Finding Array Coverings using Evolutionary Algorithms
BioComputing, CSE 598
Assignment 1, Spring Semester 2019
Due: Feb. 11, 2024 (11:59 pm)

1 Introduction

In this assignment we will use evolutionary computation (EC) to find covering arrays, which can be used for many purposes, including software testing. In software, a program may have many possible configurations, any of which could be faulty. One possible approach to this problem is to test exhaustively all possible configurations to ensure that the software cannot get into a faulty state. However, this space increases combinatorially with the number of configurations / inputs and becomes very time-consuming for software of any reasonable complexity. So, the goal is to devise a method that eliminates redundant tests and provides good coverage of the configurations at a reasonable cost. Here, "good coverage" depends on the size of the input configurations, which dramatically increases the problem difficulty as the configuration size increases. Covering arrays are an abstraction that is used to represent the different configurations such that for any choice of program components of that given size, all possible ways of testing those components are realized in at least one test. Your assignment is to use an evolutionary algorithm to find covering arrays for different system and configuration sizes.

You are free to write your own EC algorithm or to use a public domain package. However, if you choose to use a pre-existing package you are responsible for understanding how it works, getting it to work with your problem, and analyzing results. The DEAP package, written in Python, is a popular package, although it is designed for multi-objective optimization, which is not what this assignment is about, so if you use DEAP make sure that you run it in single-objective mode and that you understand how it is implementing the basic components of an evolutionary algorithm, as it has extra complexities for handling multi-objective optimization. It is available here: `https://github.com/DEAP/deap`.

Late Policy: You are allowed three "free" late days during the semester. Once you have used up those late days, 10% of your grade will be deducted for each day that it is late.

2 Covering Arrays

Coverage quantifies how well a set of sample configurations (referred to as *covers*) captures the set of all configurations. Configurations are represented as a two-dimensional array, where each row represents a configuration and each column represents a variable that defines part of the configuration. For example, if a program can be run either in debug mode or not, then one column would represent the `debug?` switch and another column might represent what operating system the software is running on. We characterize configurations in terms of four parameters: n, k, v, and t. n is the number of rows; k tells us how many variables, or columns, there are in the array; and v specifies how many values each variable can have (we will assume that v is constant for all the variables). In general, it is not feasible to test all possible configurations, so we use the parameter t to define intermediate levels of coverage (in this assignment, we will use $t = 2$ except possibly for the extra credit. An array is *t-covering* iff for all choices of t columns from the k possible columns, each of the possible v^t t-tuples appear in those columns in at least one row of the array.

So, for example, if $v = 2$, $t = 2$ and $k = 5$, we would have a 2-coverage of the configurations if all possible pairwise interactions between the k variables are tested. This may or may not require testing using all rows in the array. Here is an example (that happens to have the minimum number of rows possible for $v = 2, t = 2, k = 5$):

0	0	1	0	0
0	1	0	1	1
1	0	0	1	0
1	1	1	0	1
0	1	0	0	0
0	0	1	1	1

3 Assignment

There are three aspects to this assignment: Implementing the fitness function; implementing an EC algorithm to find good coverings; and analyzing your results. In Part 1, we will start with simple versions of these aspects, and then progress to more complex versions in Part 2.

The input files for each part of the assignment are available on the class Canvas site. The data will be organized in csv format.

Part 1: Finding good covers

Given a set of configurations and with parameter values n, k, v, and t as inputs, use your EC algorithm to find the smallest t-cover that you can. A natural representation for this problem is bit string with n positions, where each position in the string corresonds to one row of the configuration array. At each position, a 1 indicates that the corresponding row is tested, and if it is set to 0, it is not tested. For this problem, you can use traditional mutation and single- or two-point crossover, but for the size of problem we will be considering, single-point should be fine.

The fitness function is simply how few rows your evolutionary algorithm can find that generate a t-covering of the input array.

Part 2: Generating good covers

Given a set of configurations with parameter values k, v and t specified, generate a set of tests (each test represented as a row in an array) that satisfies the t-coverage property. This will require reworking your representation, so each individual is an entire two-dimensional array. One way to accomplish this is by concatenating all the rows into one long string. For example, if you have a 4 rows, each with 5 variables, then your individual will have 20 genes.

It is unlikely that the standard crossover operator will work well on this problem, at least for more challenging instances, although you are welcome to try it. Instead, you should consider implementing a version of the crossover operator that respects row boundaries, i.e., where the only allowed crossover points are between blocks corresponding to each configuration.

Once you have this up and running for the input, t, try rerunning your experiments with successively smaller values of t. That is, if we give you a problem with $t = 5$ and your GA can solve it, then try rerunning with $t = 4$ to see how small you can make t and still find a solution.

The datasets for Part II are available in Canvas.

Extra Credit: Adding weights

Here are three extra credit options:

1. Redesign your crossover algorithm, so that instead of exchanging data between rows, it exchanges data between columns. Rerun your algorithm on the more challenging examples and compare results for the two different versions of crossover.

2. Have a different number of values per-column, since not every component has the same number of values. If you attempt this option, design your own arrays to test your algorithm, and inform your instructor so we can supply some test arrays for you to report on in your report.

3. If your EC is performing well on all of the assigned datasets, we can supply more challenging n, k, v values, and if you beat the known best answer, you might have a publishable paper.

Datasets for Extra Credit will be made available on request.

4 Details

4.1 Reporting results

Please hand in a 5-7 page report using the ACM format, written in appropriate academic style with proper citations. Please print a copy of your assignment paper and bring it to class on Feb. 13 and submit a .pdf of the writeup and a .zip file with your code, instructions for running it through Canvas, and .csv files with the best output covering arrays you found for each part of the assignment (one .csv file for each part or subpart, with unambiguous names). Your Canvas submission time will document when you completed the assignment. If you miss class on Feb. 13, then bring a printed copy the next time you come to class. Your paper should be 5-7 pages and describe the following:

1. The problem you are trying to solve (in your own words);

2. How you set out to solve the problem, e.g., which GA software you used, its basic algorithms (what kind of selection, crossover, mutation, etc.)[1];

3. The representation (how did you represent covering arrays to the EC);

4. The fitness function(s) you tried;

5. The basic parameters for your runs (population size, generation time, crossover rate, mutation rate, etc);

6. Inputs and outputs;

7. Experimental design (what experiments did you run and why);

[1] You can assume that your reader is familiar with EC if not the exact version of EC and algorithmic decisions that you made.

8. Experimental results, including results about performance on each dataset (e.g., the lowest t value you found in Part I and its covering; the best covering you found for each dataset in Part II (and lowest t value), etc.) and about population dynamics; Specifically, your report should include:

   (a) How well did your EC perform? Report in table format the results you obtained, e.g., the lowest k that your EC found, on each input example and discuss your results. Did crossover contribute to your EC's performance? For each dataset, report at least one covering that your EC found (not in a table).[2]

   (b) Evolution curves: Include plots that show generation vs. fitness for sample runs in both parts of the assignment and discuss.

   (c) Statistical analysis: How much does the performance of the GA vary from run to run? Document how you aggregate performance across many runs.

   (d) In-depth analysis of one successful run for each part: Study how the GA solved the problem, then report out several key mutations or recombinations that it used along the way, which improved performance.

[2]For the larger examples, you can print out the solutions you found as arrays in a supplement (appendix) which does not count against your page limit.