Sudoku Pattern Discovery

Poojal Katiyar

Roll No:220770

AIM:

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.

Important Terms:

- **Constraint:** A condition or requirement that variables in a mathematical model must satisfy. For example, col(6,1) != 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.

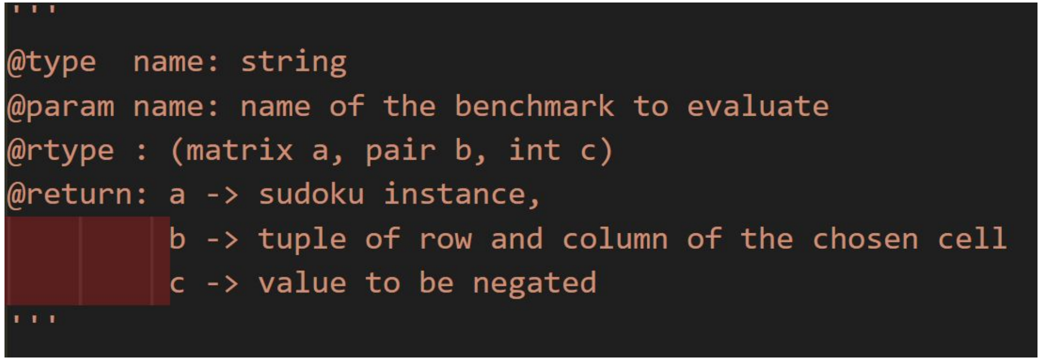
- **Satisfiability:** A formula/constraint F is satisfiable if there is some assignment of appropriate values to its uninterpreted symbols under which F evaluates to true.

- **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.

- **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.

Adding the constraints to the Sudoku:

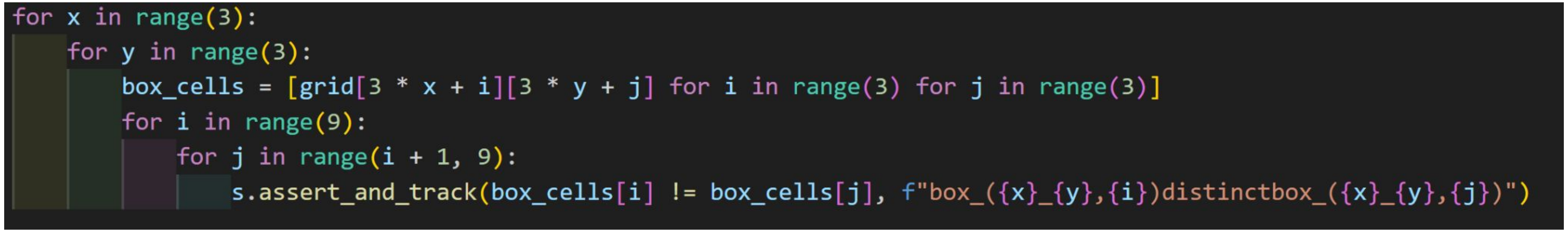
- Firstly I am taking as an input an unfilled sudoku, a pair of (r,c) which represents the cell of a sudoku whose value I will negate and the value which has to be negated. I will be initializing a solver by putting in the constraints. I will put in the unique row, column and block constraints in the solver. Apart from these constraints, I will put another constraint to negate the value in a cell (whose value I would have found by solving sudoku). I used `assert_and_track(...)`to each constraint and attached a descriptive label like `(f"box_({x}_{y},{i})distinctbox_({x}_{y},{j})")` to track which constraint causes an unsat condition.
- I am assuming that each sudoku has a unique solution. Now since that value exists in that cell, negating it will result in unsat.



```
'''
@type  name: string
@param name: name of the benchmark to evaluate
@rtype : (matrix a, pair b, int c)
@return: a -> sudoku instance,
         b -> tuple of row and column of the chosen cell
         c -> value to be negated
'''
```

Overview of The Algorithm:

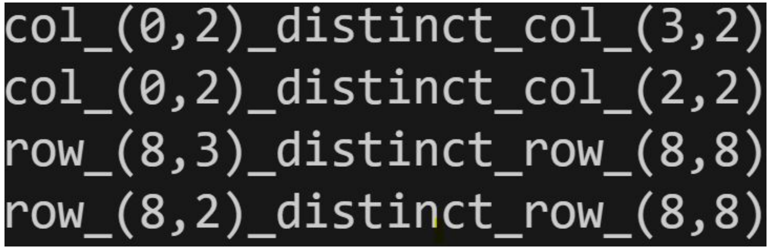
Example of a constraint added along with assert_and_track to track the constraints:



```python
for x in range(3):
    for y in range(3):
        box_cells = [grid[3 * x + i][3 * y + j] for i in range(3) for j in range(3)]
        for i in range(9):
            for j in range(i + 1, 9):
                s.assert_and_track(box_cells[i] != box_cells[j], f"box_({x}_{y},{i})distinctbox_({x}_{y},{j})")
```

I will generate the unsat core for the given Sudoku. The unsat core represents the minimal constraints that have caused the problem to be unsatisfiable.

