Assignment #1

CS 228 (Logic in Computer Science)

TAs: Yuvraj Gupta & K. Gautam Siddharth

Due Date: August 23, 2025

General Instructions

- This is a graded assignment. This assignment contains 3 questions. You must complete each as instructed in its respective file. Q1 and Q2 are compulsory. Q3 is a bonus question.
- **Academic Integrity:** The institute's plagiarism policy applies strictly. Any instance of plagiarism will result atleast in zero marks for the assignment and may lead to harsher penalties, including grade drops and escalation to the DADAC. This applies equally to both giver and taker. Both teammates will be penalized equally, regardless of who plagiarised.
- **Do not use Large Language Models (LLMs) such as ChatGPT, Gemini, or Copilot to complete this assignment.** Any use will be considered a violation of academic integrity.
- You are only allowed to use the standard Python libraries and packages explicitly mentioned in each question.

Submission Instructions

- You should submit a single zip file named `<rollnumber1>_<rollnumber2>_assignment1.zip`, which on un-zipping should have the following directory structure:

```
<rollnumber1>_<rollnumber2>_assignment1/
|- q1.py
|- q2.py
|- q3.py
|- report.pdf
```

- The deadline is **August 23, 2025 EOD**. Submissions beyond this deadline will not be considered for grading.
- Submit your ZIP file on Moodle before the deadline. Follow the exact submission format as mentioned above. Any deviations from the submission format will invite atleast 30% penalty.
- Only one person from the team should submit the assignment.
- You should submit a SINGLE `report.pdf`. Report must be properly typeset in LaTeX. Handwritten reports will not be evaluated.
- Your `report.pdf` should clearly contain the individual contribution of each member.
- You should acknowledge any code snippets you have used, clearly stating the reference in the comments of your code.

Bonus Policy

Question 3 is a bonus question. The grading of this question will be done using the following policy:

- This question will be autograded. You either get full marks for this question or *zero* marks. No cribs will be entertained for any partial marks.
- No template file has been given for this question. You should clearly follow the input-output specifications. Failure to do so will result in your submission not being evaluated.
- The bonus marks you get for this question can be carried forward to other assignments. But the total marks (including bonus) across all the assignments will be capped at 20% of the course weightage.
- The total marks for this question is not disclosed and will depend on number of correct submissions.

Question 1. SAT-Based Sudoku Solver (30 Marks)

The objective of this task is to implement a Sudoku solver using propositional logic and a SAT solver. You will encode Sudoku rules as propositional formulas in Conjunctive Normal Form (CNF) and solve them using the PySAT Python library.

You are given a partially filled Sudoku grid, where `0` denotes an empty cell. Your tasks are:

1. Encode the rules of Sudoku (row, column, and subgrid constraints) as CNF clauses.
2. Complete the function `solve_sudoku(grid: List[List[int]]) -> List[List[int]]` using PySAT.
3. Extract the satisfying assignment returned by the SAT solver and reconstruct the solved Sudoku grid.

Files Provided

- `q1.py` – Boilerplate code with the function signature. **ONLY MODIFY THIS FILE PLEASE**
- `tester.py` – Script to test your solver on multiple puzzles.
- `pysat_tutorial.md` – Brief tutorial on using the PySAT toolkit.
- `testcases` – A file containing sample Sudoku puzzles.
- Make sure to not change anything in the function; only fill that function. Only your one file will be considered for grading.

Installation Instructions

Install PySAT using:

```
pip install python-sat[pblib,aiger]
```

Report Instructions

The report should clearly describe your encoding of the Sudoku rules as CNF formulas. Make sure to include the following points in your report:

- **Approach Overview:** A short summary of how you converted the Sudoku puzzle into a SAT problem using propositional variables and clauses.
- **Variable Encoding:** Describe what each variable corresponds to and what constraints you have encoded

Submission Instructions

- Upload the file `sudoku_solver.py` renamed as `q1.py`

Note: You must use an **encoding scheme** and a **SAT solver** to solve this question. Any alternative method will result in **zero marks**.

Question 2. Grading Assignments Gone Wrong (70 Marks)

With the crib session for the quiz paper imminent, inside the cramped TA office, seven TAs — Gautam, Malay, Jigyasa, Kritin, Dion, Thomas, and Abhilasha — have finally finished grading the assignments. But now there's a new problem: the graded papers are scattered chaotically across the office, piled in every corner and blocking the narrow corridors.

You've been appointed to bring order to the chaos — pushing these stacks of graded assignments (the **boxes**) onto the submission desks (the **goal cells**) before the students storm in, demanding their marks.

But standing in their way is Yuvraj, the self-appointed villain TA, who has barricaded parts of the office with tables and chairs (the **walls**). Every corridor is jammed, the office is a labyrinth of walls and paper towers, and there's barely room to breathe.

The corridors are painfully narrow, and every move counts. You can navigate up, down, left, or right through the few open spaces, pushing stacks forward but never pulling them back — every move costs you a precious minute. But here's the twist: you must finish within T minutes. Miss that deadline, and the students may never receive their grades.

Can you use your newfound SAT-solving powers to save the TAs and prove whether the puzzle can be solved in time?

You are given an $N \times M$ grid representing the TA office where:

- **Walls (#):** The player and the boxes cannot move into these cells.
- **Empty cells (.):** Free space where the player can move.
- **Player (P):** Represents the current position of the player.
- **Boxes (B):** Stacks of graded assignments that need to be placed.
- **Goals (G):** Submission desks where assignments must end up.

