

Handout for lecture 1:

Course introduction and **SAT** basics

1. Course overview

2

Automated Reasoning, IMC009



dr Cynthia Kop
c.kop@cs.ru.nl



dr Sebastian Junges
sebastian.junges@ru.nl

RU, Maria Montessori Building, MM02.610 (Wednesday mornings)

3

Organization

- Course + examination
- Practical assignment
- Each 50%; each has to be at least 5.
- Practical assignment to be done in groups of at most 2.

- Deadline for the first part of the practical assignment:
27 October, 2024
- Deadline for the second part of the practical assignment:
20 December, 2024
- Thinking about solutions, and playing around with the tools **Z3**, **Yices** or **cvc5**, may start now!
- Example Python code and installation instructions are available on Brightspace.

2. Motivation

4

An open question

$$P = NP?$$

- $P \approx$ problems whose solution you can *find* easily
- $NP \approx$ problems whose solution you can *verify* easily

5

A well-known game: Sudoku

2				8	3	4	7	6
1			9					
6					4			
5		6					9	
8								4
	7					5		1
			4					9
					8			2
9	1	4	5	2				3

2	5	9	1	8	3	4	7	6
1	4	7	9	6	5	3	2	8
6	8	3	2	7	4	9	1	5
5	3	6	8	4	1	2	9	7
8	9	1	7	5	2	6	3	4
4	7	2	6	3	9	5	8	1
3	2	8	4	1	6	7	5	9
7	6	5	3	9	8	1	4	2
9	1	4	5	2	7	8	6	3

Sudoku is clearly in NP: we can *verify* the solution easily (also for an 16 by 16 and in general n^2 by n^2 Sudoku). But is it in P?

6

A thought experiment

What if: *finding* a solution is easy when you can *check* it?

Consequences for:

- **Encryption**

Given n and prime numbers p and q , you can easily *check* if $n = p * q$. If you can *find* p and q given only n , you would cause quite a lot of panic in the banking world (for instance).

- **Mathematics**

Proofs can be specified in a formal language (for instance in proof assistants like Coq or Isabelle), and automatically checked in a short time. If they could be *found* just (or almost) as easily, mathematicians would suddenly have a very different job.

- **Debugging**

In many cases, bugs can be easily confirmed by providing an input where the output is not as expected. If we could generate such inputs automatically, debugging tools would become much more powerful, and we may be able to formally prove program correctness in more cases.

- **Scheduling**

Many scheduling tasks that are currently done by hand or just by exhaustive search settle for a “good enough” solution – for example, train schedules (that minimise transit times / delays in one place causing problems in others), or school schedules that take students’ course choices and teachers’ preferences into account. As it is easy to confirm that a solution is suitable *and* better (according to a fixed measure) than a known solution, it becomes possible to find the best possible schedule.

- **Any kind of (automatic) problem solving**

For instance applications in medicine, AI, or more daily life issues like assigning students to project groups based on their preference, making fair pools for a sports tournament, ...

2				8	3	4	7	6
1			9					
6					4			
5		6					9	
8								4
	7					5		1
			4					9
					8			2
9	1	4	5	2				3

If finding a solution was as easy as checking it, you would expect to see new programming languages or libraries for common languages, dedicated to *specifying* a solution to a problem.

This could look something like this:

```

1 function clues {
2   1,1 ⇒ 2 ; 1,2 ⇒ 1 ; 1,3 ⇒ 6 ;
3   5,1 ⇒ 8 ; 6,1 ⇒ 3 ; 4,2 ⇒ 9 ; 6,3 ⇒ 4 ;
4   7,1 ⇒ 4 ; 8,1 ⇒ 7 ; 9,1 ⇒ 6 ;
5   1,4 ⇒ 5 ; 3,4 ⇒ 6 ; 1,5 ⇒ 8 ; 2,6 ⇒ 7 ;
6   8,4 ⇒ 9 ; 9,5 ⇒ 4 ; 7,6 ⇒ 5 ; 9,6 ⇒ 1 ;
7   1,9 ⇒ 9 ; 2,9 ⇒ 1 ; 3,9 ⇒ 4 ;
8   4,7 ⇒ 4 ; 6,8 ⇒ 8 ; 4,9 ⇒ 5 ; 5,9 ⇒ 2 ;
9   9,7 ⇒ 9 ; 9,8 ⇒ 2 ; 9,9 ⇒ 3 ;
10  _ = 0
11 }
12
13 =====
14
15 declare field[x,y] :: {1..9} for x ∈ {1..9}, y ∈ {1..9}
16
17 # every number occurs once in each row
18 ∀ i ∈ {1..9}. ∀ y ∈ {1..9}. ∃ x ∈ {1..9}. field[x,y] = i
19 # every number occurs once in each column
20 ∀ i ∈ {1..9}. ∀ x ∈ {1..9}. ∃ y ∈ {1..9}. field[x,y] = i
21 # every number occurs once in each block
22 ∀ i ∈ {1..9}. ∀ x1 ∈ {0..2}. ∀ y1 ∈ {0..2}. ∃ x2 ∈ {1..3}. ∃ y2 ∈ {1..3}.
23   field[3*x1+x2,3*y1+y2] = i
24
25 # the clues are satisfied
26 ∀ x ∈ {1..9}. ∀ y ∈ {1..9} with clues[x,y] != 0. field[x,y] = clues[x,y]
27
28 =====
29
30 for y := 1 to 9 do {
31   for x := 1 to 9 do {
32     print(field[x,y], ' ')
33     if x % 3 = 0 ∧ x != 9 then print('| ')
34   }
35   println()
36   if y % 3 = 0 ∧ y != 9 then println('-----+-----+-----')
37 }

