SUDOKU SOLVING USING GRAPH TRANSFORMATIONS

Graph Transformations

2024-2025

Index

Introduction

Sudoku, originated from a Swiss concept but gained worldwide popularity after being introduced in Japan in the 1980s. It is a logic-based puzzle played on a 9x9 grid, where the goal is to fill in the numbers 1 to 9 so that each row, column, and 3x3 subgrid contains all digits without repetition. Over time, it expanded to other versions, such as 4×4 grids or variations with more than the typical subgrids, using letters... It’s a widely enjoyed challenge that tests reasoning and attention to detail.

This project will show how using graphs transformations this game can be modeled and solved, testing different approaches and then comparing them to find which is the best.

Objectives

1. Design and implement a clear, efficient graph-based representation of Sudoku.
2. Develop a graph transformation algorithm to solve standard (9x9) Sudoku.
3. Generalize the solution to accommodate variations (e.g., grid sizes, irregular regions) while maintaining correctness.
4. Evaluate and compare different graph representations and solutions to determine the most optimal approach.

Description of the problem

The game's design must include a representation of each cell and a way to define relationships between them, so the graph can identify rows, columns, and subgrids.

The original Sudoku uses numbers from 1 to 9, but to make it more general, it should also support other metrics, such as letters ("a" to "i").

About the grids, there are different Sudoku systems that change their size or the number of them.

There are Sudokus that seem unsolvable because of the lack of information, but for this implementation, it should find all possible solutions that don’t break any rules, even if there is more than one.

Modeling the problem

The tool chosen for this project is GROOVE, which is well-suited for graph transformation and model checking based on graph rewriting systems.

Approaches

There are different ways to represent the game, and depending on the design, the solution doesn’t change, but the way it reaches it does. That's why it's important to understand each design and see its flaws and strong points.

All the representations have one detail in common: each cell is represented by a node. This is because it allows you to clearly define relationships and constraints using edges, making it easy to apply the rules to the graph.

Three representations of the problem are:

1. Undirected graph: Each node has its own information, meaning its position (row and column) and its value. There are two types of nodes: "Node", which don't have a value yet, and "NodeVal", where they do have it.
2. Directed graph with edges between cells: Each cell node has its own value (empty means 0), but the relations between them are through edges. There are three types of edges: "gc" (group column), "gr" (group row), and "gg" (group grid). To avoid unnecessary edges, they are only between neighboring nodes (those in the same column but one row more or less, in the same row but one column more or less, or one column and row more or less at the same time). These edges can be incoming or outgoing from that node, but they represent the same information.
3. Directed graph with edges pointing to groups: Each cell node has its own value (empty means 0), but they are related to another node type "G" that determines the group they belong to (in the original sudoku row, column, or grid). It cannot have more than one node pointing to it with the same value.

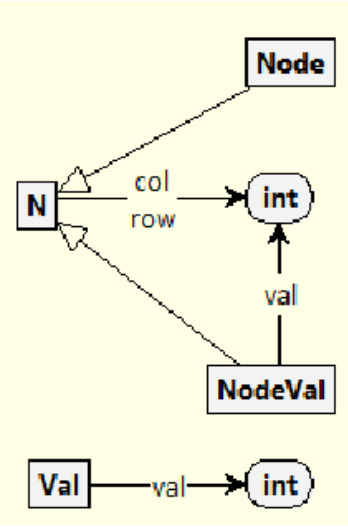
All these approaches also have a node called 'Val,' which is used as storage where the rules take one of their values to assign it to the empty nodes and, with that, arrive at the solution.

Typing

Once the idea of each approach is established it is important to set a type graph to reinforce the structure and make it more solid.

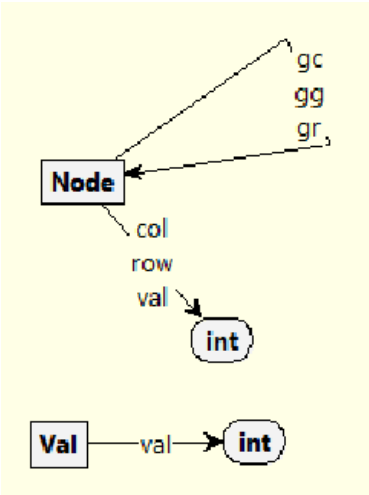
As explained before, they all share some characteristics like the “Val” node pointing to an integer.

