Sudoku Validator

using Multi-threading in C++ and Java

CMPE 180-94 Fall 2015

Operating Systems

Professor: Hungwen Li

Students:

Pritesh Chandaliya 010121527

Uday Dungarwal 010096398

Date: 11/19/2015

Summary and Objective

The main aim of the project was to learn and implement one of the Operating Systems fundamental. Multithreading is one of the crucial functionality and concept present in our OS and is used in almost all the applications we play with in our daily routine.

We developed a snippet of C++ and Java code to validate whether the given Sudoku solution is correct or not. The code not only has the logic to check for the above but incorporates multi threading too. For instance, we have to check many sudoku solutions and if we keep on checking each sequentially then a lot of time is waste, to avoid it we have written our code in such a way that at a time 3 (we can reconfigure) sudoku can be validated simultaneously. This will not only save our time but also helps to explore CPU processing power. Through this assignment, we were able to understand concept and logic beyond multithreading better.

Introduction

Sudoku puzzles are turning out to be famous among the general population both expert and layman, everywhere throughout the world. The game has now ended up well known in numerous nations and many engineers have attempted to create significantly all the more intriguing puzzles. Today, the game is in each daily paper, and in numerous sites. In this assignment we introduce a Sudoku Validator that will check whether given info is substantial and legitimate. We have checked three potential outcomes that accept an answer of sudoku puzzle.

1. To check if all elements of a row. None of the elements in a row ought to be same.
2. To check if all elements of a column. None of the elements in a column ought to be same.
3. To check for a component, its 3x3 matrix elements additionally add to 45 precisely in addition to no duplicate of elements.

Problem Statement

A Sudoku puzzle utilizes a 9 x 9 lattice in which every column and row, and each of the nine 3 x 3 sub-matrices, must contain all the the digits 1 2 3 4 5 6 7 8 9. Figure exhibits a sample of a legitimate Sudoku puzzle. This undertaking comprises of outlining a multithreaded application that figures out if the answer for a Sudoku puzzle is legitimate. There are a few unique methods for multithreading this application. One recommended methodology is to make strings that check the accompanying criteria:

1. A string to watch that every column contains the digits 1 through 9
2. A string to watch that every row contains the digits 1 through 9
3. Nine strings to watch that each of the 3 * 3 subgrids contains the digits 1 through 9 This would bring about a sum of eleven separate strings for approving a Sudoku puzzle.

Then again, you are welcome to make considerably more strings for this project. For instance, as opposed to making one string that checks every one of the nine columns, you could make nine separate strings and have each of them check one check.

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

Fig. 1 Sample Sudoku Solution

Background

In this segment we begin clarifying about what is Sudoku. After it, an examination that we carried on the prior works about the point sudoku. Likewise, we talk further about an algorithm in addition to a portrayal on how the work is carried out and furthermore displayed.

Short about Sudoku

Sudoku is a logic-based puzzle that is played by numbers from 1 to 9. The Puzzle first appeared in newspapers in November 1892 in France and then Howard Garns an American architect presented it in its modern form [1,5]. There are already many journals, papers and essays that researched about Sudoku Solvers and most of them present different type of algorithms. Sudoku's popularity is based on several reasons. First of all it is fun and fascinating, and very easy to learn because of its simple rules. There are currently many different type of Sudoku puzzles, classic Sudoku that contains a 9X9 grid with given clues in various places, mini Sudoku that consists of a grid with 4X4 or 6X6 sizes. The other type of Sudoku is Mega Sudoku that contains a grid with 12X12 or 16X16 sizes [2]. In this text, the focus is mostly on the classic Sudoku, i.e. 9X9 grid. Furthermore, Sudoku has become so popular, compared to other games, all over the world because its rules are easy to understand and it can improve our brain and also it is fun.

The structure of the puzzle is very simple, especially the classic puzzle. This essay is mainly focused on classic puzzle of a 9X9 grid. There already exist a number of digits in the board that make the puzzle solvable. It means that some numbers are already placed in the Sudoku board before starting playing. The board consists of 81 cells, which is divided into nine 3X3 sub boards and each 3X3 sub board is called "box" or "region". The main concept of the game is to place numbers from 1 to 9 on a 9X9 board so that every row, column and box contains any numbers but once. This means that no number is repeated more than once. An example of this game is illustrated in Fig.1. Generally, the puzzle has a unique solution. There are certain techniques to solve the puzzle by hand and these rules can be implemented into a computer program. These techniques are presented in more details in 2.3.

