# Creating and solving sudoku's

Rick van der Zwet, Leiden University
Leiden Institute of Advanced Computer Science

March 13, 2006

# 1 Introduction

## 1.1 History

The name Sudoku is the Japanese abbreviation of a longer phrase, "suuji wa dokushin ni kagiru" meaning "the digits must remain single"; it is a trademark of puzzle publisher Nikoli Co. Ltd in Japan.

The numerals in Sudoku puzzles are used for convenience; arithmetic relationships between numerals are absolutely irrelevant. Any set of distinct symbols will do; letters, shapes, or colours may be used without altering the rules (Penny Press' Scramblets and Knight Features Syndicate's Sudoku Word both use letters). Dell Magazines, the puzzle's originator, has been using numerals for Number Place in its magazines since they first published it in 1979. Numerals are used throughout this article.

The attraction of the puzzle is that the completion rules are simple, yet the line of reasoning required to reach the completion may be complex. Sudoku is recommended by some teachers as an exercise in logical reasoning. The level of difficulty of the puzzles can be selected to suit the audience. The puzzles are often available free from published sources and also may be custom-generated using software.[1]

### 1.2 Gameplay

The puzzle is most frequently a 99 grid, made up of 33 subgrids called "regions" (other terms include "boxes", "blocks", and the like when referring to the standard variation; even "quadrant" is sometimes used, despite this being an inaccurate term for a 99 grid). Some cells already contain numerals, known as "givens" (or sometimes as "clues"). The goal is to fill in the empty cells, one numeral in each, so that each column, row, and region contains the numerals 19 exactly once. Each numeral in the solution therefore occurs only once in each of three "directions" or "scopes", hence the "single numbers" implied by the puzzle's name.[1]

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

Figure 1: Sample sudoku

# 2 Problem

Not every combination of a board, is going to be succesfull and there is no algoritm who can predict whether a generated board is valid without completely solving the puzzle. So there has to be some other way to generate valid puzzles. First, there need to be a perfect solver writtento test whether a puzzle has got only one solution. The second way to generate a valid puzzle is by creating a completely solved valid puzzle and delete a ew spots, by checking if the puzzle still has got 1 unique solution. We are able to generate soduku's too.

# 3 Solution finding theories

There are currently 3 solutions know (by me) about solving the soduku somehow.

## 3.1 Marking up

By deleting the non-available spots at every location, the remaining possibilities will become clear, you will notice at certain points only one option will be left, so this will be the number to be filled in. As can be showed by figure 2.

Figure 2: Candidates for each empty cell have been entered. Some cells have only one candidate once obvious invalids have been excluded. Also, some mark with dots instead of numbers, simply using the position of the dot within the cell to distinguish them.[1]

## 3.2 Scanning

Scanning will be a easy solution too to be used by humans, computers however need slight more code to accomplish the same result. With scanning you can "force" certain numbers into their spot. See figure 3 for a example.

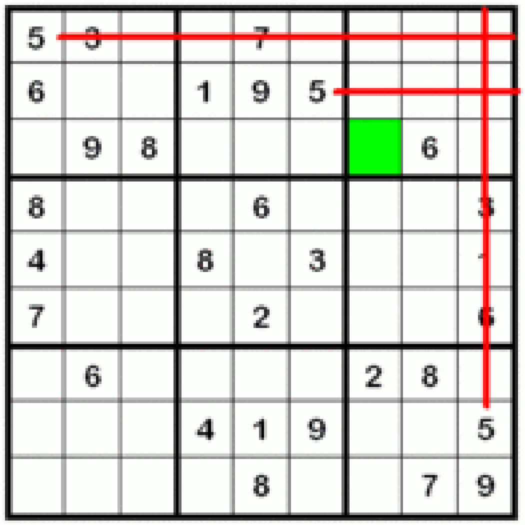


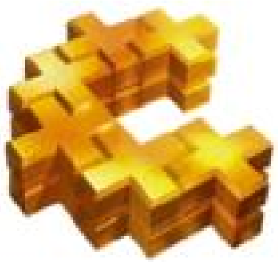
Figure 3: The 33 region in the top-left corner. The solver can eliminate all of the empty cells in the top-right corner which cannot contain a 5. This leaves only one possible cell (highlighted in green).[1]

### 3.3 Analysis

Analysis will be even more complex. It can consist many methods to logically gain a solution. There is one pointed out (and implemented). Let's pretend we have a row in which are 3 spot's still to be filled.

