Sudoku Squares’ Difficulty

Michael Swords

Degree Project in Computer Science, DD143X
Supervisor: Christian Smith
Examiner: Örjan Ekeberg

CSC, KTH 29/04/2014
Abstract

The purpose of this report is to evaluate squares in any Sudoku puzzle. The evaluation is based on how difficult the square’s number is to solve by using manual methods i.e. methods possible to use with pen and paper.

To achieve this, manual methods have been implemented as algorithms into an application. A GUI presents the results in the form of a Sudoku solution and a matrix window to present which methods were needed for each single square.

A number of Sudoku solving techniques were implemented and applied on both specific and randomly generated Sudokus. With this an evaluation of the squares could be demonstrated.

Even if the program could not show the difficulty for all squares in some Sudokus, the conditions for further development of new techniques can be made with ease which can solve these more difficult Sudokus at a later stage.
Sammanfattning

Denna rapport har som mål att värdera rutor i ett Sudoku. Värderingen baseras på hur lätt eller svårt det är att få fram rutans nummer med mänskliga metoder.

För att nå detta mål har mänskliga metoder för att lösa Sudokus implementerats som algoritmer i en application. Ett GUI presenterar resultat i form av sudoku lösning samt vilka metoder som krävdes för att lösa varje enskild ruta.

Ett antal tekniker för att lösa sudokus implementerades och med både specifika och slumpmässigt genererade sudokus kunde det påvisas att en värdering kunde göras.

Även om programmet inte lyckats få fram svårighetsgraden för alla sorters rutor i vissa sudokus så har förutsättningar skapats för att vidare utveckla tills alla rutor kan graderas.
# Table of Contents

1. Introduction .................................................................................................................. 5
   1.1 Problem statement ...................................................................................................... 5
   1.2 Definitions .................................................................................................................. 5

2. Background .................................................................................................................... 6
   2.1 Sudoku explained ....................................................................................................... 6
   2.2 Sudoku solved .......................................................................................................... 6
       2.2.1 Conventional techniques using pen and paper .................................................. 6
   2.3 Sudoku grader .......................................................................................................... 9

3. Method .......................................................................................................................... 10
   3.1 Design ....................................................................................................................... 10
       3.1.1 Technique Assessment ..................................................................................... 11
   3.2 Implementation ......................................................................................................... 12
       3.2.1 Structure ........................................................................................................... 12
       3.2.2 Algorithms ........................................................................................................ 13
   3.3 Testing ...................................................................................................................... 15

4. Results .......................................................................................................................... 16
   4.1 Test Runs .................................................................................................................. 16
       4.1.1 Random Generated Sudokus ........................................................................... 16
       4.1.2 Specific Sudokus .............................................................................................. 18

5. Discussion ..................................................................................................................... 21
   5.1 Verification ............................................................................................................... 21
   5.2 Method Choice ......................................................................................................... 21
   5.3 Comparison .............................................................................................................. 21

6. Conclusions .................................................................................................................. 22
   6.1 Other applications ..................................................................................................... 22
   6.2 Further development ............................................................................................... 22

7. References .................................................................................................................... 23

8. Appendix ....................................................................................................................... 24
   8.1 Algorithms ............................................................................................................... 24
       T1 ............................................................................................................................... 24
       T2 ............................................................................................................................... 26
       T3 ............................................................................................................................... 27
       T4 ............................................................................................................................... 28
1. Introduction

Sudoku is a well-known puzzle that can be found in almost every newspaper and bookshop. Despite its Japanese name “Su” meaning number and “Doku” meaning single, its origin is actually Switzerland and not Japan [1]. It was Leonhard Euler that created the basics of the puzzle during the decade 1740 using letters instead of the numbers we use now. Although it was created early the puzzle didn’t get popular until 1984 where it was introduced in the Japanese magazine *Monthly Nikolist* [1].

With its fame came immense research in optimum ways of solving the puzzle, with different techniques emerging both for the human mind and the computer.

