Genetic Algorithms and Sudoku

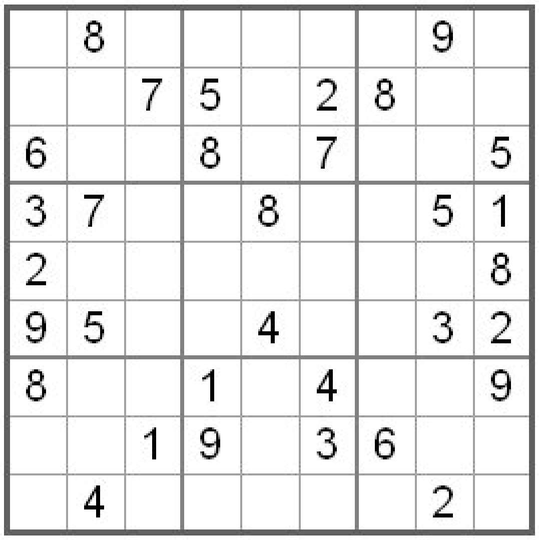
Dr. John M. Weiss
Department of Mathematics and Computer Science
South Dakota School of Mines and Technology (SDSM&T)
Rapid City, SD 57701-3995
john.weiss@sdsmt.edu
MICS 2009

Abstract

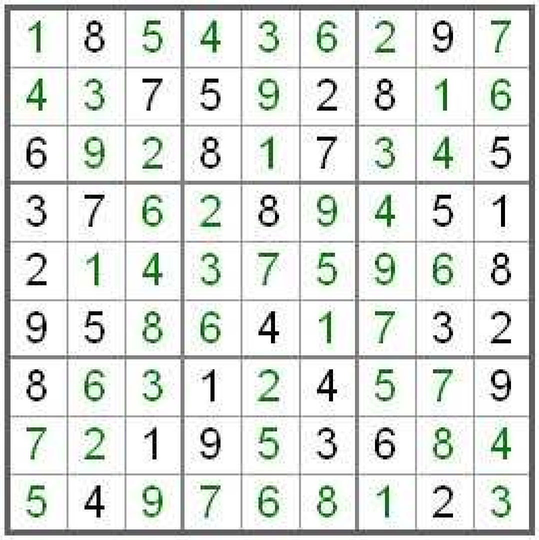
Sudoku is a number placement puzzle that has achieved remarkable popularity in the past few years. Traditional generate-and-test solution strategies work very well for the classic form of the puzzle, but break down for puzzle sizes larger than 9x9. An alternative approach is the use of genetic algorithms (GAs). GAs have proved effective in attacking a number of NP-complete problems, particularly optimization problems such as the Travelling Salesman Problem. However, GAs prove remarkably ineffective in solving Sudoku. GA performance in Sudoku suffers from slow convergence, inability to escape from local minima, and a variety of other problems. Our experience with Sudoku suggests that this is a GA-hard problem.

Introduction

Sudoku [1] is a number placement puzzle that has achieved remarkable popularity in the past few years. In its classic form, the objective is to fill a 9x9 grid with the digits 1 to 9, subject to the following constraints: each row, each column, and each of the nine 3x3 subgrids must contain a permutation of the digits from 1 to 9. The initial puzzle configuration provides a partially completed grid, such as the example grid illustrated below left. The Sudoku solver must fill in the remaining grid elements to obtain a solution like the one given on the right.



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



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

Sudoku is a special case of the Latin Square [2], with an additional constraint imposed by the permutation of digits inside each subgrid. Sudoku may be generalized to allow grid sizes other than 9x9. The most convenient sizes are perfect squares (4,9,16,…), since these contain square subgrids (2x2,3x3,4x4,…), but other variations (such as a 12x12 grid with 4x3 subgrids) are possible. In general, an *NxN* grid is filled with the digits 1 to *N*.

The standard approach to computer-based Sudoku solvers is a generate-and-test solution using backtracking [3]. This strategy works extremely well for smaller grid sizes (including the standard 9x9 grid configuration), but becomes computationally intractable for larger sizes. In fact, the general problem of solving *NxN* Sudoku puzzles is known to be NP-complete [4].

