

Table of Contents

Section 1: Introduction.....	pg. 2
Section 2: Terminology and Techniques.....	pg. 3
2.1: Basic Definitions	
2.2: Typical Techniques	
Section 3: Assumptions.....	pg. 7
Section 4: Model and Justification.....	pg. 8
4.1: Algorithm to Generate a Completed Puzzle	
4.2: Algorithm to Remove Tiles	
4.3: Solving a Sudoku Puzzle Logically	
4.4: Difficulty Analysis Formula	
4.5: Strengths and Weaknesses of Our Approach	
Section 5: Model Testing and Sensitivity Analysis.	pg. 14
5.1: Model Testing	
5.2: Sensitivity Analysis	
Section 6: Algorithms.....	pg. 15
Section 7: Conclusion.....	pg. 16
Appendix A-G: Flowcharts of Algorithms.....	pg. 17

Section 1: Introduction

Sudoku is a logic puzzle that is continuing to increase in popularity throughout the world. The objective of the game is to fill a 9x9 (typically) grid with numbers so that each row, column, and box contains each integer 1-9 once and only once. A Sudoku puzzle starts you off with a certain number of values filled in. These are clues to help you deduce the rest of the grid. A valid Sudoku board should have a unique solution, meaning you can only fill in the grid in one way.

Due to the growing popularity and demand for Sudoku puzzles, the need for new and intriguing puzzles is on the rise. We have created a program to generate a valid board as well as a unique and solvable puzzle. A program like this could be used by companies publishing Sudoku puzzles or by people in their homes playing on-line.

The task of determining a difficulty level for a puzzle is controversial, as the difficulty of a puzzle does not depend exclusively on what the givens are. Rather, the relevance of the givens is where the difficulty of a Sudoku puzzle really lies. We have developed an algorithm which calculates a puzzle's difficulty based on the number of basic versus advanced steps in logic that a normal person would typically follow. This is just one way of determining the difficulty of a puzzle.

We also recognize that the number of guesses one would have to make if a dead end was found increase the difficulty significantly. We do not generate any puzzles that require the solver to guess because we feel that you should be able to logically eliminate all the possibilities of a Sudoku puzzle without being forced to guess.

2.2 Typical Techniques

- **Single:** A single is the only candidate for a cell, with no other candidates possible.

	C1	C2	C3
R1	1 3 5	2 3	1 2
R2	1 5 9	4	1 9
R3	7	8	6

Figure 2.2

Notice that C2R3 has only one possible value. Hence C2R3 *must* be the correct position of the eight.

- **Hidden Single:** A hidden single is a candidate that can only occupy a single cell. The value is “hidden” because other candidates make the single harder to spot if the solver records all possible values for the cell.

	C1	C2	C3
R1	1 3 5	2 3	1 2
R2	5 9	4	1 5 9
R3	7	1 2 8	6

Figure 2.3

Notice that eight is only shown once in this sector. Since every area must contain each number once, C2R3 *must* be the correct position of the eight.

- **Locked Candidates:** A locked candidate can be identified in two ways.

The **first type of locked candidate** is finding a sector in which a candidate is restricted to one row or column. This implies that the candidate for that row or column must be in that sector. Hence it can be eliminated from all other cells in that row/column.

	C1	C2	C3	C4	C5	C6	C7	C8	C9
R1	1 2 7 6	4	1 6 7 6	1 5	1 2 5 5	1 8	1 6 7 6	3	1 6 7 6
R2	5	2 8 9	1 3 9	6	1 2 3 4	7	1 2	4 9	2 4 8
R3	1 2 3 7 6	1 3 7 8 6	1 3 7	1 8	9	1 2 3 4 6 8	1 6 8	5	1 6 7 8

Figure 2.4

Notice that in the rightmost sector, the only row that a 2 can be in is row 2. Hence the 2 *must* be in the middle row of that sector, and thus cannot be in middle row of any other sector. Thus the 2 of C5R2 and of C2R2 can be safely eliminated.

The **second type of locked candidate** is finding a row or column in which a candidate is restricted to one sector. This implies that the candidate for that sector must be in that row/column. Hence it can be eliminated from all other cells in the sector.

Notice that in column 1, the only sector that a 3 can be in is the top sector. Hence the 3 *must* be in the first column of that sector, and thus cannot be in any other column of that sector. Thus the 3 of C3R1 and of C3R2 can be safely eliminated.

	C1	C2	C3
R1	5	1 7 8	1 3 4 7 8
R2	3 6	2	1 3 6
R3	3 4 6 8	1 7 8 6	9
R4	6 8	5	2 7 8 6
R5	1	9	2 4 7
R6	4 8	3	2 4 7 8
R7	2	1 8	5
R8	9	4	1 3
R9	7	1 6 8	1 3 6 8

- **Naked Pairs:** A naked pair is two identical cell possibilities in a particular area.

	C1	C2	C3	C4	C5	C6	C7	C8	C9
R1	5	8 9	1 9	6	1 2	7	1 2	4 3 9	2 4 8

Figure 2.6

Notice that 1 and 2 are the only possible values in both C5R1 and C7R1. This implies that they *must* occupy those two spaces and hence, the 1 in C3R1 and the 2 in C9R1 can be safely eliminated from their perspective cells.

- **Hidden Pairs:** Hidden pairs are identified by the fact that a pair of numbers occur in only two cells of an area. They are "hidden" because the other numbers in the two cells make their presence harder to spot.

	C1	C2	C3
R1	8	1 7	2 7 9
R2	4 5 7	3 4 5 7	3 4 7
R3	1 2 3 5 9	1 3 5	6

Figure 2.7

Notice that the values 2 and 9 are present only in C3R3 and C1R3. This implies that they *must* be in those two cells and hence the 7 from C3R1 and the 1, 3, and 5 from C1R3 can be safely eliminated.

