# Symbolic Reasoning
# Tutorial: SAT and SMT encodings

1. Write down a SAT formula that will look for solutions to the 3-queens problem. Code this up in SMT-LIB 2, and solve it with Z3, to show that there are no solutions to 3-queens. (The fact that there are no solutions is obvious to a human, for such a simple configuration, of course.)

2. Use Z3 as a SAT solver to determine whether or not each of the following statements are true. When a statement is false, give an assignment that disproves the statement, from the model returned by Z3.

   (a) $a \to ((b \wedge c) \vee (d \wedge e))$ is logically equivalent to $\neg a \vee (\neg(b \wedge c) \to \neg(\neg d \vee \neg e))$

   (b) $a \to ((b \wedge c) \vee (d \vee e))$ is logically equivalent to $\neg a \vee ((\neg d \vee \neg e) \to (b \wedge c))$

   (c) $(a \vee b \vee c) \wedge \neg(a \wedge c) \wedge \neg(b \wedge c) \wedge (c \to a) \wedge (c \to b)$ is logically equivalent to $(a \vee b) \wedge \neg c$

   (d) $a \wedge (a \to b) \wedge (b \to c) \wedge (c \to d)$ is logically equivalent to $a \wedge b \wedge c \wedge d$

   (e) $\neg a \wedge (a \to b) \wedge (b \to c) \wedge (c \to d)$ is logically equivalent to $\neg a \wedge \neg b \wedge \neg c \wedge \neg d$

3. Use Z3, with the quantifier-free linear integer arithmetic theory (`QF_LIA`) to solve the following well-known "verbal arithmetic" puzzle.[1]

   Find distinct decimal digits, $S$, $E$, $N$, $D$, $M$, $O$, $R$, $Y$, that satisfy the following equation:

   $$\begin{array}{cccccc} & & S & E & N & D \\ + & & M & O & R & E \\ \hline = & M & O & N & E & Y \end{array}$$

   The first digit on a line may not be zero (e.g. $S$ cannot be zero).

   Once you have found a solution, use Z3 further to determine whether this solution is unique.

4. Adapt your solution to Question 3 so that it is expressed using the theory of bit-vectors, rather than integers. This involves changing uses of the `Int` type to use an appropriate bit-vector type, and similarly rewriting `Int` literals as bit-vector literals.

   What is the shortest bit-width you can use that suffices to find a solution?

5. A semiprime number is a number that is the product of two prime numbers. For example, 95 is semiprime as it is the product of 19 and 5. Semiprimes are useful in public key cryptography. The ability to efficiently factorise numbers, allowing the factors of semiprimes to be determined, would undermine cryptographic schemes that depend on such factorisation tasks being hard.

[1] According to Wikipedia, the puzzle was first published by H.E. Dudeney in *Strand Magazine* vol. 68 (July 1924), pp. 97 and 214.

Try using Z3 to work out the factors of the following semiprime numbers, by writing a separate SMT-LIB file for each case. Then, for each resulting factor, use Z3 to show that the factor is prime, by writing a separate SMT-LIB file for each factor. You will need to use the `QF_NIA` theory: quantifier-free *non-linear* arithmetic.

- 95
- 112,150,573
- 408,768,911
- 12,605,793,587

You will probably find that Z3 takes a long time to solve the last case (perhaps longer than you're willing to wait for!) Try giving it a leg-up, by telling it that the factors are each larger than 100,000 and smaller than 200,000.

Why is the theory of *non*-linear arithmetic required for solving this task?

The theory of non-linear integer arithmetic is undecidable. As a result, Z3 is not guaranteed to return a conclusive result for a query in this theory, no matter how much time it is given. Why does this mean that using an SMT solver to factorise numbers may be inadvisable?

6. The theory of bit-vectors is decidable. Why?

   For each of the semiprimes in Question 5, write an SMT-LIB query using the theory of bit-vectors to perform the factorisation task.

