1

Quiz

1. Provide a SAT encoding that expresses that $a+b=c$, where a, b, c are all two-bit binary numbers.

   Answer: for a we use variables a_2a_1, for b we use b_2b_1 and for c we use c_2c_1.

   To use the exact formula of the lecture, we will use x_0, x_1, x_2 for the carries. Then we get:

$$\begin{array}{rl} \neg x_0 & \text{(no initial carry)} \\ \neg x_2 & \text{(no overflows)} \\ c_1 \leftrightarrow x_0 \leftrightarrow a_1 \leftrightarrow b_1 & (c_1 = (x_0 + a_1 + b_1)\%2) \\ c_2 \leftrightarrow x_1 \leftrightarrow a_2 \leftrightarrow b_2 & (c_2 = (x_1 + a_2 + b_2)\%2) \\ x_1 \leftrightarrow ((a_1 \wedge b_1) \vee (a_1 \wedge x_0) \vee (b_1 \wedge x_0)) & (x_1 = 1 \text{ if } x_0 + a_1 + b_1 \geq 2) \\ x_2 \leftrightarrow ((a_1 \wedge b_1) \vee (a_1 \wedge x_0) \vee (b_1 \wedge x_0)) & (x_2 = 1 \text{ if } x_1 + a_2 + b_2 \geq 2) \end{array}$$

   However, when writing these formulas by hand it is very natural to leave out x_0 as its value is fixed to 0 anyway (this variable was only included in the lecture to have fewer special cases, but that does not apply to the setting where $n = 0$). This could for instance give something like:

$$\begin{array}{rl} c_1 \leftrightarrow ((a_1 \vee b_1) \wedge (\neg a_1 \vee \neg b_1)) & (c_1 = (a_1 + a_2)\%2) \\ x_1 \leftrightarrow (a_1 \wedge b_1) & (x_1 = 1 \text{ if } a_1 + b_1 \geq 2) \\ c_2 \leftrightarrow x_1 \leftrightarrow a_2 \leftrightarrow b_2 & (c_2 = (x_1 + a_2 + b_2)\%2) \\ (\neg a_1 \vee \neg b_1) \vee (\neg a_1 \vee \neg x_1) \vee (\neg b_1 \vee \neg x_1) & \text{(no overflows)} \end{array}$$

   Writing out the CNF for the truth table of $a + b = c$ is also *technically* correct, but if you're doing these quizes to practice the material from the lecture, it's the wrong choice!

2. Provide a SAT encoding that expresses $a > b$ when $a, b \in \{0, \dots, 3\}$ are encoded as unary numbers.

   Answer: As $a, b \in \{0, \dots, 3\}$, we have variables a_1, a_2, a_3 and b_1, b_2, b_3. Here, intuitively a_i is 1 if $a \geq i$. Hence, $a > b$ is given by, for instance:

$$\neg b_3 \wedge (b_2 \rightarrow a_3) \wedge (b_1 \wedge a_2) \wedge a_1$$

   Since it was not required to supply a CNF formula, you could also for instance use:

$$(a_1 \wedge \neg b_1) \vee (a_2 \wedge \neg b_2) \vee (a_3 \wedge \neg b_3)$$

   Note: to know that a and b are valid unary numbers, it is separately required that $(a_3 \rightarrow a_2) \wedge (a_2 \rightarrow a_1) \wedge (b_3 \rightarrow b_2) \wedge (b_2 \rightarrow b_1)$. You do not need to include these requirements in your solution.

3. Why is it sometimes useful to use the unary encoding instead of binary?

   Answer: for *small* numbers (or: small ranges) the unary encoding often has fewer clauses than the binary encoding, especially because arithmetic and comparison for unary numbers can be done directly as a CNF, while the binary encoding requires the Tseitin transformation (this could also lead to the unary encoding having fewer variables for small enough problems). In addition, in practice the clauses are often a bit easier for a SAT-solver to handle (partially because there is more inherent redundancy).

   However, for larger ranges, the unary encoding is exponentially larger than the binary encoding, which does lead to the binary encoding being far more efficient there.

4. Use Tseitin's Transformation to give a CNF whose satisfiability is equivalent to:

   $$x \leftrightarrow ((y \wedge \neg x) \wedge (z \to w))$$

   Answer: We introduce new variables A, B, C, D corresponding to subformulas.

   $$\underbrace{x \leftrightarrow \underbrace{(\underbrace{(y \wedge \neg x)}_{C} \wedge \underbrace{(z \to w)}_{D})}_{B}}_{A}$$

   Then we write down the defining formulas for the subformulas, and also require the main formula:

   $$\begin{array}{cl} (A \leftrightarrow (x \leftrightarrow B)) & \wedge \\ (B \leftrightarrow (C \wedge D)) & \wedge \\ (C \leftrightarrow (y \wedge \neg x)) & \wedge \\ (D \leftrightarrow (z \to w)) & \wedge \\ A & \text{(don't forget this last part!)} \end{array}$$

   Then finally, we use the CNF form for each of the components. This gives:

   $$\begin{array}{llllllll} (\neg A \vee \neg x \vee B) & \wedge & (\neg A \vee x \vee \neg B) & \wedge & (A \vee \neg x \vee \neg B) & \wedge & (A \vee x \vee B) & \wedge \\ (\neg B \vee C) & \wedge & (\neg B \vee D) & \wedge & (\neg C \vee \neg D \vee B) & \wedge & & \\ (\neg C \vee y) & \wedge & (\neg C \vee \neg x) & \wedge & (\neg y \vee x \vee C) & \wedge & & \\ (\neg D \vee \neg z \vee w) & \wedge & (z \vee D) & \wedge & (\neg w \vee D) & \wedge & & \\ A & & & & & & & \end{array}$$

5. Why are pigeonhole formulas a good testcase for SAT solvers?

   Answer: these formulas are hard for SAT solvers, because they have no redundancy, and the removal of any clause changes the satisfiability of the whole formula. Thus, you can use *almost* the same formula to test both the satisfiability and unsatisfiability checking of a SAT solver. And while solving a pigeonhole formula will typically take exponential time, there is still a lot of room for variation between exponential factors; if your solver takes too long already on small pigeonhole formulas, that is an indication that you might not be handling worst-case input wisely. On the other hand, if your solver is very fast on pigeonhole formulas, and you have not specifically built automation for such formulas into your solver, then that might be an indication that there could be an error in your tool.