Practical SAT algorithms

Oliver Kullmann
Computer Science Department
University of Wales Swansea

Complexity of Constraints
Dagstuhl, October 3, 2006

SAT algorithms for boolean CNFs:
Local search and DPLL

Introduction

- SAT algorithms as represented by the SAT conference and the SAT competition
- emphasis on "practical" algorithms, which "work" (at least reasonably often)
- Two basic approaches: Stochastic local search (SLS) and backtracking
  - not much changes regarding SLS in recent years
  - steady improvements of look-ahead backtracking solvers
  - much interest in conflict-driven backtracking solvers
- extensions of SAT increasingly popular
- many beliefs, many observations, no proofs.

Theory and Practice

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 I want to show you the “practical world” — hopefully we can learn from their observations and ideas! (Though it won’t be easy.)

Overview

Clause-sets and (partial) assignments

- We have **variables** with boolean domain $\{0, 1\}$.
- From variables we build **positive literals** and **negative literals**, stating that the variables must become true resp. false.
- Two literals **clash** if they have the same variables, but different polarities.
- **Clauses** are finite and clash-free sets of literals, understood as disjunctions.
- **Clause-sets** are finite sets of clauses, understood as conjunctions.

Emphasise on **boolean CNFs**, since they (seem) to embody the “secret” of SAT.

Theoretically clause-<u>sets</u> are convenient — are they also used in practice ?

Commutativity and associativity enforced ?

- Associativity of disjunction and conjunction is always implement (by using lists).
- Commutativity of disjunction (in a clause) may be implemented by ordering the literals in a clause. Often this is not done, but it seems that it is unimportant.
- 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”).

Idempotency and clash-freeness

- Literals are not repeated in clauses, and clauses are clash-free (simplifying the input if necessary, and maintaining these invariants).
- Repeated clauses are only removed during pre-processing (if at all; and 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).

Variables and literals

- 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.

Partial assignments

- A **partial assignment** is a map $\varphi : V \to \{0, 1\}$ for some set V of variables (perhaps empty).
- For search purposes, partial assignments act also as stacks of assignments (moving down and up the search tree), and via an additional global vector of assignments we can check in constant time, whether a variable is assigned, and which value it has.

A fundamental fact (justifying the use of *partial* assignments):

If a partial assignment φ satisfies a clause-set F (i.e., satisfies every clause), then every extension of φ also satisfies F.

Applying partial assignment

Perhaps the most fundamental process for SAT solving is the **application of partial assignments**:

Given φ and F, we can apply the assignments of φ to F (efficiently); the result is denoted by $\boldsymbol{\varphi * F}$.

$\varphi * F$ is obtained from F by removing all clauses satisfied by φ, and removing those literal occurrences from the remaining clauses which are set to false by φ:

- φ **falsifies** F if $\bot \in \varphi * F$, where $\bot$ is the **empty clause.**
- φ **satisfies** F if $\varphi * F = \top$, where $\top$ is the **empty clause-set.**

Unit clause elimination

Arguably the most important aspect of clauses for SAT solving is, that once all literals are falsified except of one, then the remaining variable gets an enforced value.

This is based on three properties of clauses:

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

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 (such clauses are also called "no-goods", when φ is a failing assignment, and they are the essence of "learning").

This correspondence establishes the close relation between the **search trees** of backtracking algorithms and **resolution refutations**.

Lists of lists

Clause-sets are

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

The basic implementation is that of a hypergraph as a list of lists.

Of utmost importance is the implementation of the application of partial assignments. This needs additional data structures.

The bipartite clause-literal graph

(General) Hypergraphs can be represented by (general) bipartite graphs. For clause-sets this means:

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

Using the standard adjacency-list representation of digraphs, and representing graphs by symmetric digraphs, we obtain a basic implementation of clause-sets, which is typically regarded as

adding literal-occurrence lists to the list-of-lists representation.

