The Pennsylvania State University
The Graduate School

AN IN DEPTH ANALYSIS OF SUDOKU WITH FOCUS IN INTEGER AND CONSTRAINT PROGRAMMING

A Thesis in
Industrial Engineering and Operations Research
by
Michael Thein

Submitted in Partial Fulfillment
of the Requirements
for the Degree of

Master of Science

May 2014

The thesis of Michael Thein was reviewed and approved* by the following:

Paul Griffin
Professor of Industrial and Manufacturing Engineering
Co-Advisor

Christopher Griffin
Research Associate and Asst. Professor of Mathematics
Co-Advisor

M. Jeya Chandra
Professor of Professor of Industrial and Manufacturing Engineering
Graduate Program Coordinator

*Signatures are on file in the Graduate School.

Abstract

We analyze sudoku in detail. We study sudoku as it pertains to computational complexity, graph coloring, and various programming methods that can be used to solve sudoku. We construct integer and constraint programs to solve sudoku problems. We conduct empirical experiments using these programs. Our results exhibit constant solve times for our integer program and varied solve times for our constraint program depending on difficulty. For easier sudokus, constraint programming performs significantly faster,but as difficulty increases, our integer program exhibited faster solve times. Additionally, we applied a heuristic. When combined with a heuristic, the constraint program solve times were significantly improved. The solve times were drastically better than those of our integer program for easy, medium, and hard difficulty, and were only slightly worse for evil difficulty. We tested to see how solve times reduced when more information is provided to our constraint solver. We observe an exponential decay function in solve times as more cells were filled in. Finally, we present future research that we wish to conduct.

Table of Contents

List of Figures vi

List of Tables vii

Acknowledgments viii

Chapter 1
Summary of Results in Thesis 1

Chapter 2
Literature Review 3
2.1 Sudoku Background . . . 3
2.2 Sudoku Complexity . . . 3
2.3 Integer Programming . . . 4
2.4 Constraint Programming . . . 4
2.5 Coloring/Graphs . . . 5

Chapter 3
IP/CP formulation and Heuristics 6
3.1 Introduction . . . 6
3.2 Integer Programming . . . 6
3.2.1 Integer Program . . . 6
3.3 Constraint Programming . . . 8
3.4 Heuristic . . . 9

Chapter 4
Results of Experimentation 11
4.1 Method . . . 11
4.2 IP/CP Results . . . 11
4.3 Heuristic . . . 12
4.4 Cells remaining reduction . . . 13

Chapter 5
Conclusions and Future Work - Deception Problem/Graph Problem **16**
5.1 Conclusions 16
5.2 Complexity 17
5.3 Desception 17

Appendix A
Integer Programming Python Code **18**

Appendix B
Constraint Program **21**

Appendix C
Heuristic **25**

Bibliography **36**

List of Figures

4.1 A bar chart comparing the run times in solving a sudoku with Integer Programming vs. Constraint Programming. 13
4.2 A bar chart comparing the run times in solving a sudoku with Integer Programming vs. Constraint Programming vs. Constraint Programming with Heuristic. . 15
4.3 Solution time as a function of percent of puzzle solved using the constraint programming. 15

List of Tables

4.1 Constraint programming run times on various sudoku puzzle types. 12
4.2 Integer programming run times on various sudoku puzzle types. 12
4.3 Comparing solve time on easy sudoku with and without a heuristic. 13
4.4 Comparing solve time on medium sudoku with and without a heuristic. 14
4.5 Comparing solve time on hard sudoku with and without a heuristic. 14
4.6 Comparing solve time on evil sudoku with and without a heuristic. 14

Acknowledgments

Gratitude is extended to the Applied Research Laboratory's Eric A. Walker Assistantship for funding my way through graduate school. Additionally much thanks to my advisors Paul Griffin and Christopher Griffin. Thank you to all the educators who have inspired me throughout my life, especially thank you to Christopher Griffin inspiring me to pursue operations research and for all of your guidance throughout these past few years. Finally, thank you to the Penn State's Industrial Engineering and Mathematics departments.

Dedication

For my parents. Thank you for all of your support.

Chapter 1

Summary of Results in Thesis

Recently, sudoku puzzles have become a popular and addictive logic puzzle played by many. The rules of sudoku are simple; fill in a grid so that each number is contained in every row, column, and box only once. Sudokus are universally accessible as they are merely an assortment of numbers scattered throughout a grid; no language barrier prevents one from attempting a sudoku. Yet despite the puzzle's simplicity, millions participate due to the challenge that these puzzles present. Countless books containing sudokus have been published. Some books even teach the reader different techniques for solving sudokus faster. Currently, you would be hard pressed to find a newspaper that doesnt publish a daily sudoku.

The general public are not the only ones interested in sudokus. Recently, many scholars have begun to explore sudoku puzzles to understand various properties about these puzzles. Some areas that have been studied are computational complexity, relation to graph structure, and the sudoku as a graph coloring problem. Computer programs have been written that can be used to solve sudoku puzzles, as well as various heuristics.

In this thesis, we conduct a literature review to gain a better understanding of sudoku puzzles and what properties sudokus have as they pertain to the aforementioned topics. We then formulate an integer program, a constraint program, and a heuristic that solve sudoku puzzles. We test our programs as a means for comparing integer programming and constraint programming solution methods to try to determine the most efficient way of solving sudokus. From out testing, we found that integer programming will solve a sudoku problem at a consistent time regardless of difficulty, whereas constraint programming solve times vary based on difficulty. For easier sudokus, use of a constraint program is superior, but as the difficulty increases, integer programming becomes a more efficient method for solving sudokus. Also, when introducing a heuristic in conjunction with the constraint program, we were able to reduce solve times significantly and only the hardest sudokus would be better solved by an integer program.

We also explore solve times while varying the number of cells filled in for a sudoku to determine what makes certain sudokus easy and others hard. We show empirically that solve times of a

sudoku follow an exponential decay function as more and more cells are filled in. An area that we would like to further explore in this problem is to determine whether specific cells in a sudoku graph have a greater impact on solve time. If such critical cells exist, is there a way to quickly identify them and make use of this information. In addition to exploring the idea of critical cells, another area of future research that we wish to conduct is applying our methods for solving sudoku to a deception problem.

Chapter 2

Literature Review

2.1 Sudoku Background

We begin by giving an introduction to the rules of sudoku. This will provide sufficient motivation for the study of sudoku. According to Peter Gordon and Frank Longo, who have written books on solving sudokus,[1] a sudoku is an $n^2 \times n^2$ grid that is initially filled in with a certain number of cells. The number of cells initially filled in generally typifies the difficulty of the sudoku; the fewer cells, the harder the sudoku. The goal is to fill in the entire grid without repeating any number 1 thru n in each row, column, or $n \times n$ box. Each digit 1 thru n will therefore be used only once. It should be noted that a valid initial grid must contain some form of symmetry. Another key consideration for a sudoku is that the solution must be unique.