Every Sudoku has a difficulty level and there has been plenty of discussion on how one would grade the difficulty of puzzle, should it be evaluated after how many numbers are given and/or their positioning, or after the amount of different developed techniques that must be used in order to solve it. Although these grading systems cover how difficult the whole Sudoku is, this paper will focus on how to evaluate a concealed square’s difficulty.

1.1 Problem statement
Problem: How to evaluate a square in a Sudoku, with its evaluation based on the amount of effort required to reveal its number.

Goal: Create an algorithm that counts the amount of different Sudoku techniques that are needed to obtain a square’s number. Since there can be several different ways to obtain the number the easiest path has to be found, where easiest being the path requiring the simplest and fewest techniques. This requires comparison of the different techniques and assessing how much more complex one is over the other.

1.2 Definitions

*Square*: Either contains a number one to nine or is empty.

*Box*: three by three grid of squares.

*Sudoku*: nine by nine grid of squares, with 9 boxes

*Candidate*: Number that is valid in a square.

*GUI*: Graphic user interface
2. Background

2.1 Sudoku explained
The classical Sudoku puzzle consists of a 9x9 grid of boxes, which is then subdivided into nine 3x3 grids. Each box can hold a number ranging from one to nine and the goal of the puzzle is to fill every box with a number. Every 3x3 grid, every row and every column has to include all of the numbers one to nine. Meaning no row, column or 3x3 grid can contain more than one of the same number.

![Sudoku Grid](image)

**Figure 1. Sudoku Explained [3].**

2.2 Sudoku solved
There are many ways of solving Sudokus both by hand and by computer. Some examples of computer algorithms are stochastic search, backtracking and brute-force [8, 9]. These techniques are impossible to do by hand as they take too long to replicate.

2.2.1 Conventional techniques using pen and paper.
There are several conventional techniques used for Sudoku I will cover some of them here but for further reading see ref [2, 3, 4, 5, 6, and 7]

**Single Position**
By systematically going through the missing numbers from a row, column or a box and checking for valid placement one can find a number with only one position available. E.g. In figure 1 the third box has ruled out positions for the number 1 leaving only one valid position (i,3). [2]
**Single Candidate**

Looking at single squares and determining if only one number can have that position. E.g. By ruling all but the number two in square (b,4). [2]
Double Pair
The method for Double pair is to find two pairs of candidates for a number as you can see in Figure 4. The 2s in row one and five makes it necessary for the number 2 to be in column four or six since both box two and five has to include a 2. This rules out three 2s in box eight giving square (d,9) only one remaining candidate, 7. [3]

Figure 4. Double Pairs [3].

Naked Pairs/Triples
This time find a single pair Figure 5 has the pair 1, 5 in column two and seven, making it impossible for 1 or 5 to be in any other column. If 5 were to be in column two, column seven would be forced to contain 1 vice versa. We can thereby rule out all 1s and 5s in row nine, in this case revealing column eight in the same row to contain 6. [3]

Figure 5. Naked Pairs/Triples [3].
X-Wing
An advanced technique used in more difficult Sudokus. It can be used when two rows carry the same candidates in the same column, with the requirement that either both rows do not contain the candidate or both the columns do not contain the candidate e.g. row 4 and 9 in Figure 6. These candidates form a ruling that either both (c,4) and (i,9) contain 6 or that both (c,9) and (i,4) contain 6. Therefore both columns c and i cannot contain 6 in any other position. [3]

![Figure 6. X-wing [3].](image)

Guessing
For the desperate, by presuming that a square contains a number one could possibly solve the Sudoku. But you would have to keep in mind that you have taken a guess incase the Sudoku suddenly becomes unsolvable.

2.3 Sudoku grader
There have been several attempts of creating Sudoku graders that recognizes the difficulty of the Sudoku as a whole, one of which is “Sudoku grader”, an implemented grader that checks for what techniques are needed in order to solve the whole Sudoku [4].

The program attempts to solve the puzzle one technique at a time, with each new technique being more advanced than the previous. If the Sudoku has been partially solved and the technique has come to an end it will restart with the easiest technique. This loop will continue until the Sudoku is solved or its progression is frozen. [4]
3. Method

3.1 Design

To be able to test the algorithms the first requirement was to create a framework or a “base of operations” by designing a Sudoku program allowing for basic actions such as read/edit squares, monitor candidates and possibly save/open specific Sudoku.

