Decision Procedures for Artificial Intelligence

INF656L

Alexandre Chapoutot and Sergio Mover

ENSTA Paris

2022-2023

Course Outline

Main goals of the course:

- Know the main principles behind logical agents in AI and System verification;
- To be able to model decision problems using logical formulas;
- Know the solving algorithms for SAT and SMT solvers.

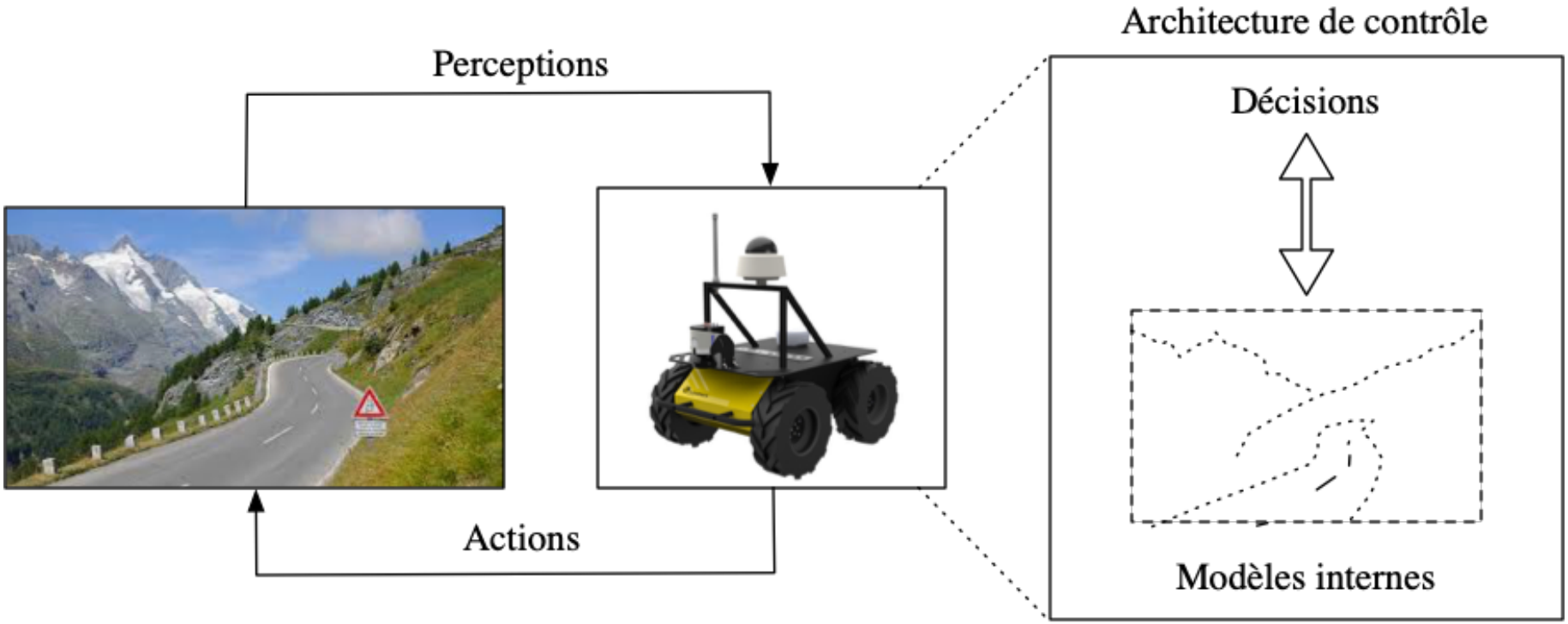
Remarks:

- consider unquantified logical formula
- consider exhaustive semantic methods

Intelligent agents

Examples

Mobile robot: Perception – Decision – Action
Software model agent: Belief – Desire – Intention



