LOG6305 - Techniques avancées de test du logiciel

Assignment 3

Search based test generation

Département de génie informatique et de génie logiciel
École Polytechnique de Montréal

Hiver 2024

1 Introduction

The input space of a software system can be vast and multidimensional, comprising various types of inputs. Search-based techniques allow to navigating this complex space to discover inputs that uncover faults or edge cases that might be missed by manual testing. One of the commonly used search techniques is evolutionary search and genetic algorithms in particular. In this assignment you are going to generate test inputs for a program using a genetic algorithm.

2 Objectives

The objectives of this work are: :

1. Understand the main concepts of search based test generation.
2. Learn to use one of the available libraries for evolutionary search.
3. Be able to configure a search-based algorithm to a given problem.

3 Search-based software testing

A good overview of the search techniques used for software engineering tasks is provided in the following article [1]. In this assignment we will mostly focus on genetic algorithms (GA), as one of the commonly used techniques for test generation [2]. Fraser and Arcuri provide a comprehensive description on how GA can be applied to test generation [3]. In the following section we provide a general overview on genetic algorithms.

3.1 Genetic algorithms

Genetic algorithms (GA) are a class of evolutionary algorithms that use a set of principles from genetics, such as selection, crossover, and mutation, to evolve a population of candidate solutions [4]. A typical genetic algorithm pipeline is shown in Fig. 1. The basic idea is to start with a set of individuals (candidate solutions) representing the initial population, usually generated randomly from the allowable range of values. Each individual is encoded in a dedicated form, such as a bitstring, and is called a chromosome. A chromosome is composed of genes. Each individual is evaluated and assigned a fitness value. Some of the individuals are selected for mating. The search is continued until a stopping criterion is satisfied or the number of iterations exceeds a specified limit.

Three genetic operators are used to evolve the solutions: selection of survivors and parents, crossover, and mutation. The operators in evolutionary algorithms serve different functions. Mutation and crossover operators promote diversity in the population, allowing for the exploration of the solution space. On the other hand, survivor and parent selection operators promote the quality of the solutions, enabling the exploitation of the search space. It is essential to select the appropriate combination of operators that offers a good balance between exploration and exploitation to achieve optimal results.

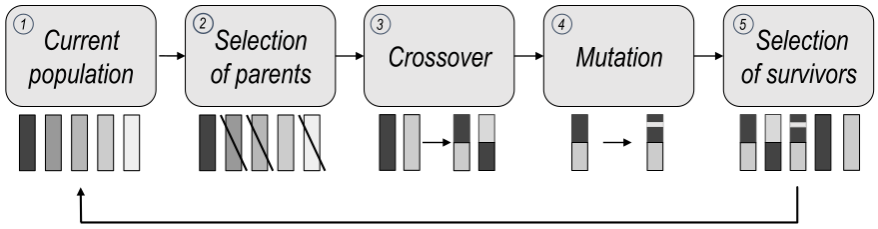


Figure 1: Genetic algorithm pipeline

Parents Selection. Selection of parents is an operator that gives solutions with higher fitness a higher probability of contributing to one or more children in the succeeding generation. The intuition is to give better individuals more opportunities to produce offsprings. One of the commonly used selection operators is tournament selection: a small subset of individuals is chosen at random, and then the best n individuals in this set are selected for the mating. The selection pressure can be adjusted by controlling the size of the subset used. Another commonly used technique is a proportional-based selection, also known as a roulette wheel selection, where the probability of choosing an individual depends directly on its fitness.

Crossover operator. The crossover operator is used to exchange characteristics of candidate solutions among themselves. In our approach, we are using a one-point crossover, illustrated in Fig. 2. We can see how the parent chromosomes are exchanged genes t_i to produce the offsprings. Other types of crossover include multi point crossover and uniform crossover, however, they are only applicable to fixed-size individuals.

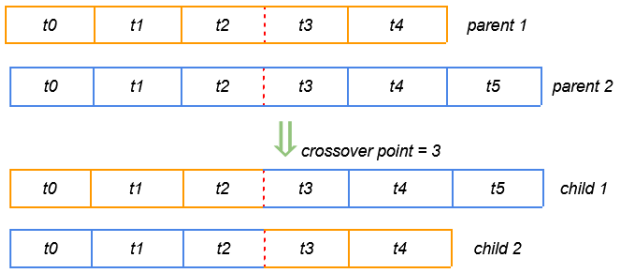


Figure 2: One point crossover operator

Mutation operator. The mutation operator has been introduced to prevent convergence to local optima; it randomly modifies an individual's genome (e.g., by flipping some of its bits, if the genome is represented by a bitstring).