Example of the unsat Core generated is:



```
col_(0,2)_distinct_col_(3,2)
col_(0,2)_distinct_col_(2,2)
row_(8,3)_distinct_row_(8,8)
row_(8,2)_distinct_row_(8,8)
```

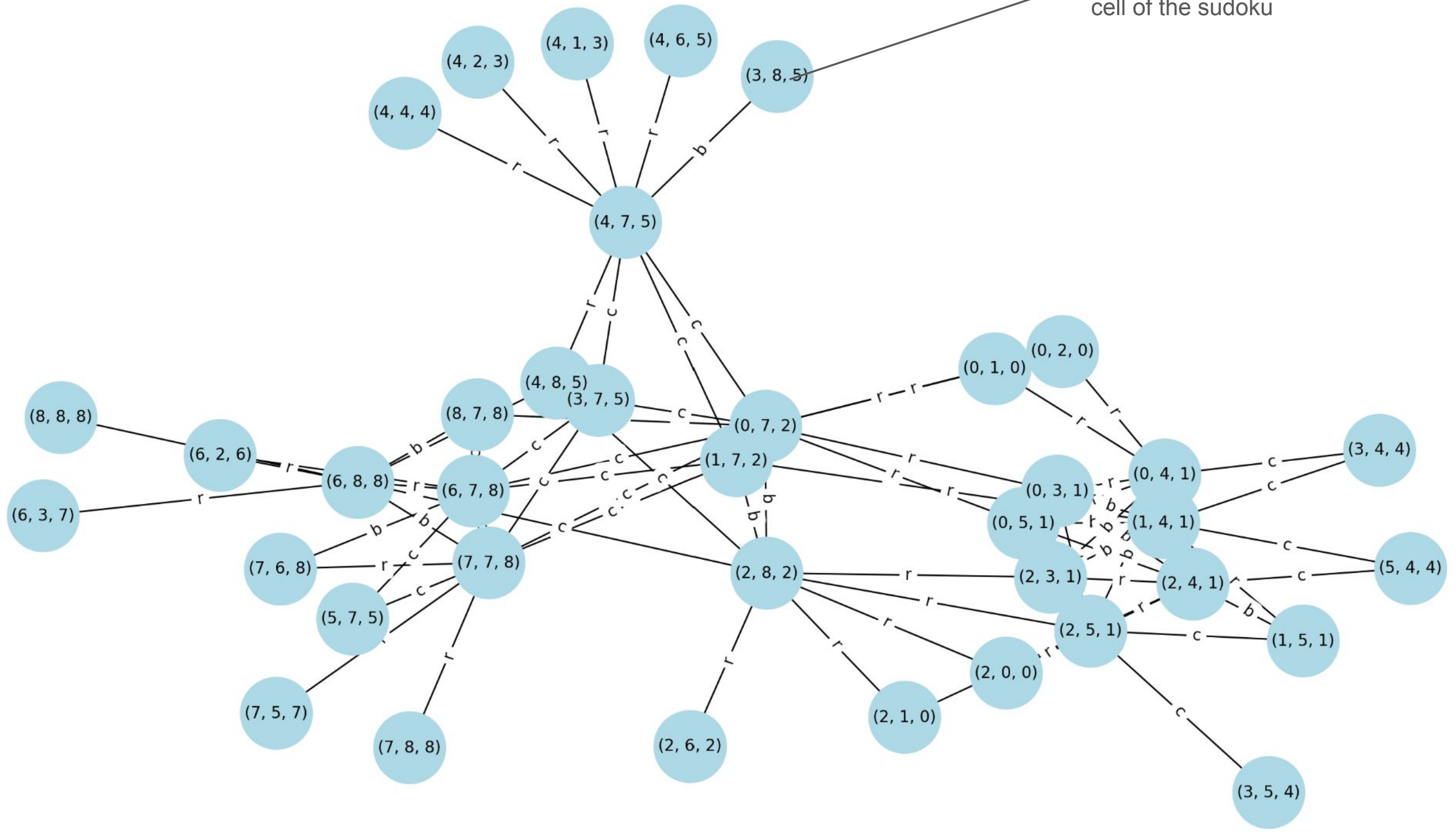
Graph Representation of Conflicting Constraints:

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. It is augmented with a list that stores the possible values the cell can take, initially set to [1, 2, ..., 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:
  - **r** for row constraints,
  - **c** for column constraints, and
  - **b** for block (subgrid) constraints.

