CS/CSYS 352 Homework 1 F'18
Assigned: Tuesday 9/4
Part 1 (done in pairs): due Tuesday 9/11
Part 2 (done solo): due Thursday 9/20

In general, for all homeworks in this class:
Softcopy zip files due in BB by 4:00 pm of the due date.
Hardcopy of code and write-ups due any time before 4:30 pm on the due date, either in class or slipped under the door of my 217 Farrell office.

PROBLEM:
You are to solve the N-queens problem (a generalization of the classic 8-queens problem) using the Matlab GA toolbox, where potential solutions are represented using permutations. To learn more about the N-queens problem and to see how the number of solutions (and number of unique solutions) varies as a function of N, check out the link here, among other places. Our goal is actually not so much to solve the N-queens problem (although you should verify that the solutions your GA reports as correct solutions are, indeed, correct solutions!) as it is to do some simple experimentation to assess the effectives of variation operators and see how well the problem scales with N.

In Part 1, you will work in pairs to develop and test low-level functions you will need for the evolutionary process, including creation of the initial population, mutation, recombination, and fitness evaluation. In Part 2, you will integrate these low-level functions with the GA toolbox and do some evolutionary experiments.

Part 1 (5 pts): Initial Development/Validation of low-level functions (to be done as pair-programming):

a) First, develop and debug individual functions for the following (THIS IS THE TIME TO GET HELP FROM ME ON PROGRAMMING AND DEBUGGING IN MATLAB IF YOU'RE HAVING TROUBLE). Again, note that you are NOT actually using the GA in part 1, you're simply writing some low-level functions that will be plugged into the GA in Part 2. Write and test the following 5 functions:

   i. A function for population creation. You are to pass in the population size P and the number of queens N and return a P × N matrix, where each row is a permutation of the numbers 1:N; **hint:** see **randperm.** Test your function to verify that it works.

   ii. Two functions that implement two different types of permutation mutation operators (pick any two from sec. 3.4.4 (1st edition) or 4.5.1 (2nd edition); which do you think might work better?); pass in a population matrix and a parent vector of length m indices (each of which must be in the range 1 to the number of rows in the population that is passed in); the parent vector specifies which m rows to mutate, and returns an m × N matrix of mutant children); **hint:** there is no P_m in this case.

   NOTE: when you're doing "bottom-up" testing of an individual function like this, you should use small example data structures for inputs that are designed to easily verify if the function is working corrections, rather than larger structures that are more realistic but harder to verify. For example, an appropriate example test call for a function named mutate.m could be:

   >> mutate(1:10,[1 1 1 1 1])

   In this sample call, the first argument is a "population" that is just a single individual "permutation" [1 2 3 4 5 6 7 8 9 10], and the parent vector specifies that you're going to make 5 mutant copies of this same individual. Make sure you verify that all children created by this crossover operator are valid permutations! Note that using this parent makes it easy to check correctness.

iii. Implement the cut and crossfill crossover (see pseudo-code in Fig. 2.3 in 1st edition or Fig. 3.3 in 2nd edition). **hint:** The GA toolbox only returns 1 offspring (not 2, as shown in this pseudo-code) when recombining each pair of parents, so when you implement the cut and crossfill crossover you only need to make and return one child, not both.

Pass in a population matrix and a row vector of 2c parent indeces; these indeces specify which pairs of adjacent individuals in the population matrix to cross to make c children. (This is how the GA toolbox will send in the parent vector, so please follow these specifications.) E.g., if the parent indeces are [5 2 6 3 2 4], this would specify to make 3 kids, one by crossing parents 5 and 2, a second by crossing parents 6 and 3, and a third by crossing parents 2 and 4.

The function should return the c × N matrix of the c children that were created by crossing the 2c parents).

An appropriate example test call might be:

>> CutAndCrossfill([1:15; 15:-1:1],[1 2 2 1 1 2 2 1])

The population passed in contains two potential solutions, [1 2 3 4 5 6 7 8 9 10 11 12 13 14 15] and [15 14 13 12 11 10 9 8 7 6 5 4 3 2 1], and you'll return 4 children from crossing these two parents multiple times. Make sure you check that all children created by this crossover operator are valid permuations! Note that using these two parents makes it easy to check correctness.

iv. A function to evaluate the fitness of the population (pass in a population and return a column vector of the corresponding fitness values).

An appropriate example text call might be:

>>fitness([2 4 1 3; 3 1 4 2; 1 2 3 4; 4 3 2 1; 1 3 2 4; 2 1 4 3]);

In this example, you're evaluating 5 potential solutions to a small 4-queens problem, two that are known solutions with 0 conflicts, two that have the maximum number of conflicts (6), and two that are in between with 2 conflicts and 4 conflicts, respectively. When testing a function, always try to include both "boundary" and "intermediate" cases like this, if possible.

b) Write a script with all your test calls, using short examples as shown above to verify that all of your functions work as expected. When you're satisfied that everything is working, make a transcript of a test call to your script by using the **diary** command.

c) Prior to actually starting Part 2, think about which mutation function that you implemented might work better on this problem, or do you think they'll work equally well? Justify your answer (I don't care if you're right or not, as long as you make a logical argument).

Hand In: your code, your diary transcript of the sample calls showing that your functions work and your answer to part c. Upload all of this to bb in a single zip file and give me stapled hardcopy in class. Please have ONE member of the pair upload a zip file to your dropbox in blackboard. In the comment headers to each function, please include the names of both partners, a short description of what the function does and what is expected for the input and output parameters, and a sample call.