Options to spot A: 1, 2 ,3 Options to spot B: 2, 3 Options to spot C: 2, 3

B and C together will use number 2 and 3, so A has to be 1. It's just an other "simple solution", but implementing it it's a bit hard.



## 4 Implementation

The solver and create program are both written in C++. It uses a 2d array to impement the sudoku board and a 3d array to implement the current available choices at the board.

# 5 Creating soduku's

There are 3 main ways to create sudoku's

1. Randomizing stupid, just push for example 17 random numbers inside the sudoku and hope none of them will interference.
2. Randoming smart, push for example 17 random numbers inside the sudoku and be sure none of them will be a violation of the structure.
3. "Cleaning" first generate a complete sudoku. Then get your wipercleaner and delete some numbers. This solution is a bit difficult and will use a lot of math to generate a full puzzle and you are not sure if the puzzle has got only one solution.

# 6 Experiments

## 6.1 Solver

First we will test the solver. The website of G. Royle[2] has got plenty solveble 17 sudoku's. A few of them are placed at Appendix C. There is a small shell script written (Appendix B) to test the result. By trying to solve 1000 puzzles, the program managed to complete 547, the other puzzles requires some more logic implemented to solve them.

## 6.2 Generator

The generator will generate of puzzle of class 2 (Random and not very stupid), the solver will try to fix this puzzle. No checks for unique solutions will be made. Table below will point out the results by running the test 10000 times.

| Percent sudoku filled initially | Solutions found 1st round | Solutions found 2nd round |
|---|---|---|
| 10% | 0 | 0 |
| 20% | 0 | 0 |
| 30% | 160 | 0 |
| 40% | 0 | 0 |
| 50% | 0 | 0 |

# 7 Conclusion

The solver ain't good enough yet, he should fix every puzzle and not 50%, need to program some aditional logic. The creator isn't very good either, only 30% will generate some solveble puzzles, It properly have something to do with the very bad random generator, the random seed if filled with the unix time in which in 100 occured process will be the same. The creator needs method 3 to be implemented to validate this assumption or a rewrite of the code.

