Backtracking Sudoku Solver

Design and Analysis of Algorithms

Individual Project

Anitha Ganesha

May 3, 2013

Contents

1 Abstract

As a part of this project two back tracking sudoku algorithms were analyzed. The first implementation includes the *naive approach* of solving, where in every the algorithm tries to check recursively for unfilled cell and try different possible options for it. The second approach that was analyzed is called the *Smart backtracking* discussed in class where in the possible values for every unfilled cell is calculated first and the cell with least possibility is chosen first for the solution path. The both approaches to the problem are compared in terms of no of backtracking it takes in order to reach solution. Further analysis is done in terms mapping backtracking steps to the no of possibilities.

2 Introduction

Sudoku puzzle is game that gained popularity recently.It can be solved in wide variety of step by step algorithms.Starting from bruteforce approach of instantiating all solutions and searching for a valid solution to smarter ways of solving it using constraints or rules that are usually applied by humans while solving the puzzle.

2.1 Problem Specification

Though there are vast variety of algorithms to solve this problem exist, this project is limited to only analyzing the backtracking algorithms of solving the sudoku puzzle. Analysis includes measuring the potential of algorithms in solving the puzzle. Two backtracking algorithms are discussed including their implementation details and the challenges faced.

2.2 Puzzle Background

Some Definitions

Box: A $3X3$ grid inside the sudoku puzzle.
Region: A Region is either a row or a column or a box in the puzzle.
Clue: A number in the original sudoku puzzle is called a clue. The clues are used to fill in new cells during the process of solving the puzzle.
Solution: A solution to the sudoku puzzle is a full sudoku grid consistent with the give puzzle.[2]

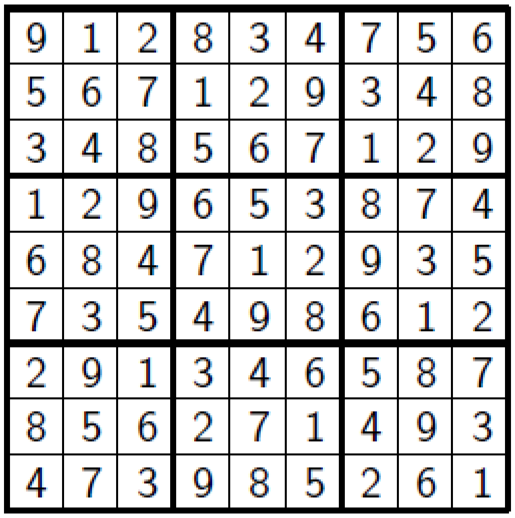
The sudoku puzzle consist of a NXN grid with a set of clues filled in some cells of the grid. The objective of the games is to fill in the rest of cells with a value from 1 to N by following a set constraints. The most common value used for N is 9. The $9X9$ sudoku grid is further divided into set of non-overlapping $3X3$ grids called *Boxes.*

The puzzle consists of three constraints listed below.

- **Row Contraint**: The value of a cell must occur exactly once in a row.
- **Column Constraint**: The value of a cell must occur exactly once in a column.

- **Box Constraint**: The value of a cell must occur exactly once in a Box.

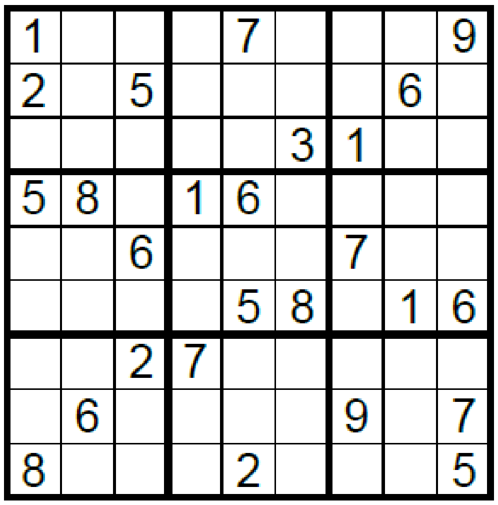
The figure below shows a completey solved sudoku puzzle[2].



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

Figure 1: A Completed Sudoku

The figure below shows a proper sudoku puzzle[2].



