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 Implikante/Primimplikante besser hier? derzeit in Model Reduction untergebracht.

For the context of CNF formulas, on which this thesis focuses, the term "subsumption" should to be explained. A clause c_1 subsumes another clause c_2 of the same formula, if and only if $c_1 \subseteq c_2$, in prose if all the literals that occur in c_1 also occur in c_2[1]. If c_1 subsumes c_2 in formula F then this means that you can remove c_2 from F because in terms of satisfying models, $F \backslash \{c_2\}$ is equivalent to F as $\{c_1\}$ is equivalent to $\{c_1, c_2\}$. This is because in this case an assignment that satisfies c_1 also satisfies c_2 automatically. There is no possible assignment that satisfies clause c_1 that doesn't satisfy it's subsuming clause c_2.

1.1.2 CDCL

All the methods to generate a backbone of a formula F that are described in this thesis essentially rely on calculating various models of F, so it makes sense to describe a method to do that as well. The current state-of-the-art algorithm to do this is the *Conflict Driven Clause Learning* algorithm, or *CDCL* for short.

Here a *CDCL table* is used as a special dataset, to store the state of the SAT solver. This table stores the succession of assignments with four values for each assignment.

- The *level* of the assignment. This level increases with each decision and starts at 0, where unit assignments before any decision are stored.
- The affected variable.
- The value that the variable was assigned to.
- The reason for the assignment. This can be one of two cases, either *Unit* or *Decision*.
  Unit assignments happen, when a clause has all but one of it's literals unsatisfied. Since all clauses have to be satisfied for a CNF formula to be satisfied, that last literal must be assigned a value that satisfies it and it's clause. Entries in the CDCL table that refer to a unit assignment also store a reference to the clause that required the assignment. A clause that fulfills the above condition and requires an assignment can be called a *unit clause* or that it *is unit*.
  Decisions happen when no clause is in the unit state. In theory, in this case you are free to pick any variable and assign it either $\top$ or $\bot$.

[1] One can filibuster whether c_1 would have to be a true subset of c_2. If a formula has two occurences of the exact same clause, then one of the occurences would be redundant, so the same rule could be applied here as well. In practice it makes more sense to filter out duplicates of clauses before running any computation on the formula. Similarly, you can safely drop all clauses where both literals of the same variable occur, as that clause would be satisfied in each and every possible model.

The purpose of unit assignments is to reduce the search space. The solving process for a formula can be modeled as the traversal of a search tree, where each node corresponds to an individual assignment and every leaf node to a complete assignment that might be satisfying or not[2]. However, given that you can stop to search once a single clause is unsatisfied, you can disqualify many branches of the tree early, for example when the assignments in your tree path so far require some additional assignment for some clause, which would be a *unit clause*. Going the other way in the tree at that particular node will never result in a satisfying model.

The solver will now fill the table with assignments, unit assignments if possible or decisions otherwise, until one of two things happens. Either the formula became satisfied[3] in which case we can return the assignments that are stored in the table.

The other possibility is that you run into a contradiction. Here a unit clause requires that a variable is assigned a value b, but it is already assigned $\neg b$. If the previous assignment has a reason (a unit clause), then some of the variables in that clause must be assigned differently(since it was unit then), so we can connect the reason for our conflict to other assignments. Repeating this we sometimes meet decisions instead of unit clauses as reason. Collecting these decisions, we end with the precise combination of assignment decisions that resulted in the conflict. This process is called *resolution* and is visualized more precisely in figure TODO referenzieren. This set of decisions must now be taken back, by reversing the assignmnents up until the first of these decisions, as this path through the search tree (TODO reicht es nicht die erste decision wegzunehmen?) results in a conflict and will not end in models.
We also add the clause to our formula as a *learned clause*. This clause serves to prevent the particular combination of assignments that led to our conflict. The resulting formula will still be completely equivalent to before. It merely stores the information the we gained through analysis of our conflict to prevent it from happening again.

TODO Urpaper von CDCL und selektionsheuristic

Concerning the decisions, depending on the particular formula, it is possible that some assignments make it easier to solve the formula than others and some decisions might lead to a completely unsatisfied clause where all of it's literals are unsatisfied. A lot of work has been done to prevent this by setting up heuristics that try to pick a variable and corresponding assignment that would lead to a satisfying model without complications. For this you would typically measure how often the variable was involved in conflicts and prefer to decide on such difficult variables. It is assumed that difficult formulas are so because of individual difficult variables. If you decide on these early, you have a 50-50 chance that you either found the correct assignment or at least learned a useful clause from a conflict.

