2.9 Implementing CDCL

The performance of modern SAT solvers is sensitive to at least six components [14]: (i) preprocessing [27], (ii) conflict-driven clause learning (CDCL) [60, 42, 50, 65], (iii) heuristics for decision variable selection [48, 12], (iv) heuristics for restarting [38, 5, 13], (v) heuristics for learned clauses deletion (forgetting) [4, 39], and (vi) reduction during a CDCL run [40], typically called *inprocessing* in the SAT context. All these components are inter-connected, so if one of them is deactivated, or the interplay is not correctly adjusted then efficiency drops down.

In addition, for an efficient CDCL implementation the underlying data-structure and algorithms play a crucial part. There is a significant gap between the rule based CDCL calculus discussed in Section **??** and an actual implementation. In this section I explain some of the main techniques.

The most important data-structure of a CDCL implementation is the *2-watched literal* scheme, introduced in the next section. For choosing the next decision variables a suitable heuristic is important. One of the standard heuristics is called *VSIDS* (Variable State Independent Decaying Sum) end explained in Section 2.9.2. Furthermore, the decision for choosing the most reasonable clause to be learned after a discovered conflict is handled by the notion of *UIPs* (Unique Implication Points). Actually, the CDCL calculus of Section **??** has learning a UIP clause build in. Finally, a suitable Restart and Forget strategy, the deletion policy, is essential for an efficient long term behavior of the calculus, discussed in Section 2.9.4.

2.9.1 Lazy Data Structure: 2-Watched Literals (2WL)

For applying the rule Propagate, all literals in a clause except one need to be false with respect to the current trail. For applying the rule Conflict all literals need to be false with respect to the current trail. Thus as long as at least two literals of a clause are undefined with respect to the current trail or at least one literal is true with respect to the current trail the clause can be discarded by the CDCL calculus rules.

This results in the well-known *2-watched literals* idea where only two literals of a clause are indexed. Indexing is needed in general, in order to efficiently access relevant clauses. After a decision or propagation of some literal L all clauses containing $\operatorname{comp}(L)$ need to be visited in order to check for rule Propagate and Conflict. Simply traversing all clauses in $N \cup U$ is too inefficient. A complete index assigns to every literal the sequences of clauses containing the literal. The 2-watched literal index reduces this to indexing only the first two literals of a clause. The invariant for the 2-watched literals with respect to the current trail is:

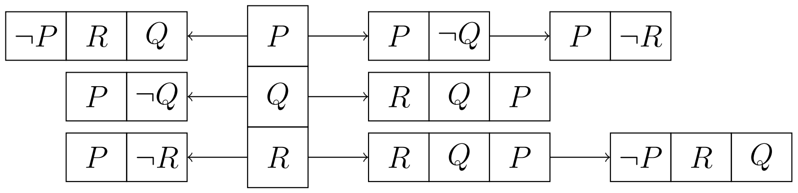
Invariant 2.9.1 (2-Watched Literal Indexing)**.** If one of the watched literals is false and the other watched literal is not true, then all other literals of the clause are false.

There are two major advantages of indexing a clause by 2WL: (i) when a literal of the clause that is not watched changes its truth value nothing needs to be done (ii) applying rule Backtrack does not result in any update on the 2WL data structure, because literals are only changed to status undefined.

For example, consider the clause set

$$N = \{P \lor \neg R,\ P \lor \neg Q,\ R \lor Q \lor P,\ \neg P \lor R \lor Q\}$$

resulting in the below 2WL index structure. Clauses are written as boxes suggesting an array implementation and the leftmost two literals are the 2WL. The entry point of the overall index is an array shown in the middle. It is indexed by propositional variables and has two entries for each variable: to the right a sequences of clauses where the positive literal of that variable is watched and to the left a sequences of clauses where the negative literal of that variable is watched. Every clause has 2-watched literals so it is linked twice in the 2WL index structure. Literals are represented by natural numbers or integers.


¬P | R | Q ← P → P | ¬Q → P | ¬R
P | ¬Q ← Q → R | Q | P
P | ¬R ← R → R | Q | P → ¬P | R | Q

Note that the clause $R \lor Q \lor P$ is not indexed at literal P and the clause $\neg P \lor R \lor Q$ is not indexed at literal Q.