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

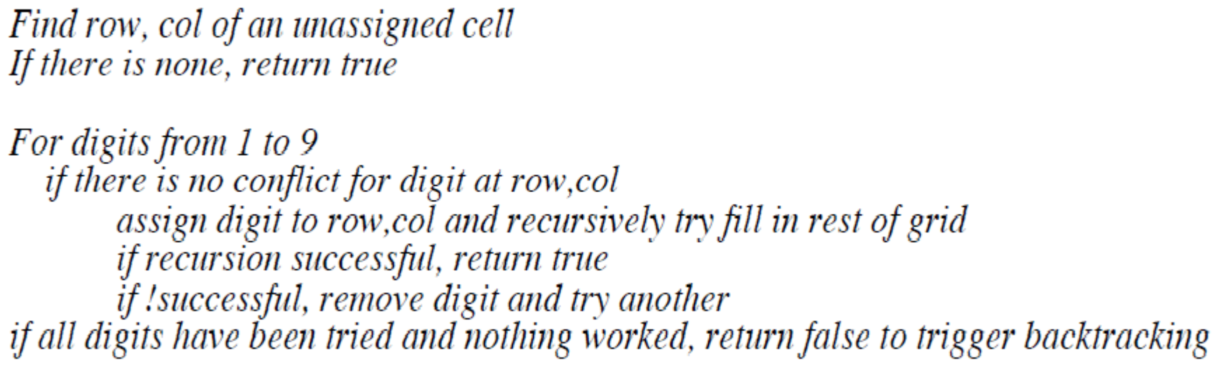
Figure 2: A proper Sudoku Puzzle

Sudoku Puzzles fall under a category of NP-Completeness problem. This means that the algorithm used to solve Sudoku which guarantee a solution to a given problem cannot be solved in polynomial time. However, the solution's correctness can be verified in polynomial time.

3 Backtracking Algorithms

3.1 Naive Backtracking Algorithm

One of the approaches in solving sudoku puzzles is Backtracking. As a part of this project two approaches to backtracking were analyzed.
In the first approach, the algorithm takes in an incomplete sudoku puzzle and recursively checks for an unfilled cell and tries to apply one of the possible values to cell which complies to the constraints discussed above. This way it tries to apply values to every cell until it reaches a point where it finds a solution to the puzzle or hits a dead end at one of the cell where no possibilities can be applied to it. At this point it backtracks to it's recursive caller to check if it has more possibilities to be tried. If available it is applied and makes further calls to non- empty cells. The algorithm ends after finding a solution or after exhaustively checking all possibilities for all cells without any solution. The pseudocode for the same is as shown below.[3]



```
Find row, col of an unassigned cell
If there is none, return true

For digits from 1 to 9
  if there is no conflict for digit at row,col
      assign digit to row,col and recursively try fill in rest of grid
      if recursion successful, return true
      if !successful, remove digit and try another
if all digits have been tried and nothing worked, return false to trigger backtracking
```

Figure 3: Algorithm for Backtracking approach 1

The other main observation to be made here is to understand the differnce between the simple exhaustive recursive search and the backtracking approach.
In exhaustive search no constraints are checked, all the values are simply assigned and solution is validated only when u reach the leaf.If the solution is incorrect it tries to find an other solution path.
On the other hand the backtracking approach is much better than this approach wherein it checks if the constraints are violated at any path, if true it backtrack from the point of violation to an other path immediately saving lot of unnecessary path checks. The figure below explains the difference

between backtracking and exhaustive recursion.[2] The red lines indicate the path traced by the two algorithms.

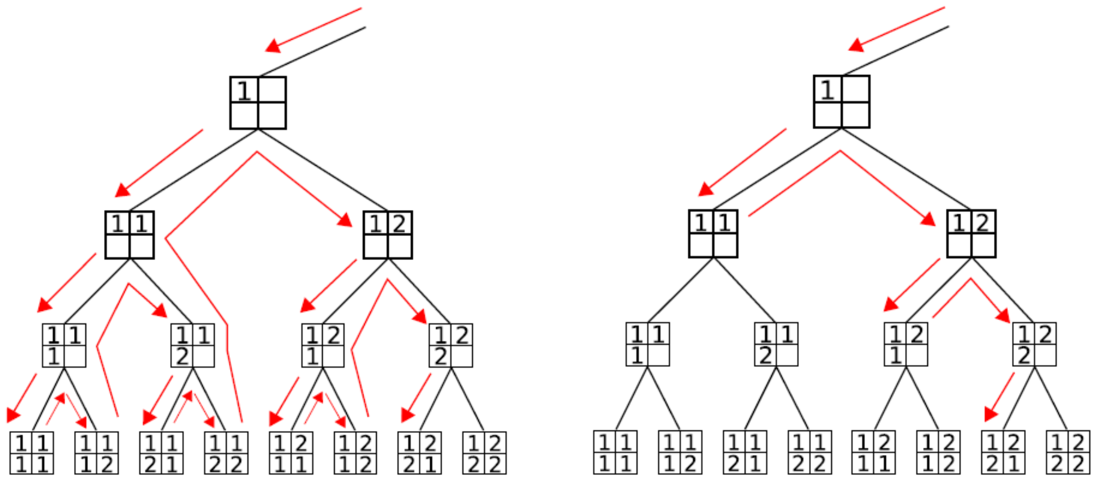


Figure 4: Exhaustive recursion versus backtracking

3.2 Smart Backtracking Algorithm

The Second approach implemented is called ***Smart Backtracking***, the approach that was discussed in one of the crowd source lectures in class.