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

[2]Sadly the size of such a tree makes it impractical to actually implement sat solving like this.

[3]In this case only an implicant is returned. If you want a complete model, you can keep making decisions until all variables are assigned and return the assignments after that. However once you have an implicant, you are free to assign the remaining variables to anything you want, as the formula is already satisfied and further assignments cannot change that.

Algorithm 1: CDCL

Input: A formula F in CNF
Output: A CDCL table which implies an implicant for F, or $\bot$ if F is not satisfiable

```
1  level ← 0
2  table ← emptyList
3  while 1 do
4      table.pushAll(F.getUnits())
5      if ∃ conflicting assignment then
6          if level = 0 then
7              return ⊥
8          else
9              level ← backtrackAndLearn(F, table)
10     else if F is fulfilled then
11         return table
12     else
13         level++
14         l ← any free variable
15         l.assign(either ⊤ or ⊥)
16         table.pushDecision(l)
```

that simply states whether F was satisfiable to begin with. Only if it equals to $\top$, the second return parameter ν 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 $(\_, \nu) = SAT(F)$ or $(outc, \_) = SAT(F)$ to indicate that either $outc$ or ν 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. [MSJL10] as well as [JLMS15] list an algorithm that does exactly this.

Algorithm 2: Enumeration-based backbone computation

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

$v_r \leftarrow \{x | x \in Var(F)\} \cup \{\neg x | x \in Var(F)\}$
while $v_r \neq \emptyset$ **do**
  $(outc, v) \leftarrow SAT(F)$
  if $outc = \bot$ **then**
    return v_r
  $v_r \leftarrow v_r \cap v$
  $\omega_B \leftarrow \bigvee_{l \in v} \neg l$
  $F \leftarrow F \cup \omega_B$
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 [JLMS15] as is described in algorithm 2.

Algorithm 3: 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*.

2.2 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 4: 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 [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 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 5: 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.

It is possible to apply the concept of preferences described in section 2.2.3 to the iterative algorithms listed here. However, this is less beneficial than adding it to the enumeration approach of *IBB*, because in the *BB* algorithm many sat calls are supposed to return *UNSAT* to positively identify an assignment as backbone. However, this case does not give us a model, so the extra effort trying to find a more valuable model is often wasted here.

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 duplikat, klausel lernen wird schon in disambiguation erklärt. 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.

TODO BB behält gelernte klauseln ausser denen die was mit assumptions zu tun haben. IBB muss aber die ganze analyse nicht machen

2.2.3 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 6: 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 [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 7: 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
```

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

[DFLBM13] 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.

Algorithm 8: 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

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

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.

3.1.2 Rotations

There is a model reduction method that is even 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 of 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. If so, we know that a is not part of the backbone, because we found an implicant without it.

You can do this faster if you make use of the lookup table described in the previous subsection, where each literal is mapped to the set of clauses where it occurs. After all, you already know that M_0 satisfies F, so it is sufficient to check only the clauses that contain the tested literal a, whether they are satisfied in a different way in $M_0 \backslash \{a\}$. Unaffected clauses must still be satisfied without a.
Whether this gives a performance benefit greatly depends on the structure of the formulas. I essentially used two benchmarks, one academic made of examples from the sat competitions and the other being a real-life example from the automobile industry. The academic one contained much fewer clauses (1.000-2.000) compared to the industrial (around 60.000), however the clauses of the academic benchmark contained three literals each, compared to two in the industrial one.
TODO individuelle vonThore benchmarks mit individuellen satcomp benchmarks vergleichen

The benefit of this approach depends on the particular example as well. If the individual models of a formula contain very few (prime-)implicants, then looking for rotatable literals might not give a benefit over calculating the prime implicants.

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

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

[1] As an example imagine any formula with only clauses of size two, where the first literals are disjunct from the second literals. 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. Since the literals in I_1 are disjunct from those in I_2, you can build a model of the combination of I_1 and I_2.

Algorithm 9: Literal rotation in models

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 R
```

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, if you want to use this feature in each iteration, make sure that your learned clauses do not stem from the means in which you enforce different models, for example the blocking clauses in *IBB* or the checked literal in *BB*. Otherwise, you might confuse the backbone literals of your original problem F with literals that only became axiomatic in your modified formula. [2]

The *BB* solver manages this by having the feature of *assumptions*, which were first proposed in [ES03]. These assumptions are assignments that are tested for whether the formula has a model that contains these assignments. For this, the CDCL table is first filled with decisions, that match the assumptions and subsequently the CDCL algorithm runs it's course ordinarily, except that the root decision level is after the entries that assign the assumptions. This has the effect that the assignments from the assumptions can not be taken back through backtracking, instead returning that F is unsatisfiable under the given assumptions.