I

Although the above figure suggests two copies of each clause in the 2WL index structure, of course, there is only one shared clause. Unit clauses are considered separately, outside the index structure, since they can be immediately used for reduction. In practice, two literal clauses are considered separately as well, because they employ more efficient update and check procedures. For simplicity, the examples treat unit and two literal clauses as any other clause. Clauses, sequences of clauses and the propositional variable index structure are all implemented via arrays.

Now consider a CDCL run deciding $\neg P$:

$$(\epsilon; N; \emptyset; 0; \top)$$
$$\Rightarrow^{\text{Decide}}_{\text{CDCL}} (\neg P^1; N; \emptyset; 1; \top)$$

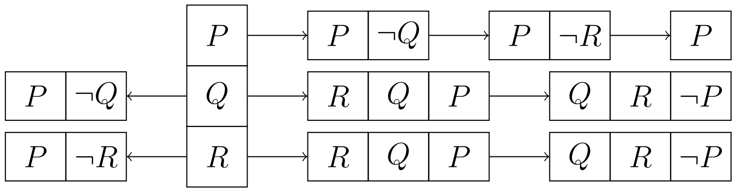
The algorithm jumps through variable P to its clause sequence on the right, because all clauses to the left of P are true. It finds the two literal clauses $P \lor \neg R$, $P \lor \neg Q$ that need not to be updated but result in the propagation of $\neg R$ and $\neg Q$. Updating means reestablishing the above 2WL Invariant 2.9.1 if it is violated. The algorithm propagates both $\neg R$ and $\neg Q$ and then runs the update procedure for both literals.

$\Rightarrow^{\text{Propagate}}_{\text{CDCL}}$ $(\neg P^1 \neg R^{P\vee\neg R}; N; \emptyset; 1; \top)$
$\Rightarrow^{\text{Propagate}}_{\text{CDCL}}$ $(\neg P^1 \neg R^{P\vee\neg R} \neg Q^{P\vee\neg Q}; N; \emptyset; 1; \top)$

The algorithm jumps through variable R to its clause sequence on the right. The first clause it finds is $R \vee Q \vee P$ which is already false with respect to the trail. So Conflict and Resolve and finally Backtrack is applied.

$\Rightarrow^{\text{Conflict}}_{\text{CDCL}}$ $(\neg P^1 \neg R^{P\vee\neg R} \neg Q^{P\vee\neg Q}; N; \emptyset; 1; R \vee Q \vee P)$
$\Rightarrow^{\text{Resolve}}_{\text{CDCL}}$ $(\neg P^1 \neg R^{P\vee\neg R}; N; \emptyset; 1; R \vee P)$
$\Rightarrow^{\text{Resolve}}_{\text{CDCL}}$ $(\neg P^1; N; \emptyset; 1; P)$
$\Rightarrow^{\text{Backtrack}}_{\text{CDCL}}$ $(P^P; N; \{P\}; 0; \top)$

The overall CDCL rule sequence does not result in any updates of the 2WL index data structure. It is always the case that executing the rules Skip, Resolve, or Backtrack does not need any update on the 2WL indexing, because the Invariant 2.9.1 is robust against removing literals from the trail. After backtracking and establishing P on the trail, the algorithm jumps through variable P to its clause sequence on the left. This time the clause $\neg P \vee R \vee Q$ violates the 2WL invariant since Q and R are undefined but $\neg P$ is false. The 2WL for this clause are changed by exchanging the literals $\neg P$ and Q and updating the 2WL index data structure.



Finally, two applications of Decide on Q and R result in a model for N without any further changes to the 2WL structure, because P is already true.

$\Rightarrow^{\text{Decide}}_{\text{CDCL}}$ $(P^P Q^1; N; \{P\}; 1; \top)$
$\Rightarrow^{\text{Decide}}_{\text{CDCL}}$ $(P^P Q^1 R^2; N; \{P\}; 2; \top)$

An alternative derivation to a model would be to first decide on $\neg Q$.

$\Rightarrow^{\text{Decide}}_{\text{CDCL}}$ $(P^P \neg Q^1; N; \{P\}; 1; \top)$
$\Rightarrow^{\text{Propagate}}_{\text{CDCL}}$ $(P^P \neg Q^1 R^{Q\vee R\vee\neg P}; N; \{P\}; 1; \top)$

Then R is propagated by the clause $Q \vee R \vee \neg P$ but before the clause 2WL index is changed by flipping the second and third literal in the clause $R \vee Q \vee P$ and moving it to the index P in the right sequence. Eventually, also this sequence results in a model for N.