To enable the creation of as many test cases as possible it would be advantageous to create a random Sudoku generator. This would allow for more time to implement techniques as well as create unthought-of tests.

With a framework and a generator created, focus was on creating a smooth and efficient way to apply techniques on said Sudokus while keeping a realistic way of testing every added technique.

With a base to step forward from, techniques would be added to the program. Each implementation simulates a human using a technique on the Sudoku. The result from researching different techniques [2.1.1] is translated into algorithms suitable for the existing Sudoku platform. Some algorithms were able to be built by combining several techniques as some of them are very similar. This would of course apply to those techniques with next to equal difficulty as to not complicate the grading of a square.

A compilation of which techniques were required to remove all but one candidate from each squares is created in order to simplistically view the results of the program.

The program’s flow would ideally be:

Basic Flow

1. Enter specific Sudoku into Sudoku gui or Use an implemented random generator where you choose a difficulty or open a saved Sudoku.
2. Use the Calculate function to start applying the techniques on the Sudoku.
3. Select a square to see its difficulty.
4. One can now see at the label “Solved Candidates” which techniques were used in order to solve that square.
5. Select “saveResults “ stripmenu option and see the whole matrix.

Alternative flow

Create specific Sudoku and save.

1. Select a square to insert a number into the square.
2. Choose to save your Sudoku to quickly be able to use the same Sudoku.
Open

1. Open a past saved Sudoku

Clear

1. Clear to clear the Sudoku allowing for another manual input.

Exit

1. Exit the program.

3.1.1 Technique Assessment

The most interesting part of the program is the techniques, but which techniques would be fit for implementation and what would their difficulty be compared to each other?

To create a valuation, a broad search was made, granting a comprehension of which were the most common, which were more advance and what similarities the discovered techniques had.

The investigation was made by researching different Sudoku tutorials and their advice regarding techniques [4, 5, 6, and 7]; these sources gave a vast quantity of techniques. So as to be sure of the techniques’ quality the most consistent throughout the sources were selected. They were also selected based on varied difficulties to be able to solve a larger quantity of Sudokus [2.2.1]. Their difficulty rank would largely depend on general consensus with regards to their required time to apply as harder puzzles require deeper logic analysis [2].

With a set of techniques chosen the algorithms could be designed ranging from 1 to X, 1 being the simplest and X being the most advanced. In this case X being 4

<table>
<thead>
<tr>
<th>Technique Nr</th>
<th>T1</th>
<th>T2</th>
<th>T3</th>
<th>T4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Technique Names</td>
<td>Single Candidate</td>
<td>Candidate Lines</td>
<td>Double-Pair &amp; Multiple lines</td>
<td>Naked Pairs/Triples</td>
</tr>
</tbody>
</table>
3.2 Implementation

3.2.1 Structure
As to easy grab and edit different squares it would be efficient to use an objective programming language. Since C# has extensive GUI manipulation tools everything was written In C#.

Framework
The framework is in control of the GUI which is the instigator of data flow.

![Figure 7. Data Flow.](image)

GUI
The Sudoku would need a 9x9 grid of editable squares so buttons became the most suitable choice for each square. The buttons would then have to be organized into the specific boxes. Each button would then act as a prompt to open an edit window for the square it represented [3.1]. This window had to contain information regarding the square’s candidates and which techniques were used (if solved).

Basic functions such as save, open, clear

![Figure 8. File Strip Menu.](image)

![Figure 9. Technique Results.](image)

![Figure 10. Main Form & Square Form.](image)

![Figure 11. Generate Form.](image)
Data
Storing the Sudoku’s information such as: the squares’ numbers, their candidates and position.

Sudoku generator
To make the Sudoku generator’s implementation quicker and more stable a suitable Sudoku generator was adopted into the program as the main focus was to test Sudokus not create them. The generator implemented was from [10] and their generated Sudokus were easy to adapt into the main program albeit it made some changes to the squares’ attributes.

Solver
Solver contains the algorithms for the chosen techniques. The algorithms are applied in order of their complexity. As they are applied they update “Data” for later use in the GUI.

