Project Review 2 and 3

CSE2003 – Data Structures and Algorithms (Project Component)

Lab Slot: L53+L54

Project Title: Solving Sudoku Problems using various Algorithms

Submitted to: Dr. Gayathri P

Submitted by:

Registration number	Name
19BCB0125	Harshvardhan Mishra
19BCE0876	Chinmay Sharma
19BCE0908	Aaditya Shankar Natarajan
19BCE0957	Vatsal Poddar
19BCE0971	Aashish Sharma

Abstract

In this project, we aim to solve sudoku puzzles with the help of various different methods. A Sudoku is a logic-based, combinatorial number-placement puzzle. It generally involves a 9x9 grid, but can be formed out of a grid of size n^2 x n^2.

Aim

To solve sudoku puzzles with the help of different algorithms

Objective

To understand the basic concepts of data structures and algorithms through a practical and project-based approach

To describe different algorithmic approaches to the same problem, and assess the trade-offs involved

To provide an efficient algorithmic solution to a problem (Sudoku puzzle)

Scope/Applicability

The scope of the project lies in mathematically devising and solving a Sudoku puzzle by applying mathematical conclusions on a Sudoku problem. The code has been structured in such a way that it can solve simple 9x9 Sudoku, as well as complex and bigger sizes such as 25x25 or 64x64. The applicability of the project lies in the algorithms discussed in the project where we further monitor the speed and complexity of these algorithms to find the most efficient and faster algorithm to solve the Sudoku problem.

Introduction

Sudoku is a logic-based, combinatorial number-placement puzzle. In classic sudoku, the objective is to fill a 9×9 grid with digits so that each column, each row, and each of the nine 3×3 sub grids that compose the grid (also called “boxes," “blocks," or "regions") contain all of the digits from 1 to 9. The puzzle setter provides a partially completed grid, such that the particular puzzle has a unique solution. However, even though the rules are seemingly simple, there are 6,670,903,752,021,072,936,960 valid sudoku puzzles. A standard sudoku is a nine by nine grid, but a puzzle can be made with any n2 by n2 grid.

The algorithms we are using are:

1. Backtracking (Non-recursive)
2. Backtracking (Recursive)
3. Constraint Propagation
4. Brute Force Search
5. Knuth’s Algorithm (Dancing Links)

Literature Review

IJCSNS International Journal of Computer Science and Network Security, VOL.10 No.8, August 2010 255. The Game of Sudoku-Advanced Backtrack Approach - Abhishek Majumder, Abhay Kumar, Nilu Das and NilotpalChakrabortyDepartment of Information Technology, Assam University, Silchar, Assam, India.

Sudoku is a puzzle game played on a grid that consists of 9 x 9 cells each belonging to three groups: one of nine rows, one of nine columns and one of nine sub grids. The game of Sudoku is basically based on Latin squares. The Sudoku was invented in the year 1979 and was first

published in the Dell Magazines as "Number Place" in the year 1984.There are several Sudoku applications that have already been developed by many programmers around the globe.

In this paper, we give an overview of the work that we have performed on the development of the game of Sudoku that generates a 9 x 9 puzzle grid with various difficulty levels. The application also enables users to input their own puzzle and to be solved by the computer. The solving algorithm of the developed Sudoku application has also been compared to some existing Sudoku applications for analysis.

Introduction to "Sudoku" is a challenging numeric puzzle that trains our logical mind!! There's no math involved in it-we just need to solve the numeric puzzle with reasoning and logic. Solving a Sudoku puzzle requires no math, not even arithmetic. Ironically, despite being a game of numbers, Sudoku demands not an iota of mathematics of its solvers. Sudoku presents mathematicians and computer scientists with a host of challenging issues. Sudoku puzzles, and their variants, have become extremely popular in the last decade, and can now be found daily in most major newspapers.

In addition to the Manuscript received August 5, 2010 Manuscript revised August 20, 2010 countless books of Sudoku puzzles, there are many guides to Sudoku strategy and logic. Unlike the three-dimensional Rubik's cube, a Sudoku puzzle is a flat, square grid. A single square in the puzzle, a group of 3X3 cells, a group of 3X3 boxes, a column of 9 cells, a row of 9 cells, a value that was already assigned to cell at the start of a game, which cannot be changed, a value that can be inserted into a cell without violating the rule of the game at time when the value is inserted.

To be satisfactory for human solvers, the solution implied by the hints should be unique, so it is desirable to generate proper puzzles. Basically,

there are two different methods to create a proper Sudoku puzzle: Incremental generation, which assigns numerals to one cell after another, until sufficient hints are given for the puzzle to have a unique solution. Decremental generation removes numerals from the cells of a full Sudoku grid for as long as desired or possible in order for the solution to stay unique.

Generate random values for val, row, Although the initial goal for the work was just to implement a simple pen and paper version of Sudoku on the computer without the computer actually solving the puzzle for the user. It would be nice to have the solver in case the user is stuck and wants to find an answer; although now-a-days there are many online solving tips and solvers are available. It is also interesting to find out what kind of algorithms work well with different types of puzzles. Here we have followed a backtracking algorithm and constraint propagation to solve the puzzle. There are two main reasons why this is not desirable: Backtracking in generally takes too much time and it is not fitting to judge the difficulty of a Sudoku puzzle. Still, we have followed this process based on the facts that it is easier than the other methods and there are only a few Sudoku applications that were based on backtracking algorithms.

Implementation

A. Solving sudoku using Backtracking Algorithm (Without Recursion)

Algorithm for Sudoku Solver using Backtracking

1. Assign numbers one by one to an empty cell.
2. Before assigning we check if it is safe or not.
3. If in a cell there is no other possibility then we go to the previous cell, empty it and then try going for some other possibility.

4. We repeat the above three steps recursively till we get the solution to the puzzle.

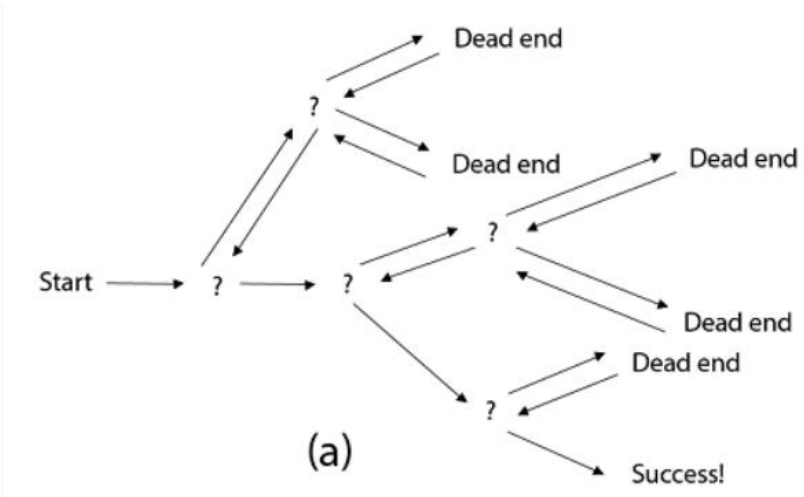
Code for Sudoku Solver using Backtracking

```
#include <stdio.h>
#include<stdbool.h>
#define UNASSIGNED 0 // UNASSIGNED is used for empty cells in sudoku grid
#define N 9
bool FindUnassignedLocation(int grid[N][N], int &row, int &col);
bool isSafe(int grid[N][N], int row, int col, int num);
bool SolveSudoku(int grid[N][N]){
    int row, col;
    if (!FindUnassignedLocation(grid, row, col))
        return true;
    for (int num = 1; num <= 9; num++){
        if (isSafe(grid, row, col, num)){
            grid[row][col] = num;
            if (SolveSudoku(grid))
                return true;
            grid[row][col] = UNASSIGNED;
        }
    }
    return false;
}
bool FindUnassignedLocation(int grid[N][N], int &row, int &col){
    for (row = 0; row < N; row++)
        for (col = 0; col < N; col++)
            if (grid[row][col] == UNASSIGNED)
                return true;
    return false;
}
bool UsedInRow(int grid[N][N], int row, int num){
    for (int col = 0; col < N; col++)
        if (grid[row][col] == num)
            return true;
    return false;
}
bool UsedInCol(int grid[N][N], int col, int num){
    for (int row = 0; row < N; row++)
        if (grid[row][col] == num)
```

```
            return true;
    return false;
}
bool UsedInBox(int grid[N][N], int boxStartRow, int boxStartCol, int num){
    for (int row = 0; row < 3; row++)
        for (int col = 0; col < 3; col++)
            if (grid[row+boxStartRow][col+boxStartCol] == num)
                return true;
    return false;
}
bool isSafe(int grid[N][N], int row, int col, int num){
    return !UsedInRow(grid, row, num) &&
    !UsedInCol(grid, col, num) &&
    !UsedInBox(grid, row - row%3 , col - col%3, num)&&
    grid[row][col]==UNASSIGNED;
}
void printGrid(int grid[N][N]) // A utility function to print grid{
    for (int row = 0; row < N; row++){
        printf("----------------------------\n");
        printf("|");
        for (int col = 0; col < N; col++)
            printf("%2d|", grid[row][col]);
        printf("\n");
    }
    printf("----------------------------\n");
}
int main(){
    int grid[N][N] = {{3, 0, 6, 5, 0, 8, 4, 0, 0},
    {5, 2, 0, 0, 0, 0, 0, 0, 0},
    {0, 8, 7, 0, 0, 0, 0, 3, 1},
    {0, 0, 3, 0, 1, 0, 0, 8, 0},
    {9, 0, 0, 8, 6, 3, 0, 0, 5},
    {0, 5, 0, 0, 9, 0, 6, 0, 0},
    {1, 3, 0, 0, 0, 0, 2, 5, 0},
    {0, 0, 0, 0, 0, 0, 0, 7, 4},
    {0, 0, 5, 2, 0, 6, 3, 0, 0}};
    printf("\n\nSolution of sudoku using Backtracking\n\n");
    if (SolveSudoku(grid) == true)
        printGrid(grid);
    else
        printf("No solution exists");
```

```
    return 0;
}
```

```
 3 1 6 5 7 8 4 9 2
 5 2 9 1 3 4 7 6 8
 4 8 7 6 2 9 5 3 1
 2 6 3 4 1 5 9 8 7
 9 7 4 8 6 3 1 2 5
 8 5 1 7 9 2 6 4 3
 1 3 8 9 4 7 2 5 6
 6 9 2 3 5 1 8 7 4
 7 4 5 2 8 6 3 1 9
[Finished in 2.0s]
```