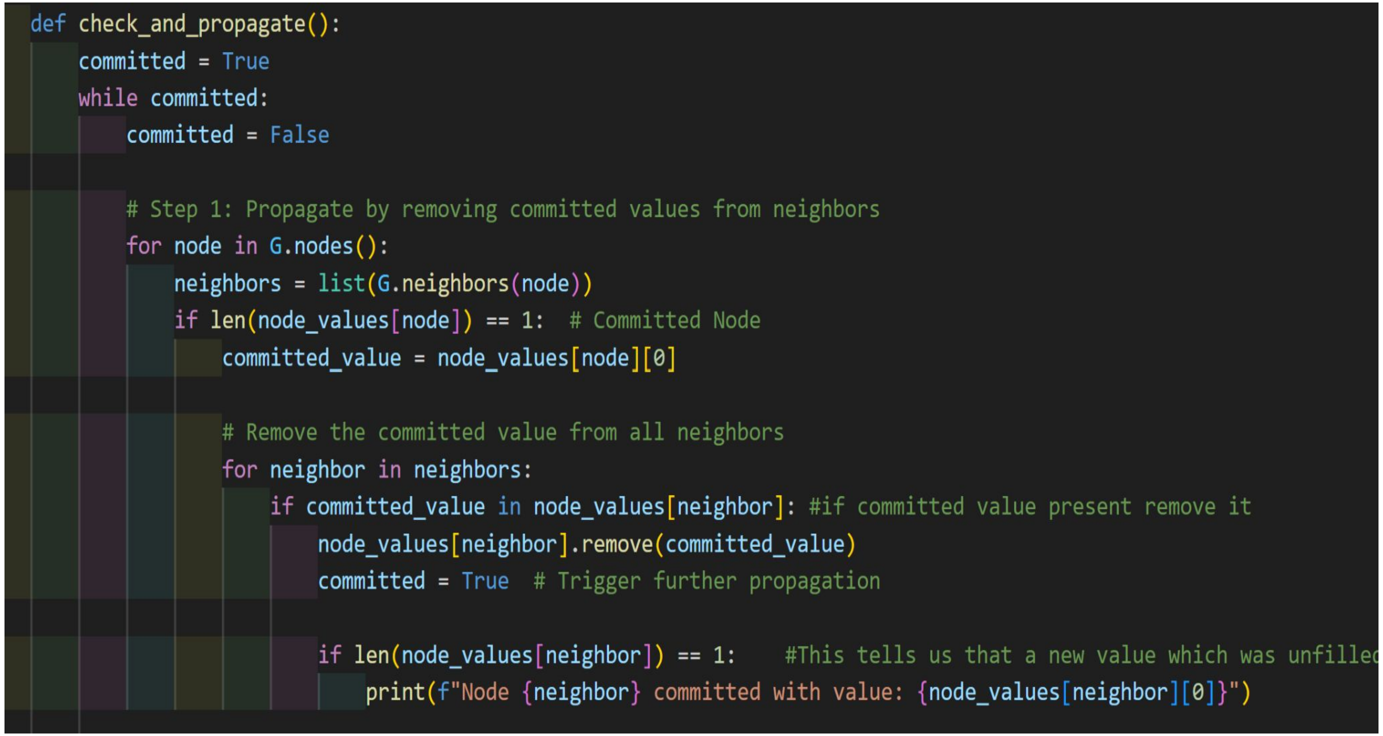
The conflicting constraints are represented as pairs of cell values that are in conflict. For cells already filled in the original grid, the list of possible values will contain only the given value, making these cells *committed*. When a node becomes committed (i.e., it holds a single value), this value 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.

The graph looks like:



Nodes present at leaf will be already having a value in the corresponding cell of the sudoku

Code to propagate the committed value from leaf



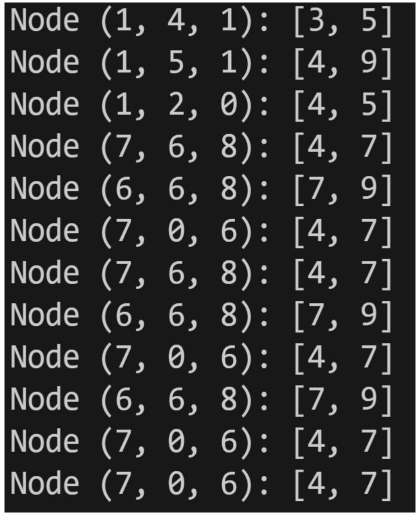
```python
def check_and_propagate():
    committed = True
    while committed:
        committed = False

        # 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 unfille
                            print(f"Node {neighbor} committed with value: {node_values[neighbor][0]}")
```

After applying this algorithm each node's augmented list looks like this:



```
Node (1, 4, 1): [3, 5]
Node (1, 5, 1): [4, 9]
Node (1, 2, 0): [4, 5]
Node (7, 6, 8): [4, 7]
Node (6, 6, 8): [7, 9]
Node (7, 0, 6): [4, 7]
Node (7, 6, 8): [4, 7]
Node (6, 6, 8): [7, 9]
Node (7, 0, 6): [4, 7]
Node (6, 6, 8): [7, 9]
Node (7, 0, 6): [4, 7]
Node (7, 0, 6): [4, 7]
```

Overview of The Algorithm:

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.

- Now we have reduced the possible values that a node can take.
- Now I will be taking each node of the graph, let’s call it center node and will make a subgraph with k hops (Nodes at distance of <=k from the center node).
- This is like an inductive learner where I will be repeating this for multiple sudoku puzzles and eventually this will confirm the pattern and even tells the steps of deducing it.
- Now I will be applying the constraints on this subgraph obtained

Overview of The Algorithm:

Each node in the subgraph represents a Z3 variable to represent its potential values.

