Swansea University
Prifysgol Abertawe

SWANSEA UNIVERSITY

REPORT SERIES

Present and future of practical SAT solving
by

Oliver Kullmann

Report # CSR 8-2008

Present and future of practical SAT solving

Oliver Kullmann*

Computer Science Department
Swansea University
Swansea, SA2 8PP, UK

email: O.Kullmann@Swansea.ac.uk

http://cs.swan.ac.uk/~csoliver

April 19, 2008

Abstract

We review current SAT solving, concentrating on the two paradigms of *conflict-driven* and *look-ahead* solvers, and with a view towards the unification of these two paradigms. A general "modern" scheme for DPLL algorithms is presented, which allows natural representations for "modern" solvers of these two types.

1 Introduction

The number as well as the breadth of applications of SAT solving, like verification of hardware and software or solving difficult concrete combinatorial problem instances, has steadily increased over the last 10 years. Two main paradigms for backtracking solvers have emerged, the "conflict-driven solver" and the "look-ahead solver", the former better suited for verification problems, the latter better for difficult problems. Both paradigms now have reached some form of plateau, and the purpose of this article is to present these two different plateaus (kind of "fixed points"), and to discuss several ideas towards possible combinations of these two approaches, to overcome the current relative stagnation (regarding the core algorithms). We focus on "practical" algorithms which "work" (at least reasonably often), as represented by the SAT conference and the SAT competition.[1)]

Two basic approaches for SAT solving in general can be identified: Local search (for satisfying assignments), and backtracking. Local search has still a stronghold for satisfiable random formulas, and in recent years the theoretically very interesting "survey propagation" algorithm was developed, but outside the domain of (satisfiable) random formulas local search lost influence, and in this article we will not consider it (unquestionably there is a lot of potential, but it needs further development). Steady improvements of look-ahead backtracking solvers are noticeable, but the strongest development took place w.r.t. conflict-driven backtracking solvers, and accordingly we will put emphasise in this article on the notion of "clause learning".

*Partially supported by EPSRC Grant GR/S58393/01

[1)] This report will appear as [18].

Motivated by the success of SAT, extensions of SAT (like pseudo-boolean formulas, quantified boolean formulas or "SAT modulo theory") become increasingly popular, but yet no clear pattern emerged here. For the whole area there are many beliefs, many observations, but no proofs; nevertheless it seems that the subjects of this article are mature enough that a more systematic treatment might be possible.

Yet, in SAT no "theoretical idea" had impact on the "practice" of SAT solving: Although there have been many attempts, they never went far enough, and we do not understand the practical applications. I believe

- practice needs a *dedicated* effort, much more details and care in some areas, and more looseness in other areas,
- but there is much more to discover than the current "trivial" solvers!

So in this article I want somehow to present the "practical world" — hopefully we can learn from their observations and ideas. The observable stagnation regarding the core algorithms can be overcome in my opinion by unifying yet separate development lines:

1. The three main paradigms for SAT solving, "conflict-driven", "look-ahead" and "local search" should be combined (in a new, intelligent way).
2. SAT (with its focus on global structure) and CSP (with its focus on local structure) need to be unified. Here with "global structure" I allude at the fact that SAT problems in CNF representation come "chopped into little pieces", and the solution process considers statistical properties arising from pieces potentially belonging to very different parts of the problem instance. In contrast, traditionally the field of constraint satisfaction studies extensively "intelligent" problem representations, but at the cost of more global (and less predictable) structures.

In this article we focus on *look-ahead versus conflict-driven* (both are resolution-based, and thus closer together), trying to bring out the (quite different) underlying ideas, and to discuss how potentially those two approaches could be brought together (and what (considerable) problems have to be overcome for such a unification).[2] An outline of this article follows:

1. Conjunctive normal forms seem to be at the heart of "SAT", and they are discussed in Section 2.
2. An overview on polynomial time methods which seem relatively close to "practice" is given in Section 3.
3. The gist of this article is given by the new general scheme $\mathfrak{G}$ for DPLL-algorithm presented in Subsection 4.1, unifying the look-ahead and the conflict-driven paradigms, which is then specialised to yield a general look-ahead scheme `la` in Subsection 5.1, and a general conflict-driven scheme `cd` in Subsection 6.1.3.

[2] The new `OKsolver`, tentatively called "OKsolver2009" and developed in the framework of the `OKlibrary` (`http://www.ok-sat-library.org`), an open-source library for generalised SAT solving (embracing CSP), aims at unifying all three paradigms.

4. DPLL in general is discussed in Section 4, while look-ahead solvers are presented in Section 5, and the main features of conflict-driven solvers are the subject of Section 6.

5. Approaches for understanding and extending clause-learning are outlined in Section 7.

6. Finally some conclusions are drawn in Section 8.

Now before going into more details, I will try to "outline in a nutshell" fundamental concepts and ideas. The basic notion for a backtracking solver (of any kind) is that of a "partial assignment" φ, fixing some variables to values determined previously (according to the current path in the search tree from the root to the current node), while leaving other variables open, either to be decided later, or to be fixed by reasoning, or left open since they do not play a role. Since it as at the core of (current) SAT solving, in this article we concentrate on problems represented by conjunctive normal forms, or, more combinatorially, represented by "clause-sets", where each clause C can be seen as a negated partial assignment φ, a constraint forbidding all (total) assignments which extend φ. We will represent this via $C = C_\varphi$ in Subsection 2.2.2. Furthermore we only consider backtracking approaches, due to its current dominance for practical applications.

The two basic paradigms for backtracking SAT solvers (also "DPLL solvers") are "look-ahead" and "conflict-driven". The look-ahead paradigm, based on stronger polynomial-time reductions and stronger heuristics, is similar to CSP solvers using appropriate constraint propagators, only that here now the emphasise is on the use of partial assignments. Look-ahead solvers are easily parallelisable, and thus might become more important again in the future, at this time however conflict-driven solvers are in the foreground. Again, the basic approach is known from constraint programming, based on "clause-learning", which is no-good learning of clauses C_φ for the current path φ, but using several mechanisms to strengthen the learning effort (this is easier in this setting, since the problem representation (via CNF) is just what is need to represent the no-goods).

Conflict-driven solvers are derived from DPLL-solvers, however it seems appropriate to describe their behaviour no longer in the usual tree-based (recursive) fashion, but as a simple iterative approach (more similar to dynamic programming, as put by David Mitchell). The basic idea is for a problem instance P to guess a satisfying partial assignment φ, guided by only the most basic look-ahead, namely unit-clause propagation, and once a contradiction was realised, then this "conflict" is analysed for its "real causes", and added via clause-learning to the clause database. The whole solution process then might completely re-start from scratch again (but using the modified problem instance P', including now the learned clauses), though in practice only a part of the current path is undone, just to the point where the freshly learned information has obvious consequences.

SAT solving based on DPLL-approaches is close to the resolution calculus, and from the early beginnings by Martin Davis and Hilary Putnam these connections have been exploited. Resolution is just the logical consequence relation restricted to the clause language and only two premises — it is easy to see that then the conclusion is either trivial or the (unique) "resolvent" of the two parent clauses. Now a "resolution-based" solver (as common in ATP at the level of first-order logic) starts with the premises, and tries to derive the empty clause from it, while a DPLL-solver in effect reverts this process: The goal becomes the input clause-set

F (the root of the tree), for which backtracking seeks to find the premises from which F can be shown unsatisfiable. Conflict-driven solvers can be understood as breaking the tree-like structure, and so in a sense they combine the resolution-based approaches with the backtracking approaches. A classic references for the resolution calculus is [45], while a recent overview on proof systems is [41]. The earliest paper on the connection of backtracking (in the form of decision trees) and resolution is [35], while a fuller account, also applicable to more general problem representations, can be found in [26].

2 Conjunctive normal forms

One peculiar aspect of "SAT solving" is the focus on a very specific form of constraint satisfaction problems:

- only boolean variables are considered;
- only (disjunctive) clauses are allowed as constraints (but involving arbitrarily many variables).

In other words, only boolean CNF's are considered for core SAT solving, and recent extensions build on top of this. Due to the tight coupling of this representation with the algorithms and data structures, the role of (boolean) CNF's goes far beyond sheer problem representation, and seems actually to a certain degree essential for the success of SAT solving for applications. In this section we will discuss the various aspects of "boolean CNF's" and their role for SAT solving.

In this article we put the emphasise on boolean clause-sets, since they (seem) to embody the "secret" of SAT, but it is natural to ask to what extend the special properties of boolean clause-sets can be generalised to CSP's (with unrestricted constraint scopes). A natural settings is given by "signed CNF", which allow for non-boolean variables v with domains D_v, and where literals are of the form "$v \notin S$" for some "sign" $S \subseteq D_v$; this is the most general form of clause-sets (as CNFs) where literals contain only a single variable, and many logical properties can be generalised (for a recent entry point to the literature see [1]). However for the more combinatorial properties of boolean clause-sets, signed clause-sets seem a vast generalisation, and the notion of "clause-sets with non-boolean variables" should better be reserved in my opinion to clauses containing only literals of the form "$v \neq \varepsilon$" for $\varepsilon \in D_v$; such "sets of no-goods" are thoroughly studied in [27].

2.1 Clause-sets and partial assignments

The theoretical foundations of SAT are best framed as follows:

- We have **variables** with boolean domain $\{0, 1\}$; the set of all variables is $\mathcal{VA}$.
- From variables we build **positive literals** and **negative literals**, stating that the variables must become true resp. false. Identifying positive literals with variables, and denoting negative literals with underlying variable v by "$\overline{v}$", we obtain the set $\mathcal{LIT} = \mathcal{VA} \cup \overline{\mathcal{VA}}$ of all literals, where now complementation becomes an involution (a self-inverse bijection) of $\mathcal{LIT}$ onto itself. The underlying variable of literal x is denoted by $\mathrm{var}(x)$.

- Two literals x, y **clash** if they have the same underlying variable, but different "polarities" (or "signs"), that is, iff $x = \overline{y}$.
- **Clauses** are finite and clash-free sets of literals, understood as disjunctions; the set of all clauses is denoted by $\mathcal{CL}$. A special clause is the empty clause $\bot := \emptyset \in \mathcal{CL}$, and $\mathrm{var}(C) := \{\mathrm{var}(x) : x \in C\}$ for $C \in \mathcal{CL}$.
- **Clause-sets** are finite sets of clauses, understood as conjunctions; the set of all clause-sets is denoted by $\mathcal{CLS}$. A special clause-set is the empty clause-set $\top := \emptyset \in \mathcal{CLS}$, and $\mathrm{var}(F) := \bigcup_{C \in F} \mathrm{var}(C)$ for $F \in \mathcal{CLS}$.
- A **partial assignment** is a map $\varphi : V \to \{0, 1\}$ for some finite set $V \subseteq \mathcal{VA}$ of variables, the set of all partial assignments is $\mathcal{PASS}$, and we use $\mathrm{var}(\varphi) := V$. $\varphi(x)$ for literals x is defined (in the obvious way) if $\mathrm{var}(x) \in \mathrm{var}(\varphi)$; a term $\langle x_1 \to \varepsilon_1, \ldots, x_m \to \varepsilon_m \rangle$ for literals x_i (with different underlying variables) and $\varepsilon_i \in \{0, 1\}$ denotes the partial assignment φ with $\mathrm{var}(\varphi) = \mathrm{var}(\{x_1, \ldots, x_m\})$ and $\varphi(x_i) = \varepsilon_i$.
- The most important operation for SAT is the operation

  $$* : \mathcal{PASS} \times \mathcal{CLS} \to \mathcal{CLS}$$

  of partial assignments on clause-sets, called "application", where $\varphi * F$ for $\varphi \in \mathcal{PASS}$ and $F \in \mathcal{CL}$ is obtained from F by removing all satisfied clauses (those $C \in F$ containing $x \in C$ with $\varphi(x) = 1$), and removing all falsified literals (i.e., literals x with $\varphi(x) = 0$) from the remaining clauses.
- Finally the set $\mathcal{SAT}$ of **satisfiable clause-sets** is defined as the set of $F \in \mathcal{CLS}$ such that there exists $\varphi \in \mathcal{PASS}$ with $\varphi * F = \top$, while $\mathcal{USAT} := \mathcal{CLS} \setminus \mathcal{SAT}$ denotes the set of **unsatisfiable clause-sets**.

