fakultät
Wilhelm-Schickard-Institut für Informatik
Symbolisches Rechnen
Jonas Bollgrün, `jonas.bollgruen@posteo.de`, 2019

Bearbeitungszeitraum:	April 1st-October 30th

Betreuer/Gutachter:	Prof. Dr. Wolfgang Küchlin, Universität Tübingen
Zweitgutachter:	Prof. Dr. Michael Kaufmann, Universität Tübingen

Selbstständigkeitserklärung

Hiermit versichere ich, dass ich die vorliegende Bachelorarbeit selbständig und nur mit den angegebenen Hilfsmitteln angefertigt habe und dass alle Stellen, die dem Wortlaut oder dem Sinne nach anderen Werken entnommen sind, durch Angaben von Quellen als Entlehnung kenntlich gemacht worden sind. Diese Bachelorarbeit wurde in gleicher oder ähnlicher Form in keinem anderen Studiengang als Prüfungsleistung vorgelegt.

Jonas Bollgrün (Matrikelnummer 3353424), 7. August 2019

Abstract

Template

Acknowledgments

If you have someone to Acknowledge ;)

Inhaltsverzeichnis

1 Introduction

introduction

wofür brauche ich backbones

wer hat sich damit beschäftigt

welche paper waren relevant

1.1 Disambiguation

1.1.1 Terminology

This thesis is an investigation on the calculation of backbones for boolean formulas in conjunctive normal form (or CNF in short). A CNF formula F is a conjunction of a set of clauses $C(F)$, meaning that all of these clauses have to be satisfied (or fulfilled) to satisfy the formula. A clause c in turn is a disjunction of a set of literals, meaning at least one of said literals must be fulfilled. A literal l can be defined as the occurence of a boolean variable v which may or not be negated and to fulfill such a literal, it's variable must be assigned $\perp$ for negated literals and $\top$ for those literals without negation. The same variable can occur in multiple clauses of the same formula but must have the same assignment in all occurences, either $false$ ($\perp$) or *true* ($\top$). A complete assignment of all variables of F (written as $Var(F)$) that leads to the formula being fulfilled is called a model. A formula for which no model can be found is called unsatisfiable.

The exact terminology can differ depending on the paper and project that you read. A formula can be called a problem and the assignment of a variable can be called it's phase. Clauses can also be called constraints and sometimes sentences. A synonym for a formula, clause or literal being fulfilled is it being satisfied. Models can also be called solutions of formulas

The backbone is a problem specific set of literals that contains all literals that occur in every model of that problem. We can also say that a variable is not part of the backbone, if neither it's positive or it's negative assignment is in the backbone. If we have an unsatisfiable formula, it's backbone can be considered undefinable, which is why this thesis concerns itself only with satisfiable CNF formulas. (TODO ref unsat backbone, aber keine einigkeit drüber)

Implikante/Primimplikante notwendig?

erklärung der sat solver, was er zurückgibt, vlt sogar CDCL

2 Base Algorithms

The algorithms that I investigated for this thesis can be grouped very broadly into two approaches, which I will describe in the following two sections.

2.1 Enumeration algorithms

2.1.1 Model Enumeration

A simple definition of the backbone is that it is the intersection of all models of it's formula. If a literal is not part of the backbone, there must exist a model that contains the negation of that literal. Therefore if we had a way to iterate over every single model of the formula and, starting with the set of both literals for every variable and removing every literal from that set that was missing in one of these models, that set would end up being the backbone of the formula. [MSJL10] as well as [JLMS15] list an algorithm that does exactly this.

Algorithm 1: Enumeration-based backbone computation

Input: A satisfiable formula F
Output: Backbone of F, ν_r

$\nu_r \leftarrow \{x | x \in Var(F)\} \cup \{\neg x | x \in Var(F)\}$
while $\nu_r \neq \emptyset$ **do**
    $(outc, \nu) \leftarrow SAT(F)$
    if $outc = \perp$ **then**
        return ν_r
    $\nu_r \leftarrow \nu_r \cap \nu$
    $\omega_B \leftarrow \bigvee_{l \in \nu} \neg l$
    $F \leftarrow F \cup \omega_B$
return ν_r

Here, found models are prevented from being found again by adding a blocking clause of said model and the algorithm terminates once all models are prohibited and the formula became unsatisfiable through this.

2.1.2 Upper Bound Reduction

Clearly, calculating every single model of a formula leaves room for optimization. Most models of a common boolean formula differ by small, independent differences that can just as well occur in other models. Therefore the intersection of only a handful of models can suffice to result in the backbone, as long as these models are chosen to be as different as possible. This was achieved in [JLMS15] as is described in algorithm 2.