```

Or rather than entirely new languages, we would see problem-solving libraries in common programming languages, such as Python.

```

1 from z3 import *
2
3 # 9x9 matrix of integer variables
4 X = [ [ Int("x_%s_%s" % (i+1, j+1)) for j in range(9) ] for i in range(9) ]
5
6 # each cell contains a value in {1, ..., 9}
7 cells_c = [ And(1 <= X[i][j], X[i][j] <= 9) for i in range(9) for j in range(9) ]
8
9 # each row contains a digit at most once
10 rows_c = [ Distinct(X[i]) for i in range(9) ]
11
12 # each column contains a digit at most once
13 cols_c = [ Distinct([ X[i][j] for i in range(9) ]) for j in range(9) ]
14
15 # each 3x3 square contains a digit at most once
16 sq_c = [ Distinct([ X[3*i0 + i][3*j0 + j] for i in range(3) for j in range(3) ])
17          for i0 in range(3) for j0 in range(3) ]
18
19 sudoku_c = cells_c + rows_c + cols_c + sq_c
20
21 # sudoku instance, we use '0' for empty cells
22 instance = ((0,0,0,0,9,4,0,3,0),
23             (0,0,0,5,1,0,0,0,7),
24             (0,8,9,0,0,0,0,4,0),
25             (0,0,0,0,0,0,2,0,8),
26             (0,6,0,2,0,1,0,5,0),
27             (1,0,2,0,0,0,0,0,0),
28             (0,7,0,0,0,0,5,2,0),
29             (9,0,0,0,6,5,0,0,0),
30             (0,4,0,9,7,0,0,0,0))
31
32 instance_c = [ If(instance[i][j] == 0, True, X[i][j] == instance[i][j])
33               for i in range(9) for j in range(9) ]
34
35 s = Solver()
36 s.add(sudoku_c + instance_c)
37 if s.check() == sat:
38     m = s.model()
39     r = [ [ m.evaluate(X[i][j]) for j in range(9) ] for i in range(9) ]
40     print_matrix(r)
41 else:
42     print ("failed to solve")

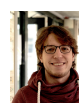
```

A spoiler for the rest of this lecture... Both of those actually exist, and work. :)

8

Topics

- How to use automatic solvers
- How do these automatic solvers work
- Automatic reasoning for predicate/equational logic
- Boolean functions, and having fun with SAT/SMT



3. Proposition logic

3.1 Lewis Carroll

9

Example: (free after Lewis Carroll)

1. Good-natured tenured professors are dynamic.
2. Grumpy student advisors play slot machines.
3. Smokers wearing a cap are phlegmatic.
4. Comical student advisors are professors.
5. Smoking untenured members are nervous.
6. Phlegmatic tenured members wearing caps are comical.
7. Student advisors who are not stock market players are scholars.
8. Relaxed student advisors are creative.
9. Creative scholars who do not play slot machines wear caps.
10. Nervous smokers play slot machines.
11. Student advisors who play slot machines do not smoke.
12. Creative good-natured stock market players wear caps.
13. Therefore no student advisor is smoking.

10

Is it true that claim 13 can be concluded from statements 1 until 12?

We want to determine this fully automatically:

- Translate all statements and the desired conclusion to a formal description.
- Apply some computer program with this formal description as input that decides fully automatically whether the conclusion is valid.

This is a step further than verifying a given human reasoning as done when studying proof assistants like Coq or Isabelle, because the human doesn't come into it other than translating the claims from human language to a formal language.

To formalise the statements, the first step is giving *names* to every “basic” notion that we want to formalise.

Next all claims (both the premisses and the desired conclusion) should be transformed to *Boolean formulas* containing these names as variables.

11

Naming basic notions

name	meaning	opposite
<i>A</i>	good-natured	grumpy
<i>B</i>	tenured	
<i>C</i>	professor	
<i>D</i>	dynamic	phlegmatic
<i>E</i>	wearing a cap	
<i>F</i>	smoke	
<i>G</i>	comical	
<i>H</i>	relaxed	nervous
<i>I</i>	play stock market	
<i>J</i>	scholar	
<i>K</i>	creative	
<i>L</i>	plays slot machine	
<i>M</i>	student advisor	

Using these names all claims can be transformed to a **proposition** over A – M , i.e., an expression composed from these letters and the boolean connectives \neg , \vee , \wedge , \rightarrow and \leftrightarrow .