# References


[1] Wikipedia Foundation Inc, http://en.wikipedia.org/wiki/Sudoku

[2] G. Royle, Minimum Sudoku http://www.csse.uwa.edu.au/~gordon/sudokumin.php

# Appendix A: sudoku.cpp

The following C++ has been used to solve the puzzles

```
/* file: sudoku.c
 * author: rick van der Zwet 0433373, 2006
 * University Leiden, LIACS
 */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>
#include <signal.h>

#define MAX_XYZ 9
#define EMPTY_SPOT '_'


#define bold "[1m"
#define normal "[0m"

#define DEBUGPLACE stderr
const char esc = 27;            /* escape character */


int debug(const int level = 0)
{
#if DEBUG
    if (level <= DEBUG)
    {
        return(1);
    }
    else
    {
        return(0);
    }
#else
    return(0);
```

```
#endif
}

class Sudoku {
    public:
        Sudoku();
        void printNice(const int boldRow = -1, const int boldCol = -1, const char emptcolSpace = EMPTY_SPOT);
        void printRaw();
        void printPossible();
        void printPossibleAtNumber(const int row, const int col);
        void reset();
        int randomNumbers(const double chance);
        int fromFile(FILE *handle);
        int insert(const int row, const int col, const int number);
        int solutionFinder();
    private:
        int _sudoku[MAX_XYZ][MAX_XYZ];
        bool _possible[MAX_XYZ][MAX_XYZ][MAX_XYZ];
        int _numbersFilled;
        void _updatePossibleList(const int row, const int col, const int number);
        bool _solution1(int &row, int &col, int &number);
        bool _solution2(int &row, int &col, int &number);
        bool _solution3a(int &row, int &col, int &number);
        bool _solution3b(int &row, int &col, int &number);
};

Sudoku::Sudoku()
{
    reset();
}


void Sudoku::printRaw()
{
    int row,col;
    for (row = 0; row < MAX_XYZ ; row++)
    {
        for (col = 0; col < MAX_XYZ; col++)
        {
            printf("%i ", _sudoku[row][col]);
```

```
        }
        printf("\n");
    }
}
void Sudoku::printNice(const int boldRow, const int boldCol,
                       const char emptcolSpace)
{
   int row,col;
   int tmpRow;
   const int maxRow = (MAX_XYZ + (MAX_XYZ / 3)) * 2 + 1;
   for(tmpRow = 0; tmpRow < maxRow; tmpRow++)
   {
       if ((tmpRow%8) == 0) { printf("+"); }else {printf("-");}
   }
   printf("\n");
   for (row = 0; row < MAX_XYZ ; row++)
   {
       printf("| ");
       for (col = 0; col < MAX_XYZ; col++)
       {
            if ( _sudoku[row][col] == 0)
            {
                printf("%c ",emptcolSpace);
            }
            else
            {
                if (row == boldRow && col == boldCol)
                {
                     printf("%c%s%i%c%s ",esc, bold, _sudoku[row][col],
                                          esc, normal);
                }
                else
                {
                    printf("%i ", _sudoku[row][col]);
                }
            }
            if ( ((col - 2) % 3) == 0 )
            {
                printf("| ");
            }
```

```
        }
       printf("\n");
       if ( ((row - 2) % 3) == 0 )
       {
            for(tmpRow = 0; tmpRow < maxRow; tmpRow++)
            {
                if ((tmpRow%8) == 0)
                {
                    printf("+");
                }
                else
                {
                    printf("-");
                }
            }
            printf("\n");
       }
    }
        printf("\n\n");
}

void Sudoku::printPossibleAtNumber(const int row, const int col)
{
    if (debug(3))
    {
        int tmpPossible;
        fprintf(DEBUGPLACE, "Possible at [%i,%i]: ", row, col);
        for (tmpPossible = 0; tmpPossible < MAX_XYZ; tmpPossible++)
        {
            if (_possible[row][col][tmpPossible] == true)
            {
                fprintf(DEBUGPLACE,"%i ", tmpPossible + 1);
            }
        }
        fprintf(DEBUGPLACE, "\n");
    }
}

void Sudoku::printPossible()
{
```

```
   int row,col;
   if (debug(3))
   {
        for (row = 0; row < MAX_XYZ ; row++)
        {
            for (col = 0; col < MAX_XYZ; col++)
            {
                if(_sudoku[row][col] == 0)
                {
                    printPossibleAtNumber(row,col);
                }
            }
        }
    }
}


void Sudoku::_updatePossibleList(const int row, const int col, const int number)
{
    int tmpRow;
    int tmpCol;
    int blockRow;
    int blockCol;
    int c;


    /* Row and Col and Number own */
    for (c = 0; c < MAX_XYZ; c++)
    {
        _possible[row][col][c] = false;
        _possible[row][c][number - 1] = false;
        _possible[c][col][number - 1] = false;
    }

    /* Block */
    tmpRow = row - (row % (MAX_XYZ / 3));
    tmpCol = col - (col % (MAX_XYZ / 3));
    for (blockRow = tmpRow; blockRow < (tmpRow + 3); blockRow++)
    {
        for (blockCol = tmpCol; blockCol < (tmpCol + 3); blockCol++)
```

```
        {
            _possible[blockRow][blockCol][number - 1] = false;
        }
    }
}

int Sudoku::insert(const int row, const int col, const int number)