3.2.2 Algorithms
After a Sudoku has been opened, inserted or generated the basic candidates are calculated by going through each square in the Sudoku and checking for numbers that are not contained in the square’s row, column or box.

With this as a starting point we begin to go through the techniques similar to [4], with the exception of saving the name of the algorithm which removed a particular square’s candidate.

As seen in figure 12 after the “calc” button has been pressed, the program steps through each implemented algorithm until either no candidates could be removed or the Sudoku has been solved. (Each box control priority is down, up, right)

![Diagram of algorithms](image)

Figure 12. Calc-Algorithm.

The description for some examples of how the algorithm work is written with pseudo code and can be found on page 14 for further reading see ref [8.1].
Single Candidate

```java
SingleCandidate()
    SudokuMatrix[]
        for i 1 to 9
    for j 1 to 9
        square = SudokuMatrix[i][j] //Sudoku square at coord i,j
        if square only has one candidate
            set the square's number to its single candidate
            remove candidate
        Go through row to check if they have candidates to remove
        same with column and box
```

Figure 13. Pseudo code – Single Candidate.

Candidate Line

```java
CandidateLines()
    for Box 1 to 9
        for Number 1 to 9
            listOfSquares[]
                for square 1 to 9
                    if square contains Number as candidate
                        add to listOfSquares[]
                        for each in listOfSquares[]
                            check if they all share the same row or they all share the same column
                            if they share the same row
                                remove Number from candidates in that row except the box they're in.
                            if they share the same column
                                remove Number from candidates in that column except the box they're in.
```

Figure 14. Pseudo code – Candidate Line.

Double pair & Multi-Line

```java
DoublePairMulti-Line()
    for Number 1 to 9
        for MatrixColumn 1 to 3 //Each column representing a set of 3 boxes
            Data[]
                for Box 1 to 3 //Go through the three boxes
                    count <- 0
                    for BoxColumn 1 to 3
                        if Number present in column
                            count++
                    if count = 2
                        Data[Box] <- store columns Number is in
                        Compare Data[1], Data[2] & Data[3]
                    if Two are the same
                        for the columns that are equal remove number from the third box's candidates.
```

Figure 15. Pseudo code – Double pair & Multi-Line.
3.3 Testing
To test the GUI, manual verification for each user case with a broad range of different input were made. Such as entering values into the squares, saving a Sudoku, generating Sudokus of varied difficulties etc.

Testing that the implemented algorithms are correct i.e. they act as the techniques are described, is quite hard since you would have to cover all different cases where the algorithm could be applied. In regards to this problem the main testing method was to create a set of Sudokus with known solutions, specifically made for testing an algorithm and then running each algorithm on their corresponding test objects.
4. Results

4.1 Test Runs

4.1.1 Random Generated Sudokus
With the method implemented, several random generated sudokus were tested. These are two of the results. The “Before” Sudoku is what the generator produced, the “After” Sudoku is how the solver changed the Sudoku. The result is a 9x9 grid representing each square, with each cell containing how many candidates each technique removed.

1. First Example

![Figure 17. Example 1 Sudoku Before](image1.jpg)

![Figure 16. Example 1 Technique Results.](image2.jpg)

![Figure 18. Example 1 Sudoku After.](image3.jpg)
2. Second Example

Result:

![Figure 19 Example 2 - Sudoku After](image19.png)

![Figure 20. Example 2 - Sudoku Before](image20.png)

![Figure 21. Example 2 Technique Results.](image21.png)
4.1.2 Specific Sudokus
These are the results from only applying a single algorithm on a specific Sudoku. As some techniques only remove candidates the Sudoku is not changed.

*T1 Example:*
A test for Single Candidate and Single Position. The created Sudoku has two squares to fill the first one (1,9) as it only contains one candidate (“1”), the second one (3,9) as it is the only square in box 3 that contains the number 1.