Several special properties of this setting need to be pointed out:

1. A fundamental fact, justifying the use of *partial* assignments, is that if a partial assignment φ satisfies a clause-set F then every **extension** of φ also satisfies F (this is just guaranteed by the process of applying partial assignments), where an "extension" of φ is just a partial assignment ψ with $\varphi \subseteq \psi$ (using the definition of partial assignments as maps, that is, as sets of ordered pairs).
2. Every clause is falsifiable, and thus the property that a partial assignment φ satisfies a clause-set F, i.e., $\varphi * F = \top$, is equivalent to the property that every extension $\psi \supseteq \varphi$ of φ can be further extended to satisfy F. But that φ **falsifies** F, i.e., $\bot \in \varphi * F$, says much less (it's trivial to falsify, but hard to satisfy) and is in general only the final visible expression (during the search process) of unsatisfiability, for example F might be unsatisfiable right away. To obtain more symmetric conditions, one could say that φ "allows satisfaction" of F if $\varphi * F$ is satisfiable, that is, there exists an extension of φ which satisfies F, while φ "disallows satisfaction" of F if $\varphi * F$ is unsatisfiable. Then one could characterise look-ahead solvers as solvers which try to give good indications that the current partial assignment (under construction) allows satisfaction, while conflict-driven solvers could be characterised as solvers trying to find better and better reasons that the current partial assignment disallows satisfaction.

3. If a variable v is set to a value ε, then we can apply the partial assignment $\langle v \to \varepsilon \rangle$ to a clause-set F and obtain a new clause-set $\langle v \to \varepsilon \rangle * F$ which does not contain the variable anymore. This allows us to replace iterated application of partial assignments as in $\psi * (\varphi * F)$ by a single application of the **composition** $\psi \circ \varphi$ of both partial assignments via $\psi * (\varphi * F) = (\psi \circ \varphi) * F$, where the construction of the composition is obvious for variables v where ψ, φ do not clash, while in case of a clash only φ is relevant, since after application of φ the variable v has been eliminated and the value of ψ on v doesn't matter.

Theoretically clause-*sets* are a convenient framework — are they also used in practice ? Let us consider the corresponding conditions:

- Associativity of disjunction and conjunction is always implemented (typically by using lists to represent clauses and clause-sets).
- Commutativity:
  - Commutativity of disjunction (in a clause) may be implemented by ordering the literals in a clause. Often this is not done, but it seems not very relevant.
  - Commutativity of conjunction is never implemented, and especially for industrial benchmarks the order of clauses is quite important (successful conflict-driven solvers employ a lot of "dirty tricks").
- Input-clauses containing clashes are removed, and are also never introduced.
- Idempotency:
  - Literals are not repeated in clauses, achieved by preprocessing the input, while maintenance is basically trivial (for the type of solvers considered, where resolution operations are restricted to the preprocessing phase).
  - However repeated clauses are only removed during pre-processing (if at all, and then applying the stronger reduction of subsumption-elimination (see Subsection 3.4.1)), while they may be created during solving.

To summarise:

1. "Clauses in practice" can be adequately understood as we defined them.
2. "Clause-sets in practice" are actually lists of clauses (order is important, and the effort of removing duplicated clauses is too high).

The notion of literals, clause and clause-sets install several normalisation conditions when regarding literals, clauses and clause-sets as boolean functions (or constraints):

- Literals are never constant true or constant false.
- A clause is never constant true, while $\bot$ is the unique clause which is constant false.
- The unique clause-set which is constant true is $\top$.
- Exactly the unsatisfiable clause-sets are constant false. The clause-sets which are falsified by every partial assignment are those containing $\bot$, which is reduced to the unique form $\{\bot\}$ by reduction r_0 (see Subsection 3.1).

2.2 Properties

Partial assignments and boolean clause-sets have many special properties, compared to the situation for constraint satisfaction, and in this subsection the properties which seem most outstanding for practical SAT solving are discussed:

1. If we have a unit-clause $\{x\} \in F$, then the assignment $\langle x \to 1 \rangle$ is enforced, and this process of "unit-clause elimination" is considered in Subsection 2.2.1 (while unit-clause propagation is the subject of Subsection 3.1.1).
2. The close relation between clauses and partial assignment, the basis for clause-learning, is considered in Subsection 2.2.2.
3. The input-problem doesn't need to be given in a special form, but we can apply the logic of clause-sets and partial assignments under very general conditions which are discussed in Subsection 2.2.3.

2.2.1 Unit-clause elimination

Arguably the most important aspect of clauses for SAT solving is:

Once all literals are falsified except of one, then
the remaining variable gets an enforced value.

This is based on three properties of clauses and literals:

1. falsification of clauses only by giving every variable the wrong value;
2. easy satisfaction of a clause by giving just one variable not the wrong value;
3. since there are only two values, there is no choice for a right value.

The first two properties are still maintained by generalised clauses, allowing non-boolean variables v with domain D_v and literals "$v \neq \varepsilon$" for some $\varepsilon \in D_v$, but the third property requires boolean variables, and thus the strong form of unit-clause elimination is characteristic for *boolean* clause-sets. Repeated elimination of unit-clauses is called "unit-clause propagation", and this most basic process for SAT solving is considered in more details in Subsection 3.1.1).

2.2.2 Correspondence between clauses and partial assignments

At least second in importance to unit-clause elimination is the 1-1 correspondence between clauses and partial assignments:

For every partial assignment φ there is exactly one clause C_φ, such that the falsifying assignments for C_φ are exactly the extensions of φ.
And conversely, for every clause C there is exactly one partial assignment φ_C such that the clauses falsified by φ_C are exactly the sub-clauses of C.

Obviously C_φ consists exactly of the literals falsified by φ, while φ_C sets exactly the literals in C to false, and these two formations are inverse to each other:

1. This correspondence establishes the close relation between the **search trees** of backtracking algorithms and **resolution refutations**, as further explained in Subsection 7.1.

2. Clauses C_φ for falsifying assignments φ are also called "no-goods", and are the essence of "learning" as explored in Subsection 6.1.

2.2.3 An axiomatic approach

A generalisation of "satisfiability" by allowing arbitrary problem representation, on which partial assignments "operate", has been introduced and studied in [26]. The key observation here is, when considering only partial *assignments*, where assigned variables get a unique value[3)], then a sequence $\psi * (\varphi * F)$ of applications of partial assignments can be elegantly represented by a unique application of the composition $(\psi \circ \varphi) * F$, and the essence of basic satisfiability considerations can be captured by a simple algebraic framework, where problem instances are left unspecified, and only via the application of partial assignments can we "query" them. Clauses and resolution then appear as a meta-structure on top of the given domain of problem instances. In Subsection 7.3 we will use this theory, which allows generalised resolution "modulo oracles", for a "compressed" form of learning.

2.3 Data structures

Especially the handling of variables can be somewhat complicated from a software engineering point of view, when maximal generality is the goal, however from the purely algorithmic point of view there are no real complications involved:

- Variables are often implemented as unsigned positive integers; literals are then signed integers (other than zero). If variables are not already positive integers themselves, then they need to be associated with an index, so that we can establish constant time access to properties of variables.

- An alternative is to implement variables and literals by pointers to associated data structures with relevant meta-data (like occurrence numbers etc.).

- In any case variables and literals need to be light-weight objects which can be easily copied (note the difference between a literal and a literal-occurrence: given n variables, there are $2n$ literals, while the number of literal-occurrences is the sum of the clause-lengths).

Regarding the implementation of clauses and clause-sets, the basic decision is whether to use "lazy" or "eager" data structures; this is further discussed in Subsection 4.2, and here it suffices to say that conflict-driven solvers are lazy (avoiding to do much work at each node, since they might backtrack soon anyway), while look-ahead solvers are more eager (since the look-ahead at each node needs better support):

1. In the lazy case it is sufficient to implement clauses as vectors (fixed after reading the input).

[3)] while multivalued assignments can allow a set of values

2. While for the eager case clauses are dynamically changed, and are implemented as doubly-linked lists of literal-occurrences.
3. The list of all clauses is not of great importance (one should avoid to run through all clauses), but clauses are accessed through the "clause-literal graph" discussed below.
4. In the eager as well as the lazy case, clauses must enable quick access to associated statistical data.

Clause-sets are

- generalisations of hypergraphs (adding signs to vertices), as well as
- special cases of hypergraphs (with literals as vertices).

Hypergraphs can be represented by bipartite graphs. For clause-sets we obtain the bipartite *clause-literal graph*, which is of fundamental importance:

- the nodes are the literals on one side, and the clauses on the other side;
- edges indicate membership of literals in clauses.

The *clause-variable graph*, connecting now clauses with variables, is also called "incidence graph". Using the standard adjacency-list representation of digraphs and representing graphs by symmetric digraphs, we obtain a basic implementation of clause-sets through the representation of the clause-literal graph, allowing quick access to the literal-occurrences in a clause as well as to the clauses in which a literal occurs. This representation can be considered as fundamental for the lazy as well as for the eager approach, where the former saves certain elements, while the latter adds further structure. Some remarks on the clause-literal graph:

1. More correct is to speak of a 3-partite graph, where the clause-literal graph is augmented with an additional layer for variables.
2. Literal-occurrence correspond to edges between clauses and literals.
3. I consider the graph and hypergraph concepts as a good conceptual framework, however it is used only implicitly by solver implementations.[4)]
4. The technique of "watched literals" together with the "lazy datastructure" for clause-sets can be considered as removing certain (directed) edges from literals to clauses in the clause-literal graph: From a clause we can still reach all contained literals, but a clause is reachable only from two "watched" literal occurrences in each clause, which are updated if necessary; see Subsection 3.1.1.

Finally, for partial assignments two complementary structures are used:

- For search purposes, partial assignments are treated as stacks of assignments (moving down and up the search tree).
- Via an additional global vector of assignments we can check in constant time, whether a variable is assigned, and which value it has.

[4)] The upcoming `OKlibrary` will give direct support for using these graph-theoretic abstractions.

Local search typically works only with "total" assignments (i.e., with partial assignments φ with $\mathrm{var}(\varphi) = \mathrm{var}(F)$, where F is the input clause-set), while for the algorithms considered in this paper partial assignments are fundamental, and then the efficient implementation of the application of partial assignments is of utmost importance (needing additional data structures). Copying is perhaps the most fundamental enemy of efficiency, and the application of partial assignments is (in non-parallel computations) performed *in-place*; more on this and the two fundamental approaches, "eager" and "lazy", can be found in Subsection 4.2.

2.4 Transformations

Let us close this section by some remarks on how to translate other problems into boolean CNFs. First there is the somewhat surprising fact that *boolean transformations are surprisingly efficient*. There are several important extensions of clauses, like

1. cardinality clauses, e.g., $v_1 + v_2 + v_3 \lessgtr k$;
2. more generally pseudo-boolean clauses, allowing constant coefficients;
3. crisp CSP.

For all these cases, direct translation (avoiding sophistication) into boolean CNFs is an efficient way to deal with them (at this time), if a reasonable amount of "logical reasoning" is required by the problem. Boolean CNFs seem to be supported by superiorly efficient data structures — every deviation from this ideal is punished by a big loss in efficiency, which can be compensated only in special situations. But there is another important advantage by using a boolean translation: Not only do we get efficient data structures for free,

but the "atomisation" of information achieved by using boolean variables can be *inherently* more efficient for backtracking algorithms (with exponential speed-ups) than the original information representation.

This important point was raised in [38]: Chopping up a problem into boolean pieces in general increases the search space, but this richer space allows also for more efficient re-combinations.

3 Reductions: poly-time methods

The purpose of this section is to introduce the main reductions used in SAT solving:

1. Unit clause propagation and generalisations are considered in Subsection 3.1.
2. Basic methods directly based on resolution are considered in Subsection 3.2.
3. Some basic comparisons between different notions of "local consistency" are given in Subsection 3.3.
4. Less common reductions are surveyed in Subsection 3.4.

A reduction here is simply a map $r : \mathcal{CLS} \to \mathcal{CLS}$ such that $r(F)$ is satisfiability-equivalent to F, and we consider here only polynomial-time computable r. Now one can study classes $\mathcal{C} \subseteq \mathcal{CLS}$ such that r is already sufficient to decide satisfiability for $F \in \mathcal{C}$, however this point of view in isolation is not very useful for SAT solving (at least not for practical SAT solving), as discussed in Subsection 3.5.