{
    if (debug(1)) { fprintf(DEBUGPLACE, "Will insert %i at [%i,%i]: %i\n",
                    number, row, col, _possible[row][col][number - 1]); }

    if (_possible[row][col][number - 1] == true)
    {
        _updatePossibleList(row, col, number);
        _sudoku[row][col] = number;
        _numbersFilled++;
        return(0);
    }
    else
    {
        return(1);
    }
}

int Sudoku::randomNumbers(const double chance)
{
   if (debug(1)) { fprintf(DEBUGPLACE, "Running createsudoku\n"); }
   int row;
   int col;
   int c;
   int randomnumber;
   int numberdone[MAX_XYZ];
   int numbertried;

   srand( (unsigned)time( NULL ) );

   for (row = 0; row < MAX_XYZ; row++)
   {
       for (col = 0; col < MAX_XYZ; col++)
       {
```

```
            if (rand() < (RAND_MAX * (chance / 100)))
            {
                numbertried = 0;
                for (c = 0; c < MAX_XYZ; c++ ) { numberdone[c] = 0; }
                do
                {
                    if (numbertried == 9)
                    {
                        return(1);
                    }
                    else
                    {
                        numbertried++;
                    }
                  do
                  {
                     randomnumber = 1 + (int) ( (double)MAX_XYZ * (random() / (RAND_MAX + 1.0)));
                  } while (numberdone[randomnumber - 1] == 1);
                  numberdone[randomnumber - 1] = 1;
                  if (debug(2)) { fprintf(DEBUGPLACE, "Random check number %i at [%i,%i]\n",
                          randomnumber, row,col); }
                } while ( insert(row, col, randomnumber) == 1);
             }
             else
             {
                 _sudoku[row][col] = 0;
             }
        }
    }
    return(0);
}

void Sudoku::reset()
{
    //temp counters
    int row;
    int col;
    int z;

    _numbersFilled = 0;
```

```
    for (row = 0; row < MAX_XYZ; row++)
    {
        for (col = 0; col < MAX_XYZ; col++)
        {
            _sudoku[row][col] = 0;
            for (z = 0; z < MAX_XYZ; z++)
            {
                _possible[row][col][z] = true;
            }
        }
    }
}

int Sudoku::fromFile(FILE *handle)
{
        int row;
        int col;
        int tmpInt;
        for (row = 0; row < MAX_XYZ; row++)
        {
            for (col = 0; col < MAX_XYZ; col++)
            {
                fscanf(handle, "%1i", &tmpInt);
                if (tmpInt != 0)
                {
                    insert(row, col, tmpInt);
                }
            }
        }
        fclose(handle);
        return(0);
}

/* Use Basic technique of finding places with just one number left to place */
bool Sudoku::_solution1(int &row, int &col, int &number)
{
    #define NOTHING_FOUND 0
    #define ONE_FOUND 1
    #define MORE_THEN_ONE_FOUND 2
    int tmpRow;
```

```
int tmpCol;
int tmpPossible;
int tmpNumber;
int searchState;

for (tmpRow = 0; tmpRow < MAX_XYZ; tmpRow++)
{
    for (tmpCol = 0; tmpCol < MAX_XYZ; tmpCol++)
    {
        if (_sudoku[tmpRow][tmpCol] == 0)
        {
            tmpNumber = 0;
            searchState = NOTHING_FOUND;
            for (tmpPossible = 0; tmpPossible < MAX_XYZ; tmpPossible++)
            {
                if (_possible[tmpRow][tmpCol][tmpPossible] == true &&
                    searchState == ONE_FOUND)
                {
                    searchState = MORE_THEN_ONE_FOUND;
                    break;
                }
                else if (_possible[tmpRow][tmpCol][tmpPossible] == true &&
                         searchState == NOTHING_FOUND)
                {
                    tmpNumber = tmpPossible + 1;
                    searchState = ONE_FOUND;
                }
            }
            if (searchState == ONE_FOUND)
            {
                row = tmpRow;
                col = tmpCol;
                number = tmpNumber;
                return(true);
            }
        }
    }
}

return(false);
```

```
}

/* Check if number can onlcol be placed somewhere, if not able to place anywhere else */
/* FIXME: Should be more modular, to reduce line length and readabilitcol */
/* FIXME: find nice solution for goto's */
bool Sudoku::_solution2(int &row, int &col, int &number)
{
    int tmpRow;
    int tmpCol;
    int tmpPossible;
    int tmpNumber;

    int blockRow;
    int blockTmpRow;
    int blockCol;
    int blockTmpCol;
    bool result;

    for (tmpRow = 0; tmpRow < MAX_XYZ; tmpRow++)
    {
        for (tmpCol = 0; tmpCol < MAX_XYZ; tmpCol++)
        {
            if (_sudoku[tmpRow][tmpCol] == 0)
            {
                for (tmpPossible = 0; tmpPossible < MAX_XYZ; tmpPossible++)
                {
                    if (_possible[tmpRow][tmpCol][tmpPossible] == true)
                    {
                       if (debug(3))
                       {
                           fprintf(DEBUGPLACE,
                                   "DEBUG SOL 2: Checking [%i,%i], number: %i...\n",
                                   tmpRow,tmpCol,(tmpPossible +1));
