Generating Sudoku Puzzles as an Inverse Problem

Team #2306

February 18, 2008

Abstract

This paper examines the generation of Sudoku puzzles as an inverse problem, with the intention of engineering Sudoku puzzles with desired properties. We examine a number of methods that are commonly used to solve Sudoku puzzles, and then construct methods to invert each. Then, starting with a completed Sudoku board, we apply these inverse methods to construct a puzzle with a small set of clues. This is accomplished with a modified breadth-first search which ensures that our puzzle will be uniquely solvable by the methods we have described. Furthermore, this search algorithm utilizes heuristics both to reduce runtime, and to construct a Sudoku puzzle with special features. In particular we would be able to generate a puzzle of a desired difficulty.

Contents

1 Introduction

Sudoku is a number-placement puzzle that has become popular within the last decade. The game accommodates casual players looking to relax and also serious players looking to challenge their logic skills. Sudoku is a logic puzzle that is played on a 9x9 grid that has nine 3x3 subgrids called blocks. The goal is to fill the grid with the numbers 1 through 9 such that every column, row, and block contains every number exactly once. At the start of the game, the sudoku board is partially filled with a set of clues. A valid clue set will lead to only one possible solved board[5]. An example puzzle and solution are shown below in Table 1.

Due to the craze over Sudoku, many enthusiasts have attempted to figure out how Sudoku puzzles work. That is, they search for underlying structure within the puzzle. The most common products spawned by these enthusiasts are puzzle generators, puzzle solvers, and difficulty rankers. These programs litter the internet and most fall into one of a few categories[1]:

- Most puzzle generators use random assignment of numbers to cells (starting from a blank sudoku board) until the puzzle can be solved to produce one unique board.
- Puzzle solvers typically use one of two methods: backtracking (or brute force), and logical deduction similar to that used by a human.
- Most difficulty ranking programs assign weights based on the difficulty of the logical methods needed to solve the puzzle. [Reference sudoku programmer website]

There are other effective algorithms enthusiasts have developed. As an example, a puzzle solver has been created that employs probability estimates[9] and another that uses genetic algorithms[2]. These approaches were not expected to be successful due to the logical nature of how humans solve the puzzles.

1.1 Mathematics in Sudoku Puzzles

Apart from the study of Sudoku generation, puzzle solving, and difficulty ranking, very little headway has been made in studying Sudoku mathematically. The following two examples are essentially the only existing mathematical topics:

1.1.1 Minimal number of starting sets

There is the open ended question as to what the minimal number of clues required to produce a unique sudoku board. It is conjectured the minimal number is 17 as people have found such examples but for now have found no valid sudoku puzzles with fewer[5].

1.1.2 Symmetric Equivalence of Sudoku Boards

It is possible to talk about an equivalence relationship between valid sudoku boards in the following way: Two boards are symmetrically equivalent if it is possible to modify one board by

(a)

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

(b)

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

Table 1: Example of a Sudoku puzzle: (left) the beginning board layout, (right) solution to the Sudoku puzzle

1. permutations of rows and columns within blocks,
2. permutations of block rows and columns, and
3. permutations of the symbols used in the board (for example, if 1 and 2 are exchanged everywhere),

so that it matches the other[5]. This equivalence relationship partitions all possible sudoku boards into equivalence classes.

2 Motivation

The most lacking of the three algorithm types given in Section 1 (puzzle generating, puzzle solving, and puzzle difficulty rating) are the generating algorithms. There are two patterns we see in the most common and effective algorithms: they involve random processes of adding or subtracting random numbers to the sudoku board[3]; and all, except one algorithm, generated the sudoku puzzle in the same direction players solve the puzzles. The only way sudoku puzzles are generated with a desired difficulty by the current algorithms is by:

1. Creating a repository of randomly generated puzzles
2. Analyzing the difficulty with a difficulty rating algorithm
3. Retaining the puzzles with the desired difficulty

Even though each puzzle is generated and analyzed within a few seconds (in some of the faster algorithms), thousands have to be created to increase the chance of creating a difficult puzzle. Thus, we seek to break away from these common methods.

