Sudoku

Adil Arhzane, Erik Englund & Oskar Persson
Program Design and Data Structures 2013/2014

Introduction

What is a Sudoku?

A sudoku is a numeric puzzle made out of a nine-by-nine grid where each row, column and three-by-three-subsection contains unique numbers from 1 to 9. The idea is that you start with a puzzle where some numbers are missing and the goal is to find out which numbers they are. An example of a typical non-solved sudoku is shown below.

figure 1

In this project, we have focused on creating a data-program intended to solve a given sudoku-puzzle.

Implementation

Algorithms

When we started out the project, the first thing we did was figuring out how we would solve a sudoku by hand, which operations we would carry out and in which order. We figured that the first thing we would do was to look whether the same number occurred two times in the same three by three horizontal or vertical column. This would mean that the third occurrence of this number should be inserted into the column that the previous two did not already cover.

In the puzzle shown above (fig. 1) we would for example find that number 5 is occurring in the first and second vertical columns (counting from left to right). Since they are located in the fourth and seventh three-by-three square (counting from top left to bottom right), the 5 in the third vertical column should be located in the first three-by-three square.

The next step would be to check the horizontal columns related to the first three-by-three square (from here on in called square) to see whether there are any 5's that cuts away any of the three possible positions in the first square. We see that there is only one 5 located in the three top-most horizontal columns and therefore we have two possible spots for the 5 in the first

square. From here we cannot proceed any further, and we have to look at other columns and numbers too see if we can find a position where we only have one option.

In the example we can see that there is a 1 occurring in the second and third vertical columns, and a 1 in the sixth horizontal column (counting top to bottom). This means there is only one option for the 1 in the first vertical column, and we can fill that in and continue with other numbers and positions.

However, if you start out with a puzzle where you can make no conclusions by using the algorithm above, you have to approach the puzzle from a slightly different angle. By looking at which square that contains the most amount of known numbers, you can produce all possible permutations of that single square and see which one of them that works together with the rest of the puzzle (ie. no number occurs more than once in the same square, row or column). By doing this, you might have to try and solve the puzzle with one permutation at a time to see which one of them "works out". This might be relatively time consuming given that in worst case your last permutation is the right one. On the other hand, if you're lucky you will get it right at the first try.

This is the operations we decided to try and implement into our program code for solving sudokus.

Tree Structure

In our code, we chose to represent possible solutions to a sudoku puzzle in a tree structure, a STree, consisting of a root that is the puzzle handled at the moment and a list of STrees that holds all potential next steps to the puzzle.
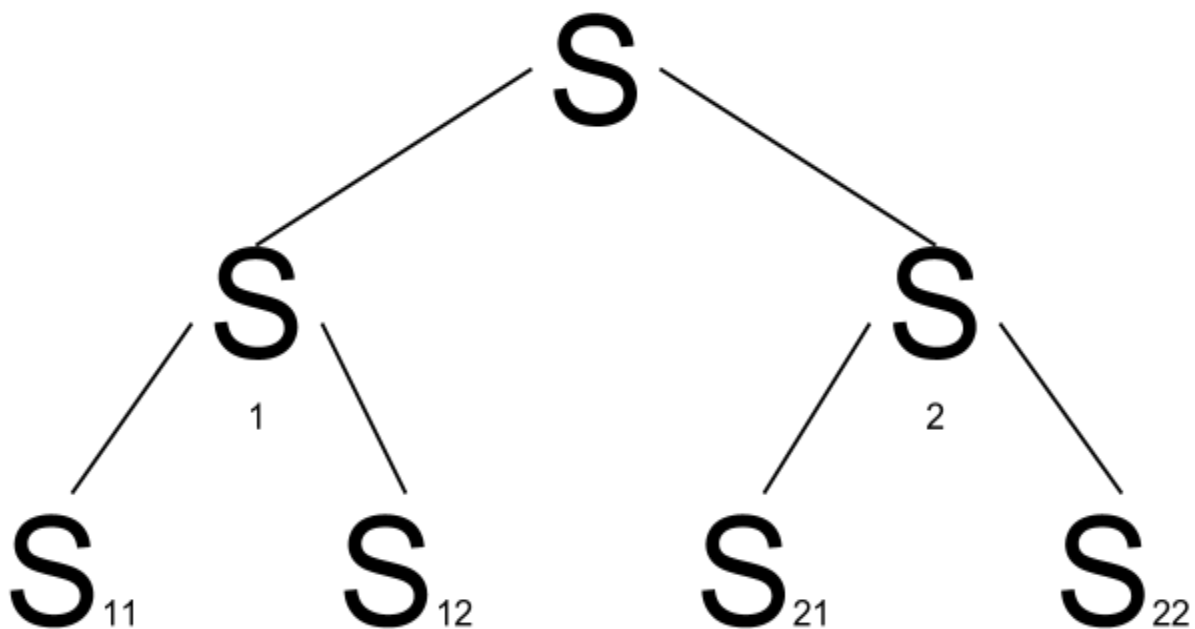