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. I consider the graph and hypergraph concepts as a good conceptual framework, however it is used only implicitly by solver implementers. (I'll try to change this with the upcoming `OKlibrary`.)
3. From a clause we can always go to its literals (efficiently). Using the clause-literal graph, we can then also go from a literal to its clauses.
4. The technique of "watched literals" together with the "lazy datastructure" for clause-sets means, that we do not consider all the neighbours of the literals, but a clause is neighbour only for two "watched" literal occurrences in each clause, which are updated if necessary. See "UCP".

Boolean transformations surprisingly efficient

Several important extensions of clauses:

1. cardinality clauses: $v_1 + v_2 + v_3 \overset{\leq}{\underset{\geq}{=}} k$
2. more generally pseudo-boolean clauses, allowing constant coefficients
3. crisp CSP.

Direct translation (avoiding sophistication) into boolean CNFs is the most efficient way to deal with them (at this time), if a reasonable amount of "logical reasoning" is required by the problem.

This surprising efficiency of (simple) boolean translations into CNF needs explanation.

Wide branching vs. binary branching

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.

(As shown by David Mitchell and Joey Hwang.)

List of “losers”

Evolutionary
: makes a lot of noise (since it’s “modern”), but is irrelevant for SAT solving: these algorithms make only sense when nearly nothing is known. Might be applicable for meta-problems such as learning of heuristics.

Continuous
: was very popular in the eighties and nineties, but disappeared now.

Survey Propagation (SP)
: is very interesting, but works only for large satisfiable uniform random instances (based on a domain-specific analysis).

Poly-time hierarchies and FPT
: yet too crude and special cases not found in practice.

Generalised unit clause propagation

We define a hierarchy r_k of poly-time reductions for $k \in \mathbb{N}_0$ as follows:

1. r_0 detects the empty clause, and otherwise does nothing.
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.

Main properties:

- Though the definition of r_k is non-deterministic, the reduction yields a unique result (is confluent).
- By applying $r_0, r_1, \dots$ until either an inconsistency is found or a satisfying assignment (exploiting r_k in the obvious way), we obtain a SAT decision algorithm which **quasi-automatises tree resolution**, and is the (real) essence of Stalmarck's solver.

Unit clause propagation

- The special case r_1 is **unit clause propagation** (UCP).
- UCP of central importance for backtracking solvers; for efficiency reasons should 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 (thus with fast access from literals to occurrences),
    - using a buffer for the unit clauses waiting to be propagated
    - together with a partial assignment (with constant time access) for keeping track of the assignments.
- So for UCP we do not need to consider satisfied clauses.

Faster UCP

- Of great importance now is to find out new unit clauses (created by applying some earlier unit clause propagation) as early as possible.
- So we seek to save also on the side of the shortened clauses.
- An interesting possibility (used by all conflict-driven solvers) here are **watched literals**:

  We do not need to know from all clauses their precise (current) length, but we need only to be alerted if possibly we have less than 2 literals in a clause.

Failed literals and extensions

- Reduction by r_2 is called “(full) failed literal reduction”.
- Not used by conflict-driven solvers, but essential for look-ahead solvers.
- Failed literal reduction relies on the efficient implementation of UCP.

The central question here:

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.)

Current “front” of research (for look-ahead): weakenings and strengthenings of r_2 (trying only “promising” variables, and locally learning binary clauses encoding the inferred unit clauses).

Comparison with local consistency notions for CSPs

- UCP is the natural mechanism for extending a partial assignment with the obvious inferences.
- In the language of constraint problems, UCP establishes node-consistency (while hyper-arc consistency for clause-sets is trivially fulfilled).

What about *strong k-consistency* for $k > 1$ 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 $\bot \notin \varphi' * F$)?

Strong local consistency

For $k \geq 1$ a clause-set F is r_k**-reduced** if $r_k(F) = F$ holds:

1. r_1-reduced is the same as node-consistency, which by definition is the same as (strong) 1-consistency.
2. r_2-reduction suffices to decide 2-CNF (that is, a 2-CNF F is unsatisfiable iff $r_2(F) = \{\bot\}$), while for every k there are unsatisfiable 2-CNFs which are strongly k-consistent.
3. Being r_2-reduced implies strong 2-consistency.
4. A clause-set F containing clauses $\{a, b, x\}, \{a, b, \overline{x}\}$ is not 3-consistent, but it can be r_k-reduced for arbitrarily large k (if only the tree-resolution complexity is high enough).

