

Chapter 5

Look-Ahead Based SAT Solvers

Marijn J.H. Heule and Hans van Maaren

5.1. Introduction

Imagine that you are for the first time in New York City (NYC). A cab just dropped you off at a crossing and now you want to see the most beautiful part of town. You consider two potential strategies to get going: A *conflict-driven* strategy and a *look-ahead* strategy.

The conflict-driven strategy consists of the following heuristics: On each crossing you look in all directions and walk towards the crossing which appears most beautiful at first sight. If at a certain crossing all new directions definitely lead to dead end situations, you write the location in a booklet to avoid the place in the future and you evaluate which was the nearest crossing where you likely chose the wrong direction. You go back to that crossing and continue with a new preferred direction.

The look-ahead strategy spends much more time to select the next crossing. First all adjacent crossings are visited. At each of them all directions are observed before returning to the current crossing. Now the next crossing is selected based on all observed directions. A schematic view of both strategies is shown in Figure 5.1.

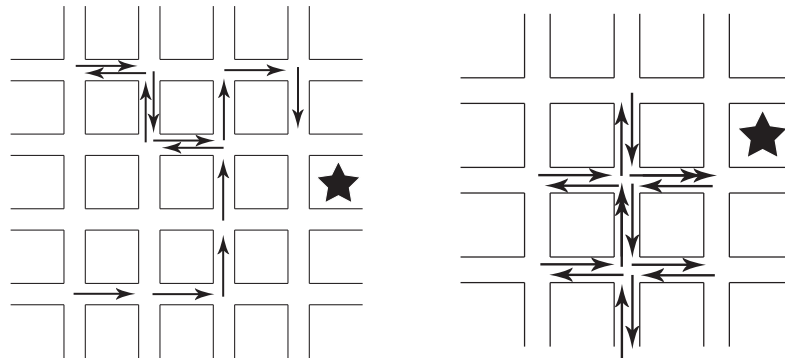


Figure 5.1. A NYC walk, conflict-driven (left) and look-ahead (right). ★ denotes the target.

As the reader has probably expected by now, the conflict-driven and look-ahead SAT solving architectures have several analogies with the conflict-driven and look-ahead NYC sightseeing strategies. Although the look-ahead NYC walk appears very costly, in practice it solves some problems quite efficiently. This chapter walks through the look-ahead architecture for SAT solvers.

5.1.1. Domain of application

Before going into the details of the look-ahead architecture, we first consider a wide range of problems and ask the question: Which architecture is the most appropriate to adequately solve a particular instance?

Traditionally, look-ahead SAT solvers are strong on random k -SAT formulae and especially on the unsatisfiable instances. [Her06] suggest that in practice, look-ahead SAT solvers are strong on benchmarks with either a low density (ratio clauses to variables) or a small diameter (longest shortest path in for instance the resolution graph¹). Figure 5.2 illustrates this. Using the structured (crafted and industrial) benchmarks from the SAT 2005 competition and SATlib, the relative performance is measured for the solvers *minisat* (conflict-driven) and *march* (look-ahead). For a given combination of density and diameter of the resolution graph, the strongest solver is plotted.

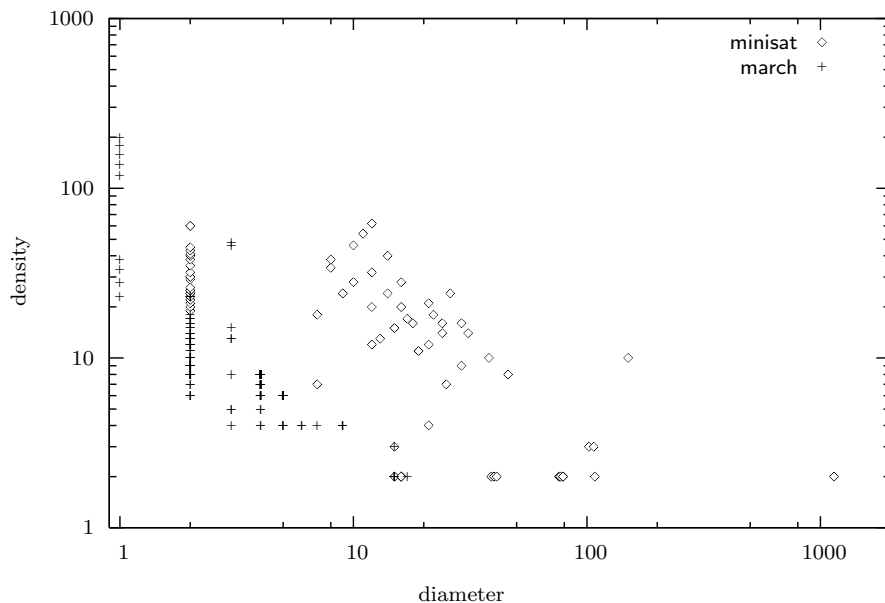


Figure 5.2. Strongest architecture on structured benchmarks split by density and diameter. Architectures represented by *minisat* (conflict-driven) and *march* (look-ahead).

¹The resolution graph is a clause based graph. Its vertices are the clauses and clauses are connected if they have exactly one clashing literal.

Notice that Figure 5.2 compares `minisat` to `march` and therefore it should not be interpreted blindly as a comparison between any conflict-driven and look-ahead SAT solvers in general. The solver `march` is the only look-ahead SAT solver that is optimized for large and structured benchmarks. Selecting a different look-ahead SAT solver would change the picture in favor of `minisat`.

However, it seems that in the current state of development both architectures have their own range of applications. The almost sharp separation shown in Figure 5.2 can be explained as follows: Density expresses the cost of unit propagation. The higher the density, the more clauses need to be reduced for each assigned variable. Therefore, look-ahead becomes very expensive for formulae with high densities, while using lazy data-structures (like `minisat`), the cost of unit propagation does not increase heavily on these formulae.

The diameter expresses the global connectivity of clauses. If just a few variables cover all literals in a set of clauses, such a set is referred to as a *local cluster*. The larger the diameter, the more local clusters occur within the formula. Local clusters reduce the effectiveness of reasoning by look-ahead SAT solvers: Assigning decision variables to a truth value will modify the formula only locally. Therefore, expensive reasoning is only expected to learn facts within the last modified cluster. On the other hand, conflict-driven solvers benefit from local clusters: Conflict clauses will likely arise from local conflicts, yielding small clauses consisting of some covering variables. Also, they may be reused frequently.

5.2. General and Historical Overview

5.2.1. The Look-Ahead Architecture

The look-ahead architecture is based on the *DPLL* framework [DLL62]: It is a complete solving method which selects in each step a decision variable x_{decision} and recursively calls DPLL for the reduced formula where x_{decision} is assigned to false (denoted by $\mathcal{F}[x_{\text{decision}} = 0]$) and another where x_{decision} is assigned to true (denoted by $\mathcal{F}[x_{\text{decision}} = 1]$).

A formula \mathcal{F} is reduced by *unit propagation*: Given a formula \mathcal{F} , an unassigned variable x and a Boolean value \mathbf{B} , first x is assigned to \mathbf{B} . If this assignment φ results in a *unit clause* (clause of size 1) then φ is expanded by assigning the remaining literal of that clause to true. This is repeated until no unit clauses are left in φ applied to \mathcal{F} (denoted by $\varphi * \mathcal{F}$) - see Algorithm 5.1. The reduced formula consists of all clauses that are not satisfied. So, $\mathcal{F}[x = \mathbf{B}] := \varphi * \mathcal{F}$.

Algorithm 5.1 UNITPROPAGATION(formula \mathcal{F} , variable x , $\mathbf{B} \in \{0, 1\}$)

```

1:  $\varphi := \{x \leftarrow \mathbf{B}\}$ 
2: while empty clause  $\notin \varphi * \mathcal{F}$  and unit clause  $y \in \varphi * \mathcal{F}$  do
3:    $\varphi := \varphi \cup \{y \leftarrow 1\}$ 
4: end while
5: return  $\varphi * \mathcal{F}$ 

```

The recursion has two kinds of leaf nodes: Either all clauses have been satisfied (denoted by $\mathcal{F} = \emptyset$), meaning that a satisfying assignment has been found, or \mathcal{F} contains an *empty clause* (a clause of which all literals have been falsified), meaning a dead end. In the latter case the algorithm backtracks.

The decision variable is selected by the LOOKAHEAD procedure. Besides selecting x_{decision} it also attempts to reduce the formula in each step by assigning *forced* variables and to further constrain it by adding clauses. Algorithm 5.2 shows the top level structure. Notice that the LOOKAHEAD procedure returns both a simplified formula and x_{decision} .

In addition, the presented algorithm uses direction heuristics to select which of the reduced formulae $\mathcal{F}[x_{\text{decision}} = 0]$ or $\mathcal{F}[x_{\text{decision}} = 1]$ should be visited first - see Section 5.3.2. Effective direction heuristics improve the performance on satisfiable instances². Although direction heuristics can be applied to all DPLL based solvers, look-ahead SAT solvers are particularly the ones that use them.

Algorithm 5.2 DPLL(formula \mathcal{F})

```

1: if  $\mathcal{F} = \emptyset$  then
2:   return satisfiable
3: end if
4:  $\langle \mathcal{F}; x_{\text{decision}} \rangle := \text{LOOKAHEAD}(\mathcal{F})$ 
5: if empty clause  $\in \mathcal{F}$  then
6:   return unsatisfiable
7: else if no  $x_{\text{decision}}$  is selected then
8:   return DPLL( $\mathcal{F}$ )
9: end if
10:  $\mathbf{B} := \text{DIRECTIONHEURISTIC}(x_{\text{decision}}, \mathcal{F})$ 
11: if DPLL( $\mathcal{F}[x_{\text{decision}} = \mathbf{B}]$ ) = satisfiable then
12:   return satisfiable
13: end if
14: return DPLL( $\mathcal{F}[x_{\text{decision}} = \neg\mathbf{B}]$ )

```

The LOOKAHEAD procedure, as the name suggests, performs *look-aheads*. A look-ahead on x starts by assigning x to true followed by unit propagation. The importance of x is measured *and* possible reductions of the formula are detected. After this analysis, it backtracks, ending the look-ahead. The rationale of a look-ahead operation is that evaluating the effect of actually assigning variables to truth values and performing unit propagation is more adequate than taking a cheap guess using some statistical data on \mathcal{F} .

This brings us to the two main features of the LOOKAHEAD procedure. The first is the *decision heuristic* which measures the importance of a variable. This heuristic consists of a *difference* or *distance heuristic* (in short DIFF) and a heuristic that combines two DIFF values (in short MIXDIFF).

A DIFF heuristic measures the reduction of the formulae caused by a look-ahead. The larger the reduction, the higher the heuristic value. This reduction can be measured with all kinds of statistics. Effective statistics are the reduction of free variables and the number of newly created (reduced, but not satisfied) clauses. The final judgment of a variable is obtained by combining $\text{DIFF}(\mathcal{F}, \mathcal{F}[x = 0])$ and $\text{DIFF}(\mathcal{F}, \mathcal{F}[x = 1])$ using a MIXDIFF heuristic. The product of these numbers is generally considered to be an effective heuristic. It aims to create a balanced search-tree. Notice that this is not a goal in its own: The trivial 2^n tree is perfectly balanced, but huge. See Chapter 7 for more details and theory.

