EECS 649: PROBLEM SET 4

Reading:

- R&N 4.1, including box on page 118; 4.2 as needed for review; 6.1-4

Total Points: 100

Format:

- Use standard sheets of paper (8.5 by 11 inches) before scanning
- Perform all work neatly. When asked to write a paragraph, it should be typed (e.g., using a computer and LaTeX or Word).

---

*The answers to the following two problems should be **typed**:*

Problem 4.1 [10 points]
[Nilsson] Specify fitness functions for use in evolving agents that

a. control an elevator,
b. control stop lights on a city main street.

Problem 4.2 [10 points]
Give a precise formulation for the following as a constraint satisfaction problem (CSP):

> Class scheduling: There is a fixed number of professors and classrooms, a list of classes to be offered, and a list of possible time slots for classes. Each professor has a set of classes that he or she can teach.

In formulating your answer, consider the definition of CSPs at the start of Section 6.1 and specify $\mathcal{X}, \mathcal{D}, \mathcal{C}$.

*This problem requires hand calculation and need **not** be typeset:*

Problem 4.3 [20 points]
Solve the cryptarithmetic problem in Figure 6.2 of R&N by hand, using the strategy of backtracking with forward checking and the MRV and least-constraining-value heuristics.

Record the steps and the reasoning/method behind them.
(You solve the problem once, using forward checking, making use of the two stated heuristics to pick variables and values, and recording what you used when.)

Does min-conflicts make sense for this type of problem?

The following two problems require programming:

Problem 4.4 [30 points]
Repeat the one-dimensional optimization experiment from the eHandout of Local Search lecture. **(See the bottom page 2 of the Local Search Handout for that day.)** Specifically, maximize

```
F(x) = 4 + 2x  + 2 sin(20x) - 4 x^2
```

on the interval [0, 1] using fitness-proportional selection (aka roulette selection) of individuals (points of the form 0.01*k, k= 0, ..., 100) and simple mutation (x-epsilon, with probability 0.3; copy with probability 0.4; x + epsilon, with probability 0.3). You may use epsilon=0.01.

Use at least N=10 individuals in your population. Report N and comment on your experiments and results. Turn in documented **code**.

Repeat the above, but add a crossover operator that is a convex combination of two individuals, x and y: a x + (1-a)y, for 0 <= a <= 1.

Use roulette selection instead of the "if fitness(x)>r" selection in my example. Specifically, choose an individual to be "reproduced" proportional to its fitness. So, if in the current generation, you had x1, …, x10, the total population fitness would be TF = sum_i F(xi). Then these would be chosen for "reproduction" in the next generation with probabilities (F(x1)/TF, F(x2)/TF, …, F(x10)/TF).

After mutation, individuals should be clipped to remain within the interval [0,1].

CROSSOVER IS LIKE THIS [you may enforce both parents to be different below if you wish]:
For each individual I want to produce in the next generation:
I pick two parents, each using a different "spin" of roulette selection:
And combine them using crossover, picking a random a

Problem 4.5 [30 points]
Consider the **8-queens Problem** from R&N. Here, you will solve 8-queens using
- Random-restart hill-climbing (RRHC; cf. Section 4.1.1 of R&N)

You will be **minimizing** "fitness," defined as the number of **non-attacking** pairs of queens (as on the bottom of p. 117 of R&N), which is 28 minus "the number of pairs of queens that are attacking each other, either directly or indirectly" (see note in the middle of p. 112 of R&N regarding intervening pieces). Thus, in this problem, when you find a configuration/state with fitness = 28, you have found a solution.

a. Implement the RRHC above. Write your algorithm so that it exits as soon as a solution is found, prints the solution (as a string of digits like those depicted in Figure 4.7 of

R&N), and prints the total number of **fitness evaluations** required from the start of the algorithm.

b. Test your routines enough to convince yourself that they work and that solutions are being found appropriately. **(Do not report on this.)**

c. You will gather statistics regarding the operation of your algorithms by running each algorithm 100 times (from different random starting positions/populations) and report only the **average** of the number of evaluations until a solution is found. **(You may wish to add a cut-off number of iterations that are not exceeded; in this case, don't include failed searches in your average.)**

Turn in your **code** and a **summary** of your results.

Note:
I don't expect you to program this from scratch. This problem is **much, much easier** if you look at or use the code that is already available:

- I have placed some (non-optimized, no-guarantees) C++ code that I used for a more extensive exploration of local search algorithms for the 8-queens problem at 8queens.ipynb. **It also includes a translation to Python of the main routines.**
- R&N's On-Line Code Repository contains code for the n-queens problem written in a variety of languages. See the linked github repository, which includes Python, Java, ...

Further Notes:

- PS5 will also use the 8-queens problem in an effort to amortize your time and effort
- Also, there is a linked EXTRA CREDIT OPPORTUNITY involving the 8-queens problem