Problem 3: NANDy's Logic Lab

In this question, we are given a Boolean formula in CNF form, and we are required to synthesize, using the minimum number of NAND gates possible, a combinational logic circuit which implements the formula. We first find a theoretical upper bound to the minimum number of NAND gates required, and by performing binary search between 1 and the upper bound, reduce it to the satisfiability problem of whether it is possible to implement the CNF formula using exactly G NAND gates. It is then modelled in propositional logic, and solved using the PySAT library.

Binary Search

We determine the upper bound for the binary search as follows:
Let the CNF have m clauses, and the clause C_i have k_i literals. Now, if a clause is of the form $\neg a_1 \vee \neg a_2 \vee \ldots \vee \neg a_n$, where all the literals are negative (the a_i's are atoms), the clause can be expressed using one NAND gate: NAND$(\neg a_1, \neg a_2, \ldots, \neg a_n)$. Considering the worst case, where all literals are positive, we would require at most $\sum_{i=1}^{m} k_i$ NAND gates to get negative literals from propositional variables, m NAND gates to make the clauses, and finally two NAND gates for linking all clauses through $\wedge$. Hence, the upper bound is given by

$$\texttt{maxNand} = m + 2 + \sum_{i=1}^{m} k_i.$$

We perform a binary search and check at each step whether it is possible to implement the circuit in G NAND gates.

Possibility of Implementation in G NAND Gates

This is done by constructing a CNF formula whose satisfiability tells us about this possibility. We encode the logical and structural constraints of a circuit created by G NAND gates in the formula, and also encode the condition that it behaves identical to the provided CNF formula over all input valuations. If the SAT formula is satisfiable, it means such a circuit exists for the current value of G, and we attempt for smaller values of G. Otherwise, we increase G.

Encoding the variables

Let n be the number of variables in the given CNF formula. Each NAND gate in the circuit may either connect to any of the n original inputs or the outputs of the previous gates. We define propositional variables for representing the structure of the circuit. (Note that these are not the variables of the original CNF of the question. We go on later to evaluate the original CNF at all 2^n valuations)

- `alpha(g, i)`: whether the gate g is connected to input i (can be original variable or previous gate). These alphas alone describe the entire circuit, and hence we will be looking at their values mostly. These are also unique for the circuit (independent of the valuation of the input variables).
- `gate(g, v)`: the output of gate g under input valuation v.
- `beta(g, i, v)`: represents $\alpha(g, i) \wedge \neg\text{input}_i$ for input valuation v (used for Tseitin transformation)

These are flattened to integer variables using the function:

`make_atom(self, numG, varType, valuationNum, gateNum, inputNum)`

- `numG`: Total number of NAND gates.
- `varType`: 0 = gate, 1 = alpha, 2 = beta.
- `valuationNum`: Index from 1 to 2^n, representing the input assignment.
- `gateNum`: The index of the gate.
- `inputNum`: Which input variable/gate the gate connects to. For a particular `gateNum`, `inputNum` varies from 1 to n+`gateNum`-1.

This function ensures that the variables are mapped to the natural numbers contiguously and without overlap. We first assign numbers to the alphas (which are unique for the circuit), and then for each of the 2^n valuations, we assign numbers to gates and betas of each valuation.

CNF Encoding

Now, we describe how the behaviour of NAND gates is translated into CNF, using the propositional varables defined earlier. We encode both the structure of the circuit and its correctness (output of circuit matches with the CNF output for each input valuation) into the SAT formula.

1. Gate Output

For each gate g and each valuation v, we ensure that the output is computed according to the semantics of the NAND gate. It computes to true iff at least one of its connected inputs are false. This is implemented using Tseitin encoding using the `beta` variables, where each $\texttt{beta}(g, i, v)$ represents the term:

$$\texttt{beta}(g, i, v) \leftrightarrow \texttt{alpha}(g, i) \wedge \neg\text{input}_i^v$$

For each input variable $x_{\texttt{inputNum}}$, we calculate its truth value, using bit operations, by the expression `(valuation-1) & (1 << (inputNum-1)) != 0`.

When input_i^v is true, beta is false and we have $\texttt{-beta}(g, i, v)$ as a clause.