Once a model was found, all the learned clauses are checked whether they are related to the assumptions, and if so, they must be discarded, as is mentioned in [WKS01]. This way, the assumptions will not have any long term effect [3] . The *IBB*

[2] A simple example: You find an implicant $\{a\}$ with only a single literal. Then you add the blocking clause $\{\neg a\}$. But in $F \cup \{\neg a\}$ $\neg a$ is suddenly a backbone literal.

[3] [ES03] actually states that "all learned clauses are safe to keep". [WKS01] differs from [ES03] in that here clauses are removed explicitely whereas assumptions are removed automatically.

on the other hand is implemented to discard all learned clauses after every iteration, which gives it a tremendous performance loss.

TODO: IBB und prefbones sind ohne verlernte klauseln immer noch sound. Wieso??

Note that there are cases where you want to keep learned clauses between SAT calls. For example you can pass the solver the set of literals of which you already know that they are part of the backbone to help the solving process. Clauses that are learned from these learned literals in combination with other clauses of the formula will also help in this, so they should also be kept. In this case, for each identified backbone literal l_b augment your formula F to $F \cup \{l_b\}$, which is actually fully equivalent to F in terms of the set of it's models, just more explicit in the fact that l_b is a backbone of it.

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.

Additionally, you can identify axiomatic literals if you check the table that is generated by CDCL when you search for a model for F. If it lists assignments that happened through unit propagation before any decision happened, then you can safely assume that they are axiomatic and part of the backbone, except of course if they were implied by modifications of F.

TODO tabelle mit wieviele literale dabei rausgefunden werden

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.

3.2.3 Implication through Cooccurrence

Another method to recognize backbone literals from other ones is described in [WBX+16] and the rationale goes as follows:

Lemma 1 Given a backbone literal a and another literal b. If b occurs in all clauses that also contain a, then $\neg b$ must be part of the backbone.

Proof: Assuming $\neg b$ was not in the backbone, then there would have to exist a model that contained b. Given that all clauses that contain b also contain a, a cannot be part of the backbone.

In other words, if we determine a new backbone literal, and all clauses that contain it also contain another literal, then we can add the negation of the latter to our backbone set.

bisschen dazu schreibseln. Diskussion unit case ist ja auch selten

refinedEssentials benchmark hat selten überhaupt irgendwelche damit festgestellt, höchstens 4, bei zwischen 200 und 400 variablen dimacs-hanoi5 hat 47

tabelle zum vergleich zwischen normalen KBB und KBB mit cooccurrence hinklatschen (je datei, nur implizierte backbone literale)

TODO überlegen, ob es sinnmacht überhaupt zwei variablen zu haben die in denselben klauseln auftauchen

File	n_{unit}	n_{coocc}
brock400-2.cnf	0	0
dimacs-hanoi5.cnf	1465	47
grieu-vmpc-s05-25.cnf	565	0
grieu-vmpc-s05-27.cnf	142	0
johnson8-2-4.cnf	0	0
fla-barthel-200-2.cnf	33	0
fla-barthel-200-3.cnf	43	0
fla-barthel-220-1.cnf	149	0
fla-barthel-220-2.cnf	0	0
fla-barthel-220-4.cnf	0	0
fla-barthel-240-2.cnf	61	0
fla-komb-200-3.cnf	166	0
fla-komb-200-5.cnf	166	0
fla-komb-220-1.cnf	190	0
fla-komb-220-3.cnf	187	0
fla-komb-220-4.cnf	188	0
fla-komb-220-5.cnf	179	3
fla-komb-240-2.cnf	211	0
fla-qhid-320-2.cnf	0	0
fla-qhid-320-5.cnf	283	2
fla-qhid-340-2.cnf	310	0
fla-qhid-340-3.cnf	309	3
fla-qhid-340-4.cnf	301	4
fla-qhid-360-1.cnf	326	0
fla-qhid-360-5.cnf	321	0
fla-qhid-380-1.cnf	344	2
fla-qhid-400-3.cnf	366	0
fla-qhid-400-4.cnf	359	0
smallSatBomb.cnf	0	0

Table 3.1: Comparison of number of backbone literals identified through cooccurrence in comparison to the number identified through unit implication.