In general, the update procedure considers the following cases, assuming a literal L is set to true by Propagate or Decide. All clauses where L is watched can be ignored, they are true anyway. So the algorithm traverses all clauses where comp(L) is watched. If for some clause the other watched literal is true, nothing needs to be done. If the other literal is false or undefined, then the clause is searched for an undefined or true literal beyond the watched literals. If it exists, the undefined/true literal and comp(L) are exchanged and the clause is removed from the comp(L) sequence and moved to the sequence of the other literal. If it does not exist, then if the second watched literal is undefined, the clause propagates that literal. If the second watched literal is false as well, then a conflict clause is found.

During the update procedure for the 2WL structure often several literals are found propagating and their respective sequences need to be considered sequentially. Still it is very useful to update their truth value already at the time they are detected.

One advantage of the 2WL data structure is that during the loop Propagate, Decide, Conflict only clauses where an involved literal is watched need to be considered. The other is that the rule Backtrack does not result in any update for the literals whose truth value is turned to undefined.

Theorem 2.9.2 (2WL and Backtrack)**.** Consider a reasonable CDCL run on a 2WL data structure satisfying the 2WL Invariant 2.9.1 before any application of rule Decide and a Backtrack application:
$(M_1K^{i+1}M_2; N; U; k; D \lor L) \Rightarrow_{\text{Backtrack}} (M_1L^{D\lor L}; N; U \cup \{D \lor L\}; i; \top)$.
Then the 2WL invariant is preserved except for potentially L.

Proof. First, note that $D \lor L$ is propagating L by construction, so the comp(L) sequence needs to be updated. Now assume that there is a clause $K_1 \lor K_2 \lor C$ with K_1, K_2 watched that violates the 2WL invariant and both $K_i \neq \text{comp}(L)$. This means without loss of generality that K_1 is false in M_1L and K_2 is not true and there are literals in C which are not false with respect to M_1L. Note that both $K_i \neq L$ because for otherwise one of them is true. So the truth value of both K_i is determined by M_1. The above described update procedure only exchanges false literals with true or undefined literals. The rule Backtrack, except for L, only changes truth values of literals to undefined. Thus already before K^{i+1} was decided, the clause $K_1 \lor K_2 \lor C$ did violate the 2WL invariant, contradicting the assumption. □

2.9.2 Dynamic Decision Heuristic: VSIDS

The actual literal selection by the rule Decide greatly influences the number of transition steps until CDCL derives a finite state. For example, the proof of Proposition 2.8.7 presents a nice decision heuristic for some satisfiable clause set, provided the actual model is known in advance.

In general, the idea of the decision heuristic is to focus the search. Thinking of a set of clauses as a graph, where the clauses are the nodes and edges are drawn between complementary literals, the search should focus on a single connected component of the graph. Inside this component the search should focus on a small highly connected part that enables conflicts on the basis of a few decisions or has a model. For a simple example, consider the clause set $N = N_1 \uplus N_2 \uplus \{P \vee Q\} \uplus N_3$, where N_1 does not share any propositional variable with N_2, N_3, the atom P does not occur in N_3 and the atom Q does not occur in N_2. Then an ideal decision heuristic will first focus on the atoms in N_1 before considering any atom from the rest of the clause set, or the other way round. Furthermore, atoms in N_3 should not be decided until all atoms in N_2 have been decided, or the other way round. Unfortunately, deriving the necessary graph properties for this behavior is, in general, as hard as the SAT problem. The way out are heuristics that can be efficiently computed and dynamically adapt towards the above described behavior. The *VSIDS* (Variable State Independent Decaying Sum) heuristic is such a heuristic [48, 33].

The VSIDS heuristic initially assigns each propositional variable a number, its initial *score*, e.g., its number of occurrences in the clause set, or all 0 or 1 or some random value. This score is updated during a CDCL run. The rule Decide always picks a variable with maximal score. In case the picked variable was not decided before it heuristically chooses it positively or negatively, but then stores the sign for later use. After a Backtrack or Restart application, if the variable is selected again by Decide, the stored sign us used. This is called *phase saving.* If the variable was already decided, the stored sign is selected. Now either the CDCL run produces a model or a conflict and afterwards Skip and Resolve become applicable. With each application of Resolve, the score of the resolved literal is incremented by a bonus $b > 0$. Depending on the initial score, the bonus b starts with some positive value, e.g., maximum of half the initial maximal score and it is updated as well during search. When Backtrack is applied, the scores of the remaining atoms of the learned clause are also incremented by b and b itself is updated by multiplication with a fixed constant $c > 1$, e.g., $c = \frac{6}{5}$ and for this value of c the new value for b after Backtrack is $\lceil \frac{6}{5} b \rceil$. If the score is implemented by doubles then the ceiling function application is not needed. In addition, every k^{th} application of Decide a random variable is chosen, where a typical value for k is 200.

