Solvoku

Sudoku Solver Design Document

Name: Brian Tran
Course: MET CS 664-D1 Artificial Intelligence
Semester: Fall 2014

Date: 12/04/2014

Table of Contents

1. Introduction

1.1 Purpose

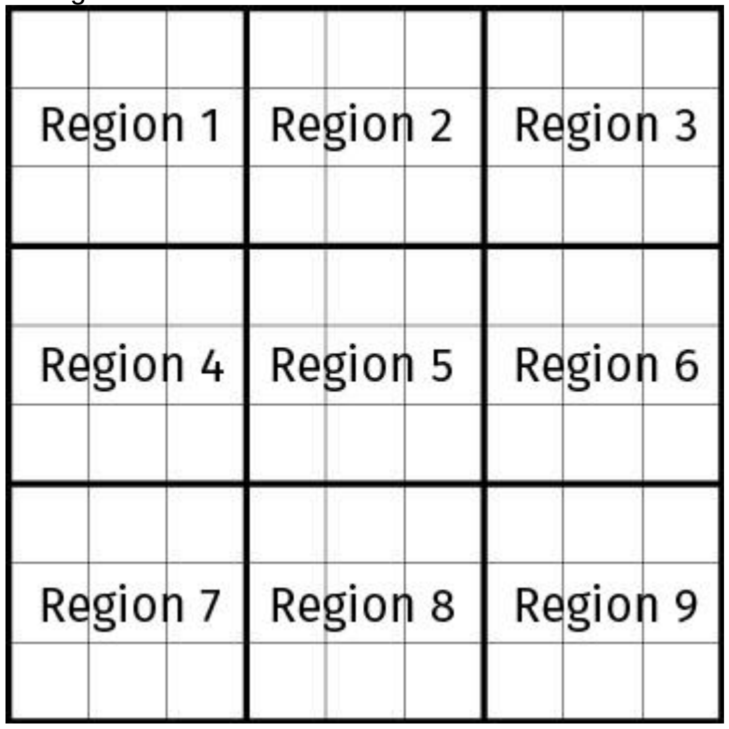
The purpose of this project is to implement a facet of Artificial Intelligence. The problem I chose to focus on is solving Sudoku boards by using Artificial Intelligent algorithms including Backtracking and algorithms that solve Exact Cover problems.

1.2 Project Proposal

- **Computer Platform:** Unity development exported to Android and Web Player.
- **Programming Language:** C#
- **Project Description:** Mainly focus on concepts of deduction, reasoning, problem solving to figure out the Sudoku Solver.
- **What I will learn from this project:** Specific techniques in terms of solving Sudoku puzzles including Backtracking, Exact Cover, Stochastic Search, and Constraint Programming. I will learn the different techniques and develop a solver for one of them.
- **Potential Implementation Problems:** Figuring out what is the most efficient technique for development and how to integrate it on a mobile device.

