Introduction to Artificial Intelligence
Program 2 – Genetic Algorithms
Fall 2021

You're preparing for a back-country camping trip. Your campsite is very remote, so you want to be prepared and have everything with you that you will need.

BUT—your vehicle will only hold so much. There just isn't room for everything you might possibly want; you're going to have to make some choices. So you begin by making a list of anything you could think of that you might possibly want—you don't want to forget something simple. But how to decide what to take?

You have assigned each item a *utility*, a degree of usefulness, as a floating-point number in the range 0-10. An item rated 0 or 1 is something you could easily do without if necessary; something rated 9 or 10 is vital. And of course, each item weighs something—a real number > 0, in pounds. Your vehicle will carry 500 pounds of cargo. How can you pack the most useful items (highest utility) into the available space?

(We will assume that weight, not volume, is the limiting factor; no matter what combination of items you select, they will fit in the vehicle, as long as the total weight is less than 500 pounds. This is an example of the *knapsack problem*; finding an exact solution is NP-complete, meaning it takes exponential time; even if we could instantly guess the best combination, confirming that it really was the best solution would also take exponential time. A brute-force solution requires n! time, and a dynamic-programming solution requires $O(n^2 2^n)$ time. Still exponential, still not practical for any but small N. So we'll settle for a 'pretty good' solution we can find in a practical time frame.)

You are given an input text file containing 400 sets of (utility, weight) pairs. Your task is to use a genetic algorithm to find a good selection of items to pack while staying within your weight guidelines. Your program will be tested against a different data file with the same format.

Programming details:

- You may write your program in Python, C, C++, C#, or Java.
- The simplest way to represent a selection is as a bitstring or something that maps to one, with '0' indicating the item is not being selected and a '1' indicating that it is. This can be an array or vector of booleans, of characters, bytes, a string, or even literal packed bits, in which case you will need only 50 bytes to represent a selection (but be prepared for lots of low-level bit manipulation).
- Use an initial population of at least 1000 random selections. Have each selection be of about 1/20 or so of the total items (otherwise everything's over weight, and with no variation in fitness at all, the algorithm has trouble getting started.)
- Use a mutation rate of 0.0001; that is, each item in each selection has, independently, a 1/10,000 chance of being changed during the mutation step.
- Use L2 normalization to convert fitness scores to a probability distribution.
- In computing fitness scores, indicate something is unacceptable because it exceeds the 500-pound limit by assigning it a fitness of 1. Yes, it will still have *some* chance of being selected; that's OK. Once fitness starts climbing, the chances of these too-heavy collections being chosen will become insignificant.
- Record the average fitness for each generation; save this in an output file. Continue iterating until the average fitness improves less than 1% across 10 generations.

- Report the highest fitness selection found: What items are taken, and what the total utility is. Put this into a text file.

Prepare a short report—a page or two—describing your program. Include a discussion of what data structures you chose, and why; any places you were able to parallelize or optimize your program; and an overview of what running the program was like—what was the overall efficiency like, how many generations were needed, etc. No references are expected.

Submit your report along with your source code and output file to Canvas. If you are using GitHub or other online repository, submit your GitHub link.

Some sample results from my implementation in C++. This was running 5000 generations; in practice, improvement was very slow after about 2500 generations (3500 for the starting population of 10,000).

Starting population: 500
Max fitness after 5000 generations: 705.1
Average fitness: 663.613

Starting population: 1000
Max fitness after 5000 generations: 703.7
Average fitness: 672.718

Starting population: 2000
Max fitness after 5000 generations: 729.1
Average fitness: 696.434

Starting population: 5000
Max fitness after 5000 generations: 729.8
Average fitness: 700.214

Starting population: 10,000
Max fitness after 5000 generations: 738.2
Average fitness: 706.14