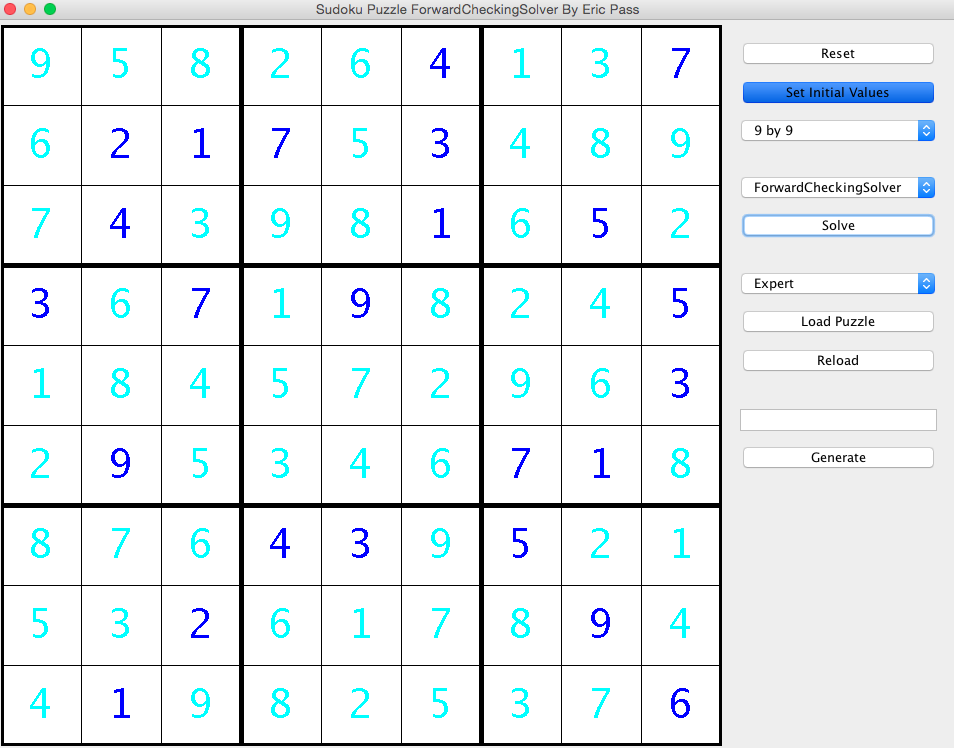
Solving Sudoku Using Artificial Intelligence

