Sudoku Madness

Team 3: Matt Crain, John Cheng, and Rabih Sallman

I. Problem Description

Standard Sudoku is a logic-based puzzle in which the user must fill a 9 x 9 board with the appropriate digits so that each row, column, and individual sub-grids (must be perfect squares) contains the digits 1-9.

Sudoku is essentially a form of a constraint satisfaction problem, therefore we will use methods discussed in literature and lecture to help us find a solution that is both fast and efficient. Some techniques that we may build from are constraint propagation (forward checking, backtracking search, etc.) and local consistency verification(arc, path, etc.).

The standard Sudoku by itself is a very well known problem. It has been solved using just about every possible method, multiple times. Brute force algorithms can solve Sudoku, but aren't efficient in any sense of the term. Constraint satisfaction is exceptionally common and Sudoku almost seems like a mascot for attracting new people to look into constraint programming, since it provides an interesting foundation for constraint solving while being difficult enough to remain interesting. Less conventional means of solving that involve various searching techniques and even genetic algorithms have also been used. In the paper "Stochastic Optimization Approaches for Solving Sudoku", Meir Perez and Tshilidzi Marwala used Cultural Genetic Algorithm, Repulsive Particle Swarm Optimization, Quantum Simulated Annealing and a Genetic Algorithm with Simulated Annealing. These gave varying results, with the Swarm algorithm being incapable of solving the problem and the Genetic Algorithm with Simulated Annealing being the fastest.

II. Detailed Background

Currently, most literature focuses on a very focused limited set of Sudoku, which typically is the standard 9 x 9. Rarely do the problems ever reach even a 16 x 16 grid. What we would like to do is push the upper bound of our program to a much larger grid size, allowing things like 25 x 25, 36 x 36, or even larger grids to be generated and solved. "A search based Sudoku solver" is one paper about using search techniques for solving sudoku that does reference these larger grids and states that there is a phase change when you get to 25 x 25 that causes the problems to become more difficult.

Solving techniques that can be done by hand have been thoroughly flushed out. There's over a dozen different inference techniques that people may use to solve very difficult puzzles. Most, if not all of these have been detailed on SadMan Software.

There's also one thing we haven't noticed much in any of the literature and that's the benefits that multi-threading can provide. In something like a backtracking search it would be very easy to see a thread being spawned for each branch of a backtrack so each branch would be evaluated in parallel. This could be very helpful with larger versions of the problem.

Solving

Solving a Sudoku is quite easy in principle, but very difficult to do with efficiency. The solving algorithm heavily utilizes the constraint network to find solution values for cells and search for solutions if need be. The basis of the algorithm is constraint propagation, so once one value is assigned to a cell that value is removed from the domain of any cells that are constrained by the cell that was assigned. Finding cells to assign values to is done by a simple search through all the cells. The search looks for cells that are most heavily constrained, therefor having a small number of minimum remaining possible values to be assigned to them. The best case is when a search returns cells that only have one possible value, because that is the solution value for that cell. If the search returns cells that have more than one possible value, then a completely different search is performed; a recursive backtracking search.

The backtracking search is effective, being able to solve any Sudoku given infinite time, but slow, so a lot of effort is spent minimizing the amount of backtracking that is performed. Using cells with the minimum remaining values is very important. Typically the minimum is two if a backtracking search is required. It is very important to perform the search on these values that only have two possible values instead of searching on other random cells. The odds on being correct on the first guess are 50%, instead of around 20% if you were to pick randomly. The cells with the minimum remaining value are ordered in a way to try and maximize correctness on the first guess. Each value in the domain is weight based on how many times it has been used already in the Sudoku, the more it has been used the higher the weight. Using these weights the possible cells are sorted in a fashion that places the highest weighted values to be picked first. The highest weighted possible value is assigned to the cell and the search continues. This heuristic is used because the more a specific value is used the lower the total number of remaining cells it can occupy. Typically if a cell contains one of the last possible locations for that value there's a higher probability of being correct if we chose it. If a branch is encountered where there are one or more cells that can't be legally given a value from their domain then the branch is deemed a fairly and the algorithm backtracks to the last branching point.