These are the type graphs for each approach:



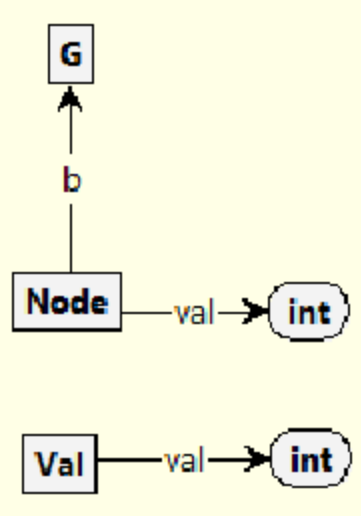
Undirected graph: This one corresponds to Figure 1. There are 3 types of nodes where “Node” and “NodeVal” are descendents of “N” because they are the same flags but “NodeVal” also has a flag for the value. All of these 3 flags are integres as can be seen because they point a int node. And, of course, there is also the “Val” node pointing to an int with the flag of the value.

Figure 1. Type graph of the Undirected graph approach.



Directed graph with edges between cells: This one corresponds to Figure 2. There are 2 types of nodes, “Node” with 3 flags to identify his positions and his value (the position is only needed to create the edges in the start graph) with a self-edge meaning that between nodes type “Node” there will be an edges representing their group relation, and the other type of node is “Val”, same as the other approaches.

Figure 2: Type graph of the Directed graph with edges between cells



Directed graph with edges pointing to groups: This one corresponds to Figure 3. There are 3 types of nodes, “Node”, which represents every cell and has his value, “G” that has incoming edges from the nodes that belong to that group, and “Val”, same as the other approaches.

Figure 3: Type graph of Directed graph with edges pointing to groups

As can be seen, the differences are clear between each approach, even though, that they share some characteristics like the “Val” node pointing to an integer.

Start graph

The initial graph is essential for solving it because it is the representation before starting the transformations and, therefore, applying the rules to it.

The start graphs for the original sudoku 9x9 are the following:

Figure 4. Start graph the Undirected graph approach

In this Figure 4, can be seen that the graph does not have any edge between different cells because they mainatin their relations with the flags of column and row as explained before.

It can also be seen that there is a node on the top lef corner that represents the possible values that can take any cell during the transformations.
Also, there are 2 type of cell nodes depending if they have or not a value. The “Node” are for the ones that does not have value yet, like the one in row 2, column 3, and there are the “NodeVal” that they have value, like row 1, column 1.

