Dakota Bryan

Graph Theory Final Paper

12/9/22

https://dbryan17.github.io/sudokuGraph/

Sudoku Graphs: Visualizing, Playing, and Solving – a Web Application

Inspiration:

For as long as I can remember, I've been someone who has loved puzzles and who tries to deeply understand every puzzle I solve. Not only in the way of finding effective and efficient strategies, but also by understanding the often abstract, mathematical nature of the puzzle. Lately (as in the past few years) I have been solving sudoku's almost daily, and have had fun pondering and researching the group theory representations and applications of sudoku puzzles. Graph theory has given me a new framework to understand sudoku's, and when it came time for the final project, this immediately came to mind. After doing some research, I found a short Wikipedia page on the subject, a couple papers, but no concert examples (solving a sudoku as a graph) programmed and available on the internet. I took this as a chance to program something that was truly unique. I had already built a sudoku web application about a year ago that I've been using to provide the Hill News weekly puzzles. If it weren't for this prewritten code, the scope of my project would have far exceeded the time I had for it. Even still, I wrote well over 1,000 additional lines of dense code. Due to the large amount of time I spent doing this and the fact that my project is a software engineering one and not a research one, my paper will be relatively short and just provide a brief overview of the subject and my application.

Sudoku Overview:

A sudoku is a logic puzzle that takes the form of a nine-by-nine grid. In this grid, there are some provided numbers (one through nine), and the objective is to fill the rest of the grid with the numbers one through nine such that no three-by-three outlined block, row, or column has a repeated number. For it to be a valid board, this must be possible, and there must be only one solution, otherwise, it would be ambiguous. One solves the puzzle by applying logical deductions to place numbers. For example, if in any given three-by-three block, every cell but one has any given number not placed in the three-by-three block yet (say two) in its row or column, then the two can be placed. The same logic applies for a row or a column. For a row, a number could be placed if it is in the column and/or block of all other cells in the row. This strategy will be known as “one option” for the remainder of the paper. An example of this logic is given below (for a block, it also works for the first column).

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

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

(These pictures and all others come directly from my application)

Another strategy, “single candidate”, is done by checking if a given cell has all numbers already placed in its block, row, and/or column. An example of this strategy is given below.

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

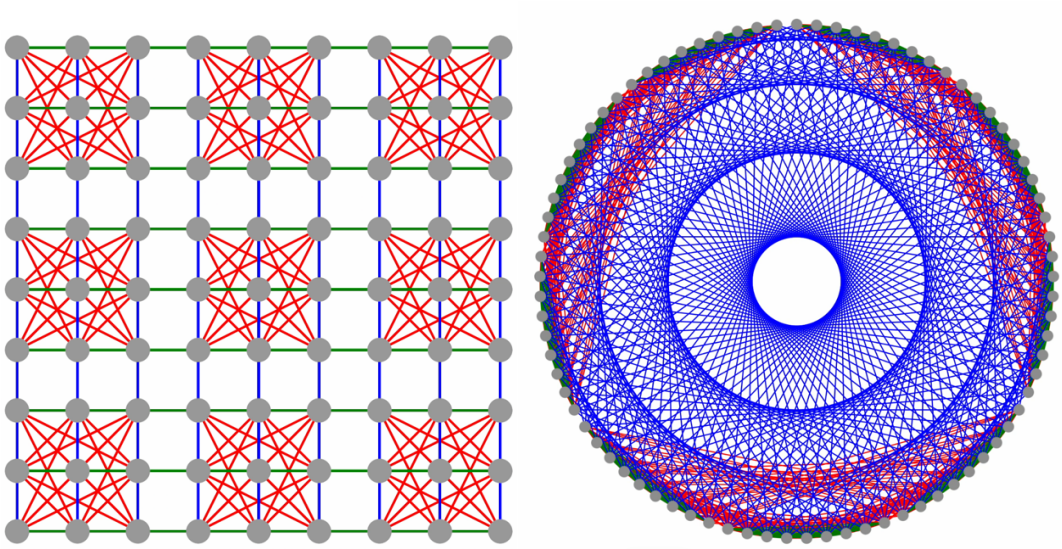
There are many strategies one can employ while solving a puzzle, but these two are the most common, and can usually allow one to solve a “hard” level puzzle. They are also the strategies I programmed into the application for the “help” feature.