B. Solving sudoku using Backtracking Algorithm (Using recursion)

Backtracking is an algorithmic-technique for solving problems recursively by trying to build a solution incrementally, one piece at a time, removing those solutions that fail to satisfy the constraints of the problem at any point of time (by time, here, is referred to the time elapsed till reaching any level of the search tree).

Algorithm:

1. Create a function that checks if the given matrix is valid sudoku or not. Keep Hashmap for the row, column and boxes. If any number has a frequency greater than 1 in the hashMap return false else return true;

2. Create a recursive function that takes a grid and the current row and column index.
3. Check some base cases. If the index is at the end of the matrix, i.e. i=N-1 and j=N then check if the grid is safe or not, if safe print the grid and return true else return false. The other base case is when the value of column is N, i.e j = N, then move to the next row, i.e. i + + and j = 0.
4. If the current index is not assigned then fill the element from 1 to 9 and recur for all 9 cases with the index of the next element, i.e. i, j+1. if the recursive call returns true then break the loop and return true.
5. If the current index is assigned then call the recursive function with index of next element, i.e. i, j+1

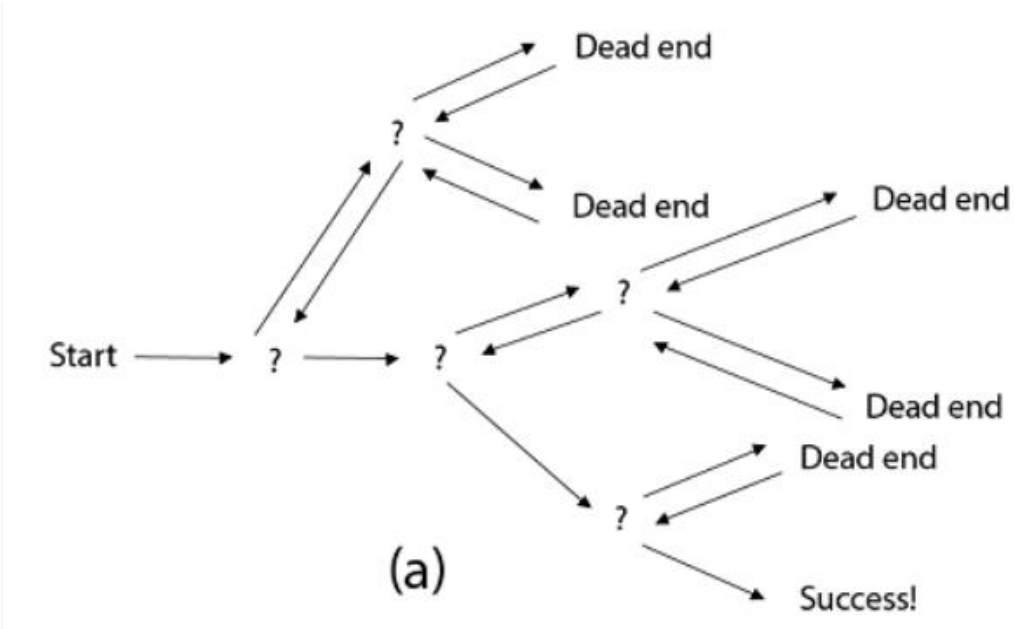
Code:

```
#include <bits/stdc++.h>
using namespace std;
#define UNASSIGNED 0
#define N 9
bool isSafe(int grid[N][N])
{
    unordered_map<int, int>
        row_[9], column_[9], box[3][3];
    for (int row = 0; row < N; row++) {
        for (int col = 0; col < N; col++) {
            row_[row][grid[row][col]] += 1;
            column_[col][grid[row][col]] += 1;
            box[row / 3][col / 3][grid[row][col]] += 1;

            if (
                box[row / 3][col / 3][grid[row][col]] > 1
                || column_[col][grid[row][col]] > 1
                || row_[row][grid[row][col]] > 1)
```

```
                return false;
        }
    }
    return true;
}

/* A utility function to print grid */
void printGrid(int grid[N][N])
{
    for (int row = 0; row < N; row++) {
        for (int col = 0; col < N; col++)
            cout << grid[row][col] << " ";
        cout << endl;
    }
}

/* Takes a partially filled-in grid and attempts
to assign values to all unassigned locations in
such a way to meet the requirements for
Sudoku solution (non-duplication across rows,
columns, and boxes) */
bool SolveSudoku(
    int grid[N][N], int i, int j)
{
    if (i == N - 1 && j == N) {
        if (isSafe(grid)) {
            printGrid(grid);
            return true;
        }
        return false;
    }
    if (j == N) {
        i++;
        j = 0;
    }
    if (grid[i][j] != UNASSIGNED)
        return SolveSudoku(grid, i, j + 1);
    for (int num = 1; num <= 9; num++) {
        grid[i][j] = num;

        if (SolveSudoku(grid, i, j + 1))
```

```
            return true;

        grid[i][j] = 0;
    }
    return false;
}

int main()
{
        int grid[N][N] = { { 3, 1, 6, 5, 7, 8, 4, 9, 2 },
                           { 5, 2, 9, 1, 3, 4, 7, 6, 8 },
                           { 4, 8, 7, 6, 2, 9, 5, 3, 1 },
                           { 2, 6, 3, 0, 1, 5, 9, 8, 7 },
                           { 9, 7, 4, 8, 6, 0, 1, 2, 5 },
                           { 8, 5, 1, 7, 9, 2, 6, 4, 3 },
                           { 1, 3, 8, 0, 4, 7, 2, 0, 6 },
                           { 6, 9, 2, 3, 5, 1, 8, 7, 4 },
                           { 7, 4, 5, 0, 8, 6, 3, 1, 0 } };
    if (SolveSudoku(grid, 0, 0) != true)
        cout << "No solution exists";
    return 0;
}
```