12

Transforming *claims* to *propositions*

The claim

good-natured tenured professors are dynamic

yields:

$$(A \wedge B \wedge C) \rightarrow D$$

The claim

grumpy student advisors play slot machines.

yields:

$$(\neg A \wedge M) \rightarrow L$$

The claim

smoking untenured members are nervous.

yields:

$$(F \wedge \neg B) \rightarrow \neg H$$

Translating the full puzzle

1. $(A \wedge B \wedge C) \rightarrow D$
2. $(\neg A \wedge M) \rightarrow L$
3. $(F \wedge E) \rightarrow \neg D$
4. $(G \wedge M) \rightarrow C$
5. $(F \wedge \neg B) \rightarrow \neg H$
6. $(\neg D \wedge B \wedge E) \rightarrow G$
7. $(\neg I \wedge M) \rightarrow J$
8. $(H \wedge M) \rightarrow K$
9. $(K \wedge J \wedge \neg L) \rightarrow E$
10. $(\neg H \wedge F) \rightarrow L$
11. $(L \wedge M) \rightarrow \neg F$
12. $(K \wedge A \wedge I) \rightarrow E$
13. Then $\neg(M \wedge F)$

Translating the full puzzle

Hence we want to prove automatically that

$$\begin{aligned}
 &(((A \wedge B \wedge C) \rightarrow D) \wedge \\
 &((\neg A \wedge M) \rightarrow L) \wedge \\
 &((F \wedge E) \rightarrow \neg D) \wedge \\
 &((G \wedge M) \rightarrow C) \wedge \\
 &((F \wedge \neg B) \rightarrow \neg H) \wedge \\
 &((\neg D \wedge B \wedge E) \rightarrow G) \wedge \\
 &((\neg I \wedge M) \rightarrow J) \wedge \\
 &((H \wedge M) \rightarrow K) \wedge \\
 &((K \wedge J \wedge \neg L) \rightarrow E) \wedge \\
 &((\neg H \wedge F) \rightarrow L) \wedge \\
 &((L \wedge M) \rightarrow \neg F) \wedge \\
 &((K \wedge A \wedge I) \rightarrow E)) \rightarrow \neg(M \wedge F)
 \end{aligned}$$

is a tautology, meaning that for every valuation of A to M the value of this formula is *true*.

How can this be treated?

3.2 SAT

15

Truth tables

Idea: compute all possible valuations

A	B	C	D	E	F	G	H	I	J	K	L	M	result
0	0	0	0	0	0	0	0	0	0	0	0	0	1
0	0	0	0	0	0	0	0	0	0	0	0	1	1
0	0	0	0	0	0	0	0	0	0	0	1	0	1

We can enumerate all possible combinations of truth values for each of the variables, compute the overall value of the formula for each combination, and list it in the respective row. If all rows evaluate to *true*, then the formula is indeed a tautology. If a single row does not, then it is not.

Number of rows: 2^n

Here, n is the number of variables. In the example, $n = 13$, so $2^n = 8192$. While that doesn't fit on a slide, a computer can easily handle this number, so this approach is feasible.

However, in many applications we have $n > 1000$ and need different methods.

Observation: if φ always evaluates to *true*, then $\neg\varphi$ always evaluates to *false*.

Hence, checking whether a formula yields *false* for all valuations is as difficult as checking whether a formula yields *true* for all valuations.

Definition

A formula is **satisfiable** if it yields *true* for some valuations.

Hence, to see if a formula φ is a tautology, it suffices to check whether $\neg\varphi$ is satisfiable. If it is **unsatisfiable**, then φ is indeed a tautology.

16

SAT

A formula composed from boolean variables and the boolean connectives \neg , \vee , \wedge , \rightarrow and \leftrightarrow is called a **propositional formula**.

The problem to determine whether a given propositional formula is satisfiable is called **SAT(isfiability)**.

So the method of truth tables is a method for SAT.

However, the complexity of this method is always exponential in the number of variables, by which the method is unsuitable for formulas over many variables.

Many practical problems can be expressed as SAT-problems over hundreds or thousands of variables.

Hence we need other methods than truth tables.

3.3 The hardness of SAT

17

Complexity of algorithms

To be able to discuss the hardness of SAT we start by some remarks on **complexity** of algorithms.

- **(time) complexity:** the number of steps required for executing an algorithm
- **space complexity:** the amount of memory required for executing an algorithm

Basic observation: (time) complexity \geq space complexity

This follows because in one step, you can only allocate one new unit of space.

In order to abstract from details we consider orders of magnitude:

$$f(n) = O(g(n)).$$

This means: there exist constants c, N such that for all $n \geq N$: $f(n) \leq c * g(n)$. Essentially this should be read as: f does not grow faster than g .

We also have:

$$f(n) = \Omega(g(n)):$$

This means: there exist constants $c > 0, N$ such that for all $n \geq N$: $f(n) \geq c * g(n)$. Essentially this should be read as: f does not grow slower than g .