Algorithm 2: Iterative algorithm with complement of backbone estimate

---

Input: A satisfiable formula F

Output: Backbone of F, v_r

$(outc, v_r) \leftarrow SAT(F)$

while $v_r \neq \emptyset$ **do**

    $bc \leftarrow \bigvee_{l \in v_r} \neg l$

    $(outc, v) \leftarrow SAT(F \cup \{bc\})$

    if $outc = \perp$ **then**

        return v_r

    $v_r \leftarrow v_r \cap v$

return v_r

It generates an upper bound v_r of the backbone by intersecting found models and inhibits this upper bound instead of individual models. This blocking clause is much more powerful, because it enforces not only that a new model is found, but also that this new model will reduce the upper bound estimation of the backbone in each iteration.

This is because what remains after the intersection of a handful of models, are the assignments that were the same in all these models and from that we make a blocking clause that prohibits the next model to contain that particular combination of assignments. The next model will then have to be different from all previous models for at least one of the variables in the blocking clause to satisfy it.

Eventually v_r will be reduced to the backbone. This can be easily recognized, because the blocking clause of the backbone or any of it's subsets makes the formula unsatisfiable, except in the case that the formulas backbone would be empty.

Note that it is not particularly important whether the blocking clauses remain in F or get replaced by the next blocking clause, because the new blocking clause bc_{i+1} always subsumes the previous one bc_i, meaning that every solution that is prohibited by bc_{i+1} is also prohibited by bc_i and $F \cup \{bc_i, bc_{i+1}\}$ is equivalent to $F \cup \{bc_{i+1}\}$ concerning the set of models.

This algorithm is implemented in the Sat4J library under the designation *IBB*.

2.1.3 Preferences

This approach still leaves much of it's efficiency to chance. Theoretically the solver might return models with only the slightest differences from each other, when other models could reduce the set of backbone candidates much more. For example the blocking clause can be satisfied with only one literal in it being satisfied, but if we were to find a model that satisfies all literals in the blocking clause, we can immediately tell that the backbone is empty and we would be finished. So it would be a good approach for backbone computation if we could direct our sat solver to generate models that disprove as many of the literals in the blocking clause as possible. Precisely this has been described by [PJ18], but has also been proposed much earlier by [Kai01].

[PJ18] describe an algorithm called $BB-PREF$ or $Prefbones$, which makes use of a slightly modified SAT solver based on CDCL, which is called $prefSAT$ in the algorithm below. It can be configured with a set of preferred literals $prefs$. Typically, when the CDCL algorithm reaches the point where it has the freedom to decide the assignment of a variable, it consults a heuristic that tries to predict the best choice of variable and assignment to reach a model, so to speak, trying to predict assignments in the model that it tries to find. Instead, $prefSAT$ uses two separate instances of these heuristics h_{pref} and h_{tail}, which by themselves may work just as the single heuristic used in the ordinary CDCL solver. The key difference in $prefSAT$ is, that h_{pref}, which contains the literals in $prefs$, is consulted first for decisions, and only when all variables with a preferred assignment are already assigned, h_{tail} is used to pick the most important literal, which only contains literals that are not preferred.

Algorithm 3: BB-PREF: BACKBONE COMPUTATION USING PREF-SAT

Input: A satisfiable formula F in CNF
Output: All literals of the backbone of F, ν_r

$(\_, \nu_r) \leftarrow SAT(F)$
Repeat
    $prefs \leftarrow \{\neg l : l \in \nu_r\}$
    $(\_, \nu) \leftarrow prefSAT(F, prefs)$
    if $\nu \supseteq \nu_r$ **then**
        return ν_r `// No preference was applied`
    $\nu_r \leftarrow \nu_r \cap \nu$

This algorithm also differs from IBB in that it does not add blocking clauses, and that is also why it cannot use the case when F becomes unsatisfiable to terminate the algorithm. Instead it relies on the preferences to be taken into account. Except for the case where a formula has only one model, CDCL must make at least one decision. That decision must come from h_{pref}, except for the case that CDCL learned axiomatic assignments for all variables in $prefs$. Depending on whether the learned value for the variables in $prefs$ contradicts all preferences it may take another call to $prefSAT$, but at the latest then no more changes will happen to v_r and the algorithm terminates. The return condition also covers the case when the backbone turns out to be empty, because then v_r was reduced to $\emptyset$ and that is a subset of every set.

Note that the algorithm was written slightly different from what is listed in [PJ18] to make the relation with common enumeration algorithms more apparent and also make it easier to read.

The return condition makes this algorithm inflexible, as the preferences have to be taken into account without exception. If not, a model might be returned that terminates the algorithm prematurely, because it did not properly test a variable assignment, instead taking a shortcut to save time in the calculation of a model. Since the purpose of this thesis was to experiment with solvers and we were interested in the concrete effects of preferences by themselves on the backbone computation, we created a variation of Prefbones, that uses the previous approach of upper bound reduction, adding a blocking clause to F in every iteration and terminating when F would become unsatisfiable. This made the preferences algorithmically completely optional and allowed to experiment with many variations on the concept.

