1 Introduction

introduction

wofür brauche ich backbones

wer hat sich damit beschäftigt

welche paper waren relevant

wie ist die thesis strukturiert,

was wurde in den sechs monaten gemacht

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 $\bot$ 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 ($\bot$) or ($\top$). 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.

When we want to know whether a model M satisfies a formula F, clause c or literal l, we write $F\langle M\rangle \rightarrow \{\top, \bot\}$ or $c\langle M\rangle$ and $l\langle M\rangle$ respectively. The result is $\top$ if the assignment satisfies what it is applied to or $\bot$ if it doesn't.

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. Sometimes assignment and literal are used interchangeably, as they both consist of a variable and a boolean value. 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 terminology for $\top$ or $\bot$ can be (T,F),$(true, false)$ or $(1,0)$.

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?

subsumption erklären (to subsume)

1.1.2 Conflict Driven Clause Learning

08 15 sat solver, interna sind wichtig zu wissen um effekte die spaeter beschrieben werden zu verstehen erklärung der sat solver, was er zurückgibt, vlt sogar CDCL

As is usually done in literature, we write calls to sat solvers such as CDCL in code listings as $(outc, v) = SAT(F)$. Here, two values are returned. $outc$ is a boolean value that simply states whether F was satisfiable to begin with. Only if it equals to $\top$, the second return parameter v can have a meaningful value, which would be the model that was found and satisfies F. In some of the algorithms listed in this thesis, one of the return parameters is not used at all. In that case we write $(\_, v) = SAT(F)$ or $(outc, \_) = SAT(F)$ to indicate that either $outc$ or v is discarded.

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: A satisfiable formula F
Output: Backbone of F, v_r

```
1 v_r ← {x|x ∈ Var(F)} ∪ {¬x|x ∈ Var(F)}
2 while v_r ≠ ∅ do
3     (outc, v) ← SAT(F)
4     if outc = ⊥ then
5         return v_r
6     v_r ← v_r ∩ v
7     ω_B ← ⋁_{l∈v} ¬l
8     F ← F ∪ ω_B
9 return v_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: A satisfiable formula F
Output: Backbone of F, v_r

```
1 (outc, v_r) ← SAT(F)
2 while v_r ≠ ∅ do
3     bc ← ⋁_{l∈v_r} ¬l
4     (outc, v) ← SAT(F ∪ {bc})
5     if outc = ⊥ then
6         return v_r
7     v_r ← v_r ∩ 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. 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 for the algorithm 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 [**?**], but has also been proposed much earlier by [**?**].

[**?**] 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, v_r

```
1 (_, v_r) ← SAT(F)
2 Repeat
3     prefs ← {¬l : l ∈ v_r}
4     (_, v) ← prefSAT(F, prefs)
5     if v ⊇ v_r then
6         return v_r                    // No preference was applied
7     v_r ← v_r ∩ v
```

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 [**?**] 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

```
1 (_, v_r) ← SAT(F)
2 Repeat
3     bc ← ⋁_{l∈v_r} ¬l
4     F ← F ∪ {bc}
5     prefs ← {¬l : l ∈ v_r}
6     (outc, v) ← prefSAT(F, prefs)
7     if outc = ⊥ then
8         return v_r
9     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 [**?**]

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