18

Typical use of big-O notation

Often $f(n)$ is the complexity of an algorithm depending on a number n , and g is a well-known function, like

- $g(n) = n$ (**linear**)
- $g(n) = n^2$ (**quadratic**)
- $g(n) = n^k$ for some k (**polynomial**)
- $g(n) = a^n$ for some $a > 1$ (**exponential**)

Roughly speaking:

- polynomial: still feasible for reasonably big values of n
- exponential: only feasible for very small values of n

Of course an algorithm operating on $O(n^{1000})$ is not going to be particularly feasible. But for large enough n , it is still better than an exponential algorithm – and with computers getting ever better, more and more polynomial algorithm become feasible, while exponential algorithms remain worse.

That being said, for **sufficiently small** n , exponential algorithms might just be good enough. For example, $1.1^{100} \approx 13780$, which is significantly less than 100^3 . So, in many practical cases, exponential algorithms may well suffice.

19

No polynomial algorithm is known for SAT.

That is: there is no algorithm:

- receiving an arbitrary propositional formula as input
- that decides in all cases by execution whether this formula is satisfiable
- that requires no more than $c * n^k$ steps if $n > n_0$, where c, k, n_0 are values independent of the input and n is the size of the input

Not even if huge values are chosen for c, k, n_0 .

This can mean two different things:

- Such an algorithm exists, but it has not yet been found.
- Existence of such an algorithm is impossible.

Although it has not been proven, the latter is most likely.

20

P versus NP

P (polynomial time): the class of decision problems admitting a polynomial algorithm.

NP (non-deterministically polynomial): the class of decision problem which can be *verified* through a polynomial algorithm. That is, there is a notion of *certificate* such that:

- if the correct result for SAT is ‘no’, then no certificate exists
- if the correct result for SAT is ‘yes’, then a certificate exists, and there is a polynomial algorithm that can decide whether a candidate for a certificate is really a certificate.

Conjecture: SAT is not in **P**.

(But we can easily see that it is **NP**: the certificate is a satisfying sequence of boolean values for the variables.)

Conjecture: $\mathbf{P} \neq \mathbf{NP}$

It is true that $\mathbf{P} \subseteq \mathbf{NP}$. The conjecture that this inclusion is strict is one of the main open problems in theoretical computer science – even though it seems extremely likely to be true.

21

NP-Completeness

A decision problem \mathcal{A} in \mathbf{NP} is called **NP-complete** if from the assumption $\mathcal{A} \in \mathbf{P}$ can be concluded that $\mathbf{P} = \mathbf{NP}$, i.e., every other decision problem in \mathbf{NP} is in \mathbf{P} too.

So for NP-complete problems the existence of a polynomial algorithm is very unlikely.

It has been proven that SAT is NP-complete (1970); in fact this was the starting point of NP-completeness results.

Other NP-complete problems:

- Given a distance table between a set of places and a number n
is there a path containing all of these places having a total length $\leq n$?
(closely related to Traveling Salesman)
- Given an undirected graph
is there a cyclic path (hamiltonian circuit) containing every node exactly once?
- Given a number of packages, each having a weight, and given a target weight
can you divide these packages into two groups each having the same total weight?

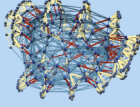
3.4 SAT solving

22

SAT solvers

Despite the inherent difficulty, many good **SAT solvers** exist!

There is an annual competition, where dozens of solvers compete.



SAT Competition 2022

Affiliated with the 25th International Conference on Theory and Applications of Satisfiability Testing taking place on the 2nd - 5th of August 2022 in Haifa, Israel.

- Overview
- Competition Tracks
- Solver Submission
- UNSAT Certificates
- StarExec Cluster
- AWS Cloud
- Benchmark Submission
- Downloads
- Results
- Organizers

Results

Main Track, Sequential Solvers

Solver	Score \star	Solved \star	Score SAT \star	Solved SAT \star	Score UNSAT \star	Solved UNSAT \star
vbs	2331.877677	329	161.348875	164	681.347329	156
Kissat_MAB-HyWak	3334.222364	290	1779.628623	144	1550.213919	146
kissat_inc	3351.932303	290	1769.021467	145	1606.754107	145
kissat_pre	3380.168998	288	1852.654662	143	1591.232273	145
ekissat-mab-db-v1	3402.947269	287	1888.567822	142	1611.862210	145
Kissat_MAB_MOSS	3404.378146	288	1967.836812	141	1532.218086	147
Kissat_MAB_UCB	3410.286408	287	1889.183907	140	1524.925058	147
kissat-mab-gb	3431.166946	285	2003.943832	139	1562.948653	146
Kissat_MAB_ESA	3446.109112	284	2058.191244	137	1544.232569	147
ekissat-mab-gb-db	3488.459816	283	2032.167165	139	1680.182765	144
ekissat-mab-db-v2	3525.053222	282	2216.387135	136	1580.344863	146
SeqFROST-ERE-AI	3534.520771	282	1715.505564	144	2131.188435	138
kissat-sc2022-bulky	3575.940505	282	2443.728270	133	1471.825422	149
SeqFROST-NoExtend	3578.233229	280	1709.416537	144	2249.672946	136
CadICAL-watchsat-fo	3593.382356	279	2074.628666	138	1904.576535	141
cadical_rel_Scovel	3614.810746	279	2013.706110	140	2023.567286	139
CadICAL_DVCL_V1	3619.004117	279	2027.659477	139	2019.650593	140
LSTeX_CadICAL	3631.170718	279	2149.418278	138	1922.844164	141
kissat-eli-v3	3636.500236	279	2120.065119	137	1967.368044	142

Typical benchmarks have thousands of variables.