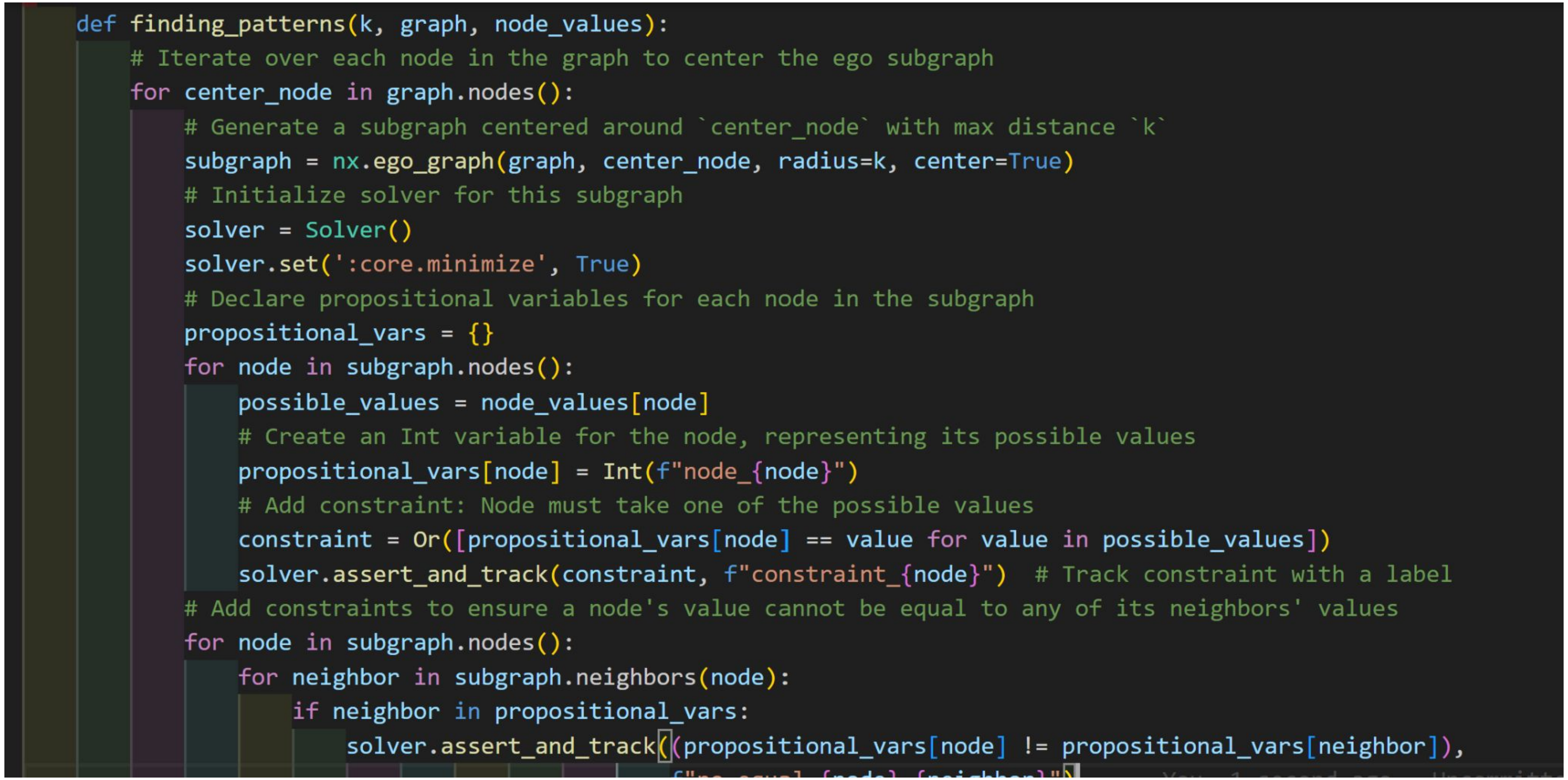
- **Value Constraints**: A logical OR constraint ensures each node selects exactly one value from its allowed list.
- **Neighbor Constraints**: A NOT constraint between neighboring nodes prevents them from sharing the same value.
- Now I will be trying for multiple instance each time choosing a value from the list that center node can have and then checking the satisfiability in 2 cases

  Case 1: I negate the value and add it as a constraint

  Case 2: Add the value as it is as constraint.

