Mathematisch-Naturwissenschaftliche Fakultät

Symbolisches Rechnen

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-September 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), September 13, 2019

Abstract

Template

Contents

List of Tables

List of Algorithms

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
hab prefbones erfunden
haben prefbones paper rausgesucht
was kann ich mit prefbones noch so anstellen
wie sehen die praktischen nutzen der prefbones varianten aus.
introduction soll aus perspektive vor dem schreiben geschrieben werden (absichten)

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 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. Any set of assignments that is sufficient to satisfy a formula is called an implicant and if has no subset that isn't itself an implicant it is called a primeimplicant. The variables that do not occur in an implicant are called optional.

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 formula specific set of literals that contains all literals that occur in every model of said formula. 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.

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. This algorithm was first published as "*GRASP*" in [PMSS03].

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

[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.

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 $\bot$.

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.

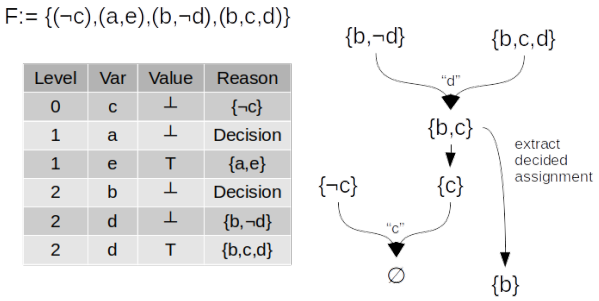

Figure 1.1: Example of a resolution. Shown is a formula, the content of the CDCL table at the point of conflict in variable d and the resolution graph.

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 assignments have a reason (a unit clause), then some of the variables in these clauses must be assigned differently(since they were unit then), so we can connect the reasons 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 of following assignments through their reasons is called *resolution* and can be briefly explained with the following formula:

$$r_\lambda(c_1, c_2) = (c_1 \setminus \{\lambda\}) \cup (c_2 \setminus \{\neg\lambda\})$$

The resolvent r consists of all literals of the two clauses that are resolved, where in this case c_1 contains λ and c_2 contains $\neg\lambda$. This precondition is given in this case,

[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.

because c_1 is the reason for the assignment of λ to $\top$ and c_2 the reason for assigning λ to $\bot$.

Figure 1.1 shows this with an example, where a conflict in variable d happened. First the clauses $\{b, \neg d\}$ and $\{c, b, d\}$ are resolved over the variable d. Then b is extracted for the learned clause, as it's reason was a decision and not a unit clause. Finally $\{c\}$ and $\{\neg c\}$ are resolved over c, resulting in the empty set. The learned clause is $\{b\}$. An alternative resolution scheme would leave literals that stem from decisions in the working set until the end, but here they are extracted on first extraction. Otherwise, these literals would be inspected multiple times.

Once collected, this set of decisions must be taken back, by reversing the assignmnents up until the first of these decisions, as this path through the search tree 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 that we gained through analysis of our conflict to prevent it from happening again.

However it is also possible that we end up with reasons for assignments that do not end in a decision. This would mean that all reasons were axiomatic assignments, for example when a formula contains a clause with only a single literal in it [4] . In this case the formula implies a contradiction and the clause that we would learn from this would be empty, unsatisfiable.

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 research 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, calls to sat solvers such as CDCL in code listings are written 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 I write $(\_, v) = SAT(F)$ or $(outc, \_) = SAT(F)$ to indicate that either *outc* or v is discarded.

[4] A simple example: A formula contains the two clauses $\{a\}$ and $\{\neg a\}$. Resolving these two results in the empty set.

Algorithm 1: CDCL ALGORITHM

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)
```

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

```
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 [JLMS15] as is described in algorithm 3.

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*, except that prime implicants are used instead of models. For specifics on these, see chapter 3.1.1.

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 a backbone literal 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 4. 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      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
```

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 5 is another one that is listed in both [MSJL10] and [JLMS15] and is implemented in the Sat4J library as BB[1].

[1] With the exception of using prime implicants instead of models, similar to the *IBB* algorithm.

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
```

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

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 2.3 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 of trying to find a more valuable model is often wasted here.

Another difficulty with the *BB* algorithm in comparison to *IBB* is that here it is more difficult to reuse learned clauses. These are based on a subset of the clauses in F. Their existence is virtual so to speak, implied through these base clauses but difficult to find. 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.