2.2 Sudoku Complexity

In Takayuki Yato's Masters Thesis titled Complexity and Completeness of finding another solution and its Application to Puzzles, Yato proved that a sudoku puzzle is NP complete for generalized nxn sudokus. Yato shows that a sudoku can be viewed as an Another Solution Problem (ASP). An ASP problem as it applies to sudoku means that given an initial grid or partially completed grid containing a solution s, find a solution to the sudoku other than s. A problem is considered ASP complete if it is contained in the class FNP. FNP is the function problem extension of an NP class To show the ASP completeness of sudokus, Yato used known results of Latin squares, which were published by Colbourn.[2] The problem of a partial Latin square completion is ASP complete via a reduction to 1 in 4 SAT. Yato then uses proofs from Colbourn's paper to show that a 1 in 4 SAT problem can be reduced to a 1 in 3 SAT problem which is known to be NP complete.[3] Hence, Yato was able to prove that sudoku puzzles are NP complete.

This information is important in the study of sudokus because it plays an important role in how one should approach solving sudokus. Hence various algorithms and heuristics have been

divised with this knowledge. We will now discuss some literature regarding IP and CP as a method of solving sudokus.

2.3 Integer Programming

In Bartlett's article titled An Integer Programming Model for the Sudoku Problem, the authors justify why integer programming is well suited to solving sudoku problems. They demonstrate how a sudoku can be modeled in the form of an IP. [4] We construct our own IP model in the next section which is similar and contains all the same properties as their model. In their paper, they show that a sudoku can also be thought of as a constraint programming problem as well. Hence the objective function for the IP is meaningless and is merely used in order to run the program.

Bartlett also looks into different properties of sudokus. At this point, all research suggests that the minimum number of initial values for a sudoku that has a unique solution is 17. Bartlett shows that an individual element of a 9×9 sudoku gives at most 29 of the decision variables of 729. In general they show that the total number of decision variables effected by an individual cell is bounded above by $3(n-1) + (m-1)^2 + 1$ out of n^3. This means that any particular cell effects at most that many other cells in the grid.

Another angle in which they attempt to understand sudokus is from the perspective of sudoku creation. They discuss how a brute force method can be used to create solutions. They then wish to find faster methods of creating sudokus. To do this, they were able to show how existing sudokus known to have unique solutions can be modified to create new sudoku puzzles that also retain the uniqueness property.

2.4 Constraint Programming

We give a brief background on constraint programming. [5] [6] Constraint programming attempts to solve problems based on satisfying constraints. In theory, constraint programs provide less cognitive burden, are more similar to how we think of problems naturally, and can determine results in similar and sometimes even faster results than a traditional Integer Linear Program.

The theory behind constraint programming is that when solving a problem that involves multiple constraints, in order to determine the solution, all constraints must be satisfied. Hence you tell the constraint program what all of the constraints are and it finds a solution that satisfies all the constraints. An example in which a constraint program is easier and more intuitive to write than an integer program is to suppose we are dealing with disequalities. If we want to write out that that in a integer program, we would need to form a disjunction of $x_i < x_j$ or $x_i > x_j$. Whereas, the constraint program uses a much more intuitive AllDifferent predicate that essentially does the same thing, but is easier to write out and requires less burden on the user writing the program.

There are many different types of ways to formulate a constraint programming problem. Python constraint is the solver that we used and some of the types of constraints include AllDifferent, AllEqual, ExactSum, InSet, NotInSet, MinSum, MaxSum, and Function. These types of constraints have specific properties. The constraint program has three different possible solvers, which include a backtracking solver, a recursive backtracking solver, and a minimum conflicts solver. Constraint programming appears to have a more limited set of applications than integer programming, which is a drawback. [7]

As alluded to previously, a sudoku can be thought of as a constraint program since it seeks to satisfy a series of constraints. Given the uniqueness of a sudoku, a constraint program should not require any form of search when determining the solution of a sudoku. The key constraint that we seek to take advantage of via constraint programming is that all the rows, columns, and boxes must have different digits. Simonis [8] pairs the all different constraint with other possible constraints and then tests these combination to determine whether or not they are able to solve the sudoku for varying difficulty and then how long it takes. Certain combinations work at better rates than others. In the next section, we will use a CP that has been shown to always solve a sudoku regardless of difficulty.

2.5 Coloring/Graphs

An article by Herzberg approaches the problem of sudoku as a graph coloring problem. It is clear that a sudoku can be thought of as a graph. [9] Each cell interacts with other cells as we discussed in Bartlett's article. Herzberg then says that a sudoku could be reduced to a coloring problem. A sudoku begins as a partial coloring and the question becomes whther the partial coloring can be completed as a total coloring. In general for an $n^2 \times n^2$ sudoku, at least $n^2 - 1$ colors must be used in the given partial coloring in order for the given puzzle to be unique.

The article then theorizes that for every sudoku graph, there is a proper coloring that uses n^2 colors, with a chromatic number of n^2. They approach sudoku from a coloring problem to verify whether or not a sudoku problem is unique. This validates the sudoku.

Chapter 3

IP/CP formulation and Heuristics

3.1 Introduction

We demonstrate various methods for solving sudoku. These methods include integer programming (IP), constraint programming (CP), and the development of a heuristic. We provide pseudo code for each and discuss how each method works.

3.2 Integer Programming

Integer programming is a good method for solving sudoku puzzles because a sudoku can be modeled as a maximization problem with a series of constraints. The objective function ensures that every cell is assigned a number. The rules of sudoku can be thought of as the constraints. We can define the rules that allow each number to be contained only once in each row, column, and box as constraints. In the following subsection, we formulate our integer programming problem.

3.2.1 Integer Program

Sets:

- I : indices $I = \{1, ..., n^3\}$
- R: rows $R = \{1, ..., n\}$
- $R_i \subseteq I$, $i \in R$ row indices
- C: columns $C = \{1, ..., n\}$
- $C_i \subseteq I$, $i \in C$ column indices
- B: boxes $B = \{1, ..., n\}$
- $B_i \subseteq I$, $i \in B$ box indices

- N: cells $N = \{1, ..., n\}$
- $N_i \subseteq I$ $i \in N$ cell indices

Constraints:

$$\sum_{k \in R_i \bigcap N_j} x_k = 1 \quad \forall\, i \in R,\ j \in N \tag{3.1}$$

$$\sum_{k \in C_i \bigcap N_j} x_k = 1 \quad \forall\, i \in C\ j \in N \tag{3.2}$$

$$\sum_{k \in B_i \bigcap N_j} x_k = 1 \quad \forall\, i \in B,\ j \in N \tag{3.3}$$

$$\sum_{k \in N_i} x_k = 1 \quad \forall i \in N \tag{3.4}$$

Objective:

$$\max \sum_{k} x_k$$

From the problem formulation, we defined the optimization as a maximization problem. We define sets for the total number of values in the puzzle, the rows, columns, boxes, and cells. Once these were defined we set up constraints to ensure each row column and box contains exactly one of each number. The maximization of our objective function ensures that every cell is assigned a number. Below we present a few simple examples to demonstrate more easily how our integer program works.