- **Naked Triples:** Naked triples, as the name suggests, are three numbers that do not have any other numbers residing in the cells with them. Unlike naked pairs, naked triples do not need all of the three candidates in every cell. Quite often only two of the three candidates will be shown.

	C1	C2	C3
R1	5	6	1 7
R2	9	8	1 2 4 7
R3	3 4	2 3 4	2 4

Figure 2.8

Notice that the candidates of 2, 3, and 4 are the only three candidates that occupy C1R3, C2R3, and C3R3. This implies that these three numbers *must* be in these three cells. Hence the 2 and 4 from C3R2 can be safely eliminated.

Section 3: Assumptions

- **Typical 9x9 Sudoku Board**

We assumed a 9 by 9 Sudoku board because this is the most widely used size of Sudoku board. Although these puzzles can also be created with letters, we created our puzzles with the typical integers from 1 to 9. The methods for creating Sudoku boards with letters, however, follow very closely with the algorithms for integer values.

- **Human would be solving by hand (not computer) and using no hints**

Since the majority of people who work on Sudoku puzzles do not use a computer to work out the puzzles (puzzles are usually solved using paper and pencil), and because most people would not know how to write a program to solve the puzzle automatically, we thought this would be a logical assumption. Furthermore, because most Sudoku puzzles on paper do not give hints, we included this fact in our assumption as well.

- **Board will be solvable by logical techniques and have a unique solution**

This assumption is not only a requirement for this modeling question; it is a requirement in general for a Sudoku puzzle to be a valid Sudoku puzzle. Real Sudoku puzzles must be solvable and have a unique solution to be valid.

- **We assume that the knowledge needed to solve a puzzle we have generated does not exceed the knowledge of a human solver.**

This assumption is for testing our model of difficulty rating. Since our method of generating a board depends on the techniques we have programmed, we cannot generate a board which requires further techniques. If we had more time, we could have incorporated more techniques.

- **We assume that each technique requires its own constant amount of time to complete.**

To make our calculations easier, we assumed that simple techniques take the least amount of time, and that subsequently harder techniques took longer amounts of time. We assigned constant values to each distinct method for calculation purposes. The values for the constants we chose are essentially irrelevant. However, we increased the magnitude of the constants as the techniques became more difficult to reflect the need for more time to execute the technique.

Section 4: Model and Justification

We created an algorithm in Java which generates a complete 9x9 Sudoku board. We found that the most common (and simplest) way to create a correct Sudoku board is to create a complete board first and then erase tiles, checking after every step that the puzzle is still solvable and unique.

4.1 Algorithm to Generate a Completed Puzzle

```

GenerateRandomPuzzle()
    set puzzle = PrepareEmptyGrid()

    PopulateCell(puzzle, 1)
Return puzzle

PopulateCell(puzzle, cellIndex)
    if cellIndex > 81 then return true

    set list = GenerateRandomList(1, 9)

    for each value in list
        puzzle.getCellByIndex(cellIndex).setValue(value)

        if puzzle.isValid() then
            if PopulateCell(puzzle, cellIndex+1) then
                return true
            end if
        end if

        puzzles.getCellByIndex(cellIndex).clearValue()
    next value

return false

```

We use the `GenerateRandomPuzzle()` method to populate a normal Sudoku grid. The newly generated board adheres to all the constraints of a Sudoku puzzle.

The algorithm starts by preparing a blank board, and then recursively filling in values (starting with the first cell). After every value it sets, it checks to see if the board is still valid. If it is, it populates the next cell. If it ever reaches a point where it cannot place any of the 9 possible values, it returns false to the previous cell which picks up trying new tiles where it left off last.

Once the index is greater than 81 (the total number of tiles) then it returns true, and the algorithm just falls back through the recursive stack.

4.2 Algorithm to Remove Tiles

The next step to generating a Sudoku puzzle is the removal of individual tiles. The method we chose to do so removes a random tile and another tile that is mirrored on both the x-axis and the y-axis.

```

RemoveTiles (numToRemove, minLevel, maxLevel)
  let difficulty = solver.getPuzzleDifficulty()

  if numToRemove ≤ 0 and difficulty ≥ minLevel and difficulty ≤ maxLevel then
    return true
  end if

  set indexQueue = GenerateRandomList(1, 42)

  for each index in indexQueue
    if getTileByIndex(indexQueue[index]).value = 0 then
      StoreValuesAndClearTileAndMirror(indexQueue[index])

      if solver.hasUniqueSolution then
        if RemoveTiles(numToRemove-1, minLevel, maxLevel)
          return true
        end if
      end if

      RestoreValuesForTileAndMirror();
    end if
  next index

return false

```

This is the general procedure we use to remove tiles from a completed puzzle (Some steps have been simplified into more readable, fabricated method calls). It starts off by checking to see if it has reached its goal of removed tiles and is within the desired range of difficulty levels. We then generate a queue of tiles to try (note: Because we also remove a “mirror” we only need to try about half of the tiles).

Then it starts a loop where it will try every value in a random order. It first stores the values of the tiles before it assigns them new values. Then it stores them and checks to see if the board still has a unique solution. If it does, it calls itself again with the number of tiles to remove reduced by one. If that method call returns true, the whole thing ends. If it returns false, then the method will try to remove more tiles until it has exhausted all possibilities, at which point it returns false.

This method can be easily adapted to generate puzzles that are asymmetrical or show other types of symmetry.

