UGP CS497

Sudoku Proof Systems

POOJAL KATIYAR – BT/CSE/220770

Supervisor: Dr. Subhajit Roy

Project Mentor: Pankaj Kalita

7th November 2024

Acknowledgement

I would like to express my heartfelt gratitude to Professor Subhajit Roy for the exceptional opportunity to collaborate with him on this project. His willingness to invest time in our regular meetings, despite his demanding schedule, is something I deeply appreciate. His invaluable guidance not only helped me navigate the complexities of the research but also inspired me to persevere in overcoming various challenges.

I would also like to extend my sincere thanks to Pankaj Kalita, whose significant contributions and dedication of time were instrumental in the success of this project. His insights and collaborative spirit greatly enriched our work and made a profound difference in our outcomes.

Sincerely,
Poojal Katiyar

Abstract

This undergraduate project aims to identify well-known Sudoku patterns using formal methods so that system automatically learns them instead of us finding it manually. It focus on developing an inductive learner to generate proof systems for Sudoku solving. This will enable us to detect new patterns from the Sudoku proof system itself by checking on multiple sudoku.

Overview

Sudoku solvers using z3
I am taking as an input an unfilled sudoku, a pair of (r,c) which represents the cell of a sudoku whose value I will negate. To solve a sudoku following sudoku constraints on the z3 variables defined in the 9*9 sudoku are added .

1. Row Uniqueness: Each element in a row must be unique within that row.

$$\forall r \in \{1,\ldots,9\},\ \forall i,j \in \{1,\ldots,9\},\ i \neq j \implies \text{grid}(r,i) \neq \text{grid}(r,j)$$

2. Column Uniqueness: Each element in a column must be unique within that column.

$$\forall c \in \{1,\ldots,9\},\ \forall i,j \in \{1,\ldots,9\},\ i \neq j \implies \text{grid}(i,c) \neq \text{grid}(j,c)$$

3. Subgrid Uniqueness: Each element in a 3×3 subgrid must be unique within that subgrid.

$$\forall k,l \in \{1,2,3\},\ \forall i,j \in \{1,2,3\},\ (i,j) \neq (k,l) \implies \text{grid}(3a{+}i, 3b{+}j) \neq \text{grid}(3a{+}k, 3b{+}l)$$

where $a, b \in \{0,1,2\}$ denote the index of the 3×3 subgrid.
These constraints are added in the solver and if my solver gives sat means that it is satisfying all constraints. Then I can use s.model(), to get a model solution satisfying the constraints.

Additional Constraint

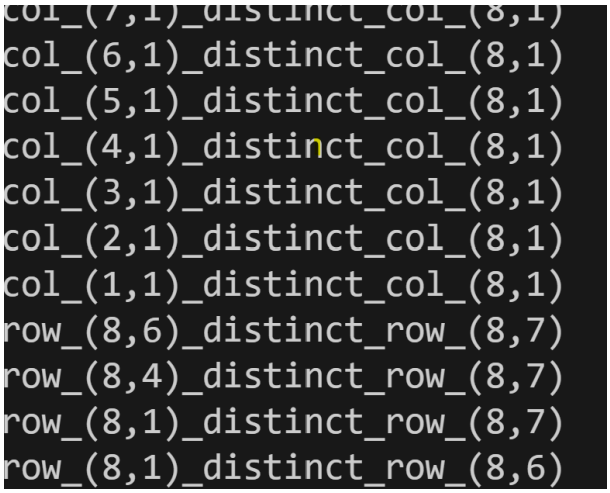
Now, apart from the constraints mentioned above, I am adding another constraint: negating a cell value (which was unfilled in the original puzzle). Consider (r,c) as that cell that has to be negated, then add Not(Cell(r,c)==value), where we would know the value from the sudoku (by generating a model solution by adding the above constraints and I am assuming that each sudoku has a unique solution). This added constraint is structured to make the grid unsatisfiable, essentially by "negating" the value that would otherwise be in that cell. To track these constraints (so that in the unsat core I can get due to which exact minimal constraints I am not getting a satisfiable solution), I used `assert_and_track` for each constraint, allowing to assign unique identifiers to these constraints, such as `"row_(X,Y)_distinct_row_(X,Y)"`, etc. The `s.unsat_core()` plays a critical role in identifying the minimal constraints that make a particular Sudoku setup unsolvable.