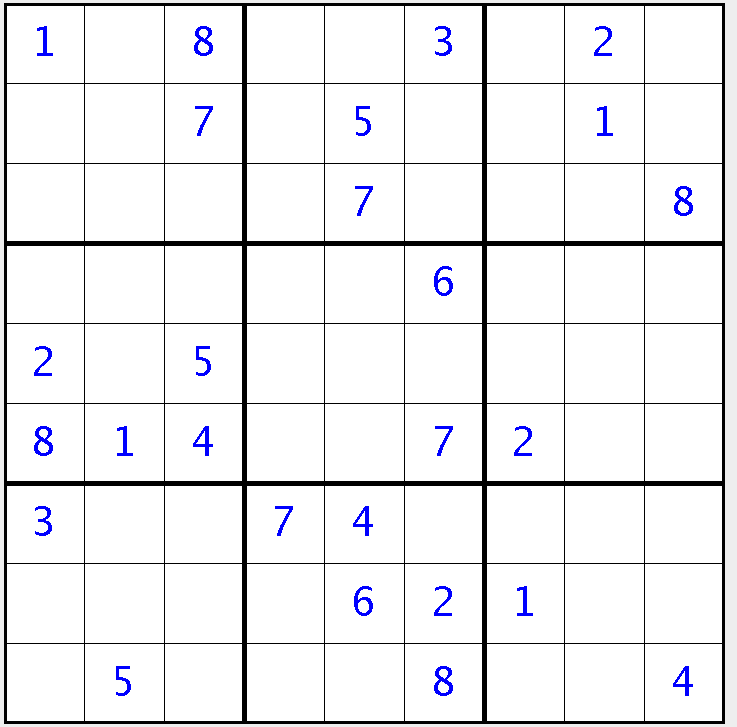
Eric Pass



# Background on Sudoku

## Overview

Sudoku problems are some of the most recognizable and popular puzzle games played around the world. A standard Sudoku puzzle is a flat square grid containing 81 cells well distributed in 9 rows and 9 columns and the gird is divided into 9 smaller squares called units which contain 9 cells each. (Sudoku Grading) For the discussion in this paper we will denote a puzzle of size or order N to mean a Sudoku puzzle that has N columns, N rows and N units.



**Figure 1:** A 9 by 9 Sudoku Puzzle with a unit highlighted in light blue.

The rules of Sudoku are an extension of the rules used in the Latin Square puzzles. (mceliece2.pdf) The rules for a general N by N Sudoku puzzle are:

1. All of the N\*N squares must be assigned a single value from 1 to N
2. Each row must contain each number from 1 to N exactly once
3. Each column must contain each number from 1 to N exactly once
4. Each unit must contain each number from 1 to N exactly once

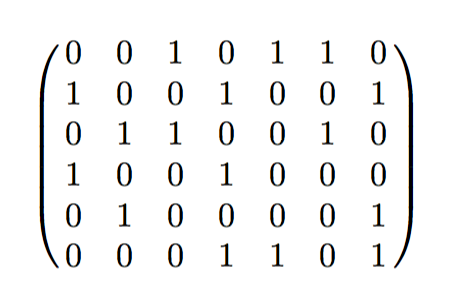
Even though the rules of Sudoku are incredibly simple, the task of correctly filling in a partially filled Sudoku puzzle is exceedingly difficult. Felgenhauer and Jarvis calculate in their paper “Enumerating possible Sudoku grids” that the are 6.671 x 1021 valid 9 by 9 Sudoku puzzles. (Felgenhauer and Jarvis) Since the state space is so enormous, considerable effort has to go into developing algorithms who will yield a correct solution in a sensible amount of computing time.

## Complexity of Sudoku

In 2002, Takayuki and Takahiro at The University of Tokyo proved, by reducing Sudoku into a Latin square completion problem, that Sudoku is NP-Complete. (NP Complete Sudoku) Saying that Sudoku is NP-Complete means that it is possible to reduce Sudoku in polynomial time to another NP-Complete problem. Two common and natural reductions are reducing Sudoku to an Exact Cover Problem and reducing Sudoku to a Boolean Satisfiability Problem.

## Sudoku as an Exact Cover

Donald Knuth in his “Dancing Links” paper explains the exact cover problem very intuitively. The problem statement is: Given a matrix of 0s and 1s, does it have a set of rows containing exactly one 1 in each column? For example, the matrix