In this course

internal model is based on logical formula

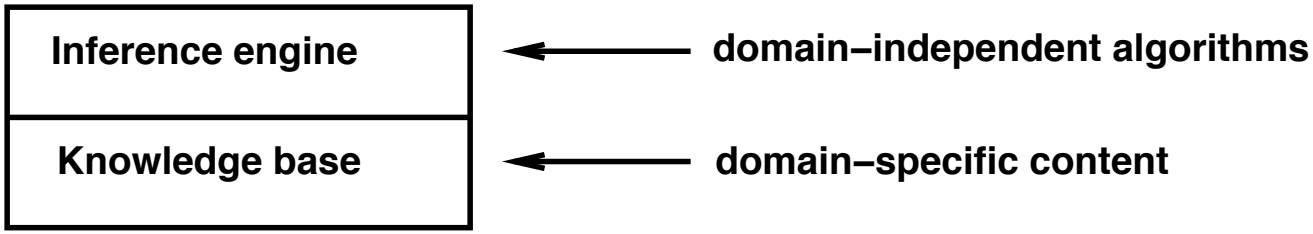
Motivations

We are interested in

- having an approach to *automate the reasoning*

in order to produce autonomous agent.

In particular, we will consider on *knowledge-based agent*. It is based on two elements:



- Knowledge base (set of sentences in a formal language) *i.e.* what the agent knows
- Inference engine *i.e.* a capability to deduce new information

Simple knowledge-based agent

function KB-AGENT(*percept*) **returns** an *action*
  static: KB, a knowledge base
    t, a counter, initially 0, indicating time

  TELL(KB, MAKE-PERCEPT-SENTENCE(*percept*, t))
  $action \leftarrow$ ASK(KB, MAKE-ACTION-QUERY(t))
  TELL(KB, MAKE-ACTION-SENTENCE($action$, t))
  $t \leftarrow t + 1$
  return *action*

The agent shall:

- Represent states (symbolic representation of the world) and actions
- Incorporate new perceptions
- Update internal representations of the world
- Deduce appropriate actions

Propositional logic will help to perform all these actions

Lecture 2: SAT solver algorithms

Entailment or Logical consequence

We denote by $\mathcal{M}(a)$ the set of all models of a *i.e.*

$$\mathcal{M}(a) = \{I : [\![a]\!]_{\mathsf{PL}}(I) = 1\}$$

We can express logical consequence or entailment denoted by

$$a \models b \qquad \text{iff} \qquad \mathcal{M}(a) \subseteq \mathcal{M}(b)$$

In other words, a entails b iff b is true in all interpretations which make a true.

Example

In arithmetic, we have

$$x = 0 \models xy = 0$$

For Knowledge-Based Agent

What we want is KB $\models g$ (g is goal for a robot)

Entailment as satisfiability

For any PL formula a and b, to show

$$a \models b$$

we can

- from a validity point of view

$$a \models b \quad \text{iff} \quad (a \implies b) \text{ is valid}$$

- from an unsatisfiability point of view (proof by contradiction)

$$a \models b \quad \text{iff} \quad (a \wedge \neg b) \text{ is unsatisfiable}$$

Process

Consider $a \wedge \neg b$ put is into CNF and apply a SAT solver

SAT as a Constraint Satisfaction Problem

Satisfiability problem (SAT)

Is there an interpretation in which a PL formula is true (*i.e.*, is there a model of a PL formula)?

- Each clause in CNF formula can be seen as **a constraint** that reduces the number of interpretations that can be models
- For example, $P \vee Q$ eliminates interpretations in which P is false and Q is false

Goal of SAT

Find a possible model that is satisfying all the constraints, *i.e.*, all clauses

Two kinds of algorithm to solve SAT

- A deterministic approach: DPLL and its extension CDCL
- A Probabilistic approach: WalkSAT

Main ideas in DPLL

Starting from a PL formula F in CNF, an algorithm explores the value space

- it builds incrementally a model M of F
- M is extended either
  - by **deduction** of the value a literal ℓ from M and F
  - or by **decision** of the value of a literal ℓ of F which is not in M yet
- If a **decision** produces a failure (*i.e.*, all literals are set to false ($\bot$)) the algorithm **backtrack** and inverse decision.