²Direction heuristics applied in a conflict-driven setting may heavily influence performance on unsatisfiable formulae.

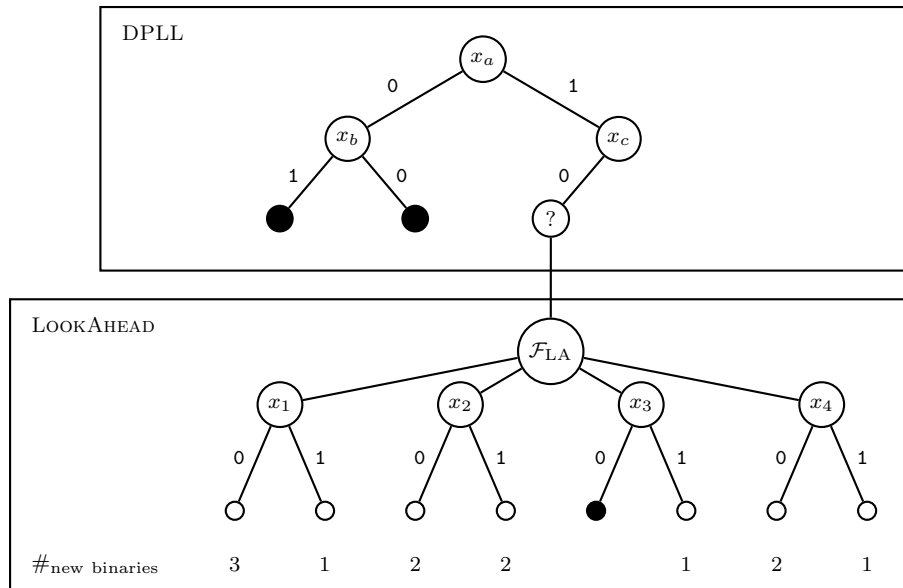


Figure 5.3. A graphical representation of the look-ahead architecture. Above, the DPLL super-structure (a binary tree) is shown. In each node of the DPLL-tree, the LOOKAHEAD procedure is called to select the decision variable and to compute implied variables by additional reasoning. Black nodes refer to leaf nodes and variables shown in the vertices refer to the decision variables and look-ahead variables, respectively.

The second main feature of the look-ahead architecture is the detection of *failed literals*: If a look-ahead on x results in a conflict, then x is forced to be assigned to false. Detection of failed literals reduces the formula because of these “free” assignments. The LOOKAHEAD procedure terminates in case both x_i and $\neg x_i$ are detected as failed literals.

Figure 5.3 shows a graphical representation of the look-ahead architecture. On the top level, the DPLL framework is used. The selection of the decision variable, reduction of the formula, and addition of learned clauses are performed by the LOOKAHEAD procedure. Black nodes refer to a dead end situation, either an unsatisfiable leaf node (DPLL) or a failed literal (LOOKAHEAD procedure).

Example 5.2.1. Consider the following example formula below:

$$\mathcal{F}_{LA} = (\neg x_1 \vee x_3) \wedge (x_1 \vee x_2 \vee x_3) \wedge (x_1 \vee \neg x_2 \vee x_4) \wedge (x_1 \vee \neg x_2 \vee \neg x_4) \wedge (x_2 \vee \neg x_3 \vee x_4)$$

Since the largest clauses in \mathcal{F}_{LA} have size three, only new binary clauses can be created. For instance, during the look-ahead on $\neg x_1$, three new binary clauses are created (all clauses in which literal x_1 occurs). The look-ahead on x_1 will force x_3 to be assigned to true by unit propagation. This will reduce the last clause to a binary clause, while all other clauses become satisfied. Similarly, we can compute the number of new binary clauses (denoted by $\#_{\text{new binaries}}$) for all look-aheads - see Figure 5.3.

Notice that the look-ahead on $\neg x_3$ results in a conflict. So $\neg x_3$ is a *failed literal* and forces x_3 to be assigned to true. Due to this forced assignment the formula changes. To improve the accuracy of the look-ahead heuristics (in this case the reduction measurement), the look-aheads should be performed again. However, by assigning forced variables, more failed literals might be detected. So, for accuracy, first iteratively perform the look-aheads until no new failed literals are detected.

Finally, the selection of the decision variable is based on the reduction measurements of both the look-ahead on $\neg x_i$ and x_i . Generally, the product is used to combine the numbers. In this example, x_2 would be selected as decision variable, because the product of the reduction measured while performing look-ahead on $\neg x_2$ and x_2 is the highest (i.e. 4). Section 7.6 discusses the preference for the product in more detail.

Several enhancements of the look-ahead architecture have been developed. To reduce the cost of the LOOKAHEAD procedure, look-aheads could be restricted to a subset of the free variables. The subset (denoted by \mathcal{P}) is selected by the PRESELECT procedure - see Section 5.3.3. However, if $|\mathcal{P}|$ is too small, this could decrease overall performance, since less failed literals will be detected and possibly a less effective decision variable is selected.

Various forms of additional look-ahead reasoning can be applied to \mathcal{F} to either reduce its size by assigning forced literals or to further constrain it by adding learned clauses. Four kinds of additional reasoning are discussed in Section 5.4.

Algorithm 5.3 LOOKAHEAD(\mathcal{F})

```

1:  $\mathcal{P} := \text{PRESELECT}(\mathcal{F})$ 
2: repeat
3:   for all variables  $x_i \in \mathcal{P}$  do
4:      $\mathcal{F} := \text{LOOKAHEADREASONING}(\mathcal{F}, x_i)$ 
5:     if empty clause  $\in \mathcal{F}[x = 0]$  and empty clause  $\in \mathcal{F}[x = 1]$  then
6:       return  $\langle \mathcal{F}[x = 0]; * \rangle$ 
7:     else if empty clause  $\in \mathcal{F}[x = 0]$  then
8:        $\mathcal{F} := \mathcal{F}[x = 1]$ 
9:     else if empty clause  $\in \mathcal{F}[x = 1]$  then
10:       $\mathcal{F} := \mathcal{F}[x = 0]$ 
11:     else
12:        $H(x_i) = \text{DECISIONHEURISTIC}(\mathcal{F}, \mathcal{F}[x = 0], \mathcal{F}[x = 1])$ 
13:     end if
14:   end for
15: until nothing (important) has been learned
16: return  $\langle \mathcal{F}; x_i \text{ with greatest } H(x_i) \rangle$ 

```

Algorithm 5.3 shows the LOOKAHEAD procedure with these enhancements. Mostly the procedure will return a simplified formula \mathcal{F} and a decision variable x_{decision} . Except in two cases this procedure returns no decision variable: First, if the procedure detects that the formula is unsatisfiable by failed literals. Second, if all (pre-selected) variables are assigned. In the latter case, we either detected a satisfying assignment or we need to restart the LOOKAHEAD procedure with a new set of pre-selected variables.

Detection of failed literals or other look-ahead reasoning that change \mathcal{F} , can influence the DIFF values. Also the propagation of a forced literal could result

in the failure of the look-head on other literals. Therefore, to improve both the accuracy of the decision heuristic and to increase the number of detected failed literals, the LOOKAHEAD procedure can be iterated until no (important) facts are learned (in the last iteration).

The unit propagation part within the LOOKAHEAD procedure is relatively the most costly aspect of this architecture. Heuristics aside, performance could be improved by reduction of these costs. Section 5.5 discusses three optimization techniques.

5.2.2. History of Look-Ahead SAT Solvers

The Böhm solver [BS96] could be considered as the first look-ahead SAT solver. In each node it selects the variable that occurs most frequently in the shortest active (not satisfied) clauses. Although no look-aheads are performed, it uses *eager data-structures* which are common in look-ahead SAT solvers: Two dimensional linked lists are used to cheaply compute the occurrences in active clauses. This data-structure is also used in the OKsolver. The Böhm solver won the first SAT competition in 1991/1992 [BKB92].

The first SAT solver with a LOOKAHEAD procedure, **posit**, was developed by Freeman [Fre95] in 1995. It already contained aspects of the important heuristics used in the “modern” look-ahead SAT solvers:

- Decision variables are selected by a DIFF heuristic that measures the reduction of free variables. The MIXDIFF heuristic used in **posit**, is still used in most look-ahead SAT solvers. Let $L := \text{DIFF}(\mathcal{F}, \mathcal{F}[x = 0])$ and $R := \text{DIFF}(\mathcal{F}, \mathcal{F}[x = 1])$ then MIXDIFF is computed by $1024 \cdot LR + L + R$. The product is selected to create a balanced search-tree. The addition is for tie-breaking purposes. The factor 1024, although seemingly arbitrary, gives priority to the product (balancedness of the emerging search-tree).
- A direction heuristic is used to improve the performance on satisfiable instances: Using a simple heuristic, it estimates the relative costs of solving $\mathcal{F}[x = 0]$ and $\mathcal{F}[x = 1]$. The smallest one is preferred as first branch.
- Variables in **posit** are pre-selected for the LOOKAHEAD procedure using three principles: First, important variables (ones with a high estimated MIXDIFF) are selected to get an effective decision variable. Second, literals that occur frequently negated in binary clauses are selected (so possibly not their complement) as candidate failed literals. Third, many variables are selected near the root of the tree, because here, accurate heuristics and failed literals have more impact on the size of the tree.

Results on hard random 3-SAT formulae were boosted by **satz**. This solver was developed by Li and Anbulagan in 1997. All heuristics were optimized for these benchmarks:

- The DIFF heuristic measures the newly created binary clauses in a weighted manner based on the literals in these binary clauses. This heuristic is called *weighted binaries heuristic* and is discussed in Section 5.3.1.2. The same MIXDIFF heuristic is used as in **posit**.
- No direction heuristics are used. $\mathcal{F}[x_{\text{decision}} = 1]$ is always preferred.

- Variables are pre-selected using the prop_z heuristic [LA97a] - see Section 5.3.3.1. On random 3-SAT formulae, it pre-selects most free variables near the root of the search-tree. As the tree deepens, the number of selected variables decreases.

In 1999 Li added a `DOUBLELOOK` procedure to `satz` and further reduced the size of the search-tree to solve random formulae [Li99]. This procedure - see Section 5.4.3 - attempts to detect more failed literals by also performing some look-aheads on a second level of propagation: If during the look-ahead on x more than a certain number of new binary clauses are created, then additional look-aheads are performed on the reduced formula $\mathcal{F}[x = 1]$ to check whether it is unsatisfiable. The parameter (called Δ_{trigger}) which expresses the number of new binary clauses to trigger the `DOUBLELOOK` procedure is fixed to 65 based on experiments on hard random 3-SAT formulae. Performance clearly improves using this setting, while on many structured benchmarks it results in a slowdown: The procedure is executed too frequently.

To exploit the presence of so-called equivalence clauses in many benchmarks, Li [Li00] created a special version of `satz` - called `eqsatz` - which uses equivalence reasoning. During the look-ahead phase, it searches binary equivalences $x_i \leftrightarrow x_j$ in the reduced formulae $\mathcal{F}[x = 0]$ and $\mathcal{F}[x = 1]$. Using five inference rules, it reasons about the detected equivalences. Due to this equivalence reasoning, `eqsatz` was the first SAT solver to solve the hard `parity32` benchmarks [CKS95] without performing Gaussian elimination on the detected equivalence clauses.