Previous Research

We have noticed that there is a large volume of published studies describing Sudoku problems. Furthermore, several research have been made to solve Sudoku problems in a more 8 efficient way [4]. It has conclusively been shown that solving the puzzle, by using different algorithms, is definitely possible but most developers seek for optimizations techniques such as genetic algorithms, simulated annealing, etc.

Different authors have made relative works already. Nelishia Pillay gives a solution for solving Sudoku by combining human intuition and optimization [5]. This author has investigated the use of genetic programming to improve a space of programs combined of the heuristics moves. However, we seek a solution to solve Sudoku puzzle based on human strategies, which uses techniques such as: naked single method, hidden single method etc. J.F. Crook have also discussed about solving Sudoku and presented an algorithm on how to solve the puzzles of differing difficulty with pencil-and-paper algorithm. This method has not been implemented and therefore it is hard to discuss how the algorithm performs [3]. Tom Davis has done a research about "The Mathematics of Sudoku". Tom has described all techniques that people usually use to solve the puzzles but his major attempt is to describe these techniques from mathematical perspective. However, all the strategies he mentions are not required to solve the puzzle. For instance the easy puzzles can be solved using only one or two strategies [7].

Approach/Methodology

1. To check if all elements of a row. None of the elements in a row ought to be same.
2. To check if all elements of a column. None of the elements in a column ought to be same.
3. To check for a component, its 3x3 matrix elements additionally add to 45 precisely in addition to no duplicate of elements.

Finding and Analysis

This section starts with analysis and discussions about two mentioned algorithms. A comparison is carried out between two algorithms in order to find out which algorithm is more efficient. At the end of the section, there are discussions on difficulty level of the puzzles and time complexity.

Pencil-And-Paper Solver

There are several methods that are used by human players when playing Sudoku. However, it may be impossible to implement all these methods. It is found that the hidden single method or pair method are difficult to be applied in computer programming, since a human player has a better overview over the whole Sudoku board than the computer programming does. This is due to the fact that a human player is able to scan two rows or two columns in order to check whether a certain digit is allowed to be in an empty square in the box that is supposed to be filled up. Implementing the above task in computer programming causes significant time consumption. The methods that are used in this algorithm are the following: Unique missing candidate Naked single method Backtracking

1. Unique missing candidate

This method is useful when there is just only one empty square in a row, column or box. The digit that is missing can be placed in that empty square. A similar definition is that if eight of nine empty squares are filled in any row, column or box, then the digit that is missing can fill the only empty square. This method can be useful when most of the squares are filled, especially at the end of a solution. It can also be suitable when solving easy puzzle and this method is efficient to find solution in this case. In this algorithm, the method goes through all rows, columns and boxes separately. The method then checks if a single value has missed in any row, column or box and place the single digit in that specific square (see the appendix).

2. Naked single method

The second method that is used in the pencil-and-paper algorithm is the Naked single method. This method checks every empty square in the Sudoku board and finds the square that can only take one single digits and the missing digit then is assigned to that square. Note that once the squares are filled by naked single digits other naked singles will appear. This process is repeated until the method has found all empty squares with the needed corresponding one single value and complete the board [7]. This method is a useful method when a human player solves the game. However if the corresponding method is combined with the unique missing candidate method then both the methods can solve the puzzles both in easy and medium levels quickly and more efficiently.

3. Backtracking (guessing method)

The unique missing method and the naked single method are able to solve all puzzles with easy and medium level of difficulties. In order to solve puzzles with even more difficult levels such as hard and evil the backtracking method has been used to complete the algorithm. A human player solves the puzzle by using simple techniques. If the puzzle is not solvable by using the techniques the player then tries to fill the rest of the empty squares by guessing.

The backtracking method, which is similar to the human strategy (guessing), is used as a help method to the pencil-and-paper algorithm. In other words, if the puzzle cannot be filled when using the unique missing method and the naked single method, the backtracking method will take the puzzle and fill the rest of empty squares. Generally, the backtracking method find empty square and assign the lowest valid number in the square once the content of other squares in the same row, column and box are considered. However, if none of the numbers from 1 to 9 are valid in a certain square, the algorithm backtracks to the previous square, which was filled recently.

The above-mentioned methods are an appropriate combination to solve any Sudoku puzzles. The naked single method can find quickly single candidates to the empty squares that needed only one single value. Since the puzzle comes to its end solution the unique missing method can be used to fill rest of the puzzles. Finally, if either method fills the board the algorithm calls the backtracking method to fill the rest of the board.

