CS 228 — Assignment 1

Aradhana R (24b1006) Suhas Alladaboina (24b1009)

Question 1: SAT-Based Sudoku Solver

Variable Encoding

Let propositional variable $P(i, j, n)$ for $i, j, n \in \{1, \ldots, 9\}$ be true iff cell (i, j) has n.

Bijective encoding :

$$f(i, j, n) = 81 \cdot (i-1) + 9 \cdot (j-1) + (n-1) + 1, \quad f(i, j, n) \in \{1, \ldots, 729\}$$

Decoding :

$$f^{-1}(k) = (\ ((k-1)//81) + 1,\ \ ((k-1)//9)\%9 + 1,\ \ (k-1)\%9 + 1\ )$$

Encoding of Sudoku Rules

Our encoding consists of the following 4 rules and initial conditions:

C1 Each cell has exactly one number:
For each cell (i, j), it contains at least one number $n \in \{1, \ldots, 9\}$:

$$\psi_1 = \bigwedge_{1 \le i,j \le 9} \big(P(i, j, 1) \vee P(i, j, 2) \vee \cdots \vee P(i, j, 9)\big)$$

For each cell (i, j), it doesn't contain two numbers $a, b \in \{1, \ldots, 9\}$:

$$\psi_2 = \bigwedge_{1 \le i,j \le 9} \left(\bigwedge_{1 \le a < b \le 9} \big(\neg P(i, j, a) \vee \neg P(i, j, b)\big) \right)$$

C2 Each row has all the numbers:
For each row i, every number n should be in at least one of the cells in that row:

$$\psi_3 = \bigwedge_{1 \le i \le 9} \left(\bigwedge_{1 \le n \le 9} \Big(P(i, 1, n) \vee P(i, 2, n) \vee \cdots \vee P(i, 9, n)\Big) \right)$$

C3 Each column has all the numbers:
For each column j, every number n should be in at least one of the cells in that column:

$$\psi_4 = \bigwedge_{1 \le j \le 9} \left(\bigwedge_{1 \le n \le 9} \Big(P(1, j, n) \vee P(2, j, n) \vee \cdots \vee P(9, j, n)\Big) \right)$$

C4 Each block has all the numbers:
For every 3x3 block $b \in \{0, \ldots, 8\}$, every number n should be in at least one of the cells in that block:

$$\psi_4 = \bigwedge_{0 \le b < 9} \left(\bigwedge_{1 \le n \le 9} \Big(\bigvee_{1 \le i,j \le 3,} P(i + b//3,\ \ j + b\%3,\ \ n) \Big) \right)$$

C5 Initial Conditions:
For each pre-filled cell (i, j) with $n \neq 0$, we add the clause $P(i, j, n)$.

$$\psi_5 = \bigwedge_{\substack{(i,j)\in \texttt{Prefilled} \\ n\neq 0}} P(i, j, n)$$

Thus, the final conjunctive normal form (CNF) formula that is sent to the SAT solver is

$$\Psi = \psi_1 \wedge \psi_2 \wedge \psi_3 \wedge \psi_4 \wedge \psi_5$$

We then ask the SAT solver to determine satisfiability of Ψ. If the solver returns `SAT`, we then decode the satisfying assignment and store the numbers in their corresponding cells in the final output `ans`.

Approach

The approach is to encode the rules of Sudoku as propositional logic and then solve the resulting CNF. First, each cell (i, j) should have exactly one number, this is done using two sets of clauses, one ensuring that each cell has at least one number, and another preventing it from having two numbers. We then add row, column, and block constraints to make sure that every number appears at least once in each row, each column, and each of the 3×3 block. Finally, we add the initial conditions by adding clauses with a single variable corresponding to the pre-filled cells. Once all these constraints are combined into CNF form, it is given to a SAT solver. If a satisfying assignment is found, it is decoded back into a completed Sudoku puzzle.

