Creating Sudoku Puzzles

Control # 2883

February 19, 2008

Abstract

We consider the problem of generating well-formed Sudoku puzzles. Weighing the benefits of constructing a problem bottom-up or top-down, we conclude on the top-down approach and implement a depth-first backtracking algorithm to generate complete Sudoku grids. We construct a difficulty metric which attempts to reflect the human method of solution, and implement this in the form of a logical Sudoku solver, extensible to allow the inclusion of various techniques of solution. A variant of our algorithm tests a given Sudoku and outputs a difficulty level. This agrees well with the classifications given by the existing puzzle setters. Using a recursive approach to ensure the solubility of all problems, we create an algorithm which repeatedly removes values from our completed grid, outputting problems which can only be solved through use of successively more difficult techniques. We aim to optimise the efficiency of our algorithms through reuse of prior calculations and incremental modifications at each step, recording only changes to the possible entries in cells. Finally we consider ways in which our algorithms could be improved.

Contents

1 Introduction

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

A Sudoku puzzle.

In 2005 a little known numerical puzzle, created in 1979, exploded on to the world scene. Commonly known as Sudoku, the simple yet challenging nature of this brainteaser has made it a popular pastime for a wide range of people. The puzzle is presented as a 9×9 grid, partially filled. The objective is to complete the grid so that each column, each row, and each of the nine 3×3 boxes contains the digits from 1 to 9, exactly one time each. It is assumed by the solver that the puzzle will have been generated to have both the existence and uniqueness of a full solution.

Nearly all of the daily newspapers contain Sudoku in their puzzle sections, but they are not all made by hand.... To satisfy the huge demand for new Sudoku, setters are turning to computers as a primary means of generating these puzzles. The wide range of ages and experience as well as personal preference for the difficulty of puzzles has increasingly required the setters to produce Sudoku of a defined standard. We shall attempt to produce an algorithm which will meet this goal.

2 Specifications

We require an algorithm to generate Sudoku puzzles. This algorithm must satisfy the following criteria:

1. *All generated puzzles must have a solution.*
   When people are trying to solve a puzzle they expect it to have a solution!

2. *All generated puzzles must uniquely define this solution.*
   If we do not take this criterion then even a blank grid could be considered to be a well-defined puzzle. This is clearly not in line with the public view of a good Sudoku. When comparing solutions with other solvers uniqueness is desirable. A non-unique solution cannot be attained without making an arbitrary choice at some point in the solution.

3. *The algorithm must be able to produce puzzles of at least four specified levels of difficulty.*
   In order to accommodate the wide variety of skill levels and preferences of the general public it is preferable to have multiple specified levels of difficulty.

4. *The algorithm must be able to produce a reasonably large number of puzzles.*
   The benefit of an algorithm is to produce many puzzles with relative ease and speed. To produce only a few puzzles it would be as efficient to work by hand.

5. *The algorithm should be designed so as to have as low a complexity as possible.*
   This is general good practise in algorithm design and ensures that the program can be run easily on even low powered computers. This also means that if large quantities are required then they can be produced relatively quickly.

To assist with the third point we must also develop a metric on the Sudoku puzzles which will be used to define different levels of difficulty of a puzzle. This should based upon the perceived human difficulty as opposed to computational ease or speed.

3 Definitions

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

A grid obeying the rules for Sudoku, generated by code given later.

- A **grid** is defined to be a 9×9 array.
- Each grid is arranged into 3×3 squares or **blocks**.
- Each number within the grid occupies a space called a **cell**.
- The **rules** of Sudoku are as follows:
  1. Each column contains the digits 1 to 9 exactly once.
  2. Each row contains the digits 1 to 9 exactly once.
  3. Each block contains the digits 1 to 9 exactly once.
- A **Sudoku grid** is a grid which obeys the rules of Sudoku.
- A **puzzle** is a grid supplied by a setter which contains both blanks and completed cells.
- A **solution** to the puzzle is the Sudoku grid that is consistent with the initial conditions.
- A puzzle is said to be **valid** if it admits a solution.
- A puzzle is said to be **well-defined** if there is exactly one solution.

4 Developing Our Approach

4.1 Puzzle Generation

We have considered two different approaches to generating the Sudoku puzzles.