Starting from 1998, Oliver Kullmann developed the `OKsolver`, motivation by the idea to create a “pure” look-ahead SAT solver [Kul02]. The heuristics are kept clean. They are not stuffed with magic constants:

- The `DIFF` heuristic in `OKsolver` measures the newly created clauses in a weighted manner - based on the size on the new clauses. For each of the used weights a separate experiment has been performed. This *clause reduction heuristic* is discussed in Section 5.3.1.1. The `MIXDIFF` uses the product $\text{DIFF}(\mathcal{F}, \mathcal{F}[x = 0]) \cdot \text{DIFF}(\mathcal{F}, \mathcal{F}[x = 1])$ which is based on the τ -function [KL99].
- The direction heuristics prefer $\mathcal{F}[x_{\text{decision}} = 0]$ or $\mathcal{F}[x_{\text{decision}} = 1]$ which is probabilistically the most satisfiable. Details are shown in Section 5.3.2.
- No pre-selection heuristics are used.

Besides the clean heuristics, `OKsolver` also adds some reasoning to the look-ahead architecture:

- **Local Learning**³: If the look-ahead on x assigns y to true, then we learn $x \rightarrow y$. This can be stored as binary clause $\neg x \vee y$. These learned binary clauses are valid under the partial assignment. This means that they should be removed while backtracking.
- **Autarky Reasoning**: If during the look-ahead on x no new clauses are created then an autarky is detected. This means that $\mathcal{F}[x = 1]$ is satisfiability equivalent with \mathcal{F} . Therefore, x can be freely assigned to true. If

³Local learning is an optimal feature in the `OKsolver` which was turned off in the version submitted to SAT 2002.

only one new clause is created, then an 1-autarky is detected. In this case, several local learned binary clauses can be added. Details are presented in Section 5.4.2.

- **Backjumping:** While backtracking, the OKsolver maintains a list of the variables that were responsible for detected conflicts. If a decision variable does not occur in the list, the second branch can be skipped. This technique has the same effect as the ones used in conflict-driven solvers.

To maximize the number of learned facts, the LOOKAHEAD procedure is iterated until in the last iteration no new forced literals have been detected and no new binary clauses have been added. During the first SAT competition⁴ (2002), OKsolver won both divisions of random k -SAT benchmarks.

The first version of `march`, developed by Heule et al. in 2002 was inspired by `satz` and applied most of its features. The most significant difference was a pre-processing technique which translated any formula to 3-SAT. The updated version of 2003, `march_eq`, was modified to become a look-ahead SAT solver for general purposes. New techniques were added to increase performance on structured instances, while still being competitive on random formulae. Three techniques have been added to improve efficiency. Due to some overhead these techniques are cost neutral on random formulae, but generate significant speed-ups on many structured instances:

- The binary and non-binary clauses of the formula are stored in separate data-structures. This reduces the required memory (especially on structured benchmarks) and facilitates a more efficient unit propagation - see Section 5.5.1.
- Before entering the LOOKAHEAD procedure, all inactive (satisfied) clauses are removed from the data-structures. This reduces the costs of unit propagation during this procedure - see Section 5.5.2.
- By building implication trees of the binary clauses containing two pre-selected variables, a lot of redundancy in the LOOKAHEAD procedure can be tackled - see Section 5.5.3.

Also equivalence reasoning was added. Instead of searching for equivalence clauses in all reduced formulae, `march_eq` only extracts them from the input formula. They are stored in a separate data-structure to implement a specialized unit propagation procedure. The solver won two divisions of crafted benchmarks during SAT 2004.

The `knfs` solver by Dubois and Dequen, developed in 2004, is in many aspects similar to `satz`. However, it features some important improvements with respect to hard random k -SAT formulae:

- The most effective modification is a minor one. Like `satz`, the DIFF heuristic measures the newly created binary clauses in a weighted manner - based on occurrences of the variables in these clauses. However, instead of adding the weighted literals, they are multiplied per binary clause - see Section 5.3.1.3. This *backbone search heuristic (BSH)* significantly improves the performance on random k -SAT formulae.

⁴see www.satcompetition.org

- The implementation is optimized for efficiency on random k -SAT formulae.
- Normalization has been developed for *BSH*, such that it can measure newly created clauses of any size. This is required for fast performance on random k -SAT formulae with $k > 3$.

The *knfs* solver performs very strong on random formulae on the SAT competitions. It won one random division during both SAT 2003 and SAT 2004, and even two random divisions during SAT 2005.

In 2005 two adaptive heuristics were added to *march_eq*, resulting in *march_dl*:

- It was observed [HvM06a] that there is some correlation between the number of failed literals and the optimal number of pre-selected free variables. Based on this observation, an adaptive heuristic was added which aims to converge to the optimal value.
- An adaptive algorithm for the *DOUBLELOOK* procedure [HvM07] has been added to *march_eq*. It modifies Δ_{trigger} after each look-ahead. In contrast to earlier implementations of the *DOUBLELOOK* procedure, it now reduces the computational costs on almost all formulae.

5.3. Heuristics

As the previous section showed, the look-ahead architecture enables one to invoke many heuristic features. Research tends to focus on three main categories of heuristics, which we recall here:

- **Difference heuristics:** To measure the difference between the formulae before and after a look-ahead. The quality of the used difference heuristic influences the actual impact of the decision variable.
- **Direction heuristics:** Given a decision variable x_{decision} , one can choose whether to examine first the positive branch $\mathcal{F}[x_{\text{decision}} = 1]$ or the negative branch $\mathcal{F}[x_{\text{decision}} = 0]$. Effective direction heuristics improve the performance on satisfiable formulae.
- **Pre-selection heuristics:** To reduce the computational costs, look-ahead can be restricted to a subset of the variables. As a possible negative consequence, however, a smaller pre-selected set of variables may result in fewer detected failed literals and a less effective decision variable.

5.3.1. Difference heuristics

This section covers the branching heuristics used in look-ahead SAT solvers. A general description of these heuristics is offered in Chapter 7. To measure the difference between a formula and its reduction after a look-ahead (in short *DIFF*), various kinds of statistics can be applied. For example: The reduction of the number of free variables, the reduction of the number of clauses, the reduced size of clauses, or any combination. *Posit* uses the reduced number of free variables as *DIFF* [Fre95]. All look-ahead SAT solvers which participated in the SAT competitions use a *DIFF* based on newly created (reduced, but not satisfied) clauses. The set of new clauses created during the look-ahead on x is denoted by $\mathcal{F}[x = 1] \setminus \mathcal{F}$.

All these heuristics use weights to quantify the relative importance of clauses of a certain size. The importance of a clause of size k is denoted by γ_k and the subformula of \mathcal{F} which contains only the clauses of size k is denoted by \mathcal{F}_k . This section discusses and compares three main DIFF heuristics.

5.3.1.1. Clause reduction heuristic

The DIFF implemented by Kullmann in OKsolver [Kul02] uses only the γ_k weights and is called *clause reduction heuristic*⁵ (*CRH*):

$$CRH(x_i) := \sum_{k \geq 2} \gamma_k \cdot |\mathcal{F}[x_i = 1]_k \setminus \mathcal{F}| \quad \text{with}$$

$$\begin{aligned} \gamma_2 &:= 1, \quad \gamma_3 := 0.2, \quad \gamma_4 := 0.05, \quad \gamma_5 := 0.01, \quad \gamma_6 := 0.003 \\ \gamma_k &:= 20.4514 \cdot 0.218673^k \text{ for } k \geq 7 \end{aligned}$$

These constants γ_k for $k = 3, 4, 5, 6$ are the result of performance optimization of the OKsolver on random k -SAT formulae, while the formula for $k \geq 7$ is the result of linear regression [Kul02].

5.3.1.2. Weighted binaries heuristic

Weighted binaries heuristic (*WBH*) is developed by Li and Anbulagan [LA97b] and applied in the solver *satz*. Each variable is weighted (both positive and negative) based on its occurrences in the formula. Let $\#(\neg x)$ denote the number of occurrences of literal x . Each new binary clause $x \vee y$ is weighted using the sum of the weights of their complementary literals. The sum $\#(\neg x) + \#(\neg y)$ expresses the number of clauses on which resolution can be done with $x \vee y$.

Occurrences in a clause of size k is valued 5^{k-3} . This weighting function followed from performance optimization of *satz* over random formulae. Notice that as in the OKsolver (i.e. γ_k in *CRH*) clauses of size $k + 1$ are about $\frac{1}{5}$ of the importance of clauses of size k .

$$w_{WBH}(x_i) := \sum_{k \geq 2} \gamma_k \cdot \#_k(x_i) \quad \text{with } \gamma_k := 5^{k-3}$$

$$WBH(x_i) := \sum_{(x \vee y) \in \{\mathcal{F}[x_i=1]_2 \setminus \mathcal{F}_2\}} (w_{WBH}(\neg x) + w_{WBH}(\neg y))$$

5.3.1.3. Backbone search heuristic

Backbone search heuristic (*BSH*) was developed by Dubois and Dequen [DD01]. This heuristic is inspired by the concept of the *backbone* of a formula - the set of variables that has a fixed truth value in all assignments satisfying the maximum

⁵This heuristic is also known as *MNC* and is further discussed in Section 7.7.4.1.

number of clauses [MZK⁺99]. Like *WBH*, variables are weighted based on their occurrences in clauses of various sizes. Clauses of size $k + 1$ are considered only half⁶ as important than clauses of size k .

The most influential difference between *WBH* and *BSH* is that the latter multiplies the weights of the literals in the newly created binary clauses - while the former adds them. For every new clause $x \vee y$, the product $\#(\neg x) \cdot \#(\neg y)$ expresses the number of resolvents that can be created due to the new clause.

$$w_{BSH}(x_i) := \sum_{k \geq 2} \gamma_k \cdot \#_k(x_i) \quad \text{with } \gamma_k := 2^{k-3}$$

$$BSH(x_i) := \sum_{(x \vee y) \in \mathcal{F}[x_i=1]_2 \setminus \mathcal{F}} (w_{BSH}(\neg x) \cdot w_{BSH}(\neg y))$$

Notice that both $w_{BSH}(x_i)$ and $BSH(x_i)$ express the importance of literal x_i . Since $BSH(x_i)$ is far more costly to compute than $w_{BSH}(x_i)$, BSH could only improve performance if it proves to be a better heuristic than w_{BSH} . So far, this claim only holds for random formulae.

Based on the assumption that $BSH(x_i)$ is better than $w_{BSH}(x_i)$, Dubois and Dequen [DD01] propose to iterate the above by using $BSH(x_i)$ instead of $w_{BSH}(x_i)$ for the second iteration to improve the accuracy of the heuristic. This can be repeated by using the $BSH(x_i)$ values of iteration i (with $i > 2$) to compute $BSH(x_i)$ of iteration $i + 1$. However, iterating BSH makes it much more costly. In general, performing only a single iteration is optimal in terms of solving time.