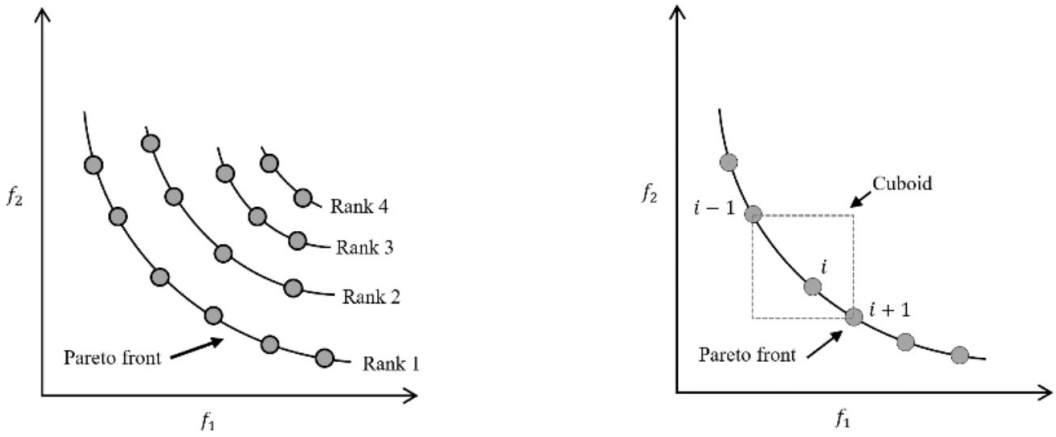
Crossover and mutation are performed with probability *pcross* and *pmut* respectively, where $pmut < pcross$. The rates at which mutation and crossover are applied are an implementation decision.

Survivors selection. In GA the population size is almost always constant. This requires a choice to be made about which individuals will be allowed into the next generation. This decision is often based on their fitness values, favoring those with higher quality, although the concept of age is also frequently used. In contrast to parent selection, which is typically stochastic, survivor selection is often deterministic. For genetic algorithms $(\mu + \lambda)$ strat-

egy is commonly used. In this strategy, the set of offspring and parents are merged and ranked according to (estimated) fitness, then the top μ individuals are kept to form the next generation.

The choice of optimal representation and operators for a genetic algorithm is problem dependent and is a complex area of research.

Genetic algorithms are effective in solving both single and multi-objective optimization problems. However, they are particularly suited to multi-objective problems since they can simultaneously handle a set of possible solutions, allowing the algorithm to find multiple members of the Pareto optimal set in a single run. Under Pareto optimality, a solution is better than another if it is superior in at least one of the individual fitness functions and not worse in any of the others. This is an alternative to simply aggregating fitness using a weighted sum of the n fitness functions. When using Pareto optimality to search for solutions, the result is a set of non-dominated solutions. Each solution in the non-dominated set is no worse than any other solution in the set, but it cannot be said to be better either. In a multi-objective genetic algorithm, the quality of a solution is determined by its non-dominance ranking, rather than by a single fitness value. An example of non-dominated sorted individuals is shown in Fig.3a (lower rank is better) [5].



(a) Non-dominated ranking

(b) Crowding distance calculation

Figure 3: Non-dominated sorting procedure and Crowding distance calculation

One is the most popular multi-objective genetic algorithms is the non-dominated sorting algorithm-II (NSGA-II) [6]. It builds a population of competing individuals, ranks and sorts each individual according to a non-dominated level, applies evolutionary search operators to create new pool of offsprings, and then combines the parents and offspring before partitioning the new combined pool into fronts. The NSGA-II then assigns a crowding distance to each member. The crowding distance value of a particular individual is the average distance to its two neighboring individuals in the Pareto front. The crowded distance of the ith solution is the average side-length of the cuboid, as shown in Fig.3b. It uses the crowding distance in its selection operator to keep a diverse front by making sure each member stays a crowding distance apart. This keeps the population diverse and helps the algorithm to explore the fitness landscape.

3.2 Whole test suite generation

The idea of a whole test suite generation is to use an evolutionary technique in which, instead of evolving each test case individually, all the test cases in a test suite are evolved at the same time, and the fitness function considers all the testing goals simultaneously [3]. The technique starts with an initial population of randomly generated test suites, and then uses a Genetic Algorithm to optimize toward satisfying a chosen coverage criterion, while using the test suite size as a secondary objective. At the end, the best resulting test suite is minimized. To facilitate the implementation we are going to use a popular evolutionary search framework Pymoo [7].

4 The tasks

In this assignment you task will be to generate a test suite for two systems under test which are *urlparse* module from urllib.parse and $HTMLParser().feed$ module from html.parser python libraries.

Required set-up

Operational system : Linux/macOS/Windows, **Python** : version 3.10. Installed Pymoo library, version==0.6.1. Obtain a local copy of the code for the first practical work, located in the github repository: `https://github.com/log6305/HIV_2024_TP3`. We suggest to create the fork of this repository and make changes to your fork (later you could send a link to your fork for evaluation).