The main issue with the backtracking search is how many different branching possibilities there are. A fair amount of effort is spent trying to minimize these branches with inference techniques that go beyond what the general constraints of Sudoku allow. Really they can be found within the constraints, however then can't be easily propagated throughout the grid when an assignment is made. One example would be if a subgrid (one of the nine 3x3 squares within a standard Sudoku) requires that a specific value, we'll use '7', must be located within a column of that subgrid. Since the 7 for that sub grid must be within the cells that make up the column, then all the cells in the rest of that column for the entire puzzle cannot contain the number 7 in their domain. This can eliminate up to 6 possible branching positions that the constraints typically do not show.

Arc consistency is a way that allows a fair amount of the information that's deduced through the inference techniques to be flushed out in a more simplified manner and in one centralized process. Most efficient arc consistency techniques are complicated and difficult to implement properly. van Dongen's AC-3d algorithm is efficient, but quite complex. Our chosen algorithm is the AC-3 algorithm which is easy to understand and almost efficient enough to justify using it. This algorithm works great on 9x9, but slows down when the size of the puzzle is increased.

Multithreading was attempted, but had to be cut. It was taking too much time to ensure that everything was properly working. Debugging is very difficult and it was also hard to find out accurately how much effort went into solving a puzzle (e.g. the number of branches explored). Given more time this could accelerate solving in a manner proportional to the number of CPUs available. If you had 16 CPUs you could explore 16 branches simultaneously and asynchronously, because branches aren't dependent on the results of other branches.

Generating

Generating Sudoku puzzles is a task that is completed with three steps. First step is to generate a complete puzzle. Second step is to generate a random permutation from this completed puzzle. Lastly, we remove hints from the puzzle in order to generate a puzzle that is playable.

Generating a complete puzzle is trivial. It can be done manually by inserting ordered rows which are shifted one after another. For example, in a 9x9 Sudoku grid, first row is defined as 1,2,3,4,5,6,7,8,9. The second row is shifted over: 2,3,4,5,6,7,8,9,1,etc... Another way is to allow a Sudoku solver, “solve” a blank puzzle. At the end it will come up with some sort of valid configuration of cell values, depending on the algorithm used.