Our goal in this paper is to develop a sudoku generation algorithm that has a non-random element of control of the final state of a sudoku boards. This will show the viability of using the inverse method to not only create boards with given difficulties, but also to study the structure of sudoku boards- and hence the mathematical side of sudoku boards. After analyzing the existing generator algorithms, our algorithm should:

- not rely on random additions or subtractions of elements,
- not look at constructing the Sudoku puzzle from scratch. Instead, we should generate the our puzzles using inverse methods.
- use the existing methods the determine difficulty. To do this, we will have to assume players solve a Sudoku by attempting the most intuitively obvious (ie. simplest) move first. To solve harder puzzles requires people to know harder techniques, and be able to recognize where to use them. Then it is reasonable to rate a puzzle by the types of technique required, and how often.

3 General Terms

Solvable A sudoku puzzle is solvable if there is only one way to fill in a sudoku board to make it valid.

Cell A cell is a position on the Sudoku board. Cells contain cell values and pencil-marks (denoted by $(n, \{a, b, ..., j\})$ with n the value of the cell and $\{a, b, ..., j\}$ the set of pencil-marks).

Pencil-Marks Pencil-marks are values placed in a cell that serve to represent the possible choices for the value of a cell.

Cell Value The only value a cell can have in order to produce a solvable sudoku board.

Neighboring Cell Two cells are neighboring if the two cells are contained within the same row, same column or same box.

Sector The sector of a cell is the set of all neighboring cells that are contained within either a row, column, or cell.

4 The Inverse Sudoku Problem

To approach Sudoku as an inverse problem, it is necessary to invert the methods that are used to solve Sudoku puzzles. Before this can be done, some discussion of the forward methods is an obvious prerequisite. Here, we construct a more robust mathematical model of a Sudoku board which provides a framework on which we can build the forward and inverse methods. Then, we define some forward methods that are commonly used, and deduce inverse methods for each one.

4.1 Some Notation

In this section, we will treat a Sudoku board as a (9×9) matrix

$$S = (s_{ij}), s_{ij} = (v_{ij}, M_{ij})$$

where the entries are ordered pairs of *values*

$$v_{ij} \in \{\emptyset, 1, \ldots, 9\}$$

and *pencil-marks*

$$M_{ij} \subset \{1, \ldots, 9\}.$$

Note that we allow $\emptyset$ as a value and not zero – we will refer to this as the "null" value, since nothing will be shown there in a Sudoku puzzle. Also, we require that a cell has a non-null value if and only if the set of pencil-marks is nonempty. It is easy to verify in the following section that all methods preserve this.

We say that a Sudoku board is *solved* when for each entry s_{ij},

$$v_{ij} \neq \emptyset \text{ and } M_{ij} = \emptyset,$$

and the typical row, column and block requirements are fulfilled. Each entry in our matrix corresponds to a *cell* in the Sudoku board, and to each cell ij we associate a *neighborhood*

$$\mathcal{N}_{ij} = \mathcal{R}_{ij} \cup \mathcal{C}_{ij} \cup \mathcal{B}_{ij}$$

the union of the cells in the i-th row, in the j-th column, and the block containing ij. If a cell ij is contained in the neighborhood of a cell kl, these cells are called neighbors (a symmetric, reflexive, nontransitive relation). In general, we will refer to rows, columns, and blocks as *sectors* when a method is general enough to allow this, and sectors will be denoted with an $\mathcal{S}$. Occasionally, subscripts will be left off of variables representing sectors, in which case the reader should be able to infer what sort of sector is being considered by the context (if required).

4.2 Some Sudoku Solution Methods with Inverses

The game of sudoku is played by using logic to deduce the values of all the cells. Well experienced players have categorized many logical steps into methods. From the perspective of the player, we call the methods forward methods. The following forward methods are all very well known, however, the authors have found no reference to inverting these methods in the literature[7]. Fortunately, when the methods are reduced to a single operation, their descriptions become quite simple. So, for each method, we provide a concise verbal description followed by a rigorous mathematical definition: we provide a list of preconditions required to perform a simple action, with the exception of the single candidate method, this action will be to remove a pencil-mark from a cell. Then, the inverse function is defined similarly, where the action will be to add a pencil-mark to a cell.