Incomparability between the reduction approaches

So being r_k-reduced and k-consistency are incomparable for $k \geq 3$:

- 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 clauses.
- Strong k-consistency for clause-sets can be achieved by adding resolvents, or by strengthening r_k by suitable forms of *local learning*.
- On the other hand, r_k-reduction exploits application of partial assignments by applying enforced assignments: F is reduced to $\langle x \to 1 \rangle * F$ iff $\langle x \to 0 \rangle * F$ was found inconsistent at "level $k - 1$".

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.

There is a theory of autarkies, which allows to embed matching theory, linear programming and combinatorial reasoning into the (generalised) satisfiability world, and which establishes a duality to resolution.

However, at this time the practical applications seem (yet) to be marginal.

Resolution

Given two clauses C, D clashing in exactly one literal

$$x \in C \wedge \overline{x} \in D,$$

the **resolvent** is

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

Resolution is (basically)

exactly what you obtain when restricting the semantical implication relation to clauses.

Thus on the clause level, **syntax and semantics coincide** !

Resolution in its various forms (especially tree-like) 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.

Resolution reductions

Just adding 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.

Clause learning (discussed later) is an important tool for adding (some) power of full resolution to tree resolution.

A general problem here (and elsewhere):

To remove or to add clauses ?!

Regarding resolvents, they are typically added.

DP reductions

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

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

1. $\mathrm{DP}_v(F)$ is sat-equivalent to F.
2. 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).

Subsumption

Removing subsumed clauses is quite costly, and so mostly done only during preprocessing.

There are a few tricks and techniques, but (currently) they seem 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.

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.
- Most recently however, these facilities are getting removed.

Blocked clauses

- 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.
- Blocked clauses can be added / removed sat-equivalently.
- Addition of blocked clauses containing new variables covers Extended Resolution; so in principle very powerful, but no clue how to apply it.
- Addition of blocked clauses without new variables also goes beyond resolution, and could be interesting.
- Elimination of blocked clauses was implemented, and on some special classes can yield “spectacular” results.

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.

Actually, 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.

Algorithms 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”.)

The Schaefer classes

Let's examine the 4 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; many attempts to integrate also the satisfiable cases, but they all failed. Forget Horn.

Affine This is the only case of some interest (and further potential), since **equivalences** do occur (as parts(!)), and resolution cannot handle them efficiently.

Remark on 2-CNF

When I say “it’s not worth looking from them”, then I mean

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

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

Some algorithms used to solve 2-CNF are important — since these algorithms can solve much more than just 2-CNF!

I hope this also illustrates the assertion “algorithms more important than classes”.

Introduction

Local search for SAT had its great time in the 80's and 90's.

Then came the revenge of DPLL.

Nowadays, local search only plays a role on satisfiable random instances (where it is dominant).

At least for SAT solvers, not so much news over the last 2 years or so, so here only a sketch (and furthermore, it's not really typical for SAT).

In recent years, SLS algorithms have been used successfully in worst-case upper bounds.

The basic architecture

The basic architecture is still that of GSAT ("G" like "greedy"):

1. consider all variables
2. flip a variable which minimises the number of falsified clauses (this number might increase by such a flip)
3. after a fixed number of rounds, restart
4. after a fixed number of tries, give up.

Improvements

- consider only a subset of variables (for example only flips in falsified clauses as in WalkSAT)
- stronger randomisation of flips (less greed, establishing also some form of completeness)
- avoid repeating moves (tabu search)
- dynamic adaption (clause weights).

Hybrid solvers

- Quite a few attempts to combine complete search and local search, but yet has to find its ways into the "practical solvers" only considered here.
- Perhaps most successful UnitWalk (Edward Hirsch), combining WalkSAT with unit propagation, motivated by the theoretical analysis of Paturi, Pudlak and Zane.