An alternate solution strategy for Sudoku puzzles uses a *genetic algorithm*. Genetic algorithms (GAs) [5,6,7,8,10,11] are a class of evolutionary algorithms that achieved popularity through the efforts of John Holland in the 1970's [7,8]. GAs are based on the fundamental concepts of evolution in biology. The classic GA starts with a random population of initial solutions, encoded as bit strings. These strings are ranked by a fitness function that determines the goodness of a solution. Better solution strings are selected and recombined using a process analogous to crossover in DNA recombination: the first part of one bit string is concatenated with the last part of a second bit string. A low level of mutation is then applied, by randomly flipping solution bits. The new solution strings, together with the most fit individuals from the previous generation, comprise the population in the next generation. The process continues until a good enough solution has been generated, or no further progress can be made.

Many variations are possible, and some are necessary for problems such as Sudoku. The bit-level encoding of solution strings may need modification to account for higher structure in the data. In Sudoku, for example, it makes sense to represent each subgrid (or row, or column) as a permutation of the digits 1..9. Crossover points exist only between subgrids, not at each bit in the solution string. Mutations should not randomly flip bits, but should generate a different permutation of digits within a subgrid. Predetermined positions (either given in the initial configuration, or with only one possible digit) should not be allowed to mutate.

Implementation

A computer program was written in C++ to solve Sudoku using a GA. This program reads in an initial grid configuration from a text file, given as a command-line argument. Genetic algorithm parameters are specified as optional arguments, including the number of initial solution strings, the maximum number of generations, percent of current population used for reproduction, and mutation rate. Program usage permits command-line specification of an input file that contains the Sudoku puzzle to be solved, and various GA parameters (population size, maximum number of generations, selection rate, mutation rate, etc.).

The input file format specifies the grid size followed by the initial grid configuration. Hyphens indicate blank grid elements, and white space (space, tab, new line) separates all text file components.

For the sample Sudoku problem above, the input file would contain:

```
9 9

- 8 -   - - -   - 9 –
- - 7   5 – 2   8 - -
6 - -   8 – 7   - - 5

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

8 - -   1 – 4   - - 9
- - 1   9 – 3   6 - -
- 4 -   - - -   - 2 –
```

The GA runs until a solution is found, or the maximum number of generations is reached. The best (minimum) and worst (maximum) cost function results are output for each generation. The final grid configuration is printed, using a format similar to the input file format, when the program completes. Incomplete solutions are indicated by placing asterisks at the end of each row or column to indicate duplicate digits.

Solution Outline

Filling in predetermined cells

Some initial positions specify a unique Sudoku solution, while others do not. As few as 19 cells on a 9x9 grid may be sufficient to uniquely specify a solution. In general, some of the cells may be filled in using simple techniques that do not involve search. For example, if 8 cells (out of 9) in a row, column, or subgrid are filled in, the remaining cell is obviously predetermined. For easy Sudoku puzzles, this approach may be sufficient for finding a solution. Even for harder puzzles, it is worth filling in predetermined cells such as these, in order to reduce the dimensionality of the problem.

With this in mind, there are two phases to solving Sudoku using GAs. The first phase fills in predetermined cells, and may partially or completely solve the puzzle. The second phase starts with the filled-in grid, and attempts to evolve a complete solution to the puzzle.

Only two simple approaches were employed to detect predetermined cells; in particular, filling cells “hidden” and “naked” single cells. This is most commonly done using a “pencil marking” approach:

```
each cell has a 9-element bool array to track available digits
initially each cell may contain any digit (all true)
for each  cell (at row r, column c, subgrid s) in the puzzle
{
        if the cell contains digit d
        {
                cross off (set to false) the digit d for each cell in row r
                cross off (set to false) the digit d for each cell in column c
                cross off (set to false) the digit d for each cell in subgrid s
        }
}
```

Cells that contain only one available digit (so-called “naked” singles) may now be filled in with that digit. Cells that contain a digit that appears only once in the row, column, and subgrid (“hidden” singles) are also filled in. When grid cell(s) are filled in, the digit arrays must be updated, and the filling in process repeated.