3.1 Generalised unit-clause propagation

We define hierarchies $r_k, r'_k : \mathcal{CLS} \to \mathcal{CLS}$ of poly-time reductions for $k \in \mathbb{N}_0$ as follows:

1. $r_0 = r'_0$ detects the empty clause, and otherwise does nothing:
$$r_0(F) := \begin{cases} \{\bot\} & \text{if } \bot \in F \\ F & \text{otherwise} \end{cases}.$$

2. r_{k+1} reduces F to $r_{k+1}(\langle x \to 1\rangle * F)$ in case r_k yields an inconsistency for $\langle x \to 0\rangle * F$ for some literal x:
$$r_{k+1}(F) := \begin{cases} r_{k+1}(\langle x \to 1\rangle * F) & \text{for literals } x \text{ with } r_k(\langle x \to 0\rangle * F) = \{\bot\} \\ F & \text{otherwise} \end{cases}.$$
r'_{k+1} also notices when a satisfying assignment was found:
$$r'_{k+1}(F) := \begin{cases} r'_{k+1}(\langle x \to 1\rangle * F) & \text{for literals } x \text{ with } r'_k(\langle x \to 0\rangle * F) = \{\bot\} \\ \top & \text{for literals } x \text{ with } r'_k(\langle x \to 1\rangle * F) = \top \\ F & \text{otherwise} \end{cases}.$$

Main properties:

- Though the definition of r_k, r'_k is non-deterministic, these reductions yields unique results (are confluent).
- There always exists partial assignments φ, φ' such that $r_k(F) = \varphi * F$ resp. $r'_k(F) = \varphi' * F$; here φ is a forced (or "necessary") assignment.
- By applying $r_0, r_1, \ldots, r_{n(F)}$ (where $n(F) := |\mathrm{var}(F)|$ is the number of variables) until either an inconsistency is found or at the end we know that F is satisfiable, we obtain a SAT decision algorithm which *quasi-automatises tree resolution*, and which is the (real) essence of Stalmarck's solver. Obviously it is preferable to use the reductions r'_k, which for k large enough (at most $k = n(F)$) will also find a satisfying assignment if F is satisfiable. See [19] for a thorough treatment of the reductions r_k, r'_k.[5)]
- In [26] the treatment of r_k, r'_k is extended to axiomatically given systems of problem instances with non-boolean variables (compare Subsection 2.2.3 in this article).

[5)] Using the hierarchy $G_k(\mathcal{U}, \mathcal{S})$ from [19] (with oracles $\mathcal{U} \subseteq \mathcal{USAT}$ for unsatisfiability and $\mathcal{S} \subseteq \mathcal{SAT}$ for satisfiability) we have $r'_k(F) = \{\bot\} \Leftrightarrow F \in G^0_k(\mathcal{U}_0, \mathcal{S}_0)$ and $r'_k(F) = \top \Leftrightarrow F \in G^1_k(\mathcal{U}_0, \mathcal{S}_0)$, where $\mathcal{U}_0$ is the basic oracle for unsatisfiability, just recognising the empty clause, and $\mathcal{S}_0$ is the basic oracle for satisfiability, just recognising the empty clause-set.

The fundamental open question is how efficiently r_k can be computed for general k:

1. r_1 is just unit-clause-propagation, and we will see in Subsection 3.1.1 that r_1 can be computed in linear time, that is, in $O(\ell(F))$ where $\ell(F) := \sum_{C \in F} |C|$ is the number of literal occurrences in F.
2. We obtain that r_k can be computed in time $O(n(F)^{2(k-1)} \cdot \ell(F))$ for $k \geq 1$.
3. Thus already for r_2 in general we obtain only a cubic-time algorithm. Can we do better ? And what about general k ?

The reductions r'_k can be naturally strengthened via the use of (weak) autarky reduction (see Subsection 3.4.3; a similar early approach is [5]):

$$r^*_{k+1}(F) := \begin{cases} r^*_{k+1}(\langle x \to 1 \rangle * F) & \text{for literals } x \text{ with } r^*_k(\langle x \to 0 \rangle * F) = \{\bot\} \\ r^*_{k+1}(r^*_k(\langle x \to 1 \rangle * F)) & \text{for } x \text{ with } r^*_k(\langle x \to 1 \rangle * F) \subseteq F \\ F & \text{otherwise} \end{cases} .$$

Conflict-driven solvers only use r_1, while look-ahead solvers employ r_k for "$k \approx 2$": The `OKsolver`-2002 (see [24]) uses exactly r^*_2 (apparently as the only solver at each node), while "modern" look-ahead solvers exclude "unpromising" variables from r_2 (thus go below $k = 2$), while employing r_3 for "promising" variables (see Subsection 5.2); the `march`-solvers also employ certain aspects of weak autarky reduction.

3.1.1 Unit-clause propagation

The special case r_1 is *unit-clause propagation* (UCP). UCP of central importance for backtracking solvers, and for efficiency reasons support for UCP needs to be integrated into the main data structure. The basic algorithm for UCP is the *linear time algorithm*, best understood as operating on the clause-literal graph (recall Subsection 2.3), which is represented by an adjacency list:

- a given unit clause $\{x\}$ is "propagated" by removing the literal occurrences of literal $\overline{x}$ (the graph representation yields quick access from a literal to its occurrences, and allows to remove edges efficiently), and it is checked whether this removal creates a new unit-clause (which by the graph representation is a constant-time operation);
- as soon as a new unit-clause is created it is pushed on the buffer (typically a queue or a stack), used for the unit-clauses waiting to be propagated;
- a partial assignment with constant time access keeps track of the assignments, discarding multiple occurrences of the same unit clause, and detecting contradictory unit clauses.

Note that for "lazy UCP", which only needs to obtain the assignments resulting from the unit-clause propagation (used by conflict-driven solvers or in the look-ahead of look-ahead solvers) and not the resulting clause-set, we do not need to consider satisfied clauses. For faster UCP (achieving a better constant factor) the main problem is to make the propagation process more efficient, so that with less work we can detected (relevant) new unit-clauses. A first step is not to remove the occurrences of $\overline{x}$ but just to decrement counters for the clause-lengths, and if this counter reaches 1 then the clause is inspected for the new unit-clause. This is

driven further by *watched literals*: We do not need to know the precise (current) length of all clauses, but we need only to be alerted if possibly we have less than 2 literals in a clause. So we can thin out the clause-literal graph by using only two literal-neighbours of a clause, and updating these neighbours (the watched literals) if one of them disappears. All these methods are only relevant for lazy UCP, which is also for look-ahead solvers of great importance due to the time spent during the look-ahead.

3.1.2 Failed literals and extensions

Reduction by r_2 is called "(full) failed literal reduction". It is not used by conflict-driven solvers, but essential for look-ahead solvers. Failed literal reduction relies on the efficient implementation of UCP, and, as already mentioned, the central question here is: How much can we do better than by just following the definition ?! (Better than just checking for all assignments to variables, whether UCP yields a conflict, and repeating this process if a reduction was found.) The current "front" of research (for look-ahead solver) considers weakenings and strengthenings of r_2 (trying only "promising" variables, and locally learning binary clauses encoding the inferred unit-clauses). See Subsection 5.2 for more information.

3.2 Resolution based reductions

Given two clauses C, D clashing in exactly one literal $x \in C \wedge \overline{x} \in D$, the **resolvent** is

$$\boldsymbol{C \diamond D} := (C \setminus \{x\}) \cup (D \setminus \{\overline{x}\}).$$

Given two clauses C, D and a clause R, the relation $\{C, D\} \models R$, that is, $C \wedge D$ logically implies R, is equivalent to the following:

- Either C and D are resolvable (clash in exactly one literal), and then $C \diamond D \subseteq R$,
- or C and D are not resolvable (clash in zero or at least two literals), and then we have $C \subseteq R$ or $D \subseteq R$.

Thus on the clause level, syntax and semantics coincide! Resolution calculi organise the iterated application of the resolution operation until either the empty clause has been derived, and thus the clause-set is unsatisfiable, or it is established that this is not possible (and thus the clause-set must be satisfiable). Resolution in its various forms, especially tree-like resolution, where the search process can be (relatively) efficiently inverted (starting with the goal), is the central tool for SAT. Via the correspondence between clauses and partial assignments, every backtracking solver is constructing a resolution refutation of its input (see Subsection 7.1). Additional resolution power (moving from tree resolution to full resolution) is gained by "clause learning", and is discussed further in Subsection 6.1 and Section 7. Via the following methods, the resolution operation can be involved in a more direct way:

Adding resolvents Just adding arbitrary resolvents is highly inefficient (except of some special cases). So only short resolvents are added (of length at most 3), and this only during preprocessing.

DP-reductions The **DP-operator** (also referred to as "variable elimination") is

$$\mathrm{DP}_v(F) := \left\{ C \diamond D : C, D \in F \wedge C \cap \overline{D} = \{v\} \right\} \cup \{C \in F : v \notin \mathrm{var}(F)\}.$$

$\mathrm{DP}_v(F)$ is sat-equivalent to F, more precisely, $\mathrm{DP}_v(F)$ is equivalent to the quantified boolean formula $(\exists\, v \in \{0,1\} : F)$, and variable v is eliminated by applying DP_v. So by applying DP until all variables are removed we can decide SAT, but in general this is very inefficient (requiring exponential space). Thus DP is only applied (during preprocessing) in "good cases" (typically when size is not increased).

A general problem here (and elsewhere) regarding reductions is:

To remove or to add clauses ?
That is, simplifying the formula or adding inference power ?

Regarding resolvents, they are typically added.

3.3 Comparison with local consistency notions for CSP's

UCP is the natural mechanism for extending a partial assignment by the obvious inferences. In the language of constraint satisfaction problems, UCP establishes node-consistency (while hyper-arc consistency for clause-sets is trivially fulfilled). More generally, for $k \geq 1$ call a clause-set F *r_k-reduced* if $r_k(F) = F$ holds (so r_1-reduced is the same as node-consistency). How is this consistency notion related to *strong k-consistency* for $k \geq 1$ and clause-sets F (i.e., for every partial assignment φ using strictly less than k variables and fulfilling $\bot \notin \varphi * F$ and for every variable v there is an extension φ' of φ with $\mathrm{var}(\varphi') = \mathrm{var}(\varphi) \cup \{v\}$, such that $\bot \notin \varphi' * F$)? Call a clause-set F *closed under k-bounded resolution* if for all resolvable $C, D \in F$ with $|C \diamond D| \leq k$ we have $C \diamond D \in F$. Now it is easy to see that $F \in \mathcal{CLS}$ is strongly k-consistent for $k \geq 1$ if and only if F is closed under $(k-1)$-bounded resolution. So the question is, how is being r_k-reduced related to being closed under k'-bounded resolution:

1. r_1 is sufficient to show unsatisfiability of all Horn clause-sets, while for every k there exists an unsatisfiable Horn clause-set which is closed under k-bounded resolution but $\bot \notin F$, simply due to the incapability of bounded resolution to handle large clauses.

2. Via small strengthenings of "bounded resolution" however, as discussed in [26], we obtain versions of "k-resolution" which properly generalise r_k for $k \geq 2$.

So in a sense by an adequate repair of the notion of strong $(k+1)$-consistency we obtain a consistency notion which is considerably stronger than being r_k-reduced, however the price is an explosion in memory consumption. One should note here the different contexts for "strong k-consistency" and "r_k-reduced":

- Algorithms for establishing strong k-consistency exploit that constraints as sets of (satisfying) tuples allow to remove arbitrary tuples (which have been found inconsistent), which is not possible with such simple "atomic constraints" as clauses.

- On the other hand, r_k-reduction exploits application of partial assignments by applying enforced assignments, which is supported due to the simplicity of clauses, while constraints typically only handle assignments which cover all their variables.

3.4 Other reductions

3.4.1 Subsumption

Removing subsumed clauses is quite costly, and so mostly done only during preprocessing. See [46] for subsumption elimination also during search, which currently seems to be worth the effort only for harder problems like QBF. This is true for many somewhat more complicated algorithms:

SAT is too easy (currently) for them.