This algorithm is better compared to the first algorithm in terms of reducing the number of backtracks required in certain scenarios. It differs from the first implementation in choosing the the unfilled cell where the cell with lesser number of possibilities are considered first. The way the recursion breadth for the higher nodes in the tree is reduced which in turn reduces the number of recursion calls.

The algorithm contains a main recursive backtrack pseudo code which first checks if the input is valid, if not it returns null. Next it checks if the input is already a solution.If yes, it returns the input. Otherwise it builds a min queue for the unfilled cells with all the possibilities for the same. It then applies the possibility one by one that is extracted from the minQueue. If the possibility is accepted it goes to the next cell extracted from min queue and applies value to it, else it backtracks to the previous cell exploring different possibility for it.

The pseudo code for the same is as listed in figure 5.[1]

```
function BACKTRACK($\mathcal{S}$)
    if $reject(\mathcal{S})$ then
        return null                                        ▷ Prune the rest of this subtree.
    else if $accept(\mathcal{S})$ then
        return $\mathcal{S}$                                           ▷ We're done!
    end if
    $childQueue \leftarrow getChildQueue(\mathcal{S})$
    $\mathcal{S}' \leftarrow childQueue.next(\mathcal{S})$
    while $\mathcal{S}' \neq$ null $\wedge$ $solution ==$ null do
        $solution \leftarrow backtrack(\mathcal{S}')$
        $\mathcal{S}' \leftarrow childQueue.next(\mathcal{S})$
    end while
    return $solution$
end function
```

Figure 5: SmartBackTracking

The GetChildQueue algorithm tries to build a minQueue for the list of all possibilities for all unfilled cell for the current state of the puzzle. The minQueue is ordered based on the cell with minimim size of the possibility list as shown in figure 6.[1]

The possibility function is responsible for building the list that is used in MinQueue.The list is filtered through checking all the constraints that was discussed earlier.The pseudocode for the same is shown in figure 7[1]. This simple strategy is called *Naked Single Strategy*. There are several other strategies that could be considered including hidden single, naked pair, naked tuple etc which is beyond the scope of this project.

Note: These are the pseudo codes discussed in crowd source lecture.

```
Algorithm 1
  function GetChildQueue(S)
      $minQueue$                               ▷ This is a min-heap ordered by cellSet.length
      for $i \leftarrow 1 \dots n$ do
          for $j \leftarrow 1 \dots n$ do
              if S[i, j] == NULL then
                  new $cellSet$
                  $cellSet.i \leftarrow i$
                  $cellSet.j \leftarrow j$
                  $cellSet.set \leftarrow Possibilities(S, i, j)$
                  $minQueue.push(cellSet)$
              end if
          end for
      end for
      return $minQueue$
  end function
```

Figure 6: Psuedocode for Minqueue implementation for a particular Sudoku state

```
Algorithm 2
  function Possibilities(S, r, c)
      $retList \leftarrow newList\{1, \dots, S.length\}$
      for $i \leftarrow 1 \dots S.length$ do
          if $S_{ri}$ != NULL then
              retList.remove($S_{ri}$)
          else if $S_{ic}$ != NULL then
              retList.remove($S_{ic}$)
          end if
      end for
      for each cell $s$ in $S.getBox(r, c)$ do
          if $s$ != NULL then
              retList.remove($s$)
          end if
      end for
      return $retList$
  end function
```

Figure 7: Pseudocode for possibility function implementing naked single strategy

4 Analysis

4.1 Runtime Asymptotic Analysis

In general all sudoku algorithms are NP-complete problems and hence none of the implemented algorithms run in polynomial time. The analysis of sudoku algorithms is different compared to analysis of all the other algorithms done in problems sets. This is because, the size of the input NXN grid remains constant here. The runtime of a sudoku puzzle depends on the number of clues given, the number of possibilities of different cells to be filled and the variation of the possibilities of different cells. An other observation to be made here is, the runtime is not directly related to the number of clues, i.e., the run time is not decreased with the increase in the number of clues, it is more affected by the possibilities of the other cells to be filled.
The runtime of exhaustive recursion technique is bounded by $\bigcirc(k^n)$ where n is the number of cells that needs to be filled in k is the number of possibilities to be filled in.(assuming all cells have same possibility.
The back tracking approaches however will be much lesser than $\bigcirc(k^n)$ since it checks for constraints in it's path.

4.1.1 Run time analysis of naive approach

The rum time analysis of backtracking approaches can be done in terms of number of back track calls. In this algorithm the backtrack calls happens only when the none of the possibilities fit in a particular cell.Since the cells are considered in a series order always, the number of back-track calls depends on the input. However, the number of backtrack remains constant for a particular since there is no randomness used in the algorithm.

4.1.2 Run time analysis of smart approach

The number of backtracks for this approach depends is lesser or equal to the previous approach. When the possibilities remain same for all the cell, there is no difference between the two approaches and the backtrack calls remain same. However, when there is a difference between the number of possibilities for every cell, then this approach always tries to try the cell with lesser possibility first and hence the back track calls reduces.
This algorithm internally uses minQueue data structure. The different oper-

ation of minQueue is shown in the table below with their asymptotic running time.