1. Bottom up approach.
   First we start with an empty grid and then insert numbers in such a way so as not to invalidate the grid until we have a well-defined puzzle. For this approach we must address the issue of which additions do not invalidate our grid.
2. Top down approach.
   We begin with a full Sudoku grid and remove cell entries whilst ensuring that the puzzle remains well-defined. To implement this approach we must find a method of producing full Sudoku grids. It would be interesting to explore which removals would be unlikely to cause a multiple solution.

A fundamental aspect of both these approaches would be the problem of determining whether a puzzle is well-defined. Our research has shown that whilst the problem of whether a general puzzle has a unique solution has been shown to have a simple mathematical expression, it has also shown that actually finding such an expression is an ongoing problem.[1] We hope to ensure by the design of our algorithm that we only produce well-defined puzzles.

If we were able to implement both these approaches, it would be interesting to compare their efficiency.

4.2 Difficulty Levels

From personal experience of solving Sudoku, the main difference in difficulty that we have experienced is that between puzzles which can be solved by simple comparisons and those which require speculative reasoning at some point in the solution. We define speculative reasoning to be anything involving the supposition of a particular possibility in a cell. A solver must then either work towards a contraction (ruling out that possibility) or a full solution. Our metric must therefore make this distinction in difficulty.

Another possible metric could rely on the number of entries given in the initial puzzle. This supposes a correlation between the density of the initial conditions and the difficulty of the puzzle. From experience, however, it is possible to reach a point at which a significant proportion of the grid is filled but the next step is hard to take. Similarly, a particular arrangement of digits in a sparse grid can still be simple to solve. For these reasons we will not take this idea further.

5 Generating full grids

5.1 Brute Force

A brute force method to generate full Sudoku grids would be to randomly fill the grid with digits from 1 to 9. Then we could check that the grid satisfied the necessary rules, discarding it otherwise. The number of 9×9 Sudoku grids was calculated in 2005 by Bertram Felgenhauer and Frazer Jarvis. The figure was of the order of 6.6×10^{21}. [2] The total number of grids that can be generated is 9^{81}. Hence the proportion of the total number of grids randomly generated which would satisfy all the Sudoku conditions is approximately $3 \times 10^{-54}\%$.

Clearly this is such a tiny proportion that it makes this method impossible. By seeding, however, ie. supplying initial information to constrain the possibilities, we can make the method more plausible. For example, we could place all 9 occurrences of the digit 1 in such a way as to comply with the rules before using the brute force method. Alternatively we could generate a random permutation of the digits from 1 to 9 and place them down the first column. This would automatically comply with the rules. To extend this we could continue generating numbers at random and only inserting them when they comply with the rules. This is the essence of the next method.

5.2 Sequential Method with Backtracking

5.2.1 Description

An overview of this method is as follows. We create an array **exceptions** which will contain a list for each cell in the grid, corresponding to the numbers which cannot go into that cell (ie. because there is already an occurrence of that number in the same row, column or block). For each cell we have a random permutation of the numbers from 1 to 9, denoting the order in which we will try them in that cell. We will traverse the cells sequentially, trying the first possibility for that cell. If it does not break the row, column or box condition (ie. is not in the **exceptions** list for that cell) then provisionally put the digit in that cell and continue to the next cell. Else continue through the permutation for that cell, stopping if we reach an acceptable number. If no number works then we return to the previous cell and the previous **exceptions** and try the other possibilities.

5.2.2 Pseudocode

```
Define cell = (row, col)

CREATE ARRAY g (will be the output)
CREATE ARRAY exceptions
CALL F((0,0),exceptions)
PRINT g

Define F(cell, exceptions)
LET $A_{cell}$ = a random permutation of the integers from 1 to 9 inclusive
 1.  FOR     $m \in A_{cell}$
     LET     $A_{cell} = A_{cell} \setminus \{ m \}$
     DO
             IF        $m \notin$ exceptions(row, col)
             THEN      g(row, col) = m
                       place m in the exceptions for all the
                       cells in that row, column and block
                       CALL F(next cell, exceptions)
             ELSEIF    $A_{cell} = \phi$
             THEN      return to previous cell and GOTO 1.
     ENDDO
```

5.2.3 Discussion

This algorithm will always return a full Sudoku grid, since when problems are encountered it can backtrack until it reaches a position from which a Sudoku grid can be reached. Moreover, our algorithm can generate any possible Sudoku grid, simply by getting the right corresponding combination of random numbers.