4 Results

4.1 Tested Backbone algorithms

The following section lists the results for various backbone implementations. As this thesis focuses on the preferences approach, which is also not as thoroughly examined as other approaches, most of these variants are slight modifications of algorithm 7, where preferences are combined with blocking clauses.

The first two algorithms are *BB* (as listed in algorithm 5) and *IBB* (algorithm 3). The next algorithm (designated as *KBB*) is a variant of *BB* that whenever a backbone literal was identified through iterative testing, unit implication as described in subsection 3.2.2 is used on the set of known backbone literals, trying to identify more backbone literals in a cheap way[1]. The following variant *KBB*(*Pref*) additionally adds preferences to *KBB*. Both differ further from *BB*, in that they reduce their model not to a prime implicant, but to the required assignments in said model. TODO KBB benutzt Rotationen (keine primimplikanten). reiner effekt der unit implikation verfälscht. Problem?

Next is the original $bb-pref$ (see algorithm 6) as described in [PJ18], designated as *PB*0, followed by my own variant of *Prefbones* that uses blocking clauses (see algorithm 7). However by default it contains two optimizations.

- Instead of intersecting models, the result of the sat call is refined to a prime implicant. The results for the case where this optimization was omitted can be seen in *PB*1(*model*).
- Preferences are $forgetting$. There is a test with this omitted as well, namely *PB*1(*noAmnes*).

The algorithms after that can also be described as slight modifications of *PB*1.

- *PB*1(*invert*) is a variant where the preferences are inverted, resulting in the sat solver trying to reproduce already found solutions.
- *PB*1x simply deactivates the preferences. Algorithmically this makes *PB*1x the same is *IBB*.
- *PB*1a removes the effect of selection order from the preferences. However, when one of the preferred literals is assigned, the assigned phase is chosen from the preferences.

[1] Note that optimisations such as this one were sometimes implemented in a simple, brute-force manner for the sake of time. Here the number of easily identified literals in combination with the pure calculation time is more interesting than the overall time that includes the unit search time.

- *PB*1*b* is the opposite of *PB*1*a*. The assigned value to a preferred literal is left to underlying heuristics, but preferred literals are picked for decisions.
- *PB*1*c* is configured in a way that only a handful of preferences are set in any point in time. It can be configured with a parameter $f \in]0..1]$. The maximum number of preferred assignments in *PB*1*c* is the number of variables in the formula multiplied with f. However, the particular order in which the subset is chosen is linear beginning at the first literal, so the set of preferences is often very similar over different sat calls.
- To alleviate this, *PB*1*d* contains an additional mechanism to ensure that the chosen subset of preferences is as different as possible in every sat computation.
- *PB*1*e* takes learned literals into account. These are put into set called bb_{sub}, which always contains a lower bound of the backbone. This allows two things. First you can omit preferences that would go against required backbone assignments, which would unnecessarily waste time. Secondly, you can add another abort condition. Instead of only stopping once the blocking clause makes the formula unsatisfiable, here you can also compare the backbones lower bound (bb_{sub}) with it's upper bound, which would be the intersection of previously found implicants. You can stop, once both upper and lower bound are the same and it is sufficient to compare their size to do this. This approach can potentially save the last sat computation. TODO: unter umständen behält der sat solver sowieso gelernte literale, dann hätten andere präferenzen eh keinen effekt.
- *PB*1*f* expands upton *PB*1*e* by initially running two satcalls, where one always assigns decided literals to $\top$ and the other to $\bot$.

*PB*2 employs a different way to implement preferences. Instead of having a preferred and a remainder set of literals and first exhausting the preferred set, here only one order is set up. However the activity values that determine the order are manipulated by applying a constant factor f to those literals that are preferred. The underlying idea is, that two variables are an equally good choice for a decision if their activity values are close, but if we have to pick one of the two, the one that is preferred is definitely to better. TODO problem mit ansatz ist, dass durch bevorzugte phase, präferenz trotzdem hart ist.