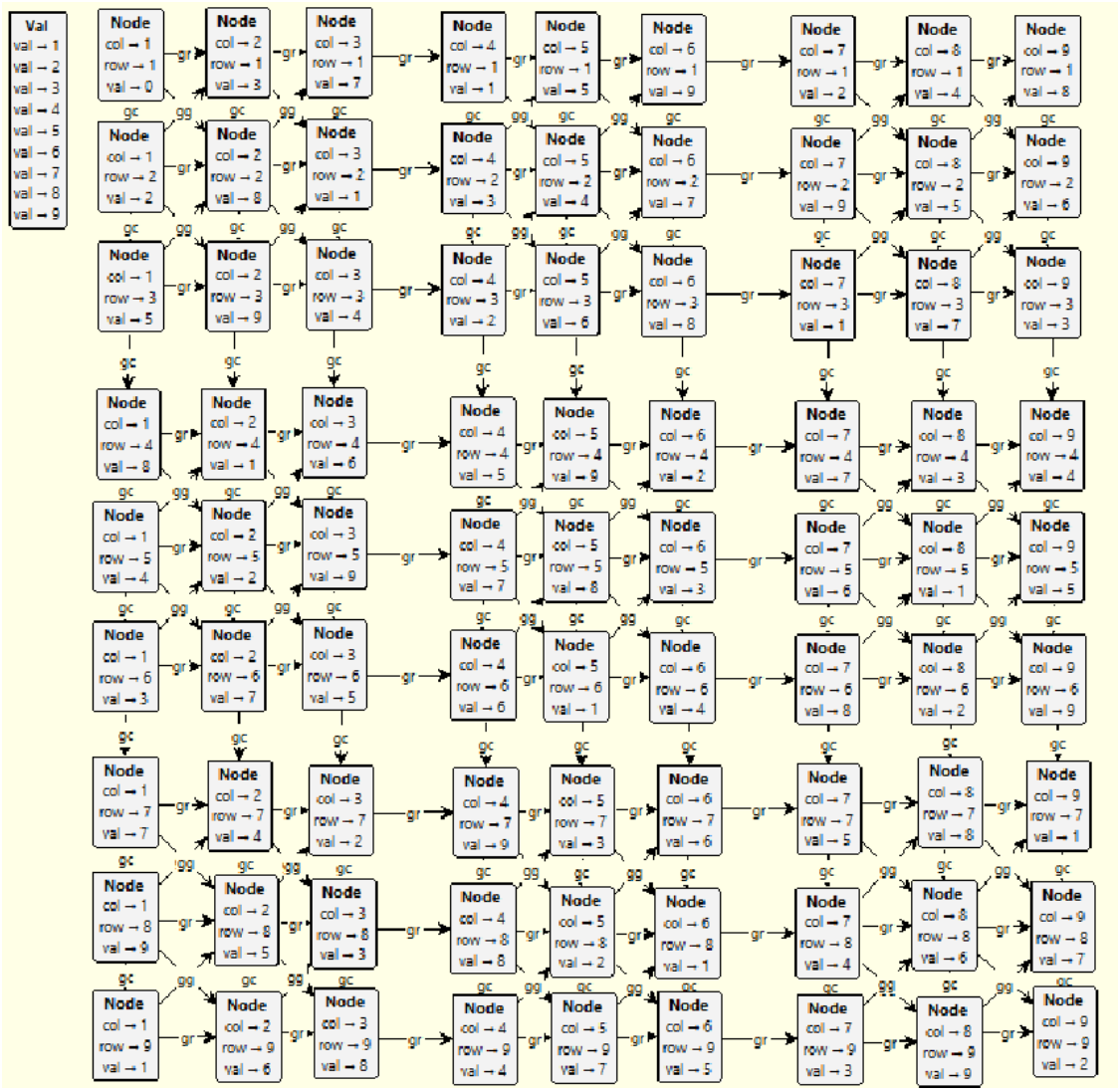


Figure 5. Start graph of the Directed graph with edges between cells

In this Figure 5, can be seen that there is the same structure as the undirected graph (flags for row and column) and that is because to generate the edges between the nodes relation they were necessary to identify them in the space. Besides, there are these edges between the neighboring cells to represent their relation.

Also, there is the node “Val” to store the possible values to later take them from there for the solution.

The cells without value are not different that if they do. The only difference is that they have a 0 value.