   **Hints:**

   - Bit-vector arithmetic wraps around if the result of an operation is larger than the maximum value that can be represented at a given bit-width. You will thus need to tell the solver to only consider factors whose product will *not* wrap around.
   - Treat bit-vector values as unsigned, by using unsigned versions of comparison and division operations.
   - You might find it useful to write bit-vector literals in hexadecimal; e.g. `#xDEADBEEF`.

7. The DIMACS CNF format is a standard way to present a CNF formula to a SAT solver. (I am not aware of an authoritative document that formalises this standard, but it is described in the web pages for the annual SAT competition.[2])

   A CNF problem is specified in DIMACS format as a text file. The file starts with a problem description: `p cnf` $n$ $m$, where `p` means "the description of the problem will now follow", `cnf` says "this is a CNF problem", $n$ indicates how many variables occur in the formula, and $m$ indicates how many clauses appear in the formula.

   The problem description is followed by $m$ clause descriptions. A clause description is a sequence of integers from the set $[-n, n] \setminus \{0\}$, followed by 0 (to indicate that the clause has been fully specified). A positive integer $i$ denotes the $i$th variable, i.e. a positive literal. A negative integer $-i$ denotes the negation of the $i$th variable, i.e. a negative literal.

[2] `http://www.satcompetition.org/2009/format-benchmarks2009.html`

Write a DIMACS CNF file for each of the following formulas, and use Z3 to determine whether they are satisfiable, giving a satisfying assignment in the event that a formula is SAT. To apply Z3 to a DIMACS CNF file, use the `-dimacs` option, followed by the name of the CNF file. If the formula is satisfiable, the model computed by Z3 is given as a series of literals, represented as positive and negative integers just like in the DIMACS format.

(a)

$$\begin{aligned}&(\neg x_2 \vee x_3 \vee x_4) \wedge (\neg x_3 \vee x_2) \wedge (\neg x_4 \vee x_2) \wedge (x_1) \\ \wedge\ &(\neg x_4 \vee x_8) \wedge (\neg x_4 \vee x_9) \wedge (\neg x_8 \vee \neg x_6 \vee x_4) \\ \wedge\ &(\neg x_1 \vee \neg x_5 \vee x_2) \wedge (x_5 \vee x_1) \wedge (\neg x_2 \vee x_1) \\ \wedge\ &(\neg x_3 \vee x_6) \wedge (\neg x_3 \vee x_7) \wedge (\neg x_6 \vee \neg x_7 \vee x_3)\end{aligned}$$

(b)

$$\begin{aligned}&(\neg x_6 \vee \neg x_{10}) \wedge (x_{10} \vee x_6) \wedge (x_9 \vee x_2) \wedge (x_4) \\ \wedge\ &(\neg x_4 \vee x_6 \vee x_3) \wedge (\neg x_6 \vee \neg x_4) \wedge (\neg x_3 \vee \neg x_4) \\ \wedge\ &(\neg x_3 \vee \neg x_7 \vee x_5) \wedge (x_7 \vee x_3) \wedge (\neg x_5 \vee x_3) \\ \wedge\ &(\neg x_7 \vee x_2 \vee x_1) \wedge (\neg x_2 \vee x_7) \wedge (\neg x_1 \vee x_7) \\ \wedge\ &(\neg x_1 \vee \neg x_8) \wedge (x_8 \vee x_1) \wedge (\neg x_5 \vee x_{12}) \\ \wedge\ &(\neg x_5 \vee x_{11}) \wedge (\neg x_{12} \vee \neg x_{11} \vee x_5) \\ \wedge\ &(\neg x_2 \vee \neg x_9)\end{aligned}$$

(c)

$$\begin{aligned}&(-x_7 \vee -x_6 \vee x_5) \wedge (-x_5 \vee x_7) \wedge (-x_4 \vee x_3 \vee x_6) \\ \wedge\ &(-x_3 \vee x_4) \wedge (-x_9 \vee -x_3 \vee x_6) \wedge (x_3 \vee x_9) \wedge (-x_2 \vee -x_5) \\ \wedge\ &(x_5 \vee x_2) \wedge (x_8) \wedge (-x_{10} \vee x_1) \wedge (-x_1 \vee x_2) \\ \wedge\ &(-x_1 \vee x_9) \wedge (x_7) \wedge (-x_8 \vee x_4) \wedge (x_{10} \vee x_9)\end{aligned}$$

8. A Sudoku is a 9x9 grid of numbers in the range $[1, 9]$, divided into nine 3x3 sub-grids, such that each row contains all of the numbers $[1, 9]$, each column contains all of the numbers $[1, 9]$, and each 3x3 sub-grid contains all of the numbers in the range $[1, 9]$.