*PB*3 reduces the models not to a prime implicant but to the set of required literals as described in section 3.1.2.

4.2 SAT Competition Benchmark

For the first benchmark, I collected a set of 71 files from a SAT competition TODO welcher. The SAT competitions generally use problems that are difficult to solve compared to other problems of the same file size. This is in order to encourage

	t_{full}	t_{sat}	n_{sat}
BB	25.767	25.765	275
IBB	27.086	27.07	7
KBB	25.454	25.29	42
KBB(pref)	54.729	54.619	39
PB0	57.177	57.175	5
PB1	131.063	131.061	5
PB1(noAmnes)	171.183	171.182	4
PB1(model)	116.853	116.852	6
PB1(invert)	164.994	164.992	11
PB1x	18.564	18.562	8
PB1a	154.091	154.089	7
PB1b	144.028	144.026	6
PB1c(50%)	119.571	119.569	6
PB1c(5%)	141.334	141.332	6
PB1d(50%)	131.602	131.6	6
PB1d(5%)	148.734	148.732	6
PB1e	143.675	142.621	6
PB1f	224.335	224.334	6
PB2(50%)	228.267	228.264	8
PB2(5%)	182.26	182.258	8
PB2(0.5%)	165.892	165.89	8
PB3	157.297	157.249	5

Table 4.1: Averages of 71 testfiles taken from sat competitions. The columns indicate: The full time that the calculation took in seconds; The time that was spent in the sat solver; The number of sat calls (all three values are averages).

development of solving strategies that reliably have good performance and not only for most of them. To save time during benchmarking, those files that took longer than around one minute for a single model computation were excluded, resulting in files that are generally around 30 kilobytes large. Additionally, problems where the duration for backbone computation averaged below one second were excluded, because here the small differences in the measurements could just as well be explained with external factors like the CPU throttling for a short time. To get meaningful testresults for such files, multiple testpasses should be conducted.

The averaged time to compute the backbones of these 71 testfiles can be seen in **??**. To prove that various computation strategies are implemented as good as possible[2], the second column shows the time that was only spent in the sat solver.

Taking a look at this table, we can quickly see that for these instances, all solver that employed preferences as part of their algorithm performed much worth than

[2]e.g. no noteworthy timeloss during setup of preferences

those that did not[3].

IBB hatte bug, darum bessere leistung von pb1x vgl IBB
zweite tabelle zeigen, wenn blocking clause nicht entfernt wurde

4.2.1 Benefits of unit implication

This subsection evaluates the effects of trying to recognize backbone literals through unit implication as described in section 3.2.2. Table **??** shows individual performance differences between the *BB* solver supplied by Sat4J and my *KBB* solver for comparison, as well as the backbone's size and the number of backbone literals identified through unit implication. The time columns also contain the number of sat calls that were comitted.[4]

File	$t_{BB}(n_{sat})$	$t_{KBB}(n_{sat})$	n_{unit}	$n_{backbone}$
brock400-2.cnf	0.041(254)	0.038(254)	0	0
fla-komb-400-3.cnf	1276.712(381)	1241.124(45)	336	379
dimacs-hanoi5.cnf	1.944(1931)	1.836(281)	1650	1931
vonThore42.cnf	0.018(398)	0.014(51)	345	346
grieu-vmpc-s05-27.cnf	261.091(678)	281.083(536)	142	677
fla-komb-360-4.cnf	14.937(337)	14.936(45)	292	333
fla-qhid-360-1.cnf	74.964(355)	76.032(29)	326	355
grieu-vmpc-s05-25.cnf	182.837(625)	179.914(51)	574	625
fla-qhid-360-5.cnf	20.678(350)	20.892(29)	321	349
fla-barthel-220-4.cnf	3.553(32)	2.741(32)	0	6
9012345.cnf	4.181(1483)	6.734(813)	670	1478
fla-barthel-220-2.cnf	8.183(23)	8.109(23)	0	4
1098765.cnf	1.208(938)	1.291(493)	444	921
smallSatBomb.cnf	0.013(26)	0.012(19)	7	9

Table 4.2: Benchmark results for a selection of files with a focus on the benefit of unit implication. Rows indicate: The number of backbone literals identified through unit implication; the actual number of backbone literals; Calculationtime and number of sat calls of the solvers *KBB* (using unit implication) and *BB* (not doing unit implication, otherwise the same).