Sudoku Graph Overview:

Due to the nature of the puzzle, it can easily be represented as a graph. This graph, known as the “Sudoku graph”, has 81 vertices and 810 edges. (https://en.wikipedia.org/wiki/Sudoku_graph) The vertices represent cells (clearly nine times nine = 81 of them), and an edge between two vertices represent that those two vertices cannot be the same “number”. These “numbers” can really be any symbols, as long as there are nine of them. Since there is an edge between every vertex that can’t be the same symbol, we can count the number of edges by way of degrees. Any given cell cannot be the same number as any cell in the row (8 others), column (8 others), and block (4 unique others). This adds up to 20, so each vertex will have degree 20, making the graph 20 regular. By FTGT, (20*81)/2 is 810, giving us the number of edges in the graph. Another notable structure of the sudoku graphs is its cliques. A

clique is a complete subgraph, of which there a 27 with 9 vertices in the sudoku graph. Each block, row, and column is a 9-vertex-clqiue because no vertex in any one of those sets can be the same symbol as any other, giving us an edge between every vertex, thus a complete subgraph. It is notable that every vertex is a part of three of those cliques, its row, column, and block.

Below is the Sudoku graph in a grid layout and also a circular layout. The edges are colored based on if it is an edge connecting a block, row, or column. The block overrides others, so if an edge is representing the same column and same block, it will be colored as per the block. In both layouts, there are edges covered up by other edges, but more so in the grid layout.

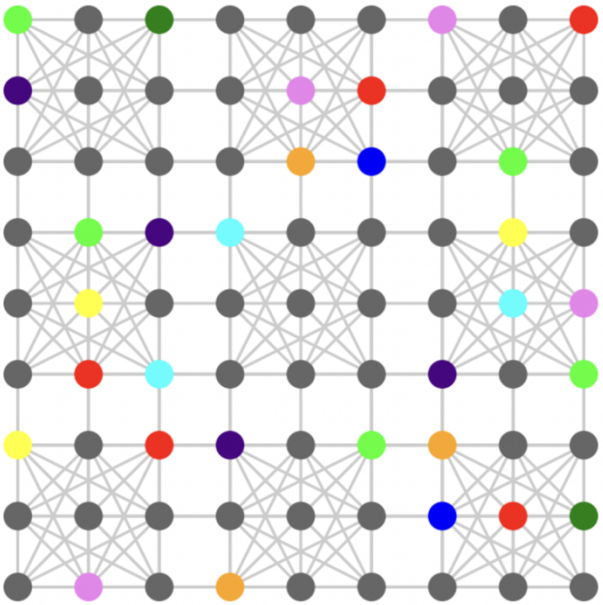


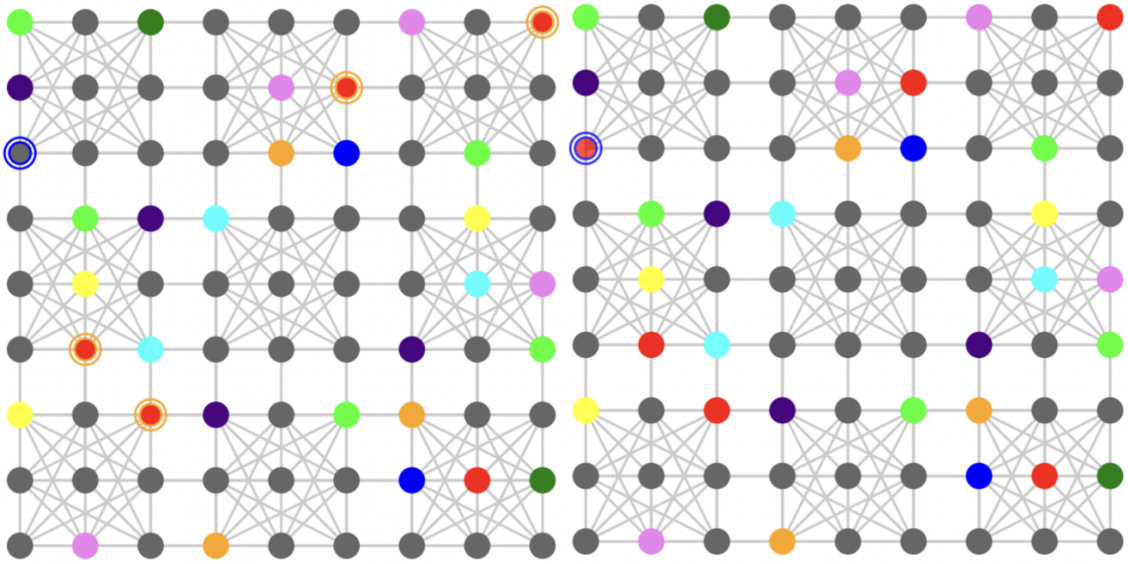
Beyond representing the "idea" of a sudoku, the sudoku graph can represent individual puzzles through colorings. The color of a vertex relates to the number on the puzzle grid at the given cell. Now, thinking about, and the act of, solving a sudoku can be done purely through the graph. This is a powerful fact, as the graph is not just a way to understand the conceptual idea of a sudoku, but once colored, can be any given puzzle. Newspapers could start printing graphs instead of grids, and it would be the exact same puzzle. The solution to any given graph puzzle is the unique coloring of the graph with 9 colors. It must be unique since it is a valid sudoku. This means that sudokus can be solved with a backtracking graph coloring algorithm. The solving