Many industrial problems are **within reach**.

23

Topics in the first month

- Week 1 and 2: how to use *SAT* solvers



- Week 3 and 4: how do *SAT* solvers actually work



4. Using SAT solvers

4.1 Input format

24

SAT solver input format

A **conjunctive normal form (CNF)** is a conjunction of clauses.

A **clause** is a disjunction of literals.

A **literal** is either a variable or the negation of a variable.

Hence a CNF is of the shape

$$\bigwedge_i (\bigvee_j \ell_{ij})$$

where ℓ_{ij} are literals.

Here $\bigwedge_{i=1}^n A_i$ is an abbreviation for $A_1 \wedge A_2 \wedge A_3 \wedge \cdots \wedge A_n$

Proposition formulas can be transformed into CNF format with **linear** overhead (using the Tseitin transformation).

25

Example input file

```
p cnf 42437 150699
-4902 4903 0
-4904 4905 0
-4908 4909 0
-11 4911 0
-11 -12 4910 0
...
```

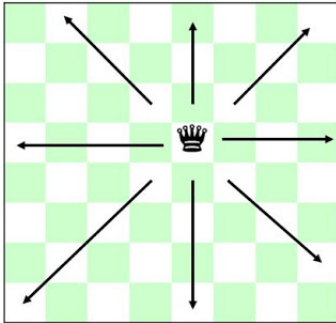
- 42437 literals (x_1 to x_{42437})
- 150699 clauses
- all lines end with 0
- minus is used for negated literals
- first clause: $\neg x_{4902} \vee x_{4903}$
- fifth clause: $\neg x_{11} \vee \neg x_{12} \vee x_{4910}$

(This is shown for reference – you are not expected to *use* this format.

4.2 Queens

26

Example: Eight Queens Problem



Can we put 8 queens on the chess board in such a way that no two may hit each other?

27

Solving the Eight Queens Problem

As usual in SAT/SMT: don't think about how to solve it, but only **specify** the problem. Your formula describes what properties the **solution** should satisfy.

Here it can be done in pure SAT: only boolean variables, no numbers, no inequalities.

For every position (y, x) on the board: boolean variable p_{yx} expresses whether there is a queen or not. We place $(1, 1)$ in the top left corner. Then, we can visualise our variables as follows:

p_{11}	p_{12}	p_{13}	p_{14}	p_{15}	p_{16}	p_{17}	p_{18}
p_{21}	p_{22}	p_{23}	p_{24}	p_{25}	p_{26}	p_{27}	p_{28}
p_{31}	p_{32}	p_{33}	p_{34}	p_{35}	p_{36}	p_{37}	p_{38}
p_{41}	p_{42}	p_{43}	p_{44}	p_{45}	p_{46}	p_{47}	p_{48}
p_{51}	p_{52}	p_{53}	p_{54}	p_{55}	p_{56}	p_{57}	p_{58}
p_{61}	p_{62}	p_{63}	p_{64}	p_{65}	p_{66}	p_{67}	p_{68}
p_{71}	p_{72}	p_{73}	p_{74}	p_{75}	p_{76}	p_{77}	p_{78}
p_{81}	p_{82}	p_{83}	p_{84}	p_{85}	p_{86}	p_{87}	p_{88}

Eight queens requirements

p_{11}	p_{12}	p_{13}	p_{14}	p_{15}	p_{16}	p_{17}	p_{18}
p_{21}	p_{22}	p_{23}	p_{24}	p_{25}	p_{26}	p_{27}	p_{28}
p_{31}	p_{32}	p_{33}	p_{34}	p_{35}	p_{36}	p_{37}	p_{38}
p_{41}	p_{42}	p_{43}	p_{44}	p_{45}	p_{46}	p_{47}	p_{48}
p_{51}	p_{52}	p_{53}	p_{54}	p_{55}	p_{56}	p_{57}	p_{58}
p_{61}	p_{62}	p_{63}	p_{64}	p_{65}	p_{66}	p_{67}	p_{68}
p_{71}	p_{72}	p_{73}	p_{74}	p_{75}	p_{76}	p_{77}	p_{78}
p_{81}	p_{82}	p_{83}	p_{84}	p_{85}	p_{86}	p_{87}	p_{88}

Note: If p_{yx} and $p_{y'x'}$ are in the same row: $y = y'$

Eight queens requirements

Row y :

$$p_{y1}, p_{y2}, p_{y3}, p_{y4}, p_{y5}, p_{y6}, p_{y7}, p_{y8}$$

At **least** one queen on row y :

$$p_{y1} \vee p_{y2} \vee p_{y3} \vee p_{y4} \vee p_{y5} \vee p_{y6} \vee p_{y7} \vee p_{y8}$$

Shorthand:

$$\bigvee_{j=1}^8 p_{ij}$$

Eight queens requirements

Row y :

$$p_{y1}, p_{y2}, p_{y3}, p_{y4}, p_{y5}, p_{y6}, p_{y7}, p_{y8}$$

At **most** one queen on row y ?