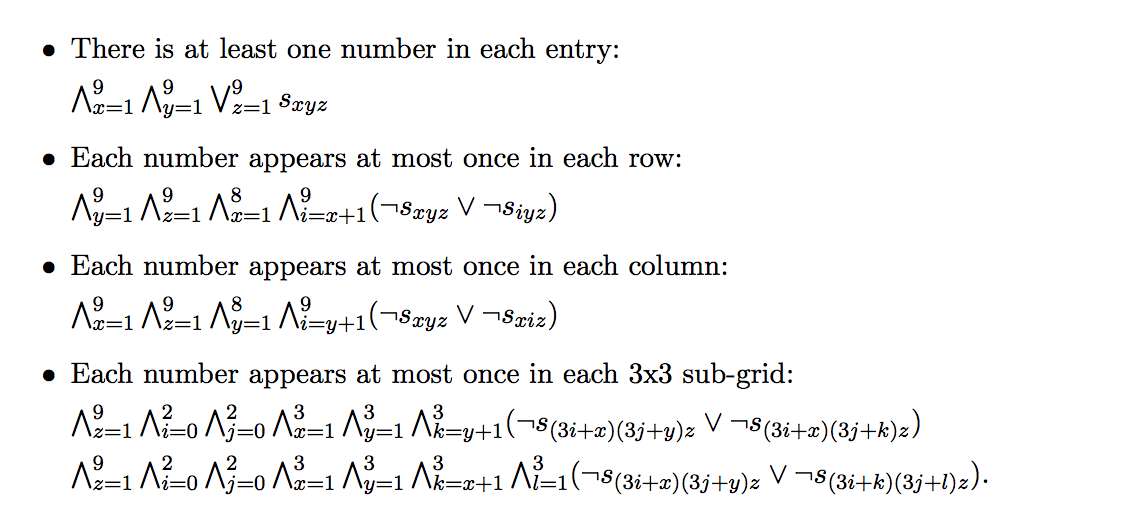


**Figure 2:** An example exact cover problem matrix from Donald Knuth’s paper “Dancing Links.”

has such a set (rows 1, 4, and 5). (Knuth Dancing Links) Sudoku can be reduced into an exact cover problem where, for a 9 by 9 puzzle, the matrix has 729 rows and 324 columns. The rows of this matrix encode information about what value is assigned to a given row and column. Since, for a 9 by 9 puzzle, there are 9 possible values which could occupy any of the 9 columns and any of the 9 rows we have a total number of 9\*9\*9 = 729 rows in this matrix. The columns of this matrix encode the four constraints of the problem of the problem that each cell must be assigned a single value between 1 and 9 and every row, column and box must contain exactly one instance of each number between 1 and 9. This means there is a total of 4\*9\*9 = 324 columns in this matrix. (Application of exact cover) Once Sudoku has been reduced to an exact cover problem, you can use various algorithms like Donald Knuth’s “Aglorithem X” which utilize the “Dancing Links” technique to solve these problems.

## Sudoku as a SAT Problem

Sudoku can just as easily be encoded as a Boolean Satisfiability Problem (SAT). In order to represent Sudoku as a SAT problem you will need a considerable amount of propositional variables. We will denote *Si,j,z* to be assigned true if and only if the cell at row *i* and column *j* is assigned value *z*. This means for a 9 by 9 Sudoku puzzle we will need 729 different propositional variables, the same as the number of rows in the Exact Cover reduction. Adding the constraints that each cell must contain a single value from 1 to 9 and that each row, column and unit must contain exactly one instance of each value from 1 to 9 the size of both the minimal and extended encoding can be shown to be *O(n4)* where n is order of the Sudoku puzzle. (Lynce and Ouaknine) As explained by Lnyce and Ouaknine in their paper “Sudoku as a SAT Problem” the minimal encoding for a Sudoku puzzle is:



By encoding Sudoku as a SAT problem you can use various SAT solving algorithms like the DPLL to solve a Sudoku puzzle.

# Goals of this Project

My goals for this project were to develop an interactive Sudoku solver generalized to handle N by N Sudoku puzzles in addition to creating a powerful and intuitive graphical user interface (GUI) that can be used to compare the efficiency of different Sudoku solving algorithms against a wide array of Sudoku puzzles diffing in size and complexity. Overall, I met these goals by creating X different Sudoku solvers, all of which can handle N by N Sudoku puzzles, though they might not produce a solution in any reasonable time. Each of the solvers can be tested interactively with a GUI I adapted and enhanced from Gilbert Le Blanc. (java articles) While using the GUI you can manually enter in the starting clues, load a puzzle from the database of a given difficulty, or generate a new puzzle allowing you to specify how many clues you want the puzzle to contain. Over the course of this project, I was able to experiment with different design patterns such as the Singleton, Strategy, and MVC design pattern as well as implement features that I did not intend to such as a database to load and save puzzles to. Additionally, my project can be run from the command line without a GUI as a tool to provide data, such as the number of nodes needed to explore and the time it took to find the solution, to analyze the performance of an AI against a large sample of Sudoku puzzles. Overall, my project which spans over X classes for a total of Y lines of code, represents a fun exploration into larger scale software development and an investigation into techniques to solve Sudoku puzzles utilizing artificial intelligence.