6 Difficulty Metric

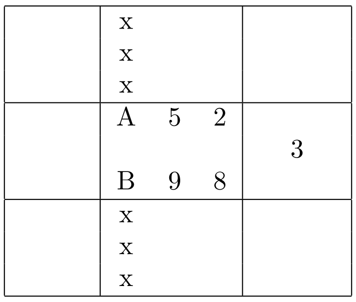


Figure 1.

Assumption: We assume all puzzles to be well-defined.
We will define our metric by considering the most advanced method which needs to be used to solve a particular puzzle. The methods are defined below in increasing levels of sophistication. Our ordering reflects our assumptions about human reasoning and the ease of various deductive processes.

Method 1 If there is only one possible digit for a specific cell under the Sudoku rules, then we may insert this digit.

Method 2 If in any row, column or box, there is only one possible cell for an unused digit, then we may insert this digit.

Method 3 If two cells on a row have the same two candidates then any other cell on that row cannot take those values, and we can therefore remove these possibilities from their candidate list. This can also occur for more than two cells.

Method 4 Interactions between blocks, columns and rows can reduce the possible candidates for other cells. For example, in Figure 1, since 3 must occur in either position A or B, it cannot occur in any of the cells marked x.

Method 5 Since we have assumed that there is a unique solution then if we have a situation where you can't distinguish between two numbers and there is another possible number then that other possibility holds.

Method 6 If speculative reasoning is required, we can look at the number of levels recursion needed.

Further Methods There are also other advanced techniques for solving Sudoku which could be used in generating a finer metric.[3]

7 The Algorithm

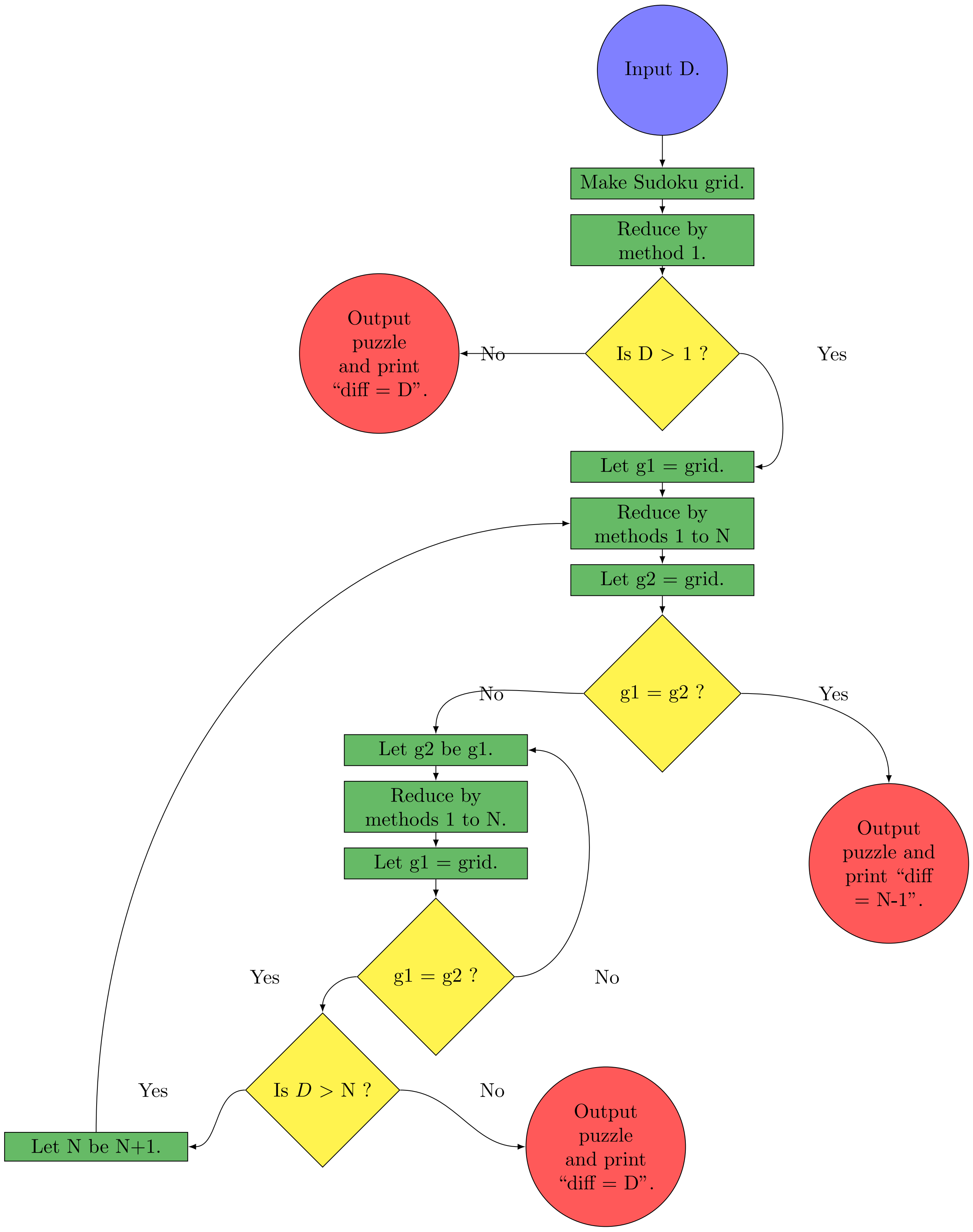
Our algorithm uses the top down approach to generate Sudoku puzzles of a defined level of difficulty. Subroutines from within our algorithm can also be used to test the difficulty level of any specific puzzle. To generate a puzzle of, say, the third difficulty level we will allow entries to be removed if the resulting puzzle can be solved using only methods 1 to 3. This also guarantees uniqueness since the solution of an ill-defined puzzle would have a point at which our methods could not determine between multiple possible entries for a cell. Flow diagram 1 below shows an overview of the algorithm, in particular the way in which we output a puzzle of a specified level of difficulty. As shown, this will produce a puzzle which will be up to the level of difficulty specified, and will inform the user of the difficulty of the produced puzzle. If it was important to have precisely the level specified then the algorithm could then be run multiple times until a puzzle of that difficulty level was reached.

