

1 Conflict-Driven Clause Learning (3+6 Points)

1.1 a) Small Formula (3 Points)

Simulate the CDCL algorithm by hand on the formula F . Select branching literals in the order x_1, x_2, x_3, \dots . Draw the implication graph once a conflict occurs, learn the 1-UIP clause and continue CDCL search with the variable implied by the asserting learned clause. If everything goes right you should only encounter a single conflict before arriving a satisfying assignment. To emphasize the structure, each variable x_i is just written as i and it's negation \bar{x}_i as $-i$. So for example, \bar{x}_4 is written as -4 and x_8 as 8 .

$$F = \{-1, -2, -4\}, \{-3, 6\}, \{-5, -6, 8\}, \{4, -8, 10\}, \tag{1}$$

$$\{7, 9\}, \{-2, -8, 9\}, \{-1, -9 - 10\}, \{-5, 8, 9\} \tag{2}$$

1.2 b) Larger Formula (6 Points)

Repeat the CDCL algorithm, now for the larger formula F' . Again, select branching literals again in the order x_1, x_2, x_3, \dots . Draw the implication graph at every conflict and give the learned 1-UIP clause. Provide the final assignment.

$$F' = \{1, 13\}, \{-1, -2, 14\}, \{3, 15\}, \{4, 16\}, \{-5, -3, 6\},$$

$$\{-5, -7\}, \{-6, 7, 8\}, \{-4, -8, -9\}, \{-1, 9, -10\},$$

$$\{9, 11, -14\}, \{10, -11, 12\}, \{-2, -11, -12\} \tag{3}$$

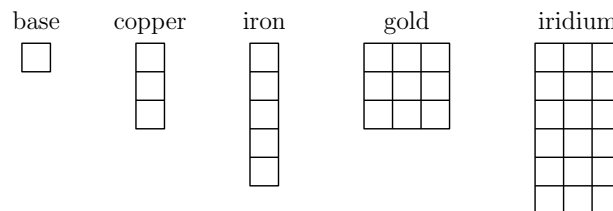
2 Model Counting with DPLL (3 Points)

How would you modify the DPLL algorithm to count the number of satisfying assignments of a given formula instead of just deciding satisfiability?

3 Stardew Valley Soil Tilling Problem (8(+7) Points)

Consider the well-known and industrially relevant *Stardew Valley Soil Tilling Problem* (SDVSTP), which we also talked about (in a slightly simpler form) in the 0th exercise.

We have a $w \times h$ 2D grid G of “soil” cells. Each of the $w \cdot h$ cells is in exactly one of the following states: *tilled* (T), *should be tilled* (+), or *must not be tilled* (-). We also define the following five *patterns* (corresponding to five increasing *charging levels* of a hoe):



We further define *action* $a = (p, r, x, y)$ as follows: Rotate the pattern $p \in \{\text{base, copper, iron, gold, iridium}\}$ by $r \in \{0^\circ, 90^\circ\}$ and then align its *upper left corner* with grid position (x, y) . Then a is *applicable* if the pattern is fully contained within G . Applying a changes all cells in G under the pattern to state T.

A *solution* for G is a set of applicable actions $A = \{a_1, a_2, \dots, a_k\}$ such that applying each action in A transforms G into the grid G' where all cells that were initially in state + are in state T and all cells that were initially in state - are still in state -. The objective is to find a solution that minimizes k .

Devise a SAT-based SDVSTP solver. Your program should take a single argument, the path to an input file in the straight-forward `.sdvstp` format, structured as in the example in Fig. 1 (left). The solver should output a line “s SOLUTION k”, where k is the minimum number of required actions, followed by k lines describing each individual action $(p, r, x, y) \in A$, as in Fig. 1 (center).

```

p sdvstp 8 5
+++++++
+++++++
++T--+++
+TT--+++
-----+

```

```

s SOLUTION 6
v gold    0 0 0
v gold    0 5 1
v iron    90 3 0
v copper  90 3 1
v base    0 0 3
v base    0 7 4

```

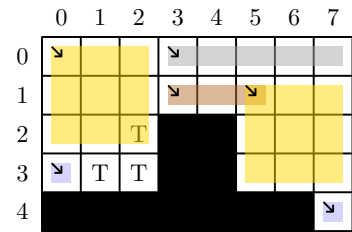


Figure 1: Left: Example `.sdvstp` input file specifying an 8×5 grid. Center: Possible corresponding output of the solver (columns aligned only for better readability). Right: illustration of the grid and the output solution, with the top-left “anchor point” of each pattern marked with an arrow.

You get 8 points for a correct SDVSTP solver that is able to solve easy instances within a few minutes. The overall most convincing submission(s) get(s) up to [seven bonus points](#).

Code skeleton: <https://github.com/satlecture/kit2025/blob/main/code/src/sdvstp/sdvstp.cc>

Some benchmark instances: <https://github.com/satlecture/kit2025/tree/main/exercises/sdvstp>