DP, DLL, DPL, DPLL

These four combinations have been used to describe

backtracking algorithms with inference and learning.

1. "DP" is incorrect, since [Davis/Putnam 1960] discussed "DP-reduction".
2. "DLL" refers to [Davis/Logemann/Loveland 1962], 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.

I use

1. "DP" for DP-reduction (as it is standard now)
2. "DLL" for simple backtrackers
3. "DPLL" for the "combination" (somehow).

The general scheme

1. reduction of F (with potential “local learning”); potentially stronger reduction at the root (pre-processing)
2. if decided:
   - return with satisfying assignment, or
   - backtrack, potentially “intelligently”, potentially with learning of the conflict (learning only at the leaves)
3. choose branching variable, potentially choosing a first branch, and branch.

Resource monitoring schemes:

- removal of old learned clauses
- restarts (keeping learned clauses)
- (pseudo-)parallelisation (taking “disastrous errors” into account)

Applying partial assignments

2 fundamental possibilities when applying partial assignments for branching (while the look-ahead is different):

eager	really apply the assignment, so that in the branches we look at the simplified instances
lazy	only record the assignment, and interprete the clauses as they are visited.

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).

Connections to resolution

DPLL solvers are based on a strong connection to tree resolution and strengthenings. 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 my articles))).

The current stage: between tree resolution and regular resolution (with aspects of full resolution).

The resolution connection explains also intelligent backtracking: By just computing the variables used in the resolution refutation found, intelligent backtracking is possible (implemented in the old `OKsolver`).

The general view of “look-ahead solvers”

- For hard problems (thus they can’t be too big; say several thousand variables).
- Failed literal reduction, intelligent heuristics and special structure recognition at the centre.
  And the choice of the first branch (“direction heuristics”, based on estimating the probability of satisfiability) important on satisfiable instances.
- Eager data structures, with lazy aspects for the look-ahead.
- Aim: As much as possible reduction of problem complexity by inferred assignments.

Failed literals

1. (Additional) lazy data structures are used (employing time stamps).
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).
3. Strengthening of r_2 by "local learning": If unsuccessfully tested $x \to 0$, but inferred $y \to 1$, then the binary clause $(x \vee y)$ may be learned.

What is the point of local learning: Isn't the clause $(x \vee y)$ already "contained", and we only get a shortcut?

No, it's actually stronger: the point here is not the direction "from $x \to 0$ infer $y \to 1$", but

$$\text{from } y \to 0 \text{ infer } x \to 1.$$

Distances

The first main task for the heuristics:

Given the current F and the envisaged branch $x \to 0$, how "far" do we get when applying $x \to 0$ (and simplifications) ?

So for each variable v we get two positive real numbers (d_0^v, d_1^v) (the bigger the better).

A good 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!

Projections

Now having for each variable the pair (d_0^v, d_1^v) of positive real numbers, and we want to "project" the pair to one value, 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.

For arbitrary branching width I have developed a little theory, 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 for width 3 and greater no reasonable approximation is known to me.)

Clause weights

As mentioned, modern look-ahead heuristics try to maximise the number of new clauses.

Since shorter clauses are better, we need weights.

Despite many efforts, yet no convincing dynamic scheme exists.

A reasonable heuristics gives weight $(\frac{1}{5})^{k-1}$ to clauses of size $k \geq 2$.

(For random 3-CNF an optimal distance is obtained around these values, and look-ahead solvers can be optimised quite well by looking at random formulas.)

General introduction

The intuitions behind “conflict-driven solvers”:

- for simple but big problems (up to millions of variables)
- exploiting special characteristics (even dirty tricks) for problems from model checking and circuit verification
- “fast and cheap” (light-weight)
- nowadays all lazy
- zero look-ahead
- aim: seek for conflicts, learn much.