2.3 Preferences

The previous approaches still leave much of their 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 $prefs$. As already stated, 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. 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.

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

```
(_, v_r) ← SAT(F)
Repeat
    prefs ← {¬l : l ∈ v_r}
    (_, v) ← prefSAT(F, prefs)
    if v ⊇ v_r then
        return v_r                          // No preference was applied
    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 I was interested in the concrete effects of preferences by themselves on the backbone computation, I 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. Coincidentally, the added blocking clause happens to be the exact same set as that of the preferred variables.

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
```

Later in the results section I will show how this variant faired against the previous algorithm from [PJ18].

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 algorithms 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 essentially about using implicants instead of models.

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 optional variables 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[1]. 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[2] 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 it not only generates a model, but also a table containing information on how that model was generated. This information can be used in this algorithm to reduce the number of literals that need to be checked on whether they are required in the resulting prime implicant. You can quickly generate the input I_r 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.

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 your CDCL SAT solver to stop once the formula is satisfied instead of stopping once every variable was assigned. Once the formula is satisfied, no more assignment is necessary, so all further ones are arbitrary decisions that are not necessary in the implicant.

It is sufficient to go over the remaining set of assignments only once each. After having determined that a literal is required to satisfy the formula, it cannot become optional later. After all, for this the algorithm would have to add assignments instead of taking them away. And after having discarded a literal from our implicant it cannot become required again. For this to happen we would have to drop a literal from our implicant where that was the last one that satisfied some particular clause. This fits the description of a required literal and we do not drop those.

Since the same model could be calculated in different ways by CDCL depending on the particular order in which the variables in it were assigned values, the partition between decided assignments and unit assignments can be different for the same model. Therefore, this method can give you different prime implicants for the same model depending on the CDCL table that you use. Additionally, the order in which you check the decided literals can also make a difference.

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

[1] The unit case

[2] These are used in CDCL to determine and store the information on whether a clause is satisfied, unsatisfied, unit or neither of them. This is done by having two pointers that rest on unassigned literals in each clause. In this algorithm you can let these pointers rest on satisfying literals.

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
```

implicant.

Any model of a CNF formula can contain multiple implicants and even prime implicants[3] , 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. This approach was described in [JLMS15].

You can do this reduction faster if you make use of the lookup table described in the previous section, 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.

Additionally, if you happen to use a computation strategy, where you have a set of positively identifed backbone literals (for example an iterative algorithm as described in section 2.2, or when you make use of axiomatic literals as described in the next section), then of course these don't have to be tested either.

The performance benefit of this approach depends on the particular example. If the individual models of a formula contain very few primeimplicants or even only one, then looking for rotatable literals might not give a benefit over calculating the prime implicants. However formulas where individual models contain many prime implicants with large differences, the benefit can be extreme. See section 5 for experimental results.

[3] 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
```

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 negation of 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.

Additionally you should check the CDCL table that your solver creates. If you look at this table you might find variable assignments through unit implication which happened before any decision. This includes all assignments from the paragraph above, but also those that are implied through these[4].

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.

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 $\top$[5]. 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.

TODO was ist mit gelerntem auf basis von assumptions

[4] Example: Formula $\{\{a\}, \{\neg a, b\}\}$ has the backbone $\{a, b\}$, but only a is immediately obvious.
[5] Which fits the resolution formula described in section 1.1.2

3.2.2 Unit Implication

When you have some backbone literals identified, there are some methods that you can apply on this set, which can potentially expand it without a complete model calculation.

One method becomes obvious, if you recall the CDCL algorithm, specifically unit propagation. Suppose you have a clause where all but one of it's variables turned out to be part of the backbone, however all of them with the exact opposite sign from that in the clause, so that the clause is still not satisfied by them. In this case you are forced to assign the remaining literal in a way that it does satisfy the clause and you have to do this in every possible model, since what forces you to do that are backbone literals. Therefore, this unit implied assignment is in the backbone.

An efficient algorithm for this would be as follows: Keep a counter for each of the clauses in your formula that indicates the number of not satisfying backbone literals in said clause. When you identify a new backbone literal, increment all the counters where that literal occurs in negation. You can drop the counter completely for clauses where the newly identified backbone literal occurs with the same sign so that it satisfies the clause[6]. If the counter reaches the length of it's clause minus one, then you know that to satisfy this particular clause, the remaining literal in it has to be in the backbone. In this case you can compare it's literals with your current inventory of backbone literals to identify the remaining one. This algorithm should be done inbetween every SAT computation with only those backbone literals that were identified in the last iteration.