figure 2

Datatypes

For this project, we have chosen to represent a sudoku as a datatype in SML consisting of three lists of integer lists. One of the lists represents the horizontal rows, another the vertical columns and the third the 9 different three-by-three squares.

Functions

Ascii

The function ascii is used to easier get a better overview of the sudoku. The function takes a puzzle and prints this in the structure of a sudoku, a nine-by-nine grid with three-by-three squares as subsections.

SumOfElements

sumOfElements is used to receive the total sum of a list of integers, this function is mostly used to find the total value of a three-by-three square. Since the sum of all integers 1-9 is 45, we know that each square must have a total value of 45 in order to be stated as solved.

OneUnknown

If there is a single zero (0) in a list, oneUnknown will replace this element with the missing integer (1-9).

NotInSquare

Lists the numbers from 1-9 that is missing from a list, so that we know what numbers we have, and which are missing and need to be filled in.

Permute

Gives you all permutations of the integer list you put in, a permutation is a combination of a list of numbers.

SquareWithLeastUnknowns

This function is used to find the first square in the sudoku with the least unknowns in it.

PossibleSolutionsForSquare

Is used to receive possible solutions for squares (pretty much as the name says) by using the permutations.

Traversal

Traversal is the main function of the program. By using the previous mentioned functions, it will traverse through a STree containing all possible permutations in each step towards the solution.

Test cases

not only the solve function, but also the auxilliary functions

Conclusion

Summary

Discussion

When we started this project our goal was to create a program that would be able to solve any sudoku in reasonable time, whether the sudoku was considered easy or extremely hard. Naturally we chose to start at the bottom with simple lists to get a working algorithm that would not take too long. When we had written the code and it worked, we encountered a slight problem. The algorithm took too long time to solve these sudokus. A sudoku considered easy, medium or hard took about 30-40 seconds to solve, and a sudoku considered one of the world's hardest took us 6 hours.

We continued to work with vectors in our code instead of lists, figuring we might be able to solve sudokus faster, since elements in a vector can be accessed in constant time regardless of where in the vector it is located. The downside is that if you need to replace an element in a vector you have to rebuild the whole vector.

When we had completed the work with vectors, we saw that the result was not what we had set out for. The time needed to solve the same puzzles as we solved with lists was almost three times longer. A possible cause of this problem might be that we build new vectors more often than we access elements (ie the code writes to more vectors than it reads), which causes the code to run slower with vectors than with lists.

After a meeting with our supervisor we realized that our algorithm and implementation would not work faster just with vectors, but that we had to use ref vectors, since ref vectors allows you to both read and update in constant time. This would, in theory, increase the algorithms efficiency and the code would run much faster and smoother.

The implementation of ref vectors were succeeded and we ran our test cases to see if our hypothesis were correct. Even if we would have loved the program to run faster it did not. ref vectors took even longer than normal vectors, we had a problem. After implementing three different versions: int list list, int vector vector, int ref vector vector and finally int ref vector ref

vector we saw that the first was also the fastest. The reason that the list implementation was the fastest was that we implemented the functions with lists in mind. If we had gone with another data structure from the start and built the functions around that, then those would have been faster.

Possible Improvements

The main problem with solving sudokus is the time taken to solve harder puzzles. With our algorithm, using lists, it takes approximately 6 hours to solve the hardest puzzle we could find, while a "standard" puzzle requires between 20-40 seconds depending on difficulty. When using vectors, it took even longer to solve the same puzzles. The most probable reason to this is that we started out writing our code using lists, and then we reworked the same code using vectors. If we had started over from the beginning, implementing the algorithm with vectors we might have reached a different result.

References