No	MinQueue Operations	Asymptotic Running time
1	push	$\bigcirc(logn)$
2	next	$\bigcirc(logn)$
3	queueIncreaseKey	$\bigcirc(logn)$
4	minQueueify	$\bigcirc(logn)$
5	queueMinimum	$\bigcirc(1)$

The minQueue is built by using the push method of minQueue object. Every push operation takes $\bigcirc(logn)$ time since it in turn calls queueIncreasekey function which takes $\bigcirc(logn)$. Similarly, the next operation of minQueue is similar to heapExtractMinimum function and hence takes $\bigcirc(logn)$ as it re adjusts the queue back to minQueue. The QueueMin always fetches the object on top and hence takes $\bigcirc(1)$ time. The minQueue is built only when a new cell is visited for the first time. This has an overhead of $\bigcirc(nlogn)$ time for building, where n here corresponds to the number of possibilities a cell could have.

4.2 Space Requirement Analysis

The space requirement analysis mainly deals with the additional space that is used by the algorithm apart from the input.

4.2.1 Space requirement analysis of naive approach

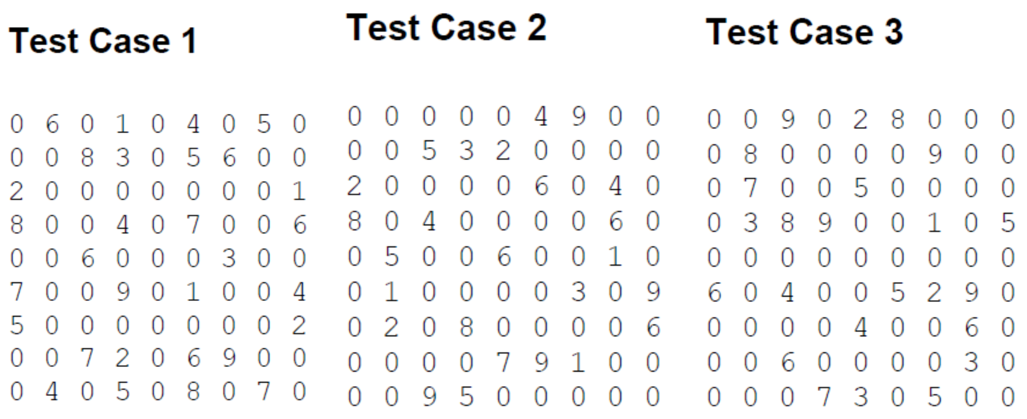
This algorithm requires a constant space. Apart from the function variable no additional space is dynamically allocated. The intermediate stages of sudoku puzzle is changed in place.Hence this algorithm is space efficient.

4.2.2 Space requirement analysis of backtrack approach

Every new state of the sudoku puzzle creates a new object for which the minQueue is created. The object lifetime is until the that particular state of sudoku either gets rejected or accepted.Hence at any intermediate step there are many objects in heap along with their queue data structure. Hence this is a space inefficient algorithm.

4.3 Test Cases

Both the algorithms were tested for several test cases.However for reporting purposes, three test cases are listed. The analysis results observed for these three test cases are discussed below.



Test Case 1

```
0 6 0 1 0 4 0 5 0
0 0 8 3 0 5 6 0 0
2 0 0 0 0 0 0 0 1
8 0 0 4 0 7 0 0 6
0 0 6 0 0 0 3 0 0
7 0 0 9 0 1 0 0 4
5 0 0 0 0 0 0 0 2
0 0 7 2 0 6 9 0 0
0 4 0 5 0 8 0 7 0
```

Test Case 2

```
0 0 0 0 0 4 9 0 0
0 0 5 3 2 0 0 0 0
2 0 0 0 0 6 0 4 0
8 0 4 0 0 0 0 6 0
0 5 0 0 6 0 0 1 0
0 1 0 0 0 0 3 0 9
0 2 0 8 0 0 0 0 6
0 0 0 0 7 9 1 0 0
0 0 9 5 0 0 0 0 0
```

Test Case 3

```
0 0 9 0 2 8 0 0 0
0 8 0 0 0 0 9 0 0
0 7 0 0 5 0 0 0 0
0 3 8 9 0 0 1 0 5
0 0 0 0 0 0 0 0 0
6 0 4 0 0 5 2 9 0
0 0 0 0 4 0 0 6 0
0 0 6 0 0 0 0 3 0
0 0 0 7 3 0 5 0 0
```

Figure 8: Test Cases

The below table shows the number of backtracks that occurs when run using both the algorithms. In order to calculate the number of backtracks a global counter is created and incremented when ever the backtrack function returns false or null in the recursion call.