Without an efficient implementation, this search for unit implied backbone literals might consume a lot of time. If you try to find the backbone of a formula where the computation takes many SAT calls which each return relatively quickly, the time spent to search for unit implied backbone literals can actually exceed the time spent in the SAT solver if it's implemented inefficiently.

I have tested this method in two solvers, *BB* and a variant of *PB*1 that identifies backbone literals only through the learned literals. The tested files were from a SAT competition. Here it showed, that for the method described in this section it is important to supply a sufficient number of already known backbone literals for it to have a positive impact on runtime. The learned backbone literals alone could rarely supply this, before the *PrefBones* algorithm terminated from other conditions. However the iterative approach of the *BB* solver, testing every yet unidentified literal individually, was much faster at providing positively identified backbone literals that you would need to imply other backbone literals through unit implication.

For further investigation on this method see section 5.2.2.

[6] In fact, you could remove the whole clause from your formula, since it will always be satisfied through the backbone literal. You could say, it get's subsumed by a clause with only the backbone literal.

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.

File	n_{unit}	n_{coocc}	n_{var}
brock400-2.cnf	0	0	400
dimacs-hanoi5.cnf	1465	47	1931
grieu-vmpc-s05-25.cnf	565	0	625
grieu-vmpc-s05-27.cnf	142	0	729
johnson8-2-4.cnf	0	0	28
fla-barthel-200-2.cnf	33	0	200
fla-barthel-200-3.cnf	43	0	200
fla-barthel-220-1.cnf	149	0	220
fla-barthel-220-2.cnf	0	0	220
fla-barthel-220-4.cnf	0	0	220
fla-barthel-240-2.cnf	61	0	240
fla-qhid-300-1.cnf	260	0	300
fla-qhid-300-4.cnf	270	0	300
fla-qhid-320-1.cnf	296	0	320
fla-qhid-320-2.cnf	0	0	320
fla-qhid-320-5.cnf	283	2	320
fla-qhid-340-2.cnf	310	0	340
fla-qhid-340-3.cnf	309	3	340
fla-qhid-340-4.cnf	301	4	340
fla-qhid-360-1.cnf	326	0	360
fla-qhid-360-5.cnf	321	0	360
fla-qhid-380-1.cnf	344	2	380
fla-qhid-400-3.cnf	366	0	400
fla-qhid-400-4.cnf	359	0	400
smallSatBomb.cnf	0	0	264

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

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.

However this strategy does not seem to be very effective, at least for the benchmark that I tested it on. Table 3.1 shows the number of literals that were identified through this method in comparison to those that were identifed through unit implication and the number of variables in the formula. It is probably very rare that a variables always occurs together with another

3.3 Assumptions

A sub problem of calculating backbones is to calculate a backbone under an assumption. This means, that we actually want to know the Backbone of $F \cup \{l\}$ where F is our base formula and l is some variable of F.

The most straightforward way to implement this is to simply call the sat solver that we use with the same assumptions every time. However depending on the

way that assumptions are implemented in your solver, the set of learned clauses may have to be discarded or at least filtered. Reusing learned clauses is very useful when you calculate a backbone through repeating calls to the SAT solver, as table 5.2 shows.

In a CDCL SAT solver, assumptions can be implemented in two ways.
You could simply add a clause for each literal that you want to assume, with the clause containing only that literal.
Alternatively, you can do a trick with in your CDCL table by starting the SAT computation with decisions that correspond to your assumptions. Then once you have to backtrack these decisions, you know that your formula is unsatisfiable with your assumptions.

The latter technique has a very special benefit over the first. If the solver runs into a conflict, it will create a learned clause to prevent the same conflict to occur again. But the content of this learned clause differs depending on the implementation of assumptions.
With the first method your assumption is taken as ground truth, as part of your formula. The resolution step for the variable associated with the assumption will remove it from the learned clause. Without the clause that implemented the assumption, that learned clause would be too strong.
However with the latter method, where this assumption is involved with a conflict a corresponding literal will simply be added to the learned clause. That way the learned clause will later only come into effect under the circumstance that the variable that has now an assumption is then assigned the same value.