3.4.2 Equivalences

- Equivalences $a \leftrightarrow b$ often are detected (for conflict-driven solvers only during preprocessing), and substituted.
- In general, clauses which correspond to linear equations over $\mathbb{Z}_2$ are sometimes (partially) detected, and some elementary reasoning on them is performed; see [14] for an overview. Most recently however, these facilities seem to be getting removed from "practical SAT solving".

3.4.3 Autarkies

A partial assignment φ is an *autarky* for a clause-set F if every clause of F touched by φ is satisfied by φ:

1. The empty assignment is always an autarky.
2. Every satisfying assignment is an autarky.
3. Composition of two autarkies is again an autarky.
4. Autarkies can be applied satisfiability-equivalently, and thus we have *autarky reduction*.
5. A simplest case of autarky reduction is elimination of pure literals.
6. Since clause-*sets* contract multiple clauses there is also the concept of a *weak autarky* for a clause-set F, a partial assignment φ such that $\varphi * F \subseteq F$. Every autarky is a weak autarky, but not vice versa. Also application of weak autarkies yields satisfiable equivalent (sub-)clause-sets.

Autarkies emerged in a natural way from improved exponential upper bounds on SAT decision ([39, 31, 32, 20]), while the accruing theory of autarkies ([22, 25, 23, 28, 27]) focuses on polynomial-time SAT decision classes on the one hand (embedding matching theory, linear programming and combinatorial reasoning into the (generalised) satisfiability world), and on the other hand on the structure of *lean*

clause-sets which are reduced w.r.t. autarky reduction. Via the notion of an *autarky system* we obtain also generalisations of the notion of minimally unsatisfiable clause-sets, parameterised by special notions of autarkies. In Subsection 3.1 we have already seen initial examples of the use of autarkies in SAT solvers; and see [33] for applications of the fundamental duality between autarkies and resolution. At this time the practical applications seem to be marginal, however I expect this to change within the next 5 years (perhaps especially regarding QBF).

3.4.4 Blocked clauses

The concept of a blocked clause was introduced in [20], with a forerunner in [40], and allows to add to or delete from $F \in \mathcal{CLS}$ a special type of clauses called "blocked":

- Clause C is **blocked** for F if there is some $v \in \mathrm{var}(C)$ such that addition resp. removal of C does not change the outcome of applying DP_v (so one could speak of "inverted DP-reduction").
- Blocked clauses can be added / removed satisfiability-equivalently.
- Addition of blocked clauses containing possibly new variables covers Extended Resolution; so in principle this is very powerful, but we have no guidelines when to perform such an addition.
- Addition of blocked clauses without new variables still goes beyond resolution, as shown in [21], and could be interesting for SAT solvers (the obtained additional inferred assignments are applied directly in [40] for special cases).
- Elimination of blocked clauses was implemented (`lsat`; see [43]), and can help solving some special classes very quickly where all other solvers fail.

3.5 Poly-time classes

Poly-time SAT-decidable classes can play a role for SAT solving as **target classes**:

The heuristics aims at bringing the clause-set closer to the class,
and finally the special algorithm is applied.

However, in practice poly-time classes play yet no role for SAT:

1. They do not occur (on their own!).
2. They do not provide good guidance for the heuristics.

The essential lesson to be learned here seems to me:

Algorithms are more important than classes!

Solvers are algorithm-driven, that is, algorithms are applied also "when they are not applicable", and they are only good, if they are better "than they should be". (And algorithms need a lot of attention and care; they have their own rights, and are not just "attachments" to classes.) For some examples, let us examine the 3 main Schaefer classes:

2-CNF Unsatisfiable instances are handled by failed literal elimination, while satisfiable instances are handled by simple autarky reduction. So some look-ahead solvers solve them "by the way"; but it's not worth looking for them.

Horn Unsatisfiable (renamable) Horn are handled by UCP; there have been many attempts to integrate also the satisfiable cases, but they all failed (in practice). Perhaps a main problem with this class is its brittleness (as clause-sets), while for example the closure under renaming makes it more complicated to deal with, and on the unsatisfiability side we still have a rather trivial class (solved by UCP).

Affine This is the only case of some interest (and further potential), since **equivalences** do occur in special cases, and resolution cannot handle them efficiently. However, due to their special character, affine formulas (resp. their expressions as clause-sets) do not serve as a target class, but are handled by dedicated reasoning mechanisms (see Subsection 3.4.2 above), which could be understood as being handled by specialised "dynamic constraints".

The above statement "it's not worth looking for 2-CNFs" means

- Applying a special test for detecting the (narrow) class of 2-CNF seems to be rather useless.
- Heuristics aiming (just) at bringing down a clause-set to a 2-CNF are too crude.

However, for look-ahead solvers 2-CNFs are kind of basic:

> Some algorithms used to solve 2-CNF (and Horn) are important —
> since these algorithms can solve much more than just 2-CNF (or Horn)!

I hope this illustrates the assertion "algorithms more important than classes".

4 DPLL in general

In this section now we outline the "modern DPLL scheme", with look-ahead solvers (see Section 5) and conflict driven solvers (see Section 6) as special cases. First a note on terminology: "DP, DLL, DPL, DPLL" — these four combinations have been used to describe backtracking algorithms with inference and learning:

1. "DP" is incorrect, since [7] only introduced DP-reduction (see Subsection 3.2) but not the splitting approach.
2. "DLL" refers to [6], the basic backtracking algorithm with unit-clause propagation, elimination of pure literals, and a simple max-occurrences heuristics.
3. "DPL, DPLL" acknowledge the influence of Putnam.

The following pattern seems reasonable (and not uncommon):

1. "DP" for DP-reduction (as it is standard now);
2. "DLL" for simple backtrackers;
3. "DPLL" for the combination of backtracking with resolution (including clause-learning).

4.1 Modern DPLL

A general scheme $\mathfrak{G}$ for DPLL algorithms is now presented, comprising look-ahead as well as conflict-driven solvers. The input is $F_0 \in \mathcal{CLS}$ (possibly after pre-processing). A global variable $\mathbb{L}_0$ contains the learned clauses, and we have $F_0 \models \mathbb{L}_0$ throughout. Thus learning as reflected by $\mathbb{L}_0$ is "global learning", that is, the learned clauses are always to be interpreted w.r.t. the original input F_0 (and not w.r.t. the respective residual clause-sets at each node). Initially $\mathbb{L}_0$ is empty. A parameter, the "history stack" $\mathcal{H}$, contains the information how to interprete $\mathbb{L}_0$ in the current situation, denoted by $\mathcal{H} * \mathbb{L}_0 \in \mathcal{CLS}$ (this might be just application of the partial assignment according to the current path, but it might also contain renamings, substitutions, etc.). Furthermore via $\mathcal{H}$ we can also perform "conflict analysis" (which is not further specified here; for a concrete example see Subsection 6.1.1), and relate a "residual conflict" to F_0. Besides the global variable $\mathbb{L}_0$ (which might be accessed from parallel processes or threads, and thus might need access-control), the procedure $\mathfrak{G}$ is a normal recursive function, with a clause-set F as first argument and $\mathcal{H}$ as second argument (following standard scope rules; initially F is F_0 and $\mathcal{H}$ is empty), and returning an element of $\{0, 1, *\}$, where "$*$" stands for "unknown". The history "stack" $\mathcal{H}$ actually needs to be readable as a whole for conflict analysis, and thus one should better speak of the "history list", but since we mention explicitely only the stack operations (mirroring the ups and downs of the current path) we stick to the notion of a "stack".

$\mathfrak{G}(F \in \mathcal{CLS}, \mathcal{H}) : \{0, 1, *\}$

0. Initialise the local history H as empty.
1. Reduction:
   (a) $F := r(F, \mathcal{H} * \mathbb{L}_0)$;
   (b) add the information about this reduction to H.
2. Analysis:
   (a) Success: If $F = \top$ then return 1.
   (b) Conflict learning and backtracking: If $\bot \in F$ then
      i. via $\mathcal{H}$ compute a set $\mathbb{L}$ of learned clauses;
      ii. $\mathbb{L}_0 := \mathbb{L}_0 \cup \mathbb{L}$;
      iii. $\mathcal{H}$.pop();
      iv. return 0.
   (c) Non-chronological backtracking: Otherwise, if appropriate then
      i. $\mathcal{H}$.pop();
      ii. return $*$.
3. Branching: Compute a finite set $\mathbb{B} \subseteq \mathcal{PASS}$ of partial assignments, and for all $\varphi \in \mathbb{B}$ do (possibly in parallel)
   (a) $\mathcal{H}$.push((H, φ));
   (b) $\delta_\varphi := \mathfrak{G}(\varphi * F, \mathcal{H})$;
   (c) if $\delta_\varphi = 1$ then $F := \top$;
   (d) if $\delta_\varphi = 0$ then $F := F \cup \{C_\varphi\}$.

If these computations are not performed in parallel, then sort $\mathbb{B}$ appropriately before these computations, and break this loop (over $\varphi \in \mathbb{B}$) in case of $\delta_\varphi = 1$.

4. Goto Step 1.

Due to the given specifications, the returned result is always correct; we are not concerned here about establishing general rules for termination (which is not too complicated), nor are we concerned about completeness (equivalent to not performing a non-chronological backtrack at the root of the search tree) — these properties are easily established for the special cases we consider later. Explanations and remarks:

- The map $r : \mathcal{CLS} \times \mathcal{CLS} \to \mathcal{CLS}$ is any "reduction", where we just require that $r(F, \mathbb{L})$ is always satisfiability-equivalent to $F \cup \mathbb{L}$; the separation into two arguments enables special treatment of learned clauses.
- The purpose of the learning step is to enable the reduction r to circumvent the same conflict earlier in the future (when a similar situation arises).
- Non-chronological backtracking is performed if the current situation is better handled at a lower level (closer to the root) in the search tree; this includes the case of a (complete) restart as a special case (through repetition of this step).
- The elements of $\mathbb{B}$ are the "decision assignments":
  - For a look-ahead solver there is a variable v with $\mathbb{B} = \{\langle v \to 0\rangle, \langle v \to 1\rangle\}$, where v is the "branching variable". Thus if both branches returned "unsatisfiable", then the analysis step will confirm the current F as unsatisfiable.
  - For a conflict-driven solver there exists a variable v (again called the "branching variable)" and $\varepsilon \in \{0, 1\}$ with $\mathbb{B} = \{\langle v \to \varepsilon\rangle\}$, and thus here the iterative character of $\mathcal{G}$ is emphasised.
- Sorting of $\mathbb{B}$ shall take advantage of an early success (i.e., a satisfying assignment was found) in some branch: Imagine the situation where one branch is a hard unsatisfiable problem, while the other is an easy satisfiable problem — if not already performed in parallel, then we gain a large speed-up if we have put the satisfiable branch first.
- Note that the return value in Step 3b might be $*$, in which case just another iteration of the loop is performed.
- Pushing the item "(H, φ)" onto the history stack means that we can reconstruct how from the current $F_0 \cup \mathbb{L}_0$ we obtained the current F (through the successive reduction steps as stored on the (whole) stack) and which decisions were involved. The only point where $\mathcal{H}$ is used is Step 2(b)i, where we compute the learned clauses derived from the conflict (and since learned clauses are "global" we need to re-connect the current F to the global level); the loose concept of the history stack is just there to make the flow of informations more visible.
- At Step 3d a "local learning" step is performed (compare Subsection 7.2), that is, the learned clause C_φ is added to the residual clause-set F, and is not

traced back to F_0. One could also apply conflict analysis here, but regarding the character of local learning it seems more appropriate to use the cheaper "full clause-learning" here.

There exist further global monitoring schemes:

- removal of "old" learned clauses;
- using some form of breadth-first search (typically at an early level);
- re-arranging the call order over an initial part of the search tree according to some statistical analysis (compare Subsection 5.3.3).

However yet these extensions have more the character of an "add-on", and time seems not ripe yet to formulate more general patterns, whence these schemes are not present in $\mathfrak{G}$. An important point here is that we have a recursive element of $\mathfrak{G}$, present in the branching step in case of $|\mathbb{B}| \geq 2$, and an iterative element by looping through Steps 1 to 3: Look-ahead solvers (see Section 5) only use this recursive (parallel(!)) aspect, while conflict driven solvers (see Section 6) focus on the iterative aspect (since there always $|\mathbb{B}| = 1$ is the case, the recursion can be eliminated altogether, as done in algorithm `cd` in Subsection 6).[6)]

4.2 The role of partial assignments and resolution

There are two fundamental possibilities when applying partial assignments for branching in Step 3b:

eager really apply the assignment, so that at each node we only see the simplified instances;

lazy only record the assignment, and interprete the clauses as they are visited.