                       }
                        /* checkRow */
                        result = true;
                        for (tmpNumber = 0; tmpNumber < MAX_XYZ; tmpNumber++)
                        {
                            if (_possible[tmpNumber][tmpCol][tmpPossible] == true &&
                                tmpRow != tmpNumber)
```

```
    {
        if (debug(3))
        {
            fprintf(DEBUGPLACE,
                    "DEBUG SOL 2: ..failed at Rowcheck bcol [%i,%i]\n",
                    tmpNumber,tmpCol);
        }
        result = false;
        break;
    }
}
if (result == true)
{
    if (debug(3)) { fprintf(DEBUGPLACE,"DEBUG SOL 2: ..succesfull at Rowcheck\n"); }
    goto solutionFound;
}

/* checkCol */
result = true;
for (tmpNumber = 0; tmpNumber < MAX_XYZ; tmpNumber++)
{
    if (_possible[tmpRow][tmpNumber][tmpPossible] == true &&
        tmpCol != tmpNumber)
    {
        if (debug(3))
        {
            fprintf(
                    DEBUGPLACE,
                    "DEBUG SOL 2: ..failed at Cowcheck bcol [%i,%i]\n",
                    tmpRow,tmpNumber);
        }
        result = false;
        break;
    }
}
if (result == true)
{
    if (debug(3)) { fprintf(DEBUGPLACE,"DEBUG SOL 2: ..succesfull at Cowcheck\n"); }
    goto solutionFound;
}
```

```
                        /* checkBlock */
                        result = true;
                        blockTmpRow = tmpRow - (tmpRow % (MAX_XYZ / 3));
                        blockTmpCol = tmpCol - (tmpCol % (MAX_XYZ / 3));
                        for (blockRow = blockTmpRow; blockRow < (blockTmpRow + 3); blockRow++)
                        {
                            for (blockCol = blockTmpCol; blockCol < (blockTmpCol + 3); blockCol++)
                            {
                               if (debug(4))
                               {
                                   fprintf(DEBUGPLACE,"DEBUG SOL 2: ..checking [%i,%i]\n",blockRow,blockCol);
                               }
                               if(_possible[blockRow][blockCol][tmpPossible] == true &&
                                  (not (tmpRow == blockRow && tmpCol == blockCol)) )
                               {
                                    if (debug(4)) { fprintf(DEBUGPLACE,"DEBUG SOL 2: ...will cause failure\n"); }
                                    result = false;
                                    goto blockCheckFailed;
                               }
                               else
                               {
                                if (debug(4)) { fprintf(DEBUGPLACE,"DEBUG SOL 2: ...will be no harm\n"); }
                               }
                            }
                        }
blockCheckFailed:
                        if (result == true)
                        {
                            if (debug(3)) { fprintf(DEBUGPLACE,"DEBUG SOL 2: ..succesfull at Blockcheck\n"); }
                            goto solutionFound;
                        }
                    }
                }
            }
        }
    }

    return(false);
```

```
solutionFound:
    row = tmpRow;
    col = tmpCol;
    number = tmpPossible + 1;
    return(true);
}

/* available striper, row based */
/* FIXME: Should be more modular, to reduce line length and readabilitcol */
bool Sudoku::_solution3a(int &row, int &col, int &number)
{
    int tmpRow;
    int tmpCol;
    int tmpPossible;
    int tmpPossible2;
    int tmpColNumber;
    int tmpNumber[MAX_XYZ];
    int tmpAllNumbers;
    int tmpAllNumbersFound;

    bool result;

    for (tmpRow = 0; tmpRow < MAX_XYZ; tmpRow++)
    {
        for (tmpCol = 0; tmpCol < MAX_XYZ; tmpCol++)
        {
            if (_sudoku[tmpRow][tmpCol] == 0)
            {
            if (debug(3)) { fprintf(DEBUGPLACE,"SOL 3a DEBUG Checking [%i,%i]...\n",tmpRow,tmpCol); }
                /* Set all numbers */
                tmpAllNumbers = 0;
                for (tmpPossible = 0; tmpPossible < MAX_XYZ; tmpPossible++)
                {
                    if (_possible[tmpRow][tmpCol][tmpPossible] == true)
                    {
                        tmpNumber[tmpPossible] = 1;
                        tmpAllNumbers++;
                    }
                    else
                    {
```

```
        tmpNumber[tmpPossible] = 0;
    }
}
if (debug(3)) { fprintf(DEBUGPLACE,"SOL 3a DEBUG %i\n", tmpAllNumbers); }
if (tmpAllNumbers != 0)
{
    for (tmpColNumber = 0; tmpColNumber < MAX_XYZ; tmpColNumber++)
    {
        if (_sudoku[tmpRow][tmpColNumber] == 0 && tmpColNumber != tmpCol)
        {
            result = true;
            for (tmpPossible2 = 0; tmpPossible2 < MAX_XYZ; tmpPossible2++)
            {
                if ( _possible[tmpRow][tmpColNumber][tmpPossible2] == true &&
                      _possible[tmpRow][tmpCol][tmpPossible2] == false)
                {
                    if (debug(3))
                    {
                        fprintf(DEBUGPLACE,"SOL 3a DEBUG [%i,%i] failed at %i\n",