4.3 Solving a Sudoku Puzzle Logically

In order to be able to analyze the difficulty of a Sudoku puzzle, we needed to approach it as a human would. Therefore, our model always uses the simplest methods first, resorting to the harder ones only when absolutely necessary.

```
SolvePuzzleLogically()
  set stuck = false

  while not stuck

    while not stuck

      set stuck = findSingles()
      loop

      set stuck = findHiddenSingles()
      loop

      findHiddenPairs()
      findNakedPairs()
      findLockedCandidates()

      if solver.isValid() then return true

return false
```

As you can see, the technique that gets run the most is (what we determined to be) the most basic technique. Only when that most basic technique fails will it move on to the next harder one.

If both `findSingles()` and `findHiddenSingles()` fail, we move on to the next cluster of techniques. The reason why we grouped these together is because we reasoned that after you exhausted all of the easiest possibilities, you now have to go through the extra effort to find possible values for many different squares and analyze them. At this point you are fully capable of recognizing situations where you can use these more advanced methods (Assuming you know them).

Simple changes to this algorithm can allow it to check for puzzles that are unique. This is, in fact, the method we use to determine uniqueness and board validity during the creation process.

Each of these “find” methods not only progress towards solving the puzzle, but they also keep track of every move they make. So we can take that data and incorporate it into a difficulty formula.

4.4 Difficulty Analysis Formula

We created an equation to establish the ranges of difficulties based on the number of various techniques the puzzle required. This allowed us to provide weight to each of the difficulty levels and gauge the actual technicality of the puzzle, which is what we use to judge difficulty.

There are three major factors which go into determining the difficulty of a Sudoku puzzle:

1. **Overall time:** ΔT

- As time increases, solvers interest in the puzzle decreases. And as interest decreases, so does the solvers ability to come to logistical conclusions.
- We assume that all techniques take a constant amount of time, and that harder techniques require longer periods of time.

$$\Delta T = (\alpha t_1 + \beta t_2 + \rho t_3 + \delta t_4 + \mu t_5)$$

t_i = the time it takes to complete a given technique

α = total number of singles

ρ = total number of naked pairs

β = total number of hidden singles

δ = total number of locked candidates

μ = total number of hidden pairs

2. **Knowledge of logistical methods:** K_p

- The number of distinct methods used in the solving process of a puzzle can increase its difficulty significantly.
- We assume that any number of methods is a set of the simplest methods available.

$$K_p = \sum \text{Distinct Methods Used}$$

3. **A solvers lack of knowledge:** K_u

- We assume that a solver who does not know a technique required by a given puzzle will have a very difficult time with that puzzle.
- We also assume that knowing advanced methods will not make a puzzle easier since the known methods would not be used in solving the puzzle.
- Therefore, we have defined K_u to be:

$$K_u = \begin{cases} K_p - K_s & \text{if } K_p \geq K_s \\ 0 & \text{if } K_p < K_s \end{cases}$$

*Where K_u is unknown knowledge, K_p is the puzzles required knowledge, and K_s is the solvers knowledge.

- While this factor is extremely important to determine the difficulty relative to an individual, we have not run any simulations that take lack of knowledge into account. This concept was developed late into the contest and we did not have any time to work with it.

Our Final Difficulty Formula:

We decided that both overall difficulty and the time it takes to solve a puzzle have nearly equal affect on the overall difficulty.

We also concluded that not possessing the needed knowledge would be detrimental to the solvers chance of completing the puzzle. So much so, that we figured the difficulty would change *exponentially*.

So our final formula is as follows:

$$D = (K_p \cdot \Delta T)^{K_u + 1}$$

*Where D is difficulty, K_p is puzzles required knowledge, ΔT is the puzzles total time to solve, K_u is unknown knowledge.

4.5 Strengths and Weaknesses of our Approach**Strengths:**

- Our algorithm models human behavior and techniques.

Since we assumed that humans would be the target audience, our model is designed to reason like a human in order to produce puzzles intended for humans

- Difficulty rating equation is straightforward and easy to follow.

Our model uses simple math to determine a difficulty of a puzzle. In order to avoid complexity, we used to most straightforward approach to evaluate a puzzles difficulty.

- We can generate a puzzle of any difficulty or style.

By simply changing a few parameters, the algorithm can produce any type of puzzle you want. You can even restrict what logic is allowed in the puzzle by simply removing method calls from the main solveLogically() routine.

- Easily expandable to include additional logic functions.

Any method call can easily be added into the main logic routine and improve the output of the algorithm.

Weaknesses:

- Due to time constraints, we were unable to program the upper level techniques to solve the hardest puzzles, so we did not create any puzzles that required those techniques.

Because the solver does not possess the most advanced solving techniques, it is incapable of producing the truly ridiculous puzzles.

- Our algorithm can only remove a maximum of 56 tiles.

Because our approach requires our algorithm to consistently solve the puzzle as it removes tiles, the algorithm tends to get overloaded when told to remove more than 56 tiles. It may sometimes succeed but it is not guaranteed.

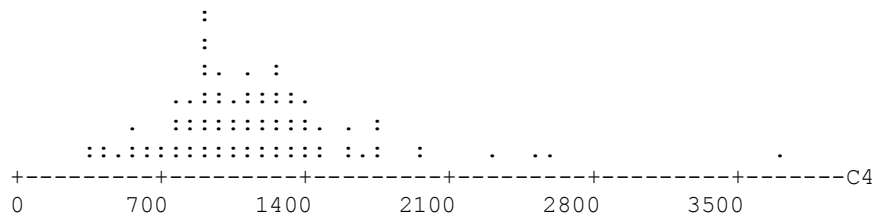
- Our algorithm does not ever make any guesses.

Using our approach, our algorithm can solve every tile in the puzzle except a few, but have to back up simply because it reaches a dead end. While we did not want to create puzzles that forced a solver to guess, perhaps enabling this feature so near the end would have been a preferred feature. It might also allow the solver to generate a puzzle beyond it's logical capacity.

