CSC447/547 Artificial Intelligence - Spring 2014
Programming Assignment #1: Evolutionary Algorithms

Sudoku is a number placement puzzle that has achieved remarkable popularity in the past few years. 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 may be generalized to allow other grid sizes. The most convenient sizes are perfect squares (4,9,16,…), since these contain square subgrids (2x2,3x3,4x4,…). In general, an *NxN* grid is filled with the digits 1 to *N*.

Write a program in C++ to solve Sudoku puzzles using a *genetic algorithm*. The initial grid configuration is specified in a text file that must be given as a command-line argument. Genetic algorithm parameters may be 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. General program usage is

```
sudoku filename population_size generations selection_rate mutation_rate ...
```

where *filename* is a string that specifies the input text file. Only the first argument is required. Default values for the optional arguments might be:

```
population_size:   100
generations:      1000
selection_rate:   0.50  (50 percent)
mutation_rate:    0.05   (5 percent)
```

The input text file format specifies the grid dimesions (row and column size, both of which must be perfect squares) followed by the initial grid configuration. Hyphens indicate blank grid elements, with white space (space, tab, new line) separating all text file components.

For the example above, the input text 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 -
```

Run the GA until a solution is found, or the maximum number of generations is reached. Print the GA parameter settings, the initial configuration, and the results of filling in any predetermined squares. Output the best (minimum) cost function result for each generation. Print the final grid configuration when the program completes, using a format similar to the input file format (see sample output below).

Solution Outline

- **Encoding**: solution strings are encoded as 1-D arrays (or lists) of symbols. If we limit the maximum grid size to 256x256, the symbols can be 1-byte chars. Each solution string for an *NxN* grid is broken into *N* segments of *N* symbols, each segment representing one subgrid.

- **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. Perhaps the simplest approach is to randomize all the digits, then swap the fixed elements back into their correct places.

- **Fitness Function**: solution strings may be evaluated by counting the number of duplicate symbols in rows or columns. Fewer duplicates presumably means a better solution string. (For extra credit, develop a more sophisticated fitness function.)

- **Selection**: rank the solutions and select the best 50% (or whatever the *selection_rate* parameter specifies) for survival and reproduction.

- **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. Again, be careful not to swap fixed elements 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

- **WARNING!** Sudoku may be an EA-hard problem. You may need to use larger population sizes and higher mutation rates to solve more challenging puzzles. Extra credit will be given for programs that handle more difficult cases, generalize to *NxN* grids, produce nicely formatted output, etc.

- Experiment with different fitness functions and different values for the GA parameters. Consider restarting the GA with a new set of random solution strings if it gets stuck in a local minimum. Another possible heuristic: store the top few solutions from each generation. When you have accumulated enough "best" solutions (local minima), create a new population from them and restart the GA. Summarize your explorations in the program header comment.

- Your solution need not be a "pure" evolutionary algorithm. Start by filling in grid elements that are constrained by the initial position to contain only one possible digit (predetermined squares).

Handing in your assignment

When you are finished writing, testing, and debugging your program, submit your source code in a *zip* or *tar* archive using the *Submit It!* link on the MCS Department Website. Usage is self-explanatory: enter your name, choose the instructor (Weiss) and click "Select Instructor", choose the appropriate course (CSC447), browse to the filename you wish to submit, and click "Upload". If you have any problems with submission, report them to your instructor by email, and submit your program via email attachment.

Submit only your source code, puzzle files, and documentation (not object files or executables). To simplify recompilation, be sure to include a Makefile. Your program must be submitted by midnight on Monday February 24 in order to receive credit. Late programs will not be accepted for partial credit, unless prior arrangements have been made with the instructor.

Your code should be readable, modular, well documented, and should compile and run correctly under the GNU C++ compiler in the Linux Lab (g++). If your program does not run correctly, try to indicate why. This will make it easier to give you partial credit.

You are encouraged to work in teams of 2-3 students on this assignment. Teams should make one joint submission, not individual submissions for each team member.

Sample Output

```
Sudoku: puzzle.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 -
 - - 3   1 8 -   4 - 2
 2 - -   4 7 -   - - 3

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

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

Generation    0: best score =   18, worst score =   47
Generation    1: best score =   18, worst score =   45
Generation    2: best score =   15, worst score =   39
                    ...
Generation   27: best score =    2, worst score =   19
Generation   28: best score =    2, worst score =   18
Generation   29: 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
```