                                tmpRow, tmpColNumber,tmpPossible2+1);
                    }
                    result = false;
                    break;
                }
            }
            if (result == true)
            {
                if (debug(3))
                {
                    fprintf(DEBUGPLACE,"SOL 3a DEBUG [%i,%i] will increase\n",
                            tmpRow, tmpColNumber);
                }
                for (tmpPossible2 = 0; tmpPossible2 < MAX_XYZ; tmpPossible2++)
                {
                    if (_possible[tmpRow][tmpColNumber][tmpPossible2] == true)
                    {
                        if (debug(3)) { fprintf(DEBUGPLACE,
                                "SOL 3a DEBUG [%i,%i] will increase possible %i to %i\n",
                                tmpRow, tmpColNumber, tmpPossible2 + 1, tmpNumber[tmpPossible2] + 1);
                        }
```

```
                    if (tmpNumber[tmpPossible2] == tmpAllNumbers)
                    {
                        goto colCheckFailed;
                    }
                    else
                    {
                        tmpNumber[tmpPossible2]++;
                    }
                }
            }
        }
    }
}

/* calculate if result found */
tmpAllNumbersFound = 1;
for (tmpPossible = 0; tmpPossible < MAX_XYZ; tmpPossible++)
{
    if (tmpNumber[tmpPossible] == tmpAllNumbers)
    {
        tmpAllNumbersFound++;
        if (debug(3))
        {
            fprintf(DEBUGPLACE,
                    "SOL 3b DEBUG %i reduced tmpAllNumbers to %i\n",
                    tmpPossible + 1, tmpAllNumbers);
        }
    }
}

if (tmpAllNumbersFound == tmpAllNumbers)
{
    for (tmpPossible = 0; tmpPossible < MAX_XYZ; tmpPossible++)
    {
        if (tmpNumber[tmpPossible] == 1)
        {
            row = tmpRow;
            col = tmpCol;
            number = tmpPossible + 1;
```

```
                            return(true);
                        }
                    }
                }
            }
        } //end if
colCheckFailed:
        if (debug(3)) { fprintf(DEBUGPLACE,"SOL 3c DEBUG no failed solution found\n"); }
        }
    }
    return(false);
}

/* available striper, col based, note row number will change ;-) */
/* FIXME: Should be more modular, to reduce line length and readabilitcol */
bool Sudoku::_solution3b(int &row, int &col, int &number)
{
    int tmpRow;
    int tmpCol;
    int tmpPossible;
    int tmpPossible2;
    int tmpRowNumber;
    int tmpNumber[MAX_XYZ];
    int tmpAllNumbers;
    int tmpAllNumbersFound;

    bool result;

    for (tmpRow = 0; tmpRow < MAX_XYZ; tmpRow++)
    {
        for (tmpCol = 0; tmpCol < MAX_XYZ; tmpCol++)
        {
            if (_sudoku[tmpRow][tmpCol] == 0)
            {
            if (debug(3)) { fprintf(DEBUGPLACE,"SOL 3b DEBUG Checking [%i,%i]...\n",tmpRow,tmpCol); }
                /* Set all numbers */
                tmpAllNumbers = 0;
                for (tmpPossible = 0; tmpPossible < MAX_XYZ; tmpPossible++)
                {
                    if (_possible[tmpRow][tmpCol][tmpPossible] == true)
```

```
    {
        tmpNumber[tmpPossible] = 1;
        tmpAllNumbers++;
    }
    else
    {
        tmpNumber[tmpPossible] = 0;
    }
}
if (debug(3)) { fprintf(DEBUGPLACE,"SOL 3b DEBUG %i\n", tmpAllNumbers); }
if (tmpAllNumbers != 0)
{
    for (tmpRowNumber = 0; tmpRowNumber < MAX_XYZ; tmpRowNumber++)
    {
        if (_sudoku[tmpRowNumber][tmpCol] == 0 && tmpRowNumber != tmpRow)
        {
            result = true;
            for (tmpPossible2 = 0; tmpPossible2 < MAX_XYZ; tmpPossible2++)
            {
                if ( _possible[tmpRowNumber][tmpCol][tmpPossible2] == true &&
                      _possible[tmpRow][tmpCol][tmpPossible2] == false)
                {
                    if (debug(3))
                    {
                        fprintf(DEBUGPLACE,
                                "SOL 3b DEBUG [%i,%i] failed at %i\n",
                                tmpRowNumber, tmpCol,tmpPossible2+1);
                    }
                    result = false;
                    break;
                }
            }
            if (result == true)
            {
                if (debug(3)) { fprintf(DEBUGPLACE,
                        "SOL 3b DEBUG [%i,%i] will increase\n",
                        tmpRowNumber, tmpCol);
                }
                for (tmpPossible2 = 0; tmpPossible2 < MAX_XYZ; tmpPossible2++)
                {
```

```
                if (_possible[tmpRowNumber][tmpCol][tmpPossible2] == true)
                {
                    if (debug(3))
                    {
                        fprintf(DEBUGPLACE,
                                "SOL 3b DEBUG [%i,%i] will increase possible %i to %i\n",
                                tmpRowNumber, tmpCol, tmpPossible2 + 1,
                                tmpNumber[tmpPossible2] + 1);
                    }

                    if (tmpNumber[tmpPossible2] == tmpAllNumbers)
                    {
                        goto rowCheckFailed;