When we remove an entry we apply the methods as many times as possible. We will then either reach a full Sudoku grid and hence can continue attempting to remove entries or a point at which the methods give no more information. The algorithm terminates when there is no entry that we can remove and still reach a full Sudoku grid. Hence the puzzles generated are minimal for the specified level of difficulty.

Flow diagram 1 on the following page has several boxes labelled "reduce by methods 1 to N". This is described by the pseudocode below. Note that methods 2 to 6 are just used to narrow down the possibilities matrix. We then use method 1 to actually update the grid.

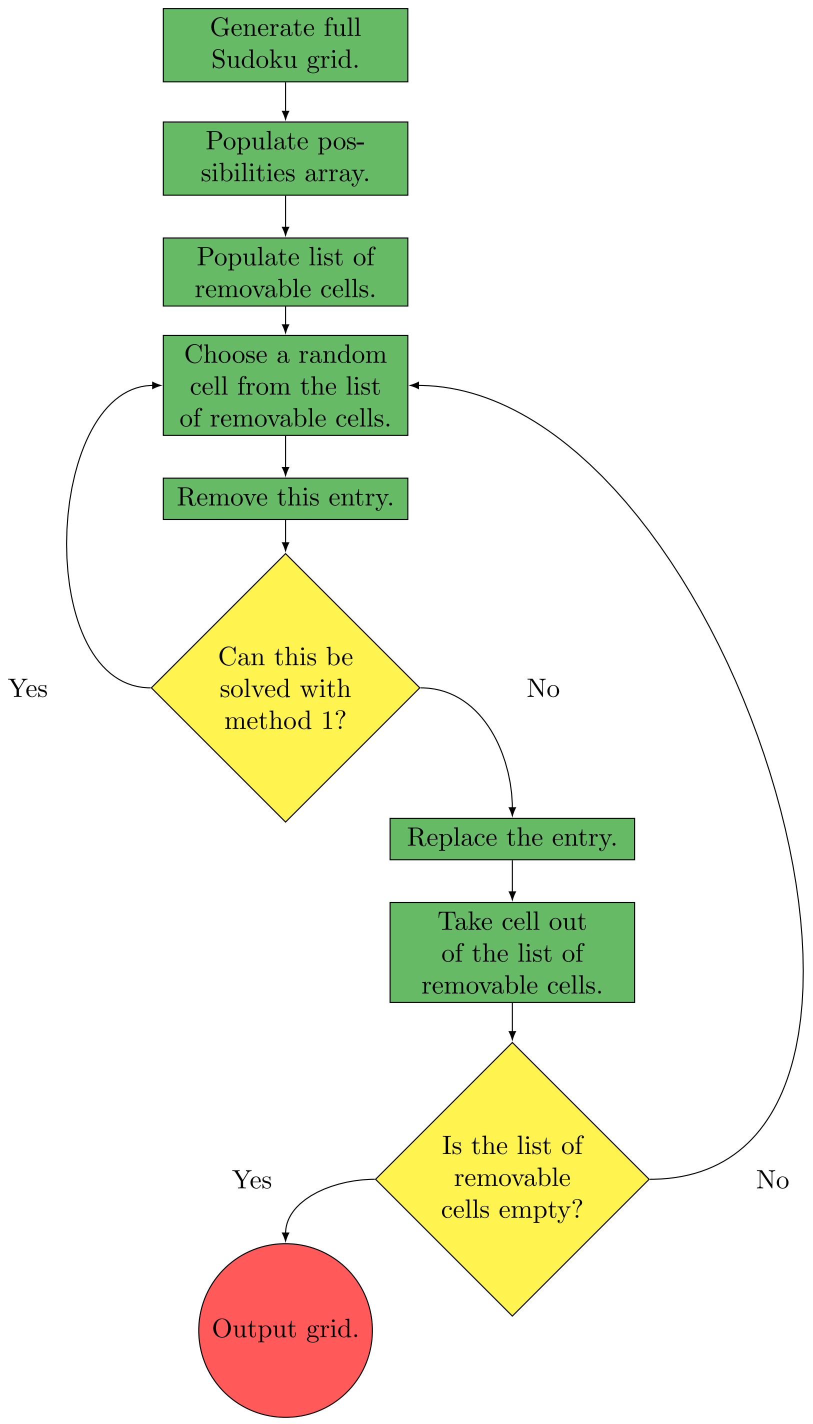