Part 2 (10 pts): Do some experiments using the GA toolbox (must be done solo)

a) First, modify your functions from part 1 so that they will work with the GA toolbox – this primarily means adjusting your headers to match the number and order of parameters expected by the GA toolbox. I suggest you look at some of the predefined GA toolbox functions for the corresponding operators to see the order of the parameters in the header. E.g.,

>>open gacreationuniform % note that the popsize must be passed in through the options structure
>>open mutationuniform % note that you will NOT need to pass in a mutationRate
>>open crossoversinglepoint % note that pairs of adjacent parents create a single child
>>open rastriginsfcn % example fitness function – note that it works on an entire population matrix, not just one individual.

Some of these parameters you may not actually need to use in your functions, but you still need to keep them in the header as placeholders (this is the price of using a standardized toolbox). When you make test calls to your functions after revising the headers, you can just send in empty vectors to any unused input arguments.

b) Try running the GA on your problem by specifying any necessary options to meet the above specifications, including function handles (using the @ operator) to your low-level functions for the initialization and variation operators using gaoptimset, and then calling ga.m.

Use the following specifications:

- Use tournament selection with a tournament size of 4.
- Do not use any elitism.
- Terminate when you find a complete solution by setting the 'FitnessLimit' to zero, and also specify 'MaxGens' to make sure you do eventually terminate on large problems where you may not find a correct solution.
- Set the crossover fraction to 0.2 (usually, GAs use a higher crossover rate, but cut and crossfill isn't really a great recombination operator for this problem and also we want to be sure to see the effects of the different mutation operators we're comparing.

c) Once you've got the basic integration with the GA toolbox done, perform the following experiments:

i. **Setting the Population Size:** Use N = 16, either one of your two mutation functions, and MaxGens = 100. Using these values, try various population sizes, working from smaller to larger (you should initially do this in fairly large jumps until you narrow in on an appropriate range), until you find a population size that enables you to reliably solve the 16-queens problem for some number of reps (e.g., 20 times out of 20).

Note: To get nrep random (but repeatable) initial populations, you can simply specify rnep unique seeds for the RNG. E.g.,

```
for seed=1:nreps
   rng(seed); % set the seed used by randperm and rand
   % do the run…
end
```

Make a plot showing how the success rate (proportion of successful runs) varies as a function of population size (once you know the appropriate range of population sizes, you should choose a range of pop sizes that give you a fairly smooth plot). Indicate what you think a good population size is that will reliably solve the 16-queens problem.

ii. **Compare your two different mutation functions:** Design and implement an experiment that compares the effectiveness (think about what metric could demonstrate this, perhaps experimenting with various options based on your results) of your two mutation functions, using the popsize you determined in the previous step. Make one or more plots that shows your results.

iii. **Do a write-up** (in Word or Latex) that incorporates all the figures you made and discusses your findings, including your experimental designs (your work should be repeatable based on what you write in the text, without my having to look at the code), whether your predictions from part 1 held up regarding which mutation operator would be better, and discussing why or why not you think this occurred. Make sure you refer to all figures in the text using figure numbers. Please try to write in a clear scientific writing style; concise, well-thought-out write-ups are more valued than long-winded write-ups without clarity. Use complete sentences with correct grammar and spelling.

Plotting Hints:

a. **boxplot** is a nice way to compare distributions
b. **errorbar** is a good way to do line plots that show both means and standard deviations
c. "hold on" will allow you to keep adding things to the same plot.
d. IN GENERAL, when making plots, please make sure they are fully labeled (**xlabel, ylabel, title, legend** if necessary, etc.).
e. Always make sure that **font sizes are large enough to be legible** when included in your write-up. Very small font sizes are a pet peeve of mine, so please do your grade a favor and make them legible! You can set fontsize as follows. E.g.,

```
plot(x,y); % plot some data
set(gca, 'fontsize',14); % set the desired font size on the current axis
xlabel('X-axis meaning'); % subsequent labels like this are at new font size
```

HAND IN:

1) Upload all m-files and a pdf of your write-up into BB, all in one zip file.
2) Hand in stapled hardcopy of your write-up and code. Only include Matlab functions or scripts *you* wrote or modified; include comment headers on your files and please insert comments to clarify particularly confusing pieces of code (the easier it is for me to understand your code the, more friendly I'll feel when assigning a grade!).

XC homeworks: In general, I encourage students to do further exploration of topics for extra credit. XC homeworks are not intended to replace assigned homeworks, exam, or the team project, but can help to boost your course grade, in case you don't get as high a grade on some of these as you would like. All XC homework is due by 10/18, both in hardcopy and zipped together as a single softcopy file uploaded to BB. I will award XC credit proportional to the amount and quality of the work performed, but XC will be capped at a maximum of 10% total for a given student toward their course grade.

Below, I am suggesting a couple of appropriate XC projects related to this homework that you may wish to challenge yourself with, but you may also propose alternative XC projects that interest you.

XC: Do a scaling study on the N-queens problem: Using whichever mutation function you found worked better, see if you can design and implement an experiment that assess how well your algorithm scales with N (e.g., on average, how many fitness evaluations are required to solve problems, as a function of N? Note that both the population size and the number of generations will impact the number of fitness evaluations). Do a write-up that clearly describes your experiment, includes one or more plots to demonstrate your results, and briefly discusses your results.

XC: Implement a better recombination operator for the N-queens problem: Cut and crossfill is not really a great recombination operator for this problem, I just had you use it because it's one of the easiest to implement and had good pseudo-code in the text. However, see section 3.5.4 (1st edition) or 4.5.2 (2nd edition) for some alternative permutation recombination operators; implement a crossover operator you think may work better and justify why you think it would be better. Do some experimentation to see what the best crossover rate is for this new operator, then do an experiment to see if it outperforms cut and crossfill. Do a write-up that clearly describes your experiment, includes one or more plots to demonstrate your results, and briefly discusses your results.