```
1  v_r ← ∅
2  for x ∈ Var(F) do
3      (outc_1, _) ← SAT(F ∪ {x})
4      (outc_2, _) ← SAT(F ∪ {¬x})
5      assert(outc_1 = ⊤ ∨ outc_2 = ⊤)     // Otherwise F would be unsatisfiable
6      else if outc_1 = ⊥ then
7          v_r = v_r ∪ {¬x}
8          F = F ∪ {¬x}
9      else if outc_2 = ⊥ then
10         v_r = v_r ∪ {x}
11         F = F ∪ {x}
12 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 still solvable. In this case ν 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 ν 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

```
1  (outc, v) ← SAT(F)
2  Λ ← v
3  v_r ← ∅
4  while Λ ≠ ∅ do
5      l ← pick any literal from Λ
6      (outc, v) ← SAT(F ∪ {¬l})
7      if outc = ⊥ then
8          v_r ← v_r ∪ {l}
9          Λ ← Λ \ {l}
10         F ← F ∪ {l}
11     else
12         Λ ← Λ ∩ v
13 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.

Hidden disadvantage

You would think that the iterative approach is just an improvement over the model enumeration algorithm. However it has a hidden disadvantage over them, which stems from how the given problem F is modified as part of the algorithm. Remember how we wrote in chapter 2.1.2 how it does not particularly matter whether the blocking clause is removed or stays. Actually it can make a slight difference to the performance of the individual sat solving calls. The CDCL algorithm develops learned clauses during it's runtime. These learned clauses do not change the set of models for the formula but make it easier to avoid partial assignments of variables that by themselves make the formula unsatisfiable

TODO beispiel, mindestens 4 variablen

Clause learning in CDCL happens through a process called resolvent building. Here, two contradicting assignments are evaluated for their reasons, eventually ending up with a set of variable assignment decisions that led to the contradiction which is subsequently prohibited by a learned blocking clause.

TODO resolventen algorithmus
TODO resolventen bildchen

These learned clauses can be reused in subsequent sat calls and can improve the speed of the solver dramatically. They prevent contradictory variable assignments, specifically those that the sat solver ran into before and have a good chance to be made again if nothing prevents the solver from it.

However reusing learned clauses is not as easy for the *BB* solver as for the *IBB* solver. Learned clauses are based on a subset of the clauses in F. Their existence is virtual so to speak, implied through these base clauses, just not easy to recognize. Adding more clauses to the formula does not remove a learned clause, as the set of base clauses is untouched. At most it might be possible to subsume it with another learned clause. However if a clause is removed from F, it might be that the learned clause is no longer implied through the formula, therefore making it invalid.

Example: Using the iterative approach on formula $F = \{(a, b, c)\}$ together with blocking clause $(\neg a)$ implies the clause (b, c). CDCL would learn this clause if it were configured to assign $\bot$ in every decision and assign $\top$ only through unit implications. This learned clause must be discarded. Otherwise when we test with the blocking clause $(\neg b)$, together with (b, c) it would imply (c), making c a backbone of F, which it clearly isn't.

In the *BB* solver, except for the very first solving step, no formula that is solved is a subset of any other, because every time a single different literal is enforced. Therefore the clauses that are learned in these solving steps must be discarded and cannot help the other solving steps [1]. The *IBB* solver on the other hand only adds further constraints, therefore it can reuse all it's learned clauses and speed up the model generation over time.

NOPE, alles falsch. BB behält in praxis gelernte klauseln. Einziger nachteil ist, dass unsat ergebnisse kein model zurückgeben und dadurch nicht die kandidaten schneiden

[1] The most you could do would be to keep track which clauses went into the generation of each learned clause and then only discard those learned clauses that were completely unrelated from the clause that was deleted from F. And that might have to be done recursively, if one of the base clauses is itself a learned clause. The SAT4J framework simply discards all learned clauses of the formula when a clause is removed from it.

3 Optimizations

The following chapter elaborates on methods to enhance the algorithms that were described in the previous chapter. Depending on the particular combination of algorithm and enhancement, applying the optimization can be be considered a no brainer. However, experimental results show that this is not true without exception and in individual cases we will give thoughts why that is. Other improvements can only be applied to some algorthms due to the data that is available.

3.1 Model Reduction

The algorithms described in section 2 all boil down to testing for each variable whether there exist two models where one assigns the variable to $\top$ and the other to $\bot$. This section describes two strategies to reduce the model that is returned by the sat solver to a subset that tells us more for the purpose of calculating a backbone. Both methods are related to the concept of the *implicant*.

An implicant is a set of assignments of the variables in the problem F that still satisfies F. The difference to models is, that here it is allowed to leave variables undefined. Let's imagine a formula where every clause contains the literal a, amongst other literals. Then (a) would be a simple implicant for this formula, because assigning a to $\top$ is sufficient to satisfy each clause. However there may also be a different implicant that does not contain a at all, satisfying the clauses in a different way. If a variable v does not occur in an implicant I, we say that v is optional in I.

Having a single implicant I that leaves some variables optional immediately tells us, that every possible combination of assignments of the optional literals can make a model, if we just add the assigned literals in I. You could say that an implicant implies a large set of models.

Without any further information, a single implicant I tells us, that every one of the variables that are missing in it cannot be part of the backbone.

Starting with complete models, implicants can be subsets of other implicants, by removing more and more assignments that are not essential. This way you will eventually reach an implicant where all of it's assignments are required and removing any literal from it, would leave some clause unsatisfied. This would be called a prime implicant I_π.

3.1.1 Prime Implicant

[?] describes an algorithm that allows to calculate the prime implicant from a model in linear time over the number of literal occurrences in the formula. This algorithm works best if you generate the model of a variable with the CDCL algorithm for multiple reasons.

First, it takes advantage of data structures that you also need in a good implementation of CDCL, namely a lookup from each variable to all clauses that contain either literal of that variable. In CDCL this lookup table improves the performance of unit propagation, because you can check exactly the set of clauses that might be affected by the assignment to determine whether one of the clauses has become exhaustively unsatisfied or implies another assignment. Here, the lookup is used to determine whether a literal is required in the implicant that you are in the process of generating, by looking for clauses that only contain a single satisfying literal anymore, which then must be part of the prime implicant. You can even reuse the watched literals of the clauses, however you would have to change the way in which the propagate, since you take assignments away instead of adding them.

The second fit with CDCL is that CDCL not only generates a model, but also a table containing information how that model was generated. This information can be used in this algorithm because if you know of some assignments that they must be in the implicant (passed as I_r), checking them can be skipped here. You can quickly generate this set by going through the table generated by CDCL and noting down every assignment that happened through unit propagation. These must be part of the prime implicant because to have been assigned through unit propagation at some point in time there must have been a clause that required that particular assignment.

TODO beweisen, dass linear durchzulaufen zu einer primimplikante führt. (dass nichts zweimal geprüft werden muss)

The only assignments that you really have to test here are those that were decisions. Additionally, you can avoid many decided assignments if you configure CDCL to stop once the formula is satisfied instead of stopping once every variable was assigned, because once the formula is satisfied, no more assignment is necessary, so only all further ones must be arbitrary decisions that are not necessary in the implicant.

The function $Req(I_m, l, c)$ tells, whether the assignment l in the implicant I_m is required to satisfy c. In other words, is l the only literal in I_m that also occurs in c.

Algorithm 7: BASE APPROACH TO COMPUTE A PRIME IMPLICANT

Input: A formula F, a model I_m, I_r containing some required literals in I_m
Output: I_r, reduced to a primeimplicant of F, being a subset of I_m