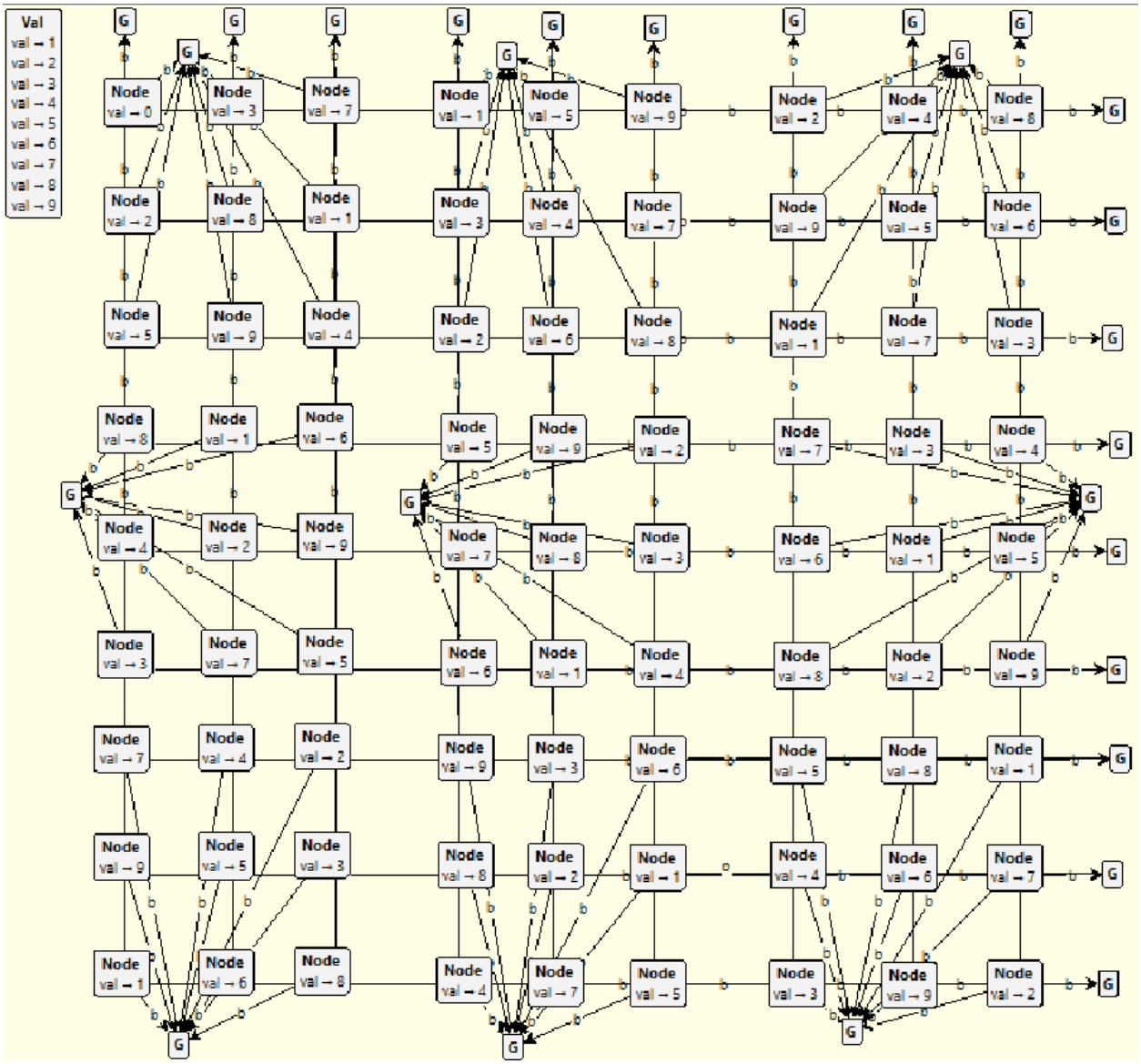


Figure 6. Start graph of Directed graph with edges pointing to groups

In this Figure 6, can be seen that the structure is very different from the other start graphs because it does not need the positions flags due to the “G” nodes that represent the groups of relations that have the cells, like rows, columns or grids.

Therefore, are 3 types of nodes, the “G”, de “Node” representing the cells, and the “Val” which stores the possible values to later take them from there.

If there is an empty cell it is represented in the flag of that node by 0 as can be seen in the top left corner cell.

Solution

Once the start and type graphs are set, it is time to create the rules that will make the graph transform and arrive at all possible solutions of it automatically.

Rule systems

Each approach has its specific needs, making every rule system very different from the others. Although, they all have just 1 rule, and these are those:

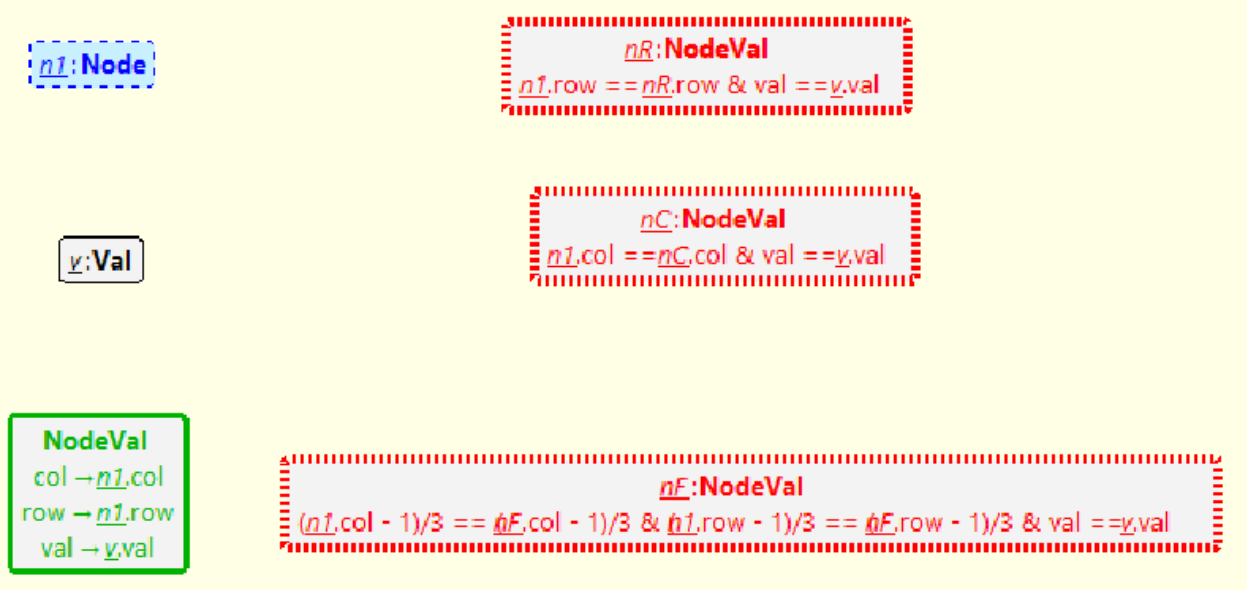


Figure 7. Rule system of the Undirected graph approach

In Figure 7 there is the rule for the Undirected graph approach.

This rule represents the deletion of the empty node “n1:Node” and the creation of the corresponding “NodeVal” in that position. This can be represented as the Left part of the rule as blue, representing the deletion of the “Node”, and then, the right part of the rule represented as green for the new “NodeVal” with the same column and row as the cell before, but now, with the new value extracted from the storage node, “Val”. There are also 3 NACs to test that this new value can be placed in this position:

1. Same row: It is the red one on top, and checks if there is any other “NodeVal” on the same row as the “Node” erased and the same value that should be placed.