Section 5: Model Testing and Sensitivity Analysis

5.1 Model Testing

We tested our difficulty equation by generating and solving Sudoku puzzles to find ones which used a variety of the techniques presented earlier. This allowed us to rework our equation and set the parameters for what score constitutes an easy, medium, hard, and harder puzzle.



* When instructed to remove 40 tiles.

By default, our algorithm finds the easiest puzzle which meets the remove tile requirements. When we want the difficulty increased, we can change the minimum difficulty variable and force it to generate harder puzzles.

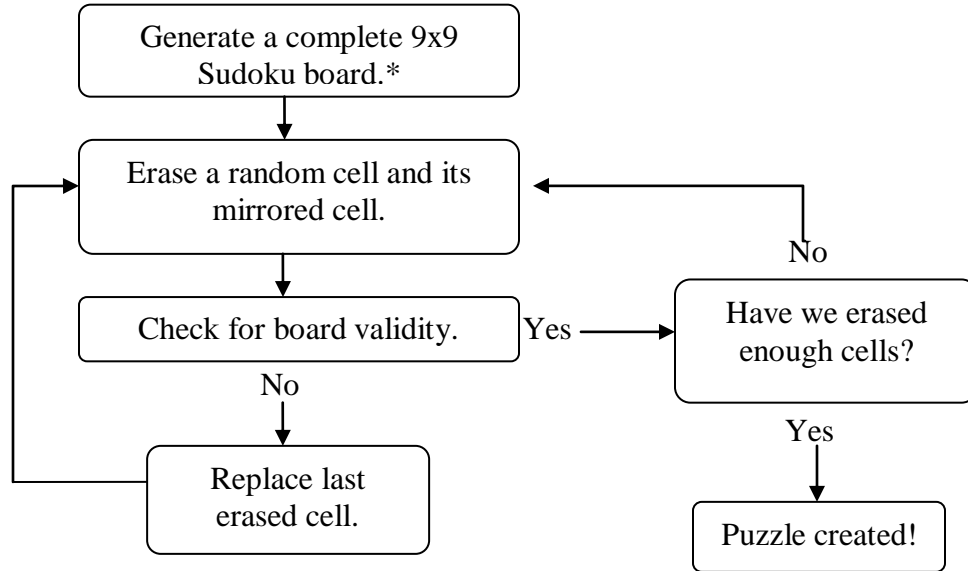
5.2 Sensitivity Analysis

By assigning a specific seed to the random number generator, we were able to run tests on the same Sudoku puzzle multiple times. We tested the same Sudoku puzzle with different restraints on the solving algorithm, by disabling a single technique in order to see how it affects its rating. These are some results we observed:

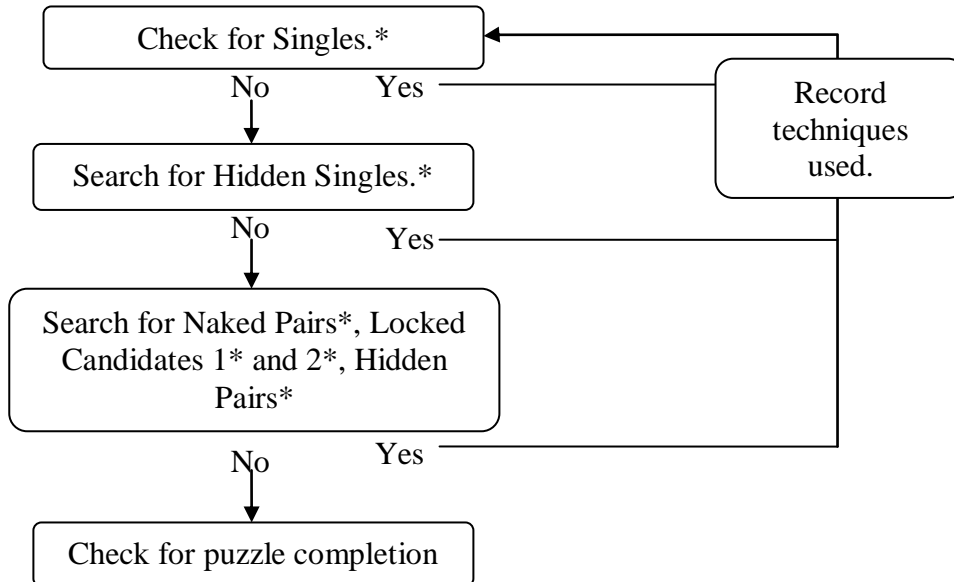
- Eliminating **singles** increased the difficulty rating from 584 to 786, it relied on higher techniques.
- Eliminating **hidden singles** increased the difficulty rating from 584 to 681, since it relied on more advanced methods such as locked candidates to force singles.
- Eliminating **naked pairs** increased the difficulty rating from 584 to 614, and seemed to rely more on locked candidates to find what the naked pairs would have.
- Eliminating **locked candidates** failed to generate a harder puzzle since it did not have the more advanced techniques.
- Eliminating **hidden pairs** did not change the rating since there was not a hidden pair found!

Section 6: Algorithms

Algorithm to Create a Sudoku Puzzle



Algorithm to Rate a Sudoku Puzzle



* See Appendix A-G for more detailed explanations.