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

Process returned 0 (0x0)   execution time : 17.035 s
Press any key to continue.
```



C. Solving sudoku using Constraint Propagation Algorithm

Algorithm:

Two important strategies that we can use to make progress towards filling in all the squares:

(1) If a square has only one possible value, then eliminate that value from the square's peers.

(2) If a unit has only one possible place for a value, then put the value there.

As an example of strategy (1) if we assign 7 to A1, yielding {'A1': '7', 'A2':'123456789', ...}, we see that A1 has only one value, and thus the 7 can be removed from its peer A2 (and all other peers), giving us {'A1': '7', 'A2': '12345689', ...}. As an example of strategy (2), if it turns out that none of A3 through A9 has a 3 as a possible value, then the 3 must belong in A2, and we can update to {'A1': '7', 'A2':'3', ...}. These updates to A2 may in turn

cause further updates to its peers, and the peers of those peers, and so on. This process is called constraint propagation.

Code for constraint Propagation:

```
#include <iostream>
#include <assert.h>
#include <unistd.h>
#include <stdexcept>

constexpr int n = 9;
constexpr int nn = 3;


typedef char grid_t[n][n];

typedef unsigned short notes_t[n][n];

int verbose = 0;
long guesses = 0;

class sudoku {
    grid_t grid;
    notes_t notes;
public:
    sudoku();

    template<typename Iter> explicit sudoku(Iter first, Iter last);

    sudoku(std::initializer_list<int> const &l) : sudoku(l.begin(), l.end()) {}
    template<typename Container> explicit sudoku(Container const &c) :
sudoku(c.begin(), c.end()) {}

    bool solve();
    void set_cell(int i, int j, int v);
    bool search(int i, int j);
    bool verify() const;
    void print_grid(char rowsep = '\n') const;
    void print_notes() const;
```

```
};

sudoku::sudoku() : grid {0} {
    for (int i = 0; i < n; ++i) {
        for (int j = 0; j < n; ++j) {
            notes[i][j] = 0x1ff;
        }
    }
}

template<typename Iter>
sudoku::sudoku(Iter first, Iter last) : sudoku() {
    int i = 0;
    for ( ; first != last; ++i, ++first) {
        if (i >= n * n) { throw std::length_error("sudoku constructor:
initializer too long"); }

        auto v = *first;
        if (v < 0 || v > 9) { throw std::out_of_range("sudoku constructor:
element out of range"); }
        set_cell(i / n, i % n, v);
    }

    if (i < n * n) { throw std::length_error("sudoku constructor: initializer
too short"); }
}

class findbit {
    char setbit[1 << n];
public:
    findbit() : setbit {0} {
        for (int i = 0; i < n; ++i) {
            setbit[1 << i] = i + 1;
        }
    }
    // Returns the 1-based bit that is set if there is a single bit, 0
    // otherwise.
    int operator()(int i) const {
        assert(i < sizeof setbit);
        return setbit[i];
    }
```

```
};
findbit findbit;
int
unknown_count(const grid_t &grid) {
    int unknowns = 0;
    for (int i = 0; i < n * n; i++) {
        if (grid[i / n][i % n] == 0) {
            ++unknowns;
        }
    }
    return unknowns;
}

void
sudoku::print_grid(char rowsep) const {
    for (int i = 0; i < n * n; ++i) {
        std::cout
            << static_cast<int>(grid[i / n][i % n])
            << (i == n * n - 1 ? '\n' : i % n == n-1 ? rowsep : ' ');
    }
}

void
sudoku::print_notes() const {
    for (int i = 0; i < n; ++i) {
        for (int j = 0; j < n; ++j) {
            assert(notes[i][j] < 0x200);
            for (int k = 0; k < 9; k++) {
                std::cout << static_cast<char>((notes[i][j] & (1 << k)) ? '1' +
k : ' ');
            }
            std::cout << (j == n-1 ? "\n" : j % nn == nn-1 ? " | " : " : ");
        }
        if (i % nn == nn - 1 && i != n - 1) {
std::cout<<"----------+-----------+-----------+-----------+-----------+-------
----+-----------+-----------+----------\n";
        }
    }
    std::cout << '\n';
}
inline int valmask(int v) { return 1 << (v - 1); }
```

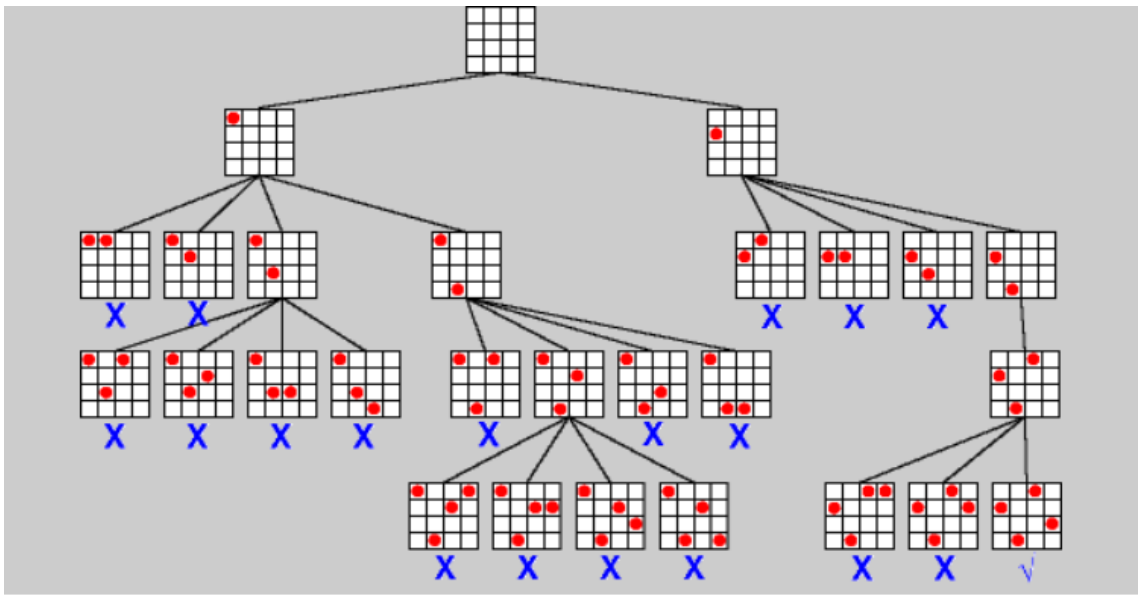
```
void
sudoku::set_cell(int i, int j, int v) {
    if (v == 0) { return; }
    // Update the cell value.
    grid[i][j] = v;
    notes[i][j] = 0;
    // a and b are top-left corner of square containing i, j
    int a = (i / nn) * nn;
    int b = (j / nn) * nn;
    // Update the notes for the cells row, column, and square.
    int vmask = ~ valmask(v) & 0x1ff;
    for (int k = 0; k < n; ++k) {
        // std::printf("  vmask: %x %x -> %x\n", vmask, notes[i][k], notes[i][k]
& vmask);
        notes[i][k] &= vmask;                   // propagate to row.
        notes[k][j] &= vmask;                   // propagate to column.
        notes[a + (k % nn)][b + (k / nn)] &= vmask;     // propagate to square.
    }
}
bool
sudoku::search(int i, int j) {
    // Find a blank cell starting from grid[i][j].
    for ( ; i < n; ++i) {
        for ( ; j < n; ++j) {
            if (grid[i][j] == 0) {
                goto found_blank;
            }
        }
        j = 0;
    }
    // If we get here the sudoku is solved.
    return true;
found_blank:
    int bits = notes[i][j];

    // if bits is 0 then we have found a blank cell but there are
    // no allowed candidates for it.
    if (bits == 0) {
        return false;
    }
    // For each of the allowed candidates for this cell, set its value to the
```

```
    // candidate and then attempt to solve the rest of the puzzle.
    for (int k = 0; k < n; ++k) {
        if (bits & (1 << k)) {
            sudoku search_state = *this;
            int v = k + 1;
            search_state.set_cell(i, j, v);
            if (search_state.search(i, j)) {
                *this = search_state;
                return true;
            }
            ++guesses;
        }
    }
    return false;
}
bool
sudoku::solve() {
    if (verbose) {
        std::printf("unknowns    before    constraint    propagation:    %d\n",
unknown_count(grid));
    }
    // Make one or more passes attempting to update the grid with values we
    // are certain about based on initial notes.  For puzzles that are not too
    // difficult this will find some values and reduce the amount of searching
    // we have to do.
    int found_update = 1;
    while (found_update) {
        found_update = 0;
        // print_notes();
        for (int i = 0; i < n; ++i) {
            for (int j = 0; j < n; ++j) {
                if (grid[i][j] == 0) {
                    int v = findbit(notes[i][j]);
                    if (v > 0) {
                        set_cell(i, j, v);
                        found_update = 1;
                    }
                }
            }
        }
    }
```