That is: for every $i < j$ not both p_{yi} and p_{yj} are true.

So $\neg p_{yi} \vee \neg p_{yj}$ for all $i < j$.

$$\bigwedge_{0 < i < j \leq 8} (\neg p_{yi} \vee \neg p_{yj})$$

31

Eight queens requirements

p_{11}	p_{12}	p_{13}	p_{14}	p_{15}	p_{16}	p_{17}	p_{18}
p_{21}	p_{22}	p_{23}	p_{24}	p_{25}	p_{26}	p_{27}	p_{28}
p_{31}	p_{32}	p_{33}	p_{34}	p_{35}	p_{36}	p_{37}	p_{38}
p_{41}	p_{42}	p_{43}	p_{44}	p_{45}	p_{46}	p_{47}	p_{48}
p_{51}	p_{52}	p_{53}	p_{54}	p_{55}	p_{56}	p_{57}	p_{58}
p_{61}	p_{62}	p_{63}	p_{64}	p_{65}	p_{66}	p_{67}	p_{68}
p_{71}	p_{72}	p_{73}	p_{74}	p_{75}	p_{76}	p_{77}	p_{78}
p_{81}	p_{82}	p_{83}	p_{84}	p_{85}	p_{86}	p_{87}	p_{88}

Column requirements for p_{yx} : the same as for rows with y, x swapped.

32

Eight queens requirements

Requirements until now:

At least one queen on every row:

$$\bigwedge_{y=1}^8 \bigvee_{x=1}^8 p_{yx}$$

At most one queen on every row:

$$\bigwedge_{y=1}^8 \bigwedge_{0 < i < j \leq 8} (\neg p_{yi} \vee \neg p_{yj})$$

33

Eight queens requirements

And similar for the columns:

At least one queen on every column:

$$\bigwedge_{x=1}^8 \bigvee_{y=1}^8 p_{yx}$$

At most one queen on every column:

$$\bigwedge_{x=1}^8 \bigwedge_{0 < i < j \leq 8} (\neg p_{ix} \vee \neg p_{jx})$$

34

Eight queens requirements

There is at most one queen on every diagonal:

p_{11}	p_{12}	p_{13}	p_{14}	p_{15}	p_{16}	p_{17}	p_{18}
p_{21}	p_{22}	p_{23}	p_{24}	p_{25}	p_{26}	p_{27}	p_{28}
p_{31}	p_{32}	p_{33}	p_{34}	p_{35}	p_{36}	p_{37}	p_{38}
p_{41}	p_{42}	p_{43}	p_{44}	p_{45}	p_{46}	p_{47}	p_{48}
p_{51}	p_{52}	p_{53}	p_{54}	p_{55}	p_{56}	p_{57}	p_{58}
p_{61}	p_{62}	p_{63}	p_{64}	p_{65}	p_{66}	p_{67}	p_{68}
p_{71}	p_{72}	p_{73}	p_{74}	p_{75}	p_{76}	p_{77}	p_{78}
p_{81}	p_{82}	p_{83}	p_{84}	p_{85}	p_{86}	p_{87}	p_{88}

p_{yx} and $p_{y'x'}$ on such a diagonal

\iff

$$y + x = y' + x'$$

35

Eight queens requirements

The diagonal in the other direction:

p_{11}	p_{12}	p_{13}	p_{14}	p_{15}	p_{16}	p_{17}	p_{18}
p_{21}	p_{22}	p_{23}	p_{24}	p_{25}	p_{26}	p_{27}	p_{28}
p_{31}	p_{32}	p_{33}	p_{34}	p_{35}	p_{36}	p_{37}	p_{38}
p_{41}	p_{42}	p_{43}	p_{44}	p_{45}	p_{46}	p_{47}	p_{48}
p_{51}	p_{52}	p_{53}	p_{54}	p_{55}	p_{56}	p_{57}	p_{58}
p_{61}	p_{62}	p_{63}	p_{64}	p_{65}	p_{66}	p_{67}	p_{68}
p_{71}	p_{72}	p_{73}	p_{74}	p_{75}	p_{76}	p_{77}	p_{78}
p_{81}	p_{82}	p_{83}	p_{84}	p_{85}	p_{86}	p_{87}	p_{88}

p_{yx} and $p_{y'x'}$ on such a diagonal

\iff

$$y - x = y' - x'$$

36

Eight queens requirements

So for all y, x, y', x' with $(y, x) \neq (y', x')$ satisfying $y + x = y' + x'$ or $y - x = y' - x'$:

$$\neg p_{yx} \vee \neg p_{y'x'}$$

stating that if (y, x) and (y', x') are two distinct positions on a diagonal, no two queens are allowed.