![Figure 22. T1 Example - Sudoku Before.](image)

![Figure 23. T1 Example - Technique Results.](image)

![Figure 24. T1 Example - Sudoku After.](image)
**T2 Example:**
A Candidate Line test. (7, 8) and (8, 8) are the only two squares in box nine containing 4 as a candidate, since box nine has to contain a 4 column eight’s 4 has to be placed in either of those squares. I.e. T2 should remove 4 as a candidate in the other squares in column eight.

![Figure 26. T2 Example - Sudoku used.](image)

**T3 Example:**
A Double pair & Multi-line test. Squares (1, 1), (1, 2), (2, 1) in box one are the only ones carrying 5 as a candidate. Similar to box one is box seven only carrying 5 as a candidate in squares (7, 2), (9, 1) and (9, 2). The test is to see if box five removes 5 as a candidate in columns one and two.

![Figure 28. T3 Example - Sudoku used.](image)
**T4 Example:**
A Test for Naked Pairs/Triples. Squares (1,4) and (1,6) contains candidates 3 and 6. The algorithm should remove either from candidates in square (1,9). Squares (2,7) and (3,7) contains candidates 4 and 6. Squares (4,7) contains 2, 5 and 6 as candidates. According to Naked pairs 6 should be removed.

![Figure 29. Sudoku used.](image1)

![Figure 30. T4 Example - Technique Results](image2)
5. Discussion

5.1 Verification
A precondition while validating the algorithms was that the techniques they represent are valid.

It was relatively easy to verify single algorithms tests within reason. Each algorithm was applied to several predefined Sudokus as described in [2.3] but since these tests are a subset of all possible puzzles, it is not certain that the algorithms are 100% verified.

It was not feasible to investigate the execution of all algorithms in sequence as they are applied. Due to the fact that the algorithms are applied several times in loops and to manually compare inputs and outputs between algorithms would prove to be very difficult to accomplish.

5.2 Method Choice
There are algorithms that solve the Sudoku more efficient than the human techniques, perhaps these could be used to evaluate the difficulty of a square, by calculating how quickly a square could be solved. A difficulty scale could be created and ranking each square based on their solving speed.

The problem with this is that it would be harder to relate to the difficulty evaluation of the squares. Since the speed of the computer technique might not be related to how difficult we would think it is. Therefore it would be more suitable to use a method revolving around using human techniques, for example this reports method.

As for the implementation there were several choices that presumably could have been made different such as the data structure, instead of creating a matrix of objects one could possibly use a more dynamic approach, perhaps by using lists for each square still containing candidates.

The creation of the GUI was a subjective decision as the user interface could have been as simple as a console with its output showing in either the console or stored in a file. The reason for choosing a GUI was for the easy access to the different squares and an advantage to constantly view the Sudoku as a whole i.e. more user-friendly.

Both the ability to save/ open Sudokus and generating Sudokus was a great help while testing the different techniques and relieved a great amount of trivial manual input. The ability to get random Sudokus also helped with testing unforeseen problems.

If the Sudoku is unsolvable by the program with the techniques currently implemented, the solver algorithm is flexible enough to allow for additional techniques to be implemented in a quick and efficient way. Its flexibility being that only this part has to be adjusted for each new technique.

5.3 Comparison
This grading technique has greater adaptability compared to the current Sudoku graders as its results are more accurate and you can pin point each square instead of only having access to the Grid’s
difficulty. The only downside is the handling of more data. I should mention that both this program and “Sudoku Grader” [4] both carry the same ability to smoothly adopt new Sudoku solving techniques, by efficiently adding it to the ladder of the solver algorithm.

All types of graders share a common problem: how can techniques be accurately evaluated in terms of difficulty? The current valuation is varied and scattered and a common ground must be found to solve this problem.

6. Conclusions

The report has reached its goal by demonstrating it is possible to create a method to evaluate a square’s difficulty. Since the method always tries to solve the Sudoku with the easiest techniques first we can be sure that the evaluation is based on the simplest way to find the square’s number.

As some Sudokus are yet to be completely solved by the implemented human techniques, there is a lot of room to apply new ones. There will probably arise new techniques to be added, maybe they are unthought-of techniques or maybe specific techniques for solving a certain set of puzzles.

One should keep in mind that even if a value is placed upon the different techniques, it is still hard to definitively value the results. Since no standardized difficulty scale is in place for the Sudoku techniques. E.g. if a square has the result “T1 T1 T1” what difficulty is it really?

