Too Much Automation? The Bellwether Effect and Its Implications for Transfer Learning

Rahul Krishna   Anwesha Das
Computer Science, North Carolina State University, USA
{rkrish11, adas4}@ncsu.edu

1. MOTIVATION

Automated Patch generation has garnered a lot of attention in the past decade. Specifically, tools like GenProg have seen tremendous success. In fact, GenProg is now used as a baseline to compare other new techniques. Albeit popular, there is a general lack of agreement on the need for complex tools like GenProg to generate effective patches. While some researchers [?] opine that GenProg is quite effective in generating patches, given the existence of test cases, others [?, ?, ?] disagree. They say that it is quite unclear if the the efficacy of GenProg is merely due to its "mutate" operator or due to genetic programming.

This issue was examined by Qi et at. [?]. With their tool RSRepair (*R*andom-*S*earch *Repair*), they demonstrated the benefit of random search over GenProg.

It is my hypothesis that the lack of efficiency of GenProg in this area is not due to the failure of Genetic Algorithms (or any evolutionary computation scheme for that matter), but due to incorrect and perhaps even incomplete modeling of the search space.

As a motivating example consider Genetic Programming. The algorithm works by trying to maximize the "fitness" score. In this case, the number of passing test cases given a patch. RSRepair, on the other hand, considers either repair effectiveness (as measured by the number patch trials required) or repair efficiency (measured by the number of test case executions). It not hard to see that each of these objectives namely, the number of passing test cases, the repair effectiveness, and the repair efficiency, are all important. Perhaps some are more important than others, but they all play a crucial role nonetheless.

Therefore, I conjecture that the research in the area of automated patch generation will be best guided by modeling the domain as a multiobjective search problem with the aforementioned objectives. This, in my opinion, will allow us to leverage the wealth of research in Search Based Software Engineering (referred to henceforth as *SBSE*). To this end, this project asks the following two research questions:

RQ1: Can automated patch generation be modeled as a multiobjective search problem?

Although it makes theoretical sense to view patch generation as containing multiple objectives, this question asks if it is practical to model it as such.

RQ2: If yes, what is the best multiobjective search to generate patches automatically?

If the answer to RQ1 is affirmative. This question aims to explore various multiobjective optimizers to identify the best algorithm. Specifically, this project will compare three standard SBSE techniques: NSGAII, MOEAD, Genetic Programming, using Random Search as a baseline. In addition to this, the usefulness of a promising new algorithm called SWAY[1] [?, ?] will be also be investigated.

2. AN OVERVIEW OF PATCH GENERATION

Generally, when a bug is reported, an automated program repair scheme undertakes the following three steps:

1. *Fault localization:* These techniques are used to identify suspicious faulty code snippet that may a have caused the bug.

2. *Patch Generation:* Once faulty code snippet has been located, many candidate patches can be generated through the modifications to that code snippet, according to specific repair rules based on either evolutionary computation or code-based contracts.

3. *Patch Validation:* When a candidate patch has been produced, regression testing, inclusive of negative test cases (reproducing the fault) and positive test cases (characterizing the normal behaviors), is commonly used to validate the correctness of produced candidate patch.

The above procedure can be iterated over and over again until some valid patch is found. Any patch passing all these test cases is considered valid.

3. A (VERY) BRIEF OVERVIEW OF MULTIOBJECTIVE SEARCH

A search is the task of finding one or more solutions which satisfy one or more specified objectives. While a single-objective optimization involves a single objective function and a single solutions, a multiobjective optimization considers several objectives simultaneously. In reference to the current problem we have three competing objectives, all of which need to be maximized: (1) Number of

---

[1]This algorithm is in current development by our research group

Measure	Description
Runtime	Time taken for the algorithm to be generate optimal solutions.
Speed Up	Ratio of time taken for the parallelized version of the algorithm over the time taken for the serial version.
Convergence	Convergence represents the accuracy of the obtained solutions. It is the distance between the obtained solutions and ideal Pareto frontier.
Diversity	Diversity represents the spread of the proposed solutions. Ideally the solutions should be well distributed across the Pareto frontier,rather than concentrated in certain regions.

Figure 1: Performance Measures

passing test cases; (2) The repair effectiveness; and (3) The repair efficiency

In such a case, a multiobjective optimizer generates a set of alternate solution with certain trade-offs. These are called *Pareto Optimal Solutions*. The solutions, in our case, will be a set of valid patches that perform equally well when measured in terms of the three objectives.

Multiobjective problems are usually complex, NP-Hard, and resource intensive. Although exact methods can be used, they consume prohibitively large amounts of time and memory. An alternative approach would be to make use of metaheuristic algorithms from RQ2. These approximate the Pareto frontier in a reasonable amount of time.

4. EVALUATION

4.1 Metrics

Figure 1 highlights some of the preliminary performance evaluation measures we use to evaluate the algorithms. Please note that this will be expanded in future to accommodate other measures.

4.1.1 How to calculate them?

Calculating these metrics are well studied [?, ?]. We have an implementation in place that can readily give these metrics, given a set of ideal solutions (in this case, valid patches).

4.2 Datasets

The data set required for evaluating will be obtained from some recent work on GenProg [?]. My reading of the literature suggests that these dataset are the "benchmarks", used by several researchers [?]. These are C programs with all the bugs reported reported in the historical versions as tabulated in Figure **??**.

4.3 Tools

As for tools, it is worth noting that AST can be generated from C programs using a tool called CIL[2].

4.4 Analysis Plan

To provide a more comprehensive comparison, the optimizers will be ranked ranked statistically. To do this, I shall use the Scott-Knott procedure recommended by Mittas & Angelis [?].

[2]https://sourceforge.net/projects/cil/

This method sorts a list of l treatments with ls measurements by their median score. It then splits l into sub-lists m, n in order to maximize the expected value of differences in the observed performances before and after divisions. E.g. for lists l, m, n of size ls, ms, ns where $l = m \cup n$:

$$E(\Delta) = \frac{ms}{ls}abs(m.\mu - l.\mu)^2 + \frac{ns}{ls}abs(n.\mu - l.\mu)^2$$

Scott-Knott then applies some statistical hypothesis test H to check if m, n are significantly different. If so, Scott-Knott then recurses on each division.

5. PROPOSED TIMELINE

Figure **??** highlights the proposed timeline for the tasks undertaken in the project. Please note again that this subject to change, please see https://github.com/ai-se/spatch/milestones for the latest milestone progress.

6. REFERENCES