Since look-ahead solvers perform a lot of work at each node, they tend to be "eager" while conflict-driven solver are all "lazy" (if they perform non-chronological backtracking then the work would get lost anyway). Important:

> Application of partial assignments happens "in place", not by copying the instance, but by modification and undoing the modification.

Naturally, undoing the assignment(s) is easier for lazy data structures, but eager data structures pay off in case of heavy local workloads (as for the look-ahead).

DPLL solvers are based on a strong connection to tree resolution and strengthenings (see Subsection 7.1). I regard this as the backbone of SAT solving: Resolution is the "logic of partial assignments", for CSP and beyond, and can be based on a simple algebraic framework (see [26]). In this sense "SAT" and "CSP" can be seen as complementary: Where SAT emphasises the (global) operation of partial assignments on the problem instance, CSP puts more emphasise on exploiting (local)

[6)] These algorithmic aspects should not be mixed up with the purely implementational aspects of simulating recursion in a look-ahead solver via an iterative procedure (using additional stacks). Actually, if the main data structures for the problem instance use in-place modifications (eagerly or lazily), and thus are not included in the recursion, then according to my experience the difference in resource consumption between the more elegant recursive approach and simulation of recursion is negligible.

structure of constraints. The resolution connection explains also intelligent backtracking: By just computing the variables (really) used in the resolution refutation found, intelligent backtracking is possible (implemented in the `OKsolver`-2002; see Subsection 7.1).

5 Look-ahead solvers

In the history of look-ahead solvers, two lines can be distinguished:

1. `Posit` ([11], `Satz` ([34]), and `kcnfs` ([8])
2. Boehm-solver ([4]) and the `OKsolver`-2002 ([24])

while the `march`-solvers ([13, 15]) can be seen as combining those two lines.

The Boehm-solver introduced the special "two-dimensional linked-list" representation of clause-sets as a prototype for an eager data-structure which allows at each node to just see the residual clause-set (after application of the current partial assignment), while the `OKsolver`-2002, using this data structure, demonstrated at the SAT2002-competition that a generic solver with full failed-literal reduction (i.e., r_2), full look-ahead and autarky reduction can be quite efficient (see [44]). The line of `Posit`, `Satz` and `kcnfs` uses a simpler data-structure, more in direction of lazy data structures, where `Posit` uses partial r_2, while `Satz` and `kcnfs` use partial r_2 and partial r_3.

The general "world view" of look-ahead solvers could be summarised as follows:

- For hard problems (thus they can't be too big; say several thousand variables).
- Failed literal reduction and extensions, intelligent heuristics and special structure recognition at the centre.
- The heuristics considers both branches as independent and assumes the worst case. The choice of the first branch (the "direction heuristics", based on estimating the probability of satisfiability) is important on satisfiable instances.
- Eager data structures, with lazy aspects for the look-ahead.
- The aim is as much as possible reduction of problem complexity by inferred assignments (now and in the future).

5.1 The general scheme for look-ahead solvers

We present a simplified version of the general algorithm $\mathfrak{G}$ from Subsection 4.1, where now no conflict learning (and thus no history) is involved, and also no non-chronological backtracking (but see below for examples of restricted usage of global learning in look-ahead solvers to achieve "intelligent backtracking"); on the other side now more details are given about the heuristics for branching.

$\mathtt{la}(F \in \mathcal{CLS}) : \{0, 1\}$

1. Reduction: $F := r(F)$.
2. Analysis: If $F = \top$ then return 1, if $\bot \in F$ then return 0.

3. Branching:
   (a) For each variable $v \in \mathrm{var}(F)$ do:
      i. For each $\varepsilon \in \{0,1\}$ do:
         A. Consider $F_\varepsilon^v := r'(\langle v \to \varepsilon\rangle * F)$,
         B. Compute a "distance vector" $\vec{d}_\varepsilon^v \in \mathbb{R}^m$, where vector $\vec{d}_\varepsilon^v(i)$ for index $i \in \{1,\ldots,m\}$ measures the progress achieved from F to F_ε^v in "dimension i".
      ii. Summarise the distance vector $\vec{d}_\varepsilon^v \in \mathbb{R}^m$ by "distances" $d_\varepsilon^v \in \mathbb{R}_{>0}$.
      iii. Combine the two distance values d_0^v, d_1^v into $\rho_v := \rho(d_0^v, d_1^v) \in \mathbb{R}_{>0}$.
   (b) Choose a branching variable v with minimal ρ_v.
   (c) Return $\max(\mathtt{la}(\langle v \to 0\rangle * F), \mathtt{la}(\langle v \to 1\rangle * F))$, where in case of a non-parallel computation the first branch $\varepsilon \in \{0,1\}$ is chosen such that F_ε^v appears more likely to be satisfiable than $F_{\overline{\varepsilon}}^v$ (and if the first branch returned 1 then the second branch is not considered).

Remarks:

1. r' is the reduction used only for the look-ahead; typically $r \approx r_k$ and $r' \approx r_{k-1}$.
2. A distance vector $\vec{d}_\varphi^v \in \mathbb{R}^m$ measures the progress from F to F_ε^v in an m-dimensional way; a simple example would be to use the number of variables, the number of clauses and the number of literal occurrences (so here $m = 3$). Since there are m components, some of them could be zero (no progress) or even negative (deterioration) if the positive entries outweigh the non-positive entries. See Subsection 5.3.1 for further discussions.
3. See Subsection 5.3.2 for the discussion of the "projection" $\rho : \mathbb{R}_{>0} \times \mathbb{R}_{>0} \to \mathbb{R}_{>0}$.
4. In order to really present an "algorithm" (with well-defined semantics, and not just an "implementation"), the `OKsolver`-2002 performs the look-ahead step 3a fully for all variables, while all other look-ahead solver only perform a "partial look-ahead" on selected variables (this can be incorporated into the above scheme by using appropriate low distance values for variables which don't get selected).

5.2 Reductions: Failed literals and beyond

The most important reductions for look-ahead solvers is given by the range from r_1 to r_3, as presented in Subsection 3.1. The following methods are used to increase efficiency:

1. (Additional) lazy data structures are used, employing time stamps to avoid permanent re-initialisation.
2. Often only "promising" variables are considered for the failed literal reduction (thus weaker than r_2), while for "very promising variables" a double-look ahead is used (reaching r_3 here); see [15] for a recent study.

3. A main problem with r_k for $k \geq 2$ is the (apparent) necessity to run over the formula over and over again to determine whether one reduction triggered other reductions. The simplest thing (perhaps first used by the `OKsolver-2002`) is to realise that if from $x \to 1$ we obtain $y \to 1$ while $x \to 1$ does not yield a contradiction, then (later) $y \to 1$ on its own won't reach a contradiction neither (if nothing has changed meanwhile). This line of reasoning has been considerably strengthened by "tree-based look-ahead" as introduced in [13].

4. Strengthening of r_2 by "local learning": If unsuccessfully $x \to 0$ has been tested (i.e., it does not yield a contradiction), but at least $y \to 1$ was inferred, then the binary clause $(x \vee y)$ may be learned (locally).

What is the point of local learning: Isn't the clause $(x \vee y)$ already "contained" in the current formula, and we only get a shortcut? The point here is that $x \vee y$ is equivalent to $\neg x \to y$ as well as to $\neg y \to x$, and the first direction, "from $x \to 0$ infer $y \to 1$", is given by the current formula, but the second direction "from $y \to 0$ infer $x \to 1$" in general needs a higher level to be inferred; see Subsection 7.2 for more on local learning. All enforced assignments found (iteratively) by r_k strengthened with local learning are also found by r_{k+1}, and thus r_2 with local learning (discussed in Section 3.5 in [19], an experimental feature of the `OKsolver-2002`, and (partially) used by the `march`-solvers) can be seen as an approximation of r_3. An equivalent process to r_2 with local learning is "hyper binary resolution" ([2]).

Regarding autarkies, basic autarky testing was included in the `OKsolver-2002`, and further extended by the `march`-solvers, but yet it seems not of great importance.

5.3 Heuristics

Given a multi-dimensional "distance vector", the simplest possible way to pack it into one number is to use a linear combination; for further information see [30, 29], while here we do not investigate this issue further, but assume that already a "distance" is given.

5.3.1 Distances

The first main task for the heuristics is:

Given the current F and the envisaged branch $v \to \varepsilon$, how "far" do we get when applying $v \to \varepsilon$ (and simplifications) ?

So for each variable v we get two positive real numbers (d_0^v, d_1^v) (the bigger the better). Motivated by the 3-SAT upper bound in [20], the `OKsolver-2002` introduced as distance

the number of *new* clauses.

This might be surprising, since we are not reducing the problem size — but we want to maximise the future gains by the look-ahead reductions! Note that a partial assignment is an autarky iff it does not produce new clauses; this was the reason why autarky testing for branching assignments is included in the `OKsolver-2002`. This distance turned out to be far better than the earlier simple counts (which can be understood as approximations of the number of new clauses created), and has

been taken over by the `march`-solvers. Now, since shorter clauses are better, we need clause weights. Despite many efforts, yet no convincing dynamic scheme exists. A reasonable heuristics gives weight $(\frac{1}{5})^{k-2}$ to new clauses of size $k \geq 2$. (For random 3-CNF an empirically optimal distance is obtained around these values, and look-ahead solvers can be optimised quite well for general purpose SAT solving by just looking at random formulas.)

5.3.2 Projections

Assume now that for each variable the pair (d_0^v, d_1^v) of positive real numbers is given, and we want to "project" the pair to one value $\rho(d_0^v, d_1^v)$, so that the variable with minimal projection is best. For 2 distances, i.e., binary branching, it turns out, that the product $d_0^v \cdot d_1^v$ is good enough, that is, since we are here going for minimisation, the reciprocal value. For arbitrary branching width, in [30, 29] a general theory on distances and projections has been developed (based on [20]), which shows that in general there is *exactly one projection*, namely the "τ-function", and that for width two the product is a good approximation (while in general approximations are given by generalised harmonic means).

5.3.3 The first branch

The most reasonable basic approximation for the probability of a clause-set F being satisfiable seems to me to consider F as a random clause-set in the constant-density model, with mixed densities for the different clause-sizes, and to compute the probability that a random assignment satisfies such a random F, which amounts to minimise

$$\sum_{C \in F} -\log(1 - 2^{-|C|}).$$

This was applied in the `OKsolver`-2002, and an alternative scheme of similar quality is to minimise $\sum_{C \in F} 2^{-|C|}$; for more information see [30]. Howsoever the approximation is obtained, the choice of the first branch is then to choose the branch which looks more likely to be satisfiable. [16] add an additional layer of control by introducing a monitoring depth m (for example $m = 15$), and when it comes to backtracking to this depth (where we have 2^m nodes minus the ones already decided), then the simple "chronological" backtracking order is interrupted, but search continues with another branch at this level according to the principles,

1. that the left branch is preferred over the right branch (because of the direction heuristics),
2. and that higher up the search tree the direction heuristics is more error-prone (since the problems are bigger).

Good results at the SAT2007-competition are demonstrated.

6 Conflict-driven solvers

In this section we give an overview on the main innovations of "conflict-driven" solvers, centred around the notion of "clause learning". The basic intuitions behind conflict-driven solvers seem to be as follows:

- for "simple" but big problems (up to millions of variables);
- optimised for problems from model checking and circuit verification;
- "fast and cheap" (light-weight), nowadays only lazy data structures are used;
- zero look-ahead for the heuristics, just unit-clause propagation reduction;
- the basic aim is: seek for conflicts, learn much.

Historically, one might distinguish 3 main phases:

1. Around 1996 learning was introduced to SAT by `Grasp` ([42]), motivated by previous work in the constraint satisfaction area, but adding the specific "SAT point of view", heavily exploiting clauses and their integration with the problem instance itself.
2. Around 2001 laziness and streamlined learning together with an associated heuristics was introduced by `Chaff` ([36]), emphasising the "fast and cheap" attitude. The success of this solver was the main breakthrough, and the related ideology of a "modern solver" started spinning.
3. Finally, `Minisat` started the "clean up" (for an introduction with algorithms and implementations see [10]).