6.1 Other applications

There are a few areas where a square’s difficulty could be used, the most obvious being to assess a whole Sudoku’s difficulty. Other examples would be Sudoku hint applications, i.e. a Sudoku game with the option to hint the player what a square is supposed to be or how it could be solved. Another option could be to give out a certain square’s number based on what amount of help the user desires and revealing a square of corresponding difficulty. Perhaps the program could be modified to generate Sudokus based on user input of squares that should be a certain difficulty.

Since each square’s difficulty is given there is more information to proceed from creating more options compared to grading the Sudoku as a whole.

6.2 Further development

The algorithm relies heavily on looping through the 9x9 matrix, row, columns and boxes a multitude of times which is the most time-consuming. There are presumably quite a few optimizations that could be made but since time is a limiting factor focus has been on testing and verifying.

As said more techniques can be applied but they could also rival each other if they both solve the same square and they share the same difficulty.
7. References


10) emreyurtseven, 2013, Sudoku(CS), http://sudokus.sourceforge.net/, 09/04/2013, 19/02/2014
8. Appendix

8.1 Algorithms

Numbers – Square class contains:
Number – the square’s revealed number
Row - square’s row
Col – square’s column
Candidates – the square’s candidates
SolvingCandidates – which techniques that has removed a candidate

theSudoku – Sudoku class stores 2 matrixes:
SudokuMatrix – Stores the matrix as it is (referred to as sudokuToSolve)
squareboxindex – Each row representing a box

t1
private bool SinglePosition(){
    bool foundone = false;
    for (int number = 1; number < 10; number++){
        //Check rows for unique candidates
        for (int row = 0; row < 9; row++){
            Numbers square = new Numbers();
            bool onlyone = false;
            for (int i = 0; i < 9; i++) {
                if (sudokuToSolve[row, i].Candidates.Contains(number)) {
                    if (onlyone){
                        square = sudokuToSolve[row, i];
                        onlyone = true;
                    }
                    else{
                        onlyone = false;
                        break;
                    }
                }
            }
            if (onlyone) {
                removecandidates(square, number)
                foundone = true;
            }
        }
        //Check col for unique candidates
        for (int col = 0; col < 9; col++){
            Numbers square = new Numbers();
            bool onlyone = false;
            for (int i = 0; i < 9; i++) {
                if (sudokuToSolve[i, col].Candidates.Contains(number)) {
                    if (onlyone){
                        square = sudokuToSolve[i, col];
                        onlyone = true;
                    }
                    else{
                        onlyone = false;
                        break;
                    }
                }
            }
            if (onlyone) {
                removecandidates(square, number)
                foundone = true;
            }
        }
    }
// Check boxes for unique candidates
for (int box = 0; box < 9; box++){
    Numbers square = new Numbers();
    bool onlyone = false;
    for (int i = 0; i < 9; i++){
        if (theSudoku.squareboxindex[box, i].Candidates.Contains(number)){
            if (onlyone){
                square = theSudoku.squareboxindex[box, i];
                onlyone = true;
            } else{
                onlyone = false;
                break;
            }
        }
    }
    if (onlyone){
        removecandidates(square, number)
        foundone = true;
    }
}

return foundone;

private void removecandidates(Numbers square, int number)
{
    while (square.Candidates.Count != 0)
    {
        square.Candidates.RemoveAt(0);
        square.SolvingCandidates.Add("T1");
    }
    square.Number = number;
    // remove candidates as a square was revealed
    RemoveRowCandidates(square.Row, 20, number, "T1");
    RemoveColCandidates(square.Col, 20, number, "T1");
    RemoveBoxCandidates(square.Box, number, "T1");
}

public bool MatrixPosCan() {
    bool foundone = false;
    for (int i = 0; i < 9; i++) {
        for (int j = 0; j < 9; j++){
            if (SingleCan(i, j)){foundone = true;}
        }
    }
    return foundone;
}