In [DD03] Dubois and Dequen offer a normalization procedure for BSH such that it weighs *all* newly created clauses - instead of only the binary ones. The normalization is required for fast performances on random k -SAT formula with $k > 3$. Backbone Search Renormalized Heuristic (in short *BSRH*) consists of two additional aspects: 1) Smaller clauses are more heavier using the γ_k weights, and 2) the w_{BSH} values to compute $BSH(x_i)$ are renormalized, by dividing them by the average weight of all literals in $\mathcal{F}[x_i = 1] \setminus \mathcal{F}$ - denoted by $\mu_{BSH}(x_i)$.

$$\mu_{BSH} := \frac{\sum_{C \in \mathcal{F}[x_i=1] \setminus \mathcal{F}} \sum_{x \in C} w_{BSH}(\neg x)}{\sum_{C \in \mathcal{F}[x_i=1] \setminus \mathcal{F}} |C|}$$

$$BSRH(x_i) := \sum_{C \in \mathcal{F}[x_i=1] \setminus \mathcal{F}} \left(\gamma_{|C|} \cdot \prod_{x \in C} \frac{w_{BSH}(\neg x)}{\mu_{BSH}(x_i)} \right)$$

Similar to BSH , $BSRH$ can be iterated to improve accuracy.

⁶Although the various descriptions of the heuristic [DD01, DD03] use a factor 2, the actual implementation of the heuristic in *kcdfs* (see <http://www.laria.u-picardie.fr/~dequen/sat/>) also uses factor 5.

5.3.1.4. Comparison of DIFF heuristics

Example 5.3.1. Consider the unsatisfiable 3-SAT formula $\mathcal{F}_{\text{heuristic}}$ below:

$$\begin{aligned} \mathcal{F}_{\text{heuristic}} = & (\neg x_1 \vee x_2 \vee \neg x_6) \wedge (\neg x_2 \vee \neg x_3 \vee \neg x_4) \wedge (\neg x_1 \vee \neg x_2 \vee \neg x_7) \wedge \\ & (\neg x_1 \vee \neg x_3 \vee x_4) \wedge (x_1 \vee \neg x_3 \vee x_4) \wedge (x_1 \vee \neg x_3 \vee \neg x_4) \wedge \\ & (x_2 \vee x_3 \vee \neg x_4) \wedge (\neg x_2 \vee x_3 \vee x_5) \wedge (\neg x_2 \vee x_3 \vee \neg x_5) \wedge \\ & (x_1 \vee x_2 \vee x_6) \wedge (x_1 \vee x_2 \vee x_7) \wedge (x_2 \vee x_4 \vee \neg x_5) \wedge (x_2 \vee x_4 \vee x_5) \end{aligned}$$

Table 5.1 compares the heuristic values of look-aheads on $\mathcal{F}_{\text{heuristic}}$ using the three heuristics *CRW*, *WBH*, and *BSH*. Notice that *CRH* prefers x_2 as decision variable, while *WBH* and *BSH* prefer x_3 . The latter choice will result in a smaller search-tree. Between *WBH* and *BSH* the preferred ordering differs as well. For *BSH*, variables x_4 and x_5 are relatively more important than for *WBH*.

BSH is the most effective heuristic on random k -SAT formulae. The relative effectiveness of these heuristics on structured instances is not well studied yet.

Table 5.1. Comparison between the heuristic values of the look-ahead on the variables in $\mathcal{F}_{\text{heuristic}}$ using *CRH*, *WBH* and *BSH*. The product is used as MIXDIFF.

	$\#(x_i)$	$\#(\neg x_i)$	MIXDIFF(<i>CRH</i>)	MIXDIFF(<i>WBH</i>)	MIXDIFF(<i>BSH</i>)
x_1	4	4	$4 \cdot 4 = 16$	$24 \cdot 21 = 504$	$30 \cdot 27 = 810$
x_2	6	3	$3 \cdot 6 = 18$	$17 \cdot 33 = 516$	$20 \cdot 40 = 800$
x_3	3	4	$4 \cdot 3 = 12$	$30 \cdot 23 = 690$	$56 \cdot 36 = 2016$
x_4	4	3	$3 \cdot 4 = 12$	$21 \cdot 24 = 504$	$36 \cdot 36 = 1296$
x_5	2	2	$2 \cdot 2 = 4$	$15 \cdot 15 = 225$	$27 \cdot 27 = 729$
x_6	1	1	$1 \cdot 1 = 1$	$7 \cdot 7 = 49$	$12 \cdot 12 = 144$
x_7	1	1	$1 \cdot 1 = 1$	$10 \cdot 7 = 70$	$24 \cdot 12 = 288$

5.3.2. Direction heuristics

Various state-of-the-art satisfiability (SAT) solvers use *direction heuristics* to predict the sign of the decision variables: These heuristics choose, after the selection of the decision variable, which Boolean value is examined first. Direction heuristics are in theory very powerful: If always the correct Boolean value is chosen, satisfiable formulae would be solved without backtracking. Moreover, existence of perfect direction heuristics (computable in polynomial time) would prove that $\mathcal{P} = \mathcal{NP}$. These heuristics are also discussed in Section 7.9.

Although very powerful in theory, it is difficult to formulate effective direction heuristics in practice. Look-ahead SAT solvers even use complementary strategies to select the first branch - see Section 5.3.2.1.

Direction heuristics may bias the distribution of solutions in the search-tree. For instance, while using *march* or *kcdfs*, such a biased distribution is observed on hard random 3-SAT formulae. A biased distribution can be used to replace the DPLL *depth-first* search by a search that visits subtrees in order of increasing likelihood of containing a solution - see Section 5.3.2.2.

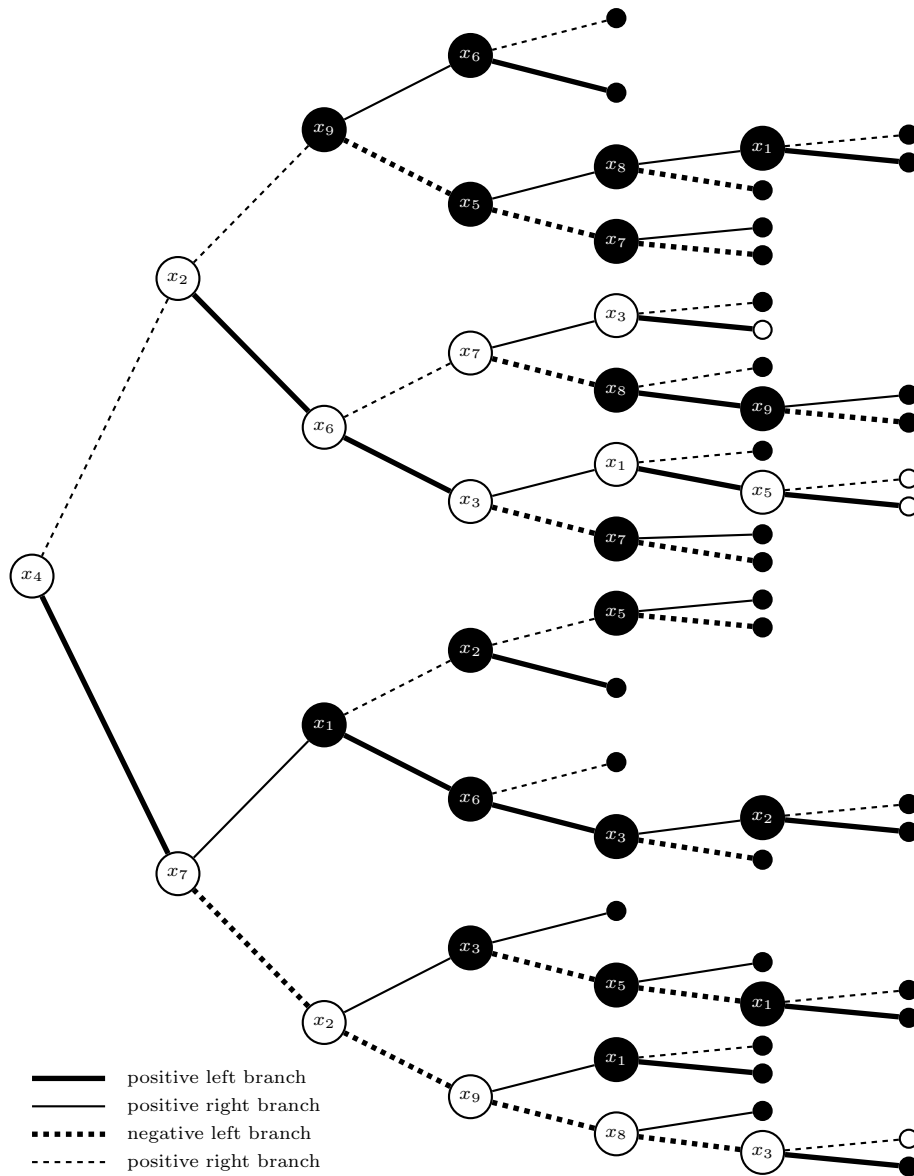


Figure 5.4. Complete binary search-tree (DPLL) for a formula with nine variables (x_1, \dots, x_9) . The decision variables are shown inside the internal nodes. A node is colored black if all child nodes are unsatisfiable, and white otherwise. The type of edge shows whether it is visited first (left branch), visited last (right branch), its decision variable is assigned to true (positive branch), or its decision variable is assigned to false (negative branch).

The search-tree of a DPLL-based SAT solver can be visualized as a binary search-tree, see Figure 5.4. This figure shows such a tree with decision variables drawn in the internal nodes. Edges show the type of each branch. We will refer to the *left branch* as the subformula that is visited first. Consequently, the *right branch* refers to the one examined later. A black leaf refers to an unsatisfiable dead end, while a white leaf indicates that a satisfying assignment has been found. An internal node is colored black in case both its children are black, and white otherwise. For instance, at depth 4 of this search-tree, 3 nodes are colored white. This means that at depth 4, 3 subtrees contain a solution.

5.3.2.1. Complementary strategies

A wide range of direction heuristics is used in look-ahead SAT solvers:

- **kcdfs**⁷ selects $\mathcal{F}[x_{\text{decision}} = 1]$ if x_{decision} occurs more frequently in \mathcal{F} than $\neg x_{\text{decision}}$. Otherwise it selects $\mathcal{F}[x_{\text{decision}} = 0]$.
- **march** selects $\mathcal{F}[x_{\text{decision}} = 1]$ if $\text{DIFF}(\mathcal{F}, \mathcal{F}[x_{\text{decision}} = 1])$ is smaller than $\text{DIFF}(\mathcal{F}, \mathcal{F}[x_{\text{decision}} = 0])$, and $\mathcal{F}[x_{\text{decision}} = 0]$ otherwise [HDvZvM04].
- **OKsolver** selects the subformula with the smallest probability that a random assignment will falsify a random formula of the same size [Kul02]. It prefers the minimum sum of

$$\sum_{k \geq 2} -|\mathcal{F}_k| \cdot \ln(1 - 2^{-k}) \quad (5.1)$$

- **posit** selects $\mathcal{F}[x_{\text{decision}} = 0]$ if x_{decision} occurs more often than $\neg x_{\text{decision}}$ in the shortest clauses of \mathcal{F} . Otherwise it selects $\mathcal{F}[x_{\text{decision}} = 1]$ [Fre95].
- **satz**⁸ does not use direction heuristics and starts with $\mathcal{F}[x_{\text{decision}} = 1]$.