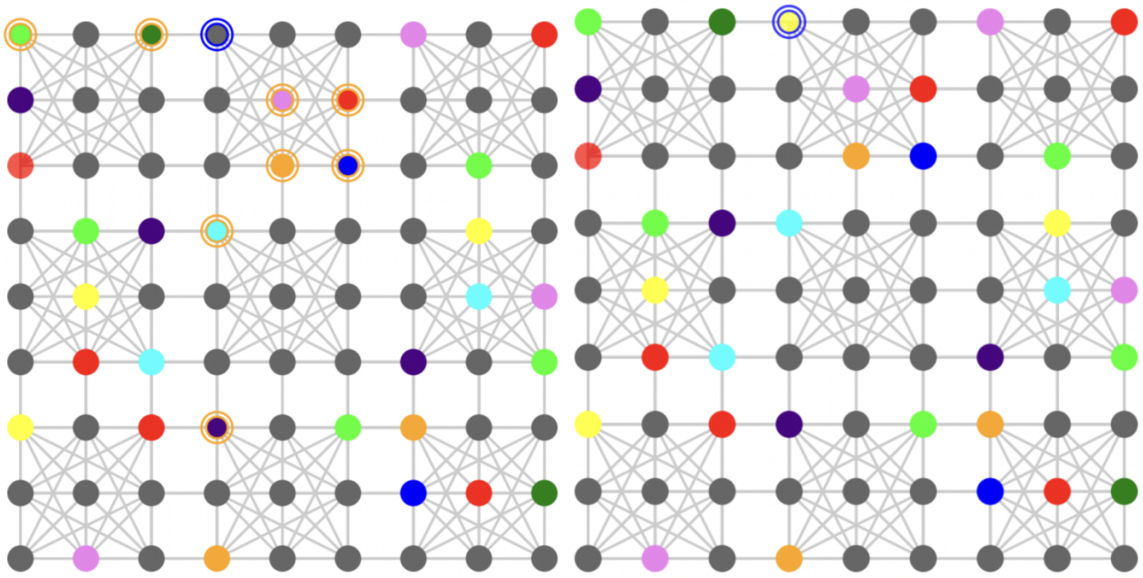
strategies outlined earlier for the grid can be translated to solving (or coloring) strategies for the graph. The “one option” strategy translates to checking if in any given clique, if all vertices in that clique except for one is adjacent to a color not found in the clique. If so, that one vertex that is not adjacent to the given color can be colored. In my opinion, this is a better definition than the one offered for the grid, as the concept of clique allows us to abstract the “one option” for a row, column, or block to just one definition. The “single candidate” strategy for the graph involves checking if any vertex is adjacent to every color but one. I found these similarities astonishing, as it shows how the sudoku puzzle is a deep mathematical concept that can be represented in other ways than just the one that is offered by newspapers. Perhaps, graphs can allow for research to be done on sudokus that offer insights into the nature of the puzzle that the grid representation hides.

Below are pictures of a sudoku represented as a grid and a graph, and the two strategies discussed. It easy to confuse the strategies for the graph as the same (in implementation not in the true abstract sense) as the ones for the grid because of how the graph is laid out. One must remember the hidden edges and what these edges represent to understand the implementation of the graph strategies.

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







Application Overview:

After spending some time doing a small amount of research, and mostly just thinking about sudoku graphs, I began modifying my existing sudoku app to add the graph representation of sudokus. My goal was to create an application that showcases the idea that the grid representation of the sudoku puzzle is just one, and graphs are a great way to represent the puzzle. The features I wanted the app to have to achieve my goal was the ability to solve the sudoku with the grid or the graph, get help for the puzzle with both the grid and graph, modify the layout of the graph by dragging vertices and changing from a grid layout to circular one (with an animation that showcases the symmetry of the graph), and an automatic solving option that shows how you can solve a sudoku puzzle with a backtracking graph coloring algorithm. I implemented all of these features into the application, made it user friendly, covered all corner cases of user input, thoroughly tested it for bugs, and made it available on the internet so anyone can use it.