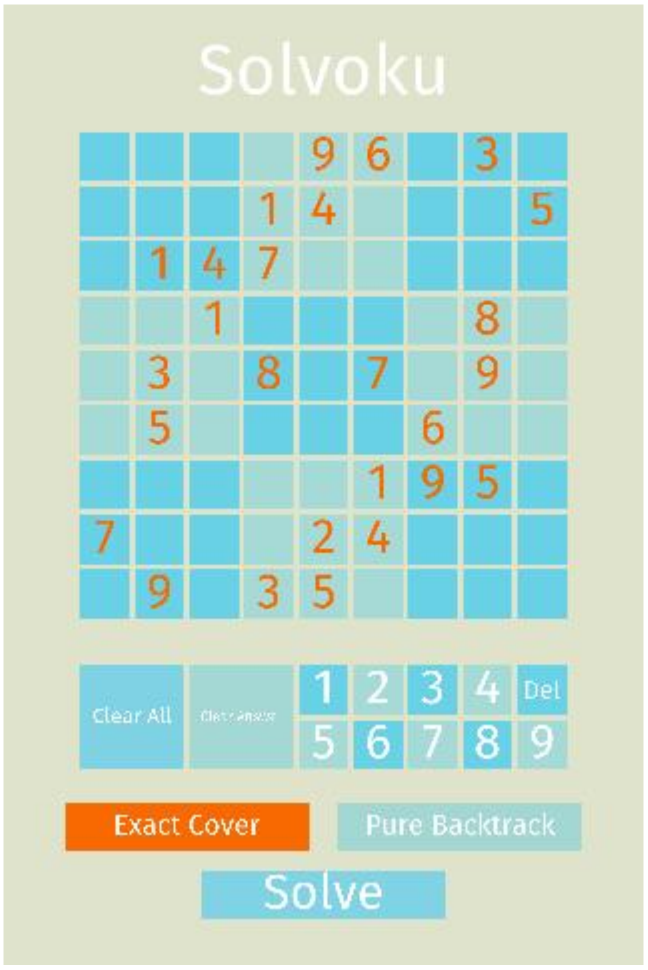
1.3 Definitions

- **Sudoku region** - A Sudoku region is the 9 quadrants that make up 9 cells as shown in this figure:



2. User Interface Design

2.1 Overview of User Interface



The primary goal of the Solvoku App user interface was to make it as easy as possible to fill the Sudoku cells with the appropriate values. Once the cells have been prepared, the "solve" button is clearly presented as a large button on its own row.

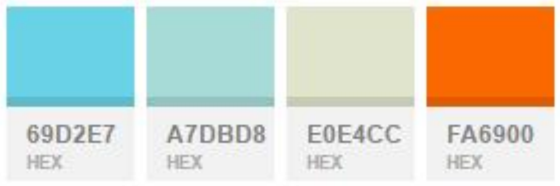
Two clear buttons are accessible to the user. The "Clear All" button clears every single cell and the "Clear Answers" button clears only the answers and keeps the setup.

The input layout is setup primarily for right-handed users by having the "clear" buttons as far away from the right thumb as possible so no accidental clears can happen.

The user can choose between the two solving algorithms: exact cover and backtracking. The selected algorithm is highlighted in orange.

Once the solution is found, a message pops up informing the user on the algorithms efficiency.

2.2 Color Scheme



Decided to use a pastel color scheme, using the blues as the cell backgrounds, white and orange for the cell text, and the off white color for the application background.

2.3 Font

Used "Fira Sans Regular" font for this project, which is coincidentally the Firefox OS Typeface.

3. System Architecture

3.1 Overview and Unity Engine

The mobile application was developed using the Unity Engine, a cross-platform game creation system. The engine can handle three languages for development: C#, JavaScript, and Boo. I used C# to develop this project.

The simple architecture has only three core parts to it: the main application, the Sudoku board, and the Sudoku solvers.

3.2 Main.cs

The main application is handled within this class. It handles all button input from the user and passes information along to the Sudoku Board. The class also handles error messaging.

3.3 SudokuBoard.cs and SudokuSlot.cs

Sudoku Board visually displays the Sudoku board, which is made up of 81 Sudoku Slots. The Sudoku Board visually updates the cells as input is from the Main is passed into it. When the Sudoku Board's "solve" function is triggered by user input from the main, Sudoku Board passes a single array of integers to the specified solver to solve.

3.4 Backtracking Solver (BacktrackingSolver.cs)

The first implementation of the solver I did was the cell specific backtracking solver. The solver will go to the first empty cell, place a value, and then check if the cell is valid. If it's valid, then the solver will continue recursively to the next empty cell, place a value in that one, and check its validity. If a cell is not valid, the algorithm will reset the cell and then BACKTRACK to the previously valid scenario. The following is the code for the backtracking solver:

```
public static void solve(int[] cells) {
    backtrack(cells, 0, 0);//pass the board and start at 0,0
}
public static bool backtrack(int[] cells, int row, int col) {
    if (col >= 9)//reached last column of row
        return backtrack(cells, row + 1, 0);//go to next row
    if (row == 9)//reached the last row
        return true;//we're done! Recurse back up.
    int cellId = getCellAt(row, col);//get index of cell
    if (cells[cellId] == SudokuBoard.EMPTY_CELL) {//cell is empty
        for (int i = 1; i <= 9; ++i) {//insert values from 1-9
            cells[cellId] = i;//set cell with value
            if (validCell(cells, cellId)) {//cell is valid!
                if (backtrack(cells, row, col+1))//recurse to next col
                    return true;//successful full chain!
            }
        }
        cells[slotId] = SudokuBoard.EMPTY_CELL;//failed. reset cell
    } else {// cell is not empty
        return backtrack(cells, row, col + 1);//recurse to next col
    }
    return false;//this scenario failed, backtrack out
}
```