These heuristics can be divided into two strategies:

A) *Estimate which subformula will require the least computational time.*

This strategy assumes that it is too hard to predict whether a subformula is satisfiable. Therefore, if both subformulae have the same expectation of containing a solution, you will find a solution faster by starting with the one that requires less computational time. From the above solvers, **posit** uses this strategy.

Notice that conflict-driven SAT solvers may prefer this strategy to focus on the most unsatisfiable subformula. This may result in an early conflict, and possibly a short conflict clause. However, since the current look-ahead SAT solvers do not add conflict clauses, this argument is not applicable for these solvers.

B) *Estimate which subformula has the highest probability of being satisfiable.*

If a formula is satisfiable, then it is expected that the subformula which is heuristically the most satisfiable will be your best bet. The other subformula may be unsatisfiable and - more importantly - a waste of time. The solvers **kcdfs**, **march**, and **OKsolver** use this strategy.

Generally these two strategies are complementary: The subformula which can be solved with relatively less computational effort is probably the one which is more constraint and therefore has a smaller probability being satisfiable. In practice,

⁷version 2006

⁸version 2.15.2

for example, satisfiable hard random k -SAT formulae are solved faster by using a direction heuristic based on strategy B , while on various structured satisfiable instances strategy A is more successful.

The *balancedness* of the DPLL search-tree could hint which is the best strategy to use: In a balanced search-tree, solving $\mathcal{F}[x_{\text{decision}} = 0]$ requires almost as much effort as solving $\mathcal{F}[x_{\text{decision}} = 1]$ for each decision variable x_{decision} . In this case, strategy A seems less appealing. Since look-ahead solvers produce a balanced search tree on hard random k -SAT formulae, strategy B would be preferred.

On the other hand, in an unbalanced search-tree one could get lucky using strategy A which would be therefore preferred. On many structured instances look-ahead SAT solvers produce an unbalanced search-tree - although the MIX-DIFF heuristic attempts to achieve a balanced search-tree.

Based on this explanation, `march_ks` uses both strategies: Strategy B is used (as implemented in `march_dl`) in case $\text{DIFF}(\mathcal{F}, \mathcal{F}[x = 0])$ and $\text{DIFF}(\mathcal{F}, \mathcal{F}[x = 1])$ are *comparable*. Otherwise, strategy A is used by preferring the complementary branch which would have been selected by `march_dl`. In `march_ks`, “comparable” is implemented as $0.1 \leq \frac{\text{DIFF}(\mathcal{F}, \mathcal{F}[x=0])}{\text{DIFF}(\mathcal{F}, \mathcal{F}[x=1])} \leq 10$.

5.3.2.2. Distribution of solutions

Given a family of benchmark instances, we can observe the effectiveness of the direction heuristic used in a certain solver on that particular family. The observation can be illustrated as follows: Consider the subtrees at depth d of the DPLL search-tree. Subtrees are denoted by T_i with $i = \{1, \dots, 2^d\}$ and are numbered in depth-first order. The histogram showing the number of formulae in the given family containing at least one solution in T_i is called a *solution distribution plot*.

Figures 5.5 and 5.6 show solution distribution plots at depth 12 for `march_dl` and `kcnfs` respectively on 3000 random 3-SAT formulae with 350 variables and 1491 clauses (at phase transition density). The observed distribution is clearly biased for both solvers. The distribution observed using `march_dl` is more biased than the one using `kcnfs`. Based on these plots, one could conclude that the direction heuristics used in `march_dl` are more effective for the instances at hand.

For both solvers, the number of formulae with at least one solution in T_i is highly related to the number of left branches that is required to reach T_i . With this in mind, the DPLL depth first search can be replaced [HvM06b] by a search strategy that, given a *jump depth*, visits the subtrees at that depth in the order of required left branches. Using jump depth 12, first subtree T_1 is visit (twelve left branches, zero right branches), followed by subtrees $T_2, T_3, T_5, T_9, \dots, T_{2049}$ (eleven left branches, one right branch), etc. Random satisfiable formulae of the same size are on average solved much faster using modified DPLL search.

Notice that for conflict-driven solvers this kind of solution distribution plots could not be made. The most important reason is that there is no clear left and right branch, because conflict clauses assign former decision variables to the complement of the preferred truth value. Another difficulty to construct these plots is the use of restarts in conflict-driven solvers. Finally, the use of lazy data-structures makes it very expensive to apply a direction heuristic that requires much statistical data.

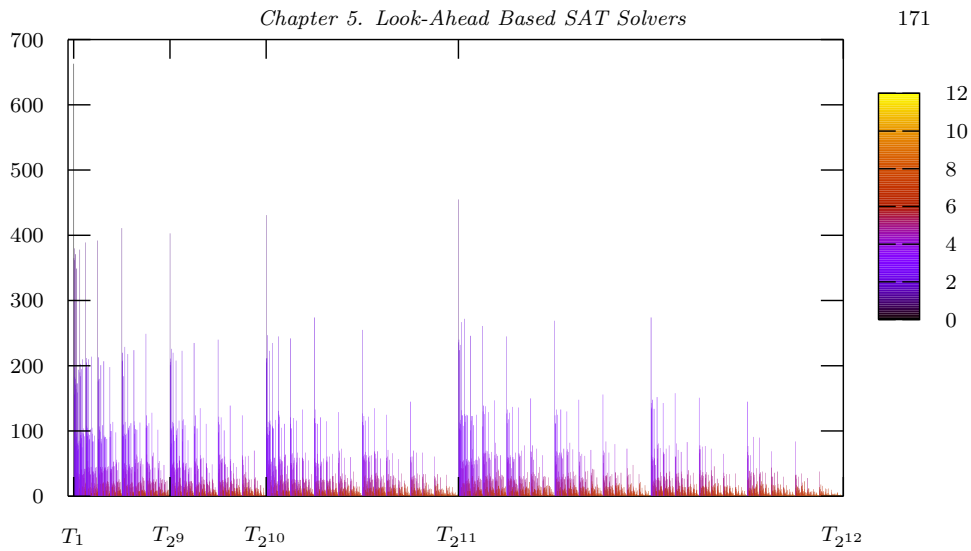


Figure 5.5. Solution distribution plot with `march_dl` showing for each subtree T_i at depth 12 the number of formulae that have at least one solution in that subtree. Used 3000 random 3-SAT formulae with 350 variables and 1491 clauses.

5.3.3. Pre-selection heuristics

Overall performance can be gained or lost by performing look-ahead on a subset of the free variables in each node of the DPLL-tree: Gains are achieved by the reduction of computational costs, while losses are the result of either the inability of the *pre-selection heuristics* (heuristics that determine the set of variables to enter the look-ahead phase) to select effective decision variables or to predict candidate failed literals. When look-ahead is performed on a subset of the variables, only a subset of the failed literals is most likely detected. Depending on the formula, this could increase the size of the DPLL-tree. This section describes the two most commonly used pre-selection heuristics in look-ahead SAT solvers: *prop_z* and *clause reduction approximation*.

5.3.3.1. *prop_z*

Li [LA97a] developed the *prop_z* heuristic for his solver `satz`, which is also used in `kcdfs`. It pre-selects variables based on their occurrences in binary clauses. The *prop_z* heuristic is developed to perform faster on hard random k -SAT formulae. Near the root of the search-tree it pre-selects all free variables. From a certain depth on, it pre-selects only variables that occur both positive and negative in binary clauses. It always pre-selects a lower bound of 10 variables.

Although this heuristic is effective on hard random k -SAT formulae, its behavior on structured instances is not clear. On some benchmarks it will pre-select all free variables in all nodes of the search-tree, while on others it always pre-selects slightly more variables than the lower bound used.

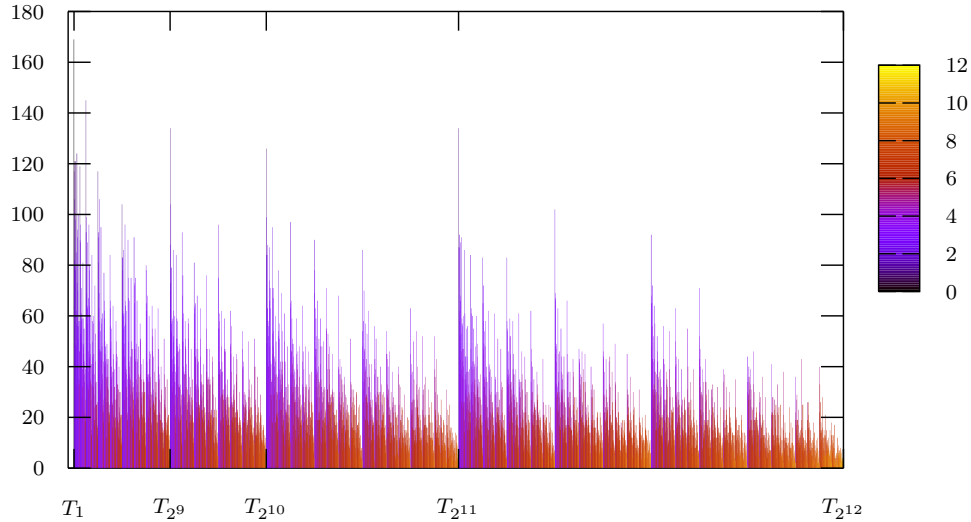


Figure 5.6. Solution distribution plot with knfs showing for each subtree T_i at depth 12 the number of formulae that have at least one solution in that subtree. Used 3000 random 3-SAT formulae with 350 variables and 1491 clauses.

5.3.3.2. Clause reduction approximation

In march the pre-selection heuristics are based on an approximation of a decision heuristic that counts the number of newly created clauses, called *clause reduction approximation (CRA)*. It uses the sizes of all clauses in the formula and is therefore more costly than prop_z . First, the set of variables to be assigned during look-ahead on x is approximated. All variables that occur with $\neg x$ in binary clauses are used as approximated set. Second, if y_i occurs in this set than all n -ary clauses in which $\neg y_i$ occurs will be reduced. This is an approximation of the number of newly created clauses, because some of the reduced ones will be satisfied. Third, the product is used as MIXDIFF. Let $\#_{>2}(x_i)$ denote the number of occurrences of literal x_i in clauses with size > 2 . For all free variables CRA is computed as:

$$CRA(x) := \left(\sum_{x \vee y_i \in \mathcal{F}} \#_{>2}(\neg y_i) \right) \cdot \left(\sum_{\neg x \vee y_i \in \mathcal{F}} \#_{>2}(\neg y_i) \right) \quad (5.2)$$

Variables with the highest CRA scores are pre-selected for the LOOKAHEAD procedure. Once the CRA scores are computed, one has to determine how many variables should be pre-selected. The optimal number varies heavily from benchmark to benchmark and from node to node in the search-tree. Early march versions used a fraction of the original number of variables. Most versions used 10% of the number of variables, which is referred to as $\text{RANK}_{10\%}$. Later versions use an adaptive heuristic discussed in the next paragraph.