With the exception of the single candidate method, the inverse method is not proper – in general, one can utilize an inverse method to remove too much information from the board. However, the inverse method may be applied to undo the forward method in every case. It important to note that these are not functions – at any given point, it may be possible to execute a method on the board in a large number of ways – so, it is no surprise that we are unable to properly invert these methods.

4.2.1 Single Candidate

If a cell has a single pencil-mark, we erase the mark and set the value of the cell to this mark.

Forward If

$$M_{ij} = \{a\},$$

set

$$\begin{aligned} M_{ij} &\leftarrow \emptyset, \\ v_{ij} &\leftarrow a. \end{aligned}$$

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

⇕

Inverse If

$$\begin{aligned} M_{ij} &= \emptyset, \\ v_{ij} &= a, \end{aligned}$$

set

$$\begin{aligned} M_{ij} &\leftarrow \{a\}, \\ v_{ij} &\leftarrow \emptyset. \end{aligned}$$

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

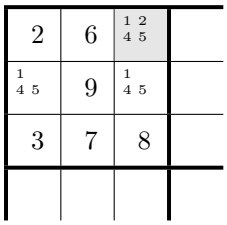
Every other method can be reduced to removing pencil-marks *only*, so that in the end, the only method to reveal a value on the board is this method (to say nothing about the steps to reduce to a single candidate).

4.2.2 Exclusion

If a cell has been assigned a value, we remove that value from the pencil-marks of each of its neighbors.

Forward

$$\begin{aligned} v_{ij} &= a, \\ kl &\in \mathcal{N}_{ij}, \\ a &\in M_{kl}, \end{aligned}$$

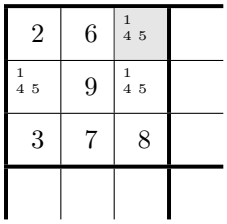


set

$$M_{kl} \leftarrow M_{kl} \setminus \{a\}$$

Inverse

$$\begin{aligned} v_{ij} &= a, \\ kl &\in \mathcal{N}_{ij}, \\ M_{kl} &\neq \emptyset, \end{aligned}$$



set

$$M_{kl} \leftarrow M_{kl} \cup \{a\}$$

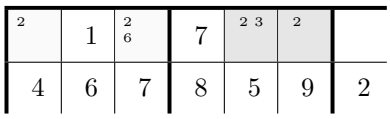
Repeated applications of this method give rise to what is commonly known as "Single Position", and is generally considered the most natural method of solution.

4.2.3 Block Intersection

If a mark appears in a row only where that row coincides with a particular block, we can deduce that this mark can be removed from any other positions on that block. Similarly, if the marks appear in a block only where the block coincides with a row, we can remove the mark from other positions in the row. The above also holds for columns in place of rows.

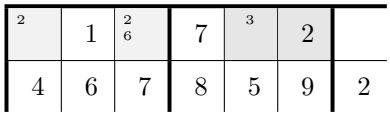
Forward If

$$\begin{aligned} mn &\in \mathcal{B}_{ij} \setminus \mathcal{R}_{ij} \\ a &\in M_{ij} \cap M_{mn} \\ kl &\in \mathcal{R}_{ij} \setminus \mathcal{B}_{ij} \Rightarrow a \notin M_{kl} \end{aligned}$$



then, set

$$M_{mn} \leftarrow M_{mn} \setminus \{a\}.$$



Inverse If

$$\begin{aligned} mn &\in \mathcal{B}_{ij} \setminus \mathcal{R}_{ij}, \\ a &\in M_{ij} \setminus M_{mn}, \\ kl &\in \mathcal{R}_{ij} \setminus \mathcal{B}_{ij} \Rightarrow a \notin M_{kl}, \\ M_{mn} &\neq \emptyset \end{aligned}$$

then, set

$$M_{mn} \leftarrow M_{mn} \cup \{a\}.$$

We may reverse the set exclusions above, $(\mathcal{R}_{ij} \setminus \mathcal{B}_{ij}) \leftrightarrow (\mathcal{B}_{ij} \setminus \mathcal{R}_{ij})$, and we can exchange the row $\mathcal{R}_{ij}$ for a column $\mathcal{C}_{ij}$.