We applied integer programming to various 9×9 sudoku puzzles using Pythons GLPK solver, which is used for linear and integer programming. We use a brief example to demonstrate of the code works. We begin by defining a grid of 81 cells. Each cell could contain a digit from 1 to 9. Hence we have 729 possible values in a sudoku, this is the index set given in the problem formulation. The first nine values are associated with the first cell since it could be any number between 1 and 9. Likewise, the second nine values pertain to the second cell. And so on and so forth. All of these values are binary. Hence if we have the following value[3] = 1, then cell 1 equals 4 (notice that python starts everything at 0 instead of 1). Whereas value[3] = 0 means that cell 1 does not equal 4. Similarly, value[15] = 1 means that cell 2 equals 7.

We define our constraints to ensure that each cell is limited to one number. From the previous example, value[3] = 1, our cell constraint then sets value[0], value[1], , value[8] = 0. We next define the row constraint. We want each row to only contain one of each number. Our constraint acts thusly, suppose value[3] = 1, meaning that cell 1 equals 4. The row constraint then sets all the other binaries in the same row that corresponds to a cell equaling 4, to zero. In this case, value[12], value[21], , value[75] all equal 0. Using the same logic, we define our column constraints and box constraints to act in the same manner.

A final observation to note about the IP solver is that it does not determine if the solution is unique. If the sudoku has multiple solutions, the IP solver will determine one solution, but will not attempt to search for more. And if no solution exists, the IP will be able to determine this quickly through an infeasibility test.

3.3 Constraint Programming

Using constraint programming, we approach the same problem from a different angle. Constraint programming utilizes the all different constraint and exact sum constraint. The exact sum constrain is used to ensure that each row/column/box add up to $\frac{(n+1)*n}{2}$, which is the summation from 1 through n^2. In the case of a $9x9$ sudoku, the row/column/box add up to 45 (summation of 1 through 9). The all different constraint ensures that all values in each row/column/box are different. The constraint program seeks to reduce the domains of each cell in terms of possible candidates for what digit a cell will be. A search is done iteratively until a solution in which all constraints are satisfied is found.

This method will also determine an optimal solution if one exists and not seek to determine if multiple optima exist. Since its approach seeks to satisfy constraints, there is much more variance in solving time as we will see in the next section.

Solve Sudoku using Constraint Programming:

Input: Sudoku Grid

Define:
    Rows
    Columns
    Boxes
    Linesum = $\frac{(n+1)*n}{2}$
For row in row set **do:**
    exactsumconstraint(linesum)
    alldifferentconstraint()
end

For column in column set **do:**
    exactsumconstraint(linesum)
    alldifferentconstraint()
end

For box in box set **do:**

exactsumconstraint(linesum)
alldifferentconstraint()
end

Output: Completed Sudoku Grid

3.4 Heuristic

We want to study the impact of a sudoku puzzle when a heuristic is implemented. Our general idea when approaching the creation of a heuristic is thus, it must be simple enough that it takes little time when executed and it should have a chance of significantly improving the solving time of the integer and/or constraint program.

When thinking about solving a sudoku puzzle, the simplest way to go about solving is to look at each particular cell and eliminate as many possible candidates until only one remains. Then the cell can be filled in accordingly. We wanted to structure our heuristic in a similar manner. The idea being such a solving method will not require much computing time due to its simplicity, yet at the same time could provide meaningful improvements to the solving time when paired with an integer program or constraint program. Obviously as puzzles become harder, this method is insufficient to solve an entire puzzle, but it could provide enough new information that the solving time is reduced.

The heuristic looks at each cell in the grid. If a cell is initially filled in, the heuristic reduces the domain of that cell to the number that is filled in. If the cell is blank, the heuristic assigns it a domain of 1 through 9 of which all are possible candidates for the value of that particular cell. This is done for all 81 cells in the sudoku. Once the domains are established, the heuristic evaluates each row. If a particular number is filled in for row one, that number is removed from the domains of all the unsolved cells in row 1. This process continues iteratively for rows 2 through 9. The same logic is again applied to columns and boxes. Once all of this takes place there could be some cells whose domain has been reduced to be one number. This means that that particular cell has been solved. The heuristic then runs another iteration using this new information. Eventually one of two possibilities will occur. First, the sudoku is solved, in which case the IP/CP are not needed. Or second, the heuristic stalls.

For the case in which the heuristic stalls, we established a counter to determine how many cells are filled in at each iteration. If the counter remains the same for two consecutive iterations, the algorithm terminates. The heuristic has done its job and now an IP or CP can be applied. The logic behind doing this is that if the heuristic fails to provide any new information in one iteration, there will be no new information the rest of the way forward, hence we should discontinue our search.

Heuristic

Input: Sudoku Grid

Define:
Rows
Columns
Boxes
Cells
Cell Domains = $\{1, ..., n\}$
Values = $\{1, ..., n\}$ Counter

While counter > 0: **do:**

For all cells in sudoku grid **do:**
If value exists in row **do**
remove value from cell domain
If value exists in column **do**
remove value from cell domain
If value exists in box **do**
remove value from cell domain
If cell domain contains one digit **do**
append value to cell
end
update counter
end

Output: Completed Sudoku Grid

Chapter 4

Results of Experimentation

4.1 Method

In order to compare different programming methods in a meaningful way, we devised a standardized procedure to test the solve times for each method. We used www.websudoku.com [10] to generate random sudokus. This particular website categorizes sudokus by four different levels of difficulty: easy, medium, hard, and evil. We tested our programs on ten different puzzles replicating each individual puzzle 20 times. We recorded the mean, median, standard deviation, minimum, and maximum for each level of difficulty. All of the times were recorded via the timer function in python.

Our integer program used the GLPK solver package in python. We used Simple API's code for our constraint solver with a slight modification in python. [11] Full code for both the integer program as well as the constraint program are provided in the appendix.

4.2 IP/CP Results

We begin by comparing the solve times of our Integer Program with our Constraint Program. The results are displayed in Table 4.1, all times are in seconds. Additionally, a bar graph is used in figure 4.1 to illustrate the comparison between the mean running times at each difficulty level.

From this data, we observe that the CP solver took longer as the level of difficulty increased. Meanwhile, the IP solver was very consistent in it's solving times. In addition, the standard deviation of running times was much greater for the constraint program. It would stand to reason that Integer Programming is preferable especially as difficulty and the size of the problem increases.

Constraint Program

	Difficulty			
	Easy	Medium	Hard	Evil
Mean	0.05212	0.14628	0.34467	0.92403
Median	0.05200	0.12798	0.23807	0.79115
Std Deviation	0.01034	0.08693	0.25957	0.76987
Minimum	0.03481	0.04164	0.08287	0.19204
Maximum	0.06815	0.30634	0.86603	2.22008

Table 4.1. Constraint programming run times on various sudoku puzzle types.