Table 5.2. Average number of variables selected for the LOOKAHEAD procedure by RANK_{10%} and *prop_z* for 300 variables and 1275 clauses random 3-SAT problems.

depth	#freeVars	prop _z	RANK _{10%}	depth	#freeVars	prop _z	RANK _{10%}
1	298.24	298.24	30	11	264.55	26.81	30
2	296.52	296.52	30	12	260.53	21.55	30
3	294.92	293.89	30	13	256.79	19.80	30
4	292.44	292.21	30	14	253.28	19.24	30
5	288.60	280.04	30	15	249.96	19.16	30
6	285.36	252.14	30	16	246.77	19.28	30
7	281.68	192.82	30	17	243.68	19.57	30
8	277.54	125.13	30	18	240.68	19.97	30
9	273.17	71.51	30	19	237.73	20.46	30
10	268.76	40.65	30	20	234.82	20.97	30

5.3.3.3. Adaptive Ranking

As observed in [HDvZvM04], the optimal number of pre-selected variables is closely related to the number of detected failed literals: When relatively many failed literals were detected, a larger pre-selected set appeared optimal. Let $\#failed_i$ be the number of detected failed literals in node i . To exploit this correlation, the average number of detected failed literals is used as an indicator for the (maximum) size of the pre-selected set in node n (denoted by RANKADAPT _{n}):

$$RANKADAPT_n := L + \frac{S}{n} \sum_{i=1}^n \#failed_i \tag{5.3}$$

In the above, parameter L refers to the lower bound of RANKADAPT _{n} (namely when the average tends to zero) and S is a parameter modelling the importance of failed literals. `March_eq` uses $L := 5$ and $S := 7$ based on experiments on structured and random benchmarks. Notice that the above adaptive pre-selection heuristics are heavily influenced by the decision heuristics - which in turn are also affected by these heuristics.

Generally RANKADAPT select a subset of the free variables, but in some cases - due to many detected failed literals - all free variables are selected. It may occur that, during the LOOKAHEAD procedure, all selected variables are forced. In that case a new set of variables is pre-selected.

5.4. Additional Reasoning

Selection of decision variables by performing many look-aheads is very expensive compared to alternative decision heuristics. As such, this could hardly be considered as an advantage. However, since the decision heuristics are already costly, some additional reasoning is relatively cheap. This section presents various techniques that can be added to the LOOKAHEAD procedure in order to improve overall performance.

Look-ahead on variables which will not result in a conflict appears only useful to determine which variable has the highest decision heuristic value. However, by applying some additional reasoning, look-ahead on some variables can also be used to reduce the formula (like failed literals). Look-ahead on the other variables can be used to add *resolvents* (learned clauses) to further constrain the formula. Three kinds of additional reasoning are used in look-ahead SAT solvers for these purposes: *Local learning* (Section 5.4.1), *autarky detection* (Section 5.4.2), and *double look-ahead* (Section 5.4.3).

5.4.1. Local learning

During the look-ahead on x , other variables y_i can be assigned by unit propagation. Some due to the presence of binary clauses $\neg x \vee (\neg)y_i$, called *direct implications*. Variables assigned by other clauses are called *indirect implications*. For those variables y_i that are assigned to true (or false) by a look-ahead on x though indirect implications, a binary clause $\neg x \vee y_i$ (or $\neg x \vee \neg y_i$, respectively) can be added to the formula. This is referred to as *local learning*. As the name suggests, these clauses are not globally valid and should therefore be removed while backtracking.

Example 5.4.1. Consider the formula $\mathcal{F}_{\text{learning}}$ below:

$$\mathcal{F}_{\text{learning}} := (\neg x_1 \vee x_2) \wedge (\neg x_1 \vee \neg x_2 \vee x_3) \wedge (\neg x_1 \vee \neg x_3 \vee x_4) \wedge (x_1 \vee x_3 \vee x_6) \wedge (\neg x_1 \vee x_4 \vee \neg x_5) \wedge (x_1 \vee \neg x_6) \wedge (x_4 \vee x_5 \vee x_6) \wedge (x_5 \vee \neg x_6)$$

Some look-aheads on literals $(\neg)x_i$ will result in unit clauses, thereby assigning other variables. These assignments are listed below:

$$\begin{array}{ll} [x_1 := 0] \rightarrow \{x_3=1, x_6=0\} & [x_4 := 0] \rightarrow \{\} \\ [x_1 := 1] \rightarrow \{x_2=1, x_3=1, x_4=1\} & [x_4 := 1] \rightarrow \{\} \\ [x_2 := 0] \rightarrow \{x_1=0, x_3=1, x_6=0\} & [x_5 := 0] \rightarrow \{x_4=1, x_6=0\} \\ [x_2 := 1] \rightarrow \{\} & [x_5 := 1] \rightarrow \{\} \\ [x_3 := 0] \rightarrow \{\} & [x_6 := 0] \rightarrow \{\} \\ [x_3 := 1] \rightarrow \{\} & [x_6 := 1] \rightarrow \{x_1=1, x_2=1, x_3=1, x_4=1, x_5=1\} \end{array}$$

Eight of the above assignments follow from indirect implications. Therefore, these can be added to $\mathcal{F}_{\text{learning}}$:

$$\begin{array}{cccc} x_1 \vee x_3 & \neg x_1 \vee x_4 & x_2 \vee \neg x_6 & x_4 \vee x_5 \\ \neg x_1 \vee x_3 & x_2 \vee \neg x_3 & x_3 \vee \neg x_6 & x_4 \vee \neg x_6 \end{array}$$

In the example above, the number of binary clauses that could be added by local learning is equal to the number of original clauses in $\mathcal{F}_{\text{learning}}$. In most real world examples, the number of local learned clauses is significantly larger than the number of original clauses. Therefore, adding all these binary clauses is generally not useful because it slows down propagation.

5.4.1.1. Necessary assignments

A cheap technique to reduce the size of the formula is detection of *necessary assignments*: If the look-ahead on x_i assigns x_j to true *and* the look-ahead on $\neg x_i$ assigns x_j to true then the assignment x_j to true is referred to as necessary. This technique requires only the short storage of implications. First, store all implications while performing look-ahead on x_i . Then continue with look-ahead on $\neg x_i$ and check for each implication whether it is in the stored list. All matches are necessary assignments. After look-ahead on $\neg x_i$ the stored implications can be removed.

In Example 5.4.1, variable x_3 is necessarily assigned to true because it is implied by both x_1 and $\neg x_1$. Necessary assignments can also be detected by adding local learned clauses: When these clauses are added, look-ahead on the complement of the necessary assignment will fail. Notice that by searching for necessary assignments, variables which are not in the pre-selected set can be forced to an assignment.

Searching for necessary assignments is a simplified technique of learning in the Stålmarck’s proof procedure [SS98]. This patented procedure learns the intersection of the new clauses in the reduced formulae (after look-ahead on x_i and $\neg x_i$). Necessary assignments are only the new unit clauses in this intersection. One could learn even more by adding clauses of size 2 and larger, but none of the current look-ahead SAT solvers adds these clauses because computing them is quite expensive. However, the HeerHugo SAT solver [GW00], inspired by the Stålmarck’s proof procedure, performs the “full” learning. The theory regarding these and other strengthenings of unit clause propagation are discussed in [Kul99a, Kul04].

5.4.1.2. Constraint resolvents

The concept of necessary assignments is not sufficient to detect all forced literals by using local learning. For example, after adding the learned clauses $\neg x_1 \vee x_4$ and $x_4 \vee x_5$, look-ahead on $\neg x_4$ will fail, forcing x_4 to be assigned to true. Now, the question arises: Which local learned clauses should be added to detect all forced literals?

Local learned clauses are useful when the number of assigned variables is increased during look-ahead on the complement of one of its literals. For instance, given a formula with clauses $x_1 \vee x_2$ and $\neg x_2 \vee x_3$. The local learned clause $x_1 \vee x_3$ is not useful because look-ahead on $\neg x_1$ would already assign x_3 to true and look-ahead on $\neg x_3$ would already assign x_1 to true.

However, given a formula with clauses $x_1 \vee x_2$ and $x_1 \vee \neg x_2 \vee x_3$ then local learned clause $x_1 \vee x_3$ is useful: Only after adding this clause, look-ahead on $\neg x_3$ will assign x_1 to true. These useful local learned clauses are referred to as *constraint resolvents* [HDvZvM04].

In Example 5.4.1, all local learned clauses except $x_2 \vee \neg x_6$ are constraint resolvents. Yet, in practice, only a small subset of the local learned clauses are constraint resolvents. In Section 5.5.1 a technique is explained to efficiently detect constraint resolvents.

Constraint resolvents and heuristics. Pre-selection heuristics rank variables based on their occurrences in binary clauses - see Section 5.3.3. Addition of constraint resolvents (or local learned clauses in general) will increase the occurrences in binary clauses of variables in the pre-selected set in particular. So, once variables are pre-selected, their rank will rise, increasing the chances of being pre-selected again in a next node. Therefore, it becomes harder for “new” variables to enter the look-ahead phase.

5.4.2. Autarky Reasoning

An *autarky* (or autark assignment) is a partial assignment φ that satisfies all clauses that are “touched” by φ . So, all satisfying assignments are autark assignments. Autarkies that do not satisfy all clauses can be used to reduce the size of the formula: Let $\mathcal{F}_{\text{touched}}$ be the clauses in \mathcal{F} that are satisfied by an autarky. The remaining clauses $\mathcal{F}^* := \mathcal{F} \setminus \mathcal{F}_{\text{touched}}$ are satisfiability equivalent with \mathcal{F} - see Chapter 11. So, if we detect an autark assignment, we can reduce \mathcal{F} by removing all clauses in $\mathcal{F}_{\text{touched}}$.

5.4.2.1. Pure literals

The smallest example of an autarky is a *pure literal*: A literal which negation does not occur in the formula. Consider the example formula below:

$$\mathcal{F}_{\text{pure literal}} := (x_1 \vee \neg x_2) \wedge (\neg x_1 \vee x_3) \wedge (\neg x_2 \vee \neg x_3)$$

Variable x_2 occurs only negated in this formula - making $\neg x_2$ a pure literal. Assigning x_2 to false will therefore only satisfy clauses. Under this assignment the only clause left is $(\neg x_1 \vee x_3)$. Now both $\neg x_1$ and x_3 are pure literals. So by assigning a pure literal to true could make other literals pure.

5.4.2.2. Autarky detection by look-ahead

Look-aheads can also be used to detect autarkies. Whether a look-ahead resulted in an autark assignment requires a check that all reduced clauses became satisfied. Recall that the *CRH* and the *BSH* look-ahead evaluation heuristics count (and weight, respectively) the newly created clauses. While using these heuristics, an autarky detection check can be computed efficiently: All reduced clauses are satisfied if and only if their heuristic value is 0.