The results drunter und drüber, auf einzeldateien eingehen

Begründungsvermutung: Hauptzeit geht für wenige schwierige Literale drauf, die nicht per unit implikation gefunden werden können. Könnte experimentell

[3] Which would be *BB*,*IBB*,*KBB* and $PB1x$

[4] Most of the performance differences in table **??** can be explained with other reasons than actual algorithmic differences. $fla-barthel-220-4.cnf$ for example should not be faster with *KBB* since here no backbone literal was determined through unit propagation. However I was able to reproduce this performance difference just through the order of what was called first, i.e. if the two solvers are called the other way around the timings are so as well. A possible explanation could be a power saving feature in modern processors.

	t_{calc}(sec)	t_{sat}(sec)	NSatCalls
BB	11.865	8.181	159098
IBB	6.719	2.887	15494
KBB	77.922	1.098	5528
KBB(pref)	64.666	1.55	6540
PB0	3.458	1.557	6929
PB1	3.42	1.367	6523
PB1(noAmnes)	3.436	1.403	6523
PB1(model)	1.968	1.479	6523
PB1(invert)	7.727	3.207	15234
PB1x	7.191	2.869	15734
PB1a	4.611	2.034	9322
PB1b	7.278	2.958	15234
PB1c(50%)	4.921	2.139	9644
PB1c(5%)	6.621	2.703	13941
PB1d(50%)	4.288	1.735	7373
PB1d(5%)	5.403	2.051	11225
PB1e	3.438	1.363	6379
PB1f	3.218	1.363	6361
PB2(50%)	4.838	2.129	9320
PB2(5%)	4.841	2.152	9324
PB2(0.5%)	4.951	2.118	9324
PB3	263.907	1.386	4466

Table 4.3: First execution of industry application. Values are not averaged, but summed up.

festgestellt werden, dur

reine sat zeit von kbb wurde auch gemessen: verlust durch suche liegt bei 0

TODO: BB mit KBB ohne implikationen prüfen, ob benefit durch schlechte implementierung verloren ging

4.3 Industrial benchmark

Another benchmark I applied on the various backbone algorithms was a DINGS-BUMS kindly supplied by Daimler AG. Here echtwelt zusammenhang beschreiben.

tabelle 1 mit ersten ergebnissen

ergebnisse sind über 407 beispiele gemittelt

Here we immediately see, that preferences give a much greater benefit than in the benchmark with the files from the sat competition.

pb3 interessant, weil niedrigste sat zeit und wenigsten sat calls überhaupt. Verliert wohl viel zeit mit drumrum also optimieren. TODO pb1e anschauen, wieviele literale gelernt wurden und mit sat größe gegenüberstellen. (schnitt daraus)

verbesserungen: - nicht brute force klauseln prüfen, sondern anhand des index für watched literal propagation (notiz: monoclauses werden in sat4j nicht gelernt. Sollten stattdessen eh als assumptions übergeben werden) - nur die literale rotieren die in der obermenge verbleiben - Außerdem: "bugfix" erzwungenes literal trat nicht in gelernten auf.

-> kein weiterer sat aufruf und Gesamtlaufzeit von 0.3 Sekunden

besonderheiten: - absolut alle backbone literale sind nach dem ersten sat aufruf gelernt. (*mit technischen besonderheiten, nur wenn man blocking clauses nimmt und dann ausgenommen die in der blocking clause) - model minus rotationen wird jedes mal zu genau dem backbone (prim implikante reicht nicht ganz) (heißt nicht, dass nur eine primimplikante existiert, sonst wäre PB1 genauso gut)

bedeutet auch: in diesem extremfall sind preferences unnötig

Bibliography

[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.

[ES03] Niklas Eén and Niklas Sörensson. Temporal induction by incremental sat solving. *Electr. Notes Theor. Comput. Sci.*, 89:543–560, 2003.

[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.

[WBX+16] Zipeng Wang, Yiping Bao, Junhao Xing, Xinyu Chen, and Sihan Liu. Algorithm for computing backbones of propositional formulae based on solved backbone information. 01 2016.

[WKS01] Jesse Whittemore, Joonyoung Kim, and Karem Sakallah. Satire: A new incremental satisfiability engine. *Proceedings - Design Automation Conference*, pages 542 – 545, 02 2001.