```
    // print_grid();
    if (verbose) {
        std::printf("unknowns    after    constraint    propagation:    %d\n",
unknown_count(grid));
    }
    // Now do search to find missing values.
    bool found = search(0, 0);
    return found;
}
bool
sudoku::verify() const {
    for (int k = 0; k < n; ++k) {
        // a and b are top-left corner of square k.
        int a = (k / nn) * nn;
        int b = (k % nn) * nn;
        int r = 0;
        int c = 0;
        int s = 0;
        for (int l = 0; l < n; ++l) {
            r |= 1 << grid[k][l];                       // validate row.
            c |= 1 << grid[l][k];                       // validate column.
            s |= 1 << grid[a + (l % nn)][b + (l / nn)]; // Validate square.
        }
        if (r != 0x3fe || c != 0x3fe || s != 0x3fe) {
            std::printf("%d r:%x c:%x s:%x\n", k, r, c, s);
            return false;
        }
    }
    return true;
}
void
skip_comment() {
    if ((std::cin >> std::ws).peek() == '#') {
        while (std::cin.get() != '\n') { ; }
    }
}
int main(int argc, char **argv) {
    char rowsep = '\n';

    skip_comment();
    int numtests = 0;
```

```
    std::cin >> numtests;
    for (int t = 0; t != numtests; ++t) {
        guesses = 0;
        sudoku sudoku;
        for (int i = 0; i < n * n; ++i) {
            skip_comment();

            int v;
            std::cin >> v;
            if (v < 0 || v > 9) {
                std::printf("Bad grid[%d][%d] value %d\n", i / n, i % n, v);
                exit(1);
            }
            sudoku.set_cell(i / n, i % n, v);
        }

        std::cout << "\n\n\n\n\n\n";


        // sudoku.print_grid();
        sudoku.solve();
        if (verbose) { std::printf("guesses: %ld\n", guesses); }
        if (! sudoku.verify()) { std::cout << "no valid solution found\n"; }
        sudoku.print_grid(rowsep);
    }
    return 0;
}
```

```
C:\Users\DeLL\Desktop\project\DSA Codes For Project>g++ ConstraintPropogation.cpp

C:\Users\DeLL\Desktop\project\DSA Codes For Project>a.exe
1
9 8 0 1 0 0 0 0 0
0 0 0 0 0 4 0 0 0
0 7 0 0 0 0 6 2 0
2 0 0 8 0 0 4 0 0
0 3 0 0 0 0 7 0 0
0 0 6 0 0 0 0 0 9
0 4 0 0 0 0 0 0 3
8 0 9 0 1 7 0 5 0
0 0 7 0 8 0 0 1 0









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



D. Solving Sudoku using Brute Force Search Algorithm

Algorithm and Code

About Brute Force Search Algorithm:

BFS is one of the most general and basic techniques for searching for possible solutions to a problem. The idea is that we want to generate every possible move within the game and then test to see whether it solves our problem. The step-by-step process is the following:

1. Generate a possible move that follows the rules of the game and has not been tested yet.
2. Test to see if that move wins the game/is a solution.
3. If the move wins the game, exit since you have found your solution! If the move does not win the game, add it to the list of attempted moves so you don't attempt it again.
4. The program would solve a puzzle by placing the digit "1" in the first cell and checking if it is allowed to be there. If there are no violations (checking row, column, and box constraints) then the algorithm advances to the next cell, and places a "1" in that cell. When checking for violations, if it is discovered that the "1" is not allowed, the value is advanced to "2". The process continues until appropriate value is filled in that particular box.

Algorithm:

1. First off, we'll set up the Sudoku grid and define a print function to print it out. As explained in the code below, certain global variables will be declared to define the actual grid and how we want it to be printed. The Sudoku grid will be defined as a 2D array of integer type. Also, one important point to be noted here is that all "BLANK" cells should be filled with 0s.

2. Next, we'll go about conveying the rules of the game to the program. Permitted moves, logic, basic rules, and telling what is considered a win is done in this part of the program. The functions—"used_in_row",

“used_in_col” and “used_in_box”—check whether a particular name has appeared in either one of the areas it is named after. The “get_unassigned_location” function is the one which loops through the grid to fill in each cell one-by-one. The “is_safe” function checks to see if placing a given number in a particular cell is legal i.e it doesn’t violate the rules of the game that are defined by the first three functions.

3. Finally, for a particular cell, we loop through all possible digits from 1 to 9 and try them out. If a digit is valid for that cell, we move on to the next unassigned cell. If we find that for a particular cell that there are no valid digits, then we **return false,** then **backtrack,** incriminating the digit in the previous cell and trying again. If we are able to assign a valid number to every cell in the grid i.e the array is totally full, then the recursive algorithm will **return true**, jumping out of all the recursive loops having filled in the whole grid with valid numbers.

Code:

```
#include <iostream>

#include <algorithm>

#include <vector>

#include <array>



using std::cout;

using std::endl;



#define DIM 9

#define BLANK 0

#define SPACE " "

#define LINE "/"
```

```
#define NEW_ROW "-----------------------------------"

#define GRID_FULL std::make_pair(9, 9)



// Prints the Soduko grid

void print_grid(int grid[DIM][DIM])

{

            for (int i = 0; i < DIM; i++)

            {

                        cout << SPACE << SPACE << SPACE << SPACE << endl;

                        cout << NEW_ROW << endl;

                        for (int j = 0; j < DIM; j++)

                        {

                                    cout << SPACE;

                                    if (BLANK == grid[i][j])

                                    {

                                           cout << SPACE;

                                    }

                                    else

                                    {

                                           cout << grid[i][j];

                                    }

                                    cout << SPACE;

                                    cout << LINE;

                        }
```

```
            }
            cout << endl << NEW_ROW << endl << endl;;
}

// Returns a boolean which indicates whether any assigned entry
// in the specified row matches the given number.
bool used_in_row(int grid[DIM][DIM], int row, int num)
{
            for (int col = 0; col < DIM; col++)
                        if (grid[row][col] == num)
                        {
                                    return true;
                        }
            return false;
}

// Returns a boolean which indicates whether any assigned entry
// in the specified column matches the given number.
bool used_in_col(int grid[DIM][DIM], int col, int num)
{
            for (int row = 0; row < DIM; row++)
                        if (grid[row][col] == num)
                        {
                                    return true;
```

```
                        }
            return false;
}

// Returns a boolean which indicates whether any assigned entry
// within the specified 3x3 box matches the given number.
bool used_in_box(int grid[DIM][DIM], int box_start_rpw, int box_start_col, int num)
{
            for (int row = 0; row < 3; row++)
                        for (int col = 0; col < 3; col++)
                                    if (grid[row + box_start_rpw][col + box_start_col] == num)
                                    {
                                           return true;
                                    }
            return false;
}

// Returns a boolean which indicates whether it will be legal to assign
// num to the given row,col location.
bool is_safe(int grid[DIM][DIM], int row, int col, int num)
{
            // Check if 'num' is not already placed in current row,
            // current column and current 3x3 box
```

```
            return !used_in_row(grid, row, num) &&
                        !used_in_col(grid, col, num) &&
                        !used_in_box(grid, row - row % 3, col - col % 3, num);
}

// Searches the grid to find an entry that is still unassigned. If
// found, the reference parameters row, col will be set the location
// that is unassigned, and true is returned. If no unassigned entries
// remain, false is returned.
std::pair<int, int> get_unassigned_location(int grid[DIM][DIM])
{
            for (int row = 0; row < DIM; row++)
                        for (int col = 0; col < DIM; col++)
                                    if (grid[row][col] == BLANK)
                                    {
                                           return std::make_pair(row, col);
                                    }
            return GRID_FULL;
}

// Takes a partially filled-in grid and attempts to assign values to
// all unassigned locations in such a way to meet the requirements
// for Sudoku solution (non-duplication across rows, columns, and boxes)
bool solve_soduko(int grid[DIM][DIM])
```