Integer Program

	Difficulty			
	Easy	Medium	Hard	Evil
Mean	0.17209	0.17813	0.18117	0.18211
Median	0.17144	0.17892	0.18006	0.18270
Std Deviation	0.00306	0.00305	0.00165	0.00254
Minimum	0.16811	0.17194	0.17937	0.17598
Maximum	0.17796	0.18178	0.18423	0.18507

Table 4.2. Integer programming run times on various sudoku puzzle types.

4.3 Heuristic

Additionally, we tested the solving times of the Constraint Program with the incorporation of a heuristic. As detailed in the previous section, our heuristic goes cell by cell and reduces the domain of possible values accounting for numbers in the same row, column, and box. Once all the cells domain's are reduced, if any cells have a domain reduced to only one value, then that cell is assigned the corresponding value. This same process is applied for each row and column so that if only one value appears in a particular row or column, that cell is assigned that particular value. Once this process is completed, the constraint program runs as usual with this new information.

The purpose of this heuristic is to provide the constraint program with additional information. The intuition being a constraint program will solve faster when there is more information being provided. Our results back up this claim. An almost every circumstance the CP was able to solve faster when using the heuristic.

The reason we only apply our heuristic with the constraint program is that we observed that the solve times for the integer program remained steady regardless of difficulty. This suggests that the amount of information given prior to solving an integer program has significantly less impact on the solve time compared to the constraint program.

We encountered the following observations while testing, see Figure 4.2.. Regardless of difficulty, the heuristic was able to significantly reduce solving times. For easy and medium sudokus, the heuristic was able to solve the sudoku without having to call the constraint program whatsoever. This was also the case for most, but not all of the hard sudokus. For evil sudokus, the heuristic filled in a number of cells but eventually stopped, causing the constraint program to be called. Since, there were less cells remaining for the constraint program, it was able to solve the

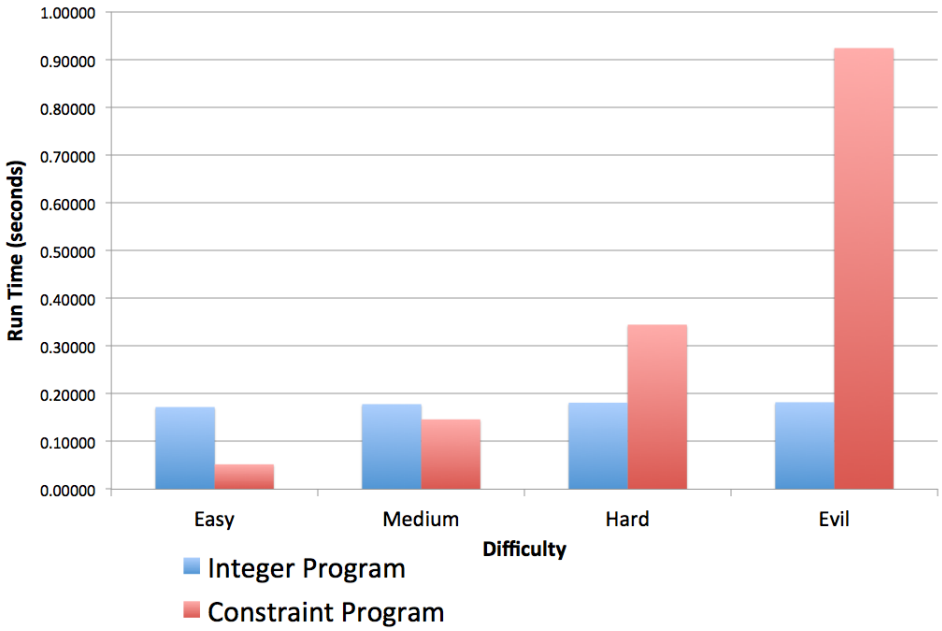

Figure 4.1. A bar chart comparing the run times in solving a sudoku with Integer Programming vs. Constraint Programming.

Difficulty: Easy

	Time Elapsed			
	w/o heuristic	w/ heuristic	% less time	% faster
Mean	0.05212	0.01749	66.4510%	198.0713%
Median	0.05200	0.01753	66.2916%	196.6623%
Std Deviation	0.01034	0.00148		
Minimum	0.03481	0.01427		
Maximum	0.06815	0.02047		

Table 4.3. Comparing solve time on easy sudoku with and without a heuristic.

problem faster.

4.4 Cells remaining reduction

Based on our research with our heuristic, the question occurred to us, is there a certain number of cells in which the sudoku can be solved very quickly with a Constrain Program? Our intuition was that the fewer cells remaining, the less time it would take the CP to solve. To measure this, we devised the following experiment. First, we took an evil puzzle. We chose evil puzzles since they have that most unfilled cells. For evil puzzles, 55 of the 81 cells are unfilled. This provides a picture for how solving times improve as more information becomes available. Additionally, we know every evil puzzle on www.websudoku.com is ensured to have a unique solution which is an important assumption for all sudoku problems. By using a random number generator, we randomly picked one cell at a time to fill in with the correct solution. Using this new information,

Difficulty: Medium

	Time Elapsed			
	w/o heuristic	w/ heuristic	% less time	% faster
Mean	0.14628	0.02856	80.4769%	412.2142%
Median	0.12798	0.03043	76.2244%	320.5994%
Std Deviation	0.08693	0.00505		
Minimum	0.04164	0.02059		
Maximum	0.30634	0.03377		

Table 4.4. Comparing solve time on medium sudoku with and without a heuristic.

Difficulty: Hard

	Time Elapsed			
	w/o heuristic	w/ heuristic	% less time	% faster
Mean	0.34467	0.05089	85.2354%	577.2969%
Median	0.23807	0.04260	82.1067%	458.8685%
Std Deviation	0.25957	0.02113		
Minimum	0.08287	0.03629		
Maximum	0.86603	0.10538		

Table 4.5. Comparing solve time on hard sudoku with and without a heuristic.

we postulate that the sudoku should solve faster. Our results are shown in Figure 4.3.

From our experiment, our results show an exponential decay in the amount of time it takes to solve a sudoku puzzle. We saw that solve times would decrease drastically with one new cell of information but there was not a particular number of cells remaining when that jump would occur. Sometimes it occurred with the first new piece of information, other times it occurred on the fourth. This leads us to the idea that there are critical cells that have a greater effect on solving time reduction. This is a topic we wish to investigate further.

Difficulty: Evil

	Time Elapsed			
	w/o heuristic	w/ heuristic	% less time	% faster
Mean	0.92403	0.27150	70.6181%	240.3461%
Median	0.79115	0.14235	82.0076%	455.7896%
Std Deviation	0.76987	0.25068		
Minimum	0.19204	0.05163		
Maximum	2.22008	0.81147		

Table 4.6. Comparing solve time on evil sudoku with and without a heuristic.