   Here is an example Sudoku:

```
1 2 3 | 4 6 9 | 5 8 7
6 8 4 | 5 3 7 | 2 1 9
9 7 5 | 1 8 2 | 4 3 6
------+-------+------
7 4 1 | 6 2 5 | 3 9 8
8 6 2 | 9 1 3 | 7 5 4
3 5 9 | 8 7 4 | 6 2 1
------+-------+------
5 3 6 | 7 9 1 | 8 4 2
2 9 7 | 3 4 8 | 1 6 5
4 1 8 | 2 5 6 | 9 7 3
```

A Sudoku puzzle involves starting with an incomplete grid, containing at least one blank cell, and determining whether the blank cells can be filled in such that the end result is a Sudoku.

For example, the following incomplete grid (where 0 denotes a blank cell):

```
1 2 3 | 0 0 0 | 0 0 0
0 0 4 | 0 0 0 | 0 0 0
0 0 5 | 0 0 0 | 0 0 0
------+-------+------
0 0 0 | 6 0 0 | 0 0 0
0 0 0 | 0 0 0 | 0 0 0
0 0 0 | 0 7 0 | 0 0 0
------+-------+------
0 0 0 | 0 0 0 | 8 0 0
0 0 0 | 0 0 8 | 0 0 0
0 0 0 | 0 0 0 | 9 0 0
```

can be completed into the following Sudoku:

```
1 2 3 | 4 6 9 | 5 8 7
6 8 4 | 5 3 7 | 2 1 9
9 7 5 | 1 8 2 | 4 3 6
------+-------+------
7 4 1 | 6 2 5 | 3 9 8
8 6 2 | 9 1 3 | 7 5 4
3 5 9 | 8 7 4 | 6 2 1
------+-------+------
5 3 6 | 7 9 1 | 8 4 2
2 9 7 | 3 4 8 | 1 6 5
4 1 8 | 2 5 6 | 9 7 3
```

Your task is to build a SAT-based Sudoku solver, that will take an incomplete grid and either report that no completion of the grid is a Sudoku, or display a Sudoku that is a completion of the grid.

You are provided with a skeleton Python program for this task, `sudoku.py`. As input, the Python program takes the path to the Z3 executable, and a text file describing an incomplete Sudoku. The incomplete Sudoku file should have 9 lines, where each line contains 9 digits, separated by spaces. A digit 0 indicates that the value of that cell is unconstrained, otherwise the digit specifies the value that the cell is required to have.

The script also contains command line options to specify the temporary file to which a DIMACS CNF query should be written, an option to control the solver timeout, and an option to only emit the CNF query and not solve it (useful for debugging).

In `sat_sudokus` and `unsat_sudokus` you will find some example inputs for which it is and is not possible, respectively, to find a complete Sudoku that respects the given constraints.

Look for `TODO` in the file to see where you should write code (a) to emit DIMACS CNF encoding the Sudoku problem, and (b) to mine a Sudoku from the solution returned by the solver. Briefly study the rest of the Python program, which contains boiler plate code to parse the user input, invoke the solver, and check that the computed Sudoku really is valid and satisfies the input constraints.

Suggested encoding:

- Introduce 9 boolean variables for each grid cell (i.e., $9 \times 9 \times 9$ boolean variables in total). For a given grid cell, the $i$-th variable should record whether that grid cell has value $i$.
- Add constraints to ensure that each grid cell has an unambiguous value.
- Add constraints to ensure that each value in the range $[1, 9]$ appears in every row, column, and 3x3 sub-grid.
- Add constraints forcing grid cells to have particular values according to the incomplete grid that is given as input.