Non-Deterministic Sudoku Solver

CSC 335

By: Binod Timalsina & Khadeeja Din

Contents

1. **Introduction**
   - 1.1 Design Problem Description
   - 1.2 Design Outline
   - 1.3 Development Environment
2. **Design Background**
   - 2.1 Problem reference from SICP
3. **Explanation of Technology Used**
   - 3.1 Amb & Call/cc
   - 3.2 Streams
4. **Outline Solution**
   - 4.1 Description of the used Algorithm
   - 4.2 Details about the modified use of Algorithm and its result
5. **Detailed Design**
   - 5.1 Filling Sudoku Puzzle
   - 5.2 Swapping Coordinates
   - 5.3 Calculating Error
   - 5.4 Solving Sudoku
6. **Sample output**
   - 6.1 Solution for an easy 4 x 4 puzzle
7. **Future Development**
8. **References**

1. Introduction

			4
	2	1	
	1	4	
			1

A 4 X 4 Sudoku problem

1	3	2	**4**
4	**2**	**1**	3
3	**1**	**4**	2
2	4	3	**1**

Solution achieved for the Sudoku

1.1 Design Problem Specification

Our project was to design a non-deterministic solver for the Sudoku puzzle. Sudoku is a logic based, combinatorial number-placement puzzle of size nxn. (Wikipedia) The puzzle is composed of a nxn grid containing nxn cells. The objective of the game is to organize the numbers 1-n in such a way that all the n rows, n columns, and n sub-grids have 1-n distinct numbers each. Sudoku is an NP-complete problem, so there is not a polynomial time algorithm for all possible problem instances. The logical algorithms that do solve sudoku, are only guaranteed to solve the puzzles specifically constructed for them following the logical rules alone.

Therefore, the purely logic-based algorithms are not always the most suitable for solving sudoku. In our project, we have picked the non deterministic method for addressing the sudoku problem. We picked this design approach to explore how efficient or inefficient a non-deterministic Sudoku solver can be. In our solution, we have found the latter to be true. The solver will give a result for an easy puzzle of size 4, but for a standard 9*9 puzzle the solver continues running infinitely. However, occasionally we may find a correct solution to the standard puzzle, as our solver is theoretically correct.

1.2 Design Outline

The solver basically follows a backtracking approach, where we start by randomly filling the empty spaces such that each subgrid contains of numbers 1-n without repetition. At this point the puzzle meets one condition of having 1-n distinct numbers within a subgrid. However, we still need to meet the same condition for every row and column in the grid. Our algorithm will randomly swap cells within a subgrid until all conditions are met, i.e. a correct solution for the puzzle is achieved. After every swap, the solver keeps track of the total error for columns and rows. If the total error is found to be reduced, the algorithm will continue forward, or else it will backtrack to the most recent choice point. For more detailed design description please look at section 5 of the report.

1.3 Development Environment

DrRacket Scheme implemented in Pretty Big.

2. Design Background

2.1 Problem Reference from SICP

The idea used in our design is taken from Section 4.3.3 of Abelson and Sussman, “The evaluation of an ordinary Scheme expression may return a value, may never terminate, or may signal an error. In nondeterministic Scheme the evaluation of an expression may in addition result in the discovery of a dead end, in which case evaluation must backtrack to a previous choice point.” (646) We designed our solution for the sudoku problem following this key idea. Where the logical sudoku solvers either solve the problem, or may or may not terminate, the non-deterministic solver can land into a dead end. By the end of our program design, we did discovered that for a wide range of puzzles, our solver does reach a deadend.

3. Explanation of Technology Used

3.1 AMB

We are using the amb evaluator given in *Structure and Interpretation of Computer Programs* to nondeterministically obtain a solution for sudoku. Amb is a special form that allows nondeterministic computation. The amb form automatically chooses a possible value from a list of options and keeps track of the choice while continuing the execution. If later the program encounters a failure the amb form backtracks to the latest pending amb form, and restarts the computation from here. (Amb Special Form) Abelson and Sussman describe the systematic search carried by their amb evaluator as, “When the evaluator encounters an application of amb, it initially selects the first alternative. This selection may itself lead to a further choice. The evaluator will always initially choose the first alternative at each choice