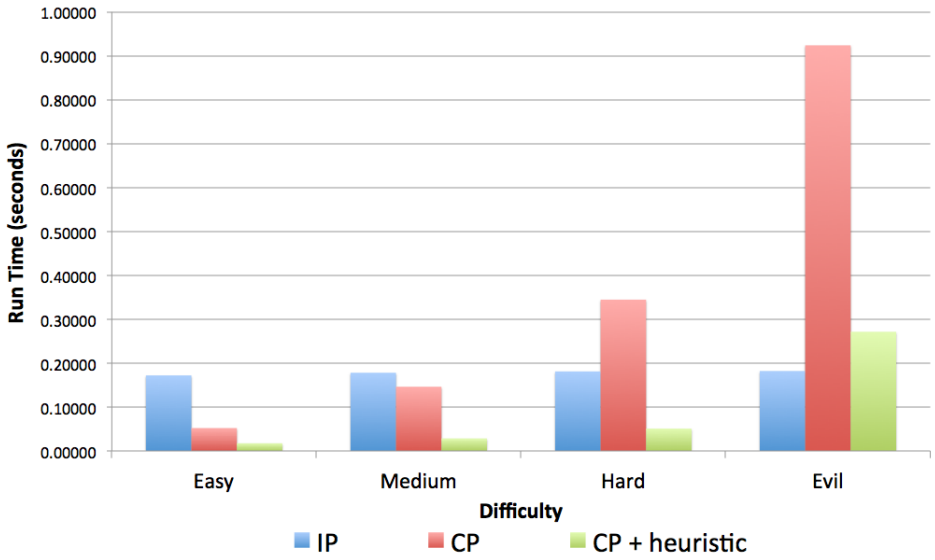

Figure 4.2. A bar chart comparing the run times in solving a sudoku with Integer Programming vs. Constraint Programming vs. Constraint Programming with Heuristic.

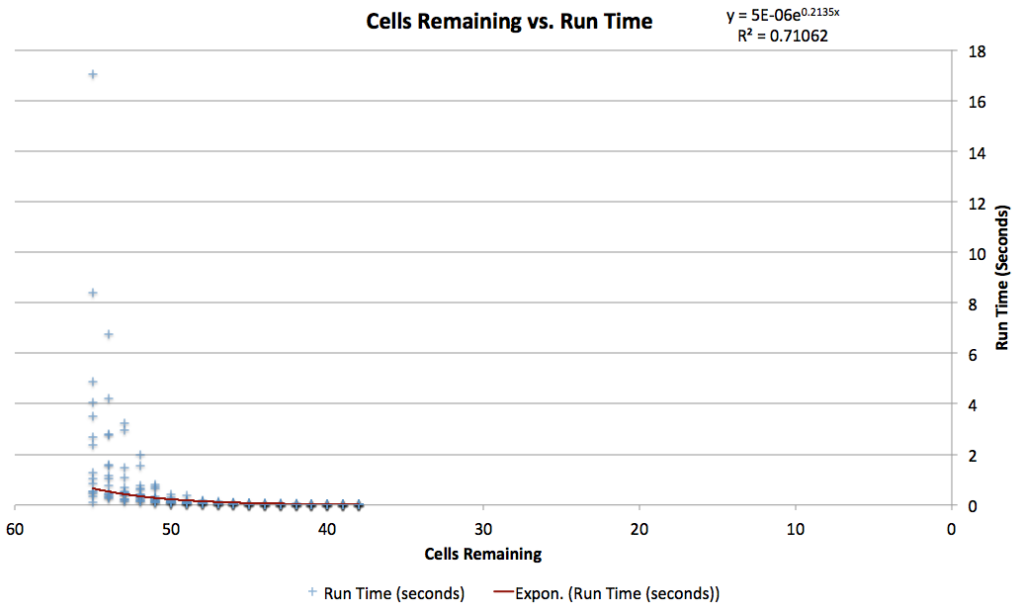

Figure 4.3. Solution time as a function of percent of puzzle solved using the constraint programming.

Chapter 5

Conclusions and Future Work - Deception Problem/Graph Problem

5.1 Conclusions

In this thesis, we presented a review of various results on sudoku. We discussed the complexity of sudoku problems, various integer and constraint programming results related to sudoku, as well as heuristics that have been used in solving sudokus. We also discussed graph properties and coloring properties of sudokus.

We then defined our own integer and constraint programs to solve sudokus. We formulated a heuristic that can aid in reducing the solving times for sudokus when using constraint programming. We tested these programs and observed that the integer program solved consistently at just under one fifth of a second regardless of difficulty. Meanwhile the constraint program was able to solve easier sudokus extremely quickly but as the difficulty increased, so did the solve times. So much so that the hardest sudokus took nearly five times as long to solve compared to the integer program.

We then paired the constraint programming solver with a heuristic. We saw significant improvement in solve time regardless of difficulty. For easy, medium, and hard sudokus the heuristic typically solved the sudoku faster than the integer program. This is usually because the constraint program was never called. But for the most difficult sudokus the solve time was still worse than that of an integer program.

This heuristic testing inspired us to look into how the number of empty cells effects solving times. We observed exponential decay in solve times as more cells were filled in. This suggests a possible area of future research on critical cells.

5.2 Complexity

As we stated above we want to learn more about the structure of sudokus and how this relates to solve times. We witnessed an exponential decay in solve time as more cells were filled in when information is provided, but in our testing we saw that solve times significantly reduced relatively early on in our testing and then steadied off. But sometimes it would take longer to get to this improvement in solve times. Other times it would happen right away. We showed in our literature review that sudokus can be thought of as graph coloring problems. Hence, we would like to explore the properties of these so called critical cells exist.

Is there a way to determine based on what cells are initially given, which cell will reduce remaining solve time the most? We could then say that this cell is more important in some respects to the other cells in the sudoku puzzle. If we knew the two or three most important cells of a sudoku puzzle, how could we then use this information to solve the sudoku faster?

This idea of critical cells should not merely be contained to sudoku puzzles. There are probably a myriad of graph coloring applications that could benefit from a more thorough analysis of these so called critical cells. We feel that studying this topic from the perspective of sudokus could provide a greater understanding and then we would be able to better apply this knowledge in these various other applications.

5.3 Desception

Finally, we wish to explore how these types of programs and heuristics that were used to solve sudokus could be used to understand deception. A possible idea of a deception problem is one in which we receive information from various sources (eg: social media, informant, etc.). We want to determine veracity of the information. Is the person giving us this information trying to deceive us in any way?

A brief example of how this could be tied back to the sudoku problem is that suppose a person wants to do the sudoku problem in the inflight magazine on an airplane. When he opens the sudoku he sees that it is already partially filled in. Would it be best to continue where the last person left off or to erase that person's answers and start over? Our heuristic could be modified and applied in this scenario to determine if there are any glaring inconsistencies in the puzzle. For instance if two of the same number are contained in the same row, we obviously would not trust the information given.

Is there a way to modify our heuristic to determine the most likely to be filled in cells based on the given information? If the cells filled in adhere to a predictable solving pattern, then that would suggest the information is good. But if the cells filled in do not follow a logical solving pattern, then there is a greater chance that the person guessed or filled that cell in with faulty logic. We ultimately would want to use this information and apply it in other settings to determine the trustworthiness of information.