2. Same column: It is the red one on the middle, and checks if is there any other “NodeVal” on the same column as the “Node” erased and the same value that should be placed.
3. Same grid: It is the one on the bottom and checks if there is any other “NodeVal” with the same value on the same grid with a formula. It’s important to understand that this NAC only works for the original 9x9 sudoku game, if it changes the shape of it or the way to represent each column and rows, for example, with letters, this calculation could not be done.

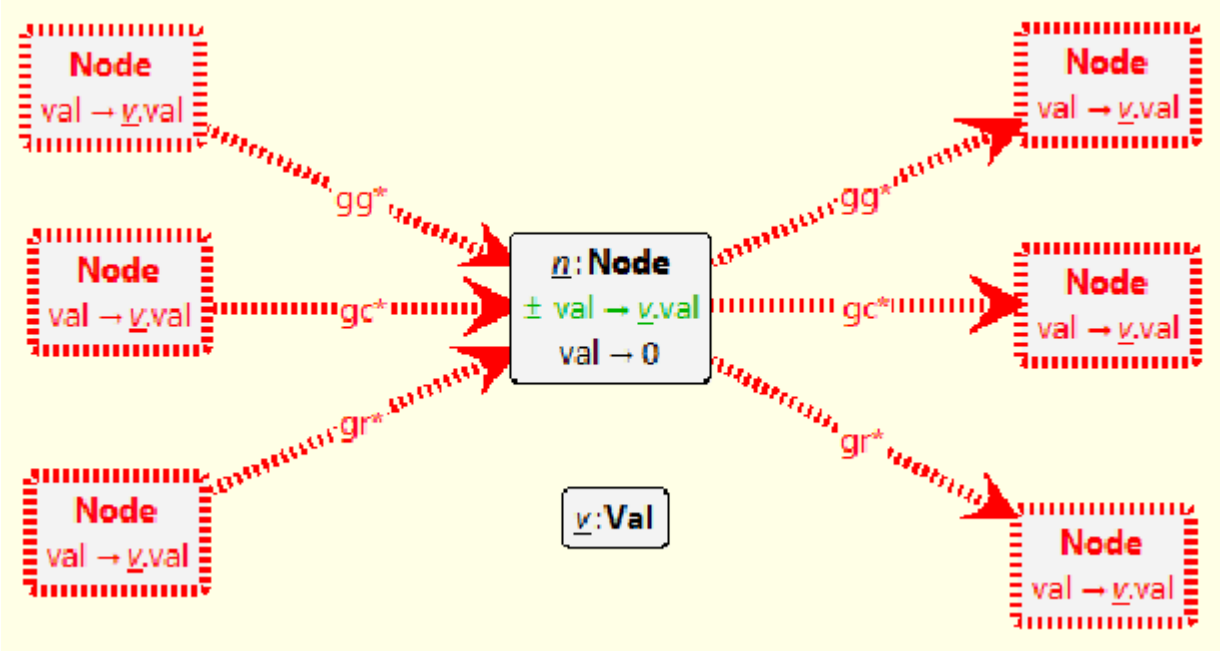


Figure 8. Rule system of the Directed graph with edges between cells

In Figure 8 there is the rule for the Directed graph with edges between cells.

This rule represents the change of the value on the cell, from nothing to one value that does not match with any value in the same row, column or grid. This is rule is represented as a left part being the match of the “Node” with value equal to 0, to the right part being green, representing that that “Node” will have one value that it is inside of the storage node “Val”.

There are a couple of NACs but they all represent the same idea, that there cannot be another “Node” in the same row (even though that only points to one “Node” it also points to the next of that one thanks to the Regular expression “*”), in the same column or in the same grid with the same value taken from the storage, “v.val”.