Unsat Core Retrieval with `s.unsat_core()`

- After adding the negating constraint, satisfiability check is performed. As expected, `s.check()` will return `unsat` since the constraint makes the puzzle impossible to solve.

- By calling `s.unsat_core()`, I retrieve the list of identifiers for constraints (which is minimal constraints) that are conflicting due to the negated value. This unsat core effectively highlights which row, column, and subgrid constraints are at odds with the new setup.

Example of unsat Core: I get a list of such minimal constraints in my unsat core:



Graph Representation of Conflicting Constraints

To represent the conflicting constraints identified in the `unsat_core`, I built a graph with the following structure:

- **Nodes:** Each node in the graph corresponds to a cell in the Sudoku grid, which is augmented with a list that stores the possible values the cell can take (initially set to $[1, 2, \ldots, 9]$). The node is represented by (r,c,b) where r,c,b are the row, column and block of the cell to which this node corresponds.
- **Edges:** An edge is drawn between nodes to represent a conflicting constraint, with edge labels assigned as follows:
  - `r` for row constraints,
  - `c` for column constraints, and
  - `b` for block (subgrid) constraints.

Example: Suppose we have in our unsatisfiable core the constraint $\text{col}_{(6,1)} \neq \text{col}_{(8,1)}$. We can represent this constraint in a graph by creating two nodes and an edge between them as follows:

1. Create two nodes:
$$(6, 1, 6) \quad \text{and} \quad (8, 1, 6)$$
2. Connect these nodes with an edge labeled c.

Thus, we have:
$$\text{Nodes:} \quad (6, 1, 6) \quad \text{and} \quad (8, 1, 6)$$
$$\text{Edge:} \quad ((6, 1, 6), (8, 1, 6)), \quad \text{labeled as } c$$

The conflicting constraints are represented as pairs of cell values that are in conflict. For cells that are already filled in the original grid, the list of possible values will contain only the single given value. Such cells are considered *committed*.

When a node becomes *committed* (i.e., it holds a single value), this value is removed from the list of possible values for all neighboring nodes connected by edges to this node, thereby helping to narrow down the possibilities of each node.

```python
#This is propagating and checking if any node is committed then we remove that value from its neighbors
    def check_and_propagate():
        committed = True
        while committed:
            committed = False

            # Step 1: Propagate by removing committed values from neighbors
            for node in G.nodes():
                neighbors = list(G.neighbors(node))
                if len(node_values[node]) == 1:  # Committed Node
                    committed_value = node_values[node][0]

                    # Remove the committed value from all neighbors
                    for neighbor in neighbors:
                        if committed_value in node_values[neighbor]: #if committed value present remove it
                            node_values[neighbor].remove(committed_value)
                            committed = True  # Trigger further propagation

                            if len(node_values[neighbor]) == 1:    #This tells us that a new value which was unfilled
                                print(f"Node {neighbor} committed with value: {node_values[neighbor][0]}")
```

1 Algorithm for Identifying Patterns in Graphs

Our algorithm is designed to identify patterns in the graph by analyzing the graph for k hops and to add the necessary constraints in the solver to make it unsat so that we can deduce from unsat core. We want to analyze in the localhood of a node and to what extent I will be analysing depends on the value of k. I will try to find patterns by changing the value of k and the node around whose neighborhood I want to analyze. Doing it inductively for many sudoku will help us deduce common patterns for sudoku.

Steps of the Algorithm:

1. **Generate Subgraphs with Ego Graphs:**
   - For each node in the graph, create an *ego subgraph* centered around this node with radius k.
   - This subgraph includes the node itself, all nodes within k hops, and the edges connecting these nodes as in the original graph.
   - By isolating local neighborhoods, this step reduces the complexity of analyzing the entire graph, allowing us to focus on manageable subgraphs for constraint checking.
2. **Define Constraints for Node Values:**
   - Each node in the subgraph represents a Z3 variable to represent its potential values.
   - *Value Constraints:* A logical OR constraint is applied to each node's possible values, ensuring the node takes exactly one value from its allowed list.
   - *Neighbor Constraints:* A `Not` constraint is imposed between neighboring nodes in the subgraph to ensure no two neighbors share the same value.
3. **Satisfiability Checking with Value Negation or without that:**
   - Let us take a non-committed node of the graph whose list has more than one element and call it as center node.
   - I can use two approaches to generate the unsat core and identify patterns:

- *Without Negation:* For each possible value of the *center node* in the subgraph, the algorithm tests whether assigning that value leads to an unsatisfiable setup. If adding this constraint gives me unsat that means that node cannot have that value and therefore my unsat core will give me the minimal constraints due to which that value cannot be there.
- *Negation Approach:* For each possible value, add a constraint negating this specific value for the center node. This negation essentially questions if the subgraph can be satisfiable when the center node is *not* assigned a specific value. If in this case I get an unsat that means that the cell has that value and my unsat core will tell me the nodes on which it is dependent.

4. **Identify and Analyze Unsat Core:**
   - If the subgraph becomes unsatisfiable in any of the above cases retrieve the *unsat core* from the solver. The unsat core is the minimal set of constraints causing the unsatisfiability.
   - The unsat core helps pinpoint the minimum cells of the sudoku that has caused the Sudoku unsat.
   - This unsat core itself can represent various patterns provided if it is concise and meaningful.

1.1 Naked Pairs

A *Naked Pair* occurs when two cells within the same *house*—a row, column, or block—each have the exact same two possible values, or *pencil marks*. This situation implies that these two values must occupy these two cells exclusively within that house, and therefore, these values can be removed from the list of possible values for all other cells in the same house.

In the algorithm above if I put the value of k=1 and negate the value at the center node, this will give me unsat for the node which is a part of the pattern and cannot have that value. This confirms that value should be present in the node. Like for example below for cell (8,7,8), if I negate the value (that is 6), I get as unsat. And in the unsat core, I get three nodes (8,7,8), (8,1,6), and (8,6,8) which form the naked Pairs pattern which looks like this:

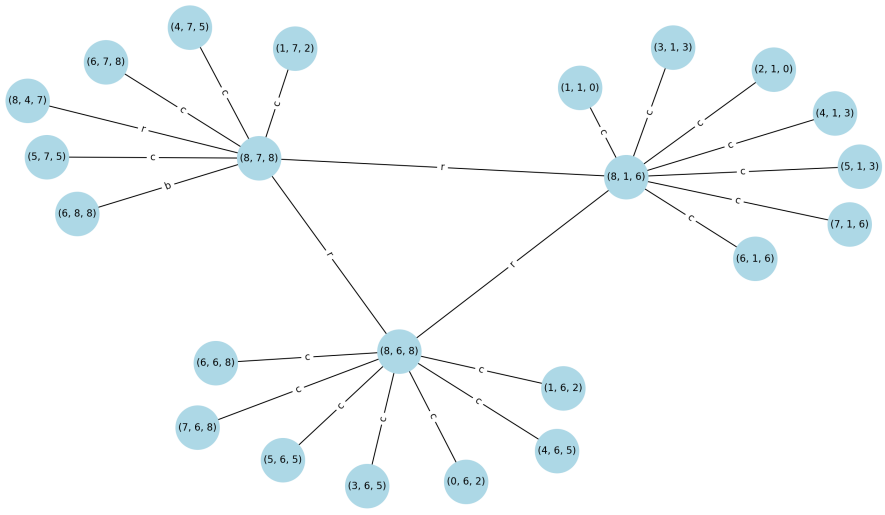


Figure 1: Subgraph showing Naked pairs

```
Subgraph centered at (8, 7, 8): Negating 6 is unsatisfiable.
Unsat core: [negate_(8, 7, 8)_6,
 no_equal_(8, 1, 6)_(8, 6, 8),
 no_equal_(8, 7, 8)_(8, 1, 6),
 no_equal_(8, 7, 8)_(8, 6, 8),
 constraint_(8, 6, 8),
 constraint_(8, 1, 6),
 constraint_(8, 7, 8)]
```

Figure 2: The unsat core for Naked pairs

1.2 X-Wings

When there are only two possible cells for a value in each of two different rows, and these candidates also lie in the same columns, then all other candidates for this value in those columns or rows can be eliminated.