```
INPUT puzzle
GENERATE RANDOM cell
MAKE LIST removables
PLACE all cells in removables

G(N, puzzle, removables, cell)
REMOVE entry from cell
REMOVE cell from removables
  DO UNTIL no change   use method 2 to reduce cell possibilities
                       ...
                       use method N to reduce possibilities
                       use method 1 to update grid
  IF grid = full   THEN   STOP
                   ELSE   replace entry in cell
                          choose new cell randomly from removables
                          G(N, puzzle, removables, new cell)
```



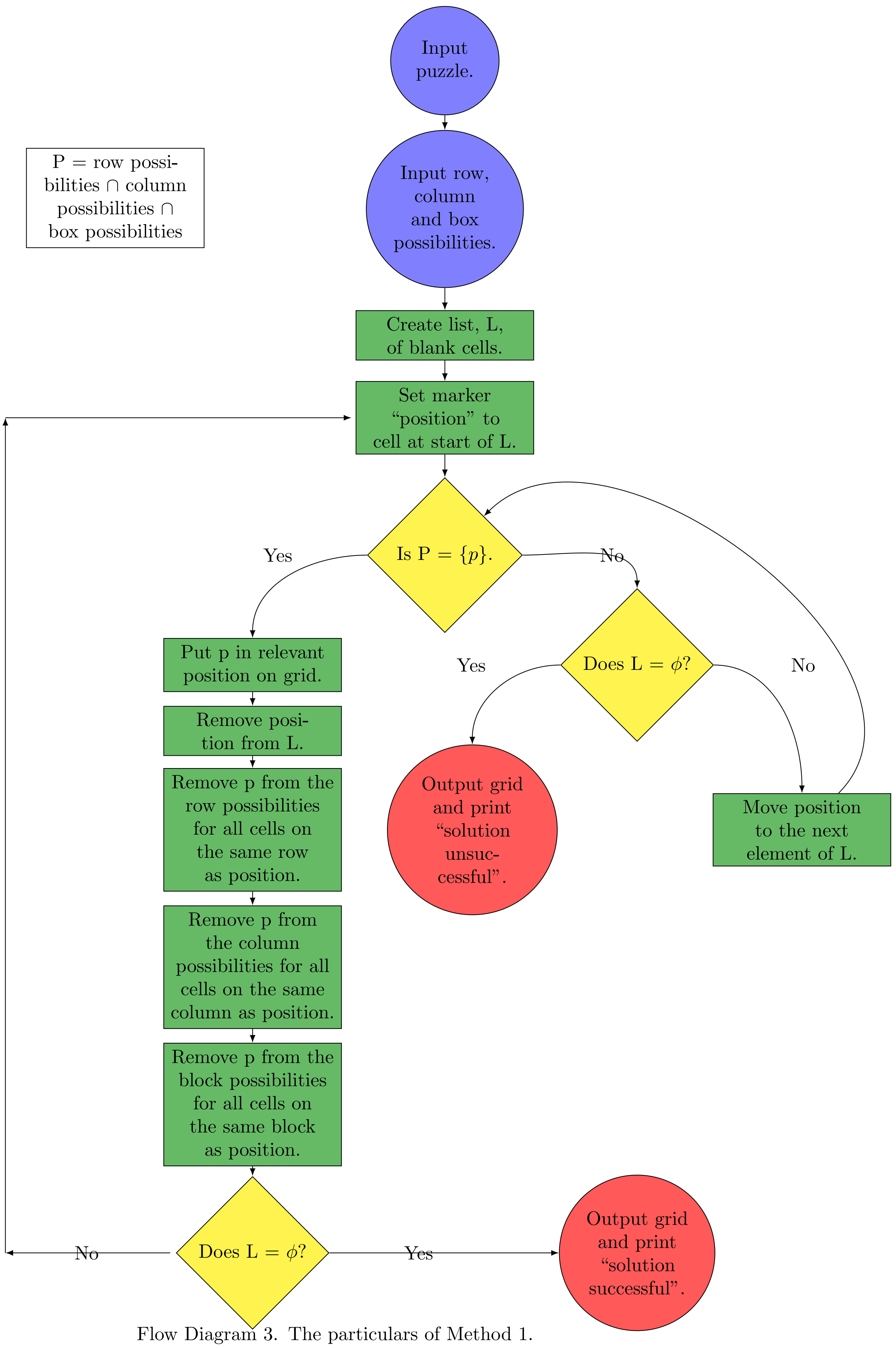
Flow Diagram 1. An overview of our algorithm.