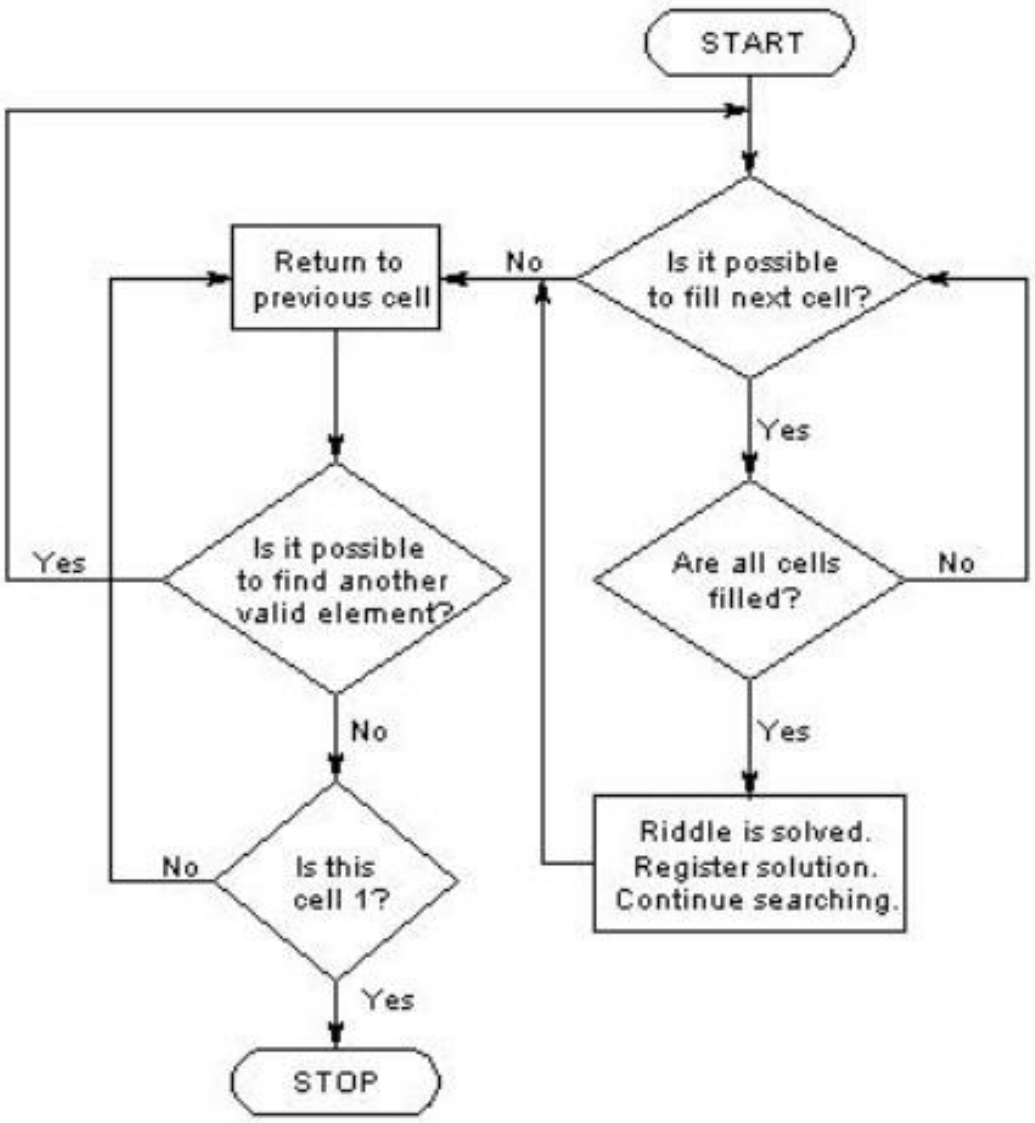
```
{
        // If the Soduko grid has been filled, we are done
        if (GRID_FULL == get_unassigned_location(grid))
        {
                return true;
        }

        // Get an unassigned Soduko grid location
        std::pair<int, int> row_and_col = get_unassigned_location(grid);
        int row = row_and_col.first;
        int col = row_and_col.second;

        // Consider digits 1 to 9
        for (int num = 1; num <= 9; num++)
        {
                // If placing the current number in the current
                // unassigned location is valid, go ahead
                if (is_safe(grid, row, col, num))
                {
                        // Make tentative assignment
                        grid[row][col] = num;

                // Do the same thing again recursively. If we go
               // through all of the recursions, and in the end
```

```
                        // return true, then all of our number placements
                       // on the Soduko grid are valid and we have fully
                                        // solved it
                                        if (solve_soduko(grid))
                                        {
                                               return true;
                                        }

                     // As we were not able to validly go through all
                     // of the recursions, we must have an invalid number
                     // placement somewhere. Lets go back and try a
                     // different number for this particular unassigned location
                                        grid[row][col] = BLANK;
                            }
             }

   // If we have gone through all possible numbers for the current unassigned
   // location, then we probably assigned a bad number early. Lets backtrack
   // and try a different number for the previous unassigned locations.
             return false;
}

int main()
{
```

```
cout << "\n\tSudoku Solver\nUsing Brute Force Search Algorithm\n" << endl << endl;


int grid[DIM][DIM] = { { 0, 9, 0, 0, 0, 0, 8, 5, 3 },
                                                                                { 0, 0, 0, 8, 0, 0, 0, 0, 4 },
                                                                                { 0, 0, 8, 2, 0, 3, 0, 6, 9 },
                                                                                { 5, 7, 4, 0, 0, 2, 0, 0, 0 },
                                                                                { 0, 0, 0, 0, 0, 0, 0, 0, 0 },
                                                                                { 0, 0, 0, 9, 0, 0, 6, 3, 7 },
                                                                                { 9, 4, 0, 1, 0, 8, 5, 0, 0 },
                                                                                { 7, 0, 0, 0, 0, 6, 0, 0, 0 },
                                                                                { 6, 8, 2, 0, 0, 0, 0, 9, 0 } };


            print_grid(grid);


            if (true == solve_soduko(grid))
            {
                        print_grid(grid);
            }
            else
```

```
            {

                        cout << "No solution exists for the given Soduko" <<
endl << endl;

            }



            return 0;

}
```

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

The time taken to execute the code Brute Force Search is 70 ms
```



E. Solving Sudoku using Knuth's Algorithm (Dancing Links)

Algorithm:

- If matrix A is empty, the problem is solved. Return success;
- Otherwise, choose a column c in matrix A;
- Choose a row r in column c;
- Include line r in the partial solution;
- For each index j such that A[r,j] = 1:

- Remove the index column j from matrix A;
- For each index i such that A[i,j] = 1:
- Remove the index line i from matrix A;

- Repeat the algorithm recursively on the reduced matrix A.

The algorithm is based on a very simple technique which Knuth thinks should be better known. Given a node x which points to two elements in a doubly linked list, L[x] points to the left element of x and R[x] points to the right element of x. For instance, L[R[x]] is pointing to x if the right element of x is pointing its left element back to x. This is easier to see if we let y = R[x] such that L[R[x]] = L[y], which describes the left element of y.

Code:

```
#include <iostream>
#include <cmath>
#include <string>
#include <ctime>
#define MAX_K 1000
#define SIZE 9
struct Node {
	Node *left;
	Node *right;
	Node *up;
	Node *down;
	Node *head;
	int size;
```

```
	int rowID[3];
};
const int SIZE_SQUARED = SIZE*SIZE;
const int SIZE_SQRT = sqrt((double)SIZE);
const int ROW_NB = SIZE*SIZE*SIZE;
const int COL_NB = 4 * SIZE*SIZE;
struct Node Head;
struct Node* HeadNode = &Head;
struct Node* solution[MAX_K];
struct Node* orig_values[MAX_K];
bool matrix[ROW_NB][COL_NB] = { { 0 } };
bool isSolved = false;
void MapSolutionToGrid(int Sudoku[][SIZE]);
void PrintGrid(int Sudoku[][SIZE]);
clock_t timer, timer2;
void coverColumn(Node* col) {
	col->left->right = col->right;
	col->right->left = col->left;
	for (Node* node = col->down; node != col; node = node->down) {
		for (Node* temp = node->right; temp != node; temp = temp->right) {
			temp->down->up = temp->up;
			temp->up->down = temp->down;
			temp->head->size--;
		}
	}
}
void uncoverColumn(Node* col) {
	for (Node* node = col->up; node != col; node = node->up) {
		for (Node* temp = node->left; temp != node; temp = temp->left)
{
			temp->head->size++;
			temp->down->up = temp;
			temp->up->down = temp;
		}
	}
	col->left->right = col;
	col->right->left = col;
}
void search(int k) {
	if (HeadNode->right == HeadNode) {
		timer2 = clock() - timer;
```