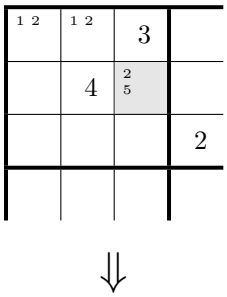
4.2.4 Covering Set

Suppose that a set of n cells is contained within a sector, and they have exactly n distinct marks between them. Then, we deduce that these marks may be removed from anywhere else in the sector. Here, we let S denote some sector on the board.

This is frequently referred to as the "Naked Subset" method. Another frequently used method is the "Hidden Subset" method, which is easily proved to be equivalent to this.

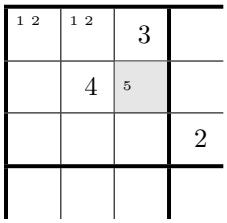
Forward If

$$\begin{aligned} U &\subset \mathcal{S}, \\ M &= \bigcup_{ij \in U} M_{ij}, \\ |M| &= |U|, \\ kl &\in \mathcal{S} \setminus U, \\ a &\in M \cap M_{kl}, \end{aligned}$$



then, we set

$$M_{kl} \leftarrow M_{kl} \setminus \{a\}.$$



Inverse If

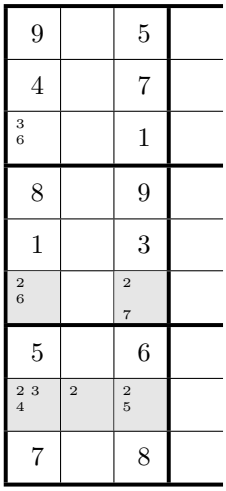
$$\begin{aligned} U &\subset \mathcal{S}, \\ M &= \bigcup_{ij \in U} M_{ij}, \\ |M| &= |U|, \\ kl &\in \mathcal{S} \setminus U, \\ a &\in M, \\ a &\notin M_{kl} \neq \emptyset \end{aligned}$$

then, we set

$$M_{kl} \quad \leftarrow \quad M_{kl} \cup \{a\}.$$

4.2.5 X-Wing

This is a somewhat more complicated method. Suppose there are four cells at the corners of a square, and that each of these cells contains a certain pencil-mark. These four cells will lie in precisely two rows and precisely two columns – then assume that the only positions in these rows where this pencil-mark occurs is at the intersection with the columns (our selected corners). Then, we deduce that this mark may be removed from any other position in those columns. Per the norm, rows and columns here may be switched.

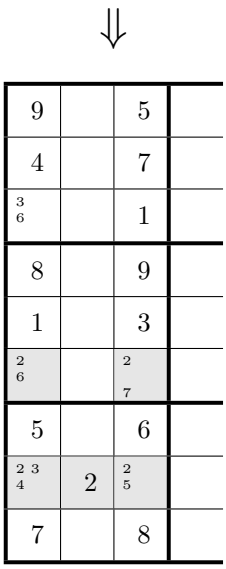


Forward If

$$\begin{aligned} ij, il, kj, kl &\in (\mathcal{R}_{ij} \cup \mathcal{R}_{kl}) \cap (\mathcal{C}_{ij} \cup \mathcal{C}_{kl}), \\ a &\in M_{ij} \cap M_{il} \cap M_{kj} \cap M_{kl}, \\ xy &\in (\mathcal{R}_{ij} \cup \mathcal{R}_{ij}) \setminus (\mathcal{C}_{ij} \cup \mathcal{C}_{kl}) \Rightarrow a \notin M_{xy}, \\ mn &\in \mathcal{C}_{kl} \setminus (\mathcal{R}_{ij} \cup \mathcal{R}_{kl}), \\ a &\in M_{mn}, \end{aligned}$$