Contributions Both team-mates (Aradhana R and Suhas Alladaboina) contributed equally to the modeling, CNF encodings, implementation. Write up for this question is the work of Aradhana.

Question 2: Grading Assignments Gone Wrong

Variable Encoding

Note: x is the x-coordinate as seen, i.e., it is the column number in the grid, and y is the row number.

1. **Player Encoding**
   Let propositional variable $P(x, y, t)$, for

   $$x \in \{0, \ldots, M-1\},\ y \in \{0, \ldots, N-1\},\ t \in \{0, \ldots, T\},$$

   be true iff coordinate (x, y) is occupied by the player at time t.

   The bijective encoding for the player is:

   $$\texttt{var\_player}(x, y, t) = N \cdot (T+1) \cdot x + (T+1) \cdot y + t + 1$$

2. **Box Encoding**
   Let propositional variable $B(b, x, y, t)$, for

   $$x \in \{0, \ldots, M-1\},\ y \in \{0, \ldots, N-1\},\ t \in \{0, \ldots, T\},\ b \in \{1, \ldots, \texttt{num\_boxes}\},$$

be true iff coordinate (x, y) is occupied by box b at time t.

The bijective encoding for boxes is:

$$\texttt{var\_box}(b, x, y, t) = N \cdot M \cdot (T+1) \cdot b + N \cdot (T+1) \cdot x + (T+1) \cdot y + t + 1$$

Decoding :

$$\texttt{inv\_box}(code+1) = \Big(code//\big(M \cdot N \cdot (T+1)\big), \quad \big(code//\big(N \cdot (T+1)\big)\big)\%M,$$
$$\big(code//(T+1)\big)\%N, \quad code\%(T+1) \Big)$$

Important: The player encoding is the same as the box encoding with box index 0 (Our box indices start from 1), during encoding we treat the player as a box with index 0 and for the same reason we don't have a inv_player function (this applies only for encoding and decoding purposes and not when writing CNF clauses).

Additional Functions and Member Variables

In addition to the functions and structure provided in the template file, we introduced the following member variables and helper methods in our implementation:

- **Member Variables**
  - `self.walls`: A list of wall coordinates, used to quickly check whether a given position is blocked.
  - `self.T`: Instead of storing T, we store $T+1$ directly, since this value is used repeatedly throughout the encoding and decoding steps.
- **Class Functions**
  - `inv_box(code)`: Given an integer code, this function returns the corresponding list $[b, x, y, t]$ that was encoded into that number using the bijective box encoding.
  - `is_valid(x,y)`: Returns `True` if (x, y) is a valid position inside the grid (i.e., within bounds and not a wall cell), and `False` otherwise.

Encoding of Sakoban Rules

Our encoding consists of the following rules and initial conditions:

C1 Initial Conditions:
For each box b, coordinate (b_x, b_y) is occupied by the box b at time $t = 0$:

$$\psi_1 = \bigwedge_{b=1}^{\texttt{num\_boxes}} B\big(b,\ x_b,\ y_b,\ 0\big)$$

The coordinate (p_x, p_y) is occupied by the player at time $t = 0$:

$$\psi_2 = P\big(x_p,\ y_p,\ 0\big)$$

C2 Player movement:

If the player is at (x, y) at time t, he can move to any of the 4 adjacent coordinates(if valid) or chose to stay still at time $t + 1$:

$$\psi_3 = \bigwedge_{x=0}^{M-1} \bigwedge_{y=0}^{N-1} \bigwedge_{t=0}^{T-1} \left(P(x,y,t) \;\rightarrow\; P(x,y,t+1) \;\vee \bigvee_{\substack{(i,j)\in \texttt{DIRS\_ME},\\ \texttt{is\_valid}(x+i,y+j)}} P(x+i,y+j,t+1) \right)$$

This clause can be converted to clause in CNF form by opening $a \rightarrow b$ into $\neg a \vee b$.