DPLL

DPLL stands for Davis–Putnam–Logemann–Loveland the names of the authors of the algorithm

DPLL - Definitions

Principle: Incremental building of a model M for the PL formula F in CNF

During the search,

- A **variable** can have the value $\top$ (true), $\bot$, (false), X not assigned
- A **clause** can be
  - **SAT** iff a least one of its literal is assigned to $\top$
  - **Unit** iff all its literals except one are assigned to $\bot$
  - **Conflict** iff all its literals are assigned to $\bot$
  - **Undef** iff it is not SAT, Unit, or Conflict
- A **CNF formula** is SAT if all its clauses are SAT

F_ℓ stands for the simplified formula obtained by replacing ℓ with its value, removing all clause with at least one $\top$ value and deleting all occurrences of $\bot$ from the reminding clauses.

Example

We want to check the satisfiability of the formula

$$(x_1 \vee x_2) \wedge (x_2 \vee \neg x_3 \vee x_4) \wedge (\neg x_1 \vee, \neg x_2) \wedge (\neg x_1 \vee \neg x_2 \vee \neg x_3) \wedge x_1$$

Operation	*Model*	*Formula*
		$\{x_1, x_2\}, \{x_2, \bar{x}_3, x_4\}, \{\bar{x}_1, \bar{x}_2\}, \{\bar{x}_1, \bar{x}_3, \bar{x}_4\}, \{x_1\}$
Propag x_1	x_1	$\{x_1, x_2\}, \{x_2, \bar{x}_3, x_4\}, \{\bar{x}_1, \bar{x}_2\}, \{\bar{x}_1, \bar{x}_3, \bar{x}_4\}, \{x_1\}$
Propag x_2	x_1, x_2	$\{x_1, x_2\}, \{x_2, \bar{x}_3, x_4\}, \{\bar{x}_1, \bar{x}_2\}, \{\bar{x}_1, \bar{x}_3, \bar{x}_4\}, \{x_1\}$
Decide x_3	x_1, x_2, x_3	$\{x_1, x_2\}, \{x_2, \bar{x}_3, x_4\}, \{\bar{x}_1, \bar{x}_2\}, \{\bar{x}_1, \bar{x}_3, \bar{x}_4\}, \{x_1\}$
Propag x_4	x_1, x_2, x_3, x_4	$\{x_1, x_2\}, \{x_2, \bar{x}_3, x_4\}, \{\bar{x}_1, \bar{x}_2\}, \{\bar{x}_1, \bar{x}_3, \bar{x}_4\}, \{x_1\}$
Undo x_3	x_1, x_2	$\{x_1, x_2\}, \{x_2, \bar{x}_3, x_4\}, \{\bar{x}_1, \bar{x}_2\}, \{\bar{x}_1, \bar{x}_3, \bar{x}_4\}, \{x_1\}$
Decide x_3	x_1, x_2, x_3	$\{x_1, x_2\}, \{x_2, \bar{x}_3, x_4\}, \{\bar{x}_1, \bar{x}_2\}, \{\bar{x}_1, \bar{x}_3, \bar{x}_4\}, \{x_1\}$

The formula is SAT with model $M = \{x_1, \neg x_2, \neg x_3\}$

DPLL Algorithm – pseudo code recursive form

Require: CNF formula F, empty model M
Ensure: UNSAT or SAT with model M

```
procedure DPLL(F, M)
    (F, M) ← UnitPropagate(F, M)
    if All clauses are true in M then
        return SAT
    end if
    if One clause is wrong in M then
        return UNSAT
    end if

    ℓ ← choose a literal not assigned in M
    if DPLL(F_ℓ, M ∪ {ℓ}) = SAT then
        return SAT
    end if
     return DPLL(F_¬ℓ, M ∪ {¬ℓ})
end procedure

procedure UnitProgagate(F, M)
    while F contains no empty clause but has a unit clause K with literal ℓ do
        F ← F_ℓ
        M ← M ∪ {ℓ}
    end whilereturn (F, M)
end procedure
```

Abstract DPLL[1]