Otherwise, when input_i^v is false, it becomes $\texttt{beta}(g, i, v) \leftrightarrow \texttt{alpha}(g, i)$ and we have the clauses:
$(\texttt{beta}(g, i, v) \vee -\texttt{alpha}(g, i))$ and $(-\texttt{beta}(g, i, v) \vee \texttt{alpha}(g, i))$.

We implement these by:

```
for gateNum in range(1, numG + 1):

    # SECTION 2: ITERATE OVER INPUTS IN x_1, x_2, ..., x_n
    for inputNum in range(1, self.numVar + 1):
        #define atoms
        if ((valuation-1) & (1 << (inputNum-1))) != 0: #x_inputNum is True
            formula.append([-beta])
        else:                                           #x_inputNum is False
            formula.append([beta, -alpha])
            formula.append([gate, -alpha])

        formula.append([-beta, alpha])
```

When input_i^v is itself a variable (a gate output), the clauses are $(\neg\texttt{beta}(g, i, v) \vee \texttt{alpha}(g, i))$, $(\neg\texttt{beta}(g, i, v) \vee \neg\text{input}_i^v)$ and $(\neg\texttt{alpha}(g, i) \vee \text{input}_i^v \vee \texttt{beta}(g, i, v))$.

The following snippet shows its implementation:

```
    # SECTION 3: ITERATE OVER INPUTS IN GATES
        #define atoms
        for gateNumInput in range(1, gateNum):
            formula.append([beta, -alpha, inputGate])
            formula.append([-beta, alpha])
            formula.append([-beta, -inputGate])
```

The NAND gate output is related by:

$$\texttt{gate}(g, v) \leftrightarrow \bigvee_i \texttt{beta}(g, i, v)$$

In CNF form, this is written as:

$$\neg\texttt{gate}(g,v) \vee \bigvee_i \texttt{beta}(g,i,v)$$

and

$$\bigwedge_i (\neg\texttt{beta}(g,i,v) \vee \texttt{gate}(g,v))$$

Some of this is implemented in the lines:

```
        gateImpliesNotAllInputsOne = [-gate]
        # loop
            gateImpliesNotAllInputsOne.append(beta)
```

Also, we ensure that every gate g is connected to some inputs. This is encoded as:

$$\bigwedge_g \bigvee_i \texttt{alpha}(g,i)$$

The following lines appearing in the code corresponds to this condition.

```
        someConnectionIsMade = []
        #loop
            #execute only once
                someConnectionIsMade.append(alpha)
```

2. Matching Circuit Output to CNF Formula

For each input valuation v, we compute whether the original CNF evaluates to true or false. This is done by evaluating each clause under the valuation using Python loops. If the CNF evaluates to true under valuation v, then the output gate G must output 1. Otherwise, it must output 0.

Evaluating the CNF:

```
    cnfOutput = 1
    for clause in self.cnf:
        # check if clause is true
        clauseSign = False
        for literal in clause:
            if literal < 0:
                # check if assignment is false
                literal = -literal
                if (valuation-1) & (1 << (self.dictVar[literal]-1)) == 0:
                    clauseSign = True
                    break
            else:
                # check if assignment is true
                if (valuation-1) & (1 << (self.dictVar[literal]-1)) != 0:
                    clauseSign = True
                    break
        if not clauseSign:
            cnfOutput = -1
            break
```

This is enforced by adding the clause $\texttt{gate}(G,v)$ if CNF is true under v or the clause $\neg\texttt{gate}(G,v)$ if the CNF evaluates to false.

```
    formula.append([cnfOutput*self.make_atom(numG, 0, valuation, numG, 0)])
```

Binary Searching the Answer

We binary search for the number of gates required to model the given CNF using the `searchAnswer()` function.

Decoding the SAT Model

Once we reach the minimum value of G such that there exists an equivalent NAND formula, we decode the model. In particular, we look at the assignments of the variables $\texttt{alpha}(g, i)$ as they alone decide the entire configuration of the gates. We further use them to output the entire structure of the circuit. This is implemented in the `decode(model, encoder, numG)` function in the code.

Code Acknowledgments

No external code was used. Some inspiration for the code structure was taken from Problem 2. All CNF encodings were implemented from scratch.