Test Case	No.Clues	No.RecursiveCalls	No.Backtracks in Algo1	No.Backtracks in Algo2
1	29	1589	1538	987
2	25	21666	21610	12643
3	24	234290	234201	95681

From the above results, we observe that the number of backtracks for the naive approach is more or less the same as the number of recursive calls done. However, the no of backtracks for the algorithm 2 is much lesser than the first algorithm. One observation on the difference in the backtrack calls is that the difference remains closer when the number of backtrack calls are lesser. However, deriving a mathematical relation for the difference in backtrack is difficult.

5 Other approaches to solving sudoku problems

- ***Constraint Problem*:**[5] This method tries to solve the problem using constraint formulation and building up modeling techniques to solve the problem without using search and hence differs from back tracking. The paper[4] discuss about various ways in which this can be achieved.
- ***Boltsmann Machine:*** The algorithm is based on the artificial neural network [ANN] problem. There are relatively complex and are found to solve the harder problems inefficiently. This dissertation discusses the implementation of the boltsmann machine in detail.[boltsmann]
- Several other approaches used in solving sudoku includes exact cover, stochastic approaches, etc.

6 Conclusion

In this project, two backtracking algorithms for solving sudoku were implemented and analyzed. The first approach is a naive approach where it tries to randomly pick an unfilled cell and apply possible value until final solution is reached.The second approach is a smart backtracking approach, where the cell under consideration is extracted from a MinQueue which is ordered based on the length of possibilities of the cell.
Several analysis were carried out on these two algorithms.

- The first observation is that these algorithms behave almost the same in terms of number of backtracks if the length of possibility list is the same for all unfilled cells. However, the smarter backtracking algorithm works efficiently when the length of the possibilities of each cell vary.

- The second observation is that if the difference between the length of possibilities is large, then the number of backtracks required by smarter algorithm is much lower than the naive algorithm with an additional overhead of space requirement in maintaining MinQueue for all possible intermediate states.

7 References

[1]Kanakia, Anshul, and John Klinger. "Methods for Solving Sudoku Puzzles" 10.

[2]Kovacs, Tim . "Artificial Intelligence through Search: Solving Sudoku Puzzles"

[3]Zelenski, J. "Exhaustive recursion and backtracking In" 01 002 2008: 12.

[4]BERGGREN, PATRIK ,and DAVID NILSSON. "A study of Sudoku solving algorithms" 79.

[5]Simonis, Helmut . "Sudoku as a Constraint Problem" 15.

8 Appendix

8.1 Naive Backtracking code

```
/* SudokuNaive.cpp : This implements the naive backtracking algorithm
*/



#include "stdafx.h"
#include "time.h"
#include <stdio.h>

//Global variable to track the no of backtracks.
int Ucount = 0;



// This defines empty cells
#define UNDEFINED 0

// N is the size of the sudoku grid
#define N 9

// This finds a free cell
bool FindFreeLocation(int grid[N][N], int &row, int &col);

// This algorithm applies the row constraint, column constraint,
//and Box constraint and checks for any available possibilty
bool isSafe(int grid[N][N], int row, int col, int num);

/*main funciton*/
bool solveSudokuNaive(int grid[N][N])
{
    int row, col;

    // if the grid is full return
    if (!FindFreeLocation(grid, row, col))
```

```
        return true; // success!

    //check for possibilities
    for (int num = 1; num <= 9; num++)
    {
        // if available
        if (isSafe(grid, row, col, num))
        {
            // assign value
            grid[row][col] = num;

            // return if solution is accepted.
            if (SolveSudoku(grid))
                return true;

            // if not try with new possibilty
            grid[row][col] = UNDEFINED;
        }
    }
    //Increments backtrack calls.
    Ucount++
    return false; // this triggers backtracking
}

/* checks if there are any free location */
bool FindFreeLocation(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;
}

/* checks for row constraint */
bool rowConstraint(int grid[N][N], int row, int num)
```

```
{
    for (int col = 0; col < N; col++)
        if (grid[row][col] == num)
            return true;
    return false;
}

/* checks for colconstraint */
bool colConstraint(int grid[N][N], int col, int num)
{
    for (int row = 0; row < N; row++)
        if (grid[row][col] == num)
            return true;
    return false;
}

/* checks for box constraint */
bool boxConstraint(int grid[N][N],int boxStartRow,int boxStartCol,int nu
{
    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;
}

/* checks alll constraint */
bool isSafe(int grid[N][N], int row, int col, int num)
{
    /* Check if 'num' is not already placed in current row,
       current column and current 3x3 box */
    return !rowConstraint(grid, row, num) &&
           !colConstraint(grid, col, num) &&
           !boxConstraint(grid, row - row%3 , col - col%3, num);
}
```

```
/* print function */
void printGrid(int grid[N][N])
{
    for (int row = 0; row < N; row++)
    {
       for (int col = 0; col < N; col++)
             printf("%2d", grid[row][col]);
        printf("\n");
    }
}

/* Test function*/

int _tmain(int argc, _TCHAR* argv[])
{

   clock_t begin, end;
   double time_spent;
   begin = clock();
        // 0 means unassigned cells
    int grid[N][N] = {{1, 2, 8, 0, 6, 0, 3, 0, 9},
                      {3, 0, 4, 2, 1, 9, 8, 5, 6},
                      {9, 0, 6, 0, 3, 0, 0, 4, 2},
                      {0, 6, 5, 0, 0, 8, 0, 2, 3},
                      {0, 4, 0, 0, 7, 0, 0, 0, 1},
                      {0, 0, 3, 0, 0, 2, 9, 0, 0},
                      {5, 0, 0, 3, 8, 0, 0, 1, 4},
                      {0, 8, 0, 0, 2, 0, 6, 0, 5},
                      {0, 3, 0, 7, 0, 4, 2, 9, 8}};


         printGrid(grid);
         printf( "\n");
    if (solveSudokuNaive(grid) == true)
          cout << grid;
    else
```

```
            cout << "No solution";
          end = clock();
    time_spent = (double)(end - begin) / CLOCKS_PER_SEC;
    printf("No of Back Tracks = %d\n", Ucount);
    printf(" \nTime Taken  :%f  seconds\n", time_spent);
          return 0;
}
```