It is based on a transition system where states are of the form

$$\text{fail or } M \parallel F$$

with

- F a PL formula in CNF
- M is a (partial) interpretation of the form

$$M = \ell, \ell, \ldots, \ell^d, \ldots, \ell$$

  with d an integer associated to a *decision level*.
- **Initial state** $\emptyset \parallel F$
- **Final states** two cases are considered
  - fail if F is unsatisfiable (UNSAT)
  - $M \parallel G$ with M of model of G and G is logically equivalent to F

[1] Nieuwenhuis et al. (2004)

Abstract DPLL algorithm

Basic DPLL

The original DPLL algorithm from 1960 is defined with the following rules

UnitProp	unit propagation
Decide	decision step
Fail	UNSAT
Backtrack	backtracking rule

iterated until a model M is built or a failure is obtained.

Abstract DPLL - correctness

Definitions

Irreducible state state from which one no transition can be taken

Execution sequence of transitions allowed by the rules from the initial state $\emptyset \parallel F$

Saturated execution execution finishing on one irreducible state

Theorem (Strong termination)

Every execution of DPLL algorithm is finite

Theorem (Correctness)

Every saturated execution of DPLL algorithm starting from $\emptyset \parallel F$ and finishing at $M \parallel F$ then $M \models F$

Theorem (Completeness)

If F is UNSAT then all saturated executions starting from $\emptyset \parallel F$ finish with fail state

Basic DPLL – Drawbacks

Drawback 1

When UNSAT is false not explanation are learnt and so all the work explaining why the current assignment is bad is thrown away.

Consequence current partial assignment producing UNSAT may be revisited more than once!

Drawback 2

Only chronological backtracking is performed (*i.e.*, undo the latest decision)

Consequence time is waste

Drawback 3

The choose of decision variable is almost done randomly without considering the structure of clauses.

Abstract DPLL – modern algorithms

New rules are now considered (mainly since 2000)

Learn	learning from conflicts
Forget	conflict clauses
Restart	
Backjump	non chronological backtracking

Modern DPLL – CDCL

Modern DPLL are also named CDCL which stands for "Conflict Driven Clause Learning".

Modern DPLL

It is defined with the following rules

- UnitProp
- Decide
- Fail
- Backjump
- Learn
- Forget
- Restart

iterated until a model M is built or a failure is obtained.

Simple CDCL Algorithm – pseudo code iterative form

Require: CNF formula F, empty model M
Ensure: UNSAT or SAT with model M
  procedure CDCL(F, M)
    $(F, M) \leftarrow$ UnitPropagate(F, M)
    if One clause is wrong in M **then**
      return `UNSAT`
    end if
    dl $\leftarrow 0$
    while not AllVariableAssigned(F, M) **do**
      $(\ell, \eta) \leftarrow$ choose a literal not assigned in M and its value
      dl $\leftarrow$ dl $+ 1$
      $M \leftarrow M \cup \{(\ell, \eta)\}$
      $(F, M) \leftarrow$ UnitPropagate(F, M)
      if One clause is wrong in M **then**
        $(\beta, ccl) =$ ConflictAnalysis(F, M)
        if $\beta < 0$ **then**
          return UNSAT
        else
          BackJump(F, M, β, ccl)
          dl $\leftarrow \beta$
        end if
      end if
    end while
     return SAT
  end procedure

CDCL - correctness

Theorem (Termination)

All executions in which

1. Learn/Forget rules are applied only a finite number of times
2. Restart is applied with an increasing periodicity

is finite

Theorem (Adequacy)

All executions starting from $\emptyset \parallel F$ and finishing in a irreducible state $M \parallel F$ is such that $M \models F$.

Theorem (Completeness)

If F is UNSAT then all execution starting from $\emptyset \parallel F$ and finishing at an irreducible state S then $S = \text{fail}$

CDCL - Clause learning

The main strength of CDCL algorithm is the conflict analysis and the clause learning steps. **Finding the minimal conflict clause is important.**

Setting a truth value to a variable

A the decision level d, assigning $\top$ to variable x_i is denoted by $x_i@d$ while assigning $\bot$ is denoted by $\bar{x}_i@d$

Antecedents

When the truth value of x_i is given by propagation through a unit clause c, c is named **antecedent** of x_i.