3.5 Exact Cover Solver with Algorithm X and Dancing Links (SudokuDancingLinks.cs)

3.5.1 Overview of Algorithm X and Dancing Links

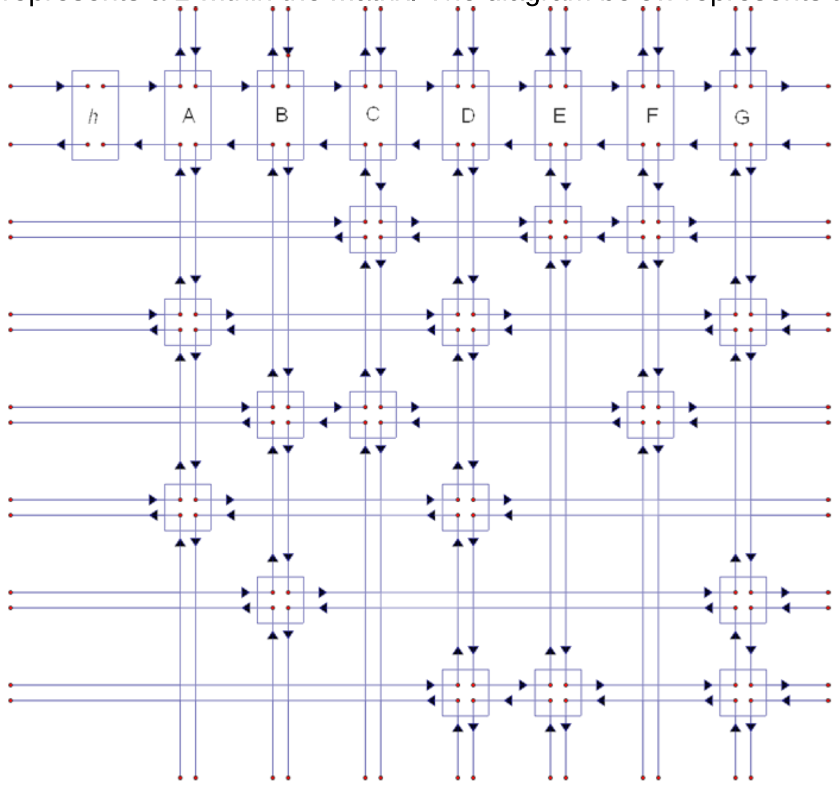
The second implementation of the solver uses Knuth's Algorithm X, which finds all solutions to an exact cover problem, such as Sudoku. The most efficient way to implement Algorithm X is Donald Knuth's Dancing Links technique.[1]

Given a matrix of 1's and 0's, Algorithm X will find a set or more of rows that will have exactly one 1 in each column. Referencing Knuth's paper is the following matrix:

0 0 1 0 1 1 0
1 0 0 1 0 0 1
0 1 1 0 0 1 0
1 0 0 1 0 0 0
0 1 0 0 0 0 1
0 0 0 1 1 0 1

The rows in bold will be the three rows chosen by Algorithm X because each column is covered by only one 1.

The Dancing Links technique takes advantage of doubly-linked lists and uses the linked lists to represent the matrix. The headers keep track of the matrix top and each node that exists represents a 1 within the matrix. The diagram below represents the matrix shown earlier:

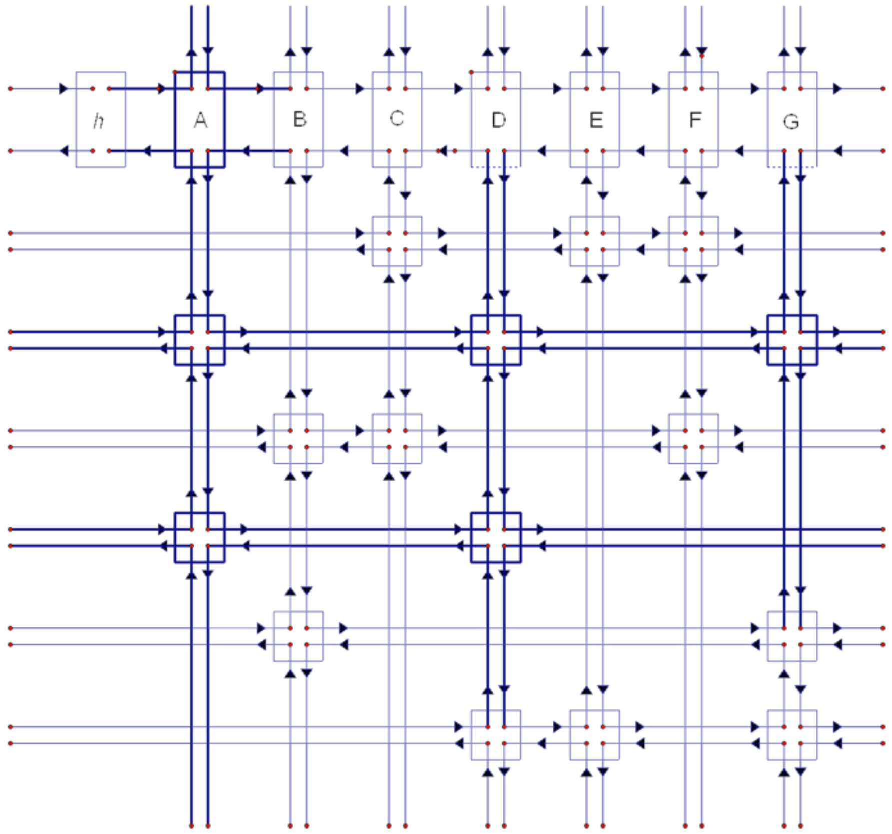


1 https://www.ocf.berkeley.edu/~jchu/publicportal/sudoku/0011047.pdf

3.5.2 Dancing Links and Backtracking with "Cover" and "Uncover"

The algorithm uses backtracking to find the rows that fit the solution. However, unlike the backtracking algorithm discussed earlier, this backtracking algorithm "covers" and "uncovers".

"Cover" is the main algorithm that removes a column from the matrix and removes all nodes within the same row. The diagram below highlights the A column being removed as well as all nodes connected to the A column:



The "cover" method is used similarly to how the pure backtracking algorithm inserted a value into a cell; we "cover" a column and continue covering columns until we have no more columns to cover. If we have covered all columns, then a solution has been found. If all columns were not covered, then the algorithm will backtrack and "uncover" a column, which means insert it back into the dancing links.

"Uncover" function works because the column that was removed still remembers its original links within the matrix.

3.5.3 Algorithm X and Sudoku

In Sudoku, the rows and columns of the Dancing Links matrix are represented in a unique way. The columns represent the constraints of the puzzle. In Sudoku, there are four constraints:

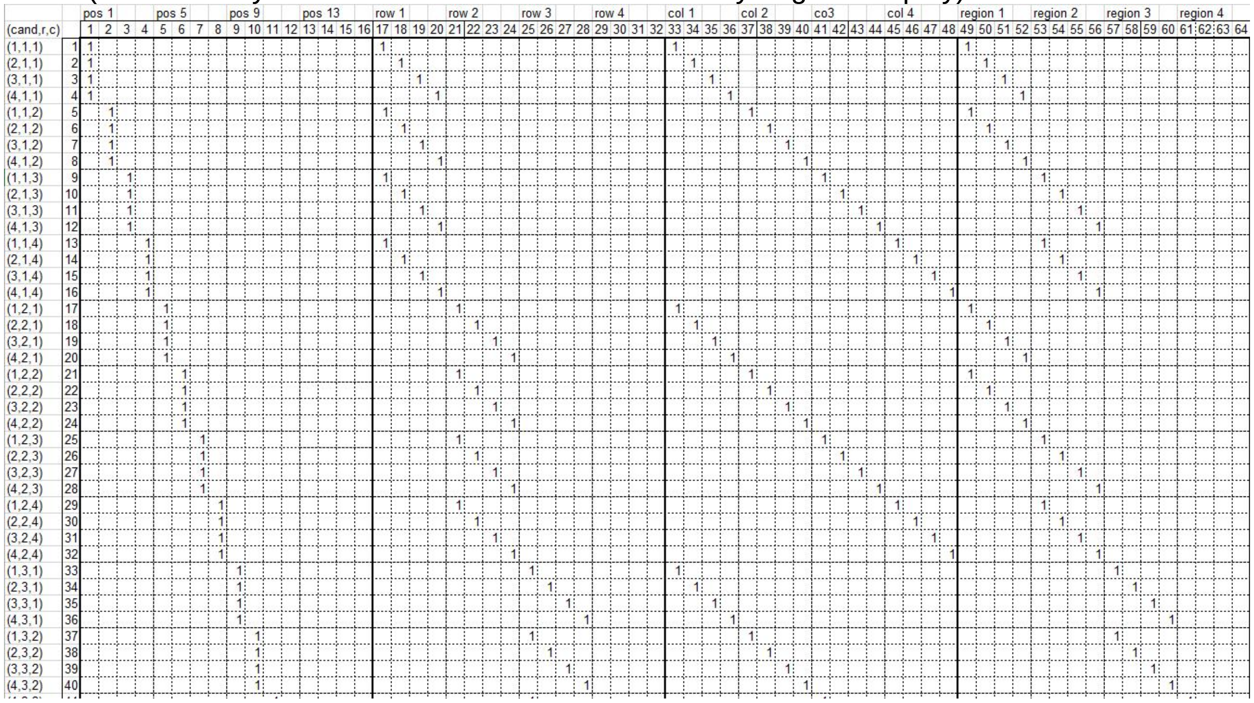
1. **A position constraint:** Each cell can contain a single digit (81 constraints).
2. **A row constraint:** Each row can contain 1 of each digit (9 rows x 9 digits = 81 constraints).
3. **A column constraint:** Each column can contain 1 of each digit (9 columns x 9 digits = 81 constraints).

4. **A region constraint:** Each region can contain 1 of each digit (9 regions x 9 digits = 81 constraints).

A total of 324 (81 * 4) constraints need to be satisfied so there are 324 columns within our matrix.

To make sure we have an exhaustive list, we must cover all 9 digits in 81 cells. Therefore we have 729 rows, which represents every single possible candidate within a Sudoku board.

The following is a glimpse of an exhaustive list of every possible constraint for a 4x4 Sudoku board (unfortunately a 9x9 Sudoku board is unrealistically large to display):



		pos 1				pos 5				pos 9				pos 13				row 1				row 2				row 3				row 4				col 1				col 2				co3				col 4				region 1				region 2				region 3				region 4			
(cand,r,c)		1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63	64
(1,1,1)	1	1																1																1																1															
(2,1,1)	2	1																	1																	1																	1														
(3,1,1)	3	1																		1																		1																		1													
(4,1,1)	4	1																			1																			1																			1												
(1,1,2)	5		1															1																				1												1																	
(2,1,2)	6		1																1																					1													1														
(3,1,2)	7		1																	1																						1														1													
(4,1,2)	8		1																		1																							1															1												
(1,1,3)	9			1														1																								1																1											
(2,1,3)	10			1															1																									1																	1										
(3,1,3)	11			1																1																										1																		1									
(4,1,3)	12			1																	1																											1																				1								
(1,1,4)	13				1													1																												1															1											
(2,1,4)	14				1														1																														1															1										
(3,1,4)	15				1															1																														1																	1									
(4,1,4)	16				1																1																																1																			1								
(1,2,1)	17					1																1												1																1															
(2,2,1)	18					1																	1													1																	1														
(3,2,1)	19					1																		1														1																		1													
(4,2,1)	20					1																			1															1																			1												
(1,2,2)	21						1															1																1												1															
(2,2,2)	22						1																1																	1													1														
(3,2,2)	23						1																	1																		1														1													
(4,2,2)	24						1																		1																			1															1												
(1,2,3)	25							1														1																				1																1											
(2,2,3)	26							1															1																					1																1										
(3,2,3)	27							1																1																						1																1									
(4,2,3)	28							1																	1																							1																1								
(1,2,4)	29								1													1																								1														1											
(2,2,4)	30								1														1																										1															1										
(3,2,4)	31								1															1																										1																1									
(4,2,4)	32								1																1																												1																	1								
(1,3,1)	33									1																1								1																								1							
(2,3,1)	34									1																	1									1																									1						
(3,3,1)	35									1																		1										1																										1					
(4,3,1)	36									1																			1											1																											1				
(1,3,2)	37										1															1												1																				1							
(2,3,2)	38										1																	1													1																					1						
(3,3,2)	39										1																		1														1																						1					
(4,3,2)	40										1																			1															1																							1				

The first 4 rows represent an exhaustive list of having the values 1, 2, 3, and 4 in the top left slot. All empty cells are considered 0's.