```
            int Grid[SIZE][SIZE] = { {0} };
            MapSolutionToGrid(Grid);
            PrintGrid(Grid);
            std::cout << "Time Elapsed: " << (float)timer2 / CLOCKS_PER_SEC <<
" seconds.\n\n";
            std::cin.get(); //Pause console
            timer = clock();
            isSolved = true;
            return;
      }
      Node* Col = HeadNode->right;
      for (Node* temp = Col->right; temp != HeadNode; temp = temp->right)
            if (temp->size < Col->size)
                  Col = temp;
      coverColumn(Col);
      for (Node* temp = Col->down; temp != Col; temp = temp->down) {
            solution[k] = temp;
            for (Node* node = temp->right; node != temp; node = node->right) {
                  coverColumn(node->head);
            }
            search(k + 1);
            temp = solution[k];
            solution[k] = NULL;
            Col = temp->head;
            for (Node* node = temp->left; node != temp; node = node->left) {
                  uncoverColumn(node->head);
            }
      }
      uncoverColumn(Col);
}
void BuildSparseMatrix(bool matrix[ROW_NB][COL_NB]) {
      int j = 0, counter = 0;
      for (int i = 0; i < ROW_NB; i++) { //iterate over all rows
            matrix[i][j] = 1;
            counter++;
            if (counter >= SIZE) {
                  j++;
                  counter = 0;
            }
      }
      int x = 0;
```

```
	counter = 1;
	for (j = SIZE_SQUARED; j < 2 * SIZE_SQUARED; j++) {
		for (int i = x; i < counter*SIZE_SQUARED; i += SIZE)
			matrix[i][j] = 1;

		if ((j + 1) % SIZE == 0) {
			x = counter*SIZE_SQUARED;
			counter++;
		}
		else
			x++;
	}
	j = 2 * SIZE_SQUARED;
	for (int i = 0; i < ROW_NB; i++)
	{
		matrix[i][j] = 1;
		j++;
		if (j >= 3 * SIZE_SQUARED)
			j = 2 * SIZE_SQUARED;
	}
	x = 0;
	for (j = 3 * SIZE_SQUARED; j < COL_NB; j++) {
		for (int l = 0; l < SIZE_SQRT; l++) {
			for (int k = 0; k<SIZE_SQRT; k++)
				matrix[x + l*SIZE + k*SIZE_SQUARED][j] = 1;
		}
		int temp = j + 1 - 3 * SIZE_SQUARED;
		if (temp % (int)(SIZE_SQRT * SIZE) == 0)
			x += (SIZE_SQRT - 1)*SIZE_SQUARED + (SIZE_SQRT - 1)*SIZE + 1;
		else if (temp % SIZE == 0)
			x += SIZE*(SIZE_SQRT - 1) + 1;
		else
			x++;
	}
}
	Node* header = new Node;
	header->left = header;
	header->right = header;
	header->down = header;
	header->up = header;
	header->size = -1;
```

```
header->head = header;
Node* temp = header;
for (int i = 0; i < COL_NB; i++) {
	Node* newNode = new Node;
	newNode->size = 0;
	newNode->up = newNode;
	newNode->down = newNode;
	newNode->head = newNode;
	newNode->right = header;
	newNode->left = temp;
	temp->right = newNode;
	temp = newNode;
}
int ID[3] = { 0,1,1 };
for (int i = 0; i < ROW_NB; i++) {
	Node* top = header->right;
	Node* prev = NULL;
	if (i != 0 && i%SIZE_SQUARED == 0) {
		ID[0] -= SIZE - 1;
		ID[1]++;
		ID[2] -= SIZE - 1;
	}
	else if (i!= 0 && i%SIZE == 0) {
		ID[0] -= SIZE - 1;
		ID[2]++;
	}
	else {
		ID[0]++;
	}
	for (int j = 0; j < COL_NB; j++, top = top->right) {
		if (matrix[i][j]) {
			Node* newNode = new Node;
			newNode->rowID[0] = ID[0];
			newNode->rowID[1] = ID[1];
			newNode->rowID[2] = ID[2];
			if (prev == NULL) {
				prev = newNode;
				prev->right = newNode;
			}
			newNode->left = prev;
			newNode->right = prev->right;
```

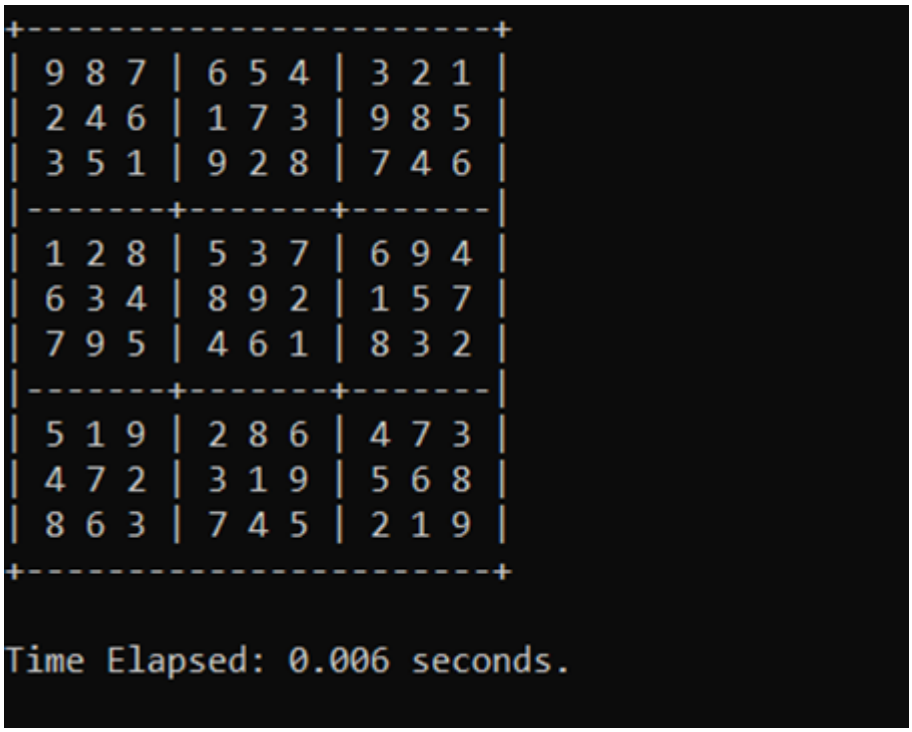
```
                        newNode->right->left = newNode;
                        prev->right = newNode;
                        newNode->head = top;
                        newNode->down = top;
                        newNode->up = top->up;
                        top->up->down = newNode;
                        top->size++;
                        top->up = newNode;
                        if (top->down == top)
                                top->down = newNode;
                        prev = newNode;
                }
            }
      }
      HeadNode = header;
}
void TransformListToCurrentGrid(int Puzzle[][SIZE]) {
      int index = 0;
      for(int i = 0 ; i<SIZE; i++ )
            for(int j = 0 ; j<SIZE; j++)
                  if (Puzzle[i][j] > 0) {
                        Node* Col = NULL;
                        Node* temp = NULL;
                        for (Col = HeadNode->right; Col != HeadNode; Col = Col-
>right) {
                              for (temp = Col->down; temp != Col; temp = temp-
>down)
                                    if (temp->rowID[0] == Puzzle[i][j] &&
(temp->rowID[1] - 1) == i && (temp->rowID[2] - 1) == j)
                                          goto ExitLoops;
                        }
ExitLoops:          coverColumn(Col);
                        orig_values[index] = temp;
                        index++;
                        for (Node* node = temp->right; node != temp; node =
node->right) {
                              coverColumn(node->head);
                        }