If you later want to solve the same formula with different assumptions (or with none at all), then the first strategy might result in invalid learned clauses, which would restrict the set of models of the formula. This is something a learned clause should never do.

Having a sound implementation of assumptions is especially relevant for the *BB* algorithm, because this algorithm uses a different assumption in every one[7] of it's SAT calls.

[7] Except for the first.

4 Experiments with Preferences

As is mentioned in [PJ18], the concept of preferences has not experienced much research as of yet. That is why I experimented a lot with various modifications on the concept for the purpose of this thesis.

All of the following modifications were implemented ontop of algorithm 7 (*PB*1). Algorithm 6 (PB0) depends on that all available preferences are taken into account strictly, so there is not much space to experiment there without making the algorithm unreliable.

4.1 Forgetting preferences

During the course of solving a CNF formula with the CDCL algorithm, many assignment conflicts occur which must be resolved through backtracking. This means that some assignments need to be taken back. Looking at this from the other direction, it can very well happen, that the same variable can occur in decisions multiple times over such a calculation. If a variable assignment is even slightly involved in a conflict, then, lacking any further knowledge, it is a good strategy to simply try it's negation in the future. It is not guaranteed, but simply more likely that the other assignment resolves the conflict that occurred with the previous assignment.

The default way in that preferences are implemented stands in the way of this. If you tell the SAT solver to always assign the same boolean value to a specific variable, then the strategy above cannot be applied. That is why I have implemented an option to *PB*1 where preferences are forgotten after the first time when they are taken into account. This means that when a variable is selected for a decision, it is removed from the primary heap that is consulted first for this selection[1].

The secondary heap always contains all used variables for selection, but in the default configuration where preferences are not forgotten the preferred variables would already be assigned at the point in time when it is consulted. Here this means, that once a preference was forgotten, the associated variables can still occur in decisions after they were reverted through backtracking and can potentially be assigned a different value.

The most important effect of this strategy is, that the solver quickly falls back to his default behaviour when it turns out that the formula is difficult to solve when

[1] In the Sat4J library, what decides the order of decisions and what decides the value assigned in these decisions are actually two separate data structures. This means that this removal must be written in two separate places.

preferences for the resulting model are to be taken into account.

It is quite likely that a formula contains models that are relevant for the backbone, because they contain the only occurence of some literal. However, if they contradict the current set of preferred literals in many other places, giving these preferences to the SAT solver can actually make it difficult to find said model.

4.2 Subsets

During benchmarking it showed itself, that all solvers that did not employ any preferences at all were the fastest ones when it came to difficult formulas. So it might be interesting to test some algorithm that can be configured with a floating point number, where a ground value of 0.0 would be equivalent to having no preferences at all and then allowing to enable preferences in very small increments.

The simplest way to do this would be to restrict the number of preferences that could be set for each SAT call. The parameter above is multiplied with the number of variables in the formula and then the number of preferred literals is pruned in each iteration to have at most that size, without of course discarding the set of assignments that has yet to be tested.

Aside from the straightforward implementation, I also created a variant, where extra care is taken to use different subsets for the preferences for each sat call. To do this, all preferences given to the SAT solver are remembered and not picked again until all of them were tried once. At this point the memory of previously used preferences is cleared so that it builds up again.

4.3 Nudging

The previous approach to scalable preferences still works against the decision heuristic because before the SAT solver even starts, the preferred variables are chosen. What subset of the preferred variables would be easiest for the solver to take into account cannot be said upfront. Therefore it should be much more efficient to "nudge" the existing decision heuristic in the direction of the preferences while it's running.

To do this, I implemented a different kind of preference strategy. Here, similar to the previous method, you can pass a floating point number f between 0.0 and 1.0 in addition to the set of preferred variables. When a decision happens in the CDCL algorithm, the selection heuristic sorts all variables by an activity value, which is typically based on how often it was involved in conflicts. Then the resulting array is scanned, beginning with the variable with the lowest activity until one is found that is not yet assigned.

Here this activity value is multiplied with $1 - f$ during this sorting step where a variable is preferred. Therefore, if you pass 0 as value for f, the order that is returned from sorting the activities would be completely unaffected. Very small values of f would only affect the selection heuristic if a preferred value was already deemed

very important but only in second or third place with a small distance to the first one.