```
while ∃l ∈ I_m\I_r do
    if ∃c ∈ F : Req(I_m, l, c) then
        I_r ← I_r ∪ {l}
    else
        I_m ← I_m\{l}
return I_r
```

3.1.2 Rotations

There is a model reduction method that is more powerful than calculating the prime implicant. Even better, the concept is much simpler than calculating the prime implicant.

Any model of a CNF formula can contain multiple implicants and even prime implicants[1] , so if we could find out from only a single model M_0 of F, which of these literals occured in all implicants that that model covered, with each model we could reduce the set of backbone candidates even more than with just one of it's prime implicants.

Doing this is actually pretty simple. Instead of generating a small set various prime implicants, you check for each assignment a in M_0, whether it is required in M_0, by checking whether $(M_0 \backslash a)$ is still an implicant that satisfies F. You can do this faster if you make use of the lookup table described in the previous subsection so that you only need to check affected clauses, since as you start from a known model, every unaffected clause should still be satisfied. However, tests with our implementation showed that the time that it takes to test an implicant by brute force, iterating over each clause and looking for a satisfied literal, was negligible, so there is hardly a benefit in something more complicated.

This gives you the information that we would otherwise generate from all implicants that were contained in M.

[1] As an example imagine any formula with only clauses of size two. This formula has at least two prime implicants I_1 and I_2) that you can generate by collecting either the first literal of every clause for I_1 or the second literal for I_2. If the literals in I_1 are disjunct from those in I_2 then you can build a model of the combination of I_1 and I_2.

Algorithm 8: BASE APPROACH TO COMPUTE A PRIME IMPLICANT

Input: A formula F, a model M
Output: R, the required literals of all implicants in M