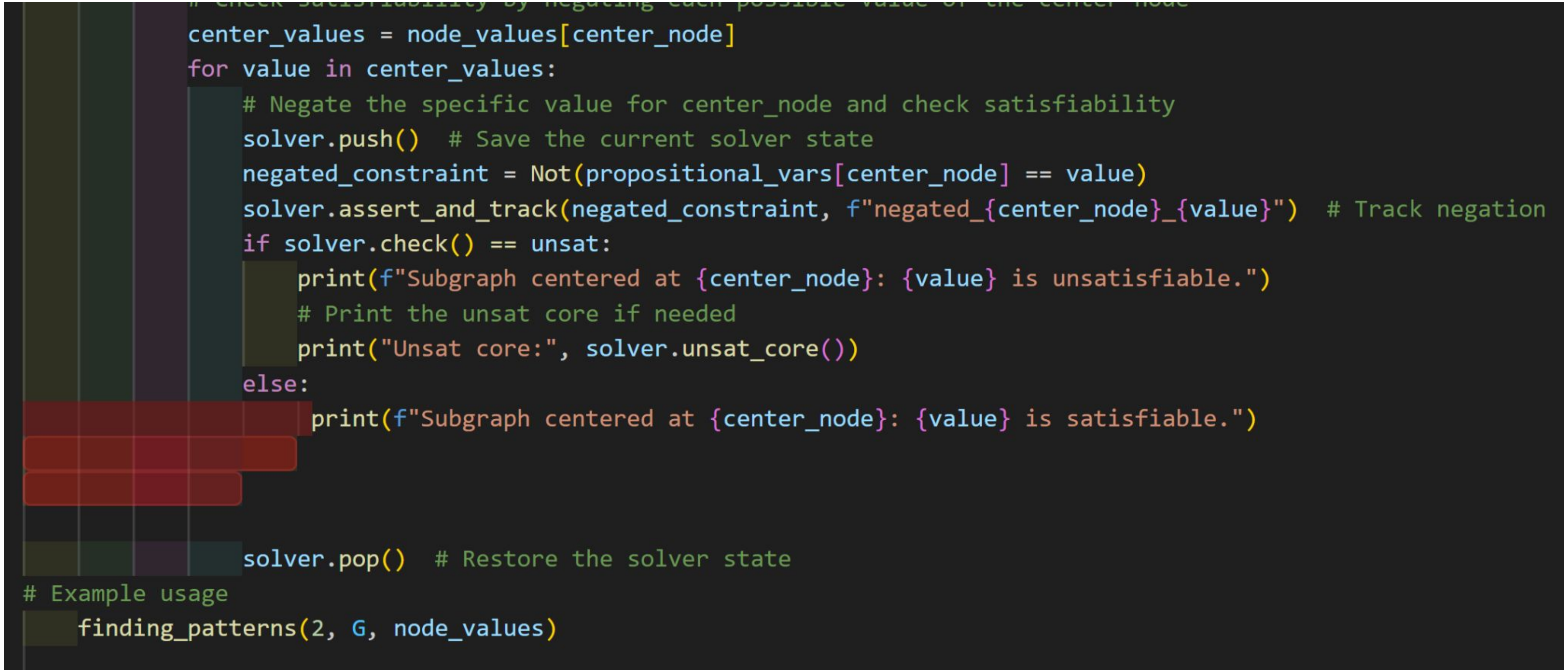
  This I will be doing for all possible values that is present in the list of center node.

Code



```python
def finding_patterns(k, graph, node_values):
    # Iterate over each node in the graph to center the ego subgraph
    for center_node in graph.nodes():
        # Generate a subgraph centered around `center_node` with max distance `k`
        subgraph = nx.ego_graph(graph, center_node, radius=k, center=True)
        # Initialize solver for this subgraph
        solver = Solver()
        solver.set(':core.minimize', True)
        # Declare propositional variables for each node in the subgraph
        propositional_vars = {}
        for node in subgraph.nodes():
            possible_values = node_values[node]
            # Create an Int variable for the node, representing its possible values
            propositional_vars[node] = Int(f"node_{node}")
            # Add constraint: Node must take one of the possible values
            constraint = Or([propositional_vars[node] == value for value in possible_values])
            solver.assert_and_track(constraint, f"constraint_{node}")  # Track constraint with a label
        # Add constraints to ensure a node's value cannot be equal to any of its neighbors' values
        for node in subgraph.nodes():
            for neighbor in subgraph.neighbors(node):
                if neighbor in propositional_vars:
                    solver.assert_and_track((propositional_vars[node] != propositional_vars[neighbor]),
                                      f"no_equal_{node}_{neighbor}")
```

Code



```
            # Check satisfiability by negating each possible value of the center node
            center_values = node_values[center_node]
            for value in center_values:
                # Negate the specific value for center_node and check satisfiability
                solver.push()  # Save the current solver state
                negated_constraint = Not(propositional_vars[center_node] == value)
                solver.assert_and_track(negated_constraint, f"negated_{center_node}_{value}")  # Track negation
                if solver.check() == unsat:
                    print(f"Subgraph centered at {center_node}: {value} is unsatisfiable.")
                    # Print the unsat core if needed
                    print("Unsat core:", solver.unsat_core())
                else:
                     print(f"Subgraph centered at {center_node}: {value} is satisfiable.")



                solver.pop()  # Restore the solver state
# Example usage
    finding_patterns(2, G, node_values)
```

Overview of The Algorithm:

The principle I will use is that F is valid only when Not(F) is unsatisfiable.

- Now suppose for the center value, I add the value of cell in the constraints like (cell==value) . Here value can be any of all values it can take from the list. Now there can be two scenarios that can come either solver gives a satisfiable solution. Now satisfiable solution does not guarantee that value will be there at that node. Since, the constraints have been reduced and have been applied to a subgraph. So it may be that the value is present.

  But if it gives an unsatisfiable solution, this confirms that value cannot be there in the node.. Therefore I can get the unsat core(which are the minimal constraints) and see which nodes are responsible for making that cell as unsat. So this will give us patterns like that value cannot be present in the cell due to certain other cell values. I will give an example of X wing in the next slide.For X wing value of k chosen is 3. This is because upto 3 hops the value due to a cell is affected.