Of course, b grows exponentially in the number of Backtrack applications. Hence, the score function as well. So eventually there will be an overflow of the respective unsigned data structures. The bonus b only depends on the number of conflicts, c, and some initial value known at the start of the CDCL procedure. Furthermore, the bit-length of the used unsigned data structure is known as well. So the minimal number of Conflicts generating the overflow can be computed once in advance. A typical value for today's computer architectures and choices for c and some initial value is above 500. So every 500 conflicts the implementation has to take care of an potential overflow by rescaling b and the score function. This can be done by right-shifting the scores of all variables and b itself. Computationally, this is a neglectable overhead, so the VSIDS heuristic can be efficiently computed. The score function is often implemented as a priority queue. The Sat solver MiniSat [28], implements VSIDS via doubles. The default for the start score value is 0, the initial bonus value is $b = 2$. The bonus increase is $c = \frac{100}{95}$. It does not precompute a bound for rescaling, instead whenever an actual increase operation exceeds a value of 10^{100} all scores and b are scaled by 10^{-100}.

The choice of a random variable every k^{th} application of Decide is a backdoor for the case that the score function accidentally focuses on a too large complicated component of the clause graph, while there exist very small beneficial components. Think of the above example, where N_1 contains only a few clauses compared to the rest and is unsatisfiable, but the VSIDS heuristic starts with atoms from N_2, because they occur by far more frequently. So the random choice part of the heuristic improves its robustness.

2.9.3 Conflict Analysis, Learning, and 1-UIPs

The SAT literature [14] discusses the relationship between an appropriate analysis of the conflict, the eventually learned clause and so called first unit implication points (1-UIPs). This section clarifies these notions in relation to the CDCL calculus of Section **??**.

Example 2.9.3. Consider the clause set $N = \{P_1 \lor P_2 \lor \neg Q_1,\ P_2 \lor \neg Q_2,\ Q_1 \lor Q_2 \lor Q_3,\ \neg Q_3 \lor P_3 \lor \neg Q_5,\ \neg Q_3 \lor \neg Q_4,\ Q_4 \lor Q_5\}$ and a CDCL run resulting in a first conflict:

$$(\epsilon; N; \emptyset; 0; \top)$$
$$\Rightarrow^*_{\text{CDCL}} (\neg P_3^1 \neg P_1^2 \neg P_2^3 \neg Q_2^{P_2 \lor \neg Q_2} \neg Q_1^{P_1 \lor P_2 \lor \neg Q_1} Q_3^{Q_1 \lor Q_2 \lor Q_3} \neg Q_4^{\neg Q_3 \lor \neg Q_4} \neg Q_5^{\neg Q_3 \lor P_3 \lor \neg Q_5}; N; \emptyset; 3; Q_4 \lor Q_5)$$

The information of the dependencies between decision literals and propagated literals can now be represented by a directed, acyclic graph, called the *implication graph* presented in Figure 2.14. The decision literals $\neg P_3^1 \neg P_1^2 \neg P_2^3$ are the sources of the implication graph, the eventual false clause $Q_4 \lor Q_5$ the sink.

The CDCL calculus, Section **??**, derives out of the above state in two steps the learned clause $P_3 \lor \neg Q_3$ and backtracks:

$$\Rightarrow^{\text{Resolve}}_{\text{CDCL}} (\neg P_3^1 \neg P_1^2 \neg P_2^3 \neg Q_2^{P_2 \lor \neg Q_2} \neg Q_1^{P_1 \lor P_2 \lor \neg Q_1} Q_3^{Q_1 \lor Q_2 \lor Q_3} \neg Q_4^{\neg Q_3 \lor \neg Q_4}; N; \emptyset; 3; \neg Q_3 \lor P_3 \lor Q_4)$$
$$\Rightarrow^{\text{Resolve}}_{\text{CDCL}} (\neg P_3^1 \neg P_1^2 \neg P_2^3 \neg Q_2^{P_2 \lor \neg Q_2} \neg Q_1^{P_1 \lor P_2 \lor \neg Q_1} Q_3^{Q_1 \lor Q_2 \lor Q_3}; N; \emptyset; 3; \neg Q_3 \lor P_3)$$
$$\Rightarrow^{\text{Backtrack}}_{\text{CDCL}} (\neg P_3^1 \neg Q_3^{Q_3 \lor P_3}; N; \{Q_3 \lor P_3\}; 1; \top)$$