For the X-Wing pattern, I will set $k = 3$ and select a specific value for a node. If this selection results in an unsatisfiable condition, it confirms that the chosen node cannot hold that value. For example, in the case below, node $(0, 2, 0)$ is given the value 6, leading to unsat. The cells causing unsat are $(3, 2, 3)$, $(8, 8, 8)$, $(3, 8, 5)$, $(3, 2, 3)$, and $(8, 2, 6)$. In the X-Wing pattern, the unsat core should contain exactly five nodes: one node that should not hold the value and four additional nodes forming the X-Wing structure, which prevents the chosen node from having that value.

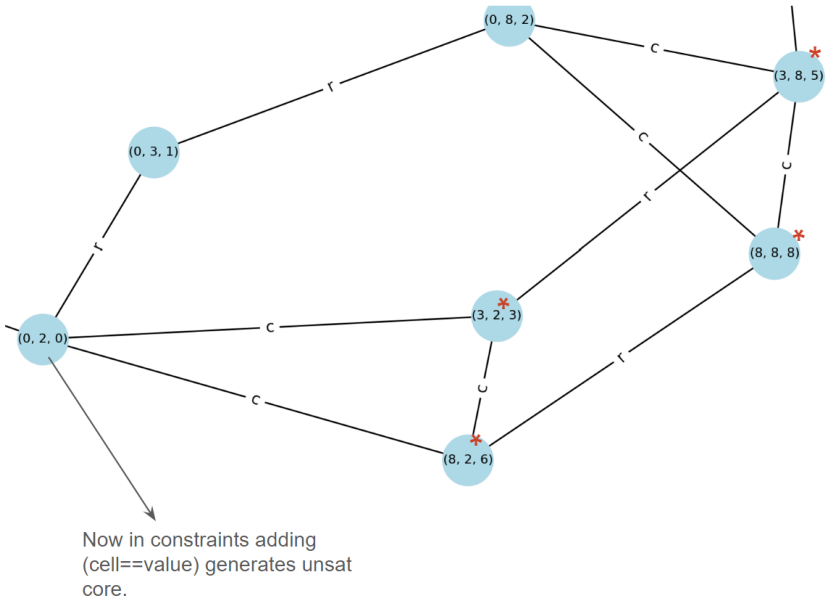


Figure 3: Subgraph showing X wing

```
Subgraph centered at (0, 2, 0): 6 is unsatisfiable.
Unsat core: [no_equal_(8, 2, 6)_(8, 8, 8),
 constraint_(3, 2, 3),
 constraint_(8, 8, 8),
 no_equal_(3, 8, 5)_(8, 8, 8),
 constraint_(3, 8, 5),
 positive_(0, 2, 0)_6,
 no_equal_(3, 8, 5)_(3, 2, 3),
 constraint_(8, 2, 6),
 no_equal_(8, 2, 6)_(0, 2, 0),
 no_equal_(3, 2, 3)_(0, 2, 0)]
```

Figure 4: Unsat Core for X wing

Glossary

- **Constraint**: A condition or requirement that variables in a mathematical model must satisfy. For example, $\text{col}_{(6,1)} \neq \text{col}_{(8,1)}$ is a constraint requiring that columns $(6, 1)$ and $(8, 1)$ hold distinct values.
- **Unsatisfiable Core (Unsat Core)**: The unsat core represents the minimal constraints that have caused the problem to be unsatisfiable.
- **Satisfiability**: A formula/constraint F is satisfiable if there is some assignment of appropriate values to its uninterpreted symbols under which F evaluates to true.
- **Unsatisfiability**: The condition where no assignment of values can satisfy all constraints. If a set of constraints is unsatisfiable, no solution exists that fulfills them simultaneously.
- **Solver**: A tool or algorithm designed to determine the satisfiability of a set of constraints. Solvers can handle logical formulas, optimization problems, or other mathematical constraints.The command Solver() creates a general purpose solver. Constraints can be added using the method add. We say the constraints have been asserted in the solver
- **Z3 Solver**: An efficient theorem prover from Microsoft Research that determines the satisfiability of logical formulas, commonly used for applications in verification, synthesis, and constraint-solving tasks. Z3 is based on first-order logic.
- **Assert and Track**: Assert a constraint (c) into the solver, and track it (in the unsat) core using the Boolean constant p

Github for code: https://github.com/Poojal04/sudoku_proof_system