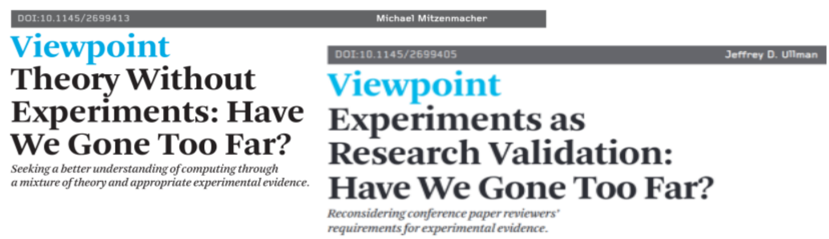
We are stuck:
How to compare techniques in solvers? And how to write such a solver?

Armin Biere, Mathias Fleury, and
Karem Sakallah

2022/10/14



DOI:10.1145/2699413

Michael Mitzenmacher

Viewpoint

Theory Without Experiments: Have We Gone Too Far?

Seeking a better understanding of computing through a mixture of theory and appropriate experimental evidence.

DOI:10.1145/2699405

Jeffrey D. Ullman

Viewpoint

Experiments as Research Validation: Have We Gone Too Far?

Reconsidering conference paper reviewers' requirements for experimental evidence.

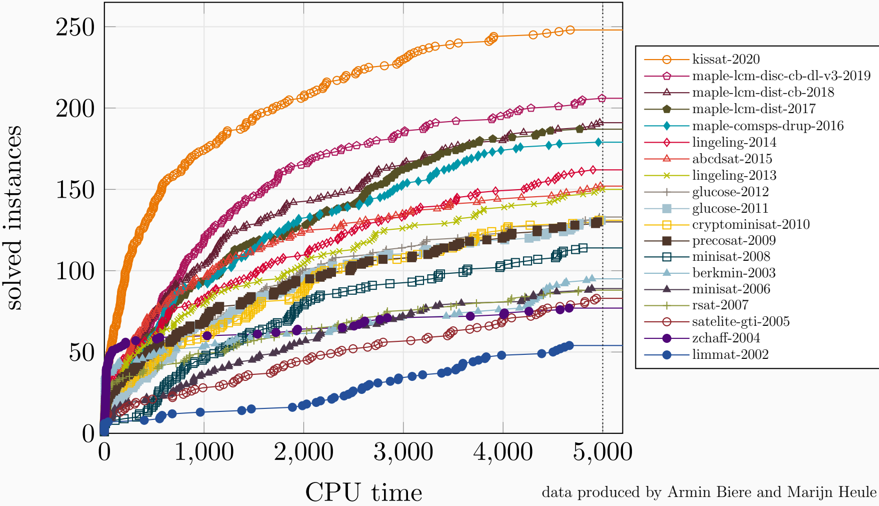
Conversation with Karem

Experts are not certain that progress was made

Even Moshe Vardi!

Do you understand what is happening in the last 25 years?

SAT Competition Winners on the SC2020 Benchmark Suite



data produced by Armin Biere and Marijn Heule

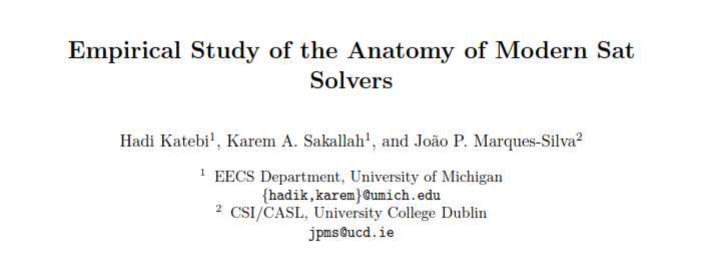
I want to redo my paper from 2011.