However, this factor f can only influence the order of variables for decisions. Which value is then assigned can not be weighted[2]. That is why the default strategy for deciding the phase was left in place.

The purpose of this scheme was to see, whether benefits of preferences would rise up faster than the drawbacks with very low values of f.

4.4 Approaching lower and upper bounds

I also tried an idea that was shortly mentioned in [MSJL10, Chapter 3.4]:

> Another approach consists of executing enumeration and iteration based algorithms in parallel, since enumeration refines upper bounds on the size of the backbone, and iteration refines lower bounds. Such algorithm could terminate as soon as both bounds become equal.

You could argue that *BB* (see algorithm 5) matches this concept, since it reduces the set of literals to further examine with every found model. It removes both the identified backbone literals and the found models from the set of literals to test and once that becomes empty, it terminates.

However, since [MSJL10] also lists the *BB* algorithm, the authors probably ment to run two different algorithms concurrently. Still, making use of both approaches makes sense, so I tried this out in combination with preferences, by checking the set of learned literals in each iteration, as proposed in section 3.2.1.

[2] Except if there was a heuristic for that, which can give a confidence value

5 Results

5.1 Tested Backbone algorithms

This chapter contains a couple of benchmarks that were all tested against a series of backbone algorithms. There is a slight focus on the preferences approach. Since this is not as thoroughly examined as other approaches in literature, most of these variants are slight modifications of algorithm 7, where preferences are combined with blocking clauses.

For clarity, the used benchmarks are listed here. All algorithms unless otherwise stated reduce the models that they find to prime implicants with the method described in section 3.1.1.

- **BB**: Sat4J implementation of algorithm 5.
- **IBB**: Sat4J implementation of algorithm 3.
- **KBB**: Algorithm *BB* in combination with unit implication as described in section 3.2.2.
- **PB0**: *PrefBones* after [PJ18]. See algorithm 6.
- **PB1**: *PrefBones* with blocking clause. See algorithm 7.
- **PB1(amnes)**: *PB*1 with forgetting preferences as described in section 4.1.
- **PB1(model)**: *PB*1 without any model reduction.
- **PB1x**: *PB*1 without preferences. Structurally identical with *IBB*[1]
- **PB1a**: TODO nötig?
- **PB1b**: *PB*1 where preferences can only affect the selection order. Given for the purpose of comparison with *PB*2.
- **PB1c**: *PB*1 with a limited number of preferences, as described in section 4.2.
- **PB1d**: Like *PB*1*c*, but with more diverse preferences in each iteration.
- **PB1e**: *PB*1 in combination with approaching upper and lower bounds as described in section 4.4. The lower bound is made up of learned literals.

[1] There are slight differences in the performance measurements between these two implementations. These probably stem from differences in the implementation. The differences divided by the number of sat calls is relatively comparable, implying that the two implementations scale comparably.

- **PB1f**: Based on *PB*1*e*, but sollte eigenltich weg
- **PB2**: Uses scalable preferences as described in section 4.3.
- **PB3**: *PB*1 with model reduction through rotation. TODO rename

5.2 SAT Competition Benchmark

For the first benchmark, I collected a set of 64 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 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 64 testfiles can be seen in 5.1. 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 solvers that employed preferences as part of their algorithm performed much worse than those that did not[3].

	t_{full}	t_{sat}	n_{sat}
BB	8.63	8.628	254
IBB	4.946	4.944	9
KBB	8.713	8.687	36
PB0	31.78	31.779	4
PB1	17.49	17.488	5
PB1(amnes)	10.867	10.865	6
PB1(model)	25.794	25.793	5
PB1x	5.064	5.062	9
PB1a	11.241	11.239	8
PB1b	9.785	9.783	6
PB1c	25.312	25.311	5
PB1c(5%)	1064.097	1064.095	6
PB1d	26.746	26.745	5
PB1e	17.187	16.591	6
PB1f	15.971	15.97	6
PB2	10.135	10.133	10
PB2(5%)	5.232	5.229	9
PB2(0.5%)	5.251	5.249	9
PB3	20.927	20.922	4

Table 5.1: Averages of 64 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).