public bool SingleCan(int x, int y){
    if (sudokuToSolve[x, y].Candidates.Count == 1) {
        int tmpnumber = Convert.ToInt32(sudokuToSolve[x, y].Candidates[0]);
        sudokuToSolve[x, y].Number = tmpnumber;
        RemoveRowCandidates(x, 20, tmpnumber, "T1");
        RemoveColCandidates(y, 20, tmpnumber, "T1");
        RemoveBoxCandidates(sudokuToSolve[x, y].Box, tmpnumber, "T1");
        return true;
    }
    return false;
}
public bool CandidateLines()
{
    // Titta varje box för row/col som har eget nummer. Ta bort kandidater från andra rutor.
    List<Numbers> oneToNine = new List<Numbers>();
    bool changed = false;
    // Every Box
    for (int boxnmbr = 0; boxnmbr < 9; boxnmbr++)
    {
        // Every number
        for (int currNumber = 1; currNumber < 10; currNumber++)
        {
            // Every Square
            oneToNine.Clear();
            for (int squarenmbr = 0; squarenmbr < 9; squarenmbr++)
            {
                if (theSudoku.squareboxindex[boxnmbr, squarenmbr].Candidates.Contains(currNumber))
                {
                    oneToNine.Add(theSudoku.squareboxindex[boxnmbr, squarenmbr]);
                }
            }
            if (oneToNine.Count == 0)
                continue;
            int row = oneToNine[0].Row;
            int col = oneToNine[0].Col;
            bool rowvalid = true;
            bool colvalid = true;
            foreach (Numbers square in oneToNine)
            {
                if (row != square.Row) {rowvalid = false;}
                if (col != square.Col) {colvalid = false;}
            }
            if (rowvalid ^ colvalid) {
                if (rowvalid)
                {
                    if (RemoveRowCandidates(row, boxnmbr, currNumber, "T2"){changed = true;}
                }
                if (colvalid)
                {
                    if (RemoveColCandidates(col, boxnmbr, currNumber, "T2"){changed = true;}
                }
            }
        }
    }
    return changed;
}
public bool Technique3() {
    bool changedsudoku = false;
    for (int i = 1; i < 10; i++) {
        for (int boxcol = 0; boxcol < 3; boxcol++) {
            bool[] collist = T3checkboxcol(boxcol, i);
            bool[] collist2 = T3checkboxcol(boxcol + 3, i);
            bool[] collist3 = T3checkboxcol(boxcol + 6, i);
            if (collist != null && collist2 != null) {
                bool equal1and2 = collist.SequenceEqual(collist2);
                if (equal1and2 && collist != null) {
                    if (Remove2ColsofCandidatesT3(collist, boxcol + 6, i)) { changedsudoku = true; }
                    continue;
                }
            }
            if (collist != null && collist3 != null) {
                bool equal1and3 = collist.SequenceEqual(collist3);
                if (equal1and3 && collist != null) {
                    if (Remove2ColsofCandidatesT3(collist, boxcol + 3, i)) { changedsudoku = true; }
                    continue;
                }
            }
            if (collist2 != null && collist3 != null) {
                bool equal2and3 = collist2.SequenceEqual(collist3);
                if (equal2and3 && collist2 != null) {
                    if (Remove2ColsofCandidatesT3(collist2, boxcol, i)) { changedsudoku = true; }
                    continue;
                }
            }
        }
    }
    return changedsudoku;
}

public bool[] T3checkboxcol(int boxnumber, int numbertofind) {
    bool[] collist = new bool[3];
    for (int i = 0; i < 9; i++) {
        Numbers square = theSudoku.squareboxindex[boxnumber, i];
        if (square.Candidates.Contains(numbertofind)) { collist[square.Col % 3] = true; }
    }
    int counttrues = 0;
    for (int i = 0; i < 3; i++) {
        if (collist[i] == true) { counttrues++; }
    }
    if (counttrues == 2) { return collist; }
    return null;
}

public bool Remove2ColsofCandidatesT3(bool[] collist, int boxnmbr, int numbertoremove) {
    bool removedatleastone = false;
    for (int i = 0; i < 9; i++) {
        int collestofofcol = theSudoku.squareboxindex[boxnmbr, i].Col % 3;
        if (collist[collestofofcol] == true) {
            if (theSudoku.squareboxindex[boxnmbr, i].Candidates.Remove(numbertoremove)) {
                theSudoku.squareboxindex[boxnmbr, i].SolvingCandidates.Add("T3");
                removedatleastone = true;
            }
        }
    }
    return removedatleastone;
}
As to ease reading I will only show Naked Doubles/Triples for the rows as the code for the columns is quite similar.