then, set

$$M_{mn} \leftarrow M_{mn} \setminus \{a\}.$$



Inverse If

$$\begin{aligned} ij, il, kj, kl &\in (\mathcal{R}_{ij} \cup \mathcal{R}_{kl}) \cap (\mathcal{C}_{ij} \cup \mathcal{C}_{kl}), \\ a &\in M_{ij} \cap M_{il} \cap M_{kj} \cap M_{kl}, \\ xy &\in (\mathcal{R}_{ij} \cup \mathcal{R}_{ij}) \setminus (\mathcal{C}_{ij} \cup \mathcal{C}_{kl}) \Rightarrow a \notin M_{xy}, \\ mn &\in \mathcal{C}_{kl} \setminus (\mathcal{R}_{ij} \cup \mathcal{R}_{kl}), \\ a &\notin M_{mn} \neq \emptyset \end{aligned}$$

then, set

$$M_{mn} \leftarrow M_{mn} \cap \{a\}.$$

5 Backtracking the Inverse Problem

As stated, our goal is to engineer a Sudoku puzzle which can be solved with a certain set of methods, and maximizes the difficulty based on a ranking of these methods. We will see that inverting the solution methods does not always result in a solvable puzzle. Because of this, a classical backtracking method becomes computationally intractable – since we would have to check for a unique solution at every node in the search tree.

To reduce the number of times that we attempt to fully solve a board, the natural choice is a breadth-first search. Then, each time we consider a puzzle, we need only check forward solutions once. However, there are a potential of 729 possible pencil markings in a board, so the search space is approximately the powerset of these markings. The breadth-first search certainly reduces the memory bound, but it remains on the order of $\binom{729}{365}$ states – a terrifying 10^{220} entries in memory. Obviously, that idea won't get us very far.

Therefore, the problem needs a better tuned algorithm. For our purpose, a combination of depth-first and breadth-first searching seems to do the job. Of particular note, this method allows one to prune the search space using *heuristic* functions. We will not define many heuristic functions here; the power of this search method is that it allows a very fine level of control over the puzzles found, merely by variation of the heuristic function.

5.1 Constrained Breadth Searching

Each solution method we will consider has the effect of adding or removing a single pencil mark. Therefore, if we consider each set of pencil marks to be a point in the search space, there is a well-defined partial ordering on these points. A *parent* of a certain point has precisely one fewer pencil mark than that point, and a *child* has precisely one more pencil mark.

The constrained breadth search uses a heuristic at every step to choose a single child. This heuristic may be random, some function of the difficulty required to come back from the child, or perhaps the number of ways to come back. We will soon show that using a heuristic in this manner will vastly reduce the number of points that consider; keeping in mind that we need only find one puzzle in this massive space, frequently rejecting a huge number of points is a necessity.

Suppose we are looking at point in space, for which applying any known method will yield a unique solution. Then, if we apply any known method to reach a parent of this point, that parent will have the same property. We will call these *good parents*, and define *good children* similarly so that all good children have unique solutions. During the search, we will consider only good children of a point whose grandparents are also good parents of that point. This ensures that if one uses only known methods to solve the resultant puzzle, they will only see board states that this search method has visited – and therefore, the paths that the search method considers in arriving at the final puzzle is precisely those paths that the user might take.

5.2 Formal Discussion of the Constrained Breadth Search

We have outlined the constrained breadth search, and we have now shown that a set of methods are valid (that is, they will not yield distinct or incorrect solutions to a uniquely

solvable puzzle), and invertible. For the rest of this section, we will consider these methods to be "all known methods", even though this is obviously false. Now, we formally define the constrained breadth search, and prove that the algorithm terminates at a point which has a unique solution, and expected runtime is essentially constant.