# Software Written

## Overview

For this project, I utilized the Model View Controller design pattern to organize my project and create a robust GUI. I have four main packages: model, view, controller, and solvers. The state of the Sudoku puzzle is maintained in the model package comprising of two classes SudokuPuzzle and SudokuCell. The model package also contains the implementation of the Singleton used to interface with the database in addition to other classes and enumerations such as PuzzleEntity, which is class abstracting a puzzle record in the database, and SudokuPuzzleSize, which is an enumeration of supported sizes for Sudoku puzzles encapsulating drawing logic per puzzle size. Most interestingly, the model package contains the implementation of the the puzzle generator whose implementation and performance will be discussed later in this paper.

The controller and view packages contain all the code related to presenting and updating the Sudoku puzzle. These two packages are entirely GUI oriented which was written using Java Swing. If the reader interested in their details they encouraged to view the code themselves via the BitBucket link listed on the cover page of this paper.

The last package is solvers, which is the home to the my implementation of the X different Sudoku solvers. The solvers I implemented are a backtracking algorithm, a backtracking algorithm with forward checking, and OTHER SOLVERS. Here I utilized the strategy design pattern to create an abstract class of a SudokuSolver of which each implementation of solver is forced extend. The three abstract methods in SudokuSolver are the two methods for solving a Sudoku puzzle with and without a GUI to update and a function to return the name of the solver. Each of the solvers will be discussed individually in the blah SECTION.

## The Sudoku Puzzle Abstraction

For my implementation of a Sudoku puzzle, I chose to use a two dimensional array of SudokuCell objects. I did this by having a private field in SudokuPuzzle called *cells* which was a two dimensional array of SudokuCell object. The general abstraction function is that given a puzzle P where *r* denotes the row and *c* denotes the column the value of the cell Pr,c, where *r=0* and *c=0* is the topmost left column, is represented by SudokuCell at *cells[c][r]* whose value is equal to Pr,c. If the value of Pr,c is not known then the value of the SudokuCell at *cells[c][r]* is set to 0. The SudokuCell object also keeps track of whether this cell was a given value in the puzzle as well as encapsulates all the logic necessary to present the cell in the GUI.

Choosing a two dimensional array to represent a Sudoku puzzle was a natural abstraction. This implementation made it very easy and efficient to traverse and interact with the model. However, because I chose to use a two dimensional array I was unable to utilize Donald Knuth’s Dancing Links technique, which calls for an exact cover matrix whose cells are all connected in in doubly linked list. Had I implemented the model as a matrix of doubly linked lists I could have utilized Donald Knuth’s Algorithm X which is an efficient recursive, non-deterministic, depth-first, backtracking algorithm for solving the exact cover problem.

## The Database and Puzzle Generation

Generating a Sudoku puzzle was a surprisingly difficult task. My initial hope was that there would exist a general Sudoku puzzle generator which could generate puzzles of a given difficulty and size. Unfortunately, there does not exist such a tool with Stephen Ostermiller’s qqwing being was the closet thing I could find. Qqwing is a very robust Sudoku generator and solver, however it is limited in that it can only generate 9 by 9 Sudoku Puzzles. One very nice feature about qqwing is that it can generate puzzles of a specified difficulty. This is important because some Sudoku puzzles are inherently harder to solve than others and it is generally difficult to tell if a given puzzle will be trivial or hard solve

My generator could not handle generating a puzzle of size 25 as after I left it running for ten minutes the only results were that my computer was noticeably hotter.