Sounds good

But I need your help to understand all the new features

that should be possible

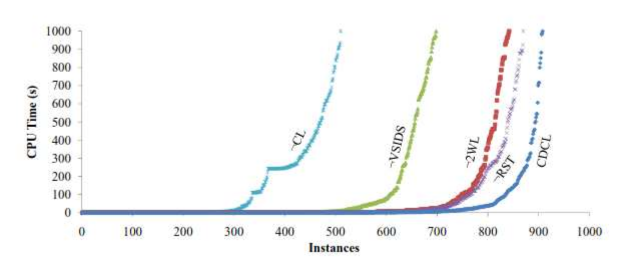
How do we understand SAT solvers?



Empirical Study of the Anatomy of Modern Sat Solvers

Hadi Katebi[1], Karem A. Sakallah[1], and João P. Marques-Silva[2]

[1] EECS Department, University of Michigan
{hadik,karem}@umich.edu
[2] CSI/CASL, University College Dublin
jpms@ucd.ie



A controversial paper

This Talk: Can we do that better?

We are stuck

We need your help here

What are the features in a SAT solver?

We need to list all the options from SAT solvers

I don’t understand them, but you do.

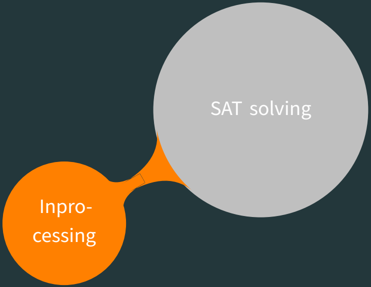
Let’s start with...