Section 7: Conclusion

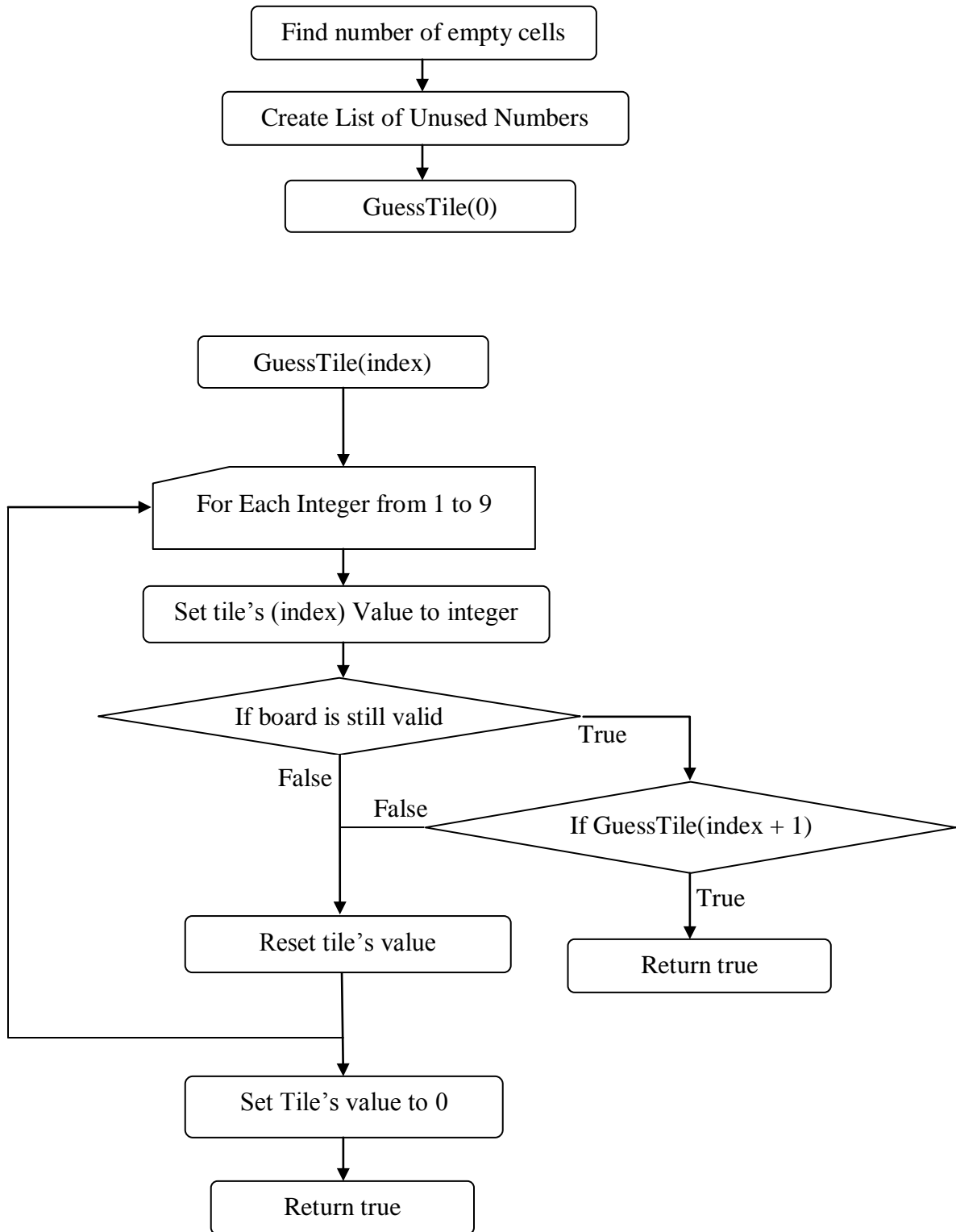
Finally, our approach was very reasonable and effective. We were able to develop an algorithm that generated valid Sudokus to be solved by individuals using their logic and reason. We were unable to incorporate many of the rare, highly advanced, techniques in order to create the most challenging puzzles, but given the time, our algorithm could have been modified to create such puzzles. A strong point of the algorithm is that it is capable of generating a puzzle based solely on difficulty rather than determining the difficulty after the puzzle has already been created. Furthermore, although we did not incorporate asymmetrical puzzles into our analysis, the algorithm is capable of generating asymmetrical puzzles. This compromises the ongoing feud of what type of symmetry verifies a Sudoku as being valid. Overall, our approach may not be the most elaborate, or even the most efficient, process for creating such puzzles. However, we find it to be well developed and appealing to use.

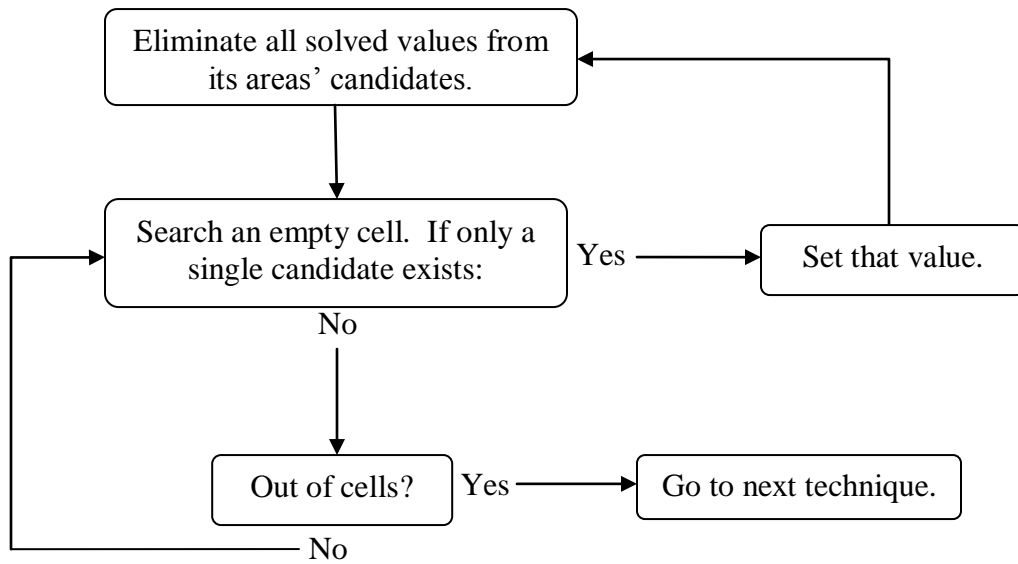
References

- http://www.associatedcontent.com/article/526113/make_your_own_sudoku_puzzles_for_free.html
- <http://angusj.com/sudoku/hints.php>
- <http://www.sudocue.net/guide.php>
- http://www.sudokuessentials.com/sudoku_tips.html
- <http://www.geometer.org/mathcircles/sudoku.pdf>