Stopping the resolution process at clause $Q_3 \lor P_3$ is forced by the Backtrack rule and hence my version of the CDCL calculus. But this is not the only choice of generating a clause for backtracking. For example, further resolution steps can be performed:

$$\Rightarrow^{\text{Resolve}}_{\text{CDCL}} (\neg P_3^1 \neg P_1^2 \neg P_2^3 \neg Q_2^{P_2 \lor \neg Q_2} \neg Q_1^{P_1 \lor P_2 \lor \neg Q_1} Q_3^{Q_1 \lor Q_2 \lor Q_3}; N; \emptyset; 3; \neg Q_3 \lor P_3)$$
$$\Rightarrow (\neg P_3^1 \neg P_1^2 \neg P_2^3 \neg Q_2^{P_2 \lor \neg Q_2} \neg Q_1^{P_1 \lor P_2 \lor \neg Q_1}; N; \emptyset; 3; Q_1 \lor Q_2 \lor P_3)$$
$$\Rightarrow (\neg P_3^1 \neg P_1^2 \neg P_2^3 \neg Q_2^{P_2 \lor \neg Q_2}; N; \emptyset; 3; P_1 \lor P_2 \lor Q_2 \lor P_3)$$
$$\Rightarrow (\neg P_3^1 \neg P_1^2 \neg P_2^3; N; \emptyset; 3; P_1 \lor P_2 \lor P_3)$$

and then a backtrack step

$\Rightarrow \qquad (\neg P_3^1 \neg P_1^2 P_2^{P_1 \lor P_2 \lor P_3}; N; \{P_1 \lor P_2 \lor P_3\}; 2; \top)$

The first state can be considered superior to the second state for two reasons. Firstly, the learned clause $Q_3 \lor P_3$ of the first state has less literals than the learned clause $P_1 \lor P_2 \lor P_3$ of the second state and hence the first state has a higher probability to eventually yield a shorter terminating CDCL derivation. Recall that CDCL with a reasonable strategy never learns a clause subsumed by an already existing clause, Corollary **??**. The clause $Q_3 \lor P_3$ subsumes $\mathrm{O}(n)$-times more clauses than the clause $P_1 \lor P_2 \lor P_3$ in the set of all non-tautologous clauses, where n is the number of propositional variables in N. Secondly, the backtrack level of the first state is smaller than the backtrack level of the second state.

From the implication graph it can be seen that node Q_3 dominates the nodes $\neg P_1$, $\neg P_2$: every path from these nodes to the sink traverses Q_3. This is a graph-based argument that replacing P_1, P_2 in $P_1 \lor P_2 \lor P_3$ with Q_3 results in a consequence of the clause set. The node Q_3 is called a *unique implication point (UIP)* and the condition of the Backtrack rule ensures that the CDCL calculus of Section **??** always learns a clause with respect to a first UIP (1-UIP) searching from the sink towards the sources.

2.9.4 Restart and Forget

So far, I have not discussed much the CDCL rules Restart and Forget. In contrast, many properties of the CDCL calculus, Section **??**, do not hold if these rules are applied in an arbitrary way. Still, they are very important for the efficiency of a SAT solver.

The rule Forget removes clauses from the set of learned clauses U. The more clauses are contained in $N \cup U$ the more time is needed for the algorithms for propagation and conflict detection. The potential number of learned clauses is exponential in the number of propositional variables of a problem. Furthermore, as soon as the size of the computer memory that is actually needed to perform propagation and conflict detection exceeds cache structures or even main memory of a computer, the SAT solver slows down to such an extend that it often becomes practically useless. In addition, although Theorem 2.10.4 shows that learned clauses are always non-redundant, this does not exclude that a learned clause causes previously learned clauses or even input clauses to be redundant. For example, looking at a sequence of learned clauses $C_1, C_2, \ldots$ of a "typical" CDCL run, learned clause C_i subsumes its predecessor C_{i-1} with a probability of about 10%.