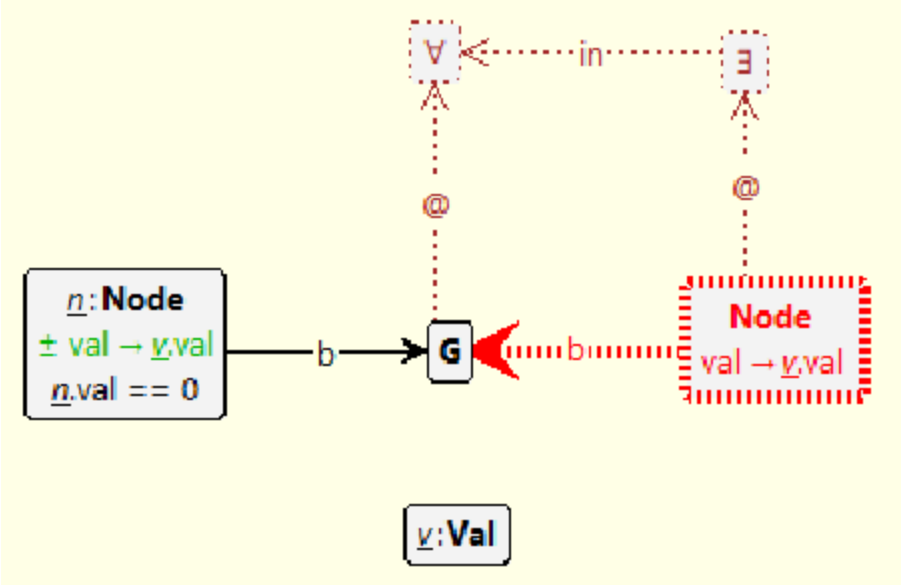


Figure 9. Rule system of the Directed graph with edges pointing to groups

In Figure 9 there is the rule for the Directed graph with edges pointing to groups.

This rule represents the change of the value on the cell, from nothing to one value that does not match with any of the other nodes that point in the same group as that node, “G”. This is represented by a left part of the rule being the value set to 0, to a right part of the rule being the value set to one of the values that is available between the options that had in the storage node, “Val”.

There is a NAC to not set same value to two cells that share any group. This is represented with the red node which is pointing to G, meaning that there cannot be a node pointing to G with the value “v.val” at the same time as other node wants to set that “v.val” to that cell. Also, there are different nodes to express the Alegra between this NAC to determine that only one cell with that value between all the groups that are pointing there.

Winning states

Once all the rules are set, it is just a matter of time for the pushout to finish and arrive to a solution, if there is any, in the start graph. That is why it is important to identify these winning states and then see which are the solutions to that sudoku. This is why there must be a graph condition, for once that the exploration is finished, identify the finale state.

Of course, for each approach the graph condition can change because of how are represented each cell and relations. But all three are very simple and share the same idea because during the pushout the rules do not tolerate conflict between values in the same row, column or grid, meaning that if there are no nodes empty, means that it is a solution.

These are the graph conditions for each approach seen:

Figure 10. Graph condition of the Undirected graph approach

This Figure 10 shows the condition that needs to match the finale state to consider it a good solution.

This condition means that there cannot be any “Node”, because if it has a value is represented as “NodeVal”, so the graph needs to be complete to be true.

Figure 11. Graph condition of the Directed graph with edges between cells

This Figure 11 shows the condition that needs to match the finale state to consider it a good solution.

This condition means that there cannot be any “Node” with value equal to 0, so the graph needs to be complete to be true.

Figure 12. Graph condition of the Directed graph with edges pointing to groups

This Figure 12 shows the condition that needs to match the finale state to consider it a good solution.

This condition is the same as the second approach because they share the same structure of cells so to represent it is the same.

Once all the graph conditions are created the finish state can be found in the simulation in green. This is an example of the third approach, but all three are similar:

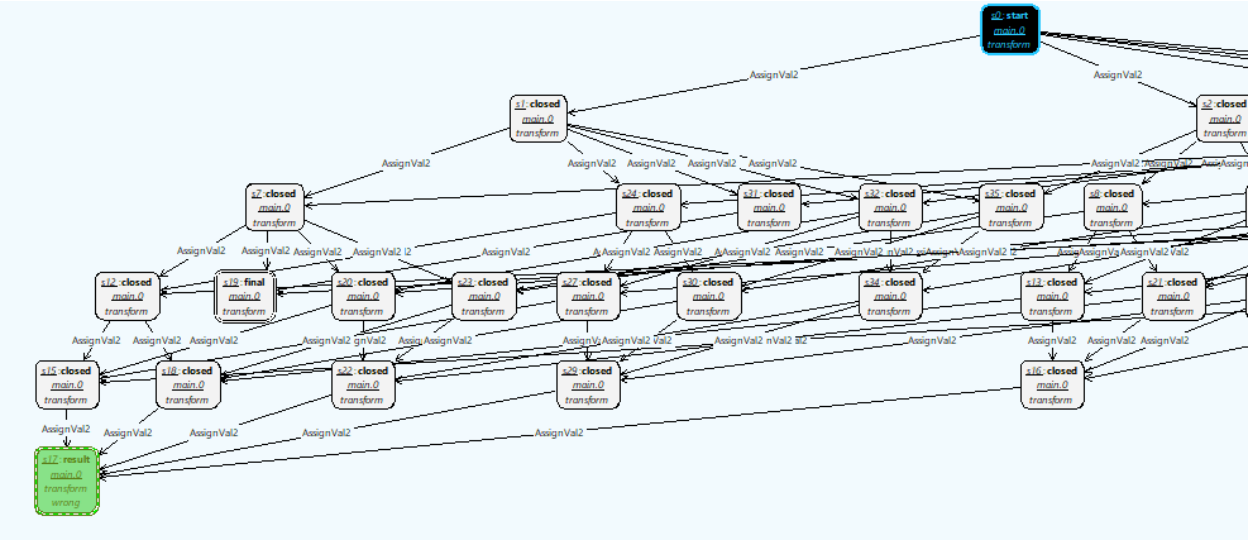


Figure 13. States of a simulation with one solution

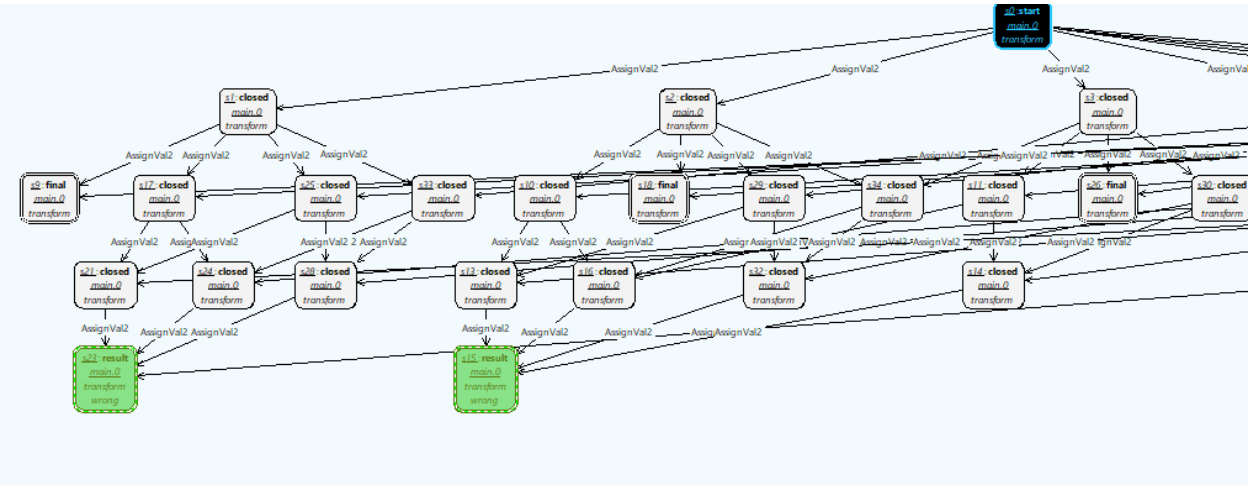


Figure 14. States of a simulation with more than one solution

In both figures 13 and 14 can be seen the winning states as green.

Both examples were really completed before running the program because it is easier to see this simulation with less states. This program scales fast in memory because there are lots of possibilities and it takes a lot of time to get to a finale state.

To get the solution faster this simulation is set to explore the states in DFS to be more possible than it gets to a solution faster than the BFS, even thought that it always terminate all the possibilities to see if there is another solution for the problem.

Pros and cons

	Undirected graph	Directed graph with edges between cells	Directed graph with edges pointing to groups
9x9 with numbers as values	YES	YES	YES
9x9 with letters as values	NO	YES	YES
Not 9x9 with numbers as values	NO	YES	YES
Not 9x9 without numbers as values	NO	YES	YES
Rows and columns represented by letters	NO	YES	YES
More than 3 groupings	NO	NO	YES
Not square shaped	NO	NO	YES

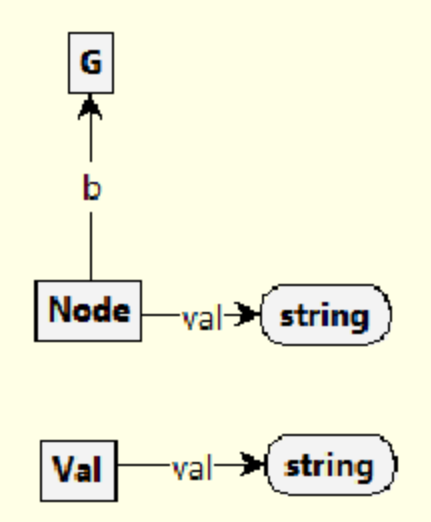
With this table can be seen that the most flexible solution is the Directed graph with edges pointing to groups, and that is because the rule system gives the start graph to increase in number of groups, change the shape of the game and let strings to be the values, and still can be solvable.