```
1 R ← ∅
2 for a ∈ M do
3     I_a ← M\a
4     if ∃c ∈ F : c⟨I_a⟩ = ⊥ then
5         R ← R ∪ {a}
6 return I_a
```

3.2 Cheap Identification of backbone literals

This section describes various ways that allow you to recognize backbone literals without an additional satisfiability check. Knowing these backbone literals early can speed up the individual calls to the SAT solver, because enforcing the backbone literals prevents the solver from trying to find solutions containing the inverse of the backbone literals, which by definition don't exist.

3.2.1 Axiomatic literals

The most straightforward method to quickly identify backbone literals is to scan the formula for clauses with only a single literal. Since these clauses have only one possible way to become satisfied, that assignment must be used in every model of the formula and is therefore backbone. It makes sense to do this check after every sat computation, as every different way that leads to a different model brings different learned clauses, that may sometimes consist of a single literal.

However you have to make sure that you don't add learned literals to your returned backbone set if they were already disproven to be part of the backbone which you can tell from your current blocking clause.

TODO beweisen oder rausnehmen.

An expansion on this would be to look for pairs of clauses (a, b), $(a, \neg b)$ for any two variables of the formula. The only way to fulfill this pair of clauses is to assign a to $\top$. This scheme can theoretically be applied to any clause size, but then you would require a quadratically increasing number of clauses to determine a backbone which would first increase computation time and secondly decrease the chance that the necessary set of clauses was available.

Außerdem: alle assignments in CDCL level 0

3.2.2 Unit Implication

yadayada

bedingung: damit c hier angewandt werden kann müssen alle literale darin backbone sein und $n_c - 1$ davon müssen bekannt sein.

We have tested this method in two solvers, *BB* and a variant of *PrefBones* that learns backbone literals only from clause learning as described in the previous subsection. This clearly showed that this method requires a critical mass of already known backbone literals to have a benefit, because as already stated, if there even exists a clause in F where this method can be applied, you need to know the $n_c - 1$ other backbone literals in c beforehand.

The learned backbone literals alone can rarely supply this, before the *PrefBones* algorithm terminates from other conditions. However the iterative approach of the *BB* solver, testing every yet unidentified literal individually, is much faster at providing positively identified backbone literals that you need to imply other backbone literals through unit implication.

TODO tabelle mit numUnitLits verglichen zwischen pb1e und kbb. Soll auch zwischen vonThore und satComb unterscheiden.

3.2.3 Implication through Cohabitation

TODO besserer name

Gegeben den Fall, Literal +a wurde als teil des Backbones identifiziert.
Wenn ein literal +b existiert das in allen Klauseln auftritt die auch +a enthalten, dann ist -b im Backbone
Beweis: Angenommen -b wäre nicht im backbone, dann gäbe es eine lösung die +b enthält
Alle Klauseln die +b enthalten enthalten auch +a, also wäre +a optional

Literaturverzeichnis

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

[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.