EBERHARD KARLS
UNIVERSITÄT
TÜBINGEN

**Mathematisch-
Naturwissenschaftliche
Fakultät**

Computergrafik

Masters Thesis

Investigations on Backbone Computation

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. [?] as well as [?] list an algorithm that does exactly this.

Algorithm 1: Enumeration-based backbone computation

```
Input: Satisfiable formula $F$
Output: Backbone of $F$, $\nu_r$
1  $\nu_r \leftarrow \{x | x \in Var(F)\} \cup \{\neg x | x \in Var(F)\}$
2  while $\nu_r \neq \emptyset$ do
3      $(outc, \nu) \leftarrow SAT(F)$
4      if $outc = \bot$ then
5          $return\ \nu_r$
6      $\nu_r \leftarrow \nu_r \cap \nu$
7      $\omega_B \leftarrow \bigvee_{l \in \nu} \neg l$
8      $F \leftarrow F \cup \omega_B$
9  $assert(\nu_r = \emptyset)$
10 return $\nu_r$
```

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 [**?**] as is described in algorithm 2.

Algorithm 2: Iterative algorithm with complement of backbone estimate

```
Input: Satisfiable formula $F$
Output: Backbone of $F$, $v_r$
1 $(outc, v_r) \leftarrow SAT(F)$
2 while $v_r \neq \emptyset$ do
3     $bc \leftarrow \bigvee_{l \in v_r} \neg l$
4     $(outc, v) \leftarrow SAT(F \cup \{bc\})$
5     if $outc = \bot$ then
6         return $v_r$
7     $v_r \leftarrow v_r \cap v$
8 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, eliminating

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

Algorithm 3: BB-pref: Backbone computation using pref-SAT

```
Input: A formula $F$ in CNF
Output: All literals of the backbone of $F$, $v_r$
1  $v_r \leftarrow SAT(F)$
2  $setPreferences(\{\neg l : l \in v_r\})$
3  Repeat
4      $(\_, v) \leftarrow prefSAT(F)$
5      if $v \supseteq v_r$ then
6          return $v_r$
7      $removePreferences(\{\neg l : l \in v_r \backslash v\})$
8      $v_r \leftarrow v_r \cap v$
```

im unsat fall gibt es kein model, kann also nicht geschnitten werden drei zeilen im unsat fall erklären 1. zur rückgabemenge hinzufügen 2. literal nicht mehr prüfen 3. backbone literal in formel aufnehmen (lernen)

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 $F$, $v_r$
1  $v_r \leftarrow SAT(F)$
2  $setPreferences(\{\neg l : l \in v_r\})$
3  Repeat
4      $bc \leftarrow \bigvee_{l \in v_r} \neg l$
5      $F \leftarrow F \cup \{bc\}$
6      $(outc, v) \leftarrow prefSAT(F)$
7      if $outc = \bot$ then
8          return $v_r$
9      $removePreferences(\{\neg l : l \in v_r \backslash v\})$
10     $v_r \leftarrow v_r \cap v$
```

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 [?]

Algorithm 5: Iterative algorithm (two tests per variable)

```
Input: A formula $F$ in CNF
Output: All literals of the backbone of $F$ $v_r$
1  $v_r \leftarrow \emptyset$
2  for $x \in Var(F)$ do
3      $(outc_1, v) \leftarrow SAT(F \cup \{x\})$
4      $(outc_2, v) \leftarrow SAT(F \cup \{\neg x\})$
5      if $outc_1 = \bot \wedge outc_2 = \bot$ then
6          $return\ \emptyset$
7      else if $outc_1 = \bot$ then
8          $v_r = v_r \cup \{\neg x\}$
9          $F = F \cup \{\neg x\}$
10     else if $outc_2 = \bot$ then
11         $v_r = v_r \cup \{x\}$
12         $F = F \cup \{x\}$
13 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 [?] and [?] 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

Algorithm 6: Iterative algorithm (one test per variable)

```
Input: A formula $F$ in CNF
Output: All literals of the backbone of $F$ $v_r$
1  $(outc, v) \leftarrow SAT(F)$
2  $\Lambda \leftarrow v$
3  $v_r \leftarrow \emptyset$
4  while $\Lambda \neq \emptyset$ do
5      $l \leftarrow pick\ any\ literal\ from\ \Lambda$
6      $(outc, v) \leftarrow SAT(F \cup \{\neg l\})$
7      if $outc = \bot$ then
8          $v_r \leftarrow v_r \cup \{l\}$
9          $\Lambda \leftarrow \Lambda \setminus \{l\}$
10         $F \leftarrow F \cup \{l\}$
11     else
12         $\Lambda \leftarrow \Lambda \cap v$
13 return $v_r$
```

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.

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.