Implication graph

Nodes are the setting of truth value at the different decision level. Edges are the propagation labelled by antecedent clauses. Decisions are nodes without antecedents (roots)

CDCL - Conflict analysis

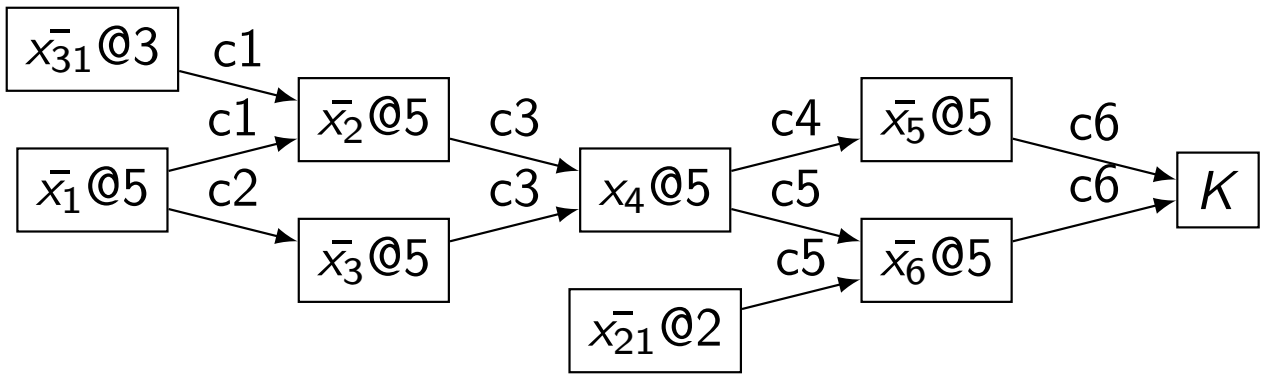
PL formula

$$F = \{c_1, c_2, c_3, c_4, c_5, c_6\}$$
$$c_1 = \{x_1, x_{31}, \neg x_2\}$$
$$c_2 = \{x_1, \neg x_3\}$$
$$c_3 = \{x_2, x_3, x_4\}$$
$$c_4 = \{\neg x_4, \neg x_5\}$$
$$c_5 = \{x_{21}, \neg x_4, \neg x_6\}$$
$$c_6 = \{x_5, x_6\}$$

Assume the following decisions:

- $\bar{x}_{21}@2$
- $\bar{x}_{31}@3$
- $\bar{x}_1@5$ (current decision level)

Hence `Propagate` deduces either x_5 and $\bar{x}_5$ or x_6 and $\bar{x}_6$ *i.e.* a conflict clause K



Conflict analysis

Starting from the conflict clause K, a new clause avoiding this conflict is built following antecedents in the implication graph.

$$K \xrightarrow{c_6} \{x_5, x_6\} \xrightarrow{c_4} \{\bar{x}_4, x_6\} \xrightarrow{c_5} \{\bar{x}_4, x_{21}\} \xrightarrow{c_3} \{x_2, x_3, x_{21}\} \xrightarrow{c_2} \{x_2, x_{21}, x_1\} \xrightarrow{c_1} \{x_1, x_{21}, x_{31}\}$$

Hence, we can add the conflict clause $\{x_1, x_{21}, x_{31}\}$ and the set of clauses. Moreover we go back to a decision level such that the conflict clause is unit to explore an other part of the value space.

Conflict analysis - cont'

Reduction of conflict clauses

- **Unique Implication Point** (UIP) is a **dominator node** in the implication graph
- There exists at least one UIP at each decision level (the decision itself)
- The conflict analysis can be stopped once the conflict clause contains the first UIP at the current decision level or of lower levels.

Example

In previous example, $x_4@5$ is the first UIP of decision level 5 so the conflict clause $\{x_{21}, \bar{x}_4\}$ can be considered instead of $\{x_{21}, x_{31}, x_1\}$.