First, we note that the height of the search tree is bounded by 729, the number of possible marks in a sudoku board. Clearly, a given point may have at most 729 parents and 729 children, furthermore, the total number of children of all parents of a point will be at most $\binom{729}{2} = 265356$. As the number of pencil-marks increases, these bounds will decrease. At a given point, let the candidate board have n pencil-marks. Then, any one of n of these pencil-marks may be removed to arrive at a parent, and $729 - n$ pencil-marks may be added to arrive at a child.

Define a *good sibling* of a point to be a good child of a good parent of that point (and note that this is distinct from a good parent of a good child – those may not have unique solutions in general). The number of good siblings of a point with n pencil-marks, then, is bounded by $(729 - n)n$, or $729n - n^2$ which is bounded from above by 132860.

Let Q be a point in the search space, and $P = \mathcal{M}(Q)$ be the set of parents of Q such that each point in P is reachable from Q by a single application of a known forward method. Also, let $C = \mathcal{M}^{-1}(Q)$ be the set of children of Q such that each point in C is reachable from Q by a single application of a known inverse method.

With this notation, we are able to define our algorithm, and prove our claims above.

```
define search(Q, GoodParents, GoodSiblings):
    #Preconditions: * All elements in GoodParents are
    #                 good parents of Q
    #               * All elements in GoodSiblings are
    #                 good siblings of Q
    #               * Q is uniquely solvable by known
    #                 methods
    Children := {}
    GoodChildren := {}
    for each inverse method W:
        Children := union(Children, {children of Q via W})

    for each forward method M:
        for each Child in Children:
            if {parents of Child via M} subset GoodParents):
                GoodChildren := union(GoodChildren, {Child})

    if GoodChildren = {}:
        return Q

    MaximalHeuristic := -Infinity
    for Child in GoodChildren:
        if Heuristic(Child) > MaximalHeuristic:
            BestChild := Child
            MaximalHeuristic := Heuristic(BestChild)
```

```
    NewParents := {}
    for each forward method M:
       NewParents := union({parents of BestChild via M},
                           NewParents)

    return search(BestChild, NewParents, GoodChildren)

define start_search(Board):
    Children := {children of Board via InverseSingleCandidate}
    Child := RandomElement{Children}
    return search(Child,{Board},Children)
```

Note that when the search is initiated, we start with a board which is fully determined but for a single cell, and uniquely solvable by the single candidate method. The only children of this initial point (all of which will be good) will be uniquely solvable by either the single candidate method, perhaps following an application of the elimination method. Also, the only siblings of the point will be other boards which are solvable by the single candidate method. Hence, the preconditions are met when the `search` method is called again. This ends the base case of our induction proof.

For the induction step, assume that we have been passed a point which is uniquely solvable by known methods, and the good parents and good siblings of that point. By their very definition, it is clear only good children will be considered, and the good parents of the chosen child will be passed along. When the algorithm reaches a point which has no good children, it is still the good child of a point whose solution is unique. This completes our proof that the result will have a unique solution.

Furthermore, we note that backtracking is never needed in this method. Therefore, the search algorithm will be called at most 729 times (in practice, less than half of that) every time incrementing n, and the number of times each method will be called is bounded by a constant. By cacheing computations intelligently, each of the known methods described above can be performed in amortized constant time. Therefore, this algorithm will run in an essentially constant amount of time.

5.3 Engineering A Sudoku Puzzle

In the previous section, we proved that our algorithm runs in an essentially constant amount of time. However, this leaves something to be desired – certainly, other methods run quickly, and create puzzles of a variety of difficulty with much simpler methods. Above, we mentioned a heuristic function which is used to select certain children, but mentioned nothing about its capabilities. At the present, little is known about how a heuristic function can be used to control the resulting puzzle, but the above search algorithm gives a strong indication of some possibilities.

Of primary interest is the runtime. While this search algorithm will run in a roughly constant amount of time, that constant is very large. So, one natural heuristic is

$$t(\texttt{Child}) = -\left|\{\texttt{good parents of Child}\}\right|.$$

This ranks children with many good parents poorly, so there will be a natural tendency for this heuristic to minimize the freedom that one has in solving the resulting puzzle. It is often claimed by puzzle masters that a hand-crafted Sudoku puzzle may force a solver to take certain moves. By utilizing the t-heuristic, this algorithm can be made to prefer such puzzles.