A reasonable CDCL run without Restart, where every learned clause is immediately forgotten via Forget after Backtrack also results in a sound and complete algorithm. It violates the invariant that the clauses propagating literals are contained in $N \cup U$, but still all rules work as desired. The rule Backtrack never deletes propagated literals left from the leftmost decision literal, so the CDCL run terminates anyway. Typically, such a run degrades to a DPLL run refined with dependency directed backtracking, called *backjumping*. Learned clauses are not needed for completeness or termination, just for efficiency.

The typical way Forget is invoked in implementations is in combination with an application of Restart. For each of the learned clauses an activity score is computed, incorporating the activity score of its literals as used by the VSIDS heuristics, see Section 2.9.2. The second established criteria is the number of decision levels a learned clause depends on for propagation. Then learned clauses are sorted with respect to the two criteria and clauses with a low score are forgotten. It may be that a forgotten clause is the justification for some propagated literal on the trail. A restart clears this potential problem.

Explaining the usefulness of the rule Restart is more difficult. It is in depth not well understood. Engineering the restart criterion is still an active are of research [5, 13]. Firstly, looking at a CDCL run without restarts, after a reasonable number of conflicts, the n decision literals on the trail will not be the maximal n literals with respect to the VSIDS heuristic. For example, assume the run starts by a first decision P^1 but the atom P is afterwards not involved in any conflict, i.e., it is not the atom of a literal resolved upon and not contained in any learned clause. Still the fact that P is true may cause unneeded propagations. It may even result in a behavior where the solver does not concentrate on “one part of the problem” but on two or more, causing a worst case exponential number of useless propagations. The rule Restart is a means to get rid of such “confusing” literals on the trail that meanwhile have a low score with respect to the dynamically evolving VSIDS heuristic. Secondly, a restart changes the status of literals on the trail. If the VSIDS variable selection heuristic did not change much after the previous restart, a restart will eventually produce a similar trail. However, the role of some literals on the trail will change from a decision literal to a propagated literal and vice versa. Learned clauses typically contain decision literals. This means, even if a trail after a restart is almost identical to the previous one, the afterwards learned clauses may look quite different because of the changed role of literals on the trail.

$\boxed{\mathtt{I}}$

MiniSat [28] combines Restart and Forget as follows: there is always a limit c for the number of learned clauses. If c clauses are learned, then they are sorted with respect to an activity score. The $\frac{c}{2}$ clauses with lowest score are thrown away, c is increased by a constant and a Restart is performed. Recall that performing a restart is needed to clear the trail. The VSIDS heuristic together with phase saving directs the search towards the same state that was generated before the restart.

2.9.5 The Overall Algorithm and Further Heuristics & Strategies

Algorithm 5 presents a CDCL solver including most aspects discussed in previous sections. It implements a reasonable strategy and includes the incorporation of the VSIDS heuristic and restarts. It does not contain a heuristic for an initial VSIDS score. Typical solutions are to start with a score of 0 for all variables or to start with the number of variable occurrences in N. Similarly for an application of the rule Decide. For a variable with maximal VSIDS score either the positive or the negative literal can be decided. Again this can be implemented via a heuristic on the number of literal occurrences in N. Important is *phase saving*: once a literal has been decided, after removal from the trail due to Restart or Backtrack, if it is decided again, it is decided with the same sign.

The restart heuristic typically considers also unit clauses. Once a unit clause is learned a restart is performed immediately. Unit clauses always propagate, so their literals are collected during a run at the start of the trail. This applies as well to literals propagating solely because of unit clauses, i.e., at level 0.

Algorithm 5: CDCL(S)

```
Input  : An initial state ($\epsilon; N; \emptyset; 0; \top$).
Output: A final state $S = (M; N; U; k; \top)$ or $S = (M; N; U; k; \bot)$
1  while (any rule applicable) do
2      ifrule (Conflict($S$)) then
3          while (Skip($S$) || Resolve($S$)) do
4              update VSIDS scores on resolved literals;
5          update VSIDS scores on learned clause;
6          Backtrack($S$);
7          if (potential VSIDS score overflow) then
8              scale VSIDS scores;
9          if (forget heuristic) then
10             Forget($S$) clauses ;
11             Restart($S$);
12         else
13             if (restart heuristic) then
14                 Restart($S$);
15     else
16         ifrule (!Propagate($S$)) then
17             Decide($S$) literal with max. VSIDS score;
18 return($S$);
```