It is important to note the features of the app that existed before the graph was added. This includes the board generation algorithm, the display of the grid, the "help" feature for the grid, the notes option; basically, the app as it stands with no graph and no automatic solving (aside from clicking help over and over again which may arrive at a dead end). The app is programmed in the three most common languages for websites and web apps, which are JavaScript, HTML, and CSS. JavaScript is used for the manipulation of pages, handling user actions, and all algorithms. HTML is used to structure the content of the page. CSS is used to style the contents of the page in a visually pleasing and usable way. I decided to not use any frameworks ("wrappers" of sorts for these three languages to make them easier to use, program, and interact with each other) because I thought it was not needed for the size of the project. (I was wrong). I did use a library called cytoscape.js (https://js.cytoscape.org/) that made it less code to display the graph. Instead of manually creating the vertices and edges with CSS, and dealing with the connections between them, cytoscape.js allowed to define a graph a list of nodes and a list of edges; then allowed me to add a visual aspect of the graph by defining parameters for how I wanted it to look (such as layout type, size of nodes, size of edges, specific styles for each node, i.e. color, etc...), and it would do the rest. The way it set up the graph also made it easy for users to be able to drag vertices around to expose edges. This is the only library I used, and all code is written entirely by me (nothing copied from the internet or other sources/people).

My code is divided into 10 different files in one folder, and this folder is hosted on GitHub pages which allows it to be on the internet. One file is an HTML file, one is CSS, and the rest are JavaScript. I will not describe the finer details of the code in this paper, as it is over 2,500 lines long and is heavily commented, but will describe the basic flow of the application and the backtracking coloring algorithm I designed. The entry point to the application is the

index.html file, which provides the initial display and loads all the JavaScript files. Once the user enters the number of given digits and clicks play or hits enter, the JavaScript files take over.

First, the number of given digits is sent to a function that will generate the board (in the generator.js file). If the number of digits is over 24, the sudoku is generated on the fly. If it is under 25 digits, it takes too much time to generate the puzzle on the fly, so it chooses from a random puzzle that I pre-generated with my algorithms. If it is under 21, it will probably crash because that few digits would probably take my algorithm days to get too, and I couldn't get any to generate that low. An interesting note is that the fewest possible digits for a sudoku is 17 (link to paper on the subject). Once the puzzle is generated or fetched, another file (sudoku.js) generates the board display, populates it, creates the buttons, adds "event listeners" (these listen for click and keyboard events and provide functions that handle these events), changes the webpage to display it, and calls a function to generate the graph from the puzzle (in graph.js). The graph is generated with the help of cyptoscape.js, then displayed on the webpage. All the buttons and event listeners are added for the graph in this file as well. If there is an error made in the puzzle, a file (errorsAndHelp.js) is called which highlights, unhighlights, and deals with all the logic pertaining to the highlighting of vertices and cells. This file is also called when the help button is pressed, and this file calls another file, strats.js, which includes all the human solving strategies that I programmed. When the user presses the button to solve the puzzle with graph colorings, another file (grahSolve.js) is called that deals with all the logic pertaining to the auto-solving, and the coloring algorithm itself.

Coloring Algorithms Overview:

I implemented two backtracking color algorithms that will solve the sudoku graph. The only difference between the two is one colors nodes in simply the order they are in (in grid layout – upper left to lower right), and the other colors the nodes in the order of how many options there are for that node. The second one is much more efficient, so I would provide an in-depth overview of it. The code and comments are given below, I expanded on the comments to be an explanation of the algorithm.

```
// backtracking coloring algorithm that is more efficient - colors the node with
fewest options first
function backtrackingFewestFirst() {
  // find all uncolored nodes by using JavaScript's .filter function. If the number
parameter of a node = "", then that node is only uncolored
  let unclored = cy.nodes().filter((node) => node.data("number") === "");
  // check if all are colored
 - then this board is completely correct always if the user didn't edit the board at
all because then there would be exactly one solution, but since the user could have
edited…
 - there maybe be no solution which, based on how this algorithm works, would mean
that the board is fully colored but not correct. This is because the algorithm will
not re color vertices that the user entered. So, either a vertex could have no options
for a color, or one before could have been wrong so it can color all the remaining
ones correctly, so need to check the graph
  if (unclored.length === 0) {
    // now check if it is correct and return result - so if it is correct, this
function will return true, otherwise this function will return false
    return checkGraph();
  }

  // find the "color" possibilities for all of the uncolored nodes by going through
each one with the .map function (takes an array and transforms it)
  let possibleNodes = unclored.map((node) => {
    // get the possible colors from a function I wrote (not included here because it
is pretty basic
    let colors = getPossibleColors(node);
    // make a new object which is the node, possible colors for the node, and the
length of the possible colors. This will be return, so the new array will be an array
of these objects
    let newObj = {
      node: node,
```

```
      possiblities: colors,
      length: colors.length,
    };
    return newObj;
  });

  // sort by fewest options first
  let sorted = possibleNodes.sort((a, b) => {
    return a.length - b.length;
  });

  // outer backtracking loop - goes through nodes left to color
  for (let i = 0; i < sorted.length; i++) {
    // get the node object
    let node = sorted[i]["node"];
    // get the possible colors
    let possibleColors = sorted[i]["possiblities"];
    // inner backtracking loop - goes through options for the node
    for (let j = 0; j < possibleColors.length; j++) {
       // gets a “random” color to color this node with from the options
      let color = possibleColors[j];
       // increase the count that will be the “steps” this algorithm took
      count++;
      // "color" the node
      node.data("number", color);
      // push onto animation queue the style, so we can animate this when we are done
      aniQueue.push({
        node: node,
        style: {
          "background-color": `${color_map[node.data("number")]}`,
          opacity: 0.8,
        },
      });
      // backtrack and return true if it is true. So this calls the function again,
and will restart all of the above with a “new graph” since the node is colored.
      if (backtrackingFewestFirst()) {
        return true;
      }
    }

    // didn’t work since we went through all the possible colors. Reset node and push
the "blank" style onto animation queue. We need to reset this node, since no color
worked. This means we have to backtrack and recolor a previous node.
    node.data("number", "");
    aniQueue.push({
      node: node,
      style: {
        "background-color": "#666",
```

```
        opacity: 1,
      },
    });
    // return false so we know to uncolor
    return false;
  }
}
```

The only difference between this algorithm and the one that is less efficient is there is no sorting of nodes to color. This is the only algorithm that I will provide an in-depth explanation of in this paper, but the rest should be easy to understand in the code.

Challenges and Takeaways:

I had a lot of challenges when programming this application. The main one was that there was a lot of buttons, inputs, and possible combinations of things users could do, which made it difficult to manage the state of the app, handle all of the combinations properly, and test the code. What would have made this much easier to manage is a framework such as React.js, and if I were to do this over again, I would start with that. That was really the only big challenge, as I found the implementation of the algorithms quite fun, and the rest of coding also fun and interesting. This one challenge was a big one though. Testing my code took a long time, and every time I implemented a new feature, I had to make a lot of functions that just handled all the possible combinations of inputs that arose from this one feature. One other thing that was somewhat of a challenge but mostly a takeaway was the fact that everything I was coding could be done in graph data structures or array data structures. They really did the same thing, but the representation of the sudoku could either be arrays (grid representation) or a graph (graph representation). For example, when I was coding the help feature, doing the strategies for a grid or graph did the same thing, so there was no reason to code it twice. The somewhat unfortunate outcome of this is that the app is mostly coded with grid logic, then just passed to the graph.

However, it could be the other way around, and I think it may be easier to code some human strategies with the graph. If I ever continue to work on this project by coding more strategies, I will do so with the graph representation. I think that will make them easier to work with a program.

Works Cited

- https://en.wikipedia.org/wiki/Sudoku_graph
    - This is the Wikipedia page I used to get a brief overview of the Sudoku Graph. I worked out everything I found on the page myself, but was a good place to start.
- https://js.cytoscape.org/
    - This is the documentation for the graph library I used
- https://digitalcommons.sacredheart.edu/cgi/viewcontent.cgi?referer=&httpsredir=1&article=1089&context=acadfest#:~:text=A%20stack%20is%20the%20set,%2C%20or%20columns%207%2D9.&text=Gary%20McGuire%20of%20the%20University,unique%20solution%2C%20which%20is%2017
    - This is a paper that states that the fewest numbers a sudoku can have is 17. I didn't use this source but couldn't remember where I first heard the 17-number theorem, so included it to back up that claim.