Earlier, we promised to provide a certain level control over the difficulty of a puzzle. For this, we first need a metric over the various known methods. For example, one can rank the methods in section 4 by their order of appearance in that section. Then, we can define a heuristic

$$d(\texttt{Child}) = \min\{\texttt{rank of forward methods applicable to Child}\}.$$

Here, the heuristic attempts to maximize difficulty at every step. Generally, this greedy approach will not maximize difficulty globally, and it is possible that as more difficult puzzles are desired, one will not always find success at the first attempt. However, the search method is easily converted into a kind of backtracking search which does not terminate until a puzzle with a certain difficulty is found.

Similarly, one might wish to attack the open problem to find a uniquely solvable puzzle with only 16 clues. Here, one would rank the single candidate method above all others, so that clues are removed as often as possible. Or, one might want to aim for multiple goals – there is no requirement that one uses a single heuristic throughout the search – one could first apply the t-heuristic to prune the search space, and then move to a different mode until the search space grows, and then return to the t-heuristic. Through such variation of heuristics, this algorithm provides a large amount of flexibility in the design of Sudoku puzzles.

6 Conclusion

As shown in section **??**, there exist Sudoku generation algorithms which provide great control over the resulting puzzle. Particularly, it is possible to generate puzzles of desired difficulty. Furthermore, we can systematically search for solvable Sudoku puzzles with fewer than 17 entries. These are both familiar problems for those interested in the mathematics of Sudoku.

However, these are not the only interesting questions one can asked about Sudoku. Such questions might include: given a certain method, how many times can we force a logical solver to use that method in the solution of a puzzle? Or, in the set of puzzles which uniquely determine a particular solution, how often are certain methods required or forbidden? In summary, this puzzle generation method may provide a useful tool to further analyze the structure of solvable Sudoku boards since we are no longer restricted to brute-force analysis of random puzzles. It is our feeling that this topic deserves deeper scrutiny.

In general, the study of inverse problems has brought fresh insight to many topics. Often, an inverse problem may seem more complicated at first glance, but lead to interesting simplification of the corresponding forward problem. The authors have not seen any previous study of the inverse Sudoku problem, which suggests that there could be benefit to looking at the problem in this way. As we saw in section 4, the inverse methods which we have examined seem to be exactly as simple (or complicated) as their corresponding forward methods, which may provide interesting insight to Sudoku.

References

[1] Sudoku Programmers. Website. Retrieved Feb. 16, 2008. www.setbb.com/phpbb/index.php?mforum=sudoku.

[2] Mantere, Timo; Koljonen, Janne. 2007. Solving, rating and generating Sudoku puzzles wth GA. *Evolutionary Computation*, CEC 2007, 1382-1389.

[3] Knuth, Donald E. 2000. Dancing Links. *Millenial Perspectives in Computer Science*, 187–214.

[4] Knuth, Donald E. 1997. *The Art Of Computer Programming* Vol 1. 3rd ed. Boston: Addison-Wesley.

[5] Lee, Wei-Meng. 2006. *Programming Sudoku*. New York: Springer-Verlag.

[6] Mckay, B. D. 1978. Computing automorphisms and canonical labellings of graphs. *Combinatorial Mathematics: Lecture Notes in Mathematics*, 686, 223-232. Springer-Verlag.

[7] Sudoku Solver Techniques. Website. Retrieved Feb. 16, 2008. www.sudoku.net/tech.php.

[8] Seress, Akos. 2003. *Permutation Group Algorithms*. Cambridge University Press.

[9] Turner, Richard. 2007. Generalized Belief Propagation. *Advanced Probabilistic Technique Workshop*. Lecture notes. Retrieved Feb 17, 2008. www.gatsby.ucl.ac.uk/ turner/workshop/workshop.html.