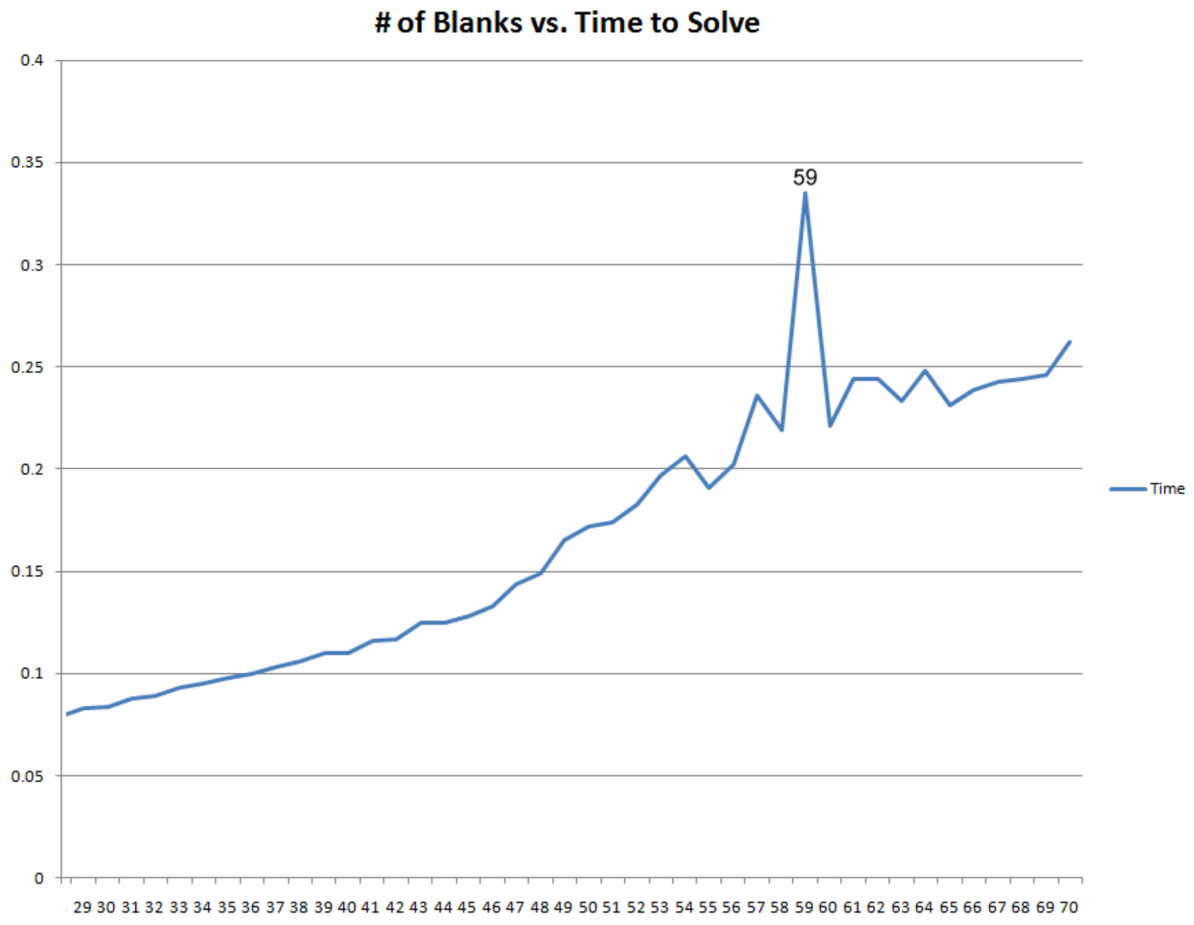
Once the solved puzzle is generated, permutations are applied to “shuffle” the puzzle to create different possible combinations while still maintaining the integrity of the game. Permutations such as reflecting horizontally, vertically, diagonally keep the puzzle valid. Other techniques such as value swapping and row swapping are also valid moves. You can view these in "Enumerating possible Sudoku grids"

Lastly, removing hints from the grid can be done either randomly or methodically. Most Sudoku games written in Javascript for a web browser, or cell phones use random cell removal and the difficulty levels are determined by how many cells are removed. For example, “easy” can be considered removing 47 cells, “medium” 54, and “hard” 59. However, a different method can be used by examining the fact that each row, column, or subgrid has to add up to 1 + 2 + 3 + .. + N where N is the size of the puzzle. Using this property, we can find out how difficult a puzzle is by seeing how many different sums are possible to complete each row, column, or subgrid.

Additionally, hints should be removed symmetrically. For example, if a hint is removed in (0, 0) on a 9x9 grid, position (8,8) should also be removed. This creates a puzzle that looks more balanced to a player and also reduces the chances of generating invalid puzzles by evenly distributing the hints. An easy way to visualize this is to imagine 40 hints being removed from a puzzle, and 3 adjacent subgrids are completely removed. The puzzle is not balanced and multiple solutions can be easily discovered.

The problem with removing hints from a completed puzzle is the introduction of possible invalid puzzles, where the puzzles have multiple solutions. This problem is solved by finding out whether or not a puzzle is unique, if it is not, generate a new puzzle. To check if a given puzzle is unique, for every blank cell, every possible value is assigned systematically and then checked to see how many solutions can be found this way.