To test the level of difficulty of a given Sudoku puzzle we apply the methods as in Flow diagram 1 classifying the puzzle as the level of the most advanced method used to solve it.


Generate full Sudoku grid.
Populate possibilities array.
Populate list of removable cells.
Choose a random cell from the list of removable cells.
Remove this entry.
Can this be solved with method 1?
Yes
No
Replace the entry.
Take cell out of the list of removable cells.
Is the list of removable cells empty?
Yes
No
Output grid.

Flow Diagram 2. The production of a difficulty level 1 puzzle by the process "Reduce by method 1."

Flow Diagram 2 shows in more detail how method 1 is used to produce a well-defined puzzle of difficulty level 1. This involves maintaining a list of potentially removable cells and then testing each cell in this list to see whether it can be removed to give a well-defined puzzle. If it cannot be removed then we replace it and update the list of removable cells so that it does not include this cell and then try the next cell in the list. If it can be removed, then take it out of the grid and move onto the next cell in the list. When there are no more cells that can be removed then we output the resulting puzzle.



Input puzzle.
Input row, column and box possibilities.
P = row possibilities ∩ column possibilities ∩ box possibilities
Create list, L, of blank cells.
Set marker "position" to cell at start of L.
Is P = {p}.
Yes
No
Put p in relevant position on grid.
Remove position from L.
Remove p from the row possibilities for all cells on the same row as position.
Remove p from the column possibilities for all cells on the same column as position.
Remove p from the block possibilities for all cells on the same block as position.
Does L = ϕ?
Yes
No
Output grid and print "solution unsuccessful".
Move position to the next element of L.
Does L = ϕ?
No
Yes
Output grid and print "solution successful".

Flow Diagram 3. The particulars of Method 1.

An example of how our algorithm will approach a particular method is shown in Flow Diagram 3 using method 1 as an example. It is clear from the description of the other methods how similar algorithms can be designed to execute each one. Due to time constraints we have not actually written these down here, although method 2 has been implemented (see results). We maintain a list for each cell of the digits that have not yet been used in that column. A similar list is maintained for the rows and for the blocks. For each cell we then intersect the three lists of possibilities and if this results in a singleton then this must be the correct digit for this cell. We iterate through the cells filling in the digits obtained in this way until a full Sudoku grid is reached and we have a successful solution or we can no longer fill in any more cells and the solution is unsuccessful.

Any well-defined Sudoku puzzle could ultimately be solved by the use of Method 6, the speculative reason method, alone. This would be an inefficient method but would guarantee a solution. Hence when we are generating puzzles of up to our top level of difficulty we are able to generate any well-defined Sudoku puzzle.