Engineering stuff: detection of unit clause

Unit clauses play an important role in CDCL algorithm. It is mandatory to have efficient algorithms to detect them (Chaff 2000)

Main ideas of **Two Watch Literals** method

- To each variable x_i is associated the set of clauses C_{x_i} containing x_i
- For each $c \in C_{x_i}$, watch two literals ℓ and ℓ'
- If x_i is set to $\top$ or $\bot$, for each $c \in C_{x_i}$
  - If ℓ (resp. ℓ') is assigned to X, the clause is not unit
  - If ℓ (resp. ℓ') is assigned to $\top$ or $\bot$, look for an other literal ℓ'' assigned to X in the clause
    - ⋆ If ℓ'' exists then watched it instead of ℓ (or ℓ')
    - ⋆ Otherwise, c is unit so propagate ℓ (or ℓ')

Note a solver spends 80% of the execution using Propagate!

Engineering stuff: automatic decision heuristic

Method coming from solver Chaff (2000): **Variable State Independent Decaying Sum** (VSIDS) decision method

- An activity counter is associated to each variable (starting from 0)
- Increasing activity counter each time a variable is involved in a conflict clause
- Every n conflicts, divide activity counter by a given constant factor. Idea to focus in most recent conflicts
- The (unassigned) variable and polarity with the highest counter is chosen at each decision. In case of ties, choose randomly.

Local Search Algorithms

Local search method is a heuristic method for solving optimization problems.

In SAT context, try minimize the number of unsatisfied clauses or maximise the number of satisfied clauses.

WalkSAT

A very simple implementation of local search method for SAT.

Main ideas: Start from a randomly generated interpretation

- Pick randomly an unsatisfied clause
- Pick an atom to flip and randomly choose between two operations
  1. Randomly
  2. To minimize the number of unsatisfied clauses

WalkSAT

function WALKSAT(*clauses*, *p*, *max-flips*) **returns** a satisfying model or *failure*

inputs: *clauses*, a set of clauses in propositional logic

p, the probability of choosing to do a "random walk" move, typically around 0.5

max-flips, number of flips allowed before giving up

model $\leftarrow$ a random assignment of *true*/*false* to the symbols in *clauses*

for $i = 1$ **to** *max-flips* **do**

if *model* satisfies *clauses* **then return** *model*

clause $\leftarrow$ a randomly selected clause from *clauses* that is false in *model*

with probability *p* flip the value in *model* of a randomly selected symbol from *clause*

else flip whichever symbol in *clause* maximizes the number of satisfied clauses

return *failure*

WalkSAT algorithm

If the algorithm returns a failure after it tries max-flips times, what can we say?

1. The sentence is unsatisfiable
2. Nothing
3. The sentence is satisfiable

Remark: most useful when we expect a solution to exist

Hard satisfiability problems

Consider random 3-CNF sentences (at most 3 literals per clauses), for example

$$(\neg D \vee \neg B \vee C) \wedge (B \vee \neg A \vee \neg C) \wedge (\neg C \vee \neg B \vee E) \wedge (E \vee \neg D \vee B) \wedge (B \vee E \vee \neg C)$$