This repository contains a baseline implementation of the test suite generation and you need to develop an improved version of it tailored to each of the systems under test. Complete the example in 'Optimize.py' the to better understand how to use the provided code.

Your tasks:

1. Question 1: Testing *urlparse*. Create a class $UrlTestSuiteGenerator$ (child of Abstract generator class) to generate randomized inputs ($generate_random_test$ method) for the *urlparse* module. It is recommended to use the grammar you defined in the previous assignment (3 points). Create a $UrlTestSuiteProblem$ class (child of AbstractProblem) to define the test evaluation metrics. You should consider two objective functions: number of lines covered (maximize) and the number of tests used (minimize). One way to combine them is to evaluate the ratio n_l/n_t, where n_l corresponds to the number of lines covered and n_t to the number of tests found. (3 points) Implement $UrlTestSuiteMutation$ and $UrlTestSuiteCrossover$ classes that are children of $AbstractMutation$ and $AbstractCrossover$ respectively (4 points). You goal is to find the least number of test inputs that cover the biggest number of lines possible. Your evaluation budget is 5000 evaluations. The final test suite should contain no more than 40 test scenarios. Use the GA algorithm [1]from the list of algorithms. Over at minimum 5 runs compare the performance of the GA algorithm when TournamentSelection and

[1] `https://pymoo.org/algorithm`

RandomSelection are used (2 points). To do this, plot the best value of the best n_l/n_t metric found at each iteration (evaluation) of the algorithm. In the same way compare the performance of GA with with TournamentSelection and the RandomSearch ($pymoo.algorithms.soo.nonconvex.random_search$) (2 points). Present your final code in 'url_optimize.py' file.

2. Question 2: Testing $HTMLParser().feed$. Perform analogical steps as in the previous question. In the names of the classes replace "Url" by "HTML". In total 14 points (the same as for Q1) can be obtained. Present your final code in 'html_optimize.py' file.

3. Question 3: Select the best best configuration (the one achieving the highest line coverage and producing the lowest number of test cases) for each of the test suite generators you obtained. Report the best maximal ratio n_l/n_t achieved based on 5 runs. Respect the indicated budget as well as the number (no more than 5000 evaluations). You will be given additional points based on the average performance of your test generator the two modules under test. Your additional points will be proportional to the ranking. The maximum additional points you can achieve equal to 5% of the assignment grade.

Important instructions:

1. If possible, do not modify the implementation of the Abstract classes. When implementing your class you can add any number of additional attibutes or functions.

2. For questions 1-2 describe the test generator you implemented (how the population is generated, what fitness function you use and what mutation and crossover operators) and discuss the evaluation results. For presenting results, use plots and tables (you should add the convergence plots i.e. best ratio n_l/n_t found vs the number of evaluations).

5 Expected deliverables

The following deliverables are expected:

- Link to your repository with the implemented fuzzers or a .zip archive of the repository.
- The report for this practical work in the PDF format.

You should submit all the files to Moodle. When adding the PDF file, do not put into a zip. Additionally, you should add a ".txt" file with the following content: "Your full name, your student id" (this might be used for the grading automation). This assignment is individual.

The report file must contain the title and number of the laboratory, your name and student id.

6 Important information

1. Check the course Moodle site for the file submission deadline
2. A delay of [0.24 hours] will be penalized by 10%, [24 hours, 48 hours] by 20% and more than 48 hours by 50%.
3. No plagiarism is tolerated (including ChatGPT). You should only submit code created by you.

References

[1] Mark Harman, Phil McMinn, Jerffeson Teixeira De Souza, and Shin Yoo. Search based software engineering: Techniques, taxonomy, tutorial. In *LASER Summer School on Software Engineering*, pages 1–59. Springer, 2008.

[2] Stephan Lukasczyk and Gordon Fraser. Pynguin: Automated unit test generation for python. In *Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: Companion Proceedings*, pages 168–172, 2022.

[3] Gordon Fraser and Andrea Arcuri. Whole test suite generation. *IEEE Transactions on Software Engineering*, 39(2):276–291, 2012.

[4] Thomas Bäck, David B Fogel, and Zbigniew Michalewicz. Handbook of evolutionary computation. *Release*, 97(1):B1, 1997.

[5] Shanu Verma, Millie Pant, and Vaclav Snasel. A comprehensive review on nsga-ii for multi-objective combinatorial optimization problems. *Ieee Access*, 9:57757–57791, 2021.

[6] Kalyanmoy Deb, Amrit Pratap, Sameer Agarwal, and TAMT Meyarivan. A fast and elitist multiobjective genetic algorithm: Nsga-ii. *IEEE transactions on evolutionary computation*, 6(2):182–197, 2002.

[7] J. Blank and K. Deb. pymoo: Multi-objective optimization in python. *IEEE Access*, 8:89497–89509, 2020.