Rules of the Game

- **Valid Player Movements and Pushes:**
  - The player may move **up**, **down**, **left**, or **right**, provided the move remains within the grid boundaries and does not cause a collision with a wall or box.
  - A box may be pushed in any of the four directions if:
    1. The box is in the cell directly adjacent to the player in the chosen direction, and
    2. The cell immediately beyond the box, in the same direction, is within bounds and unoccupied (i.e., not a wall or another box).

    In such a case, if the player moves into the box's original position, then the box is shifted one cell in the chosen direction. For example, Suppose the player is at $(2, 2)$, and a box is directly above at $(1, 2)$. If the cell above the box, $(0, 2)$, is empty and within bounds, the player can move up into $(1, 2)$, pushing the box into $(0, 2)$. If $(0, 2)$ contains a wall, another box, or lies outside the grid, the push is not allowed.

```
Before:                    After moving up:
. . . . .                  . . B . .
. . B . .                  . . P . .
. . P . .                  . . . . .
. . . . .                  . . . . .
```

- **Goal condition:** At time step T, every box must be on a goal cell.
- **Time constraint:** The sequence of moves must not exceed T steps.

Your task is to complete the class implementation:

- Implement the `encode(self)` function to which returns the CNF formula that capture the rules as well as the other helper functions.
- Implement the `decode(model, encoder)` function to parse the satisfying assignment returned by the SAT solver, decode it into a valid sequence of moves, and return this as the final solution.

Input File Format

- The first line contains an integer T, denoting the maximum allowed moves (in minutes).
- Each of the next n lines contains m space-separated characters, representing the 2D grid.

Output Format

In the `decode(model, encoder)` function,

- If a solution exists, return the sequence of moves (U, D, L, R).
- If no solution exists i.e., it is UNSAT, return `-1`.

Example

Input:

```
3 5
P . .
. B .
. . G
```

Output:

```
RDLDR
```

Files Provided

- `q2.py` – Boilerplate code with the class and function signature.
- `tester.py` – Script to test your solver, **DO NOT MODIFY THIS FILE**
- `input` – A folder containing sample sokoban puzzles. (You are highly encouraged to run your code on custom testcases as evaluation will be based on hidden testcases).
- `output` – A folder containing expected outputs for each testcase.

Instructions to Run the Code

To run all test cases, use:

```
python3 tester.py
```

To execute the program on a specific input file, use:

```
python3 tester.py <testcase_1_path> <testcase_2_path> ...
```

Note: When creating custom test cases, the grid must be represented as a space-separated sequence of characters, with each character corresponding to a single grid cell.

Report Instructions

- Clearly describe the variable encoding scheme used — what each variable represents.
- Provide a detailed explanation of how the rules of the game were translated into CNF constraints. This includes movement rules, box-pushing logic, grid boundary conditions, and any additional constraints you considered necessary for correctness or optimization.
- Describe the decoding process: how a satisfying assignment from the SAT solver can be interpreted to reconstruct a valid sequence of player moves (i.e., the solution to the original problem).

Submission Instructions

- Upload only the file `q2.py` and write your report in `report.pdf`

Note: You must use an **encoding scheme** and a **SAT solver** to solve this question. Any alternative method will result in **zero marks**.

Question 3. `NANDy's Logic Lab` (? Marks)

This is a bonus question. Check the Bonus Policy in the beginning.

In this challenge, you've been recruited by `NANDy`, the eccentric inventor who believes any circuit worth building can be built using only **NAND gates** — no matter how complex!

- You are given a Boolean formula in **CNF (Conjunctive Normal Form)**.
- Your task is to **synthesize a combinational logic circuit** using the **fewest** possible number of gates, with a twist: you may only use **NAND gates** (with arbitrary fan-in — i.e., any number of inputs).
- The circuit should output `True` if and only if the CNF formula is satisfied for a given input assignment.
- **Optimization Goal:** Minimize the number of NAND gates used in your synthesized circuit.
- **Modeling Requirement:** You must model this as a SAT formula and use a SAT solver to synthesize the circuit.

Report Instructions

- Comment your code extensively. In your report, include code snippets and explain what each variable represents, the constraints that have been encoded, and any other important implementation details.

Can you help `NANDy` design the leanest, meanest logic machine?

Input Format

- The first line contains a single integer C: the number of clauses.
- The next C lines each represent a clause in the CNF formula.
- Each clause is a space-separated list of integers, representing literals (positive for x_i, negative for $\neg x_i$).
- For example, the line `-1 2` corresponds to the clause $(\neg x_1 \vee x_2)$.

Output Format

- A description of the synthesized circuit using only NAND gates.
- Each line should define a NAND gate and its inputs.
- The final output gate must be clearly indicated.
- The last line should print the total number of NAND gates used.

Example

Input:

```
1
-1 2
```

This represents the CNF formula: $(\neg x_1 \vee x_2)$

Output:

```
G1 = NAND(x2, x2)
G2 = NAND(G1, x1)
OUTPUT = G2
Total NAND gates used: 2
```

Submission Instructions

- Submit a single file named `q3.py` containing all your code. The program should read input from the terminal (standard input) and print the output to the terminal (standard output) exactly as specified.