Appendix A

Integer Programming Python Code

In this appendix we give the python code that we used for solving sudoku using the integer programming approach.

```
import glpk
import time
start_time = time.time()

lp = glpk.LPX()
lp.name = 'sample'
lp.obj.maximize = True

################################################

cells = ['']*81
values = [0]*729

################################################

cells[0:9]   = [0,0,0,0,0,0,0,0,0]
cells[9:18]  = [0,0,0,0,0,0,0,0,0]
cells[18:27] = [0,0,0,0,0,0,0,0,0]
cells[27:36] = [0,0,0,0,0,0,0,0,0]
cells[36:45] = [0,0,0,0,0,0,0,0,0]
cells[45:54] = [0,0,0,0,0,0,0,0,0]
cells[54:63] = [0,0,0,0,0,0,0,0,0]
cells[63:72] = [0,0,0,0,0,0,0,0,0]
```

```
cells[72:81] = [0,0,0,0,0,0,0,0,0]

################################################

lp.rows.add(324)
for r in lp.rows:

    r.bounds = 1.0

lp.cols.add(729)
for c in lp.cols:
    c.name = 'x%d' % c.index
    c.kind = int

################################################

for rowconstr in range(0,729):
    if cells[(rowconstr/9)] == (rowconstr%9 + 1):
        values[rowconstr] = 1.0

for colconstr in range(0,729):
    lp.cols[colconstr].bounds = values[colconstr], 1.0

################################################

cell = ([1.0]*9 + [0.0]*729)*80 + [1.0]*9

row = ((([1.0] + [0.0]* 8)*8 + [1.0] + [0.0]*657)*9 + [0.0]*72)*8 +
(([1.0] + [0.0]* 8)*8 + [1.0] + [0.0]*657)*8 + ([1.0] + [0.0]*8)*8 +[1.0]

col = (([1.0] + [0.0]*80)*8 + [1.0] + [0.0]*81)*80 + ([1.0] +
[0.0]*80)*8 + [1.0]

box =  (((([1.0] + [0.0]*8)*2 + [1.0] + [0.0]*62)*3 + [0.0]*487)*9 +
[0.0]*18)*2 + ((([1.0] + [0.0]*8)*2 + [1.0] + [0.0]*62)*3 +
[0.0]*487)*9
+ [0.0]*180 + (([1.0] + [0.0]*8)*2 + [1.0] +  [0.0]*62)*3 + ([0.0]*487
+
(([1.0] + [0.0]*8)*2 + [1.0] + [0.0]*62)*3)*8 + [0.0]*18 + ([0.0]*487 +
(([1.0] + [0.0]*8)*2 + [1.0] + [0.0]*62)*3)*9 + [0.0]*18 + ([0.0]*487 +
```

```
(([1.0] + [0.0]*8)*2 + [1.0] + [0.0]*62)*3)*9 + [0.0]*667 + (([1.0] +
[0.0]*8)*2 + [1.0] + [0.0]*62)*3 + ([0.0]*487 + (([1.0] + [0.0]*8)*2 +
[1.0] + [0.0]*62)*3)*8 + [0.0]*18 + ([0.0]*487 + (([1.0] + [0.0]*8)*2 +
[1.0] +  [0.0]*62)*3)*9 + [0.0]*18 + ([0.0]*487 + (([1.0] + [0.0]*8)*2+
[1.0] + [0.0]*62)*3)*8 + [0.0]*487 + (([1.0] + [0.0]*8)*2 + [1.0] +
[0.0]*62)*2 + [1.0] + [0.0]*8 + [1.0] + [0.0]*8 + [1.0]

lp.obj[:] = 1.0
lp.matrix = cell+row+col+box

################################################

lp.simplex()
lp.integer()

for colconstr in range(0,729):
    if lp.cols[colconstr].value == 1:
        cells[colconstr/9] = (colconstr%9 +1)


print cells[0:9]
print cells[9:18]
print cells[18:27]
print cells[27:36]
print cells[36:45]
print cells[45:54]
print cells[54:63]
print cells[63:72]
print cells[72:81]

print 'Z = %g;' % lp.obj.value

print time.time() - start_time
```

Appendix B

Constraint Program

In this appendix we give the python code that we used for solving sudoku using the constraint programming approach.

```
import sys, math
sys.path.append("./python-constraint-1.1")

#Importing constraint
from constraint import *

import time
start_time = time.time()

cells = ['']*81

cells[0:9]   = [0,0,0,0,0,0,0,0,0]
cells[9:18]  = [0,0,0,0,0,0,0,0,0]
cells[18:27] = [0,0,0,0,0,0,0,0,0]
cells[27:36] = [0,0,0,0,0,0,0,0,0]
cells[36:45] = [0,0,0,0,0,0,0,0,0]
cells[45:54] = [0,0,0,0,0,0,0,0,0]
cells[54:63] = [0,0,0,0,0,0,0,0,0]
cells[63:72] = [0,0,0,0,0,0,0,0,0]
cells[72:81] = [0,0,0,0,0,0,0,0,0]


def solveSudoku(size = 9, originalGame = None):
```

```
    """ Solving Sudoku of any size """
    sudoku = Problem()

    #Defining size of row/col
    rows = range(size)
    cols = range(size)

    #every line got same sum
    lineSum = sum(range(1, size+1))

    #Creating board
    board = [(row, col) for row in rows for col in cols]
    #Defining game variable, a single range will be enough
    sudoku.addVariables(board, range(1, size * size + 1))

    #Row set
    rowSet = [zip([el] * len(cols), cols) for el in rows]
    colSet = [zip(rows, [el] * len(rows)) for el in cols]

    #The original board is not empty, we add that constraint to the
list of constraint
    if originalGame is not None:
        for i in range(0, size):
            for j in range(0, size):
                #Getting the value of the current game
                o = originalGame[i][j]
                #We apply constraint when the number is set only
                if o > 0:
                    #We get the associated tuple
                    t = (rows[i],cols[j])
                    #We set a basic equal constraint rule to force the
system to keep that variable at that place
                    sudoku.addConstraint(lambda var, val=o: var == val,
(t,))

    #The constraint are like that : and each row, and each
columns, got same final compute value, and are all unique
    for row in rowSet:
        sudoku.addConstraint(ExactSumConstraint(lineSum), row)
        sudoku.addConstraint(AllDifferentConstraint(), row)
```

```
    for col in colSet:
        sudoku.addConstraint(ExactSumConstraint(lineSum), col)
        sudoku.addConstraint(AllDifferentConstraint(), col)

    #Every sqrt(size) (3x3 box constraint) got same sum
    sqSize = int(math.floor(math.sqrt(size)))

    #xrange allow to define a step, here sq (wich is sq = 3 in 9x9
sudoku)
    for i in xrange(0,size,sqSize):
        for j in xrange(0,size,sqSize):
            #Computing the list of tuple linked to that box
            box = []
            for k in range(0, sqSize):
                for l in range(0, sqSize):
                    #The tuple i+k, j+l is inside that box
                    box.append( (i+k, j+l) )
            #Compute is done, now we can add the constraint for that
box
            sudoku.addConstraint(ExactSumConstraint(lineSum),
box)
            sudoku.addConstraint(AllDifferentConstraint(), box)

    #Computing and returning final result
    return sudoku.getSolution()


if __name__ == '__main__':
    rg = 9
    initValue = [cells[0:9],
                 cells[9:18],
                 cells[18:27],
                 cells[27:36],
                 cells[36:45],
                 cells[45:54],
                 cells[54:63],
                 cells[63:72],
                 cells[72:81]]

    res = solveSudoku(rg, initValue)
```

```
    print
    if res is not None:
        for i in range(0, rg):
            for j in range(0, rg):
                print res[i, j],
            print
        print
    else:
        print "No result to show"

#print "The sudoku solver took", time.time() - start_time, "to run"
print time.time() - start_time
```