We may restrict to $y < y'$, yielding

$$\bigwedge_{0 < y < y' \leq 8} \left(\bigwedge_{x, x': y+x=y'+x' \vee y-x=y'-x'} \neg p_{yx} \vee \neg p_{y'x'} \right)$$

Eight queens requirements

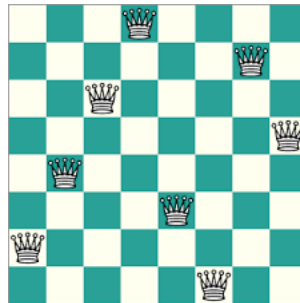
The total formula to express the 8 queens problem is thus:

$$\begin{aligned}
 & \bigwedge_{y=1}^8 \bigvee_{x=1}^8 p_{yx} \wedge \\
 & \bigwedge_{y=1}^8 \bigwedge_{0 < i < j \leq 8} (\neg p_{yi} \vee \neg p_{yj}) \wedge \\
 & \bigwedge_{x=1}^8 \bigvee_{y=1}^8 p_{yx} \wedge \\
 & \bigwedge_{x=1}^8 \bigwedge_{0 < i < j \leq 8} (\neg p_{ix} \vee \neg p_{jx}) \wedge \\
 & \bigwedge_{0 < y < y' \leq 8} \left(\bigwedge_{x, x' \text{ with } y+x=y'+x' \vee y-x=y'-x'} \neg p_{yx} \vee \neg p_{y'x'} \right)
 \end{aligned}$$

Conclusion

The resulting formula

- looks complicated, but is easily generated in SAT/SMT syntax by some *for* loops
- consists of 740 clauses
- is easily solved by current SAT solvers; resulting true values easily give the queen positions.



Generalising the Eight Queens problem

To find all 92 solutions, we use the following trick:

Find a solution, then add its negation to the formula. Repeat this until the formula is unsatisfiable.

The problem can be generalised to n queens on $n \times n$ board.

For $n = 100$ Z3 finds a satisfying assignment of the 50Mb formula within 10 seconds.

The same approach is applicable to extending a partially filled board, which is an NP-complete problem.

40

Optimising our encoding?

Note that optimisations are possible!

- there is **at least** one queen in every row
- there is **at most** one queen in every row
- there is **at least** one queen in every column
- there is **at most** one queen in every column

We can choose any one of those to leave out!

In practice: this often makes no difference, or even makes things worse.

(But: experimenting can be worthwhile.)

41

Eight Queens: conclusion

To conclude: we expressed a chess board problem in a pure SAT problem.

It is easy to generate this formula in the appropriate syntax by a small program.

Then a SAT solver immediately finds a solution.

[4.3 Sudoku](#)

42

Class exercise: Sudoku

2				8	3	4	7	6
1			9					
6					4			
5		6					9	
8								4
	7					5		1
			4					9
					8			2
9	1	4	5	2				3

Despite apparent numbers, this can be done in pure SAT: only boolean variables!

For every position (y, x) on the board and every number k : boolean variable $s_{y,x,k}$ expresses that the number at position (y, x) is k .

43

Requirements:

- each position has *exactly* one of the numbers 1–9
- in each row: every number in 1–9 occurs *exactly* once
- in each column: every number in 1–9 occurs *exactly* once:
- in each block: every number in 1–9 occurs *exactly* once

44

Requirements:

- each position has *at most* one of the numbers 1–9

$$\bigwedge_{y=1}^9 \bigwedge_{x=1}^9 \bigwedge_{1 \leq i < j \leq 9} \neg s_{y,x,i} \vee \neg s_{y,x,j}$$

- in each row: every number in 1–9 occurs *at least* once

$$\bigwedge_{y=1}^9 \bigwedge_{i=1}^9 \bigvee_{x=1}^9 s_{y,x,i}$$

- in each column: every number in 1–9 occurs *at least* once:

$$\bigwedge_{x=1}^9 \bigwedge_{i=1}^9 \bigvee_{y=1}^9 s_{y,x,i}$$

- in each block: every number in 1–9 occurs *at least* once

$$\bigwedge_{y_1=0}^2 \bigwedge_{x_1=0}^2 \bigwedge_{i=1}^9 \bigvee_{y_2=1}^3 \bigvee_{x_2=1}^3 s_{y_1*3+y_2, x_1*3+x_2, i}$$

5. Practical examples

5.1 Computer science research

45

An amazing example from academia

Prove termination of the system of three rules

$$aa \rightarrow bc, \quad bb \rightarrow ac, \quad cc \rightarrow ab$$

Solution: interpret a, b, c by matrices over natural numbers in such a way that by doing rewrite steps a particular entry in matrices always strictly decreases.