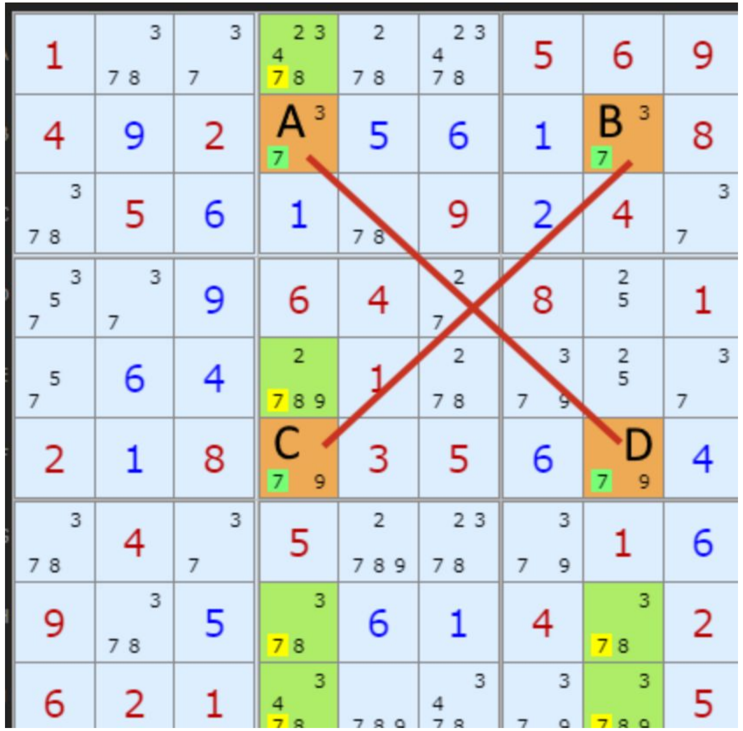
How X wing looks like

The rule is

When there are

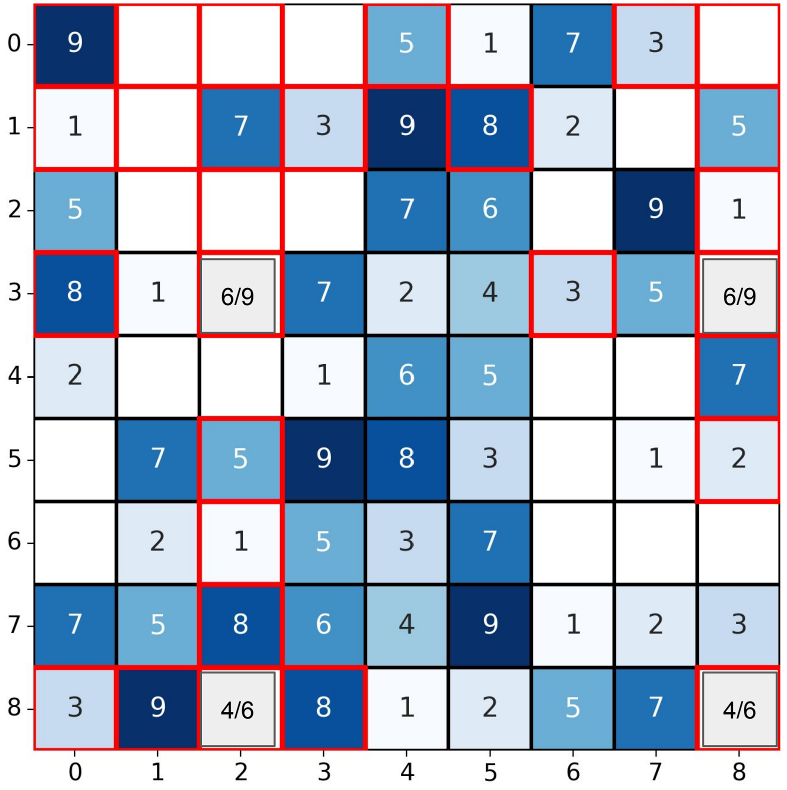
- only two possible cells for a value in each of two different rows,
- and these candidates lie also in the same columns,

then all other candidates for this value in the columns or rows can be eliminated.



X wing(using my algorithm)

Sudoku Grid with Conflicting Constraints Highlighted



	0	1	2	3	4	5	6	7	8
0	9				5	1	7	3	
1	1		7	3	9	8	2		5
2	5				7	6		9	1
3	8	1	6/9	7	2	4	3	5	6/9
4	2			1	6	5			7
5		7	5	9	8	3		1	2
6		2	1	5	3	7			
7	7	5	8	6	4	9	1	2	3
8	3	9	4/6	8	1	2	5	7	4/6