6.1 Learning

For some initial theoretical analysis (in the framework of proof complexity) see [3]; in this article we focus more on the conceptual side.

6.1.1 The basic ideas

Assume a DPLL-solver reached a conflict, that is, for the current partial assignment φ we have $\bot \in \varphi * F$ (where φ collects all the assignments leading to the current node). The idea is to learn about the conflict so that we can early (!) avoid it at other places in the search tree (thus going beyond tree resolution):

More precisely, we want to learn a "conflict-clause" L
(adding it to the clause-set F)
such that $F \models L$ and $\varphi(L) = 0$.

The condition $\varphi(L) = 0$ is equivalent to $L \subseteq C_\varphi$, and so this part of the learning condition is perfectly clear. It is the first condition, which is equivalent to $\varphi_L * F$ being unsatisfiable, which has a wide scope (being a coNP-complete problem), and where all the variation lies. In Section 7, based on the general considerations from Subsection 2.2.3, approaches towards a general theory of learning are outlined, which might give better explanations of the fundamental ideas of "clause learning" and might open new and more powerful perspectives for the future, especially for the unification with look-ahead solvers. Here now I outline the "traditional" ideas underlying learning in the context of conflict-driven solvers.

First we specify the situation. Let $\mathrm{var}(\varphi) = \{v_1, \ldots, v_{n(\varphi)}\}$ be ordered according to the sequence of unit-clause-eliminations and decisions along the path, where $i_1 < \cdots < i_d$ are the indices of the decision variables. So $d \in \mathbb{N}_0$ is the current

depth in the search tree, and, using $\varphi_i := \varphi \mid \{v_1, \ldots, v_i\}$ and $F_i := \varphi_i * F$ for $i \in \{0, \ldots, n(\varphi)\}$, we have that for $p \in \{1, \ldots, d\}$ the clause-set F_{i_p-1} is reduced w.r.t. r_1, so that decision variable $v_{i_p} \in \mathrm{var}(F_{i_p-1})$ was needed for further progress, while for $p \in \{0, \ldots, d\}$ and $i_p < j < i_{p+1}$ (with $i_0 := -\infty$ and $i_{d+1} := +\infty$) we have $\{x_j\} \in F_{j-1}$, where $\mathrm{var}(x_j) = v_j$ and $\varphi(x_j) = 1$ (this just means that x_j was obtained by unit-clause-elimination). Furthermore assume $\bot \in \varphi * F$, that is, we reached a conflict, and thus $n(\varphi) > i_d$; we also assume $d \geq 1$ (so that we are not already done). Now we have:

1. It's (nearly) completely senseless to learn L_φ (the full path), since the recursive traversal of the search tree will avoid this path anyway, except of schemes with restarts or non-systematic backjumping, where completeness is only guaranteed by clause-learning, and where actually this simplest scheme of learning just the full path is sufficient to establish completeness.

2. By definition there is a clause $L^0 \in F$ with $\varphi(L^0) = 0$. Now L^0 itself is also not very exciting (we know it already), but perhaps we can do something about it? We know that there is $i_d < j \leq n(\varphi)$ with $v_j \in \mathrm{var}(L^0)$ (at least one implied variable from the current level must be involved).

3. Consider any inferred variable $v_i \in \mathrm{var}(L^0)$, and thus $\overline{x_i} \in L^0$. When inferring v_i, an implication
$$\bigwedge_{j \in J} x_j \to x_i$$
has been established with $J \subseteq \{1, \ldots, i-1\}$. Thus we can replace $\overline{x_i}$ in L^0 and obtain $L^1 := (L^0 \setminus \{\overline{x_i}\}) \cup \{\overline{x_j} : j \in J\}$, using that the above implication is equivalent to $\neg x_i \to \bigvee_{j \in J} \neg x_j$.

4. This process of "conflict analysis", replacing inferred literals by their premises (using contraposition), can be repeated, obtaining $L^2, \ldots,$ maximally until only decision variables are left, obtaining a "strongest" conflict-clause L^*.

That's all about the basics of (current) clause-learning, and all learning schemes essentially just vary in

- which inferred literals are replaced, and how far to go back with this process;
- what conflict-clauses then actually to "learn", i.e., to add to F (we might learn several of the L^i above, and potentially also some "side-clauses" involved in the derivation);
- and when learned clauses will be eliminated again (there might be a large number of learned clauses).

The point in adding clause L to F is to enable future tree pruning by more inferred variables (see Subsection 6.1.4) and to guide the "non-chronological backjumping" process (see Subsection 6.1.2). Performing "conflict analysis", that is, recovering the implications in Step 3, is quite simple for a conflict-driven solver: Since such a solver only performs unit-clause-propagation as reduction, every inference step, that is, every new inferred assignment, is witnessed by an existing clause, in the above case by the clause $C_i := \{\overline{x_j} : j \in J\} \cup \{x_i\} \in F$, and thus together with the assignment $x_i \to 1$ a pointer to C_i is stored. This doesn't cause much space

overhead, and when doing the conflict analysis, one only has to look-up the inferred literals in the current envisaged learned clause L and to decide whether to perform the substitution process of Step 3 (which corresponds to a so-called "input resolution" step). We conclude this introduction into the learning process by some observations on clauses L^p obtained in the process of conflict analysis:

1. Every L^p contains some variable from level $d > 0$ (that is, there is $i \geq i_d$ with $v_i \in \mathrm{var}(L^p)$; this is not necessarily true for $d = 0$, since for this level there is no decision variable, but it is the initialisation of the process).
2. If L^p contains some variable from level q, then this is also true for all $p' \geq p$; for p' large enough we have $\mathrm{var}(L^{p'}) \cap \{v_i : i_q \leq i < i_{q+1}\} = \{v_{i_q}\}$ (with $\overline{x_{i_q}} \in L^{p'}$).

6.1.2 Dynamic aspects of clause-learning

One aspect of learning seems most important to me, and it is here that CNF plays an important role: The learned clauses are fully integrated into the original problem, and this can happens here in a simple way, since the original problem is given as a CNF, and we learn *clauses*. It is this dynamic aspect which gives the power to clause-learning, and also makes it a much more complicated process than it first appears.

Consider the situation from the previous Subsection 6.1.1, and moreover we consider the point where the first clause is learned, that is, we are on the left-most branch of the branching tree and encounter the first conflict (so the current clause-database F_0 is the original input-clause-set[7)]). The clearest approach is to consider a "purged" conflict-clause L^* (which only contains decision variables). To make the example simpler, let's assume that the depth d of the current leaf is 100. L^* necessarily contains the literal $\overline{x_{i_{100}}}$; in the best of all cases that's it (since conflict-driven solvers only use r_1-reductions, this would amount to a r_2-reduction for F_0), but in general many more of the literals $\overline{x_{i_1}}, \ldots, \overline{x_{i_{99}}}$ will appear in L^*. The worst case is that L^* contains all of them, and then L^* is essentially useless, but let's be optimistic and assume that $L^* = \{\overline{x_{i_1}}, \overline{x_{i_{50}}}, \overline{x_{i_{100}}}\}$.

We see now that at depth 50 of the current path, that is for $F_{i_{50}}$, we could have inferred the assignment $\langle x_{i_{100}} \to 1\rangle$ (in general, if the solver uses r_k-reduction, this amounts to a r_{k+1}-reduction). This is now the point where "non-chronological backtracking" sets in (actually taking an "eager approach"), and the whole tree starting with this node (which in our assumed case is yet just a path) is reworked, since already the node at level 50 is no longer reduced with respect to r_1, and decision variables $v_{i_{51}}, \ldots, v_{i_{99}}$ possibly could be turned into inferred variables (and furthermore, given the new situation the heuristics might decide differently).

Whatever the learned clause L is, all implied literals from level 100 should be eliminated (i.e., for $v_i \in \mathrm{var}(L)$ we have $i \leq i_{100}$), and then in any case the previous decision variable $v_{i_{100}}$ is now turned into an inferred variable, with the forced value the opposite of the previous value. Thus, while search in the "chronological" recursive approach would consider the second branch belonging to the decision level 100, the non-chronological approach goes back to decision level 50 (99 in the worst case). We see now why the algorithms for conflict-driven solvers are better expressed as an

[7)] with "current" we refer to the possible additions of learned clauses, and not to the "residual" clause-set obtained by applying the current partial assignment

iterative procedure (instead of the usual recursive presentation), where branching and backtracking is just managed by increasing resp. decreasing the decision level, and where the second branch is induced by the conflict clauses: It is not just a matter of convenience, but the clear tree structure (as used by look-ahead solvers) is blurred by doing "partial restarts" in the form of non-chronological backjumping (restarting at $F_{i_{50}}$ in the above example) and leaving the alternation of branches to the inference mechanism — so actually there is no "second branch", and in a sense we are always in the above situation, with just a left-most branch, only that F_0 grows over time (and in practice also shrinks, due to the removal of "inactive" learned clauses — this makes the whole process completely mysterious, and I will mostly ignore this aspect here). A final remark here: As already stated for the general case, the learned clause in the above case must include some variable from level 100 (since otherwise we would have found the conflict at an earlier level), and if we then would find (directly) a conflict at level 50, again the conflict clause must then use some variable from level 50; if however we backtrack to level 0, then there is no decision variable at this level, and thus here we might learn the empty clause and thus conclude that the original input is unsatisfiable.

6.1.3 The iterative solving scheme

More precisely, the iterative ("conflict-driven") procedure $\mathtt{cd} : \mathcal{CLS} \to \{0,1\}$, a special case of the general procedure $\mathfrak{G}$ from Subsection 4.1, works as follows. We use $d \in \mathbb{Z}$ for the decision level (with $d = -1$ indicating an "impossible backtrack"). Instead of managing one global current partial assignment $\varphi \in \mathcal{PASS}$, which is expanded on branching and shrunk on backtracking, for the clarity of exposition we use $\psi_i \in \mathcal{PASS}$ for the initial partial assignment at level i, while $\psi'_i \in \mathcal{PASS}$ is the extended partial assignment with forced assignments added:

$\mathtt{cd}(F \in \mathcal{CLS}) : \{0,1\}$

0. Initialisation: $d := 0$, $\psi_d := \emptyset$. If $\bot \in F$, then return 0.
1. Reduction: Let ψ'_d be obtained by unit-clause-propagation on $\psi_d * F$, that is, $\psi'_d \supseteq \psi_d$ with $\psi'_d * F = r_1(\psi'_d * F) = r_1(\psi_d * F)$.
2. Analysis: Evaluate $\psi'_d * F$.
   (a) If $\psi'_d * F = \top$, then return 1.
   (b) If $\bot \in \psi'_d * F$ then backtrack:
      i. Compute a conflict-clause $L \notin F$ (with $\mathrm{var}(L) \subseteq \mathrm{var}(F)$) w.r.t. ψ'_d.
      ii. $F := F \cup \{L\}$.
      iii. While $d \geq 0$ and $\bot \in \psi'_d * F$ do $d := d - 1$.
      iv. If $d = -1$ then return 0.
      v. While $d \geq 1$ and $r_1(\psi'_{d-1} * F) \neq \psi'_{d-1} * F$ do $d := d - 1$.
      vi. Go to Step 1.
3. Branching:
   (a) Choose a branching variable $v \in \mathrm{var}(\psi'_d * F)$ and $\varepsilon \in \{0,1\}$.
   (b) $d := d + 1$, $\psi_d := \psi'_{d-1} \cup \langle v \to \varepsilon \rangle$.
   (c) Go to Step 1.

Explanations and remarks:

1. Procedure `cd` is correct and also always terminates (whatever the conflict-clause is — if only it is just a new clause).

2. The main choices are the choice of branching variable and first branch in Step 3a, which is discussed in Subsection 6.2, and the choice of the conflict-clause L, which is discussed in Subsection 6.1.4.

3. The conflict-clause L in Step 2(b)i can be chosen here according to the most loose semantics, namely any clause L with $L \subseteq C_{\psi'_d}$ and $F \models L$. It seems not being discussed in the literature, but in the `OKlibrary` we are experimenting with an "afterburner" for clause-learning, which uses additional reasoning power for strengthening the conflict-clause[8)], and then in Step 2(b)iii possibly the "real backtrack" (the backtrack which is necessary) could be more than one step, but if learning is restricted to the conflict analysis from Subsection 6.1.1, then in Step 2(b)iii the depth d is decremented exactly once.