8.2 Smart Backtracking code

```
// sudokuSolversmart.cpp : This implements the m
//smartbacktracking algorith

#include "stdafx.h"
#include "minQueue.h"
#include "time.h"

/*global variable to count backtracks*/
int Icount = 0;

#define L 9
#define UNDEFINED 0
bool solve=true;

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

/*class definition of sudoku puzzle*/

class solveSudoku
```

```
{
public:
        solveSudoku();
        solveSudoku(bool insert);
        solveSudoku(int grid[L][L]);
        ~solveSudoku();
   List* possibilities(int row, int col);
  void getBox(int row, int col, int box[L]);
  void GetChildQueue(void);

  bool reject(void);
  bool accept(void);
  bool rowConstriant(int row, int num);
  bool colConstraint(int col, int num);
  bool boxConstraint(int boxStartRow, int boxStartCol, int num);
  bool isSafe(int row, int col, int num);
  void printGrid(void);

  solveSudoku* getNext();
  MinQueue* childQ;

private:

        int SudokuGrid[L][L];

};

/*Constructor*/
solveSudoku::solveSudoku()
{
        /*SudokuGrid = new int*[L];
        for (int i = 0; i < L; ++i)
        {
                SudokuGrid[i] = new int[L];
        }*/
}
```

```
/*Constructor*/
solveSudoku::solveSudoku(bool insert)
{
  /*SudokuGrid = new int*[L];
        for (int i = 0; i < L; ++i)
        {
                SudokuGrid[i] = new int[L];
        }*/
  if(insert)
  {
   for(int i = 0; i < L; i++)
   {
        for(int j = 0; j < L; j++)
        {
                SudokuGrid[i][j] = test[i][j];
        }
    }
  }
}

/*Constructor*/
solveSudoku::solveSudoku(int grid[L][L])
{
        /*SudokuGrid = new int*[L];
        for (int i = 0; i < L; ++i)
        {
                SudokuGrid[i] = new int[L];
        }*/
   for( int i = 0; i < L; i++)
   {
        for(int j = 0; j < L; j++)
        {
                SudokuGrid[i][j] = grid[i][j];
        }
   }
}

/*Destructor*/
solveSudoku::~solveSudoku()
{
        /*for(int i = 0; i < L; i++)
        {
                if(SudokuGrid[i])
                        delete SudokuGrid[i];
        }
                if(SudokuGrid)
                        delete SudokuGrid;
                SudokuGrid = nullptr;*/
        //delete this;
}

/*Returns the next possibilty for a particular state of sudoku*/
solveSudoku* solveSudoku::getNext()
{
        int choice;
        CellSet* cell = childQ->next();
        if(cell == nullptr)
                return nullptr;
        choice  = cell->Set->getChoice();
        solveSudoku* newS  = new solveSudoku(SudokuGrid);
        if(choice != 0)
        {
                newS->SudokuGrid[cell->i][cell->j] = choice;
        }
        if(cell->Set->listLength() > 0)
        {
                childQ->push(cell);
        }
        return newS; //add condition for choice==0
}
```

```
/*Implementation of Naked Single*/
List* solveSudoku::possibilities(int row, int col)
{
        List* retList =  new List;
        int cell[L];
        for(int i  = 0; i < L; i++)
        {
                if(SudokuGrid[row][i] != UNDEFINED)
                {
                        retList->remove(SudokuGrid[row][i]);
                }
                else if(SudokuGrid[i][col] != UNDEFINED)
                {
                        retList->remove(SudokuGrid[i][col]);
                }
        }

        getBox(row,col,cell);
        for(int i = 0; i< L; i++)
        {
                if( cell[i] != UNDEFINED)
                {
                        retList->remove(cell[i]);
                }
        }
        return retList;
}


/*Implementation of getchildqueue*/
void solveSudoku::GetChildQueue()
{
        int size = L*L;
        childQ = new MinQueue(size);
        for(int i = 0; i < L; i++)
        {
                for(int j = 0; j < L; j++)
                {
                        if(SudokuGrid[i][j] == UNDEFINED)
                        {
                                CellSet* newSet = new CellSet;
                                newSet->i = i;
                                newSet->j = j;
                                newSet->left = nullptr;
                                newSet->right = nullptr;
                                newSet->Set = possibilities(i,j);
                                childQ->push(newSet);
                        }
                }
        }
}

/*returns all box values of a cell*/
void solveSudoku::getBox(int row, int col, int box[N])
{
        int boxStartRow =row - row%3;
        int boxStartCol = col - col%3;

         for (int i= 0; i < 3; i++)
         {
        for (int j = 0; j < 3; j++)
                {
            box[i * 3 + j] = SudokuGrid[i+boxStartRow][j+boxStartCol];
                }
         }
}

/* checks row constraint. */
bool solveSudoku::rowConstriant( int row, int num)
```

```
{