Appendix C

Heuristic

We paired this heuristic with the constraint program in chapter four. Below is just the heuristic, a slight modification is necessary to combine this code with the constraint program.

```
import sys, math
sys.path.append("./python-constraint-1.1")

#Importing constraint
from constraint import *

import time
start_time = time.time()

cells = ['']*81
domain = ['']*81
for i in range(0,81):
    domain[i] = [1,2,3,4,5,6,7,8,9]

##################################################
#####

cells[0:9]   = [4,0,0,0,0,0,0,0,1]
cells[9:18]  = [0,8,0,0,7,0,2,0,3]
cells[18:27] = [0,0,0,9,0,0,0,6,0]
cells[27:36] = [5,9,0,7,0,0,0,0,6]
cells[36:45] = [0,0,0,0,2,0,0,0,0]
cells[45:54] = [6,0,0,0,0,1,0,4,2]
```

```
cells[54:63] = [0,5,0,0,0,9,0,0,0]
cells[63:72] = [3,0,6,0,4,0,0,2,0]
cells[72:81] = [2,0,0,0,0,0,0,0,8]

################################################
#####

count = 0
for i in range(0,80):
    if cells[i] == 0:
        count = count + 1
newcount = 0


def cdr():
#cell domain reducer

    for j in range(0,81):
        for k in range(1,10):
            if cells[j] == k:
                domain[j] = [k]


########################################
#row domain reducer

def rdr():

    startrow = 0
    endrow= 9
    while startrow < 80:
        for l in range(startrow,endrow):
            for m in range(1,10):
                for n in range(startrow,endrow):
                    if cells[n] == m:
                        #if len(domain[l]) > 1:
                        if m in domain[l]:
                            a = domain[l].index(m)
                            domain[l].pop(a)
        startrow = startrow + 9
```

```
        endrow = endrow + 9

##############################################
#col domain reducer

def coldr():
    cr1 = 0
    cr2 = 9
    cr3 = 18
    cr4 = 27
    cr5 = 36
    cr6 = 45
    cr7 = 54
    cr8 = 63
    cr9 = 72

    while cr1 < 9:
        for l in (cr1,cr2,cr3,cr4,cr5,cr6,cr7,cr8,cr9):
            for m in range(1,10):
                for n in (cr1,cr2,cr3,cr4,cr5,cr6,cr7,cr8,cr9):
                    if cells[n] == m:
                        if len(domain[l]) > 1:
                            if m in domain[l]:
                                a = domain[l].index(m)
                                domain[l].pop(a)
        cr1 = cr1 + 1
        cr2 = cr2 + 1
        cr3 = cr3 + 1
        cr4 = cr4 + 1
        cr5 = cr5 + 1
        cr6 = cr6 + 1
        cr7 = cr7 + 1
        cr8 = cr8 + 1
        cr9 = cr9 + 1

##############################################
#box domain reducer
def bdr():
```

```
for l in (0,1,2,9,10,11,18,19,20):
    for m in range(1,10):
        for n in (0,1,2,9,10,11,18,19,20):
            if cells[n] == m:
                if len(domain[l]) > 1:
                    if m in domain[l]:
                        a = domain[l].index(m)
                        domain[l].pop(a)

for l in (3,4,5,12,13,14,21,22,23):
    for m in range(1,10):
        for n in (3,4,5,12,13,14,21,22,23):
            if cells[n] == m:
                if len(domain[l]) > 1:
                    if m in domain[l]:
                        a = domain[l].index(m)
                        domain[l].pop(a)

for l in (6,7,8,15,16,17,24,25,26):
    for m in range(1,10):
        for n in (6,7,8,15,16,17,24,25,26):
            if cells[n] == m:
                if len(domain[l]) > 1:
                    if m in domain[l]:
                        a = domain[l].index(m)
                        domain[l].pop(a)

for l in (27,28,29,36,37,38,45,46,47):
    for m in range(1,10):
        for n in (27,28,29,36,37,38,45,46,47):
            if cells[n] == m:
                if len(domain[l]) > 1:
                    if m in domain[l]:
                        a = domain[l].index(m)
                        domain[l].pop(a)

for l in (30,31,32,39,40,41,48,49,50):
    for m in range(1,10):
        for n in (30,31,32,39,40,41,48,49,50):
            if cells[n] == m:
```

```
                if len(domain[l]) > 1:
                    if m in domain[l]:
                        a = domain[l].index(m)
                        domain[l].pop(a)

for l in (33,34,35,42,43,44,51,52,53):
    for m in range(1,10):
        for n in (33,34,35,42,43,44,51,52,53):
            if cells[n] == m:
                if len(domain[l]) > 1:
                    if m in domain[l]:
                        a = domain[l].index(m)
                        domain[l].pop(a)

for l in (54,55,56,63,64,65,72,73,74):
    for m in range(1,10):
        for n in (54,55,56,63,64,65,72,73,74):
            if cells[n] == m:
                if len(domain[l]) > 1:
                    if m in domain[l]:
                        a = domain[l].index(m)
                        domain[l].pop(a)

for l in (57,58,59,66,67,68,75,76,77):
    for m in range(1,10):
        for n in (57,58,59,66,67,68,75,76,77):
            if cells[n] == m:
                if len(domain[l]) > 1:
                    if m in domain[l]:
                        a = domain[l].index(m)
                        domain[l].pop(a)

for l in (60,61,62,69,70,71,78,79,80):
    for m in range(1,10):
        for n in (60,61,62,69,70,71,78,79,80):
            if cells[n] == m:
                if len(domain[l]) > 1:
                    if m in domain[l]:
                        a = domain[l].index(m)
                        domain[l].pop(a)
```

```
##############################################
# ROW CHECKER

def rc():

    startrc = 0
    endrc = 9

    while startrc < 80:
        for r in range(1,10):
            count = 0
            for q in range(startrc,endrc):
                if r in domain[q]:
                    count = count + 1
            if count == 1:
                for q in range(startrc,endrc):
                    if r in domain[q]:
                        cells[q] = r
        startrc = startrc + 9
        endrc = endrc + 9

##############################################
# COLUMN CHECKER

def colc():


    cc1 = 0
    cc2 = 9
    cc3 = 18
    cc4 = 27
    cc5 = 36
    cc6 = 45
    cc7 = 54
    cc8 = 63
    cc9 = 72

    while cc1 < 9:
        for r in range(1,10):
```

```
            count = 0
            for q in (cc1,cc2,cc3,cc4,cc5,cc6,cc7,cc8,cc9):
                if r in domain[q]:
                    count = count + 1
            if count == 1:
                for q in (cc1,cc2,cc3,cc4,cc5,cc6,cc7,cc8,cc9):
                    if r in domain[q]:
                        cells[q] = r
        cc1 = cc1+1
        cc2 = cc2+1
        cc3 = cc3+1
        cc4 = cc4+1
        cc5 = cc5+1
        cc6 = cc6+1
        cc7 = cc7+1
        cc8 = cc8+1
        cc9 = cc9+1



##############################################
# BOX CHECKER

def bc():


    for r in (1,10):
        count = 0
        for q in (0,1,2,9,10,11,18,19,20):
            if r in domain[q]:
                count = count + 1
        if count == 1:
            for q in (0,1,2,9,10,11,18,19,20):
                if r in domain[q]:
                    cells[q] = r

    for r in (1,10):
        count = 0
        for q in (3,4,5,12,13,14,21,22,23):
            if r in domain[q]:
```

```
            count = count + 1
    if count == 1:
        for q in (3,4,5,12,13,14,21,22,23):
            if r in domain[q]:
                cells[q] = r

for r in (1,10):
    count = 0
    for q in (6,7,8,15,16,17,24,25,26):
        if r in domain[q]:
            count = count + 1
    if count == 1:
        for q in (6,7,8,15,16,17,24,25,26):
            if r in domain[q]:
                cells[q] = r

for r in (1,10):
    count = 0
    for q in (27,28,29,36,37,38,45,46,47):
        if r in domain[q]:
            count = count + 1
    if count == 1:
        for q in (27,28,29,36,37,38,45,46,47):
            if r in domain[q]:
                cells[q] = r

for r in (1,10):
    count = 0
    for q in (30,31,32,39,40,41,48,49,50):
        if r in domain[q]:
            count = count + 1
    if count == 1:
        for q in (30,31,32,39,40,41,48,49,50):
            if r in domain[q]:
                cells[q] = r

for r in (1,10):
    count = 0
    for q in (33,34,35,42,43,44,51,52,53):
        if r in domain[q]:
```

```
                count = count + 1
        if count == 1:
            for q in (33,34,35,42,43,44,51,52,53):
                if r in domain[q]:
                    cells[q] = r

    for r in (1,10):
        count = 0
        for q in (54,55,56,63,64,65,72,73,74):
            if r in domain[q]:
                count = count + 1
        if count == 1:
            for q in (54,55,56,63,64,65,72,73,74):
                if r in domain[q]:
                    cells[q] = r

    for r in (1,10):
        count = 0
        for q in (57,58,59,66,67,68,75,76,77):
            if r in domain[q]:
                count = count + 1
        if count == 1:
            for q in (57,58,59,66,67,68,75,76,77):
                if r in domain[q]:
                    cells[q] = r

    for r in (1,10):
        count = 0
        for q in (60,61,62,69,70,71,78,79,80):
            if r in domain[q]:
                count = count + 1
        if count == 1:
            for q in (60,61,62,69,70,71,78,79,80):
                if r in domain[q]:
                    cells[q] = r

##############################################
#domain/cell reducer
def dcr():
```

```
    for p in range(0,81):
        if len(domain[p]) == 1:
            domain[p] = int(''.join(map(str,domain[p])))

    for j in range(0,81):
        for k in range(1,10):
            if domain[j] == k:
                cells[j] = k



def heuristic():
    cdr()
    rdr()
    coldr()
    bdr()
    rc()
    colc()
    bc()
    dcr()


while newcount != count:
    newcount = count
    print "New Count = " + str(newcount)
    heuristic()
    count = 0
    for i in range(0,80):
        if cells[i] == 0:
            count = count + 1

##############################################
#printer

for l in range(0,81):
    print "cell " + str(l) + " = " + str(domain[l])

for l in range(0,81):
    print "cell " + str(l) + " = " + str(cells[l])
```

```
print "cells[0:9]   = " + str(cells[0:9])
print "cells[9:18]  = " + str(cells[9:18])
print "cells[18:27] = " + str(cells[18:27])
print "cells[27:36] = " + str(cells[27:36])
print "cells[36:45] = " + str(cells[36:45])
print "cells[45:54] = " + str(cells[45:54])
print "cells[54:63] = " + str(cells[54:63])
print "cells[63:72] = " + str(cells[63:72])
print "cells[72:81] = " + str(cells[72:81])
print time.time() - start_time

################################################
#######
```