C3 Box movement:

If the position in which the player lands after moving contains a box, then we push the box to the cell in the same direction as the player's movement. If the box cannot be moved in that direction, either due to a wall or because of the boundary, we force the box to stay in its original position, thereby causing the player not to take that step (since we have a no-collision clause).

Note: While checking `is_valid()` we don't check if the position has a box in it, we just check for walls and boundary. This is fine since even if a different box was there at the poistion in which this box is getting pushed into the no collision clause will prevent the player from taking that step.

$$\psi_4 = \bigwedge_{x=0}^{M-1} \bigwedge_{y=0}^{N-1} \bigwedge_{t=0}^{T-1} \bigwedge_{\substack{(i,j)\in \texttt{DIRS\_ME},\\ \texttt{is\_valid}(x+i,y+j),\\ \texttt{is\_valid}(x+2i,y+2j)}} \bigwedge_{b\in\{1,\ldots,\texttt{num\_boxes}\}} \Big(P(x,y,t) \wedge P(x+i,y+j,t+1) \wedge B(b,x+i,y+j,t) \;\rightarrow B(b,x+2i,y+2j,t+1) \Big)$$

$$\psi_5 = \bigwedge_{x=0}^{M-1} \bigwedge_{y=0}^{N-1} \bigwedge_{t=0}^{T-1} \bigwedge_{\substack{(i,j)\in \texttt{DIRS\_ME},\\ \texttt{is\_valid}(x+i,y+j),\\ \texttt{!is\_valid}(x+2i,y+2j)}} \bigwedge_{b\in\{1,\ldots,\texttt{num\_boxes}\}} \Big(P(x,y,t) \wedge P(x+i,y+j,t+1) \wedge B(b,x+i,y+j,t) \;\rightarrow B(b,x+i,y+j,t+1) \Big)$$

C4 No overlap:

No two Boxes should be at the same position at the same time and no box should collide with the player at any instant.

$$\psi_6 = \bigwedge_{t=0}^{T} \bigwedge_{x=0}^{M-1} \bigwedge_{y=0}^{N-1} \bigwedge_{0\le b_1<b_2\le \texttt{num\_boxes}} \Big(B(b_1,x,y,t) \;\rightarrow\; \neg B(b_2,x,y,t) \Big)$$

Note: We don't have different clause for player-box collision and box-box collision because in our encoding player is just a box with index 0.

C5 Goal Conditions:

At time $t = T$ all the boxes should be in one of the goals.

$$\psi_7 = \bigwedge_{b=1}^{\texttt{num\_boxes}} \left(\bigvee_{g\in \texttt{Goals}} B\big(b,\; g_x,\; g_y,\; T\big) \right)$$

C6 One Box/Player only:

A player is in exactly one position at any given instant

$$\psi_8 = \bigwedge_{b=0}^{\texttt{num\_boxes}} \bigwedge_{t=0}^{T} \left(\bigvee_{x=0}^{M-1} \bigvee_{y=0}^{N-1} B(b,x,y,t) \right)$$

$$\psi_9 = \bigwedge_{b=0}^{\texttt{num\_boxes}} \bigwedge_{t=0}^{T} \bigwedge_{\substack{(x_1,y_1),(x_2,y_2) \\ \in\{0,\ldots,M-1\}\times\{0,\ldots,N-1\} \\ (x_1,y_1)<(x_2,y_2)}} \left(\neg B(b,x_1,y_1,t) \ \vee \ \neg B(b,x_2,y_2,t) \right)$$

C7 Boxes are mindless:

A Box doesnt move on it own unless pushed by player.

$$\psi_{10} = \bigwedge_{b=1}^{\texttt{num\_boxes}} \bigwedge_{x=0}^{M-1} \bigwedge_{y=0}^{N-1} \bigwedge_{t=0}^{T-1} \Big(B(b,x,y,t) \ \wedge \ \neg P(x,y,t+1) \ \rightarrow \ B(b,x,y,t+1) \Big)$$

Completeness: The written clauses are complete (as in they do not allow boxes to spawn in multiple places at a time), i.e. at a given time if positions of boxes and player are fixed, then at next instant the boxes move only if pushed, they are at a single position, and they move in the direction of push only if they are allowed to go there.