8 Results

8.1 Implementation

To implement our algorithm we decided to use the programming language Lisp. Lisp primarily deals with lists. We saw this as an appropriate data structure with which to encode our Sudoku grids. We also have some familiarity within our team of programming in this language. Computer implementation provided a way of testing our algorithm and producing some results. It also enables us to test our algorithm in ways that would be impractical by hand. For example, testing Sudoku that have been given specified levels of difficulty by others and comparing with our own classification.

8.2 Output

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

The Sudoku above was generated at difficulty level 1 by our program. This means that it can be solved using method 1 alone. Comparing this to Sudoku puzzles generally in use, our puzzle does seem to have fewer initial blanks. To extend the length of potential solving time for the puzzler it would be preferable to have fewer initial entries. This should be achieved when extending to the use of other methods.

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

This Sudoku was generated by our level 2 algorithm, and has slightly fewer initial entries than the puzzle generated by method 1 alone.

Due to time constraints we have not implemented any further methods.

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

The Sudoku above is taken from web Sudoku [5], where it is classified as 'easy'. There are 4 levels of difficulty specified on this website, of which easy is the simplest. Our algorithm classifies this puzzle as level 1, ie. it can be solved using method 1 alone. We have tried several other 'easy' puzzles, each of which is classified by our algorithm as level 1 or 2. When we move up to 'medium' puzzles we find that most puzzles are of level 2. Since we haven't yet implemented any further levels of difficulty we cannot determine the level of any 'hard' puzzles, but they require more complicated methods than methods 1 or 2.

We have also modified this implementation to allow us to test for difficulty levels between grids of level 1 and level 2. We do indeed find puzzles that were specifically generated to be of difficulty level 1 to be so. Similarly, for difficulty level 2.

9 Complexity

In discussing the complexity of our algorithms, we consider three factors:

- Number of operations.
- Memory usage.
- Program size.

In our implementation, we have chosen in various places to favour one or more of these conditions over the others. Where this has been done, it will be explained and justified.
The natural method of analysing complexity is to consider what must be done to create a single Sudoku problem. Under such a weighting, however, our algorithm will generally fare poorly; we take the approach of generating a number of problems at once, of varying levels of difficulty. This is done primarily for aesthetic reasons - it was felt that one should be able to generate puzzles of all levels of difficulty from the same complete grid - but also allows us to reuse some calculated data for multiple produced problems. As such, we consider the complexity of producing a set of Sudoku of varying difficulty levels.
Our solution involves multiple steps:

1. Produce a complete grid satisfying Sudoku constraints.
2. Remove points.
3. Establish whether our created problem has a unique solution.

We will consider the complexity of each part individually, highlighting how we feel this could be improved, before considering the way the entire solution works and any resultant enhancements that could be made.

9.1 Produce a complete Sudoku grid

We implement a depth-first backtracking algorithm in order to generate our complete grid. This sequentially places random numbers until it reaches a constraint violation, then backtracks until it has another viable choice and tries again. This algorithm runs fairly quickly,

generating 100 puzzles in 0.016998 seconds (system time) under an interpreted system. System memory usage remained under 5MB during the operation.
One could of course skip this step, simply adding in numbers to the board and testing for a unique solution. As such, it could be claimed that this is all wasted time. However, as showed later, our solving algorithm is (relatively) slow, since it attempts to mimic a human process, so it makes sense to limit the possibility space for that technique here, when we are using a relatively fast algorithm.

9.2 Remove points

Our algorithm here is relatively simple - we maintain a state of points which we have tried removing, and iterate until this list is empty. This step controls the generation of multiple difficulty levels, however, and it is worth noting how the algorithm could be simplified in this regard. At present, we evaluate the uniqueness of each partial puzzle uniquely, failing to take into account any results we have gained from previous partial puzzles. Since we remove points progressively, we could keep a copy of the previous unique solution - at least using method 1, one is quite likely to reach this this solution, whence we can claim uniqueness by induction.

