Project 1: Simple Genetic Algorithm

Due: Feb 22nd, 2022

Total: 100 points.

Invocation:
> sga [-h] [-g] [-G] [filename]
with filename being the name of your settings file. If this filename is left off, by default it should use a settings file called "gasettings.dat". All of the arguments are optional

-h : Should output a help message describing what the options do and then terminate..

-g : Limited debugging information should be displayed while running.

-G : Full debugging should be displayed while running.

The settings file will be described in detail later

Task 1: Coding the GA:

In this task you are coding a simple genetic algorithm. In particular, you are coding a GA that will have the following characteristics:

1. Binary representation (over strings of size n) with a population of size N
2. Random initialization of our population
3. Tournament selection (with configurable size tournaments) to determine parents
4. Uniform Crossover with configurable probability to apply
5. Bit flip mutation (with probability of 1/n)
6. Elitism replacement

Future considerations:
We will be expanding on this project as we go through the semester. Obviously every project will involve different fitness functions and even using fitness functions in the past, so make sure that through the settings file a particular fitness function can be selected. You should also make sure that adding new crossover and mutation operators, different replacement methods and so forth are relatively easy. In order to save you much work, it is a good idea to put as much code as you can in functions. Have a function that takes in two individuals and produces two offspring for a particular recombination and then call that function to do recombination. Have a function that takes in a population and selects one parent using some selection operator. The more you make it modular the easier the follow-up projects will be.

WARNING: You must code this project yourself. If I find that your code is significantly similar as other students or code found online, you will receive a 0 and be turned in to the department.

Settings file:
Our project will be using a settings file of a specific format. All parameter settings that could be changed should be stored in this file. Each individual setting should be on a different line, with a description of the setting preceding the actual value. So your file will consist of two columns, one with a description of the setting (no spaces) and the other with the value of that setting.

Each time your project is launched, it should read the settings file if it is provided, or use the default settings file if not. The following is an example of this settings file, but keep in mind that adding additional settings could be desired. For example:

```
randSeed 123
populationSizeN 100
stringSizen 50
probApplyCrossover 0.6
probApplyMutation 1.0
selectionMethod 0
tournamentSizek 2
fitnessFunction 0
```

Overall procedure:

Our GA will be executing the following loop:

```
BEGIN
  INITIALIZE population with random candidate solutions;
  EVALUATE each candidate;
  REPEAT UNTIL ( TERMINATION CONDITION is satisfied ) DO
    REPEAT UNTIL (ENOUGH OFFSPRING is satisfied) DO
      1 SELECT pair of parents;
      2 RECOMBINE the pair of parents;
      3 MUTATE the resulting offspring;
      4 EVALUATE new candidates;
      5 Add these candidates to offspring population
    OD
    5 REPLACE some individuals in the parent population with offspring;
  OD
END
```

Initialization, population and representation:

We are using binary representation, so in our population our individual solutions will be strings of 0s and 1s. I suggest storing the individual solutions in a data structure that uses an array of either characters or integers storing 0 or 1 to store the string itself. It might be useful later on to be able to add other attributes to individuals, so having a structure with an array as one element of it for the strings is a good idea (for example, a field to store it's fitness and a field to store if it has been evaluated yet). While you could use bit fields instead of an array of characters or integers, this tends to complicate the code considerably and is often slower (though more storage efficient).

You should use some sort of data structure to store the entire population of strings. Keep in mind that you will be generating a population of child solutions also that will "replace" the current generation at some point.

At the start of the GA, you should initialize the initial population of candidate solutions by filling the strings uniformly random with 1s and 0s. The average of fitness of the population at this point (using onemax) should be about n/2.

The fitness function (Our Problem):
For our first problem to solve, we are using the stereotypical problem you would first try on a GA, which is called "onemax". The fitness of a string in onemax is simply the sum of the bits that are set to 1.

For example:

The string of "001000" has a fitness of 1. The string of "101010" has a fitness of 3. The global optimum string is the string of all 1's.

While this is a trivial problem (at least for small string sizes), it is a good testing bed for seeing if we are messing something up, as our GA should be able to easily start with a random population and then find the string of all 1's, at least for reasonable sized strings and population. These "reasonable" sizes we will discuss in more detail later.

Note that you should avoid computing fitness on strings you have already calculated, so your fitness function should check to see if a string has been evaluated and if so simply return that value.

Selection:
In this project we are using k-tournament selection. In order to do this, we will select k individuals at random from the population. The most fit individual out of that is selected as parent1. We then select at random another k individuals from the population. The most fit individual out of that is selected as parent2. Note that you do not need to store a list of k individuals, but instead just pick k times and store which individual is best so far. You can select an individual again, so no worries on duplicates and both parent1 and parent2 could be the same strings in theory.

Recombination:
Our recombination operator is *uniform crossover* that will take in two parents and produce two new children to put in the offspring population. This results in mixing the bits uniformly from the two parent strings to produce two new children. Note that this operation is applied with a percentage chance from the settings file. If it is determined not to apply recombination, the two children are identical to the parents. Ensure that offspring data structures are set so their fitness will be calculated later if they are changed from the parents.

Mutation:
Our mutation operator is bit-flip mutation with the probability of a bit-flip of 1/n. Note that on average this will result in one bit flipped in a string but could flip more or less. This also has a percentage chance to be applied depending on the settings file. Again, if individuals are mutated in any way, they need to be flagged to be evaluated by the fitness function later.

Replacement:
We are using full replacement with elitism of one individual. The offspring population will fully replace the parent population, except for the best individual in the parent population. That will replace the lowest individual in the offspring population. This is done to ensure that we do not discard the best individual due to chance.

Termination criteria:
Each generation you should check to see if you have found the global fitness. If you have, terminate the procedure and output the global best string found as well as its location in your population. It should also output the average fitness and worst fitness of the population.

We also need a failsafe, to discover if we have failed to find the solution and are unlikely to do so in the future. For this, your algorithm should keep track of the fitness of the last three generations. If the best fit string has not improved in three generations and the average fitness has not improved, then terminate with a message showing the best fit string found so far, the worst fit string found so far and a brief summary of the three generations data that you have.

What the output should look like:
The output shown should vary depending on whether -g or -G is set. If no flags are set, at the end of each generation, output the best, avg and worst fitness for that generation. For example, something like:

Generation 1 : (B: 38, A: 25, W: 23)

If -g is set, you should output the above information, as well as the best and worst strings (duplicates do not matter). You should also output the entire population at this time.

If -G is set, this means essentially EVERYTHING. For this, I want all of the above information, as well as details showing the operators at work. For example, showing the individual parents selected, then the result after recombination. Mutation would also show individuals before and after mutation.

TASK 2: Using the GA

Experiments:
I want you to experiment with the GA to find what settings are good enough for solving onemax given various sizes of solution strings. In particular, I want you to write up what the population size has to be (given our settings) to consistently find the optimal string for string sizes of 20, 30, 40 and 50. Then I want you to estimate what size you think the population would have to be for a string of size 100. Then for a string of size 1000000. If you feel like you need more data points, feel free to run it for 60, 70 and so on. Depending on how efficiently you code it, it could be possible to run for strings of size 1000000, but the trend should be fairly easy to see.

Submission:
I expect the following:

1) A readme.txt file describing how to compile and run your project. This file should also outline briefly the results of your experiments above.

2) A word document or pdf (called results.pdf or appropriate suffix) describing your experiments and estimations of the growing population size as the problem gets harder.

3) All your source files

To submit, put all these files in a folder and compress them using tar or zip. Then attach the archive to canvas.