Thus, the final conjunctive normal form (CNF) formula that is sent to the SAT solver is

$$\Psi = \psi_1 \wedge \psi_2 \wedge \cdots \wedge \psi_{10}.$$

We then ask the SAT solver to determine satisfiability of Ψ. If the solver returns `SAT`, we proceed to decode the satisfying assignment .

Decoding of Returned Satisfying Assignment

From our encoding scheme of variables, we know that the propositional variables 1 to $M \cdot N \cdot (T+1)$ correspond to the player. Hence, when we read the returned satisfying assignment, we only consider variables in this range.

If a variable is positive in the satisfying assignment returned by the model, we decode its index back into the tuple $(0, x, y, t)$. The values of x, y, and t are obtained from the `inv_box` function and then we store the player's position at time t as the coordinate (x, y) in a dictionary. Then we read the dictionary from time $t = 0$ to $t = T$ and output the moves by finding the difference in coordinate from an instant to next.

Contributions Both team-mates (Aradhana R and Suhas Alladaboina) contributed equally to the modeling, CNF encodings, implementation. Write up for this question is the work of Suhas.

Question 3: NANDy's Logic Lab

Variable Encoding

Note: Nodes are indexed $i \in \{0, \ldots, n+G-1\}$ where $i \in \{0, \ldots, n-1\}$ are inputs nodes $(x_1, \ldots, x_n)$ and $i \in \{n, \ldots, n+G-1\}$ are NAND gate nodes $(G_1, \ldots, G_G)$. Assignments are indexed $t \in \{0, \ldots, 2^n-1\}$. For gate input, $g \in \{n, \ldots, n+G-1\}$, $k \in \{0, 1\}$ is itsinput position, and $j \in \{0, \ldots, g-1\}$ input node for k-th input.

1. **Node-Value Encoding**
   Let propositional variable $X(i,t)$, for

$$i \in \{0,\ldots,n{+}G{-}1\}, \quad t \in \{0,\ldots,2^n{-}1\},$$

   be true iff node i evaluates to 1 under assignment t.

   The bijective encoding for node values is:

$$\texttt{var\_node}(i,t) = (n{+}G)\cdot t + i + 1$$

2. **Gate-input Encoding**
   Let propositional variable $S(g,k,j)$, for

$$g \in \{n,\ldots,n{+}G{-}1\}, \quad k \in \{0,1\}, \quad j \in \{0,\ldots,g{-}1\},$$

   be true iff the k-th input of gate g is output of node j.

$$\texttt{var\_sel}(g,k,j) = 1 + 2^n\,(n{+}G) + (n{+}g{-}1)\,(g{-}n) + g\,k + j$$

Decoding: Instead of a closed-form inverse mapping, we iterate over possible input nodes to find the node wired to NAND gate input:

```
for j in range(gate):
    if model[self.var_gate(gate, 0, j) - 1] > 0:
        src0 = j
        break
```

Encoding of NAND only ciruit

Let $\texttt{table}[t] \in \{0,1\}$ denote the CNF's truth table on assignment t.