Notations: m number of clauses and n number of variables

Under constrained problems

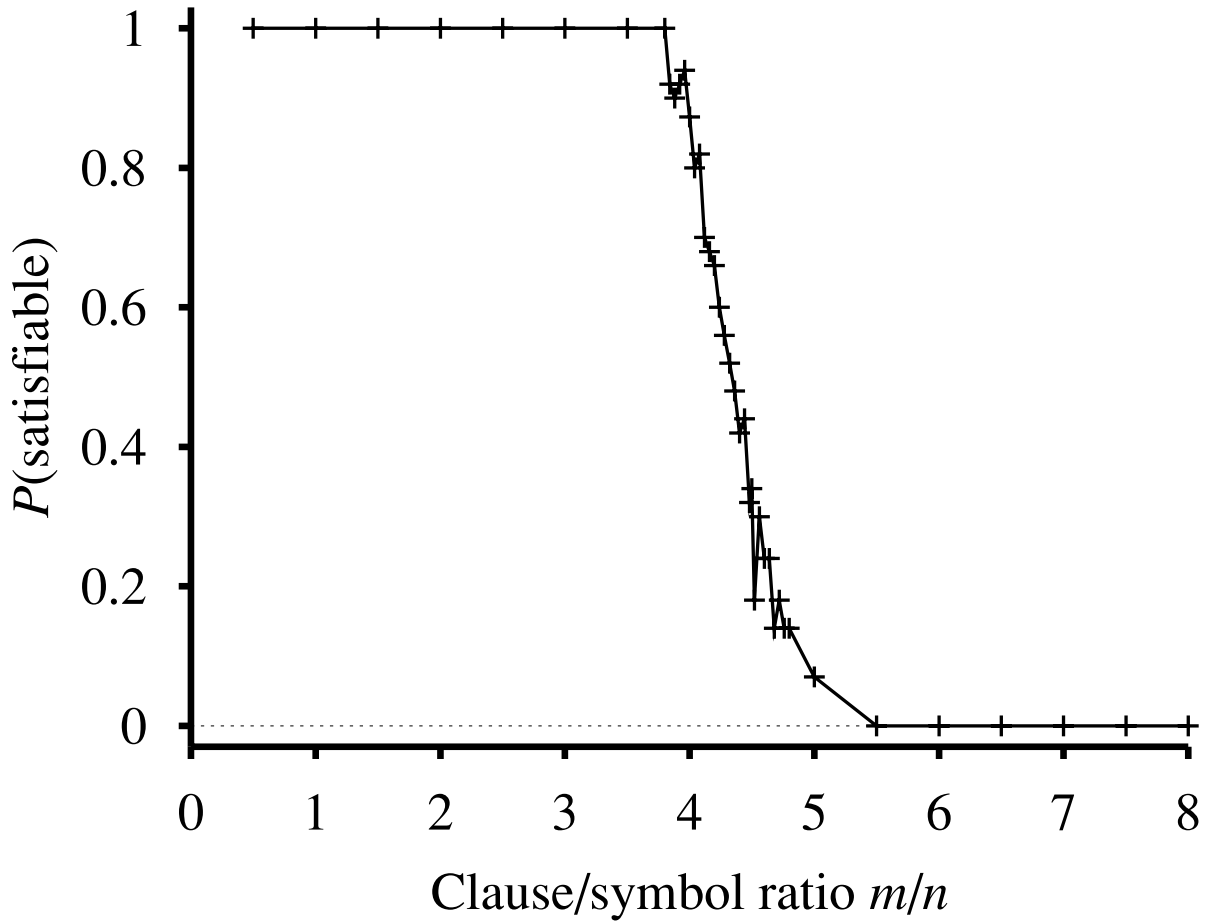
- relatively few clauses constraining the variables

For example, 16 over 32 possible interpretations are solutions for above problems (note: 2 random guesses will work on average)

What makes a SAT problem hard?

- Increase the number of clauses while keeping fixed the number of variables
  $\Longrightarrow$ problem is more constrained so fewer solutions