                    }
                    else
                    {
                        tmpNumber[tmpPossible2]++;
                    }
                }
            }
        }
    }
}

/* calculate if result found */
tmpAllNumbersFound = 1;
for (tmpPossible = 0; tmpPossible < MAX_XYZ; tmpPossible++)
{
    if (tmpNumber[tmpPossible] == tmpAllNumbers)
    {
        tmpAllNumbersFound++;
        if (debug(3))
        {
            fprintf(DEBUGPLACE,
                "SOL 3b DEBUG %i reduced tmpAllNumbers to %i\n",
                tmpPossible + 1, tmpAllNumbers);
        }
    }
}
```

```
                    if (tmpAllNumbersFound == tmpAllNumbers)
                    {
                        for (tmpPossible = 0; tmpPossible < MAX_XYZ; tmpPossible++)
                        {
                            if (tmpNumber[tmpPossible] == 1)
                            {
                                row = tmpRow;
                                col = tmpCol;
                                number = tmpPossible + 1;
                                return(true);
                            }
                        }
                    }
                }
            } //end if
rowCheckFailed:
            if (debug(3)) { fprintf(DEBUGPLACE,"SOL 3b DEBUG no failed solution found\n"); }
        }
    }
    return(false);
}
int Sudoku::solutionFinder()
{
    #define NO_SOLUTION 0
    #define SOLUTION_1 1
    #define SOLUTION_2 2
    #define SOLUTION_3a 3
    #define SOLUTION_3b 4
    #define SOLUTION_3c 5
    int row;
    int col;
    int number;

    bool gameAnalcolzed;
    int returnCode;
    int solutionNumber;

    gameAnalcolzed = false;

    while( gameAnalcolzed == false )
```

```
{
    solutionNumber = NO_SOLUTION;
    if (_numbersFilled == MAX_XYZ * MAX_XYZ)
    {
        gameAnalcolzed = true;
        returnCode = 0;
    }
    else
    {
        if (_solution1(row, col, number) == true)
        {
            solutionNumber = SOLUTION_1;
        }
        else if(_solution2(row, col, number) == true)
        {
            solutionNumber = SOLUTION_2;
        }
        else if(_solution3a(row, col, number) == true)
        {
            solutionNumber = SOLUTION_3a;
        }
        else if(_solution3b(row, col, number) == true)
        {
            solutionNumber = SOLUTION_3b;
        }
        else
        {
            gameAnalcolzed = true;
            returnCode = 1;
        }
    }

    if (solutionNumber != NO_SOLUTION)
    {
        insert(row, col, number);
        printf("Filled in %i at [%i,%i] found with finder %i\n", number, row, col, solutionNumber);
        printf("Total Numbers filled %i\n", _numbersFilled);
        printNice(row, col);
        getchar();
    }
```

```
    }
    return(returnCode);
}

int main(int argc, char *argv[])
{
    Sudoku puzzel;
    int timesFailed;
    int returnCode;

    returnCode = 0;

    if (argc == 3 && strcmp(argv[1], "-f") == 0)
    {
        if (strcmp(argv[2], "-") == 0)
        {
            if (debug(1)) { fprintf(DEBUGPLACE, "Reading from 'stdin'\n"); }
            puzzel.fromFile(stdin);
        }
        else
        {
            if (debug(1)) { fprintf(DEBUGPLACE, "Reading from %s\n",argv[2]); }
            puzzel.fromFile(fopen(argv[2], "r"));
        }
        printf("I have read this puzzle\n");
        puzzel.printNice();
        returnCode = puzzel.solutionFinder();
        if (returnCode == 0)
        {
            printf("I Guess this will be the final solution\n");
            puzzel.printNice();
        }
        else
        {
            printf("Finding Solution failed, error code: %i\n", returnCode);
            printf("Failed at puzzle\n");
            puzzel.printNice();
            puzzel.printPossible();
        }
```

```
    }
    else if (argc == 3 && strcmp(argv[1], "-c") == 0)
    {
        fprintf(DEBUGPLACE,"Please wait will generating puzzle");
        timesFailed = 0;
        while (puzzel.randomNumbers(strtod(argv[2], NULL)) == 1 )
        {
            timesFailed++;
            if (debug(1))
            {
                fprintf(DEBUGPLACE, "Failed to created the puzzel at the %i round\n", timesFailed);
            }
            else
            {
                fprintf(DEBUGPLACE,".");
            }
            puzzel.reset();
        }
        fprintf(DEBUGPLACE, "\nResult, (NOTE(!): puzzle might not be solveble)\n");
        puzzel.printRaw();
    }
    else
    {
        printf("Usage %s -f <filename>\n",argv[0]);
        printf("NOTE 1: No error checking done, valid input puzzel will be ");
        printf("generated the -c flags\n");
        printf("NOTE 2: if <filename> = -, stdin will be used as input");
        printf("Usage %s -c <percent filled> To create a new puzzle\n",argv[0]);
        printf("NOTE 1: Above 60%% it will be a bit hard to create ");
        printf("a puzzle\n");
        printf("NOTE 2: Puzzle generated will not always be solveble\n");
        returnCode = 128;
    }
    return(returnCode);
}
```