Conclusion and Recommendation

We have successfully implemented Sudoku Validator using multi-threading in both C++ and Java. As such, the solution is trivial but considering addition of multi-threading makes it more suitable for today's world of fast computing and availability of high quality multi-processor systems.

Here, we have used three approaches to check the logic of Sudoku as mentioned with respect to row, column and grid. The logic written in the code, can be used simultaneously with multiple input grids or matrix of sudoku. This will help to verify more than one solved matrixes at a particular moment.

References

Felgenhauer, Bertram and Frazer Jarvis. “Enumerating possible Sudoku grids.” 20 June 2005. 06 April 2009.

Jones, S.K., P.A. Roach, and S. Perkins. “Construction of Heuristics for a Search Based Approach to Solving Sudoku.” In the Proceedings of the 27th SGAI International Conference on Innovative Techniques and Applications of Artificial Intelligence. pp. 3749. 2007.

Lewis, Rhyd. “Metaheuristics can solve sudoku puzzles.” Journal of Heuristics. Vol 13, (2007): 387401. 06 April 2009.

Longo, Frank. Green Belt Sudoku: Martial Arts Sudoku, Level 4: NotSoEasy. New York: Sterling Publishing Company, 2006.

Longo, Frank. Red Belt Sudoku: Martial Arts Sudoku, Level 8: Super Tough. New York: Sterling Publishing Company, 2006.

Longo, Frank. White Belt Sudoku: Martial Arts Sudoku, Level 1: Easy. New York: Sterling Publishing Company, 2006.

Lynce, I., J. Ouaknine. “Sudoku as a SAT problem.” In: Proceedings of the 9th Symposium on Artificial Intelligence and Mathematics, 2006.

Russell, Stuart and Peter Norvig. Artificial Intelligence: A Modern Approach. New Jersey: Prentice Hall, 2003.

Appendix

Source Code for JAVA:

```
import java.util.*;

public class Sudoku_Validator {

    boolean isValid(int i, int j, int grid[][])
    {
      // Check whether grid[i][j] is valid at the i's row
      for (int column = 0; column < 9; column++)
        if (column != j && grid[i] [column] == grid[i] [j])
          return false;

      // Check whether grid[i][j] is valid at the j's column
      for (int row = 0; row < 9; row++)
        if (row != i && grid[row] [j] == grid[i] [j])
          return false;

      // Check whether grid[i][j] is valid in the 3 by 3 box
      for (int row = (i / 3) * 3; row < (i / 3) * 3 + 3; row++)
        for (int col = (j / 3) * 3; col < (j / 3) * 3 + 3; col++)
          if (row != i && col != j && grid[row] [col] == grid[i] [j])
            return false;

      return true; // The current value at grid[i][j] is valid
    }

    /** Check whether the fixed cells are valid in the grid */
    boolean isValid(int grid[][]) {
      // Check for duplicate numbers
      for (int i = 0; i < 9; i++)
        for (int j = 0; j < 9; j++)
          if (grid[i][j] != 0)
            if (!isValid(i, j, grid))
              return false;

      // Check whether numbers are in the range
      for (int i = 0; i < 9; i++)
        for (int j = 0; j < 9; j++)
          if ((grid[i][j] < 0) || (grid[i][j] > 9))
            return false;

      return true; // The fixed cells are valid
    }

    /** Print the values in the grid */
    void printGrid(int grid[][])
    {
      for (int i = 0; i < 9; i++)
      {
        for (int j = 0; j < 9; j++);
//        cout << grid[i] [j] << " ";
//      cout << endl;
      }
    }

}
```

mainClass.java:

```
public class mainClass{

    static Sudoku_Validator s = new Sudoku_Validator();

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

    static int grid2[][] =
         {{8,3,2,4,1,6,9,2,7},
          {2,9,6,8,5,7,4,3,1},
          {4,1,7,2,9,3,6,5,8},
          {5,6,9,1,3,4,7,8,2},
          {1,2,3,6,7,8,5,4,9},
          {7,4,8,5,2,9,1,6,3},
          {6,5,2,7,8,1,3,9,4},
          {9,8,1,3,4,5,2,7,6},
          {3,7,4,9,6,2,8,1,5}};


    public static void main(String[] args) {

    Thread a = new Thread(new Runnable() {
        public void run() {
            if(s.isValid(grid)) System.out.println("1st grid Valid");
            else System.out.println("1st grid Invalid");
        }
    });

    Thread b = new Thread(new Runnable() {
        public void run() {
            if(s.isValid(grid2)) System.out.println("2nd grid Valid");
            else System.out.println("2nd grid Invalid");
        }
    });

    Thread b = new Thread(new Runnable() {
        public void run() {
            if(s.isValid(grid2)) System.out.println("2nd grid Valid");
            else System.out.println("2nd grid Invalid");
        }
    });
    a.start();
    b.start();
}
}
```