Example 5.4.2. Consider the formula $\mathcal{F}_{\text{autarky}}$ below.

$$\mathcal{F}_{\text{autarky}} := (x_1 \vee \neg x_2) \wedge (\neg x_1 \vee x_5) \wedge (\neg x_1 \vee x_2 \vee \neg x_3) \wedge (x_3 \vee x_4) \wedge (\neg x_2 \vee \neg x_3 \vee \neg x_4) \wedge (x_2 \vee x_4 \vee x_5)$$

The heuristic values of the *CRH* look-ahead evaluation heuristic (in this case the number of newly created binary clauses) of all possible look-aheads on $\mathcal{F}_{\text{autarky}}$ are shown in Table 5.3. Both the look-aheads on $\neg x_3$ and $\neg x_5$ result in a heuristic value of 0. The first represents the autark assignment $\varphi = \{x_3 = 0, x_4 = 1\}$, and the second represents the autarky assignment $\varphi = \{x_1 = x_2 = x_5 = 0\}$.

Table 5.3. Number of new binary clauses after a look-ahead on x_i on $\mathcal{F}_{\text{autarky}}$. For this formula these values equal $CRH(x_i)$ because the largest clause has length 3.

	x_1	$\neg x_1$	x_2	$\neg x_2$	x_3	$\neg x_3$	x_4	$\neg x_4$	x_5	$\neg x_5$
$CRH(x_i)$	2	2	1	2	2	0	1	2	1	0

In general, look-ahead SAT solvers do not check whether a look-ahead satisfies all remaining clauses - thereby detecting a satisfying assignment. By detecting autarkies, satisfying assignments will also be detected as soon as they appear.

5.4.2.3. Look-ahead autarky resolvents

In case a look-ahead satisfies all but one of the reduced clauses, we have almost detected an autarky. In Example 5.4.2 the look-ahead on x_2 , x_4 , and x_5 created only one clause that was not satisfied: $\neg x_3 \vee \neg x_4$, $\neg x_2 \vee \neg x_3$, and $x_2 \vee x_4$ respectively. Due to these binary clauses none of the variables is forced to a value.

However, we know that if such a particular clause were satisfied, we would have detected an autarky. This knowledge can be added to the formula: In case of the look-ahead on x_2 , if $\neg x_3 \vee \neg x_4$ is satisfied x_2 is true. So $\neg x_3 \rightarrow x_2$ and $\neg x_4 \rightarrow x_2$. The binary clauses of these implications $x_3 \vee x_2$ and $x_4 \vee x_2$ are referred to as *look-ahead autarky resolvents* – see also Section 11.13.2 and [Kul99c, Kul99b]. These can be added to the formula to further constrain it. Generally, let C be the only reduced clause which is not satisfied after the look-ahead on x . For all literals l_i in C we can add to the formula an look-ahead autarky resolvent $x \vee \neg l_i$.

Look-ahead autarky resolvents and heuristics. Although look-ahead autarky resolvents further constrain a formula, their addition could contribute to a significant drop in performance of a solver. The reason for this paradox can be found in the effect of look-ahead autarky resolvents on the heuristics: According to difference heuristics, variables are important if they create many new clauses. Therefore, variables on which look-ahead results in only a single new clause should be considered unimportant. However, the heuristic values of *WBH*, *BSH*, and *CRA* increases after adding these look-ahead autarky resolvents. This makes it more likely that unimportant variables are chosen as decision variable. Therefore, it is useful to neglect these clauses while computing the heuristics.

5.4.3. Double look-ahead

Due to the design of MIXDIFF heuristics in look-ahead SAT solvers, unbalanced variables are rarely selected as decision variables. To compensate for this, Li developed the DOUBLELOOK procedure [Li99]. This procedure performs look-aheads on a second level of propagation (on the reduced formula after a look-ahead) to increase the number of detected forced literals.

We refer to the *double look-ahead* on x as calling the DOUBLELOOK procedure on the reduced formula $\mathcal{F}[x = 1]$. A double look-ahead on x is called successful if the DOUBLELOOK procedure detects that $\mathcal{F}[x = 1]$ is unsatisfiable. Otherwise it is called unsuccessful. Similar to a failed look-ahead on x , a successful double

look-ahead on x detects a conflict in $\mathcal{F}[x = 1]$ and thereby forces x to be assigned to false.

Although double look-aheads are expensive, they can be a relative efficient method to detect some forced literals in practice. In order to reduce the solving time, many double look-aheads must be successful. Prediction of the success of a double look-ahead is therefore essential.

Double look-ahead heuristics. The heuristics regarding double look-aheads focus on the success predictor. Li suggests to use the number of newly created binary clauses as an effective predictor [Li99]. If the number of newly created binary clauses during look-ahead on x (denoted by $|\mathcal{F}[x = 1]_2 \setminus \mathcal{F}|$) is larger than a certain parameter (called Δ_{trigger}) than the DOUBLELOOK procedure should be triggered. All double look-ahead heuristics are concerned with the optimal value of Δ_{trigger} .

The first implementation of the DOUBLELOOK procedure used a static value for Δ_{trigger} . Li implemented $\Delta_{\text{trigger}} := 65$ in his solver `satz`. This magic constant is based on experiments on hard random 3-SAT formulae. Although these instances can be solved much faster using this setting, on structured instances it frequently results in the call of too many double look-aheads, which will reduce overall performance.

Another static implementation for Δ_{trigger} is found in `knfs` by Dubois and Dequen. They use $\Delta_{\text{trigger}} := 0.17 \cdot n$, with n referring to the original number of variables. This setting also arises from experiments on random 3-SAT formulae. Especially on random and structured instances with large n , better results are obtained compared to $\Delta_{\text{trigger}} := 65$.

A first dynamic heuristic was developed by Li for a later version of `satz`. It initializes $\Delta_{\text{trigger}} := 0.17 \cdot n$. If a double look-ahead on x is unsuccessful, Δ_{trigger} is updated to $\Delta_{\text{trigger}} := |\mathcal{F}[x = 1]_2 \setminus \mathcal{F}|$. The motivation for the update is as follows: Assuming the number of newly created binary clauses is an effective predictor for success of a double look-ahead, and because the double look-ahead on x was unsuccessful, Δ_{trigger} should be at least $|\mathcal{F}[x = 1]_2 \setminus \mathcal{F}|$. After a successful DoubleLook call, Δ_{trigger} is reset to $\Delta_{\text{trigger}} = 0.17 \cdot n$. In practice, this dynamic heuristic “turns off” the DOUBLELOOK procedure on most structured instances. The procedure is only rarely triggered due to the first update rule. The performance on these instances is improved to back to normal (i.e. not calling the DOUBLELOOK at all).

A second dynamic heuristic was developed by Heule and Van Maaren for their solver `march_dl` [HvM07]. It initializes $\Delta_{\text{trigger}} := 0$. Like the dynamic heuristic in `satz`, it updates $\Delta_{\text{trigger}} = |\mathcal{F}[x = 1]_2 \setminus \mathcal{F}|$ after an unsuccessful double look-ahead on x . The important difference is that Δ_{trigger} is not decreased after a successful doublelook, but Δ_{trigger} is slightly reduced after each look-ahead. Therefore, double look-aheads are performed once in a while. On random instances this dynamic heuristic yields comparable performances on random formulae. However, the use of the DOUBLELOOK procedure with this heuristic improves the performance on many structured instances as well.

5.4.3.1. Double look-ahead resolvents

An unsuccessful double look-ahead is generally not a total waste of time. For example, the look-ahead on x_i triggers the DOUBLELOOK procedure. On the second level of propagation three failed literals are detected: x_r , $\neg x_s$, and x_t . Thus we learn $x_i \rightarrow \neg x_r$, $x_i \rightarrow x_s$, and $x_i \rightarrow \neg x_t$ - equivalent to the binary clauses $\neg x_i \vee \neg x_r$, $\neg x_i \vee x_s$, $\neg x_i \vee \neg x_t$. Since the DOUBLELOOK procedure performs additional look-aheads on free variables in $\mathcal{F}[x_i = 1]$, we know that \mathcal{F} does not contain these binary clauses - otherwise they would have been assigned already.

We refer to these binary clauses as *double look-ahead resolvents*. In case a double look-ahead is unsuccessful, double look-ahead resolvents can be added to \mathcal{F} to further constrain the formula. On the other hand, a successful double look-ahead on x_i will force x_i to false, thereby satisfying all these resolvents.

Adding double look-ahead resolvents can be useful to reduce overall costs caused by the DOUBLELOOK procedure. If a look-ahead on x_i triggers this procedure and appears to be unsuccessful, it is likely that it will trigger it again in the next node of the search-tree. By storing the failed literals at the second level of propagation as resolvents, the DOUBLELOOK procedure does not have to perform costly look-aheads to detect them again.

Double look-ahead resolvents and heuristics. Since the DOUBLELOOK procedure is triggered when a look-ahead on literal $(\neg)x_i$ creates many new binary clauses, the DIFF values for those literals is relatively high. Adding double look-ahead resolvents will further boost these DIFF values. This increases the chance that variables are selected as decision variable because the DIFF of either its positive or negative literal is very large. Concluding: By adding double look-ahead resolvents, decision heuristics may select decision variables yielding a more unbalanced search-tree.

5.5. Eager Data-Structures

Unit propagation is the most costly part of state-of-the-art complete SAT solvers. Within conflict-driven solvers, the computational costs of unit propagation is reduced by using lazy data-structures such as 2-literal watch pointers. Yet, for look-ahead SAT solvers these data-structures are not useful: The costs of unit propagation in these solvers is concentrated in the LOOKAHEAD procedure due to the numerous look-aheads. To compute the difference heuristics (as presented in Section 5.3.1), one requires the sizes of the reduced clauses. By using lazy data-structures, these sizes cannot be computed cheaply. However, by using *eager data-structures* the unit propagation costs can be reduced. The use of eager data-structures originates from the Böhm SAT solver [BS96]. This section offers three techniques to reduce the costs of unit propagation:

- **Efficient storage of binary clauses:** Binary clauses can be stored such that they require only half the memory compared to conventional storage. In general, structured instances consists mostly of binary clauses. This significantly reduces the storage of the formula. Such a way of storage

reduces the cost of unit propagation and makes it possible to cheaply detect constraint resolvents - see Section 5.5.1

- **Removal of inactive clauses:** Many of the original clauses become inactive (satisfied) down in the search-tree. Look-ahead can be performed faster if these clauses are removed from the data-structures - see Section 5.5.2
- **Tree based look-ahead:** Many propagations made during the look-ahead phase are redundant. A technique called tree-based look-ahead reduces this redundancy using implication trees - see Section 5.5.3.

5.5.1. Binary implication arrays

Generally, clauses are stored in a clause database together with a look-up table for each literal in which clauses it occurs. Such a storage is relatively expensive for binary clauses: Instead of storing the clauses, one could add the implications of assigning a literal to true directly in the “look-up table” [PH02]. We refer to such a storage as *binary implication arrays*. Notice that the latter data-structure requires only half the space compared to the former one. Figure 5.7 shows a graphical example of both data-structures for four binary clauses. Storing clauses in separate binary and non-binary (n -ary) data-structures reduces the computational costs of unit propagation: 1) Cache performance is improved because of the cheaper storage, and 2) No look-up is required for binary clauses when they are stored in binary implication arrays.