C1 Exactly-one source per gate-input:
Every gate-input must be driven by *exactly one* previous node. Similar to above questions we use atleast one and no two together.

$$\psi_1 = \bigwedge_{g=n}^{n+G-1} \bigwedge_{k\in\{0,1\}} \left(\bigvee_{j=0}^{g-1} S(g,k,j) \right)$$

$$\psi_2 = \bigwedge_{g=n}^{n+G-1} \bigwedge_{k\in\{0,1\}} \bigwedge_{0\le j_1<j_2<g} (\neg S(g,k,j_1) \vee \neg S(g,k,j_2))$$

C2 Fix input-node values for every assignment:
Set $X(i,t)$ to match the i-th bit of assignment t for all inputs:

$$\psi_3 = \bigwedge_{t=0}^{2^n-1} \bigwedge_{\substack{i=0 \\ \texttt{bit\_at}(t,i)=1}}^{n-1} X(i,t)$$

$$\psi_4 = \bigwedge_{t=0}^{2^n-1} \bigwedge_{\substack{i=0 \\ \texttt{bit\_at}(t,i)=0}}^{n-1} \neg X(i,t)$$

C3 Gate is NANDing the inputs:

$X(g,t)$ is the output of gate g on assignment t. For any chosen pair of sources (j_0, j_1),

$$S(g,0,j_0) \wedge S(g,1,j_1) \;\rightarrow\; \Big(X(g,t) \;\leftrightarrow\; \text{NAND}\left(X(j_0,t),\, X(j_1,t)\right)\Big).$$

$$\psi_5 = \bigwedge_{t=0}^{2^n-1} \bigwedge_{g=n}^{n+G-1} \bigwedge_{j_0=0}^{g-1} \bigwedge_{j_1=0}^{g-1} (\neg S(g,0,j_0) \vee \neg S(g,1,j_1) \vee X(g,t) \vee X(j_0,t))$$

$$\psi_6 = \bigwedge_{t=0}^{2^n-1} \bigwedge_{g=n}^{n+G-1} \bigwedge_{j_0=0}^{g-1} \bigwedge_{j_1=0}^{g-1} (\neg S(g,0,j_0) \vee \neg S(g,1,j_1) \vee X(g,t) \vee X(j_1,t))$$

$$\psi_7 = \bigwedge_{t=0}^{2^n-1} \bigwedge_{g=n}^{n+G-1} \bigwedge_{j_0=0}^{g-1} \bigwedge_{j_1=0}^{g-1} (\neg S(g,0,j_0) \vee \neg S(g,1,j_1) \vee \neg X(g,t) \vee \neg X(j_0,t) \vee \neg X(j_1,t))$$

C4 Correctness of the final output:

Let $g = n + G - 1$ be the last gate. We force its value to equal the CNF's truth table:

$$\psi_8 = \bigwedge_{\substack{t=0 \\ \texttt{table}[t]=1}}^{2^n-1} X(g,t)$$

$$\psi_9 = \bigwedge_{\substack{t=0 \\ \texttt{table}[t]=0}}^{2^n-1} \neg X(g,t)$$

Acyclicity: Because inputs for gate g may only be chosen from nodes $\{0, \ldots, g-1\}$, the graph is a DAG.

Final CNF: Thus, the final conjunctive normal form (CNF) formula that is sent to the SAT solver is

$$\Psi = \psi_1 \wedge \psi_2 \wedge \cdots \wedge \psi_9.$$

We then ask the SAT solver to determine satisfiability of Ψ. If the solver returns `SAT`, we proceed to decode the satisfying assignment .

Optimization of Gates: We search for the smallest G starting from 1 for which Ψ is satisfiable.

Decoding the SAT Model into a NAND Circuit

Let n be the number of inputs and G the number of NAND gates. Across all $T = 2^n$ input assignments, propositional variables

$$1, \ldots, T \cdot (n{+}G)$$

encode node values $X(i,t)$ (inputs and gate outputs). The remaining variables, starting at

$$\texttt{base} \;=\; 1 + T \cdot (n{+}G),$$

encode input node to gate g $S(g,k,j)$.