That is an absolutely terrible name

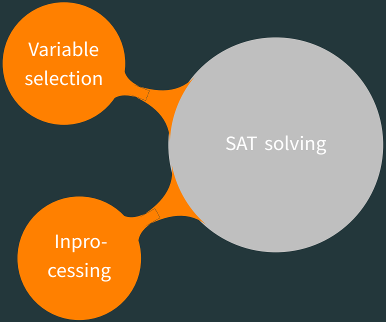
The features

SAT solving

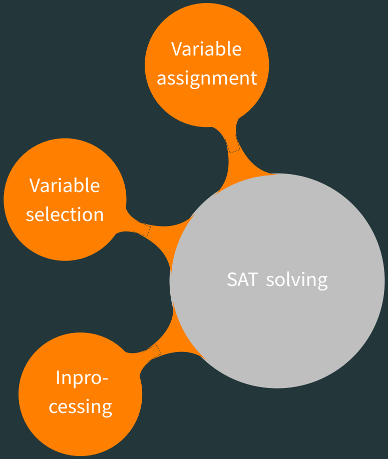
The features



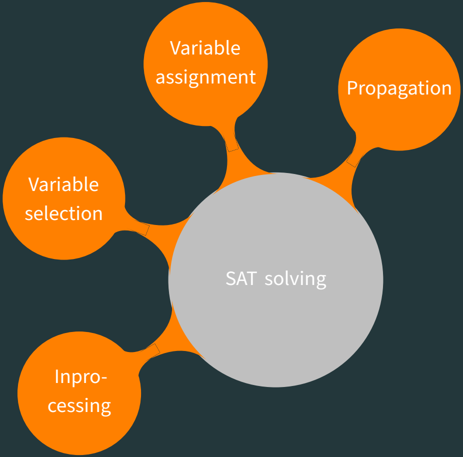
The features



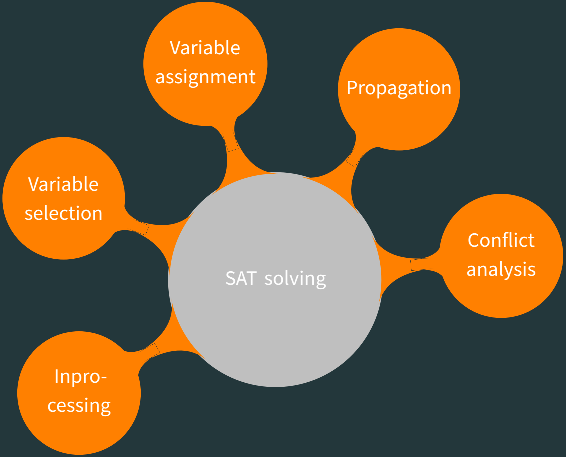
The features



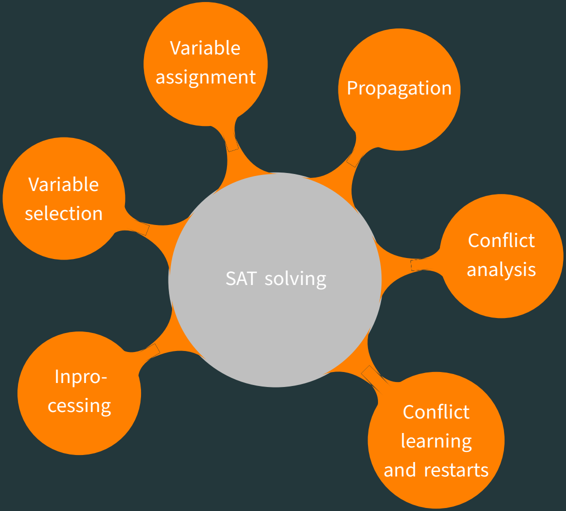
The features



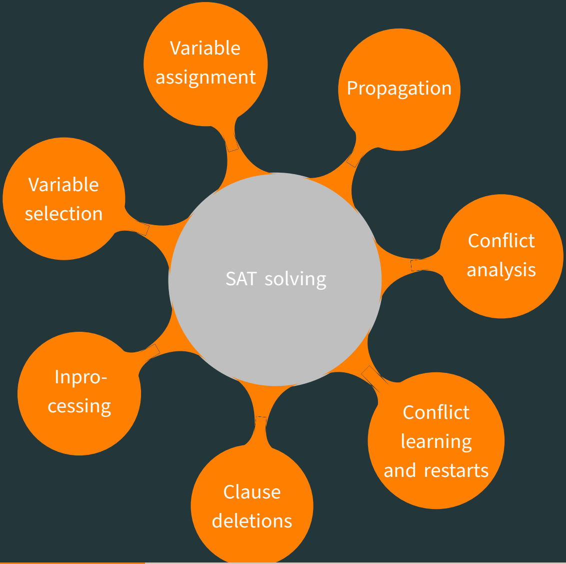
The features



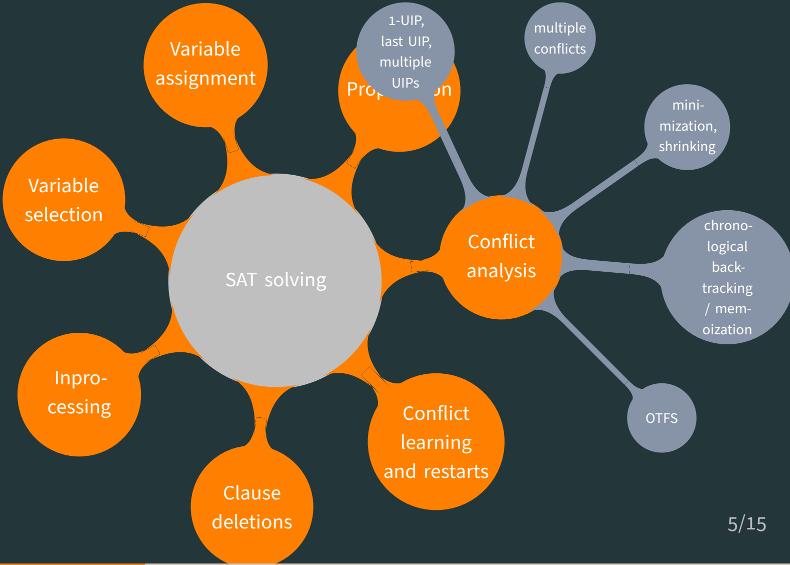
The features



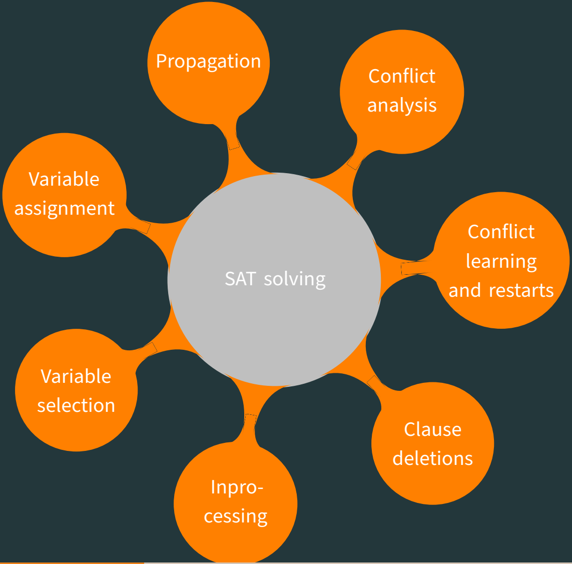
The features



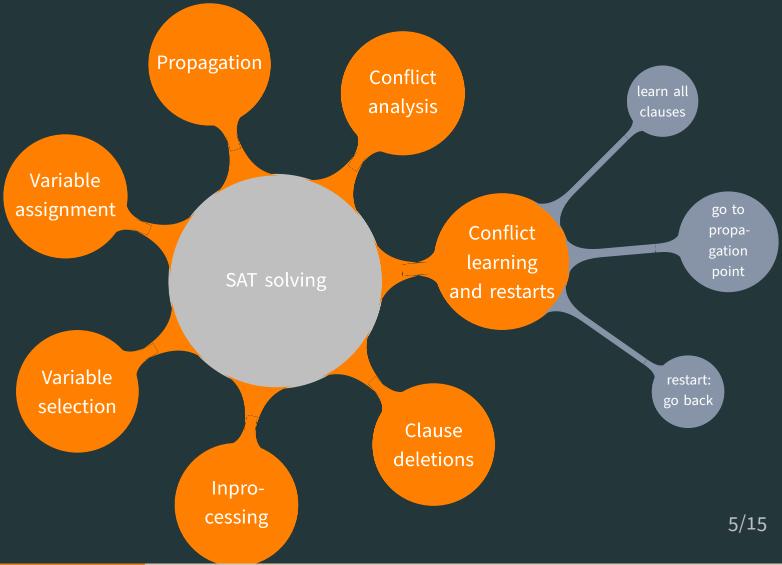
The features



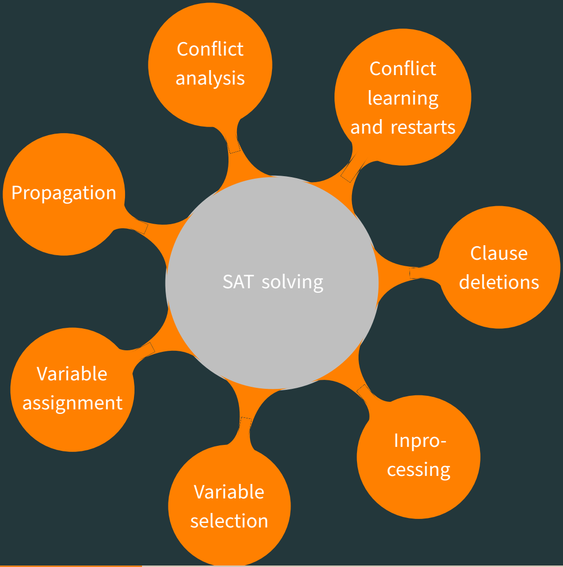
The features



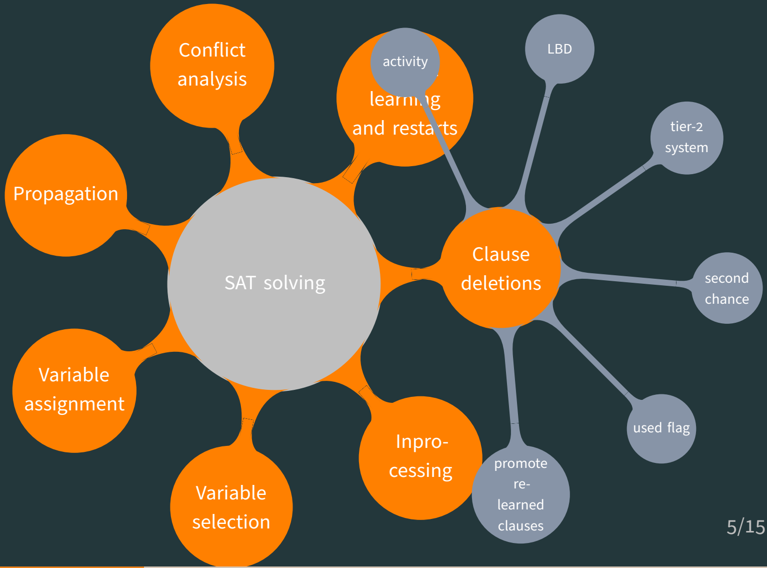
The features



The features



The features



SAT solving

Conflict analysis