Algorithm 4: BB-pref: Backbone computation using pref-SAT and blocking clause

Input: A formula F in CNF
Output: All literals of the backbone of F, v_r

```
(_, v_r) ← SAT(F)
Repeat
    bc ← ⋁_{l∈v_r} ¬l
    F ← F ∪ {bc}
    prefs ← {¬l : l ∈ v_r}
    (outc, v) ← prefSAT(F, prefs)
    if outc = ⊥ then
        return v_r
    v_r ← v_r ∩ v
```

2.2 Iterative algorithms

2.2.1 Testing every literal

Alternatively, you can define the backbone as all literals that occur with the same assignment in all models of it's problem, which implies that enforcing that variable to it's negation should make the formula unsatisfiable. This definition already leads to a simple algorithm that can calculate the backbone, by checking both assignments of every literal for whether it would make the formula unsatisfiable, see Algorithm 1. This algorithm is referenced in [MSJL10]

Algorithm 5: Iterative algorithm (two tests per variable)

Input: A satisfiable formula F in CNF
Output: All literals of the backbone of F v_r

$v_r \leftarrow \emptyset$
for $x \in Var(F)$ **do**
  $(outc_1, \_) \leftarrow SAT(F \cup \{x\})$
  $(outc_2, \_) \leftarrow SAT(F \cup \{\neg x\})$
  $assert(outc_1 = \top \vee outc_2 = \top)$ `// Otherwise F would be unsatisfiable`
  else if $outc_1 = \bot$ **then**
    $v_r = v_r \cup \{\neg x\}$
    $F = F \cup \{\neg x\}$
  else if $outc_2 = \bot$ **then**
    $v_r = v_r \cup \{x\}$
    $F = F \cup \{x\}$
return v_r

As is commonly written in literature about boolean satisfiability, the two calls to the *SAT* function return a pair which consists first of whether the given function was satisfiable at all and, secondly, the found model, which in this case is discarded. There is no good algorithm that can tell whether a boolean formula is satisfiable or not without trying to find a model for said formula, but we can use it to greatly improve the algorithm above by combining this approach with that of the enumeration algorithms.

2.2.2 Combining with Enumeration

First observe that any model of F would already reduce the set of literals to test by half, because for every assignment missing in the model, we know that it cannot be part of the backbone, so there is no need to test it.

This can be repeated with every further model that we find. The following algorithm is another one that is listed in both [MSJL10] and [JLMS15] and is implemented in the Sat4J library as *BB*

Note that both possible results of the call to the sat solver are converted to useful information. In the else branch, the formula together with the blocked literal l was still solvable. In this case v is still a valid model for F, so we can search through it to look for more variables that don't need to be checked. Note that here v must contain $\neg l$, as it was enforced.

Algorithm 6: Iterative algorithm (one test per variable)

Input: A satisfiable formula F in CNF
Output: All literals of the backbone of F v_r

$(outc, v) \leftarrow SAT(F)$
$\Lambda \leftarrow v$
$v_r \leftarrow \emptyset$
while $\Lambda \neq \emptyset$ **do**
  $l \leftarrow pick\ any\ literal\ from\ \Lambda$
  $(outc, v) \leftarrow SAT(F \cup \{\neg l\})$
  if $outc = \bot$ **then**
    $v_r \leftarrow v_r \cup \{l\}$
    $\Lambda \leftarrow \Lambda \setminus \{l\}$
    $F \leftarrow F \cup \{l\}$
  else
    $\Lambda \leftarrow \Lambda \cap v$
return v_r

In the other case, we identified l as a backbone literal. In that case it will be added to the returned set, removed from the set of literals to test and, lastly, added to the problem F, which increases performance in subsequent solving steps. However it would be even better, not only to reuse the learned backbone literals, but all learned clauses.

2.3 Optimizations

2.3.1 Reusing learned literals

vor allem für prefsat

2.3.2 Model Reduction

primimplikante rotationen

2.3.3 Implied backbone literals

Unit (küchlin bones) satz aus chinesischem paper

Literaturverzeichnis

[DFLBM13] David Déharbe, Pascal Fontaine, Daniel Le Berre, and Bertrand Mazure. Computing prime implicant. *2013 Formal Methods in Computer-Aided Design, FMCAD 2013*, pages 46–52, 10 2013.

[JLMS15] Mikoláš Janota, Inês Lynce, and Joao Marques-Silva. Algorithms for computing backbones of propositional formulae. *AI Commun.*, 28(2):161–177, April 2015.

[Kai01] Andreas Kaiser. Detecting inadmissible and necessary variables in large propositional formulae. 08 2001.

[MSJL10] João Marques-Silva, Mikoláš Janota, and Inês Lynce. On computing backbones of propositional theories. *Frontiers in Artificial Intelligence and Applications*, 215:15–20, 01 2010.

[PJ18] Alessandro Previti and Matti Järvisalo. A preference-based approach to backbone computation with application to argumentation. pages 896–902, 04 2018.