    for (int col = 0; col < L; col++)
        if (SudokuGrid[row][col] == num)
            return true;
    return false;
}

/* checks column constriant */
bool solveSudoku::rowConstraint(int col, int num)
{
    for (int row = 0; row < L; row++)
        if (SudokuGrid[row][col] == num)
            return true;
    return false;
}

/* checks box constraint */
bool solveSudoku::boxConstraint(int boxStartRow,int boxStartCol,int num
{
    for (int row = 0; row < 3; row++)
        for (int col = 0; col < 3; col++)
            if (SudokuGrid[row+boxStartRow][col+boxStartCol] == num)
                return true;
    return false;
}

/* checks all constriants. */
bool solveSudoku::isSafe(int row, int col, int num)
{
    /* Check if 'num' is not already placed in current row,
       current column and current 3x3 box */
    return !rowConstraint(row, num) &&
           !colConstraint(col, num) &&
           !boxConstraint(row - row%3 , col - col%3, num);
}
```

```
/*Implementation of reject function*/
bool solveSudoku::reject(void)
{
        bool res;
        for( int i = 0 ; i < L; i++)
        {
                for(int j = 0; j <L; j++)
                {
                        int num = 0;
                        if(SudokuGrid[i][j] != UNDEFINED)
                        {
                                num = SudokuGrid[i][j];
                                SudokuGrid[i][j] = UNDEFINED;
                                res = isSafe(i,j,num);
                                SudokuGrid[i][j] = num;
                                if(!res)
                                        return true;
                        }
                }
        }
        return false;
}

/*Implementation of accept function*/
bool solveSudoku::accept(void)
{
  if(reject())
          return false;
  else
  {
          for(int i = 0; i < L; i++)
          {
                  for(int j = 0; j < L; j++)
                  {
                        if(SudokuGrid[i][j] == UNDEFINED)
                                return false;
                }
        }
  }
  return true;
}


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



/*Smart BackTracking Algorithm*/
solveSudoku* BackTrack(solveSudoku* S)
{
        if (S->reject())
        {
        //      cout << "Rejected" <<endl;
                if(S->childQ)
                        delete S->childQ;
                if(S)
                        delete S;
                S = nullptr;
                return nullptr;
        }
        else if(S->accept())
        {
                //cout << "Accepted" <<endl;
                return S;
        }
        solveSudoku* solution = nullptr;
    solveSudoku*  newS =  nullptr;
        S->GetChildQueue();
        newS = S->getNext();
        while( newS != nullptr && solution == nullptr)
        {
                solution = BackTrack(newS);
                newS = S->getNext();
                Icount++;
        }
        return solution;
}


/*Test function*/

int _tmain(int argc, _TCHAR* argv[])
{
   clock_t begin, end;
   double time_spent;
   begin = clock();
        solveSudoku S = new solveSudoku(solve);
        S.printGrid();
        solveSudoku* solution;
        solution = BackTrack(&S);
        if(solution != nullptr)
        {
        cout << "Solution Availalbe :\n"<< endl;
                solution->printGrid();
        }
        else
        {
                cout << "No Solution Available";
        }
    end = clock();
    time_spent = (double)(end - begin) / CLOCKS_PER_SEC;
    cout << "\n BackTrackCount : " << Icount <<endl;
    cout << " \nTime Taken:" << time_spent<< "seconds" <<endl;
    return 0;
}

/* This implements the minQueue datastructure used by the
    smartbacktrackingalgorithm*/

#include <cmath>
#include <climits>
#include <string>
#include <iostream>
using namespace std;

#define len 10

#define N 81 /*size of sudoku grid*/

/*class definition for list datastructure used inside the
cellset datastructure*/
class List
{
public:
        List();
        ~List();
        void remove(int n);
        int listLength(void);
        void setLength(int);
        int getChoice(void);

private:
        int list[10];

};

/*constructor for list*/
List::List()
{
        list[0] = 10;
        for(int i = 1; i < len; i++)
        {
                list[i] = 1;//1 indicates the presence of a number.
        }
}

/*Destructor for list*/
List::~List()
{
        //delete this;
}

/*Removes a possible value from the list*/
void List::remove(int n)
{
        if (n < 10 && list[n]!= 0)
        {
                list[n] = 0;
                list[0] = list[0] - 1;
        }
}

/*Returns the length of the list*/
 int List::listLength(void)
{
        return list[0];
}
```

```
/*sets the length of the new list*/
void List::setLength( int length)
{