4. In the above formulation of `cd` we consistently used applications of partial assignments to the original input, and not to residual formulas, in order to emphasise the "lazy" aspect of handling the application of partial assignments.

5. Step 2(b)v is the non-chronological backtracking step:

   (a) Correctness and termination does not depend on this step (but only on the added conflict clauses): We could leave it out, or backtrack even further; backtracking to level $d = 0$ would be a full restart (while still keeping the learned clauses(!)).[9)]

   (b) If $r_1(\psi'_{d-1} * F) \neq \psi'_{d-1} * F$, i.e., one level down there are further unit-clause eliminations possible, then we have also $r_1(\psi'_d * F) \neq \psi'_d * F$. So in general the decision levels after adding the conflict-clause can be divided into three connected parts: At the end we have the "contradicting levels" (only d in the standard situation), then come the "active levels" where further unit-clause propagations are possible, and Step 2(b)iii jumps to the beginning of this segment, and finally (i.e., at the beginning of the decision stack) we have the "unaffected levels" (this segment is empty iff we learned a clause L which after elimination of level-0-variables contains at most one literal).

   (c) Since we only learn one clause L (instead of several clauses) and only use r_1 (instead of for example r_2), the condition "$r_1(\psi'_{d-1} * F) \neq \psi'_{d-1} * F$" is equivalent to $|\psi'_{d-1} * L| = 1$, that is, all but exactly one literal in L is falsified by the "current" partial assignment of level $d - 1$. This is the condition as normally stated, but the condition in Step 2(b)v seems to be the real underlying reasoning (also allowing to use stronger means than r_1).

[8)] this is more natural for "look-ahead solvers", since they have a stronger reasoning machinery anyway

[9)] However if this step is not performed fully then the "real backtrack" in Step 2(b)iii could involve more than one level even when sticking to the conflict analysis from Subsection 6.1.1, since earlier decision levels might have unprocessed unit-clause propagations.

6.1.4 Learning schemes

Now we consider Step 2(b)i in algorithm `cd` from Subsection 6.1.3 in more detail; in Subsection 6.1.1 we have outlined the general idea, resulting in a non-deterministic sequence $L^0, \ldots, L^*$ of conflict clauses (but with well-defined first and last element), and the question is now which L^p to learn.

As we have already explained, in order to make the backtracking system "aware" of the fact, that the decision $x_{i_d} \to 1$ was a failure, and hence $x_{i_d} \to 0$ is inferred, we need $\bot \in \psi_d * F$ after the learning step, i.e., all inferred literals from level d must have been eliminated in the learned clause L (such a conflict-clause then is called "asserting"). Just performing such elimination steps (that is, only eliminating inferred literals from level d) we obtain a well-defined conflict-clause L^+, which can be considered as the "weakest" conflict-clause, while at the other end of the spectrum we have the "strongest" conflict-clause L^*, where all possible elimination steps have been performed (and only decision variables are left). All existing learning schemes are situated between L^+ and L^*, where the choice of L^+ is called "1UIP" ([47]), while the choice of L^* has been given different names like "decision cut".[10] Unfortunately not much can really be said here, but let us consider the most fundamental decision, between L^+ and L^*. Above they have been called "weakest" and "strongest" according to the effort involved to obtain these clauses — now is this really true, that is, does the increased effort for computing L^* at least pay off in a smaller search tree? Due to the "sporadic" character of problem instances where conflict-driven solvers are successful (not on the most amenable instances for systematic studies, random formulas, where conflict-driven solvers perform badly), and due to the high sensitivity of solvers regarding the learning process (which is determinative for the heuristics) it seems that all empirical arguments are rather weak. And there are no theoretical results in any form. However, "in practice" L^+ (that is, the 1UIP-scheme) turns out to be the winner (and this for all learning schemes). It seems that despite all arguments for various schemes now the field of conflict-driven solvers converges on 1UIP.

As we have already mentioned, if some variable from level i is involved, then never level i can be emptied (because we only use the old inferences), and since the literal elimination process replaces literals by literals with smaller indices, we see that for the non-chronological backjump depth the clause L^+ is as good as L^* or anything between, so this step is not influenced by the learning scheme. In [9] it is argued that 1UIP has an advantage over other schemes because empirically more unit-clause propagations are enabled by these conflict-clauses (one could roughly say that 1UIP offers more "surface" for future conflicts (i.e., resolution steps), while going far back somehow narrows the choices). At other places it is furthermore argued that going "far back" conflicts with the idea of "locality", namely that statistics on usage of conflict clauses shall guide the branching heuristics.

6.2 Heuristic

Finally we investigate the heuristics for the choice of branching variable v and branching value ε in Step 3a of procedure `cd` from Subsection 6.1.3. It seems

[10] This is perhaps the right point to remark that I fail to see the point of the common "cut terminology", where a (fake) directed graph is constructed recording the events of literal inferences: The (in principal) very simple character of learning is obscured in this way, and if graph-theoretical notions shall be employed then one should use the appropriate notion of *directed hypergraphs* here.

that regarding the heuristics, conflict-driven solvers still live in the "stone age" of simple literal counts, far behind look-ahead solvers — but they have one special weapon, based on the dynamic nature of the "clause database" due to the added conflict-clauses. However, compared to the situation for look-ahead solvers, where we have two independent branches and thus the total workload is minimised while as first branch one is chosen which could make a difference (i.e., where actually the independence breaks down, since in case of a satisfiable assignment found we simply abort), here now the notion of "branches" is broken open, and a more global situation has to be faced. The current guideline seems to be a greedy approach, seeking for as many "profitable" conflicts as possible.[11)]

Regarding the branching value, there are two conflicting goals: Seeking for "good" conflicts, or trying to find a satisfying assignment. In [36] the greedy choice of searching for conflicts is also applied to the branching value, but apparently this wasn't very successful and the simple choice $\varepsilon := 0$ seems to be more popular and still the prevalent method (and only with SAT 2007 some discussions started regarding an improved choice). Perhaps due to their weak "statistical infrastructure", conflict-driven solvers don't have good measures at hand to estimate the probability of satisfiability, and thus do not employ a direction heuristics as discussed in Subsection 5.3.3 for look-ahead solvers. Furthermore it seems that on many instances coming from hardware verification actually truth value 0 is a reasonable choice due to the special encoding. And, as already mentioned, while for a look-ahead solver in principle the direction is clear (towards satisfiability), for a conflict-driven solver also a "fruitful conflict" might be attractive.

Now regarding the branching variable (for an overview on some techniques see [36]) the basic is just the (static) literal count for the input clause-set (preference for higher counts). This static count evolves into some *activity measurement* by dynamic updates:

- learning a clause containing the variable increases the activity;
- the activity decays over time.

Following the "locality idea", branching variables are chosen which have the highest activity. The motivation for such schemes might be summarised by

> Where are many conflicts, there will be more.
> And conflicts are good (since they cut off branches).

Various schemes about how much to increase and to decay have been proposed, but it seems to me that only the idea of variable activity regarding activity in conflict-clauses has fundamental virtues.

7 Towards a general theory of clause-learning

In this final section I want to present some general ideas and methods which extend clause-learning, and can help putting it into a wider context.

[11)] Perhaps it is this analogy which drives proponents of conflict-driven solvers to call their approach "modern", while denying this qualification to "modern" look-ahead solvers: "Modern" in a sense of "modern (disaster) capitalism", and "old-fashioned" in the sense of "socialistic planning".

7.1 From branching trees to resolution trees

The essential first step in understanding clause-learning is to understand the full translation of backtracking trees (possibly with local reductions like r_k) into resolution trees. A complete presentation in a general context is given in [26], but the principle is very simple:

1. Unfold the backtracking tree, replacing r_k-reductions by little sub-trees in (generalised) input-resolution-tree shape, finally obtaining a pure backtracking tree where nodes just represent splitting on variables.
2. At each leaf select a clause falsified by the partial assignment corresponding to the path to that leaf.
3. Starting from the leaves, perform resolution operations where branching variables now become resolution variables.
4. Cases, where actually one of the parent clauses of an envisaged resolution step does not contain the resolution variable, correspond to "intelligent backtracking": Only this branch is kept, while the other is discarded.

A "global learning step" corresponds to creating the tree from the leaves in a certain order and then allowing to link to the learned clause from later parts of the graph (which becomes a dag now). Learning "non-conflict clauses" corresponds to learning clauses from the "little sub-trees" corresponding to inference steps.

7.2 Local versus global learning

Due to their lazy datastructures, conflict-driven solvers have problem seeing the current clause-set (that is, with the current partial assignment applied), and accordingly the clauses they learn are always "global", that is, all assumptions are carried out and the clause can be added to the original input clause-set. However, it might be worth learning clauses first "locally", and unfolding the assumptions (decisions) only when backtracking.

The simplest such learning scheme for look-ahead solvers has been already mentioned in Subsection 5.2: Recall the r_2-reduction from Subsection 3.1, and assume that when testing the assignment $x \to 1$ we derived $y \to 1$ (by unit-clause propagation) but we didn't reach a failed literal (and thus r_2 was unsuccessful in this case); actually any reduction which allows from assumptions $x \to 1$ to infer forced assignments $y \to 1$ can be used here. Though the reduction attempt failed, nevertheless we gained some inference information, and how can we use it? The derivation $x \rightsquigarrow y$ is just equivalent to the fact that the clause $\{\overline{x}, y\}$ follows from the current clause-set, and thus it can be learned here. Note that the clause $\{\overline{x}, y\}$ is a valid inference only "locally", that is, w.r.t. the current partial assignment (which is carried out in look-ahead solvers), while when backtracking then either the clause must be "unfolded", that is, some conflict analysis has to be performed to make the assumption about the previous decision variable explicit, or the locally learned clauses are just discarded when backtracking.

Let us again point out what is the use of locally learning the clause $\{\overline{x}, y\}$: For global learning as described before we mentioned the tree pruning effect, exploiting conflict analysis, which obviously cannot happen if we discard the clause upon backtracking. And the implication $x \to y$ is contained in the current formula

anyway, so why learning $\{\overline{x}, y\}$? First, there is the aspect of a "short cut", but more importantly the clause $\{\overline{x}, y\}$ represents *both* implications $x \to y$ and $\neg y \to \neg x$, and while the first implication is found by the current reduction scheme like r_k, for the converse we need the next level of reduction r_{k+1} ! This effect is also active for global learning, and so learning of clauses L (as discussed in Subsection 6.1.1) which still contain inferred literals can yield inferences which are not obtained by the in a sense strongest learned clause L^* — again, the "forward" implications are all contained in L^*, but the "backward" implications need a higher level of reduction to come to light.

In the general DPLL-procedure $\mathfrak{G}$ from Subsection 4.1, local learning is the task of the reduction r: the learned clauses are discarded upon backtracking, and only via Step 2(b)i of $\mathfrak{G}$ (the formation of learned clauses) can information be saved from oblivion. $\mathfrak{G}$ doesn't allow intermediate steps, the gradual unfolding of learned information, and it seems to me that this should be an interesting possibility to explore; for now however we only consider "pure" local learning. We have already seen that r_k-reduction with local learning can be simulated by r_{k+1}-reduction. Asking about the strength as a proof system of branching trees with local learning in this sense is thus translated into the question about the strength of branching trees with reduction r_{k+1} at each node. If k is fixed, then such trees can save a polynomial factor over simple branching trees (that is, over tree resolution), which can be relevant for practice, but is less impressive from a proof-theoretical point of view: The reason about the reduced strength of local learning is that the learned clauses only gather information along the *path* from the root to the current node, while a global learned clause gathers information from its whole corresponding sub-*tree*. So aspects of global learning are definitely necessary to reach greater proof-theoretic strength.

However a (theoretical) possibility for a form of local learning which can even go beyond full resolution is that in Step 3d of $\mathfrak{G}$ we use a branching $\mathbb{B}$ such that $\{C_\varphi : \varphi \in \mathbb{B}\}$ is a hard unsatisfiable clause-set.[12)] Yet such schemes were rarely considered, and they look rather difficult to make efficient; a possibly more accessible extension of resolution is considered in the following subsection.