Historically, there have been 2 phases:

1. Around 1996 learning with UIP (Marques Silva).
2. Around 2001 laziness and Unique UIP (“fast and cheap; Chaff by Lintao Zhang)

Clause learning in general

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 now 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 clause C
(adding it to the clause-set)
such that $F \models C$ and $\varphi(C) = 0$.

It's completely senseless to learn C_φ.

There is a clause $C_0 \in F$ with $\varphi(C_0) = 0$. Now C_0 itself is also not very exciting, but perhaps we can do something about it?

Conflict analysis

Consider a literal $x \in C_0$. We have $\varphi(x) = 0$ — why?

If x is a **decision variable**, then we are done with x.

Otherwise we must find the **causes** of the (inferred) assignment $\varphi(x) = 0$: There is some $\varphi' \subset \varphi$ such that UCP for $\varphi' * F$ yields that assignment to x (look at the clause causing the unit clause for $\varphi(x) = 0$ — the assignments to the other literals are φ').

So we can go from C_0 to $C_1 := (C_0 \setminus \{x\}) \cup C_{\varphi'}$. This clause C_1 is more interesting than C_0.

This process of cause-expansion for non-decision variable can in principle be iterated

$$C_0 \to C_1 \to C_2 \to \ldots \to C_n$$

until only decision variables are left.

UIP

Conflict-driven solvers by far don't go back to the decision variables:

1. This process is considered to be too expensive.
2. In this way locality is lost, which is important for the variable activity heuristics.

So several "cheaper" variations have been considered. The current winner seems to be "unique UIP" ("unit implication point"):

Expand all literals just beyond the most recent decision variable (except of that decision variable itself).

Variable activity

Following the “locality idea”, branching variables are chosen which have the highest **activity**:

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

So the solver really is conflict-driven:

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

SAT modulo theories

In the last year or so, SAT solvers didn't develop much further, but there was a strong move towards extending the applications of SAT.

For example, applications from hardware and software applications ask for (some) integer variables x, y and literals(!) of the form e.g. $x + 6 \geq y$.

As we have seen, boolean translations are quite competitive. If however the domain-specific part gets too heavy, then the "SAT modulo theories" approach is quite successful:

SAT modulo theories II

1. The theory-literals are first “abstracted away” as propositional variables.
2. If the SAT solver finds this problem unsatisfiable, then the original problem is unsatisfiable.
3. If however the SAT solver returns a satisfying assignment, then the theory-solver is asked whether this assignment is realisable for the domain-specific variables:
    1. If yes, then the problem is satisfiable.
    2. If not, then the causes of the conflict are learned (similar to what we’ve seen before, on the propositional level), and it’s the SAT solvers turn again.

Summary

- Most essential for SAT solving seems to be to handle **unsatisfiability**, and for this efficient proof search for resolution refutations is central.
- Two different DPLL-solver “styles” (look-ahead, conflict-driven), caused by different application profiles.
- Local search as an additional tool to handle satisfiability.
- Outlook
  - In the last two years or so not much news regarding SAT solvers, but concentrating on extensions of SAT.
  - In my opinion, most important for better SAT solvers is the **integration** of the three architectures (look-ahead, conflict driven, local search).

End

Epilogue: Why intelligent methods fail in practice

Applying non-trivial methods needs an **“industrial-strength process”**:

- no single “solvers”, written by students and assistants,
- embodying a few ideas, which do not work (typical excuse “promising”, “not so many years of development” etc.),
- and which either already at the lifetime of the author become unmanageable, or at the latest after he/she leaves. (And another idea is buried.)

Generative libraries: The (bright) future

Needed is a **generative library** providing reusable abstract components:

- which can be arbitrarily combined,
- and moreover integrated into more complex combined systems due to suitable abstractions
- with every upgrade of the library, applications gain from progress made in other components
- embedded into a strong test and measurement system, which guarantees maintainability and observability for much longer than usual
- due to its high level and published concepts, applications can be maintained by people different than the author
- finally non-programmers can write SAT solvers by the use of certain domain-specific specification languages (supported by graphical user interfaces).