5.2.1 Importance of reusing learned clauses

Table 5.2 shows a comparison of individual benchmarks to highlight the importance of reusing learned clauses. While working with the *Sat*4*J* library I noticed that the *IBB* backbone algorithm was accidentally configured in a way that learned clauses would always be

[2] e.g. no noteworthy timeloss during setup of preferences
[3] Which would be *BB*,*IBB*,*KBB* and *PB*1*x*

discarded between SAT computations. However, these learned clauses are still valid in later iterations of the *IBB* algorithm. The only difference that the formula goes through during this algorithm, is that the blocking clause that ensures a new model repeatedly looses some of it's literals. As long as the set of solutions for a formula is only reduced, the learned clauses of that formula stay valid.[4]

The information contained in learned clauses is very valuable, as it prevents the solver from repeating invalid combinations of assignments that might even be likely to occur again. But if already learned clauses can guide the SAT solver away from possible conflicts, it could ideally return a model without any backtracking.

5.2.2 Benefits of unit implication

This section evaluates the effects of trying to recognize backbone literals through unit implication as described in section 3.2.2. Table 5.1 interestingly shows no performance benefit for this technique, even though the number of sat calls is much smaller.

Table 5.3[5] shows individual performance differences between the *BB* solver supplied by Sat4J and my own *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. It's content matches the results of table 5.1. Occasional cases where *KBB* trumps in performance over *BB* get balanced out by cases where this is the other way round, but the number of sat calls is always better with the *KBB* algorithm. And this is true in spite of the fact that almost all identifications happened through unit implication.

This indicates that the effect from scanning for unit implied backbone literals also appears in the *BB* solver, only not quite as obvious. When you look at line 10 of algorithm 5, you see that *BB* actually takes identified backbone literals up into the formula to speed up the solving process. Now imagine what happens, if the

[4] Example: With a formula without any clauses but three variablesyy a,b and c you can create eight models. With only the clause $\{a\}$ in your formula you can have four models, with only the clause $\{a \vee b\}$ you can have 6 models, with only the clause $\{a \vee b \vee c\}$ you can have 7 models.

[5] In this table the pure sat calculation time for the *BB* column is missing. The *BB* algorithm actually spends almost no time outside of the SAT solver in the case of the listed formulas.
Most of the performance differences in this 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 difference in timing between the first and second call are the same. A possible explanation could be a power saving feature in modern processors.

File	$t_{BB}[n_{sat}]$	$t_{KBB}(t_{sat})[n_{sat}]$	n_{unit}	$n_{backbone}$
brock400-2.cnf	0.04[254]	0.04(0.03)[254]	0	0
fla-komb-400-3.cnf	1276.71[381]	1241.12(1241.11)[45]	336	379
dimacs-hanoi5.cnf	1.94[1931]	1.84(1.23)[281]	1650	1931
vonThore42.cnf	0.02[398]	0.01(0.01)[51]	345	346
grieu-vmpc-s05-27.cnf	261.09[678]	281.08(277.1)[536]	142	677
fla-komb-360-4.cnf	14.94[337]	14.94(14.9)[45]	292	333
fla-qhid-360-1.cnf	74.96[355]	76.03(76.03)[29]	326	355
grieu-vmpc-s05-25.cnf	182.84[625]	179.91(179.44)[51]	574	625
fla-qhid-360-5.cnf	20.68[350]	20.89(20.89)[29]	321	349
fla-barthel-220-4.cnf	3.55[32]	2.74(2.73)[32]	0	6
9012345.cnf	4.18[1483]	6.73(5.92)[813]	670	1478
fla-barthel-220-2.cnf	8.18[23]	8.11(8.11)[23]	0	4
1098765.cnf	1.21[938]	1.29(0.83)[493]	444	921
smallSatBomb.cnf	0.01[26]	0.01(0.01)[19]	7	9

Table 5.3: Benchmark results for a selection of files with a focus on the benefit of unit implication. Rows indicate: Calculation time and number of SAT calls for the *BB* solver ; Calculation time, time spent in SAT solver and number of SAT calls for the *KBB* solver ; Number of backbone literals identified through unit implication (in *KBB*) ; Number of backbone variables in formula.

unit implication case happens during the course of the *BB* algorithm. You have a clause that is not satisfied, even though all but one of it's literals are already assigned through the learned backbone literals. When *BB* tests the last of that clauses literals with an assumption against the last one, the clause becomes immediately unsatisfied[6], proving the last remaining literal in the clause is part of the backbone.