7.3 Compressed learning

In [26] resolution was generalised by using "oracles" $\bot \in \mathcal{U} \subseteq \mathcal{USAT}$ for unsatisfiability, where the only condition is that $\mathcal{U}$ is stable under application of partial assignments. Using $\mathcal{U} = \mathcal{USAT}$ trivialises everything, while $\mathcal{U} = \{\bot\}$ is exactly tree resolution. The abstract point of view of [26] is that from a problem instance P we "see" only whether for a partial assignment φ we have $\varphi * P \in \mathcal{U}$ or not, where in the positive case we learn C_φ, and the set of all learned clauses constitutes F which is the basis for some ordinary resolution process. For conflict-driven solvers, $\mathcal{U}$ would be just the set of clause-sets refutable by r_1 (that is, the set of clause-sets containing an unsatisfiable renamed Horn clause-set), while a reasonable stronger $\mathcal{U}$ here could be the set of clause-sets refuted by r_2 combined with some form of equivalence reasoning. Learning only clauses consisting solely of decision variables abstracts away from the inferences, and a strong $\mathcal{U}$ yields smaller branching trees as well as shorter learned clauses. One should remark here that in [26] a kind of simple "static" point of view is taken, and learning happens only at the leaves while at

[12)] The task then is to make $\mathbb{B}$ as "similar" to F as possible — the best case is to use $\mathbb{B} = \{\varphi_C : C \in F\}$!

inner nodes just resolution steps happen, but obviously this can be made dynamic by taking the learned clauses into account for new learning steps (and by using non-chronological backtracking).

A big problem (quite literally) for the unification of look-ahead techniques with the conflict-driven approach is that the latter focuses on rather big instances, where the polynomial-time overhead of look-ahead solvers can actually result in weeks(!) of wasted run-time (on a single instance). As envisaged in the plans for the new `OKsolver` (which are contained in the `OKlibrary` themselves), the usage of an "after-burner" for learning, which only turns on the stronger inference machine for compressing clauses to be learned, could be a solution for this problem.

8 Conclusion

There is something more fundamental to "clause-learning" and "conflict-driven solvers" than, as suggested, "raw speed, super-efficient implementations and cleverly adapted heuristics". Instead, the old paradigm of backtracking-search has been made "reflective", reflecting the search meta-level onto the problem object-level. Perhaps only in the "purified context" of SAT (compared to CSP) this paradigm could have been evolved further, given our current (lack of) understanding.

However, also the traditional backtracking approach has seen substantial improvements through the look-ahead techniques and the related theory of backtracking heuristics. There is a certain agreement that SAT has reached in a certain way three "local optima" (for local search, look-ahead and conflict-driven), which actually seem to be three rather large plateaus. One opinion on this situation is that further progress with SAT lies mainly in considering applications, the "user interface", and integration with extensions. A (smaller) part of the SAT community however believes that we just started, and I hope that with this article some elements of the SAT-solvers to come have been outlined. If, metaphorically speaking, the learning-based approaches created a mirror-cabinet on the search process, the task is now to look behind the mirrors.

References

[1] Carlos Ansótegui, María Luisa Bonet, Jordi Levy, and Felip Manyà. Mapping CSP into many-valued SAT. In Marques-Silva and Sakallah [37], pages 10–15. ISBN 978-3-540-72787-3.

[2] Fahiem Bacchus and Jonathan Winter. Effective preprocessing with hyper-resolution and equality reduction. In Giunchiglia and Tacchella [12], pages 341–355. ISBN 3-540-20851-8.

[3] Paul Beame, Henry Kautz, and Ashish Sabharwal. Towards understanding and harnessing the potential of clause learning. *Journal of Artificial Intelligence Research*, 22:319–351, 2004.

[4] Max Böhm. *Verteilte Lösung harter Probleme: Schneller Lastausgleich*. PhD thesis, Universität Köln, 1996.

[5] Mukesh Dalal and David W. Etherington. A hierarchy of tractable satisfiability problems. *Information Processing Letters*, 44:173–180, 1992.

[6] Martin Davis, George Logemann, and Donald Loveland. A machine program for theorem-proving. *Communication of the ACM*, 5:394–397, 1962.

[7] Martin Davis and Hilary Putnam. A computing procedure for quantification theory. *Journal of the ACM*, 7:201–215, 1960.

[8] Gilles Dequen and Olivier Dubois. kcnfs: An efficient solver for random k-SAT formulae. In Giunchiglia and Tacchella [12], pages 486–501. ISBN 3-540-20851-8.

[9] Nachum Dershowitz, Ziyad Hanna, and Alexander Nadel. Towards a better understanding of the functionality of a conflict-driven SAT solver. In Marques-Silva and Sakallah [37], pages 287–293. ISBN 978-3-540-72787-3.

[10] Niklas Eén and Niklas Sörensson. An extensible SAT-solver. In Giunchiglia and Tacchella [12], pages 502–518. ISBN 3-540-20851-8.

[11] Jon William Freeman. *Improvements to propositional satisfiability search algorithms*. PhD thesis, University of Pennsylvania, 1995.

[12] Enrico Giunchiglia and Armando Tacchella, editors. *Theory and Applications of Satisfiability Testing 2003*, volume 2919 of *Lecture Notes in Computer Science*, Berlin, 2004. Springer. ISBN 3-540-20851-8.

[13] Marijn Heule, Mark Dufour, Joris van Zwieten, and Hans van Maaren. March_eq: Implementing additional reasoning into an efficient look-ahead SAT solver. In Hoos and Mitchell [17], pages 345–359. ISBN 3-540-27829-X.

[14] Marijn Heule and Hans van Maaren. Aligning CNF- and equivalence-reasoning. In Hoos and Mitchell [17], pages 145–156. ISBN 3-540-27829-X.

[15] Marijn Heule and Hans van Maaren. Effective incorporation of double look-ahead procedures. In Marques-Silva and Sakallah [37], pages 258–271. ISBN 978-3-540-72787-3.

[16] Marijn J.H. Heule and Hans van Maaren. Whose side are you on? Finding solutions in a biased search-tree. To appear, October 2007.

[17] Holger H. Hoos and David G. Mitchell, editors. *Theory and Applications of Satisfiability Testing 2004*, volume 3542 of *Lecture Notes in Computer Science*, Berlin, 2005. Springer. ISBN 3-540-27829-X.

[18] Oliver Kullmann. A survey on practical SAT algorithms. In Nadia Creignou, Phokion Kolaitis, and Heribert Vollmer, editors, *Complexity of Constraints*, Lecture Notes in Computer Science (LNCS). Springer.

[19] 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.

[20] Oliver Kullmann. New methods for 3-SAT decision and worst-case analysis. *Theoretical Computer Science*, 223(1-2):1–72, July 1999.

[21] Oliver Kullmann. On a generalization of extended resolution. *Discrete Applied Mathematics*, 96-97(1-3):149–176, 1999.

[22] Oliver Kullmann. Investigations on autark assignments. *Discrete Applied Mathematics*, 107:99–137, 2000.

[23] Oliver Kullmann. On the use of autarkies for satisfiability decision. In Henry Kautz and Bart Selman, editors, *LICS 2001 Workshop on Theory and Applications of Satisfiability Testing (SAT 2001)*, volume 9 of *Electronic Notes in Discrete Mathematics (ENDM)*. Elsevier Science, June 2001.

[24] Oliver Kullmann. Investigating the behaviour of a SAT solver on random formulas. Technical Report CSR 23-2002, Swansea University, Computer Science Report Series (available from `http://www-compsci.swan.ac.uk/reports/2002.html`), October 2002.

[25] Oliver Kullmann. Lean clause-sets: Generalizations of minimally unsatisfiable clause-sets. *Discrete Applied Mathematics*, 130:209–249, 2003.

[26] 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.

[27] Oliver Kullmann. Constraint satisfaction problems in clausal form: Autarkies and minimal unsatisfiability. Technical Report TR 07-055, Electronic Colloquium on Computational Complexity (ECCC), June 2007.

[28] Oliver Kullmann. Polynomial time SAT decision for complementation-invariant clause-sets, and sign-non-singular matrices. In Marques-Silva and Sakallah [37], pages 314–327. ISBN 978-3-540-72787-3.

[29] Oliver Kullmann. Fundaments of branching heuristics. In Armin Biere, Hans van Maaren, and Toby Walsh, editors, *Handbook of Satisfiability*. IOS Press, 2008.

[30] Oliver Kullmann. Fundaments of branching heuristics: Theory and examples. Technical Report CSR 7-2008, Swansea University, Computer Science Report Series (`http://www-compsci.swan.ac.uk/reports/2008.html`), April 2008.

[31] Oliver Kullmann and Horst Luckhardt. Deciding propositional tautologies: Algorithms and their complexity. Preprint, 82 pages; the ps-file can be obtained at `http://cs.swan.ac.uk/~csoliver/`, January 1997.

[32] Oliver Kullmann and Horst Luckhardt. Algorithms for SAT/TAUT decision based on various measures. Preprint, 71 pages; the ps-file can be obtained from `http://cs.swan.ac.uk/~csoliver/`, December 1998.

[33] Oliver Kullmann, Inês Lynce, and João Marques-Silva. Categorisation of clauses in conjunctive normal forms: Minimally unsatisfiable sub-clause-sets and the lean kernel. In Armin Biere and Carla P. Gomes, editors, *Theory and Applications of Satisfiability Testing - SAT 2006*, volume 4121 of *Lecture Notes in Computer Science*, pages 22–35. Springer, 2006. ISBN 3-540-37206-7.

[34] Chu Min Li and Anbulagan. Heuristics based on unit propagation for satisfiability problems. In *Proceedings of 15th International Joint Conference on Artificial Intelligence (IJCAI'97)*, pages 366–371. Morgan Kaufmann Publishers, 1997.

[35] László Lovász, Moni Naor, Ilan Newman, and Avi Wigderson. Search problems in the decision tree model. *SIAM Journal on Discrete Mathematics*, 8(1):119–132, 1995.

[36] Yogesh S. Mahajan, Zhaohui Fu, and Sharad Malik. Zchaff2004: An efficient SAT solver. In Hoos and Mitchell [17], pages 360–375. ISBN 3-540-27829-X.

[37] Joao Marques-Silva and Karem A. Sakallah, editors. *Theory and Applications of Satisfiability Testing - SAT 2007*, volume 4501 of *Lecture Notes in Computer Science*. Springer, 2007. ISBN 978-3-540-72787-3.

[38] David G. Mitchell and Joey Hwang. 2-way vs. d-way branching for CSP. In Peter van Beek, editor, *Principles and Practice of Constraint Programming — CP 2005*, volume 3709 of *Lecture Notes in Computer Science*, pages 343–357. Springer, 2005. ISBN 3-540-29238-1.

[39] B. Monien and Ewald Speckenmeyer. Solving satisfiability in less than 2^n steps. *Discrete Applied Mathematics*, 10:287–295, 1985.

[40] Paul W. Purdom. Solving satisfiability with less searching. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 6(4):510–513, 1984.

[41] Nathan Segerlind. The complexity of propositional proofs. *The Bulletin of Symbolic Logic*, 13(4):417–481, December 2007.

[42] Joao P. Marques Silva and Karem A. Sakallah. GRASP—a new search algorithm for satisfiability. Technical Report CSE-TR-292-96, University of Michigan, Department of Electrical Engineering and Computer Science, 1996.

[43] Laurent Simon and Daniel Le Berre. The essentials of the SAT 2003 competition. In Giunchiglia and Tacchella [12], pages 452–467. ISBN 3-540-20851-8.

[44] Laurent Simon, Daniel Le Berre, and Edward A. Hirsch. The SAT2002 competition. *Annals of Mathematics and Artificial Intelligence*, 43:307–342, 2005.

[45] Alasdair Urquhart. The complexity of propositional proofs. *The Bulletin of Symbolic Logic*, 1(4):425–467, 1995.

[46] Lintao Zhang. On subsumption removal and on-the-fly CNF simplification. In Fahiem Bacchus and Toby Walsh, editors, *Theory and Applications of Satisfiability Testing 2005*, volume 3569 of *Lecture Notes in Computer Science*, pages 482–489, Berlin, 2005. Springer. ISBN 3-540-26276-8.

[47] Lintao Zhang, Conor F. Madigan, Matthew H. Moskewicz, and Sharad Malik. Efficient conflict driven learning in a boolean satisfiability solver. In *Proceedings of the International Conference on Computer Aided Design (ICCAD)*, pages 279–285. IEEE Press, 2001.