The following table displays the amount of time needed to find a solution vs. the amount of blanks in the current puzzle. Each point on the line is the average from 100 trials at each number of blanks. A total of 7,000 samples were taken. The front part of the graph is left out because the time is very predictable. Any larger number of blanks than is shown produces Sudoku that have thousands of possible solutions making them trivial to solve.



III. Tools

We used the Java programming language as well as the Eclipse IDE for Java Development to complete the coding aspect of our project. We are using Google Code to host our project and enable more efficient code coordination through its version control client. All the code was created by our team and there is no previously made code in our project. The GUI was created by our team as well using the Swing toolkit for Java. We are only using third party tools to test and evaluate our program. For example, we are using Google Documents to host a spreadsheet that features our test results such as test case success and speed. Finally, we are currently using two third-party Sudoku programs to compare our own program with. One such program, *SuDoku Solver*, allows us to input a text case and in return will output the time it takes for the quickest solution to be found. The second program, *The Ultimately Fast Sudoku Solver*, is a browser based solver that also returns the time it takes for the solution to be found, however, the input must be done by hand.

IV. Data

We can generate our own set of numbers/solutions for every game that is going to be played. The difficulty of each test can be decided by the user and is handled by our generating algorithms discussed in the previous sections. We are also using previously generated puzzles (varying in difficulty) from *WebSudoku.com* and converting them into text files so they can be used by our program for solving/testing and any third-party programs that we may use for evaluation.

V. Input/Output

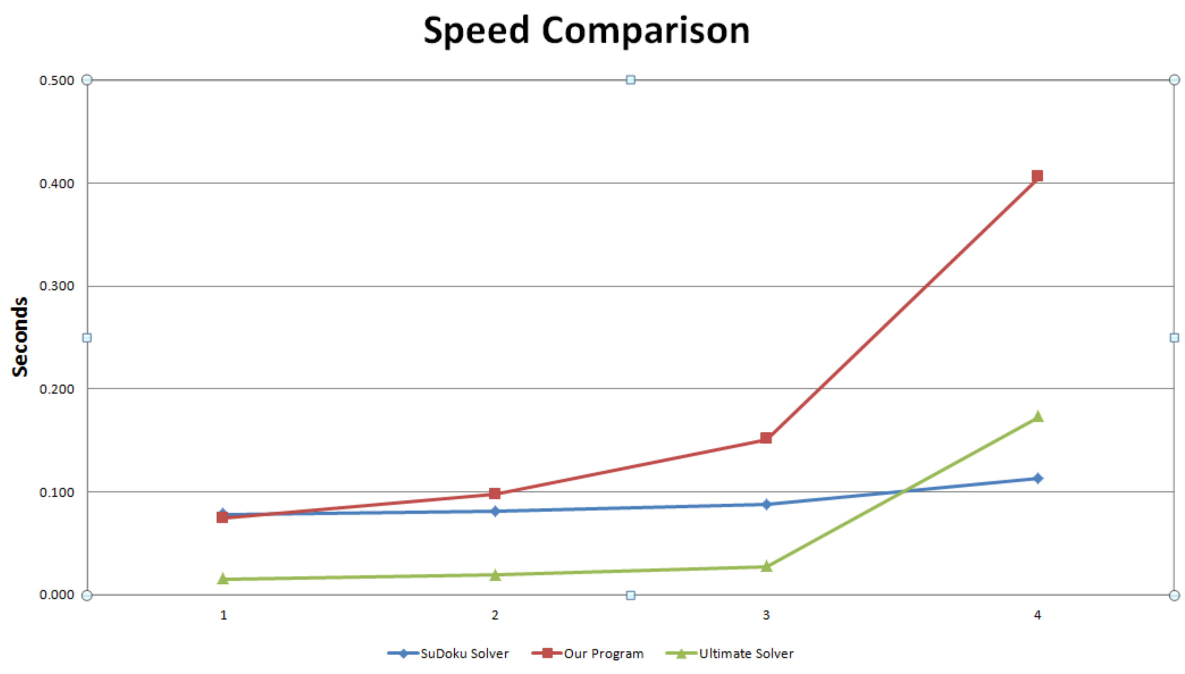
Sudoku Madness can input a previously generated puzzle (text file) and can also handle input directly through the GUI to create a Sudoku or solve an existing Sudoku. The text file contains the numbers 1through N with a “0” or “_” indicating a blank space on the Sudoku grid. The program is set to output the solution of the current Sudoku puzzle, the time for the solution to be found, and the depth of the solution.