learning and restarts

Propagation

Variable assignment

Variable selection

Inprocessing

Clause deletions

activity

LBD

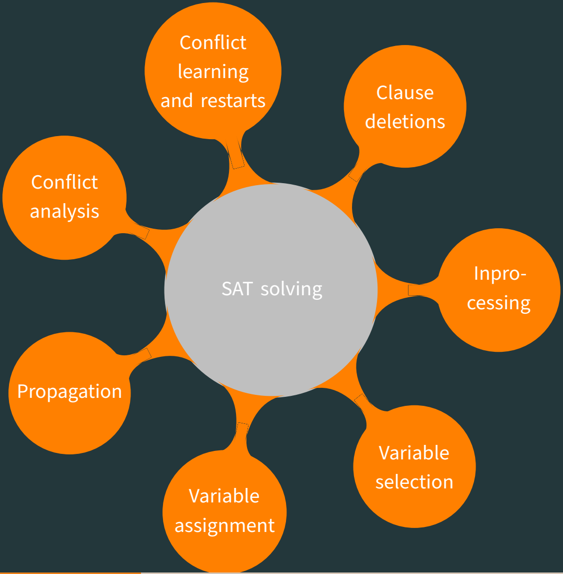
tier-2 system

second chance

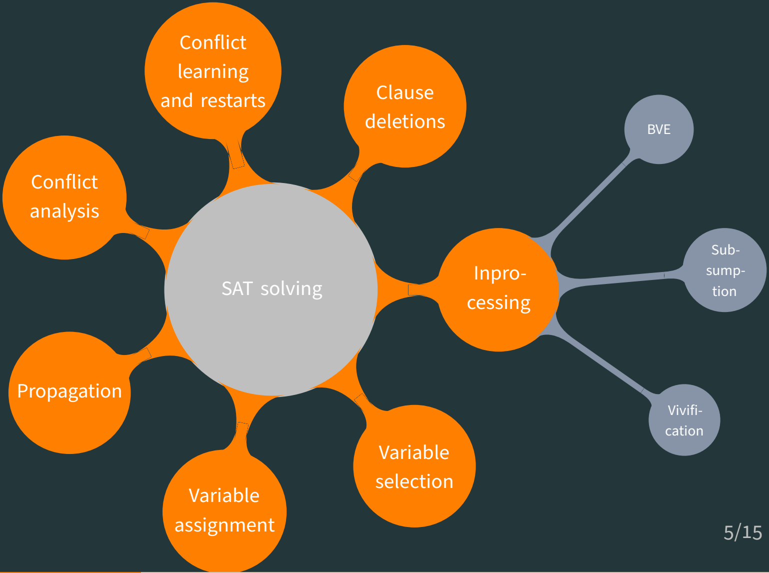
used flag

promote re-learned clauses

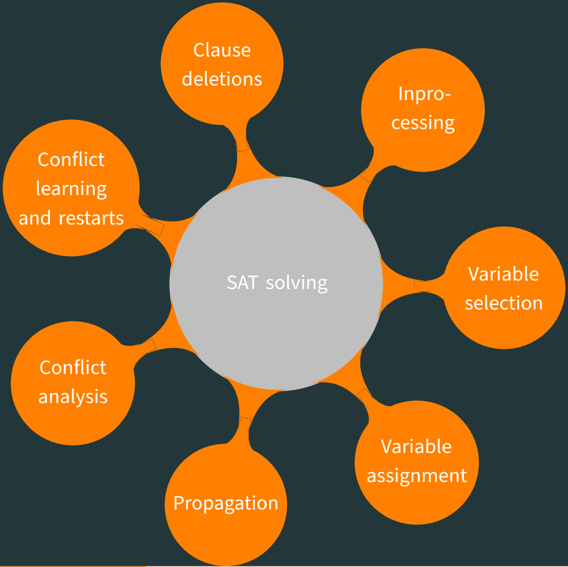
The features



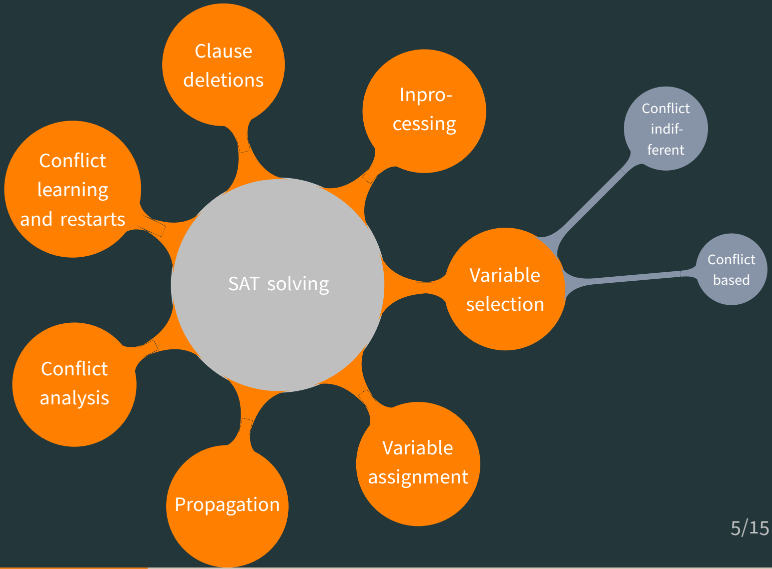
The features



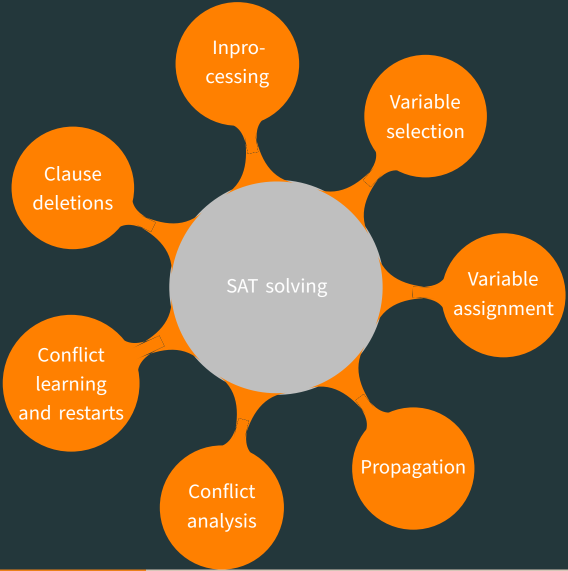
The features



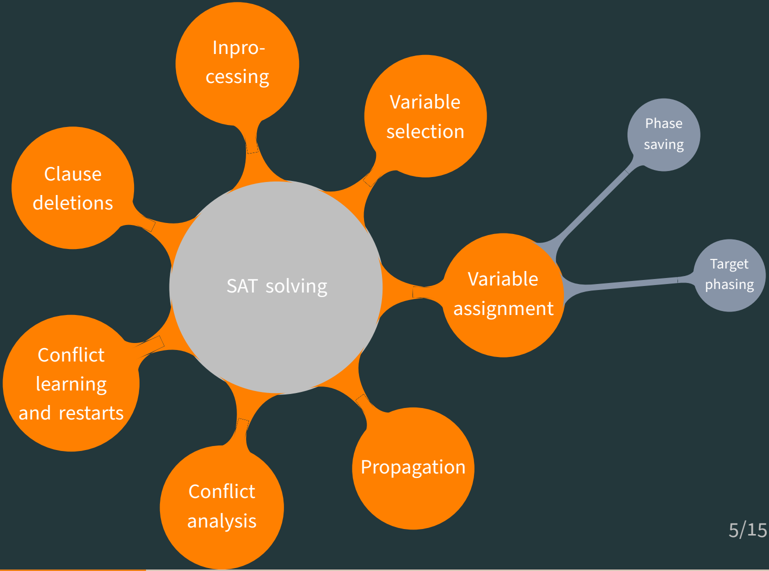
The features



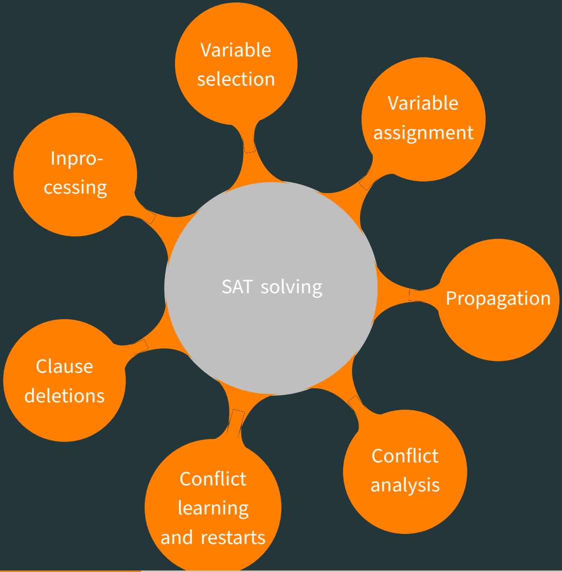
The features



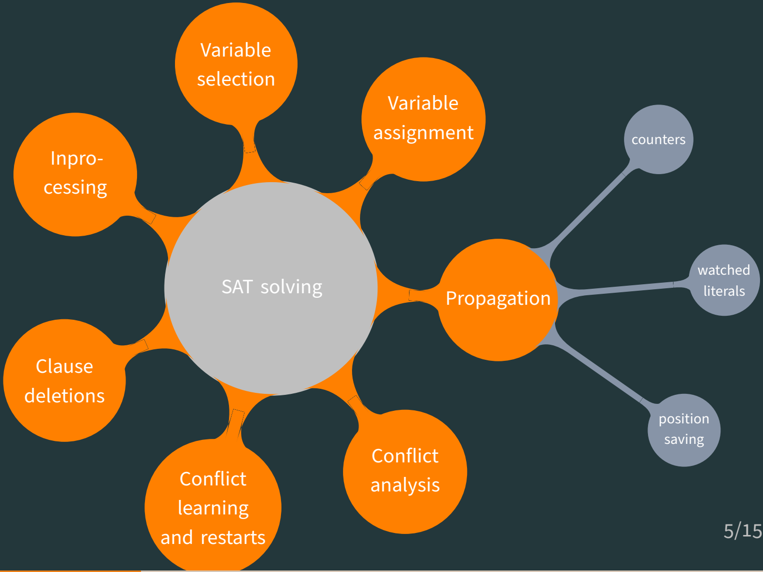
The features



The features



The features



How to write such a solver?

I have done all the hard work now it is easy:

implement it

How do I make sure that I don't get it wrong?

You will manage.

Your paper got it wrong too. How do I convince reviewers?

What are you talking about?

Approach 1: Options

Key idea:

```
if (opts.subsumelimited)
  check_limit = new_limit;
else
  check_limit = LONG_MAX;
```

This is the approach used in CaDiCaL (162 options!).

Drawbacks

- Impossible to test all combinations
- Impossible to understand which options depends on what? are glues and restarts linked if `--sat`?

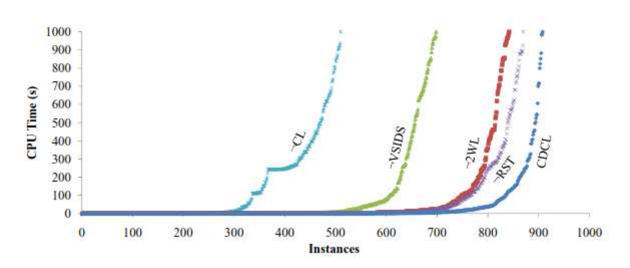
only limit are the options you pass to the solver!

- Not obvious to maintain

do these two features interact?

Drawbacks

How can you know that the ¬2WL is still using watched literals for propagations? counters are still updated during backtracking



A controversial paper

Approach 2: Compile-Time Options

Key idea: use the C preprocessor

```
#ifndef NWATCHES
  watch_clause (solver, learned);
#else
  connect_clause (solver, learned);
  count_clause (solver, learned);
#endif
  assign (solver, not_uip, learned, false);
```

This is the approach used in satch (49 options!).

Drawbacks

- Impossible to test all combinations but combination of 3 options covers a surprisingly high numbers of bugs
- Unclear how to make automatic testing we keep the implied options in a file (written by hand)
- ~~Impossible to understand which options depends on what?~~ checked by the compiler if glues not in clauses, cannot be used
- Not obvious to maintain and to program have I covered all paths now? why has my function 4 different control flows?

Approach 3: A new Solver each time

Key idea: write one solver. Write another and look at the diff!

```
+static std::vector<clause*>
+gather_reduce_candidates (void)
+{
+  std::vector<clause*> candidates;
+  mark_reason_clauses (true);
+  for (auto clause : clauses)
+    {
+      if (clause->reason)
+       continue;
       ...
+      candidates.push_back (clause);
+    }
```

This is the approach used during Armin’s lecture (4 diffs).

Drawbacks

- Impossible to test all combinations because only the written combination exists
- ~~Impossible to understand which options depends on what?~~ remove what you should not use
- Not obvious to maintain and to program how to efficiently backport fixes?

Discussion

Opinions?

How to measure effectiveness without state-of-the-art implementation?

We might (or not) have found a solution for code

Look, I know implementation is important

but I want to ease combining things

how do I do that?

Well, I am not sure

We need to measure something that is not time

Many different measures:

- solved instances (PAR-1) pure performance
- speed of solving (PAR-n) pure performance
- mems / ticks (roughly memory/cache accesses) skew heuristics to make the look better
- assume heuristics are no-cost ... but no implementation is

Conclusion

Conclusion

No conclusion… just work to do.