point. If a choice results in a failure, then the evaluator automagically backtracks to the most recent choice point and tries the next alternative. If it runs out of alternatives at any choice point, the evaluator will back up to the previous choice point and resume from there." (630) In our program, we use amb to return number choices to be swapped from a stream of numbers. So, basically the amb will drive the execution by repeatedly picking cells and swapping their numbers until the problem is solved. Abelson and Sussman define the amb execution word as the following, "the execution procedures in the amb evaluator take three arguments: the environment, and two procedures called continuation procedure. The evaluation of an expression will finish by calling one of these two continuations: If the evaluation results in a value, the success continuation is called with that value; if the evaluation results in the discovery of a dead end, the failure continuation is called. Constructing and calling appropriate continuations is the mechanism by which the nondeterministic evaluator implements backtracking." (646) The success continuation in the evaluator is simply continuing amb on the remaining items, i.e. choosing the first value from the remaining sequence. On the other hand, the failure continuation is calling the amb expression without any choices. As Abelson and Sussman write, "Amb with no choices--the expression (amb)--is an expression with no acceptable values." (628) Therefore, when (amb) is executed no value is produced and the computation is aborted.

3.2 Stream

Stream is a type of data structure. It is a delayed list i.e. it is only evaluated partially each time we call it. According to Abelson and Sussman (3.5), "The basic idea [of stream] is to arrange to construct a stream only partially, and to pass the partial construction to the program that consumes the stream. If the consumer attempts to access a part of the stream that has not yet been constructed, the stream will automatically construct just enough more of itself to produce the required part,

thus preserving the illusion that the entire stream exists. In other words, although we will write programs as if we were processing complete sequences, we design our stream implementation to automatically and transparently interleave the construction of the stream with its use." So stream is partially evaluated list that evaluates the unevaluated part only when program calls for it. For this project we used an infinite stream that randomly gives a value from a list each time we call for it. This is then passed as input for Amb function. So combining these we can do infinite Amb evaluations which is necessary for our problem.

The Amb function **an-element-of** calls first item of stream for calculation then when the calculation fails it discards whole calculation for that number and calls next item from stream. The stream calculates a random output when the Amb function seeks next option. This way we can have infinite loop without doing unnecessary calculation of calculating a large number of options.

4. Outline Solution

4.1 Description of the used Algorithm

The algorithm for solving this Sudoku problem is (Lewis):

1. Fill all the empty spaces of sudoku by assigning value such that each sub grid has 1 to n numbers. Consider these cells movable cells.
2. randomly select a subgrid and randomly select two movable grids from that subgrid and swap their values.
3. Repeat swap until the problem is solved

4.2 Details about the modified use of Algorithm and its result

We modified this problem so that we update sudoku only if the swap decreases the total error in Sudoku. This caused some calculation to remain stuck in the infinite loop. It seems that we hit a dead end in some calculations, and the function keeps swapping any way. Also even if there is no valid swap, Amb function will keep running since it has infinite options. According to Abelson and Sussman the amb evaluator will keep on trying new choices until either the evaluation succeeds or runs out of choices. The use of infinite stream ensures that the amb never runs out of choices. This is why for bigger and complex puzzles, our solver continues running infinitely. Hence, the likelihood of hitting a dead end is higher for sudoku's bigger than 4 X 4. Even for a complex 4 x 4 puzzle we had to run the code multiple times to obtain a solution.

We can observe that our algorithm is less likely to solve bigger sudoku puzzles. This is because our swapping is not completely random. To avoid this problem, the swapping should have been completely random regardless of whether it decreases or increases the error. The free swap may also not solve every sudoku instance, and would take longer to solve even if it did; however, it would never hit a dead end. The free swapping solver would take longer to solve a 4 x 4 sudoku than our solver. Consequently, for small easy puzzles our design is more suitable to obtain a solution

5. Detailed Design

Our work for this problem consisted of two main parts:

1. designing the data structure for puzzle and writing its various functions.
2. designing a solution that keeps track of error and makes use of amb.

The steps we took to follow the calculation described in solution outline are:

5.1 Fill Sudoku

One of the most complicated part of our function was filling the empty spots such that each sub grid have all the numbers from 1 to n for a n × n Sudoku. So to do that first we had to sort all the cells by sub grid. For that we used accumulate function that accumulates a function in a list. We accumulated a function that extracts first n items from each sublist inside the Sudoku to extract sqrt(n) items. It is similar to transposing row to columns. Then we have a jumbled list of sqrt (n) items which we extract to get something like the following:

For 4 × 4

(0, 0)	(0,1)	(0, 2)	(0, 3)
(1, 0)	(1,1)	(1, 2)	(1, 3)
(2, 0)	(2,1)	(2, 2)	(2, 3)
(3, 0)	(3,1)	(3, 2)	(3, 3)

We got the following list:

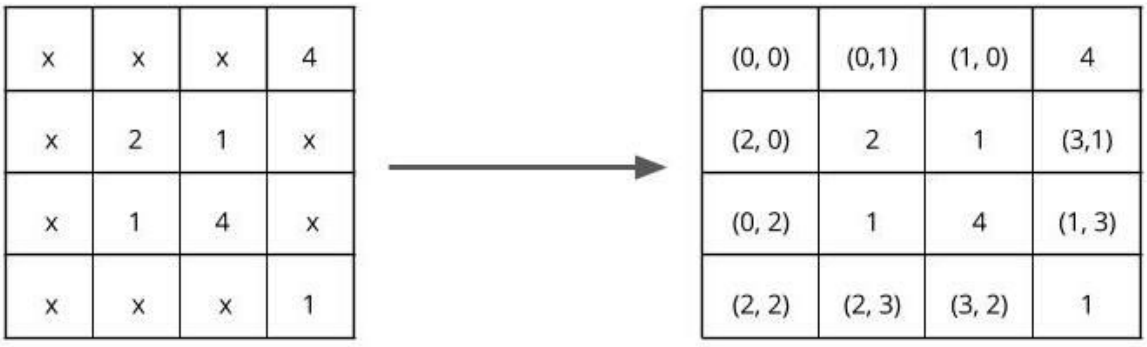
(0, 0)	(0,1)
(1, 0)	(1,1)
(2, 0)	(2,1)
(3, 0)	(3,1)
(0, 2)	(0, 3)

(1, 2)	(1, 3)
(2, 2)	(2, 3)
(3, 2)	(3, 3)

Then we split the list into n place to get list sorted by sub grid.

(0, 0)	(0,1)	(1, 0)	(1,1)
(2, 0)	(2,1)	(3, 0)	(3,1)
(0, 2)	(0, 3)	(1, 2)	(1, 3)
(2, 2)	(2, 3)	(3, 2)	(3, 3)

Using that sort function we fill the Sudoku. First we replace all empty spaces with their coordinates. Then we sort all by sub grid. Then we have list of numbers and sub grid. If we remove all number we get coordinates of empty cells sorted by subgrid.



Similarly if we sort unsolved Sudoku without any modification then we get unsolved Sudoku sorted by sub grid. Remove all empty spaces from that list and do set subtraction of all sub lists by a list of numbers from 1 to n. The output list is same size as list of coordinates of empty cells sorted by sub grid. Then just fill the Sudoku by changing each values in coordinate list with value in output list at same positions in each list. We shuffled the sub lists to get random fill. This satisfies one of the condition of Sudoku i.e. each sub grid must have all 1 to 'n' numbers.

5.2 Swapping numbers

The solution uses a random-swap function which will randomly select two coordinates from a subgrid to be swapped. The swap operation will then return the puzzle with the values for those coordinates switched. The swapping is done one coordinate at a time. We first conduct a general swap with temporary variables using let, then the puzzle is changed after each coordinate swap. The row of the coordinate is extracted, and then the value at the coordinates column position in the row is changed. The row is then appended back into its position, and the puzzle is set to the new puzzle with changed value. Carrying the operation for both the coordinates results in the puzzle with swapped values.

5.3 Calculating error

The error in Sudoku is number of repetitions in row, columns and sub grid. Since we fill the Sudoku so that there is no repetition in sub grid we only need to account for errors in rows and columns.

To calculate error in rows we simply count the number of repetitions in a row. We map this function on Sudoku so that we get a list of errors for each row. Similarly

we calculate column errors by first transposing the Sudoku and calculating row error for transposed Sudoku.

5.4 Solving the puzzle

For the main solve function we first find coordinates of empty cells sorted by sub grid. Using that we fill the sub grid to get test Sudoku. Then we send coordinates list to random infinite stream. finally we send this stream to amb evaluator that extracts an element of a list. this amb evaluator extracts a subgrid from infinite random stream. then we send this teas sudoku and coordinate to random swap function. the random swap function randomly selects two coordinates from coordinate list and swaps item on those coordinates in test-sudoku. if the swap does not increase error we accept the swap and we set test sudoku to this new value. Then assert function checks if error is 0. if not we repeat the calculation all over again. so this repeats until there is a solution.

A big problem we had here was that sometimes the swap function leads to a dead end. We accept only the swap that decreases error which improves speed of 4 X 4 sudoku solver by a lot, but for bigger problems the chance of dead end increases exponentially. We should accept all swaps to get an answer, but it could take a long time and still return no solution for a big sudoku puzzle. Also a simple backtracking would definitely give a solution for a sudoku that has a solution.