Experimental investigation of hard problems



random 3-CNF sentences, $n = 50$

Remark: Hard problems seem to cluster near $m/n = 4.3$

Examples of tools

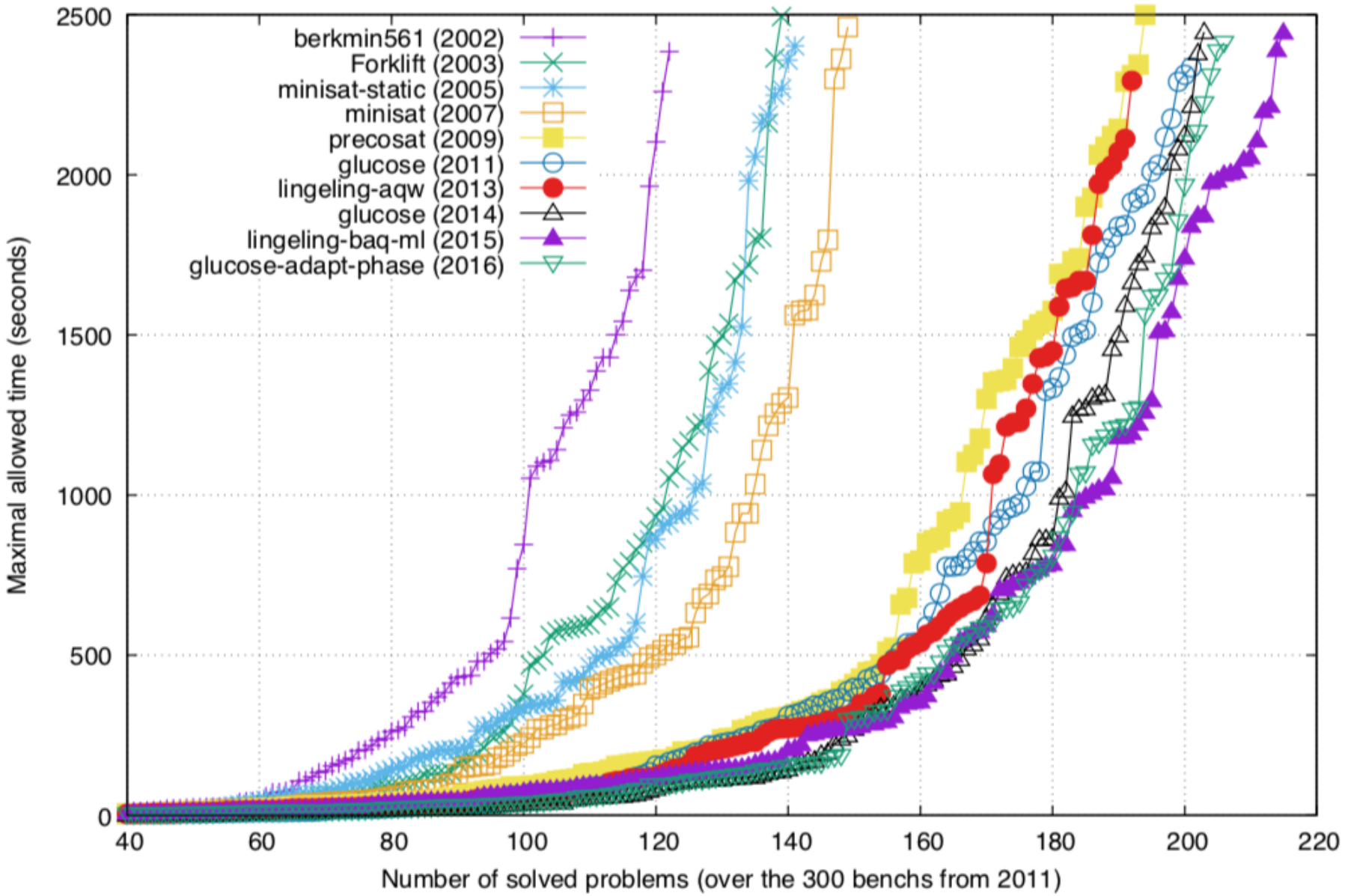
Modern tools use

- A preprocessing step to simplify the CNF before calling CDCL algorithm
- Parallel CDCL (multi-threaded)
- In-processing step to simplify the CNF during the CDCL algorithm

Some tools

- historical: chaff (2000), minisat (2004), picosat (2006)
- SAT4J (Java, embedded into Eclipse to solve library dependencies)
- Glucose 4.0 (C++, winner of several contests)
- Lingeling (C, winner of several contests)

Evolution of performance of SAT solvers



Applications of SAT

- Formal methods: Hardware model checking ; Software model checking;
- Artificial intelligence: Planning ; Knowledge representation ; Games (n-queens, sudoku, social golpher's, etc.)
- Bioinformatics: Haplotype inference Analysis of Genetic Regulatory Networks ; etc.
- Design automation: Equivalence checking ; Delay computation ; Fault diagnosis ; Noise analysis ; etc.
- Security: Cryptanalysis ; Inversion attacks on hash functions ; etc.