List corresponding the cell

Cell (3, 2, 3) can have values: {6, 9}

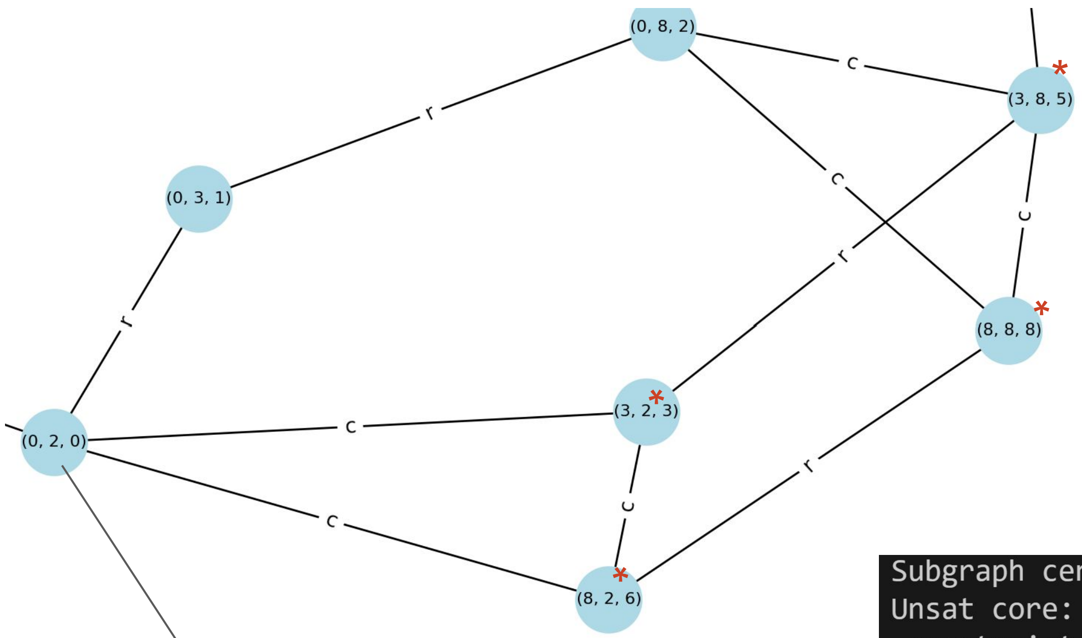
Cell (3, 8,5) can have values: { 6, 9}

Cell (8, 2, 6) can have values: {4,6}

Cell (8, 8, 8) can have values: {4, 6}

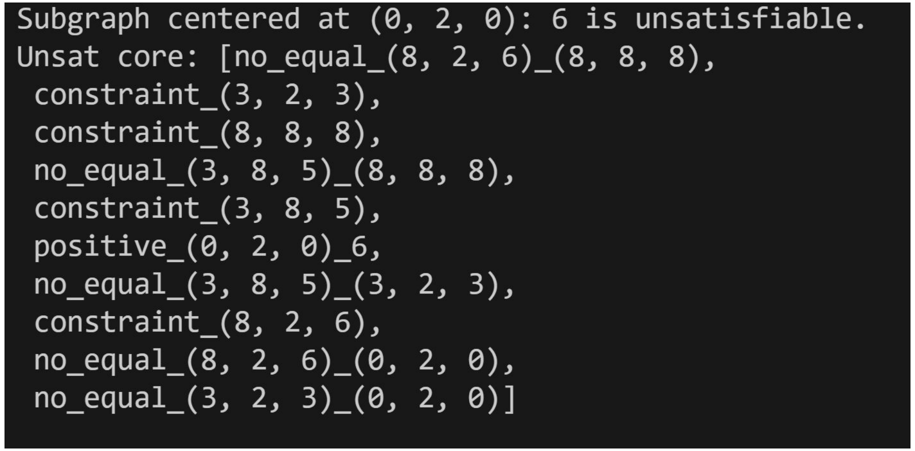
Cell (0, 2.0) can have values: {2, 4, 6}

This list of values I am obtaining from applying the step in the algorithm.For the nodes which will be committed, its neighbors cannot have that value. Therefore that value will be removed from the list of its neighboring nodes.



Cell marked with asterisk were coming in the unsat core which forms the X wing along with the cell (0,2,0) on which value was forced.

Now in constraints adding (cell==value) generates unsat core.