VI. Summary and Evaluation

Sudoku Madness is complete with a GUI, puzzle generator, solving generator, and a test suite. The program can generate a Sudoku puzzle from user input through the GUI or by importing an already existing Sudoku puzzle. The program can also generate a Sudoku puzzle based on the the generating algorithms discussed in the previous sections, and can solve the current Sudoku based on the solving algorithms discussed in previous sections. Most importantly, the program is capable of solving any difficulty of Sudoku puzzles instantaneously.

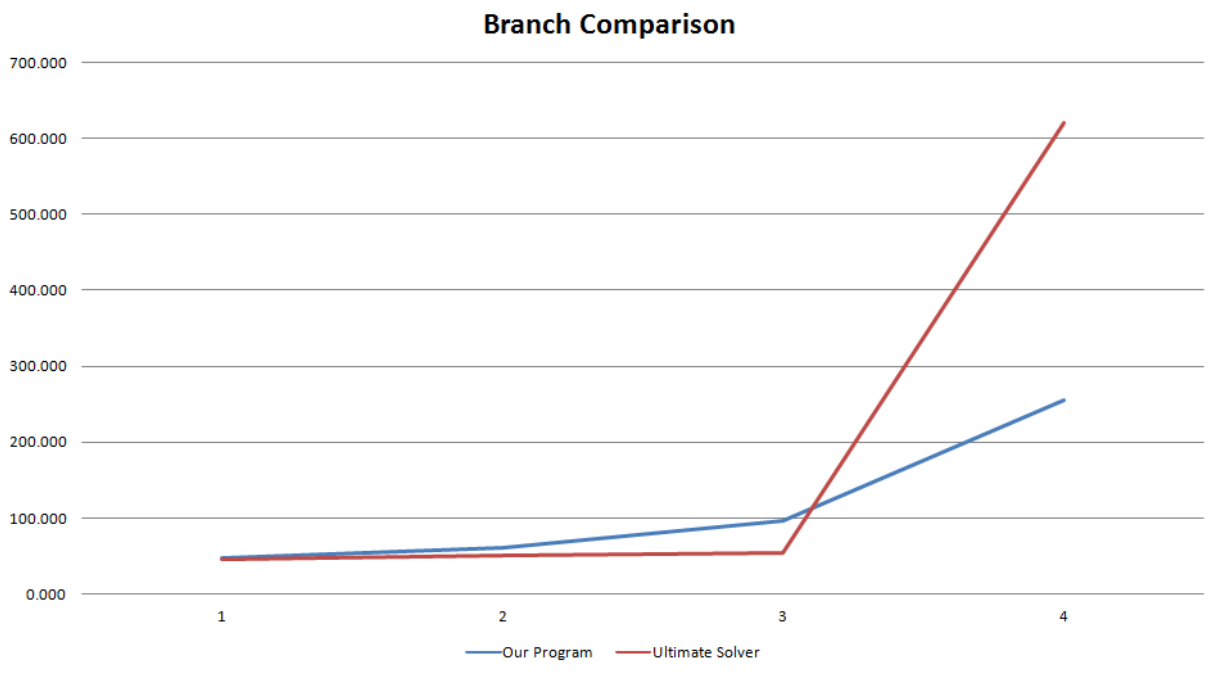
The following table displays the solving speed of Sudoku Madness compared with *SuDoku Solver* and *The Ultimately Fast Sudoku Solver* aka *Ultimate Solver.* The x-axis is based on the 4 difficulty levels of Sudoku puzzles - Easy, Medium, Hard, and

Evil. The data points represent the average solving time needed for each difficulty level (Easy: 15 tests, Medium: 15 tests, Hard: 20 tests, and Evil: 30 tests)



Our program performs extremely well for each level of difficulty, even solving easy Sudoku puzzles faster than *SuDoku Solver*. Although our program is slightly slower when solving hard or evil Sudoku puzzles, the average solution time is still instantaneous.