The results of the next chapter paint a different picture. Here the pure SAT time of *KBB* is actually smaller than that of *BB*, however even *BB* looses around a third of it's calculation time outside of pure SAT calls. This could be explained by the much higher number of SAT calls for this benchmark and the overhead that comes with preparations and cleanups for a SAT call, which appearantly became relevant at that point. Given good conditions (like in that benchmark) *KBB* can identify many backbone literals in a single search, which also requires less data structures around it compared to a complete SAT computation.

5.2.3 Benefits of forgetting preferences

schlechte ergebnisse von PB1c (0.05) vgl run2 mit run3 (run3 hat kein default vergessen)

[6]Remember that *BB* tests by trying to disprove a potential backbone literal.

5.3 Industrial benchmark

TODO strukturieren

Another benchmark I applied on the various backbone algorithms was a formula for a real world application from the automobile industry. The purpose of this formula F_{conf} was to describe a product in the context of options or features available to the customer. Some of these options can be combined, others exclude or require each other. Most of the variables in F_{conf} correspond to a boolean parameter that is set to $\top$ if the associated feature is requested by the customer. If the formula would become unsatisfiable under such assumptions equal to the requested configuration, then the combination of these features would be impossible. F_{conf} further contains a small set of additional variables to model more complex relationships between options.

File	t_{keep}	$t_{discard}$
brock400-2.cnf	0.233	0.252
dimacs-hanoi5.cnf	1.41	1.596
grieu-vmpc-s05-25.cnf	71.945	78.964
grieu-vmpc-s05-27.cnf	554.52	648.697
fla-barthel-200-2.cnf	0.634	6.019
fla-barthel-200-3.cnf	0.619	2.16
fla-barthel-220-1.cnf	2.511	9.572
fla-barthel-220-2.cnf	7.497	17.759
fla-barthel-220-4.cnf	2.24	14.113
fla-barthel-240-2.cnf	3.632	50.552
fla-komb-200-3.cnf	0.236	0.301
fla-komb-200-5.cnf	0.114	0.276
fla-qhid-320-1.cnf	6.61	6.575
fla-qhid-320-2.cnf	9.227	101.17
fla-qhid-320-5.cnf	7.861	7.962
fla-qhid-340-2.cnf	11.922	13.688
fla-qhid-340-3.cnf	11.796	17.157
fla-qhid-340-4.cnf	9.454	9.297
fla-qhid-360-1.cnf	39.998	39.797
fla-qhid-360-5.cnf	10.698	10.849
fla-qhid-380-1.cnf	189.895	249.003
fla-qhid-400-3.cnf	83.098	130.834
fla-qhid-400-4.cnf	55.213	52.811
smallSatBomb.cnf	0.011	0.022

Table 5.2: Backbone computation time of the *IBB* algorithm, once with keeping learned clauses (t_{keep}) and once discarding learned clauses between every sat call($t_{discard}$)

The particular use case that I examined was to find the implications in the formula, i.e. if a customer requests feature a, would he also have to pick feature b. A primitive way to calculate this would be to iterate over all possible pairs of features and check for satisfiability of F_{conf} under the condition that a is $\top$ and b is $\bot$. If this was unsatisfiable, then a would imply b. However, with 407 literals to choose from, you would have to do 165,649 sat calls.

A more efficient approach is to only go over the 407 options once and for each variable a of them calculate the backbone of $F_{conf} \cup \{a\}$ or in other words F_{conf} under the assumption a. If a feature b occurs in all models when a is assumed, then a implies b in some way.

Table 5.4 lists experimental results with the same set of algorithms that were run on the files from the sat competition. After analyzing performance results, I try to give a specialized algorithm that is optimized for this particular benchmark.