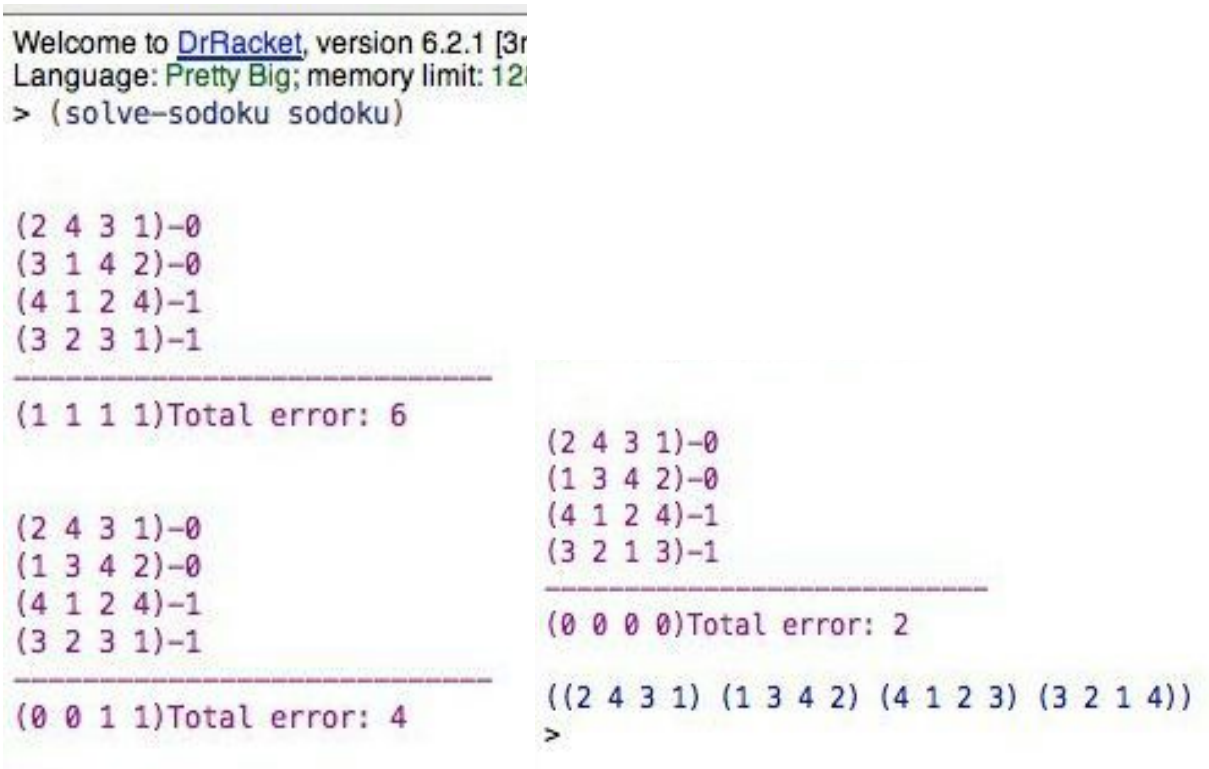
6. Sample Output

6.1 Solution for an easy 4 x 4 puzzle

We first initialize the sudoku board, with some filled and empty spaces. Little x here is acting as a placeholder for empty space.

```
(define sodoku (list (list 2 4 3 1)
                     (list x x x x)
                     (list x 1 2 x)
                     (list 3 x x x)))
```

The following is a correct solution to the above defined puzzle. The solver is randomly swapping elements within a subgrid until a correct solution is achieved. After the first swap, total error amounts to 6. With the following swaps error is reduced to 4, then to 2, and lastly to 0.



7. Future Development

Our code is good for small sudoku but for bigger problems we will need to modify it. We can create an extra backtracking algorithm that checks if we have hit a dead end, by checking if we are stuck on a same total error, and then going back few steps to redo swap. Or else we could restart the calculations altogether. This however may not much

better than doing regular backtracking. Another approach is to just swap items without caring for the error decrease from individual swap. This design would never hit a dead end, and we may get a solution. We are not sure if probability of not hitting a dead end in our algorithm is any more than probability of not getting a solution from random swapping. However in our algorithm we know when we hit a dead end and we can stop and rerun the calculations.

8. References

Harold Abelson and Gerald Jay Sussman, *Structure and Interpretation of Computer Programs.* MIT Press: Cambridge MA, 2011.

Lewis, Rhyd. “Metaheuristics can Solve Sudoku Puzzles.” *Centre for Emergent Computing.* http://www.rhydlewis.eu/papers/META_CAN_SOLVE_SUDOKU.pdf

Wikipedia. Https://en.wikipedia.org/wiki/Sudoku_solving_algorithms#Stochastic_search_.2F_optimization_methods

http://c2.com/cgi/wiki?AmbSpecialForm