        list[0] = length;
}

/*returns a new possibility*/
int List::getChoice(void)
{
        int choice = 0;
        if(list[0] == 0)
        {
                return 0;
        }
        for( int i = 1; i< len; i++)
        {
                if(list[i] != 0)
                {
                        choice = i;
                        remove(i);
                        break;
                }

        }
        return choice;
}

/*Structure of CellSet */
struct CellSet{
        int i;
        int j;
        List* Set;
        CellSet* left;
        CellSet* right;
};
```

```
/*Class definition of minQueue*/
class MinQueue
{
        public:
        CellSet** node;
        int queuesize;
        MinQueue(int n);
        ~MinQueue();
        int parent(int);
        int left(int);
        int right(int);
        void MinQueueify(int);
        void buildMinQueue(int);
        CellSet* heapMinimum();
        CellSet* next();
        int queueIncreaseKey(int, int);
        int push(CellSet *);
        void printMinQueue();
        private:
};


/*Constructor */
MinQueue::MinQueue( int n)
{
        int i;
        node = new CellSet*[N];
        queuesize = 0;
        for(i = 0; i < N; i++)
                {
                        /*node[i] = new CellSet;
                        node[i]->left = nullptr;
                        node[i]->right = nullptr;*/
                        node[i] = nullptr;
                }
}

/* Destructor */
MinQueue::~MinQueue()
{
        for( int i = 0; i < N * N; i++)
        {
                if(node[i])
                {
                        if(node[i]->Set)
                                delete node[i]->Set;
                        delete node[i];
                }
        }
        if(node)
                delete [] node;
        node = nullptr;
}



/*Returns the parent of a node*/
int MinQueue::parent(int i)
{
        return floor((i - 1)/2); /*adjusting for 0 index*/
}
/*Returns the left child of a node*/
int MinQueue::left(int i)
{
        return 2 * i + 1;
}
/*Returns the right child of a node*/
int MinQueue::right(int i)
{
        return 2 * i + 2;
}
```

```
/*Increases the key of a particular elememt */
int MinQueue::queueIncreaseKey(int i, int key)
{
        if( node[i] == nullptr)
                return 0;
        else if(node[i]->Set->listLength() > key)
                return 0;
        else
        {
                node[i]->Set->setLength(key);
                CellSet* temp;
                while(i > 0 && node[parent(i)]->Set->listLength() >
                   node[i]->Set->listLength())
                {
                        temp = node[parent(i)];
                        node[parent(i)] = node[i];
                        node[i] = temp;
                }
                return 1;
        }
}


/*Inserts a new node */
int MinQueue::push(CellSet* newnode)
{
        int key = newnode->Set->listLength();
        newnode->Set->setLength(INT_MIN);
        queuesize = queuesize + 1;
        node[queuesize - 1] = newnode; /*adjusting index*/
        return queueIncreaseKey(queuesize - 1,key);
}

/*Maintains the heap property*/
void MinQueue::MinQueueify(int i)
{
        int smallest;
        int l = left(i);
         int r = right(i);
        CellSet* temp;
        if( l < queuesize && (node[l]->Set->listLength() <
                    node[i]->Set->listLength()))
        smallest = l;
        else
        smallest = i;
        if( r < queuesize && (node[r]->Set->listLength() <
                node[smallest]->Set->listLength()))
        smallest = r;
        if( smallest != i)
        {
                temp = node[i];
                node[i] = node[smallest];
                node[smallest] = temp;
                MinQueueify(smallest);
        }
}

/*Extracts the first node of the queue*/
CellSet* MinQueue::next() /*Equivalent to Pop*/
{
        if( queuesize <= 0)
        {
        for( int i = 0; i < N * N; i++)
                {
                        if(node[i])
                        {
                                if(node[i]->Set)
                                {
                                        delete node[i]->Set;
                                        node[i]->Set =nullptr;
                                }
                                delete node[i];
                        node[i] = nullptr;
                    }
                }
                return nullptr;
        }
        CellSet* min = node[0];
        node[0] = node[queuesize - 1];/*index adjustment */
        queuesize = queuesize - 1;
        MinQueueify(0);
        return min;
}

/*Print utility*/
void MinQueue::printMinQueue()
{
        int i;
        for(i = 0; i < queuesize; i++)
        {
                cout << node[i]->i << "," << node[i]->j << "\n";
        }
}
```