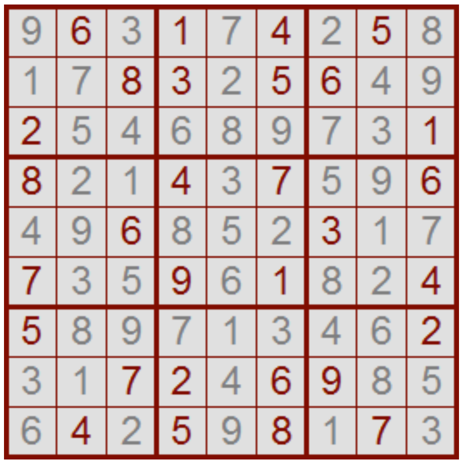
CSE496/896 AI and Heuristics in Software Engineering
Homework I
Handed out: Friday, January 15th
Due: Friday, January 29th, at 10PM
Late Assignments will be accepted until Sunday, January 31st at 10PM. A 10% late deduction will apply until Saturday at 10PM and 20% if submitted by Sunday at 10PM.

There are two pieces to this assignment. You will implement an algorithm (2 variants) to solve a small problem, and answer a set of questions related to this. ***You must complete and submit both parts.***

The popular game "Sudoku" (http://www.websudoku.com/) asks a person to fill in a 9x9 square matrix with numbers ranging from 1..9. The board is initially filled in at the start (and then a set of numbers are removed for the player to fill in). The **completed matrix** has the following properties:

1- Each row and each column contains every digit (1..9) exactly one time.
2- If you divide the grid into nine 3x3 sub-matrices where the columns/rows defining the sub-matrices are 1..3,4…6,7..9 (using a 1 based numbering system), then each of these sub-matrices also has the property that each of the digits 1..9 occur exactly once as well.



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

A sudoku square is a special type of *Latin Square*, a mathematical structure that has been well studied. It is an *N x N* matrix that contains each of the values {1,2…,N} **exactly once in each row and column of the matrix.**

In this assignment we will use a *heuristic search* to generate "**Sudoku Latin Squares**". We will then extend this program to solve a Sudoku problem from a given input.

Part I: Programming Assignment:
Base Program:
Write a ***Hill Climbing Algorithm*** to search for Sudoku Latin squares. Your program should use a random starting point each time (i.e. different runs will produce different squares). If you want to add reproducibility then you can use a random seed as an input parameter to your program. Your program should **output the original starting square and the final Sudoku Latin square,** as well as the **number of iterations** that your program required to find each solution and any other parameters you use (see the experimental section).

Extended Program:
After you have completed the experimental part of this assignment (it will be easier to tune your algorithm first and then complete this part), extend this implementation to complete a real Sudoku problem. Your program should take an input file (I will provide sample input files) with spaces (and line) delineated matrices. The values will be 0 if the cell is blank and a number (1…9) if there is a number already filled in. Use this as a starting point for creating your Sudoku square. The final square must contain the original input pattern as well as be a correct Sudoku Latin square. If I give you a matrix with all 0's this is equivalent to the first part of the assignment. Several example inputs will be posted in blackboard for you to work with.

Example input from above:

```
0 6 0 1 0 4 0 5 0
0 0 8 3 0 5 6 0 0
2 0 0 0 0 0 0 0 1
8 0 0 4 0 7 0 0 6
0 0 6 0 0 0 3 1 0
7 0 0 9 0 1 0 0 4
5 0 0 0 0 0 0 0 2
0 0 7 2 0 6 9 0 0
0 4 0 5 0 8 0 7 0
```

Guidelines/Help:

- You may want to fill in the square to start with so that all of the rows or all of the columns have every digit occurring one time. You can also start completely randomly, but this is up to you. For the rest of this document I will assume you have filled the columns in this manner. Using this method as an initial starting point will mean that you only need to keep track of the rows and grids in your fitness function.