The following table displays the amount of branches searched before a solution was found for each difficulty level.



The amount of branches needed to find a solution for the easy or medium test cases remains right in line with the *Ultimate Solver* until we reach the the two more difficult test groups. Our program searches an average of 50 more branches before finding a solution for hard test cases, however, our program completely outperforms the *Ultimate Solver* in the evil Sudoku tests. Our program searches 60% less branches than the competition when solving the most difficult Sudoku puzzles available.

For larger Sudoku puzzles such as 16x16, 25x25, and 36x36 the program requires much more time and patience for a solution to be found. Our program takes anywhere between 7 seconds to 1 minute to solve a 16x16 Sudoku with an easy or medium difficulty. The larger and more difficult the grid became increased the solving time greatly (beyond 5 minutes) and therefore was not comparable to any third party program we could find. Although most of our larger Sudoku puzzles were not solved within our predetermined time limit, the success we experienced when solving 9x9 Sudoku puzzles assures us that a correction solution will be found.

VII. Separation of tasks

Matt Crain - Completed the GUI, Solving Algorithm, and helped test generation and solving capabilities. Also researched and wrote about the solving algorithms included in our submitted reports. Collected the rest of the groups information to create a slide show for our midterm and final presentations.

John Cheng - Completed the Generating Algorithm and helped test generation capabilities. Also researched and wrote about the generating algorithms included in our submitted reports and in our slide show presentations.

Rabih Sallman - Completed the evaluation portion of our program. Created test Sudoku files and an excel worksheet to compare our program with third party programs. Also wrote and assembled the submitted reports and contributed to the slide shower presentations.

VIII. User Manual for Sudoku Madness

i) How to compile and execute the program (Windows OS)

Step 1: Download Sudoku.zip from EEE and save to a directory of your choice. Unzip the contents when the download has finished.

Step 2: Open command prompt by typing cmd in Run (located in the Start Menu for Windows XP) or the Start Menu search bar (Windows Vista/Windows 7). Once the command prompt has displayed, switch the current directory to the location of the program. To switch directory, entering the command **cd *location***; *location* is the path to the directory containing the source files.

Step 3: Once in the appropriate directory, compile the .java files by entering the command **javac *.java** and run the compiled program by entering the command **java -Xmx512m TestMain TestMain.java** – the program's GUI will appear momentarily.

*Xmx512m increases the Java heap space for the very large Sudoku. 512m is a lower bound for large Sudoku. If you want to try anything with the larger puzzles (25x25 or 36x36) I suggest you allocate more memory for it (e.g. 1024m or 2048m if you have it available).

ii) How to use the GUI

Menu Bar

The following is a short description of each menu located on the menu bar.

File	The user may clear the current grid, import a Sudoku puzzle, or export the current puzzle to a text file
Solver	Where the user can select to solve the current Sudoku puzzle.
Generation	The user can generate a new game based on difficulty – Solved, Easy, Medium, or Hard. Then can also test to see if the current Sudoku is unique.
Generalize	Where the user can select the size of the Sudoku grid – 9x9, 16x16, 24x24 or 36x36.
Testing	Used to evaluate the program with existing Sudoku files.

Sudoku Grid

The GUI will always display the current Sudoku puzzle for a simpler and more efficient user experience.

Types of Grids

Generalized: The GUI will display the grid with the coordinates of each individual box.

Blank: Displayed after the user selects to clear the grid.