No attempt was made to employ more sophisticated algorithms that consider pairs or triplets of cells to fill in the puzzle grid. At this point the GA was allowed to work its magic.

Running the GA

Evolutionary algorithms encompass a variety of biologically-inspired problem solving approaches. Genetic algorithms encode a population of solutions, rank them with a fitness function, and apply computational analogues of selection, recombination, and mutation to evolve better solutions. GAs have been successfully employed to solve a wide variety of optimization problems, and from a naïve perspective, it would appear that Sudoku is an ideal application for this approach.

- **Encoding**: Solution strings are encoded as 1-D arrays (or lists) of symbols. By limiting the maximum grid size to 256x256, the symbols can be represented with 1-byte chars. Each solution string for an *nxn* grid is broken into *n* segments of *n* symbols, with each segment representing one subgrid. For a 9x9 grid, each 3x3 subgrid is represented with a 9-element array, and there are 9 of these subgrid arrays.

- **Initial Population**: For each initial solution string, a random permutation of the digits from 1 to *n* is placed in each segment. This ensures that the subgrids (at least) are correct. Some additional bookkeeping is required to handle the fixed elements in the initial grid. The simplest approach is to go ahead and randomize the digits, then swap the fixed elements back into their correct places.

- **Fitness Function**: Solution strings are evaluated by counting the number of duplicate symbols in rows or columns. Fewer duplicates presumably means a better solution string.

- **Selection**: The entire population of solution strings is ranked by the fitness function, and a certain percentage is selected for survival and reproduction. The best solutions are selected probabilistically, so that even lower-ranked solutions have a small but finite chance of reproducing.

- **Genetic Operators**: crossover may take place at any point between subgrids. For a 9x9 Sudoku grid, there are 8 possible crossover points. Mutation causes two elements of a subgrid to swap positions. Fixed and predetermined elements may not be swapped out of position.

- **Evolutionary Algorithm**:
```
generate initial population
repeat
        rank the solutions, and retain only the percentage specified by selection rate
        repeat
            randomly select two unused solution strings from the population
            randomly choose a crossover point
            recombine the solutions to produce two new solution strings
            apply the mutation operator to the solutions
        until a new population has been produced
until a solution is found or the maximum number of generations is reached
```

- **Restart Heuristics**: The GA is restarted with a new set of random solution strings if it gets stuck in a local minimum. This is controlled by a “restart” command-line parameter. When the best fitness value does not change for “restart” generations, the GA starts over from scratch. The top few solutions from each generation are stored after each restart. When enough top solutions accumulate, a new population is created from these best solutions, and used as an initial population when the GA is restarted.

- **Reducing Search**: In addition to fixed digits that are predetermined by the initial position, it is often possible to determine other cells that can only have one possible digit. At the very least, it makes sense to fill in grid elements that are constrained by the initial position to contain only one possible digit.

Results

Results of running the GA on Sudoku were disappointing, at least in the sense of obtaining an optimal solution in a reasonable amount of time. A naïve implementation did not always solve a given Sudoku problem. Allowing longer run times (more generations) did not usually change matters. Once stuck in a local minimum, the GA was seldom able to jump out and find the best solution. Restarting the GA when it got “stuck” often worked, although it might take many restarts before getting the correct solution. An “optimal” restart strategy, using the best solutions from a number of generations, had a better chance of finding the correct solution, but did not always do so. In general, changing the fundamental GA parameters (population size, selection rate, and mutation rate) had a profound impact upon performance, but it was not obvious how to generalize results from one problem instance to the next. As expected, larger populations of solutions were helpful.

This problem was given as a programming assignment in the Artificial Intelligence class in Spring 2009 at SDSM&T. The best student results will be presented at MICS 2009.