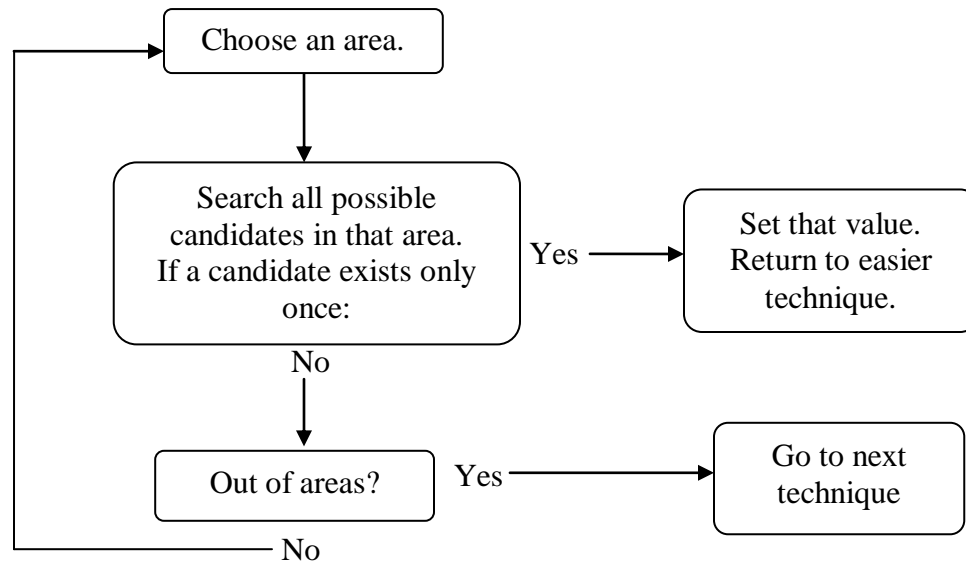
Appendix A:

Solving Algorithm Using Brute Force

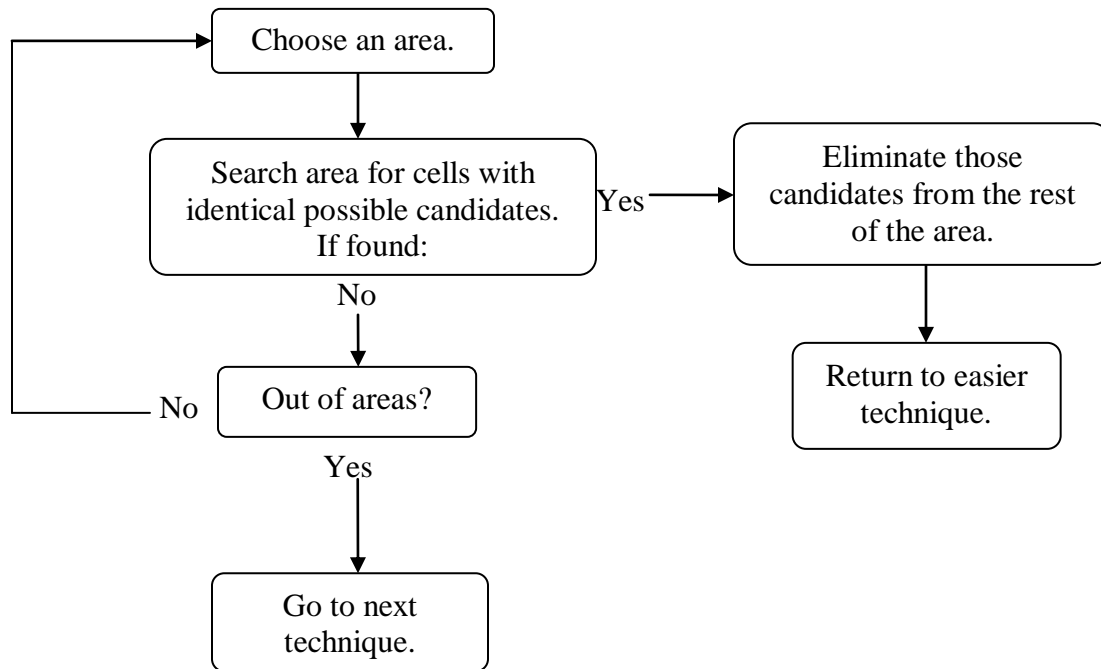


Appendix B:**Algorithm for Solving Singles**

The simplest of Sudoku strategies is to search for any cell in which there is only one possible candidate. If there is only a single candidate in a cell, set that value, eliminate it as a possible candidate in its areas, and search the next cell. Finding one single often leads to creating another. Once all of the single possibilities have been found, move on to a new technique.