The algorithm will first "cover" all columns that are part of the original setup. Then it will use the backtracking algorithm to "cover" and "uncover" until a solution is found. The following is the main algorithm to find the solution:

```
private void search(int depth)
{
    if (_root.right == _root)
    {
        <...storing solution...>
        return; // all columns have been used. we've found a solution! Store and kick out.
    }

    HeaderNode column = chooseNextColumn(); // choose next column to cover
    cover(column);

    for (Node rowNode = column.down; rowNode != column; rowNode = rowNode.down)
    { // traverse every row until we loop around
        _solutions.Push(rowNode);

        for (Node rightNode = rowNode.right; rightNode != rowNode; rightNode = rightNode.right)
        { // traverse and cover every node going right as long as we're not the current node
            cover(rightNode.header);
        }

        search(depth + 1);

        // backtrack this step
        _solutions.Pop();

        for (Node leftNode = rowNode.left; leftNode != rowNode; leftNode = leftNode.left)
        { // traverse and uncover every node going left as long as we're not the current node
            uncover(leftNode.header);
        }
    }

    uncover(column);
}
```

3.6 Pure Backtracking vs. Exact Cover Solver Comparison

3.6.1 Pure Backtracking Statistics

When running the Pure Backtracking algorithm on an extremely difficult Sudoku board, the following stats are displayed:

- Total Function Time: 3ms
    - This is the total amount of time the function took to solve the Sudoku board.
- Total Actions: 260,005
    - This is the amount of actions the algorithm did to solve the Sudoku board. "Actions" are anything that is manipulating something and are incremented within for loops and in sections within function calls.
- Max Depth: 90
    - This is how deep the recursion went. The amount of actions directly correlates with the depth level; the deeper the depth, the more actions.

3.6.2 Exact Cover Statistics

When running Algorithm X on the same Sudoku board, the following stats are displayed:

- Total Function Time: 1ms
- Total Actions: 11,446
- Max Depth: 55

Although both algorithms find the same solution and within similar amount of time, Algorithm X is 23 times more efficient in terms of actions and does not go as deep as the pure backtracking method.

4. Conclusion

4.1 Power of Backtracking

Although Algorithm X trumps the pure backtracking algorithm in terms of finding a Sudoku solution, the backtracking algorithm is a key aspect of Algorithm X and is the crux in solving puzzles like Sudoku. The power of backtracking is crucial in solving problems like Sudoku with Artificial Intelligence, especially when used with the powerful technique of Dancing Links, which impressively manages large matrices efficiently.

4.2 Thoughts on Sudoku Solver

I initially started the project thinking that a Sudoku Solver is an easy problem to handle, especially if the algorithm is developed as a human would think. However, after researching the various methods and implementing Backtracking and Exact Cover algorithms, I am extremely impressed in the complexity of a Sudoku Solver and have a new appreciation of solving seemingly simple puzzles.

4.3 Future

The entire project is within the project submitted, including an .apk file for Android users and an web version for desktop users. A web version can also be found here:
https://dl.dropboxusercontent.com/u/6009669/solvoku/WebPlayer.html

5. References

- Backtracking
  - http://en.wikipedia.org/wiki/Sudoku_solving_algorithms
    - Although Wikipedia is not a valid source, the pseudo code under the backtracking technique was sufficient enough to help with my understanding of the algorithm.
- Dancing Links
  - https://www.ocf.berkeley.edu/~jchu/publicportal/sudoku/sudoku.paper.html
  - https://www.ocf.berkeley.edu/~jchu/publicportal/sudoku/0011047.pdf
  - http://sudopedia.enjoysudoku.com/Dancing_Links.html