For circuit construction we only need the selectors(input node determinants). Given a satisfying assignment `model` from PySAT (a list of signed variable IDs), decode the wiring as follows

```
edges = {}
for gate in range(no_inputs, no_inputs + N):
    for j in range(gate):
            if model[self.var_gate(gate, 0, j)-1]>0:
                src0 = j
                break
    for j in range(gate):
            if model[self.var_gate(gate, 1, j)-1]>0:
                src1 = j
                break
    edges[gate] = (src0, src1)
```

Finally, print each gate by mapping indices to input node (x_{j+1} if $j < n$, else NAND gate node G_{j-n+1}), and report the output (the last gate):

```
for gate in range(no_inputs, no_inputs + N):
    a, b = edges[gate]
    node_a = f"x{a+1}" if a < no_inputs else f"G{a-no_inputs+1}"
    node_b = f"x{b+1}" if b < no_inputs else f"G{b-no_inputs+1}"
    print(f"G{gate-no_inputs+1} = NAND({node_a}, {node_b})")
print(f"OUTPUT = G{N}")
print(f"Total NAND gates used: {N}")
return True
```

Implementation Overview

Variable bijections. Node values and connector variables are bijectively mapped starting from 1.

```
# Node value variable: X(i, t)
def var_input(self, i, t):
    return 1 + (self.no_inputs + self.no_gates) * t + i

# Selector variable: S(g, k, j)
def var_gate(self, i, k, j):
    base = 1 + self.max_assignments * (self.no_inputs + self.no_gates)
    return base + (self.no_inputs + i - 1) * (i - self.no_inputs) + i * k + j
```

C1 Exactly-one source per (g,k):

```
    for gate in range(no_inputs, no_inputs + N):
        temp0 = []
        temp1 = []
        for i in range(gate):
            temp0.append(self.var_gate(gate, 0, i))
            temp1.append(self.var_gate(gate, 1, i))
        cnf.append(temp0)
        cnf.append(temp1)
        for i in range(gate):
            for j in range(i+1, gate):
                cnf.append([-temp0[i], -temp0[j]])
                cnf.append([-temp1[i], -temp1[j]])
```

C2 Fix inputs for every assignment t:

```
for t in range(2**no_inputs):
    for i in range(no_inputs):
        lit = self.var_input(i, t)
        if bit_at(t, i) == 1:
            cnf.append([lit])
        else:
            cnf.append([-lit])
```

C3 Gate is NANDing the inputs:

```
for t in range(2**no_inputs):
    for gate in range(no_inputs, no_inputs + N):
        y = self.var_input(gate, t)
        for j0 in range(gate):
            for j1 in range(gate):
                sel = (self.var_gate(gate, 0, j0), self.var_gate(gate, 1, j1))
                a = self.var_input(j0, t)
                b = self.var_input(j1, t)
                cnf.append([-sel[0], -sel[1], y, a])
                cnf.append([-sel[0], -sel[1], y, b])
                cnf.append([-sel[0], -sel[1], -y, -a, -b])
```

C4 Correctness of the final output:

```
truth_tablee = self.truth_table()
for i in range(len(truth_tablee)):
    truth_value = truth_tablee[i]
    y = self.var_input(no_inputs + N - 1, i)
    cnf.append([y] if truth_value == 1 else [-y])
```

Approach

For a given number of gates we model it as a DAG with nodes being either input nodes or NAND gates and egdes being the wirings. We perform a linear search on G starting from 1 (We thought about implementing a binary search but that would be wrong because if minimum gates is 7, it doesnt imply that we will be able to synthesize a circuit using exactly 8 gates). For each G, we build Ψ as above, call a SAT solver, and decode the model to a circuit. The first satisfiable G is the minimum gate count and the corresponding circuit.

Correctness according to Problem Statement

- **Soundness.** By C1 and acyclicity, every gate's inputs are well-defined. By C3, each gate's output equals NAND of its selected sources for every assignment t. By C4, the final node matches the target CNF on all t; hence the printed circuit computes the desired function.
- **Minimality.** Search returns the smallest G that makes Ψ satisfiable.

Contributions: Suhas came up with the overall approach, encoding rules and started the write-up. Aradhana continued implemention in code. Both collaboratively debugged and completed the final report.