                  }
}
```

```
void MapSolutionToGrid(int Sudoku[][SIZE]) {

	for (int i = 0; solution[i] != NULL; i++) {
			Sudoku[solution[i]->rowID[1]-1][solution[i]->rowID[2]-1]   =
solution[i]->rowID[0];
	}
	for (int i = 0; orig_values[i] != NULL; i++) {
		Sudoku[orig_values[i]->rowID[1] - 1][orig_values[i]->rowID[2] - 1]
= orig_values[i]->rowID[0];
	}
}

//---------------------------------PRINTS A SUDOKU GRID OF ANY SIZE-----------
---------------------------------//
void PrintGrid(int Sudoku[][SIZE]){
	std::string ext_border = "+", int_border = "|";
	int counter = 1;
	int additional = 0;
	if (SIZE > 9)
		additional = SIZE;
	for (int i = 0; i < ((SIZE +SIZE_SQRT - 1) * 2 +additional+ 1); i++) {
		ext_border += '-';

		if (i > 0 && i % ((SIZE_SQRT*2+SIZE_SQRT*(SIZE>9)+1)*counter +
counter-1) == 0) {
			int_border += '+';
			counter++;
		}
		else
			int_border += '-';
	}
	ext_border += '+';
	int_border += "|";

	std::cout << ext_border << std::endl;
	for (int i = 0; i<SIZE; i++){
		std::cout << "| ";
		for (int j = 0; j<SIZE; j++){
			if (Sudoku[i][j] == 0)
				std::cout << ". ";
			else
```

```
                        std::cout << Sudoku[i][j] << " ";
                if (additional > 0 && Sudoku[i][j]<10)
                        std::cout << " ";
                if ((j+1)%SIZE_SQRT == 0)
                        std::cout << "| ";
            }
            std::cout << std::endl;
            if ((i + 1) % SIZE_SQRT == 0 && (i+1)<SIZE)
                    std::cout << int_border << std::endl;
      }
      std::cout << ext_border << std::endl << std::endl;
}

//-------------------------------------------------------------------------
----//

void SolveSudoku(int Sudoku[][SIZE]) {
      timer = clock();
      BuildSparseMatrix(matrix);
      BuildLinkedList(matrix);
      TransformListToCurrentGrid(Sudoku);
      search(0);
      if (!isSolved)
            std::cout << "No Solution!" << std::endl;
      isSolved = false;
}

int main(){
            //Sudoku Hard to Brute Force
            int Puzzle[9][9] =        {     { 0,0,0,  0,0,0,  0,0,0 },
                                            { 0,0,0,  0,0,3,  0,8,5 },
                                            { 0,0,1,  0,2,0,  0,0,0 },
                                            { 0,0,0,  5,0,7,  0,0,0 },
                                            { 0,0,4,  0,0,0,  1,0,0 },
                                            { 0,9,0,  0,0,0,  0,0,0 },
                                            { 5,0,0,  0,0,0,  0,7,3 },
                                            { 0,0,2,  0,1,0,  0,0,0 },
                                            { 0,0,0,  0,4,0,  0,0,9 }
                                      };
```

```
    int EmptyPuzzle[SIZE][SIZE] = { {0} };

    SolveSudoku(Puzzle);