The reason because this approach is highly generalizable solution is because is the most abstract one. It does not recognize positions, just groups, and that changes the problem completely because there is not anymore a square grid with cells inside so it is just nodes with groups in common.

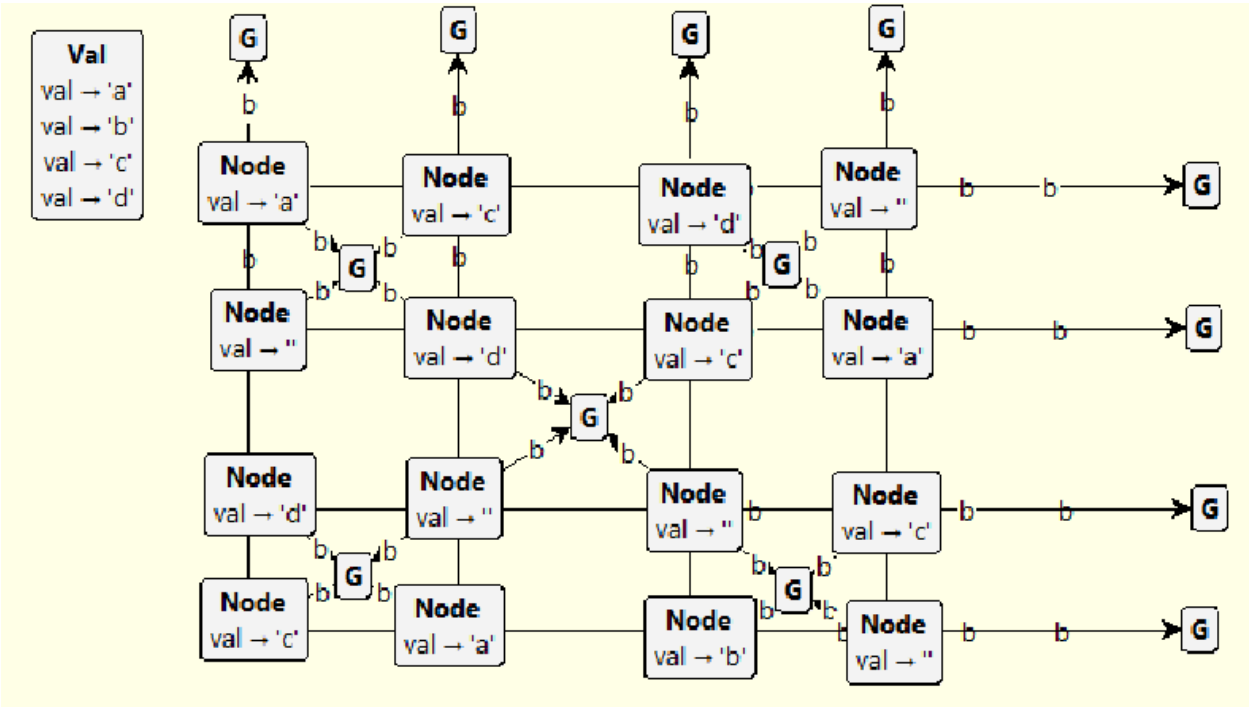
It is true that in GROOVE does not let it to be this highly generalized completely because it needs to also change the type graph depending if the solution is with integers or strings and, because of that, it no longer lets it the empty node be 0 (integer) and needs to be changed to ‘’ (string). But once this condition is changed it solves it too.

Example of 4x4 with more than 3 groups (the four cells in the center need to be different too) with strings on values:

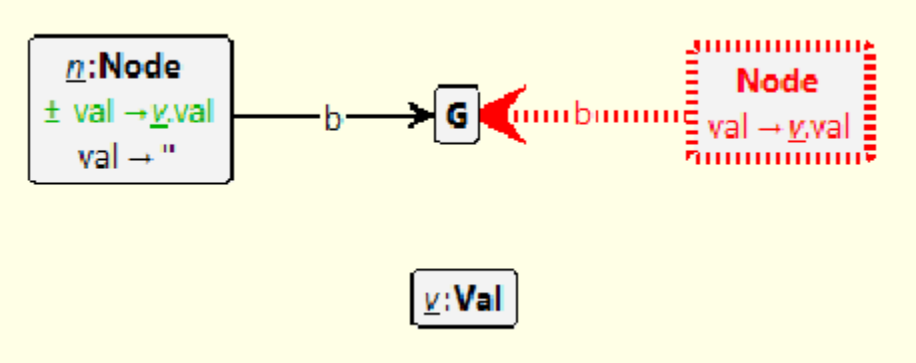
1. Type graph:



2. Start graph:



3. Rule system:



4. Wining graph condition