Appendix C:**Algorithm for Solving Hidden Singles**

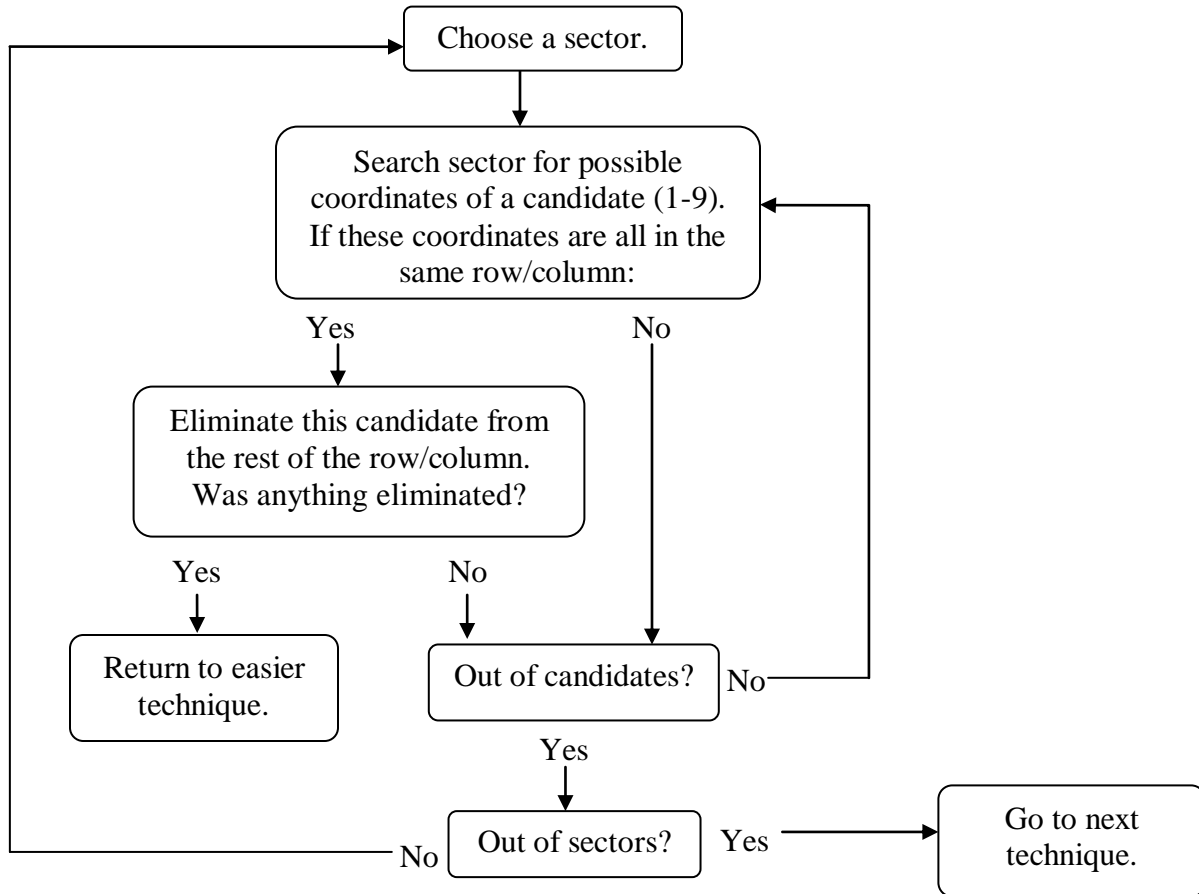
Hidden singles are very similar to singles. In order to find a hidden single, look at each individual area. If there is a candidate that exists only once, that must be its position, as the rule of Sudoku says every integer must be used once and only once per area. If you find a hidden single, set the value, eliminate it as a possible candidate in its areas, and return to an easier technique. A single may have been exposed in the process. If you do not find a hidden single, try another area. When all of the areas have been checked, move on to a more advanced technique.

Appendix D:**Algorithm for Finding Naked Pairs**

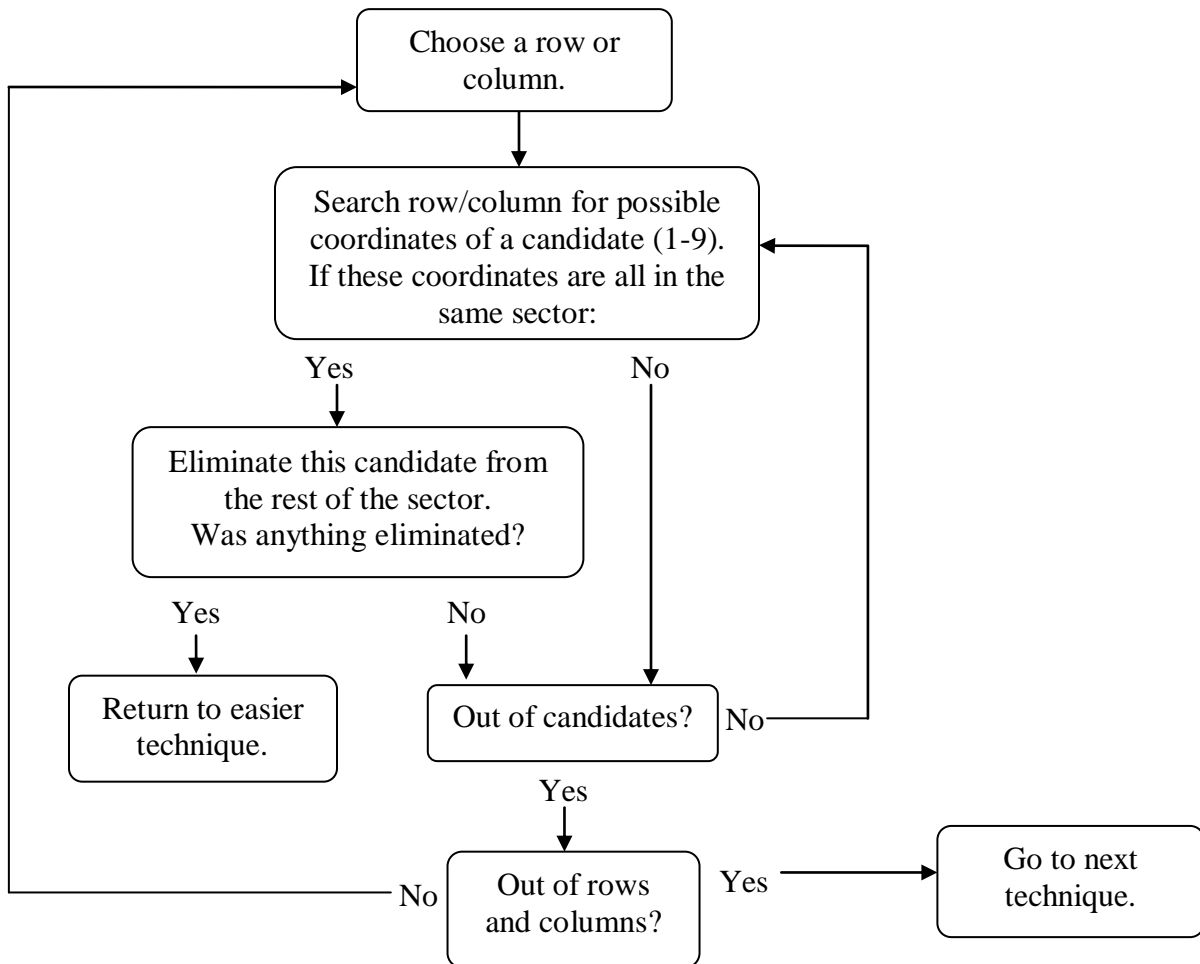
Many Sudoku puzzles require more than a basic technique to solve. Finding a naked pair does not allow you to directly solve a cell, but it may lead to a single or hidden single being exposed. To find a naked pair, search an area for two cells with identical possible candidates. If found, you can eliminate those candidates from the rest of the area and go back to easier methods. Otherwise, search the next area until you run out of areas.