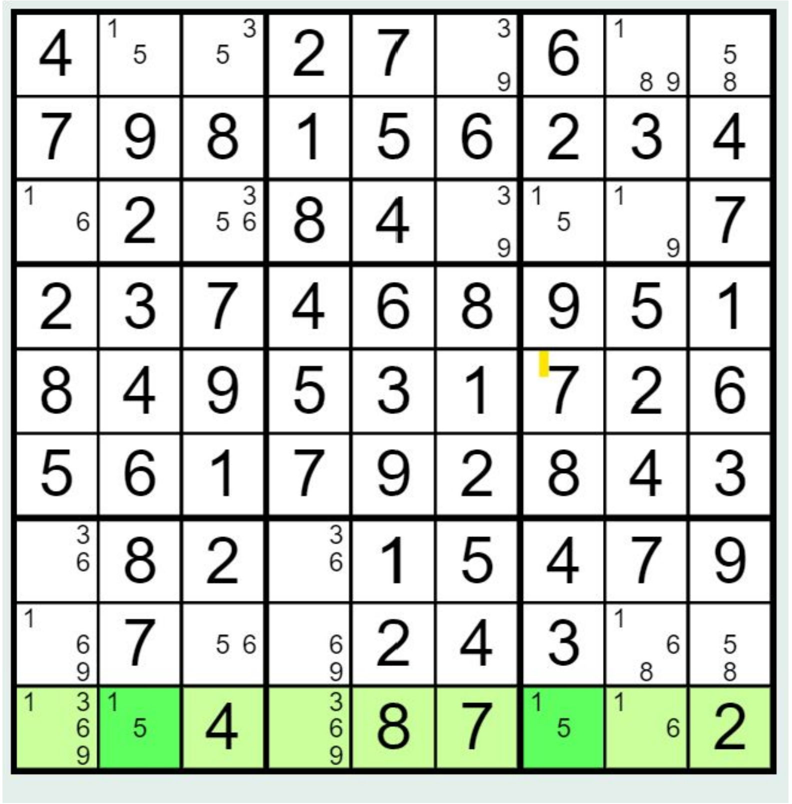
```
Subgraph centered at (0, 2, 0): 6 is unsatisfiable.
Unsat core: [no_equal_(8, 2, 6)_(8, 8, 8),
 constraint_(3, 2, 3),
 constraint_(8, 8, 8),
 no_equal_(3, 8, 5)_(8, 8, 8),
 constraint_(3, 8, 5),
 positive_(0, 2, 0)_6,
 no_equal_(3, 8, 5)_(3, 2, 3),
 constraint_(8, 2, 6),
 no_equal_(8, 2, 6)_(0, 2,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

Overview of The Algorithm:

- Now suppose for the center value, I add the negating value of cell in the constraints like Not (cell ==value) . Now there can be two scenarios that can come either solver gives an unsatisfiable solution. Now unsatisfiable solution means that Not(Not(cell==value)) is true which means Cell==value is true.
- For satisfiable solution, we are again not sure whether with all other constraints of the main graph it will still be satisfiable or not. Therefore unsatisfiable solution gives me an unsat core which confirms which other cells are responsible for the center value.One such pattern detected is Naked Pair pattern.
- For Naked Pair, value of k chosen is 1, as the cell affects just the neighbors.

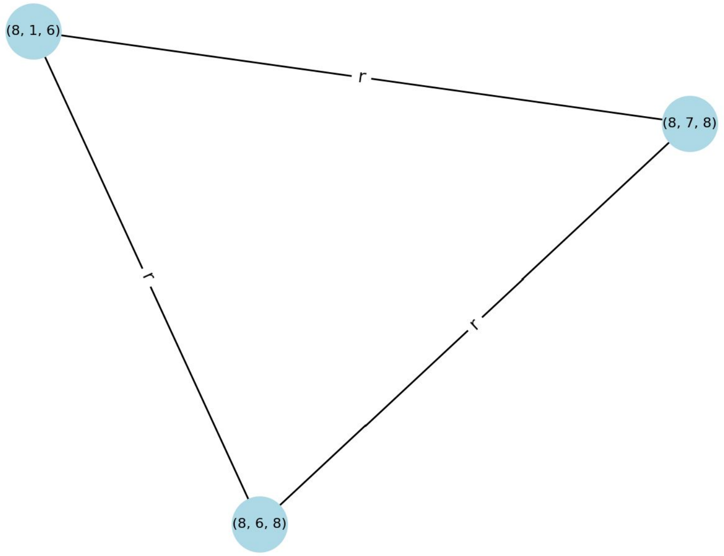
How does Naked Pair pattern looks like

- A naked pair is when you have exactly two cells within a house (which can be either row, column, or 3×3 block) that only have the exact same two candidates.I just don't know which order they go in.
- So from the house any other cell cannot have either of those 2 candidates.



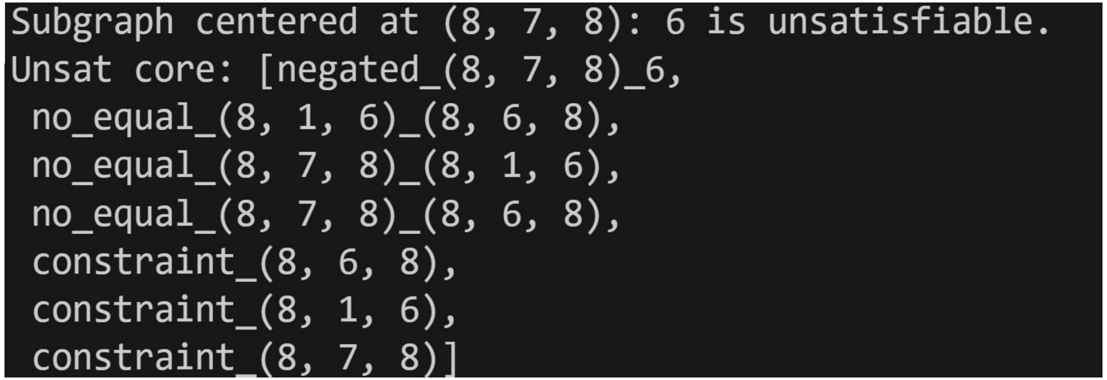
4	1 5	3 5	2	7	3 9	6	1 8 9	5 8
7	9	8	1	5	6	2	3	4
1 6	2	3 5 6	8	4	3 9	1 5	1 9	7
2	3	7	4	6	8	9	5	1
8	4	9	5	3	1	7	2	6
5	6	1	7	9	2	8	4	3
3 6	8	2	3 6	1	5	4	7	9
1 6 9	7	5 6	6 9	2	4	3	1 6 8	5 8
1 3 6 9	1 5	4	3 6 9	8	7	1 5	1 6	2

Naked Pairs(using my algorithm)



Subgraph with center_node (8,7,8)

Node (8, 7, 8): [1, 5, 6]
Node (8, 6, 8): [1, 5]
Node (8, 1, 6): [1, 5]
This list of values is deduced from the first step again. Here k=1 is chosen and in subgraph obtained, the center values are negated and imputed to get unsat core



```
Subgraph centered at (8, 7, 8): 6 is unsatisfiable.
Unsat core: [negated_(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)]
```

Thank You Sir

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