9.3 Check the uniqueness of our solution

We have to an extent made deliberate sacrifices in efficiency of this step in order to use a 'human' process and to incorporate a method of establishing difficulty. We note for example that one could easily persuade item (1) to solve any given Sudoku, fixing the initial points and allowing it to continue past finding an initial solution. One can take a brute-force approach to the problem, treating it purely as a constraint resolution problem - algorithms such as Dancing Links can be used to solve hundreds of problems per second. However, taking such approaches misses the fundamental point of the exercise for the majority of its participants, and so we did not see it appropriate to use them.
Given this, our method attempts to mimic the human solving process - a naturally inefficient affair. Our method is designed to subject the problem to logical analysis to the greatest extent possible before giving up and resorting to conditional reasoning. We loop over the grid, calculating constraints and filling in numbers where possible. We then use other rules to limit the possibilities for each cell, and repeat. To an extent our algorithm is blind to the nature of the changes it has made - upon making a change, it will loop again over the grid, irrespective of which points have had changed possibilities. A more efficient algorithm would perhaps construct a list of cells to look at, appending ones to the end when their possibilities updated. This would perhaps also more closely mimic the human process. Given the size of the grid, however, this would not make the algorithm vastly more efficient.

9.4 The algorithm as a whole

On observation, our algorithm performs a lot of backtracking in both directions - building a grid, removing numbers from it and trying to put them back in again. We may repeat calculations already performed in a number of steps, and currently take no account of the fact that we have done so. We do however avoid a lot of repeated calculation through passing things such as the possibility matrices between multiple stages, marking only the changes at each stage.

10 Conclusions and Evaluation

To summarise, we have created an algorithm to produce well-defined Sudoku of varying difficulties. We have succeeded in implementing this for two difficulty levels, and have described how this could be extended to at least four additional levels. Our metric is designed to reflect human ease of computation, each level requiring a more complicated rule. However, as we gained more familiarity with our different methods, we have uncovered some controversy. Our team is divided as to which of methods 1 and 2 is easier to use when solving Sudoku by hand. We have tried to optimise the efficiency of our algorithms through reuse of prior calculations and incremental modifications at each step, recording only changes to the possibilities array.

11 Improvements and Alternative Approaches

There are a number of ways in which our algorithm could be improved. Firstly, we could keep a record of those states which have formed a well-defined Sudoku puzzle so that if we return to these states later in the algorithm we do not need to recalculate from this point. Secondly, we may be able to infer more of these saved states by arguing by symmetry. We could also revisit the idea of seeding, as discussed earlier. The decrease in complexity of the algorithm may fully justify the possible difficulty in implementation.

A different approach to this problem, would be the bottom up approach as detailed before. It would be interesting to see how this would affect the difficulty levels of the puzzles generated. We would speculate that this would make the puzzles generated easier, since it would create inherent relationships between different cells.

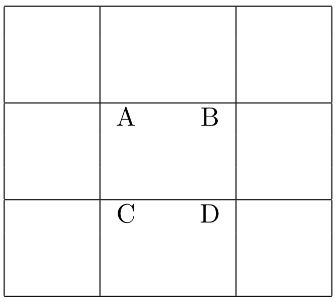


Figure 2. Possibilities for cells A, B, C and D = { 1, 2 }

A further improvement to the algorithm, mentioned earlier, would be to consider which removals would be unlikely to lead to an ill-defined puzzle. We have not found scope to take this idea further in the short time available to us. The concept behind this idea is that certain necessary conditions for well-definedness are clear. For example, if we have the arrangement in Figure 2. then taking either (A = D = 1 and B = C = 2) or (A = D = 2 and B = C = 1) leads to a consistent solution and hence the problem cannot have been well-defined. We could add an extra test to our algorithm to prevent this situation from occurring. Experimentation could help us to decide upon a sensible balance between the number of arrangements to consider and the decrease in the number of steps that this would allow if such a situation were avoided.

The types of methods we have employed within this project could be extended to investigate other problems connected to Sudoku, such as larger grid sizes. A current area of interest is the minimal Sudoku problem for 9×9 grids, ie. the minimum number of given initial entries in a well-defined puzzle. To date, there are many well-defined puzzles with 17 initial entries and it is speculated that no puzzles exist with smaller starting conditions but this has not yet been proved.[4]

References

[1] Agnes M. Herzberg and M. Ram Murty. *Sudoku Squares and Chromatic Polynomials.*

[2] Jarvis, Frazer. *Sudoku enumeration problems.* http://www.afjarvis.staff.shef.ac.uk/sudoku/

[3] *Sudoku solving techniques: Advanced techniques.* http://www.su-doku.net/tech2.php

[4] Gordon Royle. *Minimum Sudoku.* http://people.csse.uwa.edu.au/gordon/sudokumin.php

[5] *Web Sudoku.* http://www.websudoku.com/