- Your **transformation function** must be random (i.e. you randomly select the new neighbor to try). You can manipulate single cells but you may also find that a swap works well. For instance you can randomly select a column and two rows. Then swap the values in that column between the two rows as your transformation. This will keep your square consistent (i.e. all of the numbers will occur the correct number of times in the columns of the square).

- You should incorporate the possibility of allowing a bad move with a very small probability. This feature will be used in the second part of the assignment.

- Your program needs to satisfy the functionality and clearly show that it does. I am not interested in a fancy/GUI interface, **but** your program should clearly print out a solution when it is found or state if one is not found. I will be running all programs so please make it easy for me to test your program and **see** that your program works.

- You must include a README file with your program giving directions for compiling and running. List any known issues (i.e. if it isn't working properly, doesn't always converge, etc.)

- Your program must run on the "cse.unl.edu" machine. It can be written in any language that exists on cse as long as I have clear directions for compiling/running. I will start from your source code, use your readme (or Makefile/Antfile etc.) to compile and run the program.

- For this program you do not need to spend time working on efficiency. I ask a question about this below, but you do not have to implement this in your code (i.e. you can recalculate fitness each time if it is easier).

- Your program should be written in a modular fashion with sufficient comments so that it is easy to read/comprehend. I will look at your code to make sure that the answers are not hard coded and your approach makes sense.

Part II: Experiments

1) List your ***fitness function*** and your ***transformation function*** *(you may need to explain how you represent and initialize the array here)*. Provide an example of a single transformation and state what the neighborhood is for each solution during a single transformation. Also provide an example of calculating the fitness. Make sure to tell me if you are minimizing or maximizing.

2) What is your ***stopping criteria***? Explain why you have selected this particular stopping criteria.

3) Describe an efficient way of re-calculating the cost for each potential move. You should not recalculate the cost from scratch every time since this is computationally inefficient. Instead you should maintain data structures where needed so that it is easy to figure out how a single transformation affects your cost (i.e. the delta change from the current cost). ** You do not need to implement this, but must show me what you plan to do**

4) For this question, you will do some experimentation.

a. Change the algorithm so that you move only when the solution is better (i.e. your fitness is lower – assuming you are minimizing). Then try moving when the solution is the same or better (i.e. when you have the same or lower fitness). Do this at least10 times for each variant and record the number of iterations required to find a solution (or mark a run as a failure if you can't find a solution in a reasonable time).
b. Fix the version from (a) that works better. Then experiment with the probability for making a bad move. When you have a solution that has higher (worse) fitness, you will move anyway based on a coin flip. Try making the probability .000 .01, .001, and .0001. For each of these, record the number of iterations the algorithm takes to converge.
c. Try part (b) with the other option from (a).

Write a summary of these experiments. Describe what you have discovered through the experimentation. **Give a recommendation for the set of parameters that seem to work best when using your algorithm**. Best may be defined as converging quickly and reliably. You will have to decide if it is more important to converge quickly or to converge all of the time. Clearly state what you have chosen as your criteria **best**.

Grading:

- All programs will be graded on functionality and ease of use/understanding. I am not grading for (a) efficiency or (b) beautiful code, but I expect a certain level of proficiency. I primarily about the output from a perspective of understanding. If I look at your code to help me debug an error, I should be able to navigate and understand what you have done (e.g. there should be some comments and modularity). I am looking for output that clearly shows me information related to (a) your input/starting state, (b) the parameters used (c) that you have a good solution and (c) how much effort it took to get there. I will also expect a clearly written README file that explains how to compile/run your code.
- The written part will be graded based on the quality of discussion/analysis. I am looking for evidence that you have experimented as directed and have considered the impact of the various changes you made (or will make).

Handing In Your Program:

1) Submit your source code and answers to the questions using the cse webhandin. Your answers for part II should be in Word or .pdf format.

http://cse.unl.edu/~handin/.