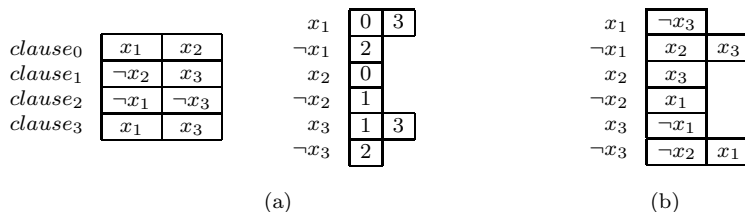


Figure 5.7. (a) A structure with binary clauses stored in a clause database, and literal look-up arrays containing the indices that point to the clauses in which they occur. (b) Binary clauses stored in implication arrays.

Besides the cheaper storage, splitting a formula in binary clauses (\mathcal{F}_2) and n -ary clauses ($\mathcal{F}_{>2}$) is also useful to efficiently detect constraint resolvents: Recall that constraint resolvents consist of the complement of the look-ahead literal and a literal that only occurs in a unit clause that originates from an n -ary clause. By performing unit propagation in such a way that unit clauses resulting from clauses in \mathcal{F}_2 are always preferred above those in $\mathcal{F}_{>2}$, the latter can be used to construct constraint resolvents. Algorithm 5.4 shows this *binary clause preferred (BCP) unit propagation* including the addition of constraint resolvents. Notice that this procedure returns both an expanded formula with learned constraint resolvents ($\mathcal{F}_2 \cup \mathcal{F}_{>2}$) and a reduced formula by the look-ahead ($\mathcal{F}[x = 1]$).

Algorithm 5.4 BCPUNITPROPAGATION(formula \mathcal{F} , variable x)

```

1:  $\varphi := \{x \leftarrow 1\}$ 
2: while empty clause  $\notin \varphi * \mathcal{F}_2$  and a unit clause  $\in \varphi * \mathcal{F}_2$  do
3:   while empty clause  $\notin \varphi * \mathcal{F}_2$  and unit clause  $\{y\} \in \varphi * \mathcal{F}_2$  do
4:      $\varphi := \varphi \cup \{y \leftarrow 1\}$ 
5:   end while
6:   if empty clause  $\notin \varphi * \mathcal{F}_{>2}$  and unit clause  $\{y\} \in \varphi * \mathcal{F}_{>2}$  then
7:      $\mathcal{F}_2 := \mathcal{F}_2 \cup \{\neg x \vee y\}$ 
8:   end if
9: end while
10: return  $\mathcal{F}_2 \cup \mathcal{F}_{>2}, \varphi * \mathcal{F}$ 

```

5.5.2. Removal of inactive clauses

The presence of inactive clauses increases the computational costs of unit propagation during the LOOKAHEAD procedure. Two important causes can be observed: First, the larger the number of clauses considered during a look-ahead, the poorer the performance of the cache. Second, if both active and inactive clauses occur in the data-structure during the look-ahead, a check is required to determine the status of every clause. Removal of inactive clauses from the data-structure prevents these unfavorable effects from taking place.

All satisfied clauses are clearly inactive clauses. In case clauses are stored in separate binary and n -ary data-structures (see Section 5.5.1), then also clauses become inactive if they are represented in both data-structures: If an n -ary clause becomes a binary one and is added to the binary clause data-structure, it can be removed from the n -ary data-structure.

5.5.3. Tree-based look-ahead

Suppose the LOOKAHEAD procedure is about to perform look-ahead on the free variables x_1 , x_2 , and x_3 on a formula that contains the binary clauses $x_1 \vee \neg x_2$ and $x_1 \vee \neg x_3$. Obviously, the look-ahead on x_1 will assign all variables that are forced by $x_1 = 1$. Also, due to the binary clauses, the look-ahead on x_2 and x_3 will assign these variables (amongst others). Using the result of the look-ahead on x_1 , the look-aheads on x_2 and x_3 can be performed more efficiently.

Generally, suppose that two look-ahead literals share a certain implication. In this simple case, we could propagate the shared implication first, followed by a propagation of one of the look-ahead literals, backtracking the latter, then propagating the other look-ahead literal and finally backtracking to the initial state. This way, the shared implication has been propagated only once.

Figure 5.8 shows this example graphically. The implications (from the binary clauses) among x_1 , x_2 and x_3 form a small tree. Some thought reveals that this process, when applied recursively, could work for arbitrary trees. Based on this idea - at the start of each look-ahead phase - trees can be constructed from the implications between the literals selected for look-ahead, in such a way that each literal occurs in exactly one tree. By recursively visiting these trees, the LOOKAHEAD procedure is more efficient. Of course, the more dense the implication graph which arises from the binary clauses, the more possibilities are available to form trees. Adding all sorts of resolvents will in many cases be an important catalyst for the effectiveness of this method.

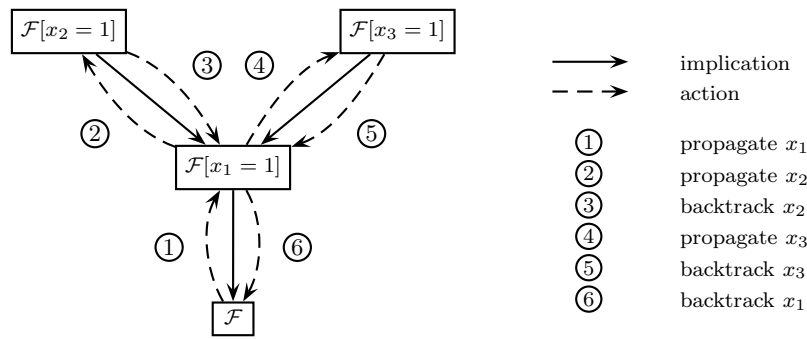


Figure 5.8. Graphical form of an implication tree with corresponding actions.

References

- [BKB92] Michael Buro and Hans Kleine Büning. Report on a sat competition, 1992.
- [BS96] Max Böhm and Ewald Speckenmeyer. A fast parallel sat-solver - efficient workload balancing. *Ann. Math. Artif. Intell.*, 17(3-4):381–400, 1996.
- [CKS95] J. M. Crawford, M. J. Kearns, and R. E. Schapire. The minimal disagreement parity problem as a hard satisfiability problem, 1995.
- [DD01] Olivier Dubois and Gilles Dequen. A backbone-search heuristic for efficient solving of hard 3-SAT formulae. In Bernhard Nebel, editor, *IJCAI*, pages 248–253. Morgan Kaufmann, 2001.
- [DD03] Gilles Dequen and Olivier Dubois. knfs: An efficient solver for random k -SAT formulae. In Enrico Giunchiglia and Armando Tacchella, editors, *SAT*, volume 2919 of *Lecture Notes in Computer Science*, pages 486–501. Springer, 2003.
- [DLL62] Martin Davis, George Logemann, and Donald Loveland. A machine program for theorem-proving. *Commun. ACM*, 5(7):394–397, 1962.
- [Fre95] Jon William Freeman. *Improvements to propositional satisfiability search algorithms*. PhD thesis, University of Pennsylvania, Philadelphia, PA, USA, 1995.
- [GW00] J. F. Groote and J. P. Warners. The propositional formula checker heerhugo. *Journal of Automated Reasoning*, 24:101–125, 2000.
- [HDvZvM04] Marijn J. H. Heule, Mark Dufour, Joris E. van Zwieten, and Hans van Maaren. March_eq: Implementing additional reasoning into an efficient look-ahead SAT solver. In Holger H. Hoos and David G. Mitchell, editors, *SAT (Selected Papers)*, volume 3542 of *Lecture Notes in Computer Science*, pages 345–359. Springer, 2004.
- [Her06] Paul Herwig. Decomposing satisfiability problems. Master’s thesis, TU Delft, 2006.
- [HvM06a] Marijn J. H. Heule and Hans van Maaren. March_dl: Adding adaptive heuristics and a new branching strategy. *Journal on Satisfiability, Boolean Modeling and Computation*, 2:47–59, 2006.

- [HvM06b] Marijn J. H. Heule and Hans van Maaren. Whose side are you on? finding solutions in a biased search-tree. Technical report, Proceedings of Guangzhou Symposium on Satisfiability In Logic-Based Modeling, 2006.
- [HvM07] Marijn J. H. Heule and Hans van Maaren. Effective incorporation of double look-ahead procedures. In Joao Marques-Silva and Karem A. Sakallah, editors, *Theory and Applications of Satisfiability Testing - SAT 2007*, volume 4501 of *Lecture Notes in Computer Science*, pages 258–271. Springer, 2007.
- [KL99] Oliver Kullmann and H. Luckhardt. *Algorithms for SAT/TAUT decision based on various measures*, 1999. Preprint, 71 pages, available on <http://cs.swan.ac.uk/~csoliver/Artikel/TAUT.ps>.
- [Kul99a] Oliver Kullmann. Investigating a general hierarchy of polynomially decidable classes of CNF’s based on short tree-like resolution proofs. Technical Report TR99-041, Electronic Colloquium on Computational Complexity (ECCC), October 1999.
- [Kul99b] Oliver Kullmann. New methods for 3-SAT decision and worst-case analysis. *Theoretical Computer Science*, 223(1-2):1–72, July 1999.
- [Kul99c] Oliver Kullmann. On a generalization of extended resolution. *Discrete Applied Mathematics*, 96-97(1-3):149–176, 1999.
- [Kul02] Oliver Kullmann. Investigating the behaviour of a SAT solver on random formulas. Technical Report CSR 23-2002, University of Wales Swansea, Computer Science Report Series (<http://www-compsci.swan.ac.uk/reports/2002.html>), October 2002. 119 pages.
- [Kul04] Oliver Kullmann. Upper and lower bounds on the complexity of generalised resolution and generalised constraint satisfaction problems. *Annals of Mathematics and Artificial Intelligence*, 40(3-4):303–352, March 2004.
- [LA97a] Chu Min Li and Anbulagan. Heuristics based on unit propagation for satisfiability problems. In *IJCAI (1)*, pages 366–371, 1997.
- [LA97b] Chu Min Li and Anbulagan. Look-ahead versus look-back for satisfiability problems. In Gert Smolka, editor, *CP*, volume 1330 of *Lecture Notes in Computer Science*, pages 341–355. Springer, 1997.
- [Li99] Chu Min Li. A constraint-based approach to narrow search trees for satisfiability. *Information processing letters*, 71(2):75–80, 1999.
- [Li00] Chu Min Li. Integrating equivalency reasoning into davis-putnam procedure. In *Proceedings of the Seventeenth National Conference on Artificial Intelligence and Twelfth Conference on Innovative Applications of Artificial Intelligence*, pages 291–296. AAAI Press / The MIT Press, 2000.
- [MZK⁺99] Rémi Monasson, Riccardo Zecchina, Scott Kirkpatrick, Bart Selman, and Lidror Troyansky. Determining computational complexity from characteristic ‘phase transitions’. *Nature*, 400:133–137, 1999.
- [PH02] S. Pilarski and G. Hu. Speeding up sat for eda. In *DATE ’02: Proceedings of the conference on Design, automation and test in*

Europe, page 1081, Washington, DC, USA, 2002. IEEE Computer Society.

- [SS98] Mary Sheeran and Gunnar Stålmarck. A tutorial on stålmarcks’s proof procedure for propositional logic. In *FMCAD ’98: Proceedings of the Second International Conference on Formal Methods in Computer-Aided Design*, pages 82–99, London, UK, 1998. Springer-Verlag.