Source Code for C++:

---

main.cpp :

```
#include <iostream>
#include <fstream>
#include <pthread.h>

#define NUM_THREADS 3
using namespace std;

void readpuzzle ( int grid[][9]);
bool isValid(int i, int j, int grid[][9]);
bool isValid(int grid[][9]);
void printGrid(int grid[] [9]);

int main()
{

    pthread_t threads[NUM_THREADS];

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

    bool rc;
    for(int i=0; i < NUM_THREADS; i++ )
    {
        cout << "main() : creating thread, " << i << endl;
        rc = pthread_create(&threads[i], NULL,
                            isValid(grid), (void *)i);
    }

    //int grid [9][9];


    //readpuzzle (grid);
    if (!rc)
```

```
    cout << "Invalid input" << endl;
    else if (rc)
    printGrid(grid);
    else
    cout << "No solution" << endl;

    pthread_exit(NULL);
    return EXIT_SUCCESS;
}

/* void readpuzzle (int grid[][9])
 {
 {
 // Create a Scanner
 cout << "Enter a Sudoku puzzle:" << endl;
 for (int i = 0; i < 9; i++)
    for (int j = 0; j < 9; j++)
 cin >> grid[i] [j];
 }
 }
 */

bool isValid(int i, int j, int grid[] [9])
{
    // Check whether grid[i][j] is valid at the i's row
    for (int column = 0; column < 9; column++)
    if (column != j && grid[i] [column] == grid[i] [j])
    return false;

    // Check whether grid[i][j] is valid at the j's column
    for (int row = 0; row < 9; row++)
    if (row != i && grid[row] [j] == grid[i] [j])
    return false;

    // Check whether grid[i][j] is valid in the 3 by 3 box
    for (int row = (i / 3) * 3; row < (i / 3) * 3 + 3; row++)
    for (int col = (j / 3) * 3; col < (j / 3) * 3 + 3; col++)
    if (row != i && col != j && grid[row] [col] == grid[i] [j])
    return false;

    return true; // The current value at grid[i][j] is valid
}

/** Check whether the fixed cells are valid in the grid */
bool isValid(int grid[][9]) {
    // Check for duplicate numbers
    for (int i = 0; i < 9; i++)
    for (int j = 0; j < 9; j++)
    if (grid[i][j] != 0)
    if (!isValid(i, j, grid))
          return false;

    // Check whether numbers are in the range
    for (int i = 0; i < 9; i++)
    for (int j = 0; j < 9; j++)
    if ((grid[i][j] < 0) || (grid[i][j] > 9))
    return false;

    return true; // The fixed cells are valid
}
/** Print the values in the grid */
void printGrid(int grid[] [9])
{
    for (int i = 0; i < 9; i++)
    {
        for (int j = 0; j < 9; j++)
        cout << grid[i] [j] << " ";
        cout << endl;
    }
}
```

Screenshots:

```
#include <iostream>
#include <fstream>
using namespace std;

void readpuzzle ( int grid[][9]);
bool isValid(int i, int j, int grid[][9]);
bool isValid(int grid[][9]);
void printGrid(int grid[] [9]);

int main()
{
//int grid [9][9];

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

//readpuzzle (grid);
  if (!isValid(grid))
    cout << "Invalid input" << endl;
  else if (isValid(grid))
```

Get URL

options | compilation | execution

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

Exit code: 0 (normal program termination)
```

Code prints the sudoku grid if it is correct and validate all the three cases

```
#include <iostream>
#include <fstream>
using namespace std;

void readpuzzle ( int grid[][9]);
bool isValid(int i, int j, int grid[][9]);
bool isValid(int grid[][9]);
void printGrid(int grid[] [9]);

int main()
{
//int grid [9][9];

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

//readpuzzle (grid);
  if (!isValid(grid))
    cout << "Invalid input" << endl;
  else if (isValid(grid))
```

Get URL

options | compilation | execution

```
Invalid input

Exit code: 0 (normal program termination)
```

If the input is invalid, then corresponding message is sent as the output