Partially Solved: A grid in which some, but not all the values have been placed.

Solved Grid: A fully complete grid filled with legal values in each individual box and sub-grids.

How to change values

The user is allowed to change the value of each individual box by selecting that box and choosing the value from the drop down menu. Any illegal value will appear red and will have to be changed.

Importing Sudoku puzzles

The user may also import a Sudoku puzzle from the file menu. The .txt file must be in accordance with the formatting rules discussed in the Input/output section.

Exporting the current Sudoku puzzle

The user may also export (save) the currently displayed Sudoku puzzle from the file menu. The resulting .txt file will be formatted in accordance with the formatting rules discussed in the Input/output section.

Testing

The testing menu provides a variety of ways to test the program. Almost all of them focus around the test Sudoku files that are provided with the code. These are labeled with their respective difficulties: easy, medium, hard, and evil. When you choose these options you are prompted for two different files. The first file will be the location and name of the file that the test results are saved into. The second file is one of the test files that you will be testing; if you are testing evil then you will browse and find evil1 which should be grouped with all other evil difficulty files. This will tell the program where all the files are located so they can be iterated through.

The only test that doesn't work like this is the super test. The super test will prompt where you want the results to be saved. The program then generates a variety of test Sudoku with different numbers of blanks, starting from 1 up to 70 and it generates 100 test cases for each number of blanks. This means a total of 7000 Sudoku will be generated and solved. This test assumes a 9x9 Sudoku configuration.

iii) Program Behavior

Sudoku Madness can display generating or solving results on the command prompt or can save the information in a text file if running through a test suite. When a Sudoku grid is generated by the program, the number of permutations through the algorithms discussed earlier will be displayed to the user.The program's ability to solve will be judged by its solving time and the number of branches it went through before finding the solution. While a complete solution will be displayed on the GUI in the form of a solved grid, the information displayed on the command prompt can be useful for program evaluation.

If a puzzle was generated the following will be displayed on the command prompt:

Permutating through: ___ cycles

If a solution was found the following will be displayed on the command prompt:

"Branches: _____
Solved in _____ seconds"

*With the actual results in the place of the blanks.

When testing for a puzzles uniqueness the program will print all separate solutions found to the command prompt. Then when all solutions have been found it will popup a message saying it's done and how many. A limit is set at 1000 solutions before it stops itself. Otherwise it could be searching for solutions until the end of time.

Wait Time and Solving Time Limit

Processes such as generating and solving larger and more difficult puzzles, generalizing larger grids, and testing may require longer waiting times than usual. For solving, the program has a default time limit of 5 minutes. This value may be changed by modifying "Constants.java" with a text editor and changing the TIME_LIMIT variable.

Default is set at 5 minutes:
public static int TIME_LIMIT = ***5*** ** 60 * 1000;*

To change the time limit to 10 minutes change the variable to:
public static int TIME_LIMIT = ***10*** ** 60 * 1000;*

Recompile and run the program after the changes have been saved.

IX. References

1. Simon Armstrong: SadMan Software: http://www.sadmansoftware.com/sudoku/solvingtechniques.htm
2. Meir Perez and Tshilidzi Marwala "Stochastic Optimization Approaches for Solving Sudoku": http://arxiv.org/ftp/arxiv/papers/0805/0805.0697.pdf
3. Tristan Cazenave "A search based Sudoku solver": http://www.ai.univ-paris8.fr/~cazenave/sudoku.pdf
4. The AC-3 algorithm: http://en.wikipedia.org/wiki/AC-3_algorithm
5. Bertram Felgenhauer "Enumerating possible Sudoku grids": http://www.afjarvis.staff.shef.ac.uk/sudoku/sudoku.pdf
6. M.R.C. van Dongen "AC-3d an Efficient Arc-Consistency Algorithm with a Low Space-Complexity": http://www.springerlink.com/index/017mrfcnpb1qjx19.pdf