Appendix E:

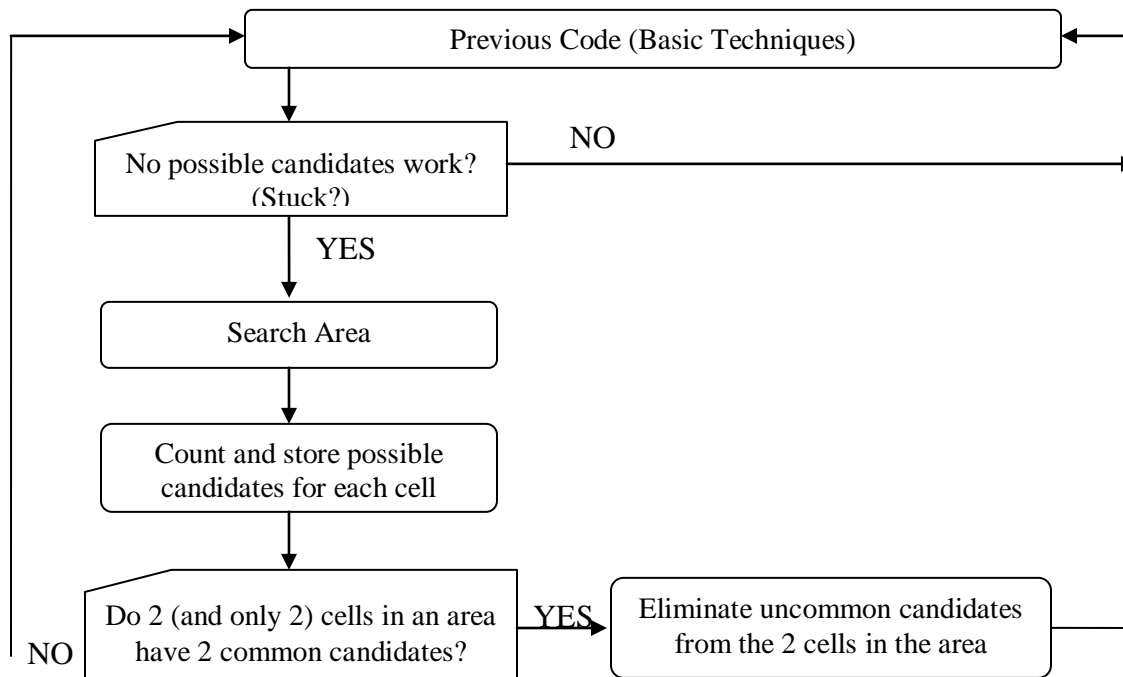
Algorithm for Finding Locked Candidate 1



If no other technique has worked so far, searching for a locked candidate is the next step to try. To find a locked candidate 1, search a sector for any candidate that is only present in one row or column. If you find one, then that candidate can be eliminated from the rest of that row/column. If you were able to eliminate something, try going back to the basic techniques. If not, keep searching the sector for all candidates. When you are out of candidates, try the next sector.

Appendix F:**Algorithm for Finding Locked Candidate 2**

Finding a locked candidate 2 is very similar to finding a locked candidate 1. Search a row or column for any candidate that is only present in one sector. If you find one, then that candidate can be eliminated from the rest of the sector. If you were able to eliminate something, try going back to the basic techniques. If not, keep searching the row/column for all candidates. When you are out of candidates, try the next row or column.

Appendix G:**Algorithm for Finding Hidden Pairs**

When the basic techniques of solving the puzzle do not suffice in progressing through the puzzle, then a slightly more advanced technique is needed. This hidden pairs algorithm finds exactly two cells in the same area of a Sudoku puzzle that share two common possible candidates for the respective cells. If two such cells are found, then the uncommon values in the cells may be safely eliminated. The program then returns to the basic solving techniques, and proceeds to solve the puzzle.

This is a technique that is needed on the intermediate level Sudoku puzzles. So, if a solver must initiate this method, then the puzzle is rated more difficult. That difficulty rating is based on how many times this technique must be used, along with the number of advanced techniques that must be used.