Here is sample output from a successful run of the program:

```
Sudoku: med3.txt
population size = 10000
number of generations = 1000
selection rate = 0.5
mutation rate = 0.05
restart threshold = 100

Initial configuration (9x9 grid):

 - 1 -   2 - 5   - 8 -
 - - 3   - - -   4 - -
 2 - -   - 7 -   - - 3

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

 4 - -   - 6 -   - - 9
 - - 2   - - -   8 - -
 - 6 -   9 - 1   - 4 -

Filling in predetermined squares:

 - 1 4   2 3 5   - 8 -
 7 - 3   1 8 -   4 - 2
 2 8 -   4 7 -   - - 3

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

 4 - -   8 6 -   2 - 9
 - - 2   - - -   8 - -
 - 6 -   9 2 1   - 4 -

Generation    0: best score =   15, worst score =   43
Generation    1: best score =   15, worst score =   41
Generation    2: best score =   15, worst score =   37
Generation    3: best score =   13, worst score =   35
Generation    4: best score =   13, worst score =   35
Generation    5: best score =   11, worst score =   33
Generation    6: best score =   11, worst score =   33
Generation    7: best score =   10, worst score =   33
Generation    8: best score =    9, worst score =   30
Generation    9: best score =    9, worst score =   30
Generation   10: best score =    8, worst score =   29
        . . .
Generation   25: best score =    2, worst score =   20
Generation   26: best score =    2, worst score =   23
Generation   27: best score =    2, worst score =   18
Generation   28: best score =    2, worst score =   17
Generation   29: best score =    2, worst score =   13
Generation   30: best score =    0, worst score =   15

Best solution:

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

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

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

Discussion

Why is Sudoku so easy to solve via conventional backtracking algorithms, but seemingly so intractable for GAs? It is possible that Sudoku is inherently a GA-hard problem. The theory of GA-hard problems is not well understood, but other researchers have also found that GAs are inefficient at solving Sudoku [9]. The nature of the problem, in which solution states occupy discrete quanta rather than a continuum of possibilities, undoubtedly makes it harder to optimize. The search space is likely to have a high density of local minima, at least with the naïve fitness function used in this study. A better fitness function would introduce finer granularity between solutions, and permit optimization to proceed more effectively.

Conclusions

A GA-based Sudoku solver is discussed in this paper. The GA approach performs poorly in relation to standard backtracking algorithms, at least for a 9x9 grid. By choosing large enough population sizes, and reasonable settings for other GA parameters, it is able to solve most Sudoku problems. However, the Sudoku solver oftens gets stuck in local minima and requires restarts in order to successfully find the problem solution. Reasons for poor GA performance with Sudoku are perplexing, since this appears to be an optimization problem that is well suited for evolutionary strategies.

Nonetheless, Sudoku proved to be an entertaining (and challenging) way to introduce GAs in an Artificial Intelligence class. Students enjoyed the assignment, and were generally successful in getting it to work properly.

References

[1] http://en.wikipedia.org/wiki/Sudoku

[2] http://en.wikipedia.org/wiki/Latin_square

[3] http://en.wikipedia.org/wiki/Algorithmics_of_sudoku

[4] http://en.wikipedia.org/wiki/Mathematics_of_Sudoku

[5] http://en.wikipedia.org/wiki/Genetic_algorithm

[6] Goldberg, David. *Genetic Algorithms in Search, Optimization, and Machine Learning.* Addison-Wesley, 1989.

[7] Holland, John. “Genetic algorithms”, *Scientific American*, July 1992, pp.66-72.

[8] Holland, John, *Adaptation in Natural and Artificial Systems*, MIT Press, 1992.

[9] Timo Mantere and Janne Koljonen, “Solving and Rating Sudoku Puzzles with Genetic Algorithms”, New Developments in Artificial Intelligence and the Semantic Web, Proceedings of the 12th Finnish Artificial Intelligence Conference, 2006.

[10] Mitchell, Melanie. *An Introduction to Genetic Algorithms.* MIT Press, 1996.

[11] Riccardo Poli, William B. Langdon, Nicholas Freitag McPhee, *A Field Guide to Genetic Programming,* Lulu Enterprises, UK Ltd, 2008.