Doubles

```csharp
private bool Nakeddoublesrows(int row){
    List<Numbers> validsquares = new List<Numbers>();
    for (int squarenr = 0; squarenr < 9; squarenr++){
        Numbers square = sudokuToSolve[row, squarenr];
        if (square.Candidates.Count == 2){validsquares.Add(square);}
    }
    bool isrow = true;
    return Comparedoubles(validsquares, isrow, false);
}

private bool Comparedoubles(List<Numbers> squarelist, bool isrow, bool changedsudoku){
    bool newchangedsudoku = changedsudoku;
    if (squarelist.Count < 2) {return newchangedsudoku;}
    int can1 = squarelist[0].Candidates[0];
    int can2 = squarelist[0].Candidates[1];
    int[] pos1 = {squarelist[0].Row, squarelist[0].Col};
    squarelist.RemoveAt(0);
    for (int i = 0; i < squarelist.Count; i++){
        if (can1 == squarelist[i].Candidates[0] && can2 == squarelist[i].Candidates[1]){int[] pos2 = {squarelist[i].Row, squarelist[i].Col};
            if (isrow){
                //removes candidates from the row, with the exception of the 2 positions
                if (Removefromrow(can1, can2, pos1, pos2)){newchangedsudoku = true;}
            }else{
                //removes candidates from the column, with the exception of the 2 positions
                if (Removefromcol(can1, can2, pos1, pos2)){newchangedsudoku = true;}
            }
            squarelist.RemoveAt(i);
            return Comparedoubles(squarelist, isrow, newchangedsudoku);
        }
    }
    return Comparedoubles(squarelist, isrow, newchangedsudoku);
}
```
private bool NakedTriplesRows(int row)
{
    List<Numbers> validsquares = new List<Numbers>();
    for (int squarenr = 0; squarenr < 9; squarenr++)
    {
        Numbers square = sudokuToSolve[row, squarenr];
        if (square.Candidates.Count == 3)
        { validsquares.Add(square); }
    }
    bool isrow = true;
    return CompareTriples(validsquares, isrow, false);
}

private bool CompareTriples(List<Numbers> squarelist, bool isrow, bool changedsudoku)
{
    bool newchangedsudoku = changedsudoku;
    if (squarelist.Count < 3) { return newchangedsudoku; }
    int can1 = squarelist[0].Candidates[0];
    int can2 = squarelist[0].Candidates[1];
    int can3 = squarelist[0].Candidates[2];
    int[] pos1 = { squarelist[0].Row, squarelist[0].Col };
    squarelist.RemoveAt(0);
    int counter = 0;
    for (int i = 0; i < squarelist.Count; i++)
    {
        if (can1 == squarelist[i].Candidates[0] &&
            can2 == squarelist[i].Candidates[1] &&
            can3 == squarelist[i].Candidates[2])
        { counter++; }
        int[] pos2 = {squarelist[i].Row, squarelist[i].Col };
        squarelist.RemoveAt(i);
        if (counter == 2)
        {
            int[] pos3 = {squarelist[i].Row, squarelist[i].Col };
            if (isrow)
            { //removes candidates from the row, with the exception of the 3 positions
                if (RemoveFromRowTripple(can1, can2, can3, pos1, pos2, pos3))
                { newchangedsudoku = true; }
            }
            else
            { //removes candidates from the column, with the exception of the 3 positions
                if (RemoveFromColTripple(can1, can2, can3, pos1, pos2, pos3))
                { newchangedsudoku = true; }
            }
            squarelist.RemoveAt(i);
            return CompareTriples(squarelist, isrow, newchangedsudoku);
        }
    }
    return CompareTriples(squarelist, isrow, newchangedsudoku);
}