We immediately see, that preferences give a much greater benefit than in the benchmark with the files from the sat competition. However there are some more things that stand out:

- The fastest algorithm overall is the one that does not reduce the models at all. The number of sat calls is even exactly the same as that of that variant with prime implicant reduction. However the time that was spent in the sat solver is very similar to that execution with prime implicant reduction (*PB*1), so the difference must be the time that was spent to reduce the model and the benefit, namely the number of optional literals in the prime implicant, was very small.

	t_{calc}(sec)	t_{sat}(sec)	NSatCalls
BB	11.117	7.235	159545
IBB	6.353	2.316	15748
KBB	13.747	2.11	15266
PB0	3.659	1.469	6531
PB1	3.918	1.611	6531
PB1(amnes)	4.067	1.682	6531
PB1(model)	2.148	1.59	6531
PB1x	7.596	2.94	15748
PB1a	5.096	2.112	9335
PB1b	7.754	3.199	15248
PB1c	5.376	2.248	9680
PB1c(5%)	7.026	2.693	13955
PB1d	4.464	1.804	7386
PB1e	3.524	1.354	6387
PB1f	3.603	1.526	6369
PB2	8.341	3.457	15752
PB2(5%)	8.313	3.484	15752
PB2(0.5%)	8.364	3.529	15752
PB3	3.872	1.204	4471

Table 5.4: First execution of industry application. Values are not averaged, but summed up over 407 different benchmarks.

- However, the fastest implementation when it comes to pure sat solver time as well as number of sat calls, was *PB*3, which differs from the others in that it uses literal rotation to reduce models instead of prime implicant computation. Apparantly the models that occur in this formula contain many different prime implicants with little distance to the model.
  A previous variant of this algorithm took around 250 seconds to compute overall. This version did not use the lookup from literal to containing clauses (as described in later paragraphs in section section 3.1.2), but iterated over all clauses in the formula. Such a drastic effect of an efficient implementation did not occur for the sat competition benchmark.
  This has primarily two reasons. First, the number of sat calls per instance is twice as much in this benchmark compared to those from the sat competitions[7], therefore the model reduction happens twice as often. Secondly, the formula of this benchmark is much larger than those from the sat competition, with the filesize being around twenty times as large. Therefore there are much more clauses to search through.

[7]The 4471 calls span over 407 problem instances.

- The two algorithms in third place of overall computation time were $PB1e$ and $PB1f$. Both of these make use of learned literals so appearantly the backbone literals in this formula are easy to identify by checking the set of literals that the solver learns when it computes new models.
  In fact, on inspection, all literals that later turn out to be part of the backbone occur in this set after the very first sat call(with an exception of the forced literal, which can be extracted by searching the formula for clauses with only one literal).

- Another noteworthy algorithm is the *KBB* algorithm that uses unit implication to identify backbone literals. The implementation of this algorithm was based on that of the *BB* algorithm and here the number of sat calls with *KBB* is a tenth of the number in the case of *BB*. However this advantage is not expressed equally strong when it comes to the time spent in the sat solver(around a fourth), which means that the identification through calculating a new model can sometimes be more efficient than regularly checking unit implications.
  The overall computation time is even worse in comparison, meaning that the bene-

> fit through cheap backbone literal identification is less than the time that it takes to check the clauses for the unit case. It may be possible that this method requires an improved implementation. The search for clauses where the unit implication for backbone literals applies could be implemented in a similar way as was used to calculate required and rotatable assignments, using the lookup from variables to clauses where they occur (see section 3.1.2)

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. - präferenzen nutzen

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)

Line 5 bis Line 8 do while breche ab sobald nix neues gefunden wird. unitImplied gibt nur neue zurück

besondere bedeutung für diesen benchmark: wiederverwendung von gelernten klauseln über unterschiedliche literale

Algorithm 10: Specialized algorithm for industrial application

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

```
1  (outc, mdl, learnt) ← SAT(F)
2  bb_u ← required(mdl, F)
3  bb_l ← learnt
4  while bb_l ≠ bb_u do
5      do
6          unitBackbones ← unitImplied(F, bb_l)
7          bb_l ← bb_l ∪ unitBackbones
8      while unitBackbones ≠ ∅
9      blocker ← ⋁_{l∈bb_u} ¬l
10     prefs ← {¬l : l ∈ bb_u}
11     (outc, mdl, learnt) ← prefSAT(F ∪ blocker ∪ bb_l, prefs)
12     if ¬outc then
13         return bb_u
14     bb_l = bb_l ∪ learnt
15     bb_u = bb_u ∩ required(mdl, F, bb_l)
16 return bb_u
```

6 Conclusions

prefbones ist besser für echtwelt beispiele, ohne prefs zuverlässiger bei komplizierten beispielen

kombination aus strategien auf fallbeispiel optimieren(?)

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.

[PMSS03] J P. Marques-Silva and K Sakallah. Grasp - a new search algorithm for satisfiability. 01 2003.

[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.