## Appendix B: sudoku.cpp

Sample of 17-clue puzzles found at Minimum Sudoku[2]. Input Row based.

```
000000010400000000020000000000050407008000300001090000300400200050100000000806000
000000010400000000020000000000050604008000300001090000300400200050100000000807000
000000012000035000000600070700000300000400800100000000000120000080000040050000600
000000012003600000000007000410020000000500300700000600280000040000300500000000000
000000012008030000000000040120500000000004700060000000507000300000620000000100000
000000012040050000000009000070600400000100000000050000087500601000300200000000000
000000012050400000000000030700600400001000000000080000920000800000510700000003000
000000012300000060000040000900000500000001070020000000000350400001400800060000000
000000012400090000000000050070200000600000400000108000018000000000030700502000000
000000012500008000000700000600120000700000450000030000030000800000500700020000000
000000012700060000000000050080200000600000400000109000019000000000030800502000000
000000012800040000000000060090200000700000400000501000015000000000030900602000000
000000013000500070000802000000400900107000000000000200890000050040000600000010000
000000013000700060000508000000400800106000000000000200740000050020000400000010000
000000013000700060000509000000400900106000000000000200740000050080000400000010000
000000013000800070000502000000400900107000000000000200890000050040000600000010000
000000013020500000000000010300007000080200000400000000034050067000020000000010000
000000013040000080200060000609000400000800000000300000030100500000040706000000000
000000013040000080200060000906000400000800000000300000030100500000040706000000000
000000013040000090200070000607000400000300000000900000030100500000060807000000000
```

...

# Appendix C: sudoku.cpp

The following Shell code has been used to generate the test results

```
#!/bin/sh
# file: testcase.sh
# author: Rick van der Zwet, 0433373
# University Leiden, LIACS
timesPlayed=0
timesWon=0
timesLost=0
if [ "x$1" == "x" ]; then
    totalNumber=100
else
    totalNumber=$1
fi

for puzzle in `cat puzzles17`
do
    let "timesPlayed+=1"
    echo "${puzzle}" | ./sudoku -f - >/dev/null
    if [ "x$?" == "x0" ]; then
        let "timesWon+=1"
    else
        let "timesLost+=1"
    fi
    echo "Played: ${timesPlayed}"
    echo "won   : ${timesWon}"
    echo "lost  : ${timesLost}"
    if [ "x${timesPlayed}" == "x${totalNumber}" ]; then
        break
    fi
done
```