    std::cin.get();
    return 0;
}
```

Output:



List of Modules and Module Description

1. Backtracking (Non-Recursive)

Backtracking is a general algorithm for finding all (or some) solutions to some computational problems, notably constraint satisfaction

problems, that incrementally builds candidates to the solutions, and abandons each partial candidate ("backtracks") as soon as it determines that the candidate cannot possibly be completed to a valid solution.

Function PrintGrid: To print the sudoku puzzle and solution as a grid
Function FindUnassignedLocation: Searches the grid to find an entry that is still unassigned. If found, the reference parameters row, col will be set the location that is unassigned, and true is returned. If no unassigned entries remain, false is returned.
Function isSafe: Returns a boolean which indicates whether it will be legal to assign a particular number to the given row and column location.
Function SolveSudoku: Takes a partially filled-in grid and attempts to assign values to all unassigned locations in such a way to meet the requirements for Sudoku solution (non-duplication across rows, columns, and boxes)
Function UsedInRow: Returns a boolean which indicates whether an assigned entry in the specified row matches the given number.
Function UsedInCol: Returns a boolean which indicates whether an assigned entry in the specified column matches the given number.
Function UsedInBox: Returns a boolean which indicates whether an assigned entry within the specified 3x3 box matches the given number.

2. Backtracking (Recursive)

Backtracking can be defined as a general algorithmic technique that considers searching every possible combination to solve a computational problem.
When a function calls itself, it's called Recursion.
Generally, every constraint satisfaction problem which has clear and well-defined constraints on any objective solution, that incrementally builds candidate to the solution and abandons a candidate

(“backtracks”) as soon as it determines that the candidate cannot possibly be completed to a valid solution, can be solved by Backtracking. However, most of the problems that are discussed, can be solved using other known algorithms like Dynamic Programming or Greedy Algorithms in logarithmic, linear, linear-logarithmic time complexity in order of input size, and therefore, outshine the backtracking algorithm in every respect (since backtracking algorithms are generally exponential in both time and space). However, a few problems still remain, that only have backtracking algorithms to solve them until now.

3. Constraint Propagation

Constraint propagation is an algorithm that solves Sudoku problems by using possibility arrays and multiple iterations. For this algorithm we will initially fill all the space with the values given in the question, after that we will put zeroes in the blank locations. In this we will find all the possible numbers for a particular blank location and we will have an array for each blank location, if for a particular blank location there is only one possibility then we will directly insert the value in that location, otherwise we will have multiple iterations following to complete the Sudoku problem. In some cases where the Sudoku problem is of hard or extreme level, we won’t have a possible value after multiple iterations. For most of the cases constraint propagation will be enough to fill the grid completely and without any conflict.

4. Brute Force Search

Brute Force Algorithms are straightforward methods of solving a problem that rely on sheer computing power and trying every possibility rather than advanced techniques to improve efficiency.

For example, imagine having a small padlock with 4 digits, each from 0-9. Imagine forgetting your combination, but don't want to buy another padlock. Since you can't remember any of the digits, you have to use a brute force method to open the lock.

So you set all the numbers back to 0 and try them one by one: 0001, 0002, 0003, and so on until it opens. In the worst case scenario, it would take 10^4, or 10,000 tries to find your combination.

5. Knuth's Algorithm

Dancing Links, or DLX for short, is the technique suggested by Donald Knuth for implementing Algorithm X efficiently. Given a binary matrix, DLX will represent the 1s as data objects. Each data object x has the fields L[x], R[x], U[x], D[x] and C[x]. The fields are for linking to any other cell with an occupying 1 to the left, right, up and down. Any link that has no corresponding 1 in a suitable cell will link to itself instead. The last field is a link to the column object y which is a special data object that has two additional fields, S[y] and N[y], which represents the column size and a symbolic name chosen arbitrarily. The column size is the number of data objects that are currently linked together from the column object. Each row and each column is a circular doubly linked list, and if a data object is removed from the column, the size is decremented. Remember that the removed data object still has the links pointing as they did before the data object was removed. One of the most important parts in this thesis is the algorithm for reducing a grid to an exact cover problem. Without understanding this reduction, the Dancing Links algorithm cannot be used for the purpose of solving a grid. The rules of Sudoku were described in the introduction and every rule can be described as a constraint. Cell Each cell can only contain one integer between 1 and 9. Row Each row can only contain nine unique integers in the range of 1 to 9. Column Each column can only contain nine unique integers in the range of 1 to 9. Box Each box can only contain nine unique integers in the range of 1 to 9. Consider a grid G with at least 17 clues and with a unique solution. The clues will have an effect on all four constraints since a clue cannot be moved or changed.

The relationship between the clues will then decide in which cells in the row, column and box the remaining integers can be placed. In other words, the clue determines the candidates for the remaining cells on the same row, column and box. The reduction from the grid G must preserve the constraints in a binary matrix M. In M there must then exist a selection of rows such that a union between them covers all columns, otherwise there is no solution. This reduction does not seem plausible at first glance since a binary matrix can only contain 1s and 0s, and a grid can only contain integers between 1 and 9. It also does not seem plausible to preserve the constraints for all cells. What makes this reduction possible is having each row in M describing all four constraints for each cell in G. When a row is chosen from M as part of the solution, what is actually chosen is an integer which complies with all four constraints for the cell in G that the row describes. Obtaining the solution to the grid is quite easy but requires a slight modification of DLX. Each data object must also include a field for storing the row placement in the original binary matrix M, because when M is transformed by DLX into the links, there is no way of knowing from the found solution which data object is for which row in M. When a solution is found, the list of data objects must be sorted in descending order based on their row number. The solution should have 81 rows, one for each cell in the grid that is being solved. Since our domain is 1 to 9, we will use modulus 9 with every row. Before applying modulus, we must increment the row number with one since the implementation deals with zero-based indices while the solution does not. Thus the solution found by DLX can because of this be easily transformed back into the original grid again but without any unknown cells. The grid can then be verified if it complies with the constraints or not.

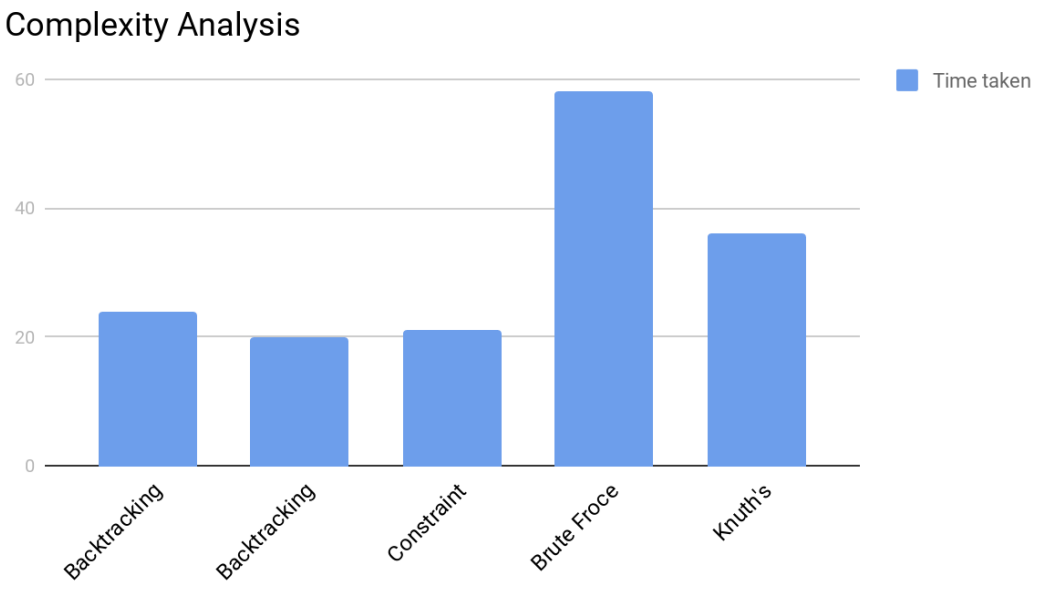
Results and Discussion

The time taken on average by the covered 5 algorithm for a a 9x9 sudoku problem are:

1. Backtracking (Non- Recursive) takes on average 25ms to solve a sudoku puzzle.
2. Backtracking (Recursive) takes on average 20ms to solve a sudoku puzzle.
3. Constraint propagation takes an average 20ms to solve a sudoku puzzle.
4. Brute Force Search takes on average about 60ms to solve a sudoku puzzle.
5. Knuth's Algorithm takes on average about 37ms to solve a sudoku puzzle

Detailed time complexity analysis of modules and its graphical representations

On the basis of complexity and time taken we can make the following graph depicting the effectiveness and efficiency of the algorithm.



1. Backtracking

Time complexity: O(9^(n*n)).
For every unassigned index there are 9 possible options so the time complexity is O(9^(n*n)).

2. Backtracking(Recursive)

Time complexity: O(9^(n*n)).
For every unassigned index there are 9 possible options so the time complexity is O(9^(n*n)).

3. Constraint Propagation

The time complexity of (d,k)-CP, the problem of deciding satisfiability of a constraint system.

Open image in new window

c with n variables, domain size d, and at most k variables per constraint require exponential time $O(c^*n)$ for some $c>0$ independent of d.

4. Brute Force Search

The total time complexity of the algorithm becomes O(n1r). This sounds like a very high complexity to solve the Sudoku. The brute force algorithm requires O(np) time complexity. The benefit of this algorithm is that by applying constraint after every permutation we can reduce the number of possibilities in permutation can be applied.

5. Knuth's Algorithm

Theoretically, the computation time of Algorithm X is O(N^3), where N equals to 9.

References

A. Simonis, H. (2005, October). Sudoku as a constraint problem. In *CP Workshop on modeling and reformulating Constraint Satisfaction Problems* (Vol. 12, pp. 13-27). Citeseer. (http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.88.2964&rep=rep1&type=pdf)

B. Felgenhauer, B., & Jarvis, F. (2006). Mathematics of sudoku I. *Mathematical Spectrum*, *39*(1), 15-22. (http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.646.8115&rep=rep1&type=pdf)

C. Grados, F., & Mohammadi, A. (2013). A REPORT ON THE SUDOKU SOLVER. (https://www.diva-portal.org/smash/record.jsf?pid=diva2%3A668213&dswid=-9596)

D. Bessière, C., & Régin, J. C. (2001, August). Refining the Basic Constraint Propagation Algorithm. In *IJCAI* (Vol. 1, pp. 309-315). (https://www.researchgate.net/profile/Christian_Bessiere2/publication/220937293_Refining_the_Basic_Constraint_Propagation_Algorithm/links/0912f5089f45feb3d2000000.pdf)

E. Vilain, M. B., & Kautz, H. A. (1986, August). Constraint propagation algorithms for temporal reasoning. In *Aaai* (Vol. 86, pp. 377-382). (https://www.aaai.org/Papers/AAAI/1986/AAAI86-063.pdf)

F. Ekne, S., & Gylleus, K. (2015). Analysis and comparison of solving algorithms for sudoku: Focusing on efficiency. (https://www.diva-portal.org/smash/record.jsf?pid=diva2%3A811020&dswid=-530)

G. Reeson, C. G., Huang, K. C., Bayer, K. M., & Choueiry, B. Y. (2007, July). An interactive constraint-based approach to Sudoku. In *PROCEEDINGS OF THE NATIONAL CONFERENCE ON ARTIFICIAL INTELLIGENCE* (Vol. 22, No. 2, p. 1976). Menlo Park, CA; Cambridge, MA; London; AAAI Press; MIT Press; 1999. (https://www.aaai.org/Papers/AAAI/2007/AAAI07-362.pdf)

H. Job, D., & Paul, V. (2016). Recursive Backtracking for Solving 9* 9 Sudoku Puzzle. *Bonfring International Journal of Data Mining*, *6*(1), 07-09.

(http://www.journal.bonfring.org/abstract.php?id=2&archiveid=484)

I. Felgenhauer, B., & Jarvis, F. (2005). Enumerating possible Sudoku grids. *Preprint available at http://www. afjarvis. staff. shef. ac. uk/sudoku/sudoku. pdf*. (http://members.home.nl/jfhm-bours/Dutch/Projecten/Sudoku/sudoku.pdf)

J. Crook, J. F. (2009). A pencil-and-paper algorithm for solving Sudoku puzzles. *Notices of the AMS*, *56*(4), 460-468. (https://www.ams.org/notices/200904/rtx090400460p.pdf)

K. Cazenave, T., & Labo, I. A. (2006). A search based sudoku solver. *Labo IA Dept. Informatique Universite Paris*, *8*, 93526. (http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.64.459&rep=rep1&type=pdf)

L. Weber, T. (2005, December). A SAT-based Sudoku solver. In *LPAR* (pp. 11-15). (https://www.cs.miami.edu/home/geoff/Conferences/LPAR-12/ShortPapers.pdf#page=15)

M. Schermerhorn, M. (2015). A Sudoku Solver. (https://cs.rochester.edu/~brown/242/assts/termprojs/Sudoku09.pdf)

N. Di Scala, D. L. (2020). *How to Solve a Sudoku-A logical analysis and algorithmic implementation of strategically Solving Sudoku puzzles* (Bachelor's thesis). (https://dspace.library.uu.nl/handle/1874/396282)

O. Lewis, R. (2007, October). On the combination of constraint programming and stochastic search: the Sudoku case. In *International Workshop on Hybrid Metaheuristics* (pp. 96-107). Springer, Berlin, Heidelberg. (https://link.springer.com/chapter/10.1007/978-3-540-75514-2_8)