Bibliography

[1] P. Gordon and F. Longo, *Mensa Guide to Solving Sudoku: Hundreds of Puzzles plus Techniques to Help You Crack Them All.* Sterling, 2006.

[2] C. Colbourn, M. Colbourn, and D. Stinson, "The computational complexity of recognizing critical sets," in *Graph Theory Singapore 1983*, ser. Lecture Notes in Mathematics, K. Koh and H. Yap, Eds. Springer Berlin Heidelberg, 1984, vol. 1073, pp. 248–253.

[3] T. Yato, "Complexity and completeness of finding another solution and its application to puzzles," Master's thesis, University of Tokyo, January 2003.

[4] A. Bartlett, T. Chartier, A. Langville, and T. Rankin, "An integer programming model for the sudoku problem," *The Journal of Online Mathematics and Its Applications*, 2008.

[5] K. Apt, *Principles of Constraint Programming.* Cambridge, 2003.

[6] M. Milano, *Constraint and Integer Programming Toward a Unified Methodology.* Kluwer Academic, 2004.

[7] T. Fruhwirth, *Essentials of Constraint Programming.* Springer, 2003.

[8] *Sudoku as a Constraint Program.* 4th International Workshop on Modelling and Reformulating Constraint Satisfaction Problems, 2005.

[9] A. Herzberg and M. Murty, "Sudoku squares and chromatic polynomials," *Notices of the AMS*, vol. 54, no. 6, pp. 708–716, June/July 2007.

[10] [Online]. Available: www.websudoku.com

[11] (2012, November). [Online]. Available: http://simplapi.wordpress.com/2012/11/02/python-constraint-and-sudoku/

Vita

Michael Thein

Academic Vita

Michael W. Thein
michaelthein90@gmail.com

Education

The Pennsylvania State University, May 2014
Master of Science in Industrial Engineering and Operations Research

The Pennsylvania State University, 2012
Bachelor of Science in Mathematics
Minor in Statistics
Minor in Economics

Honors and Awards

Eric A. Walker Graduate Assisantship

Experience

Research Assistant at Penn State's Applied Research Laboratory, 2012-2014
Research Assistant at Penn State's Laboratory for Economic Management and Auctions, 2011

Activities

INFORMS Member - The Pennsylvania State University Chapter, 2012-2014