Since this entry is a natural number, this cannot go on forever.

$$\begin{pmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{pmatrix} * \begin{pmatrix} b_{11} & b_{12} & b_{13} & b_{14} \\ b_{21} & b_{22} & b_{23} & b_{24} \\ b_{31} & b_{32} & b_{33} & b_{34} \\ b_{41} & b_{42} & b_{43} & b_{44} \end{pmatrix} >_{ij} \begin{pmatrix} b_{11} & b_{12} & b_{13} & b_{14} \\ b_{21} & b_{22} & b_{23} & b_{24} \\ b_{31} & b_{32} & b_{33} & b_{34} \\ b_{41} & b_{42} & b_{43} & b_{44} \end{pmatrix} * \begin{pmatrix} c_{11} & c_{12} & c_{13} & c_{14} \\ c_{21} & c_{22} & c_{23} & c_{24} \\ c_{31} & c_{32} & c_{33} & c_{34} \\ c_{41} & c_{42} & c_{43} & c_{44} \end{pmatrix}$$

Restricting to binary encoded numbers of fixed size, and fixing matrix dimension (both ≈ 4) this was encoded in a SAT problem, and the obtained satisfying assignment was transformed to a formal proof of the above shape.

Until now all known proofs of this problem are variants of this idea, all found by SAT solving.

No human intuition is available.

5.2 Education

46

An example from education

In the course **Software Development Entrepreneurship** students work together in groups of 4–5.

- All members of a group must have at least 4 hours per week available for collaborative work. Yet, many students do not have too many free slots.
- In a group, there should always be at least one business student and 2 confident programmers. There also should not be more than 2 business students.

- Each group should contain at least one and at most two members who already have an idea.
- Some students have a friend they really prefer to work with. Some students have someone they *don't* want to work with.
- There may also be individual requirements (e.g., “I would really prefer to have another girl in my group”).

5.3 Sports

47

An example from sports:

We wish to make **fight assignments** for a martial arts tournament.

- There are nine participants and every fighter should have 4 matches.
- For every fighter: they never face the same opponent more than once.
- For every fighter: they have at least two matches rest between each fight.
- We have a **preliminary ranking**, indicating advanced, medium and beginner fighters. We want every fighter to have roughly the same total difficulty.
- If everyone wins as expected by their ranking, then the final ranking is the same as the initial ranking (at least for the top 4).

5.4 Industry

48

Example: verification of a microprocessor

Goal:

$$\neg(\text{specified behavior} \leftrightarrow \text{actual behavior})$$

is not satisfiable.

This can be expressed as a propositional formula, and proving that this formula is unsatisfiable implies that the microprocessor is correct with respect to its specification.

We will see methods for SAT that may be used for huge formulas over many variables, like these.

However, all known methods are worst case exponential.

6. Other

49

Extensions

Not all kinds of desired automated reasoning can be described in propositional logic.

In this course we will also consider other forms / extensions.

- **SMT:** Satisfiability Modulo Theories. We will particularly consider the theory of linear inequalities, in which apart from boolean variables also integer or real values may occur, and linear inequalities like $3a + 4b - c \leq 17$.
- **Predicate logic:** reasoning with \forall and \exists
all men are mortal
Socrates is a man
hence Socrates is mortal
- **Equational logic:** reasoning with equalities
given: $0 + x = x$ and
 $(x + 1) + y = (x + y) + 1$
conclude: $(0 + 1 + 1) + (0 + 1 + 1) = 0 + 1 + 1 + 1 + 1$.

50

Practical assignment

- All doable with plain SAT!
- But: that doesn't mean it's the best idea. Full capacity of Z3 may be used.
 - you can put in arbitrary formulas (not just CNF)
 - you can use integers with $+$, $*$, $=$ and $>$, \geq
 - otherwise, just use the ideas of this lecture: use Boolean logic (with a tiny bit extra) to describe the properties of the *solution* of the problem
- Advice: do **not** try to find solutions in esoteric Z3 abilities. Everything can easily be done with just boolean logic + integer arithmetic and a “distinct” statement.
- Do not write SAT input files by hand (other than as a **small** test). Generate them, or use a library.

- In the document you hand in, use clear notation, such as

$$\bigvee_{i=1}^n A_i \quad \text{for } A_1 \vee \dots \vee A_n$$

and

$$\bigwedge_{P(i)} A_i \quad \text{for the conjunction of } A_i \text{ for all } i \text{ that satisfy } P(i)$$

51

Practical exercise

					1	0		0	
3		6	5	6		3	3	2	
			6	6	5	3	3	3	
			4			5	4	4	2
5				4		4	4		2
							6	6	
	1	1			7		6	6	
				5		6		4	
	4	2		4		6			2
			3		4		4		

The blocks containing a number indicate the number of *black* squares surrounding that number (including the block itself).

Good practice: encode this using the capabilities of SMT.

You can use the Sudoku example on Brightspace as a basis to solve this puzzle using Python.

(Or you can immediately start on the practical assignment.)

52

Quiz

1. Indicate for each of the following statements whether it is **true**, **untrue** or **unknown**.
 - (a) SAT is in P.
 - (b) SAT is in NP.
 - (c) SAT is NP-complete.

(d) $P \neq NP$.

2. How can you use a SAT solver to prove that the statement

if $A \rightarrow B$ and $B \rightarrow C$ then $A \rightarrow C$

is a tautology?